Universidade do Minho

Mestrado em Engenharia Informática

Engenharia de segurança

Ficha de exercício 4

Grupo 1

Rui Carlos Azevedo Carvalho - PG47633
Daniel Barbosa Miranda - PG47123
Ana Luísa Lira Tomé Carneiro - PG46983

Parte VII: Criptografia de chave pública

Pergunta P1.1

O problema 1 consistia em criar um programa que conseguisse cifra e decifrar um ficheiro de texto utilizando um envelope digital. Esta técnica de criptografia é implementada segundo os seguintes passos:

> Sabendo que Kc representa a chave pública do Bob, Kd a chave privada do Bob, E é uma cifra assimétrica e Ek é uma cifra simétrica, temos:
>
> 1. A Alice tem uma mensagem M a enviar ao Bob
> 2. A Alice gera uma chave de sessão - K.
> 3. A Alice envia ao Bob um ficheiro com a chave K cifrada - $E(Kc, K)$ - e outro ficheiro com a mensagem M cifrada com a chave K - $Ek(M)$.
> 4. O Bob decifra o ficheiro com a chave K utilizando a sua chave privada, isto é, $E(Kd, E(Kc, K)) = K$.
> 5. O Bob decifra o ficheiro com a mensagem M utilizando a chave K obtida no passo 4.
>
> Para a implementação do problema foi utilizado o algoritmo Chacha20 para a cifra simétrica Ek e o algoritmo RSA para a cifra assimétrica E.

A implementação do programa foi realizado em Python e utilizou a biblioteca *PyCryptodome* para a implementação da cifra simétrica e assimétrica. A implementação inicia-se com a apresentação de um menu ao utilizador, para que este consiga escolher se pretende cifrar ou decifrar o ficheiro.

```
# Opções do menu do utilizador
menu = {
    0: 'SAIR',
    1: 'Cifragem',
    2: 'Decifragem',}

# Menu para determinar se o utilizador pertende
# cifrar ou  decifrar uma mensagem
while True:
    for key in menu.keys():
        print(key, '--', menu[key] )
    option = int(input('=>  '))
    if option == 1:
        encrypt()
    elif option == 2:
        decrypt()
    elif option == 0:
        exit()
    else:
        print('Opção Inválida.')
```

Caso o utilizador pretenda cifrar um ficheiro é chamada a função *encrypt* que irá perguntar ao utilizador qual o ficheiro a cifrar, o nome do ficheiro cifrado com a mensagem e o nome do ficheiro cifrado com a chave de sessão K. Esta chave é gerada utilizando a função *get_random_bytes* que gera 32 bytes de forma pseudo-aleatória.

De seguida é perguntado ao utilizador se pretende ler a chave pública de um ficheiro ou se pretende gerar um novo par de chaves assimétricas. Caso este pretenda ler de um ficheiro, então é utilizado o método *import_key* da primitiva RSA que através de um ficheiro *PEM* devolve o valor da chave que lá se encontra armazenada. Caso o utilizador pretenda gerar novos parâmetros para as chaves, então o programa gera um par de chaves de 2048 bits e pede ao utilizador o nome dos ficheiros *PEM* para armazenar a chave pública e privada, utilizando o método *export_key* da primitiva RSA.

É de notar que o programa, sempre que gera um novo par de chaves assimétricas, terá de as armazenar obrigatoriamente em ficheiros, pois só assim é possível que, no processo de decifra, o programa consiga aceder à chave privada gerada. Esta chave privada será protegida por uma *password*, que irá ser indicada pelo utilizador. Por vezes a *password* indicada terá uma entropia baixa que terá de ser amplificada por funções KDF (*Key Derived Functions*), contudo o método *export_key*, que permite ao programa armazenar chaves públicas e privadas em ficheiros assim como proteger a chave privada utilizando uma *password*, já utiliza algoritmos KDF como o PBKDF2 e MD5 de forma a aumentar a entropia da *password*. O uso destes algoritmos depende do formato do ficheiro onde se armazena a chave. Caso o formato seja do tipo DER então o método utiliza o algoritmo PBKDF2, no caso do formato ser PEM então a função utiliza o algoritmo MD5. Desta forma não é necessário ao programa reutilizar o algoritmos KDF utilizado no problema 1 do TP anterior.

Finalmente, após a geração e armazenado das chaves ou leitura destas, utilizamos a função *buildCiphertext* para criar o conteúdo dos dois ficheiros cifrados, um com a mensagem outro com a chave K. Este conteúdo será armazenado nos ficheiros indicados pelo o utilizador.

```
#FUNÇÃO: Cifra ficheiro
def encrypt():

    #ficheiro a cifrar
    doc = input('Ficheiro a cifrar:  ')
    #nome do ficheiro cifrado
    fileName = input('Ficheiro com a mensagem cifrada:  ')
    #nome do ficheiro cifrado com a chave da sessão
    fileNameKey = input('Ficheiro com a chave cifrada:  ')

    #retira a mensagem do ficheiro a cifrar
    f = open(doc, "r")
    msg = f.read()
    f.close()

    print("MENSAGEM")
    print(msg)

    #cria a chave de sessão
```

```
K = get_random_bytes(32)

print("Deseja gerar novas chaves? [Y/N] ")
option = input('=>  ').upper()

if option == 'Y':

    #gera o par de chaves utilizando RSA
    rsaKey = RSA.generate(2048)

    publicFile = input('Ficheiro onde armazenar ' +
                           'a chave pública (***.pem):  ')
    privateFile = input('Ficheiro onde armazenar ' +
                           'a chave privada (***.pem):  ')

    #Password de proteção da chave privada
    password = getpass.getpass('Password de proteção da ' +
                                'chave privada:  ').encode('utf-8')

    #Exporta a chave pública e privada para um ficheiro
    privateKey = rsaKey.export_key(format='PEM',passphrase=password)
    publicKey = rsaKey.publickey().export_key(format='PEM')

    #Armazena a chave privada no ficheiro
    private_out = open(privateFile, "wb")
    private_out.write(privateKey)
    private_out.close()

    #Armazena a chave publica no ficheiro
    public_out = open(publicFile, "wb")
    public_out.write(publicKey)
    public_out.close()

else:

    #Le a chave publica do ficheiro fornecido
    keyEncrypted = input('Ficheiro da chave pública:  ')
    rsaKey = RSA.import_key(open(keyEncrypted).read())

#cria a mensagem e a chave cifrada
ciphertext, cipherKey = buildCiphertext(msg,K,rsaKey)

print("CIPHERTEXT - ** Armazenado no Ficheiro " + fileName + " **")
print(json.dumps(ciphertext))

#escreve a mensagem cifrada em bytes no ficheiro com
# nome igual à variavel filename
```

```
    output = open(fileName, "w")
    output.write(json.dumps(ciphertext))
    output.close()

    print("CIPHERKEY - ** Armazenado no Ficheiro " + fileNameKey + " **")
    print(cipherKey)

    #escreve a chave cifrada em bytes no ficheiro
    # com nome igual à variavel filenameKey
    outputKey = open(fileNameKey, "wb")
    outputKey.write(cipherKey)
    outputKey.close()
```

Na função *buildCiphertext*, vamos gerar 2 ciphertext, um com a mensagem e outro com a chave K cifrada. Vamos utilizar a chave K e um valor *nonce* de 12 bytes no algoritmo simétrico Chacha20, para formar a mensagem cifrada a ser armazenada no ficheiro que foi indicado pelo utilizador. Para cifrar a chave da sessão K, será utilizada a chave pública lida ou gerada na função *encrypt* no algoritmo PKCS1 OAEP baseado na primitiva RSA e com *padding* OAEP.

```
#FUNÇÃO: Cria o ciphertext através da cifra Chacha20
# e cifra a chave da sessão com o RSA
def buildCiphertext(msg,sessionKey,rsa):

    #cria o nonce de 12 bytes de forma pseudo-aletória
    nonce = get_random_bytes(12)

    #cria a cifra Chacha20 utilizando
    # a chave da sessão e o nonce
    cipher = ChaCha20.new(key=sessionKey, nonce=nonce)
    #cifra a mensagem
    ciphertext = cipher.encrypt(dumps(msg))

    #transforma em bytes a mensagem, nonce e
    # o salt para geração da chave
    ct = b64encode(ciphertext).decode('utf-8')
    nc = b64encode(nonce).decode('utf-8')

    #mensagem cifrada - ciphertext
    pkg = {'ciphertext' : ct, 'nonce': nc}

    #cifrar a chave da sessão utilizando a chave pública
    cipher_rsa = PKCS1_OAEP.new(rsa)
    k = cipher_rsa.encrypt(sessionKey)

    return pkg, k
```

Caso o utilizador pretenda decifrar um ficheiro é chamada a função *decrypt* que irá perguntar ao utilizador qual os ficheiros com a mensagem e chave K a decifrar e o

nome do ficheiro com a mensagem decifrada. É também perguntado ao utilizador qual o nome do ficheiro PEM que contém a chave privada e qual a password que protege a chave privada que será utilizada no processo de decifra.

É utilizada a função *import_key* para que através da password indicada na função *encrypt* e do ficheiro fornecido, o programa consiga ter acesso à chave privada a ser utiliza na função *buildPlaintext* e assim obter a mensagem decifrada.

```
#Função: Decifra um ficheiro
def decrypt():

    #ficheiro a decifrar com a mensagem
    doc = input('Ficheiro com a mensagem a decifrar:  ')
    #ficheiro a decifrar com a chave decifrada
    docKey = input('Ficheiro com a chave a decifrar:  ')
    #nome do ficheiro com a mensagem decifrada
    fileName = input('Ficheiro de output com a mensagem cifrada:  ')

    #retira bytes da mensagem do ficheiro a decifrar
    f = open(doc, "rb")
    msg = f.read()
    ciphertext = json.loads(msg)

    print("MENSAGEM")
    print(msg)

    #retira bytes da chave da sessão a decifrar
    f = open(docKey, "rb")
    msgKey = f.read()

    #Ficheiro que contém a chave privada
    privateFile = input('Ficheiro onde ler a ' +
                            'chave privada (***.pem):  ')
    #Password de proteção da chave privada
    password = getpass.getpass('Password de proteção da ' +
                               'chave privada:  ').encode('utf-8')
    #importa a chave privada presente no ficheiro fornecido
    rsaKey = RSA.import_key(open(privateFile).read(), passphrase=password)

    #cria a mensagem decifrada
    plaintext = buildPlaintext(ciphertext,msgKey,rsaKey)

    print("PLAINTEXT - ** Armazenado no Ficheiro " + fileName + " **")
    print(loads(plaintext))

    #escreve no ficheiro a mensagem cifrada
    output = open(fileName, "w")
    output.write(loads(plaintext))
```

Na função *buildPlaintext*, vamos gerar o *plaintext* representativo da mensagem decifrada. Vamos utilizar a chave privada obtida da função *decrypt* e o algoritmo PKCS1 OAEP para obter a chave K decifrada. Esta chave será utilizada juntamente com o valor *nonce* retirado do ficheiro cifrado fornecido pelo utilizador no algoritmo simétrico Chacha20 para gerar o *plaintext* da mensagem.

```
#FUNÇÃO: Cria o plaintext através da cifra ChaCha20
def buildPlaintext(msg,msgKey,privateKey):

    #Decifra a chabe da sessão utilizando a chave privada
    cipher_rsa = PKCS1_OAEP.new(privateKey)
    sessionKey = cipher_rsa.decrypt(msgKey)

    #retira o valor nonce da mensagem cifrada
    nonce = b64decode(msg['nonce'])
    #retira a mensagem cifrada do ciphertext
    ciphertext = b64decode(msg['ciphertext'])

    #cria a decifra Chacha20 utilizando a chave K e o non
    cipher = ChaCha20.new(key=sessionKey, nonce=nonce)
    #decifra mensagem
    plaintext = cipher.decrypt(ciphertext)

    return plaintext
```

Finalmente, a envelope digital desenvolvida em Python encontra-se implementada na íntegra no seguinte link do repositório do Github. Para conseguir correr o programa pode-se utilizar o ficheiro *.txt* com uma mensagem a ser cifrada e posterior decifrada presente no link.

Pergunta P1.2

O programa implementado permite executar 3 funcionalidades. A primeira gera duas chaves RSA com 2048 bits, uma pública e uma privada. Já a segunda permite assinar um ficheiro e a última verificar essa assinatura.

De facto, a primeira funcionalidade inicia-se pela instância de um objeto gerador de chaves RSA, sendo que, ambas as chaves são escritas em dois ficheiros separados cujo path é indicado pelo utilizador.

```
1 private static void keyGenHandler(String pubKeyFile, String
  ↪ privKeyFile){
2     try{
3         KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
4         keyGen.initialize(KEY_SIZE);
5         KeyPair keyPair = keyGen.generateKeyPair();
6         writePrivateKey(keyPair.getPrivate(), privKeyFile);
7         writePublicKey(keyPair.getPublic(), pubKeyFile);
8     } catch(NoSuchAlgorithmException e) {
```

```
9           System.out.println(e.getMessage());
10
11      }
12  }
```

De seguida a função que permite realiza a assinatura começa por ler o ficheiro onde se encontra a chave privada (indicado como argumento pelo utilizador) e cria uma instância de um objeto que permite criar assinaturas RSA, sendo o hash utilizado o SHA-256. Este objeto é inicializado através do método "initSign" utilizando a chave privada como argumento.

Posteriormente, o conteúdo do ficheiro que contém a mensagem é lido através de uma BufferedInputStream e utiliza-se o método update sobre o objeto que irá gerar a assinatura passando-lhe como argumento os bytes lidos da mensagem.

Por fim, escreve-se num ficheiro de output, sendo este também fornecido pelo utilizador, a assintura que é gerada pelo objeto.

```
1   private static void signatureHandler(String privKey, String input,
    ↪  String output){
2       byte [] buffer = new byte[2048]; //Buffer para leitura do ficheiro
3
4       try{
5           PrivateKey privateKey = readPrivateKey(privKey);
6           Signature sign = Signature.getInstance("SHA256withRSA");
7           sign.initSign(privateKey);
8
9           BufferedInputStream in = new BufferedInputStream(new
            ↪  FileInputStream(input));
10
11          BufferedOutputStream out = new BufferedOutputStream(new
            ↪  FileOutputStream(output));
12
13          int readBytes;
14
15          while((readBytes = in.read(buffer)) != -1)
16              sign.update(buffer, 0, readBytes);
17
18          out.write(sign.sign());
19
20          out.close();
21          in.close();
22
23      } catch(NoSuchAlgorithmException | SignatureException | IOException
        ↪  e) {
24          System.out.println(e.getMessage());
25
26      } catch(InvalidKeyException e){
27          System.out.println("A chave é inválida.");
28      }
```

```
29
30 }
```

O método que implementa a verificação da assinatura começa por carregar a chave pública guardada num ficheiro para um objeto do tipo "PublicKey". De seguida cria-se um objeto que permite gerar assinaturas RSA exatamente igual ao criado para realizar a assinatura só que desta vez inicializa-se o objeto com o método "initVerify" passando-lhe como argumento a chave pública que foi carregada.

De seguida, lê-se o conteúdo do ficheiro que contém a mensagem e utiliza-se novamente o método update com os bytes que vão sendo lidos. No final, carrega-se o ficheiro que contém assinatura e utiliza-se o método verify sobre o objeto "sign" que devolve um `boolean` caso a assinatura carregada através do ficheiro e aquela presente no objeto são as mesmas.

```
1  private static boolean verifyHandler(String pubKey, String input,
   ↪  String signatureFile){
2      byte [] buffer = new byte[2048]; //Buffer para leitura do ficheiro
       ↪  de input
3      try{
4          PublicKey publicKey = readPublicKey(pubKey);
5          Signature sign = Signature.getInstance("SHA256withRSA");
6          sign.initVerify(publicKey);
7
8          BufferedInputStream in = new BufferedInputStream(new
           ↪  FileInputStream(input));
9
10         int readBytes;
11
12         while((readBytes = in.read(buffer)) != -1)
13             sign.update(buffer, 0, readBytes);
14
15         in.close();
16
17         Path signaturePath = Paths.get(signatureFile);
18         byte [] signature = Files.readAllBytes(signaturePath);
19
20         return sign.verify(signature);
21     } catch(NoSuchAlgorithmException | SignatureException | IOException
       ↪  e) {
22         System.out.println(e.getMessage());
23
24     } catch(InvalidKeyException e){
25         System.out.println("A chave é inválida.");
26     }
27
28     return false;
29 }
```

De modo a carregar a chave privada através de um ficheiro utiliza-se o método "readPrivateKey" que utiliza o "PKCS8EncodedKeySpec" para conseguir ler corretamente o conteúdo do ficheiro que contém a chave e devolve um objeto do tipo RSAPrivateKey que é uma interface para um objeto PrivateKey e é compatível com os métodos utilizados anteriormente.

```
1  private static PrivateKey readPrivateKey(String fileName){
2      try{
3          Path path = Paths.get(fileName);
4          byte [] privKey = Files.readAllBytes(path);
5
6          KeyFactory keyFactory = KeyFactory.getInstance("RSA");
7          PKCS8EncodedKeySpec privKeySpec = new
           ↪  PKCS8EncodedKeySpec(privKey);
8          return (RSAPrivateKey) keyFactory.generatePrivate(privKeySpec);
9
10     } catch(NoSuchAlgorithmException | InvalidKeySpecException |
       ↪  IOException e) {
11         System.out.println(e.getMessage());
12     }
13     return null;
14 }
```

Um procedimento similar é utilizado no método que permite carregar a chave pública mas utiliza-se "X509EncodedKeySpec" em vez de "PKCS8EncodedKeySpec". O objeto devolvido é do tipo RSAPublicKey que tal como o caso anterior é uma interface para um objeto PublicKey.

```
1  private static PublicKey readPublicKey(String fileName){
2      try{
3          Path path = Paths.get(fileName);
4          byte [] pubKey = Files.readAllBytes(path);
5
6          KeyFactory keyFactory = KeyFactory.getInstance("RSA");
7          EncodedKeySpec publicKeySpec = new X509EncodedKeySpec(pubKey);
8          return (RSAPublicKey) keyFactory.generatePublic(publicKeySpec);
9
10     } catch(NoSuchAlgorithmException | InvalidKeySpecException |
       ↪  IOException e) {
11         System.out.println(e.getMessage());
12     }
13     return null;
14
15 }
```

Os dois métodos que se seguem permitem escrever as chaves em ficheiro e procedem de forma similar, sendo que, utilizam o método "getEncoded" do objeto gerado no passo de geração de chaves para poder escrever os bytes da chave em ficheiro.

```java
private static void writePrivateKey(PrivateKey key, String output){
    try{
        FileOutputStream out = new FileOutputStream(output);
        out.write(key.getEncoded());
    } catch(IOException e){}
}
```

```java
private static void writePublicKey(PublicKey key, String output){
    try{
        BufferedOutputStream out = new BufferedOutputStream(new
        ↪  FileOutputStream(output));
        out.write(key.getEncoded());
        out.close();
    } catch(IOException e){}
}
```

O código desenvolvido encontra-se no seguinte link e para o compilar basta realizar "javac RSA_PSS.java".

Para executar a funcionalidade de gerar as chaves basta correr o seguinte comando:

java RSA_PSS keyGen [PUBKEY OUTPUT FILE] [PRIVKEY OUTPUT FILE]

- PUBKEY OUTPUT FILE:ficheiro que irá armazenar a chave pública RSA.
- PRIVKEY OUTPUT FILE: ficheiro que irá armazenar a chave privada RSA.

Para executar a funcionalidade de assinar o ficheiro utiliza-se o seguinte comando:

java RSA_PSS sign [INPUT FILE] [OUTPUT FILE] [PRIVATE KEY]

- INPUT FILE: ficheiro que contém a mensagem.
- PUBLIC KEY: ficheiro que irá guardar a assinatura gerada.
- PUBLIC KEY: ficheiro que contém a chave privada.

Por fim, para verificar uma assinatura basta utilizar o seguinte comando:

java RSA_PSS verify [INPUT FILE] [PUBLIC KEY] [SIGNATURE FILE]

- INPUT FILE: ficheiro que contém a mensagem.
- PUBLIC KEY: ficheiro que contém a chave pública.
- PUBLIC KEY: ficheiro que contém a assinatura.