Database Vulnerabilities

Ibéria Medeiros

Departamento de Informática

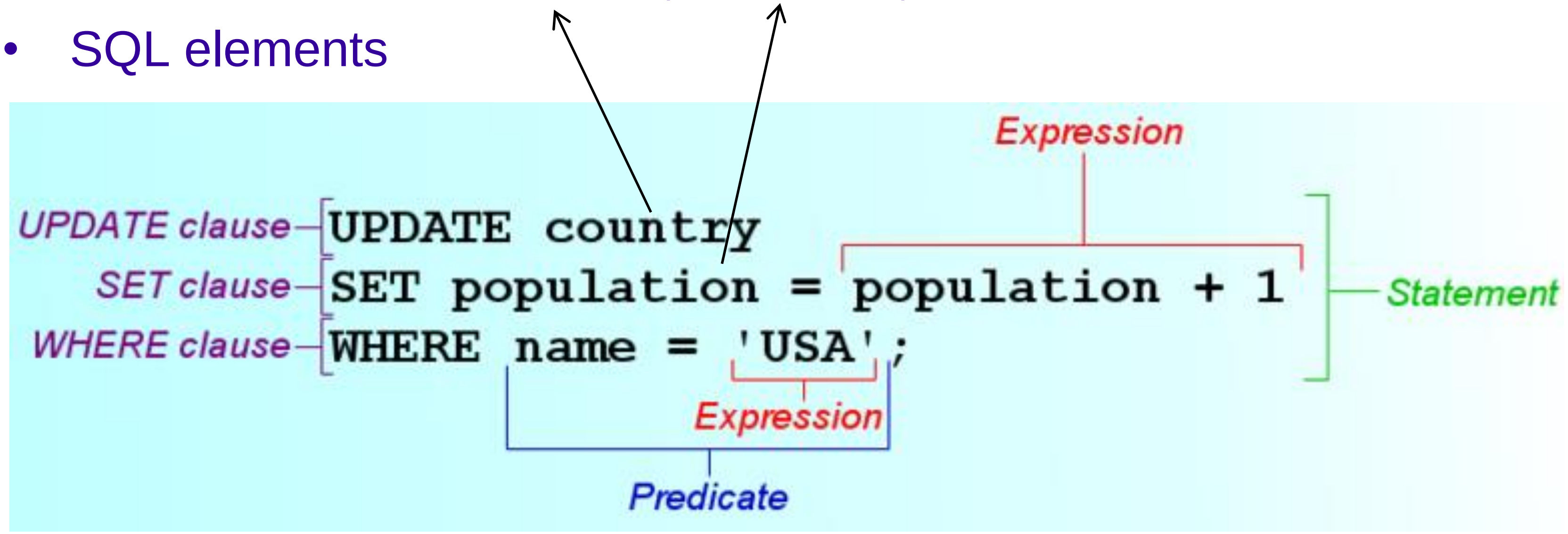
Faculdade de Ciências da Universidade de Lisboa

BPC Banking Software Has SQL Injection Vulnerability (Oct 2017)

- A vulnerability in the BPC ecommerce platform has gone unpatched since it was privately disclosed to the Swiss company BPC Banking Technologies in April 2017. The flaw could be exploited through an ***SQL injection attack*** to steal sensitive data. To exploit the flaw, an attacker would need to be authenticated to a computer that is running the vulnerable software.

Introduction to SQL

- Structured Query Language (SQL)
  - ☞ Language for retrieval and management of data in *Relational DBMS*
  - ☞ ANSI/ISO standard with proprietary extensions
- Relational model: relations, attributes, domains
- SQL elements



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

Introduction to SQL (cont)

- Data manipulation
  - ☞ SELECT column1, column2, … FROM table1, table2, … WHERE predicate ORDER BY columns2, columns1, …
  - ☞ UPDATE table SET column1=val1, column2=val2, … WHERE predicate
  - ☞ INSERT INTO table1 (column1,column2,…) VALUES (val1,val2,…)
  - ☞ DELETE FROM table1 WHERE predicate
- Data definition
  - ☞ CREATE TABLE, DROP TABLE, ALTER TABLE
- Others
  - ☞ SELECT columns FROM table1 <u>UNION</u> SELECT columns FROM table2

SQL injection

- Causes of these vulnerabilities
  - ☞ User input to the web application pasted into SQL commands ***and***
  - ☞ SQL uses several metacharacters

- 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(); else generateID();
```

unfiltered

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

metacharacter

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

SQL injection (cont)

- Avoiding returning the whole table

```
SELECT * FROM logintable WHERE user = 'root' --
  ' AND pass = ''
```

becomes comment

one space here

- Not necessarily with ', if the variable is an integer

```
$order_id = $HTTP_POST_VARS ['order_id'];
$query = "SELECT * FROM orders WHERE id="
    . $order_id;
$result = mysql_query($query);
```

☞ order_id can be set to 1 OR 1=1

Injection mechanisms

- There are different ways for the adversary to provide malicious data to the application

- User input *(the previous examples)*

- Cookies (can be used by the server to build SQL commands)
  - ☞ Ex: `SELECT …. WHERE cookie='%s' …`

- ServerVariables of ASP.NET
  - ☞ Collection that retrieves the values of environment variables and the HTTP request header information (at the server)
    `Request.ServerVariables("CONTENT_LENGTH")`

Injection mechanisms (cont)

- Second-order injection (vs the others – 1st order injection)
  - ☞ Input is provided so that it is kept in the system and later used

  - ☞ For example, attacker registers as a new user called admin' --
  - ☞ Site correctly escapes the input and accepts
  - ☞ Later on the attacker changes the passw

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

becomes a comment

Not only Web Applications!

- The problem can occur in any application that manages data through a database

```
snprintf(buf, sizeof(buf), “SELECT * FROM logintable
  WHERE user = \'%s\' AND pass = \'%s\'”, user,
  pass);
```

☞ what happens if “ `AND pass=…` ” is truncated?

Attack steps

- 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

$\Big\}$ *attacks, respectively, against the BD's data confidentiality, integrity and availability*

- Evading detection
  - ☞ cleaning fingerprints and history data in the DB
- Arbitrary command execution in the database
- Privilege escalation

Syntax

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

- Most of the following examples are in 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();
```

Next: types of SQL injection attacks based on this example

1. Tautologies

- Inject code in one or more **conditional statements** so that it always evaluate to true
- Problem: injectable fields used in a query's WHERE clause in commands like SELECT, UPDATE, DELETE
- Most common uses
  - ☞ bypass authentication
  - ☞ extract data
- Example
  - ☞ login is `' or 1=1 -- ` *(with a space in the end, makes the rest look like a comment)*
- Query done

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

2. Union query

- Idea is to trick the application into returning arbitrary data by injecting UNION SELECT <…>
  - ☞ only works with the SELECT command
  - ☞ <…> is used to extract data from some particular table
  - ☞ returns union of the intended data with injected query
  - ☞ the ***domain** and **number*** of the attributes of both SELECT needs to be equal
- Example: injection in the login field

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

  - ☞ acctNo is the account number
  - ☞ Two selects instead of one ...

3. Piggy-backed queries

- The idea is to add more (independent) queries
  - ☞ often can be injected in any field and allows the injection of any type of command
  - ☞ requires DB configured to accept multiple statements in a single string (e.g., command `mysqli()`)
- Example: injection in the pass field

```
SELECT accounts FROM users WHERE login='doe' AND
pass=''; DROP TABLE users -- ' AND pin=123
```

  - ☞ ; is the query delimiter, a metacharacter
  - ☞ two queries are executed: the normal one and *drop table users*
  - ☞ others might be trivially added

4. Stored procedures

- Stored procedures are saved and executed inside the DBMS
  - ☞ PL/SQL for Oracle; Transact-SQL, C# and VB.NET for Microsoft SQL Server; SPL for MySQL
- Usually it is assumed that stored procedures ARE the solution to avoid SQL injection attacks
- In practice, if one is not careful, since the procedural language is an extension of SQL, they might be vulnerable to the same attacks

- Exploit a vulnerability while executing a stored procedure
  - ☞ vulnerability can either be SQL injection **or** the execution of commands at the OS level, as the procedural language supports the execution of OS commands through library functions
  - ☞ depends on the language in which the procedure is written

Example in next page →

4. Stored procedures (cont)

```
$name = $_POST['userName'];
$pass = $_POST['password'];
$query = "CALL isAuthenticated('".$name."','".$pass."')";
$result = $mysqli->query($query$);
If ($result) ...

-----------------------------------------------------------

DELIMITER $$
CREATE PROCEDURE isAuthenticated
   (IN name VARCHAR(50), IN pass VARCHAR(50))
BEGIN
   SET @sql=CONCAT("SELECT name FROM users WHERE login='",
   name, "' AND passwd='", pass,"'");
   PREPARE cmd FROM @sql;
   EXECUTE cmd;
   DEALLOCATE PREPARE cmd;
END $$
DELIMITER ;
```

Password equal to:
admin\' OR passwd <> \' admin

5. Illegal/incorrect queries

- The objective is to find *injectable parameters*, *DBMS type/version*, *schema* by causing errors
  - ☞ syntax errors: allow the identification of injectable parameters
  - ☞ type errors: support the deduction of data types of certain columns or extract data
  - ☞ logical errors: often reveal names of tables and columns that caused the error
- Targets usually database configuration/management tables
- Requires: capability to do SQL injection + verbose error messages

- Next example: Microsoft SQL Server
  - ☞ Attacker would like to run something like

```
SELECT * FROM sysobjects WHERE xtype = 'u'
```

that returns the names of the tables of the user (u)

Illegal/incorrect queries- example

- Example: type conversion that reveals relevant data
  - ☞ Attacker injects in the pin and the resulting query is

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

  - ☞ Attempts to extract 1st user table name (indicated by xtype='u') from the metadata table (sysobjects)
  - ☞ Then, tries to convert it to integer (convert (int,…)), which is illegal so the following error is generated

    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 repeated to find name of other user tables

6. Inference

- Objective the same as illegal/incorrect queries
  - ☞ for DBMS protected from those attacks by not returning any result with indicative information
  - ☞ Attack attempts to infer if a certain condition on the state of the DBMS is true or false

- Two attack techniques
  - ☞ **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)
  - ☞ **Timing attacks**
    - – information is inferred from timing delays in the response
    - – usually with a branch that executes a WAITFOR DELAY

Inference - examples

- **Blind injection**
  - ☞ Goal is to find out if field login is vulnerable to SQL injection
  - ☞ If they return different output, login is vulnerable to injection

```
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   (always true)
```

- **Timing attack**
  - ☞ Objective is to 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 1st character from the 1st table is greater than X then wait

7. Alternate encodings

- Useful to complement other attacks but a good trick to evade detection mechanisms (e.g., black lists) of the web application
- The 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 in ASCII
  - ☞ in this case: exec(shutdown)

Preventing SQL injection (1)

- Parameterized SQL commands
  - ☞ separate the command from the potentially malicious data that is used at execution time
  - ☞ ensure that data is only interpreted as data (and not commands)

```
PreparedStatement cmd = conn.prepareStatement(
“SELECT accounts FROM users WHERE login=? AND
pass=?”);
cmd.setString(1, login);  // indicate the data domain
cmd.setString(2, password);
```

- Input type checking
  - ☞ remove malicious input
  - ☞ transform malicious input into something that cannot be attacked

Preventing SQL injection (2)

- Whitelisting
  - ☞ accept good input (instead of rejecting bad input)

- Encoding of input to something that can be trusted
  - ☞ 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**)

Other vulnerabilities in DBMSs

- Blank and default passwords
- Default (privileged) accounts

- Unprotected communication
  - ☞ in virtually all DBMSs by default
- Several open ports

Bibliography

- M. Correia, P. Sousa, Segurança no Software, FCA Editora, 2017 (see chapter 9)

Other references: