Trabalho de Implementação 2 – HTTPS

Giulia Moura Ferreira, 20/00018795

[1]Dep. Ciência da Computação – Universidade de Brasília (UnB)
CIC0201 - Segurança Computacional

giulia.ferreira@aluno.unb.br

1. Introdução

A segurança da comunicação na internet é essencial para garantir a privacidade, a autenticidade e a integridade dos dados transmitidos entre clientes e servidores. O protocolo HTTPS, baseado na camada TLS (Transport Layer Security), proporciona uma conexão segura por meio da criptografia e autenticação dos dados trafegados. Este relatório apresenta a implementação de um servidor e cliente HTTPS, explicando a geração e utilização de certificados SSL/TLS, bem como a análise do tráfego de rede utilizando o Wireshark.

O repositório pode ser acessado através desse link, os arquivos de certificado e chave privadas não foram incluídos no repositório por questões de segurança.

2. Objetivos

O principal objetivo desta implementação é demonstrar o funcionamento do protocolo HTTPS na prática, abordando a geração de certificados SSL/TLS, a configuração de um servidor seguro e a comunicação com um cliente HTTPS. Além disso, busca-se analisar e validar a segurança da comunicação por meio da captura de pacotes com o Wireshark, verificando a proteção dos dados transmitidos.

3. Pesquisa e detalhamento dos protocolos de segurança

O **SSL** (Secure Sockets Layer) e seu sucessor, o **TLS** (Transport Layer Security), são protocolos de segurança essenciais para garantir a privacidade, a autenticidade e a integridade das comunicações na internet. O principal objetivo desses protocolos é proteger dados transmitidos entre clientes e servidores, impedindo que terceiros não autorizados interceptem ou alterem as informações. O **SSL** foi o primeiro protocolo desenvolvido para essa finalidade, mas, devido a vulnerabilidades identificadas ao longo do tempo [4], foi substituído pelo **TLS**, que trouxe melhorias significativas em segurança e desempenho. [3]

O **TLS** é amplamente utilizado para proteger diferentes tipos de comunicações, como navegação na web (HTTPS), e-mails (SMTP, IMAP e POP3 com TLS), chamadas de voz sobre IP (VoIP) e mensagens instantâneas. Ele funciona adicionando uma camada de criptografia à comunicação entre duas partes, garantindo que os dados transmitidos não possam ser lidos ou modificados por terceiros. Esse processo de proteção ocorre por meio da criptografia, autenticação e verificação de integridade. [1]

A implementação do **TLS** segue um conjunto de etapas para garantir uma comunicação segura. O processo começa com o handshake **TLS**, no qual cliente e servidor negociam parâmetros de segurança antes da troca de dados. Esse handshake envolve as seguintes fases:

1. **Negociação da versão do protocolo TLS**: O cliente informa ao servidor quais versões do TLS suporta, e o servidor seleciona a versão mais recente que ambos compartilham.
2. **Escolha do conjunto de algoritmos criptográficos (Cipher Suite)**: O cliente e o servidor concordam sobre quais algoritmos serão usados para criptografia, autenticação e verificação de integridade.
3. **Autenticação do servidor**: O servidor envia seu certificado digital, emitido por uma Autoridade de Certificação (CA), para que o cliente possa verificar sua identidade.
4. **Troca de chaves**: O cliente gera um segredo compartilhado, que será usado para derivar uma chave de sessão simétrica. Esse segredo é protegido por criptografia assimétrica durante a troca.
5. **Criação de uma chave de sessão simétrica**: Depois que o segredo compartilhado é estabelecido, cliente e servidor derivam uma chave de sessão que será usada para criptografar os dados da comunicação.

Os algoritmos envolvidos na segurança do **TLS** incluem técnicas de criptografia assimétrica, criptografia simétrica e funções hash para garantir a integridade dos dados. A criptografia assimétrica (como RSA, ECDSA e Diffie-Hellman) é usada durante o handshake para estabelecer a chave de sessão, enquanto a criptografia simétrica (como AES-GCM e ChaCha20) protege os dados após a negociação. Para garantir a integridade das mensagens, o TLS usa funções de hash criptográficas, como SHA-256, que permitem verificar se os dados não foram alterados durante a transmissão. [2]

O protocolo **TLS** evoluiu a partir do **SSL**, que teve três versões principais antes de ser considerado obsoleto. O **SSL** 1.0 nunca foi lançado publicamente, pois continha falhas críticas de segurança. O SSL 2.0 foi lançado em 1995, mas rapidamente substituído pelo **SSL** 3.0 em 1996 devido a vulnerabilidades graves. Em 1999, a Internet Engineering Task Force (IETF) desenvolveu o **TLS 1.0**, como um sucessor direto do **SSL 3.0**, trazendo melhorias na segurança e no processo de handshake. [3]

As versões subsequentes do **TLS** continuaram a aprimorar a segurança. O **TLS 1.1**, lançado em 2006, adicionou proteções contra ataques de encadeamento de blocos de criptografia (CBC). Em 2008, o TLS 1.2 trouxe suporte para algoritmos de hash mais seguros e possibilitou o uso de criptografia baseada em Elliptic Curve Cryptography (ECC). A versão mais recente, TLS 1.3, lançada em 2018, eliminou algoritmos inseguros, como RC4 e SHA-1, e reduziu o tempo do handshake, melhorando a eficiência sem comprometer a segurança.

O **HTTPS** (HyperText Transfer Protocol Secure) é a implementação do protocolo HTTP com uma camada adicional de segurança baseada no **TLS**. Ele garante que os dados transmitidos entre um navegador e um servidor web sejam criptografados, protegendo informações sensíveis como credenciais de login, números de cartão de crédito e dados pessoais. O **HTTPS** é amplamente adotado na internet e é identificado pelo prefixo *"https://"* na barra de endereços do navegador, além do ícone de cadeado, que indica que a conexão é segura. [6]

A principal diferença entre HTTP e **HTTPS** está na proteção dos dados. Enquanto o HTTP transmite informações em texto simples, tornando-as vulneráveis a interceptações, o **HTTPS** uusa o TLS para criptografar os dados e garantir sua integri-

dade. Isso impede que hackers realizem ataques do tipo Man-in-the-Middle (MITM) e modifiquem o conteúdo da comunicação. Além disso, o **HTTPS** melhora a confiança dos usuários, pois a identidade do servidor pode ser verificada por meio do certificado digital, garantindo que não se trata de um site fraudulento. [5] A implementação do **HTTPS** em um site requer a obtenção de um certificado **SSL/TLS**, que pode ser emitido por Autoridades de Certificação (CAs) confiáveis. Existem diferentes tipos de certificados, como os EV (Extended Validation), que oferecem o mais alto nível de autenticação e exibem o nome da empresa na barra do navegador, os OV (Organization Validation), que validam a identidade da organização, e os DV (Domain Validation), que apenas confirmam a propriedade do domínio. Além desses, há certificados Wildcard, que protegem um domínio e seus subdomínios, e Multi-Domain (MDC/UCC), que protegem múltiplos domínios.

Com a evolução das ameaças cibernéticas, o uso do **HTTPS** se tornou essencial para proteger a privacidade dos usuários e a integridade dos dados na web. Navegadores modernos como o *Google Chrome* e o *Mozilla Firefox* sinalizam sites sem HTTPS como ”não seguros”, desencorajando usuários de acessá-los. Além disso, o Google prioriza sites que utilizam HTTPS em seus rankings de busca, tornando sua adoção uma prática recomendada para qualquer site que queira garantir segurança e credibilidade.

A segurança do HTTPS pode ser aprimorada com a implementação de HTTP Strict Transport Security (HSTS), que força os navegadores a sempre estabelecer conexões seguras com o site, prevenindo ataques de downgrade para HTTP. O uso de TLS 1.3 também melhora a eficiência e a robustez da proteção, garantindo que a comunicação seja rápida e segura. [5]

Dessa forma, a evolução do SSL para o TLS e a adoção generalizada do HTTPS representam avanços fundamentais na segurança da internet. Esses protocolos garantem que as informações transmitidas entre clientes e servidores permaneçam protegidas contra interceptação, modificação e falsificação, proporcionando um ambiente digital mais seguro para todos.

4. Implementação prática

4.1. Módulos Client e Server

4.1.1. Geração de Certificados SSL/TLS

A comunicação HTTPS exige um certificado digital para autenticar o servidor e garantir a segurança da transmissão de dados. O seguinte script, *gerar_certificado.py*, foi utilizado para criar um certificado autoassinado, garantindo a autenticação sem a necessidade de uma autoridade certificadora externa.

```python
from OpenSSL import crypto

# Configurações do certificado
CERT_FILE = "cert.pem"
KEY_FILE = "key.pem"

# Criar uma chave privada
key = crypto.PKey()
key.generate_key(crypto.TYPE_RSA, bits: 2048)
```

Figure 1. gerar_certificado.py parte 1

O script utiliza o módulo *crypto* da biblioteca OpenSSL para manipular certificados e chaves criptográficas. Primeiramente, são definidos os nomes dos arquivos onde a chave privada e o certificado digital serão armazenados.

O próximo passo é criar um objeto PKey, que representa a chave criptográfica. Em seguida, uma chave privada RSA de 2048 bits é gerada. Essa chave será utilizada posteriormente para assinar digitalmente o certificado:

```python
# Criar um certificado autoassinado
cert = crypto.X509()
cert.get_subject().C = "BR"  # País
cert.get_subject().ST = "Distrito Federal"  # Estado
cert.get_subject().L = "Brasília"  # Cidade
cert.get_subject().O = "UniversidadeDeBrasilia"  # Organização
cert.get_subject().OU = "CIC"  # Unidade Organizacional
cert.get_subject().CN = "localhost"  # Nome Comum

# Definir período de validade do certificado
cert.set_serial_number(1000)
cert.gmtime_adj_notBefore(0)  # Início da validade
cert.gmtime_adj_notAfter(365 * 24 * 60 * 60)  # Expiração (1 ano)

# Associar a chave privada ao certificado
cert.set_issuer(cert.get_subject())
cert.set_pubkey(key)
cert.sign(key, digest: "sha256")  # Assinar com SHA-256
```

Figure 2. gerar_certificado.py parte 2

Em seguida, o script cria um certificado digital no formato X.509, que é o padrão

amplamente utilizado para autenticação e criptografia em redes. Os detalhes do certificado, como o nome do emissor e do sujeito, são preenchidos:

- **C (Country)**: BR - Brasil
- **ST (State)**: Distrito Federal
- **L (Locality)**: Brasília
- **O (Organization)**: Univerisdade de Brasília
- **OU (Organizational Unit)**: CIC (Departamento)
- **CN (Common Name)**: localhost (Nome comum usado para validar domínios)

O certificado recebe um número de série e um período de validade. Ele é configurado para ser válido imediatamente e terá uma validade de 1 ano.

Como este é um certificado autoassinado, ele mesmo é definido como sua própria Autoridade Certificadora (CA). Além disso, a chave pública do certificado é associada a ele, garantindo que possa ser validado posteriormente.

Para garantir autenticidade e integridade, o certificado é assinado digitalmente com a chave privada gerada anteriormente, utilizando o algoritmo SHA-256.

```
# Salvar o certificado e a chave privada em arquivos
with open(CERT_FILE, "wb") as cert_file:
    cert_file.write(crypto.dump_certificate(crypto.FILETYPE_PEM, cert))

with open(KEY_FILE, "wb") as key_file:
    key_file.write(crypto.dump_privatekey(crypto.FILETYPE_PEM, key))

print(f"Certificado '{CERT_FILE}' e chave privada '{KEY_FILE}' gerados com sucesso!")
```

Figure 3. gerar_certificado.py parte 3

Após a geração do certificado, ele precisa ser salvo em um formato adequado para ser utilizado em servidores HTTPS.

- O certificado é salvo no formato **PEM** (Privacy-Enhanced Mail), um padrão em Base64 amplamente usado para certificados SSL/TLS:

Figure 4. Certificado gerado

- Da mesma forma, a chave privada é armazenada no formato PEM em um arquivo separado, garantindo que possa ser usada posteriormente para autenticação e renovação do certificado:

```
-----BEGIN PRIVATE KEY-----
MIIEvwIBADANBgkqhkiG9w0BAQEFAASCBKkwggSlAgEAAoIBAQCl7WcHWqNwM0Io
yjLCdG7AsbkmpRMFqmqXX/bWjwFCMB9XW8vCaiWzNO9IV3y2kKj6/abnj4+X1iEB
L/zY45Q8wt+f9Qmx2/6XWbS6QgdzVZpZ58R/OcjSqDC1E4ONn3qLVDi/q6Z/I1Ig
/b0fNFKRidDi1GA4JJwBAlMRfyqQ6+9dQsxrL7ZHm/VOochDWBuutowOM3CwMuAj
t929n0lP33l+AEDf/Tc2vDRhkKoWQMU4PWWHOy9wfNUuickq+CW/gNFSf2U25Mr3
uUYXJZHPsgoLU12wGFmaiHgWDzfrCVYBwoa66Kv/uddGexXtasemAjPoGCyKDB5c
j/LcjPoDAgMBAAECggEASLj1nHwFwPG+T+la+O4V8n+SQhU8cpG/2OuW4G/Z4dv8
iImzkth6AIK1UxQBFXTukt7kLX42tZoEaiRRoe+Qs1bHszAu2RSuD0CiXLX+1PZS
SZAyh35Yfbd4bYyJbmNUzscRfjQpq3RcFRiQsKH6hE4OTrQ9Ha4x76FIrhT5n30L
SiQ+H+L3t7C+gASopUD6k+JQTJ3pd/yr+5j7OFzem5yOUwpzC0xA9CvINRDkTLel
0Az+/MNFnYYPJYhKIZFEKz8tOTawzCdAa/9jKg867AmDKGs14CXRDJz4nzXJRFWk
OuAnVo6oRsRLmLayaO3nWdEcCziPhTEOyYrs+5/awQKBgQDhsXufGGtiAKWP9RSr
98JzTzPOVgGt9AO3euBfoyBBqmMWXjgQeuZ8YUEwFRQcENDdx3I1xACFtgebqwKd
zFDS8EJPPyQKd3ym9UANnH2gyHRykWQTDy1iRXXpXuHx0dinTH6ImwfnAJEcy4zl
iC6wsFmG10W/yLIYotzmxIkvQQKBgQC8NWIdeJ4kdJc3iXInLH8Y2K2PU2Z7cKBw
Ea0VcvmRpzaln9b3HtwOYYWDLeuGutN5lkgZqoMmY+N9R5ZOUlZdNdw/U3GC0eip
4g9rC4a2EJlwOdrFrTu4mzf4Z07H4vL5yrOLfBwViAPrt7K4B7iod/X+Zzy8dmjg
FjSYd1qcQwKBgQCdzZnxKILUmD0qxaK5htd4BnXjnHE6LrZ2kLzXQdio8TitCB3N
MBF+AAYzV1mjMgLIPs7VVwFCfH8fUdfXe2c7xe+nNy+8/cSzfCFYIwYX6HC7Nnsx
dFg4MPb7wHLcwOvqNer3GubQlq3f41bVmn2C3xsJ0/UZ3nrgDOfFlr+QwQKBgQCq
jj3xsyKsqrN2cqnt8hbjb0gpQEz4xCHALWSKsXyUj9Gn2ee2XlLQ/XM+Y1iy6Y9G
+fcwuyLKoW7AUyVUqKz3Hx4OjRGqmYqcmO2b9gAH8Zb66pLk3I3oR1E7XHM6FOJX
l2eJmdjhcjcLX5cnymd+0+cqumeB6lQz7iuPl2mXfQKBgQCHkTboSbHPXMt80L3h
LVgt8cXJRgqFQEU+0n0Rt8YAIzbMVHuZJ+UGuDa1s3AdxPAfn1lz7JZpGfS46i8C
DMXY4d7Sm/9vYBoE9NnFP9AJ0hRrayT7zAbvUTy0vxJp+VKSGPJLX1a0qtcWJ6Il
pcFxKArKwJ+1btKKzHSTzjoE0g==
-----END PRIVATE KEY-----
```

Figure 5. Chave gerada

Por fim, uma mensagem é exibida no terminal para informar que os arquivos foram criados com sucesso, concluindo o processo de geração do certificado autoassinado.

4.1.2. Implementação do Servidor HTTPS

```python
from flask import Flask
import ssl

app = Flask(__name__)


@app.route('/')  new *
def home():
    return "Servidor HTTPS ativo e seguro!"


if __name__ == '__main__':
    context = ('cert.pem', 'key.pem')
    app.run(host='0.0.0.0', port=4433, ssl_context=context)
```

Figure 6. server.py

A biblioteca Flask é utilizada para criar um servidor HTTPS, garantindo a comunicação segura entre cliente e servidor. Para isso, os arquivos do certificado SSL/TLS previamente gerados são especificados no código.

O servidor Flask é configurado para ser executado na porta 4433, permitindo conexões seguras via HTTPS. Como estamos utilizando um certificado autoassinado, a requisição HTTPS desativa a verificação do certificado para evitar erros de validação durante os testes.

Essa abordagem permite estabelecer uma conexão segura mesmo sem um certificado emitido por uma autoridade certificadora (CA), sendo útil para ambientes de desenvolvimento e testes locais.

4.1.3. Implementação do Cliente HTTPS

Para validar a conexão HTTPS, foi implementado um cliente que se conecta ao servidor seguro e verifica a resposta recebida. O código do cliente está descrito a seguir:

```python
import requests
import urllib3

url = "https://localhost:4433"

# Ignorar avisos de SSL inseguros
urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)

# Enviar a requisição ignorando a verificação do certificado
response = requests.get(url, verify=False)

print(f"Resposta do servidor: {response.text}")
```

Figure 7. client.py

O cliente utiliza a biblioteca requests para enviar uma requisição HTTPS ao servidor. Como estamos usando um certificado autoassinado, a verificação do certificado é desativada com a opção verify=False. Isso permite que a requisição seja processada sem erros de validação, o que é útil em ambientes de teste e desenvolvimento.

Essa abordagem garante que a comunicação segura via HTTPS está funcionando corretamente antes de integrar um certificado emitido por uma Autoridade Certificadora (CA) em produção.

4.2. Testes de Comunicação e Análise no Wireshark

Para garantir que a comunicação esteja criptografada, foi utilizada a ferramenta Wireshark para capturar pacotes durante a execução do cliente.

4.2.1. Captura do Handshake TLS

A captura do Client Hello e Server Hello confirmou que o handshake TLS ocorreu corretamente. A versão do protocolo utilizada foi TLS 1.2, conforme indicado pelos bytes **0x0303**

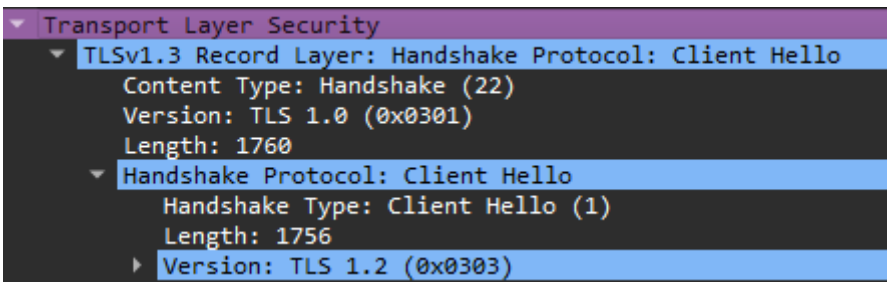

Figure 8. Hello Client

4.2.2. Verificação da Criptografia

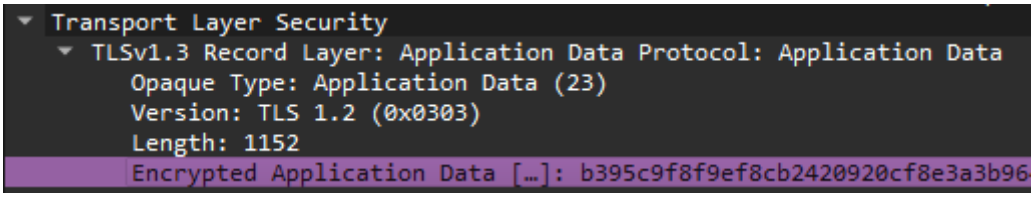

Figure 9. Application Data

Os pacotes identificados como Application Data demonstraram que a comunicação foi criptografada. O conteúdo das mensagens não pôde ser visualizado, garantindo a proteção dos dados transmitidos.

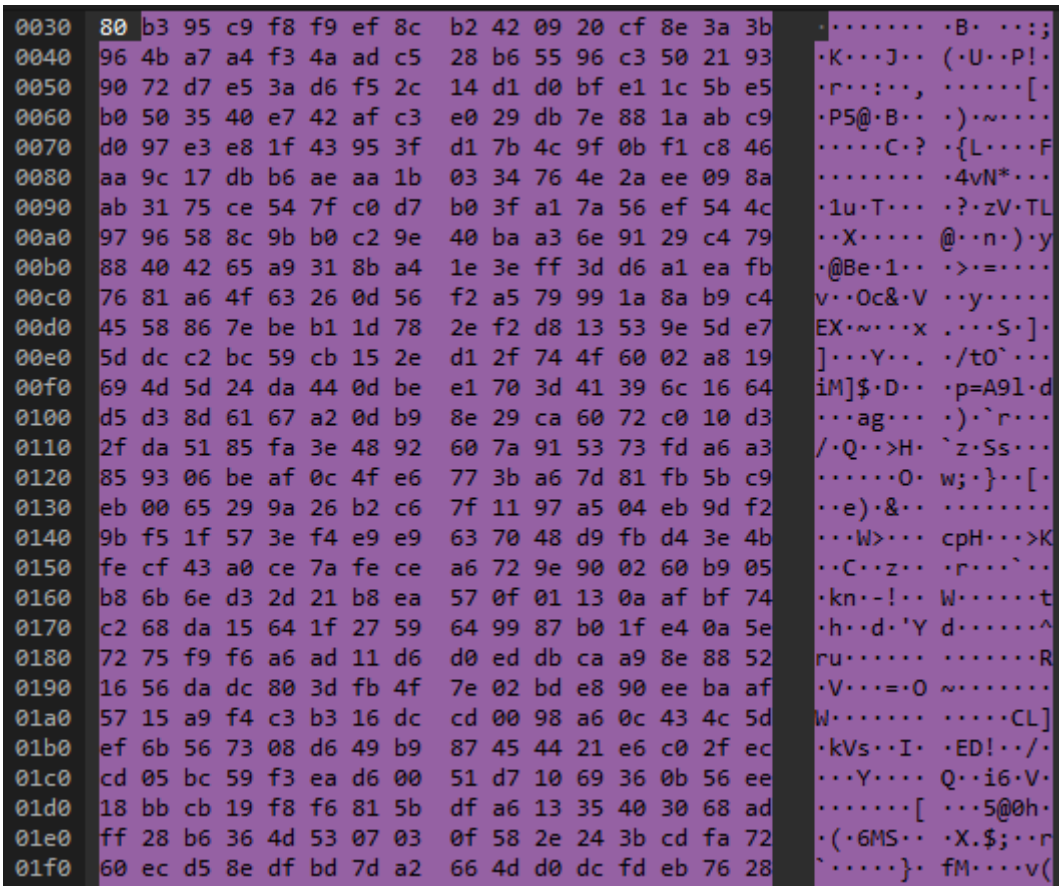

Figure 10. Application Data Dados

4.2.3. Conclusão da Análise

A análise do wireshark confirmou que:

- O handshake TLS foi realizado com sucesso
- A comunicação entre cliente e servidor foi criptografada
- O protocolo usado foi TLS 1.2

5. Código Principal (main.py)

O código implementa um menu interativo que permite ao usuário iniciar e parar um servidor HTTPS, além de abrir automaticamente o navegador para acessar o cliente. Para garantir a comunicação segura, o sistema gera certificados SSL/TLS antes da execução, evitando a necessidade de configuração manual.

A função **gerar_certificado()** executa o script **gerar_certificado.py**, que cria um certificado SSL/TLS autoassinado (cert.pem) e sua chave privada (key.pem). Em seguida, a função **iniciar_servidor()** executa server.py em segundo plano usando subprocess.Popen(), permitindo que o menu continue acessível enquanto o servidor está rodando. Para garantir que o servidor tenha tempo de inicializar corretamente, um pequeno atraso (time.sleep(2)) é inserido.

Já a função **iniciar_cliente()** utiliza webbrowser.open() para abrir automaticamente o navegador na URL **https://127.0.0.1:4433**, permitindo que o usuário acesse o servidor sem precisar digitar o endereço manualmente. No menu interativo, o usuário pode iniciar o servidor, acessar o cliente, parar o servidor (caso já esteja rodando) e sair do programa, garantindo um controle total do sistema.

Além disso, antes de sair da aplicação, o código verifica se o servidor está em execução e o finaliza corretamente (terminate()) para evitar processos em segundo plano desnecessários. Essa implementação garante que o sistema seja executado de maneira automatizada, segura e intuitiva, proporcionando uma comunicação criptografada via HTTPS sem exigir configurações avançadas do usuário.

```python
import os
import subprocess
import webbrowser
import time

# Armazena o processo do servidor
server_process = None


def gerar_certificado():  1 usage
    os.system("python gerar_certificado.py")


def iniciar_servidor():  1 usage
    global server_process
    if server_process is None:
        # Inicia o servidor em um processo separado
        server_process = subprocess.Popen( args: ["python", "server.py"], stdout=subprocess.PIPE, stderr=subprocess.PIPE)
        print("Servidor iniciado em segundo plano!")
        time.sleep(2)  # Espera alguns segundos para garantir que o servidor esteja rodando


def iniciar_cliente():  1 usage
    url = "https://127.0.0.1:4433"  # URL do servidor
    print(f"Abrindo o navegador em {url}...")
    webbrowser.open(url)  # Abre a URL no navegador padrão
```

Figure 11. main.py parte 1

```python
def main():  1 usage
    global server_process
    while True:
        print("\n===== Sistema HTTPS =====")
        print("1 - Iniciar Servidor")
        print("2 - Iniciar Cliente (Abrir navegador)")
        print("3 - Parar Servidor")
        print("4 - Sair")
        opcao = input("Escolha uma opção: ")

        if opcao == "1":
            iniciar_servidor()
        elif opcao == "2":
            iniciar_cliente()
        elif opcao == "3":
            if server_process is not None:
                server_process.terminate()
                server_process = None
                print("Servidor parado!")
        elif opcao == "4":
            if server_process is not None:
                server_process.terminate()
            print("Encerrando...")
            break
        else:
            print("Opção inválida. Tente novamente.")


if __name__ == "__main__":
    gerar_certificado()
    main()
```

Figure 12. main.py parte 2

6. Conclusão

A implementação de um servidor e cliente HTTPS permitiu demonstrar, na prática, como ocorre a comunicação segura utilizando o protocolo TLS. A geração e uso de certificados SSL/TLS foram fundamentais para garantir a autenticidade do servidor, enquanto a análise dos pacotes no Wireshark validou que os dados trafegados estavam protegidos contra interceptação. Essa abordagem reforça a importância do HTTPS na segurança da internet, sendo essencial para proteger informações sensíveis e garantir a confiabilidade das conexões online.

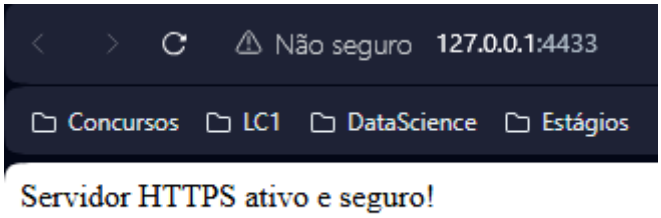

Figure 13. Mensagem no navegador

References

[1] Cloudfare. O que é o tls (transport layer security)? https://www.cloudflare.com/pt-br/learning/ssl/transport-layer-security-tls/. Accessed: 2025-02-10.

[2] Bernardo Martins Costa. Segurança na internet - protocolo ssl/tls. https://www.gta.ufrj.br/ensino/eel879/trabalhos_vf_2010_2/bernardo/tls.html. Accessed: 2025-02-10.

[3] Ivan de Souza. O que é tls e quais são as diferenças entre ele e ssl? https://www.cloudflare.com/pt-br/learning/ssl/transport-layer-security-tls/. Accessed: 2025-02-10.

[4] Digicert. O que é um certificado ssl? https://www.digicert.com/pt/what-is-an-ssl-certificate#:~:text=inserir%20informaμes%20confidenciais.-,O%20que%20%20SSL%20(Secure%20Sockets%20Layer)%3F,por%20exemplo%2C%20o%20Outlook). Accessed: 2025-02-10.

[5] Fortinet. O que é hypertext transfer protocol secure (https)? https://www.fortinet.com/br/resources/cyberglossary/what-is-https. Accessed: 2025-02-10.

[6] Wikipédia. Hyper text transfer protocol secure — wikipédia, a enciclopédia livre. https://pt.wikipedia.org/w/index.php?title=Hyper_Text_Transfer_Protocol_Secure&oldid=68867257. Accessed: 2025-02-10.