SQL Injection – Attacks and Defenses

Rui Zhao, Zhiju Yang, Yi Qin, and Chuan Yue

Outline

• SQL Injection
  – Unchecked inputs change SQL execution logic

• Defense in practice - new applications
  – Prepared Statements
  – Stored procedures
  – User input escaping

• Three research papers – detecting vulnerabilities in legacy applications

What is SQL Injection

• A type of injection attack: SQL commands are injected into data-plane input in order to effect the execution of predefined SQL commands.

• It occurs when:
  – Data enter a program from an untrusted source
  – The data used to dynamically construct a SQL query

(https://www.owasp.org/index.php/SQL_Injection)

SQL Injection Consequence

• Allow attackers to
  – Drop data from database
  – Alter or insert data
  – Dump sensitive data for attacker to retrieve
  – Take control of the database

• No. 1 at OWASP Top 10 Vulnerabilities – 2013

A typical example of SQL Injection

• A SQL call construction
  – String query = "SELECT * FROM accounts WHERE acct= " + request.getParameter("name") + " '";

• The value of “name” could be
  – " Bob "
    • SELECT * FROM accounts WHERE acct= 'Bob'
    – " ' or '1'=1 ' "
    • SELECT * FROM accounts WHERE acct= "or '1'=1'
    – " ' or 1=1 --" -- comment the rest of the query
    • SELECT * FROM accounts WHERE acct= " or 1=1;"
SQL Injection – Illustrated

1. Application presents a form to the attacker
2. Attacker sends an attack in the form data
3. Application forwards attack to the database in a SQL query
4. Database runs query containing attack and sends encrypted results back to application
5. Application decrypts data as normal and sends results to the user

Defenses - New Applications

• Prevent user supplied input (which contains malicious SQL) from affecting the logic of the executed query
  – Prepared statements
  – Stored procedures
  – User input escaping

Defense Option 1

String custname = request.getParameter("customerName");
String query = "SELECT account_balance FROM user_data WHERE user_name = ?";
PreparedStatement pstmt = connection.prepareStatement(query);
pstmt.setString(1, custname);
ResultSet results = pstmt.executeQuery();

// look for a customerName which literally matched the entire string

Defense Option 2

String custname = request.getParameter("customerName");
CallableStatement cs = connection.prepareCall("call sp_getAccountBalance(?)");
cs.setString(1, custname);

Avoiding SQL Injection Flaws

Recommendations
• Avoid the interpreter entirely, or
• Use an interface that supports bind variables (e.g., prepared statements, or stored procedures).
  @Bind variables allow the interpreter to distinguish between code and data
• Encode all user input before passing it to the interpreter
• Always perform 'white list' input validation on all user supplied input
• Always minimize database privileges to reduce the impact of a flaw

References
• For more details, read the https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet
Defense Option 3

- Escaping All User Supplied Input (e.g., OWASP ESAPI library)
  - Cannot guarantee it will prevent all SQL Injection in all situations
  - Should only be used, with caution, to retrofit legacy code in a cost effective way

```java
Codec ORACLE_CODEC = new OracleCodec();

String query = "SELECT user_id FROM user_data WHERE user_name = " + 
  ESAPI.encoder().encodeForSQL/ORACLE_CODEC, req.getParameter("userID") + 
  " and user_password = " + 
  ESAPI.encoder().encodeForSQL/ORACLE_CODEC, req.getParameter("pwd") + "";
```

Interesting Research on SQL Injection (more on vulnerability detection)

- “AMNESIA: Analysis and Monitoring for NEutralizing SQL Injection Attacks”, ASE, 2005
  - William G. J. Halfond, Alessandro Orso

  - Michael Martin, Monica S. Lam

  - Dennis Appelt, Cu Duy Nguyen, Lionel C. Briand, Nadia Alshahwan

“AMNESIA: Analysis and Monitoring for NEutralizing SQL Injection Attacks”, ASE, 2005
William G. J. Halfond, Alessandro Orso

- Combined static & dynamic program analysis
  - Static part: automatically build a model of the legitimate queries that could be generated by the application;
  - Dynamic part: monitors the dynamically generated queries at runtime and checks them for compliance with the statically-generated model.
  - Queries that violate the model are classified as illegal, prevented from executing on the database, and reported to the application developers and administrators.

Michael Martin, Monica S. Lam

- Proposed QED, a goal-directed model-checking system
  - Automatically generates attacks exploiting taint-based vulnerabilities in large Java web applications.
  - Model checking: given a model of a system, exhaustively and automatically check whether queries meet the model specification.

Automatic Generation of XSS and SQL Injection Attacks

- SQL injection and cross-site scripting are both instances of taint vulnerabilities:
  - untrusted data from the user is tracked as it flows through the system,
  - if it flows unsafely into a security-critical operation, a vulnerability is flagged.

- We need to analyze more than just individual requests to be sure we have found all vulnerabilities in a web application.
Automatic Generation of XSS and SQL Injection Attacks

• The input application is first instrumented according to the provided PQL query which specifies the vulnerability.
• The instrumented application and a set of seed input values form a harnessed program.
• The harnessed program is then fed to the model checker, along with stub implementations of the application server’s environment to systematically explore the space of URL requests.
• The results of that model checker correspond directly to sequences of URLs that demonstrate the attack paths.

Automated Testing for SQL Injection Vulnerabilities

• Mutation Operations
  – Behavior-changing: alter logic
  – Syntax-repairing
  – Obfuscation

Mutation Operations

<table>
<thead>
<tr>
<th>Mutation Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Behavior-changing</td>
<td>Add an AND clause to the input.</td>
</tr>
<tr>
<td>Syntax-repairing</td>
<td>Add a comment command (--) or (--) to an input.</td>
</tr>
<tr>
<td>Obfuscation</td>
<td>Add a single or double quote to an input.</td>
</tr>
</tbody>
</table>

Summary

• SQL Injection
  – Unchecked inputs change SQL execution logic

• Defense in practice - new applications
  – Prepared Statements
    – Prepared Statements
  – Stored procedures
  – User input escaping

Thank you!
Q & A

Three research papers - vulnerability detection