TRABALHO 1

Grupo: Gustavo Mello Tonnera (211055272)

Disciplina: Tópicos Avançados em Computadores

Professora: Lorena de Souza Bezerra Borges

Universidade de Brasília (UnB) – 2025.1

1. INTRODUÇÃO

O estudo e a aplicação de mecanismos de segurança são essenciais para a formação de profissionais capacitados em ciência da computação, especialmente no contexto de sistemas distribuídos e aplicações web. Este relatório descreve o desenvolvimento de um trabalho prático cujo principal objetivo é proporcionar ao estudante a oportunidade de aprender, de forma aplicada, conceitos fundamentais relacionados à autenticação segura, criptografia e assinatura digital.

Proposto no âmbito da disciplina Tópicos Avançados em Segurança Computacional (2025/1), oferecida pelo Departamento de Ciência da Computação da Universidade de Brasília, o trabalho visa explorar a construção de uma aplicação cliente-servidor capaz de realizar comunicação segura utilizando o padrão REST e tokens JWT (JSON Web Token) assinados digitalmente. A atividade busca desenvolver habilidades práticas na implementação de mecanismos de autenticação baseados nos algoritmos criptográficos HMAC (Hash-based Message Authentication Code) e RSA, com foco em suas variantes PKCS#1 v1.5 e RSA-PSS.

A experiência envolve tanto o armazenamento seguro de credenciais quanto a proteção da comunicação entre cliente e servidor, promovendo o entendimento sobre aspectos como confidencialidade, integridade e validade dos dados trocados. Para isso, o aluno é desafiado a implementar dois cenários distintos de autenticação: um utilizando HMAC e outro com assinaturas digitais baseadas em RSA. Cada abordagem é analisada com base em sua segurança, aplicabilidade e resistência a ataques, por meio de testes com tokens inválidos, expirados ou modificados.

As APIs foram implementadas utilizando a linguagem de programação Python e a biblioteca “http.server”, as bibliotecas “ssl”, “bcrypt” e “jwt” foram usadas para implementar as camadas de segurança do servidor e o programa “Wireshark” foi utilizado para analisar a troca de mensagens entre o servidor e o cliente em diferentes situações. As demais seções do relatório apresentam os resultados obtidos no desenvolvimento do trabalho.

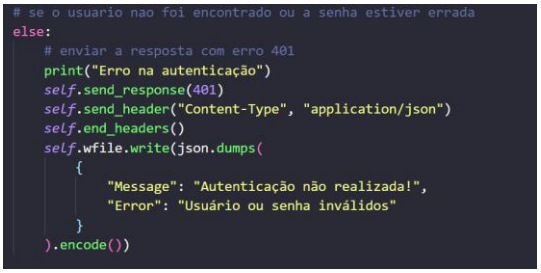
2. AUTENTICAÇÃO

Para implementar a autenticação de usuários, a API desenvolvida possui um método POST, o qual recebe como parâmetros o nome do usuário e a senha do usuário. O método

pode ser acessado pela URL “https://localhost:8000/auth” e, assim que o servidor recebe uma requisição nesse endereço, procura-se o usuário com nome especificado no banco de dados. Caso o usuário seja encontrado, o hash da senha recebida é comparado com a hash da senha armazenado no banco de dados. Se os hashs forem iguais, significa que o usuário que está tentando se autenticar no sistema forneceu credenciais válidas para usar o sistema. Então, um token JWT com tempo de expiração de 10 minutos é gerado e enviado no cabeçalho da resposta do servidor. Dessa forma, o usuário autenticado pode usar o token para acessar os demais métodos da API que necessitam de autenticação.

Caso o usuário que está tentando se autenticar informe um nome de usuário que não esteja cadastrado no banco de dados ou uma senha cujo hash não seja igual com o hash armazenado no banco de dados, a API retorna uma resposta de erro, indicando que não foi possível realizar a autenticação.

Figura 1- Trecho do código que realiza a criação do token JWT



```
# se o usuario nao foi encontrado ou a senha estiver errada
else:
    # enviar a resposta com erro 401
    print("Erro na autenticação")
    self.send_response(401)
    self.send_header("Content-Type", "application/json")
    self.end_headers()
    self.wfile.write(json.dumps(
        {
            "Message": "Autenticação não realizada!",
            "Error": "Usuário ou senha inválidos"
        }
    ).encode())
```

Figura 2 - Trecho do código executado quando o usuário fornece credenciais inválidas

3. API PROTEGIDA COM VERIFICAÇÃO DE ASSINATURA

Além do método de autenticação, foi implementado um método GET o qual pode ser acessado pela URL “https://localhost:8000/contacartao/id_conta/info”, cuja função é enviar ao usuário informações sobre a conta de um usuário. Para acessar essa rota, o usuário precisar ter se autenticado e passar o token de autenticação no cabeçalho da requisição. Caso o usuário forneça um token inválido ou um token expirado, a API retorna uma mensagem informando que o token especificado é inválido ou está expirado, respectivamente. Ademais, foi implementada uma verificação para validar se a conta especificada na URL existe no banco de dados. Caso contrário, a API envia uma mensagem de erro informando que a conta não existe.

```python
# validar o token
try:
    # decodificando o token via RSA
    decoded = jwt.decode(token, self.publicKey, algorithms=["RS256"])
    print("✅ Valid token:", decoded)

# token expirado
except jwt.ExpiredSignatureError:
    print("❌ Token expired")
    self.send_response(401)
    self.send_header("Content-Type", "application/json")
    self.end_headers()
    self.wfile.write(json.dumps({"Message": "Erro de autenticação!", "Error": "Token expirado."}).encode())

# token invalido
except jwt.InvalidTokenError:
    print("❌ Invalid token")
    self.send_response(401)
    self.send_header("Content-Type", "application/json")
    self.end_headers()
    self.wfile.write(json.dumps({"Message": "Erro de autenticação!", "Error": "Token inválido."}).encode())
```

Figura 3 - Trecho do código responsável por validar o token JWT

4. CENÁRIO 1

No primeiro Cenário, foi solicitado que os tokens JWT fossem assinados por meio de um algoritmo HMAC. Na implementação do Cenário 1, foi utilizado o algoritmo SHA256 para assinar os tokens.

```python
# gerando token JWT e assinando por SHA256
token = jwt.encode(payload, self.secret, algorithm="HS256")
```

Figura 4 - Trecho do código que implementa a assinatura do token com SHA256

4.1 TESTES DE VALIDAÇÃO DO TOKEN JWT

Utilizando o programa “Wireshark”, conseguimos analisar a troca de mensagens entre o servidor e um cliente. A aplicação por padrão utiliza HTTPS e é de suma importância que esse protocolo essa usado, o que é explicado na próxima sessão. Entretanto, a fim de realizar os testes solicitados, desativou-se a criptografia dos pacotes para que seja possível de analisar as mensagens trocadas entre o servidor e o cliente.

Com o objetivo de testar a rejeição de token JWT, utilizamos um token JWT inválido para tentar acesso o método GET da API. O token usado está representado na figura 5 e a resposta do servidor na figura 6. Analisando a resposta do servidor, observamos que o servidor não atendeu a requisição, pois o token não é válido.

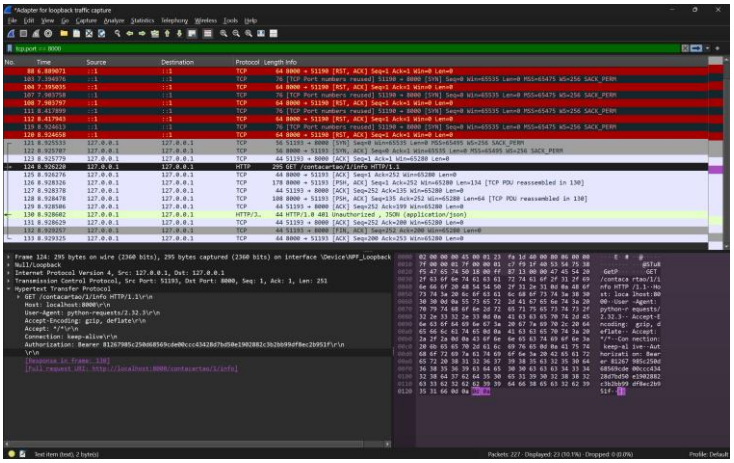


Figura 5 - Token inválido no Wireshark no teste com SHA256

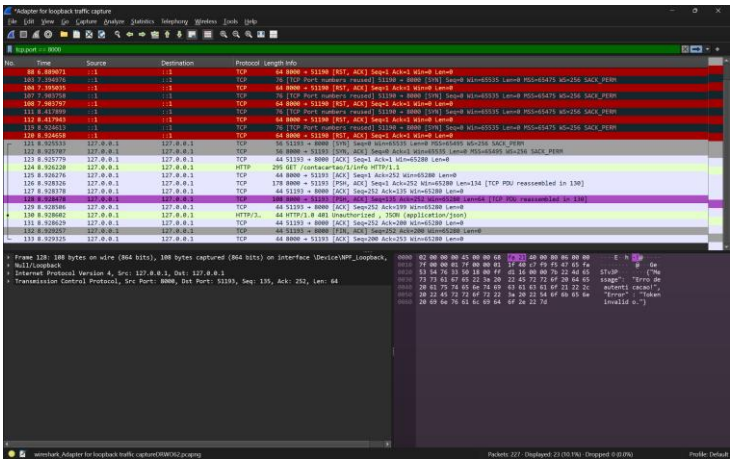


Figura 6 - Resposta do servidor no Wireshark no teste com SHA256

5. CENÁRIO 2

No segundo cenário, foi solicitado que os tokens JWT fossem assinados por meio do algoritmo RSA. Para isso, utilizou-se o programa “openssl” para gerar as chaves pública e privada, as quais são usadas na assinatura e na verificação dos tokens.

```
# gerando token JWT e assinando por RSA
token = jwt.encode(payload, self.privateKey, algorithm="RS256")
```

Figura 7 - Implementação da assinatura do token JWT com RSA

5.1 TESTE DE VALIDAÇÃO DO TOKEN JWT

Utilizando o programa “Wireshark”, conseguimos analisar a troca de mensagens entre o servidor e um cliente. A aplicação por padrão utiliza HTTPS e é de suma importância que esse protocolo essa usado, o que é explicado na próxima sessão. Entretanto, a fim de realizar os testes solicitados, desativou-se a criptografia dos pacotes para que seja possível de analisar as mensagens trocadas entre o servidor e o cliente.

Com o objetivo de testar a rejeição de token JWT, utilizamos um token JWT inválido para tentar acesso o método GET da API. O token usado está representado na figura 8 e a resposta do servidor na figura 9. Analisando a resposta do servidor, observamos que o servidor não atendeu a requisição, pois o token não é válido.

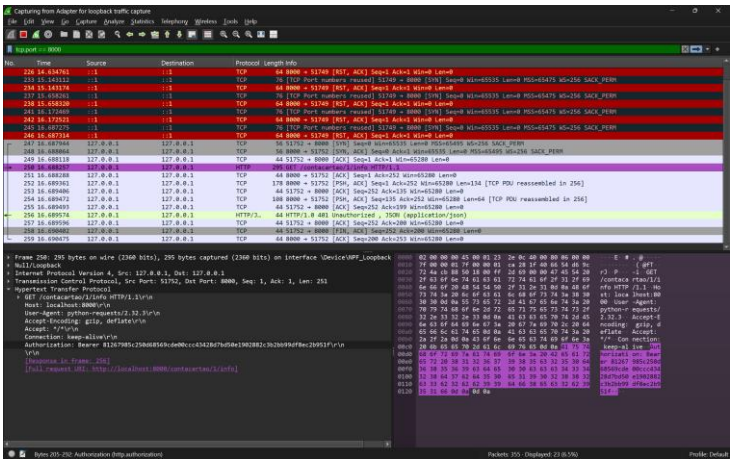


Figura 8 - Token inválido no Wireshark no teste do RSA

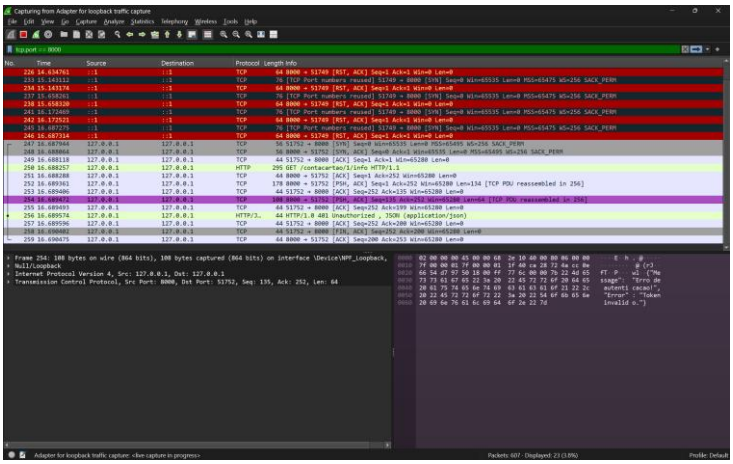


Figura 9 - Resposta do servidor no Wireshark no teste com RSA

6. POSSÍVEIS VULNERABILIDADES

Nesses dois tipos de API, uma vulnerabilidade comum é o tráfego das requisições e respostas sem nenhum tipo de criptografia. Isso permite que atacantes tenham acesso a tokens JWT válidos ao interceptar pacotes na rede. Dessa forma, os atacantes podem extrair os tokens dos pacotes e utilizá-los para acessar os dados da API sem a necessidade de realizar a autenticação. Para contornar esse problema, precisamos criptografar os pacotes que são enviados entre os clientes e o servidor. Isso foi feito por meio da biblioteca “ssl”, fazendo uso do protocolo HTTPS.

```
context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
context.load_cert_chain(certfile='trabalho1/src/keys/cert.pem', keyfile='trabalho1/src/keys/key.pem')
httpd.socket = context.wrap_socket(httpd.socket, server_side=True)
```

Figura 10 - Trecho do código que realiza a criptografia dos pacotes trocados entre o servidor e o cliente

Outra vulnerabilidade muito comum, é o uso do chaves muito curtas ou muito previsíveis no algoritmos de assinatura dos tokens. Isso permite que os atacantes utilizem ataques de força bruta para descobrir a chave e assinar token arbitrários. Assim, os atacantes conseguem acessar os dados da API por meio dos token arbitrários assinados. Para impedir esse tipo de ataque, precisa-se usar chaves secretas longas, aleatórias e de

alta entropia, o que foi feito com ajuda do programa “openssl”, o que consegue gerar chaves com essas características.

Além disso, esse tipo de API sempre está sujeito a ataques relacionados a captura do token JWT na parte do cliente, por meio de uma exposição ou armazenamento indevido do token. Assim, atacantes podem reutilizar o token capturado para realizar requisições na API. Para minimizar, as chances de um atacante conseguir reutilizar um token, devemos determinar um tempo de expiração adequado para os tokens e armazenar os tokens em cookies seguros. Na implementação da API, o tempo de expiração dos tokens é de 10 minutos.

```
# tempo de expiracao do token
expTokenTime = 600

# inicializacao das secret
secret = "ec84e00a057d658c81fa531b727fe14fd1642106018777b86908fa39e34b9639"
```

Figura 11 - Implementação do tempo de expiração do token e do secret usado no SHA256

7. COMO EXECUTAR O CÓDIGO

Para executar o código do trabalho, baixe o repositório completo e execute o arquivo “servidor.py” da maneira explicitada no imagem X e o arquivo “cliente.py” em um outro terminal como explicitado na imagem Y. Ao executar o arquivo “servidor.py”, precisamos passar como parâmetro o algoritmo utilizado na assinatura dos tokens. Para usar a API com o HMAC, basta passar o parâmetro “SHA”. Para usar a API com RSA, basta passar o parâmetro “RSA”. Além disso, os pacotes capturados nos testes com SHA256 e com RSA estão salvos nos arquivos “docs/wireshark/teste_SHA.pcapng” e “docs/wireshark/teste_RSA.pcapng”, respectivamente.

```
PS C:\Users\Gustavo\Documents\repositories\tac-seguranca-tarefas> py trabalho1/src/server.py RSA
```

Figura 12 - Comando para executar o servido com assinatura de tokens com RSA

```
PS C:\Users\Gustavo\Documents\repositories\tac-seguranca-tarefas> py trabalho1/src/client.py
```

Figura 13 - Comando para executar o cliente que interage com a API implementada