IJARSCT
ISSN (Online) 2581-9429
IJARSCT
Impact Factor: 6.252
International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)
Volume 2, Issue 1, August 2022

# Review Paper on a Study on SQL Attacks and Defense

**Bhuvana[1], Bindu[2], Chandan H[3], Brijesh Reddy KH[4], Mr. Pradeep V[5]**
Assistance Professor, Department of Information Science and Engineering[1]
Students, Department of Information Science and Engineering[2,3,4,5]
Alvas Institute of Engineering and Technology, Mijar, Moodbidri, Karnataka, India

**Abstract:** *In today's era, every person is utilizing websites and so many different web applications for online administrations, for example booking of railway tickets, movie ticketing, shopping, communication, and so forth. These websites consist sensitive and confidential information. With the linearity of web applications in the last decade, the unconstructive crash of security has also matured either. SQL injection attack is one such attack where the anonymous user can append SQL code to the input query. This research paper starts with developing criteria for a systematic literature review based on research questions, quality assessment, and data samples. The paper presents various SQL injection techniques with their intended attacks. Further studies explore different techniques to prevent attacks. Existing vulnerabilities of Web systems threaten the regular work of information systems. The most common Web system vulnerability is SQL injection. There are known approaches to protect Web applications against SQL injection attacks in the article. To improve the Web software security is developed defense mechanism that protects Web resources from SQL injection performing. To implement this software it is used PHP, JavaScript, and formal language theory known as regular expressions. As a result, it is received a software tool that allows protecting Web software from SQL injection vulnerability. The developed software tool allows users to protect their Web applications from an attack using SQL.*

**Keywords:** SQL Attacks.

## I. INTRODUCTION

**SQL injection** is a code injection technique used to attack data-driven applications, in which malicious SQL statements are inserted into an entry field for execution (e.g. to dump the database contents to the attacker).SQL injection must exploit a security vulnerability in an application's software, for example, when user input is either incorrectly filtered for string literal escape characters embedded in SQL statements or user input is not strongly typed and unexpectedly executed. SQL injection is mostly known as an attack vector for websites but can be used to attack any type of SQL database.

SQL injection attacks allow attackers to spoof identity, tamper with existing data, cause repudiation issues such as voiding transactions or changing balances, allow the complete disclosure of all data on the system, destroy the data or make it otherwise unavailable, and become administrators of the database server.

In a 2012 study, it was observed that the average web application received four attack campaigns per month, and retailers received twice as many attacks as other industries.

Since the dawn of web programming, companies started putting their databases on the Internet for public access. These databases sometimes contained confidential and valuable information which were good targets of attack. SQL injection attacks (SQLIA) are among the most common database attacks which try to access the sensitive data directly. They work by injecting malicious SQL codes through the web application and cause unexpected behavior from the database. The 2002 Computer Security Institute and FBI revealed that on a yearly basis, over half of all database experience at least one security breach and an average episode results in close to $4 million in losses [4]. We have presented six SQL injection prevention techniques in this paper which will cover a wide range of SQL injection attacks. A combination of these prevention techniques may lead to a more secure and reliable database system.

### 1.1 What is SQL injection (SQLi)?

SQL injection is a web security vulnerability that allows an attacker to interfere with the queries that an application makes to its database. It generally allows an attacker to view data that they are not normally able to retrieve. This might include data belonging to other users, or any other data that the application itself can access. In many cases, an attacker can modify or delete this data, causing persistent changes to the application's content or behavior.

In some situations, an attacker can escalate an SQL injection attack to compromise the underlying server or other back-end infrastructure or perform a denial-of-service attack.

### 1.2 How and Why Is an SQL Injection Attack Performed

To make an SQL Injection attack, an attacker must first find vulnerable user inputs within the web page or web application. A web page or web application that has an SQL Injection vulnerability uses such user input directly in an SQL query. The attacker can create input content. Such content is often called a malicious payload and is the key part of the attack. After the attacker sends this content, malicious SQL commands are executed in the database.

SQL is a query language that was designed to manage data stored in relational databases. You can use it to access, modify, and delete data. Many web applications and websites store all the data in SQL databases. In some cases, you can also use SQL commands to run operating system commands. Therefore, a successful SQL Injection attack can have very serious consequences.

- Attackers can use SQL Injections to find the credentials of other users in the database. They can then impersonate these users. The impersonated user may be a database administrator with all database privileges.
- SQL lets you select and output data from the database. An SQL Injection vulnerability could allow the attacker to gain complete access to all data in a database server.
- SQL also lets you alter data in a database and add new data. For example, in a financial application, an attacker could use SQL Injection to alter balances, void transactions, or transfer money to their account.
- You can use SQL to delete records from a database, even drop tables. Even if the administrator makes database backups, deletion of data could affect application availability until the database is restored. Also, backups may not cover the most recent data.
- In some database servers, you can access the operating system using the database server. This may be intentional or accidental. In such a case, an attacker could use SQL Injection as the initial vector and then attack the internal network behind a firewall.

### 1.3 Types of SQL Injections Attacks

SQL injections typically fall under three categories: In-band SQLi (Classic), Inferential SQLi (Blind), and Out-of-band SQLi. You can classify SQL injections types based on the methods they use to access backend data and their damage potential.

1. **In-band SQLi**: The attacker uses the same channel of communication to launch their attacks and to gather their results. In-band SQLi's simplicity and efficiency make it one of the most common types of SQLi attacks. There are two sub-variations of this method:
2. **Error-based SQLi**—the attacker performs actions that cause the database to produce error messages. The attacker can potentially use the data provided by these error messages to gather information about the structure of the database. Error-based SQLi is an in-band SQL Injection technique that relies on error messages thrown by the database server to obtain information about the structure of the database. In some cases, error-based SQL injection alone is enough for an attacker to enumerate an entire database. While errors are very useful during the development phase of a web application, they should be disabled on a live site or logged to a file with restricted access instead.
3. **Union-based SQLi**—this technique takes advantage of the UNION SQL operator, which fuses multiple select statements generated by the database to get a single HTTP response. This response may contain data that can be leveraged by the attacker. Union-based SQLi is an in-band SQL injection technique that leverages the UNION SQL operator to combine the results of two or more SELECT statements into a single result which is then returned as part of the HTTP response.

IJARSCT
ISSN (Online) 2581-9429
International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)
Volume 2, Issue 1, August 2022
IJARSCT
Impact Factor: 6.252

4. **Inferential (Blind) SQLi:** The attacker sends data payloads to the server and observes the response and behavior of the server to learn more about its structure. This method is called blind SQLi because the data is not transferred from the website database to the attacker, thus the attacker cannot see information about the attack in-band. Blind SQL injections rely on the response and behavioral patterns of the server so they are typically slower to execute but maybe just as harmful. Blind SQL injections can be classified as follows:
   a. **Boolean**—that attacker sends a SQL query to the database prompting the application to return a result. The result will vary depending on whether the query is true or false. Based on the result, the information within the HTTP response will modify or stay unchanged. The attacker can then work out if the message generated a true or false result. Boolean-based SQL Injection is an inferential SQL Injection technique that relies on sending an SQL query to the database which forces the application to return a different result depending on whether the query returns a TRUE or FALSE result. Depending on the result, the content within the HTTP response will change, or remain the same. This allows an attacker to infer if the payload used returned true or false, even though no data from the database is returned. This attack is typically slow (especially on large databases) since an attacker would need to enumerate a database, character by character.
   b. **Time-based**—attacker sends a SQL query to the database, which makes the database wait (for a period in seconds) before it can react. The attacker can see from the time the database takes to respond, whether a query is true or false. Based on the result, an HTTP response will be generated instantly or after a waiting period. The attacker can thus work out if the message they used returned true or false, without relying on data from the database. Time-based SQL Injection is an inferential SQL Injection technique that relies on sending an SQL query to the database which forces the database to wait for a specified amount of time (in seconds) before responding. The response time will indicate to the attacker whether the result of the query is TRUE or FALSE. Depending on the result, an HTTP response will be returned with a delay or returned immediately. This allows an attacker to infer if the payload used returned true or false, even though no data from the database is returned. This attack is typically slow (especially on large databases) since an attacker would need to enumerate a database character by character.
5. **Out-of-band SQLi:** The attacker can only carry out this form of attack when certain features are enabled on the database server used by the web application. This form of attack is primarily used as an alternative to the in-band and inferential SQLi techniques. Out-of-band SQLi is performed when the attacker can’t use the same channel to launch the attack and gather information, or when a server is too slow or unstable for these actions to be performed. These techniques count on the capacity of the server to create DNS or HTTP requests to transfer data to an attacker.

### 1.4 The Escape Character is not Filtered Correctly

This form of injection attack occurs when the user's input is not filtered for escape characters, and it is passed to an SQL statement. This will result in the end user of the application manipulating the statements on the database. For example, the following line of code demonstrates this vulnerability: Statement: = "SELECT * FROM users WHERE name ='" + userName + ";" The purpose of this code is to extract a particular user from its user table, but if the user name is forged by a malicious user in a specific way, the operation performed by this statement may not be just what the author of the code expects. For example, set the username variable to: 'a' or 't' = 't, when the original statement changes: SELECT * FROM users WHERE name ='a' OR 't'='t'; If this code is used in an authentication process, this example can force the selection of a legitimate username, because assigning 't'='t is always correct. On some SQL servers, such as in SQL Server, any SQL command can be injected in this way, including executing multiple statements. The value of username in the following statement will result in the deletion of the "users" table and the selection of all data from the "data" table (which in fact reveals the information of each user). a'; DROP TABLE users; SELECT * FROM data WHERE name LIKE'% This makes the final SQL statement look like the following: SELECT * FROM users WHERE name ='a'; DROP TABLE users; SELECT * FROM DATA WHERE name LIKE'%'; Other SQL executions do not take executing multiple commands in the same query as a security measure. This prevents an attacker from injecting a completely independent query, but doesn’t prevent an attacker from modifying the query.

1. **Incorrect type handling:** This form of attack occurs if a user-supplied field is not of a strong type, or if type coercion is not enforced. When a numeric field is used in an SQL statement, this attack occurs if the programmer does not check the validity of user input (whether it is numeric or not). For example: Statement: = "SELECT * FROM data WHERE id = + a_variable +"; As can be seen from this statement, the author expects a variable to be a number related to the "id" field. However, if the end user chooses a string, the need for escape characters is bypassed. For example, set a variable to: 1; DROP TABLE users, which will delete the "users" table from the database, and the SQL statement becomes: SELECT * FROM DATA WHERE id = 1; DROP TABLE users;
2. **Vulnerabilities in database servers :** Sometimes, there are vulnerabilities in database server software, such as mysql real_ escape _string() function vulnerability in MYSQL server. This vulnerability allows an attacker to perform a successful SQL injection attack based on incorrect unified character encoding.
3. **Blind SQL Injection Attack:** When a Web application is vulnerable to attack and the result is not visible to the attacker, a so-called blind SQL injection attack occurs. Vulnerable pages may not display data, but display different content based on the results of logical statements injected into legitimate statements. This attack is time-consuming because a new statement must be carefully constructed for each byte acquired. But once the location of the vulnerability and the location of the target information are established, a tool called Absinthe can automate this attack.
4. **Conditional response:** Note that there is an SQL injection that forces the database to compute the value of a logical statement on a common application screen: SELECT booktitle FROM booklist WHERE bookId ='00k14cd'AND 1 = 1 This leads to a standard face, while the statement SELECT booktitle FROM booklist WHERE bookId ='00k14cd'AND 1 = 2 When a page is vulnerable to SQL injection attacks, it may give a different result. Such a single injection will prove that blind SQL injection is possible, and it will enable an attacker to design statements that can judge authenticity based on the content of a field in another table.
5. **F. Conditional errors:** If the WHERE statement is true, this type of blind SQL injection forces the database to judge a statement that causes an error, resulting in an SQL error. For example: SELECT 1/0 FROM users WHERE username='Ralph'. Obviously, if Ralph exists, dividing by zero will lead to errors. G. Time Delay Time delay is a kind of blind SQL injection. According to the logic injected, it can cause the SQL engine to execute a long queue or a time delay statement. An attacker can 177 measure the page load time to determine whether the injected statement is true or not.

## II. SQL BLOCK IMPLEMENTATION

In order to check the authorization of the executing SQL statement, we need a way to get it. This can be done by implementing a proxy driver which is simply another database connectivity driver. This proxy driver calls another database driver as a client instead of passing the database query to the database server directly. This proxy driver also hides the error messages so the attacker will not be able to benefit from the information about the database schema contained in these error messages. After normalizing the SQL statement, the proxy driver will search the ready list to see if that particular normalized SQL statement exists or not. In case of existence, if the variables are within expected bounds this SQL execution is allowed and otherwise not.

## III. AMNESIA

In this part we will introduce a tool, AMNESIA, which implements a new technique for detecting and preventing SQL injection attacks. This technique detects illegal queries before being executed on the database using a model-based approach. This technique consists of a static part in which it automatically builds a model of legal queries using program analysis, and also a dynamic part in which it inspect and checks the dynamically-generated queries against the statically built model using runtime monitoring. Queries violating the model are potential SQLIAs and are prevented from execution. The technique is consisted of four steps which we will go through respectively.

**Step 1:** Identifying the hotspots Hotspots are some particular places in the application program which generate SQL queries to the database. In this step a simple scanning is performed to identify these points in the application code .

**Step 2:** Build SQL Query Model In this step a SQL-query model is built for each identified hotspot in order to compute the possible values of the query string passed to the database. Using the Java String Analysis (JSA) library a flow graph

is constructed which abstracts away the control floe of the program represents the operations performed on string variables. This technique analyzes the flow graph for each desired string and simulates the string manipulation operations that are performed on the string resulting in a NonDeterministic Finite Automata (NDFA) which represents all the possible values the considered string can take. A depth first traversal on the NDFA for the hotspot is performed in order to build the SQL-query model for that particular hotspot. Variable strings are presented using symbol ù e.g. the value of password .

**Step 3:** Instrument Application In this step, we instrument the application in such a way that, there will be a call to the monitor before the call to the database, so the queries will be checked at runtime. By the time of calling the monitor, a string that contains the query and also a unique identifier for the hotspot are passed to it as its parameters. This unique identifier is used by the runtime monitor in order to relate the hotspot with the specific SQL-query model generated for that point.

**Step 4:** Runtime Monitoring While executing normally when an application reaches a hotspot, the query will be sent to the monitor where it is parsed to a sequence of tokens according to SQL syntax. Empty string constants are identified by their syntactic position and are presented using İ. The important issue about this technique is that, it interprets the query the same way as the database does according to the SQL grammar, therefore it doesn't result in false positives and problems with user input caused by a simple keyword matching. After parsing the query, the runtime monitor checks it against the SQL query model associated with the hotspot from which the monitor has been called. To check whether a query is compliant with the model, the runtime monitor can check whether the NDFA accepts the query or not. In case of acceptance the monitor lets the query to be executed, otherwise considers it as an SQLIA. 4. Prepared Statements SQL (Structured Query Language) is a standard and common language which is designed for modifying and querying data and managing database.SQL statements are queries which are created with a standardized language for defining and manipulating data in a database that returns a single result set from a relational database. If theattacker can change the logic of the SQL statement by inserting SQL characters and keywords into SQL statement's input it is shown that SQL injection vulnerability exists. This vulnerability permits the attacker to modify SQL statement's structure and performs the SQL injection attack. A brief explanation about SQL structure will be presented which will be followed by its related vulnerabilities.

## III. SQL INJECTION AND XSS DEFENSES

Today, XSS and SQL Injection attacks are highly focused on filtering. This makes a certain amount of sense—both attack models are the same, though the targets differ.

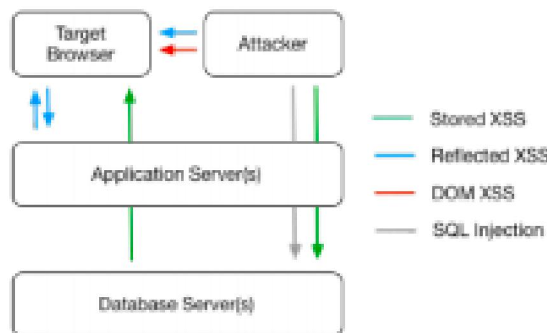



**Figure 1**: *A*ttack models of various XSS and SQL injection attacks.

Stored XSS attacks involve injecting content through the application server into the underlying data store, where the injected script is eventually sent to the target browser. Reflected XSS attacks are usually instigated via some external program, like an email client, chat program, or SMS client that coerces the user to click on a specially-crafted URL with a script payload, which is reflected from an application server back to the client with the trust properties of the reflecting server. DOM XSS attacks inject a script payload into the browser DOM, where it is extracted and executed by the browser. Finally, SQL Injection attacks send specially-crafted data to the application server, which then forwards that data to the underlying data store where that data is interpreted as valid SQL directives. Note that the first stages of stored XSS and SQL Injection attacks are the same - injecting content into the data store.

All mainstream defenses against these attacks use filtering in one form or another but in very particular orders. OWASP goes into detail with respect to what kind of filtering happens in what order to defend against which types of attacks—suffice to say that it's complex, and you see developers messing the order up all the time (Ever see "&" in a dynamic section of a web site? This is why). Escaping content is straightforward, even if the ways you need to combine the escaping isn't, but it ends up being pretty obtuse. CSRF defenses are much more interesting and more difficult to understand.

### 3.1 CSRF Defenses

First, in order for any of these defenses to work, you can't have any XSS vulnerabilities. XSS isn't required to make CSRF work, but it does give attackers the ability to circumvent every CSRF defense. The simplest way to detect CSRF attacks is to just confirm that the target and origin URLs are the same. If they are, there's no cross-site activity—nothing to see here, go ahead and fulfill the request. We do this by checking the origin or referrer HTTP header values (or both) and comparing them to the target HTTP header value. While this approach can give you insight into the validity of a request, these headers frequently don't exist and are modifiable in a variety of ways. This isn't really sufficient to defend against CSRF, and may not even be necessary, as it's so easy to alter.

The other three CSRF specific defenses include synchronizer tokens, double submit cookies, and encrypted tokens.

Synchronizer tokens are randomly generated values that an application server inserts into sensitive elements as a hidden field or within a URL (think forms, images, and any other tag that will initiate an HTTP request). The server will then extract this token and compare it to the token saved in the user session, rejecting the request if the two don't match. Usually, this is maintained for an entire user session, though it can be implemented at the request level as well (though this approach leads to problems, like using the back button or bookmarking a site).

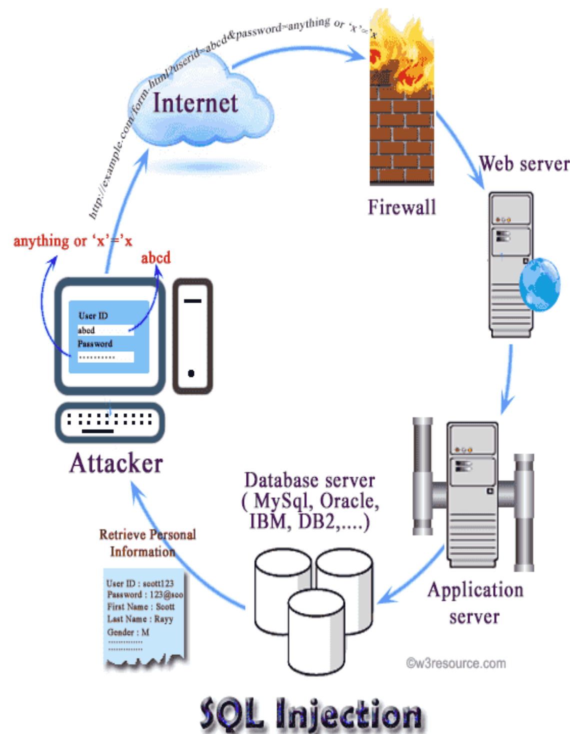
Furthermore, including the token in the URL can lead to token leaking. URLs are frequently saved in logs and browser histories, where they can be accessed by attackers and used to circumvent CSRF protections.

Double submit cookies take advantage of the fact that attackers can't access cookies when crafting CSRF requests due to same-origin policies baked into browsers. Basically, a server submits a random value as a cookie separate from the session ID cookie. The client then inserts this random value into a hidden field (or similar) where the random value can be validated by the receiving server. This way, the server need not maintain any state; the cookie and the hidden field are delivered by the client on resource requests, and the server has access to both pieces of data, allowing it to validate that they are in fact both equal. Keep in mind though, cookies aren't always available, and with cookies, you can run into subdomain issues as well.

Encrypted tokens are the final CSRF defense in use today. This is similar to the double submit cookie approach, in that some combination of a user ID, a timestamp, and a nonce are inserted into a data structure that is then encrypted by the server, using a stored key. The token is submitted to the client in a hidden field in generated HTML. The client then uses this encrypted value with any HTTP requests sent to the server, including the value in either HTTP attributes or hidden fields. When the server receives this token, it will decrypt the token with its stored key and compare the extracted values, using the timestamp to protect against replay attacks.

SQL Injection Prevention There are several solutions to solve this problem. One common solution is to discrete the SQL structure from the SQL input by applying prepared statements. Prepared statements specify the SQL structure explicitly. Since replacing vulnerable SQL statements with prepared statements manually is time consuming, boring and error-prone, by using a prepared statement replacement algorithm (PSRAlgorithm) the task will be performed automatically. This algorithm collects information from source code which is containing SQL injection vulnerability and creates secure prepared statement code with respect to functional integrity. It means that it analyzes source code including SQL Injection Vulnerabilities and creates a specific recommended code structure which includes prepared statements and makes apart the SQL structure from SQL statement's input. For each string object applied to generate the SQL statement the PSR Algorithm generates an additional string object and an assistant vector which are used for finding any SQL input in the original string objects. Therefore the SQL injection vulnerability can be eliminated from vulnerable SQL statements by converting them with secure prepared statements[18]. The logic of this algorithm is not restricted to any specific language so it can be extended to fit the syntax of any language.

IJARSCT
ISSN (Online) 2581-9429
International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)
IJARSCT
Impact Factor: 6.252
Volume 2, Issue 1, August 2022




By specifying the structure of the statement and defining the limitation for inputs a prepared statement is prepared. Then the SQL statement structure with the limited or bind variable is sent to the database and the database compiles and saves the structure of the statement for implementing with input variables in the future. Therefore the statement structure cannot be altered by input parameters when prepared statements are generated because the statement structure set clearly before the runtime. Consequently the risk of the SQL injection vulnerability will be decreased. Consider the following prepared statement.

## IV. CONCLUSION

This paper is a survey of different SQL Injection prevention techniques. Generally, the presented techniques focus on standardization of SQL statements in a way that prevent attackers from gaining unauthorized access to an organization's database. First we started by studying different attacks which were more common and advanced. The research continued with organization and classification of different prevention techniques which we found and finally a discussion with a summary of the best SQL injection prevention techniques were presented. The paper discusses the appropriate SQL prevention techniques for a given attack. We believe that each prevention technique cannot provide complete protection against SQL Injection Attacks but a combination of the presented mechanisms will cover a wide range of SQL injection attacks which will culminate in a more secure and reliable database which is protected against SQL Injection Attacks.SQL injection attackers are smarter and more comprehensive in finding vulnerable websites. There are some new methods of SQL attack. Hackers can use various tools to speed up the process of exploiting vulnerabilities. Let's take a look at the Asprox Trojan, which is spread mainly through a botnet that publishes mail. The whole process of its work can be described as follows: First, the Trojan is installed on the computer through spam sent by the controlled host. Then, the computer infected by the Trojan will download a binary code, and when it starts, it will use search. Index engine search uses Microsoft's ASP technology to build forms and vulnerable websites. The results of the search become a list of targets for SQL injection attacks. Then, the Trojan Horse will launch SQL injection attacks on these sites, making some sites under control and destruction. Users visiting these controlled and destroyed sites will be tricked into downloading a malicious piece of JavaScript code from another site. Finally, this code guides users to the third site, where there are more malware, such as Trojan horses stealing passwords.

## REFERENCES

[1]. "An Introduction to SQL Injection Attacks for Oracle Developers," [Online]. Available:www.integrigy.com/Integrigy_Oracle_SQL_Injection_Attacks [Accessed: Oct.02, 2010].

[2]. "SQL Injection" [Online] Available: http://projects.webappsec.org/w/page/13246963/SQL-Injection [Accessed: Oct 12,2010]

[3]. "Wikipedia," [Online]. Available: http://en.wikipedia.org/wiki [Accessed: Oct.5,2010].

[4]. "Prepared Statements" [Online] Available: http://www.owasp.org/index.php/SQL_Injection_Prevention [Accessed: Nov 12, 2010].

[5]. "Blind SQL Injection "KavinSpet White Paper. [Online] Available:

[6]. "Second Order Code Injection Attack" Gunter Ollmann [Online] Available:

[7]. "Web Application Attack Prevention for Tiered Internet Service "Susanta Nanda, Lap Chung Lam, Fourth Intenational Conference IEEE 2008.

[8]. 2017. Exploit DataBase [online]. https://www.exploit-db.com. (July 14, 2017).

[9]. 2017. Firebase - Google Inc. [online]. https://firebase.google.com/. (July 10, 2017).

[10]. 2017. Packet Storm [online]. https://goo.gl/X3v37z. (July 9, 2017).

[11]. 2017. SQL Map [online]. http://sqlmap.org/. (July 10, 2017).

[12]. 2017. WASP Top 10 [online]. https://www.owasp.org/index.php/Top_10_2017-Top_10. (July 10, 2017).

[13]. 2017. Wired - WannaCry [online]. https://www.wired.com/tag/wannacry/. (July11, 2017).

[14]. Gregory Buehrer, Bruce W Weide, and Paolo AG Sivilotti. 2005. Using parse treevalidation to prevent SQL injection attacks. In Proceedings of the 5th internationalworkshop on Software engineering and middleware. ACM, 106–113.

[15]. J. Choi, H. Kim, C. Choi, and P. Kim. 2011. Efficient Malicious Code Detection Us-ing N-Gram Analysis and SVM. In 2011 14th International Conference on Network-Based Information Systems. 618–621. https://doi.org/10.1109/NBiS.2011.104

[16]. William GJ Halfond and Alessandro Orso. 2005. AMNESIA: analysis and monitor-ing for NEutralizing SQL-injection attacks. In Proceedings of the 20th IEEE/ACMinternational Conference on Automated software engineering. ACM, 174–183.

[17]. R. Komiya, I. Paik, and M. Hisada. 2011. Classification of malicious web code by machine learning. In 2011 3rd International Conference on Awareness Science and Technology (iCAST). 406–411. https://doi.org/10.1109/ICAwST.2011.6163109

[18]. A. Makiou, Y. Begriche, and A. Serhrouchni. 2014. Improving Web Application Firewalls to detect advanced SQL injection attacks. In 2014 10th International Conference on Information