Cryptography, Malware, and Ransomware

Created: Fall 2022, Updated: Spring 2024

Monday sections submit by 3/17

Wednesday sections submit by 3/19

See instructions for all labs in the Lab 0 document.

In this lab, we will learn more about **malware**; specifically, we will learn some of the main *defenses against malware*, as well as study the design of *ransomware*, one of the most problematic types of malware. Both topics – *defense against malware* and *design of ransomware* – make extensive use of *Cryptography,* so, this will also provide a way for us to introduce a bit more of cryptography.

Cryptography 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 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.*

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 to receive the decryption key. We will explore the use of *public key cryptosystem, shared key cryptosystem* and *obfuscation.*

QUESTION 1 (10 POINTS):

In this question we will learn the use of *cryptographic hash functions* to ensure the integrity of software downloads, i.e., to ensure the download is of the intended, authentic software, and not of a malware impersonating 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*. This application relies on the **collision-resistance** property of cryptographic hash functions. Basically, a hash function *h* is **collision-resistant** if, given the digest *hash(m)* of any input string *m*, it is infeasible to find a *different string m'≠m,* which hashes to the same digest: *h(m')=h(m).*

The collision resistance property is often used to ensure **integrity** – and, in particular, the integrity of software downloads. By ensuring integrity, we mean that we can be sure that the software has not been altered since it was originally published by its authors. Software is often made available via online 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).*
Note that other applications of hash functions rely on other properties, for example, in the passwords lab we relied on the **one-way** property.

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. As usual, you may want to use *man* (or even a Google search) to learn a bit about *sha256sum*.

Submit to autograder and to HuskyCT: the name of the matching file.

QUESTION 2 (15 POINTS):

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.* Note that the file contains the hash in bytes, not encoded as text. There should be exactly one match.

Your Python program may use the SHA-256 functionality from either:

1. the `PyCryptodome` library, which should be installed on the VM. You can import it into your Python program with the `import Crypto` statement.
2. the `cryptography` library.

Use one or more of these crypto libraries for all relevant questions in this lab.

Submit to autograder: the name of the matching file.
Submit to HuskyCT: your program, *Q2.py*, as a separate file, and the name of the matching file.

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***, only *integrity* (verify the software against a known hash value). By ensuring authenticity, we mean that we can be sure that the software came from the stated authors. Fortunately, cryptography also provides tools to ensure authenticity; the most important of these would be *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.* Such schemes, that use a pair of matching keys, one public and one private, are called *public key cryptographic schemes.*

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 used keys as long as the file being hashed. Describe your experiments and results in your HuskyCT lab report.

Submit to autograder: the name of the correctly signed program file.
Submit to HuskyCT: your program (as a file Q3.py), the name of the correctly signed file, the description of your experiments and results.

QUESTION 4 (15 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 *Q4files* subdirectory of *Lab3*. This folder contains a file *Encrypted* 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 would 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…

Submit to autograder: the results of decrypting the *Encrypted* file.
Submit to HuskyCT: your program (as a file, D4.py) and the results of decrypting the *Encrypted* file.

<u>**Note:** Before encryption, the plaintext was *padded* so that its length would be a multiple number of `blocks', and the padding must be removed after applying the block-cipher's decryption function.</u>

QUESTION 5 (20 POINTS):

In 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.*

Submit to autograder: the results of decrypting the *Encrypted* file.
Submit to HuskyCT: your program (as a file, D5.py) and the results of decrypting the *Encrypted* file.

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 (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 understand exactly 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 parameter the *EncryptedSharedKey* file, and output, to standard output, the corresponding shared key *k*. *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 in standard input 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.*

Show your program and your tests of it to the TA for review and approval!

Submit to autograder: the "approval code" from your TA.

Submit to HuskyCT: your programs (KG6.py, R6.py, AD6.py, D6.py), as separate files, and the key files *e.key* and *d.key.* In your report, identify and justify your choice of public key cryptosystem and key size, and include screen recording showing how you test your program.