

Lab Report 1-4

Author: Joo Kai Tay (22489437)

Lab 1: Introduction & Setup

Section 1: Creation of AWS Account

1. Navigate to <https://489389878001.signin.aws.amazon.com/console> and login using the provided username and password, changing the password to a secure one on login.

 AWS login screen


2. Navigate to the security credentials tab within IAM. Here, the user can view their details such as their ARN and canonical ID.

 Security credentials screen


3. Select the option to create an access key for access to programmatic calls to AWS from the AWS CLI.

 Create Access key

4. Select the option for use with the command line interface (CLI)

 Access key best practices

5. Give the access key a meaningful description tag. This will be useful in the event that you create multiple access keys. The tag should describe the purpose of the key and where it will be used.

 Access key tag

6. Once the key has been successfully created, download the access key and secret access key in the .csv file and store it securely. If the key is not stored securely, any user with access to the key can use it to create and use resources associated with your account. For privacy purposes, the key is censored in this image.

 Successful creation of access key

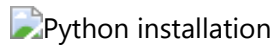
Section 2: VM Setup

The Ubuntu virtual machine was already setup prior to the commencement of this unit. The specifications can be found in the attached image.

 Virtual machine specifications

Section 3: Software Setup

1. Python 3.10.6 and pip3 were already installed on the machine prior to the commencement of the unit.



2. The AWS CLI was installed and configured using the access key generated as part of step 6 in section 1

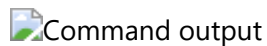


3. Boto3 was installed as per the instructions. Boto3 is the AWS SDK for python and it provides a python API for AWS infrastructure services.



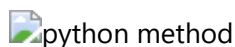
Section 4: Exploring and testing the environment

1. Running the command `aws ec2 describe-regions --output table` is used to retrieve information about the available AWS regions for the EC2 service and display the output in a tabular format. The output of the command is shown below:



2. We can retrieve similar information using Python3 by running the following commands. This returns and prints the same data as before. However, this data is not tabulated.

```
python3
>>> import boto3
>>> ec2 = boto3.client('ec2')
>>> response = ec2.describe_regions()
>>> print(response)
```



3. The following code allows us to tabulate the data and print out only the columns containing the endpoint and RegionName:

```
import boto3

# Get the region data from boto3
ec2 = boto3.client('ec2')
response = ec2.describe_regions()

# Print the header for the columns
print(f"{'Endpoint':<40} {'RegionName'}")

# Print each region's endpoint and region name
```

```
for region in response['Regions']:
    print(f"{region['Endpoint']:<40} {region['RegionName']}")
```

The execution of this code (stored in a file named `cits5503_lab1.py`) is shown below:

python file


Lab 2: EC2 and Docker

Section 1: Creating an EC2 instance using awscli


1. Using the command `aws ec2 create-security-group --group-name <student number>-sg --description "security group for development environment"`, a security group is created. The security group ID is noted for use in future steps.

Creating security group


2. The following command is used to authorise inbound traffic for ssh: `aws ec2 authorize-security-group-ingress --group-name <student number>-sg --protocol tcp --port 22 --cidr 0.0.0.0/0`. Some things to note about this command: a. The TCP protocol is specified using `--protocol tcp --port 22` and the default port 22 is selected. b. The command `--cidr 0.0.0.0/0` is used as a wildcard. This allows all IP addresses to connect to SSH port that was opened. This is considered a major security risk as it allows anyone on the internet to attempt SSH connections to instances associated with this security group.

Authorise security group

3. The command `aws ec2 create-key-pair --key-name <student number>-key --query 'KeyMaterial' --output text > <student number>-key.pem` creates a key pair that will allow a user to ssh to the EC2 instance. The private key is stored in the output file `22489437-key.pem`

Creating private key

4. The private key file is moved into the SSH directory and its permissions are changed to be read only for the owner of the file, and no permissions for others.

changing permissions

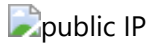
5. The command `aws ec2 run-instances --image-id ami-d38a4ab1 --security-group-ids -sg --count 1 --instance-type t2.micro --key-name -key --query 'Instances[0].InstanceId'` is used to create the EC2 instance. The command returns the instance ID of the created instance.

creating EC2 instance

6. Using the instance ID from step 5, a tag is added to the instance.

create tag

- The public IP address of this instance is determined using the command `aws ec2 describe-instances --instance-ids i-<instance id from above> --query 'Reservations[0].Instances[0].PublicIpAddress'`



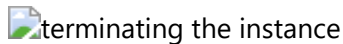
- Using the private key file from step 3 and the IP address from step 7, the user is able to SSH into the EC2 instance.



- The created instance can be viewed from the AWS console



- The instance can be terminated from the awscli using the following command: `aws ec2 terminate-instances --instance-ids i-<your instance id>`



Section 2: Creating an AWS EC2 Instance with Python Boto Script

- The following code is used to replicate steps 1-7 from section 1:

```
import boto3
import os
import shutil

student_number = "22489437"
region = "ap-southeast-2"

# Initialize the EC2 client
ec2 = boto3.client('ec2', region)

# Create a security group
response = ec2.create_security_group(
    GroupName=f"{student_number}-sg",
    Description="security group for development environment"
)
security_group_id = response['GroupId']

# Authorize inbound SSH traffic for the security group
ec2.authorize_security_group_ingress(
    GroupId=security_group_id,
    IpProtocol="tcp",
    FromPort=22,
    ToPort=22,
    CidrIp="0.0.0.0/0"
)

# Create a key pair and save the private key to a file
```

```

response = ec2.create_key_pair(KeyName=f"{student_number}-key")
private_key = response['KeyMaterial']

private_key_file = f"{student_number}-key.pem"
# Allow writing to the private key file
os.chmod(private_key_file, 0o666)
with open(private_key_file, 'w') as key_file:
    key_file.write(private_key)

# Set the correct permissions for the private key file
os.chmod(private_key_file, 0o400)

# Copy the private key file to ~/.ssh directory
ssh_directory = os.path.expanduser("~/.ssh")
if not os.path.exists(ssh_directory):
    os.makedirs(ssh_directory)
shutil.copy(private_key_file, ssh_directory)


# Launch an EC2 instance
response = ec2.run_instances(
    ImageId="ami-d38a4ab1",
    SecurityGroupIds=[security_group_id],
    MinCount=1,
    MaxCount=1,
    InstanceType="t2.micro",
    KeyName=f"{student_number}-key"
)
instance_id = response['Instances'][0]['InstanceId']

# Wait for the instance to be up and running
ec2.get_waiter('instance_running').wait(InstanceIds=[instance_id])

# Describe the instance to get its public IP address
response = ec2.describe_instances(InstanceIds=[instance_id])
public_ip_address = response['Reservations'][0]['Instances'][0]
['PublicIpAddress']
print(f"Instance created successfully with Public IP: {public_ip_address}")


```

2. An error was shown when running the script. This was due to the existing key pair that was created in step 1. In order to resolve this error, we head into the AWS console and delete the previously created key pair.

key pair error

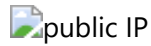
deleting key pair

3. A second error was shown when running the script. This was due to the existing security group that was created in step 1. In order to resolve this error, we head into the AWS console and delete the previously created security group.

security group error



4. The script ran successfully and returned the public IP of the created EC2 instance:



5. The details of the EC2 instance on the AWS console:

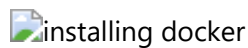


6. This instance was terminated using the AWS console instead of the AWSCLI, demonstrating the second way to terminate an EC2 instance:



Section 3: Using Docker

1. Docker was installed using the command `sudo apt install docker.io -y`



2. Enabling docker and checking the version installed:



3. Creating index.html:



4. Creating the Dockerfile



5. Building the docker image from the dockerfile and index.html



6. Running the docker container:



7. Confirming the "Hello World!" output:



8. Viewing what is running:



9. Terminating the container:



Lab 3: Configure S3 Buckets

1. Preparation: a. Downloading cloudstorage.py



b. Creating rootdir



c. Creating subdir



2. Editing cloudstorage.py to take the initialise argument. The following code segment uses the argparse library to allow the argparse library to take 2 arguments `-i` or `-initialise`. Using either of these 2 arguments will set the `bucketFlag` variable to True. If `bucketFlag` is True, we will attempt to create a bucket in S3. We also catch any exceptions such as `BucketAlreadyExists` and `BucketAlreadyOwnedByYou`.

```
argParser = argparse.ArgumentParser()
argParser.add_argument("-i", "--initialise", action='store_true',
help="create bucket on s3")
args = argParser.parse_args()
bucketFlag = args.initialise
# Insert code to create bucket if not there
if(bucketFlag):
    try:
        response = s3.create_bucket(Bucket=ROOT_S3_DIR,
CreateBucketConfiguration=bucket_config)
        print("Bucket created successfully:", ROOT_S3_DIR)
        print(response)
    except s3.exceptions.BucketAlreadyExists:
        print("Bucket already exists:", ROOT_S3_DIR)
    except s3.exceptions.BucketAlreadyOwnedByYou:
        print("Bucket already owned by you:", ROOT_S3_DIR)
    except Exception as error:
        print("An error occurred:", error)
```

3. The `upload_file()` method found in `cloudstorage.py` was modified to add the code required to upload files to S3. The path of the file in S3 is constructed by adding the `ROOT_S3_DIR` which was created in step 2 to the folder path and file name. The file is then uploaded to S3 using the `s3.upload_file()` command.

```
def upload_file(folder_name, file, file_name):
    # Construct the path for files to be uploaded
    s3Path = f'{ROOT_S3_DIR}/{folder_name}/{file_name}'

    print("Uploading %s" % file)

    # Upload the file to S3
    try:
        s3.upload_file(file, ROOT_S3_DIR, s3Path)
        print(f"Uploaded {file_name} to S3")
    except Exception as e:
        print(f"Error uploading {file_name}: {e}")
```

 uploads  uploads2  uploads3  uploads4

4. The following code which is saved in `restorefromcloud.py` reads the content of the s3 bucket using the command `s3.list_objects` and writes them to the local directory. If the folder structure does not exist in the local directory, the code uses `os.makedirs` to recreate the folder structure from s3 in the local directory.

```
import os
import boto3
ROOT_DIR = '.'
ROOT_S3_DIR = '22489437-cloudstorage'
REGION = 'ap-southeast-2'

s3 = boto3.client("s3", region_name=REGION)
bucket_config = {'LocationConstraint': 'ap-southeast-2'}

for key in s3.list_objects(Bucket = ROOT_S3_DIR)['Contents']:
    if not os.path.exists(os.path.dirname(key['Key'])):
        os.makedirs(os.path.dirname(key['Key']))

    print("Downloading %s" % key['Key'])
    s3.download_file(ROOT_S3_DIR, key['Key'], key['Key'])
```

 downloads

5. Setting up DynamoDB locally

 createdir

 installjre



6. Creating a table with the following attributes: `CloudFiles = { 'userId', 'fileName', 'path', 'lastUpdated', 'owner', 'permissions' })`



7. The following code which is saved in storeinfo.py is used to get the file information from s3 and put it into the DynamoDB table created in step 6.

```
import boto3

ROOT_S3_DIR = '22489437-cloudstorage'
REGION = 'ap-southeast-2'
TABLE = 'CloudFiles'

# Connect to S3 and dynamodb
dynamodb = boto3.resource('dynamodb', region_name=REGION,
endpoint_url='http://localhost:8000')
table = dynamodb.Table(TABLE)
s3 = boto3.client('s3', region_name=REGION)

# List objects in the specified S3 bucket
response = s3.list_objects(Bucket=ROOT_S3_DIR)

# Extract relevant information from the S3 response
userId = str(response['Contents'][0]['Owner']['ID'])
owner = response['Contents'][0]['Owner']['DisplayName']
permission = s3.get_bucket_acl(Bucket=ROOT_S3_DIR)['Grants'][0]
['Permission']

# Iterate through each object in the S3 bucket
for content in response['Contents']:
    # Get attributes requested by lab sheet
    item = {
        'userId': userId,
        'fileName': content['Key'].split('/')[-1],
        'path': content['Key'],
        'lastUpdated': str(content['LastModified']),
        'owner': owner,
        'permissions': permission
    }

    print('Putting the following item into DynamoDB table:\n', item,
'\n')

# Add the item to the DynamoDB table
table.put_item(Item=item)
```

8. The code is run which stores the details of the two files into the **CloudFiles** table:

```
jookai@jookai:~/Desktop/cits5503/lab3$ python3 storefileinfo.py
Putting the following item into CloudFiles table:
{'userId': '2a5fac7aada1ad2caa48c9ab08cc4e2428d4eb596108daa3b59f1204ae96482e', 'fileName': 'rootfile.txt',
'path': '22489437-cloudstorage/rootdir/rootfile.txt', 'lastUpdated': '2023-08-14 03:55:05+00:00', 'owner':
'zhi.zhang', 'permissions': 'FULL_CONTROL'}

Putting the following item into CloudFiles table:
{'userId': '2a5fac7aada1ad2caa48c9ab08cc4e2428d4eb596108daa3b59f1204ae96482e', 'fileName': 'subfile.txt',
'path': '22489437-cloudstorage/rootdir/subdir/subfile.txt', 'lastUpdated': '2023-08-14 03:55:05+00:00', '
owner': 'zhi.zhang', 'permissions': 'FULL_CONTROL'}
```

9. The **CloudFiles** table is scanned using the command `aws dynamodb scan --table-name CloudFiles --endpoint-url http://localhost:8000`. This reveals the details of the two files which were added in step 8.

```
jookai@jookai:~/Desktop/cits5503/lab3$ aws dynamodb scan --table-name CloudFiles --endpoint-url http://localhost:8000
{
  "Items": [
    {
      "owner": {
        "S": "zhi.zhang"
      },
      "path": {
        "S": "22489437-cloudstorage/rootdir/rootfile.txt"
      },
      "lastUpdated": {
        "S": "2023-08-14 03:55:05+00:00"
      },
      "fileName": {
        "S": "rootfile.txt"
      },
      "userId": {
        "S": "2a5fac7aada1ad2caa48c9ab08cc4e2428d4eb596108daa3b59f1204ae96482e"
      },
      "permissions": {
        "S": "FULL_CONTROL"
      }
    },
    {
      "owner": {
        "S": "zhi.zhang"
      },
      "path": {
        "S": "22489437-cloudstorage/rootdir/subdir/subfile.txt"
      },
      "lastUpdated": {
        "S": "2023-08-14 03:55:05+00:00"
      },
      "fileName": {
        "S": "subfile.txt"
      },
      "userId": {
        "S": "2a5fac7aada1ad2caa48c9ab08cc4e2428d4eb596108daa3b59f1204ae96482e"
      },
      "permissions": {
        "S": "FULL_CONTROL"
      }
    }
  ],
  "Count": 2,
  "ScannedCount": 2,
  "ConsumedCapacity": null
}
```

Lab 4: Encryption

Section 1: Applying Policy to Restrict Permissions

1. The following code was used to apply a policy to allow only my username (22489437@student.uwa.edu.au) to access to the S3 bucket identified by `arn:aws:s3:::22489437-cloudstorage` as well as the objects inside the bucket.

```

ROOT_DIR = '.'
ROOT_S3_DIR = '22489437-cloudstorage'
REGION = 'ap-southeast-2'
student_number = '22489437'

s3 = boto3.client("s3", region_name=REGION)
bucket_config = {'LocationConstraint': 'ap-southeast-2'}

policy = {
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowAllS3ActionsInUserFolderForUserOnly",
            "Effect": "DENY",
            "Principal": "*",
            "Action": "s3:*",
            "Resource": [
                "arn:aws:s3:::22489437-cloudstorage/*",
                "arn:aws:s3:::22489437-cloudstorage"
            ],
            "Condition": {
                "StringNotLike": {
                    "aws:username": f"{student_number}@student.uwa.edu.au"
                }
            }
        }
    ]
}

def main(argv):
    policyJson = json.dumps(policy)
    s3.put_bucket_policy(Bucket=ROOT_S3_DIR, Policy=policyJson)
    print("Updated bucket policy")
    return 0

if __name__ == "__main__":
    main(sys.argv[1:])

```

```

jookai@jookai:~/Desktop/cits5503/lab3$ python3 applypolicy.py
Updated bucket policy

```

2. Inspecting the bucket from the AWS console reveals the updated policy that was added to the bucket.

Bucket policy

The bucket policy, written in JSON, provides access to the objects stored in the bucket. Bucket policies don't apply to objects owned by other accounts. [Learn more](#)



Public access is blocked because Block Public Access settings are turned on for this bucket

To determine which settings are turned on, check your Block Public Access settings for this bucket. [Learn more about using Amazon S3 Block Public Access](#)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowAllS3ActionsInUserFolderForUserOnly",
      "Effect": "Deny",
      "Principal": "*",
      "Action": "s3:*",
      "Resource": [
        "arn:aws:s3:::22489437-cloudstorage/*",
        "arn:aws:s3:::22489437-cloudstorage"
      ],
      "Condition": {
        "StringNotLike": {
          "aws:username": "22489437@student.uwa.edu.au"
        }
      }
    }
  ]
}
```

Attempting to view this bucket from another user with username `22687382@student.uwa.edu.au` shows that they have insufficient permission to list objects in this bucket. This shows that the policy applied has its intended effect.

The screenshot shows the AWS S3 console interface. The top navigation bar includes the AWS logo, a search bar, and a user profile dropdown showing the username `22687382@student.uwa.edu.au`. A red arrow points to this username with the label "Different username". The main content area displays the bucket `22489437-cloudstorage` with tabs for Objects, Properties, Permissions, Metrics, Management, and Access Points. The "Objects" tab is selected, showing a list of objects. A red error box at the bottom of the console states: "Insufficient permissions to list objects. After you or your AWS administrator has updated your permissions to allow the s3:ListBucket action, refresh the page. Learn more about Identity and access management in Amazon S3".

Section 2: AES Encryption using KMS

1. The following code creates a KMS key and attaches an alias (22489437_2) to it. The code also attaches a policy to the key which makes 22489437@student.uwa.edu.au the user and administrator. The policy, represented by the `key_policy` variable which can be found in Lab Sheet 4 is not added to this report.

```
def main(argv):
    kms_client = boto3.client('kms', region_name=REGION)
    # Create Key
    response = kms_client.create_key(
        Description='22489437_key_2',
        KeyUsage='ENCRYPT_DECRYPT',
        Origin='AWS_KMS'
    )
    key_id = response['KeyMetadata']['KeyId']
    print("KMS key id:", key_id)

    # Attach key policy from lab sheet
    kms_client.put_key_policy(
        KeyId=key_id,
        PolicyName='default',
        Policy=json.dumps(key_policy)
    )

    # Create alias for key
    kms_client.create_alias(
        AliasName='alias/22489437_2',
        TargetKeyId=key_id
    )
    print(f"Created alias '22489437' for KMS key with ID: {key_id}")

    return 0

if __name__ == "__main__":
    main(sys.argv[1:])
```

```
jookai@jookai:~/Desktop/cits5503/lab3$ python3 createkey.py
KMS key id: 34005dc0-f101-4523-947c-ec969b05484f
Created alias '22489437' for KMS key with ID: 34005dc0-f101-4523-947c-ec969b05484f
```

2. Inspecting the created key in the KMS console reveals the same key ID as step 1 as well as the attached key policy.

KMS > Customer managed keys > Key ID: 34005dc0-f101-4523-947c-ec969b05484f

34005dc0-f101-4523-947c-ec969b05484f

Key actions Edit

General configuration

Alias 22489437_2	Status Enabled	Creation date Aug 24, 2023 16:13 GMT+8
ARN arn:aws:kms:ap-southeast-2:489389878001:key/34005dc0-f101-4523-947c-ec969b05484f	Description 22489437_key_2	Regionality Single Region

Key policy Cryptographic configuration Tags Key rotation Aliases

Key policy

Edit Switch to default view

```

1 {
2   "Id": "key-consolepolicy-3",
3   "Version": "2012-10-17",
4   "Statement": [
5     {
6       "Sid": "Enable IAM User Permissions",
7       "Effect": "Allow",
8       "Principal": {
9         "AWS": "arn:aws:iam::489389878001:root"
10      },
11      "Action": "kms:*",
12      "Resource": "*"
13    },
14    {
15      "Sid": "Allow access for Key Administrators",
16      "Effect": "Allow",
17      "Principal": {
18        "AWS": "arn:aws:iam::489389878001:user/22489437@student.uwa.edu.au"
19      },
20      "Action": [
21        "kms:Create*",
22        "kms:Describe*",
23        "kms:Enable*",
24        "kms:List*",
25        "kms:Put*",
26        "kms:Update*",
27        "kms:Revoke*",
28        "kms:Disable*",
29        "kms:Get*"

```

3. The following code will encrypt the file and upload it to S3. The code uses the `kms.generate_data_key()` function in combination with the keyID from step 1 to generate a data key that will be used to encrypt the file. This will return a data key in plaintext and ciphertext. The plaintext data key is used to encrypt the file and the encrypted data key is written into the file where it can be retrieved for decryption later. The encrypted file is then uploaded to S3.

```

ROOT_DIR = '.'
ROOT_S3_DIR = '22489437-cloudstorage'
REGION = 'ap-southeast-2'
KEY_ID = '34005dc0-f101-4523-947c-ec969b05484f'
FILE_NAME = 'enc_test.txt'
NUM_BYTES_FOR_LEN = 4
s3 = boto3.client("s3", region_name=REGION)
kms = boto3.client("kms", region_name=REGION)

def encrypt_file():
    # Create Data Key
    try:
        response = kms.generate_data_key(KeyId=KEY_ID, KeySpec='AES_256')
    except ClientError as e:
        logging.error(e)

```

```

    data_key_encrypted, data_key_plaintext = response['CiphertextBlob'],
base64.b64encode(response['Plaintext'])
    print("Data key encrypted:", data_key_encrypted)
    print("Data key plaintext:", data_key_plaintext)

# Read File
try:
    with open(FILE_NAME, 'rb') as file:
        file_contents = file.read()
except IOError as e:
    logging.error(e)
    return False

# Encrypt file
f = Fernet(data_key_plaintext)
file_contents_encrypted = f.encrypt(file_contents)

# Write the encrypted data key and encrypted file contents together
try:
    with open(FILE_NAME + '.encrypted', 'wb') as file_encrypted:

file_encrypted.write(len(data_key_encrypted).to_bytes(NUM_BYTES_FOR_LEN,
byteorder='big'))
        file_encrypted.write(data_key_encrypted)
        file_encrypted.write(file_contents_encrypted)
except IOError as e:
    logging.error(e)
    return False

# Upload the file to S3
file_name = FILE_NAME + '.encrypted'
try:
    s3.upload_file(file_name, ROOT_S3_DIR, file_name, ExtraArgs=
{'ServerSideEncryption': "aws:kms", "SSEKMSKeyId": KEY_ID})
    print(f"Uploaded {file_name} to S3")
except Exception as e:
    print(f"Error uploading {file_name}: {e}")

```

4. The following file named `enc_test.txt` will be the subject of the encryption and decryption for the next step

```

jookai@jookai:~/Desktop/cits5503/Lab3$ cat enc_test.txt
This text will be encrypted as part of lab 4

```

The output of the encryption:

The file can be viewed in the AWS console. Note the server-side encryption setting matching the key generated in step 1.

- The following code will download the file from S3 and decrypt it. The file is downloaded from S3. The encrypted data key which was placed in the file in step 4 will be read from the file and decrypted. The decrypted data key will be used to decrypt the rest of the file.

16 / 18


```
# Write the decrypted file contents
try:
    with open(FILE_NAME + '.decrypted', 'wb') as file_decrypted:
        file_decrypted.write(file_contents_decrypted)
except IOError as e:
    logging.error(e)
    return False

print(f"Decrypted {FILE_NAME}")
```

The output of the decryption:

```
jookai@jookai:~/Desktop/cits5503/lab3$ cat enc_test.txt.decrypted
This text will be encrypted as part of lab 4
jookai@jookai:~/Desktop/cits5503/lab3$
```

Section 3: AES Encryption using local python library pycryptodome

1. The example code in `fileencrypt.py` was used for the local encryption and decryption process following the same steps as above. The file was encrypted, uploaded to S3 then downloaded and decrypted. The code below shows the main program of the program and does not include the `encrypt_file` and `decrypt_file` functions which have not been modified from the example code:

```
s3 = boto3.client("s3", region_name=REGION)
password = 'kitty and the kat'
encrypt_file(password, "enc_test.txt", out_filename="enc_test.txt.enc")
try:
    s3.upload_file("enc_test.txt.enc", ROOT_S3_DIR, "enc_test.txt.enc")
    print(f"Uploaded enc_test.txt.enc to S3")
except Exception as e:
    print(f"Error uploading enc_test.txt.enc: {e}")

s3.download_file(ROOT_S3_DIR, "enc_test.txt.enc", "enc_test.txt.enc")
print(f"Downloaded enc_test.txt.enc from S3")


decrypt_file(password, "enc_test.txt.enc", out_filename="enc_test_decrypted.txt")

print("--- %s seconds ---" % (time.time() - start_time))
```

```
jookai@jookai:~/Desktop/cits5503/lab3$ cat enc_test.txt.enc
-^####$,H_x YIz>ZV=e, /W####Z_uL5Gjookai@jookai:~/Desktop/c
its5503/lab3$
```

```
jookai@jookai:~/Desktop/cits5503/lab3$ cat enc_test_decrypted.txt
This text will be encrypted as part of lab 4
jookai@jookai:~/Desktop/cits5503/lab3$
```

Section 4: Answer the question

1. Both programs were timed in their execution. The program using the AWS KMS encryption took 1 second to run while the local encryption using PyCryptoDome took 0.55 seconds to run. 

```
jookai@jookai:~/Desktop/cits5503/lab3$ python3 fileencrypt.py
Uploaded enc_test.txt.enc to S3
Downloaded enc_test.txt.enc from S3
--- 0.5549328327178955 seconds ---
```