# Security Testing of Web Applications: a Search Based Approach for Cross-Site Scripting Vulnerabilities

Andrea Avancini, Mariano Ceccato

FBK-irst Trento, Italy
Email: {anavancini,ceccato}@fbk.eu

## Abstract

*More and more web applications suffer the presence of cross-site scripting vulnerabilities that could be exploited by attackers to access sensitive information (such as credentials or credit card numbers). Hence proper tests are required to assess the security of web applications.*

*In this paper, we resort to a search based approach for security testing web applications. We take advantage of static analysis to detect candidate cross-site scripting vulnerabilities. Input values that expose these vulnerabilities are searched by a genetic algorithm and, to help the genetic algorithm escape local optima, symbolic constraints are collected at run-time and passed to a solver. Search results represent test cases to be used by software developers to understand and fix security problems. We implemented this approach in a prototype and evaluated it on real world PHP code.*

## I. Introduction

Among all the programming errors that threaten the reliability of web applications, security faults are probably the most critical and latent. In fact, successful attacks may have dramatic consequences that range from simple denial of service (web-site is down) to sensitive information disclosure, possibly resulting in criminal acts, such as frauds and identity thefts.

One of the top-ranked vulnerabilities is cross-site scripting (XSS). This threat is due to inadequate validation of data coming from untrusted sources (e.g. user inputs). Tainted data may contain portions of HTML code (e.g. java-scripts) that could flow into the page under attack. Malicious scripts are meant to be eventually executed by web-browsers of legitimate users who access the pages under attack, for example to steal sensitive data.

Despite the crucial implications of security faults, manual code inspection is still a largely adopted practice to reveal and fix them. As applications are larger and larger, static analysis represents a valuable support by suggesting candidate vulnerabilities as starting points for the manual review, or conditions under which a vulnerability may appear. However, static analysis does not provide any executable example of how these (possibly complicated) conditions may turn into a security problem.

The main contribution of this paper is to integrate static taint analysis, genetic algorithms and constraint solving to automatically generate test cases that are able to expose XSS vulnerabilities on PHP web applications.

The problem of identifying a test case that is able to satisfy the critical conditions returned by static analysis can be formulated as a search problem. In this paper, we turn conditions detected by static analysis into structural constraints, branches to be traversed. Then, we adopt a genetic algorithm to find those input values that make the execution take such branches. In order to avoid the genetic algorithm to converge on local optima, we adopt a local search strategy. Symbolic constraints are collected along the execution as conditions on input values. A solver is then resorted to find new input values, able to improve the current solution and escape from the local optimum. This approach has been implemented in a prototype and applied on a case study. Test cases are generated that expose reflected XSS vulnerabilities on PHP code.

Test cases generated by our approach do not actually exploit a vulnerability, as they are not meant to inject code into a web page. However, they represent a crucial help for developers who need to understand vulnerabilities before fixing them. Test cases improve the limited support offered by static analysis.

Complicated vulnerabilities may require tainted data to be stored into the database, before being used in actual attacks. However, in this paper we focus on *reflected* vulnerabilities, those caused directly by input data, without considering the persistent storage.

While existing approaches for automatic generation of test cases (such as concolic testing or evolutionary testing) are meant to achieve a general and high structural coverage, we target just selected conditions. In fact, security problems may manifest just in strange and rare paths identified by static analysis, that even large test suites could miss.

After recalling the background on cross-site scripting vulnerabilities on Section II, the test case generation procedure is discussed on Section III. Then, implementation (Section IV) and empirical results (Section V) are presented. Related works and conclusions close the paper in Section VI and VII.

## II. Web site vulnerabilities

Cross-site scripting vulnerabilities (XSS for short) are caused by improper validation of input data (e.g., coming from the user). Input data may contain 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 disclose sensitive data to third parties. Even if this paper focuses on PHP, this kind of vulnerability affects web applications written in any programming language.

```
 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-like code fragment that contains a reflected XSS vulnerability. This web page register a new user, using input values coming from a web page that contains a form filled by the user. Input values *username*, *password* and *password2* from the incoming HTML request, are represented in PHP as the special array *$_GET*. Input values are assigned to local variables *$user*, *$pass* and *$pass2* (on lines 1 to 3).

Input values are then validated. At line 5, dangerous characters are removed from variable $user. PHP provides the function *htmlspecialchars* to sanitize strings. It changes special HTML characters (e.g. "<", ">" and "`"`") in their encoded form ("`&lt;`", "`&gt;`", and "`&quot;`"), safe when printed in a web page. However, the example fails to properly validate the content of variable $user. In fact, it is sanitized (line 5) only conditionally. The condition on line 4 fails to cover all the dangerous cases. For example, a string 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 by other users. After this, the password is validated. On line 8, password length is enforced and later, on line 10, password is checked to be typed twice the same.

In case the validation phase passes, the new user is created[1] and a web page notifies successful creation. Notification contains the name of the new user, variable *$user* is printed on the web page (line 13). This is a potential vulnerability, because of inadequate validation at lines 4 and 5.

The vulnerability can be exploited by executing a path that reaches the sink statement on line 13, while skipping sanitization on line 5. An example of successful attack is represented by an HTML request containing the parameters *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 and it does not contain the `<script` string. Moreover, password is typed twice the same and it is long enough.

When such user name is appended on the response page, 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, his/her cookie is encode as a page parameter and sent to the attacker site (`evil.php?data='+document.cookie'`). With the stolen cookie, the attacker can pretend to be the legitimate user, hijack his/her session and access his/her sensitive data from the web-site under attack. 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.

[1]The act of inserting tainted values in the database may result in a different threat known as persistent XSS, which is not addressed here, as this paper focuses on reflected XSS.

## III. Security testing

### A. Taint analysis

Taint analysis is a static analysis technique devoted to track the tainted/untainted status of variables throughout the application control flow. A vulnerability is reported whenever a possibly tainted variable is used in a sensitive (sink) statement. In the case of XSS [1], tainted values are those that come from the external world (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 to the variable on the left hand side, when an expression on the right hand side uses a tainted value. Tainted variables become untainted upon sanitization (e.g., function *htmlspecialchars* in PHP) and when they are assigned untainted values, either a constant or an expression that does not contain tainted values.

Taint analysis is formulated as a flow analysis [2] problem, where the information propagated in the control flow graph is the set of variables holding tainted values. The information generated and killed at each node (statement) *n* can be defined as follow:

$$GEN[n] = \{v \mid \textit{statement } n \textit{ assigns a tainted value to } v\} \quad (1)$$

$$KILL[n] = \{v \mid \textit{statement } n \textit{ sanitizes the value of } v\} \quad (2)$$

The flow propagation is in the forward direction, with union as the meet operator at junction nodes. Flow analysis terminates when the following equations produce the least fix-point:

$$IN[n] = \bigcup_{p \in pred(n)} OUT[p] \quad (3)$$

$$OUT[n] = GEN[n] \cup (IN[n] \setminus KILL[n]) \quad (4)$$

where *pred(n)* indicates the set of nodes that precede immediately $n$ in the control flow graph. A *candidate vulnerability* is reported when a tainted value reaches a sink statement:

$$n \textit{ is a sink for variable } v \quad (5)$$

$$v \in OUT[n] \quad (6)$$

Static analysis is conservative and flow values propagate also on infeasible paths. Vulnerabilities are just *candidate*, because they contain false positives, i.e. vulnerabilities that insists on infeasible paths.

Statement 13 in Figure 1 is a sink for variable $user, because such variable is appended to the web page (`echo` statement) at that line. Because such variable is possibly tainted, a candidate vulnerability is reported, i.e. $user@13.

Taint analysis does not report executable test cases, but just the data slice that gave raise to the vulnerability, consisting of the chain of those assignments that make an input value flow into a sink statement, skipping validation. In the example, the assignment chain is {$user@1, $user@13}. In general, this list of assignments does not identify a single vulnerable path in the control flow, but a (possibly infinite) set of paths. For example, we do not know how many iterations to take on loops and which branch to take in all those decision points (e.g. `if-then-else`) not directly involved in the chain.

From the chain of assignments, we can compute the list of branches to traverse in order to reach vulnerable point with a tainted value. We call these branches *target branches*. In the running example, 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 algorithms can be resorted to find proper input values that make the execution traverse all the *target branches*.

### B. Genetic algorithms

```
1 population = generateRandomPopulation();
2 for (T in vulnerabilities) {
3   while ( not covered(T) AND
                     attempt < maxTry ) {
4      selection = select(population);
5      offspring = crossOver(selection);
6      population = mutate(offspring);
7      attempt = attempt + 1;
    } }
```

**Figure 2. Genetic algorithm for path sensitization**

The genetic algorithm evolves a set of solutions by combining together the fittest solutions, with the hope of generating fitter ones, until the optimum solution is found. The details of the algorithm are shown in Figure 2. An initial population composed of a random set of input values (line 1), is evolved until the maximum number of trials is completed or the solution is found (line 3). On each evolution iteration, a subset of the population is selected (line 4) 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 (line 5) and by *mutating* them (line 6), with the hope of generating better solutions.

*Chromosomes:* Individuals are represented as chromosomes. They contain the input values for the page under analysis. A chromosome is a set of triples, each triple contains parameter name, value and type (*S* for strings and *N* for numbers). Individuals can be directly turned into HTML requests for the application under test by encoding them in the corresponding URL.

While parameter names can be found from the parameters used in the web page source code, parameter values are random values. Types are also randomly chosen, because PHP is a loosely typed language and the same variable can be interpreted as number or string. In fact, the language implicitly converts variables on the fly, depending on their use.

*Fitness function:* The fitness function corresponds to the *approach level*, i.e., the amount of target branches that are executed when the application is run with the inputs from the current individual. The solution for the current vulnerability is found when an individual is able to traverse 100% of the target branches. The more an individual is near to this condition, the higher value of fitness function it will have.

*Selection:* At each iteration a sample of the population is selected for evolution, the probability of selecting an individual for the next generation is proportional to the value of its fitness function. In other words, input values more near to cover a vulnerability are more likely to be selected for contributing to the new generation. Then the new generation is subject to mutation for a possible improvement of the fitness function.

*One point cross over:* When two individuals are selected for crossing over, their chromosomes are randomly divided in two pieces. Two brand new individuals are generated by recombining two halves together. In the subsequent example, chromosomes *A* and *B* have been split. *C* is the result of joining the first part of *A* with the second part of *B*, while *D* is the union of the remaining two parts:

Example:

$A : \{(name, S, \text{``}john''), (surname, S, \text{``}smith''), (\mathbf{age}, \mathbf{N}, \mathbf{23})\}$

$B : \{(name, S, \text{``}mark''), (address, S, \text{``}broad''), (\mathbf{job}, \mathbf{S}, \text{``}\mathbf{teacher}''))\}$

$\downarrow$

$C : \{(name, S, \text{``}john''), (surname, S, \text{``}smith''), (\mathbf{job}, \mathbf{S}, \text{``}\mathbf{teacher}''))\}$

$D : \{(name, S, \text{``}mark''), (address, S, \text{``}broad''), (\mathbf{age}, \mathbf{N}, \mathbf{23})\}$

In case, after crossing over, the same parameter appears twice in a chromosome (possibly with different values), one of them is randomly removed, keeping the chromosome valid.

*Mutation of parameter value:* The value of a parameter is randomly changed. One pair in the chromosome is chosen with uniform probability and its parameter value is changed in two alternative ways. Either (1) one character of the 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.

*Mutation by insertion of a new parameter:* A new triple is added to the chromosome. The parameter name is randomly selected among the available parameter names and its value and type are generated randomly.

*Mutation by removal of an existing parameter:* A triple is randomly selected from the chromosome and removed.

*Generation of random values:* 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 constant 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/java-script 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]`.

## C. Solver based local search

As genetic algorithms are heuristics, they are not ensured to converge to the solution because, for example, the optimization could end in a local optimum. When a local optimum is found, we adopt a local search strategy based on a constraint solver, inspired by concolic testing [3].

Relying on a solver may not be feasible to address the full problem because (i) the search space might be too big or involve too complex constraints; and (ii) we do not have a full specification of the path to execute but just of a portion of it, i.e., the target branches. However, a solver should be adequate when the big problem is partially solved by the genetic algorithm, and only local improvements are demanded.

In particular, considering the running example, the genetic algorithm should generate quite easily a test case like this:

$$\{(username, S, \text{“}ddeerer''), (password, S, \text{“}xxsdsed''), (password2, S, \text{“}dded33e'')\} \quad (7)$$

A similar test case would traverse most of the target branches, but the last one. The last branch, namely branch 10-11, is quite hard because it requires two strings to be equal. As strings are randomly generated, it is very unlikely that they contain exactly the same characters. However, a solver will address quite easily this kind of constraint.

*Symbolic values:* Path conditions are collected in terms of constraints on inputs, so we trace the symbolic values of program variables in terms of their relation with input values. This is done by instrumenting the code. Figure 3 shows the instrumented version of the running example (Figure 1) to trace symbolic values. After each assignment, a dynamic map, i.e. *SYMB*, is updated with the new symbolic value of the assigned variables, in this way:

- The symbolic value of an expression is the string representing the quotation of the expression, where variables are replaced with their symbolic values. However, in case of operators not supported by the solver (e.g., non-linear arithmetic) we resort to concrete values;
- The symbolic value of an input is the name of the input parameter;
- The symbolic value of another variable can be found in the *SYMB* map.

A simple example is on line 1, after the assignment *$user = $_GET[”username”];*, the symbolic value of *$user* is set to the input parameter, i.e. *SYMB($user) = ”GETusername”*.

A case where we have to resort to the concrete value is on line 5, in fact the solver can not cope with regular expressions. Since the variable *$user* is both used and defined, we introduce the local variable *$tmp* for the intermediate result, that is later re-assigned to *$user*. The assignment *$tmp = htmlspecialchars($user)* results in updating the symbolic value *SYMB($tmp)* with the concrete value of the expression, i.e. *SYMB($tmp) = htmlspecialchars($user)*.

*Symbolic constraints:* Using symbolic values of program variables, symbolic constraints can be generated and collected at decision points as shown in Figure 4. A normalization step is required to add explicit clauses as in the case of `if` statements without else (e.g., statement 5). Conditions are collected on decision points, they are the strings resulting from the quotation of the decision conditions, where variables are replaced by their symbolic values from *SYMB*. However, when conditions involve operators not supported by the solver, we resort to concrete values. Conditions are collected together with the index

```
 1 $user = $_GET["username"];
   SYMB($user) = "GETusername";
 2 $pass = $_GET["password"];
   SYMB($pass) = "GETpassword";
 3 $pass2 = $_GET["password2"];
   SYMB($pass2) = "GETpassword2";
 4 if ( strpos($user, "<script") )
 5     $tmp=htmlspecialchars($user);
       SYMB($tmp) = htmlspecialchars($user);
       $user = $tmp;
       SYMB($user) = SYMB($tmp);
 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 3. Example of instrumentation to trace symbolic values.**

of the traversed branch using function *COND*. They are collected as follows:

- **If statements:** The condition in the *if* is turned into a symbolic constraint and collected in the *then* branch. On the *else* branch, the same condition is negated before being collected, using the negation (i.e., !) unary operator.
- **Switch statements:** Decision points with just two alternative branches can be inverted when a wrong path is taken. So *switch* statements are converted to the corresponding *if-then-else* chain and then instrumented as in the previous case.
- **Cycles:** We collect two symbolic conditions, one for cycling and one for exiting from the cycle. Inside cycles, proper instrumentation is also applied to *break* and *continue* statements as they are also branches.

At the end of the PHP program (and at each *exit* statement) all the path constraints collected at run-time are appended to the current page, using a recognizable markup. Constraints are labeled with the branches they control.

An execution of the running example on input values shown in equation (7), would cover three out of the four target branches and it would collect the path conditions shown in Table I. As the user name is considered valid, the first two conditions will be collected at branch 4-6 and 6-8. Then, the password passes only the first check on length, so the corresponding condition is collected on branch 8-10. The last condition on *password* not matching

```
 1 $user = $_GET[”username”];
   SYMB($user) = ”GETusername”;
 2 $pass = $_GET[”password”];
   SYMB($pass) = ”GETpassword”;
 3 $pass2 = $_GET[”password2”];
   SYMB($pass2) = ”GETpassword2”;
 4 if ( strpos($user, ”<script”) )
      COND(4,5) = ”strpos(” . SYMB($user) . ”,<script)”;
 5    $tmp=htmlspecialchars($user);
      SYMB($tmp) = htmlspecialchars($user);
      $user = $tmp;
      SYMB($user) = SYMB($tmp);
   else
      COND(4,6) = ”!strpos(” . SYMB($user) . ”,<script)”;
 6 if ( $user in $users )
      COND(6,7) = $user in $users;
 7    echo ”username␣already␣taken”;
   else
      COND(6,8) = $user in $users;
 8    if ( strlen($pass) < 5)
         COND(8,9) = ”strlen(” . SYMB($pass) . ”)<5”;
 9       echo ”password␣too␣short”;
      else
         COND(8,10) = ”!strlen(” . SYMB($pass) . ”)<5”;
10       if ( $pass == $pass2 )
            COND(10,11) = SYMB($pass) . ”==” . SYMB($pass2);
11          new_user($user, $pass);
12          echo ”new␣account␣for”;
13          echo $user;            //sink
         else
            COND(10,14) = ”!” . SYMB($pass) . ”==” . SYMB($pass2);
14          echo ”passwords␣do␣not␣match”;
```

**Figure 4. Example of instrumentation to collect symbolic constraints.**

*password2* is collected on branch 10-14.

| Branch | Condition | Target branch |
|---|---|---|
| 4-6 | $!strpos(GETusername, \text{"} < script\text{"})$ | 4-6 |
| 6-8 | $true$ | 6-8 |
| 8-10 | $!strlen(GETusername) < 5$ | 8-10 |
| 10-14 | $!GETpassword == GETpassword2$ | **10-11** |

**Table I. Symbolic conditions collected during execution.**

*Constraint selection:* Before passing path conditions to the solver, they must be filtered, depending on the branches where they have been collected. In fact, when a constraint is collected on a decision point that makes the execution traverse a target branch, such constraint represents an important property of input values that we should preserve when generating new values. Conversely, if a constraint is collected on a branch that makes the execution diverge from a target branch, new input values should be able to avoid such condition, and take the missed target branch.

Branches that label the collected path conditions are compared to the target branches, in order to identify what is the first branch that diverges from a target branch, which is called *divergent branch*. The list of constraints to pass to the solver are composed of:

1) All the conditions that precede the divergent branch, because they should be valid also for the new input values;
2) The negation of the condition labeled by the divergent branch, as the new input values should take the opposite path; and
3) Only conditions that involve input values, the others are discarded (e.g. for branch 6-8).

The conditions following the divergent branch are ignored, and not passed to the solver.

Comparing the conditions collected on the example and the target branches (see Table I), we can see that the divergent branch is the last one, i.e. 10-14, as it deviates from the target branch 10-11. So we merge the first three conditions with the negation of the fourth one. After filtering, they correspond to the subsequent conditions:

$$!strpos(GETusername, \text{"} < script\text{"}) \tag{8}$$

$$!strlen(GETusername) < 5 \quad (9)$$
$$GETpassword == GETpassword2 \quad (10)$$

This last condition (i.e., Equation 10) requires the two passwords to match. It is needed to improve the current best individual and achieve the final solution.

Equations 8-10 are passed to the solver and, since they are satisfiable, a new set of input values is returned, for example:

$$\{(username, S, \text{``}ddeerer''), (password, S, \text{``}xxsdsed''), (password2, S, \text{``}xxsdsed'')\}$$

These new input values result in a substantial improvement over the previous best results elaborated by the genetic algorithm as one more target path is covered. After using the solver to escape from a local optimum, the genetic algorithm continues until a new local optimum is found.

## IV. Tool support

The approach has been implemented in a tool prototype. First of all, static analysis is applied to extract candidate vulnerabilities, then the code is instrumented to trace symbolic values and path constraints. At this stage the genetic algorithm searches for input values that make the execution traverse the vulnerabilities. Solver based local search is used to improve the search whenever a local optimum is found.

### A. Static analysis

Taint analysis is applied using Pixy [4], an open source tool for static analysis of PHP code. Pixy reports the sequence of assignments that make a tainted value reach vulnerable sinks (skipping sanitization).

A static analysis module is implemented in Txl [5], to extract control dependencies from PHP code and to translate the chain of assignments into target branches. This is done by steps. First of all, branches are extracted to form the Control Flow Graph (CFG). Then the CFG is processed by flow analysis to compute control dependencies. At this stage, we identify all the branches that control the execution of statements in the chain. These branches represent the target branches to be used by the genetic algorithm.

### B. Instrumentation

The application under analysis is instrumented using a Txl transformation. Probes are inserted on branches so that when a branch is traversed, the corresponding probe is triggered. Data about the traversed branches are stored in the resulting web page as an easily recognizable annotation.

Symbolic values are traced by extending the PHP back-end with a module implemented in C. This module exposes to the PHP code some brand new functions, to update and retrieve symbolic/concrete values (i.e., *SYMB*). In particular, concrete values are resorted to, whenever symbolic values are not available or not up-to-date because, for example, they are changed by code implemented in C, not instrumented by us. The same module also exposes functions to collect path constraints labeled by branch indexes (i.e., *COND*) and to print all of them at the end of the execution.

These functionalities have been implemented as part of the PHP back-end to be able to work with pointers (not available in PHP).

### C. Genetic algorithm

The genetic algorithm is implemented in Java and run on the instrumented code. Parameters of the genetic algorithm are set up according to what is proposed in the literature [6]. In particular an elitist approach is adopted, with the 10% of the best individuals kept alive across generations. The population is composed of 70 individuals and evolution is ran over 500 generations. The cross over probability is set to $P_c = 0.7$ and mutation probability to $P_m = 0.01$.

The genetic algorithm works as a client application that simulates a web-browser. HTTP requests are sent to the instrumented server. Requests encode individuals as parameter values passed either by GET or by POST. For each request, a page is processed and returned by the server, together with data on traversed branches and path constraints, to be used for the computation of the fitness function.

When the genetic algorithm does not achieve any improvement after 50 consecutive generations, we assume that a local optimum is found so the execution switches to the solver.

### D. Solver

We use the Yices[2] solver to solve path constraints. Since Yices accepts a Lisp-like syntax, while constraints are collected using using PHP syntax, we implemented (in Txl) a PHP-to-Yices translator. Yices does not support the string type, so the translator also takes care of encoding strings into bit-vectors, supported by the solver. Once constraints are available in the correct syntax, they are solved by

[2]Ices website: http://yices.csl.sri.com/

| Page | # | Target branches covered | | | |
|---|---|---|---|---|---|
| | | 0% | 38-46% | 50-75% | 100% |
| add_comment | 1 | | | 1 | |
| add_gallery | 6 | | 4 | 1 | 1 |
| admin | 1 | 1 | | | |
| delete_gallery | 4 | 2 | | 1 | 1 |
| modify_gallery | 6 | 3 | | 1 | 2 |
| modify_phid | 6 | 3 | | 1 | 2 |
| slideshow | 9 | | | | 9 |
| upload | 3 | | 2 | | 1 |
| view | 2 | 1 | | 1 | |
| **Total** | 38 | 10 | 6 | 6 | 16 |

**Table II. Test case coverage by page.**

Yices. If they are satisfiable, new values are translated into a new individual that is added to the current population.

## V. Empirical results

In order to demonstrate the feasibility of our approach, in this section we apply it on case study application, 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.

To collect empirical data, we had to set up the application in a working environment. First of all we deployed the application on the local web server and we ran the installation scripts to configure the application parameters, such as administrator password and the folder to store gallery pictures. Then, we populated the gallery folder with sample data, otherwise some functionalities would not have been available, e.g. to modify existing galleries.

As different pages support parameters passed in different ways, we run the prototype twice, once configured to encode parameters by GET, then by POST, and we took the union of the results. Each complete run took respectively 46 and 76 minutes.

### A. Vulnerabilities by web page

Empirical data on the generated test cases are shown in Table II. Vulnerabilities are grouped according to the vulnerable pages where they are located (column 1). The table reports the number of vulnerabilities per page (column 2) and how many of them are partially or totally covered (columns 3-6). Columns 3 to 6 represent the quartiles of the distribution of coverage.

Static analysis reports 55 candidate vulnerabilities. However, among them, 17 vulnerabilities are trivial to cover, because all the assignments in their data chain are at top level. As statements are not guarded by any branch condition, any test case would trivially execute them, so we exclude them from the analysis. Out of the remaining 38 non-trivial vulnerabilities, 16 can be completely covered by the generated test cases, while for 6 vulnerabilities we still achieve a high coverage (between 50% and 75% of the target branches). For the remaining vulnerabilities the coverage is more limited, for 6 of them we cover between 38% and 46% of the target branches, and the other 10 test cases cover no branch.

We performed further manual inspection of those pages where vulnerabilities were not fully covered. We found that the vulnerability of *admin.php* can not be executed after the application is successfully installed. One of the uncovered vulnerabilities of *modify_gallery.php* is also infeasible, while the other 3 depend on data from the database, e.g. picture file names or image size. As we exclude permanent storage from input search space, our approach is not able to generate test cases to satisfy such conditions. Similar constraints on the database prevent the generation of test cases for 3 vulnerabilities of *modify_phid.php* and for the 2 vulnerabilities of *view.php*. Vulnerabilities of *add_comment.php* and *delete_gallery.php* are in the error management code that handles unexpected data coming from the database.

The remaining vulnerabilities (5 on *add_gallery.php*, 1 on *modify_phid* and 2 in *upload.php*) are not covered by the generated test cases because of limitations of our approach, even if they are feasible.

### B. Vulnerabilities by cardinality

To study how our approach works when the amount of target branches increases, we classify vulnerabilities also according to their cardinality, i.e. by the amount of target branches. Table III shows vulnerabilities grouped in conformity with the quartiles of the cardinality. The first class contains vulnerabilities with only one or two target branches. The second and third classes contain vulnerabilities respectively with three and four branches. The last class contains the largest vulnerabilities, with 5 to 13 target branches.

Intuitively we could consider more easy to test those vulnerabilities with smaller cardinality, however empirical results suggest a different trend. In fact, a large amount of the smallest vulnerabilities (7 out of 15) are totally uncovered (0% branches). As the cardinality increases, we achieve bigger coverage, as most of the vulnerabilities with 3 and 4 branches are fully covered (100% branches), respectively 5 out of 7 and 4 out of 7 vulnerabilities. This trend is only partially confirmed on the last class, that contains vulnerabilities with the largest cardinality. Even if none of them is totally uncovered, most of them are just partially covered.

The worst performances are shown on extreme cases,

| Cardinality | # | Target branches covered | | | |
|---|---|---|---|---|---|
| | | 0% | 38-46% | 50-75% | 100% |
| 1-2 | 15 | 7 | | 3 | 5 |
| 3 | 7 | 2 | | | 5 |
| 4 | 7 | 1 | | 2 | 4 |
| 5-13 | 9 | | 6 | 1 | 2 |
| **Total** | 38 | 10 | 6 | 6 | 16 |

**Table III. Test case coverage by cardinality.**

when the cardinality is very large or very small. Manual inspection on vulnerabilities that involve a large set of target branches revealed that they are actually feasible, but our tool could not generate test cases for them. When many constraints are imposed on the control flow, covering it becomes very difficult by random input mutation. Moreover, the solver could not help on too complex conditions, when we have to resort to concrete values.

## VI. Related works

The adoption of static analysis for identifying vulnerabilities was initially proposed as a way to support manual inspection [7]. Initially called type-state analysis [8], taint analysis has been largely adopted to detect inadequate or missing input validation, resulting in cross-site scripting [1] [4], SQL-injection [9] and buffer overflow [10] vulnerabilities. In order to mitigate inaccuracy of pure taint analysis due to conservativity, more sophisticated analyses have been integrated, such as string analysis [1], program slicing [11], points-to analysis [12] and model checking [13]. The present paper takes a different direction, instead of going for a more complex but still inaccurate static analysis, candidate vulnerabilities reported by static analysis are subject to search based software engineering to find test cases that execute them.

Instead of statically searching for security faults, other approaches resort on monitoring the application (often calling it *dynamic analysis*) while it runs in production. For instance, in [14] the application execution is monitored by several anomaly reasoners and a firewall blocks those interactions classified as abnormal. The principal drawback in monitoring is represented by run-time and memory overhead. Program slicing has been proposed by Walter et al. [15] to limit code instrumentation only to the actual vulnerable part.

Genetic search has been used on procedural code [16], to generate new test cases and improve coverage, considering the distance from an uncovered structural properties (e.g., a branch or a def-use chain) as fitness function. This approach has been extended [17] to test object oriented code, by searching not only for input values, but also for a method invocation sequence. Del Grosso et al. [6] applied genetic algorithms to identify tests to expose buffer overflows using a complex fitness function. Tests are searched with high statement coverage, that execute lots of vulnerable statements and deeply nested code, and that write data as near as possible to the buffer boundary. An evolutionary inspired approach has been used also by Zulkernine et al. [18] on security, for a totally different purpose. Network scanner rules are searched that maximize the ability in revealing and classifying intrusions.

Concolic execution [3], [19], [20], [21] is an alternative popular approach that mixes symbolic and concrete execution to generate test cases. The code under test is executed initially on random inputs and symbolic constrains are collected at run-time on assignments and decision points (branches). Symbolic constraints are selectively negated and passed to a solver that generates new test cases, to try to cover previously uncovered execution flows.

Inkumsah and Xie [22] connected concolic testing and genetic algorithms to maximize test suite coverage. They shown improvements on coverage of Object Oriented code when (i) a genetic algorithm follows concolic testing and (ii) concolic testing follows a genetic algorithm, when compared with the two techniques alone. Instead of a two-steps approach, we propose a more tight integration, with multiple swaps between the two search strategies. In fact, use the genetic algorithm for global search and the solver for local search. Our objective is also different. While the purpose of Inkumsah and Xie is to generate a test suite that achieves a generic large coverage of the system under test, we target a single and selected condition (i.e., target branches), identified using static analysis.

## VII. Conclusion

In this paper we presented a novel approach to generate test cases that cover Cross-Site Scripting vulnerabilities on web application. Despite the fact that we focus on PHP, this kind of security threats is very general, as it affects a large number of web applications, written in any programming language.

Our test cases execute a vulnerable path, but they do not represent actual attacks, as they do not try to inject HTML code in the final page. Nonetheless, they represent a valuable help for the developers, because tests can be used to understand the vulnerability problems, before fixing them. This is a major improvement over the limited support offered by static analysis.

As generation of test cases relies on static analysis, it suffers its limitations. In particular no test will be ever provided for false negatives, vulnerability missed by static analysis, and false positives, vulnerabilities that consists in unfeasible paths. However, static analysis is conservative and just few cases should be missed. Moreover, we limit the number of iterations of the genetic algorithm, so as to avoid losing too much time on infeasible paths.

As future works we will improve our approach to cope with more and more complex symbolic constraints, so as to limit the need to back up to concrete values. Moreover, we plan to extend our approach and turn our test cases into actual attacks, tainted values should bring portions of HTML to be injected into the final page. We will also investigate on how to adapt the preset approach to cope with other kind of vulnerabilities, for example including database values in the search scope.

## References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

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

[6] C. D. Grosso, G. Antoniol, E. Merlo, and P. Galinier, "Detecting buffer overflow via automatic test input data generation," *Computers and Operations Research*, vol. 35, no. 10, pp. 3125 – 3143, 2008, special Issue: Search-based Software Engineering.

[7] B. Chess and J. West, *Secure programming with static analysis*. Addison-Wesley Professional, 2007.

[8] R. Strom and D. Yellin, "Extending typestate checking using conditional liveness analysis," *Software Engineering, IEEE Transactions on*, vol. 19, no. 5, pp. 478–485, May 1993.

[9] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, "Securing web application code by static analysis and runtime protection," in *WWW '04: Proceedings of the 13th international conference on World Wide Web*. New York, NY, USA: ACM, 2004, pp. 40–52.

[10] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner, "Detecting format string vulnerabilities with type qualifiers," in *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2001, pp. 16–16.

[11] J. Krinke, "Information flow control and taint analysis with dependence graphs," in *3rd International Workshop on Code Based Security Assessments (CoBaSSA 2007)*, 2007, pp. 6–9.

[12] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis," in *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2005, pp. 271–286.

[13] L. Wang, Q. Zhang, and P. Zhao, "Automated detection of code vulnerabilities based on program analysis and model checking," in *Source Code Analysis and Manipulation, 2008 Eighth IEEE International Working Conference on*, Sept. 2008, pp. 165–173.

[14] C. Criscione and S. Zanero, "Masibty: an anomaly based intrusion prevention system for web applications," in *Black Hat Europe 2009*, 2009.

[15] W. Chang, B. Streiff, and C. Lin, "Efficient and extensible security enforcement using dynamic data flow analysis," in *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*. New York, NY, USA: ACM, 2008, pp. 39–50.

[16] R. Pargas, M. J. Harrold, and R. Peck, "Test-data generation using genetic algorithms," *Journal of Software Testing, Verifications, and Reliability*, vol. 9, pp. 263–282, September 1999.

[17] P. Tonella, "Evolutionary testing of classes," in *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*. New York, NY, USA: ACM, 2004, pp. 119–128.

[18] R. H. Gong, M. Zulkernine, and P. Abolmaesumi, "A software implementation of a genetic algorithm based approach to network intrusion detection," in *Sixth International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, May 2005, pp. 246–253.

[19] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su, "Dynamic test input generation for web applications," in *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*. New York, NY, USA: ACM, 2008, pp. 249–260.

[20] R. Majumdar and K. Sen, "Hybrid concolic testing," in *ICSE '07: Proceedings of the 29th international conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 416–426.

[21] D. E. Cristian Cadar, Daniel Dunbar, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2008, pp. 209–224.

[22] K. Inkumsah and T. Xie, "Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution," *Automated Software Engineering, International Conference on*, vol. 0, pp. 297–306, 2008.