Public Key Steganography Tool

Description

A Python based tool for encrypting data using RSA/AES and storing it in a an animated Gif file. The tool also handles the generation and storage of RSA keys using a login system in order to keep operation as simple as possible. All functions are acessed using a simple command line interface.

External libraries and their documentation

PyCryptodome: https://pycryptodome.readthedocs.io/en/latest/#
Pillow: https://pillow.readthedocs.io/en/stable/

Operation

To start the program, run **main.py**. The tool is operated with a command line interface below is a bried description of all commands.

To encrypt files for users on other devices, their public key (user_public.der) must be obtained beforehand and placed in the program directory.

>help

Prints a list of all commands.

>register

Asks the user to input a username and a passphrase. If the user doesn't exist already, creates a new entry in 'users.csv' with the username, password hash and salt. Also generates matching .der files for public and private RSA keys. Private key is encrypted using the user's passphrase.

>login

Asks the user to input a username and a passphrase. Checks that the user exists and validates the passphrase against the hash stored int 'users.csv'. If successful, login is considered valid and the passphrase is kept in memory.

>**encrypt**

Asks the user for the username of the recipient, a data file and a Gif file. If the files are valid and the recipient exists, the data is encrypted using the recipient's public key file (recipient_public.der) and encoded in the image using a XOR operation.

>**decrypt**

Asks the user for an encoded Gif, the original Gif and an output file name. In addition to checking the files are valid, the encoded Gif contains an MD5 checksum used to verify that the original Gif

matches the on used for encryption. If all conditions are met, the data is decoded from the image, decrypted and stored in the specified output file.

Since private keys needed for decryption are encrypted themselves, the user must be logged in to perform this operation.

>quit

Exits the program.

Structure

The program is designed to be modular, with access management, encryption, encoding, UI and generic file handling separated in their own components. All of these components are contained within their own .py files.

All other components are accessed throuth the UI module, which contains the main program loop. Below is a graph of the structure and a bried description of each module.

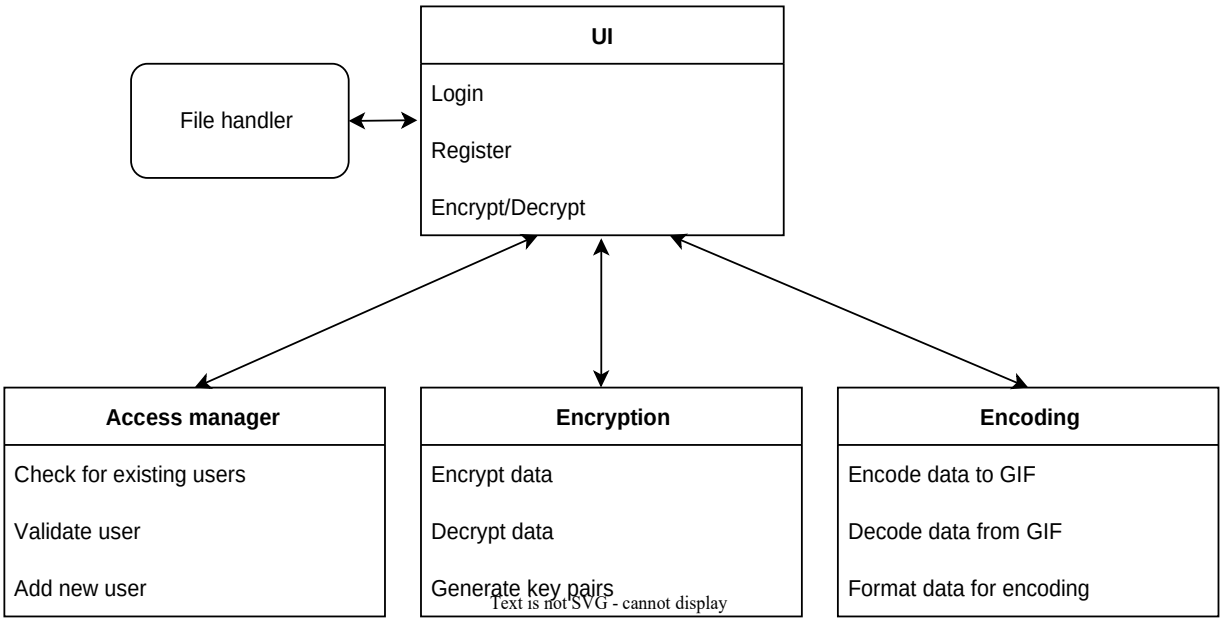

Figure 1: The modular components and their interactions

UI

As stated above, this is the main program loop. It parses commands and calls the other modules to perform the functions of the program.

File handler

This small module is only used for the secure loading of data from input files. Interaction with more modules was initally planned, but proved impractical.

Access manager

Responsible for the login system, using data stored in 'users.csv'. Performs password hashing and salt generation using SHA256 and a cryptographic random number generation.

Encryption

Handles the encryption and decryption of data. Primary encryption method is AES in EAX mode, followed by encrypting the AES key with RSA using the PKCS1_OAEP cipher.

RSA keys are also generated here from a username and a passphrase. UI is responsible for synchronizing this with registration in the Access manager.

Encoding

Encodes and decodes data to and from Gif files using Pillow. Also includes a formatting function to map the data to the desired amount of bits in each pixel and a reverse function to return it to normal.

Information required for decoding is passed as part of the encoded data, in a custom header format.

Secure programming: Standards

The first part of our approach to secure programming is to use secure library implementations of standard cryptographic function where possible. The primary tool we used was the PyCryptodome library, in particular the following components:

Crypto.Random

A cryptographically secure random number generator. Used for generating AES keys and salt for hashing.

Crypto.Hash.SHA256

A fast but secure 256-bit hashing function. While it's not necessarily the state of the art method for password protection, with sufficient (256-bit) random salt it should be more than secure enough for our purposes.

Crypto.PublicKey.AES

Used to generate and store AES keys in a secure manner. We use the binary .der format, as the documentation suggested the encryption used for .pem files is outdated.

Crypto.Cipher.PKCS1_OAEP

Standard implementation of an RSA cipher with automatic padding. Used to encrypt AES keys.

Crypto.Cipher.AES

Standard implementation of an AES cipher with several possible modes of operation. We chose

EAX (encrypt-then-authenticate-then-translate), which is relatively modern and uses a tag/nonce system for authentication and integrity validation.

PIL.Image

While not a security-related function, using an established method for reading and modifying Gif files allows us to avoid a large amount of potential bugs and crashes from a custom implementation.

Secure programming: Other considerations and testing

File handling

While we don't use any temporary files, file handling is required in all modules to some extent. For this reason, it is necessary to handle things like invalid file types and missing files without crashing and avoid corrupting files as much as possible.

The basic tools for this are try-except blocks and with-statements. The with-statement is designed to close a file gracefully even in the event of an exception, and the try-except block allows us to catch the exception and react to it.

These were tested with empty and invalid files and filenames, as well as incorrect file types where applicable. We were able to find and fix some structural mistakes, as well as sitations where the output of the except block wasn't handled correctly in the next function over.

We fixed the immediate issues, but also added more try-except blocks to other modules to be able to avoid crashes even if the initial checks failed.

UI

The UI doesn't handle many security related tasks directly, but we made sure to test various inputs in all situations to discover potential issues in communication with other modules. Inputs are always handled as strings and modules impose their own specific restrictions, so no injection-type attacks should be possible.

None of our testing revealed any issue in the UI itself, other than few typos in If-statements.

Access manager

Testing here mostly involved verifying that saved hashes and salt values could be read and verified correctly. We foud and fixed some issues related to how a bytes object was converted to a string and then back again in order to give the hashing function a correct input type.

Additionally, we tested that the salting was behaving as expected using the same password with several users.

Encryption

The encryption module was tested by running trough random data with valid and invalid keys and passphrases. Through testing the UI we discovered that the encryption module fails with empty

input data.

To fix this we implemented a check for empty data in the UI, as well as try-except blocks in the encryption module as a secondary measure. These also catch any other encryption-related error, such as missing or wrong keys, which also caused crashes in testing.

Encoding

This module required by far the most testing. First, we tested differing amounts of random data over a large number of iteration to make sure that what goes in also comes out. This also included printing out header information each time, which allowed as to realiably verify that at least something made it through.

We discovered many bugs, such as Pillow not supporting saving multiple frames directly, saving the original frames in an array causing them to revert changes after searching to the next frame on the original Image object, as well as many mistakes in our calculations related to how much data should go in each individual frame.

Once we were able test encryption->encoding->decoding->decryption all in one go, the EAX mode of operation was also helpful due to its verification capabilities, giving us quick integrity reading for the whole chain.

One more issue was discovered during UI testing, as giving the decryption function the wrong 'original' image could cause unpredictable behaviour. Rather than designing anything more complicated and risk more bugs, we decided to encode an MD5 checksum of the original image with the data, allowing us to be fairly confident that both images are the same.

We decided to give the user a choice to ignore this check, with a warning that the program may crash. Some of those crashes could be avoided, but doing so would've taken more time than we had left, and this isn't the regular behaviour of the program anyway.

TODO and other notes

While the program is functional and relatively secure, there are a few features and potential issues we didn't have enough time to work on.

>bitdepth=n -command

This was removed due to incompatibility with some other late fixes. Currently the bit depth (the amount of bits in each pixel to use for data storage) is hardcoded, with a default value of 2. Other tested safe values include 1, 4 and 6.

mode switch

Currently, our system encodes data with a XOR function for slightly increased security. We could also provide a mode where bits are simply replaced with data, allowing decoding without a copy of the original picture.

authenticating public keys

Because public keys are designed to be imported manually, there is no built in process for validating

them. This means a user must be careful when obtaining key files, and they could even be replaced afterwards if an attacker had access to the user's PC somehow.

overwriting public keys

Since external and self-generated public keys are stored in the same location, a generated public key could overwrite an imported key of the same name. This isn't a security issue as such, but would potentially prevent a recipient from opening a file.

performance improvements

By far the slowest part of the system is the bit manipulation required to format the data for encoding. Currently the program can take up to a few minutes to encode 10MB of data depending on the bit depth, and easily 80-90% of this is formatting. Unfortunately there may not be an easy solution using Python, implementing this part in C could be necessary if much larger files are necessary.