NETWORK SCANNING TOOL

A Project Report

Submitted By

Krenil Raj
Shubh Patel
Prarthan Christian
Kartikay Mistry

210303126010, 210303126042, 210303126058, 210303126060

in Partial Fulfilment For the Award of

the Degree of

BACHELOR OF TECHNOLOGY

COMPUTER SCIENCE & ENGINEERING

Under the Guidance of

Dr. Mohammad Shahnawaz Shaikh

Associate Professor

VADODARA

October - 2024

PARUL UNIVERSITY

CERTIFICATE

This is to Certify that Project - 2 (203105400) of 7^{th} Semester entitled "Network Scanning Tool" of Group No. PUCSE_377 has been successfully completed by

- Krenil Raj-210303126010
- Shubh Patel-210303126042
- Prarthan Christian-210303126058
- Kartikay Mistry-210303126060

under my guidance in partial fulfillment of the Bachelor of Technology (B.Tech) in Computer Science & Engineering of Parul University in Academic Year 2024- 2025.

Date of Submission :_____

Dr. Mohammad Shahnawaz Shaikh,

Project Guide

Project Coordinator:-

Dr. Amit Barve,

Head of Department,

CSE, PIET,

Parul University.

Acknowledgements

"The single greatest cause of happiness is gratitude."

-Auliq-Ice

Behind every significant achievement lies the support of those who help individuals overcome challenges to reach their goals. I am immensely grateful to my esteemed guide, Dr. Mohammad Shahnawaz Shaikh, for his unwavering support, exceptional cooperation, and insightful guidance. It has been a privilege to receive mentorship from him. His encouragement transforms complexities into manageable tasks. Throughout the entire project, I benefited from his invaluable advice and timely suggestions. I will always be indebted to him and take pride in having worked under his guidance. I would also like to express my heartfelt appreciation to Dr. Amit Barve, Head of the Computer Science Engineering Department. I feel truly fortunate to have received their invaluable advice, guidance, and leadership.

Krenil Raj , Shubh Patel , Prarthan Christian , Kartikay Mistry

CSE, PIET

Parul University,

Vadodara

Abstract

In today's cybersecurity landscape, where threats are both persistent and evolving, proactive defense strategies are essential. Network scanning tools have become crucial elements of cybersecurity frameworks, allowing organizations to identify vulnerabilities, monitor network activity, and proactively mitigate potential cyber threats. This document aims to provide a comprehensive overview of network scanning tools, focusing on their operational functionalities, various classifications, and their significant role in strengthening organizational security.

This documentation explores the fundamental principles of network scanning, highlighting its importance in examining network infrastructure, identifying connected devices, and evaluating their configurations. It covers a range of scanning techniques, from basic port scanning to advanced vulnerability assessments. Additionally, it explains how these tools contribute to compliance audits, enhance risk management processes, and improve incident response preparedness.

Moreover, the documentation emphasizes the key features of advanced network scanning tools, including automation capabilities, detailed reporting functions, and seamless integration with threat intelligence platforms. These attributes empower cybersecurity professionals to perform thorough assessments, prioritize remediation efforts, and proactively defend against emerging cyber threats.

In addition to the technical aspects, this documentation addresses the practical implications of using network scanning tools in organizational contexts. It discusses deployment considerations, integration challenges, and scalability options needed to meet evolving security demands. Furthermore, it underscores the necessity of aligning network scanning efforts with broader cybersecurity strategies and organizational goals.

In conclusion, this project documentation highlights the critical role of network scanning tools in reducing cybersecurity risks and enhancing organizational resilience. By utilizing advanced scanning technologies, organizations can fortify their defenses, protect vital assets, and promote a culture of proactive cybersecurity awareness in the face of ongoing threats. Network scanning tools are indispensable in modern cybersecurity, enabling organizations to implement preemptive security measures. Effective deployment strategies, resource allocation, and scalability considerations are vital for optimizing the success of network scanning initiatives.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Networking

A computer network functions as an integrated system that connects multiple independent computers to enable the exchange of information and resources. This connection enhances communication efficiency among users.

Consisting of two or more computer systems, a network can be set up using either wired or wireless mediums, supported by various hardware and software components. These elements facilitate smooth connectivity and collaboration throughout the network.

In a computer network, various nodes serve different purposes. These nodes include servers, networking devices, personal computers, and specialized or general-purpose hosts. Each node is assigned a unique identification through host names and network addresses.

1.2 Networking Services

Networking services comprise a wide array of protocols and functionalities essential for the effective operation of computer networks. These services enable communication, resource sharing, and network management among various devices connected within a network infrastructure.

Examples of these services include protocols for file transfer, email communication, remote access, web services, domain name resolution, directory services, time synchronization, and network security management.

These services form the foundation of contemporary networking architectures, allowing organizations and individuals to establish secure communication channels, manage network resources efficiently, and maintain seamless connectivity across both local and global networks.

1.3 Ports And Port Scanning

In computer networking, ports act as communication endpoints within a network or between different networks. They are numerical identifiers linked to specific services or processes that run on a device, enabling multiple applications or services to function simultaneously on a single device without interference.

Port scanning involves systematically probing a target device or network to determine which ports are open, closed, or filtered. This technique is essential for network reconnaissance, security auditing, and vulnerability assessment. Port scanning aids in identifying potential entry points or services exposed to the network, assisting in the detection of security weaknesses or misconfigurations.

1.4 Network Scanning

Network scanning is an essential procedure in computer networking that involves systematically probing and analyzing network infrastructure to collect information about connected devices, services, and potential vulnerabilities. This proactive approach is employed by network administrators, security professionals, and penetration testers to evaluate the overall security posture.

Typically, network scanning encompasses checking for open ports, discovering active hosts, mapping the network topology, identifying services running on devices, and detecting potentially vulnerable or outdated services. This gathered information is vital for maintaining network health, diagnosing problems, planning for network expansions, and enhancing security measures.

Several types of network scanning techniques exist, including port scanning, host discovery, vulnerability scanning, and network mapping, each fulfilling specific roles in network management and security. Automated scanning tools facilitate the scanning process, enabling administrators to efficiently monitor and manage network infrastructure.

1.5 Problem Statement

In today's landscape of network administration, the rapid growth and complexity of network infrastructures present significant challenges for network administrators and IT security professionals. The increasing variety of devices, operating systems, and evolving security threats have added to this complexity, rendering traditional methods of network monitoring and security assessment insufficient.

Manual processes for tasks such as device discovery, port scanning, and vulnerability assessment are not only time-consuming but also susceptible to errors, leaving networks exposed to unnoticed threats. Moreover, current solutions frequently lack the ability to integrate seamlessly with other security systems, which hampers effective information sharing and coordinated incident response.

To address these challenges, there is a pressing need for an advanced Network Scanning Tool that can thoroughly tackle the complexities and vulnerabilities inherent in modern network infrastructures. Such a tool would provide network administrators with a centralized platform to manage and secure their networks efficiently.

By automating scanning tasks, performing detailed analyses of network assets, and enabling prompt remediation of vulnerabilities, the Network Scanning Tool aims to significantly improve network visibility, strengthen security posture, and streamline operational workflows. With its advanced features designed specifically for the needs of network administrators and IT security professionals, the tool seeks to alleviate the difficulties arising from the complexity of contemporary network environments.

In conclusion, this problem statement highlights the necessity for a comprehensive and efficient solution to navigate the challenges and vulnerabilities of modern network infrastructures. The development of the Network Scanning Tool represents a dedicated effort to address these issues and empower network administrators and IT security professionals to protect their networks effectively.

1.6 Scope

The scope of the Network Scanning Tool includes a broad array of features designed to assist network administrators and IT security professionals in effectively managing and securing network infrastructures. Key components within the project's scope include:

1. Network Discovery

- The tool will utilize various protocols, including ICMP, ARP, and SNMP, to identify devices within the network.
- Discovery methods will encompass IP range scanning, DNS resolution, and MAC address lookup to pinpoint all active network assets.
- It will support both IPv4 and IPv6 addressing schemes, ensuring compatibility across various network environments.

2. Port Scanning

- The tool will implement comprehensive port scanning techniques, such as TCP connect scans, SYN scans, UDP scans, and service version detection.
- It will identify open ports, the services operating on those ports, and associated vulnerabilities through a combination of active and passive scanning methods.
- Detection of common protocols and applications on open ports will help in evaluating potential security risks and attack vectors.

3. Vulnerability Assessment

- The tool will perform vulnerability scans utilizing a vast database of known vulnerabilities, CVE (Common Vulnerabilities and Exposures) lookups, and signature-based detection methods.
- It will assess the severity and impact of identified vulnerabilities, categorizing them based on risk levels and providing actionable recommendations for remediation.
- Vulnerability assessments will cover network devices, operating systems, applications, and services running within the network.

4. Reporting and Analysis

- The tool will generate detailed reports summarizing scan results, discovered vulnerabilities, and recommended remediation actions.
- Reports will be customizable, allowing users to filter and prioritize findings based on specific needs and organizational policies.
- Graphical representations, trend analysis, and historical comparisons will offer insights into the network's security posture over time.

5. Logging and Auditing

- The tool will maintain comprehensive logs of scan activities, including start and end times, scanned IP addresses, scan configurations, and detected vulnerabilities.
- Logging capabilities will support auditing, forensic analysis, and compliance with regulatory requirements such as GDPR and HIPAA.

- Logs will be encrypted and tamper-evident, ensuring the integrity and confidentiality of stored information.

6. Integration Capabilities

- APIs (Application Programming Interfaces) will be available to facilitate seamless integration with third-party security systems, including SIEM platforms, ticketing systems, and vulnerability management solutions.
- Integration with external systems will enable automated incident response, threat intelligence sharing, and workflow orchestration.

7. User Interface and Experience

- A user-friendly graphical interface will be created, featuring intuitive navigation, drag-and-drop functionality, and contextual tooltips to assist users in configuring scan parameters and interpreting results.
- Customization options will allow users to tailor the interface layout, color schemes, and dashboard widgets to meet their preferences.

8. Performance Optimization

- The tool will focus on optimizing scanning algorithms, reducing network overhead, and employing multithreading techniques to enhance scanning speed and efficiency.
- Load balancing mechanisms will be integrated to distribute scanning tasks across multiple nodes, ensuring scalability and resilience in large network environments.

9. Security Measures

- The tool will enforce robust authentication mechanisms, including password policies, multi-factor authentication, and integration with LDAP (Lightweight Directory Access Protocol) servers.
- Data encryption will be utilized to protect sensitive information transmitted over the network, including scan results, user credentials, and configuration settings.
- Role-based access control will restrict access to sensitive features and data based on user roles and permissions, adhering to least privilege principles.

10. Scalability and Flexibility

- The tool will be designed for both horizontal and vertical scalability, accommodating networks of various sizes and complexities, from small office setups to large enterprise environments.
- Flexible deployment options, including on-premises installations, cloud-based solutions, and hybrid deployments, will address diverse organizational needs and compliance requirements.

The comprehensive scope of the Network Scanning Tool aims to equip network administrators and IT security professionals with a robust solution for managing and securing modern network infrastructures effectively.

1.7 Aim And Objective

The goal of the Network Scanning Tool project is to create a comprehensive and efficient software solution that enables network administrators and IT security professionals to manage and secure network infrastructures effectively. By offering advanced scanning, analysis, and reporting capabilities, the project aims to enhance network visibility, improve security posture, and streamline operational workflows across various network environments.

Objectives:

1. Enhanced Network Visibility

- Develop tools for thorough device discovery, port scanning, and vulnerability assessments to deliver detailed insights into the network's topology and configuration.
- Facilitate real-time monitoring and tracking of network assets, including devices, services, and potential security threats.

2. Improved Security Posture

- Identify and prioritize vulnerabilities, misconfigurations, and possible attack vectors within the network infrastructure.
- Provide actionable recommendations and strategies to address identified security risks, thereby enhancing overall security resilience.

3. Efficient Operational Workflows

- Automate scanning tasks and streamline workflows to minimize manual effort and resource expenditure.

- Integrate with existing security systems and tools to promote seamless information exchange, incident response, and workflow management.

4. Customizable Reporting and Analysis

- Generate comprehensive reports and summaries to offer stakeholders actionable insights into the network's security posture and compliance status.
- Allow customization of reporting templates, formats, and delivery options to meet organizational needs and regulatory requirements.

5. User-Friendly Interface and Experience

- Design an intuitive interface with features like drag-and-drop functionality, contextual tooltips, and customizable dashboards to improve user experience and productivity.
- Provide thorough documentation, tutorials, and training materials to support users with varying levels of technical expertise.

6. Scalability and Flexibility

- Ensure that the software solution is scalable and flexible enough to accommodate networks of different sizes, complexities, and deployment architectures.
- Support both on-premises and cloud-based deployments, along with integration capabilities for third-party systems and APIs to address evolving organizational needs and compliance standards.

7. Security and Compliance

- Implement robust security measures, such as authentication mechanisms, data encryption, and access controls, to protect sensitive information and ensure adherence to regulatory standards.
- Conduct regular security assessments and audits to identify and mitigate potential vulnerabilities, ensuring the integrity and confidentiality of data processed by the tool.

By accomplishing these objectives, the Network Scanning Tool aims to empower organizations to proactively manage and secure their network infrastructures, reduce security risks, and maintain a strong security posture amid evolving cyber threats and regulatory challenges.

Chapter 2

Literature Survey

2.1 PAPER 1: A Remote Active OS Fingerprinting Tool Using ICMP

In conclusion, this research paper has detailed the creation and implementation of a remote active OS fingerprinting tool that utilizes Internet Control Message Protocol (ICMP). OS fingerprinting is an essential aspect of network reconnaissance, allowing for the identification of the operating systems of remote hosts without requiring direct access to their systems.

Through a thorough examination of ICMP-based fingerprinting techniques and methodologies, this study has shown the effectiveness and efficiency of the proposed tool in accurately identifying target operating systems across various network environments. By using ICMP packets for active probing and analyzing the responses, the tool provides a non-intrusive yet effective method for OS detection.

The development of this remote active OS fingerprinting tool makes a notable contribution to network security and management practices. Unlike passive fingerprinting methods that rely on monitoring network traffic, the active nature of this tool facilitates direct interaction with target hosts, leading to more reliable and precise identification of operating systems.

Empirical evaluations have validated the tool's effectiveness in detecting a broad spectrum of operating systems while minimizing detection footprints and false positives. By optimizing techniques for packet crafting, transmission, and response analysis, the tool achieves high accuracy rates while keeping network overhead and resource consumption low.

Looking ahead, future research may focus on enhancing the tool's capabilities by incorporating additional probing techniques and refining response analysis algorithms to further improve accuracy and robustness. Additionally, investigating the tool's effectiveness in dynamic and heterogeneous network environments, including cloud-based infrastructures and IoT ecosystems, could offer

valuable insights into its versatility.

In summary, the development of a remote active OS fingerprinting tool utilizing ICMP signifies a substantial advancement in network reconnaissance capabilities. By capitalizing on the inherent features of ICMP for OS detection, the tool enables network administrators and security professionals to gain deeper insights into their network environments, identify potential security vulnerabilities, and strengthen defenses against malicious activities effectively.

2.2 PAPER 2: An Overview Of OS Fingerprinting Tools On The Internet

The importance of OS fingerprinting in network security cannot be emphasized enough. It acts as a crucial initial step in evaluating a network's security posture by enabling administrators to identify vulnerabilities and threats linked to specific operating systems. Gaining insight into the OS landscape within a network allows administrators to implement targeted security measures and strengthen defenses against potential attacks.

This paper has provided a detailed analysis of various tools utilized for OS fingerprinting, showcasing a range of approaches from active probing to passive analysis of network traffic. For example, Nmap is highlighted as a versatile and widely utilized tool that can perform numerous network scanning tasks, including OS detection. In contrast, p0f employs a distinctive passive fingerprinting method that analyzes network packets to infer the characteristics of an operating system without directly interacting with the target hosts.

Additionally, this paper emphasizes the need for responsible and ethical use of OS fingerprinting tools. Although these tools are vital for improving network security, improper use can raise privacy concerns and potentially violate legal regulations. Consequently, it is essential for administrators and security professionals to use these tools judiciously, ensuring that their application adheres to ethical standards and relevant laws.

As the landscape of OS fingerprinting continues to evolve rapidly, driven by advancements in network technologies and emerging security threats, future research may focus on developing more sophisticated fingerprinting techniques that can accurately identify elusive or heavily obfuscated operating systems. Furthermore, there is an increasing demand for tools that can seamlessly integrate with existing network security frameworks, providing real-time OS detection and response capabilities.

In conclusion, this research paper serves as a comprehensive resource for network administrators, security professionals, and researchers aiming to deepen their understanding of OS fingerprinting tools and their roles in enhancing network defenses. By applying the insights derived from this

study, organizations can strengthen their cybersecurity posture and protect against evolving threats in an increasingly interconnected digital landscape.

2.3 PAPER 3: A Passive Approach to Wireless Device Fingerprinting

In this research paper, we have introduced a novel passive approach to wireless device fingerprinting, which offers significant potential for enhancing network security and management within wireless environments. Wireless device fingerprinting is essential for network security, allowing administrators to identify and classify devices based on their unique characteristics and behavior patterns. Our passive methodology exploits the inherent properties of wireless communication, such as fluctuations in signal strength and transmission timing, to fingerprint devices without necessitating active interaction.

Through a series of experiments and analyses, we have demonstrated the effectiveness and reliability of our passive fingerprinting approach in accurately identifying wireless devices across a variety of network environments. By passively monitoring wireless traffic, our method can detect subtle variations in device behavior, generating unique fingerprints while avoiding alerting or interfering with the target devices.

A primary advantage of our passive approach is its non-intrusive nature, which reduces the risks of detection and interference commonly associated with active fingerprinting techniques. Operating quietly in the background, our method safeguards the privacy and integrity of wireless networks while providing valuable insights into device composition and behavior.

Additionally, our passive fingerprinting technique offers scalability and adaptability, making it well-suited for deployment across various wireless environments, including enterprise networks, public hotspots, and IoT ecosystems. With minimal resource requirements and low computational overhead, our approach can be seamlessly integrated into existing network infrastructures without disrupting normal operations.

Looking forward, future research may focus on enhancing our passive fingerprinting technique by incorporating machine learning algorithms for improved device classification and anomaly detection. There is also a need to assess the robustness of our approach against adversarial attacks and environmental factors that could impact fingerprinting accuracy.

In summary, this research paper advocates for the adoption of passive techniques in wireless device fingerprinting as an invaluable tool for network security and management. By leveraging the power of passive observation, organizations can gain deeper insights into their wireless ecosystems, proactively addressing security risks and operational challenges.

2.4 PAPER 4: Device Fingerprinting in Wireless Networks:Challenges And Opportunities

In conclusion, this research paper has thoroughly examined device fingerprinting in wireless networks, revealing both the challenges and opportunities present in this area. Device fingerprinting is a vital component of wireless security, allowing for the identification and classification of individual devices based on their unique characteristics and behaviors within a network.

Through an in-depth analysis of current techniques and methodologies, this study has highlighted the complex nature of device fingerprinting and the difficulties in accurately identifying and classifying various types of wireless devices. This paper explored a range of methods, from traditional MAC address-based fingerprinting to more sophisticated approaches that utilize device-specific features, such as signal strength patterns and traffic analysis.

Despite progress in device fingerprinting, several challenges remain. The increasing number of mobile and IoT devices, each with distinct communication protocols and behaviors, complicates the fingerprinting process. Furthermore, malicious actors often employ spoofing and evasion techniques to bypass detection, while the dynamic nature of wireless environments demands adaptive and resilient fingerprinting strategies to address real-world complexities.

Nonetheless, these challenges present significant opportunities for innovation within the field of wireless device fingerprinting. Emerging technologies, such as machine learning and artificial intelligence, offer promising pathways for enhancing the accuracy and efficiency of fingerprinting techniques, facilitating more reliable device identification in diverse and dynamic wireless settings.

Moreover, integrating device fingerprinting with broader security frameworks could significantly strengthen overall network security. By utilizing device fingerprints for access control, anomaly detection, and intrusion prevention, organizations can proactively address security threats and protect their wireless networks from unauthorized access and malicious activities.

In summary, this research emphasizes the critical role of device fingerprinting in wireless networks and highlights the necessity for ongoing research and innovation to overcome existing challenges and capitalize on emerging opportunities. By adopting a multidisciplinary approach that combines insights from networking, security, and data science, researchers and practitioners can develop more robust and effective device fingerprinting solutions, thereby enhancing the security and resilience of wireless networks in an increasingly interconnected environment.

2.5 PAPER 5: CANVuS: Context-Aware Network Vulnerability Scanning

In conclusion, this research paper has introduced CANVuS (Context-Aware Network Vulnerability Scanning), an innovative approach to network vulnerability assessment that utilizes contextual information to improve scanning accuracy and efficiency. By incorporating environmental factors such as network topology, asset criticality, and user behavior, CANVuS overcomes the shortcomings of traditional vulnerability scanning tools, which often generate numerous false positives and overlook significant vulnerabilities.

Through a thorough evaluation and comparison with existing scanning tools, CANVuS has demonstrated enhanced performance regarding vulnerability detection rates and the reduction of false positives. By using contextual cues to prioritize scanning efforts and customize scanning parameters to specific network scenarios, CANVuS significantly improves the accuracy of vulnerability assessments while minimizing scan duration and resource consumption.

Additionally, CANVuS features adaptive scanning strategies and dynamic risk scoring, enabling organizations to modify their scanning approaches in response to changing threat landscapes and network conditions. By delivering actionable insights and prioritized vulnerability reports, CANVuS empowers security teams to concentrate their remediation efforts on the most critical vulnerabilities, thereby strengthening their overall security posture and reducing exposure to cyber threats.

Beyond its technical innovations, CANVuS emphasizes the importance of context-awareness in vulnerability management and highlights the advantages of integrating contextual information into security tools and frameworks. By understanding the broader context in which vulnerabilities arise, organizations can make more informed decisions about risk mitigation strategies and resource allocation, ultimately enhancing their capacity to detect, prioritize, and address security vulnerabilities effectively.

Looking ahead, future research on CANVuS may focus on further refining contextual modeling techniques, integrating with threat intelligence sources, and enhancing scalability to support large-scale enterprise networks. Additionally, investigating the applicability of CANVuS in various network environments, including cloud-based and IoT infrastructures, could provide valuable insights into its versatility and effectiveness across different deployment scenarios.

In summary, CANVuS marks a significant advancement in the realm of network vulnerability scanning, presenting a context-aware approach that enhances the accuracy, efficiency, and effectiveness of vulnerability assessment processes. By prioritizing context-awareness in vulnerability management, organizations can bolster their cyber resilience and better defend against

evolving threats in an increasingly complex and dynamic digital landscape.

2.6 PAPER 6: Port scan detection

In conclusion, this research paper has tackled the significant challenge of detecting port scans in network security, aiming to improve organizations' capabilities in identifying and mitigating such activities effectively. Port scanning is a prevalent reconnaissance method utilized by attackers to uncover vulnerabilities and potential entry points within networks, posing serious threats to security and integrity.

This study has conducted a thorough review of existing port scan detection techniques, emphasizing the difficulties in accurately differentiating between legitimate network traffic and malicious scanning actions. Traditional methods, such as threshold-based detection and signature matching, often face issues with high false positive rates and limited scalability, highlighting the need for more sophisticated and resilient detection mechanisms.

By employing machine learning algorithms and anomaly detection techniques, this research proposes innovative strategies for port scan detection that promise enhanced accuracy and efficiency. By analyzing network traffic patterns and identifying deviations from typical behavior, these methods facilitate the detection of port scanning activities with greater precision while reducing false positives.

Moreover, incorporating contextual information—such as network topology and historical traffic patterns—improves the effectiveness of port scan detection by providing essential context for analyzing and interpreting network behavior. This broader perspective enables organizations to better differentiate between benign and malicious activities, fostering more proactive and targeted responses to potential threats.

Through empirical evaluation and comparative analysis, the proposed port scan detection techniques have shown promising outcomes regarding detection accuracy, false positive rates, and computational efficiency. By surpassing traditional detection methods, these approaches present practical solutions for strengthening network security and defending against port scanning attacks.

Looking ahead, future research may focus on refining machine learning models, exploring ensemble-based detection strategies, and integrating with existing network security frameworks for real-time threat responses. Additionally, ongoing collaboration between researchers and practitioners is vital for validating the proposed detection techniques in real-world network settings and effectively addressing evolving threats.

In summary, this research contributes to the advancement of port scan detection techniques,

presenting innovative methods that leverage machine learning and contextual information to improve detection accuracy and efficiency. By adopting these techniques, organizations can enhance their network defenses and mitigate the risks associated with port scanning activities, thereby protecting their assets and ensuring the integrity of their networks.

2.7 PAPER 7: The Application Of ICMP Protocol In Network Scanning

In conclusion, this research paper has explored the role of the Internet Control Message Protocol (ICMP) in network scanning, underscoring its adaptability and efficacy in collecting essential information about networked devices. As a fundamental component of the Internet Protocol Suite, ICMP is a vital resource for network administrators and security professionals aiming to evaluate the health and connectivity of devices on a network.

This study provides a thorough review of existing literature and empirical analysis to illustrate the various applications of ICMP in network scanning. From basic connectivity tests using ICMP Echo Request (ping) packets to more sophisticated methods such as ICMP Timestamp and Address Mask requests, ICMP encompasses a variety of functions that facilitate detailed network reconnaissance.

One significant advantage of ICMP-based scanning is its lightweight and non-intrusive nature, which makes it particularly suitable for conducting preliminary network assessments and identifying reachable hosts without generating excessive network traffic or disrupting normal operations. Furthermore, ICMP techniques can yield valuable insights into network topology, device availability, and potential security vulnerabilities, enabling organizations to proactively address issues and strengthen their overall network security posture.

The paper also examines the limitations and challenges associated with ICMP-based scanning, including the risks of network filtering and evasion tactics that adversaries may employ to hide their activities or hinder scanning efforts. By understanding these challenges, network administrators can adopt mitigation strategies to enhance the reliability and accuracy of ICMP-based scanning initiatives.

Through empirical evaluation and comparative analysis, this research has demonstrated the practical benefits of utilizing ICMP in network scanning, effectively showcasing its capacity to detect hosts, assess reachability, and identify potential security risks. By implementing ICMP-based scanning techniques, organizations can streamline their network reconnaissance, deepen their understanding of networked environments, and bolster their overall security defenses.

Looking ahead, future research should focus on refining ICMP-based scanning methods, exploring innovative applications in emerging network frameworks such as cloud and IoT

environments, and addressing potential privacy issues related to the collection of ICMP data. Additionally, ongoing collaboration between researchers and practitioners is vital for advancing the field of ICMP-based network scanning and effectively addressing evolving threats in the dynamic landscape of network security.

In summary, this research enhances the understanding of the application of the ICMP protocol in network scanning, providing insights into its capabilities, limitations, and potential for improving network reconnaissance efforts. By leveraging ICMP effectively, organizations can gain crucial visibility into their networked environments, identify possible security vulnerabilities, and proactively mitigate risks, thereby fortifying their overall security posture and resilience against cyber threats.

2.8 PAPER 8: Network Forensic System For Port Scanning Attack

In conclusion, this research paper presents a detailed network forensic system specifically developed to detect and mitigate port scanning attacks. Port scanning is a prevalent reconnaissance method utilized by attackers to discover vulnerable systems and potential access points within a network, posing serious threats to network security and integrity.

This study thoroughly examines existing literature and conducts empirical analyses to identify the primary challenges associated with port scanning attacks. It emphasizes the urgent need for advanced detection and response mechanisms to effectively address these threats. Traditional methods for detecting port scans often face high false positive rates and limited scalability, highlighting the necessity for more robust and efficient forensic systems.

The proposed network forensic system utilizes a blend of signature-based detection, anomaly detection, and behavioral analysis techniques to accurately identify and respond to port scanning activities in real-time. By monitoring network traffic patterns, analyzing packet payloads, and correlating events across multiple network devices, the system provides a comprehensive view of network activity, facilitating the rapid detection and mitigation of port scanning attacks.

Additionally, incorporating contextual information—such as network topology, historical traffic data, and known attack signatures—enhances the forensic system's effectiveness by providing crucial context for analyzing and interpreting suspicious activities. By understanding the broader context in which port scanning occurs, organizations can more effectively differentiate between benign and malicious behaviors, allowing for more targeted and proactive responses to potential threats.

Empirical evaluations and comparative analyses demonstrate that the proposed network forensic

system achieves promising results in detection accuracy, false positive rates, and response times. By surpassing traditional detection methods, the system offers practical solutions for improving network security and defending against port scanning attacks.

Looking ahead, future research may focus on refining detection algorithms, exploring machine learning and artificial intelligence techniques for anomaly detection, and integrating the system with existing security frameworks for automated incident response. Additionally, ongoing collaboration between researchers and practitioners is crucial for validating the system's effectiveness in real-world network environments and effectively addressing emerging threats.

In summary, this research advances network forensic techniques for detecting and mitigating port scanning attacks, presenting a comprehensive system that integrates multiple detection methods and contextual information to improve detection accuracy and response capabilities. By adopting this forensic system, organizations can fortify their network defenses, mitigate risks related to port scanning attacks, and protect their assets and sensitive information from malicious actors.

2.9 PAPER 9: Advanced Passive Operating System Fingerprinting Using Machine Learning and Deep Learning

In conclusion, this research paper has investigated the evolution of passive operating system fingerprinting techniques through the application of machine learning (ML) and deep learning (DL) methods. OS fingerprinting is crucial for network security, as it facilitates the identification of devices and potential vulnerabilities within a network without the need for active probing.

Through an in-depth analysis of current passive OS fingerprinting techniques and the incorporation of ML and DL algorithms, this study has highlighted significant enhancements in both accuracy and efficiency. By utilizing features derived from network traffic and system behaviors, ML and DL models can effectively identify subtle characteristics and patterns of operating systems, leading to more precise OS identification.

The incorporation of ML and DL in passive OS fingerprinting presents several benefits, such as increased adaptability to various network environments, greater resilience against evasion tactics, and less dependence on manual feature selection. Additionally, these advanced techniques allow for continuous learning and adaptation to changes in OS behaviors, ensuring ongoing effectiveness in dynamic network environments.

Empirical evaluations have illustrated the superior performance of ML and DL-based passive OS fingerprinting methods compared to traditional techniques. By utilizing extensive datasets and

advanced learning algorithms, these methods achieve higher accuracy while reducing false positives and false negatives, thereby strengthening overall network security.

Looking ahead, future research could focus on refining ML and DL models, exploring ensemble learning methods, and integrating these techniques with real-time network monitoring systems for proactive threat detection. Moreover, it is vital to investigate the implications of adversarial attacks and privacy issues on ML-based fingerprinting methods to ensure their robustness and ethical application in real-world scenarios.

In summary, this research signifies a noteworthy advancement in passive OS fingerprinting by employing ML and DL techniques to attain exceptional levels of accuracy and efficiency. By adopting these innovative approaches, organizations can improve their capability to identify and address potential security risks, thereby enhancing their overall network defenses and resilience against evolving cyber threats.

2.10 PAPER 10: A Network Scanning Detection Method Based On TCP Flow State

In conclusion, this research paper has presented an innovative approach to network scanning detection through the analysis of TCP flow states. Network scanning, often a precursor to cyberattacks, poses significant risks to the security and integrity of networks, highlighting the need for effective detection mechanisms.

This study provides a comprehensive examination of existing literature alongside empirical analysis, demonstrating the viability and effectiveness of utilizing TCP flow state information for identifying scanning activities. By assessing the temporal and spatial characteristics of TCP flows, the proposed method can accurately detect scanning behaviors while maintaining a low rate of false positives.

A primary strength of this detection method is its capacity to identify subtle anomalies in TCP flow patterns that suggest scanning activities, all while minimizing interference with regular network traffic. By monitoring variations in flow rates, packet sizes, and inter-arrival times, this approach offers a thorough solution for recognizing various scanning techniques, including SYN, ACK, and FIN scans.

Empirical evaluations have confirmed the proposed method's effectiveness in detecting a broad range of scanning activities across different network environments. By employing statistical analysis and machine learning techniques, the method achieves high detection rates and adapts to changing

scanning behaviors and network conditions.

Looking ahead, future research may focus on refining detection algorithms, exploring ensemble learning methods, and integrating this approach with real-time network monitoring systems for proactive threat detection. Additionally, it will be essential to investigate the scalability and performance implications of this method in large-scale network environments to ensure its practical applicability.

In summary, the development of a network scanning detection method based on TCP flow state analysis marks a significant step forward in enhancing network security. By leveraging TCP flow data, this method provides a robust and efficient means of detecting scanning activities, enabling organizations to proactively identify and mitigate potential security risks, thus protecting their networks from cyber threats.

2.11 PAPER 11: Comparing Intrusion Detection Systems (IDS) and Intrusion Prevention Systems (IPS)

In conclusion, this research paper has presented a detailed comparison of Intrusion Detection Systems (IDS) and Intrusion Prevention Systems (IPS), which are essential elements of contemporary network security frameworks. By thoroughly examining their functionalities, deployment strategies, response mechanisms, levels of control, performance implications, and security effectiveness, this study has clarified the key differences between IDS and IPS.

IDS functions as a passive monitoring tool that identifies suspicious network activity and generates alerts for further investigation. Although it provides detailed insights into network traffic and system behavior, IDS does not possess the capability to take immediate action against intrusions, relying instead on human intervention for incident response and remediation.

Conversely, IPS operates actively, intercepting and analyzing network traffic in real-time to enforce predefined security policies. By automatically blocking or filtering harmful traffic, IPS delivers proactive protection against security threats, thereby reducing the likelihood of breaches and limiting the attack window for potential intruders.

The deployment models for IDS and IPS also differ significantly; IDS is often deployed in passive or inline configurations, while IPS is typically placed inline to actively thwart or mitigate threats. This difference in deployment directly impacts their performance, with IDS generally imposing less performance overhead compared to IPS.

Moreover, the level of control provided by IPS enables administrators to define and enforce

security policies with precision, allowing for detailed management of network traffic. However, this heightened level of control can introduce added complexity in policy management and configuration.

Despite their differences, IDS and IPS fulfill complementary functions within network security, with IDS offering visibility and detection capabilities and IPS providing proactive defense and threat mitigation. The decision to deploy IDS or IPS should be based on an organization's specific security needs, operational considerations, and risk tolerance.

In summary, this research underscores the importance of understanding the strengths, limitations, and operational implications of IDS and IPS to make well-informed choices about their deployment and integration within network security infrastructures. By effectively utilizing both IDS and IPS, organizations can strengthen their overall security posture and enhance their resilience against the evolving landscape of cyber threats.

2.12 PAPER 12: Intrusion/Prevention and Intrusion detection system For Wi-fi Networks

In conclusion, this research paper has explored the significance of Intrusion Detection Systems (IDS) and Intrusion Prevention Systems (IPS) in bolstering the security of Wi-Fi networks. Given the widespread nature of Wi-Fi networks and their susceptibility to various security threats, robust intrusion detection and prevention mechanisms are essential to protect against unauthorized access and malicious activities.

Through an extensive analysis of the functions, deployment considerations, and effectiveness of IDS and IPS within Wi-Fi environments, several key insights have emerged. IDS operates as a passive monitoring tool that identifies suspicious activities and generates alerts for further investigation, while IPS functions actively by intercepting and neutralizing threats in real-time.

Implementing IDS and IPS in Wi-Fi networks necessitates careful consideration of network topology, traffic patterns, and security needs. IDS can be deployed in a passive mode to analyze Wi-Fi traffic for anomalies, while IPS is typically positioned inline to actively block or filter malicious traffic based on established policies.

The effectiveness of IDS and IPS in Wi-Fi networks hinges on their capacity to detect and address a broad spectrum of security threats, including rogue access points, unauthorized devices, and malicious activities such as denial-of-service attacks and intrusion attempts. Both systems fulfill complementary roles in threat mitigation, with IDS providing essential visibility and detection capabilities, and IPS delivering proactive protection and prevention measures.

Nonetheless, deploying IDS and IPS in Wi-Fi networks may pose challenges, including performance overhead, false positives, and complexities in policy management. Overcoming these challenges requires meticulous planning, optimization of configurations, and continuous monitoring and adjustment of IDS and IPS implementations.

In summary, this research highlights the crucial role of IDS and IPS in strengthening the security of Wi-Fi networks, offering valuable insights into their functionalities, deployment strategies, and effectiveness in mitigating security threats. By effectively leveraging the capabilities of IDS and IPS, organizations can enhance their Wi-Fi network security posture, reducing the risks of unauthorized access and malicious activities and ensuring the integrity, confidentiality, and availability of their Wi-Fi networks.

2.13 PAPER 13: Intrusion Detection Using Network Monitoring Tools

In conclusion, this research paper has examined the role of network monitoring tools in intrusion detection, highlighting their importance in strengthening cybersecurity measures. Intrusion Detection Systems (IDS) are essential components of contemporary network defense strategies, designed to identify and mitigate unauthorized access attempts and malicious activities.

Through a thorough analysis of network monitoring tools and their functions, this study has showcased the various capabilities and deployment options available for intrusion detection. Tools such as packet sniffers, flow analyzers, and network-based IDS solutions provide different levels of visibility and detection effectiveness.

The success of intrusion detection with network monitoring tools relies on several factors, including the extent of monitoring, detection algorithms, and response protocols. By utilizing real-time analysis of network traffic patterns, anomalies, and known signatures, these tools can pinpoint suspicious activities that may indicate security breaches.

Moreover, deploying network monitoring tools for intrusion detection requires careful consideration of the network's architecture, traffic volume, and established security policies. Integrating IDS with existing security frameworks and incident response processes enhances the overall security posture of organizations and facilitates proactive threat management.

Nevertheless, challenges such as false positives, alert fatigue, and evasion techniques used by attackers highlight the need for ongoing refinement and enhancement of intrusion detection strategies. Advanced methods, including machine learning algorithms and behavior analysis, present promising opportunities to improve detection accuracy and minimize false positives.

In summary, this research emphasizes the critical role of intrusion detection through network

monitoring tools in protecting against cyber threats. By effectively utilizing the capabilities of IDS and other network monitoring solutions, organizations can improve their ability to promptly detect and respond to security incidents, thereby reducing potential risks and safeguarding essential assets and information.

2.14 PAPER 14: Quantitative Assessment of Vulnerability Scanning

In conclusion, this research paper has delivered a quantitative evaluation of vulnerability scanning techniques, emphasizing their significance in managing cybersecurity risks. Vulnerability scanning is essential for identifying and prioritizing security weaknesses across networks, systems, and applications, enabling organizations to take proactive steps to reduce risks and improve their overall security posture.

Through a detailed analysis of vulnerability scanning methodologies, tools, and metrics, this study has illustrated the effectiveness and limitations of quantitative approaches in assessing security vulnerabilities. By measuring the severity, likelihood, and potential impact of these vulnerabilities, organizations can prioritize remediation efforts according to risk, resource availability, and business objectives.

The research highlights the necessity of a systematic and standardized approach to vulnerability scanning, which includes selecting suitable scanning tools, determining scanning frequencies, and establishing assessment criteria. Moreover, integrating vulnerability scanning with broader risk management frameworks and compliance obligations ensures that organizational goals align with regulatory requirements.

Empirical evaluations have indicated that quantitative vulnerability assessment techniques provide valuable insights into an organization's security posture, facilitating informed decision-making and effective resource allocation. By utilizing quantitative metrics like Common Vulnerability Scoring System (CVSS) scores, organizations can focus on vulnerabilities based on their potential impact on business operations and data integrity.

However, challenges such as false positives, inaccuracies in scanning, and limitations in coverage persist in vulnerability scanning practices. Tackling these issues necessitates ongoing enhancements in scanning methodologies, tool capabilities, and the integration of threat intelligence.

Looking ahead, future research could focus on developing advanced vulnerability scanning techniques, such as dynamic and contextual scanning approaches, to improve accuracy and coverage. Additionally, investigating the use of machine learning and artificial intelligence technologies for automated prioritization and remediation of vulnerabilities can further streamline the vulnerability

management process.

In summary, this research enriches the understanding of quantitative vulnerability assessment techniques and their role in managing cybersecurity risks. By implementing a systematic and data-driven approach to vulnerability scanning, organizations can effectively identify, prioritize, and mitigate security weaknesses, ultimately decreasing the likelihood and impact of cyber threats on their operations and assets.

2.15 PAPER 15: Advanced Network Scanning

In conclusion, this research paper has thoroughly examined advanced network scanning techniques, emphasizing their critical role in contemporary cybersecurity practices. Network scanning is essential for network reconnaissance, allowing organizations to evaluate their security posture, identify vulnerabilities, and proactively guard against potential threats.

Through an in-depth investigation of various network scanning methodologies, tools, and strategies, this study has illuminated the evolving landscape of reconnaissance techniques. From traditional port scanning to more sophisticated methods such as service fingerprinting, operating system detection, and vulnerability assessment, numerous techniques provide differing levels of insight into network environments.

The research highlights the necessity of a multi-faceted approach to network scanning, advocating for the integration of complementary techniques to achieve a comprehensive understanding of network assets and configurations. By utilizing automated scanning tools, threat intelligence feeds, and machine learning algorithms, organizations can significantly enhance their capability to efficiently detect and respond to security risks.

Empirical evaluations have shown the effectiveness of advanced network scanning techniques in uncovering vulnerabilities, misconfigurations, and potential attack vectors within networks. By performing thorough scans and analyzing the results, organizations can prioritize remediation efforts, allocate resources effectively, and bolster their overall cybersecurity posture.

Nonetheless, challenges such as evasion tactics, false positives, and issues with scan accuracy persist in advanced network scanning practices. Overcoming these challenges will require ongoing research and development to refine scanning methodologies, enhance tool capabilities, and improve threat intelligence integration.

Looking ahead, future research may explore emerging technologies such as software-defined networking (SDN) and Internet of Things (IoT) security, which introduce unique challenges and opportunities for network scanning. Additionally, incorporating advanced analytics and visualization

techniques can enable deeper analysis and better interpretation of scanning results.

In summary, this research underscores the significance of advanced network scanning techniques in bolstering cybersecurity resilience and facilitating proactive threat mitigation. By adopting the insights gained from this study and utilizing advanced scanning tools and methodologies, organizations can strengthen their defenses, reduce security risks, and protect against the ever-evolving cyber threats present in today's interconnected digital landscape.

2.16 PAPER 16: An Examination of Software-Defined Networking

In conclusion, this research paper has conducted a thorough examination of Software-Defined Networking (SDN), highlighting its transformative potential and significant implications for contemporary network architectures. SDN marks a fundamental shift in how networks are managed and controlled, offering centralized programmability, agility, and scalability to address the evolving demands of today's digital environments.

Through a detailed analysis of SDN concepts, architectures, technologies, and applications, this study has clarified the foundational principles and practical considerations associated with the adoption of SDN. By separating network control from data forwarding functions, SDN facilitates more flexible, efficient, and responsive network operations, fostering innovation and enabling the deployment of new services and applications.

The research has emphasized the wide-ranging capabilities and advantages of SDN across various sectors, including data centers, wide area networks (WANs), and edge computing environments. From dynamic network provisioning and traffic management to network slicing and virtualization, SDN provides unprecedented levels of control, visibility, and automation, empowering organizations to optimize their network resources and improve user experiences.

Empirical evaluations and case studies have illustrated the real-world benefits and potential of SDN in enhancing network performance, reliability, and security. By abstracting network control into software-based controllers and utilizing programmable network devices, SDN allows organizations to respond effectively to changing traffic patterns, address security threats, and drive innovation.

Nevertheless, challenges related to interoperability, scalability, and security are crucial considerations in the deployment and operation of SDN. Overcoming these challenges necessitates ongoing research and collaboration to establish standardized protocols, create interoperable solutions, and develop robust security mechanisms.

Looking ahead, future research may focus on emerging trends such as intent-based networking

(IBN), the integration of artificial intelligence (AI), and blockchain-enabled SDN architectures. Additionally, examining the effects of SDN on network management practices, organizational workflows, and business models can yield valuable insights into its long-term impact and trajectory for adoption.

In summary, this research highlights the transformative potential of Software-Defined Networking (SDN) in reshaping network architectures and operations. By adopting the principles of SDN and leveraging its capabilities, organizations can unlock new opportunities for innovation, agility, and efficiency, paving the way for a more dynamic and resilient digital future.

2.17 PAPER 17: Examining the Use of Software Defined Networking for Enhancing Network Security: A Survey

In conclusion, this research paper has performed a thorough survey of how Software-Defined Networking (SDN) can enhance network security, revealing its promising potential and important considerations in cybersecurity practices. SDN is a transformative technology that offers centralized control, programmability, and automation, allowing organizations to strengthen their security defenses and effectively address evolving cyber threats.

By examining SDN-based security solutions, architectures, and deployments, this study has showcased the various applications and advantages of SDN in improving the overall security posture of networks. From dynamic policy enforcement and threat detection to network segmentation and access control, SDN provides a flexible and scalable framework for implementing effective security measures across diverse network environments.

The research emphasizes the necessity of integrating SDN with existing security technologies and frameworks to tackle emerging security challenges such as distributed denial-of-service (DDoS) attacks, insider threats, and data breaches. By leveraging the programmability and real-time visibility of SDN, organizations can enhance their capability to detect, respond to, and mitigate security incidents.

Empirical evaluations and case studies have illustrated the effectiveness of SDN-based security solutions in improving threat detection accuracy, shortening incident response times, and reducing security risks. By abstracting network control and applying security policies at a centralized level, SDN allows organizations to achieve greater consistency, agility, and resilience in their security operations.

Nonetheless, challenges related to interoperability, scalability, and complexity are significant

considerations in deploying SDN-based security solutions. Addressing these challenges calls for collaboration among researchers, industry stakeholders, and policymakers to establish standardized protocols, interoperable solutions, and best practices for secure SDN implementations.

Looking ahead, future research may explore advanced security features within SDN architectures, such as encryption, authentication, and anomaly detection. Additionally, investigating the effects of SDN on regulatory compliance, privacy, and data protection can provide valuable insights into its broader societal and ethical implications.

In summary, this research underscores the transformative potential of Software-Defined Networking (SDN) in enhancing network security, offering a flexible and scalable framework for implementing proactive security measures in today's dynamic threat landscape. By adopting SDN-based security solutions and effectively leveraging their capabilities, organizations can strengthen their security posture, reduce risks, and protect their assets and data from cyber threats.

2.18 PAPER 18: Advanced Passive Operating System Fingerprinting

In conclusion, this research paper has examined the field of advanced passive operating system (OS) fingerprinting techniques, highlighting their importance in network reconnaissance and cybersecurity. OS fingerprinting is vital for comprehending networked environments, allowing organizations to identify and categorize devices without active probing or intrusion.

Through a thorough analysis of advanced passive OS fingerprinting methodologies, tools, and strategies, this study has unveiled the varied capabilities and applications of these techniques. By examining network traffic patterns and packet headers and correlating passive observations with known OS characteristics, a spectrum of methods offers different levels of accuracy and reliability in OS identification.

The research emphasizes the benefits of passive OS fingerprinting compared to active probing methods, such as minimizing network footprint, reducing detection risks, and increasing stealth. Utilizing passive observation techniques allows organizations to collect valuable OS-related data without directly interacting with target devices, thereby lowering the likelihood of detection by intrusion detection systems.

Empirical evaluations and case studies have shown the effectiveness of advanced passive OS fingerprinting techniques in accurately identifying target operating systems across various network environments. By scrutinizing subtle variations in network behavior and packet characteristics, passive fingerprinting methods can achieve high accuracy rates while reducing false positives and detection footprints.

However, challenges such as evasion techniques, obfuscation methods, and dynamic network conditions continue to pose obstacles in passive OS fingerprinting practices. Overcoming these challenges necessitates ongoing research and development to enhance fingerprinting algorithms, data analysis techniques, and evasion detection methods.

Looking ahead, future research may explore innovative passive fingerprinting approaches, harnessing emerging technologies like machine learning and artificial intelligence to improve OS identification accuracy. Furthermore, examining the effects of encrypted traffic and privacy concerns on passive fingerprinting techniques could yield valuable insights into their applicability and limitations in real-world network environments.

In summary, this research highlights the significance of advanced passive operating system fingerprinting techniques in network reconnaissance and cybersecurity operations. By adopting passive fingerprinting methodologies and effectively leveraging their capabilities, organizations can deepen their understanding of networked environments, identify potential security risks, and strengthen their defenses against cyber threats.

2.19 PAPER 19: A Survey of OS Fingerprinting Tools Available Online

In conclusion, this research paper has provided a thorough survey of online OS fingerprinting tools, offering significant insights into their functionalities, features, and relevance to network reconnaissance and cybersecurity practices. OS fingerprinting is essential for identifying and categorizing devices within networked environments, enabling organizations to evaluate their security posture and identify potential vulnerabilities.

Through an extensive analysis of OS fingerprinting tools, this study has showcased a wide array of options, including both open-source and commercial solutions, as well as web-based and command-line interfaces. Each tool presents unique capabilities and methodologies for collecting OS-related information, such as examining network responses, packet headers, or device behavior patterns.

The research emphasizes the importance of choosing the right OS fingerprinting tool based on specific use cases, network environments, and desired results. By considering factors like accuracy, speed, user-friendliness, and compatibility with target devices, organizations can make well-informed decisions regarding tool selection and implementation strategies.

Empirical evaluations and comparative analyses have yielded valuable insights into the strengths and weaknesses of various OS fingerprinting tools, assisting organizations in identifying the most appropriate options for their security requirements. By employing multiple tools and

techniques together, organizations can enhance the precision and reliability of OS identification while minimizing the risk of false positives and negatives.

However, challenges such as evasion techniques, detection avoidance strategies, and the evolving behavior of operating systems present ongoing obstacles for OS fingerprinting practices. Addressing these challenges necessitates ongoing research and development efforts to enhance tool capabilities, update fingerprinting databases, and adapt to changing network conditions.

Looking ahead, future research may explore advanced fingerprinting techniques, utilize machine learning and artificial intelligence algorithms for automated OS identification, and foster collaboration and information sharing among tool developers and cybersecurity practitioners.

In summary, this research enhances the understanding of available online OS fingerprinting tools, providing insights into their functionalities, strengths, and limitations. By utilizing the findings of this survey and selecting suitable tools for their security needs, organizations can improve their network reconnaissance capabilities, deepen their comprehension of networked environments, and effectively bolster their overall cybersecurity posture.

2.20 PAPER 20: Device Fingerprinting in Wireless Networks

In conclusion, this research paper has explored the topic of device fingerprinting in wireless networks, highlighting its critical role in enhancing network security and management. Device fingerprinting is a vital element of wireless network reconnaissance, allowing organizations to recognize and categorize devices based on distinct characteristics and attributes.

Through a thorough examination of various device fingerprinting techniques, methodologies, and their applications within wireless networks, this study has uncovered the different capabilities and implications of these practices. By assessing factors such as device MAC addresses, signal strength, and unique device identifiers, a variety of techniques provide different levels of accuracy and reliability in identifying devices.

The research has emphasized the significance of device fingerprinting in reducing security risks, identifying unauthorized devices, and implementing access control measures in wireless environments. By keeping an accurate inventory of authorized devices and monitoring for unusual activities or unauthorized access attempts, organizations can strengthen their network security posture and defend against potential threats.

Empirical studies and case analyses have illustrated the success of device fingerprinting techniques in effectively identifying and classifying devices across various wireless network contexts. By utilizing both passive and active fingerprinting methods, organizations can achieve

high accuracy levels while minimizing false positives and detection footprints.

However, challenges such as device variety, mobility, and privacy concerns continue to pose issues in device fingerprinting practices. Overcoming these challenges necessitates continuous research and development to refine fingerprinting algorithms, improve scalability and adaptability, and consider privacy and ethical implications.

Looking ahead, future research could focus on investigating advanced fingerprinting techniques, employing machine learning and artificial intelligence for automated device identification, and integrating device fingerprinting within comprehensive network security frameworks.

In summary, this research highlights the vital role of device fingerprinting in wireless networks and its contribution to improving security and management practices. By adopting effective fingerprinting methodologies and utilizing their capabilities, organizations can bolster their network defenses, reduce security risks, and protect their wireless environments against evolving cyber threats.

Chapter 3

Analysis / Software Requirements Specification (SRS)

3.1 Introduction

The goal of this document is to outline the specifications for creating a Network Scanning Tool. It intends to offer network administrators a complete solution for overseeing and analyzing network systems. The software will include features for detecting devices, scanning ports, evaluating vulnerabilities, and producing reports. It will be adaptable to different network environments, from small setups to large enterprise networks.

3.2 Overall Description

1. Product Perspective

The Network Scanning Tool will function as an independent software application designed to integrate within existing network infrastructure. It will engage with network devices and systems to execute scanning and analysis operations.

2. Product Features

- Device discovery utilizing protocols like ICMP, ARP, and SNMP.
- Port scanning features encompassing TCP, UDP, and SYN scans.
- Vulnerability assessment through methods such as CVE lookup and signature-based detection.
- Customizable reporting capabilities, allowing data export in various formats.
- Logging features to capture scan results and track user activities.

3. User Classes and Characteristics

User classes will comprise network administrators, security professionals, and IT staff. Users may possess varying levels of technical expertise, ranging from beginners to advanced users.

4. Operating Environment

The software will be designed for compatibility with major operating systems, including Windows, Linux, and macOS. It will also support both IPv4 and IPv6 network protocols.

3.3 Specific Requirements

1. External Interface Requirements

The software will include a user-friendly graphical interface designed for easy navigation and configuration. It will offer APIs for integration with external systems such as SIEM (Security Information and Event Management) platforms, ticketing systems, and solutions for vulnerability management.

2. Functional Requirements

The device discovery feature will support SNMPv1, SNMPv2c, and SNMPv3 protocols, allowing users to customize polling intervals and SNMP community strings. Port scanning capabilities will encompass TCP connect scans, SYN scans, and UDP scans, with adjustable scan parameters and timeout settings. The vulnerability assessment function will check for known vulnerabilities using databases like CVE and NVD (National Vulnerability Database), and will provide options for custom vulnerability definitions and exclusions. Reporting features will enable the generation of executive summaries, in-depth reports, and trend analysis charts, along with scheduling and email notification options. The logging functionality will track scan activities, errors, warnings, and user interactions, with features for log rotation, archiving, and exporting.

3. Performance Requirements

The software is expected to demonstrate rapid scanning speeds while minimizing its impact on network performance and resource usage. It should effectively manage large-scale network environments containing thousands of devices and services.

4. Security Requirements

The software will incorporate secure authentication processes, including password policies, multi-factor authentication, and LDAP integration. Data encryption will be utilized to safeguard sensitive information transmitted over the network, such as scan results and user credentials. Role-based access control will ensure that access to sensitive features and data is limited based on user roles and permissions.

5. Software Quality Attributes

Reliability will be achieved through robust error handling, automated testing, and consistent software updates. Maintainability will be supported by a modular design, comprehensive documentation, and version control practices. Usability will be improved with user-centered design, contextual help features, and training materials for users.

6. Documentation Requirements

Documentation will include user manuals, installation guides, and API documentation, available in both digital and printed formats. Technical documentation for developers will address architecture, APIs, data models, and options for customization.

7. Constraints

The development timeline will align with project milestones and delivery deadlines. Budget constraints and resource availability will be taken into account during the development and implementation phases.

8. Assumptions and Dependencies

Assumptions include the availability of network devices, necessary access permissions, and network connectivity during scanning processes. Dependencies may involve third-party libraries, frameworks, and APIs for functionalities like reporting, logging, and vulnerability assessment.

3.4 Visual Representations

3.4.1 Level 0 DFD

Figure 3.1: Level 0 DFD

3.4.2 Level 1 DFD

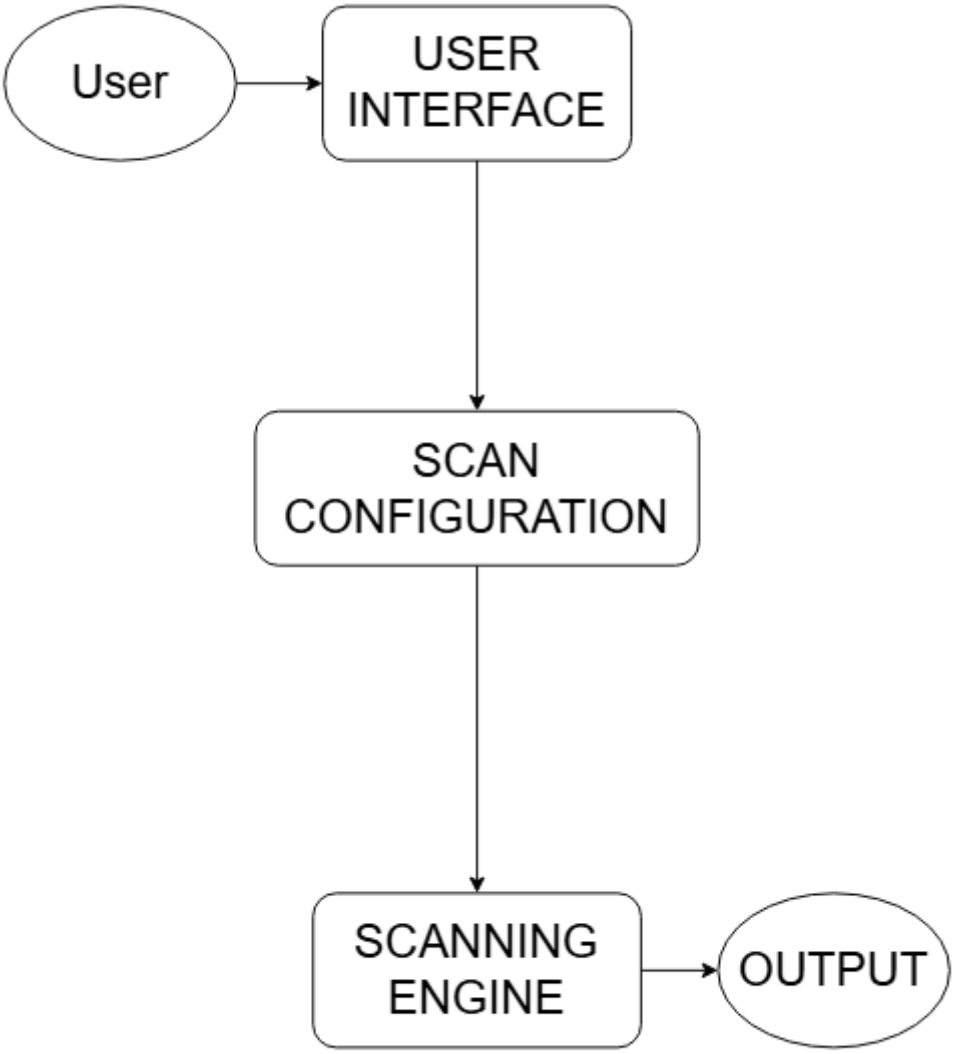

Figure 3.2: Level 1 DFD

3.4.3 Level 2 DFD

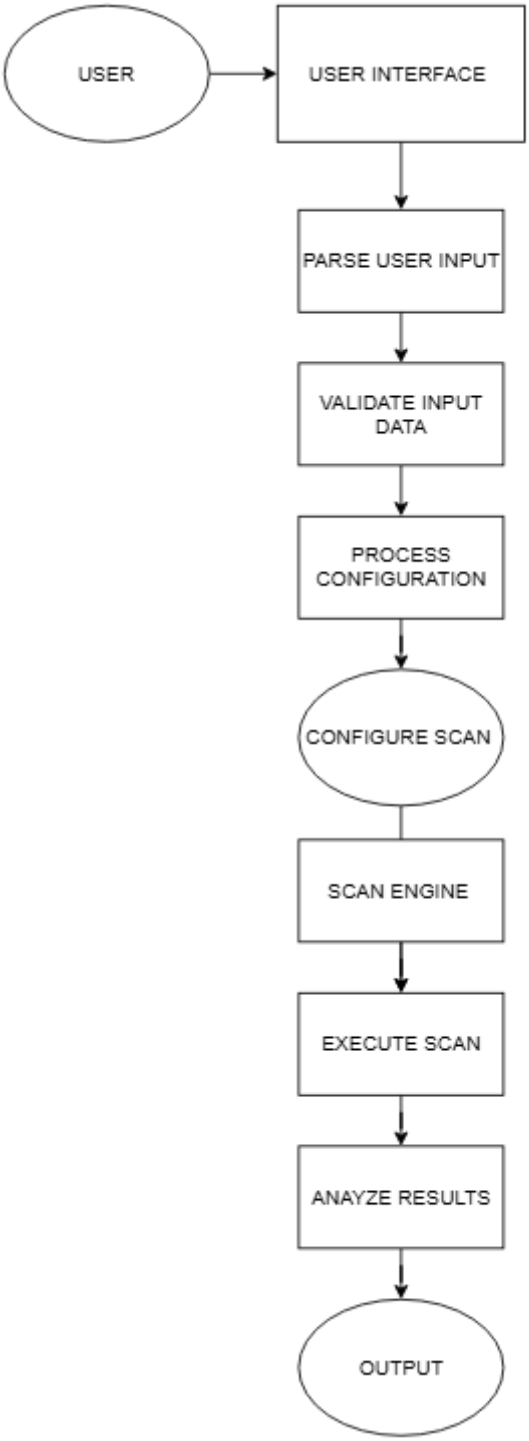

Figure 3.3: Level 2 DFD

3.4.4 UML Diagram

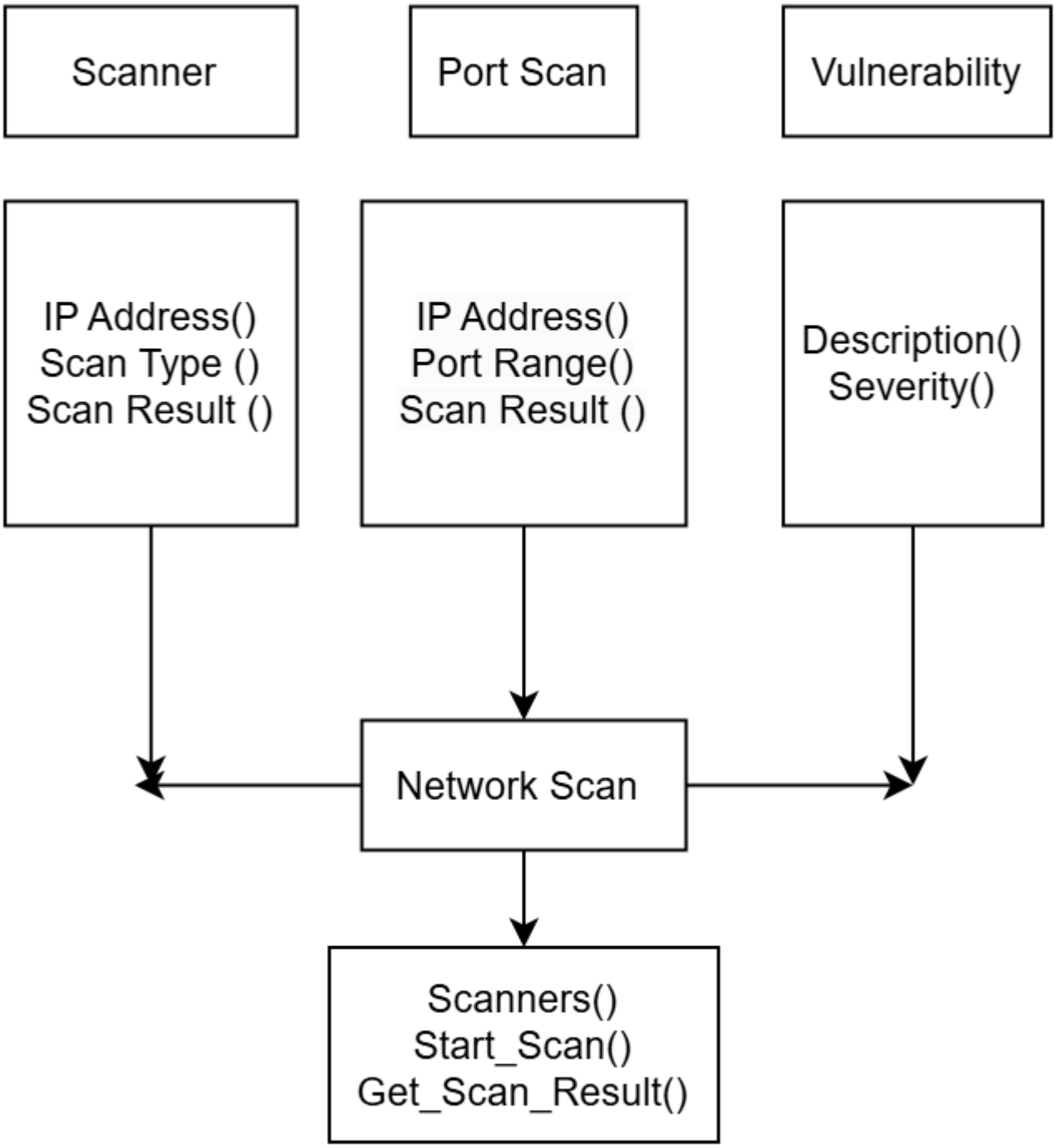

Figure 3.4: UML Diagram

3.4.5 Activity Diagram

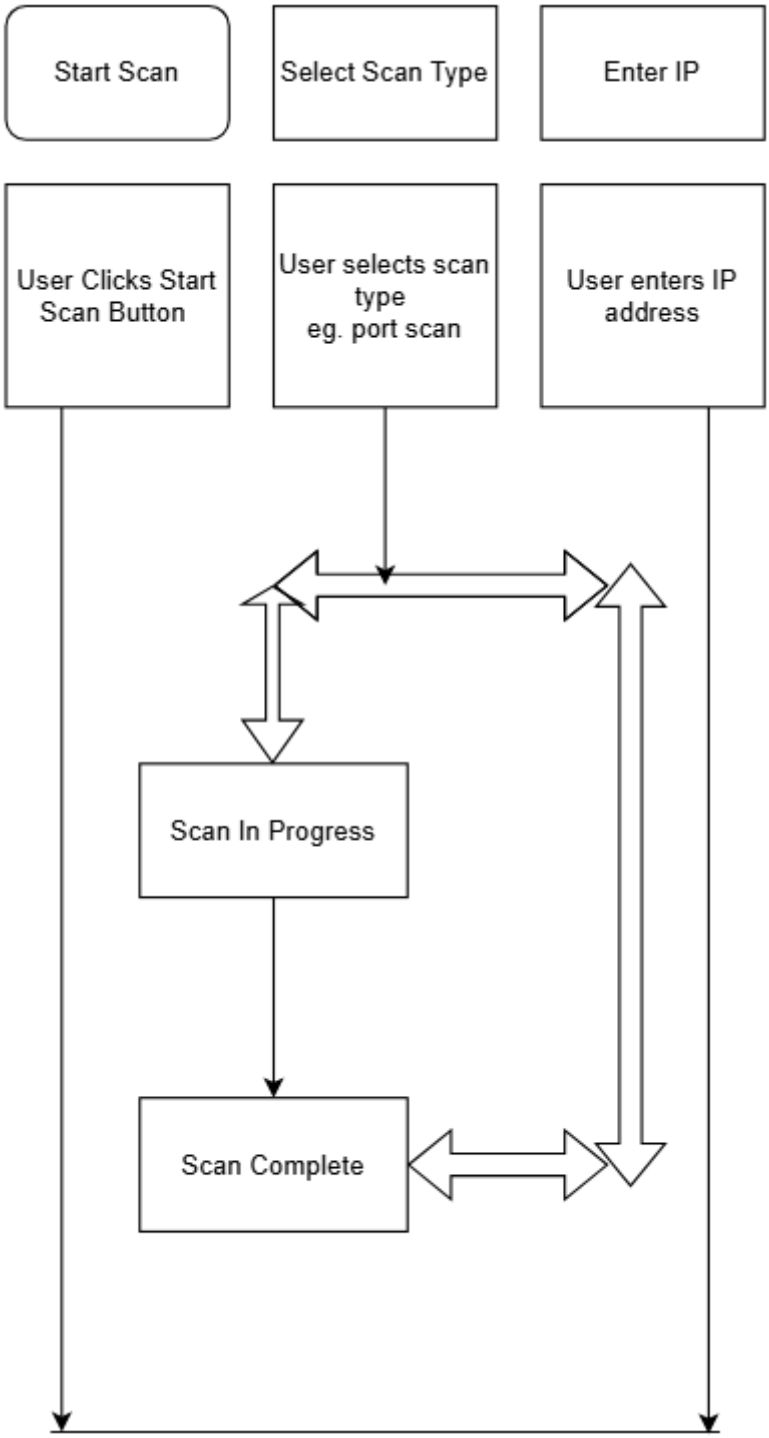

Figure 3.5: Activity Diagram

Chapter 4

System Design

4.1 System Architecture

Overview

Provide a description of the architecture of the Network Scanning Tool, detailing its various components and their interactions. For instance, the tool may include modules for device discovery, port scanning, vulnerability assessment, reporting, and logging.

Monolithic vs. Modular

Discuss the architectural approach of the tool, considering whether it will adopt a monolithic design, where all features are integrated into a single executable file, or a modular design, where functionalities are divided into separate modules or libraries.

4.2 Component Design

Component Breakdown

Provide a detailed description of each component or module within the Network Scanning Tool. For instance, the Device Discovery module may focus on locating devices within the network by employing various protocols, including ICMP, ARP, or SNMP.

Responsibilities

Outline the specific responsibilities assigned to each component or module. For example, the Port Scanning module could be tasked with scanning both TCP and UDP ports on the identified devices to detect open ports and the services running on them.

Interactions

Explain how the different components or modules communicate with each other. For instance, the Vulnerability Assessment module may rely on data from the Port Scanning module to determine

known vulnerabilities linked to the open ports and services identified.

4.3 Data Flow and Processing

Data Flow

Describe the pathway that data takes within the system throughout the scanning procedure. This includes the initial data inputs, such as IP addresses or ranges, the various processing stages, and the ultimate output, such as scan reports.

Processing Algorithms

Provide details about the algorithms utilized for processing the data obtained from scans. For instance, the Vulnerability Assessment component may leverage vulnerability databases or signatures to recognize known vulnerabilities related to open ports and services.

4.4 Scanning Algorithms and Techniques

Device Discovery

Outline the methods and algorithms employed for discovering devices on a network. Techniques may include sending ICMP echo requests (commonly known as pings), utilizing ARP requests, conducting SNMP (Simple Network Management Protocol) polling, and performing DNS (Domain Name System) queries.

Port Scanning

Discuss the various algorithms and techniques applied for scanning ports. This may encompass TCP connect scans, SYN scans, UDP scans, and methods for detecting service versions.

Vulnerability Assessment

Explain the process of assessing vulnerabilities, which may involve comparing identified services and their versions against known vulnerabilities sourced from databases such as CVE (Common Vulnerabilities and Exposures) or NVD (National Vulnerability Database).

4.5 Integration with External Systems

APIs and Protocols

Identify the APIs, protocols, and data formats utilized for integrating the Network Scanning Tool with external systems. For instance, the tool may offer RESTful APIs to facilitate integration with SIEM (Security Information and Event Management) platforms or support SNMP (Simple Network Management Protocol) traps for sending alerts.

Data Exchange

Explain the methods of data exchange between the Network Scanning Tool and external systems. This may involve mechanisms for event notifications, data queries, and response strategies aimed at automating incident management.

4.6 Security Design

Authentication

Describe the methods employed for user authentication within the Network Scanning Tool, such as utilizing username and password combinations or integrating with LDAP (Lightweight Directory Access Protocol) for centralized user management.

Encryption

Provide details on how sensitive information is safeguarded through encryption during both transmission and storage to prevent unauthorized access and protect against eavesdropping.

Access Control

Outline the mechanisms implemented to enforce access control, ensuring that access to critical features or sensitive data is restricted according to user roles and permissions.

4.7 Performance Optimization

Efficient Algorithms

Discuss the implementation of effective algorithms and data structures aimed at enhancing performance. For instance, the tool may incorporate structures such as hash tables or binary search trees to facilitate rapid data retrieval.

Parallel Processing

Explain the approach taken by the tool to leverage multi-threading or asynchronous I/O, allowing scanning tasks to be executed simultaneously, thereby increasing throughput and efficiency.

Resource Management

Detail the strategies employed for managing system resources, including memory and CPU usage, to reduce resource contention and improve overall performance.

4.8 User Interface Design

Layout and Navigation

Explain the arrangement and navigation features of the user interface. For instance, the tool may include a dashboard to showcase scan results along with navigation tabs to access various functionalities.

Visualization

Describe the methods used to present scan results, helping users gain insights. This might involve utilizing charts, graphs, or tables to summarize the findings from scans.

4.9 Error Handling and Recovery

Error Handling Mechanisms

Discuss the approach taken to manage errors, exceptions, and failures within the Network Scanning Tool. This may involve logging errors to a file or database to assist in troubleshooting.

Recovery Procedures

Outline the processes established for recovering from errors or failures. For instance, the tool might implement retry mechanisms for unsuccessful scanning tasks or rollback procedures for database transactions.

4.10 Testing and Quality Assurance

Testing Strategies

Explain the testing strategies employed to verify the reliability and accuracy of the Network Scanning Tool. This may encompass unit testing, integration testing, system testing, and user acceptance testing.

Test Cases

Offer examples of test cases that address various scenarios, including both positive and negative cases for scanning different network configurations.

4.11 Deployment and Maintenance

Deployment Process

Detail the procedure for deploying the Network Scanning Tool in different environments. This could include providing installation scripts or deployment guides to help users set up the tool on various operating systems. Additionally, include configuration instructions to guarantee optimal performance in each specific environment.

Maintenance Procedures

Describe the maintenance processes necessary for the ongoing functionality of the tool. This should cover applying software updates or patches, monitoring the health of the system, and managing user support requests. These procedures are essential for maintaining the tool's performance and reliability over time.

4.12 Documentation

Documentation Requirements

Detail the necessary documentation for the Network Scanning Tool, including user manuals, installation instructions, API documentation, and technical specifications. Each type of documentation should be tailored to meet the needs of different users and stakeholders.

Documentation Templates

Offer templates or guidelines for developing documentation materials, ensuring uniformity and thoroughness in all documentation outputs. These templates should outline the structure, required content, and formatting standards to maintain consistency across all documentation deliverables.

Chapter 5

Methodology

5.1 Project Overview

The project overview outlines the primary goals of the network scanning tool. Depending on its intended purpose, the tool can act as a network security auditing solution, helping to uncover vulnerabilities and security flaws in the network infrastructure. Alternatively, it may operate as a network monitoring application, delivering real-time analysis of traffic patterns and interactions among devices. In troubleshooting contexts, the tool can assist in identifying connectivity issues or performance problems. At its core, raw socket programming provides the foundational functionality by granting direct access to network packets at the protocol level. This low-level access allows for the execution of advanced scanning methods, such as SYN scanning or ICMP probing, which are crucial for thorough network reconnaissance.

5.2 Research and Requirements Gathering

The Npcap/Winpcap library and Windows API are essential components in the tool's development. Npcap/Winpcap enables packet capture by offering a framework to interact with network interfaces and capture traffic efficiently. This allows the tool to monitor network packets and perform detailed analysis. The Windows API provides a connection between the tool and the operating system, facilitating integration with system-level functions. Through this, the tool can handle processes, manage memory, and configure system settings, enhancing its adaptability and effectiveness on the Windows platform.

In the research and requirements gathering phase, an in-depth study of existing network scanning tools, such as Nmap, Wireshark, and Angry IP Scanner, is carried out. This analysis helps in understanding common features, identifying strengths and weaknesses, and gathering insights for

development. Engaging with stakeholders, including network administrators and security experts, is essential to collect specific project requirements. Collaboration helps define the necessary protocols, scanning methods, and critical functionalities. Additionally, research is conducted to address potential limitations and security concerns related to network scanning, such as legal implications, performance impact, and ethical considerations.

5.3 Design

The design phase establishes the core architecture of the tool, outlining its overall structure and how the components will interact. A modular design is employed to ensure scalability and ease of maintenance, allowing for smooth integration of additional features in the future. Critical modules such as packet capture, packet analysis, user interface, and networking utilities are defined according to project needs. Additionally, efficient data structures and algorithms are selected to optimize packet processing, filtering, and analysis, ensuring the tool performs well and uses resources effectively.

5.4 Implementation Strategy

In the implementation strategy phase, the project is broken down into smaller tasks, with timelines set for each to ensure effective progress tracking. Responsibilities are assigned to team members based on their skill sets, promoting teamwork and cohesion. Key milestones are established to highlight major achievements and keep the project on schedule. Comprehensive testing is a vital part of the process, with a detailed plan covering unit, integration, system, and user acceptance testing. Test scenarios are created to account for different network setups, protocols, and edge cases, ensuring the tool performs accurately in practical applications. Validating the tool in real-world network environments further strengthens its reliability and performance.

5.5 Testing and Validation

The testing and validation phase is essential in ensuring the network scanning tool's reliability, accuracy, and overall performance. A well-structured testing plan is developed to assess various aspects of the tool's functionality.

Unit testing focuses on verifying individual components or modules independently, ensuring they function as intended and meet the defined requirements. This involves writing test cases for specific functions, methods, and classes, executing them to check for any errors, and ensuring proper functionality.

Integration testing evaluates how different modules interact when combined, ensuring they work together smoothly. This testing verifies data flow, communication protocols, and error handling between modules, ensuring consistent and correct behavior across the tool.

System testing involves validating the tool as a whole in either a simulated or real-world environment. This includes assessing its interface, networking functions, and data processing capabilities. System test cases are designed to cover various network configurations, protocols, and scenarios to confirm the tool performs reliably under different conditions.

User acceptance testing (UAT) engages real users to assess the tool's usability, effectiveness, and whether it meets their expectations. Feedback from users is gathered to address any usability concerns and guide enhancements in future versions of the tool.

Test scenarios are designed to cover a broad range of network configurations, protocols, and edge cases, ensuring thorough assessment of the tool's features and behavior. This testing ensures the tool can handle diverse network environments, detect different devices and services, and identify potential security vulnerabilities or performance issues.

Validation in a real-world network setting is crucial to assess how the tool performs in actual conditions. This includes testing it in a network environment that mimics real-world traffic and configurations, ensuring the tool meets performance standards and delivers accurate results.

Comprehensive documentation is maintained throughout the testing process, capturing test plans, test cases, results, and any identified issues. This serves as a reference for future development and continuous improvement efforts. Feedback from users, testers, and project sponsors is gathered to help refine the tool, with prioritized feature requests and bug fixes incorporated to ensure ongoing enhancement and alignment with user needs.

5.6 Documentation

Documentation is crucial for recording key design choices, implementation specifics, and instructions for users. A design document outlines the tool's overall architecture, the way components interact, and the reasoning behind significant design decisions, offering valuable context for future development. The implementation document provides in-depth details about the code structure, algorithms, and dependencies, supporting ongoing maintenance and debugging efforts. User documentation offers guidance on installing, setting up, and using the tool, including examples and troubleshooting advice to enhance the user experience.

Input from stakeholders—such as users, testers, and project sponsors—is gathered to pinpoint areas for improvement. This feedback informs ongoing adjustments to the tool's design and

functionality. Addressing feature requests and resolving bugs is prioritized to ensure that the tool evolves continuously in line with user expectations and needs.

5.7 Feedback and Iteration

The feedback and iteration phase is essential for improving the network scanning tool by utilizing insights from stakeholders, users, testers, and project sponsors. This stage involves gathering, analyzing, and applying feedback to enhance the tool's functionality, usability, and overall performance.

Stakeholder engagement begins by reaching out to end-users, network administrators, security experts, and project sponsors. Feedback is collected through various methods such as surveys, interviews, testing sessions, and direct communication. Stakeholders are encouraged to share their experiences, raise concerns about usability, suggest new features, and provide recommendations for improvement.

The tool is refined through an iterative process, where feedback is analyzed to identify areas for enhancement. The most impactful and feasible suggestions are prioritized for implementation. This feedback could include feature requests, bug reports, or ideas for usability improvements. Critical fixes and enhancements are prioritized to maximize the tool's value and effectiveness for users. By assessing the significance of each feedback item, resources are allocated to address the most important changes.

The iterative improvement process involves several cycles of gathering feedback, analyzing it, and making adjustments. Each cycle builds on the previous one, leading to continuous refinement of the tool's design, functionality, and performance.

Consistent communication with stakeholders throughout the feedback and iteration process is key. Keeping them updated on progress, gathering ongoing input, and ensuring alignment with project goals fosters collaboration and promotes ongoing improvements. This open approach encourages stakeholders to remain invested and engaged in the tool's development, driving both innovation and quality enhancement.

Chapter 6

Implementation

6.1 Setup Environment

At the initial stage of implementation, the development environment is established by installing the necessary tools and libraries.The pthread library for multithreading support, Visual Studio or another preferred IDE, and the Windows SDK. Once these installations are complete, configuring environment settings and managing dependencies is crucial to ensure compatibility and streamline the development process.

6.2 Tools

C++ was selected as the primary language for implementing the network scanning tool because of its efficiency and precise control over system resources. The project incorporates the Standard Template Library (STL) to handle containers and algorithms, enabling efficient and structured data management. Multithreading is achieved using the pthread library for Windows, allowing the tool to execute network operations concurrently. Additionally, the Windows SDK is employed to access essential Windows-specific APIs necessary for network communication and scanning functions.

6.3 Libraries

The network scanning tool utilizes the pthread library for Windows to implement multithreading, allowing for concurrent execution of tasks. This library provides a robust threading model, facilitating the efficient management of multiple threads running network scans simultaneously. By employing pthreads, the tool can create and control threads that work independently, significantly enhancing the tool's performance and responsiveness. This parallel execution is essential for processing multiple network requests, reducing scan times, and optimizing efficiency when scanning large networks or multiple IP addresses.

6.4 Flowchart

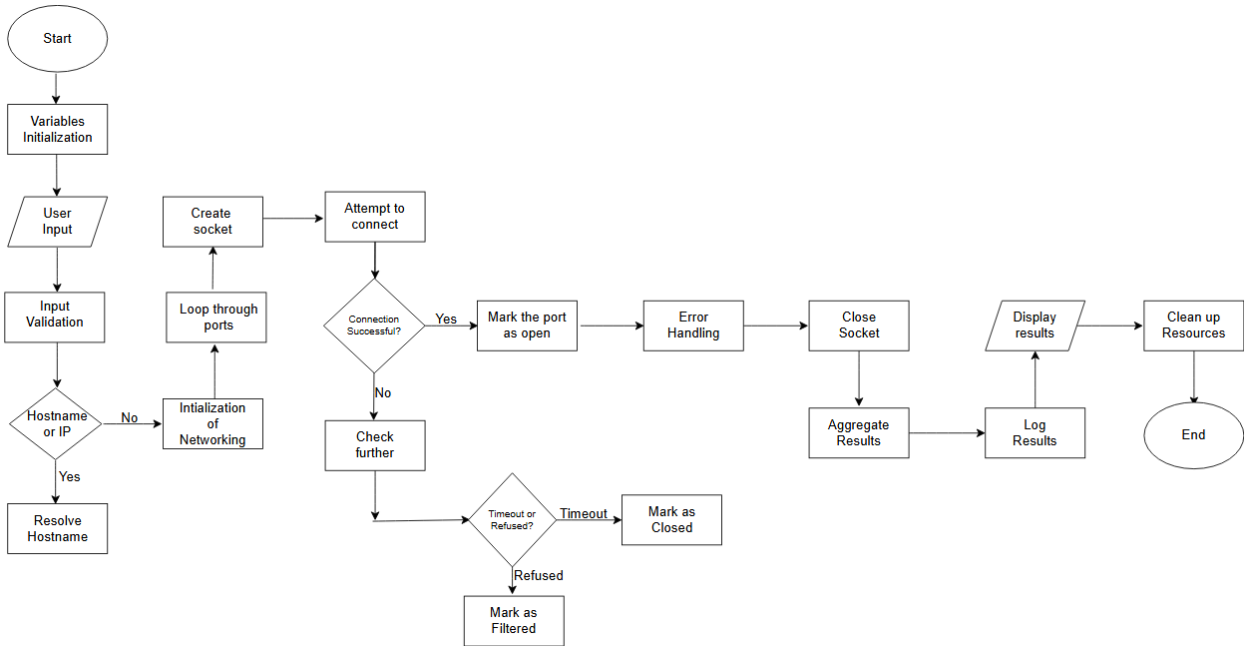

Figure 6.1: Flowchart of the Tool

Chapter 7

Testing

7.1 Phase 1

The ScanQuanta.exe -h command provides essential help or usage information for the Network Scanning Tool. Upon execution, it generates a detailed list of the available options, parameters, and their descriptions, helping users understand how to execute various scanning tasks correctly. The -h flag acts as a reference point, allowing users to quickly review supported features, the syntax structure, and argument options, such as defining IP addresses, port ranges, and scan types. This command is particularly valuable for new users or those seeking clarification on different scan types and tool functionalities. By offering a clear guide, it improves the user experience and ensures accurate use of the tool during scanning operations.

Command- ScanQuanta.exe -h

```
E:\Visual Studio 2022 Project\FINAL PROJECT IMPLEMENTATION\NetProbe\Debug>ScanQuanta.exe -h
Scanning......

./portScanner [option1, ..., optionN]
            --help. Example: "./portScanner --help".
            --ports <ports to scan>. Example: "./portScanner --ports 1,2,3-5".
            --ip <IP address to scan>. Example: "./portScanner --ip 127.0.0.1".
            --prefix <IP prefix to scan>. Example: "./portScanner --prefix 127.143.151.123/24".
            --file <file name containing IP addresses to scan>. Example: "./portScanner --file filename.txt".
            --speedup <parallel threads to use>. Example: "./portScanner --speedup 10".
            --scan <one or more scans>. Example: "./portScanner --scan SYN NULL FIN XMAS".
```

Figure 7.1: Phase 1 Testing

7.2 Phase 2

The command ScanQuanta.exe –ip 127.0.0.1 –ports 80 –scan FIN XMAS demonstrates a targeted network scan aimed at a specific IP address and port using multiple scan techniques. In this case, the tool is instructed to scan the local machine (127.0.0.1) on port 80, which is commonly associated with HTTP services. The –scan option includes two distinct scan types: FIN and XMAS.

- **FIN scan** sends packets with only the FIN flag set, attempting to exploit how some systems handle TCP termination requests, often bypassing firewalls or packet filters.
- **XMAS scan** sends packets with multiple flags set (FIN, PSH, URG), resembling a ”Christmas tree” in terms of flag status, and is designed to detect open or closed ports based on system responses.

Command- ScanQuanta.exe –ip 127.0.0.1 –ports 80 –scan FIN XMAS

```
E:\Visual Studio 2022 Project\FINAL PROJECT IMPLEMENTATION\NetProbe\Debug>ScanQuanta.exe --ip 127.0.0.1 --ports 80 --scan FIN XMAS
Scanning......
Entered createJobQueue() Function
Jobs created: 2
Current SYSTEM IP: 192.168.168.1
Scanning took: 0 seconds
Entering printJobStats() Function
------------------------------------------------------------------ Scanned Results Stats----------------------------------------------------------------

IP Address: 127.0.0.1


Open Ports:
Port   Service Name   Results                                          Version                Conclusion
-------------------------------------------------------------------------------------------------------------------------------------------------


Closed/Filtered/Unfiltered Ports:
Port   Service Name   Results                                          Version                Conclusion
-------------------------------------------------------------------------------------------------------------------------------------------------
80     http           FIN() XMAS()                                     NA                     Filtered
```

Figure 7.2: Phase 2 Testing

Chapter 8

Appendix

8.1 Appendix A: CommonUtilities.h file

```
#pragma once

#ifndef COMMONUTILITIES_H_
#define COMMONUTILITIES_H_

#include <string>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <iphlpapi.h>
#include <iostream>
#include <stdlib.h>
#include <string.h>
#include <sstream>
#include <time.h>
#include "work.h"
#include "pthread.h"

using namespace std;

class CommonUtilities {

public:

    static pthread_mutex_t mutexPoll;
    static pthread_mutex_t mutexPoll2;
```

```
static pthread_mutex_t mutexCreateSocket;

int sniffAPacket(const char* target, const char* port,
string scanType, int protocol, Job* job, SOCKET sockDescProt,
SOCKET sockDescICMP);

static SOCKET createRawSocket(int protocol);

void buildDestIPStruct(struct sockaddr_in* victim, const char* ip,
const char* portNumber);

string getServiceInfo(struct sockaddr_in victim, const char* port);

string probeSSHVersion(struct sockaddr_in victim);

string probeWHOISVersion(struct sockaddr_in victim);

string probeHTTPVersion(struct sockaddr_in victim);

string probePOPVersion(struct sockaddr_in victim);

string probeIMAPVersion(struct sockaddr_in victim);

string probeSMTPVersion(struct sockaddr_in victim);

bool checkIfIPMatch(const char* ip, struct iphdr* ptrToIPHeader);

int lookIntoThePacket(const char* ip, const char*
portNumber, char* ptrToRecievedPacket, string scanType, Job* job);

int parseUDPResponse(const char* ip, const char* portNumber,
unsigned char* ptrToRecievedPacket, Job*);

int parseICMPResponse(const char* ip, const char* portNumber,
unsigned char* sockReadBuffer, Job* job);

int ParseTCPResponse(const char* ip, const char* portNumber,
unsigned char* ptrToRecievedPacket, string scanType, Job* job);
```

```
    SOCKET bindRawSocket(int protocol, struct sockaddr_in* victim,
    const char* ip);
};

#endif /* COMMONUTILITIES_H_ */
```

8.2 Appendix B: DnsHeader.h file

```
#pragma once

#ifndef DNS_HEADER_H_
#define DNS_HEADER_H_

typedef struct
{
 unsigned short id;
 unsigned char rd : 1;
 unsigned char tc : 1;
 unsigned char aa : 1;
 unsigned char opcode : 4;
 unsigned char qr : 1;
 unsigned char rcode : 4;
 unsigned char cd : 1;
 unsigned char ad : 1;
 unsigned char z : 1;
 unsigned char ra : 1;
 unsigned short q_count;
 unsigned short ans_count;
 unsigned short auth_count;
 unsigned short add_count;
} DNS_HEADER;



typedef struct
{
 unsigned short qtype;
 unsigned short qclass;
```

```
} QUESTION;

#endif
```

8.3 Appendix C: IcmpHeader.h file

```
//ICMP Header File

#pragma once
#include <cstdint>

struct icmphdr
{
    uint8_t type;
    uint8_t code;
    uint16_t checksum;
    uint16_t id;
    uint16_t seq;
};
```

8.4 Appendix D: IPHeader.h file

```
//IP Header File

#pragma once
#ifndef IPHEADER
#define IPHEADER

typedef struct iphdr
{
    unsigned char  ip_header_len : 4;
    unsigned char  ip_version : 4;
    unsigned char  ip_tos;
    unsigned short ip_total_length;
    unsigned short ip_id;

    unsigned char  ip_frag_offset : 5;
```

```
    unsigned char  ip_more_fragment : 1;
    unsigned char  ip_dont_fragment : 1;
    unsigned char  ip_reserved_zero : 1;

    unsigned char  ip_frag_offset1;

    unsigned char  ip_ttl;
    unsigned char  protocol;
    unsigned short ip_checksum;
    unsigned int   ip_srcaddr;
    unsigned int   daddr;
} IPV4_HDR, * PIPV4_HDR, *LPIPV4_HDR, IPHeader;

#endif
```

8.5 Appendix E: OptionsClass.h file

```
#pragma once

#include <stdio.h>
#include <iostream>
#include <string>
#include <map>
#include <vector>
#include <algorithm>
#include <iterator>
#include <string.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <list>
#include <sstream>
#include "getopt.h"
#include <errno.h>
#include <fstream>

using namespace std;
```

```
class optionsManager {

 map<string, string> optionDict;

 static optionsManager* m_optManager;

 vector<string> scanList;

 vector<string> portList;

 vector<string> ipList;

public:

 void readOptions(int argc, char* argv[]);

 static optionsManager* Instance();

 string GetStandardUsageOptionScreen();

 map<string, string> getOptionDictionary();

 //void setPeerInfo(int numOfPeers, char* ptrToPeerString);

 //ist<string> getpeerInfoList();

 vector<string> split(string input, char delimiter);

 vector<string> getScanList();

 void unRollPortRange();


 void calculateIPaddresesBitwise(const char* ipWithPrefix);

 void printHostAddresses(unsigned long networkAddress, unsigned
 long broadcastAddress);
```

```
 void processIPFile(string fContent);

 vector<string> getIPList();

 vector<string> getPortList();

 void deleteAllList();

 void deleteSingleTon();
 string ReadIPFile(const char* filename);
};
```

8.6 Appendix F: TCPHeader.h file

```
#pragma once

#ifndef TCPHEADER
#define TCPHEADER

// TCP header
typedef struct tcp_header
{
    unsigned short source_port;   // source port
    unsigned short dest_port;     // destination port
    unsigned int sequence;        // sequence number - 32 bits
    unsigned int acknowledge;     // acknowledgement number - 32 bits

    unsigned char ns : 1;          //Nonce Sum Flag Added in RFC 3540.
    unsigned char reserved_part1 : 3; //according to rfc
    unsigned char data_offset : 4;    /*The number of 32-bit word in the
                                       TCP header.This indicates where
                                       the data begins. The length of
                                       the TCP header is always a
                                       multiple of 32 bits.*/

    unsigned char fin : 1; //Finish Flag
    unsigned char syn : 1; //Synchronise Flag
    unsigned char rst : 1; //Reset Flag
```

```
    unsigned char psh : 1; //Push Flag
    unsigned char ack : 1; //Acknowledgement Flag
    unsigned char urg : 1; //Urgent Flag

    unsigned char ecn : 1; //ECN-Echo Flag
    unsigned char cwr : 1; //Congestion Window Reduced Flag

    ////////////////////////////////

    unsigned short window; // window
    unsigned short checksum; // checksum
    unsigned short urgent_pointer; // urgent pointer
} TCP_HDR, * PTCP_HDR, *LPTCP_HDR , TCPHeader, TCP_HEADER;

#endif // !TCPHEADER
```

8.7 Appendix G: TCPClass.h file

```
#pragma once
/*
 * TCPUtilities.h
 */

#ifndef TCPUTILITIES_H_
#define TCPUTILITIES_H_

#include <string>
#include <string.h>
#include <stdio.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <mstcpip.h>
#include <iostream>
#include <sstream>
#include <process.h> // For _beginthreadex
#include <errno.h>
#include "CommonUtilities.h"
#include "work.h"
```

```
#define PACKET_LENGTH 2048

using namespace std;

class TCPUtilities
{
    //"comUtil" Object of Class "CommonUtilities"
    CommonUtilities comUtil;

    HANDLE createPacketLock = CreateMutex(NULL, FALSE, NULL);
    // Windows equivalent for pthread_mutex_t

public:

    //Default Constructor
    TCPUtilities();

    unsigned short csum(uint8_t* data, int length);

    //CheckSum Calculator
    uint16_t calculateCheckSum(uint32_t ipSource, uint32_t ipDest,
    uint8_t protocol, uint16_t tcpLength, struct tcp_header tcpSegment);

    //Packet Creation
    void createPacket(string scanType, const char* destIP,
    const char* portNumber, char*, char*);

    //TCP Header Creater
    void createTCPHeader(struct tcp_header* tcpHeader, int sourcePort,
    const char* destPort, string scanType);

    //Send TCP Packet
    void sendTCPPacket(Job* job, char*);
};

#endif /* TCPUTILITIES_H_ */
```

8.8 Appendix H: UDPHeader.h file

```
//UDP Header

#pragma once
#include <cstdint>

struct udphdr {
    uint16_t source;
    uint16_t dest;
    uint16_t length;
    uint16_t checksum;
};
```

8.9 Appendix I: UDPClass.h file

```
#pragma once
/*
 * UDPUtilities.h
 */

#ifndef UDPUTILITIES_H_
#define UDPUTILITIES_H_

#include <iostream>
#include <string.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <vector>
#include "CommonUtilities.h"
#include "work.h"

#define PACKET_LENGTH 2048

using namespace std;

class UDPUtilities
{
```

```
        //"comUtil" Object of Class "CommonUtilities"
        CommonUtilities comUtil;

    public:
        //Creating UDP Header Content
        void createUDPHeader(struct udphdr* udpHeader, int sourcePort,
        const char* destPort);

        //Creating DNS Header Content
        void createDNSPacket(char* ipAddress, char* packet);

        void convertToDNSNameFormat(unsigned char* dnsHeader,
        char* destinationHost);

        //Fills in the UDP Packet
        int createPacketUDP(int sourcePort, const char* destPort,
        char* destIpAddress, char* packet);

        //Send the UDP Packet
        void sendUDPPacket(Job* job);
};

#endif /* UDPUTILITIES_H_ */
```

8.10 Appendix J: Work.h file

```
#pragma once

#ifndef JOB_H_
#define JOB_H_
#include <string>

using namespace std;

enum Status
{
 ASSIGNED,
 INPROGESS,
```

```
 COMPLETED,
 NOTNOW
};

class Job
{

 public:

  string scanType;
  string port;
  string IP;
  Status jobStatus;
  string conclusion;
  string serviceName;
  string serviceVersion;
  string scanResult;

  Job();

  Job(string, string, string);

  void* (*funcPointer)(void*);

  Job* args;

  void setJob(void* (*funcPointer)(void*));

  void execute();

  ~Job();
};

#endif
```

8.11 Appendix K: CommonUtilities.cpp file

```
#include "CommonUtilities.h"
```

```
#include "tcp_header.h"
#include "ip_header.h"
#include "icmp_header.h"
#include "udp_header.h"
#include "work.h"
#include <winsock2.h>
#include <ws2tcpip.h>
#include <windows.h>
#include <iphlpapi.h>

int CommonUtilities::sniffAPacket(const char* target, const char*
portNumber, string scanType, int protocol, Job* job, SOCKET
sockDescProt, SOCKET sockDescICMP) {

 int status = -1;
 u_long mode = 1;

 ioctlsocket(sockDescProt, FIONBIO, &mode);
 ioctlsocket(sockDescICMP, FIONBIO, &mode);

 struct pollfd fileDesc[2];
 struct sockaddr_in recievedIPStruct;

 memset(&recievedIPStruct, 0, sizeof(recievedIPStruct));

 fileDesc[0].fd = sockDescProt; fileDesc[0].events = POLLIN;
 fileDesc[1].fd = sockDescICMP; fileDesc[1].events = POLLIN;

 int pollStat = WSAPoll(fileDesc, 2, 4000);
 int packetRecievedType = -1, recievedSize = -1; int supposedToBeRecievedPacket = -1;

 socklen_t size = sizeof(recievedIPStruct);

 const int MAX_RECIEVED_PACKET_LENGTH = 200;
 char sockReadBuffer[MAX_RECIEVED_PACKET_LENGTH];

 memset(sockReadBuffer, '\0', MAX_RECIEVED_PACKET_LENGTH);
```

```
 time_t startTime = time(0);
 double timeout = 4;

 while (pollStat == 1) {

  time_t current = time(0);
  double timeElapsed = difftime(current, startTime);

  if (timeElapsed > timeout) {
   break;
  }
  if (fileDesc[0].revents & POLLIN) {
  recievedSize = recvfrom(sockDescProt, sockReadBuffer,
  MAX_RECIEVED_PACKET_LENGTH, 0, (sockaddr*)&
  recievedIPStruct, &size);
  }
  if (fileDesc[1].revents & POLLIN) {
  recievedSize = recvfrom(sockDescICMP, sockReadBuffer,
 MAX_RECIEVED_PACKET_LENGTH, 0,
  (sockaddr*)&recievedIPStruct, &size);
  }
  if (recievedSize > 0) {
  status = lookIntoThePacket(target, portNumber,
  sockReadBuffer, scanType, job);
   if (status >= 0)
    break;
  }
 }

 return status;
}

SOCKET CommonUtilities::createRawSocket(int protocol)
{
 SOCKET sockfd = INVALID_SOCKET;
 while (sockfd == INVALID_SOCKET)
  sockfd = socket(AF_INET, SOCK_RAW, protocol);
 if (sockfd != INVALID_SOCKET) {
```

```
  BOOL optval = TRUE;
  setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, (char*)&optval, sizeof(optval));
 }
 return sockfd;
}

int CommonUtilities::lookIntoThePacket(const char* ip,
const char* portNumber, char* sockReadBuffer,
string scanType, Job* job)
{
 int status = -1;

 struct iphdr* ptrToIPHeader = NULL;
 struct tcp_header* ptrToTCPHeader = NULL;
 struct sockaddr_in ipSource {};
 struct servent* ptrToserviceInfo = NULL;

 unsigned char* ptrToRecievedPacket = NULL;

 ptrToRecievedPacket = (unsigned char*)sockReadBuffer;
 ptrToIPHeader = (struct iphdr*)ptrToRecievedPacket;
 ptrToRecievedPacket += sizeof(iphdr);

 if (checkIfIPMatch(ip, ptrToIPHeader)) {

  if (ptrToIPHeader->protocol == IPPROTO_TCP)
   status = ParseTCPResponse(ip, portNumber, ptrToRecievedPacket, scanType, job);
  else if (ptrToIPHeader->protocol == IPPROTO_UDP)
   status = parseUDPResponse(ip, portNumber, ptrToRecievedPacket, job);
  else if (ptrToIPHeader->protocol == IPPROTO_ICMP)
   status = parseICMPResponse(ip, portNumber, ptrToRecievedPacket, job);
 }
 else if (ptrToIPHeader->protocol == IPPROTO_ICMP)
  status = parseICMPResponse(ip, portNumber, ptrToRecievedPacket, job);

 return status;
}
```

```
bool CommonUtilities::checkIfIPMatch(const char* ip, struct iphdr* ptrToIPHeader)
{
 struct sockaddr_in ipSource;
 memset(&ipSource, 0, sizeof(ipSource));
 ipSource.sin_addr.s_addr = ptrToIPHeader->ip_srcaddr;

 if (strcmp(ip, inet_ntoa(ipSource.sin_addr)) == 0) {
  return true;
 }
 return false;
}

int CommonUtilities::parseUDPResponse(const char* ip,
const char* portNumber, unsigned char* ptrToRecievedPacket,
Job* job) {
 int status = -1;
 struct udphdr* udpHeader = NULL;
 udpHeader = (struct udphdr*)ptrToRecievedPacket;
 if (atoi(portNumber) == ntohs(udpHeader->source)) {
  job->scanResult = "Open";
  status = 0;
 }
 return status;
}

int CommonUtilities::ParseTCPResponse(const char* ip, const char*
portNumber, unsigned char* ptrToRecievedPacket, string scanType,
Job* job)
{
 int status = -1;

 struct tcp_header* ptrToTCPHeader = NULL;
 struct servent* ptrToserviceInfo = NULL;

 ptrToTCPHeader = (struct tcp_header*)ptrToRecievedPacket;
 ptrToRecievedPacket += ptrToTCPHeader->data_offset * 4;

 if (atoi(portNumber) == ntohs(ptrToTCPHeader->source_port)) {
```

```
  if (scanType == "SYN") {

   if (ptrToTCPHeader->rst == 1) {
    job->scanResult = "Closed";
    status = 1;
   }
   if (ptrToTCPHeader->syn == 1 && ptrToTCPHeader->ack == 1) {
    job->scanResult = "Open";
    status = 0;
   }
  }
  else if (scanType == "ACK") {
   if (ptrToTCPHeader->rst == 1) {
    job->scanResult = "Unfiltered";
    status = 1;
   }
  }
  else if (scanType == "NULL" || scanType == "XMAS" || scanType == "FIN") {
   if (ptrToTCPHeader->rst == 1) {
    job->scanResult = "Closed";
    status = 1;
   }
  }
 }
 return status;
}

int CommonUtilities::parseICMPResponse(const char* ip,
const char* portNumber, unsigned char* ptrToPacketData,
Job* job)
{
 struct sockaddr_in ipDest;
 memset(&ipDest, 0, sizeof(ipDest));
 int status = -1;
 bool flag = true;
 struct icmphdr* icmpPtr = (struct icmphdr*)ptrToPacketData;
 ptrToPacketData += sizeof(struct icmphdr);
```

```
 struct iphdr* ipHeader = (struct iphdr*)ptrToPacketData;
 ptrToPacketData += sizeof(struct iphdr);
 ipDest.sin_addr.s_addr = ipHeader->daddr;
 if (strcmp(inet_ntoa(ipDest.sin_addr), ip) == 0)
 {
  if (ipHeader->protocol == IPPROTO_TCP)
  {
   struct tcp_header* tcpHeader = (struct tcp_header*)ptrToPacketData;
   if (atoi(portNumber) == ntohs(tcpHeader->dest_port))
    status = 1;
  }
  else if (ipHeader->protocol == IPPROTO_UDP)
  {
   struct udphdr* udpHeader = (struct udphdr*)ptrToPacketData;
   if (atoi(portNumber) == ntohs(udpHeader->dest)) {
    status = 1;
    flag = false;
   }
  }
  if (status == 1)
  {
   if (flag && icmpPtr->type == 3 && (icmpPtr->code == 1 || icmpPtr-
>code == 2 || icmpPtr->code == 3 || icmpPtr->code == 9 || icmpPtr-
>code == 10 || icmpPtr->code == 13))
    job->scanResult = "Filtered";
   else if (!flag && icmpPtr->type == 3 && (icmpPtr->code == 1 || icmpPtr-
>code == 2 || icmpPtr->code == 9 || icmpPtr->code == 10 || icmpPtr->code == 13))
    job->scanResult = "Filtered";
   else if (!flag && icmpPtr->type == 3 && icmpPtr->code == 3)
    job->scanResult = "Closed";
  }
 }
 return status;
}

string CommonUtilities::probeHTTPVersion(sockaddr_in victim)
{
 int newSock;
```

```
string getRequest;
stringstream ss;

int sentBytes, recievedSize = -1, versionLen;
char sockReadBuffer[100];

memset(sockReadBuffer, '\0', sizeof(sockReadBuffer));
ss << "GET / HTTP/1.1 \r\nHost: " << inet_ntoa(victim.sin_addr)
<< "\r\nConnection: close\r\n\r\n";

getRequest = ss.str();
string stringedData;
size_t bytesToRead{}, bytesRead{};

struct timeval timeout;
fd_set fileDesc;

memset(&timeout, 0, sizeof(timeout));
newSock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

int status = connect(newSock, (struct sockaddr*)&victim, sizeof(victim));
if (status == -1)
{
 closesocket(newSock);
 return "No response";
}

timeout.tv_sec = 10; timeout.tv_usec = 0;

FD_ZERO(&fileDesc);
FD_SET(newSock, &fileDesc);

sentBytes = send(newSock, getRequest.c_str(), getRequest.length(), 0);
bytesToRead = sizeof(sockReadBuffer) - 1;

status = 0;
```

```
 while (status < bytesToRead)
 {
  status = select(newSock + 1, &fileDesc, NULL, NULL, &timeout);
  if (status > 0)
   status = recv(newSock, &sockReadBuffer[status], bytesToRead - status, 0);
 }
 closesocket(newSock);

 stringedData = sockReadBuffer;
 versionLen = stringedData.find("HTTP/1.1");

 if (versionLen != string::npos)
 {
  versionLen = stringedData.find("\r\n", versionLen);
  if (versionLen != string::npos)
   return stringedData.substr(0, versionLen);
 }
 return "No response";
}

string CommonUtilities::probeSMTPVersion(sockaddr_in victim) {

 char smtpRequest[10] = "EHLO\n";
 char sockReadBuffer[1000];

 int recievedSize = -1, sentBytes{}, versionLen = 1000;

 size_t pos{}, pos1{};

 memset(sockReadBuffer, '\0', sizeof(sockReadBuffer));

 int newSock;
 string stringedData;

 newSock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

 if (connect(newSock, (struct sockaddr*)&victim, sizeof(victim)) == 0) {
  recievedSize = recv(newSock, sockReadBuffer, 2048, 0);
```

```
  if (recievedSize < 0)
   return string("ERROR");
  else {
   stringedData = string(sockReadBuffer);
   if ((pos = stringedData.find("220")) != string::npos) {
    versionLen = stringedData.length() - pos;
    const int tup = 10000;
    char temp[tup];
    stringedData.copy(temp, versionLen, pos + strlen("220"));
    temp[versionLen] = '\0';
    stringedData = string(temp);
   }
  }
 }
 return stringedData;
}

string CommonUtilities::getServiceInfo(struct sockaddr_in victim,
const char* port) {

 string versionInfo;

 switch (atoi(port)) {
 case 22: versionInfo = probeSSHVersion(victim); break;
 case 43: versionInfo = probeWHOISVersion(victim); break;
 case 80: versionInfo = probeHTTPVersion(victim); break;
 case 110: versionInfo = probePOPVersion(victim); break;
 case 143: versionInfo = probeIMAPVersion(victim); break;
 case 587: versionInfo = probeSMTPVersion(victim); break;
 }
 return versionInfo;
}

SOCKET CommonUtilities::bindRawSocket(int protocol,
struct sockaddr_in* victim, const char* ip)
{
 SOCKET sock = socket(AF_INET, SOCK_RAW, protocol);
```

```
 struct timeval timeout;
 timeout.tv_sec = 10; timeout.tv_usec = 0;

 memset(&timeout, 0, sizeof(timeout));

 if (sock == -1)
  return -1;
 if (protocol == IPPROTO_TCP)
  if (setsockopt(sock, IPPROTO_IP, IP_HDRINCL,
  (const char*)&timeout, sizeof(timeout)) == -1)
   return -1;
 if (protocol == IPPROTO_ICMP)
  if (setsockopt(sock, SOL_SOCKET, SO_RCVTIMEO,
  (const char*)&timeout, sizeof(timeout)) == -1)
   return -1;
 if (protocol == IPPROTO_UDP)
  if (setsockopt(sock, SOL_SOCKET, SO_SNDTIMEO,
  (const char*)&timeout, sizeof(timeout)) == -1)
   return -1;
 memset(victim, 0, sizeof(struct sockaddr_in));
 victim->sin_family = AF_INET;
 victim->sin_addr.s_addr = inet_addr(ip);
 return sock;
}

void CommonUtilities::buildDestIPStruct(struct sockaddr_in* victim,
const char* ip, const char* portNumber) {

 victim->sin_family = AF_INET;
 victim->sin_port = htons(atoi(portNumber));
 victim->sin_addr.s_addr = inet_addr(ip);
}

string CommonUtilities::probeSSHVersion(sockaddr_in victim) {
 char sockReadBuffer[50];
 int recievedSize = -1;

 memset(sockReadBuffer, '\0', 50);
```

```
 int newSock;
 string stringedData;

 newSock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

 if (connect(newSock, (struct sockaddr*)&victim, sizeof(victim)) == 0) {
  recievedSize = recv(newSock, sockReadBuffer, 50, 0);
  if (recievedSize < 0)
   return string("ERROR");
  else {
   stringedData = string(sockReadBuffer);
  }
 }
 return stringedData;
}

string CommonUtilities::probeWHOISVersion(sockaddr_in victim) {

 char sockReadBuffer[512];
 memset(sockReadBuffer, '\0', 512);

 int recievedSize = -1, sentBytes{}, versionLen{} ;
 size_t pos{}, pos1{};

 string stringedData;

 int newSock;

 memset(sockReadBuffer, '\0', sizeof(sockReadBuffer));
 newSock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

 if (connect(newSock, (struct sockaddr*)&victim, sizeof(victim)) == 0) {
  recievedSize = recv(newSock, sockReadBuffer, 511, 0);
  if (recievedSize < 0)
   return string("ERROR");
  else {
   stringedData = string(sockReadBuffer);
```

```
   if ((pos = stringedData.find("Version")) != string::npos) {
    versionLen = pos1 - (pos + strlen("Version "));
    char temp[7];
    memset(temp, '\0', 7);
    stringedData.copy(temp, 6, pos + strlen("Version "));
    stringedData = string(temp);
   }
  }
 }
 return stringedData;
}

string CommonUtilities::probePOPVersion(sockaddr_in victim) {
 char popRequest[10] = "ABCD";
 char sockReadBuffer[100];
 int recievedSize = -1, sentBytes = 0, versionLen;
 size_t pos, pos1;
 memset(sockReadBuffer, '\0', sizeof(sockReadBuffer));
 int newSock; string stringedData;
 newSock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
 if (connect(newSock, (struct sockaddr*)&victim, sizeof(victim)) == 0) {

  sentBytes = send(newSock, popRequest, 22, 0);
  recievedSize = recv(newSock, sockReadBuffer, 100, 0);
  stringedData = string(sockReadBuffer);
  if (recievedSize < 0)
   return string("ERROR");
  else {
   if ((pos = stringedData.find("+OK")) != string::npos) {
    if ((pos1 = stringedData.find("ready")) != string::npos) {
     versionLen = pos1 - (pos + strlen("+OK "));
     const int tup = 10000;
     char temp[tup];
     stringedData.copy(temp, versionLen, pos + strlen("+OK "));
     temp[versionLen] = '\0';
     stringedData = string(temp);
    }
```

```
   }
  }
 }
 return stringedData;
}

string CommonUtilities::probeIMAPVersion(sockaddr_in victim) {

 char imapRequest[10] = "\r\n";
 char sockReadBuffer[2048];

 int recievedSize = -1, sentBytes{}, versionLen{} ;
 size_t pos, pos1;
 int newSock;

 string stringedData;
 memset(sockReadBuffer, '\0', sizeof(sockReadBuffer));

 newSock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

 if (connect(newSock, (struct sockaddr*)&victim, sizeof(victim)) == 0) {
  recievedSize = recv(newSock, sockReadBuffer, 2048, 0);
  if (recievedSize < 0)
   return string("ERROR");
  else {
   stringedData = string(sockReadBuffer);
   if ((pos = stringedData.find("]")) != string::npos) {
    if ((pos1 = stringedData.find("ready")) != string::npos) {
     versionLen = pos1 - (pos + strlen("] "));
     const int tup = 10000;
     char temp[tup];
     stringedData.copy(temp, versionLen, pos + strlen("] "));
     temp[versionLen] = '\0';
     stringedData = string(temp);
    }
   }

  }
```

```
 }
 return stringedData;
}
```

8.12 Appendix L: NetProbe.cpp file

```
//Entry Point of the Main File

#include <stdio.h>
#include <errno.h>
#include <pthread.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <iphlpapi.h>
#include <iostream>
#include <string.h>
#include <vector>
#include <map>
#include <math.h>
#include <windows.h>
#include "optionsClass.h"
#include "tcpClass.h"
#include "udpClass.h"
#include "work.h"
#include <iomanip>

#define PACKET_LENGTH 2048

#pragma comment(lib, "Ws2_32.lib")
#pragma comment(lib, "Iphlpapi.lib")

using namespace std;



vector<Job*> jobQueue;
map<string, bool> activeJobs;
typedef map<string, vector<Job*>> innerMap;
map<string, innerMap> reportMap;
```

```
pthread_mutex_t perJob = PTHREAD_MUTEX_INITIALIZER, perActiveJob = PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_t jobindex = PTHREAD_MUTEX_INITIALIZER;
int maxJobSize = 0; int jobsTaken = 0; size_t maxJobId = 0;


string getService(const char* protocol, const char* portNumber) {

 string serviceName = "NA";
 struct servent* serviceInfo;
 serviceInfo = getservbyport(htons(atoi(portNumber)), protocol);
 if (serviceInfo != NULL)
  serviceName = string(serviceInfo->s_name);
 return serviceName;
}


void getCurrentSystemIP(char* ip) {
 PIP_ADAPTER_INFO AdapterInfo;
 DWORD dwBufLen = sizeof(AdapterInfo);
 char* ipAddr = nullptr;
 AdapterInfo = (IP_ADAPTER_INFO*)malloc(sizeof(IP_ADAPTER_INFO));
 if (AdapterInfo == NULL) {
  printf("Error allocating memory needed to call GetAdaptersinfo\n");
  return;
 }

 if (GetAdaptersInfo(AdapterInfo, &dwBufLen) == ERROR_BUFFER_OVERFLOW) {
  AdapterInfo = (IP_ADAPTER_INFO*)malloc(dwBufLen);
  if (AdapterInfo == NULL) {
   printf("Error allocating memory needed to call GetAdaptersinfo\n");
   return;
  }
 }

 if (GetAdaptersInfo(AdapterInfo, &dwBufLen) == NO_ERROR) {
  PIP_ADAPTER_INFO pAdapterInfo = AdapterInfo;
  while (pAdapterInfo) {
```

```
   if (pAdapterInfo->Type == MIB_IF_TYPE_ETHERNET || pAdapterInfo-
>Type == IF_TYPE_IEEE80211) {
    ipAddr = pAdapterInfo->IpAddressList.IpAddress.String;
    if (ipAddr && strcmp(ipAddr, "0.0.0.0") != 0) {
     strcpy(ip, ipAddr);
     printf("Current SYSTEM IP: %s\n", ip);
     break;
    }
   }
   pAdapterInfo = pAdapterInfo->Next;
  }
 }
 if (AdapterInfo)
  free(AdapterInfo);
}
bool checkIfActiveJobWithSameIPandPort(Job* job) {

 if (!activeJobs.empty() && job != NULL) {
  if (activeJobs.find(job->IP + job->port) != activeJobs.end()) {
   return false;
  }
 }
 return true;
}
string conclude(int inputArray[5]) {
 int big = 0, iOfBig{};
 string conclusion;
 for (int i = 0; i < 4; i++) {
  if (inputArray[i] > big) {
   big = inputArray[i];
   iOfBig = i;
  }
 }
 switch (iOfBig) {
  cout << "inside switch";
 case 0: conclusion = "Filtered"; break;
 case 1: conclusion = "Open|Filtered"; break;
 case 2: conclusion = "Unfiltered"; break;
```

```
 case 3: conclusion = "Closed"; break;
 case 4: conclusion = "Open"; break;
 }
 return conclusion;
}

void printJobStats() {

 std::cout << "Entering printJobStats() Function" << '\n';

 map<string, innerMap>::iterator reportMapItr;
 map<string, vector<Job*>>::iterator innerMapItr;
 vector<Job*> jobList, openList, closedList;
 vector<Job*>::iterator jobListIter, try1, openListIter, closedListIter;
 reportMapItr = reportMap.begin();
 innerMap tempIm; string tempScanList;
 int openfiltered = 0, unfiltered = 0, filtered = 0, open = 0, closed = 0;
 int portConclusion[5]; string protocolType, serviceName;
 memset(&portConclusion, 0, sizeof(portConclusion));
 vector<string> tempScanResult;
 cout << "------------------------------------------------------------------
Scanned Results Stats--------------------------------------------------------------------
" << endl;
 while (reportMapItr != reportMap.end()) {
  cout << "" << endl;
  cout << "IP Address: " << reportMapItr->first << endl;
  tempIm = reportMapItr->second;
  innerMapItr = tempIm.begin();
  openList.clear(); closedList.clear();
  while (innerMapItr != tempIm.end()) {
   tempScanList.clear();
   jobList = innerMapItr->second;
   try1 = jobListIter = jobList.begin();
   string Conclusion = "Unknown";
   while (jobListIter != jobList.end()) {
    tempScanList.append((*jobListIter)->scanType);
    tempScanList.append("(");
    tempScanList.append((*jobListIter)->scanResult);
```

```
    tempScanList.append(") ");
    if ((((*jobListIter)->scanType == "SYN" && (*jobListIter)-
>scanResult == "Open") || (((*jobListIter)->scanType == "UDP") && (*jobListIter)-
>scanResult == "Open")))
     Conclusion = "Open";
    else if ((*jobListIter)->scanType == "SYN" && (*jobListIter)-
>scanResult == "Closed")
     Conclusion = "Closed";
    else if ((*jobListIter)->scanResult == "Filtered")
     portConclusion[0] = ++filtered;
    else if ((*jobListIter)->scanResult == "Open|Filtered")
     portConclusion[1] = ++openfiltered;
    else if ((*jobListIter)->scanResult == "Unfiltered")
     portConclusion[2] = ++unfiltered;
    else if ((*jobListIter)->scanResult == "Closed")
     portConclusion[3] = ++closed;
    else if ((*jobListIter)->scanResult == "Open")
     portConclusion[4] = ++open;
    jobListIter++;
   }
   (*try1)->scanResult = tempScanList;
   if (Conclusion == "Unknown")
    Conclusion = conclude(portConclusion);
   (*try1)->conclusion = Conclusion;
   if ((*try1)->conclusion == "Open")
    openList.push_back(*try1);
   else
    closedList.push_back(*try1);
   memset(&portConclusion, 0, sizeof(portConclusion));
   openfiltered = 0; unfiltered = 0; filtered = 0; open = 0; closed = 0;
   innerMapItr++;
  }
  cout << endl << endl;
  cout << "Open Ports: " << endl;
  cout << left << setw(7) << "Port" << left << setw(15)
  <<"Service Name" << left << setw(50) << "Results" << left
  << setw(25) << "Version" << setw(10) << "Conclusion" << endl;
  cout << "-------------------------------------------------------------------------------
```

```
----------------------------------------------------------------------" << endl;
  if (openList.size() > 0) {
   openListIter = openList.begin();
   while (openListIter != openList.end()) {
    if ((*openListIter)->scanType == "UDP")
     protocolType = "udp";
    else
     protocolType = "tcp";
    serviceName = getService(protocolType.c_str(), ((*openListIter)->port).c_str());
    cout << left << setw(7) << (*openListIter)->port
    << left << setw(15) << serviceName << left << setw(50) << (*openListIter)-
>scanResult << left << setw(25) << (*openListIter)-
>serviceVersion << setw(10) << (*openListIter)->conclusion << endl;
    openListIter++;
   }
  }
  cout << endl << endl;
  cout << "Closed/Filtered/Unfiltered Ports: " << endl;
  cout << left << setw(7) << "Port" << left << setw(15)
  s<<"Service Name" << left << setw(50) << "Results" << left <<
  setw(25) << "Version" << setw(10) << "Conclusion" << endl;
  cout << "-------------------------------------------------------------------------------
-------------------------------------------------------------" << endl;
  if (closedList.size() > 0) {
   closedListIter = closedList.begin();
   while (closedListIter != closedList.end()) {
    if ((*closedListIter)->scanType == "UDP")
     protocolType = "udp";
    else
     protocolType = "tcp";
    serviceName = getService(protocolType.c_str(), ((*closedListIter)->port).c_str());
    cout << left << setw(7) << (*closedListIter)-
>port << left << setw(15) << serviceName << left << setw(50) << (*closedListIter)-
>scanResult << left << setw(25) << (*closedListIter)-
>serviceVersion << setw(10) << (*closedListIter)->conclusion << endl;
    closedListIter++;
   }
  }
```

```
  reportMapItr++;
 }
}
void reportCompletedJob(Job* job) {
 innerMap portMap;
 map<string, vector<Job*>>::iterator innerMapItr;
 vector<Job*> tempJobs;
 auto ipvalue = reportMap.find(job->IP);
 if (ipvalue != reportMap.end()) {
  portMap = ipvalue->second;
  auto portvalue = portMap.find(job->port);
  if (portvalue != portMap.end()) {
   tempJobs = portvalue->second;
   tempJobs.push_back(job);
   portMap.erase(portvalue);
   portMap.insert(pair<string, vector<Job*>>{job->port, tempJobs});
  }
  else {
   tempJobs.push_back(job);
   portMap.insert(pair<string, vector<Job*>>{job->port, tempJobs});
  }
  reportMap.erase(ipvalue);
  reportMap.insert(pair<string, innerMap>{job->IP, portMap});
 }
 else
 {
  tempJobs.push_back(job);
  portMap.insert(pair<string, vector<Job*>>{job->port, tempJobs});
  reportMap.insert(pair<string, innerMap>{job->IP, portMap});
 }
}

void* sendPacket(void* message) {

 TCPUtilities tcpUtil;
 UDPUtilities udpUtil;
 Job* job;
 int returnValue{};
```

```
char* ip = (char*)message;

while (true) {

 pthread_mutex_lock(&jobindex);
 if (maxJobId < jobQueue.size()) {
  job = jobQueue.at(maxJobId);
  maxJobId++;
  if (!checkIfActiveJobWithSameIPandPort(job)) {
   --maxJobId;
   job->jobStatus = NOTNOW;
  }
  else {
   job->jobStatus = ASSIGNED;
   activeJobs.insert(make_pair(job->IP + job->port, true));
  }
 }
 else {
  pthread_mutex_unlock(&jobindex);
  break;
 }
 pthread_mutex_unlock(&jobindex);
 if (job->jobStatus != NOTNOW) {

  if (job->scanType.compare("UDP") == 0)
   udpUtil.sendUDPPacket(job);
  else
   tcpUtil.sendTCPPacket(job, ip);

  pthread_mutex_lock(&perActiveJob);
  if (job->jobStatus == COMPLETED) {
   auto value = activeJobs.find(job->IP + job->port);
   if (value->second) {
    activeJobs.erase(value->first);
    reportCompletedJob(job);
   }
  }
  pthread_mutex_unlock(&perActiveJob);
```

```
  }

 }
 return NULL;
}

pthread_t createThreads(int threadCount)
{
 vector<pthread_t> threads(threadCount);
 int createStatus;
 pthread_t thread;

 for (int i = 0; i < threadCount; i++) {
  createStatus = pthread_create(&threads[i], NULL, sendPacket, (void*)NULL);
  if (createStatus != 0) {
   cout << "Create thread failed" << endl;
  }
  else {
   cout << "Thread " << i << " created successfully." << endl;
  }
  thread = threads[i];
 }
 return thread;
}

void destroyJobQueue() {

 for (vector<Job*>::iterator jobIter = jobQueue.begin(); jobIter != jobQueue.end(); ++jobIter)
  delete* jobIter;

}

void createJobQueue() {

 std::cout << "Entered createJobQueue() Function" << '\n';

 vector <string> ipList = optionsManager::Instance()->getIPList();
 vector <string> scanList = optionsManager::Instance()->getScanList();
```

```
 vector <string> portList = optionsManager::Instance()->getPortList();

 for (vector<string>::iterator sc = scanList.begin(); sc != scanList.end(); ++sc) {
  for (vector<string>::iterator ipIter = ipList.begin();
  ipIter != ipList.end(); ++ipIter) {
   for (vector<string>::iterator portIter =
   portList.begin(); portIter != portList.end(); ++portIter) {
    jobQueue.push_back(new Job(*ipIter, *portIter,
    *sc));
   }
  }
 }

 cout << "Jobs created: " << jobQueue.size() << endl;

 optionsManager::Instance()->deleteAllList();
}

int processCommand(map<string, string> opDict) {

 int returnVal = 0;
 string ip;
 string targetPort;
 auto value = opDict.find("help");
 if (value != opDict.end()) {
  cout << endl;
  cout << value->second;
  return 0;
 }

 value = opDict.find("ipaddressfile");
 if (value != opDict.end()) {
  string ipAddressFile = value->second;
  cout << "IP File: " << ipAddressFile << endl;
  optionsManager::Instance()->processIPFile(ipAddressFile);
 }

 value = opDict.find("prefix");
```

```
 if (value != opDict.end())
  optionsManager::Instance()->calculateIPaddresesBitwise(value->second.c_str());

 return returnVal;
}

int main(int argc, char* argv[])
{

 WSADATA wsaData;
 int iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
 if (iResult != 0) {
  cout << "WSAStartup failed: " << iResult << endl;
  return 1;
 }

 time_t start, end = 0, elapsed = 0;
 cout << "Scanning......" << endl;
 if (argc < 2)
  cout << " For Usage type :  ./portScanner -h" << endl;
 else {
  optionsManager::Instance()->readOptions(argc, argv);

  map<string, string> opDict = optionsManager::Instance()->getOptionDictionary();

  auto value = opDict.find("help");
  if (value != opDict.end()) {
   cout << endl;
   cout << value->second;
   return 0;
  }
  else {

   start = time(NULL);

   int numberOfThreads = 1;
   value = opDict.find("speedup");
   if (value != opDict.end())
```

```
 numberOfThreads = stoi(value->second);

 //Processing Command
 processCommand(opDict);

 //Creating Job
 createJobQueue();

 vector<pthread_t> threads;

 int createStatus;
 char ip[INET_ADDRSTRLEN];

 //Getting System IP
 getCurrentSystemIP(ip);
 pthread_t thread;


 for (int i = 0; i < numberOfThreads; i++) {
  createStatus = pthread_create(&thread, NULL, sendPacket, (void*)ip);
  if (createStatus != 0) {
   cout << "Create thread failed" << endl; //return;
  }
  threads.push_back(thread);
 }

 for (int i = 0; i < numberOfThreads; i++) {
  pthread_join(threads[i], NULL);
 }

}
end = time(NULL);
elapsed = end - start;
cout << "Scanning took: " << elapsed << " seconds" << endl;

printJobStats();

destroyJobQueue();
```

```
  optionsManager::Instance()->deleteSingleTon();
 }

 WSACleanup();
}
```

8.13 Appendix M: OptionsClass.cpp file

```
#include "optionsClass.h"
#include <iostream>
#include <sstream>
#include <fstream>
#include <vector>
#include <map>
#include <cstring>
#include <cstdlib>
#include <winsock2.h>
#include <WS2tcpip.h>
#include "getopt.h"

using namespace std;

optionsManager* optionsManager::m_optManager = NULL;



void optionsManager::readOptions(int argc, char* argv[])
{
    int getOptChar = 0;
    int option_index = 0;
    const char* shortOptions = "hp:i:r:f:s:u:";

    struct option longOptions[] =
    {
        {"help",          no_argument,       NULL, 'h'},
        {"ports",         required_argument, NULL, 'p'},
        {"ip",            required_argument, NULL, 'i'},
        {"prefix",        required_argument, NULL, 'x'},
        {"file",          required_argument, NULL, 'f'},
```

```
    {"scan",          required_argument, NULL, 's'},
    {"speedup",       required_argument, NULL, 'u'},
    {NULL,            0,                 NULL, 0  }
};

while ((getOptChar = getopt_long(argc, argv, shortOptions,
longOptions, &option_index)) != -1)
{
    switch (getOptChar)
    {
    case 'h':
        optionDict.insert(pair<string, string>("help", GetStandardUsageOptionScreen()));
        break;
    case 'p':
        optionDict.insert(pair<string, string>("ports", optarg));
        portList = split(optarg, ',');
        break;
    case 'i':
        optionDict.insert(pair<string, string>("ip", optarg));
        ipList.push_back(string(optarg));
        break;
    case 'x':
        optionDict.insert(pair<string, string>("prefix", optarg));
        break;
    case 'f':
        optionDict.insert(pair<string, string>("ipaddressfile", optarg));
        break;
    case 's':
        optionDict.insert(pair<string, string>("scan", optarg));
        if (strcmp(optarg, "SYN") == 0 || strcmp(optarg, "NULL") == 0
            || strcmp(optarg, "ACK") == 0 || strcmp(optarg, "UDP") == 0
            || strcmp(optarg, "XMAS") == 0 || strcmp(optarg, "FIN") == 0) {
            scanList.push_back(optarg);
        }
        else {
            cout << "INVALID SCAN " << endl;
            exit(0);
        }
```

```
            break;
        case 'u':
            optionDict.insert(pair<string, string>("speedup", optarg));
            break;
        default:
            fprintf(stderr, "ERROR: Unknown option '-%c'\n", getOptChar);
            exit(1);
        }
    }

    if (portList.size() == 0)
    {
        portList = split("1-1024", ',');
    }

    if (optind < argc)
    {
        while (optind < argc)
        {
            if (strcmp(argv[optind], "SYN") == 0 || strcmp(argv[optind], "NULL") == 0
                || strcmp(argv[optind], "ACK") == 0 || strcmp(argv[optind], "UDP") == 0
                || strcmp(argv[optind], "XMAS") == 0 || strcmp(argv[optind], "FIN") == 0) {
                scanList.push_back(argv[optind++]);
            }
            else
            {
                optind++;
            }
        }
    }

    unRollPortRange();
}

optionsManager* optionsManager::Instance()
{
    if (!m_optManager)
        m_optManager = new optionsManager();
```

```
    return m_optManager;
}

vector<string> optionsManager::split(string input, char delimiter)
{
    stringstream ss(input);
    vector<string> outputList;
    string temp;

    while (getline(ss, temp, delimiter))
    {
        outputList.push_back(temp);
    }

    return outputList;
}

void optionsManager::unRollPortRange()
{
    vector<string> tempList;
    for (auto& port : portList)
    {
        size_t pos = port.find('-');
        if (pos != string::npos)
        {
            int start = stoi(port.substr(0, pos));
            int end = stoi(port.substr(pos + 1));

            for (int i = start; i <= end; ++i)
            {
                tempList.push_back(to_string(i));
            }
        }
        else
        {
            tempList.push_back(port);
        }
    }
```

```
    portList.swap(tempList);
}

string optionsManager::GetStandardUsageOptionScreen()
{
    return  "./portScanner [option1, ..., optionN] \n \
            --help. Example: \./portScanner --help".\n \
            --ports <ports to scan>. Example: \./portScanner --ports 1,2,3-5".\n \
            --ip <IP address to scan>. Example: \./portScanner --ip 127.0.0.1".\n \
            --prefix <IP prefix to scan>. Example: \./portScanner --
prefix 127.143.151.123/24".\n \
            --
file <file name containing IP addresses to scan>. Example: \./portScanner --
file filename.txt".\n \
            --speedup <parallel threads to use>. Example: \./portScanner --
speedup 10". \n \
            --scan <one or more scans>. Example: \./portScanner --
scan SYN NULL FIN XMAS".\n";
}

map<string, string> optionsManager::getOptionDictionary()
{
    return optionDict;
}

vector<string> optionsManager::getScanList()
{
    return scanList;
}

vector<string> optionsManager::getIPList()
{
    return ipList;
}

vector<string> optionsManager::getPortList()
{
    return portList;
```

```
}

void optionsManager::deleteAllList()
{
    ipList.clear();
    portList.clear();
    scanList.clear();
    optionDict.clear();
}

void optionsManager::deleteSingleTon()
{
    delete m_optManager;
}

void optionsManager::printHostAddresses(unsigned long
networkAddress, unsigned long broadcastAddress)
{
    struct in_addr address;

    for (unsigned long i = ntohl(networkAddress) + 1; i < ntohl(broadcastAddress); ++i)
    {
        address.s_addr = htonl(i);
        ipList.push_back(string(inet_ntoa(address)));
    }
}

void optionsManager::calculateIPaddresesBitwise(const char* ipWithPrefix)
{
    struct in_addr ipaddress;
    struct in_addr ipMask;

    char* inputIP;

    int prefix;
    unsigned long networkID, hostBits, broadcastID;

    char* pch = strtok((char*)ipWithPrefix, "/");
```

```
    inputIP = pch;
    pch = strtok(NULL, "/");
    sscanf(pch, "%d", &prefix);

    inet_pton(AF_INET, inputIP, &ipaddress);
    unsigned long subnetMask = 0;
    for (int i = 0; i < prefix; ++i)
    {
        subnetMask |= 1 << (31 - i);
    }

    ipMask.s_addr = htonl(subnetMask);

    networkID = ntohl(ipaddress.s_addr) & ntohl(ipMask.s_addr);
    ipaddress.s_addr = htonl(networkID);

    ipList.push_back(inet_ntoa(ipaddress));
    hostBits = ~ntohl(ipMask.s_addr);

    broadcastID = networkID | hostBits;
    ipaddress.s_addr = htonl(broadcastID);

    ipList.push_back(inet_ntoa(ipaddress));
    printHostAddresses(networkID, broadcastID);
}

void optionsManager::processIPFile(string fileName)
{
    string fileContent = ReadIPFile(fileName.c_str());
    if (!fileContent.empty())
    {
        istringstream iss(fileContent);
        string line;
        while (getline(iss, line))
        {
            ipList.push_back(line);
        }
    }
```

```
}

string optionsManager::ReadIPFile(const char* filename)
{
    ifstream file(filename);
    stringstream buffer;
    buffer << file.rdbuf();
    return buffer.str();
}
```

8.14 Appendix N: TcpClass.cpp file

```
#include "tcpClass.h"
#include <iostream>
#include <string>
#include <cstring>
#include <ctime>
#include <winsock2.h>
#include <ws2tcpip.h>
#include "CommonUtilities.h"
#include "work.h"
#include "tcp_header.h"

#define PACKET_LENGTH 2048

using namespace std;

TCPUtilities::TCPUtilities() {}

unsigned short TCPUtilities::csum(uint8_t* data, int length)
{
    long checkSum = 0;

    while (length > 0)
    {
        checkSum += (*data << 8 & 0xFF00) + (*(data + 1) & 0xFF);
        data += 2;
        length -= 2;
```

```
    }
    if (checkSum >> 16)
        checkSum = ((checkSum >> 16) & 0x00ff) + (checkSum & 0xFFFF);

    uint16_t finalSum = (uint16_t)(~checkSum);

    return finalSum;
}

uint16_t TCPUtilities::calculateCheckSum(uint32_t ipSource,
uint32_t ipDest, uint8_t protocol, uint16_t tcpLength,
struct tcp_header tcpSegment)
{
    char packet[PACKET_LENGTH];
    int checkSumLength = 0;

    memcpy(packet, &ipSource, sizeof(ipSource));
    checkSumLength += sizeof(ipSource);

    memcpy(packet + checkSumLength, &ipDest, sizeof(ipDest));
    checkSumLength += sizeof(ipDest);

    packet[checkSumLength] = 0;
    checkSumLength += 1;

    memcpy(packet + checkSumLength, &protocol, sizeof(protocol));
    checkSumLength += sizeof(protocol);

    memcpy(packet + checkSumLength, &tcpLength, sizeof(tcpLength));
    checkSumLength += sizeof(tcpLength);

    char* tcpheader = (char*)&tcpSegment;
    memcpy(packet + checkSumLength, tcpheader, 20);
    checkSumLength += 20;

    return csum((uint8_t*)packet, checkSumLength);
}
```

```
void TCPUtilities::createPacket(string scanType, const char* destIP,
const char* portNumber, char* packetData, char* srcIP)
{
    struct tcp_header* tcp = (struct tcp_header*)packetData;
    memset(tcp, 0, sizeof(struct tcp_header));

    int min = 30000, max = 60000;
    srand(static_cast<unsigned int>(time(nullptr)));
    int sourcePort = min + rand() % (max - min + 1);

    createTCPHeader(tcp, sourcePort, portNumber, scanType);
    tcp-
>checksum = htons(calculateCheckSum(inet_addr(srcIP), inet_addr(destIP), IPPROTO_TCP,
    htons(sizeof(struct tcp_header)),*tcp));
}

void TCPUtilities::createTCPHeader(struct tcp_header* tcpHeader,
int sourcePort, const char* destPort, string scanType) {
    tcpHeader->source_port = htons(static_cast<uint16_t>(sourcePort));
    tcpHeader->dest_port = htons(static_cast<uint16_t>(atoi(destPort)));
    tcpHeader->syn = 0;
    tcpHeader->sequence = 0;
    tcpHeader->ack = 0;
    tcpHeader->window = htons(1024);
    tcpHeader->checksum = 0;
    tcpHeader->rst = 0;
    tcpHeader->urgent_pointer = 0;
    tcpHeader->data_offset = 5;

    if (scanType == "SYN") {
        tcpHeader->syn = 1;
        tcpHeader->sequence = htonl(1);
    }
    else if (scanType == "XMAS") {
        tcpHeader->psh = 1;
        tcpHeader->urg = 1;
    }
    else if (scanType == "FIN") {
```

```
        tcpHeader->fin = 1;
    }
    else if (scanType == "ACK") {
        tcpHeader->ack = 1;
    }
}

void TCPUtilities::sendTCPPacket(Job* job, char* srcIP)
{
    const char* ip = job->IP.c_str();
    const char* portNumber = job->port.c_str();
    string scanType = job->scanType;
    int probeCounter = 3;
    struct sockaddr_in victim, victim_copy;
    memset(&victim, 0, sizeof(struct sockaddr_in));
    comUtil.buildDestIPStruct(&victim, ip, portNumber);
    memcpy(&victim_copy, &victim, sizeof(victim));
    char packData[PACKET_LENGTH];
    createPacket(scanType, ip, portNumber, packData, srcIP);

    WSADATA wsaData;
    int wsResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (wsResult != 0)
    {
        cerr << "WSAStartup failed with error: " << wsResult << endl;
        return;
    }

    SOCKET sockDesc = socket(AF_INET, SOCK_RAW, IPPROTO_TCP);
    if (sockDesc == INVALID_SOCKET)
    {
        cerr << "Socket creation failed with error: "
        << WSAGetLastError() << endl;
        WSACleanup();
        return;
    }

    int status = -1;
```

```
    while (status < 0 && probeCounter > 0)
    {
        if (sendto(sockDesc, packData, sizeof(struct tcp_header), 0, (sockaddr*)&victim, sizeof(stru
        {
            status = comUtil.sniffAPacket(ip, portNumber, scanType, IPPROTO_TCP, job, sockDesc, sock
        }
        probeCounter--;
    }

    closesocket(sockDesc);
    WSACleanup();

    if (status == 0)
    {
        static HANDLE createPacketLock = CreateMutex(NULL, FALSE,
        NULL);

        WaitForSingleObject(createPacketLock, INFINITE);

        job->serviceVersion = comUtil.getServiceInfo(victim_copy, portNumber);

        ReleaseMutex(createPacketLock);
    }

    job->jobStatus = COMPLETED;
}
```

8.15 Appendix O: UcpClass.cpp file

```
#include "udpClass.h"
#include "DNS_Header.h"
#include "udp_header.h"
#include "work.h"

void UDPUtilities::createUDPHeader(struct udphdr* udpHeader,
int sourcePort, const char* destPort)
{
    udpHeader->source = htons(sourcePort);
```

```
    udpHeader->dest = htons(atoi(destPort));
    udpHeader->length = htons(sizeof(struct udphdr));
    udpHeader->checksum = 0;
}

void UDPUtilities::createDNSPacket(char* ipAddress, char* packet)
{
    DNS_HEADER* dnsHeader = (DNS_HEADER*)packet;

    dnsHeader->id = htons(rand());
    dnsHeader->qr = 0;
    dnsHeader->opcode = 0;
    dnsHeader->aa = 0;
    dnsHeader->tc = 0;
    dnsHeader->rd = 1;
    dnsHeader->ra = 0;
    dnsHeader->z = 0;
    dnsHeader->ad = 0;
    dnsHeader->cd = 0;
    dnsHeader->rcode = 0;
    dnsHeader->q_count = htons(1);
    dnsHeader->ans_count = 0;
    dnsHeader->auth_count = 0;
    dnsHeader->add_count = 0;
}

void UDPUtilities::convertToDNSNameFormat(unsigned char* dnsHeader,
char* destinationHost)
{
    unsigned char* rvIterator = dnsHeader;
    int count = 0;

    while (*destinationHost)
    {
        if (*destinationHost == '.')
        {
            *rvIterator++ = count;
            count = 0;
```

```
        }

        else
        {
            *rvIterator++ = *destinationHost;
            count++;
        }
        destinationHost++;
    }
    *rvIterator++ = count;
    *rvIterator = '\0';
}

int UDPUtilities::createPacketUDP(int sourcePort, const char*
destPort, char* destIpAddress, char* packet)
{
    struct udphdr* udpPack = (struct udphdr*)packet;

    size_t totalSize = sizeof(struct udphdr);

    createUDPHeader(udpPack, sourcePort, destPort);
    if (strcmp(destPort, "53") == 0)
    {
        createDNSPacket(destIpAddress, packet + sizeof
        (struct udphdr));
    }
    return totalSize;
}

void UDPUtilities::sendUDPPacket(Job* job)
{
    const char* destPort = job->port.c_str();
    const char* destIpAddress = job->IP.c_str();
    string scanType = job->scanType;

    WSADATA wsaData;
    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
    {
```

```
    cout << "Failed to initialize Winsock.\n";
    return;
}

SOCKET sockDesc = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
if (sockDesc == INVALID_SOCKET) {
    cout << "Socket creation failed.\n";
    WSACleanup();
    return;
}

char packData[PACKET_LENGTH];

memset(packData, 0, PACKET_LENGTH);
size_t totalSize = sizeof(struct udphdr);

int min = 30000, max = 60000;
srand((unsigned int)time(NULL));

int sourcePort = min + rand() % (max - min + 1);
totalSize = createPacketUDP(sourcePort, destPort,
(char*)destIpAddress, packData);

struct sockaddr_in destAddr;
destAddr.sin_family = AF_INET;
destAddr.sin_port = htons(atoi(destPort));
destAddr.sin_addr.s_addr = inet_addr(destIpAddress);

int bytesSent = sendto(sockDesc, packData, totalSize, 0,
(struct sockaddr*)&destAddr, sizeof(destAddr));
if (bytesSent == SOCKET_ERROR) {
    cout << "Send failed with error: " << WSAGetLastError()
    << "\n";
}

closesocket(sockDesc);
WSACleanup();
```

```
    job->jobStatus = COMPLETED;
}
```

8.16 Appendix P: Work.cpp file

```
#include "work.h"

void Job::setJob(void* (*fptr)(void*))
{
 funcPointer = fptr;
}



void Job::execute()
{
 (*funcPointer)(this);
}

Job::Job(string ipAddress, string portNum, string scan)
{
 IP = ipAddress;
 port = portNum;
 scanType = scan;
 jobStatus = NOTNOW;
 serviceName = "NA";
 serviceVersion = "NA";
 conclusion = "NOTAVAILABLE";
}

Job::Job() {}

Job::~Job() {};
```

References

1. Abedin, M., Nessa, S., Al-Shaer, E., Khan, L.: Vulnerability analysis for evaluating quality of protection of security policies. In: Proceedings of the 2nd ACM Workshop on Quality of Protection (QoP 2006), Alexandria VA (October 2006)
2. Fyodor, "The Art of Port Scanning", Phrack Magazine, Volume 7, Issue 51, September 01 1997, Article 11 of 17.
3. George Kurtz and Hacking Exposed, "Network Security Secrets Solutions [M]" in , McGraw-Hill Companies, 2001.
4. R. Sira, "Network Forensics Analysis Tools: An Overview of an Emerging Technology", GSEC Version 1.4, Jan. 2003.
5. A.Sridharan, T. Ye, et al. "Connectionsless Port Scan Detection on the Backbone." Proceedings of the 25th IEEE International Performance, Computing, and Communications Conference, 2007,pp.566-576.
6. A. Aksoy, S. Louis, and M. H. Gunes. Operating system fingerprinting via automated network traffic analysis. In IEEE Congress on Evolutionary Computation (CEC). IEEE, 2017.
7. N. Sklavos and O. Koufopavlou, "Mobile communications world: security implementations aspects-a state of the art," CSJM Journal, Institute of Mathematics and Computer Science, vol. 11, no. 2, pp. 168–187, 2003.
8. S. Bratus, C. Cornelius, D. Kotz, and D. Peebles. Active behavioral fingerprinting of wireless devices. In WiSec '08: Proceedings of the first ACM conference on Wireless net work security, pages 56-61, New York, NY, USA, 2008. ACM.
9. J. Franklin, D. McCoy, P. Tabriz, V. Neagoe, J. Van Rand wyk, and D. Sicker. Passive data link layer 802.11 wireless device driver fingerprinting. In USENIX-SS'06: Proceed ings of the 15th conference on USENIX Security Symposium, Berkeley, CA, USA, 2006. USENIX Association.
10. Z. Durumeric, E. Wustrow, and J. A. Halderman, "Zmap: Fast internet-wide scanning and its security applications," in 22nd USENIX Security Symposium (USENIX Security 13). Washington, D.C.: USENIX Association, Aug. 2013, pp. 605–620. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/durumeric
11. D. B. K. Ramakrishnan, S. Floyd, "The Addition of Explicit Congestion Notification (ECN) to IP," Internet Requests for Comments, RFC Editor, RFC 3168, September 2001. [Online]. Available: https://www.rfc-editor.org/rfc/rfc3168.txt

12. B. Anderson and D. McGrew, “Os fingerprinting: New techniques and a study of information gain and obfuscation,” in 2017 IEEE Conference on Communications and Network Security (CNS), 2017, pp. 1–9.

13. D. Veitch, S. Babu and A. Psztor, ”Robust Synchronization of Software Clocks Across the Internet”, Proc. Fourth ACM SIGCOMM Conf. Internet Measurement, 2004.

14. F. Veysset, O. Courtay and O. Heen, ”New Tool and Technique for Remote Operating System Fingerprinting”, 2002.

15. R. S. Shirbhate and P. A. Patil ”Network Traffic Monitoring Using Intrusion Detection System”, International Journal of Advanced Research in Computer Science and Software Engineering, Volume 2, Issue 1, January 2012, ISSN: 2277 128X.

16. Research paper by Engineering Research Council of Canada and Dalhousie University Electronic Commerce Executive Committee.

17. “An implementation of intrusion detection system using genetic algorithm” Mohammad Sazzadul Hoque1, Md. Abdul Mukit2 and Md. Abu Naser Bikas3 1Student, Department of Computer Science and Engineering, Shahjalal University of Science and Technology, Sylhet, Bangladesh sazzad@ymail.com 2Student, Department of Computer Science and Engineering, Shahjalal University of Science and Technology, Sylhet, Bangladesh mukit.sust027@gmail.com 3Lecturer, Department of Computer Science and Engineering, Shahjalal University of Science and Technology, Sylhet, Bangladesh bikasbd@yahoo.com

18. Karen, Scarfone Peter Mell, (2007) “Guide To Intrusion Detection And Prevention Systems (IDPS)”. Washington, D.C.: National Institute of Standards and Technology, Special Publication 800 94, 128 p.

19. L. T. Heberlein K. N. Levitt B. Mukherjee, (1991) “A Method To Detect Intrusive Activity in a Networked Environment”. In: 14th National Computer Security Conference. Washington, D.C.: National Institute of Standards and Technology, National Computer Security Center, pp. 362-371

20. Tews, Erik Beck, Martin, (2009) “Practical attacks against WEP and WPA” In: Proceedings of the second ACM conference on Wireless network security. ACM, p. 79-86.