**Ôn tập chuẩn bị checkpoint**

**Câu 1: (Authen-Author) Trong lab Username enumeration via response timing, tại sao response sẽ bị delay thời gian trả về khi mình nhập đúng user và password có độ dài cực lớn. Phân biệt phân quyền theo chiều ngang và chiều dọc**

**Trả lời**

1. Trong lab Username enumeration via response timing, response sẽ bị delay thời gian trả về khi mình nhập đúng user và password có độ dài cực lớn là do: Khi kiểm tra đúng username thì mới kiểm tra đến password. Khi đó thực hiện hash mật khẩu và so sánh với mật khẩu được lưu trong cơ sở dữ liệu để xác thực người dùng. Cơ chế hash thường sẽ thu được độ dài cố định, bởi vậy khi mà nhập password quá dài vào thì thời gian hash để đưa về độ dài cố định sẽ lâu hơn là password ngắn.

2. Phân biệt phân quyền theo chiều dọc và phân quyền theo chiều ngang

+ Phân quyền theo chiều dọc (**Vertical access controls**): Vai trò thấp có khả năng truy cập chức năng của vai trò cao hơn có nghĩa là **user bình thường có thể truy cập vào các chức năng của admin**.

+ Phân quyền theo chiều ngang (**Horizontal access controls**): Hai vai trò có quyền hạn bằng nhau (2 người dùng thông thường) có khả năng truy cập chức năng dưới vai trò của người kia -> **giữa các user ngang hàng với nhau, IDOR.**

**IDOR (Insecure Direct Object Reference):**

**+** Lỗ hổng này xảy ra khi chương trình cho phép người dùng truy cập tài nguyên (dữ liệu, file, thư mục, database..) một cách trực tiếp thông qua dữ liệu do người dùng cung cấp nhưng kém an toàn. Hậu quả dẫn tới cho phép attacker có thể bypass phân quyền và truy cập trực tiếp tới các tài nguyên trái thẩm quyền bằng cách sửa đổi tham số – cái được dùng để tham chiếu đến đối tượng. Attacker có thể xem được các thông tin nhạy cảm của các người dùng khác, thực thi các chức năng mà bình thường không được phép, thậm chí có thể dẫn tới chiếm đoạt tài khoản khác.

+ Một số ví dụ:

* **Lỗ hổng IDOR với tham chiếu trực tiếp đến các đối tượng cơ sở dữ liệu**

+ Ví dụ truy cập tài khoản khách hàng thông qua ID thì có thể sửa ID để vào tài khoản của khách hàng khác => IDOR leo thang đặc quyền theo chiều ngang.

+ IDOR leo thang đặc quyền theo chiều dọc: Thay đổi người dùng thành một người dùng có đặc quyền khi bỏ qua các biện pháp kiểm soát truy cập. Hoặc khai thác rò rỉ mật khẩu, sửa đổi các tham số khi kẻ tấn công đã truy cập vào trang tài khoản của người dùng.

* **Lỗ hổng IDOR với tham chiếu trực tiếp đến các tệp tĩnh:** Phát sinh khi tài nguyên nhạy cảm được đặt trong các tệp tĩnh.

**+** Ví dụ: một trang web có thể lưu bản chép lại tin nhắn trò chuyện vào đĩa bằng cách sử dụng tên tệp tăng dần và cho phép người dùng truy xuất những bản ghi này bằng cách truy cập một URL. Kẻ tấn công có thể chỉ cần sửa đổi tên tệp để truy xuất bản ghi do người dùng khác tạo và có khả năng lấy thông tin đăng nhập của người dùng cũng như dữ liệu nhạy cảm khác.

**Câu 2: (Sqli) Khai thác boolean-based sqli như nào? Có cách nào tối ưu để tốn ít request không?**

**Trả lời**

**Khai thác boolean-based sqli:** Boolean-based SQL injection là một kiểu tấn công SQL injection trong đó kẻ tấn công gửi một truy vấn SQL tới CSDL, tùy thuộc vào kết quả của truy vấn đó là TRUE hay FALSE thì ứng dụng sẽ trả về các kết quả khác nhau. => **Câu truy vấn đúng trả về response khác với câu truy vấn sai.**

**Cách tối ưu để tốn ít request:** Sử dụng tìm kiếm nhị phân.

Cách dùng tìm kiếm nhị phân trong boolean-based sqli: Ý tưởng cơ bản là gửi tải trọng sẽ trả về đúng hoặc sai, sau đó sử dụng thuật toán tìm kiếm nhị phân để thu hẹp phạm vi giá trị có thể cho đến khi lấy được thông tin mong muốn.

1. Xác định giá trị mục tiêu và phạm vi của các giá trị có thể.

2. Gửi payload để kiểm tra xem giá trị mục tiêu có lớn hơn, nhỏ hơn hoặc bằng điểm giữa của phạm vi hay không.

3. Dựa trên phản hồi, hãy chia phạm vi thành hai phần nhỏ hơn và lặp lại quy trình cho đến khi tìm thấy giá trị mục tiêu.

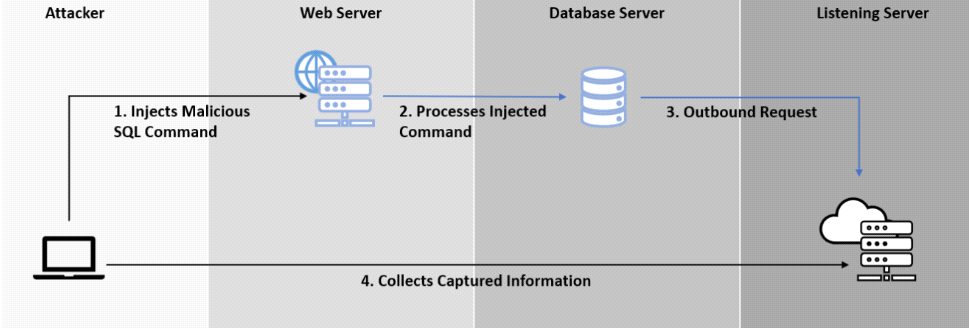
**Câu 3: Trình bày kỹ thuật out of band để khai thác SQLi. Trong trường hợp nào thì dùng out of band**

**Trả lời**

**Trình bày kỹ thuật out of band để khai thác SQLi:**

**+** Là một loại SQL injection mà kẻ tấn công không nhận được phản hồi từ ứng dụng bị tấn công trên cùng một kênh liên lạc mà thay vào đó có thể khiến ứng dụng gửi dữ liệu đến một điểm cuối từ xa mà họ kiểm soát.

+ Dùng DNS out of band: Dữ liệu được lọc ra bằng cách sử dụng các truy vấn DNS. Kịch bản tấn công



Kẻ tấn công gửi sql độc hại đến web server -> Web server truy vấn tới DB server -> DB server gửi dns outbound request đến server mà kẻ tấn công kiểm soát -> Kẻ tấn công lấy dữ liệu qua dns

**Trường hợp dùng out of band:** Khi các kỹ thuật tiêm suy luận như dựa trên thời gian hay phân biệt theo đúng sai không hoạt động hoặc khi không có response trả về từ máy chủ.

**Cách lấy dữ liệu qua dns:** Sniff(bắt) các gói tin đi qua cổng 53 thì subdomain chính là dữ liệu cần tìm.

**Câu 4: (FileUpload) Trình bày một trường hợp unsafe file upload như nào? Trình bày một số định dạng file malicious**

**Trả lời**

**Một trường hợp unsafe file upload:**

**+** Không check extension (hay còn gọi là phần mở rộng, hay đuôi của file), **cho phép upload webshell** (**một tập lệnh độc hại** - là một tập lệnh cho phép truy cập shell từ xa vào hệ điều hành của máy chủ web thông qua kết nối HTTP) ví dụ php.

Một số hàm phổ biến nhất được sử dụng để thực thi các lệnh shell trong PHP: system(), exec(), shell\_exec(), passthru(), proc\_open(),…

+ Upload một file không phải là ảnh nhưng có phần header là ảnh. (Convert sang HEX thêm các byte header ảnh sau đó convert lại ascii).

**Một số định dạng file malicious:** php, asp, jsp, hta, js,….

**Câu 5: (FileUpload) Lab4 File Upload, vai trò của .htaccess. Tại sao sửa .htaccess có thể gây RCE**

**Trả lời**

**Lab4 File Upload vai trò của .htaccess là:** htaccess có thể được sử dụng để thêm trình xử lý nhằm xử lý một số loại tệp nhất định. Trong bài lab này, do file .php không được phép upload nên sử dụng .htaccess để **add thêm trình xử lý PHP (php handler) cho extension** khác không phải .php.

**Sửa .htaccess có thể gây RCE (Remote code execution – thực thi mã từ xa)** nếu tệp chứa mã độc hại hoặc nếu sử dụng cú pháp không chính xác. Tệp .htaccess được sử dụng để định cấu hình cài đặt phía máy chủ cho trang web, chẳng hạn như chuyển hướng URL, thiết lập trang lỗi tùy chỉnh và giới hạn quyền truy cập vào một số địa chỉ IP nhất định. Nếu tệp .htaccess chứa mã cho phép thực thi mã tùy ý, kẻ tấn công có thể khai thác lỗ hổng này để chạy mã của chính chúng trên máy chủ bị ảnh hưởng. Ngoài ra, nếu tệp .htaccess được viết với cú pháp không chính xác, nó có thể khiến máy chủ web gặp trục trặc, có khả năng cho phép kẻ tấn công thực thi mã.

**Câu 6: (InfoDisclosure) Trình bày 3 trường hợp info disclosure thường gặp. Theo em để tìm các lỗi này nhanh nhất thì nên làm như nào?**

**Trả lời**

**Ba trường hợp info disclosure thường gặp:** lộ qua thông báo lỗi, lộ file backup, lộ .git

**Để tìm các lỗi này nhanh nhất thì dùng tool để quét.** Ví dụ: **Dirsearch**

**Câu 7: (XXE) XXE là gì, dấu hiệu nhận biết website có thể bị XXE?**

**Trả lời**

**XXE (XML external entity):** là một lỗ hổng cho phép kẻ tấn công đưa dữ liệu độc hại vào một tệp XML. Khi bộ xử lý XML xử lý tệp này, dữ liệu độc hại có thể khiến bộ xử lý rò rỉ thông tin nhạy cảm, tạo kết nối mạng trái phép hoặc thậm chí thực thi mã tùy ý. Lỗ hổng này xảy ra khi bộ xử lý XML không được cấu hình đúng cách để xử lý các thực thể bên ngoài và có thể bị khai thác bởi kẻ tấn công có thể kiểm soát nội dung của tệp XML mà bộ xử lý sẽ xử lý.

**Dấu hiện nhận biết website có thể bị XXE:**

1. Ứng dụng gửi request có XML data

2. Xử lý dữ liệu XML mà không vô hiệu hóa các thực thể bên ngoài.

3. Xử lý dữ liệu XML từ các nguồn không đáng tin cậy.

**Câu 8: (XXE) Website có endpoint bị lỗi blind XXE. Có trường hợp nào vẫn khai thác được lỗ hổng này để leak data không?**

**Trả lời**

Thử DNS để nhận biết XXE

Có outbound để sử dụng extermal dtd để leak data

**WEB07: Cross-site Scripting (XSS)**

Là một loại vấn đề tiêm (injection), trong đó các script độc hại được tiêm vào các trang web đáng tin cậy. XSS xảy ra khi một kẻ tấn công sử dụng ứng dụng web để gửi mã độc hại (thường tiêm vào link hoặc mã nguồn của một trình duyệt) tới một người dùng khác.

XSS hoạt động bằng cách thao túng một trang web dễ bị tấn công để trả về JS độc hại cho những người dùng khác. Khi đoạn code độc hại được thực thi trong trình duyệt của nạn nhân, kẻ tấn công có thể chiếm hoàn toàn sự tương tác cả nạn nhân với ứng dụng

**Các dạng tấn công XSS**

- 3 dạng tấn công chính:

**+ Reflected XSS - Type 1**

- Ứng dụng nhận dữ liệu (script độc hại) tới từ HTTP request và bao gồm dữ liệu đó trong phản hồi một cách không an toàn.

- Kẻ tấn công gửi đường link độc hại đến nạn nhân thông qua e-mail, tin nhắn hoặc trên một trang web khác. Khi nạn nhân click vào URL, script độc hại được thực thi trên browser của nạn nhân

- Tác động của các cuộc tấn công reflected XSS: Kẻ tấn công có thể

* Thực hiện bất kỳ hành động nào trong ứng dụng mà người dùng có thể thực hiện.
* Xem bất kỳ thông tin nào mà người dùng có thể xem.
* Sửa đổi bất kỳ thông tin nào mà người dùng có thể sửa đổi
* Bắt đầu tương tác với những người dùng ứng dụng khác, bao gồm cả các cuộc tấn công nguy hiểm, có vẻ như bắt nguồn từ người dùng nạn nhân ban đầu

- Cách tìm và test reflected XSS:

* Tìm thấy nhanh chóng và đáng tin cậy bằng cách sử dụng trình quét lỗ hổng web của Burp Suite.
* Test reflected XSS theo cách thủ công:

1. Kiểm tra tất cả đầu vào: Xem có thể nhập dữ liệu tại những đâu
2. Gửi giá trị ngẫu nhiên xem có reflected trong response không
3. Xác định context mà xảy ra reflected: có thể là thẻ HTML, cũng có thể là đoạn javacscript
4. Gửi payload thường: <script> alert()</script
5. Gửi các payload thay thế tương đương
6. Nếu tìm thấy payload hoạt động, thử test trên browser

- Sự khác nhau giữa Reflected XSS và Stored XSS:

* Reflected XSS: Lấy đầu vào từ HTTP request và nhúng đầu vào đó vào phản hồi ngay lập tức theo một cách không an toàn
* Stored XSS: Ứng dụng lưu trữ đầu vào và những vào phản hồi sau theo một cách không an toàn
* **Giống nhau: Các đoạn mã nguy hiểm sau khi được chèn vào sẽ được thực thi sau response của server, có nghĩa là lỗi nằm phía server.**

**-** Sự khác nhau giữa reflected XSS và self-XSS:

* Reflected XSS: Payload được gửi qua URL và được thực thi bởi nạn nhân khi họ truy cập URL => khai thác mỗi quan hệ tin cậy giữa website và người dùng
* Self-XSS: Payload do chính nạn nhân thực hiện khi họ dán vào thanh địa chỉ của trình duyệt của chính họ. Thực hiện self-XSS thường liên quan đến social engineering để nạn nhân dán một số thông tin đầu vào do kẻ tấn công cung cấp => khai thác mối quan hệ tin cậy giữa kẻ tấn công và nạn nhân.

**+ Stored XSS - Type 2 (XSS liên tục)**

**-** Phát sinh khi một ứng dụng nhận dữ liệu từ một nguồn không đáng tin cậy và bao gồm dữ liệu đó trong các HTTP response sau này theo một cách không an toàn

- script độc hại được lưu trữ vĩnh viễn trên server mục tiêu, ví dụ trong database

- script độc hại có thể được gửi lên ứng dụng thông qua HTTP request: comments trong blog, user nicknames, thông tin liên hệ trong order của khách hàng,....

- Chỉ ảnh hưởng đến người dùng đã login

**+ DOM-based XSS - Type 0**

DOM based XSS là kỹ thuật khai thác XSS dựa trên việc thay đổi cấu trúc DOM của tài liệu, cụ thể là HTML. Lỗ hổng bảo mật tồn tại phía client.

**Trường hợp phổ biến nhất**: Input sẽ được điền từ một phần của HTTP request, ví dụ URL query string parameter cho phép kẻ tấn công thực hiện cuộc tấn công bằng URL độc hại => tương tự reflected XSS.

window.location: lấy URL.

Giá trị có thể kiểm soát của kẻ tấn công => **source**

Chức năng nguy hiểm => **sink**

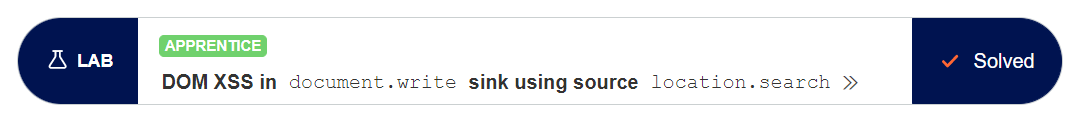
- Phát sinh khi JS lấy dữ liệu từ một nguồn có thể kiểm soát được của kẻ tấn công (source) sau đó chuyển dữ liệu này đến sink

**Sources** là thuộc tính JS chấp nhận dữ liệu có khả năng bị kẻ tấn công kiểm soát. Nói rộng ra thì bất kỳ property có thể được kiểm soát bời kẻ tấn công đều là một nguồn tiềm năng. Ví dụ: *location.search,*

**Sinks** là một hàm JS hoặc đối tượng DOM nguy hiểm tiềm ẩn có thể gây ra các tác động không mong muốn nếu dữ liệu do kẻ tấn công kiểm soát (source) được chuyển đến nó. Ví dụ: *document.write(), document.domain, element.innerHTML, innerHTML,….*

**Khai thác DOM XSS với sources và sinks khác nhau**

Một trang web dễ bị tấn công bởi DOM XSS nếu có một đường dẫn thực thi mà qua đó dữ liệu có thể truyền từ source đến sink



**location.search:** trả về một phần của URL bắt đầu bằng dấu “?” và chứa các query string parameter của URL. Sau đó query string parameter có thể được lấy bằng hàm JS: URLSearchParams.

**document.write:** được dùng để viết HTML cho tài liệu hiện tại

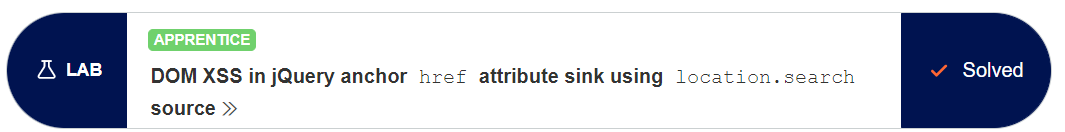
The *document.write* sink works with *script* elements



**innerHTML**: Đặt hoặc trả về nội dung HTML bên trong phần tử (Na ná document.write)

innerHTML không chấp nhận script => dùng img hoặc iframe

**Sources và sinks trong phần phụ thuộc của bên thứ ba**

****

**attr():** đặt hoặc trả về các thuộc tính và giá trị của các phần tử được chọn.

**DOM XSS kết hợp dữ liệu được phản chiếu và lưu trữ**

****

Trong lỗ hổng DOM XSS được phản ánh, máy chủ xử lý dữ liệu từ yêu cầu và lặp lại dữ liệu vào phản hồi. Dữ liệu được phản ánh có thể được đặt vào một chuỗi ký tự JavaScript hoặc một mục dữ liệu trong DOM, chẳng hạn như trường biểu mẫu. Sau đó, một tập lệnh trên trang sẽ xử lý dữ liệu được phản ánh theo cách không an toàn, cuối cùng ghi nó vào phần sink nguy hiểm.



Trong lỗ hổng DOM XSS được lưu trữ, máy chủ nhận dữ liệu từ một yêu cầu, lưu trữ và sau đó bao gồm dữ liệu trong phản hồi sau. Một script trong response sau này chứa sink mà sau đó xử lý dữ liệu theo cách không an toàn.

**Cách tìm và test XSS**

Test với Reflected và Stored XSS: Gửi đầu vào đơn giản vào mọi điểm nhập trong ứng dụng, xác định mọi vị trí mà đầu vào được trả về trong HTTP response và kiểm tra từng vị trí xem có chỗ nào phù hợp để thực thi js hay không

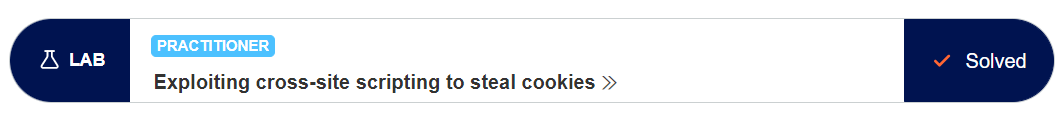
Test với DOM-based XSS:

+ Phát sinh từ URL parameter: Đặt đầu vào đơn giản vào tham số, tìm kiếm đầu vào trong DOM và kiểm tra từng vị trí xem có khai thác được không

+ Không dựa trên URL hoặc sinks không dựa trên HTML (vd: setTimeout): Không có cách nào khác ngoài xem source JS

**Exploiting cross-site scripting vulnerabilities**

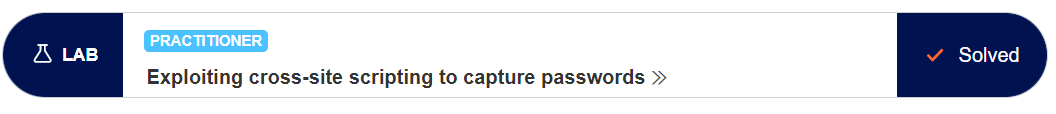
- 3 cách trong số những cách phổ biến và mạnh mẽ nhất để khai thác lỗ hổng XSS



Có thể khai thác lỗ hổng cross-site scripting để gửi cookie của nạn nhân đến miền của riêng bạn, sau đó đưa cookie vào trình duyệt theo cách thủ công và mạo danh nạn nhân.

Trong thực tế, phương pháp này có một số hạn chế đáng kể:

* Nạn nhân có thể không login
* Nhiều ứng dụng ẩn cookie của chúng khỏi JavaScript bằng cách sử dụng cờ HttpOnly
* Các phiên có thể bị khóa đối với các yếu tố bổ sung như địa chỉ IP của người dùng
* Phiên có thể hết trước khi bạn có thể chiếm đoạt nó



Nhiều người dùng sử dụng password manager để tự động điền mật khẩu. Khi trình duyệt tìm thấy một thẻ đầu vào loại “mật khẩu”, nó sẽ tự động điền mật khẩu vào thẻ đó. Có thể tận dụng điều này bằng cách tạo đầu vào mật khẩu, đọc mật khẩu được điền tự động và gửi mật khẩu đến miền riêng.



Đổi email mà không cần nhập lại mật khẩu

**Cross-site scripting contexts**

Khi test reflected và stored XSS, nhiệm vụ chính là xác định bối cảnh:

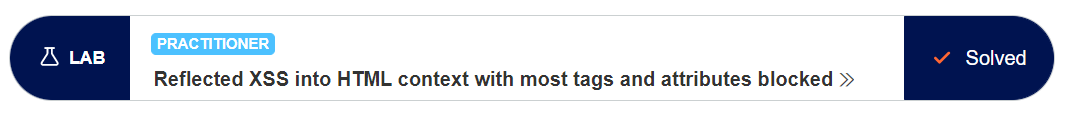
1. Vị trí trong response xuất hiện dữ liệu có thể kiểm soát bởi kẻ tấn công
2. Mọi xác thực đầu vào hoặc quá trình xử lý khác đang được ứng dụng thực hiện trên dữ liệu đó.

Dựa vào hai yếu tố này để chọn payload phù hợp

**XSS between HTML tags**

- Khi bối cảnh XSS là văn bản giữa các thẻ HTML, cần sử dụng một số thẻ HTML mới được thiết kế để kích hoạt thực thi JavaScript.

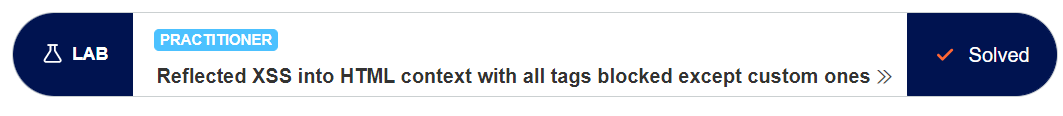
Ví dụ: *<script>alert(document.domain)</script>, <img src=1 onerror=alert(1)>*



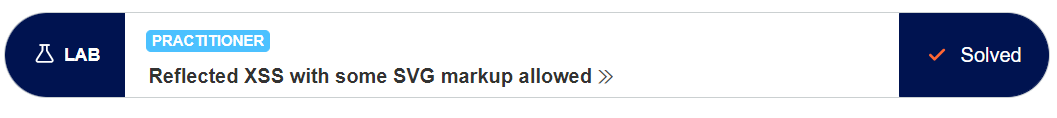
Sử dụng Burp Intruder để tìm tag và event được phép sử dụng.

Sự kiện onresize(): Gọi một hàm khi cửa sổ trình duyệt bị thay đổi kích thước.

gọi hàm print() khi cửa sổ trình duyệt thay đổi kích thước



Sử dụng **tabindex (bỏ qua câu này)**

****

Bỏ qua câu này

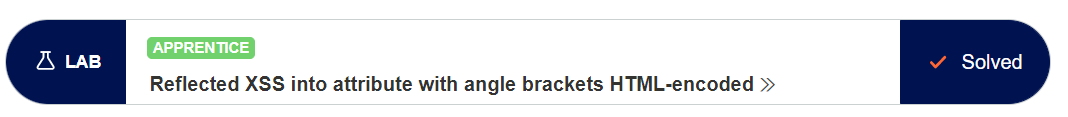
**XSS in HTML tag attributes**

Khi bối cảnh XSS ở trong một giá trị thuộc tính thẻ HTML, đôi khi có thể **chấm dứt giá trị thuộc tính, đóng thẻ** và giới thiệu một giá trị mới.

Ví dụ: *"><script>alert(document.domain)</script>*

Trường hợp phổ biến hơn, dấu ngoặc nhọn bị chặn hoặc mã hóa, vì vậy thông tin nhập không thể thoát ra khỏi thẻ chứa thông tin đó. Với điều kiện **có thể chấm dứt giá trị thuộc tính**, thông thường bạn có thể giới thiệu một thuộc tính mới để tạo ngữ cảnh có thể viết tập lệnh, chẳng hạn như **thêm trình xử lý sự kiện**

Ví dụ: *" autofocus onfocus=alert(document.domain) x="*



Đôi khi ngữ cảnh XSS là một loại thuộc tính thẻ HTML mà chính nó có thể tạo ngữ cảnh có thể viết được. Tại đây, bạn có thể thực thi JavaScript mà không cần chấm dứt giá trị thuộc tính. Nếu bối cảnh XSS nằm trong thuộc tính href của thẻ **a**, bạn có thể sử dụng giao thức giả javascript để thực thi tập lệnh.

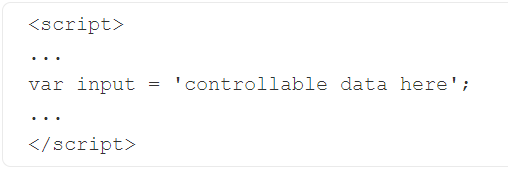


**XSS into JavaScript**

Khi bối cảnh XSS là một số JavaScript hiện có trong phản hồi.

- Chấm dứt tập lệnh hiện có (đóng script)

Trong trường hợp đơn giản nhất, có thể chỉ cần đóng thẻ script đang bao quanh JavaScript hiện có và giới thiệu một số thẻ HTML mới sẽ kích hoạt thực thi JavaScript.



Lý dó hoạt động là vì: Trình duyệt thực hiện phân tích cú pháp HTML để xác định các thành phần trang bao gồm các khối script trước, sau đó mới thực hiện phân tích cú pháp JavaScript để hiểu và thực thi các tập lệnh được nhúng.

- Thoát ra khỏi chuỗi JS

Trong trường hợp bối cảnh XSS nằm trong một chuỗi ký tự được trích dẫn, thường có thể thoát ra khỏi chuỗi và thực thi JavaScript trực tiếp. (TH1)

Một số ứng dụng cố gắng chặn đầu vào thoát ra khỏi chuỗi JS bằng cách thoát ‘ bằng dấu \ (**nghĩa là thêm ‘ thì ứng dụng tự động thêm \**). Trong tình huống này, các ứng dụng thường mắc lỗi không thoát khỏi ký tự dấu gạch chéo ngược. Kẻ tấn công có thể sử dụng ký tự dấu gạch chéo ngược của riêng chúng để vô hiệu hóa dấu gạch chéo ngược do ứng dụng thêm vào (TH2).

**Sử dụng mã hóa HTML**

Khi bối cảnh XSS là một số JavaScript hiện có trong thuộc tính thẻ được trích dẫn, chẳng hạn như trình xử lý sự kiện, có thể sử dụng mã hóa HTML để xử lý xung quanh một số bộ lọc đầu vào.

Khi trình duyệt đã phân tích cú pháp các thẻ và thuộc tính HTML trong một phản hồi, trình duyệt sẽ thực hiện giải mã HTML các giá trị thuộc tính thẻ trước khi chúng được xử lý thêm. Nếu ứng dụng phía máy chủ chặn hoặc khử trùng một số ký tự cần thiết để khai thác XSS thành công, bạn thường có thể bỏ qua xác thực đầu vào bằng cách mã hóa HTML các ký tự đó.

**XSS trong JavaScript template literals**

Template literals là một cú pháp mới để khai báo String trong JS. String được viết trong dấu **`** thay vì dấu ‘ và cho phép biểu thức js được nhúng.

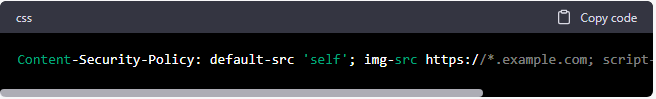
Khi bối cảnh XSS là trong JavaScript template literal, không cần thiết phải thoát chuỗi. Thay vào đó, Chỉ cần sử dụng cú pháp ${...} để nhúng một biểu thức JavaScript sẽ được thực thi khi chữ được xử lý.

**Content Security Policy(CSP):**

- Là một cơ chế bảo mật trình duyệt nhằm giảm thiểu tác động của XSS và một số lỗ hổng khác. Nó hoạt động bằng cách hạn chế các tài nguyên (chẳng hạn như tập lệnh và hình ảnh) mà một trang có thể tải và hạn chế liệu một trang có thể được đóng khung bởi các trang khác hay không.

- Để bật CSP: thêm trường **Content-Security-Policy** trong HTTP response

- Ví dụ về CSP



- Một số ví dụ về giá trị của trường CSP

+ default-src 'self': Tất cả nội dung phải đến từ cùng một nguồn với chính trang đó

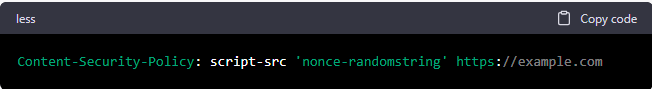
+ script-src 'self': script được tải từ cùng một nguồn gốc với chính trang đó

+ 'unsafe-inline' [https://example.com](https://example.com/): cho phép chạy inline script trong trang.

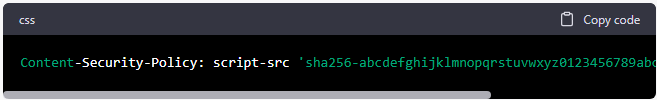
- Hai cách để chỉ định các tài nguyên đáng tin cậy: **nonces** và **hash**

+ nonces(number used once) và hash là các giá trị được tạo bởi CSP header cho mỗi request tới web để xác minh rằng một tập lệnh hoặc tài nguyên cụ thể được phê duyệt để được thực thi.

+ **nonces:** CSP có thể chỉ định một nonce (một giá trị ngẫu nhiên) và một giá trị tương tự phải được sử dụng trong **tag** tải script. Nếu random string mà không khớp thì script sẽ không được thực thi.



+ **hash:** Giá trị được sinh ra từ script (hash nội dung của script) và được sử dụng trong **script-src** của CSP header. Nếu hàm băng của script thực tế không khớp với giá trị trong **script-src** thì script không được thực thi. Nếu nội dung của script thay đổi thì cần cập nhật lại giá trị băm trong **script-src**



- Bỏ qua CSP bằng cách tiêm chính sách (Chưa hiểu)

- Bảo vệ chống clickjacking bằng CSP

**frame-ancestor:** Được sử dụng trong CSP header để kiểm soát và giới hạn các nguồn mà một trang web có thể được nhúng trong iframe. Chỉ định danh sách các nguồn gốc được phép nhúng trang web.

+ frame-ancestor ‘self’: Chỉ cho phép trang được đóng khung bởi các trang khác từ cùng một nguồn gốc

+ frame-ancestor ‘none’: Ngăn hoàn toàn việc đóng khung



Trong ví dụ này, chỉ trang web hiện tại và trang web https://example.com mới được phép nhúng trang web hiện tại vào iframe.

+ Sử dụng CSP để ngăn clickjacking linh hoạt hơn so với sử dụng tiêu đề X-Frame-Options vì có thể chỉ định nhiều miền và sử dụng ký tự đại diện

+ CSP cũng xác thực từng khung trong hệ thống phân cấp khung chính, trong khi X-Frame-Options chỉ xác thực khung cấp cao nhất.

**Dangling markup injection:** Là một kỹ thuật sử dụng để thu thập dữ liệu trên nhiều miền trong các trường hợp không thể khai thác XSS do các bộ lọc đầu vào hoặc do các biện pháp bảo vệ khác.

**WEB08: Cross-site request forgery (CSRF)**

**-** Là một lỗ hổng bảo mật web cho phép kẻ tấn công xúi giục người dùng thực hiện các hành động mà họ không có ý định thực hiện. Nó cho phép kẻ tấn công phá vỡ một phần chính sách same origin được thiết kế để ngăn các trang web khác nhau can thiệp lẫn nhau.

- **CSRF hoạt động thế nào?**

Để có thể tấn công CSRF, phải có ba điều kiện chính

* Một hành động có liên quan. Có một hành động trong ứng dụng mà kẻ tấn công có lý do để gây ra. Đây có thể là một hành động đặc quyền hoặc bất kỳ hành động nào đối với dữ liệu dành riêng cho người dùng
* Xử lý phiên dựa trên cookie: ứng dụng chỉ dựa vào cookie phiên để xác định người dùng đã thực hiện yêu cầu mà không dùng cơ chế nào khác.
* Không có parameter không thể đoán trước: Các yêu cầu thực hiện hành động không chứa bất kỳ tham số nào có giá trị mà kẻ tấn công không thể xác định hoặc đoán trước.

**- Cách phân phối khai thác CSRF**

**+** Cơ bản giống như Reflected XSS

+ Một số khai thác CSRF đơn giản sử dụng phương thức GET và có thể hoàn toàn độc lập với một URL duy nhất trên trang web dễ bị tấn công. Trong tình huống này, kẻ tấn công có thể không cần sử dụng trang web bên ngoài và có thể trực tiếp cung cấp cho nạn nhân một URL độc hại trên miền dễ bị tấn công.

**- XSS và CSRF**

**+** Sự khác nhau giữa XSS và CSRF

* XSS cho phép kẻ tấn công thực thi JavaScript tùy ý trong trình duyệt của người dùng nạn nhân.
* CSRF cho phép kẻ tấn công khiến người dùng nạn nhân thực hiện các hành động mà họ không có ý định.
* Hậu quả của XSS nghiêm trọng hơn:
* CSRF thường chỉ áp dụng cho một tập hợp con các hành động mà người dùng có thể thực hiện. Ngược lại, khai thác XSS thành công, kẻ tấn công có thể thực hiện bất kỳ hành động nào mà người dùng có thể thực hiện
* CSRF: kẻ tấn công có thể khiến nạn nhân đưa ra yêu cầu HTTP, nhưng chúng không thể truy xuất phản hồi từ yêu cầu đó. XSS: tập lệnh được chèn của kẻ tấn công có thể đưa ra các yêu cầu tùy ý, đọc phản hồi và lọc dữ liệu sang miền bên ngoài do kẻ tấn công chọn.

- Giả mạo request từ site khác. Ví dụ: Người dùng đăng nhập vào tài khoản ngân hàng của mình. Ngân hàng gửi lại người dùng session token (cookie). Hacker gửi cho người dùng một email chứa đường link độc hại trông có vẻ như là trỏ đến một site tin cậy nhưng thực tế là nó trỏ tới ngân hàng của người dùng. Khi người dùng click vào link, nó sẽ sử dụng session token trước đó, request của hacker được thực thi và tài khoản của người dùng bị hack**. Request của hacker tới ngân hàng đã giả mạo bằng cách sử dụng session token của người dùng mà nó không yêu cầu người dùng đăng nhập lại lần nữa.**

**=>** Hacker có thể sử dụng session token của nạn nhân mà không có thêm xác thực nào.

- Các bài lab:



Trong bài lab này thì ứng dụng chỉ dựa vào cookie session để xác định người dùng đã thực hiện yêu cầu. Không có cơ chế nào khác để theo dõi phiên hoặc xác thực yêu cầu của người dùng

**Dạng 1: Bypassing CSRF token validation**

**CSRF token là gì?**

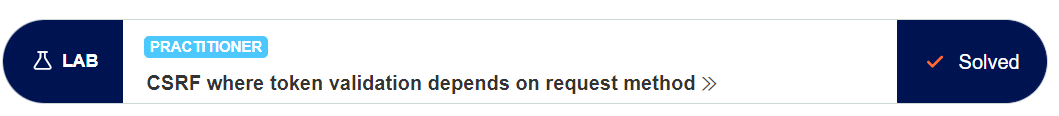
- Là một giá trị duy nhất, bí mật và không thể đoán trước được tạo bởi ứng dụng phía máy chủ và được chia sẻ với máy khách.

- Khi đưa ra yêu cầu thực hiện một hành động nhạy cảm, chẳng hạn như gửi biểu mẫu, ứng dụng khách phải bao gồm mã thông báo CSRF chính xác. Nếu không, máy chủ sẽ từ chối thực hiện hành động được yêu cầu.

- Một cách phổ biến để chia sẻ mã thông báo CSRF với khách hàng là bao gồm chúng dưới dạng tham số ẩn trong biểu mẫu HTML

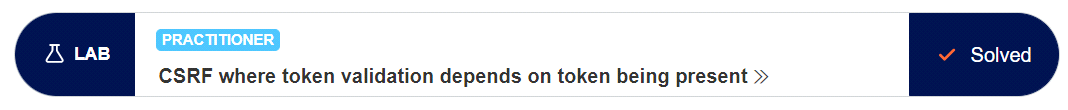
**Các lỗi phổ biến trong xác thực mã thông báo CSRF**

**Các lỗ hổng CSRF thường phát sinh do xác thực mã thông báo CSRF bị lỗi**



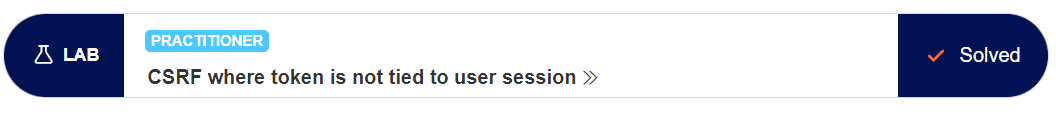
**Một số ứng dụng xác thực csrf token khi request sử dụng method POST nhưng bỏ qua xác thực khi sử dụng method GET**

Trong bài lab này, request có thêm tham số **crsf token,** nếu xóa tham số csrf token và gửi request thì request không được gửi đi (method POST). Nếu sửa method thành GET và xóa tham số csrf token thì request được gửi thành công



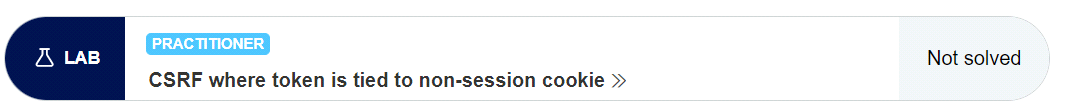
**Một số ứng dụng xác thực csrf token khi nó xuất hiện nhưng sẽ bỏ qua xác thực nếu không có csrf token**

Tương tự như bài lab trên, bài lab này request cũng có thêm tham số **crsf token,** nhưng nếu xóa tham số csrf token đi thì request vẫn được chấp nhận.

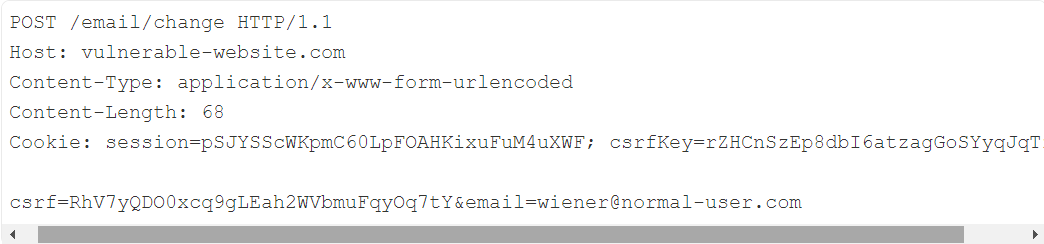


Một số ứng dụng không xác thực rằng csrf token thuộc cùng một phiên với người dùng đang thực hiện yêu cầu. Thay vào đó, ứng dụng duy trì **một nhóm csrf token chung** mà nó đã phát hành và **chấp nhận bất kỳ mã thông báo nào xuất hiện trong nhóm này**

Sử dụng csrf token change email chưa sử dụng của user này làm crsf token change email của user kia



**Một số ứng dụng liên kết csrf token với cookie (csrfKey cookie), nhưng không liên kết với cùng một cookie được sử dụng để theo dõi phiên (session cookie)**



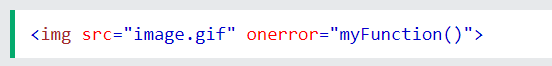
Trong request trên thì **csrf token** được liên kết với **csrfKey cookie** chứ không được liên kết với **session cookie.** Điều này xảy ra khi ứng dụng **sử dụng 2 frameworks khác nhau,** một để quản lý session cookie, một để quản lý csrfKey và **2 framework này không ràng buộc lẫn nhau.**

**(cơ chế bảo vệ csrf: csrfKey và csrf token không gắn với hệ thống xử lý session)**

Nếu trang web có bất kỳ hành vì nào cho phép hacker đặt csrfKey cookie trong trình duyệt của nạn nhân thì cuộc tấn công có thể xảy ra. Hacker truy cập vào tài khoản của mình, lấy giá trị csrfKey cookie và csrf token và tiến hành tấn công CSRF.

Kỹ thuật sử dụng để đặt csrfKey cookie của hacker vào trình duyệt của nạn nhân là : **HTTP Header Injection** - lỗ hổng xảy ra khi **http header được tạo tự động dựa trên đầu vào của người dùng**, do đó tùy thuộc vào cách nó xác thực đầu vào của người dùng hoặc nếu không có xác thực nào cho đầu vào của người dùng đó, có thể khai thác để yêu cầu trình duyệt đặt csrfKey cookie thành giá trị mà mình sở hữu.

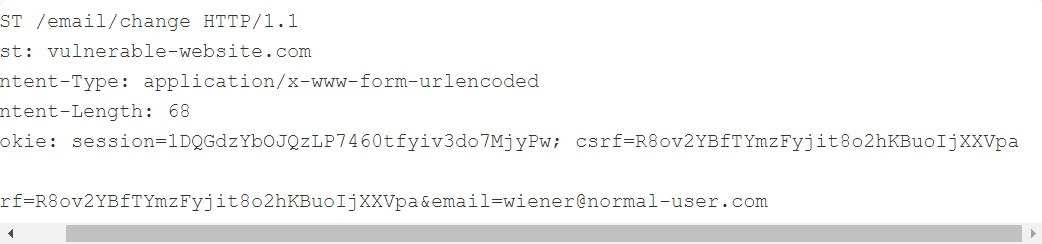
**onerror:** Thực thi JavaScript nếu xảy ra lỗi khi tải hình ảnh



**Cookie SameSite=None:** Cookie sẽ được gửi trong mọi ngữ cảnh, tức là trong các phản hồi đối với cả yêu cầu của bên thứ nhất và yêu cầu giữa các trang web. Nếu SameSite=None được đặt, thuộc tính **Secure** cookie cũng phải được đặt (nếu không cookie sẽ bị chặn).



**Một số ứng dụng không lưu token ở server (backend) mà thay vào đó lưu token trong COOKIE (csrf cookie) và REQUEST PARAM (csrf token). Khi yêu cầu tiếp theo được xác thực, ứng dụng chỉ cần xác minh csrf cookie = csrf token là được => được gọi là double submit defense**



Kẻ tấn công có thể thực hiện lại cuộc tấn công CSRF nếu trang web chứa bất kỳ chức năng cài đặt cookie nào và kẻ tấn công không cần lấy csrf token hợp lệ của mình.

csrf token vầ csrf cookie có giá trị như nhau => ứng dụng sử dụng **double submit cookie** hoăc **double submit defense**. Nếu csrf token và csrf cookie bằng nhau thì request mới được chấp nhận còn nếu không bằng nhau thì request không được chấp nhận.

Thông thường với cookie gửi hai lần, giá trị cookie không thành vấn đề vì nó không được lưu trữ ở backend, tất cả những gì nó làm là khi gửi yêu cầu thì check xem csrf token có bằng csrf cookie hay không => **giá trị của csrf token và csrf cookie không quan trọng, miễn là nó bằng nhau.**

Vẫn sử dụng kỹ thuật **HTTP Header Injection** để đặt csrf cookie vào trình duyệt của nạn nhân.

**Dạng 2: Bypassing SameSite cookie restrictions**

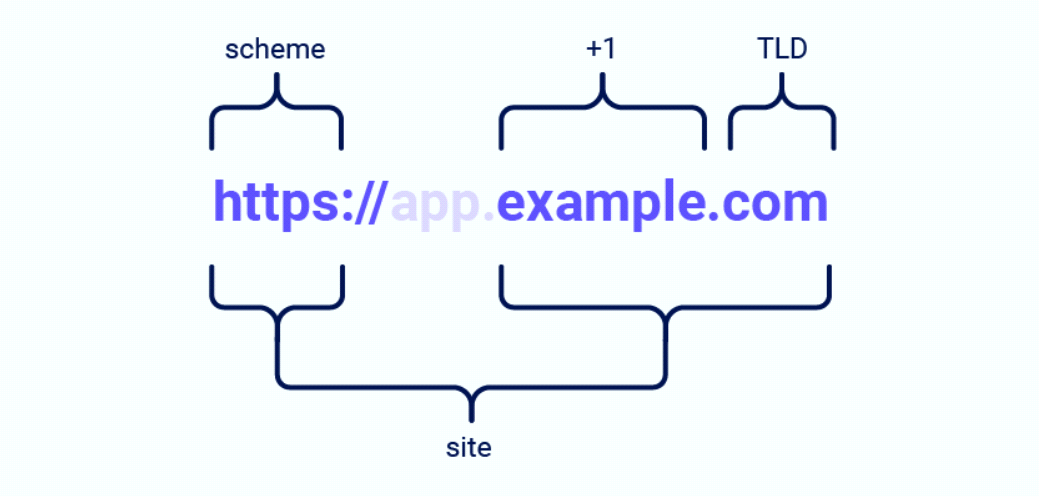
SameSite là một cơ chế bảo mật của trình duyệt xác định khi nào cookie của một trang web được bao gồm trong các request bắt nguồn từ trang web khác. Các hạn chế SameSite cookie cung cấp khả năng bảo vệ một phần trước nhiều cuộc tấn công giữa các trang, bao gồm CSRF, rò rỉ giữa các trang và một số khai thác CORS.

SameSite cookie:

Trong bối cảnh hạn chế SameSite cookie, site được định nghĩa là TLD (top-level domain) ex: *.com, .net*, cộng với một cấp độ bổ sung của tên miền. Điều này thường được gọi là TLD+1.

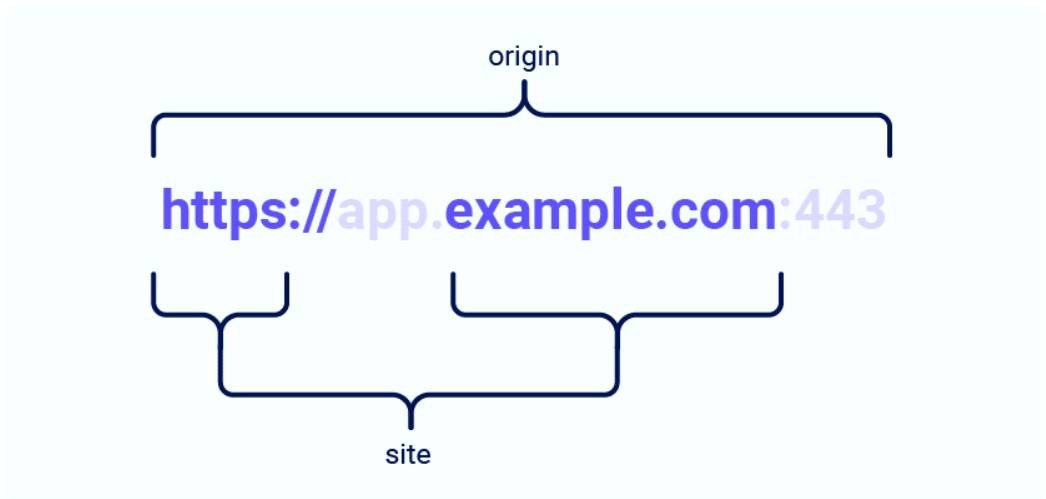
Khi xác định xem một yêu cầu có same-site hay không, URL scheme cũng được xem xét. Có nghĩa là:

[*http://app.example.com*](http://app.example.com/) *và* [*https://app.example.com*](https://app.example.com/)được hầu hết các trình duyệt coi là trang chéo

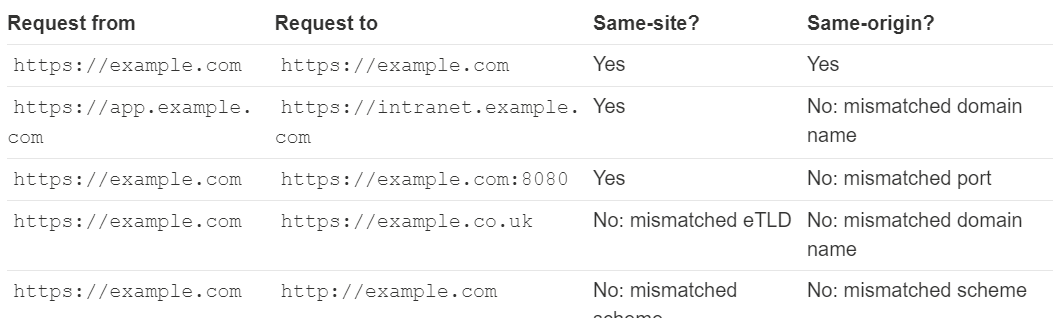


=> Khác **site = scheme + TLD + (TLD+1)** thì là **trang chéo**

Khác nhau giữa site và origin chính là **phạm vi:** Một site có nhiều domain names, một origin chỉ có một. Hai URL được coi là same origin nếu chúng chia sẻ chính xác cùng một scheme, domain và port.



=> cross-origin request can same-site.



SameSite: Mục đích của SameSite là **ngăn chặn trình duyệt truyền đi các cookie** do yêu cầu giữa các trang web. SameSite được thiết lập với ba cấp độ gồm Strict, Lax và None nhằm hạn chế việc truyền cookie từ mức độ nghiêm ngặt đến lỏng lẽo

+ **Strict:** Cấm hoàn toàn việc truy cập cookie giữa các trang web. Các trình duyệt không gửi cookie trong bất kỳ request cross-stie nào. Hiểu đơn giản là targer site không trùng với site hiện trên thành địa chỉ thì không gửi kèm cookie

+ **Lax:** Chỉ cho phép một số cookie tiến hành truy cập web khi có yêu cầu Get. Trong hầu hết các trường hợp thì truy cập cookie chéo trang vẫn bị cấm. Cho phép gửi cookie trong cross-site request nhưng chỉ khi đáp ứng hai điều kiện sau:

* **Sử dụng GET method** (Vì POST thường được dùng để thực hiện các hành động sửa đổi dữ liệu hoặc trạng thái => dễ trở thành mục tiêu của tấn công CSRF, script, load ảnh, iframe hoặc các nguồn khác đều không được bao gồm cookie)
* Yêu cầu xuất phát từ điều hướng cấp cao nhất của người dùng, ví dụ như ấn vào link

+ **None:** Cho phép tất cả cookie truy cập chéo trang web. Vô hiệu hóa hoàn toàn các hạn chế của SameSite, bất kể trình duyệt là gì => các trình duyệt sẽ gửi cookie này trong tất cả các request đến trang web đã phát hành nó, ngay cả những request được kích hoạt bởi các trang web bên thứ ba hoàn toàn không liên quan (Mặc định trừ Chrome)

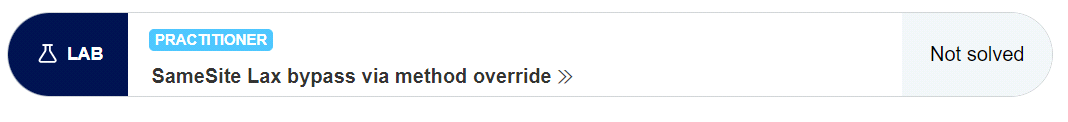
Khi đặt cookie với **SameSite=None**, trang web cũng phải bao gồm thuộc tính **Secure**, đảm bảo rằng cookie chỉ được gửi trong message được mã hóa qua HTTPS. Nếu không, các trình duyệt sẽ từ chối cookie và nó sẽ không được đặt.

Cách hoạt động của SameSite:

+ Trước khi cơ chế SameSite được giới thiệu, các trình duyệt đã gửi cookie trong mọi request đến miền đã cấp chúng, ngay cả khi yêu cầu được kích hoạt bởi một trang web bên thứ ba không liên quan.

+ SameSite hoạt động bằng cách cho phép các trình duyệt và chủ sở hữu trang web giới hạn cross-site request, nếu có nên bao gồm cookie cụ thể

**Bypassing SameSite Lax restrictions using GET request**



Nếu sử dụng **Lax** cho cookie, vẫn có thể thực hiện một cuộc tấn công CSRF bằng cách tạo ra một request GET từ máy nạn nhân.

Trong bài lab này, khi Set-Cookie trang web không chỉ định rõ ràng bất kỳ hạn chế SameSite nào nên trình duyệt sẽ sử dụng mức độ mặc định là **Lax** => method được chấp nhận để bao gồm cookie là GET, nhưng để thay đổi email ứng dụng chỉ chấp nhận method POST => phải ghi đề GET bằng POST.

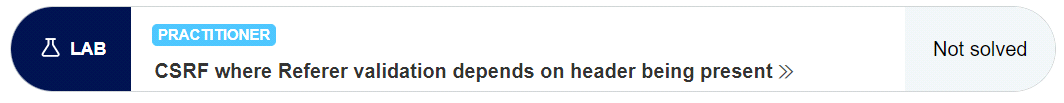
**\_method** parameter được sử dụng trong Symfony (framework của PHP) để ghi đè phương thức HTTP của yêu cầu

(Đây là một kỹ thuật phổ biến trong các ứng dụng web nơi form HTML chỉ cho phép gửi dữ liệu bằng cách sử dụng phương thức “GET" hoặc "POST", nhưng logic phía máy chủ yêu cầu một phương thức HTTP khác như "PUT" hoặc "DELETE"). Để sử dụng tham số “\_method”, form phải bao gồm một trường ẩn có name=”\_method” và value=”Phương thức muốn ghi đè”. Khi form được gửi trình duyệt sẽ gửi một request với phương thức được ghi đè.

Tham số “\_method” có thể gửi dưới dạng query parameter hoặc trong HTTP header tùy thuộc vào cách triển khai và cấu hình cụ thể. Trường ẩn vẫn là phổ biến nhất.

**Dạng 3: Bypassing Referer-based CSRF defense**

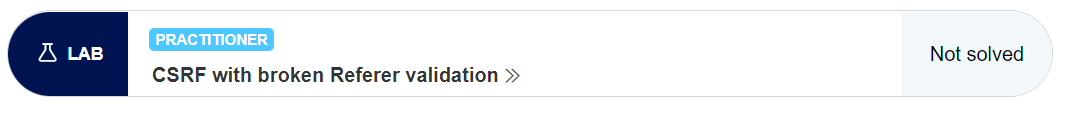
Ngoài các biện pháp bảo vệ sử dụng CSRF token, một số ứng dụng sử dụng HTTP header *Referer* để cố gắng bảo vệ trống lại các cuộc tấn công CSRF, thông thường bằng cách xác định rằng yêu cầu bắt nguồn từ miền riêng của ứng dụng.



**Một số ứng dụng xác thực Refere header khi nó xuất hiện trong các yêu cầu nhưng bỏ qua xác thực nếu Referer header bị bỏ qua. Kẻ tấn công có thể khai thác CSRF theo cách khiến trình duyệt của nạn nhân bỏ Referer header trong yêu cầu kết quả. Có nhiều cách để đạt được điều này nhưng cách dễ nhất là sử dụng thẻ META trong HTML tấn công CSRF**

*<meta name="referrer" content="never">*

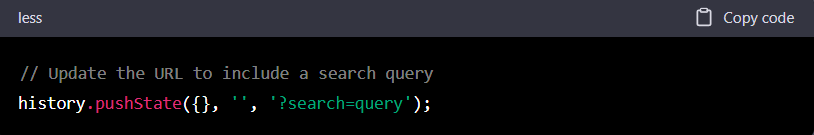
Một số ứng dụng sử dụng Referer header để cố gắng bảo vệ chống lại các cuộc tấn công CSRF bằng cách đảm bảo rằng yêu cầu bắt nguồn từ cùng một domain với website. Nếu yêu cầu đến từ bất kỳ nơi nào khác nó sẽ bị từ chối.



**Một số ứng dụng xác thực tiêu đề Referer theo cách ngây thớ có thể bỏ qua. Ví dụ nếu ứng dụng xác thực rằng domain trong Referer header bắt đầu với các giá trị dự kiến, thì kẻ tấn công có thể đặt domain này làm domain con của domain riêng của hacker. Hoặc nếu ứng dụng chỉ xác thực Referer header chứa tên miền của chính nó, thì kẻ tấn công có thể đặt giá trị được yêu cầu ở nơi khác trong URL.**

Trong bài lab này, ứng dụng không xác thực toàn bộ Referer header bằng với domain của ứng dụng thay vào đó chỉ xác thực rằng domain của ứng dụng được chứa trong Referer header.

**history.pushState:** Sửa đổi URL của một trang web mà không thực sự điều hướng đến một trang khác. Phương thức cập nhật URL trong thanh địa chỉ của trình duyệt, nhưng không khiến trình duyệt tải lại trang hoặc đưa ra request mới tới máy chủ.



Tham số đầu tiên là một đối tượng đại diện cho trạng thái của ứng dụng. Trạng thái này có thể được sử dụng để lưu trữ dữ liệu tồn tại ngay cả khi người dùng điều hướng khỏi trang rồi quay lại

Tham số thứ hai là tiêu đề

Tham số thứ ba là URL để cập nhật. URL có thể tương đối (như ví dụ) hoặc tuyệt đối.

**URL được cập nhật phải cùng domain với trang hiện tại.**

*Referrer-Policy: unsafe-url:* Sử dụng để kiểm soát thông tin được gửi trong Referer header (chứa URL của trang đã bắt đầu yêu cầu) của HTTP request.

*unsafe-url:* URL đầy đủ của trang web được gửi trong Referer header, bao gồm thông tin nhạy cảm ví dụ như query parameter.

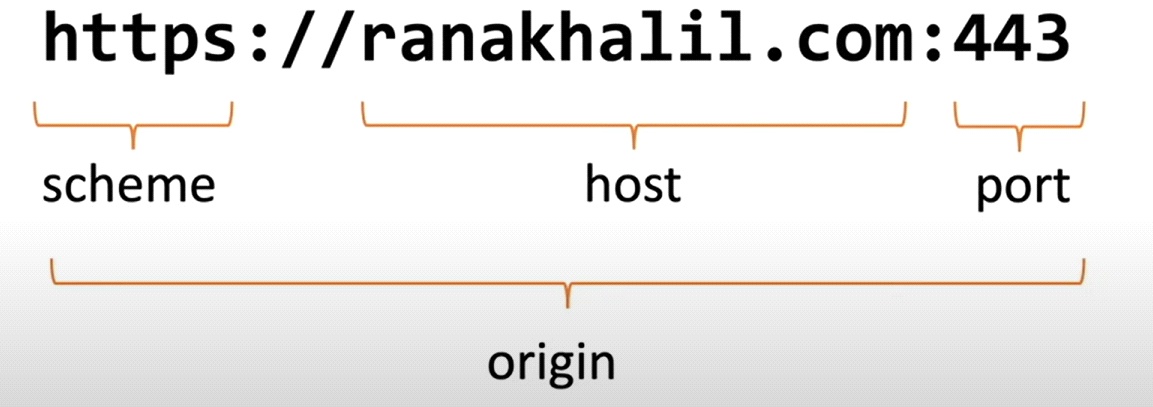
**WEB09: Cross-origin resource sharing (CORS) và Clickjacking**

**Cross-origin resource sharing (CORS)**

- Same-Origin Policy (SOP): Là cơ chế được các trình duyệt thực thi để kiểm soát dữ liệu giữa các ứng dụng web. SOP hạn chế các tập lệnh trên một origin truy cập dữ liệu từ một origin khác. Dùng SOP để không cho các domain khác có thể lấy được dữ liệu trên domain của mình. (**một cơ chế bảo vệ quan trọng hạn chế việc trao đổi dữ liệu giữa 2 origin khác nhau).**

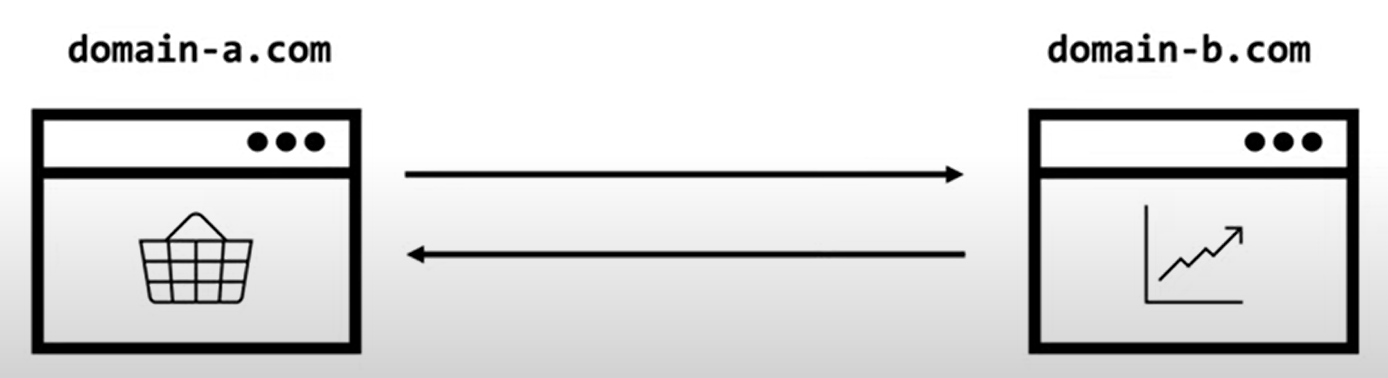
+ Không ngăn việc **ghi** giữa các ứng dụng web mà nó ngăn việc **đọc** giữa các ứng dụng web.

+ Quyền truy cập xác định dựa trên **origin: Hai URL được xác định cùng một origin nếu protocol (scheme), hostname(domain), port giống nhau.**



CORS: Là cơ chế cho phép các tài nguyên trên máy chủ được yêu cầu từ một tên miền khác, vì vậy trong một số trường hợp, có thể muốn nới lỏng chính sách same-origin và cho phép tương tác cross-origin.

Khá thường xuyên với các ứng dụng hiện đại. Nhiều trang web cần tương tác với tên miền phụ của chúng hoặc bên thứ ba và trong trường hợp này, không muốn same origin policy ngăn chặn sự tương tác đó vì vậy những gì các website làm là sử dụng Cross-origin resource sharing protocol

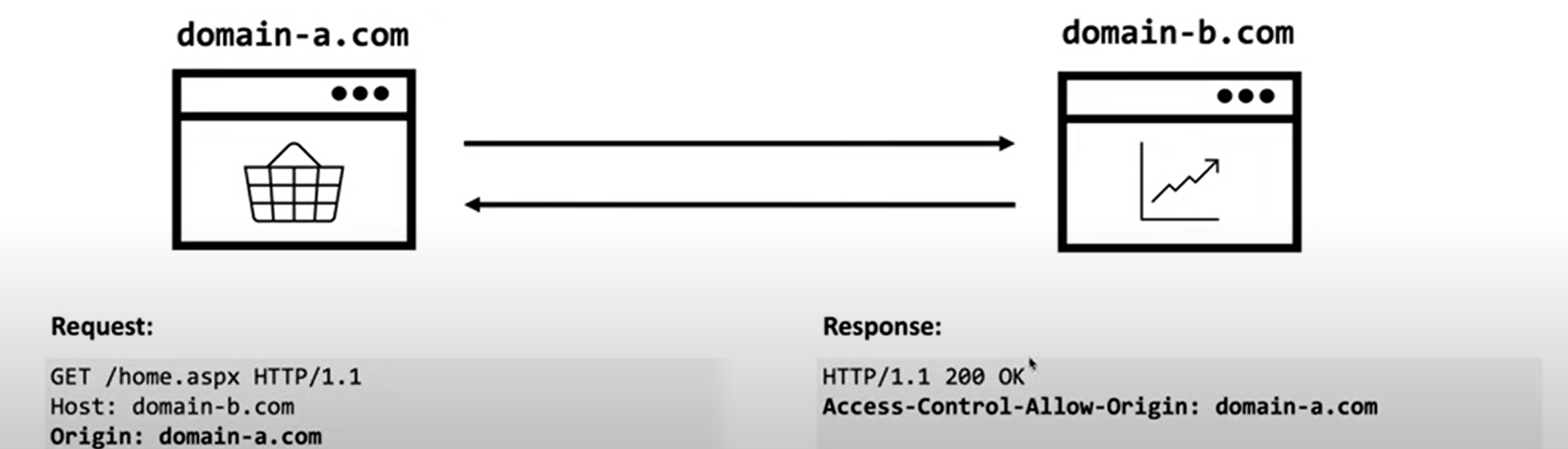


Nếu cấu hình CORS protocol ở phía domain-b thì yêu cầu từ domain-a sẽ được thực hiện.

**CORS là cơ chế sử dụng HTTP header để xác định origins mà trình duyệt cho phép tải tài nguyên.**

2 HTTP header sẽ bắt gặp khi thực hiện các request CORS:

+ Access-Control-Allow-Origin: Cho trình duyệt biết nếu origin được phép truy cập tài nguyên của một trang web cụ thể.



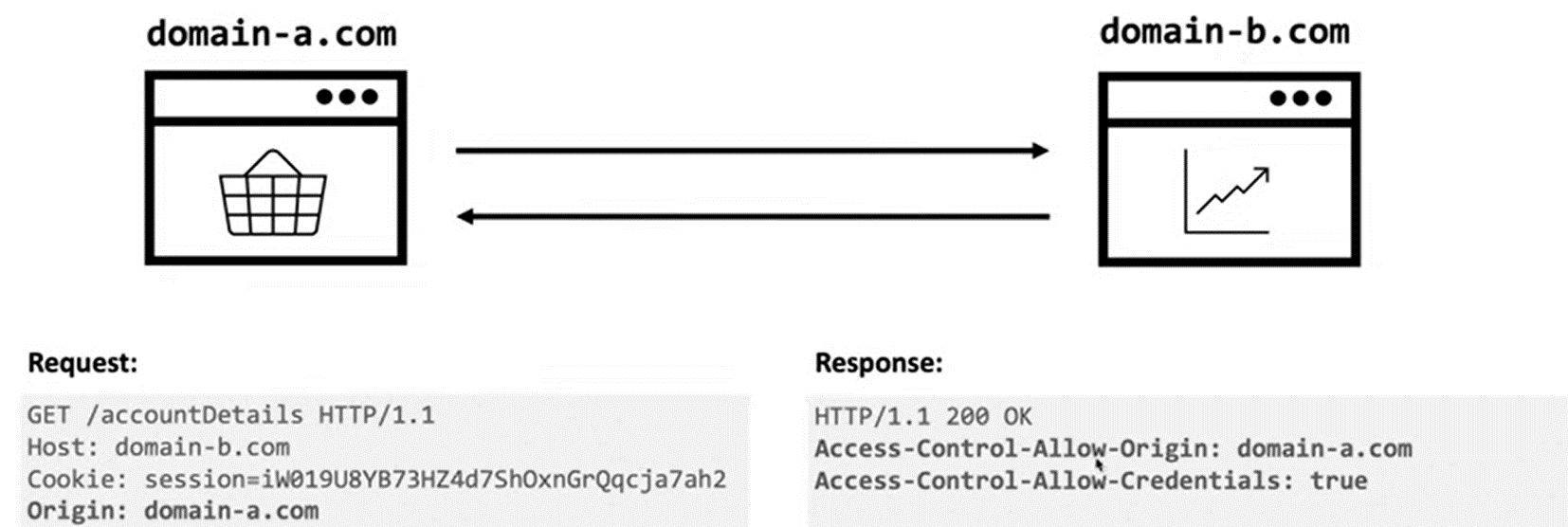
Các quy tắc sử dụng: Có 3 lựa chọn

* Access-Control-Allow-Origin: \*: Bất kỳ trang web nào cũng được truy cập vào tài nguyên
* Access-Control-Allow-Origin: <origin>: Cho phép một origin duy nhất truy cập vào tài nguyên. (single not multi)
* Access-Control-Allow-Origin: null

Cho phép truy cập vào các trang công khai

+ Access-Control-Allow-Credentials: Truy cập vào các trang web được xác thực.

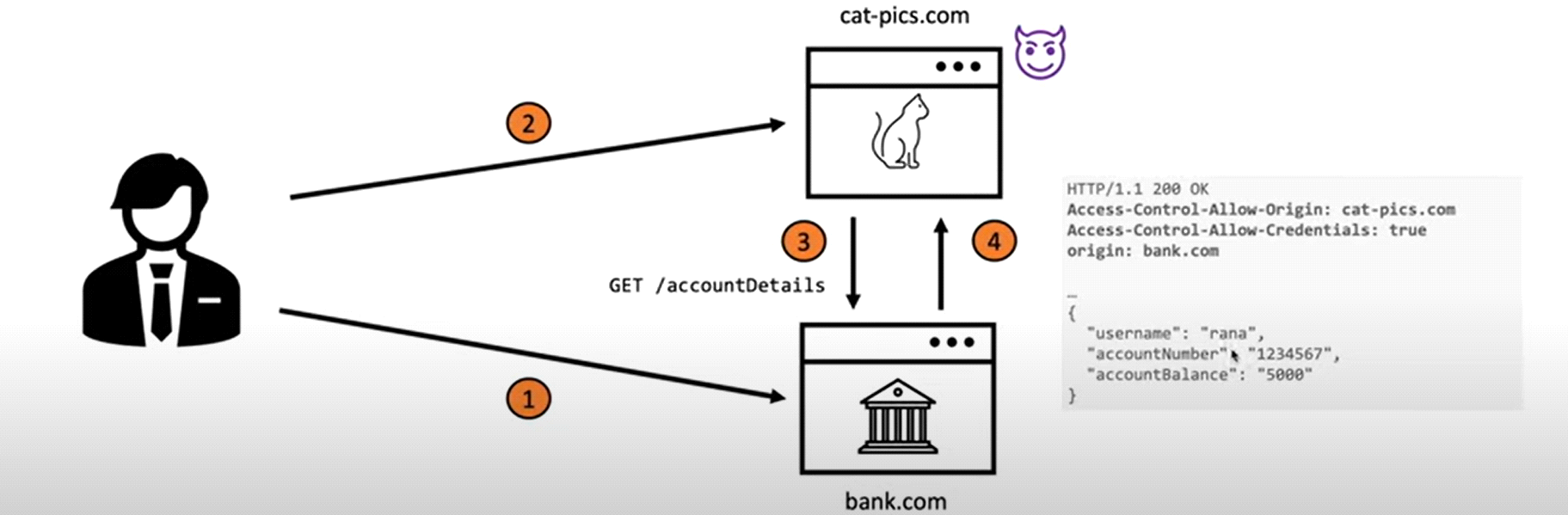
- Cho phép cookie (hoặc thông tin khác của user) được thêm vào cross-origin request



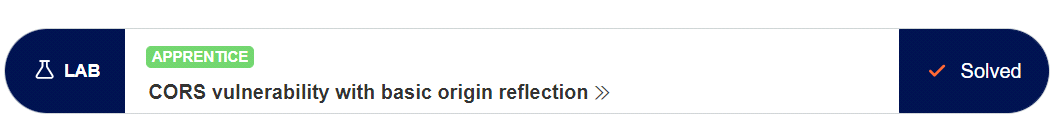
Access-Control-Allow-Credentials: true -> **được phép chuyển thông tin xác thực trong request** và do đó domain-a có thể truy cập tài nguyên được xác thực từ domain-b.

Note: Nếu Access-Control-Allow-Origin: \* thì Access-Control-Allow-Credentials không được đặt.

**CORS vulnerabilities**



- Xảy ra do các vấn đề về cấu hình.



**WEB10: DOM-based vulnerabilities and Websockets**

**1. DOM-based vulnerabilities**

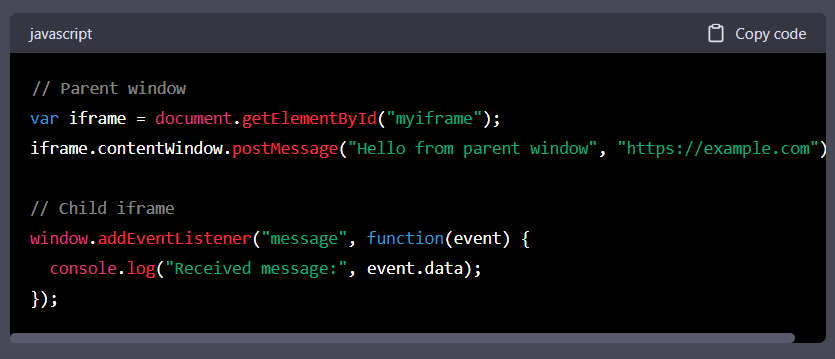
Dạng 1: Controlling the web message source

**postMessage method:**

**+** Cho phép giao tiếp giữa các miền khác nhau một cách an toàn. Nó cho phép giao tiếp giữa **parent window** và **child iframe** hoặc giữa các cửa sổ riêng biệt.

+ Có thể sử dụng để gửi tin nhắn từ cửa sổ này sang cửa sổ khác. Message có thể là bất kỳ loại dữ liệu nào: chuỗi, đối tượng, mảng và có thể được cửa số đích nhận bằng sự kiện `**message**`.

Ví dụ gửi message từ parent đến child



**Thằng gửi.postMessage(message, thằng nhận)**



**KHÁI NIỆM, CÁCH TẤN CÔNG VÀ CÁCH PHÒNG THỦ**

**1. XSS**

**- Khái niệm:** Là một loại vấn đề tiêm (injection), trong đó các script độc hại được tiêm vào các trang web đáng tin cậy. XSS xảy ra khi một kẻ tấn công sử dụng ứng dụng web để gửi mã độc hại (thường tiêm vào link hoặc mã nguồn của một trình duyệt) tới một người dùng khác.

- **Cách tấn công:**

**+ Stored XSS:** Kẻ tấn công gửi script độc hại lên ứng dụng thông qua HTTP request để script độc hại được lưu trên server máy chủ ví dụ DB

+ **Reflected XSS:** Lừa người dùng ấn vào đường link

+ **DOM XSS:** Lừa người dùng ấn vào đường link

- **Cách phòng thủ:**

**+** Lọc dữ liệu đầu vào

+ Mã hóa dữ liệu đầu ra

+ Lập whitelist

+ Sử dụng response header phù hợp

+ Sử dụng CSP

**2. CSRF**

**- Khái niệm:** là một loại lỗ hổng cho phép kẻ tấn công gửi các yêu cầu độc hại từ một trang web bị xâm nhập đến một trang web khác nơi nạn nhân được xác thực. Những kẻ tấn công lừa nạn nhân gửi yêu cầu độc hại, thường bằng cách sử dụng biểu mẫu hoặc hình ảnh giả mạo, để đánh cắp thông tin nhạy cảm hoặc thực hiện các hành động trái phép.

- **Cách tấn công:**

**+** Kẻ tấn công sử dụng XSS để đưa mã độc vào một trang web và đánh cắp mã thông báo phiên của người dùng**.**

**+** Chuyển hướng URL: Những kẻ tấn công lừa người dùng truy cập một trang web độc hại gửi yêu cầu giả mạo đến trang web mục tiêu.

+ Clickjacking: Những kẻ tấn công sử dụng một kỹ thuật để phủ một trang web độc hại lên một trang web hợp pháp và lừa người dùng nhấp vào liên kết độc hại.

**- Cách phòng chống**

**+** Sử dụng csrf token

+ Sử dụng các hạn chế cookie samesite nghiêm ngặt

+ Sử dụng Referer HTML header

**3. CORS**

**- Khái niệm:** Đề cập đến các điểm yếu bảo mật trong việc triển khai chính sách CORS của ứng dụng web. Nó có thể xảy ra khi máy chủ không xác thực đúng nguồn gốc của các yêu cầu đến, cho phép kẻ tấn công truy cập hoặc thao tác dữ liệu nhạy cảm trên các miền. Mức độ nghiêm trọng của lỗ hổng phụ thuộc vào bản chất của dữ liệu nhạy cảm và mức độ truy cập được cấp bởi chính sách CORS.

- **Cách tấn công**

**- Cách phòng thủ:** Các lỗ hổng CORS phát sinh chủ yếu do cấu hình sai. Một số biện pháp phòng thủ hiệu quả chống lại các cuộc tấn công CORS

+ Cấu hình đúng các cross-site request

+ Chỉ cho phép các trang web đáng tin cậy

+ Tránh sử dụng tiêu đề Access-Control-Allow-Origin: null

+ Tránh các ký tự đại diện trong mạng nội bộ

**4. Clickjacking**

**- Khái niệm:** là một loại lỗ hổng bảo mật xảy ra khi một trang web độc hại phủ một lớp trong suốt hoặc vô hình lên một trang web hợp pháp, lừa người dùng nhấp vào các phần tử trong lớp độc hại.

- **Cách phòng chống: X-Frame-Options, CSP**

**5. DOM-based**

**- Khái niệm:** Lỗ hổng DOM-based phát sinh khi một trang web chứa js lấy giá trị có thể kiểm soát của kẻ tấn công (source) và chuyển giá trị đó vào một chức năng nguy hiểm (sink) => Lấy source ném vào sink.

- **Cách phòng chống:** Cách hiệu quả nhất để tránh các lỗ hổng dựa trên DOM là tránh cho phép dữ liệu từ bất kỳ nguồn không đáng tin cậy nào tự động thay đổi giá trị được truyền tới bất kỳ sink nào**.**