SQL Injection and Concurrency Control

This lecture includes materials taken from:

"What Every Web Programmer Needs To Know About Security", by Arkajit Dey and Neil Daswani. The content of this presentation is licensed under the Creative Commons 3.0 License.

"Database Replication," by Bettina Kemme, Ricardo Jiménez-Peris, Marta Patiño-Martínez. Morgan & Claypool Publishers

SQL Injection

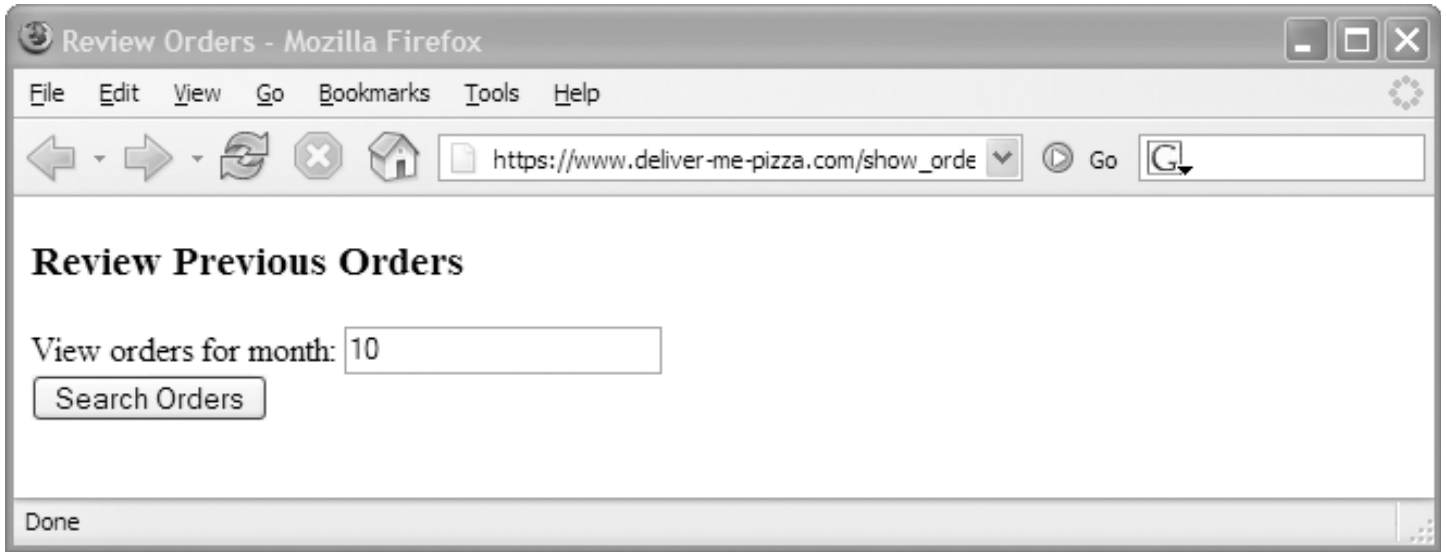
- Untrusted input inserted into query or command
- Attack string alters intended semantics of command

SQL Injection Real World Impact

- November 8, 2010 the British Royal Navy website was compromised by a hacker using SQL injection
- May 2012, the website for *Wurm Online*, a massively multiplayer online game, was shut down.
- July 2012, 450,000 login credentials stolen from Yahoo!
- October 2012, personal records of students, faculty, employees, and alumni from 53 universities published on pastebin.com

Attack Scenario (1)

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



  - HTTP request:

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

Attack Scenario (2)

- App constructs SQL query from parameter:

```
sql_query = "SELECT pizza, toppings, quantity, order_day " +
            "FROM orders " +
            "WHERE userid=" + $_SESSION['userId']+ " " +
            "AND order_month=" + $_REQUEST['mont'];
```

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
```

Attack Scenario (3)

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!

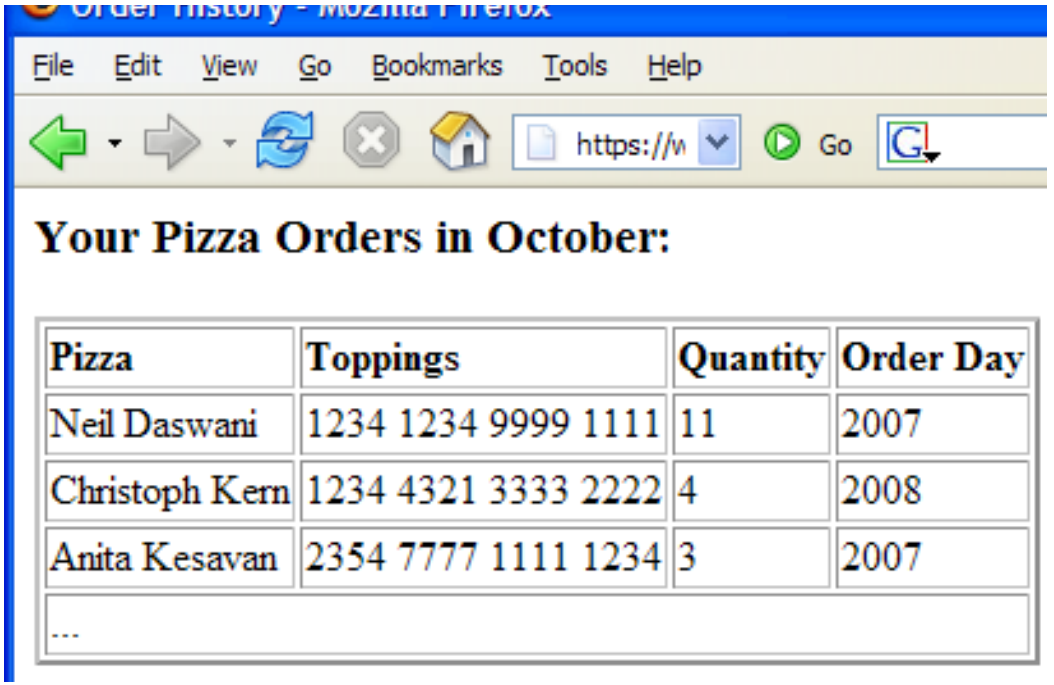
All User Data Compromised

Attack Scenario (4)

- 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



Attack Scenario (4)

- 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…

Attack Scenario (5)

- Injecting String Parameters: Topping Search

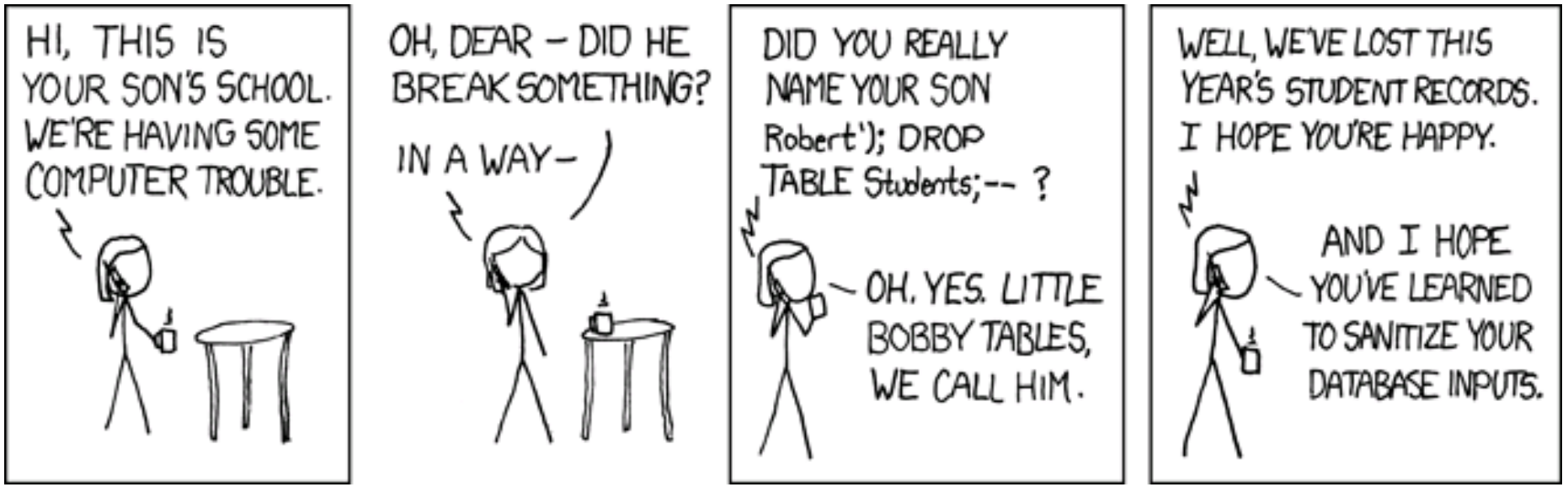
```
sql_query =
  "SELECT pizza, toppings, quantity, order_day " +
  "FROM orders " +
  "WHERE userid=" + $_SESSION[“userId”] + " " +
  "AND topping LIKE '%" + $_REQUEST["topping”] + "%' ";
```

- Attacker sets: `topping=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; --%'
```

Attack Scenario (6)



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

Solutions

- Variety of Techniques: Defense-in-depth
- Input Validation
- Escaping
- Prepared Statements
- Mitigate Impact

Input Validation

- 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

Escaping

- Escape quotes
- 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');
```

- Only works for string inputs
- Numeric parameters could still be vulnerable

CodeIgniter Escaping Support

- **$this->db->escape()**
  - Automatically adds single quotes around the data
  - $sql = "INSERT INTO table (title) VALUES(".$this->db->escape($title).")";
- Active Records
  - Values are escaped automatically by the system

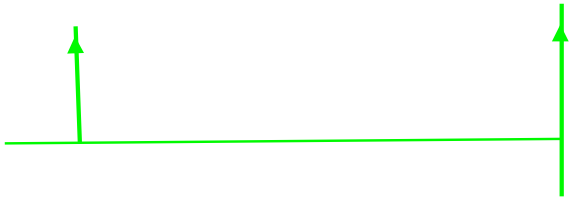
Parameterized Statements

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

CodeIgniter Query Bindings

```
$sql = "SELECT * FROM some_table WHERE id = ? AND status = ? ";

$this->db->query($sql, array(3, 'live''));
```



Bind Variable:
Data Placeholder

- Bind values are automatically escaped

Second-Order SQL Injection

- *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 + "'";
```

  - Username not escaped, b/c originally escaped before entering 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

- 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: `escape(uname)`

Mitigating the Impact of SQL Injection Attacks

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

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!
  - Input validation & escaping
  - Parameterized statements

Concurrency Control

- Multiple users interacting with web service simultaneously
- Multiple concurrent updates to database
- Challenge: Ensure that database remains consistent at all times

Example: Sale a ticket

$available ← select available from flight where id=$id

if ($available > 0)
    insert into ticket ….
    update flight set available = ($available-1)

else
    tell user the flight has sold out

Example: Sale a ticket

Schedule 1

User 1	User 2	avail
		1
$a ← avail		
if ($a> 0)		
insert ticket		0
$a → avail		
	$a ← avail	
	else	
	soldout	

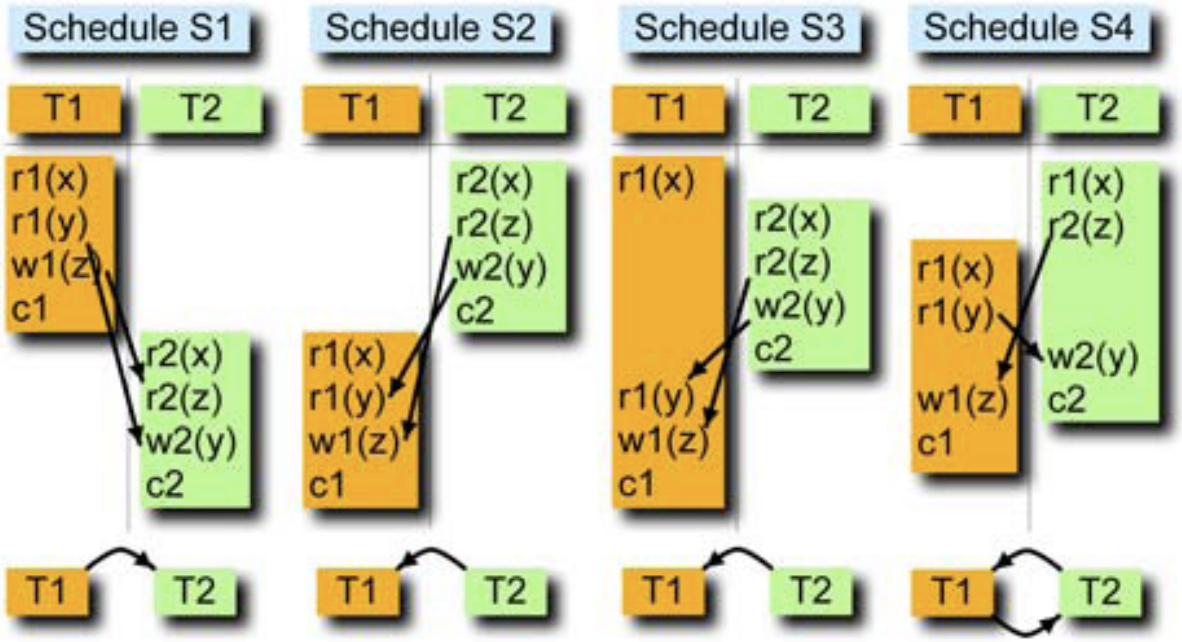
Schedule 2

User 1	User 2	avail
		1
$a ← avail		
if ($a> 0)	$a ← avail	
insert ticket		
$a → avail	if ($a> 0)	0
	insert ticket	
	$a → avail	0

Serializability

- Most well-known isolation model
- Provides strongest isolation level
- Concurrent execution of a set of transactions most be equivalent to some possible serial execution of the set
- Conflict: two operations that access the same item, are from different transactions, and at least 1 is a write
- To be serializable all conflicts have to execute in the same order
  - *Ti* has to appear to execute before *Tj* or *Tj* before *Ti*
  - Acyclic serialization graph

Serializability Examples

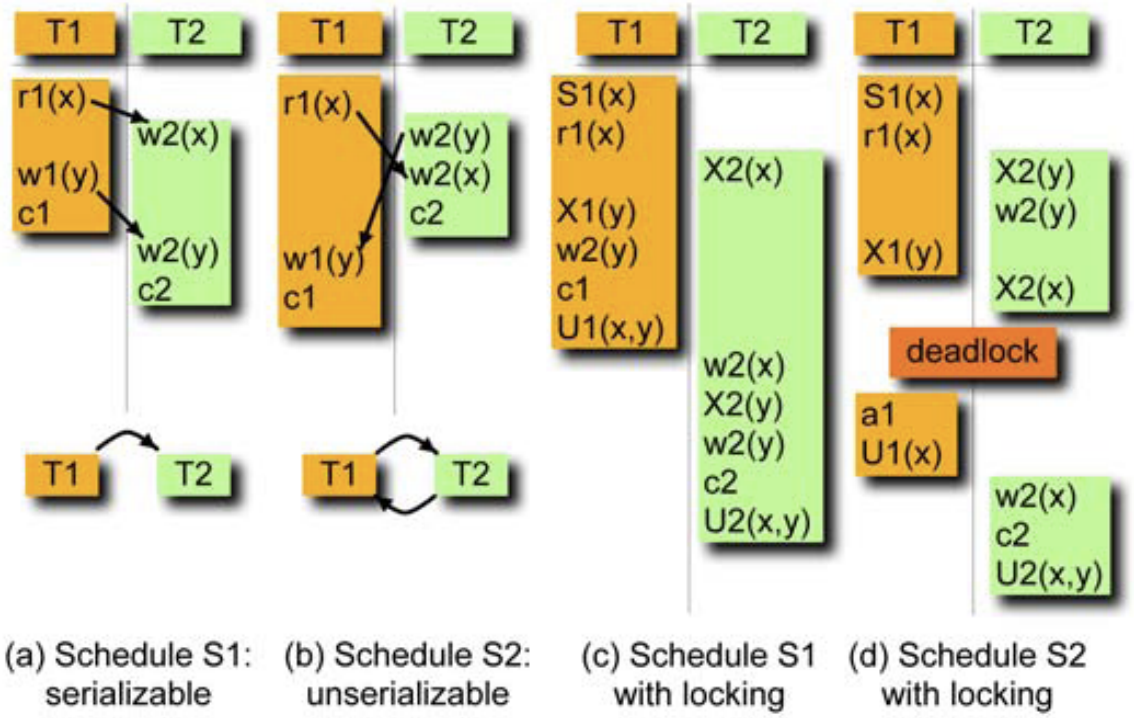


S1, S2 are serial executions
S3 is serializable
S4 is unserializable

Enforcing Serializability

- 2-phase locking
  - Acquire share lock before read
  - Acquire exclusive lock before write
  - Release all locks at end of transaction

2-Phase Locking Example



(a) Schedule S1: serializable

(b) Schedule S2: unserializable

(c) Schedule S1 with locking

(d) Schedule S2 with locking

MySQL Isolation Levels

- SERILIZABLE
- REPEATABLE READ (default)
  - By default reads do not acquire locks
  - A snapshot is created at transaction start
    - Multiple reads to the same item will return consistent value
    - Writes by others will not be seen
  - Possible to acquire explicit locks
    - SELECT * FROM sometable FOR UPDATE
    - SELECT * FROM sometable LOCK IN SHARE MODE