International Journal of Security and Its Applications
Vol.7, No.5 (2013), pp.53-66
http://dx.doi.org/10.14257/ijsia.2013.7.5.05

A Simple and Fast Technique for Detection and Prevention of SQL Injection Attacks (SQLIAs)

Z. Lashkaripour[1,] * and A. Ghaemi Bafghi[1]

[1]*Data and Communication Security Laboratory, Department of Computer, Faculty of Engineering, Ferdowsi University of Mashhad, Mashhad, Iran*
Zeinab.Lashkaripour@stu-mail.um.ac.ir (Z. Lashkaripour), ghaemib@um.ac.ir (A. Ghaemi Bafghi).
[*] *Corresponding author.* Tel.: +98 511 8763306; fax: *+98 511 8763306*.

Abstract

In SQLIA, attacker injects an input in the query in order to change the structure of the query intended by the programmer and therefore, gain access to the data in the underlying database. Due to the significance of the stored data, web application's security against SQLIA is vital. In this paper we propose a new technique based on static analysis and runtime validation for detection and prevention of SQLIAs. In this technique user inputs in SQL queries are removed and some information is gathered in order to make the detection easier and faster at runtime. Our experiments show that our proposed technique is fast, it has a low error rate and its detection rate is nearly 100%.

Keywords: *Web application; SQLIA; detection; prevention; static analysis; runtime validation; security*

1. Introduction

Nowadays web applications (applications with client/server model communication that are accessed via internet or intranet [1]) are widely used in various applications due to the accessibility and convenience they provide. This makes them a suitable target for attackers, so their security becomes necessary. These kinds of applications have different sorts of attacks. According to OWASP Top 10 in 2010 [2] and other related reports such as [3, 4]; SQLIA has the highest frequency among web application attacks. This shows the significance of securing web applications against it in order to protect the application and its data. Despite the significance of web application security less attention has been considered which can have the reasons given here. Web applications are written by developers that have less programming and security skills, some of the web applications are produced by the integration of works from several developers and therefore, it is not always possible to completely review and verify the code and finally, many site owners ask the developers to focus on functionality rather than security therefore, as a result we might have insufficient input validation [5]. SQLIA is the attempt of injecting data that part of it is treated as code and therefore, changes the semantic of the intended query. The result of this attack is unrestricted access to the database which is due to the reasons mentioned earlier.

In order to maintain the security of web applications against SQLIAs we have proposed a technique that is a combination of static analysis and runtime validation. This technique is an extension of [6] which would be explained in details later on. The remainder of the paper is organized as follows: Section 2 is about different types of SQLIAs with an example for each of them. Next section would introduce related works. In Section 4 our proposed technique

ISSN: 1738-9976 IJSIA
Copyright ⓒ 2013 SERSC

would be explained. Finally practical results of the technique are illustrated in Section 5 and conclusion in the last section.

2. SQLIA Types

SQLIAs have different types [7, 8] that we would briefly define them based on [7] (for more information, see the references mentioned earlier) and also give an example for each of them in this section.

In all of the examples in this section we have used a query with three inputs which is given below:

*Query = "SELECT * FROM Accounts WHERE user=' " + username + " ' AND pass=' " + password + " ' AND eid=" + id ;*

2.1. Tautology

In tautology the attacker tries to use a field in the WHERE clause to inject and turn the condition into a tautology which is always true. The simplest form of tautology is given in the example below.

Example: attacker inserts “' or 1=1 --” into the user field and nothing for the other fields so the result is:

*SELECT * FROM Accounts WHERE user=' ' or 1=1-- ' AND pass=' ' AND eid=*

The result would be all the data in Accounts table because the condition of the WHERE clause is always true.

2.2. Illegal/Logically Incorrect

In this kind of attack the attacker gathers some important information about the type and structure of the database. This information is obtained from error pages returned from default servers and can be used for further attacks.

Example: attacker inserts “convert(int,(SELECT TOP 1 name FROM sysobjects WHERE xtype='u'))” into the eid field and nothing for the rest of the fields so the result is:

*SELECT * FROM Accounts WHERE user=' ' AND pass=' ' AND eid=convert(int,(SELECT TOP 1 name FROM sysobjects WHERE xtype='u'))*

In this example the attacker attempts to convert the name of the first user defined table in the metadata table of the database to ‘int’. As you know this type conversion is not legal therefore, the result is an error which reveals some information that should not be shown.

2.3. Union

As it can be inferred from the name, the result of the attack is some data from the database which is the union of the main query and the injected one together. So in this type of attack the data returned from the query is changed.

Example: attacker inserts “' UNION SELECT * FROM Students --” into the user field and nothing for the other fields so the result is:

*SELECT * FROM Accounts WHERE user=' ' UNION SELECT * FROM Students -- ' AND pass=' ' AND eid=*

 Copyright ⓒ 2013 SERSC

The result of the first query in the example above is null and the second one returns all the data in Students table so the union of these two queries is the Students table.

2.4. Piggy Backed

In piggy backed the attacker attempts to inject an extra query in the main one so that beside the main query the injected one is also executed.

Example: attacker inserts "'; drop table Accounts --" into the user field and nothing for the two remaining fields so the result is:

*SELECT * FROM Accounts WHERE user=' '; drop table Accounts -- ' AND pass=' ' AND eid=*

The result of the above example is losing the credential information of the Accounts table because it would be dropped.

2.5. Blind Injection

As inferred from the name, the attacker is blind so he tries to attack the web application by asking true/false questions therefore, depending on the reply of the web application he can gain information about the database although no error message is shown.

Example: attacker inserts "user1' AND 1=1 --" into the user field and nothing for the rest of them so the result is:

*SELECT * FROM Accounts WHERE user='user1' AND 1=1-- ' AND pass=' 'AND eid=*

The injected part is always evaluated to true so if there is no login error message, the attacker realizes that the attack has passed and the user parameter is vulnerable to injection.

2.6. Timing Attacks

In timing attacks, attacker gains information depending on the delays of the database responses.

Example: attacker inserts "user1' AND ASCII(SUBSTRING((SELECT TOP 1 name FROM sysobjects),1,1)) > X WAITFOR DELAY '000:00:07' --" into the user field and nothing for the other fields so the result is:

*SELECT * FROM Accounts WHERE user='user1' AND ASCII(SUBSTRING((SELECT TOP 1 name FROM sysobjects),1,1)) > X WAITFOR DELAY '000:00:07' -- ' AND pass=' ' AND eid=*

In the above example the attacker is trying to find the first character of the first table by comparing its ASCII value with X. If there is a 7 second delay he realizes that the answer to his question is yes, so by continuing the process the name of the first table would be discovered (with similar attacks attacker can obtain other information about the database).

2.7. Alternate Encoding

In this type, the injected text is changed in order to evade detection by defensive coding practices and most of the automatic prevention techniques. Encodings such as hexadecimal, ASCII and Unicode character encoding can be used for attack strings.

Copyright ⓒ 2013 SERSC

Example: attacker inserts "user1'; exec(char(0x73687574646f776e)) --" into the user field and nothing for the rest so the result is:

*SELECT * FROM Accounts WHERE user='user1'; exec(char(0x73687574646f776e)) -- ' AND pass=' ' AND eid=*

In the above example char() function and ASCII hexadecimal encoding are used. The function gets an integer number as a parameter and returns a sample of that character. In this example the function will return "SHUTDOWN", so whenever the query is interpreted the SHUTDOWN command is executed.

2.8. Stored Procedure

This type of SQLI executes the stored procedures available at the underlying database. Many databases have built in stored procedures in addition to user defined stored procedures. The built in stored procedures are used for extending the functionality of the database and interacting with the operating system. Thus, once the attacker has identified the underlying database he tries to execute these built in stored procedures in order to exploit information.

Example: attacker inserts "'; exec xp_logininfo 'BUILTIN\Administrators'; --" into user field and nothing for the pass and eid fields:

*SELECT * FROM Accounts WHERE user=' '; exec xp_logininfo 'BUILTIN\Administrators'; -- ' AND pass=' ' AND eid=*

In this example the built in stored procedure "xp_logininfo" is executed in order to get the information about the BUILTIN\Administrators Windows group.

On the other hand the user defined stored procedures are coded by the programmer and therefore vulnerable. It should be mentioned that all of the SQLIAs can take place at the stored procedures of the underlying database by means of their parameters as well as the web application side. That means that stored procedures can be vulnerable to the same SQLIAs as the web application code.

Example: attacker inserts "user1" into user field and "'; SHUTDOWN; --" into the pass field and nothing for the eid:

```
CREATE PROCEDURE DBO.isAuthenticated
@userName varchar2, @pass varchar2, @pin int
AS
   EXEC("SELECT * FROM Accounts
   WHERE user='" + @username + "' and pass='" + @password +
            "' and eid=" +@id);
GO
```

The resulted query would be:

*SELECT * FROM Accounts WHERE user='user1' AND pass=' '; SHUTDOWN; -- ' AND eid=*

In the above example we have a piggyback attack where the injected part which is database shutdown is executed beside the first query.

 Copyright ⓒ 2013 SERSC

3. Related Work

In this section we have divided the related works into three groups: static analysis, dynamic analysis and combinational. Each of them has their own advantages and disadvantages that would be mentioned in this part.

3.1. Static Analysis

These techniques can be used in the application's development and debugging phases (before deployment) and also in protecting existing web applications therefore, they do not have any runtime overhead. They help developers to identify the weaknesses and vulnerabilities that invite attackers so as to reduce and/or remove them in order to make applications more reliable. Despite their advantages their shortcomings are: developer needs to manually alter the vulnerable parts, which is tedious and time consuming [6], not being successful in identifying stored procedure attacks [9] and not paying attention to dynamic queries because their structures are not specified till runtime.

SQL DOM [10] and Safe Query Objects [11] change the process of creating a query from an irregular concatenation process to a systematic process that uses a type checking API in order to make the database access secure and reliable. On the other hand they have disadvantages such as the need of learning a new API by the developer and being expensive for legacy codes [7].

Penetration testing tools such as MySQLInjector [12], V1p3R (Viper) [13] and Sania [14] also lye in the static group. At first these tools gather information from the web application and after that inject attacks according to the information gathered in order to analyze the application's response. V1p3R uses the stored patterns in its error pattern library and Sania uses SQL parse tree comparison for SQLIA detection while in MySQLInjector the output is the results of the attacks. Success in these tools depends on the completeness of the injected attacks and this is a shortcoming but, their advantage is identifying vulnerabilities without any modifications to the web application.

3.2. Dynamic Analysis

These kinds of techniques use a model for SQLIA detection. They generate the model at runtime and because of that they are called dynamic techniques. Due to runtime generation of the model they do pay attention to dynamic queries which are generated at runtime but on the other hand they have the overhead of generating the model at runtime.

SQLGuard [15] and CANDID [16] are based on the runtime comparison of the parse tree intended by the programmer with the runtime parse tree. So that whenever they do not match the query would not be sent to the database for execution and therefore SQLIA is prevented. The runtime comparison of parse trees has an overhead which is a disadvantage for them both. The advantage of SQLGuard is partially covering dynamic queries due to making the parse tree at runtime. On the other hand its shortcomings are not being capable of identifying stored procedure attacks [9] and the need for the developer to change the code. But the advantage of CANDID is no need for changing the code manually, but its disadvantage is partial (not complete) detection of different kinds of attacks [9].

3.3. Combinational

Combinational techniques have two phases: static analysis and dynamic analysis. Due to fulfilling part of the operations in the static phase there is no overhead at runtime for them and this is the benefit of these techniques. In the static phase first of all the hotspots are

Copyright ⓒ 2013 SERSC

identified, after that a model is created indicating all the valid queries that can be made at that hotspot. Finally at runtime, the runtime queries are examined to see whether they match their model or not. If not, the query would not be sent to the database for execution and therefore SQLIA is prevented. None of the techniques mentioned below are capable of identifying stored procedure attacks except [6] that can identify them partially, and because of generating the model at the first phase none of them pay total attention to dynamic queries.

AMNESIA [17] creates an NDFA for each hotspot. After that the web application is adjusted so that the call to the runtime monitor is added before the query execution. At runtime, the runtime query is compared against the static model and if the automaton does not accept the query, it would not be executed.

SQLCHECK [18] marks the input with a key. The query made with such an input is called augment query. In order to prevent SQLCIAs in these queries an augment grammar is generated therefore, only the queries that are parsed by this grammar are valid. Valid queries are then sent to the database without the keys for execution. The security of SQLCHECK dependents on the attacker not being able to discover the key, another shortcoming is the need to manually alter the code in order to insert the keys in SQL queries which has the problem of incompleteness [20].

In [19] the behavior of SQL queries is represented in the form of a SQL-graph which is produced by static code analysis. This graph is used so that there would be no need to modify the code of the web application, which makes it an advantage because it will spare money and time. Furthermore, in the static phase, a Finite State Automata (FSA) is generated for each hotspot. Since inspecting all the queries at runtime is time consuming [19] uses the SQL-graph so that only those SQL queries that are supersets of other queries in the graph are inspected and their static and dynamic SQL-FSMs are compatible. Another advantage of [19] is that it has used a parallel implementation to decrease runtime execution.

WASP [20] is based on positive tainting. Before sending queries to the database WASP performs automatic syntax aware validation. In other words the query is tokenized into a sequence containing SQL keywords, operators and literals. Then it checks that all of the tokens except the literals are made from trusted data. If all the tokens pass this check, the query is safe and can be executed by the database. The disadvantage of WASP is the need of specifying trusted external data sources because they are not hard coded in the application's code (if not specified false positive is generated).

In [6] the opinion of removing attribute values is used to detect and prevent SQLIAs. In order to detect, attribute values are removed from both of the static and dynamic queries and for comparison they are XORed. If the result of the comparison is equal to zero the query is safe for execution. Simplicity is the advantage of [6] but its disadvantages are doing unnecessary inspections at runtime which leads to overhead increase and also considering simple conditions where the operator is equality whereas other operators need to be considered.

4. Proposed Technique

As mentioned earlier our proposed technique works in two phases. Figure 1 shows them which the first phase is done statically and the second one at runtime. At the first phase we have an instrumentation module that gets as input the original web application and outputs the instrumented web application. The web application is used for generating query structures (models of valid queries) and gathering information that are all needed for the latter phase. The web application is changed according to our needs to result the instrumented web application. The second phase works with the instrumented web application obtained from the previous phase and contains a dynamic validation checker which would generate the

Copyright ⓒ 2013 SERSC

structures of the dynamic queries and compare them with their corresponding static models so that whenever identical, the query is allowed to be executed but if not, SQLIA is detected and the malicious query would be prevented in order to preserve the security of the web application and its underlying database. Each of these two phases would be explained in the following section.

4.1. Static Analysis

In static analysis we need to instrument the original web application to be able to approach to our goal. The instrumentation operation is done by means of our instrumentation module that contains a scanner and an analyzer. In this phase first of all, hotspots should be identified. As you know hotspots are those spots in a web application that have interaction with the underlying database. After that, we can access the query of that hotspot which is needed for generating the static model. These operations are done by the scanner. When the query is obtained, static analysis is done in order to generate the static model and also gather the information needed for runtime so as to simplify and speedup SQLI detection. For the static model that indicates the structure of a valid query, all user inputs surrounded by ‘’ need to be removed and besides that some information which contains the location of the inputs are gathered in order to be used later on. Finally at this phase calls to the dynamic validation checker are added before the execution of the queries so that whenever the runtime query does not contain any malicious input it is handed to the database for execution.

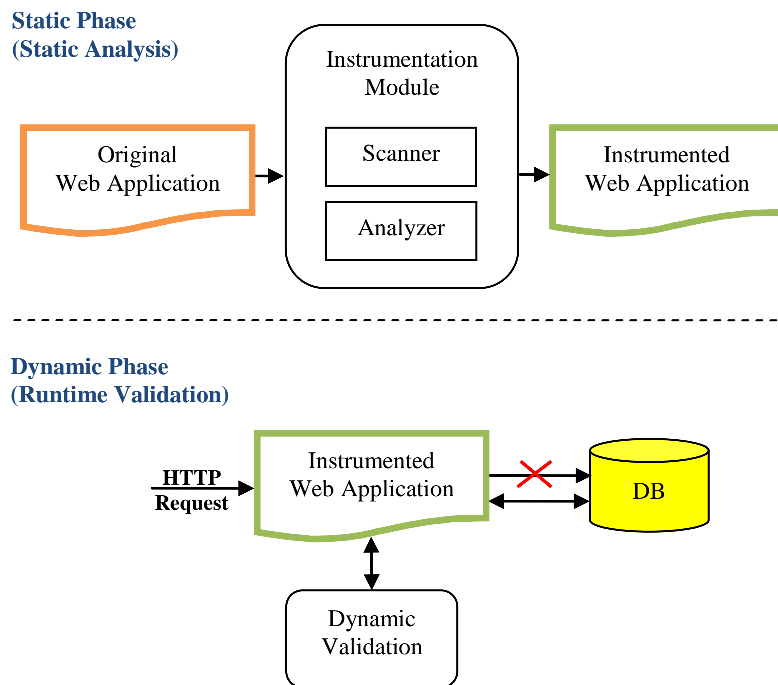

Figure 1. Proposed Technique Architecture

4.2. Runtime Validation

In the second phase named runtime validation, user inputs of the dynamic queries are removed according to the information gathered at the first phase by the dynamic validation checker in order to obtain the structure of the dynamic query. Dynamic validation checker

Copyright ⓒ 2013 SERSC

checks the location of the inputs based on the information gathered in the static phase. This inspection is done for each input in the query. At this point two situations might occur for malicious inputs: 1) in one of the checks mentioned earlier it would be detected because of the changes it has made to the query, or 2) would not be detected in these checks. In the first one whenever detected no further validation is taken place and because of being malicious the query would not be executed and therefore SQLIA is prevented. In the other condition we need to do a final check which is checking the equality of the static (obtained from the first phase) and dynamic (obtained from the second phase) structures of the related query. So when the two models in hand have the same structure the query is valid and can be sent to the database for execution but if not, the query is malicious and should not be executed, which prevents SQLIAs.

We would give an example to demonstrate our technique. Consider the query below as the query intended by the programmer:

*Original Query:"SELECT * FROM Accounts WHERE user=' "+ username +" 'AND pass=' "+ password + " '"*

where the inputs submitted by the user are:

user: user1' or 'sqli' like '%sq%' --

pass: null

As you can see the "user" input contains SQLI of type tautology that our technique should be able to detect it as follows.

The static query model after using the proposed technique and the dynamic query obtained by including the user inputs are:

*Static query model: “SELECT * FROM Accounts WHERE user=' ' AND pass=' ' ”*

*Dynamic query: “SELECT * FROM Accounts WHERE user='user1' or 'sqli' like '%sq%' -- ' AND pass=' ' ”*

At runtime our technique would go to the place of the inputs one by one based on the static information to check their locations and after that remove the input values. Therefore, in this example the first step is checking the location of the first input (user) and removing its value which is bolded. The outcome of this operation would be the dynamic query model given below:

*Dynamic query model: “SELECT * FROM Accounts WHERE user=' ' or 'sqli' like '%sq%' -- ' AND pass=' ' ”*

Up to now no problem exists but, when we want to repeat the same steps for the second input (pass) we would recognize that our input is not in its place due to the injection of the first input. Without further inspection our technique would identify that injection has taken place and would not let the query to be executed in order to prevent SQLIA and preserve the security of the web application and its underlying database.

For those injections that are not identified in the steps mentioned above, the final step which is checking the equality of the two query models in hand, would identify them. The example below shows such a situation. Consider the submitted inputs as:

Copyright ⓒ 2013 SERSC

user: user1

pass: '; exec xp_logininfo 'BUILTIN\Administrators'; --

In this example the "pass" input contains SQLI of type stored procedure and our technique will detect it as follows.

The static query model and also the dynamic query obtained by including the user inputs are:

*Static query model: "SELECT * FROM Accounts WHERE user=' ' AND pass=' ' "*

*Dynamic query: "SELECT * FROM Accounts WHERE user='user1' AND pass=' '; exec xp_logininfo 'BUILTIN\Administrators'; -- ' "*

At runtime our technique would go to the place of the inputs one by one based on the static information to check their locations and after that remove the input values specified in bold. Thus, the outcome of this operation which is the dynamic query model is given below:

*Dynamic query model: "SELECT * FROM Accounts WHERE user=' ' AND pass=' '; exec xp_logininfo 'BUILTIN\Administrators'; -- ' "*

Up to now no injection has been identified because, both of the inputs are in their own places and this is due to the fact that injection has taken place in the last input. Therefore, the last step which is checking the equality of the two query models (static and dynamic) would detect and prevent SQLI since they are not identical.

Static query model ≠ Dynamic query model

5. Experiment and Evaluation

In order to evaluate our technique and show our expectations in practice we used the test suite obtained from AMNESIA. For the input suite we have also used the ones from AMNESIA's test bed which contained both attack and legitimate (non attack) inputs for each application. We have two types of inputs which are string or number and two types of queries which are static or dynamic. The experiments are based on static queries containing number or string and the inputs related to dynamic queries are not considered due to obtaining the query structures statically. But, with extension the solution would be an effective method against SQLIAs that no matter what the query (static or dynamic) or the input type (string or number) is, it would be capable.

We implemented [6] and compared our proposed technique with it under the same condition. For this purpose we ran the experiment on a system which had a Core Duo 2.00GHz CPU with Windows XP Professional SP2 OS and 512MB RAM.

It should be mentioned that as part of our evaluation we have used the experiment results obtained by [7], [9] and [6] in order to be able to compare our proposed technique with dynamic and combinational techniques introduced in Sections 3.2 and 3.3. The further details are given in the Sections 5.1 through 5.3.

5.1. Detection and Prevention Rate Analysis

The experiment results are shown in Table 1 that we will discuss them briefly. In order to start the test of our technique we ran the first phase as outlined in Section 4 and then after getting the instrumented form of our web application we started the experiment. For calculating the overhead of the techniques we used the legitimate inputs to get an accurate

Copyright ⓒ 2013 SERSC

result because our technique stops whenever an injection occurs; therefore, the execution time for the attacks in our proposed technique would be less than what is shown in Table 1.

Table 1. Experiment Results

Subject	Technique	Legitimate Queries			Attack Queries	
		Attempt/Allowed	Timing Avg (ms)	Error Rate (%)	Attempt/Prevented	Detection Rate (%)
Employee Directory	[6]	290/320	0.0450	9.38	3707/3707	100
	Proposed	300/320	0.0341	6.25	3707/3707	100
Events	[6]	290/328	0.0453	11.59	3212/3220	99.75
	Proposed	300/328	0.0364	8.54	3212/3220	99.75
Classifieds	[6]	116/128	0.0432	9.38	3295/3295	100
	Proposed	120/128	0.0344	6.25	3295/3295	100

Legitimate inputs: As you can see our solution is faster than [6] and therefore can give better service to legitimate users. It is important to protect our web applications with solutions that have the least overhead which the ideal overhead is zero. Both techniques have errors and that is because the inputs contain numbers and numbers are not surrounded by ' ' therefore, both technique don't support them. Beside numeric variables there are inputs that although legitimate, contain ' and both of the technique would consider them as attack and prevent their execution. But, whenever the legitimate inputs use the value ' (escapes it with backslash) not the operator ', our technique would not prevent their execution and consider them as non attack inputs, resulting a lower error rate in comparison to the other technique.

Attack inputs: After using the techniques for protecting the web applications, the attacks were injected to them in order to compute the detection rate. Both techniques detected and prevented all of the attacks in two of the web applications resulting a 100% detection rate while in the other one all of the attacks except the ones related to numeric variables where detected and prevented therefore, the detection rate was less than 100%.

5.2. Comparison of Techniques with Respect to SQLIA Types

As it is shown in Table 2 (AMNESIA, SQLCheck and SQLGuard are from the results obtained by [7], CANDID is from [9] and RemovingAttributeValues is from [6]), the techniques have been compared by the type of SQLIAs they can support. As we can see in Table 2 three different symbols have been used so as to be able to show the capabilities of various techniques against SQLIA types. The symbol "•" indicates that a technique can prevent all attacks of that type and thus this kind of attack is impossible. On the other hand the symbol "×" is used for indicating that a technique cannot prevent any of the attacks of this kind which means that the attacks are totally possible. The last symbol used is "◦" and is used for indicating attacks that are partially possible.

 Copyright ⓒ 2013 SERSC

Table 2. Comparison of Techniques with Respect to SQLIA Types

Technique	Tautologies	Illegal/Incorrect queries	Union queries	Piggy-Backed Queries	Stored procedures	Inference	Alternate encodings
AMNESIA [17]	•	•	•	•	×	•	•
CANDID [16]	◦	◦	◦	◦	◦	◦	◦
SQLCheck [18]	•	•	•	•	×	•	•
SQLGuard [15]	•	•	•	•	×	•	•
RemovingAttributeValues[6]	•	•	•	•	◦	•	•
Proposed technique	•	•	•	•	◦	•	•
Symbols	•: impossible ◦: partially possible ×: totally possible						

Among the techniques CANDID is able to detect and prevent all of the attack types partially, whereas other techniques can detect and prevent all the attack types except stored procedures. Attacks through the stored procedure are critical because all of the techniques are unable to stop them either partially or at all. That is because of only considering the queries generated within the web application. Therefore, these types of attacks (which consist of all the SQLIA types) make serious problem for the techniques. Between the rest of the techniques only RemovingAttributeValues and our proposed technique are partially vulnerable to this type of attack where partially means that they cannot prevent all of the attacks that occur through the stored procedures due to the fact that both of them focus on SQLIAs in the application layer and therefore, SQLIAs through the stored procedures of the database layer still exist. But, we are planning to improve our solution in order to be able to cover these types of attacks as well. From this table we can conclude that except CANDID all the other techniques are somehow suitable. That is because at least they can protect the web applications against the mentioned SQLIAs totally, although that is not enough because as we know all SQLIA types are possible through stored procedures.

5.3. Comparison of Techniques with Respect to Deployment Requirements

Table 3 shows the deployment requirements of different techniques. It shows if any code modification is needed by the developer, if the detection and prevention of the attacks is done automatically or not and finally if any additional infrastructure is needed. In this table AMNESIA, SQLCheck and SQLGuard are from [7], CANDID is from [9] and RemovingAttributeValues is from [6]. Among the techniques, only SQLCheck and SQLGuard need code modification which can be time consuming, expensive and also error prone due to human effort. Attack detection is automatic in all of them except SQLCheck and SQLGuard, whereas attack prevention in all of them is automatic. Having a detection or prevention which is automatic would simplify the usage of the technique and therefore become an advantage. Finally the last column shows that only SQLCheck needs an additional infrastructure which is key management whereas other techniques do not need any. The result of this comparison is that SQLCheck has the most development requirements, after that becomes SQLGuard while the rest of the techniques require none.

Table 3. Comparison of Techniques with Respect to Deployment Requirements

Technique	Modify code base	Detection	Prevention	Additional infrastructure
AMNESIA [17]	Not needed	Automatic	Automatic	None
CANDID [16]	Not needed	Automatic	Automatic	None
SQLCheck [18]	Needed	Semi automatic	Automatic	Key management
SQLGuard [15]	Needed	Semi automatic	Automatic	None
RemovingAttributeValues[6]	Not needed	Automatic	Automatic	None
Proposed technique	Not needed	Automatic	Automatic	None

5.4. Discussion

In our proposed technique user inputs of SQL queries are removed and some information is gathered in the static phase, which makes the attack detection easier and faster than [6] at runtime (the overhead improvement for the web applications listed in Table 1 respectively are: 24.22%, 19.65% and 20.37%). Therefore, whenever the input is not in its location, injection has taken place and before further inspection we can detect and also prevent it. It has to be mentioned that because most of the attackers try to inject through the first input so that later parts of the query are commented and have no impact, the attack is identified on the initial steps. Another advantage related to the previous one is that we only traverse the input which is a small fraction of the total query while in [6] the entire query is traversed. In this way we give a faster solution for SQLIAs which is a very important issue in real world web applications that need a real time interaction between the users and web applications that makes it another advantage of the proposed technique.

Despite the advantages of our proposed technique there are some shortcomings. Not being capable of totally identifying stored procedure attacks, not working for numeric variables and dynamic queries, and having false positives when the legitimate input contains operator '. The first problem is because stored procedures are at the database layer and the query structures inside them are not available at the application server, but the second one is because numeric variables are not surrounded by '' to be able to use our technique, for dynamic queries because we generate the structure of queries statically we are not capable of managing them, and finally the last problem is due to the strategy that we have in generating the query models.

6. Conclusion

In this paper we propose a new technique based on static analysis and runtime validation for detecting and preventing SQLIAs. Therefore, as a result the security of the web application and the underlying database that contains valuable data is preserved. In this technique user inputs in SQL queries are removed and some information gathered in order to make the detection easier and faster at runtime. Furthermore we evaluated the performance of our proposed technique and compared it with other techniques. The evaluation showed that beside the benefits, our proposed technique is not capable of totally identifying stored procedure attacks, it also does not work for dynamic queries and numeric variables and finally considers legitimate inputs containing operator ' as attacks. These shortcomings can be solved and thus we are planning to work on them as a future work to improve the method.

 Copyright ⓒ 2013 SERSC

References

[1] M. Monga, R. Paleari and E. Passerini, “A hybrid analysis framework for detecting web application vulnerabilities”, Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems, IEEE Computer Society, Vancouver, Canada, **(2009)** May 19.
[2] OWASP Top 10-2010, **(2010)**.
[3] B. Martin, M. Brown, A. Paller and D. Kirby, 2011 CWE/SANS Top 25 Most Dangerous Software Errors, **(2011)**.
[4] C. Anley, “Advanced SQL Injection in SQL Server Applications”, Next Generation Security Software Ltd, **(2002)**.
[5] G. Lawton, J. Computer, vol. 40, no. 13, **(2007)**.
[6] I. Lee, S. Jeong, S. Yeo and J. Moon, J. Mathematical and Computer Modelling, vol. 55, no. 58, **(2011)**.
[7] W. G. J. Halfond, J. Viegas and A. Orso, “A Classification of SQL Injection Attacks and Countermeasures”, Paper presented at the Proceeding on International Symposium on Secure Software Engineering, Arlington, VA, USA, **(2006)** March.
[8] C. Song, “SQL Injection Attacks and Countermeasures”, California State University, Sacramento, **(2010)**.
[9] A. Tajpour, S. Ibrahim and M. Sharifi, International Journal of Computer Science Issues, vol. 9, no. 332, **(2012)**.
[10] R. A. McClure and I. H. Kruger, “SQL DOM: compile time checking of dynamic SQL statements”, Paper presented at the Proceedings of the 27th international conference on Software engineering. IEEE, St. Louis, Missouri, USA, **(2005)** May 15-21.
[11] W. R. Cook and S. Rai, “Safe Query Objects: Statically Typed Objects as Remotely Executable Queries”, Paper presented at the Proceedings of the 27th International Conference on Software Engineering, IEEE, St. Louis, Missouri, USA, **(2005)** May 15-21.
[12] A. B. M. Ali, A.Y. I. Shakhatreh, M. S. Abdullah and J. Alostad, J. Procedia Computer Science, vol. 3, no. 453, **(2011)**.
[13] W. Jie, R. C. W. Phan, J. N. Whitley and D. J. Parish, “Augmented attack tree modeling of SQL injection attacks”, Paper presented at the Information Management and Engineering (ICIME), The 2nd IEEE International Conference on. IEEE, Chengdu, **(2010)** April 16-18.
[14] Y. Kosuga, K. Kernel, M. Hanaoka, M. Hishiyama and Y. Takahama, “Sania: Syntactic and Semantic Analysis for Automated Testing against SQL Injection”, Paper presented at the Computer Security Applications Conference, ACSAC, Twenty-Third Annual, Miami Beach, FL, **(2007)** December 10-14.
[15] G. Buehrer, B. W. Weide and P. A. G. Sivilotti, “Using parse tree validation to prevent SQL injection attacks”, Paper presented at the Proceedings of the 5th international workshop on Software engineering and middleware, Lisbon, Portugal, **(2005)** September.
[16] P. Bisht, P. Madhusudan and V. N. Venkatakrishnan, J. ACM Trans. Inf. Syst. Secur, vol. 13, **(2010)**.
[17] W. G. Halfond and A. Orso, “AMNESIA: analysis and monitoring for neutralizing SQL-injection attacks”, Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, Long Beach, California, USA, **(2005)** November 7-11.
[18] Z. Su and G. Wassermann, “The essence of command injection attacks in web applications”, SIGPLAN Not, Charleston, South Carolina, USA, vol. 41, no. 1, **(2006)** January 11-13.
[19] M. Muthuprasanna, W. Ke and S. Kothari, “Eliminating SQL Injection Attacks-A Transparent Defense Mechanism”, Web Site Evolution, WSE '06, Eighth IEEE International Symposium on, Philadelphia, PA, **(2006)** September 23-24.
[20] W. G. J. Halfond, A. Orso and P. Manolios, J. Software Engineering, IEEE Transactions on, Software Engineering, vol. 34, no. 65, **(2008)**.

Authors

Zeinab Lashkaripour received her BS degree in computer engineering from the University of Sistan and Baluchestan, Iran in 2009. She is currently working toward the MS degree in computer engineering in Ferdowsi University of Mashhad, Iran. Her research work is web application and security and she is a member of Data and Communication Security Laboratory of Ferdowsi University. Other areas of interest include Database Security and Cryptography.

Abbas Ghaemi Bafghi received his BS degree in Applied Mathematics in Computer from Ferdowsi University of Mashhad, Iran in 1995, and MS degree in Computer engineering from Amirkabir (Tehran Polytechnique) University of Technology, Iran in 1997. He received his PhD degree in Computer engineering from Amirkabir (Tehran Polytechnique) University of Technology, Iran in 2004. He is a member of Computer Society of Iran (CSI) and Iranian Society of Cryptology (ISC). He is an assistant professor in Department of Computer Engineering, Ferdowsi University of Mashhad, Iran. His research interests are cryptology and security. It should be mentioned that he has published more than 50 conference and journal papers.

Copyright ⓒ 2013 SERSC