What's in an Exploit? An Empirical Analysis of Reflected Server XSS Exploitation Techniques

Ahmet Salih Buyukkayhan
Microsoft

Can Gemicioglu
Northeastern University

Tobias Lauinger
New York University

Alina Oprea
Northeastern University

William Robertson
Northeastern University

Engin Kirda
Northeastern University

Abstract

Cross-Site Scripting (XSS) is one of the most prevalent vulnerabilities on the Web. While exploitation techniques are publicly documented, to date there is no study of how frequently each technique is used in the wild. In this paper, we conduct a longitudinal study of 134 k reflected server XSS exploits submitted to XSSED and OPENBUGBOUNTY, two vulnerability databases collectively spanning a time period of nearly ten years. We use a combination of static and dynamic analysis techniques to identify the portion of each archived server response that contains the exploit, execute it in a sandboxed analysis environment, and detect the exploitation techniques used. We categorize the exploits based on the exploitation techniques used and generate common exploit patterns. We find that most exploits are relatively simple, but there is a moderate trend of increased sophistication over time. For example, as automated XSS defenses evolve, direct code execution with `<script>` is declining in favour of indirect execution triggered by event handlers in conjunction with other tags, such as `<svg onload`. We release our annotated data, enabling researchers to create diverse exploit samples for model training or system evaluation.

1 Introduction

Vulnerability reward programmes have emerged as a way of encouraging and coordinating the discovery and disclosure of vulnerabilities by independent researchers [7, 18, 21]. As a popular example, the commercial platform HackerOne [9] partners with software vendors and administers bug bounty programmes on their behalf. Non-commercial platforms such as XSSED [6] and OPENBUGBOUNTY [16], which specialise in Web vulnerabilities, are *one-sided* [18] in the sense that they allow submissions of vulnerabilities affecting any website instead of restricting them to partner organisations.

Bug bounty programmes and platforms have been the subject of a large number of studies, which to date have nearly exclusively focused on the economics of vulnerability discovery such as incentives and return on investment [7, 21], as well as the user population, affected websites, patching delays, and high-level categorisation of vulnerability types [18, 21]. However, little is known in terms of technical details about the exploits submitted by users.

In this paper, we perform a longitudinal study of the exploitation techniques used by the authors of reflected server XSS exploits in XSSED and OPENBUGBOUNTY over a period of nearly ten years. This required us to solve a number of technical challenges. The exploits submitted to the two platforms are not available in isolation, but must be extracted from the archived request and server response. This is particularly difficult, given that we have no insight into the inner workings of the web application, and no access to a comparable, "clean" server response without the exploit. To this end, we design techniques to isolate the reflected exploit by comparing the request and response data, and accounting for server transformations such as encoding or escaping.

Once the attack string is isolated, we need to extract features for exploitation techniques. Some techniques, such as syntax tricks attempting to confuse filters in web apps and firewalls by exploiting a parsing disparity, are not visible when a page is opened in a browser. We detect this class of techniques statically and comprehensively validate our method.

Other features are visible only at runtime, such as call stacks, or whether the injected code actually works. In fact, many archived server responses contain multiple instances of reflected code, but input sanitisation appears to be applied inconsistently, causing some instances to execute while others do not. We collect this information by dynamically executing archived exploits. Often, exploits have environmental dependencies or require certain events to be triggered before they can execute, thus we developed a sandboxed environment with lightweight browser instrumentation that can simulate different network conditions and user interaction.

For our analysis, we integrate the data collected by the static and dynamic analysis methods. When the results of both static and dynamic analysis are available, we detect the exploitation techniques with 100 % true positive rate. Our methodology enables us to perform a comprehensive longitud-

inal study of 134 k reflected server XSS exploits. Surprisingly, a large percentage of exploits, 65.0 % in XSSED and 11.7 % in OPENBUGBOUNTY, use no technique beyond inserting a `<script>` tag after possibly closing the previous tag, and make no attempt to bypass filters.

We also analyse common exploit patterns, defined as a collection of techniques used in combination. We find that the most common pattern accounting for about half of all exploits in XSSED is closing the previous tag and inserting a `<script>` tag with the payload. Presumably due to increased filtering of this tag, in the more recent submissions to OPENBUGBOUNTY, we observe a trend of replacing the `<script>` tag with `<img>` or `<svg>` tags and indirect code execution using the `onload` and `onerror` event handlers.

Our analysis sheds light on XSS exploitation trends and popular exploit patterns. We also discuss how exploitation techniques evolve over time, as automated defenses are implemented by modern browsers. To summarise, this paper makes the following contributions:

- We demonstrate that it is possible to accurately identify proof-of-concept exploits in archived web pages. We implement a system to execute exploits and extract features with a unified static and dynamic approach.
- We analyse ten years of reflected server XSS exploits to quantify the use of various exploitation techniques.
- We show that there is only a moderate change in techniques and sophistication despite the long time span, supporting the hypothesis of an "endless" supply of low-complexity vulnerabilities on the Web, and suggesting a lack of incentives for users of the platforms to find and contribute more sophisticated XSS vulnerabilities.
- We release the dataset [1] used in this paper.

2 Background

Cross-Site Scripting (XSS) is a class of attacks that consist in injecting attacker-controlled JavaScript code into vulnerable websites. The attack targets the visitors of a compromised website. To the victim's browser, the injected code appears to originate from the compromised website, thus the Same Origin Policy does not apply and the browser interprets the code in the context of the website. Consequently, attackers can exfiltrate sensitive information, or impersonate the victim and initiate transactions such as purchases or money transfers.

This study focuses on reflected server XSS in which the vulnerability arises from server-side templating and the inability of the browser to distinguish the trusted template from the untrusted user input. To prevent attacks, the developer must escape sensitive characters in the user input. However, which characters are sensitive and how they can be escaped depends on the sink context. For example, inside a JavaScript string context, quotes must be escaped as `\"`, inside HTML tags as the HTML entity `&quot;` and outside of HTML tags, they do not have a special meaning. General-purpose server-side programming languages typically do not escape user input since they are unaware of the sink context. Furthermore, in some cases developers may wish to allow certain types of markup in user input, such as style-related tags in articles or blog posts submitted by users. Unfortunately, developers often omit input sanitisation entirely, use an incorrect type of escaping, or implement custom sanitisation code that is not sufficient to block the attacks. For example, developers who would like to allow certain tags but not script may remove the string `"script"` from user input, but they may fail to account for case differences (`<ScRiPt>`) or indirect execution through event handlers (`onload="alert(1)"`). Web Application Firewalls are often implemented using regular expressions and may be vulnerable to similar issues [4, 13].

Cross-Site Scripting exploits bugs in websites as opposed to browsers. As such, XSS is not to be confused with JavaScript-based attacks that exploit browser bugs to cause memory corruption and gain code execution at the operating system level. However, XSS could be used as a vehicle to inject such types of payloads into websites. In this work, we do not study such malicious payloads; rather, we focus on the tricks that authors of XSS exploits use to bypass filters up to the point where they gain script execution capabilities. The two datasets that we use, XSSED and OPENBUGBOUNTY, contain only benign proof-of-concept (PoC) payloads such as `alert(1)`, but exploits do differ in how they achieve code execution.

2.1 Vulnerability Rewards and Platforms

Bug bounty platforms provide a formalised way for bug hunters and website operators to interact. Furthermore, they often provide cash rewards for verified bug reports. Economic aspects of vulnerability rewards have been studied extensively, such as for Chrome and Firefox by Finifter et al. [7], or for the HACKERONE and WOOYUN platforms by Zhao et al. [21]. The arguably most well-known platform, HACKERONE, operates vulnerability reward programmes on behalf of other organisations. The platform is "closed" in the sense that vulnerabilities can be submitted only for websites or software developed by participating organisations. Furthermore, those organisations are typically responsible for verifying vulnerability reports, determining the amount of a potential reward, and deciding on whether to publish the vulnerability.

In contrast to closed commercial platforms, our research uses data obtained from two open, non-profit bug bounty websites specialised in XSS. In this model, users may submit exploits targeting any website. The first of the two platforms, XSSED [6], was founded in 2007 and appears to be largely dormant since 2012. The second platform, OPENBUGBOUNTY [16] was launched in 2014 under the name XSSposed, and is still active at the time of writing.

2.2 Related Work

Prior studies on vulnerability reward programmes have focussed on economic considerations (e.g., [7, 18, 21]). Zhao et al. [21] dedicated a small part of their study to the submitted exploits, but these encompassed many different vulnerability classes, and the analysis was restricted to a few high-level metrics such as "severity" provided by the WOOYUN platform. Furthermore, the authors noted that "Wooyun may ignore vulnerabilities that are considered irrelevant or of very low importance, such as many reflected XSS vulnerabilities." In their analysis of OPENBUGBOUNTY, Ruohonen and Allodi [18] considered platform metrics such as patching delays and user productivity. Instead of these general-purpose exploit or platform metrics, we conduct a deeper analysis of the different techniques used specifically in reflected server XSS.

Cross-Site Scripting has been considered in a large body of scholarly research, mostly from a vulnerability detection and exploitation prevention perspective. However, we are aware of only very few works that quantify the occurrence of different XSS exploitation techniques. For example, while evaluating their proposed XSS filter, Bates et al. [3] classified the sink contexts of 145 random exploits from XSSED. Our findings in Section 4.2 are in line with theirs, and provide additional detail about unnecessary context escaping, and multiple or non-executing reflections in server responses. Furthermore, we work with archived server responses, whereas Bates et al. could only consider live pages that were still vulnerable.

In DOM-based client XSS, several works [14, 15, 20] reported characteristics of exploitable data flows, such as the source and sink types, and cognitive complexity. In our context of reflected server XSS, data flows and transformations occur in (hidden) server-side code, thus we need different methods. Furthermore, the goal of the analysis was to characterise the "root causes" of vulnerabilities. In contrast, we use human-curated data sets to quantify different exploit techniques.

Closest to our work is a study by Scholte et al. [19] of 2,632 XSS and SQL injection attacks found in the National Vulnerability Database. The authors measured the complexity of XSS exploits using five static features such as having encoded characters or event handlers. Scholte et al. found that vulnerabilities were simple, and their complexity did not increase. Our results confirm the trends reported in 2011. We extend them in depth, scale, and time span by using a fifty times larger data set, extracting five times more static features, adding dynamic analysis, and characterising exploit authors.

Similar in spirit, but different in the covered environment, are two studies analysing Java exploits [12] and malware [5].

3 Methodology

In this work, we quantify the techniques used by the authors of XSS exploits archived in XSSED and OPENBUGBOUNTY. This required us to solve many challenges such as locating attack strings embedded in HTML responses, triggering and executing the attack in a controlled environment, and identifying XSS techniques used. We address them by combining static and dynamic analysis methods, as outlined in Figure 1.

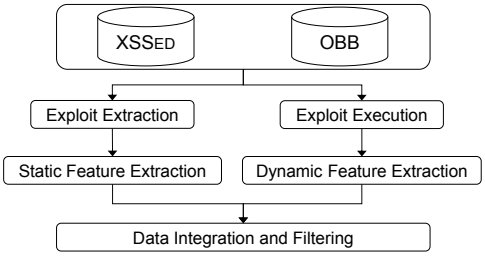


Figure 1: Overview of our static and dynamic analysis system.

Database	Date Range	Authors	Submissions	Websites
XSSED	2007-01-24 – 2015-03-12	2,579	43,939	33,747
OBB	2014-06-18 – 2017-04-11	980	119,946	86,339

Table 1: Raw XSS data collected (before filtering).

3.1 Exploitation Techniques

Reflected server XSS exploits can use a range of techniques to bypass incomplete input sanitisation and achieve code execution. Various cheatsheets [2, 8, 10, 11] collect these techniques. Our work models portable techniques in the sense that they work in most modern browsers rather than in specific browsers or versions. We also exclude techniques that we implemented based on the cheatsheets, but that appear rarely in our datasets. On the other hand, we include additional techniques that we observed while working with our datasets. Table 4 on page 10 lists all exploitation techniques considered in this paper, grouped into several categories.

3.2 Data Collection

Our analysis of XSS exploits aims to cover a broad range of techniques and study how they evolve over time. To do so, we extracted all XSS exploits publicly disclosed on XSSED and OPENBUGBOUNTY until April 2017, as shown in Table 1. This represents the latest available dataset, as XSSED is now largely dormant, and OPENBUGBOUNTY has implemented anti-crawl measures. The operators of the databases claim that all entries were confirmed to work at the time of submission.

3.3 Exploit String Extraction

XSSED and OPENBUGBOUNTY archive XSS vulnerabilities by storing the request data (URL, cookies, POST data) and the HTML page returned by the server. Therefore, first we must isolate the attack string embedded in request and reponse. Prior work found that nearly half of sampled XSSED entries

had been fixed or the affected website was not reachable [3]. Instead, we extract XSS exploits by looking for reflected strings that exist both in the request and the response.

Since nearly all exploits only show a message using the methods `alert()`, `prompt()` or `confirm()`, these method names typically appear in the URL. In case of obfuscation, the URL contains other strings, such as `eval()` or related method names. Thus, for the limited purpose of locating these payloads, we search the request and response for these keywords.

To extract attack strings, we implemented a greedy heuristic that expands the matching range from the keyword to the left and right as long as the character sequences match. Servers can transform the inputs before reflecting them, such as encoding characters, truncating or expanding the string. Without knowledge of the server-side logic, we cannot solve this issue in a generally sound way. However, we do support the most common transformations, specifically HTML entity, Unicode, URL, and certain cases of double URL encoding, and our matching is case-insensitive. Additionally, we allow up to two consecutive non-matching characters, provided that there are more matching characters afterwards. Our validation shows that this works well in the context of our dataset.

One server response may contain multiple matches when one injected parameter is reflected multiple times, or when exploits are injected into multiple parts of the URL. We initially match all reflection combinations, but eventually select a single pair that has been confirmed to be functional, and is unlikely to be a false positive (Sections 3.5 and 3.7).

3.4 Static Feature Extraction

Once the attack string has been isolated, we need to detect how it achieves JavaScript code execution. Especially syntax-based tricks such as whitespace or comment insertion, or non-standard syntax are not visible from the JavaScript runtime, as the browser parser converts the non-standard textual HTML into a canonical DOM representation. Given that our feature extraction operates on a previously known dataset, we implement it using regular expressions and iteratively refine and validate our implementation through manual labelling.

The techniques we detect statically (marked ◐ and ● in Table 4, on page 10) include escaping from the injection sink context by ending string literals or closing a tag, case tricks such as `<ScRiPt>`, and using a slash instead of a space, as in `<svg/onload`. Additionally, we extract features such as event handlers so that we can combine them with dynamically detected features to reduce false positives. Depending on the technique, we make multiple attempts to match the respective regular expression, such as before and after decoding encoded sequences, or removing whitespace.

	Submissions		Messages TP / FP	
	XSSED	OBB	XSSED	OBB
No Trigger	38,009	94,693	38,926 / 188	95,779 / 286
Mouse Move	289	3,959	288 / 5	3,994 / 15
Mouse Click	145	103	123 / 23	97 / 7
Network Error	0	4	0 / 0	4 / 0

Table 2: Exploits executed by triggering events.

3.5 Exploit Execution

Almost two thirds of archived server responses in XSSED and OPENBUGBOUNTY contain multiple reflections of attack strings. Some of them appear to be sanitised or truncated, preventing execution. In order to extract XSS techniques only from attack strings that execute, we need to run the exploits.

To execute exploits, we load the archived response pages in Chrome and Firefox. We add instrumentation to detect whether the exploit is working, and sandbox network traffic by serving all requests from an isolated local proxy. As for the static exploit string extraction, we leverage the fact that nearly all exploits contain a simple payload showing a message with `alert()`, `prompt()` or `confirm()`, and use the calling of any such method as a sign for a successfully executed exploit.

When simply opened in our sandbox environment, many pages do not result in the exploit being executed. In some cases, exploits require user interaction. In other cases, exploits only execute when external resources such as images or scripts are loaded successfully, or when resources fail to load. Unfortunately, neither XSSED nor OPENBUGBOUNTY archive these resources, thus we replace them with generic versions. Our system attempts to execute each exploit in up to four rounds, as summarised in Table 2. In the first round, the proxy returns the archived page and then answers each resource request with an empty response and the HTTP 200 OK status code. For exploits not successfully executed, in the second round we move the mouse over all elements in order to trigger user interaction event handlers, and wait for ten seconds to allow delayed execution using `setTimeout()`. This results in 4.2 % additional executions in OPENBUGBOUNTY. In the third round, we click on all elements, and in the fourth round, the proxy answers resource requests with empty documents and the HTTP 404 Not Found status code to trigger the `onerror` event.

During our attempts, we may accidentally trigger unrelated code in the page. When clicking a button, for instance, the page may display an error message. To distinguish page-related dialogues from alerts caused by injected exploit code, we semi-automatically classify the displayed messages. Messages displayed by exploits are often very similar, such as "XSSED" or "OBB," or the name of the author. We manually looked through the top 30 message texts in each database, which account for 70.3 % of messages in XSSED, and 99.7 %

in OPENBUGBOUNTY. We confirmed that 58 of the 60 messages were related to exploits. For the remaining messages, we built a dictionary of attack-related keywords, such as the names of exploit authors commonly appearing in our data. When a message text contains none of these exploit keywords, and is not part of the 58 most frequent messages labelled as exploit-related, we assume it is a false positive and discard it.

For validation, we picked two random samples of 100 unique messages that were considered exploit-related and not in the top 30. In XSSED, all sampled messages were confirmed based on the source code to originate from an injected exploit. In OPENBUGBOUNTY, our sample had a 2 % false positive rate. Yet, as OPENBUGBOUNTY contains only 324 distinct messages with attack keywords beyond the top 30, we estimate a total of 7 distinct false positive messages, which corresponds to 0.007 % of all message instances.

During our experiments, we found minimal differences between Chrome and Firefox. Around 0.3 % of submissions in XSSED, and 1.0 % in OPENBUGBOUNTY worked only in Firefox. On the other hand, 2.4 % of XSSED, and 3.9 % of OPENBUGBOUNTY worked only in Chrome. This disparity is mainly due to technical differences in our instrumentation that may prevent data extraction in corner cases such as pages crashing the browser. Browser families also differ in how they handle (partially) broken pages, and in which exploitation techniques they "support." Chrome executes more exploits than Firefox, and returns richer metadata that we require to combine static and dynamic features. When using Firefox, the lack of metadata allows combining features for only 23 % of OPENBUGBOUNTY submissions, as opposed to 83 % with Chrome. For simplicity, we only report Chrome results.

3.6 Dynamic Feature Extraction

While executing exploits, our instrumentation detects the use of certain exploitation techniques. For some, detection is only practical at runtime, such as exploit code interacting with page code, or assignment of the `alert` method to a variable and subsequent calling of the variable. We also extract a number of features in parallel to the static feature detection to reduce false positives. This includes the use of obfuscation or quote avoidance methods such as `eval()` or regular expressions.

From an implementation point of view, our browser extension injects a content script that intercepts calls to a range of methods and property accesses. It logs the values and types of parameters such as evaluated strings or message texts, and records the stack trace of the call. We extract line numbers and offsets to distinguish features detected in exploits from detections in unrelated page functionality (Section 3.7).

Due to our choice of a light-weight browser instrumentation, we do not detect certain corner cases. These include exploit code in `data:` or `javascript:` URLs or an `<iframe>`, use of `innerHTML` or `outerHTML`, and element creation using the DOM API `document.createElement()`. Some of these limitations are due to incorrect line number and offset values returned by the browser, which prevents us from combining these dynamically detected features with their static detection counterpart. Among all features, only page code interaction (F6) is exclusively based on dynamic data. In a random sample of 25 out of 145 exploits where this feature was detected, we did not observe any false positives.

3.7 Data Integration and Filtering

We combine the data collected during the static and dynamic analysis to address the shortcomings of each individual approach. By using only exploits that were detected both statically and dynamically, we exclude exploit reflections that do not execute, or other false positives of static detection. By restricting dynamic detections to source code ranges statically detected as the attack string, we exclude false positives of dynamic detection due to code unrelated to the exploit.

The static extraction returns byte ranges of the reflected data. From the dynamic execution, we obtain stack traces of method invocations with line number and offset values. At a high level, integrating static and dynamic data means checking whether the dynamic line number and offset are within the static exploit range. Our implementation adds three preparatory steps. First, we extract the character encoding used by the browser so that we can correctly compare offsets involving multi-byte characters. Second, we normalise endline characters in the server responses. Third, in several special cases, the offsets returned by Chrome do not point to the actual beginning of the respective statement but are shifted by the length of a syntactical construct or point to the end of the tag, and need to be adjusted accordingly.

Almost two thirds of server responses reflect the injected exploit multiple times. When matching static and dynamic data, we combine all pairs and select a single exploit to represent each HTML page. We choose as the representative exploit one where the message is a known exploit text, and which is the shortest string among the matches with the most frequent set of exploit features detected on the page. Only 1.6 % of submissions had more than one working reflection and more than one unique feature set, thus in the vast majority of cases, our selection does not make any difference.

Similarly to the exploits, we also combine features by restricting them to cases where both the static and the dynamic occurrence was observed. These are marked as ● in Table 4. We validated a subset of important features, shown in Table 3, by labelling 25 random exploits from three detection categories: both static and dynamic data were combined ($S \cap D$), only static features ($S - D$), or only dynamic features ($D - S$) were detected. Features with only static detection are either false positives of static detection, or false negatives of dynamic detection, and vice versa for features with only dynamic detection. The results show that our conservative approach resulted in no false positives in feature detections. Detections made by

S: static, D: dynamic	*S − D*			*D − S*			*S ∩ D*		
(% of all detections)	%	TN	FN	%	TN	FN	%	TP	FP
I1: document.write	8.5	10	1	0.8	1	0	90.7	25	0
I2, I3: eval, setIv/To	11.5	24	1	1.8	9	0	86.7	25	0
O1: char. code to str.	1.2	25	0	0.0	0	1	98.8	25	0
O2: regular expr.	0.1	21	4	0.3	24	1	99.6	25	0

True/false negatives/positives in samples of 25 detections (if available).

Table 3: Validation of Static and Dynamic feature extraction.

only one type of analysis sometimes contain false negatives, but the combination of static and dynamic analysis results in perfect accuracy.

3.8 Validation

We successfully combine static and dynamic data for 36,229 XSS submissions in XSSED (82 %), and 97,677 XSS submissions in OPENBUGBOUNTY (83 %). In the remaining 17.1 % of submissions, we cannot match a statically extracted attack string in the server response with a dynamically observed attack message, thus we exclude these exploit submissions from further analysis. To further investigate, we manually labelled a random sample of 1,000 such cases. The most frequent reason for exclusion was the inability to execute the exploit (37.7 % of cases, or 6.4 % of the entire dataset). These cases were apparently caused by the use of nonstandard syntax not supported by Chrome, missing dependencies or event triggers, or the exploit not executing before our 10 second timeout. Exploits that fingerprint the environment and stop execution to prevent analysis would also fall into this category, but we did not observe any example in our data. Another 22.1 % of excluded submissions in the two databases contained incomplete data, or an unsuitable format such as screenshot submissions that we could not process. Around 0.3 % of excluded submissions were false negatives due to line number and offset matching issues. The remaining 39.9 % of excluded submissions (6.8 % of the entire dataset) were attack types that our analysis framework does not support. We do not further analyse exploits injecting references to remote JavaScript files (2.9 % of the entire dataset), as it is difficult to distinguish benign from malicious script URLs, and Chrome returns line numbers that are ambiguous when multiple such remote scripts are injected into the same page. Similar line number issues exist with exploits contained in `data:` or `javascript:` URIs (found in 0.4 % of all submissions). Around 0.2 % of the exploits redirect to a different page, and 0.9 % inject text into the HTML page instead of showing a message, causing their exclusion. Furthermore, we exclude exploits that are not reflected server XSS, as they imply the absence of the exploit from the URL (stored server XSS: 0.05 % of all submissions) or from the server response (DOM-based client XSS: 1.9 %), thus preventing us from isolating the attack string. Lastly, we exclude 0.09 % of the dataset because these submissions appear to involve techniques unrelated to HTML and JavaScript, such as SQL injection.

3.9 Limitations

Our datasets contain potential biases due to their open nature. The data may be biased towards simpler exploits, favour the use of identical exploits on multiple websites, and result in fewer submissions per website while covering several orders of magnitude more websites than closed platforms. Furthermore, we may under-report exploits for sites that operate their own (cash-based) bug bounty programme, as users may be motivated to submit their findings directly to that programme instead of reporting them to XSSED or OPENBUGBOUNTY. Because only 10.1 % and 13.3 % of websites in XSSED and OPENBUGBOUNTY, respectively, have multiple submitted exploits, our dataset should not be used to derive findings about the security posture of individual websites. There is no guarantee that users find vulnerabilities on a given website, that they submit them to one of the two databases covered in our study, and that the submitted exploit is as simple as the vulnerability allows. These limitations notwithstanding, we believe that our dataset (134k exploits submitted by 3k authors) gives us a unique insight into the XSS techniques used by a wide range of security testers.

Our method has some limitations. The detection being based on keywords, we might fail to locate exploits when none of these keywords is used in the exploit. Since we expect that every database entry contains an exploit, we can estimate the false negative rate through manual validation of entries without a match (Section 3.8). Our approach does not support "split" exploits with multiple cooperating parts that are injected separately, and would only detect the part with the keyword. Furthermore, static features fail to indicate techniques hidden under an obfuscation layer. However, our datasets contain only a few hundred cases of obfuscation, and our dynamic analysis lets us gain insight into the de-obfuscated strings. Out of the twelve static-only features that we validated, only the detection of HTML tag attributes other than `src` had any false positives (11 %), but we do not consider it an exploitation technique and only use it to complement our analysis. While such shortcomings are unacceptable for general-purpose attack detection, in our case we are only concerned with post-hoc analysis of two datasets, and we comprehensively validate our method.

4 Analysis

To provide some context, we start our analysis with an overview of the two exploit databases XSSED and OPENBUGBOUNTY. We report all results relative to the 133.9 k exploits (82.9% of raw submissions) for which our analysis framework successfully extracted and combined static and dynamic data. The XSSED data contains 36.2 k exploits for 28.7 k websites

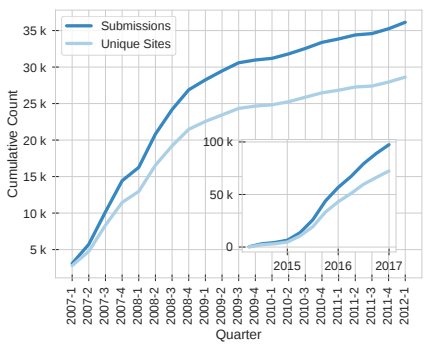


Figure 2: Quarterly exploit submissions and unique affected domains in XSSED (outer) and OPENBUGBOUNTY (inset). Most new submissions are found on new domains, suggesting a large supply of vulnerable domains on the Web.

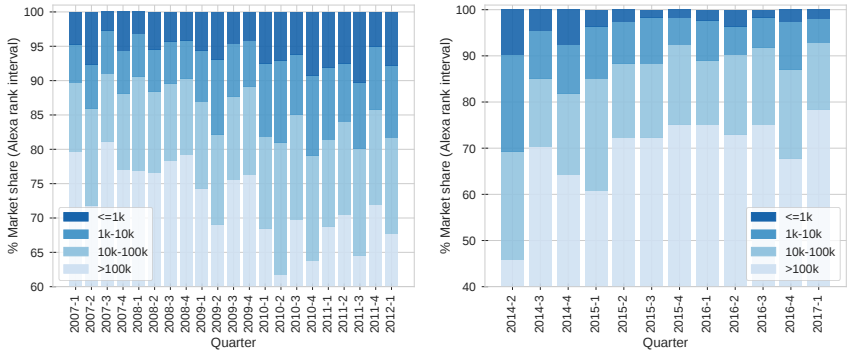


Figure 3: Quarterly distribution of the popularity of domains affected by exploit submissions (XSSED left, OPENBUGBOUNTY right). Domains grouped by popularity according to their Alexa ranks; the last interval includes unranked domains. More than half of submissions are for unpopular websites. The rank interval distribution is almost uniform over time, illustrating that XSS vulnerabilities continue to be found even on the most popular websites.

submitted between January 2007 and March 2015, although the site appears to have been mostly dormant since May 2012, as there have been fewer than ten monthly submissions since then. Our OPENBUGBOUNTY dataset covers June 2014 to April 2017 with 97.7 k submissions for 72.6 k websites. (We cannot practically extract newer submissions from OPENBUGBOUNTY due to newly implemented anti-crawling measures.)

The exploits in XSSED were submitted by 2.2 k users, whereas the much larger OPENBUGBOUNTY had only 883 active authors and 244 anonymous submissions. The dataset is heavily biased towards highly active users. The top 10 users accounted for 28.6 % of submissions in XSSED, and 38.3 % in OPENBUGBOUNTY. In contrast, around half of users in XSSED, and a third of users in OPENBUGBOUNTY, submitted only a single exploit. The dominance by a few active users implies that decisions made by these users, including their choice of exploits and their period of activity, can skew averages in our data. To account for this effect, we analyse aggregates not only in terms of submission quantities, but also by how many distinct authors submitted such exploits.

If submitters use automated tools, which appears to be the case for at least the most prolific users, there is a likely bias towards more shallow vulnerabilities, depending on the capabilities of these tools, and the possibility that submitted exploits contain techniques that are not necessary for the exploit to work on the respective website. Consequently, our analysis of exploits and their sophistication is to be read primarily as trends among exploit submitters, and as only loose upper bounds on the complexity of the website's vulnerability.

4.1 Affected Websites

The submitted exploits refer to 28,671 unique registered domains (below the public suffix) in XSSED and 72,607 in OPENBUGBOUNTY. Most domains receive a single submission; only 10.1 % and 13.3 % of them, respectively, appear in multiple vulnerability entries. These are often submitted in batches, as 37.2 % of delays between consecutive submissions for the same domain are below one day in XSSED, and 43.7 % in OPENBUGBOUNTY. Figure 2 shows that over time, total exploit submissions in both databases increase only slightly faster than the unique number of websites, suggesting that there is an "endless" supply of new websites with XSS vulnerabilities. In aggregate, the users submitting exploits appear to favour breadth over depth. The dataset is thus unsuitable for analysing the security posture of individual websites.

Since exploit authors are free in their choice of websites to scrutinise for XSS vulnerabilities, and we do not know about unsuccessful attempts, our data does not allow conclusions about vulnerability rates of websites. In the following, we study vulnerability reports according to the popularity of the affected website. We utilise the websites' (presumably contemporaneous) Alexa ranks reported in the submissions, and group them into exponentially increasing intervals. More popular sites tend to be overrepresented in our data, as users submitted vulnerabilities for 38 % of the 100 most popular websites in XSSED (40 % in OPENBUGBOUNTY), but only 3.2 % (8.9 %) of the 90 k websites ranked in the range 10 k – 100 k according to Alexa. As shown in Figure 3, these proportions remain somewhat stable over time. This suggests that the ability of exploit authors to find XSS vulnerabilities even in the most popular websites has not changed significantly.

4.2 Sink Analysis

Injected exploits can be reflected by the website in different parts of the server response. Since reflection vulnerabilities often arise from server-side templating with

user inputs, such reflection sinks could for example appear between HTML tags `<div>{sink}</div>`, in an HTML attribute `<div name="{sink}"`, or in a JavaScript context `<script>var name="{sink}";`. We manually labelled two random samples of 250 submissions from each of the two databases to determine the sink context of the exploit reflection. Nearly all sinks appear in an HTML-related context. In both databases, sinks are located between HTML tags, or inside HTML tag attribute values at comparable rates, with 52 % and 46.4 % in XSSED, and 44.4 % and 47.2 % in OPENBUGBOUNTY, respectively. JavaScript sinks make up only 1.2 % in XSSED and 7.2 % in OPENBUGBOUNTY.

Depending on the sink context, the exploit may need to *escape* from the current context and set up a new context suitable for the payload [14]. If the payload is `alert(1)` and the sink is located inside the value of an HTML attribute such as `<div name="{sink}"`, the payload needs to terminate the string, and create an event handler attribute that can accept the JavaScript payload, e.g., `" onmouseover=alert(1)`. Alternatively, it can close the tag, and then inject a script tag with the payload, such as `"><script>alert(1)</script>`. When exploit authors manually craft their attack string, they can customise it using their understanding of the website. For example, only a restricted set of tags can trigger the `onload` event handler, whereas it would have no effect in the other tags. Alternatively, exploit authors can use generic escape sequences that work in a variety of different sink types in order to make their exploit more versatile. In the same two random samples, 50.8 % of exploits in XSSED and 39.6 % in OPENBUGBOUNTY did not need any escaping, as the sink already had a JavaScript or HTML between-tag context suitable for the payload. However, 26.4 % of sampled XSSED exploits (OPENBUGBOUNTY: 28 %) contain an escaping sequence even though it is not necessary. This suggests that our datasets contain a significant fraction of general-purpose exploits. Around 41.6 % of sampled XSSED exploits, and 50.4 % in OPENBUGBOUNTY, contain an escape sequence that is both necessary and minimal. The remainder contains either no escaping, or additional unnecessary escaping.

Many websites reflect injected exploits more than once. In XSSED and OPENBUGBOUNTY, 37.0 % and 32.5 % of submissions in our successfully merged dataset have more than one working exploit reflection. This typically occurs due to one URL parameter being used in multiple places in the server-side template, but there are also a few submissions where different exploits are injected into multiple parameters. In the manually labelled sample, the multiple working reflections occur in 45.2 % for XSSED and 30.0 % for OPENBUGBOUNTY. These can be further divided into 32.4 % of the XSSED sample, and 24.4 % of OPENBUGBOUNTY, where the exploit is reflected multiple times in sink contexts of the same type, and 12.8 % and 5.6 %, respectively, with multiple reflections in different sink contexts. An exploit with correct escaping for one context can also appear in a different context where the escaping sequence may be ineffective.

To that end, it is worth noting that 44.7 % of all XSSED submissions, and 52.9 % of OPENBUGBOUNTY contain at least one additional potential exploit reflection where the exploit does not execute. This data is to be seen as a coarse approximation, as it contains false positives that are not actual exploit reflections, but matches between the request data and similar but potentially unrelated page code. For this reason, outside of this section, we only analyse executing reflections, where our dynamic analysis rules out such false positives. Overall, 62.1 % of submissions in XSSED, and 63.4 % in OPENBUGBOUNTY, contain multiple reflections, with at least one working and potentially more that do not. In addition to exploits being reflected in a sink context for which the escaping sequence is ineffective, another possible explanation for reflections not executing are server-side transformations.

A typical server-side transformation is encoding of sensitive characters to prevent XSS. In exploit reflections that do execute, few special characters such as < or > for tags, and " or ' for attribute values or strings are HTML entity encoded (i.e., `&lt;` `&#60;` or `&#x3c;`) – less than 0.1 % for angled brackets, and no more than 0.25 % for quotes in either database. In reflections that do not execute, the share of such encoding is significantly higher, with 7.3 % of these reflections in XSSED containing HTML-encoded angled brackets (OBB: 8.7 %), and 6.0 % (8.0 %) containing encoded quotes. HTML-encoding of alphanumeric characters is close to zero in either case. Since many server responses contain both working and non-working reflections, vulnerable applications appear to sanitise some user input, but inconsistently.

Some reflections appear to be mirroring the full request URL instead of a single request parameter. Around 6.4 % of submissions in XSSED and 15.1 % in OPENBUGBOUNTY contain at least one URL reflection, and 2.7 % and 4.4 % of submissions, respectively, contain at least one such URL reflection that executes. Similar to HTML encoding, URL reflections that execute injected exploits rarely contain any URL-encoded characters at all (XSSED: 3.6 %, OPENBUGBOUNTY: 0.9 %), whereas non-executing URL reflections do (81.5 % in XSSED and 73.0 % in OPENBUGBOUNTY).

Other factors that might prevent execution of an injected exploit, which we do not examine here, include an incompatible sink context, other types of encoding or escaping, and more complex server transformations such as substrings.

4.3 Exploit Analysis

Nearly all exploit submissions contain simple proof-of-concept payloads showing a JavaScript dialogue with a message. In XSSED, the earlier dataset covering five years since 2007, 99.7 % of all submissions use `alert()`. However, the prevalence of `alert()` appears to be decreasing over time in favour of `prompt()`. The former is used in 52.4 % of OPENBUGBOUNTY submissions, the latter in 40.7 %, and 7.6 %

use `confirm()`. These numbers may be heavily dependent on the behaviour of a few very active users, as 86.1 % of all authors in OPENBUGBOUNTY submit at least one exploit with `alert()`, and the fraction of active authors with at least one such exploit remains over 77.4 % in each quarter.

4.3.1 Overview of Exploitation Techniques

We group exploitation techniques into five categories, as shown in Table 4: Context (Escaping and Creation), Syntax Confusion, String Obfuscation & Quote Avoidance, Control Flow Modification, and String Interpretation. Each category corresponds to a specific goal of exploit authors, such as bypassing (incomplete) server-side sanitisation, or setting up a context where JavaScript code can be executed. The techniques within each category are alternative means of achieving that goal. Exploits might use multiple exploitation techniques at once. Very few submissions (17.3 % in XSSED, 3.8 % in OPENBUGBOUNTY) use no special technique at all; they are simple exploits such as `<script>alert(1)</script>`. The most common category is context escaping and creation with 75.5 % of submissions in XSSED, and 83.8 % in OPENBUGBOUNTY, most likely because techniques of this category are often needed to set up the proper execution context for the exploit, depending on the sink type. The remaining categories appear in only a small fraction of exploits in the early XSSED, but become more popular in the later OPENBUGBOUNTY. As an illustration, 67.6 % of submissions in XSSED use no technique other than possible context escaping and creation, showing that older exploits tend to be relatively simple. In OPENBUGBOUNTY, this percentage is only 15.6 % of submissions, as some techniques gained popularity and more authors have submitted at least one exploit with a technique from the other categories.

While categories such as control flow modification and syntax confusion have an upwards trend in OPENBUGBOUNTY, both in terms of submissions and authors, the string interpretation category remains rare, appearing in less than 1 % of submissions in either database, and used by only 1.9 % of XSSED and 5.0 % of OPENBUGBOUNTY authors. In the following, we look into each category in more detail.

4.3.2 Context (Escaping and Creation)

Depending on the sink context (Section 4.2), exploits need to escape from the current context and set up their own in which the payload can run. In our two datasets, most exploits (73.8% in XSSED, and 77.6 % in OPENBUGBOUNTY) close the previous tag, escaping to a context where new HTML tags can be inserted (e.g., `><script>`, C3 in Table 4). Only a few exploits escape from an HTML attribute but remain inside the tag to insert an event handler, such as `" onload=` (C4 in Table 4, 1.1 % in XSSED and 3.1 % in OPENBUGBOUNTY). Interestingly, though, a much higher fraction of

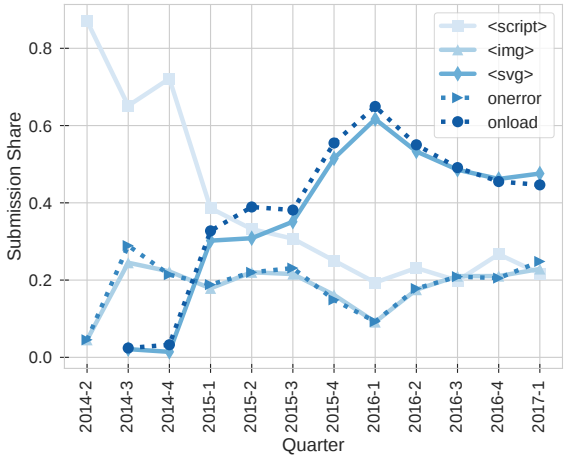


Figure 4: Quarterly tag and event handler market share (in terms of exploit submissions) in OPENBUGBOUNTY. Script tags decline in favour of alternative tags with event handlers.

authors, 20.4 % in OPENBUGBOUNTY, have submitted one such exploit at least once. Even fewer exploits insert their dialogue-based payload directly into a JavaScript context by possibly terminating a string and chaining the statement with `;` or an operator such as `+` (C5 in Table 4, 0.9 % in XSSED and 1.8 % in OPENBUGBOUNTY). While the latter finding is probably due to JavaScript sinks being much less common in our datasets than HTML sinks, the dominance of HTML tag escaping over remaining inside the tag is more likely attributed to preferences of exploit authors, as both types of HTML sinks are similarly common.

As most exploits close the previous tag, they must insert a new tag in order to be able to execute the payload. Indeed, 98 % of exploits in XSSED, and 93.5 % in OPENBUGBOUNTY contain at least one tag. In the older XSSED dataset, with 95.6 % of submissions, this is nearly always a `<script>` tag. In the more recent OPENBUGBOUNTY, this tag is found in an average of only 26.8 % of submissions. As Figure 4 shows, its use declined from 87.1 % of submissions in the second quarter of 2014 to 21.7 % in the first quarter of 2017. Author numbers similarly declined from 100 % to 65 % submitting at least one exploit with a `<script>` tag in a quarter. While barely used at all in the beginning of OPENBUGBOUNTY, `<svg>` and `<img>` tags have become more popular, with an overall use in 45.6 % and 18.5 % of exploits, respectively. Interestingly, many of these `<svg>` submissions appear to originate from a smaller set of users, as only 34.8 % of authors have ever submitted such an exploit. In the case of `<img>`, the trend is opposite. The fewer submissions were made by a larger 49.3 % of authors. Presumably, alternative tag names can circumvent filtering in websites. They appear to be used in combination with event handlers, which we investigate in Section 4.3.5.

Table 4: Detected Exploitation Techniques by Category.
(◐ static, ◑ dynamic, ● combined detection methodology; percentages for XSSED / OPENBUGBOUNTY)

Context (Escaping and Creation)

	Technique	Example	Detection	Submissions	Authors
C1	HTML comment	`--><script>alert(1)</script>`	◐	2.4 % / 10.5 %	9.2 % / 21.7 %
C2	JavaScript comment	`/*/* */prompt(1)//`	◐	0.5 % / 3.3 %	2.7 % / 12.9 %
C3	HTML tag escape and tag insertion	`"><script>alert(1)</script>`	◐	73.8 % / 77.6 %	66.7 % / 80.7 %
C4	HTML attribute escape and event handler	`" autofocus onfocus=alert(1)`	◐	1.1 % / 3.1 %	5.6 % / 20.4 %
C5	chaining onto prior JavaScript expression	`"-alert(1)` or `;prompt(1)`	◐	0.9 % / 1.8 %	5.0 % / 14.4 %
	(any in category)			75.5 % / 83.8 %	68.6 % / 83.6 %

Syntax Confusion

	Technique	Example	Detection	Submissions	Authors
S1	extraneous parentheses	`(alert)(1)`	◐	0.0 % / 0.3 %	0.0 % / 1.8 %
S2	mixed case	`<scRipT>alert(1)</scRipT>`	◐	4.9 % / 4.4 %	8.8 % / 19.8 %
S3	JavaScript encoding (uni, hex, oct)	`\u0061lert(1)` or `top["\x61lert"](1)`	◐	0.0 % / 0.1 %	0.1 % / 2.3 %
S4	malformed img tag	`<img """><script>alert(1)</script>">`	◐	0.2 % / 0.2 %	0.6 % / 1.6 %
S5	whitespace characters	`<svg%0Aonload%20=␣alert(1)>`	◐	0.0 % / 0.0 %	0.1 % / 0.7 %
S6	slash separator instead of space	`<body/onpageshow=alert(1)>`	◐	0.1 % / 42.8 %	0.1 % / 31.7 %
S7	multiple brackets (tag parsing confusion)	`<<script>alert(1)<</script>`	◐	8.2 % / 1.7 %	5.8 % / 8.8 %
	(any in category)			13.0 % / 47.8 %	14.2 % / 39.7 %

String Obfuscation & Quote Avoidance

	Technique	Example	Detection	Submissions	Authors
O1	character code to string	`alert(String.fromCharCode(88,83,83))`	●	4.5 % / 3.7 %	11.1 % / 11.7 %
O2	regular expression literal	`prompt(/XSS/)`	●	13.7 % / 65.1 %	16.7 % / 62.8 %
O3	base64 encoding	`alert(atob("WFNT"))`	●	0.0 % / 0.1 %	0.0 % / 0.6 %
O4	backtick	``prompt`XSS` ``	◐	0.0 % / 2.4 %	0.1 % / 8.3 %
	(any in category)			18.2 % / 71.1 %	25.0 % / 67.7 %

Control Flow Modification

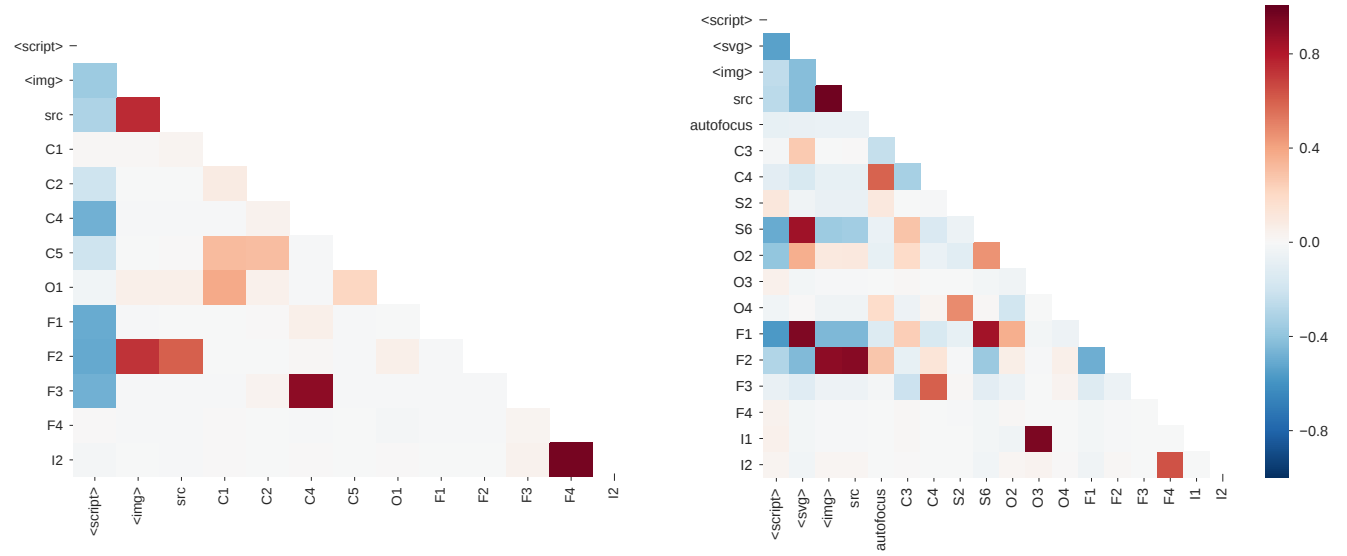
	Technique	Example	Detection	Submissions	Authors
F1	automatically triggered events	`<svg onload=alert(1)>`	●	1.2 % / 48.2 %	5.9 % / 38.1 %
F2	exploit-triggered events	`<input autofocus onfocus=alert(1)>`	●	1.2 % / 20.8 %	3.6 % / 50.6 %
F3	user interaction events	`onmouseover=prompt(1)`	●	1.0 % / 1.7 %	5.1 % / 16.9 %
F4	dialogue assignment to variable	`a=alert;a(1)`	●	0.6 % / 0.1 %	0.7 % / 2.3 %
F5	throw exception	`window.onerror=alert;throw/1/`	●	0.0 % / 0.2 %	0.0 % / 1.6 %
F6	page code interaction	exploit invoked by existing code in the page (call stack)	◑	0.0 % / 0.1 %	0.3 % / 4.2 %
	(any in category)			4.1 % / 70.8 %	11.3 % / 67.7 %

String Interpretation

	Technique	Example	Detection	Submissions	Authors
I1	document.write	`document.write("<script>alert(1)</script>")`	●	0.1 % / 0.1 %	0.8 % / 0.5 %
I2	eval	`eval("alert(1)")`	●	0.7 % / 0.2 %	1.3 % / 2.9 %
I3	setInterval/setTimeout	`setTimeout("alert(1)", 20000)`	●	0.0 % / 0.0 %	0.0 % / 0.2 %
I4	top/window key access	`top["alert"](1)`	◐	0.0 % / 0.1 %	0.0 % / 1.5 %
	(any in category)			0.8 % / 0.4 %	1.7 % / 4.5 %

Summary

	Technique	Example	Submissions	Authors
	(no technique used at all)	`<script>alert(1)</script>`	17.3 % / 3.8 %	50.7 % / 42.2 %
	(no technique or Context Creation)	`"><script>alert(1)</script>`	67.6 % / 15.6 %	84.0 % / 61.4 %



(a) XSSED (b) OPENBUGBOUNTY

Figure 5: Correlation matrix of exploitation techniques and selected tags and attributes for (a) XSSED and (b) OPENBUGBOUNTY. Use of `<script>` is negatively correlated with most other techniques, meaning they are used alternatively. The popular exploit pattern `<img src=# onerror>` is visible as the pairwise correlation between `<img>`, `src`, and F2. Rarely used techniques such as I1 and O3, or I2 and F4, may be correlated because most uses occur in similar submissions by the same author.

4.3.3 Syntax Confusion

Under Syntax Confusion, we consider techniques using non-standard or intentionally "confusing" syntax that is tolerated by browsers, but not understood by web application filters and their parsers. Only three techniques of this category see non-negligible use in the two exploit databases. In OPENBUGBOUNTY, 42.8 % of submissions use a slash instead of a space (S6 in Table 4; `<svg/onload`); this is mostly due to its use in a frequently submitted exploit pattern (Section 4.4). The slash technique is barely used in the older XSSED. On the other hand, using multiple angled brackets in an apparent attempt to confuse simple filter parsers (S7 in Table 4; `<<script>`) is most popular in XSSED with 8.2 % of submissions. Both databases see a low but persistent use of mixed case tag names or event handlers (S2 in Table 4; `<scRipT>`). Other syntax confusion techniques, such as JavaScript source code character encodings (S3 in Table 4; `\u0061lert(1)`) or whitespace characters (S5; `<svg%0Aonload`), appear in very few submissions.

4.3.4 String Obfuscation & Quote Avoidance

In this category, we group obfuscation using built-in JavaScript methods to decode character code sequences or base64 strings containing potentially filtered keywords, and use of regular expression literals or backticks to avoid injecting quotes and/or parentheses. The regular expression technique (O2 in Table 4; `alert(/message/)`) is used in a majority of OPENBUGBOUNTY submissions, but much less in XSSED. The three remaining techniques occur in small percentages of exploits. Two of them, character code sequence decoding (O1 in Table 4) and backticks (O4 in Table 4; `alert`message``), are known to a larger fraction of authors, but used in fewer submissions overall.

4.3.5 Control Flow Modification

Control flow modification techniques mostly avoid straightforward use of potentially filtered `<script>` tags in favour of alternative ways of executing code. None of them is used widely in XSSED, but automatically triggered event handlers (F1 in Table 4) can be found in almost half (48.2 %) of OPENBUGBOUNTY submissions. The most frequently used event handler of this category is `onload` (47.7 % of OPENBUGBOUNTY submissions, 38.1 % of authors). Other event handlers can be triggered automatically if the exploit contains the appropriate setup (F2 in Table 4). Among those, we observe `onerror` together with an invalid URL (18.7,% of submissions, 48.6 % of authors), and `onfocus`, which commonly co-occurs with the `autofocus` attribute (2.1 % of submissions, 12.2 % of authors). Comparing the submission to author percentages, `onload` appears to be used by disproportionately active submitters, whereas `onerror` is known to more users, but submitted less frequently. OPENBUGBOUNTY exploits with

XSSED					
Rank		Techniques	Example	Submissions	Authors
1	1	C3	`"><script>alert(1)</script>`	49.0 %	53.2 %
2	2	(none)	`<script>alert(1)</script>`	17.3 %	50.7 %
3	3	C3, O2	`"><script>alert(/XSS/)</script>`	7.8 %	11.0 %
4		C3, S7	`"><script>alert(1)</script>>`	7.3 %	3.8 %
5	4	O2	`<script>alert(/XSS/)</script>`	3.4 %	8.0 %
	5	C3, S2	`"><ScRipt>alert(1)</sCripT>`	2.9 %	5.4 %
OPENBUGBOUNTY					
Rank		Techniques	Example	Submissions	Authors
1		C3, S6, F1, O2	`"><svg/onload=prompt(/XSS/)>`	30.9 %	18.5 %
2	3	C3, F2, O2	`"><img src=x onerror=prompt(/XSS/)>`	9.6 %	27.7 %
3	1	C3	`"><script>alert(1)</script>`	8.5 %	42.1 %
4	4	C3, O2	`"><script>alert(/XSS/)</script>`	4.7 %	26.0 %
5	2	(none)	`<script>alert(1)</script>`	3.8 %	42.1 %
	5	F2, O2	`<img src=x onerror=prompt(/XSS/)>`	2.7 %	23.4 %

Table 5: Most common exploit patterns (ordered by submissions; rank for author use in second column).

event handlers that require user interaction (F3 in Table 4) are dominated by `onmouseover` (1.5 % of submissions, 14.5 % of authors) and `onmousemove` (0.1 % of submissions, 2.0 % of authors).

The most frequent combinations of tags and event handlers in OPENBUGBOUNTY are `<svg>` with `onload` (45.1 % of submissions), `<img` with `onerror` (18.0 %), `<iframe>` with `onload` (1.2 %), `<body>` with `onload` (1.1 %), and many additional combinations each used in fewer than 1 % of submissions. As shown in Figure 4, event handlers in OPENBUGBOUNTY have not always been equally popular. Rather, their rise coincides with a decline in the use of `<script>` tags, which are more likely to be filtered than the more innocuous-looking `<svg>` or `<img>` tags. Direct code execution with `<script>` tags appears to be replaced by indirect execution using a combination of alternative tags and event handlers.

We also detect technically interesting techniques, such as interleaving exploit code so that it will be called by existing page code (F6 in Table 4), but they are observed rarely.

4.3.6 String Interpretation

Code and markup string interpretation methods such as `eval()` and `document.write()` could be used for custom exploit obfuscation. Our analysis of the 491 strings passed to these methods revealed only 9 additional instances of techniques that were not already visible during the static analysis. In the two databases, string interpretation methods appear to be used as proof-of-concept exploits rather than for actual obfuscation. Overall, the use of string interpretation techniques is very low, both in terms of submissions and authors. The most frequent technique is `eval()` (I2 in Table 4) with 0.7 % of submissions in XSSED, and 0.2 % in OPENBUGBOUNTY (1.3 % and 2.9 % of authors, respectively).

4.4 Exploit Patterns

Many exploitation techniques co-occur in characteristic combinations. (A correlation matrix is shown in Figure 5.) This suggests that authors may be using common patterns or templates for exploits. An interesting question is whether users submit *similar* exploits. To this end, we do not consider string equality to be a useful metric, as two exploits displaying different messages may be identical from a technical point of view, even though their string representation differs. Instead, we cluster submissions by the set of techniques (in Table 4) that are used in the exploit. We do not distinguish whether exploits use `alert()` or `prompt()`, do not consider tag names, and we only make a difference between three classes of event handlers. This level of abstraction is intended to balance robustness against minor modifications that do not result in a different control flow, yet reflect the author's use of exploitation techniques such as syntax tricks.

Our approach results in 178 distinct exploit patterns in XSSED, out of which 60.1 % are submitted at least twice and 51.7 % are used by at least two authors. In OPENBUGBOUNTY, we detect 484 exploit patterns, with 65.9 % used multiple times and 54.3 % used by multiple authors. For comparison, exact string matching finds 15,567 and 31,337 unique exploit strings, with only 13.5 % and 12.6 % of them used more than once (2.9 % and 3.4 % used by multiple authors).

The ten most frequent patterns account for 92 % of all submissions in XSSED, and 69.9 % in OPENBUGBOUNTY. Out of all authors, 93.8 % in XSSED, and 85.4 % in OPENBUGBOUNTY, have submitted at least one exploit based on one of these top ten patterns. Example string representations of these patterns are shown in Table 5. An example for the most frequently submitted pattern in XSSED is `"><script>alert("XSS")</script>`; it accounts for 49.0 % of all submissions and is used by 53.2 % of users. The

same pattern, or its variant without tag closing, is also the most popular in OPENBUGBOUNTY when considering the number of authors who have submitted it at least once. However, both patterns together account for only 12.3 % of total submissions. The most frequently submitted OPENBUGBOUNTY pattern is `"><svg/onload=prompt(/XSS/)>`, adding space and quote avoidance, and indirect code execution using an automatically triggered event handler. It accounts for 31.3 % of all submissions, but is used by only 19.1 % of users, which implies that these participants are disproportionately active.

4.5 Findings and Implications

We summarise below the main findings and implications of our analysis.

- Roughly half of submissions in both databases contain at least one non-executable input reflection in addition to the working exploit. Web developers may be aware of the need for input sanitisation, but apply it inconsistently.
- The older XSSED contains 67.6 % submissions with no particular exploitation technique at all or just context escaping, whereas almost half in the newer OPENBUGBOUNTY use automatically triggering event handlers. This trend away from direct execution using injected script tags towards indirect execution using non-script tags is presumably to circumvent simple input filters.
- Some techniques, such as using a slash instead of a space for separation `<svg/onload`, or using a regular expression instead of quotes `alert(/message/)`, are frequent in large part because a few submitters use them in their large bulk submissions. Especially OPENBUGBOUNTY is dominated by a small number of very active users, likely due to automation. The choices made by these users have an outsize effect on aggregate results when it comes to the prevalence of exploitation techniques.
- Many vulnerabilities appear to be detected using automated tools, as evidenced by large bulk submissions and general-purpose exploits that include unnecessary context escaping. If website operators proactively used similar tools to test their own websites, they might be able to prevent a large number of the submitted vulnerabilities.
- Users appear to favour breadth by covering new websites more than existing websites, but they are still able to find new vulnerabilities even on the most popular websites. The discovery of cross-site scripting vulnerabilities on the Web seems to be far away from saturation.
- The databases contain few submissions with more complex techniques such as control flow modification or string interpretation. Possibly due to a lack of incentives, users may be looking for new websites that are vulnerable to an existing general-purpose exploit, as opposed to searching for new exploits on an existing website.

5 Discussion & Conclusion

Our longitudinal analysis of exploitation techniques in XSSED and OPENBUGBOUNTY submissions has shown that most reflected server XSS exploits are surprisingly simple, with an only moderate increase in sophistication over ten years. For example, few exploits use obfuscation, possibly because users lack incentives to submit more complex exploits. Similarly, in additional experiments we found that the vast majority of exploits could have been blocked by the default settings of filters that were available in browsers at the time the exploits were submitted, such as XSS Auditor in Chrome, the XSS filter in Internet Explorer, or NoScript for Firefox.

Currently, submitters gain recognition primarily through the number of exploits they submit. This appears to encourage submitters to use automated tools and scan a nearly boundless supply of vulnerable websites with simple, general-purpose exploits. We believe that XSS databases could increase their utility by leveraging human skill for tasks that are more difficult to automate. Bug bounty programmes could encourage depth over breadth by restricting the set of eligible websites, or by scoring submissions by the "complexity" or uniqueness of the techniques needed to make the exploit work.

The relative simplicity of exploits in the two databases, also anecdotically observed by Pelizzi and Sekar [17], has implications for researchers using them for model training or system evaluation. Ideally, a sample of exploits used for these purposes should cover a diverse set of technical conditions. In reality, however, most exploits in the two databases are technically equivalent. As a result, random samples of exploits contain only a few complex examples that would challenge the system to be evaluated. For a more diverse sample, researchers could apply an approach similar to ours and select exploits based on different patterns.

Our data does not allow conclusions about the security posture of individual websites and how it evolves over time. Yet, standard, low-sophistication exploits appear to be effective on a large set of websites including the most popular ones, demonstrating that shallow XSS vulnerabilities are still extremely widespread on the Web.

We hope that our research will spark new directions with the goal of improving the security posture of existing websites and mitigating the prevalent XSS attacks. The data used in this paper is available online [1].

6 Acknowledgements

We would like to thank Ahmet Talha Ozcan for his help with the data collection. We thank our anonymous reviewers for their valuable feedback on drafts of this paper. This work was supported by the National Science Foundation under grant CNS-1703454.

References

[1] Annotated XSS dataset for this paper. http://cdn.buyukkayhan.com/public/xss_sqlite.db, 2020.

[2] Rudolfo Assis. XSS cheat sheet. https://brutelogic.com.br/blog/cheat-sheet/, 2017.

[3] Daniel Bates, Adam Barth, and Collin Jackson. Regular expressions considered harmful in client-side XSS filters. In *WWW*, 2010.

[4] Khalil Bijjou. Web application firewall bypassing - How to defeat the blue team. In *OWASP Open Web Application Security Project*, 2015.

[5] Alejandro Calleja, Juan Tapiador, and Juan Caballero. A look into 30 years of malware development from a software metrics perspective. In *RAID*, 2016.

[6] K. Fernandez and D. Pagkalos. XSSed | Cross site scripting (XSS) attacks information and archive. http://xssed.com.

[7] Matthew Finifter, Devdatta Akhawe, and David Wagner. An empirical study of vulnerability rewards programs. In *Usenix Security*, 2013.

[8] Mauro Gentile. Snuck payloads. https://github.com/mauro-g/snuck/tree/master/payloads, 2012.

[9] HackerOne.com. Bug bounty – hacker powered security testing | HackerOne. https://hackerone.com/.

[10] Robert Hansen, Adam Lange, and Mishra Dhira. OWASP XSS filter evasion cheat sheet. https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet, 2017.

[11] Mario Heiderich. HTML5 security cheatsheet. https://html5sec.org/, 2011.

[12] Philipp Holzinger, Stefan Triller, Alexandre Bartel, and Eric Bodden. An in-depth study of more than ten years of Java exploitation. In *CCS*, 2016.

[13] Vladimir Ivanov. Web application firewalls: Attacking detection logic mechanisms. In *BlackHat*, 2016.

[14] Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later - Large-scale detection of DOM-based XSS. In *ACM CCS*, 2013.

[15] William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. Riding out DOMsday: Towards detecting and preventing DOM cross-site scripting. In *NDSS*, 2018.

[16] OpenBugBounty.org. Open Bug Bounty | Free bug bounty program & coordinated vulnerability disclosure. https://openbugbounty.org.

[17] Riccardo Pelizzi and R Sekar. Protection, usability and improvements in reflected XSS filters. In *ASIACCS*, 2012.

[18] Jukka Ruohonen and Luca Allodi. A bug bounty perspective on the disclosure of Web vulnerabilities. In *WEIS*, 2018.

[19] Theodoor Scholte, Davide Balzarotti, and Engin Kirda. Quo vadis? A study of the evolution of input validation vulnerabilities in Web applications. In *Financial Crypto*, 2011.

[20] Ben Stock, Stephan Pfistner, Bernd Kaiser, Sebastian Lekies, and Martin Johns. From facepalm to brain bender: Exploring client-side cross-site scripting. In *ACM CCS*, 2015.

[21] Mingyi Zhao, Jens Grossklags, and Peng Liu. An empirical study of Web vulnerability discovery ecosystems. In *CCS*, 2015.