An Analysis of Black-Box Web Application Security Scanners against Stored SQL Injection

Nidal Khoury, Pavol Zavarsky, Dale Lindskog, Ron Ruhl
Concordia University College of Alberta,
Edmonton, Canada
nidkhoury@gmail.com, {pavol.zavarsky, dale.lindskog, ron.ruhl}@concordia.ab.ca

Abstract—Web application security scanners are a compilation of various automated tools put together and used to detect security vulnerabilities in web applications. Recent research has shown that detecting stored SQL injection, one of the most critical web application vulnerabilities, is a major challenge for black-box scanners. In this paper, we evaluate three state of art black-box scanners that support detecting stored SQL injection vulnerabilities. We developed our custom testbed that challenges the scanners capability regarding stored SQL injections. The results show that existing vulnerabilities are not detected even when these automated scanners are taught to exploit the vulnerability. The weaknesses of black-box scanners identified reside in many areas: crawling, input values and attack code selection, user login, analysis of server replies, mis-categorization of findings, and the automated process functionality. Because of the poor detection rate, we discuss the different phases of black-box scanners’ scanning cycle and propose a set of recommendations that could enhance the detection rate of stored SQL injection vulnerabilities.

Keywords—stored SQL injection; black-box scanners; vulnerabilities

I. INTRODUCTION

SQL Injection is a form of attack, normally against a database-driven web application, in which the attacker executes unauthorized SQL commands by taking advantage of non-validated input vulnerabilities. In a successful attack, the attacker will pass an SQL attack code to the back end database for execution. He may also execute system calls through the database server [1].

The Open Web Application Security Project (OWASP) 2010 [2] report for the top ten web application vulnerabilities shows that SQL injection ranked first among other vulnerabilities. Many categories of SQL injection exist such as reflected, blind and stored. Reflected and blind SQL injections occur once the SQL commands are accepted by the web application and processed by the backend database. On the other hand, stored SQL injection, also known as persistent or second order SQL injection, occurs when malicious code is injected into an application, is not immediately executed, but is instead stored by the application (e.g., temporarily cached, logged, stored in a database) and then later retrieved, rendered and executed by the victim [3]. This research addresses stored SQL injection cases where the storage medium is a database.

A stored SQL injection attack is a form of chained SQL injection attack as it requires two steps to complete the attack:

- The first step is storing a record holding the SQL command in the database. At this stage, the stored SQL code is not yet executed.
- The second step is fetching the stored malicious code and executing it. This takes place when an SQL query uses one or more fields from the database as parameters. Because these fields are carrying crafted attack code, they are appended to the original query instead of serving as a parameter. The malicious code is executed, resulting in a successful stored SQL injection attack.

In order to minimize the likelihood of successful attacks against web applications, security professionals use a variety of techniques to identify existing web application vulnerabilities. Black-box web vulnerability scanning is a technique that became widely adopted due to the ease of use, automation, and independence from the web application technology used. It tends to exercise the security vulnerabilities of a web application and generate reports and statistics about the findings. However, many researchers [4, 5, 6] have shown the limitations of black-box scanners in detecting those vulnerabilities by testing multiple black-box scanners against a variety of vulnerable applications. The reports have invariably shown that black-box security scanners fail to detect all categories of SQL Injection attacks, and in two of those reports, the detection rate was in fact 0% [4, 5]. The aforementioned reports confirmed the poor detection rate of stored SQL injection vulnerabilities.

Previous research identified the challenges of detecting stored SQL injections at a higher level, but failed to highlight the technical aspect of these challenges. Our paper aims to address this gap by discussing the root cause of the difficulty to detect stored SQL injections. We do so based on a close observation of scanners behaviors’ and captured network traffic between scanners and web applications.

Further, previous studies [4,5] used multiple black-box scanners, among which some scanners did not support the detection of stored SQL injection vulnerabilities. To address this shortcoming, our experiments are rather based on scanners that fully support the detection for stored SQL injection.
To obtain sound results, our testbed “MatchIt” presented almost no complexities or challenges for black-box scanners to exploit the existing vulnerability, whereas testbeds used in previous research [4,5] presented many challenges for black-box scanners. The latter testbeds also contained other types of vulnerabilities.

In this paper we focus exclusively on stored SQL injection vulnerabilities. We:

- Show a scenario of a stored SQL injection attack
- Discuss the different stages black-box scanning.
- Explain our test methodology with different scanning profiles used and test three black-box scanners against three custom testbeds.
- Discuss our test results and highlight the major challenges and limitations identified.
- Based on our analysis, give recommendations on how to improve the poor detection rate for stored SQL injection.

The main contributions of our paper are as follows:

- We present the technical details of our experiments that help further researchers put forward better recommendations for the problems identified.
- We propose a set of best practices for security professionals to obtain better results when using black-box scanners.
- We designed a testbed that we plan to publicly release to be used by other researchers.

II. STORED SQL INJECTION

A typical stored SQL injection vulnerability exists in a form or a webpage where an unsanitized input field can be used to store malicious code in the database. An attacker trying to exploit this vulnerability will need an understanding of the application’s functionalities, behavior, and the logical flow of data inside the application. To execute the stored attack code, the attacker will rely on his understanding of where and how the unsanitized data will later be used by the application.

To illustrate the different stages of a stored SQL injection, we present a scenario by exploiting an existing vulnerability in the WackoPicko [7] testbed application. This application allows a user to register to the website under /users/register.php. The registration form has an unsanitized input field for the user’s first name. In the first phase of the attack we store the attack code ‘ or 1 = 1 in the database. WackoPicko has a user’s directory /users/similar.php where a logged in user can search for other members with similar usernames. This directory automatically triggers a query on the logged-in user’s behalf, using his first name as a query parameter. The code below is taken from WackoPicko (refer to Section IV.A. for the technology being used):

```php
$pattern = '/%{$login}%' . $query = 'SELECT * from `users` where `firstname` like "%{$login}%" and firstname != "{$login}"';
```

When the query above is executed, the $login variable is replaced by the user’s first name. A user with a first name John would trigger the following query:

```sql
SELECT * from `users` where `firstname` like %John% and firstname != 'John';
```

If we applied our attack code ‘ or 1 = 1 to the query above, it becomes:

```sql
SELECT * from `users` where `firstname` like %’ or 1=1%’ and firstname != ’ or 1 = 1’;
```

The resulting query would simply return all rows in the users’ table since it has the statement or 1=1 which is always true. The attack code above changed the original query result, and it can be designed to change the purpose of the query such as using ‘ ; update users set password = ’ ; as an attack code which sets all users’ passwords to the empty string. If the database user account used by the web application has sufficient privileges (e.g., root), more harmful attacks can be performed. For example, using ‘ ; drop table creditcards; as an attack code will delete the creditcards table. The attack code may also make system calls, such as sending the sleep signal to the SQL server instance by using ‘ ; select sleep(2000) as GoToSleep;.

III. BLACK-BOX SCANNERS

Although black-box web application vulnerability scanners have strengths, previous studies have shown their limitation in detecting stored SQL injection; however, some of the scanners used by previous researchers [4,5] did not support the detection of stored SQL injection. For this purpose, we conducted a market analysis and sent inquiries to different black-box scanner vendors (McAfee, Cenzic Hailstorm, Acunetix, QualysGuard, WebInspect, Rational AppScan, N-Stalker and NeXpose). Many vendors confirmed their scanner’s capability of detecting stored SQL injection vulnerabilities, whereas others denied it, citing the difficulty of automating this type of attack. For our experiments, we selected three black-box scanners (see Table I) that support detecting stored SQL injection.

<table>
<thead>
<tr>
<th>Scanner</th>
<th>Vendor</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acunetix WVS</td>
<td>Acunetix</td>
<td>7.0 Build 20100921</td>
</tr>
<tr>
<td>Rational AppScan Enterprise</td>
<td>IBM</td>
<td>8.00.0.0 (Build 444)</td>
</tr>
<tr>
<td>QualysGuard Express Suite</td>
<td>QualysGuard</td>
<td>6.16.60-1</td>
</tr>
</tbody>
</table>

A black-box scanner crawls through a web application’s pages, and searches the application for vulnerabilities by simulating attacks against it [8]. We identified four different stages in a black-box scanning cycle:

1) Crawling: The scanner attempts to browse all the possible links and directories in a web application in order to obtain the HTML source code. A major challenge in this first phase is crawling pages that are protected, such as pages that
require passwords or human input such as captcha. Another challenge is that scanners need to know when and where to perform another round of crawling after submitting data to a web application. This is done when a scanner knows the state of an application, its inputs and its outputs. By the end of this phase, the scanner should have all server replies in HTML format.

2) Forms and Entry points identification: This step can be described as a reverse engineering task since the scanner parses the HTML source code in order to identify any possible forms, methods, and entry points. A typical username field is an input of type “Text” and has the description of “login” or “username”. A typical submission form exists if it contains a GET or POST action with a submit function.

3) Attack code construction and submission: Scanners produce data that matches the required input data type before submitting it to the web server. This data is either generated randomly or fetched from a dictionary. A generally useful attack, called fuzzing, submits random inputs of various sizes to the application [8]. Scanners attempt to use malicious SQL patterns as inputs in order to detect the vulnerabilities in further stages. Some scanners can be configured by the user to utilize a predefined data set while filling web forms. For example, they can be configured to use a predefined email address whenever an email address entry point is identified. Once the form input fields are filled, the scanner uses the submission form and action identified from the previous phase to send the payload to the server. It then waits for a reply.

4) Analysis of replies: The server reply is generated according to many factors, and is influenced by the submitted data. The scanner in this case has to know which data is considered valid, is accepted by the server, and would generate a useful and acceptable reply. This stage too requires the scanner to reverse engineer the server response in order to make further analysis. This is considered a major challenge for a scanner since it has to make a decision as to whether this reply is valid or not, knowing that those replies are primarily designed for human consumption [9]. For this purpose, scanners are equipped with a list of error messages that can be matched. If a reply contains an error that has a match, the scanner then decides to which category this error belongs. For example, if an error was an SQL syntax error, the scanner should conclude that an SQL injection vulnerability exists.

Later in this paper, we highlight the major weaknesses identified in each stage, and propose enhancements based on these observations.

IV. TEST METHODOLOGY

We obtained access to three black-box scanners and tested each of them against three web applications for stored SQL injection. As noted previously, Table I lists the scanners that we used for our experiments.

A. Custom Testbeds

The testbeds used have different levels of complexities and they were designed to challenge diverse functionalities and features in black-box scanners. Table II shows the number of stored SQL injection vulnerabilities in each testbed. It also shows the number of vulnerabilities that require a user login to be successfully exploited.

<table>
<thead>
<tr>
<th>Testbed</th>
<th>Stored SQLI</th>
<th>Login required</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCI</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>WackoPicko</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MatchIt</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

The first testbed, called PCI, was built by Bau et al. [4] in order to evaluate eight different black-box scanners. Among other types of vulnerabilities, PCI has three stored SQL injection vulnerabilities. We obtained permissions from the authors to use this testbed in our experiments.

The second testbed, called WackoPicko [7], was publicly released by Poue et al. [5]. WackoPicko was built to assess eleven black-box scanners, and it contains a stored SQL injection vulnerability in the user registration form. We obtained a virtual image of WackoPicko for our experiments. To exploit the existing vulnerability in WackoPicko [4], the scanner must progress through a complex series of steps. The scanner must (1) create a user account, (2) use a valid attack code in the firstname field while registering a user, and then later (3) log in using this user account and (4) perform a post scan in order to exploit and, finally, (5) detect the vulnerability.

Finally, we built our own custom testbed and we named it “MatchIt”. This web application allows a guest with no privileges to submit his information (name, email and website) to a directory, and later browse the directory to check for names, emails or websites that could match with other records. Based on a recommendation by Fong et al. [8], this application is designed to test a scanner’s ability to detect and exploit a single stored SQL injection vulnerability. Unlike the other two testbeds, this testbed was designed to eliminate any factors or challenges (e.g., user registration, required user login, complicated crawling or data submission) that could prevent the scanner from completing and detecting a successful chained exploit.

The first two testbeds ran on LAMP (Linux, Apache, MySQL, Php) and we used the same environment as previous research [4,5]. “MatchIt” was deployed under a virtual WAMP workstation running Window XP Professional, Apache 2.2.6, MySQL 5.0.45 and Php 5.2.5.

B. Black-box scanning profiles

A scanning profile allows us to specify which type of vulnerabilities to target during a scan. We used four profiles. The first profile (Blind/Reflected) targets reflected and blind SQL injection vulnerabilities. The second profile (Stored) targets stored SQL injection vulnerabilities. The third profile (Full SQLI) targets all types of SQL injection vulnerabilities: reflected, blind and stored SQL injections. Finally we used a fourth profile (full scanning profile) targeting all types of web application vulnerabilities. This is to ensure that we fully tested the scanners’ detection capabilities.
We configured all scanners with regular user logins as well as admin logins in the case where an admin interface existed. For each of the first three scanning profiles above, we performed three tests: a first test with a regular user login, a second test with an admin login, and a third test with no login configuration. The third test was performed to assess scanners’ capabilities to register a new user and use it to log in.

To obtain reliable results, for each experiment we followed the procedure depicted in Table III, going through each of the four phases in sequence (preparation, execution, data mining and analysis). We captured all the network traffic targeting the web server using Wireshark [10]. We analyzed the traffic to identify attack codes and forms being submitted to the web server. We also analyzed server replies and whether their interpretations by black-box scanners were correct and efficient. Because our objective is to analyze the behavior of black-box scanners in general, we kept the scanners’ names anonymous in the following sections, and we did not focus on comparing different scanners’ performances and results.

### TABLE III. TESTING PROCEDURE

<table>
<thead>
<tr>
<th>Phase</th>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preparation</td>
<td>1</td>
<td>Set Web Application to initial state / replace files</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Reload database init_script()</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Restart database &amp; web server</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Restart &amp; Configure black-box scanner with scanning profile and login credentials</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>Restart and Configure Wireshark to capture traffic between web server and scanner</td>
</tr>
<tr>
<td>Execution</td>
<td>6</td>
<td>Start Capturing using Wireshark</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>Start the vulnerability scanning process</td>
</tr>
<tr>
<td>Data Mining</td>
<td>8</td>
<td>Stop Wireshark and save network traffic in pcap formats</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>Save scanner’s results/report</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>Dump the database to a backup directory</td>
</tr>
<tr>
<td>Analysis</td>
<td>11</td>
<td>Check scanner’s report and detection results</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>Analyze database records</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>Analyze Wireshark captured packets</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>Go back to step 1 to start a different experiment</td>
</tr>
</tbody>
</table>

### V. TESTING RESULTS AND DISCUSSIONS

For each test we performed, we recorded the network traffic (associated with crawling) for pages used to store the attack code, represented by Traffic (store) in our figures and tables, as well as network traffic for pages used to execute the stored attack code, represented by Traffic (execute). We also extracted the number of relevant and distinct attack codes successfully stored in the database, represented by Attack code. This latter number was only extracted from fields involved in stored SQL injection vulnerabilities. In this section we report only scanner 1’s performance results. We discuss later the generalizability of the results and compare those obtained from all three scanners.

Running tests using the full SQL injection profile (see Section IV.B) gave almost the same results as using the blind/ reflected profile (Figure 1). The results apply to both the traffic recorded to the web pages, and to the number of valid attack codes injected (as presented in Figure 1). The latter results also explain the poor outcome when running tests using the stored SQLI profile, which did not have a great impact by itself. Table IV shows the results of scanner 1 under different profiles for all three web applications. The traffic recorded in Table IV was collected by running each test using three different user privileges (when available) for each profile (Full SQLI, Blind/Reflected, Stored). Figure 1 shows a huge gap between results for the full SQLI and Reflected/Blind profile on the one hand, and results for the stored SQLI profile on the other hand. This gap exists for all criteria: traffic and attack code stored. For better results, the stored SQLI feature should be used when running a scan against all types of SQL injection vulnerabilities, since its main job as a post scan depends on the values and attack code injected by other methods (blind / reflected).

The traffic recorded for pages used to execute the attack code was very low and in some cases it was 0 (see Table IV-MatchIt). The scanner was able to successfully inject attack code in the database, but it failed to make a second pass to check for new pages that would execute the attack code.

Furthermore, some scanners were not successful at storing attack code to the database (Table IV-WackoPicko). As mentioned earlier, the registration form in WackoPicko is used to store the attack code. Further packet and traffic analysis captured by Wireshark [10] showed that the scanner was sending two different values in the Password and Password Again fields. This weakness in generating correct input values resides in the third phase of a scanners’ cycle, i.e., “Attack code construction and submission” (see Section III). The scanner usually predicts input values based on a field or a variable name found in the source code. The registration form source code showed password / againpass, where againpass is the variable corresponding to a repeat password field. The name of this variable does not, for a scanner, make its function obvious. Looking for keywords before and after entry points could help the scanner generate more accurate inputs. For
example, the againpass input field had a label of Password again. Analyzing the tags before and after an entry point would help scanners overcome this type of challenge. Moreover, the server response was showing a passwords do not match error, but the scanner couldn’t analyze this response and take corrective actions: it kept sending the same wrong values. Previous research has used this challenge to explain the limitation of detecting stored SQL injection in WackoPicko [5]. To test this explanation, we developed our own testbed – MatchIt – that requires no user registration or login, thus controlling any limitation caused by this challenge, or even a user authentication challenge. We also performed an additional series of experiments against WackoPicko. We used three workarounds to help the scanner overcome the registration challenge that affected the overall detection result.

### TABLE IV. SCANNER 1 RESULTS FOR EACH TESTBED

<table>
<thead>
<tr>
<th>Testbed</th>
<th>Scanning Profile</th>
<th>Traffic (store)</th>
<th>Traffic (execute)</th>
<th>Attack code</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCI</td>
<td>Full SQLI</td>
<td>1409</td>
<td>778</td>
<td>92</td>
</tr>
<tr>
<td></td>
<td>Blind / Reflected</td>
<td>1346</td>
<td>775</td>
<td>90</td>
</tr>
<tr>
<td>Wacko</td>
<td>Stored</td>
<td>14</td>
<td>13</td>
<td>5</td>
</tr>
<tr>
<td>Picko</td>
<td>Full SQLI</td>
<td>600</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Blind / Reflected</td>
<td>600</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Stored</td>
<td>6</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>MatchIt</td>
<td>Full SQLI</td>
<td>176</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Blind / Reflected</td>
<td>176</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Stored</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The workarounds, as described below, were performed using the full SQLI profile:

1) We configured the scanner with correct password and againpass values, and configured it to use a set of predefined input values containing an attack code in the vulnerable field. The attack code we used in all our workarounds was ‘ OR I = I: drop table users:. The server response would show an SQL syntax error when this code is executed.

The scanner was able to successfully create a user account that contains our attack code; however, it was not capable of completing the second phase of the attack, i.e., executing the code. This second phase only required the scanner to crawl one page after registering the user. We noticed that the scanner never uses the user accounts it creates for logins and crawling activities, and that the only purpose for creating those accounts is for detecting any vulnerabilities on the registration form, just like any other form that accepts user inputs on the website.

2) We faked the scanner and configured it to register a user instead of logging a user in, forcing it to create an account containing attack code. Then we forced it to visit the page that would execute the code. The scanner was able to complete both attack phases successfully, but still wasn’t able to detect the vulnerability. Analyzing the traffic showed that the scanner received an SQL syntax error exception. However, the report did not show any detection rate for any type of SQL injection vulnerabilities based on this server’s reply. This limitation is due to the server response analysis phase of a scanner’s cycle. The scanner was successful in completing the attack, but was not able to see it. The scanner however could have ignored any errors received while executing the steps that were preconfigured by an end user.

3) We created a user account carrying a valid attack code in the vulnerable first name field. Then we configured the scanner to use this user account for login during the scanning process. Again, the scanner was able to exploit the vulnerability and receive an SQL syntax error response but the server’s response wasn’t handled correctly.

Finally, for each of the workarounds above, we performed a scan using the full vulnerability scanning profile (see Section IV.B). The scanner was able to detect a discrepancy and showed an alert for the page that is used to execute the attack code. The medium-severity alert, as described by the scanner, does not match OWASP’s critical ranking [2] for SQL injection vulnerabilities. The SQL syntax error matched one of the scanner’s patterns during the server response analysis phase. Although the message shows a pure SQL syntax error, the scanner did not categorize it as an SQL injection incident, but as a generic “error on page”. Besides, the scanner could not determine whether this is a stored SQL injection, or where the attack code was input. Scanners usually categorize their findings and show them in a well documented report.

![Figure 2. Comparing activities of three different scanners](image)

Figure 2 shows the activities as well as the results collected for the three scanners. Although the results are not equal, they are to some point proportional. Typically, the scanner that generated more traffic was able to store more attack code in the database fields.

Figure 3 compares the activities performed under three different profiles while using different privileges. Scanners were able to generate more results when they were preconfigured to log in to the web application, as some links required authentication. Although some scanners were able to create user accounts, they never used those accounts to login and crawl the protected links while logged in. This significantly impacts the effectiveness of the final results. For that purpose, the automation of user creation followed by a user login should be improved. The only extra steps a scanner needs to take, besides registering a user, is remembering the
Scanners succeeded at storing attack codes in the database, but most of those attack codes would not successfully work with stored SQL injection vulnerabilities. They were meant to exploit a reflected or blind SQL injection, where an immediate response was expected. This type of vulnerability simply requires a new set of attack codes. Unlike reflected SQL injection: (1) receiving an error while trying to store an attack code could mean that the first step has failed and the code was not stored. (2) Detecting a blind SQL injection could mean that the first step of a stored SQL injection has succeeded. The scanner should know if the attack code was stored successfully and keep its state. It should remember which form was used to store the code, and what other data was submitted along with the code. Later it should start another crawling process to find related data and try executing its own code. Having a state full scanner would be more efficient for the scanner to relate the application’s inputs and outputs, and thus understand the state-based transactions that take place. This is crucial because when a vulnerability is detected, a scanner should know where the attack code was injected and where it was executed, and show it on the generated report. In normal cases, the process should be fully automated since users do not know that a vulnerability exists, and thus cannot configure or help a scanner to detect it as we did. The full automation is still questionable for this type of vulnerability.

In addition to the challenge above, sometimes a stored SQL injection exploitation could be successful and output data that has a logical error, rather than a syntax error. In this case, it is hard for a scanner to detect any discrepancy in the server’s response, and only a human user would be able to detect this inconsistency.

VI. RECOMMENDATIONS

Based on our experiments and observations, we recommend researchers and practitioners to always perform a full scan (for all type of vulnerabilities) despite their initial interest in certain types of vulnerabilities. We also believe that, when reviewing black-box generated reports, auditors should verify the details of every point on a report. They should be able to relate events, such as connecting medium-severity error on a page alert to a stored SQL injection vulnerability with a much higher severity because of the SQL error reported.

We also wish to emphasize that configuring a scanner with login credentials improves the overall results, since it will be more likely to access pages that require authentication. Indeed, even though some scanners were able to create user accounts, they didn’t use those user accounts to login during the scanning process.

We also believe that improving some functionalities in black-box scanners such as state full scanning, input selection based on field name and label, attack vector novelty, server reply analysis and post scanning, would result in a better detection rate.

VII. RELATED WORK

Many researchers from the academic and private sectors have been putting efforts into web application security. They assessed black-box scanners, tested their performances, analyzed their behavior and gave details about the limitations identified.

Bau et al. [4] evaluated eight black-box scanners and presented their detection capabilities. They also highlighted that no scanner was able to detect stored SQL injection. The authors discussed several ways that scanners’ performance could be improved, such as attacks using novel and non-standard keywords, or using appropriate test vectors for newly discovered vulnerabilities. The authors, however, did not specify whether the black-box scanners used are designed to detect Store SQL injection vulnerabilities. Huang et al. [11] introduced a testing framework and presented the different phases of a testing model, including modeling web application dataflow. More recently, Poue et al. [5] evaluated eleven black-box scanners, and confirmed the poor performance of black-box scanners when challenged with stored SQL injections. Because their custom testbed, WackoPicky [7], was designed to test both the scanner’s capability of detecting vulnerabilities and its performance, a scanner had to succeed at three or more different challenges in order to detect the stored SQL injection vulnerability, since the exploitation, from a scanners’ perspective, was not straightforward (see Section IV.A.). Conversely, our experiments used a custom testbed “MatchIt” that presents no cumulative challenges to black-box scanners. Poue et al. [5] also attempted to determine the errors in a scanner’s cycle, as well as whether a scanner can precisely keep track of the state of an application, which could have an impact on the soundness of detecting stored SQL injections. McAllister et al. [12] proved that guided and stateful fuzzing mechanisms can improve a scanner’s performance regarding stored XSS. They also explained a scanner’s limited capability by relating it to its ability to generate enough requests to reach the vulnerability entry points, and not only its ability to inject malformed input. The authors however did not test their tool against stored SQL injection. Since other reports [4] have shown that black-box scanners are able to detect stored XSS, but not stored SQL injection, we cannot assume that the tool
proposed by McAllister et al. [12] can detect stored SQL injections. The correlation between the mechanisms in detecting stored XSS and stored SQL detection is still an ambiguous area and a potential area of research.

Other papers discussed this advanced type of SQL injection attacks and gave details on where they can be found (log files processing, databases). They also gave recommendations on how to prevent it by using programming best practices and defense mechanisms [3,13].

VIII. CONCLUSION

We assessed the effectiveness of black-box scanners in detecting stored SQL injection. Our tests against three custom testbeds using three black-box scanners show that these scanners are poor at detecting stored SQL injection vulnerabilities.

The major challenges identified start with selecting proper input values. This should be based not only on a field name, but also on its description (e.g., password and repeat password fields). Also, scanners didn’t use proper attack codes to exploit stored SQL injections vulnerabilities. When configured to do so, they failed to do a post scan in order to complete the attack by executing the code. We inferred that the scanner was not aware whether the first step of the attack (storing the attack code) was successful. Even when scanners were taught to execute the attack code, the response analysis function failed to perceive the vulnerability even though a pure SQL syntax error message was received.

Although we were targeting stored SQL injection vulnerabilities, running a scanner with predefined attack code and using a full scanning profile uncovered its capability to detect a medium-severity error on page. This capability, however, was coupled with a failure to correctly categorize this error as a stored SQL injection, or else a failure to relate it to where the code was injected. Based on this, we believe that a full scan would give more efficient results.

ACKNOWLEDGMENTS

We thank Jason Bau and Elie Bursztein for providing us with their testbed PCI that we used in our experiments. We also thank Acunetix, IBM and QualysGuard for their participation in this research.

REFERENCES