SQL Injection Attacks

- Many web servers have backing databases
  - Much of their information stored in a database
- Web pages are built (in part) based on queries to a database
  - Possibly using some client input . . .

SQL Injection Mechanics

- Server plans to build a SQL query
- Needs some data from client to build it
  - E.g., client's user name
- Server asks client for data
- Client, instead, provides a SQL fragment
- Server inserts it into planned query
  - Leading to a "somewhat different" query

An Example

```
"select * from mysql.user
   where username = ' " . $uid . " ' and
   password=password(' ". $pwd " ');"
```

- Intent is that user fills in his ID and password
- What if he fills in something else?

  ```
  'or 1=1; -- '
  ```

What Happens Then?

- `$uid` has the string substituted, yielding

```
"select * from mysql.user
   where username = ' ' or 1=1; -- ' ' and
   password=password(' ". $pwd " ');"
```

- This evaluates to true
  - Since 1 does indeed equal 1
  - And `--` comments out rest of line
- If script uses truth of statement to determine valid login, attacker has logged in

Basis of SQL Injection Problem

- Unvalidated input
- Server expected plain data
- Got back SQL commands
- Didn't recognize the difference and went ahead
- Resulting in arbitrary SQL query being sent to its database
  - With its privileges

Some Example Attacks

- 130 million credit card numbers stolen in 2009 with SQL injection attack
- Used to steal 1 million Sony passwords
- Yahoo lost 450,000 passwords to a SQL injection in 2012
- Successful SQL injections on Bit9, British Royal Navy, PBS
- Ruby on Rails had built-in SQL injection vulnerability in 2012

Solution Approaches

- Carefully examine all input
- Use database access controls
- Randomization of SQL keywords
- Avoid using SQL in web interfaces
- Parameterized variables

Examining Input for SQL

- SQL is a well defined language
- Generally web input shouldn't be SQL
- So look for it and filter it out
- **Problem**: proliferation of different input codings makes the problem hard
- **Problem**: some SQL control characters are widely used in real data
  - E.g., apostrophe in names

Using Database Access Controls

- SQL is used to access a database
- Most databases have decent access control mechanisms
- Proper use of them limits damage of SQL injections
- **Problem**: may be hard to set access controls to prohibit all dangerous queries

Randomization of SQL Keywords

- Change all SQL keywords into something random
- Then translate all your internal queries to that new “language”
- Those trying SQL injection need to inject your language, not standard SQL
- **Problem**: security is based on a secret
- **Problem**: could cause unexpected errors from otherwise correct behavior

Avoid SQL in Web Interfaces

- Never build a SQL query based on user input to web interface
- Instead, use predefined queries that users can't influence
- Typically wrapped by query-specific application code
- **Problem**: may complicate development

Use Parameterized Variables

- SQL allows you to set up code so variables are bound parameters
- Parameters of this kind aren't interpreted as SQL
- Pretty much solves the problem, and is probably the best solution

Malicious Downloaded Code

- The web relies heavily on downloaded code
  - Full language and scripting language
  - Mostly scripts
- Instructions downloaded from server to client
  - Run by client on his machine
  - Using his privileges
- Without defense, script could do anything

Types of Downloaded Code

- Java
  - Full programming language
- Scripting languages
  - JavaScript
  - VB Script
  - ECMAScript
  - XSLT

Drive-By Downloads

- Often, user must request that something be downloaded
- But not always
  - Sometimes visiting a page or moving a cursor causes downloads
- These are called *drive-by downloads*
  - Since the user is screwed just by visiting the page

Solution Approaches

- Disable scripts in your browser
- Use secure scripting languages
- Isolation mechanisms
- Vista mandatory access control
- Virus protection and blacklist approaches

Disabling Scripts

- Browsers (or plug-ins) can disable scripts
  - Selectively, based on web site
- The bad script is thus not executed
- **Problem**: Cripples much good web functionality
  - So users re-enable scripting

Use Secure Scripting Languages

- Some scripting languages are less prone to problems than others
- Write your script in those
- **Problem**: secure ones aren't popular
- **Problem**: many bad things can still be done with "secure" languages
- **Problem**: can't force others to write their scripts in these languages

Isolation Mechanisms

- Architecturally arrange for all downloaded scripts to run in clean VM
  - Limiting the harm they can do
- **Problem**: they might be able to escape the VM
- **Problem**: what if a legitimate script needs to do something outside its VM?

Vista Mandatory Access Control

- In Vista, browser ran at low privilege level
- So scripts it downloaded did, too
- Limiting damage they could do
- **Problem**: also limited desirable things good scripts could do

Signatures and Blacklists

- Identify known bad scripts
- Develop signatures for them
- Put them on a blacklist and distribute it to others
- Before running downloaded script, automatically check blacklist
- **Problem**: same as for virus protection