# Implementação de Protocolos Criptográficos

Manuel Barbosa @ DI Universidade do Minho

Setembro de 2001

# Introdução

- A operação mais básica num protocolo de comunicação criptográfico é aquele em que apenas se pretende transmitir um item de informação x de uma entidade A para uma entidade B.
- Pressupõe-se que A e B são processos independentes que podem, ou não, ser executados em máquinas distintas.
- Assim, o primeiro problema com que nos encontramos consiste em saber como estabelecer um canal de comunicação entre duas entidades.
- Visto estarmos usar como linguagem de implementação o JAVA, vamos considerar que x pode ser um objecto qualquer.
- Uma das formas mais frequentes de abordar este problema consiste na utilização de sockets. Os sockets permitem que a comunicação entre processos seja independente da sua localização física e das características particulares do meio de comunicação utilizado.

# Introdução (cont)

- Para a implementação desta operação é também necessário garantir que ambos os processos estão sincronizados, i.e. B deve estar disponível para receber x quando A o transmitir.
- Sincronização a alto-nível - Parte deste problema resolve-se inserindo a operação num protocolo ao qual, por definição, todas as entidades devem obedecer. Assim, B sabe que chegado a um destes passos A lhe transmitirá x, e vice-versa.
- Sincronização a baixo-nível - A outra parte diz respeito à necessidade de alocar as estruturas de dados para receber a informação e desencadear fisicamente o processo. Para resolver este problema as linguagens disponibilizam, normalmente, primitivas de alto nível que permitem ao processo A desencadear a operação e ao processo B ficar à espera até que A lhe envie alguma coisa, sendo depois estabelecido um canal de comunicação entre os dois processos. No JAVA, estes requisitos são também satisfeitos pelos sockets.

# Utilização de sockets

- Na linguagem JAVA a entidade com um papel passivo (neste caso B) deve criar um ServerSocket associado a uma porta lógica da sua máquina e ficar à espera até que um canal de comunicação seja estabelecido.

```
try {
   ServerSocket ss = new ServerSocket(2000);
   Socket s = ss.accept();
   // comunicar
   s.close();
} catch (Exception e) {
   System.out.println("Excepcao: "+e.getMessage());
}
```

# Utilização de sockets (cont)

- No caso do processo A a implementação é mais simples:

```
try {
Socket s = new Socket(“nome.da.maquina.de.B", 2000);
//comunicar
s.close();
} catch (Exception e) {
System.out.println("Excepcao: "+e.getMessage());
}
```

- Nos slides seguintes assume-se que já se encontra estabelecido o canal de comunicação, através de um objecto do tipo Socket armazenado numa variável s.

# Utilização de socket (cont)

- Depois de estabelecido o canal de comunicação, o envio de x é muito simples: cada entidade obtém, a partir do socket, uma stream para transferencia de objectos (de input no caso de B e de output no caso de A).
- O código necessário para enviar/receber x seria, no caso do processo A:

```
OutputStream os = s.getOutputStream();
ObjectOutputStream oos = new ObjectOutputStream(os);
oos.writeObject(x);
```

- No caso do processo B teriamos:

```
InputStream is = s.getInputStream();
ObjectInputStream ois = new ObjectInputStream(is);
TipoDeX nomeB = (TipoDeX) ois.readObject();
```

# Operações básicas

- Todos os protocolos criptográficos são construídos a partir de um conjunto de operações básicas, para as quais se pode utilizar a seguinte notação:
  - { x }k - transmissão de um item x cifrado, sendo necessária a chave k para o decifrar.
  - (x) k - assinatura de x com a chave privada k.
  - < x ; y > - transmissão do par composto por x e y.
  - $\overline{x}$ - transmissão do código de hash de x.
- De seguida iremos descrever como implementar cada uma destas operações básicas.

# Operações básicas (cont)

- Assinaturas e códigos de hash
  - O método apresentado acima para a transmissão de um objecto qualquer x pode ser utilizado sem alteração para enviar assinaturas e códigos de hash, pois estes são simplesmente arrays de bytes.
- Tuplos
  - A transmissão de um par de informação < x ; y > pode ser decomposta na transmissão de x seguida da transmissão de y, que também podem ser transmitidos com o método já apresentado.

# Operações básicas (cont)

- Objectos cifrados
  - Para a transmissão de { x } k também se poderia utilizar o método apresentado, se optássemos por transmitir o array de bytes que resulta da cifragem.
  - No entanto, isso obriga a transformar x num array de bytes antes de poder ser submetido aos mecanismos de cifra apresentados no capítulo anterior.
  - Uma forma mais linear de o fazer consiste em utilizar os SealedObjects disponibilizados pela JCA. Como já foi referido, um SealedObject não é mais do que um "cofre" onde é guardado um objecto qualquer "fechado" com uma determinada chave.

# Operações básicas (cont)

- Objectos cifrados (cont)
  - Basicamente, estes objectos recebem como parâmetro, aquando da sua instanciação, um outro objecto qualquer e um mecanismo de cifra inicializado no modo de cifrar, e encapsulam o resultado de cifrar o objecto numa variável de instância.
  - Considerando que k é a chave usada para cifrar x, temos o seguinte código para enviar { x } k :

```
OutputStream os = s.getOutputStream();
ObjectOutputStream oos = new ObjectOutputStream(os);
Cipher c = Cipher.getInstance("Nome do mecanismo de cifra");
c.init(Cipher.ENCRYPT_MODE, k');
SealedObject so = new SealedObject(x,c);
oos.writeObject(so);
```

# Operações básicas (cont)

- Objectos cifrados (cont)
  - Para recuperar o objecto inicial é necessário passar-lhe como parâmetro o mesmo mecanismo de cifra inicializado no modo de decifrar com a chave respectiva.
  - Quem recebe { x } k deve efectuar as seguintes operações:

```
InputStream is = s.getInputStream();
ObjectInputStream ois = new ObjectInputStream(is);
Cipher c = Cipher.getInstance("Nome do mecanismo de cifra");
c.init(Cipher.DECRYPT_MODE, k);
SealedObject so = (SealedObject) ois.readObject();
TipoDeX x = (TipoDeX) so.getObject(c);
```

# Operações básicas (cont)

- Tuplos cifrados
  - No caso da transmissão de tuplos cifrados é possível simplificar o código, transmitindo em vez de { < x; y > } k , a informação { x } k seguida de { y } k .
  - Se o tamanho do bloco for inferior aos tamanhos de x e y e se utilizarmos o modo de cifra ECB então esta transformação não deverá comprometer a segurança do protocolo.
  - Com esta optimização evita-se a declaração de uma nova classe de objectos para agregar os pares.
  - Caso a operação a realizar sobre o tuplo seja o cálculo de uma cifra com o modo CBC, uma assinatura ou um hash, esta técnica não poderá ser aplicada pois o resultado dessas operações dependem de toda a mensagem.

# Exemplos

- Como exemplos da utilização das ferramentas discutidas nas aulas anteriores na a implementação de protocolos criptográficos, vamos analisar três casos:
  - O protocolo Diffie–Hellman de acordo de chaves.
  - O protocolo Station–to–Station de acordo de chaves.
  - O procolo Schnorr de identificação.

# Diffie-Hellman

- O protocolo de acordo de chaves Diffie-Hellman assenta no estabelecimento de dois parâmetros da comunidade: um número primo p e um parâmetro ? que seja um elemento primitivo de $?_p$.
- Cada uma das partes U e V gera um número, $a_U$ e $a_V$ respectivamente, e trocam entre si os valores módulo p de $? ^{a_U}$ e $? ^{a_V}$.
- A chave K é calculada independentemente por cada uma das partes como: $? ^{a_U a_V} \bmod p$ .

# Station-to-Station

- O protocolo Diffie-Hellman está sujeito a ataques do tipo man-in-the-middle.
- O protocolo Station-to-Station é uma versão melhorada do protocolo anterior, por forma a incluir uma identificação mutua das duas partes baseada em certificados e assinaturas digitais.
- Com efeito, dois passos adicionais são incluidos no protocolo Station-to-Station:
  - Troca de certificados de chave pública entre as duas entidades e verificação da sua validade mediante uma Certification Authority.
  - Troca de informação assinada digitalmente (as sequências $??^{a_U},?^{a_V}?$ e $??^{a_V},?^{a_U}?$) e verificação das assinaturas com base nos certificados trocados anteriormente.
- Apenas depois de completados estes passos com sucesso as entidades confiam no segredo acordado.

# Schnorr

- Este esquema de identificação baseia-se na existência de uma Trusted Authority (TA) que escolhe os parâmetros comuns da comunidade:
  - um número primo p “grande”;
  - um número primo q, divisor de p-1, também ele “grande”;
  - um parâmetro ? ? Zp, de ordem q.
- A entidade U que se pretende identificar solicita um certificado da TA para a sua chave pública, calculada como sendo $v ? ?^{? a}$ mod $p$ , em que $a$ é um número aleatório escolhido por U.

# Schnorr (cont)

- O processo de identificação decorre da seguinte forma:
  - U gera um número aleatório 0 ? k ? q-1, que utiliza para calcular um valor $? ? ? ^{?k} \bmod p$
  - U envia a V (a entidade que procede à identificação) o seu certificado e o valor anterior.
  - V verifica a validade do certificado e envia a U um valor r gerado aleatóriamente.
  - U calcula y = k + $a$ r mod q, e devolve y à entidade V.
  - Para verificar a identidade de U, V verifica que:

$$y ? ? ^{y} v^{r} \bmod p$$

# KeyAgreement

- Esta classe da JCE fornece a funcionalidade de um protocolo de acordo de chaves (e.g. Diffie-Hellman) em que se estabelece um segredo partilhado entre duas partes.
- As chaves envolvidas podem ter a sua origem em qualquer das classes de geração ou transformação de chaves da JCA.
- Ambas as partes envolvidas têm de obter uma instância desta engine class utilizando o método **getInstance**.
- Para a inicialização do objecto utiliza-se o método **init**, ao qual é necessáro fornecer, no mínimo, a chave que vai ser utilizada no estabelecimento do acordo.
- Adicionalmente é possível parametrizar o algoritmo de forma específica, bem como fornecer um gerador de números pseudo-aleatórios.

# KeyAgreement (cont)

- Os protocolos deste tipo consistem num conjunto de passos/fases que têm de ser completados por ambas as partes.
- A utilização desta classe consiste em chamar sucessivas vezes o método **doPhase**, que representa este processo.
- Esta método aceita como parâmetro a chave a ser processada no passo/fase correspondente.
- Esta chave poderá ser a chave pública de uma das partes, ou uma chave intermédia obtida numa fase anterior como resultado de uma invocação de **doPhase**.
- Note-se que as partes têm de trocar entre si as suas chaves públicas e, possivelmente, outras chaves intermédias, para se poder prosseguir com o protocolo.

# KeyAgreement (cont)

- O método **doPhase** aceita também um parâmetro que indica se o protocolo já está completo, ou se são necessários passos adicionais na sua execução.
- Quando todos os passos necessários ao “agreement” são completados por ambas as partes, o método **generateSecret** permite obter a chave secreta estabelecida:

```
public byte[] generateSecret();
public int generateSecret(byte[] sharedSecret, int offset);
public SecretKey generateSecret(String algorithm);
```