Hunting Bugs Inside Web Applications*

David Hauzar and Jan Kofroň

Department of Distributed and Dependable Systems
Faculty of Mathematics and Physics
Charles University in Prague, Czech Republic

Abstract. In recent years, focus of business world has been moved towards the Internet. Web applications provide a generous interface non-stop thus offering to malicious users a wide spectrum of possible attacks. Consequently, the security of web applications has become a crucial issue. The state-of-the-art tools for bug discovery in languages used for web application development, such as PHP, suffer from a relatively high false-positive rate and low coverage of real errors; this is caused mainly by the dynamic nature of such languages and path-insensivity of the tools. In this paper, we will demonstrate weaknesses of the tools and describe our novel approach addressing these issues. It combines path-sensitive static analysis, concrete and symbolic execution, literal analysis, taint analysis and type analysis to handle the dynamism of PHP. We show how our technique handles some of the situations where other tools fail on examples.

1 Introduction

Recently, as business world has moved its focus towards the Internet, a number of applications have been moved on-line, and this trend is far from being dropped. According to CENSUS [16], the US retail sales in 2010 realized on-line are estimated to reach over 160 billion US$. Safety and security of the web applications involved in such transactions is therefore of highest interest of (not only) business people.

A typical web application is available and running in the 24/7 mode, thus not putting any time pressure on hackers and malicious users trying to exploit security holes inside them; a quite generous interface these applications provide further widens the hackers' field. Amongst the 25 most common programming errors, those specific to web applications form a significant part of this group [5]; the examples include improper neutralization of SQL commands, cross-site request forgery, and missing authorization.

The most common programming language used at the server side is PHP [12]. Although it is currently losing a bit of market share, there are a huge number of applications written in this language that deserve attention and effort towards their security [15]. While the current, fifth version of PHP was released already

* This work was partially supported by Charles University Foundation grant 431011 and the grant SVV-2011-263312.

some time ago, it was the fourth version more than ten years ago that introduced objects into the language making it more convenient for development of larger projects. Despite this fact, however, PHP features many special attributes that make it different from common programming languages, especially in the sense of dynamism. The examples are inclusion of a file specified by a runtime-computed filename and the *eval* construct allowing runtime construction of code that is executed afterwards.This makes it hard or sometimes even impossible to apply the same techniques and tools for finding bugs or for correctness verification as in the case of "non-web" programming languages.

1.1 Problem statement and goals

Security issues related to web applications can have (and they do) significant impact on the trustworthiness and reliability of on-line transactions, not only in the business domain; consider, e.g., leakage of classified information from a "secured" database. A lot of attention have been paid to the development of methods and tools that would help in debugging of these applications or establishing their correctness in some sense, since the methods for "non-web" languages cannot be easily applied. The current state-of-the-art tools, however, still suffer from low error coverage, a relatively high false-positive rate, and often also from a weak support of language constructs, such as classes, dynamic includes, and the *eval* statement [8, 17].

In this paper, we propose a method for identification of bugs inside web applications written in PHP caused by data flow of unsanitized inputs from the user to sinks (SQL queries, URL constructions, output in general, etc.). We describe our method and demonstrate benefits of our approach over those present in related tools on a simple example.

The rest of the paper is structured as follows: In Sect. 2, we describe the most problematic errors inside of web applications written in PHP. In Sect. 3, we discuss properties of the tools in this area and present the results of running one of them on an example. Sect. 4 proposes our approach to analysis of PHP source code, and demonstrates our approach on examples. Sect. 5 summarizes the paper and proposes directions for future work.

2 Errors inside web applications

A huge number of security holes inside web applications can be grouped together under common roof that is characterized by allowing data propagation from user input (e.g. form fields on a web page) into database queries, URLs, JavaScript code, etc. (*sinks*) without checking for being malicious. These can be prevented by filtering user input, escaping the output, and by keeping track of the input data. Filtering input is a process of preventing invalid data from entering the application. Blacklist filtering excludes malicious data, while whitelist filtering excludes all data except for that explicitly listed; thus it is distinctively safer

than blacklist filtering due to the possibility of a missing item in the list. Escaping or encoding special characters the application outputs prevents injection of malicious code or data. Keeping track of the input data comprises of identifying those data that the input can influence, identifying the influence points, and determining the extent of the influence.

2.1 Improper flow of data

To track the flow of data in an application, taint analysis can be used to find the data paths from *sources* to *sinks*. Taint analysis marks data representing sources as tainted and then propagates the taint markings. Data is tainted if it can be influenced by the user and it is not sanitized.

Sources are program points through which data enters a program. An attacker can provide malicious data as user input encoded in URL and HTTP headers, data stored in cookies, and elements of the `$_SERVER` array. It is also appropriate to track data obtained from a filesystem, session data store, and output of database queries. Even though in theory this data should be safe, in practice there are security exploits giving the attacker control over this data. It is clear that some sources represent larger security threat than others and it is necessary not only to "taint" data but also to distinguish between different taint sources.

Sinks represent the program points (commands) where inappropriate input can cause a security threat. Examples of the sinks include commands for sending data to a browser, sending data to a database, executing data, names of dynamically included files, opening files, and executing arbitrary system commands.

The process of sanitization is specific for each kind of sources and sinks. Moreover, the extent of sanitization depends on the level of required protection and also on the application logic.

Basic level of protection can be achieved by escaping the output. Escaping the data that can be manipulated by an attacker using a built-in function `htmlentities` prior to sending them to a browser prevents cross site scripting (XSS) attacks while SQL injection attacks can be prevented by escaping the data before sending them to a database—e.g., by using a built-in function `mysql_real_escape_string` in case of MySQL database.

Escaping the output does not protect against sensitive information leakage, though. Consider an application that lets a user to choose among multiple topics and then displays messages related to selected topic. Consider that some topics are available only for registered users. If the application does not filter the topics for non-registered users, even a non-registered user can manipulate the input and see messages available only for registered users. Filtering the input should be employed for example whenever input data reaches a database query, dynamic include, operations that executes the input, or open a file. Next, filtering the input prevents inserting invalid data into a database. The user input going from a source to a sink is considered as sanitized with respect to whitelist filtering if the data in source have a finite number of possible values. It is considered as sanitized with respect to blacklist filtering if there are some restrictions over

possible values in the source. Note that the range of possible values in the source and restrictions over possible values depends on the specification of a particular application and should be inspected by a developer or a security auditor. Also note that symbolic execution is necessary to determine this information.

Filtering itself does not assure the absolute security of the application. Consider an application that tracks the user name of a user that is logged in via a URL (or a hidden form field, or a cookie), reads an e-mail address from a form and then associates the e-mail with the user name in the database. The fact that the user name is tracked in the URL means that it is a part of the input, the attacker can manipulate it and change the e-mail of another user. Note that escaping or filtering the user name does not prevent this vulnerability. This kind of attacks is called semantic URL attacks, spoofed form submissions, and spoofed HTTP requests. An indicator of vulnerabilities that can lead to such attacks is updating information that is identified by the data that can be manipulated by the user. In these cases, the developer should thoroughly check whether it is not a security issue.

A common fix to the vulnerabilities mentioned above is to use a session mechanism (via URL or cookie). However, even this protection can be broken by so called Cross-Site Request Forgeries (CSRF) attacks. These vulnerabilities are indicated by use of data that can be manipulated by the user in a critical command that can be executed only under certain privileges. The fix of the example is to generate a random token prior to requesting data from the user, store the token on the server, embed it to the URL or the form and then to check whether the request contains the token.

3 State of the art

Huang et al. [9] developed intraprocedural static analysis for PHP applications in WebSSARI tool. Xie [20] discusses the limitations of his approach, in particular that it is interprocedural and it does not model dynamic features such as dynamic arrays, objects, dynamic variables, and dynamic includes. To identify vulnerabilities, the approach performs taint analysis. Their approach does not consider a custom sanitization, the data are considered to be sanitized if they are processed with a specified sanitization function.

The approach of Xie et al. [20] uses inter-procedural analysis to find SQL injection vulnerabilities in PHP applications. They model automatic conversion of particular scalar types, uninitialized variables, simple tables, and include statements. However, they leave important parts of PHP unmodeled. In particular, they do not model references, object oriented features of PHP, and they ignore recursive function calls. To model sanitization process, the approach performs taint analysis. Sanitization can occur via calls to specified sanitization functions, casting to safe types, and regular expression match. That is, the approach keeps a database of sanitizing regular expressions.

Wasserman et al. [18, 19] use grammar-based string analysis following Minamide [11] to find the set of possible string values of a given variable at a given

program point and gains this information to detect SQL injections. However, the employed analysis has an incomplete support for references and does not track type conversions.

Pixy [10] performs taint analysis of PHP programs and it provides information about the flow of tainted data using dependence graphs. It uses literal analysis to resolve include statements and perform alias analysis. However, it does not model aliases between variables and members of an array. Next, Pixy lacks type inference, does not model variable-variables and variable-indices and provides only a very limited support of object oriented features. Moreover, similarly to WebSSARI it performs only simple taint analysis and does not consider custom sanitization routines.

Balzarotti et al. [3] extended Pixy to perform the analysis of the sanitization process and thus are able to deal with a custom sanitization. They combine static and dynamic analysis techniques to verify PHP programs. They perform string analysis through language-based replacement and represent values of variables in concrete program points using finite state automata. They also track what parts of strings are tainted. Static analysis that they employ is based on Pixy thus has the same limitations. Moreover, the database of attack strings may not be complete, it can miss vulnerabilities, and can cause false alarms.

Yu et al. [21] developed an automata-based approach for verification of string operations in PHP programs and incorporate the widening operator to tackle the problem of handling variables updated in loops. Similarly as [3], they extended Pixy to perform the analysis of the sanitization process; however, they do not employ the dynamic phase.

Biggar et al. [4] perform context sensitive, flow sensitive, interprocedural static analysis of PHP in order to gain information usable for code optimizatons in their PHP compiler. They combine alias analysis, type inference and literal analysis, model arrays, variable variables, objects, references, scalar operations, casts, and weak type conversions. However, their analysis is closely tailored with their intent—to gain information usable for code optimizations. They gather information that must hold and track information that can hold only in a very limited way. In most cases they approximate information that can hold as unknown. This is not appropriate when the intent is to explore all possible behavior of the code.

The approach of Artzi et al. [1] generates test inputs automatically, monitors web applications for crashes, and validates that the output conforms to the HTML specification. The approach utilizes symbolic execution to capture logical constraints on inputs, based on these constraints, it creates new inputs that would increase the code coverage. By running an application on concrete inputs and using PHP runtime, they avoid the problem of modeling dynamic statements of PHP, undefined semantics of PHP, and their approach is naturally path-sensitive.

Up to our knowledge, a path-sensitive approach to static analysis for PHP has not been yet published. A completely path-sensitive analysis is expensive, however, there has been a lot of research done in the context of other languages,

especially C language to tackle this problem. ESP [6] involves light-weight path-sensitive analysis that selectively joins or separate the contributions according to the different paths based on a heuristics that conditional tests resulting in different property-related behavior should be tracked separately, while other branches should be merged. Note that in the context of taint analysis, property-related behavior would be given by taint statuses of variables. Unfortunately, this heuristics sometimes fail. Dhurjati et al. [7] tackle this problem by iteratively adjusting the merge criterion with new path predicates that are selected using several heuristics. Balakrishnan et al. [2] improve path-insensitive analysis to obtain the effects of path-sensitive analyses by a detection of semantically unfeasible paths using path-insensitive abstract interpreter and performing a sequence of backward and forward runs. Next, they use a technique of syntactic language refinement to exclude semantically unfeasible paths from a program during static analysis. Snelting et al. [13] use program slicing and constraint solving to construct and analyze path conditions—conditions that are defined on program's input variables and must hold for information flow between two program points. Their approach is not complete. The solution of the conditions that they construct can be false witness. That is, it may not lead to intended information flow. Taghdiri et al. [14] tackle this problem by employing counterexample-guided abstraction refinement (CEGAR). They recognize false witness by executing them and monitoring their executions, and eliminating them by automatically refining path conditions in an iterative way.

3.1 Demonstrating existing tool on examples

In this section, we show the limits and weaknesses of the Pixy tool [10] on a few PHP code fragments. We decided to demonstrate just the Pixy tool, since its analysis engine represents, as to our best knowledge, the best analysis engine available for finding vulnerabilities in PHP code. The Stranger [21] tool employs more sophisticated techniques such as string analysis and thus provides more information for vulnerabilities detection; however it is built on the same analysis engine and shares the same limitations.

Due to the fact that Pixy does not model array aliasing correctly, a possible XSS attack is reported at line 6 in Fig. 1. Another issue connected with arrays is the representation of indices. That is, it does not handle variable indices. A different source of false-positives is path-insensitivity; the `$name` variable is sanitized by the routine `htmlspecialchars` in all cases, however, Pixy reports a possible XSS attack at line 15. The last type of a code fragment that causes a false positive alarm and that we present here inheres in omitting the semantics of string operations. The code fragment starting at line 17, simplified for the sake of explanation, includes a file named "*included.php*" (note that the body of the while cycle is not executed at all, since the string `$filename` does not contain the "`..`" substring). Pixy, at this point, since it is not able to evaluate the `$filename` value, simply ignores the `include` statement and reports error at line 24 independently of the content of the included file, which can cause a false positive, as well as a false negative alarm.

```
1   $users[1] = 'fred';
2   $users[2] = $_GET['from_user'];
3
4   $t_users = & $users;
5   // Pixy reports the XSS vulnerability
6   echo $t_users[1];
7
8   $tainted = true;
9   if (tainted) {
10          $name = $_GET['name'];
11  } else {
12          $name = 'bob';
13  }
14  // Pixy reports the XSS vulnerability here
15  echo $tainted ? htmlspecialchars($name) : $name;
16
17  $ext = ".php";
18  $filename = 'included' . $ext;
19  while (strpos($filename, '..')) {
20          $filename = preg_replace('..', '.', $filename);
21  }
22  include($filename);
23  // Pixy reports the XSS vulnerability
24  echo $users[2];
25
26  // Pixy misses the XSS vulnerability here
27  printFirstIndex('tainted', $users[1], $users[2]);
28  function printFirstIndex($varName, $untainted, $tainted) {
29          echo $$varName;
30  }
31
32  $user_ids = 2;
33  // because $user_ids is scalar, the following line does nothing
34  $user_ids[2] = $_GET['user_id'];
35  // Pixy reports the XSS vulnerability here
36  echo $user_ids[2];
```

Fig. 1. Dynamic features of PHP causing false alarms and missed vulnerabilities in Pixy tool.

Besides false positives, Pixy also reports several cases in the false-negative manner. The first PHP construct that is not correctly handled by Pixy are *variable variables*, represented by the `$$varName` at line 29. Another type of false negative stems from insufficient modeling of the type system. The fragment starting at line 32 demonstrates this issue.

The last limitation of Pixy we mention here is that Pixy does not model attributes of objects, so, according to the use of the objects, both false negative and false positive alarms can arise.

At the end of Sect. 4, we demonstrate how these situations are handled when following the approach proposed in this paper.

4 Our approach

In Sect. 2 we claim that most of security errors inside web applications can be prevented by sanitizing data paths from sources of untrusted data to critical commands—sinks. Our approach is to provide the developer with sufficient information so that he/she can assure a correct sanitization. In our case, this means employing an analysis that computes data flow information using *dependence graphs* [3], identify sources of sensitive data, sinks, and at each program point tracking:

- the taint status for each variable at this program point,
- the set of possible values of each variable at this program point,
- the set of conditions defined on program's variables that hold at this program point, and
- the set of possible types of each variable at this program point.

In this section, we describe how we gain this information, how we use it to detect vulnerabilities, and demonstrate it on examples.

4.1 Outline

The main challenge of the analysis is the combination of an arbitrary user input and the dynamism of PHP. To address this problem, we propose the analysis that consists of the following steps:

1. Construction of the control-flow graph (CFG).
2. Static analysis of constructed CFG.
3. Detection of vulnerabilities.
4. A path-sensitive validation of vulnerabilities.

Particular dynamic statements such as dynamic includes, eval statements, and polymorphic calls to methods make a precise construction of CFG impossible. To face this problem, the analysis first resolves only such dynamic statements that are directly given by literals and leaves the remaining statements unresolved. Then, it performs static analysis of obtained CFG. Static analysis gains information about possible values of variables and about possible types and aliases. This information is used to construct more precise CFG that is analyzed again. This can lead to resolution of additional dynamic statements, and this process is iteratively repeated as long as new dynamic statements are resolved.

Next, vulnerabilities are detected based on gained information from static analysis. The employed static analysis is path-insensitive. However, it tracks the fact whether the given information holds on all paths from an entry point of the application to a given program point—are *certain* or only on some paths—are *uncertain*. If there is some information that is uncertain and it is needed to mark a vulnerability, the feasibility of the information is validated path-sensitively.

To improve the precision of CFG construction, each literal value in dynamic statement that is uncertain can be also validated path-sensitively. However, because this validation can be expensive, this is optional.

4.2 Modeling of PHP data structures

Correct modeling PHP data structures constitutes the basis of static analysis. We use a points-to graph similar to that introduced in [4] to model variables, array indices, and object fields. The points-to graph contains three types of nodes. A *storage node* represents a symbol-table, an array, or an object. An *index node* represents a variable, an array index, or an object field. Each index node is a child of a single storage node. Next, a *value node* represents a scalar value and it is a child of a single index node. The points-to graph contains directed edges from a storage node to each index node that belongs to the storage node, directed edges from each index node to the value or the storage nodes representing possible values of the index node. Finally, in the case of aliases, there is a reference edge between two storage or two index nodes.

There is one storage node for each array or object, one storage node for each class holding static fields of the class, one storage node for each called function local symbol table, and one additional storage node for the global symbol table.

Uncertainty is captured in this model as follows: each edge has the certainty information—it represents either certain or uncertain information. Next, each storage node contains an *unknown* field representing index nodes that have statically unknown indices, e.g., $\$a[\$dyn]$ or $\$\$dyn$ where the value of the variable $\$dyn$ is statically unknown. Note that in the latter case, the corresponding storage node can be local or global symbol table. The analysis propagates the uncertainty information through assignments and performs strong updates when possible. A strong update occurs when it is known that an assignment completely overwrites a reference relation or a previous value of a given variable, weak update occurs if the information about the target of the assignment is uncertain. That is, the assignment has more possible targets.

The same way as in [4], the value of an uninitialized node takes its value from the *unknown* field of the appropriate storage node. If the node does not exist, the uninitialized value is set to `null`.

4.3 Static analysis

We use context sensitive, control-flow sensitive inter-procedural path-insensitive static analysis. For each program variable and each program point, we track information about its possible literal values, its types, its taint status, and the set of conditions over variables that must hold in this program poin tand the certainty of this information. At joint points, the combining operation for both literal values of variables and types is union; if some information is not present in all branches, it is uncertain.

Information about the taint status is tracked in the following way. We use different taint markings for different sources of data—i.e., we use different taint marking for an input from http headers, user cookies, a database, session, etc. Note that contrary to other approaches, we do not remove taint status after processing data with any operation. This has two reasons: (1) A correct escaping operation is identified not only by the source of data but also by the sink—the critical command in which data is used. (2) We use the taint information to detect data that can be manipulated by the user. Then, we use this information to detect additional vulnerabilites to those caused by improper escaping. Instead of removing the taint status, we track also the sanitization status. That is, for each sanitization routine and for each taint marking, we track whether the data with the taint marking are sanitized using a sanitization routine. The taint status is propagated through an operation if there exists an input of the operation that is tainted, sanitization status is propagated if all inputs of an operation that are tainted are sanitized.

We use a combination of concrete and symbolic execution to handle operations with literals. For each operation, we use two versions of instructions—explicit and symbolic. If all inputs of an operation are concrete, an explicit version of the operation is used, if some input of the operation is symbolic, the symbolic version of the operation is used. By using concrete operations, we reduce the imprecision caused by modeling of such operations. We face the problem of undefined semantics of PHP in the same way as in [4] and [1] by using the

reference PHP implementation instead of reimplementing the concrete versions of instructions.

As to modeling of the symbolic versions of instructions, we model arithmetic operations as well as operations with strings. For modeling string operations, we use automata-based approach presented in [21]. It makes it possible to handle string concatenation, string replacement, and string restriction. We use string restriction to restrict the value of a string variable based on the branch condition.

4.4 Identifying vulnerabilities

After CFG is constructed and static analysis is completed, we use CFG and alias information to construct the static single assignment form (SSA) of the program. Then, we infer the set of conditions that must hold at each program point using conditional statements (e.g., if and while statements). That is, at the start of a positive branch corresponding to a given conditional statement, the condition corresponding to this statement is added and at the start of a negative branch, negation of this condition is added. At the joint point corresponding to the conditional statement, the condition is removed.

Now, we have all data necessary to identify potential vulnerabilities in the analyzed application. We divide the vulnerabilities into several categories and introduce filters to display security warnings of selected categories only.

To identify vulnerabilities, we have to find all critical commands that input suspicious data. That is, tainted data—those which can be influenced by the user of the web application and *null* values, which can arise due to the bugs in filtering. Next, we analyze the taint and sanitization statuses of this data and identify those that are not properly escaped. For each taint marking and critical command there is a list of escaping operations. The list is stored in the configuration of the program.

We handle custom sanitization routines in the following way. First, by employing literal analysis we are able to prove the absence of given attack patterns in the same way as in [21]. Then, we track sanitization status also for string replacement operations. These operations are potential sanitizers. When the analysis detects a vulnerability, it supplies a list of such potential sanitizers to the developer. Inspection of potential sanitizers can help the developer to reveal false alarms.

Next, we mark all data that are used in critical commands and are tainted or contain *null* values as potentially not filtered. We mark the critical commands that update data identified (e.g., the *where* clause in SQL queries) by tainted data with the highest importance and the critical commands that use tainted data in other way with a lower importance. The developer can then analyze a filtering status of such data by inspecting their possible values. We also make it possible to inspect the restrictions on data by showing the conditions that must hold at a given program point. This way, the developer can discover errors in the blacklist filtering.

Next category of potential vulnerabilites consists of those that can make the CSRF attacks possible. We identify the critical commands that update data and

use data related to the current session and are not guarded by any condition comparing the token in the request with some data stored at the server, e.g., using a session mechanism.

Finally, for vulnerabilities that are caused by the flow of data from a source to a sink through some data path, we construct dependence graphs using the technique of slicing. We reduce this graph to contain only those parts that are relevant for the checked vulnerability. Dependence graphs can be manually inspected by the developer and used to identify vulnerable influence points in the data flow and thus to reveal the cause of the detected vulnerability.

4.5 Path-sensitivity

The analysis described in the previous section is path-insensitive. This is one of the reasons why the vulnerabilities reported by the analysis may not be real (false positives). That is, all paths leading to a given vulnerability can be unfeasible. To deal with this problem, we use certainty information gained during the analysis to identify vulnerabilities that do not depend on path-sensitivity. We report these vulnarebilities immediately together with the reduced dependency graph to the user. Next, for each vulnerability that is uncertain we try to prove the unfeasibility of paths leading to the vulnerability and report only the vulnerabilities corresponding to paths that were not proven to be unfeasible, again, together with the reduced dependency graphs.

To prove the unfeasibility of paths leading to a vulnerability, we identify program points that contribute to the uncertainty of the vulnerability. These program points correspond to (1) join points of branching statements where some branches do not lead to the vulnerability or causes of the vulnerability is different and (2) assignments of data that cannot be certainly identified and that can lead to the vulnerability. An example of the case (1) is at line 7 of Figure 2. The fact whether the variable $\$message$ is tainted is uncertain and it depends on the condition $\$input = 1$. An example of the case (2) is in the line 9 of Figure 2. Again, the fact whether the variable is tainted is uncertain and it depends on the same condition.

We collect the conditions that must hold in order that these program points lead to a vulnerability and the conditions that must hold to reach the critical command corresponding to the vulnerability. Then we use a theorem prover to prove the conjunction of these conditions. If the theorem prover finds a contradiction in these conditions, the vulnerability is unfeasible. If the theorem prover proves the conditions, the vulnerability can be still unfeasible, because of dependencies between variables in the conditions. Using the information from the dependence graph, the information about the solution of the conditions, and symbolic execution, we try to find these dependencies and add the conditions corresponding to these dependencies.

Note that we do not model all operations precisely, hence the approach cannot be complete. That is, false positives can still appear even after the path-sensitive analysis.

```
1  $message = "";
2  $value = "";
3  if ($input == 3) {
4    $message = $GET['message'];
5  } // $input == 3 => $message is tainted
6  $a = array('1', '2', $_GET['user_value']);
7  $value = a[$input]; // $input == 3 => $value is tainted
8  echo $message . $value;
```

Fig. 2. Program points that contribute to the uncertainty of a vulnerability and conditions that must hold for the vulnerability to be feasible.

4.6 Demonstration of our approach—Evaluation

We demonstrate our approach on two examples. First, we show that our approach is capable to handle the code fragments in Fig. 1 correctly. These code fragments contain dynamic statements. Then, we show how our approach can detect more complex vulnerabilities present in the code in Fig. 3.

Handling dynamic features. Using the code fragments in Fig. 1 we show, how our approach models PHP data structures, references, operations, and how it resolves dynamic statements. We also show how it is capable to prove the unfeasibility of a vulnerability detected by the path-insensitive step of the analysis.

If Fig. 1, the statement at line 1 creates an index node representing the variable $\$users$ and makes it a child of the storage node that represents the global symbol table. Because it is the first use of this variable, the presence of the square brackets means that the variable is of the array type. Hence, the statement creates a storage node representing the array and a directed edge from the index node representing the variable $\$users$ to this storage node. The statement also creates an index node representing the index 1 in the array, an edge from the node representing the array to the value node, a value node representing the literal $'fred'$ and an edge between the index node and the value node. The statement at line 2 creates another index node representing the index 2 and a value node, appropriate edges and associates a taint status with the value node. Next, the statement at line 4 creates an index node representing the variable t_users, makes it a child of the storage node that represents the global symbol table and then creates a reference edge between this node and the index node representing the variable $\$users$. This reference is then used at line 6 to find out the appropriate storage node. The index node representing the index 1 in this storage node is not tainted, thus no vulnerability is reported—contrary to Pixy that reports a false alarm.

In the same way as Pixy, path-insensitive phase of our approach detects a potential vulnerability at line 15. However, the information about the taint status is uncertain. The condition that must hold to make the variable *name* tainted is $tainted = true$, the condition that must hold to reach the appropriate critical command is $tainted = false$. The conjunction of these conditions is unfeasible, thus no vulnerability is reported.

All inputs to the operation of concatenation at line 18 are concrete, hence the analysis can use the concrete version of the instruction which results in a

concrete, precise value. Similarly, the analysis uses the concrete version in the case of the operation *strpos* at line 19. Consequently, the value of the variable $\$filename$ is known at line 22, the dynamic include can be resolved. Then, more precise CFG is constructed and the static analysis step is performed with this refined CFG. Note that because we model the important operations symbolically, our analysis is able to handle even more complex cases where inputs of operations are not concrete.

The analysis correctly handles variable variable at line 29. The literal analysis determines that the value of variable $\$varName$ is $'tainted'$; the index node corresponding to this value is then searched in the storage node corresponding to the local symbol table of the function and then also in the storage node corresponding to the global symbol table. Here, the node corresponding to the variable $\$tainted$ is found in the former one and a vulnerability is reported.

The analysis performs type inference, hence, it is known that the variable $\$user\_ids$ is scalar at line 34, and does not perform the assignment. Clearly, this statement is likely a bug, so the analysis reports a warning in such cases.

Discovering more complex vulnerabilities. Now we demonstrate how our approach handles the code in Fig. 3 and how it helps to find vulnerabilities within it. The presented code can be a part of an application that provides an interface for viewing messages of selected topics. Every user has associated one topic that he/she is not allowed to view and has to pay if he/she wants to change this topic. The user can also update his/her email address and under certain circumstances, tracked in the session, insert a message. We will describe all steps of the analysis:

(1) Construction of CFG and static analysis

First, the analysis constructs CFG of the analyzed program. Here, it encounters a problem at line 16. The method to be called depends on the type of the object and it is not known yet. Hence, this call is ignored and since it is the last statement of the "main" part, the construction of CFG is complete. Constructed CFG contains the nodes corresponding to the switch statement and the calls to constructors.

Next, the static analysis phase is performed on constructed CFG. This analysis determines a set of possible types of the object $\$action$ and thus the set of possible methods that could be called at line 16. Using this information, the analysis constructs new CFG. For each possible type, there will be one call to the method *exec_action* in the context of this type. This method contains a dynamic include at line 22. From the first pass of the static analysis phase, a set of possible values of the variable $\$\_GET['action']$ is known. The set equals to $\{'view\_message', 'update\_topic', 'update\_email', 'insert\_message', *\}$. Note that $*$ represents an arbitrary value. Moreover, because there is no additional node in new CFG that manipulates this variable, this information would not be refined by performing the static analysis phase on new CFG. Clearly, the precise value of the variable $\$\_GET['action']$ is tied with the type of the object that calls the method *exec_action* in the *switch* statement; however this informa-

tion is lost at the joint point of the switch statement. There are several options how to proceed: (1) The analysis can employ a path-sensitive step and try to prove unfeasibility of the values in the set. Alternatively, (2) it can resolve the include statement for all values in the set. Finally, (3) it can keep the include unresolved and report a warning to the developer. In the former two cases, the value $*$ corresponds to a vulnerability. It is uncertain, thus it will be validated in the path-sensitive step. Note that if files corresponding to infeasible values would be included and if there would be any vulnerabilites in the included files, the analysis would employ the path-sensitive phase for these vulnerabilities and prove unfeasibility of these values on demand. Also note that such an indirect relation between the type of the object and the value of a global variable not only complicates the analysis but also readability of the code and should be avoided in the first place. We will investigate all these strategies and possibly make the strategy selection configurable. Then, the version of the method *do_action* corresponding to the type of the object *action* is processed. Now, there is no dynamic statement present and hence the construction of CFG is straightforward.

(2) Identifying vulnerabilities

Based on the information gained from the static analysis step, the analysis detects vulnerabilities and divides them into several categories. The first category of vulnerabilities is formed by unescaped data in critical commands. The analysis detects that the variable $\$topic$ used in the critical command *mysql_query* has the taint status from the $\$\_GET$ array and has not sanitization status necessary for this command—*mysql_real_escape_string*. This information is certain, thus this vulnerability is reported to the user together with reduced dependence graph showing the propagation of the taint status. Next, the variable $\$message$ used in the critical command *echo* has the same taint status and again has not sanitization status necessary for this command—*htmlentities* in this case. However, the taint status is uncertain, thus this vulnerability is passed to the path-sensitive phase of the analysis.

The second category of vulnerabilities is formed by those that are present due to improper (custom) sanitization routines. We detect that the tainted variable $\$new\_email$ used in the critical command *mysql_query* at line 66 is not sanitized with any standard sanitizing function. However, it has the sanitization status *preg_replace*. Moreover, the analysis models this function and the intersection of the symbolical value of the variable with the attack patterns is empty. Because the list of the attack patterns is not complete, this does not imply that data are properly sanitized. However, the importance of this potential vulnerability is relatively low, hence, only a warning pointing to the *preg_replace* function is reported.

Another category is formed by the vulnerabilities that are caused by an unfiltered input of critical commands that update data. The variable $\$\_GET['user']$ in the critical command *mysql_query* at line 54 is not filtered, thus a vulnerability is reported. Note that updating data of an arbitrary user is suspicious, however, it could be intended. Next, a vulnerability is reported due to a usage of the variable $\$ban\_topic$ in the same command. At first sight, this variable seems

to be properly filtered, however, there is the *null* value present in the set of its possible values. The absence of the default branch in the *switch* statement starting at line 46 causes the variable $\$ban\_topic$ to be uninitialized and thus produces the *null* value that is then inserted into the database. Note that this enables a user to view messages of an arbitrary topic—see the command that selects messages from the database at line 34. Note that this vulnerability is uncertain and it is thus passed to the path sensitive phase.

The next category of potential vulnerabilities is the use of unfiltered data in other (not data-updating) critical commands. The variable $\$topic$ in the critical command that selects data from the database at line 34 is not filtered and thus reported. Note that in this case, the filtering is done via the condition $message.topic\,!= users.ban\_topic$ in the database query; this report is therefore a false alarm. Next, the variable $\$user$ in the same command is not filtered. This is a real vulnerability, because it enables an attacker to view messages as a different user. Note that the variable $\$importance$ in the same command is filtered, and not reported. All these vulnerabilities are certain, and, therefore, immediately reported. Finally, the variable $\$message$ used in the *echo* command at line 89 is also detected not to be filtered. However, in this case the information is uncertain and thus passed to the path-sensitive phase.

The last category consists of vulnerabilities that make CSRF attacks possible. CSRF vulnerability can arise when a critical command that updates some data uses data tainted with a session status and this command is not guarded by any condition comparing the token in the request with the token stored at the server. This is the case of the command *mysql_query* at line 65, thus a vulnerability is reported.

(3) Path-sensitive validation of vulnerabilities

Uncertain vulnerability corresponds to use of unescaped and unfiltered data in the critical command at line 89. At this program point, the condition $\$insert = false$ holds. We conjoin this condition with conditions gained using program points that contribute to the uncertainty of the vulnerability. The only such program point is the joint point of the *switch* statement starting at line 80. To expose the vulnerability, the variable $\$message$ must be tainted. It gains an uncertain taint status at this program point. That is, it is tainted only if the control flow goes with the first or second branch. This corresponds to the condition $(\$message = 1 \vee \$message = 2)$. The conjunction of these two conditions is passed to the theorem prover. Since it is satisfiable, the unfeasibility of the vulnerability is not detected. However, there is a dependency between the variable $\$user\_status$ and the variable $\$insert$ captured using the *if* statement starting at line 76. This dependency corresponds to the condition:

$$((\$user\_status = 1 \vee user\_status = 2) \Rightarrow \$insert = true) \wedge$$

$$((\$user\_status \neq 1 \wedge user\_status \neq 2) \Rightarrow \$insert = false)$$

This condition is conjoined with the previous (conjoined) condition and again passed to the theorem prover. Now the theorem prover proves the unfeasibility[1] of the vulnerability.

Next, a vulnerability that is uncertain is the vulnerability corresponding to the presence of the *null* value in the set of values of the variable $\$ban\_topic$ at the line 54. The critical command is not guarded by any condition. The variable keeps the value *null*, if the condition ($\$\_GET['ban\_topic'] \neq' world' \wedge \$\_GET['ban\_topic'] \neq' science'$) holds. This condition is satisfiable and there are no data dependencies between any variables in the condition and thus the vulnerability is feasible and it is passed to the user together with the solution of the condition.

```
1  <?php
2  switch ($_GET['action']) {
3    case 'view_message':
4      $action = new ViewMessageAction();
5      break;
6    case 'update_topic':
7      $action = new UpdateBannedTopicAction();
8      break;
9    case 'update_email':
10     $action = new UpdateEmailAction();
11     break;
12   case 'insert_message':
13     $action = new InsertMessageAction();
14     break;
15 }
16 $action->exec_action();
17
18 abstract class Action {
19   abstract protected function do_action();
20   public function exec_action() {
21     include $_GET['action'] . ".inc";
22     $this->do_action();
23   }
24 }
25
26 class ViewMessageAction extends Action {
27   protected function do_action() {
28     $topic = $_GET['topic'];
29     $user = $_GET['user'];
30     $importance = $_GET['importance'];
31     if ($importance < MIN_IMPORTANCE || $importance >=
             MAX_IMPORTANCE) {
32       exit();
33     }
34     $result = mysql_query( "SELECT * FROM messages, users
35       WHERE messages.topic = '" . $topic . "' AND
36         messages.topic != users.ban_topic AND
37         messages.importance <= '" . $importance . "' AND
38         users.name = '" . mysql_real_escape_string($user) . "'" );
39     // a code that displays messages follows
40   }
41 }
42
43 class UpdateBannedTopicAction extends Action {
44   protected function do_action() {
45     if (payment_successfull()) {
46       switch($_GET['ban_topic']) {
47         case 'world':
48         case 'science':
49           $ban_topic = $_GET['ban_topic'];
50           break;
51       }
52       $result = mysql_query( "UPDATE users SET
53             ban_topic = '" . $ban_topic . "'
54           WHERE user = '" . mysql_real_escape_string($_GET
                 ['user']) . "'" );
55     }
56   }
57 }
58
59 class UpdateEmailAction extends Action {
60   protected function do_action() {
61     session_start();
62     $user = $_SESSION['user'];
63     $new_email = $_GET['email'];
64     $new_emaill = preg_replace("/[^A-Za-z0-9.\-@]/","",
             $new_email);
65     $result = mysql_query( "UPDATE users SET
66       email = '" .$new_emaill . "'
67       WHERE user = '" . $user . "'" );
68   }
69 }
70
71 class InsertMessageAction extends Action {
72   protected function do_action() {
73     session_start();
74     $user_status = $_SESSION['user_status'];
75     if ($user_status == 1 || $user_status == 2) {
76       $insert = true;
77     } else {
78       $insert = false;
79     }
80     switch ($user_status) {
81       case 1:
82         $message = $_GET['message'];
83       case 2:
84         $message = substr(0, 15, $_GET['message']);
85       case 3:
86         $message = "You cannot insert messages.";
87     }
88     if ($insert == false) {
89       echo $message;
90       exit();
91     }
92     // a code that inserts the message follows
93   }
94 }
95 ?>
```

Fig. 3. Code fragment that contains several vulnerabilities.

[1] *user_status* has to be 1 or 2 AND *insert* has to be false which contradicts the first part of the last formula.

5 Conclusion and future work

While symbolic execution, literal analysis, taint analysis, and type analysis for PHP applications have been already developed, to our best knowledge, this is the first approach of combining them into a single analysis. The combination of these features is powerful and open doors for precise analysis of dynamic languages such as PHP. The novel contribution of our approach is its path-sensitivity. Even though false alarms caused by path-insensitivity of existing tools were reported, to our best knowledge, no approach that addresses this issue has been published. Last but not least, our approach detects most of vulnerabilities that can be present in web applications by checking whether the user input is properly filtered, output of the application is escaped, and keeping track of the input data. This is done by employing advanced taint analysis, analyzing possible literal values of variables in critical commands, and using dependence graphs to keep track of the data.

We believe that the analysis is scalable, expensive path-sensitive step of the analysis is performed only when it is necessary, e.g. to confirm vulnerabilities that can be false alarms. This is possible because the path-insensitive step of the analysis tracks whether the collected information is precise or can be refined by the path-sensitive step. Moreover, the path-sensitive step can be completely disabled or performed on demand, e.g., to confirm vulnerabilities of particular category. Our approach is not complete. That is, false positives can still appear even after the path-sensitive step. However, precise analysis combined with path-sensitive step and employed vulnerability detection promise both a lower false-positive rate and higher error coverage than related approaches.

Once a prototype implementation is completed, we will evaluate its scalability on real web applications. Future work will also investigate and evaluate existing techniques to analyze and refine path-conditions and possibly adapt them to be usable in the context of PHP applications.

References

1. S. Artzi, a. Kiezun, J. Dolby, F. Tip, D. Dig, a. Paradkar, and M. D. Ernst. Finding Bugs in Web Applications Using Dynamic Test Generation and Explicit-State Model Checking. *IEEE Transactions on Software Engineering*, 36(4):474–494, July 2010.
2. G. Balakrishnan, S. Sankaranarayanan, F. Ivancic, O. Wei, and A. Gupta. Slr: Path-sensitive analysis through infeasible-path detection and syntactic language refinement. In M. Alpuente and G. Vidal, editors, *Static Analysis*, volume 5079 of *Lecture Notes in Computer Science*, pages 238–254. Springer Berlin / Heidelberg, 2008.
3. D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 387–401, May 2008.
4. P. Biggar and D. Gregg. Static analysis of dynamic scripting languages, 2009.

5. Common weakness enumeration. `http://cwe.mitre.org/top25/`.
6. M. Das, S. Lerner, and M. Seigle. Esp: path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 57–68, New York, NY, USA, 2002. ACM.
7. D. Dhurjati, M. Das, and Y. Yang. Path-sensitive dataflow analysis with iterative refinement. In K. Yi, editor, *Static Analysis*, volume 4134 of *Lecture Notes in Computer Science*, pages 425–442. Springer Berlin / Heidelberg, 2006.
8. J. Fonseca, M. Vieira, and H. Madeira. Testing and comparing web vulnerability scanning tools for sql injection and xss attacks. In *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing*, pages 365–372, Washington, DC, USA, 2007. IEEE Computer Society.
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 *Proceedings of the 13th international conference on World Wide Web*, WWW '04, pages 40–52, New York, NY, USA, 2004. ACM.
10. N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: a static analysis tool for detecting Web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 6 pp.–263. Ieee, 2006.
11. Y. Minamide. Static approximation of dynamically generated web pages. In *Proceedings of the 14th international conference on World Wide Web*, WWW '05, pages 432–441, New York, NY, USA, 2005. ACM.
12. PHP—Personal Home Pages. `http://www.php.net`.
13. G. Snelting, T. Robschink, and J. Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM Trans. Softw. Eng. Methodol.*, 15:410–457, October 2006.
14. M. Taghdiri, G. Snelting, and C. Sinz. Information flow analysis via path condition refinement. In *Proceedings of the 7th International conference on Formal aspects of security and trust*, FAST'10, pages 65–79, Berlin, Heidelberg, 2011. Springer-Verlag.
15. Tiobe Software. Tiobe programming community index for june 2011. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`.
16. U.S. Census Bureau News, May 2011. `http://www.census.gov/retail/mrts/www/data/pdf/ec_current.pdf`.
17. M. Vieira, N. Antunes, and H. Madeira. Using web security scanners to detect vulnerabilities in web services. In *Dependable Systems Networks, 2009. DSN '09. IEEE/IFIP International Conference on*, pages 566 –571, 29 2009-july 2 2009.
18. G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 32–41, New York, NY, USA, 2007. ACM.
19. G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. *Proceedings of the 13th international conference on Software engineering - ICSE '08*, page 171, 2008.
20. Y. Xie and A. Aiken. 11. static detection of security vulnerabilities in scripting languages. In *15th USENIX Security Symposium,*, pages 179–192, July 2006.
21. F. Yu, M. Alkhalaf, and T. Bultan. Stranger: An automata-based string analysis tool for php. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 154–157, 2010.