Effect of Coding Styles in Detection of Web Application Vulnerabilities

Ibéria Medeiros, Nuno Neves
LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal
imedeiros@di.fc.ul.pt, nuno@di.fc.ul.pt

***Abstract*—Web application security has become paramount for the organisation's operation, and therefore, static analysis tools (SAT) for vulnerability detection have been widely researched in the last years. Nevertheless, SATs often generate errors (false positives & negatives), whose cause is recurrently associated with very diverse coding styles, i.e., similar functionality is implemented in distinct manners, and programming practices that create ambiguity, such as the reuse and share of variables. The paper presents an analysis of SAT's behaviour and results when they process various relevant web applications coded with different coding styles. Furthermore, it discusses if the SQL injection vulnerabilities detected by SATs as true positives are really exploitable. Our results demonstrate that SATs are built having in mind how to detect specific vulnerabilities, without considering such forms of programming. They call to action for a new generation of SATs that are highly malleable to be capable of processing the codes observed in the wild.**

***Index Terms*—Web application vulnerabilities, static analysis tools, coding styles, SQLi exploitation, software security.**

I. Introduction

Web applications have become an essential resource for accessing services on a variety of subjects (e.g., financial, healthcare) available on the Internet. As their use and applicability in different contexts have increased, it is a priority for organisations to ensure their security against cyber attacks and protect the data they handle. However, despite the efforts that have been made in the security of web applications, specifically on researching of better techniques to identify vulnerabilities in its source code, the number of vulnerabilities exploited has not decreased [1]. For example, the SQL injection (SQLi) vulnerability appears on top of them, representing two-thirds (65%) of cyber attacks of all web attacks [2].

Nowadays, developers use open-source and re-usable (third-party) components in their applications to accelerate their development. Such practice is justified by the easy way of finding code available on the Internet that implements a given software functionality and the constant pressure to be the first to put in the market new and innovative software solutions. However, third-party components have introduced and propagated vulnerabilities in software development [3], and so they can be seen as a mean to increase the number of vulnerabilities. In addition to these flaws, developers themselves can leave the code vulnerable, as they may not have sufficient knowledge about how to write secure code [4].

In light of these facts, static analysis tools (SATs) are the frequently and preferred resources used by organisations to test the security of their applications, as they search and identify vulnerabilities in their source code. Moreover, the outcomes provided by them can help developers to correct the code [5].

For web applications, for example, there are SATs that detect specifically SQLi and cross-site scripting (XSS) vulnerabilities [6][7], as these two vulnerability classes are the ones most exploited and range the OWASP Top 10 [8], and others detect a few more classes of vulnerabilities [9][10]. However, the conducted investigation made over SATs pinpointed they often generate errors (false positives & negatives, FP & FN), whose cause is recurrently associated with diverse coding styles employed by developers, i.e., similar functionality is implemented in distinct manners, and programming practices that create ambiguity, such as the reuse and share of variables.

The paper presents an analysis of SAT's behaviours in the detection of vulnerabilities when they process applications that were coded considering different coding styles and practices, that we call *coding style scenarios*. To the effect, we tested three open-source SATs (phpSAFE, RIPS and WAP) with a use case coded in six scenarios, and analysed their behaviours to understand why they produce FP and FN and in which scenarios they behave correctly. The use case is based on multiple forms contained in a same webpage and their inputs received and processed by a same server-side file. Multiple forms have been a practice used in current websites (e.g., Facebook[1], Elsevier[2]), for login and register a user, for example, and the code that supports and processes them is usually in a single file. The variables used to process the data provided by the forms are the same, as only one form is processed at a time. The paper also presents a discussion of whether the SQLi vulnerabilities detected by SATs as true positives are really exploitable.

Our results demonstrated that SATs are built having in mind how to detect specific vulnerabilities, without considering such forms of programming. Moreover, these forms underlie the SATs errors and, as far as we know, their impact on SATs has never been studied. Such results call to action for a new generation of SATs that are highly malleable to be capable of processing the codes observed in the wild.

The contributions of the paper are: (1) the insights from the behaviours of SATs when they analyze applications written with different coding styles and programming practices; (2)

[1]https://www.facebook.com/
[2]https://checkout.elsevier.com/auth

978-1-7281-8936-9/20/$31.00 ©2020 IEEE
DOI 10.1109/EDCC51268.2020.00027

a discussion about the exploitability of SQLi vulnerabilities detected by SATs as true positives.

II. Context and Related Work

A. Vulnerabilities and False Positives and Negatives

The most well-known and exploitable web vulnerabilities are those related to user inputs, which, if their contents are not sanitised or validated before being used in any function sensitive to them, unexpected outcomes and application's behaviours can occur. This class of vulnerabilities is known by input validation or surface because inputs suffer of sanitisation before being used and they are inserted in the web application through its (attack) surface, e.g., HTML forms. The entries of the application surface are known as *entry points* and the functions that can be exploitable with malicious contents from user inputs are called *sensitive sinks*, or simply *sinks*.

SQLi is one of this kind of vulnerability. It is associated with malcrafted user inputs (e.g., ', OR) that are inserted in SQL statements without any sanitisation and used in a sink that sends them to be executed in the database. Listing 2, lines 9–12, shows an example of a SQLi. The goal of this code is to verify if a user exists in the database, after providing its login credentials – email address and password. The credentials are received through the *$_GET* entry points (lines 9 and 10), used in line 11 to compose the SQL query, and sent to the database by the *mysqli_query* sink (line 12).

Based on the concepts of entry point and sink, a vulnerability is generally defined as being a set of entry points (one or more) used in a sensitive sink, without first being sanitised. More strictly and without considering input sanitisation, we can define a vulnerability as follows:

- *Vul-C1:* a *same* set of entry points that reaches a *same* sink, meaning that a sensitive sink only receives a set of entry points. The example above fits in this definition. That sink only receives those entry points.
- *Vul-C2: distinct* sets of entry points that reach a *same* sink, meaning that a sensitive sink receives different (distinct) sets of entry points from different places of the application. Considering the code of Listing 2, lines {9–11, 38} and lines {17–19, 29, 38} are examples of this category, where the sink of line 38 receives two sets of entry points from different places.

The first definition is the most usual; however, we are more interested in cases that fit the second definition since they can be in the root of false positives and negatives.

For the other hand, generically a false positive is defined as being a vulnerability flagged by a tool wrongly. More strictly, we can envisage three definitions for false positives, namely:

- *FP-C1:* a *real* execution path that starts in entry points and ends in a sink, where one or more entry points are modified by string manipulation operations (e.g., extract substring) or validated (e.g., type checking) before reaching the sink. In the example above, if was used in query of line 11, the firsts characters of the email (line 9) which were extracted by function *substr*.
- *FP-C2:* a *real* execution path, from one or more entry points until a sink, that for some reason the analysis made by the tool was not correct. For example, if a developer codes his/her sanitization functions (e.g., implements a blacklist function), a SAT will not consider them in the analysis, if it is not configured for them. Hence, the tool will output an existent (real) execution path containing such functions as being a vulnerability.
- *FP-C3:* a *false* execution path, from an entry point to a sink, that for some reason the analysis made by the tool was not correct. Due ambiguity of different coding styles and programming practices, the tool does not interpret them correctly, outputting nonexistent (false) execution paths as being vulnerabilities. In Listing 2, lines {17–19, 29, 49} is an example of this type, where is not possible to reach line 49 if the code was executed.

FP-C1 and FP-C2 were already investigated by Medeiros et. al. [10]. Also, FP-C2 can be handled by configuring the tools with sanitisation functions developed by programmers. We are more interested in the FP-C3 case since it is the case that needs to be studied to understand why SATs produce them and give some insights on how to avoid them.

Lastly, for false negatives, we can define them as undetected execution paths for the same reasons as the third case of false positives. We are also interested in understanding this case.

B. Vulnerability Detection through Static Analysis

Static analysis is one of the most used techniques to detect vulnerabilities in source code. The technique requires access to the source code but can normally be performed without having a running version of the application. One common issue with static analysis tools is that they are forced to make approximations in many cases which may lead to false positives (e.g., it is impossible to statically determine the outcome of a conditional statement). Taint analysis is a kind of static analysis that is used in detection of vulnerabilities. It tracks the users inputs and verifies if they are used in a sensitive sink. Several tools implement this technique, having achieved promising results [6][7][9][10] and even offering functionalities such as automatic code correction [10][4][11].

In order to increase the precision of SATs, [12][13][14] combined SATs, and others explored their outputs through machine learning algorithms [15][16]. However, none of them explain why some SATs are better than others in certain cases and indicate which are the causes of this. WAP [10][4][11] also classifies such alerts as being FPs or not by using data mining. Although WAP minimizes FPs, it only pinpoints false positives of FP-C1 and FP-C2, which are related to the presence of functions that sanitize, validate or modify inputs along an execution path; not to coding styles. Another set of works leverage from software metrics to improve the precision of vulnerability detection and the trustworthiness of software systems [17][18]. Medeiros et al. [19] leverages natural language processing (NLP) to classify slices statically but taking into account the order in which the code elements appear in the slice.

III. Use Case and Scenarios

A. Multiple Forms Use Case

The use case we chose involves the reuse and share variables, a common programming practice used by developers to (re)use the same memory space on some parts of the program since they know that some variables will not be used from a given point of the program. Therefore, they can (re)use its memory space (by (re)using the variable name) to allocate other content which will be affected by other program functionality. To illustrate this, the use case we present is the use of multiple forms in a single webpage.

Multiple forms have been a practice adopted in current websites, usually for login or register a user (e.g., Facebook[3]), but they are not limited to this. Although they can be built with different and stylized manners, such as *modal* or *tabulator* window (e.g., IEEE Xplore[4], for institutional sign in, and ForumEngine[5]), or even by the typical way (e.g., Elsevier[6]), the HTML code that supports them is always the same, i.e., multiple form elements for representing the different forms, each one containing input elements (e.g., buttons, input box, check box) for designing the various pieces of the form.

Another aspect that is common to multiple forms is that the server-side code (e.g., PHP) that receives and processes them is usually in the same file. Moreover, since only a form can be processed at a time, developers use the same variables to receive the user data provided through the forms, and some parts of the code are common to both. Besides these practices, which are not incorrect and allow to reduce the system resources used, the way that they are coded can differ between developers, but for the same purpose.

Figure 1 shows an example of multiple forms for logging or registering a user, illustrating its input elements, and Listing 1 presents the HTML code that supports them. To facilitate the read of the HTML, we only present the necessary code to characterize the use case.

Both forms have in common the `E-mail Address` and `Password` input elements, named `email` and `email_pass` (lines 3 and 5, and 13 and 15 in Listing 1), and, therefore, they have in common the entry points `$_GET["email"]` and `$_GET["email_pass"]`. In addition, the PHP file `LogReg.php`, specified in the `action` attribute on both forms (lines 1 and 9), will process the data provided by them. Each form has a button to submit the data (lines 6 and 18), which will allow to distinguish each form in the PHP file – `$_GET["login"]` and `$_GET["register"]` for login and register forms, respectively.

Listing 2 presents the PHP code of `LogReg.php`, which contains the code for user login and user register operations. In the code, variables `$email` and `$pw` are the ones that will receive the common entry points we referred above. Therefore, they are the common variables in both forms but

[3]https://www.facebook.com/
[4]https://ieeexplore.ieee.org/Xplore/home.jsp
[5]https://forumengine.enginethemes.com/
[6]https://checkout.elsevier.com/auth

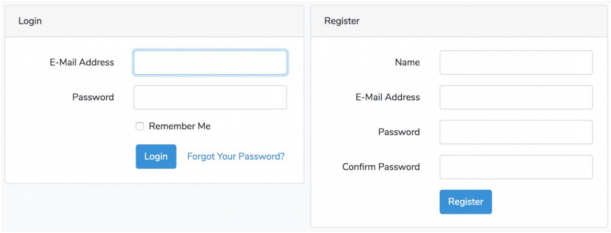

Fig. 1. Login and register user forms in a same webpage.

are processed separately, i.e., in distinct contexts (login or register). In addition, the `$sql` and `$res` variables will be (re)used along the file to compose SQL queries and get their results after their executions in the database. Nevertheless, the code contains three SQLi vulnerabilities, one in the user login and two in the user registration, which are presented for each scenario in Table I (see next section).

```
1  <form name="flogin" method="get" action="LogReg.php">
2  <label for="email">E-mail Address</label>
3  <input id="email" type="email" name="email" value="">
4  <label for="email_pass">Password</label>
5  <input id="email_pass" type="password"
       name="email_pass" value="">
6  <input type="submit" name="login" value="Login">
7  </form>
8
9  <form name="fregister" method="get" action="LogReg.php">
10 <label for="name">Name</label>
11 <input id="name" type="text" name="name" value="">
12 <label for="email">E-mail Address</label>
13 <input id="email" type="email" name="email" value="">
14 <label for="email_pass">Password</label>
15 <input id="email_pass" type="password"
       name="email_pass" value="">
16 <label for="email_pass_conf">Confirm Password</label>
17 <input id="email_pass_conf" type="password"
       name="email_pass_conf" value="">
18 <input type="submit" name="register" value="Register">
19 </form>
```

Listing 1. HTML code to generate the forms of Figure 1.

B. Coding Style Scenarios

Based on the code of the Listing 2 we derived six coding style scenarios for processing the data provided from login and register forms. In order to define these scenarios, first we defined different code blocks – `cb` – that could process the data and follow the programming practices described in the previous section, and then we built the scenarios. Some of these scenarios are the most employed by programmers, and others are not so viewed, but used. The definition of `cb` involves the definition of common `cb` to receive the user inputs common to both forms and to execute a query composed of another `cb`, and `cb` for each specific operation but containing variables that are (re)used throughout them.

Figure 2 illustrates the scenarios with their code blocks, specifying for each the numbers of the lines of code (between braces) that compose them. For all, the HTML code of the forms is equal (grey blocks). Specific `cb` for login operation are green, whereas `cb` for register operation are blue. Common `cb` (orange) are those that are common in processing of both forms. The `cb` on light green is the code of the login

operation split on two cb (P1 and P2). Analyzing the code of Listing 2, the first scenario (SC-1) is active. To activate the other scenarios, it is necessary to activate their lines, by uncommenting some lines and commenting others. Finally, the notation $\{Lx - Ly\} \setminus \{Lz\}$ means that the cb is defined by the $\{Lx - Ly\}$ range of lines of code, except the Lz line.

```
1 <?php
2
3 // common variables to both forms.
4 //$email = $_GET["email"];
5 //$pw = $_GET["email_pass"];
6
7 // code to process the login form
8 if (isset($_GET["login"])) {
9   $email = $_GET["email"];
10  $pw = $_GET["email_pass"];
11  $sql= "SELECT * FROM users WHERE addr ='".$email."'
        AND addr_pass ='".$pw."'";
12  $res = mysqli_query($con, $sql);
13 }
14
15 // code to process the register form
16 if (isset($_GET["register"])) {
17  $name = $_GET["name"];
18  $email = $_GET["email"];
19  $pw = $_GET["email_pass"];
20  $pw_conf = $_GET["email_pass_conf"];
21
22  $sql= "SELECT * FROM users WHERE addr ='".$email."'";
23  $res = mysqli_query($con, $sql);
24
25  if (mysqli_num_rows($res) != 0)
26    echo "This user already exist";
27  else{
28    if ($pw === $pw_conf){
29      $sql = "INSERT INTO users ('name', 'email',
            'password') VALUES ('".$name."',
            '".$email."', '".$pw."')";
30      $res = mysqli_query($con, $sql);
31    }
32    else
33      echo "Passwords do not match";
34    }
35 }
36
37 // code to process the final query of both forms
38 //$res = mysqli_query($con, $sql);
39
40 // code to process the login form
41 //if (isset($_GET["login"])) {
42 // $email = $_GET["email"];
43 // $pw = $_GET["email_pass"];
44 // $sql= "SELECT * FROM users WHERE addr ='".$email."'
       AND addr_pass ='".$pw."'";
45 //}
46
47 // differentiated code to be processed by each form
48 if (isset($_GET["login"])) {
49   //$res = mysqli_query($con, $sql);
50   if (mysqli_num_rows($res) == 1)
51     echo "Login successful"
52 }
53 else
54   if ($res === TRUE)
55     echo "User registered with success"
56 ?>
```

Listing 2. PHP code for processing the forms of Fig. 1.

For example, the composition of the second scenario (SC-2) is read as follows: a common cb which is executed for both operations to receive entry points, a cb that is only executed for login operation, and a cb that is only executed for register operation. To enable this scenario, its lines must be uncommented and the remaing lines commented.

Table I describes each scenario, in the first two columns.

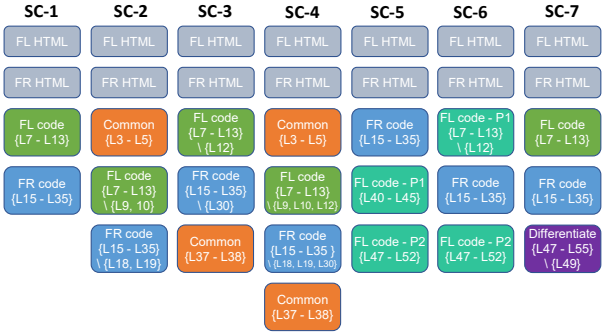

Fig. 2. Considered scenarios to process multiple forms on a single PHP file.

As we stated before, the code contains three vulnerabilities (one in the user login and two in the user registration), and depending on the scenario, they can be from Vul-C1 or Vul-C2 (see Section II-A). The last two columns present the execution path of each vulnerability (i.e., the code lines that handle entry points and their dependencies, and a sink) and the vulnerability case it fits. As a final note, both Fig. 2 and Table I show a seventh scenario which will be discussed in Section IV-C.

IV. Analysis of SAT's Behaviours and Detection

This section presents the analysis of SAT's behaviours and detection when they analyse the use case over the six coding style scenarios. However, before presenting it, first, it gives an overview of the SATs we used. The section ends presenting a discussion about if the SQLi cases detected by SATs as true positives are indeed exploitable, i.e., if those SQLi vulnerabilities represent a real danger.

A. Static Analysis Tools

To assess ours coding style scenarios, we selected three open-source PHP SATs that employ taint analysis to find SQLi vulnerabilities, namely phpSAFE [7], RIPS [9] and WAP [10].

RIPS is the oldest of them and the most cited in the literature, but its open-source version does not analyse object-oriented programming (OOP) code. phpSAFE and WAP are more recent tools which can analyse OOP code, and both also analyse WordPress plugins, besides of web applications. WAP comprises a module to predict false positives (FP-C1 and FP-C2, Section II-A) and another to remove the vulnerabilities it founds, by inserting functions that sanitise the entry points.

An important aspect about SATs, specifically those that employ taint analysis, is that, before performing the analysis of the language for what they were programmed, they remove any other languages contained in files of the target application. For example, for a PHP web application that its files contain HTML and Javascript (besides PHP), PHP SATs will remove these codes first, and then will perform the PHP analysis. This means that SATs are agnostic of the client-side context from which inputs are provided. Therefore, they track *all* application entry points at the same time, regardless of the client-side context. Another feature is that, for performing the analysis of a given instruction and propagating the variable taintedness, they take into account the last assignment of that variable. For

TABLE I
CODING STYLE SCENARIOS OF THE USE CASE AND DEFINED FROM THE LISTING 2.

SC	Description	Vul-C1	Vul-C2
SC-1	The code of each operation (login and register) is clearly delimited, without having common code that serves both operations. All three vulnerabilities are from Vul-C1.	{L9, L10, L11, L12} {L18, L22, L23} {L17, L18, L19, L29, L30}	
SC-2	There is a common `cb` for receiving the common entry points and assign them to `$email` and `$pw` variables. The remaining code is specific and clearly delimited for each operation. All three vulnerabilities are from Vul-C1.	{L4, L5, L11, L12} {L4, L22, L23} {L4, L5, L17, L29, L30}	
SC-3	The code of each operation is well delimited. However, there is a common `cb` for processing the final query of both operations. One vulnerability is from Vul-C1, whereas the other two are from Vul-C2.	{L18, L22, L23}	{L9, L10, L11, L38} {L17, L18, L19, L29, L38}
SC-4	There are two common `cb`, one for receiving the entry points and other for processing the final query for both operations. Between them, there are two `cb` containing the specific code for each operation. One vulnerability is from Vul-C1, whereas the other two are from Vul-C2.	{L4, L22, L23}	{L4, L5, L11, L38} {L4, L5, L17, L29, L38}
SC-5	The code of each operation (login and register) is clearly delimited. However, the `cb` of the login operation is split on two `cb`, but they are executed sequentially. All three vulnerabilities are from Vul-C1.	{L18, L22, L23} {L17, L18, L19, L29, L30} {L42, L43, L44, L49}	
SC-6	The code of each operation is clearly delimited. However, the `cb` of the login operation is split on two `cb`, but they are intercalated by the `cb` of the register operation. All three vulnerabilities are from Vul-C1.	{L9, L10, L11, L49} {L17, L18, L19, L29, L30} {L18, L22, L23}	
SC-7	Scenario equal to SC-1, but has one more `cb` containing different code for each operation (purple `cb` on Figure 2). All three vulnerabilities are from Vul-C1.	{L9, L10, L11, L12} {L18, L22, L23} {L17, L18, L19, L29, L30}	

Vul-C1: Vulnerability of Case 1; Vul-C2: Vulnerability of Case 2 both defined in Section II-A

example, supposing that the variable `x` has two assignments in lines 10 and 12 and it is used in an expression on line 15, the analysis will consider the assignment of line 12.

B. SAT's Behaviours and Detection

We run the three tools over the six scenarios and we analysed their outcomes intending to understand their behaviours and check the veracity of their results. *All SATs had the same results and behaviours*. Table II, columns 4 to 7, presents such results for the vulnerabilities contained in each scenario (columns 1 to 3). As expected, none SAT generated any false positive of FP-C1 and FP-C2 (column FP in the table) once the code does not contain functions related to these cases (e.g., sanitisation). Next, we detail the analysis of the behaviours of the tools by scenario.

a) Scenarios SC-1 and SC-2: For these scenarios, the tools correctly detected all vulnerabilities. The fact of the code of each operation is well delimited on both scenarios justifies these results. Although SC-2 has a common `cb`, containing the shared entry point instructions for both login and register operations, it does not affect the SAT's behaviour since the variables $\$email$ and $\$pw$ that receive such entry points are not changed (reassigned) along the `cb` of each operation. Another aspect that contributed for these results is the fact of all vulnerabilities are from Vul-C1, meaning that each sink only receives a distinct set of entry points, which one belongs to an operation. So the probability of SATs can incur in a wrong analysis is minor. These reasons make that the multiple forms case programmed with these coding styles do not affect the SAT's behaviour, and so SATs can continue agnostic to the client-side context.

b) Scenarios SC-3 and SC-4: All tools correctly detected two vulnerabilities and had a false negative. Both scenarios include common blocks. SC-4 has the same common `cb` as SC-2, which, as we already stated, it does not interfere in the SAT's analysis since the involved variables are not reassigned along with the code. On the other hand, the common `cb` that ends both scenarios affects the analysis performed by tools. This `cb` contains a sink to execute the queries built in the above `cb` and the (re)used `$res` variable that receives the result of the sink execution. Moreover, because of this common sink (L38), two vulnerabilities are from Vul-C2, meaning that two distinct entry points sets reach that sink (one of each operation). Also, the `$sql` variable is used in both operations.

Since only one operation is expected to be executed, the programmer's decision of making these variables and sink equals for both operation is correct, however, SATs do not

TABLE II
RESULTS OF SATS OVER THE SIX CODING STYLE SCENARIOS.

SC	Vulnerability	Case	TP	FN	FP	FFP
SC-1	{L9, L10, L11, L12}	Vul-C1	1			
	{L18, L22, L23}	Vul-C1	1			
	{L17, L18, L19, L29, L30}	Vul-C1	1			
SC-2	{L4, L5, L11, L12}	Vul-C1	1			
	{L4, L22, L23}	Vul-C1	1			
	{L4, L5, L17, L29, L30}	Vul-C1	1			
SC-3	{L18, L22, L23}	Vul-C1	1			
	{L9, L10, L11, L38}	Vul-C2		1		
	{L17, L18, L19, L29, L38}	Vul-C2	1			
SC-4	{L4, L22, L23}	Vul-C1	1			
	{L4, L5, L11, L38}	Vul-C2		1		
	{L4, L5, L17, L29, L38}	Vul-C2	1			
SC-5	{L18, L22, L23}	Vul-C1	1			
	{L17, L18, L19, L29, L30}	Vul-C1	1			
	{L42, L43, L44, L49}	Vul-C1	1			
SC-6	{L9, L10, L11, L49}	Vul-C1		1		
	{L17, L18, L19, L29, L30}	Vul-C1	1			1: L49
	{L18, L22, L23}	Vul-C1	1			

TP: true positive; FN: false negative; FP: false positive from FP-C1 and FP-C2; FFP: *false* false positive (FP-C3)

have this knowledge as they do not analyze the client-side context. These factors are in the root of the false negative. Therefore, SATs only detected the vulnerability whose `$sql` variable is assigned closer to the common sink (i.e., the second vulnerability of the register operation), and generated a false negative for the vulnerability that uses that sink, where the `$sql` variable is assigned farther from it. Lastly, the third vulnerability is from Vul-C1 and is included in the register operation code. It is detected by all tools. From this analysis, we conclude that these coding styles of sharing sinks and reusing variables underlie the production of false negatives.

c) Scenario SC-5: All tools detected the three vulnerabilities. Their outcomes are justified by this scenario does not have common `cb`, the register operation code is well delimited and the login operation code although is split on two blocks (P1 and P2), they are placed sequentially, and, hence, they will work as a single block. Also, vulnerabilities are from Vul-C1 which may not interfere with SAT's analysis, as we have seen so far. Generally, this scenario can be compared to SC-1, and so have the same results than it.

d) Scenario SC-6: This scenario was where the tools had the worst results; they correctly detected two vulnerabilities and produced one false negative and one false positive of FP-C3 (an inexistent execution path). The composition of SC-6 is similar to SC-5; however, the P1 and P2 blocks are intercalated by the register operation block. P2 is the block that contains the sink for executing the query composed on P1.

Despite all vulnerabilities are from Vul-C1, the vulnerability associated with the login operation is not detected (the false negative, FN) and in its place is produced a (false) false positive (FFP). Since `$sql` is reused by the register operation on line 29, it is used twice on sinks of lines 30 and 49. For line 30, the detection is correct, but for line 49 it is incorrect, producing thus a FFP, and consequently a FN. This happens because tools do not know which is the form that belongs line 49 and only consider the last variable assignment and its taintedness for the sink analysis (i.e., the reassignment of `$sql` on line 29). We recall that this approach of SATs considering the last variable assignment and its taintedness is the usual form of they performing the analysis and propagating the taintedness. Moreover, the resulting FN in SC-3 and SC-4 scenarios rely on this form of analysis. Associated with this, the fact of they do not consider the client-side context, and so are not capable of distinguishing the parts of code that belong to each operation, results on the production of false execution paths (FFP) and FN. Lastly, these factors and this coding style make clear the importance and necessity of SATs have to consider the client-side context in their analysis.

From the analysis we made and the experience we have on manual source code analysis, the most coding styles applied by programmers are those illustrated by SC-1 to SC-4 scenarios, followed by SC-6. This means that the common blocks and reuse of variables are a usual practice. These manners of programming are correct, but they induce SATs on performing a wrong analysis. Nevertheless, they underlie the generation of (false) false positives and, worst of that, false negatives. Another aspect that contributes for this is the fact of SATs are totally agnostic to the client-side context, and so they are not capable of separating and distinguishing which blocks of code are associated with the different parts (e.g., HTML forms) that composing the web application surface.

Finally, we conclude that SATs to be more precise and accurate, they need to be improved towards the integration of client-side context inspection before processing the server-side code. That way, SATs will have information about the sets of entry points that work together, and then should perform their analysis taking into account these sets separately. For instance, for our use case and considering the HTML code of Listing 1, we would have two sets (one for each form): `Login: <$_GET["email"], $_GET["email_pass"], $_GET["login"]>` and `Register: <$_GET["name"], $_GET["email"], $_GET["email_pass"], $_GET["register"], $_GET["email_pass_conf"]>`. The analysis should be guided through these two sets. In other words, it would be performed twice, one for each form. In the first time, the analysis will only consider the `Login` set, and so only these entry points will be tainted and considered. The code regarding other entry points will be ignored. Next, the analysis is applied for the `Register` set. For example, for SC-3, when the analysis is made using the `Login` set, the green and orange `cb` would be analyzed since `$_GET["login"]` belongs to the set in analysis. On contrast, the blue `cb` would be ignored because `$_GET["register"]` does not belong to that set. However, there are new challenges SATs will face. As a first challenge, how to identify the server-side code that processes a specific set of entry points? For instance, in the SC-6 scenario, how will tools identify P1 and P2 blocks for login set of entry points. Secondly, how to ensure that a given variable assignment will be used on a given function (e.g., a sink), instead of its last assignment (as usually happens)?

C. SQLi Exploitation

The main goal of a SQLi attack is to retrieve data from the database and/or insert data there for later the attacker using it in its benefit. As we said before, SATs are built having in mind how to detect vulnerabilities by following the vulnerability definitions stated in Section II-A. For that, their targets are entry points and sinks, and by employing a top-down taint analysis (most of them), they check if the former attain the latter, without suffering any sanitisation or validation process.

Based on the vulnerability definition SATs follow, a characteristic they have in their inspections is that the analysis of a given data flow (execution path) is stopped when they found a vulnerability, i.e., when a sink is attained. A vulnerability is exploitable if the result of the use of malcrafted (malicious) inputs on sensitive sinks is immediate and visible, i.e., the application behaves differently from expectable. For example, considering the PHP instruction `echo $_GET["name"];` that has the goal of outputting to the screen the `$_GET["name"]` content (the user name), and which is a typical cross-site scripting (XSS) vulnerability. If

we assign a malicious JavaScript to `$_GET["name"]` and use it on the `echo` sink, the result will be the script execution instead of outputting the user name as expected. The result of this action is immediate and visible; therefore, we can say that this vulnerability is indeed exploitable.

While all vulnerability classes are exploitable based on this principle and the vulnerability definition, for SQLi, both principles may not be always valid. This means that a web application can contain SQLi vulnerabilities, by definition, but they are not exploitable, by the exploitation principle. Furthermore, SATs, when detect SQLi flaws as true positives, this does not mean that they are real exploitable SQLi.

The following analysis explores this assumption for register and login operations from Listing 2, and based on the SC-7 scenario from Figure 2 (described on last row of Table I). SC-7 composition is equal to SC-1 plus a `cb` with distinct code for each operation. This `cb` ends the login or register operation based on the result of the last query made by it (lines 12 and 30, respectively). For the purpose, for login a user, he needs to provide his credentials (email and password), which are inserted in a query (lines 9 – 11) to be executed in the database (line 12). If the user exists in the database a successful login message is outputted (lines 48 – 51). On the other hand, for register a user in the database, he has to provide his name, email, password and its confirmation (lines 17 –20). Before registering a user in the database, it is verified if a user with that email already exists there (lines 25 – 26). If not the case, a second verification is made to verify if both passwords match, and, if such the case, then the user is inserted in the database (lines 28 – 34). In the end, the differentiate `cb` verifies if the user was correctly inserted in the database, outputting a message accordingly (lines 54 – 55).

The code contains three SQLi vulnerabilities identified by SATs, namely {L9, L10, L11, L12} for logging a user, {L18, L22, L23} for checking the email and {L17, L18, L19, L29, L30} for inserting the user in the database.

a) Check a user on the database with a malicious input: Supposing, we want to retrieve data from the database unduly. To do so, we insert as email the code `' OR True --` (line 18), which when put in SQL will result the final query `"SELECT * FROM users WHERE addr ='' OR True"` (line 22), a query where the condition is always true. The query is executed in the database and returns all users from `users` table (line 23). Variable $\$res$ receives the database records as an object, but it is not returned to the malicious user. At this point, the vulnerability has not yet been exploited because the records have not been returned to the user. Therefore, its real exploitation depends on what $\$res$ variable is used by the application and the way it is iterated.

The real user verification is made by line 25, which verifies if the number of resulting records is not zero. If it is the case, this means that a user with that email already exists in the database, and the registration process ends. Given that input, the number of records on `$res` is not zero, this verification is true, and the message of line 26 is displayed. As result, an unexpected application behaviour does not occur, and nothing happens. Therefore, we can say that this SQLi vulnerability can not be exploitable by malicious code that intends to retrieve more than one record from the database.

On the other hand, when we consider as email the code `' AND False --` for inserting a user in the database, the resulting query `"SELECT * FROM users WHERE addr =' AND False"` when executed returns zero records. In this case, the email check is passed and the code is put in the insert query (line 29), but its execution will not be allowed because the resulting query is not syntactically correct.

Therefore, despite SATs identified such vulnerabilities as true positives, actually they are not exploitable. We recall that a SQLi is exploited if the exploitation result is immediate and visible by an unforeseen application behaviour. Also, its exploitation depends on how the result object (`$res`) is iterated. Hence, if this iteration is for counting the number of records, the exploitation is not possible. Otherwise, if the iteration is to access the data contained on records (e.g., using the `mysqli_fetch_assoc` function) and outputs it to the user, in this case, the exploitation is effective. We recall that a way to return the result of an attack to the attacker is by outputting it to a file or screen, by iterating over it. However, there is need that the application contains the code that performs this.

b) Register a user with a sanitised email: As we stated before, a manner to avoid vulnerabilities in applications is sanitising the entry points. For SQLi and accordingly with the `mysqli_query` sink (used in Listing 2), the `mysqli_real_escape_string` sanitization function is applied. The function escapes some characters that alter the structure of the query, such as the prime and double-quote.

Considering one of the above malicious codes as the user email (e.g., the first code), when it is sanitised, the result is `\' OR True --` . Also, suppose we insert the sanitization instruction `$email = mysqli_real_escape_string($con, $email)` on line 21, before using `$email` in query on line 22. The resulting query is `"SELECT * FROM users WHERE addr ='\' OR True -- '"`. Afterwards, the query will be executed in the database (line 23), which will return zero records to `$res` since there are no users with that email. That means that the check of line 25 is false, the password validation proceeds (line 28), and if they match the query of line 29 will receive that sanitised email, and it will be inserted on the database (line 30). The process ends at line 55, outputting a successful user registration message.

As we can observe, there is no vulnerability on {L18, L21, L22, L23}. But, unexpectedly a sanitised malicious input was registered as a valid user in the database, something that could not happen, i.e., an unexpected and visible application behaviour occurred, like a vulnerability exploitation. Moreover, this sanitised code will be stored in the database as malicious. MySQL before inserting it in the database removes the sanitisation it has [20][21], and, therefore, it can be used later by an attacker to perform a 2nd-order SQLi. Furthermore, when SATs analyse this code (and considering all entry points

as being sanitised), they do not report any vulnerability.

c) Login a user: For a first analysis and considering the first malicious code as the user email for the login operation. The resulting query is the one presented in a), and so when executed extracts all registers from the `users` table, which are stored in `$res` variable. The check of line 50 verifies if the number of records on `$res` is equal to one, meaning that the user was correctly logged in. For our case, this check is false, and nothing happens. Notice that the same result is got when we use the second malicious code. Once again, as observed in a) the vulnerability {L9, L10, L11, L12} may not be exploitable, for the reasons already indicated in a).

The only chance we have to exploit this case is to provide a valid email, as `mail@mail.com'--` , considering that `mail@mail.com` exists in the database. However, if we employ sanitisation to this code (on line 9), this exploitation is avoided. That allows us to say that, for this case, the sanitisation is effective and works.

Therefore, for login, SATs correctly identify the SQLi as being really exploitable. Moreover, if the application resorts from sanitisation, they correctly will not identify this case.

After analysing these SQLi exploitation cases, we can summarize the following: (1) the execution of the query *per se* does not ensure the immediate and visible exploitation, but what action is made over its results; (2) the analysis of SATs behaviour is not complete for SQLi detection, meaning that it is necessary to analyse what is done with the query results in order to understand if there is an effective exploitation or not. For that, the variable that receives the query result needs to be tracked and verify what iterations are made over it; (3) the applicability of sanitisation functions on the vulnerabilities identified by SATs needs to be studied in order to understand if they are needed or not.

V. Conclusion

The paper presents an analysis of the SAT's behaviours and detection when they process applications developed by programmers with different coding styles and programming practices, such as the reuse and share of variables. Due to these practices, SATs often generate false positives and negatives. Also, it presents an analysis and discussion about the exploitation of SQL injection (SQLi) vulnerabilities detected by SATs as being true positives. Our analysis demonstrated that SATs are built having in mind how to detect specific vulnerabilities, without considering such forms of programming and the effective exploitation of SQLi. These results call to action for a new generation of SAT to be capable of understanding the code they process.

Acknowledgments. This work was supported by FCT through project SEAL (PTDC/CCI-INF/29058/2017), and the LASIGE Research Unit (UIDB/50021/2020).

References

[1] CVE, "CVE Details. The ultimate security datasource," https://www.cvedetails.com/browse-by-date.php.

[2] DarkReading, "Sql injection attacks represent two-third of all web app attacks," 2019, https://www.darkreading.com/attacks-breaches/sql-injection-attacks-represent-two-third-of-all-web-app-attacks/d/d-id/1334960.

[3] WhiteHat Security, "Application Security Statistics Report. The case for DevSecOps," Nov. 2017.

[4] I. Medeiros, N. F. Neves, and M. Correia, "Detecting and removing web application vulnerabilities with static analysis and data mining," *IEEE Transactions on Reliability*, vol. 65, no. 1, pp. 54–69, March 2016.

[5] WhiteHat Security, "The DevSecOps Approach - Using AppSec Statistics to Drive Better Outcomes," Nov. 2019.

[6] N. Jovanovic, C. Kruegel, and E. Kirda, "Precise alias analysis for static detection of web application vulnerabilities," in *Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security*, Jun. 2006, pp. 27–36.

[7] P. Nunes, J. Fonseca, and M. Vieira, "phpSAFE: A security analysis tool for OOP web application plugins," in *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, Jun. 2015.

[8] J. Williams and D. Wichers, "OWASP Top 10 2017 – The Ten Most Critical Web Application Security Risks," 2017.

[9] J. Dahse and T. Holz, "Simulation of built-in PHP features for precise static code analysis," in *Proceedings of the 21st Network and Distributed System Security Symposium*, Feb 2014.

[10] I. Medeiros, N. F. Neves, and M. Correia, "Automatic detection and correction of web application vulnerabilities using data mining to predict false positives," in *Proceedings of the International World Wide Web Conference*, Apr. 2014, pp. 63–74.

[11] ——, "Equipping WAP with weapons to detect vulnerabilities," in *Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2016.

[12] P. Nunes, I. Medeiros, J. Fonseca, N. Neves, M. Correia, and M. Vieira, "On combining diverse static analysis tools for web security: An empirical study," in *Proceedings of the 13th European Dependable Computing Conference*, 2017, pp. 121–128.

[13] ——, "Benchmarking static analysis tools for web security," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1159–1175, Sept 2018.

[14] A. Algaith, P. J. C. Nunes, J. Fonseca, I. Gashi, and M. Vieira, "Finding SQL injection and cross site scripting vulnerabilities with diverse static analysis tools," in *14th European Dependable Computing Conference (EDCC)*, 2018, pp. 57–64.

[15] L. Flynn, W. Snavely, D. Svoboda, N. VanHoudnos, R. Qin, J. Burns, D. Zubrow, R. Stoddard, and G. Marce-Santurio, "Prioritizing alerts from multiple static analysis tools, using classification models," in *Proceedings of the 1st International Workshop on Software Qualities and Their Dependencies*, May 2018, pp. 13–20.

[16] J. D. Pereira, J. R. Campos, and M. Vieira, "An exploratory study on machine learning to combine security vulnerability alerts from static analysis tools," in *2019 9th Latin-American Symposium on Dependable Computing (LADC)*, Nov. 2019, pp. 1–10.

[17] N. Medeiros, N. Ivaki, P. Costa, and M. Vieira, "Software metrics as indicators of security vulnerabilities," in *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, Oct. 2017, pp. 216–227.

[18] ——, "An approach for trustworthiness benchmarking using software metrics," in *2018 IEEE 23rd Pacific Rim International Symposium on Dependable Computing (PRDC)*, Dec. 2018, pp. 84–93.

[19] I. Medeiros, N. F. Neves, and M. Correia, "DEKANT: a static analysis tool that learns to detect web application vulnerabilities," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, Jul. 2016.

[20] I. Medeiros, M. Beatriz, N. Neves, and M. Correia, "SEPTIC: Detecting Injection Attacks and Vulnerabilities Inside the DBMS," *IEEE Transactions on Reliability*, vol. 68, no. 3, pp. 1168–1188, Sept 2019.

[21] ——, "Demonstrating a Tool for Injection Attack Prevention in MySQL," in *Proceedings of the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, Jun. 2017.