Using Parse Tree Validation to Prevent SQL Injection Attacks

Greg Buehrer, Bruce Weide, and Paul Sivilotti
Computer Science & Engineering
The Ohio State University
paolo@cse.ohio-state.edu
http://www.cse.ohio-state.edu/~paolo

Web Application Architecture

Client
http request (simple URL)
html response

Application
http request (parameters)
greg secret
result set
SQL query

Database

SQL Injection: Anatomy of a Vulnerability

Statement s = conn.createStatement();
String q = "SELECT * FROM scenarios " +
"WHERE username='" + user +
" AND password='" + pass +
";"
ResultSet RS = s.executeQuery(q);

SELECT * FROM scenarios
WHERE username='greg'
AND password='secret';

SELECT * FROM scenarios
WHERE username='greg';--'
AND password='anything';

SELECT * FROM scenarios
WHERE username='greg';--'
AND password='anything';

SELECT * FROM scenarios
WHERE username='greg';

Many Forms of SQL Injection

- Add comment character
  - --
- Remove end of query
- Add a tautology
  - OR 1=1, OR 'a'='a', ...
- All table rows satisfy the query
- Add a statement
  - ;DROP TABLE scenarios
- Can be complicated, multi-step
  - Insert unexpected value in table
    - eg admin'-- as a username
  - Exploit this value in subsequent query

Preventing SQL Injection (1)

- "Sanitize" the input, removing all bad characters
  - Remove ' --
  - Flag occurrence of valid SQL keywords (DROP, OR, ...)
  - Problem: sometimes these characters are ok! (e.g., O'Reilly)
- Replace ' in user input with ' \\
  - SQL treats \ as a single quote literal
- SELECT * FROM scenarios WHERE name='O' 'Reilly'
- Problem 1: undermined by other ways to escape input
- Example: backslash also yields literal: \'
  - username passed in: \'; DROP TABLE users; --'
- Double quotes to get: \'; DROP TABLE users; --'
- SELECT * FROM users WHERE name='\\'; DROP TABLE users; --'
- Problem 2: other encodings (e.g., unicode)
- Problem 3: integer fields do not have quotes
  - SELECT * FROM users WHERE userid=23 OR 1=1;
Preventing SQL Injection (2)

- "Sanitize" the input, confirming it contains only good characters
- Regular expressions, static analysis
- Problems: hard to get right, a lot to ask of programmers
- SQL Rand
  - Replace SQL tokens with secret random values
  - Attacker can not write syntactically correct SQL
  - Problem: easy to leak secret (e.g., error messages)
- Prepared statements
  - Create query as a parameterized statement
  - String q = "SELECT * FROM scenarios WHERE username=? AND password=?";
  - PreparedStatement ps = conn.prepareStatement(q);
  - ps.setObject(1, user);
- The best solution, really the right way to do things
- Problems:
  - Annoying to use (syntax for replacing ?’s can be awkward)
- Prepared statement benefits:
  - Guarantee preservation of parse tree
  - Easy, natural programming style
  - Consistent with state-of-the-practice

Our Approach: Best of Both Worlds

- Benefits of prepared statements
- Guarantee preservation of parse tree
- Benefits of dynamic string concatenation
- Easy, natural programming style
- Consistent with state-of-the-practice

Parse Tree Validation

- Assume the existence of a special character, ♣
  - Never part of user input, never part of an SQL query
- Use this character to demarcate user input
  - SELECT * FROM scenarios WHERE username='greg'
  - AND password='secret';

Compare the parse trees of two strings:
1. The string with all text between pairs of ♣ replaced by a literal token (i.e., a special leaf in the parse tree)
   - SELECT * FROM scenarios WHERE username='greg'
   - AND password='secret';
2. The string with all occurrences of ♣ removed
   - SELECT * FROM scenarios WHERE username='greg'
   - AND password='secret';

Parse Tree Comparison Rule

- Accept the SQL query if
  - The structures of the two parse trees are exactly the same (node types), and
  - The values of corresponding nodes in the two parse trees are exactly the same, except for the special (green) leaf nodes.
- Otherwise, reject the query as insecure
Revisiting our Assumption

- Technique is predicated on a reliable way
to demarcate user input
- Special character (♣) that can not be part of
user input
- Otherwise, attacker could circumvent
  - input: secret OR `a`=''a''
- But user input could include anything
- Solution:
  - Generate unique, fixed-length, random
    (unpredictable), private key for each query
  - Prefix each query with its key
  - Our prototype uses 64-bit keys (8 char)

Ease of Use

```java
Connection conn = SafeDriverManager.getConnection(DB);
...
Statement s = conn.createStatement();
String q = "SELECT * FROM scenarios " +
         "WHERE username='' + SQLGuard.wrap(user) + "' AND password='' +
         SQLGuard.wrap(pass) + ";
ResultSet RS = s.execute(q);
...
RS.close();
conn.close()
```

Architecture

Multithreading

- For scalability to many users, web
  applications are highly concurrent and
  multithreaded
- SQLGuard must generate and keep track of
  multiple keys simultaneously
- Concurrent calls to SQLGuard must be
  serialized
- Calls to wrap() must be associated with
  correct (matching) call to init()
  - Thread id used to distinguish instances

Performance Considerations
Using Parse Tree Validation to Prevent SQL Injection Attacks

Greg Buehrer, Bruce Weide, and Paul Sivilotti

Computer Science & Engineering
The Ohio State University
paolo@cse.ohio-state.edu
http://www.cse.ohio-state.edu/~paolo/software

Performance Considerations

Conclusions
- All known forms of SQL injection involve modification of the intended parse tree
  - Our approach guarantees that dynamically provided parameters are leaf nodes only
  - Modification of intended parse tree is not possible
- Limitations:
  - Unguessability of demarcation key
  - Quality of error message available for client
  - SQL tutorial web application
- Implementation available at
  - http://www.cse.ohio-state.edu/~paolo/software

Discussion Points
- Tension between weak & strong typing
  - weak typing: flexibility, rapid development, prototyping
  - strong typing: correctness, confidence, security
  - trend: weaker typing
    - e.g., scripting languages, HTTP