Aulas TP de SSI
Criptografia Aplicada

SDC — MEI

Ano Lectivo de 2014/15

1 Aulas 1 e 2

Estes projectos deverão ser realizados por grupos de 2. O código deverá ser enviado por e-mail ao responsável pela disciplina à medida que as tarefas forem concluídas. A submissão regular de soluções com qualidade será um dos parâmetros da avaliação TP. Quaisquer dúvidas podem ser esclarecidas por e-mail.

1.1 Ambiente de Desenvolvimento

O objectivo principal desta alínea é o de escolher/instalar o ambiente de desenvolvimento Java que será utilizado durante o curso. Pretende-se também recordar os princípios básicos da programação em Java, bem como a implementação de aplicações distribuídas utilizando sockets.

Como actividade de programação (para experimentar o ambiente escolhido), deve desenvolver uma pequena aplicação, composta por dois comandos executáveis na consola.

- Esses comandos deverão ser implementados na forma de duas classes Java: a classe Cliente e a classe Servidor.
- A aplicação deverá permitir a um número arbitrário de invocações da aplicação Cliente comunicar com um Servidor que escuta num dado `port` (e.g. 4567).
- O servidor atribuirá um número de ordem a cada cliente, e simplesmente fará o `dump` do texto enviado por cada cliente (prefixando cada linha com o respectivo número de ordem).
- Quando um cliente fecha a ligação, o servidor assinala o facto (e.g. imprimindo `=[n]=`, onde `n` é o número do cliente).
- A aplicação Cliente deverá permitir ao utilizador inserir do teclado as mensagens a enviar para o Servidor.

1.2 Cifra de ficheiro utilizando JCA/JCE

Pretende-se cifrar o conteúdo de um ficheiro. Para tal far-se-á uso da funcionalidade oferecida pela JCA/JCE, em particular a implementação de cifras simétricas.
O objectivo é então o de definir um pequeno programa Java que permita cifrar/decifrar um ficheiro utilizando a cifra simétrica RC4. A sua forma de utilização pode ser análoga a:

```
prog -genkey <keyfile>
prog -enc <keyfile> <infile> <outfile>
prog -dec <keyfile> <infile> <outfile>
```

Sugestões:

- Para simplificar, pode começar por utilizar uma chave secreta fixa, definida no código na forma de um array de bytes. Nesse caso, deverá utilizar a classe `SecretKeySpec` para a converter para o formato adequado.
- É também interessante verificar que o criptograma gerado é compatível com outras plataformas que implementam a mesma cifra. O comando seguinte utiliza o `openssl` para decifrar um ficheiro cifrado com `RC4` (a chave tem de ser fornecida em hexadecimal).

  ```
  openssl enc -d -rc4 -in <infile> -out <outfile> -K <chave>
  ```

Algumas classes relevantes:

- javax.crypto.Cipher
- javax.crypto.KeyGenerator
- javax.crypto.SecretKey (interface)
- javax.crypto.spec.SecretKeySpec
- javax.crypto.CipherInputStream
- javax.crypto.CipherOutputStream

1.3 Implementação da cifra RC4

Esta alínea representa um projecto extra, para aqueles alunos com interesse em perceber melhor a forma de funcionamento das cifras sequenciais, bem como os desafios colocados pela implementação de software criptográfico.

Pretende-se implementar de raiz a cifra RC4 e comprovar que a implementação realizada é compatível com as implementações comerciais da cifra.

Para isso, a classe desenvolvida na questão anterior deverá ser adaptada, por forma a que as chamadas à API do Java para efectuar a cifragem/decifragem sejam substituídas por chamadas a funções desenvolvidas especificamente para esse efeito.

Os detalhes funcionamento da cifra RC4 pode encontrar-se nos slides da disciplina e também em múltiplas referências on-line.

A verificação da correcção da implementação poderá ser feita testando a sua compatibilidade com a classe desenvolvida na alínea anterior, ou directamente com o `openssl`.

2 Aula 2

2.1 Confidencialidade na comunicação Cliente-Servidor

Pretende-se nesta tarefa modificar as classes Cliente e Servidor desenvolvidas nas aulas anteriores por forma a garantir a confidencialidade nas comunicações estabelecidas. Pretende-se ainda experimentar o impacto da escolha da cifra/modo na comunicação entre o cliente/servidor. Para tal é conveniente reforçar a natureza interactiva da comunicação modificando os ciclos de leitura/escrita para operarem sobre um byte de cada vez:

Cliente

```
CipherOutputStream cos = ...
int test;
while((test=System.in.read())!=-1) {
     cos.write((byte)test);
     cos.flush();
}
```

Servidor

```
CipherInputStream cis = ...
int test;
while ((test=cis.read()) != -1) {
      System.out.print((char) test);
}
```

Experimente agora as seguintes cifras (e modos) e verifique qual o respectivo impacto nas questões de buffering e sincronização:

- RC4
- AES/CBC/NoPadding
- AES/CBC/PKCS5Padding
- AES/CFB8/PKCS5Padding
- AES/CFB8/NoPadding
- AES/CFB/NoPadding

Procure explicar a diferenças detectadas na execução da aplicação.
Note que em muitos dos modos sugeridos necessita de considerar um IV. Considere para o efeito que o IV é gerado pelo cliente e enviado **em claro** para o servidor (no início da comunicação).
Algumas classes relevantes (para além das já estudadas...):

- javax.crypto.spec.IvParameterSpec
- java.security.SecureRandom

3 Aula 3

3.1 Protocolo Diffie-Hellman

O objectivo desta tarefa é implementar o acordo de chaves Diffie-Hellman. Algumas sugestões para atacar o problema:

- Comece por utilizar a classe `BigInteger` e codifique cada passo do protocolo explicitamente. Pode começar por utilizar os seguintes parâmetros para o grupo (P tem 1024 bit):

```
P = 99494096650139337106186933977618513974146274831566768179581759037259
78879815149981465395149272436547131625365146334225578531174860292245879 5
20138244532349993162545127260017318013612324544120413351580049591724201 1
86355872172330366152337257247721162014403880967369251202556667374699359 3
38460066704737369220358 3
G = 44157404837960328768872680677686802650999163226766694797650810379076
41646314726540108449111366762405455733539476160487688244692492984068199 0
10697431493501550157133302477317244035247535875066821344460735387275465 0
80503191286669211981937704190164273245591150986772821839454274533001407 1
04032685684699011971967 5
```

- Na JCA, podemos gerar valores apropriados para os parâmetros necessários através de uma instância apropriada da classe `AlgorithmParameterGenerator`.
- Em vez de trabalharmos directamente com a classe `BigInteger`, pode-se fazer uso da classe `KeyAgreement`.
- No JCE Reference Guide está disponível um exemplo de codificação do protocolo.
- Finalmente, complete a sua implementação utilizando a classe `AlgorithmParameterGenerator` para gerar os parâmetros `P` e `G` do algoritmo; e a classe `KeyPairGenerator` para gerar os pares de chaves ((x, g^x) e (y, g^y) para cada um dos intervenientes);

Novas classes:

- java.math.BigInteger
- java.security.AlgorithParameterGenerator
- javax.crypto.spec.DHParameterSpec
- javax.crypto.KeyAgreement
- java.security.KeyPairGenerator
- java.security.KeyPair
- java.security.spec.X509EncodedKeySpec

3.2 Autenticação do canal

Pretende-se complementar o programa com a autenticação das mensagens cifradas trocadas entre Cliente e Servidor.

Neste sentido, as aplicações devem derivar duas chaves simétricas a partir da chave de sessão k acordada. Para isso devem calcular $k_1 = H(k,'1')$ e $k_2 = H(k,'2')$, sendo H uma função de hash criptográfica. Chave k_1 será utilizada para parametrizar a cifra simétrica.

Todos os criptogramas enviados deverão ser acompanhados de um MAC (sugere-se a utilização do algoritmo HMAC) parametrizado com a chave k_2.

Novas Classes:

- java.security.MessageDigest
- javax.crypto.Mac

4 Aula 4

4.1 Codificação do protocolo Station-to-Station

Pretende-se complementar o programa com o acordo de chaves Diffie-Hellman para incluir a funcionalidade do protocolo Station to Station. Recorde que nesse protocolo é adicionado uma troca de assinaturas (cifrada com a chave de sessão negociada K), i.e.:

$$Alice \rightarrow Bob \quad : \quad g^x \tag{1}$$
$$Alice \leftarrow Bob \quad : \quad g^y, E_K(Sig_B(g^y, g^x)) \tag{2}$$
$$Alice \rightarrow Bob \quad : \quad E_K(Sig_A(g^x, g^y)) \tag{3}$$

Um requisito adicional neste protocolo é a manipulação de pares de chaves de cifras assimétricas (e.g. RSA). Para tal deve produzir um pequeno programa que gere os pares de chaves para cada um dos intervenientes e os guarde em ficheiros que serão lidos pela aplicação Cliente/Servidor.
Novas Classes:

- java.security.Signature
- java.security.KeyFactory
- java.security.spec.RSAPrivateKeySpec
- java.security.spec.RSAPublicKeySpec

4.2 Utilização de certificados X509 em Java

Pretende-se certificar as chaves públicas utilizadas no protocolo Station-to-Station com base em certificados X509. Para tal, disponibiliza-se:

- Certificado de chave pública do Servidor: server.cer
- Chave privada do Servidor (codificada em PKCS8): server.pk8
- Certificado de chave pública do Cliente: client.cer
- Chave privada do Cliente (codificada em PKCS8): client.pk8
- Certificado auto-assinado da autoridade de certificação: ca.cer

A utilização de certificados pressupõe a sua validação. O Java disponibiliza uma API específica que deverá utilizar para o efeito (documentação abaixo). Para facilitar esse estudo recomenda-se o estudo/adaptação de um programa de exemplo que verifica a validade de uma cadeia de certificação. Utilizando esse programa, podemos verificar a validade do certificado do servidor através da linha de comando:

```
java ValidateCertPath ca.cer servidor.cer
```

Uma segunda questão que surgirá neste trabalho é a manipulação dos formatos das chaves: as chaves privadas correspondentes aos certificados fornecidos estão codificadas num formato standard PKCS8. Para se converter esse formato num objecto Java apropriado terá de se utilizar uma instância da classe `KeyFactory`. O fragmento de código que se apresenta ilustra esse processo:

```
byte[] encodedKey;      // read from file
PKCS8EncodedKeySpec keySpec = new PKCS8EncodedKeySpec(encodedKey);
KeyFactory keyFactory = KeyFactory.getInstance("RSA");
RSAPrivateKey privKey =
(RSAPrivateKey)keyFactory.generatePrivate(keySpec);
```

Classes requeridas:

- java.security.cert.Certificate
- java.security.cert.X509Certificate
- java.security.cert.CertificateFactory
- java.security.cert.TrustAnchor
- java.security.cert.PKIXParameters
- java.security.cert.CertPath
- java.security.cert.CertPathValidator
- java.security.KeyFactory
- java.security.spec.PKCS8EncodedKeySpec

Outra documentação relevante:

- Java Certification Path API Programmer's Guide