SQL Injection – Attacks and Defenses

Rui Zhao, Zhiju Yang, and Chuan Yue

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

Outline

• SQL Injection
• Defense in practice - new applications
• 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

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"
SQL Injection – Illustrated

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

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
- Prepared Statements (with Parameterized Queries)
  - First define all the SQL code
  - Then pass in each parameter to the query later
- Allows the database to distinguish between code and data, regardless of what user input is supplied

Defense Option 2
- Stored Procedures
  - The same effect as the use of prepared statements
  - Stored procedures is that its SQL code is defined and stored in the database itself, and then called from the application

```java
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
```
```java
String custname = request.getParameter("customerName");
CallableStatement cs = connection.prepareCall("CALL sp_getAccountBalance(?)");
    cs.setString(1, custname);
```
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")]) + ";
```

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.

Automatic Generation of XSS and SQL Injection Attacks

- 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.

Michael Martin, Monica S. Lam

- Instrumentation: adding calls to the monitor that check the queries at runtime

- Analysis:
  - Query to model mapping

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

**Summary**

- SQL Injection
  - Unchecked inputs change SQL execution logic
- Defense in practice - new applications
  - Prepared Statements
  - Stored procedures
  - User input escaping

- Three research papers - vulnerability detection

*“Automated Testing for SQL Injection Vulnerabilities: An Input Mutation Approach”, ISSTA, 2014*

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

- A black-box automated testing approach
- Applies a set of **mutation operators** that are specifically designed to increase the likelihood of generating successful SQL Injection attacks
  - Some of the mutation operators aims to obfuscate the injected SQL code fragments to bypass security filters

**Automated Testing for SQL Injection Vulnerabilities**

- WSDL: Web Service Definition Language
- WAF: Web Application Firewall
- SUT: Web Service Under Test

*Summary*

- SQL Injection
  - Unchecked inputs change SQL execution logic
- Defense in practice - new applications
  - Prepared Statements
  - Stored procedures
  - User input escaping

Thank you!

Q & A