Relatório – Trabalho Prático 01
REST API com Autenticação Segura e Criptografia

Universidade de Brasília – UnB
Departamento de Ciência da Computação - CIC
Disciplina: Tópicos Avançados em Segurança Computacional – 2025/1
Professora: Lorena Borges
Aluno(s): **Rafael Hamú - 202006448 e Augusto Suffert - XXXXXXXX**

20 de maio de 2025

1 Introdução

Este relatório apresenta o desenvolvimento de uma aplicação cliente-servidor baseada no padrão REST, utilizando autenticação segura via tokens JWT assinados digitalmente, empregando os algoritmos HMAC e RSA. O objetivo é demonstrar a implementação prática dos conceitos de autenticação, integridade e confidencialidade na troca de informações, além de analisar as vantagens, limitações e possíveis vulnerabilidades de cada abordagem.

2 Fundamentação Teórica

2.1 REST API

REST (Representational State Transfer) é um estilo de arquitetura para construção de serviços web. Ele utiliza métodos HTTP (GET, POST, etc.) para operar sobre recursos, sendo amplamente adotado por sua simplicidade, escalabilidade e independência de plataforma.

2.2 JSON Web Token (JWT)

JWT é um padrão aberto para troca segura de informações entre partes. Um token JWT é composto por três partes:

- **Header:** identifica o algoritmo de assinatura.
- **Payload:** contém os dados (claims).
- **Signature:** resultado da assinatura criptográfica, garantindo integridade.

2.3 HMAC

HMAC (Hash-based Message Authentication Code) é um mecanismo que utiliza uma função hash (como SHA-256) e uma chave secreta para gerar códigos de autenticação. É eficiente, mas depende do compartilhamento seguro da chave entre as partes.

2.4 RSA

RSA é um algoritmo de criptografia assimétrica. Utiliza um par de chaves (privada e pública). É adequado para assinatura digital, pois qualquer pessoa pode verificar a autenticidade da assinatura com a chave pública, sem precisar conhecer a chave privada.

2.5 Assinatura RSA: PKCS#1 v1.5 vs PSS

O padrão RSA para assinatura digital possui duas formas principais usadas com JWT:

- **RS256:** RSA com esquema PKCS#1 v1.5, um padrão mais antigo e amplamente suportado.
- **PS256:** RSA com esquema PSS (Probabilistic Signature Scheme), considerado mais seguro por oferecer proteção contra certos ataques de assinatura.

No projeto, a implementação utiliza o algoritmo PS256, devido a sua maior segurança. A professora solicitou a possibilidade de usar ambos, e essa distinção está documentada neste relatório. Em implementações futuras, poderia ser feito um parâmetro para escolher qual usar.

3 Implementação

A solução foi desenvolvida em Python, sem frameworks de API. O projeto contém três principais arquivos:

- **servidor.py**: implementa a API REST, controle de autenticação e proteção dos dados.
- **cliente.py**: simula as ações do cliente (login, requisições protegidas, testes de ataque).
- **gerar_chaves.py**: gera o par de chaves RSA.
- **requirements.txt**: lista as dependências necessárias.

3.1 Cadastro e Armazenamento Seguro de Senhas

As senhas dos usuários são armazenadas no servidor apenas em formato de hash (`bcrypt`), garantindo que a senha original não seja exposta em caso de vazamento.

Listing 1: Armazenamento do hash da senha

```
usuario_db = {
    "usuario1": bcrypt.hashpw("senha123".encode(), bcrypt.gensalt())
}
```

3.2 Autenticação e Emissão do JWT

Quando o cliente envia login e senha, o servidor verifica o hash. Se correto, gera um token JWT contendo:

- O nome do usuário;

- Data/hora de expiração (5 minutos a partir da emissão).

O modo de assinatura é escolhido no início da execução do servidor:

- **HMAC:** JWT assinado com chave secreta.
- **RSA:** JWT assinado com chave privada RSA.

3.3 Acesso Protegido com Validação do JWT

Para acessar o recurso protegido (`/dados`), o cliente deve fornecer o JWT via cabeçalho `Authorization`. O servidor verifica:

- Se a assinatura é válida;
- Se o token não está expirado.

Apenas nessas condições o dado secreto é retornado ao cliente.

3.4 Cliente – Simulação de Testes

O cliente executa diversos cenários:

- Login e obtenção do JWT;
- Acesso normal com token válido;
- Acesso com token modificado (assinatura inválida);
- Acesso sem token;
- Acesso com token expirado (aguarda-se expirar).

4 Envio Seguro das Credenciais

No código, o envio das credenciais ocorre via HTTP em texto claro, sem criptografia do canal, o que não é seguro em ambientes reais. Para garantir a confidencialidade durante a transmissão, é imprescindível utilizar HTTPS (TLS), que criptografa todo o tráfego entre cliente e servidor. Esta limitação foi adotada para fins didáticos e pela simplicidade da implementação.

5 Testes Realizados

Exemplos de saídas dos testes realizados:

5.1 Autenticação com Sucesso

```
[CLIENTE] Token recebido: eyJhbGciOi...
[CLIENTE] Acesso normal (token válido): {"dados": "Segredo revelado!"}
```

5.2 Token Modificado (Assinatura Inválida)

```
[CLIENTE] Acesso com token modificado (assinatura inválida): Token invalido
```

5.3 Acesso Sem Token

```
[CLIENTE] Acesso sem token (não autenticado):
```

5.4 Token Expirado

```
[CLIENTE] Aguarde alguns segundos para o token expirar...
[CLIENTE] Acesso com token expirado: Token expirado
```

6 Comparativo dos Algoritmos

Critério	**HMAC (HS256)**	**RSA (PS256)**
Chave	Secreta e simétrica	Par público/privada
Verificação	Só quem tem a chave secreta	Qualquer um com chave pública
Performance	Mais rápido	Mais lento
Escalabilidade	Difícil escalar	Escalável
Segurança	Chave deve ser secreta	Assinatura não revela chave privada
Aplicação	Sistemas pequenos	Ambientes distribuídos

Tabela 1: Comparativo entre HMAC e RSA

7 Análise de Vulnerabilidades e Segurança

7.1 Potenciais Vulnerabilidades

- **Exposição da chave secreta (HMAC):** Se o segredo for comprometido, qualquer um pode gerar tokens válidos.
- **Replay de tokens:** Um token obtido pode ser reutilizado até expirar.
- **Falta de HTTPS:** As credenciais trafegam em texto puro, suscetível a ataques man-in-the-middle.
- **Força bruta no login:** Sem limitação de tentativas, pode-se tentar múltiplas senhas.
- **Persistência de hash em memória:** Ao reiniciar o servidor, usuários cadastrados são perdidos (no modelo atual).

7.2 Mitigações Adotadas e Recomendadas

- Uso de expiração curta do token (5 minutos) para reduzir o impacto de tokens roubados.
- Validação rigorosa da assinatura para impedir tokens alterados.
- Uso de bibliotecas confiáveis para evitar ataques conhecidos ao JWT.
- Adoção de HTTPS para proteger o tráfego e manter confidencialidade.
- Persistência dos hashes em banco de dados para garantir a disponibilidade dos usuários.

7.3 Análise do Tráfego com Wireshark

Foi feita captura do tráfego HTTP entre cliente e servidor utilizando Wireshark. Note-se

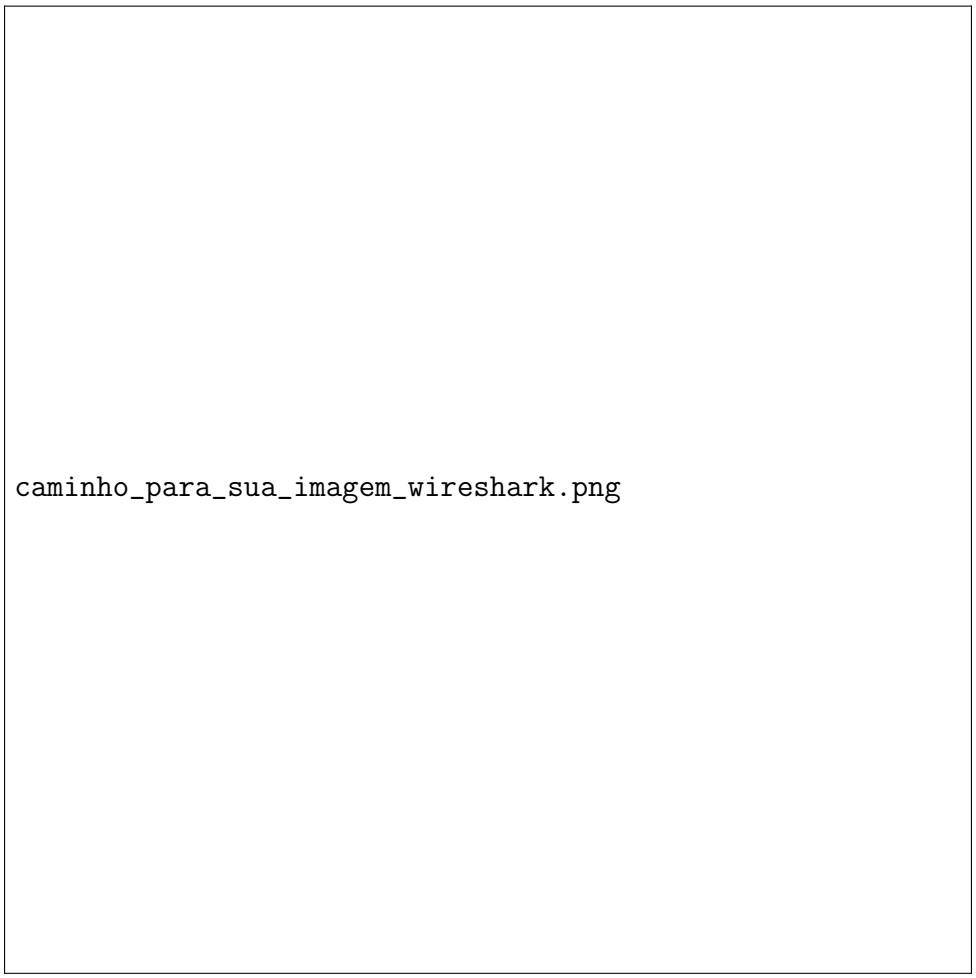

Figura 1: Captura do tráfego HTTP mostrando envio das credenciais em texto claro e resposta com token JWT

que as credenciais trafegam sem criptografia, reforçando a necessidade de HTTPS para produção.

8 Armazenamento das Credenciais no Servidor

No código, os hashes das senhas são mantidos apenas em memória dentro do dicionário `usuario_db`, o que significa que qualquer reinicialização do servidor perde os usuários cadastrados. Para ambientes reais, recomenda-se persistir esses dados em banco ou arquivo criptografado para garantir disponibilidade e segurança dos usuários.

9 Considerações Finais

O projeto atendeu aos requisitos propostos, mostrando o funcionamento prático de autenticação e assinatura digital em APIs REST usando HMAC e RSA, com destaque para a segurança da informação. Foram simulados ataques comuns, comprovando a robustez dos mecanismos implementados. O relatório também evidenciou a importância do uso de HTTPS e da proteção adequada de segredos em sistemas reais.

10 Instruções para Execução

1. Instale as dependências:

   ```
   pip install -r requirements.txt
   ```

2. Gere as chaves RSA:

   ```
   python gerar_chaves.py
   ```

3. Execute o servidor:

   ```
   python servidor.py
   ```

   Escolha o modo de assinatura ao iniciar.

4. Execute o cliente em outro terminal:

   ```
   python cliente.py
   ```