**Lab 19: File upload vulnerabilities**

**Objective**:

* This lab is designed to delve into File Upload vulnerabilities, a type of security issue where an application allows users to upload files that can be maliciously crafted to exploit the system. Participants will interact with a simulated web application that contains vulnerabilities in its file upload mechanism. The goal is to understand how to identify, exploit, and mitigate file upload vulnerabilities, focusing on the risks they pose to web applications and servers.

In this lab, students need to:

* Answer the following questions:
  + What are File Upload vulnerabilities, and how do they present a risk to web applications and servers? Discuss the potential consequences of improperly handled file uploads, including the execution of malicious code, system compromise, and data breaches. Explain how these vulnerabilities differ from other input-related security issues.
  + Describe the process of exploiting a File Upload vulnerability in a web application. What types of files and content might an attacker use to exploit such vulnerabilities, and how can they bypass common security checks?
* Perform challenge:
  + [Web shell upload via obfuscated file extension](https://portswigger.net/web-security/file-upload/lab-file-upload-web-shell-upload-via-obfuscated-file-extension)
* 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 19: File upload vulnerabilities**

**What are File Upload vulnerabilities, and how do they present a risk to web applications and servers? Discuss the potential consequences of improperly handled file uploads, including the execution of malicious code, system compromise, and data breaches. Explain how these vulnerabilities differ from other input-related security issues.**

File Upload vulnerabilities are security flaws in web applications that arise when user-uploaded files are not properly validated, sanitized, or handled. These vulnerabilities can pose significant risks to web applications and servers, as they can allow attackers to upload and execute malicious files, compromise the underlying system, or steal sensitive data.

Here's an overview of File Upload vulnerabilities and their risks:

1. \*\*Improper Validation\*\*: File Upload vulnerabilities occur when web applications fail to properly validate the type, size, or content of uploaded files. Attackers can exploit these vulnerabilities by uploading files containing malicious code or executable payloads.

2. \*\*Execution of Malicious Code\*\*: If an attacker successfully uploads a file containing malicious code, such as PHP, JavaScript, or other scripting languages, they may be able to execute arbitrary commands on the server or inject malicious code into the application.

3. \*\*System Compromise\*\*: Exploiting File Upload vulnerabilities can lead to the compromise of the underlying server or system. Attackers may upload files that exploit vulnerabilities in server-side software, escalate privileges, or gain unauthorized access to sensitive files or resources.

4. \*\*Data Breaches\*\*: In addition to executing malicious code or compromising the system, attackers may also use File Upload vulnerabilities to steal sensitive data. For example, attackers may upload files containing malware or backdoors to exfiltrate sensitive information from the server or infiltrate the internal network.

5. \*\*Difference from Other Input-Related Issues\*\*: File Upload vulnerabilities differ from other input-related security issues, such as Cross-Site Scripting (XSS) or SQL Injection (SQLi), in that they involve the handling of files rather than textual or structured data. While XSS and SQLi vulnerabilities may allow attackers to manipulate or inject code into web pages or database queries, File Upload vulnerabilities specifically relate to the uploading and processing of files within web applications.

To mitigate the risks associated with File Upload vulnerabilities, developers should implement the following best practices:

- \*\*File Type Validation\*\*: Validate the type of uploaded files to ensure that only allowed file types are accepted. This can help prevent the execution of malicious scripts or executable files.

- \*\*File Size Limitation\*\*: Implement file size limitations to prevent the upload of excessively large files, which could consume server resources or lead to denial-of-service conditions.

- \*\*Content-Type Headers\*\*: Verify the Content-Type headers of uploaded files to ensure they match the expected file type. Attackers may attempt to bypass file type validation by manipulating these headers.

- \*\*Secure File Storage\*\*: Store uploaded files in a secure location outside of the web root directory to prevent direct access by attackers. Additionally, consider implementing access controls and encryption to protect sensitive files.

- \*\*Regular Security Audits\*\*: Conduct regular security audits and vulnerability assessments to identify and remediate File Upload vulnerabilities before they are exploited by attackers.

By following these best practices, organizations can reduce the risk of File Upload vulnerabilities and protect their web applications and servers from potential exploitation and compromise.

**Describe the process of exploiting a File Upload vulnerability in a web application. What types of files and content might an attacker use to exploit such vulnerabilities, and how can they bypass common security checks?**

Exploiting a File Upload vulnerability in a web application involves manipulating the file upload functionality to upload and execute malicious files. Here's a general process an attacker might follow to exploit a File Upload vulnerability:

1. \*\*Identify the Vulnerability\*\*: The attacker first needs to identify a vulnerable file upload functionality within the web application. This could include file upload forms, profile picture uploads, document uploads, or any other feature that allows users to upload files.

2. \*\*Understand File Handling\*\*: The attacker needs to understand how the uploaded files are handled by the application. This includes understanding any client-side or server-side validation, file type checks, and file storage mechanisms employed by the application.

3. \*\*Craft Malicious File\*\*: The attacker crafts a malicious file containing code or payloads that they want to execute on the server. This could include malicious scripts, backdoor shells, executable files, or any other file type capable of executing code.

4. \*\*Manipulate File Extension\*\*: The attacker may attempt to bypass file type checks by manipulating the file extension or content type headers. For example, they may rename a PHP script to have a .jpg extension to bypass checks that only allow image files to be uploaded.

5. \*\*Upload the Malicious File\*\*: The attacker uploads the malicious file using the vulnerable file upload functionality. They may use tools or scripts to automate this process and test different file types and configurations.

6. \*\*Execute the Payload\*\*: If the File Upload vulnerability is successfully exploited, the attacker's malicious file will be stored on the server. They can then trigger the execution of the payload by accessing the uploaded file through the web application or by exploiting other vulnerabilities to execute the file.

Types of files and content that an attacker might use to exploit File Upload vulnerabilities include:

- \*\*Malicious Scripts\*\*: Attackers may upload files containing malicious JavaScript, PHP, or other scripting languages to execute arbitrary code on the server or in the client's browser.

- \*\*Executable Files\*\*: Attackers may upload executable files, such as binary executables or shell scripts, to execute commands on the server or compromise the underlying system.

- \*\*Backdoor Shells\*\*: Attackers may upload backdoor shells or web shells that provide remote access to the server, allowing them to perform further actions, such as reconnaissance, data theft, or further exploitation.

- \*\*Malware\*\*: Attackers may upload malware or viruses disguised as legitimate files to compromise the security of the server or infect other users who download the files.

To bypass common security checks implemented by web applications, attackers may employ various techniques, including:

- \*\*File Extension Manipulation\*\*: Changing the file extension or content type headers to trick the application into accepting files with prohibited or unexpected file types.

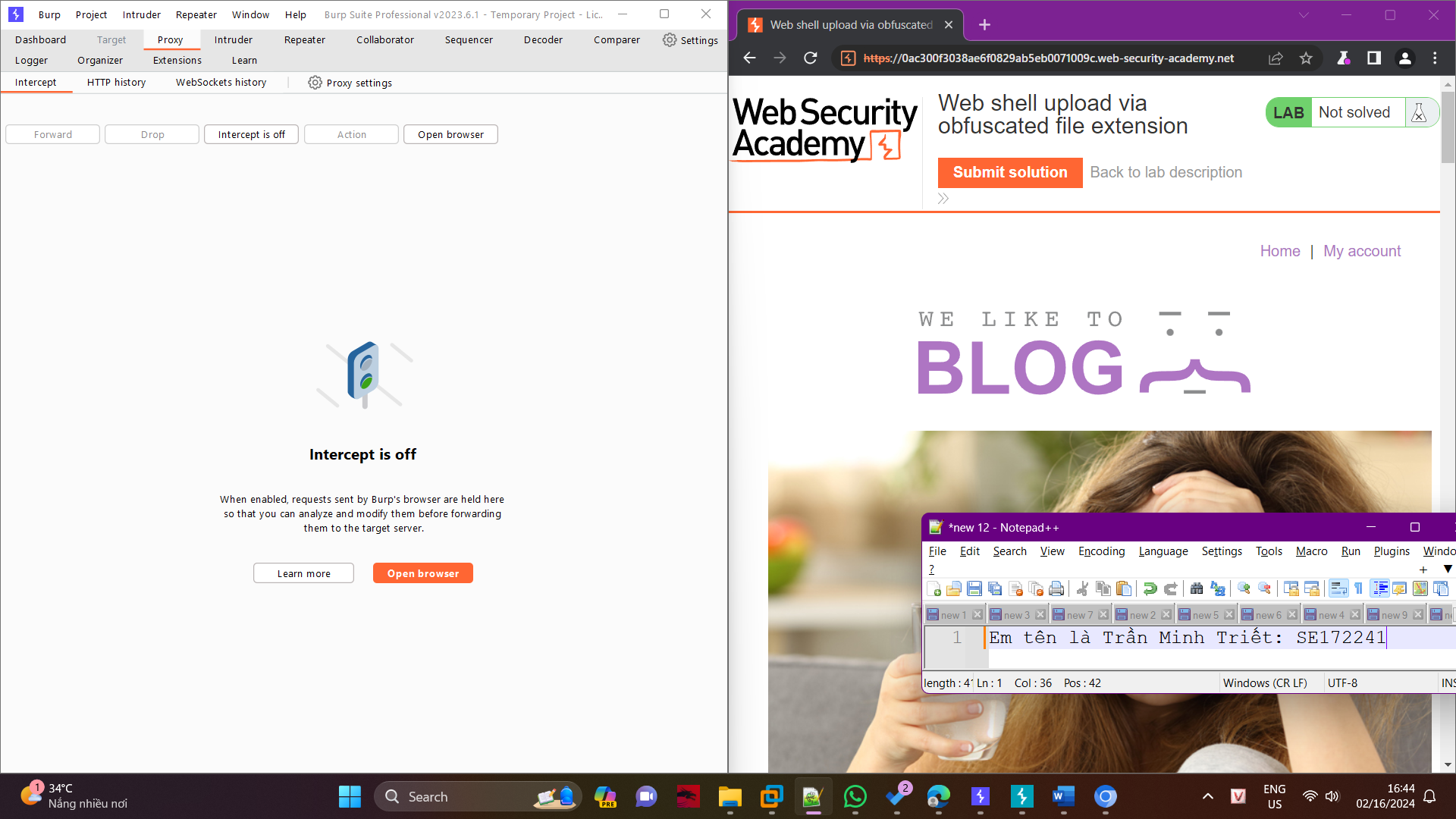
- \*\*Double Extensions\*\*: Using double file extensions, such as "file.php.jpg," to bypass checks that only look at the last extension.

- \*\*Null Bytes\*\*: Adding null bytes (\x00) or other encoding techniques to manipulate file names and bypass checks that use string-based file type detection.

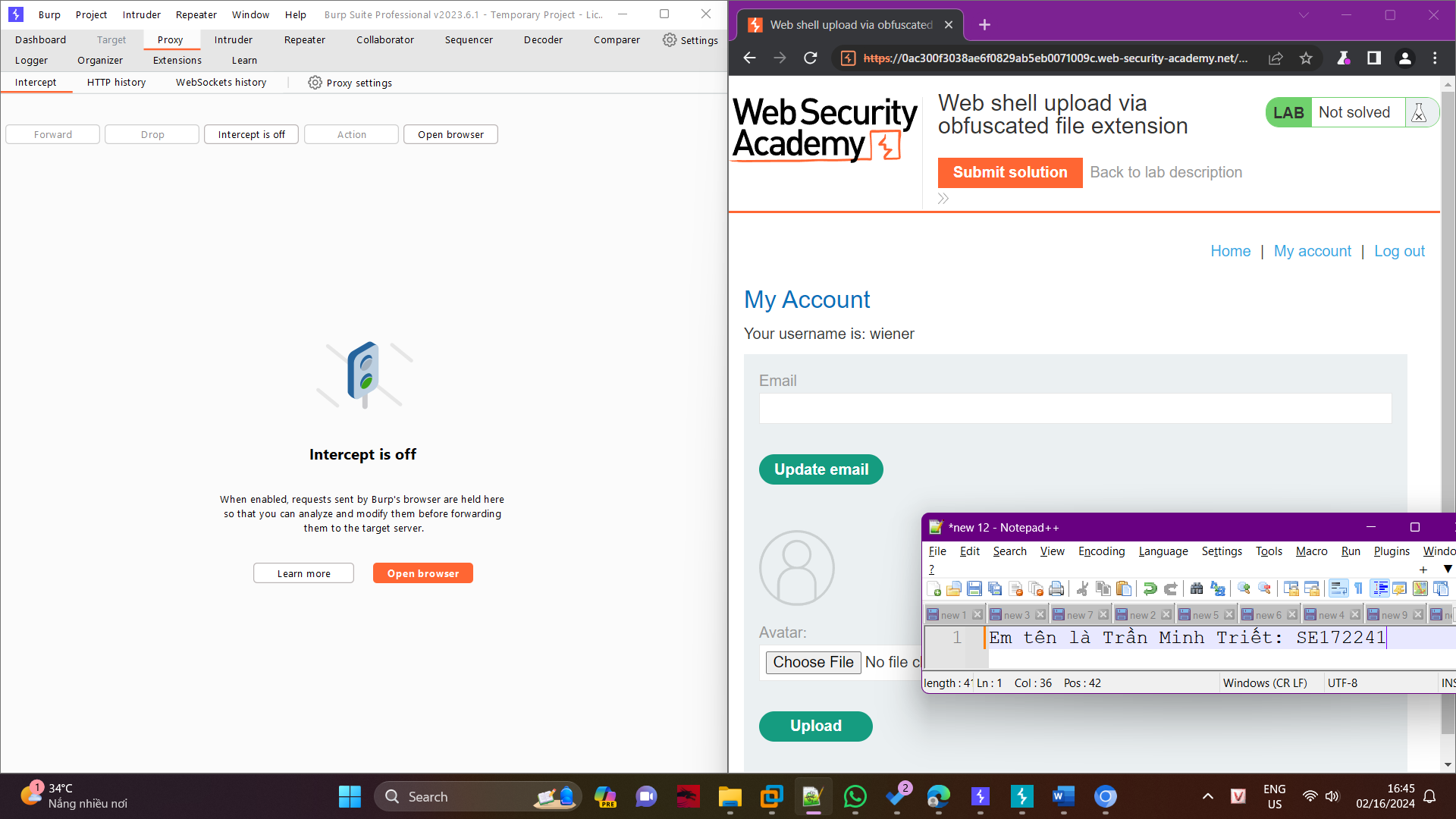
- \*\*MIME Type Spoofing\*\*: Manipulating the MIME type headers to trick the application into interpreting the file content differently than expected.

By exploiting File Upload vulnerabilities, attackers can upload and execute malicious files on the server, leading to unauthorized access, data breaches, system compromise, and other security risks. To mitigate these risks, developers should implement robust file upload validation, including proper file type checks, content type validation, file size limitations, and secure file storage practices. Additionally, regular security testing and monitoring can help detect and remediate File Upload vulnerabilities before they are exploited by attackers.

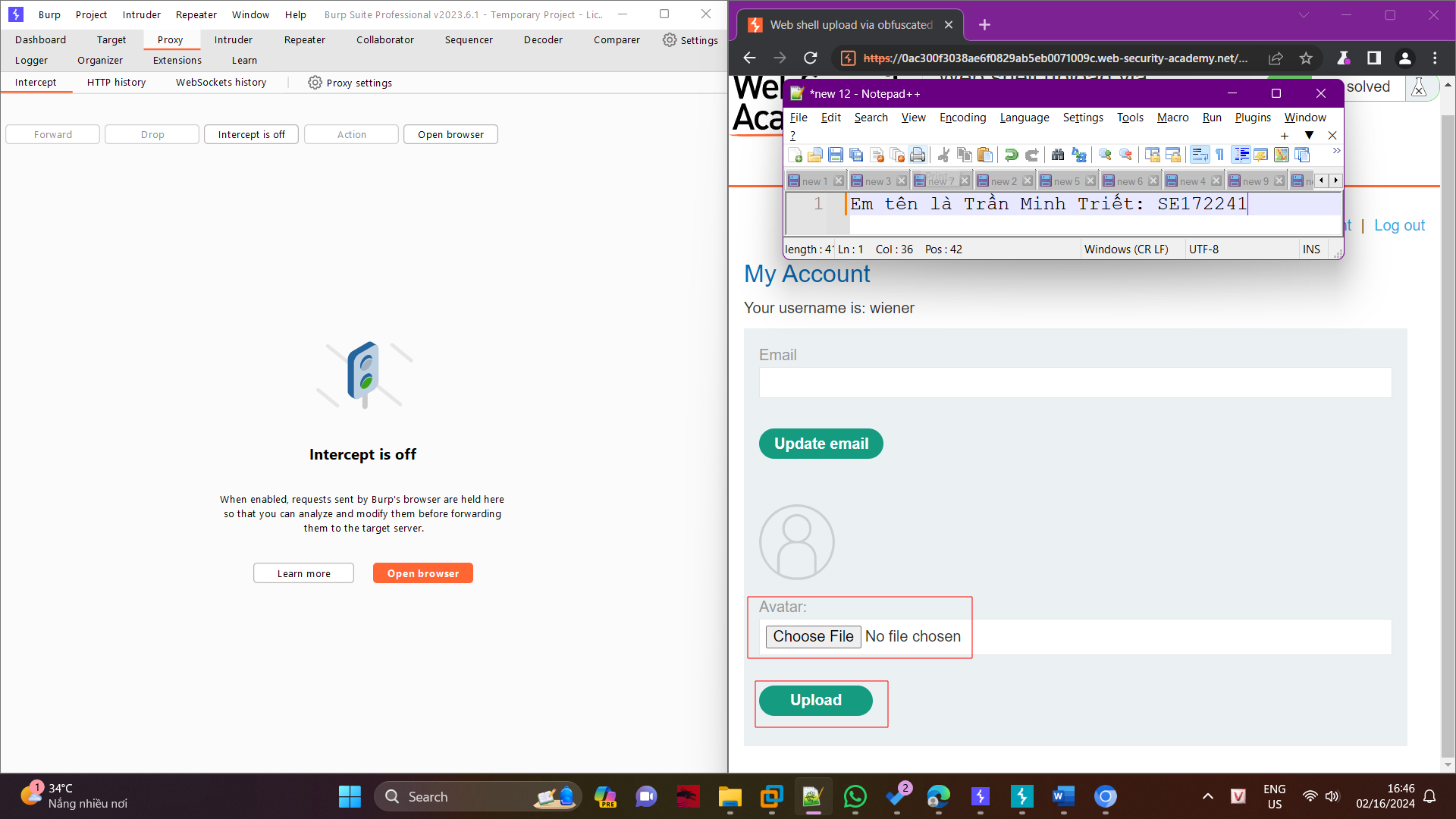
**Challenge**



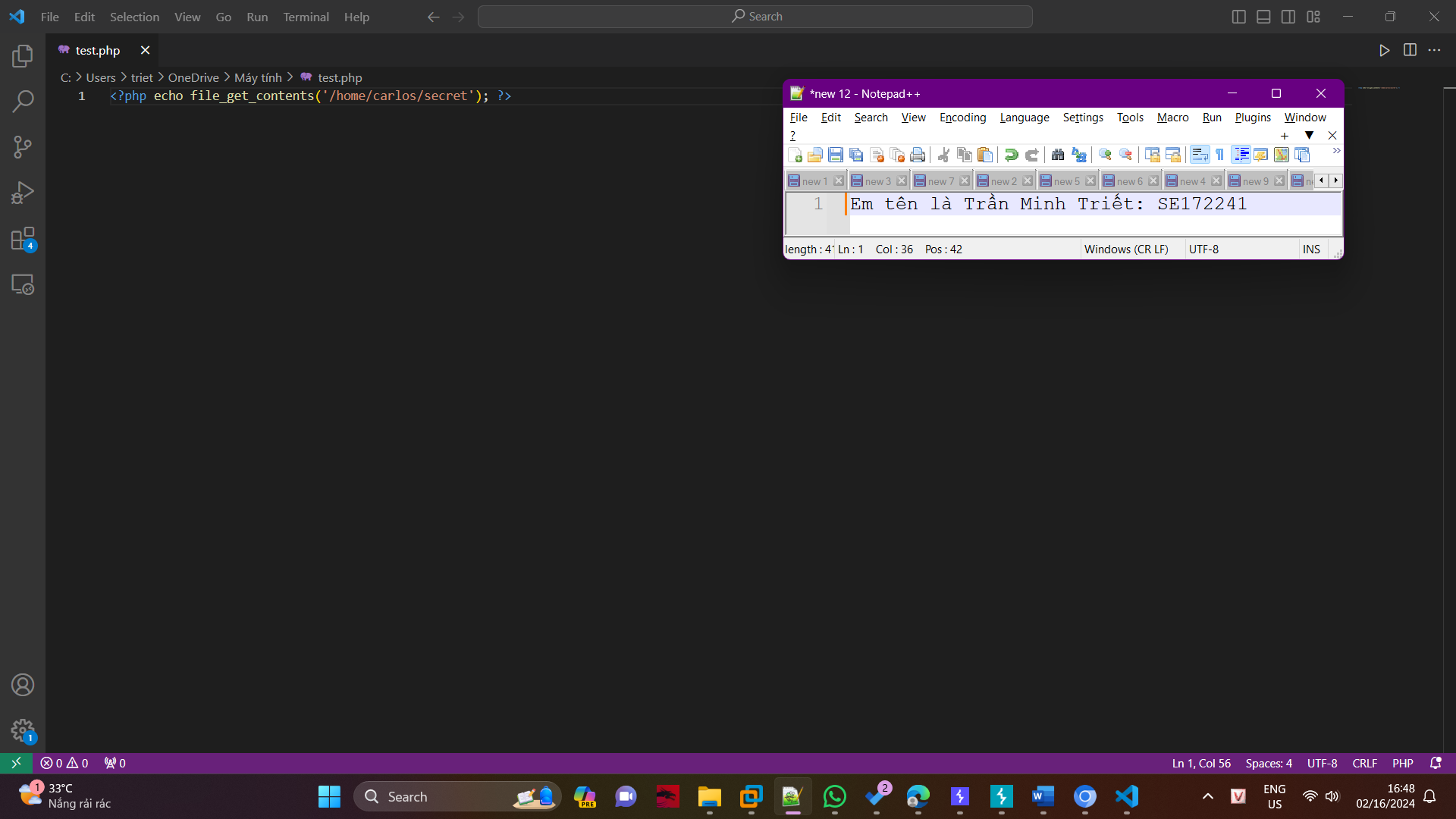
Then we’ll use the following credentials to log in: Username: weiner, Password: Peter.



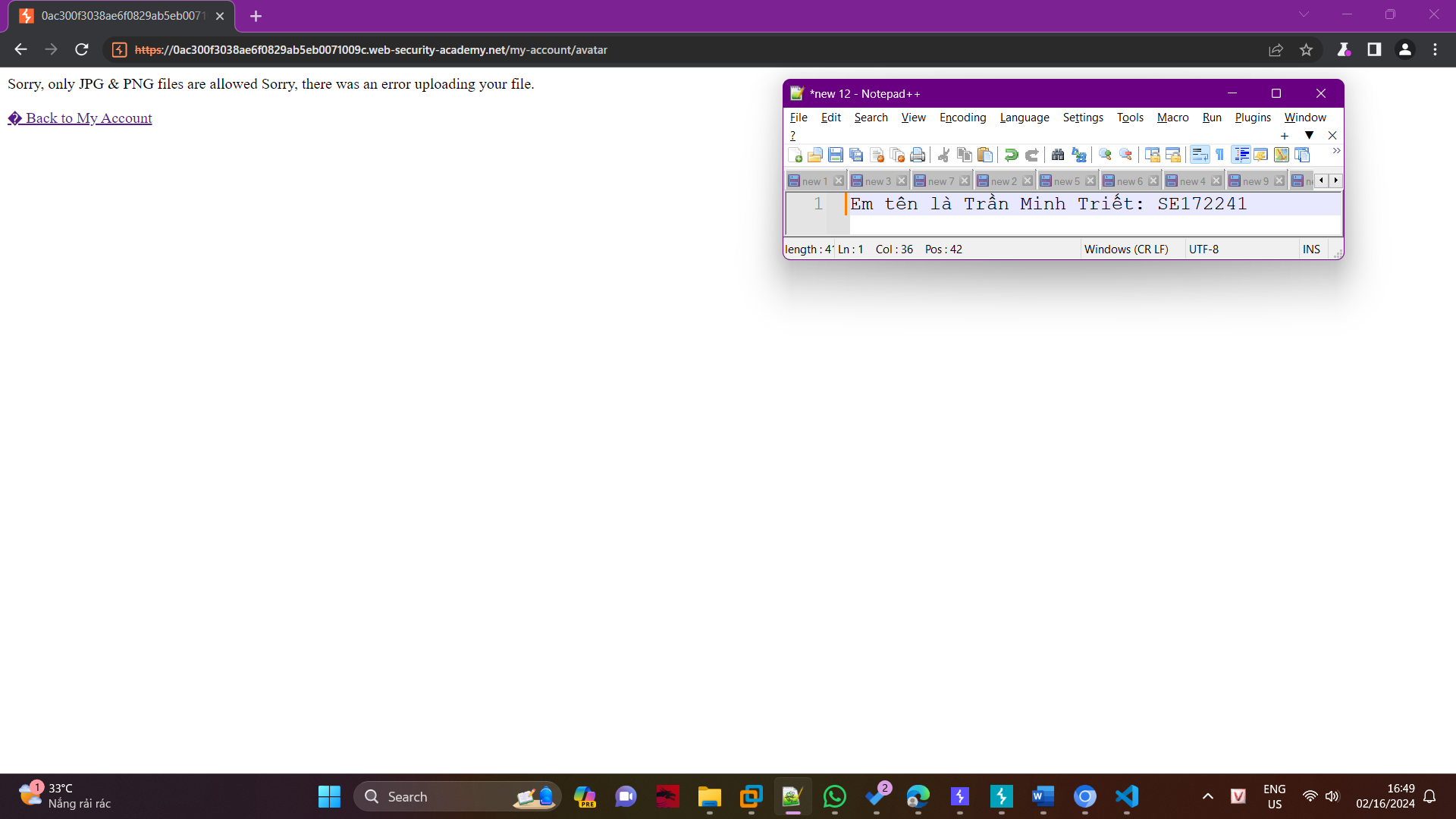
We can see an area to upload files; now we’ll see if we can upload a PHP file using this section.



As per the labs requirement, we must exfiltrate the contents of the file from/home/carlos/secretand to exfiltrate the file, we will construct a PHP file named test.php.



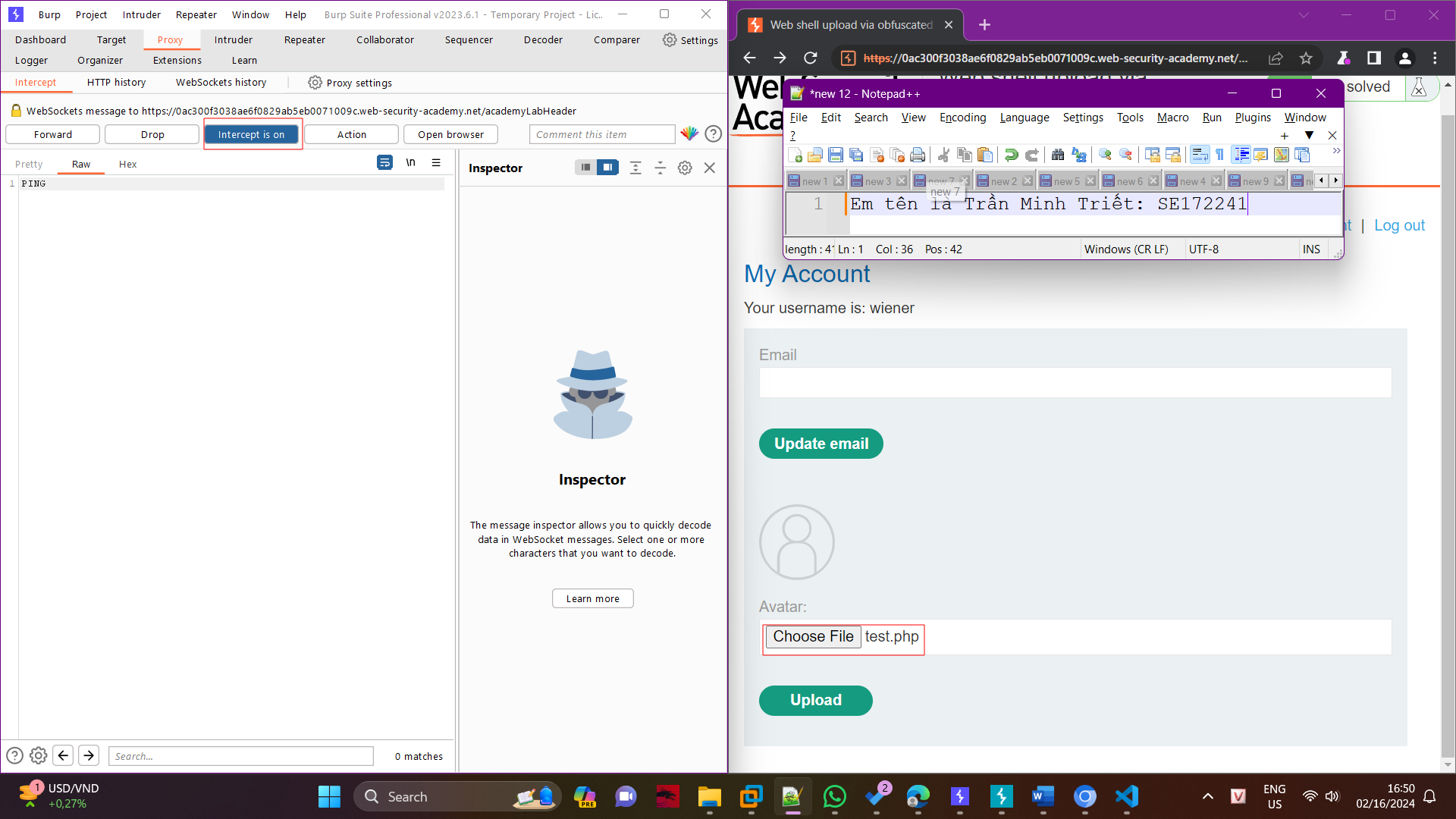
When we try to upload the php file, we get an error message claiming that only JPG and PNG files are allowed.

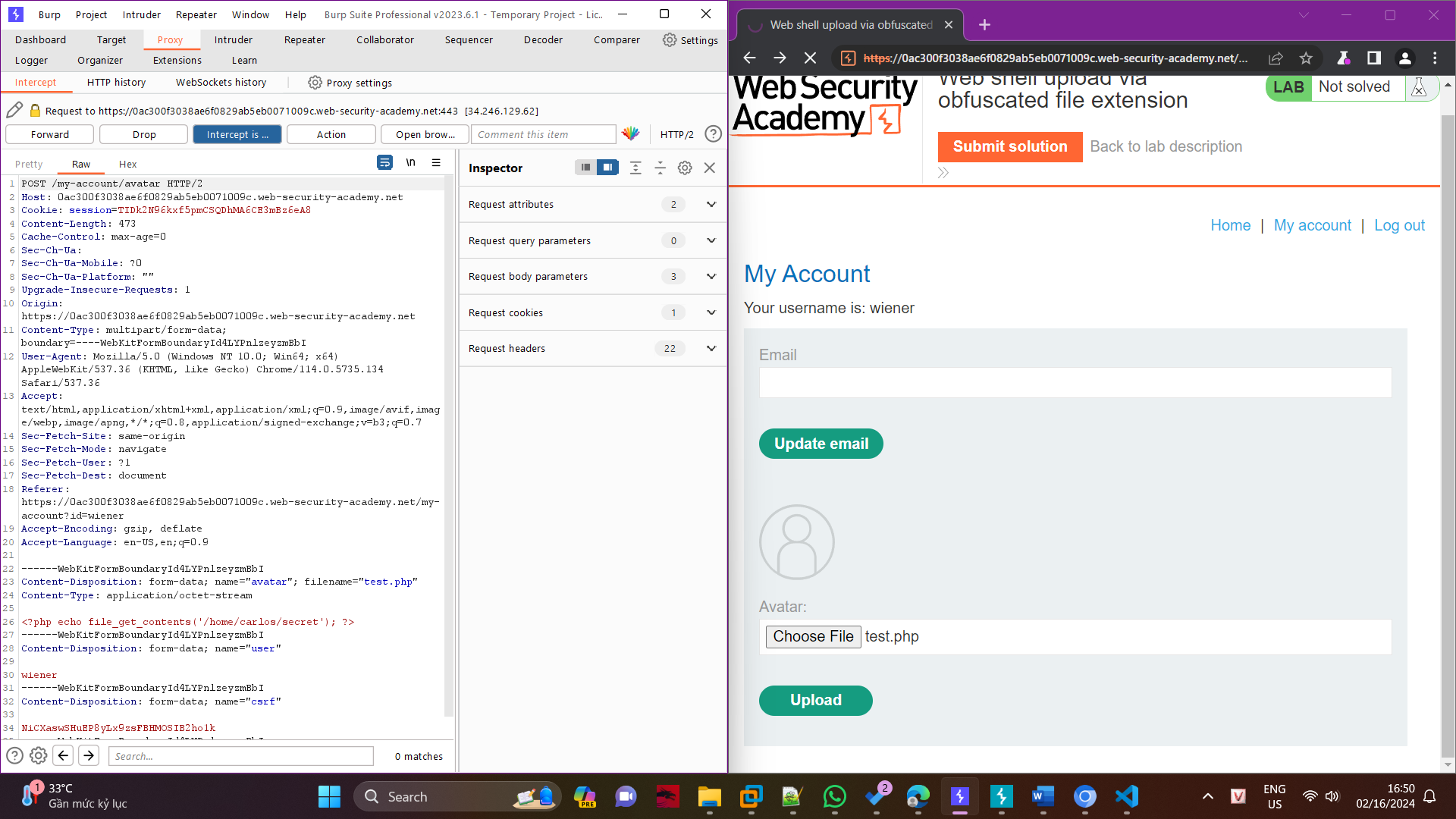


Now that we know it only accepts JPG and PNG files, we can make the PHP file to look like a PNG or JPG file.

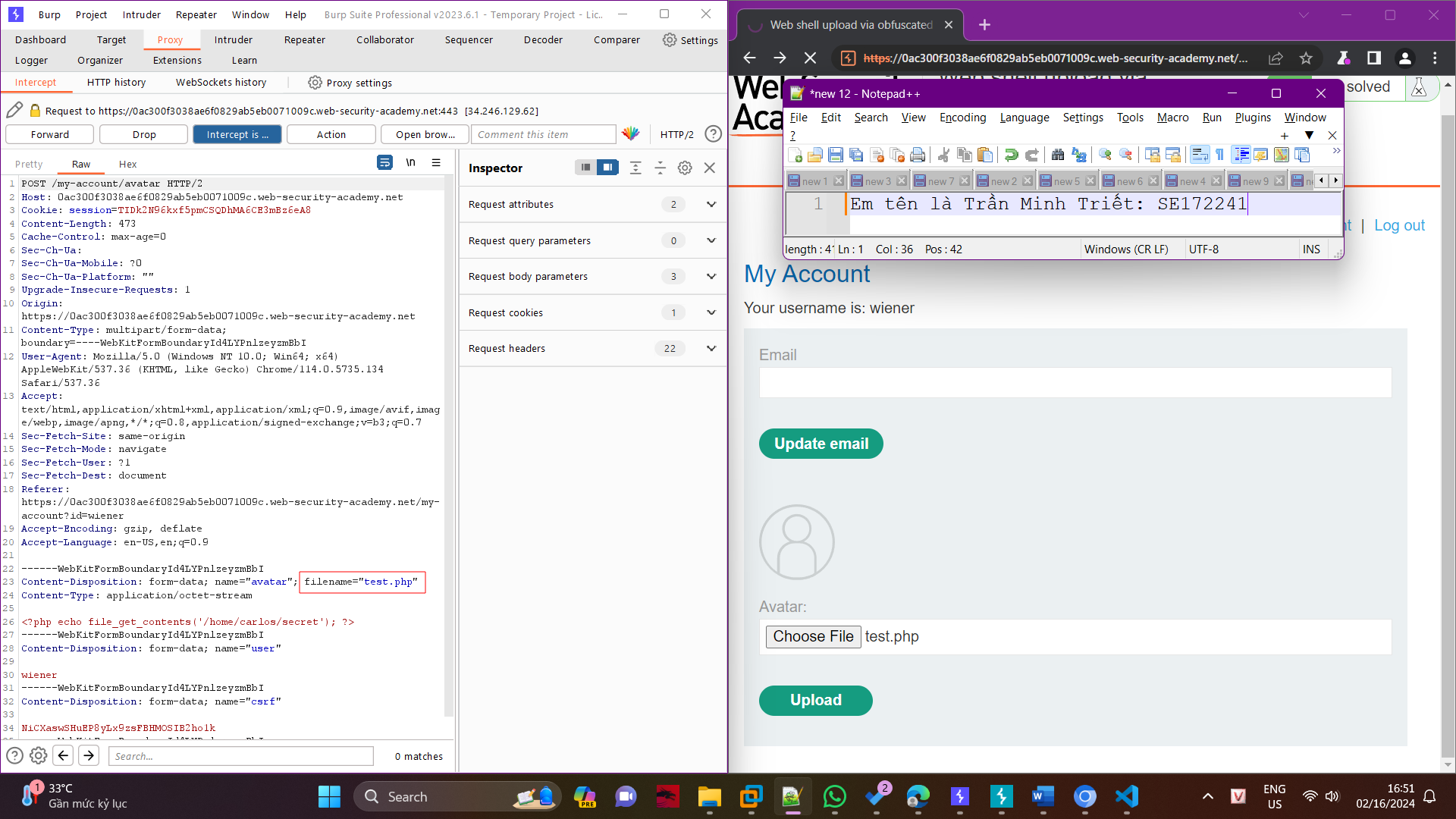
For that,

We will intercept the post request with Burp Suite.

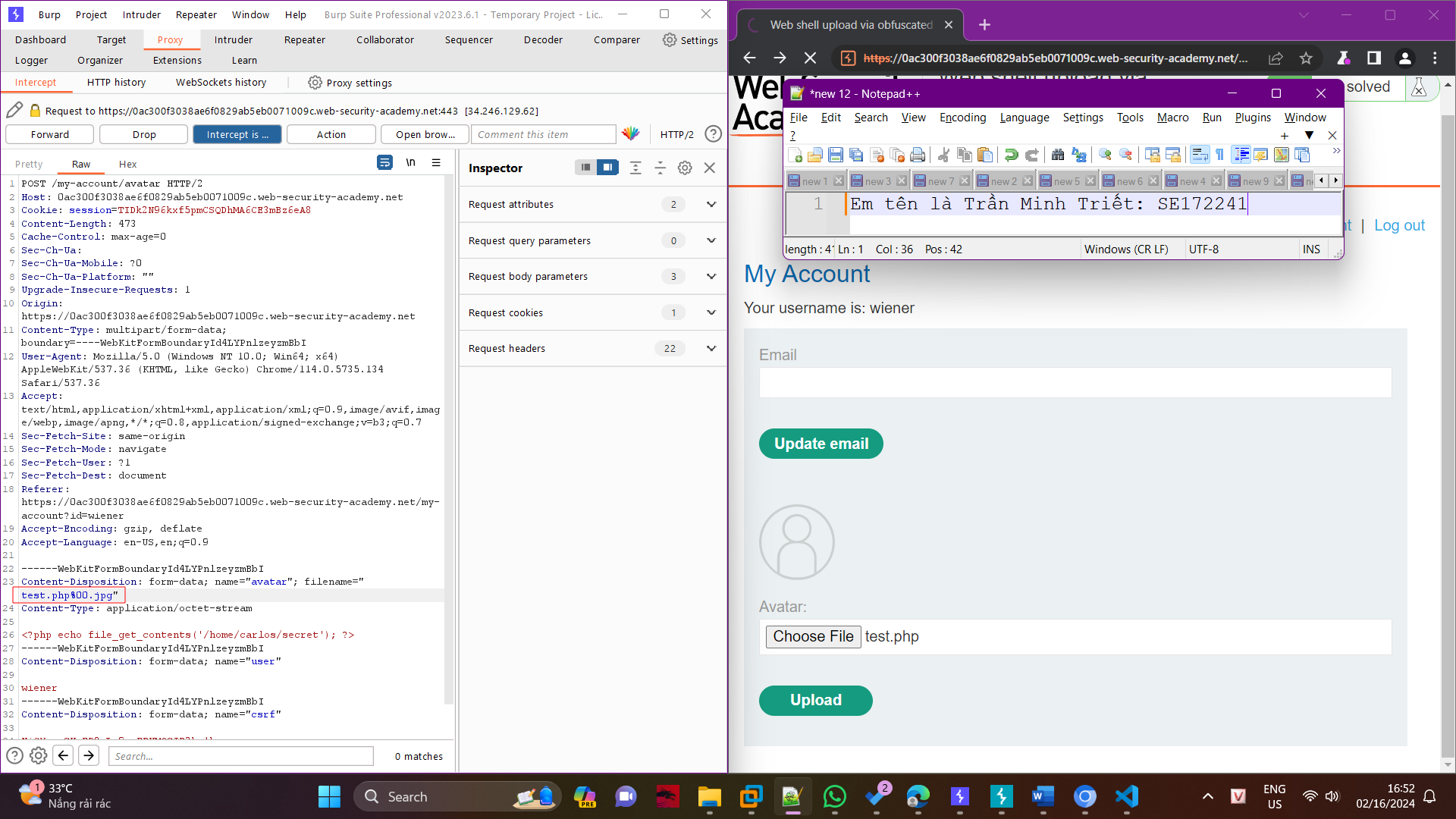




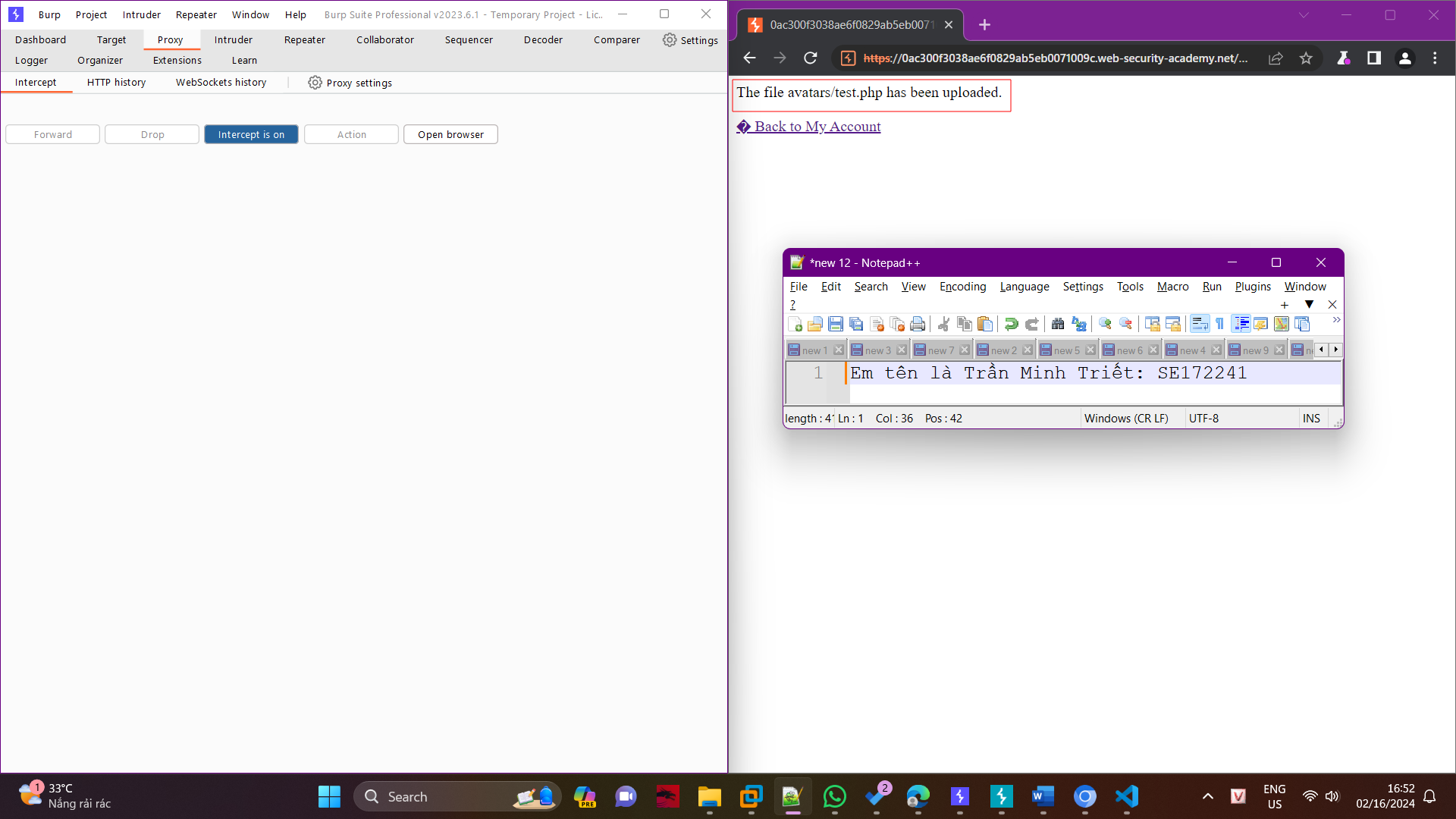
Then we can see that there is a section with the file that we are uploading, i.e., test.php



We will now attempt to obfuscate the file by changing the extension from test.php to test.php%00.jpg. The %00 in programming represents a null byte, a specific character that often denotes the end of a string and is used to fool file systems that rely just the file extension to identify the type of file. Even though it includes a PHP script, it can be mistaken for a safe.jpg picture.



And after sending the request, we receive a response that allows us to verify that the PHP file has been uploaded. At this point, we must locate the directory containing the file in order to execute it and obtain the secret — a prerequisite for the task at hand.



In order to access the data we are attempting to gain, we will need to change the directory to /files/avatars/test.php.

