ATIVIDADE 5 — Implementar, Codificar e Verificar Algoritmos

Disciplina: Sistemas Computacionais e Segurança – SCS

Aluno: Arthur de Sousa Sales

Professor: Prof. Calvetti

Instituição: USJT

Local: São Caetano do Sul – SP

Objetivo da Atividade

Implementar e verificar: (i) um algoritmo de criptografia simétrica, (ii) um algoritmo de criptografia assimétrica e (iii) uma função hash, em Python, com evidências de execução.

Abaixo seguem os códigos prontos e um roteiro rápido para teste.

Pré-requisitos (execução local)

```
# 1) Instale Python 3
# 2) (Opcional, mas recomendado) Crie um ambiente virtual
# 3) Instale a biblioteca 'cryptography'
pip install cryptography

# 4) No VS Code, crie um arquivo: atividade5.py
# 5) Cole os códigos das três seções abaixo
```

```
# 6) Execute:
python atividade5.py
```

---

1) Criptografia com Chaves Simétricas — AES-256 (modo GCM)

Resumo: usa a mesma chave para cifrar e decifrar; GCM provê confidencialidade e integridade (autenticação).

```
import os
from cryptography.hazmat.primitives.ciphers.aead import AESGCM

def demo_aes_gcm():
    plaintext = b"Exemplo de mensagem confidencial."
    aad = b"contexto-opcional"  # dados associados (não cifrados, mas autenticados)

    key = AESGCM.generate_key(bit_length=256)  # 32 bytes
    nonce = os.urandom(12)  # recomendado p/ GCM

    aesgcm = AESGCM(key)
    ciphertext = aesgcm.encrypt(nonce, plaintext, aad)
    decrypted = aesgcm.decrypt(nonce, ciphertext, aad)

    print("[AES-GCM] Chave(hex):", key.hex())
    print("[AES-GCM] Nonce(hex):", nonce.hex())
    print("[AES-GCM] Ciphertext(hex):", ciphertext.hex())
    print("[AES-GCM] Decifrado:", decrypted.decode())

# Chamada de teste
if __name__ == "__main__":
    demo_aes_gcm()
```

Resultado esperado (exemplo ilustrativo):

```
[AES-GCM] Chave(hex): 2f8c... (32 bytes em hex)
[AES-GCM] Nonce(hex): a3b1... (12 bytes em hex)
[AES-GCM] Ciphertext(hex): 5abf... (dados variam a cada execução)
[AES-GCM] Decifrado: Exemplo de mensagem confidencial.
```

---

2) Criptografia com Chaves Assimétricas — RSA (OAEP) + Assinatura (PSS)

Resumo: usa par de chaves (pública/privada). Criptografia com OAEP/SHA-256; assinatura com PSS/SHA-256.

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

def demo_rsa():
    message = b"Contrato aprovado em 24/10/2025."

    # Gera par de chaves
    private_key = rsa.generate_private_key(public_exponent=65537, key_size=2048)
    public_key = private_key.public_key()

    # (Opcional) exporta em PEM
    pem_priv = private_key.private_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PrivateFormat.PKCS8,
        encryption_algorithm=serialization.NoEncryption(),
    )
    pem_pub = public_key.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo,
    )
    print("[RSA] Chave privada PEM (início):\n", pem_priv.decode().splitlines()[0])
    print("[RSA] Chave pública  PEM (início):\n", pem_pub.decode().splitlines()[0])

    # Criptografa com chave pública (OAEP + SHA-256)
    ciphertext = public_key.encrypt(
        message,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None,
        ),
    )

    # Decifra com chave privada
    decrypted = private_key.decrypt(
        ciphertext,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None,
        ),
    )

    # Assina com chave privada (PSS + SHA-256)
```

```
    signature = private_key.sign(
        message,
        padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH,
        ),
        hashes.SHA256(),
    )

    # Verifica assinatura com chave pública
    try:
        public_key.verify(
            signature,
            message,
            padding.PSS(
                mgf=padding.MGF1(hashes.SHA256()),
                salt_length=padding.PSS.MAX_LENGTH,
            ),
            hashes.SHA256(),
        )
        print("[RSA] Assinatura VÁLIDA (PSS/SHA-256).")
    except Exception as e:
        print("[RSA] Falha na verificação da assinatura:", e)

    print("[RSA] Ciphertext(hex):", ciphertext.hex())
    print("[RSA] Decifrado:", decrypted.decode())

# Chamada de teste adicional
if __name__ == "__main__":
    demo_rsa()
```

Resultado esperado (exemplo ilustrativo):

```
[RSA] Chave privada PEM (início):
 -----BEGIN PRIVATE KEY-----
[RSA] Chave pública  PEM (início):
 -----BEGIN PUBLIC KEY-----
[RSA] Assinatura VÁLIDA (PSS/SHA-256).
[RSA] Ciphertext(hex): 6c1a... (muda a cada execução)
[RSA] Decifrado: Contrato aprovado em 24/10/2025.
```

---

3) Função Hash — SHA-256

Resumo: hash não é cifragem; serve para integridade (detectar modificações).

```
import hashlib

def demo_hash():
    dados1 = b"Arquivo-versao-1"
    dados2 = b"Arquivo-versao-2"

    h1 = hashlib.sha256(dados1).hexdigest()
    h2 = hashlib.sha256(dados2).hexdigest()

    print("[HASH] SHA-256 (v1):", h1)
    print("[HASH] SHA-256 (v2):", h2)
    print("[HASH] Iguais?", h1 == h2)

# Chamada de teste adicional
if __name__ == "__main__":
    demo_hash()
```

Resultado esperado (exemplo ilustrativo):

```
[HASH] SHA-256 (v1): 2e4f... (64 hex)
[HASH] SHA-256 (v2): 91bd... (64 hex)
[HASH] Iguais? False
```

Guia de Execução e Evidências

1) Salve o arquivo como atividade5.py e execute no VS Code (ou terminal).

2) Faça capturas de tela (prints) mostrando as saídas do console para as três seções.

3) Insira os prints abaixo (ou anexe ao final do documento):

[Espaço para print 1] — Cole aqui a captura da saída correspondente.

[Espaço para print 2] — Cole aqui a captura da saída correspondente.

[Espaço para print 3] — Cole aqui a captura da saída correspondente.

Conclusão

Os testes demonstraram: (i) confidencialidade e integridade com AES-GCM; (ii) confidencialidade e autenticidade com RSA (OAEP + PSS); (iii) integridade com SHA-256. Essas construções são a base dos protocolos modernos (TLS/HTTPS, VPNs e assinaturas digitais).