Me

• 20+ years experience
  • Application/SDK developer
  • Support, Training, Proj Mgmt
  • C, Java, Perl, PHP

• SQL maven

• Community contributor

• Author of new book *SQL Antipatterns*
SQL Injection

- Executing unintended SQL by interpolating dynamic content as part of your code:

  ```sql
  SELECT * FROM Bugs
  WHERE bug_id = $bug_id
  ```

  `user input`
SQL Injection

• Executing unintended SQL by interpolating dynamic content as part of your code:

```
SELECT * FROM Bugs
WHERE bug_id = 1234 OR TRUE
```
SQL Injection

- Compromises security in many ways:
  
  UPDATE Accounts
  SET password = SHA2('$password')
  WHERE account_id = $account_id
SQL Injection

- Compromises security in many ways:

  ```sql
  UPDATE Accounts
  SET password = SHA2('xyzzy'), is_admin=('1')
  WHERE account_id = 1234 OR TRUE
  ```

  changes password for all accounts
  changes account to administrator
Fallacy #1

“SQL Injection is an old problem—so I don’t have to worry about it.”
Biggest identity theft in history

- 130 million credit card numbers
- Albert Gonzalez used SQL Injection to install his packet-sniffer code onto credit-card servers
- Sentenced 20 years in March 2010
- Cost to victim company Heartland Payment Systems: $12.6 million

http://www.cio.com/article/492039/Security_Breach_Cost_Heartland_12.6_Million_So_Far
Other cases of SQL Injection

• (April 2008) Oklahoma Department of Corrections leaked tens of thousands of social security numbers to an SQL Injection attack.  

• (August 2008) 500,000 web pages affected with JavaScript malware using SQL Injection on Microsoft IIS and SQL Server  
  http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=9080580

• (December 2009) Facebook game maker RockYou! attacked using SQL Injection, exposing 32 million plaintext usernames and passwords  
“When hackers are required to work to gain access, SQL injection appears to be the uncontested technique of choice.

“In 2008, this type of attack ranked second in prevalence (utilized in 16 breaches) and first in the amount of records compromised (79 percent of the aggregate 285 million).”
Myth #2

“Quoting/escaping input prevents SQL injection.”
<?php

$password = $_GET['password'];
$password_quoted = $pdo->quote($password);

$id = intval($_GET['account']);

$sql = "UPDATE Accounts
    SET password = SHA2({$password_quoted})
    WHERE account_id = {$id}"
;

 PDO->query($sql);
<?php
$column = $_GET["order"];  
$column_delimited = $pdo->???($column);
$direction = $_GET["dir"];  
$sql = "SELECT * FROM Bugs ORDER BY {$column_delimited} {$direction}";
$pdo->query($sql);

no API supports a delimiter helper

no quoting for keywords
Myth #3

“I can write my own quoting code.”
Please Don’t

- Character set subtleties make this hard

- `addslashes()` isn’t good enough

- Please use driver-provided functions:
  - `mysql_real_escape_string()`
  - `PDO::quote()`
Fallacy #4

“Unsafe data comes from users—if it’s already in the database, then it’s safe.”
Not Necessarily

```
$sql = "SELECT product_name FROM Products";
$prodname = $pdo->query($sql)->fetchColumn();

$sql = "SELECT * FROM Bugs WHERE MATCH(summary, description) AGAINST ('{$prodname}')";

not safe input
```
Myth #5

“Using stored procedures prevents SQL Injection.”
CREATE PROCEDURE BugsOrderBy
(IN column_name VARCHAR(100),
IN direction VARCHAR(4))
BEGIN
  SET @query = CONCAT(
    'SELECT * FROM Bugs ORDER BY ',
    column_name, ', ', direction);
  PREPARE stmt FROM @query;
  EXECUTE stmt;
END

CALL BugsOrderBy('date_reported', 'DESC')
Dynamic SQL in Procedures

CREATE PROCEDURE QueryAnyTable
  (IN table_name VARCHAR(100))
BEGIN
  SET @query = CONCAT('SELECT * FROM ', table_name);
  PREPARE stmt FROM @query;
  EXECUTE stmt;
END

CALL QueryTable( 'SELECT * FROM ...' )

Fallacy #6

“Conservative SQL privileges limit the damage.”
User-supplied SQL

- Crash database server with one SELECT:
  
  ```sql
  SELECT * FROM Bugs JOIN Bugs JOIN Bugs JOIN Bugs JOIN Bugs JOIN Bugs JOIN Bugs
  ```

  100 bugs = 1 trillion rows
User-supplied SQL

- Filesort uses temporary disk space

```sql
SELECT * FROM Bugs JOIN Bugs JOIN Bugs
JOIN Bugs JOIN Bugs JOIN Bugs
ORDER BY 1
```
Fallacy #7

“It’s just an intranet application—it doesn’t need to be secure.”
What Stays on the Intranet?

- Your casual code could be copied & pasted into external applications

```sql
UPDATE Accounts
SET password = SHA2('$password')
WHERE account_id = $account_id
```

```sql
UPDATE Accounts
SET password = SHA2('$password')
WHERE account_id = $account_id
```
What Stays on the Intranet?

- You could be told to give business partners access to an internal application

```
UPDATE Accounts
SET password = SHA2('$password')
WHERE account_id = $account_id
```
What Stays on the Intranet?

- It’s hard to justify a security review/rewrite for a “finished” application

```
UPDATE Accounts
SET password = SHA2('$password')
WHERE account_id = $account_id
```
Myth #8

“My framework prevents SQL Injection.”
Dynamic SQL always risks SQL Injection, for example Rails ActiveRecord:

```ruby
Bugs.all(  
  :joins => "JOIN Accounts ON reported_by = account_id"
  :order => "date_reported DESC"
)
```
Whose Responsibility?

- No SQL database, connector, or framework can prevent SQL injection all the time
- Security is the application developer’s job
Fallacy #9

“Query parameters do quoting for you.”
Interpolating Dynamic Values

• Query needs a dynamic value:

```
SELECT * FROM Bugs
WHERE bug_id = $bug_id
```

*user input*
Using a Parameter

- Query parameter takes the place of a dynamic value:

```
SELECT * FROM Bugs
WHERE bug_id = ?
```
How the Database Parses It

query

SELECT expr-list

FROM simple-table

WHERE expr

equality

bug_id = ?

parameter placeholder
How the Database Executes It

SELECT expr-list * FROM simple-table bugs
WHERE expr = bug_id
    bug_id = 1234
SELECT expr-list *
FROM simple-table bugs
WHERE expr

OR

equality
1234

equality
1

SQL injection
SELECT expr-list * FROM simple-table bugs

WHERE expr equality bug_id = 1234 OR TRUE

no parameter can change the tree
Sequence of Prepare & Execute

Client
- prepare query
- execute query

Server
- parse query
- optimize query
- bind parameters
- execute query
- convert to machine-readable form

Client → Server:
- send SQL
- send parameters

Server → Client:
- return results

Repeat with different parameters
Myth #10

“Query parameters prevent SQL Injection.”
One Parameter = One Value

SELECT * FROM Bugs
WHERE bug_id = ?
Not in Other Cases

• Can’t use a parameter for a lists of values:

SELECT * FROM Bugs WHERE bug_id IN ( ? )
Not in Other Cases

• Can’t use a parameter for a table name:

```
SELECT * FROM ? WHERE bug_id = 1234
```
Not in Other Cases

- Can’t use a parameter for a column name:
  SELECT * FROM Bugs ORDER BY ?
Not in Other Cases

• Can’t use a parameter for an SQL keyword:

```
SELECT * FROM Bugs
ORDER BY date_reported  \\
```

‘ASC’ or ‘DESC’
## Interpolation vs. Parameters

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Example Value</th>
<th>Interpolation</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>single value</td>
<td>‘1234’</td>
<td><code>SELECT * FROM Bugs WHERE bug_id = $id</code></td>
<td><code>SELECT * FROM Bugs WHERE bug_id = ?</code></td>
</tr>
<tr>
<td>multiple values</td>
<td>‘1234, 3456, 5678’</td>
<td><code>SELECT * FROM Bugs WHERE bug_id IN ($list)</code></td>
<td><code>SELECT * FROM Bugs WHERE bug_id IN ( ?, ?, ? )</code></td>
</tr>
<tr>
<td>table name</td>
<td>‘Bugs’</td>
<td><code>SELECT * FROM $table WHERE bug_id = 1234</code></td>
<td>NO</td>
</tr>
<tr>
<td>column name</td>
<td>‘date_reported’</td>
<td><code>SELECT * FROM Bugs ORDER BY $column</code></td>
<td>NO</td>
</tr>
<tr>
<td>other keywords</td>
<td>‘DESC’</td>
<td><code>SELECT * FROM Bugs ORDER BY date_reported $direction</code></td>
<td>NO</td>
</tr>
</tbody>
</table>
Example: Fixing SQL Injection

http://www.example.com/order=date&dir=up

```php
<?php
$sortorder = $_GET['order'];
$direction = $_GET['dir'];

$sql = "SELECT * FROM Bugs
    ORDER BY {$sortorder} {$direction}";

$stmt = $pdo->query($sql);
```

Unsafe inputs:

- $sortorder
- $direction

SQL Injection
Fix with a Whitelist Map

<?php
$sortorders = array(
    "status" => "status",
    "date" => "date_reported"
);

$directions = array(
    "up" => "ASC",
    "down" => "DESC"
);

$sortorder = "bug_id";
$direction = "ASC";
if (array_key_exists('order', $sortorders))
{
    $sortorder = $sortorders['order'];
}
if (array_key_exists('dir', $directions))
{
    $direction = $directions['dir'];
}
$sql = "SELECT * FROM Bugs
ORDER BY $sortorder $direction";
$stmt = $pdo->query($sql);
1. I don’t have to worry about SQL injection
2. Quoting is the answer
3. I can implement quoting myself
4. Data in my database is safe to use
5. Stored procedures are the answer
6. SQL privileges are the answer
7. My app doesn’t need to be secure
8. Frameworks are the answer
9. Query parameters do quoting
10. Query parameters are the answer
SQL Antipatterns

• Shipping in July from Pragmatic Bookshelf
• Download the Beta e-book now

http://www.pragprog.com/titles/bksqla/