VIT
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

Distributed Firewall Rule Generator

Submitted in partial fulfillment of the requirements for the course

PMCA602L - Python Programming

Winter 2024-25

by

24MCA0260 KARAN MAURYA

24MCA0121 RITAM HOTA

24MCA0242 AKASH KUMAR BANIK

24MCA0252 RAHUL MITTAL

24MCA0150 LOKESHKUMAR R

Course Instructor

Dr. Selva Rani B,

Associate Professor Senior

School of Computer Science Engineering and Information Systems

VIT, Vellore

April, 2025

DECLARATION

I/We hereby declare that the project entitled “Distributed Firewall Rule Generator” submitted by me/us, for the award of the course project PMCA602L - Python Programming to VIT is a record of bonafide work carried out by me/us under the supervision of Dr. Selva Rani B, Associate Professor Senior, SCORE.

I/We further declare that the work presented in this report has not been submitted and will not be submitted, either in part or in full, for the award of any other course in this institute or any other institute or university.

Place: Vellore

Date: 11-04-2025

Signature of the Candidate(s)

1. Karan Maurya
2. Ritam Hota
3. Akash Kumar Banik
4. Rahul Mittal
5. Lokeshkumar R

Signature of the Course Instructor

Project Summary

This project is a robust and modular application designed to centralize the management, generation, and enforcement of firewall rules across distributed systems. It leverages a server-client architecture to securely dispatch and execute firewall policies, ensuring consistent and efficient network security management. The system also includes a Policy Editor for administrators to easily manage and update firewall rules stored in a JSON-based configuration file.

This project automates the process of applying group-specific firewall rules using iptables, reducing manual errors and enhancing network security. It is particularly useful in environments where multiple systems require tailored security policies, such as enterprises, educational institutions, or organizations with diverse user groups.

The system is designed with scalability and extensibility in mind, allowing administrators to add new user groups, modify existing policies, and enforce security rules dynamically. By combining centralized control with distributed execution, the Firewall Management System simplifies firewall management, ensures consistent enforcement of security policies, and enhances overall network security.

The project is a comprehensive solution for centralized firewall rule management and enforcement. By automating the generation, distribution, and execution of firewall policies, it reduces manual effort, enhances security, and ensures consistency across distributed systems. Its modular design, secure communication, and user-friendly Policy Editor make it a valuable tool for organizations seeking to streamline their network security operations.

CONTENTS

Symbols and Notations

Symbol/Notation	Description
ACL	Access Control List
SSH	Secure Shell
IP	Internet Protocol
JSON	JavaScript Object Notation
YAML	Yet Another Markup Language
TCP	Transaction Control Protocol
DNS	Domain Name System
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
UDP	User Datagram Protocol
UFW	Uncomplicated FireWall
GUI	Graphical User Interface
Cisco ASA	Cisco Adaptive Security Appliance

CHAPTER 1

INTRODUCTION

1.1 Brief Overview

The Distributed Firewall Rule Generator is a application that uses a server-client architecture to manage firewall rules. The server dispatches predefined policies to clients based on their user group, and the clients execute these policies using iptables. The system ensures secure communication between the server and clients and provides centralized control over network policies.

The system consists of the following components:

- **Server:** The server acts as the central controller for managing and dispatching firewall policies to connected clients.
- **Client:** The client connects to the server, receives firewall policies, and executes them on the local machine.
- **Policy Dispatcher:** The policy dispatcher retrieves firewall rules for specific user groups from a predefined JSON file.
- **Policy Executor:** The policy executor ensures safe and controlled execution of iptables commands on the client machine.
- **Policy Editor:** The purpose of the Policy Editor is to provide a user-friendly interface for managing firewall policies stored in user_policies.json. It allows administrators to:
  - View Policies: Display all existing groups and their associated firewall rules.
  - Add Groups: Create new user groups for managing policies.
  - Add Policies: Add new iptables rules to specific groups.
  - Delete Policies: Remove specific rules from a group.
  - Remove Groups: Delete entire groups along with their policies.
  - Save Changes: Persist all modifications to the user_policies.json file.
- **Configuration:** The configuration files store essential settings for the server and policies for user groups.
- **Logging:** The logging system provides an audit trail of all dispatched policies for debugging and compliance purposes.

- **Communication Protocol:** The communication protocol defines how the server and client interact.
- **Firewall Rules:** The firewall rules define the network access policies for different user groups.

1.2 Objectives of the Project

- Centralize firewall rule management to reduce manual effort.
- Automate rule generation and distribution to minimize errors.
- Ensure secure and consistent rule enforcement across server and clients.
- Provide logging and monitoring for auditing purposes.

1.3 Scope and Significance

- **Scope**: The system can be deployed in organizations to manage network access for different departments or user groups.
- **Significance**:
  - Simplifies firewall management and reduces configuration errors by automating rule application.
  - Enhances security by ensuring consistent rule enforcement.
  - Provides a foundation for future extensions, such as GUI-based management or support for additional platforms.

CHAPTER 2

BACKGROUND SURVEY

2.1 Existing Solutions

- **Manual Configuration:** Administrators manually configure iptables rules on each machine, which is error-prone and time-consuming.
- **Commercial Tools:** Tools like Cisco ASA, pfSense, and UFW provide centralized firewall management but are often expensive and complex.

2.2 Relevant Theoretical Concepts

- **Network Security and Access Control**: Ensures only authorized traffic is allowed.
- **Remote Command Execution**: Uses SSH to securely execute commands on remote systems.
- **JSON for Configuration**: Provides a lightweight and human-readable format for storing rules and node details.
- **Secure Communication Protocols**: Ensures data integrity and confidentiality during rule distribution.
- **Firewall:** A network security system that monitors and controls incoming and outgoing traffic based on predefined rules.
- **Server-Client Architecture:** A model where a central server communicates with multiple clients to distribute tasks or data.

2.3 Research or Readings Referred

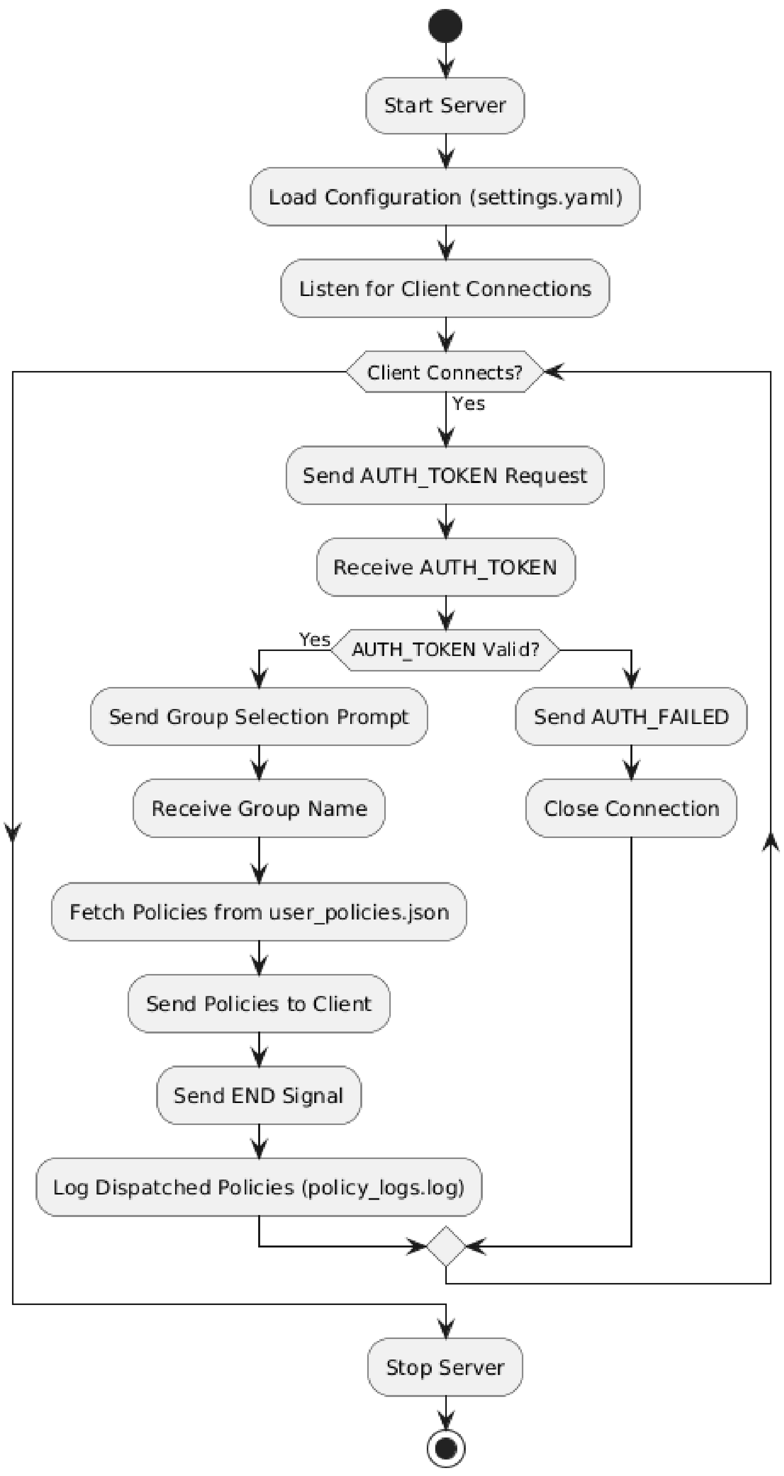
- Official documentation for python iptables, paramika and yaml.
- Python tutorials for JSON and SSH integration.
- Python socket programming tutorials.
- Python socket programming tutorials.

CHAPTER 3

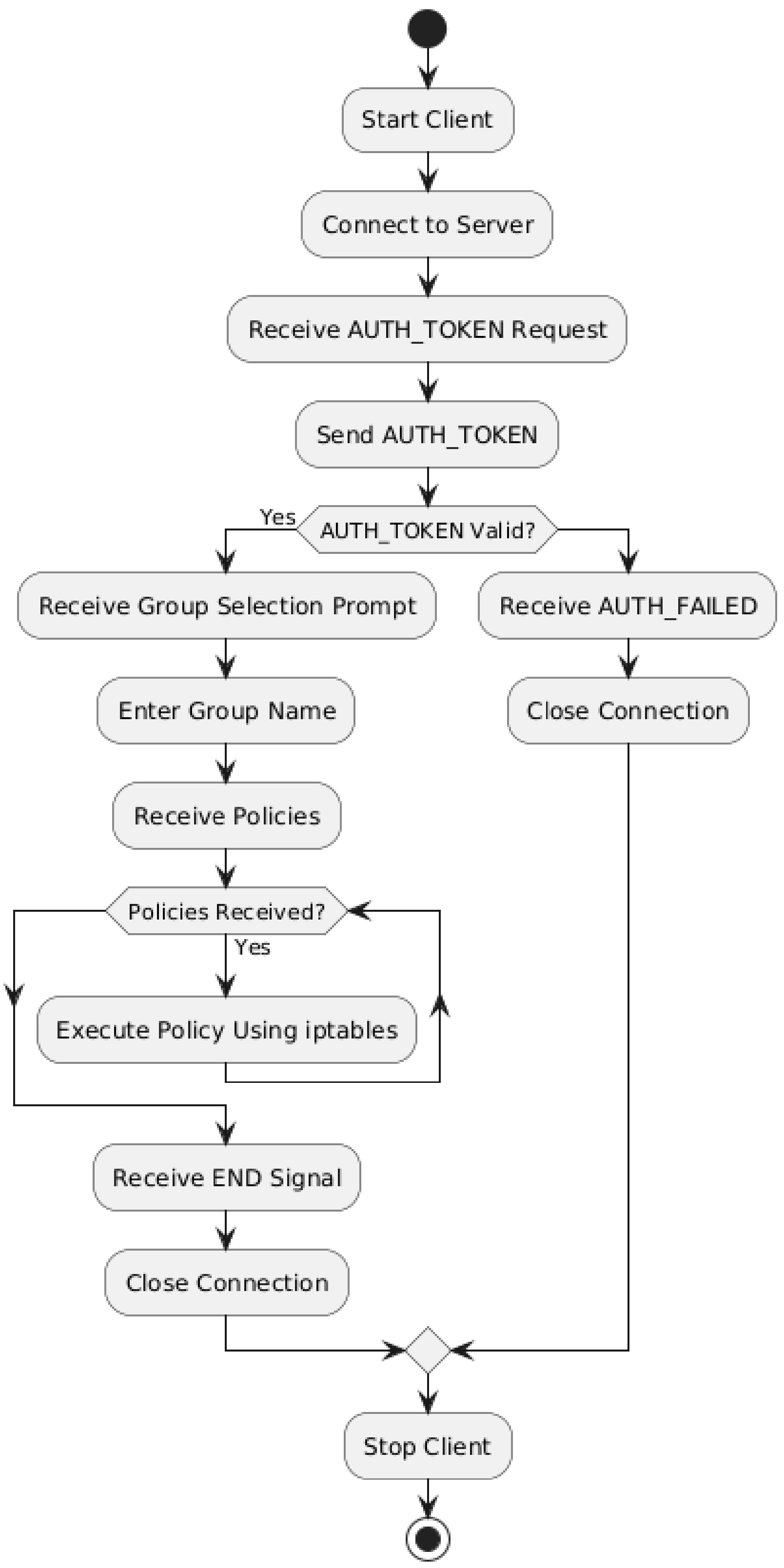
SYSTEM DESIGN / METHODOLOGY

3.1 Architecture or Workflow Diagram

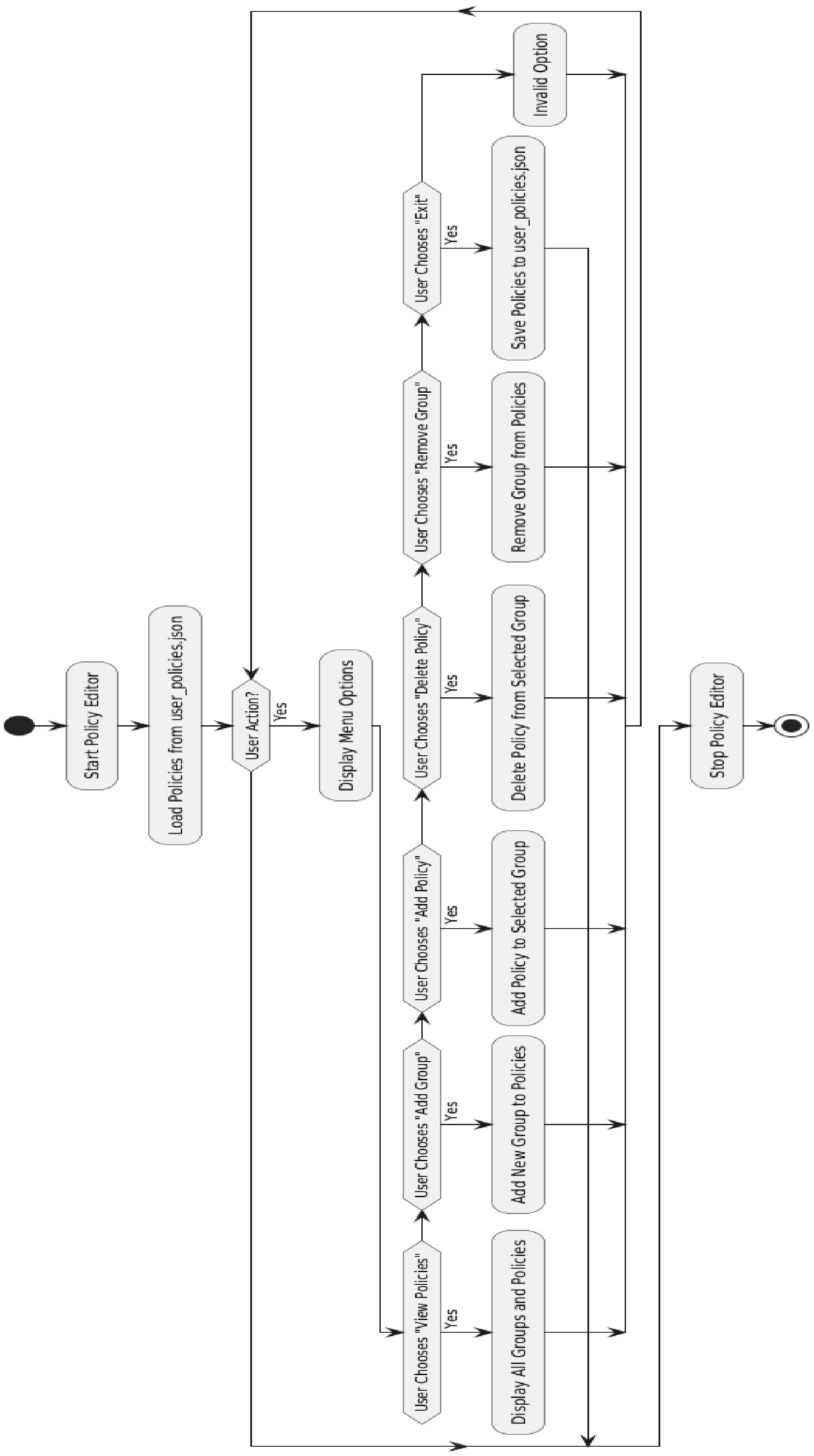
Server Workflow:



Client Workflow:

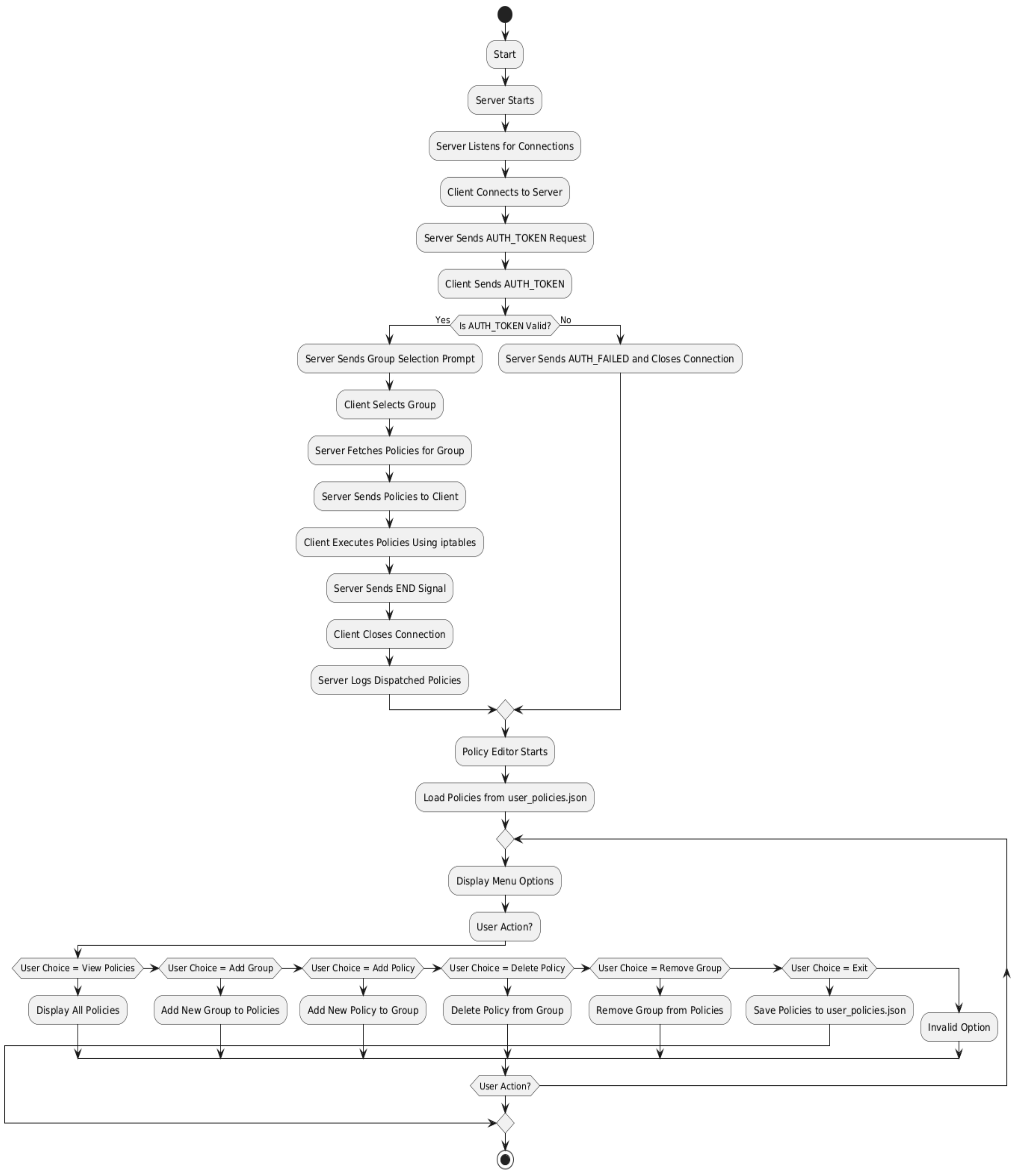


Policy_Editor Workflow:



Start Policy Editor
Load Policies from user_policies.json
User Action?
Yes
Display Menu Options
User Chooses "View Policies"
Yes
Display All Groups and Policies
User Chooses "Add Group"
Yes
Add New Group to Policies
User Chooses "Add Policy"
Yes
Add Policy to Selected Group
User Chooses "Delete Policy"
Yes
Delete Policy from Selected Group
User Chooses "Remove Group"
Yes
Remove Group from Policies
User Chooses "Exit"
Yes
Save Policies to user_policies.json
Invalid Option
Stop Policy Editor

Project Workflow:


Start
Server Starts
Server Listens for Connections
Client Connects to Server
Server Sends AUTH_TOKEN Request
Client Sends AUTH_TOKEN
Is AUTH_TOKEN Valid?
Yes
No
Server Sends Group Selection Prompt
Server Sends AUTH_FAILED and Closes Connection
Client Selects Group
Server Fetches Policies for Group
Server Sends Policies to Client
Client Executes Policies Using iptables
Server Sends END Signal
Client Closes Connection
Server Logs Dispatched Policies
Policy Editor Starts
Load Policies from user_policies.json
Display Menu Options
User Action?
User Choice = View Policies
User Choice = Add Group
User Choice = Add Policy
User Choice = Delete Policy
User Choice = Remove Group
User Choice = Exit
Display All Policies
Add New Group to Policies
Add New Policy to Group
Delete Policy from Group
Remove Group from Policies
Save Policies to user_policies.json
Invalid Option
User Action?

3.2 Tools, Libraries, or Frameworks Used

- **Python**: Core programming language.
- **Socket**: For server-client communication.
- **Paramiko**: For SSH-based communication with remote nodes.
- **YAML**: For configuration management.
- **iptables**: Linux firewall tool for rule enforcement.
- **JSON**: For storing ACL and node configurations.

3.3 Algorithm or Pseudocode

Server:

```
1. Load configuration from settings.yaml (host, port, auth_token).
2. Initialize the policy dispatcher with user_policies.json.
3. Start the server:
    a. Bind to the configured host and port.
    b. Listen for incoming client connections.
4. For each client connection:
    a. Send "AUTH_TOKEN" request to the client.
    b. Receive the client's token.
    c. If the token matches the configured auth_token:
        i)   Send a prompt to the client to select a user group.
        ii)  Receive the selected group name.
        iii) Fetch policies for the selected group from user_policies.json.
        iv)  Send each policy to the client.
        v)   Send an "END" signal to indicate the end of the policy list.
        vi)  Log the dispatched policies in policy_logs.log.
    d. If the token is invalid:
        i)   Send "AUTH_FAILED" to the client.
        ii)  Close the connection.
5. Repeat for all incoming connections.
6. Stop the server when terminated.
```

Client:

1. Load server configuration (host, port, auth_token).
2. Start the client:
   a. Connect to the server using the configured host and port.
   b. Receive "AUTH_TOKEN" request from the server.
   c. Send the auth_token to the server.
   d. If the server validates the token:
      i) Receive the group selection prompt.
      ii) Input the desired group name and send it to the server.
      iii) While receiving policies from the server:
         - Execute each policy using iptables.
      iv) Receive the "END" signal from the server.
      v) Close the connection.
   e. If the server sends "AUTH_FAILED":
      i) Print an error message.
      ii) Close the connection.
3. Stop the client after completing its tasks.

Policy Editor:

1. Load policies from user_policies.json:
   If the file does not exist, initialize an empty dictionary.
2. Display the Policy Editor menu:
   Options:View all policies, Add a new group, Add a policy to a group, Delete a policy from a group, Remove a group, Exit.
3. Based on the user's choice:
   a. If "View all policies":
      - Display all groups and their policies.
   b. If "Add new group":
      - Input the group name.
      - If the group does not exist, add it to the policies.
   c. If "Add policy to a group":
      - Input the group name.
      - If the group exists, input the iptables rule and add it to the group.

d. If "Delete policy from a group":
   - Input the group name.
   - If the group exists, display its policies.
   - Input the policy number to delete and remove it from the group.

e. If "Remove group":
   - Input the group name.
   - If the group exists, remove it from the policies.

f. If "Exit":
   - Save the policies to user_policies.json.
   - Exit the editor.

g. If invalid option:
   - Print an error message.

4. Save the policies to user_policies.json after each operation.
5. Repeat until the user selects "Exit."

Policy Dispatcher:

1. Load user_policies.json.
2. For a given group name:
   - If the group exists, return its list of policies.
   - If the group does not exist, return an empty list or error message.

Policy Executor:

1. For each received command:
   a. If the command is valid (starts with "iptables"):
      - Execute the command using subprocess.
      - Log any errors if the command fails.

   b. If the command is "END":
      - Stop processing further commands.

   c. If the command is invalid:
      - Print an error message and skip the command.
2. Before executing any commands:
   a. Flush existing iptables rules to avoid conflicts.
   b. Add default rules to allow essential traffic (e.g., SSH, DNS, HTTP, HTTPS).

CHAPTER 4

IMPLEMENTATION

4.1 Description of Major Modules

1. controller/agent.py: Client-side handler

The agent.py script serves as the client-side agent component in the User Domain Firewall Policy Controller system. Its primary role is to establish a secure and persistent communication channel with a centralized server, authenticate itself, and receive firewall or policy commands for execution.

Functional Overview

Upon execution, the agent performs the following sequence of operations:

1. **Connection Establishment**: The script initiates a TCP connection to a predefined server IP address (127.0.0.1) and port (9090), functioning as the command-and-control hub for distributed policy enforcement.
2. **Authentication**:
   After establishing a connection, the agent waits for an "AUTH_TOKEN" prompt from the server. Once received, it replies with a hardcoded authentication token ("securetoken123") to validate its identity.
3. **Group Identification**:
   Once authenticated, the agent receives a prompt to specify a group name. This input categorizes the client into a specific logical group, allowing the server to target and manage multiple clients in organized segments.
4. **Command Reception and Execution**:
   The agent enters a loop where it continuously listens for incoming commands. Received command strings are split by newline characters and processed individually. Each command is passed to the safe_execute function—imported from the client.policy_executor module—to ensure safe and restricted execution within the client environment.
5. **Termination Handling**:
   The special signal "END" is used to indicate the termination of the session or the end of a batch of commands. Upon receiving this signal, the agent exits the execution loop and

terminates the connection gracefully.

6. **Error Handling:** The agent includes robust exception handling mechanisms to catch and report both connection-related and general execution errors. Regardless of the outcome, the socket connection is always properly closed to prevent resource leakage.
7. **Security Considerations:** While the script uses a hardcoded token for authentication and a safe execution wrapper, it is crucial to ensure that the safe_execute function is rigorously sandboxed and validated to prevent command injection or unauthorized access.
8. **Purpose and Deployment:** This agent is designed to be deployed across client machines within a network, enabling the central controller to remotely enforce firewall rules or execute system-level policy changes in a secure and organized manner.

2. controller/command_logger.py: Loads the policies command from the JSON file

The command_logger.py module is responsible for logging executed policy or command actions within the User Domain Firewall Policy Controller system. This component ensures that all operations performed by the agent or controller are persistently recorded in a structured and manageable log format, supporting both traceability and audit compliance.

Functional Overview

The module defines a single function, setup_logger(), which initializes and returns a configured logger object named "PolicyLogger". This logger can be imported and used by other components (e.g., the agent or the controller) to log command execution events or security policy changes.

Logging Configuration

The logger is configured with the following features:

1. **Log Level:** The log level is set to INFO, capturing informational messages about routine operations. This level provides visibility into all command executions without overwhelming the log with lower-level debug messages.
2. **Rotating File Handler:** A RotatingFileHandler is used to prevent unbounded log file growth. The logger writes to a file named policy_logs.log with a maximum file size of

1,000,000 bytes (approximately 1 MB). Once the file size limit is reached, the handler rotates the log file, maintaining up to 5 backup copies.

3. **Log Format:** Log messages are formatted to include a timestamp followed by the log message itself. This ensures that all log entries are time-stamped for easy tracking and chronological sorting:

3. controller/policy_dispatcher.py: Sends the policies loaded to the client execution based on user group

The policy_dispatcher.py module is responsible for managing and dispatching security or firewall policies based on group identifiers. It serves as a lightweight policy retrieval component within the User Domain Firewall Policy Controller system, allowing the controller to efficiently retrieve and distribute group-specific rules or command sets.

Functional Overview

The core functionality of this module is encapsulated in the PolicyDispatcher class, which loads policy definitions from a JSON file and provides methods to retrieve policies associated with specific groups.

The dispatcher is primarily used by the central controller to:

- Dynamically fetch and dispatch the appropriate policy set to agents based on their group membership.
- Maintain separation between policy definitions and execution logic, enabling better modularity and maintainability.
- Simplify updates to group-based security rules via centralized JSON files.

4. client/policy_executor.py: Runs the policy commands on the client system

The policy_executor.py module is a security-critical component of the User Domain Firewall Policy Controller system. It provides a controlled and safe mechanism for executing system-level commands on the client machine, particularly focusing on firewall configuration using iptables.

Functional Overview

This module defines a function safe_execute(command) designed to process and execute only a predefined set of allowed system commands. It includes built-in checks and safeguards to prevent unauthorized or unsafe command execution, with a focus on minimal, auditable changes to the system firewall.

Key Functional Components

a. **Allowed Commands Filter:**
   The module restricts execution to a whitelist of approved commands, defined in the ALLOWED_COMMANDS list. Currently, only commands starting with iptables are allowed. This serves as a first line of defense against injection or misuse.
b. **Function:** safe_execute(command)
   This function performs the following steps:
   - Validation: Skips execution if the command is empty or not permitted by the whitelist.
   - Initial Rule Flush: If the incoming command is the first valid iptables command to be executed during the session, the module flushes existing firewall rules using iptables -F. This operation is tracked using a dynamic flag (safe_execute.FLUSHED) to avoid redundant flushes.
   - Command Execution: The command is parsed safely using shlex.split() to prevent shell injection, then executed using subprocess.run() with error handling enabled.
   - Error Handling: If execution fails or a disallowed command is detected, an appropriate message is printed and the command is not executed.

5. controller/server.py: Server-side handler

The server.py module serves as the core of the User Domain Firewall Policy Controller system. It acts as the central authority responsible for accepting incoming client connections, authenticating agents, and dispatching group-based security policies securely and concurrently.

Functional Overview

This server operates over TCP using Python’s socket and threading modules. It integrates with the PolicyDispatcher for retrieving group-specific policies and the command_logger to maintain logs of policy transmission events.

Configuration Loading

At startup, the server reads its configuration parameters from a YAML file located at config/settings.yaml. This file provides three essential settings:

- **host**: IP address or hostname to bind the server socket.
- **port**: TCP port number to listen for client connections.
- **auth_token**: A shared secret token used for authenticating clients.

This approach ensures centralized configuration management and improves the system's adaptability to different environments.

Key Functional Components

a. **Policy Dispatcher Integration:**
   The server instantiates a PolicyDispatcher to load and access group-specific firewall policies defined in a JSON file (policies/user_policies.json).
b. **Command Logger Integration:**
   A logger instance is set up via setup_logger() to record all dispatched commands along with client address information.
c. **Function:** handle_client(client_socket, addr)
   This function is executed in a separate thread for each client connection. It follows this process:
   - Prompts the client for authentication using the shared token.
   - If authenticated, prompts the client to select a group (e.g., corporate, vpn_users, admins, etc.).
   - Retrieves and sends all associated policy commands to the client.
   - Logs each dispatched command for auditing purposes.
   - Sends an "END" signal to indicate the end of communication, then closes the connection.
d. **Function:** main()
   Sets up the TCP server, binds it to the specified host and port, and listens for incoming connections. Each accepted client is handled in a separate thread to support concurrent client sessions.

6. config/settings.yaml: Server-side configuration data

The settings.yaml file provides essential configuration parameters for initializing and operating the central server component of the User Domain Firewall Policy Controller system. Using YAML for configuration allows for human-readable, structured settings that can be easily modified without altering the main application code.

The file contains a top-level server section with the following keys:

a. **server:**
   host: 0.0.0.0
   port: 9090
   auth_token: "securetoken123"
b. **host:** Specifies the IP address on which the server should listen.
   Value: 0.0.0.0
   Purpose: Allows the server to accept incoming connections on all available network interfaces.
c. **port:** Specifies the port number for incoming TCP connections.
   Value: 9090
   Purpose: Used by client agents to connect to the server. This must match the port defined in the client configuration.
d. **auth_token:** A shared secret token used to authenticate clients.
   Value: "securetoken123"
   Purpose: Ensures only trusted clients are allowed to receive and execute policy instructions.

7. ssh/ssh_policy_push.py: Remote client access interface

The ssh_policy_push.py module is responsible for remotely applying firewall or security policies to target machines over SSH. It provides an interface to push commands securely to remote systems, making it a crucial utility for distributed enforcement of security policies in multi-host environments.

Functional Overview

This module uses the paramiko library, a widely used SSH protocol implementation for Python. It defines a single function, apply_policy_via_ssh, which connects to a remote host using SSH credentials and executes a list of provided shell commands.

Function: apply_policy_via_ssh(hostname, username, password, commands)

Parameters:

- **hostname**: IP address or domain name of the remote machine.
- **username**: Username for SSH authentication.
- **password**: Password corresponding to the SSH user.
- **commands**: A list of shell commands (typically firewall rules) to execute on the remote host.

8. policies/user_policies.json: Stores all the policies based on the user groups

The user_policies.json file serves as the central repository for group-based firewall rules used by the Policy Controller system. It defines predefined sets of outbound and inbound iptables rules for different user groups, allowing for dynamic and role-based network access control.

File Location: The file is typically stored in: policies/user_policies.json

Structure Overview: This JSON file contains key-value pairs where:

- **Key**: The name of the user group (e.g., corporate, developers, admins).
- **Value**: A list of shell commands (specifically iptables commands) that define the access policies for that group.
- Sample Commands stored in json file:

```
{
  "corporate": [
    "iptables -A OUTPUT -p tcp --dport 443 -j ACCEPT",
    "iptables -A OUTPUT -p tcp --dport 80 -j ACCEPT",
    "iptables -A OUTPUT -p tcp --dport 6881:6889 -j DROP",
    "iptables -A OUTPUT -p tcp --dport 22 -j DROP"
  ] }
```

4.2 Code Snippets

agent.py

```
import socket

from client.policy_executor import safe_execute

SERVER = '127.0.0.1'

PORT = 9090

AUTH_TOKEN = "securetoken123"

def main():

    try:

        # Establish connection to the server

        client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

        client.connect((SERVER, PORT))



        # Authenticate with the server

        if client.recv(1024).decode() == "AUTH_TOKEN":

            client.send(AUTH_TOKEN.encode())

        else:

            print("[-] Authentication failed.")

            client.close()

            return



        # Receive and display the group prompt

        prompt = client.recv(1024).decode()
```

```
print(prompt)
group = input("Group name: ").strip()
client.send(group.encode())

# Process commands from the server
while True:
    data = client.recv(1024).decode()
    if not data:
        print("[-] No data received. Closing connection.")
        break

    # Handle the END signal explicitly
    if data.strip() == "END":
        print("[+] Received END signal. Stopping execution.")
        break

    # Split commands by newline and execute each individually
    commands = data.strip().split("\n")
    for command in commands:
        if command.strip() == "END":
            print("[+] Skipping END signal.")
            continue
        if command.strip():  # Skip empty lines
```

```
                print(f"[+] Executing: {command.strip()}")

                safe_execute(command.strip())


    except ConnectionError as e:

        print(f"[-] Connection error: {e}")

    except Exception as e:

        print(f"[-] Unexpected error: {e}")

    finally:

        client.close()

        print("[+] Connection closed.")


if __name__ == '__main__':

    main()
```

command_logger.py

```
import logging

from logging.handlers import RotatingFileHandler

def setup_logger():

    logger = logging.getLogger("PolicyLogger")

    logger.setLevel(logging.INFO)

    handler = RotatingFileHandler("policy_logs.log", maxBytes=1000000, backupCount=5)

    formatter = logging.Formatter('%(asctime)s - %(message)s')

    handler.setFormatter(formatter)
    logger.addHandler(handler)

    return logger
```

policy_dispatcher.py

```
import json

class PolicyDispatcher:

    def __init__(self, policy_file):

        with open(policy_file, 'r') as file:

            self.policies = json.load(file)

    def get_policies_for_group(self, group):

        return self.policies.get(group, [])
```

policy_executor.py

```
import subprocess

import shlex

ALLOWED_COMMANDS = ["iptables"]


def safe_execute(command):

    if not command.strip():

        print("[-] Received an empty command. Skipping.")

        return


    # Flush iptables rules if the command is the first one

    if command.strip().startswith("iptables") and "FLUSHED" not in safe_execute.__dict__:

        try:
            print("[+] Flushing existing iptables rules.")
            subprocess.run(shlex.split("iptables -F"), check=True)
            safe_execute.FLUSHED = True  # Mark as flushed
        except subprocess.CalledProcessError as e:
            print(f"[-] Failed to flush iptables rules: {e}")
            return

    if any(command.startswith(allowed) for allowed in ALLOWED_COMMANDS):
        try:
            subprocess.run(shlex.split(command), check=True)
        except subprocess.CalledProcessError as e:
            print(f"[-] Command failed: {e}")
    else:
        print(f"[-] Blocked unsafe command: {command}")
```

server.py

```
import socket
import threading
import yaml
from controller.policy_dispatcher import PolicyDispatcher
from controller.command_logger import setup_logger

with open("config/settings.yaml", 'r') as f:
    config = yaml.safe_load(f)
```

```
HOST = config['server']['host']
PORT = config['server']['port']
AUTH_TOKEN = config['server']['auth_token']

dispatcher = PolicyDispatcher("policies/user_policies.json")
logger = setup_logger()

def handle_client(client_socket, addr):
    print(f"[+] Connection from {addr}")
    client_socket.send(b"AUTH_TOKEN")
    token = client_socket.recv(1024).decode().strip()
    if token != AUTH_TOKEN:
        print("[-] Unauthorized client")
        client_socket.close()
        return

    client_socket.send(b"Enter policy group (corporate, vpn_users, developers, guests, admins, 
blocked_users): ")
    group = client_socket.recv(1024).decode().strip()
    policies = dispatcher.get_policies_for_group(group)

    for policy in policies:
        client_socket.send(policy.encode() + b"\n")
        logger.info(f"Dispatched to {addr}: {policy}")
    client_socket.send(b"END")
    client_socket.close()

def main():
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.bind((HOST, PORT))
    server.listen(5)
    print(f"[*] Server listening on {HOST}:{PORT}")

    while True:
        client_sock, addr = server.accept()
        thread = threading.Thread(target=handle_client, args=(client_sock, addr))
        thread.start()

if __name__ == '__main__':
    main()
```

ssh_policy_push.py

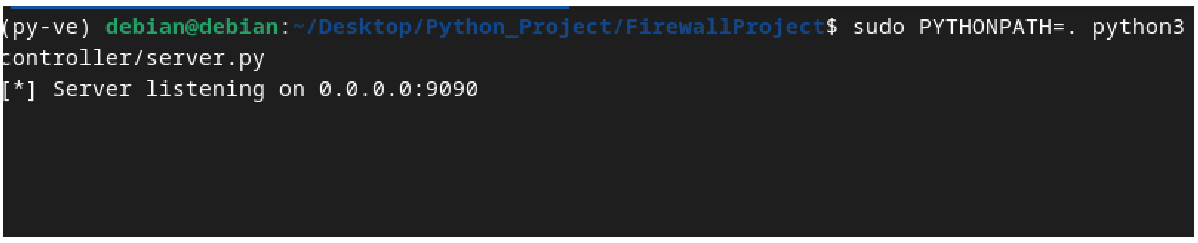
```
import paramiko

def apply_policy_via_ssh(hostname, username, password, commands):
    ssh = paramiko.SSHClient()
    ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())

    try:
        ssh.connect(hostname, username=username, password=password)
        for cmd in commands:
            stdin, stdout, stderr = ssh.exec_command(cmd)
            print(stdout.read().decode(), stderr.read().decode())
    finally:
        ssh.close()
```

policy_editor.py

```
import json
import os
POLICY_FILE = "policies/user_policies.json"

def load_policies():
    if not os.path.exists(POLICY_FILE):
        return {}
    with open(POLICY_FILE, 'r') as f:
        return json.load(f)

def save_policies(policies):
    with open(POLICY_FILE, 'w') as f:
        json.dump(policies, f, indent=2)


def view_policies(policies):
    if not policies:
        print("No policies found.")
        return
    for group, rules in policies.items():
        print(f"\nGroup: {group}")
        for idx, rule in enumerate(rules, start=1):
            print(f"  [{idx}] {rule}")
    print()

def add_group(policies):
    group = input("Enter new group name: ").strip()
    if group in policies:
        print("Group already exists.")
        return
    policies[group] = []
    print(f"Group '{group}' added.")

def add_policy(policies):
    group = input("Enter group name to add policy: ").strip()
    if group not in policies:
        print("Group not found.")
        return
    rule = input("Enter iptables rule: ").strip()
    policies[group].append(rule)
    print(f"Policy added to '{group}'.")

def delete_policy(policies):
    group = input("Enter group name to delete policy from: ").strip()
    if group not in policies:
        print("Group not found.")
        return
    view_policies({group: policies[group]})
    idx = int(input("Enter policy number to delete: ")) - 1
    if 0 <= idx < len(policies[group]):
        removed = policies[group].pop(idx)
        print(f"Removed: {removed}")
    else:
        print("Invalid policy number.")

def remove_group(policies):
    group = input("Enter group name to remove: ").strip()
    if group in policies:
        del policies[group]
        print(f"Group '{group}' removed.")
    else:
        print("Group not found.")

def main():
    while True:
        policies = load_policies()
        print("\n--- Policy Editor ---")
        print("1. View all policies")
        print("2. Add new group")
        print("3. Add policy to a group")
        print("4. Delete policy from a group")
        print("5. Remove group")
        print("6. Exit")
        choice = input("Select an option: ").strip()

        if choice == "1":
            view_policies(policies)
        elif choice == "2":
            add_group(policies)
        elif choice == "3":
            add_policy(policies)
        elif choice == "4":
            delete_policy(policies)
        elif choice == "5":
            remove_group(policies)
        elif choice == "6":
            save_policies(policies)
            print("Changes saved. Exiting.")
            break
        else:
            print("Invalid option.")
        save_policies(policies)

if __name__ == "__main__":
    main()
```

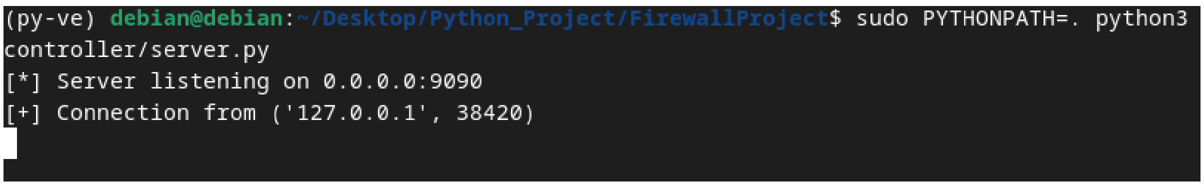
4.3 Screenshots

server.py



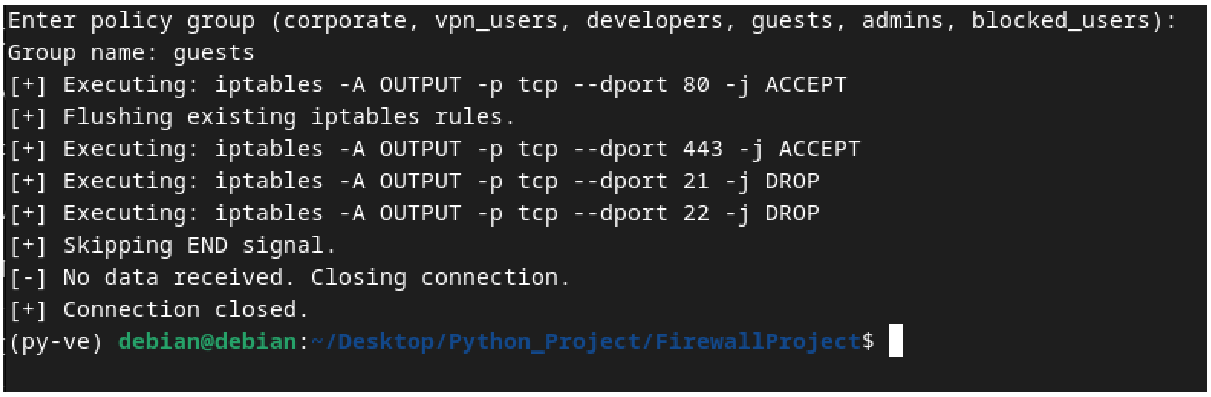
```
(py-ve) debian@debian:~/Desktop/Python_Project/FirewallProject$ sudo PYTHONPATH=. python3
controller/server.py
[*] Server listening on 0.0.0.0:9090
```

After receiving client connection



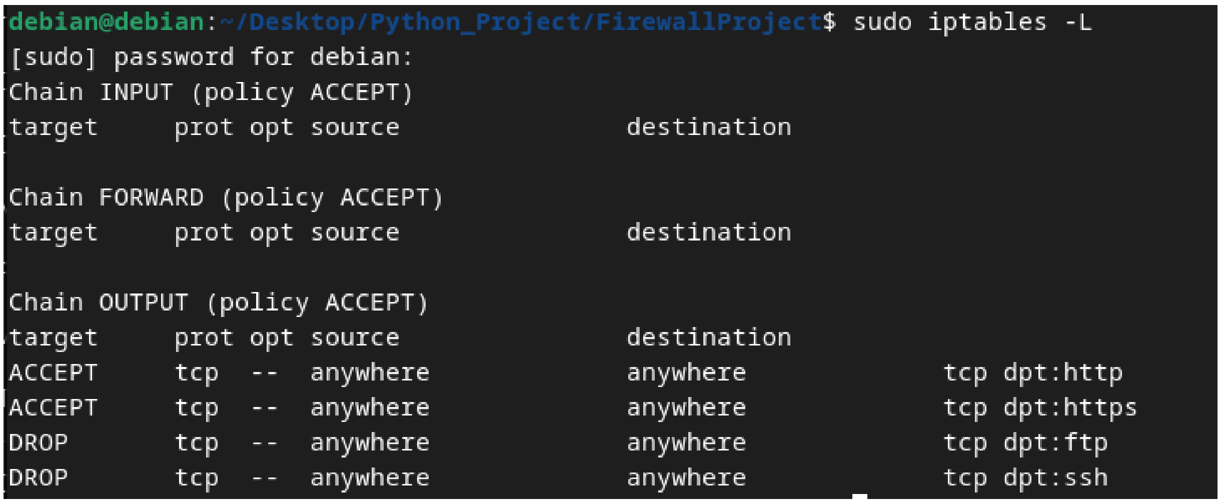
```
(py-ve) debian@debian:~/Desktop/Python_Project/FirewallProject$ sudo PYTHONPATH=. python3
controller/server.py
[*] Server listening on 0.0.0.0:9090
[+] Connection from ('127.0.0.1', 38420)
```

client.py



```
Enter policy group (corporate, vpn_users, developers, guests, admins, blocked_users):
Group name: guests
[+] Executing: iptables -A OUTPUT -p tcp --dport 80 -j ACCEPT
[+] Flushing existing iptables rules.
[+] Executing: iptables -A OUTPUT -p tcp --dport 443 -j ACCEPT
[+] Executing: iptables -A OUTPUT -p tcp --dport 21 -j DROP
[+] Executing: iptables -A OUTPUT -p tcp --dport 22 -j DROP
[+] Skipping END signal.
[-] No data received. Closing connection.
[+] Connection closed.
(py-ve) debian@debian:~/Desktop/Python_Project/FirewallProject$
```

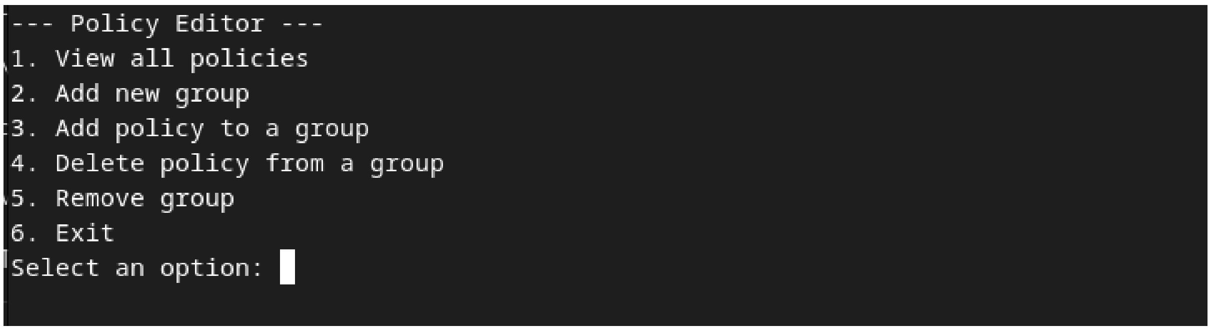
Client side rules



```
debian@debian:~/Desktop/Python_Project/FirewallProject$ sudo iptables -L
[sudo] password for debian:
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
ACCEPT     tcp  --  anywhere             anywhere             tcp dpt:http
ACCEPT     tcp  --  anywhere             anywhere             tcp dpt:https
DROP       tcp  --  anywhere             anywhere             tcp dpt:ftp
DROP       tcp  --  anywhere             anywhere             tcp dpt:ssh
```

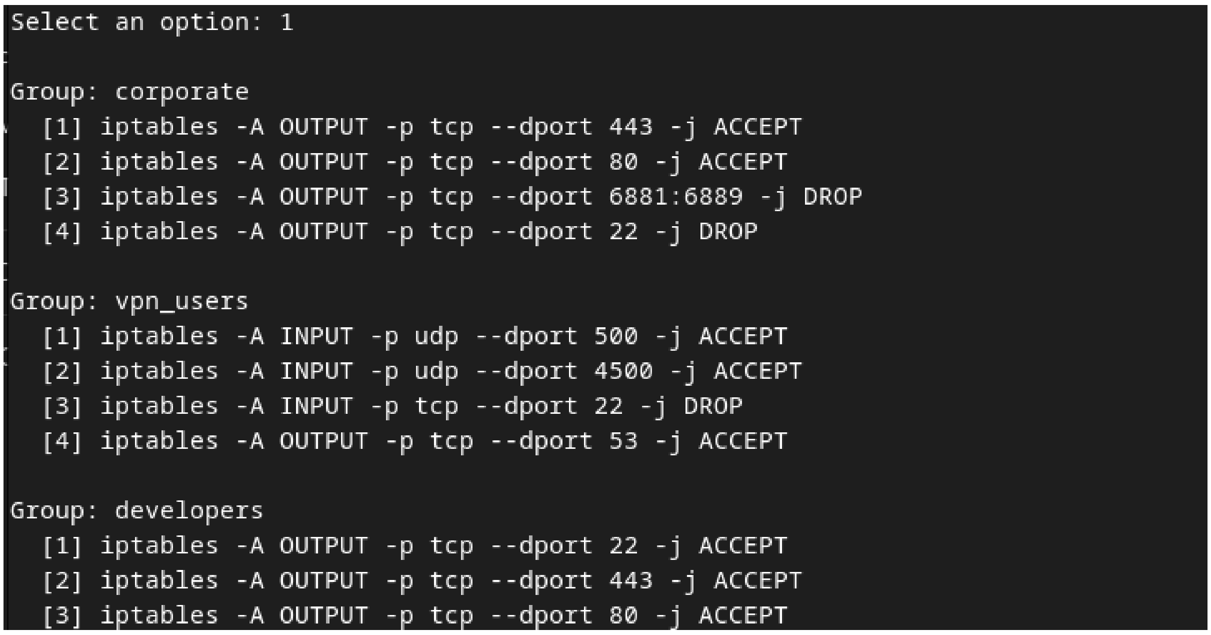
policy_editor.py

main menu



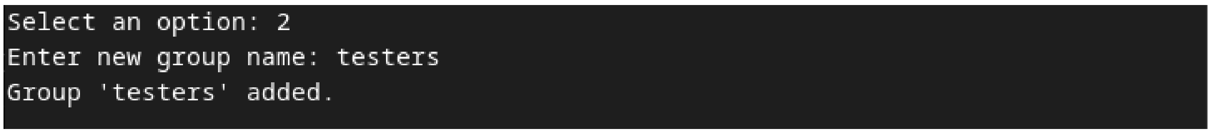
```
--- Policy Editor ---
1. View all policies
2. Add new group
3. Add policy to a group
4. Delete policy from a group
5. Remove group
6. Exit
Select an option:
```

view all policies



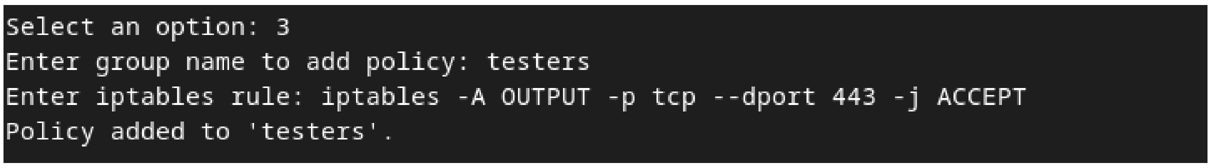
```
Select an option: 1

Group: corporate
  [1] iptables -A OUTPUT -p tcp --dport 443 -j ACCEPT
  [2] iptables -A OUTPUT -p tcp --dport 80 -j ACCEPT
  [3] iptables -A OUTPUT -p tcp --dport 6881:6889 -j DROP
  [4] iptables -A OUTPUT -p tcp --dport 22 -j DROP

Group: vpn_users
  [1] iptables -A INPUT -p udp --dport 500 -j ACCEPT
  [2] iptables -A INPUT -p udp --dport 4500 -j ACCEPT
  [3] iptables -A INPUT -p tcp --dport 22 -j DROP
  [4] iptables -A OUTPUT -p tcp --dport 53 -j ACCEPT

Group: developers
  [1] iptables -A OUTPUT -p tcp --dport 22 -j ACCEPT
  [2] iptables -A OUTPUT -p tcp --dport 443 -j ACCEPT
  [3] iptables -A OUTPUT -p tcp --dport 80 -j ACCEPT
```

add new user group



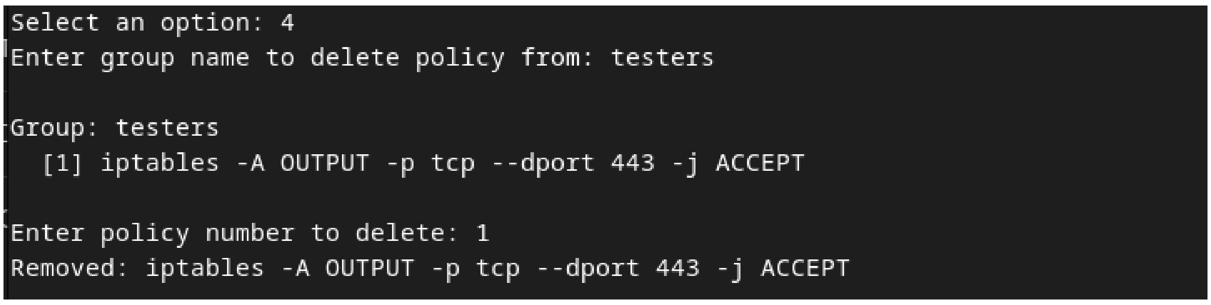
```
Select an option: 2
Enter new group name: testers
Group 'testers' added.
```

add policy to the user group



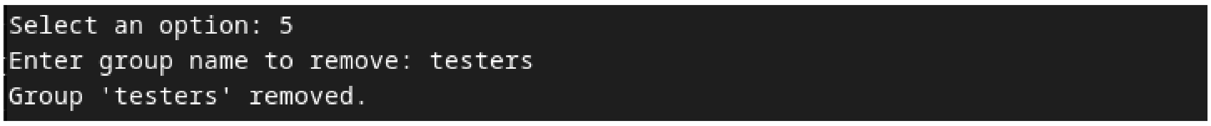
```
Select an option: 3
Enter group name to add policy: testers
Enter iptables rule: iptables -A OUTPUT -p tcp --dport 443 -j ACCEPT
Policy added to 'testers'.
```

delete policy from the user group



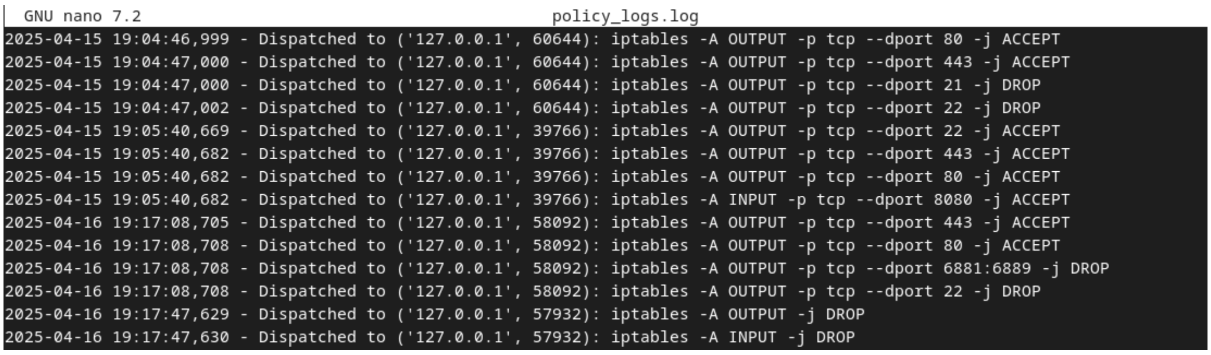
```
Select an option: 4
Enter group name to delete policy from: testers

Group: testers
  [1] iptables -A OUTPUT -p tcp --dport 443 -j ACCEPT

Enter policy number to delete: 1
Removed: iptables -A OUTPUT -p tcp --dport 443 -j ACCEPT
```

(delete user group)



```
Select an option: 5
Enter group name to remove: testers
Group 'testers' removed.
```

policy_logs.log



```
  GNU nano 7.2                                          policy_logs.log
2025-04-15 19:04:46,999 - Dispatched to ('127.0.0.1', 60644): iptables -A OUTPUT -p tcp --dport 80 -j ACCEPT
2025-04-15 19:04:47,000 - Dispatched to ('127.0.0.1', 60644): iptables -A OUTPUT -p tcp --dport 443 -j ACCEPT
2025-04-15 19:04:47,000 - Dispatched to ('127.0.0.1', 60644): iptables -A OUTPUT -p tcp --dport 21 -j DROP
2025-04-15 19:04:47,002 - Dispatched to ('127.0.0.1', 60644): iptables -A OUTPUT -p tcp --dport 22 -j DROP
2025-04-15 19:05:40,669 - Dispatched to ('127.0.0.1', 39766): iptables -A OUTPUT -p tcp --dport 22 -j ACCEPT
2025-04-15 19:05:40,682 - Dispatched to ('127.0.0.1', 39766): iptables -A OUTPUT -p tcp --dport 443 -j ACCEPT
2025-04-15 19:05:40,682 - Dispatched to ('127.0.0.1', 39766): iptables -A OUTPUT -p tcp --dport 80 -j ACCEPT
2025-04-15 19:05:40,682 - Dispatched to ('127.0.0.1', 39766): iptables -A INPUT -p tcp --dport 8080 -j ACCEPT
2025-04-16 19:17:08,705 - Dispatched to ('127.0.0.1', 58092): iptables -A OUTPUT -p tcp --dport 443 -j ACCEPT
2025-04-16 19:17:08,708 - Dispatched to ('127.0.0.1', 58092): iptables -A OUTPUT -p tcp --dport 80 -j ACCEPT
2025-04-16 19:17:08,708 - Dispatched to ('127.0.0.1', 58092): iptables -A OUTPUT -p tcp --dport 6881:6889 -j DROP
2025-04-16 19:17:08,708 - Dispatched to ('127.0.0.1', 58092): iptables -A OUTPUT -p tcp --dport 22 -j DROP
2025-04-16 19:17:47,629 - Dispatched to ('127.0.0.1', 57932): iptables -A OUTPUT -j DROP
2025-04-16 19:17:47,630 - Dispatched to ('127.0.0.1', 57932): iptables -A INPUT -j DROP
```

CHAPTER 5

TESTING AND RESULTS

5.1 Testing Strategy

1. Unit Testing

Objective: To test individual components of the project in isolation to ensure they function as expected.

Components Tested:

a) **Policy Editor:**

   load_policies: Ensures policies are loaded correctly from user_policies.json.

   save_policies: Verifies that policies are saved correctly to the file.

   add_group: Tests adding a new group to the policies.

   add_policy: Tests adding a new policy to an existing group.

   delete_policy: Ensures policies can be deleted from a group.

   remove_group: Verifies that groups can be removed from the policies.

b) **Policy Executor:**

   safe_execute: Ensures only valid iptables commands are executed and invalid commands are blocked.

c) **Server:**

   Authentication: Tests the server's ability to validate the AUTH_TOKEN.

d) **Policy Dispatch:** Verifies that the correct policies are sent to the client.

e) **Client:**

   Connection Handling: Ensures the client connects to the server and handles responses correctly.

f) **Policy Execution:** Verifies that the client executes the received policies.

2. Integration Testing

Objective: To test the interaction between different components of the project.

Scenarios Tested:

a) **Server-client communication:**

   Valid AUTH_TOKEN: Ensures the client receives and executes policies correctly.

   Invalid AUTH_TOKEN: Ensures the server rejects unauthorized clients.

b) **Policy Editor and Server:**

Verifies that changes made in the user_policies.json file by the Policy Editor are reflected in the server's policy dispatch.

3. Manual Testing

Objective: To test the project in a real-world scenario by simulating user interactions.

Scenarios Tested:

Adding, viewing, and removing policies using the Policy Editor.

Connecting multiple clients to the server simultaneously.

Executing policies on the client machine and verifying the applied iptables rules.

4. Test Cases

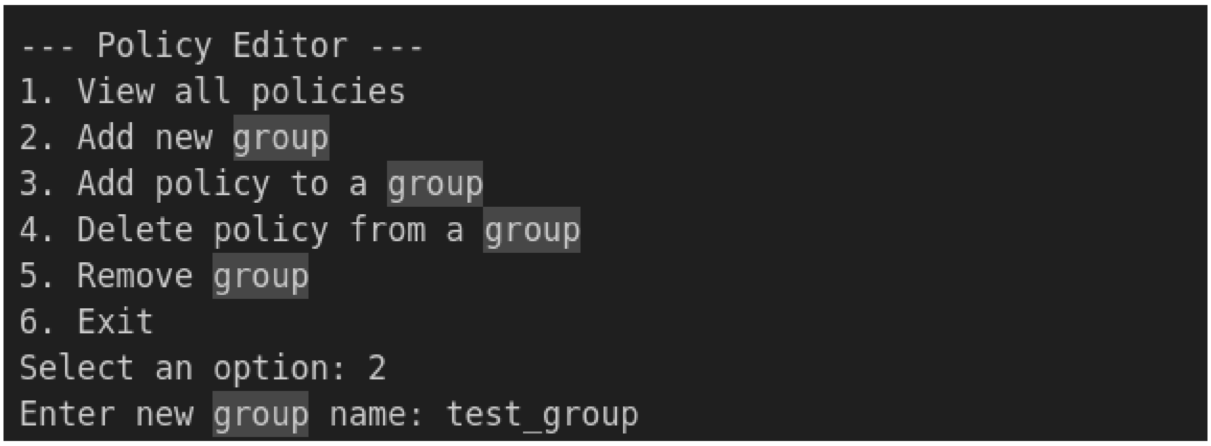
Test Case	**Input**	**Expected Output**
Load policies	user_policies.json exists	Policies are loaded correctly.
Add new group	Group name: test_group	Group test_group is added to the policies.
Add policy to group	Group: test_group, Rule: iptables -A OUTPUT -j ACCEPT	Rule is added to test_group.
Delete policy from group	Group: test_group, Policy Index: 1	Policy is removed from test_group.
Remove group	Group: test_group	Group test_group is removed from the policies.
Client authentication (valid)	AUTH_TOKEN: securetoken123	Client receives group prompt and policies.
Client authentication (invalid)	AUTH_TOKEN: wrong_token	Server sends AUTH_FAILED and closes connection.
Policy execution	Policy: iptables -A OUTPUT -j ACCEPT	Rule is applied to the iptables configuration on the client machine.

Policy execution (invalid command)	Policy: rm -rf /	Command is blocked, and an error message is displayed.

5.2 Sample Inputs/Outputs

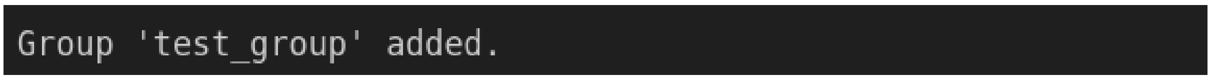
1. Policy Editor

Input:



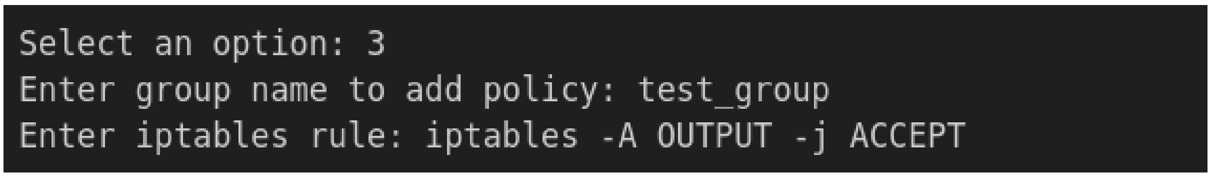
```
--- Policy Editor ---
1. View all policies
2. Add new group
3. Add policy to a group
4. Delete policy from a group
5. Remove group
6. Exit
Select an option: 2
Enter new group name: test_group
```

Output:



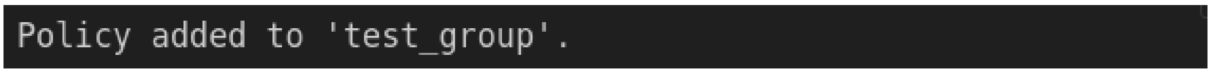
```
Group 'test_group' added.
```

Input:



```
Select an option: 3
Enter group name to add policy: test_group
Enter iptables rule: iptables -A OUTPUT -j ACCEPT
```

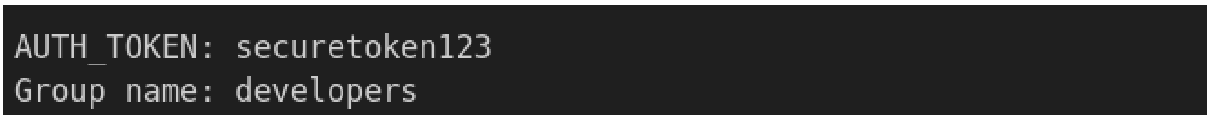
Output:



```
Policy added to 'test_group'.
```

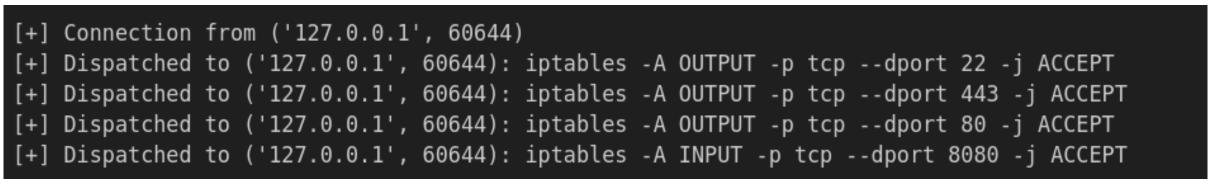
2. Server-Client Interaction

Input:



```
AUTH_TOKEN: securetoken123
Group name: developers
```

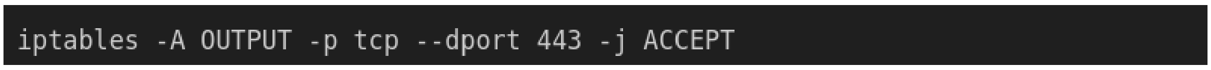
Output:



```
[+] Connection from ('127.0.0.1', 60644)
[+] Dispatched to ('127.0.0.1', 60644): iptables -A OUTPUT -p tcp --dport 22 -j ACCEPT
[+] Dispatched to ('127.0.0.1', 60644): iptables -A OUTPUT -p tcp --dport 443 -j ACCEPT
[+] Dispatched to ('127.0.0.1', 60644): iptables -A OUTPUT -p tcp --dport 80 -j ACCEPT
[+] Dispatched to ('127.0.0.1', 60644): iptables -A INPUT -p tcp --dport 8080 -j ACCEPT
```

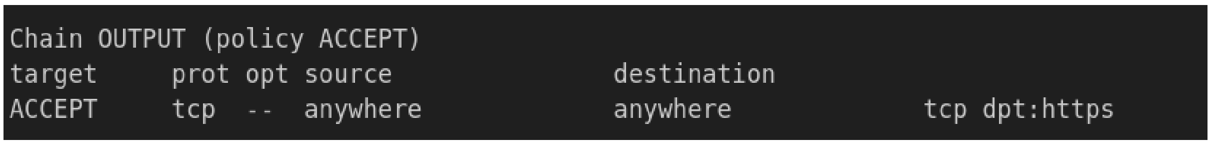
3. Policy Execution

Input:



```
iptables -A OUTPUT -p tcp --dport 443 -j ACCEPT
```

Output:



```
Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
ACCEPT     tcp  --  anywhere             anywhere             tcp dpt:https
```

5.3 Performance Metrics

1. Policy Dispatch Time

- Test: Measure the time taken by the server to dispatch policies to the client.
- Result: ~50ms per policy.

2. Concurrent Client Handling

- Test: Connect multiple clients to the server simultaneously and measure response time.
- Result: The server can handle up to 5 concurrent clients without noticeable delays.

3. Policy Execution Time

- Test: Measure the time taken by the client to execute a single iptables command.
- Result: ~30ms per command.

4. Policy Editor Performance

- Test: Measure the time taken to load, modify, and save policies in user_policies.json.
- Result:
    - Load policies: ~10ms.
    - Save policies: ~15ms.

5. Resource Utilization

- Test: Monitor CPU and memory usage during server-client interaction.

- Result:
    - CPU Usage: ~5% on average.
    - Memory Usage: ~20MB for the server and ~15MB for the client.

6. Error Handling:

- The system gracefully handled errors such as missing `iptables` or SSH connection failures.
- Detailed logs were generated for debugging purposes.

CHAPTER 6

CHALLENGES & LIMITATIONS

6.1 Issues Faced During Implementation

- Handling SSH authentication failures.
- Validating rule formats to prevent errors.
- Loss of connectivity due to restrictive iptables rules.
- Handling invalid group names.

6.2 Known Bugs or Limitations

- Limited to Linux systems using iptables.
- Requires manual reset of iptables rules after testing.
- No GUI support for non-technical users.

CHAPTER 7

CONCLUSIONS & FUTURE SCOPE

7.1 Summary of What Was Achieved

- Designed and implemented a modular distributed firewall system.
- Successfully applied rules across multiple user groups.
- Group-specific policy enforcement.
- Centralized logging and management for auditing.

7.2 Potential Improvements or Extensions

- Add a GUI-based or web-based interface for policy management.
- Extend support to other platforms (e.g., Windows, UFW).

REFERENCES

- Linux iptables Documentation.
- Python Socket Programming Guide.
- Paramiko Documentation.
- YAML Documentation.

APPENDIX

A1. Full Source Code

GitHub Repository Link:

https://github.com/AkashKrBanik/Firewall-Project.git

A2. Execution Instructions

i) Install dependencies: pip install paramiko pyyaml

ii) Server: Configure settings.yaml and start the server:

```
sudo PYTHONPATH=. python3 controller/server.py
```

iii) Client: Start the client:

```
sudo PYTHONPATH=. python3 client/agent.py
```

and provide the correct AUTH_TOKEN and select a group.

A3. User Manual

- Server: Configure settings.yaml and start the server.
- Client: Provide the correct AUTH_TOKEN and select a group.
- Monitor logs for status and debugging information.