A Novice Approach for Web Application Security

Jignesh Doshi and Bhushan Trivedi

Abstract Number of websites hosted increased exponentially in the past few years. More and more organizations are doing their business on web. As a result the attacks on web applications are increased. It is found that about 60 % of web resources are vulnerable. So computer security is critical and important for Web applications. There are various types of solutions exists for mitigating security risks. Developer Skills and efforts are required in most of the solutions. In this paper, the authors have proposed a model for remote database health check. The focus of model is to provide higher level security assessment. The proof of concepts has been implemented using python. The proposed model has been tested on various test scenarios. Authors have also compared model with the topmost 3 vulnerability scanners. The results were found promising and satisfactory.

Keywords Web application attacks · SQLI · Defensive coding · Hardening · Vulnerability scanner

1 Introduction

Computer Security is the biggest challenge of the current era [1–3]. Data and computer systems are key targets of attacks. As per IBM Data Breach Report, 12 % increase in security events year-to-year [4]. Top 10 web application risks remain same in the past few years [1]. Approx. 60 % of attacks are because of vulnerable application code [5].

Most common approaches used to manage web application attacks are defensive coding, hardening (filtering), static/dynamic code analysis or black box testing.

J. Doshi (✉)
LJ Institute of Management Studies, Ahmedabad, India
e-mail: doshijig@gmail.com

B. Trivedi
GLS Institute of Computer Technology, Ahmedabad, India
e-mail: bhtrivedi@gmail.com

N. Modi et al. (eds.), *Proceedings of International Conference on Communication and Networks*, Advances in Intelligent Systems and Computing 508,
DOI 10.1007/978-981-10-2750-5_1

Solutions based application adversely affect cost and developer's efforts [6]. Testing is used for building secure applications. The major problem with testing is that it requires code and web server access [6].

The authors have proposed a security Model to mitigate security risks. Our focus is to develop Model which can be used for web application database health check and act as a utility. Model which neither require developer skills nor code.

The remainder of the paper is formed as follows: Sect. 2 explains the importance of Web application Security and SQL Injection attempts. Section 3 discuss the problem statement (issues), Sect. 4 describe the proposed model, examining results and comparison is provided in Sects. 5 and 6 respectively. The conclusion is provided in Sect. 7.

2 Literature Survey

Injection attack is one of the top three attacks since 2010 [1, 7–15]. SQL Injection and Blind SQL Injection are key attacks under Injection attacks. Most commonly SQL Injection attacks are executed from application using user inputs or URLs [5, 9, 10]. The key impacts of SQLI attacks are data loss, application downtime, brand damage, and customer turnover [7, 11, 16, 17]. Blind SQL Injection attacks are used to List database information and dump data [1]. Both attack use Structure query language for execution of attacks.

Most common approaches used to manage SQLI attacks are defensive coding, hardening (filtering), static/dynamic code analysis, Intrusion detection system and black box testing [8, 6].

Web application communities have developed various approaches for detection and prevention of SQLI [11, 16–19]. Observations of various techniques (existing and proposed) are summarized in Table 1 with reference to efforts, resource requirements (Code and web server) [6, 16, 17, 20–27].

It is observed that most of the solutions require Developer Skills, developer efforts and web server/code access (refer to Table 1).

Gap: A systematic, dynamic and effective solution is required to detect and prevent SQLI [20, 21].

Table 1 Comparison of web application attack solution categories

Approach	Developer		Source code	Web server
	Skill	Effort	Required	Required
Defensive coding	X	X	X	
Static analysis	X	X	X	
Static and dynamic analysis	X	X	X	
Black-box/penetration testing	M	X	X	X
IDS	X	X		X
Hardening	X	X		X

3 Problem Statement

The authors have found that model with following functionalities is required.

(1) Any beginner can run model i.e. no or little technical knowledge is required to execute the model [6, 17, 20–22].
(2) Model work as remote penetration testing i.e. access for source code is not required [6, 17, 20–22].
(3) Web server access is not required i.e. model can be executed from remote PC without installing it on server [6, 17, 20–22].
(4) Model can work as utility [6, 17, 20–22].

4 Proposed Model: Model for Remote Database Health Check

In this research paper, the authors have proposed a novice approach for performing remote database health check (web vulnerability checks).

4.1 Objectives of Model

The objectives of model are to develop model which can work as a utility with minimum technical skills, companies of any size can perform investigations, developers can develop highly secure web applications and organizations can mitigate with web vulnerabilities.

4.2 Overview

Prototype model is developed using python and will focus on top 2 vulnerabilities (SQL Injection and Blind SQL attacks). Model diagram is described in Fig. 1.

Following subsections describe each phase of the proposed model.

(i) *Analyse Web Application* This step will verify the existence of user entered web application host name.
(ii) *Information Gathering* This step describes the process of investigating, examining and analyzing the target website in order to gather information. System Information (like Operating system name, Version etc.), Database Information (like Database Name, Version, table/column Names etc.) and Links (like number of static links, database links mailing and other links) are gathered.

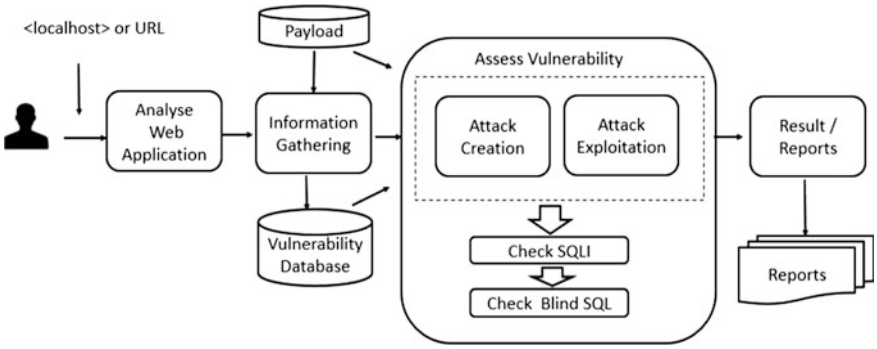

Fig. 1 Remote database health check model diagram

(iii) *Vulnerability Assessment* In this step model will check the vulnerability of web application using data gathered and rule database (payloads) for SQLI and Blind SQL Injection attacks turn by turn. This task is divided into two sub tasks. First, attacks are build using payload i.e. create injection strings using payloads. Then using identified entry points, it will execute attacks. During vulnerability check, Model will check for all types (attack vectors) of attacks. The model is using payload database. Various payloads are used for building and exploiting attacks like Login, Table and column names, attack payload, rule and words. These payloads provide scalability for any new attacks which may found in future.

The authors have prepared a prototype for implementing and testing this model.

5 Testing

5.1 Testing Environment

Figure 2 shows the test environment created for proposed model testing.

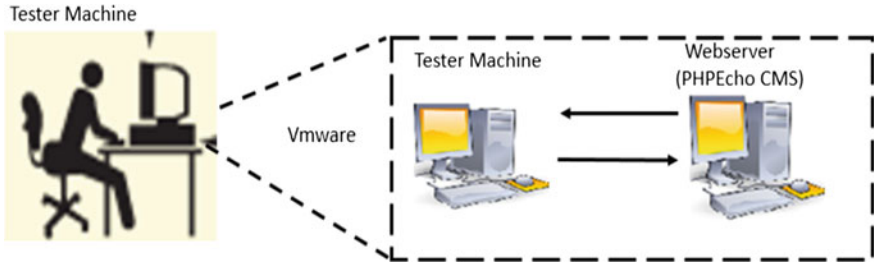

Fig. 2 Test environment

Two virtual machines named VICTIM and HACKER are created on testing machine. PHPEchoCMS web application is deployed on VICTIM machine and proposed model is installed on HACKER Machine. For testing HACKER machine is used.

5.2 Test Scenarios

For proof of concept verification, three test scenarios were considered.

(A) *Test Scenarios 1 PHPEchoCMS*
A Deliberately vulnerable web site is created for testing model using PHPEcho CMS. The first test scenario ran with PHPEchoCMS, a deliberately insecure J2EE web application developed. The purpose of this test campaign to verify and test the proposed model.

(B) *Test Scenario 2*
Custom web applications (developed and hosted on local host). The web site is developed using PHP and database as MySQL. The authors considered two types of websites (static and dynamic) under this scenario.

(C) *Test Scenario 3*
Purpose of this scenario is to execute unit testing of developed screen. Under this category one single login page is developed and testing is performed.

6 Results

6.1 Results

Testing results of above scenarios are summarized in Table 2.

Table 2 Testing results—vulnerability assessment

Test scenario	No. of components	No. of DB links	SQLI	Blind SQL
TS-1: Dynamic (PHPEchoCMS)	25+	11	TP	TP
TS-2: Custom (static)	28+	04	TN	TN
TS-2: Custom (dynamic)	127+	04	TP	TP
TS-3: Dynamic (under development)	1	4	TP	TP
TS3: (Under maintenance)	80+	5	TN	TN

TN True Negative, *TP* True positive

6.2 Performance

Table 3 summarize performance data of all testing scenarios. Performance data shows that proposed model is quick in assessment.

6.3 Comparison

Four parameters used for comparison are Vulnerability coverage (SQLI and Blind SQL), Feature (is solution GUI based), Developer Skill required and developer efforts required. The comparison between proposed model and top 10 open source tools is presented in Table 4.

It is found that

- Only eight out of top ten open source solution provide vulnerability assessment for SQLI and Blind SQL Injection, while proposed model can do for both.
- Six out of top ten open source model do not provide Graphical interface, while proposed model is menu driven

Table 3 Testing results—performance

Application	Duration (s)
TS-1: Dynamic (PHPEchoCMS)	135
TS-2: Custom (static)	119
TS-2: Custom (dynamic)	181
TS-3: Dynamic (under development)	040
TS3: (Under maintenance)	177

Table 4 Comparison—top 10 open source solutions

Name	SQLI	Blind SQL	GUI	Skill/efforts required
Grabber	X	X	No	Yes
Vega	X	X	Yes	Yes
Wapiti	X	X	No	Yes
W3af	X	X	Yes	Yes
Web scarab	X	X	Yes	Yes
Skipfish	X	X	No	Yes
Ratproxy	X	X	No	Yes
SQLMap	X	X	No	Yes
Wfuzz	X		No	Yes
Arachni	X		Yes	No
Proposed model	X	X	Yes	No

- Due to command line interface, technical knowledge is required in most of the open source solution. While proposed model does not require developer efforts for execution.

6.4 Comparison of Proposed Model with Top 3 Vulnerability Scanners

Table 5 describes comparison between proposed mode and top 3 vulnerability scanners.

Table 6 represents resource requirement comparison between Net Sparker and proposed model.

From Tables 5 and 6, we can conclude that proposed model can works with little resource i.e. works as a utility. It do not require developer efforts, skills and configuration. It is easy to use.

Table 5 Comparison—top 3 vulnerability scanners

	Wapiti	OWASP ZAP	Net sparker	Proposed model
Function	Scanner (act as fusser)	Fusser	Scanner	Scanner
Required technical skills	Yes	No	No	No
Requires source code access	No	No	Yes	No
Configuration required	Yes	Yes	Yes	No
False positive	Medium	High	Low	No
SQL injection	Yes	Yes	Yes	Yes
Blind SQL	Yes	Yes	No	Yes
Vulnerability assessment	Yes	Yes	Yes	Yes
Operations	Command line	Auto and manual	GUI	Menu driven
Report	Yes	Yes	Yes	Yes
Purpose	Audit	Detect training	Detect/exploit	Detect and exploit

Table 6 Comparison—resource requirement

	Net sparker	Proposed model
RAM requirement	1 GB RAM (min.)	<512 MB
HDD	100 MB + 100 Mb per scanning and 4.2 GB per scan	1 MB
Installation	Yes	No
Developed using	.NET	Python 2.X
Platform	Windows	UNIX/Windows

7 Conclusion and Future Work

Some investigation challenges for web vulnerabilities are exemplified in the proposed model. It provides bases for utility. The model emphasizes on the requirements of changes needed in Vulnerability risk mitigation using a light weight utility. To address challenges of multi tenancy of web application, Authors have proposed a logging mechanism, which can be useful to address known as well as unknown threats.

One of the key characteristic of Model is that it does not try to obtain sensitive data. However, it extracts weaknesses to prepare attacks and evaluate web application for vulnerability. The attack results are collected which can be used for further analysis and code fix. As mentioned Model neither need code nor server access to determine. Authors can run from any PC by giving an URL to the health check.

Authors can conclude that they have successfully tested web applications using proof of concept. The performance found was excellent. Model correctly identified vulnerability in web applications.

References

1. OWASP. Top Ten project 2013: https://www.owasp.org/index.php/Category: OWASP_Top_Ten_Project: accessed 31st May 2014.
2. Internet user statistics: http://www.internetworldstats.com/stats.htm: visited on 23rd November 2014.
3. Internet user in world: http://www.internetlivestats.com/internet-users/: visited on 23rd November 2014.
4. IBM Data Breach Statistics: http://www-935.ibm.com/services/us/en/it-services/security-services/data-breach/; visited on 23rd November 2014.
5. Eugene Lebanidze: Securing Enterprise Web Applications at the Source: An Application Security Perspective: https://www.owasp.org/images/8/83/Securing_Enterprise_Web_Applications_at_the_Source.pdf, pp. 1, 15, 32.
6. Jignesh Doshi, Bhushan Trivedi, Assessment of SQL Injection Solution Approaches, IJARCSSE, October 2014, Vol 4, Issue 10, ISSN: 2277 128X.
7. White Paper: Cutting the Cost of Application Security: An ROI White Paper: https://www.imperva.com/lg/lgw.asp?pid=349: accessed 31st August 2014.
8. Robert Richardson, "15th Annual 2010/2011 Computer Crime and Security Survey", 2011: gatton.uky.edu/FACULTY/PAYNE/ACC324/CSISurvey2010.pdf: accessed 1st December 2014.
9. Michael Howard, David LeBlanc and John Viega. "24 Deadly Sins of Software Security". "Sin 1: SQL Injection." Page 3–27. McGraw-Hill. 2010.
10. Nina Godbole and Sunit Belapure, "Cyber Security: Understanding Cyber Crimes, Computer Forensic and Legal Perspective", Wiley India Pvt. Ltd, First Edition 2011.
11. WG Hallfond, J Viegas and A Orso: "A Classification of SQL Injection attacks and Countermeasures", IEEE 2006.
12. Z. Su and G. Wassermann, "The Essence of Command Injection Attacks in web Applications", The 33rd Annual Symposium on Principles of Programming Languages (POPL 2006), 2006.

13. Open Web Application Security Project (OWASP) SQLI page: http://www.owasp.org/index.php/SQL_Injection; last visited 28th Aug 2014.
14. National Vulnerability Database (NVD) Security Checklists: http://web.nvd.nist.gov/view/ncp/repository; last visited 28th August 2014.
15. Common Weakness Enumeration: http://cwe.mitre.org/data/definitions/89.html: accessed 3rd August 2014.
16. Rahul Johri and Pankaj Sharma "A Survey on Web Application Vulnerabilities (SQLIA and XSS) Exploitation and Security Engine for SQL Injection", IEEE 2012.
17. A. Tajpour, M. Masrom and M. Z. Heydari, "Comparison of SQL Injection Detection and Prevention Techniques", 2nd International Conference on Education Technology and Computer (ICETC), 2012.
18. Chad Dougherty, Practical Identification of SQL Injection Vulnerabilities: www.uscert.gov/sites/default/files/publications/Practical-SQLi-Identification.pdf.
19. Carnegie Mellon University, Computing Services Information Security Office, Information security 101: www.cmu.edu/iso/aware/presentation/security101-v2.pdf: visited on 23rd November 2014.
20. A. Tajpour, M. Masrom and M. Z. Heydari, S Ibrahim: "SQL Injection Detection and Prevention Tools Assessment", IEEE 2010, 978-1-4244-5540-9 Aug.
21. A. Tajpour, M.J Shooshtari: "Evaluation of SQL Injection Detection and Prevention Techniques": 2010-Second International Conference on Computational Intelligence, Communication Systems and Networks: 978-0-7695-4158-7/10. doi:10.1109/CICSSyn, 2010 IEEE.
22. Diallo Abdoulaye and Al-Sakib Khan Pathan, "A Survey on SQL Injection: Vulnerabilities, attacks and Prevention Techniques", IEEE 15th International Symposium on Consumer Electronics, 2011.
23. William G.J. Halfond, Allesandro Orso, "AMNESIA: Analysis and Monitoring for NEutralizing SQL Injection Attacks", ACM, USA, 2005, pp 174–183.
24. Bisht, P., Madhusudan, P., and Venkatakrishnan, V.N., CANDID: Dynamic Candidate Evaluations for Automatic Prevention of SQL Injection Attacks. ACM Transactions on Information and System Security, Volume 13 Issue 2, (2010). doi:10.1145/1698750.1698754.
25. Sam M.S. N.G, "SQL Injection Protection by Variable Normalization of SQL Statement". www.securitydocs.com/library/3388, 06/17/2005.
26. Buehrer, G., Weide, B.W., and Sivilotti, P.A.G., Using Parse Tree Validation to Prevent SQL Injection Attacks. Proc. of 5th International Workshop on Software Engineering and Middleware, Lisbon, Portugal (2005) 106–113.22.
27. Kemalis, K. and T. Tzouramanis. SQL-IDS: A Specification-based Approach for SQLinjection Detection. SAC'08. Fortaleza, Ceará, Brazil, ACM (2008), pp. 2153–2158.
28. Stop SQL Injection Attacks Before They Stop You: http://msdn.microsoft.com/en-us/magazine/cc163917.aspx: accessed 3rd August 2014.