UDC 519.2

B. ARSLAN, R. GAMZAYEV, E. KARAÇUHA, M. TKACHUK

ALGORITHMS AND SOFTWARE SOLUTIONS FOR SQL INJECTION VULNERABILITY TESTING IN WEB APPLICATIONS

Software security gains importance day by day and developers try to secure web applications as much as possible to protect confidentiality, integrity and availability that are described in the fundamental security model so-called CIA triad. SQL injection vulnerability which can violate the confidentiality and integrity principles of the CIA triad is reviewed, and SQL injection attack execution and protection techniques are explained. The common frameworks' solutions against SQL injection vulnerability were compared, and this comparison shown the most used techniques in this domain. Error-based and time-based detection algorithms for SQL injection's identification are developed to create a vulnerability scanner that can detect SQL attacks which cause vulnerability in web applications, and these algorithms are represented in form of UML-activity diagrams. In order to discover all possible links and forms to perform SQL injection vulnerability tests in the entire website, a web crawler is needed. Breadth-First Search (BFS) algorithm for developing the web crawler is proposed, and the appropriate pseudo code and activity diagram are provided. Besides, Common Vulnerability Scoring System (CVSS) that is used to measure severity score of attacks that can violate CIA triad principles is reviewed. Qualitative severity score rating scale of CVSS is explained. An example of CVSS calculation is represented. Necessary components of a vulnerability scanner are explained. A vulnerability scanner prototype is developed using explained algorithms. Process results of this vulnerability scanner's usage for real web applications are represented. Conclusions are made, and goals of future work are defined.

Keywords: software security, web application, vulnerability, scanner, CIA triad, SQL injection, error-based detection, time-based detection, web-crawler, BFS-search algorithm, common vulnerability scoring system.

Б. АРСЛАН, Р. А. ГАМЗАЄВ, Е. КАРАКУХА, М. В. ТКАЧУК

АЛГОРИТМИ ТА ПРОГРАМНІ РІШЕННЯ ДЛЯ ТЕСТУВАННЯ ВРАЗЛИВОСТІ В ІНТЕРФЕЙСІ SQL У ВЕБ-ПРОГРАМАХ

Безпека програмного забезпечення щоденно набуває все більшого значення, і розробники намагаються максимально захистити веб-програми, щоб забезпечити їх конфіденційність, цілісність та доступність, які описані в основній моделі безпеки так званої тріади CIA. Розглянута вразливість SQL-ін'єкцій, яка може порушувати принципи конфіденційності та цілісності тріади ЦРУ та пояснюються виконання SQL-атак та методи захисту від них. Було проведено порівняння загальних структурних рішень для усунення вразливості SQL-ін'єкцій, яке виявило найпоширеніші технології у цій галузі. Розроблені алгоритми виявлення на основі помилок та на основі вимірі часу для ідентифікації SQL-ін'єкцій для створення сканера вразливості, який може виявити SQL-атаки, що викликають уразливість в веб-додатках, і ці алгоритми представлені у формі UML-діаграм активності. Щоб виявити всі можливі посилання та форми для виконання тестів вразливості на всьому веб-сайті, потрібен пошуковий веб-робот. Запропоновано алгоритм Breadth-First Search (BFS) для розробки веб-сканеру, для нього наведено псевдокод та діаграма активності. Розглядається система загальної оцінки вразливості (CVSS), яка використовується для вимірювання ступеня тяжкості атак, що можуть порушувати принципи захисту тріади ЦРУ. Роз'яснено якісну оціночну шкалу CVSS. Представлений приклад розрахунку CVSS. Розроблено прототип сканера вразливості з використанням запропонованих алгоритмів. Результати застосування цього сканеру вразливості представлені прикладами оцінки реальних веб-застосувань. Зроблено висновки, визначені цілі майбутньої роботи.

Ключові слова: безпека програмного забезпечення, веб-застосування, вразливість, сканер, тріада ЦРУ, SQL-ін'єкція, визначення на основі помилок, визначення на основі вимірів часу, загальна модель безпеки, пошуковий веб-робот, BFS - алгоритм пошуку, загальна система оцінки вразливості.

Б. АРСЛАН, Р. А. ГАМЗАЕВ, Э. КАРАКУХА, Н. В. ТКАЧУК

АЛГОРИТМЫ И ПРОГРАММНЫЕ РЕШЕНИЯ ДЛЯ ТЕСТИРОВАНИЯ УЯЗВИМОСТИ В SQL-ИНЪЕКЦИЯХ В ВЕБ-ПРИЛОЖЕНИЯХ

Безопасность программного обеспечения ежедневно приобретает все большее значение, и разработчики стараются максимально защитить веб-приложения, чтобы обеспечить их конфиденциальность, целостность и доступность, которые описаны в основной модели безопасности так называемой триады ЦРУ. Рассмотрена уязвимость SQL-инъекций, которая может нарушать принципы конфиденциальности и целостности триады ЦРУ, и объясняются как выполнение SQL-атак так и методы защиты от них. Было проведено сравнение общих структурных решений для устранения уязвимости SQL-инъекций, которое выявило самые распространенные технологии в этой области. Разработанные алгоритмы обнаружения на основе ошибок и на основе измерении времени для идентификации SQL-инъекций для создания сканера уязвимости, который может обнаружить SQL-атаки, вызывающие уязвимость в веб-приложениях, и эти алгоритмы представлены в форме UML-диаграмм активности. Чтобы выявить все возможные ссылки и формы для выполнения тестов уязвимости на всем сайте, нужен поисковый веб-робот. Предложен алгоритм Breadth-First Search (BFS) для разработки веб-сканера, для него приведены псевдокод и диаграмма активности. Рассматривается система общей оценки уязвимости (CVSS), которая используется для измерения степени тяжести атак, которые могут нарушать принципы защиты триады ЦРУ. Разъяснено качественную оценочную шкалу CVSS. Представлен пример расчета CVSS. Разработан прототип сканера уязвимости с использованием предложенных алгоритмов. Результаты применения этого сканера уязвимости представлены примерами оценки реальных веб-приложений. Сделаны выводы, определены цели будущей работы.

Ключевые слова: безопасность программного обеспечения, веб-приложение, уязвимость, сканер, триада ЦРУ, SQL-инъекция, определение на основе ошибок, определение на основе измерений времени, общая модель безопасности, поисковый веб-робот, BFS - алгоритм поиска, общая система оценки уязвимости.

Introduction: Problem Actuality and Research Goal. In direct proportion to the popularity of web applications, information security gains importance to protect the data from adversaries, because insecure applications can leak sensitive information to their users. This information can be used by adversaries to manipulate the data for different motivations [1], such as; curiosity, wealth, recognition, national security, etc. The fundamental security model [2] which is also referred as CIA triad, is designed to provide a baseline standard for evaluating and implementing information security regardless of the underlying system. CIA triad has three main principles [2] as follows:

(1) Confidentiality is the protection of information from unauthorized access. Confidentiality principle states that access to information must be granted only on a "need-to-know" basis, so that information which is only available to some individuals should not be accessible by everyone;
(2) Integrity is maintaining the consistency, accuracy and trustworthiness of the information over its entire life-cycle. Integrity principle makes sure that the information is not tampered whenever it travels from source to destination or even at rest;
(3) Availability principle ensures that the information concerned is readily accessible to authorized individuals when it is needed.

In order to have a wider perspective on the subject, web application vulnerabilities are classified by the main phases of each software development life cycle as shown in Fig. 1. This classification does not show all existing vulnerabilities. Instead, it contains the most commonly known and dangerous vulnerabilities.

In this paper, implementation caused SQL injection (SQLI) vulnerability that can violate confidentiality (1) and integrity (2) principles of the CIA triad is analyzed. SQLI vulnerability analysis includes the attack execution, countermeasures against such attack and solutions that are applied by commonly used frameworks to secure applications against such attacks.

Moreover, error-based and time-based SQLI vulnerability detection algorithms are developed and explained. The main purpose for these detection algorithms is to develop a vulnerability scanner (VS) that will simplify penetration testers' job to find vulnerabilities. That is why, it is also important to discover all links and forms in the targeted website for SQLI vulnerability detection test executions. Therefore, Breadth-First Search (BFS) [4] algorithm is used to develop a web crawler to explore the targeted website for SQLI tests to be performed. Furthermore, Common Vulnerability Scoring System [5] (CVSS) is analyzed to give testers an opportunity to measure severity score of attacks to compare different possible attack scenarios to find out the most dangerous one for the application. This will help to prioritize vulnerabilities for developers to apply countermeasures.

Ultimately, the main research goal of this paper is to consider SQLI attack execution, its countermeasures, commonly used frameworks' solutions and to develop algorithms for exploring the targeted website and for performing SQLI vulnerability detection tests.

Overview of the SQLI vulnerability. Any application that has an SQL database (DB) to save any type of data is at risk of SQLI attacks [6]. A SQLI attack consists of insertion or injection of an SQL query via the input data from the client to the application and a successful SQLI exploit can perform CRUD operations meaning that it can read sensitive data from the DB, modify DB data by Insert, Update, Delete operations. Consequently, such attack can lead to violation of confidentiality (1) and integrity (2) principles of the CIA triad.

Alternatively, if the administrative operations are allowed to be executed remotely by DB Management System (DBMS), then adversaries can use SQLI vulnerability to execute administrative operations to shut down the DBMS which can cause violation of availability (3) principle of the CIA triad.

Because of the fact that SQLI attacks consist of injection of SQL query via the input data from the client to the application, it could be also said that SQLI attacks are a type of injection attacks [7] even though it was classified by the main phases of the software development life-cycle phases in Fig. 1.

According to statistics between the year 2000 to 2018 from the National Vulnerability Database (NVD) of National Institute of Standards and Technology (NIST) [8], there were upward trends on SQLI vulnerability from the year 2000 to 2006. The percentage of the SQLI vulnerability fell slightly from 14.60% in 2006 to 10.87% in 2007, and then rose significantly to 19.55% in 2008 which was the most popular time of SQLI vulnerability. Starting from the year 2009, the reported SQLI vulnerability percentage started decreasing until 2013 and stayed around 4% until 2015. In 2016, it fell to 1.35% and increased up to 3.52% in 2017. In the time when this statistic was obtained from NVD in 2018, there was 3.11 % of SQLI vulnerability reported among all. This may change by the end of 2018.

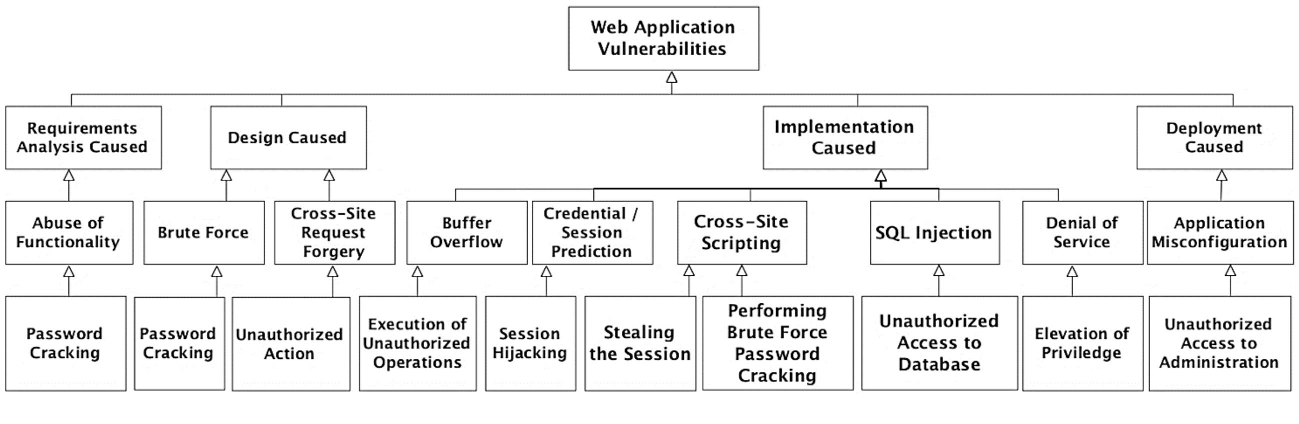

Web Application Vulnerabilities
Requirements Analysis Caused
Design Caused
Implementation Caused
Deployment Caused
Abuse of Functionality
Brute Force
Cross-Site Request Forgery
Buffer Overflow
Credential / Session Prediction
Cross-Site Scripting
SQL Injection
Denial of Service
Application Misconfiguration
Password Cracking
Password Cracking
Unauthorized Action
Execution of Unauthorized Operations
Session Hijacking
Stealing the Session
Performing Brute Force Password Cracking
Unauthorized Access to Database
Elevation of Priviledge
Unauthorized Access to Administration

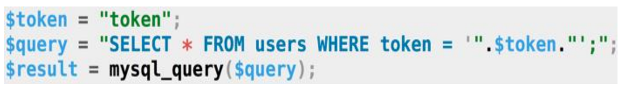
Figure 1 – Classification of web application vulnerabilities [3]

Even though the total amount of reported SQLI vulnerabilities are decreased after year 2008 and it is currently around 3.11% percent in 2018, this vulnerability is in existence, and it can still be dangerous to violate confidentiality (1) and integrity (2) principles of the CIA triad.

Overview of the underlying causes of SQLI vulnerability and SQLI attack execution. Above all, main reasons that causes SQLI vulnerability must be well-known to execute SQLI attacks, to detect SQLI vulnerability and to secure applications against such vulnerabilities. There are mainly two reasons as follows:

1. Direct usage of the input data from tainted source;
2. Direct usage of the input data to dynamically construct the SQL query.

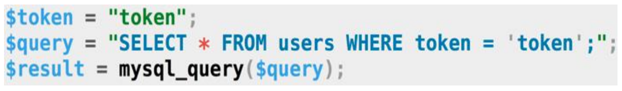
The underlying issue in the SQLI vulnerability is the fact that one string combines the code and the data to construct a query. In order to illustrate this issue, a basic SQL query to select data from DB can be used. Such queries have simply three keywords, such as: select, from, where (see Fig. 2).



```
$token = "token";
$query = "SELECT * FROM users WHERE token = '".$token."';";
$result = mysql_query($query);
```

Figure 2 – SQL select query sample

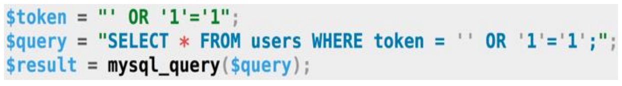
The above figure shows an example of SQL query to select all users from table named as ‘users’ where token matched. Expected query to be processed in DBMS is represented in Fig. 3.



```
$token = "token";
$query = "SELECT * FROM users WHERE token = 'token';";
$result = mysql_query($query);
```

Figure 3 – Expected SQL select query

An adversary can trick this query easily by adding extra code by providing unexpected token as shown in Fig. 4.



```
$token = "' OR '1'='1";
$query = "SELECT * FROM users WHERE token = '' OR '1'='1';";
$result = mysql_query($query);
```

Figure 4 – SQL injection with unexpected input data

Provided input value for token, which is ‘ OR ‘1’=‘1 in this case, perfectly fits on the query. Since there is no syntax error and ‘1’ always equals to ‘1’, DBMS will think query is fine to execute and return all users.

Sometimes when adversaries are not sure about how the query does look like to be sure that injection will work, they can use SQL comments to force DBMS to skip the rest of query. MySQL servers do accept two types of SQL comments:

1. Single line comment: such comments start with ‘#’ character or ‘--‘ sequence and end at the end of line;
2. Multi-line comments: such comments start with ‘/*’ sequence and end with ‘*/‘ sequence.

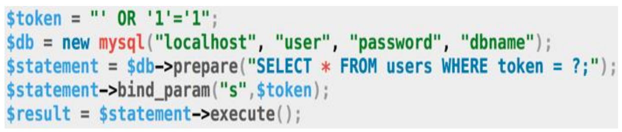
Additionally, injection with comment character or sequence of characters may not work without adding an extra space character at the end of input data after the comment character or character sequence. Thus, it is occasionally needed to add space character at the end of the malicious SQL query that contains comment character to execute a successful SQL injection attack.

Moreover, using URL encoded version of the malicious input data is also useful to successfully perform a SQLI attack.

Once the SQLI attack is successfully executed, adversaries understand the structure of the SQL query and they perform other attacks using different queries to manipulate the DB as they wish.

Overview of the SQLI vulnerability countermeasures. As explained above, the main reason for SQLI vulnerability existence is using the tainted input data directly to construct the SQL query without sanitization and using the tainted input data to construct SQL query dynamically. Therefore, securing the application against SQLI vulnerability is about increasing the trustworthiness of the input data or separating the data and the code. Techniques [9] used to eliminate SQLI vulnerability as follows:

1. Blacklisting: The action of detecting and deleting the special characters that can break SQL query, such as; quotation mark (“), single quotation mark (‘), semicolon (;), comment (/*, */, #, --), etc.;
2. Escaping: The action of replacing the problematic characters with the safe ones, such as; changing single quotation mark (‘), quotation mark (“), semicolon (;) to (\’), (\”), (\;) respectively;
3. Whitelisting: The action of allowing input data to be one of pre-defined values or in a pre-defined range. Whitelisting is hard to implement, because input data may not be predicted for rich input data;
4. Prepared statements: This is a feature that DBMS provides. In order to work with prepared statements, an SQL query template must be prepared. Prepared statement object contains not just SQL statement, but also pre-compiled SQL statement. Needed data will be provided later on and placed into related part of the query in prepared statement. An example prepared statement usage is shown in Fig. 5.



```
$token = "' OR '1'='1";
$db = new mysql("localhost", "user", "password", "dbname");
$statement = $db->prepare("SELECT * FROM users WHERE token = ?;");
$statement->bind_param("s",$token);
$result = $statement->execute();
```

Figure 5 – Prepared statement example

When the prepared statement is executed, DBMS can just run the SQL statement without compiling it first. This will result with faster work time and help to separate the code and the data in the SQL query. Thus, the best way to secure applications against SQLI vulnerability is to use prepared statements.

Web application frameworks are applying different solutions to SQLI vulnerability to help developers to easily get over this problem, see table 1.

It is important to understand that below table does not compare frameworks' capabilities, instead it compares their solutions against SQLI vulnerability that is described in their official documentations.

Many of these frameworks provide escaping feature to get rid of SQLI vulnerability while prepared statements resolve the main problem that causes SQLI vulnerability. That is why, encouraging developers to use prepared statements is a better solution to develop secure applications against SQLI vulnerability. Table 1 shows that only three frameworks, ASP.NET, Spring and Zend use prepared statements for this purpose.

Even though these frameworks provide solutions to SQLI vulnerability, applications developed using these frameworks may have this vulnerability if developers do not use the framework properly. Therefore, organizations hire penetration testers to perform tests to find existing vulnerabilities even if they are using web application frameworks which provide solutions against vulnerabilities. Penetration testers or ethical hackers are also using SQLI attack technique, that is discussed above, to find SQLI vulnerability in web applications.

Since it takes significant amount of time to manually perform vulnerability tests to detect vulnerabilities, automating the work has to be completed by penetration testers with the help of VS is a good idea because of the fact that it will increase the speed of these tests and provide with result much faster.

In order to create such VS, several algorithms are needed. These algorithms are explained later in the paper and they are as follows:

1. Error-based SQLI detection algorithm which can detect SQLI vulnerability by searching for DBMS error messages;
2. Time-based SQLI detection algorithm which can detect SQLI vulnerability based on the server response time;
3. Breadth-First Search (BFS) algorithm which can be used to develop a web crawler that explores the targeted website to find all links and forms for future use in SQLI vulnerability testing;
4. Common Vulnerability Scoring System (CVSS) which measures the qualitative severity score of vulnerabilities.

If VS detects a vulnerability, then it means that the application is not well secured, and immediately better solutions must be applied.

Overview of Breadth-First Search algorithm and its usage in development of a web crawler. BFS algorithm [4] is important for developing a web crawler to discover all links and forms in a website. These links and forms will be tested against SQLI vulnerability one after another until all links and forms in the entire website is tested. As a result, penetration tester will be providing URL for the home page of the website and VS will discover all links and forms in the website by the help of web crawler developed based on the BFS algorithm to perform SQLI test. Fig. 6 represents the pseudo-code of BFS algorithm.

```
unmark all vertices
choose some starting vertex x
mark x
list L = x
tree T = x
while L nonempty
choose some vertex v from front of list
visit v
for each unmarked neighbor w
    mark w
    add it to end of list
    add edge vw to T
```

Figure 6 – Pseudo-code of BFS algorithm

BFS algorithm starts at the root URL which is the one for home page of the website (represented as vertex in the pseudo-code) and searches all the neighboring URLs at the same level meaning that it searches for all links and forms in the current page. If the goal is reached, then it is reported as success and the search ends. If it is not, search proceeds down to the next level, sweeping the search across the neighboring URLs at that level and so on until the goal is reached. Goal is to discover all links and forms in the entire website. In order to have better understanding on the algorithm, Fig. 7 shows an activity diagram of BFS algorithm.

BFS algorithm is more suitable for applications and situations where the desired results can be obtained in the upper levels of a deeper tree. Its performance will get affected if the results will be found in the deeper levels. Since the purpose of the web crawler is to get all links and forms in entire website, this is not going to be an issue.

Table 1 – Web application frameworks comparison for SQLI vulnerability countermeasure

Framework	Blacklisting	Escaping	Whitelisting	Prepared statement
ASP.NET[10]	–	+	–	+
CodeIgniter[11]	–	+	–	+/–
Laravel[12]	–	+	–	+/–
Node.js[13]	–	+	–	+/–
Phalcon[14]	–	+	–	+/–
Ruby on Rails[15]	–	+/–	–	–
Spring[16]	–	–	–	+
Zend[17]	–	–	–	+

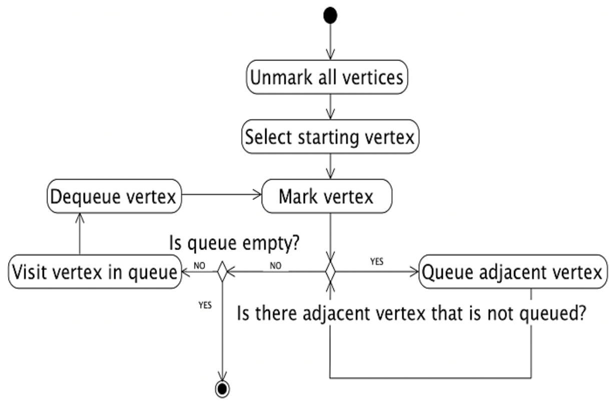


Figure 7 – Activity diagram of BFS algorithm

Developing an error-based SQLI vulnerability detection algorithm. Forms and links obtained from the web crawler which is developed based on the BFS algorithm, will be used to execute SQLI vulnerability tests to detect the vulnerability itself. Error-based SQLI detection algorithm is used in one of these tests. Fig. 8 contains activity diagram for this algorithm.

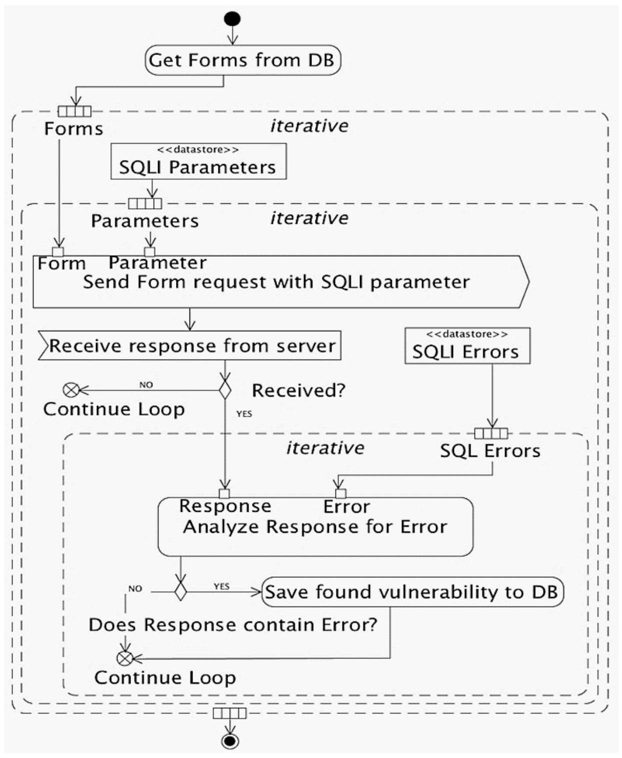


Figure 8 – Error-based SQLI detection algorithm

The main idea is to send malicious input data via found forms and links to the application. These malicious input data is explained earlier, but this time the goal is to break SQL query and force DBMS to produce an error. That is why, input data must be a character and must not complete the SQL query in a correct form. Therefore, input data can be a quotation mark (“), or single quotation mark (‘) or any character that will break the query including their URL encoded versions.

Next, the malicious input data will be processed by the application. If there is a SQLI vulnerability and injection of malicious code is successful, then the input data will break the SQL query and DBMS will prompt a DB error message. These error messages contain followings:

1. “error in your SQL syntax”;
2. “Microsoft OLE DB Provider for ODBC Driver error”;
3. “Invalid Query String”;
4. etc.

Finally, VS must analyze the response from the application to find one of pre-defined DB error messages.

Developing a time-based SQLI vulnerability detection algorithm. Activity diagram of time-based SQLI vulnerability detection algorithm for found forms is shown in Fig. 9 for a clear vision on the algorithm. Similarly, the same algorithm must be applied for the found links.

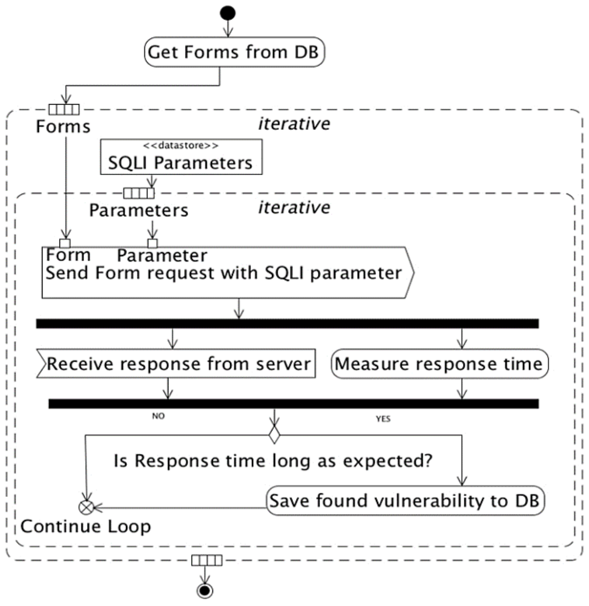


Figure 9 – Activity diagram of time-based detection algorithm

Time-based SQLI vulnerability detection technique consists of sending input data which forces DB server to wait for a certain amount of time that is defined in the malicious input data and on the other hand counting the time to get response from the server.

Several query examples which make DB server sleep for a while are as follows:

1. 1\” and sleep (10)-- ;
2. 1; wait for delay \’00:00:10\’;

These example queries make DB server wait for 10 seconds. This time is absolutely more than normal response time of any web server.

After sending the malicious input data to the application, if the response time is equal or greater than the pre-defined and expected time which is 10 seconds for the queries above, then it can be said that SQLI vulnerability has been found.

To accomplish this task, there must be two simultaneous actions running in the VS, such as; one action is sending a request to the server with malicious input data, and the another is measuring the response time of the server.

Overview of the Common Vulnerability Scoring System. The Common Vulnerability Scoring System (CVSS) provides a way to capture the principal characteristics of a vulnerability, and produce a numerical score reflecting its severity, as well as a textual representation of that score. The numerical score can then be translated into a qualitative representation (such as; low, medium, high and critical) to help organizations properly assess and prioritize their vulnerability management process.

CVSS is composed of three metric groups, such as; base, temporal and environmental.

The exploitability metrics reflect the ease and technical means by which the vulnerability can be exploited. On the other hand, the impact metrics reflect the direct consequence of a successful exploit and represent the consequence to the thing that suffers the impact, which is referred to formally as the impacted component.

The temporal metric group reflects the characteristics of a vulnerability that may change over time but not across user environments.

The environmental metric group represents the characteristics of a vulnerability that are relevant and unique to a particular user's environment.

When the base metrics are assigned values by analyst, the base equation computes a score ranging from 0.0 to 10.0. These metrics and equations are explained in CVSS v3.0 documentation [5].

Specifically, the base equation is derived from two sub equations: the exploitability sub score equation, and the impact sub score equation.

The exploitability sub score equation is derived from the base exploitability metrics, while the impact sub score equation is derived from the base impact metrics.

The base score can then be refined by scoring the temporal and environmental metrics in order to more accurately reflect the risk posed by a vulnerability to a user's environment. However, scoring the temporal and environmental metrics is not required.

The environmental metrics are specified by end-user organizations because they are best able to assess the potential impact of a vulnerability within their own computing environment.

The score calculated in CVSS can be between 0.0 and 10.0. Qualitative severity rating scale for both of numerical and textual score is shown in table 2.

Table 2 – Qualitative severity rating score

Metric value	Description
Not	0.0
Low	0.1…3.9
Medium	4.0…6.9
High	7.0…8.9
Critical	9.0…10.0

As an example, a CVSS base score of 4.0 has an associated severity rating of medium. The use of these qualitative severity ratings is optional and there is no requirement to include them when publishing CVSS scores. They are intended to help organizations properly assess and prioritize their vulnerability management processes.

Software architecture of a vulnerability scanner. In general, a VS is made up of four main modules, namely; a scan engine, a scan database, a report module and a user interface. To illustrate these modules, a component diagram of VS is shown in Fig. 10.

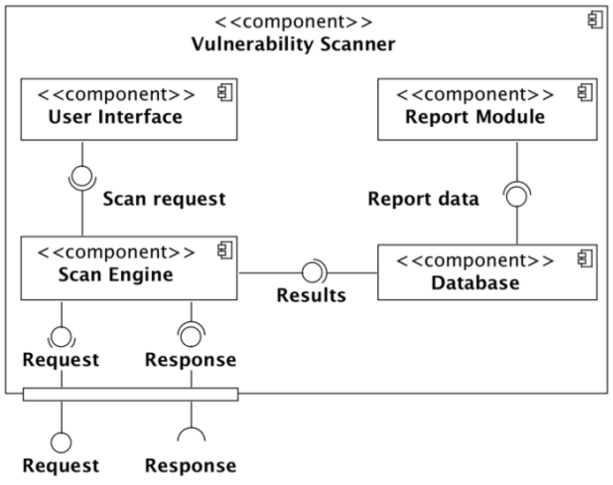


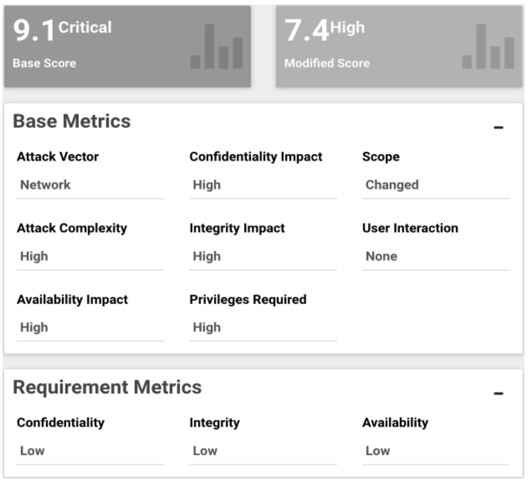
Figure 10 – Component diagram of vulnerability scanner [18]

The scan engine executes security checks according to its installed plugins, identifying system information and vulnerabilities while scan DB stores vulnerability information, scan results and other data used by scanner. Scan engine cal also execute web crawler functionalities.

On the other hand, the report module provides different levels of reports on the scan results, such as; detailed technical reports, summary reports, etc.

The user interface allows the user to operate the scanner. It may be either a graphical user interface or just a command line interface.

Software implementation and SQLI vulnerability test results. Algorithms to develop a VS for measuring vulnerability severity score and SQLI vulnerability testing are explained earlier. Fig. 11 shows a sample CVSS calculation.


9.1 Critical
Base Score

7.4 High
Modified Score

Base Metrics

Attack Vector	Confidentiality Impact	Scope
Network	High	Changed

Attack Complexity	Integrity Impact	User Interaction
High	High	None

Availability Impact	Privileges Required
High	High

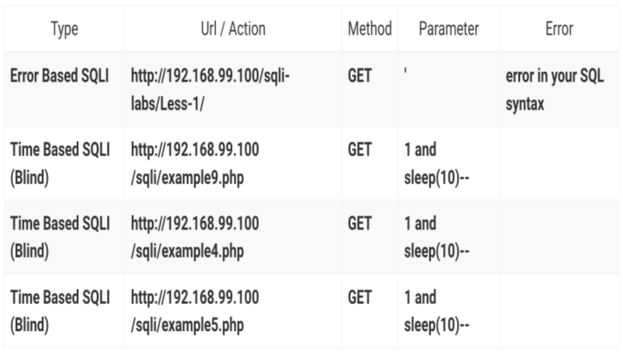
Requirement Metrics

Confidentiality	Integrity	Availability
Low	Low	Low

Figure 11 – An example of CVSS calculation

The prototype VS is using URL that points to the home page of the targeted website to discover forms and links to perform SQLI scans. When the URL is provided, and necessary forms and links are discovered, VS starts to perform necessary tests to find SQLI vulnerabilities in entire website.

Once the VS finishes all tests for the targeted website, it is ready to report found vulnerabilities. If there is SQLI vulnerability found in the website, then it is reported as in the Fig. 12.



Type	Url / Action	Method	Parameter	Error
Error Based SQLI	http://192.168.99.100/sqli-labs/Less-1/	GET	'	error in your SQL syntax
Time Based SQLI (Blind)	http://192.168.99.100 /sqli/example9.php	GET	1 and sleep(10)--	
Time Based SQLI (Blind)	http://192.168.99.100 /sqli/example4.php	GET	1 and sleep(10)--	
Time Based SQLI (Blind)	http://192.168.99.100 /sqli/example5.php	GET	1 and sleep(10)--	

Figure 12 – SQLI vulnerability test result shows found vulnerabilities

This result contains information about the type of detection algorithm that is used to find the vulnerability, the related URL, used HTTP request method, the injected parameter, and an error if the vulnerability was found using error-based SQLI vulnerability detection algorithm. This information will be helpful to re-produce the error or to find the vulnerability manually.

Conclusions and Future Work. In this paper, the fundamental security model so-called CIA triad is analyzed, web application vulnerabilities are classified to have a better view on the research area and SQLI vulnerability was analyzed. SQLI effects on the CIA triad are discussed. SQLI attack execution and its countermeasures are explained. Moreover, web application frameworks' solutions to the SQLI vulnerability are compared. BFS algorithm for a web crawler is explained. Furthermore, error-based and time-based SQLI vulnerability detection algorithms are introduced and CVSS is overviewed. Proposed algorithms are used to develop a VS and test results of the VS is represented.

Overall, this VS prototype can be used by organizations to find SQLI vulnerabilities in the website faster than manual tests that have to be done by penetration testers or ethical hackers.

Our future work concerns final implementation of the proposed VS, as well as improving algorithms for SQLI vulnerability detection to make it more reliable.

References

1. Madarie R. Hackers' Motivations: Testing Schwartz's Theory of Motivational Types of Values in a Sample of Hackers. *International Journal of Cyber Criminology,* 2017, vol.11, issue 1, pp. 78 – 97.
2. Rhodes-Ousley M. Information Security: The Complete Reference – *2nd* ed., 2013. pp. 85 – 87.
3. Meshram B.B., Savita B. C. Classification of Web Application Vulnerabilities. *International Journal of Engineering Science and Innovative Technology (IJESIT),* March 2013, vol.2, issue 2, pp. 226 – 234.
4. Zhou R., Hansen E. A. Breadth-First Heuristic Search. *Journal Artificial Intelligence,* April 2006, vol. 170, issue 4 – 5, pp. 701 – 709.
5. *Common Vulnerability Scoring System v3.0 Specification Document.* Available at: https://www.first.org/cvss/specification-document (accessed 11.05.2018).
6. *SQL Injection.* Available at: https://www.owasp.org/index.php/SQL_Injection (accessed 11.05.2018).
7. *Top 10 2017 – Injection Flaws.* Available at: https://www.owasp.org/index.php/Top_10_2007-Injection_Flaws (accessed 11.05.2018).
8. *Vulnerability Search Page.* Available at: https://nvd.nist.gov/vuln/search (accessed 11.05.2018).
9. *SQL Injection Prevention Cheat Sheet.* Available at: https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet (accessed 11.05.2018).
10. *Security Considerations (Entity Framework)*. Available at: https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ef/security-considerations (accessed 11.05.2018).
11. *CodeIgniter User Guide.* Available at: https://www.codeigniter.com/user_guide/general/security.html (accessed 11.05.2018).
12. *Laravel Documentation.* Available at: https://laravel.com/docs/5.6/eloquent (accessed 11.05.2018).
13. *A Pure Node.js JavaScript Client Implementing the MySQL Protocol.* Available at: https://github.com/mysqljs/mysql#escaping-query-values (accessed 11.05.2018).
14. *Phalcon Documentation: Database Abstraction Layer.* Available at: http://phalcon-php-framework-documentation.readthedocs.io/en/latest/reference/db.html (accessed 11.05.2018).
15. *Rail Guides: Securing Rail Applications.* Available at: http://guides.rubyonrails.org/security.html (accessed 11.05.2018).
16. *Spring Framework v.5 Documents: Data Access.* Available at: https://docs.spring.io/spring/docs/current/spring-framework-reference/data-access.html (accessed 11.05.2018).
17. *Zend Framework Documentation: Zend-DB.* Available at: https://docs.zendframework.com/zend-db/sql/ (accessed 11.05.2018).
18. *An Overview of Vulnerability Scanners.* Available at: https://www.infosec.gov.hk/english/technical/files/vulnerability.pdf (accessed 11.05.2018)

References (transliterated)

1. Madarie R. Hackers' Motivations: Testing Schwartz's Theory of Motivational Types of Values in a Sample of Hackers. *International Journal of Cyber Criminology,* 2017, vol.11, issue 1, pp. 78 – 97.
2. Rhodes-Ousley M. Information Security: The Complete Reference – *2nd* ed., 2013. pp. 85 – 87.
3. Meshram B.B., Savita B. C. Classification of Web Application Vulnerabilities. *International Journal of Engineering Science and Innovative Technology (IJESIT),* March 2013, vol.2, issue 2, pp. 226 – 234.
4. Zhou R., Hansen E. A. Breadth-First Heuristic Search. *Journal Artificial Intelligence,* April 2006, vol. 170, issue 4 – 5, pp. 701 – 709.
5. *Common Vulnerability Scoring System v3.0 Specification Document.* Available at: https://www.first.org/cvss/specification-document (accessed 11.05.2018).
6. *SQL Injection.* Available at: https://www.owasp.org/index.php/SQL_Injection (accessed 11.05.2018).
7. *Top 10 2017 – Injection Flaws.* Available at: https://www.owasp.org/index.php/Top_10_2007-Injection_Flaws (accessed 11.05.2018).
8. *Vulnerability Search Page.* Available at: https://nvd.nist.gov/vuln/search (accessed 11.05.2018).
9. *SQL Injection Prevention Cheat Sheet.* Available at: https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet (accessed 11.05.2018).

10. *Security Considerations (Entity Framework).* Available at: https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ef/security-considerations (accessed 11.05.2018).
11. *CodeIgniter User Guide.* Available at: https://www.codeigniter.com/user_guide/general/security.html (accessed 11.05.2018).
12. *Laravel Documentation.* Available at: https://laravel.com/docs/5.6/eloquent (accessed 11.05.2018).
13. *A Pure Node.js JavaScript Client Implementing the MySQL Protocol.* Available at: https://github.com/mysqljs/mysql#escaping-query-values (accessed 11.05.2018).
14. *Phalcon Documentation: Database Abstraction Layer.* Available at: http://phalcon-php-framework-documentation.readthedocs.io/en/latest/reference/db.html (accessed 11.05.2018).
15. *Rail Guides: Securing Rail Applications.* Available at: http://guides.rubyonrails.org/security.html (accessed 11.05.2018).
16. *Spring Framework v.5 Documents: Data Access.* Available at: https://docs.spring.io/spring/docs/current/spring-framework-reference/data-access.html (accessed 11.05.2018).
17. *Zend Framework Documentation: Zend-DB.* Available at: https://docs.zendframework.com/zend-db/sql/ (accessed 11.05.2018).
18. *An Overview of Vulnerability Scanners.* Available at: https://www.infosec.gov.hk/english/technical/files/vulnerability.pdf (accessed 11.05.2018).

Received 12.05.2018

Відомості про авторів / Сведения об авторах / About the Authors

Арслан Берк (Арслан Берк, Arslan Berk) – Національний технічний університет «Харківський політехнічний інститут», студент; м. Харків, Україна; ORCID: https://orcid.org/0000-0002-9493-0430; e-mail: berk.arslan93@gmail.com

Гамзаєв Рустам Олександрович (Гамзаев Рустам Александрович, Gamzayev Rustam Olexandrovich) – кандидат технічних наук, доцент, Національний технічний університет «Харківський політехнічний інститут», доцент кафедри програмної інженерії та інформаційних технологій управління; м. Харків, Україна; ORCID: https://orcid.org/0000-0002-2713-5664; e-mail: rustam.gamzayev@gmail.com

Ертугрул Каракуха (Эртугрул Каракуха, Ertuğrul Karaçuha) – професор, доктор, декан Інституту інформатики Стамбульського технічного університету (Istanbul Technical University), м. Стамбул, Турція; ORCID: https://orcid.org/0000-0002-7555-8952; e-mail: ertugrulkaracuha@gmail.com

Ткачук Микола Вячеславович (Ткачук Николай Вячеславович, Tkachuk Mykola Vyacheslavovich) – доктор технічних наук, професор, Національний технічний університет «Харківський політехнічний інститут», професор кафедри програмної інженерії та інформаційних технологій управління; Харківський національний університет імені В.Н. Каразіна, професор кафедри моделювання систем і технологій, м. Харків, Україна; ORCID: https://orcid.org/0000-0003-0852-1081; e-mail: tka.mobile@gmail.com