An Approach to Detection of SQL Injection Attack Based on Dynamic Query Matching

Debasish Das, Utpal Sharma & D.K. Bhattacharyya
Department of Computer Science & Engineering
Tezpur University
Napaam (INDIA)

ABSTRACT

A large number of web applications, especially those deployed by companies for e-business operations involve high reliability, efficiency and confidentiality. Such applications are often written in script languages like PHP embedded in HTML, allowing establishing connection to databases, retrieving data, and putting them in the Web. One of the most common in web application attacks is SQL Injection. In this an attacker attempts to use malicious crafted input strings so that the dynamic SQL queries generated by the web application is different from the structure designed by the developer. In this paper, an attempt has been made to classify the SQL Injection attacks based on the vulnerabilities in web applications. A brief review of the existing approaches for the detection of SQL injection attack also has been presented. Further paper presents an effective detection method (DUD) for the SQL injection based on dynamic query matching. The DUD approach is independent of the developer's initialization of syntactical rules, valid trusted string database, static or pre-generated program code checking, etc. Also, DUD is significant in view of its simple detection mechanism as well as its high detection rate.

Categories and Subject Descriptors

D.4.6 [Security and Protection]: Access Control, Authentication, Information Flow Controls, Verifications.

General Terms

Algorithms, Performance, Design, Reliability, Experimentation, Security.

Keywords

Web, PHP, SQL injection, classification, DUD.

1. INTRODUCTION

A web application is software which end users can access through client modules that run in web browsers. The client modules are coded in browser-supported language (such as HTML, Java, ASP, PHP etc.). The client module connects to application module over

* The department is funded by UGC's DRS-Phase I under the SAP a network such as the internet or an intranet. In three tire web applications, the user provides query specification as input fields in predefined input form. These input values are used to construct SQL queries by the application server in the middle tire. Web applications are popular due to the ubiquity of web browsers, and the convenience of using a web browser as a client, sometimes called a thin client. The ability to update and maintain web applications without distributing and installing software on potentially thousands of client computers is a key reason for their popularity. Common web applications include web mail, online retail sales, online auctions, online banking, and many other functional applications. There are two types of web applications-

Presentation-oriented: A presentation-oriented web application generates interactive web pages in various types of markup language (HTML, XML, and so on) and containing dynamic content in response to requests.

Service-oriented: A service-oriented web application implements the endpoint of a web service. Presentation-oriented applications are often clients of service-oriented web applications.

Based on Mitre's Vulnerability statistics[10] (Evaluation from 2004-2006), there are five most reported vulnerability classes – SQL Injection, Cross Site Scripting(XSS), PHP File Inclusion, Buffer Overflow caused Denial of Service, and Directory Traversals. XSS and SQL Injection are consistently at or near the top 21.7% and 16% of the reported vulnerabilities in 2006-2007.

2. SQL INJECTION ATTACK (SQLIA):

SQL Injections are one of the most common and easiest techniques adopted by the attackers, to attack the web server, data server and sometimes the network. This category of attack is conducted by spammers for unauthorized web application access, breaking the role based accessibility and violating the integrity of the data storage. SQL injection attack (SQLIA) poses a serious threat to the security of web applications. Spammers conduct such attack by changing the developer's intended structure of an SQL command by inserting specially crafted input, in the form of SQL keywords and operators. Formal definition of SQL injection attack is given by Su and Wassermann [14].

3. CLASSIFICATION BASED ON VULNERABILITIES

Our literature survey reveals that the SQLIA can be classified into five basic classes based on the vulnerabilities in web applications.

3.1 Bypassing Web Application Authentication [1][6][11]

This is the most common usage adopted by the attackers to bypass authentication pages, used in web applications. In this category of attack, an attacker exploits an input field that is used in a query's 'where' condition part. An example is given below:

Example 1: Suppose there is an input form with the fields "username" and "password", using this user can retrieve his balance. The following PHP code for the application server, written by a web application developer has vulnerability for SQL injection attack:

```
[1]    $connection=mysql_connect();
[2]    mysql_select_db("test");
[3]    $user=$HTTP_GET_VARS['username'];
[4]    $pass=$HTTP_GET_VARS['password'];
[5]    $query="select balance from users where login='$user'
and password ='$pass'";
[6]    $result=mysql_query($query);
[7]    if (mysql_num_rows($result)==1)    echo "Authorized"
[8]     else   echo "authorization failed";
```

User data typed in a web form are assigned to variables "user" and "pass" and then used to obtain the SQL statement. Query (i) given below is generated after entering valid username 'devid' and valid password 'd123' by legitimate user.

Query = "select balance from users where login='devid' and password='d123'"; ---(*i*)

If an intruder types:′ or 1=1--′ in the username field leaving the password field empty, the structure of the SQL query will be changed. Query (ii) given below is generated with SQL injection by the attacker.

Query = "select balance from users where login='' or 1=1 --' and password=''"; --(i*i*)

Two dashes comment the following text. Boolean expression 1=1 is always true and as a result user will be logged with privileges of the first user stored in the table users.

3.2 Getting Knowledge of Database Fingerprinting [1][2][11]

This attack is considered as pre-attack preparation by an attacker. This category of attack is performed by entering some inputs by which it generates an illegal or the logically incorrect queries. The error messages reveal the names of the tables and the columns that cause error. The attacker also comes to know about the application database used in the backend server. Following is an example:

Example 2: An attacker enters as input "convert(select host from host)". The resulting query with respect to the PHP code given in (i), is shown below:

Query = "select * from users where login='devid' and password=convert(select host from host)"; Thus the injected query generated, first tries to execute the column host from host table. The host table consists of the information about the users privileges. The query tries to convert the host column data into an integer. As this is not a legal type conversion, the database server returns an error message as follows:

Error message by *MySQL* Server:

ERROR 1064: You have an error in your SQL syntax. Check the manual that corresponds to your MySQL server version for the right syntax to use near 'user='devid' and password= convert(select host from host)' at 1.

Thus, attacker will come to know the database used in server and the name of the columns. Sometimes it displays the table name also.

3.3 Injection with UNION query [1][11]

In such an attack, an attacker extracts data from a table which is different from the one that was intended in the web application by the developer. Following is an example:

Example 3: By entering an input "UNION SELECT <injected select query>", the runtime query generates with the SELECT query, with respect to the query given in Example 1, as -

Query = "Select * from users where login= ' ' UNION select balance_amt from customer_savings where account_no = 622289 -- and password='d123'"; In this example the actual runtime query returns null data, however, the injected query generates data from the table customer_savings. Some of the database applications return balance amount along with the user details.

3.4 Damaging with additional injected query [1][6][11]

This category of attack is generally very harmful. An attacker enters input such that an additional injected query is generated along with the original query. Following is an example:

Example 4: If the attacker inputs ″ ′ ; drop table user – ″ into the password field, the corresponding runtime query with respect to the query in the Example 1 generates -

Query = "select * from users where login=′devid′ and password= ′ ′ ; drop table user"; When the database server executes the second injected query, a harmful operations may also be performed on the database with such injected query(s).

3.5 Remote execution of stored procedures [1][3][4][6][8][11]

This category of attack is conducted by executing the procedures, stored previously by the web application developer. Following is an example

Example 5: Entering ″ ′ ; SHUTDOWN; -- ″ to any of the input field, the query generates as given below:

Query = "Select accounts from users WHERE login=′ ′ ; SHUTDOWN; -- password=′ ′;

Based on the above classification, the attack detection and prevention approaches are reported in the next section.

4. SQLIA DETECTION AND PREVENTION APPROACHES

Approaches for detection of SQL injection attack can be categorized into – pre-generated and post-generated. Runtime or dynamic or post generated approaches are useful for analysis of dynamic SQL query, generated by web application. Before posting a query to the back tire or database server for execution, analysis, and detection followed by blocking or correction of the query is done. Pre-generated or static approaches are desirable during the testing phase of software. While developing the web applications, programmers should followed some steps for SQL injection attack (SQLIA) detection. An effective validity checking mechanism for the input variable data is also a requirement for the pre-generated technique of the detection of SQLIA. Some of the existing post generated and pre-generated approaches implemented for detection of SQLIA, are cited below.

4.1 Post-Generated Approach

We discuss three popular detection mechanisms of SQLIA using post-generated approach.

4.1.1 Positive Tainting and Syntax Aware Evaluation[13]:

In this approach valid input strings are initially provided to the system for detection of SQLIA. At runtime, it categorizes input strings and propagates the untrusted or other-than-trusted markings based on the initialization. After that, a 'syntax aware evaluation' is performed for evaluating the propagated strings. Thus, based on the evaluation, if untrusted strings are found, such queries are restricted from passing into the database server for processing. During initialization of the trusted strings, it performs identification and marking based on inputs. The strings are categorized as: (i) hard coded strings, (ii) strings implicitly created by Java and (iii) strings originated from external sources. In case of syntax-aware evaluation, it performs syntax evaluation at the database interaction point. Syntax defines the trust policies which are the functions defined by the web programmer. Functions perform pattern matching and if the result of matching gives positive outcome, the tool allows the query to be executed on the database server. Following issues are there in this method - (i) Initialization of trusted strings are developers dependent and (ii) Persistent storage of trusted strings may cause second order attack[1].

4.1.2 Context Sensitive String Evaluation (CSSE) [12]:

The basic idea behind this approach is to find out the root cause of SQLIA. The root cause is the origin of the data (information about the data, termed as metadata) i.e., user-provided or developer-provided. Thus, any data provided by the user is marked as untrusted and data provided by the applications are termed as trusted. The untrusted metadata are used for syntactic analysis based on 'Context Sensitive String Evaluation (CSSE)'. Injection vulnerabilities may also occur due to programming flaws during developments. CSSE is basically based on syntactical analysis, which first distinguishes string constants (e.g., select * from users where login='$login_name') and numerical constants (e.g., select * from users where pin=$pin). It then removes all unsafe characters (un-escaped quotes) in alphanumeric identifiers and non-numeric characters in numeric identifiers. This operation is performed before sending the query to the database server. Following issues are there in this approach - (i) Initialization of the unsafe characters is dependent on the web programmer, and (ii) Removal of unsafe characters restricts the application functionality.

4.1.3 *Parse tree evaluation based on grammar [14]:*

The basic idea of this method is to block those queries generated from user input, which defy the syntactic structure of the query, as defined by the developer. SQL queries generated at runtime are parsed based on a pre-defined grammar. Runtime SQL generated is parsed based on the grammar. Special literals '(|' and '|)' are used to mark the beginning and end of each input string. Each such string within markers-pairs is matched with the augmented grammar constructed for the purpose. If the query parses successfully, it meets the syntactic constraints and is declared as legitimate. Otherwise, it is declared as illegitimate and is blocked. A major issue of this method is that an attacker may manipulate the input string by entering the marking symbol ′ |)′ . Thus the syntactical confinement of the string surrounding with ′(|′ and ′|)′ may be affected.

4.2 Pre-Generated Approach

4.2.1 Pixy : A Static Analysis Tool [7]:

Pixy is used for detecting web application vulnerabilities. It is based on statistical approach which uses data flow analysis for detecting tainted data i.e. data entered by malicious user. Using a set of suitable operations, tainted data can be sanitized, i.e. the harmful properties can be removed. The authors assume that the SQLIA occurs only due to concrete values of some parameters. Identifying such parameters and their removal makes the application free from SQLIA. Data flow analysis has been applied to statistically compute certain information for every single program point. A parse tree is developed based on the input from the users and a taint analysis tool is used to identify the points where tainted data can enter into the program. It then propagates the tainted values along the assignments and similar constructs in the program. This tool also performs alias analysis to handle the effect of tainting other aliases. A literals analysis gives knowledge about the literal values that variables and constants may hold at each program point. A major issue of this method is that- since Pixy is an open source tool, an attacker may have scope(s) to bypass it by exploiting the features available in it.

4.2.2 Program Query Language(PQL) [9]:

A PQL is developed specially for web application programmers to retrieve attack related queries. It also incorporates a static technique which finds the solutions to such attack related queries. The static analyzer finds all potential matches conservatively using context-sensitive as well as flow-insensitive analysis. This static result guides the runtime or dynamic analysis. A PQL is a pre-defined grammar based language. It has query variables (arguments), statements (primitive, compound), subqueries (recursive event sequences or recursive object relations) reacting to match (print or abort etc.). A static checker and optimizer, translates by PQL into queries. The translation of the PQL into 'datalog' (another more expressive language) provides sufficient support to programs to resolve the attack related queries. One of the major issues of this method is that resolving the attack related queries, based on the output (available with the datalog) is mostly developer dependent.

Based on our survey, we observe that -

- The pre-generated detection techniques for SQLIA depend on the effectiveness of the validity checking by the web programmer and the effectiveness of the tools applied to detect the integrity of the code that causes SQLIA.
- The post generated approaches for detection of SQLIA are based on the initialization of trusted or untrusted strings, which are developer-dependent. The approach given in [14] may be considered better than the others. However, it also has the possibility of manipulation of the strings by the attacker.

In the next section we present an effective matching mechanism DUD, for dynamic SQLIA detection which can successfully overcome the shortcomings of the above methodologies.

5. DUD: AN SQLIA DETECTION APPROACH

Our proposed strategy for SQLIA termed DUD, is a post generated approach based on query classification. DUD is dependent on a user defined threshold Є. It first generates SQL Master File(SQLMF) for a web application. The SQLMF for a web application consists of the legitimate distinct SQL queries generated dynamically. The flow diagram is shown in figure 1.

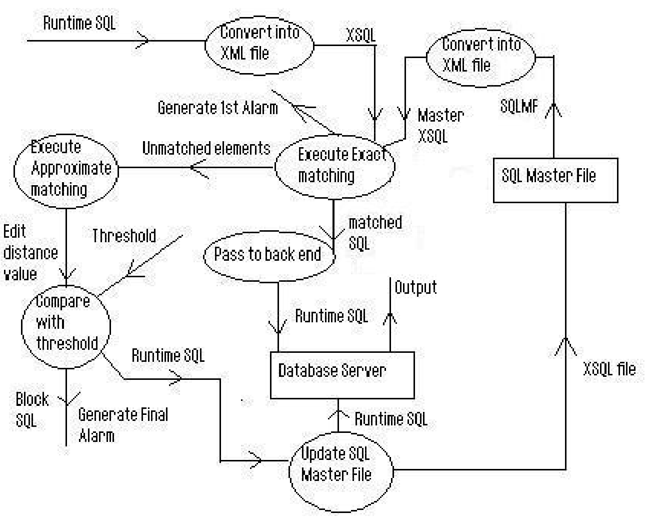

Figure 1 Flow diagram of our proposed solution to SQLIA Detection and Blocking

The matching mechanism used in DUD consists of the following steps -

```
(1) Read a dynamic or runtime input SQL and convert it into XML form called XSQL;
(2) Initiate exact matching process with SQLMF and XSQL -
        (a) if XSQL∈SQLMF, then
                - declare as 'Safe Query' ;
                - pass to the database server for processing and go to step (5);
        (b) else generate 'attack alarm 1';
                - call approximate matching() and store the result, Edit_Distance in υ;
(3) If υ > Є then
            { - generate 'attack alarm final';
              - block the query from passing to the database server; }
(4) Else
            { - allow the query to pass to the database server for processing;
              - update SQLMF with input SQL;}
(5) Stop;
```

The mechanism used in DUD is fast and effective due to the following points:

- Avoids the initialization of trusted/untrusted strings/characters;
- Easy to implement matching logic;
- SQL master file is adaptively updated;
- No restriction is imposed on user input strings/characters;
- Developer independent;

It is considered that the structure of the dynamically generated SQL queries, are not constant. Thus, we have converted each SQL query into XML form before initiating the matching process of DUD. SQL queries in XML form are defined as XSQL. DTD (Document Type Definition) of the XML equivalent of an SQL query is given in figure 2 below:

```
<!Element Select (attribute)*
<!Element From (table)*
<!Element Where (expression, (Logical_Operator, Expression)*)
<!Element Expression ( Identifier, Relational_Operator, Value)
<!Element Logical_Operator(AND|OR|NOT)
<!Element Identifier(#pcdata)
<!Element Relational_Operator (= | != | < | > | ≤ | ≥ |)
<Element Value (#cdata)
```

Figure 2 Document type definition (DTD) of XML equivalent of SQL query

An automatic parser based on the DTD given in figure 2 convert SQL query into XML form. For query given in example 1, the XML equivalent called XSQL is shown in figure 3.

```
<select>
<attribute   attribute_name=balance      </attribute>
<from>
<table     table_name=users   </table>
</from>
<where>
<expression>
<identifier            identifier_name  =  use  </identifier>
<relational operator  relational operator =  = </relational
operator>
<value     value = abc            </value>       </expression>
<logical operator    logical  operator  =  AND    </logical
operator>
<expression>
<identifier identifier_name=password   </identifier>
<relational operator  relational  operator   = = </relational
operator>
<value     value = ab123         </value>
</expression>
</where>
</select>
```

Figure 3 XML record based on DTD given in figure 1

5.1 Algorithms

The text content of SQLMF(T) are a set of *n* number of legitimate SQL queries (where, $1 \leq n$). Each query is expressed as a sequence of elements $\{s_1, s_2, .., s_{n'}\}$. Each element is a string of characters. The text pattern (*P*) of the dynamic query, is expressed as one or more elements $\{s'_1, s'_2, .. s'_{n''}\}$, where, $1 \leq n''$. An element may have one or more sub elements, identifiers and values. A function element_count(*P*) computes the number of elements in *P*. Each element is separated from others by semicolon (;). The algorithms for performing exact matching and approximate matching are given in 5.1.1 and 5.1.2 below:

5.1.1 Exact Matching:

```
Input   : T, P
Output:  Safe Query, Attack Alarm I
[a]  match_count ← 0;
[b] For i= 1 to n do
Begin
[d] If (P=T_i) then    {
        -    Add 1 to match_count;
        -    Declare ‘Safe Query’;
        -    Exit;    }
[e] End if;
[f] End For Loop;
[g] If (match_count=0) then {
        -    Declare ‘Attack Alarm I’;
        -    Call Approximate Matching;      }
[h] Stop;
```

5.1.2 Approximate Matching:

```
Input   : T, P, Є
Output:  Safe Query, Attack Alarm Final
[a]  k = element_count(P);
[b] For i  = 1 to n do  {
[d] For j = 1 to k do   {
[d] If  (P[j] ≠ T[i][j]) then
[e] D[i] ← D[i] + 1 ;
[f] Enf if ;    }   }
[e] Edit_Distance ← 0 ;
[f] For i = 1 to n do  {
[g] Edit_Distance = MIN (D[i]);  }
[h] If (Edit_Distance < Є) then  {
            - Declare ‘Safe Query’ ;
            - Execute P;  }
[h] Else   {
[i]         - Declare ‘Attack Alarm Final’ ;
[j]         - Block P;            }
[k] End if;
[l]  Stop;
```

5.2 Architecture of DUD

The architecture of DUD is shown in figure 4. The detection module has installed in web server in three tire architecture of web application system. The web server is connected with database server and is accessible to web clients.

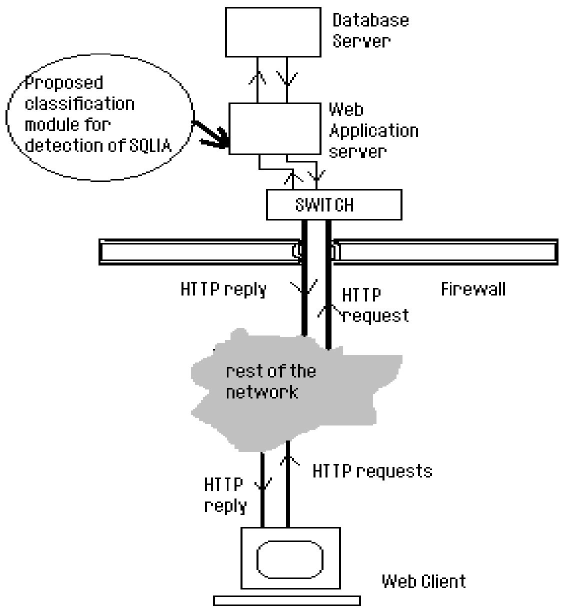

Figure 4 Architecture of the proposed approach implementation

6. EXPERIMENTAL RESULTS

We implemented our method DUD in a three tire web application having *MySQL* database in back tire. The middle tire is configured as web application server. We tested the proposed DUD in a simulated environment using three files file 1, file2 and file 3. File 1 contains SQL statements due to legitimate queries to one of the web application packages of our University intranet web server. File 1 is considered as SQLMF. File 2 contains SQL injected queries due to queries from attackers. The third file - file 3 contains SQL queries and considered as input from legitimate users. At first file 1 and file 3 are matched using DUD. Then file 1 and file 2 are matched. In table 1, table 2 and table 3, a sample shot of file 1, file 2 and file 3 are shown. The file 3 consists of some new legitimate queries not available in the file 1. The '*edit distance*' value while matching file 1 with file 3 are found to be zero. It may be mentioned that while matching we have avoided the null value ' '. We found the minimum *edit distance* value 12, while matching between file 1 and file 2. Thus, from this experiment we can come to the initial conclusion that attackers' SQL injected queries has edit distance value at least 12.

7. CONCLUSION AND FUTURE WORK

In this paper, we have presented a survey on different classes of SQLIA and some of the important approaches for detection of SQLIA. We have also presented a new technique DUD for detection of SQLIA. The detection result produced, are based on the simulated experiment. The *edit distance* value may be changed with the matching of more and more injected queries. Thus, the threshold value for detection of SQLIA based on matching is an empirical one. To avoid stolen key attack on SQLMF, the encoding of SQLMF is required. To increase the efficiency of matching, the indexing of SQLMF is required. To accommodate the variable structure of dynamically generated query based on user input, the SQL query is converted into XML for before initiating the matching process of DUD. Thus, future evaluation work should focus on efficiency of the matching technique. More experimentation is required for generating an adaptive threshold value. Empirical evaluations such as those presented in related work with more real life dataset would allow for comparing the performance of DUD. While generating SQLMF for a web application, it is recommended that it should have at least one query with respect to its one application.

Table 1. A Sample shot of SQL master file stored in File 1 constructed with legitimate query of a web application

Query id	Actual Query
1	Select emp_code from employee where user='mks' and password='mk123'
2	Select emp_code from employee where user='ddas' and password='deb_dd'
3	Select emp_code from employee where user='rgos' and password='rg123'
4	Select basic,da,hra from allowance where emp_code=102 and user='pkb' and password='pk421'
5	Select pf, ptax, itax from deduction where emp_code=102 and user='pkb' and password='pk421'
6	Select basic,da,hra from allowance where emp_code=221 and user='sis' and password='si128'
7	Select pf, ptax, itax from deduction where emp_code=221 and user='sis' and password='si128'

Table 2. A Sample shot of SQL File 2 constructed with illegitimate or SQL injected query of a web application

Input Query
Select emp_code from employee where user='ddas' and password=' ' or 1=1
Select basic, da, hra from allowance where login='devid' and password = convert (int(select top 1 tname from syscatalog where tabletype='u'))";
Select pf, ptax, itax from deduction where login=' ' UNION Select balance from account where emp_code=122;
Select * from balance where login=' ' and password=' '; drop table user;
Select * from balance where login=' '; SHUTDOWN; and password=' ' or 1=1;

Table 3. A Sample shot of SQL File 3 constructed with dynamic SQL query generated from legitimate input of a web application

Input Query
Select emp_code from employee where user='ddas' and password='deb_dd'
Select basic,da,hra from allowance where emp_code=221 and user='sis' and password='si128'
Select basic,da,hra from allowance where emp_code=441 and user='dkr' and password='dk987'

8. REFERENCES

[1] C Anley. Advanced SQL Injection in SQL Server Applications. White Paper Next Generation Security Software Ltd., 2002. http://www.nextgenss.com/papers/advanced sql injection.pdf

[2] D. Litchfield. Wep Application Dissembly with ODBC Error Messages. Technical document, @Stake,Inc.,2002.

[3] E. M. Fayo. Advanced SQL Injection in Oracle Databases. Technical Report, Agencies Information Security, Balck Hat Briefings, Black hat U.S.A., 2005

[4] F. Bouma. Stored Proceduresare Bad, O'Kay? TechnicalReport. Asp.Net Weblogs, November 2003. http://weblogs.asp.net/fbouma/archive/2003/11/18/38178.aspx

[5] K. Krithivasan and R. Sitalakshmi, "Efficient Two imensional Pattern Matching in the Presence of Errors", Information Sciences, Vol. 43, 1987, pp. 169-184.

[6] M. Howard and D Le Blane. Writing Secure Code. Microsoft Press, Redmond, Washington, second edition, 2003.

[7] Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilitiesby Nenad Jovanovic , Christopher Kruegel , Engin Kirda, IN 2006 IEEE SYMPOSIUM ON SECURITY AND PRIVACY

[8] P. Finnigan. SQL Injection and Oracle – Part 1 and Part 2. Technical Report, Security Focus, November 2002.

[9] Finding Application Errors and Security Flaws Using PQL: a Program Query Language. OPSLA'05, October 16-20, 2005, San Diego, California, USA

[10] Steve Christey. Vulnerability Type Distributions in CVE, October 2006. http://cwe.mitre.org/documents/vuln-trends.html

[11] S.McDoland. SQL Injection. Modes of Attack, defence and why it matters. White paper, GovernmentSecurity.org, April 2002

[12] Tadeusz Pietraszek and Dhris Vanden Berghe. Defending against Injection Attacks through Context-Sensitive String Evaluation. Proceedings of Recent Advances in Intrusion Detection (RAID2005).

[13] William G.J. Halfond, Alessandro Orso and Panagiotis Manolios. Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks. . SIGSOFT'06/FSE-14, November 5-11, 2006, Portland, Oregon, USA.

[14] [Z.Su and G. Wassermann. The Essence of Command Injection Attacks in Web Application. In the 33rd Annual Symposium on Principles of Programming languages, pages 372-382, Jan. 2006.