SQL Injection Attacks

Prof. Jim Whitehead
CMPS 183: Spring 2006
May 17, 2006
Context and Observations on this Slide Deck

- This slide deck was developed for use in a senior-level course providing an introduction to Web technologies and Web Engineering (UCSC CS 183) at UC Santa Cruz.
- The course was geared towards use of PHP and MySQL, and hence some examples in the deck identify specific PHP functions, or use PHP syntax.
- Students are assumed to already know SQL (a database course is prerequisite for CS 183)
- This was the first time this deck was used. It could be improved by adding a more in-depth scenario showing a SQL injection attack, and by showing a more in-depth code example protecting against SQL injection.
- Generally, students understood the concept of SQL injection, but were more shaky on how to integrate all of the techniques to guard against it in an actual Web application.
What is a SQL Injection Attack?

• Many web applications take user input from a form

• Often this user input is used literally in the construction of a SQL query submitted to a database. For example:
  – SELECT productdata FROM table WHERE productname = ‘user input product name’;

• A SQL injection attack involves placing SQL statements in the user input
An Example SQL Injection Attack

Product Search: `blah OR 'x' = 'x`

- This input is put directly into the SQL statement within the Web application:
  - `$query = "SELECT prodinfo FROM prodtable WHERE prodname = " . $_POST['prod_search'] . "";`
- Creates the following SQL:
  - `SELECT prodinfo FROM prodtable WHERE prodname = 'blah OR 'x' = 'x'`
  - Attacker has now successfully caused the entire database to be returned.
A More Malicious Example

• What if the attacker had instead entered:
  – `blah'; DROP TABLE prodinfo; --`

• Results in the following SQL:
  – `SELECT prodinfo FROM prodttable WHERE prodname = 'blah'; DROP TABLE prodinfo; --`
  – Note how comment (--) consumes the final quote

• Causes the entire database to be deleted
  – Depends on knowledge of table name
  – This is sometimes exposed to the user in debug code called during a database error
  – Use non-obvious table names, and never expose them to user

• Usually data destruction is not your worst fear, as there is low economic motivation
Other injection possibilities

• Using SQL injections, attackers can:
  – Add new data to the database
    • Could be embarrassing to find yourself selling politically incorrect items on an eCommerce site
    • Perform an INSERT in the injected SQL
  – Modify data currently in the database
    • Could be very costly to have an expensive item suddenly be deeply ‘discounted’
    • Perform an UPDATE in the injected SQL
  – Often can gain access to other user’s system capabilities by obtaining their password
Defenses

• Use provided functions for escaping strings
  – Many attacks can be thwarted by simply using the SQL string escaping mechanism
    • ‘ → \’ and “ → \
    – mysql_real_escape_string() is the preferred function for this

• Not a silver bullet!
  – Consider:
    • SELECT fields FROM table WHERE id = 23 OR 1=1
    – No quotes here!

• Can also use stripslashes()
  – Removes slashes in the input (i.e., makes it difficult to inject \n   or \r)
  – But, sometimes you want these characters…
This function escapes special characters in a string for use in SQL statements.

```
$esc_input = mysql_real_escape_string($user_input);
```

Specifically, it prepends backslashes to the following characters:
- `\x00`, `\n`, `\r`, `\`, `'`, `"` and `\x1a`.
- By escaping quotes, makes it much more difficult to perform SQL injection attacks

This function assumes a MySQL connection is active
- It considers the current character set used by the connection
More Defenses

• Check syntax of input for validity
  – Many classes of input have fixed languages
    • Email addresses, dates, part numbers, etc.
    • Verify that the input is a valid string in the language
    • Sometime languages allow problematic characters (e.g., ‘*’ in email addresses); may decide to not allow these
      • If you can exclude quotes and semicolons that’s good
  – Not always possible: consider the name Bill O’Reilly
    • Want to allow the use of single quotes in names

• Have length limits on input
  – Many SQL injection attacks depend on entering long strings
Even More Defenses

• Scan query string for undesirable word combinations that indicate SQL statements
  – INSERT, DROP, etc.
  – If you see these, can check against SQL syntax to see if they represent a statement or valid user input

• Limit database permissions and segregate users
  – If you’re only reading the database, connect to database as a user that only has read permissions
  – Never connect as a database administrator in your web application
More Defenses

• Configure database error reporting
  – Default error reporting often gives away information that is valuable for attackers (table name, field name, etc.)
  – Configure so that this information is never exposed to a user

• If possible, use bound variables
  – Some libraries allow you to bind inputs to variables inside a SQL statement
  – The mysqli library permits this
  – Allows limited type checking of parameters
  – Most useful for non-string parameters
mysqli Example

```php
<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "world");
$city = "Amersfoort";

/* create a prepared statement */
if ($stmt = mysqli_prepare($link, "SELECT District FROM City WHERE Name=?")) {
    /* bind parameters for markers */
    mysqli_stmt_bind_param($stmt, "s", $city);
    /* execute query */
    mysqli_stmt_execute($stmt);
    /* bind result variables */
    mysqli_stmt_bind_result($stmt, $district);
    /* fetch value */
    mysqli_stmt_fetch($stmt);
    printf("%s is in district %s
", $city, $district);
    /* close statement */
    mysqli_stmt_close($stmt);
}
?>

```