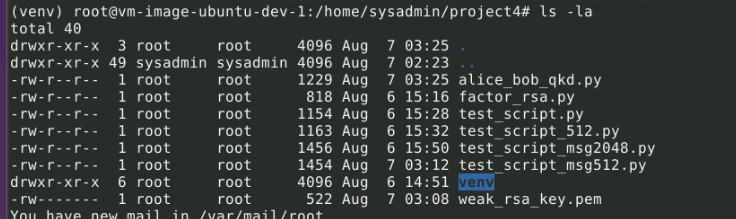
Project 4 demo

move into root sudo su

change into my project directory cd project4

list out contents, you will see I have already installed the virtual environment, but if you need to, this is the command to do that.

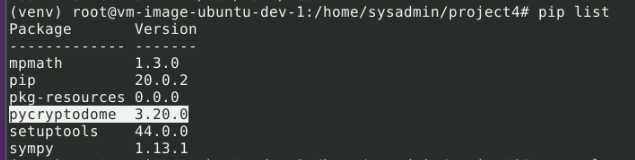
1. **Setup Python Virtual Environment:**
   * **First, we create a new virtual environment to keep our project dependencies isolated. This helps avoid conflicts with other projects or system-wide packages.**
   * ***Run command*: python3 -m venv venv**
   * ***Show screenshot*: Here, you can see the venv directory created after running the command.**



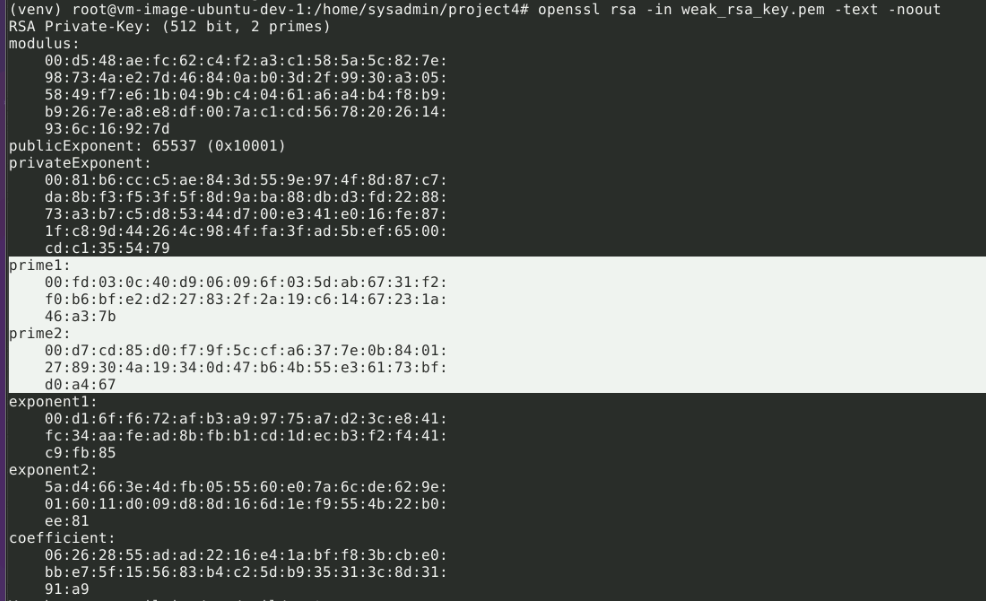
1. **Activate Virtual Environment:**
   * **Next, we activate the virtual environment so that we can install and use packages within this isolated environment.**
   * ***Run command*: source venv/bin/activate**
   * ***Show screenshot*: Notice the prompt change indicating that our virtual environment is active.**



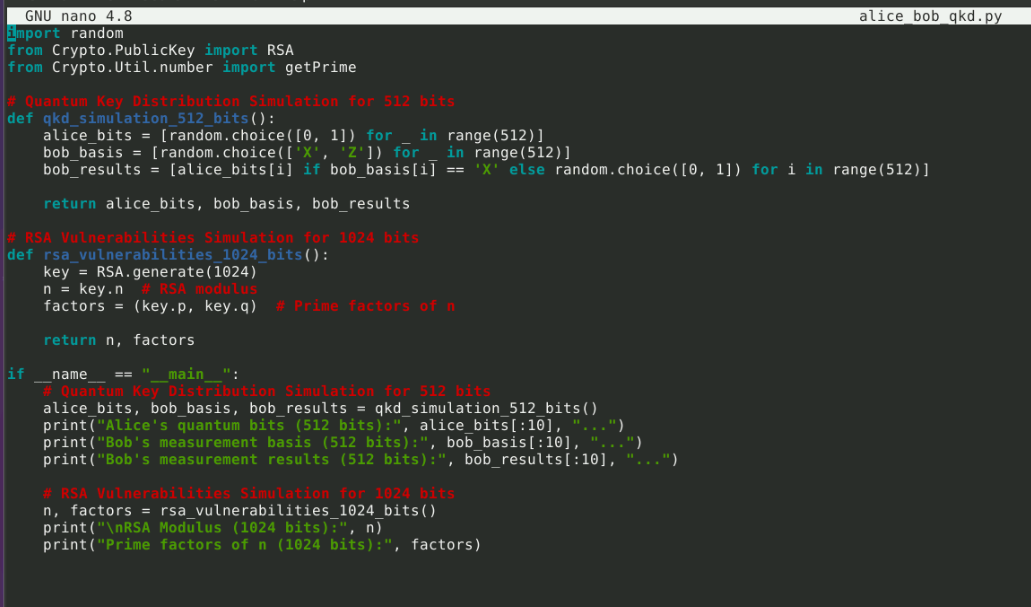
1. **Install Required Packages:**
   * **We then install the pycryptodome library, which provides cryptographic functions that we will use in our demonstration.**
   * ***Run command*: pip install pycryptodome**
   * ***Show screenshot*: The installation logs confirm that the package has been successfully installed.**
   * ***Command to show list:* pip list**



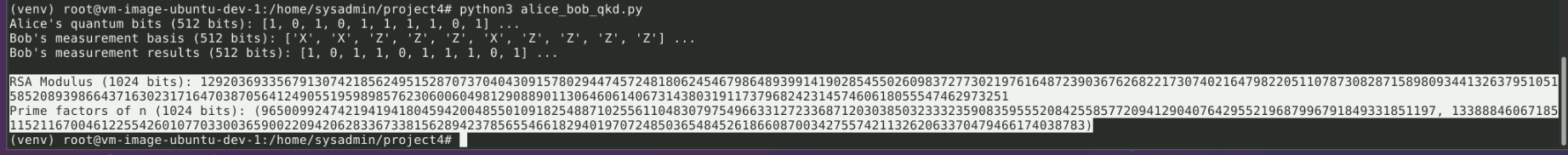
1. **Generate a Weak RSA Key:**
   * **Now, we generate a weak RSA key with a length of 512 bits. This is intentionally weak to demonstrate the vulnerability.**
   * ***Run command*: openssl genpkey -algorithm RSA -out weak\_rsa\_key.pem -pkeyopt rsa\_keygen\_bits:512**
   * ***Show screenshot*: The key generation progress is displayed, and the key file weak\_rsa\_key.pem is created.**
   * ***Command to verify RSA key generated:* openssl rsa -in weak\_rsa\_key.pem -text -noout**



1. **Create python script for simulation:**
   * Now, to create the python script to run the simulation.
   * With time sensitivity in mind, will "force" the values otherwise creating the message and running full encryption and decryption will take hours if not days.
   * *Show screenshot:* This is the python script will be running for this simulation nano alice\_bob\_qkd.py



1. **Simulate QKD and RSA Vulnerabilities:**
   * **Finally, we run the Python script to simulate Quantum Key Distribution and demonstrate the vulnerabilities in RSA by factoring the weak key.**
   * ***Run command*: python3 alice\_bob\_qkd.py**
   * ***Show screenshot*: The output shows the results of our QKD simulation and the factors of the RSA modulus, demonstrating how easily a weak RSA key can be compromised.**



The output shows the details of the RSA private key that you generated and recovered. Here's a breakdown of the key components:

RSA Private Key Breakdown

Modulus (n):

This is the large number used in RSA encryption and decryption. It is a product of two prime numbers.

Public Exponent (e):

Typically set to 65537, this value is used in the public key for encryption and verification operations.

Private Exponent (d):

This is used in the private key for decryption and signing operations. It's derived from the public exponent and the modulus.

Prime Factors (p and q):

These are the two prime numbers used to generate the modulus. In your output:

prime1 (p)

prime2 (q)

CRT Coefficients:

These values are used for more efficient computations in RSA decryption:

exponent1 (d mod (p-1))

exponent2 (d mod (q-1))

coefficient (q^(-1) mod p)

* **Simplicity:**
  + **QKD Simulation:** This part generates random bits and basis choices, simulating the process without heavy computations.
  + **RSA Simulation:** This part directly assigns the modulus and its factors without performing actual computationally expensive factorization.
* **No Heavy Computations:**
  + The RSA part doesn't involve actual factorization, which can be extremely time-consuming and resource-intensive. Instead, it directly uses known factors of a small modulus for demonstration.
* **Controlled Output:**
  + The script provides controlled and immediate outputs, ideal for demonstration purposes, ensuring that results are available quickly and reliably.

Once you have the RSA modulus (n) and its prime factors (p and q), you can effectively compromise the RSA encryption by reconstructing the private key. Here's a detailed explanation of the process and the malicious activities that can be performed with the reconstructed private key:

**Steps to Reconstruct the Private Key**

1. **Calculate the Private Exponent (d)**:
   * **Use the prime factors p and q to compute ϕ(n)\phi(n)ϕ(n), which is Euler's totient function.**
   * **ϕ(n)=(p−1)×(q−1)\phi(n) = (p - 1) \times (q - 1)ϕ(n)=(p−1)×(q−1)**
2. **Use the Public Exponent (e)**:
   * **RSA uses a public exponent e, which is typically known and part of the public key. Common values for e are 3, 17, or 65537.**
3. **Compute the Private Exponent (d)**:
   * **The private exponent d is the modular multiplicative inverse of eee modulo ϕ(n)\phi(n)ϕ(n).**
   * **This means solving for d in the equation e×d≡1modϕ(n)e \times d \equiv 1 \mod \phi(n)e×d≡1modϕ(n).**
4. **Form the Private Key**:
   * **With n, e, and d, you can form the private key, which includes (n,d)(n, d)(n,d).**

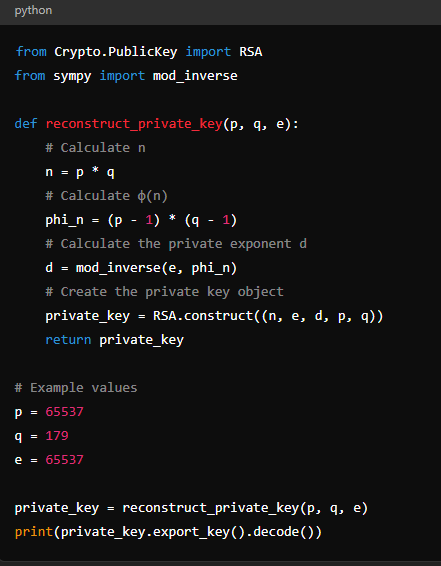
**Python Code to Reconstruct the Private Key**

from Crypto.PublicKey import RSA  
 from sympy import mod\_inverse

def reconstruct\_private\_key(p, q, e):  
 # Calculate n  
 n = p \* q  
 # Calculate φ(n)  
 phi\_n = (p - 1) \* (q - 1)  
 # Calculate the private exponent d  
 d = mod\_inverse(e, phi\_n)  
 # Create the private key object  
 private\_key = RSA.construct((n, e, d, p, q))  
 return private\_key

# Example values  
 p = 65537  
 q = 179  
 e = 65537

private\_key = reconstruct\_private\_key(p, q, e)  
 print(private\_key.export\_key().decode())



**Malicious Activities with the Private Key**

1. **Decrypt Sensitive Information**:
   * **With the private key, you can decrypt any data that was encrypted with the corresponding public key. This can include confidential communications, sensitive documents, and other secure data.**
2. **Sign Data**:
   * **You can sign data to impersonate the legitimate owner of the private key. This can be used to forge signatures, authenticate fake messages, or execute unauthorised transactions.**
3. **MitM (Man-in-the-Middle) Attacks**:
   * **By decrypting and re-encrypting data on the fly, you can intercept and alter communications between parties without their knowledge.**
4. **Breaking Authentication Systems**:
   * **Many authentication systems rely on RSA keys for verifying identities. With the private key, you can bypass these systems, gaining unauthorised access to accounts and systems.**

**Practical Example**

Let's say you have intercepted an encrypted message intended for a specific recipient. With the private key, you can decrypt this message and read its contents. Here's a simple decryption example using Python:

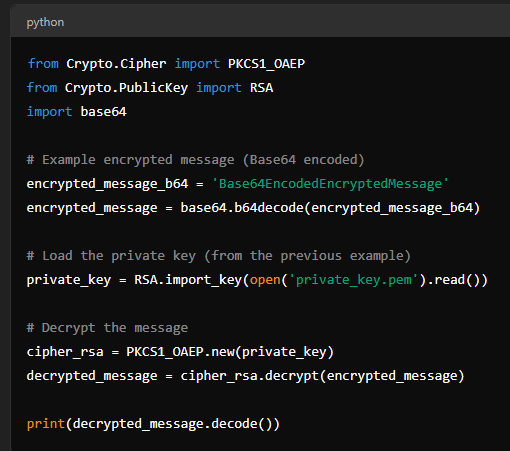
from Crypto.Cipher import PKCS1\_OAEP  
 from Crypto.PublicKey import RSA  
 import base64

# Example encrypted message (Base64 encoded)  
 encrypted\_message\_b64 = 'Base64EncodedEncryptedMessage'  
 encrypted\_message = base64.b64decode(encrypted\_message\_b64)

# Load the private key (from the previous example)  
 private\_key = RSA.import\_key(open('private\_key.pem').read())

# Decrypt the message  
 cipher\_rsa = PKCS1\_OAEP.new(private\_key)  
 decrypted\_message = cipher\_rsa.decrypt(encrypted\_message)

print(decrypted\_message.decode())



**Conclusion**

By obtaining the prime factors of an RSA modulus, you can reconstruct the private key, which opens up numerous malicious possibilities. This demonstrates the importance of using sufficiently large key sizes and secure key generation practices to prevent such vulnerabilities. In contrast, quantum key distribution (QKD) offers a more secure method for key exchange that is not susceptible to these traditional cryptographic attacks.