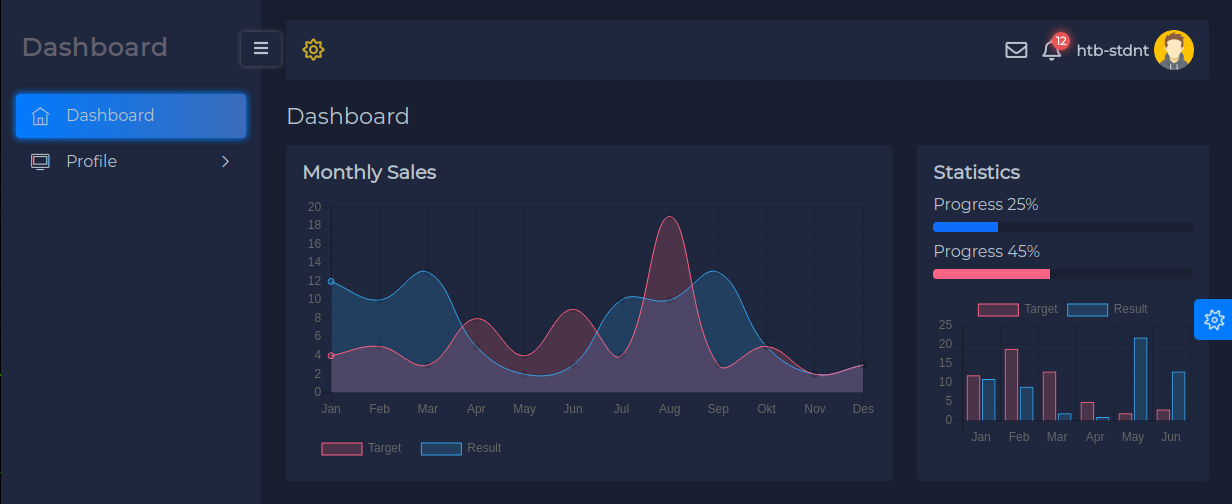
**Second-Order Command Injection**

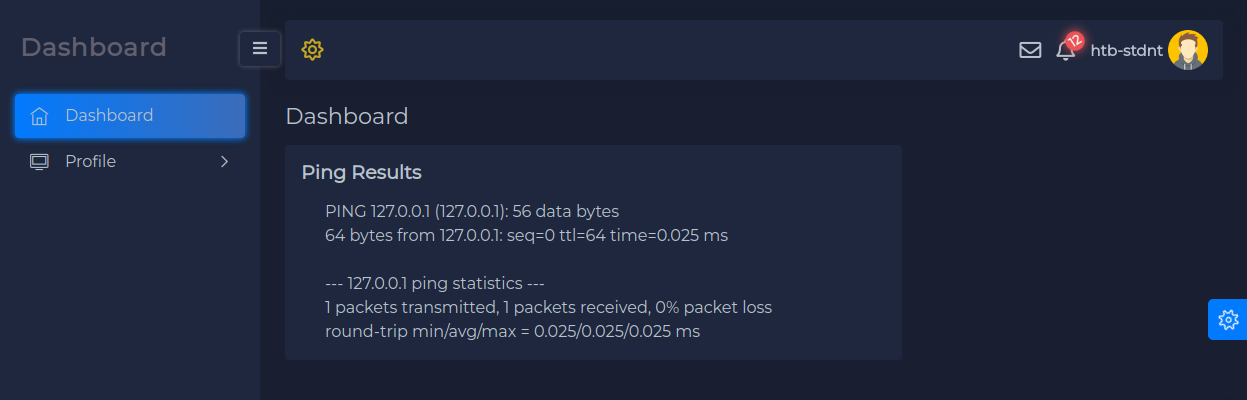
As our final example of a second-order vulnerability, we will explore a second-order command injection vulnerability. It is often apparent when a web application executes system commands. However, since command injection is a common and severe vulnerability, web developers often secure these obvious code execution entry points with proper filters, making command injection impossible. Though, many web applications implement additional tasks in the background that interact with the operating system. An external attacker often does not know about these background tasks as they are not displayed in the web application. Therefore, checking all input fields for potential command injection issues is crucial, even if there does not seem to be an obvious code execution entry point.

**Testing the Web Application**

After registering a test user in our sample web application, we can see a simple admin dashboard:

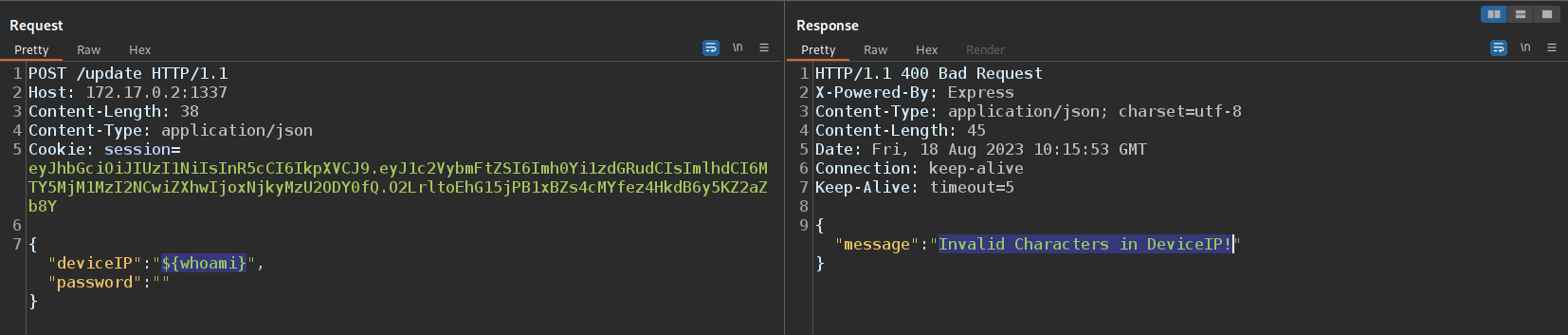


In the /ping endpoint, the web application allows us to set and ping an IP address, displaying the result back to us:



This functionality is an obvious entry point for a potential code execution vulnerability, as the web application executes the ping command with the IP address we supplied in our profile; if the web application uses system commands without proper sanitation, there is a command injection vulnerability.

We can apply the methodology taught in the [Command Injections](https://academy.hackthebox.com/module/details/109) module to test for this potential issue. Let us try a simple command execution payload that executes the whoami command. However, as we can see, the web application rejects our payload due to invalid characters:



So we know that the web application implements a filter that prevents us from injecting arbitrary characters into the deviceIP parameter. To achieve code execution, we need to determine if we can inject any characters that would trigger executing an additional command. Assuming the filter blocks certain characters, we can quickly achieve this using a fuzzer such as wfuzz. Let us determine if we can inject any special characters by using the [special-chars.txt](https://github.com/danielmiessler/SecLists/blob/master/Fuzzing/special-chars.txt) wordlist from SecLists. Since a successful change of the IP address results in an HTTP 200 status code and an unsuccessful attempt results in an HTTP 400 status code, we can match all 200 status codes to filter all blocked characters:

yovecio@htb[/htb]$ $wfuzz -u http://172.17.0.2:1337/update -w ./special-chars.txt -d '{"deviceIP":"FUZZ","password":""}' -H 'Content-Type: application/json' -b 'session=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6Imh0Yi1zdGRudCIsImlhdCI6MTY5MjM1MzI2NCwiZXhwIjoxNjkyMzU2ODY0fQ.O2LrltoEhG15jPB1xBZs4cMYfez4HkdB6y5KZ2aZb8Y' --sc 200

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Wfuzz 3.1.0 - The Web Fuzzer \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Target: http://172.17.0.2:1337/update

Total requests: 32

=====================================================================

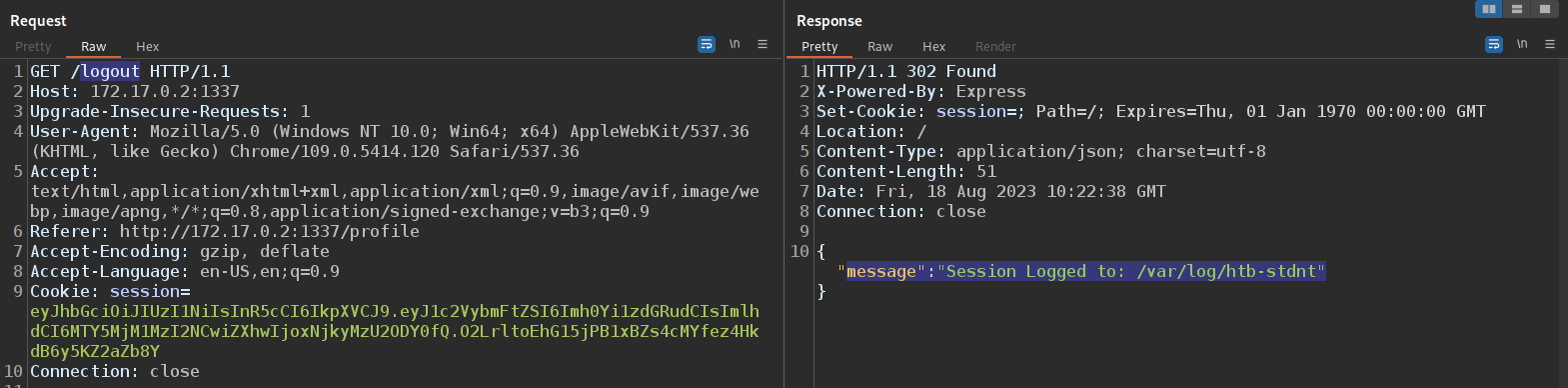
ID Response Lines Word Chars Payload

=====================================================================

000000024: 200 0 L 3 W 40 Ch "."

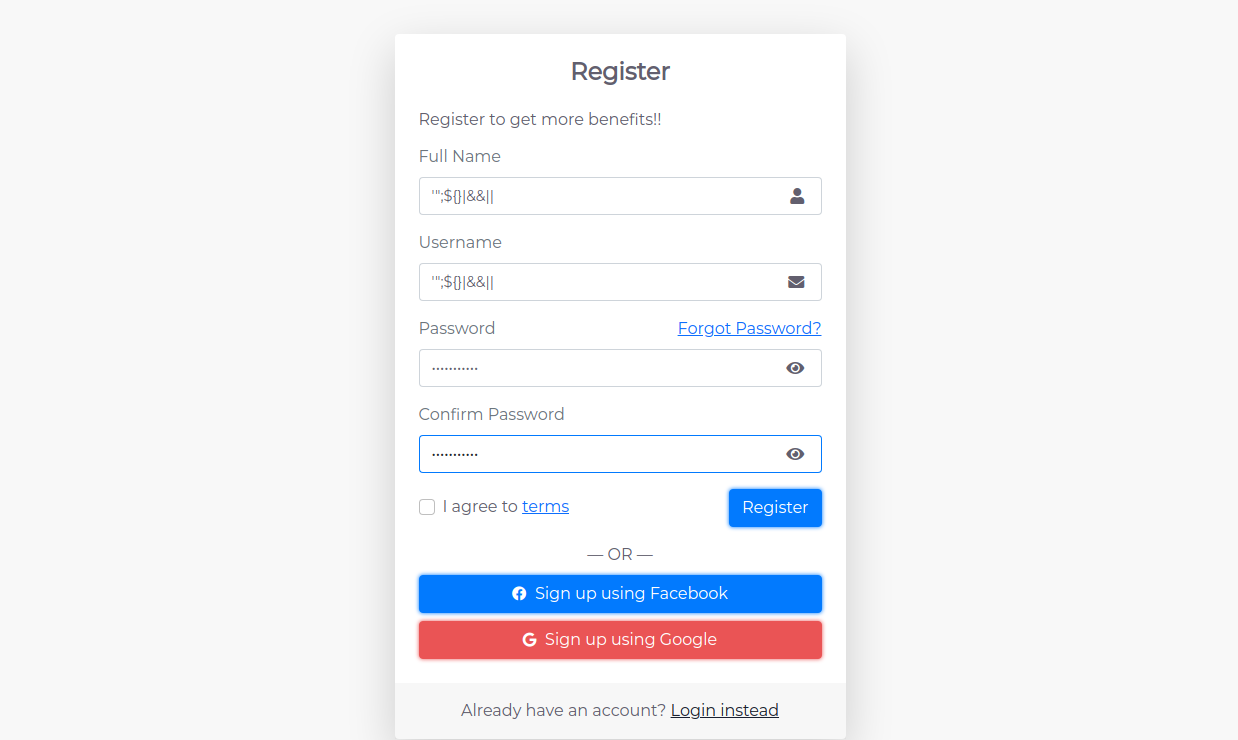
We can see that the only special character allowed is the period; thus, we cannot inject any payload that would result in code execution. Therefore, we need to determine if there is another way to change the IP address that bypasses the filter or if the web application potentially executes system commands at a different endpoint.

If we analyze the network traffic closely, we can observe interesting behavior. When we log out of the web application, we are redirected to the login page. However, the response also contains the following content:

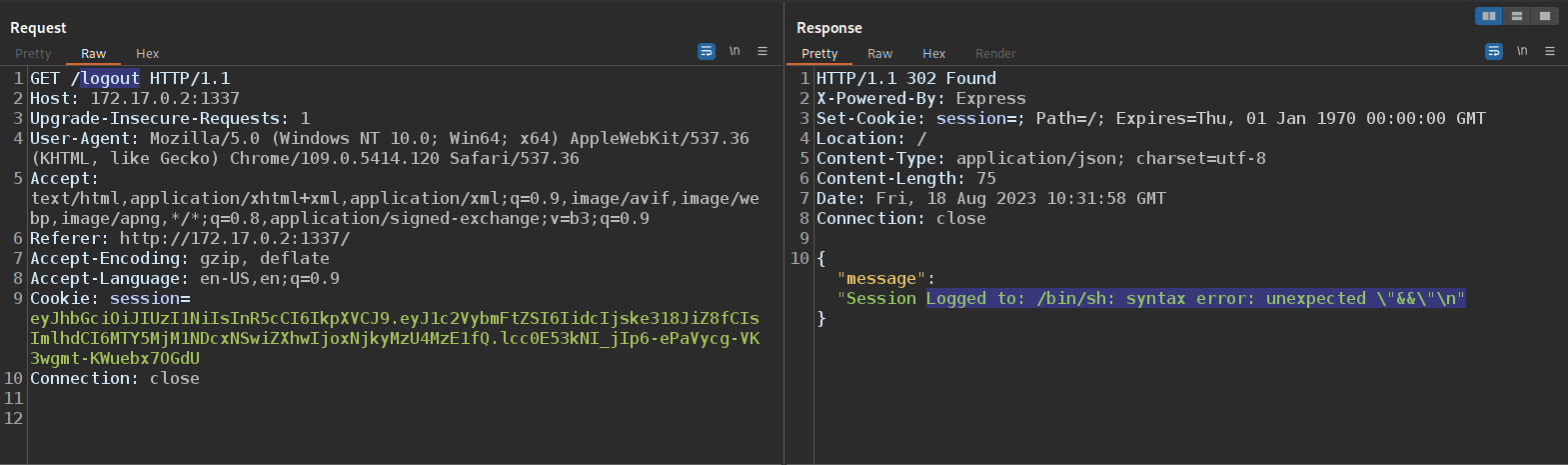


While it may initially appear to be a debug message that inadvertently discloses a path on the web server (essentially an information disclosure issue), it also suggests that the web application logs data based on user profiles. Depending on how the web application implements logging, there may be an opportunity to inject a payload into our user data, potentially leading to code execution.

Regardless that the web application contains no functionality to modify user data, we can instead register a new user and inject a payload into the user data. We will attempt to register a user with special characters in the name and username parameters to identify a potential vulnerability.

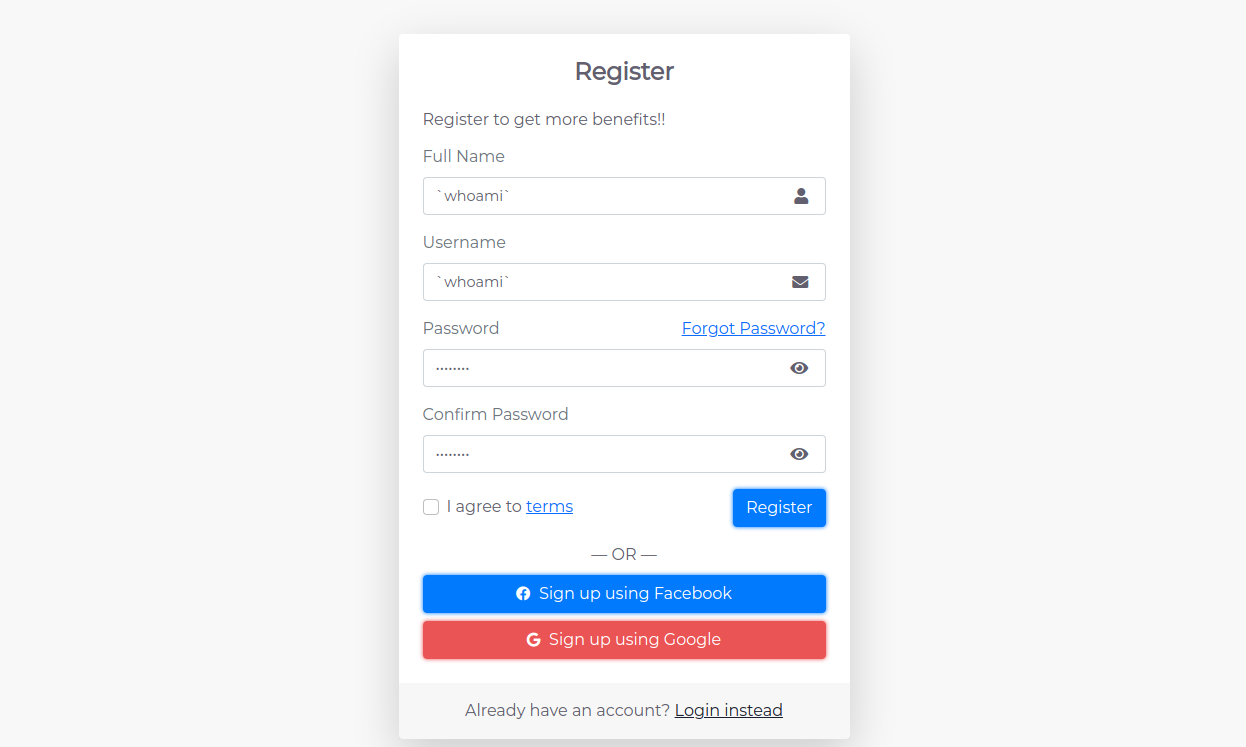


Afterward, we can log in as the newly registered user. If we now log out and analyze the response sent by the web application, we can indeed identify a potential command injection vulnerability:

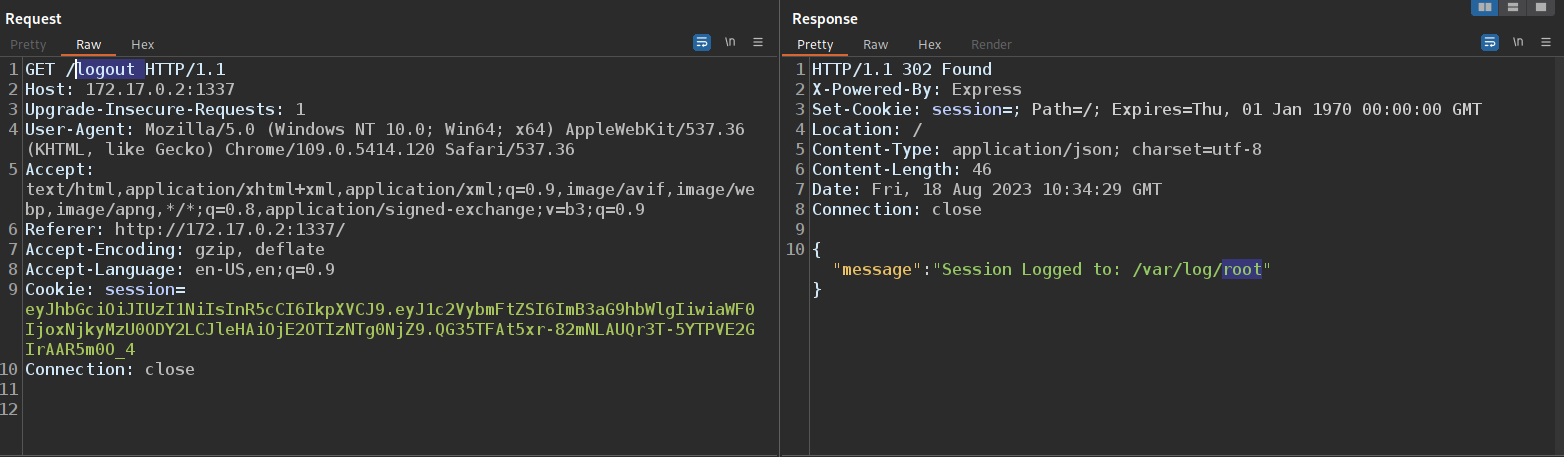


**Exploitation**

To exploit the vulnerability, we need to register a user with any valid command injection payload, for instance, by using back-ticks:



After logging in and logging out, the injected command is executed:



The web application implements a filter to protect against obvious command injection entry points; however, it lacks a proper filter for the background logging mechanism. If the debug messages at user logout were not left over, there would be no way for us to know about this mechanism.

Therefore, testing all user input fields for potential security vulnerabilities is crucial; this can include hundreds or even thousands of input fields in real-world web applications, so we need to rely on automated scanners to help us save time. However, we should always perform manual testing on input fields we deem of particular interest, such as inputs related to our user profile (like our username), which the web application may use in background processes, such as a hidden logging mechanism.