Security Methods Implementation and Quality of Experience (QoE) for Web Applications Performance

Ustijana Rechkoska-Shikoska
University for Information Science and Technology UIST "St. Paul the Apostle"
Ohrid, Macedonia
e-mail: ustijana@gmail.com

Abstract— **Web apps have a big impact in most of our activities nowadays. Unfortunately, they are also a target for illegal actions. When attacking Web apps, a hacker will try several means of compromising the applications, paying special attention to the database driven Web apps. The Structured Query Language (SQL) Injection Attacks (SQLIAs) are one of the most common methods of data theft on Web apps. SQLIA is a hacking technique that attackers use to compromise the database in most of Web apps, by manipulating SQL queries to change their behavior. Concomitantly, the attackers get full access harvesting sensitive information and taking control over the application for their personal benefit. The aim of this work is to acknowledge multiple security methods, such as parameterized statements, parameterized stored procedures, customized error messages and input validation type as efficient means for preventing SQLIA simulated on an online-based database application, MoviesBox. The successful prevention of the attack was confirmed through conducting a series of performance tests after the injection of malicious codes and Quality of experience (QoE) methods implementation.**

Keywords-Web Application; SQL Injection; Cyber Security; Defense; Database Security.

I. INTRODUCTION

The growth of corporate Web applications (Web apps) provides many opportunities for e-businesses to grow faster. These have become a significant communication channel among different kinds of service providers and clients over the Internet. However, the beneficial opportunities of Web apps also increased the security issues of a third party interfering. Even though there are many approaches for us to test the flaws and vulnerabilities, Web apps demand a more technology-independent solution.

Web apps are frequently vulnerable to attacks due to time and financial constraints, poor programming skills and lack of security awareness. These flaws provide opportunities for accessing sensitive information data that can lead to serious consequences and great damage. Therefore, an attacker can compromise this configuration faults and gain a full illegal access to user sensitive data. For this reason, governments, as well as many corporations and research communities, are paying increased attention to this issue in order to prevent its progression [1].

One of the top ten most dangerous attacks on Open Web Application Security Project (OWASP) is the SQL Injection Attack. This type of attack can do serious damage to database-driven applications, such as manipulation of the user input data, silent spying and monitoring, even corrupt and delete an entire database and gain unauthorized access to other network servers. Normally, this attack is done by injecting some malicious SQL code to the actual query driven by the application program in order to traverse, insert, update or delete the data. Even though there are number of mechanisms to detect the SQL injection attack, serious research has to be carried out to reveal the hidden and unexploited paths of these mechanisms, which may strengthen the defense against SQLIA. Based on literature review, our research did not find sufficiently favorable results concerning coding flaws level. Thus, there is a solid need to provide additional "rules" to the developers to secure Web apps from attacks. Numerous organizations have spent a great deal of money on antivirus programs, data leakage prevention systems and network firewalls on the off chance that software engineers follow the appropriate guidelines, hopefully saving a considerable amount of cash on cyber-attack prevention. As mentioned earlier, the coding flaws are crucial and they can lead to serious vulnerabilities in Web apps simply because they are easy to discover and abuse [2]. Therefore, the purpose of this work is to serve as basic guidelines to programmers to write code in a more secure manner, with the end goal of shielding Web apps from cyber-attacks.

The rest of the paper is structured as follows. In Section II, there is a detailed overview of the latest studies published regarding the SQLIA and the preventive approach that will be established in contrast to the already proposed methods. Section III gives a general idea and benefits of using Web apps, a detailed description of security, as well as types of malicious attacks and a background detail of SQLIA. In Sections IV, V, VI and VII, a vulnerable Web app made for SQLIA testing is presented and a method is proposed for protection of the aforementioned attack, execution of the proposed method and a penetration testing [3] that will prove the effectiveness of the implemented prevention techniques. Section VIII includes a test of the performance of the proposed Web app conducted on thirteen people. Sections IX and X present the conclusions and future work.

II. RELATED WORK

Because of the significant impact of Web apps in modern networking, attempts must be made for ensuring their safety. Following this prospect, many computer engineers are constantly trying to establish new and improve the already used techniques for prevention of SQLIA. A detailed overview of the studies published concerning these attacks has shown that, so far, promising efforts have been made in these fields, which guarantee the safety of Web apps and serve as tools for decreasing the rates of data theft.

A. Input validation attack including SQLIAs and Cross Side Scrpting (XSS)

An application is considered vulnerable when it does not properly filter or validate the entered data by a user on a Web page [4].

B. Static analysys and automated reasoning

Static analysis is one of the most often used techniques for analyzing the code [5] [6]. These techniques are used to detect the vulnerable code by scanning the Web app with the use of heuristic or information flow analysis. Moreover, they can also produce a false positive and false negative result because of the conversion of suspicious input. A combination of static analysis and automated reasoning techniques is the most suitable for detection of queries that contain tautologies by the Web app.

C. Encryption of confidential data

Encryption of confidential data stored in a database will not allow an unauthorized user to read confidential data even if it gets access by employing any kind of malicious technique [7].

In contrast to other published papers regarding investigations of SQLIA techniques and concisely describing each one of them, in this paper, SQLIA will be described in detail supported by appropriate examples. Additionally, the exploited vulnerabilities used for injection of malicious queries will provide knowledge to Web developers about the most exploited vulnerabilities, at the same time briefly describing the impact of SQL Injection. As opposed to [8]-[12], this paper will also propose useful guidelines and will illustrate a different approach to differentiate types, techniques and tools of SQLIA.

III. WEB APPLICATION AND SECURITY

Web development is generally associated with building Web sites for the Internet. It includes the development of a wide range of applications from simple plain text Web pages to complex Internet applications, mostly intended for electronic businesses and social networks. In scientific terms, a Web application is any computer program which uses Web browsers or technology as means for performing various tasks on the Internet.

The security of Web apps is the most important component of any e-commerce corporation, but, when deployed online, it is in the Internet's nature to expose properties for attacking Web apps from various locations using different levels of scale and complexity. Therefore, Web application security is essential and it deals specifically with security surrounding Websites, Web apps and Web services. Common methods of attacks, or "vectors", range from targeted database exploitation to large-scale network disruption. Such methods are:

- Cross Site Scripting (XSS) – attack where the hacker attaches code at the end of the Websites URL or it posts directly onto a page with user content and that attack will mostly execute when the victim loads the Website. Moreover, XSS is a client-side code injection attack.
- SQL Injections Attacks (SQLIAs) – typically occur by sending malicious SQL queries to an Application Programming Interface (API) endpoint provided by a Website or service, allowing the attacker to gain root access to a machine.
- Denial-of-Service (DoS) and Distributed Denial-of-Service (DDoS) – type of cyberattack where the hacker is able to perform traffic attack on a targeted server. The server then will no longer process incoming requests effectively and eventually deny service to legitimate users' incoming requests.
- Memory Corruption- this happens when a memory location is unintentionally changed and attackers take advantage of it to perform code injections or buffer overflow attacks.
- Buffer Overflow – by injecting malicious code into the memory, the buffer's ability to overflow can be exploited, potentially creating faults in the target machine.
- Cross-Site Request Forgery (CSRF) – type of attack that accidently tricks a user to change passwords, emails or transfer funds, which allows attackers to take control of the Web application.

IV. SQL INJECTION ATTACK

In this paper, more details will be given about SQLIA. Right now, SQLIA stands among the most dangerous threats to databases and Web apps. It typically includes malicious updates, modification of the user SQL input, either by changing the structure of existing conditions or by including additional conditions. Figure 1 demonstrates how an SQL Injection Attack is performed.

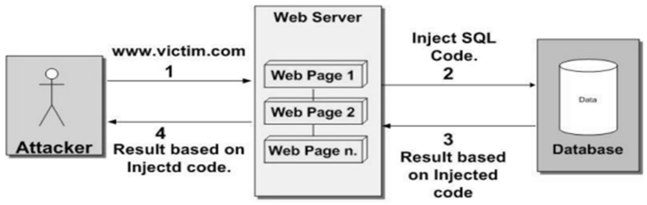


Figure 1. SQL Injection Attack

a) A user is accessing a Web application by typing the address in the URL.

b) The attacker injects malicious code to the Web application.

c) The malicious SQL-query is passed to the database server from the Web server.

d) The database management system sends the results based on the injected code back to the Web server. The results can be some data or error message or confirmation, depending on the injection type.

e) The Web server sends/shows the same result back to the attacker.

V. WEB APPLICATION AND SQLIA IMPLEMENTATION

MoviesBox is developed as an online database Web app, which is later used for SQLIA simulation. It contains three pages front-end written in Hypertext Markup Language 5 (HTML5), Cascading Style Sheet 3 (CSS3), Java Script and Bootstrap framework. For the server-side functionality, we used Hypertext Preprocessor (PHP) Language, MySQL-an Oracle-backed open source relational database management system (RDBMS) based on Structured Query Language (SQL), MySQL database and The Movie Database 3 (TMDb3) API. The application uses three types of authentication such as guest session, user and admin authentication.

In Figure 2, the Back End (Server side) and the Front End (Client Side) are presented.

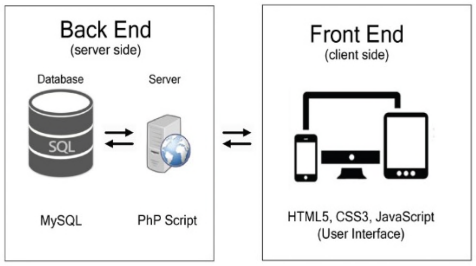


Figure 2. Back End (Server side) and Front End (Client Side)

A. Types of SQLIA

In order to prevent SQLIA, it is necessary to know the various techniques by which the attackers explore the vulnerabilities of the code and find the way to attack. They can be executed in the following ways:

1) Boolean-Based Blind: SQL injection technique that relies on statements that are always true. It is based on Boolean values (true or false), as suggested by the name, so the queries always give results by evaluating the condition WHERE.

SELECT * FROM users WHERE username = 'john' AND password = '1234';

In this scenario, it was assumed that the user had John as username and 1234 as password. This code can be exploited by adding a condition that is always true and just comment out the password part. By inputting the following, the attacker can easily gain access as user: `xxx@xxx.xxx'` OR 1=1 LIMIT -- '] for the username field; `xxx` as password. The new dynamic statement will be:

SELECT * FROM users WHERE username = 'xxx@xxx.xxx' OR 1=1 LIMIT -- '] ' AND password = 'xxx';

- xxx' ends with a single quote, which is a character limiter in SQL. With ' we delimit strings and we can test if the strings are properly escaped in the application or not.
- OR 1=1 LIMIT is a condition that is always true and limits the returned results to only one record.
- --'] is an SQL comment that removes the password part.

2) Union- Based Blind: type of attack that attackers use to obtain information from the database by extending the original query results. In other words, the attacker takes advantage of the UNION operator, which is used in SQLIA to join a query to the original, intentionally forged by the tester. The tester can access the values of columns of other tables by joining the results of the forged and original query. This means, the attackers are using this technique because they are not able to edit the original query to obtain what they want and this is the only way of running two or more SELECT statements into a single result.

SELECT username FROM userdata WHERE id='23';

The injected query will look as follows:

SELECT username FROM userdata WHERE id=' '
UNION
SELECT * FROM userdata.

This code will return all the detailed information of the table userdata.

3) Time-Based Blind: If there is a possibility when the hacker has no other way to retrieve information from the database server, the time-based blind method is used. The attacker uses an SQL statement which contains a particular database function to cause a time delay. The possibility of obtaining some information depends on the time it takes to get the server response. This kind of attack is not only used for determining the vulnerabilities, but also for extracting data from the server by integrating a time delay in a conditional statement. Consider the followin SQL statement:

SELECT * FROM users WHERE id=2;

By using time-based blind technique, the new injected query will be:

SELECT * FROM users WHERE id=2-SLEEP(20);

With the injected query, the attacker can only identify if the parameter is vulnerable to SQLIA. As mentioned before, time-based blind technique can not only check Web app's vulnerability, but also verify its database version and extract data from it. If the server responds in 20s, it can be concluded that the database is running on MySQL 5.0 server. This is done by using the malicious query:

```
SELECT * FROM users WHERE id=2-
IF(MID(VERSION(),1,1) = '5', SLEEP(20), 0);
```

4) Error-Based Blind: this technique is based on errors. Getting these errors indicates that the Web app is connected to a database and it is vulnerable to SQLIA. The injection of malicious code in the query that produces errors is done by sending or typing additional text to the server by Uniform Resource Locator (URL). After getting the error, it can be assumed what target is going to be next. To test whether the Web app is vulnerable, we can just put a single quote at the end.

VI. SQL Injection Prevention Techniques

1) Parameterized Queries

The use of parameterized statements, also known as prepared statements, can reduce the SQLIA by constructing the SQL-queries in a more secure way. If used exclusively, these statements completely remove the risk of all the SQLIA types such as tautology, timing attack, end of line comment and piggy backed attacks. Besides securing the Web app, prepared statements have another advantage because they help increase the work speed when executing the same or similar statements repeatedly.

```
$q = $conn->prepare("SELECT * FROM userdata WHERE
Username = ? && Password = ?");
$->bind_param("ss", $username, $password);
$q->execute();
```

In this code, implemented in MoviesBox, it is obvious that for the prepared statement we use a question mark (?) to substitute the parameters (integer, string, double or blob value). The second function binds the parameters and it is sending information to the database about what the parameters are. The "ss" informs the database server that the parameter is a string. The last function executes the parameters, where a dangerous SQL string will look as follows:

```
$q = "SELECT * FROM userdata WHERE Username =
'$username' && Pass = '$password'";
```

The main difference is the $q->execute(); method where the data is being passed. In the code with prepared statements, the parameterized string and parameters are passed to the database separately, enabling the driver to read them correctly, while the SQL statement in the second code is created before invoking the driver, meaning it is vulnerable to malicious parameters. This method can be very useful against SQLIA.

It is safe to say that these methods are currently the only and fundamental way to defend Web apps from this attack.

2) Parameterized Stored Procedures

Another prevention technique is using parameterized stored procedures. This includes a prepared SQL code that can be saved and reused as many times as needed without having to duplicate. Furthermore, they help reduce the network traffic between the database server and Web server just by sending the name of the stored procedure without having to send the SQL statement. As far as the security of the Web app is concerned, these procedures write the query in advance by placing parameter markers, so that data can be collected later.

```
/*!50003 CREATE DEFINER=`root`@`localhost`
PROCEDURE `validate_login`(
IN _username varchar(20),
  _pass VARCHAR(50)
)
BEGIN
SELECT * FROM userdata WHERE Username =
_username && Pass = _pass;
END */$$
DELIMITER ;
```

Second, in order to execute the stored procedure, a call function is used in the php code.

```
$q = $conn->prepare("CALL validate_login(?,?)");
$q->bind_param("ss", $username, $password);
$q->execute();
```

The main idea on how parameterized stored procedures work concerning security is that we can allow access to a stored procedure that updates a table, but forbid access to the table itself. This means, users would not have direct access to the database tables, but can only execute particular stored procedures.

3) Customized Error Messages

Error messages are also flaws that attackers use to gain access. Errors are visible when an invalid SQL statement is performed. This means, for any invalid SQL instruction that is identified when executing, the database will produce an error. By getting these messages, attackers gain information regarding the database and how to easily attack the Web app. It should also be noted that some powerful SQLIAs are entirely based on database errors such as unexpected quote, incorrect table name, etc. If these errors are completely removed, then attacking will become a difficult task. In order to prevent this, the use of customized error messages will reduce the possibility of SQLIA.

```
if ($conn->connect_error) {
  die("Error: There is something error".$conn->connect_error);
} else {
   echo "";
}
```

As mentioned before, error messages are retrieved from the SQL server as a response to any error query that is sent. The hacker can get important information about the target and retrieve table's name, stored procedure's name, etc. By using this method, we are not giving the attacker full access.

4) Input Data Type Validation

There are two ways on how SQLIA can be performed. The first is by injecting a command into a numeric parameter and the second into a string parameter. Programmers can avoid small attacks even if they do a simple input checks. The correct validation of the input data type such as string or numeric type plays a great role in the prevention of getting attacked. For example, if the user enters the input data incorrectly, then the incorrect input would be rejected due to the declared data type.

```
$q->bind_param("ss", $username, $password);
```

In the code above "ss" specifies the variable type, which is: "string, string".

VII. Penetration Test and Results from proposed techniques

Penetration testing, also known as pen test, is an authorized simulated cyberattack on a computer system for its security evaluation. It identifies the weaknesses as well as strengths.

To execute the proof of demonstration, SqlMap and WebSpy Chrome plug-in were used for penetration testing. SqlMap is an open source penetration testing tools that automates the process of detecting and exploiting SQL injection flaws and taking database servers, while WebSpy monitors HTTP GET/POST requests of any Website and allows them to be viewed and to be tested.

Firstly, the penetration testing was made on vulnerable MoviesBox. After testing and analyzing the Web app MoviesBox, it can be concluded that most of the parameters in the pages are vulnerable to Boolean-based blind, Error-based, Time-based and Union-based blind injections (Table I). Besides, SqlMap was able to fetch the versions and type of the technology used. After more simulations, SqlMap obtained information not only about the database wanted, but all databases and tables on the server.

TABLE I. SQLMAP TEST RESULTS FROM VULNERABLE WEB APP

#	SqlMap results on vulnerable pages of MoviesBox			
	SQL Injection type	***Login Form***	***Register Form***	***Request Form***
1	Error-based Blind	√	√	√
2	Boolean-based Blind	√	√	
3	Time-based Blind	√	√	√
4	Union-based Blind			√

With implementing the abovementioned prevention techniques in MoviesBox, a penetration testing in SqlMap has been made by testing the login, register and request parameters; it showed great success (Table II).

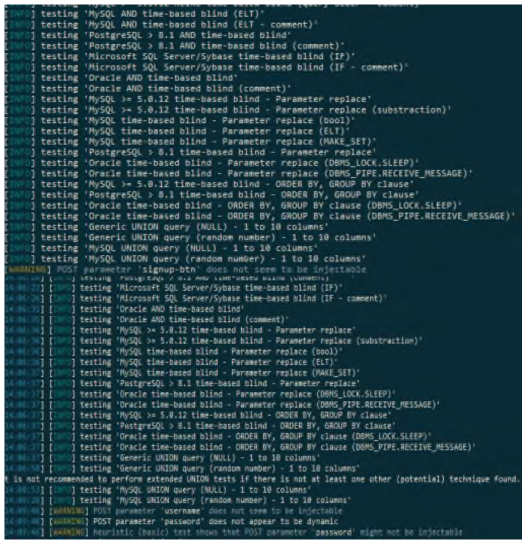
TABLE II. SQLMAP TEST RESULTS FROM PROTECTED WEB APP

#	SqlMap results on protected pages of MoviesBox			
	SQL Injection type	***Login Form***	***Register Form***	***Request Form***
1	Error-based Blind	Χ	Χ	Χ
2	Boolean-based Blind	Χ	Χ	
3	Time-based Blind	Χ	Χ	Χ
4	Union-based Blind			Χ

The evaluation of the proposed techniques though the penetration testing showed that our preventive approach is effective. Table II shows that various SQLIA simulated on the Web app after implementing parameterized queries, stored procedures, customized error message and input validation. The SqlMap testing tool cannot exploit the parameters, which means that our approach is effective. Furthermore, extended security analysis was made through an entire Web app, not just the mentioned, and spotted a number of injectible points, which later on were corrected with the aforementioned techniques. Figure 3 presents various SQLIA simulated in the penetration test on the Web app after implementing the preventive methods. As it can be noticed from the tables, SQL Injection type, Login form, Register form, and Request form presented a suitable approach for this research.

For each of these issues, Error-based blind, Boolean based blind, Time-based blind and Union-based blind are included for testing. SqlMap test results on vulnerable pages of MoviesBox are performed and commented in Table I.

The SqlMap test results from the protected Web app are presented and commented in Table II. The Error-based blind concept enhances the Login Form, Register form and Request form in both scenarios. Additional testing will be done on this Web app once is in production to detect possible server or hosting failures. The evaluation of the proposed techniques though the penetration testing show that our preventive approach is effective.



```
[INFO] testing 'MySQL AND time-based blind (ELT)'
[INFO] testing 'MySQL AND time-based blind (ELT - comment)'
[INFO] testing 'PostgreSQL > 8.1 AND time-based blind'
[INFO] testing 'PostgreSQL > 8.1 AND time-based blind (comment)'
[INFO] testing 'Microsoft SQL Server/Sybase time-based blind (IF)'
[INFO] testing 'Microsoft SQL Server/Sybase time-based blind (IF - comment)'
[INFO] testing 'Oracle AND time-based blind'
[INFO] testing 'Oracle AND time-based blind (comment)'
[INFO] testing 'MySQL >= 5.0.12 time-based blind - Parameter replace'
[INFO] testing 'MySQL >= 5.0.12 time-based blind - Parameter replace (substraction)'
[INFO] testing 'MySQL time-based blind - Parameter replace (bool)'
[INFO] testing 'MySQL time-based blind - Parameter replace (ELT)'
[INFO] testing 'MySQL time-based blind - Parameter replace (MAKE_SET)'
[INFO] testing 'PostgreSQL > 8.1 time-based blind - Parameter replace'
[INFO] testing 'Oracle time-based blind - Parameter replace (DBMS_LOCK.SLEEP)'
[INFO] testing 'Oracle time-based blind - Parameter replace (DBMS_PIPE.RECEIVE_MESSAGE)'
[INFO] testing 'MySQL >= 5.0.12 time-based blind - ORDER BY, GROUP BY clause'
[INFO] testing 'PostgreSQL > 8.1 time-based blind - ORDER BY, GROUP BY clause'
[INFO] testing 'Oracle time-based blind - ORDER BY, GROUP BY clause (DBMS_LOCK.SLEEP)'
[INFO] testing 'Oracle time-based blind - ORDER BY, GROUP BY clause (DBMS_PIPE.RECEIVE_MESSAGE)'
[INFO] testing 'Generic UNION query (NULL) - 1 to 10 columns'
[INFO] testing 'Generic UNION query (random number) - 1 to 10 columns'
[INFO] testing 'MySQL UNION query (NULL) - 1 to 10 columns'
[INFO] testing 'MySQL UNION query (random number) - 1 to 10 columns'
[WARNING] POST parameter 'signup-btn' does not seem to be injectable
[illegible] [INFO] testing 'Microsoft SQL Server/Sybase time-based blind (IF)'
[illegible] [INFO] testing 'Microsoft SQL Server/Sybase time-based blind (IF - comment)'
[illegible] [INFO] testing 'Oracle AND time-based blind'
[illegible] [INFO] testing 'Oracle AND time-based blind (comment)'
[illegible] [INFO] testing 'MySQL >= 5.0.12 time-based blind - Parameter replace'
[illegible] [INFO] testing 'MySQL >= 5.0.12 time-based blind - Parameter replace (substraction)'
[illegible] [INFO] testing 'MySQL time-based blind - Parameter replace (bool)'
[illegible] [INFO] testing 'MySQL time-based blind - Parameter replace (ELT)'
[illegible] [INFO] testing 'MySQL time-based blind - Parameter replace (MAKE_SET)'
[illegible] [INFO] testing 'PostgreSQL > 8.1 time-based blind - Parameter replace'
[illegible] [INFO] testing 'Oracle time-based blind - Parameter replace (DBMS_LOCK.SLEEP)'
[illegible] [INFO] testing 'Oracle time-based blind - Parameter replace (DBMS_PIPE.RECEIVE_MESSAGE)'
[illegible] [INFO] testing 'MySQL >= 5.0.12 time-based blind - ORDER BY, GROUP BY clause'
[illegible] [INFO] testing 'PostgreSQL > 8.1 time-based blind - ORDER BY, GROUP BY clause'
[illegible] [INFO] testing 'Oracle time-based blind - ORDER BY, GROUP BY clause (DBMS_LOCK.SLEEP)'
[illegible] [INFO] testing 'Oracle time-based blind - ORDER BY, GROUP BY clause (DBMS_PIPE.RECEIVE_MESSAGE)'
[illegible] [INFO] testing 'Generic UNION query (NULL) - 1 to 10 columns'
[illegible] [INFO] testing 'Generic UNION query (random number) - 1 to 10 columns'
t is not recommended to perform extended UNION tests if there is not at least one other (potential) technique found.
[illegible] [INFO] testing 'MySQL UNION query (NULL) - 1 to 10 columns'
[illegible] [INFO] testing 'MySQL UNION query (random number) - 1 to 10 columns'
[illegible] [WARNING] POST parameter 'username' does not seem to be injectable
[illegible] [WARNING] POST parameter 'password' does not appear to be dynamic
[illegible] [WARNING] heuristic (basic) test shows that POST parameter 'password' might not be injectable
```

Figure 3. Results from pen test on protected MoviesBox.

VIII. Quality of Experience (QoE) for Web apps performances

Performance testing is an important aspect when running a Web app. The main goal is to evaluate user experience on the Web app.

Quality of Service (QoS) refers to the technical and operational aspects of a service, such as time to support services, capacity, and transport. Quality of Experience (QoE) measures the difference between what users expected and what they actually received. Using the QoE is beneficial to estimate the perception of the user about the quality of a particular service and it depends on the customer's satisfaction in terms of usability, accessibility, retaining ability and integrity of using a specific service. QoE means overall acceptability of an application or service, as perceived subjectively by the end-user and represents a multidimensional subjective concept that is not easy to evaluate. In this work, QoE evaluation is used in order to measure the quality of the Web apps performances.

This test establishes whether the App meets user expectations such as: speed, usage, security, etc. Similarly, a test has been performed among 100 users from our university in order to evaluate the performance of the Web app MoviesBox. The results for testing the Usability, Load Time, GUI and Security, are presented in the following figures. The results were obtained according to a survey sent to university staff and students. As it can be seen later on, the concept of Security gives the best results based on Quality of Experience Analysis and tests performed.

A. Usability Test

The usability performance for MoviesBox showed that 87.0% of the people (Figure 4) answered that the Web application is easy to use.

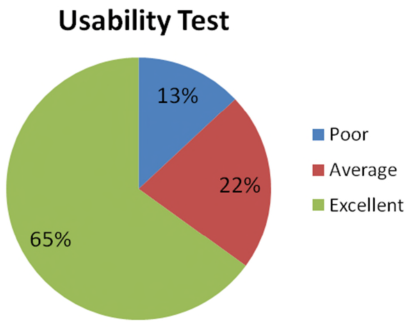


Figure 4. Usability Test

The instructions provided in the Web app are clear and satisfy its purpose. As far as user task analysys is concerned, MoviesBox showed excelent results regarding efficiency and accuracy. The provided main menu on each page was graded as user friendly, as well as the content. 13.0% of the people where not satified and proposed that more content will improve the Web app.

B. Graphic User Interface (GUI)

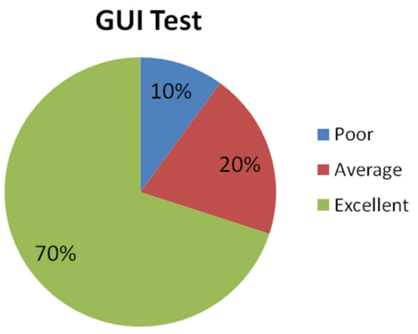


Figure 5. Graphic User Interface Test

GUI test showed favorable results since 90.0% of the people answered that the font used is readable, the alignment of the text is proper, as well as the color of the font (Figure 5). Regarding the GUI elements such as size, position, width, length and acceptance of characters and numbers in the inputs fields in MoviesBox, users were satisfied. GUI is an important concept of great interest for a large number of users. That is why this concept was tested for this application as well.

Interfaces are an important part of each application performance. The graphic display of the interface is important, especially with users' experience of this kind of applications.

Tests have also shown that the images used in MoviesBox had good clarity and error messages were displayed correctly. 10.0% of the people did not like the font and the colors used, as well as the error messages.

C. *Load Time Test*

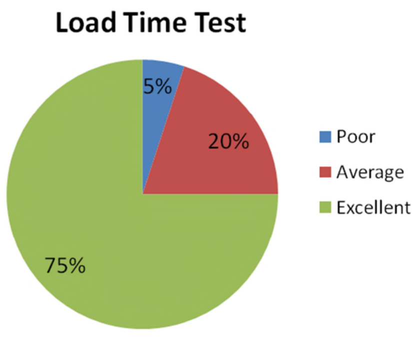


Figure 6. Load Time Test

After simultaneous usage by thirteen different accounts, this Web app showed huge rates of success. As shown in Figure 4, to 95.0% it took 3000 ms to load the Web page and it showed no error. It is known that usually users wait 10s and give up. Users were also satisfied with the short time for login (2505 ms). 5.0% were not satisfied and suggested implementation of a cache buster parameter to the URLs, which will make the request to bypass any full page.

D. *Security Test*

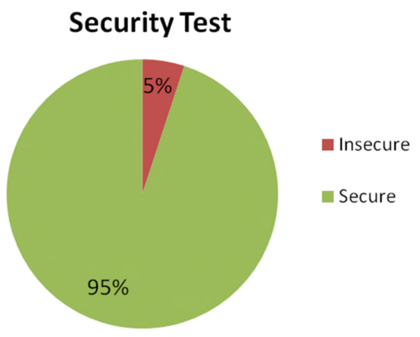


Figure 7. Security Test

Besides automatic penetration testing, a manual security test was done on MoviesBox among several people. The security test showed unauthorized access was not permitted and sessions were killed after prolonged user inactivity due to the aforementioned methods that were implemented. The threats were identified and showed that there were no potential vulnerabilities. As shown in Figure 7, 5.5% suggested captcha codes in the registration form and restriction of use of special characters, as well as proxy based application firewall, which is useful for detecting and blocking anything malicious.

IX. CONCLUSION

Web apps, as some of the most important "means of modern life", are still highly vulnerable to cyberattacks, mostly coming from unknown sources capable of inflicting serious damage. SQLIA is one of the most often used ways of compromising Web apps and since it is not enabled by technological flaws, it cannot be solved by the technology itself. Generally it is caused by the naive coding habits of developers, which make Web app firewalls or cloud computing powerless in resolving the security question against SQLIA, even though they can provide some level of protection. Therefore, this paper provides simple mechanisms, such as prepared statements, stored procedures, customized error messages as well as input data validation that showed a successful prevention of SQLIA on Web apps. Furthermore, the separate evaluation of these approaches showed that prepared statements are really the ones that are protecting the Web app. Without these, using just the rest of the methods still leaves the Web app vulnerable to SQLIA. Therefore, it is necessary to combine the right hardware together with multiple security approaches and more efficient coding in order to make the modern database systems safer.

X. FUTURE WORK

For future work, this application is planned to be working on Cloud of Things (CoT), due to many benefits including a small amount of disk storage, memory and resources necessary for execution of the app itself. Also, it can be approached by different users regardless of location and device with full support on different platforms and operating systems. There is independence of the app's upgrades from those of the machine software. Moreover, migrating this app on CoT will contribute to the reduction of the inconveniences (cost and other complexities) of direct hardware management. One major concern is that the CoT environment is susceptibility to cyberattacks. Therefore, the implementation of the aforementioned methods for SQLIA prevention is inevitable.

REFERENCES

[1] R. K. Knake, C.o.F.R.I. Institutions, and G. G. Program, Internet Governance in an Age of Cyber Insecurity. 2010: Council on Foreign Relations.

[2] M. van Steen and A. S. Tanenbaum, Distributed Systems. 2017: CreateSpace Independent Publishing Platform.

[3] T. O'Connor, Violent Python: A Cookbook for Hackers, Forensic Analysts, Penetration Testers and Security Engineers. 2012: Elsevier Science.

[4] X. G. R Chaudari and M.V. Vaidya, A Survey on security and Vulnerabilities of Web Application. 2014 IJCSIT, (IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 5 (2) , 2014, 1856-1860.

[5] Y. Xie and A. Aiken, Static detection of security vulnerabilities in scripting languages, in Proceedings of the 15th conference on USENIX Security Symposium - Volume 15. 2006, USENIX Association: Vancouver, B.C., Canada.

[6] V. B. Livshits and M.S. Lam, Finding security vulnerabilities in java applications with static analysis, in Proceedings of the 14th conference on USENIX Security Symposium - Volume 14. 2005, Jul 31, 2005 - SSYM'05 Proceedings of the 14th conference on USENIX Security Symposium Baltimore, MD — July 31 - August 05, 2005.

[7] U. S. M. Agarwal and K. S. Rana, "A Survey of SQL Injection Attacks". International Journal of Advanced Research in Computer Science and Software Engineering. Vol.(3): pp. 286-289, 2015.

[8] S. Sarasan, "Detection and Prevention of Web Application Security Attacks". International Journal of Advanced Electrical and Electronics Engineering. Vol.(3): pp. 29-34, 2013.

[9] R. Johari and P. Sharma, "A Survey on Web Application Vulnerabilities (SQLIA, XSS) Exploitation and Security Engine for SQL Injection". 2012 International Conference on Communication Systems and Network Technologies, 2012. pp. 453-458.

[10] M. R. Borade and N. A. Deshpande, "Extensive Review of SQLIA ‘ s Detection and Prevention Techniques", 2013.

[11] D. A. Kindy and A. S .K. Pathan, "A Detailed Survey on Various Aspects of SQL Injection in Web Applications: Vulnerabilities, Innovative Attacks, and Remedies". IJCNIS, 2013.

[12] D. A. Kindy and A. K. Pathan, "A survey on SQL injection: Vulnerabilities, attacks, and prevention techniques". 2011 IEEE 15th International Symposium on Consumer Electronics (ISCE), 2011. pp. 468-471.