Journal of Theoretical and Applied Information Technology
15th September 2018. Vol.96. No 17

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

AN OPTIMIZED ATTACK TREE MODEL FOR SECURITY TEST CASE PLANNING AND GENERATION

***HABEEB OMOTUNDE[1], ROSZIATI IBRAHIM[1] MARYAM AHMED[2]**

[1]Department of Software Engineering, FSKTM, UTHM

Batu Pahat, Malaysia

hi130033@siswa.uthm.edu.my, rosziati@uthm.edu.my

[2]College of Computer Science and Information Technology, IAU

Dammam, KSA

mtahmed@iau.edu.sa

ABSTRACT

Securing software assets via efficient test case management is an important task in order to realize business goals. Given the huge risks web applications face due to incessant cyberattacks, a proactive risk strategy such as threat modeling is adopted. It involves the use of attack trees for identifying software vulnerabilities at the earliest phase of software development which is critical to successfully protect these applications. Although, many researches have been dedicated to security testing with attack tree models, test case redundancy using this threat modeling technique has been a major issue faced leading to poor test coverage and expensive security testing exercises. This paper presents an attack tree modeling algorithm for deriving a minimal set of effective attack vectors required to test a web application for SQL injection vulnerabilities. By leveraging on the optimized attack tree algorithm used in this research work, the threat model produces efficient test plans from which adequate test cases are derived to ensure a secured web application is designed, implemented and deployed. The experimental result shows an average optimization rate of 41.67% from which 7 test plans and 13 security test cases were designed to mitigate all SQL injection vulnerabilities in the web application under test. A 100% security risk intervention of the web application was achieved with respect to preventing SQL injection attacks after applying all security recommendations from test case execution report.

Keywords: *Security Testing, SQL injection, Attack trees, Threat Modeling, MOTH.*

1.0 INTRODUCTION

The overwhelming dependence of organizations and individuals on real-time and fault-tolerant web applications demand that service providers adopt critical risk management approaches to ensure these software assets not only work correctly and safely but also guarantee that they do not execute tasks they are not designed for [1]. Although, the huge demand for these applications have triggered a rapid evolution of web technologies with huge gains, unfortunately content developers focus more on functionality while treating security as a system feature rather than an intrinsic property the application must possess throughout its life cycle [2]. Evidently, this is responsible for the wide adoption of reactive software security measures as a remedy for inexistent built-in security. Symantec's Website and Internet Security Threat Reports [3, 4] between 2014 and 2017 continues to reveal huge data breaches due to vulnerabilities in web applications. A 36.4% rise in unauthorized access to private data between 2014 and 2016 was recorded among other attacks to IoT and mobile devices, cloud computing infrastructure and in particular instances, the sabotage of nation states to mention a few. This was corroborated by WhiteHat security report [5] in 2016 with organizations such as IT, Education, Retail and Health care having the most unresolved vulnerabilities. Given this premise, it is important to review the way these application are being developed with more emphasis on preventing further rise in web application attacks by resolving

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

these vulnerabilities in the earliest phases of the Software Development Life-Cycle (SDLC) using a proactive risk-based approach called threat modeling.

Many researchers have adopted the use of attack trees (a threat modeling technique) for providing security specification but the challenge remains optimizing the attack paths derived from these trees [6, 7] in order to reduce redundancy in the derived test cases. A test case is said to be redundant if the software requirements it exercises are also exercised by another test case. This means given any two test-cases *TC1* and *TC2* such that any fault that can be detected by *TC1* is also identified by *TC2*, then these identical test cases are redundant. This redundancy adds to the cost of software testing, maintenance, processing time as well as the size of the test suite making the testing process tedious and uneconomical [8]. Though many methods have been proposed to extract an optimal subset of test cases from a test-suite [9-11] however while these methods do reduce the number of test-cases in the target test-suite, experiments have shown a drastic drop in the overall quality of the test suite while attempting to reduce the test case redundancy [12]. While the security requirement being tested might be exercised by the remaining test cases after reduction, the test-suite's capability to detect software vulnerabilities is significantly reduced. This might prove really hazardous and costly if safety or mission critical systems are affected. Therefore, while threat modeling with attack trees, the attack path optimization process must improve the quality of test cases hence reducing redundancy and achieving high coverage. This will aid the elicitation of effective security requirements needed to fortify the security of the web application. It is important that the attack tree design effort takes advantage of existing graphical or textual design data from repositories such as SVRS or CAPEC [13, 14], if available, and adapt to the requirement of the current case study. This research work intends to answer two main research questions as follows:

- RQ1: How effective is Threat Modeling in reducing the redundancy of test cases?
- RQ2: Does reduction in test case redundancy improve software quality from a security standpoint?

In this paper, the proposed methodology attempts to answer these research questions using a novel attack tree optimization algorithm to detect every attack path liable to exploit a database-driven web application leading to a successful SQL injection attack. With these paths, test plans and security test cases can be designed. The mitigation strategies to avoid these vulnerabilities are derived from test case execution reports.

A significant contribution of this paper is the optimization of attack paths which contributes to an improved SQL injection vulnerability detection. The algorithm removes the redundant attack vectors before test cases are planned and generated. Secondly, this paper demonstrates how effective attack trees are in deriving test cases contrary to positions held in [7]. Being an informal threat model, it facilitates communication and inclusiveness among all stakeholders while deliberating about the security of the application under test (AUT). This will immensely help the development and security team to provide the best set of security requirements which will guide the design, implementation and deployment of a secure web application.

The rest of this paper is organized as follows. Section 2 discusses the background of the study and the related works. In section 3, the methodology is presented along with the attack tree optimization algorithm used in deriving optimized attack paths for minimal and effective test plans and security test cases. A case study is presented in section 4 along with the experimental results to verify improvement in security of the web application under test. Discussion of result is presented in section 5 while section 6 details the study's significance. Section 7 concludes the paper with a brief mention of the future work.

2.0 BACKGROUND OF STUDY AND RELATED WORKS

As web application vulnerabilities continues to increase, previous studies carried out have shown that it is very rare to have unauthorized access to half a billion records without compromising the database in question via SQL injection attacks [15-17]. This has been a major influence in selecting SQL Injection as the main attack vector to be studied in this research paper with a view to mitigating its threat using the proposed threat model. By focusing on the prominent SQL injection types namely: Tautology, Union Based and Boolean Based Blind SQL injection from which other types could be derived [18], we threat model an application named the BodgeIt Store downloaded from GitHub [19] as our case study. After this exercise we have 3 SQL injection attack trees on which the algorithm is executed. Attack trees have the ability to

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

systematically break down high-level threats into a detailed set of steps to execute a potential attack. They are therefore useful for identifying and linking low-level threats to security issues that are applicable to many types of systems.

Many researchers have adopted the use of attack trees in resolving many software security issues. Earlier in 2007, Edge et al [20] proposed attack and protection trees for the purpose of analyzing online banking system security issues. This method attempts to increase the cost of attacks so as to discourage attackers from exploiting the system while on the other hand reduces the cost of protection as a means to encourage the implementation of security measures to protect the bank's customers. Using metrics such as the probability of a successful attack and the cost of executing such attack, the impact of a successful attack is quantified as the risk value of every attack path extracted from the trees. This makes it easier for businesses to make the best decision on where to invest their security budget.

A variant of the attack and protection trees is the attack-defense tree proposed by Kordy et al as a method to analyze complex security and privacy problems [21]. This technique was applied in managing complex attack and defense mechanism required by web applications to prevent attacks as the application grows in size. Unlike attack and protection trees, this tree model presents the activities of both attacker and defenders on the same tree thereby making the model complex but comprehensive using a newly developed formal model. It allows the system's security needs to be understood as it evolves over time from the perspective of both pre-existing defensive measures and newer attacks [21]. By exposing the interaction between the opposing tree structures in one single model, countermeasures against impending threats can be elicited. Due to the complexity of the model, it was difficult to implement this idea without a tool built for this purpose. This was a major limitation in adopting this idea. Attack and defense trees, on the other hand, are compact and subject to continuous iterations while the security metrics are determined for the purpose of deterring future attacks.

Generally, researches in the field of graphical security modeling can be subdivided into 2 namely: Unification and Specification [22]. The first approach, unification, focuses on developing a set of methodologies that unifies all existing approaches for improving the security posture of software assets. These models usually have sound formal foundations and are extensively studied from a theoretical point of view. They are augmented with formal semantics and a general mathematical framework for quantitative analysis [23, 24]. The attack–defense trees and Bayesian attack graphs are good examples of such models in this category.

On the other hand, the specification category develops methodologies to solve security problems peculiar to a domain just as intended in this research where attack tree models are adopted for planning, generating and executing security test cases. Other domains in this category could include:

- intrusion detection using attack and protection trees [20],
- secure software development through the use of security activity graphs, finite state machines [25] and
- security requirements engineering via misuse cases [26].

Formalisms developed within this trend are often based on empirical studies and practical needs. Although, graphical security modeling in the specification category continues to receive wide attention and adoption, of particular interest are the works of Wang et al [27], Marback et al [6, 28] and Xu et al [7] with respect to test case generation from security models. Xu et al [7] and Marback et al [6] used petri nets and attack trees respectively in deriving attack paths while Wang et al [27] used sequence diagram in modeling threat traces which was also similar to attack paths. These attack paths are central to the core of these research efforts as the software security testing exercise depended on security test cases generated from the paths. However, few problems were identified in their methodologies.

Firstly was the lack of optimization of attack paths which could lead to the creation of redundant security test cases. This invariably leads to the need for huge computation resources in the event that the threat model is large. Secondly, the assumptions on which Marback [6] designed the attack trees do not mimic real-world scenarios. In this approach, the researchers assumed that the siblings of a branch must have the same operators and they must be ordered. These assumptions are neither necessary nor valid as attackers do not conform to this kind of conditions. The attacker's approach involves a mixture of every possible attack steps with a goal of reaching the root of the attack tree. This is a reality that must be captured by an effective system as evident in the attack trees used in this research paper. Finally, Wang used the

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

attack paths derived at runtime to monitor for suspicious activities after the web application has been deployed. This could pose significant overhead and also expose the application to denial of service attacks if system resources get used up. Also, attack paths not captured during design cannot be patched easily without substantial changes to the application's source code which would defeat the purpose of using threat modeling in the first place. The threat model is expected to guide the elicitation of security requirements, secure design and implementation of the AUT.

3.0 THREAT MODELING PROCESS

Threat modeling is a software security practice utilized by software developers, architects and security experts at the design phase of software development to document the key assets found in a software application and intentionally expose risks to those assets in a thorough and disciplined manner. The goal of a threat modeling exercise is to discover hidden security risks or software vulnerabilities regarded as "entry points" [29] that may elude the application developers using threat modeling procedures. This information can then be used to develop risk management and mitigation strategies and provide a roadmap for proactive security plans [30]. In order to design attack tree models for the purpose of securing web applications from SQL injection attacks, it is important to have an idea of what SQL injection is.

3.1. SQL Injection

SQL injection involves the insertion of specially crafted string input or encoded SQL query into the web forms or HTTP requests sent to web servers in order to alter the logic of the original SQL query for malicious intent such as bypassing authentication, gaining unauthorized access to confidential data etc. This attack type exploits input validation flaws, use of dynamically generated SQL statement which mixes SQL code and user data together, violation of the principle of least privilege and poor exception handling [15]. The attack comes in many forms which can be summarized into 3 types namely: Tautology, Union Based and Boolean Based Blind SQL injection attacks.

3.1.1. Tautology SQL injection attack

This attack vector exploits the vulnerability in the conditional clause of SQL statements by injecting code construct which will always evaluate to true [31]. In most cases, an in-line comment is appended to the code so as to ensure the remaining portion of the SQL statement is ignored by the database engine at runtime transforming the statement into something of this nature:

select * from tablename where username= 'username' or 1=1 -- and password=''

This is commonly adopted for the purpose of bypassing authentication, verifying web application vulnerability and identifying injectable form variables.

3.1.2. Union-based SQL injection attack

Union-Based SQL injection attack is a form of Server-Side Error-Based SQLIA. It relies on the deficiencies of input validation flaws and improper error handling in the DBMS (Database Management System) which tries to return more query outputs using the UNION operator [32]. As mentioned in the previous section, this comes after an attacker has successfully established that some form parameters are vulnerable to SQLIA. By appending the UNION operator with SELECT queries to the vulnerable parameter, the attacker can fetch data from any part of the back-end database far from the desired usecase. This is mainly employed in data extraction asides from bypassing authentication. A sample input including a union query takes this form:

' or 1=1 union select 1,2,3,4,5,database() -- '

produces this query:

select * from tablename where uname= 'username' union select 1,2,3,4,5,database() from tablename - -' and password=''

3.1.3. Boolean-based blind SQL injection attack

Boolean-Based Blind SQL injection is a type of SQLIA which is resorted to when the web application is configured to report generic error messages. This is one of the best practices expected on any organization's production system. This configuration hides many useful information thereby making exploitation difficult however not impossible [31]. Attackers tend to ask the database true or false questions and carefully monitor the variation in application response as a means of stealing information from the database [33]. In the event that the malicious statement parsed by the database engine is true, the web application will continue to function properly but if false, there's a noticeable deviation from a normal behaviour. In some cases, the server response causes the web page to appear deformed as the HTML (Hypertext Mark-up Language) response is rendered poorly. This is mostly because such response was neither

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

planned for nor styled for the sake of presentation at the client side. This is a viable indication of vulnerability to SQLIAs and an avenue for extracting meta-data about the database. Though difficult, however, by carefully noting the subtle behaviour of the site in response to different inputs and function calls, the attacker can correctly infer the vulnerable parameters and useful information from the database. A sample input including a boolean-based blind query takes this form:

' username' and substring(@@version,1,1) = 4 - - '

produces this query:

select * from tablename where uname= 'username' and substring(@@version,1,1) = 4 - - '' and pass=''

In the event that the web application loads normally, this is an indication that the version of the back-end database starts with 4 otherwise the page throws an error. This indicates the form variable in vulnerable to SQL injection and liable to be exploited.

3.2. Designing the SQL Injection Attack Trees

Attack trees are informal in nature hence difficult to define accurately in formal terms. However, being a tree data structure, it bears huge similarities with a directed acyclic graph[22] which make it easier to formally define it and understand its properties as seen in Definition 1.

Definition1:

Let Attack Tree T represent a finite set of steps an attacker would perform to exploit a web application W. Then T is a 4-tuple digraph defined as

$$(n_0, B, L, E)$$

Where

1. Vertices $V = (n_0\ U\ B\ U\ L)$ is the set of all Branches B, leaves L and the root node, n_0.
2. The root node, n_0 represents the main goal of the attack tree.
3. The set of leaves $L = \{L_1, \dots, L_N\}$ represents attacks which could be true or false if the attack was successful or not.
4. The set of branches $B = \{B_1, \dots, B_M\}$ represents special nodes with logical functions $\wedge and \vee$ i.e. and / or functions. The function $\wedge$ evaluates to true if all of its children evaluate to true and function $\vee$ evaluates to true, if any of its children evaluate to true. A branch connects leaves, L together and it could be a sub-goal of the root node, n_0
5. E = $\{(i, j): i, j \in V\}$ is the set of all directed edges connecting all adjacent nodes.

A successful attack only occurs if there exist a set of events that create a path through a set of leaves to the root node, n_0

With this definition, the three attack trees are designed as seen in Figures 1, 2 and 3 with respect to the web application being tested. Each attack tree is represented in a table called the Tree Implementation Matrix (TIM). Table 1 represents a TIM which was derived by extracting all information from the attack tree model of Figure 1.

The TIM captures all properties and characteristics of the SQL injection attack tree showing actions executed by the adversary to exploit the SQL injection vulnerability.

The information in this table is key to detecting unique attack paths using the analyze tree model algorithm. These unique paths represent route from the leaf nodes to the root node which serves as a guide to a successful security testing process. Given the research scope, each SQL injection type studied will have a TIM.

In the TIM of Table 1, the Tautology attack tree is described. Each node is identified based on its type, trust level (malicious or general) and capacity to invoke SQL statement. Mapping the description from Table 1 to the attack tree, some leaves will implicitly have a true value because they are completely harmless and do not trigger or invoke any SQL commands. These are general activities accessible to every user who can access the system. Therefore, our concern is sand-boxing only the malicious activities whose node type is a leaf that invokes SQL command and monitoring them to ensure that they do not evaluate to true.

Journal of Theoretical and Applied Information Technology
15th September 2018. Vol.96. No 17

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

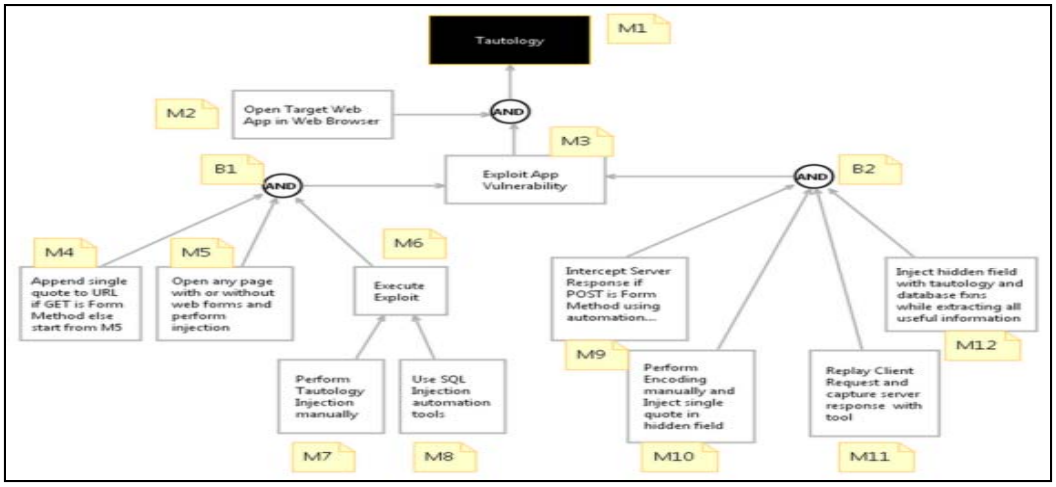


Figure 1: Tautology SQLIA Attack Tree

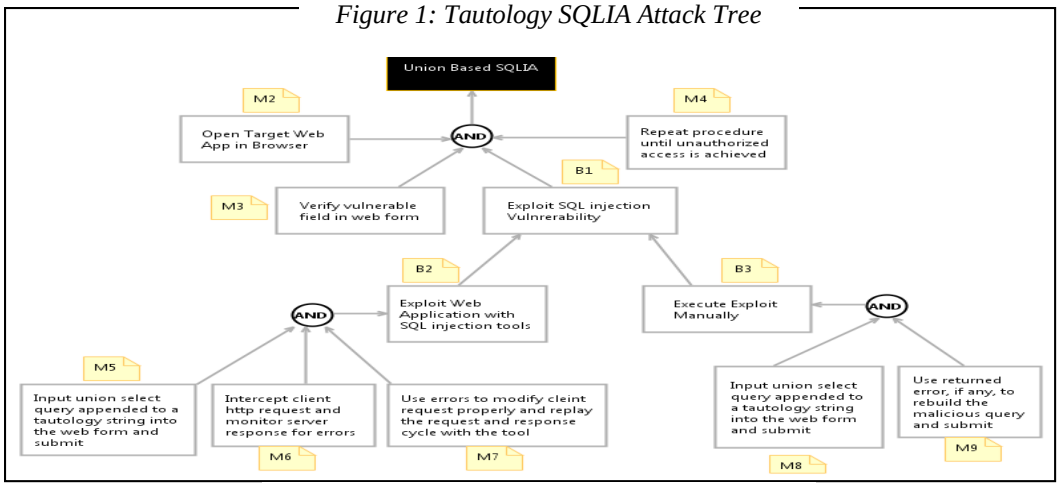
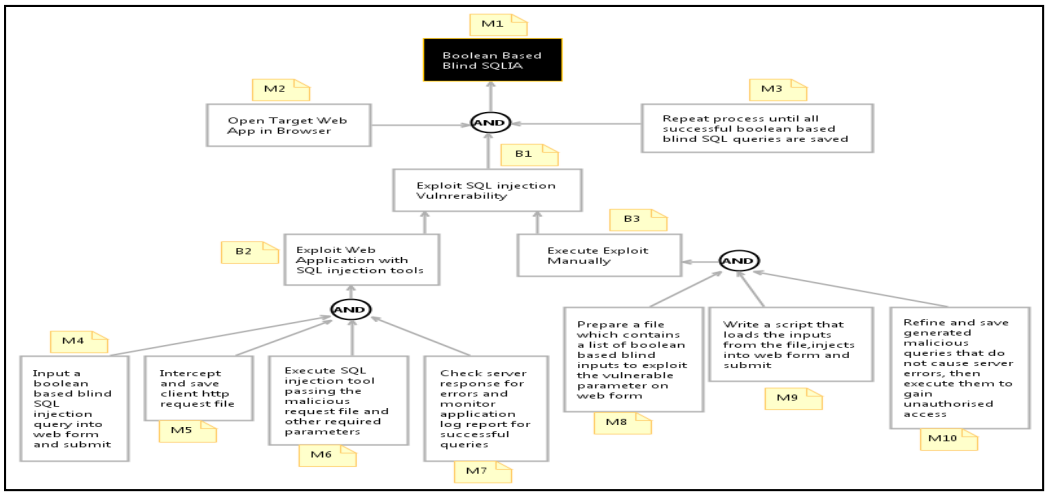


Figure 2: Union-Based SQLIA Attack Tree



ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Table 1: Tree Implementation Matrix (TIM) for Tautology Attack Tree Model

Label	Node Type	Action	Activity	Invoke SQL
M1	root, n_0	Tautology	malicious	Yes
M2	leaf	Open Target Web App in Web Browser	general	No
M3	branch	Exploit App	malicious	No
M4	leaf	Append single quote to URL if GET is Form Method else start from M5	malicious	No
M5	leaf	Open any page with or without web forms and perform injection	malicious	No
M6	branch	Execute Exploit	malicious	No
M7	leaf	Perform Tautology Injection manually	malicious	Yes
M8	leaf	Use SQL Injection automation tools	malicious	Yes
M9	leaf	Intercept Server Response if POST is Form Method using automation tools	malicious	No
M10	leaf	Perform Encoding manually and Inject single quote in hidden field	malicious	No
M11	leaf	Replay Client Request and capture server response with tool	malicious	Yes
M12	leaf	Inject hidden field with tautology and database functions while extracting all useful information	malicious	Yes

3.3 Attack Path Detection and Optimization algorithm

Algorithm 1 showcases the analyze tree model algorithm for traversing the attack trees in Figures 1 to 3. The algorithm crawls the attack tree recursively using the depth first search technique in order to identify all possible attack paths. As stated earlier in Table 1 (Annex1), leaves marked general are not considered harmful in the attack paths except the malicious leaf nodes that invoke SQL commands. This helps in reducing the time to compute all paths to the root node and guarantees efficient test cases are designed using the optimized attack paths.

Algorithm 1: Attack Path Detection via MOTH's Depth First Search Algorithm

Input: AT: [Attack tree composed of Nodes with 'AND'/'OR' operators]
Output: Attack Path: [A set of Optimized Attack Paths to the Root Node,n_0]

```
1: Initialize AT: Digraph AT ⇐ new Digraph()
2: Create Nodes: Nodes[] ⇐ new Node(name, flag, fxn, nodeType)
3: Add Nodes to Machine: Machine[]⇐ Nodes[]
4: Add Edges to Nodes: AT.addEdges(Nodei, Nodej)
5: Set Start Machine: M1 = n0
6: Set End Machine: End[] ⇐ getInvokeSQLMachines(Machine[])
7: Initialize Visited Node: Visited[] ⇐ n0
8: Start Search
9: while End[] ≠ 0 do
10:        Nodes[] = AT.adjacentNodes[Visited.getLast()]
11:        MDFS( AT, Visited )
12:        for each node ∈ Nodes[] do
13:            if Visited.contains(node) then
14:                continue
15:            end if
16:            if node.equals(End[i]) then
17:                Visited.add(node)
18:                print(Attack Path)
```

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

```
19:             end if
20:         end for
21:         for each node ∈ Nodes[] do
22:             if Visited.contains(node) OR
node.equals(End[i]) then
23:                 Visited.addLast(node)
24:                 MDFS( AT, Visited )
25:                 Visited.removeLast()
26:             end if
27:         end for
28:  AttackPathOptimization.optimizer(Attack
Path)
29: end while
```

3.4 Attack Path Optimization for SQL Injection Attack Trees.

This algorithm was designed as a feature in a Hybrid Threat Modeling tool called MOTH which was implemented in Eclipse using Java programming language.

A summary of the results obtained from the 3 attack trees are listed below including the optimized attack paths.

3.4.1 Tautology SQLIA attack paths

1. Attack Path, AP_{t1} from M1←M7 : M1.(M2.M3).B1.(M4.M5.M6).M7
2. Attack Path, AP_{t2} from M1← M8 : M1.(M2.M3).B1.(M4.M5.M6).M8
3. Attack Path, AP_{t3} from M1← M11 : M1.(M2.M3).B2.(M9.M10.M11.M12)
4. Attack Path, AP_{t4} from M1← M12 : M1.(M2.M3).B2.(M9.M10.M11.M12)

Therefore,

$$AP_t = \{AP_{t1}, AP_{t2}, AP_{t3}, AP_{t4}\}$$

is optimized to

$$AP'_t = \{AP_{t1}, AP_{t2}, AP_{t3}\}$$

3.4.2 Union-based SQLIA attack paths

1. Attack Path, AP_{u1} from M1←M5 : M1:(M2:M3:B1:M4):B2:(M5:M6:M7)
2. Attack Path, AP_{u2} from M1← M7 : M1:(M2:M3:B1:M4):B2:(M5:M6:M7)
3. Attack Path, AP_{u3} from M1← M8 : M1:(M2:M3:B1:M4):B3:(M8:M9)
4. Attack Path, AP_{u4} from M1← M9 : M1:(M2:M3:B1:M4):B3:(M8:M9)

Therefore,

$$AP_u = \{AP_{u1}, AP_{u2}, AP_{u3}, AP_{u4}\}$$

is optimized to

$$AP'_u = \{AP_{u1}, AP_{u3}\}$$

3.4.3. Boolean-Based Blind SQLIA Attack Paths

1. Attack Path, AP_{b1} from M1←M4 : M1:(M2:M3:B1):B2:(M4:M5:M6:M7)
2. Attack Path, AP_{b2} from M1← M6 : M1:(M2:M3:B1):B2:(M4:M5:M6:M7)
3. Attack Path, AP_{b3} from M1← M9 : M1:(M2:M3:B1):B3:(M8:M9:M10)
4. Attack Path, AP_{b4} from M1← M10 : M1:(M2:M3:B1):B3:(M8:M9:M10)

Therefore,

$$AP_b = \{AP_{b1}, AP_{b2}, AP_{b3}, AP_{b4}\}$$

is optimized to

$$AP'_b = \{AP_{b1}, AP_{b3}\}$$

4.0 CASE STUDY

The web application being considered in this research is The Bodgeit Store downloaded from GitHub [19]. It was designed by security experts from OWASP for benchmarking security tools. Bodgeit store is a vulnerable web application with many security bugs especially SQL injection amongst others however focus is given to SQL injection being the scope of the studies. This vulnerability affects many of the services provided by the application under test (AUT) such as login, registration, search, account management and basket functionalities as seen in Table 2 which shows a total number of 15 vulnerable SQL statements in the web application. An overview of the web application is summarized in Table 3 while its services are shown in its use case diagram in Figure 4.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Table 2: Distribution of SQL injection Vulnerabilities across the AUT

	Login.jsp	Basket.jsp	Register.jsp	Advance.jsp	Password.jsp	Total
Select	1	3	1	1	1	6
Update	2	1	2	0	1	6
Insert	0	2	1	0	0	3
Total	3	6	4	1	1	15

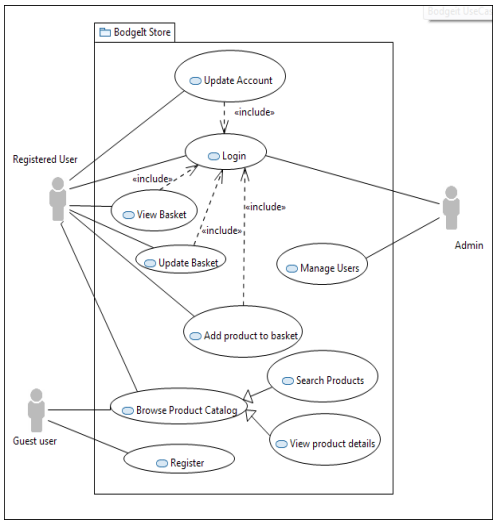


Figure 4: BodgeIT Store UseCase Diagram

4.1. Security Test Plan Derivation

In order to execute security tests on the AUT, adequate test plans (TP) must be prepared. These test plans are designed using optimized attack paths, AP, derived in section 3.4. Table 4 shows a mapping of the attack paths to vulnerable assets in order to create effective test plans from which security test cases will be designed.

Table 3: Overview of bodgeit store web application modules

#	Module(.jsp)	#LOC	Use case
1	about	18	about app
2	admin	92	manage users
3	advanced	193	search product catalog
4	contact	134	contact us
5	footer	9	Page Footer
6	header	117	Page header
7	home	74	Home page
8	init	280	web application initialization
9	login	149	Authentication
10	logout	11	Authentication
11	password	106	manage account
12	product	137	view product catalog
13	register	161	Add new user
14	score	78	Rate achievements
15	search	102	search product catalogue

Table 4: Mapping Attack Paths to Assets vulnerable to SQL Injection

	Login	Basket	Register	Advanced	Password
Tautology SQLIA Test Suite	TP1: AP_{t1}, AP_{t3}	TP4: AP_{t2}, AP_{t3}		TP6: AP_{t3}	TP7: AP_{t1}
Union Based SQLIA Test Suite	TP2: AP_{u1}, AP_{u3}		TP5: AP_{u3}		
Union Based Blind SQLIA Test Suite	TP3: AP_{b1}				

4.2. Security Test Case Generation from Test Plan

A total of 13 security test cases were generated from the 7 test plans in Table 4. After executing these test cases, some security recommendations were specified. These recommendations were implemented and the security improvement of the AUT was measured again to determine the effectiveness of the test cases derived from the threat model.

Table 5 (Annex 1) shows the test case report which contains the test cases, status of the test case execution before security recommendations and proposed security requirement specifications for resolving the SQL injection vulnerabilities detected in the AUT.

Journal of Theoretical and Applied Information Technology
15th September 2018. Vol.96. No 17

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

5. EXPERIMENTAL RESULTS AND DISCUSSION

The experiment was conducted on a Core i5 PC running 64bits Windows 7 OS with 16GB Ram. Firstly, the setup includes deploying the web application (bodgeit.war) to apache tomcat server before simulating SQL injection attacks by executing test cases TC1-TC13 with selenium and other SQL injection attack tools such as SQL Map and Tamper Data. As seen in Figure 5, the 3 SQL injection attacks trees were designed using a Hybrid threat modeling tool called MOTH, which was built during the course of this research. MOTH offers other features apart from attack path detection and optimization, however, this is not the subject of discussion.

Table 6: Attack Path Optimization of Attack Trees

Attack Tree (AT) Type	Before Path Optimization	After Path Optimization	% Optimization
Tautology AT	4	3	25
Union Based AT	4	2	50
Boolean-Based Blind AT	4	2	50

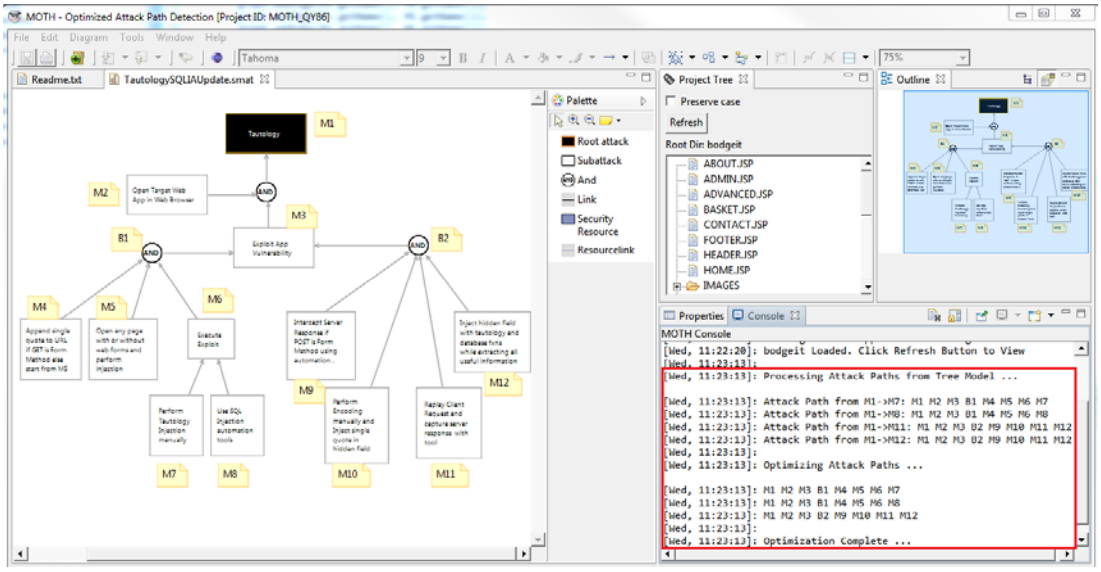


Figure 5: Optimized Attack Path Detection using Analyze Tree Model Algorithm

Table 6 shows the rate of optimization while the value was quantified using the stated formula. An average optimization rate of 41.6% was achieved. This answers the first research question (RQ1) which seeks to determine the effectiveness of threat modeling approach in reducing test case redundancy. It is worthy of mention that the new test suite is not only effective, the overall quality and coverage of the test suite remains unchanged.

$$= \frac{\sum_{AP} Before\ Opt. - \sum_{AP} After\ Opt.}{\sum_{AP} Before\ Opt.} \times 100\%$$

$$Optimization_{Avg} = \frac{12 - 7}{12} \times 100\% = 41.67\%$$

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

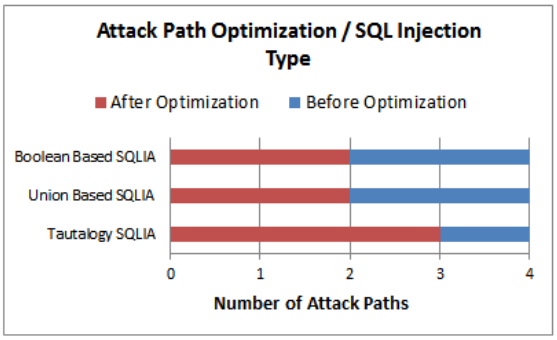


Figure 6: Attack Path Optimization

Furthermore, after all security recommendations of Table 7 (Annex 2) were implemented, the test cases were executed again on the vulnerable assets in order to quantify security improvement. This table also summarizes the result before and after security recommendations were applied. After this process, security improvement of the AUT is quantified using the formula below,

where,

$$IMPV_{security} = \frac{No.\ of\ vulnerabilities\ fixed}{No.\ of\ detected\ vulnerabilities} \times 100\%$$

$$IMPV_{security} = \frac{15}{15} \times 100\% = 100\%$$

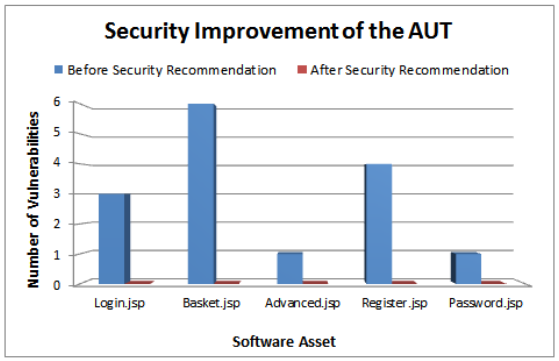


Figure 7: Improvement in security after Applying Security Recommendations

This evidently answers the second research question (RQ2) showing that the reduction in test case redundancy actually improves software security. The chart in Figure 7 show that all SQL injection vulnerabilities found in each software module were completely resolved hence a 100% improvement in security. This is make threat modeling a viable option for eliciting security requirements for any application of any size as threat models provide every attack paths through which the application can be exploited. With this information at hand at the early stages of requirement gathering and design in the SDLC, both business and the development team are better equipped with information on what risks to mitigate, ignore or transfer. Furthermore the test plan serves as a viable document to measure compliance with security policies set by the software quality assurance team. With this guide, content developers and security teams know exactly what security requirement to implement and where to implement them as the web application evolves.

6. SIGNIFICANCE OF STUDY

Defining and verifying security requirements of web applications are an important process in the earlier phase of the SDLC. With the new concept presented in this paper, it is feasible to test those requirements with a minimal test suite at an affordable cost while managing scarce resources efficiently. Given the ease of setting up this test environment, several iterations of attack tree models can be prepared to exploit found vulnerabilities, design the test suite and finally improve the security of the AUT.

Given this premise, the significance of the analyze tree model algorithm over Marback's breadth first search technique are its capacity to process a mixture of operations between a group of siblings connected by a branch. The algorithm efficiently processes unordered siblings irrespective of their branch operators thereby representing real world conditions. In addition to this, it further truncates redundant test cases without sacrificing the quality of the test suite which is absent in previous threat modeling research efforts targeted at deriving test cases from security models.

Furthermore, while designing the attack trees, all leaf nodes that represent potential SQL injection attacks were identified using the tree implementation matrix. This makes attack path derivation faster as the focus on security testing is separated from functional testing. It is worth mentioning that [6, 7, 27] execute security test via

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

tainting and security mutants. These mutants are copies of the tested programs deliberately corrupted with vulnerabilities by the researchers which are subject to bias.

It is worthy of mention that the contribution of this research paper is in two folds. Firstly, our approach can generate test plans from which executable security tests cases can be derived from attack tree threat models dedicated to SQL injection attacks. Since these test plans are dependent on the size of the attack trees, optimizing the attack paths for the purpose of minimizing test case redundancy is a plus and a major task achieved in this paper. Secondly, this research work was able to demonstrate the capacity of threat modeling with attack trees as a viable and feasible method of improving software security contrary to previous assertion in other research efforts.

7. CONCLUSION AND FUTURE WORK

In this paper, we have shown the importance of threat modeling web applications using an optimized attack tree algorithm to derive minimal test plans. This is necessary to generate effective test cases in order to elicit adequate security requirements needed to prevent SQL injection attacks. Although, this technique has proved effective, it is however important to extend the attack vector considered as the threat scape is very large. A major strength in the use of attack trees is the ease of reuse of these models. They can be easily adapted with minimal effort while considering other attack vectors.

Although the results are promising, this study has few limitations. One limitation involves how the threat trees are designed. There's a direct correlation between the quality of the threat models and the technical competence of the individuals developing the attack trees. Furthermore, the scope of SQL injection attacks covered might look narrow though it's very possible to derive the rest of the SQL injection types from the base types treated in this paper.

Furthermore, this testing technique can be extended to other applications such as those running on mobile and IoT devices as they have also been reported to suffer huge attacks in recent threat reports by software security organizations.

REFERENCES

[1] M. Paul. (2014). *Software Security: Being Secure in an Insecure World [White paper]*. Available: https://www.isc2.org/uploadedFiles/(ISC)2_Public_Content/Certification_Programs/CSSLP/CSSLP_WhitePaper_3B.pdf

[2] G. McGraw, *Software security: building security in* vol. 1: Addison-Wesley Professional, 2006.

[3] Symantec. (2014, Web Security Threat Report. *WSTR Volume 19*.

[4] C. Kavitha, C. Gillian, C. Orla, L. Hon, N. Benjamin, O. Brigid, *et al.* (2017, Symantec Internet Security Threat Report. *ISTR Volume 22*.

[5] WhiteHat-Security. (2014, 06/09/2016). *Web Application Security Statictics Report.* Available: https://info.whitehatsec.com/rs/675-YBI-674/images/WH-2016-Stats-Report-FINAL.pdf

[6] A. Marback, H. Do, K. He, S. Kondamarri, and D. Xu, "A threat model-based approach to security testing," *Software-Practice & Experience,* vol. 43, pp. 241-258, Feb 2013.

[7] D. Xu, M. Tu, M. Sanford, L. Thomas, D. Woodraska, and W. Xu, "Automated security test generation with formal threat models," *IEEE Transactions on Dependable and Secure Computing,* vol. 9, pp. 526-540, 2012.

[8] H. Omotunde, R. Ibrahim, M. Ahmed, R. F. Olanrewaju, N. Ibrahim, and H. Shah, "A framework to reduce redundancy in android test suite using refactoring," *Indian Journal of Science and Technology,* vol. 9, 2016 2016.

[9] G. Raj, D. Singh, and I. Tyagi, "Test Case Optimization and Prioritization of Web Service Using Bacteriologic Algorithm," in *Intelligent Computing and Information and Communication*, Singapore, 2018, pp. 731-744.

[10] J. Ahmad and S. Baharom, "Factor Determination in Prioritizing Test Cases for Event Sequences: A Systematic Literature Review," *Journal of Telecommunication, Electronic and Computer Engineering (JTEC),* vol. 10, pp. 119-124, 2018.

[11] S. P. R. Asaithambi and S. Jarzabek, "Towards Test Case Reuse: A Study of Redundancies in Android Platform Test

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Libraries," Berlin, Heidelberg, 2013, pp. 49-64.

[12] G. Fraser and F. Wotawa, "Redundancy based test-suite reduction," in *International Conference on Fundamental Approaches to Software Engineering*, 2007, pp. 291-305.

[13] MITRE. (2017, 24/06/2017). *Common Attack Pattern Enumeration and Classification*. Available: https://capec.mitre.org/index.html

[14] S. LiU and E. Rios, "D2. 2 Initial Modelling Methods and Prototype Modelling Tools," 2008.

[15] OWASP. (2015, 03-06-2015). Testing for SQL Injection. Available: https://www.owasp.org/index.php/Testing_for_SQL_Injection

[16] L. K. Shar and H. B. K. Tan, "Predicting SQL injection and cross site scripting vulnerabilities through mining input sanitization patterns," *Information and Software Technology,* vol. 55, pp. 1767-1780, 10// 2013.

[17] V. Shanmughaneethi and S. Swamynathan, "Detection of SQL Injection Attack in web applications using web services," *IOSR Journal of Computer Engineering (IOSRJCE),* vol. 1, pp. 13-20, 2012.

[18] H. Omotunde and R. Ibrahim, "Mitigating SQL Injection Attacks via Hybrid Threat Modelling," in *Information Science and Security (ICISS), 2015 2nd International Conference on*, 2015, pp. 1-4.

[19] S. Bennetts. (2014). *The BodgeIt Store*. Available: https://github.com/psiinon/bodgeit

[20] K. Edge, R. Raines, M. Grimaila, R. Baldwin, R. Bennington, and C. Reuter, "The use of attack and protection trees to analyze security for an online banking system," in *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, 2007, pp. 144b-144b.

[21] B. Kordy, S. Mauw, S. Radomirović, and P. Schweitzer, "Foundations of Attack–Defense Trees," in *Formal Aspects of Security and Trust*. vol. 6561, P. Degano, S. Etalle, and J. Guttman, Eds., ed: Springer Berlin Heidelberg, 2011, pp. 80-95.

[22] B. Kordy, L. Piètre-Cambacédès, and P. Schweitzer, "DAG-based attack and defense modeling: Don't miss the forest for the attack trees," *Computer science review,* vol. 13, pp. 1-38, 2014.

[23] R. Jhawar, B. Kordy, S. Mauw, S. Radomirović, and R. Trujillo-Rasua, "Attack trees with sequential conjunction," in *IFIP International Information Security Conference*, 2015, pp. 339-353.

[24] N. Poolsappasit, R. Dewri, and I. Ray, "Dynamic security risk management using bayesian attack graphs," *IEEE Transactions on Dependable and Secure Computing,* vol. 9, pp. 61-74, 2012.

[25] S. Chen, Z. Kalbarczyk, J. Xu, and R. K. Iyer, "A Data-Driven Finite State Machine Model for Analyzing Security Vulnerabilities," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2003, pp. 605-614.

[26] G. Sindre and A. L. Opdahl, "Eliciting security requirements with misuse cases," *Requirements Engineering,* vol. 10, pp. 34-44, Jan 2005.

[27] L. Wang, E. Wong, and D. Xu, "A Threat Model Driven Approach for Security Testing," presented at the Proceedings of the Third International Workshop on Software Engineering for Secure Systems, 2007.

[28] A. Marback, H. Do, K. He, S. Kondamarri, and D. Xu, "Security test generation using threat trees," in *Automation of Software Test, 2009. AST'09. ICSE Workshop on*, 2009, pp. 62-69.

[29] A. Shostack, *Threat modeling: Designing for security*: John Wiley & Sons, 2014.

[30] SecurityInnovation. (2011). *Threat Modelling for Secure Embedded Software [White paper]*. Available: http://web.securityinnovation.com/threat-modeling-embedded/

[31] W. Halfond, J. Viegas, and A. Orso, "A classification of SQL-injection attacks and countermeasures," in *Proceedings of the IEEE International Symposium on Secure Software Engineering*, 2006, pp. 65-81.

[32] R. Dharam and S. G. Shiva, "Runtime Monitors to Detect and Prevent Union Query based SQL Injection Attacks," *Proceedings of the 2013 10th International Conference on Information Technology: New Generations,* pp. 357-362, 2013.

[33] R. Chandrashekhar, M. Mardithaya, S. Thilagam, and D. Saha, "SQL Injection Attack Mechanisms and Prevention Techniques," in *Advanced Computing, Networking and Security*. vol. 7135, P. S. Thilagam, A. Pais, K. Chandrasekaran, and N. Balakrishnan, Eds., ed: Springer Berlin Heidelberg, 2012, pp. 524-533.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

ANNEX1 *Table 5: Test Case Report and Security Recommendations*

Test Case ID	Test Case Description	Status	Observations and comment	Security Recommendations for mitigating SQL Injection Vulnerabilities in the AUT
TC1	Login with Invalid username(U) and password(P)	Fail	1. Input validation missing for both username and password fields 2. Length of user input is not restricted 3. SQL statements were generated dynamically via string concatenation.	**Login.jsp** 1. All dynamically generated SQL statements on the login form must be replaced with parameterized queries. 2. Use of string concatenation with parameterized queries must not be allowed as it violates secure coding practices 3. All user input must be validated on the server side as client side validation can be bypassed 4. Ensure strict enforcement of username and password policy to prevent SQL injection attacks by specifying an acceptable format and length that meets system security requirement 5 Exception handling must reveal only generic messages to users 6. Database connection must only be initiated with an account having the necessary privileges to make the application function properly.
TC2	Login with username (U) only	Fail	All observations in TC1 also observed here and login attempts are allowed without providing a password.	
TC3	Login with password (P) only	Fail	All observations in TC1 also observed here and login attempts are allowed without providing a username.	
TC4	Login using a request file and SQL Map	Fail	1. All observations in TC1 also observed here and parameterized queries are not used in writing SQL statements. 2. Database connections are carried out with excessive privilege	
TC5	Access all tables on database using SQL Map	Fail	Same as TC4	
TC6	Login using Union based Injection without password (P)	Fail	Observations from TC1 and TC2 also observed here and DB server does not restrict messages displayed in the event of an exception.	
TC7	Access all user information using a request file and SQL Map	Fail	Same as TC4	
TC8	Access another user's basket by injecting values into cookie using Tamper Data	Fail	1. Dynamic SQL query used to obtain basket ID from cookies to display user basket. 2. User specific data stored in cookies unencrypted and Session management is not enforced in managing unique user data 3. Guest Users are allowed to add items to basket which violates the System's Use Case. 4. Input Data type not enforced on cookies	**Basket.jsp** 7. Enforce verification of cookie value so it matches with that of the currently logged in user. 8. Ensure guest users can not add item to any basket 9. Verify cookie data type before getting user's basket items 10. All SQL queries must be designed using parameterized queries to avoid loss of semantics of the SQL query 11. All form parameters must be verified before passing parameter values to the SQL statements with validation routines implemented on the server
TC9	Corrupt basket Database with a post request file using SQL Map	Fail	1. Attackers exploit basket.jsp via a valid referral link in the http header to corrupt the basket DB pretending to add item to basket. 2. Input validation not enforced on form parameter found in the request file. 3. Dynamically generated SQL queries allow user input to change semantics of the query on the basket.jsp asset	
TC10	Access Database Tables using Union Based SQLIA	Fail	1.Search strings for user not validated for correctness and length. 2. Invalid inputs not properly escaped 3. Database connections established with over-privileged account	**Advanced.jsp** 12. Use regular expression to define a concise and acceptable format of search strings including value of product prices

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

TC11	Register with username (U) containing a malicious statement and a normal password (P)	Fail	1. The registration form lacks input validation routines for verifying inputs against malicious content 2. SQL statements derived from string concatenation are susceptible to SQLIAs. 3. SQL keywords allowed as username	**Register.jsp** 14. Recommendation 10 must be enforced 15. Enforce input validation on the server to ensure malformed inputs are dropped and generic information is shown to users 16. No SQL keyword or database specific operators are accepted as either username or password
TC12	Register with normal username (U) but with password (P) containing a malicious SQL statement	Fail	1. The registration form lacks input validation routines for verifying inputs against malicious content 2. SQL statements derived from string concatenation are susceptible to SQLIAs 3. Proper password policy not set to enforce acceptable password properties	
TC13	Access Database Tables using Union Based SQLIA	Fail	1. Tainted input with malicious queries processed by back-end database. 2. Proper password policy not set to enforce acceptable password properties 3. Use of dynamic SQL statement vulnerable to SQLI vulnerabilities for password update	**Password.jsp** 17. Recommendation 10 must be enforced 18. Recommendation 11 must be enforced.

Annex 2 *Table 7: Comparison of test case execution results before and after application of security requirements*

	Before Applying Security Recommendation			After Applying Security Recommendation		
Test Cases	**Expected Result**	**Actual Result**	**Status**	**Expected Result**	**Actual Result**	**Status**
TC1	Invalid username or password	Login Successful	Fail	Invalid username or password	Invalid username or password	pass
TC2	Invalid username or password	Login Successful	Fail	Invalid username or password	Invalid username or password	pass
TC3	Invalid username or password	Login Successful	Fail	Invalid username or password	Invalid username or password	pass
TC4	Login not Successful	Login Successful	Fail	Login not Successful	Login not Successful	pass
TC5	Table not accessible	table accessible	Fail	Table not accessible	Table not accessible	pass
TC6	Invalid username or password	Login Successful	Fail	Invalid username or password	Invalid username or password	pass
TC7	parameter pass not injectable	parameter pass vulnerable	Fail	parameter pass not injectable	parameter pass not injectable	pass
TC8	Cannot access another users shopping basket	Unauthorized access to other users basket successful	Fail	Cannot access another users shopping basket	Cannot access another users shopping basket	pass
TC9	Basket Database information intact	Basket Database Corrupted	Fail	Basket Database information intact	Basket Database information intact	pass
TC10	No results found	Database Tables revealed	Fail	No results found	No results found	pass
TC11	Invalid username or password	You have successfully registered with The Bodgeit Store	Fail	Invalid username or password	Invalid username or password	pass
TC12	Invalid username or password	You have successfully registered with The Bodgeit Store	Fail	Invalid username or password	Invalid username or password	pass
TC13	Invalid password	Your Password has been changed	Fail	Invalid password	Invalid password	pass