Pedro Henrique Scheidt Prazeres

Relatório do Desafio do Processo Seletivo Labsec

Florianópolis
2025.1

Pedro Henrique Scheidt Prazeres

Relatório do Desafio do Processo Seletivo Labsec

Este relatório foi elaborado para o processo seletivo do LabSEC (UFSC) em 2025.1, com foco na implementação de soluções em Java para tarefas de criptografia, como geração de resumos criptográficos, chaves e certificados digitais, além da assinatura e verificação de documentos. O objetivo é demonstrar domínio prático dos conceitos de segurança da informação por meio do desenvolvimento e aplicação dessas funcionalidades.

Universidade Federal de Santa Catarina

Florianópolis
2025.1

RESUMO

Este foi um desafio muito interessante, aprendi muito com ele sobre criação e manipulação de várias classes e métodos relacionados a criptografia em Java. Contudo, dado que o objetivo do trabalho é observar as soluções dos candidatos, e não simplesmente implementar as funções fornecidas, eu elegi fazer modificações nos códigos de várias funções, com objetivo de tornar o código mais escalável em algumas áreas, como permitir parâmetros opcionais, ou as vezes mudando os parâmetros (como na etapa 4). Independentemente, essas alterações são explicitadas neste documento, e é possível verificá-las nos relatórios de cada etapa.

Outra observação, disponibilizar a chave privada em um repositório pública (como o meu repositório github Scheidt/Desafio-Labsec) seria uma grave falha de segurança em um ambiente real de segurança, mas dada a natureza deste desafio como algo educativo e de demonstrar meus conhecimentos, e uma vez que esta chave privada não será usada para real troca de mensagens, decidi que publicá-la no meu Github não representa real problema. Ademais, ela estar disponível mostra o ambiente completo que utilizei ao fazer o desafio.

Ademais, elegi deixar no código as verificações de debugging que utilizei para verificar se o código está funcionando corretamente. Naturalmente eu testei o código antes de enviar e verifiquei que este estava funcionando, e, portanto, seria possível remover estes trechos do código. Contudo, decidi por manter estas partes do código, para mostrar ao(à) avaliador(a) o método por qual verifiquei a funcionalidade do código.

SUMÁRIO

1 RESUMO CRIPTOGRÁFICO

1.1 PrimeiraEtapa.java

A primeira etapa consiste em implementar um código capaz de criar o resumo criptográfico de um texto:

```
1 package br.ufsc.labsec.pbad.hiring.etapas;
2
3 public class PrimeiraEtapa {
4     public static void executarEtapa() {
5         // TODO implementar
6     }
7 }
```

Inicialmente, eu havia simplesmente implementado o código inteiro no PrimeiraEtapa.java, contudo, posteriormente observei a existência do arquivo “criptografia” que serviria de *utils* para a execução das etapas.

```
1  public class PrimeiraEtapa {
2
3      public static void executarEtapa() {
4
5          try {
6              MessageDigest hasher = MessageDigest.getInstance(Constantes.
                   algoritmoCResumo);
7
8              Path caminhoTexto = Paths.get(Constantes.caminhoTextoPlano);
9              Path caminhoOutput = Paths.get(Constantes.caminhoResumoCriptografico);
10
11             byte[] bytesTextoPlano = Files.readAllBytes(caminhoTexto);
12
13             byte[] hashed = hasher.digest(bytesTextoPlano);
14
15             String hashHex = hashPraHex(hashed);
16
17             Files.createDirectories(caminhoOutput.getParent());
18             Files.write(caminhoOutput, hashHex.getBytes());
19
20             System.out.println("Sucesso na primeira etapa :)");
21
22         } catch (NoSuchAlgorithmException | IOException e) {
23             e.printStackTrace();
24         }
25     }
26
27     public static String hashPraHex(byte[] hash) {
28         HexFormat hexFormat = HexFormat.of();
29         return hexFormat.formatHex(hash);
30     }
31 }
```

Portanto, modifiquei o código para alinhar ao formato recomendado, servindo melhor como uma biblioteca, que poderia ser utilizada para outras tarefas (modularidade).

1.2 Resumidor.java

Decidi modificar a função “resumir”, o recomendado era o envio do arquivo como um “File”, contudo, considero melhor enviar o caminho do arquivo, ao invés do arquivo em si, pois dependendo do tamanho do arquivo, ele pode ser muito pesado e adicionar o delay de enviar o arquivo entre funções. Esse problema poderia ser resolvido também enviando um pointer do File, mas essa é a forma mais simples.

```java
/**
 * Calcula o resumo criptográfico do arquivo indicado.
 *
 * @param caminhoTexto caminho do arquivo a ser processado.
 * @return Bytes do resumo.
 */
public byte[] resumir(Path caminhoTexto) throws IOException {
    try {
        byte[] textoBytes = Files.readAllBytes(caminhoTexto);
        return md.digest(textoBytes);
    } catch (IOException e) {
        throw new IOException("Erro com o caminho: " + caminhoTexto, e);
    }
}
```

Após iniciar a etapa dois, percebi que seria melhor incluir um argumento opcional, para caso o usuário queira utilizar um algoritmo diferente:

```java
/**
 * Classe responsável por executar a função de resumo criptográfico.
 *
 * @see MessageDigest
 */
public class Resumidor {

    private MessageDigest md;
    private String algoritmo;

    /**
     * Construtor.
     */
    public Resumidor() throws NoSuchAlgorithmException {
        this.algoritmo = Constantes.algoritmoResumo;
        try {
            this.md = MessageDigest.getInstance(this.algoritmo);
        } catch (NoSuchAlgorithmException e) {
            throw new NoSuchAlgorithmException("Erro: Não foi reconhecido algoritmo: "
                + this.algoritmo, e);
        }
    }

    /**
```

```
24      * Constructor com parâmetro optativo para outro algoritmo
25      * @param algoritmo string com o nome do algoritmo utilizado
26      */
27
28     public Resumidor(String algoritmo) throws NoSuchAlgorithmException {
29         this.algoritmo = algoritmo;
30         try {
31             this.md = MessageDigest.getInstance(this.algoritmo);
32         } catch (NoSuchAlgorithmException e) {
33             throw new NoSuchAlgorithmException("Erro: Não foi reconhecido algoritmo: "
                   + this.algoritmo, e);
34         }
35     }
```

As outras funções utilizadas são simples e implementam argumentos nativos dos tipos utilizados:

```
1      /**
2       * Calcula o resumo criptográfico do arquivo indicado.
3       *
4       * @param caminhoTexto caminho do arquivo a ser processado.
5       * @return Bytes do resumo.
6       */
7      public byte[] resumir(Path caminhoTexto) throws IOException {
8          try {
9              byte[] textoBytes = Files.readAllBytes(caminhoTexto);
10             return md.digest(textoBytes);
11         } catch (IOException e) {
12             throw new IOException("Erro com o caminho: " + caminhoTexto, e);
13         }
14     }
15
16     /**
17      * Escreve o resumo criptográfico no local indicado.
18      *
19      * @param resumo          resumo criptográfico em bytes.
20      * @param caminhoArquivo caminho do arquivo.
21      */
22     public void escreveResumoEmDisco(byte[] resumo, Path caminhoArquivo) throws
           IOException {
23         HexFormat formatadorHex = HexFormat.of();
24         String resumoHex = formatadorHex.formatHex(resumo);
25
26         try {
27             Files.createDirectories(caminhoArquivo.getParent());
28             Files.write(caminhoArquivo, resumoHex.getBytes());
29         } catch (IOException e) {
30             throw new IOException("Erro no caminho: " + caminhoArquivo, e);
31         }
32     }
33
34 }
```

1.3 PrimeiraEtapa.java (continuado)

Por fim, no Arquivo PrimeiraEtapa temos a criação do resumo criptográfico (junto do tratamento de dados):

```
1  public class PrimeiraEtapa {
2
3      public static void executarEtapa() {
4          System.out.println("Início Etapa 1");
5          try {
6              Path caminhoTexto = Paths.get(Constantes.caminhoTextoPlano);
7              Path caminhoOutput = Paths.get(Constantes.caminhoResumoCriptografico);
8
9              Resumidor resumidor = new Resumidor();
10             byte[] resumo = resumidor.resumir(caminhoTexto);
11             resumidor.escreveResumoEmDisco(resumo, caminhoOutput);
12             System.out.println("Sucesso na Etapa 1!");
13         } catch (NoSuchAlgorithmException | IOException e) {
14             System.err.println("Erro ao executar a Primeira Etapa: " + e.getMessage());
15         }
16     }
17 }
```

Verifiquei se o resumo estava correto por meio deste site (EMN178, 2025), colocando o conteúdo do arquivo de texto e verificando se o resumo gerado era igual ao do site.

Referências

BAELDUNG. **SHA-256 Hashing in Java**. 2023. Disponível em: <https://www.baeldung.com/sha-256-hashing-java>. Acesso em: 17 jul. 2025.

EMN178. **SHA256 Online Tool**. Disponível em: <https://emn178.github.io/online-tools/sha256.html>. Acesso em: 17 jul. 2025.

ORACLE CORPORATION. **Class HexFormat (Java SE 17 | Java API Documentation)**. 2021. Disponível em: <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/HexFormat.html>. Acesso em: 17 jul. 2025.

2 GERAR CHAVES ASSIMÉTRICAS

Agora na etapa 2, tendo em vista a pasta “criptografia” consegui realizar a tarefa mais facilmente, tendo os métodos e atributos recomendados.

Importante notar também que, ao commitar o código pronto desta etapa, também enviei a chave privada gerada. Reconheço que isso seria uma vulnerabilidade de segurança, caso se tratasse de um projeto real, mas dado que é um exercício, e essa chave privada não será usada para criptografar dados privados, elegi manter a chave privada no repositório.

2.1 GeradorDeChaves.java

Observei os tipos dos atributos, e estudei a documentação dos tipos destes atributos, assim vendo os métodos que ele permite (nominalmente o “KeyPairGenerator” (documentação está nas fontes).

Inspirado na primeira etapa, criei uma segunda função constructor, que não recebe nenhum argumento, e pega o algoritmo padrão de criação de chaves do arquivo de constantes. Assim, o parâmetro de algoritmo fica como optativo:

```
1  private String algoritmo;
2      private KeyPairGenerator generator;
3
4      static {
5          // Garante que o provedor Bouncy Castle esteja disponível
6          if (Security.getProvider("BC") == null) {
7              Security.addProvider(new org.bouncycastle.jce.provider.BouncyCastleProvider
                   ());
8          }
9
10     }
11     /**
12      * Construtor, caso não receber nenhum argumento, utiliza como default o algorítmo
            do arquivo de constantes.
13      *
14      * @throws NoSuchAlgorithmException se o algoritmo não for reconhecido.
15      * @throws NoSuchProviderException se o provedor "BC" não for encontrado.
16      */
17
18     public GeradorDeChaves() throws NoSuchAlgorithmException, NoSuchProviderException {
19         this.algoritmo = Constantes.algoritmoChave;
20         try {
21             this.generator = KeyPairGenerator.getInstance(this.algoritmo, "BC");
22         } catch (NoSuchAlgorithmException e) {
23             throw new NoSuchAlgorithmException("Erro: O algoritmo de chave '" + this.
                   algoritmo + "' não é válido ou não foi encontrado.", e);
24         } catch (NoSuchProviderException e) {
25             throw new NoSuchProviderException("Erro: O provedor de segurança 'BC' (
                   Bouncy Castle) não foi encontrado.");
```

```
26          }
27      }
28
29      /**
30       * Construtor com argumento opcional, este argumento sobrescreve o algorítmo
             utilizado como padrão.
31       *
32       * @param algoritmo algoritmo de criptografia assimétrica a ser usado.
33       * @throws NoSuchAlgorithmException se o algoritmo não for reconhecido.
34       * @throws NoSuchProviderException se o provedor "BC" não for encontrado.
35       */
36
37      public GeradorDeChaves(String algoritmo) throws NoSuchAlgorithmException,
             NoSuchProviderException {
38          this.algoritmo = algoritmo;
39          try {
40              this.generator = KeyPairGenerator.getInstance(this.algoritmo, "BC");
41          } catch (NoSuchAlgorithmException e) {
42              throw new NoSuchAlgorithmException("Erro: " + this.algoritmo + " não é
                     reconhecido como algorítmo válido", e);
43          } catch (NoSuchProviderException e) {
44              throw new NoSuchProviderException("Erro:  BC não é um provedor válido");
45          }
46      }
```

O método gerarParDeChaves é simples, inicializando o KeyPairGenerator com o tamanho de chave fornecido, e retornando um KeyPair:

```
1  /**
2   * Gera um par de chaves, usando o algoritmo definido pela classe, com o
3   * tamanho da chave especificado.
4   *
5   * @param tamanhoDaChave tamanho em bits das chaves geradas.
6   * @return Par de chaves.
7   * @see SecureRandom
8   */
9  public KeyPair gerarParDeChaves(int tamanhoDaChave) {
10     this.generator.initialize(tamanhoDaChave);
11     return this.generator.generateKeyPair();
12 }
```

Foi detectado também um erro, explicado na sessão 2.4, que necessitou a adição do BouncyCastle como provedor da geração de chaves:

```
1  public class GeradorDeChaves {
2
3      private String algoritmo;
4      private KeyPairGenerator generator;
5
6      static {
7          // Garante que o provedor Bouncy Castle esteja disponível
8          if (Security.getProvider("BC") == null) {
9              Security.addProvider(new org.bouncycastle.jce.provider.BouncyCastleProvider
                   ());
10         }
11
12     }
```

2.2 EscritorDeChaves.java

Inicialmente o caminho recomendado que encontrei envolvia definir as strings "CERTIFICATE/PRIVATE-KEY BEGIN/END" por meio de concatenação de string, verificando se a chave é privada ou pública. Em seguida, lembrei de pesquisar se a biblioteca BOuncy Castle possuia alguma função capaz de realizar essa funcionalidade, e encontrei na documentação a existência da classe *PEM-Writer*. Contudo, na mesma documentação (PROJECT, 2023) havia a informação desta classe estar depreciada e a alternativa a ser utilizada no lugar (*JcaPEMWriter*). Por fim, então, utilizei essa classe:

```java
/**
 * Essa classe é responsável por escrever uma chave assimétrica no disco. Note
 * que a chave pode ser tanto uma chave pública quanto uma chave privada.
 *
 * @see Key
 */
public class EscritorDeChaves {

    /**
     * Escreve uma chave no local indicado.
     *
     * @param chave chave assimétrica a ser escrita em disco.
     * @param nomeDoArquivo nome do local onde será escrita a chave.
     */
    public static void escreveChaveEmDisco(Key chave, String nomeDoArquivo) {
        try {
            Path caminhoOutput = Paths.get(nomeDoArquivo);
            if (caminhoOutput.getParent() != null) {
                Files.createDirectories(caminhoOutput.getParent());
            }

            try (JcaPEMWriter pemWriter = new JcaPEMWriter(new FileWriter(nomeDoArquivo
                ))) {
                pemWriter.writeObject(chave);
            }

        } catch (IOException e) {
            System.err.println("Erro ao escrever a chave PEM: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

Contudo, ao tentar ler a chave escrita por esse código, recebi um erro referente à codificação das chaves. Elas estavam sendo interpretado como um keypair incompleto na hora da leitura, e, portanto, o código necessitou ser modificado para melhor identificar se está escrevendo uma chave pública, ou uma chave privada:

```java
/**
 * Essa classe é responsável por escrever uma chave assimétrica no disco. Note
 * que a chave pode ser tanto uma chave pública quanto uma chave privada.
 * * @see Key
 */
public class EscritorDeChaves {
```

```
7
8       /**
9        * Escreve uma chave no local indicado.
10       * * @param chave          chave assimétrica a ser escrita em disco.
11       * @param nomeDoArquivo  nome do local onde será escrita a chave.
12       */
13      public static void escreveChaveEmDisco(Key chave, String nomeDoArquivo) {
14          try {
15              Path caminhoOutput = Paths.get(nomeDoArquivo);
16              if (caminhoOutput.getParent() != null) {
17                  Files.createDirectories(caminhoOutput.getParent());
18              }
19
20              String tipoPem = (chave.getFormat().equals("PKCS#8")) ? "PRIVATE KEY" : "
                    PUBLIC KEY";
21              PemObject pemObject = new PemObject(tipoPem, chave.getEncoded());
22
23              try (JcaPEMWriter pemWriter = new JcaPEMWriter(new FileWriter(nomeDoArquivo
                    ))) {
24                  pemWriter.writeObject(pemObject);
25              }
26
27          } catch (IOException e) {
28              System.err.println("Erro ao escrever a chave PEM: " + e.getMessage());
29              e.printStackTrace();
30          }
31      }
32  }
```

Contudo, essa forma de comparar strings não é elegante, e modifiquei para uma forma melhor:

```
1   public static void escreveChaveEmDisco(Key chave, String nomeDoArquivo) throws
        IOException {
2           try {
3               Path caminhoOutput = Paths.get(nomeDoArquivo);
4               if (caminhoOutput.getParent() != null) {
5                   Files.createDirectories(caminhoOutput.getParent());
6               }
7
8               final String tipoPem;
9               if (chave instanceof PrivateKey) {
10                  tipoPem = "PRIVATE KEY";
11              } else if (chave instanceof PublicKey) {
12                  tipoPem = "PUBLIC KEY";
13              } else {
14                  throw new IllegalArgumentException("A chave fornecida não é uma chave p
                        ública ou privada reconhecida.");
15              }
16
17              PemObject pemObject = new PemObject(tipoPem, chave.getEncoded());
18
19              try (Writer fileWriter = Files.newBufferedWriter(caminhoOutput,
                    StandardCharsets.UTF_8);
20                   JcaPEMWriter pemWriter = new JcaPEMWriter(fileWriter)) {
21                  pemWriter.writeObject(pemObject);
22              }
23
```

```
24          } catch (IOException e) {
25              throw new IOException("Erro ao escrever a chave PEM em: " + nomeDoArquivo,
                    e);
26          } catch (IllegalArgumentException e) {
27              throw new IOException("Erro ao tentar interpretar a chave fornecida: " + e.
                    getMessage(), e);
28          }
29      }
```

2.3 LeitorDeChaves.java

Inicialmente, tentei ler o arquivo e remover o cabeçalho utilizando um Regex como proposto neste site (BAELDUNG, 2022). Contudo recebi um erro A, e precisei modificar a leitura de forma a utilizar a biblioteca Bouncy Castle.

Para leitura das chaves, é necessário adição do Bouncy Castle como um provider válido:

```
1 static {
2         Security.addProvider(new org.bouncycastle.jce.provider.BouncyCastleProvider());
3     }
```

```
1  /**
2   * Lê a chave privada do local indicado.
3   *
4   * @param caminhoChave local do arquivo da chave privada.
5   * @param algoritmo    algoritmo de criptografia assimétrica que a chave
6   * foi gerada.
7   * @return Chave privada.
8   * @throws IOException caso ocorra um erro na leitura do arquivo ou o formato da chave
        seja inválido.
9   */
10 public static PrivateKey lerChavePrivadaDoDisco(String caminhoChave,
11                                                 String algoritmo) throws IOException {
12     try (FileReader fileReader = new FileReader(caminhoChave);
13          PEMParser pemParser = new PEMParser(fileReader)) {
14
15         Object object = pemParser.readObject();
16         JcaPEMKeyConverter converter = new JcaPEMKeyConverter().setProvider("BC");
17
18         if (object instanceof PEMKeyPair) {
19             // Se o objeto for um par de chaves, extrai a chave privada
20             PEMKeyPair pemKeyPair = (PEMKeyPair) object;
21             KeyPair keyPair = converter.getKeyPair(pemKeyPair);
22             return keyPair.getPrivate();
23         } else if (object instanceof org.bouncycastle.asn1.pkcs.PrivateKeyInfo) {
24             // Se o objeto for informações de chave privada
25             org.bouncycastle.asn1.pkcs.PrivateKeyInfo privateKeyInfo = (org.
                    bouncycastle.asn1.pkcs.PrivateKeyInfo) object;
26             return converter.getPrivateKey(privateKeyInfo);
27         } else {
28             throw new IOException("Erro: O formato da chave privada no arquivo '" +
                    caminhoChave + "' não é suportado.");
29         }
30     } catch (IOException e) {
```

```
31          throw new IOException("Erro: Falha ao ler o arquivo da chave privada: " +
                caminhoChave, e);
32      }
33  }
34
35  /**
36   * Lê a chave pública do local indicado.
37   *
38   * @param caminhoChave local do arquivo da chave pública.
39   * @param algoritmo    algoritmo de criptografia assimétrica que a chave
40   * foi gerada.
41   * @return Chave pública.
42   * @throws IOException caso ocorra um erro na leitura do arquivo ou o formato da chave
         seja inválido.
43   */
44  public static PublicKey lerChavePublicaDoDisco(String caminhoChave,
45                                                 String algoritmo) throws IOException {
46      try (FileReader fileReader = new FileReader(caminhoChave);
47           PEMParser pemParser = new PEMParser(fileReader)) {
48
49          Object object = pemParser.readObject();
50          JcaPEMKeyConverter converter = new JcaPEMKeyConverter().setProvider("BC");
51
52          if (object instanceof org.bouncycastle.asn1.x509.SubjectPublicKeyInfo) {
53              org.bouncycastle.asn1.x509.SubjectPublicKeyInfo publicKeyInfo = (org.
                    bouncycastle.asn1.x509.SubjectPublicKeyInfo) object;
54              return converter.getPublicKey(publicKeyInfo);
55          } else {
56              throw new IOException("Erro: O formato da chave pública no arquivo '" +
                    caminhoChave + "' não é suportado.");
57          }
58      } catch (IOException e) {
59          throw new IOException("Erro ao ler o arquivo da chave pública: " + caminhoChave
                , e);
60      }
61  }
```

2.4 SegundaEtapa.java

Acredito ser importante de relatar que estou sempre testando quaisquer possíveis erros que imagino no código, especialmente se é um erro de arquitetura. Então, a versão inicial do meu código utilizava um teste que julguei de extrema importância:

```
1   public class SegundaEtapa {
2
3       public static void executarEtapa() {
4           String algoritmo = Constantes.algoritmoChave;
5           GeradorDeChaves gerador = new GeradorDeChaves(algoritmo);
6           KeyPair chaves = gerador.gerarParDeChaves(256);
7           PublicKey pubKey = chaves.getPublic();
8           PrivateKey privateKey = chaves.getPrivate();
9           EscritorDeChaves.escreveChaveEmDisco(privateKey, Constantes.
                caminhoChavePrivadaUsuario);
10
```

```
11          PrivateKey priv2 = LeitorDeChaves.lerChavePrivadaDoDisco(Constantes.
                caminhoChavePrivadaUsuario, algoritmo);
12          System.err.println(privateKey.equals(priv2));
13      }
14
15  }
```

Esta linha verifica se a chave que eu gerei e a chave que carreguei são iguais. Claramente, se elas forem diferentes, um erro ocorreu no processo de salvar ou no processo de carregar a chave. Após várias tentativas, isolei o problema no fato de dois provedores diferentes serem utilizados (o provedor default do java quando a chave é criada e o Bouncy Castle quando ela é lida. E portanto, modifiquei a criação de chave para utilizar o Bouncy Castle também), assim retornando um true para esta linha.

Após essa correção, deu-se continuidade ao código. O código principal da segunda etapa cria as chaves, as salva, carrega novamente e as compara com as originais:

```
1  public class SegundaEtapa {
2
3      public static void executarEtapa() {
4          System.out.println("\nInício Etapa 2");
5          try {
6              String algoritmo = Constantes.algoritmoChave;
7              GeradorDeChaves gerador = new GeradorDeChaves(algoritmo);
8
9              // Gera e escreve as chaves do usuario em disco
10             KeyPair chavesUsuario = gerador.gerarParDeChaves(256);
11
12             PublicKey pubKeyUsuario = chavesUsuario.getPublic();
13             EscritorDeChaves.escreveChaveEmDisco(pubKeyUsuario, Constantes.
                   caminhoChavePublicaUsuario);
14             PrivateKey privateKeyUsuario = chavesUsuario.getPrivate();
15             EscritorDeChaves.escreveChaveEmDisco(privateKeyUsuario, Constantes.
                   caminhoChavePrivadaUsuario);
16
17             // Gera e escreve as chaves do AC em disco
18             KeyPair chavesAC = gerador.gerarParDeChaves(521);
19             PublicKey pubKeyAC = chavesAC.getPublic();
20             EscritorDeChaves.escreveChaveEmDisco(pubKeyAC, Constantes.
                   caminhoChavePublicaAc);
21             PrivateKey privateKeyAC = chavesAC.getPrivate();
22             EscritorDeChaves.escreveChaveEmDisco(privateKeyAC, Constantes.
                   caminhoChavePrivadaAc);
23             System.out.println("Sucesso na Etapa 2!");
24         } catch (NoSuchAlgorithmException | NoSuchProviderException | IOException e) {
25             System.err.println("Erro ao executar a Segunda Etapa: " + e.getMessage());
26         }
27     }
28
29 }
```

Referências

BAELDUNG. **How to Read PEM File to Get Public and Private Keys in Java**. 2022. Disponível em: <https://www.baeldung.com/java-read-pem-file-keys>. Acesso em: 18 jul. 2025.

BOUNCY CASTLE. **PEMWriter (bcprov-jdk14 Documentation)**. 2023. Disponível em: <https://downloads.bouncycastle.org/java/docs/bcpkix-jdk14/javadoc/org/bouncycastle/openssl/PEMWriter.html>. Acesso em: 18 jul. 2025.

ORACLE. **KeyPairGenerator (Java Platform SE 8)**. 2015. Disponível em: <https://docs.oracle.com/javase/8/docs/api/java/security/KeyPairGenerator.html>. Acesso em: 18 jul. 2025.

PROJECT, Bouncy Castle. **Class PEMWriter**. Documentação da classe `PEMWriter` no pacote `org.bouncycastle.openssl`. 2023. Disponível em: <https://downloads.bouncycastle.org/java/docs/bcpkix-jdk14-javadoc/org/bouncycastle/openssl/PEMWriter.html>. Acesso em: 23 jul. 2025.

STACK OVERFLOW. **How can I generate a valid ECDSA EC key pair?** 2016. Disponível em: <https://stackoverflow.com/questions/40155888/how-can-i-generate-a-valid-ecdsa-ec-key-pair>. Acesso em: 18 jul. 2025.

_____. **How to generate public and private key in PEM format**. 2013. Disponível em: <https://stackoverflow.com/questions/20598126/how-to-generate-public-and-private-key-in-pem-format>. Acesso em: 18 jul. 2025.

_____. **Use JcaPEMWriter to export PEM file**. 2022. Disponível em: <https://stackoverflow.com/questions/72922139/use-jcapemwriter-to-export-pem-file>. Acesso em: 18 jul. 2025.

3 GERAR CERTIFICADOS DIGITAIS

Inciei lendo a documentação daquelas bibliotecas e classes que já estavam sendo importadas por padrão em cada um dos arquivos (no template do desafio)

3.1 GeradorDeCertificados.java

Analisando a estrutura desse arquivo, o workflow deste arquivo é gerarEstruturaCertificado -> geraValorDaAssinaturaCertificado -> gerarCertificado.

Portanto implementei essas funções nesta ordem. Analisei o tipo de retorno da função, e observei a documentação desses tipos, com fim de descobrir como implementar.

Interessantemente, no processo de pesquisar como fazer esta etapa, encontrei uma pergunta no stackoverflow (STACK OVERFLOW, 2020) deste desafio de mais de 5 anos atrás.

Após uma pesquisa inicial, notei que "V3TBSCertificateGenerator" possui funções outdated (BOUNCY CASTLE, 2023b), e elegi substituir a classe de gerarEstruturaCertificado pelo "X509v3CertificateBuilder":

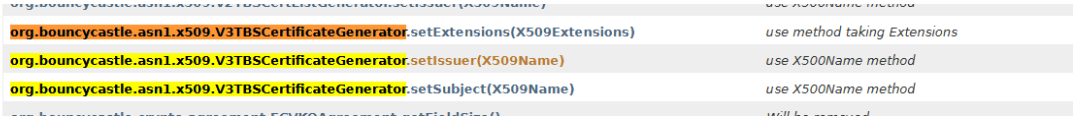

Figura 3.1.1 – Troca da classe para "X509v3CertificateBuilder"

Essa mudança permitiu tornar o código mais moderno. Criei também um parametro optativo para o algoritmo de assinatura utilizado, caso não for enviado um argumento, ele utiliza o algorítmo padrão do arquivo de constantes. Ademais, no template não era enviada a chave privada do emissor, assumo que o objetivo era que a chave fosse pega usando o caminho da chave como escrito no arquivo de contantes. Contudo, alinhado a outras mudanças que fiz no código do desafio, adicionei como argumento a chave privada, por fim de universalizar o código, tratando como uma biblioteca que será usada para outros casos:

```
1  public class GeradorDeCertificados {
2      /**
3       * Esta classe foi completamente modificada, observe no relatório minha razão para
            isso.
4       * Em suma, a forma anterior utilizava classes depreciadas, então mudei a classe
            usada e isso gerou uma modificação completa da forma do código.
5       */
6
7      /**
8       * Gera a estrutura de informações de um certificado.
9       *
10      * @param chavePublicaTitular chave pública do titular.
11      * @param chavePrivadaAc      chave privada da AC para assinar o certificado.
```

```
12      * @param numeroDeSerie       número de série do certificado.
13      * @param nomeTitular         nome do titular.
14      * @param nomeAc              nome da autoridade emissora.
15      * @param diasDeValidade      a partir da data atual, quantos dias de validade
16      * terá o certificado.
17      * @return Estrutura de informações do certificado.
18      * @throws OperatorCreationException caso ocorra um erro durante a criação da
            assinatura.
19      * @throws CertificateException      caso ocorra um erro durante a conversão do
            certificado.
20      */
21     public X509Certificate gerarCertificado(PublicKey chavePublicaTitular, PrivateKey
           chavePrivadaAc,
22                                             long numeroDeSerie, String nomeTitular,
23                                             String nomeAc, int diasDeValidade) throws
                                                   OperatorCreationException,
                                                   CertificateException {
24         return this.gerarCertificado(chavePublicaTitular, chavePrivadaAc, numeroDeSerie
               , nomeTitular, nomeAc, diasDeValidade, Constantes.algoritmoAssinatura);
25     }
26
27
28     /**
29      * Gera a estrutura de informações de um certificado. Possui um argumento optativo
30      * para escolha de um outro algoritmo de assinatura.
31      *
32      * @param chavePublicaTitular chave pública do titular.
33      * @param chavePrivadaAc      chave privada da AC para assinar o certificado.
34      * @param numeroDeSerie       número de série do certificado.
35      * @param nomeTitular         nome do titular.
36      * @param nomeAc              nome da autoridade emissora.
37      * @param diasDeValidade      a partir da data atual, quantos dias de validade
38      * terá o certificado.
39      * @param algoritmo           algoritmo de assinatura utilizado
40      * @return Estrutura de informações do certificado.
41      * @throws OperatorCreationException caso ocorra um erro durante a criação da
            assinatura.
42      * @throws CertificateException      caso ocorra um erro durante a conversão do
            certificado.
43      */
44     public X509Certificate gerarCertificado(PublicKey chavePublicaTitular, PrivateKey
           chavePrivadaAc,
45                                             long numeroDeSerie, String nomeTitular,
46                                             String nomeAc, int diasDeValidade, String
                                                   algoritmo) throws
                                                   OperatorCreationException,
                                                   CertificateException {
47         //Nomes
48         X500Name issuer = new X500Name(nomeAc);
49         X500Name subject = new X500Name(nomeTitular);
50
51         //Validade
52         Calendar calendar = Calendar.getInstance();
53         Date dataInicio = calendar.getTime();
54         calendar.add(Calendar.DAY_OF_YEAR, diasDeValidade);
55         Date dataFim = calendar.getTime();
56
57         //Criação do builder
```

```
58          X509v3CertificateBuilder builder = new JcaX509v3CertificateBuilder(
59                  issuer,
60                  BigInteger.valueOf(numeroDeSerie),
61                  dataInicio,
62                  dataFim,
63                  subject,
64                  chavePublicaTitular
65          );
66
67          X509CertificateHolder holder;
68          try {
69              //Definir algoritmo de assinatura
70              ContentSigner signer = new JcaContentSignerBuilder(algoritmo)
71                      .build(chavePrivadaAc);
72              //Build e assinatura
73              holder = builder.build(signer);
74          } catch (OperatorCreationException e) {
75              throw new OperatorCreationException("Erro durante build de algoritmo de
                    assinatura. Algoritmo utilizado: " + Constantes.algoritmoAssinatura, e)
                    ;
76          }
77
78          X509Certificate certificado;
79          try {
80              //retorna o certificado no formato X509Certificate
81              certificado = new JcaX509CertificateConverter().getCertificate(holder);
82              System.out.println("    Certificado gerado com sucesso!");
83          } catch (CertificateException e) {
84              throw new CertificateException("Erro durante conversão de certificado: ", e
                    );
85          }
86          return certificado;
87      }
88
89  }
```

3.2 LeitorDeCertificados.java

Uma função simples, a qual foi pesadamente inspirada nesta pergunta do StackOverflow (STACK OVERFLOW, 2010):

```
1  /**
2   * Classe responsável por ler um certificado do disco.
3   *
4   * @see CertificateFactory
5   */
6  public class LeitorDeCertificados {
7
8      /**
9       * Lê um certificado do local indicado.
10      *
11      * @param caminhoCertificado caminho do certificado a ser lido.
12      * @return Objeto do certificado.
13      * @throws CertificateException caso ocorra um erro ao inicializar a fábrica de
              certificados ou ao gerar o certificado.
```

```
14      * @throws IOException          caso o arquivo de certificado não seja encontrado
            ou ocorra um erro de leitura.
15      */
16     public static X509Certificate lerCertificadoDoDisco(String caminhoCertificado)
           throws CertificateException, IOException {
17         CertificateFactory certFactory;
18         try {
19             certFactory = CertificateFactory.getInstance(Constantes.formatoCertificado)
                   ;
20         } catch (CertificateException e) {
21             throw new CertificateException("Erro: O formato de certificado '" +
                   Constantes.formatoCertificado + "' não é reconhecido como um dormato vá
                   lido.", e);
22         }
23
24         try (FileInputStream is = new FileInputStream(caminhoCertificado)) {
25             X509Certificate cert = (X509Certificate) certFactory.generateCertificate(is
                   );
26             System.out.println("    Certificado lido de disco com sucesso");
27             return cert;
28         } catch (FileNotFoundException e) {
29             throw new IOException("Erro: Não encontrado arquivo de certificado  no
                   caminho: " + caminhoCertificado, e);
30         } catch (CertificateException e) {
31             throw new CertificateException("Erro: Falha ao coletar os dados do
                   certificado no arquivo: " + caminhoCertificado, e);
32         }
33     }
34
35 }
```

3.3 EscritorDeCertificados.java

Não entendi o motivo da função receber o certificado no formato Bytes[] e não X509Certificate, me parece estranho essa transformação ser feita em "TerceiraEtapa.java", mas utilizei o formato recomendado. Nesta etapa também não utilizei do objeto PEMWriter, escolhendo utilizar a biblioteca mais recente.

```
1  /**
2   * Classe responsável por escrever um certificado no disco.
3   */
4  public class EscritorDeCertificados {
5
6      /**
7       * Escreve o certificado indicado no disco.
8       *
9       * @param nomeArquivo           caminho que será escrito o certificado.
10      * @param certificadoCodificado bytes do certificado.
11      * @throws IOException caso ocorra um erro ao escrever o arquivo no disco.
12      */
13     public static void escreveCertificado(String nomeArquivo,
14                                           byte[] certificadoCodificado) throws
                                                 IOException {
15         try (JcaPEMWriter pemWriter = new JcaPEMWriter(new FileWriter(nomeArquivo))) {
```

```
16              CertificateFactory certFactory = CertificateFactory.getInstance("X.509");
17              Certificate certificate = certFactory.generateCertificate(new
                    ByteArrayInputStream(certificadoCodificado));
18
19              pemWriter.writeObject(certificate);
20
21              System.out.println("     Certificado escrito em disco com sucesso");
22          } catch (CertificateException e) {
23              throw new IOException("Erro ao decodificar os bytes do certificado.", e);
24          } catch (IOException e) {
25              throw new IOException("Erro ao escrever o arquivo de certificado no caminho
                    : " + nomeArquivo, e);
26          }
27      }
28
29  }
```

3.4 TerceiraEtapa.java

Ao rodar a primeira versão do código, o certificado gerado estava incorreto:

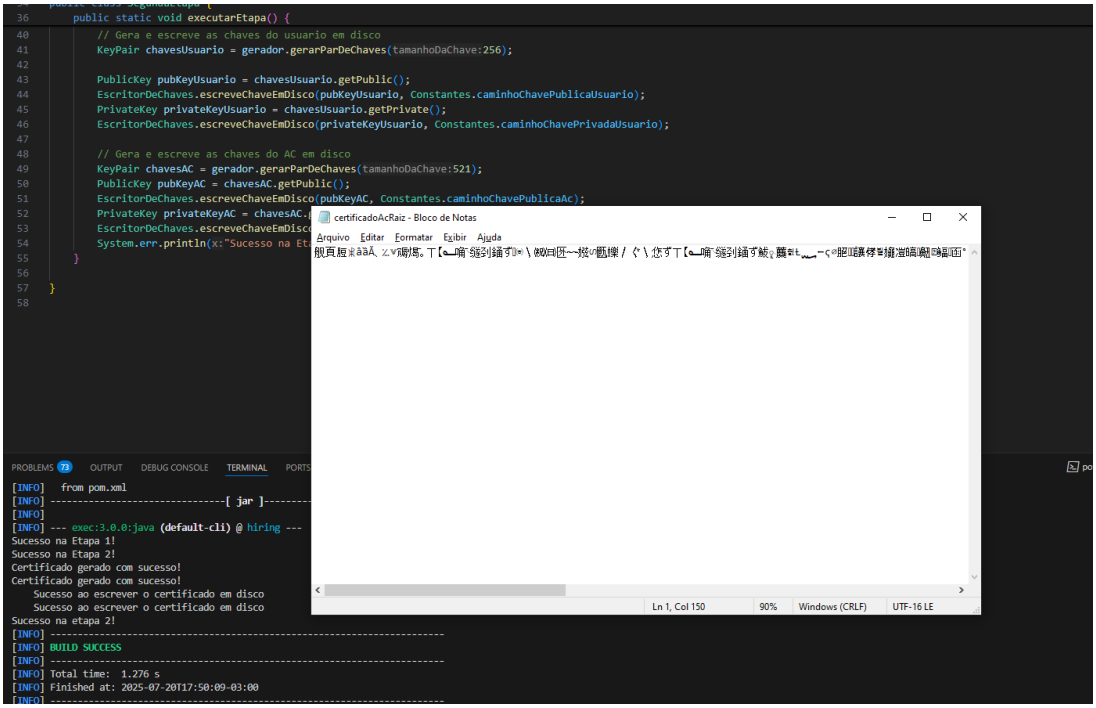

Figura 3.4.1 – Erro ao gerar certificado

Portanto, parti em busca da origem do erro, e identifiquei um erro de codificação que necessitou modificação do “EscritorDeChaves.java”, contudo, após esse erro, o código funcionou:

```
1  public class TerceiraEtapa {
2
3      public static void executarEtapa() {
4          System.out.println("\nInício Etapa 3");
5          try {
```

```
6               // Carrega chaves necessárias
7               PrivateKey privadaAc = LeitorDeChaves.lerChavePrivadaDoDisco(Constantes.
                    caminhoChavePrivadaAc, Constantes.algoritmoChave);
8               PublicKey publicaAc = LeitorDeChaves.lerChavePublicaDoDisco(Constantes.
                    caminhoChavePublicaAc, Constantes.algoritmoChave);
9               PublicKey publicaUsuario = LeitorDeChaves.lerChavePublicaDoDisco(Constantes
                    .caminhoChavePublicaUsuario, Constantes.algoritmoChave);
10
11              // Gera os certificados
12              GeradorDeCertificados geradorCert = new GeradorDeCertificados();
13
14              X509Certificate certificadoCA = geradorCert.gerarCertificado(publicaAc,
15                      privadaAc,
16                      Constantes.numeroSerieAc,
17                      Constantes.nomeAcRaiz,
18                      Constantes.nomeAcRaiz,
19                      10);
20
21              X509Certificate certificadoUsuario = geradorCert.gerarCertificado(
                    publicaUsuario,
22                      privadaAc,
23                      Constantes.numeroDeSerie,
24                      Constantes.nomeUsuario,
25                      Constantes.nomeAcRaiz,
26                      5);
27
28              // Escreve os certificados em disco
29              EscritorDeCertificados.escreveCertificado(Constantes.
                    caminhoCertificadoAcRaiz, certificadoCA.getEncoded());
30              EscritorDeCertificados.escreveCertificado(Constantes.
                    caminhoCertificadoUsuario, certificadoUsuario.getEncoded());
31
32              // Lê os certificados do disco para verificação
33              X509Certificate certificadoCarregadoCa = LeitorDeCertificados.
                    lerCertificadoDoDisco(Constantes.caminhoCertificadoAcRaiz);
34              X509Certificate certificadoCarregadoUsuario = LeitorDeCertificados.
                    lerCertificadoDoDisco(Constantes.caminhoCertificadoUsuario);
35
36              // Compara os certificados gerados com os lidos do disco
37              if (certificadoCA.equals(certificadoCarregadoCa) && certificadoUsuario.
                    equals(certificadoCarregadoUsuario)) {
38                  System.out.println("    Verificação de integridade dos certificados
                        concluída com sucesso.");
39                  System.out.println("Sucesso na etapa 3!");
40              } else {
41                  System.err.println("Falha na verificação: Os certificados escritos em
                        disco não são idênticos aos lidos.");
42              }
43
44          } catch (IOException | OperatorCreationException | CertificateException e) {
45              System.err.println("Erro ao executar a Terceira Etapa: " + e.getMessage());
46          }
47      }
48  }
```

A sessão final do código verifica se os dados salvos, caso forem carregados, estão em concordância aos dados iniciais que geraram os certificados originalmente.

Conteúdo dos certificados gerados, observados por meio do openssl nas figuras 3.4.2 e 3.4.3.

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 1 (0x1)
        Signature Algorithm: ecdsa-with-SHA256
        Issuer: CN = AC-RAIZ
        Validity
            Not Before: Jul 20 20:54:06 2025 GMT
            Not After : Jul 30 20:54:06 2025 GMT
        Subject: CN = AC-RAIZ
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (521 bit)
                pub:
                    04:00:32:cc:34:fd:0a:8d:04:1e:7e:3a:c5:e8:7c:
                    71:fc:c5:5c:50:54:15:12:b9:cf:5c:71:ac:b1:fd:
                    6e:7c:d1:2b:e4:d7:24:9b:5f:3b:57:c3:37:e4:fc:
                    33:e1:47:f6:96:cd:63:10:3d:ed:f3:9a:ae:f8:4b:
                    05:7d:f1:2b:03:ea:ed:00:77:ed:20:3a:c6:0d:9e:
                    db:4a:5e:b7:bd:06:0d:9e:b0:15:9e:63:4d:62:3f:
                    e7:68:bf:0c:86:4c:d0:58:69:68:45:a9:da:67:30:
                    e3:26:d5:cc:0a:5b:84:e0:40:4d:08:5f:75:a9:ca:
                    ff:90:9d:4e:5a:a9:7a:03:89:ac:7f:80:60
                ASN1 OID: secp521r1
                NIST CURVE: P-521
    Signature Algorithm: ecdsa-with-SHA256
    Signature Value:
        30:81:88:02:42:01:c9:1c:dd:97:76:cd:a5:09:c0:95:8a:3c:
        7c:d7:4a:dd:68:48:61:e8:84:cf:d5:a1:cc:20:c4:ed:77:d6:
        2e:10:19:68:95:7d:e9:44:12:ed:6f:15:2c:95:d6:14:98:21:
        15:55:49:2f:42:31:7c:66:2d:30:ea:f8:45:0a:d4:f1:62:02:
        42:00:91:18:57:5a:87:7c:74:ea:7d:f1:7c:8a:36:d1:31:92:
        80:cf:ee:f4:c5:a6:9b:0b:5d:38:01:5e:82:c2:ae:41:43:1d:
        b0:34:86:80:bb:e1:50:2e:f8:e7:62:49:3e:64:b8:6a:30:9f:
        9a:d6:6b:8e:e5:64:ea:f1:b4:1d:9d:fd:30
-----BEGIN CERTIFICATE-----
MIIBmDCB+qADAgECAgEBMAoGCCqGSM49BAMCMBIxEDAOBgNVBAMMB0FDLVJBSVow
HhcNMjUwNzIwMjA1NDA2WhcNMjUwNzMwMjA1NDA2WjASMRAwDgYDVQQDDAdBQy1S
QUlaMIGbMBAGByqGSM49AgEGBSuBBAAjA4GGAAQAMsw0/QqNBB5+OsXofHH8xVxQ
VBUSuc9ccayx/W580Svk1ySbXztXwzfk/DPhR/aWzWMQPe3zmq74SwV98SsD6u0A
d+0gOsYNnttKXre9Bg2esBWeY01iP+dovwyGTNBYaWhFqdpnMOMm1cwKW4TgQE0I
X3Wpyv+QnU5aqXoDiax/gGAwCgYIKoZIzj0EAwIDgYwAMIGIAkIByRzdl3bNpQnA
lYo8fNdK3WhIYeiEz9WhzCDE7XfWLhAZaJV96UQS7W8VLJXWFJghFVVJL0IxfGYt
MOr4RQrU8WICQgCRGFdah3x06n3xfIo20TGSgM/u9MWmmwtdOAFegsKuQUMdsDSG
gLvhUC7452JJPmS4ajCfmtZrjuVk6vG0HZ39MA==
-----END CERTIFICATE-----
```

Figura 3.4.2 – CertificadoAcRaiz

(Importante ressaltar que o conteúdo dos certificados enviados não é igual a esses, uma vez que eles foram sobrescritos ao rodar o código novamente para testar etapas subsequentes).

Referências

BOUNCY CASTLE. **ASN1EncodableVector (BC FIPS 2.1.0 Documentation)**. 2023. Disponível em: <https://downloads.bouncycastle.org/fips-java/docs/bc-fips-2.1.0/javadoc/org/bouncycastle/asn1/ASN1EncodableVector.html>. Acesso em: 19 jul. 2025.

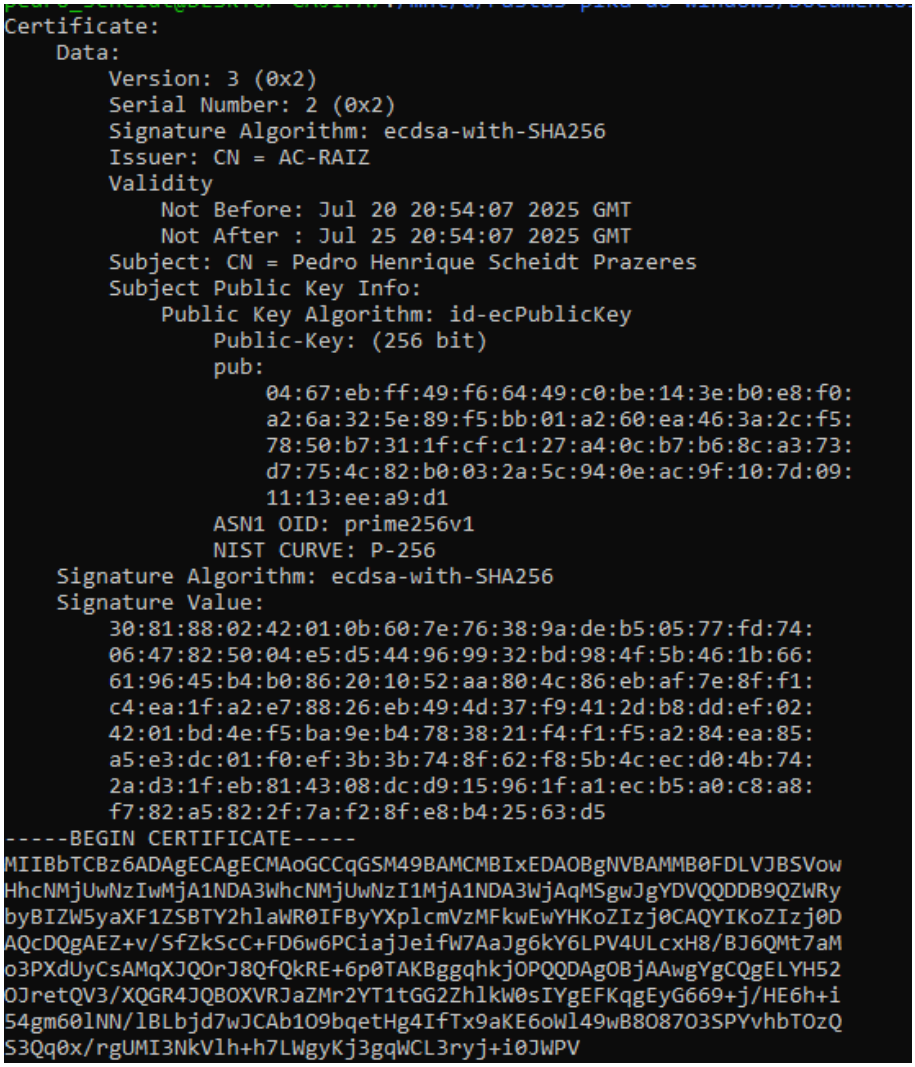

Figura 3.4.3 – CertificadoUsuario

BOUNCY CASTLE. **Deprecated List (bcprov-jdk14 1.74)**. 2023. Disponível em: <https://javadoc.io/doc/org.bouncycastle/bcprov-jdk14/1.74/deprecated-list.html>. Acesso em: 19 jul. 2025.

_____. **TBSCertificate (bcprov-jdk15to18 Documentation)**. 2023. Disponível em: <https://downloads.bouncycastle.org/java/docs/bcprov-jdk15to18/javadoc/index.html?org/bouncycastle/asn1/x509/TBSCertificate.html>. Acesso em: 19 jul. 2025.

MAYRHOFER, Nikolaus. **Creating X.509 Certificates in Java**. 2020. Disponível em: <https://www.mayrhofer.eu.org/post/create-x509-certs-in-java/>. Acesso em: 19 jul. 2025.

ORACLE. **Calendar (Java Platform SE 8)**. 2015. Disponível em: <https://docs.oracle.com/javase/8/docs/api/java/util/Calendar.html>. Acesso em: 19 jul. 2025.

ORACLE. **X509Certificate (Java Platform SE 8)**. 2015. Disponível em: <https://docs.oracle.com/javase/8/docs/api/java/security/cert/X509Certificate.html>. Acesso em: 19 jul. 2025.

STACK OVERFLOW. **Creating X509 Certificate in Java with TBSCertificate**. 2020. Disponível em: <https://stackoverflow.com/questions/60963379/creating-x509-certificate-in-java-with-tbscertificate>. Acesso em: 19 jul. 2025.

_____. **How to read information from SSL certificate file**. 2018. Disponível em: <https://stackoverflow.com/questions/50462304/how-to-read-information-from-ssl-certificate-file>. Acesso em: 19 jul. 2025.

_____. **Write X509 certificate into PEM formatted string in Java**. 2010. Disponível em: <https://stackoverflow.com/questions/3313020/write-x509-certificate-into-pem-formatted-string-in-java>. Acesso em: 19 jul. 2025.

4 GERAR REPOSITÓRIO DE CHAVES SEGURO

Iniciei esta etapa observando o template e estudando as documentações das classes e métodos que já estavam no arquivo.

4.1 GeradorDeRepositorios.java

Como em outras questões, criei um argumento optativo para uma informação dos arquivos de constantes. Neste caso, o algorítmo utilizado pode ser modificado na chamada da função, embora essa mudança não seja tão seamless quanto em outras, pois todo o resto do código é feito assumindo que o algoritmo utilizado é o PKCS#12, elegi não extender o tempo em uma parte do código optativa (e que provavelmente não será avaliada na correção do desafio), portanto, o argumento optativo não foi o foco nesta questão.

```
1  /**
2   * Classe responsável por gerar um repositório de chaves PKCS#12.
3   *
4   * @see KeyStore
5   */
6  public class GeradorDeRepositorios {
7
8      /**
9       * Gera um PKCS\#12 para a chave privada/certificado passados como parâmetro.
10      *
11      * @param chavePrivada  chave privada do titular do certificado.
12      * @param certificado   certificado do titular.
13      * @param caminhoPkcs12 caminho onde será escrito o PKCS\#12.
14      * @param alias         nome amigável dado à entrada do PKCS\#12, que
15      * comportará a chave e o certificado.
16      * @param senha         senha de acesso ao PKCS\#12.
17      * @throws KeyStoreException        em caso de erro com o tipo de repositório.
18      * @throws NoSuchProviderException  se o provedor "BC" não for encontrado.
19      * @throws IOException              em caso de erro de I/O ao carregar ou salvar o
            repositório.
20      * @throws NoSuchAlgorithmException se o algoritmo para verificação de integridade
            do repositório não for encontrado.
21      * @throws CertificateException     em caso de erro com o certificado.
22      */
23     public static void gerarPkcs12(PrivateKey chavePrivada, X509Certificate certificado
            ,
24                                    String caminhoPkcs12, String alias, char[] senha)
                                          throws KeyStoreException,
                                          NoSuchProviderException, IOException,
                                          NoSuchAlgorithmException, CertificateException {
25         gerarPkcs12(chavePrivada, certificado, caminhoPkcs12, alias, senha, Constantes.
               formatoRepositorio);
26     }
```

```
27
28      /**
29       * Gera um PKCS#12 para a chave privada/certificado passados como parâmetro.
30       *
31       * @param chavePrivada  chave privada do titular do certificado.
32       * @param certificado   certificado do titular.
33       * @param caminhoPkcs12 caminho onde será escrito o PKCS#12.
34       * @param alias         nome amigável dado à entrada do PKCS#12, que
35       * comportará a chave e o certificado.
36       * @param senha         senha de acesso ao PKCS#12.
37       * @param algoritmo     algoritmo utilizado (padrão PKCS#12)
38       * @throws KeyStoreException        em caso de erro com o tipo de repositório.
39       * @throws NoSuchProviderException  se o provedor "BC" não for encontrado.
40       * @throws IOException              em caso de erro de I/O ao carregar ou salvar o
             repositório.
41       * @throws NoSuchAlgorithmException se o algoritmo para verificação de integridade
             do repositório não for encontrado.
42       * @throws CertificateException     em caso de erro com o certificado.
43       */
44      public static void gerarPkcs12(PrivateKey chavePrivada, X509Certificate certificado
             ,
45                                     String caminhoPkcs12, String alias, char[] senha,
                                           String algoritmo) throws KeyStoreException,
                                           NoSuchProviderException, IOException,
                                           NoSuchAlgorithmException, CertificateException {
46          KeyStore pkcs12KeyStore;
47          try {
48              pkcs12KeyStore = KeyStore.getInstance(algoritmo, "BC");
49              pkcs12KeyStore.load(null, null);
50          } catch (KeyStoreException e) {
51              throw new KeyStoreException("Erro: O formato de repositório '" + algoritmo
                    + "' não é suportado.", e);
52          } catch (NoSuchProviderException e) {
53              throw new NoSuchProviderException("O provedor de segurança 'BC' não foi
                    encontrado.");
54          } catch (NoSuchAlgorithmException | CertificateException | IOException e) {
55              throw new IOException("Erro ao inicializar um repositório do tipo '" +
                    algoritmo + "' vazio.", e);
56          }
57
58          try {
59              // Coloca a chave no keystore
60              pkcs12KeyStore.setKeyEntry(alias, chavePrivada, senha, new java.security.
                    cert.Certificate[]{certificado});
61          } catch (KeyStoreException e) {
62              throw new KeyStoreException("Erro ao inserir a chave e o certificado no
                    repositório com o alias '" + alias + "'.", e);
63          }
64
65          // Grava em disco
66          try (FileOutputStream fileOutputStream = new FileOutputStream(caminhoPkcs12)) {
67              pkcs12KeyStore.store(fileOutputStream, senha);
68              System.out.println("    Repositório " + algoritmo + " escrito em disco com
                    sucesso.");
69          } catch (IOException e) {
70              throw new IOException("Eroo ao escrever o arquivo de repositório em: " +
                    caminhoPkcs12, e);
71          } catch (KeyStoreException | NoSuchAlgorithmException | CertificateException e)
```

```
            {
72              throw new KeyStoreException("Erro ao finalizar e salvar o repositório no
                    arquivo: " + caminhoPkcs12, e);
73          }
74      }
75
76  }
```

4.2 RepositorioChaves.java

Assim como na etapa 2 sessão 2.4, houve um erro nesta parte quanto ao provedor e necessitei definir o bouncycastle como provedor para meu código passar em meus testes de verificação. Esta mudança foi feita e o código obteve sucesso.

Este arquivo é uma biblioteca compreensível de acesso a PKCS#12. O formato sugerigo no template parece ser pegando a chave dentro da função diretamente do arquivo de constantes. Embora isso torne mais simples de utilizar a biblioteca, isso não permite escalabilidade como biblioteca, portanto, mantendo em concordância com outras mudanças que fiz ao longo do trabalho, alterei a estrutura para receber a chave como argumento. Ademais, embora a senha seja única para os diferentes PKCS#12 criados e poderia ser apenas pega do arquivo de constantes, em um ambiente real, cada repositorio teria sua senha de acesso diferente, e por esse motivo, a senha é utilizada como parâmetro no momento de chamar a função “abrir()”:

```
1  public void abrir(String caminhoRepositorio, char[] senha) throws KeyStoreException,
       IOException, NoSuchAlgorithmException, CertificateException {
2          this.senha = senha;
3
4          try (FileInputStream fileInputStream = new FileInputStream(caminhoRepositorio))
               {
5              // Carrega o keystore do arquivo usando a senha fornecida.
6              this.repositorio.load(fileInputStream, this.senha);
7          } catch (FileNotFoundException e) {
8              throw new IOException("Erro com o caminho: " + caminhoRepositorio, e);
9          } catch (IOException e) {
10             throw new IOException("Erro ao carregar o repositório com chave em
                   Repositorio Chaves", e);
11         }
12
13         Enumeration<String> aliases = this.repositorio.aliases();
14         if (aliases.hasMoreElements()) {
15             this.alias = aliases.nextElement();
16         } else {
17             throw new KeyStoreException("Erro: O repositório em '" + caminhoRepositorio
                   + "' está vazio.");
18         }
19     }
```

4.3 QuartaEtapa.java

O código em si é simples, somente instanciado e salvando os repositorios por meio das funções de util. Contudo, há uma longa sessão de debbuging, que carrega os repositórios do disco após eles serem salvos e compara o carregado com os dados que o geraram (semelhante com aquilo que foi feito na etapa 3):

```
1  public class QuartaEtapa {
2
3      public static void executarEtapa() {
4          System.out.println("\nInício Etapa 4");
5          try {
6              // Carrega as chaves e certificados do disco
7              PrivateKey privadaAc = LeitorDeChaves.lerChavePrivadaDoDisco(Constantes.
                   caminhoChavePrivadaAc, Constantes.algoritmoChave);
8              X509Certificate certificadoAc = LeitorDeCertificados.lerCertificadoDoDisco(
                   Constantes.caminhoCertificadoAcRaiz);
9              PrivateKey privadaUsuario = LeitorDeChaves.lerChavePrivadaDoDisco(
                   Constantes.caminhoChavePrivadaUsuario, Constantes.algoritmoChave);
10             X509Certificate certificadoUsuario = LeitorDeCertificados.
                   lerCertificadoDoDisco(Constantes.caminhoCertificadoUsuario);
11
12             // Gera e salva os repositórios PKCS#12
13             GeradorDeRepositorios.gerarPkcs12(privadaAc, certificadoAc, Constantes.
                   caminhoPkcs12AcRaiz, Constantes.aliasAc, Constantes.senhaMestre);
14             GeradorDeRepositorios.gerarPkcs12(privadaUsuario, certificadoUsuario,
                   Constantes.caminhoPkcs12Usuario, Constantes.aliasUsuario, Constantes.
                   senhaMestre);
15
16             // Sessão de verificação
17             boolean acIguais = false;
18             boolean usuarioIguais = false;
19
20             // Abre e verifica o repositório da AC-Raiz
21             RepositorioChaves repositorioAc = new RepositorioChaves();
22             repositorioAc.abrir(Constantes.caminhoPkcs12AcRaiz, Constantes.senhaMestre)
                   ;
23             if (privadaAc.equals(repositorioAc.pegarChavePrivada()) &&
24                     certificadoAc.equals(repositorioAc.pegarCertificado())) {
25                 acIguais = true;
26             } else {
27                 System.err.println("Falha na verificação do PKCS#12 da AC-Raiz.");
28             }
29
30             // Abre e verifica o repositório do Usuário
31             RepositorioChaves repositorioUsuario = new RepositorioChaves();
32             repositorioUsuario.abrir(Constantes.caminhoPkcs12Usuario, Constantes.
                   senhaMestre);
33             if (privadaUsuario.equals(repositorioUsuario.pegarChavePrivada()) &&
34                     certificadoUsuario.equals(repositorioUsuario.pegarCertificado())) {
35                 usuarioIguais = true;
36             } else {
37                 System.err.println("Falha na verificação do PKCS#12 do Usuário.");
38             }
39
40             if (acIguais && usuarioIguais) {
```

```
41                  System.out.println("Sucesso na etapa 4!");
42              }
43
44          } catch (KeyStoreException | CertificateException | IOException |
                NoSuchAlgorithmException | NoSuchProviderException |
                UnrecoverableKeyException e) {
45              System.err.println("Erro ao executar a Quarta Etapa: " + e.getMessage());
46          }
47      }
48  }
```

Referências

ORACLE. **Administering Key and Certificate Keystores**. 2005. Disponível em: <https://docs.oracle.com/cd/E19509-01/820-3503/ggfhb/index.html>. Acesso em: 20 jul. 2025.

_____. **KeyStore (Java Platform SE 8)**. 2015. Disponível em: <https://docs.oracle.com/javase/8/docs/api/java/security/KeyStore.html>. Acesso em: 20 jul. 2025.

STACK OVERFLOW. **How to create a certificate into a PKCS12 keystore with keytool**. 2013. Disponível em: <https://stackoverflow.com/questions/14375185/how-to-create-a-certificate-into-a-pkcs12-keystore-with-keytool>. Acesso em: 20 jul. 2025.

_____. **How to create a PKCS12 keystore**. 2020. Disponível em: <https://stackoverflow.com/questions/64442156/how-to-create-a-pkcs12-keystore>. Acesso em: 20 jul. 2025.

_____. **What does the certificate chain in the keystore setKeyEntry mean and how to**. 2019. Disponível em: <https://stackoverflow.com/questions/56523093/what-does-the-certificate-chain-in-the-keystore-setkeyentry-mean-and-how-to>. Acesso em: 20 jul. 2025.

5 GERAR UMA ASSINATURA DIGITAL

Como de praxe, observei os tipos utilizados no template e li a documentação dos tipos. Escolhi iniciar pelas funções privadas, uma vez que elas seriam chamadas antes das públicas. Diferentemente das outras etapas até o momento, esta etapa será dividida por funções, ao invés de por arquivo (dado que trata com apenas um arquivo e com várias funções complexas).

No início desta etapa estava em dúvida quanto a forma que deveria ser feita. Iniciei pensando que a função deveria reutilizar o resultado da primeira etapa 1, herdando o hash já criado para suimular um Hardware Security Module, por isso entrei em contato com a equipe do Labsec para verificar o que deveria ser feito, e fui respondido:

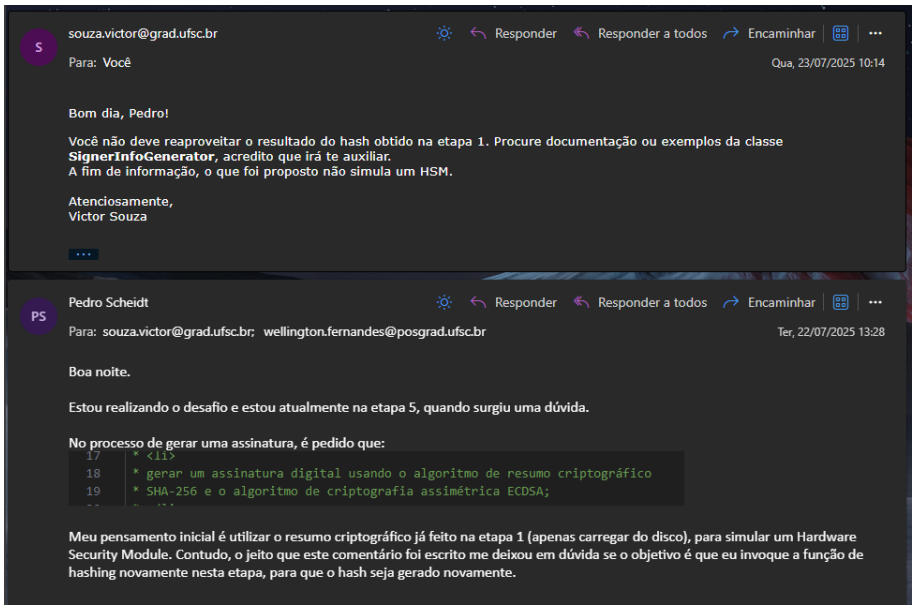

Figura 5.0.1 – Questão à equipe do Labsec

Portanto, não foi reutilizado o resultado da etapa 1.

O método que utilizei para essa etapa foi mais longo, nas duas funções mais complexas da etapa (preparaInformacoesAssinante 5.3 e assinar 5.4), observei o tipo da saída, fui na documentação, e verifiquei qual o tipo do argumento necessário para instanciar a classe, e repetindo este processo com o tipo deste argumento “recursivamente”.

Importante destacar que todo o código foi feito para apenas um assinante, não sendo possível múltiplas chaves privadas assinarem o mesmo documento.

5.1 escreveAssinatura

Uma função simples, que apenas salva o resultado do código em disco:

```java
/**
 * Escreve a assinatura no local apontado.
 *
 * @param arquivo    arquivo que será escrita a assinatura.
 * @param assinatura objeto da assinatura.
 * @throws IOException caso ocorra um erro ao codificar ou escrever a assinatura no
     arquivo.
 */
public void escreveAssinatura(OutputStream arquivo, CMSSignedData assinatura) throws
    IOException {
    try {
        byte[] bytesAssinatura = assinatura.getEncoded();
        arquivo.write(bytesAssinatura);
        System.out.println("    Sucesso em salvamento em disco!");
    } catch (IOException e) {
        throw new IOException("Erro ao escrever a assinatura no arquivo.", e);
    }
}
```

5.2 preparaDadosParaAssinar

Outra função simples, que converte o documento no formato requerido:

```java
/**
* Transforma o documento que será assinado para um formato compatível
* com a assinatura.
*
* @param caminhoDocumento caminho do documento que será assinado.
* @return Documento no formato correto.
*/
private CMSTypedData preparaDadosParaAssinar(String caminhoDocumento) {
File arquivo = new File(caminhoDocumento);
return new CMSProcessableFile(arquivo);
}
```

5.3 preparaInformacoesAssinante

Esta função é bem complexa, com conversão entre vários tipos diferentes, os quais não são auto-explicativos:

```java
/**
 * Gera as informações do assinante na estrutura necessária para ser
 * adicionada na assinatura.
 *
 * @param chavePrivada chave privada do assinante.
 * @param certificado  certificado do assinante.
 * @return Estrutura com informações do assinante.
 * @throws CertificateEncodingException se houver erro na codificação do certificado.
 * @throws OperatorCreationException    se houver erro na criação do assinador ou do
     provedor de digest.
 */
```

```
11 private SignerInfoGenerator preparaInformacoesAssinante(PrivateKey chavePrivada,
12                                                         Certificate certificado) throws
                                                               CertificateEncodingException
                                                               , OperatorCreationException
                                                                {
13     try {
14         // Nesta linha é definido o algorítmo  de hashing + assinatura, coletados do
               arquivo de constantes.
15         JcaContentSignerBuilder contentSignerBuilder = new JcaContentSignerBuilder(
               Constantes.algoritmoAssinatura);
16
17         // Junta o objeto anterior com a chave privada, formando a primeira metade do
               output final.
18         ContentSigner contentSigner = contentSignerBuilder.build(chavePrivada);
19
20         // Instancia e configura a estrura que calcula o hash.
21         JcaDigestCalculatorProviderBuilder providerBuilder = new
               JcaDigestCalculatorProviderBuilder();
22         DigestCalculatorProvider digestCalculatorProvider = providerBuilder.build();
23
24         // Inicia o construtor principal com a função que calcula o hash.
25         JcaSignerInfoGeneratorBuilder infoGeneratorBuilder = new
               JcaSignerInfoGeneratorBuilder(digestCalculatorProvider);
26
27         // Junta o construtor com o contentSigner (que representa qual algorítmo de
               hash + assinatura é utilizado, junto da chave privada)
28         // e o certificado, e retorna a build pronta.
29         SignerInfoGenerator siginfo = infoGeneratorBuilder.build(contentSigner, (
               X509Certificate) certificado);
30         System.out.println("    Sucesso em gerar informações do signatário");
31         return siginfo;
32     } catch (OperatorCreationException e) {
33         throw new OperatorCreationException("Erro: Falha ao construir as informações do
                assinante. Verifique o algoritmo de assinatura ('" + Constantes.
               algoritmoAssinatura + "') e a chave privada.", e);
34     } catch (CertificateEncodingException e) {
35         throw new CertificateEncodingException("Erro: Falha ao codificar o certificado
               para gerar as informações do assinante.", e);
36     }
37 }
```

Portanto, uma explicação passo a passo:

Nesta linha é definido o algorítmo de hashing + assinatura, coletados do arquivo de constantes:

```
1 JcaContentSignerBuilder contentSignerBuilder = new JcaContentSignerBuilder(Constantes.
      algoritmoAssinatura);
```

Junta o objeto anterior com a chave privada, formando a primeira metade do output final:

```
1 ContentSigner contentSigner = contentSignerBuilder.build(chavePrivada);
```

Instancia e configura a estrura que calcula o hash:

```
1 JcaDigestCalculatorProviderBuilder providerBuilder = new
      JcaDigestCalculatorProviderBuilder();
2 DigestCalculatorProvider digestCalculatorProvider = providerBuilder.build();
```

Inicia o construtor principal com a função que calcula o hash:

```
JcaSignerInfoGeneratorBuilder infoGeneratorBuilder = new JcaSignerInfoGeneratorBuilder(
    digestCalculatorProvider);
```

Junta o construtor com o contentSigner (que representa qual algorítmo de hash + assinatura é utilizado, junto da chave privada) e o certificado, e retorna a build pronta:

```
SignerInfoGenerator siginfo = infoGeneratorBuilder.build(contentSigner, (
    X509Certificate) certificado);
System.out.println("    Sucesso em gerar informações do signatário");
return siginfo;
```

5.4 assinar

Este método basicamente compila todas as informações do signatário, e realiza o passo final de realmente assinar:

```
/**
 * Gera uma assinatura no padrão CMS.
 *
 * @param caminhoDocumento caminho do documento que será assinado.
 * @return Documento assinado.
 * @throws CertificateEncodingException se houver erro na codificação do certificado.
 * @throws OperatorCreationException    se houver erro na criação do assinador.
 * @throws CMSException                 se houver erro na geração da assinatura CMS.
 */
public CMSSignedData assinar(String caminhoDocumento) throws
    CertificateEncodingException, OperatorCreationException, CMSException {
    CMSTypedData dadosParaAssinar = this.preparaDadosParaAssinar(caminhoDocumento);
    SignerInfoGenerator informacoesAssinante;
    Store<X509CertificateHolder> armazemCertificados;

    try {
        // Prepara informações do assinante (certificado, chave, algoritmo de hash +
            assinatura)
        informacoesAssinante = this.preparaInformacoesAssinante(this.chavePrivada, this
            .certificado);
        this.geradorAssinaturaCms.addSignerInfoGenerator(informacoesAssinante);

        // Adiciona o certificado do signatário
        List<Certificate> listaCertificados = new ArrayList<>();
        listaCertificados.add(this.certificado);
        armazemCertificados = new JcaCertStore(listaCertificados);
        this.geradorAssinaturaCms.addCertificates(armazemCertificados);
    } catch (CertificateEncodingException e) {
        throw new CertificateEncodingException("Erro ao adicionar certificado em
            GeradorDeAssinatura: ", e);
    } catch (OperatorCreationException e) {
        // Mensagem já criada em prepara informacoes
        throw e;
    }

    try {
        return this.geradorAssinaturaCms.generate(dadosParaAssinar, true);
```

```
34      } catch (CMSException e) {
35          throw new CMSException("Erro ao gerar assinatura em GeradorDeAssinatura: ", e);
36      }
37  }
```

Importante destacar o argumento “true” na função de geração, este argumento serve para anexar o documento na assinatura, conforme requisitado na QuintaEtapa.java:

```
1  * <li>
2  * assinar o documento {@code textoPlano.txt}, onde a assinatura deverá ser do
3  * tipo "anexada", ou seja, o documento estará embutido no arquivo de
4  * assinatura;
5  * </li>
```

5.5 QuintaEtapa.java

Como todos os “mains” das etapas, esse arquivo apenas invoca as funções criadas, no caso desta etapa, é instanciada o GeradorDeAssinatura, informado o certificado e a chave privada, e enviado o caminho do arquivo. Decidi retirar as informações da chave privada/certificado do repositório PKCS#12 (resultado da etapa 4) ao invés de carregar ambas diretamente do disco (resultado das etapas 2 e 3) para simular que o código está sendo feita na máquina signatária, que teria estas informações em um repositório seguro, para envio a outra máquina. Diferentemente da etapa 6, que carrega o certificado direto do disco (explicação da motivação desta em SextaEtapa 6)

```
1  public class QuintaEtapa {
2
3      public static void executarEtapa() {
4          System.out.println("\nInicio Etapa 5");
5          try {
6              RepositorioChaves repo = new RepositorioChaves();
7              repo.abrir(Constantes.caminhoPkcs12Usuario, Constantes.senhaMestre);
8              X509Certificate certificado = repo.pegarCertificado();
9              PrivateKey chavePrivada = repo.pegarChavePrivada();
10
11             GeradorDeAssinatura assinador = new GeradorDeAssinatura();
12             assinador.informaAssinante(certificado, chavePrivada);
13
14             CMSSignedData assinado = assinador.assinar(Constantes.caminhoTextoPlano);
15
16             try (FileOutputStream fileOutput = new FileOutputStream(Constantes.
                   caminhoAssinatura)) {
17                 assinador.escreveAssinatura(fileOutput, assinado);
18                 System.out.println("Final da etapa 5! (Sucesso é verificado na etapa 6)
                       ");
19             }
20         } catch (KeyStoreException | IOException | NoSuchAlgorithmException |
               CertificateException | NoSuchProviderException | UnrecoverableKeyException
               | OperatorCreationException | CMSException e) {
21             System.err.println("Erro ao executar a Quinta Etapa: " + e.getMessage());
22         }
```

```
23      }
24  }
```

Utilizando o ans1dump, é possível verificar o resultado desta operação, o resultado completo está no B. Contudo, as partes centrais são o conteúdo do texto plano “Pedro Henrique Scheidt Prazeres – 22100919", que foi assinado corretamente:

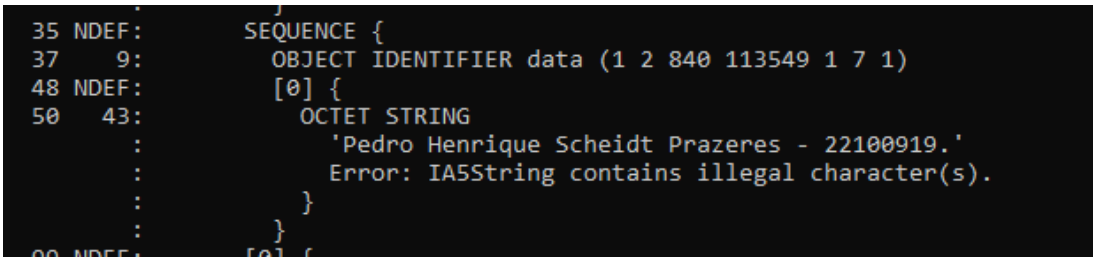

Figura 5.5.1 – Verificação do ans1dump

Também é possível verificar que o certificado utilizado foi o do usuário:

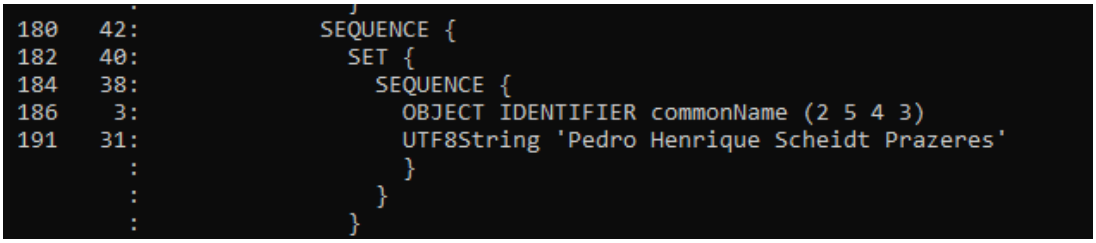

Figura 5.5.2 – Verificação do certificado do usuário

Referências

ANONRIG. **ECDSA Java Example with Bouncy Castle**. Exemplo auxiliar — não diretamente sobre CMS. 2023. Disponível em: <https://github.com/anonrig/bouncycastle-implementations/blob/master/ecdsa.java>. Acesso em: 21 jul. 2025.

BOUNCY CASTLE. **CMSProcessableFile**. 2023. Disponível em: <https://downloads.bouncycastle.org/java/docs/bcpkix-jdk14/javadoc/org/bouncycastle/cms/CMSProcessableFile.html>. Acesso em: 21 jul. 2025.

_____. **CMSSignedData**. 2023. Disponível em: <https://javadoc.io/doc/org.bouncycastle/bcmail-jdk15+/latest/org/bouncycastle/cms/CMSSignedData.html>. Acesso em: 21 jul. 2025.

BOUNCY CASTLE. **CMSSignedDataGenerator**. 2023. Disponível em: <https://javadoc.io/doc/org.bouncycastle/bcmail-jdk15/latest/org/bouncycastle/cms/CMSSignedDataGenerator.html>. Acesso em: 21 jul. 2025.

_____. **CMSTypedData**. 2023. Disponível em: <https://downloads.bouncycastle.org/java/docs/bcpkix-jdk18on/javadoc/org/bouncycastle/cms/CMSTypedData.html>. Acesso em: 21 jul. 2025.

_____. **ContentInfo**. 2023. Disponível em: <https://javadoc.io/doc/org.bouncycastle/bcprov-jdk15on/1.64/org/bouncycastle/asn1/cms/ContentInfo.html>. Acesso em: 21 jul. 2025.

_____. **ContentSigner**. 2023. Disponível em: <https://downloads.bouncycastle.org/java/docs/bcpkix-jdk18on/javadoc/org/bouncycastle/operator/ContentSigner.html>. Acesso em: 21 jul. 2025.

_____. **ContentSigner (repetida para contexto diferente)**. 2023. Disponível em: <https://downloads.bouncycastle.org/java/docs/bcpkix-jdk18on/javadoc/org/bouncycastle/operator/ContentSigner.html>. Acesso em: 21 jul. 2025.

_____. **DigestCalculatorProvider**. 2023. Disponível em: <https://downloads.bouncycastle.org/java/docs/bcpkix-jdk18on/javadoc/org/bouncycastle/operator/DigestCalculatorProvider.html>. Acesso em: 21 jul. 2025.

_____. **JcaContentSignerBuilder**. 2023. Disponível em: <https://downloads.bouncycastle.org/java/docs/bcpkix-jdk18on/javadoc/org/bouncycastle/operator/jcajce/JcaContentSignerBuilder.html>. Acesso em: 21 jul. 2025.

_____. **JcaDigestCalculatorProviderBuilder (FIPS)**. 2023. Disponível em: <https://downloads.bouncycastle.org/fips-java/docs/bcpkix-fips-1.0.7/javadoc/org/bouncycastle/operator/jcajce/JcaDigestCalculatorProviderBuilder.html>. Acesso em: 21 jul. 2025.

_____. **JcaDigestCalculatorProviderBuilder (JDK18on)**. 2023. Disponível em: <https://downloads.bouncycastle.org/java/docs/bcpkix-jdk18on/javadoc/org/bouncycastle/operator/jcajce/JcaDigestCalculatorProviderBuilder.html>. Acesso em: 21 jul. 2025.

BOUNCY CASTLE. **JcaSignerInfoGeneratorBuilder**. 2023. Disponível em: <https://javadoc.io/doc/org.bouncycastle/bcmail-jdk15on/1.46/org/bouncycastle/cms/jcajce/JcaSignerInfoGeneratorBuilder.html>. Acesso em: 21 jul. 2025.

_____. **SignerInfoGenerator**. 2023. Disponível em: <https://downloads.bouncycastle.org/java/docs/bcpkix-jdk18on/javadoc/org/bouncycastle/cms/SignerInfoGenerator.html>. Acesso em: 21 jul. 2025.

ENTRUST. **What Are Hardware Security Modules?** 2023. Disponível em: <https://www.entrust.com/resources/learn/what-are-hardware-security-modules>. Acesso em: 21 jul. 2025.

GOOGLE GEMINI. **Compartilhamento de sessão Gemini**. 2024. Disponível em: <https://g.co/gemini/share/4526add4c7a9>. Acesso em: 21 jul. 2025.

ORACLE. **API Reference (Oracle Fusion Middleware)**. 2011. Disponível em: <https://docs.oracle.com/cd/E21764_01/apirefs.1111/e10667/toc.htm>. Acesso em: 21 jul. 2025.

WIKIPEDIA. **Cryptographic Message Syntax**. 2024. Disponível em: <https://en.wikipedia.org/wiki/Cryptographic_Message_Syntax>. Acesso em: 21 jul. 2025.

6 VERIFICAR UMA ASSINATURA DIGITAL

Assim como na etapa 5, esta etapa consiste em somente um arquivo com funções complexas, e, portanto, este documento será organizado com uma sessão por função do arquivo e não como uma sessão para o arquivo todo. Assim como nas outras etapas, pesquisei os métodos de output das funções:

- SignerInformationVerifier do geraVerificadorInformacoesAssinatura
  - Tendo em vista a ultima etapa, também verifiquei o JcaSimpleSignerInfoVerifierBuilder
- SignerInformation do pegaInformacoesAssinatura

Assim como na quinta etapa, este código funciona somente com documentos assinados por uma única pessoa, não verificando multiplas assinaturas.

Como em todos os outros códigos, eu adiciono a BouncyCastle como provedor. Isso se faz para manter a uniformidade em todas as partes do código, isso é feito no início da classe VerificadorDeAssinatura:

```
1 public class VerificadorDeAssinatura {
2
3     static {
4         Security.addProvider(new BouncyCastleProvider());
5     }
6 ...
```

6.1 verificarAssinatura

A função que compila as outras funções do *utils* desta etapa, utiliza o método *verify* do SignerInformation para verificar com as informações do assinante:

```
1  /**
2   * Verifica a integridade de uma assinatura digital no padrão CMS.
3   *
4   * @param certificado certificado do assinante.
5   * @param assinatura  documento assinado.
6   * @return {@code true} se a assinatura for íntegra, e {@code false} do
7   * contrário.
8   * @throws OperatorCreationException se houver erro ao criar o verificador.
9   * @throws CMSException              se houver erro ao processar os dados da assinatura
        CMS.
10  */
11 public boolean verificarAssinatura(X509Certificate certificado,
12                                     CMSSignedData assinatura) throws
                                        OperatorCreationException, CMSException {
```

```
13      SignerInformationVerifier verificador = geraVerificadorInformacoesAssinatura(
            certificado);
14      SignerInformation sigInfo = pegaInformacoesAssinatura(assinatura);
15
16      try {
17          boolean resultado = sigInfo.verify(verificador);
18          System.out.println("    Sucesso na verificação");
19          return resultado;
20      } catch (CMSException e) {
21          throw new CMSException("Erro ao verificar assinatura: ", e);
22      }
23  }
```

6.2 geraVerificadorInformacoesAssinatura

Uma função relativamente simples que instancia o verificador de assinaturas utilizando o certificado daquele que originalmente assinou o documento:

```
1   /**
2    * Gera o verificador de assinaturas a partir das informações do assinante.
3    *
4    * @param certificado certificado do assinante.
5    * @return Objeto que representa o verificador de assinaturas.
6    * @throws OperatorCreationException se houver erro ao criar o verificador.
7    */
8   private SignerInformationVerifier geraVerificadorInformacoesAssinatura(X509Certificate
        certificado) throws OperatorCreationException {
9       try {
10          JcaSimpleSignerInfoVerifierBuilder builder = new
                JcaSimpleSignerInfoVerifierBuilder();
11          SignerInformationVerifier verificador = builder.build(certificado);
12          System.out.println("    Sucesso em gerar verificador");
13          return verificador;
14      } catch (OperatorCreationException e) {
15          throw new OperatorCreationException("Erro ao buildar o verificador de
                assinaturam em VerificadorDeAssinatura: Verifique se o certificado é válido
                 e seu algoritmo de chave pública é suportado.", e);
16      }
17  }
```

6.3 pegaInformacoesAssinatura

Assim como na Etapa 5, o código é feito apenas para uma assinatura, não sendo capaz de ligar com assinaturas de múltiplas pessoas. Por isso é pego apenas a primeira assinatura nesta função.

```
1   /**
2   * Classe responsável por pegar as informações da assinatura dentro do CMS.
3   *
4   * @param assinatura documento assinado.
5   * @return Informações da assinatura.
```

```
6  * @throws CMSException se a assinatura não contiver informações de assinante.
7  */
8  private SignerInformation pegaInformacoesAssinatura(CMSSignedData assinatura) throws
       CMSException {
9  SignerInformationStore sigInfoStore = assinatura.getSignerInfos();
10 Collection<SignerInformation> assinadores = sigInfoStore.getSigners();
11 SignerInformation sigInfo = assinadores.iterator().next();
12 System.out.println("    Sucesso em coletar informações do assinante");
13 return sigInfo;
14 }
```

6.4 SextaEtapa.java

Por fim, esta etapa pega o certificado de disco, conforme dito na etapa 5, para simular o recebimento de um certificado no processo de handshake, uma vez que é esta etapa que verifica a assinatura, a etapa 6 faz o papel de uma máquina recebendo um documento assinado e a chave pública. Por verificar se a assinatura criada na parte 5 está válida, esta etapa valida também o resultado da etapa anterior.

```
1  public class SextaEtapa {
2
3      static {
4          Security.addProvider(new BouncyCastleProvider());
5      }
6
7      public static void executarEtapa() {
8          System.out.println("\nInicio etapa 6");
9          try {
10             X509Certificate certificado = LeitorDeCertificados.lerCertificadoDoDisco(
                   Constantes.caminhoCertificadoUsuario);
11             byte[] assinaturaBytes = Files.readAllBytes(Paths.get(Constantes.
                   caminhoAssinatura));
12
13             CMSSignedData assinatura = new CMSSignedData(assinaturaBytes);
14
15             VerificadorDeAssinatura verificador = new VerificadorDeAssinatura();
16
17             boolean igual = verificador.verificarAssinatura(certificado, assinatura);
18             if (igual) {
19                 System.out.println("O certificado utilizado para gerar a assinatura da
                       etapa 5 é o do usuário");
20                 System.out.println("Sucesso na etapa 5!");
21                 System.out.println("Sucesso na Etapa 6!");
22             } else {
23                 System.err.println("Verificação de assinatura falhou. A assinatura pode
                       ser inválida ou o certificado incorreto.");
24             }
25
26         } catch (IOException | CertificateException | CMSException |
               OperatorCreationException e) {
27             System.err.println("Ocorreu um erro na sexta etapa: " + e.getMessage());
28         }
29     }
```

```
30
31 }
```

Referências

BOUNCY CASTLE. **JcaSimpleSignerInfoVerifierBuilder**. 2023. Disponível em: <https://downloads.bouncycastle.org/java/docs/bcpkix-jdk14/javadoc/org/bouncycastle/cms/jcajce/JcaSimpleSignerInfoVerifierBuilder.html>. Acesso em: 22 jul. 2025.

_____. **SignerInformation (bcmail-jdk15+)**. 2023. Disponível em: <https://javadoc.io/doc/org.bouncycastle/bcmail-jdk15+/latest/org/bouncycastle/cms/SignerInformation.html>. Acesso em: 22 jul. 2025.

_____. **SignerInformation (bcmail-jdk16)**. 2023. Disponível em: <https://javadoc.io/doc/org.bouncycastle/bcmail-jdk16/latest/org/bouncycastle/cms/SignerInformation.html>. Acesso em: 22 jul. 2025.

_____. **SignerInformationVerifier**. 2023. Disponível em: <https://downloads.bouncycastle.org/java/docs/bcpkix-jdk13/javadoc/org/bouncycastle/cms/SignerInformationVerifier.html>. Acesso em: 22 jul. 2025.

YOUTUBE. **CMS - SignedData Explained**. 2023. Disponível em: <https://www.youtube.com/watch?v=502iGDxuiRk>. Acesso em: 22 jul. 2025.

ANEXO A – CÓDIGO DE ERRO

```
java.security.spec.InvalidKeySpecException: java.security.
   InvalidKeyException: Unable to decode key
        at java.base/sun.security.ec.ECKeyFactory.engineGeneratePrivate(
           ECKeyFactory.java:168)
        at java.base/java.security.KeyFactory.generatePrivate(KeyFactory
           .java:388)
        at br.ufsc.labsec.pbad.hiring.criptografia.chave.LeitorDeChaves.
           lerChavePrivadaDoDisco(LeitorDeChaves.java:55)
        at br.ufsc.labsec.pbad.hiring.etapas.SegundaEtapa.executarEtapa(
           SegundaEtapa.java:45)
        at br.ufsc.labsec.pbad.hiring.ExecutarEtapas.main(ExecutarEtapas
           .java:14)
        at org.codehaus.mojo.exec.ExecJavaMojo$1.run(ExecJavaMojo.java
           :254)
        at java.base/java.lang.Thread.run(Thread.java:1575)
Caused by: java.security.InvalidKeyException: Unable to decode key
        at java.base/sun.security.pkcs.PKCS8Key.decode(PKCS8Key.java
           :137)
        at java.base/sun.security.pkcs.PKCS8Key.<init>(PKCS8Key.java:96)
        at java.base/sun.security.ec.ECPrivateKeyImpl.<init>(
           ECPrivateKeyImpl.java:81)
        at java.base/sun.security.ec.ECKeyFactory.implGeneratePrivate(
           ECKeyFactory.java:243)
        at java.base/sun.security.ec.ECKeyFactory.engineGeneratePrivate(
           ECKeyFactory.java:164)
        ... 6 more
Caused by: java.io.IOException: algid parse error, not a sequence
        at java.base/sun.security.x509.AlgorithmId.parse(AlgorithmId.
           java:389)
        at java.base/sun.security.pkcs.PKCS8Key.decode(PKCS8Key.java
           :112)
        ... 10 more
```

ANEXO B – PRINTS DO ASN1DUMP

```
  0 NDEF: SEQUENCE {
  2    9:   OBJECT IDENTIFIER signedData (1 2 840 113549 1 7 2)
 13 NDEF:   [0] {
 15 NDEF:     SEQUENCE {
 17    1:       INTEGER 1
 20   13:       SET {
 22   11:         SEQUENCE {
 24    9:           OBJECT IDENTIFIER sha-256 (2 16 840 1 101 3 4 2 1)
       :           }
       :         }
 35 NDEF:       SEQUENCE {
 37    9:         OBJECT IDENTIFIER data (1 2 840 113549 1 7 1)
 48 NDEF:         [0] {
 50   43:           OCTET STRING
       :             'Pedro Henrique Scheidt Prazeres - 22100919.'
       :             Error: IA5String contains illegal character(s).
       :           }
       :         }
 99 NDEF:       [0] {
101  365:         SEQUENCE {
105  207:           SEQUENCE {
108    3:             [0] {
110    1:               INTEGER 2
       :               }
113    1:             INTEGER 2
116   10:             SEQUENCE {
118    8:               OBJECT IDENTIFIER ecdsaWithSHA256 (1 2 840 10045 4 3 2)
       :               }
128   18:             SEQUENCE {
130   16:               SET {
132   14:                 SEQUENCE {
134    3:                   OBJECT IDENTIFIER commonName (2 5 4 3)
139    7:                   UTF8String 'AC-RAIZ'
       :                   }
       :                 }
       :               }
148   30:             SEQUENCE {
150   13:               UTCTime 23/07/2025 01:02:13 GMT
165   13:               UTCTime 28/07/2025 01:02:13 GMT
       :               }
180   42:             SEQUENCE {
182   40:               SET {
184   38:                 SEQUENCE {
186    3:                   OBJECT IDENTIFIER commonName (2 5 4 3)
191   31:                   UTF8String 'Pedro Henrique Scheidt Prazeres'
       :                   }
       :                 }
       :               }
224   89:             SEQUENCE {
226   19:               SEQUENCE {
228    7:                 OBJECT IDENTIFIER ecPublicKey (1 2 840 10045 2 1)
237    8:                 OBJECT IDENTIFIER prime256v1 (1 2 840 10045 3 1 7)
       :                 }
247   66:               BIT STRING
       :                 04 44 29 77 47 5C 95 9C 8D 1F 2A 60 3F 32 77 C5
       :                 56 F7 EF CB 71 7E 57 48 C0 78 FE 49 0D AB 37 44
       :                 71 D9 6B 0E FE 41 C9 0F 71 9F 84 90 6A EF 78 8B
       :                 79 2D C4 A6 F1 15 90 4B 23 57 29 75 EB AD 4B 2C
       :                 CA
       :               }
       :             }
315   10:           SEQUENCE {
```

Figura B.0.1 – Primeira saída do `asn1dump`

```
315   10:           SEQUENCE {
317    8:             OBJECT IDENTIFIER ecdsaWithSHA256 (1 2 840 10045 4 3 2)
       :             }
327  140:           BIT STRING, encapsulates {
331  136:             SEQUENCE {
334   66:               INTEGER
       :                 00 D0 42 99 C4 64 49 7D 11 3C 0E 81 F0 A7 B7 2B
       :                 4C 12 2E 89 63 DE AC 33 6A 45 75 A4 BA 18 A7 48
       :                 85 05 6A 4C 36 98 11 09 08 33 43 1F C3 C2 8B 18
       :                 EB 28 98 06 C3 40 07 3F 17 8C A2 A8 70 4F AB CD
       :                 37 B7
402   66:               INTEGER
       :                 01 96 CF DD 61 36 98 27 FB 25 18 F0 51 00 44 D7
       :                 86 3E 6C FE FE AC 88 93 47 14 E8 2C EA 4A 16 F6
       :                 69 FC 98 73 1B 2A EF E8 5C 2D 1E 8F 4A 4C 5D A0
       :                 66 A6 1D EB EF 85 5C 75 34 24 A4 9D 37 FD BD DF
       :                 54 8E
       :               }
       :             }
       :           }
       :         }
472  280:       SET {
476  276:         SEQUENCE {
480    1:           INTEGER 1
483   23:           SEQUENCE {
485   18:             SEQUENCE {
487   16:               SET {
489   14:                 SEQUENCE {
491    3:                   OBJECT IDENTIFIER commonName (2 5 4 3)
496    7:                   UTF8String 'AC-RAIZ'
       :                   }
       :                 }
       :               }
505    1:             INTEGER 2
       :             }
508   11:           SEQUENCE {
510    9:             OBJECT IDENTIFIER sha-256 (2 16 840 1 101 3 4 2 1)
       :             }
521  147:           [0] {
524   24:             SEQUENCE {
526    9:               OBJECT IDENTIFIER contentType (1 2 840 113549 1 9 3)
537   11:               SET {
539    9:                 OBJECT IDENTIFIER data (1 2 840 113549 1 7 1)
       :                 }
       :               }
550   28:             SEQUENCE {
552    9:               OBJECT IDENTIFIER signingTime (1 2 840 113549 1 9 5)
563   15:               SET {
565   13:                 UTCTime 23/07/2025 01:02:14 GMT
       :                 }
       :               }
580   40:             SEQUENCE {
582    9:               OBJECT IDENTIFIER
       :                 cmsAlgorithmProtection (1 2 840 113549 1 9 52)
593   27:               SET {
595   25:                 SEQUENCE {
597   11:                   SEQUENCE {
599    9:                     OBJECT IDENTIFIER
       :                       sha-256 (2 16 840 1 101 3 4 2 1)
       :                     }
610   10:                   [1] {
```

Figura B.0.2 – Segunda saída do `asn1dump`

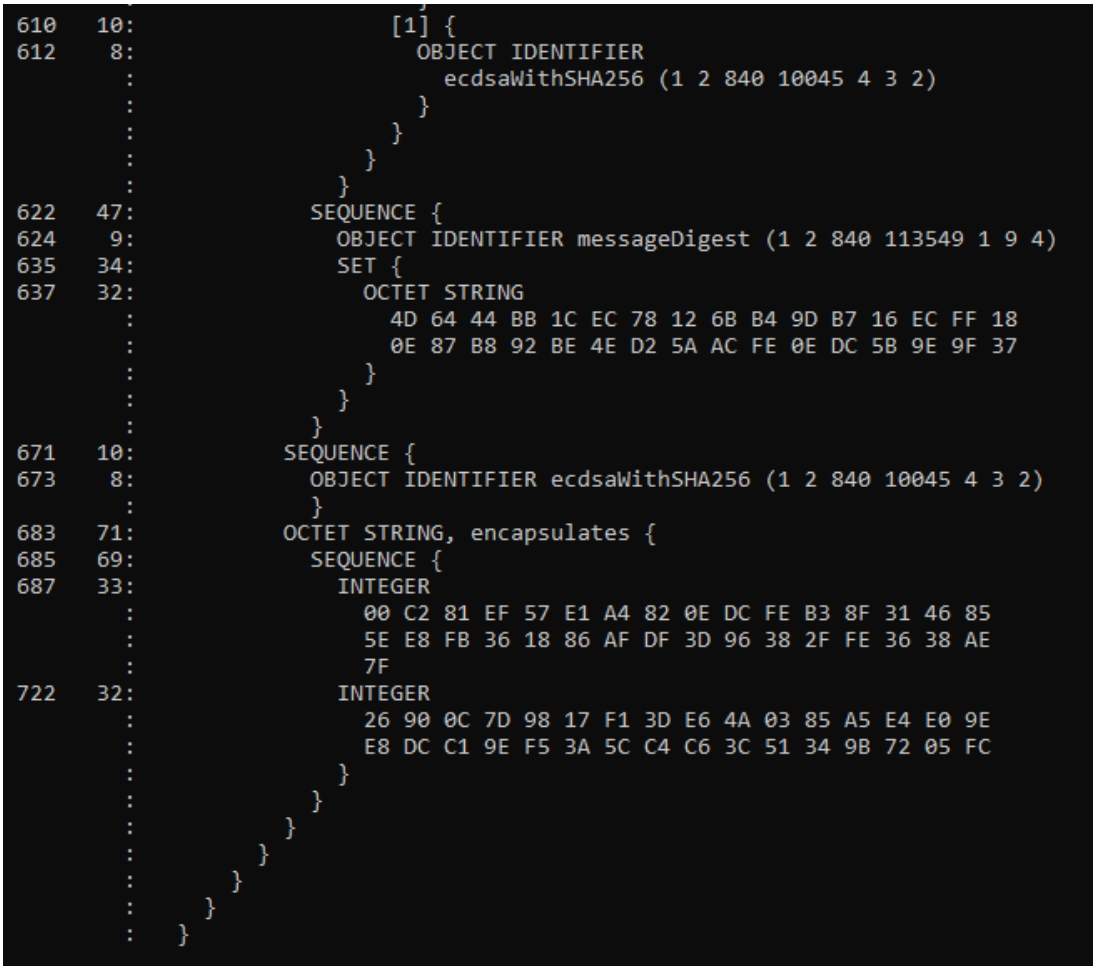

Figura B.0.3 – Terceira saída do `asn1dump`