Analysis of Effectiveness of Black-Box Web Application Scanners in Detection of Stored SQL Injection and Stored XSS Vulnerabilities

Muhammad Parvez[1], Pavol Zavarsky[1], Nidal Khoury[2]
[1] Information System Security Management
Concordia University of Edmonton, Edmonton, Alberta, Canada
[2] IBM Canada, Toronto, Ontario, Canada
parvezshahar@gmail.com, pavol.zavarsky@concordia.ab.ca, nkhoury@ca.ibm.com

***Abstract*–Stored SQL injection (SQLI) and Stored Cross Site Scripting (XSS) are the top most critical web application vulnerabilities in present time. Previous researches have shown that black-box scanners have relatively poor performance in detecting these two vulnerabilities. In this paper, we analyze the performance and detection capabilities of latest black-box web application security scanners against stored SQLI and stored XSS. Our analysis shows that the recent scanners are showing improved performance in detection of stored SQLI and stored XSS. We developed our custom test-bed to challenge the scanners' capabilities to detect stored SQLI and stored XSS. Our analysis revealed that black box scanners still need improvements in detecting stored SQLI and stored XSS vulnerabilities. In addition to the results of performance tests, the paper provides a set of recommendations that could enhance performance of scanners in detecting stored SQLI and stored XSS vulnerabilities.**

Keywords—stored Cross-Site Scripting; stored SQL injection; XSS vulnerabilities; black-box scanners

I. INTRODUCTION

Stored SQL Injection (SQLI) and stored Cross-Site Scripting (XSS) are the most critical web application vulnerabilities. A stored SQLI attack type involves two steps to complete the attack. In the first step, SQL command is stored in the back-end storage or database. In the next step, the stored SQL command is executed [8].

Stored XSS Injection is a form of attack that allows a malicious user to bypass defined web application rules and policies. This kind of attack takes a form of injecting a malicious code into the database used by a web application. Once the web application fetches data from storage for display on user's browser, the malicious code is executed by the web browser.

To identify existing web application vulnerabilities including the stored SQLI and stored XSS, a variety of techniques are used in order to minimize the likelihood of successful attacks against web applications. Black-box web vulnerability scanning is the one of the techniques that has been widely adopted due to the ease of use, automation, and independence from underlying web application technology.

Recent research revealed that black-box scanners perform relatively poorly in detecting stored SQLI and stored XSS vulnerabilities [2,3,5,8]. Bau et al.[3] and Doupe et al.[5] investigated black box web application security scanners in respect of detecting stored SQLI and stored XSS. The following sections provide new insights on the detection capabilities. Khoury et al.[8] performed an extended analysis on effectiveness of black-box scanners in detecting stored SQLI. Alassami et al. [2] conducted a similar research but for stored XSS vulnerabilities. This paper reports on our findings from replicated tests originally performed by Khoury et al.[8] and Alassami et al.[2].

The overall objectives of this paper are as follows:

i. To measure performance of the state-of-the art of black box scanners in detecting stored SQLI and stored XSS vulnerabilities.
ii. To study reasons for the existing limitations of black box web application security scanners in detecting stored SQLI and stored XSS vulnerabilities.
iii. To present some recommendations on how detecting rate can be improved.

II. RELATED WORK

Performance of black-box scanners has been a concern for a while. Khoury et al.[8] and Alassami et al.[2] conducted research on stored SQLI and stored XSS. Three scanners were evaluated in [2] in detection of stored XSS. The results in [2] demonstrate that the scanners successfully stored script in the test-bed database, but failed to detect stored XSS vulnerabilities. Eight scanners were used in [8] with reported detection rate for stored SQLI of 0% and 15% for stored XSS. Similar results were reported in [5] by testing eleven scanners.

The authors in [8] and [2] performed an expanded analysis for stored SQLI and stored XSS compared to experimental finding in [3] and [5]. Along with other tests, in our performance assessment tests we replicated the tests of Khoury et al. [8] and Alassami et al. [2] by using state-of-the-art Black-Box scanners. In our test-bed, we have included login profile for stored XSS vulnerability and tested the

978-1-908320-52/0/$31.00 ©2015 IEEE

performance of scanners through it. Jason Bau et al [3] wrote, "Indeed, multiple vendors confirmed their difficulty in designing tests which detect second-order vulnerabilities". In detecting second order (stored) vulnerabilities, many factors have to be considered to determine validity of an input.

III. STORED XSS EXAMPLE

To illustrate the different stages of a stored XSS attack, we consider a scenario of exploiting an existing vulnerability in WackoPicko [9]. There is a stored XSS vulnerability in the comments' page of the images. The comment field is not properly sanitized, and therefore, an attacker can exploit this stored XSS vulnerability by the following steps:

- Logs in and visits pages containing images;
- Selects an image to comment on;
- Types a malicious Java Script code in the comment field and submits it for preview; and
- Confirms the comment to be submitted.

When an image page loads, the comment section loads too and attack is executed. The attacker can use any malicious Java Script in the comment field to be stored in the database to steal the session or cookies of a user.

IV. STORED SQLI EXAMPLE

Different stages of a stored SQLI can be explained with a scenario of exploiting an existing vulnerability in WackoPicko. In WackoPicko, users can do registration under /user/registration.php by filling and submitting a form. In this registration form, the first-name field is an unsanitized input text field. While registering for a user, an attacker could store *'or 1 = 1 or "'; DROP TABLE users;#"* in backend database. There is *"similar.php"* page in WackoPicko where a user can search for other users with similar first-name. Authentication is required to visit this *"similar.php"* page. When a user visits *"similar.php"*, a query will be triggered on behalf of the logged in user and that query uses the first name of logged in user as a query parameter. Below is the embedded query code that is used in *"similar.php"* page from WackoPicko application.

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

Once the above query is executed, the $login variable is replaced by the user's first name. If we apply attacker's SQL code *'or 1 = 1'* to the query above, it becomes:

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

The resulting query returns all rows from the users' table since it has the statement *'or 1=1'* which is always true. Instead of the attack code *'or 1 = 1',* other more destructive SQL attack codes can be used.

V. TEST METHODOLOGY

To conduct our experiments, we selected three black-box scanners with claimed capabilities of detecting stored XSS and stored SQLI vulnerabilities. Scanners used in our performance tests are listed in Table I.

TABLE I. LIST OF BLACK BOX SCANNERS USED IN THE TESTS

Scanner	Vendor	Version
Acunetix WVS [1]	Acunetix	8.0
Rational AppScan Enterprise [7]	IBM	9.0
ZAP [10]	OWASP	2.3.1

Black-Box scanners have their preset scanning profiles. A scanning profile can be used to specify which category of vulnerabilities to target during a scan. The scanning phases performed by scanners listed in Table 1 have been verified as the same as described in [5] and [8].

Performance in detecting stored SQLI and stored XSS of each scanner was analyzed in two test-beds: (i) WackoPicko [5] and (ii) our own custom build Scan-bed. Both test-beds were designed to challenge miscellaneous features and functionalities of black-box scanners and present different levels of complexities. Table II and Table III show the number of stored SQLI and stored XSS vulnerabilities in each test-bed along with the number of vulnerabilities that require a user login to be successfully exploited. To exploit the stored SQLI vulnerability in WackoPicko [9], a scanner must execute few steps. The steps are the scanner must (1) register a user account, (2) submit a valid attack code in the first-name field to be stored database while registering a user, and (3) log in using this user account and (4) do a post scan in order to exploit and (5) detect the vulnerability.

TABLE II. STORED SQLI VULNERABILITIES IN TEST-BED

Test-bed	Stored SQLI	Login required	No login required
WackoPicko	1	1	0
Scan-bed	1	0	1

TABLE III. STORED XSS VULNERABILITIES IN TEST-BED

Test-bed	Stored XSS	Login required	No login required
WackoPicko	2	1	1
Scan-bed	1	1	0

In WackoPicko, one stored XSS vulnerability exists in the comment section of all images. To exploit and detect the vulnerability, a scanner must progress through a complex series of steps. The scanner (1) must create a user account and login with the user credentials, (2) should visit *"recent.php"* page to select an image (3) should place a malicious JavaScript code in the comment field and submit it for the preview; (4) confirm the comment; (5) perform a post scan in order to exploit and (6) detect the vulnerability.

"Guestbook.php" has the other stored XSS vulnerability, though the scanner doesn't need to login to detect this vulnerability. The exploitation and detection of stored XSS vulnerability on *"Guestbook.php"* page requires the following steps performed by the scanner:

- Visit the *"Guestbook.php"* page;

- Inject a malformed script in the comment field and submit it to be saved in the database; and
- Revisit pages and analyze the response to confirm the injected script was successfully executed.

The custom build "Scan-bed" contains one stored XSS and one stored SQLI vulnerabilities. The page named *"exploit.php"* contains two text fields named "subject" and "comment" with the comment text field not sanitized. So, the attacker can exploit stored XSS attack using the comment field. The "*exploit.php*" page requires authentication to visit. Scanners can create a user through *"register.php"* page and then visit the *"exploit.php"* page for identification and exploitation of the vulnerability. It allows testing of performance of scanners in detecting stored XSS vulnerability on pages where authentication is required.

We designed a stored SQLI vulnerability in our Scan-bed test-bed. The page named *"information.php"* allows a guest without any privileges to submit their first-name, last-name, gender and age info. The last-name field is an unsanitized input text field. While registering information as a user, attacker could store attack code like *' or 1 = 1 or ''; DROP TABLE users;#''* through last-name field in the backend database. There is another page named *"names.php"* where user can search for other users with the same last-name. When any guest visits *"names.php"*, this page triggers a query on behalf of that guest, and that query uses the last-name of that user as a query parameter. WackoPicko test-bed has a similar type of stored SQLI vulnerability. To exploit or detect the vulnerability in WackoPicko, scanners need to complete a user registration. The simplified scenario in our test-bed allows analysis of the performance of scanners in detecting stored SQLI vulnerability on a page by not requiring scanners to perform any complex tasks like user registration or authentication, which could prevent the scanners from detecting the stored SQLI vulnerability.

A. Black-box web application scanning profiles

For the testing purpose, we used three scanning profiles. (i) Full scanning profile that targets all types of vulnerabilities, (ii) SQL scanning profile targets all types of SQL injection vulnerabilities, and (iii) XSS scanning profile targeting all types of XSS vulnerabilities.

TABLE IV. TEST STEPS

1. Configure web application to its initial state 2. Set database to default state 3. Configure scanners to use specific profile with login credentials as needed 4. Configure Burp suite to monitor the traffic between the scanners and the web server	Preparation
5. Start capturing the traffic 6. Start the black-box scanner	Start
7. Stop Burp Suite and save traffic 8. Save scanning results 9. Export database entries for analysis	Data Collection
10. Check scanner's report 11. Analyze Burp Suite files and database records 12. Repeat the previous steps for new test	Analysis

For each of the scanning profiles, we performed two tests: (1) a test without providing any login credentials to scanners, and (2) a test with providing login credentials. We performed all tests by following the steps summarized in Table IV. These testing steps were adapted from the research of Khoury et al. [8]. We captured all the network traffic from tests targeting the web server using Burp Suite [4]. We analyzed all the traffic to identify attack vectors were used by the scanners. Server replies were also analyzed to determine whether scanners were interpreting replies efficiently or not.

VI. RESULTS AND DISCUSSION

The results and findings are organized into four subsections as follows: (*A*) performance of black-box scanners without authentications; (*B*) performance of black-box scanners with authentications; (*C*) findings with the custom test-bed; and (*D*) special workarounds, like an interactive assistance to scanners to detect and explore vulnerabilities.

A. Performance of black-box scanners without login credentials

The stored XSS and stored SQLI vulnerabilities in WackoPicko are outlined in Section III and Section IV. Black-box scanners do not require authentication to detect the stored XSS vulnerability in WackoPicko's *"guestbook.php"*. To detect the stored XSS in the image comment section, scanners need to login as a user. Also, login is required by scanners to detect the stored SQLI vulnerability in *"similar.php"*.

All three scanners were able to detect stored XSS in "*guestbook.php*" while testing with full profile and XSS profile. All the scanners were able to complete all three phases of the scan. Scanners crawled "*guestbook.php*" page and correctly identified the input field for comment, injected the right attack vectors like malformed scripts through the input field to be saved in backend and analyzed the response from server in a right way to detect the stored XSS vulnerability. It contradicts the finding reported in [2] that scanners were able to successfully identify input fields, inject malicious attack vectors and store it in the backend database but were unable to analyze the response back from server efficiently, which in [2] resulted in failure to detect of stored XSS vulnerabilities. Our performance test confirms that the current versions of scanners are able to analyze server responses and to detect stored XSS vulnerabilities from vulnerable pages that do not require login authentication.

After analyzing the test results of Scanner 1, generated traffic, and used attack vectors, we observed two attack vectors or alert java scripts were used by Scanner 1 to detect stored XSS vulnerability from "*guestbook.php*". One of those alert scripts contained text like *666*. It can be elaborated as follows. When the scanner visited *"guestbook.php",* it inserted a script *alert(666)* to store in the database, and when it revisited the page and analyzed the response, it found *alert(666)* script in the response, which was the script inserted on the first visit. The scanner reports this as a stored XSS vulnerability.

Fig. 1 shows the number of users created by each scanner using a full profile and how many of the user accounts were actually used to scan individual pages where a vulnerability exists. As shown in the figure, the tested scanners were able to create users using *"registration.php"* page. Packet and traffic analysis for *"registration.php"* captured by Burp Suite [4] confirmed that most of the time scanners are giving the same input for password and again-password fields while creating users. Our tests confirmed that the latest scanners were efficient in creating users, which was not the case in the previous research [8] and [5] when half of the scanners from were not able to create users.

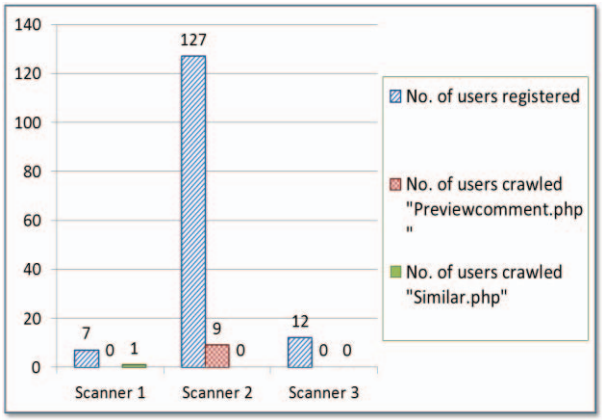

Figure 1. Scanners activity statistics with total registered users for full profile without login credentials

Scanner 1 registered seven users and out of those seven users, only one user was used to visit *"similar.php"* page for once. Scanner 1 successfully crawled *"login.php"*, injected one stored SQLI attack vector through *firstname* input field in the backend database while registering a new user. Later, Scanner 1 logged in with that user account, crawled *"similar.php"* page and executed the stored SQLI attack vector from database. Analysis of the response traffic from *"similar.php"* revealed that Scanner 1 received an SQL syntax error exception but did not confirm any SQL injection vulnerability. The server response analysis phase of Scanner 1 was not efficient. Scanner 1 successfully completed the attack but due to the limitation of server analysis phase, it was unable to detect the stored SQLI vulnerability. After scanning the same test-bed with SQL profile, the scanner detected one SQLI attack in *"login.php"* page based on the same error message as a response from web server. This is a concern regarding performance of the Scanner 1.

As described above, similar to the stored XSS attack that exists on the guestbook page, comments on pictures are susceptible to a stored XSS attack. However, this vulnerability is more difficult to exploit because scanner must be logged in to select an image, submit a malicious script as a comment to be stored in backed database, and to confirm the preview of the comment before the attack is actually triggered. In [2], all scanners used failed in detecting the multi-step stored XSS vulnerability because of scanners lacked an understanding of the interaction between *"previewcomment.php"* and *"comments.php"* pages. In our tests, we experienced an improved performance of Scanner 1 and Scanner 2 when login credentials were provided by us manually for the testing purpose. However, without providing login credentials manually, among all three scanners, Scanner 2 crawled only *"previewcomment.php"*. Scanner 2 used registered user accounts to crawl this page. When analyzing collected data we noticed that a big amount of comments, including XSS attack codes, were submitted for preview by Scanner 2. Scanner 2 did not realize that the submission was just a redirection to a page to confirm the entered comment. As a result, not a single comment was entered by the Scanner 2 through *"previewcomment.php"*.

Scanner 1 just use one account to crawl one protected link *"similar.php"* for once and no accounts were used to crawl protected page like *"previewcomment.php*". Scanner 2 did not use any account to crawl *"similar.php"*. Performance of the Scanner 3 was very poor regarding using registered accounts for the crawling purpose. Though all three scanners were able to create user accounts, the scanners did not use those accounts in an accurate and efficient way to login and crawl protected links. This significantly impacted the scanning results. Besides creating users, scanners should remember the login credentials and use those credentials in an accurate way to craw all login protective links.

After completion of scanning with full profile, XSS and SQL profile without provided login credentials by us were used separately by all three scanners to scan. Table V and VI show the results for XSS and SQL profile respectively. One noticeable fact for Scanner 2, between full profile and XSS profile, only 9 users among 127 created users visited the page *"previewcomment.php"* using full profile, whereas 31 users among 33 created users visited the page using XSS profile. However, in both cases, no single comment was confirmed through *"previewcomment.php"*. Table VI shows that using the SQL profile, Scanners 2 and 3 created 22 and 3 users respectively, but no user crawled the *"similar.php"* page.

TABLE V. SCANNERS ACTIVITY WITH REGISTERED USERS USING XSS PROFILE WITHOUT LOGIN CREDENTIALS

Scanner	No. of users registered	No. of users used in crawling *"Previewcomment.php"*
Scanner 1	1	0
Scanner 2	33	31
Scanner 3	2	0

TABLE VI. SCANNERS ACTIVITY WITH REGISTERED USERS USING SQL PROFILE WITHOUT LOGIN CREDENTIALS

Scanners	No. of user registered	No. of users used in crawling *"similar.php"*
Scanner 1	0	0
Scanner 2	22	0
Scanner 3	3	0

B. Performance of black-box scanners with login credentials

Though user login credentials were provided by us to all three scanners before the scan, scanners also registered users while scanning WackoPicko. Though Scanner 1 registered 133 users by itself, only the user that was manually provided by us was used to crawl login protective pages. Unlike the

performance from the previous section, this time, with our provided login credentials using full profile and XSS profile, Scanner 1 was able to confirm comments successfully through *"previewcomment.php"* page. All the comments successfully confirmed and stored in backend by Scanner 1 contained the text 'abcd'. Interestingly, using the full profile the Scanner 2 didn't confirm any comment through *"previewcomment.php"* but was able to put comment through XSS profile. Using XSS profile, Scanner 2 used 30 users to submit comment for preview. A good amount of comments was submitted for preview by all these users and only one user was successful to confirm one comment through *"previewcomment.php"* page. Interestingly, this single user was created by Scanner 2 itself. Here, we observed XSS profile performed better than the full profile. As a comment, Scanner 2 stored "1" which is not a XSS attack vector. Scanner 3 did not use provided login credentials to crawl *"previewcomment.php"*. Scanner 1 and 2 did not submit any XSS attack vector as comment that could have exploited the vulnerability. If Scanners 1 and 2 had used any XSS attack script instead of the default text, they would have exploited the stored XSS successfully and reported it as a stored XSS vulnerability. Scanners should be improved to trigger right attack vectors in this kind of situation to exploit and detect stored vulnerability.

Analysis of the traffic data revealed that no scanner visited *"similar.php"* with our provided login credentials.

Out of three scanners, only Scanner 1 used the provided login credentials to crawl *"previewcomment.php*" protected page. We can conclude that scanners are not using provided login credentials efficiently which significantly impacts the scanners' performance. The scanners need use the provided login credentials more effectively to crawl not only few but all login-protected pages of the web application.

C. Findings with the custom developed "Scan-bed"

In the "Scan-bed" test-bed, we included one stored SQLI and one stored XSS vulnerabilities. As discussed in Section V, no login is required to exploit the stored SQLI, as opposed to the stored XSS vulnerability that requires authentication.

For each test with scanners, we monitored the crawling network traffic for *"information.php*" to record attack vectors. The traffic is denoted in Table VII as the "Traffic stored". Similarly, the network traffic for "*names.php*" to execute the stored attack vectors is denoted as the "Traffic execute". The numbers of relevant and distinct attack vectors stored in the database for fields vulnerable to stored SQLI are also shown in Table VII.

No scanner is able to detect stored SQLI vulnerability after scanning the page *"information.php"* and *"names.php"*. Scanners did not store any convenient attack code (see Table VII) for the stored SQLI vulnerability. Sufficient amount of traffic was used to use and execute attack vectors, but the attack vectors that were stored in the database were mainly used to exploit and detect blind, reflected or normal SQLI vulnerabilities. Scanners need to be improved to contain appropriate attack vectors to detect stored SQLI vulnerability.

TABLE VII. SCANNERS ACTIVITY IN DETECTION OF STORED SQLI

Scanner	Scanning profile	Traffic store	Traffic Execute	Attack Code
Scanner 1	Full	734	229	0
	SQL	216	76	0
Scanner 2	Full	657	321	0
	SQL	124	58	0
Scanner 3	Full	224	47	0
	SQL	67	23	0

Scanner 2 showed good performance in detecting stored XSS in Scan-bed. All three scanners registered users (as presented in Table VIII), but only eight users by scanner 2 were used to crawl *"comment.php"* where stored XSS vulnerability exists. After doing traffic analysis with scanners, we confirmed that scanner 2 was able to complete all three scan phases of the scan. They crawled the pages, injected the right and convenient amount of attack vectors and analyzed the server response to detect the stored XSS.

TABLE VIII. SCANNERS ACTIVITY IN DETECTION OF STORED XSS

Scanners	Scanning profile	Total Users registered	No: of users crawled "*exploit.php*"	Stored XSS detected
Scanner 1	Full	5	0	No
	XSS	0	0	No
Scanner 2	Full	121	8	Yes
	XSS	32	29	Yes
Scanner 3	Full	6	0	No
	XSS	3	0	No

D. Special workarounds

We performed additional series of experiments with WackoPicko and Scan-bed. We used the following workarounds, using "full profiles", to help the scanners to choose the right attack vectors and to visit the pages where vulnerability exist to exploit and detect the vulnerabilities.

1. This workaround was conducted on WackoPicko. We configured Scanner 2 to register a user containing an attack code in the *firstname* field. We forced the user to visit *"similar.php"* page that would execute the attack code from *firstname* field. The scanner was able to complete attack phases successfully. Analysis of the traffic revealed that scanner received an SQL syntax error exception as a response from the server. Based on this SQL syntax error exception, Scanner 2 reported a SQL injection vulnerability. Similar result was reported in [8].However, while in [8] the scanner did not report any SQL injection for that SQL syntax error exception due to the server response analysis phase of a scanner's cycle, our test results demonstrate improved scanner's functionality regarding server response analysis.

We performed the same test also with our Scan-bed. In *"information.php"* page, we inserted the same attack vector in *last-name* field and forced the scanner to scan *"names.php"*. As a response, the scanner received the same SQL Syntax error but, surprisingly, this time the scanner did not report it as a SQLI vulnerability. We checked the source code of WackoPicko and Scan-bed and found that because of using same mysql error function in coding part, the server is

generating the same SQL syntax error as a response. The ambiguity for the same error in two different test-beds raises questions on the scanner's performance.

2) In another workaround, we configured our browser to use Scanner 3 as a proxy and initiated to browse the WackoPicko. As we are using Scanner 3 as a proxy, every request and response while browsing WackoPicko would go through Scanner 3 and any vulnerability encountered would be reported by the scanner. We visited the *"registration.php"* page and created a user containing the same attack code that we have used in our previous workarounds. Afterwards, we visited the page that would execute the attack code. In response to that one, SQL syntax error exception was shown in that page. This is the same error that had been generated as a response in previous workarounds. However, instead of reporting as any SQLI vulnerability, Scanner 3 reported it as a medium severity application error message. This contradiction in the results raises some questions about the performance and integrity of the scanner results.

3) In section VI *A* and VI *B*, an insights are provided on why all scanners with and without provided login credentials failed to detect the multi-step stored XSS vulnerability in WackoPicko. In this workaround, we created a user account first. We configured the Scanner 1 to use this user account for the login process. We instructed the scanner how to input a malicious XSS attack code in image comment section along with how to confirm the comment submission through *"previewcomment.php"* page. Scanner 1 completed all three scanning phases successfully and detected the stored XSS vulnerability. We can conclude that such help can assist scanners to detect stored XSS.

VII. Recommendations

A) Improvement is required in scanners' functionality to use XSS attack vectors in a more efficient way in detecting stored XSS vulnerabilities. In our tests (See section VI*B*), using "full" and "XSS profile" to scan WackoPicko, Scanners 1 and 2 were successful to confirm comments through *"previewcomment.php"* page. All the comments successfully confirmed by Scanner 1 and 2 contained normal texts. However, the scanners did not submit any XSS attack vector that could exploit the vulnerability. If scanners had used any XSS attack script instead of the default text, it would have exploited the stored XSS and reported it as a stored XSS vulnerability. The scanners need to be improved to use right attack vectors in right situations.

B) Improvement is required in scanners' functionality to analyze server response in a more efficient way to report SQL injection vulnerability. In our tests (see Section VI*A*), for similar SQL syntax error exception message as the server response, Scanner 1 behaved differently in different scanning profiles. While scanning WackoPicko through "full profile", for a SQL syntax error exception message as server response for *"similar.php"* page, Scanner 1 did not report any SQL injection vulnerability. After scanning the same test-bed with SQL profile, Scanner 1 reported one SQLI vulnerability in *"login.php"* page based on the error message response from web server. This also raises concerns regarding performance of the scanner to analyze server responses.

C) Adding interactive multistep options to scan. Out of the three scanners, only one scanner had that feature. While the option to set login credentials helps to provide application's login info for the scanning, it does not instruct the scanners how to visit the pages and where to put attack vectors to exploit and to detect the vulnerabilities.

D) Scanners need to be improved to contain effective attack codes to detect stored SQLI vulnerability.

E) Using login credentials helps scanners to work in a more efficient way. We were able to verify that without provided login credentials, scanners were able to register user account, but out of the registered users only few were actually used to visit the pages which needed authentication.

VIII. Conclusion

In this paper, performance of three black-box scanners in two test-beds in detecting stored SQLI and stored XSS was assessed. Compared to previous reports by other researchers, our results demonstrate that the black-box scanners' performance to detect stored XSS has recently improved and the scanners are able to provide higher detection rate for stored XSS vulnerabilities. However, we were able to confirm that scanners' functionality needs to be improved to correctly analyze server's response on SQL syntax error and to detect server's response as a SQL injection vulnerability. Our experimental findings confirmed that choosing the right attack vectors for the detection and exploitation of stored XSS and stored SQLI remains a big challenge for black box scanners.

References

[1] Acunetix Vulnerability Scanner. [Online]. Available https://www.acunetix.com/vulnerability-scanner/. 12 October, 2014.

[2] S. Alassmi, P. Zavarsky, "Analysis of Effectiveness of Black-Box Web Application Scanners in Detection of Stored XSSI", 2011.

[3] J. Bau, E. Bursztein, D. Gupta, J. Mitchell, "State of the Art: Automated Black-Box Web Application Vulnerability Testing", 2010..

[4] Burp Suite, Available https://portswigger.net/burp/

[5] A. Doup´e, M. Cova, G. Vigna, "Why Johnny Can't Pentest: An Analysis of Blackbox Web Vulnerability Scanners", 2010.

[6] Y. Huang, C. Tsai,, T. Lin, S. Huang, S. Kuo, "A testing framework for Web application security assessment", 2010.

[7] IBM Rational Appscanner. [Online]. Available http://www-03.ibm.com/software/products/en/appscan/. 15 October, 2014.

[8] N. Khoury, P. Zavarsky, D. Lindskog, R. Ruhl, "An Analysis of Black-Box Web Application Security Scanners against Stored SQLI" Proc. IEEE 3rd Int. Conf. Privacy, Security, Risk and Trust, Boston, 2011.

[9] WackoPicko TestBed Application. Online. Available https://github.com/adamdoupe/WackoPicko. Accessed: Nov.29, 2014].

[10] Zed Attack Proxy (ZAP) Scanner. Online. Available https://github.com/zaproxy/zaproxy. 19 October, 2014.

[11] Q. Zhang, H. Chen, J. Sun, "An Execution-flow Based Method for Detecting Cross-Site Scripting", 2010.