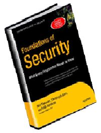
CHAPTER 8

SQL Injection



Agenda

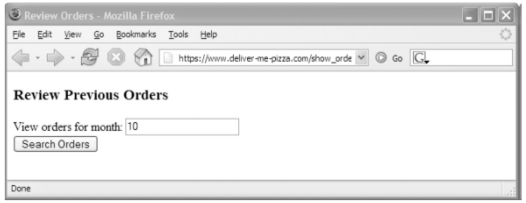
- *Command injection* vulnerability - untrusted input inserted into query or command
  - □ Attack string alters intended semantics of command
  - □ Ex: *SQL Injection* - unsanitized data used in query to back-end database (DB)
- SQL Injection Examples & Solutions
  - □ Type 1: compromises user data
  - □ Type 2: modifies critical data
  - □ Whitelisting over Blacklisting
  - □ Escaping
  - □ Prepared Statements and Bind Variables

SQL Injection Impact in the Real World

- 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
- Awareness Increasing: # of reported SQL injection vulnerabilities tripled from 2004 to 2005

8.1. Attack Scenario (1)

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


Review Orders - Mozilla Firefox
File
Edit
View
Go
Bookmarks
Tools
Help
https://www.deliver-me-pizza.com/show_orde
Go
Search Orders
Done

Review Previous Orders

View orders for month: 10

  - □ HTTP request:

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

8.1. Attack Scenario (2)

- App constructs SQL query from parameter:

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

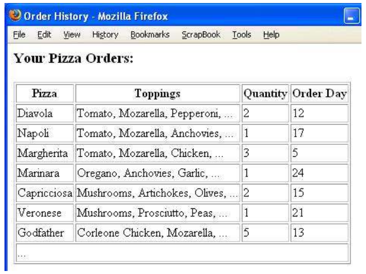
8.1. Attack Scenario (3)

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


Order History - Mozilla Firefox
File Edit View History Bookmarks ScrapBook Tools Help

Your Pizza Orders:

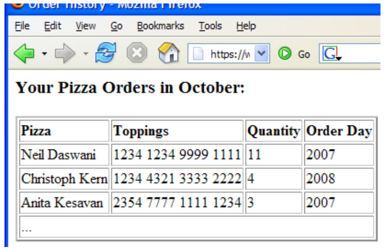
Pizza	Toppings	Quantity	Order Day
Diavola	Tomato, Mozarella, Pepperoni, ...	2	12
Napoli	Tomato, Mozarella, Anchovies, ...	1	17
Margherita	Tomato, Mozarella, Chicken, ...	3	5
Marinara	Oregano, Anchovies, Garlic, ...	1	24
Capricciosa	Mushrooms, Artichokes, Olives, ...	2	15
Veronese	Mushrooms, Prosciutto, Peas, ...	1	21
Godfather	Corleone Chicken, Mozarella, ...	5	13
...			

8.1. 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


Order History - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
https://w
Go

Your Pizza Orders in October:

Pizza	Toppings	Quantity	Order Day
Neil Daswani	1234 1234 9999 1111	11	2007
Christoph Kern	1234 4321 3333 2222	4	2008
Anita Kesavan	2354 7777 1111 1234	3	2007
...			

8.1. 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…

8.1. Attack Scenario (5)

- Injecting String Parameters: Topping Search

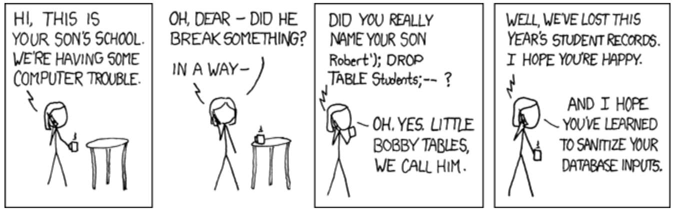
```
sql_query =
  "SELECT pizza, toppings, quantity, order_day " +
  "FROM orders " +
  "WHERE userid=" + session.getCurrentUserId() + " " +
  "AND topping LIKE '%" + request.getParamenter("topping") + "%' ";
```

- 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; --%'
```

8.1. Attack Scenario (6)


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.

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

8.2. Solutions

- Variety of Techniques: Defense-in-depth
- Whitelisting over Blacklisting
- Input Validation & Escaping
- Use Prepared Statements & Bind Variables
- Mitigate Impact

8.2.1. Why Blacklisting Does Not Work

- 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();
}
```

8.2.1. 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
- How to store O'Brien in DB if quotes blacklisted?

8.2.2. Whitelisting-Based Input Validation

- *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

8.2.3. 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

8.2.4. Second-Order SQL Injection (1)

- *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

8.2.4. Second-Order SQL Injection (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: `escape(uname)`

8.2.5. 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

8.2.5. 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)

8.2.5. 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 `$year` 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

8.2.5. 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));
```

8.2.6. Mitigating the Impact of SQL Injection Attacks

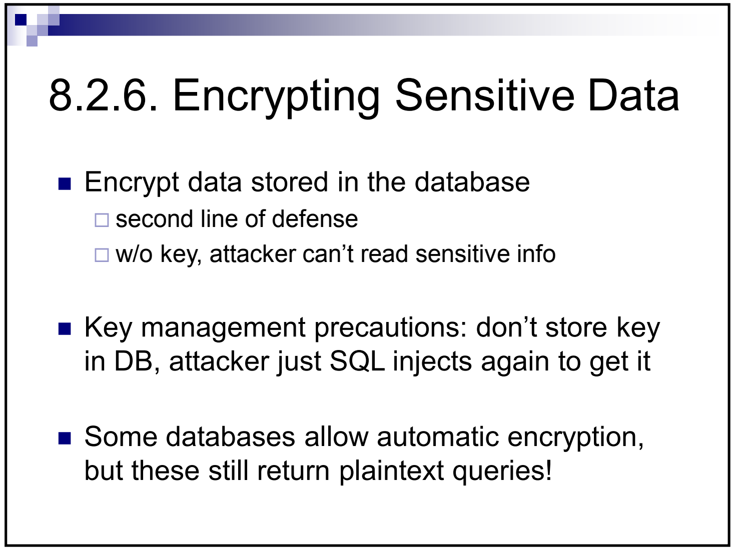
- Prevent Schema & Information Leaks
- Limit Privileges (Defense-in-Depth)
- Encrypt Sensitive Data stored in Database
- Harden DB Server and Host O/S
- Apply Input Validation

8.2.6. 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

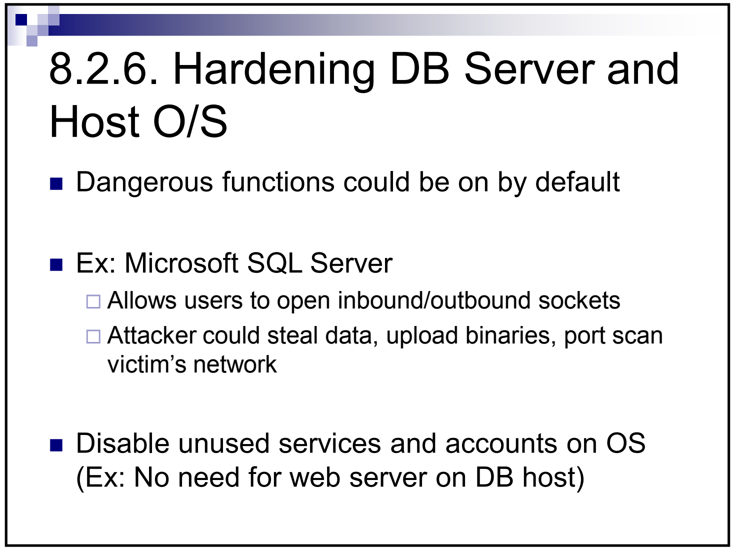
8.2.6. Limiting Privileges

- 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



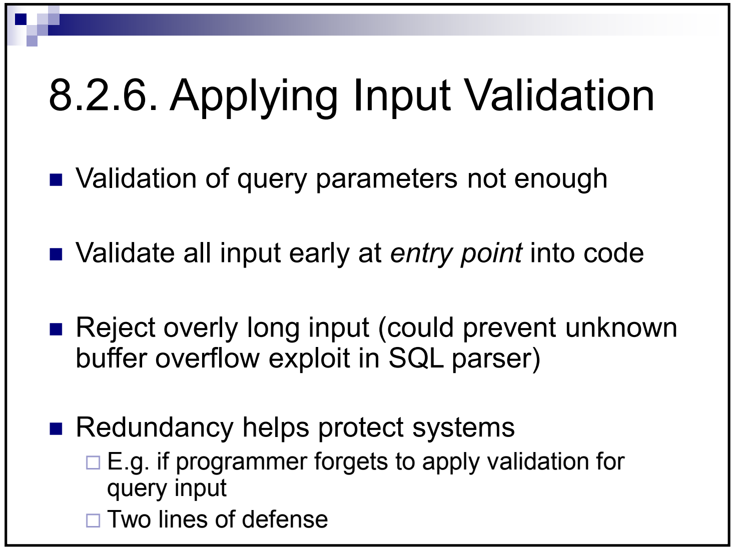
8.2.6. Encrypting Sensitive Data

- 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!



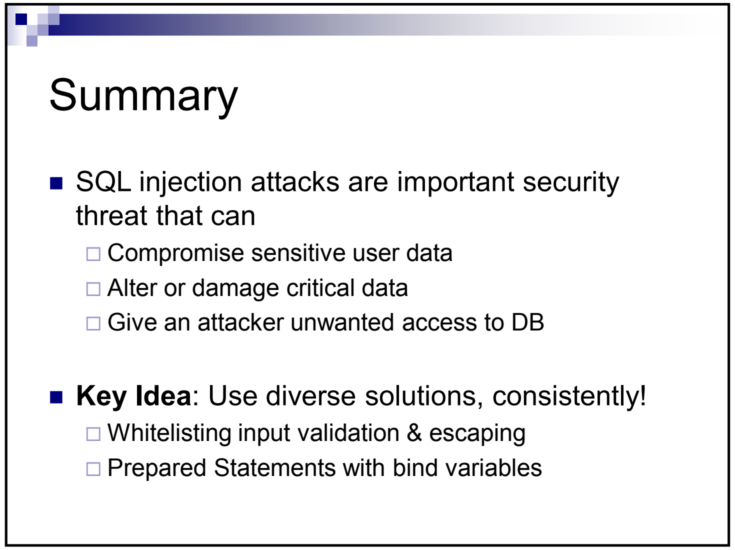
8.2.6. Hardening DB Server and Host O/S

- Dangerous functions could be on by default
- Ex: Microsoft SQL Server
  - Allows 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)



8.2.6. Applying 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