Encryption using AES

& self created encryption algorithm

Palesa Mabula 37124315
Britney Soya 40921050
Lungile Sifumba 40913295
Simphiwe Nhlapo 42304954
Thomas Steenbock 49793594

An encryption project presented for the
module CMPG 215

North-West University
South Africa
May 15, 2023

Contents

1 Introduction

In order to use our app you need to install different packages and libraries. Furthermore, we provide an explanation for the methods we use like hashing 1.2 and the xor operation 2.3.

1.1 Prerequisite

Following libraries are to install in order to use the app:

- `pip install customtkinter`
- `pip install pycryptodome`

We used customtkinter to create our GUI [Sch22][Mai][Pyt] and pycryptodome for AES encryption. [Bas22][Vol19][Leg22]

1.2 Hashing

We use hashing to safely store users passwords. We need to store the credentials provided by the user to verify that they provide the correct credentials whenever they want to access the application. We hash the password provided by the user to make the storage as safe as possible. If we saved the password in cleartext a possible attacker who get access to our database would be able to steal the passwords of all users and access each user account. This problem is compounded by the fact that many users reuse or use variations of a single password, potentially allowing the attacker to access other services different from the one being compromised. A more secure way to store a password is to transform it into data that cannot be converted back to the original password. That is hashing.

In cryptography, a hash function is a mathematical algorithm that maps data of any size to a bit string of a fixed size. This fixed-size string function output is known as the hash, the hash value or the message digest. The hash functions used in cryptography are easy and practical to compute the hash, but di cult or impossible to regenerate the original input if only the hash value is known and it's di cult to create an initial input that would match a specific desired output (hash). Thus, in contrast to encryption, hashing is a one-way mechanism. The data that is hashed cannot be practically reverted.

Another critical property that makes hash functions suitable for password storage is that they are deterministic. A deterministic function is a function that given the same input always produces the same output. instead of storing the password in cleartext, we hash the password and store the username and hash pair in the database table. When the user logs in, we hash the password sent and compare it to the hash connected with the provided username. If the hashed password and the stored hash match, we have a valid login. We never store the cleartext password in the process, we hash it and then forget it. The password hash doesn't need to be encrypted at rest.

Hash functions have a limitation. if an attacker breaks into the server and steals the password hashes, all that the attacker can see is random-looking data that can't be reversed to plaintext due to the architecture of hash functions. Since hash functions are deterministic, if a couple of users were to use the same password, their hash would be identical. If a significant amount of people are mapped to the same hash that could be an indicator that the hash represents a commonly used password and allow the attacker to significantly narrow down the number of passwords to use to break in by brute force. Additionally, through a rainbow table attack, an attacker can use a large database of precomputed hash chains to find the input of stolen password hashes. We can mitigate a rainbow table attack by boosting hashing with a procedure that adds unique random data to each input at the moment they are stored. This practice is known as adding salt to a hash and it produces salted password hashes. With a salt, the hash is not based on the value of the password alone. The input is made up of the password plus the salt. When the attacker gets a hold of the salt, the rainbow table now needs to be re-computed, which ideally would take a very long time. [Gal19]

2 Our self-created encryption algorithm

2.1 Encryption

We start our encryption with saving the name of the file we want to encrypt based on the file path the user enters. In order to never save the user's password, we hash it and save the hash instead. For that purpose, we use salted hashing by adding some random bytes, the salt, to the password entered by the user after encoding it to bytes. As result we obtain the hashed password. Salted hashing ensures different hash values even if multiple users enter identical passwords in order to prevent that a possible attacker with access to our database has any indicator of commonly used passwords what would be the case as various user got identical hash values. There is no need to keep the salt secret but it should be random and chosen for each key derivation. The salt has a fix length of 16 bytes. This is going to be important as we write the salt to the file. The salt is needed for decryption. We can simply read in the salt by reading the fix length of 16 bytes. Finally we receive the hash value in a hexadecimal representation. The hash has a length of 32 bytes. The sha256 (secure hash algorithm 256) gets it name by the calculation of

$$32 \text{ bytes} * \frac{8 \text{ bits}}{1 \text{ byte}} = 256 \text{ bits}$$

```
def encrypt(self, path: str, pw: str):
    file = Path(path).name

    salt = get_random_bytes(16)
    pwrd = bytes(pw, 'utf-8')
    key = salt + pwrd
    h = sha256()
    h.update(key)
    hash = h.hexdigest()
```

To be able to observe the hash value in its hexadecimal representation, we safe it to a file in our output folder. It would also be possible to write the hash to the encrypted output file in the way we do it with the salt. We create and open the file for writing with `open(),"w"` function and write the hash value to file. It's not necessary to use the *binary mode* as we want the hash as it is. We keep the original name of the file and add `_hash.bin` as suffix. Eventually we close the the file.

```
file_out = open(f"output/{file}_hash.bin", "w")
file_out.write(hash)
file_out.close()
```

The next step is to open the input and output file. The input file reads in our unencrypted file. We use again the `open(),"rb"` function but this time in *binary mode*. The data type of the file does not matter when using the *binary mode*. Every file is just bits. Afterwards, we read it in as bytes using the `.read()` function. The output file does not exist at that point. We create and open the output file in *write mode* using `open(),"wb"` and take the original file name added with `.encrypted` as suffix.

```
input_file = open(path, 'rb').read()
output_file = open(f'output/{file}.encrypted', 'wb')
```

All preparation are done. The actual encryption uses XOR 2.3. The encryption uses the unsalted user password in bytes. We call our xor encryption and give the file to encrypt and the password as arguments and write the encrypted data (cipher) to our opened output file together with our salt. The salt is needed for decryption in order to ensure the same user gains access by producing the same hash value and compare it to the saved one created when encrypting. Eventually we close the file.

```
cipher = xorOperation(input_file, pwrd)
output_file.write(salt)
output_file.write(cipher)
output_file.close()
```

Finally, we delete the original file. As we check all exceptions when clicking the "Encrypt" button in our GUI we can just delete the original file.

```
os.remove(path)
```

The following function is executed by clicking the "Encrypt" button in our GUI. We assign the input entered in the entry widget to the variable `path` as well as the password entered to the variable `pw`. We check various exceptions like:

- Was a path specified?
- Does a file exist under the specified path?
- Was a password specified?
- Was a encryption method chosen?

```
def button_encrypt(self):
    path = self.entry.get()
    pw = self.entry2.get()
    if path == "":
        self.labelChange("Please enter file location")
    elif not Path(path).is_file():
        self.labelChange("File does not exist")
    elif pw == "":
        self.labelChange("Please enter password")
    elif self.rButtonVar.get() == 0:
        self.labelChange("Please choose method")
    elif self.rButtonVar.get() == 1:
        AESencryption.Encryption.encrypt(self, path, pw)
        self.labelChange("Encryption with AES successful")
    else:
        OwnAlgo.Encryption.encrypt(self,path, pw)
        self.labelChange("Encryption successful. File removed")
```

If everything was successful we encrypt the entered file with the encryption method based on the chosen method by calling the respective encryption function with path and password as arguments and display a confirmation of the successful encryption.

2.2 Decryption

We start our decryption same like encryption by saving the name of the file we want to decrypt based on the file path the user enters. For this purpose, we use the `.stem` function what receives the final path component as string, minus its last suffix and assign it to the variable *file*.

```
def decrypt(self, path: str, pw: str):
    file = Path(path).stem
```

Afterwards we open the hash file which we need to check whether the password the user enters is valid. Therefore we use `open(),"r"`. Due to the fact we did not write the hash in *binary mode* there is no need to read in the hash in same. Additionally we use the `.read()` in order to obtain a string. Eventually, we close the file.

```
file_in = open(f"output/{file}_hash.bin", "r")
hash_from_file = file_in.read()
file_in.close()
```

After we have the hash in memory we read in the encrypted file. We wrote the salt with a fix length of 16 bytes to the ciphertext. Now we can read in the salt by reading 16 bytes with `.read(16)` because we know the length of the salt. Afterwards, we read in the rest as ciphertext using the same method without an argument. Both salt and ciphertext are available as bytes. We assign both to variables and close the file.

```
file_in = open(path, "rb")
salt = file_in.read(16)
ciphertext = file_in.read()
file_in.close()
```

The following procedure is basically a repetition of what we do in our encryption to obtain the hash value. We convert the entered user password to bytes and add the salt we just read in from the encrypted file to produce the same hash. As we used multiple steps in our encryption function to produce the hash, we now put it in one line of code. The result is the identical hash value in hexadecimal representation.

```
pwrd = bytes(pw, 'utf-8')
key = salt + pwrd
hash = hashlib.sha256(key).hexdigest()
```

Due to the fact we do not safe the user's password directly but the salted hash value, we need to check whether our saved hash and our newly created hash based on user's input are equal. If so, the password entered by the user is correct and we can proceed. Otherwise, we use our `labelChange` method and tell the user that the hashes are not identical.

```
if (not hash == hash_from_file):
    App.App.labelChange(self, 'Hashes are not identical')
```

In the case the hashes are equal, we presume by opening an output file for the unencrypted file using `open(),'wb'` to write it in *binary mode*. Like we did for encryption, we call our `xorOperation` function, commit the encrypted file and the password as arguments and assign the result to the variable *plain*. Furthermore, we write the unencrypted data to our output file and close it. Finally we inform the user that the decryption was successful.

```
else:
    output_file = open(f'output/{file}', 'wb')
    plain = xorOperation(ciphertext, pwrd)
    output_file.write(plain)
    output_file.close()
    App.App.labelChange(self, "Decryption with successful")
```

The following function gets executed when the user presses the "Decrypt" button. Like we do for the encrypt button, we check various exceptions 2.1 and eventually call the applicable encryption method.

```
def button_decrypt(self):
    path = self.entry.get()
    pw = self.entry2.get()
    if path == "":
        self.labelChange("Please enter file location")
    elif not Path(path).is_file():
        self.labelChange("File does not exist")
    elif pw == "":
        self.labelChange("Please enter password")
    elif self.rButtonVar.get() == 0:
        self.labelChange("Please choose method")
    elif self.rButtonVar.get() == 1:
        AESencryption.Decryption.decrypt(self, path, pw)
    else:
        OwnAlgo.Decryption.decrypt(self, path, pw)
```

2.3 Exclusive OR encryption algorithm

In the following we use the shortcut XOR for exclusive OR.
Our encryption is a symmetric encryption. That means we use the same key for encryption and decryption. In order to encrypt our file we use XOR. We read in the file to encrypt as well as our key as binary. The following table of a XOR gate explains how XOR works:

Input		**Output**
A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

Table 1: XOR gate
[Wik23a]

An XOR statement is true if and only if one is true and the other is false. Let's have a look to an example how XOR works with letters:

```
a = 3
b = 5
print(a^b)
```

The output is 6. Why is that? The 3 in binary is `0b011` and the 5 is `0b101`. We use XOR bit by bit of the binary representation of the 3 and the 5. According to our table 1 the output is `0b110` which is 6 in decimal. [Kha23] Our XOR encryption works the same. Our file and key are in binary representation. We use XOR bit by bit of file and key and put the result in a list. In assumption that the file is longer than the key, we loop over the key with its own length using modulo operation. Finally, we return the resulting list as bytes. [Gee20]

```
list = []
for i in range(len(file)):
    list.append(file[i] ^ key[i % len(key)])
return bytes(list)
```

3 Usage of the App

You can choose the location of the input file but the output location is fix. It is the *output* directory. We add `.encrypted` as suffix to the encrypted file as well as `_hash.bin` as suffix to the hash file. The hash file gets saved in the *output* directory as well and got automatically read in for decryption. So, it has to be stored in the *output* directory. Please do not change the default names of the files. Possible exception get printed to the label on our GUI to inform the user.

4 AES encryption

4.1 Why we chose the encryption method

Whether it be sending and receiving emails, transferring files or simply browsing the web, data transfer exists everywhere in our technological world and cybercriminals are trying to access that data at every turn. AES encryption, or advanced encryption standard, is a symmetric block cipher used to encrypt sensitive data. With equal parts security and speed, AES has become a security standard for users and applications that need easy-to-use encryption. This method was first conceptualized in 1997 when the National Institute of Standards and Technology (NIST) became vulnerable to brute force attacks and needed a stronger encryption method. Vincent Rijmen and Joan Daemen developed AES in 1998. AES has been the encryption standard for the NIST since its adoption in 2002. AES is not the safest encryption method but applicable for the most situations in regards of usability. It's also one of the leading encryption methods on the market and trusted by the National Institute of Standards and Technology since 2002. Additionally, it is free, open source, safe and used for different purposes: [Med23]

- **VPNs**: to provide secure and private online browsing by protecting user data against leaks and cyberattacks
- **Password manager**: to safely store login credentials under a single master key
- **Wi-Fi**: typically uses many encryption methods such as WPA2, and AES can often be found in these connections as well
- **Mobile apps**: Any app that involves messaging or photo sharing typically utilizes AES

4.2 What are the advantages and disadvantages of this algorithm

AES is the preferred encryption method because it excels in many key performance metrics. Some benefits are:

- **Security**: Even the lowest level of AES encryption, AES-128, would take an estimated 1 billion billion years to crack if using a brute force method
- **Cost**: AES encryption is available for free, as it was originally developed to be released on a royalty-free basis

- **Ease of use**: The AES algorithm is easy to implement across a multitude of applications, and is known for its simplicity and ability to conform across hardware and software platforms
- **Speed**: Compared to other encryption methods, AES is known for its speed, boasting faster encryption and decryption times than other methods

[Med23] AES is implemented in both hardware and software. That makes it a robust security protocol. It uses higher key sizes such as 128, 192 and 256 bits forencryption. Hence it makes AES algorithm more robust against hacking.

Nevertheless, AES has some disadvantages. The security of AES depends like other symmetric algorithm on the key. The owner of the key can decrypt the cipher. Hence, the key is the biggest vulnerability. Furthermore, AES uses a too simple algebraic structure. Even AES is safe it is possible to reduce the key length using algebraic approaches. A key of 128 bit length could be diminished to a size that makes it vulnerable for brute force attacks. Moreover, every block is always encrypted in the same way. On programmers' side, AES is hard to implement with software. In sum, the advantages predominate the disadvantages. AES is a very secure encryption what is the reason we used it.

5 Compare own algorithm with others

XOR gets used inside most of the encryption algorithms and block ciphers like AES. After we have read in the file to encrypt we can work with the binary representation of a specific length. If we encrypted our file with a random bytes string of the same length as our file we would have an one-time pad encryption. The first U.S. patent issued to *Gilbert Vernam* on July 22, 1919, for the encryption of a one-time pad used the XOR operation. [Wik23b] The one-time pad encryption is provably unbreakable. Nevertheless, we have to abide some rules to have a genuine one-time pad encryption. The key must:

- have the same length as our data
- be truly random
- be entirely sampled from a non-algorithmic, chaotic source such as a hardware random number generator
- be patternless in meaning of no loops or other pattern
- never be reused in whole or in part
- be kept completely secret by the communicating parties

Of course, in most circumstances, using such long keys would be extremely impractical. Additionally, it is highly inconvenient to always reproduce a new key and thinking about way to commit the key to the other communicating party. Our encryption is not an one-time pad. Our key does not have the same length as our data. Instead, we loop over our key as long as necessary. That way we have a pattern in our key what makes our encryption less secure. It is not necessary for our encryption that key and file need to have the same length. That makes encryption preparations like padding the key in order to ensure same length as file redundant. Padding is an obligated step for other encryption methods or modes like the CBC Mode in AES. Like other symmetric encryptions, our most crucial part of our encryption is to keep the key completely secret. The safety is pending with the secret of our key. Our encryption algorithm is far from being that secure like e.g., AES. Nevertheless, it offers some level of safety. The advantages of using XOR for cryptography are:

- very fast computable, especially in hardware
- XOR is commutative. That means there is no difference between the right side (data) and the left side (key)
- XOR is associative. It does not matter how many and in which order you XOR values
- easy to understand and analyse

References

[Bas22] Basile. *AES Encryption and Decryption In Python: Implementation, Modes and Key Management*. 2022. URL: https://onboardbase.com/blog/aes-encryption-decryption/.

[Gal19] Frank Galuszka. *Hashing Passwords: One-Way Road to Security*. 2019. URL: https://auth0.com/blog/hashing-passwords-one-way-road-to-security/.

[Gee20] Geeks for Geeks. *Encrypt using XOR Cipher with Repeating Key*. 2020. URL: https://www.geeksforgeeks.org/encrypt-using-xor-cipher-with-repeating-key/.

[Kha23] Muhammad Waiz Khan. *XOR in Python*. 2023. URL: https://www.delftstack.com/howto/python/xor-in-python/.

[Leg22] Legrandin. *Modern modes of operation for symmetric block ciphers*. 2022. URL: https://pycryptodome.readthedocs.io/en/latest/src/cipher/modern.html#eax-mode.

[Mai] Maitry. *Intro To CustomTkinter – Make Your Modern-Looking GUIs In Python*. URL: https://python-hub.com/intro-customtkinter-modern-looking-guis-in-python/.

[Med23] Panda Mediacenter. *What Is AES Encryption? A Guide to the Advanced Encryption Standard*. 2023. URL: https://www.pandasecurity.com/en/mediacenter/security/what-is-aes-encryption/.

[Pyt] PythonTutorial.net. *Tkinter Grid*. URL: https://www.pythontutorial.net/tkinter/tkinter-grid/.

[Sch22] Tom Schimansky. *CustomTkinter*. 2022. URL: https://github.com/TomSchimansky/CustomTkinter/wiki.

[Vol19] Brent Vollebregt. *Python Encryption and Decryption with PyCryptodome*. 2019. URL: https://nitratine.net/blog/post/python-encryption-and-decryption-with-pycryptodome/.

[Wik23a] Wikipedia contributors. *Exclusive or — Wikipedia, The Free Encyclopedia*. [Online; accessed 13-April-2023]. 2023. URL: https://en.wikipedia.org/w/index.php?title=Exclusive_or&oldid=1139577080.

[Wik23b] Wikipedia contributors. *One-time pad — Wikipedia, The Free Encyclopedia*. [Online; accessed 2-May-2023]. 2023. URL: https://en.wikipedia.org/w/index.php?title=One-time_pad&oldid=1147326021.