Criptografia: Implementações em Python e Java

Observação de segurança

Os exemplos usam AES-GCM, RSA-OAEP, RSA-PSS e SHA-256. Em produção, proteja chaves privadas, registr
nonces/IVs, e use bibliotecas atualizadas. Para senhas, prefira PBKDF2/Argon2/scrypt.

1) Python (crypto_demo.py) - dependência: cryptography

Instalação:
```
pip install cryptography
```

Código (salve como crypto_demo.py):

```
from cryptography.hazmat.primitives import hashes, serialization
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives import hmac
import os
import hashlib

# -------------------------
# Função Hash (SHA-256)
# -------------------------
def sha256_hash(data: bytes) -> str:
    h = hashlib.sha256()
    h.update(data)
    return h.hexdigest()

# -------------------------
# Criptografia Simétrica AES-GCM
# -------------------------
def generate_aes_key(length=32):
    return os.urandom(length)

def aes_gcm_encrypt(key: bytes, plaintext: bytes, associated_data: bytes = None):
    aesgcm = AESGCM(key)
    nonce = os.urandom(12)
    ct = aesgcm.encrypt(nonce, plaintext, associated_data)
    return nonce, ct

def aes_gcm_decrypt(key: bytes, nonce: bytes, ciphertext: bytes, associated_data: bytes = None):
    aesgcm = AESGCM(key)
    pt = aesgcm.decrypt(nonce, ciphertext, associated_data)
    return pt

# -------------------------
# Criptografia Assimétrica RSA
# -------------------------
def generate_rsa_keypair(key_size=2048):
    private_key = rsa.generate_private_key(public_exponent=65537, key_size=key_size)
    public_key = private_key.public_key()
    return private_key, public_key

def rsa_encrypt(public_key, plaintext: bytes):
    ct = public_key.encrypt(
        plaintext,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
```

```python
        )
    )
    return ct

def rsa_decrypt(private_key, ciphertext: bytes):
    pt = private_key.decrypt(
        ciphertext,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )
    return pt

def rsa_sign(private_key, message: bytes):
    signature = private_key.sign(
        message,
        padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
    )
    return signature

def rsa_verify(public_key, message: bytes, signature: bytes):
    try:
        public_key.verify(
            signature,
            message,
            padding.PSS(
                mgf=padding.MGF1(hashes.SHA256()),
                salt_length=padding.PSS.MAX_LENGTH
            ),
            hashes.SHA256()
        )
        return True
    except Exception:
        return False

# -------------------------
# Exemplo rápido
# -------------------------
if __name__ == "__main__":
    msg = b"Mensagem secreta de teste"
    aad = b"header-autenticado"

    print("=== HASH SHA-256 ===")
    print(sha256_hash(msg))

    print("\n=== AES-GCM (simetrico) ===")
    aes_key = generate_aes_key(32)
    nonce, ciphertext = aes_gcm_encrypt(aes_key, msg, aad)
    print("nonce:", nonce.hex())
    print("ciphertext:", ciphertext.hex())
    decrypted = aes_gcm_decrypt(aes_key, nonce, ciphertext, aad)
    print("decrypted equals original:", decrypted == msg)

    print("\n=== RSA (assimétrico) ===")
    priv, pub = generate_rsa_keypair(2048)
    ct = rsa_encrypt(pub, msg)
    print("rsa ciphertext len:", len(ct))
    pt = rsa_decrypt(priv, ct)
    print("rsa decrypted equals original:", pt == msg)

    print("\n=== RSA Sign/Verify ===")
    sig = rsa_sign(priv, msg)
    ok = rsa_verify(pub, msg, sig)
    print("assinatura válida?", ok)
```

2) Java (CryptoDemo.java)

Código (salve como CryptoDemo.java):

```
// CryptoDemo.java
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.spec.GCMParameterSpec;
import java.security.*;
import java.security.spec.MGF1ParameterSpec;
import java.security.spec.PSSParameterSpec;
import java.util.Base64;
import javax.crypto.spec.SecretKeySpec;
import java.nio.charset.StandardCharsets;
import java.security.spec.OAEPParameterSpec;
import java.security.spec.PSource;

public class CryptoDemo {
    // ----- Hash SHA-256 -----
    public static String sha256(String input) throws Exception {
        MessageDigest md = MessageDigest.getInstance("SHA-256");
        byte[] digest = md.digest(input.getBytes(StandardCharsets.UTF_8));
        return bytesToHex(digest);
    }

    // ----- AES-GCM -----
    public static SecretKey generateAesKey(int bits) throws Exception {
        KeyGenerator kg = KeyGenerator.getInstance("AES");
        kg.init(bits);
        return kg.generateKey();
    }

    public static class AesGcmResult {
        public byte[] iv;
        public byte[] ciphertext;
        public AesGcmResult(byte[] iv, byte[] ciphertext) { this.iv = iv; this.ciphertext = cipherte
    }

    public static AesGcmResult aesGcmEncrypt(SecretKey key, byte[] plaintext, byte[] aad) throws Exc
        Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding");
        byte[] iv = new byte[12];
        SecureRandom sr = new SecureRandom();
        sr.nextBytes(iv);
        GCMParameterSpec spec = new GCMParameterSpec(128, iv);
        cipher.init(Cipher.ENCRYPT_MODE, key, spec);
        if (aad != null) cipher.updateAAD(aad);
        byte[] ct = cipher.doFinal(plaintext);
        return new AesGcmResult(iv, ct);
    }

    public static byte[] aesGcmDecrypt(SecretKey key, byte[] iv, byte[] ciphertext, byte[] aad) thro
Exception {
        Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding");
        GCMParameterSpec spec = new GCMParameterSpec(128, iv);
        cipher.init(Cipher.DECRYPT_MODE, key, spec);
        if (aad != null) cipher.updateAAD(aad);
        return cipher.doFinal(ciphertext);
    }

    // ----- RSA keypair, encrypt/decrypt, sign/verify -----
    public static KeyPair generateRsaKeyPair(int bits) throws Exception {
        KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA");
        kpg.initialize(bits);
        return kpg.generateKeyPair();
    }

    public static byte[] rsaEncrypt(PublicKey pub, byte[] plaintext) throws Exception {
        Cipher cipher = Cipher.getInstance("RSA/ECB/OAEPWithSHA-256AndMGF1Padding");
        cipher.init(Cipher.ENCRYPT_MODE, pub);
        return cipher.doFinal(plaintext);
    }

    public static byte[] rsaDecrypt(PrivateKey priv, byte[] ciphertext) throws Exception {
        Cipher cipher = Cipher.getInstance("RSA/ECB/OAEPWithSHA-256AndMGF1Padding");
        cipher.init(Cipher.DECRYPT_MODE, priv);
        return cipher.doFinal(ciphertext);
    }

    public static byte[] rsaSign(PrivateKey priv, byte[] message) throws Exception {
        Signature sig = Signature.getInstance("RSASSA-PSS");
        PSSParameterSpec pss = new PSSParameterSpec("SHA-256", "MGF1", MGF1ParameterSpec.SHA256, 32,
        sig.setParameter(pss);
        sig.initSign(priv);
        sig.update(message);
        return sig.sign();
    }

    public static boolean rsaVerify(PublicKey pub, byte[] message, byte[] signature) throws Exceptio
        Signature sig = Signature.getInstance("RSASSA-PSS");
        PSSParameterSpec pss = new PSSParameterSpec("SHA-256", "MGF1", MGF1ParameterSpec.SHA256, 32,
        sig.setParameter(pss);
        sig.initVerify(pub);
        sig.update(message);
        return sig.verify(signature);
    }

    // ----- Helper -----
    private static String bytesToHex(byte[] bytes) {
        StringBuilder sb = new StringBuilder();
        for (byte b : bytes) sb.append(String.format("%02x", b));
        return sb.toString();
    }

    public static void main(String[] args) throws Exception {
        byte[] msg = "Mensagem secreta de teste".getBytes(StandardCharsets.UTF_8);
        byte[] aad = "header-autenticado".getBytes(StandardCharsets.UTF_8);

        System.out.println("=== SHA-256 ===");
        System.out.println(sha256(new String(msg, StandardCharsets.UTF_8)));

        System.out.println("\n=== AES-GCM ===");
        SecretKey aesKey = generateAesKey(256);
        AesGcmResult res = aesGcmEncrypt(aesKey, msg, aad);
        System.out.println("IV: " + bytesToHex(res.iv));
        System.out.println("CT: " + bytesToHex(res.ciphertext));
        byte[] pt = aesGcmDecrypt(aesKey, res.iv, res.ciphertext, aad);
        System.out.println("decrypted equals original? " + java.util.Arrays.equals(pt, msg));
        System.out.println("\n=== RSA ===");
        KeyPair kp = generateRsaKeyPair(2048);
        byte[] ct = rsaEncrypt(kp.getPublic(), msg);
        System.out.println("rsa ct len: " + ct.length);
        byte[] dt = rsaDecrypt(kp.getPrivate(), ct);
        System.out.println("rsa decrypted equals original? " + java.util.Arrays.equals(dt, msg));

        System.out.println("\n=== RSA Sign/Verify ===");
        byte[] signature = rsaSign(kp.getPrivate(), msg);
        boolean ok = rsaVerify(kp.getPublic(), msg, signature);
        System.out.println("assinatura válida? " + ok);
    }
}
```

3) Testes / Como verificar

Python: execute `python crypto_demo.py` e verifique saídas. Java: compile com `javac CryptoDemo.java
`java CryptoDemo`.

4) Sugestões para produção

- Para senhas use PBKDF2/Argon2/scrypt.
- Para chaves melhor desempenho, considere ECDSA/Ed25519/ECDH.
- Nunca reusar IVs com AES-GCM.
- Use formatos PEM para salvar chaves privadas e proteja com senha.