SQL Injection

COMP620

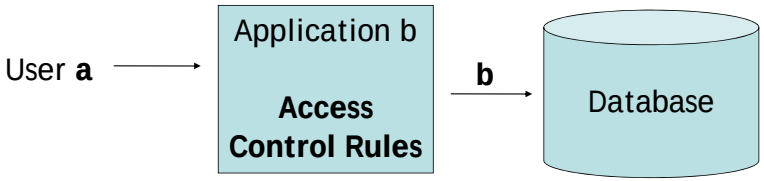
Serious Threat

- On August 17, 2009, the United States Justice Department charged an American citizen Albert Gonzalez and two unnamed Russians with the theft of 130 million credit card numbers using an SQL injection attack. It was reportedly "the biggest case of identity theft in American history"

from the BBC

Targets for Attack

- Database applications often need to serve multiple users
- Programmers often give their applications elevated privileges



Characterization of Attack

- Not a weakness of SQL
  - ...at least not in general
  - SQL Server may run with administrator privileges, and has commands for invoking shell commands
- Not a weakness of database, PHP/scripting languages, or Apache
- Building executable code using data from an untrusted user

Quick SQL Review (1)

- Querying tables:

```
select column1, column2 from table_name;
```

or

```
select * from table_name;
```

- Conditions:

```
select columns from table_name
  where condition;
```

Quick SQL Review (2)

- Inserting new rows:

```
insert into table_name values (value1,value2, ...);
```

or

```
insert into table_name set column1=value1,
  column2=value2, ...;
```

- Updating rows:

```
update table_name set column1=value1 where
  condition;
```

Quick SQL Review (3)

- Deleting rows:

```
delete from table_name where condition;
```

- Set values in conditions:

```
select * from table_name
  where column in (select_statement);
```

or

```
select * from table_name
  where column in (value1, value2, ...);
```

Quick SQL Review (4)

- Joining tables:

```
select * from table1, table2 where
  table1.attribute1 = table2.attribute2;
```

- Built-in Functions

```
select count(*) from test;
```

Quick SQL Review (5)

- Pattern Matching

```
select * from test where a like '%c_t%';
```

- Other Keywords

```
select * from test where a is null;
```

- Metadata Tables
  - Highly vendor-specific
  - Available tables, table structures are usually stored in some reserved table name(s).

Example Application

- This simple PHP program gets a name from a web form and creates a database query

```
$name = $HTTP_POST_VARS["name"];
$query = "select * from
  restaurants where name =
  '".$name."'";
$result = mysql_query($query);
```

Simple SQL Injection Example

- Logging in with:

```
select count(*) from login where
  username = '$username' and
  password = '$password';
```

- Setting the password to "x' or 'a' = 'a" gives:

```
select count(*) from login where
  username = 'alice' and
  password = 'x' or 'a' = 'a';
```

- In fact, username doesn't even have to match anyone in the database

Another Example

- Logging in with:

```
select count(*) from login where
  username = '$username' and
  password = '$password';
```

- Setting the password to

  "**x';insert into login set username='me' password = 'mySecret**" gives:

```
select count(*) from login where
  username = 'alice' and password = 'x';
  insert into login set username='me',
  password = 'mySecret';
```

Yet Another Example

- Logging in with:

```
select count(*) from login where
  username = '$username' and
  password = '$password';
```

- Setting the password to

  "**x'; update table_name set password = 'mySecret' where username = 'admin**" gives:

```
select count(*) from login where
   username = 'alice' and password = 'x';
   update table_name set password =
   'mySecret' where username = 'admin';
```

Inferring Database Layout

- The attacker may be able to determine the names of your tables and fields
- Guess at column names with input such as:

```
' and email is null--

' and email_addr is null--
```

- The presence or absence of error messages will tell the attacker if they are successful

Inferring Database Layout (2)

- Guess at table name

```
' and users.email_addr is null--

' and login.email_addr is null--
```

  - Can be done with an automated dictionary attack
  - Might discover more than one table in the query

- Guess at other table names

```
' and 1=(select count(*) from test)--
```

15

Input Verification

- Input should be checked for improper values
- It is much more effective to use pattern matching to recognize good input (white list) than to try to detect bad input (black list)
- May be tricky if we want to allow arbitrary text
- This can be much harder than it looks if you consider escaped characters, %27 is a quote
- International sites need to consider multiple character sets

No Special Characters Needed

- In the input value is numeric, it may not be necessary to use quotes or other special characters

```
SELECT fields FROM table
    WHERE id = 23 OR 1=1
```

- You may be able to scan the input string for undesirable word

Parameterized Statements

- Database systems allow you to create queries that take data values from variables.
- `PreparedStatement prep = conn.prepareStatement("SELECT * FROM USERS WHERE USERNAME=? AND PASSWORD=?");`
- `prep.setString(1, username);`
- `prep.setString(2, password);`
- `prep.executeQuery();`

System Oriented Prevention

- MySQL doesn't allow multiple queries per request
- Limit database privileges of application
- Run in non-admin user space to prevent system calls (*e.g.* MS SQL Server)
- Have length limits on input
- Hide error messages

Blind SQL Injection

- Some production systems display detailed error messages that give the attacker lots of information
- You can still use SQL injection if you only get a "True" or "False" response
- This can be a slow process, but there are tools to automate the attack

Benefits of an Automated Tool

- We can ask the server as many yes/no questions as we want
- Finding the first letter of a username with a binary search takes 7 requests
- Finding the full username if it's 8 characters takes 56 requests
- To find the username **is** 8 characters takes 6 requests
- 62 requests just to find the username
- This adds up

Automated Hacking Benefit

- Assuming it takes 10 seconds to make each request
- Assuming no mistakes are made, an8 character username takes over ten minutes
- What if we want the schema or the entire database?