UNIVERSIDADE FEDERAL DE SANTA CATARINA - CAMPUS TRINDADE

DESAFIO EM JAVA - LABSEC

Candidato: Lucas Brand Samuel Martins

Florianópolis 2024

SUMÁRIO

1. Primeira Etapa

1.1 Introdução ao Desafio

Na primeira etapa o objetivo era criar um resumo criptográfico do arquivo textoPlano.txt com o algoritmo “SHA-256”, e salvá-lo em disco no caminho indicado.

1.2 Implementação

1.2.1 PrimeiraEtapa

1.2.1.1 executarEtapa: esse método começa criando uma nova instância para a classe Resumidor [1.2.2], após isso converte o caminho do arquivo ‘textoplano.txt’ para o tipo File, já que o método ‘resumir’ [1.2.2.1] recebe como entrada uma variável desse tipo, após chamar o método, que gera um resumo criptográfico do arquivo, imprime um pequeno texto para confirmar que a etapa foi concluída, a variável gerada, com o nome de ‘resumoCriptografico’, é então enviada para o método ‘escreveResumoEmDisco’ [1.2.2.2] para ser salva em disco no local especificado, imprimindo uma mensagem de sucesso após a conclusão.

1.2.2 Resumidor

A classe Resumidor utiliza a biblioteca padrão do java.security para manipulação de mensagens criptográficas, a classe MessageDigest (md). No construtor do Resumidor temos a inicialização do atributo algoritmo com o que foi especificado na classe Constantes, esse algoritmo é então passado como parâmetro para a instanciação do md.

1.2.2.1 resumir: esse método recebe como entrada um arquivo para ser processado. Para isso, o método “digest()”, usado para efetuar o resumo, necessita de um array de bytes como entrada, demonstrado na documentação da classe do md [1], os bytes do arquivo são lidos utilizando o método readAllBytes da classe Files e, em seguida, o “digest()” é usado para calcular o resumo criptográfico. O resultado é retornado como uma array de bytes.

1.2.2.2 escreveResumoEmDisco: o método recebe como entrada uma array de bytes contendo o resumo criptográfico previamente gerado e uma string contendo o caminho para onde o arquivo deverá ser escrito. O método converte os bytes para uma representação hexadecimal e escreve esses dados no arquivo utilizando um FileOutputStream.

2. Segunda Etapa

2.1 Introdução ao Desafio

A etapa tem como objetivo gerar pares de chaves usando o algoritmo ECDSA, um com o tamanho de 256 bits para o usuário e outro de 521 bits para a AC-Raiz, ambos pares devem ser salvos em arquivos .pem no final.

2.2 Implementação

2.2.1 SegundaEtapa

2.2.1.1 executarEtapa: este método inicia criando uma instância da classe 'GeradorDeChaves' [2.2.2] com o algoritmo especificado, então é chamado duas vezes o método 'gerarParDeChaves' [2.2.2.1], gerando um par com chaves de tamanho de 256 para o usuário e um par com tamanho de 521 para a AC-Raiz, cada vez imprimindo uma pequena mensagem demonstrando a conclusão do método, após isso, é chamado o método 'escreveChaveEmDisco' [2.2.3.1] quatro vezes para salvar as chaves públicas e privadas do usuário e da AC-Raiz.

2.2.2 GeradorDeChaves

A classe GeradorDeChaves, utiliza principalmente a classe KeyPairGenerator (generator), padrão do java.security para a geração de chaves. No construtor da classe, é recebido como entrada o algoritmo que será utilizado para a geração das chaves, ele é então utilizado para a instanciação do generator.

2.2.2.1 gerarParDeChaves: o método é responsável por gerar um par de chaves com o tamanho especificado. Ele utiliza o gerador de chaves inicializado no construtor para gerar um par de chaves com o tamanho desejado e retorna esse par de chaves.

2.2.3 EscritorDeChaves

2.2.3.1 escreveChaveEmDisco: o método recebe como entrada, uma chave, um caminho para onde a chave será salva, e uma descrição que será usada no arquivo, o tipo da chave neste caso. Para salvar a chave em um arquivo do tipo pem, foi utilizado a classe PemWritter da biblioteca BouncyCastle, a classe necessita como entrada um PemObject, como descrito em sua documentação [2], que por sua vez necessita como entrada uma descrição do tipo do objeto e uma array de bytes [3], que obtemos usando "chave.getEncoded()", ao criar o objeto ele é então inserido no arquivo pem com "pemWriter.writeObject(pemObject)".

2.2.4 LeitorDeChaves

2.2.4.1 lerChavePrivadaDoDisco: o método recebe como entrada uma string indicando o caminho onde a chave está salva e o algoritmo que foi usado para gerar a chave. Com a classe PemReader da biblioteca BouncyCastle, usando os métodos "readPemObject.getContent()" é possível pegar as especificações da chave, e com a classe KeyFactory retornamos uma chave privada a partir das especificações utilizando "KeyFactory.getInstance(algoritmo).generatePrivate(keySpecs) " [5].

2.2.4.2 lerChavePublicaDoDisco: este método é análogo ao lerChavePrivada [3.2.1.1], com uma pequena mudança no tipo que é retornado ao lermos o arquivo pem e pegarmos suas especificações, que passa a ser do tipo X509EncodedKeySpecs, isso se

deve ao fato do método “generatePublic()” não aceitar dados do tipo PKCS8EncodedKeySpec.

3. Terceira Etapa

3.1 Introdução ao Desafio

Esta etapa consiste em gerar dois certificados do tipo X.509, um auto assinado para a AC-Raiz e outro para o usuário, este assinado pela AC-Raiz. E ao final salvar ambos arquivos em formato pem.

3.2 Implementação

3.2.1 TerceiraEtapa

3.2.1.1 executarEtapa: este método começa criando uma instância do ‘GeradorDeCertificados’ [3.2.2], após isso, utilizando o ‘LeitorDeChaves’ [2.2.4], são adquiridas as chaves públicas do usuário e da AC-Raiz, assim como a chave privada da AC.

O método é dividido em duas partes, na primeira parte é criado o certificado da AC-Raiz, o processo começa criando uma estrutura do certificado (TBSCertificate, to be signed certificate) usando o método ‘gerarEstruturaCertificado’ [3.2.2.1], é enviado como entrada, a chave pública da AC, o número de série do certificado, o nome do subject, como o certificado é auto assinado, o nome da AC-Raiz, o nome do issuer e a validade do certificado, configurado em 7 dias neste código, é então criado uma string de bits do valor da assinatura da AC, com o seu ‘tbsCertificate’ e a chave privada do issuer, usando o método ‘geraValorDaAssinaturaDoCertificado’ [3.2.2.2], com a bit string feita é chamado o método ‘gerarCertificado()’ [3.2.2.3] que irá gerar o certificado, tendo como entrada a estrutura do certificado, o algoritmo que será usado, algoritmo esse que é adquirido usando o método ‘find(Constantes.algoritmoAssinatura)’ da classe ‘DefaultSignatureAlgorithmFinder’ [4], e a string de bits. Com o certificado gerado, é chamado ‘escreveCertificados’ [3.2.3.1], que finalizará o processo.

Na segunda parte do código, o mesmo processo é repetido com a diferença que o subject do certificado agora será o usuário.

3.2.2 GeradorDeCertificados

3.2.2.1 gerarEstruturaCertificado: o método recebe uma chave pública do subject, o número de série do certificado, o subject, o issuer e a quantidade de dias da validade. As informações e os seus tipos necessários foram retirados da documentação da classe ‘V3TBSCertificateGenerator’ [6], os dados que serão usados são convertidos para o tipo requerido e são inseridos com setters no ‘tbsGen’, ao final é retornado um TBSCertificate com o método ‘generateTBSCertificate()’.

3.2.2.2 gerarValorDaAssinaturaCertificado: este método tem como entrada um TBSCertificate e uma chave privada da autoridade certificadora. Primeiramente, pegamos uma instância da classe Signature com o algoritmo a ser usado, usamos o método ‘initSign(chavePrivadaAc)’ para iniciar o processo de assinatura e settar a chave a ser

usada, depois são adicionado os bytes do TBSCertificate com o ‘update()’, no final do método é retornado uma DERBitString contendo os bytes gerados pelo ‘sign()’, o uso da classe foi apresentada na seção 6.1 de um tópico sobre assinaturas digitais do site Baeldung [7].

3.2.2.3 gerarCertificado: o método recebe como entrada uma estrutura de certificado, um algoritmo que será usado e o valor da assinatura. Para gerar o certificado era necessário usar o método ‘engineGenerateCertificate()’ da classe ‘CertificateFactory’ [8], esse por sua vez necessita de um ‘inputStream’ como entrada, por isso as informações necessárias foram inseridas em um ASN1EncodableVector [9] e transformadas em uma DERSequence [10] que é transformada em uma ByteArrayInputStream [11] com o método ‘getEncoded()’, a CertificateFactory pode ser usada corretamente agora, retornando um certificado tipado em X509Certificate.

3.2.3 EscritorDeCertificados

3.2.3.1 escreveCertificado: recebe como entrada o nome de arquivo pem em que será salvo os dados do certificado, os bytes do certificado e a descrição do objeto, “CERTIFICATE” neste caso. O método funciona de maneira análoga ao método ‘escreveChaveEmDisco’ [2.2.3.1], usando o PemWriter [2] e obtendo um PemObject [3] e o inserindo no arquivo com ‘writeObject()’.

3.2.4 LeitorDeCertificados

3.2.4.1 lerCertificadoDoDisco: funciona de forma parecida aos métodos da classe ‘LeitorDeChaves’ [2.2.4], é inserido uma instância do tipo do certificado na classe ‘CertificateFactory’ [12] e fornecido um ‘FileInputStream’ com o arquivo que está salvo o certificado que vai ser lido, o método retorna um certificado gerado a partir do método ‘generateCertificate()’.

4. Quarta Etapa

4.1 Introdução ao Desafio

A quarta etapa tem como finalidade gerar dois repositórios seguros de chaves assimétricas no formato PKCS#12, um para a AC-Raiz e outro para o usuário, ambos os repositórios devem conter uma senha, um alias, um certificado e uma chave privada.

4.2 Implementação

4.2.1 QuartaEtapa

4.2.1.1 executarEtapa: o método começa adquirindo informações necessárias para a criação do repositório do usuário, a chave privada, usando ‘lerChavePrivadaDoDisco()’ [2.2.4.1], o certificado, usando ‘lerCertificadoDoDisco()’ [3.2.4.1], e a fim de simplificar a leitura do código, as variáveis indicadas na classe Constantes são salvas. É então chamado o método ‘gerarPkcs12()’ [4.2.2.1], que gera um repositório e o salva em disco, após a

criação, uma mensagem de conclusão é impressa na tela. A geração do repositório da AC-Raiz é feita da mesma maneira, com a diferença dos dados salvos.

4.2.2 GeradorDeRepositorios

4.2.2.1 gerarPkcs12: recebe como entrada uma chave privada, um certificado do tipo X509, uma string contendo o local que será salvo o repositório, um alias e uma senha. A classe 'KeyStore' foi usada para gerar o repositório, em um primeiro momento instanciamos a classe com o tipo de repositório que será criado, carregamos ele com a senha e se define a entrada de uma chave enviando através do método 'setKeyEntry()' um alias, uma chave privada, a senha e uma cadeia de certificados com o certificado recebido na entrada. O repositório é então salvo no caminho especificado usando 'store(fos, senha)', que o armazena com a proteção de uma senha. O uso da classe foi apresentado em um tópico sobre repositórios usando KeyStore no site Baeldung [13].

4.2.3 RepositorioChaves

A classe consiste em um construtor que recebe o formato do repositório, uma senha e um alias, na classe é possível inicializá-la com o método 'abrir()' que recupera o repositório salvo em disco e o deixa disponível para uso através da classe com o auxílio de getters.

5. Quinta Etapa

5.1 Introdução ao Desafio

Esta etapa tem como finalidade a criação de uma assinatura usando o algoritmo de resumo criptográfico sha-256 e o algoritmo de criptografia assimétrica ECDSA.

5.2 Implementação

5.2.1 QuintaEtapa

5.2.1.1 executarEtapa: o método começa abrindo o repositório do usuário com a classe 'RepositorioChaves' [4.2.3], é então inicializado o 'GeradorDeAssinatura' [5.2.2] e os dados do assinantes são informados através do 'informaAssinante()' [5.2.2.1], então o método 'assinar()' [5.2.2.2] é chamado retornando uma assinatura do tipo CMSSignedData que é salva no arquivo especificado com o método 'escreveAssinatura() [5.2.2.5].

5.2.2 GeradorDeAssinatura

5.2.2.1 informaAssinante: funciona como setter do certificado e da chave privada do assinante.

5.2.2.2 assinar: o método recebe como entrada o caminho indicando o local que será salvo a assinatura. Começa criando uma lista de certificados para inserir o certificado salvo no setter 'informaAssinante()' [5.2.2.1], então é chamada a função

‘preparaDadosParaAssinar()’ [5.2.2.3] que retorna o documento em um formato compatível para ser assinado, usando o método ‘preparaInformacoesAssinante()’ [5.2.2.4] se consegue gerar uma estrutura necessária para o processo de assinatura, os dados gerados são adicionados ao gerador instanciado no construtor da classe, no final, o método retorna através do ‘generate()’ a assinatura finalizada. O exemplo usado para assinatura do documento foi fornecido na documentação da classe ‘CMSSignedDataGenerator’ [14]

5.2.2.3 preparaDadosParaAssinar: o método recebe o local que o documento que será usado está e o transforma numa array de bytes como se faz no exemplo da documentação da classe ‘CMSSignedGenerator’ [14], retornando ao final, o documento corretamente convertido através do ‘CMSProcessableByteArray’.

5.2.2.4 preparaInformacoesAssinante: converte o certificado e a chave privada que serão usados para a assinatura, retorna um ‘SignerInfoGenerator’. O processo para a conversão dos dados foi retirado de um exemplo fornecido na documentação da classe ‘CMSSignedGenerator’ [14].

5.2.2.5 escreveAssinatura: o método recebe como entrada um arquivo e uma assinatura que é salva no arquivo enviado.

6. Sexta Etapa

6.1 Introdução ao Desafio

A última etapa consiste em verificar a integridade da assinatura gerada no processo anterior.

6.2 Implementação

6.2.1 SextaEtapa

6.2.1.1 executarEtapa: inicia adquirindo a assinatura feita na etapa anterior e a transformando em uma byte array para ser utilizada na criação de um ‘CMSSignedData’, esse dado é então enviado, junto a um certificado, para o método ‘verificarAssinatura()’ [6.2.2.1], que retorna um booleano, se ele for verdadeiro a assinatura é válida e uma mensagem é impressa na tela confirmando a sua validade.

6.2.2 VerificadorDeAssinatura

6.2.2.1 verificarAssinatura: o método começa gerando um verificador de assinatura através do ‘geraVerificadorInformacoesAssinatura()’ [6.2.2.2], após isso, com o método ‘pegaInformacoesAssinatura()’ [6.2.2.3] adquire informações de dentro do ‘CMS’, no final o método retorna o resultado da verificação feita através do método ‘verify()’. A exemplificação do método foi vista no github do repositório poreid, no arquivo de nome SOD.java [15].

6.2.2.2 geraVerificadorInformacoesAssinatura: este método gera um verificador de assinaturas baseado no certificado do assinante, ele adiciona os dados requisitados na documentação ao 'SignerInformationVerifier' [16]. A exemplificação do método foi vista no github do repositório poreid, no arquivo de nome SOD.java [15].

6.2.2.3 pegaInformacoesAssinatura: recebe como entrada uma assinatura, e através dos métodos 'assinatura.getSignerInfos().getSigners().iterator().next()' retorna as informações da primeira assinatura encontrada dentro do CMS. A exemplificação do método foi vista no github do repositório poreid, no arquivo de nome SOD.java [15].

7. Referências

[1]https://docs.oracle.com/javase/8/docs/api/java/security/MessageDigest.html
[2]https://www.bouncycastle.org/docs/docs1.8on/org/bouncycastle/util/io/pem/PemWriter.html
[3]https://www.bouncycastle.org/docs/docs1.8on/org/bouncycastle/util/io/pem/PemObject.html
[4]https://www.bouncycastle.org/docs/pkixdocs1.5on/org/bouncycastle/operator/DefaultSignatureAlgorithmIdentifierFinder.html
[5]https://www.baeldung.com/java-read-pem-file-keys
[6]https://www.bouncycastle.org/docs/docs1.8on/org/bouncycastle/asn1/x509/V3TBSCertificateGenerator.html
[7]https://www.baeldung.com/java-digital-signature
[8]https://www.bouncycastle.org/docs/docs1.8on/org/bouncycastle/jcajce/provider/asymmetric/x509/CertificateFactory.html
[9]https://www.bouncycastle.org/docs/docs1.8on/org/bouncycastle/asn1/ASN1EncodableVector.html
[10]https://www.bouncycastle.org/docs/docs1.8on/org/bouncycastle/asn1/DERSequence.html
[11]https://docs.oracle.com/javase/8/docs/api/java/io/ByteArrayInputStream.html
[12]https://docs.oracle.com/javase/8/docs/api/java/security/cert/CertificateFactory.html
[13]https://www.baeldung.com/java-keystore
[14]https://www.bouncycastle.org/docs/pkixdocs1.5on/org/bouncycastle/cms/CMSSignedDataGenerator.html
[15]https://github.com/poreid/poreid/blob/master/sod.verify/src/main/java/org/poreid/verify/sod/SOD.java#L89
[16]https://javadoc.io/static/org.bouncycastle/bcpkix-jdk15on/1.68/org/bouncycastle/cms/SignerInformationVerifier.html