Lab 12	
Name	Dang Hoang Nguyen
Student ID	SE171946

What is an OS Command Injection vulnerability, and how does it differ from other types of injection attacks?

An OS Command Injection vulnerability is a severe security weakness that allows attackers to execute arbitrary commands on the underlying operating system of an application. This grants them unauthorized access and control, podendo levar a uma série de consequências nocivas, incluindo:

- **Data theft:** Attackers can steal sensitive data like usernames, passwords, and financial information.
- **System damage:** They can delete files, corrupt data, or even wipe the entire system.
- **Lateral movement:** They can use the compromised system to gain access to other systems on the network.
- **Malware installation:** They can install malware to maintain persistent access or launch further attacks.

Attackers use various techniques to exploit OS Command Injection vulnerabilities, including:

- **Classic code injection:** As shown in the example above, directly injecting malicious commands into user input.
- **Path manipulation:** Exploiting flaws in how the application constructs file paths to inject commands.
- **Environment variable manipulation:** Attacking variables used to build commands, potentially influencing their execution.
- **Piggybacking on legitimate commands:** Appending malicious code to existing, permitted commands.

Explain how an attacker can exploit this vulnerability to execute unauthorized commands on the server's operating system and discuss the potential impact of such an attack on an application's security and data integrity.

1. Crafting the Exploit:

Attackers first identify vulnerable input points in the application. This could be a search bar, a comment field, or even a URL parameter. They then craft malicious code disguised as user input, aiming to achieve their desired actions. Examples include:

- **Direct Code Injection:** Injecting commands directly, like '; rm -rf /' to delete files.
- **Path Manipulation:** Exploiting path handling to execute unintended commands, like ../../../../bin/bash.
- **Environment Variable Abuse:** Modifying environment variables that influence command execution, like setting PATH to include attacker-controlled directories.

2. Executing the Exploit:

Once crafted, the attacker submits the malicious input through the vulnerable point. The application, unaware of the danger, incorporates it into the operating system command and executes it. This essentially runs the attacker's code on the server, granting them unauthorized access.

3. Gaining Control and Causing Damage:

With this access, attackers can:

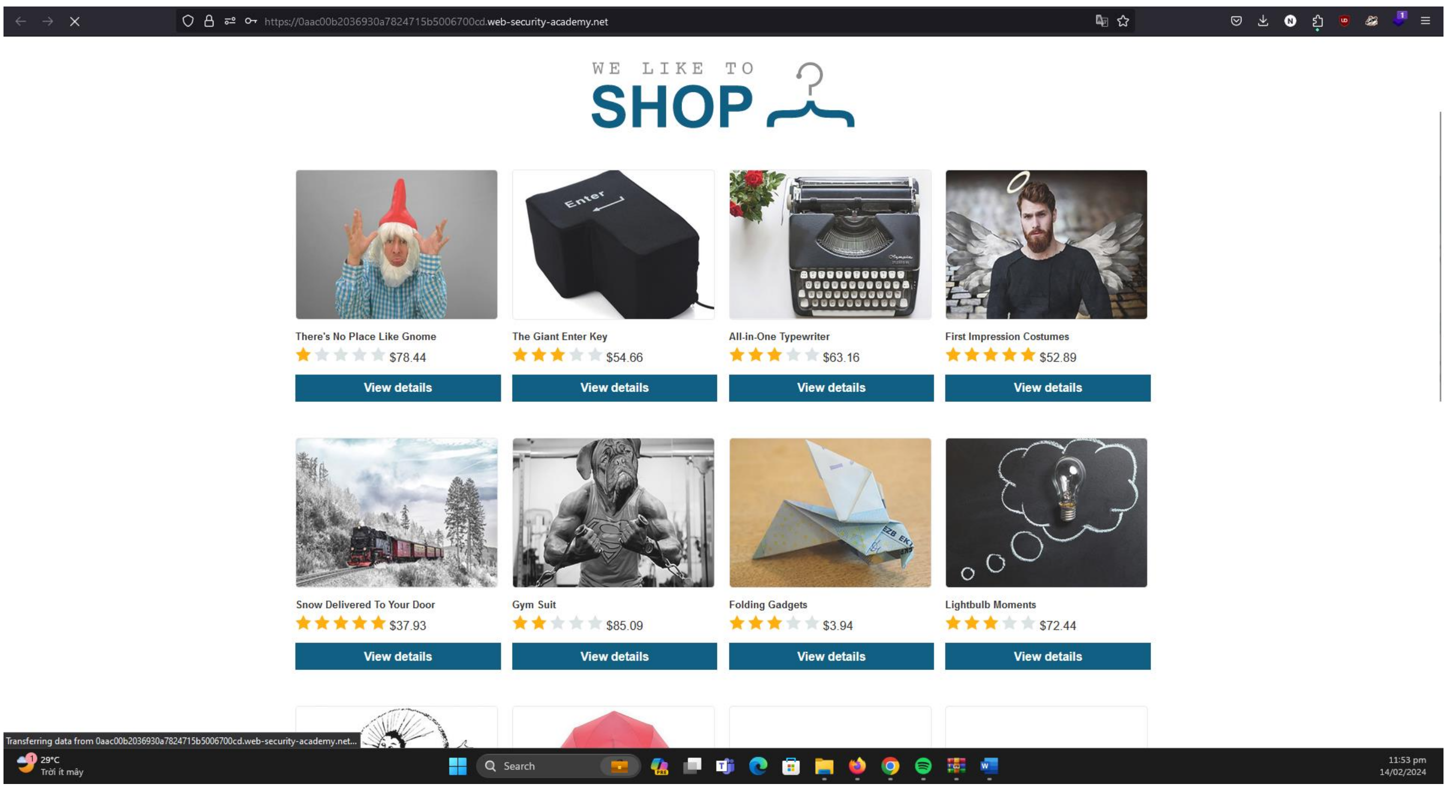
- **Steal Data:** They can access and exfiltrate sensitive data like passwords, credit card information, or confidential documents.

- **Install Malware:** They can install backdoors, ransomware, or other malicious software for persistent access or further attacks.
- **Disrupt Operations:** They can delete files, corrupt databases, or disable critical services, causing downtime and financial losses.
- **Escalate Privileges:** They can leverage compromised systems to gain access to other parts of the network, potentially escalating the attack's impact

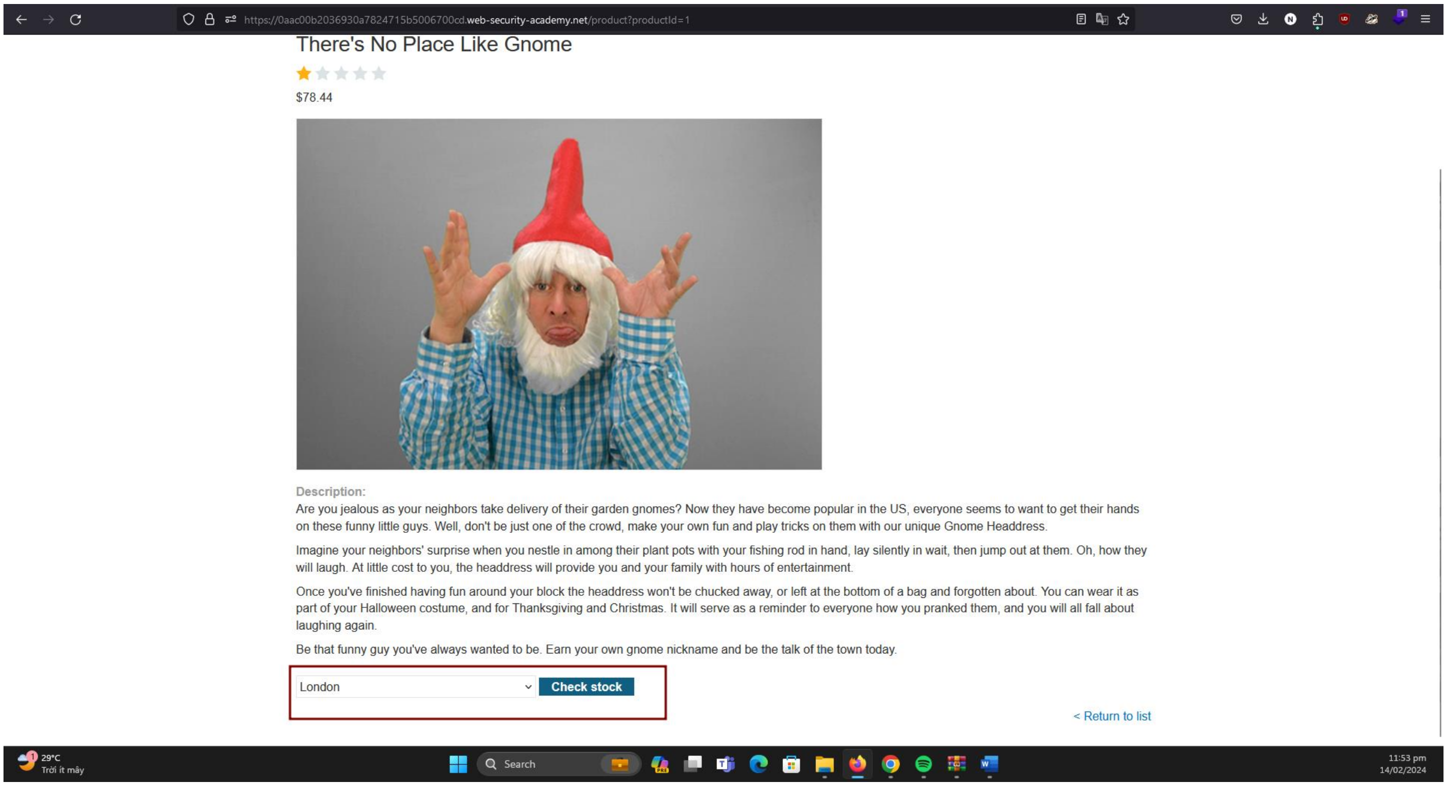
LAB 1 : OS command injection, simple case

The product stock checker in this lab has an OS command injection vulnerability. The program runs a shell command with product and store IDs entered by the user, and it returns the command's raw output as part of its answer. Use the whoami command to find the name of the current user in order to solve the lab.

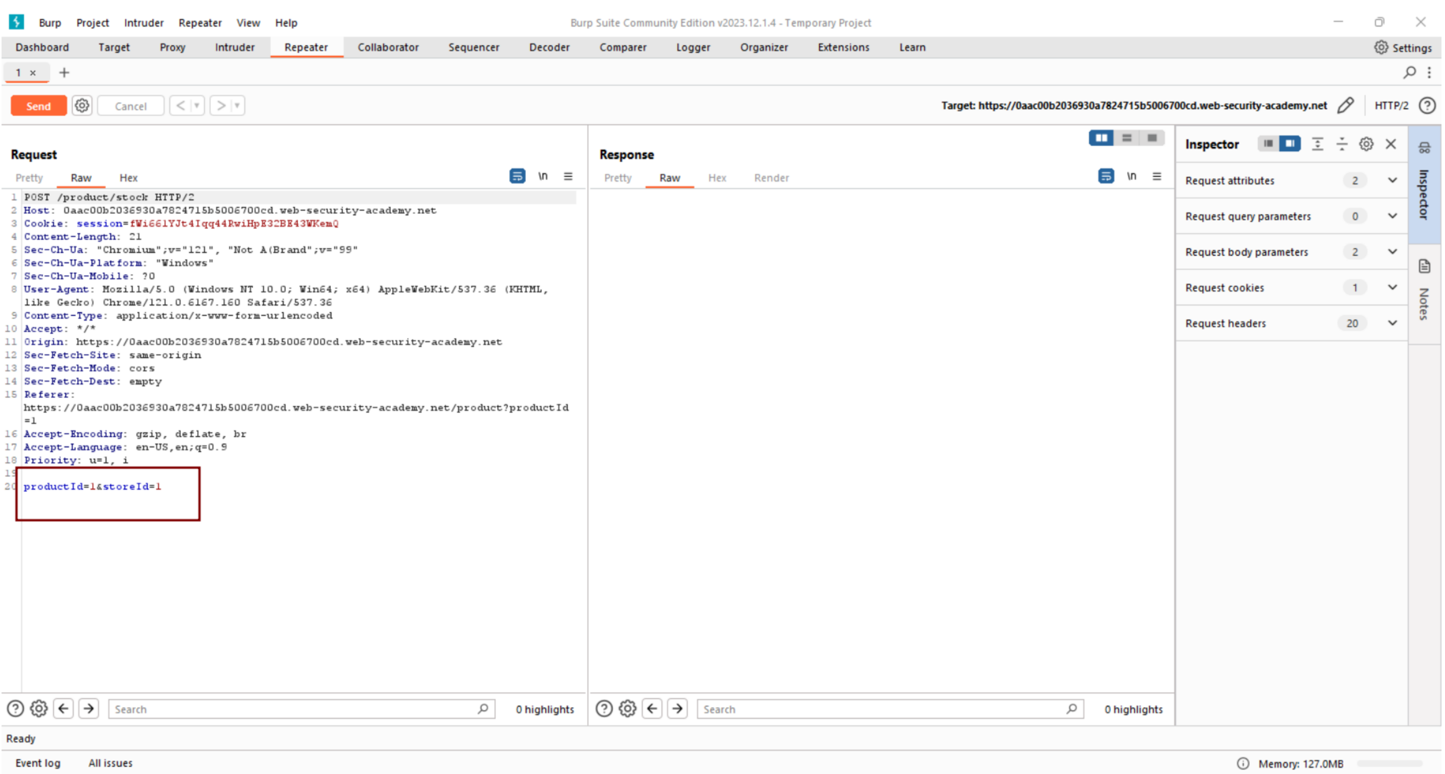
The web application has been shown to us in the lab. It has numerous products mentioned.as shown in the image below.



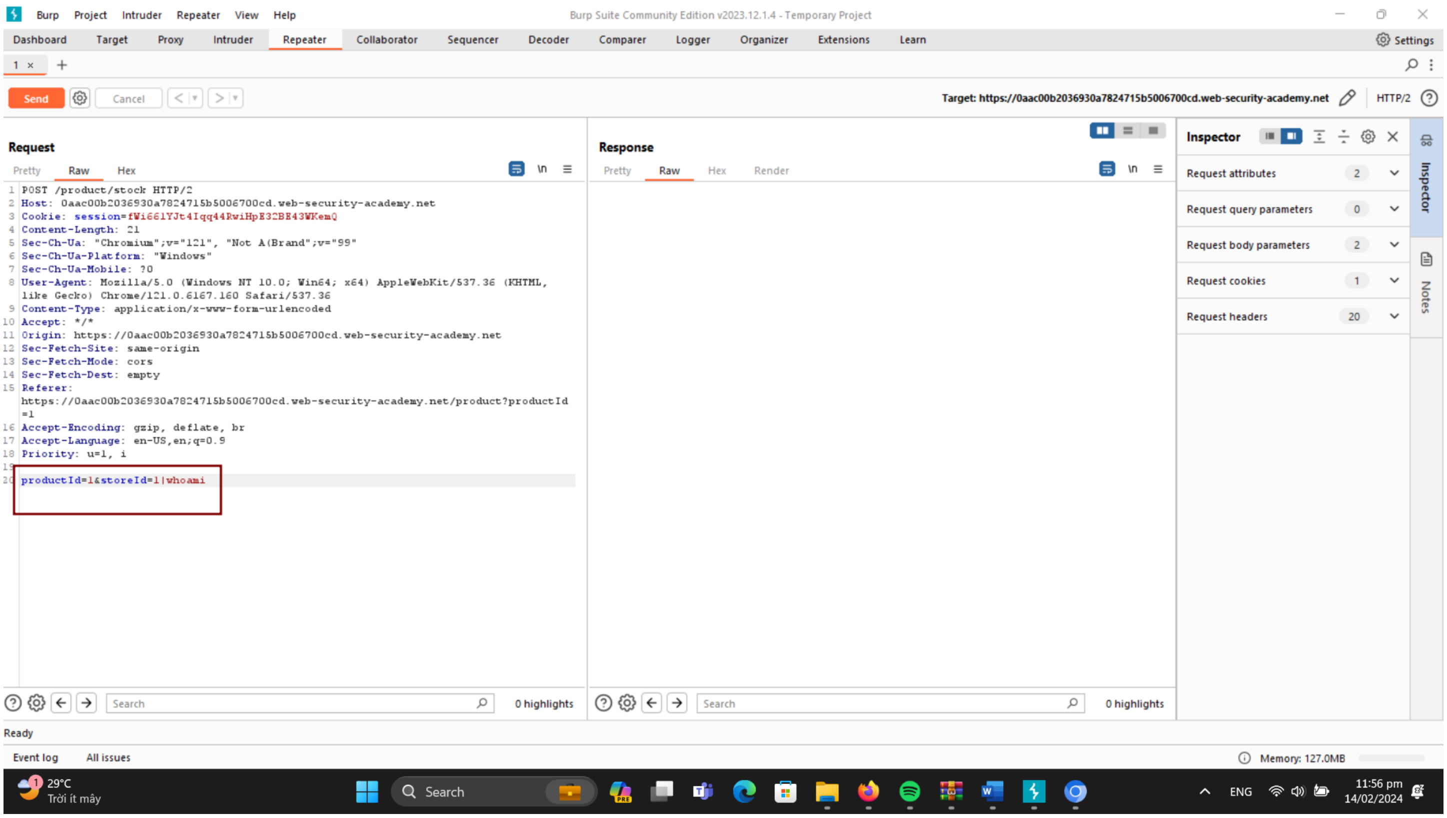
We receive detailed information about a product when we see its view details, and we can observe that a product-specific parameter called productId is being set when we visit a particular product. This is the point where we can attempt command injection, which I tried, but it didn't work because the information is retrieved via an API that blocks all special characters. However, there is an additional feature that lets us verify the stock availability. as shown in the image below.



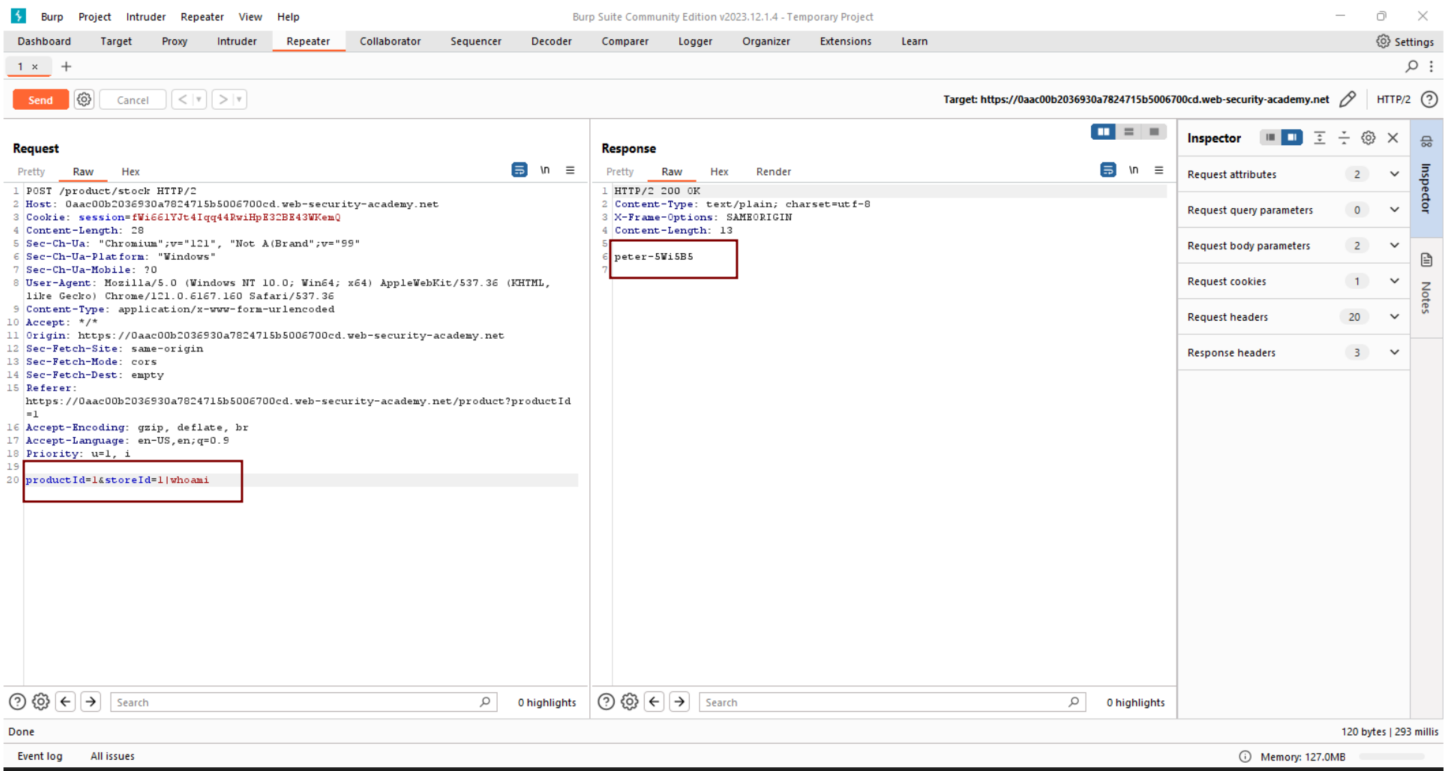
We may observe that there are results when we use the stock check functionality. However, we are not seeing any parameter changes. Perhaps there is a covert aspect to the request that we are unaware of. The tool burp suite enters the picture at this point. A web proxy tool called Burp Suite can be used to analyze requests, change requests, and do much more.



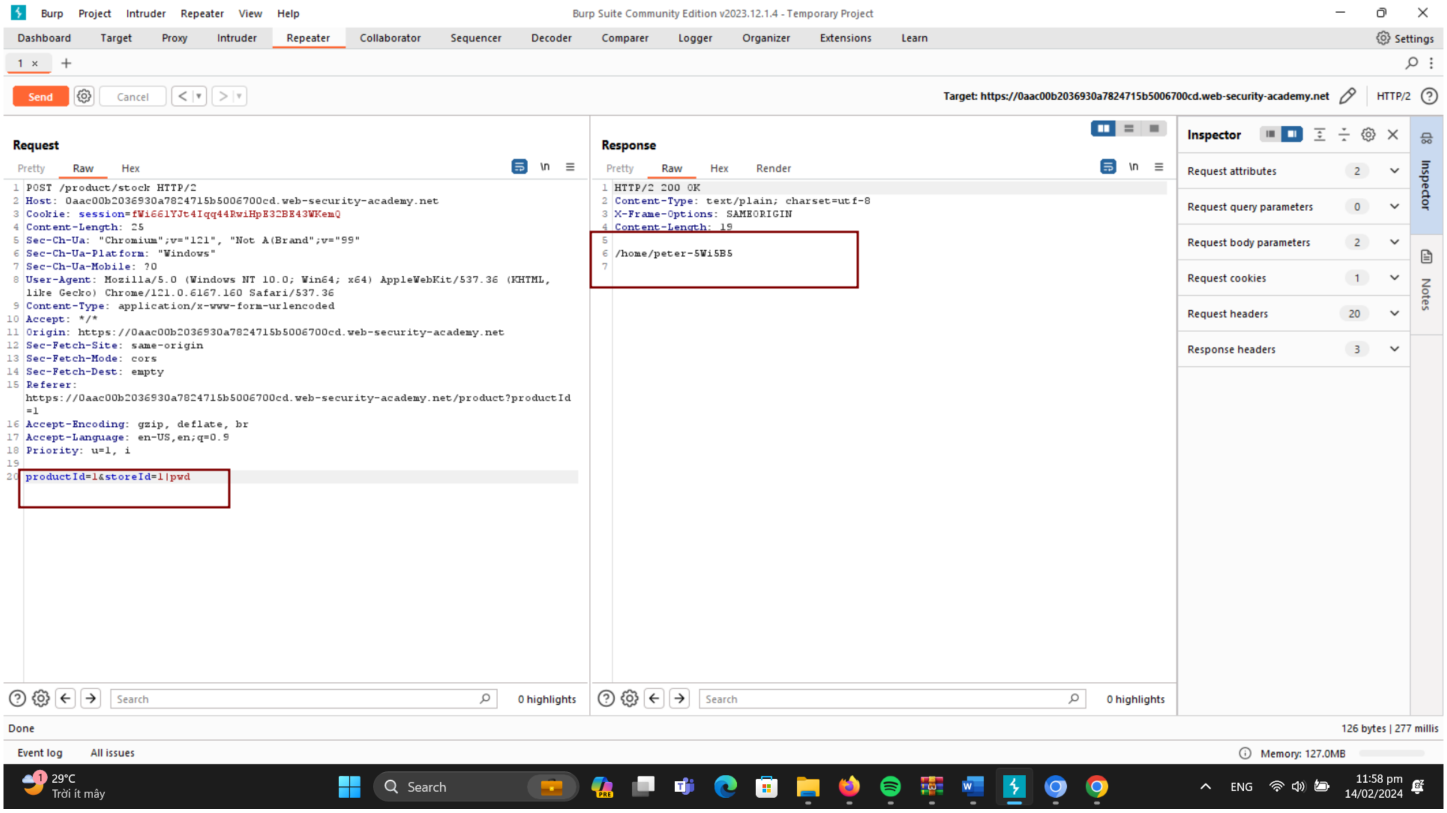
As seen in the image above, a secret parameter named storeId is used to retrieve information when we capture a request to access the stock checker's functionality. It's time to check the storeId parameter since we have already tested the productId parameter for OS command injection and found that it is not susceptible to it.



It's time to test OS command injection on the hidden parameter storeId as we capture the request in the burp suite. A special character can be used to carry out several commands simultaneously. As seen in the image above, we test command injection by using a pipe character to run an OS command on the back-end server. Other special characters that we can employ include'and &, among many others. One can independently test it. The outcome is seen in the picture below as we transmit the amended request mentioned above.



From the image above, we can see that our arbitrary command executed correctly. This allows us to use OS command injection.Additionally, we may check for other commands like netstate, pwd, uname -a, and so on. You can try it on your own, however as you can see in the image below, I tested another command with pwd.



As we forward this request, we can see that our command executed properly and provided us with information about the target system's current working directory, as shown in the image below.