

# Relatório Etapa Final - PBAD/LabSEC

Arthur de Farias Salmoria - 25100792

Universidade Federal de Santa Catarina

23 de julho de 2025

- 1 Introdução
- 2 Resolução das atividades
  - 2.0 Execução das Etapas
    - 2.1 Primeira Etapa
    - 2.2 Segunda Etapa
    - 2.3 Terceira Etapa
    - 2.4 Quarta Etapa
    - 2.5 Quinta Etapa
    - 2.6 Sexta Etapa
- 3 Conclusão
- 4 Referências

# 1 Introdução

Primeiramente gostaria de esclarecer que sempre tive interesse por segurança e criptografia, quando criança gostava de fazer enigmas envolvendo certas codificações e mais recentemente tenho buscado estudar a fundo o mundo da segurança computacional. Utilizando o livro de William Starling como guia de estudos junto dos cursos que fiz, da documentação provida e de diversos sites como fonte de consulta. Recorri também a LLMs para verificar informações, analisar o código e auxiliar com a sintaxe de Java e da biblioteca Bouncy Castle. Acredito ter produzido um trabalho sólido, resultado de muito esforço e estudo, pois todos os conceitos aplicados foram aprendidos de forma autodidata, fora do currículo universitário, uma vez que acabo de finalizar o primeiro semestre.

Busquei seguir a estrutura de arquivos, classes e métodos já providos e organizados, tentando também levar o sistema para além do exigido, porém, construir o sistema com a base já provida foi essencial para eu não me perder durante o processo.

Além disso, gostaria de destacar que mesmo tendo um conhecimento prévio na utilização de java, OpenSSL, teoria de criptografia assimétrica, hashes e de assinatura digital, a aplicação desses diferentes conceitos aprendidos em um sistema integrado foi algo que me fez crescer muito como desenvolvedor e estudante. Já as etapas referentes à certificação digital e a verificação de assinaturas sendo aplicadas em Java foram novidade para mim, já que anteriormente ao desafio eu apliquei tais conceitos apenas pela linha de comando com o OpenSSL. Testei todas as etapas utilizando o OpenSSL, lembrando que as chaves e certificados criados (as respostas) são diferentes a cada execução, pois é sempre criada uma nova, portanto, não necessariamente as imagens do relatório fazem referência à mesma chave.

Também gravei um vídeo executando o programa no IntelliJ IDEA, para fazer a gravação utilizei o OBS Studio no computador que já tinha o programa baixado e o software de edição (linux não tem o adobe premiere). O arquivo estaria localizado na mesma pasta deste relatório, porém, o arquivo ficou muito grande. Então fiz o upload no youtube e para acessá-lo pelo, esse é o link clicável: <https://www.youtube.com/watch?v=CKci6NrJl4c>.

Para assinar esse documento utilizei o ADES, aplicação do LabSEC apresentada pelo professor Custódio no LabSEC DAY que é uma aplicação de assinatura digital a prova de

computadores quânticos mesmo utilizando criptografia clássica, isso pelo encadeamento dos documentos assinados e por descartar a chave privada após cada assinatura.

Na página 23, após a conclusão, apresento um diagrama de classes simples que fiz para ilustrar como o sistema foi pensado com base na estrutura já provida.

## 2 Resolução das atividades

### 2.0 Execução das Etapas

Primeiramente fiz um menu simples na classe “ExecutarEtapas”, pois já havia feito um sistema parecido e acredito deixar toda a aplicação mais bonita e organizada. Sobre o menu, você deve utilizá-lo inserindo o número correspondente de cada etapa, sabendo que caso você execute a quinta etapa antes das outras, ela utilizará os artefatos da última execução. Além disso, caso você gere novas chaves e certificados e verifique a assinatura antiga com esses novos artefatos, a aplicação dará a assinatura como inválida, pois de fato, considerando que os artefatos utilizados na assinatura foram reescritos com novos completamente diferentes. Preferi permitir a execução errada da aplicação para que se possa provar que de fato ela não está considerando qualquer assinatura como válida.

Figura 2.0.1

```
===== DESAFIO FINAL LABSEC =====  
Sistema de Certificação e Assinatura Digital  
=====
```

```
===== MENU PRINCIPAL =====  
1. Primeira Etapa: Calcular resumo criptográfico (hash)  
2. Segunda Etapa: Gerar par de chaves assimétricas  
3. Terceira Etapa: Gerar certificados digitais  
4. Quarta Etapa: Gerar repositório de chaves seguro  
5. Quinta Etapa: Gerar uma assinatura digital  
6. Sexta Etapa: Verificar uma assinatura digital  
7. Sair do programa  
Escolha uma opção:
```

Figura 2.0.2

```
Escolha uma opção: 6
Sexta Etapa: Verificação completa da assinatura
Arquivo de assinatura lido: src/main/resources/artefatos/assinaturas/assinatura.der
Âncora de confiança carregada: CN=AC-RAIZ

--- RESULTADO FINAL DA VERIFICAÇÃO ---
>>> ASSINATURA INVÁLIDA OU NÃO CONFIÁVEL <<<
Sexta Etapa concluída.

Pressione Enter para continuar...
Erro de validação do caminho: Path does not chain with any of the trust anchors
O caminho de certificação do assinante não é válido.
```

## 2.1 Primeira Etapa

A primeira etapa é basicamente a implementação de uma função de Hash usando o algoritmo SHA-256, essa foi uma tarefa relativamente tranquila, pois já fiz um algoritmo similar. Utilizei a classe Resumidor para ser o componente central e reutilizável para o cálculo de resumos criptográficos, os hashes. Primeiramente fiz um algoritmo em linguagem natural para me guiar no processo da primeira etapa, sendo ele:

Classe Resumidor

```
Resumidor():
```

```
    Utilizar algoritmoResumo da classe Constantes
```

```
    resumir(arquivoDeEntrada):
```

```
        Ler todos os bytes do arquivoDeEntrada
```

```
        Retornar o resumo (hash) dos bytes lidos
```

```
    bytesToHex(bytesDoHash):
```

```
        Converter os bytesDoHash para uma String em formato
```

```
        Hexadecimal e retorná-la
```

Classe PrimeiraEtapa

```
    executarEtapa():
```

```
        ArquivoDeEntrada = definir caminho para
```

```
        textoPlano.txt // Alterei para importar o caminho
```

```
        das constantes depois, mas esse foi o raciocínio do
```

```
        algoritmo inicial
```

```
SaidaHash = definir caminho para
resumoTextoPlano.hex
resumidor = criar uma nova instância da classe
Resumidor
ArquivoResumidoBytes = chamar
resumidor.resumir(ArquivoDeEntrada)
ArquivoResumidoHex = chamar
Resumidor.bytesToHex(ArquivoResumidoBytes)
Armazenar ArquivoResumidoHex no arquivo definido por
SaidaHash
```

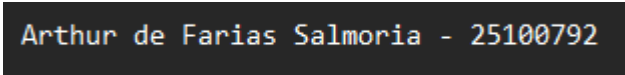
Após isso comecei a implementação. Ao ser instanciado, o construtor (“Resumidor”) busca o nome do algoritmo de hash a partir da classe “Constantes”. Utiliza “MessageDigest.getInstance(this.algoritmo)” para obter uma instância. O construtor declara que pode lançar uma “NoSuchAlgorithmException”, delegando o tratamento de erro para a classe que o chama. Isso é apropriado, pois a falha na obtenção do algoritmo é um erro de que deve ser tratado no contexto mais amplo da aplicação.

Para a implementação do método “resumir”, foi adotada uma abordagem direta e simplificada, adequada ao projeto. A decisão foi carregar o arquivo inteiro na memória antes de processá-lo. Embora menos escalável para arquivos de grande volume, essa técnica resulta em um código mais conciso e de fácil compreensão, e como o desafio utiliza um arquivo pequeno não teria problema, caso fossem arquivos maiores seria melhor utilizar um buffer e alocar memória de pouco em pouco, pois até mesmo um ator malicioso poderia fornecer um arquivo de entrada com vários gigabytes, o que levaria a uma exceção “OutOfMemoryError”, causando a interrupção abrupta da aplicação. Porém, como antes dito, não é o caso do cenário proposto no desafio.

O resultado de um hash é um conjunto de bytes, que não é legível. O requisito era armazenar uma representação textual em hexadecimal, portanto criei o método “bytesToHex”.

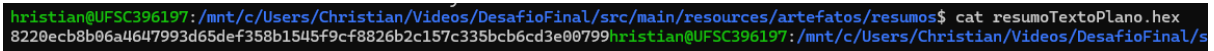
Já a classe “PrimeiraEtapa” utiliza dos métodos da classe “Resumidor” para resumir o arquivo de entrada contendo meu nome e minha matrícula, no final ela armazena o hash em hexadecimal no arquivo desejado.

Figura 2.1.1



```
Arthur de Farias Salmoria - 25100792
```

Figura 2.1.2



```
hristian@UFSC396197:/mnt/c/Users/Christian/Videos/DesafioFinal/src/main/resources/artefatos/resumos$ cat resumoTextoPlano.hex
8220ecb8b06a4647993d65def358b1545f9cf8826b2c157c335bcb6cd3e00799hristian@UFSC396197:/mnt/c/Users/Christian/Videos/DesafioFinal/s
```

## 2.2 Segunda Etapa

A segunda etapa é a criação e armazenamento de chaves públicas e privadas criadas por curva elíptica. Ela se baseia na matemática de curvas elípticas para criar chaves criptográficas que são menores, mais eficientes e oferecem segurança equivalente ou superior em comparação com métodos tradicionais como RSA, sendo que ambos usam conceitos da matemática discreta como a aritmética modular em seus algoritmos.

O algoritmo que usei como base para a implementação foi feito em linguagem natural, sendo ele:

Classe GeradorDeChaves:

GeradorDeChaves():

Utilizar algoritmoChave da classe Constantes

gerarParDeChaves(tamanhoDaChave):

Mapear o tamanhoDaChave (256 ou 521) para o nome da curva elíptica

Inicializar a geração com os parâmetros da curva

Retornar o par de chaves gerado

Classe EscriitorDeChaves:

escreveChaveEmDisco(chave, nomeDoArquivo):

Verificar se a chave é pública ou privada

Botar a chave num objeto no formato PEM

Colocar no arquivo

Classe SegundaEtapa:

executarEtapa():

caminhoChavePrivadaUsuario = "caminho"

caminhoChavePublicaUsuario= "caminho"

```
caminhoChavePrivadaRaiz= "caminho"
caminhoChavePublicaRaiz= "caminho"
gerador = GeradorDeChaves()
parDeChavesUsuario = gerador.gerarParDeChaves(256)
Chamar EscriitorDeChaves.escreveChaveEmDisco para a
chave privada do parDeChavesUsuario
Chamar EscriitorDeChaves.escreveChaveEmDisco para a
chave pública do parDeChavesUsuario
parDeChavesRaiz = gerador.gerarParDeChaves(521)
Chamar EscriitorDeChaves.escreveChaveEmDisco para a
chave privada do parDeChavesRaiz
Chamar EscriitorDeChaves.escreveChaveEmDisco para a
chave pública do parDeChavesRaiz
```

Após isso, comecei a implementação do algoritmo. Para garantir que Bouncy Castle estivesse sempre disponível, usei um bloco de inicialização estático que registra o provedor na JVM. Fiz isso, pois ao executar a segunda etapa sem executar a primeira o programa voltava com erro, então implementei isso nas outras etapas também. O construtor da classe “GeradorDeChaves” não recebe parâmetros, e busca o algoritmo de chave (“EC”) da classe Constantes. O método “gerarParDeChaves” mapeia um tamanho de chave em bits (256 ou 521) para o nome da curva correspondente, sendo secp256r1 ou secp521r1, e inicializa o gerador com esses parâmetros. Pesquisei sobre qual curva usar e escolhi essa após ver em diversos sites inclusive um post no site “crypto.stackexchange” que descreve bem a comparação entre as duas opções que havia encontrado: *“The main difference is that secp256k1 is a Koblitz curve, while secp256r1 is not. Koblitz curves are known to be a few bits weaker than other curves, but since we are talking about 256-bit curves, neither is broken in “5-10 years” unless there’s a breakthrough.”*. Portanto, escolhi secp256r1 e secp521r1, pois embora a curva secp256k1, ofereçam vantagens de performance devido à sua estrutura Koblitz, a secp256r1 representa uma escolha mais conservadora e alinhada com os padrões de infraestruturas de chave pública de larga escala de acordo com as minhas pesquisas. Porém, ambas são as curvas recomendadas pelo NIST (National Institute of Standards and Technology) em publicações como FIPS 186-5 para uso governamental, garantindo um alto nível de segurança e ampla interoperabilidade.

Para a persistência das chaves, a classe “EscritorDeChaves” foi implementada com a responsabilidade de salvar uma chave em disco no formato PEM. O código recebe as chaves e utiliza o Bouncy Castle para escrever as chaves no formato PEM. Adicionei a verificação se a chave fornecida é uma instância de PublicKey ou PrivateKey, permitindo seja escrito o cabeçalho correto no arquivo PEM (“PUBLIC KEY” ou “PRIVATE KEY”).

Por fim, a classe “SegundaEtapa” utiliza os serviços das outras duas classes para cumprir os requisitos da tarefa. Primeiro, ela instancia o “GeradorDeChaves” e o utiliza para criar dois pares de chaves distintas: um de 256 bits para o usuário (que seria o meu par de chaves) e outro par de 521 bits para a AC-Raiz. Para cada par gerado, ela chama o método “EscritorDeChaves.escreveChaveEmDisco” para salvar a chave privada e a pública nos arquivos .pem. Todo o fluxo é encapsulado em um bloco try-catch para lidar com possíveis exceções durante o processo.

Após as chaves serem criadas para AC-raiz e a do usuário, testei no OpenSSL, lembrando que não será a mesma quando o programa é executado novamente, pois é criada uma nova.

Figura 2.2.1

```
christian@FSC396197: /mnt/c/Users/Christian/Videos/DesafioFinal/src/main/resources/artefatos/chaves$ openssl ec -in chavePrivadaAcRaiz.pem -text -noout
read EC key
Private-Key: (521 bit)
priv:
  01:44:62:fa:1b:5e:85:d1:9f:ae:7d:30:04:b9:28:
  95:d6:28:f0:c1:85:32:14:7e:36:a5:aa:5c:c1:31:
  12:f6:19:af:df:ad:2e:26:c3:db:d3:aa:67:36:13:
  1b:ca:9b:6d:70:35:4a:b9:42:53:de:b9:ed:ed:47:
  cc:9d:8a:36:fa:80
pub:
  04:01:93:cf:ab:d3:2c:6b:52:52:00:39:b8:fc:2a:
  fe:38:5f:a7:ea:0c:98:c5:5d:83:db:82:a6:7f:cb:
  74:7c:47:94:6c:5d:5d:a0:98:5f:76:f0:bb:43:1d:
  cb:e2:2e:d3:07:3c:0f:7e:a2:3f:d6:19:42:c8:84:
  20:b7:a7:5b:da:c0:81:01:f7:e1:01:1c:55:36:3f:
  cb:cb:48:63:3f:eb:0f:92:d8:03:d7:82:c7:da:8a:
  d8:d4:9c:02:9c:1a:3e:8c:d4:32:52:4c:cc:4c:6f:
  d6:ba:c7:df:8a:79:a3:05:53:b6:6c:a6:e7:c0:77:
  d2:ec:f3:a5:08:32:a1:fa:f4:3f:b7:77:cf
ASN1 OID: secp521r1
NIST CURVE: P-521
```

Figura 2.2.2

```
christian@FSC396197: /mnt/c/Users/Christian/Videos/DesafioFinal/src/main/resources/artefatos/chaves$ openssl ec -in chavePrivadaUsuario.pem -text -noout
read EC key
Private-Key: (256 bit)
priv:
  64:21:88:29:91:99:43:aa:20:39:db:24:53:a9:14:
  48:c4:b5:12:22:a6:17:a9:45:f1:48:0f:32:f2:8d:
  60:86
pub:
  04:bc:01:01:ed:83:c4:02:7e:ea:08:14:29:f6:5b:
  fd:5d:61:62:04:4f:97:bf:6c:9a:e4:4a:5a:6e:4f:
  49:d9:81:3c:0c:07:4e:5f:96:e0:5d:9a:e0:79:bf:
  7e:e4:d9:2d:79:bf:33:b2:e8:80:e3:e8:33:05:3b:
  d9:fb:4a:a7:0b
ASN1 OID: prime256v1
NIST CURVE: P-256
```

## 2.3 Terceira Etapa

Na terceira etapa tive que implementar um sistema que gera dois certificados sendo um certificado auto-assinado da AC-Raiz, e um certificado para o usuário, que seria o meu. Essa foi



uma etapa mais complexa, pois nunca havia feito programa algum com certificado X.509, apenas no OpenSSL. Também, porque fiz junto a classe “GeradorDeAssinatura”, pois no algoritmo que eu havia pensado inicialmente seria mais prático já codificá-la e reutilizá-la na quinta etapa. Já conhecia grande parte dos conceitos, mas mesmo assim fui procurar saber mais sobre uma possível implementação e então cheguei em um algoritmo:

Classe LeitorDeChaves:

```
lerChavePublicaDoDisco(caminhoDoArquivo, algoritmo):  
    Ler o arquivo PEM do caminhoDoArquivo  
    Converter o conteúdo lido para um objeto de Chave  
    Pública  
    Retornar a Chave Pública  
  
lerChavePrivadaDoDisco(caminhoDoArquivo, algoritmo):  
    Ler o arquivo PEM do caminhoDoArquivo  
    Converter o conteúdo lido para um objeto de Chave  
    Privada  
    Retornar a Chave Privada
```

Classe GeradorDeCertificados:

```
gerar(dados do certificado):  
    Chamar o método interno gerarEstruturaCertificado  
    para criar a parte principal do certificado.  
    Chamar o método interno  
    geraValorDaAssinaturaCertificado para criar a  
    assinatura dessa estrutura  
    Chamar o método interno montarCertificado para  
    juntar tudo e retornar o certificado final.
```

Classe EscritorDeCertificados:

```
escreveCertificado(nomeDoArquivo, certificado):  
    Garantir que a pasta de destino para o nomeDoArquivo  
    exista  
    Empacotar o certificado em um objeto no formato PEM  
    com a descrição "CERTIFICATE"
```

Escrever este objeto PEM no arquivo definido por  
nomeDoArquivo

Classe TerceiraEtapa:

executarEtapa():

```
    caminhosChavesEntrada = "Caminhos"
    caminhosCertificadosSaida = "caminhos"
    Usuário = meu nome, minha matrícula
    geradorDeCertificados = criar uma nova instância da
    classe GeradorDeCertificados
    Carregar a chave pública do usuário e as chaves
    pública e privada da AC-Raiz usando o LeitorDeChaves
    Para o certificado da AC-Raiz (autoassinado):
        certificadoRaiz = chamar
        geradorDeCertificados.gerar, passando:
            Chave pública da AC-Raiz como titular
            Chave privada da AC-Raiz como emissor
            Nome "AC-Raiz" como titular e emissor
        Chama EscriitorDeCertificados.escreveCertificado
        para salvar certificadoRaiz
    Para o certificado do Usuário:
        certificadoUsuario = chamar
        geradorDeCertificados.gerar, passando:
            Chave pública do usuário como titular
            Chave privada da AC-Raiz como emissor
            Nome do usuário como titular e "AC-Raiz"
            como emissor
        Chama EscriitorDeCertificados.escreveCertificado
        para salvar o certificadoUsuario
```

E então, comecei a codificar a terceira etapa. Como visto, foram utilizadas quatro classes principais, porém, tive de alterar, retirar e adicionar certos elementos nesse algoritmo.

A classe "LeitorDeChaves" carregava as chaves em formato PEM, enquanto o "GeradorDeCertificados" criava os certificados, que eram salvos em disco pelo

“EscritorDeCertificados”. Contudo, durante a execução da sexta etapa, ao validar a assinatura com a ferramenta OpenSSL, foi encontrado um erro. Após analisar o código e o resultado, descobri que poderiam ter faltado as extensões X.509 v3 no certificado da AC-Raiz, que de acordo com o meu entendimento são indispensáveis para definir seu propósito. A ausência da extensão BasicConstraints impedia que ele fosse reconhecido como uma Autoridade Certificadora (AC) válida.

Diante disso, a classe “GeradorDeCertificados” foi refatorada para uma abordagem moderna, utilizando o JcaX509v3CertificateBuilder do Bouncy Castle. Para o certificado da AC-Raiz, foram adicionadas as extensões BasicConstraints(cA=TRUE) e KeyUsage(keyCertSign). Para o certificado de usuário, foram incluídas as extensões BasicConstraints(cA=FALSE), KeyUsage(digitalSignature) e AuthorityKeyIdentifier, criando um vínculo com a AC. A classe “TerceiraEtapa”, que orquestra a geração, foi ajustada para essa nova lógica, passando a gerar uma hierarquia de certificados funcional e compatível com os padrões.

Por fim, a classe “TerceiraEtapa” foi ajustada para orquestrar essa nova lógica de geração. Suas chamadas ao método gerar foram adaptadas para incluir novos parâmetros: um booleano para indicar se o certificado a ser criado é de uma AC (ehAC), e o próprio objeto de certificado do emissor. Com essa correção, a “TerceiraEtapa” passou a gerar uma hierarquia de certificados funcional e compatível com os padrões de mercado, permitindo que a validação na sexta etapa fosse concluída com sucesso. Havia mudado o tempo de validade para 3652 dias para contar ano bissexto, porém, ao ver que a validade do ADES por exemplo, não segue exatamente 100 anos após a emissão, acreditei que seria melhor deixar algo padronizado como se fosse realmente 365 dias (1 ano) multiplicado por quantos anos devem ser.

Figura 2.3.1

```
christian@UFSC396197:/mnt/c/Users/Christian/Videos/DesafioFinal/src/main/resources/artefatos/certificados$ openssl x509 -in certificadoUsuario.pem -text -noout
Certificate:
  Data:
    Version: 3 (0x2)
    Serial Number: 25100792 (0x17f01f8)
    Signature Algorithm: ecdsa-with-SHA256
    Issuer: CN = AC-RAIZ
    Validity
      Not Before: Jul 17 01:21:18 2025 GMT
      Not After : Jul 17 01:21:18 2026 GMT
    Subject: CN = Arthur de Farias Salmoria
    Subject Public Key Info:
      Public Key Algorithm: id-ecPublicKey
      Public-Key: (256 bit)
      pub:
        04:84:a8:01:88:c6:e6:ed:a5:e6:13:bc:69:54:a8:
        a0:64:bc:3a:b9:43:02:13:84:05:73:b1:9e:62:26:
        40:d7:ef:78:65:47:89:90:31:29:54:3e:a0:c7:d8:
        d0:74:c5:1e:b3:d9:81:f3:12:76:8f:1f:bd:1c:ec:
        46:9c:79:5d:d2
      ASN1 OID: prime256v1
      NIST CURVE: P-256
    X509v3 extensions:
      X509v3 Subject Key Identifier:
        98:CD:4D:A2:6F:C4:22:8A:50:47:55:1C:DF:DF:8C:6B:26:CE:F1:DA
      X509v3 Basic Constraints: critical
        CA:FALSE
      X509v3 Key Usage: critical
        Digital Signature
      X509v3 Authority Key Identifier:
        keyid:8F:15:48:6A:2A:2C:CA:F8:DA:1A:02:63:D4:D3:3A:0D:AF:D3:C3:FF
        DirName:/CN=AC-RAIZ
        serial:01
    Signature Algorithm: ecdsa-with-SHA256
    Signature Value:
      30:81:88:02:42:00:9a:15:2e:73:af:cd:01:78:fe:6c:bb:6f:
      50:b8:17:50:a2:d5:1e:7f:35:00:91:93:f8:0d:c4:40:87:ba:
      59:63:b9:08:8e:ab:ff:0b:09:ed:d6:67:d8:2d:72:8c:29:7c:
      c8:57:5f:2c:4f:84:e5:f2:b3:d1:0e:bb:6d:ec:bb:32:4d:02:
      42:01:09:74:32:0a:f6:55:dc:c1:e5:c5:5a:2b:c8:75:21:4e:
      8f:c4:76:46:0d:c6:1b:ac:ee:a9:be:d5:4f:25:aa:f5:fd:3c:
      2a:dc:18:9f:26:5e:2c:0c:63:15:d3:a2:e9:71:ba:08:ed:b3:
      ec:d4:7d:a6:4b:dd:56:a0:ea:56:d5:0e:56
```

Figura 2.3.2

```
christian@UFSC396197:/mnt/c/Users/Christian/Videos/DesafioFinal/src/main/resources/artefatos/certificados$ openssl x509 -in certificadoAcRaiz.pem -text -noout
Certificate:
  Data:
    Version: 3 (0x2)
    Serial Number: 1 (0x1)
    Signature Algorithm: ecdsa-with-SHA256
    Issuer: CN = AC-RAIZ
    Validity
      Not Before: Jul 17 01:21:18 2025 GMT
      Not After : Jul 15 01:21:18 2035 GMT
    Subject: CN = AC-RAIZ
    Subject Public Key Info:
      Public Key Algorithm: id-ecPublicKey
      Public-Key: (521 bit)
      pub:
        04:01:5b:bb:6a:10:5f:60:d3:d9:10:85:6a:43:e5:
        be:52:2f:dd:f1:01:af:3d:89:44:8f:55:78:fd:d0:
        bf:cf:25:06:fc:08:13:41:52:2b:79:02:ec:b1:01:
        0d:42:10:6f:53:44:34:e6:5c:af:67:22:48:b2:10:
        00:63:45:91:01:1c:64:00:64:f5:ee:3e:73:a3:9f:
        a1:da:73:5d:80:4f:a7:4a:3f:a9:4b:9c:5b:b4:b3:
        ed:21:73:1f:73:54:da:37:db:b8:45:24:69:77:79:
        dd:a7:83:9b:b5:66:f8:ba:47:fa:32:68:19:1a:79:
        c2:d5:b2:97:ce:9f:dc:cd:25:06:36:18:e0
      ASN1 OID: secP521r1
      NIST CURVE: P-521
    X509v3 extensions:
      X509v3 Subject Key Identifier:
        8F:15:48:6A:2A:2C:CA:F8:DA:1A:02:63:D4:D3:3A:0D:AF:D3:C3:FF
      X509v3 Basic Constraints: critical
        CA:TRUE
      X509v3 Key Usage: critical
        Certificate Sign, CRL Sign
    Signature Algorithm: ecdsa-with-SHA256
    Signature Value:
      30:81:87:02:42:01:ed:09:d1:84:51:92:91:37:96:bf:64:d5:
      37:e1:62:e4:4f:93:73:84:4d:9a:20:75:0b:55:1d:af:29:05:
      8f:9b:64:78:a4:2d:23:66:51:5d:59:3a:6f:94:1c:02:bf:b8:
      39:a2:c0:18:ef:a1:56:30:b2:d0:54:6e:72:6c:33:67:27:02:
      41:3e:39:a5:cd:4a:7e:ca:50:ea:71:60:c6:f2:88:16:5d:fc:
      bb:48:f8:a6:24:2a:53:51:4d:b7:ce:e5:ba:6d:56:9c:84:2b:
      ad:7d:3b:6f:d9:32:fc:2e:8d:87:20:8a:b0:f9:f3:cf:ff:ce:
      55:36:ea:65:84:2b:cc:db:df:48:26:0e
```

## 2.4 Quarta Etapa

Ao ler o problema pela primeira vez surgiram algumas dúvidas, no trecho “O PKCS#12 ainda conta com uma senha, que serve para cifrar a estrutura (isso é feito de modo automático).” da documentação do desafio é citado uma senha utilizada para cifrar a estrutura, e então me perguntei

“que senha seria essa?”, “como seria feita a cifração?”, “seria utilizada uma cifração simétrica utilizando a senha como chave?” e entre outras perguntas.

Não havia utilizado da maneira correta a classe “Constantes”, portanto acabei salvando os arquivos com nomes errados e ao fazer essa etapa com nome certo o programa não funcionava, e eu não entendia o porquê. Ao corrigir esse erro, acabei por melhorar o código das últimas etapas por deixá-los mais limpos utilizando a classe. Vi também que a própria senha estava na classe que seria utilizada para acessar o arquivo e seria a minha matrícula, essa sendo uma senha extremamente insegura e não recomendável, porém, vou seguir o que já foi pré-estabelecido nas constantes, mesmo que a utilização de uma senha mais forte seja recomendada. E então desenvolvi um algoritmo em linguagem natural como nas outras:

Classe LeitorDeChaves:

```
lerChavePrivadaDoDisco(caminhoDoArquivo, algoritmo):  
    Ler o arquivo no formato PEM  
    Extrair os bytes  
    Transformar os bytes em um objeto de Chave Privada  
    Retornar a Chave Privada  
  
lerChavePublicaDoDisco(caminhoDoArquivo, algoritmo):  
    Ler o arquivo no formato PEM  
    Extrair os bytes  
    Transformar os bytes em um objeto de Chave Pública  
    Retornar a Chave Pública
```

Classe LeitorDeCertificados:

```
lerCertificadoDoDisco(caminhoDoArquivo):  
    Criar uma "fábrica de certificados" com formato  
    X.509  
    Ler o arquivo  
    Converter o conteúdo do arquivo em um objeto de  
    Certificado  
    Retornar o Certificado
```

Classe GeradorDeRepositorios:

```
gerarPkcs12(chavePrivada, certificado, caminhoDoArquivo,
alias, senha):
```

```
    Criar um repositório no formato PKCS#12
```

```
    Adicionar ao repositório o alias
```

```
    Guardar a chavePrivada e o certificado
```

```
    Salvar o repositório no arquivo com senha
```

Classe QuartaEtapa:

```
    executarEtapa():
```

```
        // Repositório do Usuário
```

```
        Chamar o LeitorDeChaves para carregar a chave
privada do usuário
```

```
        Chamar o LeitorDeCertificados para carregar o
certificado do usuário
```

```
        Chamar o GeradorDeRepositorios para criar o arquivo
.pl2, passando a chave e o certificado recém-lidos,
junto com o caminho de saída, o alias e a senha do
usuário
```

```
        // Repositório da AC-Raiz
```

```
        Chamar o LeitorDeChaves para carregar a chave
privada da AC-Raiz
```

```
        Chamar o LeitorDeCertificados para carregar o
certificado da AC-Raiz
```

```
        Chamar o GeradorDeRepositorios para criar o arquivo
.pl2, passando a chave e o certificado da AC-Raiz,
junto com o caminho de saída, o alias e a senha da
AC-Raiz
```

A classe “GeradorDeRepositorios” foi o componente central, com a função de empacotar uma chave privada e um certificado em um único arquivo. Ao analisar a implementação padrão de PKCS#12 percebi uma lacuna de segurança que eu poderia melhorar: por padrão, de acordo com as minhas pesquisas, muitas bibliotecas utilizam o esquema de criptografia baseado em senha pbeWithSHA1And3-keyTripleDES-CBC. Este esquema é considerado obsoleto e vulnerável a ataques de força bruta, devido à sua dependência do SHA-1 e do 3DES.

A princípio eu utilizei esse esquema de criptografia, isso pois era o pedido pelo desafio e queria seguir à risca o recomendado. Porém, acredito que essa percepção de segurança e de achar por vulnerabilidades é uma habilidade que poderia ser demonstrada por uma resolução diferente desta etapa, então, após pesquisas e diversas análises do código, entendi os riscos e comecei a implementar as mudanças. Adotei o esquema PBES2 (Password-Based Encryption Scheme 2), conforme recomendado pelo PKCS#5 v2.1. A cifragem da chave privada foi configurada para usar AES-256-CBC, e a derivação da chave a partir da senha foi realizada com PBKDF2, utilizando uma contagem de iterações elevada e o HMAC-SHA256 como função pseudo-aleatória (conceito que eu já havia estudado em um curso de criptografia que fiz do Coursera). Esta abordagem aumenta drasticamente a resistência do repositório a ataques de adivinhação de senha por força bruta.

A classe “QuartaEtapa” atuou como orquestradora, gerenciando a criação de dois repositórios seguros distintos: um para o usuário e outro para a AC-Raiz. Em ambos os casos, ela primeiro carregou as credenciais individuais dos arquivos .pem e, em seguida, invocou o GeradorDeRepositorios para criar os arquivos .p12 com os parâmetros de segurança aprimorados. Todas as informações, como caminhos, aliases e a senha mestra, foram fornecidas pela classe Constantes.

Figura 2.4.1

```

Christian@UFS396197: /mnt/c/Users/Christian/Videos/DesafioFinal/src/main/resources/artefatos/repositorios$ openssl pkcs12 -in repositorioAcRaiz.p12 -info
Enter Import Password:
MAC: sha1, Iteration 102400
MAC length: 20, salt length: 20
PKCS7 Data
Shrouded Keybag: pbeWithSHA1And3-KeyTripleDES-CBC, Iteration 51200
Bag Attributes
    friendlyName: AC-RAIZ
    localKeyID: 5C E0 85 B7 28 5C 11 EB ED 00 20 AB AB 5F 3A B2 04 77 3B 52
Key Attributes: <No Attributes>
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
-----BEGIN ENCRYPTED PRIVATE KEY-----
MIIBXTBqkqhkiG9w0BBQwSjApBgkqhkiG9w0BBQwHAQIE3EhEVfS92cCaggA
NAwGCCqGSIb3DQIJBQAwHQYJYIZIAU0BAEqBBCr6SY6czS+clxLb+nnf4Mtv8IIB
ALBo/huc0MesbLW0PF2aARysQ6CKEqY28BDmxQuYJLd17Iup0741XmB1x2XjAntQ
6ACAYz30Byr8EppYN12bdcCY1yC7ALJHdfswad/dArwx9iREEBsVpJfH6NdaWAMK
FINZYwFfDNFwUPb+zfIguFede58q6vFo8HzCB/LVRC1m61TXwe5WqnmCt2tsejOf
+A8LIgdd+0jA0UK23RfGhPlh+BKZApJpPR8vRxlSkZvOCZLrJjx7GIENuouEgluL
2X5RSmKfQNB4aPsS/djcQkUyMPBY9Ys113KuRh6QzmqZz3wQs0JOHe8QUpZWdm+
7sDnnk0b/MHYvisSjczupIQ=
-----END ENCRYPTED PRIVATE KEY-----
PKCS7 Encrypted data: pbeWithSHA1And40B1tRC2-CBC, Iteration 51200

```

Figura 2.4.2

```

Christian@UFS396197: /mnt/c/Users/Christian/Videos/DesafioFinal/src/main/resources/artefatos/repositorios$ openssl pkcs12 -in repositorioUsuario.p12 -info
Enter Import Password:
MAC: sha1, Iteration 102400
MAC length: 20, salt length: 20
PKCS7 Data
Shrouded Keybag: pbeWithSHA1And3-KeyTripleDES-CBC, Iteration 51200
Bag Attributes
    localKeyID: D4 DB 18 0E 41 8C A5 87 0F F1 FD 5D 95 21 60 A9 2E E0 F1 05
    friendlyName: Arthur de Farias Salmoria
Key Attributes: <No Attributes>
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
-----BEGIN ENCRYPTED PRIVATE KEY-----
MIHsHFcGCSqGSIb3DQEFDTBKMCKGCSqGSIb3DQEFDDAcBAIXswg9gd+t/AICCAAw
DAYIKoZIhvcNAgkFADAdBgkqhkgBZQMEASoEEA622x32iVcEJuoXFCWIKMEgZCE
KYCRW58gzBeybjw7T0mb9WnX4N680PFWL Lu+8h0wat+GyGYqbnufEZNQdYG6oM/
AVpP68670XFr0aR1AjSdTG6vBrE1Mph+CURWsQirUoNr4y9WwYUsRSyk0zS3kzUH
K8pJvnOVRKs3dz0Pr4kt/mhCD9ooy1oZsSeqdPvxQnttdL1ns4mBQxmFrA2/TGs=
-----END ENCRYPTED PRIVATE KEY-----
PKCS7 Encrypted data: pbeWithSHA1And40B1tRC2-CBC, Iteration 51200

```

Figura 2.4.3



Figura 2.4.4



## 2.5 Quinta Etapa

Já tive contato e possuo conhecimento na área de assinaturas digitais, porém, também nunca havia codificado uma aplicação dela, mesmo já sabendo como um possível algoritmo poderia ser codificado. Pesquisei então sobre CMS, *“The Cryptographic Message Syntax (CMS) is the IETF's standard for cryptographically protected messages. It can be used by cryptographic schemes and protocols to digitally sign, digest, authenticate or encrypt any form of digital data. CMS is based on the syntax of PKCS #7, which in turn is based on the Privacy-Enhanced Mail standard.”*

Após a pesquisa, passei a pensar em um possível algoritmo que deve receber o arquivo, a chave privada e o meu certificado, faça o hash do meu documento, criptografe ele com a chave privada e no final, empacote o documento original com o hash cifrado e o certificado. Pensei em implementar a classe “Resumidor” para pegar o hash, porém, pelo padrão CMS de assinatura aparentemente não seria possível por, pelo meu entendimento, ser um padrão que foi feito para “fazer tudo sozinho”; ele mesmo calcula o hash do documento internamente antes de assinar. Segue o algoritmo que pensei:

Classe GeradorDeAssinatura:

GeradorDeAssinatura():

Preparar a ferramenta principal

(CMSSignedDataGenerator) para gerar assinaturas



```

informaAssinante(certificado, chavePrivada):
    Guardar o certificado e a chavePrivada
assinar(caminhoDocumento):
    Chamar preparaDadosParaAssinar para preparar o
    arquivo
    Chamar preparaInformacoesAssinante para criar o
    objeto SignerInfoGenerator do assinante.
    Adicionar SignerInfoGenerator do assinante ao
    gerador principal
    Adicionar o certificado do assinante ao gerador
    principal
    Gerar o pacote de assinatura final, que inclui o
    documento
    Retornar o pacote de assinatura
preparaInformacoesAssinante(chavePrivada, certificado):
    Criar o objeto ContentSigner que vai usar a
    chavePrivada para criptografar o hash
    Preparar um construtor de informações do assinante
    Usar o construtor para empacotar o ContentSigner e
    o certificado
    Retorna essas informações empacotadas
escreveAssinatura(arquivo, assinatura):
    Pegar os bytes do pacote de assinatura
    Escrever os bytes no arquivo

```

Classe QuintaEtapa

```

executarEtapa():
    repositorioUsuario = criar uma nova instância
    configurando o alias e a senha do usuário
    Chamar repositorioUsuario.abrir para carregar o
    repositório de chaves do usuário
    chavePrivadaAssinante = pegar chave privada do
    repositorio

```

```
certificadoAssinante = pegar certificado do
repositorio
geradorAssinatura = criar uma nova instância de
GeradorDeAssinatura
Chamar geradorAssinatura.informaAssinante passando o
certificadoAssinante e a chavePrivadaAssinante
assinatura = chamar geradorAssinatura.assinar,
passando o caminho do documento de texto a ser
assinado
Abrir um arquivo para escrita no caminho de saída da
assinatura
Chamar geradorAssinatura.escreveAssinatura, passando
o arquivo e a assinatura
```

Para a realização da assinatura digital, a implementação foi dividida em duas classes já criadas. O primeiro é a classe de serviço “GeradorDeAssinatura”, projetada para ser o motor criptográfico responsável por criar um pacote de assinatura completo, seguindo o padrão CMS. O seu processo interno inicia com a configuração da identidade do assinante, através do método “informaAssinante”. Em seguida, o método “assinar” faz a criação da estrutura da assinatura: ele prepara o documento a ser assinado de forma eficiente, constrói a estrutura de informações do assinante (“SignerInfoGenerator”) e, por fim, utiliza o “CMSSignedDataGenerator” do Bouncy Castle para empacotar o documento original, a assinatura digital e o certificado do assinante numa única entidade.

O segundo componente é a classe “QuintaEtapa”. A sua primeira tarefa é pegar as informações credenciais do utilizador, que foram previamente armazenadas num repositório PKCS#12. Para isso, ela utiliza a classe “RepositorioChaves” para carregar o arquivo .p12 e extrair a chave privada e o certificado do assinante. Com estas credenciais em mãos, ela instancia o “GeradorDeAssinatura”, configura-o com a identidade do utilizador e invoca o método assinar para gerar a assinatura CMS para o documento de texto.

O passo final do fluxo é utilizar o método “escreveAssinatura” para salvar o pacote de assinatura resultante em um arquivo no disco, concluindo a etapa. Após executar o programa,

utilizei o site “lapo.it/asn1js” para conferir a assinatura na estrutura do arquivo .txt e estava tudo certo e assinado, contendo até mesmo os algoritmos de hash e das chaves.

Figura 2.5.1

```

----- INTEGER (127 bit) 128087855099855233032551836648087377254
signatureAlgorithm SignatureAlgorithmIdentifier SEQUENCE (1 elem)
  algorithm OBJECT IDENTIFIER 1.2.840.10045.4.3.2 ecdsaWithSHA256 (ANSI X9.62 ECDSA algorithm with SHA256)
signature SignatureValue OCTET STRING (71 byte) 3045022100F104302EABB428ECC8E71CE2E38F6CBED848A8E9A2B5D5DA19933E485C83...
  SEQUENCE (2 elem)
    INTEGER (256 bit) 1090147964388910211155273182696951354439770659352785791017500904300309...
    INTEGER (252 bit) 5758600628818957274240888478952309319266088628943097300803726649607194...

```

Figura 2.5.2

```

hristian@UFS396197:/mnt/c/Users/Christian/Videos/DesafioFinal/src/main/resources/artefatos/assinaturas$ openssl asn1parse -in assinatura.der -inform DER
0:d=0 hl=2 l=inf cons: SEQUENCE
2:d=1 hl=2 l= 9 prim: OBJECT :pkcs7-signedData
13:d=1 hl=2 l=inf cons: cont [ 0 ]
15:d=2 hl=2 l=inf cons: SEQUENCE
17:d=3 hl=2 l= 1 prim: INTEGER :01
20:d=3 hl=2 l= 13 cons: SET
22:d=4 hl=2 l= 11 cons: SEQUENCE
24:d=5 hl=2 l= 9 prim: OBJECT :sha256
35:d=3 hl=2 l=inf cons: SEQUENCE
37:d=4 hl=2 l= 9 prim: OBJECT :pkcs7-data
48:d=4 hl=2 l=inf cons: cont [ 0 ]
50:d=5 hl=2 l= 37 prim: OCTET STRING :Arthur de Farias Salmoria - 25100792

89:d=5 hl=2 l= 0 prim: EOC
91:d=4 hl=2 l= 0 prim: EOC
93:d=3 hl=2 l=inf cons: cont [ 0 ]
95:d=4 hl=4 l= 361 cons: SEQUENCE
99:d=5 hl=3 l= 204 cons: SEQUENCE
102:d=6 hl=2 l= 3 cons: cont [ 0 ]
104:d=7 hl=2 l= 1 prim: INTEGER :02
107:d=6 hl=2 l= 4 prim: INTEGER :017F01F8
113:d=6 hl=2 l= 10 cons: SEQUENCE
115:d=7 hl=2 l= 8 prim: OBJECT :ecdsa-with-SHA256
125:d=6 hl=2 l= 18 cons: SEQUENCE
127:d=7 hl=2 l= 16 cons: SET
129:d=8 hl=2 l= 14 cons: SEQUENCE
131:d=9 hl=2 l= 3 prim: OBJECT :commonName
136:d=9 hl=2 l= 7 prim: UTF8STRING :AC-RAIZ
145:d=6 hl=2 l= 30 cons: SEQUENCE
147:d=7 hl=2 l= 13 prim: UTCTIME :250713224119Z
162:d=7 hl=2 l= 13 prim: UTCTIME :260713224119Z
177:d=6 hl=2 l= 36 cons: SEQUENCE
179:d=7 hl=2 l= 34 cons: SET
181:d=8 hl=2 l= 32 cons: SEQUENCE
183:d=9 hl=2 l= 3 prim: OBJECT :commonName
188:d=9 hl=2 l= 25 prim: UTF8STRING :Arthur de Farias Salmoria
215:d=6 hl=2 l= 89 cons: SEQUENCE
217:d=7 hl=2 l= 19 cons: SEQUENCE
219:d=8 hl=2 l= 7 prim: OBJECT :id-ecPublicKey
228:d=8 hl=2 l= 8 prim: OBJECT :prime256v1
238:d=7 hl=2 l= 66 prim: BIT STRING
306:d=5 hl=2 l= 10 cons: SEQUENCE
308:d=6 hl=2 l= 8 prim: OBJECT :ecdsa-with-SHA256
318:d=5 hl=3 l= 139 prim: BIT STRING
460:d=4 hl=2 l= 0 prim: EOC
462:d=3 hl=4 l= 283 cons: SET

```

## 2.6 Sexta Etapa

A sexta etapa se resume em verificar uma assinatura, no caso a feita na quinta etapa. A verificação da assinatura digital se resume em decifrar a assinatura utilizando a chave pública (é contraintuitivo esse processo considerando a decifragem pela chave pública, mas seria como uma engenharia reversa) e compará-la com o hash do documento, se eles forem iguais e outros atributos como data de validade estiverem de acordo, a assinatura é válida.

E então passei a fazer um algoritmo em linguagem natural visando a implementação em java:

## Classe VerificadorDeAssinatura

`verificarAssinatura(certificado, assinatura):`

Chamar o método interno `pegaInformacoesAssinatura` para extrair os dados do assinante de dentro da assinatura

Chamar o método interno

`geraVerificadorInformacoesAssinatura` para criar um objeto "verificador" a partir da chave pública contida no certificado

Usar o "verificador" para validar as informações do assinante. Este passo automaticamente recalcula o resumo do documento, decifra a assinatura e compara os resultados

Retornar verdadeiro se a verificação for bem-sucedida, ou falso caso contrário

`geraVerificadorInformacoesAssinatura(certificado):`

Usar um construtor do Bouncy Castle para criar e retornar um objeto verificador, configurado com a chave pública do certificado fornecido

`pegaInformacoesAssinatura(assinatura):`

Acessar a lista de assinantes dentro do objeto de assinatura

Retornar o primeiro (e único) assinante da lista

## Classe SextaEtapa:

`executarEtapa():`

Ler do disco o arquivo de assinatura gerado na etapa anterior e carregá-lo como um objeto `CMSSignedData`

Criar uma nova instância da classe

`RepositorioChaves`, configurada com o alias e a senha do usuário, obtidos da classe `Constantes`

Chamar o método `abrir` do repositório para carregar o arquivo PKCS#12 do usuário

Chamar o método `pegarCertificado` do repositório para obter uma cópia confiável do certificado do usuário  
Criar uma nova instância do `VerificadorDeAssinatura`.  
Chamar o método `verificarAssinatura` do verificador, passando o certificado confiável e a assinatura carregada  
Imprimir na tela uma mensagem clara indicando se a assinatura é "VÁLIDA" ou "INVÁLIDA", com base no resultado retornado

A primeira é a classe de serviço “VerificadorDeAssinatura”, projetada para ser o motor criptográfico responsável por validar a integridade de um pacote de assinatura no padrão CMS. O seu processo interno inicia-se com o método “verificarAssinatura”, que recebe o certificado do suposto assinante e os dados da assinatura. Este método orquestra a validação: ele extrai as informações do assinante (“SignerInformation”) do pacote CMS, prepara um objeto verificador “SignerInformationVerifier” utilizando a chave pública contida no certificado fornecido e, por fim, invoca o método “verify()” do Bouncy Castle. Essa chamada é o passo crucial, pois ela recalcula o hash do documento original, decifra o hash que foi assinado com a chave privada e compara os dois para confirmar a autenticidade e a integridade da assinatura.

O segundo componente é a classe “SextaEtapa”, que atua como a orquestradora de todo o processo de verificação. A sua primeira tarefa é ler o arquivo de assinatura (o pacote CMS) que foi previamente salvo em disco. Em seguida, para obter a chave pública necessária para a verificação, ela acessa o repositório de credenciais do utilizador (o arquivo PKCS#12). Para isso, utiliza a classe “RepositorioChaves” para carregar o arquivo .p12 e extrair o certificado do assinante. Com o certificado em mãos e a assinatura carregada, ela instancia o “VerificadorDeAssinatura” e invoca o método “verificarAssinatura”. O passo final do fluxo é analisar o resultado booleano retornado pelo verificador e informar ao utilizador se a assinatura é Válida e Íntegra ou se é Inválida, concluindo a etapa de verificação.

Figura 2.6.1

```
christian@UFSC398197:/mnt/c/Users/Christian/Videos/DesafioFinal/src/main/resources/artefatos$ openssl cms -verify -in assinaturas/assinatura.der -inform DER -certfile certificados/certificadoUsuario.pem -CAfile certificados/certificadoAcRaiz.pem -out /dev/null  
CMS Verification successful
```

Figura 2.6.2

```
Escolha uma opção: 6
Sexta Etapa: Verificação de assinatura digital
Arquivo de assinatura lido com sucesso de: src/main/resources/artefatos/assinaturas/assinatura.der
Verificando com o certificado de: CN=Arthur de Farias Salmoria
>>> A assinatura é VÁLIDA E ÍNTEGRA. <<<
Sexta Etapa concluída.

Pressione Enter para continuar...
```

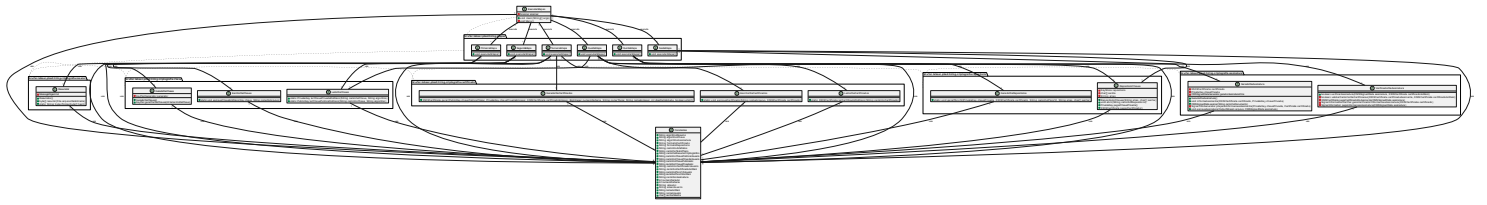
### 3 Conclusão

A resolução do desafio me permitiu aprofundar conhecimentos essenciais, especialmente na aplicação da biblioteca Bouncy Castle em Java. O esforço dedicado foi recompensador, e sinto que minha capacidade técnica evoluiu desde o início do processo.

Destaco a qualidade do desafio proposto, que incentiva a autonomia e o aprendizado contínuo, uma abordagem que acredito ser extremamente válida para vagas voltadas a laboratórios e pesquisa. Essa experiência reforçou meu grande interesse pela área de segurança e, principalmente, pelos projetos desenvolvidos no Laboratório de Segurança em Computação (o LabSEC).

Reitero minha forte vontade de integrar a equipe. Acredito que posso contribuir de forma relevante para os objetivos do laboratório no curto, médio e longo prazo, alinhando meus objetivos a uma trajetória acadêmica de excelência pelo laboratório.

A seguir, apresento o diagrama de classes já citado e feito para ilustrar a estrutura do sistema feita para o processo seletivo. Recomendo aproximar bastante utilizando um zoom, pois ao exportar, o diagrama de uml ficou muito pequeno.



## 4 Referências

BOUNCY CASTLE. Java Documentation. In: BOUNCY CASTLE, [s.d.]. Disponível em: <https://www.bouncycastle.org/documentation/documentation-java/#documentation>.

CRYPTOGRAPHIC MESSAGE SYNTAX. In: WIKIPÉDIA: a enciclopédia livre. São Francisco, CA: Wikimedia Foundation, 2024. Disponível em: [https://en.wikipedia.org/wiki/Cryptographic\\_Message\\_Syntax](https://en.wikipedia.org/wiki/Cryptographic_Message_Syntax).

INTERNET ENGINEERING TASK FORCE (IETF). RFC 7292: PKCS #12: Personal Information Exchange Syntax v1.1. K. Moriarty et al., Jul. 2014. Disponível em: <https://datatracker.ietf.org/doc/rfc7292/>.

INTERNET ENGINEERING TASK FORCE (IETF). RFC 8018: PKCS #5: Password-Based Cryptography Specification Version 2.1. K. Moriarty et al., Jan. 2017. Disponível em: <https://datatracker.ietf.org/doc/html/rfc8018>.

IS SECP256R1 more secure than secp256k1?. In: STACK EXCHANGE, 2015. Disponível em: <https://crypto.stackexchange.com/questions/18965/is-secp256r1-more-secure-than-secp256k1>.

LAPO, L. ASN.1 JavaScript Decoder.: [s.n.], [s.d.]. Disponível em: <https://lapo.it/asn1js/>.

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY (NIST). FIPS PUB 186-5: Digital Signature Standard (DSS). Fev. 2023. Disponível em: <https://csrc.nist.gov/pubs/fips/186-5/final>.

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY (NIST). Special Publication 800-186: Recommendations for Discrete Logarithm-based Cryptography: Elliptic Curve Domain Parameters. Fev. 2023. Disponível em: <https://csrc.nist.gov/pubs/sp/800/186/final>.

PKCS #7. In: WIKIPÉDIA: a enciclopédia livre. São Francisco, CA: Wikimedia Foundation, 2023. Disponível em: [https://en.wikipedia.org/wiki/PKCS\\_7](https://en.wikipedia.org/wiki/PKCS_7).

STALLINGS, William. Cryptography and Network Security: Principles and Practice. 6. ed. Boston: Pearson, 2013.