XSS PEEKER: Dissecting the XSS Exploitation Techniques and Fuzzing Mechanisms of Blackbox Web Application Scanners

Enrico Bazzoli[1], Claudio Criscione[2], Federico Maggi[1], and Stefano Zanero[1]

[1]Politecnico di Milano, [2]Google Zurich

Abstract. Black-box vulnerability scanners can miss a non-negligible portion of vulnerabilities. This is true even for cross-site scripting (XSS) vulnerabilities, which are relatively simple to spot. In this paper, we focus on this vulnerability class, and systematically explore 6 black-box scanners to uncover how they detect XSS vulnerabilities, and obtain useful insights to understand their limitations and design better detection methods. A novelty of our workflow is the retrofitting of the testbed so as to accommodate payloads that triggered no vulnerabilities in the initial set. This has the benefit of creating a systematic process to increase the number of test cases, which was not considered by previous testbed-driven approaches.

1 Introduction

Web application vulnerabilities are one of the most important and popular security issues, constantly making it to the top list of disclosed vulnerabilities [13, 18]. Cross-site scripting (also known as XSS) is a prevalent class of web application vulnerabilities [10]. In June 2013, XSS was reported as the most prevalent class of input validation vulnerabilities by far [16].

Black-box vulnerability scanners are widely used in the industry but, unfortunately, they can miss a non-negligible portion of vulnerabilities [2, 14, 17] or report non-existing, non-exploitable or uninteresting vulnerabilities [17]. Although XSS vulnerabilities are relatively easy to discover, previous work showed that black-box scanners exhibit shortcomings even in the case of XSS flaws.

To our knowledge there is no detailed study of black-box web vulnerability scanners that focused specifically on XSS vulnerabilities and their detection approaches. Previous work and commercial benchmarks included XSS bugs as part of the set of flaws in testbed web applications, but not as the main focus. Also, previous work measured the detection rate and precision of the scanners mainly with the goal of benchmarking their relative performance. Although these indicators are important, we believe that providing precise insights on the structure, generality, fuzzing mechanism and overall quality of the XSS detection approaches could help web application developers to design better escaping and validation routines, and scanner vendors to *understand* the reasons behind scanner weaknesses.

These goals require a new perspective. A naïve approach would be to launch the scanners against large set of web applications, with difficult-to-find entry points, complex session mechanisms, etc. Normally, this kind of testbeds are adopted by vendors to

demonstrate their product's sophistication. However, since our goal is *not* to challenge a scanner, but to analyze its payloads, exploitation approaches and fuzzing algorithms, we need a comprehensive set of targeted test cases that the scanner can easily find and exercise, such that the number of extracted payloads is maximized. From here the name of our tool, "XSS PEEKER," that implements our workflow.

XSS PEEKER works at the network level and decodes the HTTP layer to find and extract the XSS payloads. In addition, XSS PEEKER streamlines tedious tasks such as finding groups of related payloads, which we call *templates*, making automatic analysis and result visualization feasible even in the case of large amounts of payloads. Our research is not limited to individual scanners: we observed how cross-scanner analysis yields very interesting results.

Part of our contributions is a testbed web application, *Firing Range* (`http://public-firing-range.appspot.com`) It is designed such that it is easy for the scanner to find the vulnerable entry points. Moreover, it is very easy to add new vulnerable entry points. Indeed, as part of our workflow, whenever XSS PEEKER encounters a payload that does not trigger any vulnerability, it displays an informative alert to the developer who can quickly prepare a test case that would satisfy specifically that payload. We applied this process iteratively, running new scans and collecting new payloads. Last, a difference of our testbed application with respect to the state of the art is the inclusion of specific test cases for DOM XSS vulnerabilities, one of the most recently discovered and emerging sub-classes of XSS flaws [5, 7].

Overall, our methodology and results are useful to answer questions such as: which payloads are used by security scanners to trigger XSS vulnerabilities? Can we rank them and identify a set of "better" payloads? Is a scanner exercising all the entry points with the expected payloads? A direct application of our results toward improving the quality of a scanner is, for example, to minimize the size and number of characters used in the payloads, while keeping the detection rate constant. Payloads of short size and with a smaller character set have in general higher chances of bypassing filters, and thus higher possibilities of uncovering unknown vulnerabilities. Using better payloads also means generating fewer requests and thus completing a test in a shorter time (i.e., better scanning efficiency). Clearly, these are just a couple of examples, which we formalize through a set of criteria later in the paper. This type of analysis should be focused on a scanner testing phase in isolation, which is best achieved with a fully synthetic test bed: a realistic test application would not improve the quality of our analysis in any of our measures and would introduce yet another experimental parameter to factor in all our considerations.

In summary, we make the following contributions:

- A publicly available testbed web application that exposes a wide range of non-trivial XSS vulnerabilities, augmented with the new cases progressively discovered while running the scanners.
- A methodology to analyze how black-box web application scanners work by (1) extracting the payloads from the HTTP requests, (2) clustering them to scale down the evaluation challenge and keep it feasible, and (3) evaluating each cluster in terms of use of evasion, compactness, and other quality indicators.
- A publicly available prototype of XSS PEEKER (`https://goo.gl/zJrJ1i`)

– A detailed evaluation of 6 scanners: Acunetix 8.0, NetSparker 3.0.15.0, N-Stalker 10.13.11.28, NTOSpider 6.0.729, Skipfish 2.10b and w3af 1.2.

2 Background

Before describing our approach in detail, we introduce the essential background concepts and terminology on web application XSS vulnerabilities and black-box scanners.

XSS *attacks* consist in the execution of attacker-controlled code (e.g., JavaScript) in the context of a vulnerable web application. In this paper, we refer to the portion of malicious code as *payload*. Without aiming for a complete taxonomy, XSS vulnerabilities and attacks can be divided in *stored* and *reflected*. In reflected attacks the victim is tricked (e.g., through links or short URLs [8] embedded in e-mail or instant messages) into sending a specially crafted request—which embeds the actual payload, which is bounced back to the client immediately. In stored XSS attacks the moment the payload is injected is decoupled from the moment that it is effectively displayed and executed by the victim, as the attacker's code achieves some form of persistence.

DOM-based attacks [6] can be seen as an orthogonal category. They rely on the insecure handling of untrusted data through JavaScript (e.g., `document.write('<script...>')`) rather than static inclusion of payload (e.g., `<script...>`) in the rendered page.

Black-box web vulnerability scanners leverage a database of known exploits, including XSS payloads, used to trigger and detect potential vulnerabilities. They start by *crawling* the target web application to enumerate all reachable *entry points* (e.g., links, input fields, cookies), then they generate (mutations of) input strings based on their database, inject the resulting payload in the entry points and finally analyze the HTTP responses using an *oracle* to validate the presence of vulnerabilities (e.g., by looking for the injected payload in the output).

3 Firing Range: Test Case Generation

The implementation of the testbed web application is a key point. Given our goals and needs, the requirements of such a testbed are: (i) to have clearly defined vulnerabilities and entry points, (ii) to be easily customizable, (iii) to contain the main types of XSS vulnerabilities. Large, full-fledged testbed web applications have been implemented in previous works [1, 2, 3] and are constantly made available to the public, but they do not entirely meet our requirements and approach. Although requirement (iii) is easy to ensure by modifying existing testbed applications, ensuring (i) and (ii) implies a complete redesign and re-engineering. In fact, existing testbed applications are not focused on *extracting* as many payloads as possible from the scanner. Contrarily, they are focused on challenging the scanner's capabilities of discovering hard-to-find vulnerable entry points. Given these premises, we decided that it was easier to implement our own testbed and release it to the community.

3.1 Design Challenges

Meeting the above requirements is tricky. On one hand, as pointed out in [2], the complexity of some applications can hinder the coverage of the scanner. On the other hand, there is no guarantee of comprehensiveness of the testbed.

We decided to address these shortcomings explicitly while designing *Firing Range*. Our testbed exposes all vulnerabilities through HTML anchors, and vulnerable parame-

ters are provided with sample input to improve discoverability. This approach allows to run testing sessions by providing the full list of vulnerable URLs, thus removing any crawling-related failure entirely. Furthermore, each vulnerable page is served as a standalone component with no external resources such as images or scripts, as to create a minimal, clean and efficient test case. The result is that the scanner focuses on the juicy part: the exploit generation and fuzzing.

3.2 Implementation Challenges

The main implementation challenge when creating a testbed is of course deciding which tests to include. We wanted our initial set to cover as many cases as possible, but there was no such list readily available in previous literature.

Since we basically wanted to test the detection of XSS on an HTML page, we observed that the HTML parsers in modern browsers have a finite set of states, which make a good starting point to create test cases. Thus, we created one test per HTML parser state. To this end we analyzed the different contexts identified by the contextual auto-escaper described in [12]: simply reversing the perspective of a parser, tasked with applying proper escaping to malicious payloads, provided us with clear samples of the different contexts. We generated test cases covering each of them with the goal of (1) producing a "vulnerable baseline" of the different states of an HTML parser and (2) inducing the scanners to inject as many payloads as possible.

HTML contexts are however not enough to generate test cases for DOM XSSs, which exploits interactions between the DOM generated by parsing the original HTML and JavaScript code. For DOM XSS, we started from the XSS Wiki[1], and other openly available collections of sample vulnerabilities, and generated a list of valid DOM sinks and sources—which, notably, include sources other than URLs such as cookies and browser storage. Each one of our DOM tests couples one of these sinks and sources. In the following example, the source is `location.hash` and the sink is `innerHTML` of a `<div>` node:

DOM XSS from location.hash to innerHTML.

```
<body>
  <script>
    var payload = window.location.hash.substr(1);
    var div = document.createElement('div');
    div.id = 'divEl';
    document.documentElement.appendChild(div);

    var divEl = document.getElementById('divEl');
    divEl.innerHTML = payload;
  </script>
</body>
```

We then varied the sources and sinks, obtaining test cases like the following ones:

DOM XSS from `documentURI` to `document.write()`.

```
<body>
  <script>
    var payload = document.documentURI;
    document.write(payload);
  </script>
</body>
```

DOM XSS from `window.name` to `eval()`.

```
<body>
  <script>
```

[1] https://code.google.com/p/domxsswiki/wiki/Introduction

```
    var payload = window.name;
    eval(payload);
  </script>
</body>
```

We manually verified all the initial tests as exploitable with an appropriate payload or attack technique.

3.3 Iteratively Discovered Test Cases

During our experimental evaluation XSS PEEKER discovered payloads that were not exercising any test case (i.e., vulnerability). Instead of limiting our analysis to report this gap, our approach is to iteratively construct new test cases and progressively re-run all the scanners. In other words, our testbed application is dynamic by design. The details of this iterative procedure are described in Section 4.4, whereas the numbers of test cases that we had to add in order to accommodate an existing exploitation payload are reported in Section 5.4.

This iterative process produced 42 new test cases that were not identified by our initial seeding. In other word, we use the testbed web application as an oracle with respect to the scanner, and we use the scanner as an oracle with respect to the web application. As a result, this dual approach greatly improved the testbed and provided new insights on the internals of each scanner.

When considering XSS PEEKER's analysis perspective, there is no functional difference between stored and reflected XSS. The difference is whether the echoed payload is stored or not. However, from the point of view of the payload analysis, which is our core focus, reflected or stored XSSs are equivalent. Therefore, for ease of development and of experimental repeatability, our testbed web application only contains reflected vulnerabilities.

Our publicly available testbed includes full details of the vulnerabilities. For the sake of brevity, we refer the reader directly to `http://public-firing-range.appspot.com` for a complete list of live test cases and commented source code.

4 XSS PEEKER: Analysis Workflow

XSS PEEKER automates the extraction and analysis of XSS payloads by following an iterative approach, divided in four phases (the first three completely automated, the fourth partially relying on manual inputs).

4.1 Phase 1 (Payload Extraction)

The high-level goal of this phase is to obtain, for each scanner, the entire set of XSS payloads used by each scanner for each entry point in the testbed application. To this end, this phase first captures (using `libpcap`) the traffic generated by the scanners during the test and performs TCP stream reassembly to the full HTTP requests.

The first challenge is to automatically separate the HTTP requests used for the actual injection (i.e., containing one or more payload) from the requests used for crawling or other ancillary functionalities, which are not interesting. The ample amount of payloads generated makes manual approaches unfeasible. Therefore, we rely on two heuristics:

Signature Payloads: Most scanners use *signature payloads* (i.e., payloads that contain strings that are uniquely used by that scanner). Therefore, we derived the signature

payloads for each scanner and compiled a whitelist that allows this heuristic to discard the uninteresting requests.

Attack Characters: Since we know the testbed application, we guarantee that there is no legitimate request that can possibly contain certain characters in the header or body parameters. These characters include, for example, `<`, `>`, `'`, `"`, and their corresponding URL-percent encodings. Such characters should not be present in a crawling request by construction, and since they are often required to exploit XSS vulnerabilities, we have empirically observed them as linked to vulnerability triggering.

To complement the previous heuristics and maximize the number of identified payloads, we perform pairwise comparisons between requests issued by each couple of scanners. For each couple, we extract the body and URL of the two requests and check if they have the same path and the same query parameters. If so, we compare the values of each query parameter. By construction, Firing Range provides only a single value for each parameter, thus any mismatch has to be originated by the scanner fuzzing routine. Once a pair of requests is flagged as a mismatch, we performed manual inspection to isolate the payload. The number of such cases is rare enough to make this a manageable process. We iteratively applied and improved these heuristics until this cross-scanner analysis generated empty output, and we could confirm through manual inspection that no more test requests were missed (i.e., all payloads considered).

We can focus just on query parameters because of the design of our testbed, which provides injection points exclusively on query parameters. Even if almost all scanners also test path injection, we chose not to analyze them. Indeed, manual analysis confirmed that scanners used the very same set of payloads observed during parameter injection.

4.2 Phase 2 (Payload Templating)

Given the large number of payloads generated by each scanner, manually analyzing and evaluating each of them separately is practically unfeasible. A closer inspection of the payloads, however, revealed self-evident clusters of similar payloads. For example, the following payloads: `<ScRiPt>prompt(905188)</ScRiPt>` and `<ScRiPt>prompt(900741)</ScRiPt>` differ only for the parameter value. To cluster similar payloads, inspired by the approach presented in [11], we developed a recursive algorithm for string templating. Without going into the details, the approach presented in [11] is to start from a set of template elements that produce fully random or dictionary based sequences of symbols. Using a large corpus of spam emails, the approach is to derive the full spam email template by generalizing the template elements. Emails, however, are much larger, structured, and richer of contextual symbols than our exploitation payloads. Therefore, the approach described in [11] cannot be applied as is. Essentially, we cannot easily define the concept of "template elements" in such a small space. Therefore, as part of our contributions, we create a new approach that is well suited for short strings. In a nutshell, rather than following a top-down approach that starts from template elements, our idea is to elicit the template following a bottom-up approach, starting from the strings that it supposedly generated.

More formally, a *template*, in our definition, is a string composed by *lexical tokens* (e.g., a parenthesis, a function name, an angular bracket), that are common to all the payloads that generated it, and *variable parts*, which we represent with placeholders. The `NUM` placeholders replace strings that contains only digits, whereas the `STR` placeholders

replace strings that contains alphanumeric characters. For instance, the template for the above example is `<ScRiPt>prompt(90_NUM_)</ScRiPt>`

To generate the templates we leveraged the Levenshtein (or edit) distance (i.e., the minimum number of single-character insertions, deletions, or substitutions required to transform string A to string B).

At each recursion, our algorithm receives as an input a list of strings and performs a pairwise comparison (without repetition) between elements of the input list. If the Levenshtein distance between each two compared strings is lower than a fixed threshold, we extract the matching blocks between the two strings (i.e., sequences of characters common to both strings). If the length of all matching blocks is higher than a given threshold, the matches are accepted. Non-matching blocks are then substituted with the corresponding placeholders. The output of each recursion is a list of generated templates. All payloads discarded by the Levenshtein or matching-block thresholding are appended to the list of output templates, to avoid discarding "rarer" payloads (i.e., outliers) and losing useful samples. The thresholds (maximum Levenshtein distance and minimum matching block length) are decremented at each cycle by an oblivion factor, making the algorithm increasingly restrictive. We selected the parameters, including the oblivion factor, through empirical experimentation, by minimizing the number of templates missed. This automatic selection yielded the following values: 20, 0.9 (default case); 20, 0.9 (Acunetix), 15, 0.5 (NetSparker); 15, 0.8 (NTOSpider); 15, 0.9 (Skipfish); 15, 0.5 (W3af). The algorithm stops when a recursion does not yield any new templates.

4.3 Phase 3 (Template Evaluation)

We want to assess the quality of payloads in terms of *filter-evasion* capabilities and amount of *mutations* used by the scanner. Given our observations above, we apply such evaluation to templates, as opposed to each single payload.

More precisely, the quality of a template is expressed by the following template metrics, which we aggregate as defined in Section 5.3. Note that the *rationale* behind each metric is explained on payloads, whereas the metric itself is *calculated* on the templates.

M1 (Length), int: The longer a payload is, the easier to spot and filter (even by accident). Thus, we calculate the length of each payload template to quantify the level of evasion capability.

M2 (Number of distinct characters), int: The presence of particular characters in a payload could hit server-side filters, or trigger logging. The presence of a character instead of another could reveal an attempt to mutate the string (e.g., fuzzing). A symbol can have different meanings depending on the actual context. From this rationale we obtain that a payload with a small set of characters is "better" than one leveraging rare characters. We calculate this metric on the variable part of each template (i.e., excluding the `STR` ad `NUM` tokens).

M3 (Custom callbacks) bool: Rather than using standard JavaScript functions like `alert`, a scanner can use custom JavaScript function callbacks to bypass simple filters. We interpret this as an evasion attempt. If a template contains a function outside the set of built-in JavaScript functions, we set this metric to true.

M4 (Multiple encodings), bool: Encoding a payload may let it pass unnoticed by some web applications' filters. However, some applications do not accept certain encodings, resulting in the application not executing the payload. A payload that uses multiple

encodings is also more general because, in principle, it triggers more state changes in the web application. We set this metric to true if the template contains symbols encoded with a charset other than UTF-8 and URL-percent, thus quantifying the level of evasion.

M5 (Number of known filter-evasion techniques), int: With this metric we quantify the amount of known techniques to avoid filters in web applications. For each template we calculate how many known techniques are used by matching against the OWASP list[2].

Although other metrics could be designed, we believe that these metrics are the bare minimum to characterize a scanner's capabilities and understand more deeply the quality of the payloads that it produces and process.

4.4 Phase 4 (Retrofitting Negative Payloads)

At the end of a scan, each scanner produces a report of the detected vulnerabilities. We use a report-parsing module that we developed (and released) for each scanner, and correlate the results with the payloads extracted. In this way we identify payloads that triggered vulnerabilities, which we call *positive payloads* and those that did not, called *negative payloads*.

We manually verified each negative payload to ensure that it was not our report-parsing module failing to correlate. We found that there are at least four reasons for which a negative payload occur:

- The payload was malformed (e.g., wrong or missing characters, incorrect structure) and it was not executed. This is a functional bug in the scanner.
- The payload was designed for a different context than the one it was mistakenly injected in.
- The scanner used what appears to be the "right" payload for the test case, but the detection engine somehow failed to detect the exploit.
- The payload was redundant (i.e., the scanner already discovered a vulnerability) in the same location thanks to another payload, and thus will not report it again.

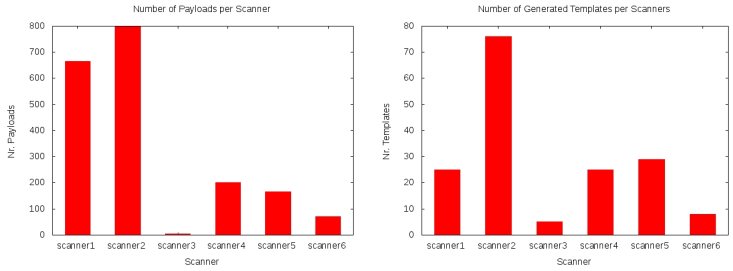
Since one of our goals was to create a testbed application as complete as possible, we wanted to ensure that all negative payloads had a matching test case in our application. With manual analysis, we proceeded to discard malformed and redundant payloads from the list of negative payloads. For each remaining negative payloads we produced a specific vulnerable test case.

To avoid introducing a bias in the results, we crafted each new test case to be triggered exclusively by the payload type for which it has been created, whereas the other payloads of the same scanner are rejected, filtered, or escaped. Of course, nothing would prevent other scanners from detecting the case with a different payload and that was indeed the intended and expected behavior.

5 Experimental Results

We tested 4 commercial scanners (in random order, Acunetix 8.0, NetSparker 3.0.15.0, N-Stalker 10.13.11.28, NTOSpider 6.0.729), for which we obtained dedicated licenses with the support of the vendors, and 2 open-source scanners (in random order, Skipfish

[2] `https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet`



(a) Extracted payloads. (b) Generated templates.

Fig. 1: Output summary of Phase 1 and Phase 2, respectively.

2.10b, and w3af 1.2). Because performing a comparative analysis is *not* the point of this paper, the scanners are weakly anonymized and they appear as `Scanner1`, `Scanner2`, etc., in our results.

We installed each scanner on a dedicated virtual machine (VM) to guarantee reproducibility and isolation (i.e., Debian 6.0.7 for Skipfish and W3af, and Windows 7 Professional for Acunetix, NetSparker, N-Stalker and NTOSpider). We used Wireshark to capture the traffic. When possible, we configured each scanner to only look for XSS vulnerabilities, and to minimize the impact of other variables, we left the configuration to its default values and kept it unchanged throughout all the tests.

We tested Firing Range several times with each scanner. No scanner reported false positives, which is an expected result since we did not design any test cases to trick them like Bau et al. [1] did in their testbed application.

Of course, simply running scanners against a test application and analyzing their reports is not enough to evaluate their payloads. As Doupé et al. [2] did in their study, we wanted to understand the behavior of a scanner in action to be able to explain their results. Our approach, however, differs noticeably since Doupé et al.'s main concern is about the crawling phase of the scanner, whereas we focus on the attack phase, and specifically on the payloads.

5.1 Payload Extraction Results

The number of extracted payloads for all scanners is shown in Fig. 1(a).

Since the detection rate is approximately the same for all scanners (on the first version of Firing Range, before **Phase 4 (Retrofitting Negative Payloads)**), the number of distinct payloads, shown in Fig. 1(a), is interesting: the detection technique of **Scanner 3** uses far fewer payloads, while achieving the same detection of others. The comparatively larger number of payloads observed in the first 2 scanners is due to the use of unique identifiers tied to each of requests. Arguably, these identifiers are used to link a result back to the request that originated it even if server side processing had moved it around—this property is important when detecting stored XSS.

Recommendation (Baseline Payloads) Having a good, strong source of baseline payloads (e.g., from exploit databases[3]) makes a significant difference in terms of payload-to-detection rate. Having a diverse set of distinct payloads is better (although harder) than having a small set of initial payloads used to generate many payloads by automatic mutation or fuzzing. This does not mean that scanner designers should not mutate the baseline payloads to generate new ones. Mutation and fuzzing are fundamental, but should not substitute the research effort by the vendor, who should ensure a rich supply of exploitation payloads whenever new vulnerabilities are discovered.

5.2 Payload Templating Results

The number of payloads alone, however, does not tell much about the actual quality and type of the payloads. More interesting conclusions about the fuzzying algorithm adopted by each scanner can be drawn by comparing Fig 1(a) vs. Fig 1(b). Indeed, after applying the clustering process of Phase 2 (Payload Templating), we notice immediately the limited number of templates (i.e., reflecting the attack patterns), as shown in Fig. 1(b).

The larger number of templates generated for **Scanner 2** is an index of lower efficiency of the scanner, in terms of amount of requests and time spent for detecting a given set of vulnerabilities. In fact, while detecting the very same set of vulnerabilities as the other scanners, **Scanner 2** employs 3–4 times the number of payload *templates* (i.e., distinct exploit patterns).

At this point in the analysis we could already see some scanners emerging as clearly more efficient due to the smaller number of templates they use. For example, **Scanner 2** uses more complex payload templates such as:

```
-STR-`"--></style></script><script>alert(0x0000-STR-_NUM_)
    </script>
```

The templates for **Scanner 4** are numerous and very different from each other, due to the wide variety of generated payloads. The (low number of) templates from **Scanner 5** show that its payloads are significantly different from the rest. The templates that cover most of the payloads are:

```
 -->">'>'"<obf000084v209637>
 .htaccess.aspx-->">'>'"<obf000085v209637>
 .htaccess.aspx-->">'>'"<obf000_NUM_v209637>
 -STR--->">'>'"<obf_NUM_v209637>
```

which capture the scanner developer's particular interest in generic payloads that can highlight the incorrect escaping of a number of special characters at once. This approach is not found in any other scanner.

Scanner 6 created a large number of templates, sign of strong use of non-trivial mutations and fuzzying.

Recommendations (Mutation and Context) From our results we learn that designing good mutation mechanism is not easy to implement. Naïve approaches such as those adopted by **Scanner 1**, which only appends a progressive number to "fuzz" the payload, do not pay back in more vulnerabilities detected. Instead, it is inefficient as it exercises the application with the very same (small) number of payload templates, which are actually all duplicate (except for the progressive number). This can be partially mitigated if the scanner implements an intelligent algorithm that figures out that *N* "very similar

[3] `http://exploit-db.com/search/?action=search&filter_description=cross&filter_platform=0&filter_type=6`

Table 1: Summary of template evaluation.

SCANNER	MUTATIONS (M4)	CALLBACKS (M3)	FILTER EVASION (M2, M4, M5)
1	✓		✓
2	✓	✓	✓
3	✓	✓	✓
4	✓		✓
5			
6	✓		✓

payloads" (e.g., same strings, except for 1–2 characters) are continuously generating the same (negative) result. Setting a limit on this is a simple yet effective technique to keep efficiency under control. Moreover, a contextual parsing approach is recommended to select a candidate subset of potentially successful payloads, *before* attempting the injection. Although having an identifier (e.g., incremental number) for each injected payload is useful (e.g., for matching stored XSS or multiple injection points in a single page), it should not be used alone as a fuzzying mechanism.

5.3 Template Evaluation Results

During this phase we evaluated each of the templates on the metrics defined in Section 4.3. Fig. 2 reports the mean of **M1 (Length)** calculated over the number of templates produced by each scanner. This is an interesting finding, which can be interpreted in two ways. On the one side, the length of the templates is in line with the minimum length of real-world payloads required to exploit XSS vulnerabilities, which is around 30 characters [4, 15], which somehow justifies the choice of the payloads. On the other hand, such a long string may fail to exploit a vulnerable entry point that simply cuts the payload off, even without a proper filter. Although this can be a good way for the scanner to avoid flagging unexploitable vulnerabilities (false positives), it has been shown that multiple small payloads can be combined to generate a full attack [9]. However, the scanners that we examined miss these occurrences.

We notice that **Scanner 1** employs significantly longer payloads than **Scanner 5**. This can be explained, considering that **Scanner 5**'s M5 is zero, meaning that it uses no known filter-evasion techniques: thus, **Scanner 5** is less sophisticated than **Scanner 1**.

Using M2–M5, we derived Table 1, which gives a bird's eye view on the use of mutations, filter evasion and use of callbacks from each scanner. Regarding mutations and callbacks, we use M3 (Custom callbacks) and M4 (Multiple encodings), respectively, whereas for filter evasion, if at least one template has a non empty character set (from M2), uses multiple encodings (from M4), and adopt at least one evasion technique (from M5) we conclude that the scanner performs filter evasion.

As it can be seen, both random mutations and filter-evasion techniques are widely employed in the scanners that we tested. Nevertheless, these techniques are ineffective at triggering all the vulnerable entry points. In fact, most of them yield poor-quality payloads, as detailed in Section 5.4. Instead, the use of custom callbacks over parsing or standard JavaScript functions is not common among the scanners that we tested.

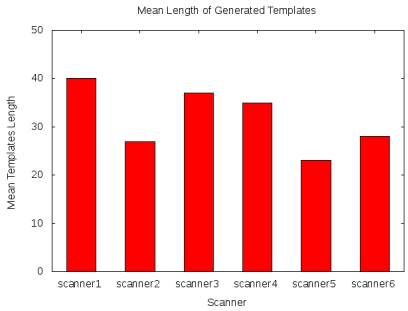


Fig. 2: Mean M1 (Length) over the templates of each scanner.

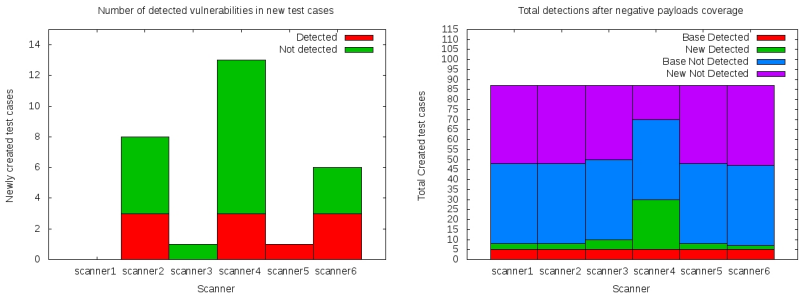
Recommendations (Payload Quality Ranking) This section answers one of the question that we stated in the introduction of this paper, which is actually one of the questions that motivated us to pursue this work: Can we rank [them] and identify a set of "better" payloads? Although the results of our experiments can be used to rank the scanners based, for example, on the quality metrics, unfortunately none of the scanners that we reviewed offer feedback to the analyst regarding the "quality" of the payloads that successfully triggered vulnerabilities. Although we can assume that it always possible for an attacker to send an arbitrarily crafted URL to the victim, certain payloads are more likely to pass undetected by network- or application-level protections (e.g., ModSecurity), which in case of legacy web applications are the only viable alternative to patching. Therefore, a successful payload with, say, a small character set, which adopts filter-evasion techniques while keeping a short length overall, should be ranked as "high risk". This additional feedback could be part of the typical vulnerability-ranking part that some vendors include in their reports.

5.4 Retrofitting Negative Payloads Results

XSS PEEKER's workflow iteratively adds new test cases in our testbed to account for payloads that triggered no vulnerabilities (i.e., negative payloads). This section answers one of the questions that we stated in the introduction, that is: "Is a scanner exercising all the entry points with the expected payloads?" The expected result was the reduction of the number of negative payloads to zero. So, we ran all the scanners against the new testbed and analyzed the results (note that the payloads of **Scanner 1** were all already covered by our initial test case, thus we created no additional cases).

Unfortunately, as Fig. 3(a) shows, scanners failed to detect most of the new vulnerabilities. The most surprising finding is that the very same scanners that generated a negative payload would still fail to detect and report the new test case introduced for it, even if we manually confirmed that all of the negative payloads do in fact trigger an XSS on their respective new cases.

This may indicate a bugged response-parsing routine (i.e. a functional bug in the scanner). In other cases, by manually analyzing the requests toward the new cases, we discovered that some scanners did not exercise the test case with the "right" payload: a faulty (or random) payload-selection procedure somehow failed to choose it, using it instead in test cases where it would turn out to be ineffective; a different bug class in scanners. Another interesting result is that, after the introduction of the new cases, some



(a) "New" test cases alone. (b) Overall comparison.

Fig. 3: Summary of final results after Phase 4. "New", "Base" (initial test cases), "Detected" (true positive), and "Not Detected" (false negative).

scanners started using payloads we had not observed before. This behavior suggests some degree of context awareness, as scanners would only generate this new set of templates after having observed these new contexts. However, even in this case we observed a staggering high rate of failures for the new corner cases we added.

These findings would have been very difficult to reveal with a static set of vulnerabilities, as opposed to the incremental approach that we adopted. Fig. 3(b) shows the overall results produced by each scanner after including the new test cases. Although scanners did not achieve the expected results, this process allowed us to greatly increase our coverage of test cases for attacks supported by the analyzed scanners, and to produce a state of the art testbed for future work.

Recommendations (Continuous Testing) Although scanner vendors take testing very seriously, it is not trivial to account for the side-effect caused by adding new vulnerability cases to the testbed web applications. Adopting a retrofitting approach similar to the one that we implemented could be a first step toward finding corner cases (e.g., exercising a vulnerability with an incorrect payload) or bugs similar to the ones that we discovered.

6 Conclusions

This vertical study on XSS vulnerability scanners proposes quality metrics of 6 commercial and open-source products through passive reverse engineering of their testing phases, and manual and automated analysis of their payloads. Furthermore, we created a reusable and publicly available testbed.

By iterating on payloads that triggered no test cases, we were able to noticeably improve our test application and draw important conclusions about each scanner's inside workings. One of the key results is that, despite having some kind of awareness about context, all of the tested scanners were found wanting in terms of selecting the attack payloads and optimizing the number of requests produced. A significant number of detection failures suggest bugs and instability in the detection engines, while the high variance in types and features of the payloads we inspected makes the case for cooperation in defining common, efficient and reliable payloads and detection techniques.

References

[1] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell. State of the art: Automated black-box web application vulnerability testing. In *IEEE SSP*, pages 332–345, May 2010. doi: 10.1109/SP.2010.27.

[2] A. Doupé, M. Cova, and G. Vigna. Why johnny can't pentest: An analysis of black-box web vulnerability scanners. In *DIMVA*, pages 111–131, Berlin, Heidelberg, 2010. Springer-Verlag.

[3] Foundstone. Hacme Bank v2.0, 2006. URL `http://www.foundstone.com/us/resources/proddesc/hacmebank.html`.

[4] Gnarlysec. XSS and ultra short URLs, 2010. URL `http://gnarlysec.blogspot.ch/2010/01/xss-and-ultra-short-urls.html`.

[5] M. Heiderich, J. Schwenk, T. Frosch, J. Magazinius, and E. Z. Yang. mXSS attacks: Attacking well-secured web-applications by using innerHTML mutations. In *CCS*, pages 777–788. ACM, 2013.

[6] A. Klein. DOM based cross site scripting or XSS of the third kind, 2005. URL `http://www.webappsec.org/projects/articles/071105.shtml`.

[7] S. Lekies, B. Stock, and M. Johns. 25 million flows later: Large-scale detection of dom-based xss. In *CCS*, pages 1193–1204. ACM, 2013.

[8] F. Maggi, A. Frossi, S. Zanero, G. Stringhini, B. Stone-Gross, C. Kruegel, and G. Vigna. Two years of short urls internet measurement: Security threats and countermeasures. In *WWW*, pages 861–872, 2013.

[9] P. Mutton. XSS in confined spaces, 2011. URL `http://www.highseverity.com/2011/06/xss-in-confined-spaces.html`.

[10] Open Web Application Security Project. Top ten, 2013. URL `https://www.owasp.org/index.php/Top_10_2013-Top_10`.

[11] A. Pitsillidis, K. Levchenko, C. Kreibich, C. Kanich, G. M. Voelker, V. Paxson, N. Weaver, and S. Savage. Botnet judo: Fighting spam with itself. In *DNSS*, San Diego, California, USA, March 2010. The Internet Society.

[12] M. Samuel, P. Saxena, and D. Song. Context-sensitive auto-sanitization in web templating languages using type qualifiers. In *CCS*, pages 587–600. ACM, 2011.

[13] D. SecureWorks. Dell SecureWorks Threat Report for 2012, 2012. URL `http://www.secureworks.com/cyber-threat-intelligence/threats/2012-threat-reviews`.

[14] L. Suto. Analyzing the accuracy and time costs of web application security scanners, 2010. URL `http://www.ntobjectives.com/files/Accuracy_and_Time_Costs_of_Web_App_Scanners.pdf`.

[15] B. Toews. XSS shortening cheatsheet, 2012. URL `http://labs.neohapsis.com/2012/04/19/xss-shortening-cheatsheet`.

[16] J. Tudor. Web Application Vulnerability Statistics 2013. available online at `http://www.contextis.com/files/Web_Application_Vulnerability_Statistics_-_June_2013.pdf`, June 2013.

[17] M. Vieira, N. Antunes, and H. Madeira. Using web security scanners to detect vulnerabilities in web services. In *IEEE/IFIP DSN*, pages 566–571, June 2009.

[18] I. X-Force. IBM X-Force 2013 Mid-Year Trend and Risk Report, 2013. URL `http://securityintelligence.com/cyber-attacks-research-reveals-top-tactics-xforce`.