SQL Injection

EECS3421 - Introduction to Database Management Systems

Credit

"Foundations of Security: What Every Programmer Needs To Know" (Chapter 8)

by Neil Daswani, Christoph Kern, and Anita Kesavan

https://link.springer.com/book/10.1007%2F978-1-4302-0377-3

Agenda

- *Code injection* vulnerability - untrusted input inserted into query or command
  - Attack string alters intended semantics of command
  - Ex: ***SQL Injection***
    - unsanitized data used in query to back-end database
- SQL Injection Attack Scenarios
  - **First-order SQL Injection**
    - Type 1: compromises user data
    - Type 2: modifies critical data
  - **Second-order SQL Injection**
    - Two-phases attach (first store data, then exploit)
- SQL Injection Solutions
- Mitigating the impact of SQL Injection Attacks

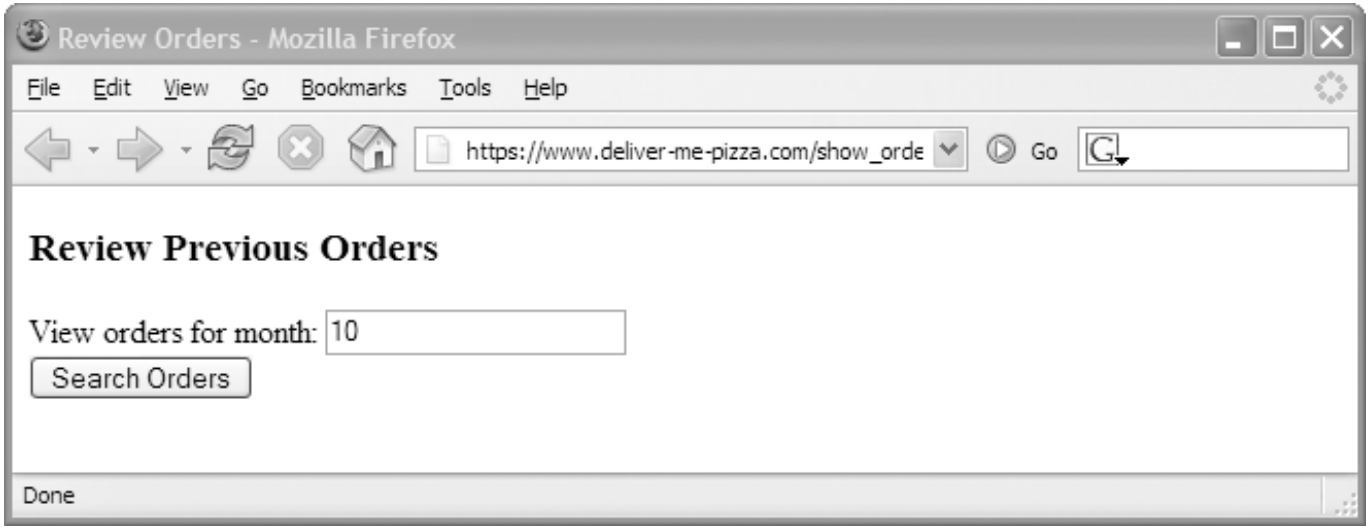
SQL Injection Impact

- CardSystems, credit card payment processing ruined by SQL Injection attack in June 2005
  - 263,000 credit card #s stolen from its DB
  - #s stored unencrypted, 40 million exposed
- Heartland Payment Systems (2005-2007)
  - 130 million cards were hacked
  - Hackers sentenced for SQL injections that cost $300 million
- Awareness Increasing
  - SQL injection vulnerabilities tripled from 2004 to 2005
  - In 2012, average web app gets: 4 attacks/per month
- More examples:
  - http://en.wikipedia.org/wiki/SQL_injection#Examples
  - https://moneywise.com/a/worst-data-breaches-of-the-century

SQL Injection
Attack Scenarios

First-order SQL Injection (1/6)

- Ex: Pizza Site Reviewing Orders
  - Form requesting month # to view orders for



  - HTTP request:

```
https://www.deliver-me-pizza.com/show_orders?month=10
```

First-order SQL Injection (2/6)

- App constructs SQL query from parameter:

```
sql_query = "SELECT pizza, toppings, quantity, order_day " +
            "FROM orders " +
            "WHERE userid=" + session.getCurrentUserId() + " " +
            "AND order_month=" + request.getParamenter("month");
```

Normal SQL Query

```
SELECT pizza, toppings, quantity, order_day
FROM orders
WHERE userid=4123
AND order_month=10
```

- Type 1 Attack: inputs `month='0 OR 1=1'` !
- Goes to encoded URL: (space -> `%20`, = -> `%3D`)

`https://www.deliver-me-pizza.com/show_orders?month=0%20OR%201%3D1`

First-order SQL Injection (3/6)

Malicious Query

```
SELECT pizza, toppings, quantity, order_day
FROM orders
WHERE userid=4123 AND order_month=0 OR 1=1
```

- WHERE condition is always true!
  - OR precedes AND
  - Type 1 Attack: Gains access to other users' private data!

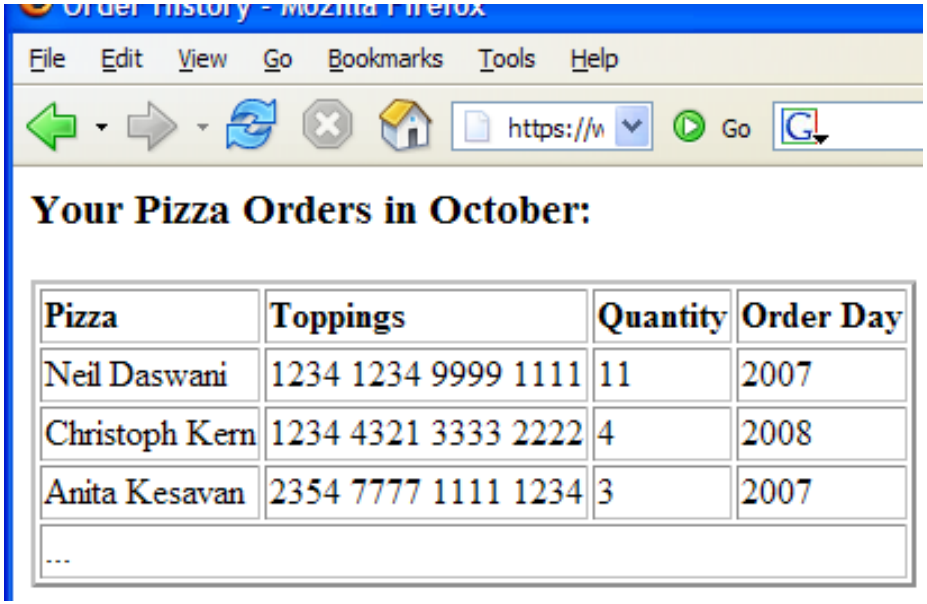
All User Data Compromised

First-order SQL Injection (4/6)

More damaging attack: attacker sets

```
month='0 AND 1=0
UNION
SELECT cardholder, number, exp_month, exp_year
FROM creditcards'
```

- Attacker is able to
  - Combine 2 queries
  - 1st query: empty table (where fails)
  - 2nd query: credit card #s of all users



First-order SQL Injection (5/6)

- Even worse, attacker sets

```
month='0;
DROP TABLE creditcards;'
```

- Then DB executes
  - Type 2 Attack: Removes `creditcards` from schema!
  - Future orders fail: DoS!

```
SELECT pizza, toppings,
quantity, order_day
FROM orders
WHERE userid=4123
AND order_month=0;
DROP TABLE creditcards;
```

- Problematic Statements:
  - Modifiers: `INSERT INTO admin_users VALUES ('hacker',...)`
  - Administrative: shut down DB, control OS…

First-order SQL Injection (6/6)

- Injecting String Parameters: Topping Search

```
sql_query =
  "SELECT pizza, toppings, quantity, order_day " +
  "FROM orders " +
  "WHERE userid=" + session.getCurrentUserId() + " " +
  "AND topping LIKE '%" + request.getParamenter("topping") + "%' ";
```

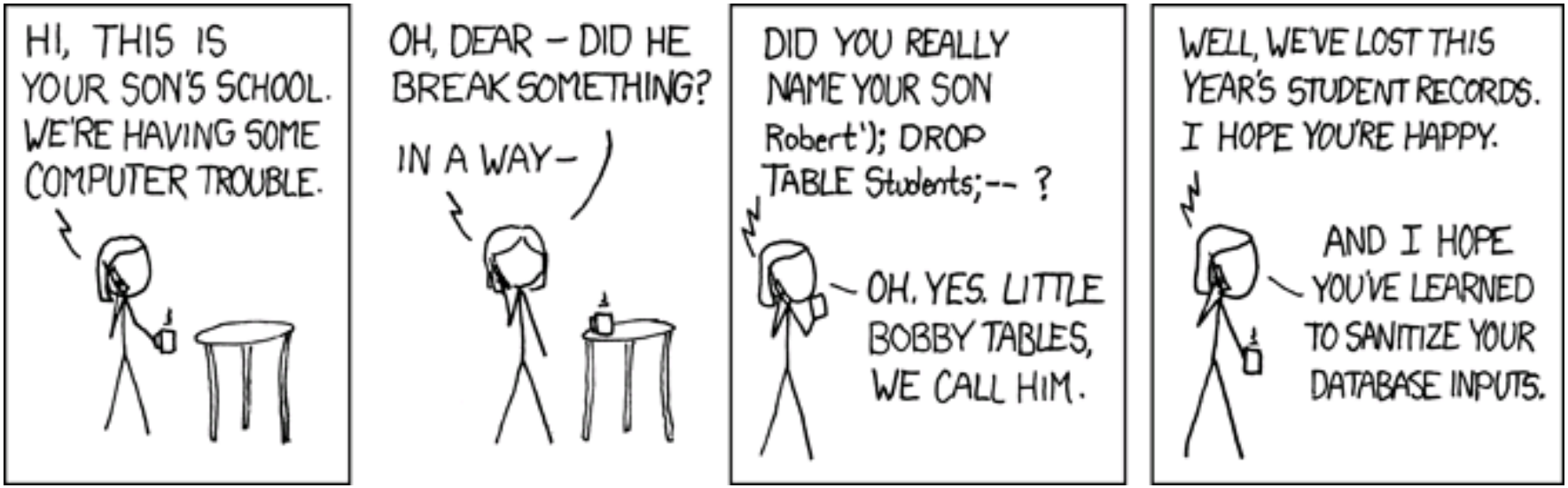
- Attack searches for:

```
brzfg%'; DROP table creditcards; --
```

- Query evaluates as:
  - SELECT: empty table
  - -- comments out end
  - Credit card info dropped

```
SELECT pizza, toppings,
quantity, order_day
FROM orders
WHERE userid=4123
AND topping LIKE '%brzfg%';
DROP table creditcards; --%'
```

Sanetize your Database Inputs



Source: http://xkcd.com/327/

Second-Order SQL Injection (1/2)

- *Second-Order SQL Injection*: data stored in database is later used to conduct SQL injection
  - Common if string escaping is applied inconsistently
  - Ex: `o'connor` updates passwd to `SkYn3t`

```
new_passwd = request.getParameter("new_passwd");
uname = session.getUsername();
sql = "UPDATE USERS SET passwd='"+ escape(new_passwd) +
      "' WHERE uname='" + uname + "'";
```

  - `uname` not escaped, b/c originally escaped before entering into the DB, now inside our trust zone:

```
UPDATE USERS SET passwd='SkYn3t' WHERE uname='o'connor'
```

  - Query fails b/c **'** after **o** ends command prematurely

Second-Order SQL Injection (2/2)

- Even Worse: What if user set `uname=`**`admin'--`** !?

```
UPDATE USERS SET passwd='cracked' WHERE uname='admin' --'
```

  - Attacker changes `admin`'s password to `cracked`
  - Has full access to `admin` account
  - Username avoids collision with real `admin`
  - `--` comments out trailing quote

- All parameters dangerous

Solutions

Solutions

A. Blacklisting
B. Whitelisting over Blacklisting
C. Input Validation & Escaping
D. Use Prepared Statements & Bind Variables

A. Blacklisting

- Eliminating quotes enough (blacklist them)?

```
sql_query =
"SELECT pizza, toppings, quantity, order_day " +
"FROM orders " +
"WHERE userid=" + session.getCurrentUserId() + " " +
"AND topping LIKE
'kill_quotes(request.getParamenter("topping")) + "%'";
```

- `kill_quotes` (Java) removes single quotes:

```
String kill_quotes(String str) {
  StringBuffer result = new StringBuffer(str.length());
  for (int i = 0; i < str.length(); i++) {
    if (str.charAt(i) != '\'')
      result.append(str.charAt(i));
  }
  return result.toString();
}
```

A. Pitfalls of Blacklisting

- Filter quotes, semicolons, whitespace, and…?
  - Could always miss a dangerous character
  - Blacklisting not comprehensive solution
  - Ex: `kill_quotes()` can't prevent attacks against numeric parameters
- May conflict with functional requirements
  - Ex: How to store O'Brien in DB if quotes blacklisted?

B. Whitelisting

- *Whitelisting* – only allow input within well-defined set of safe values
  - set implicitly defined through *regular expressions*
  - *RegExp* – pattern to match strings against

- Ex: `month` parameter: non-negative integer
  - RegExp: `^[0-9]*$` - 0 or more digits, safe subset
    - The `^`, `$` match beginning and end of string
    - `[0-9]` matches a digit,
    - `*` specifies 0 or more

C. Input Validation and Escaping

- Could escape quotes instead of blacklisting
- Ex: insert user `o'connor`, password `terminator`

```
sql = "INSERT INTO USERS(uname,passwd) " +
      "VALUES (" + escape(uname)+ "," +
      escape(password) +")";
```

  - `escape(o'connor) = o''connor`

```
INSERT INTO USERS(uname,passwd) VALUES ('o''connor','terminator');
```

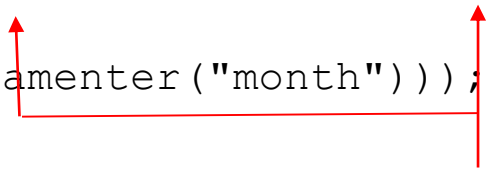
- Like `kill_quotes`, only works for string inputs
- Numeric parameters could still be vulnerable

D. Prepared Statements & Bind Variables

- Metachars (e.g. quotes) provide distinction between data & control in queries
  - most attacks: data interpreted as control
  - alters the semantics of a query
- *Bind Variables*: ? placeholders guaranteed to be data (not control)
- *Prepared Statements* allow creation of static queries with bind variables
  - Preserves the structure of intended query
  - Parameters not involved in query parsing/compiling

Java Prepared Statements

```
PreparedStatement ps =
db.prepareStatement("SELECT pizza, toppings, quantity, order_day "
                  + "FROM orders WHERE userid=? AND order_month=?");
ps.setInt(1, session.getCurrentUserId());
ps.setInt(2, Integer.parseInt(request.getParamenter("month")));
ResultSet res = ps.executeQuery();
```



Bind Variable:
Data Placeholder

- Query parsed without parameters
- Bind variables are **typed**: input must be of expected type (e.g. int, string)

PHP Prepared Statements

```
$ps = $db->prepare(
      'SELECT pizza, toppings, quantity, order_day '.
      'FROM orders WHERE userid=? AND order_month=?');
$ps->execute(array($current_user_id, $month));
```

- No explicit typing of parameters like in Java
- Apply consistently: adding `$month` parameter directly to query still creates SQL injection threat

- Have separate module for DB access
  - Do prepared statements here
  - Gateway to DB for rest of code

SQL Stored Procedures

- ***Stored procedure***: sequence of SQL statements executing on specified inputs
- Ex:

```
CREATE PROCEDURE change_password
                 @username VARCHAR(25),
                 @new_passwd VARCHAR(25) AS
UPDATE USERS SET passwd=new_passwd WHERE uname=username
```

- Vulnerable use:

```
$db->exec("change_password '"+$uname+"','"+new_passwd+"'");
```

- Instead use bind variables w/ stored procedure:

```
$ps = $db->prepare("change_password ?, ?");
$ps->execute(array($uname, $new_passwd));
```

Mitigating the Impact of SQL Injection Attacks

YORK
UNIVERSITÉ
UNIVERSITY

Mitigating the Impact of SQL Injection Attacks

A. Prevent Schema & Information Leaks
B. Limit Privileges (Defense-in-Depth)
C. Encrypt Sensitive Data stored in Database
D. Harden DB Server and Host O/S
E. Apply Early Input Validation

A. Prevent Schema & Information Leaks

- Knowing database schema makes attacker's job easier
- *Blind SQL Injection*: attacker attempts to interrogate system to figure out schema
- Prevent leakages of schema information
- Don't display detailed error messages and stack traces to external users

B. Limiting Privileges

- Apply Principle of Least Privilege! Limit
  - Read access, tables/views user can query
  - Commands (are updates/inserts ok?)
- No more privileges than typical user needs
- Ex: could prevent attacker from executing INSERT and DROP statements
  - But could still be able do SELECT attacks and compromise user data
  - Not a complete fix, but less damage

C. Encrypting Sensitive Data

- Encrypt data stored in the database
  - second line of defense
  - w/o key, attacker can't read sensitive info
- Key management precautions: don't store key in DB, attacker just SQL injects again to get it
- Some databases allow automatic encryption, but these still return plaintext queries!

Facebook Stored Hundreds of Millions of User Passwords in Plain Text for Years (21 MAR'19)

D. Hardening DB Server and Host O/S

- Dangerous functions could be on by default
- Ex: Microsoft SQL Server
  - Allowed users to open inbound/outbound sockets
  - Attacker could steal data, upload binaries, port scan victim’s network
- Disable unused services and accounts on OS (Ex: No need for web server on DB host)

E. Applying Early Input Validation

- Validation of query parameters not enough
- Validate all input early at *entry point* into code
- Reject overly long input (could prevent unknown buffer overflow exploit in SQL parser)
- Redundancy helps protect systems
  - E.g. if programmer forgets to apply validation for query input
  - Two lines of defense

Summary

- SQL injection attacks are important security threat that can
  - Compromise sensitive user data
  - Alter or damage critical data
  - Give an attacker unwanted access to DB

- **Key Idea**: Use diverse solutions, consistently!
  - Whitelisting input validation & escaping
  - Prepared Statements with bind variables