

*Get Exam Results
earlier via SMS
on 22 Dec 2020!*



For more
information



<https://myportal.nus.edu.sg/studentportal/academics/all/scheduleofresultsrelease.html>



Subscribe via NUS EduRec System
by **8 Dec 2020.**

Examination Matters

1. SoC has prepared **E-exam SOPs** for students, please refer to the guide: <https://mysoc.nus.edu.sg/academic/e-exam-sop-for-students/>

Please ensure that you read thru and set up what you need before the exam date.

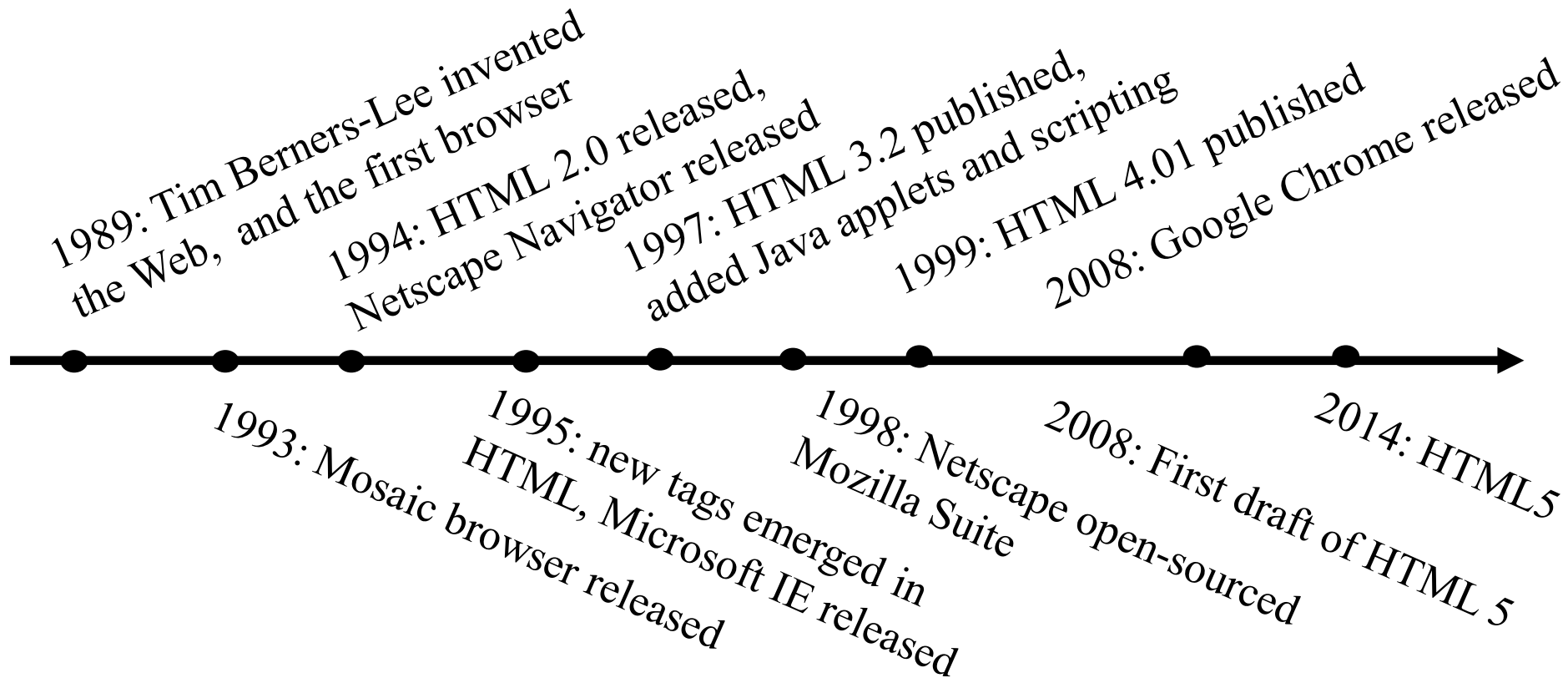
2. You should sit for the exam in insolation in a quiet environment with all the required hardware and software. If there's any extenuating circumstances that require you to **attempt your exams in campus**, please email your request to socexams@comp.nus.edu.sg by [**09 November 2020, Monday**](#) latest.

Lecture 9: Web Security

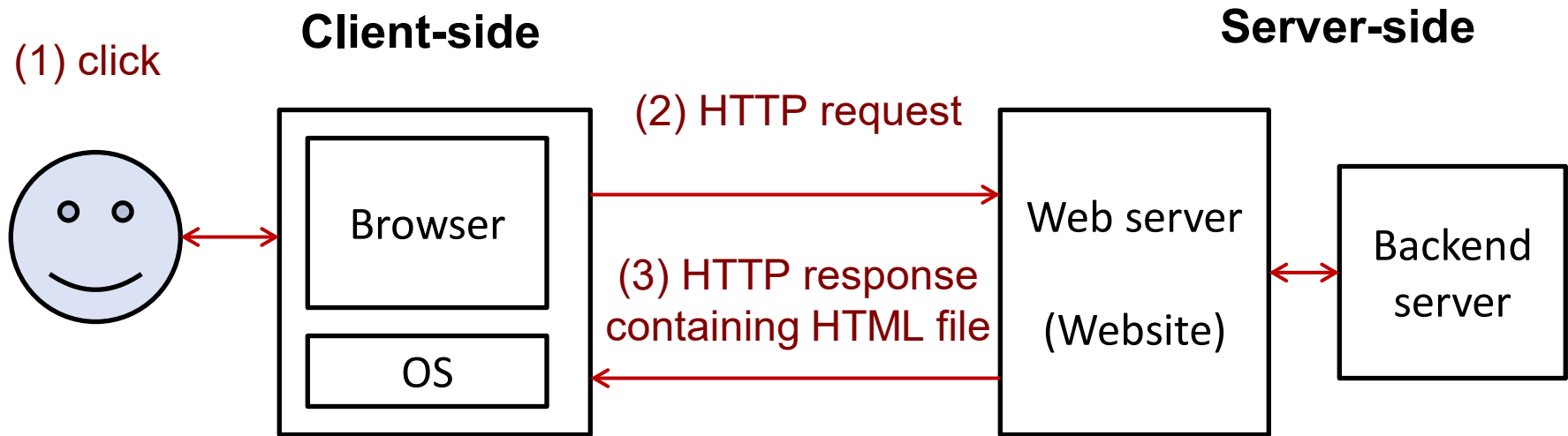
- 9.1 Background
- 9.2 Security issues and threat models
- 9.3 Vulnerabilities in the “secure” communication channel (TLS/SSL)
- 9.4 Misleading web user (UI/visual-based attacks)
- 9.5 Cookies and the same-origin policy
- 9.6 Cross-site scripting (XSS) attacks
- 9.7 Cross-site request forgery (CSRF) attacks

9.1 Background

Evolution of the Web



Overview of HTTP: A Web-Page Access Process



(4) render (including running **some scripts** in the HTML file)

1. User clicks on a **URL/“link”**, for example `luminus.nus.edu.sg/`
2. A **HTTP request** is sent to the server (with any in-scope **cookies** included)
3. Server constructs and include a **“HTML file”** inside its **HTTP response** to the browser (possibly with **set-cookie headers**)
4. The browser **renders the HTML file**, which describes the layout to be rendered and presented to the user, and any cookies are stored in the browser

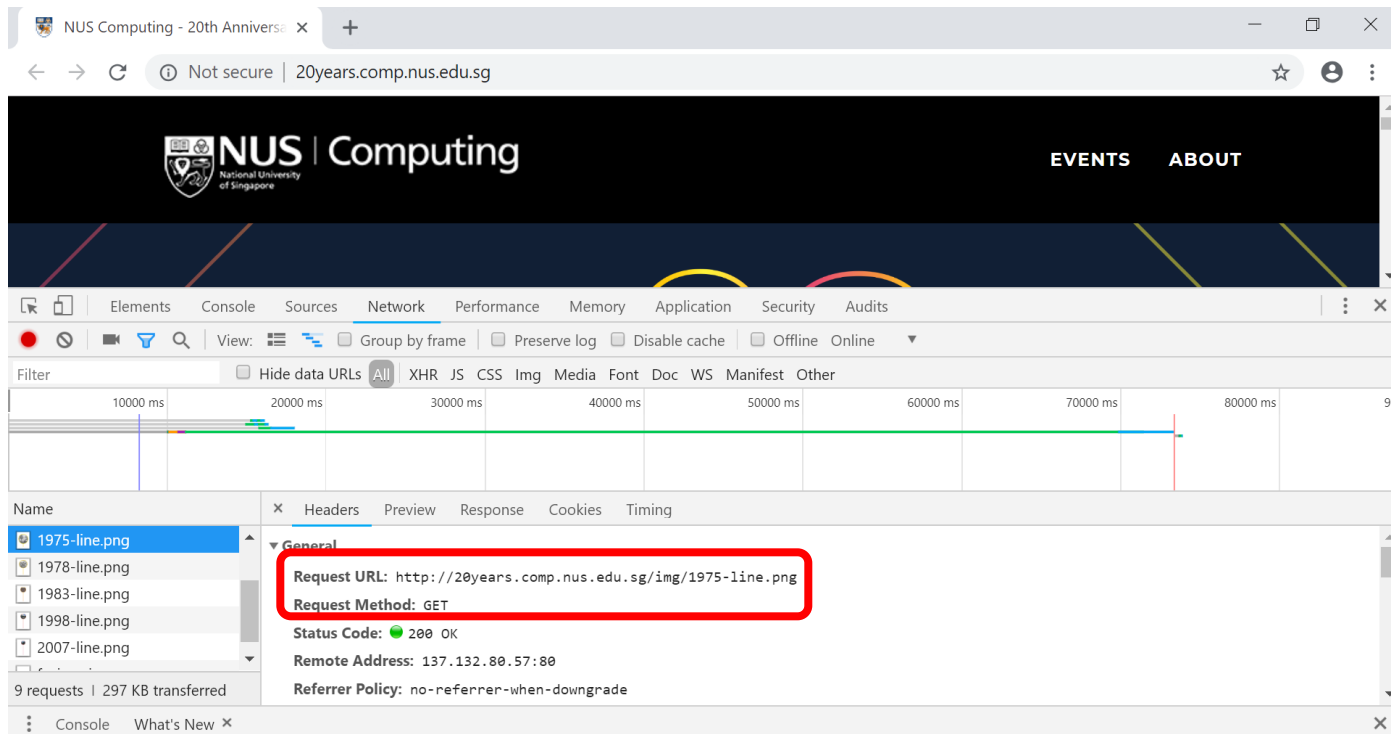
Notes: To **view (the raw form of)** the HTML file sent from the server to the browser:

(in Firefox) right-click a page, choose **“View page source”**; (in Chrome) View → Developer → **“View Source”**.

Note that there are many occurrences of the tag `<script>`, which marks the beginning of a **script**.

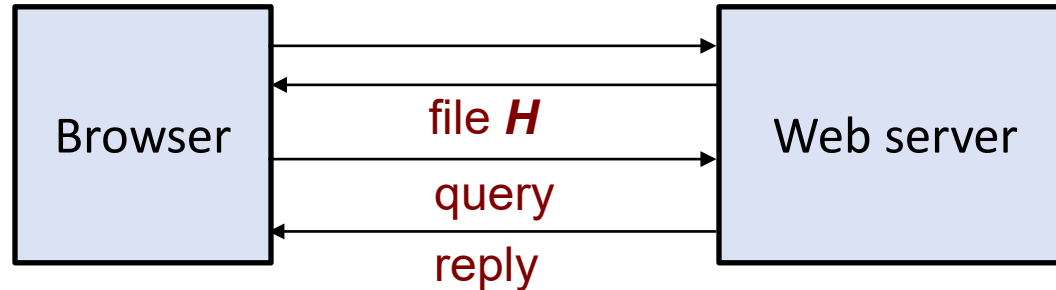
Sub-Resources of A Web Page

- A HTML page may contain ***sub-resources*** (e.g. images, multimedia files, CSS, scripts) including from **external/third-party** websites
- When parsing a page with sub-resources, browser also contacts the **respective server** for each sub-resource
- A **separate HTTP request** for **every single file** on a page: since each file requires its own HTTP request



A Closer Look into an Interactive Query-Page Example

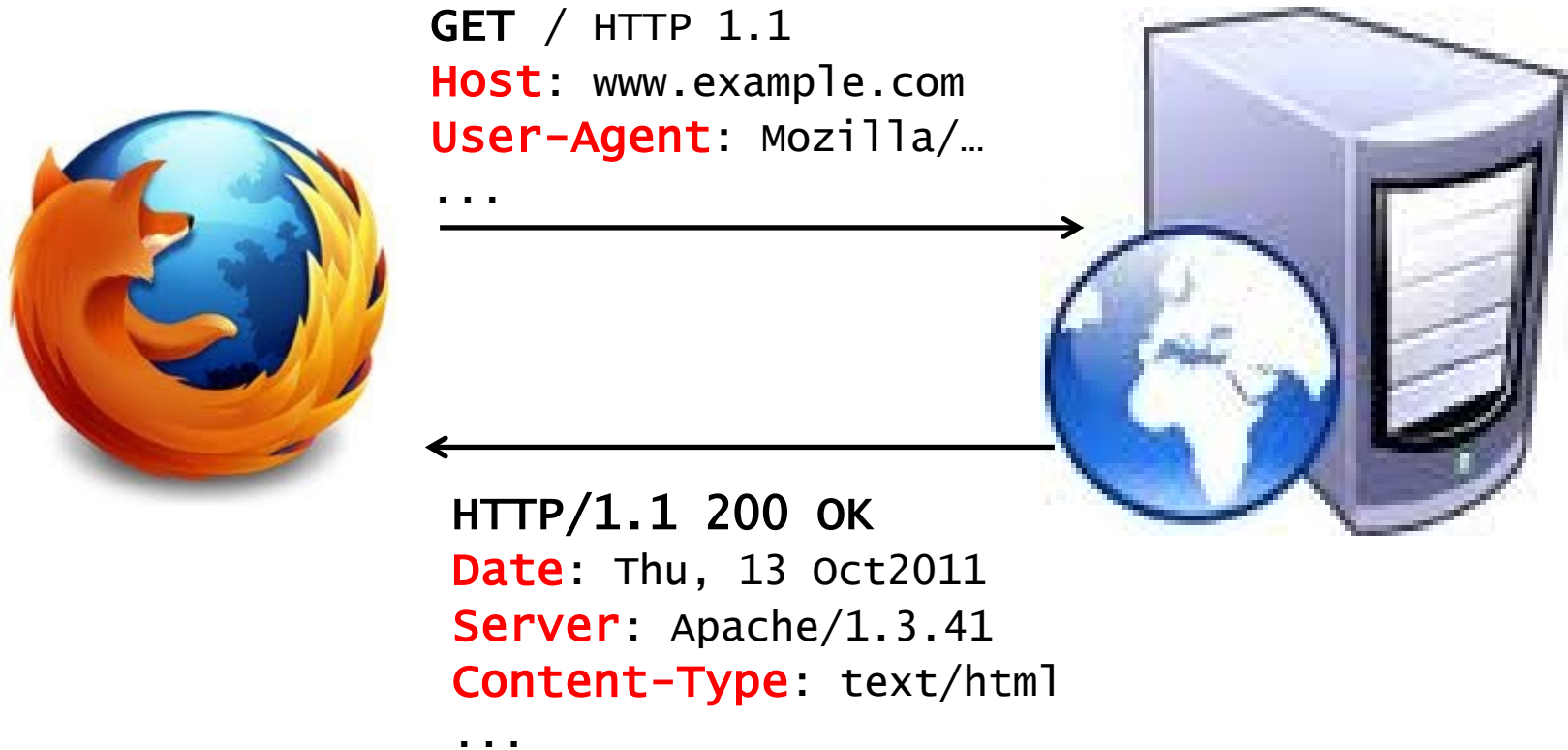
- 1) Browser visits **Google Search page** (`www.google.com.sg`).
A HTML file ***H*** is sent by the server to the browser.
The browser renders ***H***.



- 2) Browser user enters the search **keywords** “**CS2017 NUS**”
- 3) The browser, by running ***H***, constructs a **query**, for instance:
`https://www.google.com.sg/webhp?sourceid=chrome-instant&ion=1&espv=2&ie=UTF-8#q=CS2107+NUS`
- 4) Note that additional information is added as **URL parameters**.
These info is useful for the server.
The info could even be in the form of **script**.
- 5) The server constructs a **reply**.
(Notice that in some cases, the reply **contain substrings** sent in Step 3.)

HTTP Request and Response Messages

- Note that various **request** and **response header fields** are used (see https://en.wikipedia.org/wiki/List_of_HTTP_header_fields)



HTTP Request and Response Formats

- **HTTP Request** contains:

- **Request line**, e.g.: `GET /test.html HTTP/1.1`
- **Request headers**, such as:
`Accept: image/gif, image/jpeg, */*`
`Cookie: theme=light; sessionToken=abc123;`
- An empty/blank line
- An optional message body

- **HTTP Response** contains:

- **Status line** containing *status code & reason phrase*, e.g.:
`HTTP/1.1 200 OK`, `HTTP/1.0 404 Not Found`
- **Response headers**, such as:
`Content-Type: text/html`
`Content-Length: 35`
`Set-Cookie: theme=light`
`Set-Cookie: sessionToken=abc123; Expires=Wed, 09 Jun 2021 10:18:14 GMT`
- An empty/blank line
- An optional message body

Web Client and Server Components

- **Client-side** components:
 - Hypertext Markup Language (**HTML**): webpage **content**
 - Cascading Style Sheets (**CCS**): webpage **presentation**
 - **JavaScript**: webpage **behavior**, making pages “active” (interactive and responsive)
- **Server-side** components:
 - **Web server**: nowadays a **scripting language** is typically used as well, e.g. PHP, for “dynamic” pages
 - **Database server**: interaction between web server and database server via **SQL**

JavaScript for “Active” Pages

- Example of **JavaScript** in HTML:

```
<script type="text/javascript"> document.write('Hello World!');  
</script>
```

- What **can JavaScript do** in a browser?

- **Write a text** into an HTML page:

```
document.write("<h1>" + studentname + "</h1>")
```

- **Read and change HTML elements:**

```
var doc = document.childNodes[0];
```

- **React to events**, such as when a page has finished loading or when a user clicks on an HTML element:

```
<a href="someURL.html" onclick="alert('User just clicked me!')">
```

- **Validate user data**, e.g. form inputs

- **Access cookies!**

```
var doccookie = document.cookie;
```

- Interact with the server, e.g. using **AJAX** (*Asynchronous JavaScript And XML*)

PHP: A Popular Server-Side Scripting Language

- **PHP**: a widely used, free server scripting language for making **dynamic** web pages
- Sample PHP page:

```
<!DOCTYPE html>
```

```
<html>
```

```
<body>
```

```
<?php
```

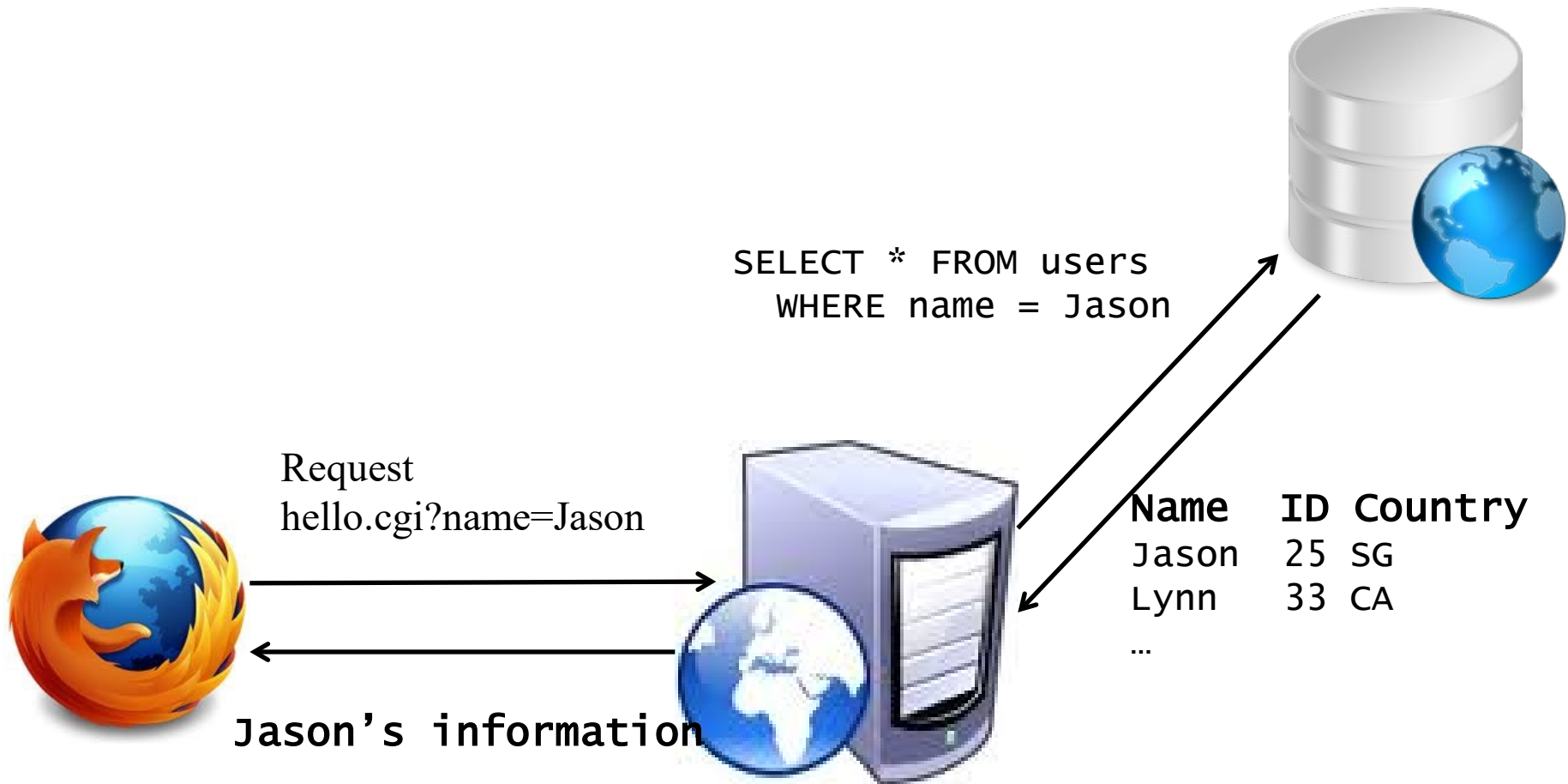
```
echo "My first PHP script!";
```

```
?>
```

```
</body>
```

```
</html>
```

Three-tiered Web Applications with Database Server



Shared Web Hosting

Optional

- A *shared web hosting* service: **many websites** reside on **1 web server**
 - **IP-based** virtual hosting:
 - Each virtual host has a **different IP address**
 - The web server uses **multiple** physical or virtual network interfaces **on the same physical interface**
 - **Name-based** virtual hosting:
 - Uses **multiple hostnames** on a single machine with a **single IP address**
 - **HTTP/1.1** it has `Host` HTTP header field
 - See: https://en.wikipedia.org/wiki/Shared_web_hosting_service
- **For HTTPS?**
 - Problem:
TLS handshake happens ***before*** the server sees any HTTP headers
 - Solution: ***Server Name Indication (SNI) TLS extension***, by which a client indicates which **hostname** it is attempting to connect to at the start of the **handshaking process**
 - See: https://en.wikipedia.org/wiki/Server_Name_Indication

9.2 Security Issues and Threat Models

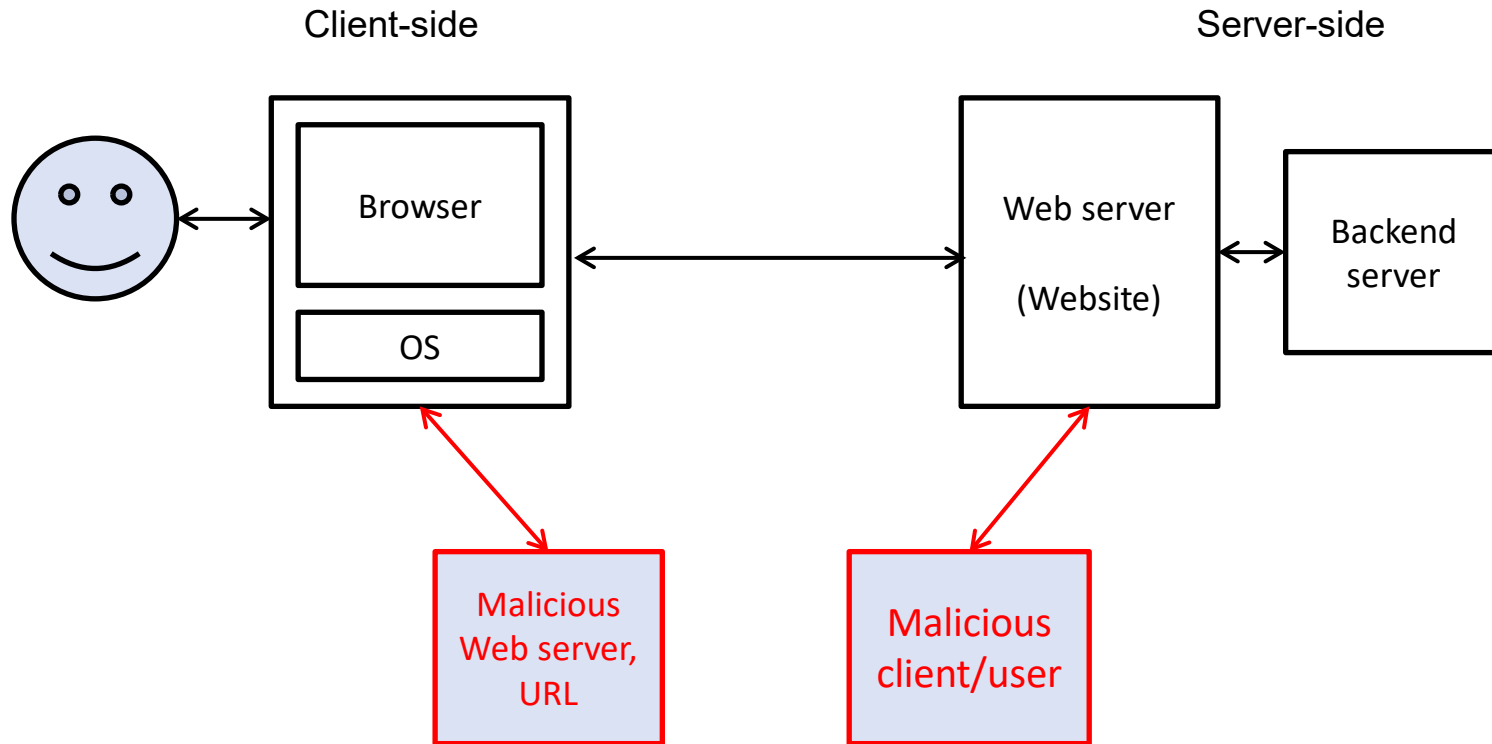
Complications of Web Security: Browser Operations

- Browsers run with the **same privileges** as the user: the browsers therefore can access the **user's files**
- At any particular instance, **multiple servers** (with different domain names) could provide the content: **access isolation** among sites is thus required
- Browsers support a **rich command set** and **controls** for content providers to render the **content**
- For enhanced functionality, many browsers support ***plugins, add-ons, extensions*** by third parties
(Note: the definitions and differences of plugins, add-ons, extensions may not be clear & depends on the developers)

Complications of Web Security: Browser Usage

- Browsers keep **user's info & secrets**:
e.g. stored in (permanent) cookies
- Users could **update content** in the server:
e.g. forum, social media sites,
where names are to be displayed
- More and more users' **sensitive data** is stored
in the web/cloud
- For PC, the browser is becoming the **main/super application**:
→ in some sense, **the browser “is” the OS**

Threat Model 1: Attackers as Another *End Systems*

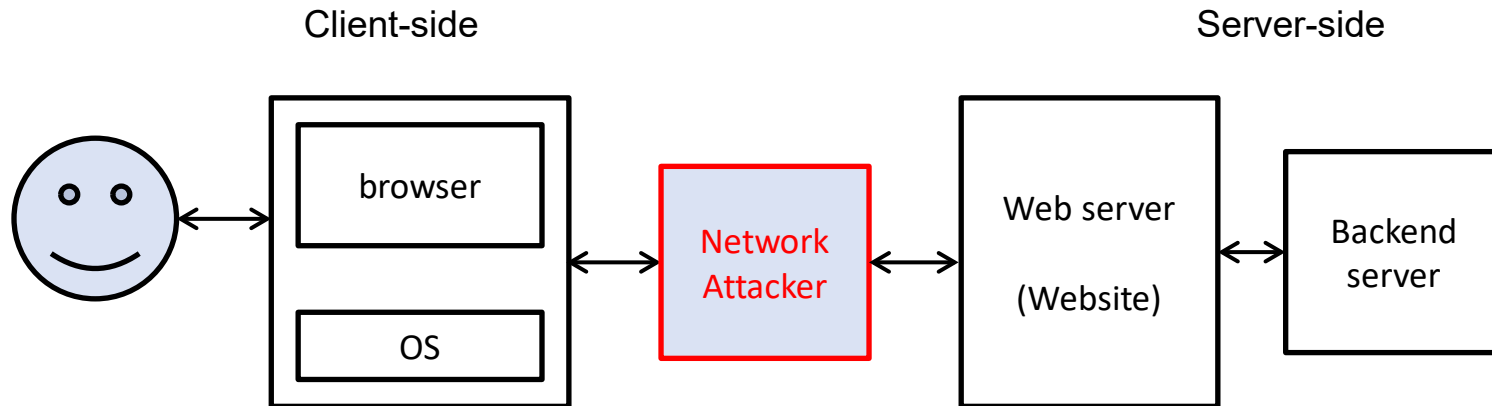


- In this scenario, the attackers are just another *end systems*
- Examples:
 - A **malicious web server** that lures the victim to visit it
 - A **malicious web client** who has access to the targeted server

Attacker Types in Threat Model 1

- **1A: *Forum poster*:**
 - The weakest attacker type
 - A **user** of an existing web app
 - **Doesn't** register domains or host application content
- **1B: *Web attacker*:**
 - Owns a **valid domain & web server** with an SSL certificate
 - Can **entice** a victim to visit his site, e.g.:
 - Via “[Click Here to Get a Free iPad](#)” link
 - Via an advertisement (no clicks needed)
 - **Can *not*** intercept/read traffic for other sites
 - The most **commonly-assumed** attacker type. *Why?*

Threat Model 2: Attackers as *Network Attackers*



- Here, the attacker has **access** to exchanged network packets (at the IP layer)
- Example: a **malicious café-owner** who provides the free WiFi services in our previous examples

Attacker Types in Threat Model 2

- **2A: *Passive network attacker:***
 - Eve who can **passively eavesdrop** on network traffic, but cannot manipulate or spoof traffic
 - Can **additionally act** as a web attacker
- **2B: *Active network attacker:***
 - Mallory who can launch **active attacks** on a network, e.g. **MiTM**
 - Can **additionally act** as a web attacker
 - The **most powerful** threat model
 - Yet, it is **not** generally considered to be capable of **presenting valid certificates** for HTTPS sites that are not under his control:
Why not?

Web Attacks and Classification

- Yet, it can be **difficult** to clearly classify web attacks
- Many attacks uses a **combination** of other attacks
- This lecture describe some **web attacks** and relevant common **protection mechanisms**

9.3 Attacks on the “Secure” Communication Channel (TLS/SSL)

- **HTTPS** protocol:
 - HTTPS = HTTP + TLS/SSL
 - Netscape SSL 2.0 [1993] ... TLS 1.3 [2018]
- Provisions a ***secure channel***, which establishes between 2 programs a **data channel** that has **confidentiality, integrity** and **authenticity**, against a computationally-bounded “network attacker”
- How does HTTPS work?
 - Ciphers negotiation
 - Authenticated key exchange (AKE)
 - Symmetric key encryption and MAC

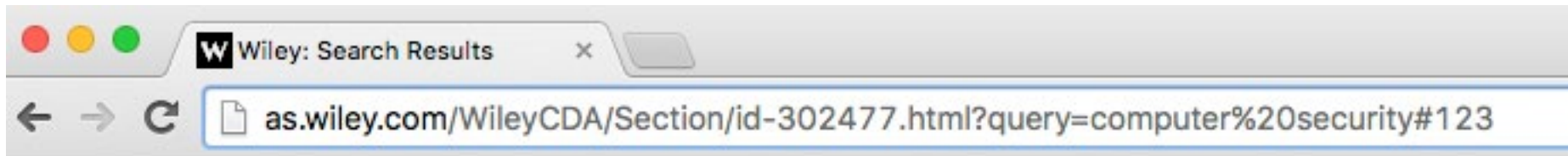
Attacks on a Secure Channel by a MiTM

- Two **pre-conditions** of a MiTM attack:
 - The attacker is a MiTM in between the browser and web server
 - The attacker is able to sniff & spoof packets at the TCP/IP layers
- Note that if the connection is HTTPS, such MiTM is **unable** to compromise both confidentiality & authenticity, *unless*:
 - Web user accepts a **forged certificate** or a **rouge CA**
- Yet, this might not be the case when **there exist vulnerabilities** in the **protocol's design** or **implementation**
- We have already covered some **HTTPS attacks**: re-negotiation attack, FREAK attack, Heartbleed (attack on protocol design or implementation)
- Other well-known attacks (*not required in this module*): BEAST attacks (attack on cryptography)

9.4 Misleading Web User (UI/Visual-based Attacks)

URL (Uniform Resource Locator)

- A **URL** consists of a few **components**
(see https://en.wikipedia.org/wiki/Uniform_Resource_Locator):
 1. Scheme
 2. Authority (a.k.a. the hostname)
 3. Path
 4. Query
 5. Fragment
- Example:
<http://www.wiley.com/WileyCDA/Section/id-302477.html?query=computer%20security#123>



WILEY

Question: Why the URL is typically displayed with **two levels** of intensity?

URL: Misleading Delimiter

- Suppose there is **no** clear **visual distinction** between the hostname and path of a URL
- The supposed “**delimiter**” that separates hostname & the path can actually be a character **in the hostname or path**

www.wiley.com/WileyCDA/Section/id-302477.html?query=computer%20security



Hostname

path

- Example: a malicious website's hostname that contains the targeted hostname followed by a character resembling the **delimiter** “/” (e.g. www.wiley.com.lwiley.in/Section/id-302477.html)
- Another example: nuslogin.789greeting.co.uk (from phishing email)
- The displayed different intensities could help user **spot** the attack

Address Bar Spoofing

- **Address bar** is an important browser's component to protect: the ***only* indicator** of what URL the page is actually rendering
- *What if the address bar can be “modified” by a webpage?*
- An attacker could trick the user to **visit** a malicious URL **X**, while making the user **wrongly believe** that the URL is **Y**
- A poorly-designed browser may allow attacker to achieve the attack

Address Bar Spoofing: Example

- In the early design of some browsers, a webpage could render objects/pop-ups in **an arbitrary location**
- This allows a malicious page to ***overlay* a spoofed address bar** on top of **the actual address bar**
- Current versions of popular browsers have mechanisms to prevent this issue
- Yet, a recent attack, e.g. [Android Browser All Versions - Address Bar Spoofing Vulnerability - CVE-2015-3830](https://www.rafaybaloch.com/2017/06/android-browser-all-versions-address.html)
(<https://www.rafaybaloch.com/2017/06/android-browser-all-versions-address.html>)



9.5 Cookies and the Same-Origin Policy

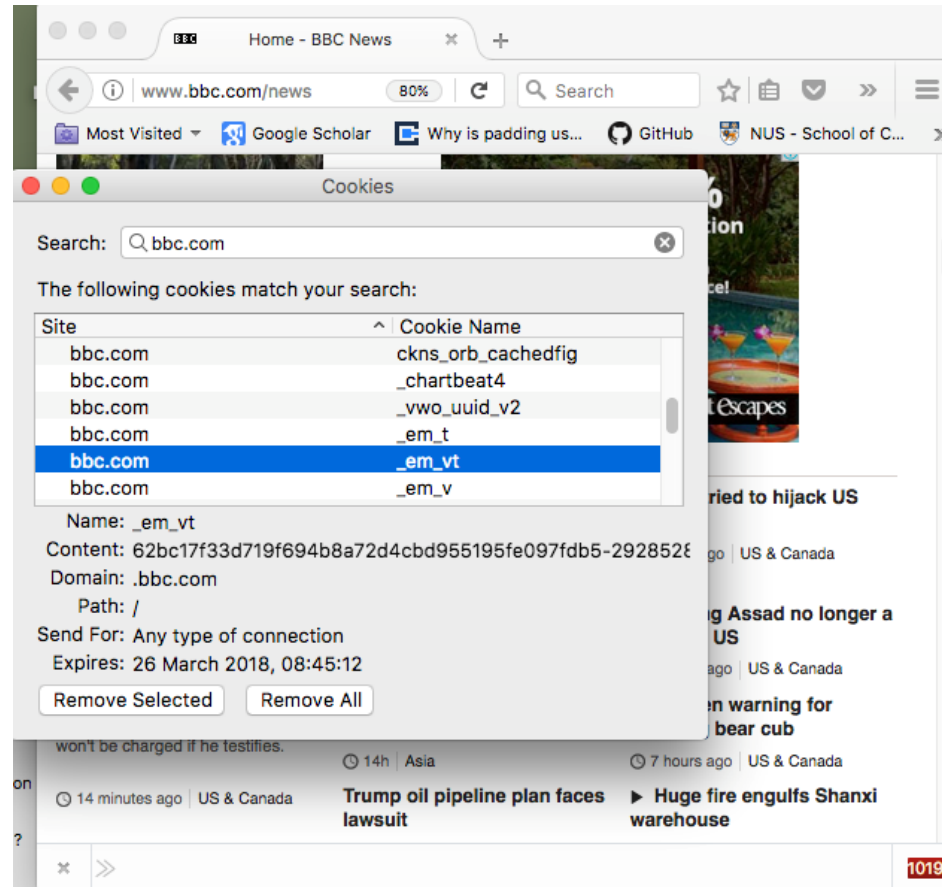
Remark:

The same-origin policy (SOP) is **not** an attack, but is a protection mechanism to protect cookies.

Cookies

- **A HTTP cookie:**
 - Is a piece of textual data that is set **by a web server**, and sent in an HTTP response's "**Set-Cookie**" header field
 - Consists of a **name-value pair**
 - Can be used to indicate a **user preference**, shopping cart content, a server-based ***session identifier***, etc.
- Cookies are stored on the user's browser while the user is browsing
- Whenever the user revisits the website (i.e. submit another HTTP request), the browser **automatically** sends all "**in-scope**" cookies to the server in its HTTP request's "**Cookie**" header field
- (Optional) **In-scope** cookies: cookies set by the "***same-cookie***" **origin**, i.e. the "origin" of the cookies
(Note: the scheme/protocol checking *may* be optional, *see later*)

Viewing Cookies



In Firefox:

- Right-click → View Page Info → Security → View Cookies; or
- Tools → Web developer → Developer toolbar → Storage

In **Chrome**: `chrome://settings/content/cookies`

Cookies: Usage

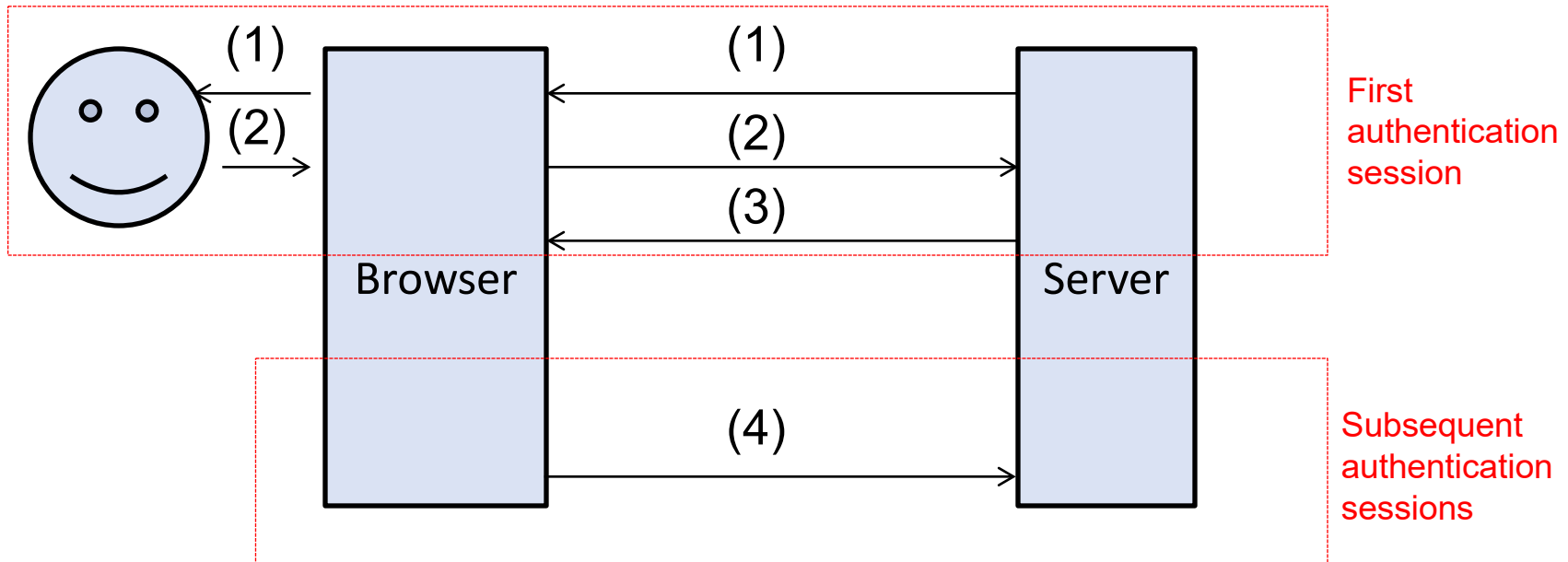


- There are **a few types** of cookie, such as:
 - ***Session cookie***: deleted after the browsing session ends
 - ***Persistent cookie***: expires at a specific date or after a specific length of time
 - ***Secure cookie***: can only be transmitted over **HTTPS**
- See https://en.wikipedia.org/wiki/HTTP_cookie#Terminology
- Note: the checking on scheme in the “same origin” for cookies is **optional**, except for **secure cookies** which strictly require HTTPS
 - ***Question***: How can cookie be really useful to web apps?
 - Since HTTP is **stateless**, there is a need to keep track of a **web session**
 - Cookie is commonly used to **set & indicate** a ***session ID*** in “*token-based*” web authentication scheme

Token-based Authentication and Cookie

- To ease a web user's tedious task of repeated logins, many websites use “***token-based***” ***authentication***:
 1. After a user U is **authenticated** (for e.g. **password verified**), the server sends a value t , known as the **token**, to U
 2. In subsequent connections, whoever presents the **correct token** is thus **accepted** as the authentic user U
- **Remarks:**
 - A token can identify a ***session***: hence, the token is also called ***session ID (SID)***
 - A token typically has an **expiry date** for a **limited** session lifetime
 - In web applications, a token is often stored in a **cookie**
 - Using **cookie** is a better approach than attaching the SID as a **URL-encoded parameter** or as a **form field**, but it has its own issues too

Token-based Authentication: Diagram



- (1) **Authentication challenge** (e.g. asking for password)
- (2) **Authentication response** that involves the user
- (3) Server sends a token t , and Browser keeps the token t
- (4) Browser presents the token t with HTTP request, and Server verifies the token t

Note: We assume that the **communication channel is secure**: it is done over HTTPS (with server being authenticated) and the HTTPS is free from vulnerabilities.

Choices of Token and Storage Requirement

- A token t needs to be **random** and sufficiently **long**
- Suppose token t is a **randomly chosen number**, then the server has to keep a **table** storing all issued tokens
- To **avoid** storing the table, one could use:
 - **(Insecure)** The cookie is some meaningful information concatenated with a **predictable sequence number**
E.g. $t = \text{"alicetan:16/04/2015:128829"}$
 - **(Secure)** The cookie consists of **two parts**:
a randomly chosen value or meaningful information like the expiry date; and concatenated with the **Message Authentication Code (MAC)** computed using the server's secret key
E.g. $t = \text{"alicetan:16/04/2015:adc8213kjd891067ad9993a"}$

Choices of Token and Storage Requirement

- For both methods, when the server finds out that the token is **not** in the correct format or MAC, the server **rejects** the token
- The **first** method is **insecure**:
an attacker, who knows how the token is generated (e.g. by observing its own token), can **forge it**
- This illustrates the weakness of “security by obscurity”:
a wrong assumption that attackers don’t know the format
- The **second** method is **secure**:
it relies on the security of MAC

Scripts & Same-Origin Policy (SOP): Browser Access Control

- A **script** running in the browser could access **cookies** (and also webpage's objects)
- Important question: **which *scripts* can access what *cookies*?**
- Due to security concern, browser employs the following ***Same-Origin Policy* access control**
- The **script** in a webpage *A* (identified by its URL) can access **cookies** stored by another webpage *B* (identified by its URL), only if both *A* and *B* have the **same *origin***
- ***Origin*** is defined as the combination of:
protocol, hostname, and port number
- The above is simple and thus seemingly safe
- However, there are a number of possible **complications**

Same-Origin Policy (SOP): Some Complications

- Example of **origin determination rules**:
URLs with the same origin as <http://www.example.com>
(from http://en.wikipedia.org/wiki/Same-origin_policy)

Compared URL	Outcome	Reason
http://www.example.com/dir/page2.html	Success	Same protocol, host and port
http://www.example.com/dir2/other.html	Success	Same protocol, host and port
http://username:password@www.example.com/dir2/other.html	Success	Same protocol, host and port
http://www.example.com:81/dir/other.html	Failure	Same protocol and host but different port
https://www.example.com/dir/other.html	Failure	Different protocol
http://en.example.com/dir/other.html	Failure	Different host
http://example.com/dir/other.html	Failure	Different host (exact match required)
http://v2.www.example.com/dir/other.html	Failure	Different host (exact match required)
http://www.example.com:80/dir/other.html	Depends	Port explicit. Depends on implementation in browser.

- **Limitation:** there are many *exceptions*, and also exceptions of exceptions: very confusing and thus prone to errors
- An example: unlike other browsers, **Microsoft IE** does *not* include the port in the calculation of the origin, using the **Security Zone** in its place
(See <https://blogs.msdn.microsoft.com/ieinternals/2009/08/28/same-origin-policy-part-1-no-peeking/>.)

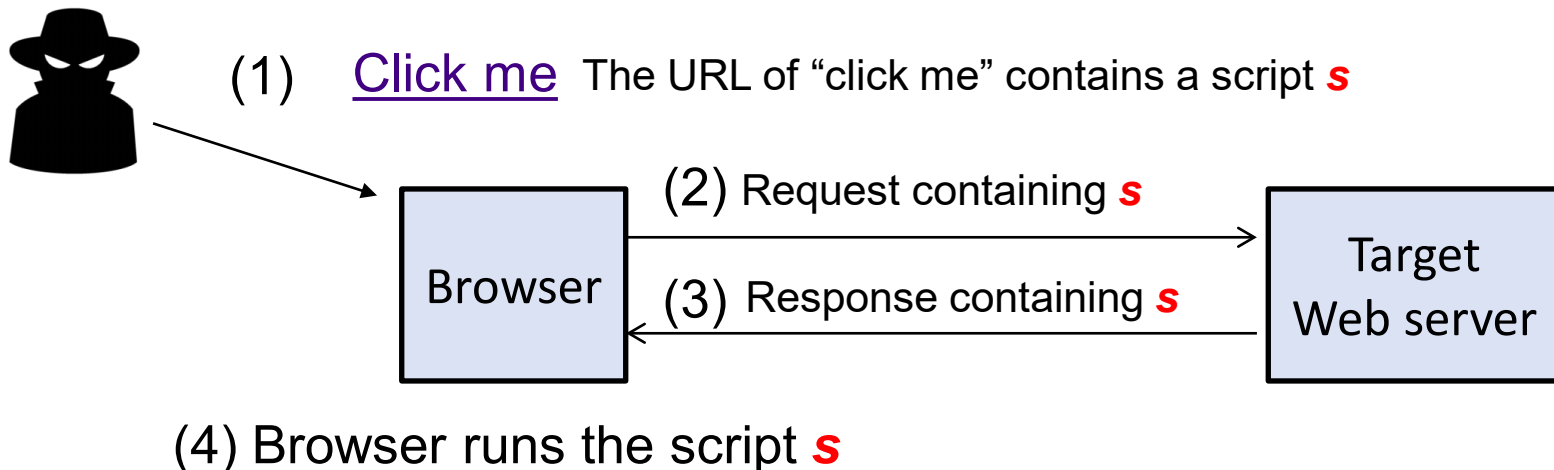
9.6 Cross Site Scripting (XSS) Attacks

Reflected (Non-Persistent) XSS Attack: Background

- In many websites, the client can enter a string **s** in the browser, which is to be sent to the server
- The server then responses with a HTML that also **contains s**, which is then rendered and displayed by the client's browser
- Examples:
 - Enter a wrong address:
http://www.comp.nus.edu.sg/nonsense_test
 - Search for a book in library:
<http://nus.preview.summon.serialssolutions.com/#!/search?q=hehehe>
- Important question: what if the string **s** contains a **script**?
 - Example: [http://www.comp.nus.edu.sg/<script>alert\("hehehe!"\);</script>](http://www.comp.nus.edu.sg/<script>alert("hehehe!");</script>)
- Note that the attack above **won't** work if the server performs **HTML (entity) encoding**: replaces the special character **<** with **<**;

Reflected (Non-Persistent) XSS Attack: The Attack

1. The attacker tricks a user to **click** on a URL, which contains the target website and a malicious script **s** (For example, the link could be sent via email with “click me”, or a link in a malicious website.)
2. The request is **sent to the server**
3. The server constructs a **response HTML**: the server doesn't check the request carefully, and its response **contains s**
4. The browser renders the HTML page, and **runs** the script **s**



Why is This an Attack?

- A script can be a **benign** script
- However, a **malicious** script could (among others):
 1. Deface the original webpage
 2. Steal cookies
- Recall the same-origin policy:
Because the script comes from the **target web server**, it **can access cookies** previously sent by the web server
- This is an example of **privilege escalation**:
a malicious script coming from the attacker has the privilege of the web server and read the latter's cookies
- The attack above exploits the **client's trust of the server**:
the browser believes that the injected script is from the server

Stored (Persistent) XSS

- The script **s** is **stored** in the target web server
- For instance, it is stored in a **forum page**:
the attacker is a **malicious forum poster**
- Another example: **Samy XSS worms** on Myspace.com,
where Samy became a friend of 1M users in less than 20 hours!
(See [https://en.wikipedia.org/wiki/Samy_\(computer_worm\)](https://en.wikipedia.org/wiki/Samy_(computer_worm)))
- **More dangerous** than reflected XSS attacks:
 - The malicious script is **rendered automatically**,
without the need to lure target victims to a 3rd-party website
 - The victim-to-script ratio is **many:1**

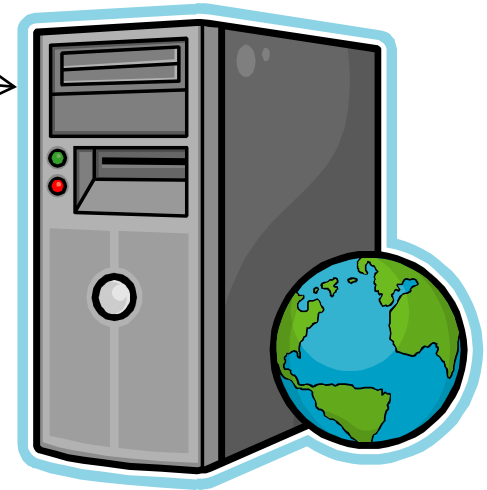
Stored (Persistent) XSS



`http://victim.com/store.cgi?<XSSCODE>`

`http://victim.com/view.cgi`

Page with
`<XSSCODE>`



XSS Attacks: Summary

- What is **XSS** in short?
 - "A type of **injection attack** on web apps, whereby a **forum poster** or **web attacker** attacks **another web user** by causing the latter run a (malicious) script from the former in the **execution context** of a page from **an involved web server**, thus subverting the Same Origin Policy"
- The attack works by exploiting **the victim's trust of the involved server**:
 - In reflected XSS:
the web server that **returns** a page reflecting the injected script
 - In persistent XSS:
the web server that **stores** a page containing the injected script

Defenses

- Most defense rely on mechanisms carried out in the **server-side**:
 - The server **filters and then removes/sanitizes** any malicious script in a **HTTP request** while constructing its response page
 - The server **filters and then removes/sanitizes** any malicious script in a **user's post** before it is saved into the forum database
- Some example techniques:
 - Script filtering
 - Noscript region:
do not allow JavaScript to appear in certain region of a webpage.
- However, this defense is **not** a fool-proof method
- To additionally detect **reflected XSS attack**, some browsers employ a **client-side** detection mechanism: e.g. **XSS auditor**

9.7 Cross Site Request Forgery (CSRF) Attacks

CSRF Attack: *With* the Victim Clicking on a URL

A.k.a. “sea surf”, *session riding*

An attack **example** (*with* the victim clicking on a URL):

- Suppose a client Alice is **already** authenticated by a target website S , say www.bank.com, and S accepts an **authentication-token cookie**
- The attacker Bob tricks Alice to **click** on a **URL** of S e.g. in a phishing email, which maliciously **requests** for a service, say transferring \$1,000 to Bob:
www.bank.com/transfer?account=Alice&amount=1000&to=Bob
- Alice’s cookie will also be **automatically sent** to S , indicating that the request comes from already-authenticated Alice
- Hence, the transaction will be **carried out** by S

(For more details, see https://en.wikipedia.org/wiki/Cross-site_request_forgery.)

CSRF Attack: *Without* the Victim Clicking on a URL

A **web attacker** can perform a CSRF attack *without* any victim user's UI actions

An attack **example** (*without* the victim clicking a URL):

- Again, suppose Alice is **already** authenticated by a target website *S* (www.bank.com), and *S* accepts an authentication-token cookie
- **Alice visits the attacker's site**, whose page contains the following:
<IMG SRC="www.bank.com/transfer?account=Alice&amount=1000&to=Bob"
WIDTH="1" HEIGHT="1" BORDER="0">
- Alice's browser issues **another HTTP request** to obtain the image
- Alice's cookie will also be **automatically sent** to *S*
- Hence, the transaction will be **carried out** by *S*

CSRF Attacks

- What is the **CSRF** in short?
"A type of **authorization attack** on web apps, whereby a **web attacker** attacks **a web user** by issuing a **forged request** to **a vulnerable web server** 'on behalf' of the victim user"
- The attack disrupts the **integrity** of the target user's session
- This is, in a way, the reverse of XSS:
it exploits **the server's trust of the client**
(the server believes that the issued request is from the client)

Defenses

- CSRF is **relatively easier** to prevent compared to XSS
- The **SID** or **authentication-token cookie** that is automatically sent by the browser is *insufficient*:
→ the server must issue and require an **extra (dynamic) information**, i.e. “*anti-CSRF token*”
- In the considered attack scenario, the server **includes** its anti-CSRF token in the **money-transfer request** page:
 - In the request **URL**:
`www.bank.com/transfer?account=Alice&amount=1000&to=Bob&CSRFtoken=xxk34n890ad7casdf897e324`
 - As a HTTP request **header field**
 - As a **hidden form field**

Other Web Attacks and Terminologies

- ***Drive-by download***
- ***Web bug*** (a.k.a. *web beacon, tracking bug, page tag*)
- ***Clickjacking*** (User Interface redress attack)
See <https://www.owasp.org/index.php/Clickjacking>
- ***Click fraud***

Question: Could **merely visiting** a malicious website (for e.g. by clicking on a phishing email) make a web user become a subject to web attacks?