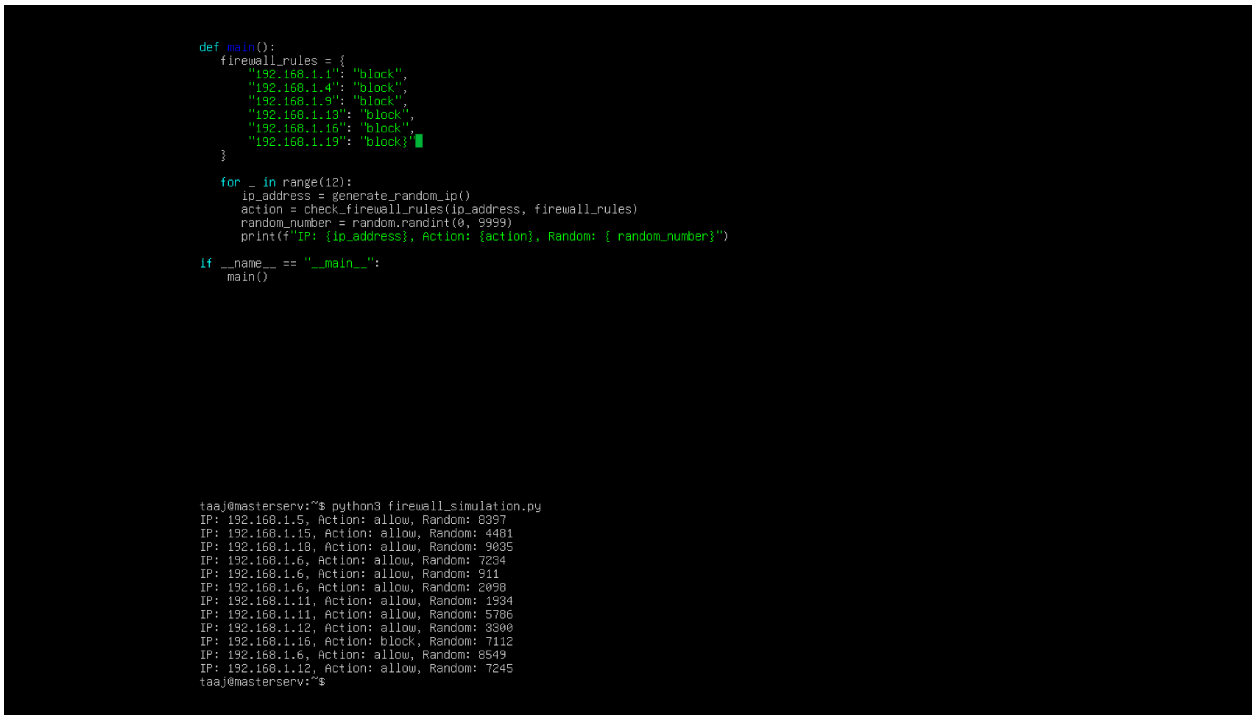
🔎 Group Project Summary: Learning Python Indentation and Firewall Simulation and 🚀 Encryption & Decryption Simulation Adventure 🚀

Aaron Davidson
Shannon Kelly
Danielle Reid, Nyah Hepburn
TaZahnae Matthews



```
def main():
   firewall_rules = {
       "192.168.1.1": "block",
       "192.168.1.4": "block",
       "192.168.1.9": "block",
       "192.168.1.13": "block",
       "192.168.1.16": "block",
       "192.168.1.19": "block}"
   }

   for _ in range(12):
      ip_address = generate_random_ip()
      action = check_firewall_rules(ip_address, firewall_rules)
      random_number = random.randint(0, 9999)
      print(f"IP: {ip_address}, Action: {action}, Random: { random_number}")

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

```
taaj@masterserv:~$ python3 firewall_simulation.py
IP: 192.168.1.5, Action: allow, Random: 8397
IP: 192.168.1.15, Action: allow, Random: 4481
IP: 192.168.1.18, Action: allow, Random: 9035
IP: 192.168.1.6, Action: allow, Random: 7234
IP: 192.168.1.6, Action: allow, Random: 911
IP: 192.168.1.6, Action: allow, Random: 2098
IP: 192.168.1.11, Action: allow, Random: 1934
IP: 192.168.1.11, Action: allow, Random: 5786
IP: 192.168.1.12, Action: allow, Random: 3300
IP: 192.168.1.16, Action: block, Random: 7112
IP: 192.168.1.6, Action: allow, Random: 8549
IP: 192.168.1.12, Action: allow, Random: 7245
taaj@masterserv:~$
```

For this project, we built a firewall simulation that checks whether an IP address should be allowed or blocked. First, we created a list of random IP addresses and chose specific ones to block, like "192.168.1.1" and "192.168.1.4". The simulation generates random IPs and checks each one against the firewall's block list to decide if it should be allowed or denied.

While doing this, we ran into challenges with Python indentation. Python is strict about indentation because it uses spaces or tabs to define blocks of code. If tabs and spaces are mixed—even by mistake—it causes errors like IndentationError or TabError. These errors were tricky because tabs and spaces look the same but act differently

Here is a bullet-point summary of what we did

- **import random:**
  Imports the random module, which lets you generate random choices (for deciding
  whether packets are allowed or blocked).

- **def ():**
  - Defines a function . A **function** is a reusable block of code that performs a specific task. In this case, it simulates the decision-making process for a packet.
- **random.choice(['safe', 'blocked']):**
  - Uses random.choice to randomly pick either "allow" or "blocked" from a list, simulating the firewall's packet decision.
- **if statements (optional):**
  - You could use **if statements** to further control logic, for example, checking if the packet is "allow" or "blocked" and performing different actions for each.
- **for loop:**
  - Runs the simulate_packet() function multiple times .Simulating multiple packets.

🚫 Errors & Fixes

- **Indentation Error:** Mixed tabs/spaces → Fixed with 4 spaces only.
- **Syntax Error:** Mistyped MY ERROR >>: if __name__ == "__main__": → Fixed spelling/punctuation.
- **Undefined Variable:** random_number used outside scope → Moved inside loop.
- **Scope Error:** firewall_rules used outside main() → Moved inside main().

🛠️ Key Fixes (correct way)

- All logic is now inside main().
- Added proper entry point: if __name__ == "__main__": main()

⚠️ Reminders

- Always check indentation (no tabs).
- Double-check function and variable names.
- Watch punctuation & syntax carefully.

🤔 Identifying Lines and Indentation in Python 🤔

During the learning process, one of the key challenges faced was **understanding how Python uses indentation** and how **errors appear when indentation is wrong.** Python uses **indentation (spaces or tabs) to define blocks of code,** and mixing tabs and spaces often causes **IndentationError** or **TabError.**

Common errors encountered:

- IndentationError: unexpected indent
- IndentationError: unindent does not match any outer indentation level
- TabError: inconsistent use of tabs and spaces in indentation

---

🚩 Key Lessons Learned:

1️⃣ Error Messages Help Locate Problems:

- Python shows **exact line numbers** where errors occur.
- Example:
  `File "firewall_simulation.py", line 8 ... IndentationError`

👉 **Lesson:** Always check the **line number** first to narrow down the issue.

2️⃣ Identifying Tabs vs. Spaces:

- Tabs and spaces **look the same** in the shell but behave differently.
- Moving the cursor line-by-line helped notice that:
  - **Tabs jump the cursor further.**
  - **Spaces move the cursor little by little.**

👉 **Lesson:** Be careful of invisible characters; they can break the code even if it *looks fine*.

3️⃣ Viewing Hidden Characters:

- **In Vim:**
  Using `:set list` shows:
  - `^I` for tabs
  - `·` for spaces
- **In Nano:**
  Watch the **cursor movement** to detect tabs vs. spaces.
- **In VS Code or GUI editors:**
  Turn on **"Render Whitespace"** to see dots and arrows for spaces and tabs.

👉 **Lesson:** Choose an editor that lets you **see hidden characters** clearly.

4️⃣ **Using Command-Line Tools:**

- `nl filename.py` or `cat -n filename.py` prints the file with **line numbers** (helpful for quick reviews).
- Running the file with:

  ```
  python -tt filename.py
  ```

  checks for **mixed tab/space issues** even if the code seems OK.
- If you want to **save a copy of your file with line numbers in it** (for sharing or printing): `nl firewall_simulation.py > numbered_firewall_simulation.py`
- ✅ Now you'll have a **new file** called `numbered_firewall_simulation.py` with the line numbers included as part of the text.
-

👉 **Lesson:** Simple tools can **quickly spot problems** without opening an editor.

5️⃣ **Best Practices Moving Forward:**

- Always use **4 spaces per indentation** (Python's standard).
- Avoid using tabs unless the entire file uses tabs consistently.
- Consider running **auto-formatters** like `autopep8` to clean up files automatically.

✅ Final Takeaway:

Understanding where lines and indentation break in Python is **essential for error-free scripts.** Although it can be frustrating at first (with invisible tabs/spaces causing unexpected errors), tools like **Vim, Nano, nl, and Python's own error messages** provide clear ways to **identify and fix these issues quickly.**

The biggest lesson: **always check your indentation carefully, keep it consistent, and use tools that make invisible formatting visible.**

🚀 Encryption & Decryption Simulation Adventure 🚀

Our Step-by-Step Journey

Before we start our live demo let's give some slight background

Encryption is the process of converting readable data into an unreadable format to protect its confidentiality. Only individuals with the correct password or decryption key can access the original information.

Decryption is the process of converting the encrypted data back into its original, readable form using the appropriate password or key.

In summary, encryption secures the data, while decryption restores access to the protected information.

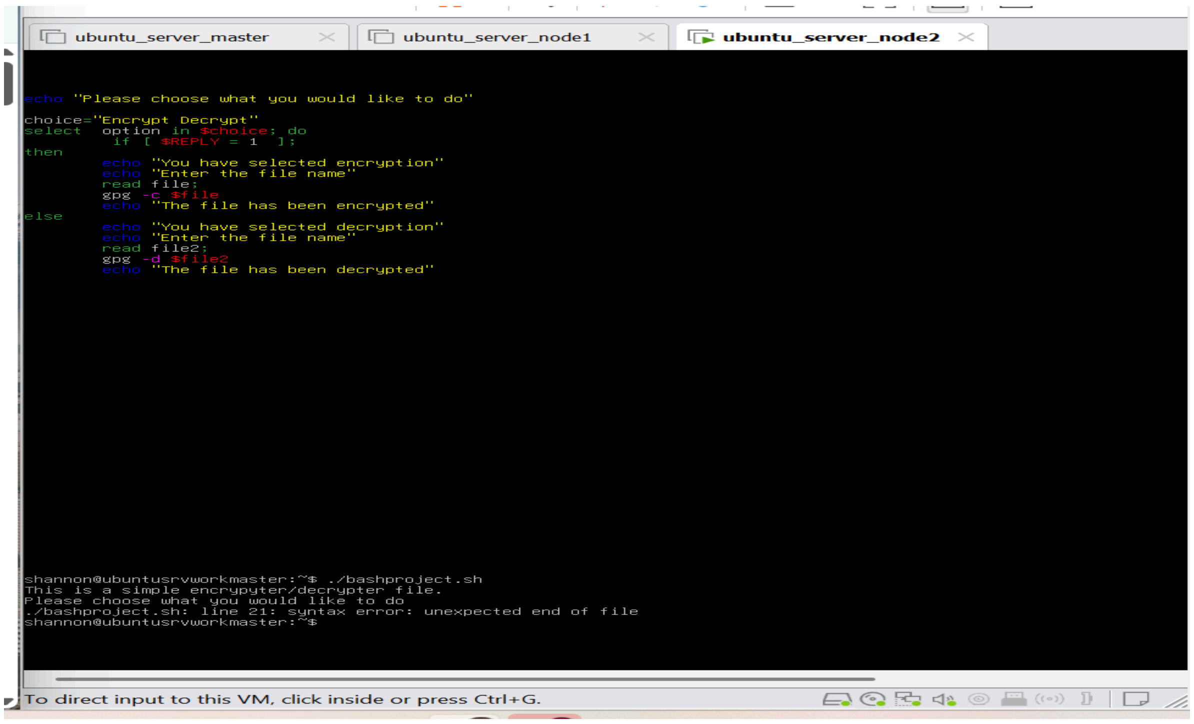
In short:
🔐 **Encryption = locking** the information.
🔓 **Decryption = unlocking** the information.

🔧 File Creation

We kicked things off by creating a test file on the virtual machine — our secret message was ready to be protected!

💥 Crafting the Magic Script

We wrote a Bash script with cool options to encrypt, decrypt, or exit like pros:



```
echo "Please choose what you would like to do"

choice="Encrypt Decrypt"
select  option in $choice; do
         if [ $REPLY = 1  ];
then
        echo "You have selected encryption"
        echo "Enter the file name"
        read file;
        gpg -c $file
        echo "The file has been encrypted"
else
        echo "You have selected decryption"
        echo "Enter the file name"
        read file2;
        gpg -d $file2
        echo "The file has been decrypted"
```

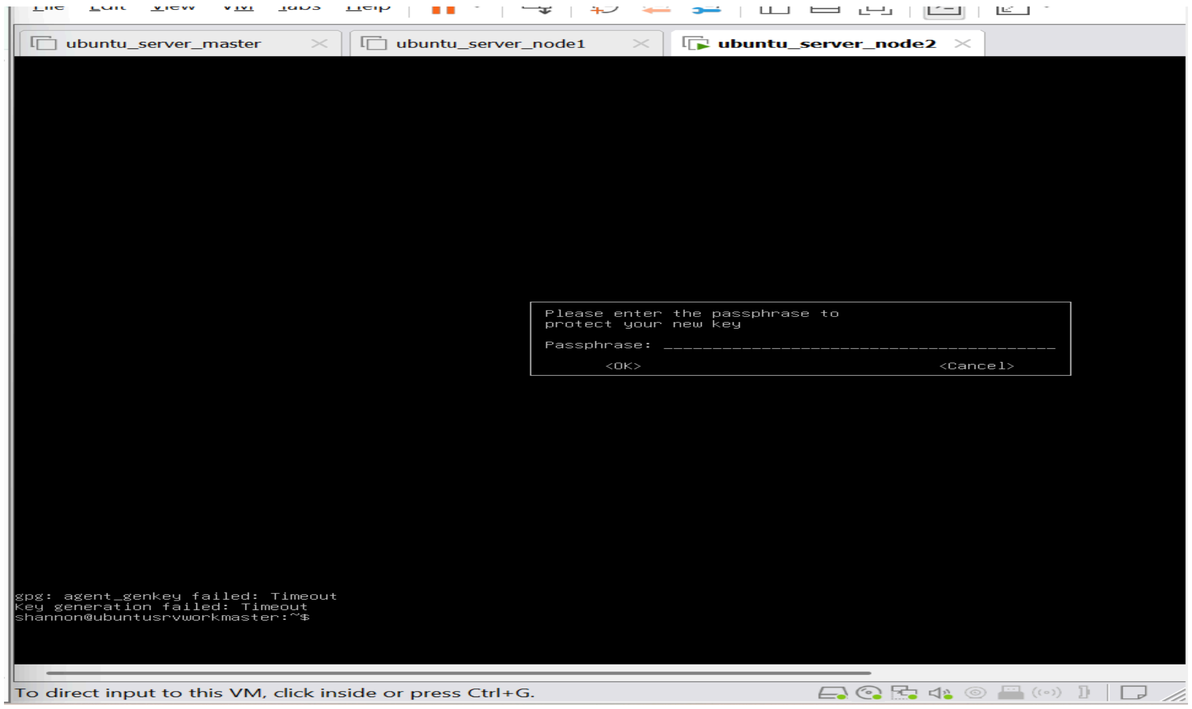
```
shannon@ubuntusrvworkmaster:~$ ./bashproject.sh
This is a simple encrypyter/decrypter file.
Please choose what you would like to do
./bashproject.sh: line 21: syntax error: unexpected end of file
shannon@ubuntusrvworkmaster:~$
```

To direct input to this VM, click inside or press Ctrl+G.

💻

Running the Script

Once we added the code, the script asked if we wanted to encrypt or decrypt. We chose encrypt — but unfortunately, we timed out on the password prompt and ran into an error. 😬 (YOU HAVE NOT ENTERED A PASSPHRASE)

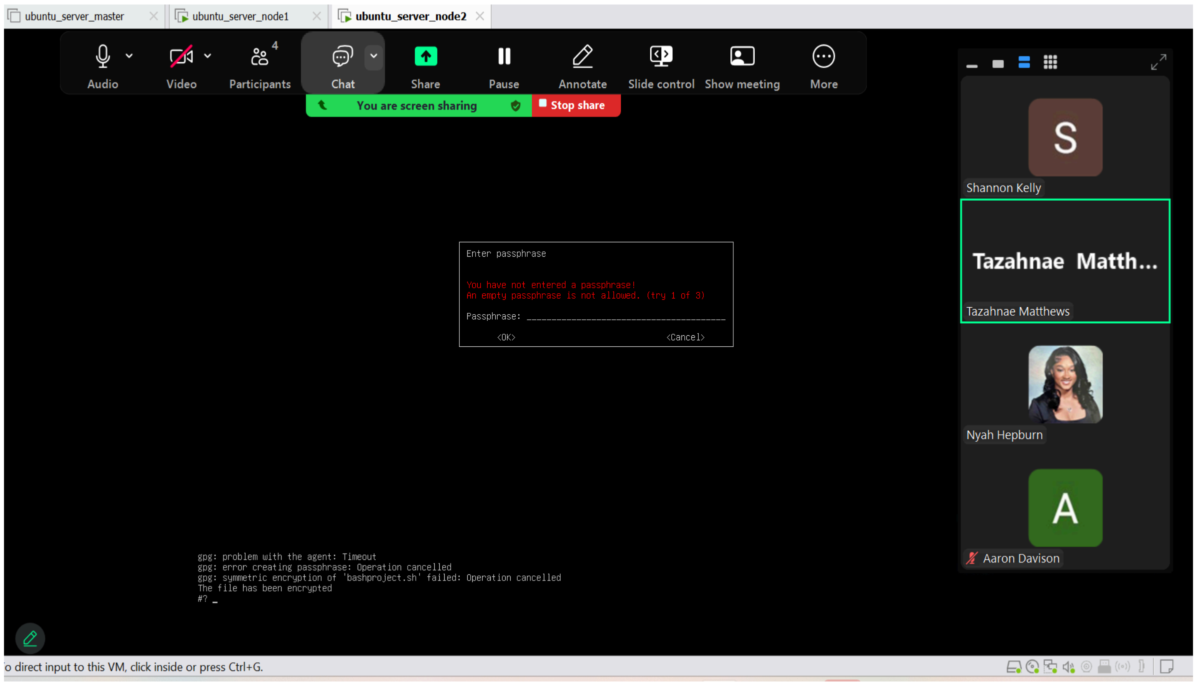
3. 

ubuntu_server_master | ubuntu_server_node1 | ubuntu_server_node2

```
Please enter the passphrase to
protect your new key

Passphrase: ________________________________________

     <OK>                              <Cancel>
```

```
gpg: agent_genkey failed: Timeout
Key generation failed: Timeout
shannon@ubuntusrvworkmaster:~$
```

To direct input to this VM, click inside or press Ctrl+G.



ubuntu_server_master | ubuntu_server_node1 | ubuntu_server_node2

```
Enter passphrase


You have not entered a passphrase!
An empty passphrase is not allowed. (try 1 of 3)

Passphrase: ________________________________________

     <OK>                              <Cancel>
```

```
gpg: problem with the agent: Timeout
gpg: error creating passphrase: Operation cancelled
gpg: symmetric encryption of 'bashproject.sh' failed: Operation cancelled
The file has been encrypted
#? _
```

o direct input to this VM, click inside or press Ctrl+G.

4. THIS IS US WORKING IT OUT .... 🤔 ⏰ **The Password Hiccup**

Here's where it got interesting — Shannon's timeout caused the encryption to fail, and we couldn't continue.

💡 Finding a Workaround💡

We explored solutions and for documentation purposes, we noted two possible fixes for future reference:

🛠️ Fix Option 1 (documented): Use `-pass stdin` to pass the password AFTER it starts

This makes OpenSSL **wait for the password via standard input** and avoids any timeout issues. (You can run **openssl enc -aes-256-cbc -salt -in "$file" -out "${file}.enc" -pass pass:$pass**

In your script type

```
 echo -n "Enter password: "
        read -s pass echo "$pass" | openssl enc -aes-256-cbc -salt -in "$file"
        -out "${file}.enc" -pass stdin
```

🛠️ Fix Option 2: Don't Pass Password Inline—Let OpenSSL Prompt

If you don't **pass `-pass` at all**, OpenSSL **automatically prompts you**, and you get two prompts:

1. Enter encryption password
2. Verify encryption password

You can change the script to:openssl enc -aes-256-cbc -salt -in "$file" -out "${file}.enc"

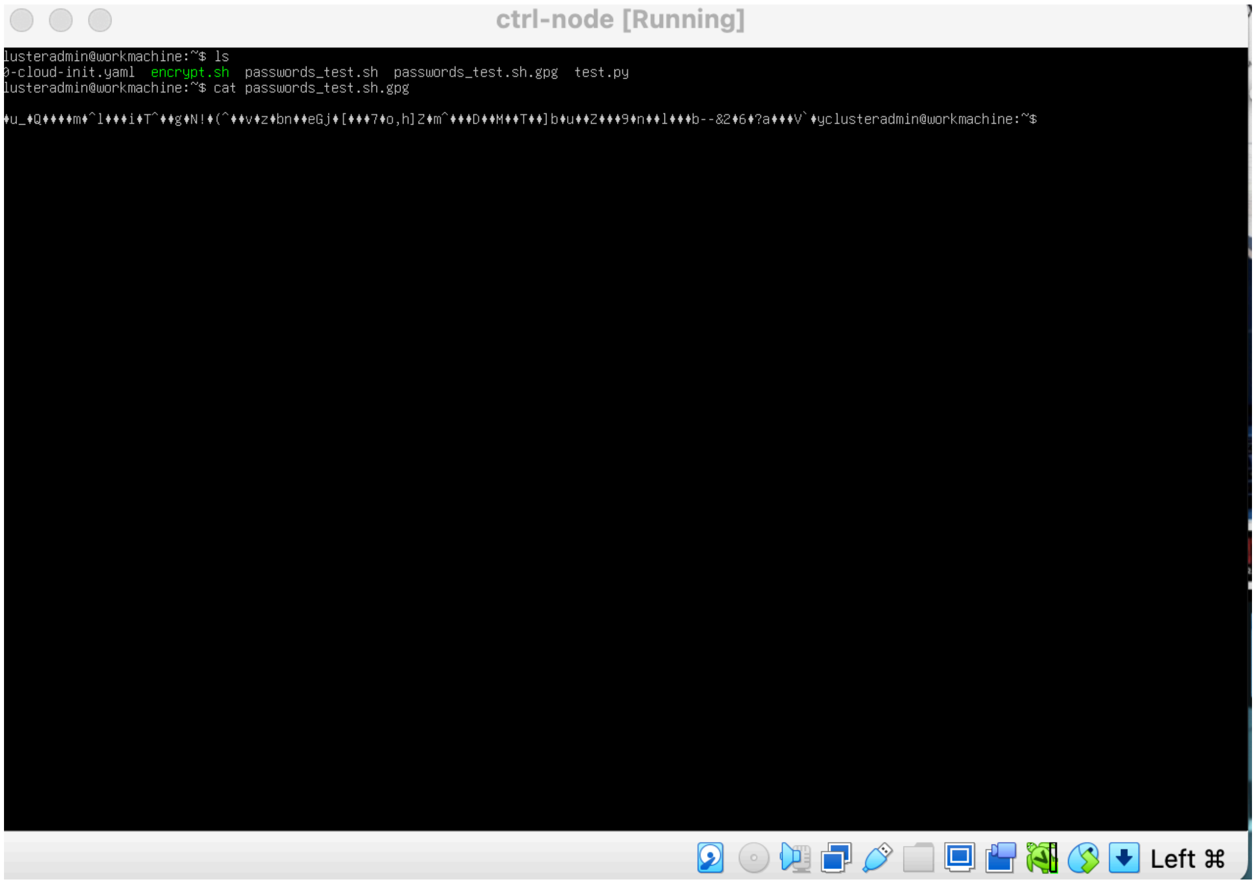
This way, **YOU control the timing**, and there's **no 60-second timeout**—OpenSSL just waits until you type the password and confirm it.

🌍 Switching Machines

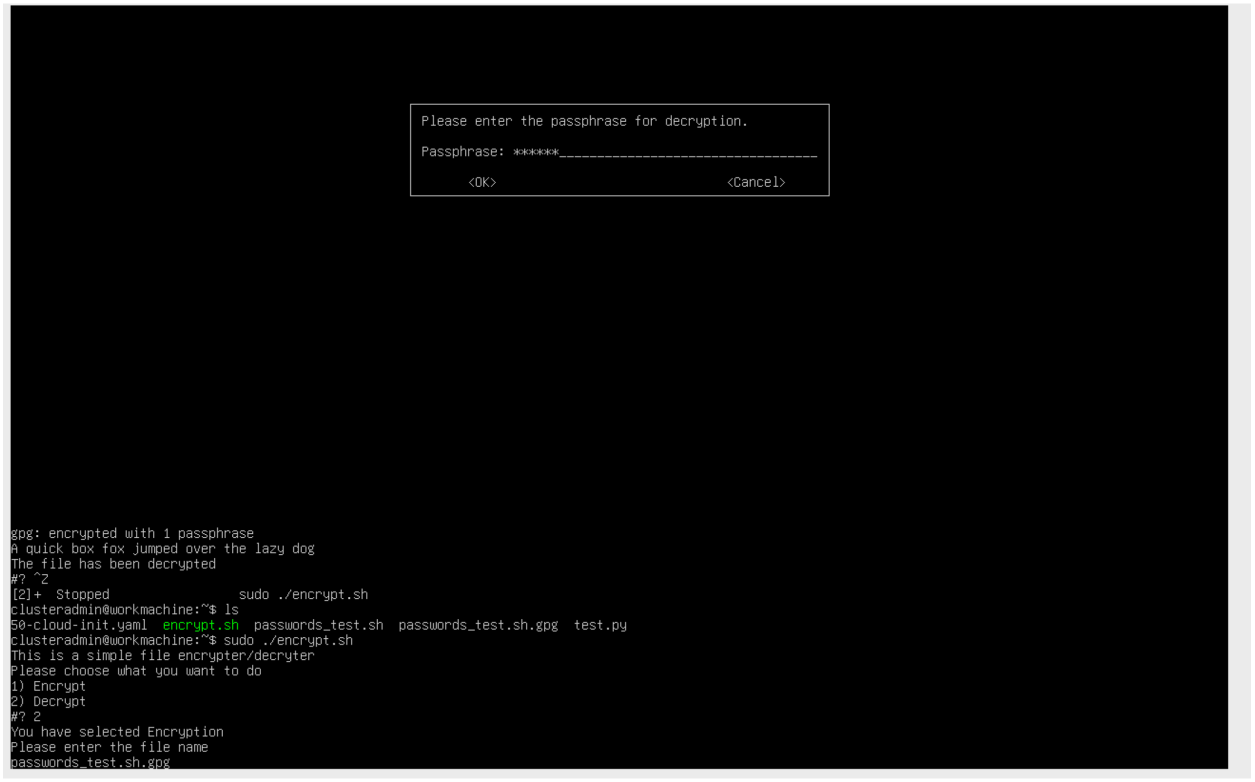
When all else failed, we hopped onto another teammate's machine. The energy was high, keyboards were clacking, and together we powered through encryption and decryption.

🔐 Encryption Phase (Locking the File)

This is when we take a normal file (like a text document) and scramble its contents so no one can read it unless they have the correct password. Think of it like locking your diary with a secret code — only someone with the code can open and read it.



```
lusteradmin@workmachine:~$ ls
0-cloud-init.yaml  encrypt.sh  passwords_test.sh  passwords_test.sh.gpg  test.py
lusteradmin@workmachine:~$ cat passwords_test.sh.gpg

♦u_♦Q♦♦♦♦m♦^l♦♦♦i♦T^♦♦g♦N!♦(^♦♦v♦z♦bn♦♦eGj♦[♦♦♦7♦o,h]Z♦m^♦♦♦D♦♦M♦♦T♦♦]b♦u♦♦Z♦♦♦9♦n♦♦l♦♦♦b--&2♦6♦?a♦♦♦V`♦yclusteradmin@workmachine:~$
```

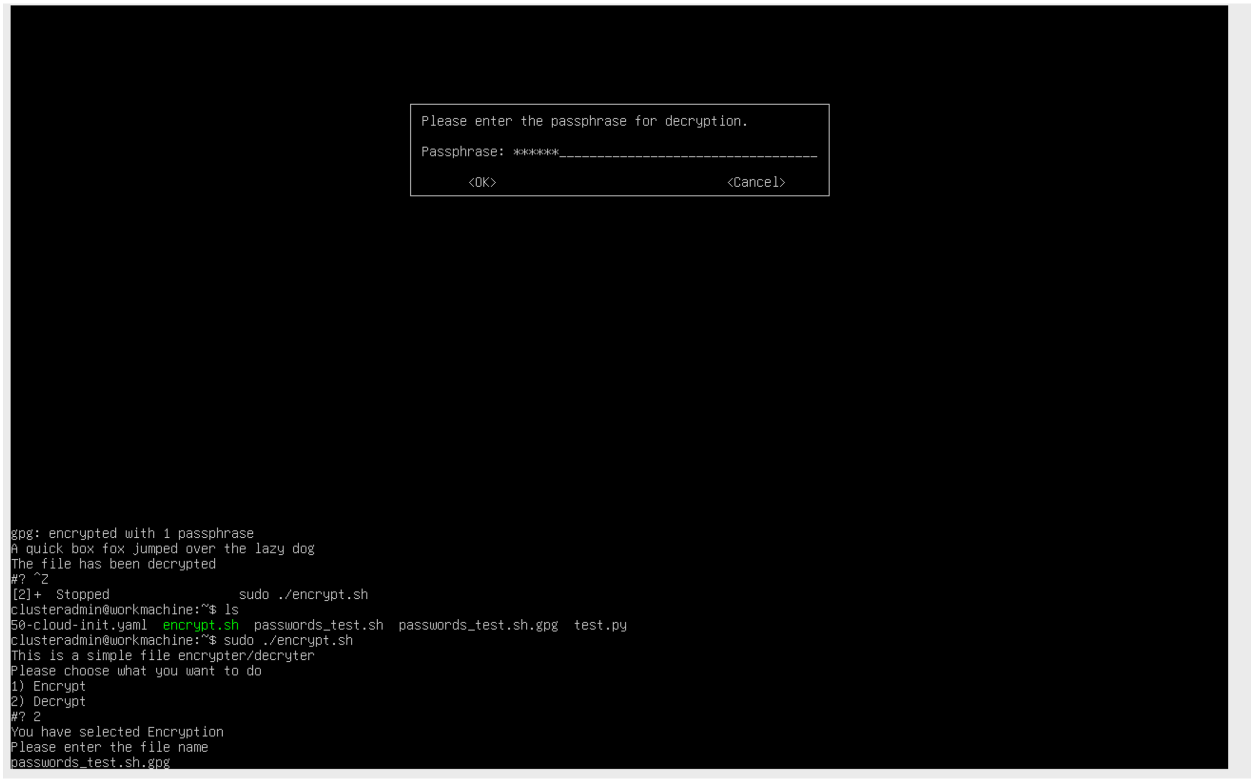


```
Please enter the passphrase for decryption.

Passphrase: ******___________________________________

      <OK>                              <Cancel>
```

```
gpg: encrypted with 1 passphrase
A quick box fox jumped over the lazy dog
The file has been decrypted
#? ^Z
[2]+  Stopped                 sudo ./encrypt.sh
clusteradmin@workmachine:~$ ls
50-cloud-init.yaml  encrypt.sh  passwords_test.sh  passwords_test.sh.gpg  test.py
clusteradmin@workmachine:~$ sudo ./encrypt.sh
This is a simple file encrypter/decryter
Please choose what you want to do
1) Encrypt
2) Decrypt
#? 2
You have selected Encryption
Please enter the file name
passwords_test.sh.gpg
```

.🔓 Decryption Phase (Unlocking the File)

This is when we take the scrambled (encrypted) file and use the right password to unlock it, turning it back into a readable file.



```
Please enter the passphrase for decryption.

Passphrase: ******___________________________________

      <OK>                                  <Cancel>
```

```
gpg: encrypted with 1 passphrase
A quick box fox jumped over the lazy dog
The file has been decrypted
#? ^Z
[2]+  Stopped                 sudo ./encrypt.sh
clusteradmin@workmachine:~$ ls
50-cloud-init.yaml  encrypt.sh  passwords_test.sh  passwords_test.sh.gpg  test.py
clusteradmin@workmachine:~$ sudo ./encrypt.sh
This is a simple file encrypter/decryter
Please choose what you want to do
1) Encrypt
2) Decrypt
#? 2
You have selected Encryption
Please enter the file name
passwords_test.sh.gpg
```

🧠 Lessons Learned

We learned that teamwork, adaptability, and clever troubleshooting make all the difference in tech projects.

🌟 Final Takeaway

We mastered file encryption and decryption, cracked the password timeout challenge, and most importantly, worked together like champions. Way to go, team!