

CSA NOV 2024 Assignment 1

Cryptography

Prepared By : S Divya

Table of Contents - Overview

1. Encode your name using Base64 and provide the resulting text. Decode it back to your name
2. Encrypt your name using AES + ECB and key hackerspace12345 and provide the cipher text - Try to decrypt it back
3. Generate an RSA Key pair - Use public key encrypt your name and provide the resulting ciphertext - try to decrypt the ciphertext using the private key
4. Generate SHA256 hash of your name and provide the resulting hash - Try to reverse the hashing by any method

1. Encode your name using Base64 and provide the resulting text. Decode it back to your name

Encoding the Name

Text to Encode: "S Divya"

Convert to Bytes: Before encoding, converting the string "S Divya" into its byte representation using the ASCII value of each character.

The byte representation for "S Divya" :-

- 'S' → 83 (ASCII value)
- ' ' (space) → 32 (ASCII value)
- 'D' → 68 (ASCII value)
- 'i' → 105 (ASCII value)
- 'v' → 118 (ASCII value)
- 'y' → 121 (ASCII value)
- 'a' → 97 (ASCII value)

Base64 Encoding works by converting this byte stream into a readable string, grouping the data in 6-bit chunks and mapping it to the corresponding Base64 characters.

Base64 encoding takes input data and divides it into chunks of 3 bytes (24 bits). Each 3-byte chunk is then split into 4 groups of 6 bits, and each 6-bit group is mapped to a corresponding Base64 character.

This gives the following bytes : [83, 32, 68, 105, 118, 121, 97]

Divide the bytes into groups of 3:

- Group 1: [83, 32, 68] (3 bytes)
- Group 2: [105, 118, 121] (3 bytes)
- Group 3: [97] (only 1 byte, so we need to pad the rest)

The last group only has 1 byte of data, so to make it a 3-byte group, pad it with two zero bytes. The padding will later be represented as == in the Base64 output

S		D	i	v	y	a
83	32	68	105	118	121	97
01010011	00100000	01000100	01101001	01110110	01111001	01100001

Each 3-byte chunk is then split into 4 groups of 6 bits, and each 6-bit group is mapped to a corresponding Base64 character

010100	110010	000001	000100	011010	010111	011001	111001	011000	010000
20	50	1	4	26	23	25	57	24	16
U	y	B	E	a	X	Z	5	Y	Q

So the Base64 encoded result for “S Divya” :- UyBEaXZ5YQ==

Decoding the Base64 String

To decode the Base64 encoded string back to the original name -

- Take the encoded string "**UyBEaXZ5YQ==**"
- Decode it using Base64. This process reverses the encoding step by converting the Base64 string back to the original byte values.

Remove the padding characters. Then **Convert the Base64 Characters to 6-bit Groups**: 010100 110010 000001 000100
011010 010111 011001 111001 011000 010000

Combine the 6 Bits chunk into a Single Stream: 010100110010000001000100011010010111011001111001011000010000

Split the Stream into 8-bit (Byte) Chunks: 01010011 00100000 01000100 01101001 01110110 01111001 01100001

Convert the 8-bit Groups Back to ASCII Character: 83 32 68 105 118 121 97

Then Reconstruct the Original Data which is : **S Divya**

Decoded result: When decoded "**UyBEaXZ5YQ==**" gives the original string "**S Divya**" as originally encoded
So, The result after decoding "**UyBEaXZ5YQ==**" is "**S Divya**"

•**Original Text** : "S Divya"

•**Base64 encoded result**: "UyBEaXZ5YQ=="

•**Decoded result**: "S Divya"

2. Encrypt your name using AES + ECB and key hackerspace12345 and provide the cipher text - Try to decrypt it back

AES Encryption and Decryption with ECB Mode

- **AES** requires a key of a specific length. AES can use 128-bit (16 bytes), 192-bit (24 bytes), or 256-bit (32 bytes) keys. In the case given, the key "**hackerspace12345**" is already 16 characters long, which corresponds to 128 bits (1 byte = 8 bits, so 16 bytes = 128 bits).

Padding the Plaintext

- AES operates on **blocks** of 16 bytes (128 bits). If the length of the data is not a multiple of 16 bytes, it must be padded to fit the 16-byte boundary. The plaintext "**S Divya**" has a length of 7 characters, which is less than 16 bytes, so needed to **pad** it to 16 bytes.
- **PKCS7** padding is commonly used for AES, which ensures the data length is a multiple of 16 bytes. The padding scheme works by adding bytes that indicate the number of bytes added (so that during decryption, the padding can be removed)

Encrypting the Data

- With AES, use **ECB mode**, which means each block of data is encrypted independently. Encrypting the padded plaintext with the `cipher.encrypt()` method. The result is the ciphertext (which will be in hexadecimal format)

Decrypting the Ciphertext

- The decryption process reverses the encryption steps. After decrypting the ciphertext, will remove the padding that was added during encryption to recover the original plaintext message " S Divya". Using the `cipher.decrypt()` method to decrypt the ciphertext.

Python Code:

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
from binascii import hexlify

# Key and plaintext preparation
key = b'hackerspace12345' # 16-byte key (128 bits)
plaintext = "S Divya" # The plaintext message

# Pad the plaintext to make its length a multiple of 16 bytes
padded_plaintext = pad(plaintext.encode(), AES.block_size)

# Create AES cipher in ECB mode
cipher = AES.new(key, AES.MODE_ECB)

# Encrypt the padded plaintext
ciphertext = cipher.encrypt(padded_plaintext)

# Decrypt the ciphertext
decrypted_padded_plaintext = cipher.decrypt(ciphertext)

# Unpad the decrypted plaintext to get the original
decrypted_plaintext = unpad(decrypted_padded_plaintext, AES.block_size).decode()

# Print results
print(f"Ciphertext (in hex): {hexlify(ciphertext).decode()}")
print(f"Decrypted plaintext: {decrypted_plaintext}")
```

Output

Ciphertext (in hex): 7004e623e61cc32bf4659cd0ca7698d2

Decrypted plaintext: S Divya

The image shows a Visual Studio Code editor window with a Python script named `aes.py` open. The script implements AES encryption and decryption using the `Crypto` library. The output of the script is displayed in the terminal window at the bottom.

Python Script (aes.py):

```
1 from Crypto.Cipher import AES
2 from Crypto.Util.Padding import pad, unpad
3 from binascii import hexlify
4
5 # Step 1: Key and plaintext preparation
6 key = b'hackerspace12345' # 16-byte key (128 bits)
7 plaintext = "S Divya" # The plaintext message
8
9 # Step 2: Pad the plaintext to make its length a multiple of 16 bytes
10 padded_plaintext = pad(plaintext.encode(), AES.block_size)
11
12 # Step 3: Create AES cipher in ECB mode
13 cipher = AES.new(key, AES.MODE_ECB)
14
15 # Step 4: Encrypt the padded plaintext
16 ciphertext = cipher.encrypt(padded_plaintext)
17
18 # Step 5: Decrypt the ciphertext
19 decrypted_padded_plaintext = cipher.decrypt(ciphertext)
20
21 # Step 6: Unpad the decrypted plaintext to get the original
22 decrypted_plaintext = unpad(decrypted_padded_plaintext, AES.block_size).decode()
23
24 # Step 7: Print results
25 print(f"Ciphertext (in hex): {hexlify(ciphertext).decode()}")
26 print(f"Decrypted plaintext: {decrypted_plaintext}")
27
```

Terminal Output:

```
PS C:\Users\User\Documents> & 'c:\Users\User\AppData\Local\Programs\Python\Python312\python.exe' 'c:\Users\User\.vscode\extensions\ms-python.debugpy-2024.14.0-win32-x64\bundled\libs\debugpy\adapter\..\..\debugpy\launcher' '60042' '--' 'C:\Users\User\Documents\aes.py'
Ciphertext (in hex): 7004e623e61cc32bf4659cd0ca7698d2
PS C:\Users\User\Documents> & 'c:\Users\User\AppData\Local\Programs\Python\Python312\python.exe' 'c:\Users\User\.vscode\extensions\ms-python.debugpy-2024.14.0-win32-x64\bundled\libs\debugpy\adapter\..\..\debugpy\launcher' '60042' '--' 'C:\Users\User\Documents\aes.py'
Ciphertext (in hex): 7004e623e61cc32bf4659cd0ca7698d2
Ciphertext (in hex): 7004e623e61cc32bf4659cd0ca7698d2
Decrypted plaintext: S Divya
PS C:\Users\User\Documents>
```


3. Generate an RSA Key pair - Use public key encrypt your name and provide the resulting ciphertext - try to decrypt the ciphertext using the private key

Generating an RSA Key Pair

Generate two keys: the **private key** (Used for decryption) and the **public key** (Used for encryption). These keys are mathematically linked.

In RSA, encryption can only be performed with the public key, while decryption can only happen with the private key.

Encryption using the Public Key

Using the **public key** to encrypt the message "**S Divya**". The encryption process transforms your plaintext into ciphertext (scrambled data) which cannot be read by anyone without the private key.

Decryption using the Private Key

The encrypted message (ciphertext) can only be decrypted by someone with the **private key**. By using the private key to decrypt, should get back the original message, which is "**S Divya**" in this case.

Python Code

```
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
from base64 import b64encode, b64decode

# Generate RSA Key Pair
key = RSA.generate(2048) # 2048-bit key for good security
private_key = key.export_key() # Export the private key
public_key = key.publickey().export_key() # Export the public key

# Print out the private and public keys
# print("Private Key:", private_key.decode())
# print("Public Key:", public_key.decode())

# Encrypt the message (Public Key encryption)
message = "S Divya".encode() # Convert the message to bytes

# Initialize the cipher with the public key for encryption
cipher = PKCS1_OAEP.new(RSA.import_key(public_key)) # Using OAEP padding for RSA
ciphertext = cipher.encrypt(message) # Encrypt the message

# Output the Ciphertext (Base64 encoding for readability) The resulting ciphertext is binary, so it's encoded into a Base64 format
print("Ciphertext (Base64 encoded):", b64encode(ciphertext).decode())

# Decrypt the message (Private Key decryption)
decipher = PKCS1_OAEP.new(RSA.import_key(private_key)) # Initialize with private key for decryption
decrypted_message = decipher.decrypt(ciphertext) # Decrypt the ciphertext

# Output the decrypted message
print("Decrypted message:", decrypted_message.decode()) # Convert bytes back to string
```

Output

Ciphertext (Base64 encoded): <Base64 string> which is:

l7inPVp01yBuYcJ2yvml4qj9AQxh3+J4wuxuEBWgeTOmbVr4c6ZpTUms6xiZHoyJPYoYIEOHTP8kZf7gnSNP6761Us4pf7fnl5TSyKk74OMcYP7W2x/7cOdWOdlfzEMOVExkWqPurF17OT83vm8gHO8fgNoi5jjYmqGmwg1njwUdYqZzgXeBlvshXwXCDLe8ef3ButuHBED5Xkg091YmBJ2PCfsHq+cohDMkAaCHDt9XI3uQ4FsYEjdiMt49Y3Dh/HtBh+J+VVc4+63DzsZsoteVoOP1SgHqW5Xci10SpPuy6ILSUuqSd791dsE22T7JgpLZNRaj822ACf0FhClvWg==
SUuqSd791dsE22T7JgpLZNRaj822ACf0FhClvWg==

Decrypted message: S Divya

File Edit Selection View Go Run Terminal Help

Search

Python Debug Console

RUN AND DEBUG

Run and Debug

To customize Run and Debug, open a folder and create a launch.json file.

Show automatic Python configurations

BREAKPOINTS

Raised Exceptions

Uncaught Exceptions

User Uncaught Exceptions

aes.py

rsa.py

C:\Users\User\Documents> rsa.py ...

```
1 from Crypto.PublicKey import RSA
2 from Crypto.Cipher import PKCS1_OAEP
3 from base64 import b64encode, b64decode
4
5 # Step 1: Generate RSA Key Pair
6 key = RSA.generate(2048) # 2048-bit key for good security
7 private_key = key.export_key() # Export the private key
8 public_key = key.publickey().export_key() # Export the public key
9
10 # Print out the private and public keys (for your reference)
11 print("Private Key:", private_key.decode())
12 print("Public Key:", public_key.decode())
13
14 # Step 2: Encrypt the message (Public Key encryption)
15 message = "S Divya".encode() # Convert the message to bytes
16
17 # Initialize the cipher with the public key for encryption
18 cipher = PKCS1_OAEP.new(RSA.import_key(public_key)) # Using OAEP padding for RSA
19 ciphertext = cipher.encrypt(message) # Encrypt the message
20
21 # Step 3: Output the Ciphertext (Base64 encoding for readability)
22 print("Ciphertext (Base64 encoded):", b64encode(ciphertext).decode())
23
24 # Step 4: Decrypt the message (Private Key decryption)
25 decipher = PKCS1_OAEP.new(RSA.import_key(private_key)) # Initialize with private key for decryption
26 decrypted_message = decipher.decrypt(ciphertext) # Decrypt the ciphertext
27
28 # Step 5: Output the decrypted message
29 print("Decrypted message:", decrypted_message.decode()) # Convert bytes back to string
```

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

PORTS

Ciphertext (Base64 encoded): l7inPVp01yBuYcJ2yvml4qj9AQxh3+J4wuxuEBWgeTOmbVr4c6ZpTUms6xiZHoyJPYoYIEOHTP8kZf7gnSNP6761Us4pf7fnl5TSyKk74OMcYP7W2x/7cOdWOdlfzEMOVExkWqPurF17OT83vm8gHO8fgNoi5jjYmqGmwg1njwUdYqZzgXeBlvshXwXCDLe8ef3ButuHBED5Xkg091YmBJ2PCfsHq+cohDMkAaCHDt9XI3uQ4FsYEjdiMt49Y3Dh/HtBh+J+VVc4+63DzsZsoteVoOP1SgHqW5Xci10SpPuy6ILSUuqSd791dsE22T7JgpLZNRaj822ACf0FhClvWg==

Decrypted message: S Divya

PS C:\Users\User\Documents>

Spaces: 4

UTF-8

{}

Python

3.12.2 64-bit

01:47 AM

24-01-2025

4. Generate SHA256 hash of your name and provide the resulting hash - Try to reverse the hashing by any method

Generating SHA256 Hash for "S Divya"

Python code:

```
import hashlib

# Name input
name = "S Divya"

# Generate SHA256 hash
sha256_hash = hashlib.sha256(name.encode()).hexdigest()

print("SHA256 Hash of 'S Divya':", sha256_hash)
```

Output:

SHA256 Hash of 'S Divya':
3416c497c07ae3801b22b435eda7e36399b325a58ec6927ef1e997a0ccbd9342

Reversing SHA256 Hash

SHA256 is a **one-way cryptographic function**, meaning that reversing it directly (or finding the original input from the hash) is computationally infeasible.

Why it's hard to reverse:

One-Way Nature of Hash Functions

- SHA256 is a **one-way function**, meaning that once applying the algorithm to the input, it's designed to be irreversible. So that it's almost impossible to retrieve the original input from the hash.
- Once you hash the name "S Divya" the resulting hash does not contain any inherent reversible relationship with the original input.
- **You can easily generate a hash from an input**, but there's no straightforward way to take the hash and regenerate the input data.

Pre-image Resistance

- This property means that even knowing the output (the hash value), it's nearly impossible to reverse-engineer the original input. Without the original input (in this case, the name "S Divya"), there's no way to directly get back to it.
- A SHA256 hash doesn't store any information that could directly point back to the original string. It only generates a fixed-length output that doesn't map back easily to the input.

Brute-Force Attack

- Could attempt to use a **brute-force attack**, where to try every possible combination of strings, hash each one, and check if it matches the original hash. However, given the vast number of possible inputs, this would take an impractical amount of time.

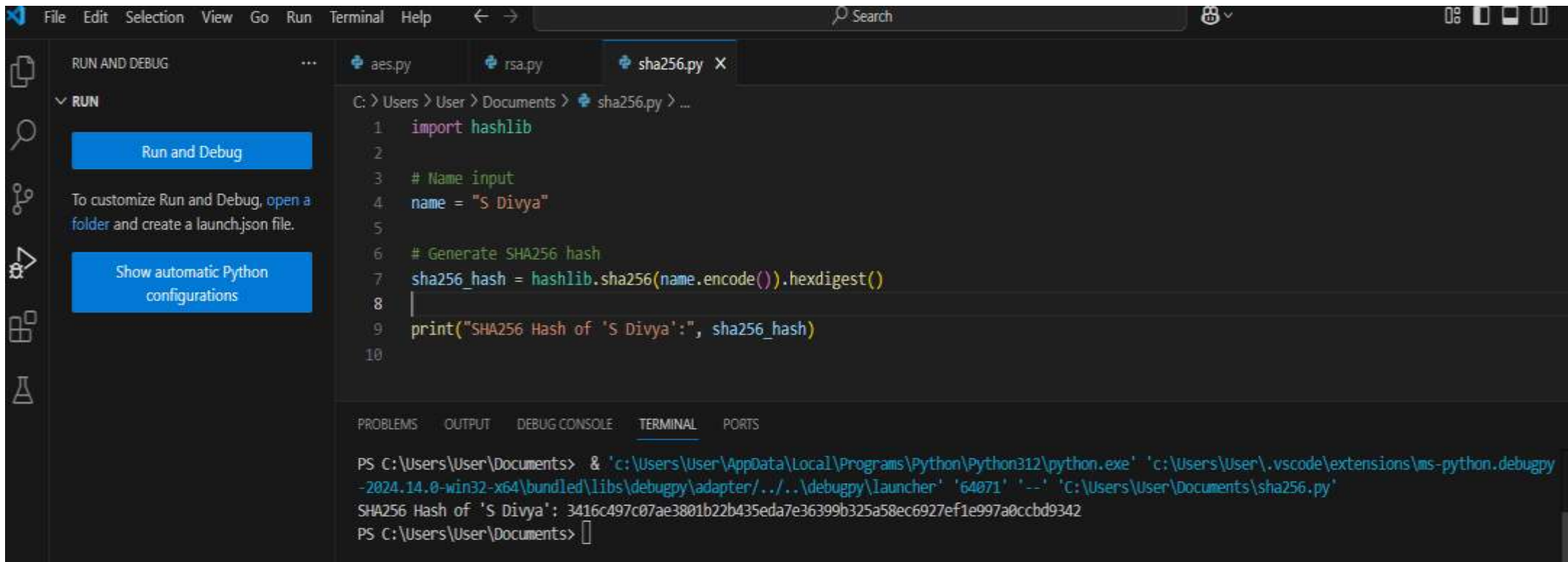
Rainbow tables

- They are precomputed tables of hash values for common inputs, but they don't typically include names like "S Divya" Additionally, this approach only works for simple, predictable inputs.

Conclusion: Reversing SHA256

- Reversing a SHA256 hash is **not feasible**. There's no practical way to reverse a SHA256 hash directly back to its original string (which in the case is my name)
- Therefore, once generated the hash of "**S Divya**" using SHA256, there's **no efficient way to reverse it** back to the original string without already knowing what the string was.
- The only reliable way to know what the original input was is if you already have it, or if you happen to encounter it in a precomputed hash table (which is unlikely for unique names).
- In short, the hash **cannot be reversed**, which is a critical property of cryptographic hashes like SHA256. This is what makes SHA256 and other cryptographic hash functions so secure.

Output:SHA256 Hash for "S Divya"



The screenshot shows the Visual Studio Code interface. The editor has three tabs: `aes.py`, `rsa.py`, and `sha256.py`. The `sha256.py` tab is active, displaying the following Python code:

```
1 import hashlib
2
3 # Name input
4 name = "S Divya"
5
6 # Generate SHA256 hash
7 sha256_hash = hashlib.sha256(name.encode()).hexdigest()
8
9 print("SHA256 Hash of 'S Divya':", sha256_hash)
10
```

On the left, the 'RUN AND DEBUG' sidebar is visible with options like 'Run and Debug' and 'Show automatic Python configurations'. At the bottom, the 'TERMINAL' panel shows the command prompt output:

```
PS C:\Users\User\Documents> & 'c:\Users\User\AppData\Local\Programs\Python\Python312\python.exe' 'c:\Users\User\.vscode\extensions\ms-python.debugpy-2024.14.0-win32-x64\bundled\libs\debugpy\adapter\..\..\debugpy\launcher' '64071' '--' 'C:\Users\User\Documents\sha256.py'
SHA256 Hash of 'S Divya': 3416c497c07ae3801b22b435eda7e36399b325a58ec6927ef1e997a0ccbd9342
PS C:\Users\User\Documents>
```