Casey Provitera, cbp21006, Vin Gluck

1 late day used

Q1:

Matching file name:

b9843077c0b8738cc4607cb054c48e527f380e52f9693aff6ab9b09c8d5c2d44

proscenium.exe

Description/Explanation:

The solution to this problem was to brute force it. We typed the command ‘sha256sum’ on the command line for every file within ‘Q1files’ and then check the result hash against ‘Q1hash.txt’ and the one that matched is the hashed file.

Q2:

Matching file:

privative.exe

Description/Explanation:

This problem was basically Q1 but in a script. The biggest hold up was figuring out how to open the files to hash properly but the code simple otherwise.

Q3:

Correctly signed file: liliales.exe

Description/Explanation:

The task being asked in this question was to use a public key to sign files and then check if the files in the directory are correctly signed versions of the originals. The difficulty in this problem is that there are a lot of things that need to be imported to make the code work. Once the correct things are imported the functional code is not that bad. All it does is make a hash variable, make a signer variable and then sign each original file and check it against the supposedly signed files in the directory to find the correctly signed file.

Experiment Results:

With the key lengths of 1024 and 2048 there is not a very large time difference in generating the keys along with hashing since the point of hashing functions is to make something quicker to find and or access. When you get to testing for verification of different length keys is where a noticeable time difference comes into play. This time difference is approximately double for the 2048 length key compared to the 1024 length key. This is because the key needs to be used to decrypt every single character in a file. My experiment simply used different key lengths and recorded the time for the function to complete.

Q4:

Decrypted output: nestled06&

Description/Explanation:

This question wants us to reverse an encryption script and then decrypt a files that has been encrypted. This is not too bad since the encryption files uses a symmetric key meaning all we have to do is call built in functions from the Crypto library, with the same key from the encryption script. The hardest part for this question is realizing that the decryption code can be that simple. For a while we were trying to do it in a much more complicated way.

Q5:

Decrypted output: poecilogale28@

Description/Explanation:

This is almost identical to Q4 but the encryption script given has a lot of weird variable names and some things that are never used so the hard part is filtering through it and finding what the actual code that is running is. Once we filter out the code it is a very similar procedure to Q4. There was significantly less hold up on this question from 4 since the procedures are so similar.

Q6:

Description/Explanation:

The twist for this question is that we now need to write the code that encrypts everything. The system I ended up using for the key generation was simply the crypto library. The main reason I used this was because I had used it in a few of the previous problems and had become familiar with some of the functionality. The key size used was 1024, it was used because it is large enough for there to be very little chance the key is ever broken, but small enough that the code runs fast.

Generating the keys was not hard, it simply involved built-in functions. The interesting part was making the actual ransomware. Since we did not want the key to be symmetrical, we cannot use AES the whole time. We also needed to use PKCS1\_OAEP. But after that the encryption is similar to the decryption from other questions. The shared key was interesting, first we generate 16 random bytes then we encrypt them and move them to a file to be used when running the attacker’s decryption. The attacker’s decryption is easy since it will literally use the private key to decrypt the shared key and that is it. Now the victim’s decryption is slightly different since it used the shared key. We need to decode the shared key and then use it to make a cipher with AES to decrypt the files, and finally unpad the files.

Approval code:

DODSNW4