Assessment 3

Cryptography Analysis and Implementation

Name: Pawar Adwyait Shivaji

Reg. Number: 20BCE2088

COURSE: CYBERSECURITY AND ETHICAL HACKING SMARTBRIDGE EXTERNSHIP PROGRAM

VIT Mail ID: pawaradwyait.shivaji2020@vitstudent.ac.in

Objective: The objective of this assignment is to analyse cryptographic algorithms and implement them in a practical scenario.

Instructions:

Research: Begin by conducting research on different cryptographic algorithms such as symmetric key algorithms (e.g., AES, DES), asymmetric key algorithms (e.g., RSA, Elliptic Curve Cryptography), and hash functions (e.g., MD5, SHA-256). Understand their properties, strengths, weaknesses, and common use cases.

Analysis: Choose three cryptographic algorithms (one symmetric, one asymmetric, and one hash function) and write a detailed analysis of each. Include the following points in your

1. Briefly explain how the algorithm works.
2. Discuss the key strengths and advantages of the algorithm.
3. Identify any known vulnerabilities or weaknesses.
4. Provide real-world examples of where the algorithm is commonly used.

Advanced Encryption Standard (Symmetric cryptographic algorithm):

The symmetric encryption method known as AES (Advanced Encryption Standard) is used to safeguard sensitive data. It employs a configurable key length of 128, 192, or 256 bits and operates with fixed block sizes of 128 bits. AES creates the ciphertext by applying a sequence of mathematical operations to the plaintext data while employing a secret key.

Multiple rounds of substitution, permutation, and mixing procedures are used throughout the AES encryption process. The input data and a string of round keys created by the encryption key are used in these processes. SubBytes, ShiftRows, and MixColumns are the three primary steps in the encryption process. Depending on the key size, these steps are repeated for a specific number of rounds.

Working of the Advanced Encryption Standard:

1. **Key Expansion:** The original key undergoes a key expansion process, generating a set of round keys that will be used in the encryption and decryption process.
2. **Initial Round:** The input plaintext is divided into 16-byte blocks. Each block is XORed with the first-round key.
3. **Rounds:** AES performs multiple rounds (10, 12, or 14 rounds depending on the key length) that consist of four operations:
    - SubBytes: Each byte of the block is substituted with a corresponding byte from the S-box, a predefined substitution table.
    - ShiftRows: Bytes in each row of the block are shifted cyclically.
    - MixColumns: Each column of the block is transformed using a mathematical operation that provides diffusion.
    - AddRoundKey: The block is XORed with the round key.
4. **Final Round**: The final round omits the MixColumns operation.
5. **Output:** After all rounds are completed, the resulting ciphertext is generated.

Key strengths and advantages:

1. Security: AES has undergone a thorough analysis by cryptographers and is thought to be very secure. When used properly, it has resisted many assaults and is currently regarded as robust to known cryptographic flaws.
2. Efficiency: AES is computationally efficient, especially when used on current processors with specialised instructions or when implemented in hardware. It is appropriate for a variety of applications since it can swiftly encrypt and decrypt data.
3. Simplicity: The AES algorithm has a straightforward structure, making it easy to understand and implement correctly. The algorithm's security benefits from this simplicity because it lessens the possibility of implementation mistakes.
4. Standardization: AES is a widely used worldwide standard that was adopted by the National Institute of Standards and Technology (NIST) of the United States. The standardisation of it guarantees the compatibility and interoperability of various systems and applications.

Vulnerabilities and disadvantages:

1. Side-channel attacks: Techniques including timing assaults, power analyses, and electromagnetic analyses can be used to compromise AES implementations. These attacks use information that was exposed during encryption, such as timing differences or power usage, to recoup the secret key.
2. Key management: The administration and security of encryption keys play a critical role in AES security. Weak key management procedures, such as employing insecure key storage or weak passwords, might compromise the system's overall security.
3. Quantum attack: AES is susceptible to attacks from quantum computers, despite the fact that it is currently thought to be secure against attacks from conventional computers. To lessen this vulnerability as quantum computing develops, it is crucial to create quantum-resistant encryption techniques.

Real-world examples of AES are:

1. Secure File Transfer: The file transfer protocols FTPS (FTP over SSL/TLS) and HTTPS (secure version of HTTP) both frequently use AES. AES is used to encrypt the data being communicated when you browse a website using HTTPS or transfer files using FTPS, guaranteeing its anonymity.
2. Disk Encryption: AES is utilized in disk encryption software like BitLocker (Windows) and FileVault (Mac) to protect the data stored on hard drives or other storage devices. These encryption systems use AES to encrypt the entire disk, making the data inaccessible without the correct decryption key.
3. Messaging Apps: Many popular messaging applications employ AES for end-to-end encryption, ensuring that only the intended recipients can decrypt and read the messages. Examples include WhatsApp, Signal, and Telegram, which use AES to secure the communication between users.

Rivest-Shamir-Adleman (Asymmetric cryptographic algorithm):

The RSA algorithm is based on the modular arithmetic and large prime number features of mathematics. It uses two separate keys—a public key for encryption and a private key for decryption—because it is an asymmetric encryption technique. The difficulty of factoring huge numbers into RSA's prime factors is the foundation of its security.RSA is a widely used asymmetric cryptographic algorithm.

Working of the RSA:

1. **Key Generation:**

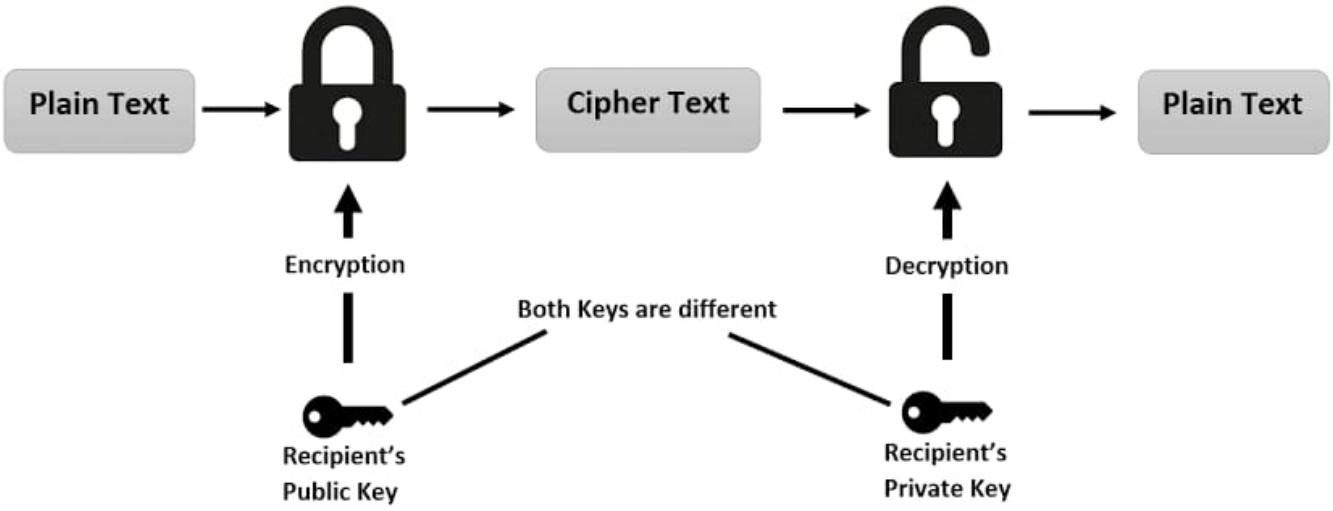
- Select two large prime numbers, p and q.
- Calculate n = p * q.
- Compute φ(n) = (p-1) * (q-1), where φ is Euler's totient function.
- Choose a public exponent, e, which is relatively prime to φ(n) and less than φ(n).
- Calculate the private exponent, d, such that (e * d) % φ(n) = 1.
- The public key is (e, n), and the private key is (d, n).

2. **Encryption:**

- Convert the message into a numerical representation.
- Use the recipient's public key (e, n) to encrypt the message.
- Apply the encryption formula: Ciphertext = Message^e % n.

3. **Decryption:**

- Use the recipient's private key (d, n) to decrypt the ciphertext.
- Apply the decryption formula: Message = Ciphertext^d % n.



Key Strengths and Advantages of RSA:

1. **Security:** In situations when huge key sizes are employed, RSA offers robust security. Factoring big numbers, which involves a lot of work, is a requirement for breaking RSA encryption.

2. **Asymmetric Encryption:** RSA uses distinct keys for encryption and decryption, it is an asymmetric algorithm. This makes it possible for partners that have never exchanged a secret key to communicate securely.
3. **Digital Signatures:** Digital signatures can be generated using RSA and then verified. Message authenticity and integrity are ensured by using the private key to sign and the public key to verify the signature.

Weaknesses and Vulnerabilities of RSA:

1. **Key Size:** The security of RSA hinges on how challenging it is to factor huge numbers. The required key sizes have grown as computational power has grown to ensure security. Key sizes that are too small can be attacked.
2. **Key Management:** Secure key creation, distribution, and storage are essential to RSA. An attacker can decrypt any message encrypted with the appropriate public key if the private key is compromised.
3. **Timing Attacks:** Implementation flaws, such as discrepancies in the timing of decryption, can be used by attackers to obtain data on the private key.

Real-World Examples of RSA:

1. RSA is widely used in various real-world applications, including:
2. Secure communication protocols like HTTPS, SSH, and SSL/TLS.
3. Digital signatures for document signing, code signing, and certificate authorities.
4. Encryption and decryption of sensitive data in storage and transit.
5. Key exchange in secure protocols like Diffie-Hellman key exchange.

Secure Hash Algorithm (Hash function):

SHA-256 (Secure Hash Algorithm 256-bit) is a widely used cryptographic hash function that belongs to the SHA-2 family of hash functions. It takes an input message of any size and produces a fixed-size (256-bit) hash value, which is typically represented as a sequence of hexadecimal digits.

Working of the SHA-256:

1. Message Padding: The input message is processed in blocks of 512 bits. If the message is not already a multiple of 512 bits, padding is applied to ensure it reaches the appropriate length. The padding consists of a '1' bit, followed by a number of '0' bits, and finally, the length of the original message in binary format.
2. Initialization: SHA-256 starts with an initial set of constant values known as "initial hash values" or "IVs." These values are predefined and specific to the SHA-256 algorithm.
3. Message Processing: The padded message is divided into blocks of 512 bits each. For each block, a series of operations is performed to transform the input block and update the internal state of the hash function.
    a. Break the block into 16 words: Each 512-bit block is divided into 16 words of 32 bits each.
    b. Prepare the message schedule: The 16 words are expanded into a larger set of words called the "message schedule." This involves applying various bitwise operations, such as XOR, AND, and rotations.
    c. Initialize working variables: SHA-256 uses eight working variables (a, b, c, d, e, f, g, h) to hold intermediate hash values.
    d. Main compression loop: The compression loop consists of 64 iterations, where the working variables are updated based on the message schedule and the current block.
4. Final Hash Value: Once all the blocks have been processed, the final hash value is derived by concatenating the values of the working variables (a, b, c, d, e, f, g, h) after the last block has been processed.
5. The resulting 256-bit hash value is unique to the input message, meaning even a slight change in the input will produce a significantly different hash. This property makes SHA-256 suitable for a variety of applications, such as data integrity checks, digital signatures, password hashing, and blockchain technology.

Key strengths and advantages of SHA-256:

1. Collision Resistance: SHA-256 is made to be very resistant to collisions, which means it is highly improbable that two distinct inputs will result in the same hash value. This characteristic guarantees data integrity and hinders unauthorised changes.
2. Deterministic Output: SHA-256 will consistently provide the same output when given the same input. This characteristic is crucial for confirming data accuracy and guaranteeing consistency across diverse applications.
3. Fixed Output Size: The output size of SHA-256 is always fixed at 256 bits (or 32 bytes). This makes it appropriate for applications that demand a particular hash size and makes managing and storing hash data easier.
4. Wide Adoption: SHA-256 is a secure hash algorithm that has gained widespread adoption. Various cryptographic methods, digital signatures, and password hashing algorithms all use it

Weaknesses and Vulnerabilities of SHA-256:

1. Quantum Vulnerability: Like most hash functions, SHA-256 is susceptible to attacks by quantum computers. Quantum algorithms, such as Shor's algorithm, could potentially break the security of SHA-256 and other similar hash functions. However, practical quantum computers capable of executing such attacks are not yet available.
2. Length Extension Attacks: SHA-256 is vulnerable to length extension attacks, where an attacker who knows the hash of a message can append additional data to the message without knowing the original content, while still being able to generate a valid hash for the modified message. To mitigate this vulnerability, it is recommended to use HMAC (Hash-based Message Authentication Code) construction or other secure methods when using SHA-256 for message security.
3. Brute Force Attacks: While SHA-256 is designed to be computationally secure, its security relies on the difficulty of finding preimage or collision attacks. As computational power increases over time, the possibility of brute force attacks also increases. It is important to use a sufficiently long and random input to make brute force attacks infeasible.
4. Dependency on the Hash Function: SHA-256's security depends on the assumption that the underlying hash function is secure. If any weaknesses or vulnerabilities are discovered in the algorithm itself, it could compromise the security of SHA-256.

To address some of the weaknesses and vulnerabilities, newer hash functions, such as SHA-3, have been developed as alternatives to SHA-256. However, as of my knowledge cut-off in September 2021, SHA-256 remains widely used and considered secure for most practical purposes.

SHA-256 is widely used in various real-world applications:

1. Blockchain Technology: SHA-256 is a fundamental component in blockchain technology, including cryptocurrencies like Bitcoin. Bitcoin uses SHA-256 extensively for mining, transaction verification, and ensuring the integrity of the blockchain.
2. Digital Signatures: SHA-256 is often employed in digital signature schemes to ensure the authenticity and integrity of digital documents. It is used in conjunction with asymmetric cryptography algorithms, such as RSA or DSA, to sign and verify digital signatures.
3. Password Hashing: Many systems and websites use SHA-256 (or its variants) as a secure hashing algorithm to store and verify user passwords. Passwords are hashed using SHA-256 and then stored in a database, enhancing security by protecting user passwords even if the database is compromised.
4. File Integrity Checking: SHA-256 can be used to verify the integrity of files and ensure they haven't been tampered with. By computing the SHA-256 hash of a file and comparing it to a known hash value, one can confirm that the file hasn't been modified or corrupted.
5. SSL/TLS Certificates: SHA-256 is used in the process of generating and signing SSL/TLS certificates. It ensures the authenticity and integrity of the certificate by generating a hash value that is used to sign the certificate, thereby securing communication between web servers and clients.
6. Digital Forensics: In digital forensics investigations, SHA-256 can be used to create hash values of digital evidence such as files or disk images. These hash values can be used for data integrity verification and identification of known malicious files.
7. Its strength, wide adoption, and cryptographic properties make it suitable for a range of security-critical applications.

Implementation:

Aim: To implement the SHA-256 algorithm for password hashing.
Step 1: Import the hashlib library.
Step 2: Define a function to hash the password.The hash_password function takes a password as input. First, the password is encoded as UTF-8 because the hashlib library requires byte-like objects. Then, a new SHA-256 hash object is created. The hash object is updated with the password bytes, and finally, the hexadecimal representation of the hash value is obtained using hexdigest().
Step 3: Use the function to hash a password. Hash the password using the hash_password function and store the hashed password in the hashed_password variable. Finally, we print the hashed password to verify the result.

Code Snippets:

```
import hashlib

def hash_password(password):
    password = password.encode('utf-8')
    sha256_hash = hashlib.sha256()
    sha256_hash.update(password)
    hashed_password = sha256_hash.hexdigest()
    return hashed_password

password = "myPassword123"
hashed_password = hash_password(password)
print("Hashed Password:", hashed_password)
```

```
import hashlib
```

```
def hash_password(password):
    password = password.encode('utf-8')# Encode the password as UTF-8 before hashing
    sha256_hash = hashlib.sha256()# Create a new SHA-256 hash object
    sha256_hash.update(password) # Update the hash object with the password bytes
    hashed_password = sha256_hash.hexdigest()# Get the hexadecimal representation of the hash value
    return hashed_password
```

```
password = "myPassword123"
hashed_password = hash_password(password)
print("Hashed Password:", hashed_password)
```

```
Hashed Password: 71d4ec024886c1c8e4707fb02b46fd568df44e77dd5055cadc3451747f0f2716
```

Results:

Hashed Password: 71d4ec024886c1c8e4707fb02b46fd568df44e77dd5055cadc3451747f0f2716

The password "myPassword123" is hashed using SHA-256, and the resulting hashed password is " 71d4ec024886c1c8e4707fb02b46fd568df44e77dd5055cadc3451747f0f2716". Each time you run the code with the same password, you should get the same hashed password. Hashing the password ensures that it is not stored in plain text, providing a higher level of security.

Security Analysis:

Potential Threats or Vulnerabilities:

1. Brute Force Attacks: By continuously hashing various character combinations until a match is found, an attacker could attempt to guess the original password.
2. Rainbow Table Attacks: Attackers can swiftly reverse engineer hashed passwords by using precomputed tables (rainbow tables), which contain mappings of hashed passwords to their original values.
3. Dictionary Attacks: Attackers can attempt to match a hashed password by using a list of frequently used passwords or a dictionary.
4. Collision Attacks: A collision could happen when two different passwords generate the same hash, enabling an attacker to get beyond the hash function's security.

Countermeasures or Best Practices:

1. Salted Hashing: Adding a unique random salt value to each password before hashing makes rainbow table attacks ineffective. The salt must be specific to each user and kept in the same location as the hashed password.
2. Iterative Hashing: Using many iterations of hashing (key stretching) makes brute force and dictionary assaults more time-consuming for attackers.
3. Password Complexity and Policy: Promote the usage of complex passwords that contain a mix of uppercase, lowercase, numerals, and special characters. Implement password policies that demand a certain level of complexity and length.
4. Constantly Updated Hashing Algorithms: Use the most advanced hashing algorithms available. Older methods may have known flaws, therefore stay up to date with the newest discoveries in this field.

Limitations or Compromises:

Since hashing is a one-way operation, it is impossible to deduce the original password from the hashed value. Although this is ideal for password storage, it might provide problems when password recovery is required. Hashing a password by itself is insufficient for full security. To offer a complete security solution, further steps including safe password storage, secure network connection, and secure access controls should be put into place.

Conclusion:

Cybersecurity and ethical hacking both rely heavily on cryptography. For the purpose of safeguarding user credentials, a secure password hashing technique like SHA-256 must be implemented. It's crucial to realise that security is a continual process and that no single step can guarantee total security. It is advised to stay current with the