IERG 4130 – 2026 Fall

香港中文大學
CUHK

SQL Injection

Yajin ZHOU

https://yajin.org

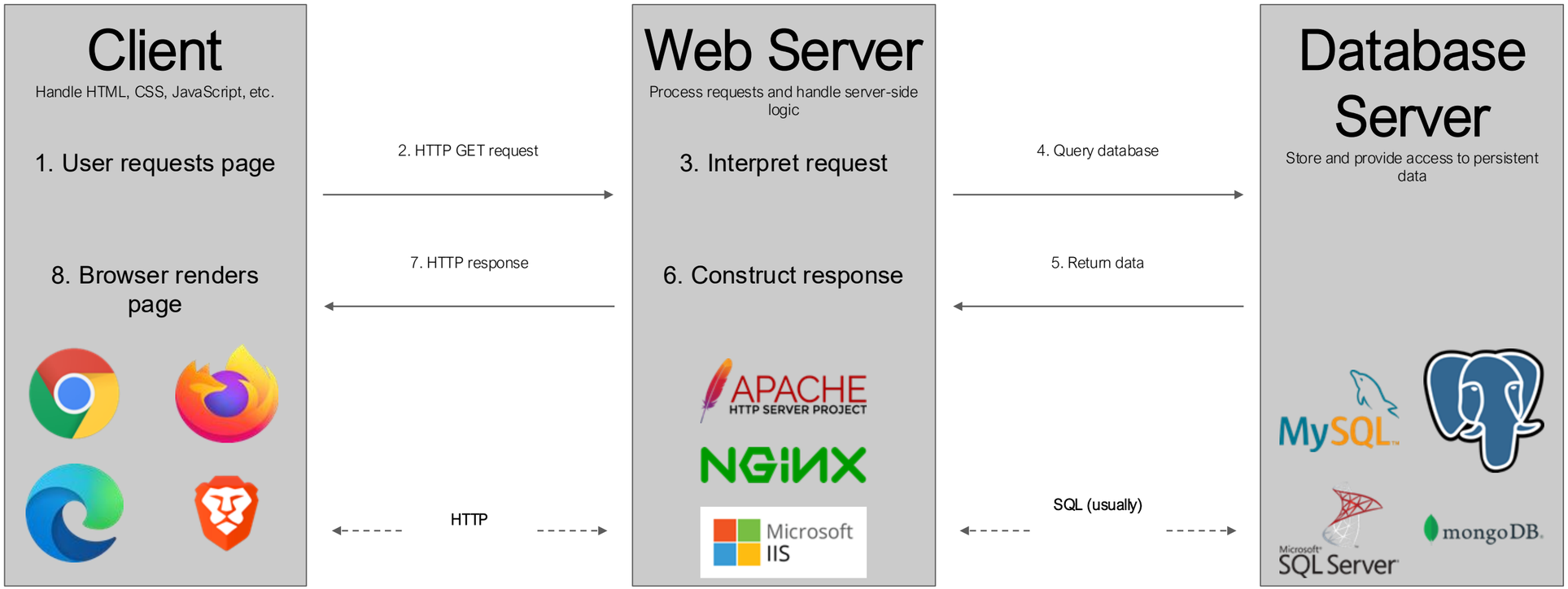
Credits: the slides are from CS161 of UC Berkeley

Top 25 Most Dangerous Software Weaknesses (2020)



Rank	ID	Name	Score
[1]	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	46.82
[2]	CWE-787	Out-of-bounds Write	46.17
[3]	CWE-20	Improper Input Validation	33.47
[4]	CWE-125	Out-of-bounds Read	26.50
[5]	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	23.73
[6]	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	20.69
[7]	CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	19.16
[8]	CWE-416	Use After Free	18.87
[9]	CWE-352	Cross-Site Request Forgery (CSRF)	17.29
[10]	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	16.44
[11]	CWE-190	Integer Overflow or Wraparound	15.81
[12]	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	13.67
[13]	CWE-476	NULL Pointer Dereference	8.35
[14]	CWE-287	Improper Authentication	8.17
[15]	CWE-434	Unrestricted Upload of File with Dangerous Type	7.38
[16]	CWE-732	Incorrect Permission Assignment for Critical Resource	6.95
[17]	CWE-94	Improper Control of Generation of Code ('Code Injection')	6.53

Structure of Web Services



SQL Databases



- **A SQL database contains multiple tables**
- **Each table is like a spreadsheet: one row per entry, one column per field**
- **Provides ACID (atomicity, consistency, isolation, and durability)**
  - Essentially ensures that the database will never crash partway through a partial operation, return an invalid state, or be vulnerable to race conditions

bots			
id	**name**	**likes**	**age**
1	evanbot	pancakes	3
2	codabot	hashes	2.5
3	pintobot	beans	1.5
3 rows, 4 columns			

SQL



- **Structured Query Language** (**SQL**): The language used to interact with and manage data stored in a database
  - Defined by the International Organization for Standardization (ISO) and implemented by many SQL servers
- **Good SQL servers are ACID (atomicity, consistency, isolation, and durability)**
  - Essentially ensures that the database will never store a partial operation, return an invalid state, or be vulnerable to race conditions
- **Declarative programming language, rather than imperative**
  - Declarative: Use code to define the result you want
  - Imperative: Use code to define exactly what to do (e.g. C, Python, Go)

SQL: SELECT



- SELECT is used to select some columns from a table
- Syntax:
  SELECT [columns] FROM [table]

bots

id	name	likes	age
1	evanbot	pancakes	3
2	codabot	hashes	2.5
3	pintobot	beans	1.5

3 rows, 4 columns

SQL: SELECT



- **SELECT name, age FROM bots**

Selected 2 columns from the table, keeping all rows.

name	age
evanbot	3
codabot	2.5
pintobot	1.5
3 rows, 2 columns	

bots

id	name	likes	age
1	evanbot	pancakes	3
2	codabot	hashes	2.5
3	pintobot	beans	1.5
3 rows, 4 columns			

SQL: SELECT



- **SELECT * FROM bots**

The asterisk (*) is shorthand for “all columns.”
Select all columns from the table, keeping all rows.

id	name	likes	age
1	evanbot	pancakes	3
2	codabot	hashes	2.5
3	pintobot	beans	1.5
3 rows, 4 columns			

bots

id	name	likes	age
1	evanbot	pancakes	3
2	codabot	hashes	2.5
3	pintobot	beans	1.5
3 rows, 4 columns			

SQL: WHERE

- **WHERE can be used to filter out certain rows**
  - Arithmetic comparison: **<**, **<=**, **>**, **>=**, **=**, **<>**
  - Arithmetic operators: **+**, **-** , * , **/**
  - Boolean operators: **AND**, **OR**
    - AND has precedence over OR

bots			
id	**name**	**likes**	**age**
1	evanbot	pancakes	3
2	codabot	hashes	2.5
3	pintobot	beans	1.5
3 rows, 4 columns			

SQL: WHERE

SELECT * FROM bots WHERE likes = 'pancakes'

Choose only the rows where the likes column has value pancakes

id	name	likes	age
1	evanbot	pancakes	3
1 row, 4 columns			

bots

id	name	likes	age
1	evanbot	pancakes	3
2	codabot	hashes	2.5
3	pintobot	beans	1.5
3 rows, 4 columns			

A Go HTTP Handler (Again)



Handler

```
func handleGetItems(w http.ResponseWriter, r *http.Request) {
    itemName := r.URL.Query()["item"][0]
    db := getDB()
    query := fmt.Sprintf("SELECT name, price FROM items WHERE name = '%s'", itemName)
    row, err := db.QueryRow(query)
    ...
}
```

Remember this string manipulation issue?

URL

https://vulnerable.com/get-items?item=matcha

Query

SELECT item, price FROM items WHERE name = 'matcha'

A Go HTTP Handler (Again)



Handler

```
func handleGetItems(w http.ResponseWriter, r *http.Request) {
    itemName := r.URL.Query()["item"][0]
    db := getDB()
    query := fmt.Sprintf("SELECT name, price FROM items WHERE name = '%s'", itemName)
    row, err := db.QueryRow(query)
    ...
}
```

URL

https://vulnerable.com/get-items?item=‘

Query

SELECT item, price FROM items WHERE name = ‘’

Invalid SQL executed by the server, 500 Internal Server Error

A Go HTTP Handler (Again)



Handler

```
func handleGetItems(w http.ResponseWriter, r *http.Request) {
    itemName := r.URL.Query()["item"][0]
    db := getDB()
    query := fmt.Sprintf("SELECT name, price FROM items WHERE name = '%s'", itemName)
    row, err := db.QueryRow(query)
    ...
}
```

This is essentially OR TRUE, so returns every item!

URL

https://vulnerable.com/get-items?item=‘ OR ‘1’ = ‘1

Query

SELECT item, price FROM items WHERE name **= '' OR '1' = '1'**

A Go HTTP Handler (Again)

Handler

```
func handleGetItems(w http.ResponseWriter, r *http.Request) {
    itemName := r.URL.Query()["item"][0]
    db := getDB()
    query := fmt.Sprintf("SELECT name, price FROM items WHERE name = '%s'", itemName)
    row, err := db.QueryRow(query)
    ...
}
```

For this payload: End the first quote ('), then start a new statement (DROP TABLE items), then comment out the remaining quote (--)

URL

https://vulnerable.com/get-items?item **='; DROP TABLE items --**

Query

SELECT item, price FROM items WHERE name **= '; DROP TABLE items --'**

SQL Injection



- **SQL injection** (**SQLi**): Injecting SQL into queries constructed by the server to cause malicious behavior
  - Typically caused by using vulnerable string manipulation for SQL queries
- **Allows the attacker to execute arbitrary SQL on the SQL server!**
  - Leak data
  - Add records
  - Modify records
  - Delete records/tables
  - Basically anything that the SQL server can do

SQL Injection Defenses



- **Defense: Input sanitization / Output escaping**
  - Option #1: Disallow (filter out / sanitize) special characters
  - Option #2: Escape special characters
    - Like XSS, SQL injection relies on certain characters that are interpreted specially
    - SQL allows special characters to be escaped with backslash (\) to be treated as data
- **Drawback: Difficult to build a good escaper that handles all edge cases**
  - You should *never* try to build one yourself!

```
func handleGetItems(w http.ResponseWriter, r *http.Request) {
    itemName := r.URL.Query()["item"][0]
    itemName = sqlEscape(itemName)
    db := getDB()
    query := fmt.Sprintf("SELECT name, price FROM items WHERE name = '%s'", itemName)
    row, err := db.QueryRow(query)
    ...
}
```

SQL Injection Defenses



- **Defense: Prepared statements**
  - SQL statements can contain a placeholder, represented a question mark (**?**), that will be filled in with arguments (passed separately) at runtime
  - If you pass untrusted input in the arguments (rather than in the SQL statement), it will only be treated as data and never be parsed

```
func handleGetItems(w http.ResponseWriter, r *http.Request) {
    itemName := r.URL.Query()["item"][0]
    db := getDB()
    row, err := db.QueryRow("SELECT name, price FROM items WHERE name = ?", itemName)
    ...
}
```