Lab - Hash Functions

In this guide we will develop programs that use cryptographic methods, relying in the Python3 Cryptography module. The module can be installed using the typical package management methods (e.g, `apt install python3-cryptography`), or using the `pip3` tool (e.g. `pip3 install cryptography`).

It will be useful to visualize and edit files in binary format. For that purpose, if you are using Linux, you may install `GHex` or `hexedit` from the repositories.

We will be exploring the low level interface of the `python cryptography` library, for educational purposes. If you plan to use this library in real world application, stay with the Fernet interface.

As the documentation clearly states:

```
This is a “Hazardous Materials” module. You should ONLY use it if you’re 100% absolutely sure that you know what you’re doing because this module is full of land mines, dragons, and dinosaurs with laser guns.
```

Cryptographic hash functions

Create a program to produce the hash of the contents of a file. The user must provide the following input to the program: (i) the name of the file with the data to create the hash, and (ii) the name of the cryptographic hash function to use (`MD5`, `SHA256`, `SHA384`, `SHA512` and `Blake2`. The program must print the hash in the screen, in hexadecimal format.

Questions:

- How can you obtain the original text from the resulting digest?

Avalanche effect

A very important requirement for cryptographic hash functions is that a small change in a text must produce a completely different hash – avalanche effect. In this exercise we are going to verify this requirement.

Select or create a file to calculate the hash. Using your program to obtain the hash of the file you just selected and save it in a file. Repeat the operation to obtain the hash of the same file, but using different cryptographic hash functions, and save the result hash in separated files.

Change one single bit of data in the source file you used above. Using this altered source file, obtain the hash produced by each of the cryptographic functions used in the previous step, and save them in separate files.

Compare the similarity of each pair of hashes produced by the same cryptographic hash function, and using source files differing only in a single bit.

Statistical analysis of avalanche effect

Based on the program you developed to calculate cryptographic hashes, create a new program to calculate the statistical distribution of the differences in the hashes of a set of messages that differ in one single bit from an original message. The program must receive from the user the following input: (i) the name of the file with the source message and (ii) the number of messages (`N`), differing one bit from the original message, to calculate the hash. The creation of single bit altered messages must use a random number generator to calculate the position of the bit to alter.

After the calculation of the hashes of all the `N` one-bit altered messages, evaluate the difference between each of these hashes and the hash of the original message, in terms of number of bits (Hamming distance, in bits). Comment the distribution of the differences that you obtained.

Hint: Use the `XOR` operation to detect the number of different bits between between two hashes (number of bits with the value 1 in the result of `XOR` operation).

Symmetric key generation

Before a cipher is used, it is required the generation of proper arguments. These arguments are the key, the cipher mode, and potentially the IV. The cipher mode is chosen at design time, and the `IV` should always be a large random number (size similar to the block size or key) that is never repeated. This lab will discuss the IVs in the next sections.

The key can be obtained from good sources of random numbers, or generated from other primitive material such as a password. When choosing the last source (a password), it is imperative to transform the user text into a key of the correct complexity. While there are many methods, we will consider the Password Based Key Derivation Function 2 (PBKDF2), which takes a key, a random value named salt, a digest algorithm, and a number of iterations (you should

use several thousands). The algorithm will iterate the digest algorithm in a chain starting in the concatenation of the salt and key, for the specified number of iterations. Using `SHA2`, the result is at least 256 bits, which can be used as a key.

Exercise: Construct a small function that generates and returns a symmetric key from a password. The algorithm name for the symmetric key and the file name should be provided as an argument. For testing purposes, the algorithm should be provided by the user as a argument and you can use the following algorithms: `3DES`, `AES128` and `ChaCha20`.

To provide the password, check the `getpass` Python3 module.

2021

PREVIOUS
Lab - Asymmetric Cryptography

NEXT
Lab - SSH and OTP authentication

Last updated on 1 Oct 2021

Related

- Project 2 - Authentication
- Assignment 1 - Vulnerable App
- Assignment 2 - Application Exploitation
- Assignment 3 - Binary Vulnerabilities
- Lab - Asymmetric Cryptography