Detecting SQL Injection Attacks Using Syntax Analysis of Dynamically Generated Queries

[1]Supriya Gupta, [2]Dr Lalitsen Sharma
[1]PhD Student, [2]Associate Prof.
Dept. of Com Sc & IT, University of Jammu
e-mail: mangotwin22@gmail.com, lalitsen@yahoo.com

Abstract: *Web applications are popular targets of security attacks. One common type of such attacks is SQL injection, where an attacker using specially crafted inputs, causes a web application to generate and send a query that functions differently than the programmer intended. Thus a diagnostic feature of SQL injection attacks is that they change the intended syntactic structure of queries issued. This paper presents a query intent evaluation technique to detect possible SQL Injection attacks by tracing the queries in which the input substrings modify the syntactic structure of the rest of the query. This approach has been implemented in a tool which takes an SQL query as input and detects if it is a command injection attack.*

Introduction

The ubiquity and popularity of World Wide Web has attracted the developers to develop applications web based. In a competition to develop online services for general public, web applications have often been deployed with minimal attention to security risks, as a result most applications are vulnerable to attacks [2]. SQL injection attacks are one of the topmost threats for web applications. An SQL injection attack targets interactive web applications that employ database services. Such an application accepts user input, such as form fields, and then includes this input in database requests by constructing database queries dynamically, and then dispatches

these queries over an API to appropriate databases for execution. In such a way, a web application retrieves and presents data to the user based on the user's input as part of the application's functionality. However, if the user's input is not handled properly, serious security problems can occur. This is because queries are constructed dynamically in an ad hoc manner through low-level string manipulations. This is ad hoc because databases interpret query strings as structured, meaningful commands, while web applications often view query strings simply as unstructured sequences of characters. This semantic gap, combined with improper handling of user input, makes web applications susceptible to a large class of malicious attacks known as SQL command injection attacks (SQLCIA).
For example, if a database contains user names and passwords, the application may contain code such as the following:

```
String query = "SELECT * FROM accounts WHERE
name='" + request.getParameter("name") + "' AND
password='" + request.getParameter("passwd") + "'";
```

Figure 1

The code in Figure 1 generates a query intended to be used to authenticate a user who tries to login to a web site. However, if a malicious user enters admin into the name field and 'OR' a'='a. into the password field, the query string generated is shown in figure 2, whose condition always evaluates to true, and the user will bypass the authentication logic.

```
SELECT * FROM accounts WHERE name='admin'
AND password= '' OR 'a'='a'
```

Figure 2

SQL injection attacks are extremely prevalent, and ranked as the second most common form of attack on web applications in 2010 [1]. The percentage of these attacks among the overall number of attacks reported rose from 5.5% in 2004 to 14% in 2006 to 24% in 2008[4]. Out of various vulnerabilities reported in 2010, SQL injection vulnerabilities amount significantly with 18%

and they continued to make up the largest percentage of the reported vulnerabilities [1]. The SQLCIA on Card Systems Solutions Inc. [3] that exposed several hundreds of thousands of credit card numbers is an example of how such attack can victimize an organization and members of the general public.

Related Work

Research on SQL injection attacks can be broadly classified into two basic categories: 1) vulnerability identification approaches and 2) attack prevention approaches. The former category consists of techniques that identify vulnerable locations in a web application that may lead to SQL injection attacks. In order to avoid SQL injection attacks, most of these protections rely on traditional signature detection techniques which inspect web traffic to identify SQL injection-related text patterns. A reasonably sized signature database does not provide reliable protection while a comprehensive signature database results in excessive management overhead, dramatic performance limitations, and false positives [6, 7]. The static analysis techniques for vulnerability identification presented in [8, 9, 15, 16] are based on tainted information flow tracking. They require manually written specifications, either for each query or for bug patterns and they are not fully automated and may require user intervention at various points in the analysis. They do not provide any way to check the correctness of the input validation routines, and programs using incomplete input validation routines may indeed pass these checks and still be vulnerable to injection attacks.

A much more satisfactory treatment of the problem is provided by the class of attack prevention techniques that retrofit programs to shield themselves against SQL injection attacks [10, 11, 12, 13,]. In order to avoid SQL injection attacks, a programmer often subjects all inputs to input validation and filtering routines that either detects attempts to inject SQL commands or sans the input. Relying on input validation routines as the sole mechanism for SQL injection defense is problematic. Although they can serve as a first level of defense, it is widely agreed [13] that they cannot defend against sophisticated attack techniques. For example, if an application

forbids the use of the single-quote in input, SQLCIAs may still be possible because numeric literals are not delimited with quotes. The problem is that web applications generally treat input strings as isolated lexical entities. Input strings and constant strings are combined to produce structured output (SQL queries) without regard to the structure of the output language (SQL).

A more fundamental technique to the problem of defending SQL injection comes from the commercial database world, in the form of PREPARE statements [14]. These statements allow a programmer to declare and finalize the structure of every SQL query in the application. Once issued, these statements do not allow malformed inputs to further influence the SQL query structure, thereby avoiding SQL vulnerabilities altogether. This is in fact a robust mechanism to prevent SQL injection attacks; however, if other portions of the query are being built up with unescaped input, SQL injection is still possible. Also retrofitting an application to make use of PREPARE statements requires manual effort in specifying the intended query at every query point, and the effort required is proportional to the complexity of the web application.

Overview of the Approach

In this paper the problem is approached by tracking through the program the substrings from user input and restricting those substrings syntactically. A web application generates a query by combining filtered inputs and constant strings [15].

```
String query = “SELECT * FROM accounts WHERE
name=’” + sanitizedName + “‘ AND password=’” +
paswd + “‘“;
```

Figure 3

For example in figure 3, “sanitizedName” is a filtered input, and “SELECT * FROM accounts” is a constant string for building dynamic queries. If the user inputs admin as his user name and password into the password field, the query string generated is shown in figure 4.

String query = “SELECT * FROM accounts WHERE name= ‘admin’ AND password= ‘ password ‘“;

Figure 4

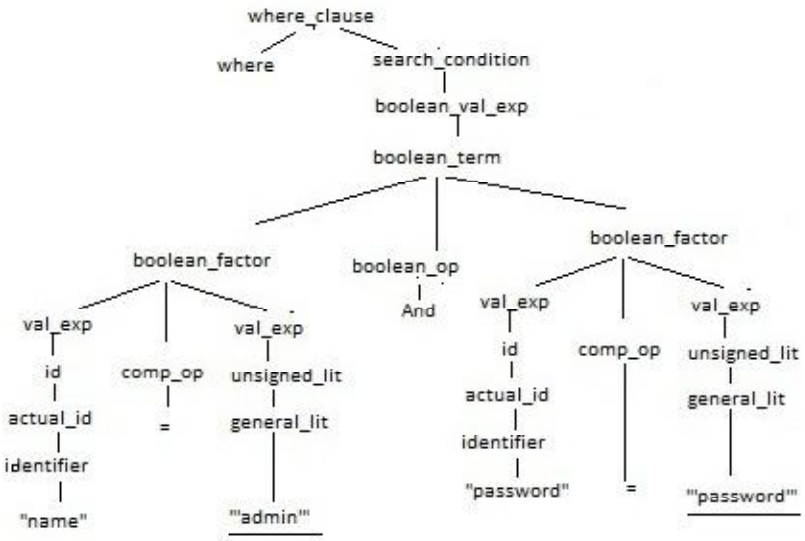

Figure 5a: Parse tree for where clause of query in figure 4. Sub strings from user input are underlined

A malicious user may replace the password in the input with ’OR’ a’= ‘a in order to bypass the authentication logic, the generated query is represented in figure 2. Figure 5 shows a parse tree for each query. For demonstration purpose, consider the simplified grammar for SQL SELECT statement’s where_clause in Figure 7(a). This is the grammar used to generate the parse trees in Figure 5. Note that in Figure 5a, for each substring from input there exists a node in the parse tree whose descendant leaves comprise the entire input substring and no more: general_lit the first substring and general_lit for the second, as shown with underline.

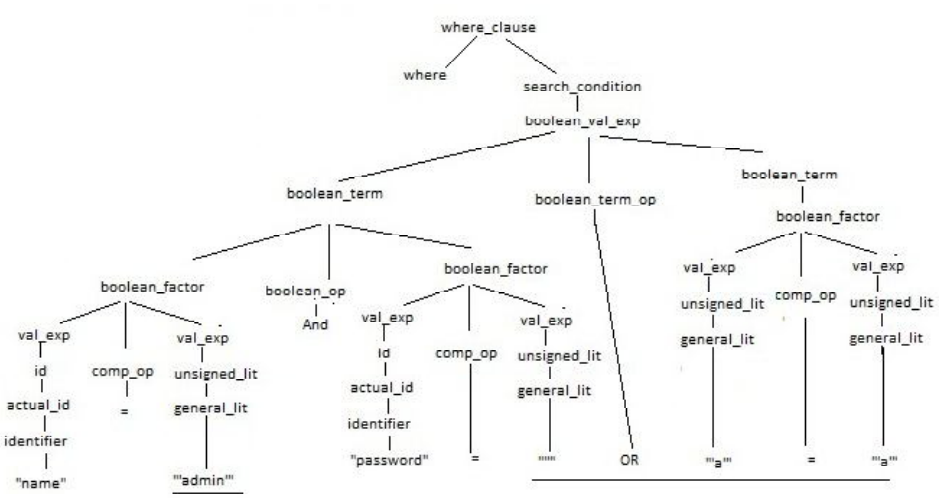

Figure 5b: Parse tree for where clause of query in figure 2. Sub strings from user input are underlined

No such parse tree node exists for the second input substring in Figure 5b. This distinction is common to all examples of legitimate vs. malicious queries that we have seen. The intuition behind this distinction is that the malicious user attempts to cause the execution of a query beyond the constraints intended by the programmer, while the normal user does not attempt to break any such constraints. This distinction is used to identify queries in which the input substrings change the syntactic structure of the rest of the query. Such queries are *command injection attack.* To track user's input we use meta-data, displayed as '&' and '&,' to mark the beginning and end of each input string so that when a query is ready to be sent to the database, it has matching pairs of markers identifying the substrings from input. This annotated query is called an augmented query. Figure 6 shows

instance of augmented query for the queries generated in figure 4 and figure 2 respectively.

```
SELECT * FROM accounts WHERE name=& 'admin'&
AND password=& ' password'&

SELECT * FROM accounts WHERE name= &'admin'&
AND password= &'' OR 'a'='a'&
```

Figure 6: augmented queries

In order to forbid input substrings from modifying the syntactic structure of the rest of the query we construct an *augmented grammar* for augmented queries based on the standard grammar for SQL queries [17]. The algorithm for constructing augmented grammar is presented in [19].In the augmented grammar, the only productions in which '&' occur have the following form: nonterm ::= &symbol &, where *symbol* is either a terminal or a non-terminal.

```
where_clause ::=          "where" search_condition
search_condition ::=      boolean_value_exp
boolean_value_exp ::=     { boolean_term, boolean_term_op } ;
boolean_term_op ::=       "or" ;
boolean_term ::=          { boolean_factor, boolean_factor_op } ;
boolean_factor_op ::=     "and"
boolean_factor ::=        { val_exp , comp_op }
val_exp ::=               unsigned_lit|id
unsigned_lit ::=          unsigned_num_lit |' general_lit' ;
id ::=                    ["_" char_set_spec] actual_id ;
actual_id ::=             identifier ;
comp_op ::=               = | < | > | <= | >= | !=
```

Figure 7a:Simplified grammar for where clause

where_clause ::=	“where” search_condition
search_condition ::=	boolean_value_exp
boolean_value_exp ::=	{ boolean_term, boolean_term_op } ;
boolean_term_op ::=	“or” ;
boolean_term ::=	{ boolean_factor, boolean_factor_op } ;
boolean_factor_op ::=	“and”
boolean_factor ::=	{ val_exp , comp_op }
val_exp ::=	unsigned_lita | id
unsigned_lita ::=	unsigned_lit| “&” unsigned_lit “&”;
unsigned_lit ::=	unsigned_num_lit | ‘general_lit’ ;
id ::=	[“_” char_set_spec] actual_id |
	actual_ida ;
actual_ida ::=	“&” actual_id “&” ;
actual_id ::=	identifier
comp_op::=	= | < | > | <= | >= | !=

Figure 7b:Simplified augmented grammar for where clause

For an augmented query to be in the language of this grammar, the substrings surrounded by ‘&’ must be syntactically confined. Figure 7 shows the **s**implified grammar for the where_clause and the corresponding augmented grammar (Figure 7b).

Now consider the augmented queries shown in Figure 6.Using the augmented grammar, the parse tree for the first query would look the same as Figure 5a, except that the sub trees shown in Figures 8 would be substituted in for the first and second input strings, respectively. No parse tree could be constructed for the second augmented query.

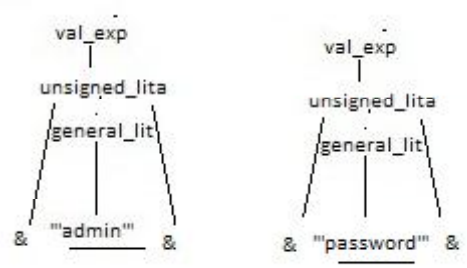

Figure 8

Therefore to prevent invalid queries augment the query and then attempt to parse the augmented query with the augmented grammar. If the query fails to parse it is invalid.

We use a parser generator ProGrammar [18] to build a parser for the augmented grammar and attempt to parse each augmented query. If the query parses successfully, it meets the syntactic constraints and is legitimate. Otherwise, it fails the syntactic constraints and either is a command injection attack or is meaningless to the interpreter that would receive it. A tool called SQL Injection Detector is written in visual basic 6.0 using the augmented grammar of the SQL language and a policy specifying permitted syntactic forms. Each input that is to be propagated into some query, regardless of the input's source, is augmented with the meta-character '&' and generated augmented query, is inputted which our tool attempts to parse. If a query parses successfully, the query is safe and otherwise it is an SQL command injection attack. Figure 8 shows the result obtained by SQL Injection Detection tool for the second augmented query in figure 4. The tool is tested with different forms of injection attacks and is successful in detecting SQLCIAs with 0 false positives.

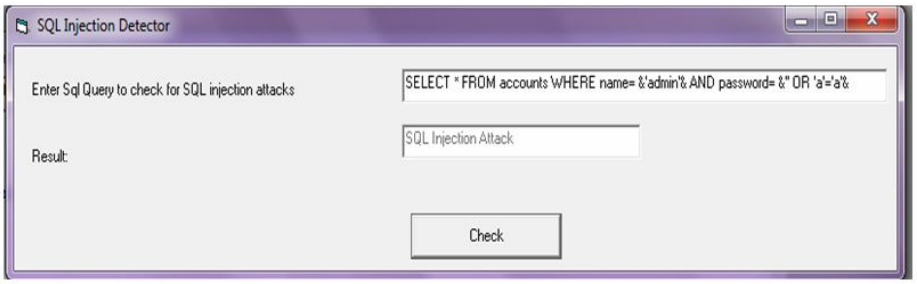

Figure 8: Screenshot SQL Injection Detector tool.

Acknowledgment

The authors are thankful to University Grants Commission (UGC), and Ministry of Human Resource Development (MHRD), Government of India for providing financial assistance to carry out this research work. The authors are also thankful to Prof. Devanand, Head, Department of Computer Science and IT, University of Jammu, for his kind support.

References

1. Cenzic Web Application Security Trends Report – Q3-Q4, 2010, Cenzic Inc. Retrieved on on 9-8-2011 from http://www.cenzic.com/ Cenzic_AppSecTrends_Q3-Q4-2010.pdf.
2. Pettit S. (2001). Anatomy of a Web Application: Security Considerations. Sanctum, Inc.
3. Kerk J. (2006). SQL injection attacks against databases rise sharply: Legal issues. Retrieved on 9-8-2011 from http://www.computerworld.com/s/article/ 9001878/ SQL_injection_attacks_against_databases_rise_sharply
4. Cenzic Web Application Security Trends Report – Q3-Q4, 2008, Cenzic Inc. Retrieved on 9-4-2011 from http://www.cenzic.com/Cenzic_AppSecTrends_Q3-Q4-2008.pdf.
5. Anley, C (2002). Advanced SQL injection in SQL server applications, White paper, Next Generation Security Software Ltd. Tech. rep.

6. O. Maor and A. Shulman (2002). SQL injection signatures evasion. White paper, Imperva. Tech. rep.

7. Xie, Y., and Aiken (2006), A. Static detection of security vulnerabilities in scripting languages. In USENIX Security Symposium.

8. Livshits, V. B., and Lam, M. S (2005). Finding security vulnerabilities in Java applications with static analysis. In USENIX Security Symposium.

9. Halfond, W., and Orso, A (2005). AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In the Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering (ASE 05), pp. 174–183.

10. Valeur, F., Mutz, D., and Vigna, G (2005). A learning-based approach to the detection of sql attacks. In the Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA*)*, Vienna, Austria, pp. 123–140.

11. Nguyen-Tuong A.,Guarnieri S., Greene D., Shirley J. & D. Evans (2005). Automatically hardening web applications using precise tainting. 20th IFIP International Information Security Conference ,pp. 295–308.

12. Pietraszek, T., and Berghe, C. V (2005). Defending against injection attacks through context-sensitive string evaluation. In the Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID).

13. Buehrer, G., Weide, B. W., and Sivilotti, G (2005). Using parse tree validation to prevent sql injection attacks. In Proceedings of the 5th international workshop on Software engineering and middleware (SEM 05).

14. Prepared statements and stored procedures, Retrieved on 9-8-2011 from http:/ /php. net /manual/en/pdo .prepared -statements.php

15. Wassermann G., Su Z (2007). Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI'07).

16. Wassermann G., Su Z. (2004, May). An Analysis Framework for Security in Web Applications. Department of Computer Science, University of California.

Specification and Verification of Component Based Systems Workshop at ACM SIGSOFT 2004 (SAVCBS'04).

17. BNF Grammars for SQL-92, SQL-99 and SQL-2003, Retrieved on 9-8-2011 from http://savage.net.au/SQL/
18. ProGrammar, Parser Development Toolkit. http://www.programmar.com/
19. Wassermann G., Su Z. (2006). The essence of command injection attacks in web applications. In the Proceedings of the 2006 POPL Conference.