SET 1

**Q1. If you receive an encrypted message and the encryption algorithm is AES with a 128-bit key, how would you decrypt it if you only have the ciphertext and the key?**

**1. Understanding AES Decryption**

AES (Advanced Encryption Standard) is a **symmetric encryption algorithm**, meaning the same key is used for both **encryption and decryption**. Since you have the **ciphertext and key**, you can decrypt the message using the appropriate AES mode.

**2. Steps for Decrypting AES-128 Ciphertext**

To decrypt the message, you need:

* **Ciphertext** (the encrypted message)
* **AES-128 key** (128-bit key = 16 bytes)
* **Encryption mode** (ECB, CBC, GCM, etc.)
* **IV (Initialization Vector)** (for CBC, GCM modes)

**Step 1: Determine the AES Mode**

* If it's **ECB (Electronic Codebook)** mode, decryption is straightforward.
* If it's **CBC (Cipher Block Chaining)** mode, you need the **IV** (typically stored with the ciphertext).
* If it's **GCM (Galois/Counter Mode)**, you need **IV and authentication tag**.

**Step 2: Decrypt Using the Key**

* Use an AES decryption library such as **OpenSSL, Python (PyCryptodome), or Java (javax.crypto.Cipher)**.
* Example using Python (PyCryptodome):

from Crypto.Cipher import AES

key = b'sixteen\_byte\_key' # 128-bit key

iv = b'16byte\_iv\_vector' # Required for CBC mode

cipher = AES.new(key, AES.MODE\_CBC, iv)

decrypted\_text = cipher.decrypt(ciphertext)

print(decrypted\_text.decode())

* Ensure **padding is removed** after decryption (PKCS#7 or Zero Padding).

**3. Considerations & Challenges**

* If the **AES mode is unknown**, try **common modes** like **ECB, CBC, or GCM**.
* If the **IV is missing**, brute-force attacks on the IV may be required (for CBC).
* If an **authentication tag is present**, ensure it is **verified before decryption** (GCM).
* If the **key is incorrect**, decryption will produce **random gibberish**.

**4. Security Implications**

* If an attacker has both the ciphertext and key, the **message is completely compromised**.
* AES encryption is **secure only when the key is secret**.

**Q2. You need to ensure that a message hasn’t been tampered with during transmission. What cryptographic techniques would you use to verify the integrity and authenticity of the message?**

To verify that a message **has not been modified**, you can use **cryptographic hash functions and digital signatures**.

**1. Message Integrity Techniques**

**a) Hashing (HMAC)**

* Generate a **cryptographic hash (SHA-256, SHA-512)** before sending the message.
* The receiver recalculates the hash and **compares it**.
* Example:

Sender: H(message) → Send (message, hash)

Receiver: Compute H(message) → Compare with received hash

* If the hash values match, **message integrity is preserved**.

**b) HMAC (Hash-based Message Authentication Code)**

* Uses a **secret key + hash function** (e.g., HMAC-SHA256).
* Prevents attackers from modifying the message and **recomputing a valid hash**.
* Example using Python:

import hmac

import hashlib

message = b"Hello, Secure World!"

key = b"secret\_key"

hmac\_hash = hmac.new(key, message, hashlib.sha256).hexdigest()

print(hmac\_hash)

* The recipient verifies the **HMAC** using the same key.

**2. Message Authenticity Techniques**

**a) Digital Signatures**

* Uses **asymmetric encryption (RSA, ECDSA)** for **authentication & integrity**.
* The **sender signs** the message with a **private key**.
* The **receiver verifies** it using the sender’s **public key**.

**b) Process of Digital Signatures**

1. Sender generates a **hash of the message**.
2. Signs the hash with **their private key**.
3. Sends the **message + digital signature**.
4. Receiver:
   * Hashes the received message.
   * **Decrypts the signature using the sender’s public key**.
   * If both hashes match, the message is **authentic and untampered**.

**c) Example with OpenSSL (RSA Signature)**

openssl dgst -sha256 -sign private\_key.pem -out signature.sig message.txt

openssl dgst -sha256 -verify public\_key.pem -signature signature.sig message.txt

**3. Mitigation Against Tampering**

* Use **TLS (Transport Layer Security)** to prevent MITM (Man-in-the-Middle) attacks.
* Use **Nonce and Timestamps** to prevent replay attacks.

**✅ Summary of Techniques:**

| **Goal** | **Technique** |
| --- | --- |
| Ensure message integrity | **Hashing (SHA-256, SHA-512)** |
| Prevent modification | **HMAC (with a secret key)** |
| Ensure authenticity | **Digital Signatures (RSA, ECDSA)** |
| Secure transmission | **TLS, Encrypted Channels** |

These methods ensure that the message remains **unaltered and from a legitimate sender**. 🚀

SET 2

**Q1. Given a piece of data that has been encrypted using RSA, how would you decrypt it if you only had the private key? What considerations should you keep in mind?**

**1. Understanding RSA Decryption**

RSA (**Rivest-Shamir-Adleman**) is an **asymmetric encryption algorithm** that uses:

* A **public key** for encryption.
* A **private key** for decryption.

If you have the **private key**, you can decrypt the RSA-encrypted ciphertext.

**2. Steps to Decrypt an RSA-encrypted Message**

* Ensure the **private key is in PEM format**.
* Use an **RSA decryption tool** (e.g., OpenSSL, Python, Java).

**Example: Using OpenSSL to Decrypt RSA Encrypted Data**

openssl rsautl -decrypt -inkey private\_key.pem -in encrypted\_data.bin -out decrypted\_data.txt

**Example: Using Python (PyCryptodome)**

from Crypto.PublicKey import RSA

from Crypto.Cipher import PKCS1\_OAEP

import base64

# Load private key

private\_key = RSA.import\_key(open("private\_key.pem").read())

# Initialize decryption

cipher\_rsa = PKCS1\_OAEP.new(private\_key)

# Decrypt message

ciphertext = base64.b64decode("ENCRYPTED\_MESSAGE")

decrypted\_message = cipher\_rsa.decrypt(ciphertext)

print(decrypted\_message.decode())

**3. Considerations When Decrypting**

| **Factor** | **Description** |
| --- | --- |
| **Key Size** | Ensure the key size matches the encryption settings (e.g., 2048-bit, 4096-bit). |
| **Padding Scheme** | RSA uses **PKCS1 v1.5 or OAEP padding**. The decryption process must match the original padding used. |
| **Key Format** | The private key should be in a readable format (PEM, DER). |
| **Performance** | RSA decryption is computationally expensive; it’s often used to encrypt **small** data like session keys (not large files). |
| **Security** | Ensure the private key is **securely stored** and not exposed. |

If the ciphertext was **signed** (instead of encrypted), use **digital signature verification** instead of decryption.

**Q2. In a public-key infrastructure (PKI) system, how do you verify the authenticity of a digital certificate before trusting it?**

**1. Understanding Digital Certificates**

A **digital certificate** (e.g., X.509 SSL/TLS certificate) ensures **trust** in **public-key cryptography**. It is issued by a **Certificate Authority (CA)** and contains:

* **Public Key** of the entity.
* **Certificate Authority (CA) Signature**.
* **Certificate Owner Information**.
* **Expiration Date**.

To **verify** a digital certificate’s authenticity, follow these steps:

**2. Steps to Verify a Digital Certificate**

| **Step** | **Action** |
| --- | --- |
| **Step 1: Check Validity** | Verify if the certificate is **not expired** or **revoked** (CRL/OCSP). |
| **Step 2: Check Chain of Trust** | Ensure the certificate was issued by a trusted **CA (Certificate Authority)**. |
| **Step 3: Verify Digital Signature** | Use **public key cryptography** to check the CA’s signature on the certificate. |
| **Step 4: Check Certificate Transparency** | Look for logs that confirm it was **legitimately issued**. |

**Example: Using OpenSSL to Verify a Certificate**

openssl verify -CAfile ca\_certificate.pem server\_certificate.pem

**Example: Checking Certificate Details**

openssl x509 -in certificate.pem -text -noout

This will display:

* Issuer (CA Name)
* Validity Period
* Subject Name
* Public Key
* Signature Algorithm

**3. Revocation Checking**

A certificate might be **revoked** before its expiry due to compromise. Check:

* **Certificate Revocation List (CRL)**
* **Online Certificate Status Protocol (OCSP)**

**Example: Checking OCSP Status**

openssl ocsp -issuer ca\_certificate.pem -cert server\_certificate.pem -text -url http://ocsp.example.com

**Summary**

| **Verification Method** | **Description** |
| --- | --- |
| **Check Expiry Date** | Ensure the certificate is still valid. |
| **Validate CA Signature** | Use OpenSSL to verify authenticity. |
| **Confirm Chain of Trust** | Ensure the certificate links back to a trusted CA. |
| **Check Revocation** | Use **CRL or OCSP** to ensure the certificate is still active. |

**✅ Key Takeaways**

* **RSA private keys** allow **decryption** of messages **encrypted with the corresponding public key**.
* **PKI verification** ensures **digital certificates are legitimate**, preventing **MITM attacks**.
* Always **use OpenSSL or a trusted tool** to validate certificates before trusting them.

SET 3

**Q1. You are tasked with securely transmitting data over an insecure channel. How would you combine symmetric and asymmetric encryption to ensure confidentiality and integrity?**

**1. Hybrid Encryption Approach**

To securely transmit data over an **insecure channel**, we use a **hybrid encryption system** that combines:

1. **Asymmetric Encryption (RSA, ECC)** – Used to encrypt a symmetric key securely.
2. **Symmetric Encryption (AES, ChaCha20)** – Used for encrypting the actual message efficiently.

This approach ensures both **confidentiality** (only the recipient can read the message) and **integrity** (message has not been tampered with).

**2. Steps in Hybrid Encryption**

| **Step** | **Description** |
| --- | --- |
| **Step 1: Generate a Symmetric Key** | Use AES-256 or ChaCha20 to generate a random session key. |
| **Step 2: Encrypt the Data** | Encrypt the plaintext message using the symmetric key. |
| **Step 3: Encrypt the Symmetric Key** | Encrypt the symmetric key using the recipient’s public key (RSA/ECC). |
| **Step 4: Send Both Encrypted Parts** | Send the **encrypted symmetric key + encrypted message** to the recipient. |
| **Step 5: Decrypt at Recipient Side** | - Decrypt the symmetric key using their private key. - Decrypt the message using the decrypted symmetric key. |

**3. Example of Hybrid Encryption**

**Python Example Using PyCryptodome (AES + RSA)**

from Crypto.Cipher import AES, PKCS1\_OAEP

from Crypto.PublicKey import RSA

import os

# Generate a random AES key

aes\_key = os.urandom(32)

# Load recipient's public key

recipient\_key = RSA.import\_key(open("public\_key.pem").read())

rsa\_cipher = PKCS1\_OAEP.new(recipient\_key)

# Encrypt AES key with RSA

encrypted\_aes\_key = rsa\_cipher.encrypt(aes\_key)

# Encrypt data using AES

aes\_cipher = AES.new(aes\_key, AES.MODE\_GCM)

ciphertext, tag = aes\_cipher.encrypt\_and\_digest(b"Secret message")

# Send: (encrypted\_aes\_key, ciphertext, tag, nonce)

**4. Ensuring Integrity**

* Use **HMAC (Hash-based Message Authentication Code)** for integrity.
* Use **digital signatures (RSA/ECDSA)** to verify authenticity.

**Example: HMAC for Integrity**

import hmac, hashlib

key = b'shared\_secret\_key'

message = b'Secret message'

hmac\_hash = hmac.new(key, message, hashlib.sha256).hexdigest()

print(hmac\_hash)

**5. Why Use Hybrid Encryption?**

| **Feature** | **Symmetric Encryption** | **Asymmetric Encryption** |
| --- | --- | --- |
| **Speed** | Fast | Slow |
| **Security** | Key exchange problem | Secure key exchange |
| **Best Use** | Encrypt large data | Encrypt small data (keys) |

By combining both methods, **hybrid encryption provides high security with efficiency**.

**Q2. Explain how a man-in-the-middle (MITM) attack could be carried out on an unencrypted communication channel. How would you mitigate this risk using SSL/TLS?**

**1. What is a Man-in-the-Middle (MITM) Attack?**

A **MITM attack** occurs when an attacker intercepts and manipulates communications between two parties **without their knowledge**. This allows the attacker to:

* **Eavesdrop** on sensitive data (usernames, passwords, bank details).
* **Modify** messages (changing account numbers in transactions).
* **Impersonate** one or both parties (phishing, session hijacking).

**2. How MITM Attacks Work on an Unencrypted Channel**

1. **Sniffing** – Attackers capture packets using tools like Wireshark.
2. **Session Hijacking** – Attackers steal cookies to impersonate users.
3. **Packet Injection** – Attackers modify HTTP requests/responses.
4. **DNS Spoofing** – Redirects victims to fake websites.
5. **SSL Stripping** – Downgrades HTTPS to HTTP to intercept credentials.

**Example: MITM Attack Using Wireshark**

1. Attackers connect to the same **Wi-Fi network** as the victim.
2. They use **packet sniffing tools** to capture unencrypted traffic.
3. If **HTTP** (not HTTPS) is used, credentials and sensitive data can be stolen.

**3. Mitigating MITM Attacks Using SSL/TLS**

To **prevent MITM attacks**, **SSL/TLS (Secure Sockets Layer / Transport Layer Security)** must be used.

| **Defense Mechanism** | **How It Works** |
| --- | --- |
| **Encryption** | TLS encrypts communication, preventing eavesdropping. |
| **Authentication** | Certificates verify server authenticity to prevent impersonation. |
| **Integrity Checks** | HMAC and digital signatures detect message tampering. |
| **Perfect Forward Secrecy (PFS)** | Uses **ephemeral keys** to prevent key compromise. |

**Example: Enforcing HTTPS with HSTS (Strict Transport Security)**

To prevent **SSL stripping**, enable **HSTS** in the web server.

Header always set Strict-Transport-Security "max-age=31536000; includeSubDomains; preload"

This forces **all** future connections to use **HTTPS only**.

**Example: Checking SSL Certificate Validity**

openssl s\_client -connect example.com:443 -showcerts

**4. Summary of MITM Attack Prevention**

| **Mitigation Technique** | **Description** |
| --- | --- |
| **Use TLS 1.3** | Ensures encrypted connections with better security. |
| **Verify SSL Certificates** | Prevents fake certificate attacks. |
| **HSTS (HTTP Strict Transport Security)** | Forces all HTTP traffic to HTTPS. |
| **DNSSEC (Domain Name System Security Extensions)** | Prevents DNS spoofing. |
| **Use VPN or Encrypted Networks** | Protects against Wi-Fi sniffing attacks. |

**✅ Key Takeaways**

* **Hybrid encryption (AES + RSA)** ensures **secure communication** over **insecure networks**.
* **MITM attacks** occur when attackers intercept unencrypted traffic.
* **SSL/TLS encryption, HSTS, and certificate validation** prevent MITM attacks.

SET 4

**Q1. How would you perform a brute-force attack on an AES-encrypted file, and what factors would determine whether the attack would be successful or not?**

**1. What is a Brute-Force Attack?**

A **brute-force attack** involves systematically trying all possible keys until the correct one is found. For **AES (Advanced Encryption Standard)**, this means guessing the correct **128-bit, 192-bit, or 256-bit key**.

**2. Steps to Perform a Brute-Force Attack on AES**

1. **Obtain the Encrypted File** – Get the AES-encrypted ciphertext.
2. **Identify AES Mode** – Determine if AES is used in **ECB, CBC, GCM, or CTR mode** (some modes are more vulnerable).
3. **Choose a Brute-Force Method**:
   * **Dictionary Attack** – If the key is weak (e.g., user-generated passwords).
   * **Exhaustive Key Search** – Try all possible key combinations.
4. **Use GPU Acceleration** – Leverage **hashcat, John the Ripper, or custom Python scripts** to speed up attacks.
5. **Verify Decryption** – Check if the decrypted output makes sense.

**Example: Brute-Force AES Using Python**

from Crypto.Cipher import AES

import itertools

ciphertext = b'...' # AES encrypted text

iv = b'...' # Initialization vector (if CBC mode)

# Trying all possible 128-bit keys (not practical, just for small key space)

for key in itertools.product(range(256), repeat=16):

key\_bytes = bytes(key)

cipher = AES.new(key\_bytes, AES.MODE\_CBC, iv)

try:

plaintext = cipher.decrypt(ciphertext)

if b'known text' in plaintext: # Check for expected output

print("Key found:", key\_bytes)

break

except:

continue

**3. Factors That Affect Brute-Force Success**

| **Factor** | **Impact** |
| --- | --- |
| **Key Length (128-bit, 192-bit, 256-bit)** | AES-128 has **2¹²⁸** possible keys, making brute force infeasible. AES-256 is even stronger. |
| **Use of Secure Key** | A **randomly generated** key is much harder to brute-force than a user-chosen password. |
| **Hardware (GPUs, Quantum Computing)** | **Quantum computers** (Grover’s Algorithm) can reduce key search time but are not yet practical. |
| **Attack Method (Dictionary vs Exhaustive)** | If a **weak password-based key** is used, dictionary attacks might work. |
| **AES Mode (ECB, CBC, GCM)** | **ECB is weaker** (patterns remain visible), while **CBC and GCM add complexity**. |

**Key Takeaways**

* Brute-force is impractical for **128-bit or higher AES keys** unless the key is weak.
* **Dictionary attacks** can succeed if AES keys are derived from passwords.
* **Use long, random keys** and **KDFs (PBKDF2, bcrypt, Argon2)** to protect against brute-force.

**Q2. Given a scenario where an attacker has intercepted encrypted traffic between two parties, what cryptographic techniques or vulnerabilities could they exploit to decrypt the data?**

**1. Attacks on Encrypted Traffic**

If an attacker intercepts encrypted traffic, they might try the following techniques:

| **Attack Type** | **Description** |
| --- | --- |
| **Brute-force Attack** | Attempting to decrypt captured data by guessing encryption keys. |
| **Key Reuse Attack** | If the same key is used across multiple messages, statistical analysis can reveal plaintext patterns. |
| **Padding Oracle Attack** | Exploits weaknesses in CBC mode padding to decrypt messages byte by byte. |
| **Man-in-the-Middle Attack (MITM)** | Modifies data in transit or performs SSL stripping to downgrade security. |
| **Replay Attack** | Re-sends old encrypted messages to trick the recipient. |
| **Quantum Computing (Future Threat)** | Grover’s Algorithm could halve brute-force time for symmetric encryption. |

**2. Exploiting Weaknesses in Cryptographic Protocols**

| **Vulnerability** | **How It Works** | **Mitigation** |
| --- | --- | --- |
| **Weak Ciphers (e.g., DES, 3DES, RC4)** | Older ciphers have small key sizes or known vulnerabilities. | Use AES-256 or ChaCha20. |
| **TLS 1.0/1.1 (Deprecated)** | Lacks modern security features, allowing downgrade attacks. | Enforce TLS 1.2+ or TLS 1.3. |
| **RSA Key Exchange (No Forward Secrecy)** | If an RSA key is leaked, past traffic can be decrypted. | Use **ECDHE** (Ephemeral Diffie-Hellman) for **Perfect Forward Secrecy (PFS)**. |
| **Public Key Misuse (Key Exchange Attacks)** | If the same RSA key is used for signing and encryption, attackers can forge messages. | Use separate keys for signing and encryption. |

**3. Real-World Examples of Attacks on Encrypted Traffic**

| **Attack** | **How It Works** |
| --- | --- |
| **BEAST Attack (Browser Exploit Against SSL/TLS)** | Exploits weaknesses in **TLS 1.0** CBC mode to decrypt HTTPS traffic. |
| **POODLE Attack (Padding Oracle On Downgraded Legacy Encryption)** | Forces clients to downgrade from **TLS** to **SSL 3.0**, which is vulnerable to padding attacks. |
| **Heartbleed (OpenSSL Vulnerability, 2014)** | Leaked sensitive information from server memory due to a flaw in **TLS heartbeat extension**. |

**4. How to Secure Encrypted Communications**

| **Best Practice** | **Why It’s Important** |
| --- | --- |
| **Use TLS 1.3** | Removes outdated ciphers and reduces attack surface. |
| **Enable Forward Secrecy (PFS)** | Prevents past messages from being decrypted if keys are compromised. |
| **Use Strong Encryption (AES-256, ChaCha20, ECDSA)** | Protects against brute-force and key reuse attacks. |
| **Verify Digital Certificates** | Prevents MITM attacks with rogue certificates. |
| **HMAC for Message Integrity** | Prevents message tampering in transit. |

**Example: Checking TLS Security of a Website**

Run this command to check a website’s TLS security:

openssl s\_client -connect example.com:443 -showcerts

**✅ Key Takeaways**

* **Brute-forcing AES is infeasible** unless the key is weak.
* **MITM attacks, padding oracle attacks, and key reuse vulnerabilities** can allow attackers to decrypt traffic.
* **TLS 1.3, forward secrecy, and strong key management** help mitigate risks.

SET 5

**Q1. Explain how elliptic curve cryptography (ECC) compares to RSA in terms of key size and security. In what scenarios would ECC be preferred over RSA?**

**1. ECC vs. RSA: Key Size and Security**

Elliptic Curve Cryptography (ECC) and RSA are both **public-key encryption algorithms**, but ECC provides **stronger security with smaller key sizes**.

| **Factor** | **RSA** | **ECC** |
| --- | --- | --- |
| **Key Size** | Requires **2048-bit or higher** for strong security. | A **256-bit ECC key** provides equivalent security to a **3072-bit RSA key**. |
| **Performance** | Slower for **key generation, encryption, and decryption**. | Faster due to **smaller keys and lower computational cost**. |
| **Storage & Bandwidth** | Larger keys require more **storage** and **transmission bandwidth**. | Smaller keys **reduce bandwidth** and **storage requirements**. |
| **Security Strength** | RSA-2048 is currently secure, but **RSA-1024 is breakable**. | ECC-256 is highly secure, even against quantum attacks. |
| **Quantum Resistance** | Not quantum-resistant (Shor’s algorithm can break it). | Not fully quantum-resistant, but harder to break than RSA. |

🔹 **Example Comparison of Key Size Equivalents**

* **RSA-2048 ≈ ECC-224**
* **RSA-3072 ≈ ECC-256**
* **RSA-7680 ≈ ECC-384**
* **RSA-15360 ≈ ECC-521**

**2. When is ECC Preferred Over RSA?**

ECC is **preferred in environments** where **performance, storage, and security efficiency** are important:

| **Use Case** | **Why ECC is Better** |
| --- | --- |
| **Mobile Devices & IoT** | ECC consumes less power and works efficiently on constrained devices. |
| **Web Security (TLS/SSL)** | ECC provides faster SSL handshakes and reduces load on servers. |
| **Blockchain & Cryptocurrencies** | Bitcoin, Ethereum, and other cryptocurrencies use ECC (e.g., **secp256k1 curve**). |
| **Government & Military Applications** | Used in **NSA's Suite B Cryptography** for high-security systems. |
| **Messaging Apps (WhatsApp, Signal)** | Secure messaging apps use ECC-based encryption (e.g., **X25519 curve**). |

✅ **Conclusion**: ECC is preferred over RSA in scenarios requiring **high security with minimal computational overhead**.

**Q2. How would you implement secure password storage using hashing and salting in a web application? What algorithms would you choose and why?**

**1. Why Use Hashing and Salting for Password Storage?**

Storing passwords in plaintext is a major security risk. **Hashing** converts passwords into **irreversible hashes**, and **salting** adds random data to prevent **rainbow table attacks**.

**2. Steps to Securely Store Passwords**

1. **Use a Strong Hashing Algorithm (bcrypt, Argon2, PBKDF2, scrypt)**
2. **Generate a Unique Salt for Each Password**
3. **Store the Salt + Hashed Password in the Database**
4. **Use a Key Derivation Function (KDF) to Slow Down Attacks**
5. **Verify Passwords Securely During Login**

**3. Best Algorithms for Password Hashing**

| **Algorithm** | **Why It’s Secure?** | **Use Case** |
| --- | --- | --- |
| **Argon2** (Recommended) | Winner of the **Password Hashing Competition (PHC)**. **Resistant to GPU and ASIC attacks**. | Best for modern systems. |
| **bcrypt** | Adds an adaptive work factor to slow brute-force attacks. | Used in **Django, Laravel**. |
| **PBKDF2** | Uses **multiple iterations** to slow down attackers. | Used in **Apple iOS Keychain**. |
| **scrypt** | Memory-hard function that resists **ASIC brute-force**. | Used in **Litecoin, Dogecoin**. |

🚀 **Example: Hashing a Password Using bcrypt (Python)**

import bcrypt

# Generate a hashed password

password = b"SecurePassword123"

salt = bcrypt.gensalt()

hashed\_password = bcrypt.hashpw(password, salt)

print("Hashed Password:", hashed\_password)

# Verify Password

input\_password = b"SecurePassword123"

if bcrypt.checkpw(input\_password, hashed\_password):

print("Password is valid!")

else:

print("Invalid password!")

**4. Additional Security Measures**

| **Measure** | **Purpose** |
| --- | --- |
| **Peppering** | Adding a server-side secret key before hashing for extra security. |
| **Rate Limiting** | Prevents attackers from making unlimited login attempts. |
| **HMAC (Hash-based Message Authentication Code)** | Prevents tampering with stored hashes. |
| **Zero-Knowledge Password Proofs (ZKPs)** | Secure authentication without exposing passwords. |

**✅ Key Takeaways**

* **ECC is more secure and efficient** than RSA, making it ideal for modern cryptography.
* **Argon2, bcrypt, PBKDF2, and scrypt** are the best choices for secure password hashing.
* **Salting, key stretching, and rate limiting** help protect against brute-force attacks.