|  |  |  |  |
| --- | --- | --- | --- |
| **Course Name:** | **Applied Cryptography (216H02L401)** | **Semester:** | **IV** |
| **Date of Performance:** | **12/03/2025** | **DIV/ Batch No:** | **C - 3** |
| **Student Name:** | **Romil Lodaya** | **Roll No:** | **16010122096** |

**Experiment No: 8**

|  |
| --- |
| **Title: Implementation of authentication algorithms** |

|  |
| --- |
| **Aim and Objective of the Experiment:** |
| Students should generate Captcha and one of the following authentication mechanisms. Students may use libraries to implement the experiment.   * **CAPTCHA Integration**: Create a CAPTCHA system to differentiate between human users and bots. Students can generate simple text based or math based captchas.   Use video tutorial: https://www.youtube.com/watch?v=bfKwizfuuOU   * **One-Time Pads (OTP)**: Implement a one-time pad system that generates a series of unique codes for users during login. When executed on two diff systems, the onetime pad sequence has to be the same. * **Challenge-Response Systems**: Design a challenge-response mechanism where the server issues a challenge (e.g., a random number or phrase), and the user must respond correctly based on a shared secret key. * **Password-Based Authentication with Salt**: Accept password in plain, generate a random salt. Store salt and hash (encrypted password + salt) in the table. When a password is entered, use the same encryption algorithm, use the corresponding salt stored, compute hash function and allow only when there’s a match. Students may use any encryption algorithm. |

|  |
| --- |
| **COs to be achieved:** |
| **CO3: Comprehend cryptographic hash functions, Message Authentication Codes and**  **Digital Certificates and their uses for Authentication** |

|  |
| --- |
| **Books/ Journals/ Websites referred:** |
| 1. <https://www.tutorialspoint.com/cryptography_with_python/cryptography_with_python_one_time_pad_cipher.htm> 2. <https://www.geeksforgeeks.org/implementation-of-vernam-cipher-or-one-time-pad-algorithm/> |

|  |
| --- |
| **Theory: 🡺 Students should write theory post studying with references** |
| ****1. One-Time Pads (OTP)********Theory:**** A **One-Time Pad (OTP)** is a cryptographic method that provides perfect secrecy when used correctly. It was first described by **Claude Shannon** in the 1940s. The OTP relies on the concept of **key randomness** and **key secrecy**, ensuring that the key (used to encrypt the message) is random, as long as the message, and only used once. ****Key Features:****  * **Randomness**: The key used in an OTP must be completely random. This randomness is what ensures the security of the encryption. * **Key Length**: The length of the key must be equal to or longer than the message being encrypted. * **One-Time Use**: The key can only be used once. If a key is reused, the encryption becomes vulnerable. * **Symmetric Encryption**: Both the sender and receiver must have access to the same OTP, which is a major challenge in practical systems. * **Perfect Secrecy**: If the OTP is truly random, the encryption is theoretically unbreakable, as each ciphertext has an equal probability of mapping to any plaintext.  ****How OTP Works:****  * A random key (OTP) is generated. * The message is XORed with the key to produce the ciphertext. * The ciphertext is sent over the channel. * The receiver, who has the same key, XORs the ciphertext with the key to retrieve the original message.  ****Drawbacks****:  * **Key Distribution**: The biggest challenge with OTPs is securely distributing the key to the recipient before any communication can happen. * **Key Management**: If the OTP is lost or stolen, the security of the system is compromised.  ****2. Challenge-Response Authentication********Theory:**** **Challenge-Response Authentication** is a cryptographic authentication protocol where the server challenges the client with a question or challenge, and the client responds with an answer (response) that proves it has knowledge of a secret. This type of authentication is widely used in secure communication protocols like SSL/TLS. ****Key Features:****  * **Authentication Without Transmission of Secrets**: The shared secret is never transmitted, reducing the risk of interception. * **Resistance to Replay Attacks**: Since the challenge changes with each authentication request, even if an attacker intercepts a valid response, it cannot be reused. * **Symmetric Secret**: Both the client and server share a secret key that is used to generate the response.  ****How Challenge-Response Works:****  1. The **server** sends a challenge (typically a random number or string). 2. The **client** combines the challenge with a shared secret and generates a response (e.g., using a hash function). 3. The **server** compares the response with its own generated response. If they match, the authentication succeeds.  ****Example:****  * The client and server share a secret password. * The server sends a random challenge. * The client computes a response by hashing the challenge with the shared password. * The server compares the response with its own computed response.  ****Advantages:****  * **Security**: The secret is never transmitted over the network, making it resistant to eavesdropping and man-in-the-middle attacks. * **Flexibility**: It can be used in a variety of systems where user identity verification is required.  ****3. Password-Based Authentication with Salt********Theory:**** **Password-Based Authentication** with **Salt** is a security technique designed to protect user passwords from being easily cracked in case a database is compromised. Salting involves adding random data (the salt) to the password before hashing it, making identical passwords look different in storage and defending against rainbow table attacks. ****Key Features:****  * **Salt**: A salt is a random string of characters that is concatenated with the password before hashing. It ensures that even if two users have the same password, their hashed password will be different due to different salts. * **Hashing**: Hashing algorithms like SHA-256 or bcrypt are used to convert the password (plus salt) into a fixed-length string. * **Storage**: Instead of storing plaintext passwords, the system stores the hash of the password combined with the salt.  ****How Password-Based Authentication with Salt Works:****  1. **User Registration**:    * A random salt is generated.    * The password is combined with the salt and hashed using a cryptographic hashing algorithm (e.g., SHA-256).    * The salt and the hashed password are stored in the database. 2. **User Login**:    * The user enters the password.    * The system retrieves the salt from the database, combines it with the entered password, and hashes it again.    * If the generated hash matches the stored hash, the user is authenticated.  ****Advantages:****  * **Protection Against Rainbow Tables**: Since each password has a unique salt, even if two users have the same password, the resulting hash will be different. * **Improved Security**: Even if an attacker gains access to the hashed passwords, they cannot easily reverse the hashes back to the original passwords without also having access to the salts.  ****Conclusion**** These three authentication techniques — **One-Time Pads (OTP)**, **Challenge-Response Authentication**, and **Password-Based Authentication with Salt** — are essential to building secure systems. OTPs provide perfect secrecy but are difficult to manage, while challenge-response systems enhance security through dynamic exchanges of secrets. Password-based authentication with salt mitigates the risks of password theft by making it harder to reverse-engineer stored hashes. All of these methods are crucial in modern cybersecurity practices to ensure the protection of sensitive information. |

|  |
| --- |
| **Code :** |
| 1] One Time Pad:  import os  *def* generate\_otp(*length*):      return os.urandom(length)  *def* encrypt\_decrypt(*otp*, *message*):      if isinstance(message, *str*):          message\_bytes = message.encode('utf-8')      else:          message\_bytes = message      return *bytes*([m ^ otp[i] for i, m in enumerate(message\_bytes)])  otp = generate\_otp(16)  *# generate OTP of 16 bytes*  message = "Hello, World!"  *# Encrypt the message*  ciphertext = encrypt\_decrypt(otp, message)  print("Ciphertext:", ciphertext)  *# Decrypt the message (using the same OTP)*  decrypted\_message = encrypt\_decrypt(otp, ciphertext).decode('utf-8')  *# Convert bytes back to string*  print("Decrypted Message:", decrypted\_message)  2] Challenge Response Systems:  import random  import hashlib  *# Shared secret key*  SECRET\_KEY = "sharedsecret"  *def* generate\_challenge():      return random.randint(1000, 9999)  *def* generate\_response(*challenge*, *secret\_key*):      return hashlib.sha256(*f*"{challenge}{secret\_key}".encode()).hexdigest()  *def* authenticate\_user(*challenge*, *user\_response*, *secret\_key*):      correct\_response = generate\_response(challenge, secret\_key)      return correct\_response == user\_response  challenge = generate\_challenge()  print("Server Challenge:", challenge)  *# User generates the response using the shared secret*  user\_response = generate\_response(challenge, SECRET\_KEY)  print("User Response:", user\_response)  *# Server authenticates the user*  is\_authenticated = authenticate\_user(challenge, user\_response, SECRET\_KEY)  print("Authentication Successful:", is\_authenticated)  3] Password Based Authentication with Salt: |

|  |
| --- |
| **Output:** |
| 1] One Time Pad:    2] Challenge Response Systems:    3] Password Based Authentication with Salt: |

|  |
| --- |
| **Post Lab Subjective/Objective type Questions:** |
| 1. Combine approaches to implement multifactor authentication   To implement **multifactor authentication (MFA)** by combining the three approaches you mentioned—**One-Time Pads (OTP)**, **Challenge-Response Authentication**, and **Password-Based Authentication with Salt**—we can create a system where each approach adds an extra layer of security. MFA is essentially a method that requires more than one factor for authentication, typically something the user knows (password), something the user has (OTP or token), and something the user is (biometric verification, though we're not using biometrics here). ****Steps to Combine These Approaches for MFA****  1. **Step 1: Password-Based Authentication with Salt**    * The user provides a password. This password is then salted and hashed and stored securely in the system.    * The server retrieves the stored hash and compares it with the user input to authenticate the first factor (something the user knows). 2. **Step 2: OTP Generation (Something the User Has)**    * After verifying the password, an OTP is generated using the system's random number generator or time-based OTP generation methods (like TOTP).    * The OTP is sent to the user via a secure channel (e.g., via email, SMS, or an app like Google Authenticator).    * The user is then required to enter the OTP they receive, providing a second factor of authentication. 3. **Step 3: Challenge-Response Mechanism (Additional Security)**    * Once the password and OTP are successfully validated, the system can challenge the user with a random challenge (e.g., a random number or phrase).    * The user computes a response using a shared secret and the challenge. The server verifies if the response matches its expected value (using a hash function or another verification method).    * This adds another layer of authentication that ensures the user has knowledge of a shared secret.  ****Code Implementation for MFA**** import os  import random  import hashlib  import time  from datetime import datetime, timedelta  SHARED\_SECRET = "supersecretkey"  *def* generate\_salt():      return os.urandom(16)  *def* hash\_password(*password*, *salt*):      return hashlib.sha256(salt + password.encode()).hexdigest()  *def* store\_password(*password*):      salt = generate\_salt()      hashed\_password = hash\_password(password, salt)      return salt, hashed\_password  *def* verify\_password(*stored\_salt*, *stored\_hashed\_password*, *input\_password*):      input\_hashed\_password = hash\_password(input\_password, stored\_salt)      return input\_hashed\_password == stored\_hashed\_password  *def* generate\_otp():      return (*int*(time.time()) % 10000)  *def* generate\_challenge():      return random.randint(1000, 9999)  *def* generate\_response(*challenge*, *secret\_key*):      return hashlib.sha256(*f*"{challenge}{secret\_key}".encode()).hexdigest()  *def* authenticate\_user(*challenge*, *user\_response*, *secret\_key*):      correct\_response = generate\_response(challenge, secret\_key)      return correct\_response == user\_response  *def* password\_authentication(*stored\_salt*, *stored\_hashed\_password*):      entered\_password = input("Enter your password: ")        if verify\_password(stored\_salt, stored\_hashed\_password, entered\_password):          print("Password verified successfully.")          return True      else:          print("Invalid password.")          return False  *def* otp\_authentication():      otp = generate\_otp()      print(*f*"Generated OTP: {otp}")        entered\_otp = *int*(input("Enter the OTP: "))        if entered\_otp == otp:          print("OTP verified successfully.")          return True      else:          print("Invalid OTP.")          return False  *def* challenge\_response\_authentication():      challenge = generate\_challenge()      print(*f*"Generated Challenge: {challenge}")    *# User computes the response*      user\_response = input("Enter your response (hash of challenge + shared secret): ")      expected\_response = generate\_response(challenge, SHARED\_SECRET)        if user\_response == expected\_response:          print("Challenge-Response verified successfully.")          return True      else:          print("Invalid response.")          return False  *def* multifactor\_authentication():      password = "userpassword"      salt, hashed\_password = store\_password(password)      print("User registered successfully.")      if not password\_authentication(salt, hashed\_password):          return "Authentication failed at password step."      if not otp\_authentication():          return "Authentication failed at OTP step."      if not challenge\_response\_authentication():          return "Authentication failed at challenge-response step."      return "Authentication successful!"  if \_\_name\_\_ == "\_\_main\_\_":      result = multifactor\_authentication()      print(result) ****Explanation of the Code:****  1. **Password Authentication with Salt:**    * We use a salted password hash for the first authentication step. The password is hashed with a randomly generated salt and stored in the database.    * During login, the system hashes the entered password and compares it with the stored hash. 2. **OTP Generation:**    * The OTP is generated using the current time (time.time() % 10000) for simplicity, though in real systems you'd use a time-based OTP library like pyotp for more robust security.    * The user is prompted to enter the OTP, and it is compared with the generated one for validation. 3. **Challenge-Response Authentication:**    * A random challenge is generated, and the user is required to respond by hashing the challenge with a shared secret.    * The server verifies the response against its own computed value. 4. **MFA Workflow:**    * The three authentication steps are combined into the multifactor\_authentication function.    * Each step (password, OTP, challenge-response) is validated sequentially. If any step fails, authentication fails.  ****How MFA Works in This System:****  * **Factor 1: Password** — The user must provide a correct password (with salted hash). * **Factor 2: OTP** — After the password is validated, an OTP is generated and the user must enter it correctly. * **Factor 3: Challenge-Response** — Finally, a random challenge is presented to the user, and they must respond with the correct answer based on a shared secret.  ****Conclusion**:** This MFA system provides a robust way of verifying users through multiple authentication methods. By combining ****password authentication with salt****, ****OTP****, and ****challenge-response authentication****, we are implementing a multifactor authentication mechanism that significantly enhances security by ensuring that an attacker must compromise multiple factors to gain unauthorized access. |

|  |
| --- |
| **Conclusion:** |
| This MFA system enhances security by combining password authentication, OTP, and challenge-response mechanisms, ensuring multiple layers of user verification. |