Relatório Seleção PBAD/LabSEC - Etapa Final

Maria Eduarda Teixeira Costa

July 2025

1 Introdução

Esta etapa do processo seletivo teve como objetivo a implementação de operações fundamentais relacionadas à segurança computacional.

A aplicação foi desenvolvida utilizando a linguagem Java, com apoio da biblioteca Bouncy Castle, em ambiente de desenvolvimento IntelliJ IDEA.

Durante o desenvolvimento, optei por documentar o código com comentários explicativos para facilitar a compreensão do funcionamento individual de cada método. Também utilizei referências externas para embasar as decisões técnicas adotadas ao longo do projeto.

O desafio foi dividido em seis etapas:

2 Etapas

1. **Primeira Etapa - Obter o resumo criptográfico de um documento**

   Para gerar o resumo criptográfico de um documento, foi utilizada a classe MessageDigest, que fornece a funcionalidade de calcular um resumo hash — uma sequência de bytes de tamanho fixo — a partir de uma entrada de tamanho arbitrário. O algoritmo utilizado foi o SHA-256, conforme definido na classe Constantes.

   Após a obtenção do array de bytes referente ao resumo, foi realizada sua conversão para o formato hexadecimal. Para isso, utilizou-se a classe StringBuilder, os bytes convertidos em strings hexadecimais utilizando o método String.format().

   Por fim, o resumo em formato de texto foi escrito em um arquivo no disco. A escrita foi realizada com a classe BufferedWriter, que abre (ou cria) o arquivo no caminho especificado, e o método write(), responsável por gravar o conteúdo no arquivo.

2. **Segunda Etapa - Gerar chaves assimétricas**

   Para a geração do par de chaves assimétricas, foi utilizada a classe KeyPairGenerator. Conforme definido na classe Constantes, o algoritmo adotado foi o "EC"

(ECDSA). Também foi empregada a classe SecureRandom, que fornece uma fonte de aleatoriedade criptograficamente segura, essencial para garantir a imprevisibilidade na geração das chaves.

Para armazenar as chaves assimétricas geradas, foi utilizada a classe EscritorDeChaves, que escreve tanto chaves públicas quanto privadas em arquivos no formato PEM. Foi utilizado JcaPEMWriter para escrever as chaves em formato PEM. No caso de chaves privadas, é utilizado o JcaPKCS8Generator para garantir que a chave seja salva no formato PKCS#8, adequado para chaves privadas. Já as chaves públicas são salvas diretamente no formato PEM sem necessidade de conversão.

Para a leitura, a classe LeitorDeChaves usa o PEMParser para interpretar o conteúdo do arquivo PEM e, dependendo do tipo da chave, converte o objeto lido para o tipo adequado (PrivateKey ou PublicKey) utilizando o JcaPEMKeyConverter.

3. **Terceira Etapa - Gerar certificados digitais**

   O primeiro passo para a criação de um certificado digital é a construção de sua estrutura da parte assinável do certificado. Essa tarefa é realizada pelo método gerarEstruturaCertificado, que retorna um objeto do tipo TBSCertificate (To Be Signed Certificate) através da classe V3TBSCertificateGenerator.

   A assinatura digital é o mecanismo que garante a integridade e autenticidade do certificado digital. No método geraValorDaAssinaturaCertificado, a estrutura do certificado TBSCertificate é codificada em bytes e assinada com a chave privada da Autoridade Certificadora (AC) usando a classe Signature (a chave obtemos na etapa anterior). A assinatura resultante é retornada como um objeto do tipo DERBit-String.

   O método gerarCertificado é responsável por construir o certificado digital X.509 a partir de três elementos: a estrutura do certificado TBSCertificate, o identificador do algoritmo de assinatura AlgorithmIdentifier e a assinatura digital DERBitString. Esses dados são agrupados em um vetor ASN.1 ASN1EncodableVector e convertidos em uma sequência DERSequence. Por fim, essa sequência codificada é transformada em um objeto X509Certificate utilizando a CertificateFactory, finalizando assim a criação do certificado digital.

   Para escrever o certificado no disco, da mesma forma que as chaves foram escritas, foi feito o uso da classe JcaPemWriter para salvar os arquivos em formato pem.

   A leitura do certificado também é parecida com a leitura das chaves, utilizando o PEMParser, só que aqui foi utilizada a classe JcaX509CertificateConverter para fazer a conversão para o formato adequado (de X509CertificateHolder para X509Certificate).

4. **Quarta Etapa - Gerar repositório de chaves seguro**

   O processo de criação de um repositório consistiu em usar a classe KeyStore com o formato PKCS#12, inicialmente vazio e protegido pela senha, depois inserir suas entradas alias, chave privada e certificado. Por fim, salvar o repositório no disco com o método store() da própria classe KeyStore.

5. **Quinta Etapa - Gerar uma assinatura digital**

   Para a criação de uma assinatura digital com algoritmo de resumo criptográfico SHA-256 e e algoritmo de assinatura ECDSA foi feito o uso da classe CMSSignedDataGenerator que cria uma assinatura no padrão CMS.

   Alguns métodos auxiliares foram implementados para realizar esse processo. Primeiramente o método informaAssinante serve para informar o certificado e a chave privada do assinante. O método preparaDadosParaAssinar lê os bytes do arquivo do caminho indicado e retorna um CMSProcessableByteArray que implementa a interface CMSTypedData. O método preparaInformacoesAssinante utiliza a classe JcaSimpleSignerInfoGeneratorBuilder para construir um SignerInfoGenerator que contém todas as informações necessárias sobre o assinante, incluindo seu certificado digital e chave privada, além do algoritmo de assinatura a ser utilizado. O método assinar é responsável por gerar a assinatura digital. Esse método utiliza as funções previamente implementadas, além disso, adiciona o certificado digital do assinante à estrutura com auxílio da classe X509CertificateHolder, e o método generate é utilizado para gerar o objeto CMSSignedData.

   Por fim, para escrever a assinatura no disco só foi necessário o uso do método write() que apenas codifica os dados no formato DER.

6. **Sexta Etapa - Verificar uma assinatura digital**

   Assim como na etapa anterior, nesta etapa foram criados métodos auxiliares para um método principal. O método geraVerificadorInformacoesAssinatura() utiliza a classe JcaSimpleSignerInfoVerifierBuilder para criar um verificador de assinatura CMS com base nas informações do assinante. O método pegaInformacoesAssinatura retorna um SignerInformation que contém as informações extraídas do documento CMS. Para realizar a verificação, o método verificarAssinatura faz uso desses métodos auxiliares e, em seguida, utiliza o método verify da classe SignerInformation, o qual retorna true ou false, indicando a validade da assinatura.