Week 9 revision notes

IFB240 Week 9 - Asymmetric Cryptography

Overview

Asymmetric cryptography, also known as **public-key cryptography**, uses a **pair of keys**: one for encryption and one for decryption. This provides a solution to the key distribution problem that arises in symmetric cryptography, especially in large systems. Asymmetric cryptography is used for both **confidentiality** and **authentication**, and it plays a crucial role in **digital signatures** and **secure communications**.

---

1. Introduction to Asymmetric Cryptography

What is Asymmetric Cryptography?

- **Asymmetric Cryptography** uses two related keys: a **public key** (known to everyone) and a **private key** (kept secret by the owner).
- **Public-Key Cryptography**:
  - The **public key** is used for encryption.
  - The **private key** is used for decryption.
- This approach contrasts with **symmetric cryptography**, where the same key is used for both encryption and decryption.

Key Components

- **Public Key**: Shared openly, used to encrypt messages.
- **Private Key**: Kept secret, used to decrypt messages or create digital signatures.

Solving the Key Distribution Problem

- In symmetric cryptography, secure key distribution is challenging because each pair of users needs a unique shared secret key.
- In **asymmetric systems**, each participant needs only one **key pair**:
  - **Public key** can be shared openly.
  - **Private key** remains confidential, solving the key distribution challenge in large networks.

---

2. Uses of Asymmetric Cryptography

Confidentiality

- **Encryption for Confidentiality**:
  - **How to Use**:
    - To encrypt a message with **asymmetric cryptography**, follow these steps:
      1. **Generate Key Pair**:
         - Use an algorithm like **RSA** or **ECC** to generate the key pair.
         - In Python, you can use the `cryptography` library:

         ```
         from cryptography.hazmat.primitives.asymmetric import rsa

         private_key = rsa.generate_private_key(
             public_exponent=65537,
             key_size=2048,
         )
         public_key = private_key.public_key()
         ```

      2. **Encrypt the Message**:
         - Use the recipient's **public key** to encrypt the plaintext.

         ```
         from cryptography.hazmat.primitives import serialization, hashes
         from cryptography.hazmat.primitives.asymmetric import padding

         ciphertext = public_key.encrypt(
             b"Secret Message",
             padding.OAEP(
                 mgf=padding.MGF1(algorithm=hashes.SHA256()),
                 algorithm=hashes.SHA256(),
                 label=None
             )
         )
         ```

      3. **Send Ciphertext**:
         - Send the **ciphertext** to the recipient.
    - **Decryption**:
      1. The recipient decrypts the ciphertext using their **private key**:

      ```
      plaintext = private_key.decrypt(
          ciphertext,
          padding.OAEP(
              mgf=padding.MGF1(algorithm=hashes.SHA256()),
              algorithm=hashes.SHA256(),
              label=None
      ```

```
    )
)
print(plaintext)
```

- **Outcome**: The encrypted message is unreadable by anyone except the intended recipient, providing **confidentiality**.

Authentication and Digital Signatures

- **Digital Signatures**:
  - **How to Use**:
    - To sign a message and verify its authenticity:
      1. **Generate a Signature**:
         - Alice signs a message with her **private key**.

         ```
         from cryptography.hazmat.primitives import hashes
         from cryptography.hazmat.primitives.asymmetric import padding

         message = b"This is Alice's message."
         signature = private_key.sign(
             message,
             padding.PSS(
                 mgf=padding.MGF1(hashes.SHA256()),
                 salt_length=padding.PSS.MAX_LENGTH
             ),
             hashes.SHA256()
         )
         ```

      2. **Verify the Signature**:
         - Bob, upon receiving the message, verifies the signature using **Alice's public key**.

         ```
         try:
             public_key.verify(
                 signature,
                 message,
                 padding.PSS(
                     mgf=padding.MGF1(hashes.SHA256()),
                     salt_length=padding.PSS.MAX_LENGTH
                 ),
                 hashes.SHA256()
             )
             print("Signature is valid.")
         ```

```
except:
    print("Signature is invalid.")
```

- **Outcome**: The verification process proves the authenticity of the sender and ensures that the message has not been altered in transit.

---

3. Common Algorithms and Key Features

RSA (Rivest-Shamir-Adleman)

- **Key Features**:
  - Developed in 1977 at MIT.
  - Based on the **difficulty of factoring large numbers**.
  - **Key Pair Generation**:
    - Choose two large primes (p) and (q).
    - Calculate (n = p \times q).
    - Choose a public exponent (e) and compute a private key (d).
  - **Usage**: RSA can be used both for encryption and for creating digital signatures.
  - **Security**: RSA is secure as long as the factorization of (n) remains computationally infeasible.

Elliptic Curve Cryptography (ECC)

- **Key Features**:
  - Uses algebraic structures of elliptic curves.
  - **Advantages**:
    - Provides equivalent security to RSA but with much **smaller key sizes**, making it more efficient.
  - **How to Generate Keys**:

```
from cryptography.hazmat.primitives.asymmetric import ec

private_key = ec.generate_private_key(ec.SECP256R1())
public_key = private_key.public_key()
```

Key Takeaway

- **RSA** and **ECC** are the most widely used asymmetric algorithms, providing **confidentiality**, **integrity**, and **non-repudiation**.

---

4. Practical Use Cases of Asymmetric Cryptography

Secure Website Access (HTTPS)

- **How It Works**:
  - HTTPS uses a combination of **asymmetric and symmetric cryptography**.
  - **Step 1**: During the **TLS handshake**, the server shares its **public key** with the client.
  - **Step 2**: The client generates a **session key** (symmetric key) and encrypts it with the server's **public key**.
  - **Step 3**: The server decrypts the session key with its **private key**.
  - **Step 4**: Both parties use the session key for the rest of the communication, ensuring efficiency.

Email Encryption (PGP)

- **Pretty Good Privacy (PGP)**:
  - **How to Use**:
    1. **Key Generation**: Users generate a **public-private key pair**.
    2. **Sharing Public Key**: Exchange public keys to securely communicate.
    3. **Encrypting Emails**: Encrypt email content with the recipient's **public key**.
    4. **Decrypting Emails**: Recipient uses their **private key** to decrypt.

Digital Certificates

- **Purpose**: Authenticate the identity of the owner.
- **Example**:
  - **SSL/TLS Certificates**: Issued by a **Certificate Authority (CA)** to verify the identity of a website.
  - **Verification Process**:
    - The browser checks the certificate's signature using the CA's public key.
    - This proves the legitimacy of the website and establishes a secure connection.

---

5. Post-Quantum Security and Future Considerations

Quantum Computing and Cryptography

- **Quantum Threats**:
  - Quantum computers, using phenomena like **superposition** and **entanglement**, could potentially solve problems that are computationally infeasible for classical computers.
  - As a result, asymmetric schemes like **RSA** and **ElGamal** could be broken by quantum computers.
- **Post-Quantum Cryptography (PQC)**:
  - The **NIST** launched a program in 2016 to identify suitable **PQC algorithms**.

    - In 2022, **four algorithms** were selected (one for key establishment and three for digital signatures).
- **Enterprise Migration**:
    - Organizations need to start planning for migration to **post-quantum cryptosystems** to ensure the future security of their assets.

6. Summary and Key Takeaways

- **Asymmetric Cryptography** uses a **pair of keys** (public and private) to solve the key distribution problem found in symmetric cryptography.
- **RSA, ElGamal, and ECC** are key asymmetric algorithms that provide **confidentiality**, **integrity**, **authentication**, and **non-repudiation**.
- **Digital Signatures** provide a method to authenticate the sender and ensure that messages are not altered.
- **Quantum Computing** presents a future threat, pushing for the development of **post-quantum cryptographic algorithms**.
- **Hybrid Cryptosystems** combine the advantages of both symmetric and asymmetric cryptography to provide a balance between **security** and **efficiency**.

Practical Tips

- Use **RSA** or **ECC** for secure communications but consider moving to **post-quantum algorithms** in the future.
- When implementing **digital signatures**, use robust **hash functions** to ensure integrity and security.
- **Hybrid Encryption** (e.g., TLS) is the most efficient method to combine asymmetric and symmetric cryptography for real-world use.

These notes provide an in-depth understanding of asymmetric cryptography, practical implementation instructions, and the knowledge needed to effectively apply these methods in cybersecurity.