The Essence of Command Injection Attacks in Web Applications

Zhendong Su
University of California, Davis
su@cs.ucdavis.edu

Gary Wassermann
University of California, Davis
wassermg@cs.ucdavis.edu

Abstract

Web applications typically interact with a back-end database to retrieve persistent data and then present the data to the user as dynamically generated output, such as HTML web pages. However, this interaction is commonly done through a low-level API by dynamically constructing query strings within a general-purpose programming language, such as Java. This low-level interaction is ad hoc because it does not take into account the structure of the output language. Accordingly, user inputs are treated as isolated lexical entities which, if not properly sanitized, can cause the web application to generate unintended output. This is called a *command injection attack*, which poses a serious threat to web application security. This paper presents the first formal definition of command injection attacks in the context of web applications, and gives a sound and complete algorithm for preventing them based on context-free grammars and compiler parsing techniques. Our key observation is that, for an attack to succeed, the input that gets propagated into the database query or the output document must change the intended syntactic structure of the query or document. Our definition and algorithm are general and apply to many forms of command injection attacks. We validate our approach with SQLCHECK, an implementation for the setting of SQL command injection attacks. We evaluated SQLCHECK on real-world web applications with systematically compiled real-world attack data as input. SQLCHECK produced no false positives or false negatives, incurred low runtime overhead, and applied straightforwardly to web applications written in different languages.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Reliability, Validation; D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Syntax; F.4.2 [*Mathematical Logic and Formal Languages*]: Grammars and Other Rewriting Systems—Parsing, Grammar Types

General Terms Algorithms, Experimentation, Languages, Reliability, Security, Verification

Keywords command injection attacks, web applications, grammars, parsing, runtime verification

1. Introduction

Web applications are designed to present to any user with a web browser a system-independent interface to some dynamically generated content. They are ubiquitous. For example, when a user logs on to his bank account through a web browser, he is using a web database application. These applications normally interact with databases to access persistent data. This interaction is commonly done within a general-purpose programming language, such as Java, through an application programming interface (API), such as JDBC. A typical system architecture for applications is shown in Figure 1. It is normally a three-tiered architecture, consisting of a web-browser, an application server, and a back-end database server. Within the underlying general-purpose language, such an application constructs database queries, often dynamically, and 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; it is not intended to be simply an interface for arbitrary interaction with the database.

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 *command injection attacks*.

We use one common kind of such attacks to illustrate the problem, namely the *SQL command injection attacks* (SQLCIA). An SQLCIA injection attack occurs when a malicious user, through specifically crafted input, causes a web application to generate and send a query that functions differently than the programmer intended. For example, if a database contains user names and passwords, the application may contain code such as the following:

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

This code 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 "`badguy`" into the name field and "`'OR' a'='a`" into the password field, the query string becomes:

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

whose condition always evaluates to true, and the user will bypass the authentication logic.

Command injection vulnerabilities continue to be discovered on large, real-world web applications [37], and the effects can be severe. A recent news article [23] told about a major university whose student-application login page had a vulnerability much like the example shown above. Using appropriate input, an attacker could retrieve personal information about any of the hundreds of thou-

POPL '06 January 11–13, 2006, Charleston, South Carolina, USA.

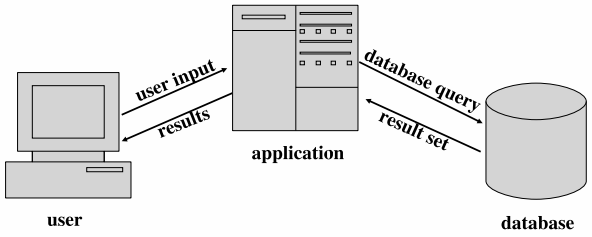


Figure 1. A typical system architecture for web applications.

sands of that school's applicants. The university had to notify every applicant whose records were in the database about the possibility that the applicant was now the victim of identity theft. This consequence was both an expense and a blow to public relations for the university.

The problem goes beyond simply failing to check input that is incorporated into a query. Even web applications that perform some checks on every input may be vulnerable. For example, if the application forbids the use of the single-quote in input (which may prevent legitimate inputs such as "`O'Brian`"), 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 number of approaches to dealing with the SQLCIA problem have been proposed, but to the best of our knowledge, no formal definition for SQLCIAs has been given. Consequently, the effectiveness of these approaches can only be evaluated based on examples, empirical results, and informal arguments. This paper fills that gap by formally defining SQLCIAs and presenting a sound and complete algorithm to detect SQLCIAs using this definition combined with parsing techniques [1].

This paper makes the following contributions:

- A formal definition of a web application, and in that context the first formal definition of an SQLCIA.
- An algorithm for preventing SQLCIAs, along with proofs of its soundness and completeness. Both the definition and the algorithm apply directly to other settings that generate interpreted output (see Section 4).
- An implementation, SQLCHECK, which is generated from lexer- and parser-generator input files. Thus SQLCHECK can be modified for different dialects of SQL or different choices of security policy (see Section 3.1) with minimal effort.
- An empirical evaluation of SQLCHECK on real-world web applications written in PHP and JSP. Web applications of different languages were used to evaluate SQLCHECK's applicability across different languages. In our evaluation, we used lists of real-world attack- and legitimate-data provided by an independant research group [13], in addition to our own test data. These lists were systematically compiled and generated from sources such as CERT/CC advisories. SQLCHECK produced no false positives or false negatives. It checked in roughly 3ms per query, and thus incurred low runtime overhead.

The rest of this paper is organized as follows: Section 2 gives an overview of our approach and Section 3 formalizes it with definitions, algorithms, and correctness proofs. Section 4 discusses other settings to which our approach applies. Sections 5 and 6 present our implementation and an evaluation of the implementation, respectively. Section 7 discusses related work, and finally, Section 8 concludes.

```
<%!
  // database connection info
  String dbDriver = "com.mysql.jdbc.Driver";
  String strConn = "jdbc:mysql://"
                       + "sport4sale.com/sport";
  String dbUser = "manager";
  String dbPassword = "athltpass";

  // generate query to send
  String sanitizedName =
    replace(request.getParameter("name"),"'","''");
  String sanitizedCardType =
    replace(request.getParameter("cardtype"),
            "'","''");
  String query = "SELECT cardnum FROM accounts"
     + " WHERE uname='" + sanitizedName + "'"
     + " AND cardtype=" + sanitizedCardType + ";";

  try {
    // connect to database and send query
    java.sql.DriverManager.registerDriver(
        (java.sql.Driver)
        (Class.forName(dbDriver).newInstance()));
    javaq.sql.Connection conn =
        java.sql.DriverManager.getConnecion(
                  strConn, dbUser, dbPassword);
    java.sql.Statement stmt =
        conn.createStatement();
    java.sql.ResultSet rs =
        stmt.executeQuery(query);

    // generate html output
    out.println("<html><body><table>");
    while(rs.next()) {
      out.println("<tr> <td>");
      out.println(rs.getString(1));
      out.println("</td> </tr>");
    }
    if (rs != null) {
      rs.close();
    }
    out.println("</table> </body> </html>");
  } catch (Exception e)
  { out.println(e.toString()); }
%>
```

Figure 2. A JSP page for retrieving credit card numbers.

2. Overview of Approach

Web applications have injection vulnerabilities because they do not constrain syntactically the inputs they use to construct structured output. Consider, for example, the JSP page in Figure 2. The context of this page is an online store. The website allows users to store credit card information so that they can retrieve it for future purchases. This page returns a list of a user's credit card numbers of a selected credit card type (*e.g.*, Visa). In the code to construct a query, the quotes are "escaped" with the `replace` method so that any single quote characters in the input will be interpreted as literal characters and not string delimiters. This is intended to block attacks by preventing a user from ending the string and adding SQL code. However, `cardtype` is a numeric column, so if a user passes

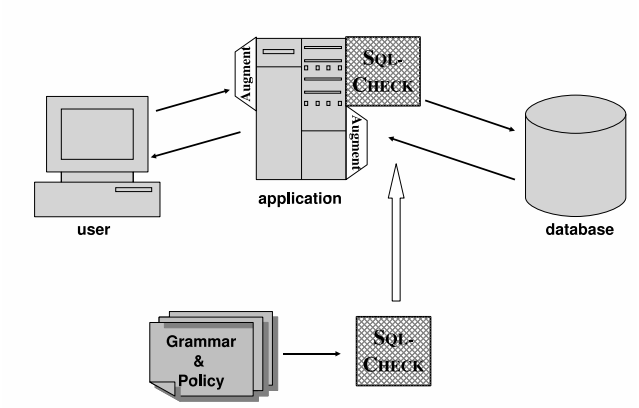


Figure 3. System architecture of SQLCHECK.

"2 OR 1=1" as the card type, all account numbers in the database will be returned and displayed.

We approach the problem by addressing its cause: we track through the program the substrings from user input and constrain those substrings syntactically. The idea is to block queries in which the input substrings change the syntactic structure of the rest of the query. Such queries are *command injection attacks* (SQLCIAs, in the context of database back-ends). We track the user's input by using meta-data, displayed as '⟪' and '⟫,' to mark the beginning and end of each input string. This meta-data follows the string through assignments, concatenations, etc., so that when a query is ready to be sent to the database, it has matching pairs of markers identifying the substrings from input. We call this annotated query an *augmented query*.

We want to forbid input substrings from modifying the syntactic structure of the rest of the query. To do this we construct an *augmented grammar* for augmented queries based on the standard grammar for SQL queries. In the augmented grammar, the only productions in which '⟪' and '⟫' occur have the following form:

$$nonterm \ ::= \ ⟪ \ symbol \ ⟫$$

where *symbol* is either a terminal or a non-terminal. For an augmented query to be in the language of this grammar, the substrings surrounded by '⟪' and '⟫' must be syntactically confined. By selecting only certain symbols to be on the *rhs* of such productions, we can specify the syntactic forms permitted for input substrings in a query.

One reason to allow input to take syntactic forms other than literals is for stored queries. Some web applications read queries or query fragments in from a file or database. For example, Bugzilla, a widely used bug tracking system, allows the conditional clauses of queries to be stored in a database for later use. In this context, a tautology is not an attack, since the conditional clause simply serves to filter out uninteresting bug reports. Persistent storage can be a medium for second order attacks [2], so input from them should be constrained, but if stored queries are forbidden, applications that use them will break. For example, in an application that allows conditional clauses to be stored along with associated labels, a malicious user may store "`val = 1; DROP TABLE users`" and associate a benign-sounding label so that an unsuspecting user will retrieve and execute it.

We use a parser generator 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.

Figure 3 shows the architecture of our runtime checking system. After SQLCHECK is built using the grammar of the output language and a policy specifying permitted syntactic forms, it resides on the web server and intercepts generated queries. Each input that is to be propagated into some query, regardless of the input's source, gets augmented with the meta-characters '⟪' and '⟫.' The application then generates augmented queries, which SQLCHECK attempts to parse. If a query parses successfully, SQLCHECK sends it sans the meta-data to the database. Otherwise, the query is blocked.

3. Formal Descriptions

This section formalizes the notion of a web application, and, in that context, formally defines an SQLCIA.

3.1 Problem Formalization

A web application has the following characteristics relevant to SQLCIAs:

- It takes input strings, which it may modify;
- It generates a string (*i.e.*, a query) by combining filtered inputs and constant strings. For example, in Figure 2, `sanitizedName` is a filtered input, and `"SELECT cardnum FROM accounts"` is a constant string for building dynamic queries;
- The query is generated without respect to the SQL grammar, even though in practice programmers write web applications with the intent that the queries be grammatical; and
- The generated query provides no information about the source of its characters/substrings.

In order to capture the above intuition, we define a *web application* as follows:

Definition 3.1 (Web Application). *We abstract a* web application $P : \langle \Sigma^*, \ldots, \Sigma^* \rangle \to \Sigma^*$ *as a mapping from user inputs (over an alphabet Σ) to query strings (over Σ). In particular, P is given by $\{\langle f_1, \ldots, f_n \rangle, \langle s_1, \ldots, s_m \rangle\}$ where*

- $f_i : \Sigma^* \to \Sigma^*$ *is an* input filter;
- $s_i : \Sigma^*$ *is a* constant string.

The argument to P is an n-tuple of input strings $\langle i_1, \ldots, i_n \rangle$*, and P returns a* query $q = q_1 + \ldots + q_\ell$ *where, for* $1 \leq j \leq \ell$*,*

$$q_j = \begin{cases} s & \text{where } s \in \{s_1, \ldots, s_m\} \\ f(i) & \text{where } f \in \{f_1, \ldots, f_n\} \ \wedge \ i \in \{i_1, \ldots, i_n\} \end{cases}$$

That is, each q_j is either a static string or a filtered input.

Definition 3.1 says nothing about control-flow paths or any other execution model, so it is not tied to any particular programming paradigm.

In order to motivate our definition of an SQLCIA, we return to the example JSP code shown in Figure 2. If the user inputs "`John`" as his user name and perhaps through a dropdown box selects credit card type "`2`" (both expected inputs), the generated query will be:

```
SELECT cardnum FROM accounts WHERE uname='John'
                                 AND cardtype=2
```

As stated in Section 2, a malicious user may replace the credit card type in the input with "`2 OR 1=1`" in order to return all stored credit card numbers:

```
SELECT cardnum FROM accounts WHERE uname='John'
                                 AND cardtype=2 OR 1=1
```

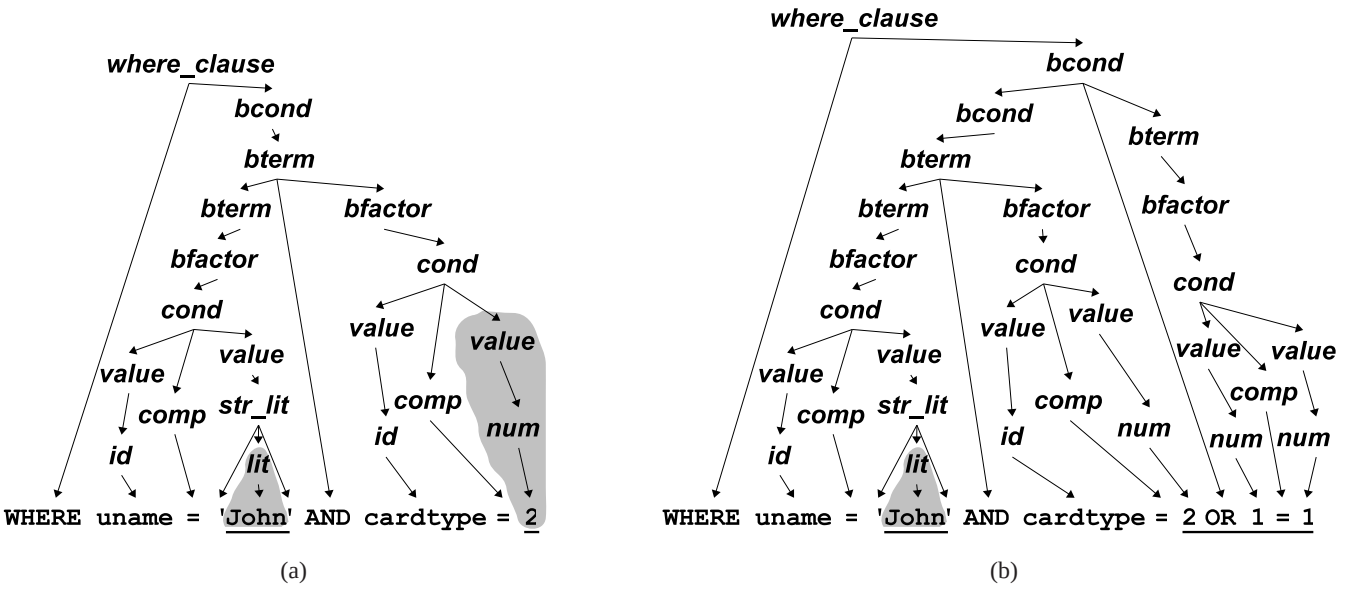


Figure 4. Parse trees for `WHERE` clauses of generated queries. Substrings from user input are underlined.

Figure 4 shows a parse tree for each query. Note that in Figure 4a, for each substring from input there exists a node in the parse tree whose descendant leaves comprise the entire input substring and no more: *lit* for the first substring and *num_lit*/*value* for the second, as shown with shading. No such parse tree node exists for the second input substring in Figure 4b. 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. We use this distinction as our definition of an SQLCIA. The definition relies on the notion of a parse tree node having an input substring as its descendants, and we formalize this notion as a *valid syntactic form*:

Definition 3.2 (Valid Syntactic Form). *Let $G = \langle V, \Sigma, S, P\rangle$ be a context-free grammar with non-terminals V, terminals Σ, a start symbol S, and productions P. Let $U \subseteq V \cup \Sigma$. Strings in the sub-language L generated by U are called* valid syntactic forms *w.r.t. U. More formally, L is given by:*

$$L = (U \cap \Sigma) \cup \bigcup_{u \in U \cap V} \mathcal{L}(\langle V, \Sigma, u, P\rangle)$$

where $\mathcal{L}(G)$ denotes the language generated by the grammar G.

Definition 3.2 allows for a modifiable security policy: The set U can be assigned such that L includes only the syntactic forms that the application programmer wants to allow the user to supply. Having a definition for valid syntactic forms, we define SQL command injection attacks as follows:

Definition 3.3 (SQL Command Injection Attack). *Given a web application P and an input vector $\langle i_1, \ldots, i_n\rangle$, the following SQL query:*

$$q = P(i_1, \ldots, i_n)$$

constructed by P is an SQL command injection attack (SQLCIA) *if the following conditions hold:*

- *The query string q has a valid parse tree T_q;*
- *There exists k such that $1 \leq k \leq n$ and $f_k(i_k)$ is a substring in q and is not a valid syntactic form in T_q.*

The first condition, that q have a valid parse tree, prevents query strings that would fail to execute from being considered attacks. The second condition includes a clause specifying that valid syntactic forms are only considered within the context of the query's parse tree. This is necessary because the same substring may have multiple syntactic forms when considered in isolation. For example, "`DROP TABLE employee`" could be viewed either as a `DROP` statement or as string literal data if not viewed in the context of a whole query.

Note that these definition do not include all forms of dangerous or unexpected behavior. Definition 3.1 provides no means of altering the behavior of the web application (*e.g.*, through a buffer overflow). Definition 3.3 assumes that the portions of the query from constant strings represent the programmer's intentions. If a programmer mistakenly includes in the web application a query to drop a needed table, that query would not be considered an SQLCIA. Additionally, Definition 3.3 constrains the web application to use input only where a valid syntactic form is permitted. By Definition 3.2, a valid syntactic form has a unique root in the query's parse tree. Consider, for example, the following query construction:

```
query = "SELECT * FROM tbl WHERE col " + input;
```

If the variable `input` had the value `"> 5"`, the query would be syntactically correct. However, if the grammar uses a rule such as "$e \rightarrow e\ op_r\ e$" for relational expressions, then the input cannot have a unique root, and this construction will only generate SQLCIAs.

However, we believe these limitations are appropriate in this setting. By projecting away the possibility of the application server getting hacked, we can focus on the essence of the SQLCIA problem. Regarding the programmer's intentions, none of the literature we have seen on this topic ever calls into question the correctness of the constant portions of queries (except Fugue [9], Gould et

```
select_stmt  ::= SELECT select_list from_clause
               | SELECT select_list from_clause where_clause
select_list  ::= id_list
               | *
id_list      ::= id
               | id , id_list
from_clause  ::= FROM tbl_list
tbl_list     ::= id_list
where_clause ::= WHERE bool_cond
bcond        ::= bcond OR bterm
               | bterm
bterm        ::= bterm AND bfactor
               | bfactor
bfactor      ::= NOT cond
               | cond
cond         ::= value comp value
value        ::= id
               | str_lit
               | num
str_lit      ::= ' lit '
comp         ::= = | < | > | <= | >= | !=
```

Figure 5. Simplified grammar for the SELECT statement.

al.'s work on static type checking of dynamically generated query strings [12], and our earlier work on static analysis for web application security [41], which consider this question to some limited degree). Additionally, programmers generally do not provide formal specifications for their code, so taking the code as the specification directs us to a solution that is fitting for the current practice. Finally, we have not encountered any examples either in the literature or in the wild of constructed queries where the input cannot possibly be a valid syntactic form.

3.2 Algorithm

Given a web application P and query string q generated by P and input $\langle i_1, \ldots, i_n \rangle$, we need an algorithm $\mathcal{A}$ to decide whether q is an SQLCIA, i.e., $\mathcal{A}(q)$ is true iff q is an SQLCIA. The algorithm $\mathcal{A}$ must check whether the substrings $f_j(i_j)$ in q are valid syntactic forms, but the web application does not automatically provide information about the source of a generated query's substrings. Since the internals of the web application are not accessible directly, we need a means of tracking the input through the web application to the constructed query. For this purpose we use *meta-characters* '$\langle\!\!|$' and '$|\!\!\rangle$,' which are not in Σ. We modify the definition of the filters such that for all filters f,

- $f : (\Sigma \cup \{\langle\!\!|, |\!\!\rangle\})^* \rightarrow (\Sigma \cup \{\langle\!\!|, |\!\!\rangle\})^*$; and
- for all strings $\sigma \in \Sigma^*$, $f(\langle\!\!|\sigma|\!\!\rangle) = \langle\!\!|f(\sigma)|\!\!\rangle$.

By *augmenting* the input to $\langle \langle\!\!|i_1|\!\!\rangle, \ldots, \langle\!\!|i_n|\!\!\rangle \rangle$, we can determine which substrings of the constructed query come from the input.

Definition 3.4 (Augmented Query). *A query q^a is an* augmented query *if it was generated from augmented input,* i.e., $q^a = P(\langle\!\!|i_1|\!\!\rangle, \ldots, \langle\!\!|i_n|\!\!\rangle)$.

We now describe an algorithm for checking whether a query is an SQLCIA. This algorithm is initialized once with the SQL grammar and a policy stating the valid syntactic forms, with which it constructs an *augmented grammar*.

Definition 3.5 (Augmented Grammar). *Given a grammar $G = \{V, \Sigma, S, P\}$ and a set $U \subseteq V \cup \Sigma$ specifying the valid syntactic forms, an* augmented grammar *G^a has the property that an augmented query $q^a = P(\langle\!\!|i_1|\!\!\rangle, \ldots, \langle\!\!|i_n|\!\!\rangle)$ is in $\mathcal{L}(G^a)$ iff:*

- *The query $q = P(i_1, \ldots, i_n)$ is in $\mathcal{L}(G)$; and*
- *For each substring s that separates a pair of matching '$\langle\!\!|$' and '$|\!\!\rangle$' in q^a, if all meta-characters are removed from s, s is a valid syntactic form in q's parse tree.*

```
select_stmt  ::= SELECT select_list from_clause
               | SELECT select_list from_clause where_clause
select_list  ::= id_list
               | *
id^a         ::= id
               | ⟨| id |⟩
id_list      ::= id^a
               | id^a , id_list
from_clause  ::= FROM tbl_list
tbl_list     ::= id_list
where_clause ::= WHERE bcond
bcond        ::= bcond OR bterm
               | bterm
bterm        ::= bterm AND bfactor
               | bfactor
bfactor      ::= NOT cond^a
               | cond^a
cond^a       ::= cond
               | ⟨| cond |⟩
cond         ::= value comp value
value        ::= id^a
               | str_lit
               | num^a
num^a        ::= num
               | ⟨| num |⟩
lit^a        ::= lit
               | ⟨| lit |⟩
str_lit      ::= ' lit^a '
comp         ::= = | < | > | <= | >= | !=
```

Figure 6. Augmented grammar for grammar shown in Figure 5. New/modified productions are shaded.

A natural way to construct an augmented grammar G^a from G and U is to create a new production rule for each $u \in U$ of the form $u^a \rightarrow \langle\!\!|u|\!\!\rangle \mid u$, and replace all other *rhs* occurrences of u with u^a. We give our construction in Algorithm 3.6.

Algorithm 3.6 (Grammar Augmentation). *Given a grammar $G = \langle V, \Sigma, S, R \rangle$ and a policy $U \subseteq V \cup \Sigma$, we define G's augmented grammar as:*

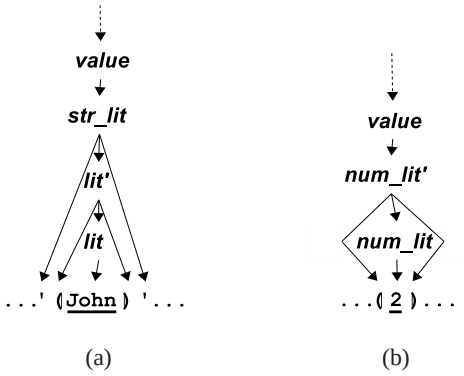
$$G^a = \langle V \cup \{v^a | v \in U\}, \Sigma \cup \{\langle\!\!|, |\!\!\rangle\}, S, R^a \rangle$$

where v^a denotes a fresh non-terminal. Given $rhs = v_1 \ldots v_n$ where $v_i \in V \cup \Sigma$, let $rhs^a = w_1 \ldots w_n$ where

$$w_i = \begin{cases} {v_i}^a & \text{if } v_i \in U \\ v_i & \text{otherwise} \end{cases}$$

R^a is given by:

$$R^a = \{v \rightarrow rhs^a \mid v \rightarrow rhs \in R\} \cup \{v^a \rightarrow v \mid v \in U\} \cup \{v^a \rightarrow \langle\!\!|v|\!\!\rangle \mid v \in U\}$$



(a) (b)

Figure 7. Parse tree fragments for an augmented query.

To demonstrate this algorithm, consider the simplified grammar for SQL's `SELECT` statement in Figure 5. This is the grammar used to generate the parse trees in Figure 4. If a security policy of $U = \{cond, id, num, lit\}$ is chosen, the result of Algorithm 3.6 is shown in Figure 6. Suppose the queries shown in Figure 4 were augmented. Using the augmented grammar, the parse tree for the first query would look the same as Figure 4a, except that the subtrees shown in Figures 7a and 7b would be substituted in for the first and second input strings, respectively. No parse tree could be constructed for the second augmented query.

A GLR parser generator [28] can be used to generate a parser for an augmented grammar G^a.

Algorithm 3.7 (SQLCIA Prevention). *Here are steps of our algorithm $\mathcal{A}$ to prevent SQLCIAs and invalid queries:*

1. *Intercept augmented query q^a;*
2. *Attempt to parse q^a using the parser generated from G^a;*
3. *If q^a fails to parse, raise an error;*
4. *Otherwise, if q^a parses, strip all occurrences of '$\langle\!|$' and '$|\!\rangle$' out of q^a to produce q and output q.*

3.3 Correctness

We now argue that the algorithms given in Section 3.2 are correct with respect to the definitions given in Section 3.1. Lemmas 3.8 and 3.9 prove the soundness and completeness respectively of Algorithm 3.6 for constructing augmented grammars. Using these lemmas, Theorem 3.10 proves the soundness and completeness of Algorithm 3.7 for preventing SQLCIAs.

Lemma 3.8 (Grammar Construction: Sound). *Let G^a be the augmented grammar constructed from grammar G and set U. For all $\langle i_1, \ldots, i_n \rangle$, if $P(i_1, \ldots, i_n) \in \mathcal{L}(G)$ and $P(i_1, \ldots, i_n)$ is not an SQLCIA, then*

$$P(\langle\!|i_1|\!\rangle, \ldots, \langle\!|i_n|\!\rangle) \in \mathcal{L}(G^a)$$

Proof. Consider an arbitrary query $q = P(i_1, \ldots, i_n)$ for some $\langle i_1, \ldots, i_n \rangle$ such that $q \in \mathcal{L}(G)$ and q is not an SQLCIA. Because $q \in \mathcal{L}(G)$, there exists a parse tree T_q for q from G's productions R. For each parse tree node v in T_q with children $v_1, \ldots, v_m$, there exists a rule $v \rightarrow v_1, \ldots, v_m \in R$. For each rule $v \rightarrow v_1, \ldots, v_m \in R$, Algorithm 3.6 specifies a rule $v \rightarrow w_1, \ldots, w_m \in R^a$ where $w_i = {v_i}^a$ if $v_i \in U$ and $w_i = v_i$ otherwise. For each $v_i \in U$, Algorithm 3.6 specifies a rule ${v_i}^a \rightarrow v_i$. Consequently, there exists a parse tree T_q^a for q from G^a's productions R^a. By assumption, q is not an SQLCIA, which by Definitions 3.2 and 3.3 means that for each $\sigma = f_j(i_j)$ used in the construction of q, there exists some parse tree node v in T_q such that $v \in U$ and v's leaf-descendants are exactly σ. Find such a parse tree node v for each σ. The construction of T_q^a from T_q creates a mapping from the parse tree nodes in T_q to the parse tree nodes in T_q^a. Using that mapping, find the corresponding node v in T_q^a. By the definition of meta-characters and of augmented queries, $P(\langle\!|i_1|\!\rangle, \ldots, \langle\!|i_n|\!\rangle)$ produces a query identical to q, except that each σ is replaced with $\langle\!|\sigma|\!\rangle$. Algorithm 3.6 specifies that there is a rule ${v_i}^a \rightarrow \langle\!|v_i|\!\rangle$ for each $v_i \in U$. These rules allow each v identified above in T_q^a to be replaced with $\langle\!|v|\!\rangle$. This results in the parse tree $T_{q^a}^a$, which proves that $q^a \in \mathcal{L}(G^a)$. □

Lemma 3.9 (Grammar Construction: Complete). *Let G^a be the augmented grammar constructed from grammar G and set U. For all $P(\langle\!|i_1|\!\rangle, \ldots, \langle\!|i_n|\!\rangle) = q^a \in \mathcal{L}(G^a)$, $P(i_1, \ldots, i_n) = q \in \mathcal{L}(G)$ and q is not an SQLCIA.*

Proof. Suppose for contradiction that there exists some $\langle i_1, \ldots, i_n \rangle$ such that $P(\langle\!|i_1|\!\rangle, \ldots, \langle\!|i_n|\!\rangle) = q^a \in \mathcal{L}(G^a)$, but $P(i_1, \ldots, i_n) = q$ is an SQLCIA. This implies that q^a has a parse tree $T_{q^a}^a$ from R^a in G^a. Because q is an SQLCIA, there exists some substring $\langle\!|\sigma|\!\rangle$ in q^a where σ is not a valid syntactic form, *i.e.*, no node v in $T_{q^a}^a$ both has σ as it descendant leaves and has $v \in U$. If $v \notin U$, then Algorithm 3.6 specifies no rule of the form $v^a \rightarrow \langle\!|v|\!\rangle \in R^a$. Consequently $T_{q^a}^a$ cannot exist, and this contradicts our initial assumption. □

Theorem 3.10 (Soundness and Completeness). *For all $\langle i_1, \ldots, i_n \rangle$, Algorithm 3.7 will permit query $q = P(i_1, \ldots, i_n)$ iff $q \in \mathcal{L}(G)$ and q is not an SQLCIA.*

Proof. Step 2 of Algorithm 3.7 attempts to parse the augmented query $q^a = P(\langle\!|i_1|\!\rangle, \ldots, \langle\!|i_n|\!\rangle)$. By Lemma 3.8, if q is an SQLCIA or if q is not a syntactically correct SQL query, q^a will fail to parse. If q^a fails to parse, step 3 will prevent q from being executed. By Lemma 3.9, if q is syntactically correct and is not an SQLCIA, q^a will parse. Step 4 causes the query to be executed. □

3.4 Complexity

Theorem 3.11 (Time Complexity). *The worst-case time bound on Algorithm 3.7 is:*

$O(\|q\|)$		*LALR*
$O(\|q\|^2)$	*if G^a is*	*not LALR but is deterministic*
$O(\|q\|^3)$		*non-deterministic*

Proof. These time bounds follow from known time-bounds for classes of grammars [1]. Achieving them for Algorithm 3.7 is contingent on the parser generator being able to handle each case without using an algorithm for a more expressive class of grammar. □

4. Applications

Although we have so far focused on examples of SQL command injections, our definition and algorithm are general and apply to other settings that generate structured, meaningful output from user-provided input. We discuss three other common forms of command injections.

4.1 Cross Site Scripting

Web sites that display input data are subject to cross site scripting (XSS) attacks. This vulnerability is perhaps more widespread than SQLCIAs because the web application need not access a back-end database. As an example of XSS, consider an auction website that allows users to put items up for bid. The site then displays a list of item numbers where each item number is a link to a URL to bid on the corresponding item. Suppose that an attacker enters as the item to add:

```
><script>document.location=
  'http://www.xss.com/cgi-bin/cookie.cgi?
  '%20+document.cookie</script
```

When a user clicks on the attacker's item number, the text in the URL will be parsed and interpreted as JavaScript. This script sends the user's cookie to `http://www.xss.com/`, the attacker's website. Note that the string provided by the attacker is not a valid syntactic form, since the first character completes a preceding tag.

4.2 XPath Injection

A web application that uses an XML document/database for its back-end storage and accesses the data through dynamically constructed XPath expressions may be vulnerable to XPath injection attacks [19]. This is closely related to the problem of SQLCIAs, but the vulnerability is more severe because:

- XPath allows one to query all items of the database, while an SQL DBMS may not provide a "table of tables," for example; and
- XPath provides no mechanism to restrict access on parts of the XML document, whereas most SQL DBMSs provide a facility to make parts of the database inaccessible.

The following piece of ASP code is vulnerable to XPath injection:

```
XPathExpression expr =
     nav.Compile("string(//user[name/text()='"
       +TextBox1.Text+"' and password/text()='"
       +TextBox2.Text+"']/account/text()");
```

Entering a tautology as in Figure 4b would allow an attacker to log in, but given knowledge of the XML document's node-set, an attacker could enter:

```
NoUser'] | P | //user[name/text()='NoUser
```

where `P` is a node-set. The surrounding predicates would always be false, so the constructed XPath expression would return the string value of the node-set `P`. Such attacks can also be prevented with our technique.

4.3 Shell Injection

Shell injection attacks occur when input is incorporated into a string to be interpreted by the shell. For example, if the string variable `filename` is insufficiently sanitized, the PHP code fragment:

```
exec("open(".$filename.")");
```

will allow an attacker to be able to execute arbitrary shell commands if `filename` is not a valid syntactic form in the shell's grammar. This vulnerability is not confined to web applications. A `setuid` program with this vulnerability allows a user with restricted privileges to execute arbitrary shell commands as `root`. Checking the string to ensure that each substring from input is a valid syntactic form would prevent these attacks.

5. Implementation

We implemented the query checking algorithm as SQLCHECK. SQLCHECK is generated using an input file to `flex` and an input file to `bison`. For meta-characters, we use two randomly generated strings of four alphabetic characters: one to represent '⟪' and the other to represent '⟫.' We made this design decision based on two considerations: (1) the meta-characters should not be removed by input filters, and (2) the probability of a user entering a meta-character should be low.

First, we selected alphabetic characters because some input filtering functions restrict or remove certain characters, but generally alphabetic characters are permitted. The common exceptions are filters for numeric fields which allow only numeric characters. In this case either the meta-characters can be added after applying an filter, or they can be stripped off leaving only numeric data which cannot change the syntactic structure of the generated query. We added them after the filter, where applicable.

Second, ignoring case, $26^4 = 456,976$ different four-character strings are possible. To avoid using meaningful words as meta-characters, we forbid meta-characters from being represented as strings that occur in the dictionary. The default dictionary for `ispell` contains 72,421 words, and if these are forbidden for use as meta-characters, 384,555 unique strings remain. It is difficult to quantify precisely the probability of a user accidentally entering a substring identical to one of the strings used for meta-characters because of several unknown factors. If we assume (1) that 90% of the words that a user enters occur in the dictionary and the remaining 10% are chosen uniformly at random from non-dictionary words, (2) that the average number of words entered on a web session is 100, and (3) that the word length is 4, the probability of a user accidentally entering a meta-character string in one web session is:

$$1 - \left(1 - \frac{2}{26^4 - 72,421}\right)^{(.1 \times 100)} \approx .000052$$

This probability can be further reduced by using longer encodings of meta-characters. We expect that the actual probability is less than what is shown above, since the numbers chosen for the calculation were intended to be conservative. Also, in settings where input is least expected to occur in a dictionary (*e.g.*, passwords), sequences of alphabetic characters are often broken up by numeric and "special" characters, and the same non-dictionary words are repeatedly entered. Section 6.3 addresses the possibility of a user guessing the meta-character encodings.

The input to `flex` requires roughly 70 lines of manually written C code to distinguish meta-characters from string literals, column/table names, and numeric literals when they are not separated by the usual token delimiters.

The algorithm allows for a policy to be defined in terms of which non-terminals in the SQL grammar are permitted to be at the root of a valid syntactic form. For the evaluation we selected literals, names, and arithmetic expressions to be valid syntactic forms. Additional forms can be added to the policy at the cost of one line in the `bison` input file per form, a find-and-replace on the added symbol, and a token declaration. Additionally, if the DBMS allows SQL constructs not recognized by SQLCHECK, they can be added straightforwardly by updating the `bison` input file. The `bison` utility includes a `glr` mode, which can be used if the augmented grammar is not LALR. For the policy choice used here, the augmented grammar is LALR.

6. Evaluation

This section presents our evaluation of SQLCHECK.

6.1 Evaluation Setup

To evaluate our implementation, we selected five real-world web applications that have been used for previous evaluations in the literature [13]. Each of these web applications is provided in mul-

Subject	Description	LOC		Query Checks Added	Query Sites	Metachar Pairs Added	External Query Data
		PHP	JSP				
Employee Directory	Online employee directory	2,801	3,114	5	16	4	39
Events	Event tracking system	2,819	3,894	7	20	4	47
Classifieds	Online management system for classifieds	5,540	5,819	10	41	4	67
Portal	Portal for a club	8,745	8,870	13	42	7	149
Bookstore	Online bookstore	9,224	9,649	18	56	9	121

Table 1. Subject programs used in our empirical evaluation.

Language	Subject	Queries		Timing (ms)	
		Legitimate (Attempted/allowed)	Attacks (Attempted/prevented)	Mean	Std Dev
PHP	Employee Directory	660 / 660	3937 / 3937	3.230	2.080
	Events	900 / 900	3605 / 3605	2.613	0.961
	Classifieds	576 / 576	3724 / 3724	2.478	1.049
	Portal	1080 / 1080	3685 / 3685	3.788	3.233
	Bookstore	608 / 608	3473 / 3473	2.806	1.625
JSP	Employee Directory	660 / 660	3937 / 3937	3.186	0.652
	Events	900 / 900	3605 / 3605	3.368	0.710
	Classifieds	576 / 576	3724 / 3724	3.134	0.548
	Portal	1080 / 1080	3685 / 3685	3.063	0.441
	Bookstore	608 / 608	3473 / 3473	2.897	0.257

Table 2. Precision and timing results for SQLCHECK.

tiple web-programming languages, so we used the PHP and JSP version of each to evaluate the applicability of our implementation across different languages. Although the notion of "applicability across languages" is somewhat qualitative, it is significant: the more language-specific an approach is, the less it is able to address the broad problem of SQLCIAs (and command injections in general). For example, an approach that involves using a modified interpreter [32, 31] is not easily applicable to a language like Java (*i.e.*, JSP and servlets) because Sun is unlikely to modify its Java interpreter for the sake of web applications. To the best of our knowledge, this is the first evaluation in the literature run on web applications written in different languages.

Table 1 lists the subjects, giving for each subject its name, a brief description of its function, the number of lines of code in the PHP and JSP versions, the number of pairs of meta-characters added, the number of input sites, the number of calls to SQLCHECK added, and the number of points at which complete queries are generated. The number of pairs of meta-characters added was less than the number of input sites because in these applications, most input parameters were passed through a particular function, and by adding a single pair of meta-characters in this function, many inputs did not need to be instrumented individually. For a similar reason, the number of added calls to SQLCHECK is less than the number of points at which completed queries are generated: In order to make switching DBMSs easy, a wrapper function was added around the database's `SELECT` query function. Adding a call to SQLCHECK within that wrapper ensures that all `SELECT` queries will be checked. Calling SQLCHECK from the JSP versions requires a Java Native Interface (JNI) wrapper. We report both figures to indicate approximately the numbers of checks that need to be added for web applications of this size that are less cleanly designed. For this evaluation, we added the meta-characters and the calls to SQLCHECK manually; in the future, we plan to automate this task using a static flow analysis.

In addition to real-world web applications, the evaluation needed real-world inputs. To this end we used a set of URLs provided by Halfond and Orso. These URLs were generated by first compiling one list of attack inputs, which were gleaned from CERT/CC advisories and other sources that list vulnerabilities and exploits, and one list of legitimate inputs. The data type of each input was also recorded. Then each parameter in each URL was annotated with its type. Two lists of URLs were then generated, one ATTACK list and one LEGIT list, by substituting inputs from the respective lists into the URLs in a type consistent way. Each URL in the ATTACK list had at least one parameter from the list of attack inputs, while each URL in the LEGIT list had only legitimate parameters. Finally, the URLs were tested on unprotected versions of the web applications to ensure that the ATTACK URLs did, in fact, execute attacks and the LEGIT URLs resulted in normal, expected behavior.

The machine used to perform the evaluation runs Linux kernel 2.4.27 and has a 2 GHz Pentium M processor and 1 GB of memory.

6.2 Results

Table 2 shows, for each web application, the number of attacks attempted (using URLs from the ATTACK list) and prevented, the number of legitimate uses attempted and allowed, and the mean and standard deviation of times across all runs of SQLCHECK for that application. SQLCHECK successfully prevented all attacks and allowed all legitimate uses. Theorem 3.10 predicted this, but these results provide some assurance that SQLCHECK was implemented without significant oversight. Additionally, the timing results show that SQLCHECK is quite efficient. Round trip time over the Internet varies widely, but 80–100ms is typical. Consequently, SQLCHECK's overhead is imperceptible to the user, and is also reasonable for servers with heavier traffic.

In addition to the figures shown in Table 2, our experience using SQLCHECK provides experimental results. Even in the absence of an automated tool for inserting meta-characters and calls to SQLCHECK, this technique could be applied straightforwardly. Most existing techniques for preventing SQLCIAs either cannot make syntactic guarantees (*e.g.*, regular expression filters) or require a tool with knowledge of the source language. For example, a type-system based approach requires typing rules in some form for each

construct in the source language. As another example, a technique that generates automata for use in dynamic checking requires a string analyzer designed for the source language. Forgoing the use of the string analyzer would require an appropriate automaton for each query site to be generated manually, which most web application programmers cannot/will not do. In contrast, a programmer without a tool designed for the source language of his choice can still use SQLCHECK to prevent SQLCIAs.

6.3 Discussions

We now discuss some of our design decisions and limitations of the current implementation.

First, we used a single policy U for all test cases. In practice we expect that a simple policy will suffice for most uses. In general, a unique policy can be defined for each pair of input site (by choosing a different pair of strings to serve as delimiters) and query site (by generating an augmented grammar according to the desired policy for each pair of delimiters). However, even if U were always chosen to be $V \cup \Sigma$, SQLCHECK would restrict the user input to syntactic forms in the SQL language. In the case where user input is used in a comparison expression, the best an attacker can hope to do is to change the number of tuples returned; no statements that modify the database, execute external code, or return columns other than those in the column list will be allowed.

Second, because the check is based on parsing, it would be possible to integrate it into the DBMSs own parser. From a software engineering standpoint, this does not seem to be a good decision. Web applications are often ported to different environments and interface with different backend DBMS's, so the security guarantees could be lost without the programmer realizing it.

Finally, the test cases used for the evaluation were generated by an independant research group from real-world exploits. However, they were not written by attackers attempting to defeat the particular security mechanism we used. In its current implementation, our technique is vulnerable to an exhaustive search of the character strings used as delimiters. This vulnerability can be removed by modifying the augmenting step: in addition to adding delimiters, it must check for the presence of the delimiters within the input string. If the delimiters occur, it must "escape" them by prepending them with some designated character. SQLCHECK must also be modified so that first, its lexer will not interpret escaped delimiters as delimiters, and second, it will remove the escaping character after parsing.

7. Related Work

7.1 Input Filtering Techniques

Improper input validation accounts for most security problems in database and web applications. Many suggested techniques for input validation are signature-based, including enumerating known "bad" strings necessary for injection attacks, limiting the length of input strings, or more generally, using regular expressions for filtering. An alternative is to alter inputs, perhaps by adding slashes in front of quotes to prevent the quotes that surround literals from being closed within the input (*e.g.*, with PHP's `addslashes` function and PHP's `magic_quotes` setting, for example). Recent research efforts provide ways of systematically specifying and enforcing constraints on user inputs [5, 35, 36]. A number of commercial products, such as Sanctum's AppShield [34] and Kavado's InterDo [17], offer similar strategies. All of these techniques are an improvement over unregulated input, but they all have weaknesses. None of them can say anything about the syntactic structure of the generated queries, and all may still admit bad input; for example, regular expression filters may be under-restrictive. More significantly, escaping quotes can also be circumvented when subtle assumptions do not hold, as in the case of the second order attacks [2]. In the absence of a principled analysis to check these methods, security cannot be guaranteed.

7.2 Syntactic Structure Enforcement

Other techniques deal with input validation by enforcing that all input will take the syntactic position of literals. Bind variables and parameters in stored procedures can be used as placeholders for literals within queries, so that whatever they hold will be treated as literals and not as arbitrary code. SQLrand, a recently proposed instruction set randomization for SQL in web applications, has a similar effect [4]. It relies on a proxy to translate instructions dynamically, so SQL keywords entered as input will not reach the SQL server as keywords. The main disadvantages of such a system are its complex setup and security of the randomization key. Halfond and Orso address SQL injection attacks through first building a model of legal queries and then ensuring that generated queries conform to this model via runtime monitoring [13], following a similar approach to Wagner and Dean's work on Intrusion Detection Via Static Analysis [8]. The precision of this technique is subject to both the precision of the statically constructed model and the tokenizing technique used. Because how their model is generated, user inputs are confined to statically defined syntactic positions. These techniques for enforcing syntactic structure do not extend to applications that accept or retrieve queries or query-fragments, such as those that retrieve stored queries from persistent storage (*e.g.*, a file or a database).

7.3 Static and Runtime Checking

Many real-world web applications have vulnerabilities, even though measures such as those mentioned above are used. Vulnerabilities exist because of insufficiency of the technique, improper usage, incomplete usage, or some combination of these. Therefore, black-box testing tools have been built for web database applications. One from the research community is called WAVES (Web Application Vulnerability and Error Scanner) [14]. Several commercial products also exist, such as AppScan [33], WebInspect [38], and ScanDo [17]. While testing can be useful in practice for finding vulnerabilities, it cannot be used to make security guarantees. Thus, several techniques based on static analysis or runtime checking have been proposed, most of which are based on the notion of "taintedness," similar to Perl's "tainted mode" [40]. In particular, there are two recent techniques using static analysis to track the flow of untrusted input through a program: one based on a type system [15] (similar to CQual [10]) and one based on a points-to analysis [24] (using a precise points-to analysis for Java [43] and policies specified in PQL [22, 26]). Both systems trust user filters, so they do not provide strong security guarantee. There is also recent work on runtime taint tracking [31, 32]. Pietraszek *et al.* suggest the use of meta-data for tracking the flow of input through filters [32]. The closest work to ours is by Buehrer *et al.* [6]. They bound user input, and at the point where queries are sent, they replace input by dummy literals and compare the parse trees of the original query and the substituted query. In this case, a lexer would suffice for the check, since input substrings must be literals. We do not address the question of completeness of usage (*i.e.*, that all input and query sites in the application source code are augmented and checked, respectively). However, a web application programmer using SQLCHECK need not make the false-positive/false-negative tradeoffs that come with less rigorous approaches. Consquently, a guarantee of completeness of usage for SQLCHECK implies that SQLCIAs will not occur.

This work also relates to some recent work on security analysis for Java applications. Naumovich and Centonze propose a static analysis technique to validate role-based access control policies

in J2EE applications [30]. They use a points-to analysis to determine which object fields are accessed by which EJB methods to discover potential inconsistencies with the policy that may lead to security holes. Koved *et al.* study the complementary problem of statically determining the access rights required for a program or a component to run on a client machine [21] using a dataflow analysis [16, 18].

7.4 Meta-Programming

To be put in a broader context, our research can be viewed as an instance of providing runtime safety guarantee for meta-programming [39]. Macros are a very old and established meta-programming technique; this was perhaps the first setting where the issue of correctness of generated code arose. Powerful macro languages comprise a complete programming facility, which enable macro programmers to create complex meta-programs that control macro-expansion and generate code in the target language. Here, basic syntactic correctness, let alone semantic properties, of the generated code cannot be taken for granted, and only limited static checking of such meta-programs is available. The levels of static checking available include none, syntactic, hygienic, and type checking. The widely used `cpp` macro pre-processor allows programmers to manipulate and generate arbitrary textual strings, and it provides no checking. The programmable syntax macros of Weise & Crew [42] work at the level of correct abstract-syntax tree (AST) fragments, and guarantee that generated code is syntactically correct with respect (specifically) to the C language. Weise & Crew macros are validated via standard type checking: static type checking guarantees that AST fragments (e.g., Expressions, Statements, etc.) are used appropriately in macro meta-programs. Because macros insert program fragments into new locations, they risk "capturing" variable names unexpectedly. Preventing variable capture is called hygiene. Hygienic macro expansion algorithms, beginning with Kohlbecker *et al.* [20] provide hygiene guarantees. Recent work, such as that of Taha & Sheard [39], focuses on designing type checking of object-programs into functional meta-programming languages. There are also a number of proposals to provide type-safe APIs for dynamic SQL, including, for example Safe Query Objects [7], SQL DOM [27], and Xen [3, 29]. These proposals suggest better programming models, but require programmers to learn a new API. In contrast, our approach does not introduce a new API, and it is suited to address the problems in the enormous number of programs that use existing database APIs. There are also research efforts on type-checking polylingual systems [11, 25], but they do not deal with applications interfacing with databases such as web applications.

8. Conclusions and Future Work

In this paper, we have presented the first formal definition of command injection attacks in web applications. Based on this definition, we have developed a sound and complete runtime checking algorithm for preventing command injection attacks and produced a working implementation of the algorithm. The implementation proved effective under testing; it identified SQLCIAs precisely and incurred low runtime overhead. Our definition and algorithm are general and apply directly to other settings that produce structured, interpreted output.

Here are a few interesting directions for future work:

- First, we plan to experiment with other ways to evaluate SQLCHECK. A natural choice will be to use SQLCHECK in some online web applications to expose SQLCHECK to the real world. By logging the blocked and permitted queries, we hope to validate that it does not disrupt normal use and does not allow attacks. A more novel approach to evaluating SQLCHECK will be to generate queries with "place-holder" user inputs. Then, using a modified top-down parser, we will generate random inputs that, when put in place of the place-holder inputs, form syntactically correct queries. By feeding these randomly generated inputs to the web application, we will test SQLCHECK on randomly generated yet meaningful queries.
- Second, we plan to explore static analysis techniques to help insert meta-characters and calls to SQLCHECK automatically. The challenge will be to insert the meta-characters such that no constant strings are captured and the control-flow of the application will not be altered.
- Third, we plan to adapt our technique to other settings, for example, to prevent cross-site scripting and XPath injection attacks.

Acknowledgments

We thank William Halfond and Alex Orso for providing legitimate and attack data for use in our evaluation. We are also grateful to Earl Barr, Christian Bird, Prem Devanbu, Kyung-Goo Doh, Alex Groce, Ghassan Misherghi, Nicolas Rouquette, Bronis Supinski, Jeff Wu, and the anonymous POPL reviewers for useful feedback on drafts of this paper.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] C. Anley. Advanced SQL Injection in SQL Server Applications. An NGSSoftware Insight Security Research (NISR) publication, 2002. URL: `http://www.nextgenss.com/papers/advanced_sql_injection.pdf`.

[3] G. Bierman, E. Meijer, and W. Schulte. The essence of data access in C^{ω}. In *The 19th European Conference on Object-Oriented Programming (ECOOP)*, 2005. To appear.

[4] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL injection attacks. In *International Conference on Applied Cryptography and Network Security (ACNS), LNCS*, volume 2, 2004.

[5] C. Brabrand, A. Møller, M. Ricky, and M. I. Schwartzbach. Powerforms: Declarative client-side form field validation. *World Wide Web*, 3(4), 2000.

[6] G. T. Buehrer, B. W. Weide, and P. A. Sivilotti. Using parse tree validation to prevent SQL injection attacks. In *Proceedings of the International Workshop on Software Engineering and Middleware (SEM) at Joint FSE and ESEC*, Sept. 2005.

[7] W. R. Cook and S. Rai. Safe Query Objects: Statically Typed Objects as Remotely Executable Queries. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, 2005.

[8] D. Dean and D. Wagner. Intrusion detection via static analysis. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Oakland, CA, May 2001. IEEE Computer Society, Technical Committee on Security and Privacy, IEEE Computer Society Press.

[9] R. DeLine and M. Fähndrich. The Fugue protocol checker: Is your software baroque? Technical Report MSR-TR-2004-07, Microsoft Research, Jan. 2004. `http://research.microsoft.com/~maf/Papers/tr-2004-07.pdf`.

[10] J. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 192–203, Atlanta, Georgia, May 1–4, 1999.

[11] M. Furr and J. S. Foster. Checking type safety of foreign function calls. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 62–72, 2005.

[12] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, pages 645–654, May 2004.

[13] W. G. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *Proceedings of 20th ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2005.

[14] Y.-W. Huang, S.-K. Huang, T.-P. Lin, and C.-H. Tsai. Web application security assessment by fault injection and behavior monitoring. In *World Wide Web*, 2003.

[15] Y.-W. 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 *World Wide Web*, pages 40–52, 2004.

[16] J. B. Kam and J. D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, 1976.

[17] Kavado, Inc. InterDo Vers. 3.0, 2003.

[18] G. A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual Symposium on Principles of Programming Languages (POPL)*, pages 194–206, Oct. 1973.

[19] A. Klein. Blind XPath Injection. Whitepaper from Watchfire, 2005.

[20] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *Conference on LISP and Functional Programming*, 1986.

[21] L. Koved, M. Pistoia, and A. Kershenbaum. Access rights analysis for Java. In *Proceedings of the 17th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 359–372, Nov. 2002.

[22] M. S. Lam, J. Whaley, V. B. Livshits, M. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *Proceedings of the ACM Conference on Principles of Database Systems (PODS)*, June 2005.

[23] R. Lemos. Flawed USC admissions site allowed access to applicant data, July 2005. `http://www.securityfocus.com/news/11239`.

[24] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *Usenix Security Symposium*, Aug. 2005. To appear.

[25] K. J. L. Mark Grechanik, William R. Cook. Static checking of object-oriented polylingual systems. `http://www.cs.utexas.edu/users/wcook/Drafts/FOREL.pdf`, Mar. 2005.

[26] M. Martin, V. B. Livshits, and M. S. Lam. Finding application errors using PQL: a program query language. In *20th Annual ACM Conference on Object-Oriented Programming, Systems, Languages*, oct 2005. To appear.

[27] R. A. McClure and I. H. Krüger. SQL DOM: compile time checking of dynamic SQL statements. In *Proceedings of the 27th International Conference on Software Engineering*, pages 88–96, 2005.

[28] S. McPeak. Elsa: An Elkhound-based C++ Parser, May 2005. `http://www.cs.berkeley.edu/~smcpeak/elkhound/`.

[29] E. Meijer, W. Schulte, and G. Bierman. Unifying tables, objects and documents, 2003.

[30] G. Naumovich and P. Centonze. Static analysis of role-based access control in J2EE applications. *SIGSOFT Software Engineering Notes*, 29(5):1–10, 2004.

[31] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *Twentieth IFIP International Information Security Conference (SEC'05)*, 2005.

[32] T. Pietraszek and C. V. Berghe. Defending against Injection Attacks through Context-Sensitive String Evaluation. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Sept. 2005.

[33] Sanctum Inc. Web Application Security Testing-Appscan 3.5. URL: `http://www.sanctuminc.com`.

[34] Sanctum Inc. AppShield 4.0 Whitepaper., 2002. URL: `http://www.sanctuminc.com`.

[35] D. Scott and R. Sharp. Abstracting application-level web security. In *World Wide Web*, 2002.

[36] D. Scott and R. Sharp. Specifying and enforcing application-level web security policies. *IEEE Transactions on Knowledge and Data Engineering*, 15(4):771–783, 2003.

[37] Security Focus. `http://www.securityfocus.com`.

[38] SPI Dynamics. Web Application Security Assessment. SPI Dynamics Whitepaper, 2003.

[39] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, 1997.

[40] L. Wall, T. Christiansen, and R. L. Schwartz. *Programming Perl (3rd Edition)*. O'Reilly, 2000.

[41] G. Wassermann and Z. Su. An Analysis Framework for Security in Web Applications. In *Proceedings of the FSE Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2004)*, pages 70–78, 2004.

[42] D. Weise and R. Crew. Programmable syntax macros. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 156–165, 1993.

[43] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 131–144, June 2004.