# Grammar Based Oracle for Security Testing of Web Applications

Andrea Avancini and Mariano Ceccato
*Fondazione Bruno Kessler*
*Trento, Italy*
{*anavancini,ceccato*}*@fbk.eu*

***Abstract*—The goal of security testing is to detect those defects that could be exploited to conduct attacks. Existing works, however, address security testing mostly from the point of view of automatic generation of test cases. Less attention is paid to the problem of developing and integrating with a security oracle.**

**In this paper we address the problem of the security oracle, in particular for Cross-Site Scripting vulnerabilities. We rely on existing test cases to collect HTML pages in safe conditions, i.e. when no attack is run. Pages are then used to construct the *safe model* of the application under analysis, a model that describes the structure of an application response page for safe input values. The oracle eventually detects a successful attack when a test makes the application display a web page that is not compliant with the safe model.**

***Keywords*-security testing; test oracle; cross site scripting**

## I. Introduction

Web applications are continuously exposed to a potentially hostile and dangerous environment. In fact, they are often targets of attacks aiming at turning them down (denial-of-service) or at stealing sensitive information. Cross-Site Scripting vulnerabilities (XSS hereafter) are one of the most common class of flaws related to web applications. XSS are caused by improper validation of input data (e.g., coming from the user). Input data may contain JavaScript or HTML fragments that could flush to the web page, altering the resulting content such that malicious code is injected. When executed by the user browser, such code may display false messages or add malicious features (e.g., to disclose sensitive data).

Goal of security testing is to guarantee a high quality of web applications from the security point of view, so as to limit the possibility to suffer the consequences of attacks. In literature, many approaches [1], [2], [3], [4], [5], [6] have been proposed for automating the generation of security test cases. However, a reusable security oracle, required to check whether security test cases actually expose application vulnerabilities, is still an open problem. In fact, the security oracle has been addressed either manually [1] or by approaches that, even if automatic, they are tailored on the specific test case generation algorithm [2], [3], [4], [5], [6], so that they can not be easily reused in different contexts. For example, in [2] the oracle consists in checking if a web page contains the same JavaScript fragment used for the automatic generation of a test case.

In the present paper, we cope with the problem of the security oracle for XSS vulnerabilities, that (i) is not strictly dependent on the test case generation step, and that (ii) is able to detect a more broad class of code injections. Our oracle is based on the *safe model*, the intended syntactic structure of the web application in safe conditions. An attack is classified as successful when it causes structural changes that violate the safe model. Alterations detected by the oracle are not just JavaScript code injections, but more generally any HTML code injection (e.g., new links or iframes) that modifies the intended page content.

In Section II we summarize the background on web application vulnerabilities and security testing, required for presenting our contribution on the security oracle in Section III. Preliminary experimental results are presented in Section IV. Comparison with the state of the art (Sections V) and conclusions (Section VI) close the paper.

## II. Background

### A. Cross-Site Scripting Vulnerabilities

```
 1  $user = $_GET["username"];
 2  $pass = $_GET["password"];
 3  $pass2 = $_GET["password2"];
 4  if ( strpos($user, "<script") )
 5     $user=htmlspecialchars($user);
 6  if ( $user in $users )
 7     echo "username already taken";
    else
 8     if ( strlen($pass) < 5)
 9        echo "password too short";
       else
10        if ( $pass == $pass2 )
11           new_user($user, $pass);
12           echo "new account for";
13           echo $user;               //sink
          else
14           echo "passwords do not match";
```

Figure 1. Running example of a XSS vulnerability on PHP code.

Figure 1 shows a portion of PHP code fragment that contains a reflected XSS vulnerability, i.e. there exists an execution flow where a non validated input value (variable $user) is displayed in the output page (line 13). Even if this

example focuses on PHP, this kind of vulnerability affects web applications written in any programming language.

The example web page registers a new user, using input values coming from a previous web page that contains a form. Input values *username*, *password* and *password2* from the incoming HTML request, are represented in PHP as the special associative array *$_GET*. Input values are assigned to local variables *$user*, *$pass* and *$pass2* (lines 1 to 3). These variables should not be appended to the output response page, because they are still not validated and may contain HTML or JavaScript code fragments.

Validation starts at line 5. Possibly dangerous characters are removed from variable *$user* using the PHP function *htmlspecialchars*. This function changes special HTML characters (e.g. "<", ">" and """) into their encoded form ("<", ">", and """), safe when printed in a web page. The *sanitized* content of variable *$user* is then stored again in variable *$user*. Validation (line 5), however, is done only when condition on line 4 holds. Such condition, unfortunately, does not cover all the possible dangerous cases. For example, harmful code containing a different tag (e.g. <a>) or with a space between < and script would skip sanitization.

The validation of *$user* continues on line 6, checking that user name has not been already taken. Then, line 8 enforces password length, and line 10 checks that password is typed twice the same.

In case the validation phase passes, a user is created, the web page notifies successful creation of such a new user and variable *$user* is printed on the confirmation web page (line 13), possibly causing a security threat. Because of inadequate validation at lines 4 and 5, *$user* may contain the original untrusted value.

### B. Vulnerability Exploit

The vulnerability shown in Figure 1 can be exploited by an attack that:

1) makes the execution follow a path that satisfies conditions on username and password so that it reaches the *sink* statement on line 13;
2) skips the sanitization of line 5; and
3) stores an attack vector in variable *$user*.

An example of successful attack is represented by an HTML request containing the parameter *username* set to <a href="" onclick="this.href='evil.php?data='%2Bdocument.cookie"> click here</a> and *password* and *password2* equal to xxxxx. In fact, the corresponding user name would pass validation as it is not already in use (line 6) and it does not contain the <script string (line 4). Then, the password is long enough (line 8) and it is typed twice the same (line 10).

When such user name is appended to the response page (on line 13), it alters the HTML structure (%2B is decoded as "+"), as a brand new link (the <a> tag) is injected, pointing to an external web site controlled by the attacker (i.e., *evil.php*). In case such link is triggered by the legitimate user, her/his cookie is encoded as a page parameter and sent to attacker controlled site (evil.php?data='+document.cookie'). With the stolen cookie, the attacker may pretend to be the legitimate user, hijack her/his session and access his/her sensitive data. This attack can be achieved, for example, by sending the mentioned link to legitimate users by e-mail, and by convincing them to click on it.

### C. Security Test Case Generation

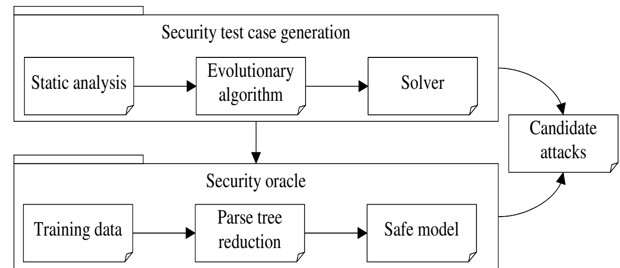


Figure 2. Tool chain.

In previous works [7], [8] we presented an approach for automatic generation of security test cases, based on the integration of a genetic algorithm and a constraint solver. The mentioned approach is summarized here for completeness. Security test case generation is composed of three steps: static analysis, generic algorithm and solver, as shown in Figure 2-top.

**Static analysis:** the identification of vulnerabilities relies on taint analysis [9], a static analysis technique that tracks the tainted/untainted status of variables throughout the application control flow. A vulnerability is reported whenever a possibly tainted variable is used in a sink statement (e.g. print). In case of XSS [10], tainted values are those values that come from untrusted sources (data base and user input) and sinks are all the print statements that append a string into the web page. Tainted status is propagated on assignments and tainted variables become untainted upon sanitization (e.g., function *htmlspecialchars* in PHP). Taint analysis is formulated as a flow analysis [11] problem, where the information propagated in the control flow graph is the set of variables holding tainted values.

Taint analysis does not provide executable test cases, but just the *data slice* that gives raise to the vulnerability. The data slice consists of the chain of those assignments that make a tainted value flow into a sink statement, skipping validation. This list of assignments does not identify a path in the control flow, but a (possibly infinite) set of paths. Data slice for Figure 1 is composed by lines {1, 13}.

We collect all the control statements that hold a control dependence on the assignments in the data slice, because they drive the code execution into (or away from) a vulnerable path. The branches to traverse in order to reach the vulnerable statement with a tainted value are called *target branches*, because they are those branches that a security test case must take to execute the vulnerable data slice. In the running example of Figure 1, the target branches are {4-6, 6-8, 8-10, 10-11}. The first (4-6) is required to skip sanitization and the others are required to pass validation and reach the vulnerable sink.

**Genetic algorithm:** The genetic algorithm starts from an initial set of random tests and evolves them by combining together the fittest solutions, with the hope of generating fitter ones, until the final solution is found. The fitness is computed by a *fitness function* that corresponds to the *approach level*, i.e., the percentage of target branches that are taken by the execution of a test case. The genetic algorithm terminates either when a test case is found that is able to traverse 100% of the target branches or when a time-out is reached.

On each evolution iteration, a subset of the current population is selected to form the next population, by giving more chances to those individuals that are more likely to generate the final solution, i.e., they have a better value of the fitness function. Selected individuals are paired to generate offspring by *crossing over* their chromosomes and by *mutating* them, with the hope of generating better solutions. Test cases to evolve are input values for the page under analysis. Input values are in the form of {*parameter name*, *parameter value*}, and they can be turned directly into HTML requests. As genetic algorithms are heuristics, they are not ensured to converge to the final solution, especially when input data must satisfy complex conditions. For example, in Figure 1, condition at line 10 requires that the two random strings on *password* and *password2* are equal. The probability of randomly generating two identical strings is very low. Moreover, genetic algorithms suffer local optimum. Whenever optimization finds a local optimum, it could be difficult to move forward from it.

**Solver based local search:** When the genetic algorithm stops improving, a local optimum or a particularly hard to satisfy condition are probably found. In these cases, we switch to a local search strategy based on a constraint solver. This approach has been inspired by dynamic symbolic execution [12].

The source code of the application under analysis is instrumented such that, on decision points, path conditions are collected in terms of conditions on symbolic inputs. This requires to update a run-time map of the symbolic values of program variables, in terms of their relation with input values. On assignments, the dynamic map is updated with the new symbolic values of the assigned variables. A SMT solver would easily address constraints on string equality (e.g., line 10). However, due to limitations of solvers in handling complex expression on strings and non-linear arithmetic, sometimes symbolic expressions can not be used and concrete values are resorted. For this reason, adopting the solver alone is of limited benefit, but it is quite helpful when complemented by a genetic algorithm.

Path conditions are collected until the execution takes a branch that diverges from one of the target branches. The condition on the diverging branch is then negated before being added to the others. All the conditions are then passed to the solver that possibly elaborates a new test case with input values that satisfy the missed target branch. The new test case improves the fitness function, because it takes one target branch more than any previously generated test. At this stage, control passes to the genetic algorithm again for further optimization.

Test cases generated according to this procedure show the presence of a potential security fault. Test cases demonstrate how untrusted input values may flow into a response page without proper validation. These test cases, however, do not represent actual attacks, as they do not perform any code injection. In order to generate proof-of-concept attacks, a security oracle is required.

## III. Security Oracle

The goal of an XSS attack is to inject JavaScript or HTML code fragments into a web page. Thus, consequences of injection should be evident as structural changes in the parse tree of the page under attack, when compared with the parse tree of the same page running on normal conditions.

However, content of a dynamic web application depends either on input values and on specific executions. The parse tree of the same page may vary a lot, even without code injection. The same PHP script may take different alternative actions or, at least, display different results, because of different queries or computations.

The security oracle should distinguish between those variations that are safe because due to the dynamic behavior of the web applications and those variations caused by code injection due to successful attacks. A model of the parse trees on *safe* cases is constructed according to the steps shown in Figure 2-bottom:

1) **Training data:** several test cases are generated and run on the web page under analysis. The HTML pages resulting from the test case executions are collected; and
2) **Reduction:** HTML code is parsed and each parse tree is processed to remove all those details that are not relevant for a code injection attack, in order to obtain a more compact representation of each particular execution; and
3) **Safe model:** all the reductions are combined together into a common structure, which is abstract enough to

reasonably represent the structure of the executions of the target branches in *safe* conditions.

Eventually, a new parse tree that would not satisfy the *safe model* would be classified as code injection, because it would represent a successful XSS attack.

### A. Training Data

The test case generation procedure summarized in Section II computes just few (or just one) test cases for each vulnerability. More distinct test cases are needed to build a model with an appropriate level of generality. All the training test cases, however, must still cover the same vulnerability, i.e. to satisfy the conditions on the target branches.

To increase the number of test cases, we mutate the initial set of test case(s) using the following mutation operators:

**Change parameter value:** The value of a parameter is randomly changed. One parameter is chosen with uniform probability and its value is changed in two alternative ways. Either (1) one character of the current string is randomly selected and substituted with a random character or (2) a random string is concatenated to the existing parameter value. The same applies on numeric values, the only difference is that just digits are used instead of characters.

**Insertion of a new parameter:** A new parameter is added to the test case. The parameter name is randomly selected among the available parameter names and its value and type are generated randomly.

All the constant strings that appear in the page source code are collected and stored into a pool. When a new random string is required, such string is either chosen from the string pool (probability 1/2) or randomly generated (probability 1/2). In the latter case, the following algorithm is resorted to. A character is randomly selected from a set containing alphanumeric characters and special HTML/JavaScript characters, i.e. from [a-zA-Z0-9], and [<>?&+-*/=\()[ ]"']. After the first character, a second one is added with probability 1/2, so the probability of having a string of length 2 is 1/2. In case the second character has been added, a third one is added with probability 1/2, so the probability of a string of 3 characters is $1/2^2$ = 1/4. More characters are added with a probability that decays exponentially. In general the probability of generating a string of length *n* is $1/2^{n-1}$.

To generate a random numeric value, we adopt the same algorithm, selecting random characters from the class [0-9].

### B. Reduction

Test cases are executed on the instrumented web application to record what branches are traversed at run-time. For those tests that take 100% of the target branches, we collect the generated HTML code and the corresponding parse tree. Then each parse tree is reduced applying the subsequent abstraction rules:

**Remove text and formatting:** The parse tree is reduced by removing text and comments, so just HTML tags and scripts remain. We remove also all the formatting tags (e.g., <tt>, <i>, <b>, <big>, <small>, <br> and <hr>) that do not specify event attributes (*onclick*, *ondblclick*, *onmousedown*, *nmousemove* and similar).

**Compact lists:** A list of tags that contains the same elements/attributes is replaced with a repetition pattern. For example, an item list with consecutive identical <li> entries is replaced by the pattern {<li>}+.

**Compact repeated sequence:** When a sequence of tags is repeated with exactly the same attributes, it is replaced with a list pattern. For example a sequence of many cells <td></td> in a table row is replaced by the pattern {<td></td>}+. Possibly, a sequence of table rows <tr>{<td></td>}+</tr> is replaced by pattern {<tr>{<td></td>}+ </tr>}+

**Merge same attribute:** When previous *compact* rules do not apply because matching tags have a common attribute but with different values, the common attribute is replaced by a regular expression with the alternative values and then tags are transformed. For example, tags <li class="c1"> <li class="c2"> are replaced by pattern {<li class={"c1" | "c2"}>}+

**Merge different attributes:** When *compact* rules do not apply because matching tags have different attributes, the attributes are replaced by a regular expression containing the alternatives and then the previous rules apply. For example, tags <li class="c1"> <li lang="EN"> are replaced by pattern {<li {class="c1" | lang="EN"}>}+

A special case of this rule is when a tag misses an attribute. In this case the pattern to use relies on the empty alternative $\epsilon$. For example, tags <li class="c1"> <li> are replaced by pattern {<li {class="c1" | $\epsilon$}>}+

**Pattern merge:** A special case of *merge attributes* may require to merge intermediate results of already composed patterns. Since only attribute patterns use the *alternatives* operator (i.e., "|"), the merge consists in the union of all the possible alternatives. For example, sequence of patterns {<li {class="c1" | lang="EN"}>}+ {<li {class="c2" | $\epsilon$}>}+ is replaced by pattern {<li {class="c1" | lang="EN" | class="c2" | $\epsilon$}>}+

### C. Safe Model

When parse tree reductions are available for all the test cases, they need to be combined into a single abstraction. This abstraction models the safe execution of the vulnerable data slice when no injection takes place. We call this abstraction the *safe model*.

Combination is done pairwise and it starts by combining two parse tree reductions. The result is then combined with the third tree reduction. The combination process continues combining one reduction after the other, until the last parse tree reduction is merged into the final model.

The trees to merge are visited in parallel using a breadth-first strategy. Starting from the root, all the children nodes are fully processed, before continuing with the grand-children nodes. Figure 3 shows an example of the combination, reductions (a) and (b) are combined in the safe model (c). Both of the trees start with the same <body> tag, so this tag is reported in the safe model (c). The second level of (a) and (b) (i.e., the children nodes of <body>) contain some differences. The difference is computed using the *longest common subsequence* algorithm [13]. The algorithm reports nodes <table> and <img> as common, so they are simply copied in the result (c). Non-common nodes are combined in patterns, before they are reported in the result (c) according to these cases:

- If the difference consists in the substitution of consecutive nodes, alternatives are reported in the pattern. In the example, the children in second position are different, <ul> and <a>, so the result contains the pattern {<ul> | <a>}; or
- If the difference consists in the insertion (or deletion) of consecutive nodes, the empty alternative $\epsilon$ is used in the pattern. In the example, the fourth child (<a>) of (a) does not match any node of (b). So, the results contains the pattern {<a> | $\epsilon$}.

When two nodes are merged into a pattern, also their children are recursively merged into patterns. In the example, nodes <ul> and <a> are merged. The former has a child node {<li>}+, while the latter has no child. So, the merged node of (c) will have a brand new child, consisting of the union pattern with the empty alternative $\epsilon$, i.e. {{<li>}+ | $\epsilon$}.

### D. Attack Validation

Once the safe model is available, it can be used to assess whether a new test case is a successful attack. The decision procedure starts by running the candidate attack and collecting the HTML output page. Its parse tree is then reduced and combined with the existing safe model into a potentially different model, the *evaluation model*. A difference between the safe model and the evaluation model means that the candidate attack caused structural changes in the page. Thus code injection took place and the attack is classified as successful.

## IV. Preliminary Results

### A. Implementation

Security testing described in Section II relies on a previous tool [8] for vulnerability identification and vulnerability coverage. The security oracle described in Section III has been implemented in a proof-of-concept prototype and integrated in the security testing process.

**Training data:** New test cases are collected by mutating the test cases generated by the security testing tool. Mutation partially reuses operators available in the genetic algorithm. Vulnerability coverage is checked using the same instrumentation capabilities available in security testing.

**Reduction & Safe model:** The TXL language [14] is used to implement rules for parse tree reduction and for merging reductions into the safe model. TXL supports the definition of grammar-based rules to perform a source-to-source code transformation. Transformations are defined on a grammar obtained from the base HTML grammar extended by adding patterns for tags and attributes.

**Attack generation:** For turning security test cases into code injection exploits, attack strings are injected into test input data. Attack strings come from a library of typical fragments of attacks (e.g., HTML tags containing scripts and links) that were used in penetration testing and in black-box fuzzing [15]. An attack fragment is randomly chosen from the library and injected into one of the parameter values of the original test case. Then, the newly created candidate attack is executed to check if it still traverses all the *target branches*. Finally, the response page is reduced and the evaluation model based on it is compared with the safe model to assess code injection.

### B. Empirical Data

The tool-prototype is applied on a case study, Yapig version 0.95b, an open-source PHP application that implements an image gallery management system. It consists of 9,113 lines of code and 53 source files, with 160 user-defined functions and 2,638 branches.

Static analysis reported a total of 25 candidate XSS vulnerabilities (including false positives) and test cases generation was run on them. To assess the security oracle we had to select a vulnerability among those covered by an automatically generated test case. The selected vulnerability was caused by a missing sanitization routine in the page to upload pictures. An attacker may inject malicious code into one of the parameters accepted by this page and, if the other parameters are consistent with the application logic, the execution proceeds. The injected code flows into the sink and it is printed into the output page, delivering harmful code to the victim browser. This vulnerability was covered by 7 distinct test cases, to be used for generating the safe model.

Parse tree reduction and pairwise merge into the final safe model took just few seconds, because the HTML pages were relatively small. In fact, the initial pages on average consisted of 150 LoC, while their reductions consisted of 90 LoC, with a compression factor of 1.6. The most frequently occurring reduction pattern was a sequence of <a> tags with different attribute values, often used to build linked menus in various parts of the web page. These tags were replaced by a single pattern, according to the *compact list* rule in the *merge same attribute* variant. This pattern applied 3 times in the reduced HTML. The same reduction rule replaced the sequences of <meta> tags in the HTML header. The rule *compact list* in the *merge different attributes* replaced

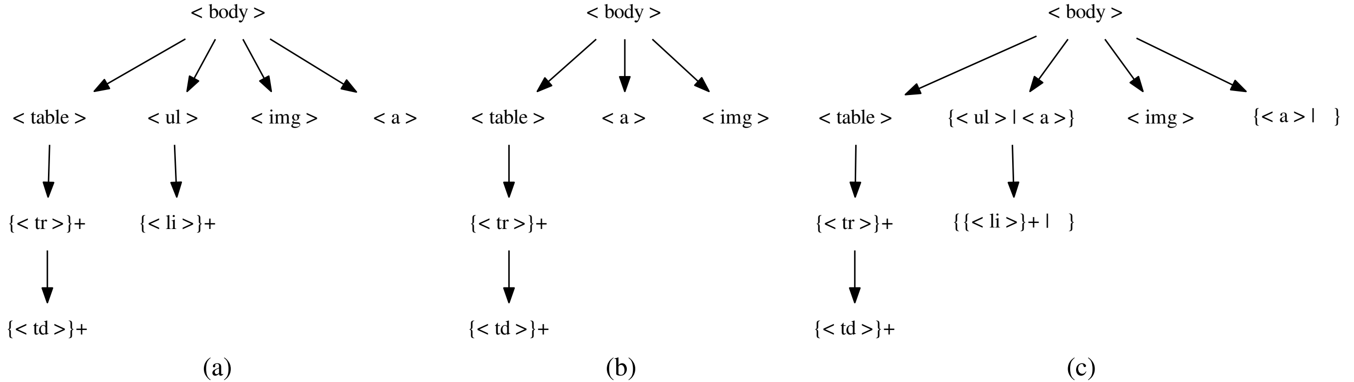


Figure 3. Example of combination of two parse tree reductions (a) and (b) into the safe model (c).

a sequence of <li> tags with different elements. The 7 reductions have been finally merged into the safe model.

Then, the library of attack patterns was used to mutate test case input values and generate candidate attacks. Out of 19 candidate attacks, only 5 of them were still covering all the target branches. We collected the output pages produced by them, we transformed them into 5 reductions and then into 5 evaluation model. None of the 5 models satisfied the safe model, so attacks were classified by the oracle as successful. Manual inspection of the response pages revealed that new code was effectively injected, although injection consisted of always different HTML tags, because different attack patterns were used. Thus the classification by the oracle was correct.

## V. Related Works

A fundamental part of security testing is deciding about successful attacks, i.e. when a test case is able to inject malicious code. Initially, checking code injection was a manual task delegated to programmers. For instance, in the work by Tappenden et al. [1] security testing is approached with an agile methodology using HTTP-unit and the verification of test outcome is a manual task.

Other approaches provide a higher level of automation. In [2] a library of documented known attacks is used to generate valid inputs for a web application. A symbolic data base is implemented to propagate the tainted status of input values through the data base to the final attack sinks. A first stage of the oracle adopts dynamic taint analysis to verify that tainted data are used in a sink. The second stage of the oracle is based on a comparison of safe pages with pages on candidate attacks. This check consists in verifying if pages differ with respect to “script-inducing constructs”, i.e. new scripts or different *href* elements.

In other works [3], [4], [5], [6], the oracle consists in checking if a response page contains the same *<script>* tag passed as input. McAllister et al. [3] adopt a black-box approach to detect XSS vulnerabilities. Data collected during the interaction with real users are subject to fuzzing, so as to increase test coverage. The oracle for XSS attacks checks if the script passed as input is also present in the output page. We adopt a similar approach to artificially multiply the limited amount of test cases initially available, in order to make the safe model more general.

In Huang et. al. [4] data-entry-points are identified in web applications and attack patterns are used on them. The oracle consists in checking that input data containing the ”<script>” substrings are sanitized before using them in output construction.

The paper by Halfond et al. [5] presents a complete approach to identify XSS and SQLI vulnerabilities in Java web applications. (1) Input vectors are identified as parameters read by the page under analysis, together with their expected type and domain. Flow analysis is resorted to group input vectors and domains into input interfaces. Then (2) attack patterns are used to generate many attacks for these interfaces. Eventually (3) page execution is monitored and HTTP response is inspected to verify if attacks are successful, i.e. the executed SQL statement is dangerous or new HTML scripts have been injected. Also in this case the oracle consists in detecting if the response page contains a script tag injected by input data.

In [6] a scanner identifies input fields on forms, they are later used to mount attacks. The oracle consists in detecting whether a script injected in a form appears in the response page. As commented by the authors, this approach suffers false negatives.

Limiting the check to injected script tags guarantee a high precision, but recall may be low, because of vulnerabilities depending on other tags may not be detected in these oracles. Our approach is more general, because we build a model of a safe execution that is general enough to capture any form of code injection that does not satisfy the model of a safe execution.

A different kind of security oracle is adopted in other works on mutation testing [16], [17], [18]. Several mutation operators are defined to expose SQL-injection vulnerabilities on JSP applications [16], format string bugs on C [17] and

XSS vulnerabilities on PHP code [18]. Test case adequacy is evaluated on the ability to kill mutants.

## VI. Conclusion

Security testing of web applications requires the adoption of a security oracle, able to classify test cases as successful attacks. In this paper we propose a security oracle for Cross-Site Scripting vulnerabilities. In order to be quite general and not dependent on training test cases, we propose to base the oracle on the safe model of the web application, i.e. on an abstraction of the parse trees of the HTML code resulting from safe executions.

The proposed security oracle has been assessed on a proof-of-concept case study, and empirical results suggest that the level of abstraction of the safe model is an appropriate trade off between generality and ability to detect attacks. In fact, on the one hand the oracle accepts a super-set of the training test cases, so abstraction makes the oracle quite independent from training data. On the other hand, even if abstract, the oracle was effective in classifying actual code injections as attacks.

As future works we plan to adopt the proposed oracle on bigger case studies and investigate on how the level of abstraction may impact precision and recall in classifying attacks. In fact, a too abstract oracle would be independent from training tests, but it could be too generic and miss real attacks. However, a too concrete model will not miss any attack, but it could also classify safe test cases as attacks.

## References


[1] A. Tappenden, P. Beatty, J. Miller, A. Geras, and M. Smith, "Agile security testing of web-based systems via httpunit," in *Agile Conference, 2005. Proceedings*, july 2005, pp. 29 – 38.

[2] A. Kieyzun, P. Guo, K. Jayaraman, and M. Ernst, "Automatic creation of sql injection and cross-site scripting attacks," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, may 2009, pp. 199 –209.

[3] S. McAllister, E. Kirda, and C. Kruegel, "Leveraging user interactions for in-depth testing of web applications," in *Recent Advances in Intrusion Detection*, ser. Lecture Notes in Computer Science, R. Lippmann, E. Kirda, and A. Trachtenberg, Eds. Springer Berlin / Heidelberg, 2008, vol. 5230, pp. 191–210.

[4] Y.-W. Huang, C.-H. Tsai, D. Lee, and S.-Y. Kuo, "Non-detrimental web application security scanning," in *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, nov. 2004, pp. 219 – 230.

[5] W. G. J. Halfond, S. R. Choudhary, and A. Orso, "Improving penetration testing through static and dynamic analysis," *Software Testing, Verification and Reliability*, vol. 21, no. 3, pp. 195–214, 2011. [Online]. Available: http://dx.doi.org/10.1002/stvr.450

[6] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic, "Secubat: a web vulnerability scanner," in *Proceedings of the 15th international conference on World Wide Web*, ser. WWW '06. New York, NY, USA: ACM, 2006, pp. 247–256. [Online]. Available: http://doi.acm.org/10.1145/1135777.1135817

[7] A. Avancini and M. Ceccato, "Towards security testing with taint analysis and genetic algorithms," in *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems*. ACM, 2010, pp. 65–71.

[8] ——, "Security testing of web applications: A search-based approach for cross-site scripting vulnerabilities," in *Source Code Analysis and Manipulation (SCAM), 2011 11th IEEE International Working Conference on*. IEEE, 2011, pp. 85–94.

[9] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities (short paper)," in *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 258–263.

[10] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *ICSE '08: Proceedings of the 30th international conference on Software engineering*. New York, NY, USA: ACM, 2008, pp. 171–180.

[11] M. Sharir and A. Pnueli, *Program Flow Analysis: Theory and Applications*. Prentice Hall, 1981, ch. Two approaches to interprocedural data flow analysis, pp. 189–233.

[12] K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for c," in *Proceedings of the 10th European software engineering conference*. New York, NY, USA: ACM, 2005, pp. 263–272.

[13] L. Bergroth, H. Hakonen, and T. Raita, "A survey of longest common subsequence algorithms," in *String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings. Seventh International Symposium on*, 2000, pp. 39 –48.

[14] J. Cordy, "The TXL source transformation language," *Science of Computer Programming*, vol. 61, no. 3, pp. 190–210, August 2006.

[15] N. Surribas, "Wapiti, web application vulnerability scanner/security auditor," 2006-2010. [Online]. Available: http://www.ict-romulus.eu/web/wapiti

[16] H. Shahriar and M. Zulkernine, "Music: Mutation-based sql injection vulnerability checking," in *Quality Software, 2008. QSIC '08. The Eighth International Conference on*, aug. 2008, pp. 77 –86.

[17] ——, "Mutation-based testing of format string bugs," in *High Assurance Systems Engineering Symposium, 2008. HASE 2008. 11th IEEE*, dec. 2008, pp. 229 –238.

[18] ——, "Mutec: Mutation-based testing of cross site scripting," in *Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems*, ser. IWSESS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 47–53. [Online]. Available: http://dx.doi.org/10.1109/IWSESS.2009.5068458