Information Security

Chapter 04: Web security

Nguyễn Đăng Quang

Contents

THE BASICS

CROSS-SITE REQUEST FORGERY (CSRF)

CROSS-SITE SCRIPTING ATTACK

SQL INJECTION ATTACK

Web Security Basics

01

Web Browser

02

Web Server

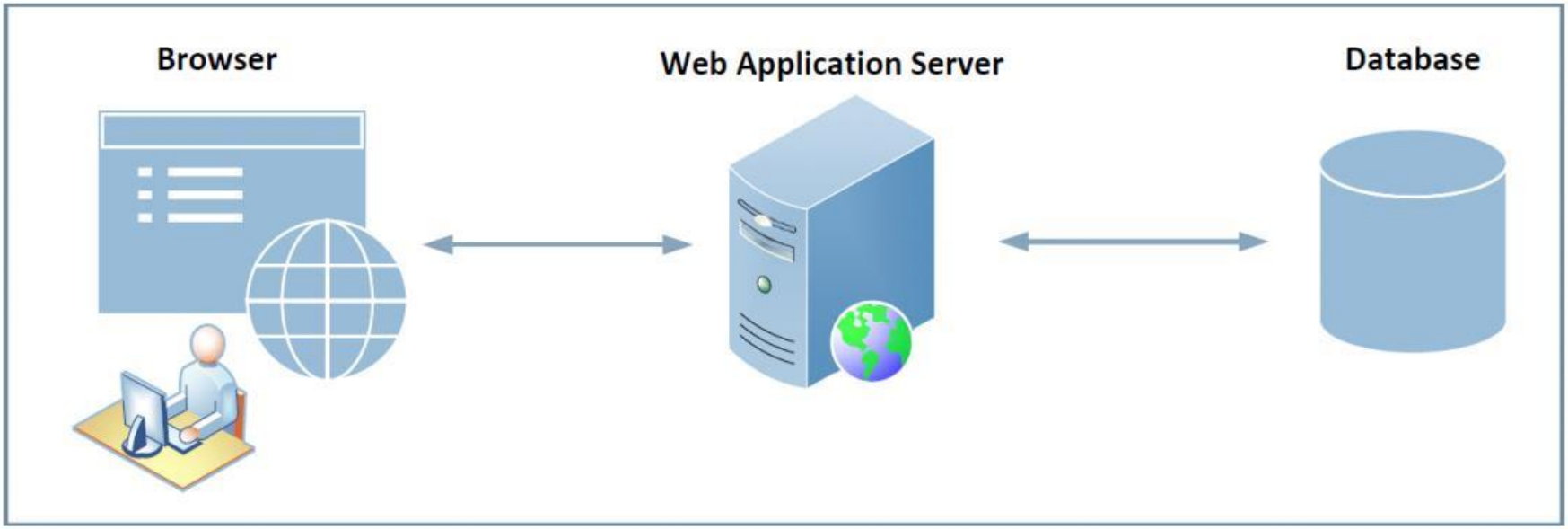
03

Cookies and Sessions

04

Ajax, WebSocket, Same-origin Policy

Web Architecture



Web Browser (Client side)

HTML

(Content)

```html
<html>
    <head>
        <title>My title</title>
    </head>
    <body>
        <h1>My header</h1>
        <a href="www.example.com">Link</a>
    </body>
</html>
```

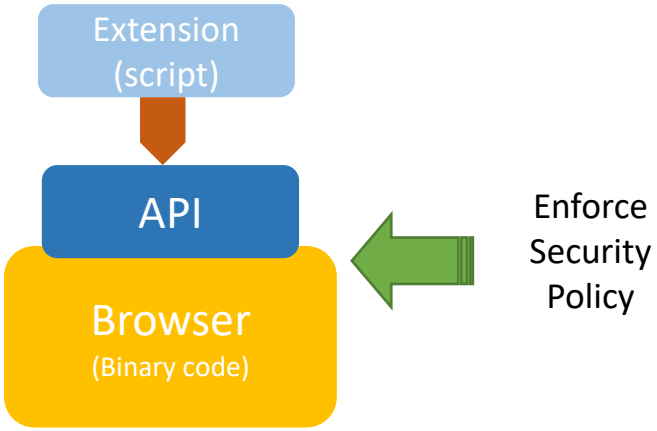
CSS

(Presentation)

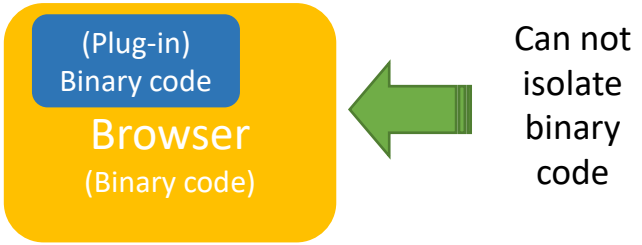
```html
<html>
    <head>
        <style>
            body {background-color: powderblue;}
            h1   {color: blue;}
            p {color: red;}
        </style>
    </head>
    <body>
        <h1>Hello world</h1>
        <p>This is a paragraph</p>
    </body>
</html>
```

Web Browser (Client side)

Extensions



Plug-ins



Web Browser - Dynamic Contents

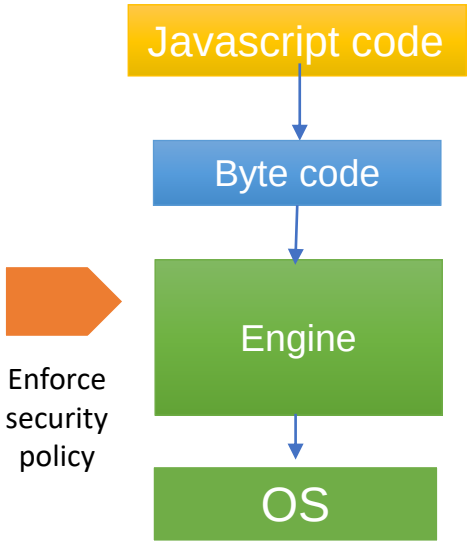
- Adobe flash
- Microsoft Silverlight
- ActiveX
- Java Applet
- **JavaScript**

Javascript & Sandbox

- **Javascript**: code inside a web page that brings dynamic contents to a page.
- The untrusted code may be malicious and that can be harmful to our computer.
- The code needs to be in a **sandbox** being enforced security policy to prevent those code from tampering the computer.

Javascript Engines

- V8 from Google (Chrome & MS Edge)
- SpiderMonkey (Mozilla Firefox)
- JavascriptCore from Apple (Safari)



JavaScript engine

- V8 from Google (Chrome and MS Edge)
- SpiderMonkey from Mozilla (Firefox)
- JavaScriptCore from Apple (Safari)

JavaScript inside a page

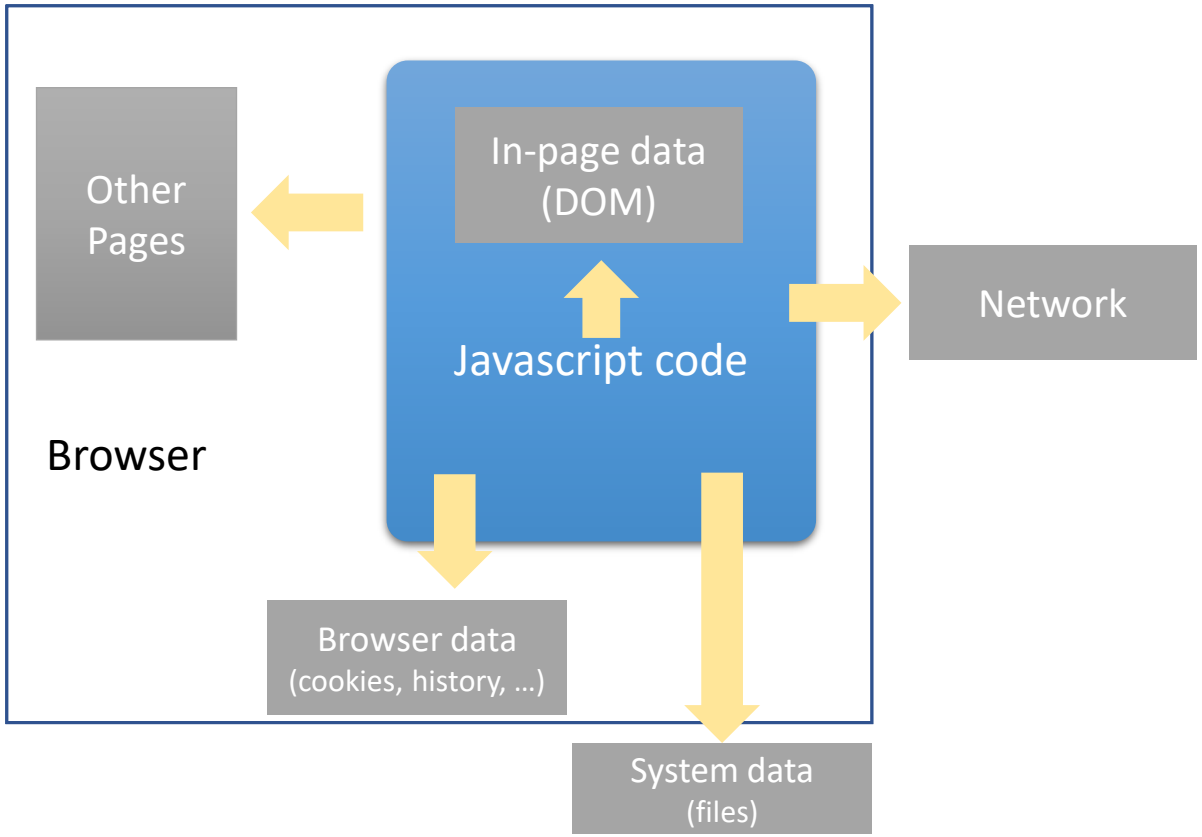
```
<script>
    ... code ...
</script>

<script src="myScript.js"></script>
<script src="https://www.example.com/myScript.js"></script>

<button type="button" onclick="myfunction()">Click here</button>
```

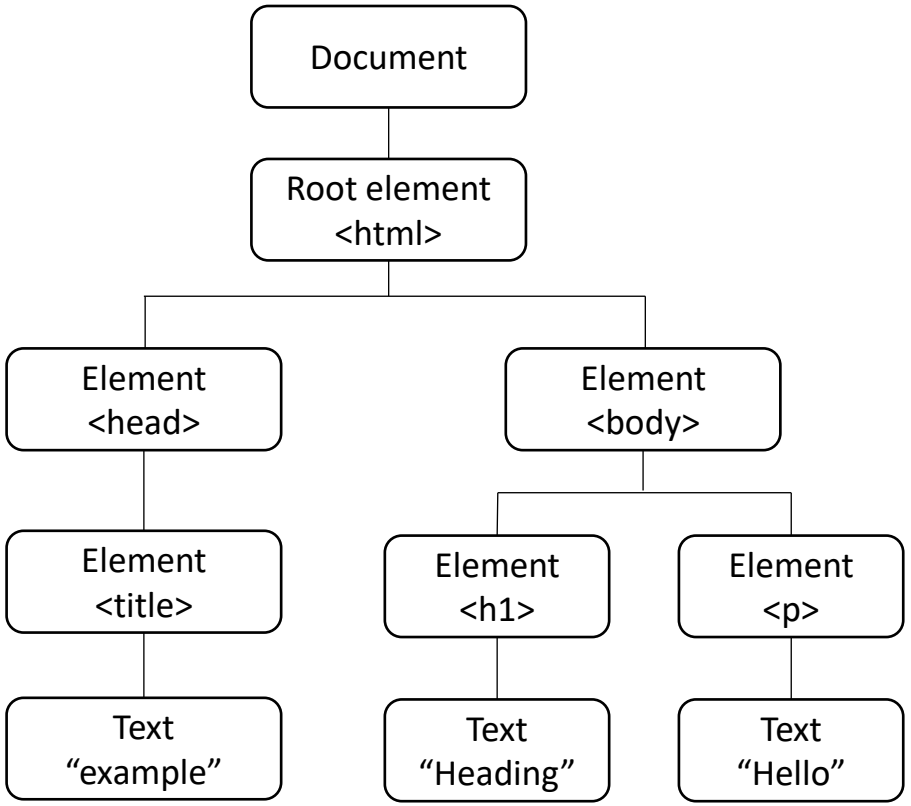
Sandboxing JavaScript

Javascript code can access:



Access Page Contents

HTML DOM (Document Object Model)




```
document.getElementById(‘demo’ ).innerHTML = ‘Hello world’;
```

Access Page Contents

HTML Objects

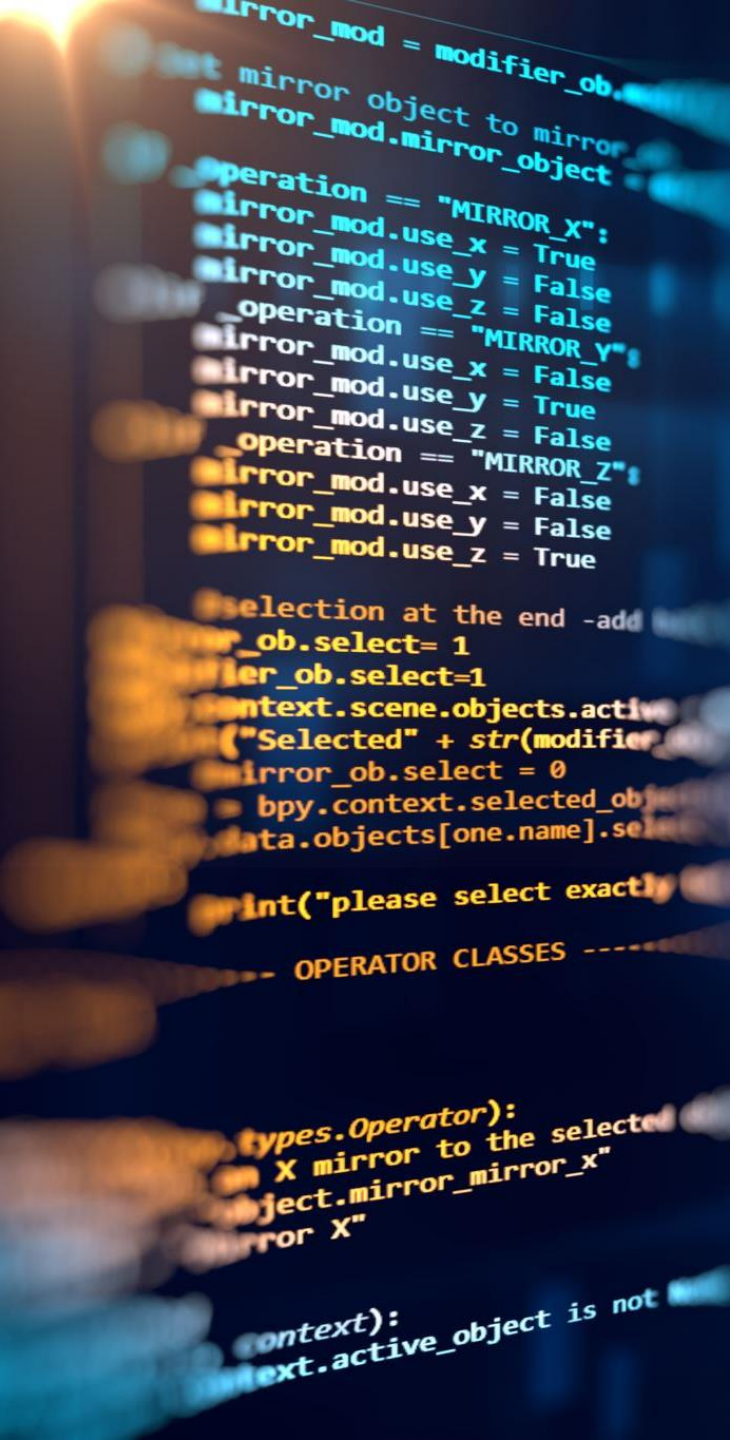
- window.history
- window.images
- document.forms
- document.URL

Browser Objects

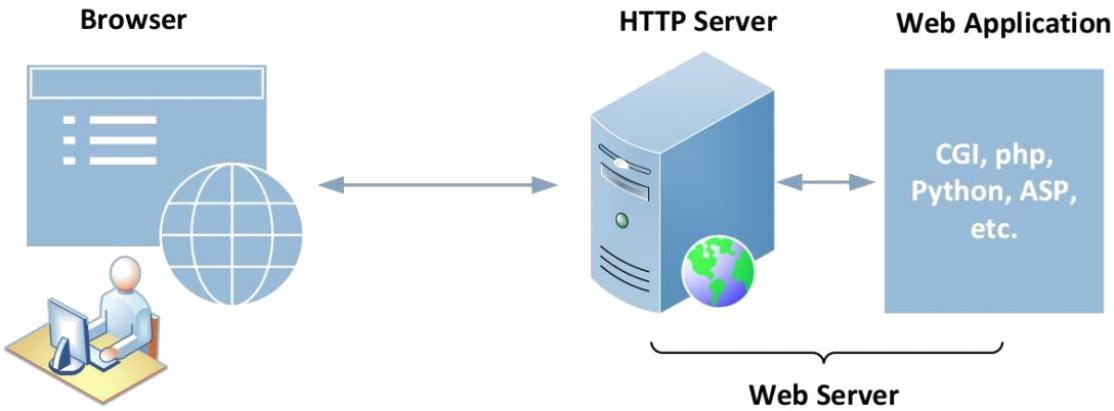
- window.history
  - back() and forward() APIs
- window.navigator
  - Navigator.geolocation.getCurrentPosition()

```
<input type="file" id="file-selector">
<script>
    function showfile(){
        var files = document.getElementById('file-selector').files;
        if (!files.length){
            alert('Please select a file!');
            return;
        }
        var reader = new FileReader();
        reader.onloadend = function(evt){
            if (evt.target.readyState == FileReader.DONE) {
                content = evt.target.result;
                document.getElementById('demo').textContent = content;
            }
        };
        reader.readAsBinaryString(files[0]);
    }
</script>
```

Accessing Files



Web server (server side)



Interacting with Server

HTTP REQUEST

AJAX REQUEST

WEB SOCKET

HTTP Request & Response

```
GET /hello.htm HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
Host: www.tutorialspoint.com
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive
```

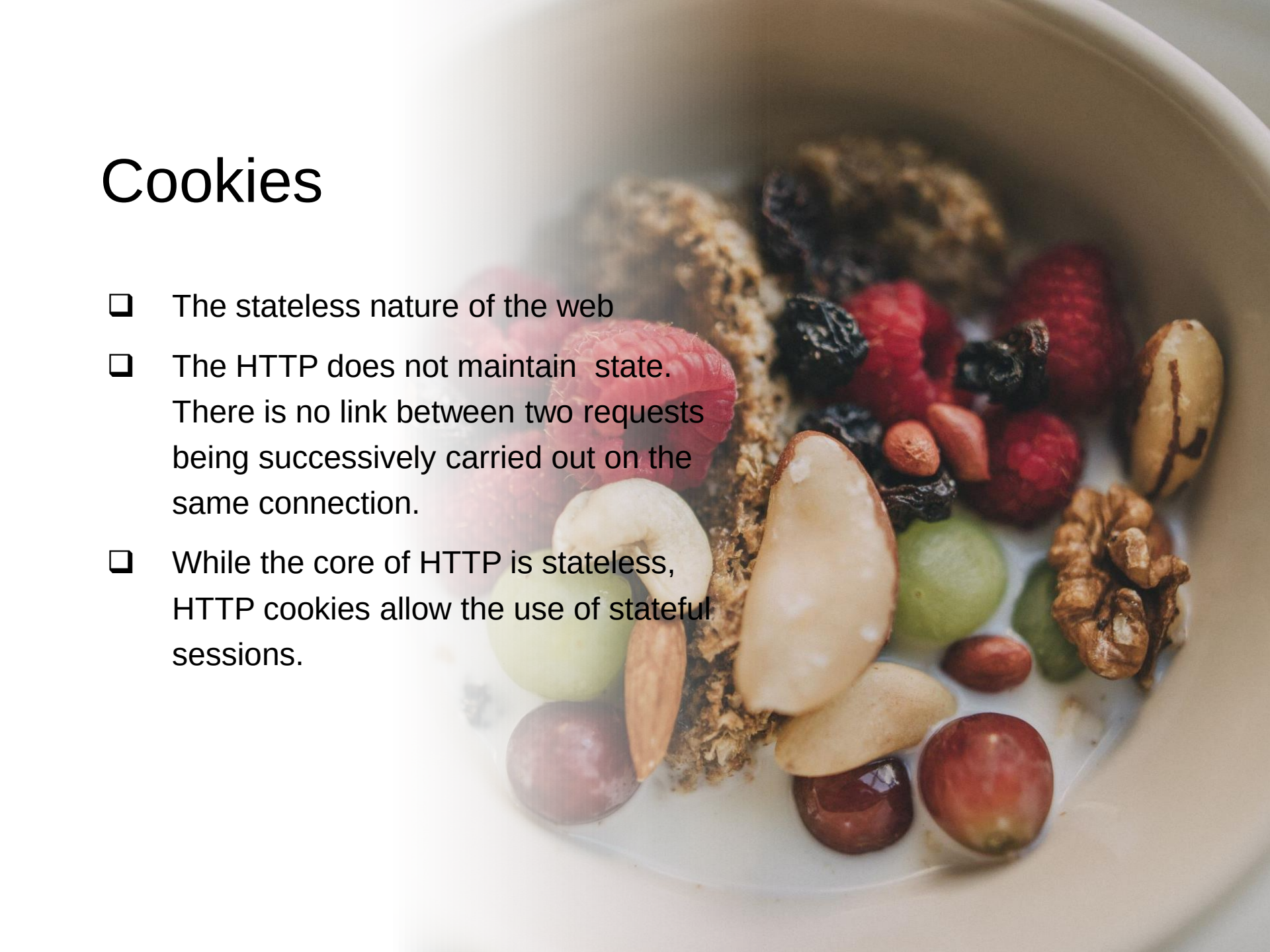
```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache/2.2.14 (Win32)
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
Content-Length: 88
Content-Type: text/html
Connection: Closed
```

GET vs POST

```
POST /post_form.php HTTP/1.1
Host: www.example.com
Cookie: SID=xsdfgergbghedvrbeadv
Content-Length: 19
foo=hello&bar=world
```

```
GET /post_form.php?foo=hello&bar=world HTTP/1.1
Host: www.example.com
Cookie: SID=xsdfgergbghedvrbeadv
```

Cookies



- The stateless nature of the web
- The HTTP does not maintain state. There is no link between two requests being successively carried out on the same connection.
- While the core of HTTP is stateless, HTTP cookies allow the use of stateful sessions.

```php
<?php
  setcookie('cookieA', 'aaaaaa');
  setcookie('cookieB', 'bbbbbb', time() + 3600);

  echo "<h2>Cookies are set</h2>"
?>
```

Setting Cookies

- Attaching cookies in http request

```
http://www.bank32.com/index.html
Host: www.bank32.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; ...
Accept: text/html,application/xhtml+xml,...
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Cookie: cookieA=aaaaaa; cookieB=bbbbbb
Upgrade-Insecure-Requests: 1
```

- Setting cookies in http response

```
GET: HTTP/1.1 200 OK
Date: Wed, 25 Aug 2021 20:40:15 GMT
Server: Apache/2.4.41 (Ubuntu)
Set-Cookie: cookieA=aaaaaa
cookieB=bbbbbb; expires=Wed, 25-Aug-2021 21:40:15 GMT; Max-Age=3600
Content-Length: 28
Keep-Alive: timeout=5, max=99
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8
```

Same-Origin Policy (SOP)

The same-origin policy restricts scripts on one origin from accessing data from another origin.

An origin consists of a URI scheme, domain and port number.

http://www.website.com/example/index.html

Example of SOP

http://normal-website.com

URL accessed	Access permitted?
http://normal-website.com/example/	Yes: same scheme, domain, and port
http://normal-website.com/example2/	Yes: same scheme, domain, and port
https://normal-website.com/example/	No: different scheme and port
http://en.normal-website.com/example/	No: different domain
http://www.normal-website.com/example/	No: different domain
http://normal-website.com:8080/example/	No: different port*

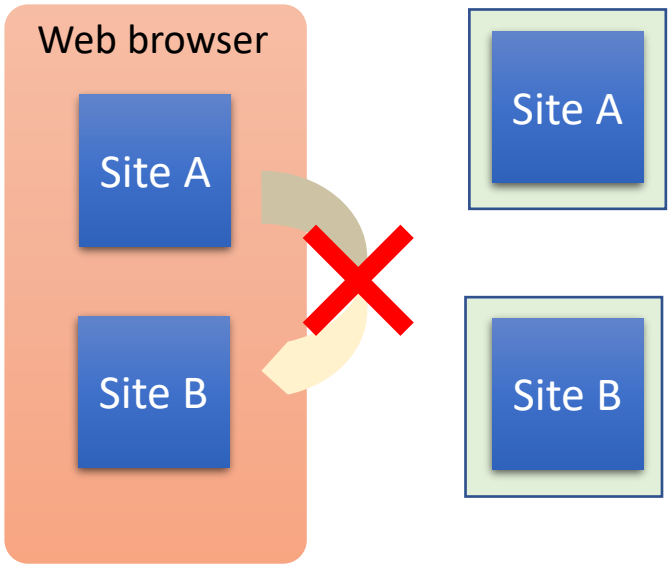
Why same-origin policy important?

It isolates sites from one another

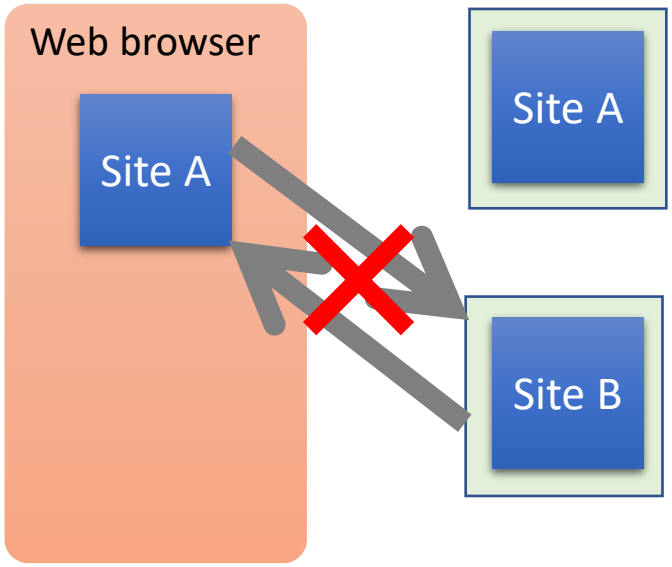
Prevent data from being read from other sites

Allow you to browse the web safely

SOP

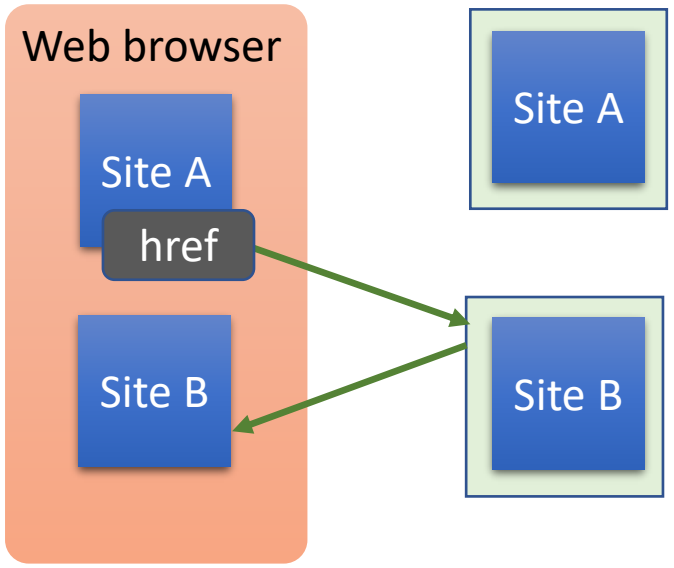


Accessing site resources within the Browser



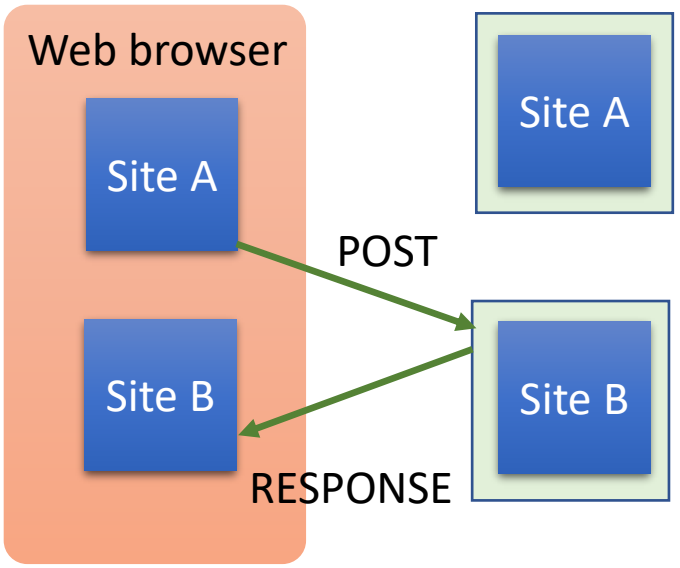
Requesting resources from the server

SOP with href



- Site can hyperlink to any other site <a href=”http://siteb.com/”>
- The response loads in a new context, the originating site is replaced by the new site.
- Any site can link, but can’t read the response

How does SOP apply to forms?



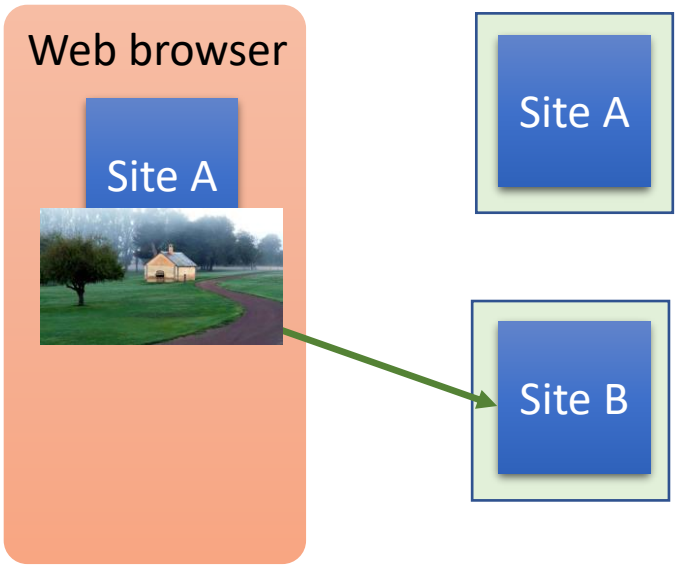
- Forms can POST to any site

  <form action = “http://siteb.com/” method=“POST” >

- The response loads in a new context, the originating site is replaced by the new site.

- Any site can POST, but can’t read the response

How does SOP apply to images?

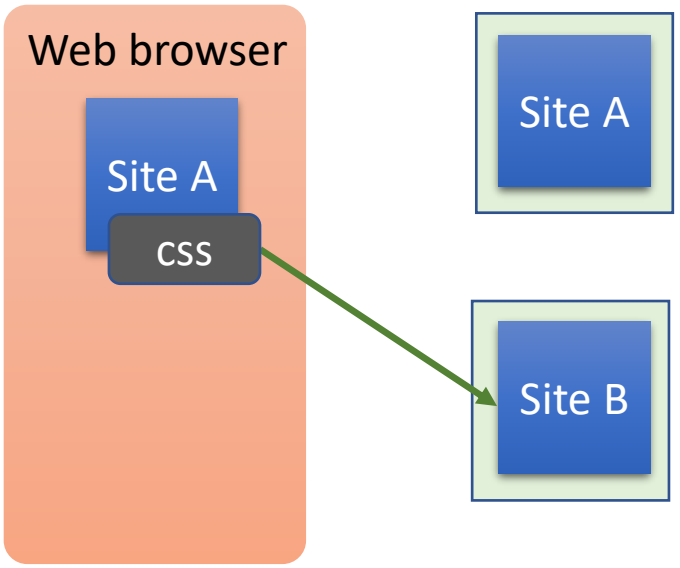


- Any site can display images from other sites

- However, sites can’t read the image data from other sites.

- Also applies to other static media like movies, audio, etc

How does SOP apply to CSS?

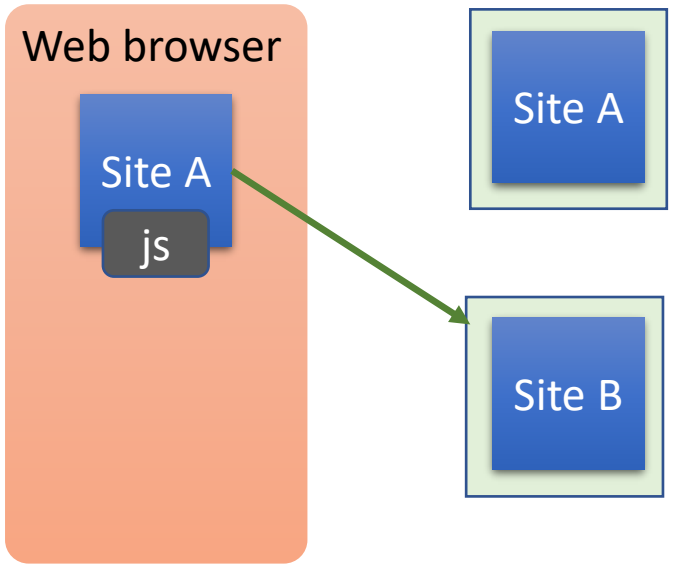


- Any site can include CSS from any other site

  <link rel = “stylesheet” type = ”text/css” href= “http://siteb.com/site.css” />

- However, sites can’t read the CSS data from other sites.

How does SOP apply to javascript includes?



- Any site can include javascript from any other site

  <script src = ”http:siteb.com/jb.js” />

- The javascript runs in the originating site

SOP with web storage

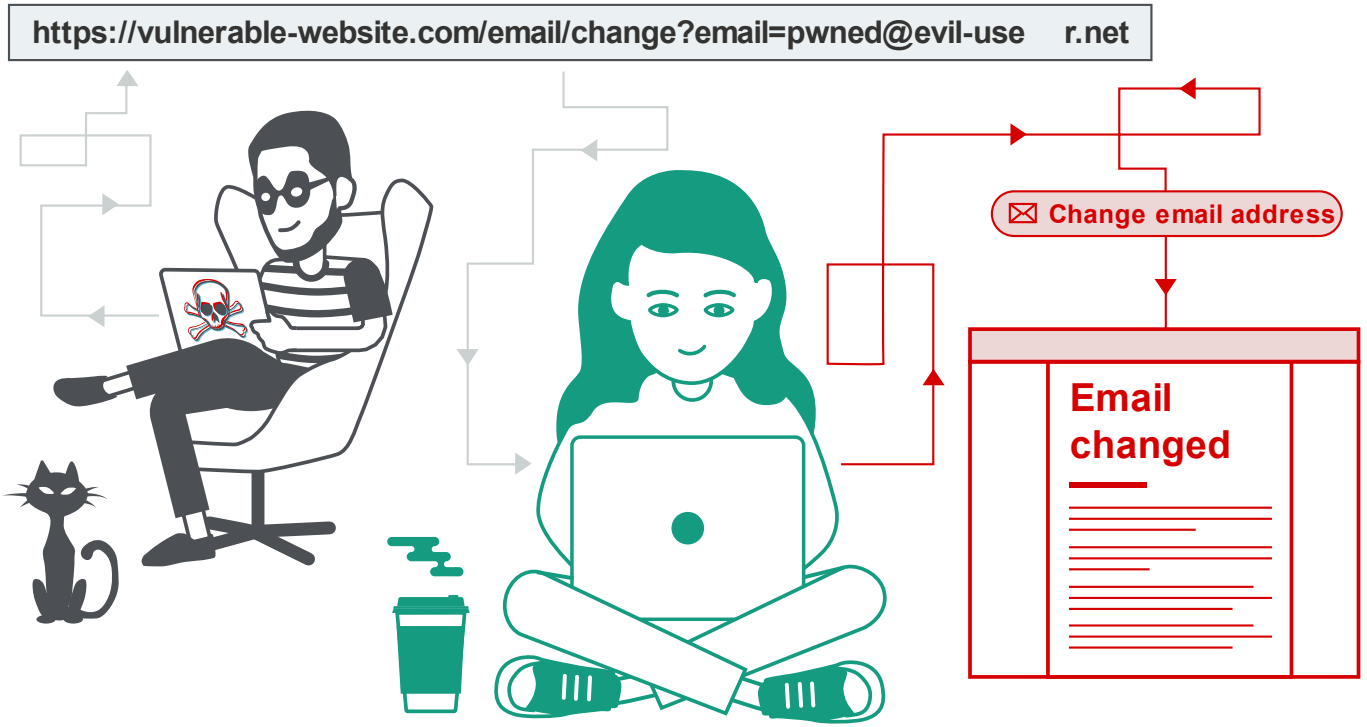
Local storage

- Share between windows with the same origin
- Survives when windows is closed

Session storage

- Only the current window
- Limited to lifetime of window

SOP Demonstration



CSRF Attack

Cross-site request forgery (also known as CSRF) is a web security vulnerability that allows an attacker to induce users to perform actions that they do not intend to perform. It allows an attacker to partly circumvent the same origin policy, which is designed to prevent different websites from interfering with each other.

How CSRF works

Via a CSRF attack, the attacker can bypass the authentication process or perform actions with higher privileges. What to do to execute this attack:

- Create the custom payload
- Embed the request into a hyperlink
- Trick the victim into clicking the link which will send your request to the website.
- Forge the request to conduct malicious action

How CSRF works

This attack only works if the victim is an **authenticated user** because when the request is made, the application will check if the cookies of a valid session are available. If the relevant cookies are available, those will need to be sent with the request. If the session is valid and the website approves the sent cookies, CSRF attack becomes successful.

The Impact of a CSRF Attack

With CSRF attacks, the impact of the attack depends on the level of permissions that the victim has on the application. As also mentioned above, some of the actions that a CSRF attack can cause are given below:

- Changing victim's password
- Allowing monetary transaction on behalf of the victim
- Changing/Adding/Deleting actions available on the website

CSRF Example - GET

```
24  <form action="http://vulnerable/endpoint" method="POST">
25      <input name="parameter" type="hidden" value="CSRFd" />
26      <input type="submit" value="Submit Request" />
27  </form>
```

CSRF Example - POST

```
33  <form id="autosubmit" action="http://vulnerable/endpoint" method="POST">
34      <input name="parameter" type="hidden" value="CSRFd" />
35      <input type="submit" value="Submit Request" />
36  </form>
37  <script>
38      document.getElementById("autosubmit").submit();
39  </script>
```

CSRF Payloads

HTML GET:

Click

HTML GET (no interaction):

<img src="http://vulnerable/endpoint?parameter=CSRFd">

CSRF Payloads

HTML POST

```
<form action="http://vulnerable/endpoint" method="POST">
    <input name="parameter" type="hidden" value="CSRFd" />
    <input type="submit" value="Submit Request" />
</form>
```

HTML POST (no interaction)

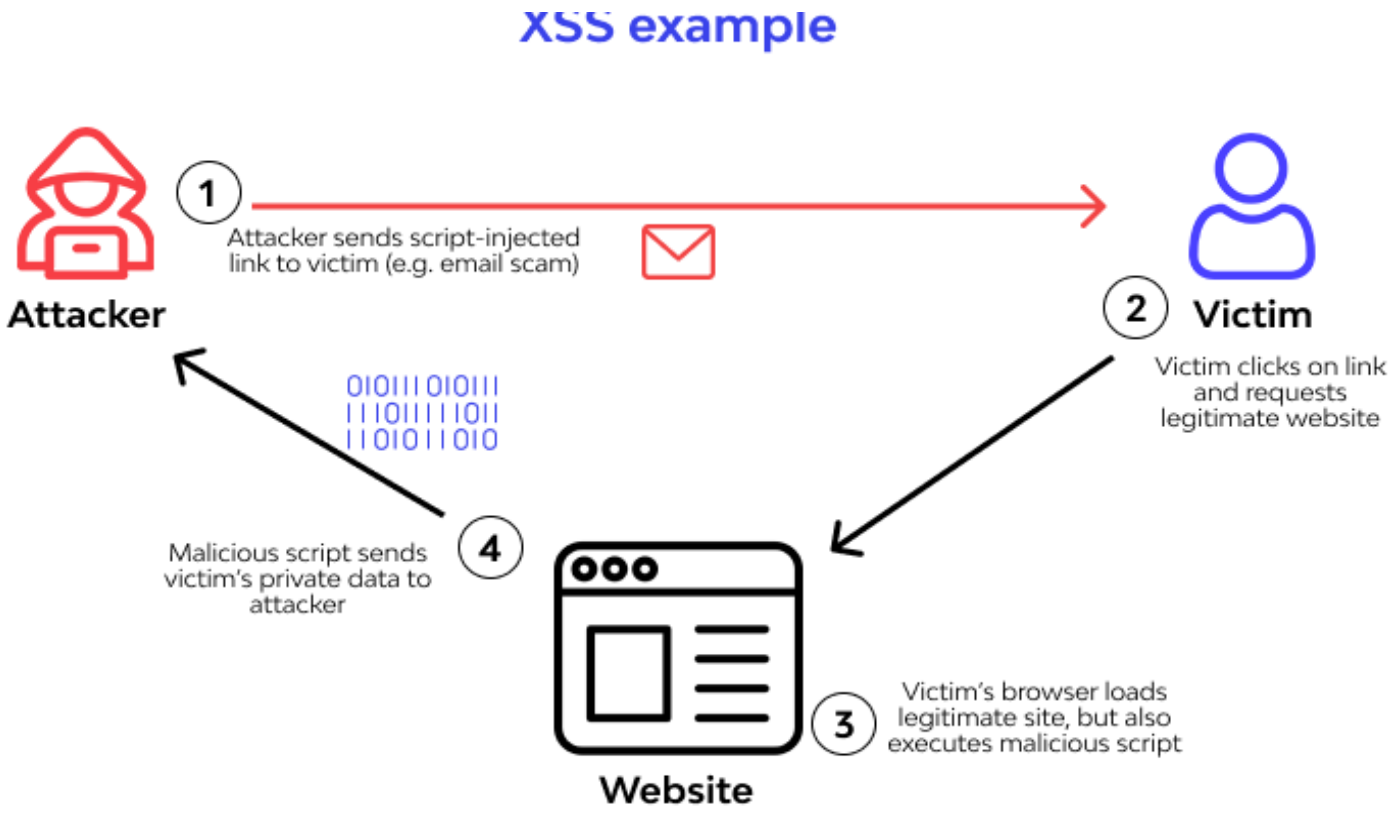
```
<form id="autosubmit" action="http://vulnerable/endpoint" method="POST">
    <input name="parameter" type="hidden" value="CSRFd" />
    <input type="submit" value="Submit Request" />
</form>
<script>
    document.getElementById("autosubmit").submit();
</script>
```

CSRF Countermeasures

- Use CSRF token in HTTP header and match its value on server side.
- Do not use GET HTTP method for critical operations such as Create/Update/Delete
- Implement “Same-site” Attribute to cookies.
- Remember to check the OWASP cheat sheet.

CSRF Demonstration

With DWVA (Damn Vulnerable Web App)



XSS Attack

Cross-site scripting (also known as XSS) is an injection attack in which an attacker tries to inject some malicious code in the website's input field. This code is merely one- or two-line script written in any language that a web browser executes.

How XSS works

In XSS attack an attacker tries to inject some HTML code or some JavaScript code in the input field of the website. The input field can be anywhere in the website's page. The user gives his input using this input field. The input field can be a HTML form, Signup button.

Types of XSS Attack

01

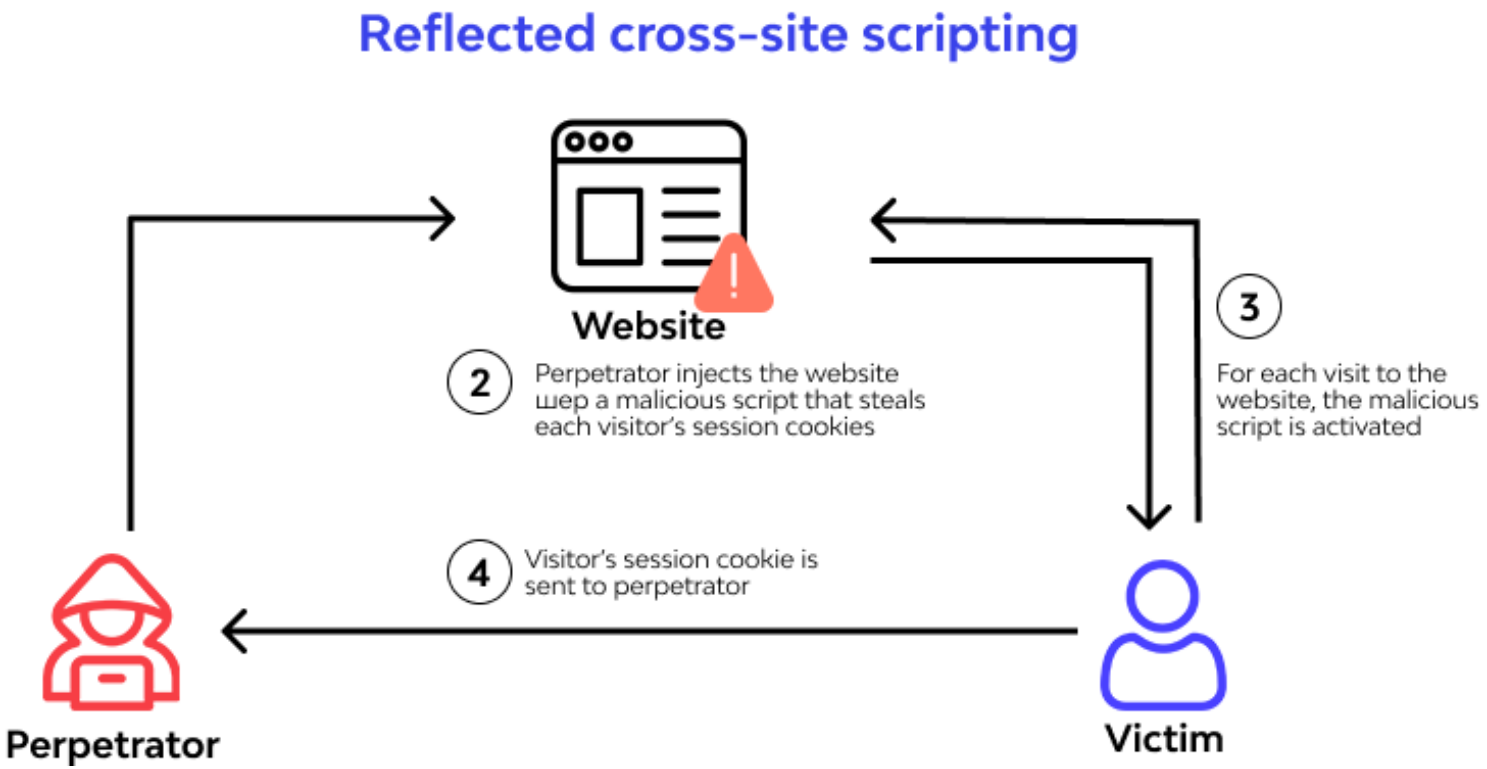
Reflected XSS

02

Stored XSS

03

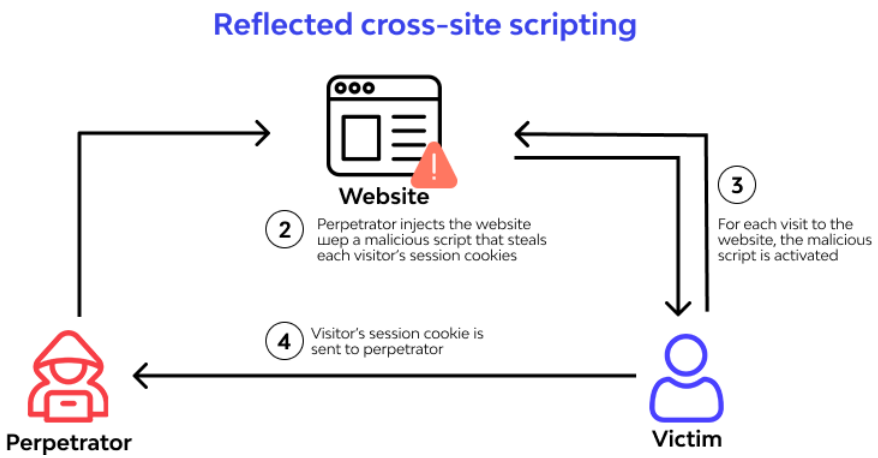
DOM Based XSS



Reflected/Non-Persistent/Type-II XSS

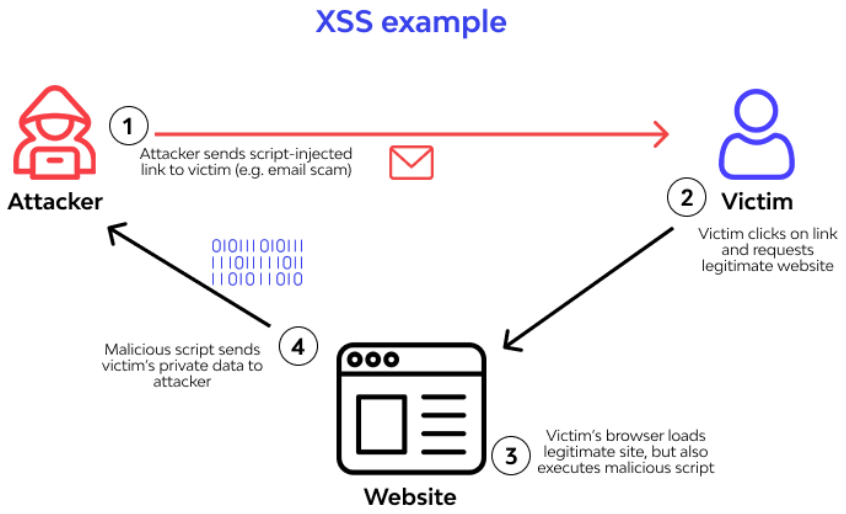
The input supplied by the user reflects back in the browser window or inside page source of the web page

Stored/Persistent/Type I XSS



This type of vulnerability arises whenever a web application stores user supplied data for later use in backend without performing any filter or input sanitization.

Since the web application does not apply any filter therefore an attacker can inject some malicious code into this input field. This malicious code can also be a valid XSS payload. So, whenever any person visit the vulnerable page where malicious code is injected, he will get a popup on his browser window.



Stored XSS is most seen in

- Forum or Message Board
- Blog comments of any post
- Shopping cart of e-commerce website
- User Profiles Page: the application allows the user to edit/change profile details such as first name, last name, avatar, picture, address, etc.
- File Manager: application that allows upload of files
- Application settings or preferences: application that allows the user to set preferences
- Log: if the application stores some users input into logs.


To be continued