Using Parse Tree Validation to Prevent SQL Injection Attacks

Michele Spagnuolo
Politecnico di Milano,
Milan,
Italy,
michele@spagnuolo.me

July 1, 2011

Abstract

An SQL injection attack targets interactive web applications that employ database services. Such applications accept user input, such as form fields, and then include this input in database requests, typically SQL statements. In SQL injection, the attacker provides user input that results in a different database request than was intended by the application programmer. That is, the interpretation of the user input as part of a larger SQL statement, results in an SQL statement of a different form than originally intended. We describe a technique to prevent this kind of manipulation and hence eliminate SQL injection vulnerabilities. The technique is based on comparing, at run time, the parse tree of the SQL statement before inclusion of user input with that resulting after inclusion of input. We wrote a simple Bison **grammar** for a subset of SQL and a **lexer** in Flex, then a **PHP frontend** that presents the user differences between parse trees of two queries: a *reference query* and a *query to test.*

Based on *Using Parse Tree Validation to Prevent SQL Injection Attacks* **by Gregory T. Buehrer, Bruce W. Weide, and Paolo A. G. Sivilotti.**

1.2 SQL injections defined

SQL injection is a type of security attack whereby a malicious user's input is parsed as part of the SQL statement sent to the underlying database. Many variations exist, but the central theme is that the user manipulates knowledge of the query language to alter the database request. The goal of the attack is to query the database in a manner that was not the intent of the application programmer.

1.3 Our Bison grammar

We wrote a simple yacc/Bison grammar for a subset of SQL. We could have used real-world production-ready grammars from open source projects such as PostgreSQL[1] or MySQL[2], but they are too complex for our purpose. Semantic actions are C functions to build the tree.

Parser.y

```
%{

#include <stdlib.h>
#include <string.h>
#include "TypeParser.h"
#include "ParserParam.h"

static SExpression *e;

SExpression* assign(SExpression* element) {
SExpression* temp = (SExpression *) malloc(sizeof(SExpression));
memcpy(temp, &element, sizeof(element));
return temp;
}

SExpression* createHead(SExpression *firstchild) {
SExpression* head = (SExpression *) malloc(sizeof(SExpression));

if ( NULL != head ) {
strncpy(head->m_type, "start", sizeof(head->m_type));

head->m_first = firstchild;
head->m_second = NULL;
head->m_third = NULL;
head->m_fourth = NULL;
head->m_fifth = NULL;
}
```

[1]yacc grammar can be found in src/backend/parser/, filename: gram.y

[2]http://www.w3.org/2005/05/22-SPARQL-MySQL/sql_yacc

```
return head;
}

%}

%token <strval> NAME
%token <strval> STRING
%token <intval> INTNUM

/* operators */

%left OR
%left AND
%left NOT
%left LESSER GREATER EQUALS
%left PLUS MINUS
%left TIMES DIV

/* literal keyword tokens */

%token BY
%token CREATE
%token DELETE
%token DISTINCT
%token DROP
%token FROM
%token INSERT
%token INTEGER
%token INTO
%token NULLX
%token ORDER
%token STR20
%token SELECT
%token TABLE
%token VALUES
%token WHERE
%token SEMICOLON
%token PERIOD
%token COMMA
%token EOL
%token LPAREN RPAREN

%type <expression> sql table statement drop_table_statement insert_statement
%type <expression> create_table_statement select_statement delete_statement
%type <expression> opt_where opt_column_commalist opt_orderby opt_distinct
%type <expression> search_condition boolean_term boolean_factor
%type <expression> boolean_primary comparison_predicate
%type <expression> base_table_element_commalist  values_or_query_spec
```

```
%type <expression> select_expr_di_list table_references select_expr_list
%type <expression> expr atom column_ref opt_column_ref base_table_element
%type <expression> column_commalist insert_atom_commalist insert_atom term

%start start

%%

start
: sql { e = createHead($1); }

sql: /* empty */ {$$ = createNode("sql", NULL, NULL, NULL, NULL, NULL); }
| sql SEMICOLON {$$ = createNode("sql", $1, NULL, NULL, NULL, NULL);
e = createHead($$); printTree(e, 1); }
| sql statement SEMICOLON {$$ = createNode("sql", $1, $2, NULL, NULL, NULL);
e = createHead($$); printTree(e, 1); }
;

statement:
        create_table_statement {$$ = createNode("statement", $1, NULL, NULL, NULL, NULL);}
        | drop_table_statement {$$ = createNode("statement", $1, NULL, NULL, NULL, NULL);}
| insert_statement {$$ = createNode("statement", $1, NULL, NULL, NULL, NULL);}
| select_statement {$$ = createNode("statement", $1, NULL, NULL, NULL, NULL);}
| delete_statement {$$ = createNode("statement", $1, NULL, NULL, NULL, NULL);}
;

drop_table_statement:
DROP TABLE table {$$ = createNode("drop_table_statement", $3, NULL, NULL, NULL, NULL);}
;

create_table_statement:
        CREATE TABLE table LPAREN base_table_element_commalist RPAREN
        {$$ = createNode("create_table_statement", $3, $5, NULL, NULL, NULL);}
        ;

insert_statement:
  INSERT INTO table opt_column_commalist values_or_query_spec
  {$$ = createNode("insert_statement", $3, $4, $5, NULL, NULL);}
  ;

select_statement:
SELECT opt_distinct select_expr_di_list
FROM table_references
opt_where
opt_orderby {$$ = createNode("select_statement", $2, $3, $5, $6, $7);}
| SELECT opt_distinct select_expr_di_list opt_where opt_orderby
{$$ = createNode("select_statement", $2, $3, $4, $5, NULL);}
;
```

```
delete_statement:
DELETE FROM table opt_where
{$$ = createNode("delete_statement", $3, $4, NULL, NULL, NULL);}
;

opt_where: /* empty */
{$$ = createNode("opt_where", NULL, NULL, NULL, NULL, NULL);}
| WHERE search_condition
{$$ = createNode("opt_where", $2, NULL, NULL, NULL, NULL);}
;


opt_orderby: /* empty */
{$$ = createNode("opt_orderby", NULL, NULL, NULL, NULL, NULL);}
| ORDER BY column_ref
{$$ = createNode("opt_orderby", $3, NULL, NULL, NULL, NULL);}
;

select_expr_di_list: select_expr_list
{$$ = createNode("select_expr_di_list", $1, NULL, NULL, NULL, NULL);}
| TIMES
{$$ = createNode("select_expr_di_list", NULL, NULL, NULL, NULL, NULL);}
;

select_expr_list: column_ref
{$$ = createNode("select_expr_list", $1, NULL, NULL, NULL, NULL);}
| INTNUM
{$$ = createNode("select_expr_list", NULL, NULL, NULL, NULL, NULL);}
| select_expr_list COMMA column_ref
{$$ = createNode("select_expr_list", $1, $3, NULL, NULL, NULL);}
;

search_condition: boolean_term
{$$ = createNode("search_condition", $1, NULL, NULL, NULL, NULL);}
| boolean_term OR search_condition
{$$ = createNode("search_condition", $1, $3, NULL, NULL, NULL);}
;

boolean_term: boolean_factor
{$$ = createNode("boolean_term", $1, NULL, NULL, NULL, NULL);}
| boolean_factor AND boolean_term
{$$ = createNode("boolean_term", $1, $3, NULL, NULL, NULL);}
;

boolean_factor: boolean_primary
{$$ = createNode("boolean_factor", $1, NULL, NULL, NULL, NULL);}
| NOT boolean_primary
{$$ = createNode("boolean_factor", $2, NULL, NULL, NULL, NULL);}
;
```

```
boolean_primary: comparison_predicate
{$$ = createNode("boolean_primary", $1, NULL, NULL, NULL, NULL);}
| LPAREN search_condition RPAREN
{$$ = createNode("boolean_primary", $2, NULL, NULL, NULL, NULL);}
;
comparison_predicate:
expr GREATER expr
{$$ = createNode("comparison_predicate", $1, $3, NULL, NULL, NULL);}
| expr LESSER expr
{$$ = createNode("comparison_predicate", $1, $3, NULL, NULL, NULL);}
|   expr EQUALS expr
{$$ = createNode("comparison_predicate", $1, $3, NULL, NULL, NULL);}
;
expr:
term {$$ = createNode("expr", $1, NULL, NULL, NULL, NULL);}
| expr PLUS expr {$$ = createNode("expr", $1, $3, NULL, NULL, NULL);}
| expr MINUS expr {$$ = createNode("expr", $1, $3, NULL, NULL, NULL);}
;

term:
atom {$$ = createNode("term", $1, NULL, NULL, NULL, NULL);}
| atom TIMES term   {$$ = createNode("term", $1, $3, NULL, NULL, NULL);}
| atom DIV term   {$$ = createNode("term", $1, $3, NULL, NULL, NULL);}
;

table_references: table
{$$ = createNode("table_references", $1, NULL, NULL, NULL, NULL);}
| table_references COMMA table
{$$ = createNode("table_references", $1, $3, NULL, NULL, NULL);}
;

values_or_query_spec:
VALUES LPAREN insert_atom_commalist RPAREN
{$$ = createNode("values_or_query_spec", $3, NULL, NULL, NULL, NULL);}
| select_statement
{$$ = createNode("values_or_query_spec", $1, NULL, NULL, NULL, NULL);}
;

insert_atom_commalist:
insert_atom
{$$ = createNode("insert_atom_commalist", $1, NULL, NULL, NULL, NULL);}
| insert_atom_commalist COMMA insert_atom
{$$ = createNode("insert_atom_commalist", $1, $3, NULL, NULL, NULL);}
;

insert_atom:
atom {$$ = createNode("insert_atom", $1, NULL, NULL, NULL, NULL);}
| NULLX {$$ = createNode("insert_atom", NULL, NULL, NULL, NULL, NULL);}
```

```
;

base_table_element_commalist:
base_table_element_commalist COMMA base_table_element
{$$ = createNode("base_table_element_commalist", $1, $3, NULL, NULL, NULL);}
| base_table_element
{$$ = createNode("base_table_element_commalist", $1, NULL, NULL, NULL, NULL);}
;

opt_column_commalist:
/* empty */
{$$ = createNode("opt_column_commalist", NULL, NULL, NULL, NULL, NULL);}
| LPAREN column_commalist RPAREN
{$$ = createNode("opt_column_commalist", $2, NULL, NULL, NULL, NULL);}
;


column_commalist:
column_ref
{$$ = createNode("column_commalist", $1, NULL, NULL, NULL, NULL);}
| column_commalist COMMA column_ref
{$$ = createNode("column_commalist", $1, $3, NULL, NULL, NULL);}
;

opt_distinct:
/* empty */    {$$ = createNode("opt_distinct", NULL, NULL, NULL, NULL, NULL);}
| DISTINCT    {$$ = createNode("opt_distinct", NULL, NULL, NULL, NULL, NULL);}
;

base_table_element:
column_ref INTEGER
{$$ = createNode("base_table_element", $1, NULL, NULL, NULL, NULL);}
| column_ref STR20
{$$ = createNode("base_table_element", $1, NULL, NULL, NULL, NULL);}
;

atom:
STRING   {$$ = createNode("atom", NULL, NULL, NULL, NULL, NULL);}
| INTNUM {$$ = createNode("atom", NULL, NULL, NULL, NULL, NULL);}
| column_ref {$$ = createNode("atom", $1, NULL, NULL, NULL, NULL);}
| LPAREN expr RPAREN {$$ = createNode("atom", $2, NULL, NULL, NULL, NULL);}
;

column_ref:
NAME opt_column_ref  {$$ = createNode("column_ref", $2, NULL, NULL, NULL, NULL);}
;

opt_column_ref: /* empty*/
{$$ = createNode("opt_column_ref", NULL, NULL, NULL, NULL, NULL);}
```

```
| PERIOD NAME
{$$ = createNode("opt_column_ref", NULL, NULL, NULL, NULL, NULL);}
;

table:
NAME  {$$ = createNode("table", NULL, NULL, NULL, NULL, NULL);}
;

%%

int main()
{
e = NULL;

        yyparse();

        return 0;
}

int yyerror (const char* errmsg) {
printf("Error: %s at line %d, expected next token: %d\n",errmsg,yylineno,yychar);
return 0;
}
```

1.4 The lexer

Here's our simple (and a bit dirty) Flex lexer:

Lexer.l

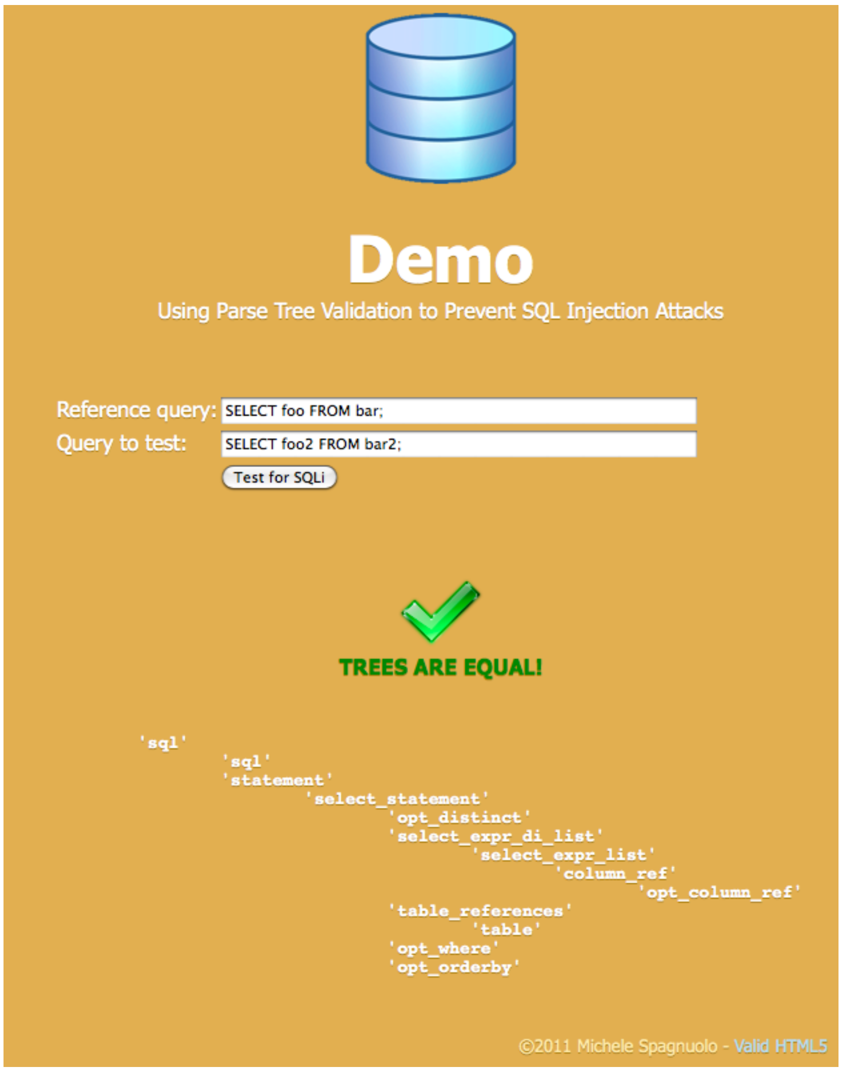
```
%option noyywrap yylineno case-insensitive
%{
#include "TypeParser.h"
#include "Parser.h"
#include <stdio.h>
#include <string.h>

#define returntoken(tok) {return (tok);}

extern int yyerror(const char* errmsg);

%}

%%

"AND" returntoken(AND);
"BY" returntoken(BY);
"CREATE" returntoken(CREATE);
"DELETE" returntoken(DELETE);
```

```
"DISTINCT" returntoken(DISTINCT);
"DROP" returntoken(DROP);
"FROM" returntoken(FROM);
"INSERT" returntoken(INSERT);
"INT" returntoken(INTEGER);
"INTO" returntoken(INTO);
"NOT" returntoken(NOT);
"NULL" returntoken(NULLX);
"ORDER" returntoken(ORDER);
"OR" returntoken(OR);
"STR20" returntoken(STR20);
"SELECT" returntoken(SELECT);
"TABLE" returntoken(TABLE);
"VALUES" returntoken(VALUES);
"WHERE" returntoken(WHERE);

"-" returntoken(MINUS);
"+" returntoken(PLUS);
"*" returntoken(TIMES);
"/" returntoken(DIV);
"." returntoken(PERIOD);
";" returntoken(SEMICOLON);
"=" returntoken(EQUALS);
"<" returntoken(LESSER);
">" returntoken(GREATER);
"(" returntoken(LPAREN);
")" returntoken(RPAREN);
"," returntoken(COMMA);

[A-Za-z][A-Za-z0-9_]* {yylval.strval = strdup(yytext); returntoken(NAME);}

[0-9]+ |
[0-9]+"."[0-9]* |
"."[0-9]* {yylval.intval = atoi(yytext); returntoken(INTNUM);}

"'"[^'\n]*"'" {
int c = input();

unput(c); /* just peeking */
if(c != '\'') {
yylval.strval = strdup(yytext);
returntoken(STRING);
} else
yymore();
}

"'"[^'\n]*$ { yyerror("Unterminated string"); }

"\r\n" { yylineno++; }
```

```
\n { yylineno++; }

[ \t\f\v]+ { /* Ignore whitespace. */ }
[ \t\r]+ { /* Ignore whitespace. */ }

"--".*$ { }

%%
```

1.5 PHP frontend

Here is a screenshot with two queries with the same parse tree:



Demo

Using Parse Tree Validation to Prevent SQL Injection Attacks

Reference query: SELECT foo FROM bar;

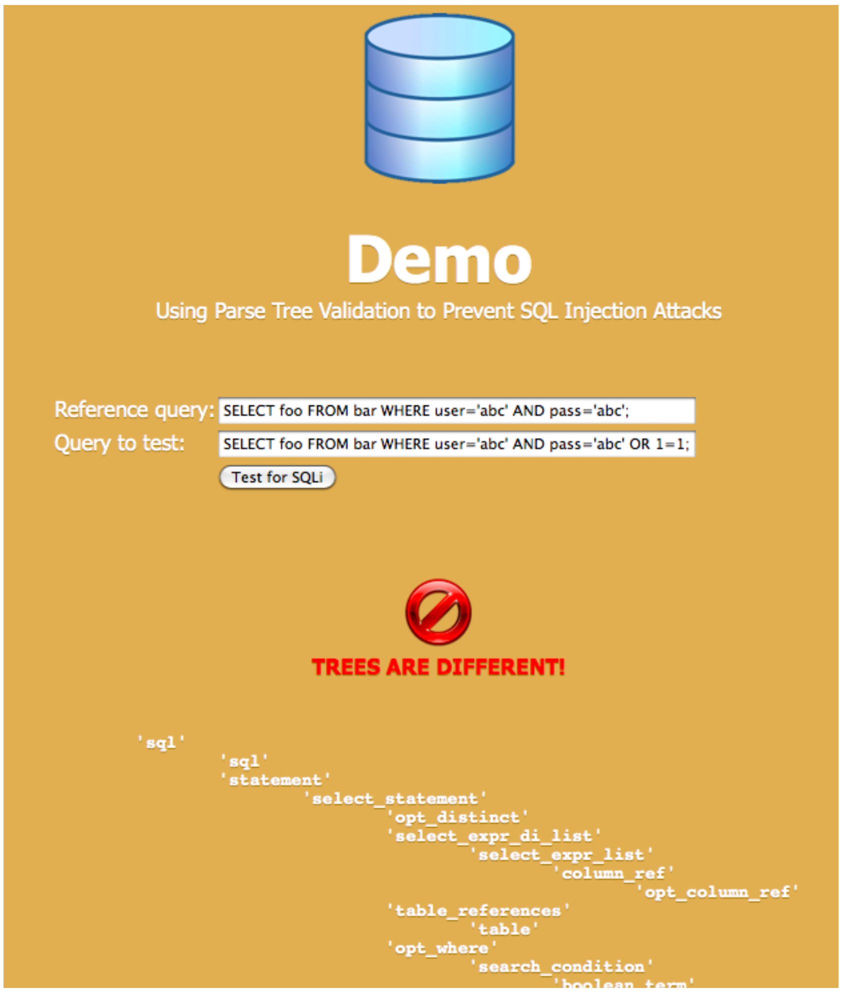
Query to test: SELECT foo2 FROM bar2;

TREES ARE EQUAL!

```
'sql'
    'sql'
    'statement'
        'select_statement'
            'opt_distinct'
            'select_expr_di_list'
                'select_expr_list'
                    'column_ref'
                        'opt_column_ref'
            'table_references'
                'table'
            'opt_where'
            'opt_orderby'
```

©2011 Michele Spagnuolo - Valid HTML5

Here is a screenshot with two queries with different parse trees (we simulate a classic SQL injection on a login form):



Demo

Using Parse Tree Validation to Prevent SQL Injection Attacks

Reference query: SELECT foo FROM bar WHERE user='abc' AND pass='abc';

Query to test: SELECT foo FROM bar WHERE user='abc' AND pass='abc' OR 1=1;

TREES ARE DIFFERENT!

```
'sql'
        'sql'
        'statement'
                'select_statement'
                        'opt_distinct'
                        'select_expr_di_list'
                                'select_expr_list'
                                        'column_ref'
                                                'opt_column_ref'
                        'table_references'
                                'table'
                        'opt_where'
                                'search_condition'
                                        'boolean_term'
```