Obfuscation-based Analysis of SQL Injection Attacks

Raju Halder
Dipartimento di Informatica
Università Ca' Foscari di Venezia, Italy
halder@unive.it

Agostino Cortesi
Dipartimento di Informatica
Università Ca' Foscari di Venezia, Italy
cortesi@unive.it

Abstract—**In this paper, we propose an obfuscation/deobfuscation based technique to detect the presence of possible SQL Injection Attacks (SQLIA) in a query before submitting it to a DBMS. This technique combines static and dynamic analysis. In the static phase, the queries in the application are replaced by queries in obfuscated form. The main idea behind obfuscation is to isolate all the atomic formulas from other control elements of the query. During the dynamic phase, the user inputs are merged into the obfuscated atomic formulas, and the dynamic verifier analysis the presence of possible SQLIA at atomic formula level. Finally, a deobfuscation step is performed to recover the original query before submitting it to the DBMS.**

Keywords-**SQL Injection Attack; Obfuscation; Deobfuscation;**

I. Introduction

The recent surge in the growth of the Internet results in the offering of a wide range of web-based services, such as online stores, e-commerce, social network services, etc. However, this increasing popularity of web-based services make them an ideal target for different attacks. One of the most serious type of attacks against web applications is the family of the so called SQL injection attacks (SQLIA). In an SQL injection attack, data provided by the user during run-time are included in an SQL query in such a way that part of the user's input is treated as SQL code. The web applications which receive input from users and incorporate it into SQL queries to an underlying database possibly suffers from SQLIA.

For example, suppose a database contains user names and passwords, and the underlying application contains the following code:

```
query="SELECT * FROM emp WHERE username=‘" +
       request.getParameter("username") + "’ AND password=‘" +
       request.getParameter("password") + "’;"
```

This query is used to authenticate the user who tries to login to the web site, by checking username and password against data stored in the database. However, if a malicious user enters the input: "’ OR ‘1’=‘1’--" into the username field, the query string submitted to the database would be:

```
SELECT * FROM emp WHERE
username=‘ ’ OR ‘1’=‘1’--’ AND password=‘ ’;
```

Since the atomic formula ‘1’=‘1’ is a tautology, the user will bypass the check, and authentication will be successful.

In [1] authors classify the SQL injection attacks into different types: *i*) Injection through user input; *ii*) Injection through cookies; *iii*) Injection through server variables; *iv*) Second-order injection. The characterization of attacks is based on goals or intents of the attacker. The different types of attack intents are: Identifying injectable parameters, Performing database finger-printing, Determining database schema, Extracting data, Adding or modifying data, Performing denial of service, Evading detection, Bypassing authentication, Executing remote commands, Performing privilege escalation.

In this paper, we propose a novel scheme to detect the presence of SQLIA, combining static and dynamic analysis whose main features are: (*i*) It is based on obfuscation and deobfuscation of SQL commands, (*ii*) SQLIA can easily be detected and has a negligible run-time overhead because of dynamic verification is carried out on the obfuscated queries at atomic formula level and the number of verifications for possible SQLIA has been reduced by introducing the notion of secure and vulnerable terms and formulas, (*iii*) The obfuscation and deobfuscation techniques are application-independent and developers need not to be aware about this.

The proposed scheme has three phases, the first one is performed statically, while the latter two are performed dynamically.

1) Obfuscating the legitimate query Q into Q' at each hotspot of the application.
2) After merging the user inputs into the obfuscated query at run-time, the dynamic verifier checks the obfuscated query at atomic formula level in order to detect the presence of possible SQLIA.
3) Reconstruction of the original query Q from the obfuscated query Q' before submitting it to the database, if no possible SQLIA was detected.

We denote any SQL command Q by a tuple $Q \triangleq \langle A, \phi \rangle$. We call the first component A the *active part* and

978-1-4244-7755-5/10/$26.00 ©2010 IEEE

the second component ϕ the *passive part* of Q. Any SQL command Q first identifies an active data set from the database using the pre-condition ϕ, and then performs the appropriate operations on that data set using the SQL action A. The pre-condition ϕ appears in SQL commands as a well-formed formula in first-order logic. The active data set on which A performs operations must satisfy the pre-condition ϕ [2]. The control part of the SQL queries includes the SQL keywords such as WHERE, TABLE, SELECT etc. whereas the data part includes the constants and variables. We consider two types of variables: application variables and database variables. The application variables are defined in the application whereas database variables represent the table attributes of the underlying database. Either the active and passive part of the SQL queries can have data as well as control elements.

In the static analysis phase, the queries in the application are replaced by the queries in obfuscated form. The main idea behind the obfuscation of a query string is just to avoid the string concatenation operation in query generation process, which is considered as the possible root cause of SQLIA [3]. The obfuscation is carried out by converting the SQL command $Q \triangleq \langle A, \phi \rangle$ into a form such that the pre-condition ϕ is partitioned into two sets S_c and S_f, where the former contains all connectives (AND, OR, NOT) of ϕ, and the latter contains all atomic formulas present in ϕ.

During run-time, the inputs provided by the user are merged into the atomic formulas in S_f of the obfuscated query. The dynamic verifier, then, verifies at atomic formula level for the possible existence of SQLIA. The task of the dynamic verifier is to determine whether the elements in the set S_f of the obfuscated query are valid atomic formulas or not. Thus the input of the dynamic verifier is an atomic formula merged with user input and the output is a boolean value indicating whether it is a valid atomic formula or not. If any violation is detected in the verification phase, the dynamic verifier reports it as a possible SQLIA, otherwise the run-time converter (which may be part of the verifier) converts the obfuscated query into the original form before submitting it to the DBMS. The overall architecture of our approach is depicted in Figure 1 where at the beginning we assume that the user launches a job involving one or more web applications interacting with the DBMS.

The attractive features of this scheme is that the static phase removes the traditional string concatenation operations (which is the root cause of possible SQLIA) used to build all the SQL query strings in the application and replace them by the obfuscated form of queries. Since the user inputs are merged into the atomic formulas in obfuscated form, there is no chance to mix-up the malicious input with the other legitimate control elements of the query string and dynamic verification at atomic formula level can easily detect the presence of possible SQLIA. The number of dynamic verification is reduced by introducing two categories of terms and formulas: secure and vulnerable ones. The verification is carried out only over the vulnerable atomic formulas. Moreover, we can avoid obfuscation for those queries which have secure pre-conditions.

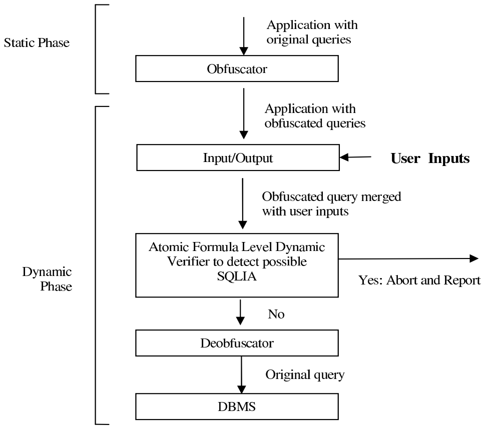
Static Phase
Application with original queries
Obfuscator
Application with obfuscated queries
Input/Output
User Inputs
Obfuscated query merged with user inputs
Dynamic Phase
Atomic Formula Level Dynamic Verifier to detect possible SQLIA
Yes: Abort and Report
No
Deobfuscator
Original query
DBMS

Figure 1. Architecture of the proposed scheme

The structure of the paper is as follows: Section 2 discusses related works in the literature. Section 3 defines the notion of secure and vulnerable terms and formulas involved in pre-condition ϕ. Section 4 and 5 describe the obfuscation and deobfuscation techniques. Section 6 discusses static versus dynamic issues. Section 7 concludes the paper.

II. Related Works

In [4], [5], [6], [7] the authors follow a model-based approach and introduce the tools AMNESIA, SQL-Guard and SQLCheck respectively. All these schemata have two phases: static and dynamic. In AMNESIA, the static phase builds models corresponding to all legitimate queries present in the application by analysing the program. In its run-time phase, before submitting the request to database, all dynamically generated queries are checked against the corresponding model, and the queries violating the models are identified as SQLIA. Observe that, the accuracy of the statically-built model is the measure of the success of AMNESIA. In SQLGuard and SQLCheck, the model is based on a set of grammar-rules against which the dynamically generated queries are checked to detect the possibility of SQLIA. In both cases, the user inputs are augmented by some delimiter generated from a private key. Thus the success of this two schemata are completely dependent on the fact that the attacker is not being able to discover the private key. In [7] the static model is based on graph representation of the query.

McClure and Krüger [8] provides a completely different query development platform by changing the so-called unregulated query generation process that uses string concatenation, to a new systematic one. This solution consists of two parts. The first is an abstract object model. The second is an executable which is executed against a database and the output is a Dynamic Link Library (DLL) containing classes that are strongly-typed to the database schema. This DLL is used as a concrete instantiation of the abstract object model. However, as they provide a completely new paradigm for query development process which is not as easy as previous one, the developer need to learn before its use.

SQLrand [9] is based on the concept of Instruction-Set Randomization. The SQL standard keywords are manipulated by appending a random integer to them. The attacker is not aware about that random integer. Thus if any malicious user attempting to SQL injection attack would immediately be thwarted as the injective codes in the randomized query are treated as non-keywords.

In [10] Valeur et al. propose a learning-based Intrusion Detection System (IDS) to detect SQLIA. The IDS is trained using a set of typical application queries. The technique builds statistical models of the typical queries and then monitors the application at run-time to identify the queries that do not match the model. However, the fundamental limitation is that the success of such system completely depends on the quality of the training set used. Poor training set would result large number of false positive and false negative.

Scott and Sharp, in their work [11] propose a solution to provide an application level security including SQLIA for web-based applications. They use a security policy description language (SPDL) to specify a set of validation constraints and transformation rules to be applied to application parameters as they flow from the web page to the application server. The compiled SPDL codes are kept on a security gateway which acts as application level firewall. However, the developers are completely responsible for this. They have to know not only which data needs to be filtered, but also what patterns and filters need to apply to data.

Many testing techniques [12], [13], [14] have been proposed to test whether the web applications are vulnerable to SQLIA. In [12] the proposed technique is based on black-box approach, whereas, [13], [14] describe white-box approach.

In [15], [16], [3], [17], authors provide taint-based approach to SQLIA: static analysis is used to check taint flows against preconditions for sensitive functions. The analysis detects the points in which preconditions have not been met and can suggest filters and sanitization functions that can be automatically added to the application to satisfy these preconditions.

Among the most recent works, [18] is a defensive coding practices to prevent SQLIA. It describes input validation (whitelist, blacklist) and encoding techniques (Sanitize input) to ensure the safety of input. It also introduces a hybrid strategy combining them. The defensive approach suffers from the false positive or false negative problems. Defensive coding is prone to human error and is not as rigorously and completely applied as automated techniques.

III. Secure and Vulnerable Terms and Formulas

In this section, we define the terms and formulas in first-order logic, and we introduce the notion of secure and vulnerable terms with regards to different attacks.

Definition 1: (Terms) The set of terms of a first-order language L is the set of strings of symbols formed according to the following rules:

- All the variable symbols $x_1, x_2, x_3, ...$ and all the constant symbols c_i in L are terms.
- If f_n is an n-ary function symbol in L and $t_1, t_2, ..., t_n$ are terms, then $f_n(t_1, t_2, ..., t_n)$ is a term.

Definition 2: (Atomic Formula) An atomic formula is a string of symbols of the form $R_n(t_1, t_2, ..., t_n) \in \{true, false\}$, where R_n is a relation symbol of arity n of the language and $t_1, t_2, ..., t_n$ are terms.

If the language contains the equality relation, then any string of the form $t_1 = t_2$, where t_1, t_2 are terms, is also an atomic formula.

Definition 3: (Well-formed Formula) The set of formulas of a language L is the set of strings of symbols formed according to the following rules:

- All atomic formulas are Well-formed Formulas (wffs).
- If ϕ and ψ are formulas and x_i is a variable symbol, then so are $\neg\phi, \phi \wedge \psi, \phi \vee \psi, \phi \rightarrow \psi, \phi \leftrightarrow \phi, \forall x_i \phi, \exists x_i \phi$.

Now we define the notion of secure and vulnerable terms, atomic formulas and well-formed formulas. Let C_ϕ and V_ϕ denote the set of constants and variables, respectively, appearing in the pre-condition ϕ. Let C_{sec} and V_{sec} are set of secure constants and variables respectively whereas, V_{vul} stands for the set of public variables which are vulnerable to different attacks. Observe that, since constants and database variables are provided by the developers, we assume them as secure. Application variables may or may not be secure depending on whether they are used as public variables directly or influenced by other vulnerable variables indirectly. Therefore,

$$C_\phi \subseteq C_{sec} \ and \ V_\phi \subseteq V_{sec} \cup V_{vul}$$

Let T_{sec}, AF_{sec} and WFF_{sec} represent the set of secure terms, atomic formulas and well-formed formulas, whereas T_{vul}, AF_{vul} and WFF_{vul} represent the set of vulnerable terms, atomic formulas and well-formed formulas respectively. We can define inference rules for terms, atomic and well-formed formulas being secure and vulnerable as shown in Table I.

Observe that the rules in Table I are not closed under logical equivalence, *i.e.* $\phi_1 \in WFF_{sec}$ and $\phi_2 \equiv \phi_1$ do not imply that $\phi_2 \in WFF_{sec}$. For instance, let $x \in V_{vul}$, $y \in V_{sec}$ and $c \in C_{sec}$, we have: $(y = c) \in WFF_{sec}$ while $(x = x) \wedge (y = c) \in WFF_{vul}$ even though the two formulas are equivalent. This is due to the fact that $(x = x)$ is a potential gateway for SQLIA.

$$\frac{c \in C_{sec}}{c \in T_{sec}} \qquad \frac{v \in V_{sec}}{v \in T_{sec}} \qquad \frac{v \in V_{vul}}{v \in T_{vul}}$$

$$\frac{\forall t_i \in T_{sec},\ i=1,\dots,n}{f(t_1,\dots,t_n) \in T_{sec}} \qquad \frac{\exists t_i \in T_{vul},\ i=1,\dots,n}{f(t_1,\dots,t_n) \in T_{vul}}$$

$$\frac{\forall t_i \in T_{sec},\ i=1,\dots,n}{R_n(t_1,\dots,t_n) \in AF_{sec}} \qquad \frac{\exists t_i \in T_{vul},\ i=1,\dots,n}{R_n(t_1,\dots,t_n) \in AF_{vul}}$$

$$\frac{t_1 \in T_{sec} \qquad t_2 \in T_{sec}}{(t_1 = t_2) \in AF_{sec}} \qquad \frac{t_1 \in T_{vul}}{(t_1 = t_2) \in AF_{vul}}$$

$$\frac{t_2 \in T_{vul}}{(t_1 = t_2) \in AF_{vul}} \qquad \frac{a \in AF_{sec}}{a \in WFF_{sec}} \qquad \frac{a \in AF_{vul}}{a \in WFF_{vul}}$$

$$\frac{w \in WFF_{sec}}{\neg w \in WFF_{sec}} \qquad \frac{w \in WFF_{vul}}{\neg w \in WFF_{vul}}$$

$$\frac{w \in WFF_{sec}}{(\vartheta\ x)\ w \in WFF_{sec}} \qquad \frac{w \in WFF_{vul}}{(\vartheta\ x)\ w \in WFF_{vul}}$$

$$\frac{w_1 \in WFF_{vul}}{(w_1 \theta\ w_2) \in WFF_{vul}} \qquad \frac{w_2 \in WFF_{vul}}{(w_1 \theta\ w_2) \in WFF_{vul}}$$

$$\frac{w_1 \in WFF_{sec} \qquad w_2 \in WFF_{sec}}{(w_1 \theta\ w_2) \in WFF_{sec}}$$

Table I
INFERENCE RULES FOR TERMS, ATOMIC AND WELL-FORMED FORMULAS BEING SECURE AND VULNERABLE, WHERE $\vartheta \in \{\forall, \exists\}$ AND $\theta \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$ AND x IS A BOUND VARIABLE

For the sake of simplicity, in the rest of the paper we consider the following assumptions:

1) The passive part ϕ of the SQL command is a function of user input, whereas the active part is not.
2) The developer-provided part of SQL command is reliable, whereas the user-provided part is not trusted and vulnerable to SQLIA.
3) Users are not permitted to provide any control elements of query. They are only allowed to provide the data elements to the SQL command.

Observe that the assumptions above do not yield to severe limitations, and are reasonable in practice.

IV. Obfuscation of SQL Commands

In this section, we discuss the obfuscation scheme. We first illustrate it on a simple example, and then we present the actual algorithm.

Suppose, $\phi_1, \phi_2, ..., \phi_n$ represent a set of atomic predicate formulas involved in a pre-condition ϕ of a SQL command Q. For example, let the following formula represent a pre-condition ϕ which contains the atomic formulas $\phi_1, \phi_2, ..., \phi_7$:
$\phi = (\phi_1\ AND\ \phi_2\ OR\ \phi_3) AND (\phi_4\ OR\ (NOT \phi_5))\ OR\ \phi_6\ AND\ \phi_7$
For the simplicity of representation, we denote *AND*, *OR*, *NOT* by $\times, +$ and ! respectively. Thus,

$$\phi = (\phi_1 \times \phi_2 + \phi_3) \times (\phi_4 + (!\phi_5)) + \phi_6 \times \phi_7$$

Since the connectives are unary or binary, the parse tree of ϕ represents a binary tree as shown in Figure 2. The obfuscation of the original query Q is obtained by converting the pre-condition ϕ in such a form in which it is separated into two distinct partitions. The first partition contains all the connectives $(\times, +, !)$ of ϕ whereas, the second partition contains all the atomic formulas $\phi_1, ..., \phi_n$ of ϕ.

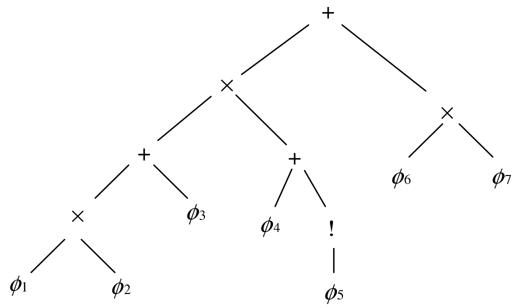

Figure 2. The parse-tree of the example pre-condition ϕ

Observe that if we convert ϕ into some prefix or postfix form considering $\times, +, !$ as operators and ϕ_i, i=1,...,n as operands, still there is a mixing of connectives and formulas. For example, if we convert the above formula ϕ into prefix form we get: $+ \times + \times \phi_1\phi_2\phi_3 + \phi_4!\phi_5 \times \phi_6\phi_7$. Since only a fraction of the connectives $(\times, +, !)$ of ϕ has been separated, still there is a chance of possible SQLIA.

To remove this problem and to obtain two exclusive partitions of connectives and atomic formulas, we adopt a different technique consisting of the following steps:
Assign a unique label to each of the connectives $(\times, +, !)$ *and atomic formulas ϕ_i, i=1,..,n in ϕ, and partition the connectives and atomic formulas into two different sets.*

The task of assigning unique label is performed as follows: We start assigning a bit to each node in the parse tree as follows: assign 0 to the root node. If any internal node has single child, assign NULL to that child node. Otherwise, the left and right child of the node are assigned with 0 and 1 respectively. Continue this process until all the nodes of the parse tree has been assigned by bits. The bit-assigned binary parse tree of the formula ϕ of our example is shown in Figure 3. We know that each node in a tree has a unique path from the root to that node. To get a unique label for a node v, traverse the path from root to the node v in the tree. Collect all the bits of the nodes appearing in that path and concatenate them in the direction of traversing. This gives a unique binary string which is used as a unique label. For example, in the bit-assigned parse tree of Figure 3, the unique label for ϕ_4 is 0010, and for ϕ_5 is 0011 (equivalently, 0011NULL). This unique labels allow to reconstruct the original query from the obfuscated form (as described later).

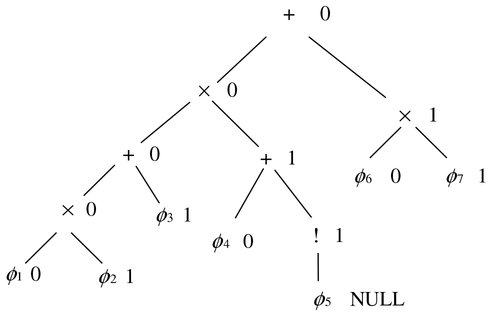

Figure 3. The bit-assigned parse-tree of the example pre-condition ϕ

We convert the above pre-condition ϕ into a new form in which each token is represented by a tuple $\langle c, l_c \rangle$ or $\langle f, l_f \rangle$, where $c \in \{\times, +, !\}$ and $f \in \{\phi_i \mid i = 1, .., n\}$. The unique labels l_c and l_f obtained from the bit-assigned parse tree, are associated with c and f respectively. The converted ϕ results into:

$((\langle \phi_1, 00000 \rangle \ \langle \times, 0000 \rangle \ \langle \phi_2, 00001 \rangle \ \langle +, 000 \rangle \ \langle \phi_3, 0001 \rangle) \ \langle \times, 00 \rangle \ ((\langle \phi_4, 0010 \rangle \ \langle +, 001 \rangle \ ((\langle !, 0011 \rangle \ \langle \phi_5, 0011 \rangle))) \ \langle +, 0 \rangle \ \langle \phi_6, 010 \rangle \ \langle \times, 01 \rangle \ \langle \phi_7, 011 \rangle$

Now we are in position to construct two distinct sets S_c and S_f taking all the connectives into one and all the atomic formulas into the other respectively. Thus,

$S_c = \{\langle \times, 0000 \rangle, \langle +, 000 \rangle, \langle \times, 00 \rangle, \langle +, 001 \rangle, \langle !, 0011 \rangle, \langle +, 0 \rangle, \langle \times, 01 \rangle\}$

$S_f = \{\langle \phi_1, 00000 \rangle, \langle \phi_2, 00001 \rangle, \langle \phi_3, 0001 \rangle, \langle \phi_4, 0010 \rangle, \langle \phi_5, 0011 \rangle, \langle \phi_6, 010 \rangle, \langle \phi_7, 011 \rangle\}$

These two sets S_c and S_f represent the obfuscated pre-condition. So, the obfuscated SQL command can be written as: $Q' = \langle A, [S_c, S_f] \rangle$.

It is worthwhile to mention that the order of the elements in the two partitions S_c and S_f are not relevant at all. At run-time, inputs given by the users are merged into the atomic formulas in S_f. Since S_f is a set of atomic formulas and user inputs are part of the elements in S_f at run-time, there is a strong chance for the attacker to change the number of elements (atomic formulas) in S_f. But their target can not be successful anymore because of the fixed number of connectives present in the set S_c. We can see in the next section that, any change in the number of atomic formulas in S_f would yield the failure of the deobfuscation phase.

The dynamic verifier will verify the atomic formulas for the possible SQLIA after getting run-time inputs. The input of the dynamic verifier is an atomic formula merged with user input and the output is a boolean value indicating whether it is a valid atomic formula or not. To reduce the number of verifications, we categorize each atomic formula in S_f based on whether it belongs to AF_{sec} or AF_{vul} and tag them accordingly at the end of the static phase. During the dynamic phase, only the atomic formulas which are tagged as vulnerable will be checked for possible SQLIA. Also note that for any query $Q \triangleq \langle A, \phi \rangle$, we do not obfuscate it if $\phi \in WFF_{sec}$. In case of nested queries, the obfuscation-verification-deobfuscation procedure is performed from the inner-most query to the outer-most query.

We are now in position to formalize the algorithm performing this obfuscation part, as shown in Figure 4.

V. Deobfuscation of SQL Commands

We already mentioned that deobfuscation is done only when no SQLIA has been detected by dynamic atomic formula level verifier. Of course, if the original query is kept, then there is no need of deobfuscation. If this is not the case, the following technique reconstructs the original query $Q \triangleq \langle A, \phi \rangle$ from the obfuscated query $Q' \triangleq \langle A, [S_c, S_f] \rangle$.

To restore ϕ from S_c and S_f (hence, from Q' to Q) in the dynamic phase before submitting it to the database engine, the following steps are performed repeatedly until S_c is empty:

Step 1:Identify the predicate formulas in S_f whose label matches with the label of the connectives in S_c *i.e.* $\exists \langle c, l_c \rangle \in S_c, \exists \langle f, l_f \rangle \in S_f$: $|l_c| = |l_f|$ and $l_c \otimes l_f = 0$. $\otimes$ represents bit-wise XOR operation. Apply the unary connective c on the corresponding matched formula f to get resulting formula f_r. Replace $\langle f, l_f \rangle$ by $\langle f_r, l_c \rangle$ in S_f. Remove $\langle c, l_c \rangle$ from S_c.

978-1-4244-7755-5/10/$26.00 ©2010 IEEE

```
Algorithm 1: Obfuscate_Query
Input: Original Query String $Q \triangleq \langle A, \phi \rangle$
Output: Obfuscated Query $Q' \triangleq \langle A, [S_c, S_f] \rangle$
1. Check whether $\phi \in WFF_{sec}$ or not. If not, perform steps 2-8.
2. Generate binary parse tree of the pre-condition $\phi$ of Q.
3. Assign root node by 0. If any node has single child, assign NULL to this child and if any node has two children, assign left child by 0 and right child by 1, respectively. Continue this step until all the nodes of the tree are assigned.
4. Assign each of the atomic formulas and connectives in $\phi$ by a unique label obtained from the parse tree as follows: for a node v, traverse from root to node v and collect all bits of the nodes appearing in the path. Concatenate all those bits in the direction of traversing. This gives a bit string which is used as the unique label for the node v.
5. After performing step 4, all connectives $c$ and atomic formulas $f$ of $\phi$ will be of the form $\langle c, l_c \rangle$ and $\langle f, l_f \rangle$, where $l_c$ and $l_f$ denote the unique labels assigned to $c$ and $f$, respectively.
6. Partition the connectives $\langle c, l_c \rangle$ and atomic formulas $\langle f, l_f \rangle$ of $\phi$ into two distinct sets $S_c$ and $S_f$, respectively.
7. If for $\langle f, l_f \rangle \in S_f$ and $f \in AF_{sec}$, tag it as secure i.e. $tag(f) := sec$; otherwise, tag it as vulnerable i.e. $tag(f) := vul$.
8. The obfuscated form of the query Q is, therefore, $Q' \triangleq \langle A, [S_c, S_f] \rangle$.
```

Figure 4. Query Obfuscation Algorithm

Step 2: Identify all the pair of elements $\{\langle f_1, l_1 \rangle, \langle f_2, l_2 \rangle\} \subseteq S_f$ such that, $|l_1| = |l_2|$ and only the last bit of l_1 and l_2 differs. Identify the connective $\langle c, l \rangle \in S_c$ where $|l| = |l_1| - 1 = |l_2| - 1$ and l is equal to the common part of l_1 and l_2. Apply the binary connective c on that pair to obtain the resulting formula f_r. Replace the pair by a new tuple $\langle f_r, l \rangle$ in S_f and remove $\langle c, l \rangle$ from S_c.

The formal description of possible SQLIA detection and deobfuscation algorithm is shown in Figure 5.

We illustrate the deobfuscation technique with the same example of section 4. Suppose, the obfuscated query $Q' \triangleq \langle A, [S_c, S_f] \rangle$ where S_c and S_f are:

$S_c = \{\langle \times, 0000 \rangle, \langle +, 000 \rangle, \langle \times, 00 \rangle, \langle +, 001 \rangle, \langle !, 0011 \rangle, \langle +, 0 \rangle, \langle \times, 01 \rangle\}$

$S_f = \{\langle \phi_1, 00000 \rangle, \langle \phi_2, 00001 \rangle, \langle \phi_3, 0001 \rangle, \langle \phi_4, 0010 \rangle, \langle \phi_5, 0011 \rangle, \langle \phi_6, 010 \rangle, \langle \phi_7, 011 \rangle\}$

We perform the two steps above repeatedly until the set S_c is empty.

1) In the example, the label of $\langle \phi_5, 0011 \rangle \in S_f$ matches with the label of $\langle !, 0011 \rangle \in S_c$. So after performing step 1 we get:

```
Algorithm 2: Deobfuscate_Query
Input: Obfuscated Query $Q' \triangleq \langle A, [S_c, S_f] \rangle$
Output: Claiming for possible SQLIA as true or false; Original Query String $Q \triangleq \langle A, \phi \rangle$
1. Perform dynamic verification on all atomic formulas $\langle f, l_f \rangle \in S_f \wedge tag(f) = vul$ merged with run-time inputs given by the users, for any possible violation. If violates, claim:=true else claim:=false.
2. If claim=false, perform step 3-5, until $S_c$ is empty.
3(a). Identify the predicate formulas in $S_f$ whose label matches with the label of the connectives in $S_c$ i.e. $\exists \langle c, l_c \rangle \in S_c, \exists \langle f, l_f \rangle \in S_f$: $|l_c| = |l_f|$ and $l_c \otimes l_f = 0$. $\otimes$ represents bit-wise XOR operation.
3(b). Apply the unary connective $c$ on the corresponding matched formula $f$ to get resulting formula $f_r$. Replace $\langle f, l_f \rangle$ by $\langle f_r, l_c \rangle$ in $S_f$. Remove $\langle c, l_c \rangle$ from $S_c$.
4(a). Identify all the pair of elements $\{\langle f_1, l_1 \rangle, \langle f_2, l_2 \rangle\} \subseteq S_f$ such that, $|l_1| = |l_2|$ and only the last bit of $l_1$ and $l_2$ differs.
4(b). Identify the connective $\langle c, l \rangle \in S_c$ where $|l| = |l_1| - 1 = |l_2| - 1$ and $l$ is equal to the common part of $l_1$ and $l_2$. Apply the binary connective $c$ on that pair to obtain the resulting formula $f_r$. Replace the pair by a new tuple $\langle f_r, l \rangle$ in $S_f$ and Remove $\langle c, l \rangle$ from $S_c$.
5. If $S_c$ is empty, $S_f$ contains the original form of pre-condition $\phi$ and submit $Q \triangleq \langle A, \phi \rangle$ to the DBMS.
```

Figure 5. Algorithm to Detect possible SQLIA and deobfuscation of the Query

$S_c = \{\langle \times, 0000 \rangle, \langle +, 000 \rangle, \langle \times, 00 \rangle, \langle +, 001 \rangle, \langle +, 0 \rangle, \langle \times, 01 \rangle\}$

$S_f = \{\langle \phi_1, 00000 \rangle, \langle \phi_2, 00001 \rangle, \langle \phi_3, 0001 \rangle, \langle \phi_4, 0010 \rangle, \langle (!\phi_5), 0011 \rangle, \langle \phi_6, 010 \rangle, \langle \phi_7, 011 \rangle\}$

2) Following step 2, we get three pairs whose labels are equal in length and differs by the last bit only: $\{\langle \phi_1, 00000 \rangle, \langle \phi_2, 00001 \rangle\}$, $\{\langle \phi_4, 0010 \rangle, \langle (!\phi_5), 0011 \rangle\}$, and $\{\langle \phi_6, 010 \rangle, \langle \phi_7, 011 \rangle\}$. Since the label of $\langle \times, 0000 \rangle$ equals to the common part of the pair $\{\langle \phi_1, 00000 \rangle, \langle \phi_2, 00001 \rangle\}$ and similarly, $\langle +, 001 \rangle$ and $\langle \times, 01 \rangle$ for the pairs $\{\langle \phi_4, 0010 \rangle, \langle (!\phi_5), 0011 \rangle\}$ and $\{\langle \phi_6, 010 \rangle, \langle \phi_7, 011 \rangle\}$ respectively, after performing step 2, we get,

$$S_c = \{\langle +, 000 \rangle, \langle \times, 00 \rangle, \langle +, 0 \rangle\}$$

$S_f = \{\langle \phi_1 \times \phi_2, 0000 \rangle, \langle \phi_3, 0001 \rangle, \langle \phi_4 + (!\phi_5), 001 \rangle, \langle \phi_6 \times \phi_7, 01 \rangle\}$

3) Since S_c is not empty, we repeat the same until S_c is empty and finally we obtain,

$$\phi = ((\phi_1 \times \phi_2 + \phi_3) \times (\phi_4 + (!\phi_5)) + \phi_6 \times \phi_7)$$

In this way, we can recover the original query $Q \triangleq \langle A, \phi \rangle$ from the obfuscated query $Q' \triangleq \langle A, [S_c, S_f] \rangle$.

Lemma 1: (Time complexity) The time complexity of obfuscation and deobfuscation of a query are $O(n + m)$ and $O(n^2)$ respectively, where n and m are the number of atomic formulas and connectives in the pre-condition ϕ.

Proof: The time complexity of the obfuscation step mainly depends on two facts: *i*) bit-assigned parse tree generation and, *ii*) assigning unique label to each connectives and atomic formulas of ϕ.

Let n and m be the number of atomic formulas and connectives present in the pre-condition ϕ. All the leaf nodes of the parse tree are atomic formulas whereas internal nodes are the connectives.

The generation of parse tree as well as traversing of it to assign bits and extracting unique labels for each of the nodes, needs traversing every nodes exactly once. Hence, the time complexity of the obfuscation is $O(n+m)$.

Time complexity of the deobfuscation step depends on two main operations: *i*) match the labels of unary connectives with the labels of elements in S_f, *ii*) find the pairs in S_f and binary connectives in S_c with specific criteria. Let p be the number of unary connectives in S_c. It is obvious that $m-p = n-1$. The worst case time complexity for the first operation is $O(np)$ whereas best case is $O(p^2)$. The time needed to perform the second operation is $O(n^2 + nm)$. Since, $n \geq m$ and $n \geq p$, the worst-case time complexity for the deobfuscation step is, therefore, $O(n^2)$. ■

Let us illustrate the overall scheme. Recall the example of introduction part where an application generates the following SQL query:

```
query="SELECT * FROM emp WHERE username=‘" +
        request.getParameter("username") + "’ AND password=‘" +
        request.getParameter("password") + "’;"
```

The above query can be represented by the following components:

$$\begin{aligned} A &: \text{“SELECT * FROM emp”} \\ \phi_1 &: \text{“username= ‘}input_1\text{’”} \\ \phi_2 &: \text{“password= ‘}input_2\text{’”} \\ \phi &: \phi_1 \text{ AND } \phi_2 \\ Q &: \langle A,\ \phi \rangle \end{aligned}$$

where, $input_1$ and $input_2$ stand for `request.getParameter ("username")` and `request.getParameter("password")` respectively.

The bit-assigned binary parse tree of ϕ of this example is shown in Figure 6. Thus, the obfuscated query is: $Q' = \langle\ A,\ [S_c, S_f]\ \rangle$ where, $S_c = \{\ \langle \times,\ 0 \rangle\ \}$ and $S_f = \{\ \langle \phi_1, 00 \rangle, \langle \phi_2, 01 \rangle\ \}$.

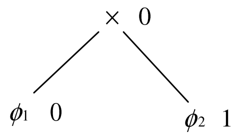

Figure 6. The bit-assigned parse-tree of the example ϕ

Suppose, the attacker gives the following inputs: “`alice’ OR ‘1’=‘1`” and “`secret`” for $input_1$ and $input_2$ respectively. After merging by these inputs, the atomic formulas in S_f of the obfuscated query would be:

$$\begin{aligned} \phi_1 &: \text{“username= ‘alice’ OR ‘1’=‘1’”} \\ \phi_2 &: \text{“password= ‘secret’”} \end{aligned}$$

Clearly, the dynamic verification at atomic formula level says that ϕ_1 is no more indicating a valid atomic formula. So it can be identified as a possible SQLIA.

VI. Static Vs. Dynamic Issues

Previous works [4], [5], [7] proposed mechanisms to generate static models that are used to verify against the queries at dynamic time where the user inputs are allowed to merge into the original query. However, theses schemes may yield to false positive in cases where the users are allowed to provide structural attributes as input. For instance, suppose that a web application generates the following SQL query:

```
query="SELECT * FROM users WHERE id=" +
        request.getParameter("id") + ";"
```

If the web application allows users to provide arbitrary arithmetic expression as input, the structure of the SQL query depends on the expression in the *id* field. Since [4], [5], [7] rely on the fixed static model built at compile time, they yield to a false positive. However our scheme can treat them as valid atomic formulas.

Systems like ModSecurity [19] are provided with input validation using defensive coding practices. They use a white-list/black-list approach to allow/block the good/bad inputs in order to prevent possible SQLIA. These systems are application-specific and developers are responsible to create and maintain white/black-list for specific applications: this is prone to possible human error and can cause both false positives and false negatives. The advantages of our scheme over defensive coding practices is that the developers need not to be aware about the obfuscation-deobfuscation, and completely application independent.

The technique in [9] is based on randomization of the keywords of all SQL queries in the application by appending each keyword with a random value. In practice, it suffers from many aspects. First, a modified database would require all applications submitting SQL queries to conform to its new language. Second, The proxy server which is responsible to check the syntactic validity of the whole randomized queries and de-randomize the instruction set for syntactically valid ones, incurs a significant infrastructure overhead. Third, this technique completely relies on the fact that the attacker is unable to discover the random secret number used to randomize [1]. However, in our scheme, the dynamic verifier checks only the atomic formulas appearing in the queries and this run-time overhead is further minimized by introducing secure and vulnerable atomic formulas: verification is carried out over vulnerable atomic formulas only. Furthermore, unlike [9], for the SQL queries that contain secure passive part, we do not apply obfuscation-deobfuscation to avoid unnecessary processing.

VII. Conclusion

The obfuscation/deobfuscation approach presented in this paper can be seen as a valid alternative to the techniques discussed in section 2. It has the following advantages: the verification for the presence of possible SQLIA is performed at atomic formula level and only on those atomic formulas which are tagged as vulnerable; the scheme avoids the root cause (string concatenation operation) of SQLIA in traditional dynamic query generation; the developer can enjoy the traditional application development techniques and need not to be aware about the obfuscation/deobfuscation techniques. Unlike most of the existing methods, this scheme does not depend on the static models or the private key. It depends only on the accuracy of the dynamic verifier at atomic formula level. We are currently working on the extension of this work in order to deal with cases where the SQL statements are known only at run-time, like in the case of Java Servlets.

Acknowledgement

Work partially supported by Italian MIUR COFIN'07 project "SOFT" and by RAS project TESLA - Tecniche di enforcement per la sicurezza dei linguaggi e delle applicazioni.

References

[1] W. G. Halfond, J. Viegas, and A. Orso, "A classication of sql-injection attacks and countermeasures," in *Proc. of the Intern. Symposium on Secure Software Engineering (ISSSE 2006)*, March 2006.

[2] R. Halder and A. Cortesi, "Abstract interpretation for sound approximation of database query languages," in *Proceedings of the IEEE 7th International Conference on INFOrmatics and Systems (INFOS2010), Advances in Data Engineering and Management Track*. Cairo, Egypt: IEEE Catalog Number: IEEE CFP1006J-CDR, 28–30 March 2010, pp. 53–59.

[3] T. Pietraszek and C. V. Berghe, "Defending against injection attacks through context-sensitive string evaluation," in *Proceedings of Recent Advances in Intrusion Detection (RAID2005)*, 2005.

[4] W. G. Halfond and A. Orso, "Amnesia: Analysis and monitoring for neutralizing sql-injection attacks," in *Proceedings of the IEEE and ACM International Conference on Automated Software Engineering (ASE 2005)*, Long Beach, California, USA, Nov 2005.

[5] G. T. Buehrer, B. W. Weide, and P. A. G. Sivilotti, "Using parse tree validation to prevent sql injection attacks," in *International Workshop on Software Engineering and Middleware (SEM)*, Lisbon, Portugal, Sept 2005.

[6] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," in *The 33rd Annual Symposium on Principles of Programming Languages (POPL 2006)*, January 2006.

[7] M. Muthuprasanna, K. Wei, and S. Kothari, "Eliminating sql injection attacks - a transparent defence mechanism," in *Eighth IEEE International Symposium on Web Site Evolution (WSE'06)*, 2006, pp. 22–32.

[8] R. McClure and I. Krüger, "Sql dom: Compile time checking of dynamic sql statements," in *Proceedings of the 27th International Conference on Software Engineering (ICSE 05)*, St. Louis, Missouri, USA, May 2005.

[9] S. W. Boyd and A. D. Keromytis, "Sqlrand: Preventing sql injection attacks," in *Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference*, June 2004, pp. 292–302.

[10] F. Valeur, D. Mutz, and G. Vigna, "A learning-based approach to the detection of sql attacks," in *Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, Vienna, Austria, July 2005.

[11] D. Scott and R. Sharp, "Abstracting application-level web security," in *Proceedings of the 11th International Conference on the World Wide Web (WWW 2002)*, 2002, pp. 396–407.

[12] Y. Huang, S. Huang, T. Lin, and C. Tsai, "Web application security assessment by fault injection and behavior monitoring," in *Proceedings of the 11th International World Wide Web Conference (WWW 03)*, May 2003.

[13] H. Shahriar and M. Zulkernine, "Music: Mutation-based sql injection vulnerability checking," in *The Eighth International Conference on Quality Software*, 2008, pp. 77–86.

[14] Y. Kosuga, K. Kernel, M. Hanaoka, M. Hishiyama, and Y. Takahama, "Sania: Syntactic and semantic analysis for automated testing against sql injection," in *Twenty-Third Annual Computer Security Applications Conference*, 2007, pp. 107–117.

[15] V. Haldar, D. Chandra, and M. Franz, "Dynamic taint propagation for java," in *Proceedings 21st Annual Computer Security Applications Conference*, December 2005.

[16] Y. Huang, F. Yu, C. Hang, C. H. Tsai, D. T. Lee, and S. Y. Kuo, "Securing web application code by static analysis and runtime protection," in *Proceedings of the 12th International World Wide Web Conference (WWW 04)*, New York, USA, May 2004.

[17] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans, "Automatically hardening web applications using precise tainting information," in *Twentieth IFIP International Information Security Conference (SEC 2005)*, May 2005.

[18] J.-C. Lin, J.-M. Chen, and C.-H. Liu, "An automatic mechanism for adjusting validation function," in *22nd International Conference on Advanced Information Networking and Applications - Workshops*, Okinawa, 2008, pp. 602–607.

[19] Mod-Security, "http://www.modsecurity.org."