XSSDS: Server-side Detection of Cross-site Scripting Attacks

Martin Johns[1], Björn Engelmann[2], and Joachim Posegga[1]

[1]University of Passau, ISL
Innstr. 43, D-94032 Passau, Germany
(martin.johns|posegga)@uni-passau.de

[2] University of Hamburg, SVS
Vogt-Koelln-Str. 30, D-22527 Hamburg, Germany
bjoern@noxss.org

Abstract

Cross-site Scripting (XSS) has emerged to one of the most prevalent type of security vulnerabilities. While the reason for the vulnerability primarily lies on the server-side, the actual exploitation is within the victim's web browser on the client-side. Therefore, an operator of a web application has only very limited evidence of XSS issues. In this paper, we propose a passive detection system to identify successful XSS attacks. Based on a prototypical implementation, we examine our approach's accuracy and verify its detection capabilities. We compiled a data-set of 500.000 individual HTTP request/response-pairs from 95 popular web applications for this, in combination with both real word and manually crafted XSS-exploits; our detection approach results in a total of zero false negatives for all tests, while maintaining an excellent false positive rate for more than 80% of the examined web applications.

1. Introduction

Cross-site Scripting (XSS) in web applications emerged in the last years to one of the most prevalent types of security vulnerabilities [3]. Unlike related problems, such as SQL injection, XSS attacks do not affect the server-side but clients: The actual exploitation is within the victim's web browser. Therefore, the operator of a web application has only very limited evidence of successful XSS attacks. XSS-releated problems are therefore often overlooked or recognized rather late.

This paper proposes a server-side Cross-site Scripting Detection System (XSSDS); our approach is is based on passive HTTP traffic monitoring (cf. Fig. 1) and relies upon the following observations:

- There is a strong correlation between incoming parameters and reflected XSS issues.
- The set of all legitimate JavaScripts in a given web application is bounded.

This forms the basis for two novel detection approaches to identify successfully carried out reflected XSS attacks (Sec. 3) and to discover stored XSS code (Sec. 4). Our approach does not require any changes to the actual application or infrastructure: Both attack detection methods depend solely on access to the HTTP traffic. Our approach is therefore applicable to all current web application technologies, i.e., programming languages, web servers, and applications.

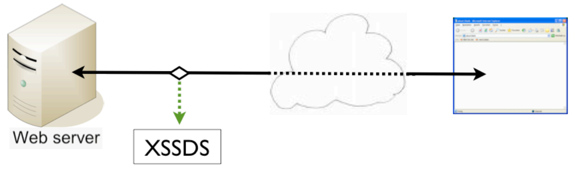

Figure 1. Passive XSS attack detection

2. Technical background: Cross-site Scripting

The term Cross-site Scripting (XSS) [11] denotes a class of string-based code injection attacks on web applications. If a web application implements insufficient input validation and/or output sanitation an adversary might be able to inject arbitrary script code into the application's HTML encoding. A successful XSS attack can lead to, e.g., compromized authentication information, privilege escalation, or disclosure of confidential data. In the context of this paper we will concentrate on injected JavaScript-code. However, our methods are also applicable to other client-side scripting languages, like VBScript. XSS can be classified in three different kinds: reflected, stored and DOM-based XSS:

- **Reflected XSS** denotes all non-persistent XSS issues, which occur when a web application blindly echos parts of the HTTP request in the corresponding HTTP response's HTML (see Listing 1). For successfully exploiting a reflected XSS vulnerability, the adversary

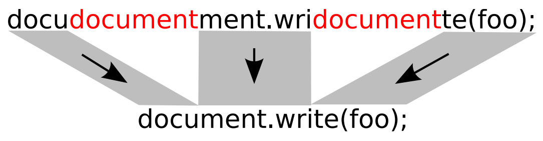

Figure 2. Dysfunctional removal filter removing the keyword "document"

has to trick the victim into sending a fabricated HTTP request. This can, for instance, be done by sending the victim a malicious link, or by including a hidden Iframe into page controlled by an attacker.

- **Stored XSS** refers to all XSS vulnerabilities, where the adversary is able to permanently inject the malicious script in the vulnerable application's storage. This results in every user that accesses the poisoned web page receiving the injected script without further actions by the adversary.

- **DOM-based XSS** [12] is a special variant of reflected XSS, where logic errors in legitimate JavaScript and careless usage of client-side data result in XSS conditions. N.B.: Since in this case the offending data exists solely in the browser and is not sent to the server, such vulnerabilities are out of scope of of this paper.

```
$name = $_GET['name'];
echo "Hallo " + $name + "!";
```

Listing 1. Reflected XSS through direct data-inclusion

Current defence strategies: There are largely two distinct countermeasures for XSS prevention in "real-life" web applications: Input filtering and output sanitation. *Input filtering* describes the process of validating all incoming data. "Suspicious" input that might contain a code injection payload is either rejected, encoded, or the "offensive" parts are removed using so called "removal filters". The protection approach implemented by these filters relies on removing preddefined keywords, such as `<script`, `javascript`, or `document`. Such filtering approaches are, however, error-prone due to incomplete keyword-lists or non-recursive implementations (see [2] and Fig. 2). If *output sanitation* is employed, certain characters, such as <, ", or ', are HTML encoded before user-supplied data is inserted into the outgoing HTML. As long as all untrusted data is "disarmed" this way, XSS can be prevented. Both of the above protections are known to frequently fail [3], either through erroneous implementation, or because they are not applied to the complete set of user-supplied data.

3. Reflected XSS detection by request/response matching

3.1. Overview

Our detection mechanism for reflected XSS is based on the observation that reflected XSS implies a direct relationship between the input data (e.g., HTML parameters) and the injected script. More precisely: The injected script is fully contained both in the HTTP request and the HTTP response. Reflected XSS should therefore be detectable by simply matching incoming data and outgoing JavaScript using an appropriate similarity metric. It is crucial to emphasise that we match the incoming data only against *script code* found in HTML. Non-script HTML content is ignored. See Section 5 for our script-extraction technique.

For the sake of readability we will uses the term *parameters* as a generalized term for all user-provided data in the sequel.

We can formulate the problem to be solved as follows:

Problem 1 *Given a set of parameters $P = \{p_1, p_2, ..., p_m\}$ and a set of scripts $S = \{s_1, s_2, ..., s_n\}$ find all matches between P and S in which p_i was used to define parts of s_j.*

3.2. Normalization

External data used by web applications is often repeatedly de- and encoded during HTML generation (see Fig. 3 and Sec. 2). Input-data and the data used within the HTML can therefore differ even in the case of direct data-inclusion XSS (see Listing 1). For instance, an application might initially remove all URL-encoding from the input data, and HTML-encode certain characters before inserting the result into the webpage content being generated. Note that some of the decoding steps are not necessarily executed by the actual application, but could as well be performed by the web server. Figure 5 lists all relevant encodings we could identify for this particular problem domain.

Obviously, we have to "normalize" both strings in order to match the parameters with the scripts despite their different encodings. We therefore transform both the incoming parameters, as well as the scripts found in the final HTML, into a non-ambiguous representation by iteratively removing all encodings (see Fig. 4). Note, that this algorithm al-

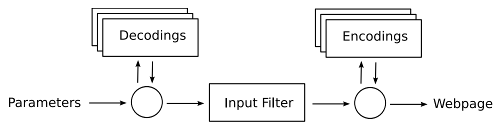

Figure 3. Parameter de- and encoding

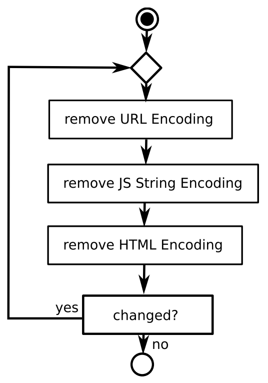

Figure 4. Algorithm for encoding removal

ways terminates for finite strings, since each decoding step shortens the string.

3.3. Simple matching

The outlined normalization-step is sufficient to ensure reliable detection for web applications which employ input-filters that never remove parts of the parameter-string: Matching parameters against scripts can here be carried out with a variant of a standard Longest Common Substring (LCS) Algorithm with a certain length-threshold t (we would advice $t = 15$, as explained below). This would find all matches with $\geq t$ characters in time linear to the combined length of the scripts + the combined length of the parameters.

3.4. Robustness against removal filters

In the case of a removal-filter, or in environments where the nature of the filter is unknown, we have to deal with the fact that some parts of the parameter may not occur in the script. Therefore, a straight forward LCS matching algorithm is insufficient: The input data may contain substrings that are not present in the corresponding script. Our refined problem can be formalized as follows:

Problem 2 *Given a set of parameters $P = \{p_1, p_2, ..., p_m\}$ and a set of scripts $S = \{s_1, s_2, ..., s_n\}$, find all strings x that*

1. $\exists p \in P : x$ *is subsequence of* p
2. $\exists s \in S : x$ *is substring of* s
3. $|x| \geq t$ *with* $t \in \mathbb{N}$ *being some threshold*

Subsequences are, contrary to substrings, allowed to skip over an arbitrary number of characters. Thus, this notion allows parts of the substring to be present in the parameter while missing in the script. This will unfortunately increase the complexity of our matching algorithm.

Name	Example
HTML character entity references	$<\rightarrow$ `&lt;`
HTML decimal character references	$<\rightarrow$ `&#60;`
HTML hexadecimal character references	$<\rightarrow$ `&#x3C;`
JavaScript single escape character	$<\rightarrow$ `\<`
JavaScript hex escape sequence	$<\rightarrow$ `\x3C`
JavaScript unicode escape sequence	$<\rightarrow$ `\u003C`
URL encoding	$<\rightarrow$ `%3C`

Figure 5. Relevant character encodings

The set of all subsequences of any parameter-string p can be expressed as a deterministic finite automaton (DFA) as shown in the following definition. The example D_{cocoa} is shown in Figure 6.

Definition 1 *Given a string $p = p_1 p_2 ... p_n \in \Sigma^*$, the DFA $D_p = (Q, \Sigma, \delta, s_0, F) \in DFA$ with*

$$\begin{aligned}
&Q = \{s_0, s_1, ..., s_n\} && \text{(states)}\\
&\delta : Q \times \Sigma \mapsto Q : \delta(s_i, w_j) = && \text{(transitions)}\\
&\quad s_j \leftrightarrow i < j \wedge [\nexists j' : i < j' < j \wedge p_j = p_{j'}] &&\\
&F = Q && \text{(final states)}\\
&s_0 \in Q && \text{(starting state)}\\
&\Sigma && \text{(alphabet)}
\end{aligned}$$

accepts exactly the set of all subsequences of p.

Our problem is therefore equivalent to creating such a DFA for every parameter, and letting them match the set of all substrings of all scripts and filtering out all matches with length $< t$. As most LCS-Algorithms do, we use Generalized Suffix Trees to efficiently search the set of all substrings of all scripts. We are able to construct this structure in time linear in respect to the combined length of the scripts with the algorithm by Esko Ukkonen [23] and an adaption by Dan Gusfield [5]. A method proposed by Baeza-Yates & Gonnet [1] is then applied to match each parameter's DFA over this tree, while ignoring all matches shorter than t. The overall time complexity of finding all L

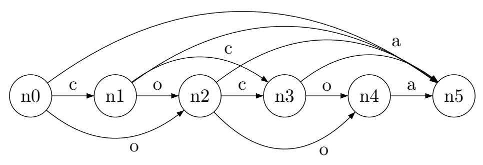

Figure 6. An example DFA accepting all subsequences of the word "cocoa". All states are final.

occurrences of subsequences with length $\geq t$ of the parameters $p_1, p_2, ..., p_m$ with combined length M in the scripts $s_1, s_2, ..., s_n$ with combined length N using this algorithm is: $O(m * N * t + M + L)$.

In our practical evaluation (see Sec. 6) we used a threshold $t = 15$. This choice is based on the assumption that no reasonably complex malicious script will be shorter than 15 characters. We will illustrate this by some examples:

`location=http://`	— 16 chars
`document.write()`	— 16 chars
`document.cookies`	— 17 chars

3.5. Avoiding false positives

Although our algorithm is able to find all relevant parameter-script matches, a naive implementation may produce a number of unneccessary false-positives: Data from parameters is often inserted into a script as a string constant. The name of a variable or function might also coincide with a parameter (they are English words, after all). All these cases, however, can be captured with a JavaScript tokenizer on the script, and by filtering all matches involving only a single token. This will not introduce any false-negatives, as no reasonable complex malicious script can possibly be written consisting only of a single token; the exception to this rule are string constants for dynamic code generation, we will discuss this case in Section 5.1).

3.6. External scripts

Unlike direct script injection attacks, where the actual script is contained in the parameters, external script inclusion relies on the attacker's ability to inject `script`-tags including a URL to an attacker controlled location.

```
<script src="http://a.org/e.js"></script>
```

Listing 2. External script inclusion attack

For external scripts, we are formulating a second policy: *An alarm is raised if the above algorithm finds a match with* ≥ 10 *characters in length and with the entire script's URL being a subsequence of the parameter.* It is based on two assumptions:

- No absolute URL can be shorter than 10 characters: The mandatory `http://` consumes 7, and no regular domain shorter than 3 characters can be set up.
- An adversary has to control at least the prefix of the URL to take advantage of the script. In web applications it is much more likely that the prefix is fixed and the suffix is controlled by a parameter than the other way around. In practically all exploitable cases the adversary controls the entire URL.

3.7. Limitation

The proposed detector relies upon a direct comparison of incoming HTTP-parameters and outgoing HTML. Stored XSS is therfore not always detectable with this approach: the required direct relationship between HTTP request and response does not necessarily exist. It might be possible to detect the initial exploiting request/response pair, if the given stored XSS takes effect immediately. However, in certain cases, the HTTP request that injects the malicious payload permanently in the application and the poisoned HTML response are not created consecutively.

4. Generic XSS detection using a list of known scripts

4.1. Overview

The detection approach presented in Section 3 cannot reliably detect stored XSS; we propose a second, generic XSS detector to address this shortcoming in this section. The set of all employed JavaScripts of a given web application is bounded. There might be variations within these script's code and position (see below), but the absolute number of unique scripts is necessarily limited by the number of server-side code-fragments in the actual application which inserts JavaScript-code into the final HTML, and the number of possible script templates used by such fragments.

Thus, we can build a mechanism to keep track of all used JavaScripts within a given web application and detect variances in the application's script-set. Such a variance is a strong indicator for either an update of the actual web application's code (which should be known by the application's administrator) or a successful XSS attack. We do propose to use a training based XSS detector: After a training phase, which consists of building up the detector's list of known scripts, all outgoing scripts are checked against this list. If an unlisted script is encountered, the detector raises an alarm.

4.2. Script types

Whilst this concept is appealingly simple, scripts do vary both in code and location within a webpage in reality. We therefore need a representation in which the script-set can be considered static for most dynamic web applications and distinguish between two general types of scripts:

1. *Static scripts*: JavaScripts that never change. Such scripts are usually contained in the application's HTML templates or they are part of supporting JavaScript libraries.

2. *Dynamic scripts*: Such JavaScripts are composed during runtime; i.e.: the state of the application or the submitted parameters influence the script's final code. In many cases input parameters are used as the value of string-constants within dynamic scripts (see Listing 3).

While training our detector to recognise static scripts is straightforward, dynamic scripts require more attention.

4.3. Dealing with dynamic scripts

In a manual analysis of dynamic scripts, we were able to identify two subtypes: Firstly dynamic scripts that solely vary in the values of constants, and second dynamic scripts that also expose variations in the script's actual code.

Dynamic scripts with varying constants: We often observed a striking resemblance in the code of different occurrences of the same dynamic script. The code was in most cases completely static apart from its string constants (as exemplified in Listing 3). As constants are often used to pass data to scripts, our representation should also allow them to change.

```
echo "<script>";
echo "alert('Hello " + $name + "!');";
echo "</script>";
```

Listing 3. Example of a dynamic script

This is taken into account by tokenizing each script and substituting all constant-tokens by generic representations. The string-tokens thus are substituted by `STRING`, numeric constants by `NUMERIC` and regular expressions by `REGEXP`. The resulting scripts are static for all possible variants and therefore usable for our matching algorithm (see Listing 4).

```
<script>
alert(STRING);
</script>
```

Listing 4. Normalized version of Listing 3

Dynamic scripts with varying code blocks: The second subclass of dynamic scripts we encountered consisted of JavaScript-blocks which also revealed a variance in the script's code portion. While each of these respective scripts were apparently generated by the same source (indicated through position, purpose, and form of the JavaScript), they contained at least one dynamic code block. An analysis of these scripts revealed two variants of this particular script-type:

- *Code repetition*: These JavaScripts included blocks of repeated code which apparently was created by loops. (see Listing 5).

- *Selective code omission:* In some cases –depending on the state of the web application– a given JavaScript can contain or omit blocks of script code. E.g., according to the authorizations of the logged-in user, a dynamic JavaScript-driven dialogue might contain varying options.

```
<script>
a[1] = "value 1";
a[2] = "value 2";
...
a[10] = "value 10";
...
</script>
```

Listing 5. Repeated code in dependence on the number of values

Each of these cases is not a problem for our approach. As the number of possible variants of such scripts in tokenized form is bounded, we treat every possible variant as an individual script; thus, it is added to the list of known scripts. This solution increases the total number of known scripts only and does not introduce further problems. However, in future work we intend to investigate also methods for detecting and normalizing cases of code repetition.

4.4. Training

As explained above, the detector's training phase consists of building a list of known JavaScripts. Besides the actual, tokenized script-code we also recorded the type of script (i.e.: inline script, event handler, or external script), its position within the HTML page (`head` or `body`), and, if applicable, the event-name (for event handlers). The success of this training phase depends on the degree of application coverage which is achieved: note, that the detector is only able to analyse HTML output which was generated by actual usage of the given application. We do, however, not consider this as a drawback: Our detector is positioned at the server side during actual use. Therefore, it receives the complete HTML output of all usages of the application. Even with large applications, it is reasonable to assume that every major variation of every possible script is included in at least one visited page. As every script has to be seen only once by the detector, the training phase can be completed in reasonable time (see Sec. 6 for our practical evaluation of this). To ensure that the training-set is in fact attack-free, the actual training can be done during the application development, quality assurance testing, and, if applicable, the closed beta-phase (a common practice introduced by Web2.0 applications).

External scripts: External scripts need to be handled differently, since the actual code of these scripts is not processed by the detector. We build a list of known domains, containing the top-level- and first-level domain name as well as the protocol and port using the URLs of external scripts encountered during the training phase. Afterwards, external scripts are legitimate if and only if they originate from one of these domains. This policy allows the use of an external script-server under a different domain (as practiced by many, e.g. eBay). However, we assume that an adversary is not able to place a malicious script on one of the trusted servers.

Extending the list of legitimate scripts after application updates: Some web applications are rapidly updated without extensive internal testing. If in such a case the application's script-set is altered, the detector's list of known scripts has to be extended during actual production use. To ensure that the list remains attack-free, it is advisable that every addition to the list is manually confirmed. This is feasible since after applying our normalization and tokenization steps the absolute number of known scripts will be rather limited. During our practical evaluation (see Sec. 6) we observed even for large applications an average script-set size between 50 and 300 scripts. Under the assumption that any given update touches approximately 10% of all scripts, a total of 5 to 30 new entries would have to be manually confirmed. Furthermore, during such phases our other detection approach continues to work, so it will presumably detect all reflected XSS flaws before they may enter the list of learned scripts.

4.5. Detection

After the training phase has been completed, the detection of unknown scripts consists solely of verifying that a given script is not contained in the detectors training-set. Our test implementation (see Sec. 5) achieved this with simple hash-table lookups using the tokenized scripts as hashkeys. Moreover, it is verified that the other recorded characteristics, such as script type and position, also match. As explained above, every unknown script then indicates either a successful XSS attack, or an update of the application. The latter case has to be handled by the web application's operator: The operator must either restart the detector's training period, or manually confirm each addition to the trained list as described above.

5. Implementation

We successfully implemented prototypes for both our detection approaches. Besides the implementation of the core algorithms, we encountered two tasks that required special attention: Script extraction and script parsing.

5.1. Reliable script extraction

It is crucial to the success of both our detection approaches that a given implementation can reliably identify all JavaScripts in a web page. As web browsers tend to implement a forgiving policy towards rendering illegitimate HTML, deterministically identifying all scripts is a very non-trivial task (see [8]). Therefore, instead of manually implementing an HTML parser, we modified the HTML rendering engine of the popular open-source browser Mozilla Firefox. This ensures that the set of found scripts exactly matches the scripts that would be executed by the browser.

Client-side code creation: JavaScript provides several techniques for on-the-fly creation of executable script code from string values (e.g., using `eval`). This can result in XSS vulnerabilities if an adversary is able to inject code into the respective string constants (see Listing 6). In such cases, the injected code is contained in a string-constant of an otherwise legitimate script. As such string-constants are ignored by both approaches because of the respective tokenization steps (see Sec. 3.5 and 4.3), the script-extraction step has to pay special attention to encountered string-constants: All found JavaScript string-constants are examined for valid script code using our JavaScript-parser. Every JavaScript that is found this way is added as a separate, individual script to the result-set of the extraction component.

```
// XSS condition if the attacker controls 'some_var'
eval(some_var);
```

Listing 6. Dynamic client-side code creation

Our implementation was tested on a number of tricky and obfuscated HTML-based attack vectors from [8], and it proofed to be able to reliably detect all of them. It should be noted though, that our current implementation does not detect scripts in any other content than HTML. However, this is not a conceptual shortcoming, but it is due to the proof-of-concept nature of our implementation: It can easily be extended by other parsers (Flash, SVG, etc.).

As some XSS attack vectors are browser-specific, securing a web application should ideally involve one instance of each supported browser for parsing and merge the resulting lists based on the script's location within the page. We leave this task as future work.

5.2. Script parsing

Both of our detectors rely upon a JavaScript tokenizer for preprocessing. Our implementation was written in Ruby, so we opted for rbNarcissus [22], a Ruby port of the Mozilla Project's JavaScript parser. The tokenization step in the generic XSS detector, however, turned out to be the most significant performance bottleneck. So we strongly suggest that any practical implementation uses a solution optimized for performance.

6. Evaluation

6.1. Dataset and experiment set-up

First, we used a crawler to collect about 500.000 HTTP-request/response pairs from 95 popular web applications, thus simulating web application usage. This dataset includes high profile sites like `myspace.com`, `blogger.com`, and `news.google.com`; the crawling process was manually optimized to ensure page diversity. This data was used as a basis for evaluating our approach; the actual evaluation was done offline, by passing the dataset's contents and the attack data (see below) to our PoC implementation.

6.2. Methodology

Generally, any attack detection system should have two major capabilities: detecting as many attacks as possible whilst having a false-positive rate as low as possible. We assessed the detection abilities of both approaches by applying them to

- crafted attacks injected into otherwise benign data and
- real-world attack data of disclosed XSS problems.

Furthermore, we measured the false-positive rate by applying the detectors to our collected dataset which we assumed contained no attacks. This time every alarm was counted as a false-positive and logged. Afterwards all alarms were reviewed by hand to make sure there really were no attacks in the data. The error-rates were divided by the number of pages used for testing in order to remove the influence of different web application sizes.

6.3. Generic XSS detector evaluation

Since our generic XSS detector needs to be trained for meaningful results, we will use a common statistical practice called k-fold cross-validation (for $k = 10$): The data is partitioned into $k = 10$ subsamples, $k - 1 = 9$ of which are used for training and the one remaining sample is retained for testing. This process is repeated a total of k=10 times, retaining each of the samples exactly once.

Detection evaluation (manual script injection): As malicious scripts we used a common credential stealing attack (see Listing 7) as well as an obfuscated version using a method posted by a member called ".mario" in the sla.ckers forum [16] (see Listing 8). Both variants were injected

- as inline scripts (`<script>...<\script>`),
- as event handlers into random suiting HTML-nodes (`onload="..."`), and
- into existing scripts either directly behind string constants or appended at the end.

External scripts (e.g., `http://hack0r.net/xss.js`) were used as well, yielding a satisfactory number of combinations spanning all major types of XSS occurrences. We relied upon the cross-validation-method for every application in our dataset. During the training phase, every script found in the training-samples was added to the detector's list of legitimate scripts. While testing, a random number (1-5) of randomly picked malicious scripts was injected into each page in the test-sample before applying the detector. The number of injected scripts not causing an alarm indicates false-negatives.

```
c = escape(document.cookie);
document.write("<img src='http://ha.cx/xss?c="+c+"'>");
```

Listing 7. Cookie stealing XSS payload

```
top/**/['\x65\x76\x61\x6c']/**/('\x73\x74\x72\x20\x65
[...]
\x73\x74\x72\x20\x2b\x20\x22\x27\x3e\x22\x20\x29\x3b')
```

Listing 8. Obfuscated JavaScript [16]

Detection evaluation (real-world vulnerabilities): Since we wanted to gain insight into the practical usefullness of our detection approach, we set up three different web applications, each in a version known to be vulnerable to stored XSS attacks ([2], [19], [13]). We trained our generic XSS detector with non-compromised versions of the vulnerable pages, before we exploited each vulnerability to inject some malicious script. Afterwards the detector was applied to the now infected pages and every injected script not causing an alarm was counted as a false-negative.

False-positives: To measure the false-positive-rate, we again used the cross-validation-method. This time, unaltered pages were used both for training and for testing. Every alarm during the testing-phase was counted as a false-positive in the respective web application. The false-positive-rate was divided by the number of pages in the respective test-sample and the alarms were reviewed for real attacks.

Results: The generic XSS detector was able to reliably identify all injected attacks as well as the real-life examples, resulting in a false-negative rate of 0 for all tested web applications. Furthermore, about 80% of the web applications also did not encounter any false-positives. These web applications either solely utilized static JavaScripts or employed dynamic code generation techniques which are recognised by our detector (see Sec. 4.3). The remaining 20% caused varying amounts of false alarms (see Fig. 7) due to non-trivial, dynamic code-generation techniques which are not yet handled by our detector, such as dynamic generation of variable names. However, most of these issues can be resolved easily by a custom normalization step (e.g. by mapping dynamic variable names to a special token type). Rare cases (0.2%) caused a considerable amount of false alarms. The false-positive-rate in such cases was about 7 alarms per 10 pages. Thus, this detector is (without adjusting it to the application) only useful for about 80-85% of the web applications.

6.4. Reflected XSS detector evaluation

Since the reflected XSS detector does not need to be trained, evaluation is relatively straightforward.

Detection evaluation (manual script injection): Our evaluation of the reflected XSS detector is again based on the collected dataset described earlier; our set of malicious scripts discussed above was also used in this case. We randomly selected 100 request/response-pairs from each web application. For each pair, a randomly picked malicious script was encoded using a randomly selected encoding; then it was injected into one of the parameters while the unencoded version was injected into a randomly picked (but suitable) location on the corresponding webpage.

Afterwards, the reflected detector was used on that request-response pair and every injected script not raising an alarm was counted as a false-negative. This procedure was repeated 10 times for each request/response-pair.

Detection evaluation (real-world vulnerabilities): Disclosed reflected XSS vulnerabilities can be found on the net in large numbers. Since we needed the complete request/response-pair, we used 10 different vulnerabilities disclosed on [4] with malicious payloads from our list and we recorded the required HTTP-traffic. The detector was in turn applied to each request/response pair and every injected script not causing an alarm counted as a false-negative.

False-positives: For measuring the false-positive-rate, we applied the detector to every request/response-pair of every web application in our dataset. We again assumed that there are no attacks in the data, so we assumed each alarm to be a false-positive and reviewed these afterwards. The false-positive-rate was here divided by the total number of pages.

Results: As with the generic XSS detector, the detector was able to find each and every given malicious script, so we encountered zero false-negatives. With the reflected XSS detector, about 95% of the web applications did not cause any false-positives at all; the worst case (which is only encountered in approximately 1% of the cases), lies at about 5 alarms per 100 pages. The results of our measurements are depicted in Figure 7.

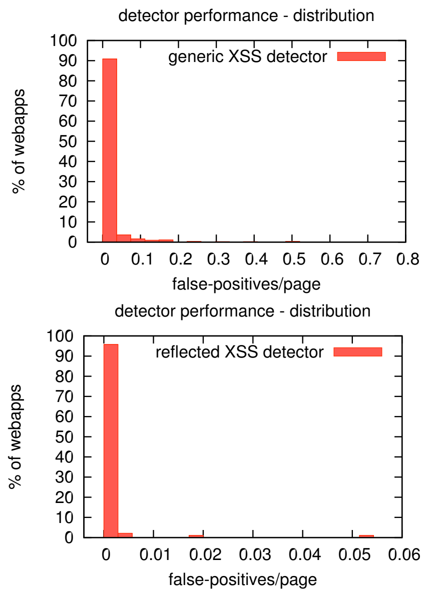

Figure 7. False-positives of the detectors.

7. Discussion

7.1. Combination of the detectors

While exposing excellent abilities in attack detection and a promising "noise level", both proposed detectors have certain, individual advantages and disadvantages: The approach to identify reflected XSS (see Sec. 3) is immediately usable without any training overhead and exposes very few false positives. However, due to its incomplete capabilities in the detection of stored XSS, potential protection holes remain when solely relying on this detection approach. The generic detector (Section 4) exhibits a slightly higher number of false positives and requires a training phase. Our experiments also suggest that there could be a fraction of web applications that employ methods of dynamic JavaScript generation which is incompatible with our proposed method. However, its ability to reliably detect stored XSS makes the approach a suitable companion for the reflected detector. Thus, a combined usage of both detectors is advisable. More precisely: We suggest to implement the generic detector as primary sensor, while the reflected detector can, due to its low false positive rate, aid prioritizing alarms for manual review; even more significantly, it will also help to ensure that the generic's list of known scripts remains attack-free during retraining phases.

7.2. Capabilities of the adversary

In this section we discuss a few possible attack vectors against our approach.

Reflected XSS detector: Due to the nature of our matching algorithm, an adversary is able to reliably create a large number of false positives: The trick is to create HTTP requests which contain parameters filled with substrings of the application's legitimate JavaScripts. As these parameters can be found within the response's scripts, the reflected XSS detector triggers an alarm. However, such attempts can easily be identified by a quick inspection of the offending HTTP parameters; thus, notifying the site's operator that somebody deliberately tries to cause false alarms is easily possible. Furthermore, as the JavaScripts are legitimate, they should be contained in the generic detector's list of known scripts. Therefore, in a combined usage scenario such false alarms could be suppressed.

Generic XSS detector: It should be noted, that the generic detector's algorithm allows an adversary to re-inject any known script already existing in the web application with altered constants. Therefore, it is possible to construct a scenario in which a sophisticated combination of several legitimate scripts enables the adversary to fulfill his objectives. We have not yet encountered an application with a script-set that would allow such an attack and consider this case to be highly unlikely and rather contrived – however, it remains as a possible attack.

8. Related work

Server-side approaches: Cross-site Scripting is essentially an input filtering failure. Consequently, methods have been developed to target malicious inputs even before they reach the web server. Traditionally, web application firewalls (WAFs) are either scanning for attack signatures in the parameters passed on to the web application [11] (including POST-parameters, cookies, etc.), or require an administrator to manually specify a ruleset to match requests against [21]. Both ways can be regarded as an external second input filtering layer. A first anomaly-based intrusion detection system for web applications was proposed by Kruegel and Vigna in [14]. Their system derives a number of statistical characteristics from observed HTTP requests, regarding the parameter's length, character distribution, structure, presence and order. However, unlike our methods, both approaches concentrate solely on the incoming query parameters while ignoring the respective HTTP response, thus either causing unnecessary false positives or missing certain attacks.

Ismail et al. [9] describe an XSS detection mechanism which follows an approach similar to our reflected detector (see Sec. 3). Using a server-side proxy incoming parameters are checked for contained HTML markup. If such a parameter could be identified, the respective HTTP response is examined if the same HTML markup can be found in the response's HTML content. In comparison to our approach the proposed technique has several shortcomings. The HTML-based matching approach is inaccurate, as it fails to identify in-script- and attribute-injections. Furthermore, unlike our technique, the proposed detector also does not consider transformation-processes, such as character-encoding or removal filters, that may alter the incoming parameters before their reflection on the outgoing HTML.

Taint analysis has been proven to be a powerful tool for detecting code injection vulnerabilities. Taint analysis tracks the flow of untrusted data through the application. All user-provided data is "tainted" until its state is explicitly set to be "untainted". This allows the detection if untrusted data is used in a security sensible context. Taint analysis was first introduced by Perl's taint mode. More recent work describes finer grained approaches towards dynamic taint propagation. These techniques allow the tracking of untrusted input on the basis of single characters. In independent concurrent works Nguyen-Tuong et al [17] and Pietraszek and Vanden Berghe [18] proposed fine grained taint propagation to counter various classes of injection attacks. Halfond et al. [6] describe a related approach ("positive tainting") which, unlike other proposals, is based on the tracking of trusted data. Xu et al [24] propose a fine grained taint mechanism that is implemented using a C-to-C source code translation technique. Their method detects a wide range of injection attacks in C programs and in languages which use interpreters that were written in C. To protect an interpreted application against injection attacks the application has to be executed by a recompiled interpreter. However, in any case dynamic taint tracking requires profound changes to either the monitored application or the application server/run-time. Thus, access to the source code of one of these components is required. Also, a taint tracking based solution is necessarily always specific for a certain technology (programming language, application server), while real-life web applications are often composed of heterogeneous systems. Finally, real-time data-tracking always introduces certain performance penalties. In comparison, our approach is applicable with all languages and application servers, does not require any changes to the executed code, and is able to monitor highly heterogeneous set-ups. Also, as we propose a passive offline detector, no performance penalties are introduced.

Client-side approaches: We are proposing detection methods that are positioned exclusively at the server-side. For the sake of completeness, this section lists related approaches that incorporate the client-side web browser:

In concurrent and independent work an XSS filter for the Internet Explorer browser [20] was implemented which follows an approach that is closely related to our reflected detector: Based on an analysis of outgoing HTTP parameters, signatures are generated which are then checked against the corresponding HTTP response. Furthermore, the NoScript-plugin for Firefox [15] provides a simple protection mechanism against reflected XSS: Outgoing HTTP parameters are checked if they potentially contain JavaScript code. If such parameters are detected, the plugin warns the user before sending the respective HTTP request. As the incoming HTTP response is ignored, the plugin produces unnecessary false positives. Both browser-based approaches are unable to detect stored XSS.

With Browser-Enforced Embedded Policies (BEEP) [10], the web server includes a whitelist-like policy into each page, allowing the browser to detect and filter unwanted scripts. As the policy itself is a JavaScript, this method is very flexible and for instance allows the definition of regions, where scripts are disallowed. BEEP requires the usage of a modified web browser. [10] does not elaborate how the list of legitimate scripts is supposed to be compiled. Instead this step is left to the application's developers. As our generic detector is specifically designed to establish the list of legitimate scripts, a combination of the two approaches appears to be promising.

Finally, Hallaraker and Vigna in [7] modified Mozilla's SpiderMonkey Engine to track the behaviour of client-side JavaScript. The activity profile of each script then is matched against a set of high-level policies for detecting malicious behaviour.

9. Conclusion

We described XSSDS a server-side Cross-site Scripting detection system. The systems uses two novel detection-approaches that are based on generic observations of XSS attacks and web applications. A prototypical implementation demonstrated our approach's capabilities to reliably detect XSS attacks while maintaining a tolerable false positive rate. As our approach is completely passive and solely requires reading access to the application's HTTP traffic, it is applicable to a wide range of scenarios and works together with all existing web technologies.

References

[1] R. A. Baeza-Yates and G. H. Gonnet. Fast text searching for regular expressions or automaton searching on tries. *Journal of the ACM*, 43(6):915 – 936, November 1996.

[2] Blwood. Multiple xss vulnerabilities in tikiwiki 1.9.x. mailing list Bugtraq, `http://www.securityfocus.com/archive/1/435127/30/120/threaded`, May 2006.

[3] S. Christey and R. A. Martin. Vulnerability type distributions in cve, version 1.1. [online], `http://cwe.mitre.org/documents/vuln-trends/index.html`, (09/11/07), May 2007.

[4] K. Fernandez and D. Pagkalos. Xssed.com - xss (cross-site scripting) information and vulnerabile websites archive. [online], `http://xssed.com` (03/20/08).

[5] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, New York, USA, 1997. ISBN 0521585198.

[6] W. G. Halfond, A. Orso, and P. Manolios. Using positive tainting and syntax-aware evaluation to counter sql injection attacks. In *14th ACM Symposium on the Foundations of Software Engineering (FSE)*, 2006.

[7] O. Hallaraker and G. Vigna. Detecting malicious javascript code in mozilla. In *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 85–94, June 2005.

[8] R. Hansen. XSS (cross-site scripting) cheat sheet - esp: for filter evasion. [online], `http://ha.ckers.org/xss.html`, (05/05/07).

[9] O. Ismail, M. Eto, Y. Kadobayashi, and S. Yamaguchi. A proposal and implementation of automatic detection/collection system for cross-site scripting vulnerability. In *8th International Conference on Advanced Information Networking and Applications (AINA04)*, March 2004.

[10] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *16th International World Wide Web Conference (WWW2007)*, May 2007.

[11] A. Klein. Cross site scripting explained. White Paper, Sanctum Security Group, `http://crypto.stanford.edu/cs155/CSS.pdf`, June 2002.

[12] A. Klein. Dom based cross site scripting or xss of the third kind. [online], `http://www.webappsec.org/projects/articles/071105.shtml`, (05/05/07), Sebtember 2005.

[13] J. Kratzer. Jspwiki multiple vulnerabilitie. Posting to the Bugtraq mailinglist, `http://seclists.org/bugtraq/2007/Sep/0324.html`, September 2007.

[14] C. Kruegel and G. Vigna. Anomaly detection of web-based attacks. In *Proceedings of the 10th ACM Conference on Computer and Communication Security (CCS '03)*, pages 251–261. ACM Press, October 2003.

[15] G. Maone. Noscript firefox extension. Software, `http://www.noscript.net/whats`, 2006.

[16] Misc. New xss vectors/unusual javascript. [online], `http://sla.ckers.org/forum/read.php?2,15812` (04/01/08), 2007.

[17] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *20th IFIP International Information Security Conference*, May 2005.

[18] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection (RAID2005)*, 2005.

[19] A. Pigrelax. Xss in nested tag in phpbb 2.0.16. mailing list Bugtraq, `http://www.securityfocus.com/archive/1/404300`, July 2005.

[20] D. Ross. IE 8 XSS filter architecture/implementation. [online], `http://blogs.technet.com/swi/archive/2008/08/18/ie-8-xss-filter-architecture-implementation.aspx` (09/09/08), August 2008.

[21] D. Scott and R. Sharp. Abstracting application-level web security. In *WWW 2002*, pages 396 – 407. ACM Press New York, NY, USA, 2002.

[22] P. Sowden. rbnarcissus. Software, `http://code.google.com/p/rbnarcissus/` (04/01/08), 2008.

[23] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249 – 260, 1995.

[24] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *15th USENIX Security Symposium*, August 2006.