# 2550 Intro to cybersecurity

## L20-21: Sql & its vulnerabilities

abhi shelat

Key insight: security vulnerabilities arise when external input is not verified.

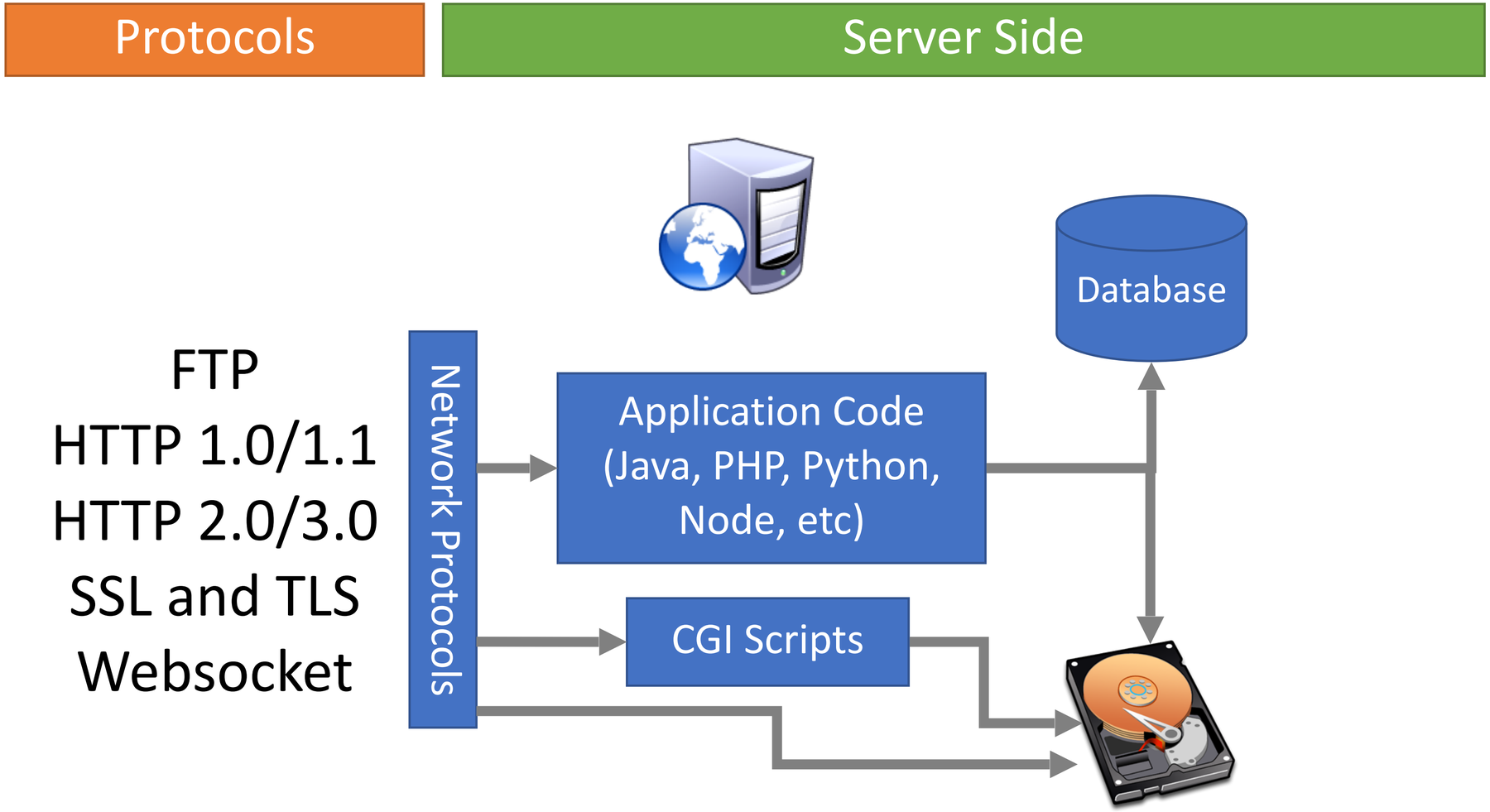
# Structured Query Language (SQL)

CREATE, INSERT, UPDATE

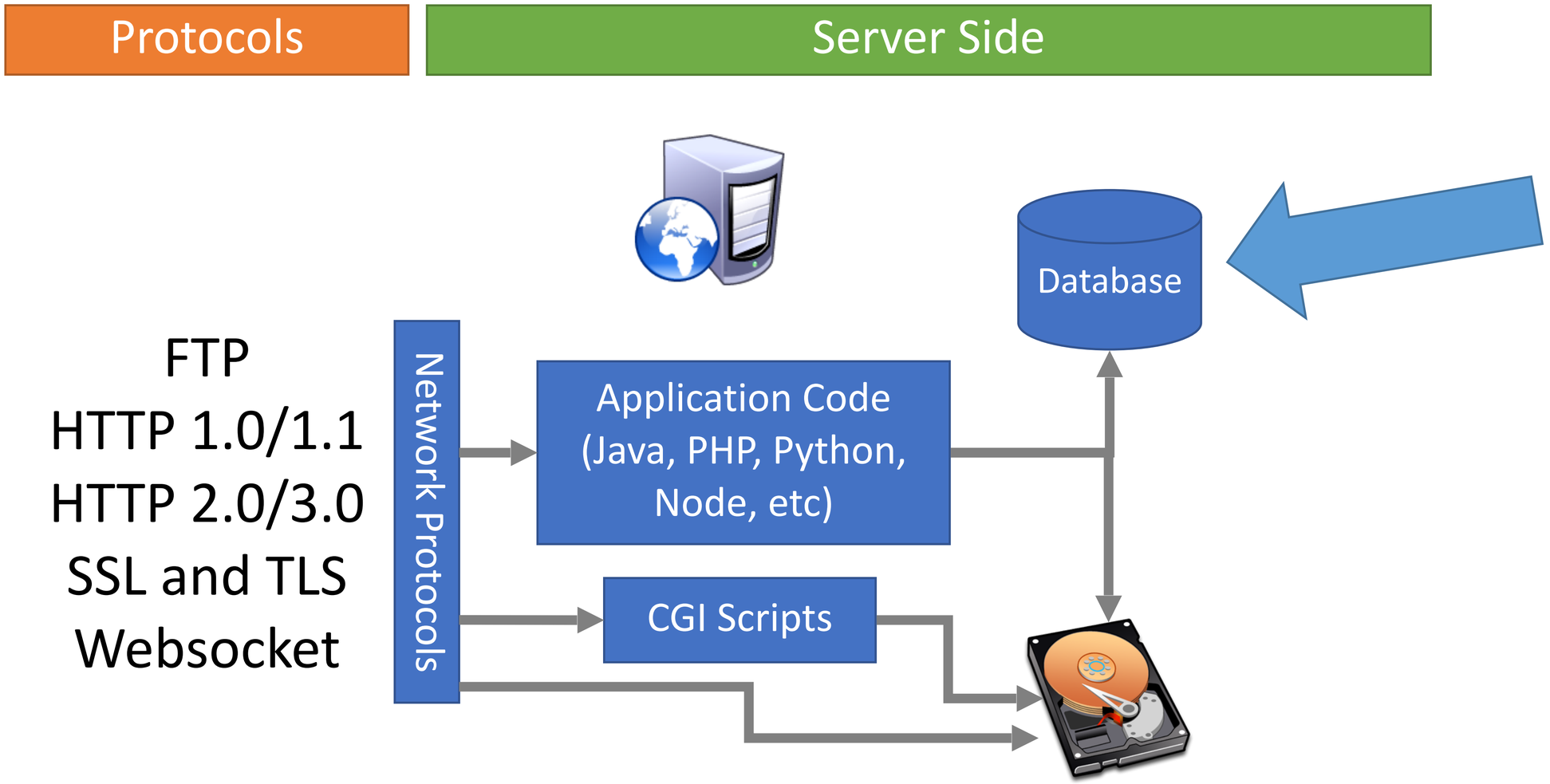
SELECT

Thanks to Christo for sharing slides!

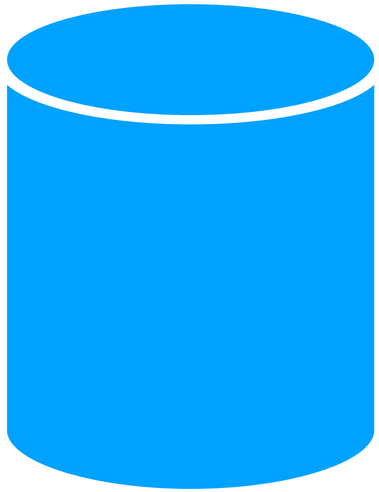
# Older Web Architecture



# Older Web Architecture



# What is a database?



Often drawn
like this

# SQL

- Structured Query Language
  - Relatively simple declarative language
  - Define relational data
  - Operations over that data
- Widely supported: MySQL, Postgres, Oracle, sqlite, etc.
- Why store data in a database?
  - Persistence – DB takes care of storing data to disk
  - Concurrency – DB can handle many requests in parallel
  - Transactions – simplifies error handling during complex updates

# SQL Operations

- Common operations:
  - CREATE TABLE makes a new table
  - INSERT adds data to a table
  - UPDATE modifies data in a table
  - DELETE removes data from a table
  - SELECT retrieves data from one or more tables
- Common SELECT modifiers:
  - ORDER BY sorts results of a query
  - UNION combines the results of two queries

# CREATE

- Syntax

  CREATE TABLE name (column1_name *type*, column2_name *type*, ...);

- Data types
  - TEXT – arbitrary length strings
  - INTEGER
  - REAL – floating point numbers
  - BOOLEAN

# CREATE

- Syntax

  CREATE TABLE name (column1_name *type*, column2_name *type*, …);

- Data types
  - TEXT – arbitrary length strings
  - INTEGER
  - REAL – floating point numbers
  - BOOLEAN
- Example

  CREATE TABLE people (name TEXT, age INTEGER, employed BOOLEAN);

**People:**

| name (string) | age (integer) | employed (boolean) |
|---|---|---|

# INSERT

- Syntax

  INSERT INTO name (column1, column2, …) VALUES (val1, val2, …);

- Example

  INSERT INTO people (name, age, employed) VALUES (“abhi”, 78, True);

**People:**

| name (string) | age (integer) | employed (boolean) |
|---|---|---|

# INSERT

- Syntax

  INSERT INTO name (column1, column2, …) VALUES (val1, val2, …);

- Example

  INSERT INTO people (name, age, employed) VALUES (“abhi”, 78, True);

**People:**

| name (string) | age (integer) | employed (boolean) |
|---|---|---|
| Abhi | 78 | True |

# SELECT

- Syntax

  SELECT col1, col2, ... FROM name WHERE condition ORDER BY col1, col2, ...;

- Example

  SELECT * FROM people;

**People:**

| name (string) | age (integer) | employed (boolean) |
|---|---|---|
| Abhi | 78 | True |
| Alice | 29 | True |
| Bob | 41 | False |

# SELECT

- Syntax

  SELECT col1, col2, ... FROM name WHERE condition ORDER BY col1, col2, ...;

- Example

  SELECT * FROM people;

  SELECT name, age FROM people;

**People:**

| name (string) | age (integer) |
|---|---|
| Abhi | 78 |
| Alice | 29 |
| Bob | 41 |

# SELECT

- Syntax

  SELECT col1, col2, ... FROM name WHERE condition ORDER BY col1, col2, ...;

- Example

  SELECT * FROM people;

  SELECT name, age FROM people;

  SELECT * FROM people WHERE name="abhi" OR name="Alice";

**People:**

| name (string) | age (integer) | employed (boolean) |
|---|---|---|
| Abhi | 78 | True |
| Alice | 29 | True |

# SELECT

- Syntax

  SELECT col1, col2, … FROM name WHERE condition ORDER BY col1, col2, …;

- Example

  SELECT * FROM people;

  SELECT name, age FROM people;

  SELECT * FROM people WHERE name=“abhi” OR name=“Alice”;

  SELECT name FROM people ORDER BY age;

**People:**

| name (string) |
| --- |
| Alice |
| Bob |
| Abhi |

# UPDATE

- Syntax

  UPDATE name SET column1=val1, column2=val2, … WHERE condition;

- Example

  UPDATE people SET age=42 WHERE name=“Bob”;

**People:**

| name (string) | age (integer) | employed (boolean) |
|---|---|---|
| Abhi | 78 | True |
| Alice | 29 | True |
| Bob | 41 | False |

# UPDATE

- Syntax

  UPDATE name SET column1=val1, column2=val2, … WHERE condition;

- Example

  UPDATE people SET age=42 WHERE name=“Bob”;

**People:**

| name (string) | age (integer) | employed (boolean) |
|---|---|---|
| Abhi | 78 | True |
| Alice | 29 | True |
| Bob | 42 | False |

# UNION

- Syntax

  SELECT col1, col2, … FROM name1 UNION SELECT col1, col2, … FROM name2;

- Example

  SELECT * FROM people UNION SELECT * FROM dinosaurs;

**People:**

| name (string) | age (integer) | employed (boolean) |
|---|---|---|
| Abhi | 78 | True |
| Alice | 29 | True |

# UNION

- Syntax

  SELECT col1, col2, … FROM name1 UNION SELECT col1, col2, … FROM name2;

- Example

  SELECT * FROM people UNION SELECT * FROM dinosaurs;

**People:**

| name (string) | age (integer) | employed (boolean) |
|---|---|---|
| Abhi | 78 | True |
| Alice | 29 | True |

| name (string) | weight (integer) | extinct (boolean) |
|---|---|---|
| Tyrannosaurus | 14000 | True |
| Brontosaurus | 15000 | True |

# UNION

- Syntax

  SELECT col1, col2, … FROM name1 UNION SELECT col1, col2, … FROM name2;

- Example

  SELECT * FROM people UNION SELECT * FROM dinosaurs;

**People:**

| name (string) | age (integer) | employed (boolean) |
|---|---|---|
| Abhi | 78 | True |
| Alice | 29 | True |

| name (string) | weight (integer) | extinct (boolean) |
|---|---|---|
| Tyrannosaurus | 14000 | True |
| Brontosaurus | 15000 | True |

Note: number of columns must match (and sometimes column types)

# Comments

- Syntax

  command; -- comment

- Example

  SELECT * FROM people; -- This is a comment

**People:**

| name (string) | age (integer) | employed (boolean) |
|---|---|---|
| Abhi | 78 | True |
| Alice | 29 | True |
| Bob | 41 | False |

# Get access to the lecture machine

ssh <github username>@l18.neucrypt.org

Requires you to have your ssh key uploaded to Github.

# Exercise 1

Write the sql that inserts a row into dinosaurs that adds the lightest non-avian theropod:
'Parvicursor remotus’ that weighs 1 kg

# SQL Injection

Blind Injection

Mitigations

# SQL Injection

SQL queries often involve untrusted data

- App is responsible for interpolating user data into queries
- Insufficient sanitization could lead to modification of query semantics

Possible attacks

- Confidentiality – modify queries to return unauthorized data
- Integrity – modify queries to perform unauthorized updates
- Authentication – modify query to bypass authentication checks

# Server Threat Model

Attacker's goal:

- Steal or modify information on the server

Server's goal: protect sensitive data

- Integrity (e.g. passwords, admin status, etc.)
- Confidentiality (e.g. passwords, private user content, etc.)

Attacker's capability: submit arbitrary data to the website

- POSTed forms, URL parameters, cookie values, HTTP request headers

# Threat Model Assumptions

Web server is free from vulnerabilities

- Apache, nginx, caddy are pretty reliable

No file inclusion vulnerabilities

Server OS is free from vulnerabilities

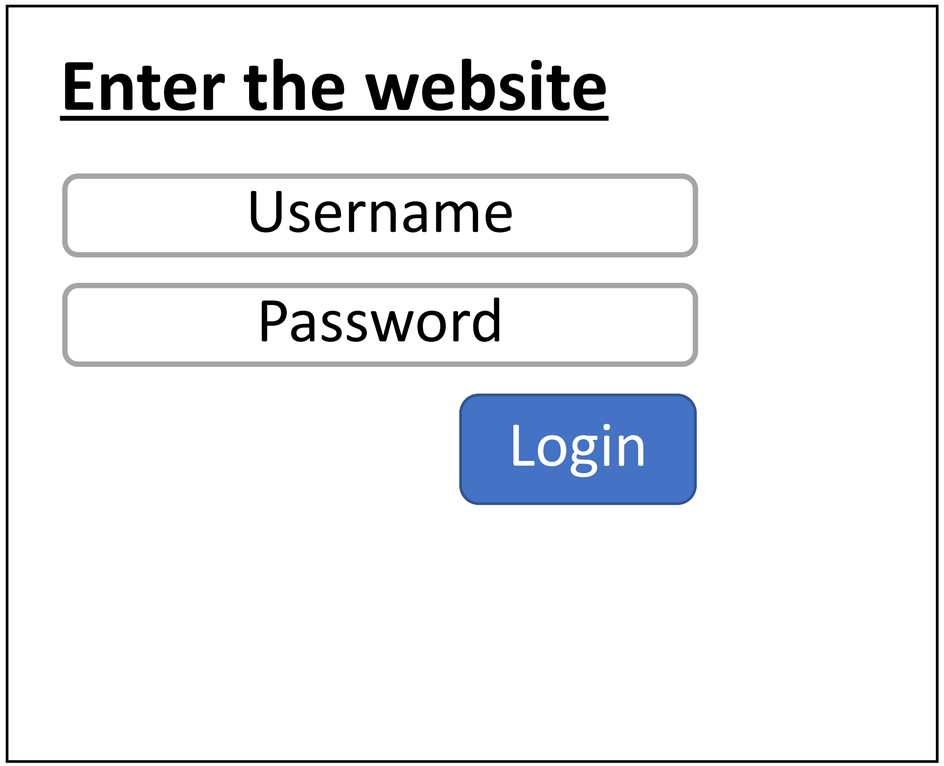
- No remote code exploits

Remote login is secured

- No brute forcing the admin’s SSH credentials

# Website Login Example

Client-side




Server-side

```
if flask.request.method == 'POST':
  db = get_db()
  cur = db.execute(
     'select * from user_tbl where
      user="%s" and pw="%s";' % (
        flask.request.form['username'],
        flask.request.form['password']))
  user = cur.fetchone()
  if user == None:
    error = 'Invalid username or password'
  else:
    …
```

# Login Examples

```
'select * from user_tbl where user="%s" and pw="%s";'
```

| form[‘username’] | form[‘password’] | Resulting query |
| --- | --- | --- |
| | | |
| | | |
| | | |
| | | |

# Login Examples

```
'select * from user_tbl where user="%s" and pw="%s";'
```

| form[‘username’] | form[‘password’] | Resulting query |
|---|---|---|
| alice | 123456 | ‘… where user="alice" and pw="123456";’ |
| bob | qwerty1# | ‘… where user="bob" and pw="qwery1#";’ |

# Login Examples

```
'select * from user_tbl where user="%s" and pw="%s";'
```

| form[‘username’] | form[‘password’] | Resulting query |
|---|---|---|
| alice | 123456 | ‘… where user="alice" and pw="123456";’ |
| bob | qwerty1# | ‘… where user="bob" and pw="qwery1#";’ |
| goofy | a"bc | ‘… where user="goofy" and pw="a"bc";’ |

# Login Examples

```
'select * from user_tbl where user="%s" and pw="%s";'
```

| form[‘username’] | form[‘password’] | Resulting query |
|---|---|---|
| alice | 123456 | ‘… where user="alice" and |
| bob | qwerty1# | ‘… where user="bob" and pw="qwerty1#";’ |
| goofy | a"bc | ‘… where user="goofy" and pw="a"bc";’ |

Incorrect syntax, too many double quotes. Server returns 500 error.

# Login Examples

```
'select * from user_tbl where user="%s" and pw="%s";'
```

| form[‘username’] | form[‘password’] | Resulting query |
|---|---|---|
| alice | 123456 | ‘… where user="alice" and pw="123456";’ |
| bob | qwerty1# | ‘… where user="bob" and pw="qwery1#";’ |
| goofy | a"bc | ‘… where user="goofy" and pw="a"bc";’ |
| weird | abc" or pw="123 | ‘… where user=“weird" and pw="abc" or pw="123";’ |

# Login Examples

```
'select * from user_tbl where user="%s" and pw="%s";'
```

| form[‘username’] | form[‘password’] | Resulting query |
|---|---|---|
| alice | 123456 | ‘… where user="alice" and pw="123456";’ |
| bob | qwerty1# | ‘… where user="bob" and pw="qwery1#";’ |
| goofy | a"bc | ‘… where user="goofy" and pw="a"bc";’ |
| weird | abc" or pw="123 | ‘… where user=“weird" and pw="abc" or pw="123";’ |
| eve | " or 1=1; -- | ‘… where user=“eve" and pw="" or 1=1; --";’ |

# Login Examples

```
'select * from user_tbl where user="%s" and pw="%s";'
```

| form[‘username’] | form[‘password’] | Resulting query |
|---|---|---|
| alice | 123456 | ‘… where user="alice" and pw="123456";’ |
| bob | qwerty1# | ‘… where user="bob" and pw |
| goofy | a"bc | ‘… where user="goofy" and |
| weird | abc" or pw="123 | ‘… where user=“weird" and pw="a or pw="123";’ |
| eve | " or 1=1; -- | ‘… where user=“eve" and pw="" or 1=1; --";’ |

1=1 is always true ;)
-- comments out extra quote

# Login Examples

```
'select * from user_tbl where user="%s" and pw="%s";'
```

| form[‘username’] | form[‘password’] | Resulting query |
|---|---|---|
| alice | 123456 | ‘… where user="alice" and pw="123456";’ |
| bob | qwerty1# | ‘… where user="bob" and pw="qwery1#";’ |
| goofy | a"bc | ‘… where user="goofy" and pw="a"bc";’ |
| weird | abc" or pw="123 | ‘… where user=“weird" and pw="abc" or pw="123";’ |
| eve | " or 1=1; -- | ‘… where user=“eve" and pw="" or 1=1; --";’ |
| mallory"; -- | | ‘… where user="mallory"; --" and pw="";’ |

# Login Examples

```
'select * from user_tbl where user="%s" and pw="%s";'
```

| form[‘username’] | form[‘password’] | Resulting query |
|---|---|---|
| alice | 123456 | ‘… where user="alice" and pw="123456";’ |
| bob | qwerty1# | ‘… where user="bob" and pw="qwery1#";’ |
| goofy | a"bc | ‘… where user="goofy" and |
| weird | abc" or pw="123 | ‘… where user=“weird" and |
| eve | " or 1=1; -- | ‘… where user=“eve" and pw= -- ;’ |
| mallory"; -- | | ‘… where user="mallory"; --" and pw="";’ |

None of this is evaluated. Who needs password checks? ;)



# KEEP CALM AND HACK ON

keep-calm.net

# Blind SQL Injection

Basic SQL injection requires knowledge of the schema

- e.g., knowing which table contains user data…
- And the structure (column names) of that table

Blind SQL injection leverages information leakage

- Used to recover schemas, execute queries

Requires some observable indicator of query success or failure

- e.g., a blank page (success/true) vs. an error page (failure/false)

Leakage performed bit-by-bit

# SQL Injection Examples

**Original query:**

```
"SELECT name, description FROM items WHERE id='" + req.args.get('id', '') + "'"
```

# SQL Injection Examples

**Original query:**

```
“SELECT name, description FROM items WHERE id=‘” + req.args.get(‘id’, ‘’) + “’”
```

**Result after injection:**

```
SELECT name, description FROM items WHERE id='12'
    UNION SELECT username, passwd FROM users;--';
```

# SQL Injection Examples

**Original query:**

```
“SELECT name, description FROM items WHERE id=‘” + req.args.get(‘id’, ‘’) + “’”
```

**Result after injection:**

```
SELECT name, description FROM items WHERE id='12'
    UNION SELECT username, passwd FROM users;--';
```

**Original query:**

```
“UPDATE users SET passwd=‘” + req.args.get(‘pw’, ‘’) + “’ WHERE user=‘” + req.args.get(‘user’, ‘’)
                                                                    + “‘”
```

# SQL Injection Examples

**Original query:**

```
“SELECT name, description FROM items WHERE id=‘” + req.args.get(‘id’, ‘’) + “’”
```

**Result after injection:**

```
SELECT name, description FROM items WHERE id='12'
    UNION SELECT username, passwd FROM users;--';
```

**Original query:**

```
“UPDATE users SET passwd=‘” + req.args.get(‘pw’, ‘’) + “’ WHERE user=‘” + req.args.get(‘user’, ‘’)
                                                                    + “‘”
```

**Result after injection:**

```
UPDATE users SET passwd='...' WHERE user='dude' OR 1=1;--';
```

# SQL Injection Examples

**Original query:**

```
“SELECT name, description FROM items WHERE id=‘” + req.args.get(‘id’, ‘’) + “’”
```

**Result after injection:**

```
SELECT name, description FROM items WHERE id='12'
    UNION SELECT username, passwd FROM users;--';
```

**Original query:**

```
“UPDATE users SET passwd=‘” + req.args.get(‘pw’, ‘’) + “’ WHERE user=‘” + req.args.get(‘user’, ‘’)
                                                                    + “‘”
```

**Result after injection:**

```
UPDATE users SET passwd='...' WHERE user='dude' OR 1=1;--';
```

# SQL Injection Examples

**Original query:**

```
“SELECT name, description FROM items WHERE id=‘” + req.args.get(‘id’, ‘’) + “’”
```

**Result after injection:**

```
SELECT name, description FROM items WHERE id='12'
  UNION SELECT username, passwd FROM users;--';
```

**Original query:**

```
“UPDATE users SET passwd=‘” + req.args.get(‘pw’, ‘’) + “’ WHERE user=‘” + req.args.get(‘user’, ‘’)
                                                                + “‘”
```

**Result after injection:**

```
UPDATE users SET passwd='...' WHERE user='dude' OR 1=1;--';
```

- Problem arises when delimiters are unfiltered

# SQL Injection Examples

**Original query:**

```
SELECT * FROM users WHERE id=$user_id;
```

**Result after injection:**

```
SELECT * FROM users WHERE id=1 UNION SELECT ... --;
```

- Vulnerabilities also arise from improper validation
  - e.g., failing to enforce that numbers are valid

# SQL Injection Defenses

```
SELECT * FROM users WHERE user='{{sanitize($id)}}';
```

- Sanitization
- Prepared statements
  - Trust the database to interpolate user data into queries correctly
- Object-relational mappings (ORM)
  - Libraries that abstract away writing SQL statements
  - Java – Hibernate
  - Python – SQLAlchemy, Django, SQLObject
  - Ruby – Rails, Sequel
  - Node.js – Sequelize, ORM2, Bookshelf
- Domain-specific languages
  - LINQ (C#), Slick (Scala), ...

# Practical demonstrations

# Example (vulnerable)

```
#!/usr/bin/python3
import sqlite3
import sys

if len(sys.argv) < 2:
    print(f'Usage: {sys.argv[0]} <name of person to query>')
    exit()

conn = sqlite3.connect('step3.db')
c = conn.cursor()

query = f'SELECT * FROM people WHERE name="{sys.argv[1]}";'
print(f'Query to be executed: {query}')

for row in c.execute(query):
    print(row)
```

# Example (vulnerable)

```
#!/usr/bin/python3
import sqlite3
import sys

if len(sys.argv) < 2:
    print(f'Usage: {sys.argv[0]} <name of person to query>')
    exit()

conn = sqlite3.connect('step3.db')
c = conn.cursor()

query = f'SELECT * FROM people WHERE name="{sys.argv[1]}";'
print(f'Query to be executed: {query}')
for row in c.execute(query):
    print(row)
```

Result after injection: `SELECT * FROM people WHERE name="`

# Better

```
#!/usr/bin/python3
import sqlite3
import sys

if len(sys.argv) < 2:
    print(f'Usage: {sys.argv[0]} <name of person to query>')
    exit()

conn = sqlite3.connect('step3.db')
c = conn.cursor()

query = f'SELECT * FROM people WHERE name=?;'
arg = (sys.argv[1],)
print(f'Query to be executed: {query}')

for row in c.execute(query, arg):
    print(row)
```

# Class CTF

```
$ /home/abhi/nov20/vuln Alice
FOUND

$ /home/abhi/nov20/vuln someone-who-does-not-exist
NOT FOUND
```

`vuln` is supposed to let you "look up a student by name", but it answers only **one bit** of information: `FOUND` or `NOT FOUND`. It never prints the query, the matching row, or any data.

It is `setuid` program, and `secret.db` is not readable by you.

Your job: there is a secret table (with a secret, unknown name) holding a flag in one of its rows. Recover the flag using all of the techniques discussed in class.

The broken line:

```
snprintf(query, sizeof(query),
         "SELECT * FROM students WHERE name=\"%s\"", argv[1]);
```

Because the only feedback is FOUND/NOT FOUND, you cannot use UNION to dump data onto the screen like the earlier `bad.py` demo.

Use the yes/no answer as an oracle and reconstruct data character by character.

# Hint

```
$ vuln "\" OR 1=2 --"    # → NOT FOUND, as expected
```

Two sql ideas:

```
length(X) > N
```

```
substr(X, POS, 1) = "c"
```