Tópicos Avançados em Segurança Computacional - 2025/01

—

Trabalho Prático 01: REST API com Autenticação Segura e Criptografia

27 de maio de 2025

Participantes:
Jean Bueno Karia - 211055290 Taariq de Jesus Padilha - 211038502

Resumo

Este trabalho implementa API REST contendo requisições POST e GET utilizando JWT sobre HMAC ou RSA-PSS. Além disso, analisa o tráfego construído em HTTPS com TLS 1.3.

1 Introdução

A Internet é amplamente utilizada justamente por viabilizar a comunicação e integração de infinitas aplicações em diferentes localidades. O protocolo HTTP como a arquitetura REST (Representational State Transfer) tem contribuições fundamentais para garantir o funcionamento de sistemas distribuído de maneira viável e escalável. Contudo, o atual cenário digital também demanda o uso de algoritmos criptográficos e de autenticação para garantir o tráfego seguro dos dados.

Logo, este trabalho tem como objetivo principal implementar uma aplicação cliente-servidor capaz de realizar a troca segura de informações de forma autenticada via padrão REST e tokens JWT (JSON Web Tokens). Além disto, construir dois cenários com diferentes esquemas de autenticação, isto é, utilizando o HMAC e outro RSA-PSS.

2 TLS (*Transport Layer Security*)

Conforme foi introduzido, garantir a segurança das informações transmitidas pela internet tornou-se essencial. O Transport Layer Security (TLS) é um protocolo aprimorado do SSL (*Security Socket Layer* e atualmente considerado como principal protocolo de proteção para comunicações. Atuando como uma camada entre a aplicação e a rede do da pilha TCP/IP, o TLS assegura que os dados trocados entre clientes e servidores sejam criptografados, íntegros e autênticos, impedindo que informações sensíveis sejam interceptadas ou manipuladas por terceiros. Este protocolo é amplamente utilizado na WEB (navegadores, e-mails, serviços de mensagens e APIs) sendo a camada que proporciona interação entre usuários e sistemas de forma segura e confiável.

2.1 *Handsahke*

O TLS é composto de um processo inicial, *handshake*, responsável por estebelecer a conexão. Basicamente o cliente e o servidor trocam informações para estabelecer uma comunicação segura. Durante esse processo, eles negociam a versão do protocolo TLS, escolhem algoritmos criptográficos, trocam chaves e autenticam identidades (quando necessário). Cada versão do

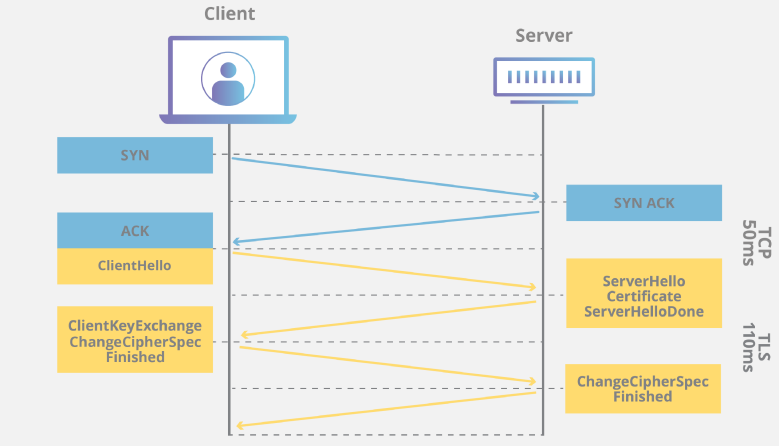

Figura 1: Etapas do *Handshake* TLS.

TLS (como TLS 1.0, 1.2 ou 1.3) define como esse *handshake* ocorre, trazendo melhorias de segurança e eficiência particulares.

Por padrão, o protocolo é composto da chave sessão compartilhada (simétrica) responsável pela confidencialidade dos dados. Já a integridade é satisfeita com o uso do HMAC (*Hash-based Message Authentication Code*) om SHA-256. Ademais, é englobado o certificado digital responsável por autenticar se aquele servidor é confiável e verificado por uma CA (*Certificate Authority*). O esquema de assinatura digital é baseado na criptografia assimétrica e verificada através da cháve publica da CA.

A seguir são listadas as etapas presentes no processo de *handshake* TLS. Por exemplo, a versão TLS 1.3 é a mais atual e implementa o *handshake* simplificado, tornando-o mais rápido e seguro ao remover algoritmos vulneráveis e reduzir o número de etapas.

- **Troca de mensagens:** O cliente envia uma mensagem "olá" para o servidor, informando a versão TLS e os conjuntos de cifras que ele suporta.
- **Resposta do servidor:** O servidor responde com sua própria mensagem de "olá", incluindo seu certificado SSL (para autenticação), o conjunto de cifras escolhido e um valor aleatório gerado pelo servidor ("server random").
- **Autenticação e Verificação:** O cliente verifica o certificado SSL do servidor, confirmando sua identidade por meio da autoridade certificadora, o que garante que o servidor é legítimo.
- **Geração do Segredo Pré-Mestre:** O cliente gera um segredo pré-mestre, criptografado com a chave pública do servidor (obtida a partir do certificado SSL), e o envia ao servidor.
- **Descriptografia e Criação das Chaves de Sessão:** O servidor usa sua chave privada para descriptografar o segredo pré-mestre. Com isso, tanto o cliente quanto o servidor geram as mesmas chaves de sessão, utilizando o client random, o server random e o segredo pré-mestre.
- **Conclusão do Handshake:** O cliente envia uma mensagem "finalizado", criptografada com a chave de sessão. O servidor responde com sua própria mensagem "finalizado", também criptografada, sinalizando que a comunicação segura foi estabelecida.
- **Comunicação Segura:** Com a troca das chaves de sessão, a comunicação passa a ser criptografada de forma simétrica, garantindo a segurança na transmissão dos dados a partir deste ponto.

3 HTTP e HTTPS

O *Hypertext Transfer Protocol* é o protocolo utilizado para sistematizar a transferência inicialmente de texto na internet. Ele define como as mensagens são formatadas e transmitidas, além

de como os navegadores devem responder a comandos. No entanto, o HTTP não oferece criptografia, o que significa que as informações enviadas — como senhas e dados pessoais — podem ser interceptadas e lidas por terceiros. A fim de agregar securança para o meio, criou-se uma versão aperfeiçoada: HTTPS *Secure*. O HTTPS é amplamente utilizado, e seu funcionamento é baseado na combinação do HTTP legado com o TLS.

4 Implementação

Para a implementação da API REST foi utilizado Python3 como linguagem junto das seguintes bibliotecas: **request**, **http.server**, **ssl** e **json**. Além disso o projeto utilizou TLS v1.3 proporcionando *handshake* simplista, 0-RTT e a ausência de algoritmos inseguros - RSA estático Hash MD5 - deixando disponível apenas algoritmos de criptografia moderna, como AES-GCM e ChaCha20-Poly1305.

Para o servidor, inicialmente foi definido socket com porta 4443 e host (localhost) para tornar-se endereçável e assim ser possível estabelecer conexão com o mesmo. Em seguida, um servidor HTTP é inicializado utilizando a biblioteca http.server, tendo como rota o endereço previamente definido e uma página HTML simples como resposta ao método GET. Caso a requisição seja autenticada via JWT válido, o servidor retorna uma mensagem.

O programa do servidor HTTP basicamente funciona criando um contexto de criptografia para gerenciar os protocolos SSL/TLS utilizando token JWT baseado em HMAC (cenário 1) ou RSA-PSS (cenário 2). O parâmetro ssl.PROTOCOL_TLS_SERVER indica que será utilizado um servidor TLS com a versão mais recente disponível. Na sequência, o certificado digital e a chave privada são carregados no contexto de criptografia, permitindo que o servidor comprove sua identidade aos clientes. Para implementar uma conexão mútua (autenticação de mão dupla), exige-se que o cliente também envie um certificado válido para acessar o servidor. Ao final, o contexto TLS é aplicado ao socket do servidor HTTP, convertendo-o em um socket seguro, capaz de realizar conexões criptografadas utilizando TLS.

O programa cliente foi responsável somente para automatizar o processo de requisições sem a necessidade de interação com usuário. Primeiramente, definiu-se a URL do servidor de destino e, em seguida, é enviada uma requisição por meio da biblioteca requests. Essa requisição inclui a URL, o certificado do cliente — composto pelo certificado assinado e sua chave privada, que comprova a posse do certificado — e a verificação da Autoridade Certificadora (CA), responsável por validar a identidade do servidor. Como resultado, o cliente recebe uma resposta enviada pelo servidor, podendo ser a página inicial (index), login ou o resultado provindo de autenticação JWT.

4.1 OpenSSL

O OpenSSL é uma biblioteca de código aberto amplamente utilizada para implementar segurança em comunicações digitais por meio dos protocolos SSL (*Secure Sockets Layer*) e TLS. Ela trabalha com chaves criptográficas, certificados digitais e protocolos de segurança para proteger a comunicação entre sistemas. Para este projeto, a biblioteca OpenSSL foi utilizada para gerar e gerenciar certificados SSL/TLS, garantindo uma comunicação segura entre o cliente e o servidor. Para isso, utilizamos os seguintes comandos via terminal:

Script 1: Geração de Chaves e certificados via openSSL.

```
# Gerar uma nova chave privada da CA
openssl genrsa -out myCA.key 2048

# Gerar um certificado da CA
openssl req -x509 -new -nodes -key myCA.key -sha256 -days 365 -out myCA.pem

# Gerar uma nova chave privada para o servidor
openssl genrsa -out server.key 2048

#Criar uma nova solicitacaoo de certificado
openssl req -new -key server.key -out server.csr -subj "/CN=localhost"

# Assinar o certificado com a CA
```

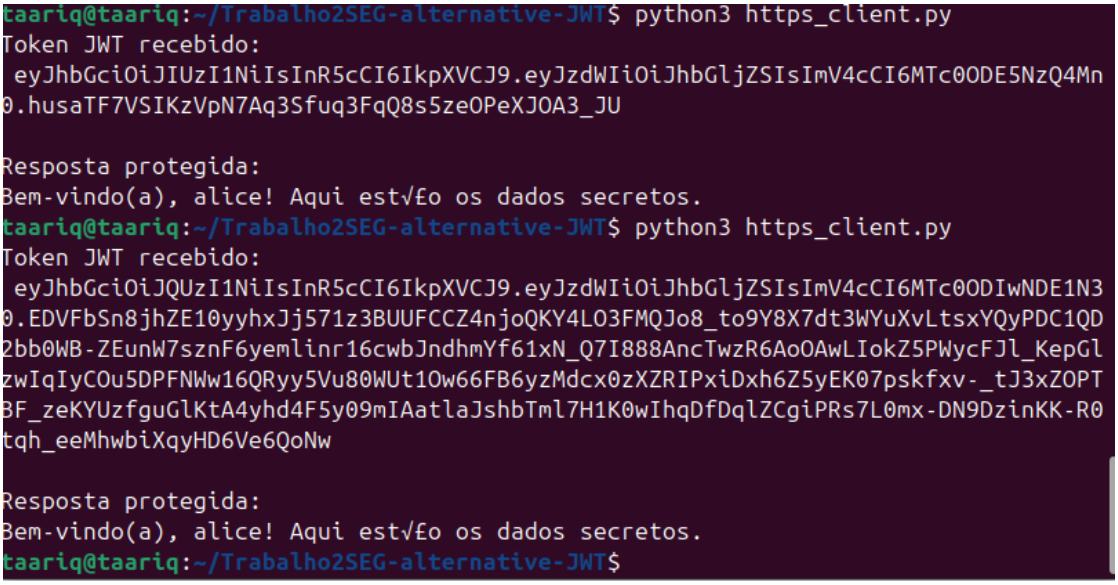

Figura 2: Saída do programa python https_client.

```
openssl x509 -req -in server.csr -CA myCA.pem -CAkey myCA.key -CAcreateserial -out

# Gerar uma nova chave privada para o cliente
openssl genrsa -out client.key 2048

# Criar uma nova solicitacao de certificado
openssl req -new -key client.key -out client.csr -subj "/CN=ClienteHTTPS"

# Assinar o certificado com a CA
openssl x509 -req -in client.csr -CA myCA.pem -CAkey myCA.key -CAcreateserial -out

#Gerar par de chaves RSA
openssl genpkey -algorithm RSA -out keys/private.pem -pkeyopt rsa_keygen_bits:2048
openssl rsa -in keys/private.pem -pubout -out keys/public.pem
```

5 Análise

Antes de analisar cada cenário, é inprescindível a observância do fluxo completo. A figura 6 apresenta as cpaturas de interações solicitadas pelo programa do cliente. É importante salientar a presença do famoso three-way handshake do TCP antes do fluxo HTTPS. Observando A saída do programa cliente autenticação tanto para HMAC quanto RSA-PSS é apresentada na figura 2. Importante salientar seção tratará observações mais granulares nas sub-seções a seguir.

5.1 Funcionamento de HTTPS

Para o contexto do HTTPS, é foi anexada 3 capturas - figuras 3, 4 e 5 - detalhando o conteúdo das requisições GET, POST e OK no cenário 1 com o objetivo de apresentar a estrutura do datagrama:

5.2 Funcionamento de TLS

é necessário compreender como o protocolo TLS foi utilizado no projeto.

Este trecho descreverá como o TLS foi utilizado neste projeto. Na figura 7, observa-se que antes mesmo de utilizar TLS para o HTTPS, foi feito o handshake TLS junto da camada de registro (Record Layer). As figuras 8 e 10 detalham os respectivos pacotes de "Hello"e "Finished

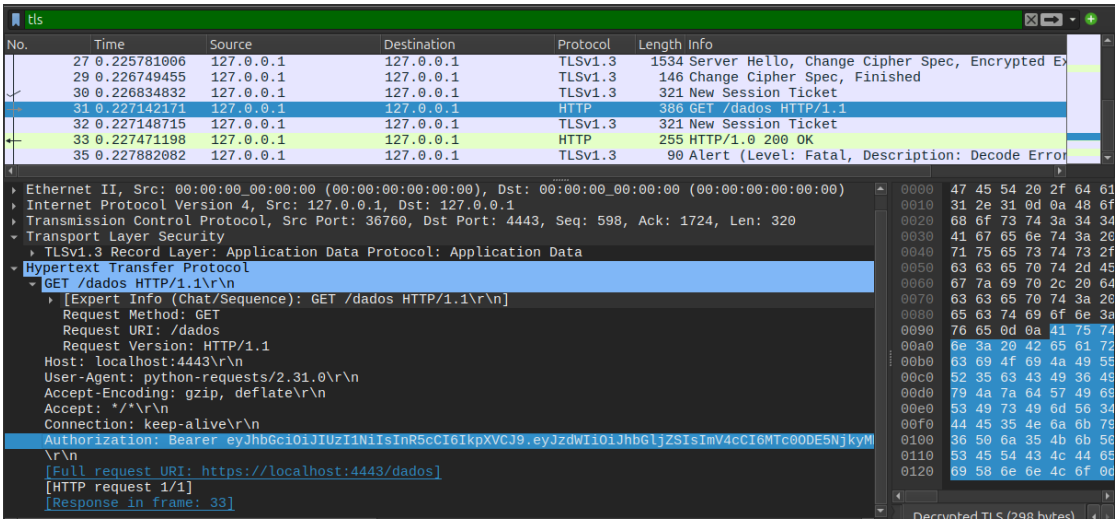

Figura 3: GET request capturado pelo Wireshark.

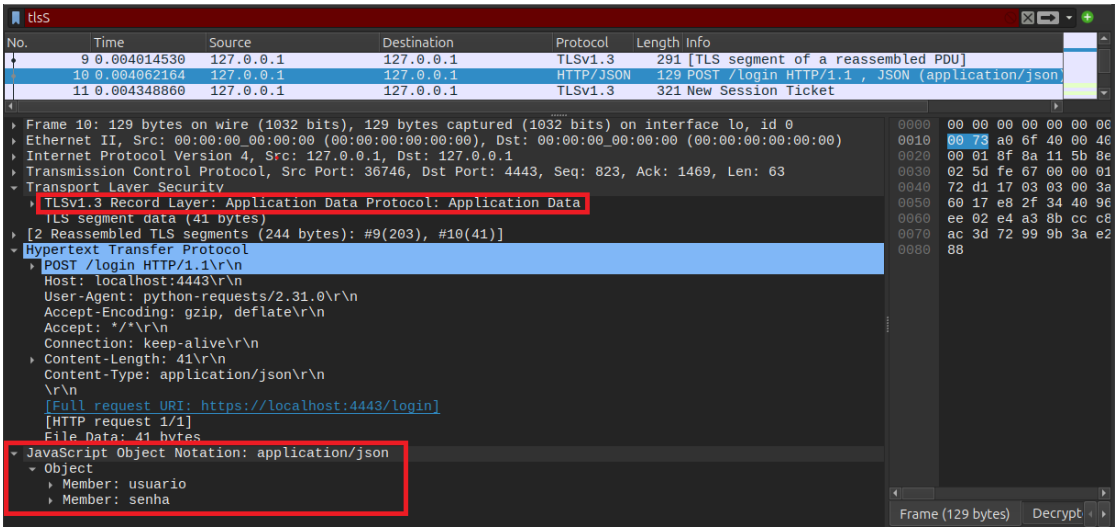

Figura 4: POST request capturado pelo Wireshark.

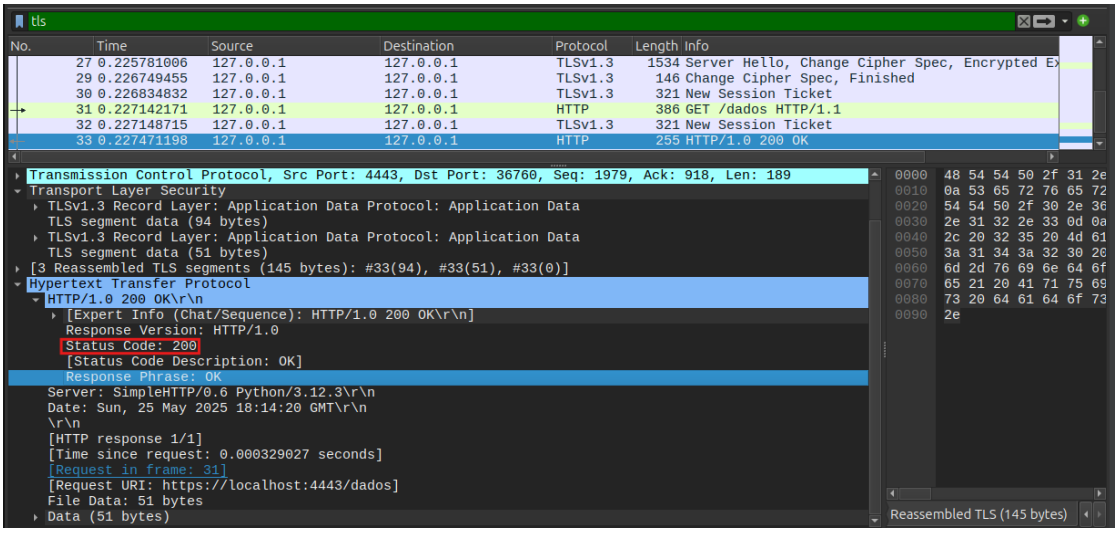

Figura 5: OK response capturado pelo Wireshark.

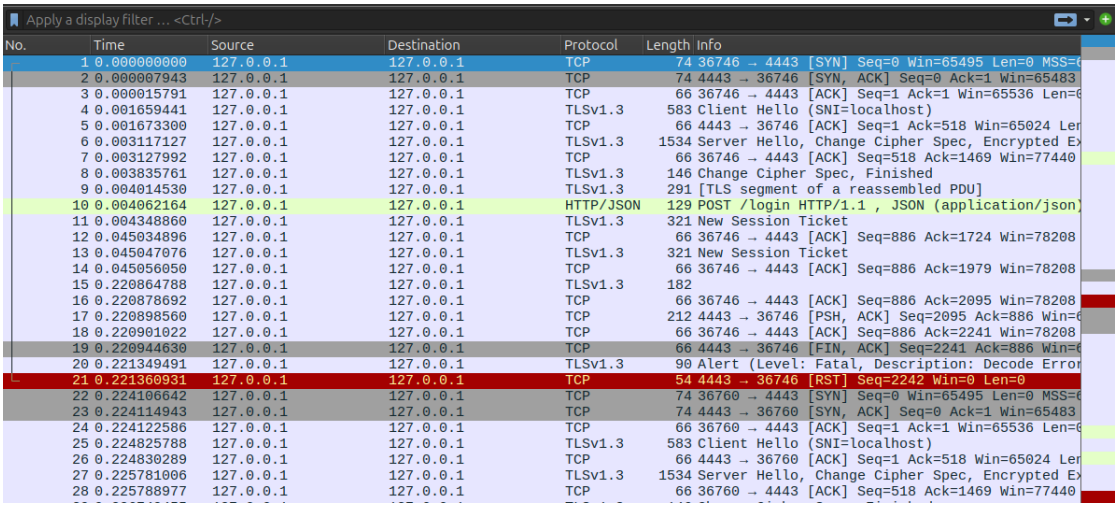

Figura 6: Pacotes capturados pelo Wireshark.

Handshake"enviados pelo cliente. Ademais que o algoritmo RSA foi a criptografia utilizada não somente pelo TLS, mas pelo JWT baseado em RSA-PSS.

5.3 Cenário 1 - HMAC

Considerando que a técnica utilizada no primeiro cenário para autenticar o token foi HMAC,o código Python do servidor HTTPS foi iniciado em ambiente local através da porta 4443 conforme a figura 11. Acessando o nevegador, é carrregada página com campos de login e senha para o usuário logar. Tendo o usuário inserido login/senha corretos, o cliente envia uma requisição GET para o servidor. A figura 12 registra o resultado em tela, retornando o valor de token gerado pelo servidor com timestamp de 1 minuto.

Observando especificamente para o token, percebe-se que o mesmo é formado somente pelos campos de **header**, **payload**. Pois o algoritmo HMAC tem uma construção baseada somente no HASH da mensagem para garantir autenticidade. A figura 13 apresenta os valores dos respectivos campos, como usuário e senha utilizados na verificação inicial.

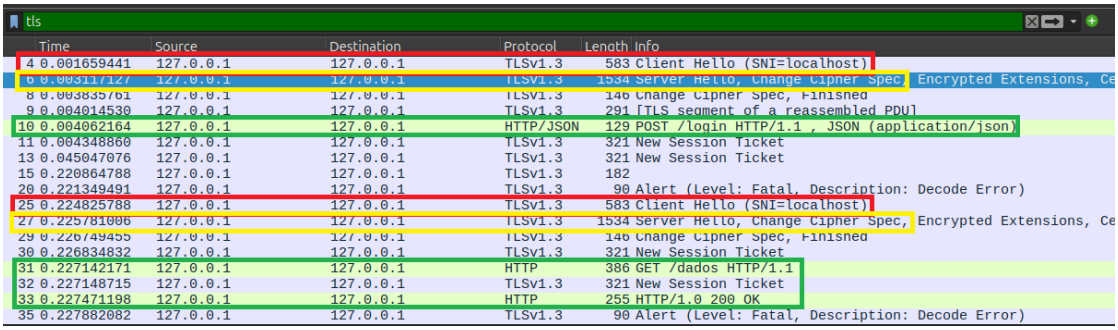

Figura 7: Pacotes TLS capturados pelo Wireshark.

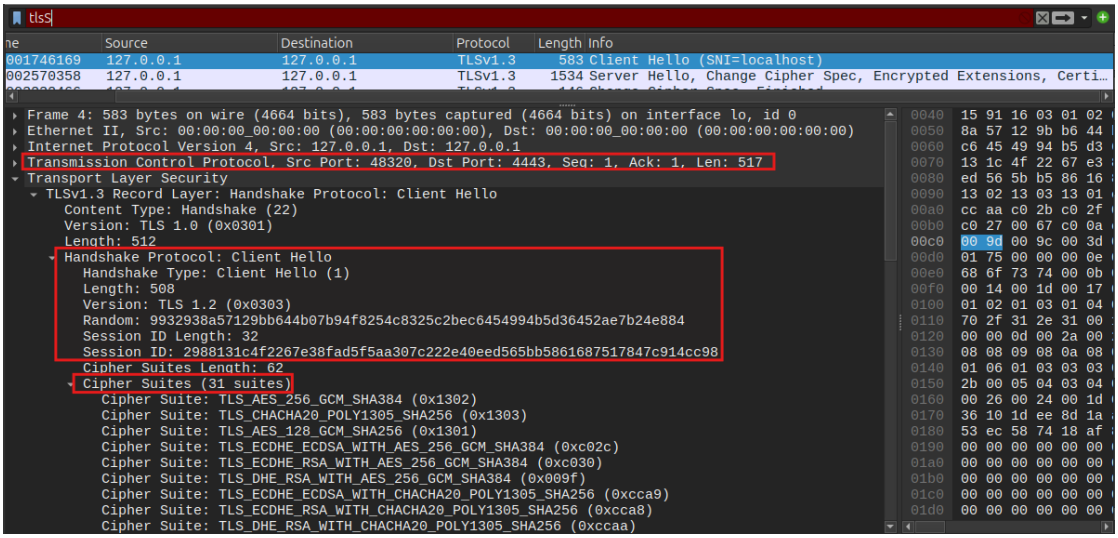

Figura 8: Pacote Client Hello capturado pelo Wireshark.

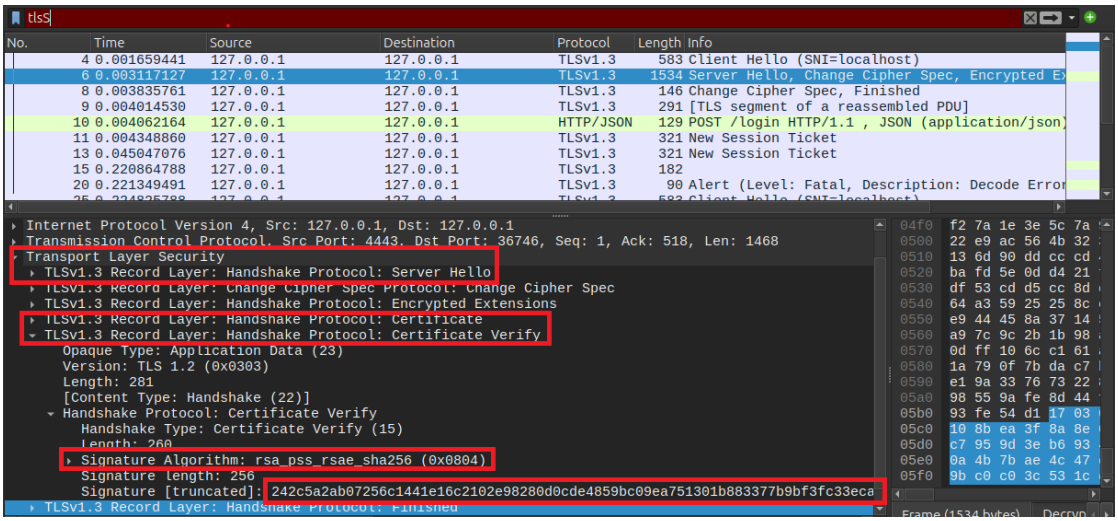

Figura 9: Pacote Server Hello capturado pelo Wireshark.

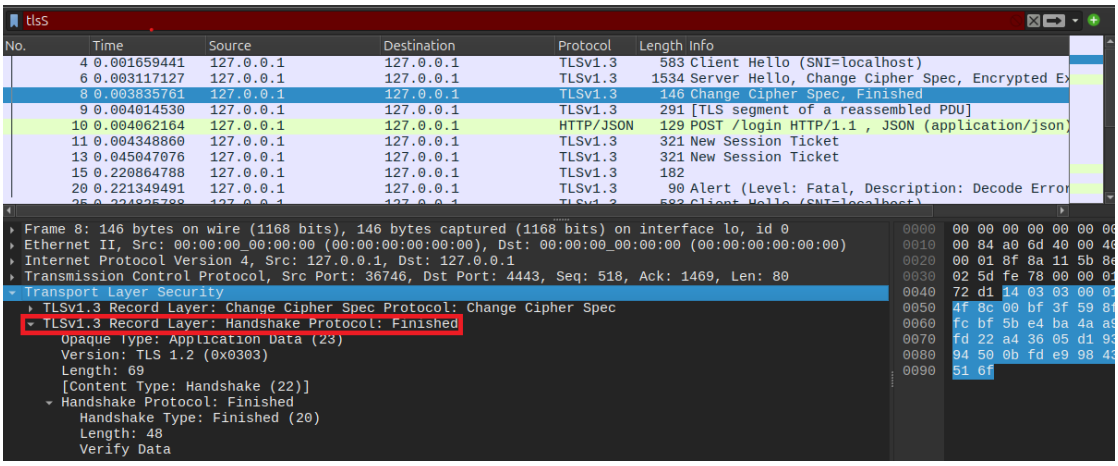

Figura 10: Pacote Finished handshake capturado pelo Wireshark.

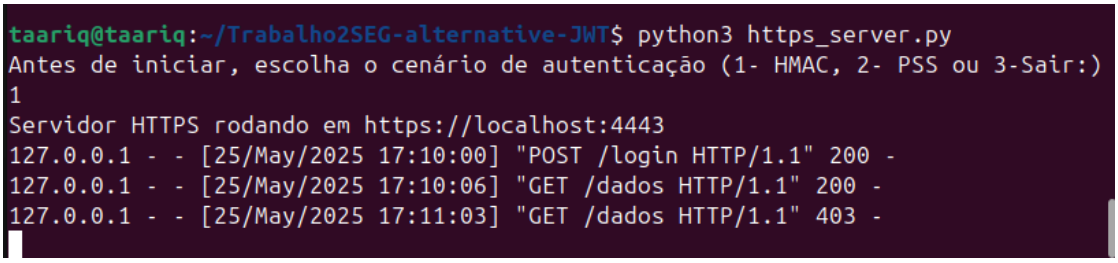

Figura 11: Funcionamento de servidor HTTPS baseado em HMAC.

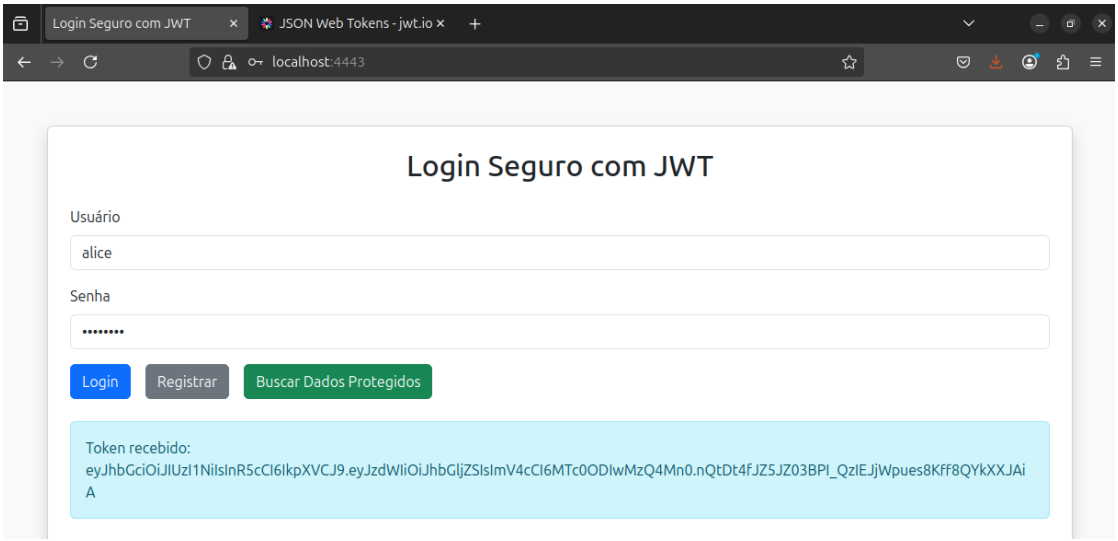

Figura 12: Captura de resposta para requisição GET para HMAC.

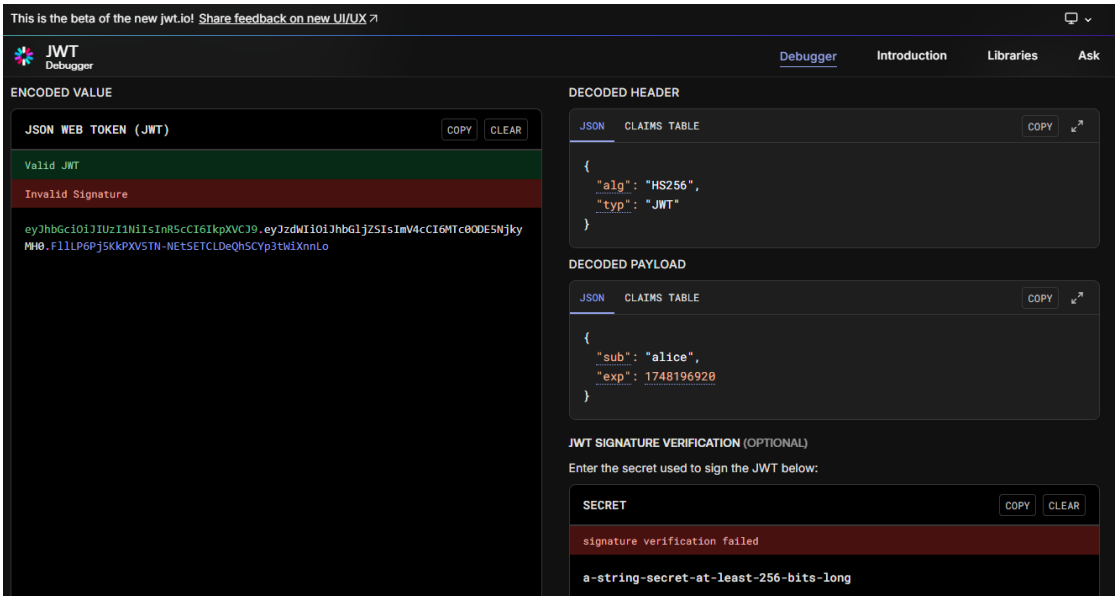

Figura 13: Dados contidos em token JWT-HMAC.

```
taariq@taariq:~/Trabalho2SEG-alternative-JWT$ python3 https_server.py
Antes de iniciar, escolha o cenário de autenticação (1- HMAC, 2- PSS ou 3-Sair:)
2
Servidor HTTPS rodando em https://localhost:4443
127.0.0.1 - - [25/May/2025 17:11:59] "POST /login HTTP/1.1" 200 -
127.0.0.1 - - [25/May/2025 17:12:12] "GET /dados HTTP/1.1" 200 -
127.0.0.1 - - [25/May/2025 17:12:51] "GET /dados HTTP/1.1" 200 -
127.0.0.1 - - [25/May/2025 17:12:54] "GET /dados HTTP/1.1" 200 -
127.0.0.1 - - [25/May/2025 17:13:03] "GET /dados HTTP/1.1" 403 -
```

Figura 14: Funcionamento de servidor HTTPS baseado em RSA-PSS.

5.4 Cenário 2 - RSA-PSS

Assim como foram apresentadas evidências referentes ao funcionamento de cliente/servidor no caso do HMAC, as figuras a seguir (14 e 15) apresentam resultados obtidos pela autenticação RSA-PSS.

No contexto do token JWT, a figura 16 apresenta os valores associado a cada atributo presente no token coletado da sessão. Em comparação com token HMAC, percebe-se que agora o token contém o campo de assinatura. Ou seja, se o usuário possuir a chave pública, é possível verificar a veracidade do token através de sua assinatura. Voltando para a figura 14, pode-se mencionar que a última requisição presente foi frustada. Pois o token recebido do servidor já havia sido expirado, tornando inválido.

6 Conclusão

Pode-se concluir que o trabalho implementou todos os cenários de forma satisfatória. Pois foi possível registrar requisições GET, POST e OK como também os tokens JWT. Importante salientar que com o objetivo de se reduzir a complexidade do projeto foi escolhido não utilizar um banco de dados apropriado. Assim, as senhas relacionadas aos usuários cadastrados foram armazenadas em formato json com hash das senhas, apenas para implementar a funcionalidade de persistência de senhas e usuários.

Em relação ao JSON Web Tokens para autenticação, a escolha do algoritmo de assinatura

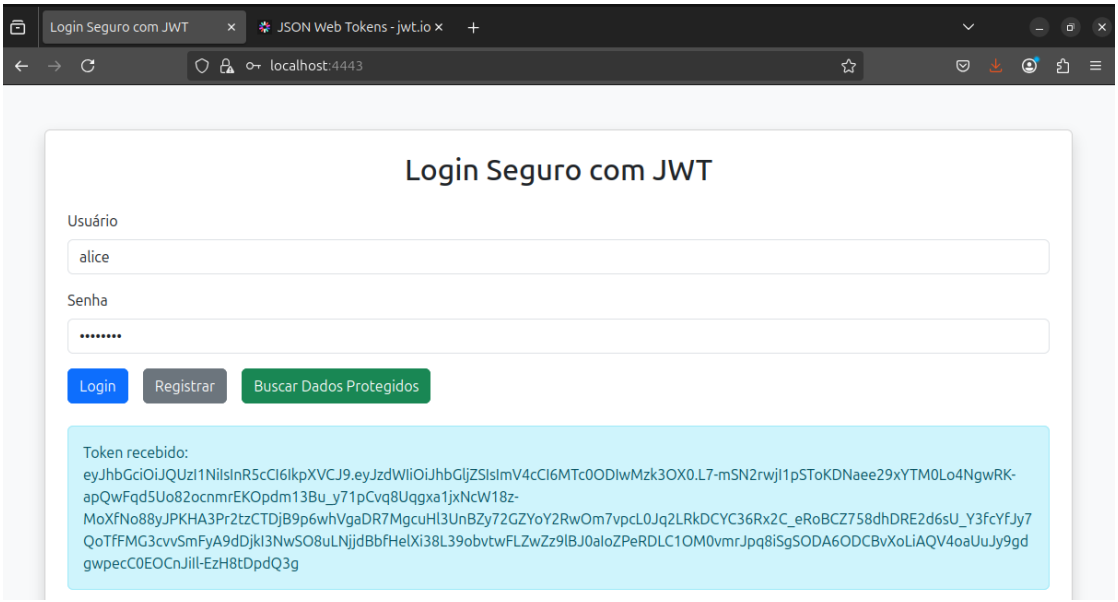

Figura 15: Captura de resposta para requisição GET para RSA-PSS.

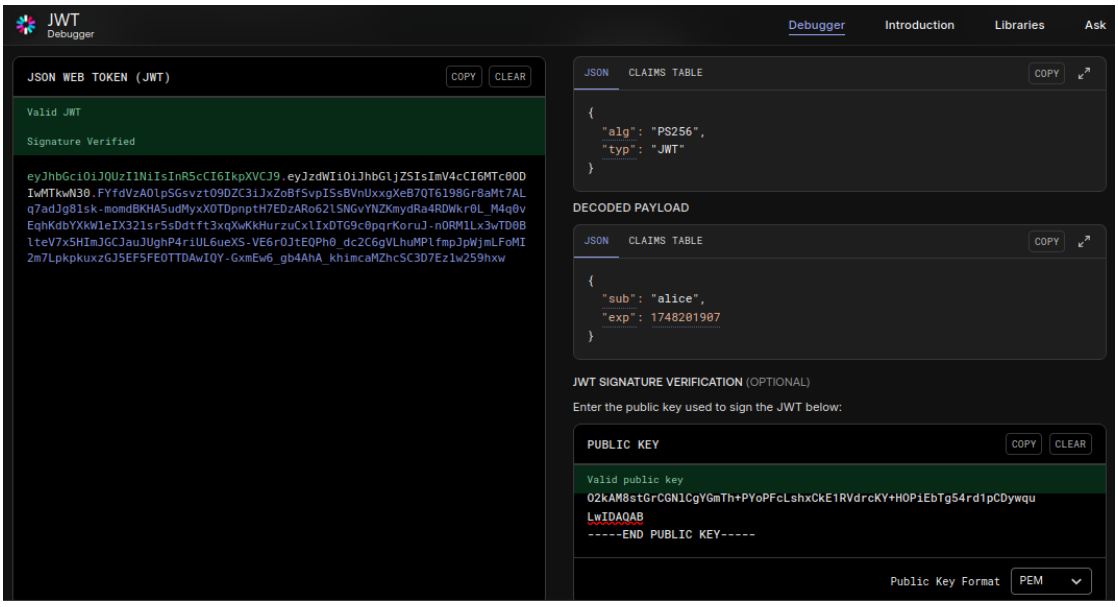

Figura 16: Dados contidos em token JWT-PSS.

tem impacto direto na segurança e na arquitetura da aplicação. Uma abordagem comum é o uso de HMAC (como o HS256), que utiliza criptografia simétrica. Essa simplicidade torna a implementação mais leve e eficiente, interessante para aplicações menores ou centralizadas. No entanto, o modelo simétrico impõe desafios importantes: a chave secreta precisa ser compartilhada com todos os serviços que validam tokens, aumentando a superfície de vulnerabilidade. Caso essa chave seja fraca ou mal gerenciadas, o sistema torna-se totalmente vulnerável a ataques de força bruta. Outra ponto é a flexibilidade existente na validação do cabeçalho do JWT. Caso mal implementada, pode permitir que o algoritmo seja alterado (por exemplo, de HS256 para "none"), resultando em bypass da verificação de assinatura. Outra vulnerabilidade existe é para o ataque *man-in-the-middle.*

Por outro lado, o uso de RSA-PSS engloba assinatura digital com padding probabilístico, trazendo benefícios significativos para segurança do sistema. Como trata-se de criptografia assimétrica, o servidor mantém a chave privada para assinar os tokens, enquanto qualquer serviço pode verificar a autenticidade utilizando apenas a chave pública. Isso elimina a necessidade de compartilhar segredos entre serviços e facilita o escalonamento da aplicação. O uso do esquema PSS, em vez de RSA tradicional com padding determinístico (PKCS#1 v1.5), adiciona robustez contra ataques criptográficos modernos, como ataques de reuso ou pré-imagem. Contudo, o RSA-PSS também tem desempenho é inferior ao HMAC, especialmente em ambientes de alta carga. Pois operações de chave pública e privada são computacionalmente mais caras. A gestão de chaves também exige atenção — embora a chave pública possa ser exposta sem risco, a chave privada deve ser bem protegida, e o uso de um sistema de gerenciamento de chaves (KMS) pode ser necessário.

Em resumo, o uso de HMAC com JWT é simples e eficiente, mas apresenta riscos significativos quando aplicado em ambientes distribuídos ou sem boas práticas de segurança. Já a adoção de RSA-PSS eleva o nível de segurança e facilita a separação entre quem assina e quem valida tokens, tornando-se uma escolha mais apropriada para sistemas distribuídos ou de maior criticidade, mesmo com o custo adicional em termos de desempenho e complexidade operacional.

Referências

[1] JWT Debugger:
`jwt.io`

[2] Como Lidar com JWTs em Python:
`https://auth0.com/blog/pt-how-to-handle-jwt-in-python/#Concluindo`