|  |  |  |
| --- | --- | --- |
|  |  | **Cybersecurity Lab, CSE 3140** |
|  |  | **Fall 2022, Prof. Amir Herzberg** |
| **Cryptography, Malware and Ransomware**  **Submit by: noon, 10/18(sections 003, 005)**  **Noon, 10/23 (sections 001, 002, 004)**  **Noon, 10/17 (Stamford section)** | | |
| **Section # :** | | |
| **Team #:** | | |
| **Names:**  **Net-IDs:** | | |

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

As in all labs, each group’s VM is connected on a dedicated Internet Protocol address (IP address), of the form 172.16.50.x, where 172.16 identifies a block of IP addresses allocated to UConn, 50 identifies (part of) the Altschuler lab dedicated network, and, finally, x is a number, between 0 and 255, which identifies the VM for your group. If you are in team t of section s, then use x=s\*20+t. For example, team 3 of section 2 will use IP 172.16.50.43. The VMs are always initialized with user *cse* and default password *cse3140*; change the password when you begin the lab.

**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 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*. 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. 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).*

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

**Submit in HuskyCT and write in file *Q1a* in directory *Lab3/Solutions*:** 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, if any, 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.

Your program may use the SHA-256 from (1) the [PyCryptodome](https://pycryptodome.readthedocs.io/en/latest/src/hash/hash.html) library, which should bes installed on the VM, (2) the *hashlib* library, or (3) the *cryptography* library . Use one or more of these crypto libraries for all relevant questions in this lab.

**Submit in HuskyCT:** your program, *Q2.py*, and the name of the matching file.

**Save in directory *Lab3/Solutions:*** the program Q2.py, and the name of the matching file, as the contents of the file *Q2a.*

**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). 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 use the RSA signature algorithm and the SHA-256 hash function.

File *Q3pk.pem* in directory *Lab3* the public key used by the legitimate software vendor to sign programs. In sub-directory *Q3files* you’ll find several program files, each with the (supposed) signature. Note: the signature was created using [PKCS#1 v1.5 (RSA)](https://pycryptodome.readthedocs.io/en/latest/src/signature/pkcs1_v1_5.html) with SHA-256. You may find these two links helpful; check [link1](https://www.youtube.com/watch?v=b2pj0yDhDp4) and [link2](https://pycryptodome.readthedocs.io/en/latest/src/signature/signature.html).

Write an efficient program, *Q3.py*, that will find which of these files is correctly signed. There will be at least one, but there could be more.

**Submit in HuskyCT:** your program and its output (names of well-signed program file(s)).

**Save in directory *Lab3/Solutions:*** the program Q3.py, and a file *Q3a*, containing the name(s) of the matching file(s) (one name in each line)*.*

***Extra:***we recommend you experiment a bit 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.

**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 *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 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 in HuskyCT:** your program (D4.py) and the results of decrypting the *Encrypted* file.

**Save in directory *Lab3/Solutions:*** the program D4.py, and a file *Q4a*, containing the results of decrypting the *Encrypted* file*.*

**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 *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 in HuskyCT:** your program (D5.py) and the results of decrypting the *Encrypted* file.

**Save in directory *Lab3/Solutions:*** the program D5.py, and a file *Q5a*, containing 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 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*.

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

The question has few parts (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 search the folder in which it runs, and encrypt 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 two files, *example.txt.encrypted* and *example.txt.note.* The example.txt.encrypted will be the encrypted version, and *example.txt.note* will contain a `ransom note’; be creative with the text in the note, but you should include a unique, random identifier which should be given to the attacker together with the payment, to allow the attacker to send the decryption key. A different decryption key should be required for every file, so the identifier should be unique too! Save the identifier too, in file *example.txt.ID*. The program should have public key *e* ‘burned-into’ it, i.e., it should not read it from a file.
3. Write the attacker’s decryption program, *AD6.py.* This program will receive as a parameter the identifier, i.e., the value saved in the identifier file (e.g., in *example.txt.ID).* and output, to standard output, the corresponding decryption key. This program will make use of the private decryption key *d,* again `burnt-in’ into the program
4. Write the victim’s decryption program *D6.py*. This program will receive, as parameter,the name of an encrypted file, e.g, *example.txt.encrypted*, and receive in standard input the decryption key sent by the attacker. It should output the original file (with the original name), e.g. *example.txt*.

**Submit in HuskyCT and save in directory *Lab3/Solutions*:** your programs (KG6.py, R6.py, AD6.py, D6.py), and the key files *e.key* and *d.key.*