UNIVERSIDADE FEDERAL DE JUIZ DE FORA

FACULDADE DE ENGENHARIA

ENGENHARIA ELÉTRICA - HABILITAÇÃO EM ROBÓTICA E AUTOMAÇÃO INDUSTRIAL

Luiggi Arthur Bernardino e Oliveira

Segurança de dados no protocolo de comunicação MQTT

Juiz de Fora

2022

Luiggi Arthur Bernardino e Oliveira

Segurança de dados no protocolo de comunicação MQTT

Trabalho de conclusão de curso apresentado ao Departamento de Energia Elétrica da Universidade Federal de Juiz de Fora como requisito para aprovação na disciplina - Trabalho de Final de Curso.

Orientador: Prof. Dr. Guilherme Márcio Soares

Juiz de Fora

2022

Ficha catalográfica elaborada através do Modelo Latex do CDC da UFJF
com os dados fornecidos pelo(a) autor(a)

Oliveira, Luiggi Arthur Bernardino e.
Segurança de dados no protocolo de comunicação MQTT / Luiggi Arthur Bernardino e Oliveira. – 2022.
66 f. : il.

Orientador: Guilherme Márcio Soares
Trabalho de Conclusão de Curso de Graduação – Universidade Federal de Juiz de Fora, Faculdade de Engenharia. Engenharia Elétrica - Habilitação em robótica e automação industrial, 2022.

1. Internet das coisas. 2. MQTT. 3. Segurança de dados. I. Soares,Guilherme M., orient. II. Título.

Luiggi Arthur Bernardino e Oliveira

Segurança de dados no protocolo de comunicação MQTT

Trabalho de conclusão de curso apresentado ao Departamento de Energia Elétrica da Universidade Federal de Juiz de Fora como requisito para aprovação na disciplina - Trabalho de Final de Curso.

BANCA EXAMINADORA

Prof. Dr. Guilherme Márcio Soares - Orientador
Universidade Federal de Juiz de Fora

Prof. Dr. Daniel Discini Silveira
Universidade Federal de Juiz de Fora

Prof. Dr. Leandro Rodrigues Manso Silva
Universidade Federal de Juiz de Fora

AGRADECIMENTOS

A Deus, pela minha vida, e por me permitir cumprir os objetivos encontrados ao longo da graduação.

Aos meus pais, pelo apoio incondicional e incentivo nos momentos difíceis.

Aos amigos, por todo apoio e companheirismo ao longo de toda a graduação.

Ao meu orientador, Guilherme, por todo o conhecimento compartilhado e pelo suporte durante a elaboração do trabalho de conclusão de curso.

Aos professores membros da banca, por aceitarem ler e contribuir com este trabalho.

Aos professores que tive durante a graduação, pelos ensinamentos e pelo aprendizado compartilhado.

A todos aqueles que contribuíram, de alguma forma, para a realização deste trabalho.

"Foi o tempo que dedicastes à tua rosa que a fez tão importante"

Antoine de Saint-Exupéry

RESUMO

Este trabalho evidencia os principais aspectos relacionados à segurança de dados com foco na área da internet das coisas detalhando o uso do protocolo de comunicação *MQ Telemetry Transport* (MQTT) e sua implementação, analisando possíveis falhas e vulnerabilidades que podem ser exploradas por invasores que buscam capturar dados ou prejudicar o usuário e como mitigar as possibilidades de ataques e invasões nesse contexto. Foram feitos testes de penetração em cenários sem implementação de camadas de segurança, com utilização de autenticação, com criptografia somente de mensagem e com uso de criptografia SSL/TLS (do inglês "*Secure Sockets Layer/Transport Layer Security*") demonstrando possíveis brechas na segurança e os dados que poderiam ser capturados em cada um deles, além disso, foi medido quanto cada uma dessas camadas acrescenta no tamanho dos pacotes enviados pelo protocolo MQTT.

Palavras Chave: Internet das coisas. MQTT. Segurança de dados.

ABSTRACT

This work highlights the main aspects related to data security with a focus on the internet of things, detailing the use of the Message Queuing Telemetry Transport (MQTT) communication protocol and its implementation, analyzing possible flaws and vulnerabilities that can be exploited by attackers seeking to capture data or to harm the user and how to mitigate those possible attacks and intrusions in this context. Penetration tests were made in scenarios without implementing security layers, using authentication layer, message-only encryption layer and using SSL/TLS (from english "Secure Sockets Layer/Transport Layer Security") encryption, demonstrating possible security breaches and the type of data that could be captured in each of the scenarios, in addition, it was measured how much each of these layers added to the size of the packets sent by the MQTT protocol.

Keywords: Internet of Things. MQTT. Data security.

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1 Introdução

Nos últimos anos a IoT (internet das coisas, do inglês *"internet of things"*) tornou-se cada vez mais importante e parte do nosso cotidiano, com a proposta de interconectar dispositivos para simplificar e facilitar tarefas, dispositivos e objetos que fazem uso da IoT estão presentes no dia a dia dos seres humanos coletando e transmitindo diversos tipos de dados. Uma das principais questões da tecnologia de IoT é a forma como esses dados são transmitidos através da rede, ou seja, qual o protocolo a ser utilizado para esta finalidade. Existem diversas possibilidades atualmente e o desenvolvedor deve buscar soluções prezando pela eficiência, confiabilidade e segurança.

A tecnologia IoT possibilita que aparelhos simples como luminárias, termômetros e aparelhos de som por exemplo, se conectem à internet para enviar e receber dados, tornando o monitoramento e o controle desses aparelhos muito mais prático. A variedade de dispositivos que podem ser utilizados com IoT possibilitou a formação de novos conceitos como o de casas inteligentes, cidades inteligentes e Indústria 4.0 que são a aplicação da IoT em cada uma dessas situações.

Os dispositivos IoT muitas vezes são utilizados em ambientes distantes, onde não há disponibilidade de rede elétrica, portanto, sendo alimentados através de baterias e por essa razão busca-se muitas vezes a eficiência energética para garantir seu funcionamento.

A confiabilidade, na indústria principalmente, é fator relevante pois garantir a disponibilidade de dados e consequentemente o funcionamento dos equipamentos ligados à IoT é crucial para a otimização de produção e diminuição de desperdícios.

A segurança dos dispositivos além de garantir que dados sensíveis não sejam compartilhados com qualquer um, engloba também a confiabilidade e a eficiência pois dispositivos podem sofrer ataques que afetam diretamente a disponibilidade de dados e ataques que causem uso desnecessário de energia.

Dessa forma, esse trabalho propõe uma solução com a implementação do protocolo MQTT, bem como a implementação de camadas de segurança e análise da proteção dos dados coletados e transmitidos no protocolo.

1.1 Justificativa do trabalho

A internet das coisas não possui um padrão definido de comunicação, sendo assim, vários protocolos são discutidos e utilizados de acordo com a necessidade das aplicações. Atualmente há uma gama de protocolos diferentes que atuam na camada de aplicação, sendo o protocolo MQTT e o CoAP (*Constrained Application Protocol*) exemplos bem populares.

Um dos temas de maior importância na IoT é a segurança, pois uma quantidade

significativa de dados é coletada e transmitida durante a comunicação desses protocolos e esses dados podem conter informações críticas para a segurança das pessoas, como localização por exemplo. Deseja-se então que essas informações não sejam interceptadas por pessoas não desejadas e por esse motivo a segurança deve ser um dos focos ao se escolher um protocolo.

Muitos métodos de segurança já bem conhecidos e utilizados não têm aplicabilidade nos sistemas com IoT devido às limitações de hardware que inviabilizam sua implementação, por isso há uma busca por soluções mais leves que cumpram a função de segurança como esses métodos convencionais.

Para este trabalho, foi escolhido retratar o protocolo MQTT, sua estrutura facilita a implementação e uso em ambientes onde a comunicação ocorre entre vários clientes simultaneamente, além disso, utiliza o protocolo TCP na camada de transporte, garantindo maior confiabilidade na ordem e entrega das mensagens.

1.2 Objetivo

Analisar e garantir a segurança dos dados transmitidos entre clientes e com o *broker* pelo protocolo *MQ Telemetry Transport* (MQTT) através da implementação e do uso de ferramentas de segurança como autenticação, autorização e criptografia, além da adoção de boas práticas de segurança no uso do protocolo.

1.3 Estrutura do Trabalho

O trabalho, além da Introdução, está estruturado em 4 capítulos, os quais são brevemente descritos a seguir:

- Capítulo 2 - Fundamentação Teórica: Apresenta os conceitos sobre os protocolos de comunicação com ênfase no MQTT, conceitos necessários para que o leitor possa compreender e extrair o máximo proveito do trabalho.
- Capítulo 3 - Avaliação de estratégias de segurança da informação utilizando o protocolo MQTT: Apresenta a aplicação da teoria apresentada na realização e obtenção de resultados práticos com a implementação de clientes e de camadas de segurança no protocolo MQTT e a avaliação da segurança obtida em cada uma.
- Capítulo 4 - Considerações finais: Apresenta a análise dos resultados obtidos no experimento e uma análise teórica do experimento na visão do autor.

1.4 Revisão bibliográfica

Estudos sobre protocolos de comunicação na IoT, tornam-se cada dia mais numerosos. Entre eles, destacam-se o trabalho de Nitin Naik [5] que faz comparativos de performance entre 4 diferentes protocolos de comunicação utilizados na IoT baseando-se em evidências empíricas de estudos anteriores.

O estudo de S. Quincozes, E. Tubino, e J. Kazienko [6] faz um comparativo entre os protocolos MQTT e CoAP através de experimentação utilizando a ferramenta de simulação Cooja que permite a captura e medição do tempo de resposta no envio de mensagens nos dois protocolos, na simulação foram enviadas 500 mensagens considerando ainda os 3 níveis QoS do MQTT.

O trabalho de Fernando Camargo de Andrade [7] implementa uma criptografia para o protocolo MQTT e analisa sua viabilidade considerando o tempo de envio das mensagens com e sem a implementação da criptografia proposta.

O artigo de Bernardo Martins Costa em [8] detalha sobre o funcionamento da criptografia TLS.

O estudo sobre segurança e teste de penetração no protocolo MQTT de [9] utiliza a ferramenta *Metasploit* para explorar vulnerabilidades e demonstrar possíveis brechas na segurança do protocolo MQTT.

2 Fundamentação Teórica

2.1 Internet das coisas

À medida em que a tecnologia evolui e se desenvolve, passamos a ter cada vez mais formas de comunicação, conexão e transmissão de dados através dos mais diversos tipos de ferramentas, após o advento e a evolução da internet chega-se à internet das coisas onde os variados dispositivos formam uma rede de compartilhamento de dados através da internet, a IoT (*internet of things*), ou, internet das coisas, inclui desde dispositivos para uso doméstico quanto uso industrial ou comercial. Para a utilização da Internet das coisas, faz-se necessário o uso de protocolos de comunicação que possuam algumas características específicas, em sistemas embarcados essas características variam de acordo com a função e necessidade desses sistemas. Gasto de energia, velocidade de transferência de dados e custo são exemplos. A seguir será mostrado alguns dos principais protocolos de comunicação conhecidos.

2.2 Protocolos de comunicação

Nessa sessão serão discutidos os principais protocolos da camada de aplicação utilizados na comunicação dos dispositivos IoT.

2.2.1 *MQ Telemetry Transport* (MQTT)

É um protocolo de mensagem de publicação/assinatura projetado para comunicação M2M (*machine to machine*). Foi desenvolvido pela IBM (*International Business Machines Corporation*) e hoje é *open standard*, ou seja, está disponível gratuitamente para adoção, implementação e atualizações.

O *MQ Telemetry Transport* (MQTT) é formado por um servidor e por clientes, o servidor é conhecido como *broker* e usa comunicação TCP (Protocolo de Controle de Transmissão, do inglês "*Transmission Control Protocol*"). O *broker* é orientado a mensagens, cada mensagem é publicada em um tópico. Os clientes conseguem subscrever em cada tópico e cada cliente subscrito recebe todas as mensagens do tópico. A função do *broker* é atuar como intermediário entre os clientes, gerenciando as publicações e subscrições em cada tópico, dessa forma há um desacoplamento entre os clientes, ou seja, a comunicação pode ser feita sem que um cliente conheça o outro.

O protocolo MQTT suporta QoS (qualidade de serviço, do inglês "*quality of service*") com três níveis de serviço. Eles são "*at most once*", "*at least once*"e "*exactly once*", esses níveis de serviço serão detalhados em 2.3.1.3. Quando um cliente se inscreve em um tópico, qualquer mensagem publicada neste tópico será enviada ao cliente.

Ao contrário da fila de mensagens, os *brokers* MQTT não permitem mensagens salvas arquivadas no servidor. Por segurança, o MQTT pode exigir autenticação de usuário e senha para clientes se conectarem.

2.2.2 *Constrained Application Protocol* (CoAP)

Feito pelo grupo IETF CoRE (*Constrained RESTful Environments) Working Group* para uso de número limitado de serviços, o *Constrained Application Protocol* (CoAP) usa pacotes pequenos e comunicação UDP (Protocolo de Datagrama do Usuário, do inglês "*user datagram protocol*"). O modelo utilizado é o de cliente/servidor e tem suporte tanto para a forma requisição/resposta e publicação/assinatura. Este protocolo foi criado para funcionar em conjunto com o HTTP (Protocolo de Transferência de Hipertexto, do inglês "*Hypertext transfer protocol*"), ele não utiliza o sistema de tópicos como o MQTT, mas sim o URI (*uniform resource identifier*) onde o publicador publica dados e o assinante assina um determinado recurso indicado pelo URI. Quando um publicador publica novos dados no URI, todos os assinantes são notificados sobre o novo valor conforme indicado pelo URI. O QoS deste protocolo consiste em dois níveis de serviço, sendo eles: confirmado e não confirmado.

2.2.3 *Advanced Message Queuing Protocol* (AMQP)

É um protocolo M2M(*machine-to-machine*) leve, de mensagens corporativas projetado para confiabilidade, segurança, provisionamento e interoperabilidade desenvolvido em 2003 por John O'Hara no JPMorgan Chase em Londres.

O *Advanced Message Queuing Protocol* (AMQP) oferece suporte às arquiteturas de requisição/resposta e publicação/assinatura. Ele oferece recursos relacionados para mensagens, como um enfileiramento confiável, publicação baseada em tópicos, mensagens e assinatura, roteamento flexível e transações.

O protocolo troca mensagens de várias maneiras: diretamente, em forma de *fanout*, por tópico ou com base em cabeçalhos. Ele é um protocolo binário e normalmente requer cabeçalho fixo de 8 bytes, mensagem com cargas úteis pequenas até o tamanho máximo dependente do corretor/servidor ou a tecnologia de programação. Utiliza o TCP como um protocolo de transporte padrão e TLS/SSL e SASL (Camada Simples de Autenticação e Segurança, do inglês *Simple Authentication and Security Layer*) para segurança, possui dois níveis preliminares de QoS para entrega de mensagens: *Unsettle Format* (não confiável) e *Settle Format* (confiável) [5].

2.2.4 *Hypertext transfer protocol* (HTTP)

Originalmente inventado por Tim Berners-Lee e sua equipe, o HTTP é a base do modelo cliente-servidor utilizado para a comunicação na internet. Ele oferece suporte

a arquitetura *request/response RESTful Web architecture*. Como acontece no CoAP, utiliza-se o *Universal Resource Identifier* (URI) no lugar de tópicos. O HTTP adota o formato de texto ao invés do binário utilizado nos outros citados até agora. O protocolo TCP é utilizado como protocolo de transporte e o SSL/TLS (do inglês "*Secure Sockets Layer/Transport Layer Security*") como segurança. O HTTP consome bastante energia e tempo comparado a alguns outros protocolos como o CoAP e foi pensado para comunicação entre 2 sistemas ao mesmo tempo apenas.

2.2.5 Comparação

O estudo [5] classifica características como consumo de energia vs. requisito de recursos, largura de banda vs. latência e tamanho da mensagem vs. cabeçalho da mensagem dos protocolos HTTP, AMQP, CoAP e MQTT através de gráficos, nos quais entre esses 4 protocolos o HTTP e AMQP se posicionam mais distantes das características desejadas considerando as limitações existentes em grande parte dos dispositivos IoT.

Esse estudo utiliza como base componentes estáticos e evidências empíricas da literatura, com o resultado podendo variar de acordo com as circunstâncias.

O CoAP e o MQTT são dois exemplos de protocolos populares e que se aproximam mais dessas características segundo [5], porém, deve-se atentar às diferenças entre eles, o CoAP utiliza o protocolo UDP em sua camada de transporte, sendo assim não deve ser utilizado em aplicações que necessitem de confiabilidade na ordem de chegada das mensagens. Outro ponto importante é a diferença entre a comunicação, o MQTT possui uma comunicação centralizada no *broker*, que pode se comunicar com vários clientes ao mesmo tempo, já o CoAP funciona com o modelo de comunicação 1:1, ou seja, um dispositivo pode se comunicar com apenas um dispositivo por vez. Dessa forma, a escolha entre os protocolos deve ser feita buscando o que melhor se adapte a aplicação em questão.

Para este trabalho, foi escolhido retratar o protocolo MQTT que utiliza o protocolo TCP na camada de transporte, dando maior confiabilidade na ordem e entrega das mensagens.

A Tabela 1 classifica os protocolos de acordo com as características vistas em [5] em ordem crescente, onde o 1º é o menor, através dela é possível observar um menor gasto de energia vs. consumo de recursos no CoAP e no MQTT e a maior Confiabilidade/QoS vs. Interoperabilidade no MQTT por exemplo.

Tabela 1 – Comparativo entre os protocolos

Protocolo	1º	2º	3º	4º
Consumo de energia vs. requisito de recursos	CoAP	MQTT	AMQP	HTTP
Largura de banda vs. latência	CoAP	MQTT	AMQP	HTTP
Tamanho da mensagem vs. cabeçalho da mensagem	CoAP	MQTT	AMQP	HTTP
Confiabilidade/QoS vs. Interoperabilidade	HTTP	CoAP	AMQP	MQTT

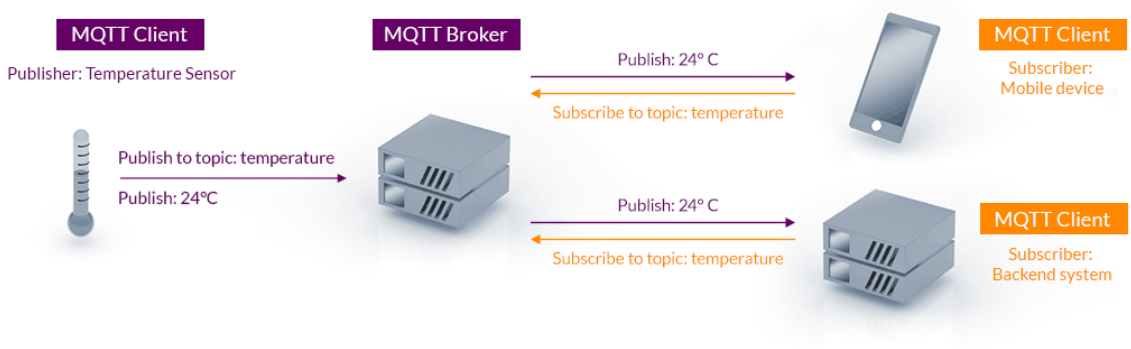
2.3 *MQ Telemetry Transport*

O protocolo MQTT foi desenvolvido em 1999 por Andy Stanford-Clark (IBM) e Arlen Nipper (Arcom, agora Cirrus Link), que necessitavam de um protocolo leve e de implementação simples capaz de utilizar o mínimo de energia e largura de banda para a comunicação entre satélites e sensores em oleodutos. Em 2010 o protocolo passou a ser livre de royalties e em 2014 foi padronizado pela OASIS (Organização para o Avanço de Padrões em Informação Estruturada, do inglês "*Organization for the Advancement of Structured Information Standards*"). Atualmente é um dos protocolos mais utilizados nas aplicações de internet das coisas.

2.3.1 Funcionamento

Seu funcionamento baseia-se no modelo publicação/assinatura e é composto por clientes e o servidor, também conhecido como *broker*. Clientes se inscrevem em tópicos gerenciados pelo *broker* e a partir do *broker* recebem todas as mensagens publicadas por clientes nos tópicos em que se inscreveram, ou seja, toda comunicação entre clientes é feita de maneira indireta pois passa necessariamente pelo *broker* que filtra essa comunicação e separa quais clientes devem receber as mensagens. O esquema de clientes e *broker* no MQTT pode ser visto na figura 1.

Figura 1 – Esquema de clientes e *broker* no MQTT [1]



2.3.1.1 *Broker* MQTT

Como dito anteriormente, o servidor utilizado no protocolo MQTT é conhecido como *broker*. Hoje, há muitas opções de *brokers* comerciais e *brokers open source* no mercado. Dentre os *brokers open source*, temos exemplos como Mosquitto, Emqttd e o HiveMQ. Entre os *brokers* comerciais temos o Azure, CloudMQTT e o AWS.

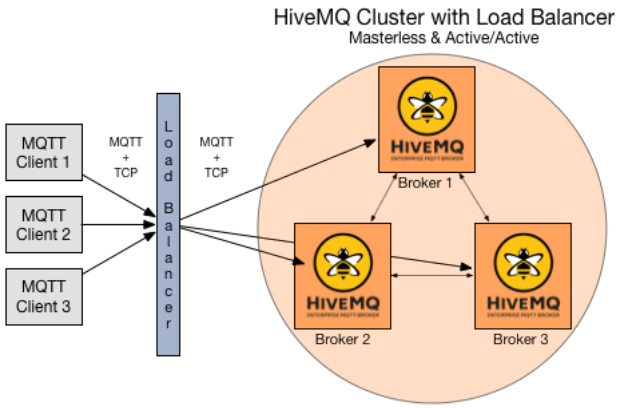
Uma das características mais importantes do modelo publicação/assinatura utilizado no *broker* é o desacoplamento entre os clientes publicadores e subscritores, com isso, não há necessidade de que os clientes conheçam um ao outro, o *broker* pode ser configurado

para guardar a última mensagem, assim não é necessário que os clientes funcionem ao mesmo tempo também em alguns casos.

Devido a centralização que ocorre no *broker*, o MQTT pode ser altamente escalável com funcionamento de vários clientes em paralelo pois o peso do processamento de muitos clientes ocorrerá somente no *broker*. Apesar das vantagens, essa centralização da comunicação no *broker* representa também um risco, uma vez que há uma dependência do sistema nele, soluções práticas para isso envolvem *brokers* redundantes ou operando em *clusters*.

Um *cluster* é um sistema distribuído que representa um *broker* MQTT lógico, ou seja, um *cluster* de *brokers* se comporta como um único *broker* MQTT do ponto de vista dos clientes. Ele consiste em muitos nós diferentes, normalmente instalados em máquinas físicas diferentes e conectados em uma rede [10]. Usualmente esses *clusters* são utilizados juntamente com um *Load Balancer*, que tem como função encontrar os endereços dos *brokers* para os clientes e balancear a carga entre os *brokers* para que não haja sobrecarga em algum dos *brokers*, além disso, muitas vezes a encriptação SSL/TLS é utilizada diretamente no *Load Balancer*, o que diminui o uso de recursos nos *brokers* [11]. A figura 2 exemplifica um esquema utilizando *cluster* e *Load Balancer* em conjunto.

Figura 2 – *Cluster* com utilização de um *Load Balancer* [2]



2.3.1.2 Sistema de mensagens

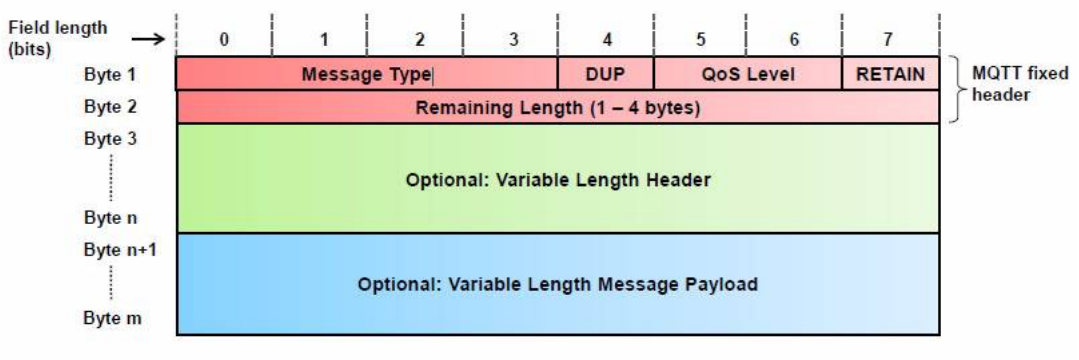
A identificação das mensagens no MQTT se dá através de tópicos. O tópico tem níveis separados por barras (“/”) e não precisam ter sido criados previamente para que se envie uma publicação. Há ainda alguns artifícios que podem ser utilizados para a filtragem de informações no *broker*.

- O “+” aceita qualquer valor naquele nível do tópico;

- O “#” significa “qualquer coisa abaixo de determinado nível do tópico”;
- O “$” representa tópicos especiais sendo geralmente reservados para uso interno do *broker* [12];

O cabeçalho do MQTT varia entre 2 a 5 bytes, sendo o primeiro obrigatório. Como pode ser visto na Figura 3, no primeiro byte, os bits de 0 à 3 referem-se ao tipo de mensagem, o bit 4 refere-se ao indicador de mensagem duplicada, o 5 e 6 ao nível de QoS e o 7 se a mensagem será retida ou não. Os próximos 4 bytes definem o tamanho do resto do pacote, juntos compõem o cabeçalho fixo do MQTT.

Figura 3 – Cabeçalho do protocolo MQTT [3]



O cabeçalho variável, onde não há um padrão estabelecido pode receber os seguintes componentes da tabela 2, de acordo com o comando enviado.

2.3.1.3 Qualidade de serviço (QoS)

No MQTT as mensagens enviadas possuem 3 níveis de QoS (*quality of service*) que definem a confiabilidade da mensagem enviada, essa definição é feita pelo cliente ao enviar a mensagem ou se subscrever em um tópico, o *broker* utiliza a mesma configuração do cliente na comunicação com ele. Os três níveis são:

QoS 0 (*at most once*): Neste nível não se tem confirmação de entrega de mensagem. A mensagem não é armazenada para futuras retransmissões.

QoS 1 (*at least once*): Neste nível há confirmação de entrega de uma mensagem. Atende situações onde quem envia acaba gerando várias mensagens iguais possivelmente por um atraso na chegada de confirmação de recebimento. Neste caso, é garantido que uma delas terá o reconhecimento realizado. A mensagem é armazenada por parte de quem envia até a posterior confirmação (PUBACK).

QoS 2 (*exactly once*): Neste nível garante-se a entrega da mensagem exatamente uma vez. Existem confirmações nos dois sentidos para o que é trafegado. Enquanto uma mensagem não é confirmada, ela é mantida.

Tabela 2 – Comando de mensagens [13]

Valor	Nome	Direção	Descrição
0	Reservado	Proibido	Reservado
1	CONNECT	Cliente para Servidor	Requisição do Cliente para conectar ao Servidor
2	CONNACK	Servidor para Cliente	Reconhecimento da conexão
3	PUBLISH	Cliente para Servidor ou Servidor para Cliente	Publicar mensagem
4	PUBACK	Cliente para Servidor ou Servidor para Cliente	Reconhecimento da publicação
5	PUBREC	Publicação recebida	Publicação recebida (parte 2 do QoS=1)
6	PUBREL	Cliente para Servidor ou Servidor para Cliente	Publicação lançada (parte 2 do QoS=2)
7	PUBCOMP	Cliente para Servidor ou Servidor para Cliente	Publicação completa (parte 3 do Qos =2)
8	SUBSCRIBE	Cliente para Servidor	Pedido de inscrição
9	SUBACK	Servidor para Cliente	Reconhecimento de inscrição
10	UNSUBSCRIBE	Cliente para Servidor	Pedido de designação
11	UNSUBACK	Servidor para Cliente	Reconhecimento de inscrição
12	PINGREQ	Requisição	Requisição PING
13	PINGRESP	Servidor para Cliente	Resposta PING
14	DISCONNECT	Cliente para Servidor	Cliente está desconectado
15	Reservado	Proibido	Reservado

Segundo S. Quincozes, E. Tubino, e J. Kazienko em [6], há maior tempo de resposta com níveis maiores de QoS.

2.3.1.4 Persistência

Ao se conectar ao *broker* um cliente tem a opção de criar uma sessão com persistência, sua função é manter salva informações relevantes para essa conexão, a sessão é identificada pelo *broker* através do *clientID* (identificador de cliente, do inglês, "*client identifier*"), que tem como função identificar cada cliente MQTT que se conecta a um *broker* MQTT.

As informações salvas são: Existência de uma sessão, as assinaturas do cliente, as mensagens de QoS 1 ou 2 que o cliente ainda não confirmou ou que o cliente perdeu enquanto estava offline, mensagens de QoS 2 recebidas do cliente que ainda não foram totalmente confirmadas.

2.3.1.5 Mensagens retidas

É possível utilizar uma *flag retained*, a mensagem salva com essa *flag* no *broker* é enviada sempre que o tópico for assinado. Essa característica permite que um dispositivo receba o valor publicado em um tópico que possua a *flag*, logo após inscrever-se nele.

2.3.1.6 *Last will and testament* (LWT)

É uma característica do protocolo MQTT que permite uma mensagem ser enviada a um tópico caso um dispositivo se desconecte acidentalmente, essa mensagem possui a forma padrão contendo tópico, *flag retained*, QoS e conteúdo especificado pelo dispositivo. A mensagem fica armazenada no *broker* até que ele detecte que o dispositivo perdeu a conexão, então envia a mensagem para todos os dispositivos que estejam inscritos no tópico especificado na mensagem. Esta mensagem também é descartada se o dispositivo se desconectar utilizando uma mensagem "DISCONNECT".

2.3.1.7 *Keep alive*

Garante que a conexão entre o *broker* e o cliente ainda esteja aberta e que o *broker* e o cliente estejam cientes de que estão conectados. Quando o cliente estabelece uma conexão com o *broker*, o cliente comunica ao *broker* um intervalo de tempo em segundos que define o período máximo de tempo que o *broker* e o cliente podem não se comunicar. Se o cliente não enviar uma mensagem durante o período de *keep alive*, ele deve enviar um pacote PINGREQ ao *broker* para confirmar que está disponível e para certificar-se de que o *broker* ainda está disponível. O *broker* deve desconectar um cliente que não envia uma mensagem ou um pacote PINGREQ em uma vez e meia o intervalo *keep alive*. Da mesma forma, espera-se que o cliente feche a conexão se não receber uma resposta do *broker* em um período de tempo razoável.

2.4 Segurança no MQTT

O conceito de internet das coisas envolve a conexão de cada vez mais dispositivos, objetos dos mais variados como sensores, televisões, *smartphones* e carros. Todas essas conexões envolvem a utilização de dados e com o avanço da tecnologia, cada vez mais dispositivos e dados são compartilhados. Muitos desses dados contêm informações sensíveis que podem colocar em risco os usuários e por isso a segurança de dados é uma das áreas mais importantes da internet das coisas. Devido a utilização do MQTT ser direcionada a necessidade de uso mínimo de energia, baixa capacidade de memória e processamento, muitas vezes a utilização de segurança é negligenciada pois acaba afetando a performance dos dispositivos IoT.

2.4.1 Objetivos de segurança

O objetivo de segurança da Internet das Coisas é fornecer uma conexão confiável, mecanismos de autenticação adequada e confidencialidade sobre os dados para cada dispositivo conectado em toda a rede. A tríade de segurança da informação, é definida como Confidencialidade, Integridade e Disponibilidade [14]. Ameaças e violações em

qualquer uma dessas áreas pode causar sérios danos ao sistema e ter impacto direto no trabalho do sistema.

- Confidencialidade - É a capacidade de garantir a privacidade do usuário oferecendo uma conexão segura apenas para usuários permitidos.
- Integridade - A integridade de dados é integrada à rede para proteger os dados de cibercriminosos durante a comunicação, para evitar que a adulteração de dados aconteça sem que o sistema perceba e capture a ameaça. O *Checksum* e a verificação de redundância cíclica são métodos de detecção de erros usados para verificar a integridade dos dados. Sincronização contínua de dados para fins de *backup* de uma versão controle também é usado.
- Disponibilidade - A disponibilidade diz que o acesso imediato de dados dos recursos por seu usuário em quaisquer condições deve ser garantido. Os *firewalls* são incorporados à rede para impedir ataques aos serviços como DDOS (negação de serviço distribuído, do inglês "*distributed denial of service*") que tem como alvo negar a disponibilidade dos dados para o usuário final.

2.4.2 Superfície de ataque

A superfície de ataque é a soma de vulnerabilidades, caminhos ou métodos, às vezes chamados de vetores de ataque, que os *hackers* podem usar para obter acesso não autorizado à rede ou a dados sensíveis, ou para realizar um ciberataque [15]. O aumento do número de dispositivos IoT, complexidade, heterogeneidade, expandem as superfícies de ataque nas redes de IoT. De acordo com [16] a superfície de ataque pode se dividir em dois ramos da rede IoT. O primeiro é rede local, onde as entidades são: controlador de IoT, *gateway* de IoT, coordenador de IoT e objetos inteligentes. o segundo são as redes públicas que consistem em controlador IoT, serviços IoT, *gateway* IoT e *Cloud*. A tabela 3 mostra a divisão completa da superfície de ataque.

Tabela 3 – Superfície de ataques [15]

Rede	Superfície de ataque
Rede local	Dispositivo - Dispositivo
Rede local	Dispositivo - Coordenador
Rede local	Dispositivo - *Gateway*
Rede local	Dispositivo - Controlador
Rede local	Dispositivo - Provedor de serviços Iot
Rede pública	Serviço - Serviço

2.4.3 Ferramentas de segurança do MQTT

O MQTT possui alguns recursos que podem ser implementados com a função de aumentar a segurança do protocolo. Ao implementar esses recursos deve-se considerar caso a caso quais ferramentas serão implementadas devido a limitação de recursos do cliente e do *broker* já citadas anteriormente.

2.4.3.1 Autenticação com nome de usuário e senha

O protocolo MQTT fornece campos de nome de usuário e senha na mensagem CONNECT para autenticação. O cliente tem a opção de enviar um nome de usuário e senha quando se conecta a um *broker* MQTT. Por padrão não há utilização dessa autenticação, ela deve ser configurada no *broker*, uma configuração de senha forte garante maior segurança, há a possibilidade de criptografar o usuário e senha no envio do cliente para o *broker* para evitar ataques comuns onde esses dados são capturados durante a comunicação.

2.4.3.2 Autenticação com *clientID*

Cada cliente MQTT possui um identificador de cliente único chamado de *clientID*. O cliente fornece esse ID exclusivo ao *broker* na mensagem CONNECT e esse ID é utilizado para criar e manter uma sessão. O ID do cliente pode ter no máximo 65.535 caracteres. Uma maneira comum de confirmar se um cliente pode acessar o *broker* MQTT é validar o nome de usuário/senha e o ID do cliente corretos para essa combinação de credenciais. É possível configurar a autenticação com somente o ID do cliente, no entanto, esse método não é uma boa prática de segurança pois facilita o acesso de invasores.

2.4.3.3 Autorização

Um cliente pode publicar ou assinar tópicos quando se conecta a um *broker*, caso não seja configurado algum tipo de autorização, ele poderá publicar e assinar todos os tópicos disponíveis. Para restringir um cliente a publicar ou assinar apenas tópicos autorizados as permissões de tópico devem ser implementadas no lado do *broker*. Essas permissões precisam ser configuráveis e ajustáveis durante o tempo de execução do *broker*. As permissões incluem tópicos permitidos, operações permitidas (publicar, assinar, ambos) e nível de QoS permitido. Uma forma de se fazer essa limitação no *broker* é a utilização de uma ACL (listas de controle de acesso, do inglês, "*access control list*") que define uma lista de permissões para um recurso.

2.4.3.4 Tamanho da mensagem

O MQTT define um tamanho máximo de mensagem de 256 MB. Na maioria dos cenários de implantação do MQTT, as mensagens são menores que um *kilobyte*. Por

questões de segurança, em cenários onde se sabe o tamanho máximo da mensagem que pode ocorrer, é recomendado diminuir o tamanho máximo permitido para esse limite, isso busca evitar ataques maliciosos que enviem mensagens grandes (o que pode resultar em consumo excessivo de memória e de largura de banda).

2.4.4 Ferramentas de segurança adicionais ao MQTT

2.4.4.1 Criptografia da mensagem

A criptografia de mensagem no MQTT é a criptografia de dados na camada de aplicação. Essa abordagem permite a criptografia de ponta a ponta dos dados, mesmo em ambientes não confiáveis. Esse tipo de criptografia não está definido na especificação MQTT e é exclusivo da aplicação. Há duas formas de criptografia nesse modelo, cliente-cliente e cliente-servidor.

- Cliente-Cliente: Nesse modelo de encriptação, o *broker* não possui nenhuma ferramenta de desencriptação, ou seja, somente os clientes com a chave serão capazes de desencriptar a mensagem. Neste caso não é possível encriptar as informações da mensagem como *clientID*, senha e tópicos por exemplo, porém não há acréscimo no processamento do *broker*, todo trabalho é feito nos clientes.

- Cliente-servidor: Nesse caso, é necessário que haja ferramentas capazes de encriptar e desencriptar a mensagem no *broker*, assim é possível encriptar tópico, senhas e outras informações da mensagem. Há três possibilidades no caso cliente-servidor.

  Somente os dados transmitidos pelo publicador são criptografados, o *broker* desencripta e transmite as informações em texto puro.

  Somente os dados transmitidos do *broker* para os subscritores são codificados, o publicador e o *broker* se comunicam sem criptografia.

  Toda a comunicação é criptografada, o publicador transmite a informação codificada, o *broker* decodifica e criptografa novamente para enviar aos subscritores [7].

2.4.4.1.1 Integridade de dados

As verificações de integridade têm como função garantir que terceiros não tenham modificado o conteúdo das mensagens transmitidas no MQTT.

As mensagens PUBLISH podem conter uma assinatura digital/MAC/*checksum* que verifica seu conteúdo,o cliente que receber esse conteúdo pode verificar a integridade dos dados recalculando/validando, isso garante que a mensagem não foi adulterada por terceiros.

O *checksum* garante que os dados não foram modificados de forma não intencional, porém caso um terceiro mal intencionado saiba qual o algoritmo utilizado no *checksum*, ele poderá modificá-lo junto com a mensagem para passar despercebido pela verificação.

MAC (Algoritmos de código de autenticação de mensagem, do inglês "*message authentication code*") são normalmente muito rápidos em comparação a assinaturas digitais. O cálculo do MAC é feito com uma função *hash* e uma chave criptográfica. Somente remetentes que conheçam a chave secreta podem assinar e verificar a integridade da mensagem.

A função *hash* é um algoritmo matemático que transforma qualquer bloco de dados em uma série de caracteres de comprimento fixo. Independentemente do comprimento dos dados de entrada, o mesmo tipo de *hash* de saída será sempre um valor *hash* do mesmo comprimento.As funções de *hash* criptográficas são usadas para garantir a integridade da mensagem. Pode-se garantir que a mensagem ou um arquivo não seja adulterado através de uma examinação do *hash* criados antes e depois da transmissão de dados. Se os dois *hash* são idênticos, garante-se que não houve adulteração [17].

Assinaturas digitais usam criptografia de chave pública/privada. O remetente assina a mensagem com sua chave privada e o destinatário valida a assinatura com a chave pública do cliente remetente. Apenas a chave privada pode criar a assinatura. Não é possível para um invasor falsificar a assinatura sem a chave privada. Usualmente é utilizada junto com a autorização para garantir o conhecimento entre destinatário e remetente.

2.4.4.2 Criptografia SSL/TLS

O protocolo TLS tem a função de aumentar a segurança durante a comunicação de dados através da criptografia, ele é independente do protocolo de aplicação, ou seja, consegue estabelecer a conexão entre cliente e servidor independente da forma que foram construídos. O TLS possui suporte para extensões futuras, isso evita a necessidade de criar novas bibliotecas de segurança. O TLS utiliza processos criptográficos diferentes como a criptografia de chave pública para fornecer autenticação e a criptografia de chave secreta com funções de *hash* para fornecer privacidade e integridade de dados.

A função principal da criptografia é tornar o conteúdo transmitido durante a comunicação entre grupos inteligível para usuários não autorizados através de algoritmos de encriptação, esses algoritmos são divididos em criptografia de chave secreta e criptografia de chave pública. Uma chave é uma *string* de bits utilizada pelos algoritmos de encriptação para criptografia, somente a chave correta é capaz de desencriptar uma mensagem criptografada.

O protocolo pode ser unilateral, onde somente o servidor é autenticado e o cliente permanece anônimo, ou, bilateral, onde cliente e servidor são autenticados garantindo maior segurança à conexão. (Alguns tipos de ataque forjam o certificado e buscam descobrir

usuário e senha do cliente comprometendo o acesso aos dados comunicados).

Para garantir a integridade e autenticidade de todas as mensagens transferidas, os protocolos SSL e TLS também contam com um processo de autenticação usando códigos de autenticação de mensagem (MAC).

O protocolo TLS fica entre as camadas de Aplicação e Transporte. Ele encapsula os protocolos de aplicação e trabalha em cima de um protocolo de transporte, no caso do MQTT, usualmente o TCP. A tabela 4 mostra a posição do protocolo TLS na estrutura do modelo de camadas.

Tabela 4 – Estrutura do MQTT no modelo de camadas [13]

Estrutura do Protocolo no modelo de camadas
Aplicação (MQTT)
TLS
Transporte (TCP)
Rede
Enlace
Física

O TLS pode ser dividido em duas camadas, a primeira consiste em 4 sub-protocolos, o protocolo de *handshaking*, protocolo ChangeCipherSpec, o protocolo de alerta e o próprio protocolo de aplicação. A segunda camada consiste no protocolo de registro.

2.4.4.2.1 Protocolo de Registro (*Record Protocol*)

Fragmenta as mensagens em blocos e então adiciona a eles um código MAC (*Message Authentication Code*), encripta e transmite o resultado. Ele também é organizado em camadas. Em cada uma delas, as mensagens podem incluir campos com o tamanho da mensagem, a descrição e o conteúdo [13]. Os protocolos de *handshaking*, protocolo de aplicação, protocolo ChangeCipherSpec e o protocolo de alerta utilizam o protocolo de registro.

2.4.4.2.2 Protocolo de *handshaking*

É o protocolo responsável por negociar uma sessão, que consiste nos seguintes itens: Identificador de sessão, certificado de par, método de compressão, especificação de cifra, segredo mestre.

Ao iniciar uma conexão entre servidor e cliente, estabelece-se a versão do protocolo que está sendo utilizada, os algoritmos criptográficos e é feita a autenticação um do outro. O protocolo implementa isso através da troca de mensagens de "*Hello*", da troca de parâmetros de criptografia e da troca de certificados visando a autenticação dos pares.

Certificado de chave pública: É emitido por uma organização confiável e fornece identificação para o portador. Uma organização confiável que emite certificado de chave pública é conhecida como CA (Autoridade de Certificação, do inglês, "*certification authority*"). Para obter um certificado de uma CA, deve-se fornecer prova de identidade, quando a CA confirma a identidade do requerente, a CA assina o certificado atestando a validade das informações contidas no certificado [13].

2.4.4.2.3 Protocolo ChangeCipherSpec

Notifica os lados quando há mudança no método de criptografia, uma única mensagem encriptada e comprimida é enviada para ambos os lados da conexão para notificar que o novo método será utilizado. Caso o outro lado esteja executando alguma operação computacionalmente custosa, a mensagem poderá ser armazenada num *buffer* por um período curto, até que a operação termine e as novas especificações sejam utilizadas [13]. O fluxograma na figura 4 mostra a sequência do protocolo TLS.

2.4.4.2.4 O Protocolo de Alerta (*Alert Protocol)*

As mensagens do protocolo de alerta se dividem entre avisos e erros fatais e são compostas pelas suas devidas descrições. Alertas fatais resultam em fechamento imediato da conexão, tornando o identificador da sessão inválido, evitando então que esta sessão seja utilizada em novas conexões. Essas mensagens de alerta também são comprimidas e encriptadas para serem transmitidas.

Em casos de alertas de fechamento, ambos os lados da conexão devem estar cientes que há intenção de fechar a conexão para evitar alguns tipos de ataques. Qualquer uma das partes pode iniciar a troca de mensagens de fechamento [13].

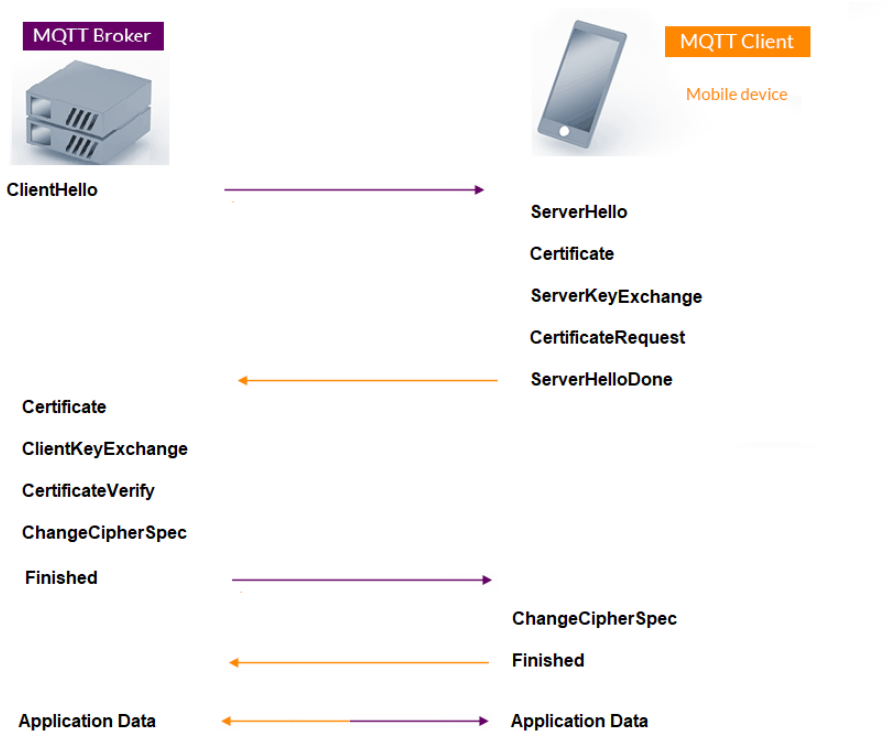
A mensagem close_notify notifica o destinatário de que o remetente não enviará mais mensagens nesta conexão, quaisquer dados recebidos após um alerta de fechamento são ignorados e a menos que algum outro alerta fatal tenha sido transmitido, cada parte deve enviar um alerta close_notify antes de fechar o lado de escrita da conexão. A outra parte responde com um alerta close_notify próprio para a conexão imediatamente, nesse caso qualquer escrita pendente é descartada.

A figura 4 demonstra os passos do funcionamento no protocolo TLS.

2.4.4.2.5 Criptografia de chave pública

A criptografia de chave pública também é conhecida como criptografia assimétrica. Consome alto poder de processamento sendo utilizado para pequenos pacotes de mensagem. Exemplo de algoritmo: RSA (Rivest Shamir Adleman).

Figura 4 – Fluxograma do protocolo TLS



O algoritmo Rivest Shamir Adleman (RSA) é muito utilizado para criptografar mensagens. Ele é composto por um par de chaves, uma pública e a outra privada. A chave pública é de conhecimento de todos os lados, a chave privada não é compartilhada e é mantida apenas por um deles. Um dos lados divulga a chave pública que será utilizada para que o outro lado da conexão faça a encriptação da mensagem, somente a chave privada desse par é capaz de decriptar essa mensagem, ou seja, somente o lado que divulga a chave pública terá capacidade de decriptar a mensagem [8].

2.4.4.2.6 Criptografia de chave secreta

É também conhecida como criptografia simétrica. Neste caso, ambos os grupos utilizam a mesma chave para encriptar e desencriptar as mensagens. Fornece boa segurança e criptografia de forma relativamente rápida. No momento em que a chave é compartilhada com o outro grupo deve-se atentar ao risco de o invasor capturar essa chave, caso ocorra, a criptografia é comprometida. Exemplos de algoritmos: *Advanced Encryption Standard* (AES), *Triple Data Encryption Standard* (3DES), e *Rivest Cipher* 4 (RC4) [18].

2.4.4.2.7 Certificado X509

São os certificados dos clientes, possuem uma chave pública e uma chave privada únicas que podem ser utilizadas durante o *handshake* do TLS para autenticação do cliente, garantindo maior segurança no processo. Como essa verificação é feita antes de estabelecer

a comunicação entre cliente e *broker*, na camada de transporte, ela pode poupar recursos do *broker* ao evitar conexões inválidas.

A dificuldade no uso desse certificado se dá pela necessidade de prover certificados para todos os clientes que irão se conectar, o que se torna um processo difícil caso o cliente seja desconhecido ou nunca tenha se conectado antes.

2.4.5 Principais ataques

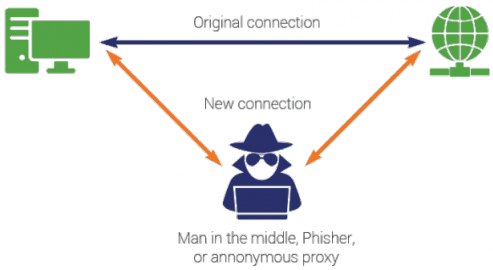
Os ataques ao MQTT podem ser divididos em alguns tipos:

- Privacidade de dados - Como por padrão o MQTT não utiliza criptografia, caso não seja implementada de alguma forma, mesmo utilizando mecanismos de autenticação um invasor ainda consegue roubar dados durante a comunicação.
- Autenticação - Caso o invasor esteja conectado na mesma rede do publicador, ele poderá ter acesso aos dados de usuário e senha utilizados na autenticação através do pacote MQTT CONNECT, conseguindo assim permissão para publicar ou subscrever.
- Integridade de dados - O invasor que já possui conhecimento dos pacotes de dados, modifica os dados em trânsito. Nesse cenário é possível modificar mensagens e controlar dispositivos.
- Segurança por obscuridade - O número da porta definido pelo IANA (Autoridade para Atribuição de Números da Internet, do inglês "*Internet Assigned Numbers Authority*") utilizado pelo MQTT é 1883 para o MQTT sem criptografia e 8883 para o MQTT com SSL / TLS. Porém o *broker* pode ser configurado para utilizar outras portas. Caso utilize apenas a segurança padrão do MQTT, o invasor ainda pode usar filtros de dados em softwares como *Wireshark* e verificar as mensagens de CONNECT filtrando apenas por MQTT.

2.4.5.1 Ataques

Man in the Middle Attack (MitM) - Ocorre quando o invasor se posiciona entre a comunicação do *broker* e cliente, nessa situação o invasor pode interceptar as mensagens, alterá-las e se passar por uma das partes na comunicação sem que as partes percebam. É possível para o invasor conseguir informações sensíveis como dados de autenticação de usuário e senha, as mensagens podem ser direcionadas para outro usuário ou a transmissão de dados entre remetente e destinatário pode ser interrompida. Tanto autenticação quanto encriptação são necessários para evitar ataques MitM. O uso do protocolo SSL/TLS não impede totalmente o ataque do tipo MitM pois o invasor pode explorar técnicas para interceptar a conexão e descriptografar as mensagens caso o usuário esteja em uma rede pública ou não segura. A figura 5 exemplifica o esquema desse ataque.

Figura 5 – Esquema de ataque *Man in the Middle* [4]



Intrusão - Uma intrusão de rede refere-se a qualquer atividade não autorizada em uma rede digital. As invasões de rede geralmente envolvem o roubo de recursos valiosos da rede e quase sempre colocam em risco a segurança das redes e/ou seus dados [19]. Ataques de intrusão no MQTT podem utilizar a combinação entre portas mais utilizadas pelo MQTT e o comando "#"para se inscrever em todos os tópicos possíveis e adquirir informações sobre eles.

Denial of Service Attack (DoS) - Um ataque de negação de serviço tem como objetivo deixar a máquina, sistema ou rede indisponível para os usuários através de sobrecarregamento do mesmo. No caso do MQTT são enviadas continuamente múltiplas requisições de conexões ao *broker* causando sobrecarga e impedindo -o de novas requisições de conexões causando a negação de serviço. No MQTT isso acontece, pois, após receber a requisição de conexão através da mensagem CONNECT, o *broker* precisará enviar a mensagem CONNACK para reconhecer a conexão com o cliente. Uma variação desse ataque é o DDoS, onde as requisições são enviadas de múltiplas fontes distintas. A figura 6 exemplifica a diferença entre o Dos e DDoS.

Figura 6 – Diferença entre um ataque DoS e DDos

Autenticação à força bruta - *Brute Force Authentication* (BTA) tem como alvo o sistema de autenticação do *broker*. Visando obter os dados de autenticação o algoritmo criado testa combinações de autenticação até conseguir acesso ao usuário e senha corretos, esse modelo de ataque explora a vulnerabilidade de combinações de senhas fracas.

Negação de Serviço via ID de Cliente Duplicado - Quase todas as implementações de MQTT têm o mesmo comportamento quando se trata de duplicar IDs de clientes. Quando um cliente se conecta ao servidor, se o ID do cliente, apresentado pelo novo cliente que enviou esta solicitação CONNECT, for idêntico ao que estiver em uso por outro cliente que está conectado ao servidor, o servidor desconecta o cliente conectado e permite que o novo cliente se conecte. Esse comportamento tem o potencial de causar um DoS na rede MQTT. As informações necessárias são a lógica para gerar IDs de clientes e um *script* que gere os IDs do cliente e se conecte ao *broker* para efetivamente expulsar os clientes atualmente conectados, causando um MQTT DDoS [20].

ID do cliente utilizado para autenticação/controle de acesso - Às vezes os IDs do cliente são usados para autenticar um cliente ou fornecer acesso a tópicos confidenciais. Nesse caso, usando as mesmas técnicas acima, pode-se obter acesso à rede MQTT e receber ou manipular dados confidenciais.

Clonagem de Cliente - Dependendo se a autenticação e uso de credenciais/chaves é implementado ou não, é possível falsificar o cliente legítimo e manipular os dados de telemetria como se fossem do cliente real ou receber informações confidenciais destinadas ao legítimo cliente.

Ataque ao aplicativo por meio de entrada mal-intencionada - Dependendo de como os dados serão processados no lado do aplicativo, se os dados de telemetria contiverem uma mensagem criada com códigos maliciosos, eles poderão explorar o aplicativo. Supondo que o invasor tenha acesso à rede MQTT e possa publicar no mesmo tópico, um *payload* XSS (do inglês "*Cross-site Scripting*") pode ser publicado, esse *script* obrigará o navegador a executá-lo se determinados dados não forem filtrados pelo aplicativo.

XSS é um tipo de ataque de injeção de código malicioso em aplicações web, classificado entre as principais vulnerabilidades no OWASP Top 10 2017 (Projeto Aberto de Segurança em Aplicações Web, do inglês "*Open Web Application Security Project®*").

2.4.5.2 Boas práticas de segurança no MQTT

Todo dispositivo IoT está sujeito a ataques e invasões que podem comprometer dados ou até aparelhos em si, como hoje em dia os dados têm cada vez mais relevância e importância deve-se ficar atento à segurança dos dispositivos IoT. Para mitigar riscos, algumas práticas podem ser adotadas no protocolo MQTT e em sua configuração.

- Atualização de firmware

  A princípio deve-se manter o *firmware* dos dispositivos IoT sempre atualizados para evitar que vulnerabilidades e problemas já corrigidos possam ser explorados por invasores. Dito isso, nem todo IoT possui suporte para atualização de *firmware* e apenas alguns fornecedores oferecem suporte à atualização automática de *firmware*.

Outra preocupação é que a forma de atualização de *firmware* deve ser feita de forma segura, protegida de invasores que podem tentar incluir *malwares* a fim de explorar vulnerabilidades durante a atualização de *firmware*.

- Configurações

  Uma medida básica a ser tomada durante a configuração é evitar utilizar a porta padrão do MQTT que não utiliza a criptografia por TLS. Ao utilizar essa porta, o dispositivo torna-se mais vulnerável por ser uma porta comumente utilizada durante a implementação do MQTT, ela é muito visada em tentativas de invasões.

  Outra medida é a implementação de autenticação por usuário e senha que por padrão não é um requisito do protocolo MQTT. Dispositivos que não possuem a autenticação do cliente podem ter o *broker* acessado por qualquer cliente anônimo que conseguirá se inscrever e publicar em tópicos, conseguindo acesso às informações compartilhadas nele, bem como controlar o dispositivo e usar de ataques com spam de publicação de dados para provocar DDOS.

- Conexões

  Deve-se atentar às conexões e redes utilizadas para conectar os dispositivos IoT, dando preferência a redes privadas e conhecidas. Essa prática busca evitar ataques conhecidos como *Man in the Middle*, onde o invasor está conectado na mesma rede que a vítima.

3 Avaliação de estratégias de segurança da informação utilizando o protocolo MQTT

3.1 Descrição da prática

Neste capítulo será desenvolvida a implementação da comunicação entre clientes MQTT através do *broker* Mosquitto em um computador local, os clientes serão implementados na linguagem Python em um computador e em um ESP32 através do Arduíno IDE. Serão implementados os seguintes clientes:

- Cliente sem nenhum tipo de autenticação.
- Cliente com autenticação de usuário e senha.
- Cliente com criptografia de mensagem.
- Cliente com criptografia TLS.

  Após isso, testes de penetração utilizando as ferramentas *Wireshark*, *Network Mapper* e *Metasploit Framework* em, busca de possíveis vulnerabilidades. Os resultados esperados visam obter informações e dados dos clientes durante a comunicação MQTT dos mesmos e em sequência avaliar a segurança de cada uma dessas implementações de clientes demonstrando as falhas de segurança de cada implementação e possíveis soluções.

3.2 Ferramentas

Para a implementação do projeto proposto, são necessárias ferramentas como ambientes de desenvolvimento (IDE) para as linguagens Arduíno e para Python. Foi escolhido o Arduino IDE que será utilizado para a linguagem do Arduíno e o IDLE para Python, o terminal do sistema operacional para a execução de comandos específicos, um ESP32 atuando como cliente e um computador configurado com as ferramentas do Eclipse Mosquitto para atuar como *broker*.

- Computador: A máquina utilizada para o funcionamento do Mosquitto *broker* é um computador com sistema Windows 10, processador i5 10400f com 16gb de memória ram.
- ESP32: O módulo ESP32 é um módulo de alta performance para aplicações envolvendo WiFi, contando com um baixíssimo consumo de energia. É uma evolução do já conhecido ESP8266, com maior poder de processamento e *bluetooth* BLE 4.2 embutido. Com 4MB de memória *flash*, o ESP32 permite criar variadas aplicações para projetos de IoT, acesso remoto, *webservers* e *dataloggers*, entre outros[13].

- Mosquitto *Broker*: O Eclipse Mosquitto é um agente de mensagens de *software* livre que implementa o protocolo MQTT versões 5.0, 3.1.1 e 3.1. O Mosquitto é leve e adequado para uso em todos os dispositivos, desde computadores de placa única de baixa potência até servidores completos.

- Arduino IDE: Arduino IDE é um software de código aberto escrito na linguagem de programação Java utilizado para escrever e fazer *upload* de programas em placas compatíveis com Arduíno, sendo executado nos sistemas operacionais Windows, Macintosh e Linux.

- Python IDLE (Python 3.10 64-*bit*): IDLE é o Ambiente Integrado de Desenvolvimento e Aprendizagem do Python, funciona em Windows, Unix e macOS, é utilizado para a implementação de clientes na máquina durante o desenvolvimento.

- Biblioteca Paho Python: É uma biblioteca Python para MQTT que suporta Python 2.7 e 3.x. A biblioteca implementa uma classe de cliente que pode ser usada para adicionar suporte MQTT ao seu programa Python criando instâncias do cliente ou herdando com sua própria classe. Ele também fornece algumas funções auxiliares para tornar a publicação de mensagens únicas mais fácil.

3.3 Implementação

A seguir serão descritas as implementações do *broker* e dos clientes necessários para o desenvolvimento do trabalho.

3.3.1 *Broker*

O *broker* escolhido para ser implementado foi o Eclipse Mosquitto, *software* livre (licenciado por EPL/EDL) que implementa o protocolo MQTT versões 5.0, 3.1.1 e 3.1. O *software* pode ser baixado no site oficial mosquitto.org e instalado na máquina pessoal, após a instalação é necessário configurar os arquivos na pasta de instalação de acordo com a necessidade da aplicação, demonstrado em 3.3.2. O *broker* pode ser acessado diretamente do terminal e possui comandos específicos para configuração. Neste trabalho são utilizados os seguintes comandos no terminal:

- -v: verbose ou log detalhado;

- -c config.file: Carrega o arquivo .conf com configuração personalizada do *broker*.

  O arquivo .conf possui uma extensa lista de comandos para se definir a configuração do *broker*, os comandos utilizados neste trabalho estão explicitados na tabela 5.

Tabela 5 – Comandos do arquivo de configuração do Mosquitto *broker*

Comando	Entrada	Função
allow_anonymous	*true/false*	Permite conexões desconhecidas.
listener	<porta>	Seleciona a porta a ser utilizada.
per_listener_settings	*true/false*	Define se as senhas são exigidas globalmente ou por ouvinte
password_file	Caminho do arquivo .txt com a autenticação de usuário e senha	Seleciona a porta a ser utilizada.
cafile	Caminho do arquivo .crt do certificado do servidor	Define o arquivo contendo o certificado do servidor
capath	Caminho da pasta contendo o arquivo .crt do certificado do servidor	Define a pasta contendo o certificado do servidor
certfile	Caminho do arquivo .crt do certificado do cliente	Define o arquivo contendo o certificado do cliente
require_certificate	*true/false*	*true* exige do cliente o certificado válido

3.3.2 Clientes

Os clientes foram implementados usando as configurações sem autenticação, com autenticação, criptografia de mensagem e criptografia TLS no ESP32 como cliente subscritor e em um computador como cliente publicador, os códigos dos clientes estarão na sessão anexo. No *broker* há ainda a configuração para cada um desses clientes.

3.3.2.1 Clientes sem autenticação

É necessário devido o *default* false no parâmetro allow_anonymous quando não há *listener* definido na configuração.

listener 1883

per_listener_settings false

allow_anonymous true

3.3.2.2 Clientes com autenticação de usuário e senha

Para clientes com autenticação de usuário e senha é necessário indicar a porta a ser utilizada e indicar o arquivo contendo a configuração da senha.

listener 1884

per_listener_settings false

allow_anonymous false

password_file /etc/mosquitto/senha.txt

O arquivo senha.txt que contém a configuração da senha é composto apenas pela linha onde a primeira palavra define o nome de usuário e a segunda palavra define a senha, separados apenas pelo caractere ":"ficando teste:teste.

3.3.2.3 Clientes com criptografia de mensagem

Segue a mesma configuração de clientes sem autenticação em 3.3.2.1, caso seja implementado juntamente com autenticação segue a configuração de clientes com autenticação de 3.3.2.2.

3.3.2.4 Clientes com criptografia TLS

É preciso indicar a porta, o arquivo contendo a configuração de senha caso seja utilizada autenticação e a localização da pasta e dos arquivos contendo os certificados.

listener 8885

allow_anonymous false

cafile C: /etc /mosquitto /localhostcert /ca.crt

capath C: /etc /mosquitto /localhostcert

certfile C: /etc /mosquitto /localhostcert /server.crt

keyfile C: /etc /mosquitto /localhostcert /server.key

require_certificate true

password_file /etc/mosquitto/senha.txt

3.4 Testes de penetração

O teste de penetração é um exercício de segurança que visa encontrar e explorar vulnerabilidades em um sistema. O objetivo deste ataque simulado é identificar quaisquer pontos fracos nas defesas de um sistema dos quais os invasores possam se aproveitar. Utilizando alguns programas é possível escanear as portas utilizadas na rede local e retirar informações importantes que podem ser utilizadas em ataques e invasões.

3.4.1 Programas utilizados

Nmap ou *Network Mapper* é um utilitário gratuito e de código aberto para descoberta de rede e auditoria de segurança.

O *Wireshark* é um analisador de protocolo de rede. Ele permite que você veja o que está acontecendo em sua rede com detalhes.

Metasploit Framework é uma ferramenta para desenvolvimento e lançamento de *exploits* utilizada em auditorias e Teste de Invasão. O *framework* consiste em uma série de ferramentas, *exploits* e códigos que podem ser utilizados através de diferentes interfaces.

3.4.2 MQTT sem nenhuma autenticação

Os clientes no anexo 1883 foram utilizados para obtenção dos resultados neste caso. Quando a autenticação de clientes não é implementada, informações importantes como nome de tópicos e mensagens enviadas podem ser descobertas caso se tenha acesso a rede local do cliente ou *broker*. Um escaneamento da rede local com o Nmap utilizando o comando: nmap -sV -sC -p- -v 127.0.0.1 durante a comunicação dos clientes encontra as informações presentes na figura 7.

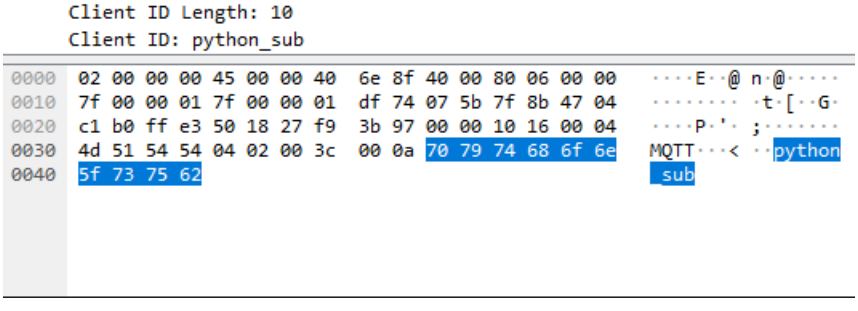
Figura 7 – Escaneamento da porta 1883 na ferramenta Nmap

```
1883/tcp  open     mosquitto version 2.0.14
| mqtt-subscribe:
|   Topics and their most recent payloads:
|     $SYS/broker/load/messages/sent/15min: 13.27
|     Topico_teste_Luiggi: sem senha
|     $SYS/broker/load/messages/received/15min: 13.56
|     $SYS/broker/bytes/received: 13704
|     $SYS/broker/load/bytes/sent/5min: 476.29
|     $SYS/broker/load/publish/received/15min: 12.65
|     $SYS/broker/version: mosquitto version 2.0.14
|     $SYS/broker/uptime: 1738 seconds
|     $SYS/broker/messages/sent: 428
```

No caso desenvolvido, são expostas informações como a porta utilizada na conexão, versão do *broker*, nome dos tópicos e conteúdo das mensagens enviadas. A partir dessas informações pode-se criar um cliente e se inscrever ou publicar em todos os tópicos de interesse. Isso demonstra a fragilidade da segurança ao não se utilizar nenhum mecanismo de segurança durante a implementação do protocolo MQTT.

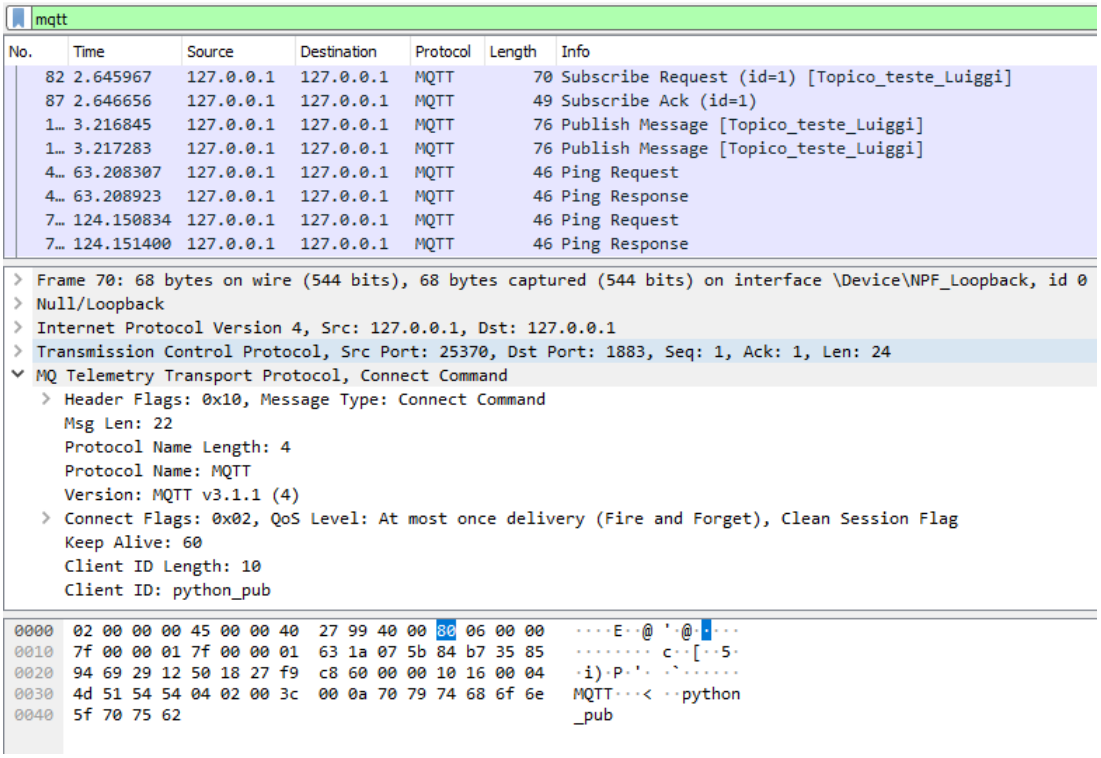
Utilizando a ferramenta *Wireshark* é possível descobrir até mesmo o *clientID* ao capturar os pacotes trafegando na rede. Ao utilizar o filtro “mqtt” para filtrar os dados recebidos da comunicação pelo protocolo MQTT obtém-se o *clientID* mostrado na figura 8.

Figura 8 – *clientID* obtido pelo *Wireshark*



A figura 9 mostra a tela completa do *Wireshark* após a captura dos pacotes. No exemplo do desenvolvimento o *clientID* utilizado é python_sub. Essa informação torna possível um ataque do tipo negação de serviço via ID de cliente duplicado visto em 2.4.5.1 e demonstrado em 3.4.2.1.

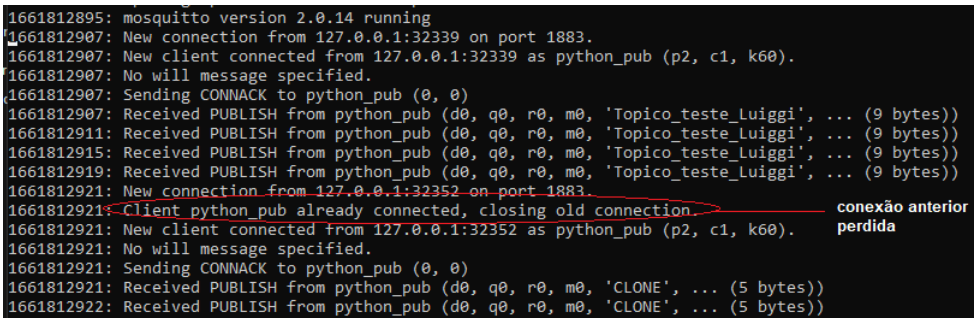
Figura 9 – Captura de pacotes no *Wireshark*



3.4.2.1 Ataque DoS

Ao utilizar um ID idêntico ao de um cliente que já está sendo executado é possível causar uma desconexão entre *broker* e o cliente, causando então a negação de serviço. Neste trabalho foi criado um cliente chamado Clone, que possui mesmo *clientID* e é executado para cancelar a conexão estabelecida anteriormente. A figura 10 destaca a conexão perdida com o cliente anterior no terminal do *broker* quando o cliente Clone é conectado.

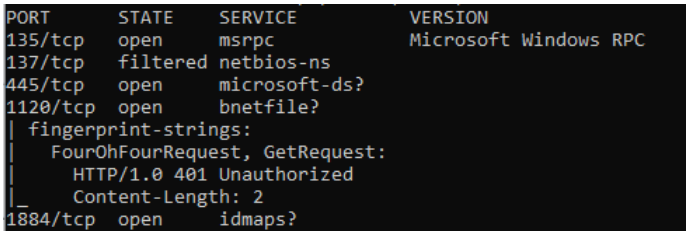
Figura 10 – Conexão fechada devido a execução de cliente com *clientID* idêntico



3.4.3 MQTT com autenticação de usuário e senha

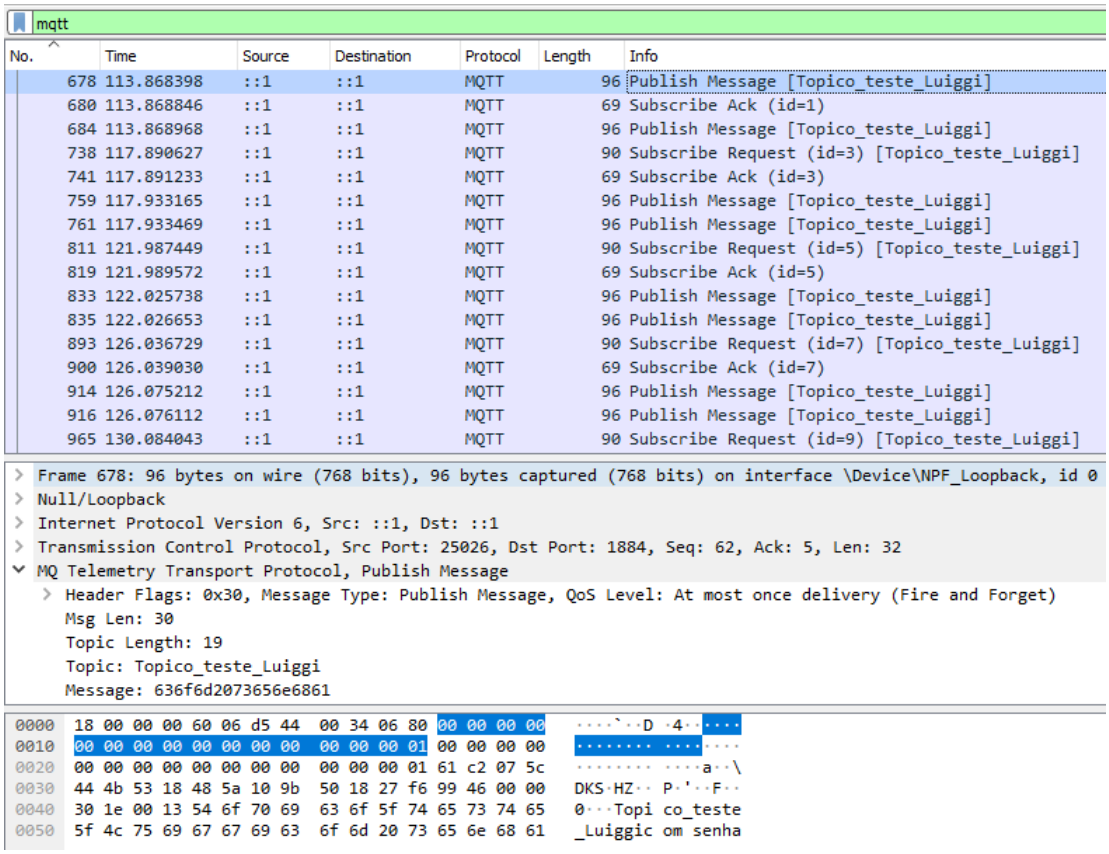
Utiliza-se o Nmap afim de verificar as portas em uso para em seguida configurar corretamente o *Wireshark* na tentativa de capturar pacotes com informações. A figura 11 exibe o resultado do escaneamento no Nmap indicando o uso da porta 1884.

Figura 11 – Escaneamento da porta 1884 no programa Nmap



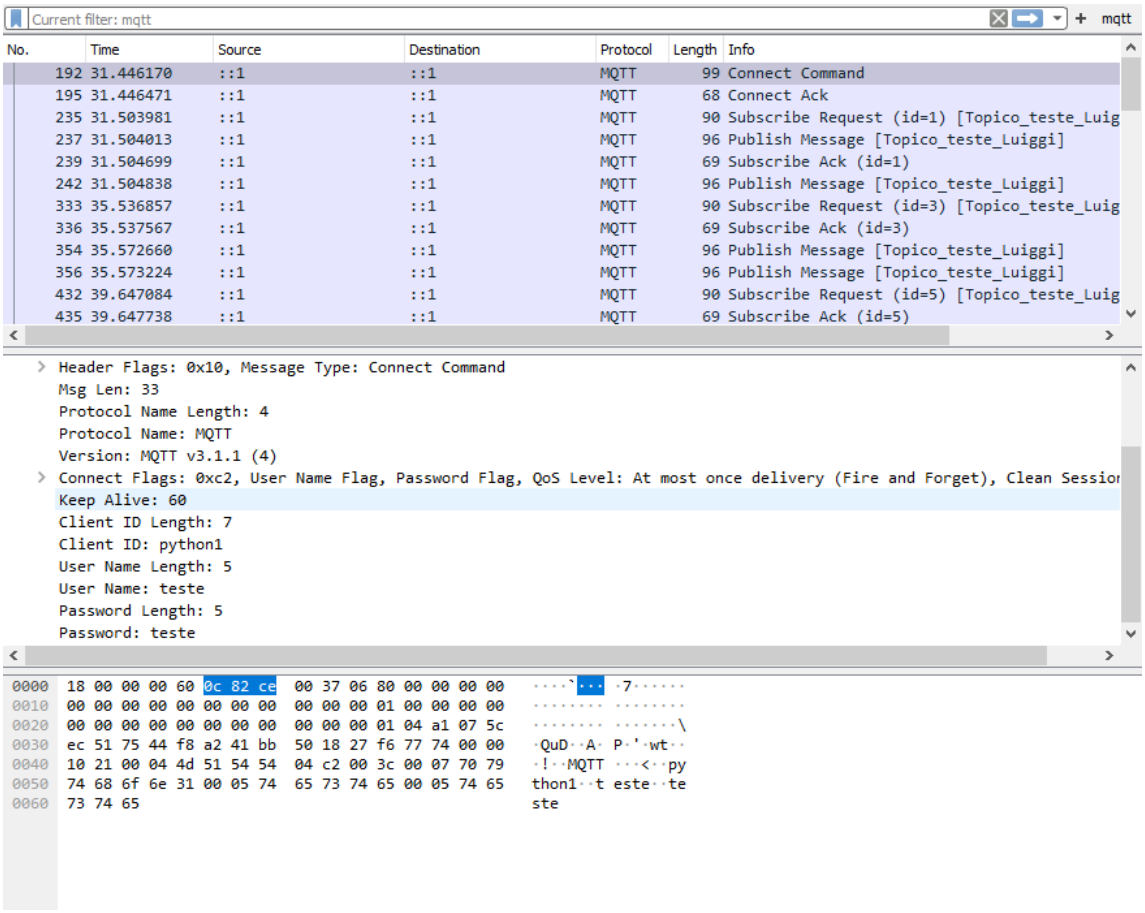
Com a autenticação através de usuário e senha corretamente implementada, os métodos acima tornam-se incapazes de capturar as informações de usuário e senha, embora ainda seja possível capturar o nome do tópico, mensagem e algumas outras informações como é mostrado na figura 12.

Figura 12 – Captura de pacotes no *Wireshark*



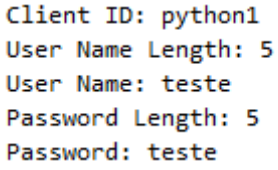
Porém no momento em que um pacote CONNECT é capturado, todas essas informações são encontradas e o invasor passa a ter acesso privilegiado através do nome de usuário e senha como anteriormente, mostrado na figura 13.

Figura 13 – Dados do pacote CONNECT revelados no *Wireshark*



Um zoom na imagem, indicado na figura 14 mostra que os campos *User Name* e *Password* que são respectivamente Nome de usuário e senha, ambos foram definidos como "teste"nas configurações mostradas em 3.3.2.2.

Figura 14 – Zoom nos dados do pacote CONNECT revelados



3.4.3.0.1 Ataque de Força bruta

Utilizando a ferramenta *Metasploit Framework* executa-se um ataque de força bruta conhecido como dicionário. Neste método, um arquivo contendo várias palavras são testados como usuário e senha até que se consiga a autenticação.

No *Metasploit Framework* selecione o módulo utilizado para o ataque com o comando: use auxiliary/scanner/mqtt/connect. Os comandos que configuram as informações necessárias para o ataque estão listados na tabela 6.

Tabela 6 – Lista de comandos utilizados no *Metasploit*

Comando	Entrada	Função
set RHOSTS	127.0.0.1	Define o IP que será alvo
set RPORT	1884	Define a porta que será alvo
set PASS_FILE	C:\passwords.txt	Seleciona o local do arquivo contendo a lista de palavras utilizadas para testar a senha.
set USER_FILE	C:\user.txt	Seleciona o local do arquivo contendo a lista de palavras utilizadas para testar o usuário.
set STOP_ON_SUCCESS	*True*	Para o escaneamento quando encontrar a senha e usuário corretos
exploit	N/A	Roda o programa com as configurações definidas acima

O sucesso desse tipo de ataque está relacionado à falta de complexidade da combinação de usuário e senha utilizados na configuração da autenticação, por isso recomenda-se não utilizar combinações simples ou que sejam muito utilizadas. A figura 15 mostra o sucesso do ataque ao obter o nome de usuário e senha do cliente.

Figura 15 – Ataque de força bruta bem sucedido

```
msf6 auxiliary(scanner/mqtt/connect) > exploit

[+] 127.0.0.1:1884        - MQTT Login Successful: teste/teste
[*] 127.0.0.1:1884        - Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf6 auxiliary(scanner/mqtt/connect) >
```

Uma solução para evitar esses casos de invasão é a utilização da criptografia TLS. Neste projeto foi desenvolvido um cliente utilizando a criptografia TLS em conjunto com a proteção já implementada de autenticação de usuário e senha através do ataque de força bruta.

3.4.4 MQTT com autenticação de usuário e senha e criptografia TLS

Utiliza-se o Nmap para verificar as portas em uso para em seguida com o *Wireshark* tentar capturar pacotes com informações. A figura 16 mostra o resultado obtido pelo escaneamento no Nmap, contendo informações sobre os certificados e a criptografia utilizada.

A porta utilizada nesse caso é a 8885, após configurar e filtrar o *Wireshark* para MQTT e porta 8885 obtemos o resultado mostrado na figura 17.

Figura 16 – Escaneamento da porta 8885 no programa Nmap

```
8885/tcp  open     ssl/unknown
|_ssl-date: TLS randomness does not represent time
| ssl-cert: Subject: commonName=DESKTOP-5P4804T/organizationName=Universidade/stateOrProvinceName=Minas/countryName=BR
| Issuer: commonName=DESKTOP-5P4804T/organizationName=UFJF/stateOrProvinceName=MG/countryName=BR
| Public Key type: rsa
| Public Key bits: 2048
| Signature Algorithm: sha256WithRSAEncryption
| Not valid before: 2022-08-22T15:53:43
| Not valid after:  2023-08-17T15:53:43
| MD5:   1357 9e67 f1cd 2862 2278 b20c a7c3 2f4a
|_SHA-1: 5ad5 791e f456 00e2 0787 75e4 8723 11dd d70b ec67
```

Figura 17 – Conteúdo dos pacotes protegidos durante captura no *Wireshark*

```
284 11.7025… fe80::d1d8:3e3a:f9ce:804a fe80::d1d8:3e3a:f9ce:804a TLSv1.3   122 Application Data
290 11.7038… fe80::d1d8:3e3a:f9ce:804a fe80::d1d8:3e3a:f9ce:804a TLSv1.3   122 Application Data
306 11.7074… fe80::d1d8:3e3a:f9ce:804a fe80::d1d8:3e3a:f9ce:804a TLSv1.3    88 Application Data
310 11.7080… fe80::d1d8:3e3a:f9ce:804a fe80::d1d8:3e3a:f9ce:804a TLSv1.3    88 Application Data

> Frame 284: 122 bytes on wire (976 bits), 122 bytes captured (976 bits) on interface \Device\NPF_Loopback, id 0
> Null/Loopback
> Internet Protocol Version 6, Src: fe80::d1d8:3e3a:f9ce:804a, Dst: fe80::d1d8:3e3a:f9ce:804a
> Transmission Control Protocol, Src Port: 3318, Dst Port: 8885, Seq: 2968, Ack: 5023, Len: 58
v Transport Layer Security
  v TLSv1.3 Record Layer: Application Data Protocol: Application Data
      Opaque Type: Application Data (23)
      Version: TLS 1.2 (0x0303)
      Length: 53
      Encrypted Application Data: 075d9131426303120cd99eb3d52f8a418ad12c7f6a3969faec0dafa485e2986385686fde…

0000  18 00 00 00 60 04 4b 3c  00 4e 06 40 fe 80 00 00   ····`·K< ·N·@····
0010  00 00 00 00 d1 d8 3e 3a  f9 ce 80 4a fe 80 00 00   ·····>: ···J····
0020  00 00 00 00 d1 d8 3e 3a  f9 ce 80 4a 0c f6 22 b5   ·····>: ···J··"·
0030  71 c2 0e e9 b5 24 7e d2  50 18 27 e2 98 8b 00 00   q····$~· P·'·····
0040  17 03 03 00 35 07 5d 91  31 42 63 03 12 0c d9 9e   ····5·]· 1Bc·····
0050  b3 d5 2f 8a 41 8a d1 2c  7f 6a 39 69 fa ec 0d af   ··/·A··, ·j9i····
0060  a4 85 e2 98 63 85 68 6f  de 27 19 a8 83 3e df 45   ····c·ho ·'···>·E
0070  95 02 cd b6 8f ef 79 51  fc 07                     ·····yQ ··
```

Como mostrado na figura 17, informações como tópico, mensagem e *clientID* estão criptografadas e não é possível observá-las ao usar o *wireshark*.

3.4.5 MQTT com criptografia de carga útil

Uma alternativa mais simples ao uso de TLS é criptografar somente a mensagem enviada com a intenção de proteger o conteúdo da mesma. Neste caso, mesmo sem autenticação de usuário e senha é possível proteger o conteúdo da mensagem transmitido durante a comunicação, porém, esse caso ainda é vulnerável a ataques como negação de serviço via ID de cliente duplicado e a captura dos dados de autenticação do usuário e senha como visto anteriormente pois somente a carga útil é encriptada nesta implementação. Para se encriptar os dados de usuário e senha é preciso que haja ferramentas de encriptação configuradas adequadamente tanto no cliente quanto no *broker*, o que torna a implementação mais complicada e menos viável em diversas situações.

A figura 18 mostra como as informações reveladas de um cliente sem configuração de autenticação e com criptografia da carga útil são capturadas pela ferramenta *Wireshark*. Observe como o *clientID* continua exposto, neste caso como "client-001".

Neste caso, observa-se pela figura 19 que o conteúdo da mensagem capturado pelo *Wireshark* está protegido pela criptografia.

Figura 18 – Informações de *clientID* no *Wireshark* mesmo com a encriptação

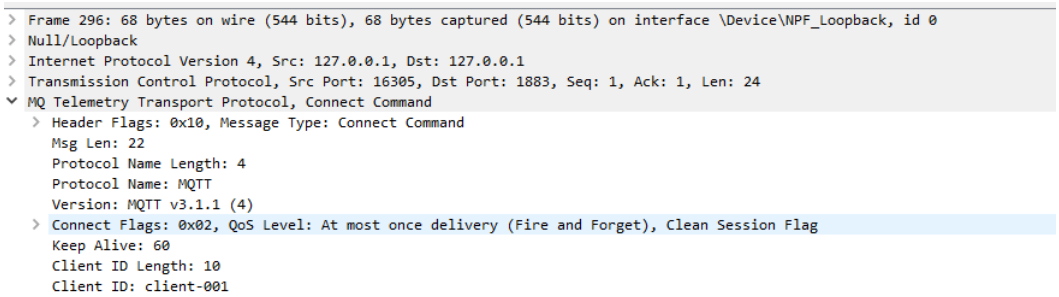
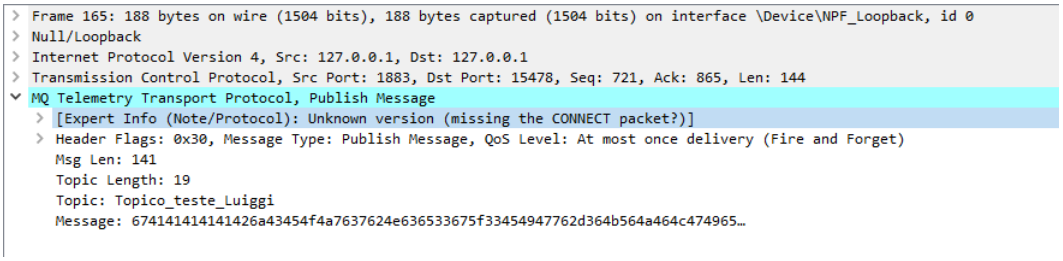


Figura 19 – Conteúdo da mensagem protegida com criptografia



3.5 Comparação entre os clientes implementados

Para o teste de penetração foram utilizadas 4 implementações de clientes: Cliente sem nenhum tipo de autenticação, cliente com autenticação de usuário e senha, cliente com criptografia de mensagem, cliente com criptografia TLS. Para comparar adequadamente o impacto de cada ferramenta implementada nesses clientes, é necessária uma padronização no *clientID*, mensagem enviada, nome do tópico e nome de usuário e senha nos casos em que há autenticação. Após adequar os clientes, foi feita a captura de pacotes durante o envio de uma única publicação de mensagem no tópico "Topico_teste_Luiggi"em cada um deles. As figuras mostram os pacotes capturados em cada caso.

Na figura 20 é possível observar 2 pacotes "*Publish Message*" isso ocorre devido os clientes serem subscritos no tópico, assim o cliente envia o pacote "*Publish Message*" e recebe do *broker* o mesmo pacote por estar subscrito no tópico em que foi enviado.

Figura 20 – Cliente sem nenhum tipo de autenticação

No.	Time	Source	Destination	Protocol	Length	Info
125	5.122267	::1	::1	MQTT	85	Connect Command
129	5.123365	::1	::1	MQTT	68	Connect Ack
137	5.125574	::1	::1	MQTT	90	Subscribe Request (id=1) [Topico_teste_Luiggi]
146	5.126543	::1	::1	MQTT	69	Subscribe Ack (id=1)
200	5.141742	::1	::1	MQTT	100	Publish Message [Topico_teste_Luiggi]
202	5.141779	::1	::1	MQTT	66	Disconnect Req
206	5.142325	::1	::1	MQTT	100	Publish Message [Topico_teste_Luiggi]

A figura 21 mostra que houve um acréscimo no tamanho do pacote de publicação em comparação ao anterior, isso se dá devido a criptografia da mensagem enviada no pacote.

No caso da figura 22, observa-se que não há aumento no pacote de publicação, mas

Figura 21 – Cliente com criptografia de mensagem

No.	Time	Source	Destination	Protocol	Length	Info
345	21.760540	::1	::1	MQTT	85	Connect Command
354	21.762670	::1	::1	MQTT	68	Connect Ack
362	21.764213	::1	::1	MQTT	90	Subscribe Request (id=1) [Topico_teste_Luiggi]
364	21.765132	::1	::1	MQTT	69	Subscribe Ack (id=1)
384	23.808799	::1	::1	MQTT	187	Publish Message [Topico_teste_Luiggi]
390	23.808851	::1	::1	MQTT	66	Disconnect Req
396	23.809676	::1	::1	MQTT	187	Publish Message [Topico_teste_Luiggi]

no pacote *connect*, devido a informações de usuário e senha necessárias presentes neste pacote.

Figura 22 – Cliente com autenticação de usuário e senha

No.	Time	Source	Destination	Protocol	Length	Info
91	5.903317	::1	::1	MQTT	99	Connect Command
95	5.905098	::1	::1	MQTT	68	Connect Ack
103	5.924002	::1	::1	MQTT	90	Subscribe Request (id=1) [Topico_teste_Luiggi]
112	5.925196	::1	::1	MQTT	69	Subscribe Ack (id=1)
179	6.032539	::1	::1	MQTT	100	Publish Message [Topico_teste_Luiggi]
181	6.032731	::1	::1	MQTT	66	Disconnect Req
187	6.033605	::1	::1	MQTT	100	Publish Message [Topico_teste_Luiggi]

A figura 23 mostra que há um número maior de pacotes capturados pelo *Wireshark*, isso ocorre devido ao protocolo TLS necessitar desses pacotes para iniciar a sessão como foi mostrado na figura 4.

Figura 23 – Cliente com autenticação de usuário e senha e criptografia TLS

No.	Time	Source	Destination	Protocol	Length	Info
54	4.606829	fe80::d1d8:3e3a:f9ce:804a	fe80::d1d8:3e3a:f9ce:804a	TLSv1.3	581	Client Hello
56	4.608861	fe80::d1d8:3e3a:f9ce:804a	fe80::d1d8:3e3a:f9ce:804a	TLSv1.3	2635	Server Hello, Change Cipher Spec, Application Data, Application Data, Application Data, Application Data, Application Data
58	4.611038	fe80::d1d8:3e3a:f9ce:804a	fe80::d1d8:3e3a:f9ce:804a	TLSv1.3	2409	Change Cipher Spec, Application Data, Application Data, Application Data
62	4.611495	fe80::d1d8:3e3a:f9ce:804a	fe80::d1d8:3e3a:f9ce:804a	TLSv1.3	1263	Application Data
64	4.611636	fe80::d1d8:3e3a:f9ce:804a	fe80::d1d8:3e3a:f9ce:804a	TLSv1.3	1263	Application Data
84	4.651231	fe80::d1d8:3e3a:f9ce:804a	fe80::d1d8:3e3a:f9ce:804a	TLSv1.3	121	Application Data
88	4.651716	fe80::d1d8:3e3a:f9ce:804a	fe80::d1d8:3e3a:f9ce:804a	TLSv1.3	90	Application Data
104	4.656443	fe80::d1d8:3e3a:f9ce:804a	fe80::d1d8:3e3a:f9ce:804a	TLSv1.3	112	Application Data
106	4.656644	fe80::d1d8:3e3a:f9ce:804a	fe80::d1d8:3e3a:f9ce:804a	TLSv1.3	91	Application Data
196	8.684400	fe80::d1d8:3e3a:f9ce:804a	fe80::d1d8:3e3a:f9ce:804a	TLSv1.3	122	Application Data
200	8.685799	fe80::d1d8:3e3a:f9ce:804a	fe80::d1d8:3e3a:f9ce:804a	TLSv1.3	122	Application Data
218	8.689284	fe80::d1d8:3e3a:f9ce:804a	fe80::d1d8:3e3a:f9ce:804a	TLSv1.3	88	Application Data
222	8.689840	fe80::d1d8:3e3a:f9ce:804a	fe80::d1d8:3e3a:f9ce:804a	TLSv1.3	88	Application Data

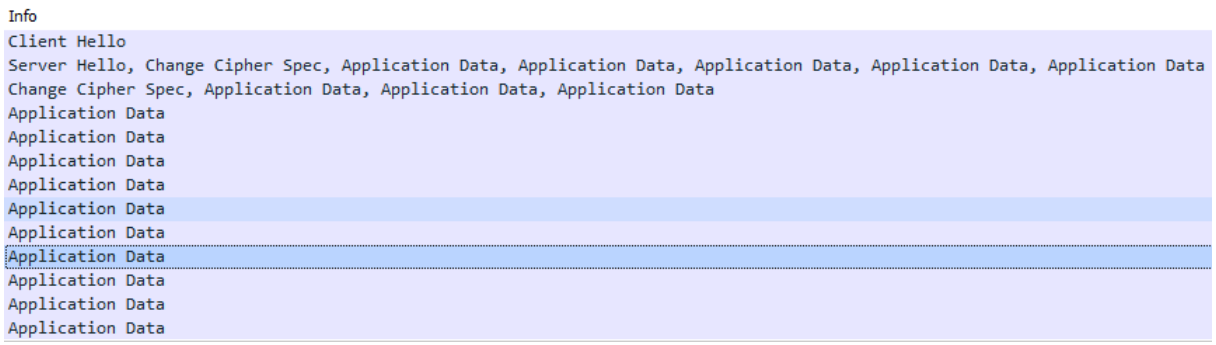
A figura 24 é um recorte da figura 23 para mostrar o tamanho do pacote que carrega a mensagem, no caso, 122 bytes.

Figura 24 – Cliente com criptografia TLS com zoom

No.	Time	Source	Destination	Protocol	Length	Info
133	7.594139	fe80::d1d8:3e3a:f9ce:804a	fe80::d1d8:3e3a:f9ce:804a	TLSv1.3	581	Client Hello
135	7.595350	fe80::d1d8:3e3a:f9ce:804a	fe80::d1d8:3e3a:f9ce:804a	TLSv1.3	2635	Server Hello, Cha
137	7.597544	fe80::d1d8:3e3a:f9ce:804a	fe80::d1d8:3e3a:f9ce:804a	TLSv1.3	2409	Change Cipher Spe
141	7.598034	fe80::d1d8:3e3a:f9ce:804a	fe80::d1d8:3e3a:f9ce:804a	TLSv1.3	1263	Application Data
143	7.598178	fe80::d1d8:3e3a:f9ce:804a	fe80::d1d8:3e3a:f9ce:804a	TLSv1.3	1263	Application Data
163	7.628273	fe80::d1d8:3e3a:f9ce:804a	fe80::d1d8:3e3a:f9ce:804a	TLSv1.3	121	Application Data
171	7.630017	fe80::d1d8:3e3a:f9ce:804a	fe80::d1d8:3e3a:f9ce:804a	TLSv1.3	90	Application Data
183	7.633954	fe80::d1d8:3e3a:f9ce:804a	fe80::d1d8:3e3a:f9ce:804a	TLSv1.3	112	Application Data
185	7.635260	fe80::d1d8:3e3a:f9ce:804a	fe80::d1d8:3e3a:f9ce:804a	TLSv1.3	91	Application Data
284	11.7025…	fe80::d1d8:3e3a:f9ce:804a	fe80::d1d8:3e3a:f9ce:804a	TLSv1.3	122	Application Data
290	11.7038…	fe80::d1d8:3e3a:f9ce:804a	fe80::d1d8:3e3a:f9ce:804a	TLSv1.3	122	Application Data
306	11.7074…	fe80::d1d8:3e3a:f9ce:804a	fe80::d1d8:3e3a:f9ce:804a	TLSv1.3	88	Application Data
310	11.7080…	fe80::d1d8:3e3a:f9ce:804a	fe80::d1d8:3e3a:f9ce:804a	TLSv1.3	88	Application Data

A figura 25 é um outro recorte da figura 23, ela mostra os passos do protocolo TLS ao iniciar uma sessão, descritos na figura 4.

Figura 25 – Passos do protocolo TLS em detalhes



A tabela 7 faz um comparativo entre os tamanhos de mensagem de cada um dos clientes implementados, ela mostra que a utilização de TLS gera um aumento significativo no tamanho total dos pacotes capturados e isso se deve aos pacotes necessários para iniciar uma sessão com o TLS, o pacote com comando PUBLISH, que carrega a mensagem, tem um aumento relativamente pequeno em comparação, ou seja, o envio de um número grande de mensagens pouco afeta a performance da comunicação em comparação a novas conexões.

Tabela 7 – Comparativo entre tamanho dos clientes

Clientes	Tamanha do pacote PUBLISH	Tamanho total
Cliente sem nenhum tipo de autenticação	100bytes	578bytes
Cliente com autenticação de usuário e senha	100bytes	592bytes
Cliente com criptografia de mensagem	187bytes	752bytes
Cliente com autenticação de usuário e senha e criptografia TLS	122bytes	8985bytes

3.5.1 Exemplo prático

Para validar o modelo proposto de implementação do cliente com TLS, o cliente com autenticação de usuário e senha e criptografia TLS foi implementado em um ESP32, para demonstrar seu funcionamento na prática este cliente subscreve no tópico "Topico_teste_Luiggi", seu funcionamento pode ser acompanhado tanto no terminal do *broker* quanto pela captura de pacotes no *Wireshark*. Para essa implementação é necessário utilizar algumas bibliotecas e funções explicadas a seguir.

ssl_client.h - Biblioteca Arduíno para adicionar a funcionalidade TLS a qualquer

classe de cliente. Implementa comunicação criptografada por meio de SSL em dispositivos que não a suportam.

WiFiClientSecure.h - Classe base que fornece SSL de cliente para ESP32.

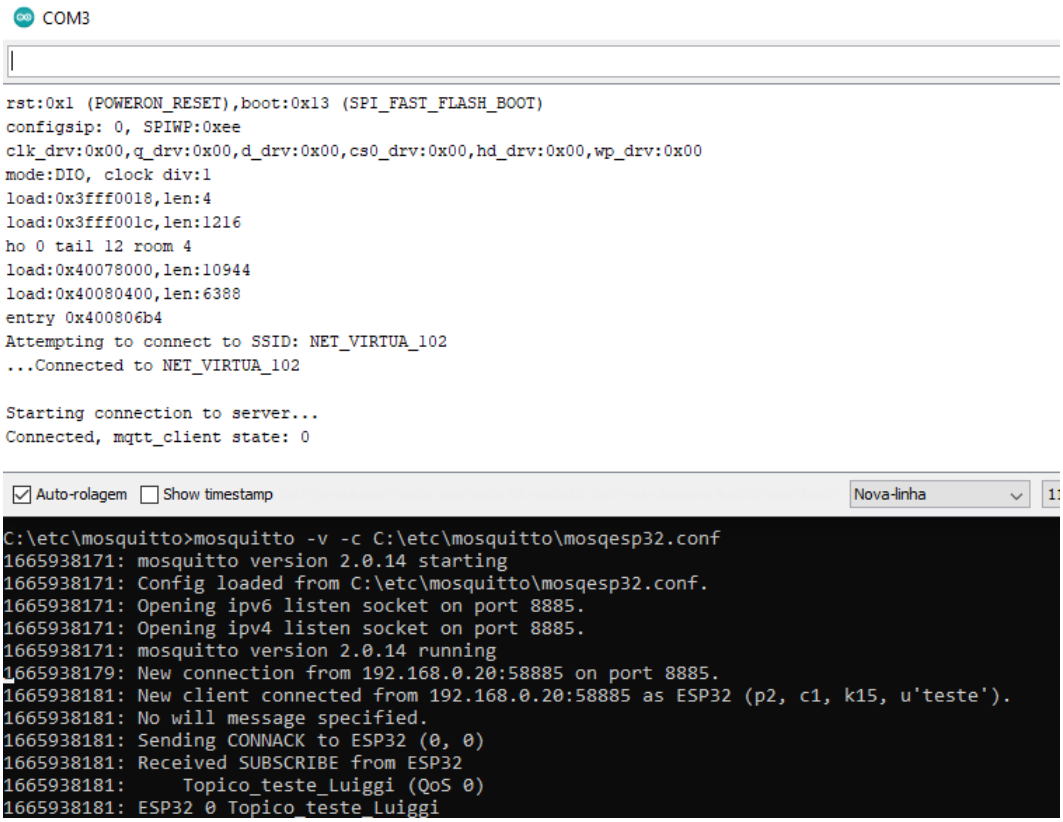
WiFi.h - Permite conexão de rede (local e Internet).

PubSubClient.h - Esta biblioteca permite enviar e receber mensagens MQTT.

O código implementado através do Arduíno IDE pode ser visto no anexo E.

A figura 26 mostra a tela *monitor serial* do Arduíno IDE e a tela do terminal do *broker* durante o funcionamento do cliente, a configuração do *broker* é a mesma descrita em 3.3.2.4.

Figura 26 – Funcionamento do cliente no ESP32



4 Considerações finais

Neste trabalho foi proposto e analisado algumas formas de transmitir informações através do protocolo MQTT de maneira segura por meio da combinação de implementações de ferramentas de segurança. Com essas ferramentas é possível aplicar um nível de segurança eficaz na comunicação entre clientes e *brokers* no mundo real, evitando que usuários indesejados acessem conteúdos privados de acordo com a aplicação IoT utilizada. A implementação proposta visa adequar cada uma das ferramentas de segurança à situação em que o protocolo MQTT venha a ser aplicado e busca configurar um nível de segurança suficiente sem que haja perda de performance que inviabilize a utilização do protocolo MQTT nas aplicações tendo em vista as limitações dos dispositivos utilizados na IoT. Os resultados obtidos em 3.5 que a utilização de criptografia TLS nos clientes causa um aumento significativo no tamanho total dos pacotes enviados durante a comunicação, porém, grande parte desse tamanho consiste na comunicação necessária para iniciar uma sessão, sendo assim é bastante viável em situações onde não há um número grande de sessões sendo iniciadas, pois a transmissão das mensagens pelos pacotes com comando PUBLISH tem um aumento pequeno.

REFERÊNCIAS

1 MQTT.ORG. *MQTT: The Standard for IoT Messaging*. 2022. Disponível em: <https://mqtt.org/assets/img/mqtt-publish-subscribe.png>.

2 HIVEMQ. *Creating highly available and ultra-scalable MQTT clusters*. 2016. Disponível em: <https://www.hivemq.com/img/blog/clustering_hivemq_w_lb.png>.

3 WORLD, R. W. *MQTT Tutorial | MQTT architecture, MQTT protocol use cases*. 2012. Disponível em: <https://www.rfwireless-world.com/images/MQTT-protocol-message-format.jpg>.

4 KAYMERA. *Mobile Critters: Part 1. Man In The Middle Attack*. 2021. Disponível em: <https://kaymera.com/how-does-the-man-in-the-middle-attack-work/>.

5 NITIN, N. Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http. *International Systems Engineering Symposium (ISSE)*, 10 2017.

6 QUINCOZES, S.; EMILIO, T.; KAZIENKO, J. Mqtt protocol: Fundamentals, tools and future directions. *IEEE Latin America Transactions*, p. 1439 – 1448, 09 2019.

7 ANDRADE, F. C. d. *Uma abordagem leve e segura para comunicação utilizando o protocolo MQTT em dispositivos IoT*. 89 p. Monografia (Trabalho Monográfico (Graduação)) — Universidade de São Paulo, 2017.

8 COSTA, B. M. *O PROTOCOLO TLS*. 2010. Disponível em: <https://www.gta.ufrj.br/ensino/eel879/trabalhos_vf_2010_2/bernardo/tls.html>.

9 CAFé, S. *IOT PENTESTING 101: HOW TO HACK MQTT – THE STANDARD FOR IOT MESSAGING*. 2022. Disponível em: <https://securitycafe.ro/2022/04/08/iot-pentesting-101-how-to-hack-mqtt-the-standard-for-iot-messaging>.

10 HIVEMQ. *Creating highly available and ultra-scalable MQTT clusters*. 2016. Disponível em: <https://www.hivemq.com/blog/clustering-mqtt-introduction-benefits/#:~:text=A%20MQTT%20broker%20cluster%20is%20a%20distributed%20system%20that%20represents,like%20a%20single%20MQTT%20broker>.

11 EMQX. *Load balancing - MQTT broker clustering part 1*. 2021. Disponível em: <https://www.emqx.com/en/blog/mqtt-broker-clustering-part-1-load-balancing>.

12 HTTPS://EMBARCADOS.COM.BR/. *MQTT – Protocolos para IoT*. 2016. Disponível em: <https://embarcados.com.br/mqtt-protocolos-para-iot/>.

13 NERI, R.; MATHEUS, L.; BULHÕES, G. *MQTT*. 2019. Disponível em: <https://www.gta.ufrj.br/ensino/eel878/redes1-2019-1/vf/mqtt/>.

14 AUDIT, I. S.; ASSOCIATION, C. (Ed.). *Cybersecurity Fundamentals Study Guide*. USA: Information Systems Audit and Control Association, 2015. ISBN 978-1-60420-594-7.

15 IBM. *O que é uma superfície de ataque?* 2022. Disponível em: <https://www.ibm.com/br-pt/topics/attack-surface>.

16 HOSSAIN, M. M.; FOTOUHI, M.; HASAN, R. Towards an analysis of security issues, challenges, and open problems in the internet of things. *2015 IEEE World Congress on Services*, 06 2015.

17 DONOHUE, B. *Hash: o que são e como funcionam*. 2014. Disponível em: <https://www.kaspersky.com.br/blog/hash-o-que-sao-e-como-funcionam/2773/>.

18 CERTIFICADORA, V. *Criptografia simétrica e assimétrica: qual é a diferença entre elas?* 2018. Disponível em: <https://blog.validcertificadora.com.br/criptografia-simetrica-e-assimetrica-qual-e-diferenca-entre-elas/>.

19 NETWORKS, A. *Network Intrusion*. 2022. Disponível em: <https://aristanetworks.force.com/AristaCommunity/s/article/Network-Intrusion>.

20 JAKHAR, A. *IoT Security - Part 10 (Introduction To MQTT Protocol and Security)*. 2020. Disponível em: <https://payatu.com/blog/aseem/iot-security---part-10-introduction-to-mqtt-protocol-and-security>.

APÊNDICE A – Obtenção de certificados x509

Para se obter os certificados utilizados em clientes com criptografia TLS neste trabalho foi utilizado o OpenSSL que é uma ferramenta de linha de comando de código aberto. Através do terminal do OpenSSL utiliza-se os seguintes comandos descritos a seguir:

Cria-se o par de chaves ca.key que é utilizado para criar o certificado ca.crt, para gerar certificados autoassinados, o campo CN pode ser qualquer:

```
openssl req -new -x509 -days 365 -extensions v3_ca
-keyout ca.key -out ca.crt -passout pass:1234 -subj /CN=TrustedCA.net
```

Cria-se o par de chaves mosquitto.key utilizado para criar o certificado mosquitto.csr que são utilizados pelo *broker*, nesse caso o CN corresponde ao endereço correto, aqui, o IP do *broker*:

```
openssl genrsa -out mosquitto.key 2048
```

```
openssl req -out mosquitto.csr -key mosquitto.key
-new -subj /CN=Mosquitto_broker_adress
```

Utiliza-se a chave ca.key para verificar e assinar o certificado do *broker*, o mosquitto.crt.

```
openssl x509 -req -in mosquitto.csr -CA ca.crt
-CAkey ca.key -CAcreateserial -out mosquitto.crt -days 365 -passin pass:1234
```

APÊNDICE B – Biblioteca Paho Python

Biblioteca que implementa a classe cliente, responsável por fornecer todas as funções para publicar mensagens e assinar tópicos.

Para usar a classe cliente, é preciso importá-la com a seguinte linha:

Import paho.mqtt.client as mqtt

Para criar uma instância é necessário definir o *clientID*, utiliza-se construtor Client(), onde somente o *clientID* é parâmetro obrigatório.

Client(client_id=””, clean_session=True, *userdata=None*, *protocol*=MQTTv311, *transport*=”tcp”)

Parâmetros:

client_id - a *string* de ID de cliente exclusiva usada ao conectar-se ao *broker*. Se client_id tiver comprimento zero ou *None*, um será gerado aleatoriamente. Nesse caso, o parâmetro clean_session deve ser *True*.

clean_session - Se *True*, o *broker* removerá todas as informações sobre este cliente quando ele se desconectar. Se *False*, o cliente é um cliente durável e as informações de assinatura e as mensagens enfileiradas serão retidas quando o cliente se desconectar.

userdata - dados definidos pelo usuário de qualquer tipo que são passados como o parâmetro *userdata* para *callback*. Ele pode ser atualizado posteriormente com a função user_data_set().

protocol - a versão do protocolo MQTT a ser usada para este cliente. Defina como "*websockets*"para enviar MQTT por *WebSockets*. Deixe o padrão de "tcp"para usar o TCP bruto.

B.1 Métodos

B.1.1 *Connect*

Para conectar-se ao *broker* utiliza-se o método *connect*, somente o *host* é parâmetro obrigatório:

connect(*host*, *port*=1883, *keepalive*=60, bind_address=)

Parâmetros:

host - o nome do *host* ou endereço IP do *broker* remoto.

Port - a porta de rede do *host* do servidor para se conectar.

keepalive - período máximo em segundos permitido entre as comunicações com o *broker*. Se nenhuma outra mensagem estiver sendo trocada, isso controla a taxa na qual o cliente enviará mensagens de *ping* para o *broker*.

bind_address - o endereço IP de uma interface de rede local para vincular esse cliente, supondo que existem várias interfaces.

B.1.2 *Publish*

Para publicar mensagens em tópicos utiliza-se o método *publish*:

publish(*topic*, *payload=None*, qos=0, *retain=False*)

Parâmetros:

Topic - o tópico em que a mensagem deve ser publicada.

payload - a mensagem a ser enviada. Se não for fornecido ou definido como Nenhum, uma mensagem de comprimento zero será usada.

qos - a qualidade do nível de serviço a ser usado.

retain - se definido como *True*, a mensagem será definida como a "última mensagem válida"/retida para o tópico.

B.1.3 *Subscribe*

Para se inscrever em algum tópico utiliza-se o método *subscribe*:

subscribe(topic, qos=0)

Parâmetros:

Topic - o tópico em que o cliente deve se inscrever.

qos - a qualidade do nível de serviço a ser usado.

B.2 *Callbacks*

Callbacks são funções que são chamadas em resposta a um evento, dependem do loop do cliente que é chamado com o método loop_start(), sem ele os *callbacks* não são acionados. Os *callbacks* mais utilizados são:

B.2.1 on_connect

Chamado quando o *broker* responde à solicitação de conexão:

on_connect(*client*, *userdata*, *flags*, rc)

Parâmetros:

client - a instância do cliente para este *callback*.

userdata - os dados do usuário privado conforme definido em Client() ou user_data_set().

flags - sinalizadores de resposta enviados pelo *broker*.

rc - indica o resultado da conexão.

0: Conexão bem-sucedida.

1: Conexão recusada - versão de protocolo incorreta.

2: Conexão recusada - identificador de cliente inválido.

3: Conexão recusada - servidor indisponível.

4: Conexão recusada - nome de usuário ou senha incorretos.

5: Conexão recusada - não autorizado.

6-255: Atualmente não utilizado.

B.2.2 on_disconnect

Chamado quando o cliente é desconectado do *broker*:

on_disconnect(*client*, *userdata*, rc)

Parâmetros:

client - a instância do cliente para este *callback*.

userdata - os dados do usuário privado conforme definido em Client() ou user_data_set().

rc - indica o estado de desconexão. Se MQTT_ERR_SUCCESS (0), o *callback* foi chamado em resposta a uma chamada de disconnect(). Para qualquer outro valor, a desconexão foi inesperada.

B.2.3 on_log

Chamado quando o cliente tem informações de *log*. É necessário defini-lo para permitir a depuração:

on_log(*client*, *userdata*, *level*, *buf*)

Parâmetros:

client - a instância do cliente para este *callback*.

userdata - os dados do usuário privado conforme definido em Client() ou user_data_set().

level - fornece a severidade da mensagem.

buf - Onde está a mensagem em si.

B.2.4 on_message

Chamado quando uma mensagem foi recebida em um tópico que o cliente assina e a mensagem não corresponde a um *callback* de filtro de tópico existente:

on_message(*client*, *userdata*, *message*)

Parâmetros:

client - a instância do cliente para este *callback*.

userdata - os dados do usuário privado conforme definido em Client() ou user_data_set()

message - uma instância de MQTTMessage. Esta é uma classe com tópico de membros, carga útil, qos, retenção.

ANEXO A – Cliente sem nenhuma autenticação

A.1 Código Python

```
import paho.mqtt.client as mqtt  #importa o client1

def on_log(client, userdata, level, buf):
        print("log: "+buf)
def on_connect(client, userdata, flags, rc):
    if rc==0:
        print("connected OK")
    else:
        print("Bad connection Returned code=",rc)
def on_disconnect(client, userdata, flags, rc=0):
        print("Disconnected result code "+str(rc))
        client.loop_stop()

broker="localhost"
client = mqtt.Client("python1")#cria nova instancia

client.on_connect=on_connect  #bind da função de callback
client.on_disconnect=on_disconnect
client.on_log=on_log

print("Connecting to broker ",broker)

client.connect(broker,1883)      #conecta no broker na porta 1883
client.subscribe("Topico_teste_Luiggi")

i=0;
client.loop_start()
while True:
        if i<1:
                client.publish("Topico_teste_Luiggi","teste_tamanho")
                i =i+1;

        else:

                client.disconnect() # disconnect
```

ANEXO B – Cliente com autenticação de usuário e senha

B.1 Código Python

```
import paho.mqtt.client as mqtt  #importa o client1
import time
def on_log(client, userdata, level, buf):
        print("log: "+buf)
        client.tls_set()
def on_connect(client, userdata, flags, rc):
    if rc==0:
        print("connected OK")
    else:
        print("Bad connection Returned code=",rc)
def on_disconnect(client, userdata, flags, rc=0):
        print("DisConnected result code "+str(rc))
        client.loop_stop()    #Stop loop

def on_message(client,userdata,msg):
        topic=msg.topic
        m_decode=str(msg.payload.decode("utf-8","ignore"))
        print("message received",m_decode)
broker = "localhost"
client = mqtt.Client("python1")#cria nova instancia
user="teste"
password="teste"

client.username_pw_set(user,password=password)
#seta as funções de callback
client.on_connect=on_connect
client.on_disconnect=on_disconnect
client.on_log=on_log
client.on_message=on_message
print("Connecting to broker ",broker)

client.connect(broker,1884)      #conecta ao broker
client.subscribe("Topico_teste_Luiggi")
i=0;
client.loop_start()  #inicia o loop
```

```
while True:
        if i<1:

                client.publish("Topico_teste_Luiggi","teste_tamanho")
                i =i+1;
        else:
                client.disconnect()
```

ANEXO C – Cliente com criptografia de mensagem

C.1 Código Python

```
 import time
import paho.mqtt.client as paho
from cryptography.fernet import Fernet

broker="localhost"

#define callback
def on_log(client, userdata, level, buf):
    print("log: ",buf)
def on_message(client, userdata, message):
   print("receive payload ",message.payload)
   if message.payload==encrypted_message:
      print("\npublished and received messages are the same")
   decrypted_message = cipher.decrypt(message.payload)
   print("\nreceived message =",str(decrypted_message.decode("utf-8")))

client= paho.Client("python1")  #cria o cliente

client.on_message=on_message
#####encryption
#cipher_key = Fernet.generate_key()
#cipher = Fernet(cipher_key)
cipher_key=b'WDrevvK8ZrPn8gmiNFjcOp2xovBr40TCwJlZOyI94IY='
cipher = Fernet(cipher_key)
message = b'teste_tamanho'

encrypted_message = cipher.encrypt(message)
out_message=encrypted_message.decode()# transforma numa string para envio

print("connecting to broker ",broker)
client.connect(broker)#conecta
i = 0;
client.loop_start() #inicia o loop para processar mensagens recebidas
print("subscribing ")
client.subscribe("Topico_teste_Luiggi")#subscribe
```

```
time.sleep(2)
print("publicando  mensagem encriptada  ",encrypted_message)

while True:
    if i<1:
        client.publish("Topico_teste_Luiggi",out_message)#publish
        i = i+1;

    else:
        client.disconnect()
```

ANEXO D – Cliente com autenticação de usuário e senha e criptografia TLS

D.1 Código Python

```
import paho.mqtt.client as paho
import time

broker = "DESKTOP-5P4804T"
port=8885
user="teste"
password="teste"
conn_flag= False
def on_connect(client, userdata, flags, rc):
    global conn_flag
    conn_flag=True
    if rc==0:
        print("usuario e senha OK")
    else:
        print("Bad connection Returned code=",rc)
    print("connected",conn_flag)
    conn_flag=True
def on_log(client, userdata, level, buf):
    print("buffer", buf)
def on_disconnect(client, userdata, rc):
    print("client disconnected ok")

client1= paho.Client("python1")
client1.username_pw_set(user,password=password)
client1.on_log=on_log
#Configura o cliente para suporte SSL/TLS com certificados.
client1.tls_set('C:\etc\mosquitto\localhostcert\ca.crt',
'C:\etc\mosquitto\localhostcert\server.crt',
'C:\etc\mosquitto\localhostcert\server.key')
client1.on_connect = on_connect
client1.on_disconnect = on_disconnect
client1.connect(broker,port)
client1.subscribe("Topico_teste_Luiggi")
while not conn_flag:
```

```
    time.sleep(1)
    print("waiting", conn_flag)
    client1.loop()
time.sleep(3)

i = 0
while True:
        if i<1:

                client1.publish("Topico_teste_Luiggi","teste_tamanho", 0, False)
                i =i+1;

        else:
            client1.disconnect() # desconecta
```

ANEXO E – Cliente ESP32

E.1 Código ESP32

```
#include <ssl_client.h>
#include <WiFiClientSecure.h>
#include <WiFi.h>
#include <PubSubClient.h>

//ca.crt
const char* CA_cert = \
"-----BEGIN CERTIFICATE-----\n" \
"MIIDCTCCAfGgAwIBAgIURffAoDCujUsPvecpZJfsm1mxHuIwDQYJKoZIhvcNAQEL\n" \
"BQAwFDESMBAGA1UEAwwJVHJ1c3RlZENBMB4XDTIyMTAxNjE1NTI1MloXDTIzMTAx\n" \
"NjE1NTI1MlowFDESMBAGA1UEAwwJVHJ1c3RlZENBMIIBIjANBgkqhkiG9w0BAQEF\n" \
"AAOCAQ8AMIIBCgKCAQEAzxfTRFRsJoUN3TVMTz2UPpEPcGzzsJyVXhTVa66QreoW\n" \
"+T1Ig4g0BasD6DceiOvApVlSiharz787lcVCEM1L7mpkoB9YWJLavdn8v8ij7a3u\n" \
"f3uXrvIGPZMJyTFdFV2LUsyFTjNLPFbcQkwbzTIPKsjkjDB6PVx1WI1QATFJdjpB\n" \
"GNXRKWwKSKR8dNHfG9tlAvxo/BUXS+eFKsRe2YPfRtYMLu+Fr4AbBtjK7y5tC4Hr\n" \
"sOI6tqCe692hPIK8VOabWdfXZjbvyx/UEdeP7B+1VdLaNgCzM2QM7KFw5X4mzjxv\n" \
"twVMwCVegek2ZIdNcnKTjEF869VmXYZkXnXiCTwnFwIDAQABo1MwUTAdBgNVHQ4E\n" \
"FgQUGMaNG1Zhet3uFyKeqqBqf/Az07kwHwYDVR0jBBgwFoAUGMaNG1Zhet3uFyKe\n" \
"qqBqf/Az07kwDwYDVR0TAQH/BAUwAwEB/zANBgkqhkiG9w0BAQsFAAOCAQEAK18H\n" \
"QXO/Th3mnWoCWvyX4EnX5GZqtlZium1ZLAVuPj9GUz4RSLq0IoBfsRL9pSVZr8Df\n" \
"4zqb5GNK5YYsglp0jE8pDd9f5+X5FYlcw6BteOnGy+w7S23jxaX0FEkT1qjktTgm\n" \
"+rB8xZSUYtVjUZ78j21i3KLxmoxa97QWAJl0lJCHD2UkMWTg4+wbvTXBrF95HN5C\n" \
"c7h9ypAOBSpxInpL2Yjb8IGk8Svsv429fPk/3pOEEfWQd94nQxJqWCSHc3LhQfnC\n" \
"VOwf3cJceN7ezkoIeU1iuLYX5lH4venJZRDRiaqmr3XOlrHXNJyHaZmGV/LdW6i7\n" \
"3mRpJsXyIlsOVOJTWw==\n" \
"-----END CERTIFICATE-----";
//mosquitto.crt
const char* ESP_CA_cert = \
"-----BEGIN CERTIFICATE-----\n" \
"MIICsTCCAZkCFArWmdkp+hN4qD5p7wdN52Et1zaUMA0GCSqGSIb3DQEBCwUAMBQx\n" \
"EjAQBgNVBAMMCVRydXN0ZWRDQTAeFw0yMjEwMTYxNTU0NTdaFw0yMzEwMTYxNTU0\n" \
"NTdaMBYxFDASBgNVBAMMCzE5Mi4xNjguMC4yMIIBIjANBgkqhkiG9w0BAQEFAAOC\n" \
"AQ8AMIIBCgKCAQEAtuuVSLFk/KK9KOOQFWJSpzButnJC0prC+LxWmuunCqU9KIpG\n" \
"WyZlfSt/BgIU9XXYxM+Mwsy2YXdNfmXg9wZGhWSCg+hl5NX5joRcZstlIp+tuW+m\n" \
"EtCeXjo4JpBkE8sP/6ZZRD2yixtei1fKo9jsTSwszYlvITso+IwO8OiP8r7Uiy5z\n" \
```

```
"ksTvK0qREGSATkoW+8fD7+Lg9IRCNyqRM8icfARGpdXhMecpag9RLLOuGeVxpzx5\n" \
"LIiIQX5amO/rPDPPWrEE4IBahovkb+nqNXoxBLtwgioC1CRJLFWQLwF7BKPPNR33\n" \
"deMEk2FkmXv1omxQijNzRmh5NQq0RrcolJfg9QIDAQABMA0GCSqGSIb3DQEBCwUA\n" \
"A4IBAQClFWcSwnJMdkmJN0Jsnujwy4tAPAXeWTqsU6hxpy0tmfcfEIDAREwdcMh3\n" \
"tDlwemnQwaRIWjefKEer2+82A3l558ncuRkGVKjO+bEv4alodP/PbMJfjV8XXWIr\n" \
"t4e6XPwoZyqL6hePy4dKP0d5dp0D86nnlndfAGyNeYIsyoMwy1n0u2k21h7Je7rj\n" \
"B2UgsNwvF4KynO+k3EhHLlNSIIJ9tNXtDeI5+L/cr13PLUIeGofQYXRvKZBwo6zD\n" \
"E4gN+k/G9GDKhAvhKnqHA/Eu7lktQtpcU8EZxUMMu/ScphJ7chyRDpStOFjymLxc\n" \
"l4swTW+Gp8yn4DphUD4Oyex0+oBX\n" \
"-----END CERTIFICATE-----";
//mosquitto.key
const char* ESP_RSA_key= \
"-----BEGIN PRIVATE KEY-----\n" \
"MIIEvQIBADANBgkqhkiG9w0BAQEFAASCBKcwggSjAgEAAoIBAQC265VIsWT8or0o\n" \
"45AVYlKnMG62ckLSmsL4vFaa66cKpT0oikZbJmV9K38GAhT1ddjEz4zCzLZhd01+\n" \
"ZeD3BkaFZIKD6GXk1fmOhFxmy2Uin625b6YS0J5eOjgmkGQTyw//pllEPbKLG16L\n" \
"V8qj2OxNLCzNiW8hOyj4jA7w6I/yvtSLLnOSxO8rSpEQZIBOShb7x8Pv4uD0hEI3\n" \
"KpEzyJx8BEal1eEx5ylqD1Ess64Z5XGnPHksiIhBflqY7+s8M89asQTggFqGi+Rv\n" \
"6eo1ejEEu3CCKgLUJEksVZAvAXsEo881Hfd14wSTYWSZe/WibFCKM3NGaHk1CrRG\n" \
"tyiUl+D1AgMBAAECggEAMS6Ue/AefALxo03UTruaB5PxIKMGLoQoCiLxkkPsLOOS\n" \
"1xSeqCFhxk7sn1vt8LWX7Ar2COZr3zbpoiOYYjLQwUruALN6uhWCPPx/r6/eaHnQ\n" \
"hvxrBcIL4r/6mvvVvLCrg4xhkgSYvehE+mhHdth7aKCJB1Z9h8zVTkayObn+AkEx\n" \
"rcVIK8zHkpUeM93Y7aKsyWP/WiFMxSKYBzvWj43U37WmRnFungGPNvZUNpTshLYu\n" \
"lPVPTsDwwQubRrlB4Ge9KpYPhqIq3ZMy5/IUGjNDaDcerHDHctRl+J2h/CHlRyKx\n" \
"bTWdOxFc8jJO0LP1cmR3uu2gg+8QtTUUD6v553GGfQKBgQC+rXwaEz3js8VChRDO\n" \
"pzXFdXfrGTSX+XhgI8SFZoSH4OLbrlNdLqFdzqqFA+i9hob/EkVAKXCqxZaqKAta\n" \
"gF3piN1CSWhUINKh4BWYtAcyovRxnhFVIYdD262BaFGK4dzIa4MDdt+Ku/FS0vkC\n" \
"Aygw+k5A1pck3uPlOP1rA5dJMwKBgQD1lcS+OG14SrECsXKvd2diVS5R4pHiUZIT\n" \
"tNKPQtYYrjZ/lLVVi6VBo/FiMUopOeFMUBMvexAV0wKpy/2s/u6WVOHOo3J7/eAw\n" \
"OanO87FphRZocbW+Xf1CByKqlD9TGdfTz0y6MYEzvDkQ2HFIhPT46VqxqnlLUzwL\n" \
"hfWwjsI9NwKBgE2NQ99bGh3lfQJsGoTibzVLbKWbD9AL8BCyG3jiVF7rcXlF8rQA\n" \
"hjMgHiyRhXSoJXnS1YWeFSvvxzKXrN53PTsBpnQSZTNqUiDygfYkqpTGwEMBtiz4\n" \
"wQoxa+UpJ5kj+ecuCxA6pfrILAOyuQI/hHY/J9qpLLobSXducWu3/y0PAoGBAIpR\n" \
"wG4i+gU2jFwn9aA+cExCDcSxB9mgd9YBklZ+HnfoJ2nUKA32iS1fWSUk/dAaeXqx\n" \
"3xxvA7BHGMjpzAcTzu19AXGRPXn4k7yVvb6CIvp7qxNIeESwS2RVAIFy3ffAJn91\n" \
"0pFMMKki6kFCpwthV1ialipbCdYTG06iNK8suNohAoGAUxLpSTprDb0bEFpBnPwk\n" \
"h4hzP4TPbpAXz4fJEiFa0O2RlBP949UDy9xx3NxytZhkG/VSlu+QYlxLd8SVEarD\n" \
"I/RK4WnKyo7FHCoIlP6LDEi1zE+jOXwg3UHObKNmd9mlKw7wWbkJcZI3LYHduhqE\n" \
"+BtUqzMfbeDBUIN1f1ecyEE=\n" \
```

```
"-----END PRIVATE KEY-----" ;


const char* ssid     = "NET_VIRTUA_102";     //  network SSID (nome do wifi)
const char* password = "abcd1234"; // senha da internet

const char* server = "192.168.0.2";
const char* mqtt_user = "teste";
const char* mqtt_pass = "teste";

void callback(char* topic, byte* payload, unsigned int length) {

  Serial.print("Message arrived in topic: ");
  Serial.println(topic);

  Serial.print("Message:");
  for (int i = 0; i < length; i++) {
    Serial.print((char)payload[i]);
  }
  Serial.println();
  Serial.println("-----------------------");
}

WiFiClientSecure client;
PubSubClient mqtt_client(client);

void setup() {
  Serial.begin(115200);
  delay(100);

  Serial.print("Attempting to connect to SSID: ");
  Serial.println(ssid);
  WiFi.begin(ssid, password);

  // tentativa de conectar no Wifi :
  while (WiFi.status() != WL_CONNECTED) {
    Serial.print(".");
    // espera 1s para nova tentativa
    delay(1000);
```

```
  }

  Serial.print("Connected to ");
  Serial.println(ssid);

  client.setCACert(CA_cert);
  //para verificação do cliente
  //caso require_certificate = true no .conf do broker
  client.setCertificate(ESP_CA_cert);
  client.setPrivateKey(ESP_RSA_key);
  mqtt_client.setServer(server, 8885);

}

void loop() {

  int var = 0;
  Serial.println("\nStarting connection to server...");
  //if you use password for Mosquitto broker
  if (mqtt_client.connect("ESP32", mqtt_user , mqtt_pass)) {
  //if you dont use password for Mosquitto broker
  //if (mqtt_client.connect("ESP32")) {
    Serial.print("Connected, mqtt_client state: ");
    Serial.println(mqtt_client.state());// 0 conectado com sucesso
    mqtt_client.subscribe("Topico_teste_Luiggi");

}
    else {
    Serial.println("Connected failed!  mqtt_client state:");
    Serial.print(mqtt_client.state());
    Serial.println("WiFiClientSecure client state:");
    char lastError[100];
    client.lastError(lastError,100);  //pega o ultimo erro para WiFiClientSecure
    Serial.print(lastError);
  }
  delay(10000);
}
```