Lab instructions

Q1:

In this question, you will find in your VM, within the Lab3 directory, a file Q1hash.txt. This file contains the result of the SHA-256 hash function applied to a ('legitimate') program file, encoded in textual form (as a sting). Your task is to identify another file in the Q1files sub-directory, which will have the same hash value (actually, Q1hash.txt is the result of hashing this other file). To do this, use the sha256sum command on the command line (e.g., sha256sum program.exe). As usual, you may want to use man (or even a Google search) to learn a bit about sha256sum.

Q2:

This question is similar to Q1; the main difference is that you should hash using a Python program, Q2.py, which you'll write, instead of using the sha256sum command. Q2.py should identify which of the files in Lab3/Q2files directory, has the same SHA-256 hash as the value of the file Q2hash.txt. Note that the file contains the hash (the hex digest) in bytes – it is not encoded as text. So, you should not try to encode the file contents before passing it to a hashing object from one of the below libraries. There should be exactly one match.

Q3:

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

File Q3pk.pem in directory Lab3 contains the public key used by the legitimate software vendor to sign programs. In sub-directory Q3files you'll find several program files (.exe files), each with the (supposed) signature (.sign files). Note: the signature was created using PKCS#1 v1.5 (RSA) with SHA-256. You may find these two links helpful; check link1 and link2. Write an efficient program, Q3.py, that will find which of these program files is correctly signed. There should be exactly one.

Experiment to get a feeling for the efficiency difference between signatures and hashing, and to better understand why the signature function hashes the file before signing it. Most cryptographic libraries allow you to generate and use keys of different lengths. So,

• compare the times to generate keys of different lengths (e.g, 1024 bits and 2048 bits), and to sign and verify signatures using keys of these different lengths.

• consider the implications, if the signature function you used did not apply hashing, but instead used keys as long as the file being hashed.

Describe your experiments and your findings in your HuskyCT lab report.

Q4:

Look in the Q4files subdirectory of Lab3. This folder contains a file Encrypted4 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. This will allow you to write the corresponding decryption program, D4.py, that will recover the original contents of the plaintext file encrypted by the ransomware. The main reason that you can 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...

Q5:

n this exercise, we have a similar task to the previous question, but a bit more challenging. Look in the Q5files subdirectory 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, and write a new decryption program, D5.py.

Q6:

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 (private) decryption key d, which is supposed to be hard to find, even when given the corresponding (public) encryption key e. That's how most ransomware actually works; as a result, even if we find the ransomware program, and even if we can reverse-engineer it and understandexactly how it works, we can typically only find within it the (public) encryption key e, which isn't sufficient to find the (private) decryption key d.

You can choose the public key cryptosystem and the key size; select a system and corresponding key length that will be reasonably efficient and sufficiently secure.

The question has few steps:

1. Write a key-generation program KG6.py, to generate a keypair of a public key e and a private key d. Save them in files e.key and d.key in sub-directory Solutions.

2. Write the ransomware program R6.py, using the public key e you generated. This program should:

    a. Have the public key e hard-coded, i.e., it is not read it from a file.

    b. Generate a random shared key k.

    c. Encrypt k using the public key encryption with key e.key; output the result as file EncryptedSharedKey. This file should be submitted with the ransom payment to the attacker, allowing the attacker to provide the unlocking file (see below).

    d. Search the folder in which it runs and encrypt, using shared-key encryption with key k, all files in this folder with extension .txt. Specifically, say the folder contains some file, say example.txt. Then R6.py should replace example.txt with file example.txt.encrypted. The example.txt.encrypted file will be the encrypted version.

3. Write the attacker's decryption program, AD6.py. This program will receive, as a command line argument, the EncryptedSharedKey filename, and output the

corresponding, decrypted shared key k as the file DecryptedSharedKey. AD6.py will use the private decryption key d, which should be hard-coded into AD6.py.

4. Write the victim's decryption program D6.py. This program will receive, as a command line argument, the DecryptedSharedKey filename (the file that contains the decryption key k sent by the attacker). It should use this key to decrypt all the encrypted files in the current directory, i.e., recover example.txt from example.txt.encrypted.