As primeiras APIs de segurança surgiram no JDK release 1.1, com a "***Java Cryptography Architecture***" (JCA), que consistia numa plataforma para desenvolver funcionalidades criptográficas em Java.
A ***Java Cryptography Extension*** (JCE) estende a JCA API para incluir *encryption*, *key exchange*, e *Message Authentication Code* (MAC ou MIC). O JCE surgiu como um *package* opcional, uma extensão ao JDK, mas desde o JDK1.4 que foi integrado na API.

Cryptographic Service Provider

Um ***Cryptographic Service Provider*** é formado por um ou vários pacotes que fornecem uma implementação concreta de um subconjunto dos detalhes criptográficos da Security API.

Usualmente, para adicionar um provider é necessária uma linha no ficheiro:

```
$JAVA_HOME/lib/security/java.security

security.provider.n=masterClassName
```

Algumas classes importantes:
(veja a descrição, construtores e principais métodos na **Java API** DOC)

-`java.security.Security`

Esta classe inclui as propriedades gerais de segurança e permite gerir os Cryptographic Service Providers instalados. É possível saber que algoritmos podem ser usados via API para cada tipo de serviço e cada provider.

-`javax.crypto.KeyGenerator`

Classe usada para gerar chaves secretas para uso em algoritmos de encriptação simétricos. A geração da chave depende do algoritmo a que se destina.

-`java.security.KeyPairGenerator`

Classe usada para gerar pares de chaves para uso em algoritmos assimétricos. É gerada uma chave privada e a sua correspondente chave pública.

-`java.security.KeyStore`

Chama-se **KeyStore** a uma BD usada para guardar chaves e certificados. Usado por aplicações que lidam com autenticação, por exemplo.
Formatos de KeyStore suportados pelo Java 1.5.0:

-PKCS12 (Public-Key Cryptography Standards)

-JKS (proprietário da Sun)

-JCEKS

O tipo de KeyStore a usar depende do tipo de dados que desejamos guardar.

`-java.security.MessageDigest`

Classe abstrata que descreve a funcionalidade disponibilizada por algoritmos de digest como o MD5 e SHA.

`-javax.crypto.Cipher`

Classe que fornece métodos para cifrar e decifrar dados. Os objetos desta classe são inicializados para um determinado tipo de transformação. Esta transformação inclui o nome do algoritmo a usar, que pode ser simétrico ou assimétrico.

`-java.security.Signature`

Classe abstrata que descreve a funcionalidade de um algoritmo de assinatura digital. As assinaturas digitais usam-se para autenticação e integridade de dados.

Notas:

Uma mensagem assinada não vai "escondida". Não confundir com confidencialidade!

Quem valida deve certificar-se de que a chave pública que vai utilizar corresponde realmente ao *signer*. É aí que entra a certificação digital.

`-javax.security.cert.X509Certificate`

Classe abstrata que representa um certificado digital do tipo X.509.

Um certificado digital é um documento assinado por uma entidade de confiança, que declara qual a chave pública de um indivíduo. A chave pública desta entidade de confiança, uma Certification Authority (CA), é obtida por um processo seguro. Os certificados são usados, por exemplo, para autenticação. Todos os browsers têm uma BD com os certificados das principais "Root CAs" internacionais.

O tipo Java String não é adequado para representar passwords... (Devido à eventual exposição do conteúdo em memória, após funcionamento do GC, e não só). Esta informação sensível deve ser representada com o tipo char array.

Ferramentas:

Ferramentas JDK de linha de comandos relacionadas com Segurança:

- `keytool` - manipulação de certificados e KeyStores
- `jarsigner` - assinar e validar assinaturas em ficheiros JAR

Ferramenta Linux:

- `openssl`

EXERCÍCIOS

1- Crie uma nova classe com o nome `Cifra`.
Através da classe Security, consulte o **nome** e **versão** dos *Cryptographic Service Providers* instalados no seu ambiente de desenvolvimento Java.

espera-se algo do género:

```
  = SUN version 15
```

```
        name: SUN
        version: 15.0
  = SunRsaSign version 15
        name: SunRsaSign
        version: 15.0
  = SunEC version 15
        name: SunEC
        version: 15.0
  = SunJSSE version 15
        name: SunJSSE
        version: 15.0
  = SunJCE version 15
        name: SunJCE
        version: 15.0
  = SunJGSS version 15
        name: SunJGSS
        version: 15.0
  = SunSASL version 15
        name: SunSASL
        version: 15.0
  = XMLDSig version 15
        name: XMLDSig
        version: 15.0
  = SunPCSC version 15
        name: SunPCSC
        version: 15.0
  = JdkLDAP version 15
        name: JdkLDAP
        version: 15.0
  = JdkSASL version 15
        name: JdkSASL
        version: 15.0
  = SunPKCS11 version 15
        name: SunPKCS11
        version: 15.0
```

2 - Para o primeiro daqueles Cryptographic Service Providers, imprima a listagem de serviços que o mesmo suporta.
exemplo:

```
servicos: [SUN: SecureRandom.NativePRNG -> sun.security.provider.NativePRNG
, SUN: SecureRandom.SHA1PRNG -> sun.security.provider.SecureRandom
  attributes: {ImplementedIn=Software}
, SUN: Signature.SHA1withDSA -> sun.security.provider.DSA$SHA1withDSA
  aliases: [DSA, DSS, SHA/DSA, SHA-1/DSA, SHA1/DSA, SHAwithDSA, DSAWithSHA1, OID.1.2.840.10040.4.3,
1.2.840.10040.4.3, 1.3.14.3.2.13, 1.3.14.3.2.27]
  attributes: {ImplementedIn=Software, KeySize=1024,
SupportedKeyClasses=java.security.interfaces.DSAPublicKey|java.security.interfaces.DSAPrivateKey}
, SUN: Signature.NONEwithDSA -> sun.security.provider.DSA$RawDSA
  aliases: [RawDSA]
  attributes:
{SupportedKeyClasses=java.security.interfaces.DSAPublicKey|java.security.interfaces.DSAPrivateKey}
, SUN: KeyPairGenerator.DSA -> sun.security.provider.DSAKeyPairGenerator
  aliases: [OID.1.2.840.10040.4.1, 1.2.840.10040.4.1, 1.3.14.3.2.12]
  attributes: {ImplementedIn=Software, KeySize=1024}
, SUN: MessageDigest.MD2 -> sun.security.provider.MD2
, SUN: MessageDigest.MD5 -> sun.security.provider.MD5
  attributes: {ImplementedIn=Software}
, SUN: MessageDigest.SHA -> sun.security.provider.SHA
  aliases: [SHA-1, SHA1]
  attributes: {ImplementedIn=Software}
, SUN: MessageDigest.SHA-256 -> sun.security.provider.SHA2
, SUN: MessageDigest.SHA-384 -> sun.security.provider.SHA5$SHA384
, SUN: MessageDigest.SHA-512 -> sun.security.provider.SHA5$SHA512
, SUN: AlgorithmParameterGenerator.DSA -> sun.security.provider.DSAParameterGenerator
  attributes: {ImplementedIn=Software, KeySize=1024}
, SUN: AlgorithmParameters.DSA -> sun.security.provider.DSAParameters
  aliases: [1.3.14.3.2.12, 1.2.840.10040.4.1]
  attributes: {ImplementedIn=Software}
, SUN: KeyFactory.DSA -> sun.security.provider.DSAKeyFactory
  aliases: [1.3.14.3.2.12, 1.2.840.10040.4.1]
  attributes: {ImplementedIn=Software}
, SUN: CertificateFactory.X.509 -> sun.security.provider.X509Factory
  aliases: [X509]
  attributes: {ImplementedIn=Software}
, SUN: KeyStore.JKS -> sun.security.provider.JavaKeyStore$JKS
  attributes: {ImplementedIn=Software}
, SUN: KeyStore.CaseExactJKS -> sun.security.provider.JavaKeyStore$CaseExactJKS
, SUN: Policy.JavaPolicy -> sun.security.provider.PolicySpiFile
, SUN: Configuration.JavaLoginConfig -> sun.security.provider.ConfigSpiFile
, SUN: CertPathBuilder.PKIX -> sun.security.provider.certpath.SunCertPathBuilder
  attributes: {ValidationAlgorithm=RFC3280, ImplementedIn=Software}
, SUN: CertPathValidator.PKIX -> sun.security.provider.certpath.PKIXCertPathValidator
  attributes: {ValidationAlgorithm=RFC3280, ImplementedIn=Software}
, SUN: CertStore.LDAP -> sun.security.provider.certpath.LDAPCertStore
  attributes: {ImplementedIn=Software, LDAPSchema=RFC2587}
, SUN: CertStore.Collection -> sun.security.provider.certpath.CollectionCertStore
  attributes: {ImplementedIn=Software}
, SUN: CertStore.com.sun.security.IndexedCollection ->
sun.security.provider.certpath.IndexedCollectionCertStore
  attributes: {ImplementedIn=Software}
]
```

3- Verifique quais são os algoritmos disponíveis para encriptar (opção `Cipher`) - para o CSProvider por omissão.

espera-se algo do género:

```
= BLOWFISH
= ARCFOUR
= PBEWITHMD5ANDDES
= RC2
= RSA
= PBEWITHMD5ANDTRIPLEDES
= PBEWITHSHA1ANDDESEDE
= DESEDE
= AESWRAP
= AES
= DES
= DESEDEWRAP
= PBEWITHSHA1ANDRC2_40
```

4- Experiência com algoritmo simétrico:
- Gere uma nova chave secreta (para usar com o algoritmo AES)
- Imprima em stdout algumas propriedades dessa chave (**algoritmo** ao qual se destina)

NOTA: apesar de poder escrever esta chave diretamente num ficheiro (via `ObjectOutputStream`), esse não é um procedimento adequado, porque a chave ficaria exposta. Deve usar um repositório protegido: `KeyStore`.

5 - Experiência com a chave secreta: encriptar uma String

```
Cipher cipher;

cipher = Cipher.getInstance(ALGORITMO);
cipher.init(Cipher.ENCRYPT_MODE, mySecretKey);
/* para vários blocos usar-se-ia o método update() (multiple-part encryption)
   até ao penúltimo bloco, seguido de doFinal()
*/

byte[] plaintext = MSG.getBytes();
byte[] ciphertext = cipher.doFinal( plaintext );  // cifrar num "bloco" só

// teste rápido para ver o q fica, que não será visível:
System.out.println( new String(ciphertext) );
```

6- Obtenha e compile o ficheiro ProcessaFicheiros.java...
Teste a aplicação fornecendo o nome de um ficheiro existente e um novo nome.
A aplicação copia o conteúdo do primeiro para o segundo.
Exemplo:

```
$ java ProcessaFicheiros a.txt a.asc
```

```
vou passar dados de (a.txt) para (a.asc)
```

```
-----------
```

```
# confirmar que são iguais:
$ diff a.txt  a.asc
# listar o conteúdo de ambos:
$ cat a.txt
$ cat a.asc
```

7 - Aumente a funcionalidade da classe, acrescentando dois métodos para:
- Receber no primeiro argumento a operação a fazer (`-copiar`, `-cifrar`, `-decifrar`)
- Com a operação "-cifrar", encriptar os dados antes de os escrever no segundo ficheiro
- Com a operação "-decifrar", desencriptar os dados antes de os escrever no segundo ficheiro

Com uma verificação semelhante a:

```
      if (args.length<3) {
           System.err.println("erro: argumentos insuficientes\n"+
                           "java ProcessaFicheiros OPCAO inFilename outFilename");
           System.exit(1);
       }
       ...
       // esta parte ate podia dar
       ProcessaFicheiros pf= new ProcessaFicheiros(args[1], args[2]);
       if (args[0].equals("-copiar"))
           pf.passaDados();
       else if (args[0].equals("-cifrar"))
           pf.cifrar();
       else if (args[0].equals("-decifrar"))
           pf.decifrar();
       else
           System.err.println("OPCAO INVALIDA: " + args[0]);
```

Esses métodos **devem ter como argumento** uma chave secreta e outros elementos que considerar necessários para uso de um algoritmo simétrico (o nome do algoritmo).
Apesar de ser possível e conveniente usar `CipherOutputStream` e `CipherInputStream`, resolva este exercício apenas com os métodos `init`(), `update`() e doFinal() da classe Cipher.

Modo:

```
$ java ProcessaFicheiros -cifrar dados.txt dados.asc
$ java ProcessaFicheiros -decifrar dados.asc dados2.txt
```

... e depois compare dados.txt com dados2.txt
Espera-se que sejam diferentes, pelo simples facto de ter usado chaves secretas diferentes no momento de cifrar e decifrar... adiante isso será resolvido.

8- Voltando à classe **Cifra**, faça um teste para criar um novo **keyStore** (tipo JCEKS)
- guardar uma nova chave secreta protegida com uma password
- confirmar que o tamanho do keystore passou de 0 para 1
- gravar o KS num ficheiro (*ver exemplo na API*)
- na consola, listar o ficheiro (cat)

9- Atualize a sua classe **ProcessaFicheiros** para ler uma **SecretKey** de um **KeyStore** já existente (passando uma password), que deve usar para o processo criptográfico.
- Mantenha os métodos criados acima.
- Acrescente métodos para ler a chave do KeyStore, que depois chamam os anteriores...

-Teste esta nova versão (encriptar e desencriptar um ficheiro) e compare o resultado de decifrar com o ficheiro inicial...

APOIO:

manipulação de chaves e keystore
cifrar e decifrar com algoritmo simétrico

Contactar suporte do site

Nome de utilizador: Rodrigo Alves (Sair)
Resumo da retenção de dados
Obter a Aplicação móvel

Fornecido por Moodle