**Lab 15: Cross-site scripting - Stored**

**Objective**:

* This lab is focused on Stored Cross-Site Scripting (XSS) vulnerabilities, a type of XSS where malicious scripts are permanently stored on the target server, such as in a database, and then rendered in a user's browser when the stored data is displayed. Participants will interact with a simulated web application that is vulnerable to stored XSS attacks.

In this lab, students need to:

* Answer the following questions:
  + What defines Stored Cross-Site Scripting (XSS), and how does it differ from Reflected XSS and DOM-Based XSS?
  + Explain the process of how malicious scripts are injected, stored, and subsequently executed in Stored XSS attacks, and discuss the potential consequences these attacks can have on both users and web applications.
* Perform challenge:
  + [Stored DOM XSS](https://portswigger.net/web-security/cross-site-scripting/dom-based/lab-dom-xss-stored)
* Explain and capture all steps (full windows screen capture).

Submit a report addressing all the questions mentioned above in either **PDF** or **Markdown** format. Additionally, include a **video** demonstrating the detailed process of your work to ensure the authenticity of your lab exercise.

**Lab 15: Cross-site scripting – Stored**

**What defines Stored Cross-Site Scripting (XSS), and how does it differ from Reflected XSS and DOM-Based XSS?**

Stored Cross-Site Scripting (XSS) is a type of XSS attack where the malicious script is permanently stored on the server-side and then served to multiple users when they request a particular page or resource. Unlike Reflected XSS, where the payload is immediately reflected back to the user in the server's response, and DOM-Based XSS, where the attack occurs entirely on the client-side, Stored XSS involves the injection and storage of the malicious script within the application's data.

Here's how Stored XSS differs from Reflected XSS and DOM-Based XSS:

1. \*\*Reflected XSS\*\*: In Reflected XSS, the injected script is reflected back to the user by the web server without being stored on the server-side. The attack typically involves injecting the payload into input fields or URL parameters, which are then included in the server's response. The malicious script is executed in the victim's browser when they interact with the compromised web page. Reflected XSS attacks are often limited to the context of a single request-response cycle.

2. \*\*DOM-Based XSS\*\*: DOM-Based XSS attacks exploit vulnerabilities in the Document Object Model (DOM) of a web page and occur entirely on the client-side. Unlike Stored and Reflected XSS, where the injection and execution of the payload involve interactions with the server, DOM-Based XSS attacks manipulate the client-side DOM directly. The attacker injects the payload into the page's HTML or JavaScript, and the script is executed in the victim's browser based on how the page's JavaScript code processes the injected data.

3. \*\*Stored XSS\*\*: Stored XSS involves injecting the malicious script into the application's database or other storage mechanisms. This payload is then permanently stored on the server-side and served to multiple users when they access the compromised page or resource. The injected script executes in the context of the victim's session when the page containing the stored payload is loaded. Stored XSS attacks can have widespread impact as they affect all users who visit the affected page, and the payload persists over time until it is discovered and removed.

In summary, while all three types of XSS attacks involve injecting and executing malicious scripts within web applications, they differ in how the payload is injected, stored, and executed, as well as the impact they can have on users and the application's security.

**Explain the process of how malicious scripts are injected, stored, and subsequently executed in Stored XSS attacks, and discuss the potential consequences these attacks can have on both users and web applications.**

In Stored Cross-Site Scripting (XSS) attacks, malicious scripts are injected into a vulnerable web application and stored on the server-side, typically in a database or other data storage mechanism. These scripts are then served to multiple users when they access the compromised page or resource. Here's how the process typically unfolds:

1. \*\*Injection\*\*: The attacker identifies a vulnerable input field or parameter within the web application where user-supplied data is not properly sanitized or validated before being stored on the server. This could be a comment form, a message board, a user profile, or any other feature that allows users to input and store data.

2. \*\*Payload Crafting\*\*: The attacker crafts a malicious payload, often JavaScript code, that they want to execute in the browsers of unsuspecting users. This payload may include actions such as stealing session cookies, redirecting users to phishing websites, defacing the web page, or performing other malicious activities.

3. \*\*Injection and Storage\*\*: The attacker submits the crafted payload through the vulnerable input field, such as a comment box. The application accepts the input without proper validation or sanitization and stores it in the server's database alongside legitimate user data.

4. \*\*Serving the Malicious Content\*\*: When other users visit the page where the injected payload is stored, the server retrieves the stored data from the database and serves it to their browsers as part of the web page's content. Because the payload is included in the server's response, it is executed within the context of the victim's browser.

5. \*\*Execution in Victims' Browsers\*\*: The victim's browser interprets and executes the injected script as part of the web page's content. This allows the attacker to achieve their malicious objectives, such as stealing sensitive information or performing unauthorized actions on behalf of the victim.

The consequences of Stored XSS attacks can be severe for both users and web applications:

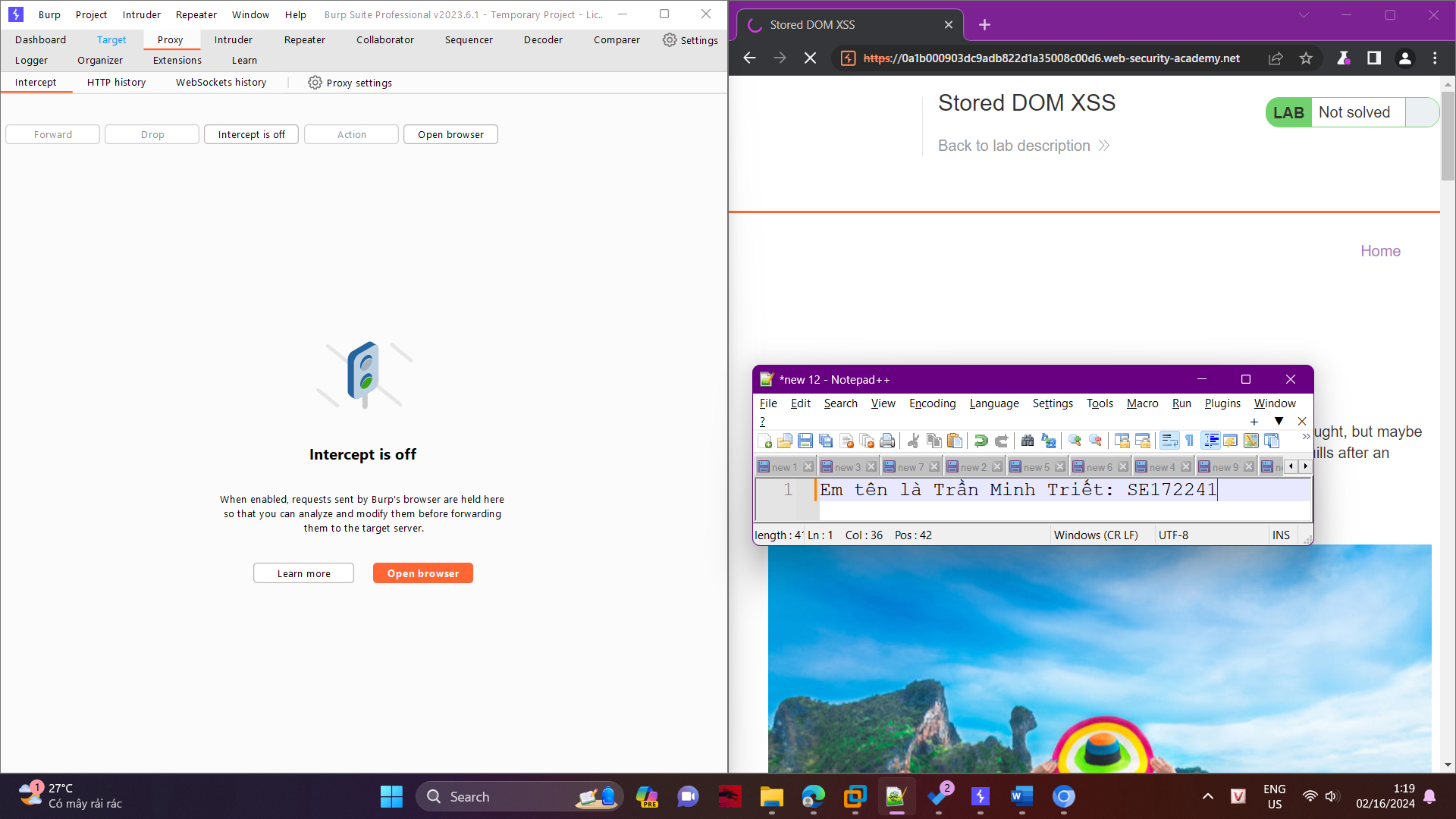
- \*\*User Privacy and Security\*\*: Stored XSS attacks can compromise the privacy and security of users by allowing attackers to steal their sensitive information, such as session cookies, login credentials, or personal data. Attackers may also use the compromised session to perform unauthorized actions on behalf of the user, such as making purchases, posting malicious content, or accessing restricted areas of the application.

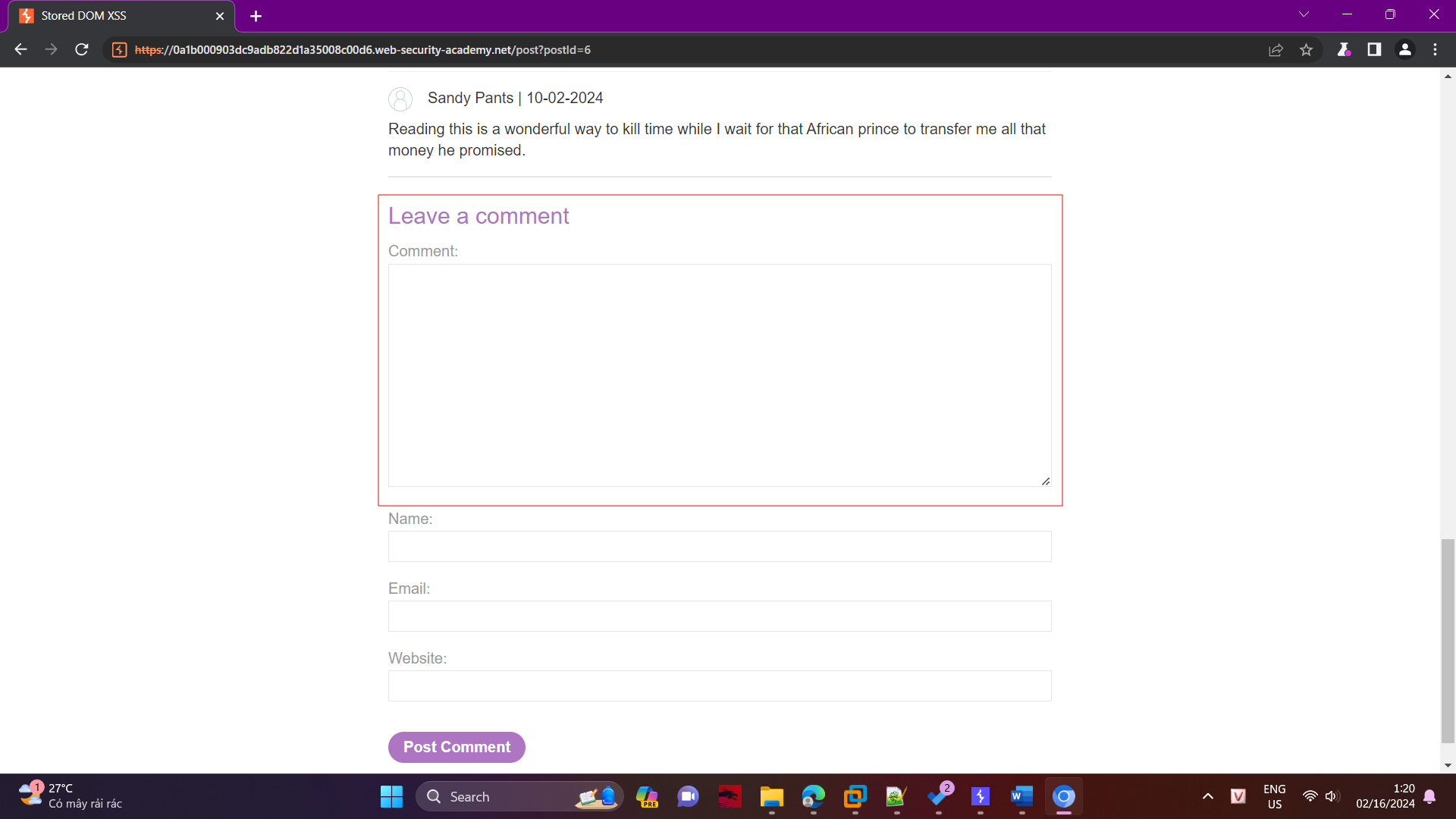
- \*\*Reputation Damage\*\*: Web applications that are vulnerable to Stored XSS attacks risk damaging their reputation and losing user trust. Users may become wary of using the application if they perceive it as insecure or prone to exploitation by malicious actors.

- \*\*Legal and Compliance Issues\*\*: Organizations that fail to adequately protect against Stored XSS attacks may face legal and regulatory consequences, especially if the compromised data includes personally identifiable information (PII) or falls under data protection laws such as the General Data Protection Regulation (GDPR) or the California Consumer Privacy Act (CCPA).

To mitigate the risks associated with Stored XSS attacks, web applications should implement robust input validation and output encoding practices, sanitize user input, and employ security mechanisms such as Content Security Policy (CSP) to prevent the execution of unauthorized scripts. Regular security audits and vulnerability assessments can also help identify and remediate XSS vulnerabilities before they are exploited by attackers.

**Challenge**

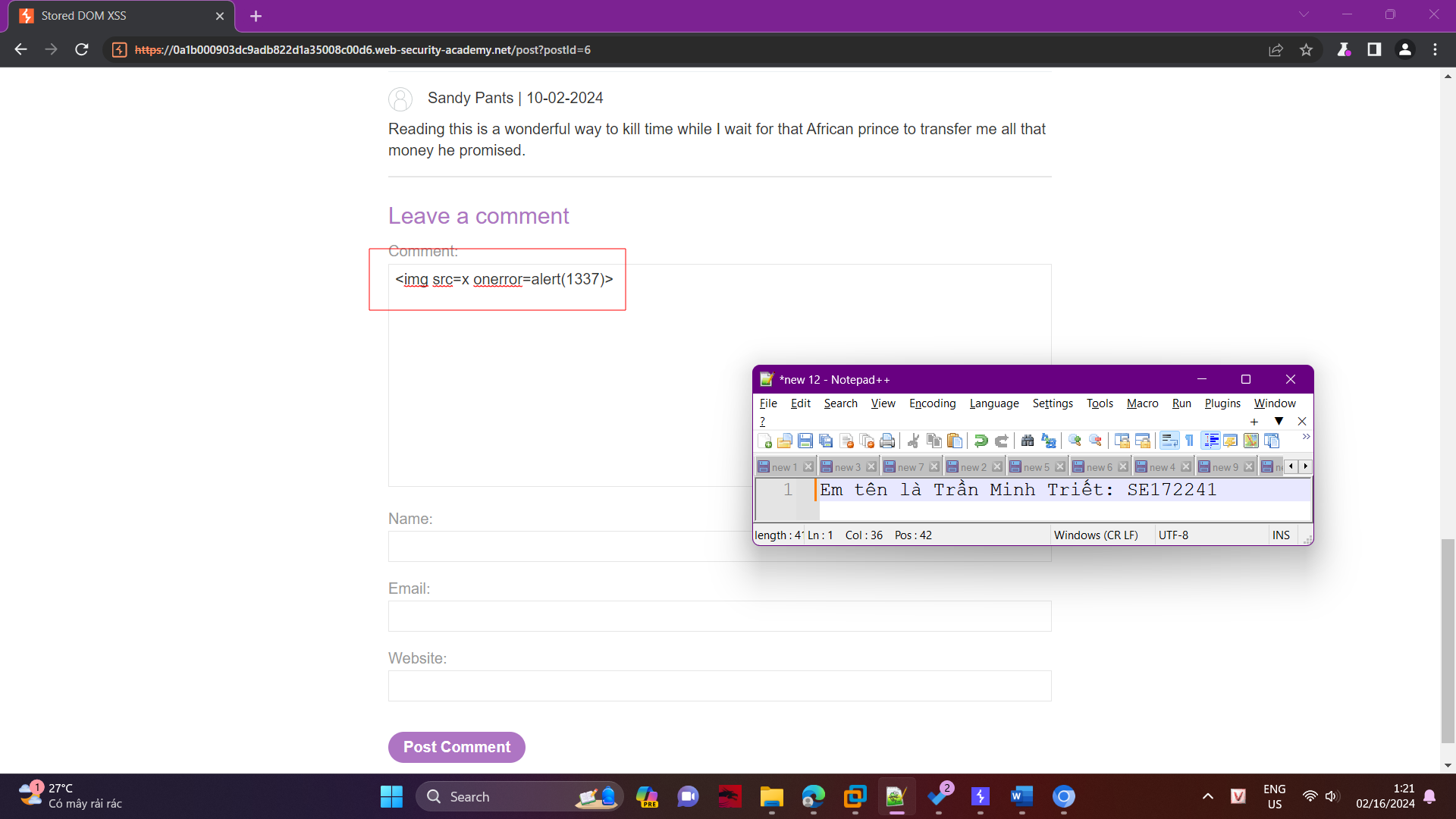


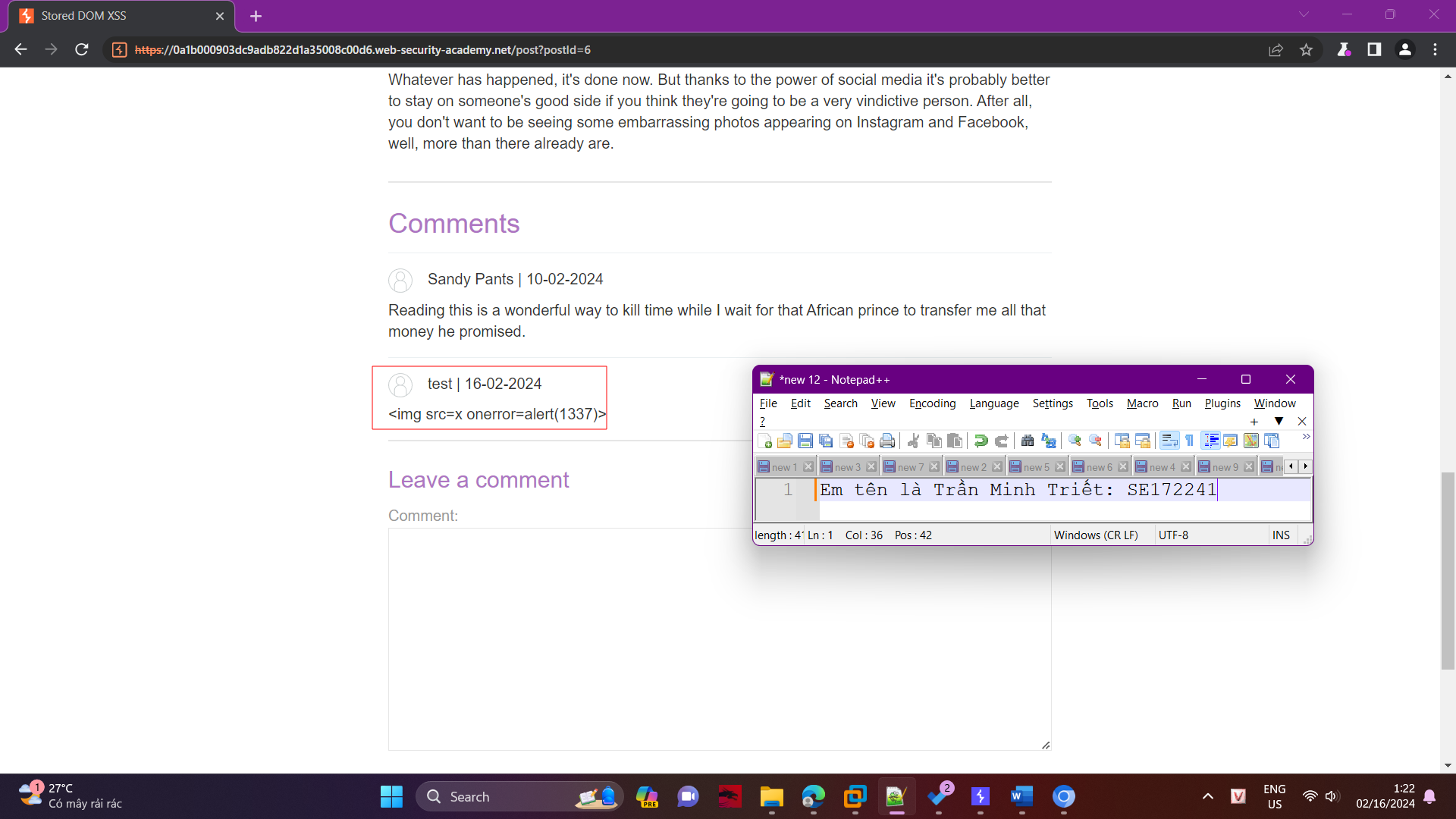


This lab is vulnerable to stored XSS in post page in the comment section. When I tried this payload in the comment input field.

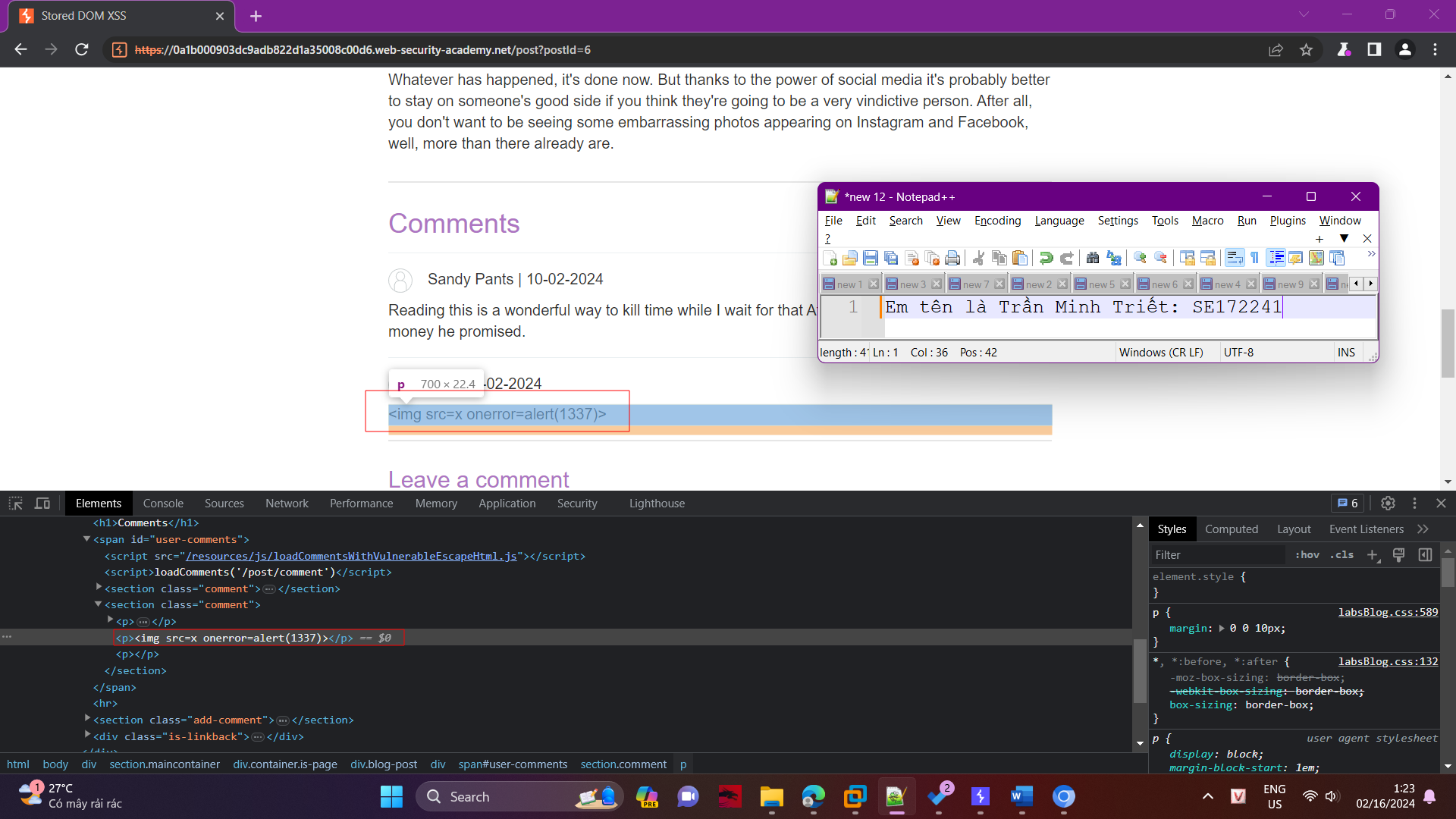
<img src=x onerror=alert(1337)>

it has been printed out as its and nothing happens!





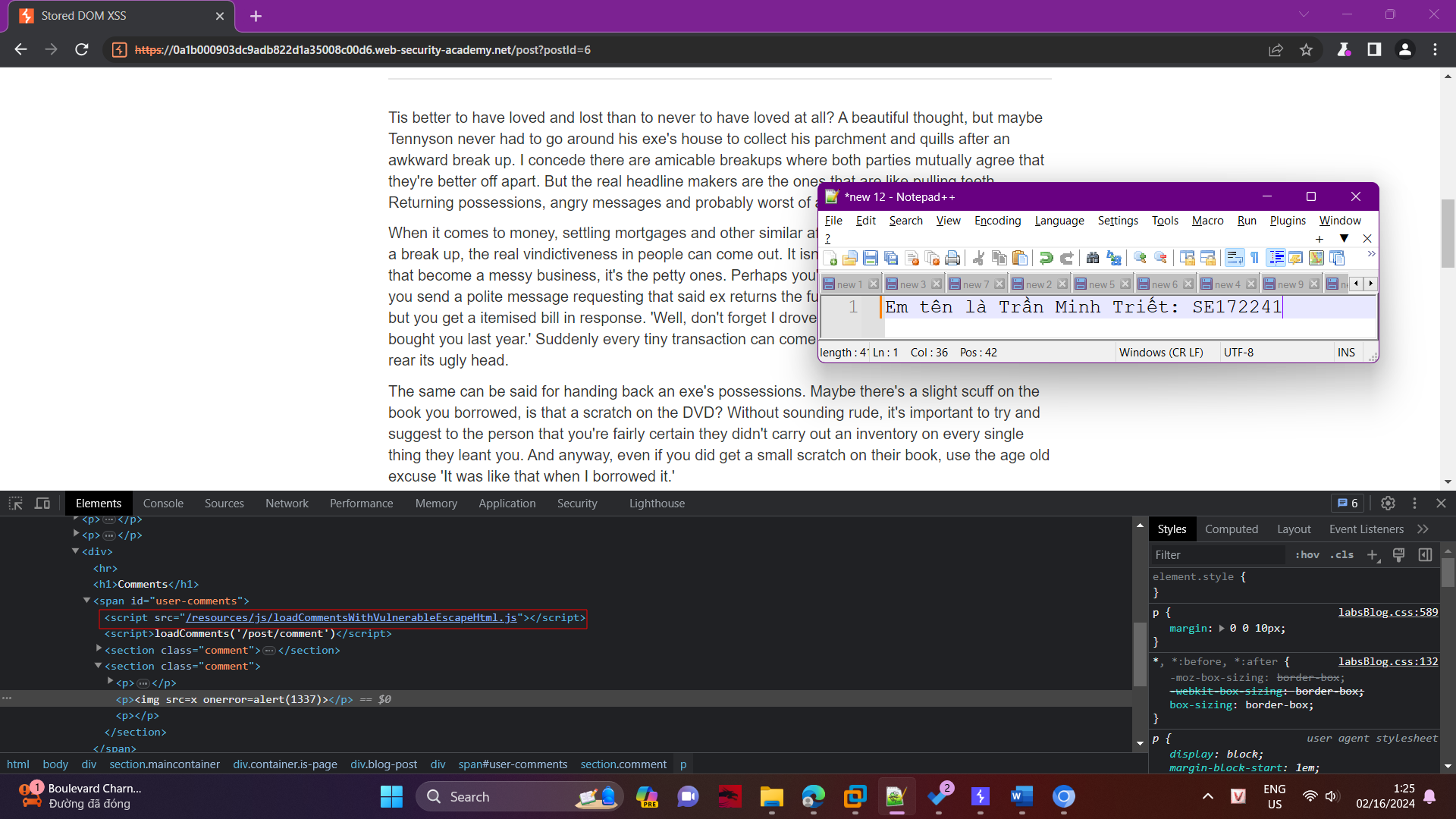
I clicked inspect element, then in the inspect page below with right click above the paragraph tag that holds the message I have copied the element

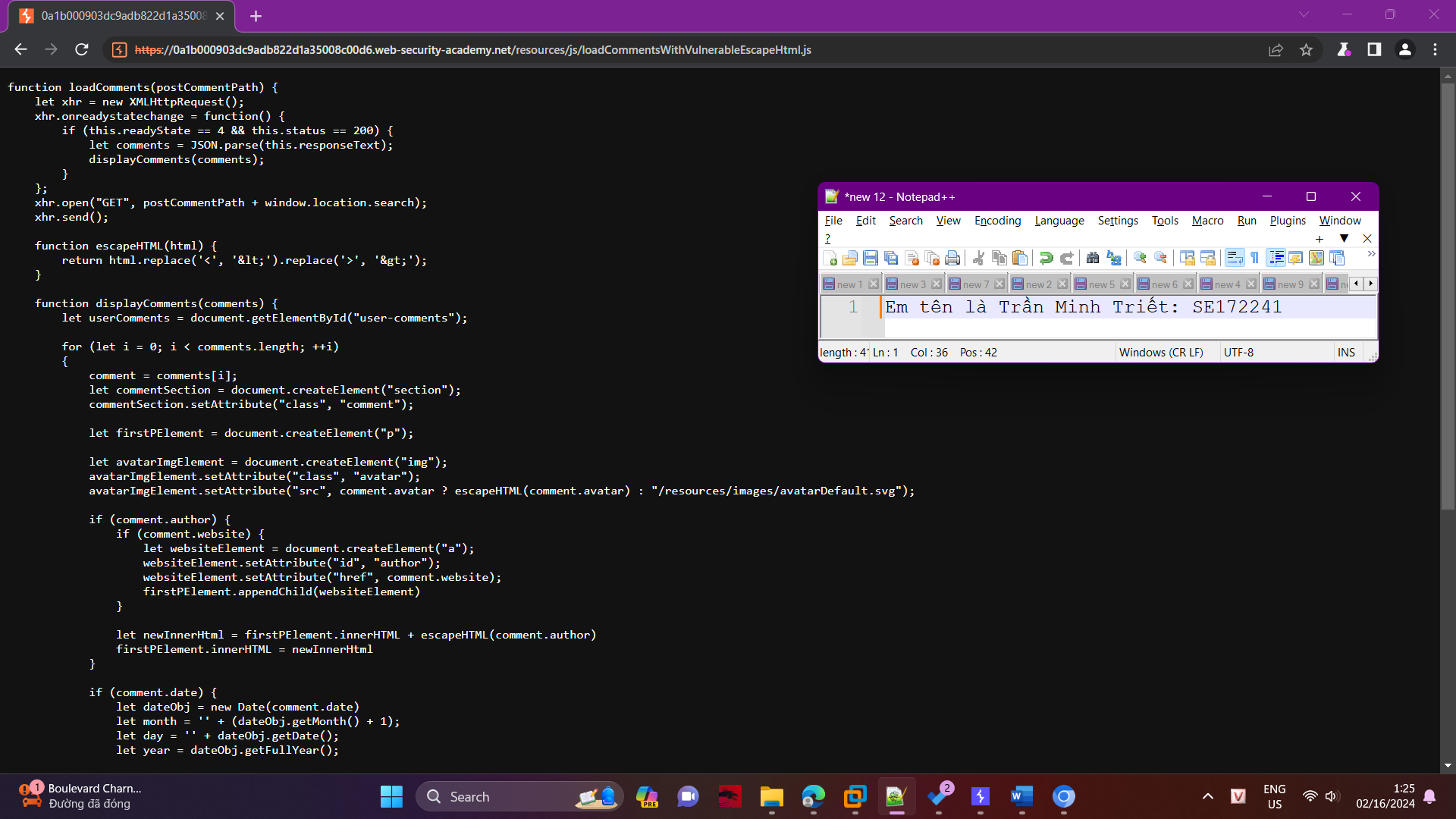


I found that the payload are shown with html encoding for the starting and closing tag which prevent these characters from being interpreted as HTML tags or entities by the web browser and to display them as regular text on the web page.

<p>&lt;img src=x onerror=alert(1337)&gt;</p>

I clicked view page source in the lab, and I have found a script file that are responsible for handling the validations for the comments that have been loaded from backend





After checking the content of this file, I found this logic for validating the comment body message



The developer sends the comment body to this function before appending it to a paragraph tag in the DOM, but this code has a BUG!

The replace() built-in function in JavaScript does replace all occurrences if you use the 'g' flag in the regular expression but the developer miss it, so it will replace the first occurrence only for the starting and the closing tag

So we can bypass this validation with an extra tags that will be escaped, then we will execute the alert function using this payload.

<img src=y><img src=x onerror=alert(1337)>

So finally we got the alert :)

