2012 IEEE 36th International Conference on Computer Software and Applications Workshops

Attack model based penetration test for SQL injection vulnerability

TIAN Wei, YANG Ju-Feng*, XU Jing, SI Guan-Nan

College of Information Technical Science
Nankai University
Tianjin, China

State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing, China

***Abstract*—The penetration test is a crucial way to enhance the security of web applications. Improving accuracy is the core issue of the penetration test research. The test case is an important factor affecting the penetration test accuracy. In this paper, we discuss how to generate more effective penetration test case inputs to detect the SQL injection vulnerability hidden behind the inadequate blacklist filter defense mechanism in web applications. We propose a model based penetration test method for the SQL injection vulnerability, in which the penetration test case generation is divided into two steps: i) Building model for the penetration test case, and ii) Instantiating the model of penetration test case. Our method can generate test case covering more types and patterns of SQL injection attack input to thoroughly test the blacklist filter mechanism of web applications. Experiments show the penetration test case generated by our method can effectively find the SQL injection vulnerabilities hidden behind the inadequate blacklist filter defense mechanism thus reduce the false negative and improve test accuracy.**

Keywords-penetration test; vulnerability; SQL injection; test case; attack model

I. INTRODUCTION

Penetration test [4] is an increasingly significant security test way for detecting web vulnerability, including the SQL injection. The philosophy behind penetration test is exposing vulnerabilities through testers' mock attack to software before attackers' real exploitation to it [11].

The information gathering, attack generation and response analysis are three key basic phases of penetration test [4]. The current academic researches on penetration test for SQL injection vulnerability mainly focus on the information gathering and/or response analysis phases:

For the information gathering phase, the web crawling techniques are very popular for they are easy to use and quickly provide information about web applications. The [8] , for example, discuss apply the improved crawler technology to find more injection points in testee web applications; nevertheless William Halfond et al. [4] point out the web crawling cannot provide the guarantee of completeness with respect to the information they collect, so they propose an information gathering method based on source code analysis (non-crawler way), their empirical results show this approach can exercise the subject applications more thoroughly and discover a higher number of vulnerabilities.

For the response analysis phase, Nuno Autunes et al. [7] propose a test accuracy improvement approach for detecting SQL Injection and XPath Injection vulnerabilities by comparing the structure of the SQL/XPath commands in and out attacks; Reference [4]also presents a new SQL injection response analysis approach in which a query is parsed right before it is issued to the database to check whether it's a successful attack, if it is, that means an attack can break through defenses then a vulnerability exists.

Improving testing accuracy is the ultimate goal and core task of the penetration test researches. Various related work, in effect, all serve for this ultimate goal directly or indirectly. The test accuracy implies the degree of the false positives (reported vulnerabilities that in fact do not exist) and the false negatives (existing vulnerabilities not been reported) contained in the test results. So reducing the false positives and false negatives are two essential aspect of accuracy improvement, which accordingly needs the study efforts to all three key basic phases of penetration test.

Most of the researches on penetration test for the SQL injection, however, pay little attention to the attack generation phases. Especially, the test case inputs (attack pattern library) in this phase are often treated as the external factors in related work [4][7][9]. The regularity and adequacy of test case inputs and their impact on test accuracy are not yet well studied. In fact, the test case input is an important factor affecting the test accuracy: an inadequate test case inputs set cannot fully test the software defenses mechanism and trigger certain vulnerabilities, thus unable to find the security vulnerabilities hidden behind the inadequate defense mechanism, which causes the false negative and impairs the test accuracy.

To address this problem, we focus on the research on the penetration test case inputs. We propose a model based testing methods for the SQL injection vulnerability, in which the penetration test case generation is divided into two steps:

1) Building model of the penetration test case

In the Step 1, we build attack models to abstractly describe the regularity of SQL injection, and under the guidance of these attack models, we establish the model of penetration test case inputs for SQL injection vulnerability.

2) Instantiating the penetration test case model

In the Step 2, we instantiate the test case model according to certain coverage criteria to generate executable test cases. In this study, we focus on the test to the blacklist filter [10] which is a common defense mechanism against the SQL injection attack. We propose several new coverage criteria to generate test cases with more patterns to more

978-0-7695-4758-9/12 $26.00 © 2012 IEEE
DOI 10.1109/COMPSACW.2012.108

thoroughly test the blacklist filter mechanism and find the SQL injection vulnerability hidden in the inadequate defense.

In brief, the step 1 reveals what test case should be used; the step 2 expounds how many test cases should be used.

Experiments show that compared with the irregular test case used in other works, the test case generated by our method can test the blacklist filter mechanism more thoroughly, and more effectively find the SQL injection vulnerabilities hidden behind the inadequate blacklist filter defense mechanism, and thus reduce the false negative and improved test accuracy.

II. SECURITY GOAL MODEL FOR SQL INJECTION ATTACK

An SQL injection attack takes place when a hacker changes the semantic or syntactic logic of an SQL text string by inserting SQL keywords or special symbols within the original SQL command, executed at the database layer of an application. The SQL injection vulnerabilities can be divided into two sub-classes: first-order and second-order. First order SQL injection vulnerabilities result in immediate SQL command execution upon user input submission, while the second order SQL injection requires unsanitized user input to be loaded from the database [3]. To simplify our discussion, we choose the first-order SQL injection vulnerability as the research subjec. In the following sections, we refer the "SQL injection" as the first-order SQL injection if not specified.

In the related academic researches, the test case is generally defined as a triple: t = (Pre, In, Out), the Pre is the precondition, the In denotes the inputs, and the Out is the expected output; the Pre often defaults to TRUE and ignored.

For the penetration test case of the SQL injection vulnerability, we set the In as the attack inputs (attack pattern library), the Out as the vulnerable response of the testee web application if it has the SQL injection vulnerability. To determine the contents of In and Out, we next build the attack models of SQL injection to describe the types and regularity of the In and Out.

The current researches on modeling of SQL injection attack generally apply the attack tree to describe the attack behavior. Reference [5], for example, propose an augmented attack tree model for the SQL injection attacks; In [6], the attack tree is used to describe the SQL injection attack process to a target web application, and accordingly generates the security test sequence to this web application.

The current researches on SQL injection attacks modeling are still few. And the models mentioned above can't desirably reflect the logic regularity of SQL injection attacks. The SQL injection attack tree model proposed in [5], for example, mainly focuses on the description of SQL injection attack inputs (signature) by regular expressions. This leads to the description of model cannot be widely applicable to all attack scenarios and cannot reflect the purpose of attackers' injection; The model proposed in [6] only abstracted the test process for a particular web application and didn't reflect the regularity of attack inputs.

Therefore, it's necessary to model the SQL injection attacks by using new modeling approach with better description ability.

The security goal model (SGM) is a new modeling method that can be used to describe vulnerabilities, security property, and attack or security software development [2]. We apply the SGM to model the SQL injection for better reflecting the regularity of SQL injection vulnerability in web applications. The detail meaning of symbols (vertex, directed edge etc.) in the SGM has been explained in [2], so we don't repeat it due to space constraint.

The SGM-based SQL injection attack models we proposed in Fig. 1 ~ 3 demonstrate the goal-based regularity of the SQL injection from a general to specific view: First, the Fig. 1 models the SQL injection attacks in a general view, the SQL injection attacks are divided into three general categories according to the immediate attack goals of attackers: the Steal System Information, Bypass Authentication, and Remote Command Execution. Based on this categories, the Fig. 1 also describes the detail attack behaviors (subgoals) constituting the three categories from a Bottom-to-up view, till the vertexes of "injecting" attacks in the most top of the model.

We use SGM to further detail the subgoal vertexes of the "steal system information" and the "Remote Command Execution" in Fig. 1 model. The models in Fig. 2 are the unfolding description of the "steal system information" subgoal vertex, in which we model this attack subgoal as two detailed part: (a) error message utilizing, and (b) blind injection. Likewise, the model in Fig. 3 unfolds the "executive malicious command" subgoal vertex in detail, which models this subgoal as two detailed part: (a) SQL command injection and (b) non-SQL command injection.

The models in Fig. 2~3 are the sub-models describing the detail regularity of subgoals in Fig. 1 model. They describe the main methods of SQL injection attack and then provide the guidance to the research on penetration test cases inputs.

Each Top-down path in Fig. 1~3 models represents a attack process realizing a certain attack subgoal: the subgoal vertexes labeled "injecting" on the most top of the models describe the attack inputs, the subgoal vertexes in the middle tier of the models (rectangles without black fill color) describe the WEB vulnerable responses to the "injecting" attacks if it has SQL injection vulnerability, and the subgoal vertexes with black ground color (the counteracting subgoal) describe the unsuccessful attack blocked by the "adequate defense mechanism of WEB" .

Define each top-down successful attack process (without the counteracting subgoal in path) in Fig. 2~3 as the attack scheme. (Put aside the consideration of input vectors[4])

Definition 1: the Attack scheme is defined as a triple: <OBJ,INP,OUT>, the OBJ denotes the attack goal, INP is the attack input, and the OUT is the vulnerable response of web application (the feature of successful attack). Our study focuses on the penetration test case inputs, so we mainly discuss the INP and research the modeling of the attack pattern library [9] in the following.

III. THE MODELING OF THE PENETRATION TEST CASE INPUTS FOR SQL INJECTION VULNERABILITY

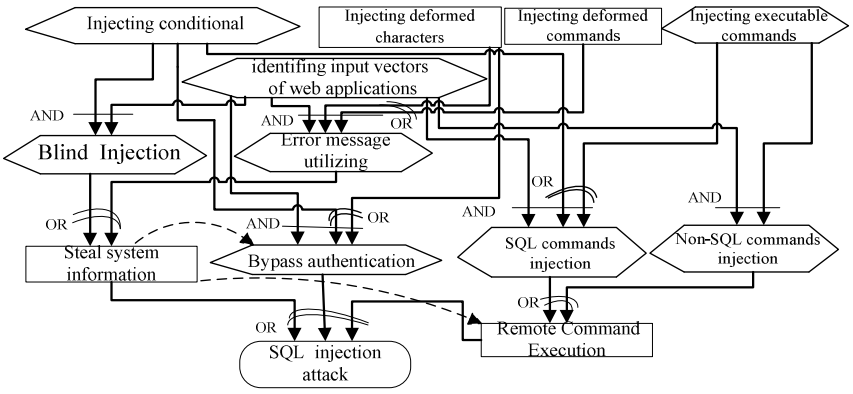
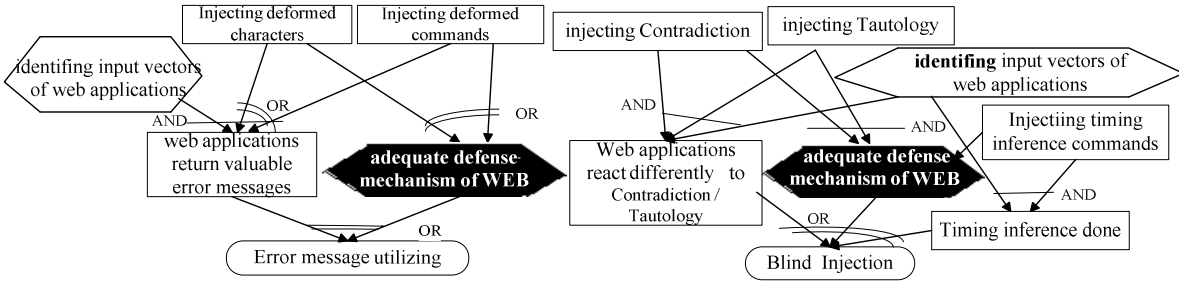
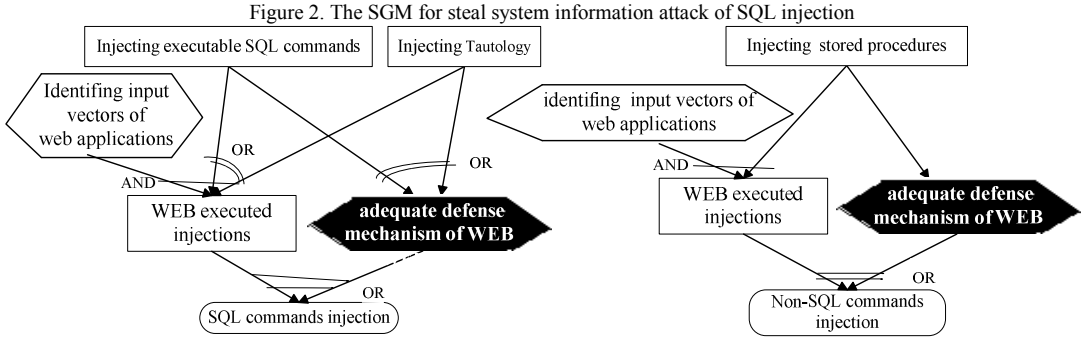


Figure 1. The SGM for SQL injection attack



(a) error message utilizing attack (b) blind injection attack

Figure 2. The SGM for steal system information attack of SQL injection



(a) SQL commands injection attack (b) Non-SQL commands injection attack

Figure 3. The SGM for Remote Command Execution attack of SQL injection

A. The symbol of injection attack inputs

We define the symbol of injection attack inputs: ∇. The ∇_i represents an attack inputs set of a certain kind i. For example, the ∇_{SQL} represents the inputs that can be used in SQL injection attack. The symbols defined for SQL injection attack inputs are listed in table Ⅰ. These symbols describe the attack inputs from the perspective of attack goals.

B. The symbol expressions of SQL injection attack inputs

Based on the symbol definition in table I, we joint these symbols together to form the symbols expressions. Therefore, we define the expressions operators of the symbols, including: $||$ as the OR operator and the && as the AND operator. That is, $\nabla_1 || \nabla_2 = \{x \mid x \in \nabla_1 \vee x \in \nabla_2\}$, which means you may use the attack inputs represented by ∇_1 and/or ∇_2 for an attack goal; $\nabla_1 \&\& \nabla_2 = \{x, y \mid x \in \nabla_1 \wedge y \in \nabla_2 \vee x \in \nabla_2 \wedge y \in \nabla_1\}$. Define "•" as the compound operator of symbols. The compound operation of symbols ∇_1 and ∇_2 is denoted as $\nabla_1 \bullet \nabla_2$, which means the symbol ∇_1 disposes the attack inputs represented by ∇_2 in certain operation and the $\nabla_1 \bullet \nabla_2$ produces new composited attack inputs.

Define the priority of above operators: the operation precedence of the compound operator • is from right to left: the $\nabla_1 \bullet \nabla_2 \bullet \nabla_3$ means $\nabla_1 \bullet (\nabla_2 \bullet \nabla_3)$; The && has higher priority than $||$, so the $\nabla_1 || \nabla_2 \&\& \nabla_3$ is equivalent to $\nabla_1 || (\nabla_2 \&\& \nabla_3)$.The priority of compound operator • is higher than both && and $||$. The parentheses have the highest priority. According to the definition of Symbols and the description of SGMs in Fig. 2~3, we detail the attack scheme in **Definiton 1** in TABLE II. :

We describe the SQL injection attack inputs (INP) in symbol expressions to present the infinite real attack inputs through finite symbols, which solves the problem of

infiniteness and irregularity of test case (attack inputs) enumeration.

TABLE I. SYMBOLS FOR SQL INJECTION ATTACK INPUTS

Symbol	Definition	Examples *
∇_{DS}	Deformed characters Set	'/"/;and user>0/admin' --/@@vesion;--
∇_{LG}	Select authentication-interference deformed characters from Deformed characters Set	admin' --/ '--/ /**/
∇_{DC}	Deformed commands Set	having 1=1--； / union select @@version；
∇_{CON}	Conditional Set	=/<>/</>/<=/>=/IN/BETWEEN/LIKE/!=/ 1=(select count(*) from admin where len(name)> 5)；
∇_{IE}	Select Tautology from conditional Set	1=1 / 'd'<'s' / 'b' in ('a','b') / something' like 'some%'
∇_{NE}	Select Contradiction from conditional Set	1=2/'d'>'s'/'b' not in ('a','b')
∇_{TI}	Build timing inference commands with conditional Set	if (*conditional*) waitfor delay '0:0:10' / SELECT pg_sleep(10) / IF (*conditional*) BENCHMARK(100,MD5(1))
∇_{COMS}	Executable Commands Set	;alter …… add test date;-- /' ;exec master..xp_cmdshell ……
∇_{SQLC}	Select Executable SQL commands from Executable Commands Set	; select * from …… / UNION select …… from ……； / ;insert into ……vaules(……)--
∇_{SP}	Select Stored Procedures from Executable Commands Set	;exec master..xp_blank>_cmdshell dir / ;exec master..xp_cmdshell
∇_{AND}	Inject conditional with AND relation	AND 1=2
∇_{OR}	Inject conditional with OR relation	OR 49>45
∇_{disg}	disguise injections through Encoding/ Deformation etc。	sEleCt / %73%45%6C%65%43%74

* Each example is separated by "/ "

TABLE II. ATTACK SCHEMA SET OF THE SQL INJECTION

#	OBJ	INP	OUT
m1	Erorr message utilizing	$\nabla_{DS} \| \nabla_{DC} \| \nabla_{disg} \bullet (\nabla_{DS} \| \nabla_{DC})$	Web applications return valuable error messages
m2	Blind Injection	$\nabla_{TI} \bullet \nabla_{CON} \| \nabla_{AND} \bullet \nabla_{IE} \bullet \nabla_{CON} \&\& \nabla_{AND} \bullet \nabla_{NE} \bullet \nabla_{CON} \| \nabla_{disg} \bullet (\nabla_{TI} \bullet \nabla_{CON} \| \nabla_{AND} \bullet \nabla_{IE} \bullet \nabla_{CON} \&\& \nabla_{AND} \bullet \nabla_{NE} \bullet \nabla_{CON})$	Web applications react differently to Contradiction / Tautology
m3	SQL command s injection	$\nabla_{SQLC} \bullet \nabla_{COMS} \| \nabla_{OR} \bullet \nabla_{IE} \bullet \nabla_{CON} \| \nabla_{disg} \bullet (\nabla_{SQLC} \bullet \nabla_{COMS} \| \nabla_{OR} \bullet \nabla_{IE} \bullet \nabla_{CON})$	Web applications executed injections
m4	Non-SQL command s injection	$\nabla_{SP} \bullet \nabla_{COMS} \| \nabla_{disg} \bullet (\nabla_{SP} \bullet \nabla_{COMS})$	Web applications executed injections
m5	bypass authentica tion	$\nabla_{OR} \bullet \nabla_{IE} \bullet \nabla_{CON} \| \nabla_{LG} \bullet \nabla_{DS} \| \nabla_{disg} \bullet (\nabla_{OR} \bullet \nabla_{IE} \bullet \nabla_{CON} \| \nabla_{LG} \bullet \nabla_{DS})$	pass the authentication of web application

The formal description (or model) of the INP are created according to the description of the "injecting" vertexes in attack models in Fig. 2~3. We model their contents and usage by the symbols expressions. In the following test, we assign the model of INP in Table 2 as the test case model.

TABLE III. AFFILIATION RELATIONSHIP BETWEEN INPUT VECTORS AND ATTACK SCHEMES

Input vectors	Attack schemes
URL GET parameters	m1. m2. m3. m4.
LOGIN FORM	m1. m3. m4. m5

IV. THE INSTANTIATION OF TEST CASE MODEL

In this section, we implement the step 2 of test case generation: instantiating the test case model according to the *fingerprint* of web application and certain *coverage criteria* to generate executable test cases.

To simplify discussion, our research focuses on the latter: the *coverage criteria* of test case. To test whether the blacklist defense can adequately block various patterns of SQL injection attack inputs, the definition principle of penetration test case coverage criteria should be: to make test case inputs cover more of attack input patterns. We propose a series of new coverage criteria to generate the penetration test case covering diverse patterns.

Definition 3: the Command Verbs Coverage Criterion:

$$\forall mv(mv \in MV \rightarrow \exists tc \in TC \wedge < tc, mv >)$$

The MV denotes the executable Command Verbs set that should be used in the test for a testee web application; the TC denotes the test case inputs set; and the <tc,mv> means the command verb *mv* is used in the construction of test case input *tc*, and no other command verbs used (contained) in *tc*.

Definition 4: the Relation Predicate Coverage Criterion:

$$\forall op(op \in OP \rightarrow \exists tc \in TC \wedge < tc, op >)$$

The OP denotes the SQL syntax Relation Predicate set that should be used in test; the <tc,op> denotes the predicate *op* is used in the construction of test case input *tc*, and no other predicates used (contained) in *tc*.

The illegal coverage ratio (ICR) and random coverage ratio (RCR), two methods of selecting finite test cases from the infinite number of illegal values, are described in [11]. The ICR chooses equally distant values based on the total range of illegal values. The RCR selects the illegal values arbitrarily from the available set of illegal inputs.

For the instantiation of the $\nabla_{SP} \bullet \nabla_{COMS}$ set that contains large number of command verbs, we also used the RCR method to randomly select a certain amount of stored procedures command verbs as the elements of the MV set (**Definition 3**) and apply the Command Verbs Coverage Criterion to generate test case. The number of randomly selected test case is determined according to the test scale.

The instantiation of the ∇_{AND} and ∇_{OR} means adding the keywords of AND or OR at the start of the conditionals. For the instantiation of $\square_{disg}$, the test coverage criterion requires

each selected disguise method of attack inputs is used in test case set. That is, for the □i generated by □ disg • TC:

Definition 5: Disguise Methods Coverage Criterion:

$$\forall dg(dg \in DG \rightarrow \exists tc \in TC \wedge < tc, dg >)$$

The DG denotes the disguise methods set that should be used in test; the <tc,op> denotes test case *tc* is disguised by the method *dg* and added into the ∇_i.

V. CASE STUDY

This study focus on the adequacy of penetration test case inputs for the SQL injection vulnerability, namely, the ability of test case inputs breakthrough the inadequate blacklist defense mechanism to find the SQL injection vulnerability hide behind it. So to verify the proposed test case generation methods, we need the subject web applications with the known number and locations of SQL injection vulnerability and the inadequate blacklist defense. We found existing applications did not satisfy our requirements. Therefore, we build an experimental platform as follows:

A. Experiment subjects

We create two web applications as the test experiment subject: a JSP and an ASP website. The HTML code are applied and the SQL server 2000 SP4 as their back-end database. The JSP subject has around 5500 lines of code and the ASP subject has around 15000 lines of code. Their function and structure reflect the traits of real common web applications, so the penetration test for them can be regarded as the representative of the actual web application testing.

TABLE IV. THE DEFENSE LEVEL OF SUBJECT WEB APPLICATIONS

Level of defense	defense mechanism
Level 0	NO defense mechanisms
Level 1	The blacklist filter (inadequate)

We apply the "levels of defense [10]" testing evaluation method, setting two levels of defense against the SQL injection attack in the channels (the ①-⑪ in table VI and VII) subject applications accessing back-end databases.

The channels adopting Level 0 have the SQL injection vulnerability caused by completely no defense to users' input; while the channels adopting Level 1 (the keywords in blacklist filter are not sufficient: only part of the possible attack keywords are contained in the blacklist, thus some other attack inputs can escape the filter) have the "SQL injection vulnerability hidden behind the inadequate defense mechanism", which is our research object. That means we add two types of SQL injection vulnerability to the subject web applications and take them as the benchmark for assessing the effect of different test cases in the following.

B. Setting for test case instantiation

We create an automated web application SQL injection vulnerability penetration test tool (NKSI scan): it apply the widely used "crawling - attack – analysis" method [1] to detect the SQL injection vulnerability in subject applications. Its crawler module traverses the testee web applications to access all the reachable pages and parse out the input vectors (URL GET parameters and login forms here) contained in these pages, next its attack module submits instantiated test case inputs to corresponding input vectors found by crawler module, and then its analyzing module analyzes the application's response to judge whether a vulnerability has been trigged

We instantiate the test case model (INP in Table 2) to generate real test case inputs according to the discussion in section IV. Based on the setting in TABLE V. , we artificially generate approximately 103 test cases inputs for the attack module of NKSI scan,.

TABLE V. SETTING FOR TEST CASE INSTANTIATION

symbols	Coverage criteria	Setting for instantiation
∇_{DS}	RCR	generating 5 random strings
$\nabla_{SQLC} \bullet \nabla_{COMS}$	the Command Verbs Coverage Criterion	MV={select, create, insert, update, delete, drop, alter}
$\nabla_{SP} \bullet \nabla_{COMS}$	the Command Verbs Coverage Criterion	MV={ Three randomly selected stored procedures command verbs }
$\nabla_{TI} \bullet \nabla_{CON}$	the Command Verbs Coverage Criterion	MV={ waitfor, benchmark, sleep}
$\nabla_{IE} \bullet \nabla_{CON}$, $\nabla_{NE} \bullet \nabla_{CON}$	the Relation Predicate Coverage Criterion	OP={<>, <,>,<=,>= , IN , LIKE ,between }
∇_{disg}	Disguise Methods Coverage Criterion	DG={ UNICODE encoding, Uppercase/lowercase blend, ASCII encoding }

C. Test evaluation

We choose two famous and often selected test tools: the Acunetix 6.5 and IBM Rational AppScan 7.7 for comparison. We refer them as the Tool A and Tool B here (with no particular order) to avoid the particular brand comparison. The Tool A and B can be regarded as the representatives of the test cases enumeration method commonly used in the current researches: many researches compile and cite the attack pattern library of these commercial tools for their test. We compare our method with them to show the superior performance over existing approaches in terms of test case.

In TABLE VI. and TABLE VI. , each vulnerable page represents a SQL injection vulnerability that should be detected (√ means this vulnerability can be detected).

We don't use the crawling-challenge devices in our two subject web applications (e. g. JavaScript, Flash etc.), so the three tools all can automatically find the pages ①~⑪ and their associated input vectors. Therefore, the test result is independent of their input vectors finding ability. Under the circumstances, for the input vectors with no defense (Level 0), three tools all can thoroughly detect their vulnerability (①②⑥⑦), while for the input vectors with inadequate defense (Level 1), the Tool A and Tool B generate false negatives: some vulnerable pages cannot be identified, our NKSI scan can detect these pages and find the SQL injection vulnerabilities hidden behind the inadequate defense (Level 1). The main reason lies in their test case:

1) The test case of NKSI scan are generated under the guidance of the attack model and the test case model, which can guide generating test case reflecting more kinds of attack methods. While the test case of the Tool A and Tool B are

mainly collected from known attack inputs, so they are difficult to ensure the full consideration for various kinds of attack inputs;

2) The test case of NKSI scan covers more patterns of SQL injection attack inputs than the Tool A and Tool B. The test result shows that: the simple test case of Tool A and Tool B are blocked by the inadequate defense (Level 1), so they conclude the testee input vector is protected and not vulnerable. So the simple test case prevents them finding some vulnerabilities hidden behind the inadequate defense and thus generate the false negative. While our test case are generated under the coverage criteria in Definitions 3-5 to cover more attack inputs patterns than simple test case, so they are more capable of breaking through the inadequate defense and detect the hidden SQL injection vulnerability.

In TABLE VIII. ,the total testing time includes the execution time of crawling-attack-analysis of three tools and doesn't include the time of artificially instantiating test case of NKSI scan. It shows our test case with larger size doesn't cause excessive time consumption compared with Tool A and B, which confirms its feasibility.

TABLE VI. TEST RESULT OF THE SQL INJECTION VULNERABILITY IN THE JSP TESTEE WEB APPLICATION

Level of defense	*Vulnerable pages*	*Type of input vectors*	*NKSI scan*	*Tool A*	*Tool B*
Level 0	①http://192.168.111.222:8080/mynews/admin/login1.jsp	LOGIN FORM	√	√	√
	②http://192.168.111.222:8080/mynews/ViewNews1.jsp	URL GET parameters	√	√	√
Level 1	③http://192.168.111.222:8080/mynews/admin/login2.jsp	LOGIN FORM	√		√
	④ http://192.168.111.222:8080/mynews/ViewNews2.jsp	URL GET parameters	√	√	
	⑤http://192.168.111.222:8080/mynews/ViewNews.jsp	URL GET parameters	√		

TABLE VII. TEST RESULT OF THE SQL INJECTION VULNERABILITY IN THE ASP TESTEE WEB APPLICATION

Level of defense	*Vulnerable pages*	*Type of input vectors*	*NKSI scan*	*Tool A*	*Tool B*
Level 0	⑥http://192.168.111.222/chat/useradmin.asp	LOGIN FORM	√	√	√
	⑦http://192.168.111.222/hzp/sub.asp	URL GET parameters	√	√	√
Level 1	⑧http://192.168.111.222/hzp/login.asp	LOGIN FORM	√	√	
	⑨http://192.168.111.222/hzp/comment.asp	URL GET parameters	√	√	
	⑩http://192.168.111.222/bbs/admin/admincheck.asp	LOGIN FORM	√		
	⑪ http://192.168.111.222/hzp/vpro.asp	URL GET parameters	√		

TABLE VIII. TESTING TIME OF EACH METHOD

	JSP (min)	ASP (hour)	Size of test case
Tool A	39	4.3	45
Tool B	11	2.3	32
NKSI scan	32	3.6	103

VI. SUMMARY

We propose a model based penetration test method for the SQL injection vulnerability in web applications. Experiment shows that compared with randomly enumerated test case, the test case generated by our formal method can detect more SQL injection vulnerabilities hidden behind the inadequate blacklist defense, and thus reduce the false negative of penetration test.

ACKNOWLEDGMENT

The research work here is sponsored by the Tianjin Science and Technology Committee under contract 08ZCKFGX01100 and 12JCZDJC20800.

REFERENCES

[1] S. Kals, E. Krida, C. Kruegel, N. Jovanovic, "SecuBat: A Web Vulnerability Scanner," Proc. 15th International Conference on World Wide Web, ACM, 2006, pp. 247-256

[2] David Byers, Nahid Shahmehri, "Unified modeling of attacks, vulnerabilities and security activities," Proc. 2010 ICSE Workshop on Software Engineering for Secure Systems, IEEE, 2010, pp. 36-42

[3] Jason Bau, Elie Bursztein,Divij Gupta,John Mitchel, "State of the Art: Automated Black-Box Web Application Vulnerability Testing," 2010 IEEE Symposium on Security and Privacy. IEEE, 2010, pp. 332 – 345

[4] William G.J. Halfond, Shauvik Roy Choudhary, Alessandro Orso, "Improving Penetration test Through Static and Dynamic Analysis," Software Testing Verification and Reliability, 2011,21(3):195-214

[5] Jie Wang, Raphael C.-W. Phan, John N. Whitley, David J. Parish, "Augmented Attack Tree Modeling of SQL Injection Attacks," Proc. 2nd IEEE International Conference on Information Management and Engineering, IEEE, 2010, pp. 182-18

[6] Aaron Marback, Hyunsook Do, Ke He, Samuel Kondamarr, Dianxiang Xu, "Security Test Generation using Threat Trees," ICSE Workshop on Automation of Software Test (AST '09), 2009, 62-69

[7] Nuno Antunes, Nuno Laranjeiro, Marco Vieira, Henrique Madeira, "Effective Detection of SQL/XPath Injection Vulnerabilities in Web Services," Proc. IEEE International Conference on Services Computing, IEEE, 2009, pp. 260-267

[8] Yao-Wen Huang ,Chung-Hung Tsai,Tsung-Po Lin, Shih-Kun Huang, D.T. Lee, Sy-Yen Kuo, "A testing framework for Web application security assessment," Computer Networks, vol 48, 2005, pp. 739-761

[9] Adam Kiezun, Philip J. Guo, Karthick Jayaraman, Michael D. Ernst, "Automatic Creation of SQL Injection and Cross-Site Scripting Attacks," Proc. International Conference on Software Engineering, IEEE, 2009, pp. 199-209

[10] Elizabeth Fong, Romain Gaucher, Vadim Okun, Paul E.Black, Eric Dalci, "Building a test suite for web application scanners," Proc. Annual Hawaii International Conference on System Sciences, IEEE, 2008: 4439178

[11] Jiao Antunes, Nuno Neves, Migue Correia, Paulo Verissimo, Rui Neves, "Vulnerability Discovery with Attack Injection," IEEE Transactions on Software Engineering, vol. 36, 2010, 357-369