University of Dhaka
Dept. of Computer Science and Engineering
Professional Masters in Information and Cyber Security (PMICS) Program

----

CSE 808 - Information Infrastructure Protection

Practical Demonstration of CIA

Lab Class 2 – Manual

Conducted by: Md. Shakhawat Hossain Robin

Lab Description

Welcome to the Cybersecurity Lab! In this hands-on session, we will delve into the fundamentals of cybersecurity, focusing on practical exercises of CIA (Confidentiality, Integrity & Availability). CIA is a fundamental concept that serves as the foundation for designing and implementing secure systems and protecting sensitive information. Our primary goal is to understand the practical scenario of CIA triad by doing various hands on task related to the Confidentiality, Integrity & Availability.

Lab Objectives:

- Understand the practical scenario of CIA triad.
- Overview of different types of data (e.g. plaintext, encoded, encrypted, etc.)
- Practical exercise of Confidentiality Pilar
- Familiarization with different types of Encoding, Encryption and Hashing Algorithm.
- Practical experience with encoding and decoding data
- Hands-on exercises of data encryption and decryption for both Symmetric and Asymmetric encryption.
- Gain practical experience and importance of Integrity Pillar.
- Demonstration Stress Testing and DoS & DDoS attack to understand the concept of Availability Pillar.

Prerequisites

- Computer with Kali Linux installed.
- Metasploitable 2
- Kali linux command line tools (openssl, hping3, hash-identifier, etc.)

Overview of CIA (Confidentiality, Integrity & Availability)

CIA, in the context of information security, stands for Confidentiality, Integrity, and Availability. It is a fundamental concept that serves as the foundation for designing and implementing secure systems and protecting sensitive information.

Confidentiality:

- Confidentiality ensures that sensitive information is accessible only to authorized individuals or entities.
- It involves measures such as encryption, access controls, authentication, and data masking to prevent unauthorized access, disclosure, or leakage of sensitive data.
- Confidentiality aims to protect sensitive information from unauthorized disclosure, espionage, or theft, maintaining privacy and preserving the secrecy of data.

Integrity:

- Integrity ensures that data remains accurate, complete, and unaltered throughout its lifecycle.
- It involves measures such as data validation, checksums, digital signatures, and access controls to prevent unauthorized modification, deletion, or corruption of data.
- Integrity aims to ensure the reliability and trustworthiness of data, safeguarding against unauthorized tampering, data manipulation, or corruption.

Availability:

- Availability ensures that information and resources are accessible and usable when needed by authorized users.
- It involves measures such as redundancy, fault tolerance, backups, disaster recovery planning, and denial-of-service (DoS) protection to maintain continuous access to critical systems and services.
- Availability aims to ensure that systems remain accessible, minimizing downtime, disruptions, or service interruptions.

CONFIDENTIALITY

Data Security

- **Plaintext:** Plaintext refers to the original, unencrypted data or message that is readable and understandable to humans. It could be a piece of text, a file, or any other form of data that is in its raw and unencrypted state.

- **Encoded text:** Encoded text is data that has been transformed from one format to another using a specific encoding scheme, such as Base64 or hexadecimal encoding. Encoding is a reversible process that alters the representation of data for purposes like compatibility, transmission, or storage, but it does not provide security or confidentiality like encryption

- **Cipher text:** Ciphertext is the encrypted form of plaintext, produced by applying an encryption algorithm and a key. It is intended to be unintelligible to anyone who does not possess the appropriate decryption key. Ciphertext appears as a scrambled or encoded version of the original plaintext and is typically unreadable without decryption.

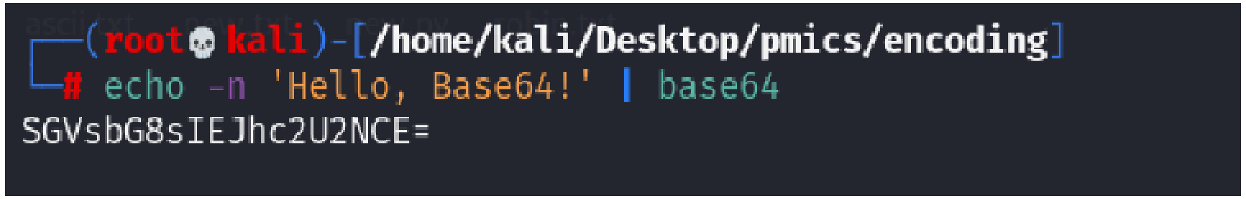
- **Symmetric Encryption:** In symmetric encryption, the same key is used for both encryption and decryption. This means that both the sender and the receiver must possess and share the same secret key. Symmetric encryption algorithms are typically faster and more efficient than asymmetric algorithms, making them suitable for encrypting large amounts of data. Examples of symmetric encryption algorithms include AES (Advanced Encryption Standard), DES (Data Encryption Standard), and 3DES (Triple Data Encryption Standard).

- **Asymmetric Encryption:** In asymmetric encryption, also known as public-key encryption, a pair of mathematically related keys is used: a public key and a private key. The public key is used for encryption, while the private key is kept secret and used for decryption. This allows for secure communication without the need to share a secret key. Asymmetric encryption is slower than symmetric encryption but provides features such as digital signatures and key exchange. Examples of asymmetric encryption algorithms include RSA (Rivest-Shamir-Adleman), ECC (Elliptic Curve Cryptography), and Diffie-Hellman key exchange.

Data Encoding and Decoding

Encoding data using the Base64 method.

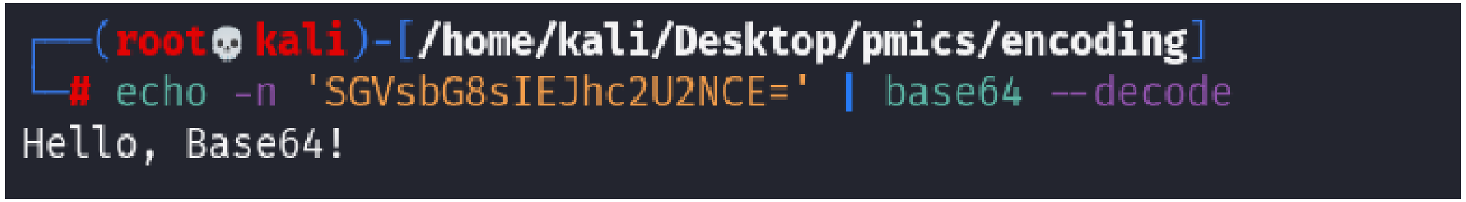
- Got to your linux terminal
- Command: `echo -n 'Hello, Base64!' | base64`



```
┌──(root💀kali)-[/home/kali/Desktop/pmics/encoding]
└─# echo -n 'Hello, Base64!' | base64
SGVsbG8sIEJhc2U2NCE=
```

Decoding data using the Base64 method.

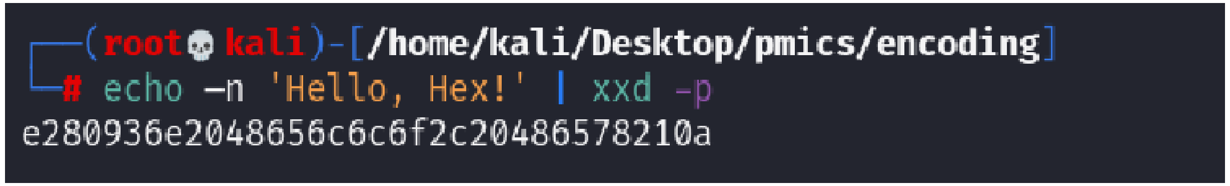
- Got to your linux terminal
- Command: `echo -n 'SGVsbG8sIEJhc2U2NCENCg==' | base64 --decode`



```
┌──(root💀kali)-[/home/kali/Desktop/pmics/encoding]
└─# echo -n 'SGVsbG8sIEJhc2U2NCE=' | base64 --decode
Hello, Base64!
```

Encoding data using Hexadecimal method

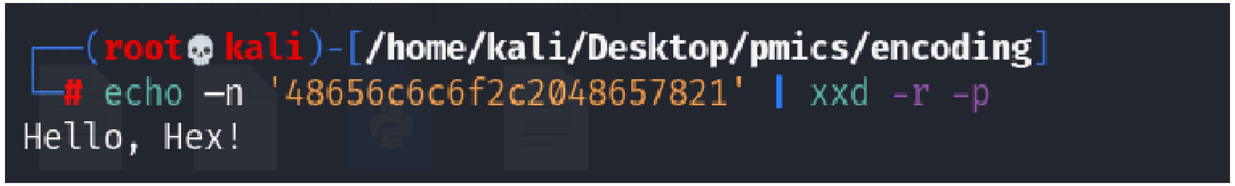
- Got to your linux terminal
- Command: `echo –n 'Hello, Hex!' | xxd -p`



```
┌──(root💀kali)-[/home/kali/Desktop/pmics/encoding]
└─# echo –n 'Hello, Hex!' | xxd -p
e280936e2048656c6c6f2c20486578210a
```

Decoding data using Hexadecimal method

- Got to your linux terminal
- Command: `echo –n '48656c6c6f2c2048657821' | xxd -r –p`



```
┌──(root💀kali)-[/home/kali/Desktop/pmics/encoding]
└─# echo –n '48656c6c6f2c2048657821' | xxd -r -p
Hello, Hex!
```

Data Encryption and Decryption

There are many encryption algorithms available, each with its own strengths and weaknesses. The "top" encryption algorithms can vary depending on factors such as security, efficiency, and suitability for specific use cases. Here's a list of ten widely recognized encryption algorithms, though it's important to note that the "top" algorithms can change over time as cryptography evolves:

Symmetric Encryption:

- In symmetric encryption, the same key is used for both encryption and decryption.
- Examples include AES (Advanced Encryption Standard), DES (Data Encryption Standard), and 3DES (Triple DES).
- Symmetric encryption is typically faster than asymmetric encryption but requires secure key exchange between parties.

Asymmetric Encryption (Public-Key Encryption):

- Asymmetric encryption uses a pair of keys: a public key for encryption and a private key for decryption.
- Examples include RSA (Rivest-Shamir-Adleman) and ECC (Elliptic Curve Cryptography).
- Asymmetric encryption is often used for key exchange and digital signatures in secure protocols like SSL/TLS.

Block Cipher vs. Stream Cipher:

- Block ciphers encrypt data in fixed-size blocks (e.g., 128 bits), while stream ciphers encrypt data bit by bit or byte by byte.
- Examples of block ciphers include AES (block size of 128 bits) and DES (block size of 64 bits).
- Examples of stream ciphers include RC4 and ChaCha20.

Full Disk Encryption:

- Full disk encryption (FDE) encrypts the entire storage device (e.g., hard drive, SSD) to protect data at rest.
- It ensures that all data on the disk is encrypted and requires authentication (e.g., password, encryption key) to access.

File-Level Encryption:

- File-level encryption encrypts individual files or directories, allowing selective encryption of specific data.
- It provides granular control over which files are encrypted and can be used to protect sensitive files or folders.

End-to-End Encryption (E2EE):

- End-to-end encryption ensures that data is encrypted from the sender to the recipient, with no intermediaries able to access the plaintext.
- It is commonly used in messaging apps, email services, and file-sharing platforms to protect user privacy.

Hashing:

- Hashing is a one-way encryption process that converts data into a fixed-length string of characters (hash value).
- Hash functions are used to generate unique hash values for input data, which are used for data integrity verification and password storage.
- Examples include SHA-1, SHA-2, SHA-3 (Secure Hash Algorithm) and MD5 (Message Digest Algorithm 5).

Symmetric Encryption

AES (Advanced Encryption Standard): AES is a symmetric key encryption algorithm widely used for securing data. It supports key lengths of 128, 192, and 256 bits and is considered highly secure and efficient.

RSA (Rivest-Shamir-Adleman): RSA is an asymmetric key encryption algorithm used for secure data transmission and digital signatures. It relies on the difficulty of factoring large prime numbers for its security.

DES (Data Encryption Standard): DES is a symmetric key encryption algorithm that was widely used in the past but is now considered insecure due to its small key size (56 bits). It has been superseded by AES.

3DES (Triple DES): 3DES is a symmetric key encryption algorithm that applies the DES algorithm three times to each data block, increasing its key size to 168 bits. It provides better security than DES but is slower and less efficient than AES.

Blowfish: Blowfish is a symmetric key encryption algorithm designed to be fast and secure. It supports variable key lengths (up to 448 bits) and is used in various applications.

Twofish: Twofish is a symmetric key encryption algorithm that was a finalist in the AES competition. It supports key lengths of 128, 192, and 256 bits and is considered secure and efficient.

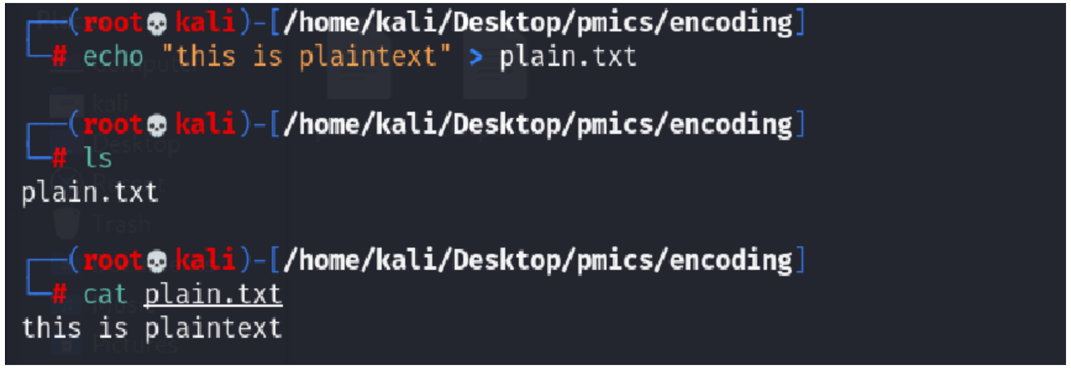
Encrypt and Decrypt data using AES (Advanced Encryption Standard)

Encryption (Symmetric)

Now we will practically encrypt the data using **symmetric encryption algorithm AES**. Here we will use the command line tool **openssl** which is used for data encryption and decryption.

Step 1 : Go to terminal

- `Command: echo “this is plaintext” > plain.txt`
- `Command: ls`
- `Command: cat plain.txt`



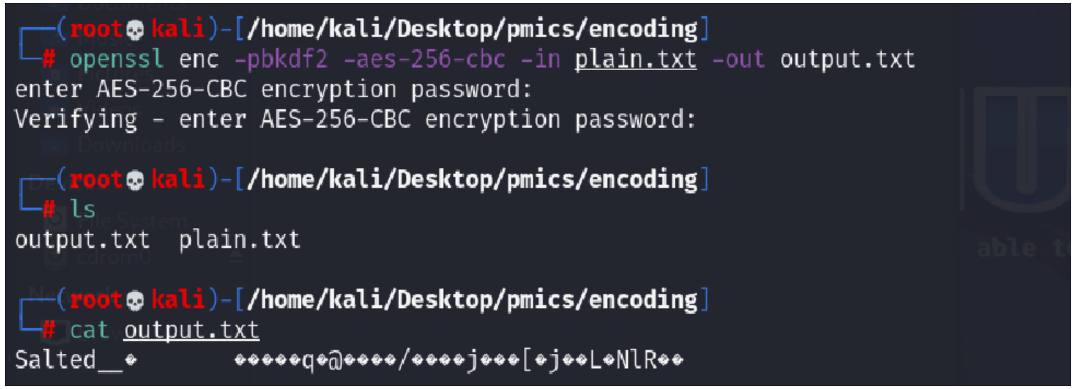
```
┌──(root💀kali)-[/home/kali/Desktop/pmics/encoding]
└─# echo "this is plaintext" > plain.txt

┌──(root💀kali)-[/home/kali/Desktop/pmics/encoding]
└─# ls
plain.txt

┌──(root💀kali)-[/home/kali/Desktop/pmics/encoding]
└─# cat plain.txt
this is plaintext
```

Step 2

- `Command: openssl enc -pbkdf2 -aes-256-cbc -in plain.txt -out output.txt`
- `Command: ls`
- `Command: cat output.txt`



```
┌──(root💀kali)-[/home/kali/Desktop/pmics/encoding]
└─# openssl enc -pbkdf2 -aes-256-cbc -in plain.txt -out output.txt
enter AES-256-CBC encryption password:
Verifying - enter AES-256-CBC encryption password:

┌──(root💀kali)-[/home/kali/Desktop/pmics/encoding]
└─# ls
output.txt  plain.txt

┌──(root💀kali)-[/home/kali/Desktop/pmics/encoding]
└─# cat output.txt
Salted__◆       ◆◆◆◆◆q◆@◆◆◆◆/◆◆◆◆j◆◆◆[◆j◆◆L◆NlR◆◆
```

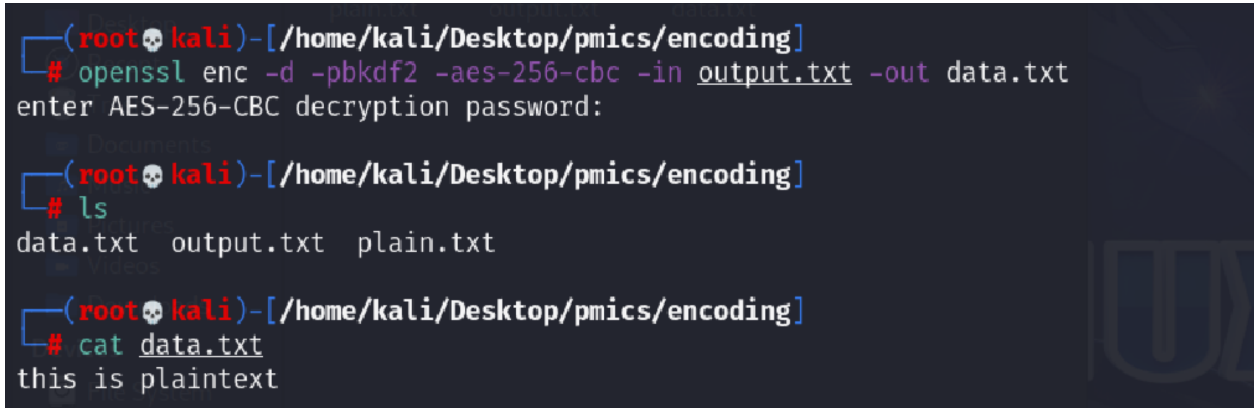
Here you can see that the data is encrypted and totally unreadable.

Decryption (Symmetric)

Now we will decrypt the data using **AES algorithm** by using the same command line tool **openssl**.

Step 1: Go to terminal

- `Command: ls`
- `Command: openssl enc -d -pbkdf2 -aes-256-cbc -in output.txt -out data.txt`
- `Command: ls`
- `Command: cat data.txt`



```
┌──(root💀kali)-[/home/kali/Desktop/pmics/encoding]
└─# openssl enc -d -pbkdf2 -aes-256-cbc -in output.txt -out data.txt
enter AES-256-CBC decryption password:

┌──(root💀kali)-[/home/kali/Desktop/pmics/encoding]
└─# ls
data.txt  output.txt  plain.txt

┌──(root💀kali)-[/home/kali/Desktop/pmics/encoding]
└─# cat data.txt
this is plaintext
```

Used commands details

- **`-pbkdf2:`** specifies the use of the PBKDF2 algorithm for key derivation, which is recommended for better security.
- **`-aes-256-cbc:`** specifies the encryption algorithm to use
- **`-in:`** specifies the input file to be encrypted (plain.txt and output.txt in this example).
- **`-out:`** specifies the output file where the encrypted data will be saved (output.txt and data.txt in this example).
- **`-d:`** used for specify the decryption algorithm

Asymmetric Encryption

RSA (Rivest-Shamir-Adleman):

- RSA is one of the oldest and most widely used asymmetric encryption algorithms.
- It is based on the difficulty of factoring large composite numbers into their prime factors.
- RSA is used for secure data transmission, digital signatures, and key exchange in various protocols like SSL/TLS.

DSA (Digital Signature Algorithm):

- DSA is an asymmetric encryption algorithm designed specifically for digital signatures.
- It is based on the difficulty of solving the discrete logarithm problem in a finite field.
- DSA is used for creating and verifying digital signatures in protocols like SSL/TLS and digital certificates.

Diffie-Hellman (DH):

- Diffie-Hellman is a key exchange algorithm used to establish a shared secret between two parties over an insecure communication channel.
- It is based on the difficulty of solving the discrete logarithm problem in a finite field.
- Diffie-Hellman is used in various protocols like SSL/TLS, SSH, and IPsec for secure key exchange.

Elliptic Curve Cryptography (ECC):

- ECC is an asymmetric encryption algorithm based on the algebraic structure of elliptic curves over finite fields.
- It offers strong security with shorter key lengths compared to RSA, making it more efficient in computation and bandwidth.
- ECC is used in protocols like SSL/TLS, PGP, and cryptocurrencies for key exchange, digital signatures, and encryption.

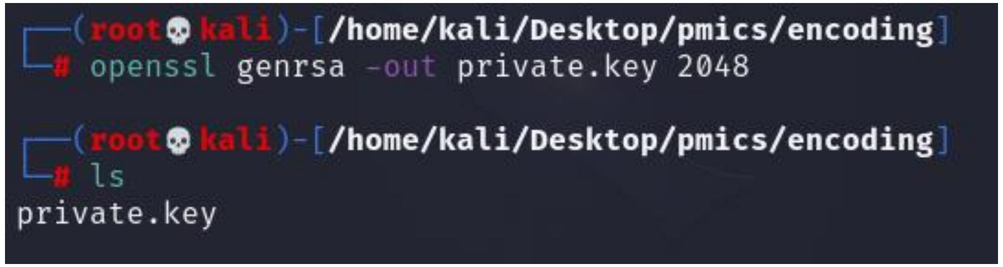
Encrypt and Decrypt data using RSA (Rivest-Shamir-Adleman)

Public key and Private key Generation

Now we will generate private.key and public.key of **RSA algorithm** by using the same tool **openssl**.

Step 1 : Go to terminal

- `Command: openssl genrsa -out private.key 2048`
- `Command: ls`

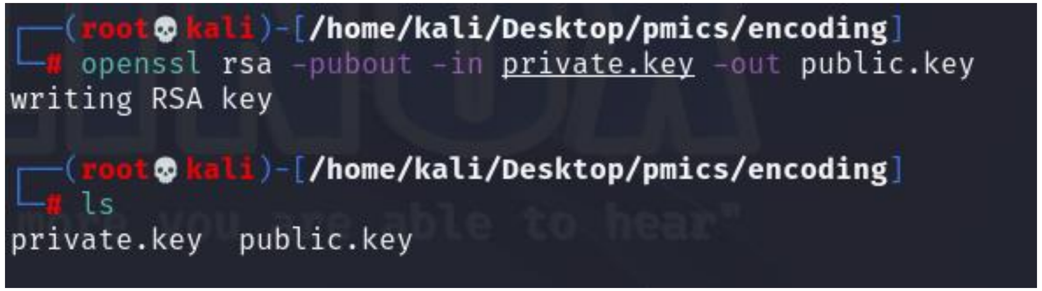


```
┌──(root💀kali)-[/home/kali/Desktop/pmics/encoding]
└─# openssl genrsa -out private.key 2048

┌──(root💀kali)-[/home/kali/Desktop/pmics/encoding]
└─# ls
private.key
```

Step 2 : Now we will generate the public.key of the corresponding private.key

- `Command: openssl rsa -pubout -in private.key -out public.key`
- `ls`



```
┌──(root💀kali)-[/home/kali/Desktop/pmics/encoding]
└─# openssl rsa -pubout -in private.key -out public.key
writing RSA key

┌──(root💀kali)-[/home/kali/Desktop/pmics/encoding]
└─# ls
private.key  public.key
```

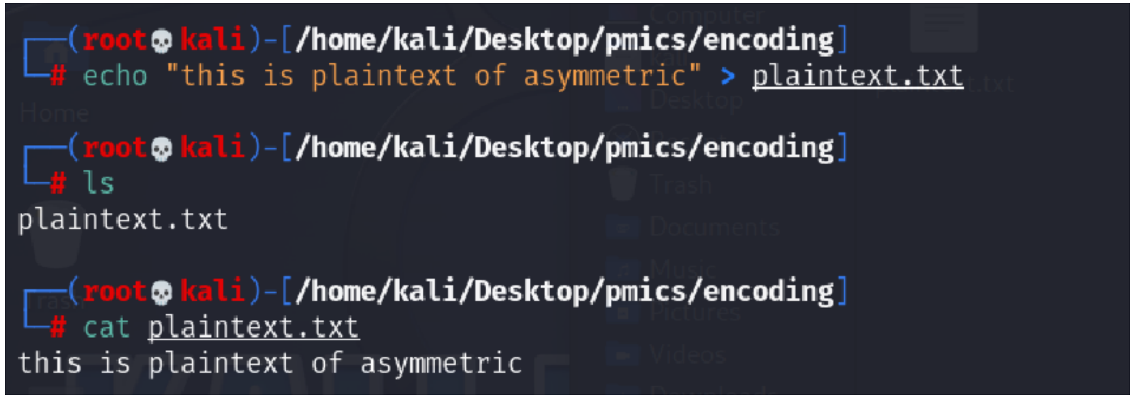
Now we have generated both of the private and public keys

Data Encryption (Asymmetric)

Now we will practically encrypt the data using **asymmetric encryption algorithm RSA** by using the tool **openssl**.

Step 1 : Go to terminal

- `Command: echo “this is plaintext of asymmetric” > plaintext.txt`
- `Command: ls`
- `Command: cat plaintext.txt`



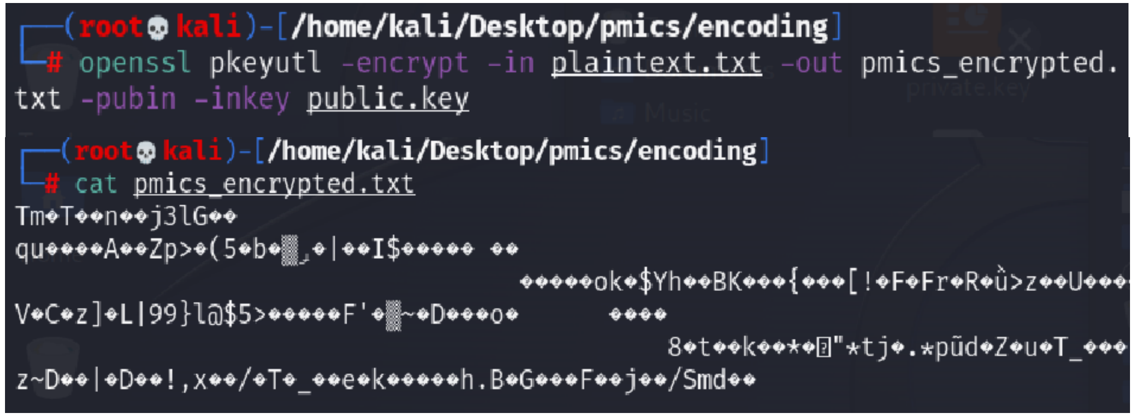
```
┌──(root💀kali)-[/home/kali/Desktop/pmics/encoding]
└─# echo "this is plaintext of asymmetric" > plaintext.txt

┌──(root💀kali)-[/home/kali/Desktop/pmics/encoding]
└─# ls
plaintext.txt

┌──(root💀kali)-[/home/kali/Desktop/pmics/encoding]
└─# cat plaintext.txt
this is plaintext of asymmetric
```

Step 2 :

- `Command: openssl pkeyutl -encrypt -in plaintext.txt -out pmics_encrypted.txt -pubin -inkey public.key`
- `command: cat pmics_encrypted.txt`



```
┌──(root💀kali)-[/home/kali/Desktop/pmics/encoding]
└─# openssl pkeyutl -encrypt -in plaintext.txt -out pmics_encrypted.
txt -pubin -inkey public.key

┌──(root💀kali)-[/home/kali/Desktop/pmics/encoding]
└─# cat pmics_encrypted.txt
Tm�T��n��j3lG��
qu����A��Zp>�(5�b�▒,�|��I$����� ��
                                      �����ok�$Yh��BK���{���[!�F�Fr�R�ů>z��U���
V�C�z]�L|99}l@$5>�����F'�▒~�D����o�       ����
                                             8�t��k���*�⍰"*tj�.*pũd�Z�u�T_���
z~D��|�D��!,x��/�T�_��e�k�����h.B�G���F��j��/Smd��
```

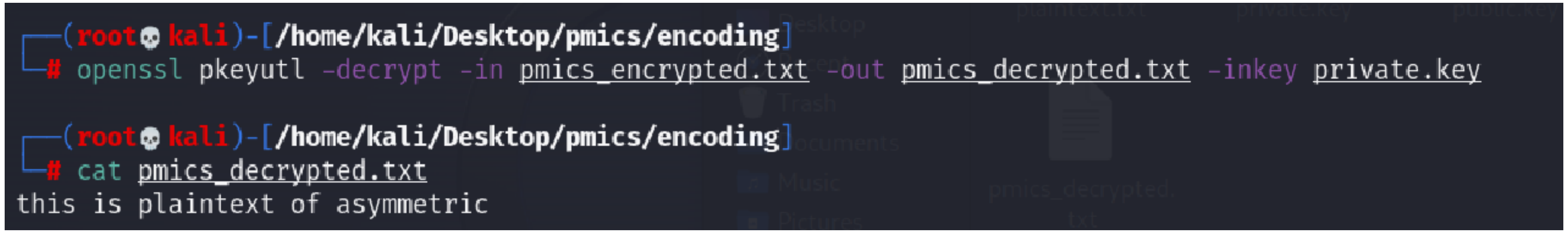
Here you can see that the file is fully encrypted by using the public.key

Data Decryption (Asymmetric)

Now we will decrypt the data by using the corresponding **private.key** which was encrypted through **public.key** of **RSA Algorithm.**

Step 1 : Go to terminal

- `Command: openssl pkeyutl -decrypt -in pmics_encrypted.txt -out pmics_decrypted.txt -inkey private.key`
- `Command: cat pmics_decrypted.txt`



```
┌──(root💀kali)-[/home/kali/Desktop/pmics/encoding]
└─# openssl pkeyutl -decrypt -in pmics_encrypted.txt -out pmics_decrypted.txt -inkey private.key

┌──(root💀kali)-[/home/kali/Desktop/pmics/encoding]
└─# cat pmics_decrypted.txt
this is plaintext of asymmetric
```

Used commands details

- **`-pkeyutl:`** The command itself for performing public key operations.
- **`-encrypt:`** Specifies that the operation to be performed is encryption.
- **`-decrypt:`** Specifies that the operation to be performed is decryption.
- **`-in:`** Specifies the input file containing the data to be decrypted.
- **`-out:`** Specifies the output file where the decrypted data will be written.
- **`-inkey:`** Specifies the public/private key file to be used for encryption/decryption.

Hashing Algorithms

Hashing is a process of converting input data (often referred to as a message) into a fixed-size string of bytes, typically using a hash function. The output string is commonly known as a hash value or hash code. Hashing is a one-way process, meaning it's computationally infeasible to reverse the transformation and obtain the original input data from the hash value.

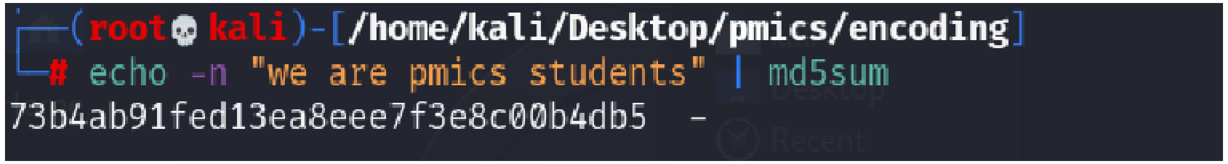
Hash functions are designed to fulfill several key properties:

- **Deterministic:** For a given input, a hash function will always produce the same hash value.
- **Fast Computation:** Hash functions should compute the hash value efficiently, even for large input data.
- **Fixed Output Size:** The output hash value has a fixed length, regardless of the size of the input data.
- **Preimage Resistance:** It should be computationally infeasible to determine the original input data from the hash value.
- **Collision Resistance:** It should be difficult to find two different inputs that produce the same hash value.

Different Types of Hashing Algorithm

❖ MD5 (Message Digest Algorithm 5):

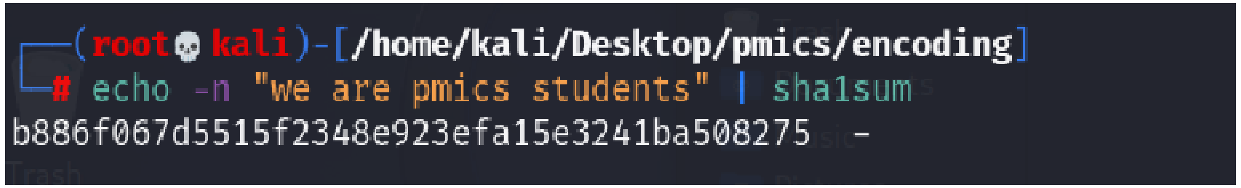
- MD5 is a widely used hashing algorithm that produces a 128-bit hash value.
- It is commonly used for checksums and data integrity verification.
- However, MD5 is considered weak for cryptographic purposes due to vulnerabilities that allow for collision attacks.



```
┌──(root💀kali)-[/home/kali/Desktop/pmics/encoding]
└─# echo -n "we are pmics students" | md5sum
73b4ab91fed13ea8eee7f3e8c00b4db5  -
```

❖ SHA-1 (Secure Hash Algorithm 1):

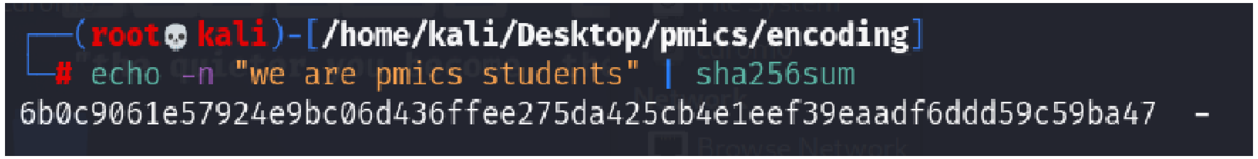
- SHA-1 is a widely used hashing algorithm that produces a 160-bit hash value.
- It was used in cryptographic applications but is now considered weak due to vulnerabilities that allow for collision attacks.
- SHA-1 use the Merkle-Damgård construction,



```
┌──(root💀kali)-[/home/kali/Desktop/pmics/encoding]
└─# echo -n "we are pmics students" | sha1sum
b886f067d5515f2348e923efa15e3241ba508275  -
```

❖ SHA-2 (Secure Hash Algorithm 2):

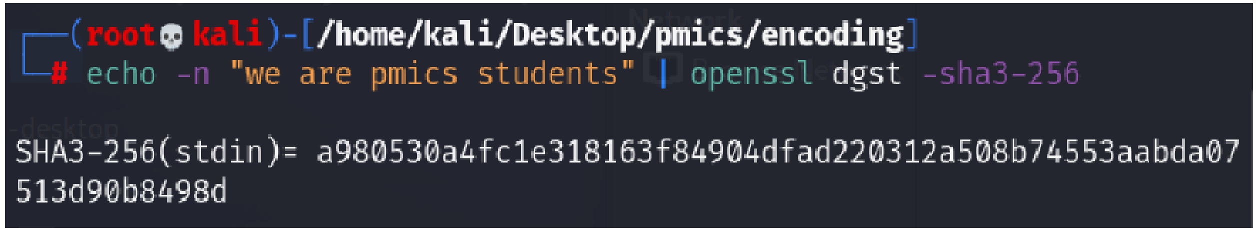
- SHA-2 is a family of hashing algorithms that includes SHA-224, SHA-256, SHA-384, and SHA-512.
- These algorithms produce hash values of various lengths (224, 256, 384, and 512 bits) and are considered secure for cryptographic purposes.
- SHA-2 use the Merkle-Damgård construction,



```
┌──(root💀kali)-[/home/kali/Desktop/pmics/encoding]
└─# echo -n "we are pmics students" | sha256sum
6b0c9061e57924e9bc06d436ffee275da425cb4e1eef39eaadf6ddd59c59ba47  -
```

❖ **SHA-3 (Secure Hash Algorithm 3):**

- SHA-3 is the latest member of the Secure Hash Algorithm family, standardized by NIST.
- It includes hash functions with different output lengths, including SHA3-224, SHA3-256, SHA3-384, and SHA3-512.
- SHA-3 is designed to provide increased security and resistance to cryptographic attacks.
- SHA-3 uses the sponge construction based on the Keccak permutation.

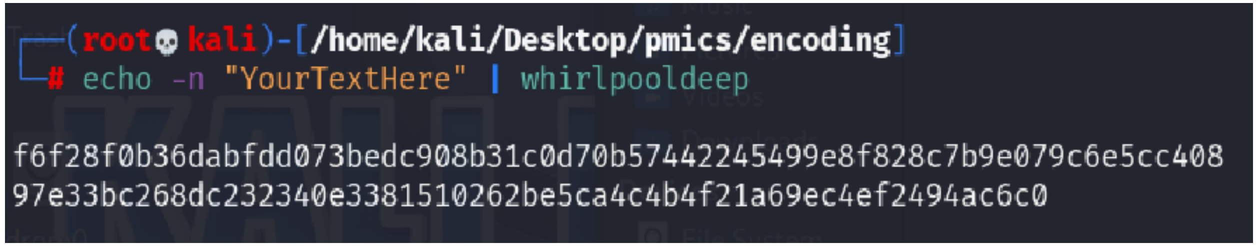


```
┌──(root💀kali)-[/home/kali/Desktop/pmics/encoding]
└─# echo -n "we are pmics students" | openssl dgst -sha3-256

SHA3-256(stdin)= a980530a4fc1e318163f84904dfad220312a508b74553aabda07
513d90b8498d
```

❖ **Whirlpool:**

- Whirlpool is a cryptographic hash function that produces a 512-bit hash value.
- It is based on a substantially modified version of the Advanced Encryption Standard (AES) algorithm.



```
┌──(root💀kali)-[/home/kali/Desktop/pmics/encoding]
└─# echo -n "YourTextHere" | whirlpooldeep

f6f28f0b36dabfdd073bedc908b31c0d70b57442245499e8f828c7b9e079c6e5cc408
97e33bc268dc232340e3381510262be5ca4c4b4f21a69ec4ef2494ac6c0
```

Thank You