Análise Técnica do Protocolo HTTPS over TLS

February 16, 2025

1 Introdução

A comunicação em redes de computadores é baseada em uma série de protocolos bem estabelecidos e conhecidos por cada nó da rede para que seja possível haver trocas de mensagens e informações entre esses diferentes nós. Com isso, na internet um dos principais protocolos de comunicação é o HyperText Transfer Protocol (HTTP) sendo hoje em dia a base de qualquer comunicação web. Porém o HTTP não surgiu com o intuito de prover segurança em suas comunicações e com o avançar da internet rapidamente percebeu-se a necessidade de criptografar essas comunicações HTTP. Assim, surgiu-se a ideia do HTTPS, uma extensão do HTTP com o intuito de fornecer segurança às comunicações. Primeiramente, o protocolo utilizado para fornecer essa segurança foi o SSL, mas devido algumas vulnerabilidades, hoje o mais recomendado é o HTTPS over TLS, outro protocolo que surgiu da evolução do SSL.

Este relatório tem como objetivo explorar os principais protocolos envolvidos na comunicação segura na web, abordando o funcionamento do HTTP, HTTPS, SSL e TLS. Além disso, será feita uma breve análise das versões de cada um desses protocolos, destacando também a transição do SSL para o TLS.

Por fim, ainda, esse relatório consta com uma parte prática, onde será implementado uma rede de comunicação do tipo cliente servidor utilizando o protocolo HTTPS over TLS. Esse experimento consiste em uma simulação de comunicação entre alguns clientes e um servidor, os clientes farão requisições sobre o protocolo HTTPS para o servidor solicitando alguns componentes HTMLs, também será feita uma pequena análise dessas comunicações, bem como a geração do certificado e chave privada do servidor. Vale ressaltar que esse experimento será implementado utilizando o módulo do python SSL para implementação do HTTPS over TLS, do módulo socket para comunicação na rede local e do openssl para criação do certificado digital e chave privada do servidor.

2 HyperText Transfer Protocol

O HyperText Transfer Protocol (HTTP) é um protocolo de comunicação da camada de aplicação que surgiu em 1991 com a crescente necessidade de transmissão de hiper texto de uma forma eficiente e com boa alta abstração. É amplamente utilizado no contexto web por navegadores e servidores que, ao interagirem, torna possível a troca de informações na web. Além da transmissão de hiper texto, rapidamente o protocolo evoluiu para transmissão de documentos, imagens, arquivos e outros tipos de dados por meio de uma estrutura simples baseada em requisições e respostas.

Porém os dados trafegados no protocolo HTTP são em formato plain text, assim, o protocolo não oferece nenhum grau de segurança por padrão, pois os dados trafegam de forma clara para qualquer usuário que tenha acesso a requisição. Dessa forma, com a crescente utilização da web para realizar

comunicações de grande importância, surgiu-se a necessidade de uma expansão do protocolo para comunicações que necessitem de segurança, o HTTPS.

Antes de partirmos para o HTTPS, primeiro vamos fazer uma breve análise das versões do HTTP. Para isso, iremos considerar apenas até o HTTP/2, já que sua versão 3 ainda não está totalmente suportada pela maioria das aplicações web.

2.1 Comparativo das Diferentes Versões

2.1.1 HTTP/0.9 (1991)

- Primeira versão do protocolo, extremamente simples.
- Suportava apenas o método `GET` e não possuía cabeçalhos nem suporte para envio de arquivos além de HTML.

2.1.2 HTTP/1.0 (1996)

- Introduziu cabeçalhos HTTP, permitindo informações adicionais nas requisições.
- Suporte para outros métodos além de `GET`, como `POST` e `HEAD`.
- Comunicação não persistente, ou seja, uma nova conexão TCP era aberta para cada requisição, tornando a comunicação lenta e ineficiente.

2.1.3 HTTP/1.1 (1997)

- Melhorias no desempenho com suporte a conexões persistentes (Keep-Alive).
- Introdução do pipelining, permitindo múltiplas requisições em uma única conexão TCP.
- Novos métodos adicionados, como `PUT`, `DELETE`, `OPTIONS`.
- Suporte a cache avançado e compressão de dados.
- Ainda é amplamente utilizado hoje em dia.

2.1.4 HTTP/2 (2015)

- Otimizações no desempenho, incluindo multiplexação, permitindo múltiplas requisições simultâneas em uma única conexão TCP.
- Redução da sobrecarga dos cabeçalhos HTTP, melhorando a eficiência da comunicação.
- Introdução do server push, permitindo que o servidor envie dados antecipadamente para o cliente sem esperar uma requisição explícita.
- Maior eficiência em conexões seguras (HTTPS é amplamente adotado junto ao HTTP/2).

3 Hyper Text Transfer Protocol Secure

Para a primeira implementação do Hyper Text Transfer Protocol Secure (HTTPS) foi utilizado o protocolo Secure Sockets Layer (SSL) o qual foi projetado pela Netscape Communications Corporation no início da década de 90 com o intuito de adicionar segurança a diversos protocolos, incluindo HTTP, SMTP e FTP. Assim, o SSL passou a permitir a transmissão criptografada de dados entre clientes e servidores, protegendo as comunicações de ataques como interceptações e manipulações de dados. Sua adoção foi ampla, sendo utilizado em bancos, lojas online e serviços de e-mail para garantir a segurança de informações sensíveis.

3.1 SSL

Agora partiremos para uma análise mais profunda do protocolo, analisando passo a passo como o SSL consegue transmitir privacidade, autenticação e integridade de dados em comunicações de internet. Na primeira tentativa de comunicação do cliente pelo servidor, o cliente e/ou servidor podem sinalizar que desejam utilizar o protocolo HTTPS, caso ambas aceitem, o protocolo do SSL começa a operar. O primeiro passo é chamado de handshake, o qual é uma etapa onde o servidor e o cliente estão acordando como a conexão irá acontecer. Por exemplo, o cliente transmite a lista de algoritmos de segurança que ele é capaz de operar e, assim, o servidor decide a lista de algoritmos que serão utilizados na comunicação.

Após essa etapa de acordo de algoritmos e funções, o servidor envia um certificado digital e sua chave pública, este certificado é previamente gerado por uma autoridade de certificação. Uma autoridade de certificação é uma instituição responsável por gerar certificados digitais para cada instituição que a solicite para tal, atuando como um terceiro confiável na comunicação entre dois agentes que ainda não possuem certeza de suas identidades.

O certificado digital é gerado pela AC a partir de informações de identificação do solicitante e uma chave pública que o solicitante queira cadastrar em seu certificado. Essa solicitação de certificado digital é um processo que é realizado em ambientes seguros e controlados, para se ter certeza que a pessoa que está se cadastrando realmente é quem diz ser. Assim, com informações de ID e a chave pública do solicitante a AC termina de gerar o certificado e por fim gera um hash desse certificado, o qual é assinado pela chave privada da AC e é concatenado ao certificado digital que deu origem ao hash. Vale-se ressaltar que essa concatenação do certificado com o hash assinado é armazenada pela AC a qual irá o disponibilizar para qualquer usuário que queira verificar qual é o certificado daquela instituição.

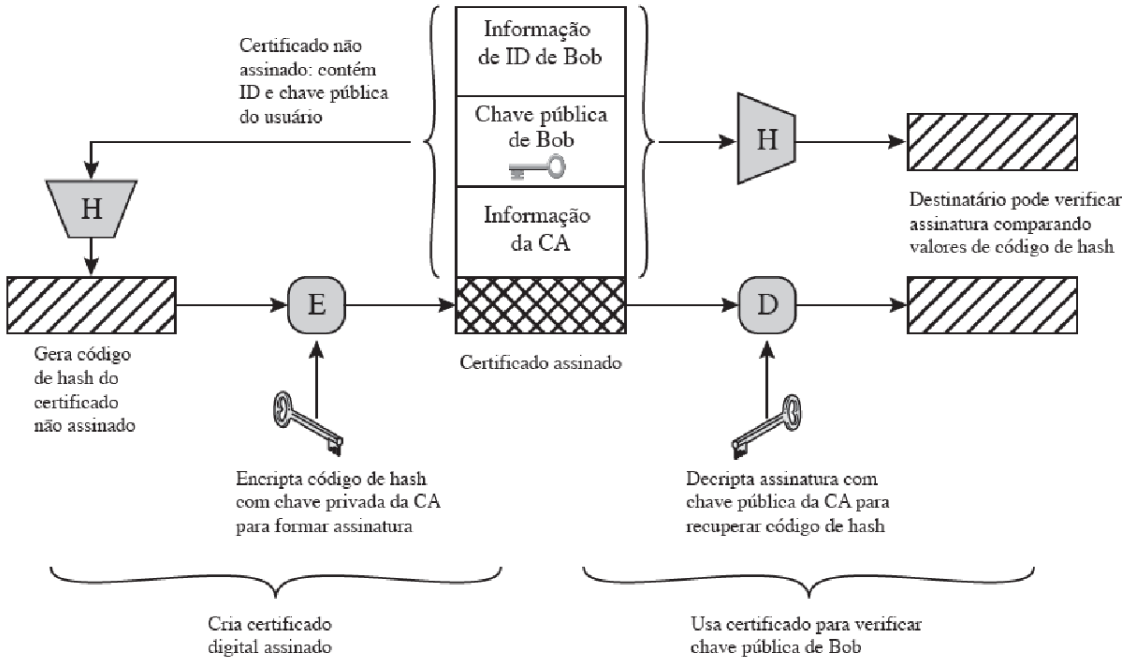
Com isso fica claro a estratégia do SSL. O que o SSL propoẽ é que quando um usuário receber o certificado digital de um emissor, ele irá fazer o hash deste certificado e irá solicitar o certificado dessa instituição na AC a qual o emissor diz ter gerado esse certificado (a AC é informada pelo servidor ao cliente durante a etapa de Handshake). Com o certificado recebido da AC, o cliente irá descriptografar o hash acoplado ao certificado, usando a chave pública da AC. Caso o emissor diz ser quem é, esses dois hashs, tanto o originado da descriptografia quanto o calculado através do certificado enviado pelo servidor, serão iguais garantindo a autenticidade do servidor, evitando-se assim, ataques de man in the middle.

Com a identidade confirmando, utilizando a chave pública fornecida pelo servidor, o cliente e o servidor vão traçar qual será a chave de criptografia simétrica utilizada para o restante da comunicação. Isso é feito ao cliente criptografar um número aleatório com a chave pública recebida do servidor e enviar o resultado ao servidor, o qual é o único capaz de desvendar a mensagem

utilizando sua chave privada. Com isso, ambos fazem utilização desse número aleatório para gerar a chave de sessão única para a criptografia simétrica da comunicação durante o restante da sessão.

Isso é feito, pois, normalmente, a criptografia simétrica é mais performática em relação a assimétrica. Assim, a chave simétrica é estabelecida, permitindo o início de uma conexão segura, na qual os dados são criptografados e descriptografados utilizando a chave de sessão ao longo de toda a comunicação. Caso qualquer uma das etapas anteriores falhe, o protocolo não será concluído e a conexão não será estabelecida.

Para tornar mais fácil a visualização do processo de validação do certificado digital, aqui está uma imagem didática que sintetiza o processo acima:



Para fins deste relatório, em relação a parte dois de experimentação, iremos sempre utilizar o RSA para nossa criptografia assimétrica e o AES para a criptografia simétrica.

3.1.1 Vulnerabilidades do SSL

Com o passar dos anos, foram encontradas algumas vulnerabilidades no protocolo e novas versões foram desenvolvidas para corrigir essas vulnerabilidades e até mesmo melhorar a capacidade de segurança para ameaças ainda não descobertas. Assim, o protocolo SSL foi se tornado obsoleto e desaconselhado, com a sua última versão suportada durando até o ano de 2014, o SSL 3.0. No ano de 2014 foi descoberta uma nova forma de ataque ao protocolo, o ataque POODLE (Padding Oracle On Downgraded Legacy Encryption). Esse ataque foi descoberto e divulgado por um trio do time de segurança da Google. No caso do SSL 3.0 eram necessárias somente 256 tentativas para desvendar 1 byte da mensagem criptografada. Após a divulgação dessa falha, navegadores retiraram sua compatibilidade com o SSL 3.0 e o algoritmo foi completamente substituído por protocolos mais recentes como o TLS 1.2 e TLS 1.3.

Os novos protocolos substituíram o SSL e trouxeram diversas melhorias, incluindo suporte para algoritmos mais seguros, melhor desempenho e maior resistência contra ataques. Atualmente,

TLS 1.2 e TLS 1.3 são os padrões recomendados para aplicações seguras na web, eliminando as vulnerabilidades do SSL e garantindo criptografia robusta, autenticação confiável e integridade dos dados.

3.2 TLS

Como vimos na subseção anterior, o Transport Layer Security (TLS) substitui o SSL como novo protocolo para garantir segurança a comunicações web. O TLS é um protocolo de segurança projetado para fornecer criptografia, autenticação e integridade na comunicação entre dispositivos conectados à internet. Atualmente o TLS é o padrão de segurança utilizado na web, sendo essencial para comunicações seguras em HTTPS, conexões de email (SMTP, IMAP, POP3) e até mesmo em protocolos de comunicação em tempo real, como VoIP e VPNs.

Porém, ele não é simplesmente um sucessor do SSL, ele é uma evolução do protocolo SSL, mantendo, assim, etapas consideráveis do protocolo anterior. Dessa forma, para não tornar esse relatório repetitivo, repetindo explicações de etapas já explicadas no protocolo SSL, iremos abordar o que é diferente entre os dois protocolos e como essas mudanças tornaram o TLS mais seguro que o SSL. Como ambos possuem diversas versões, iremos considerar a versão SSL 3.0 e a versão 1.3 do TLS.

As principais mudanças são: a substituição de algoritmos criptográficos inseguros, como RC4 e 3DES, por cifras mais robustas, como AES-GCM e ChaCha20-Poly1305; o aprimoramento da autenticação com reforço no uso de certificados digitais assinados por Autoridades Certificadoras (CAs) e a introdução do Perfect Forward Secrecy (PFS) por meio do Elliptic Curve Diffie-Hellman Ephemeral (ECDHE); a adoção do ECDHE como método de troca de chaves e a eliminação de funções de hash obsoletas, como MD5 e SHA-1, com a padronização do SHA-256 e HMAC para garantir a integridade dos dados. Essas melhorias tornaram o TLS 1.3 o protocolo recomendado para comunicações seguras, eliminando as vulnerabilidades do SSL e assegurando maior privacidade, eficiência e resistência a ataques cibernéticos.

3.2.1 Comparativo das Diferentes Versões do TLS

Versão	Ano	Características Principais
TLS 1.0	1999	Substituiu o SSL 3.0, mas herdou algumas vulnerabilidades.
TLS 1.1	2006	Melhorias na proteção contra ataques, mas ainda vulnerável.
TLS 1.2	2008	Adotado amplamente, suporta AES-GCM e SHA-256.
TLS 1.3	2018	Removidos algoritmos inseguros, handshake mais rápido e seguro.

4 Experimentação Comunicação HTTPS Cliente Servidor

Partiremos agora para a parte de experimentação deste relatório. O experimento contará com uma simulação de uma comunicação segura utilizando o protocolo HTTPS over TLS. A nossa experimentação constará com três elementos principais, um servidor responsável por hospedar 10 páginas htmls, um cliente que abre uma conexão protegida com o servidor solicitando a cada 1 segundo uma enésima página, tal que n= 0 e i = 9, e uma autoridade certificadora em que o servidor irá cadastrar um certificado digital e o cliente irá solicitar a credencial do servidor no

início da comunicação com o servidor a fim de ter certeza de que a comunicação é segura entre o cliente e o servidor.

Para fins dessa parte do relatório, iremos sempre utilizar o RSA para nossa criptografia assimétrica e o AES, TLS_AES_256_GCM_SHA384 de chave de 256 bits, para a criptografia simétrica. Então, caso seja citado chave privada ou chave pública, entenda que elas se referem às chaves do algoritmo RSA, e, caso seja citado chave simétrica ou chave de sessão, ela será a chave do algoritmo AES. O certificado digital será no padrão X.509. Já em relação aos protocolos, serão utilizados o HTTP 1.1 e o TLS na versão TLSv1.3.

Com isso o fluxo do nosso experimento será o seguinte:

- Autoridade certificadora é iniciada.
- Servidor é iniciado.
- Servidor envia dados de identificação e chave pública.
- Autoridade certificadora gera o certificado e envia para o Servidor.
- Autoridade certificadora gera hash do certificado digital, assina esse hash com sua chave privada, e salva a concatenação em um arquivo .txt.
- Cliente é iniciado.
- Cliente inicia comunicação com servidor.
- Servidor envia certificado digital para cliente.
- Cliente faz o hash desse certificado digital.
- Cliente solicita credencial do servidor para a autoridade certificadora.
- Cliente desfaz a concatenação entre o hash assinado pela autoridade certificadora.
- Cliente faz a decriptação do hash com a chave pública da autoridade certificadora.
- Cliente compara os hashes, sendo iguais, acredita que a conexão é válida, forma a chave de sessão com o servidor utilizando a chave pública do servidor e faz a transmissão de dados.

Por fim, para simulação dessas três entidades, será iniciada uma thread para cada uma das entidades, mas, ressalta-se que não foram utilizadas técnicas avançadas de sincronização de programação concorrente, apenas alguns sleeps para definir de forma simplificada a ordem de execução. (Compreendo totalmente que um sleep não define ordem de execução, mas para o escopo desse projeto, funciona bem).

4.1 Autoridade Certificadora

4.1.1 Geração das Chaves CA

Começaremos nossa implementação gerando a chave privada e chave pública das nossa AC. Faremos isso usando o openssl. Com os seguintes dois comandos:

[ ]:
```
openssl genpkey -algorithm RSA -out ca_private.key -pkeyopt rsa_keygen_bits:2048

openssl rsa -in ca_private.key -pubout -out ca_public.pem
```

Pronto, com isso temos as duas chaves necessárias para que a nossa AC seja capaz de assinar os certificados digitais emitidos.

4.1.2 Comunicação com o Servidor e Com o Cliente

Para representar a comunicação com o servidor, iremos montar um servidor TCP que irá responder a chamada do servidor para criar um certificado digital e depois irá finalizar sua execução. O servidor irá informar informações de identificação que são estáticas e definidas por:

- Country Name (2 letter code) [AU]: BR
- State or Province Name (full name) [Some-State]: Distrito Federal
- Locality Name (eg, city) []: Brasília
- Organization Name (eg, company) [Internet Widgits Pty Ltd]: SC2
- Name (eg, section) []: departamento de ciências da computação unb
- Common Name (e.g. server FQDN or YOUR name) []: localhost
- Email Address []: e.marciano.meneses@gmail.com

E, além disso, o servidor irá informar a chave pública também gerada pelo openssl.

Já para a comunicação com o cliente, a fim de simplificar a experimentação, não haverá comunicação com o servidor TCP já que seria necessário configurações adicionais que fugiriam do escopo do projeto. Com isso, o cliente acreditará na veracidado do servidor, sem realizar a checagem da identidade do servidor.

4.1.3 Servidor TCP que representa a AC

Serão usadas as seguintes bibliotecas para representação desse servidor:

[ ]:
```
import socket
import subprocess
import os
import json
```

O servidor irá rodar com as seguites informações:

[ ]:
```
HOST = '127.0.0.1'
PORT = 9000
CERT_DIR = "./certs"
```

Agora iremos definir uma função para subir nosso servidor nas informações dadas acima. O servidor irá se comunicar no protocolo TCP, assim não dará segurança a comunicação, iremos acreditar que o canal entre a AC e o servidor já é seguro, assim, não iremos definir nenhuma segurança para esse canal.

[ ]:
```
def iniciar_ca():
    """Inicia a Autoridade Certificadora como um servidor TCP"""
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as server_socket:
        server_socket.bind((HOST, PORT))
        server_socket.listen(5)
        print(f"[CA] Autoridade Certificadora rodando em {HOST}:{PORT}")

        while True:
            conn, addr = server_socket.accept()
            with conn:
```

```
                print(f"[CA] Pedido recebido de {addr}")
                data = conn.recv(1024).decode()

                try:
                    cert_info = json.loads(data)
                    gerar_certificado(cert_info)
                    conn.sendall(b"CERTIFICADO_GERADO")
                except Exception as e:
                    print(f"[CA] Erro ao processar requisição: {e}")
                    conn.sendall(b"ERRO")
            break
        conn.close()
        server_socket.close()
```

Agora, partiremos para a função de geração do certificado digital. A função irá receber os dados transmitodos pelo servidor e irá montar o certificado digital no padrão X.509.

[ ]:
```
def gerar_certificado(cert_info):
    """Gera um novo certificado SSL usando os dados recebidos do servidor"""
    key_file = os.path.join(CERT_DIR, "server.key")
    cert_file = os.path.join(CERT_DIR, "server.crt")

    if not os.path.exists(CERT_DIR):
        os.makedirs(CERT_DIR)

    print("[CA] Gerando chave privada...")
    subprocess.run(["openssl", "genpkey", "-algorithm", "RSA",
                    "-out", key_file, "-pkeyopt", "rsa_keygen_bits:2048"],␣
 ↪check=True)
    print("[CA] Chave privada gerada com sucesso.")

    print("[CA] Gerando certificado com os dados recebidos...")

    subject = (
        f"/C={cert_info['C']}"
        f"/ST={cert_info['ST']}"
        f"/L={cert_info['L']}"
        f"/O={cert_info['O']}"
        f"/OU={cert_info['OU']}"
        f"/CN={cert_info['CN']}"
        f"/emailAddress={cert_info['email']}"
    )

    subprocess.run([
        "openssl", "req", "-new", "-x509", "-key", key_file, "-out", cert_file,
        "-days", "365", "-subj", subject
    ], check=True)
```

```
    print(f"[CA] Certificado gerado com sucesso: {cert_file}")
```

Pronto, com isso temos o nosso módulo da AC totalmente funcional e cumprindo o seu propósito de geração do certificado digital.

4.2 Servidor

Agora partiremos para o módulo do servidor. O módulo do servidor irá usar as seguintes bibliotecas:

[ ]:
```
import socket
import subprocess
import os
import json
```

A primeira função que será executada pelo servidor será a de "iniciar_servidor()". O que ela fará é definir a ordem de execução do módulo. Primeiro ela irá solicitar a criação do certificado digital para a AC, a fim de simular que o servidor ainda não tinha um certificado cadastrado na entidade. Depois, irá subir o servidor para comunicação segura com com o cliente em HTTPS.

Segue sua implementação:

[ ]:
```
def iniciar_servidor():
    """Solicita o certificado e inicia o servidor HTTPS"""
    solicitar_certificado()
    time.sleep(2)
    start_https_server()
```

Para comunicação com o módulo da AC, iremos definir a função "solicitar_certificado()", que Irá definir uma comunicação TCP com o AC para informar quais são as informações que serão cadastradas no certificado digital e a chave pública que constará nesse certificado.

[ ]:
```
HOST = '127.0.0.1'
PORT = 9000
CERT_DIR = "./certs"
CA_KEY_FILE = os.path.join(CERT_DIR, "ca_private.key")  # Chave privada da CA

def gerar_certificado(cert_info):
    """Gera um novo certificado SSL usando os dados recebidos do servidor"""
    key_file = os.path.join(CERT_DIR, "server.key")
    cert_file = os.path.join(CERT_DIR, "server.crt")

    if not os.path.exists(CERT_DIR):
        os.makedirs(CERT_DIR)

    print("[CA] Gerando chave privada...")
    subprocess.run(["openssl", "genpkey", "-algorithm", "RSA",
                    "-out", key_file, "-pkeyopt", "rsa_keygen_bits:2048"],␣
 ↪check=True)
```

```
    print("[CA] Chave privada gerada com sucesso.")

    print("[CA] Gerando certificado com os dados recebidos...")

    subject = (
        f"/C={cert_info['C']}"
        f"/ST={cert_info['ST']}"
        f"/L={cert_info['L']}"
        f"/O={cert_info['O']}"
        f"/OU={cert_info['OU']}"
        f"/CN={cert_info['CN']}"
        f"/emailAddress={cert_info['email']}"
    )

    subprocess.run([
        "openssl", "req", "-new", "-x509", "-key", key_file, "-out", cert_file,
        "-days", "365", "-subj", subject
    ], check=True)

    print(f"[CA] Certificado gerado com sucesso: {cert_file}")
```

Para a comunicação com o cliente, será definida a seguinte função "start_https_server()". Essa função é o ponto principal do experimento, ela que será responsável por estabelecer a comunicação segura com o cliente através do HTTPS. Para essa função iremos usar o módulo do python ssl, conforme importando a seguir:

[ ]:
```
import ssl
```

Esse módulo será responsável por toda nossa implementação do TSL. Nele iremos definir qual é o tipo de certificado que está sendo utilizado, qual serão os algoritmos de criptografia simétrica e assimétrica, inclusive os tamanhos das chaves. Dessa forma, ele será responsável por abstrair toda a lógica relacionada com a implementação do TSL, garantindo uma maior eficiência do código e uma maior capacidade de leitura.

Assim, essa função também consta com toda a lógica do tratamento das requisições vindas do cliente, encaminhamentos das respostas e o tratamento de exceções caso o cliente enviar alguma requisição que o servidor não é capaz de lidar. Nesse experimento, o nosso cliente apenas faŕa 10 requisições GETs informando qual é o html que ele está interessado em recuperar.

[ ]:
```
# Configuração do servidor HTTPS
def start_https_server(host='127.0.0.1', port=8443):
    context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
    context.load_cert_chain(certfile="./certs/server.crt", keyfile="./certs/
 ↪server.key")

    # Definir que apenas cifras com AES serão usadas
    context.set_ciphers("AES256-GCM-SHA384")
```

```
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0) as server_socket:
        server_socket.bind((host, port))
        server_socket.listen(5)
        print(f"Servidor HTTPS rodando em {host}:{port}")

        with context.wrap_socket(server_socket, server_side=True) as␣
↪secure_socket:
            conn, addr = secure_socket.accept()
            print(f"Conexão recebida de {addr}")

            cipher_info = conn.cipher()
            print(f"Protocolo TLS usado: {conn.version()}")
            print(f"Cipher suite utilizada: {cipher_info[0]}")
            print(f"Tamanho da chave: {cipher_info[2]} bits")

            while True:
                request = conn.recv(1024).decode()
                if not request:
                    break

                # Captura o nome do arquivo solicitado na requisição
                lines = request.split("\r\n")
                if lines and "GET" in lines[0]:
                    file_name = lines[0].split(" ")[1].strip("/")
                else:
                    file_name = "html_0.html"  # Padrão caso a requisição␣
↪esteja malformada

                try:
                    with open(f"./../resource/{file_name}", "r",␣
↪encoding="utf-8") as file:
                        html_content = file.read()
                except FileNotFoundError:
                    html_content = "<html><body><h1>Erro 404: Arquivo não␣
↪encontrado</h1></body></html>"

                response = f"HTTP/1.1 200 OK\r\nContent-Type: text/
↪html\r\n\r\n{html_content}"
                conn.sendall(response.encode())

            conn.close()
            print(f"Conexão encerrada com {addr}")
```

Dessa maneira, terminamos a implementação do nosso módulo do Servidor. A função "gerar_certificado()" ficou responsável pelo cadastro do certificado digital, enquanto a função "start_https_server()" ficou responsável pela implementação do HTTPS com o cliente, utilizando o protocolo TLS implementado pela módulo ssl do python.

4.3 Cliente

O módulo do cliente será o mais simples, visto que estamos ignorando a validade do certificado digital. Isso está sendo feito, pois o módulo SSL do python, nativamente já faz a análise dos certificados validando com autoridades certificadoras reais, assim, como a nossa é fictícia, não seria encontrada e seria gerado um erro de comunicação. Dessa forma, nosso módulo cliente terá apenas uma função, a função "https_client()", a qual será responsável por realizar as requisições GET para o nosso módulo servidor, com o intuito de receber as páginas HTMLs que o servidor está hospedado.

[ ]:
```python
import ssl
import socket
from time import sleep

# Configuração do cliente HTTPS
def https_client(host='127.0.0.1', port=8443):
    context = ssl.create_default_context()
    context.check_hostname = False  # Ignora verificação de hostname
    context.verify_mode = ssl.CERT_NONE  # Ignora verificação do certificado

    with socket.create_connection((host, port)) as sock:
        with context.wrap_socket(sock, server_hostname=host) as secure_socket:
            print(f"Conectado ao servidor HTTPS em {host}:{port}")

            for i in range(10):
                file_name = f"html_{i}.html"
                request = f"GET /{file_name} HTTP/1.1\r\nHost: {host}\r\n\r\n"
                secure_socket.sendall(request.encode())

                response = secure_socket.recv(4096)
                print(f"Resposta do servidor ({file_name}):\n{response.
 ↪decode()}\n")
                sleep(1)

            secure_socket.close()
```

Agora, temos as três entidades principais da nossa aplicação desenvolvidas e explicadas, mas, ainda falta um gerenciador para manipular essas entidades e suas instânciações.

4.4 Main

Para isso, teremos o módulo main, o qual será responsável por instanciar nossas entidades na ordem correta para que a comunicação seja possível.

[ ]:
```python
from servidor import iniciar_servidor
from autoridade_certificadora import iniciar_ca
from cliente import https_client

import time
```

```python
import threading

if __name__ == "__main__":
    # Thread para a Autoridade Certificadora
    ca_thread = threading.Thread(target=iniciar_ca, daemon=True)
    ca_thread.start()

    # Aguarde a inicialização da CA antes de solicitar o certificado
    time.sleep(2)

    # Thread para o Servidor HTTPS
    server_thread = threading.Thread(target=iniciar_servidor, daemon=True)
    server_thread.start()

    # Aguarde o servidor iniciar antes do cliente conectar
    time.sleep(5)

    # Executando o cliente HTTPS
    https_client()
```

Caso tenha interesse, no final desse relatório temos o link do repositório do projeto, nele, após sua clonagem e seguindo o passo a passo definido do README, você poderá executar o módulo main que iniciará toda a comunicação descrita acima.

Para analisar a comunicação, pode-se verificar um log de execução do método main conforme o informado abaixo:

- [CA] Autoridade Certificadora rodando em 127.0.0.1:9000
- [Servidor] Solicitando certificado à CA...
- [CA] Pedido recebido de ('127.0.0.1', 40064)
- [CA] Gerando chave privada...

.+.....+.+.....+....+.+.....+.....+.....+.+..+.+.....+...+.....+...+..++++++++++++++++++++++++
...+.+..+...+.....+..+...+....+.....+..+.....+.....+....+...+.....+...+.....+..+.+.....++++++++++++++++++++++

- [CA] Chave privada gerada com sucesso.
- [CA] Gerando certificado com os dados recebidos...
- [CA] Certificado gerado com sucesso: ./certs/server.crt
- [Servidor] Certificado gerado com sucesso.
- Servidor HTTPS rodando em 127.0.0.1:8443
- Conectado ao servidor HTTPS em 127.0.0.1:8443
- Conexão recebida de ('127.0.0.1', 48812)
- Protocolo TLS usado: TLSv1.3
- Cipher suite utilizada: TLS_AES_256_GCM_SHA384
- Tamanho da chave: 256 bits
- Resposta do servidor (html_0.html):
- HTTP/1.1 200 OK
- Content-Type: text/html
- "Mensagem do HTML recebido"
- Resposta do servidor (html_1.html):

- HTTP/1.1 200 OK
- Content-Type: text/html
- “Mensagem do HTML recebido”

5 Repositório da Implementação

Para mais informações sobre a implementação, confira o repositório em: https://github.com/EduardoMarciano/SC2