Database Vulnerabilities

Segurança em Software

Pedro Adão, Ana Matos

(and Miguel Correia)

TÉCNICO LISBOA

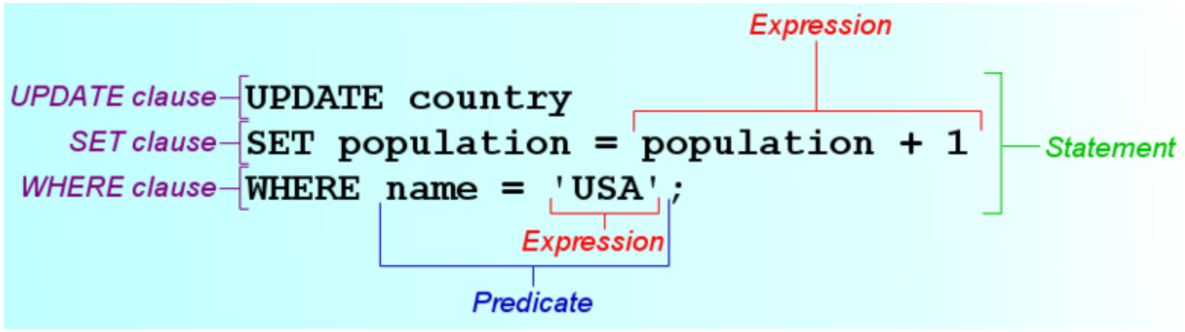
Livro: Capítulo 7 (v1) / 9 (v2)

Motivation

- According to the court document, the hackers allegedly stole more than 130 million credit and debit card numbers (...)
- Using a SQL-injection attack, the hackers allegedly broke into the 7-Eleven network in August 2007, resulting in the theft of an undetermined amount of card data. They allegedly used the same kind of attack to infiltrate Hannaford Brothers in November 2007, which resulted in 4.2 million stolen debit and credit card numbers.
  – Kim Zetter, TJX Hacker Charged With Heartland, Hannaford Breaches, Wired.com, Aug. 17, 2009.

Introduction to SQL

- Structured Query Language
  - Language for retrieval and management of data in Relational DBMSs
  - ANSI/ISO standard with proprietary extensions
- Elements of a statement:



```
UPDATE country
SET population = population + 1
WHERE name = 'USA';
```

Introduction to SQL: statements

- Data manipulation:
  - **SELECT data FROM table(s) WHERE comparison ORDER BY column(s)**
  - UPDATE table SET field1=val1 WHERE comparison
  - INSERT INTO table (field1,field2,...) VALUES (val1,val2,...)
  - DELETE FROM table WHERE comparison
- Data definition: CREATE, DROP
- Others: SELECT .... UNION SELECT ....

SQL injection

- Causes of these vulnerabilities:
  - User input pasted into SQL commands +
  - SQL uses several metacharacters / metadata
- Example (PHP/MySQL):

```
$username = $HTTP_POST_VARS[‘username’];
$password = $HTTP_POST_VARS[‘passwd’];
$query = "SELECT * FROM logintable WHERE user = ‘" .
    $username . "’ AND pass = ‘" . $password . "’ ";
$result = mysql_query($query);
if(!$result) die_bad_login();
```

SQL injection

- Causes of these vulnerabilities:
  - User input pasted into SQL commands +
  - SQL uses several metacharacters / metadata
- Example (PHP/MySQL):

```
$username = $HTTP_POST_VARS['username'];
$password = $HTTP_POST_VARS['passwd'];
$query = "SELECT * FROM logintable WHERE user = '" .
   $username . "' AND pass = '" . $password . "' ";
$result = mysql_query($query);
if(!$result) die_bad_login();
```

SQL injection

- Causes of these vulnerabilities:
  - User input pasted into SQL commands +
  - SQL uses several metacharacters / metadata
- Example (PHP/MySQL):

```
$username = $HTTP_POST_VARS['username'];
$password = $HTTP_POST_VARS['passwd'];
$query = "SELECT * FROM logintable WHERE user = '" .
   $username . "' AND pass = '" . $password . "' ";
$result = mysql_query($query);
if(!$result) die_bad_login();
```

metadata

username: root
password: root' OR pass <> 'root

Query: SELECT * FROM logintable WHERE user = 'root' AND pass = 'root' OR pass <> 'root'

Steps of a complex attack

- Identifying parameters vulnerable to injection
- Database fingerprinting
  - Discover type+version of DB using replies to queries that vary
- Discovering DB schema
  - Table names, column names, column data types
- Extracting data from the DB
- Adding/modifying data in the DB
- Denial of service

attacks, respectively, against the BD’s data confidentiality, integrity and availability

- Evading detection
  - Cleaning fingerprints

Syntax

- Notice that the specific syntax of SQL injection attacks depends on two aspects:
  - The specific DBMS used
  - The specific server-side language used

- Most of the examples that follow are Java servlets + MySQL

Example – vulnerable servlet

```
login = getParameter("login");
password = getParameter("pass");
pin = getParameter("pin");
Connection conn.createConnection("MyDataBase");
query = "SELECT accounts FROM users WHERE login=’" + login + "’ AND pass=’"
    + password + "’ AND pin=" + pin;
ResultSet result = conn.executeQuery(query);
if (result!=NULL)
    displayAccounts(result);
else
    displayAuthFailed();
```

Example – vulnerable servlet

```
login = getParameter("login");
password = getParameter("pass");
pin = getParameter("pin");
Connection conn.createConnection("MyDataBase");
query = "SELECT accounts FROM users WHERE login=’" + login + "’ AND pass=’"
    + password + "’ AND pin=" + pin;
ResultSet result = conn.executeQuery(query);
if (result!=NULL)
    displayAccounts(result);
else
    displayAuthFailed();
```

Next: types of SQL injection attacks based on this example

```
"SELECT accounts FROM users
WHERE login='" + login + "' AND
pass='" + password + "' AND
pin=" + pin;
```

1. Tautologies

- Inject code in 1 or more conditional statements so that it always evaluate to <u>true</u> (a tautology)
- Most common uses
  - Bypass authentication
  - Extract data
- Problem is with injectable field in a WHERE clause
- Example: login = <u>' or 1=1 -- </u> (with a space in the end!!!)
- Query done:

```
SELECT accounts FROM users
WHERE login='' or 1=1 -- ' AND pass='' AND pin=
"SELECT accounts FROM users
WHERE login='" + login + "' AND
pass='" + password + "' AND
pin=" + pin;
```

1. Tautologies

- Inject code in 1 or more conditional statements so that it always evaluate to true (a tautology)
- Most common uses
  - Bypass authentication
  - Extract data
- Problem is with injectable field in a WHERE clause
- Example: login = `' or 1=1 -- ` (with a space in the end!!!)
- Query done:

```
SELECT accounts FROM users
WHERE login='' or 1=1 -- ' AND pass='' AND pin=
```

\ becomes a comment

```
"SELECT accounts FROM users
WHERE login='" + login + "' AND
pass='" + password + "' AND
pin=" + pin;
```

2. Union query

- Trick the app into returning additional data by injecting UNION SELECT <rest>
  - The attacker uses <rest> to extract data from a table
  - Query returns union of data with injected query
- Example: injection in the login field

```
SELECT accounts FROM users
WHERE login=" UNION SELECT cardNo FROM CreditCards
   WHERE acctNo=10032 -- ' AND pass='' AND pin=
```

  - acctNo is the account number
  - Two SELECTs instead of one...

In relation to the previous slide, allows querying other tables

3. Piggy-backed queries

```
"SELECT accounts FROM users
WHERE login='" + login + "' AND
pass='" + password + "' AND
pin=" + pin;
```

- Does not modify a query, instead adds more queries
  - Requires DB configured to accept multiple statements in a single string (PHP doesn't let it in the common mysql_query())
- Example: injection in the pass field

> SELECT accounts FROM users WHERE login='doe' AND
> pass='<u>'; DROP table users -- </u>' AND pin=123

  - ; is the query delimiter, a metacharacter
  - Two queries are done: the first one and DROP *table users*
  - Others might be trivially added

In relation to the previous slide, allows running different statements

```
"SELECT accounts FROM users
WHERE login='" + login + "' AND
pass='" + password + "' AND
pin=" + pin;
```

4. Stored procedures

- Injection in a stored procedure
  - Vulnerability can either be SQL injection or another that depends on the language in which the procedure is written
- Example: variation of the example code above that calls the following stored procedure (that checks credentials):

```
CREATE PROCEDURE DBO.isAuthenticated
    @username varchar2, @pass varchar2, @pin int
AS
    EXEC("SELECT accounts FROM users WHERE login='" + @username + "' and
  pass='" + @password + "' and pin=" + @pin);
GO
```

- The procedure is vulnerable to the same attacks seen before
  - Piggy-backed queries, union query, etc.

5. Illegal/incorrect queries

- Find injectable parameters (DB type/version, schema) by causing errors:
  - Syntax errors – to identify injectable parameters (ex: x’)
  - Type errors – to deduce data types of certain columns or extract data (ex: xx returns error if integer field)
  - Logical errors – often reveal names of tables and columns that caused error (example next)
- Next example: Microsoft SQL Server
  - Attacker wanted to obtain the names of the user tables (u) by running:
  - SELECT * FROM sysobjects WHERE xtype = ‘u’
  - but it doesn’t work…

sysobjects is a metadata table available at MS SQL Server; in SQLite the table sqlite_master plays a similar role

Illegal/incorrect queries- example

- Example: type conversion that reveals relevant data
  - Attacker injects in the pin:

SELECT accounts FROM users WHERE login=‘’ AND pass=‘’ AND pin=convert (int,(select top 1 name from sysobjects where xtype=‘u’))

  - Extracts 1st user table (xtype=‘u’) from the metadata table (sysobjects)
  - Then, tries to convert it to integer (convert (int,…)) that is illegal so:

    Microsoft OLE DB Provider for SQL Server (0x80040E07) Error converting nvarchar value ’CreditCards’ to a column of data type int.

  - Says type of server (SQL Server) and name of 1st table (CreditCards)
  - Can be done repeatedly to find name and type of all tables

6. Inference

- Objective is the same as illegal/incorrect queries
- For DBs protected that don't return useful error data
  - e.g., because programmer knows about the previous attack
- Attack attempts to <u>infer</u> if a certain condition on the state of the DB is true or false
- Two attack techniques:
  - Blind injection
  - Timing attacks

Inference – Blind injection

Inference – Blind injection

- Information is inferred by asking true/false questions
  - if answer is true the site typically continues normally
  - if false the reply is different (although not descriptive)

Inference – Blind injection

- Information is inferred by asking true/false questions
  - if answer is true the site typically continues normally
  - if false the reply is different (although not descriptive)
- Example: If these 2 queries return different outputs, then user legalUser exists

```
SELECT accounts FROM users WHERE login='legalUser' and 1=0
  -- ' AND pass='' AND pin=0          (always false)
```

```
SELECT accounts FROM users WHERE login='legalUser' and 1=1
  -- ' AND pass='' AND pin=0          (true if legalUser exists)
```

Inference – Timing attack

Inference – Timing attack

- Information is inferred from delay in the response
- Usually with a branch that executes a WAITFOR DELAY

Inference – Timing attack

- Information is inferred from delay in the response
- Usually with a branch that executes a WAITFOR DELAY
- Example: extract the name of a table from the DB

```
SELECT accounts FROM users WHERE login='legalUser' and
    ASCII(SUBSTRING((select top 1 name from sysobjects), 1,1)) >
    X  WAITFOR DELAY 5 -- ' AND pass='' AND pin=0
```

  – If field is injectable and 1st character from the 1st table is greater (in ASCII) than *X* then delay of 5 seconds

7. Alternate encodings

- Not a full attack but a trick to evade detection
- Idea is to encode input in an unusual format
  - Hexadecimal, ASCII, Unicode
- Example

```
SELECT accounts FROM users WHERE login='legalUser';
exec(char(0x73687574646f776e)) -- AND pass='' AND pin=
```

  - char transforms an integer or hexadecimal encoding into ASCII
  - in this case: exec(shutdown)

Injection mechanisms

- GET/POST inputs (all the previous examples)
- Cookies (can be used by the server to build SQL commands)
  - Ex: SELECT …. WHERE cookie='%s' …
- Header fields (in PHP)
  - $_SERVER['HTTP_XXX']
  - substituting XXX with the name of the header field:
    - HTTP_URL,
    - HTTP_ACCEPT_LANGUAGE,
    - HTTP_HOST,
    - HTTP_USER_AGENT, …

Injection mechanisms (cont)

- Second-order injection (previous ones were 1st order injection)
  - Input is provided so that it is kept in the system and later used

- Ex: attacker registers as a new user called admin’ --
  - Site correctly escapes the input and accepts it
  - Later, when a change of password is requested (unescapes then uses):

```
UPDATE users SET password=‘newpwd’
WHERE userName= ’admin’-- ’ AND password=’oldpwd’
```

\ becomes a comment

Preventing SQL injection

Preventing SQL injection

- Parameterized commands (aka prepared statements)
  - PreparedStatement cmd = conn.**prepareStatement**(“SELECT accounts FROM users WHERE login=? AND pass=?”);
  - cmd.**setString**(1, login);
  - cmd.**setString**(2, password);

Preventing SQL injection

- Parameterized commands (aka prepared statements)
  - PreparedStatement cmd = conn.**prepareStatement**(“SELECT accounts FROM users WHERE login=? AND pass=?”);
  - cmd.**setString**(1, login);
  - cmd.**setString**(2, password);
- Input validation / whitelisting
  - Accept good input (instead of rejecting bad input)

Preventing SQL injection

- Parameterized commands (aka prepared statements)
  - PreparedStatement cmd = conn.**prepareStatement**("SELECT accounts FROM users WHERE login=? AND pass=?");
  - cmd.**setString**(1, login);
  - cmd.**setString**(2, password);
- Input validation / whitelisting
  - Accept good input (instead of rejecting bad input)
- Input sanitization
  - To allow input with characters that might be interpreted as metacharacters
  - e.g. **mysql_real_escape_string** in PHP (in a string that will be passed in a query to **mysql_query**)

More vulnerabilities in DBMSs

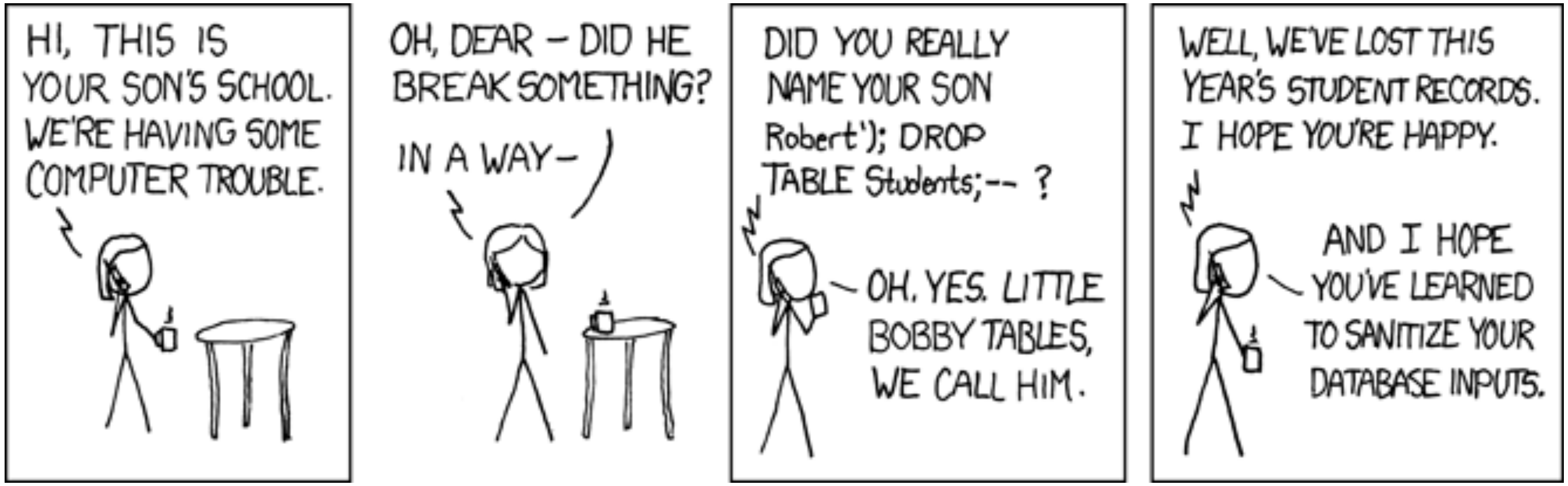
- Blank and default passwords
- Unprotected communication (client-server)
  - In virtually all DBMSs TCP is used by default
- Default (privileged) accounts
- Several open ports
- Code vulnerabilities
  - e.g. Oracle issues patches every 3 months
    - Jan., April, July, October

More vulnerabilities using DBMSs

- Encryption problems
  - Storing keys with the data
  - Fail to centralize key management (in large organizations)
  - Depending on home-brew solutions
  - Leaving backups unencrypted
  - Using out-of-date cryptographic algorithms

Summary

- Databases
  - SQL Injection introduction
  - SQL Injection – 7 attacks
  - Injection mechanisms
  - Prevention
  - Other vulnerabilities


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

'OR 1=1;--