Imperial College
London

CO331 – Network and Web Security

13. SQL injection

Dr Sergio Maffeis
Department of Computing
Course web page: http://www.doc.ic.ac.uk/~maffeis/331

SQL injection by example

Imperial College London

- If user submitted valid credentials, redirect to “authorized” page

```
$conn = mysql_connect("localhost","username","password");
$query = "SELECT userid FROM UsersTable WHERE user = '$_GET["user"]' "
       . "AND password = '$_GET["password"]'";
$result = mysql_query($query);
$rowcount = mysql_num_rows($result);
if ($rowcount ! = 0){header("Location: authorized.php");}
else {die('Incorrect username or password, please try again.')}
```

- HTTP request

```
http://www.example.com/login.php?user=foo&password=bar' OR '1' = '1
```

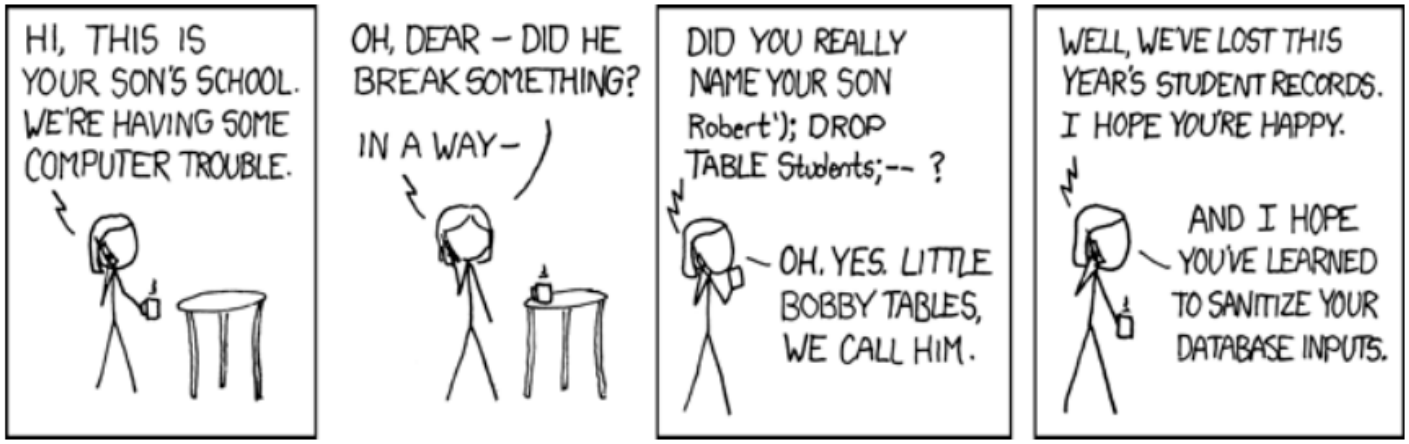
- Dynamic SQL query built from request parameters

```
SELECT userid FROM UsersTable WHERE user = 'foo'
AND password = 'bar' OR '1' = '1'
```

- WHERE condition is trivially true: attacker is redirected to `authorized.php`

SQL injection (SQLi)

Imperial College London



(xkcd.com)

- The most common example of command injection
  - It targets queries sent to a database, typically using SQL
  - 217 CVEs for SQLi in the last 3 months alone
  - Leads to large data losses: Equifax, TalkTalk hacks,
    - The SQLi Hall of Shame tries to keep track of main SQLi incidents
  - Typical systems run Oracle, SQL Server, MySQL, PostgreSQL
    - We use the latter two in examples
- Automated tools to detect and exploit SQLi
  - sqlmap, sqlix, sqlninja, and many others
  - We don't use tools, we do it by hand
- Many online SQLi cheatsheets and exploit lists

Scope of SQLi

Imperial College London

- Objectives
  - Elevation of privilege: bypass authentication
  - Information disclosure: read data that should not be accessible
  - Tampering: modify or delete data without permission
  - Denial of service: force the DB server to do costly operations
- Different ways to craft an exploit
  - Inputs can be URL parameters, HTTP headers, cookies, or other user input
  - Full range of SQL commands may be involved
    - `SELECT, INSERT, DROP, UNION, GROUPBY, …`

SQLi exploitation

Imperial College London

- Find out who you are, what privileges you have

```
SELECT user();
SELECT grantee,privilege_type FROM information_schema.user_privileges;
```

- Find out what data are available

```
SELECT table_schema,table_name,column_name FROM information_schema.columns
WHERE table_schema != 'mysql' AND table_schema != 'information_schema'
```

  - Shortcut: guess for common name: `accounts` table, `username` column, etc

- Data exfiltration using UNION statements
  - Number and type of columns must match
  - `NULL` for missing columns, convert data where possible `CAST('123' AS char)`
  - Example: exploit `products` to return `customers`

```
http://www.victim.com/products.asp?id=12+union+select
+userid,first_name,second_name,NULL+from+customers
```

```
SELECT id,type,name,price FROM products WHERE id = 12 UNION
SELECT userid,first_name,second_name,NULL FROM customers
```

ID	Type	Description	Price
12	Book	SQL Injection Attacks	50
1	Charles	Smith	
2	Lydia	Clayton	

Blind SQLi

Imperial College London

- Web-based interaction with a database may not display data as a response
- Example: web based survey
  - User submits a HTTP/ POST request with form data
  - Data is stored in a database
  - The web app replies with 200 OK "thank you"
- It may still be possible to identify if the web app is vulnerable to SQL injection using *side channels*
  - Most commonly, the time it takes to serve a response
  - Also error messages can help identify the vulnerability
- Try payloads that cause a delay in processing
  - For example `SLEEP()` in MYSQL or `pg_sleep()` in PostreSQL
  - If response takes longer than normal, the injection *might* have been successful
- Data may have to be exfiltrated a bit at a time
  - Easy to ask "*is the admin password equal to Arsenal1?*"
  - Time consuming to ask "*what is the password of admin?*"
    - Each bit of the password may take a separate query (+ delay)

Second-order SQLi

Imperial College London

- Untrusted data is handled securely when it is first inserted in the database, avoiding injection
- Other application components later read the data from the database, assuming it does not need sanitisation
  - And use it as part of a new query
  - Or apply a transformation and put it back in the database
- User may submit payloads that are dangerous only on the second usage
- Example
  - Attacker registers username "`admin' --`"
  - Password reset code

```
pwd = escape(request.getParameter("new_password"))
usr = session.getUsername();
Sql = "UPDATE USERS SET passwd='"+ pwd
+ ' WHERE uname = '" + usr + "'"
```

  - Attacker ask to reset his password to `hacked`

```
UPDATE USERS SET passwd='cracked' WHERE uname = 'admin' --'
```

SQLi countermeasures

Imperial College London

- Input filtering
  - Escape black-listed characters
    - For example, with PHP function `mysqli_real_escape_string()`
  - Hard to capture all user input
  - Hard to escape correctly across multiple trust boundaries
    - HTTP request parameters may be passed across different server modules before reaching a SQL query
    - Different modules of a web application may transform parameters in different ways
- Prepared statements
  - Avoid building SQL commands piece-wise from strings

```
$stmt = $dbh->prepare("SELECT * FROM REGISTRY WHERE name = ? AND age = ?");
$stmt->bind_param('si', $_GET['name'], $_GET['age']);
if ($stmt->execute()) { while ($row = $stmt->fetch()) { print_r($row); }}
```

- Stored procedures
  - Parameterised SQL queries stored in the database
    - So the DB offers a fixed "API" to the application
  - Need to be carefully programmed to avoid injection on themselves
  - Risk: may run with higher privileges (execution) that usual queries

SQLi countermeasures

Imperial College London

- Static/dynamic analysis of PHP code
  - Type systems
    - Ensure that a query parameter is of the expected type
    - A string cannot become a string + a SQL command
  - Taint analysis
    - Detect if an untrusted input can reach the database without passing via a sanitization function
- Rate-limit web server, or database server requests
  - Obvious performance trade-off
- IDS-style defense in front of the database
  - Detect and stop sequences of suspicious queries
- Rely on a programming framework
  - Framework may have been carefully developed, reviewed and tested
  - Drawbacks
    - Sometimes vulnerabilities are inherited from the framework itself
    - Framework may add unnecessary functionality increasing the size of trusted computing base
    - Users rarely understand all details of framework code, and consequences of using it

Injection action plan

Imperial College London

- Identify what parameters of a request you can control
  - Can you submit arbitrary values?
- Submit input that is likely to be problematic for the application that should interpret it
  - PHP, SQL, Bash, …
- Observe any changes in response content and time
  - Often error messages leak valuable information
- Submit further inputs based on the information you discover until you are sure there is a vulnerability
  - Proof-of-concept exploit that confirms vuln without disrupting target
- Consider how to leverage the vulnerability to achieve your goals
  - Even if your injected code can read data or execute remote commands, it may be tricky to send data back home
- Exploit the vulnerability