ASSESSMENT 6.2HD

SIT736-Identity, Access management and Physical security

PREMKUMAR SHARMA
S224784353

Table of Contents

SIT736: Keycloak Security Testing and Vulnerability Analysis

Abstract- This report details the security testing and vulnerability analysis of a Keycloak deployment as part of the SIT736 IAM and Physical Security course. The project simulates an attacker exploiting misconfigurations in a Keycloak server to access protected user data via phishing and enumeration techniques. The tasks include service detection, realm and client ID enumeration, crafting phishing URLs, brute forcing access tokens, and securing the application against these vulnerabilities. Using tools like Nmap, dirb, and custom Python scripts, the attacker enumerates realms and client IDs, exploits OAuth 2.0 vulnerabilities, and retrieves sensitive user information. Countermeasures, including IP restrictions, rate limiting, and OAuth 2.0 hardening, are applied to mitigate these risks. Results demonstrate successful exploitation and subsequent securing of the system, highlighting Keycloak configuration best practices.

Keywords: Keycloak, Nmap & Dirb, Python Scripts, Client ID enumeration, OAuth Vulnerability.

I. Introduction

Keycloak is an open-source Identity and Access Management (IAM) solution that sup ports OAuth 2.0 for secure authorization. However, misconfigurations in Keycloak deployments can expose vulnerabilities, allowing attackers to impersonate legitimate clients and access protected resources. This report outlines a security testing exercise for a Keycloak server, simulating an attacker's workflow to exploit OAuth 2.0 vulnerabilities and subsequently applying countermeasures. The tasks include service detection (Part 1), enumerating OAuth endpoints and realms (Part 2), client ID enumeration (Part 3), crafting a phishing URL (Part 4), brute-forcing access tokens (Part 5), and securing the application (Part 6). The methodology follows a structured approach using tools like Nmap, dirb, and cURL, with custom scripts for automation.

II. Methodology

1. Setup Pre-requisites

To commence the security testing and vulnerability analysis for the SIT736 Keycloak project, the initial step involves acquiring and organizing the external resources provided alongside the task sheet. These resources, critical for the successful execution of the project, are contained within the 6.2HD.zip archive. The archive includes the following essential files as depicted in Figure V—1(Unzipping the External Resource):

1. Task6.2HD.zip
2. Realmnames.txt
3. Clientids.txt
4. Relam-export-task6.json

Upon extraction of the Task6.2HD.zip file, the resulting directory structure reveals additional components necessary for the task: the "attacker-app folder", which hosts the malicious application used for simulating phishing attacks; the "client-application folder", containing the legitimate application's frontend and backend components; and the previously mentioned "realm-export-task6.json", "realmnames.txt", and "clientids.txt" files. The realmnames.txt file provides a dictionary of potential realm names for enumeration, while clientids.txt contains a list of client IDs to be tested during the attack simulation. The realm-export-task6.json file defines the configuration for the Keycloak realm to be used in the experiment.

After confirming the integrity and presence of these files, the next step involves setting up the Keycloak environment. A new realm is created within the Keycloak deployment Figure V—3(Successful creation of Realm), and the realm-export-task6.json file is imported to configure the realm with the necessary settings, as

illustrated in Figure V—2(Importing provided realm during a realm creation). This import process establishes the required authentication and authorization parameters, ensuring the environment aligns with the task's specifications. Upon successful creation of the realm, a user account is provisioned within this realm Figure V—4(User account provisioned) and assigned the role "myrole", as specified in the task requirements Figure V—5(Account assigned with a role). This role assignment is critical to enable the user account to interact with the protected resources in the client application, facilitating the subsequent testing and exploitation scenarios outlined in the project.

2. Normal flow of the application

To illustrate the standard operational flow of the client application, two terminal instances are opened within the client-application folder, which contains both the frontend and backend components of the application. Prior to launching these components, it is necessary to configure the relevant artifacts to utilize the IP address assigned to the virtual machine hosting the Keycloak server. As depicted in Figure V—6(Client-application configuration), the configuration process involves updating the application settings with the appropriate IP address. Subsequently, the frontend and backend components are launched by executing the npm start command in their respective terminals Figure V—7(Launching the client application), initiating the application and enabling access to its functionality for further testing and validation.

Subsequently, a web browser is launched, and the URL http://localhost:8000 is accessed, displaying the webpage as illustrated in Figure V—8(Welcome page for the client application). Authentication is performed using the credentials of the user account previously established within the newly created corporate realm. Upon successful login, the "Retrieve Health Record" button is selected Figure V—9(Client application after logging in), which presents the health record details associated with the created user in Figure V—10 (Output for Retrieve health record), enabling verification of the application's functionality and access to protected resources.

3. Service detection and information gathering

In this phase of the security testing process, the role of an attacker is assumed to conduct information gathering and identify potential vulnerabilities in the Keycloak server. The Nmap tool is employed to perform a service version scan to detect open ports and associated services that could be exploited. The following command is executed to enumerate open ports on the server:

```
$nmap -sV 192.168.56.101
```

The scan results, as illustrated in Figure V—11(Nmap Command and Output), reveal multiple open ports, one closed port, and one unidentified port. Analysis of the output, as depicted in Figure V—11(Nmap Command and Output), identifies an open HTTP port, which represents a potential vulnerability in the server configuration, susceptible to further exploitation attempts.

4. Discover all the OAuth endpoints on Keycloak

Following the identification of the open HTTP port, the attacker's objective is to exploit this vulnerability by enumerating the available endpoints within the Keycloak server. To achieve this, a Python script for realm enumeration is developed, as illustrated in Figure V—12(Realm enumerator python code) and Figure V—13(Output for Realm enumerator), which systematically identifies valid realms within the Keycloak deployment. Upon successful enumeration of the realms, the corresponding endpoints for the identified realms are retrieved, as depicted in the subsequent figures, enabling further targeting for the attack.

1. Figure V—14(Realm found as master)
2. Figure V—15(Realm found as customer)
3. Figure V—16(Realm found as corporate)

After identifying the realm, we use the Dirb tool Figure V—17(Dirb command help) Figure V—18(Dirb command on the identified IP address and its port) to check valid requests for the endpoints present in the realms that we are trying to check and the output that is generated will be used for to craft a ClientID enumeration Figure V—19(Output for dirb command).

5. ClientID enumeration

Upon identifying the available endpoints, the attacker seeks to enumerate the client IDs configured within the Keycloak deployment to pinpoint a vulnerable client that could facilitate unauthorized access to sensitive information, as observed during the normal application flow. To accomplish this, the attacker develops a Python script designed to systematically iterate through the sign-in page of the Keycloak server, identifying publicly accessible client IDs that can be used for authentication, thereby enabling potential exploitation of the system.

The script, presented in Figure V—20(Python code part 1 for clientID enumeration) Figure V—21(Python code part2 for clientID enumeration), reads a list of potential client IDs from clientids.txt and constructs authentication URLs for the corporate realm, appending necessary OAuth 2.0 parameters such as response_type, redirect_uri, state, scope, nonce, and code_challenge. It uses the subprocess module to execute cURL commands, checking the HTTP response status code for each client ID. A status code of 200 indicates a valid client ID, which is then logged and saved to valid_clientids.txt. A 0.1-second delay is introduced between requests to prevent server overload, ensuring the enumeration process is both effective and non-disruptive.

Upon executing the client ID enumeration script, the output provides a detailed log of the enumeration process, as shown in Figure V—22(Python ClientID enumeration output). The script iterates through each client ID listed in clientids.txt, printing a message for each attempt, such as "Checking clientID: {clientid}". For each client ID, the script indicates whether it is valid or invalid based on the HTTP response status code. A status code of 200 results in a message like "Valid clientID found: {clientid}", while a non-200 status code yields "Invalid clientID: {clientid} (status: {status_code})". Valid client IDs, such as patient-portal, are appended to a list and subsequently written to valid_clientids.txt Figure V—23(Output saved in a text file), with a final summary message confirming the completion of the enumeration and listing all valid client IDs, for example, "Valid clientIDs: patient-portal". This output enables the attacker to identify which client IDs can be exploited to access the Keycloak authentication flow, facilitating the next phase of the attack.

6. Exploiting the vulnerability

Having identified the vulnerable endpoint, the attacker leverages this endpoint within the malicious application to retrieve the health record previously displayed in Figure V—10 (Output for Retrieve health record). Before initializing the attacker's URL, the components of the attacker application are configured to align with the IP address of the Keycloak server, as illustrated in Figure V—24(Modified component of Attacker application) Figure V—25(Modified component of Attacker application 2). The crafted attacker URL is then embedded into the redirect URI, which, instead of directing the user to the original page observed earlier, redirects to the attacker's URI Figure V—26(Attacker's Webpage), thereby facilitating unauthorized access to the protected resources.

The output reveals that upon clicking the "Click here" hyperlink on the attacker's webpage, the health record is displayed, mirroring the data retrieved by the legitimate client application during normal operation. This unauthorized access confirms the successful exploitation of the vulnerability, exposing the same sensitive health record observed when logging into the account via the standard authentication flow as shown in Figure V—27(Health record retrieved on Attacker webpage). In this scenario, the client application had an active user session, eliminating the need for re-authentication to display the health record. This seamless access was

facilitated by the Keycloak server accepting the existing token, which enabled the attacker's application to retrieve the sensitive data without requiring additional login credentials.

7. Brute force Attempts

With access to the enumerated realm names and client IDs, the attacker proceeds to test the feasibility of directly fetching an access token from the Keycloak server, exploiting a known security weakness in OAuth 2.0's password grant type, which permits token retrieval using user credentials. Initially, a Python script is developed to brute-force the user credentials, systematically attempting various passwords from a predefined list to identify the correct credentials for the user account (s224784353) created earlier in the task. The script, presented in Figure V—28(Python code for brute forcing the user credential) Figure V—29(List of usernames to try using the python code) Figure V—30(List of passwords to try using the python code), sends HTTP POST requests to the Keycloak token endpoint, iterating through potential passwords until a successful authentication is achieved, indicated by a 200 status code Figure V—31(Output after brute-forcing).

Upon identifying the correct credentials through this brute-force method, a cURL command Figure V—32(Curl command for accessing the token) is employed to send an HTTP POST request to the appropriate Keycloak token endpoint.

The request targets the URL http://192.168.56.101:8080/auth/realms/corporate/protocol/openid-connect/token, utilizing the previously identified client ID patient-portal, the grant type password, and the now-verified credentials of the user account. This test aims to verify whether the access token can be successfully obtained, thereby exposing the vulnerability associated with the password grant type in the Keycloak configuration.

After successfully identifying the user credentials through the brute-force script, the attacker manually tests the credentials by logging into the attacker's webpage at http://localhost:9000/. Using the username s224784353 and the discovered password, the attacker authenticates through the Keycloak login page redirected by the malicious application Figure V—33(Manual Login for attacker page). Upon successful login, the attacker clicks the "Click here" hyperlink on the attacker's page, which retrieves the health record previously observed in the legitimate client application, as shown in Figure V—27(Health record retrieved on Attacker webpage). This manual login confirms that the stolen credentials can be used to bypass security controls, granting unauthorized access to sensitive health data and further validating the severity of the exploited vulnerability.

III. Securing the Application

1. Fixing the misconfiguration for Redirect URI

In the Keycloak admin console at http://192.168.56.100:8080, we navigated to the corporate realm, selected the vuln client, and updated the Valid Redirect URIs field from a wildcard (*) to http://localhost:8000/* as shown in Figure V—34(Mitigation for Valid Redirect URI). This change was saved to ensure that only the legitimate client application at http://localhost:8000 could receive authorization codes.

The permissive wildcard URI allowed attackers to redirect authorization codes to their malicious application at http://localhost:9000, enabling phishing attacks when we were exploiting the vulnerability. By restricting the URI as illustrated in , we ensured that only the intended application could complete the OAuth 2.0 flow, preventing unauthorized access to sensitive health records and enhancing the system's security.

2. Require consent before the user is directly authenticated

To enhance user privacy and control, we configured Keycloak to require user consent before direct authentication in the corporate realm. In the Keycloak admin console at http://192.168.56.100:8080, we

navigated to Clients > vuln > Consent, enabled the Consent Required option as shown in Figure V—36(Enabling consent required for the client), and specified the scopes (e.g., openid, profile, email) that users must approve. This setting ensures that when users log in via the client application at http://localhost:8000, they are presented with a consent screen detailing the data the application will access, requiring explicit approval before authentication proceeds depicted in Figure V—37(Output for Curl Command after enabling consent required)

Without consent, applications could access user data without explicit permission, risking privacy violations and non-compliance with regulations like GDPR. Requiring consent empowers users to control their data, aligns with best practices for OAuth 2.0, and mitigates unauthorized data access, enhancing trust in the patient portal system.

3. Enable Brute-Force Detection

We enabled Brute Force Detection in Realm Settings > Security Defences, setting Maximum Login Failures to 5, Wait Increment to 60 seconds, Temporary Lockout Duration to 600 seconds, a 10-minute lockout, and a 1000ms quick login check. This was saved to activate account lockouts which is shown in Figure V—38(Brute force defence configuration on Keycloak)

Without this, attackers could repeatedly guess user credentials, as demonstrated in Brute-force attempts as seen in Figure V—31(Output after brute-forcing). The detection mechanism locked accounts after excessive failures Figure V—39(Brute-forcing with wrong passwords) , thwarting brute-force attempts and protecting user accounts, particularly critical for safeguarding sensitive health data Figure V—40(Account Lockout after brute-forcing wrong passwords).

4. Disable Resource Owner Password Credential (ROPC)

In the Keycloak admin console, under Clients > vuln > Capability Config, we disabled Direct Access Grants, which enabled the ROPC grant type, and ensured Standard Flow (Authorization Code Flow) was active. This change was saved to enforce the secure flow as seen in ROPC allowed attackers to submit credentials directly to the token endpoint, enabling brute-force attacks in Part 5. Disabling it forced the use of the Authorization Code Flow, which requires user interaction and redirect URI validation, preventing automated credential attacks and securing the authentication process.

5. Enabled Firewall restrictions

To bolster network security, we configured firewall rules to restrict traffic from the IP address 192.168.56.0/24 to only the necessary TCP ports (22, 8080, 8443, 443) used by the Keycloak system using the host's UFW firewall management tool as shown Figure V—42(Configuring with wrong IP to demonstrate UFW in work). We added rules to allow incoming connections from 192.168.56.101 to these ports as shown in Figure V—43(Correct UFW configuration with demonstration for the open ports) while dropping all other traffic from this IP using the following command:

$sudo ufw allow from 192.168.56.0/24 proto tcp to any port 22,443,8080,8443

This mitigation was critical to limit potential attack vectors from 192.168.56.101, which could be an attacker's IP attempting unauthorized access or enumeration (as seen in earlier task parts). By allowing only essential ports for SSH (22), Keycloak (8080, 8443), and HTTP/HTTPS (80, 443), we minimized the attack surface, protected sensitive health data, and ensured compliance with security best practices for the patient portal system.

6. Password policy and 2-factor authentication

In Realm Settings > Authentication > Password Policy, we set a minimum password length of 12 characters, required 1 special character, 1 digit, and 1 uppercase letter, enforced a 5-password history, and set a 90-day expiration Figure V—44(Password policy for the realm). These policies were saved to apply to all users since weak passwords were vulnerable to brute-forcing and guessing. A strong policy ensured users created robust credentials, reducing the risk of compromise and enhancing the security of patient accounts accessing health records.

After configuring the password policy, in Authentication > Flows, we selected the Browser flow, added a TOTP-based OTP Form execution, and set it to "Required" as shown in Figure V—45(Authentication flow updated to enable the 2FA). In Authentication > OTP Policy, we configured a 6-digit TOTP with a look-ahead window of 1 Figure V—46(OTP Policy for 2FA). Users were prompted to set up 2FA using OTPClient on login Figure V—47(Setting up 2FA for S224784353) Figure V—48(Using 2FA to login on the S224784353 account). Single-factor authentication was insufficient against credential theft. 2FA added a second layer, requiring a time-based code, ensuring that even stolen passwords couldn't grant access, significantly strengthening user authentication.

7. Token Hardening

In Realm Settings > Tokens, we set Access Token Lifespan to 5 minutes, Session Idle to 15 minutes, Session Max to 60 minutes as depicted in Figure V—49(Token Hardening Configurations). Long-lived tokens and unprotected code exchanges risked misuse if intercepted. Short lifespans minimized the window for token abuse, while revoking refresh tokens ensured invalidation, protecting health record access.

IV. Conclusion

This security testing exercise for the SIT736 IAM and Physical Security course successfully demonstrated the exploitation of vulnerabilities within a Keycloak deployment and the subsequent implementation of robust countermeasures to secure the system. By simulating an attacker's workflow—encompassing service detection, realm and client ID enumeration, phishing URL crafting, and brute-force attacks—the project exposed critical misconfigurations, such as permissive redirect URIs, enabled Resource Owner Password Credential (ROPC) grant types, and the absence of brute-force protections, which allowed unauthorized access to sensitive health records. The successful retrieval of protected data, as evidenced by the attacker's ability to access the health record via the malicious application at http://localhost:9000/, underscored the severity of these vulnerabilities.

However, the application of targeted countermeasures, including restricting redirect URIs to http://localhost:8000/*, disabling ROPC, enabling brute-force detection with a 5-attempt lockout policy, requiring user consent, enforcing firewall restrictions on ports 22, 8080, 8443, and 443, implementing strong password policies with 2FA, and hardening token lifespans to 5 minutes, effectively mitigated these risks. These measures prevented further unauthorized access, as demonstrated by failed attack attempts post-mitigation e.g., Figure V—40(Account Lockout after brute-forcing wrong passwords). The findings highlight the critical importance of adhering to OAuth 2.0 best practices and maintaining rigorous Keycloak configuration management to safeguard sensitive data against phishing, enumeration, and credential-based attacks, providing actionable insights for enhancing the security of IAM systems in real-world deployments.

V. Appendix A

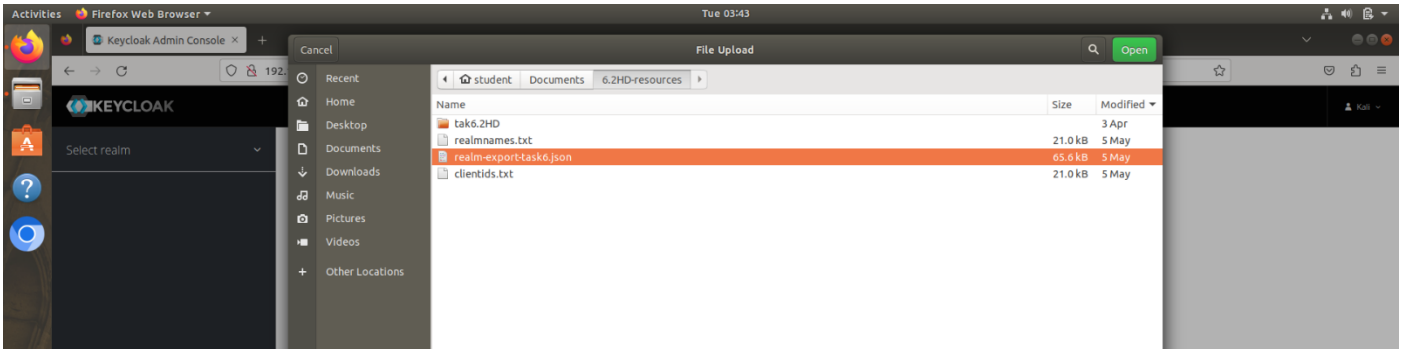

Figure V—1(Unzipping the External Resource)

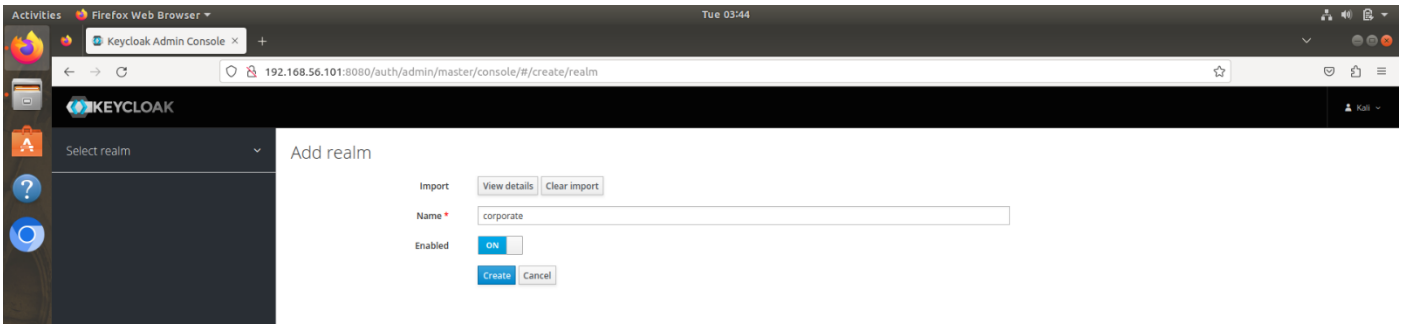

Figure V—2(Importing provided realm during a realm creation)

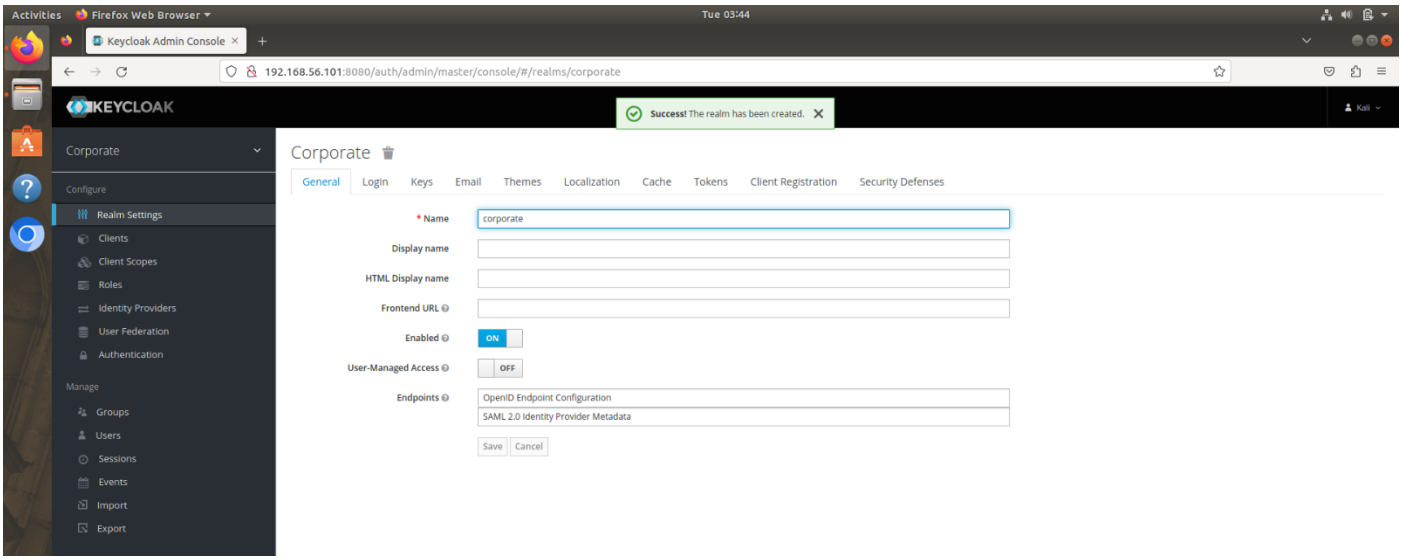

Figure V—3(Successful creation of Realm)

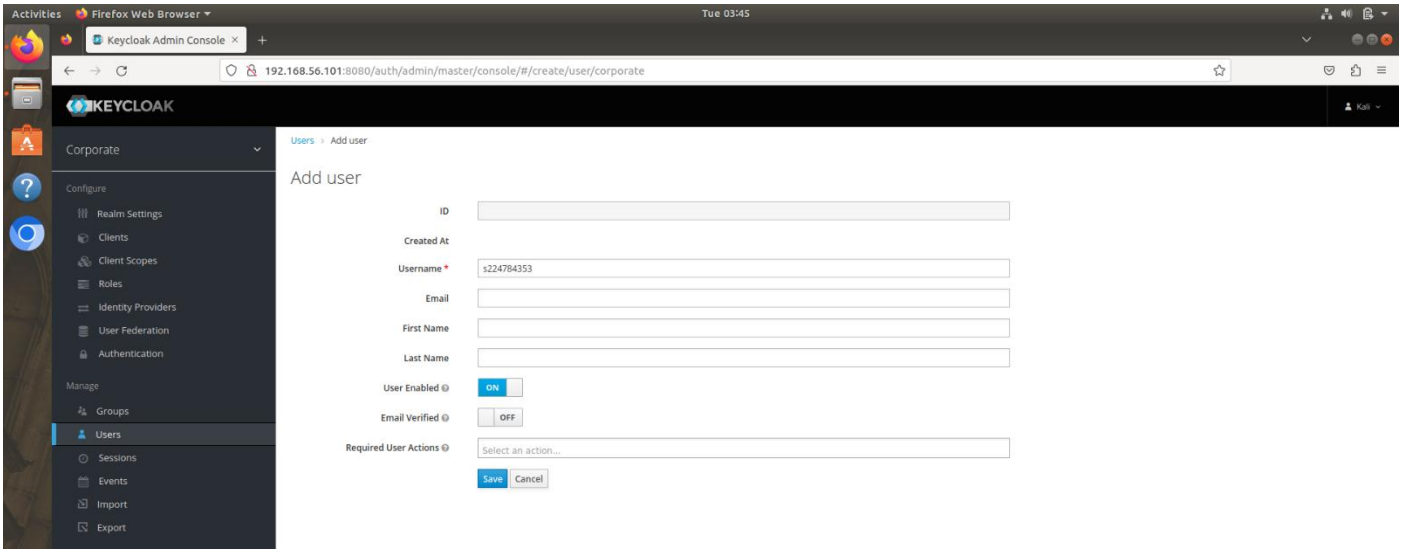

Figure V—4(User account provisioned)

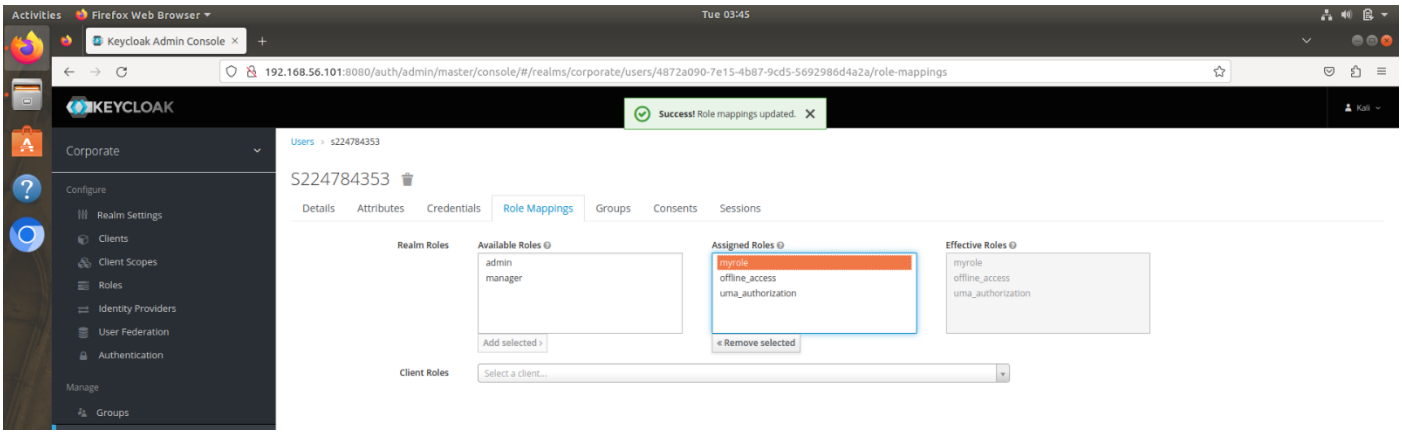

Figure V—5(Account assigned with a role)

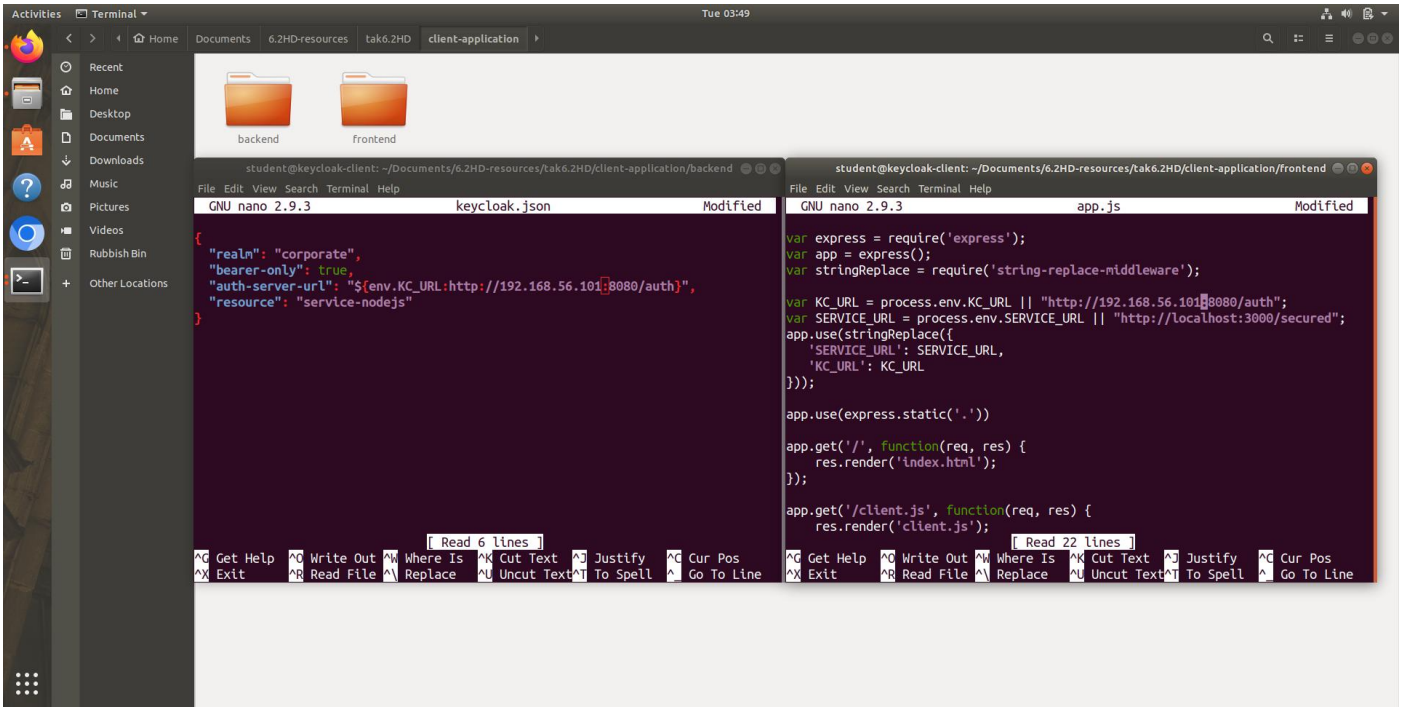

Figure V—6(Client-application configuration)

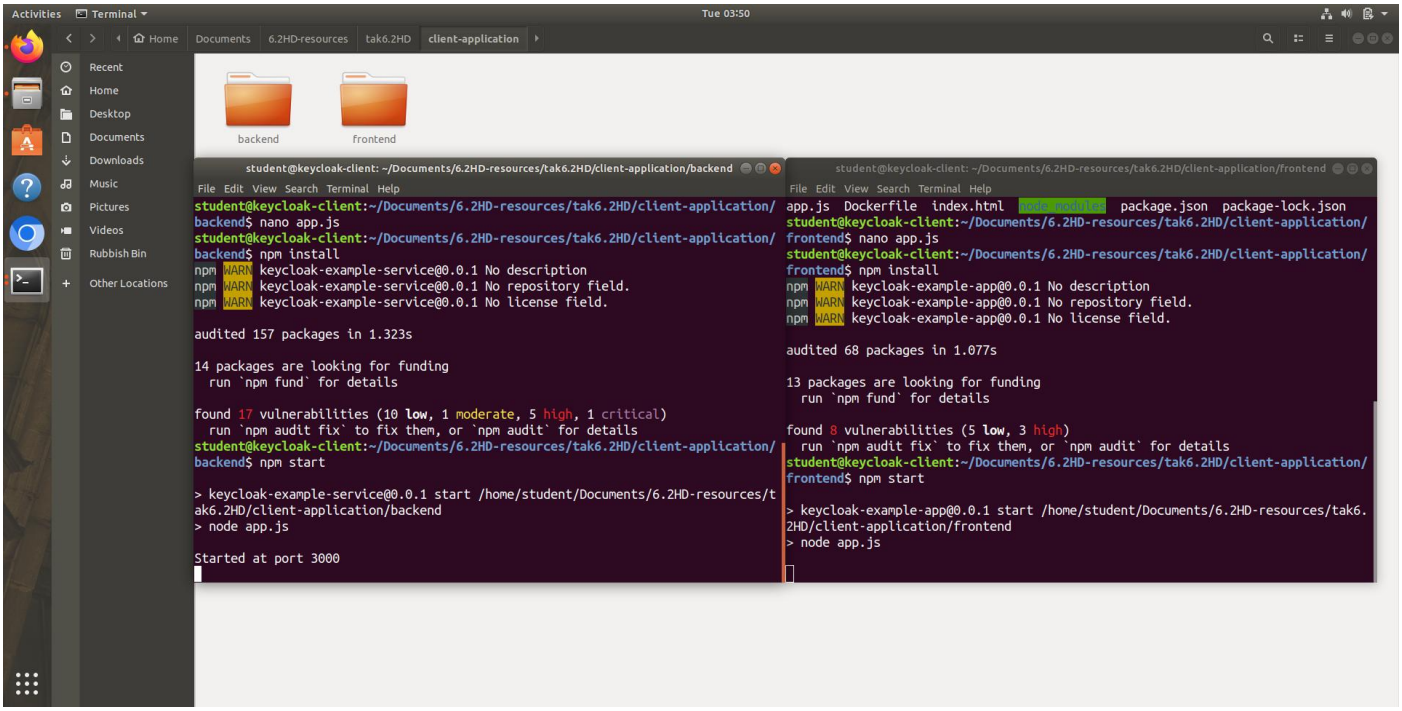

Figure V—7(Launching the client application)

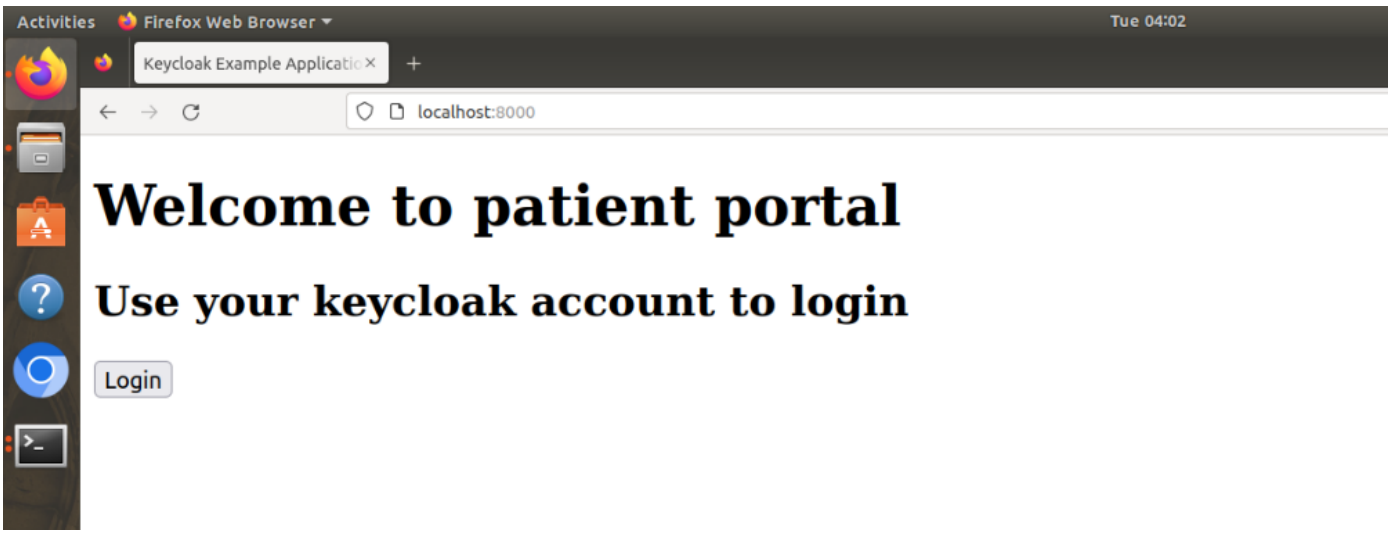

Figure V—8(Welcome page for the client application)

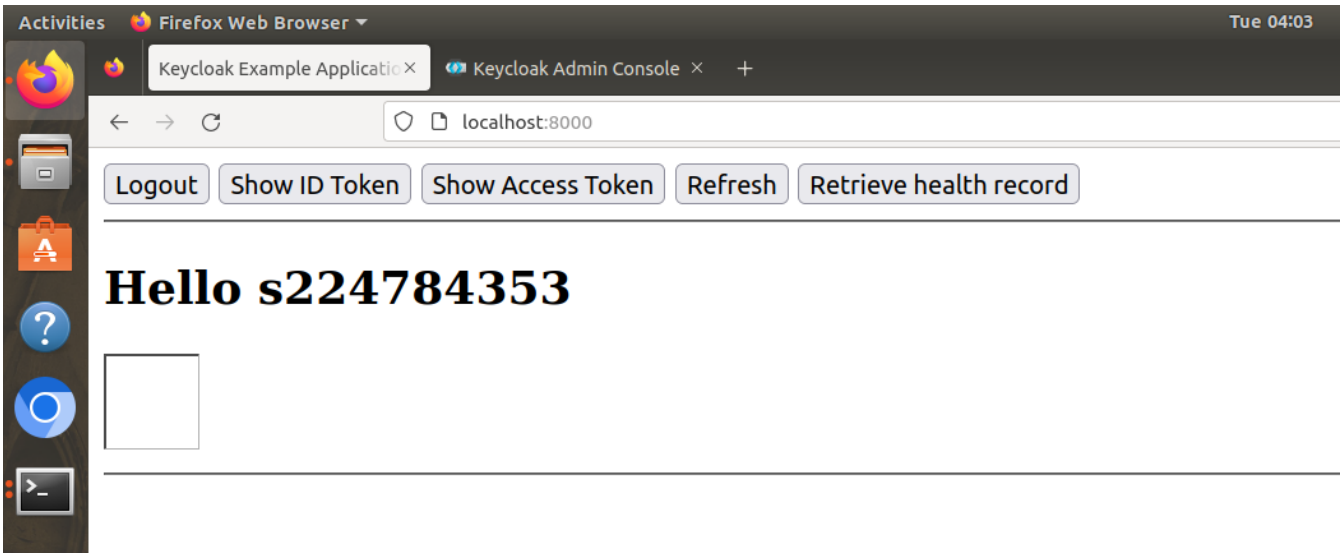

Figure V—9(Client application after logging in)

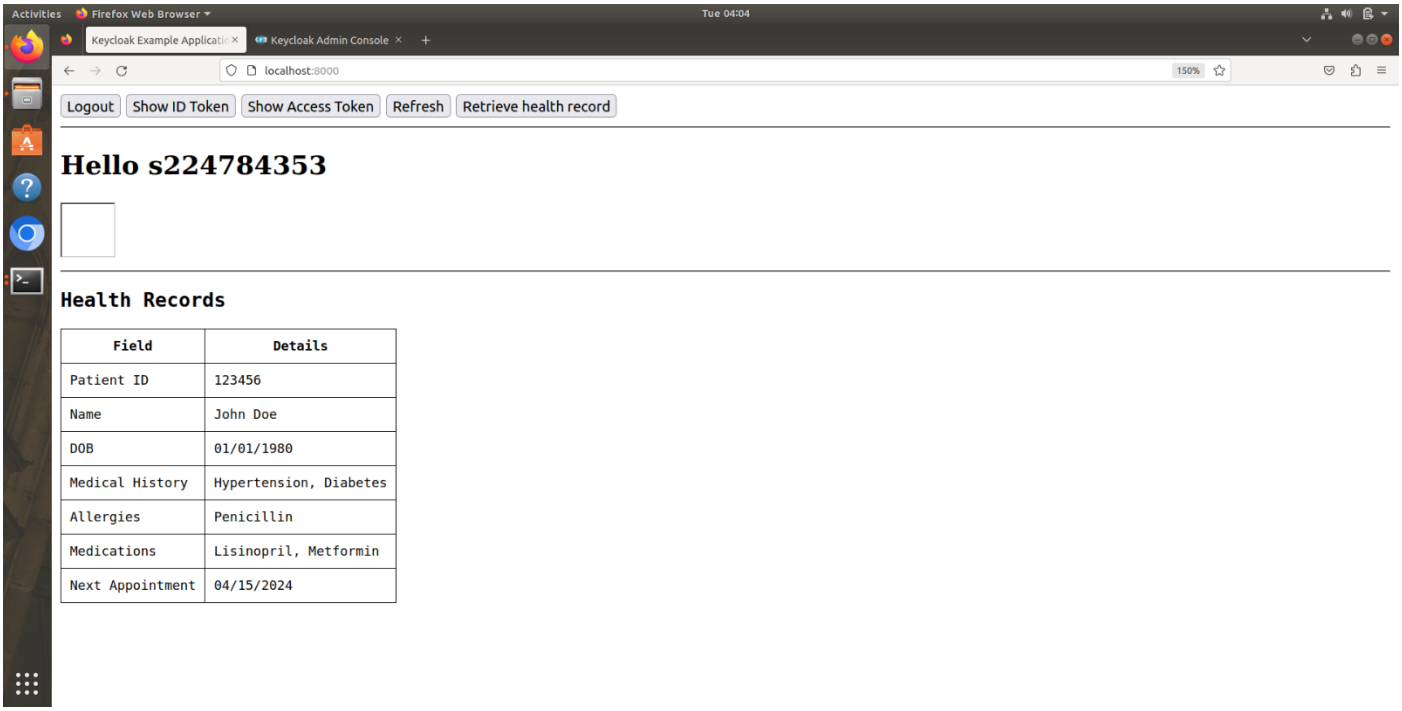

Figure V—10 (Output for Retrieve health record)

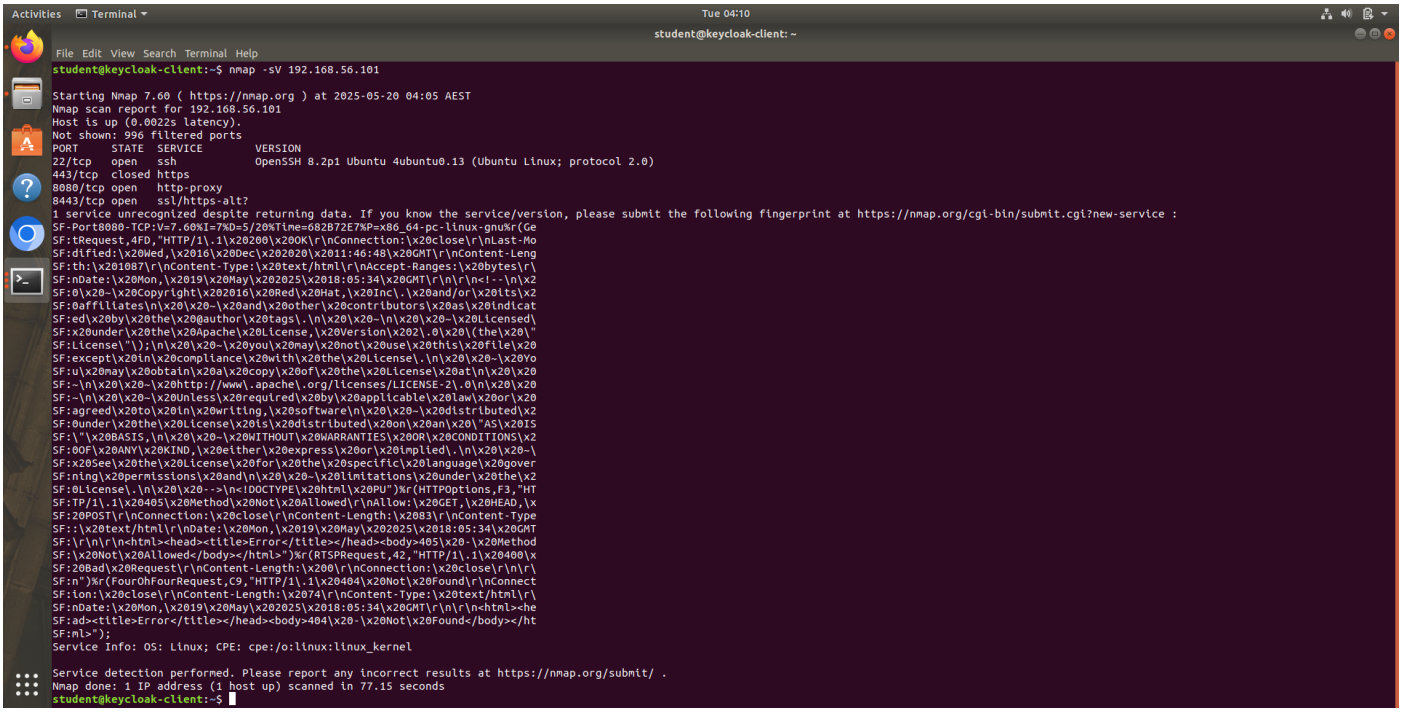

Figure V—11(Nmap Command and Output)

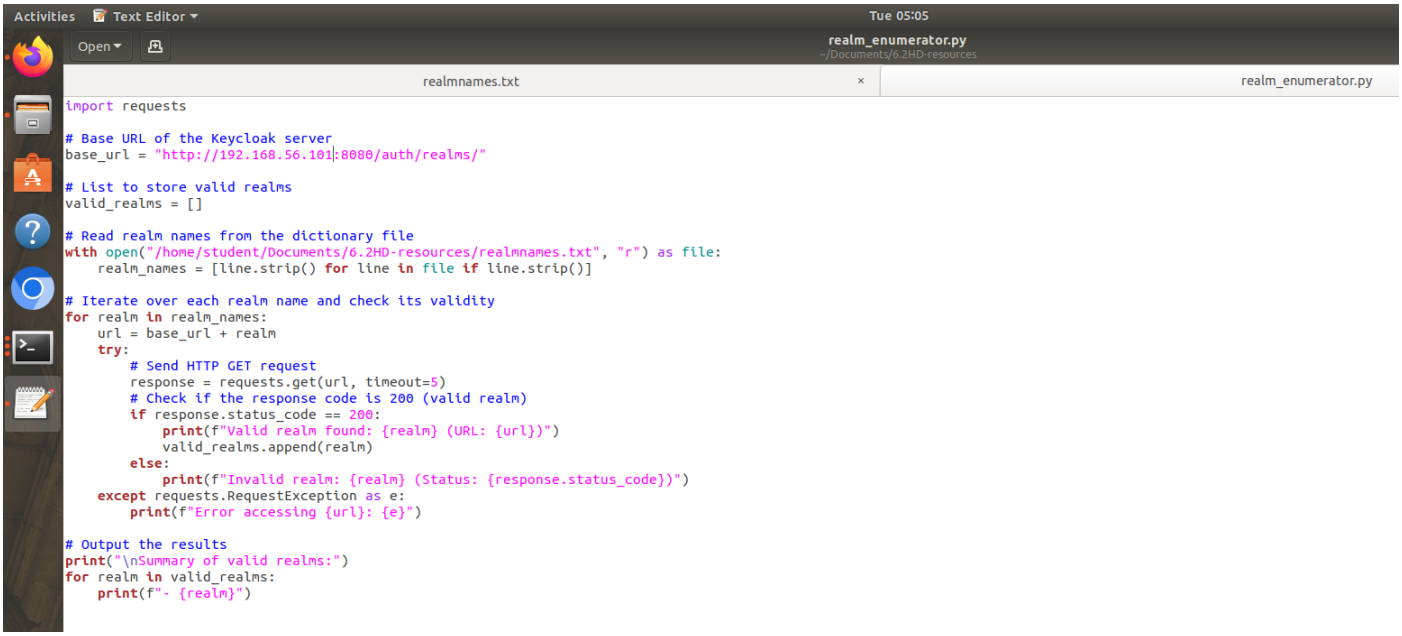

Figure V—12(Realm enumerator python code)

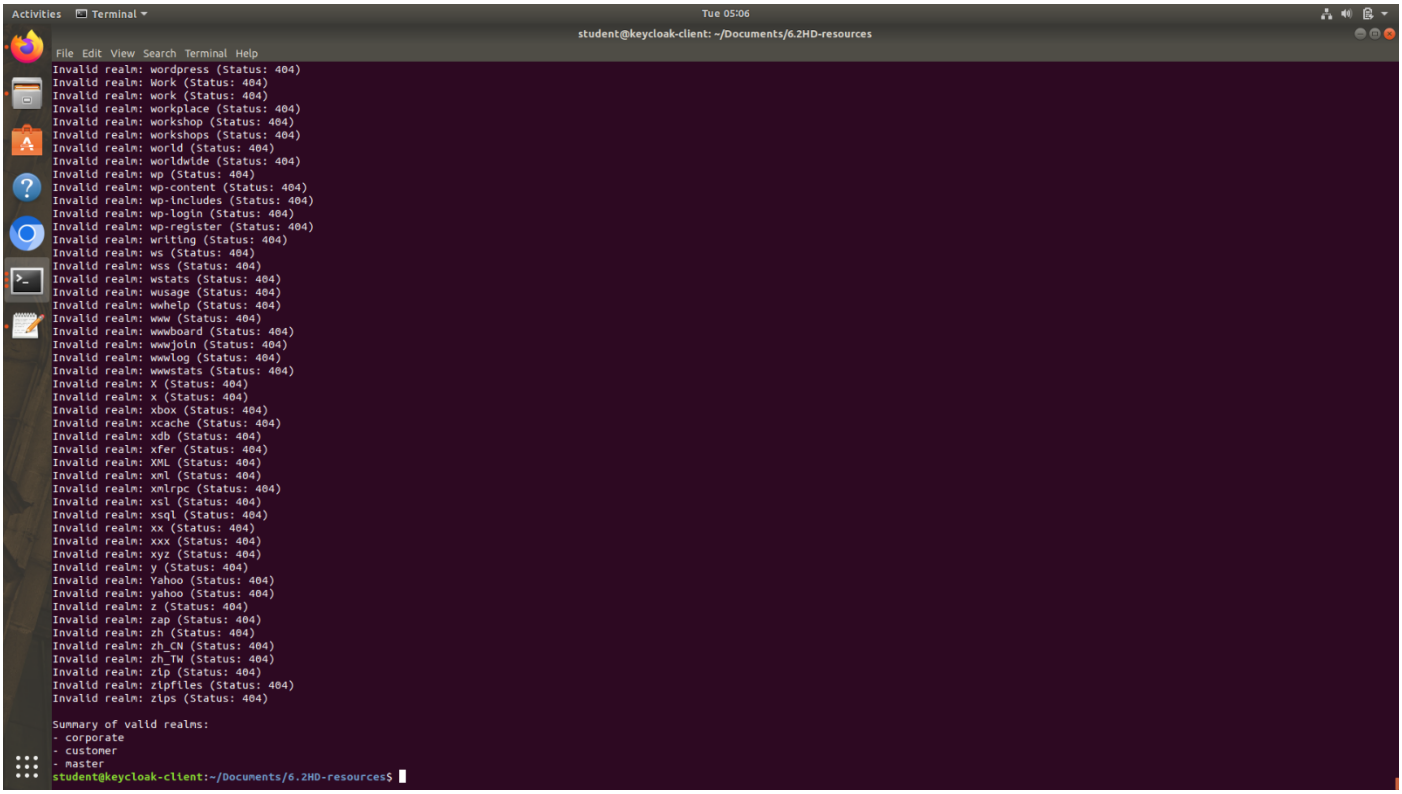

Figure V—13(Output for Realm enumerator)

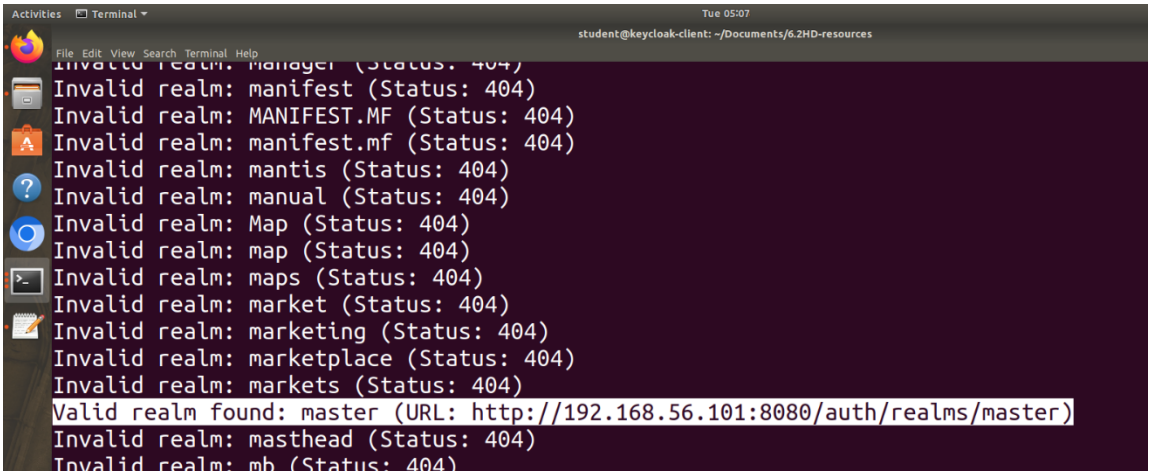

Figure V—14(Realm found as master)

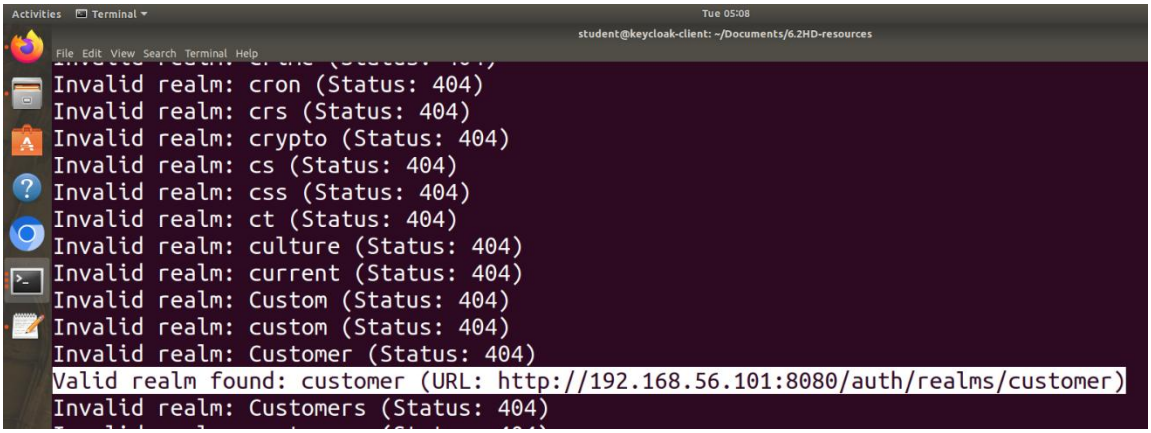

Figure V—15(Realm found as customer)

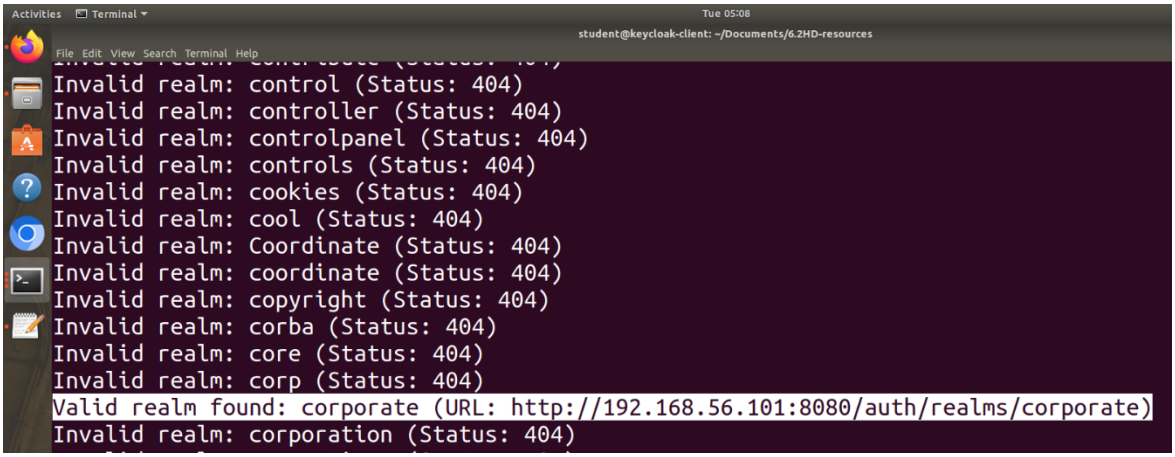

Figure V—16(Realm found as corporate)

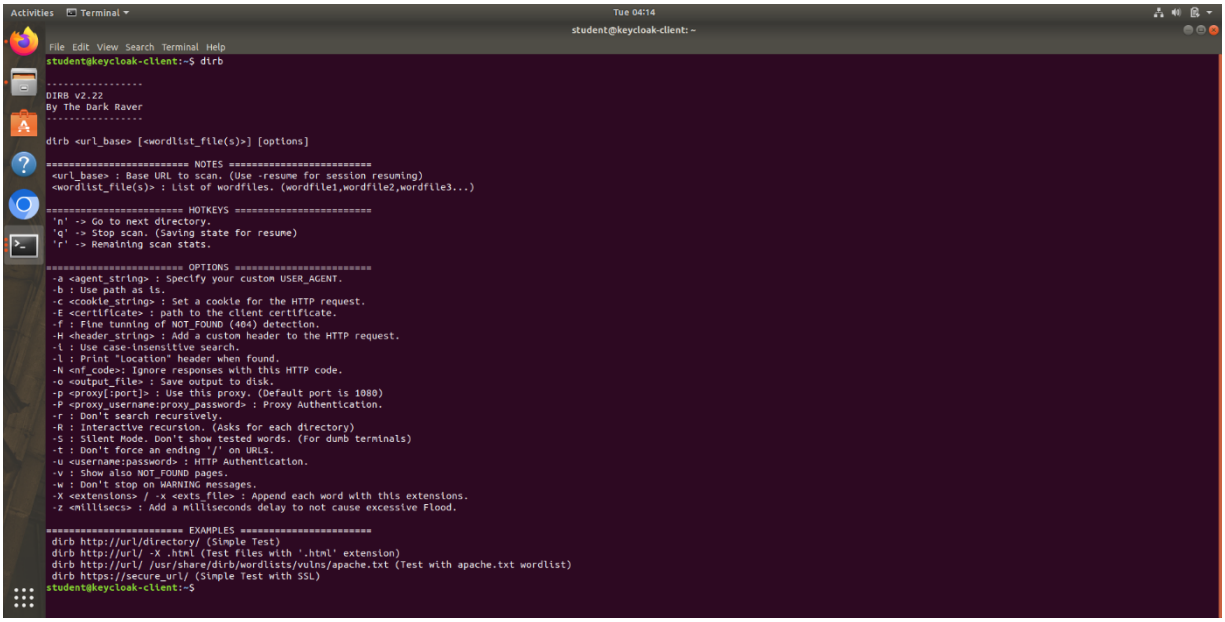

Figure V—17(Dirb command help)

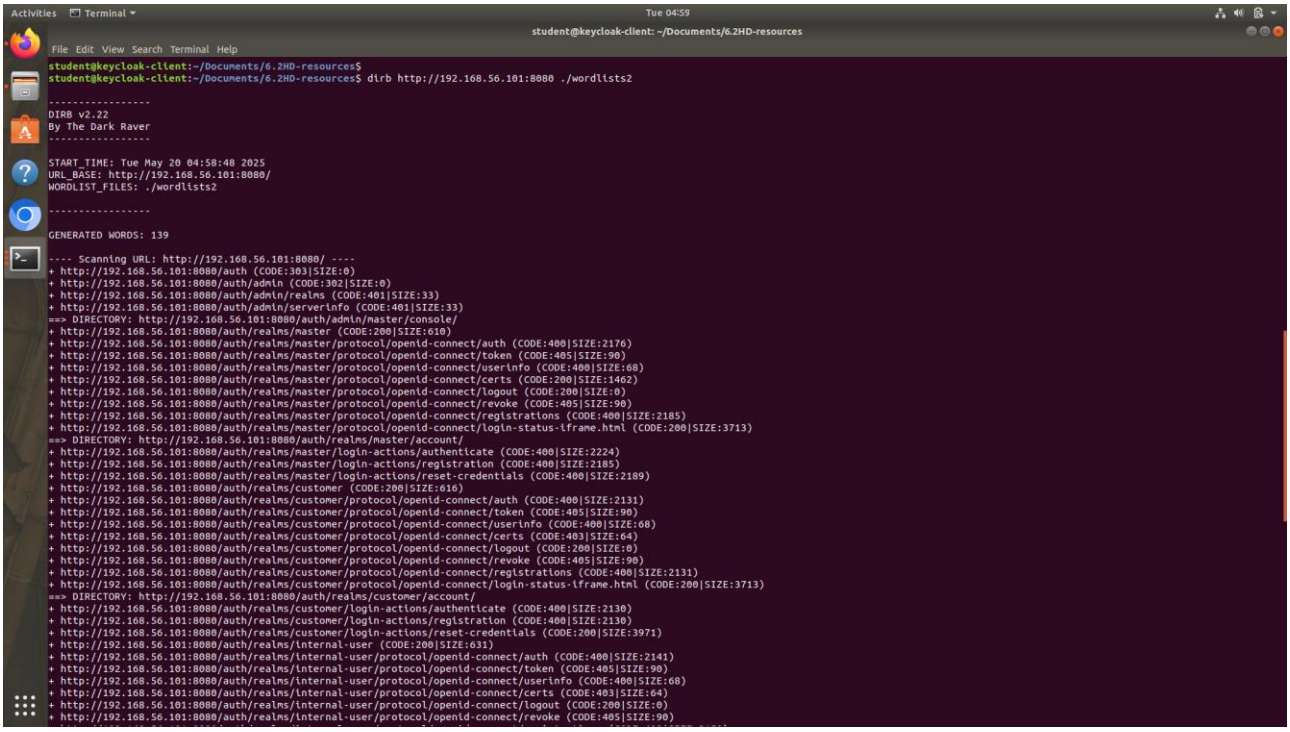

Figure V—18(Dirb command on the identified IP address and its port)

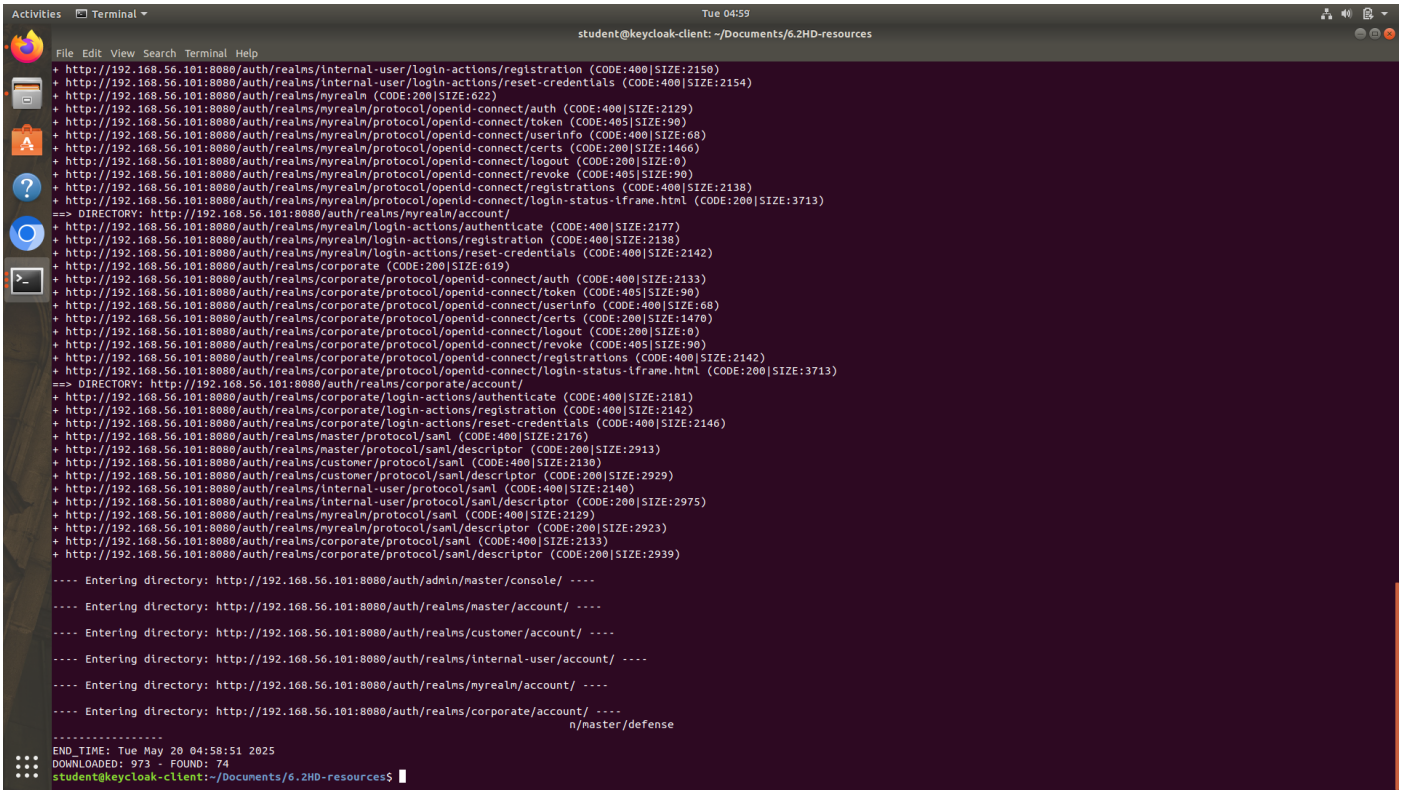

Figure V—19(Output for dirb command)

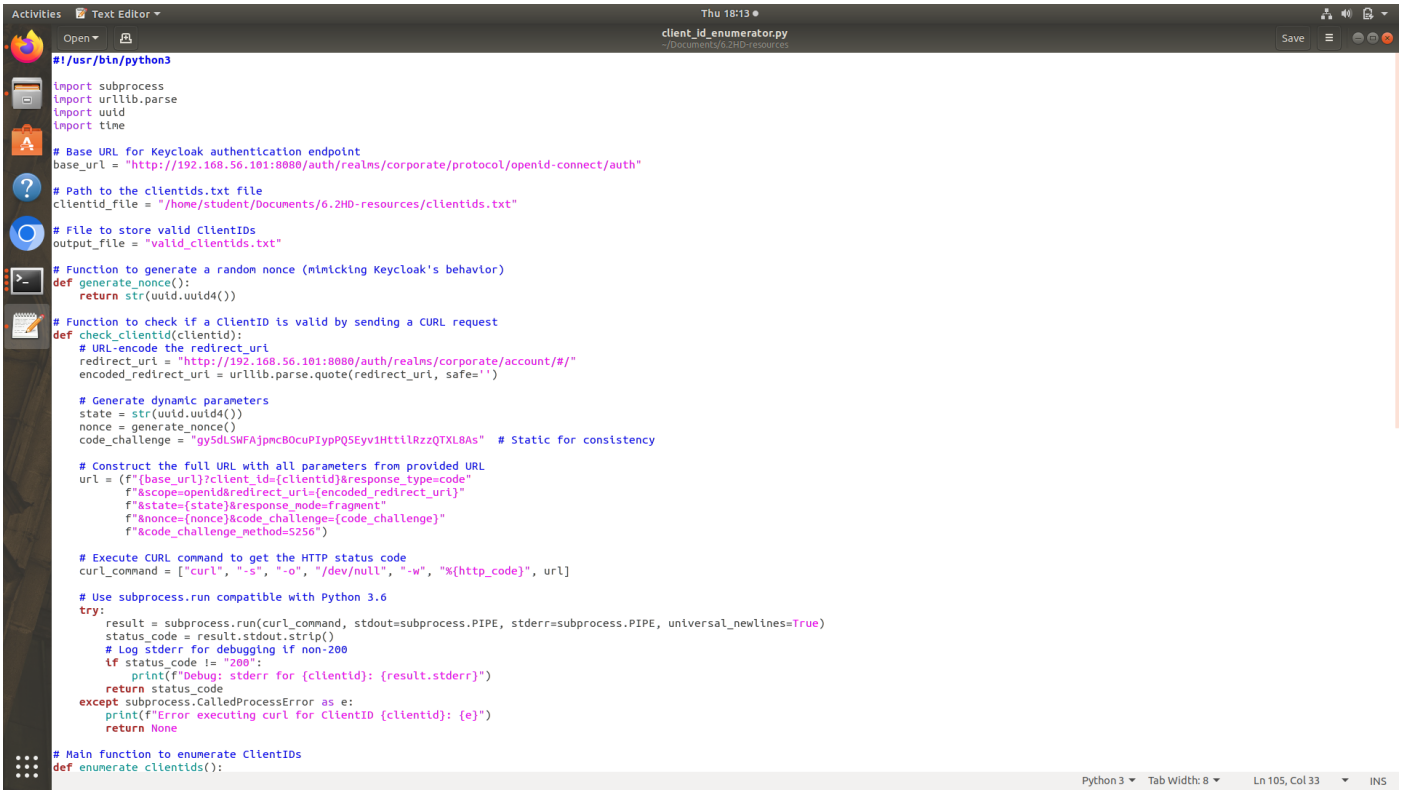

Figure V—20(Python code part 1 for clientID enumeration)

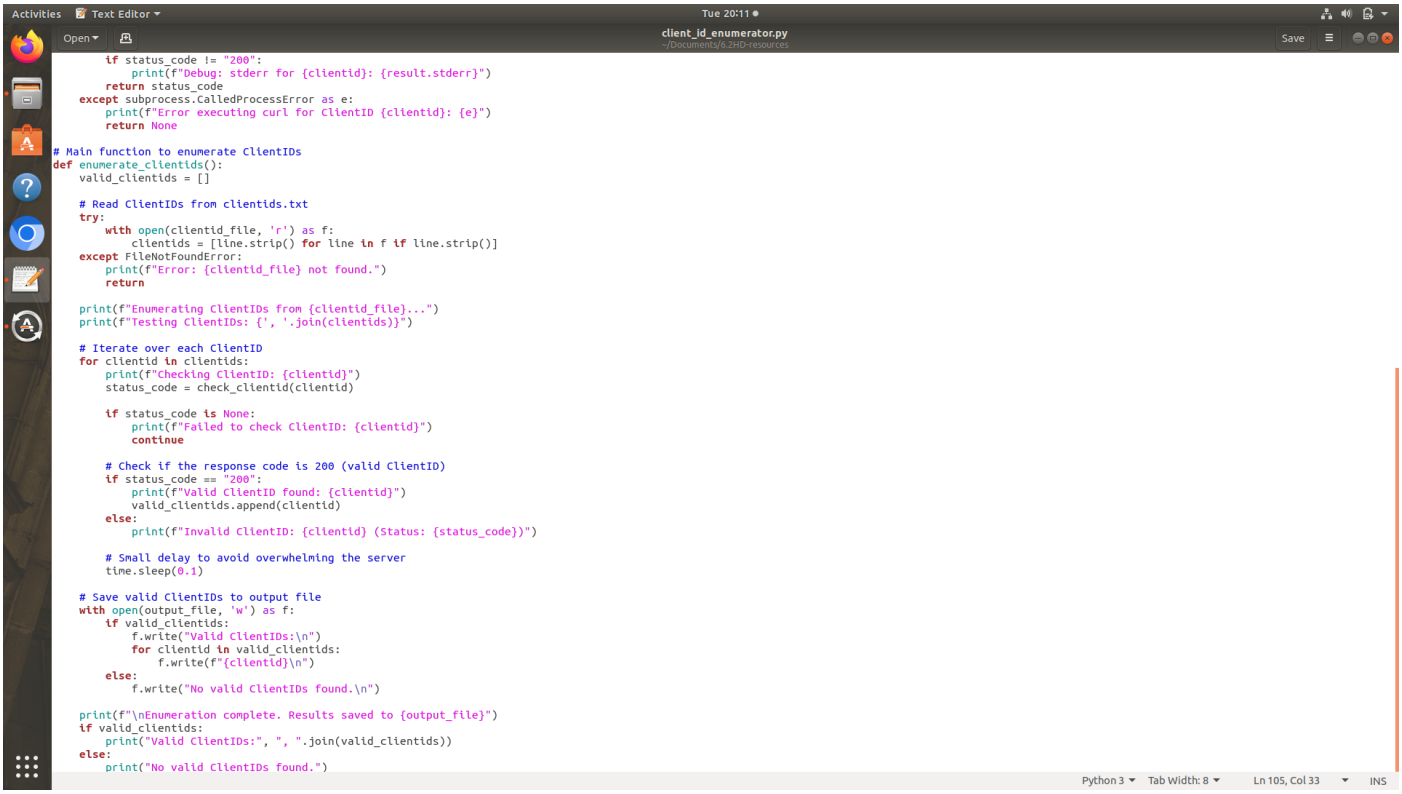

Figure V—21(Python code part2 for clientID enumeration)

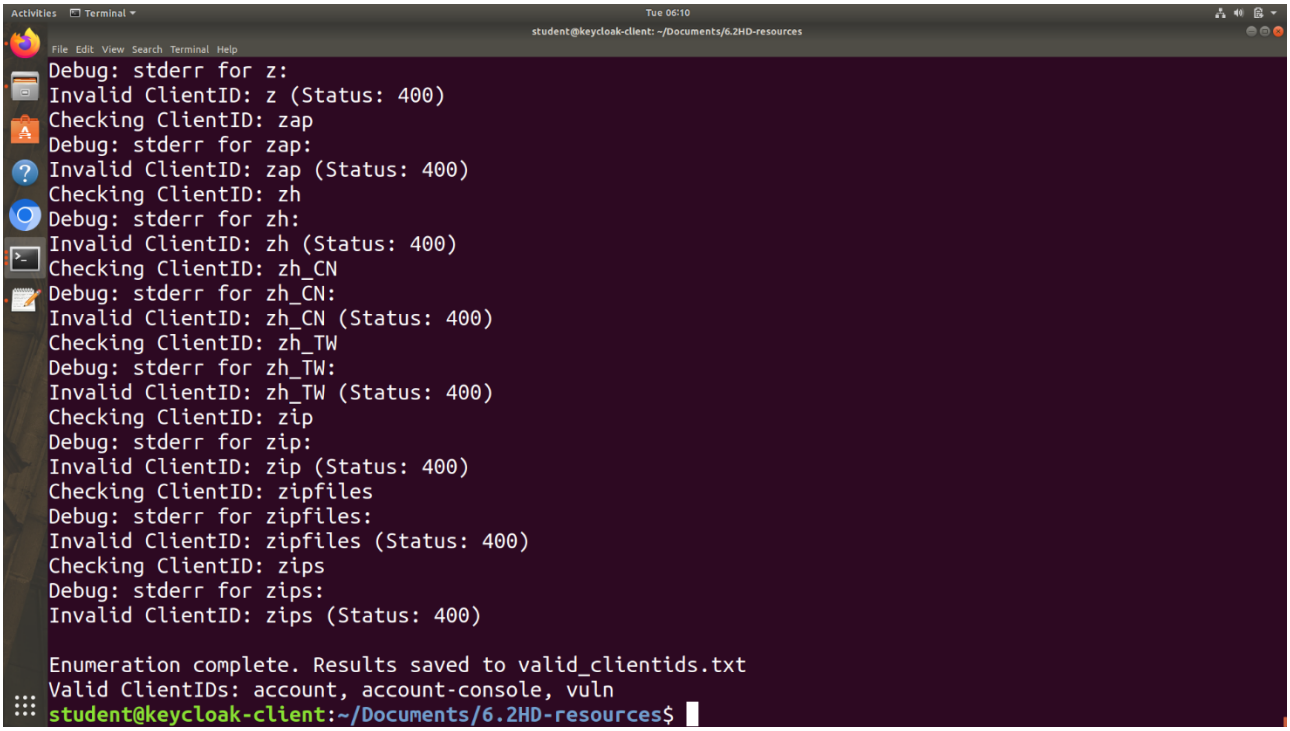

Figure V—22(Python ClientID enumeration output)

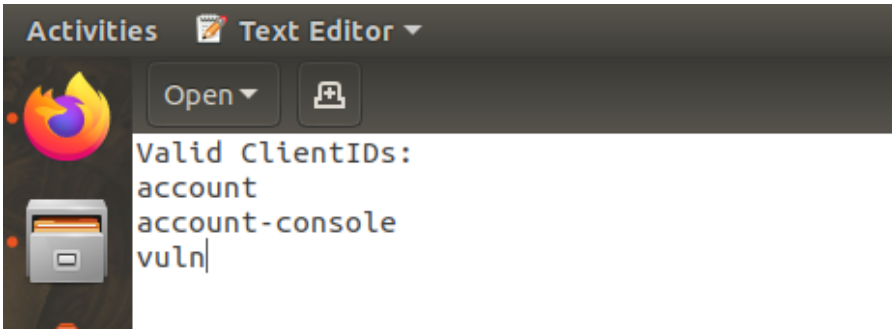

Figure V—23(Output saved in a text file)

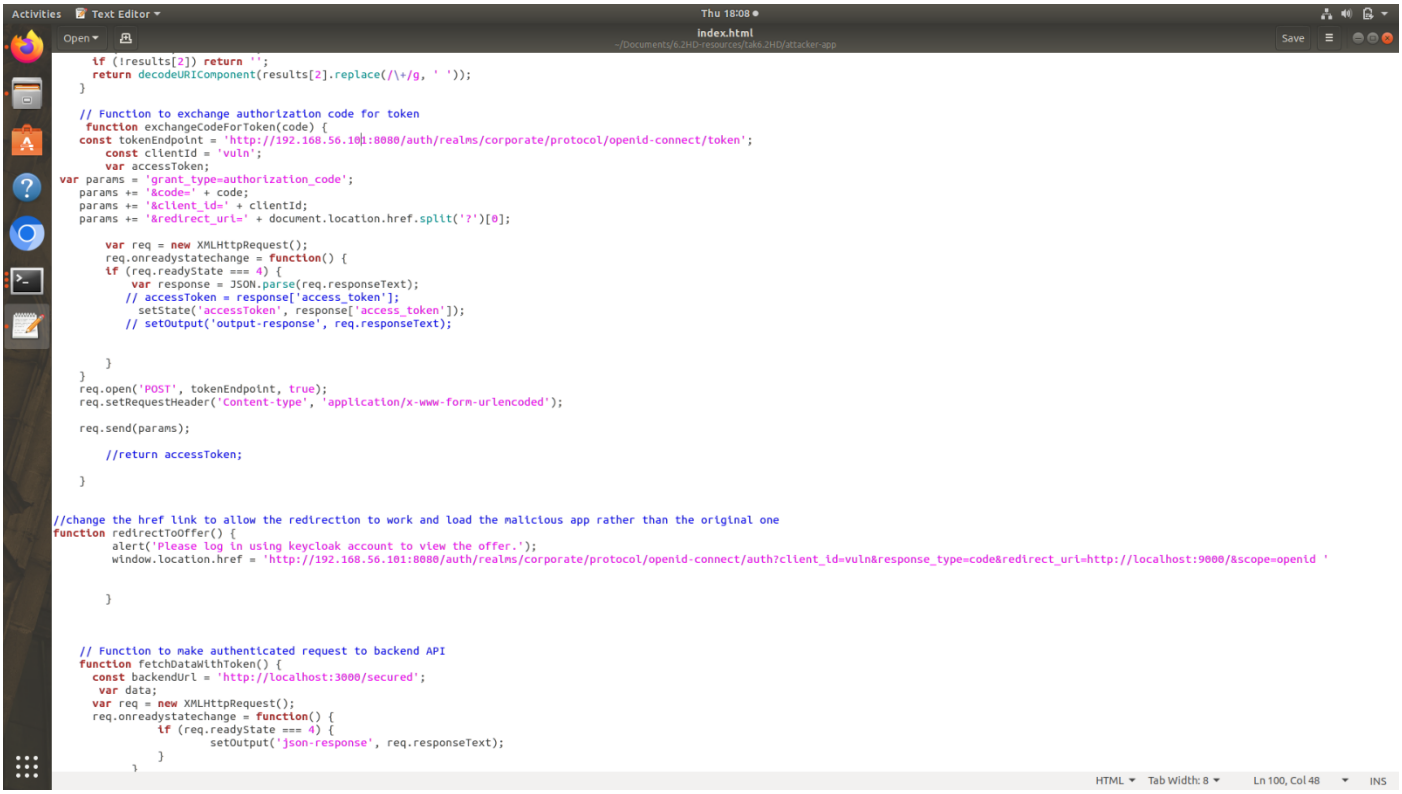

Figure V—24(Modified component of Attacker application)

Figure V—25(Modified component of Attacker application 2)

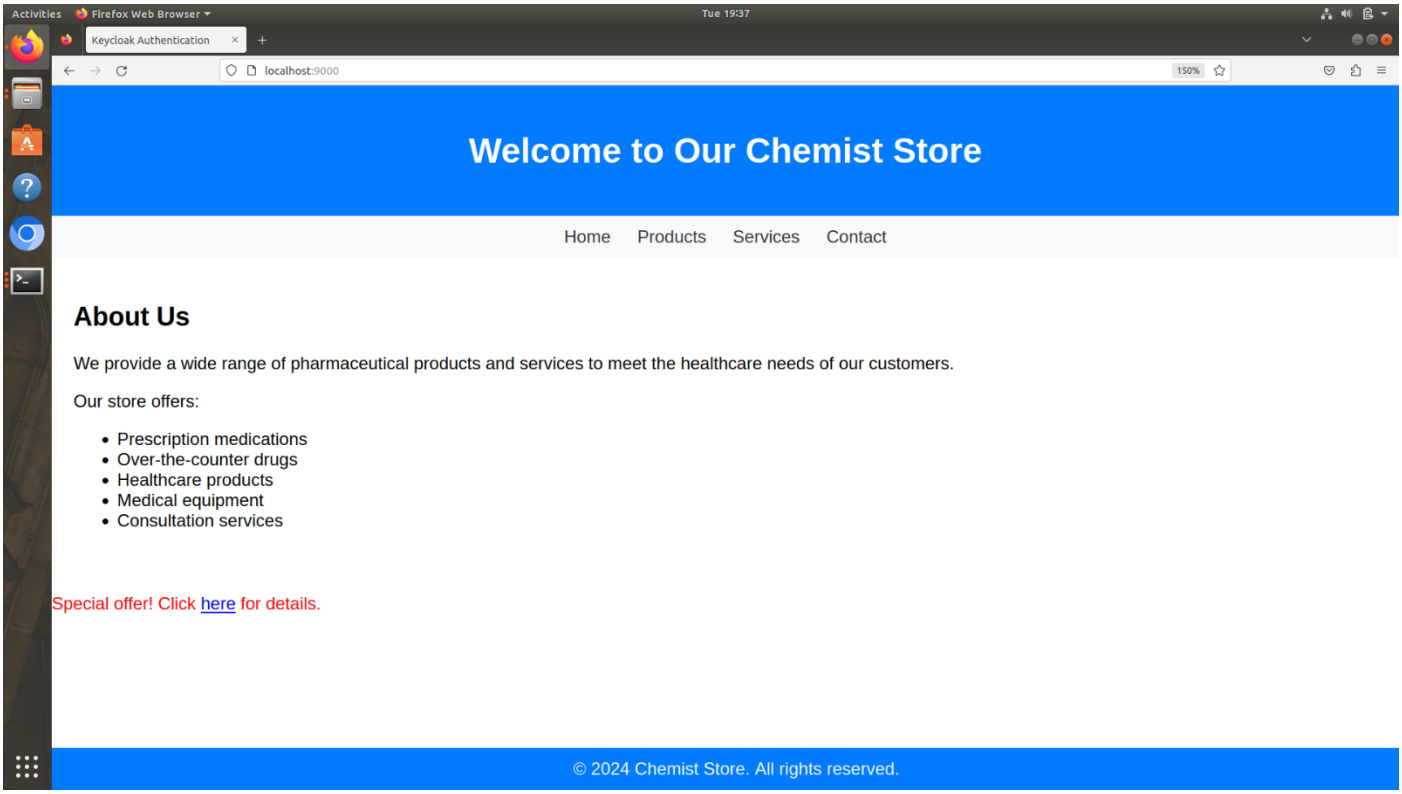

Figure V—26(Attacker's Webpage)

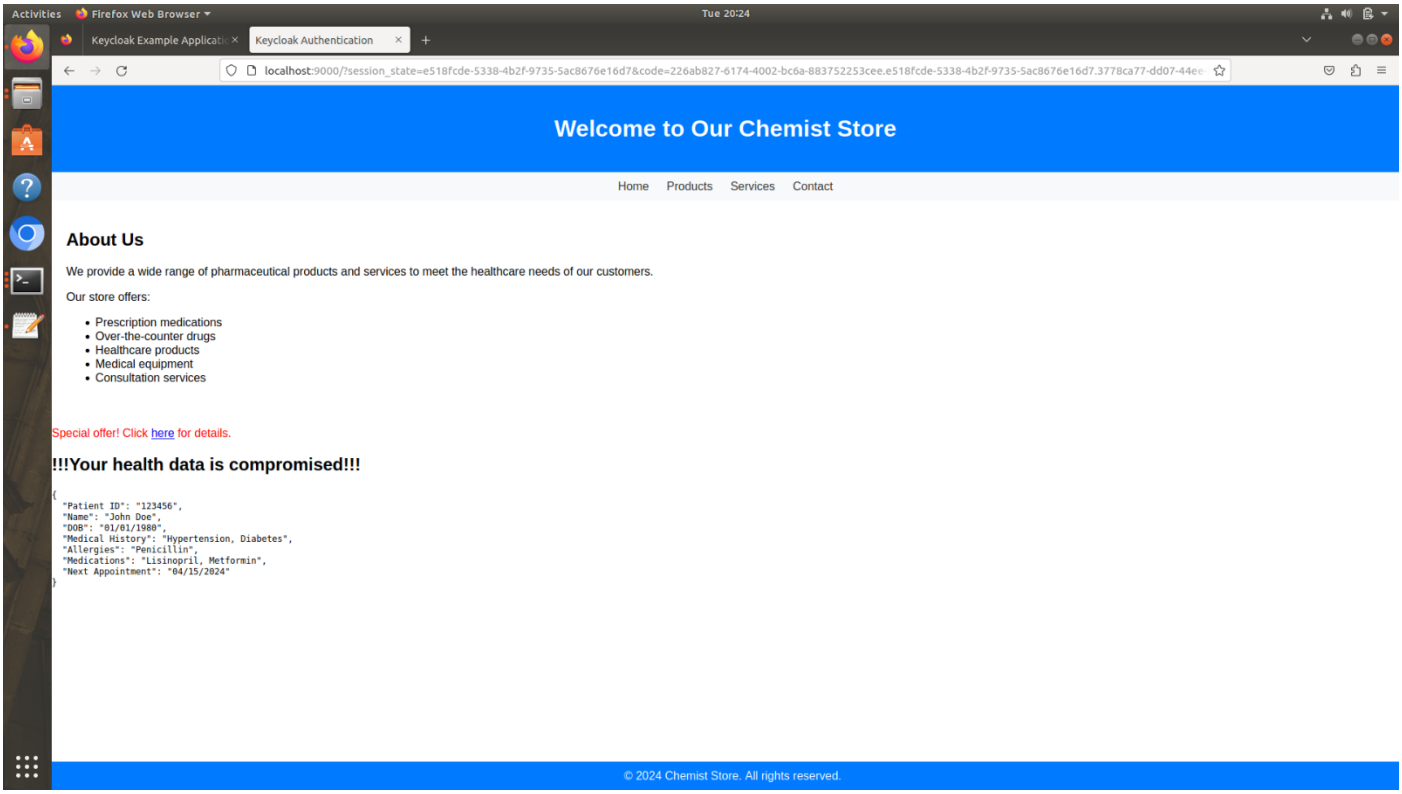

Figure V—27(Health record retrieved on Attacker webpage)

Figure V—28(Python code for brute forcing the user credential)

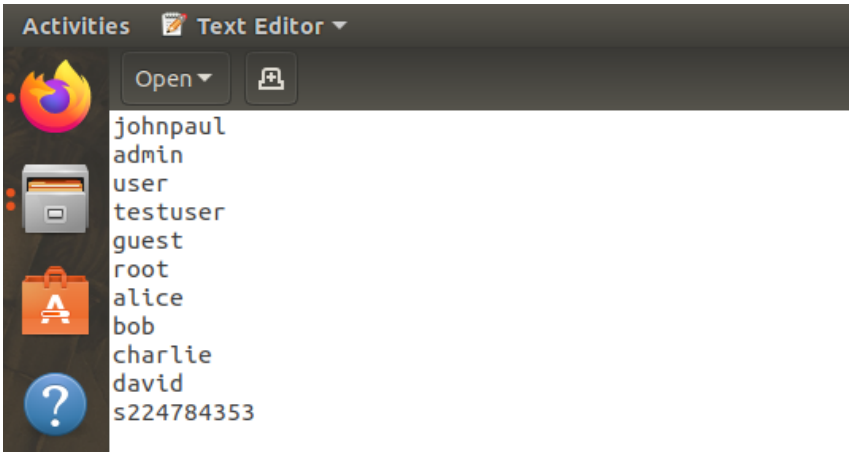

Figure V—29(List of usernames to try using the python code)

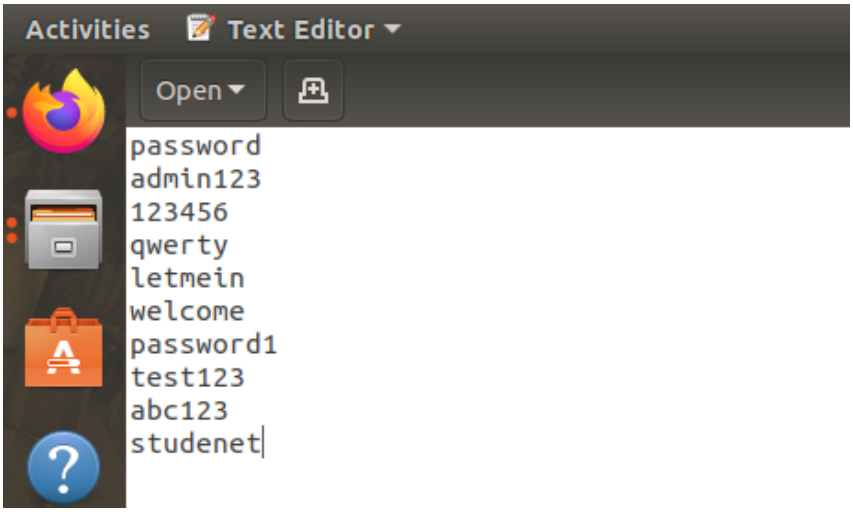

Figure V—30(List of passwords to try using the python code)

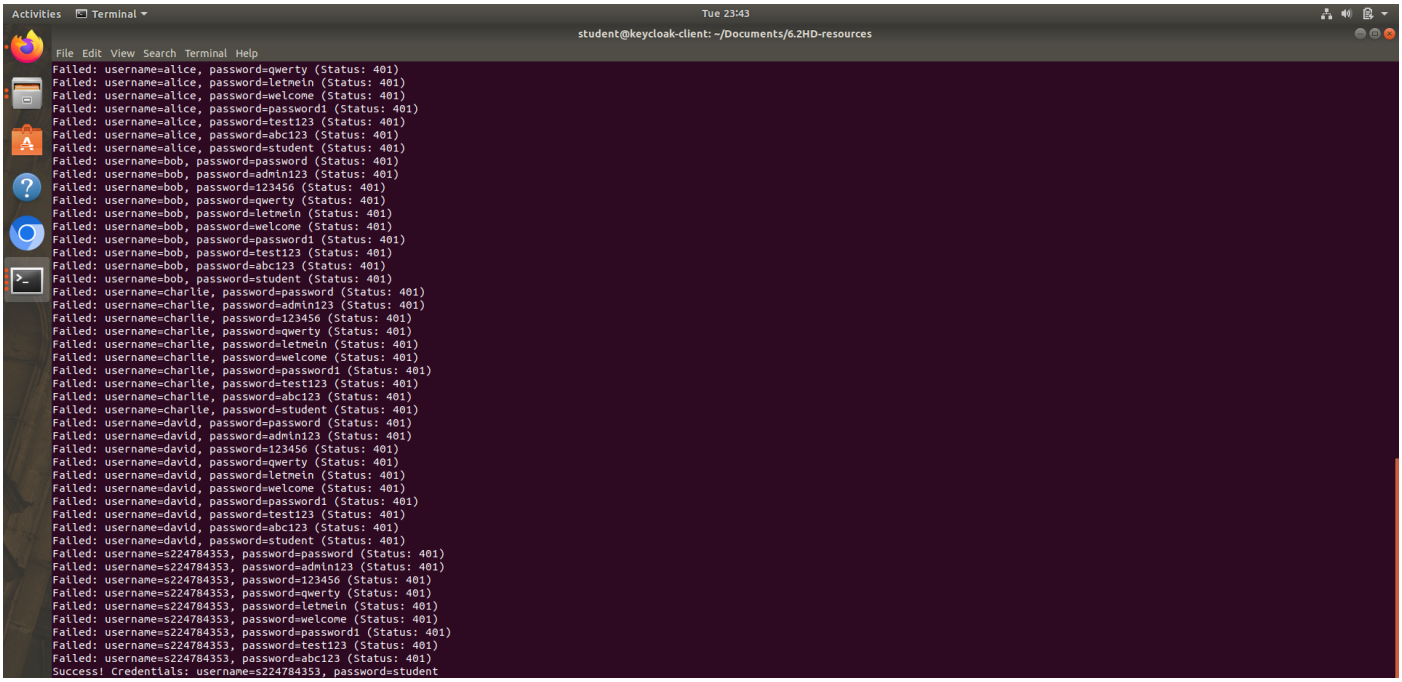

Figure V—31(Output after brute-forcing)

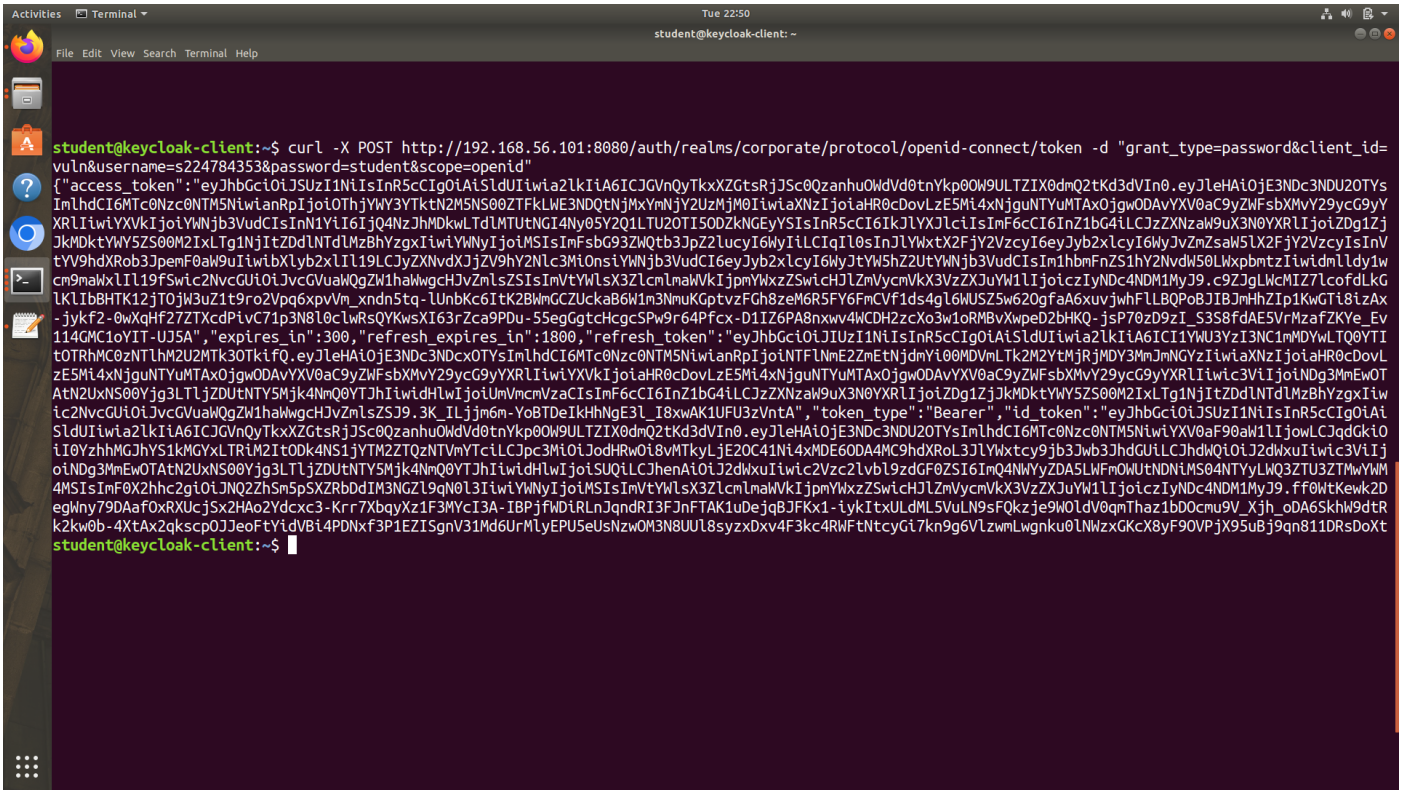

Figure V—32(Curl command for accessing the token)

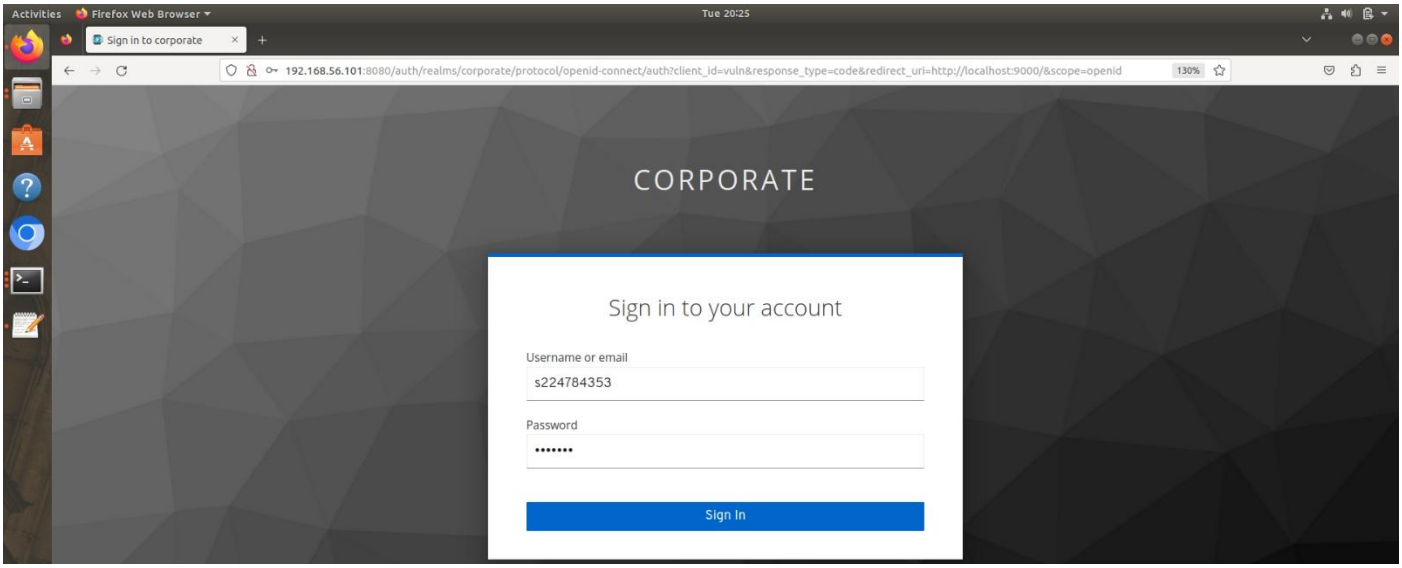

Figure V—33(Manual Login for attacker page)

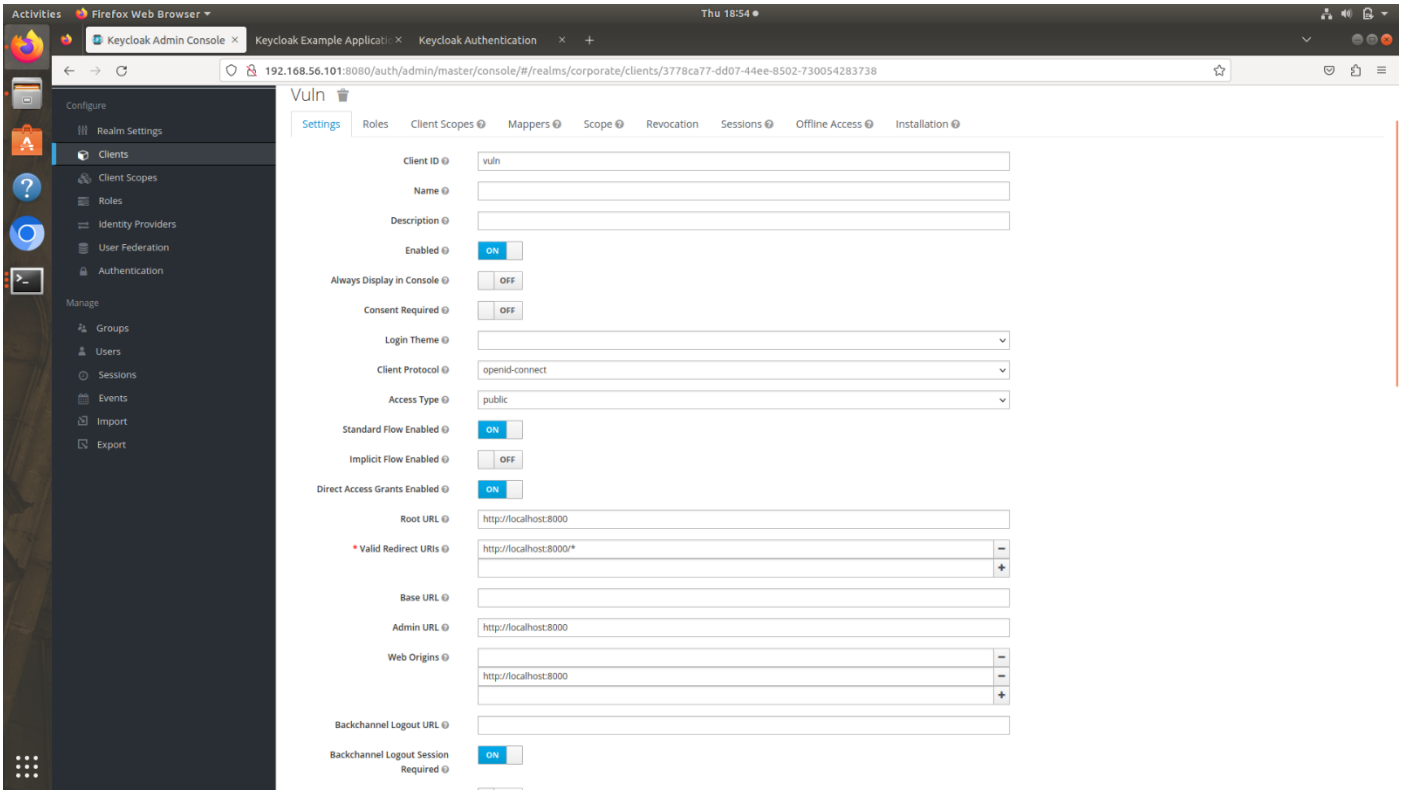

Figure V—34(Mitigation for Valid Redirect URI)

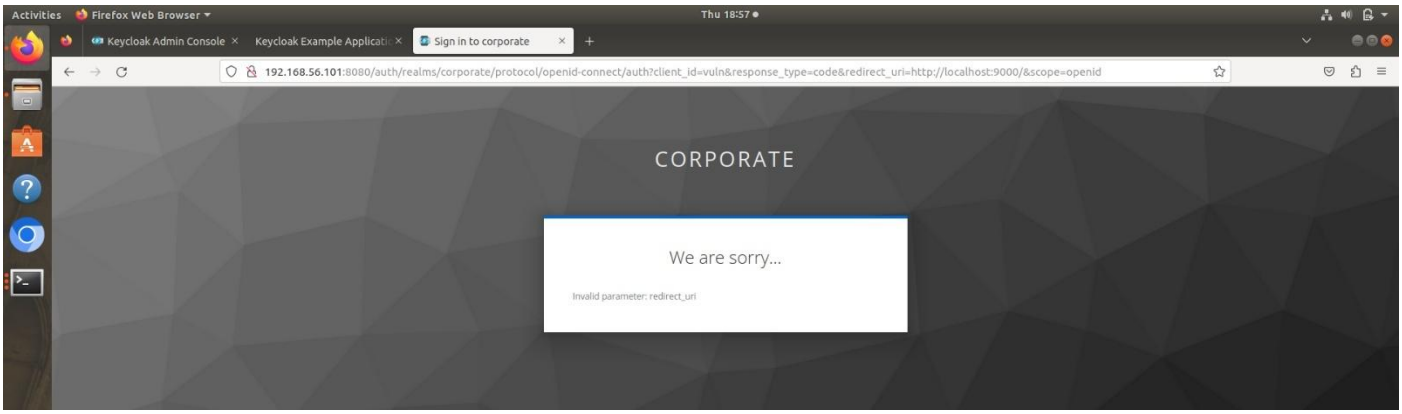

Figure V—35(Mitigation output for Valid Redirect URI)

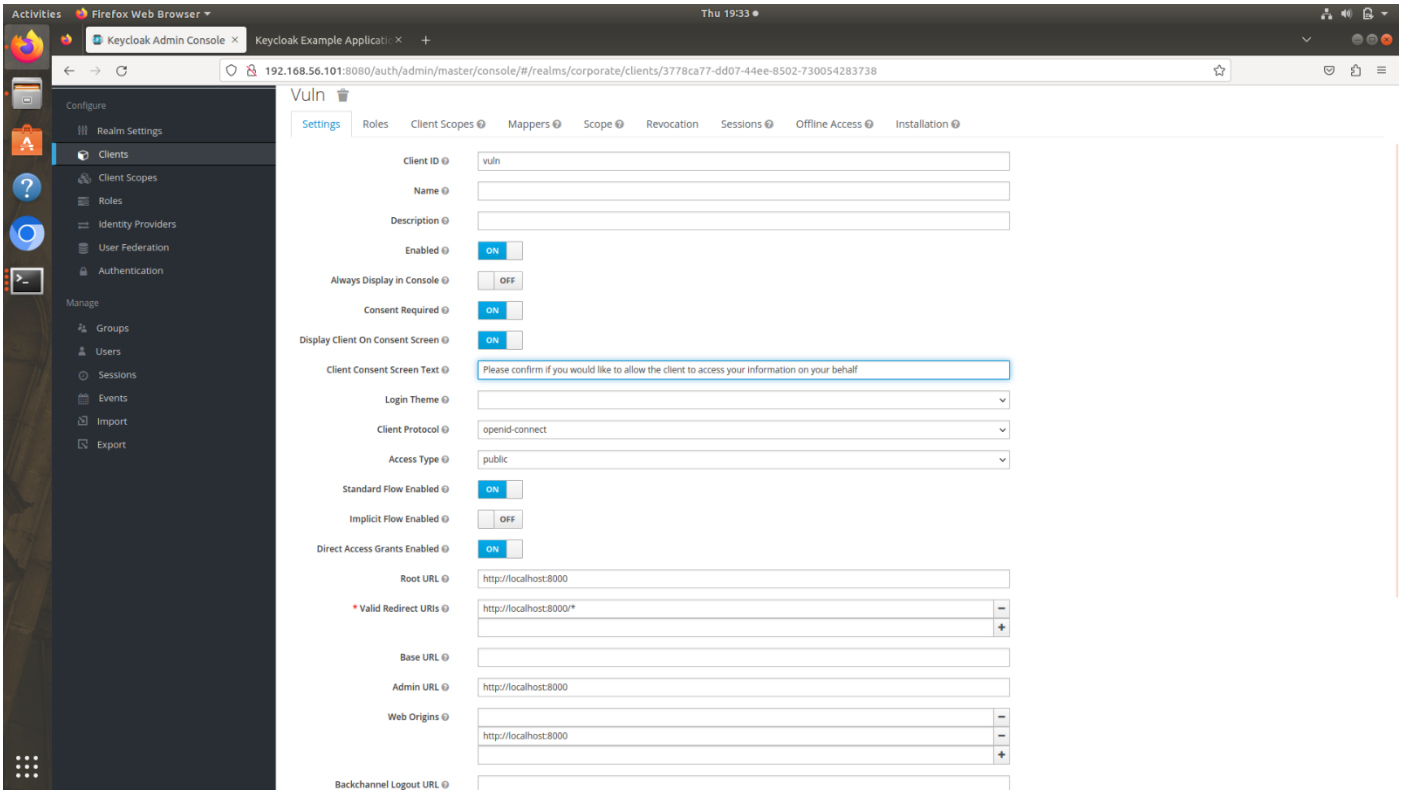

Figure V—36(Enabling consent required for the client)

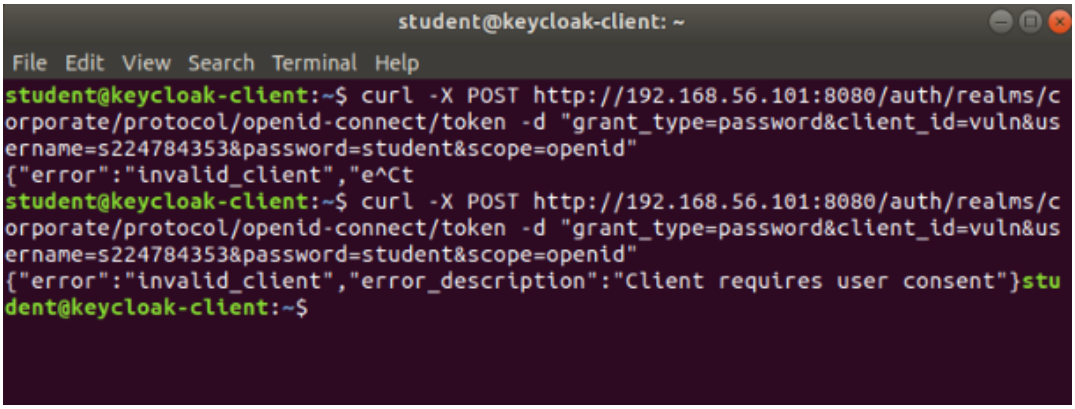

Figure V—37(Output for Curl Command after enabling consent required)

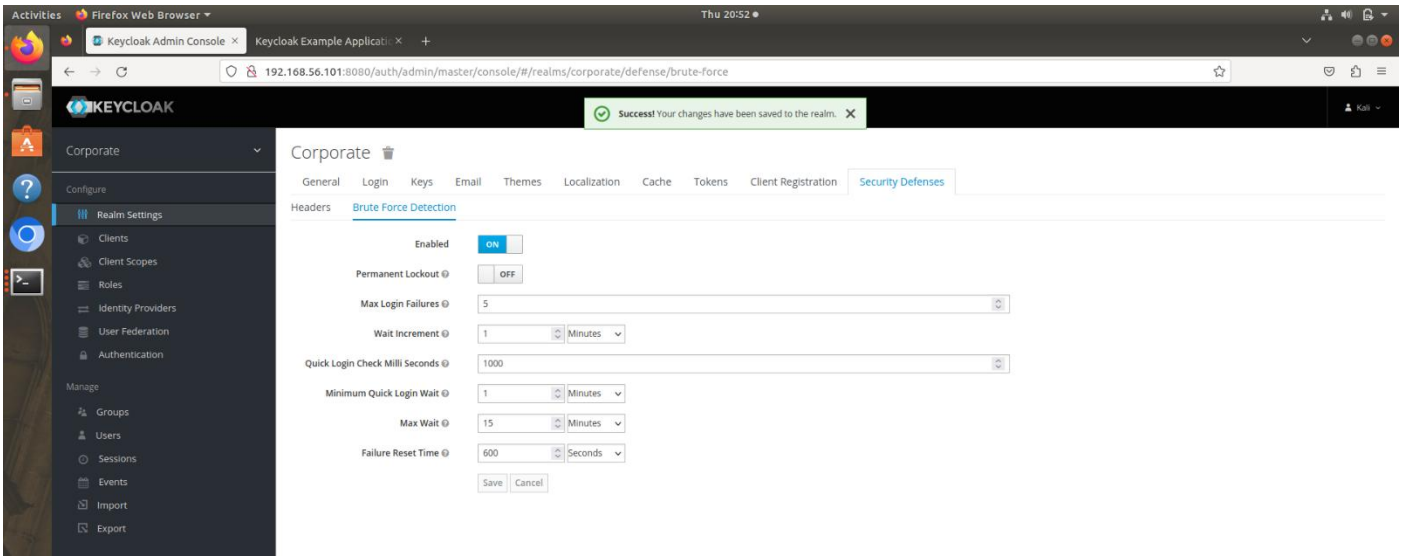

Figure V—38(Brute force defence configuration on Keycloak)

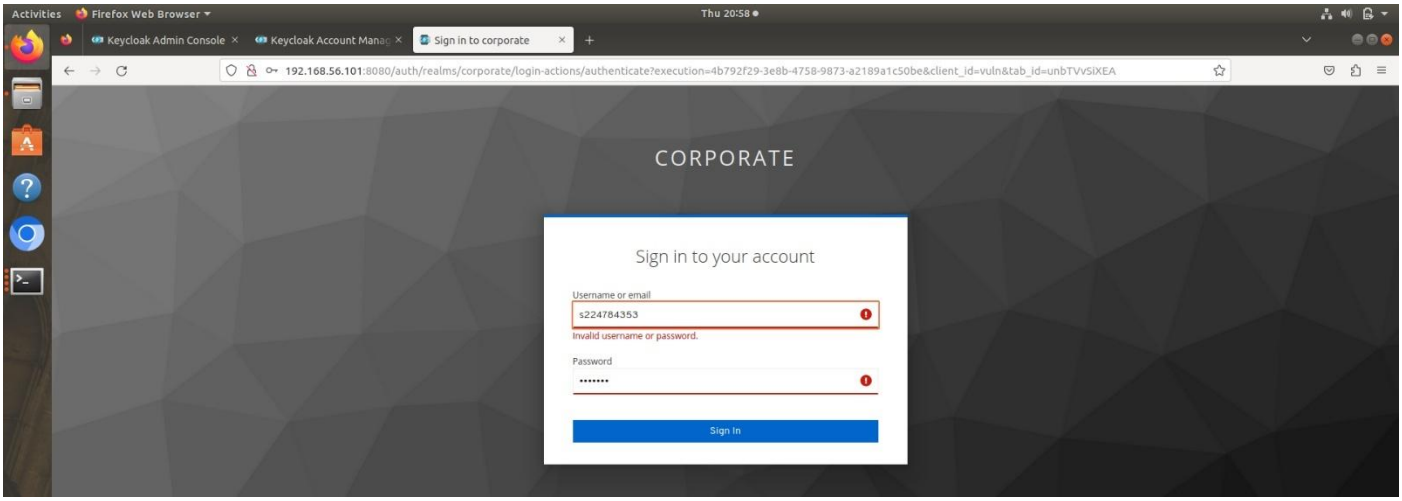

Figure V—39(Brute-forcing with wrong passwords)

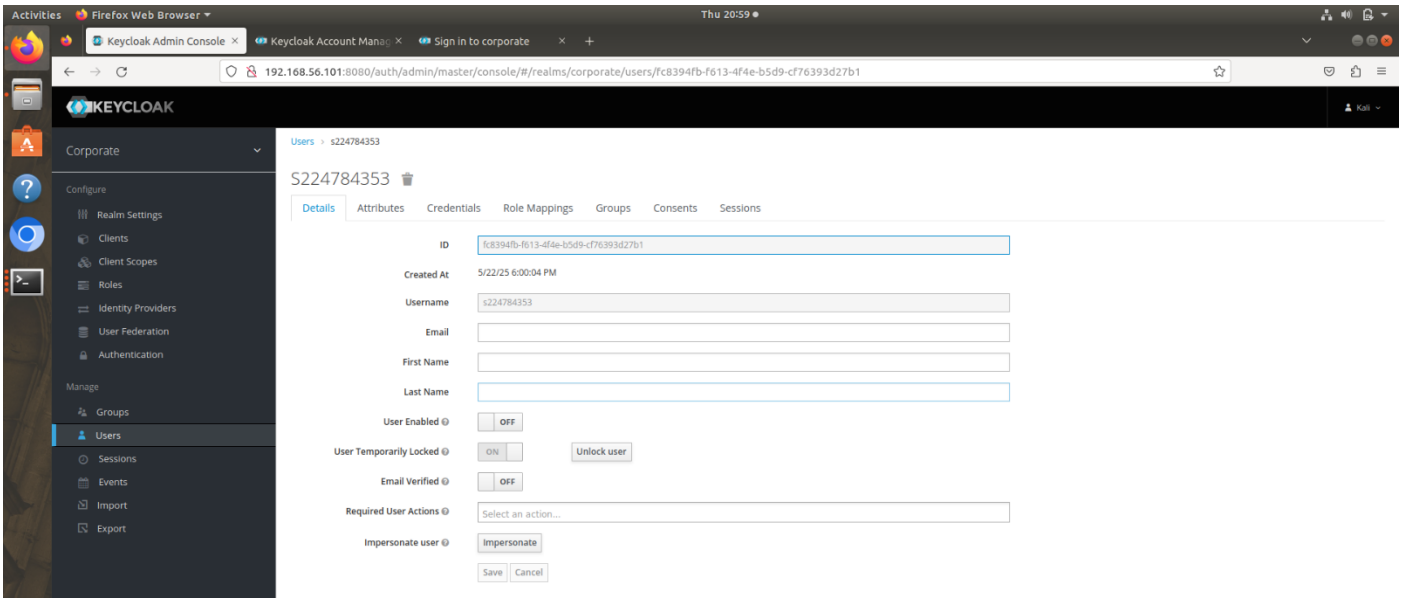

Figure V—40(Account Lockout after brute-forcing wrong passwords)

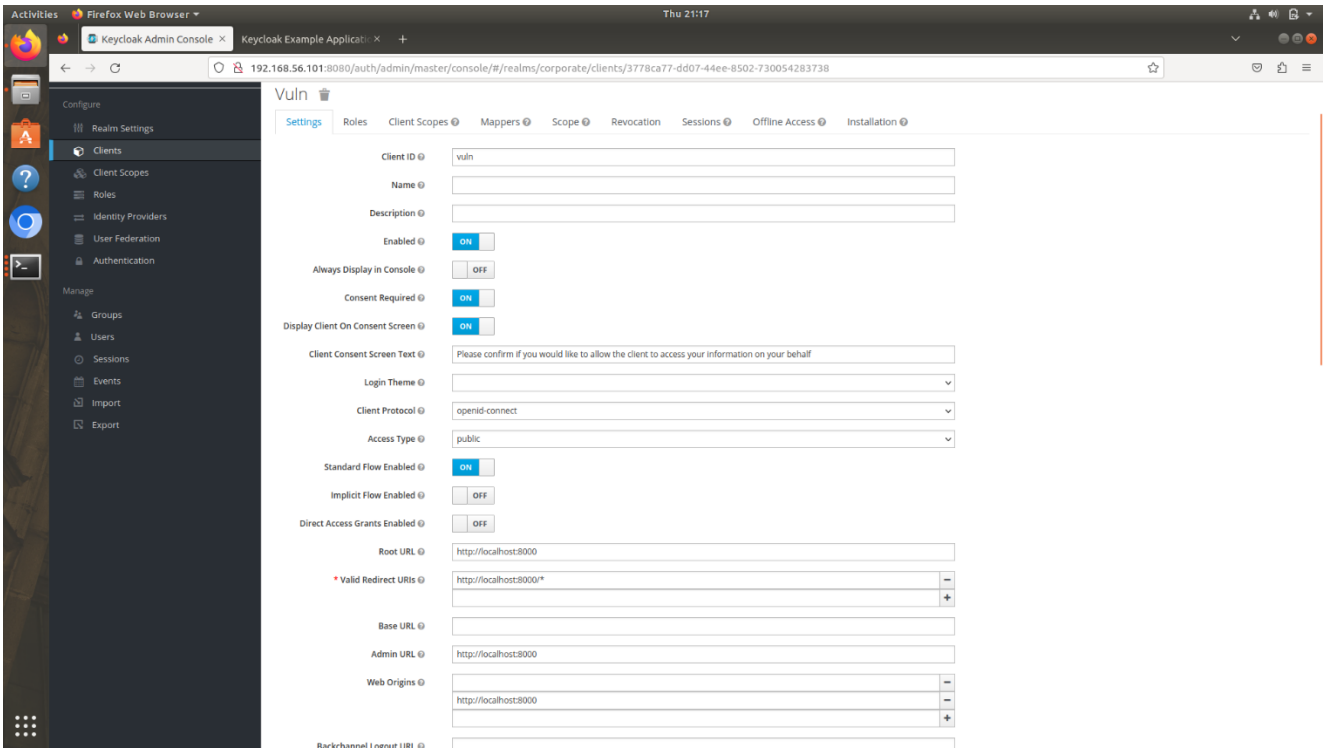

Figure V—41(Disabling Direct Access Grant to protect from ROPC)

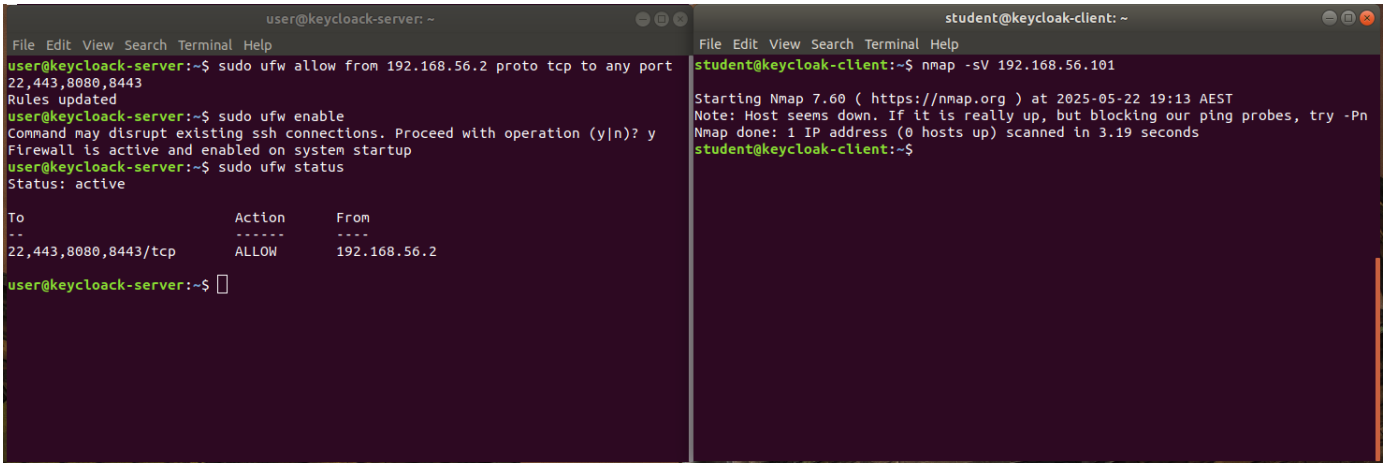

Figure V—42(Configuring with wrong IP to demonstrate UFW in work)

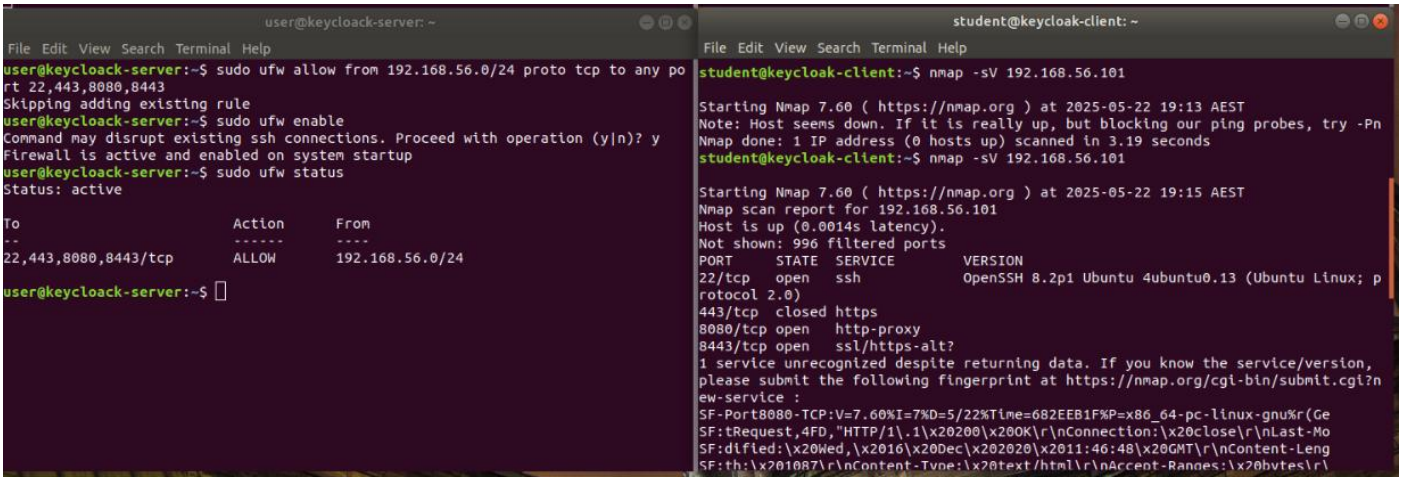

Figure V—43(Correct UFW configuration with demonstration for the open ports)

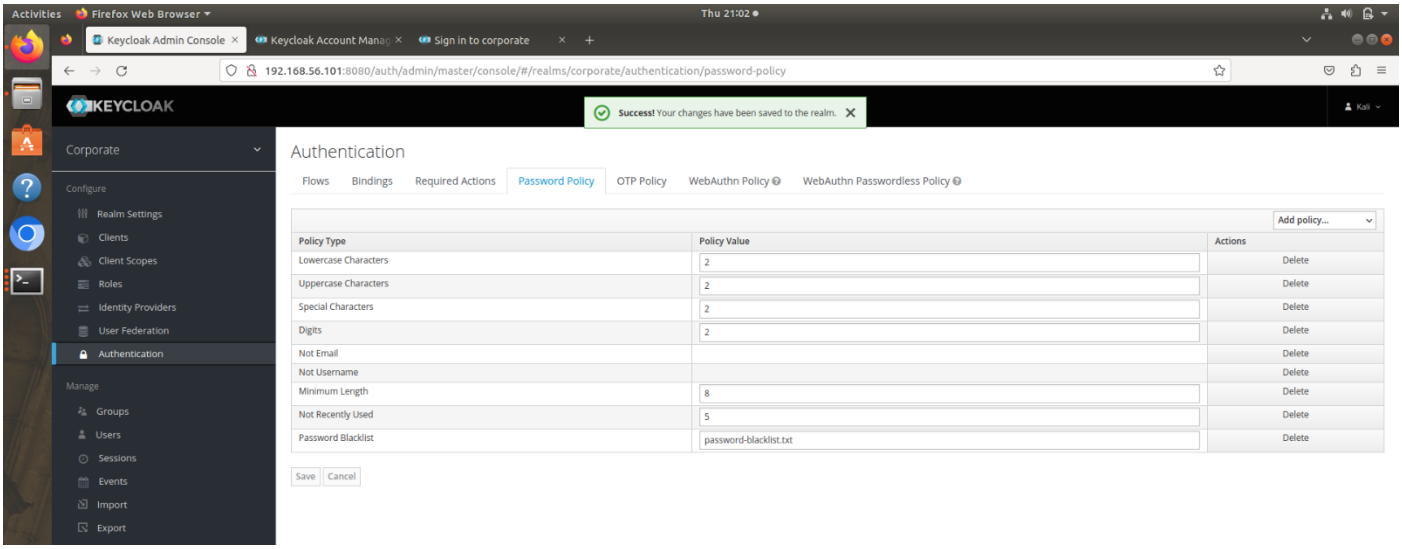

Figure V—44(Password policy for the realm)

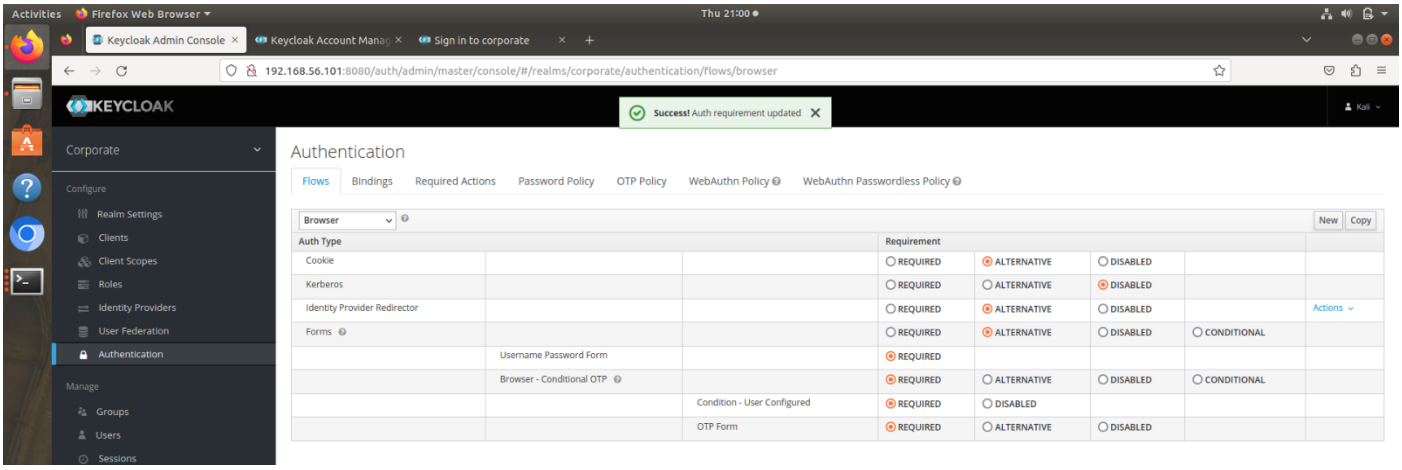

Figure V—45(Authentication flow updated to enable the 2FA)

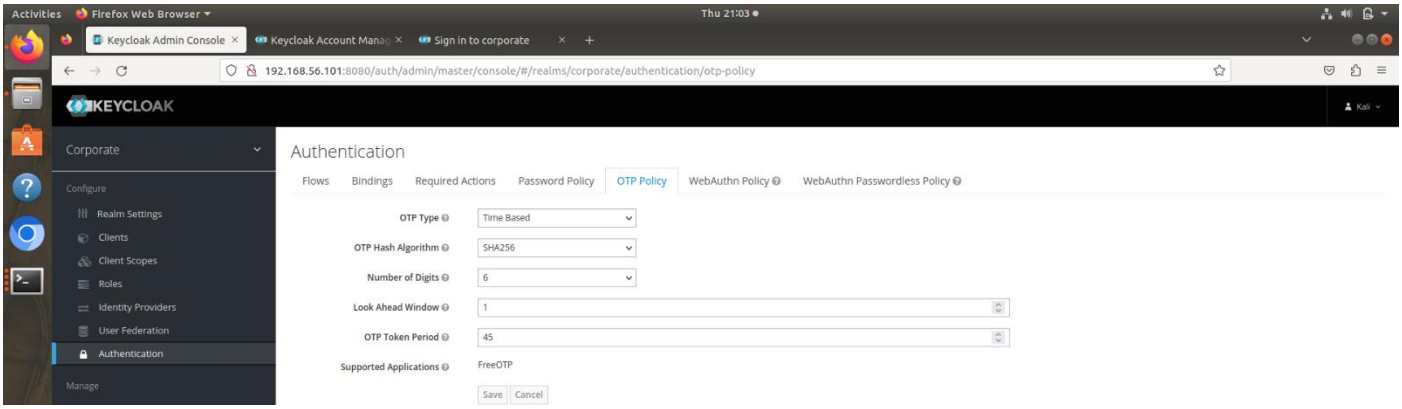

Figure V—46(OTP Policy for 2FA)

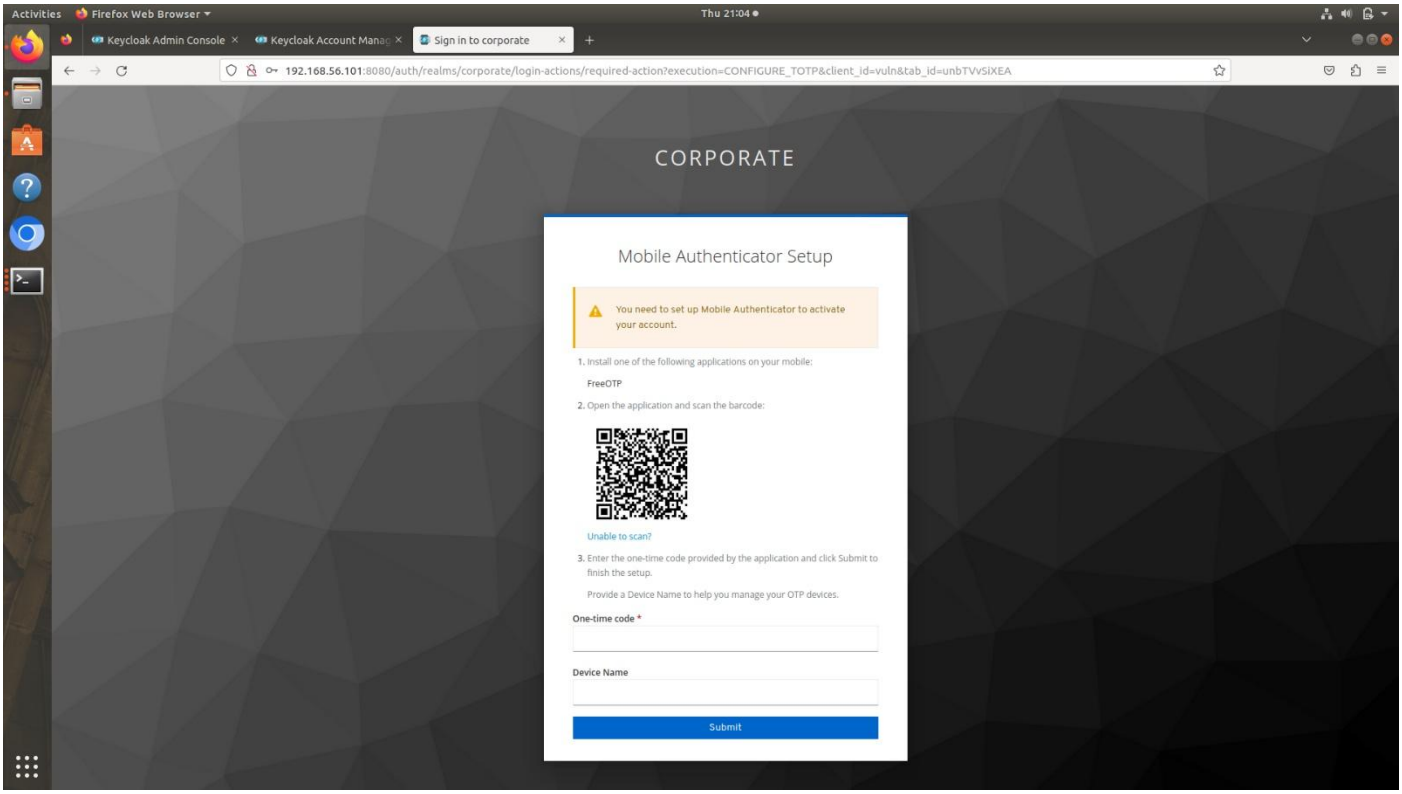

Figure V—47(Setting up 2FA for S224784353)

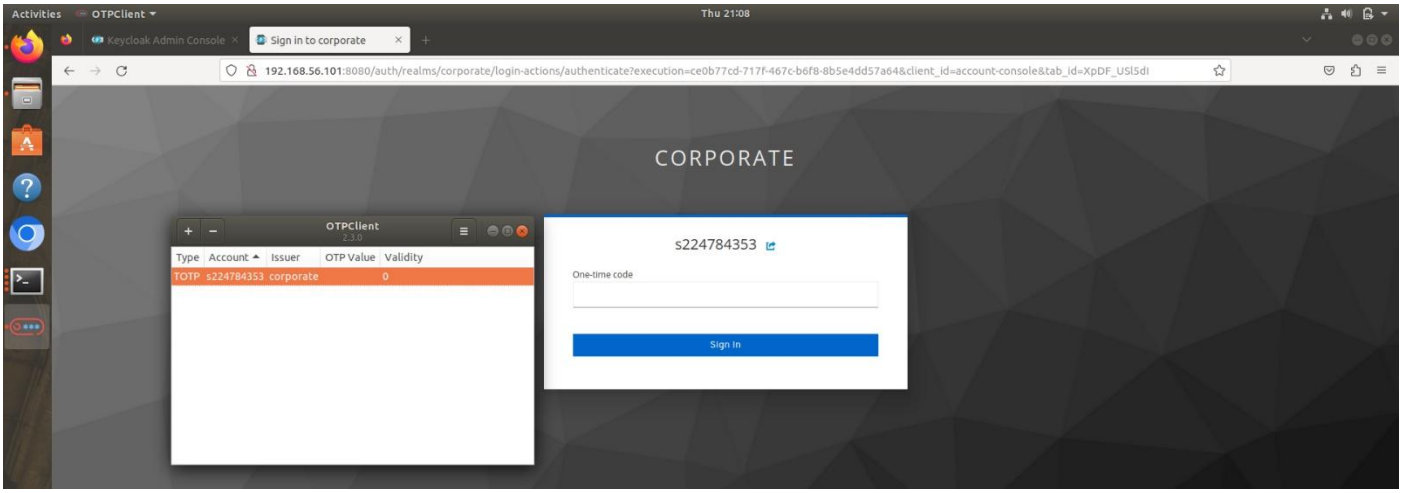

Figure V—48(Using 2FA to login on the S224784353 account)

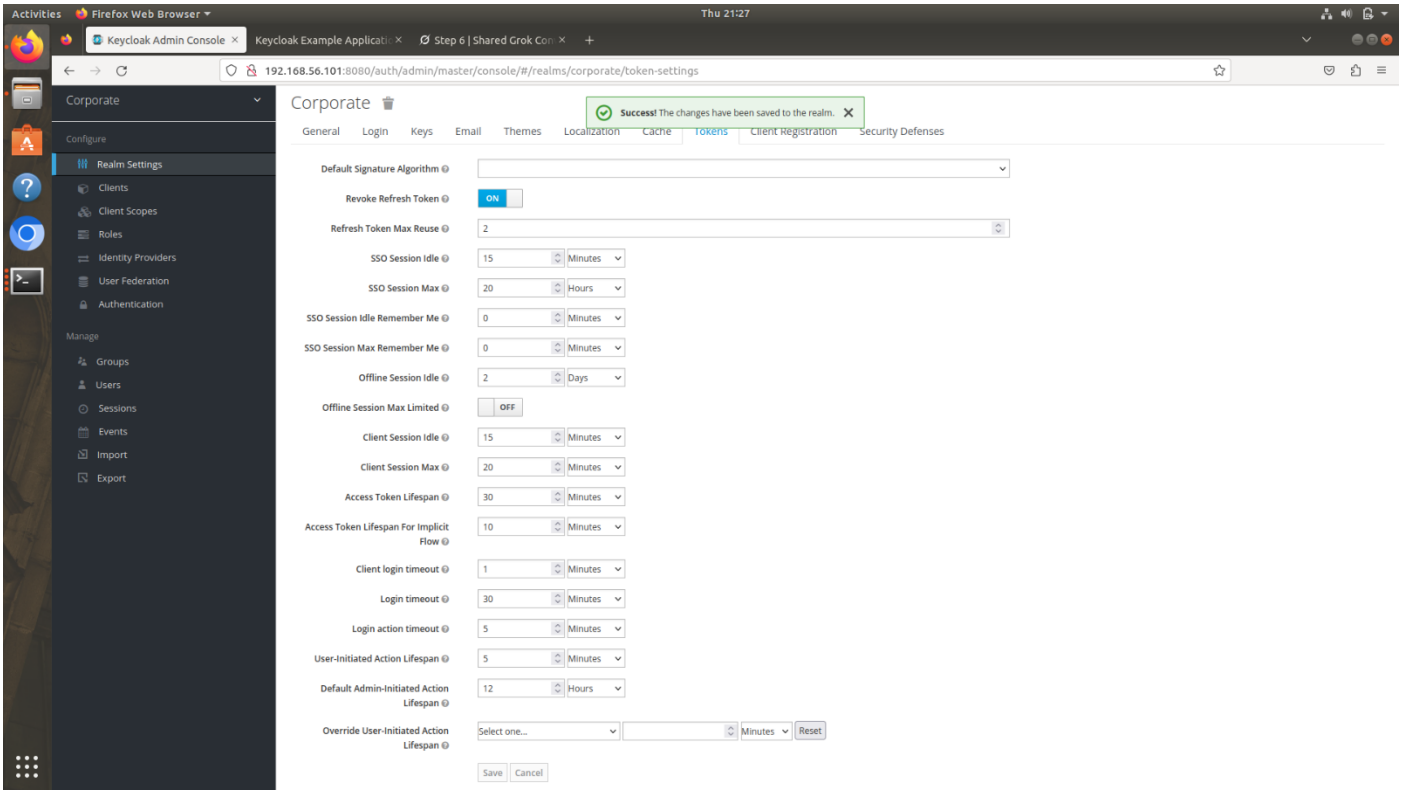

Figure V—49(Token Hardening Configurations)