Attacks against Websites

Introduction

Will discuss attacks on websites and their prevention

- Authentication failure
- SQL injection
- Cross-site scripting
- Cross-site request forgery
- Code injection

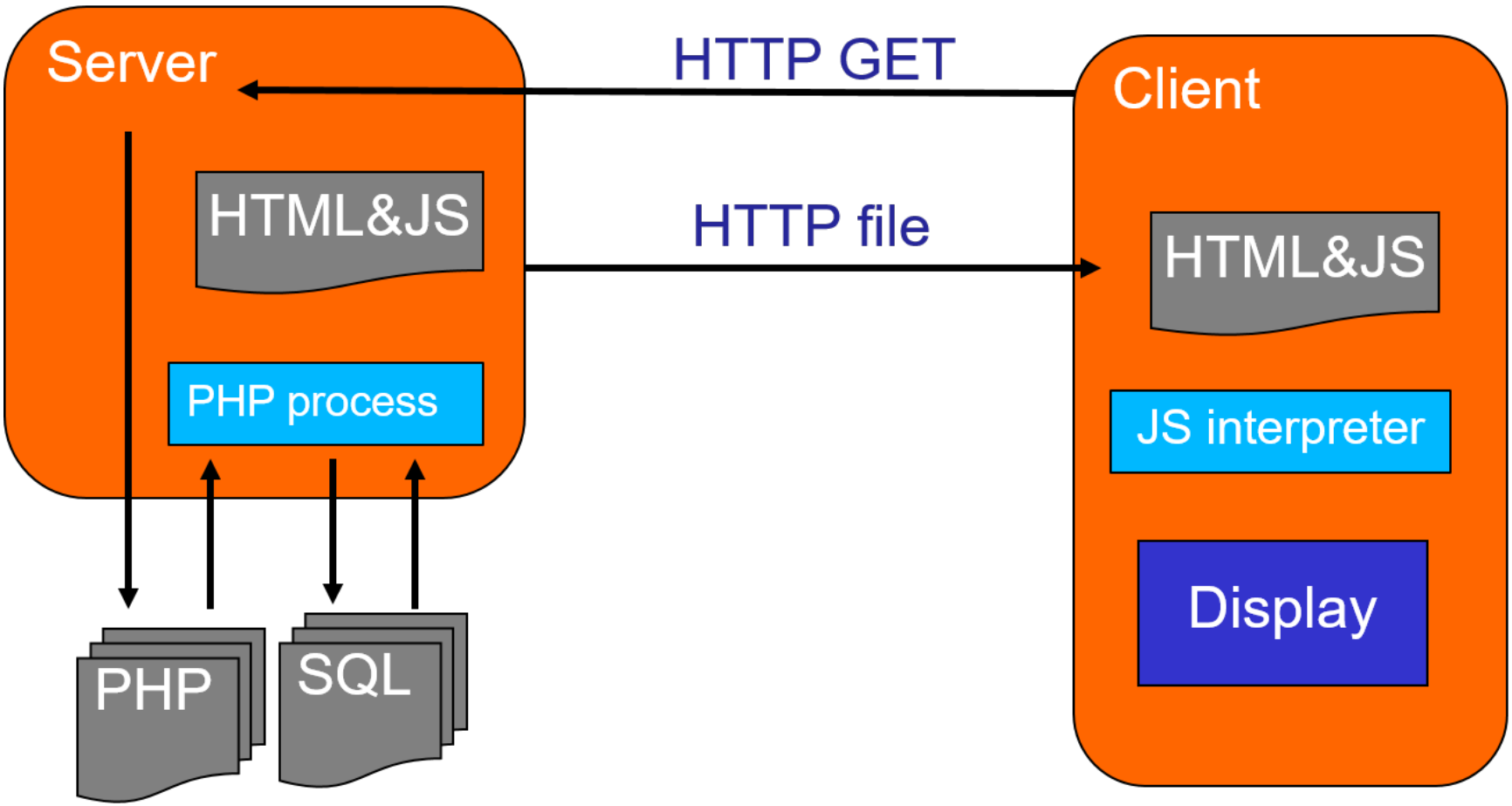
Two main sources of vulnerabilities:

- input validation
- application logic

Computer Misuse Act

- Unauthorised access to computing material.
  - 12 months in prison and/or a fine up to £5000
- Unauthorised access with intent to commit
  - 5 years in prison/fine
- Unauthorised acts with intent to impair operations of a computer.
  - Anti DoS addition in 2006.
- Making, supplying or obtaining articles for use in above offences
  - Dual use tools are OK.

Last Lecture

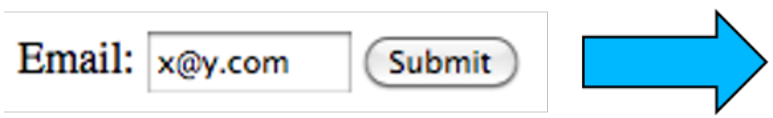


Typical Web Setup

HTTP website:

```
<form action="http://site.com/index.jsp" method="GET">
Email: <input type="text" name="email">
  <input type="submit" value="Submit">
</form>
```

User browser:



`http://site.com/index.jsp?email=x@y.com`

Typical Web Setup

```
http://site.com/index.jsp?email=x@y.com
```

PhP page reads and processes:

```
<?php
$email=$_GET["emailAddress"];
mysql_query("INSERT INTO emailsTable
                  VALUE(\‘".$email. "\’)";
?>
<b>Your e-mail has been added</b>
```

Authenticating users after log in

- IP address-based
  - NAT may cause several users to share the same IP
  - DHCP may cause same user to have different IPs
- Certificate-based
  - Who has a certificate and what is it, and who will sign it?
- Cookie-based
  - The most common

Cookies

- Cookies let server store a string on the client. Based on the server name.
  - HTTP response: Set-Cookie: adds a cookie
  - HTTP header: Cookie: gives a "cookie"
- This can be used to
  - Identify the user (cookie given out after login)
  - Store user name, preferences etc.
  - Track the user: time of last visit, etc.

Simple authentication scheme

- The Web Application:
  - Verifies the credentials, e.g., against database
  - Generates a cookie which is sent back to the user

    `Set-Cookie: auth=secret`
- When browser contacts the web site again, it will include the session authenticator

  `Cookie: auth=secret`

Fixed cookies

- Log in/out recorded on the server side.
  - Set cookie the first time browser connects,
  - Every page looks up cookie in database to get session state.
- PhP does this automatically: session cookies and `start_session()`

What can go wrong?

- **OWASP** = Open Web Application Security Project
- Public effort to improve web security:
  - Many useful documents.
  - Open public meetings and events.
- The "10 top" lists the current biggest web threats:
  `https://owasp.org/www-project-top-ten`

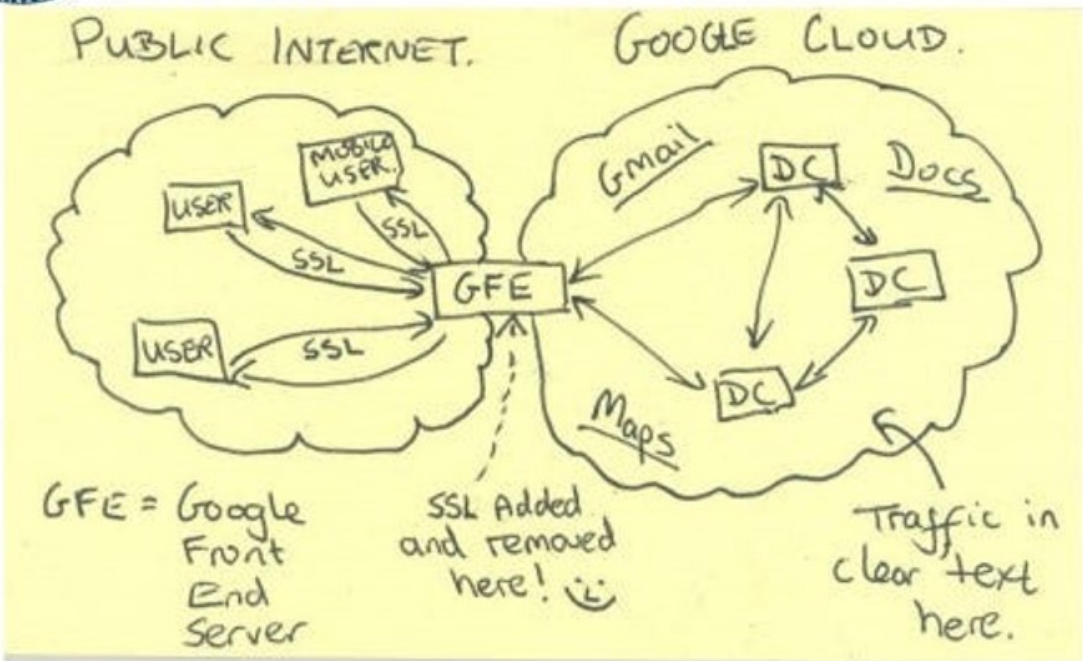
Eavesdropping

If the connection is not encrypted, it is possible to eavesdrop, by

- ISP,
- anyone on the route,
- anyone on your local network, e.g. using the same wi-fi.

TOP SECRET//SI//NOFORN

Current Efforts - Google



TOP SECRET//SI//NOFORN

https://www.businessinsider.com/leaked-nsa-slide-of-google-cloud-2013-10?r=US&IR=T

Steal the Cookie

- So the attacker does not need the username and password - just the cookie
- If the website uses https (TLS) it is secure
- But many websites dropped back to http after a secure login.

Countermeasures

- Use https (TLS) all the time.
- Set the secure flag: cookie is sent only over secure connections:

```
Cookie secureCookie =
    new Cookie("credential",c);
    secureCookie.setSecure(true);
```

Broken Authentication

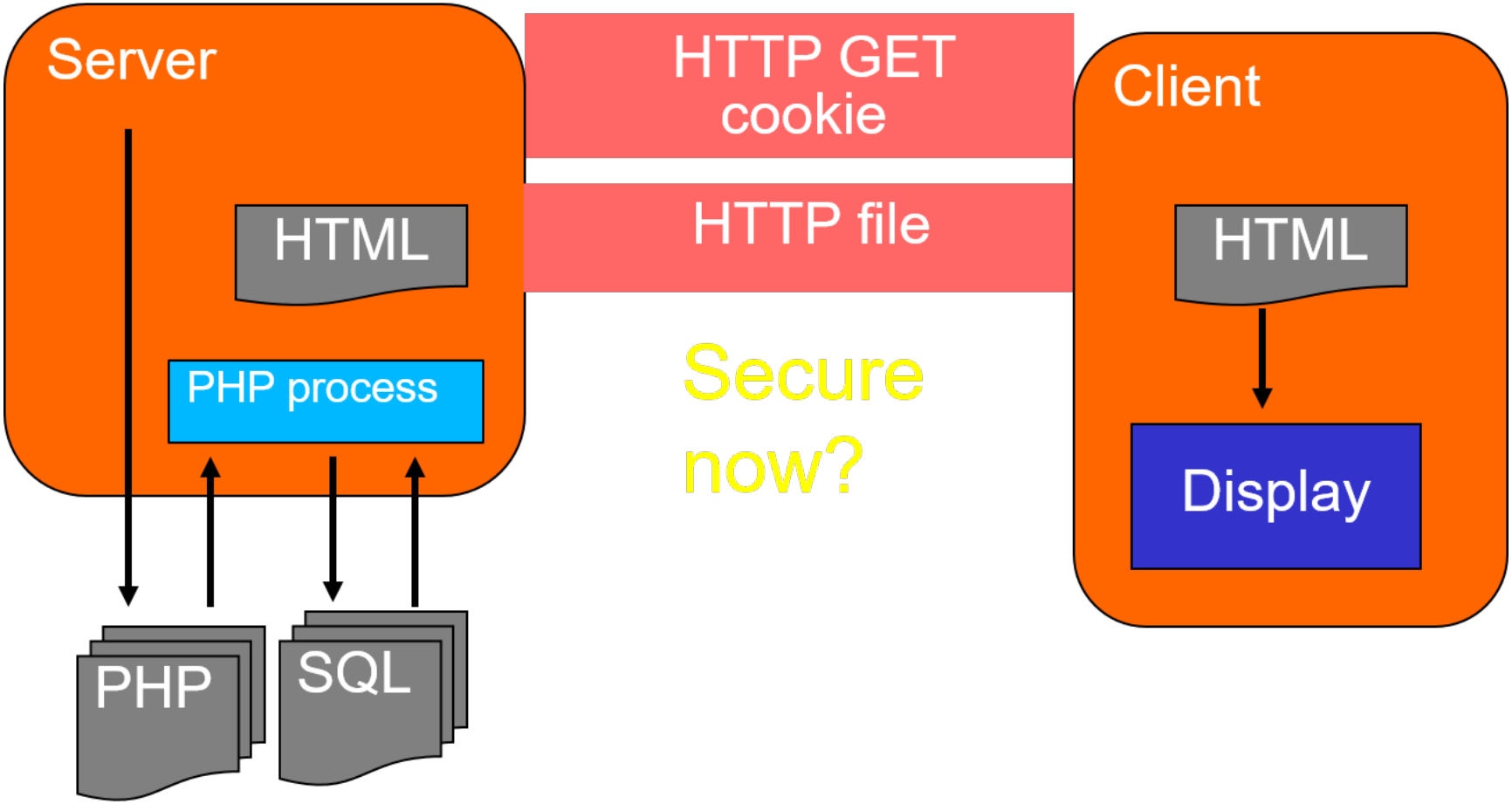
Many web developers implement their own log in systems. Often broken, e.g.

- No session time outs.
- Passwords not hashed

Sensitive Data Exposure

- Sensitive data transmitted in clear text
  (e.g. use of http instead of https)
- Sensitive data stored in clear text
  (e.g. passwords not hashed in database, credit card numbers not encrypted in database)
- Cookie stealing because https connection turns to http

A typical web set up



SQL Injection Attacks

`http://www.shop.com/page?do=buy&product=17453`
Web server looks up “17453” in a SQL DB using:

```
...
SELECT * FROM products WHERE (code='17453')
...
INSERT INTO sales VALUES (id, customer, 17453)
```

SQL Injection Attacks

```
http://www.eshop.co.uk?action=buy&product=X
```

$\Rightarrow$

```
SELECT * FROM products WHERE (code='X')
```

SQL Injection Attacks

Secret Item: `dh2*%Bgo`

$\Rightarrow$

```
SELECT * FROM items WHERE (item ='dh2*%Bgo')
```

If found, then item details are given.

SQL Injection Attacks

Secret Item:

```
' OR '1'='1' ) --
```

⇒

```
SELECT * FROM items WHERE (item='' OR '1'='1') -- ')
```

1 does equal 1! Therefore return details of all items (N.B. note the space after the comments).

SQL Attack Types

The best vulnerabilities will print the result of the SQL query.

- This lets you explore the whole database
- Information schema table can tell you the names of all other tables

Blind SQL attacks do not print the results:

- Lots of guesswork needed
- Run commands on database, e.g. add a password, delete tables
- Copy data (e.g. password) into a field you can read

Stopping SQL Attacks

Checking/cleaning the input, e.g. in PHP:
`mysqli_real_escape_string()`
e.g. `\\'OR \'1\'=\'1\'{` maps to
`\\\'OR \\\'1\\\'=\\\'1\\\'--`

However this is slightly problematic, see
`https://stackoverflow.com/questions/5741187/sql-injection-that-gets-around-mysql-real-escape-string`

Stopping SQL Attacks

Most languages these days have “prepared” statements, e.g. PHP and MySQLi:

```
// prepare and bind
$stmt = $conn->prepare
            ("INSERT INTO People (firstname, lastname)
                          VALUES (?, ?)");
$stmt->bind_param("ss", $firstname, $lastname);
// set parameters and execute
$firstname = "John";
$lastname = "Doe";
$stmt->execute();
```

https://www.w3schools.com/php/php_mysql_prepared_statements.asp

Not Just Websites

Not Just Websites


ZU 0666', 0, 0); DROP DATABASE TABLICE;

Not just SQL

Not just SQL injection, any command language can be injected, e.g. shell:

- `nc -l -p 9999 -e /bin/bash`
- Start a shell on port 9999
- `useradd tpc -p rEK1ecacw.7.c`
  - Add user tpc:npassword
- `rm -f -r /`
  - Ouch!