UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO DE JOINVILLE
ENGENHARIA MECATRÔNICA

VINICIUS SAKAGUCHI DE NOBREGA

GUIA DE CODIFICAÇÃO SEGURA DE DISPOSITIVOS IOT

Joinville
2024

VINICIUS SAKAGUCHI DE NOBREGA

GUIA DE CODIFICAÇÃO SEGURA DE DISPOSITIVOS IOT

Trabalho apresentado como requisito parcial para obtenção do título de bacharel em Engenharia Mecatrônica, no Curso de Engenharia Mecatrônica, do Centro Tecnológico de Joinville, da Universidade Federal de Santa Catarina.

Orientador: Dr. Ricardo José Pfitscher

Joinville
2024

AGRADECIMENTOS

Gostaria de agradecer ao professor Dr. Ricardo José Pfitscher por ter aceitado ser o orientador deste trabalho. Agradeço também à minha família e aos meus amigos pelo apoio constante durante a execução deste projeto.

RESUMO

A Internet das Coisas (*Internet of Things* - IoT) tem revolucionado a interação com o mundo físico, conectando diversos dispositivos à internet. No entanto, essa proliferação de dispositivos IoT trouxe desafios significativos em termos de segurança, muitas vezes devido à falta de atenção às práticas seguras de programação. Dessa forma, é possível que esses dispositivos sejam invadidos, de modo que seria viável para o invasor parar os processos em execução ou até mesmo alterar os dados presentes, problemas que podem ser reduzidos ao se aplicar boas práticas durante a programação dos dispositivos IoT. Considerando isso, apresenta-se neste trabalho um guia prático para codificação segura de dispositivos IoT. Para tanto, foi realizada uma pesquisa sobre as principais vulnerabilidades associadas à codificação destes dispositivos e foi estabelecido um conjunto de verificações que desenvolvedores devem aplicar para mitigar riscos de segurança causados por má programação. Além disso, o guia desenvolvido foi aplicado na análise de trabalhos de conclusão de curso que envolvem o desenvolvimento de software IoT para identificar possíveis vulnerabilidades, assim como comparar os resultados obtidos com o diagnóstico de segurança providenciados por ferramentas de Inteligência Artificial (IA).

Palavra-chave: Internet das Coisas; segurança de software; programação segura.

ABSTRACT

The Internet of Things (IoT) has revolutionized interaction with the physical world by connecting various devices to the internet. However, this proliferation of IoT devices has brought significant security challenges, often due to a lack of attention to secure programming practices. As a result, these devices can be compromised, allowing attackers to halt ongoing processes or even alter existing data. These issues can be mitigated by applying good practices during the programming of IoT devices. In light of this, this work presents a practical guide for secure coding of IoT devices. To achieve this, research was conducted on the main vulnerabilities associated with coding these devices, and a set of checks was established that developers should apply to mitigate security risks caused by poor programming. Additionally, the developed guide was applied to the analysis of final year projects involving IoT software development to identify potential vulnerabilities, as well as to compare the results obtained with the security diagnostics provided by Artificial Intelligence (AI) tools.

Keywords: Internet of Things; software security; secure programming.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

IoT	Internet of Things
IA	Inteligência Artificial
TCC	Trabalho de Conclusão de Curso
SI	Segurança da Informação
IP	Internet Protocol
MQTT	Message Queuing Telemetry Transport
FPGA	Field-programmable gate array
SO	Sistemas Operacionais
DoS	Denial of Service
DDoS	Distributed Denial of Service
MITM	Man-In-The-Middle
ENISA	European Union Agency for Cybersecurity
OWASP	Open Web Application Security Project
IV	Initialization Vector
AES	Advanced Encryption Standard
HTTP	Hypertext Transfer Protocol
URL	Uniform Resource Locator

SUMÁRIO

1 INTRODUÇÃO

A Internet das Coisas (*Internet of Things* - IoT) é um paradigma que está em rápida expansão quando se trata de realizar telecomunicações sem fio. A IoT refere-se à ideia de que objetos ao nosso redor podem se comunicar entre si em prol de um objetivo comum por meio de esquemas únicos de endereçamento (Atzori *et al.*, 2010). De acordo com isso, era previsto que entre 2018 e 2023 houvesse um aumento 10,9 bilhões de dispositivos conectados à rede, logo enfatizando o crescimento do uso de dispositivos IoT.

No entanto, por mais importante que a IoT tenha se tornado, ainda existem desafios de segurança de software. Os dispositivos que utilizam essa tecnologia não são normalmente programados tendo em mente o aspecto de segurança, resultando em vulnerabilidades críticas, já que os dispositivos IoT ainda não estão completamente desenvolvidos nem protegidos. Nesse contexto, é importante observar que a má programação do dispositivo pode vir a criar um ponto de entrada para ataques, fazendo com que estes tenham acesso a seus dados e possam alterá-los (Oracevic *et al.*, 2017).

Para ajudar na segurança de dispositivos IoT, já foram feitos outros guias e pesquisas, como por exemplo os trabalhos feitos por Berlanda (2021), Alhirabi *et al.* (2021) e Rocha Junior (2022), nos quais são mostrados, de uma maneira geral, vulnerabilidades nas áreas de hardware, software e peopleware e, com base nisso, fornecem diretrizes e boas práticas que podem mitigar os riscos apresentados.

Dessa forma, mesmo que já existam trabalhos focados em auxiliar na segurança de dispositivos IoT, esses não demonstram, para os problemas causados por má codificação, dicas e métodos práticos para a programação segura. Logo, objetiva-se desenvolver um guia voltado a orientar a codificação desses dispositivos de modo há evitar ou mitigar essas vulnerabilidades de software.

A metodologia da pesquisa apresentada neste trabalho investiga as vulnerabilidades potenciais resultantes de práticas inadequadas de programação em dispositivos IoT, apontando soluções para os desafios identificados, em um guia sobre programação segura.

Ao mesmo tempo, será conduzido um levantamento de outros Trabalhos de Conclusão de Curso (TCC) que envolvem dispositivos IoT e disponibilizam o código fonte feitos por alunos do curso de Engenharia Mecatrônica e Engenharia de Controle e Automação, da Universidade Federal de Santa Catarina (UFSC) e do curso de Engenharia Elétrica da Universidade do Estado de Santa Catarina (UDESC) a fim de aplicar o guia produzido e verificar a segurança de software nos conteúdos dos trabalhos identificados. Por fim, será feita uma comparação entre os resultados en-

contrados por meio do guia e os diagnósticos de segurança obtidos por meio das ferramentas *ChatGPT* e *Gemini*.

1.1 OBJETIVOS

1.1.1 Objetivo geral

Desenvolver um guia de codificação segura de dispositivos IoT.

1.1.2 Objetivos Específicos

- Discernir os possíveis riscos de segurança existentes ao codificar dispositivos IoT;
- Identificar formas para mitigar essas falhas de segurança;
- Aplicar o guia esquematizado em outros trabalhos de conclusão de curso que envolvem dispositivos IoT e disponibilizam código fonte de modo a verificar se possuem vulnerabilidades;
- Comparar os resultados encontrados com diagnósticos de segurança realizados por ferramentas de IA.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo será apresentada a fundamentação teórica do trabalho com o objetivo de ajudar no entendimento da pesquisa, uma vez que é necessária uma compreensão acerca do que é IoT e um pouco sobre a estrutura de seus dispositivos, tendo um foco em sua parte de software, assim como apresentar uma ideia geral sobre segurança de informação e de vulnerabilidades existentes em IoT que são causadas por má programação.

2.1 INTERNET DAS COISAS

Até o momento, não há uma definição amplamente consensual sobre a IoT, consequentemente, muitos autores nesse domínio adotam suas próprias interpretações, resultando em variações de conceitos, algumas mais substanciais do que outras. Uma das possíveis definições é de que a IoT é uma rede de objetos que pode detectar e trocar informações entre si, podendo esses objetos serem físicos ou digitais e identificados por meio de endereçamentos únicos baseados em protocolos de comunicação (Weber; Boban, 2016).

Assim, por meio da IoT, é possível que diversos objetos ao redor do mundo sejam identificados, conectados e tomem suas próprias decisões, tornando viável que fontes de informação diferentes, tais como sensores, dispositivos móveis e até carros, troquem dados, aumentando a quantidade de diferentes aplicações exponencialmente. Com isso em mente, é possível citar que a IoT já está sendo utilizado em áreas como a indústria, em *smart homes*, ou seja, casas automatizadas, na produção agrícola, e na área de saúde (Dudhe *et al.*, 2017).

2.2 PILARES DA SEGURANÇA DA INFORMAÇÃO

A Segurança da Informação (SI) envolve a proteção de informações com o objetivo de defender o valor que essas possuem para uma pessoa, grupo ou empresa, sendo os critérios da SI a confidencialidade, integridade, disponibilidade e autenticidade (Fukuda, 2019). Dessa forma, a quebra de qualquer um dos critérios apresentados caracteriza um problema de SI, de modo que esses critérios podem ser definidos da seguinte maneira (Campos, 2007):

- Confidencialidade: seria a limitação do acesso a uma informação para apenas os autorizados pelo proprietário dessa;
- Integridade: envolve a não alteração indevida da informação original, ou seja, quando ocorre uma modificação inadequada, seja intencional ou não, como a

falsificação de um documento ou a alteração de registros em um banco de dados, caracteriza-se à quebra de integridade da informação;
- Disponibilidade: quando uma informação está acessível para os indivíduos que tem direito de acessá-la, de modo que, acontece um incidente por quebra de disponibilidade quando nem mesmo pessoas autorizadas conseguem acessar uma informação, seja isso causado por um ataque externo ou por perda de documentos;
- Autenticidade: se baseia na veracidade da fonte da informação, isto é, se é possível verificar a identidade da entidade que disponibilizou a informação.

2.2.1 Estrutura dos dispositivos IoT

Para o funcionamento dos dispositivos IoT, é necessária a integração de diferentes tecnologias e objetos, sendo esses o que compõe a estrutura desses dispositivos. Assim, esses podem ser definidos pelo seguintes blocos (Fukuda, 2019):

- Identificação: é essencial para diferenciar os objetos dentro da rede e descobrir por meio de onde serão conectados à internet, um exemplo de uma tecnologia assim seria o endereço de protocolo da internet ou Internet Protocol (IP) address;
- Sensores: são os resposáveis por adquirir, armazenar e/ou enviar dados para *clouds* ou bases de dados, de modo que é normalmente utilizado o Wi-Fi ou Bluetooth;
- Comunicação: se refere as técnicas e tecnologias utilizadas para conectar e realizar a troca de informação entre dispositivos, sendo as mais utilizadas o Wi-Fi e o Bluetooth, podendo utilizar o protocolo Message Queuing Telemetry Transport (MQTT);
- Computação: envolve a utilização de uma unidade de processamento, ou seja, um microcontrolador, processador ou field-programmable gate array (FPGA), que tem a responsabilidade de executar os algoritmos locais nos dispositivos.

2.2.1.1 Software

Nos dispositivos IoT, devido as suas características mais restritas se comparadas a outros tipos de dispositivos, os softwares buscam ser mais compactos e econômicos. Dentre os softwares que compõe esses dispositivos, existem os Sistemas Operacionais (SO) que são vitais para o funcionamento do equipamento, e por causa das restrições mencionadas, são necessários SOs específicos para o uso em IoT. A escolha do SO a ser utilizado influencia no funcionamento de um projeto e deve levar em consideração as seguintes especificações (Cekerevac *et al.*, 2020):

- Consumo de memória: é preferível que o SO possua o menor consumo de memória possível;

- Funcionamento em tempo real: é mais favorável que o SO possa operar em tempo real uma vez que existem aplicações que dependem disso;
- Eficiência energética: quando o SO não possui um grande consumo de energia, economiza-se dinheiro e tempo na manutenção da bateria;
- Flexibilidade: se o SO for flexível, pode ser usado em diferentes tipos de dispositivos, logo os desenvolvedores não teriam a necessidade de conhecer diversos SOs;
- Segurança: o SO preferencialmente deve possuir medidas de segurança;
- Confiabilidade: como dispositivos IoT normalmente funcionam de maneira remota, é importante que um SO funcione por anos sem a ocorrência de problemas.

2.2.2 Segurança de dispositivos IoT

A segurança em IoT envolve a proteção de informações de negócios e dados pessoais que são armazenadas e transmitidas por meio de dispositivos conectados a internet, logo, abrange a proteção contra o acesso e possível alteração não autorizada desses dados. Dito isso, mesmo que seja de grande importância, a segurança de dispositivos IoT, na maioria dos casos, não é levada em consideração durante o projeto de um novo produto, o que resulta em eventuais roubos ou vazamentos de informações (Ahlmeyer; Chircu, 2016).

Dessa forma, todo dispositivo IoT, sem as devidas medidas de segurança, que é conectado a uma rede, seja em casa ou em uma empresa, pode vir a se tornar um ponto de entrada para um ataque que se aproveite da vulnerabilidade no dispositivo, levando a uma invasão ou até ataque a essa rede.

2.3 PRINCIPAIS VULNERABILIDADES E TIPOS DE ATAQUE DE SOFTWARE EM IOT

Uma vez que a maioria dos dispositivos IoT tem uma proteção fraca causada por seus recursos limitados e funções de segurança insuficientes, esses acabam se tornando alvos de *hackers*, resultando em diversos incidentes de brechas de segurança (Lim *et al.*, 2020). Com isso em mente, é importante notar que falhas na programação do software desses dispositivos podem levar a diversas vulnerabilidades de segurança, essas que serão apresentadas a seguir.

2.3.1 Vulnerabilidades em códigos de terceiros

Vulnerabilidades em códigos de terceiros são falhas de segurança que foram divulgadas publicamente através de canais de comunicação abertos e podem ser encontradas em bibliotecas de software, frameworks e outras dependências de código

que são frequentemente integradas em sistemas de software. Quando um componente vulnerável é utilizado em um sistema de software, o risco de vazamento de dados ou perda de controle sobre servidores aumenta. Essas vulnerabilidades são facilmente rastreadas por atacantes e permitem explorações automatizadas em grande escala (Cadariu *et al.*, 2015).

2.3.2 Denial of Service

Os ataques de negação de serviço (Denial of Service - DoS) ocorrem ao tornar um servidor inoperante sobrecarregando-o com diversas solicitações de serviço. Dessa forma, é realizada uma sobrecarga de pacotes ao enviar uma quantidade de dados maior que a largura de banda que uma rede ou host possa aguentar tornando a rede instável, de modo que seria possível esgotar os recursos de um dispositivo IoT, comprometendo seu funcionamento. Uma forma diferente desse mesmo ataque seria o Distributed Denial of Service (DDoS), que se baseia em ataques semelhantes ao DoS, porém utilizando uma quantidade maior de dispositivos, de modo que quando executado se torna um ataque DoS em grande escala (Fukuda, 2019).

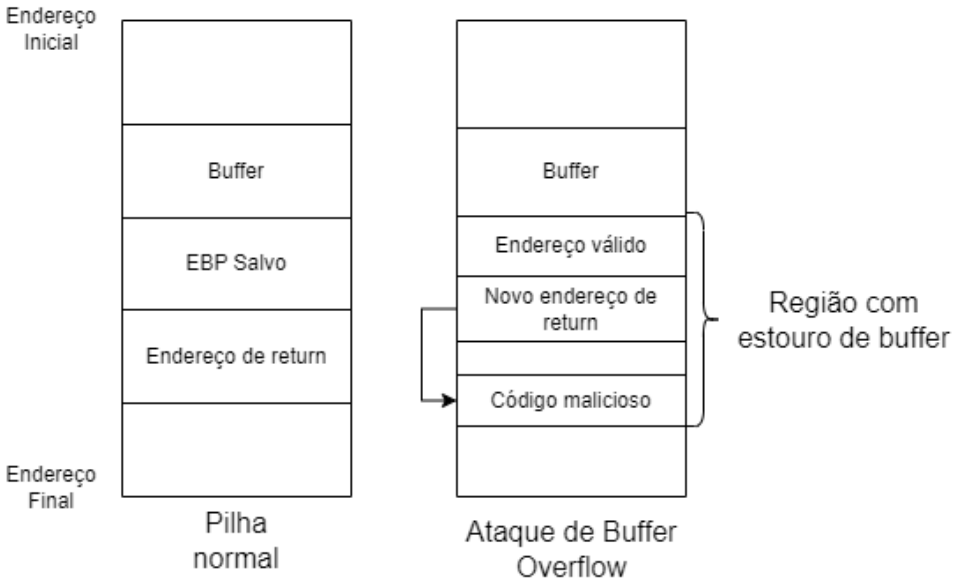
2.3.3 Buffer overflow

A vulnerabilidade de buffer overflow é uma das mais comuns de se encontrar em códigos e também uma bastante perigosa já que pode resultar em um atacante tendo acesso total ao dispositivo. Os ataques que se aproveitam dessa vulnerabilidade ocorrem por meio da subversão de uma função de um programa, de modo que o atacante pode tomar o controle do programa e, dependendo dos privilégios desse, controlar o host (Cowan *et al.*, 2003).

O Buffer overflow, ou estouro de buffer, é uma vulnerabilidade de segurança comum que ocorre quando um programa tenta armazenar mais dados em um buffer (uma área de memória temporária) do que ele foi projetado para lidar. Isso pode levar a problemas graves de segurança, como corrupção de memória, travamento do programa ou até mesmo a execução de código malicioso pelo atacante. Assim, na Figura 1 observa-se uma ilustração que mostra o estouro de um buffer possibilitando a execução de um trecho de código malicioso.

Para realizar um ataque de overflow de buffer, é preciso atingir dois objetivos principais: primeiro, garantir que o código malicioso esteja disponível em algum local do espaço de endereçamento do programa; em segundo lugar, fazer com que o programa salte para esse código, com parâmetros adequados carregados nos registradores e na memória. Dessa forma, após cumprir ambos os objetivos, se torna possível executar o programa malicioso no programa que está sendo atacado (Cowan *et al.*, 2003).

Figura 1 – Ilustração de buffer overflow



Fonte: Adaptado de KL (2023)

2.3.4 Problemas de autentificação

As restrições existentes em dispositivos IoT tais como energia e poder computacional limitados, dificultam a implementação de mecanismos complexos de autentificação. Assim, é possível que um atacante se aproveite dessa vulnerabilidade para injetar códigos maliciosos ou violar a integridade da informação, logo, nesses casos, as chaves de autentificação entram em risco de serem perdidas ou alteradas, possibilitando o acesso ao invasor (Neshenko *et al.*, 2019).

2.3.5 Criptografia falha

A criptografia é um mecanismo efetivo para armazenar e transmitir informações de maneira segura, porém, já que o funcionamento de um sistema criptográfico depende do algoritmo empregado, os recursos limitados dos dispositivos IoT complicam sua robustez e eficácia. Dessa forma, se torna viável que um atacante contorne as técnicas de criptografia utilizadas com o intuito de revelar informações privadas ou tomar o controle do dispositivo invadido (Neshenko *et al.*, 2019).

2.3.6 Abertura de portas desnecessárias

É comum que dispositivos IoT operem com portas desnecessárias abertas enquanto executam serviços vulneráveis, ou seja, esses equipamentos possuem portas de comunicação com a rede que não estão sendo usadas porém permitem o envio e recebimento de dados. Essa má prática pode resultar na conexão de um atacante ao dispositivo, de modo a permitir que ocorram uma gama de ataques explorando outras

vulnerabilidades, logo, é recomendável que portas não utilizadas sejam fechadas ou tenham seu acesso restringido (Neshenko *et al.*, 2019).

2.3.7 Vulnerabilidades de dia zero

Vulnerabilidades de dia zero são falhas de segurança em sistemas computacionais que não são conhecidas pelo fornecedor do software no momento em que são descobertas por atacantes. Este tipo de vulnerabilidade permite a construção de estratégias de ataque para obter acesso aos recursos e dados de um sistema. A principal questão em relação às vulnerabilidades de dia zero é como proteger um sistema computacional contra tais ameaças utilizando procedimentos de segurança e ferramentas atuais para identificar possíveis ataques que exploram vulnerabilidades desconhecidas tanto pelos usuários quanto pelos fornecedores de software. Essas vulnerabilidades são particularmente perigosas porque não há correções disponíveis até que sejam descobertas e remediadas pelo fornecedor (Popa, 2013).

2.3.8 Man-In-The-Middle

O ataque Man-In-The-Middle (MITM) ocorre quando um grupo malicioso de terceiros toma o controle de um canal de comunicação entre dois ou mais pontos de saída como pode ser observado na Figura 2. O MITM pode obter acesso ao canal por meio de alguns métodos diferentes, sendo um desses a personificação, onde o atacante convence as vítimas de que esse é um ponto de saída válido para a informação. Por meio desse tipo de ataque, é possível comprometer a confidencialidade, disponibilidade e integridade da mensagem enviada pelo canal, uma vez que se torna viável ler, modificar ou até mesmo apagar os dados em circulação (Conti *et al.*, 2016).

Figura 2 – Ilustração de ataque MITM

Fonte: ATAQUE man-in-the-middle (MitM) (2024)

2.4 TRABALHOS RELACIONADOS

Com o intuito de melhorar a segurança dos dispositivos IoT já foram elaborados diversos guias e trabalhos parecidos acerca desse assunto. Um desses seria o feito por Berlanda (2021), que tem como objetivo o desenvolvimento de um guia de segurança da informação para dispositivos IoT utilizados na Indústria 4.0, analisando os riscos e falhas em seus hardware, software e peopleware, de modo a avaliar o impacto de possíveis ataques que se aproveitem dessas vulnerabilidades, assim como identificar diretrizes e boas práticas para mitigar os riscos de segurança.

Outro trabalho que segue um tema semelhante, seria o proposto por Alhirabi *et al.* (2021) que foca em realizar uma pesquisa a respeito dos requisitos não funcionais, de segurança e privacidade, dos dispositivos IoT. A pesquisa envolve a revisão da evolução de notações, modelos e linguagens que auxiliam na segurança, propor e utilizar uma taxonomia de modo a compará-la as abordagens anteriores, e por fim revisar ferramentas que ajudam a cumprir os requisitos não funcionais citados nas fases de desenvolvimento ou execução do aplicativo.

Além disso, também existe o estudo elaborado por Rocha Junior (2022), no qual foi feito um levantamento de vulnerabilidades presentes em dispositivos IoT nas camadas de percepção, transporte e aplicação, assim como dos métodos de detecção e mitigação dos problemas de segurança encontrados. Não só isso, a pesquisa também aborda possibilidades para o futuro da segurança de dispositivos IoT de modo a indicar quais áreas do sistema devem ser focadas para proteger os dados adquiridos e utilizados pelo sistema.

Dito isso, o presente trabalho se difere aos citados, uma vez que esse guia de programação segura para dispositivos IoT se concentra especificamente na prática de codificação segura, com foco em exemplos de vulnerabilidades de segurança comuns e como evitá-las. Enquanto os guias de Berlanda (2021) e Junior (2022) abordam a segurança de dispositivos IoT de uma perspectiva ampla, incluindo hardware, software, peopleware e métodos de detecção e mitigação de vulnerabilidades, este trabalho se dedica principalmente ao código fonte, fornecendo dicas para programadores. O trabalho de Alhirabi *et al.* (2021) investiga requisitos não funcionais e modelos teóricos para segurança e privacidade, ao passo que o presente guia se baseia em situações reais de programação e em como implementar práticas seguras no dia a dia do desenvolvimento. Assim, pode se dizer que este TCC oferece um recurso para desenvolvedores que buscam melhorar a segurança de seus dispositivos IoT através da prática de programação segura, diferenciando-se dos enfoques mais teóricos e abrangentes dos trabalhos citados.

Os guias da European Union Agency for Cybersecurity (ENISA) (2020), do Open Web Application Security Project (OWASP) (2024) e da GSM Association (GSMA) (2020) também são relevantes nesse contexto. O trabalho feito pela ENISA (2020)

fornece um conjunto abrangente de diretrizes de segurança para dispositivos IoT, cobrindo uma ampla gama de aspectos, desde a concepção e desenvolvimento até a implementação e operação. Este guia é mais focado em políticas de segurança, gestão de risco e práticas recomendadas a nível organizacional e técnico, oferecendo uma visão holística da segurança em dispositivos IoT.

Por outro lado, o guia proposto pelo OWASP (2024) é um recurso prático voltado para a segurança de software em dispositivos IoT. Ele fornece uma lista detalhada de testes de segurança que podem ser aplicados durante o desenvolvimento de software, abordando vulnerabilidades comuns e oferecendo métodos para detectá-las e mitigá-las. Este guia é mais técnico e específico em comparação ao guia da ENISA, focando-se diretamente em testes de intrusão que podem ser realizados em dispositivos IoT de modo a verificar sua segurança.

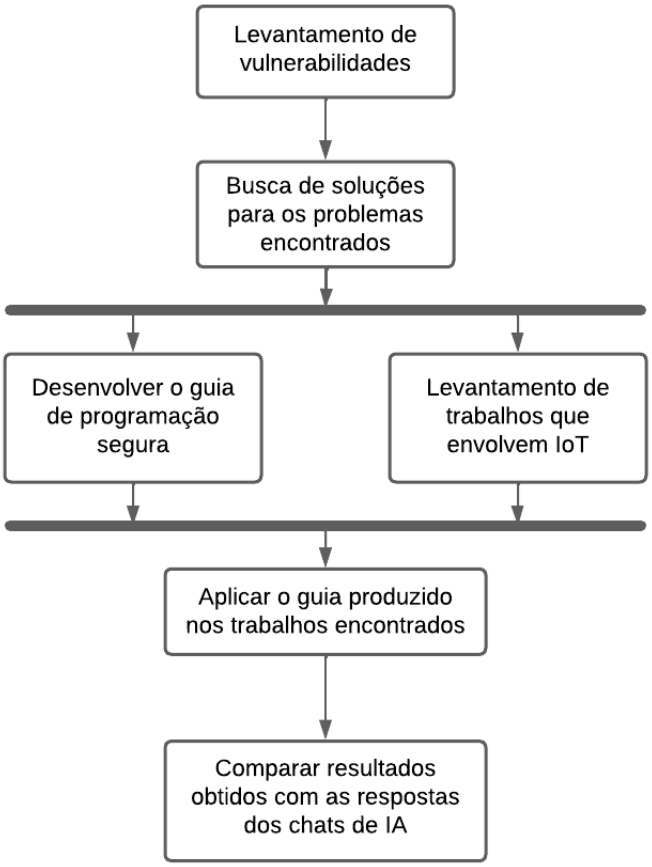
Enquanto isso, o guia realizado pela GSMA (2020) aborda um conjunto de diretrizes desenvolvido para fornecer uma estrutura de segurança para dispositivos IoT, abordando desde a fabricação até a implementação e manutenção dos dispositivos IoT. Ele enfatiza a necessidade de segurança em todo o ciclo de vida desses dispositivos e oferece um esquema de avaliação para garantir a conformidade com as práticas de segurança recomendadas.

Comparando com o presente trabalho, enquanto o trabalho da ENISA (2020) oferece um panorama amplo e organizacional da segurança em IoT, o do OWASP (2024) foca em testes específicos de segurança de software, e o da GSMA (2020) aborda a segurança ao longo de todo o ciclo de vida do dispositivo, o guia aqui proposto se concentra em fornecer diretrizes de codificação segura com exemplos práticos para desenvolvedores de dispositivos IoT. Dessa forma, o trabalho busca preencher uma lacuna ao oferecer um recurso prático e acessível que pode ser utilizado por desenvolvedores após a programação de um dispositivo IoT, além de também demonstrar a possível viabilidade do uso de ferramentas de IA para ajudar na verificação de segurança.

3 METODOLOGIA

Para o desenvolvimento do guia proposto, este trabalho segue a metodologia descrita no fluxograma ilustrado na Figura 3, de modo que cada etapa deste diagrama vai ser explicada nos parágrafos subsequentes deste capítulo.

Figura 3 – Diagrama da metodologia do projeto.



Fonte: Autor (2024)

Inicialmente, será realizado um levantamento de vulnerabilidades em dispositivos IoT com o objetivo de identificar e listar os potenciais riscos decorrentes de práticas inadequadas de codificação, incluindo fatores subjacentes que podem contribuir para essas vulnerabilidades. Essa análise busca compreender a profundidade das ameaças, abrangendo aspectos como integridade, confidencialidade e disponibilidade dos dados.

Analisando as vulnerabilidades encontradas na fase anterior, irá ser feita uma investigação de práticas de programação segura de modo a identificar abordagens eficazes que podem mitigar esses problemas de segurança. Isso possibilitará encontrar

soluções específicas e estratégias que não apenas abordem os riscos imediatos, mas também abordem os fatores que contribuem para a segurança dos dispositivos IoT, de modo que as soluções encontradas servem de base para a criação do guia.

Levando em conta as soluções encontradas para contornar os riscos a segurança de dispositivos IoT, será montado um guia listando as vulnerabilidades causadas por má programação encontradas. Para isso, cada solução identificada irá ser traduzida em diretrizes claras, passos práticos e boas práticas que serão incorporadas ao guia.

Paralelamente à montagem do guia, será realizado um levantamento de outros Trabalhos de Conclusão de Curso (TCC) de alunos dos cursos de engenharia mecatrônica e de controle e automação da UFSC e do curso de engenharia elétrica da UDESC que envolvem dispositivos IoT e disponibilizem o código fonte do trabalho.

Por fim, realizará-se a aplicação das diretrizes e boas práticas delineadas no guia nos projetos identificados, de forma que cada trabalho será analisado, avaliando a conformidade das práticas de programação empregadas com as orientações propostas no guia. Com isso, irá ser possível verificar a eficácia e a relevância do guia em cenários práticos, e também identificar áreas de aprimoramento ou expansão do guia.

3.1 MITIGAR OS PROBLEMAS ENCONTRADOS

Nesta etapa do trabalho, serão mostradas as vulnerabilidades levantadas, assim como meios para solucionar ou mitigar as mesmas. Além das vulnerabilidades encontradas durante o estudo realizado, também será abordada uma vulnerabilidade que, de acordo com as ferramentas de IA, estavam comumente presentes nos códigos de TCCs analisados porém não tinha sido explorada no checklist, sendo essa, a falta de validação de entradas no código. Com isso em mente, serão mostrados exemplos de códigos vulneráveis, assim como de códigos que aplicam a codificação segura de modo a evidenciar e explicar métodos para resolver as falhas de segurança.

3.1.1 Buffer Overflow

Um buffer overflow pode ocorrer quando um programa recebe mais dados do que o esperado ou quando não há verificação adequada do tamanho dos dados antes de armazená-los em um buffer, erro que pode ser observado no Código 3.1, onde a função *vulnerable_function* recebe os dados da variável *input* porém não se protege para caso sejam enviados mais informações do que o buffer é capaz de aguentar.

```
1 #include <stdio.h>
2 #include <string.h>
3
4 void vulnerable_function(char *input) {
```

```
5     char buffer[10];
6     strcpy(buffer, input);
7     printf("Buffer content: %s\n", buffer);
8 }
9
10 int main() {
11     char input[20];
12     printf("Enter input: ");
13     scanf("%s", input);
14     vulnerable_function(input);
15     return 0;
16 }
```

Código 3.1 – Código suscetível a ataques de *buffer overflow*.

Para corrigir ou mitigar o problema de buffer overflow neste código, é possível aplicar algumas técnicas diferentes, tais como usar funções seguras, limitar o tamanho da entrada do código ou utilizar buffers dinâmicos, técnicas essas que podem ser encontradas, respectivamente, nos Códigos 3.2, 3.3 e 3.4. No Código 3.2, foi utilizado um argumento na função *strncpy* que limita a quantidade de caracteres que serão copiados para o buffer, logo impedindo o estouro do buffer. Já no Código 3.3, observa-se a limitação do tamanho do dado que será lido e consequentemente enviado à função vulnerável, de modo que esse não será maior que o tamanho do buffer. Por fim temos também o Código 3.4 em que foi feita a implementação de um buffer dinâmico, assim seu tamanho depende da quantidade de informações recebido, impedindo seu estouro.

```
1 #include <stdio.h>
2 #include <string.h>
3
4 void safe_function(char *input) {
5     char buffer[10];
6     strncpy(buffer, input, sizeof(buffer) - 1);
7     printf("Buffer content: %s\n", buffer);
8 }
```

Código 3.2 – Código de função protegida contra ataques de *buffer overflow*.

```
1 #include <stdio.h>
2 #include <string.h>
3
4 void vulnerable_function(char *input) {
5     char buffer[10];
6     strcpy(buffer, input);
7     printf("Buffer content: %s\n", buffer);
8 }
9
10 int main() {
```

```
    char input[20];
    printf("Enter input: ");
    scanf("%9s", input);  // Limita a entrada a 9 caracteres
    vulnerable_function(input);
    return 0;
}
```

Código 3.3 – Código de função protegida contra ataques de *buffer overflow*.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void safe_function(char *input) {
    size_t input_length = strlen(input);
    char *buffer = malloc(input_length + 1);  // Aloca memoria com
   tamanho adequado
    if (buffer != NULL) {
        strcpy(buffer, input);
        printf("Buffer content: %s\n", buffer);
        free(buffer);
    }
}
```

Código 3.4 – Código de função protegida contra ataques de *buffer overflow*.

3.1.2 Denial of Service

Um ataque de DoS é uma forma de ataque cibernético que visa tornar um serviço indisponível para usuários legítimos, interrompendo ou limitando o acesso ao sistema, rede ou recurso. Isso é geralmente alcançado por meio da sobrecarga de recursos do sistema, como largura de banda de rede, capacidade de processamento da CPU, memória ou conexões de rede.

Os ataques de DoS podem ser realizados de várias maneiras, incluindo o envio de um grande volume de tráfego de rede, exploração de vulnerabilidades de software para travar um serviço específico ou a exploração de limitações de protocolos de rede. O objetivo principal de um ataque de negação de serviço é causar interrupções e causar prejuízos financeiros, de reputação ou operacionais à organização alvo. Uma vulnerabilidade em código que pode resultar em um ataque DoS seria a apresentada no Código 3.5, onde não existe uma verificação do tamanho do dado recebido, logo poderia ocorrer de, ao processar a informação, o consumo de recursos ser mais alto do que o previsto.

```
from http.server import BaseHTTPRequestHandler, HTTPServer

class SimpleHTTPRequestHandler(BaseHTTPRequestHandler):
```

```
4       def do_POST(self):
5           content_length = int(self.headers['Content-Length'])
6           post_data = self.rfile.read(content_length)
7
8           # Processamento dos dados recebidos
9           # Sem verificacao de tamanho de dados
10
11
12  def run(server_class=HTTPServer, handler_class=SimpleHTTPRequestHandler,
        port=8000):
13      server_address = ('', port)
14      httpd = server_class(server_address, handler_class)
15      print(f'Servidor rodando na porta {port}')
16      httpd.serve_forever()
17
18  if __name__ == "__main__":
19      run()
```

Código 3.5 – Código suscetível a ataques de DoS.

Dessa forma, uma maneira de prevenir que o código tente processar dados maiores do que esse é capaz, seria a implementação de um limite do tamanho da carga útil do servidor. Esse método é demonstrado no Código 3.6, de modo que quando o dado recebido for maior que a variável *CargaUtil*, será retornado um erro e não será feita uma tentativa de processamento da informação, evitando possível casos de ataques DoS.

```
1   from http.server import BaseHTTPRequestHandler, HTTPServer
2
3   CargaUtil = 1024  # Definindo o tamanho maximo da carga util
4
5   class SimpleHTTPRequestHandler(BaseHTTPRequestHandler):
6       def do_POST(self):
7           content_length = int(self.headers['Content-Length'])
8           if content_length > CargaUtil:
9               self.send_response(413)  # Numero de erro padrao para
        arquivos grandes demais
10              self.end_headers()
11              self.wfile.write(b'Dados grandes demais, envie um menor.')
12              return
13          post_data = self.rfile.read(content_length)
14
15          # Processar os dados da solicitacao
16
17          # Responder ao cliente
18          self.send_response(200)
19          self.end_headers()
20          self.wfile.write(b'Processamento bem sucedido.')
```

```
21
22 def run(server_class=HTTPServer, handler_class=SimpleHTTPRequestHandler,
       port=8000):
23     server_address = ('', port)
24     httpd = server_class(server_address, handler_class)
25     print(f'Servidor rodando na porta {port}')
26     httpd.serve_forever()
27
28 if __name__ == "__main__":
29     run()
```

Código 3.6 – Código protegido contra ataques de DoS.

3.1.3 Problemas de autentificação

Problemas de autentificação em código comprometem o processo de verificação da identidade dos usuários em um sistema, abrindo portas para acessos não autorizados e potenciais violações de segurança, podendo resultar também em ataques de MITM. Esses problemas podem surgir de diversas maneiras, incluindo autenticação inadequada ou ausente, armazenamento inseguro de credenciais, falta de proteção contra ataques de força bruta e vulnerabilidades no gerenciamento de sessões. Por exemplo, no Código 3.7 a senha está hardcoded como *123456*, o que é inseguro, já que um invasor poderia facilmente adivinhar essa senha ou usar ataques de força bruta para comprometer contas de usuário, além de que a senha está armazenada sem qualquer tipo de criptografia que a proteja de ser descoberta por invasores.

```
1  def fazer_login(usuario, senha):
2      # Verifica se a senha e "123456"
3      if senha == "123456":
4          print("Login bem-sucedido para o usuario", usuario)
5      else:
6          print("Senha incorreta. Tente novamente.")
7
8
9  # Exemplo de uso
10 usuario = input("Digite o nome de usuario: ")
11 senha = input("Digite a senha: ")
12 fazer_login(usuario, senha)
```

Código 3.7 – Código suscetível a ter problemas de autentificação.

Com o intuito de mitigar os problemas de autentificação presentes no exemplo anterior, podem ser feitas correções como as presentes no Código 3.8, de modo que ao cadastrar um usuário, uma função *cadastrar_usuario()* gera um salt aleatório para cada usuário e, em seguida, calcula o hash da senha concatenada com o salt. Durante o login, a função *verificar_senha()* utiliza o mesmo processo para calcular o hash

da senha inserida pelo usuário e compara-o com o hash armazenado no banco de dados. Dessa maneira, as senhas para cada usuário são definidas pelos mesmos e são armazenadas de um modo mais seguro graças a utilização de criptografia, além disso é uma boa prática implementar políticas de senha mais robustas, como a exigência de senhas complexas e a utilização de autenticação de dois fatores.

```
import hashlib
import secrets

# Dicionario para armazenar os usuarios e senhas (simulando um banco de dados)
usuarios = {
    "nome": {
        "senha_hash": "",
        "salt": ""
    }
}

def cadastrar_usuario(usuario, senha):
    salt = secrets.token_hex(16)  # Gerando um salt aleatorio
    senha_hash = hashlib.sha256((senha + salt).encode()).hexdigest()  # Gerando hash da senha + salt
    usuarios[usuario] = {
        "senha_hash": senha_hash,
        "salt": salt
    }
    print("Usuario cadastrado com sucesso!")

def verificar_senha(usuario, senha):
    if usuario in usuarios:
        senha_hash = usuarios[usuario]["senha_hash"]
        salt = usuarios[usuario]["salt"]
        senha_input_hash = hashlib.sha256((senha + salt).encode()).hexdigest()  # Gerando hash da senha inserida
        if senha_input_hash == senha_hash:
            print("Login bem-sucedido para o usuario", usuario)
        else:
            print("Senha incorreta. Tente novamente.")
    else:
        print("Usuario nao encontrado.")

# Exemplo de uso
cadastrar_usuario("vinicius", "senha_segura123")

# Tentativa de login
usuario = input("Digite o nome de usuario: ")
senha = input("Digite a senha: ")
```

```
verificar_senha(usuario, senha)
```

Código 3.8 – Código menos suscetível a problemas de autentificação.

3.1.4 Criptografia falha

Muitas vezes, já que dispositivos IoT no geral não tem grande preocupação em sua segurança, é comum que a criptografia no envio de mensagens seja falha, uma vez que esses não implementam corretamente técnicas de criptografia para proteger a confidencialidade e integridade das mensagens transmitidas. Isso pode resultar na exposição de informações sensíveis a interceptações ou manipulações por parte de adversários maliciosos, sendo outra vulnerabilidade que pode resultar em ataque MITM. Um exemplo disso seria o código no Código 3.9, onde por mais que seja feito uso de criptografia para tentar proteger a mensagem, essa utiliza uma chave simples e fixa, logo não pode ser considerada segura.

```
import socket
def enviar_mensagem(mensagem):
    # Conexao com o servidor
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect(('192.168.1.50', 12345))

    # Criptografia fraca: XOR com uma chave simples
    chave = 'K'
    mensagem_criptografada = ''.join(chr(ord(char) ^ ord(chave)) for
    char in mensagem)

    # Envio da mensagem criptografada
    s.send(mensagem_criptografada.encode())
    s.close()

if __name__ == "__main__":
    enviar_mensagem("Mensagem confidencial")
```

Código 3.9 – Codigo com criptografia falha.

Com isso, observa-se no Código 3.10 um exemplo onde a vulnerabilidade relacionada à criptografia foi mitigada ao implementar práticas de criptografia segura. A chave de criptografia e o IV (Initialization Vector) são gerados aleatoriamente para cada mensagem, dificultando ataques. A mensagem é criptografada usando o algoritmo AES (Advanced Encryption Standard), que é um método considerado robusto de criptografia (Scripcariu *et al.*, 2018). Uma vez que o IV é gerado aleatoriamente, esse é enviado junto com a mensagem criptografada, garantindo que cada mensagem tenha um IV único. No receptor, a mensagem é descriptografada usando a mesma chave e IV, aumentando a confidencialidade da mensagem durante sua transmissão.

```python
import socket
import os
from Crypto.Cipher import AES

# Chave e IV (Initialization Vector) devem ser seguros e secretos
chave = os.urandom(32)  # Chave AES de 256 bits
iv = os.urandom(16)     # IV de 128 bits

def criptografar_mensagem(mensagem, chave, iv):
    cipher = AES.new(chave, AES.MODE_CBC, iv)
    mensagem_criptografada = cipher.encrypt(mensagem)
    return mensagem_criptografada

# Exemplo de como descriptografar as mensagens recebidas
def descriptografar_mensagem(mensagem_criptografada, chave, iv):
    cipher = AES.new(chave, AES.MODE_CBC, iv)
    mensagem_decriptografada = cipher.decrypt(mensagem_criptografada)
    return mensagem_decriptografada

def enviar_mensagem(mensagem):
    # Conexao com o servidor
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect(('192.168.1.50', 12345))

    # Criptografia da mensagem
    mensagem_criptografada = criptografar_mensagem(mensagem, chave, iv)

    # Envio do IV seguido pela mensagem criptografada
    s.send(iv + mensagem_criptografada)
    s.close()

if __name__ == "__main__":
    enviar_mensagem("Mensagem confidencial")
```

Código 3.10 – Código menos suscetível a ter sua criptografia quebrada.

3.1.5 Abertura de portas desnecessárias

Vulnerabilidades causadas pela abertura de portas desnecessárias em um código referem-se à prática de deixar portas de comunicação abertas em um sistema, sem justificativa ou necessidade legítima. Uma porta aberta pode ser explorada por atacantes para acessar o sistema, introduzir malware, extrair dados sensíveis ou realizar ataques de DoS e ataques de força bruta. Isso pode acontecer de várias maneiras, como incluir bibliotecas ou componentes de terceiros que deixam portas abertas por padrão, implementar serviços desnecessários que abrem portas sem autorização explícita ou não configurar corretamente as permissões de firewall para bloquear portas

não utilizadas. Um exemplo disso seria o apresentado no Código 3.11 que apresenta um código simples que recebe a solicitação para abrir uma fechadura eletrônica por meio da porta TCP *12345*, porém acaba deixando ela aberta mesmo após já ter sido utilizada, causando uma possível vulnerabilidade a ser explorada.

```
import socket

def servidor_controle_acesso():
    # Abre uma porta para o servidor de controle de acesso
    servidor = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    servidor.bind(('0.0.0.0', 12345))  # Abrindo a porta 12345
    servidor.listen(5)
    print("Aguardando solicitacoes de desbloqueio na porta 12345...")
    while True:
        cliente, endereco = servidor.accept()
        print(f"Solicitacao de desbloqueio recebida de {endereco}")

        # Verifica se a solicitacao vem de um cliente autorizado (
    exemplo)
        if endereco[0] == '192.168.1.100':
            # Logica para verificar a solicitacao de desbloqueio e
    responder
            resposta = "Fechadura desbloqueada!"
            cliente.send(resposta.encode())
        else:
            resposta = "Acesso negado!"
            cliente.send(resposta.encode())

        cliente.close()

if __name__ == "__main__":
    servidor_controle_acesso()
```

Código 3.11 – Código com porta aberta desnecessariamente.

Para mitigar os riscos causados por essa má prática, a solução mais simples seria a apresentada na Figura 3.12, onde após utilizar uma porta ela deve ser fechada. No exemplo mostrado, isso é feito de modo que após a condição de *KeyboardInterruption* aconteça, a porta será fechada, sendo que é possível trocar a condição para qualquer outra que faça sentido para a aplicação necessária.

```
import socket

def servidor_controle_acesso():
    servidor = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    servidor.bind(('0.0.0.0', 12345))
    servidor.listen(5)
    print("Aguardando solicitacoes de desbloqueio na porta 12345...")
```

```
8
9       try:
10          while True:
11              cliente, endereco = servidor.accept()
12              print(f"Solicitacao de desbloqueio recebida de {endereco}")
13
14              if endereco[0] == '192.168.1.100':
15                  resposta = "Fechadura desbloqueada!"
16                  cliente.send(resposta.encode())
17              else:
18                  resposta = "Acesso negado!"
19                  cliente.send(resposta.encode())
20
21              cliente.close()
22      except KeyboardInterrupt:
23          print("Interrupcao detectada. Encerrando o servidor...")
24      finally:
25          servidor.close()  # Fecha o socket do servidor
26
27  if __name__ == "__main__":
28      servidor_controle_acesso()
```

Código 3.12 – Código com fechamento da porta utilizada.

3.1.6 Vulnerabilidades em códigos de terceiros

As vulnerabilidades em códigos de terceiros representam uma preocupação na segurança cibernética, especialmente em ambientes de desenvolvimento de software, onde a integração de bibliotecas e frameworks de terceiros é uma prática comum. Esses riscos podem surgir devido a falhas de projeto, implementação inadequada, dependências desatualizadas, falta de auditoria de segurança e até mesmo comprometimento da cadeia de suprimentos de software. Quando não identificadas e corrigidas, essas vulnerabilidades podem ser exploradas por invasores para comprometer a segurança de sistemas inteiros.

Dessa forma, para mitigar o risco associado às vulnerabilidades em códigos de terceiros, é essencial realizar avaliações de segurança das dependências de terceiros antes da sua incorporação nos projetos. Assim, uma ferramenta que pode auxiliar nesse processo seria o site *security.snyk.io*, no qual é possível pesquisar sobre bibliotecas conhecidas e verificar se essas possuem riscos de segurança ou não. Outro fator importante seria manter atualizado qualquer código ou biblioteca que esteja sendo usada no projeto, uma vez que isso pode diminuir os riscos de esses serem vulneráveis.

3.1.7 Vulnerabilidades de dia zero

Uma vulnerabilidade de dia zero (zero-day vulnerability) é uma falha de segurança em software ou hardware que é desconhecida pelos responsáveis pela sua manutenção e desenvolvimento. O termo *dia zero* refere-se ao fato de que os desenvolvedores têm zero dias para corrigir a vulnerabilidade antes que ela possa ser explorada, uma vez que, na maioria dos casos, a descoberta da vulnerabilidade e sua exploração por atacantes ocorrem simultaneamente ou em um curto espaço de tempo.

Essas vulnerabilidades podem ser causadas por vários fatores, incluindo erros de codificação, falhas de design, validação inadequada de entradas e saídas, ou até mesmo configurações incorretas. Uma vez que uma vulnerabilidade de dia zero é descoberta por um atacante, ela pode ser usada para lançar ataques antes que uma correção ou atualização esteja disponível, o que a torna particularmente perigosa. Dessa forma, para ajudar com esse problema, uma solução seria acessar sites como o *zerodayinitiative.com*, sendo que esse é um banco de dados que disponibiliza informações a respeito de vulnerabilidades de dia zero conhecidas, assim possibilitando evitar o uso de programas ou códigos com suas seguranças comprometidas.

3.1.8 Falta de verificação de entradas

A falta de verificação adequada das entradas no código é uma vulnerabilidade comum que pode levar a diversos problemas de segurança, incluindo injeção de comandos, ataques de buffer overflow e execução de código malicioso. Quando entradas fornecidas pelo usuário não são validadas ou sanitizadas corretamente, como ocorre no Código 3.13, um atacante pode enviar dados especialmente formatados para explorar falhas no software. Isso pode comprometer a integridade, a confidencialidade e a disponibilidade do sistema.

No código vulnerável, a função *eval()* é usada diretamente nos dados recebidos, sem qualquer verificação ou sanitização. Isso permite que um atacante envie comandos maliciosos que serão executados pelo servidor, o que pode levar a sérias consequências de segurança.

```
1  import socket
2
3  def servidor():
4      s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
5      s.bind(('0.0.0.0', 12345))
6      s.listen(5)
7      print("Aguardando conexoes...")
8
9      while True:
10         cliente, endereco = s.accept()
11         print(f"Conexao recebida de {endereco}")
```

```
12
13          dados = cliente.recv(1024).decode()
14          print(f"Dados recebidos: {dados}")
15
16          # Vulnerabilidade: os dados recebidos nao sao verificados
17          resultado = eval(dados)
18          cliente.send(f"Resultado: {resultado}".encode())
19
20          cliente.close()
21
22  if __name__ == "__main__":
23      servidor()
```

Código 3.13 – Código sem verificação de entrada

Uma possível correção é demonstrada no Código 3.14, no qual a entrada do usuário é verificada antes de ser processada. Apenas dados numéricos são aceitos, e qualquer outra entrada é rejeitada com uma mensagem de *Entrada inválida*. Isso impede a execução de código arbitrário e protege o sistema contra explorações. Além disso, a operação realizada é segura e não apresenta risco de injeção de comandos. A inclusão de um bloco *try-except* também garante que quaisquer erros sejam tratados de forma segura, enviando uma mensagem de erro apropriada ao cliente sem comprometer a integridade do servidor.

```
1  import socket
2
3  def servidor():
4      s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
5      s.bind(('0.0.0.0', 12345))
6      s.listen(5)
7      print("Aguardando conexoes...")
8
9      while True:
10         cliente, endereco = s.accept()
11         print(f"Conexao recebida de {endereco}")
12
13         dados = cliente.recv(1024).decode()
14         print(f"Dados recebidos: {dados}")
15
16         try:
17             # Verificacao adequada das entradas recebidas
18             if dados.isnumeric():
19                 resultado = int(dados) * 2  # Exemplo de operacao segura
20                 cliente.send(f"Resultado: {resultado}".encode())
21             else:
22                 cliente.send("Entrada invalida".encode())
23         except Exception as e:
24             cliente.send(f"Erro: {str(e)}".encode())
```

```
25
26          cliente.close()
27
28 if __name__ == "__main__":
29     servidor()
```

Código 3.14 – Código com verificação de entrada

3.2 CHECKLIST

Para lidar com as vulnerabilidades em códigos já apresentadas, foi feito um checklist, que pode ser observado no Quadro 1, para auxiliar na programação segura de dispositivos IoT, de modo a facilitar a visualização dos métodos apresentados no trabalho para a segurança do código. Dessa forma, com o intuito de diminuir o número de vezes que o código teria de ser alterado caso esse não esteja de acordo com o checklist, este está organizado de modo que problemas que têm maior probabilidade de gerarem maiores alterações ao programa quando aplicando as correções, aparecem mais acima na lista, por exemplo, se um código utiliza funções de uma biblioteca vulnerável, esse precisaria trocar por uma outra biblioteca que possuísse funções parecidas, logo, necessitando maiores alterações ao código original se comparado a soluções para mitigar outros problemas de segurança, tais como o buffer overflow.

3.3 APLICAR MÉTODOS ENCONTRADOS EM OUTROS TRABALHOS

Para que fosse possível realizar a validação dos métodos encontrados no trabalho, foi desenvolvido um levantamento de TCCs por meio do Repositório Institucional da UFSC (https://repositorio.ufsc.br) e da Biblioteca Universitária da UDESC (https://pergamumweb.udesc.br/biblioteca/index.php).

Com o intuito de encontrar trabalhos adequados a proposta deste guia, a busca no repositório da UFSC foi feita de modo à considerar apenas textos encontrados nas coleções de *TCC Engenharia Mecatrônica (Joinville)* e *TCC Engenharia de Controle e Automação*, utilizando a palavra *IoT* para realizar a busca. Dessa forma, foram encontrados 24 trabalhos com esses filtros de pesquisa, porém apenas 5 deles envolviam a programação de dispositivos IoT e disponibilizavam o código fonte em formato de texto.

Já para a busca no acervo da UDESC, essa foi feita novamente utilizando a palavra de busca *IoT* e selecionando *Monografias Digitais* como sendo o tipo de obra. Esses filtros encontram 20 trabalhos, porém apenas 5 deles pertencem a alunos do curso de engenharia elétrica e possuem códigos fonte em formato de texto disponibilizados.

Quadro 1 – Checklist

☐ Mantenha todas as bibliotecas e frameworks atualizadas para corrigir quaisquer vulnerabilidades conhecidas (verificar as Seções 3.1.6 e 3.1.7 para mais informações).
☐ Realize uma avaliação de segurança detalhada de todas as dependências de terceiros antes de incorporá-las ao projeto (verificar as Seções 3.1.6 e 3.1.7 para mais informações).
☐ Armazene credenciais e outros dados sensíveis de modo seguro utilizando criptografia (verificar a Seção 3.1.3 para mais informações).
☐ Utilize algoritmos de criptografia robustos ao realizar envio de dados (verificar a Seção 3.1.4 para mais informações).
☐ Implemente mecanismos de limitação de taxa e controle de acesso para evitar ataques de negação de serviço (verificar a Seção 3.1.2 para mais informações).
☐ Implemente mecanismos de verificação de entrada para evitar executar códigos maliciosos enviados (verificar a Seção 3.1.8 para mais informações).
☐ Revise o código para garantir que apenas as portas necessárias para o funcionamento do dispositivo IoT estejam abertas e feche essas após serem utilizadas (verificar a Seção 3.1.5 para mais informações).
☐ Utilizar funções seguras para manipulação de strings, tais como *strncpy*. (verificar a Seção 3.1.1 para mais informações)
☐ Verificar os limites de todos os buffers de entrada para garantir que não possam ser excedidos ou utilizar buffers dinâmicos. (verificar a Seção 3.1.1 para mais informações)

Fonte: Produzido pelo autor

Assim, com os códigos encontrados nos trabalhos, será feita uma verificação se esses seriam seguros ou não utilizando os pontos apresentados no checklist. Após isso, afim de tentar confirmar sobre as vulnerabilidades encontradas, os mesmos códigos irão ser colocados nas ferramentas de IA ChatGPT (versão 3.5) e Gemini (modelo 1.0 Pro). Para realizar a análise com essas ferramentas de IA, foi utilizado um chat novo para cada código de trabalho diferente. A pergunta realizada para as ferramentas foi a seguinte: "O código a seguir possui vulnerabilidades? Se sim, quais seriam?".

4 RESULTADOS

Nesta seção, será apresentada a análise realizada dos códigos extraídos de outros TCCs, com o objetivo de identificar e discutir vulnerabilidades comuns em dispositivos IoT. Utilizando o checklist de segurança desenvolvido anteriormente, foram examinados segmentos de código para verificar a presença dos problemas discutidos no trabalho. Cada exemplo analisado será acompanhado de uma verificação pelo ChatGPT e pelo Gemini, realizada durante o mês de maio de 2024, afim de possibilitar uma comparação entre os diagnósticos obtidos.

4.1 MONITORAMENTO DO CONSUMO DE ENERGIA ELÉTRICA APLICADO A CARGAS RESIDENCIAIS

Esse foi um trabalho realizado por Greselle (2022), que tinha como objetivo o desenvolvimento de um módulo que pudesse medir a potência elétrica média de eletrodomésticos conectados a ele. Para isso foi utilizado um ESP32 para controlar os sensores necessários para a aplicação, logo, a análise de segurança foi efetuada no código implementado nesse microcontrolador.

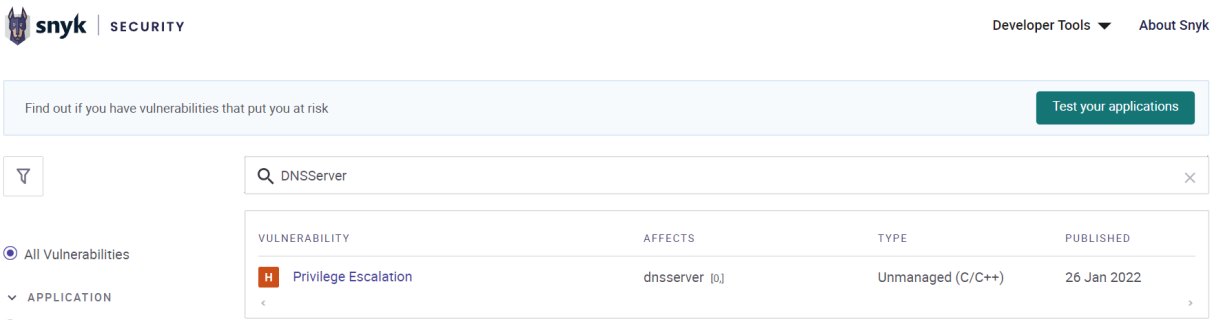
Primeiramente, verificando as bibliotecas utilizadas no programa, que podem ser observadas no Código 4.1 nas linhas 3 e 10, tem-se que ambas a DNSServer e a ArduinoJson, de acordo com as informações encontradas no site snyk, apresentam versões com vulnerabilidades conhecidas, como pode ser observado nas Figuras 4 e 5, logo seria importante tomar cuidado ao utilizá-las.

```
1 # include <FS.h>
2 # include <ArduinoOTA.h>
3 # include <DNSServer.h> // Local DNS Server used for redirecting all
    requests to the config
4 # include <Ethernet.h>
5 # include <SPI.h>
6 # include <WiFi.h>
7 # include <WiFiClient.h>
8 # include <BlynkSimpleEsp32.h>
9 # include <ArduinoJson.h> // Gerencia a gravacao de dados em arquivo .
    json
```

Código 4.1 – Trecho do código do trabalho de Greselle (2022, p. 34.)

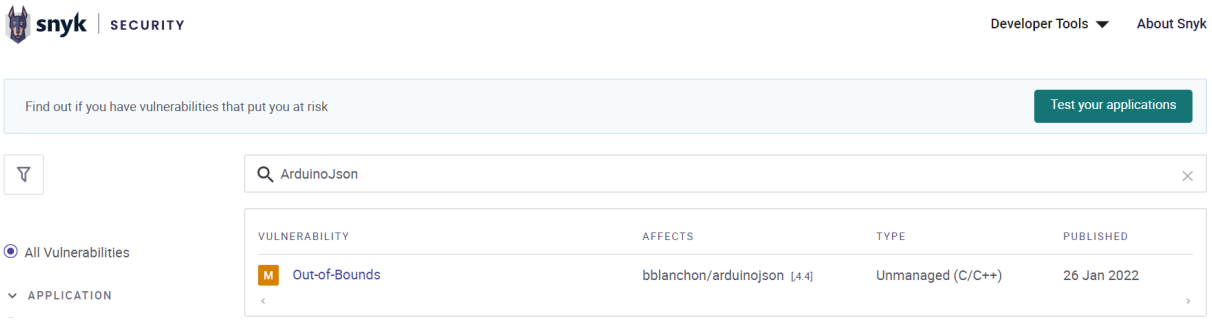
Além desse, ao realizar a análise do código, é possível observar que no trecho de Código 4.2 se encontra uma vulnerabilidade, uma vez que ocorre o armazenamento de credenciais hardcoded sem o devido cuidado de criptografar essas informações. Além disso, no Código 4.3, é feita uma transmissão de dados pela porta serial do

Figura 4 – Pesquisa realizada no site Snyk para DNSServer



Fonte: SNYK Vulnerability Database (2024)

Figura 5 – Pesquisa realizada no site Snyk para ArduinoJson



Fonte: SNYK Vulnerability Database (2024)

ESP32 onde não acontece o uso de algoritmos de criptografia para proteger o conteúdo da mensagem. Não só isso, no Código 4.4 na linha 6, existe a possibilidade de ocorrer um buffer overflow, uma vez que poderia acontecer de o arquivo json ser alterado e, por ser feito uso da função strcpy que não é segura, poderiam ser copiadas informações prejudiciais no programa, possibilitando também que ocorresse um ataque de DoS como resultado do estouro do buffer.

```
1 // WiFi data
2 char ssid [] = "FASTVILLE-3691-2G"; // WiFi Credential
3 char pass [] = "ABCDEFGHI"; // WiFi Password
4 char server [] = "blynk.cloud"; // Blynk local server IP address
```

Código 4.2 – Trecho do código do trabalho de Greselle (2022, p. 41.)

```
1 // Transmissao
2 Serial.print(tensao);
3 Serial.print(separador);
4 Serial.print(corrente);
5 Serial.println();
```

Código 4.3 – Trecho do código do trabalho de Greselle (2022, p. 39.)

```
1 if(json.success())
```

```
2 {
3     // Copiando as variaveis salvas previamente no aquivo json
4     para a memoria do ESP.
5     imprimirSerial ( true , "arquivo json analisado.") ;
6     strcpy(AUTH,json["AUTH"]);
7 }
```

Código 4.4 – Trecho do código do trabalho de Greselle (2022, p. 36.)

De acordo com o ChatGPT, o código apresentado possui vulnerabilidades de segurança, sendo as principais, as credenciais de Wi-Fi e token de autenticação Blynk estarem expostas diretamente no código, uma senha fraca para o Access Point, falta de validação adequada para os dados lidos do arquivo de configuração JSON, ausência de criptografia nas comunicações com o servidor Blynk e pela serial, risco de buffer overflow ao usar funções como strcpy sem verificação de tamanho, duplicação de código aumentando o risco de erros, modo de debug ativo que pode revelar informações sensíveis, e falta de mecanismos robustos de autenticação e autorização. Essas vulnerabilidades expõem o sistema a riscos de interceptação de dados, acesso não autorizado, e possíveis ataques que podem comprometer a integridade e a segurança do dispositivo.

Já por parte do Gemini, ele responde que o código enviado apresenta falhas graves de segurança, como armazenamento inseguro de credenciais, ausência de validação de entrada, uso de bibliotecas de terceiros não confiáveis, falta de atualizações de software e debug habilitado em produção.

4.2 ANÁLISE DA IMPLANTAÇÃO DE UMA REDE LORAWAN PARA SISTEMAS EMBARCADOS

O trabalho desenvolvido por Arruda (2019), foca no estudo e implantação de uma rede LoRaWAN, além de testes para verificar sua funcionalidade. O projeto propôs a implantação completa de uma rede LoRaWAN, detalhando sua arquitetura e configuração, com o objetivo de estudar seu funcionamento e viabilidade.

Observando o código utilizado no Arduino no projeto, nota-se que existe um problema no armazenamento de informações sensíveis, uma vez que no Código 4.5 variáveis de autentificação estão sendo definidas de modo hardcoded e são armazenadas sem a utilização de criptografia. Além disso, no Código 4.6, mesmo que seja possível ver o cuidado para não ocorrer o estouro da capacidade máxima do buffer, não é feita a verificação dos dados recebidos na linha 4 do código antes de utilizá-los na variável *buffer*, criando uma brecha para o recebimento de trechos de código maliciosos.

```
1 lora.setId("00 fb 3f bf", "669deae72a18671c", "669deae72a18671c");
```

```
2 lora.setKey("9d 96 3d 72 1f aa 67 f5 db 9e cb a3 99 ec cd 9e", "6f 0d 90
      a8 99 a7 ca bc 5e e9 41 33 89 0b 72 c7", "6f 0d 90 a8 99 a7 ca bc 5e
      e9 41 33 89 0b 72 c7");
```

Código 4.5 – Trecho do código do trabalho de Arruda (2019, p. 59.)

```
1  short length;
2  short rssi;
3  memset(buffer, 0, 256);
4  length = lora.receivePacket(buffer, 256, &rssi);
5
6  if(length)
7  {
8      SerialUSB.print("Length is: ");
9      SerialUSB.println(length);
10     SerialUSB.print("RSSI is: ");
11     SerialUSB.println(rssi);
12     SerialUSB.print("Data is: ");
13     for(unsigned char i = 0; i < length; i ++)
14     {
15         SerialUSB.print("0x");
16         SerialUSB.print(buffer[i], HEX);
17         SerialUSB.print(" ");
18     }
19     SerialUSB.println();
20 }
```

Código 4.6 – Trecho do código do trabalho de Arruda (2019, p. 60.)

Seguindo a análise feita pelo ChatGPT, o código fornecido possui várias vulnerabilidades de segurança e problemas de implementação que devem ser abordados. Entre as principais questões identificadas estão o armazenamento inseguro de credenciais sensíveis diretamente no código-fonte, a ausência de validação de entrada, permitindo que dados maliciosos ou inválidos sejam processados e a exposição de informações sensíveis na interface serial, aumentando o risco de vazamento de informações. Além disso, o ChatGPT também citou possíveis problemas relacionados à concorrência e controle de fluxo, onde o uso de variáveis globais compartilhadas sem proteção adequada poderia levar a condições de corrida em casos de interrupções, porém, como o código do trabalho não apresenta programação em threads, entende-se que essa não seria uma vulnerabilidade real no trabalho.

Assim como o ChatGPT, o Gemini também identificou vulnerabilidades de segurança no código LoRaWAN analisado, de modo que ele citou o armazenamento inseguro de credenciais no código fonte, falta de validação de entrada, e vazamento potencial de informações pela interface serial. Para mitigar essas vulnerabilidades, recomenda-se o uso de técnicas seguras de armazenamento de credenciais, imple-

mentação de validação de entrada, controle de acesso à interface serial, e revisão periódica do código.

4.3 AVALIAÇÃO E PROJETO DE LOCALIZAÇÃO INDOOR UTILIZANDO BLUETOOTH

No trabalho desenvolvido por Paula (2019), foi abordada a avaliação de localização em ambientes internos utilizando a tecnologia bluetooth. Dessa forma, utilizando 5 dispositivos com a funcionalidade de bluetooth, foram implementados alguns métodos diferentes para descobrir a posição desse em um espaço fechado.

Ao verificar o código, observa-se nos Códigos 4.7 e 4.8 o armazenamento do nome dos dispositivos que estão sendo usados na busca de maneira hardcoded, possibilitando que pessoas mal intencionadas coloquem o mesmo nome em outros dispositivos, de modo a atrapalhar os resultados obtidos. O efeito disso pode ser percebido nos Códigos 4.9 na linha 2 e 4.10 na linha 4, onde após ser feita a busca pelos dispositivos Bluetooth na área, apenas é verificado seus nomes, logo, resultando em possíveis falhas de segurança caso outros com o mesmo nome estivessem por perto.

```
1 NOME_DISPOSITIVO = "Mi A3"
```

Código 4.7 – Trecho do código do trabalho de Paula (2019, p. 39.)

```
1 nome_disp = "Xiaomi Smart Band 7 25AA"
```

Código 4.8 – Trecho do código do trabalho de Paula (2019, p. 41.)

```
1 def escaneia_dispositivos():
2     dispositivos_proximos = bluetooth.discover_devices(duration=8,
     lookup_names=True, flush_cache=True)
3     for addr, name in dispositivos_proximos:
4         if name == NOME_DISPOSITIVO:
5         return addr
6     return None
```

Código 4.9 – Trecho do código do trabalho de Paula (2019, p. 39.)

```
1 async def scan_devices():
2     rssi_encontrado = 0
3     scanner = BleakScanner()
4     devices = await scanner.discover()
5     for device in devices:
6         if device.name == nome_disp:
7             print("Encontrado")
8             rssi_encontrado = device.rssi
9             print(rssi_encontrado)
10        else :
```

```
11              print("Nao encontrado")
12      return rssi_encontrado
```

Código 4.10 – Trecho do código do trabalho de Paula (2019, p. 41.)

A análise realizada pelo ChatGPT identificou várias vulnerabilidades. Primeiro, foram encontrados valores hardcoded, como o nome do dispositivo Bluetooth, o que dificulta a manutenção e escalabilidade do código. Segundo, há falta de validação robusta dos dados recebidos dos dispositivos Bluetooth, permitindo a aceitação de dispositivos não autorizados. Terceiro, o tratamento de erros é insuficiente, pois não há manejo adequado de exceções que possam ocorrer durante a descoberta de dispositivos ou outros processos críticos. Além disso, o ChatGPT diz que a comunicação via Bluetooth não garante a segurança, pois faltam mecanismos de criptografia para proteger contra interceptações, porém, não ocorre troca de dados entre os dispositivos, logo, essa vulnerabilidade não seria real. Por fim, não há medidas para proteger fisicamente os dispositivos contra acessos não autorizados.

A avaliação conduzida pelo Gemini também destacou vulnerabilidades no trabalho, citando primeiramente o fato de que a entrada do usuário para o nome do dispositivo não é validada, possibilitando injeção de código e execução remota por dispositivos maliciosos. Essas falhas podem resultar em roubo de dados, instalação de malware ou até mesmo controle total do sistema. Além disso, os códigos estão vulneráveis a DoS, nos quais dispositivos maliciosos podem sobrecarregar o sistema com respostas falsas.

4.4 DESENVOLVIMENTO DE FIRMWARE PARA TESTES DE PRODUÇÃO DE PRODUTOS IOT

Este projeto proposto por Garcia (2022) envolve o desenvolvimento de um firmware para automatizar o teste de produtos na linha de gateways de IoT, visando reduzir o tempo de testes e minimizar falhas humanas. Com isso em mente, utiliza técnicas de programação concorrente, testes de hardware e software, e uma interface web para monitoramento.

Neste trabalho, apenas foi disponibilizado o pseudocódigo utilizado, logo a análise de segurança será feita considerando possíveis cuidados que deveriam ser tomados para evitar vulnerabilidades ao fazer o código definitivo. Inicialmente, seria importante escolher bibliotecas que possuam as funções necessárias para o código funcionar, porém checar previamente se essas escolhidas possuem ou não vulnerabilidades conhecidas e publicadas.

Além disso, no Código 4.11 na linha 3 é utilizada uma função para executar um código no destino especificado, logo seria importante a inserção de uma verificação a respeito de se o caminho existe e se a informação no arquivo não foi alterada

por um invasor, não só isso, manter o path hardcoded no código também facilitaria para um possível hacker descobrir onde ele deveria realizar alterações. Outro cuidado necessário seria em partes como a demonstrada no Código 4.12, em que, para evitar que outras pessoas tenham acesso a informação na mensagem que irá ser enviada na linha 4, essa deveria ser criptografada previamente.

```
1 func LoRaExecution():
2     protocol.RunLoRaController()
3     process <- exec.Output("path/to/lora-controller/bin", "status")
4     assert string.Contains(process, "running") == true
```

Código 4.11 – Trecho do pseudocódigo do trabalho de Garcia (2022, p. 104.)

```
1 func TestMessageHandler(): \\ LoRa case
2     protocol.MQTT()
3     mqttClient <- protocol.MQTTClient()
4     message <- protocol.client.Publish(gatewayIP, "gateway/teste", 1, "
    message")
5     assert protocol.LROK == true
6     assert protocol.ZBOK == false
```

Código 4.12 – Trecho do pseudocódigo do trabalho de Garcia (2022, p. 104.)

A análise realizada pelo ChatGPT destaca algumas vulnerabilidades. Primeiramente, a falta de tratamento de erros, onde funções como OpenFile, ReadFile e WriteFile não possuem tratamento adequado, pode levar a falhas se os arquivos não existirem, não puderem ser abertos ou escritos. Além disso, o uso de *exec.Output* para executar comandos externos pode ser perigoso se os caminhos ou parâmetros forem manipuláveis, sendo essencial validar e sanitizar todas as entradas. A publicação em tópicos MQTT (*protocol.client.Publish*) também deve validar as entradas para evitar ataques de injeção de código ou comando. Portanto, é crucial validar os dados antes de escrever para garantir que sejam seguros e corretos e por fim, é importante evitar hardcoding de valores sensíveis como *macGw* e *ipGw* diretamente no código, utilizando variáveis de ambiente ou arquivos de configuração seguros para gerenciar essas informações.

A verificação realizada pelo Gemini também identificou diversas vulnerabilidades potenciais no pseudocódigo apresentado. No módulo *protocol*, os caminhos para os binários de execução estão definidos diretamente no código, sendo mais seguro armazená-los em configurações ou variáveis de ambiente. No módulo *connectivity*, o uso do comando *exec.Output* para executar programas externos pode ser perigoso sem validação adequada dos dados de entrada, recomendando-se o uso de bibliotecas específicas para cada protocolo de comunicação. No módulo *io*, a falta de validação do conteúdo dos arquivos lidos e a ausência de verificação das operações de abertura e fechamento de arquivos representam riscos de injeção de código malicioso

e vazamentos de recursos. No módulo *sensors*, os caminhos para arquivos estão diretamente no código e o armazenamento de dados sensíveis em strings pode requerer medidas de segurança adicionais.

4.5 GEN.IO: DESENVOLVIMENTO E APLICAÇÃO DE DISPOSITIVO IOT PARA COLETA DE DADOS DE PRODUÇÃO EM AMBIENTE INDUSTRIAL

O trabalho apresentado por Chilanti (2022) desenvolveu uma solução para coleta e monitoramento de dados de produção, juntamente com o mapeamento dos processos e diagnóstico de equipamentos em uma empresa. A solução inclui um produto de hardware baseado em IIoT (Industrial Internet of Things), que conecta sensores a máquinas e equipamentos industriais, e uma aplicação de software para tratamento e exibição de informações. O firmware do dispositivo utiliza programação concorrente, sistemas de tempo real e o protocolo de comunicação MQTT voltado para IoT.

Analisando o código utilizado neste projeto, encontra-se trechos como o mostrado no Código 4.13 na linha 5, onde ocorre o *publish* de informações pelo uso de MQTT, porém essas mensagens não são criptografas previamente, e como pode ser observado no Código 4.14, mesmo que tenha sido feito uso da porta cifrada do MQTT, sendo essa a 8883, também foi utilizada a 1883 que se comunica por texto claro. Adicionalmente, no Código 4.15 nas linhas 3 e 4 observam-se duas possíveis vulnerabilidades, sendo elas o fato de que o nome dos arquivos que possuem informações sensíveis é de fácil acesso no código possibilitando a descoberta desses dados, e a outra seria a utilização da função *readFile* sem verificar a informação nos arquivos, correndo o risco de que essas informações tenham sido alteradas e possuam linhas de código prejudiciais.

```
1  if (MQTT.connected()) {
2      if (sendTimestamp == "1"){
3          double timeStampValue = xx_time_get_time();
4          const char *sensor1ValueZeroMsg = (timeStampText +
       timeStampValue + valueZero).c_str();
5          MQTT.publish(sensor1Topic.c_str(), sensor1ValueZeroMsg);
6      }
7      else{
8          MQTT.publish(sensor1Topic.c_str(), "{\"value\":0}");
9      }
10     vTaskSuspend(NULL);
11 }
```

Código 4.13 – Trecho do código do trabalho de Chilanti (2022, p. 52.)

```
1 ports:
2     - "1883:1883"
```

```
3     - "8883:8883"
```

Código 4.14 – Trecho do código do trabalho de Chilanti (2022, p. 73.)

```
1 static void WifiTaskRun(void* param) {
2     (void)param;
3     ssid = readFile("/inputSSID.txt");
4     password = readFile("/inputPASSWORD.txt");
```

Código 4.15 – Trecho do código do trabalho de Chilanti (2022, p. 53.)

Conforme analisado pelo ChatGPT, o software apresenta insuficiente validação de entrada, o que pode permitir a injeção de valores maliciosos. Além disso, operações críticas como leitura de arquivos e comandos de reinicialização não possuem tratamento de erros adequado, podendo ser exploradas para causar interrupções no serviço. O armazenamento de credenciais em texto simples representa uma vulnerabilidade significativa.

De acordo com o Gemini, as vulnerabilidades do códigos seriam, primeiramente, o armazenamento de credenciais em texto claro é inseguro, pois um invasor com acesso físico poderia facilmente obter informações sensíveis. Além disso, há uma validação limitada de entradas, o que pode permitir que um invasor altere arquivos de configuração e comprometa o sistema. Também não há checagem de entradas, o que pode permitir a leitura de arquivos arbitrários. O código carece de um gerenciamento robusto de erros, não verificando retornos de funções críticas, o que pode levar a estados inesperados. Finalmente, é importante manter o Arduino IDE e bibliotecas atualizadas para evitar vulnerabilidades conhecidas.

4.6 SISTEMA IOT PARA MONITORAMENTO DO CONSUMO DE ÁGUA E CONTROLE DA HIDRATAÇÃO CORPORAL

Este trabalho desenvolvido por Pontes (2022), apresenta um sistema IoT para monitoramento do consumo de água e controle da hidratação corporal. O sistema mede a quantidade de água consumida e envia os dados para um aplicativo mobile, que armazena as informações e monitora a hidratação. Visando baixo custo, foi criado um protótipo de garrafa com componentes eletrônicos necessários para a medição e monitoramento através do aplicativo.

Na análise do código fornecido no trabalho, observa-se no Código 4.16 o armazenamento de credenciais de Wi-Fi diretamente no código sem a utilização de criptografia. Outra vulnerabilidade seria a demonstrada no Código 4.17 na linha 6, na qual é utilizado HTTP (Hypertext Transfer Protocol) para enviar a informação do valor da vazão para uma planilha do Google Sheets por meio de um URL (Uniform Resource Locator) sem usar um algoritmo de criptografia, possibilitando que, nesse caso, não

só seja possível para um invasor ter acesso a informação que está sendo enviada, mas também alterar ela ou seu destino.

```
#define ssid "_"
#define password "_"
```

Código 4.16 – Trecho do código do trabalho de Pontes (2022, p. 56.)

```
void dados_print(){
    HTTPClient http;
    String url="https://script.google.com/macros/s/"+GOOGLE_SCRIPT_ID+"/
   exec?"+"value1="+vazao;
    // Endereco web responsavel por enviar as informacoes a planilha
    http.begin(url);
    int httpCode = http.GET();
    http.end();
    Serial.print (F("Feito")); // Confirmacao de envio das informacoes
}
```

Código 4.17 – Trecho do código do trabalho de Pontes (2022, p. 57.)

Segundo o ChatGPT, o código apresenta potenciais vulnerabilidades de segurança, citando inicialmente as credenciais de rede, como SSID e senha, que estão armazenadas em texto claro, o que pode facilitar o acesso não autorizado. Não só isso, a falta de validação adequada da entrada de dados na construção da URL pode deixar o sistema vulnerável a ataques de injeção de código. Outra preocupação é a gestão limitada de erros, especialmente ao lidar com solicitações HTTP, onde a falta de tratamento robusto de erros pode levar a falhas silenciosas. A dependência de funções de rede externas também pode introduzir riscos de segurança, especialmente se o servidor externo não for confiável. Por fim, o ChatGPT também aponta que a manipulação de interrupções e pulsos requer uma revisão cuidadosa para garantir a robustez e evitar condições de corrida, porém, pelo código não incluir threads, não é possível que ocorra uma situação de corrida.

Para o Gemini, as vulnerabilidades no projeto envolvem problemas como armazenamento de credenciais hardcoded no código, falta de validação de entrada, uso de HTTP não seguro e falhas no gerenciamento de erros representam riscos à segurança do dispositivo. Por isso, recomenda-se a implementação de medidas como armazenamento criptografado de credenciais, validação rigorosa de entrada e uso de HTTPS para comunicação segura. Ademais, é sugerido considerar melhorias como autenticação, criptografia de dados e atualizações de firmware para fortalecer a segurança do sistema.

4.7 BEBEDOURO IOT PARA MONITORAMENTO E CONTROLE DA HIDRATAÇÃO EM IDOSOS INSTITUCIONALIZADOS

O projeto desenvolvido por Freitag (2021) envolveu a criação de um bebedouro IoT para instituições de idosos, visando diminuir problemas de hidratação nessa faixa etária. O dispositivo mede constantemente a quantidade de água ingerida, armazena os dados na nuvem e os disponibiliza em um aplicativo. A comunicação por radiofrequência é utilizada para diferenciar os pacientes, com cada idoso recebendo um copo distinto que deve acompanhar sua rotina.

Ao analisar o código presente no trabalho, nota-se, mais uma vez, no Código 4.18 a inclusão de credenciais de Wi-Fi diretamente no código-fonte sem proteção criptográfica. Além disso, assim como no projeto analisado anteriormente a esse, também foi utilizado o HTTP para o envio de mensagens, como pode ser visto no Código 4.19 entre as linhas 4 e 7, que demonstra o uso do URL para enviar dados sem criptografá-los, resultando em uma possível vulnerabilidade.

```
1 const char* ssid = "rede_wifi";
2 const char* password = "senha_wifi";
```

Código 4.18 – Trecho do código do trabalho de Freitag (2021, p. 48.)

```
1  void dados_print(String params){
2      digitalWrite(ledAmar, HIGH); // Durante o envio das informacoes, os
       LEDs verde e amarelo estarao acesos, indicando a etapa
3      HTTPClient http;
4      String url="https://script.google.com/macros/s/"+GOOGLE_SCRIPT_ID+"/
       exec?"+params; // Endereco web responsavel por enviar as informacoes
       a planilha
5      // Detalhe nas variaveis GOOGLE_SCRIPT_ID e params, responsaveis por
        fornecer o codigo unico de aplicativo web, gerado pelo script da
       planilha, e os parametros a serem enviados, respectivamente
6      http.begin(url);
7      int httpCode = http.GET();
8      http.end();
9      Serial.print(F("Feito")); // Confirmacao de envio das informacoes
10     flag5 = true; // Ativa-se a flag 5
11     delay(500);
12     digitalWrite(valvula, LOW); // Valvula e LED amarelo desligados
13     digitalWrite(ledAmar, LOW);
14 }
15 }
```

Código 4.19 – Trecho do código do trabalho de Freitag (2021, p. 54.)

A análise realizada pelo ChatGPT identificou preocupações como o armazenamento de credenciais de Wi-Fi em texto claro e a ausência de criptografia em comunicações HTTP. Adicionalmente, também comentou sobre a falta de tratamento de

erros adequado, sendo que esse problema se refere a casos como o demonstrado no Código 4.20, onde, por exemplo, ocorre a conexão do dispositivo à internet durante o *setup* do microcontrolador, porém, caso durante sua execução esse se desconecte do Wi-Fi, não existe nenhuma função que resolva o problema, ou seja, seriam casos imprevistos pelo autor que poderiam resultar no não funcionamento do dispositivo. Por fim, houve preocupações em relação ao uso extensivo de variáveis globais e potenciais problemas de lógica no código.

```
1 WiFi.begin(ssid, password);
2 Serial.print("Connecting to ");
3 Serial.print(ssid); Serial.println(" ...");
4 while (WiFi.status() != WL_CONNECTED) {
5 delay(1000);
6 Serial.print(".");
7 }
8 Serial.println('\n');
9 Serial.println("Connection established!");
```

Código 4.20 – Trecho do código do trabalho de Freitag (2021, p. 49.)

Já a análise de segurança do código realizada pelo Gemini, identificou como sendo os principais problemas de segurança o armazenamento inseguro de credenciais em texto claro, a falta de validação de entrada de dados, o uso de HTTP inseguro, a ausência de gerenciamento de erros robusto, a dependência de bibliotecas externas potencialmente vulneráveis e a falta de autenticação de usuários.

4.8 BENGALA IOT PARA MONITORAMENTO E SEGURANÇA DE IDOSOS

O trabalho de Antunes (2021) propõe o desenvolvimento de um dispositivo para ser acoplado à bengala de idosos, visando prevenir quedas. O dispositivo utiliza sensores para auxiliar fisioterapeutas na avaliação da correta utilização da bengala e facilitar intervenções nos movimentos dos pacientes. Além disso, o dispositivo alertará prestadores de socorro sobre a ocorrência de quedas, visando uma resposta rápida em casos de acidentes.

Na verificação de vulnerabilidades do código fornecido no trabalho, é identificada uma falha de segurança no Código 4.21, onde as credenciais de Wi-Fi são armazenadas sem qualquer forma de criptografia. Adicionalmente, também ocorre a falta de criptografia ao enviar mensagens nos Códigos 4.22 na linha 9 e 4.23 entre as linhas 6 e 15, onde no primeiro tem-se o protocolo HTTP sendo utilizado para enviar as coordenadas do dispositivo para uma planilha do Google Sheets via URL e no segundo o envio usando a biblioteca *SoftwareSerial*, na qual se simula a saída serial por meio dos pinos do ESP32.

```
1 const char* ssid = "nome";
```

```
2 const char* password = "senha";
```

Código 4.21 – Trecho do código do trabalho de Antunes (2021, p. 53.)

```
1 if (WiFi.status() == WL_CONNECTED) {
2     String URL = "https://script.google.com/macros/s/
    AKfycbxMTk9BwJhfN54EgkBA5ckZBcUOLVfjVP1mFZe1HiX6_h8p-0
    VJHyhjCFQgn4xZqqXHQ/exec?";
3     URL = URL + "coordenada=" + latitude + "," + longitude + "&
    velocidade=" +
4     velocidade + "&carga=" + carga + "&luminosidade=" + luminosidade;
5     Serial.print(URL);
6     HTTPClient http;
7     http.begin(URL);
8     http.addHeader("Content-Type", "application/x-www-form-urlencoded");
9     int answer_code = http.GET();
10    if (answer_code > 0) {
11        Serial.println("Codigo HTTP -> " + String(answer_code));
12    } else {
13        Serial.print("Error enviando POST, codigo: " + String(
    answer_code));
14    }
15    http.end();
```

Código 4.22 – Trecho do código do trabalho de Antunes (2021, p. 54.)

```
1 #include <SoftwareSerial.h>
2
3 SoftwareSerial mySerial(16, 17);
4
5 void EnviaSMSqueda() {
6     mySerial.println("AT+CMGF=1");
7     delay(100);
8     mySerial.println("AT+CMGS=\" \"");
9     delay(100);
10    mySerial.println("ALERTA DE QUEDA BengalaIoT");
11    delay(500);
12    mySerial.print(char(26));
13    delay(100);
14    mySerial.println("");
15    Serial.print("foi");
16    delay(1000);
17 }
```

Código 4.23 – Trecho do código do trabalho de Antunes (2021, p. 53.)

O ChatGPT encontrou algumas vulnerabilidades, sendo elas o armazenamento das credenciais de Wi-Fi e dados sensíveis, como coordenadas de GPS, sem criptografia em texto claro, a falta de validação das entradas de dados, o uso de HTTP

para enviar dados, expondo o sistema a interceptações e a falta de tratamento de erros robusto, que pode deixar o sistema em um estado inconsistente ou inseguro.

A análise do Gemini identificou algumas vulnerabilidades. O armazenamento de credenciais de Wi-Fi em texto não criptografado expõe a rede a acessos não autorizados. A falta de validação de entrada de dados dos sensores pode resultar em dados inválidos sendo enviados, dificultando a interpretação correta das informações. O uso do protocolo HTTP para transmissão de dados é inseguro, permitindo a interceptação por terceiros. A ausência de autenticação no envio de dados para a planilha também é uma vulnerabilidade. Além disso, existe a possibilidade de interceptação dos SMSs enviados e a falta de um gerenciamento robusto de erros.

4.9 SISTEMA IOT PARA MONITORAMENTO E SEGURANÇA DE IDOSOS EM AMBIENTE DOMÉSTICO

No TCC realizado por Schafaschek (2021), foi desenvolvido um dispositivo de detecção de quedas e movimento de baixo custo para pessoas idosas, capaz de alertar um responsável através de um aplicativo de mensagens e armazenar registros de movimento para monitoramento. O sistema também inclui sensores para monitorar a presença de gases (CO2, GLP e Gás Natural), proporcionando proteção contra intoxicação e incêndios. Todas as informações geradas pelo sistema podem ser acessadas via internet, sendo uma ferramenta para auxiliar no monitoramento e proteção de idosos.

Observando o código utilizado no projeto, nota-se que existe um problema no armazenamento de credenciais, já que no Código 4.24 variáveis de autentificação do Wi-Fi e do bot de Telegram estão sendo definidas de modo hardcoded e são armazenadas sem a utilização de criptografia, fazendo com que sejá possível que invasores tenham acesso aos dados do bot e possam mandar mensagens por ele, ou seja, existe o risco tanto de receber informações falsas ou até receber uma quantidade muito grande de mensagens causando um ataque DoS. Além disso, no Código 4.25 na linha 5 é feito o envio de mensagens para uma planilha online por meio de HTTP sem o uso de criptografia na mensagem e na linha 18 é recebido um dado por meio da porta Serial sem realizar uma validação da entrada, possibilitando que essa fosse alterada por um terceiro em um ataque de MITM sem que isso fosse percebido.

```
1 const char* ssid = "XXXXXXXX";
2 const char* password = "XXXXXXXX";
3 #define BOTtoken "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
4 #define CHAT_ID "XXXXXXXX"
5 }
```

Código 4.24 – Trecho do código do trabalho de Schafaschek (2021, p. 57.)

```
1  void send_data (String Data)
2  {
3      // Make a HTTP request:
4      String Request = String("GET ") + "/macros/s/" + GScriptId + "/exec?
       " + "value1=" + Sensor1_estado + "&value2=" + Sensor2_estado+ "&
       value3=" + Sensor3_estado + "&value4=" + Sensor4_estado + " HTTP/1.1\
       r\n" + "Host: script.google.com\r\n"+ "User-Agent: ESP8266\r\n" + "
       Connection: close\r\n" + "\r\n\r\n";
5      client.println(Request);
6      while (client.connected())
7      {
8          String line = client.readStringUntil('\n');
9          if (line == "\r")
10         {
11             Serial.println("headers received");
12             break;
13         }
14     }
15     while (client.available())
16     {
17     char c = client.read();
18     Serial.write(c);
19     }
20     client.stop();
21 }
```

Código 4.25 – Trecho do código do trabalho de Schafaschek (2021, p. 67.)

A análise realizada pelo ChatGPT identificou possíveis vulnerabilidades do código como sendo a exposição de credenciais de Wi-Fi em formato não criptografado, o uso de HTTP sem criptografia para envio de dados e falta de autenticação para acesso aos dados da planilha. Além disso, também foi citado o armazenamento inseguro de dados dos sensores utilizados, sendo um exemplo disso o Código 4.26, onde na linha 6, a variável *PPM* recebe o valor do sensor de gases MQ-135 após esse ser lido pela função *PPM_Funcao*, e esse valor é armazenado sem uso de criptografia, fazendo com que esse seja mais suscetível a ter seus dados descobertos por terceiros.

```
1  void Monitoramento_MQ135()
2  {
3   // Faz a leitura de CO2 a cada 5 segundos
4   if (contagem(5000, aux_millis_leitura) == true)
5   {
6       PPM = PPM_Funcao();
7       aux_millis_leitura = millis();
8
9       if (PPM > CO2_Limite and alarme_reconhecido == false)
10      {
11          String mensagem = "---------------PERIGO---------------";
```

```
12          mensagem += "\nAlta Concentracao de CO2: ";
13          mensagem += PPMstring + " PPM";
14          mensagem += "\nPara parar o alerta: /desligar_alarme";
15          bot.sendMessage(CHAT_ID,mensagem, "");
16      }
17  }
18  //Caso decorra o tempo apos a pessoa ter desligado o alarme manualmente
    , ele e reativado
19  if ( contagem(tempo_desligado_alarme*minuto, aux_millis_alarme) == true
    and
20 alarme_reconhecido == true)
21  {
22      alarme_reconhecido = false;
23      bot.sendMessage(CHAT_ID,"Alarme reativado", "");
24  }
25 }
```

Código 4.26 – Trecho do código do trabalho de Schafaschek (2021, p. 57.)

De acordo com o Gemini, as vulnerabilidades seriam, o armazenamento de credenciais em formato de texto simples, falta de validação de entrada para dados dos sensores, uso de HTTP não seguro para envio de dados e ausência de autenticação. Além disso, os códigos não mencionaram como os dados do sensor são armazenados no dispositivo e esses carecem de um gerenciamento de erros robusto.

4.10 APLICAÇÃO DA INTERNET DAS COISAS PARA MONITORAMENTO DE SINAIS BIOMÉDICOS

Este trabalho realizado por Lama (2019), tem como objetivo estudar a tecnologia IoT para o monitoramento de sinais correlacionados a estados emocionais de usuários de um dispositivo a ser construído. Ele inclui um estudo sobre a captação desses sinais por sensores, a leitura, armazenamento, interpretação e transmissão dos dados para a nuvem (Cloud), e como esses dados podem ser enviados para dispositivos móveis e interpretados por uma interface gráfica. O projeto compreende o desenvolvimento de um dispositivo vestível capaz de vincular desde a captação dos sinais até a transmissão para uma nuvem de arquivos.

Ao realizar a verificação do código disponibilizado no trabalho, percebe-se no Código 4.27 o armazenamento de credenciais de modo hardcoded, expondo informações sensíveis do Wi-Fi e da chave utilizada para passar mensagens para a nuvem pelo uso da *api.thingspeak.com*, sendo que adicionalmente o autor disponibilizou suas informações ao não alterá-las no código ao publicar o texto. Outra vulnerabilidade encontrada foi no envio das mensagens mostrado no Código 4.28 entre a linhas 13 e 21, no qual não existe o uso de criptografia, possibilitando que terceiros acessem a informação e o destino dos dados em trânsito.

```
String chaveAPI = "4IVZD1GFA7D3TX7C"; //Chave de Escrita do Canal
const char* ssid = "VIVO-69A1"; //ID do WiFi
const char* senha = "1553001110"; //Senha do WiFi
```

Código 4.27 – Trecho do código do trabalho de Lama (2019, p. 99.)

```
if (client.connect(server, 80)) //Envio para a nuvem
{
    String postStr = chaveAPI;
    postStr += "&field1=";
    postStr += String(BPM); //Envio do campo 1 (BPM)
    postStr += "&field2=";
    postStr += String(SPO2); //Envio do campo 2 (SPO2)
    postStr += "&field3=";
    postStr += String(BPMPack); //Envio do campo 3 (BPMPack)
    postStr += "&field4=";
    postStr += String(SPO2Pack); //Envio do campo 4 (SPO2Pack)
    postStr += "\r\n\r\n";
    client.print("POST /update HTTP/1.1\n");
    client.print("Host: api.thingspeak.com\n");
    client.print("Connection: close\n");
    client.print("X-THINGSPEAKAPIKEY: " + chaveAPI + "\n");
    client.print("Content-Type:application/x-www-form-urlencoded\n");
    client.print("Content-Length: ");
    client.print(postStr.length());
    client.print("\n\n");
    client.print(postStr);
}
```

Código 4.28 – Trecho do código do trabalho de Lama (2019, p. 103.)

De acordo com o ChatGPT, o código apresenta vulnerabilidades, sendo a primeira dessas, o fato de as credenciais Wi-Fi e a chave para o serviço em nuvem estarem diretamente codificadas no código-fonte. Além disso, a comunicação com o serviço em nuvem é realizada de forma não segura, utilizando HTTP sem qualquer tipo de criptografia. Outra vulnerabilidade é a falta de tratamento adequado de erros, pois o código não lida com situações de falha na conexão Wi-Fi ou na transmissão de dados para o serviço em nuvem. Adicionalmente, não há validação de entrada dos dados do sensor, o que pode levar a resultados incorretos ou até mesmo falhas no programa. Por fim, o código aloca e desaloca memória de forma ineficiente, o que pode resultar em fragmentação de memória e consumo excessivo de recursos.

Conforme o analisado pelo Gemini, foram identificadas vulnerabilidades que podem comprometer a segurança do dispositivo e dos dados coletados, essas sendo o armazenamento direto de credenciais Wi-Fi e a chave de acesso a nuvem no código-fonte, a comunicação insegura sem o uso do protocolo HTTPS, tratamento inadequado de erros, falta de validação de entrada e gerenciamento ineficiente de memória.

4.11 RESUMO DAS AVALIAÇÕES DE SEGURANÇA

Nesta seção, serão discutidas as principais vulnerabilidades encontradas na análise dos trabalhos de conclusão de curso. Isto é feito através de duas tabelas, a Tabela 1, que quantifica as vulnerabilidades detectadas por cada uma das três abordagens utilizadas para verificar a segurança dos códigos, e a Tabela 2, que resume a presença das vulnerabilidades em cada um dos trabalhos.

Tabela 1 – Tabela de comparação das vulnerabilidades encontradas por cada método

Vulnerabilidades	Checklist	ChatGPT	Gemini
Buffer overflow	1	1	0
DoS	2	0	1
Problemas de autentificação	9	9	8
Criptografia falha	7	6	3
Abertura de portas desnecessárias	0	0	0
Vulnerabilidades em códigos de terceiros	1	0	0
Vulnerabilidades não consideradas no checklist	-	9	9

Fonte: Produzida pelo autor

A Tabela 4.11 mostra que as vulnerabilidades de autenticação são as mais comuns, com o checklist e o ChatGPT identificando problemas em 9 códigos cada, e o Gemini encontrando vulnerabilidades em 8 códigos. Isso mostra que problemas de autenticação são uma área que necessita de atenção especial durante o desenvolvimento de dispositivos IoT, uma vez que, é comum desprezar a programação segura nesse aspecto. Além disso, falhas de criptografia ao enviar mensagens também são frequentes, com 7 códigos afetados identificados pelo checklist, 6 pelo ChatGPT e 3 pelo Gemini.

Outras vulnerabilidades, como buffer overflow e DoS, foram identificadas menos frequentemente. O checklist e o ChatGPT detectaram buffer overflow em 1 código cada, enquanto o Gemini não encontrou nenhum. Para DoS, o checklist encontrou 2 códigos vulneráveis, o Gemini encontrou 1, enquanto o ChatGPT não identificou nenhum. A abertura de portas desnecessárias não foi detectada por nenhuma das abordagens, indicando que pode ser uma vulnerabilidade menos comum ou mais difícil de identificar com as ferramentas utilizadas. Além disso, a tabela mostra que o ChatGPT e o Gemini identificaram nos 9 trabalhos, vulnerabilidades que não foram consideradas no checklist, sugerindo que essas ferramentas de IA podem detectar problemas adicionais que não foram cobertos na realização do trabalho, sendo que a mais comum entre essas foi a falta de verificação de entradas, o que levou a inclusão desse item ao checklist. Adicionalmente, observa-se que nem todas as vulnerabilidades apontadas pelas ferramentas se encontram nos códigos, logo, sugere-se que ao utilizá-las para verificar a segurança em códigos, haja cuidado com isso.

Essas variações na detecção de vulnerabilidades sugerem que uma abordagem híbrida pode ser a mais eficaz para garantir a segurança dos dispositivos IoT. A utilização combinada dessas ferramentas pode proporcionar uma cobertura mais completa, mitigando as limitações de cada método individual.

Como parte da análise das vulnerabilidades de software em dispositivos IoT, foi desenvolvida a Tabela 4.11, que documenta quais trabalhos de conclusão de curso apresentavam determinadas vulnerabilidades e, consequentemente, quais estavam em conformidade com as práticas de codificação segura recomendadas pelo checklist. Isso foi feito com o intuito de demonstrar que mesmo os trabalhos possuindo algumas vulnerabilidades, eles também tinham conformidades de segurança em certos aspectos.

Tabela 2 – Tabela de análise dos códigos de acordo com o checklist

Ref. Trabalho	Bibliotecas	Credenciais	Criptografia	DoS	Portas	Buffers
Greselle (2022)	Não Ok	Não Ok	Não Ok	Não Ok	Ok	Não Ok
Arruda (2019)	Ok	Não Ok	Ok	Ok	Ok	Ok
Paula (2019)	Ok	Não Ok	Ok	Ok	Ok	Ok
Chilanti (2022)	Ok	Não Ok	Não Ok	Ok	Ok	Ok
Pontes (2022)	Ok	Não Ok	Não Ok	Ok	Ok	Ok
Freitag (2021)	Ok	Não Ok	Não Ok	Ok	Ok	Ok
Antunes (2021)	Ok	Não Ok	Não Ok	Ok	Ok	Ok
Schafaschek (2021)	Ok	Não Ok	Não Ok	Não Ok	Ok	Ok
Lama (2019)	Ok	Não Ok	Não Ok	Ok	Ok	Ok

Fonte: Produzida pelo autor

Assim, a Tabela 4.11 evidencia que o trabalho de Greselle (2022) apresentou vulnerabilidades em várias categorias, incluindo bibliotecas inseguras, armazenamento inadequado de credenciais, criptografia falha, e DoS, estando em conformidade apenas na categoria de portas. Em contraste, os trabalhos de Arruda (2019) e Paula (2019) demonstraram uma abordagem mais segura, ou uma situação mais simples, apresentando conformidade em várias áreas, com exceção do armazenamento de credenciais.

Os trabalhos de Chilanti (2022), Pontes (2022), Freitag (2021) e Antunes (2021) exibiram padrões semelhantes, com vulnerabilidades apenas em problemas de autentificação e criptografia de mensagens. Dito isso, evidencia-se que esses trabalhos possuiam as vulnerabilidades mais comumente encontradas pelo checklist nos códigos analisados, logo necessitando de mais atenção nessas áreas.

O trabalho de Schafaschek (2021) apresentou uma vulnerabilidade adicional em DoS, além das categorias de credenciais e criptografia, sendo seguro apenas nas bibliotecas, portas e buffers. Já Lama (2019) também apresentou vulnerabilidades em várias categorias, exceto em bibliotecas, DoS, portas e buffers.

Esta análise comparativa é importante para entender o equilíbrio entre vulnerabilidades e conformidades nos projetos de IoT. Ela destaca que muitas vezes nesses

trabalhos, não se pensa muito sobre a segurança do dispositivo em questão, o que torna vulnerabilidades como a disponibilidade de credenciais em texto claro e comunicações sem criptografia comuns, uma vez que muitos trabalhos acabam envolvendo essas áreas sem o devido cuidado.

Para avaliar a precisão das análises de segurança realizadas pelas ferramentas de IA, foi feito um teste com o ChatGPT, no qual foi desenvolvida a matriz de confusão demonstrada na Figura 6. Esta matriz mostra quantas vulnerabilidades o ChatGPT encontrou nos nove trabalhos analisados e se seus resultados foram os mesmos de uma verificação manual, de modo que, as vulnerabilidades consideradas não encontradas pela análise manual seriam as abordadas no checklist. A matriz de confusão foi utilizada com o intuito de facilitar a verificação da quantidade de erros cometidos pela IA durante a detecção de vulnerabilidades, permitindo uma comparação entre os resultados previstos pelo ChatGPT e a análise manual realizada.

Figura 6 – Matriz de confusão para verificar resultados do ChatGPT

		Análise manual	
		Encontrou vulnerabilidade	Não encontrou vulnerabilidade
Análise ChatGPT	Encontrou vulnerabilidade	Verdadeiro Positivo 28	Falso Positivo 3
	Não encontrou vulnerabilidade	Falso Negativo 7	Verdadeiro Negativo 33

Fonte: Autor (2024)

Os resultados da matriz de confusão mostram que, embora o ChatGPT seja capaz de identificar corretamente possíveis presenças e ausências de vulnerabilidades nos códigos, ele também gerou três falsos positivos e sete falsos negativos. Isso indica que, mesmo sendo possível utilizar o ChatGPT como uma ferramenta para a identificação de problemas de segurança, a combinação com outras abordagens e a validação humana se mostram importantes para aumentar a confiabilidade de suas análises de segurança em projetos de IoT.

5 CONCLUSÃO

Neste trabalho, foi explorado o impacto da IoT na interação com o mundo físico e destacou-se os desafios de segurança que acompanham a proliferação desses dispositivos. Não só isso, foi identificado que muitos desses problemas de segurança derivam de práticas inadequadas de programação, tornando os dispositivos vulneráveis a ataques que podem comprometer a integridade e funcionalidade dos sistemas IoT.

Para mitigar esses riscos, foi desenvolvido um guia prático de codificação segura para dispositivos IoT, fundamentado em uma pesquisa sobre as principais vulnerabilidades de software associadas a esses dispositivos. O guia foi projetado para ajudar os desenvolvedores a adotar práticas de programação seguras, reduzindo a probabilidade de falhas e melhorando a robustez dos dispositivos IoT.

Assim, foi feita a aplicação prática do guia em 10 Trabalhos de Conclusão de Curso (TCC) que envolvem o desenvolvimento de software IoT. Esta aplicação, permitiu a identificação de vulnerabilidades comuns e demonstrou a eficácia das diretrizes propostas. A comparação dos resultados obtidos com diagnósticos de segurança fornecidos por ferramentas de IA, como ChatGPT e Gemini, reforçou a validade e a utilidade do guia, porém, também demonstrou pontos nos quais ele poderia melhorar, uma vez que nos trabalhos analisados foram citadas vulnerabilidades não abordadas no trabalho.

Os resultados mostraram que os trabalhos analisados não possuem medidas adequadas de segurança, revelando vulnerabilidades significativas em suas implementações. As ferramentas utilizadas para checar os códigos, como o checklist, ChatGPT e Gemini, identificaram diferentes problemas, indicando que essas abordagens podem se complementar para uma análise de segurança mais robusta, sendo importante complementar que ao utilizar as ferramentas de IA na verificação de segurança de códigos, existe a possibilidade de elas apontarem vulnerabilidades inexistentes, retirando a importância de checar manualmente.

Dito isso, vulnerabilidades como buffer overflow, falhas de autenticação e criptografia inadequada foram frequentemente detectadas pelas IAs e podem ser mitigadas através da aplicação das recomendações do guia desenvolvido. A conscientização dos desenvolvedores sobre a importância da segurança na programação é crucial para o desenvolvimento de soluções IoT seguras. Assim, o guia oferece uma referência que promove a implementação de boas práticas de programação, essenciais para enfrentar os desafios de segurança no cenário IoT.

Para trabalhos futuros, seria relevante o desenvolvimento de uma ferramenta de IA que utilizasse o checklist proposto neste estudo para verificar automaticamente

a segurança dos códigos de dispositivos IoT. Essa ferramenta poderia ser projetada para realizar análises, identificando e reportando vulnerabilidades definidas no guia de codificação segura. Além disso, seria benéfico realizar testes para validar a eficácia da ferramenta em diferentes ambientes de desenvolvimento e aplicação de IoT.

REFERÊNCIAS

AHLMEYER, M.; CHIRCU, A. M. Securing the internet of things: a review. *In*: **Proceedings** of the **2016** ISSUES IN INFORMATION SYSTEMS, v. 17, n. 4, p. 21–28, 2016. Disponível em: https://www.semanticscholar.org/paper/SECURING-THE-INTERNET-OF-THINGS%3A-A-REVIEW-Ahlmeyer-Chircu/12bff696d912535ab235c4e67a1e72c39cef2fe2. Acesso em: 20 nov. 2023.

ALHIRABI, N.; RANA, O. F.; PERERA, C. Designing security and privacy requirements in internet of things: A survey. **ACM Transactions on Internet of Things**, v. 2, n. 1, p. 1–37, 2021.

ANTUNES, L. F. M. **Bengala IoT para monitoramento e segurança de idosos**. 2021. Trabalho de Conclusão de Curso (Graduação em Engenharia Elétrica) — Universidade do Estado de Santa Catarina, Joinville, 2021.

ARRUDA, C. V. **Análise da implantação de uma rede LoRaWAN para sistemas embarcados**. 2019. Trabalho de Conclusão de Curso (Graduação em Engenharia Mecatrônica) — Universidade Federal de Santa Catarina, Câmpus Joinville, Joinville, 2019.

ATAQUE man-in-the-middle (MitM). **Claranet**. 2024. https://www.claranet.com/br/blog/man-in-the-middle-o-que-e. Acesso em: 19 jun. 2024.

ATZORI, L.; IERA, A.; MORABITO, G. The internet of things: a survey. **Computer Networks**, v. 54, n. 15, p. 2787–2805, 2010.

BERLANDA, R. G. **Guia de segurança da informação para a conectividade de dispositivos IoT**. 2021. Trabalho de Conclusão de Curso (Graduação em Gestão da Tecnologia da Informação) — Instituto Federal de Santa Catarina, Câmpus Florianópolis, Florianópolis, 2021.

CADARIU, M. *et al.* Tracking known security vulnerabilities in proprietary software systems. 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), p. 516–519, 2015.

CAMPOS, A. **Sistemas de Segurança da Informação**: controlando os riscos. 2 ed. Florianópolis: Visual Books, 2007.

CEKEREVAC, Z.; DVORAK, Z.; PECNIK, T. Top seven iot operating systems in mid-2020. *In*: **Proceedings** of the MEST JOURNAL, v. 8, p. 47–68, 07 2020. Disponível em: https://www.researchgate.net/publication/342958452_TOP_SEVEN_IoT_OPERATING_SYSTEMSIN_MID-2020. Acesso em: 20 nov. 2023.

CHILANTI, D. P. **GEN.IO: Desenvolvimento e aplicação de dispositivo IoT para coleta de dados de produção em ambiente industrial**. 2022. Trabalho de Conclusão de Curso (Graduação em Engenharia de Controle e Automação) — Universidade Federal de Santa Catarina, Câmpus Florianópolis, Florianópolis, 2022.

CONTI, M.; DRAGONI, N.; LESYK, V. A survey of man in the middle attacks. *In*: **Proceedings** of the IEEE COMMUNICATIONS SURVEYS TUTORIALS, v. 18, n. 3, p. 2027–2051, 2016. Disponível em: https://ieeexplore.ieee.org/abstract/document/7442758?casa_token=ZE_aDcaQpv8AAAAA:GTmTtGTRzz327Hrz8ERuHAZLWtxP-hfAcDRwhDrgzIo_Mp7apFBO-EzYrgiJpHa4MYfGO5gfbRby. Acesso em: 20 nov. 2023.

COWAN, C. *et al.* Buffer overflows: attacks and defenses for the vulnerability of the decade. *In*: **Proceedings** of the **2003** FOUNDATIONS OF INTRUSION TOLERANT SYSTEMS, p. 227–237, 2003. Disponível em: https://ieeexplore.ieee.org/document/1264935. Acesso em: 20 nov. 2023.

DUDHE, P. *et al.* Internet of things (iot): an overview and its applications. *In*: **Proceedings** of the **2017** INTERNATIONAL CONFERENCE ON ENERGY, COMMUNICATION, DATA ANALYTICS AND SOFT COMPUTING (ICECDS), p. 2650–2653, 2017. Disponível em: https://ieeexplore.ieee.org/abstract/document/8389935. Acesso em: 20 nov. 2023.

ENISA. **Guidelines for Securing the Internet of Things**. 2020. https://www.enisa.europa.eu/publications/guidelines-for-securing-the-internet-of-things. Acesso em: 8 jul. 2024.

FREITAG, N. **Bebedouro IoT para monitoramento e controle da hidratação em idosos institucionalizados**. 2021. Trabalho de Conclusão de Curso (Graduação em Engenharia Elétrica) — Universidade do Estado de Santa Catarina, Joinville, 2021.

FUKUDA, L. M. **Segurança da informação em IOT**. 2019. Trabalho de Conclusão de Curso (Especialização em Gestão da Tecnologia da Informação e Comunicação) — Universidade Tecnológica Federal do Paraná, Curitiba, 2019.

GARCIA, M. A. **Desenvolvimento de Firmware para Testes de Produção de Produtos IoT**. 2022. Trabalho de Conclusão de Curso (Graduação em Engenharia de Controle e Automação) — Universidade Federal de Santa Catarina, Câmpus Florianópolis, Florianópolis, 2022.

GRESELLE, A. G. **Monitoramento do consumo de energia elétrica aplicado a cargas residenciais**. 2022. Trabalho de Conclusão de Curso (Graduação em Engenharia Mecatrônica) — Universidade Federal de Santa Catarina, Câmpus Joinville, Joinville, 2022.

GSMA. **IoT Security Guidelines**. 2020. https://www.gsma.com/iot/iot-security-guidelines/. Acesso em: 8 jul. 2024.

ROCHA JUNIOR, S. L. da. **SEGURANÇA EM SISTEMAS IoT**: vulnerabilidades e mecanismos de prevenção. 2022. Trabalho de Conclusão de Curso (Graduação em Engenharia Eletrônica) — Centro Tecnológico da Universidade Federal de Santa Catarina, Florianópolis, 2022.

KL, A. **CiteDrive brings reference management to Overleaf**. 2023. Https://thesecmaster.com/blog/what-is-a-buffer-overflow-attack-and-how-to-prevent-it. Acesso em: 11 de jun. de 2024.

LAMA, D. Q. del. **Aplicação da Internet das Coisas para monitoramento de sinais biomédicos**. 2019. Trabalho de Conclusão de Curso (Graduação em Engenharia Elétrica) — Universidade do Estado de Santa Catarina, Joinville, 2019.

LIM, J.-D.; SOHN, S.; KIM, J. Proposal of smart segmentation framework for preventing threats from spreading in iot. *In*: **Proceedings** of the **2020** INTERNATIONAL CONFERENCE ON INFORMATION AND COMMUNICATION TECHNOLOGY CONVERGENCE (ICTC), p. 1745–1747, 2020. Disponível em: https://ieeexplore.ieee.org/document/9289632. Acesso em: 20 nov. 2023.

NESHENKO, N. *et al.* Demystifying iot security: an exhaustive survey on iot vulnerabilities and a first empirical look on internet-scale iot exploitations. *In*: **Proceedings** of the IEEE COMMUNICATIONS SURVEYS TUTORIALS, v. 21, n. 3, p. 2702–2733, 2019. Disponível em: https://ieeexplore.ieee.org/document/8688434. Acesso em: 20 nov. 2023.

ORACEVIC, A.; DILEK, S.; OZDEMIR, S. Security in internet of things: A survey. *In*: **Proceedings** of the **2017** INTERNATIONAL SYMPOSIUM ON NETWORKS, COMPUTERS AND COMMUNICATIONS (ISNCC), p. 1–6, 2017. Disponível em: https://ieeexplore.ieee.org/document/8072001. Acesso em: 12 out. 2023.

OWASP. **OWASP IoT Security Testing Guide**. 2024. https://owasp.org/www-project-internet-of-things/. Acesso em: 8 jul. 2024.

PAULA, G. H. O. de. **Avaliação e projeto de localização indoor utilizando bluetooth**. 2019. Trabalho de Conclusão de Curso (Graduação em Engenharia Mecatrônica) — Universidade Federal de Santa Catarina, Câmpus Joinville, Joinville, 2019.

PONTES, K. R. **Sistema IoT para monitoramento do consumo de água e controle da hidratação corporal**. 2022. Trabalho de Conclusão de Curso (Graduação em Engenharia Elétrica) — Universidade do Estado de Santa Catarina, Joinville, 2022.

POPA, M. Analysis of zero-day vulnerabilities in java. Journal of Mobile, Embedded and Distributed Systems, v. 5, p. 108–117, 2013.

SILVA SCHAFASCHEK, D. da. **Sistema IoT para monitoramento e segurança de idosos em ambiente doméstico**. 2021. Trabalho de Conclusão de Curso (Graduação em Engenharia Elétrica) — Universidade do Estado de Santa Catarina, Joinville, 2021.

SCRIPCARIU, L. *et al.* Aes vulnerabilities study. 2018 10th International Conference on Electronics, Computers and Artificial Intelligence (ECAI), p. 1–4, 2018.

SNYK Vulnerability Database. **Snyk**. 2024. https://snyk.io. Acesso em: 18 jun. 2024.

WEBER, M.; BOBAN, M. Security challenges of the internet of things. *In*: **Proceedings** of the **2016** 39th INTERNATIONAL CONVENTION ON INFORMATION AND COMMUNICATION TECHNOLOGY, ELECTRONICS AND MICROELECTRONICS (MIPRO), p. 638–643, 2016. Disponível em: https://ieeexplore.ieee.org/document/7522219. Acesso em: 20 nov. 2023.