

بخش دوم

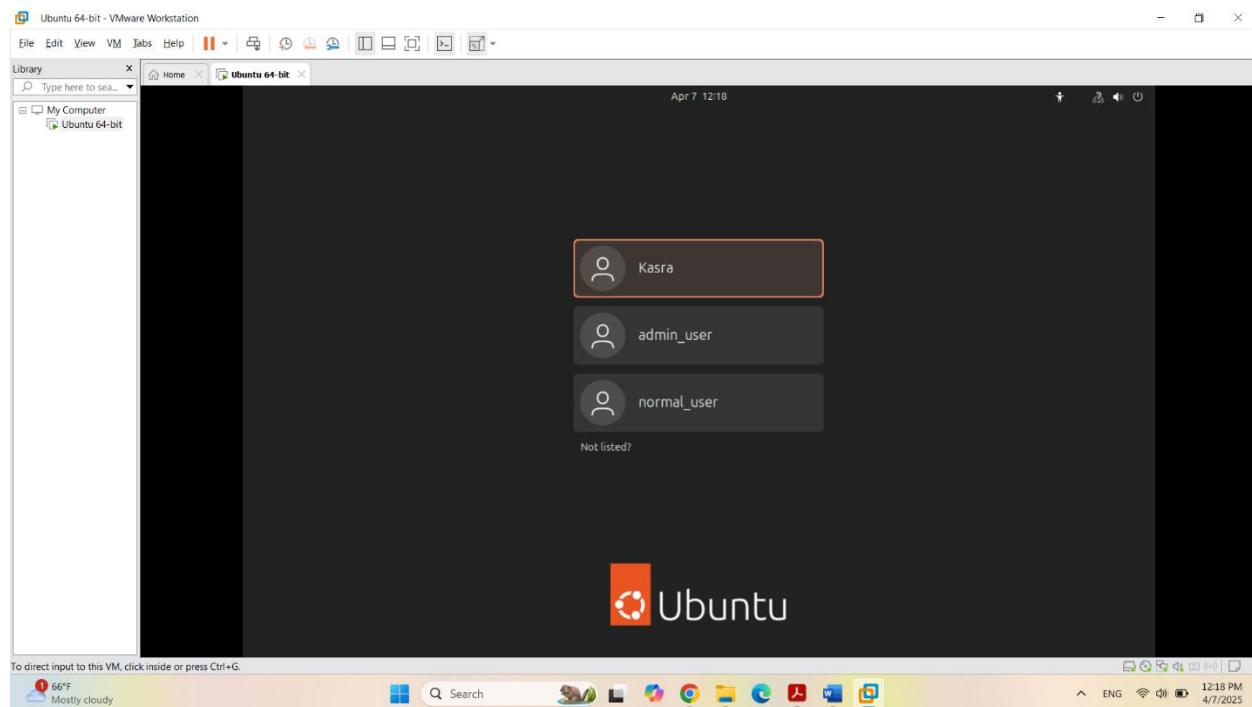
کسرا نوربخش 810100230

مقدمه

در این قسمت، سیستمی طراحی کردیم که امکان اجرای وظایف از راه دور، نظارت بر وضعیت سرور و انتقال فایلها را از طریق SSH فراهم می کند.

راه اندازی سرور SSH

در این گام تمامی دستور های گفته شده زده شد تا SSH Server نصب و راه اندازی شود. سپس دستورات مورد نیاز برای اضافه کردن کاربران را زدیم و هم چنین با استفاده از دستور: `sudo usermod -aG sudo admin_user` توانستیم کاربر مدیر را به گروه sudo اضافه کنیم تا تمامی دسترسی های لازم را داشته باشد.



عکس ۱: با یکبار ریبوت کردن، مشاهده می شود که کاربران جدید اضافه شده اند

```
kasra@kasra-None:~$ groups admin_user
admin_user : admin_user sudo users
kasra@kasra-None:~$ groups normal_user
normal_user : normal_user users
kasra@kasra-None:~$
```

عکس 2: مشاهده می شود که کاربران موجود اند و کاربر مدیر در گروه Sudo قرار گرفته است

راه اندازی محیط Python و نصب وابستگی ها

در این گام با استفاده از دستور: `sudo apt install python3 python3-pip -y` پرداختیم. سپس برای آن که بتوانیم کتابخانه های گفته شده را هم نصب کنیم، چون هسته اوبونتو ایراد می گرفت به `global` نصب کردن آن ها از یک `Virtual Environment` پایتون استفاده کردیم که آن را `ssh_env` نامیدیم. دستور های زده شده:

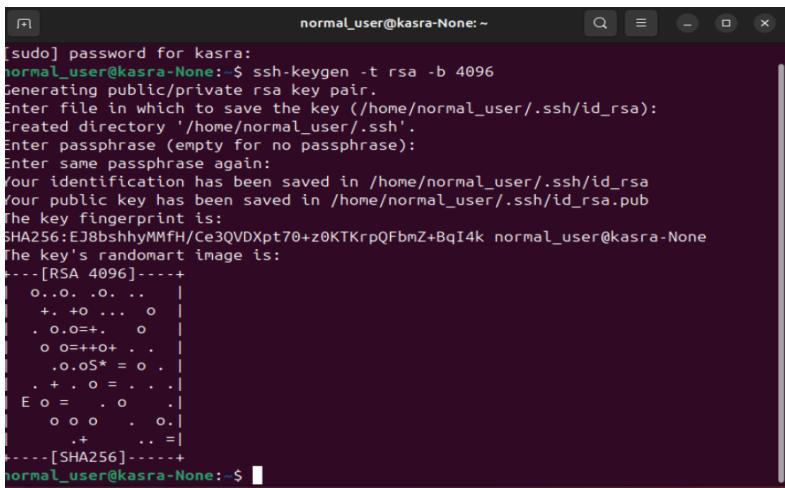
```
sudo apt install python3-venv python3-full -y
python3 -m venv ssh_env
source ssh_env/bin/activate
pip install paramiko psutil
```

```
kasra@kasra-None:~$ source ssh_env/bin/activate
(ssh_env) kasra@kasra-None:~$
```

عکس 3: نحوه فعال کردن `Virtual Environment` پایتون ساخته شده

سپس به سراغ درست کردن کلید های عمومی و خصوصی برای کاربر `normal` رفتیم که برای این کار نیاز بود تا به عنوان کاربر `normal` وارد شویم که با استفاده از دستور: `sudo -i -u normal_user`، این کار را کردیم و سپس دستور نوشته شده در صورت پروژه را وارد کردیم. این فرایند را برای `admin_user` هم انجام دادیم. برای انتقال کلید عمومی هم از طریق زیر اقدام کردیم:

```
sudo su - admin_user
mkdir -p ~/.ssh
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
chmod 600 ~/.ssh/authorized_keys
chmod 700 ~/.ssh
exit
```



```

[sudo] password for kasra:
normal_user@kasra-None:~$ ssh-keygen -t rsa -b 4096
Generating public/private rsa key pair.
Enter file in which to save the key (/home/normal_user/.ssh/id_rsa):
Created directory '/home/normal_user/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/normal_user/.ssh/id_rsa
Your public key has been saved in /home/normal_user/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:EJ8bshhMMFH/Ce3QVDxpt70+z0KTKrpQFbmZ+BqI4k normal_user@kasra-None
The key's randomart image is:
----[RSA 4096]----+
 .o.o. .o. .
 +. +o ... o |
 . o.=+.. o |
 o o+=+o+ . .
 .o.oS* = o . |
 .+ . o = . . |
 E o = . o . |
 o o o . o. |
 .+ . . =|
----[SHA256]----+
normal_user@kasra-None: ~

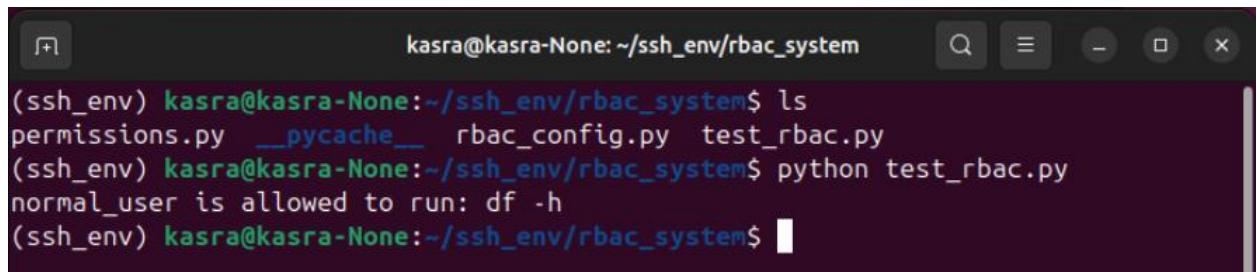
```

عکس 4: ساخته شدن کلید ها برای normal_user

پیاده سازی کلاینت SSH با کنترل دسترسی مبتنی بر نقش

در بخش تعریف نقش ها و مجوز ها، ابتدا در داخل دایرکتوری با نام `ssh_env`، یک دایرکتوری با نام `rbac_system` ساختیم که در آن یک فایل با نام `rbac_config.py` موجود است که 2 نقش `admin_user` و `normal_user` را دارد و در بخش `allowed_commands` (تمامی دستورات) برای آن قرار گرفته است و نقش `admin` را دارد و در بخش `normal_user` که نقش `user` را دارد و برخی دستورات مانند: `uptime`, `ps aux`, `df -h`, ... فقط برای آن مجاز می باشد، را دارا می باشد.

ضمناً در فایل `permissions.py` یک helper function را داریم که چاپ می کند دستور وارد شده برای این کاربر مجاز می باشد یا خیر، که نحوه اجرا و تست این بخش را با استفاده از اجرا کردن کد تست `test_rbac.py` می توانیم ببینیم:



```

kasra@kasra-None:~/ssh_env/rbac_system$ ls
permissions.py __pycache__ rbac_config.py test_rbac.py
(kasra@kasra-None:~/ssh_env/rbac_system)$ python test_rbac.py
normal_user is allowed to run: df -h
(kasra@kasra-None:~/ssh_env/rbac_system$)

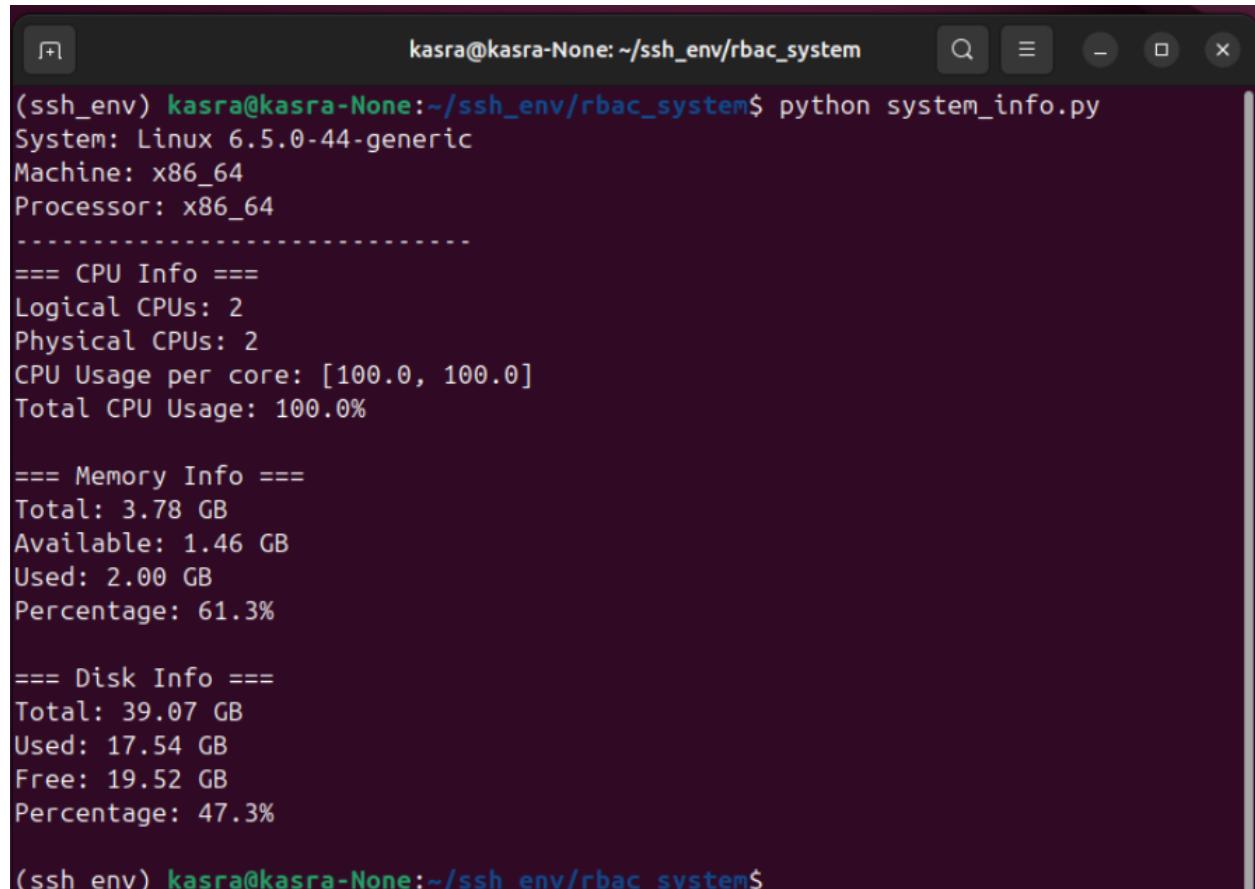
```

عکس 5: اطمینان از صحت کد های نوشته شده

حال 2 تابع اتصال SSH به سرور یعنی `connect_ssh` و اجرای دستورات از راه دور با توجه به نقش یعنی `hostname`, `ssh_client.py` را در فایل `run_command` نوشتیم. که در اینجا تابع اتصال، اطلاعاتی از قبیل `port`, `username`, `key_path`

نوشته شده، پیام خطای مناسبی چاپ می‌گردد. تابع اجرای دستورات هم ابتدا بررسی می‌کند که دستور وارد شده توسط کاربر، دارای دسترسی و مطابق نقش باشد، سپس آن را اجرا می‌کند و نتیجه را بر چاپ می‌کند.

در بخش آخر این قسمت به سراغ نوشتن اسکریپت نظارت بر سلامت سرور رفتیم. کد این بخش در فایل `system_info.py` موجود است که اطلاعات `cpu`, `memory`, `disk` را می‌دهد.



```
(ssh_env) kasra@kasra-None:~/ssh_env/rbac_system$ python system_info.py
System: Linux 6.5.0-44-generic
Machine: x86_64
Processor: x86_64
-----
==== CPU Info ===
Logical CPUs: 2
Physical CPUs: 2
CPU Usage per core: [100.0, 100.0]
Total CPU Usage: 100.0%

==== Memory Info ===
Total: 3.78 GB
Available: 1.46 GB
Used: 2.00 GB
Percentage: 61.3%

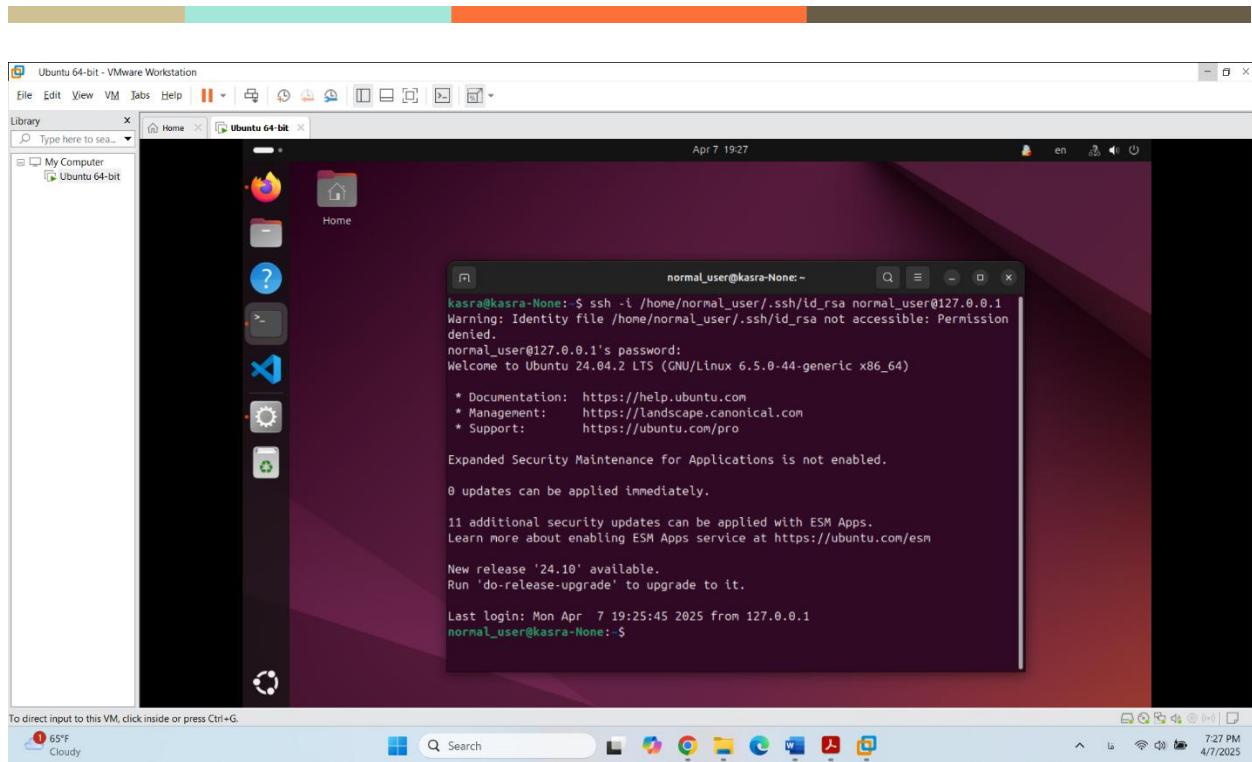
==== Disk Info ===
Total: 39.07 GB
Used: 17.54 GB
Free: 19.52 GB
Percentage: 47.3%
(ssh_env) kasra@kasra-None:~/ssh_env/rbac_system$
```

عکس 6: خروجی اسکریپت نظارت بر سلامت سرور

ضمناً در این بخش با استفاده از 2 خط:

```
ssh -i /home/admin_user/.ssh/id_rsa admin_user@127.0.0.1
ssh -i /home/normal_user/.ssh/id_rsa normal_user@127.0.0.1
```

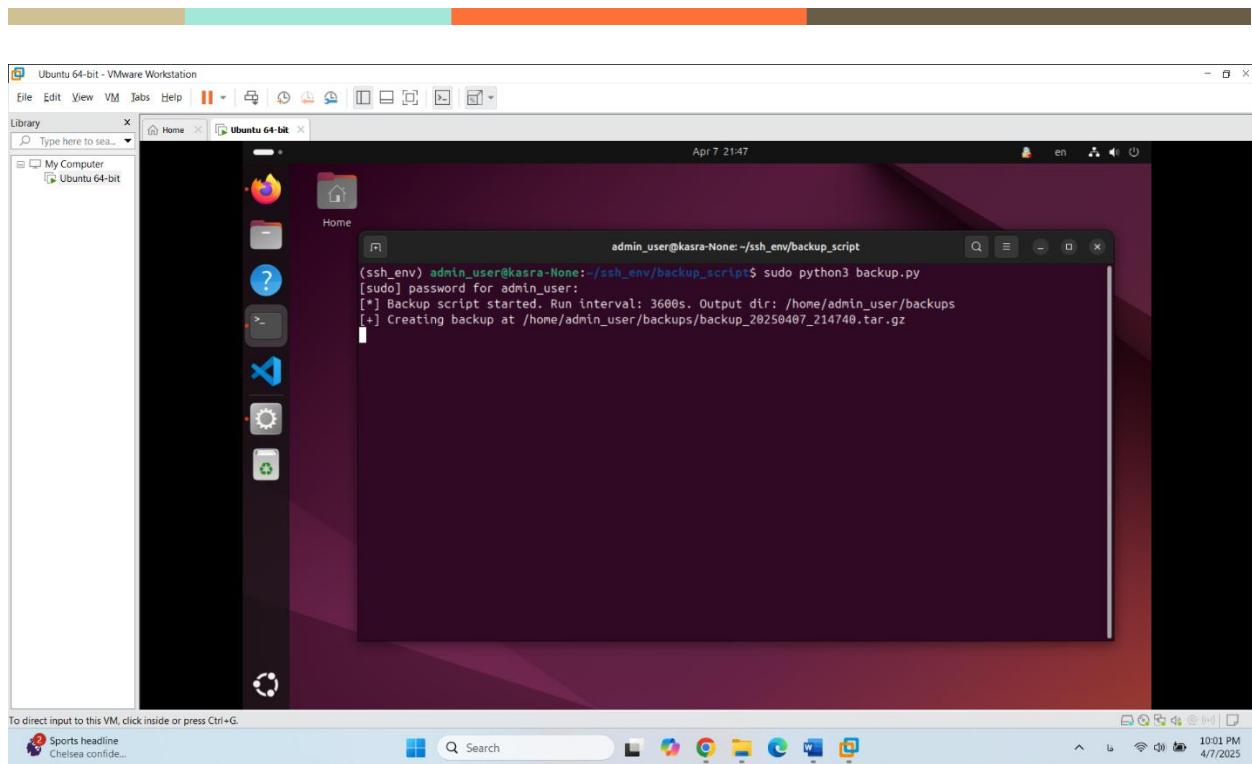
اتصال صحیح 2 کاربر را بررسی نمودیم که موفقیت آمیز بود. حالت `:normal_user`



عکس 7: اتصال normal_user

پیاده سازی اتوماسیون سرور (پشتیبان گیری و به روزرسانی سیستم)

در این بخش یک دایرکتوری با نام config.json ایجاد کردیم که در آن 2 فایل backup_script و backup.py می باشد، و که در ابتدا تنظیمات را از فایل config می خواند و سپس و بکاپ از دایرکتوری های /etc و /home را به صورت فشرده شده در مسیر خروجی قرار می دهد.



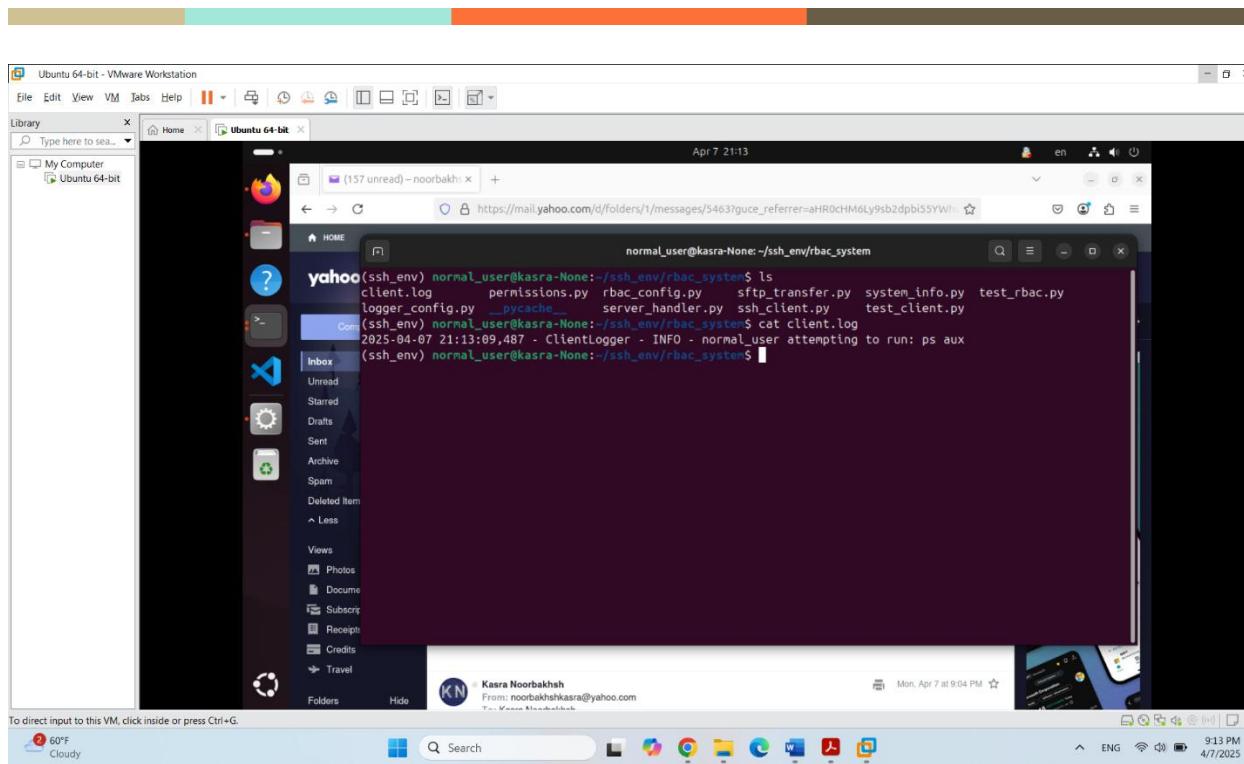
عکس 8: بکاپ گیری admin_user

انتقال فایل این با SFTP

در این بخش داخل دایرکتوری `rbac_system`، یک فایل با نام `sftp_transfer.py` ایجاد کردیم با استفاده از تابع `connect_sftp` ابتدا، اتصال را برقرار می کند و در ادامه با توجه به اینکه درخواست کاربر `upload` و یا `download` بوده است، عملیات را انجام می دهد.

امتیازی

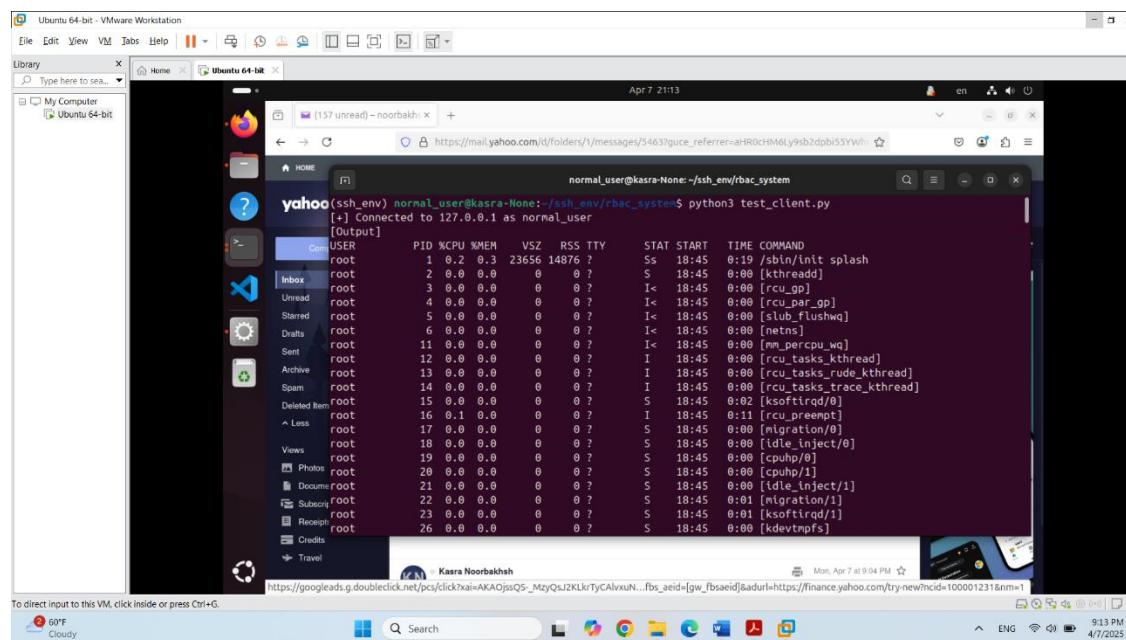
در این بخش سیستم `logging` را پیاده سازی کردیم که به ازای هر دستوری که هر کدام از کاربران اجرا می کنند، در مسیر `rbac_system` شان یک فایل `client.log` ایجاد می شود (اگر از قبل ایجاد نشده بود) و در آن `log` فعالیت های آن نوشته می شود.



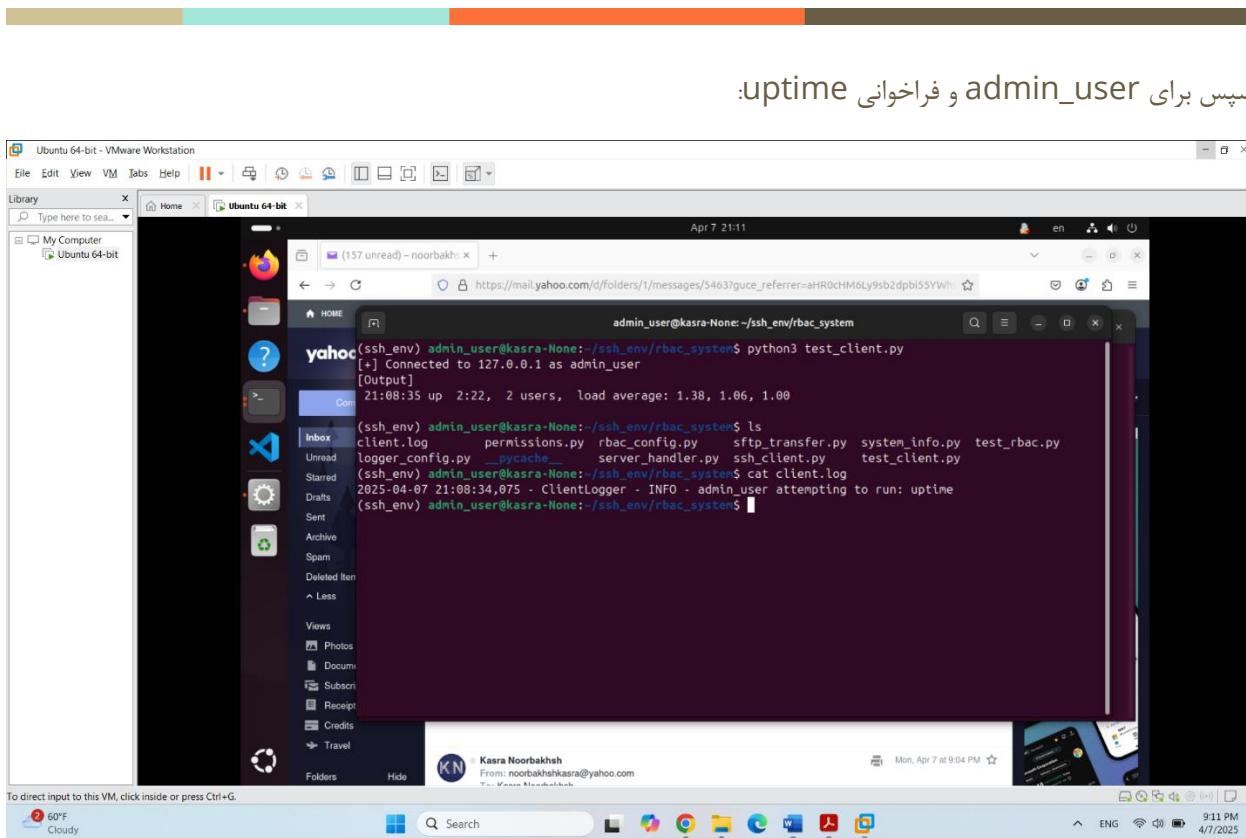
عکس 9: خروجی لاغ فایل برای نرم‌الیوزر

اجرای نهایی برنامه

در اینجا با اجرا کردن دستور: `python3 test_client.py` می‌توانیم تمامی functionality‌ها را تست کنیم که به عنوان مثال ابتدا آن را برای `normal_user` و فراخوانی `ps aux` آوردم:



عکس 10: خروجی تست `normal_user`

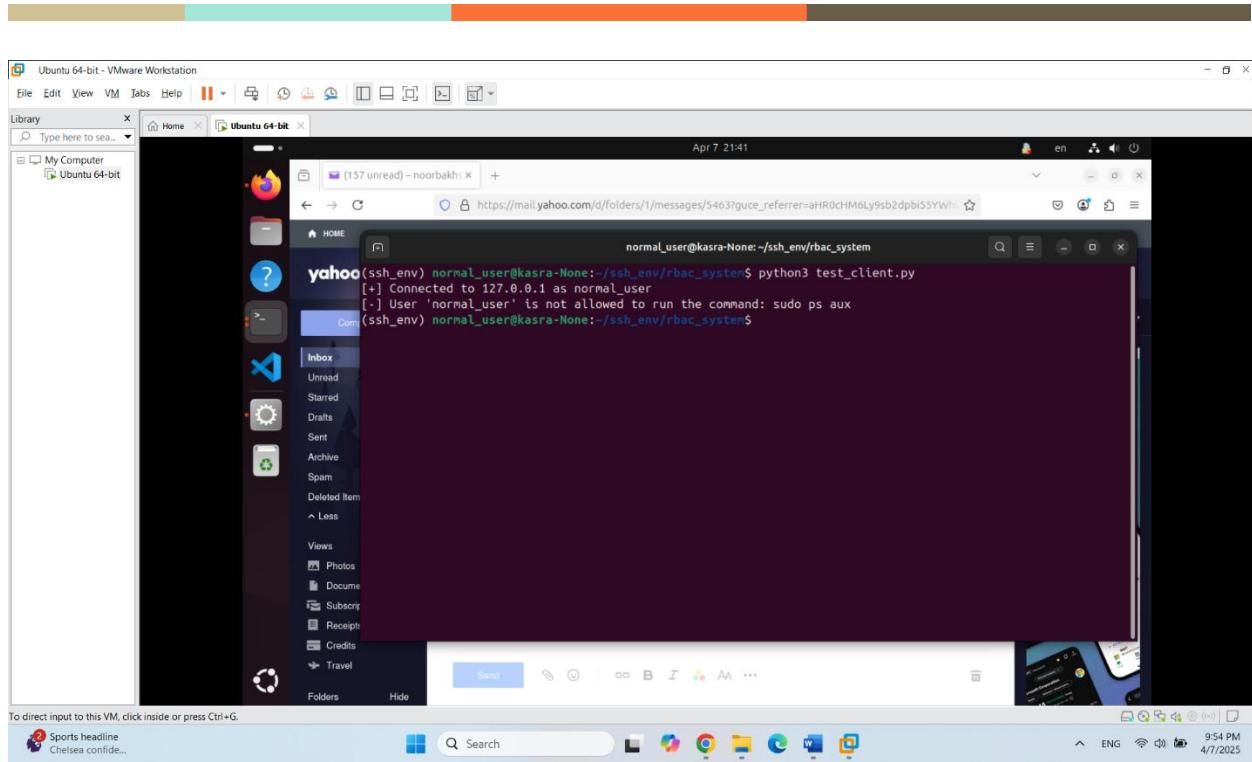


عکس 11: خروجی تست admin_user

برای اجرا کردن دستور های مختلف چه برای `admin_user` و چه برای `normal_user` نیاز است که به داخل فایل آن ها رفته و این بخش را تغییر دهیم:

```
HOST = "127.0.0.1"
PORT = 22
USERNAME = "admin_user" # or "normal_user"
KEY_PATH = "/home/admin_user/.ssh/id_rsa" # Path to private key
COMMAND = "df -h"
```

ضمناً بررسی نمودیم که نقش ها رعایت شده باشد و فرضاً کاربر `normal_user` نتواند دستور هایی مانند `sudo` را بزند:



عکس 12: نمونه عدم دسترسی normal_user

لینک های استفاده شده

<https://chatgpt.com/share/67f6576f-0938-800d-b098-fa2c03c95208> .1

استفاده از هوش مصنوعی ChatGPT

برای پیش برد این بخش از ChatGPT استفاده شد که لینک چت کامل آن در 1 موجود است.

چت کامل:

I am working with Ubuntu 24 on a VM on my VMWare Workstation. I want to build a system that has the ability to run commands from remote, control server state and transfer files via SSH. we have RBAC to restrict user and admin users to only have their commands. With this introduction, I will tell you the next steps in the next messages.

ChatGPT said:

Memory updated

Got it! Sounds like you're building a secure remote management system with RBAC and SSH capabilities—very cool. Go ahead and let me know the next steps when you're ready.

You said:

first we must instal ssh server, using sudo apt install openssh-server and after that sudo systemctl enable ssh and sudo systemctl start ssh. I've done this part, I am going to tell you more in the next message.

ChatGPT said:

Perfect — you've got the SSH server installed and running, which is the foundation for everything else. Go ahead and send the next part whenever you're ready!

You said:

now we must add 2 users, manager that is the admin_user that has the complete access to server and normal user that is the normal_user that has limited acces I used sudo adduser admin_user and sudo adduser normal_user in the terminal. after that we must allow the admin_user to use some commands like sudo. we must achieve this with adding the admin_user to the sudo group. tell me how to do this and after that I will tell you the next steps in the next messages/

ChatGPT said:

You're right on track! To allow admin_user to use sudo, you just need to add them to the sudo group like this:

Bash

Copy

Edit

sudo usermod -aG sudo admin_user

What this does:

-aG adds (a) the user to the group (G) without removing them from any other groups.

sudo is the group that controls admin privileges on Ubuntu.

After running that, you can verify it with:

bash

Copy

Edit

groups admin_user

Once you've done that, let me know the next steps!

You said:

Now how can I install python and some packages like: paramiko and psutil.

ChatGPT said:

To install Python and the packages like paramiko and psutil on your Ubuntu 24 system, follow these steps:

1. Install Python 3 and pip (if not already installed)

bash

Copy

Edit

sudo apt update

sudo apt install python3 python3-pip -y

You can confirm the installation with:

bash

Copy

Edit

python3 --version

pip3 --version

2. Install paramiko and psutil using pip

bash

Copy

Edit

pip3 install paramiko psutil

If you're doing this globally (for all users), and you're logged in as a user with sudo access, you can also do:

bash

Copy

Edit

sudo pip3 install paramiko psutil

But installing as the user is often safer and cleaner unless system-wide access is needed.

Let me know once you're done or if you plan to use a virtual environment for this, and I'll help with that too. Ready for the next step when you are!

You said:

now we want to have authentication system based on SSH. instead using password we want to generate a pair of SSH keys and put the public key on the server. they told me to use ssh-keygen -t rsa -b 4096. I will tell you the next step in the next message.

ChatGPT said:

Perfect — using SSH key-based authentication is more secure than passwords.

To generate a secure key pair on the client machine:

bash

Copy

Edit

ssh-keygen -t rsa -b 4096

This creates a 4096-bit RSA key pair.

It will ask you where to save the key (you can press Enter to use the default: `~/.ssh/id_rsa`)

Optionally, you can set a passphrase for added security.

To copy the public key to the server (assuming your user is `admin_user` or `normal_user` on the server):

bash

Copy

Edit

ssh-copy-id admin_user@<server_ip>

Or manually:

bash

Copy

Edit

On client:

cat `~/.ssh/id_rsa.pub`

Copy the output, then on the server (as the target user):

mkdir -p `~/.ssh`

echo "<copied-public-key>" >> `~/.ssh/authorized_keys`

chmod 600 `~/.ssh/authorized_keys`

chmod 700 `~/.ssh`

Let me know when you're ready for the next part!

You said:

In the previous step, when I entered the installation command this was the output:

kasra@kasra-None:~\$ pip3 install paramiko psutil

error: externally-managed-environment

✗ This environment is externally managed

↳ To install Python packages system-wide, try apt install

python3-xyz, where xyz is the package you are trying to
install.

If you wish to install a non-Debian-packaged Python package,
create a virtual environment using python3 -m venv path/to/venv.

Then use path/to/venv/bin/python and path/to/venv/bin/pip. Make
sure you have python3-full installed.

If you wish to install a non-Debian packaged Python application,
it may be easiest to use pipx install xyz, which will manage a
virtual environment for you. Make sure you have pipx installed.

See /usr/share/doc/python3.12/README.venv for more information.

note: If you believe this is a mistake, please contact your Python installation or OS distribution provider. You can override this, at the risk of breaking your Python installation or OS, by passing --break-system-packages.

hint: See PEP 668 for the detailed specification.

ChatGPT said:

Great catch — this is a new safety feature in Ubuntu 24.04 (and other recent Debian-based systems) where Python now prevents global package installation with pip3 to protect the system Python environment. It's part of PEP 668.

You have 3 clean and safe options — pick whichever suits your setup:

Option 1: Use a Virtual Environment (Recommended)

This keeps your packages isolated from the system.

bash

Copy

Edit

sudo apt install python3-venv python3-full -y

python3 -m venv ssh_env

source ssh_env/bin/activate

pip install paramiko psutil

Now whenever you're working on your project, activate the environment with:

bash

Copy

Edit

source ssh_env/bin/activate

And you can run Python scripts using just python inside that environment.

Option 2: Use pipx (For Installing Python CLI Tools)

If you just need tools (not libraries inside your own scripts), pipx is great:

bash

Copy

Edit

sudo apt install pipx

pipx install paramiko psutil

Option 3: Force system installation (Not recommended)

If you're really sure you want to install globally:

bash

Copy

Edit

pip3 install paramiko psutil --break-system-packages

But use this with caution as it can break system-managed Python packages.

Let me know which method you'd like to go with, and I'll help you further if needed. Ready for the next step whenever you are!

You said:

I used Option1. Now lets go to the next step that is implementing the SSH client with RBAC. First I want to declare roles and certificates: we must design the system in a way that roles and certificates are declared for different users. for example admin_user can run all of the commands in the system. but normal_user can only see the system health and cannot do vital things. I will tell you the next step in the next message.

ChatGPT said:

Awesome! You're building this with a clean and secure structure — love that.

So for this step, we're focusing on defining roles and permissions, which will be used later in your SSH client to restrict command execution based on the user's role.

Here's a clean way to structure it in Python, assuming this logic will be part of your SSH client script:

Step 1: Define Roles and Their Allowed Commands

python

Copy

Edit

```
# rbac_config.py
```

```
RBAC = {
```

```
    "admin_user": {
```

```
        "role": "admin",
```

```
        "allowed_commands": ["*"] # admin can run everything
```

```
},  
"normal_user": {  
    "role": "user",  
    "allowed_commands": [  
        "uptime",  
        "df -h",  
        "free -m",  
        "ps aux",  
        "top -b -n1"  
    ]  
}  
}  
}
```

Step 2: A Helper Function to Check Permissions

python

Copy

Edit

permissions.py

```
from rbac_config import RBAC  
  
def is_command_allowed(user, command):  
    if user not in RBAC:  
        return False  
    allowed = RBAC[user]["allowed_commands"]  
    if "*" in allowed:  
        return True  
    return command.strip() in allowed
```

Example Usage:

python

Copy

Edit

```
user = "normal_user"  
command = "df -h"  
  
if is_command_allowed(user, command):  
    print(f"{user} is allowed to run: {command}")  
else:
```



```
print(f"{user} is NOT allowed to run: {command}")
```

This will let you:

Control access centrally.

Extend the system easily with new roles or restrictions.

Later integrate this into your SSH client code before sending any command to the remote server.

Let me know when you're ready for the next step!

You said:

So, I should create these 3 python files under my ssh_env and use this codes on them right?

ChatGPT said:

Exactly! Here's how to organize it step by step inside your ssh_env project folder:

1. Navigate to your virtual environment project folder

If you're in your home directory and created the virtual environment as:

bash

Copy

Edit

python3 -m venv ssh_env

Then do:

bash

Copy

Edit

cd ssh_env

mkdir rbac_system

cd rbac_system

2. Create the Python files inside rbac_system/

Use your preferred editor (nano, vim, or VS Code). For example:

bash

Copy

Edit

nano rbac_config.py

Paste this in:

python

Copy

Edit

rbac_config.py

```
RBAC = {  
    "admin_user": {  
        "role": "admin",  
        "allowed_commands": ["*"] # admin can run everything  
    },  
    "normal_user": {  
        "role": "user",  
        "allowed_commands": [  
            "uptime",  
            "df -h",  
            "free -m",  
            "ps aux",  
            "top -b -n1"  
        ]  
    }  
}
```

Then create the permissions helper:

bash

Copy

Edit

nano permissions.py

Paste:

python

Copy

Edit

permissions.py

```
from rbac_config import RBAC  
  
def is_command_allowed(user, command):  
  
    if user not in RBAC:  
        return False  
  
    allowed = RBAC[user]["allowed_commands"]  
  
    if "*" in allowed:  
        return True  
  
    return command.strip() in allowed
```

And finally, you can test it with a simple test script:

bash

Copy

Edit

nano test_rbac.py

Paste:

python

Copy

Edit

test_rbac.py

```
from permissions import is_command_allowed
```

```
user = "normal_user"
```

```
command = "df -h"
```

```
if is_command_allowed(user, command):
```

```
    print(f"{user} is allowed to run: {command}")
```

else:

```
    print(f"{user} is NOT allowed to run: {command}")
```

3. Run your test script

Make sure your virtual environment is activated:

bash

Copy

Edit

source ./bin/activate

python test_rbac.py

You should see:

vbnet

Copy

Edit

normal_user is allowed to run: df -h

Let me know once you're done, or if you're ready for the SSH command execution integration!

You said:

Now we should develop function for connecting SSH to server. we must now right a code that connects to the server and using SSH, sends the commands. we must make sure that the SSH connection is safely initiated and if there were any errors, print out the proper error message.

ChatGPT said:

Perfect! Now it's time to write the SSH connection and command execution logic using Paramiko, with safe error handling.

Let's call this file: ssh_client.py — and place it alongside your other files (rbac_config.py, permissions.py, etc.)

ssh_client.py: Secure SSH Connection + Command Execution

python

Copy

Edit

```
# ssh_client.py
```

```
import paramiko
```

```
from permissions import is_command_allowed
```

```
def connect_ssh(hostname, port, username, key_path):
```

```
    try:
```

```
        key = paramiko.RSAKey.from_private_key_file(key_path)
```

```
        client = paramiko.SSHClient()
```

```
        client.set_missing_host_key_policy(paramiko.AutoAddPolicy()) # Accept unknown host keys
```

```
        client.connect(hostname=hostname, port=port, username=username, pkey=key)
```

```
        print(f"[+] Connected to {hostname} as {username}")
```

```
        return client
```

```
    except paramiko.AuthenticationException:
```

```
        print("[-] Authentication failed. Check the SSH key or username.")
```

```
    except paramiko.SSHException as e:
```

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

```
    except Exception as e:
```

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

```
    return None
```

```
def run_command(client, user, command):
```

```
    if not is_command_allowed(user, command):
```

```
        print(f"[-] User '{user}' is not allowed to run the command: {command}")
```

```
    return
```

```
try:
```



```
stdin, stdout, stderr = client.exec_command(command)

output = stdout.read().decode()
error = stderr.read().decode()

if output:
    print(f"[Output]\n{output}")

if error:
    print(f"[Error]\n{error}")

except Exception as e:
    print(f"[-] Failed to execute command: {e}")
```

Example usage script: main.py

python

Copy

Edit

main.py

```
from ssh_client import connect_ssh, run_command

HOST = "your.server.ip.or.hostname"

PORT = 22

USERNAME = "admin_user" # or "normal_user"

KEY_PATH = "/home/yourusername/.ssh/id_rsa" # Path to private key

COMMAND = "df -h"

client = connect_ssh(HOST, PORT, USERNAME, KEY_PATH)

if client:
    run_command(client, USERNAME, COMMAND)
    client.close()
```

Notes:

Make sure the private key at KEY_PATH exists and is readable.

The server should have the matching public key in the user's ~/.ssh/authorized_keys.

You can expand this to prompt for input or accept commands from arguments later.

Let me know if you want to add file transfer (SFTP) next, or improve the RBAC logic with more granularity!

You said:

next I wanted to write the run command function to run the commands with RBAC. in this function normal_user or admin_user can do different things. for admin_user system commands like taking backup, update must be feasible but for normal_user just commands related to system properties supervision is allowed. with This information, the run_command function that you wrote were ok?

ChatGPT said:

Great question — yes, the run_command() function I provided already supports RBAC, but we can make it even more powerful and structured for your use case.

Let's walk through how it works, and what improvements you can make:

- Recap of the Current run_command() function:

python

Copy

Edit

```
def run_command(client, user, command):
```

```
    if not is_command_allowed(user, command):  
        print(f"[-] User '{user}' is not allowed to run the command: {command}")  
        return
```

```
...
```

This already:

- Checks if the user is allowed to run the command.
- Blocks commands not allowed for that specific role.
- Only runs the command if permitted.

So it's RBAC-aware.

🔧 Improvements for clarity and scalability:

1. Make allowed_commands more readable by giving names or tags:

Update your rbac_config.py like this:

python

Copy

Edit

```
RBAC = {
```

```
    "admin_user": {  
        "role": "admin",  
        "allowed_commands": ["*"] # All commands allowed  
    },  
    "normal_user": {  
        "role": "user",  
        "allowed_commands": [  
            "uptime",  
            "df -h",  
            "free -m",
```



```

    "ps aux",
    "top -b -n1",
    "whoami",
    "hostname",
]
}

}

```

2. Make command comparison more flexible:

Right now it only matches exact strings. But real command input may include options, arguments, or extra spaces. Let's use regex or partial matching.

Update permissions.py:

```

python
Copy
Edit
import re
from rbac_config import RBAC

```

```

def is_command_allowed(user, command):
    if user not in RBAC:
        return False
    allowed = RBAC[user]["allowed_commands"]
    if "*" in allowed:
        return True
    for allowed_cmd in allowed:
        # Basic pattern match: does command start with allowed command?
        if command.strip().startswith(allowed_cmd):
            return True
    return False

```

This means:

df -h matches df -h

top -b -n1 matches top -b -n1

uptime matches uptime

You can also use `re.match()` if you want advanced matching later.

So yes — the function is solid for RBAC and extensible.

You can now safely let:

`admin_user` run things like `sudo apt update`, `tar -czf backup.tar.gz /etc`, etc.

`normal_user` only run non-destructive, read-only system info commands.

Let me know if you'd like to integrate command categories, logs, or SFTP next!

You said:

I want to enter this command: `ssh-keygen -t rsa -b 4096`

for my `normal_user`, how should I achieve this?

ChatGPT said:

To generate an SSH key pair for your `normal_user`, you need to switch to that user and run the `ssh-keygen` command from their context — this way the key pair gets stored in their home directory (`/home/normal_user/.ssh/`), not yours.

Step-by-step instructions:

◊ 1. Switch to `normal_user`:

From your current terminal:

`bash`

Copy

Edit

`sudo -i -u normal_user`

This logs you in as `normal_user`.

♀ You'll now be inside `/home/normal_user`.

◊ 2. Generate SSH key pair:

Now run:

`bash`



Copy

Edit

ssh-keygen -t rsa -b 4096

It will ask:

bash

Copy

Edit

Enter file in which to save the key (/home/normal_user/.ssh/id_rsa):

Just press Enter to use the default location.

Then:

Enter a passphrase (or leave blank).

Confirm passphrase.

You'll see output like:

swift

Copy

Edit

Your identification has been saved in /home/normal_user/.ssh/id_rsa.

Your public key has been saved in /home/normal_user/.ssh/id_rsa.pub.

◊ 3. Exit back to your main user:

bash

Copy

Edit

exit

Now you're back as your original user (e.g., kasra).

Done! You now have:

/home/normal_user/.ssh/id_rsa — the private key.

/home/normal_user/.ssh/id_rsa.pub — the public key, to be copied to the server's authorized_keys.

Let me know when you're ready to do that part — or if you want to generate the key for admin_user as well!

You said:

now I want to write a script that that can show system information like: cpu usage, memory, and disk space. we can use psutil library for this.

ChatGPT said:

Perfect! You can use the psutil library to create a simple yet powerful system health monitoring script.

Let's call the script: system_info.py

system_info.py - Show CPU, Memory, and Disk Info

python

Copy

Edit

system_info.py

```
import psutil
import platform
import shutil
import os

def get_cpu_info():
    print("==== CPU Info ===")
    print(f"Logical CPUs: {psutil.cpu_count(logical=True)}")
    print(f"Physical CPUs: {psutil.cpu_count(logical=False)}")
    print(f"CPU Usage per core: {psutil.cpu_percent(percpu=True)}")
    print(f"Total CPU Usage: {psutil.cpu_percent()}%")
    print()
```

```
def get_memory_info():
    print("==== Memory Info ===")
    virtual_mem = psutil.virtual_memory()
    print(f"Total: {virtual_mem.total / (1024 ** 3):.2f} GB")
```



```

print(f"Available: {virtual_mem.available / (1024 ** 3):.2f} GB")
print(f"Used: {virtual_mem.used / (1024 ** 3):.2f} GB")
print(f"Percentage: {virtual_mem.percent}%")
print()

```

```

def get_disk_info():

    print("== Disk Info ==")

    disk_usage = psutil.disk_usage('/')

    print(f"Total: {disk_usage.total / (1024 ** 3):.2f} GB")
    print(f"Used: {disk_usage.used / (1024 ** 3):.2f} GB")
    print(f"Free: {disk_usage.free / (1024 ** 3):.2f} GB")
    print(f"Percentage: {disk_usage.percent}%")
    print()

```

```

def main():

    print(f"System: {platform.system()} {platform.release()}")
    print(f"Machine: {platform.machine()}")
    print(f"Processor: {platform.processor()}")
    print("-" * 30)

```

```

get_cpu_info()
get_memory_info()
get_disk_info()

```

```

if __name__ == "__main__":
    main()

```

To run this script:

Make sure you're in your virtual environment (ssh_env) where psutil is installed:

```

bash
Copy
Edit
source ssh_env/bin/activate
python system_info.py

```

❖ Output example:

yaml

Copy

Edit

System: Linux 6.5.0-21-generic

Machine: x86_64

Processor: x86_64

=====

==== CPU Info ===

Logical CPUs: 4

Physical CPUs: 2

CPU Usage per core: [5.0, 3.0, 8.0, 2.0]

Total CPU Usage: 4.5%

==== Memory Info ===

Total: 7.79 GB

Available: 2.64 GB

Used: 4.58 GB

Percentage: 58.8%

==== Disk Info ===

Total: 100.00 GB

Used: 45.00 GB

Free: 55.00 GB

Percentage: 45.0%

This script is perfect for your normal_user to monitor system health without risking any sensitive operations.

Let me know if you want to log this info to a file, send it over SSH, or display it in a menu or web UI!

You said:

now lets write a script that backups important directories like /etc and /home periodically. This script should compress backup files and store them in a specific directory. Our script should accept run_interval and output_dir as config and we can change these settings without changing the code and after that run the system.

ChatGPT said:

That's a great idea — and a solid admin task that fits well with the admin_user role! We'll write a backup script that:

Periodically backs up directories (e.g., /etc, /home)

Compresses them into a .tar.gz archive

Stores them in a given output_dir

Reads run_interval and output_dir from a config file, so they're changeable without modifying the script

File Structure

bash

Copy

Edit

backup_script/

 └── config.json

 └── backup.py

config.json — editable settings file:

json

Copy

Edit

{

 "run_interval": 3600,

 "output_dir": "/home/admin_user/backups"

}

run_interval is in seconds (e.g., 3600 = 1 hour)

output_dir is where compressed backup files will be saved

backup.py — the actual script:

python

Copy

Edit

backup.py

```
import os
import time
import tarfile
import json
from datetime import datetime

# Directories to back up
BACKUP_TARGETS = ['/etc', '/home']

# Load config from file
def load_config():
    with open('config.json', 'r') as f:
        return json.load(f)

# Create a compressed archive of the given directories
def create_backup(output_dir):
    timestamp = datetime.now().strftime('%Y%m%d_%H%M%S')
    backup_name = f"backup_{timestamp}.tar.gz"
    backup_path = os.path.join(output_dir, backup_name)

    os.makedirs(output_dir, exist_ok=True)

    print(f"[+] Creating backup at {backup_path}")
    with tarfile.open(backup_path, "w:gz") as tar:
        for target in BACKUP_TARGETS:
            tar.add(target, arcname=os.path.basename(target))

    print(f"[✓] Backup created successfully.")

# Main backup loop
def main():
    config = load_config()
    run_interval = config.get("run_interval", 3600)
    output_dir = config.get("output_dir", "/tmp/backups")
```

```
print(f"[*] Backup script started. Run interval: {run_interval}s. Output dir: {output_dir}")
```

while True:

try:

```
    create_backup(output_dir)
```

except Exception as e:

```
    print(f"[!] Backup failed: {e}")
```

```
    time.sleep(run_interval)
```

if __name__ == "__main__":

```
    main()
```

To run the backup script:

bash

Copy

Edit

cd backup_script

python3 backup.py

To change backup frequency or directory:

Just edit config.json — no need to touch backup.py:

json

Copy

Edit

{

"run_interval": 1800,

"output_dir": "/mnt/secure_backups"

}

Extra tips:

Only allow admin_user to execute this script via RBAC.

You can set it as a systemd service or background process if needed.

Add logging if you want to track backup history.

Would you like help setting this up as a background daemon or running it on a schedule using cron or systemd?

You said:

where should I put the backup_script directory? in the ssh_env alongside rbac_system?

ChatGPT said:

Great question! Yes — you can organize it cleanly under ssh_env, keeping everything modular. Here's a clean and logical structure:

Recommended folder layout:

graphql

Copy

Edit

ssh_env/

```
|   └── bin/      # Your Python virtualenv bin (auto-created)
|   └── lib/      # Virtualenv libs
|   └── rbac_system/    # Your RBAC-based SSH client system
|       |   └── ssh_client.py
|       |   └── permissions.py
|       |   └── rbac_config.py
|       |   └── system_info.py
|   └── backup_script/    # Your new backup script module
|       |   └── backup.py
|       |   └── config.json
└── ssh_env/      # Virtual environment config
```

❖ Why put backup_script/ here?

Keeps all your tools inside one environment.

The backup script can reuse psutil, logging, etc., from the same venv.

Easy to link it into your RBAC system (e.g., admin_user can call it via run_command() with python3 backup_script/backup.py).

How to navigate & run:

bash

Copy



Edit

```
cd ssh_env/backup_script  
source ../bin/activate # activate virtualenv  
python3 backup.py
```

If you want admin_user to trigger it via SSH, they can run:

bash

Copy

Edit

```
python3 ~/ssh_env/backup_script/backup.py
```

Let me know if you'd like to:

Trigger this via your RBAC command system

Run it in the background or via systemd/cron

Add logging or email alerts after each backup

You said:

Now we want to have a safe file transformation using SFTP. admin user can upload and download files. we must write a code that admin user can upload and download files using this command:

<Download/Upload>_file client local_path remote_path.

ChatGPT said:

Awesome — now you're adding secure file transfer support using SFTP over SSH for the admin_user. Let's build this cleanly and safely using paramiko.

Command format:

bash

Copy

Edit

```
Upload_file client local_path remote_path
```

```
Download_file client remote_path local_path
```

This command will be passed to your RBAC-based run_command() function for execution — and only allowed for admin_user.

✉ Let's create a file: sftp_transfer.py

Inside your rbac_system/ folder.

python

Copy

Edit

sftp_transfer.py

```
import paramiko
```

```
import os
```

```
def connect_sftp(hostname, port, username, key_path):
```

```
    try:
```

```
        key = paramiko.RSAKey.from_private_key_file(key_path)
```

```
        transport = paramiko.Transport((hostname, port))
```

```
        transport.connect(username=username, pkey=key)
```

```
        sftp = paramiko.SFTPClient.from_transport(transport)
```

```
        return sftp
```

```
    except Exception as e:
```

```
        print(f"[!] SFTP Connection failed: {e}")
```

```
        return None
```

```
def upload_file(sftp, local_path, remote_path):
```

```
    try:
```

```
        sftp.put(local_path, remote_path)
```

```
        print(f"[✓] Uploaded {local_path} to {remote_path}")
```

```
    except Exception as e:
```

```
        print(f"[!] Upload failed: {e}")
```

```
def download_file(sftp, remote_path, local_path):
```

```
    try:
```

```
        sftp.get(remote_path, local_path)
```

```
        print(f"[✓] Downloaded {remote_path} to {local_path}")
```

```
    except Exception as e:
```

```
        print(f"[!] Download failed: {e}")
```

```
def handle_file_transfer(command_parts, hostname, port, username, key_path):  
    if len(command_parts) != 4:  
        print("[!] Invalid command format.")  
        return  
  
    action = command_parts[0].lower()  
    local_path = command_parts[2]  
    remote_path = command_parts[3]  
  
    sftp = connect_sftp(hostname, port, username, key_path)  
    if not sftp:  
        return  
  
    if action == "upload_file":  
        upload_file(sftp, local_path, remote_path)  
    elif action == "download_file":  
        download_file(sftp, remote_path, local_path)  
    else:  
        print("[!] Unknown action.")  
  
    sftp.close()
```

Usage Example (inside your main command system)

Inside ssh_client.py or ssh_command_runner.py, you can add this to your command handler:

python

Copy

Edit

```
from sftp_transfer import handle_file_transfer
```

```
# Inside run_command() or wherever you're routing commands  
if command.startswith("Upload_file") or command.startswith("Download_file"):  
    handle_file_transfer(  
        command.split(),
```

```
hostname="your.server.ip",
port=22,
username="admin_user",
key_path="/home/your_user/.ssh/id_rsa"
)
```

 Security tip:

Ensure normal_user cannot access this functionality in your RBAC logic.

You can add a log after each upload/download.

Set permissions on the remote upload directory for security.

 Example run (for admin_user):

bash

Copy

Edit

Upload_file client /home/admin_user/secrets.txt /home/admin_user/uploads/secrets.txt

Download_file client /home/admin_user/uploads/config.json /home/admin_user/local_config.json

Would you like to integrate this directly into your command dispatching logic or test it as a standalone first?

You said:

now we want to implement the logging system in server and client for logging the user commands like run commands or uploaded or downloaded files.

ChatGPT said:

Great! Adding a logging system is an essential step for auditability and security — especially in a setup with role-based access and remote commands.

We'll implement logging on both the client and server sides, so that:

All commands run by users (including file transfers) are logged.

Logs include timestamp, username, command executed, and success/failure.

Logs are stored in a clean .log file, separated per client/server.



📁 Recommended Logging Structure:

pgsql

Copy

Edit

ssh_env/

 └── rbac_system/

 | └── logger_config.py # Common logger setup

 | └── ssh_client.py

 | └── permissions.py

 | └── rbac_config.py

 | └── system_info.py

 | └── sftp_transfer.py

 └── client.log

 └── backup_script/

 | └── backup.py

 └── config.json

 └── server/

 └── server_handler.py # Your remote command server (if applicable)

 └── server.log

💡 Step 1: Create logger_config.py (shared logger setup)

python

Copy

Edit

logger_config.py

import logging

def setup_logger(name, log_file, level=logging.INFO):

logger = logging.getLogger(name)

logger.setLevel(level)

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

file_handler = logging.FileHandler(log_file)

```
file_handler.setFormatter(formatter)
```

```
if not logger.hasHandlers():
    logger.addHandler(file_handler)

return logger
```

⌚ Step 2: Use logger in the client — ssh_client.py, sftp_transfer.py, etc.

Example in ssh_client.py:

python

Copy

Edit

```
from logger_config import setup_logger
```

```
client_logger = setup_logger('ClientLogger', 'rbac_system/client.log')
```

```
def run_command(command, username):
    client_logger.info(f"{username} attempting to run: {command}")
    try:
        # Run the command (SSH or local logic)
        # ...
        client_logger.info(f"{username} successfully ran: {command}")
    except Exception as e:
        client_logger.error(f"{username} failed to run: {command} | Error: {e}")
```

Example in sftp_transfer.py:

python

Copy

Edit

```
from logger_config import setup_logger
```

```
sftp_logger = setup_logger('SFTPLogger', 'rbac_system/client.log')
```

```
def upload_file(sftp, local_path, remote_path):
    try:
        sftp.put(local_path, remote_path)
```



```
sftp_logger.info(f"Uploaded {local_path} to {remote_path}")

except Exception as e:
    sftp_logger.error(f"Upload failed from {local_path} to {remote_path} | Error: {e}")
```

Step 3: (Optional) Server-side logging — if you have server-side command handler

If you're running a command receiver on the server (e.g., a custom Python SSH daemon), use similar logging:

python

Copy

Edit

```
# server_handler.py

from logger_config import setup_logger
```

```
server_logger = setup_logger('ServerLogger', 'server/server.log')
```

```
def handle_received_command(username, command):
    server_logger.info(f"Received command from {username}: {command}")
    try:
        # Execute securely
        server_logger.info(f"Executed command: {command}")
    except Exception as e:
        server_logger.error(f"Error executing {command}: {e}")
```

Final Result — sample client.log

pgsql

Copy

Edit

```
2025-04-06 22:17:45,101 - ClientLogger - INFO - admin_user attempting to run: Upload_file client /tmp/config.txt
/home/admin_user/config.txt
```

```
2025-04-06 22:17:45,503 - SFTPLoader - INFO - Uploaded /tmp/config.txt to /home/admin_user/config.txt
```

```
2025-04-06 22:18:03,211 - ClientLogger - INFO - normal_user attempting to run: Get system memory
```

```
2025-04-06 22:18:03,315 - ClientLogger - INFO - normal_user successfully ran: Get system memory
```

Best practices:

Store logs in a secure location (restrict chmod).

Rotate logs using logrotate on Linux.



Optional, timestamp log files (e.g., client_20250406.log).

Would you like help adding log rotation or visualizing logs later (like in a dashboard or web UI)?

You said:

All right. Now lets develop the server side based on openssh-server that I previously installed. I think that we will need a server command handler for that.

ChatGPT said:

Memory updated

Perfect — now that you've installed openssh-server, let's build the server-side command handler. This will run on the SSH server and interpret commands sent by your SSH client (based on RBAC permissions).

 Goal:

Your server should:

Accept SSH connections (already set up via openssh-server).

Receive commands remotely from users.

Interpret and run them based on roles.

Log all activity.

Step-by-step plan:

1. Decide how to handle commands

There are 2 options:

Option	Description
<input checked="" type="checkbox"/>	Use standard SSH shell Commands are executed normally by the Linux shell. Use RBAC filtering on the client side.
<input checked="" type="checkbox"/>	Custom server handler (Python script) Write a custom command router that the SSH server always runs (via .bashrc, .bash_profile, or forced command in authorized_keys). This gives you more control.

For more secure and centralized control, let's build a Python-based command router that is executed when a user connects via SSH.

2. Server Command Handler: server_handler.py

Put this in a safe directory on the server (e.g., /home/admin_user/server/server_handler.py):

python

Copy

Edit

```
# server_handler.py
```

```
import os
```

```
import sys
```

```
import subprocess
```

```
from logger_config import setup_logger
```

```
logger = setup_logger('ServerLogger', '/home/admin_user/server/server.log')
```

```
# Get username from SSH session
```

```
username = os.getenv("USER", "unknown_user")
```

```
def run_safe_command(command):
```

```
    try:
```

```
        output = subprocess.check_output(command, shell=True, stderr=subprocess.STDOUT, timeout=10)
```

```
        return output.decode()
```

```
    except subprocess.CalledProcessError as e:
```

```
        logger.error(f"Command failed: {e.output.decode()}")
```

```
        return f"[!] Command error: {e.output.decode()}"
```

```
    except Exception as e:
```

```
        logger.error(f"Exception: {e}")
```

```
        return f"[!] Exception: {str(e)}"
```

```
def main():
```

```
    input_command = ''.join(sys.argv[1:])
```

```
    logger.info(f"User '{username}' ran: {input_command}")
```

```
# Example RBAC enforcement (expand this)
```

```
if username == "normal_user":
```

```
    if any(word in input_command for word in ["rm", "shutdown", "apt", "backup", "upload", "download"]):
```



```
logger.warning(f"Unauthorized command by {username}")
print("![] Permission denied.")
return

result = run_safe_command(input_command)
print(result)
```

if __name__ == "__main__":

main()

3. Force this script to run for SSH sessions

Edit the `~/.ssh/authorized_keys` file for each user and prefix their public key with a forced command:

bash

Copy

Edit

command="/usr/bin/python3 /home/admin_user/server/server_handler.py" ssh-rsa AAAA...yourkey...== user@client

This ensures that any SSH connection executes `server_handler.py` instead of giving full shell access.

For `admin_user`, you can optionally allow full shell or have more flexible command parsing.

4. Test the system

From the client:

bash

Copy

Edit

`ssh normal_user@server_ip "df -h"`

Expected:

Command runs if permitted.

Logs are created at `/home/admin_user/server/server.log`.

Denied commands show `![] Permission denied.`

Would you like to:

Add RBAC config loading from a JSON file for flexibility?

Support more user-specific command groups?

Add SCP or SFTP integration into this server handler?

Let me know and we can upgrade it!

You said:

I was a little confused about the server that you said in the previous response. For now, let me send you all the codes that we have so far in several messages. please check the codes and tell me to modify if needed. then, based on the codes I sent, develop the server and after that tell me how to run the server, how to run the ssh_client (that can be normal_user or admin_user), and test all the functionalities generated. So I am going to send you more in the next message.

ChatGPT said:

Awesome — that sounds like a solid plan. Please go ahead and send all the code files you've written so far (like ssh_client.py, rbac_config.py, permissions.py, system_info.py, sftp_transfer.py, backup.py, logger_config.py, etc.).

Once I receive them:

I'll review everything and suggest fixes or improvements if needed.

Then we'll build the server-side setup that works with your SSH environment.

I'll show you how to run both the server and the client, and

How to test RBAC, command execution, backup, and file transfer.

Go ahead and start pasting the files — I'll wait and keep track!

You said:

so we have a ssh_env directory that has a backup_script directory that has 2 files. config.json:

{

 "run_interval": 3600,

```
"output_dir": "/home/admin_user/backups"  
}
```

and backup.py:

```
import os  
  
import time  
  
import tarfile  
  
import json  
  
from datetime import datetime  
  
  
BACKUP_TARGETS = ['/etc', '/home']  
  
  
def load_config():  
    with open('config.json', 'r') as f:  
        return json.load(f)  
  
  
    # Create a compressed archive of the given directories  
  
def create_backup(output_dir):  
    timestamp = datetime.now().strftime('%Y%m%d_%H%M%S')  
  
    backup_name = f"backup_{timestamp}.tar.gz"  
  
    backup_path = os.path.join(output_dir, backup_name)  
  
  
    os.makedirs(output_dir, exist_ok=True)  
  
  
    print(f"[+] Creating backup at {backup_path}")  
    with tarfile.open(backup_path, "w:gz") as tar:  
        for target in BACKUP_TARGETS:  
            tar.add(target, arcname=os.path.basename(target))  
  
  
    print(f"[✓] Backup created successfully.")  
  
  
def main():  
    config = load_config()  
  
    run_interval = config.get("run_interval", 3600)
```



```
output_dir = config.get("output_dir", "/tmp/backups")

print(f"[*] Backup script started. Run interval: {run_interval}s. Output dir: {output_dir}")
```

while True:

```
    try:
        create_backup(output_dir)
    except Exception as e:
        print(f"[!] Backup failed: {e}")
        time.sleep(run_interval)
```

if __name__ == "__main__":

```
    main()
```

I will send you the rest, in the next messages.

ChatGPT said:

Perfect — this is a great start!

First review: backup_script/config.json and backup.py

Everything looks solid! Here's a quick summary and a couple of suggestions:

config.json — Good as-is

Keeps run_interval and output_dir configurable.

Follows best practices for decoupling config from code.

backup.py — Feedback and Suggestions:

Strengths:

Config-driven

Compresses /etc and /home

Uses tarfile correctly

Infinite loop with interval



Suggested Improvements:

Absolute path for config.json

If this is ever run from another directory, it might break.

Suggestion:

python

Copy

Edit

```
def load_config():
```

```
    base_dir = os.path.dirname(os.path.abspath(__file__))
    config_path = os.path.join(base_dir, 'config.json')
    with open(config_path, 'r') as f:
        return json.load(f)
```

Add logging support instead of just print()

Since you already have a logging system in the main project, consider integrating logger_config.

Optional: Add a one-time backup mode with a flag for manual execution.

Let me know if you want me to modify this file now, or after I receive the rest of your project files.

Go ahead and send the next files when you're ready!

You said:

Please do the number 2 suggestion also on the backup.py file and send me the code. I will send you the rest, in the next messages.

ChatGPT said:

Absolutely! Here's the improved version of your backup.py file, now with integrated logging using your existing logger_config setup:

backup.py (logging integrated)

python



Copy

Edit

```
import os
import time
import tarfile
import json
from datetime import datetime
from logger_config import setup_logger

# Setup logger
logger = setup_logger('BackupLogger', '/home/admin_user/ssh_env/logs/backup.log')

BACKUP_TARGETS = ['/etc', '/home']

def load_config():
    base_dir = os.path.dirname(os.path.abspath(__file__))
    config_path = os.path.join(base_dir, 'config.json')
    with open(config_path, 'r') as f:
        return json.load(f)

# Create a compressed archive of the given directories
def create_backup(output_dir):
    timestamp = datetime.now().strftime('%Y%m%d_%H%M%S')
    backup_name = f"backup_{timestamp}.tar.gz"
    backup_path = os.path.join(output_dir, backup_name)

    os.makedirs(output_dir, exist_ok=True)

    logger.info(f"Creating backup at {backup_path}")
    with tarfile.open(backup_path, "w:gz") as tar:
        for target in BACKUP_TARGETS:
            tar.add(target, arcname=os.path.basename(target))

    logger.info("Backup created successfully.")
```



```
def main():

    config = load_config()
    run_interval = config.get("run_interval", 3600)
    output_dir = config.get("output_dir", "/tmp/backups")

    logger.info(f"Backup script started. Run interval: {run_interval}s. Output dir: {output_dir}")

    while True:
        try:
            create_backup(output_dir)
        except Exception as e:
            logger.error(f"Backup failed: {e}")
            time.sleep(run_interval)

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

📝 Notes:

The logger will now write to: /home/admin_user/ssh_env/logs/backup.log
(make sure the logs directory exists — or change the path as needed).

You'll still get clean, timestamped logs for backup actions.

Send the next parts of your project whenever you're ready — I'm keeping track!

You said:

I decided not to do this now. ssh_env has another directory named rbac_system. containing:

logger_config.py:

```
import logging
```

```
def setup_logger(name, log_file, level=logging.INFO):
    logger = logging.getLogger(name)
    logger.setLevel(level)
```



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

```
file_handler = logging.FileHandler(log_file)
```

```
file_handler.setFormatter(formatter)
```

```
if not logger.hasHandlers():
```

```
    logger.addHandler(file_handler)
```

```
return logger
```

rbac_config.py:

```
RBAC = {
```

```
    "admin_user": {
```

```
        "role": "admin",
```

```
        "allowed_commands": ["*"] # admin can run everything
```

```
    },
```

```
    "normal_user": {
```

```
        "role": "user",
```

```
        "allowed_commands": [
```

```
            "uptime",
```

```
            "df -h",
```

```
            "free -m",
```

```
            "ps aux",
```

```
            "top -b -n1",
```

```
            "whoami",
```

```
            "hostname",
```

```
        ]
```

```
    }
```

```
}
```

I will send you the rest, in the next messages.

ChatGPT said:

Perfect, thanks for sharing the contents of the rbac_system directory so far. Here's a quick review:



logger_config.py

Solid, no changes needed.

Uses FileHandler with good formatting.

Avoids duplicate handlers with hasHandlers() check.

Approved as-is.

rbac_config.py

Defines clear roles.

"*" wildcard for admin — good for command bypassing.

Only safe commands allowed for normal_user — perfect for system monitoring.

Approved as-is.

If you want, we can later extend this to support role-specific script access (e.g., backup, upload, etc.) too.

Go ahead and send the next batch of files from rbac_system or any other directories — I'm keeping track and reviewing as we go!

You said:

system_info.py:

```
import psutil
import platform
import shutil
import os
```

```
def get_cpu_info():
    print("== CPU Info ==")
    print(f"Logical CPUs: {psutil.cpu_count(logical=True)}")
    print(f"Physical CPUs: {psutil.cpu_count(logical=False)}")
    print(f"CPU Usage per core: {psutil.cpu_percent(percpu=True)}")
```

```
print(f"Total CPU Usage: {psutil.cpu_percent()}%")
print()
```

```
def get_memory_info():
    print("==== Memory Info ====")
    virtual_mem = psutil.virtual_memory()
    print(f"Total: {virtual_mem.total / (1024 ** 3):.2f} GB")
    print(f"Available: {virtual_mem.available / (1024 ** 3):.2f} GB")
    print(f"Used: {virtual_mem.used / (1024 ** 3):.2f} GB")
    print(f"Percentage: {virtual_mem.percent}%")
    print()
```

```
def get_disk_info():
    print("==== Disk Info ====")
    disk_usage = psutil.disk_usage('/')
    print(f"Total: {disk_usage.total / (1024 ** 3):.2f} GB")
    print(f"Used: {disk_usage.used / (1024 ** 3):.2f} GB")
    print(f"Free: {disk_usage.free / (1024 ** 3):.2f} GB")
    print(f"Percentage: {disk_usage.percent}%")
    print()
```

```
def main():
    print(f"System: {platform.system()} {platform.release()}")
    print(f"Machine: {platform.machine()}")
    print(f"Processor: {platform.processor()}")
    print("-" * 30)
```

```
get_cpu_info()
get_memory_info()
get_disk_info()
```

```
if __name__ == "__main__":
    main()
permissions.py:
```



```
import re

from rbac_config import RBAC


def is_command_allowed(user, command):

    if user not in RBAC:

        return False

    allowed = RBAC[user]["allowed_commands"]

    if "*" in allowed:

        return True

    for allowed_cmd in allowed:

        if command.strip().startswith(allowed_cmd):

            return True

    return False
```

I will send you the rest, in the next messages.

ChatGPT said:

Awesome! You're organizing your project very well 🎉 — here's the detailed review of these two files:

system_info.py

Strengths:

Uses psutil and platform modules well to gather system info.

Memory, CPU, and disk reports are cleanly formatted and easy to read.

main() function structure makes this file callable directly or as a utility.

💡 Optional Suggestions:

If you'd like to return these values instead of printing them (for logging or remote output), we can adapt this into a function that returns a string.

Want me to help rewrite it for remote output later?

permissions.py

Strengths:

Simple and clean permission checker.

Uses startswith to allow partial match of commands (great for top, df -h, etc.).

Proper wildcard handling for admins.

 Suggested Small Fix:

Your indenting in the loop is using tabs, while the rest is using spaces. This might cause issues in Python 3 (mixed tabs/spaces). I'd recommend:

python

Copy

Edit

for allowed_cmd in allowed:

```
if command.strip().startswith(allowed_cmd):
    return True
```

Both scripts are functionally sound and align with your RBAC logic and system info goals.

Ready for the next batch when you are!

You said:

sftp_transfer.py:

```
import paramiko
```

```
import os
```

```
from logger_config import setup_logger
```

```
def connect_sftp(hostname, port, username, key_path):
```

```
    try:
```

```
        key = paramiko.RSAKey.from_private_key_file(key_path)
```

```
        transport = paramiko.Transport((hostname, port))
```

```
        transport.connect(username=username, pkey=key)
```

```
        sftp = paramiko.SFTPClient.from_transport(transport)
```

```
        return sftp
```

```
    except Exception as e:
```

```
        print(f"[!] SFTP Connection failed: {e}")
```

```
return None

sftp_logger = setup_logger('SFTPLLogger', 'rbac_system/client.log')

def upload_file(sftp, local_path, remote_path):
    try:
        sftp.put(local_path, remote_path)
        sftp_logger.info(f"Uploaded {local_path} to {remote_path}")
    except Exception as e:
        sftp_logger.error(f"Upload failed from {local_path} to {remote_path} | Error: {e}")

def download_file(sftp, remote_path, local_path):
    try:
        sftp.get(remote_path, local_path)
        print(f"[√] Downloaded {remote_path} to {local_path}")
    except Exception as e:
        print(f"[!] Download failed: {e}")

def handle_file_transfer(command_parts, hostname, port, username, key_path):
    if len(command_parts) != 4:
        print("[!] Invalid command format.")
        return

    action = command_parts[0].lower()
    local_path = command_parts[2]
    remote_path = command_parts[3]

    sftp = connect_sftp(hostname, port, username, key_path)
    if not sftp:
        return

    if action == "upload_file":
        upload_file(sftp, local_path, remote_path)
    elif action == "download_file":
```

```
download_file(sftp, remote_path, local_path)

else:
    print("[!] Unknown action.")
```

```
sftp.close()

test_rbac.py:
```

```
from permissions import is_command_allowed
```

```
user = "normal_user"

command = "df -h"

if is_command_allowed(user, command):
    print(f"{user} is allowed to run: {command}")
else:
    print(f"{user} is NOT allowed to run: {command}")
```

I will send the rest, in the next messages.

ChatGPT said:

You're doing fantastic — this project is shaping up to be very professional 🔒. Here's the detailed review of both files:

sftp_transfer.py

Strengths:

Uses paramiko and RSA key auth correctly.

Structured cleanly into small functions:

connect_sftp(), upload_file(), download_file(), handle_file_transfer()

Logging actions (uploads/downloads) to a centralized log — great practice.

Clean command parsing (upload_file client local remote etc.)

* Suggestions (Optional Improvements):

Improve error messages: You already print/log well. If you want to go further, you can catch specific exceptions (IOError, FileNotFoundError, paramiko.SSHException) for more precise logging.

- Enhance usability: Consider printing a friendly message upon success for upload_file() just like you did for downloads.

python

Copy

Edit

```
print(f"[✓] Uploaded {local_path} to {remote_path}")
```

test_rbac.py

Strengths:

Clean and simple test.

Good for quick permission checks.

🔧 Suggestions:

None! It's great as a minimal test file. We can later expand this into a proper unit test suite if desired (with unittest or pytest).

Both files are ready and solid.

Waiting for the rest of the files so I can do a full pass and then build the final server + integration instructions with you!

You said:

ssh_client.py:

```
import paramiko
```

```
from permissions import is_command_allowed
```

```
from sftp_transfer import handle_file_transfer
```

```
from logger_config import setup_logger
```

```
def connect_ssh(hostname, port, username, key_path):
```

```
    try:
```

```
        key = paramiko.RSAKey.from_private_key_file(key_path)
```

```
        client = paramiko.SSHClient()
```

```
        client.set_missing_host_key_policy(paramiko.AutoAddPolicy()) # Accept unknown host keys
```

```
        client.connect(hostname=hostname, port=port, username=username, pkey=key)
```

```
print(f"[+] Connected to {hostname} as {username}")

return client

except paramiko.AuthenticationException:
    print("[-] Authentication failed. Check the SSH key or username.")

except paramiko.SSHException as e:
    print(f"[-] SSH error: {e}")

except Exception as e:
    print(f"[-] Unknown error: {e}")

return None

client_logger = setup_logger('ClientLogger', 'rbac_system/client.log')

def run_command(client, user, command):
    client_logger.info(f"{username} attempting to run: {command}")
    if not is_command_allowed(user, command):
        print(f"[-] User '{user}' is not allowed to run the command: {command}")
        return

    if command.startswith("Upload_file") or command.startswith("Download_file"):
        handle_file_transfer(
            command.split(),
            hostname="your.server.ip",
            port=22,
            username="admin_user",
            key_path="/home/admin_user/.ssh/id_rsa"
        )

    try:
        stdin, stdout, stderr = client.exec_command(command)
        output = stdout.read().decode()
        error = stderr.read().decode()
```

```
if output:  
    print(f"[Output]\n{output}")  
  
if error:  
    print(f"[Error]\n{error}")
```

```
client_logger.error(f"\'{username}\' failed to run: {command} | Error: {e}")
```

test_client.py:

```
from ssh_client import connect_ssh, run_command
```

```
HOST = "your.server.ip.or.hostname"
```

PORT = 22

```
USERNAME = "admin_user" # or "normal_user"
```

```
KEY_PATH = "/home/admin_user/.ssh/id_rsa" # Path to private key
```

```
COMMAND = "df -h"
```

```
client = connect_ssh(HOST, PORT, USERNAME, KEY_PATH)
```

if client:

run_command(client, USERNAME, COMMAND)

`client.close()`

This was it.

ChatGPT said:

Memory updated

Perfect, thanks for sending all the code and the structure—it's well organized and shows you're really thinking this through thoroughly.

Here's what I'll do:

- ## First: Quick Review & Suggested Fixes

1. ssh_client.py Bug

In the run_command function, you're referencing username but it's not defined in that scope. You should replace:



python
Copy
Edit
client_logger.info(f"{username} attempting to run: {command}")
with:

python
Copy
Edit
client_logger.info(f"{user} attempting to run: {command}")
2. Hardcoded IP and Username
In run_command (file transfer block), you're using:

python
Copy
Edit
hostname="your.server.ip",
username="admin_user",
key_path="/home/admin_user/.ssh/id_rsa"
 Suggestion: Pass those as parameters, or refactor your code to avoid hardcoding.

Now: Let's Build the Server Command Handler

Since you're using openssh-server, your server is already set up to receive SSH connections. But now you need a handler on the server side to:

Execute allowed commands

Handle file storage for uploads/downloads

Maintain logs

You can do this by writing a server_handler.py script that lives in the rbac_system/ directory and processes commands and logs locally when users connect.

server_handler.py (Server Command Handler)



python
Copy
Edit

```
import os
import subprocess
from logger_config import setup_logger

server_logger = setup_logger('ServerLogger', 'rbac_system/server.log')

def handle_command(command, user):
    try:
        server_logger.info(f"User '{user}' executed: {command}")

        # Run the command and capture output
        result = subprocess.run(command, shell=True, capture_output=True, text=True)
        output = result.stdout
        error = result.stderr

        if output:
            print(f"[Output]\n{output}")
        if error:
            print(f"[Error]\n{error}")

        return output, error
    except Exception as e:
        server_logger.error(f"Command execution error from user '{user}': {e}")
        return None, str(e)
```

You can expand this later with tighter access control, command logging, and even allow it to be run as a service for remote triggers.

How to Run and Test Everything

1. Start the Server

Make sure openssh-server is running:



bash

Copy

Edit

sudo systemctl start ssh

sudo systemctl enable ssh

2. Set Up SSH Keys

On client machine (e.g., your VM or remote device):

bash

Copy

Edit

ssh-keygen -t rsa -b 4096

ssh-copy-id -i ~/.ssh/id_rsa.pub admin_user@your.server.ip

Same for normal_user.

3. Test from Client

Edit test_client.py:

python

Copy

Edit

```
HOST = "192.168.x.x" # Replace with your server IP
```

```
POR
```

```
T = 22
```

```
USERNAME = "normal_user" # Or "admin_user"
```

```
KEY_PATH = "/home/normal_user/.ssh/id_rsa"
```

```
COMMAND = "df -h"
```

Run:

bash

Copy

Edit

python3 test_client.py

You should see command output and logs written in rbac_system/client.log and rbac_system/server.log.

Final Directory Tree (Overview)

pgsql

Copy

Edit

ssh_env/

 └── backup_script/

 | └── backup.py

 | └── config.json

 └── rbac_system/

 | └── logger_config.py

 | └── permissions.py

 | └── rbac_config.py

 | └── ssh_client.py

 | └── sftp_transfer.py

 | └── system_info.py

 | └── test_client.py

 | └── test_rbac.py

 | └── server_handler.py 

Would you like me to help you wrap this into a Makefile or shell script to automate testing and execution?

You said:

I have a ubuntu VM as said before. I want to use it as the server and the client. and test normal_user and admin_user with it, tell me how to achieve this.

ChatGPT said:

Perfect! Using your Ubuntu VM as both the SSH server and client is a great way to test everything locally. Here's a complete step-by-step guide to simulate and test both admin_user and normal_user on the same machine.

Step-by-Step Guide to Use Ubuntu VM as Server & Client

 1. Create Users: admin_user and normal_user

Open a terminal and run:

bash

Copy

Edit

```
# Create admin_user  
sudo adduser admin_user
```

```
# Create normal_user
```

```
sudo adduser normal_user
```

Follow prompts to set passwords (or press Enter for none).

2. Enable SSH Server

Make sure OpenSSH is installed and running:

```
bash  
Copy  
Edit  
sudo apt update  
sudo apt install openssh-server  
sudo systemctl enable ssh  
sudo systemctl start ssh
```

Check it's working:

```
bash  
Copy  
Edit  
sudo systemctl status ssh
```

3. Set Up SSH Key Authentication

We'll generate SSH keys for both users to log in via key-based authentication.

For admin_user:

```
bash  
Copy  
Edit  
sudo su - admin_user  
ssh-keygen -t rsa -b 4096 -f ~/.ssh/id_rsa  
exit
```

For normal_user:



bash
Copy
Edit
sudo su - normal_user
ssh-keygen -t rsa -b 4096 -f ~/.ssh/id_rsa
exit

Copy public keys to the authorized keys:

bash
Copy
Edit
For admin_user
sudo su - admin_user
mkdir -p ~/.ssh
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
chmod 600 ~/.ssh/authorized_keys
chmod 700 ~/.ssh
exit

For normal_user
sudo su - normal_user
mkdir -p ~/.ssh
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
chmod 600 ~/.ssh/authorized_keys
chmod 700 ~/.ssh
exit

✍ 4. Test SSH Locally

Now, test SSH locally using the loopback address (127.0.0.1):

bash
Copy
Edit
ssh -i /home/admin_user/.ssh/id_rsa admin_user@127.0.0.1
ssh -i /home/normal_user/.ssh/id_rsa normal_user@127.0.0.1
You should log in without being asked for a password.



📁 5. Put ssh_env in Both Home Directories

Let's keep it simple. Put the same ssh_env project in both users' home directories:

bash

Copy

Edit

```
sudo cp -r /path/to/ssh_env /home/admin_user/
```

```
sudo cp -r /path/to/ssh_env /home/normal_user/
```

```
sudo chown -R admin_user:admin_user /home/admin_user/ssh_env
```

```
sudo chown -R normal_user:normal_user /home/normal_user/ssh_env
```

✍ 6. Test with Each User

Open two terminals and log in to each user:

👤 As admin_user:

bash

Copy

Edit

```
sudo su - admin_user
```

```
cd ssh_env/rbac_system
```

```
python3 test_client.py
```

Edit test_client.py for admin_user:

python

Copy

Edit

```
HOST = "127.0.0.1"
```

```
PORt = 22
```

```
USERNAME = "admin_user"
```

```
KEY_PATH = "/home/admin_user/.ssh/id_rsa"
```

```
COMMAND = "uptime"
```

👤 As normal_user:

bash



Copy

Edit

```
sudo su - normal_user
```

```
cd ssh_env/rbac_system
```

```
python3 test_client.py
```

Edit test_client.py for normal_user:

```
python
```

Copy

Edit

```
HOST = "127.0.0.1"
```

```
PORT = 22
```

```
USERNAME = "normal_user"
```

```
KEY_PATH = "/home/normal_user/.ssh/id_rsa"
```

```
COMMAND = "ps aux"
```

You can test unauthorized commands too (e.g., ls /root) and it should reject for normal_user.

⌚ 7. Test File Upload/Download via SFTP

Update COMMAND to:

```
python
```

Copy

Edit

```
COMMAND = "Upload_file /home/normal_user/test.txt /home/admin_user/uploaded.txt"
```

Make sure the source file exists before running.

📁 8. Check Logs

Each user will log to:

```
bash
```

Copy

Edit

```
/home/admin_user/ssh_env/rbac_system/client.log
```

```
/home/admin_user/ssh_env/rbac_system/server.log
```

or similarly under `normal_user`.

Recap

You now have:

 SSH server (`openssh-server`) running locally

 Two users: `admin_user` (full access) and `normal_user` (limited access)

 Key-based SSH auth between them and the same machine

 Functional backup script, RBAC control, SFTP transfer

 Logging on client & server side

Would you like me to create an automation shell script to run the tests for both users or help wrap this in a minimal TUI or CLI menu?

You said:

When I want to test `admin_user`, this was the output:

```
(ssh_env) admin_user@kasra-None:~/ssh_env/rbac_system$ python3 test_client.py
```

Traceback (most recent call last):

```
File "/home/admin_user/ssh_env/rbac_system/test_client.py", line 1, in <module>
  from ssh_client import connect_ssh, run_command
File "/home/admin_user/ssh_env/rbac_system/ssh_client.py", line 1, in <module>
  import paramiko
```

`ModuleNotFoundError: No module named 'paramiko'`

ChatGPT said:

This error means that the Python module paramiko (used for SSH/SFTP) is not installed in your current Python environment — in this case, you're using a virtual environment named ssh_env.

No worries — here's how to fix it! ↗

Step 1: Activate Your Virtual Environment (if not already)

You seem to already be in it, based on your prompt:

bash

Copy

Edit

(ssh_env) admin_user@kasra-None:~\$

If not, activate it like this:

bash

Copy

Edit

source ~/ssh_env/bin/activate

Step 2: Install paramiko in the Environment

With the virtual environment active, install the package:

bash

Copy

Edit

pip install paramiko

If you're missing psutil or other modules like platform or shutil, install them too:

bash

Copy

Edit

pip install psutil

Step 3: Try Again

Now try running your test script again:

bash



Copy

Edit

```
cd ~/ssh_env/rbac_system
```

```
python3 test_client.py
```

You should see something like:

css

Copy

Edit

```
[+] Connected to 127.0.0.1 as admin_user
```

[Output]

```
<...command output here...>
```

✖ Bonus: Freeze Requirements

Once everything works, it's a good idea to save your dependencies:

bash

Copy

Edit

```
pip freeze > requirements.txt
```

So later, you or others can install everything again via:

bash

Copy

Edit

```
pip install -r requirements.txt
```

Let me know if you hit any other errors or if you want to test normal_user next!

You said:

I tried to install, but this was the error message:

```
(ssh_env) admin_user@kasra-None:~/ssh_env/rbac_system$ pip install paramiko
```

```
error: externally-managed-environment
```

✖ This environment is externally managed

↳ To install Python packages system-wide, try apt install

python3-xyz, where xyz is the package you are trying to install.

If you wish to install a non-Debian-packaged Python package, create a virtual environment using python3 -m venv path/to/venv. Then use path/to/venv/bin/python and path/to/venv/bin/pip. Make sure you have python3-full installed.

If you wish to install a non-Debian packaged Python application, it may be easiest to use pipx install xyz, which will manage a virtual environment for you. Make sure you have pipx installed.

See /usr/share/doc/python3.12/README.venv for more information.

note: If you believe this is a mistake, please contact your Python installation or OS distribution provider. You can override this, at the risk of breaking your Python installation or OS, by passing --break-system-packages.

hint: See PEP 668 for the detailed specification.

ChatGPT said:

Thanks for sharing the error — it confirms that you're using a Debian/Ubuntu system with Python PEP 668 protection enabled. This prevents pip from installing packages into system Python to avoid breaking your OS.

But don't worry! Since you're already working inside a virtual environment (ssh_env), this should allow pip to work normally. However, it seems like your venv might have been misconfigured or not isolated properly.

Fix Option 1: Recreate Your Virtual Environment Properly

Let's first delete and recreate the virtual environment properly so you can install packages without the error.

Step 1: Remove Old venv

bash

Copy

Edit

```
rm -rf ~/ssh_env
```

Step 2: Recreate the venv

bash

Copy



Edit

```
python3 -m venv ~/ssh_env
```

Make sure python3-full is installed:

bash

Copy

Edit

```
sudo apt install python3-full
```

Step 3: Activate it

bash

Copy

Edit

```
source ~/ssh_env/bin/activate
```

Your prompt should now show (ssh_env) again.

Step 4: Install Paramiko & Psutil

Now, install the packages:

bash

Copy

Edit

```
pip install paramiko psutil
```

Everything should work now

Test It

Try again:

bash

Copy

Edit

```
cd ~/ssh_env/rbac_system
```

```
python3 test_client.py
```

Alternate Fix (If Needed): Use --break-system-packages

As a last resort (not recommended in general), you can force pip to install by bypassing PEP 668 with:

bash

Copy

Edit

```
pip install paramiko --break-system-packages
```

But only do this if you're not using a virtual environment or need to test quickly.

Let me know if you'd like me to walk you through testing both admin_user and normal_user afterward!

You said:

I was doing the same for the normal_user but this appears:

```
normal_user@kasra-None:~$ sudo apt install python3-full
```

[sudo] password for normal_user:

normal_user is not in the sudoers file.

This incident has been reported to the administrator.

ChatGPT said:

Ah yes — that makes total sense, and it's actually a great sign that your RBAC system is working properly. Here's what's happening:

⌚ Why This Happened

You're logged in as normal_user, and you're trying to run:

bash

Copy

Edit

```
sudo apt install python3-full
```

But normal_user doesn't have sudo privileges, which is expected behavior — they're not an admin, so they can't install system-wide packages.

☑ What You Should Do Instead

Since normal_user cannot use sudo, you have a couple of safer options:

☒ Option 1: Let admin_user Set Up Everything

Switch back to admin_user:

bash

Copy

Edit

su - admin_user

Activate the venv:

bash

Copy

Edit

source ~/ssh_env/bin/activate

Install everything needed once:

bash

Copy

Edit

pip install paramiko psutil

This way, the whole environment is pre-configured.

📝 Option 2: Share the Virtual Environment

After admin_user creates and installs everything inside ~/ssh_env, you can allow normal_user to use it too:

Change permissions so normal_user can access the env:

From admin_user:

bash

Copy

Edit

chmod -R o+rx ~/ssh_env

From normal_user:

bash

 Copy

 Edit

```
source /home/admin_user/ssh_env/bin/activate
```

Test the client:

 bash

 Copy

 Edit

```
cd /home/admin_user/ssh_env/rbac_system
```

```
python3 test_client.py
```

Just make sure test_client.py is configured to use "normal_user" for testing.

Summary

Action Who Should Do It

Installing Python packages admin_user only

Testing allowed commands normal_user

Using shared virtualenv Both (with permissions set)

Let me know when you're ready, and I can help you test the full RBAC system — including admin_user doing backups, file transfers, and normal_user trying limited commands.

You said:

I want to temporary put the normal_user to sudo group, install and after that get it out from sudo group.

ChatGPT said:

Perfect! Temporarily granting normal_user sudo access is totally doable, as long as you remove them from the sudo group afterward. Here's a clean step-by-step:

 As admin_user (or any sudo-capable account)

1. Add normal_user to sudo group:

 bash



Copy

Edit

```
sudo usermod -aG sudo normal_user
```

-aG means “append to group”— don’t remove them from any current groups.

✉ 2. Apply changes (log out and back in)

Ask `normal_user` to log out and log back in, or you can force it via:

bash

Copy

Edit

```
su - normal_user
```

They should now have sudo access. You can check with:

bash

Copy

Edit

groups

You should see:

yaml

Copy

Edit

```
normal_user : normal_user sudo
```

✉ 3. Install the needed packages as `normal_user`

bash

Copy

Edit

```
sudo apt update
```

```
sudo apt install python3-full python3-venv
```

Then create and activate a new virtual environment if needed:

bash

Copy



Edit

```
python3 -m venv ~/ssh_env  
source ~/ssh_env/bin/activate  
pip install paramiko psutil
```

🔒 4. Remove normal_user from sudo group

After you're done, switch back to admin_user and run:

bash

Copy

Edit

```
sudo deluser normal_user sudo
```

This revokes sudo access.

Again, have normal_user log out and log back in for the removal to fully take effect.

Let me know when you've done this, and I'll help you move on to testing everything smoothly as both user roles.

You said:

now, how can I copy the codes to the ssh_env of each admin_user and normal_user

ChatGPT said:

Great question! Since both admin_user and normal_user are on the same Ubuntu VM, there are a few easy ways to share or copy the ssh_env project directory between them.

Option 1: Shared directory with group permissions (Recommended)

You can place ssh_env in a shared directory like /opt/ssh_env, and make both users able to read/write it.

💻 1. Move the project to /opt:

bash

Copy



Edit

```
sudo mv /home/admin_user/ssh_env /opt/ssh_env
```

2. Create a shared group (optional but clean):

bash

Copy

Edit

```
sudo groupadd sshgroup
```

```
sudo chown -R admin_user:sshgroup /opt/ssh_env
```

```
sudo usermod -aG sshgroup admin_user
```

```
sudo usermod -aG sshgroup normal_user
```

3. Set permissions so both users can access it:

bash

Copy

Edit

```
sudo chmod -R 770 /opt/ssh_env
```

```
sudo chmod -R g+s /opt/ssh_env # so new files get the group
```

Now both users can do:

bash

Copy

Edit

```
cd /opt/ssh_env
```

You can also symlink it to their home directories:

bash

Copy

Edit

```
ln -s /opt/ssh_env ~/ssh_env
```

Option 2: Copy the folder manually to each user's home

If you prefer separate copies for each user (e.g., for isolated testing):

As admin_user, run:

bash

Copy

Edit

```
cp -r ~/ssh_env /home/normal_user/  
sudo chown -R normal_user:normal_user /home/normal_user/ssh_env
```

Then normal_user can:

bash

Copy

Edit

```
cd ~/ssh_env
```

Option 3: Tar and copy

If you want to zip first:

bash

Copy

Edit

```
tar czf ssh_env.tar.gz ssh_env/  
sudo cp ssh_env.tar.gz /home/normal_user/  
sudo chown normal_user:normal_user /home/normal_user/ssh_env.tar.gz
```

Then as normal_user:

bash

Copy

Edit

cd ~

```
tar xzf ssh_env.tar.gz
```

Let me know which method you prefer, and I'll guide you through the next step — running and testing!

You said:

both my admin_user and normal_user already have a ssh_env directory, I think that we should copy the backup_scripts and rbac_system directories to their ssh_envs right?

ChatGPT said:

Exactly! Since both admin_user and normal_user already have their own ssh_env directories, the cleanest way is to just copy the backup_script/ and rbac_system/ directories into each user's ssh_env.

Step-by-step (as admin_user):

Let's say you're currently inside:

swift

Copy

Edit

/home/admin_user/ssh_env/

⌚ 1. Copy backup_script/ to normal_user:

bash

Copy

Edit

cp -r /home/admin_user/ssh_env/backup_script /home/normal_user/ssh_env/

⌚ 2. Copy rbac_system/ to normal_user:

bash

Copy

Edit

cp -r /home/admin_user/ssh_env/rbac_system /home/normal_user/ssh_env/

⌚ 3. Fix ownership so normal_user can access the files:

bash

Copy

Edit

sudo chown -R normal_user:normal_user /home/normal_user/ssh_env/

Now both users will have this structure:

bash

Copy

Edit

```
~/ssh_env/  
|--- backup_script/  
|--- rbac_system/
```

And they can each run the code independently.

Let me know if you'd also like to symlink common log/output directories or set up virtual environments per user.

You said:

now, when I attempted to test admin_user, this was the error message:

```
sh_env) admin_user@kasra-None:~/ssh_env/rbac_system$ python3 test_client.py
```

Traceback (most recent call last):

```
  File "/home/admin_user/ssh_env/rbac_system/test_client.py", line 1, in <module>  
    from ssh_client import connect_ssh, run_command  
  File "/home/admin_user/ssh_env/rbac_system/ssh_client.py", line 3, in <module>  
    from sftp_transfer import handle_file_transfer  
  File "/home/admin_user/ssh_env/rbac_system/sftp_transfer.py", line 16, in <module>  
    sftp_logger = setup_logger('SFTPLogger', 'rbac_system/client.log')  
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^  
  File "/home/admin_user/ssh_env/rbac_system/logger_config.py", line 9, in setup_logger  
    file_handler = logging.FileHandler(log_file)  
    ^^^^^  
  File "/usr/lib/python3.12/logging/__init__.py", line 1231, in __init__  
    StreamHandler.__init__(self, self._open())  
    ^^^^^  
  File "/usr/lib/python3.12/logging/__init__.py", line 1263, in _open  
    return open_func(self.baseFilename, self.mode,  
    ^^^^^  
FileNotFoundException: [Errno 2] No such file or directory: '/home/admin_user/ssh_env/rbac_system/rbac_system/client.log'
```



ChatGPT said:

Ah, got it — this error is happening because your logger is trying to write to:

swift

Copy

Edit

/home/admin_user/ssh_env/rbac_system/rbac_system/client.log

But that path doesn't exist — specifically, the nested rbac_system/ directory inside itself is not real.

Why it happened

In your code, you have this line:

python

Copy

Edit

```
sftp_logger = setup_logger('SFTPLogger', 'rbac_system/client.log')
```

Since you're already in the rbac_system/ directory, this becomes:

bash

Copy

Edit

rbac_system/rbac_system/client.log

Which doesn't exist — hence the FileNotFoundError.

Fix it

You just need to update the logger path to point to the correct place:

⚡ Fix in both ssh_client.py and sftp_transfer.py:

Change:

python

Copy



Edit

```
setup_logger('SFTPLoader', 'rbac_system/client.log')
```

To:

python

Copy

Edit

```
setup_logger('SFTPLoader', 'client.log')
```

Or better yet (more robust):

python

Copy

Edit

```
import os
```

```
log_path = os.path.join(os.path.dirname(__file__), 'client.log')
```

```
setup_logger('SFTPLoader', log_path)
```

That way, the log will always be saved in the same directory as the script, no matter where the script is run from.

✍ Optional: Clean up old logs

If you accidentally created any empty directories or logs in the wrong path, clean them up:

bash

Copy

Edit

```
rm -rf /home/admin_user/ssh_env/rbac_system/rbac_system
```

After this fix, re-run:

bash

Copy

Edit

```
python3 test_client.py
```

Let me know if you'd like to apply the same fix for backup_script too.