

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/installar 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	Servidor
<pre>CipherOutputStream cos = ... int test; while((test=System.in.read())!=-1) { cos.write((byte)test); cos.flush(); }</pre>	<pre>CipherInputStream cis = ... int test; while ((test=cis.read()) != -1) { System.out.print((char) test); }</pre>

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
788798151499814653951492724365471316253651463342255785311748602922458795
201382445323499931625451272600173180136123245441204133515800495917242011
863558721723303661523372572477211620144038809673692512025566673746993593
384600667047373692203583
G = 44157404837960328768872680677686802650999163226766694797650810379076
416463147265401084491113667624054557335394761604876882446924929840681990
106974314935015501571333024773172440352475358750668213444607353872754650
805031912866692119819377041901642732455911509867728218394542745330014071
040326856846990119719675
```

- 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.AlgorithmParameterGenerator`
- `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.:

$$Alice \rightarrow Bob : g^x \quad (1)$$

$$Alice \leftarrow Bob : g^y, E_K(\text{Sig}_B(g^y, g^x)) \quad (2)$$

$$Alice \rightarrow Bob : E_K(\text{Sig}_A(g^x, g^y)) \quad (3)$$

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

5 Aula 7

5.1 Geração de Certificados X509

Nesta sessão iremos fazer uso do **openssl** para construir uma pequena PKI. Cada grupo irá ter uma pequena *autoridade de certificação*, que estará integrada na hierarquia da autoridade da UC de SSI, gerida pelo docente. O objectivo final é todos os alunos estarem dotados de certificados de chave pública que possam ser utilizados para comunicação autenticada e cifrada através de clientes de e-mail *standard*.

O processo de emissão de certificados X509 passa pelos seguintes passos:

1. O utilizador gera um par de chaves;
2. O utilizador produz um "pedido de certificado" que contém a sua identificação e a sua chave pública, que é assinado com a sua chave privada (o que certifica que quem solicita o certificado está na posse da respectiva chave privada);
3. A autoridade de certificação valida os dados contidos no certificado e, no caso de certificar positivamente esses dados, emite o certificado de chave pública correspondente.
4. O certificado é enviado ao utilizador e, eventualmente, publicado por outros meios (e.g. serviço de directoria).

Para cada um destes passos iremos fazer uso dos comandos respectivos do **openssl**.

Geração de chaves:

```
openssl genrsa -out grupoXPT0.key
```

Para utilizar a chave privada no Java é conveniente converter o seu formato para PKCS8

```
openssl pkcs8 -topk8 -nocrypt -in grupoXPT0.key \  
-outform der -out grupoXptoPrivKey.der
```

Geração do pedido de certificado:

Cada grupo deverá pedir um certificado para a sua autoridade de certificação ao docente da disciplina, utilizando o seguinte comando:

```
openssl req -new -key grupoXPT0.key -out grupoXPT0.csr
```

Emissão do certificado:

A emissão de certificados é normalmente realizada com o auxílio de *scripts* que invocam o comando *openssl* com os argumentos apropriados. Uma *script* normalmente utilizada para este efeito chama-se *CA.pl* e é distribuída com o *openssl*.

Produção de PKCS12:

Para certas utilizações (e.g. *browsers*, leitores de *email*, etc.) é conveniente encapsularmos o certificado e respectiva chave privada num ficheiro PKCS12.

```
openssl pkcs12 -export -chain -CAfile cacert.pem \  
-name GrupoXPT0 -aes128 -inkey grupoXPT0.key -in grupoXPT0.crt -out GrupoXPT0.p12
```

Verificação dos certificados:

Para validar um certificado, pode usar-se o seguinte comando

```
openssl verify -CAfile cacert.pem cert1.crt cert2.crt ...
```

6 Aula 8: Protocolo TLS/SSL

Configure um servidor HTTP para que funcione de acordo com os seguintes parâmetros:

1. O acesso à página de raiz deve ser aberto a todos os utilizadores, sem restrições.
2. A directoria **secure** (e todas as suas descendentes) deve ser acessível a todos os utilizadores, mas apenas através de sessões protegidas com TLS/SSL, utilizando chaves de pelo menos 128 bits, e os algoritmos AES, HMAC com SHA-256, e autenticação RSA.
3. A directoria **secure/private** deve ser acessível apenas a utilizadores dotados de um certificado de cliente emitido dentro da hierarquia de CSSI/GSI. Esta página deve automaticamente visualizar os dados do certificado do cliente.

Como valorização, desenvolva um pequeno exemplo que utilize a autenticação baseada no certificado do cliente para o autenticar ao nível da aplicação.