

CSIT 5740 Introduction to Software Security

Note set 5B

Dr. Alex LAM



DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING

The set of note is adapted and converted from a software security course by Prof. Antonio Bianchi and Prof. David Wagner

Cookie summary

Summary of Cookies

- Cookie: a piece of data used to maintain state across multiple requests
 - Set by the browser or server
 - Stored by the browser
 - Attributes: Name, value, domain, path, secure, HttpOnly, expires
- Cookie policy
 - Server with **domain X** can set a cookie with **domain attribute Y** if the **domain attribute** is a **domain suffix** of the **server's domain**, and the **domain attribute Y** is not a top-level domain (TLD)
 - The browser attaches a cookie on a request if the **domain attribute** is a **domain suffix** of the **server's domain**, and the **path attribute** is a **prefix** of the **server's path**
 - Cookie domain: **example.com** and cookie path: **/some/path** will be included in request to `https://foo/example.com/some/path/index.html`

Maintaining Sessions using cookies

Session Authentication

- **Session:** A sequence of requests and responses associated with the same authenticated user
 - Example: When you check all your unread emails, you make many requests to Gmail. The Gmail server needs a way to know all these requests are from you
 - When the session is over (you log out, or the session expires), future requests are not associated with you
- Naïve solution: Type your username and password before each request
 - Problem: Very inconvenient for the user!
- Better solution: Is there a way the browser can automatically send some information in a request for us?
 - Yes: Cookies!

Session Tokens

- **Session token:** A secret value used to associate requests with an authenticated user
- The first time you visit the website:
 - Present your username and password
 - If they're valid, you receive a session token
 - The server associates you with the session token
- When you make future requests to the website:
 - Attach the session token in your request
 - The server checks the session token to figure out that the request is from you
 - No need to re-enter your username and password!

Session Tokens with Cookies

- Session tokens can be implemented with cookies
 - Cookies can be used to save *any* state across requests (e.g. language, display mode, etc)
 - Session tokens are just one way to use cookies
- The first time you visit a website:
 - Make a request with your username and password
 - If they're valid, the server sends you a cookie with the session token
 - The server associates you with the session token
- When you make future requests to the website:
 - The browser attaches the session token cookie in your request
 - The server checks the session token to figure out that the request is from you
 - No need to re-enter your username and password!
- When you log out (or when the session times out):
 - The browser and server delete the session token

Session Tokens: Security

- If an attacker steals your session token, they can log in as you!
 - The attacker can make requests and attach your session token
 - The browser will think the attacker's requests come from you
- Servers need to generate session tokens *randomly* and *securely*
- Browsers need to make sure malicious websites cannot steal session tokens
 - Enforce isolation with cookie policy and same-origin policy
- Browsers should not send session tokens to the wrong websites
 - Enforced by cookie policy

Session Token Cookie Attributes

- What attributes should the server set for the session token?
 - Domain and Path: Set so that the cookie is only sent on requests that require authentication
 - Secure: Can set to True so the cookie is only sent over secure HTTPS connections
 - HttpOnly: Can set to True so JavaScript can't access session tokens
 - Expires: Set so that the cookie expires when the session times out

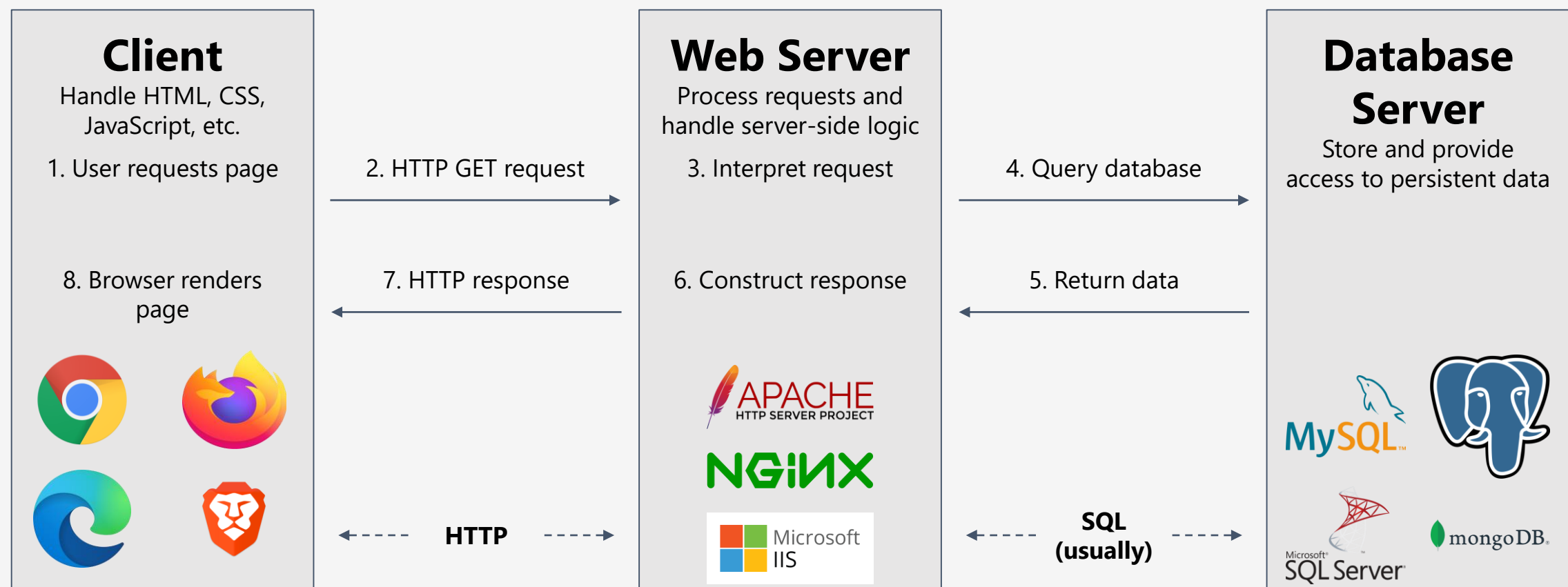
Name	token
Value	{random value}
Domain	mail.google.com
Path	/
Secure	True
HttpOnly	True
Expires	{15 minutes later}
(other fields omitted)	

***Web attack
example:
SQL Injection***

Structure of Web Services

- Most websites need to **store and retrieve data**
 - Examples: User accounts, comments, prices, etc.
- The HTTP server only handles the HTTP requests, and it needs to have some way of storing and retrieving persisted data

Structure of Web Services



Databases

- We will cover very simple SQL
 - SQL = Structured Query Language
 - Each database has a number of tables
 - Each table has a predefined structure, so it has columns for each field and rows for each entry
- Database server manages access and storage of these databases

<i>students</i>			
id	name	likes	age
1	Tom	Games	21
2	Emily	Tennis	22
3	Mike	Travel	22
3 rows, 4 columns			

- **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
- 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]

<i>students</i>			
id	name	likes	age
1	Tom	Games	21
2	Emily	Tennis	22
3	Mike	Travel	22
3 rows, 4 columns			

SQL: SELECT

Selected 2 columns from the table, keeping all rows.

```
SELECT name, age FROM students
```

name	age
Tom	21
Emily	22
Mike	22
3 rows, 2 columns	

<i>students</i>			
id	name	likes	age
1	Tom	Games	21
2	Emily	Tennis	22
3	Mike	Travel	22
3 rows, 4 columns			

SQL: SELECT

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

SELECT * FROM students

id	name	likes	age
1	Tom	Games	21
2	Emily	Tennis	22
3	Mike	Travel	22
3 rows, 4 columns			

<i>students</i>			
id	name	likes	age
1	Tom	Games	21
2	Emily	Tennis	22
3	Mike	Travel	22
3 rows, 4 columns			

SQL: SELECT

Select constants instead of columns

```
SELECT 'CSIT', '5740' FROM students
```

id	name
CSIT	5740
CSIT	5740
CSIT	5740
3 rows, 2 columns	

<i>students</i>			
id	name	likes	age
1	Tom	Games	21
2	Emily	Tennis	22
3	Mike	Travel	22
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

<i>students</i>			
id	name	likes	age
1	Tom	Games	21
2	Emily	Tennis	22
3	Mike	Travel	22
3 rows, 4 columns			

SQL: WHERE

Choose only the rows where the **likes** column has value **Games**

```
SELECT * FROM students
WHERE likes = 'Games'
```

id	name	likes	age
1	Tom	Games	21
1 row, 4 columns			

<i>students</i>			
id	name	likes	age
1	Tom	Games	21
2	Emily	Tennis	22
3	Mike	Travel	22
3 rows, 4 columns			

SQL: WHERE

Get all names of students whose **age** is less than 22 or whose **id** is 2

```
SELECT name FROM students  
WHERE age < 22 OR id = 2
```

name
Tom
Emily
2 rows, 1 column

(selected because **age** is 21)

(selected because **id** is 2)

<i>students</i>			
id	name	likes	age
1	Tom	Games	21
2	Emily	Tennis	22
3	Mike	Travel	22
3 rows, 4 columns			

SQL: INSERT INTO

- INSERT INTO is used to add rows into a table
- VALUES is used for defining constant rows and columns, usually to be inserted

<i>students</i>			
id	name	likes	age
1	Tom	Games	21
2	Emily	Tennis	22
3	Mike	Travel	22
3 rows, 4 columns			

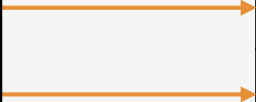
SQL: INSERT INTO

```
INSERT INTO students VALUES
```

```
(4, 'Alexander', 'Reading', 22),
```

```
(5, 'Jacky', 'Studying', 21)
```

This statement results in two extra rows being added to the table



<i>students</i>			
id	name	likes	age
1	Tom	Games	21
2	Emily	Tennis	22
3	Mike	Travel	22
4	Alexander	Reading	22
5	Jacky	Studying	21
5 rows, 4 columns			

SQL: UPDATE

- UPDATE is used to change the values of existing rows in a table
 - Followed by SET after the table name
- Usually combined with WHERE
- Syntax:

UPDATE [table]

SET [column] = [value]

WHERE [condition]

<i>students</i>			
id	name	likes	age
1	Tom	Games	21
2	Emily	Tennis	22
3	Mike	Travel	22
4	Alexander	Reading	22
5	Jacky	Studying	21
5 rows, 4 columns			

SQL: UPDATE

```
UPDATE students
```

```
SET age = 23
```

```
WHERE name = 'Alexander'
```

This statement results in this cell in the table being changed. If the **WHERE** clause was missing, every value in the **age** column would be set to 23.

<i>students</i>			
id	name	likes	age
1	Tom	Games	21
2	Emily	Tennis	22
3	Mike	Travel	22
4	Alexander	Reading	23
5	Jacky	Studying	21
5 rows, 4 columns			

SQL: DELETE

- DELETE FROM is used to delete rows from a table
- Usually combined with WHERE
- Syntax:
DELETE FROM
[table]
WHERE [condition]

<i>students</i>			
id	name	likes	age
1	Tom	Games	21
2	Emily	Tennis	22
3	Mike	Travel	22
4	Alexander	Reading	22
5	Jacky	Studying	21
5 rows, 4 columns			

SQL: DELETE

```
DELETE FROM students
WHERE age < 22
```

This statement results in two rows being deleted from the table

<i>students</i>			
id	name	likes	age
1	Tom	Games	21
2	Emily	Tennis	22
3	Mike	Travel	22
4	Alexander	Reading	22
5	Jacky	Studying	21
3 rows, 4 columns			

SQL: CREATE

- CREATE is used to create tables (and sometimes databases)

<i>students</i>			
id	name	likes	age
1	Tom	Games	21
2	Emily	Tennis	22
3	Mike	Travel	22
3 rows, 4 columns			

SQL: CREATE

```
CREATE TABLE cats (  
    id INT,  
    name VARCHAR(255),  
    likes VARCHAR(255),  
    age INT  
)
```

Note:
VARCHAR(255) is a
string type

This statement results in a
new table being created with
the given columns

<i>students</i>			
id	name	likes	age
1	Tom	Games	21
2	Emily	Tennis	22
3	Mike	Travel	22
3 rows, 4 columns			

<i>cats</i>			
id	name	likes	age
0 rows, 4 columns			

SQL: DROP

- DROP is used to delete tables (and sometimes databases)
- Syntax:

DROP TABLE [table]

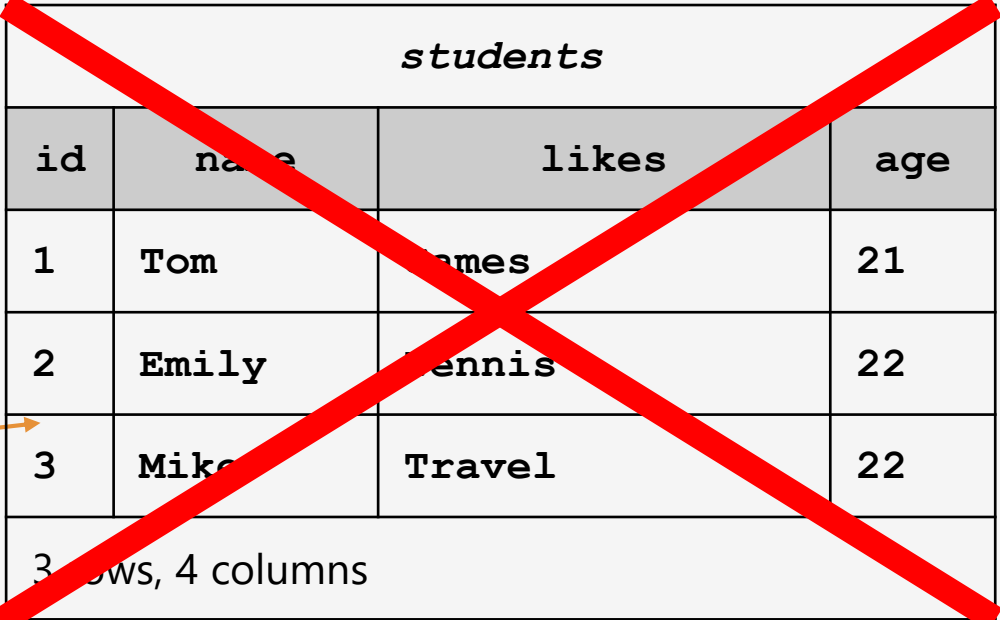
<i>students</i>			
id	name	likes	age
1	Tom	Games	21
2	Emily	Tennis	22
3	Mike	Travel	22
3 rows, 4 columns			

<i>cats</i>			
id	name	likes	age
0 rows, 4 columns			

SQL: DROP

```
DROP TABLE students
```

This statement results in the entire **students** table being deleted



<i>students</i>			
id	name	likes	age
1	Tom	Games	21
2	Emily	Tennis	22
3	Mike	Travel	22
3 rows, 4 columns			

<i>cats</i>			
id	name	likes	age
0 rows, 4 columns			

SQL: Syntax Characters

- `--` (two dashes) is used for single-line comments (like `#` in Python or `//` in C)
- Semicolons separate different statements
 - `UPDATE students SET age = 2 WHERE id = 4;`
`SELECT age FROM students WHERE id = 4`
 - Returns a single column with a row **2 only** if there already exists a row with ID 4
- SQL is complicated and it takes 2-3 lectures to cover completely, but you only need to know the basics for this class

A Go HTTP Handler

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=paperclips
```

Query

```
SELECT item, price FROM items WHERE name = 'paperclips'
```

A Go HTTP Handler

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='`

Invalid SQL executed by the
server, 500 Internal Server
Error

Query

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

A Go HTTP Handler

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=' OR '1' = '1`

This is essentially OR TRUE,
so returns every item!

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

Time flies: still remember this cartoon from lecture 1?



The 'NULL' license plate

- Security researcher Joseph Tartaro set NULL as his license plate value
- “any time a traffic cop forgot to fill in the license plate number on a citation, the fine automatically got sent to Joseph Tartaro”
- <https://www.wired.com/story/null-license-plate-landed-one-hacker-ticket-hell/>



Blind SQL Injection

- Not all SQL queries are used in a way that is visible to the user
 - Visible: Shopping carts, comment threads, list of accounts
 - Blind: Password verification, user account creation
 - Some SQL injection vulnerabilities only return a true/false as a way of determining whether your exploit worked!
- **Blind SQL injection:** SQL injection attacks where little to no feedback is provided
 - Attacks become more *annoying*, but vulnerabilities are still exploitable
 - Automated SQL injection detection and exploitation makes this less of an issue
 - Attackers will use automated tools


Blind SQL Injection Tools (Not in the exam)

- **sqlmap**: An automated tool to find and exploit SQL injection vulnerabilities on web servers
 - Supports pretty much all database systems
 - Supports blind SQL injection (even through timing side channels)
 - Supports “escaping” from the database server to run commands in the operating system itself
- **Takeaway**: “Harder” is harder only until someone makes a tool to automate the attack

; Introduction();--

sqlmap is an open source penetration testing tool that automates the process of detecting and exploiting SQL injection flaws and taking over of database servers. It comes with a powerful detection engine, many niche features for the ultimate penetration tester and a broad range of switches lasting from database fingerprinting, over data fetching from the database, to accessing the underlying file system and executing commands on the operating system via out-of-band connections.

```
$ python sqlmap.py -u "http://debiandev/sqlmap/mysql/get_int.php?id=1" --batch
```



```
{1.3.4.44#dev}
http://sqlmap.org
```

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user's responsibility to obey all applicable local, state and federal laws. Developers assume no liability and are not responsible for any misuse or damage caused by this program

[*] starting @ 10:44:53 /2019-04-30/

```
[10:44:54] [INFO] testing connection to the target URL
[10:44:54] [INFO] heuristics detected web page charset 'ascii'
[10:44:54] [INFO] checking if the target is protected by some kind of WAF/IPS
[10:44:54] [INFO] testing if the target URL content is stable
[10:44:55] [INFO] target URL content is stable
[10:44:55] [INFO] testing if GET parameter 'id' is dynamic
[10:44:55] [INFO] GET parameter 'id' appears to be dynamic
[10:44:55] [INFO] heuristic (basic) test shows that GET parameter 'id' might be injectable
(possible DBMS: 'MySQL')
```

SQL Injection Defenses

- Defense: **Input sanitization**
 - Option #1: Disallow special characters
 - Option #2: Escape special characters
 - 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!
 - Good as a defense-in-depth measure

```
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**

- Usually represented as a question mark (?) when writing SQL statements
- Idea: Instead of trying to escape characters before parsing, parse the SQL first, then insert the data
 - When the parser encounters the ?, it fixes it as a single node in the syntax tree
 - After parsing, only *then*, it inserts data
 - The untrusted input never has a chance to be parsed, only ever treated as data

```
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)  
    ...  
}
```

SQL Injection Defenses

- Biggest downside to prepared statements: Not part of the SQL standard!
 - Instead, SQL drivers rely on the actual SQL implementation (e.g. MySQL, PostgreSQL, etc.) to implement prepared statements
- Must rely on the API to correctly convert the prepared statement into implementation-specific protocol
 - Again: Consider human factors!