Chapter 1: Crafting a Rust-Based Network Sniffer

Network sniffers, essential tools in understanding and analyzing data flow across networks, can be crafted from scratch with Rust. Rust's unique blend of performance and safety makes it an ideal choice for low-level network programming. In this exploration, we'll leverage the capabilities of the **socket2** crate, a powerful Rust library that provides abstractions for working with **raw sockets**.

Building a network sniffer requires a solid understanding of the fundamentals of raw network packets in Rust. In Rust, handling raw packets involves close interaction with sockets and meticulous byte manipulation. The `socket2` crate simplifies this process, offering abstractions that facilitate efficient packet capture and processing.

Before we dive into the complexities of network sniffing, let's establish a foundation by understanding how Rust manages raw network packets. This understanding is pivotal as we construct our custom network sniffer, allowing us to delve deep into the world of low-level networking.

1. Crafting a Rust-Based UDP Host Discovery Tool

In our pursuit to develop a robust **UDP** host discovery tool using Rust, the primary objective is to identify hosts within a target network. This tool holds significance in scenarios where a comprehensive overview of potential targets is sought, facilitating the streamlining of **reconnaissance** and exploitation efforts. The methodology relies on leveraging a well-established behavior exhibited by most operating systems when confronted with UDP datagrams directed at closed ports. Typically, a responsive host generates an ICMP message indicating that the port is unreachable. This **ICMP** response serves as a valuable indicator of an active host, forming the basis for our host discovery mechanism.

The deliberate selection of the User Datagram Protocol (UDP) stems from its inherent advantages in efficiently broadcasting messages across a subnet with minimal overhead. The lightweight nature of this protocol positions it as an ideal candidate for our purposes, emphasizing the goal of maximizing coverage while minimizing potential disruptions. A pivotal aspect of our approach involves the careful selection of a UDP port that is unlikely to be in active use, enabling us to probe multiple ports for comprehensive host coverage.

Understanding the role of UDP in our host discovery tool necessitates a closer examination of its characteristics. UDP, characterized by its connectionless and lightweight nature, aligns seamlessly with our objective of quick and efficient host identification. The absence of overhead associated with connection-oriented protocols renders UDP well-suited for our purpose, as we intend to seamlessly broadcast messages and await ICMP responses. This strategic choice is not arbitrary; rather, it is a deliberate selection based on the efficiency and simplicity afforded by UDP in the context of host discovery.

Delving into the complexities of our approach, the emphasis extends beyond constructing a basic host discovery tool to meticulous decoding and analysis of various network protocol headers. The true potency of our tool lies not only in its ability to identify hosts but also in its capacity to decipher the complex layers of network communications. Decoding network protocol headers necessitates a nuanced understanding of underlying structures, and Rust, with its focus on performance and safety, emerges as a dependable companion in this journey.

As we kick off the implementation of this UDP host discovery tool, the scope extends beyond a singular operating system. The objective is to create a tool transcending platform limitations, catering to both Windows and Linux environments. This versatility is not merely a convenience but a strategic decision to enhance the tool's applicability, particularly in enterprise environments characterized by diverse operating systems.

The prospective evolution of our tool goes beyond basic host discovery. We envision the incorporation of additional logic that could trigger comprehensive **Nmap** port scans on any discovered hosts. This strategic enhancement adds a layer of sophistication, allowing for a deeper exploration of the network attack surface associated with each identified host. The decision to leave this as an exercise for the user underscores our commitment to fostering creativity and innovation within the realm of network exploration.

1.1 Network Exploration: Decoding the Essence of UDP

Packet sniffing, a fundamental aspect of network analysis, extends on Windows and Linux platforms, each requiring a distinct approach. To ensure adaptability across operating systems, we adopt a strategy that involves creating a socket object and dynamically determining the underlying platform. This platform awareness becomes particularly crucial as the complexities of raw socket access vary between Windows and Linux environments.

On Windows, the process entails additional steps due to the need for setting specific flags via a socket input/output control (IOCTL) mechanism. This mechanism serves as a means of communication between user-space programs and kernel-mode components, facilitating the configuration of network interfaces to operate in **promiscuous mode**. Promiscuous mode, a powerful but privileged state, allows the network interface to capture all incoming packets, regardless of their destination, providing a comprehensive view of network activity. The initiation of promiscuous mode on Windows involves the strategic use of IOCTL to enable the reception of all packets.

In contrast, the Linux counterpart focuses on the specificity of protocols, where the example utilizes the Internet Control Message Protocol (ICMP) for packet sniffing. Linux, by default, requires a more targeted approach, necessitating the selection of a specific protocol for packet capture. The Rust implementation adeptly accommodates these differences, showcasing its platform-aware design.

```rust
use socket2::{Domain, Protocol, Socket, Type};
use std::io::Result;
use std::mem::MaybeUninit;
use std::net::SocketAddr;

fn main() -> Result<()> {
    // Define the host to listen on
    let host: SocketAddr = "0.0.0.0:12345".parse().unwrap();

    let socket_protocol = if cfg!(target_os = "windows") {
        0
    } else {
        1
    };

    // Create a raw socket
    let sniffer = Socket::new(
        Domain::IPV4,
        Type::RAW,
        Some(Protocol::from(socket_protocol)),
    )?;
    // bind to the public interface
    sniffer.bind(&host.into())?;

    // Read one packet
    let mut buffer: [MaybeUninit<u8>; 65535] = unsafe { MaybeUninit::uninit().assume_init()
    let _ = sniffer.recv_from(&mut buffer)?;
    let raw_buffer: &[u8] =
        unsafe { std::slice::from_raw_parts(buffer.as_ptr() as *const u8, buffer.len()) };

    // Print the captured packet
    println!("{:?}", raw_buffer);

    Ok(())
}
```

The provided Rust code exemplifies the initiation of a raw socket sniffer, starting with the definition of the host IP address to listen on. The subsequent steps involve the creation of a socket object, taking into account the protocol variations between Windows and Linux. In this context, the `cfg!(windows)` macro plays a pivotal role in conditionally determining the platform and adjusting the socket protocol accordingly.

The default configuration of the socket will include IP headers in the captured packets, enhancing the depth of information gathered during the sniffing process. Moreover, the script automatically handles the complexities of promiscuous

mode, a critical feature for comprehensive packet capture.

While the provided Rust code captures a single packet for simplicity, it serves as a foundational example for more extensive network analysis tasks. The flexibility of Rust, combined with its platform-aware features, positions it as a reliable choice for crafting network tools that seamlessly operate across diverse operating systems. This illustrative example demystifies the complexities of packet sniffing on Windows and Linux, laying the groundwork for more sophisticated network exploration and analysis endeavors.

```
:dep socket2 = {version = "0.5.5", features = ["all"]}
```

```
use std::process::{Command, Output, Stdio};

// A helper function to execute a shell command from a Rust script
fn execute_command(command: &str) -> Result<(), std::io::Error> {
    let status = Command::new("bash")
        .arg("-c")
        .arg(command)
        .stderr(Stdio::inherit())
        .status()?;

    if status.success() {
        Ok(())
    } else {
        Err(std::io::Error::from_raw_os_error(status.code().unwrap_or(1)))
    }
}
```

```
// The following command will execute the sniffer.
// Set your sudo password below by replacing 'your-passowrd' accordingly

let command = "cd decoding-the-essence-of-udp && cargo build && echo 'your-passowrd' | sudo

if let Err(err) = execute_command(command) {
    eprintln!("Error executing command: {}", err);
}

// In another terminal or shell window, choose a host to ping, for example: ping google.com
```

```
[sudo] password for mahmoud:    Compiling decoding-the-essence-of-udp v0.1.0 (/home/mahmoud/
    Finished dev [unoptimized + debuginfo] target(s) in 0.28s


[69, 0, 0, 84, 0, 0, 0, 0, 113, 1, 211, 15, 142, 251, 37, 238, 192, 168, 1, 8, 0, 0, 249, 9,
```

```
()
```

The displayed output indicates the successful capture of the initial ICMP ping request destined for the specified host. Running this example on Linux would yield the response from the pinged host.

Capturing a single packet is limited in utility, prompting us to enhance the functionality to process more packets and decode their contents. Let's proceed by incorporating additional features into our sniffer code.

1.2 Decoding the IP Layer and Uncover Packet Secrets

Within the current implementation of our sniffer, we capture a lot of data, including IP headers and higher-level protocols such as **TCP**, **UDP**, or **ICMP**. However, this information currently exists in an encoded binary form, presenting a significant challenge for comprehension. Our immediate objective is to decode the IP segment of a packet, a pivotal step that enables us to extract valuable insights. This includes determining the protocol type (TCP, UDP, or ICMP) and identifying the source and destination IP addresses. This decoding process sets the stage for a more profound understanding, forming the ground for parsing additional protocols in subsequent stages of our exploration.

When examining an actual packet traversing the network, it becomes evident that decoding incoming packets requires a clear comprehension of their structure. The following table provides insight into the composition of an **IP header**, delineating its various fields.

Bit Offset	Field	Size (in bits)
0–3	Version	4
4–7	HDR length	4
8–15	Type of service	8
16–31	Total length	16
32–39	Identification	8
40–47	Flags	8
48–63	Fragment offset	16
64–71	Time to live	8
72–79	Protocol	8
80–95	Header checksum	16
96–127	Source IP address	32
128–159	Destination IP address	32
160 onward	Options	Variable

Our goal is to make sense of the IP header, excluding the last `Options` field, and

focus on pulling out important information like the type of protocol, where the data is coming from (source IP), and where it's going (destination IP). To do this, we need a smart approach to break down each part of the IP header. For this task, we are using Rust which is good at handling this kind of challenge.

As you may know, Rust provides us with a special tool called "struct", which is like a blueprint that helps us understand and organize the data we're dealing with. This tool works well with binary data, the language that computers speak.

Now, let's dive into how we can use this Rust **struct** tool to read an IP header. Think of it like having a map that shows us the layout of the information we're looking for. Rust's struct acts like a guide, helping us decode the binary data that represents an IP header. It's like a decoder ring that makes sense of the mysterious binary language.

Understanding Rust's struct is like learning the rules of a game. The rules help us play efficiently and understand how the game works. Similarly, Rust's struct has its own set of rules that help us interpret the binary data and understand what each part of the IP header means.

As we go through the steps of using Rust's struct to read an IP header, think of it as following a recipe. The recipe (Rust's struct) tells us what ingredients (binary data) we need and how to combine them to get the final dish (decoded IP header). Rust's struct acts as our recipe book, guiding us through the process of turning complex binary data into something we can easily understand.

The flexibility of Rust's struct is like having a Swiss Army knife. It not only helps us decode the IP header but also opens up possibilities for doing more advanced tasks, like understanding different types of protocols and analyzing networks in more detail. This journey with Rust's struct is not just about reading data; it's about unlocking the secrets hidden in the binary language and using them to understand networks better.

In a nutshell, our exploration of decoding the IP header in Rust is like embarking on an exciting adventure. Rust, our trusty companion, makes the journey smoother by providing us with the right tools to decipher the language of computers. As we navigate through Rust's struct, it's not just about decoding binary data; it's about gaining insights and understanding the fascinating world of networks.

1.3 The IP Header Struct

In the following code snippet, we encounter the definition of a new Rust struct object named `IP`, meticulously crafted to read and parse packet headers into distinct fields. The `IP` struct is equiped with a set of fields, each meticulously aligned with the components of the IP header, as illustrated in the earlier-mentioned IP header table. Each field is assigned a specific name, such as `ver_ihl` or `offset`, along with its corresponding data type, like `u8` or `u16`. The ability to specify bit width adds a layer of flexibility, offering the liberty to dictate

lengths beyond the byte level, an important feature that exceeds conventional constraints.

```
struct IP {
    ver_ihl: u8,
    tos: u8,
    len: u16,
    id: u16,
    offset: u16,
    ttl: u8,
    protocol_num: u8,
    sum: u16,
    src: u32,
    dst: u32,
}
```

This struct, being the cornerstone of our packet parsing work, demands a well-defined structure before instantiation. The `new` method comes to the forefront, adept at filling the fields with appropriate values. As we traverse the complexities of the `new` method, it takes a buffer as its first argument and crafts an object of the `IP` struct. The utilization of format characters within the method becomes pivotal, outlining the structure of binary data with precision.

```
impl IP {
    fn new(buff: &[u8]) -> Option<Self> {
        if buff.len() >= 20 {
            let header = IP {
                ver_ihl: buff[0],
                tos: buff[1],
                len: u16::from_be_bytes([buff[2], buff[3]]),
                id: u16::from_be_bytes([buff[4], buff[5]]),
                offset: u16::from_be_bytes([buff[6], buff[7]]),
                ttl: buff[8],
                protocol_num: buff[9],
                sum: u16::from_be_bytes([buff[10], buff[11]]),
                src: u32::from_be_bytes([buff[12], buff[13], buff[14], buff[15]]),
                dst: u32::from_be_bytes([buff[16], buff[17], buff[18], buff[19]]),
            };

            Some(header)
        } else {
            None
        }
    }
}
```

The `new` method unfolds as a meticulous orchestra of data extraction, where

each field of the IP struct is populated by interpreting the corresponding bytes from the input buffer. The method meticulously adheres to the complexities of the IP header, ensuring that each field, from version and type of service to source and destination IP addresses, is accurately represented. This methodical approach not only adheres to the Rust language's conventions but also aligns seamlessly with the binary nature of network data.

```rust
impl IP {
    fn protocol(&self) -> String {
        match self.protocol_num {
            1 => String::from("ICMP"),
            4 => String::from("IPv4"),
            6 => String::from("TCP"),
            17 => String::from("UDP"),
            255 => String::from("Reserved"),
            _ => format!("{}", self.protocol_num),
        }
    }

    fn src_address(&self) -> String {
        Ipv4Addr::from(self.src).to_string()
    }

    fn dst_address(&self) -> String {
        Ipv4Addr::from(self.dst).to_string()
    }

    fn offset(&self) -> String {
        self.offset.to_string()
    }

    fn ttl(&self) -> String {
        self.ttl.to_string()
    }

    fn ver(&self) -> String {
        self.ver_ihl.to_string()
    }

    fn len(&self) -> String {
        self.len.to_string()
    }
}
```

Beyond the instantiation complexities, the IP struct extends its capabilities with additional methods designed to provide meaningful insights into the parsed IP

header. The `protocol` method, for instance, translates the protocol number into human-readable format, offering clarity on whether it corresponds to ICMP, IPv4, TCP, UDP, or falls under the category of reserved protocols. This method encapsulates an important understanding of the IP header's composition, bridging the gap between raw data and comprehensible information.

Further enriching the `IP` struct's functionality are methods such as `src_address`, `dst_address`, `offset`, `ttl`, `ver`, and `len`, each meticulously crafted to extract and present specific elements of the IP header. The `src_address` and `dst_address` methods, for instance, leverage Rust's ability to convert raw IP addresses into human-readable strings, providing insights into the source and destination addresses with clarity. Meanwhile, `offset`, `ttl`, `ver`, and `len` offer a glimpse into the fragment offset, time to live, version, and total length of the IP header, respectively.

In essence, this Rust code snippet not only materializes the complexities of creating a robust `IP` struct for parsing IP headers but also unfolds as a complex symphony of methods that bridge the gap between raw binary data and comprehensible insights. The thoughtful design choices and meticulous data extraction techniques showcased here underscore Rust's prowess in low-level programming and its ability to handle network data with finesse.

1.4 Putting It All Together

To integrate the recently developed IP decoding struct into our network sniffer, we will incorporate the functionality within our script:

```rust
use socket2::{Domain, Protocol, Socket, Type};
use std::io::Result;
use std::mem::MaybeUninit;
use std::net::SocketAddr;
use std::net::Ipv4Addr;


struct IP {
    ver_ihl: u8,
    tos: u8,
    len: u16,
    id: u16,
    offset: u16,
    ttl: u8,
    protocol_num: u8,
    sum: u16,
    src: u32,
    dst: u32,
}
```

```rust
impl IP {
    fn new(buff: &[u8]) -> Option<Self> {
        if buff.len() >= 20 {
            let header = IP {
                ver_ihl: buff[0],
                tos: buff[1],
                len: u16::from_be_bytes([buff[2], buff[3]]),
                id: u16::from_be_bytes([buff[4], buff[5]]),
                offset: u16::from_be_bytes([buff[6], buff[7]]),
                ttl: buff[8],
                protocol_num: buff[9],
                sum: u16::from_be_bytes([buff[10], buff[11]]),
                src: u32::from_be_bytes([buff[12], buff[13], buff[14], buff[15]]),
                dst: u32::from_be_bytes([buff[16], buff[17], buff[18], buff[19]]),
            };

            Some(header)
        } else {
            None
        }
    }

    fn protocol(&self) -> String {
        // Refer to ---> https://www.iana.org/assignments/protocol-numbers/protocol-numbers
        match self.protocol_num {
            0 => String::from("HOPOPT"),
            1 => String::from("ICMP"),
            2 => String::from("IGMP"),
            3 => String::from("GGP"),
            4 => String::from("IPv4"),
            5 => String::from("ST"),
            6 => String::from("TCP"),
            7 => String::from("CBT"),
            8 => String::from("EGP"),
            9 => String::from("IGP"),
            10 => String::from("BBN-RCC-MON"),
            11 => String::from("NVP-II"),
            12 => String::from("PUP"),
            13 => String::from("ARGUS"),
            14 => String::from("EMCON"),
            15 => String::from("XNET"),
            16 => String::from("CHAOS"),
            17 => String::from("UDP"),
            18 => String::from("MUX"),
            19 => String::from("DCN-MEAS"),
```

```
            20 => String::from("HMP"),
            21 => String::from("PRM"),
            22 => String::from("XNS-IDP"),
            23 => String::from("TRUNK-1"),
            24 => String::from("TRUNK-2"),
            25 => String::from("LEAF-1"),
            26 => String::from("LEAF-2"),
            27 => String::from("RDP"),
            28 => String::from("IRTP"),
            29 => String::from("ISO-TP4"),
            30 => String::from("NETBLT"),
            31 => String::from("MFE-NSP"),
            32 => String::from("MERIT-INP"),
            33 => String::from("DCCP"),
            34 => String::from("3PC"),
            35 => String::from("IDPR"),
            36 => String::from("XTP"),
            37 => String::from("DDP"),
            38 => String::from("IDPR-CMTP"),
            39 => String::from("TP++"),
            40 => String::from("IL"),
            41 => String::from("IPv6"),
            42 => String::from("SDRP"),
            43 => String::from("IPv6-Route"),
            44 => String::from("IPv6-Frag"),
            45 => String::from("IDRP"),
            46 => String::from("RSVP"),
            47 => String::from("GRE"),
            48 => String::from("DSR"),
            49 => String::from("BNA"),
            50 => String::from("ESP"),
            51 => String::from("AH"),
            52 => String::from("I-NLSP"),
            53 => String::from("SWIPE (deprecated)"),
            54 => String::from("NARP"),
            55 => String::from("MOBILE"),
            56 => String::from("TLSP"),
            57 => String::from("SKIP"),
            58 => String::from("IPv6-ICMP"),
            59 => String::from("IPv6-NoNxt"),
            60 => String::from("IPv6-Opts"),
            61 => String::from("any host internal protocol"),
            62 => String::from("CFTP"),
            63 => String::from("any local network"),
            64 => String::from("SAT-EXPAK"),
```

```
        65 => String::from("KRYPTOLAN"),
        66 => String::from("RVD"),
        67 => String::from("IPPC"),
        68 => String::from("any distributed file system"),
        69 => String::from("SAT-MON"),
        70 => String::from("VISA"),
        71 => String::from("IPCV"),
        72 => String::from("CPNX"),
        73 => String::from("CPHB"),
        74 => String::from("WSN"),
        75 => String::from("PVP"),
        76 => String::from("BR-SAT-MON"),
        77 => String::from("SUN-ND"),
        78 => String::from("WB-MON"),
        79 => String::from("WB-EXPAK"),
        80 => String::from("ISO-IP"),
        81 => String::from("VMTP"),
        82 => String::from("SECURE-VMTP"),
        83 => String::from("VINES"),
        84 => String::from("IPTM"),
        85 => String::from("NSFNET-IGP"),
        86 => String::from("DGP"),
        87 => String::from("TCF"),
        88 => String::from("EIGRP"),
        89 => String::from("OSPFIGP"),
        90 => String::from("Sprite-RPC"),
        91 => String::from("LARP"),
        92 => String::from("MTP"),
        93 => String::from("AX.25"),
        94 => String::from("IPIP"),
        95 => String::from("MICP (deprecated)"),
        96 => String::from("SCC-SP"),
        97 => String::from("ETHERIP"),
        98 => String::from("ENCAP"),
       100 => String::from("GMTP"),
       101 => String::from("IFMP"),
       102 => String::from("PNNI"),
       103 => String::from("PIM"),
       104 => String::from("ARIS"),
       105 => String::from("SCPS"),
       106 => String::from("QNX"),
       107 => String::from("A/N"),
       108 => String::from("IPComp"),
       109 => String::from("SNP"),
       110 => String::from("Compaq-Peer"),
```

```
            111 => String::from("IPX-in-IP"),
            112 => String::from("VRRP"),
            113 => String::from("PGM"),
            114 => String::from("any 0-hop protocol"),
            115 => String::from("L2TP"),
            116 => String::from("DDX"),
            117 => String::from("IATP"),
            118 => String::from("STP"),
            119 => String::from("SRP"),
            120 => String::from("UTI"),
            121 => String::from("SMP"),
            122 => String::from("SM (deprecated)"),
            123 => String::from("PTP"),
            124 => String::from("ISIS over IPv4"),
            125 => String::from("FIRE"),
            126 => String::from("CRTP"),
            127 => String::from("CRUDP"),
            128 => String::from("SSCOPMCE"),
            129 => String::from("IPLT"),
            130 => String::from("SPS"),
            131 => String::from("PIPE"),
            132 => String::from("SCTP"),
            133 => String::from("FC"),
            134 => String::from("RSVP-E2E-IGNORE"),
            135 => String::from("Mobility Header"),
            136 => String::from("UDPLite"),
            137 => String::from("MPLS-in-IP"),
            138 => String::from("manet"),
            139 => String::from("HIP"),
            140 => String::from("Shim6"),
            141 => String::from("WESP"),
            142 => String::from("ROHC"),
            143 => String::from("Ethernet"),
            144 => String::from("AGGFRAG"),
            145 => String::from("NSH"),
            146..=252 => String::from("Unassigned"),
            253 => String::from("Use for experimentation and testing"),
            254 => String::from("Use for experimentation and testing"),
            255 => String::from("Reserved"),
            _ => format!("{}", self.protocol_num),
        }
    }

    fn src_address(&self) -> String {
        Ipv4Addr::from(self.src).to_string()
```

```
    }

    fn dst_address(&self) -> String {
        Ipv4Addr::from(self.dst).to_string()
    }

    fn offset(&self) -> String {
        self.offset.to_string()
    }

    fn ttl(&self) -> String {
        self.ttl.to_string()
    }

    fn ver(&self) -> String {
        self.ver_ihl.to_string()
    }

    fn len(&self) -> String {
        self.len.to_string()
    }
}

fn main() -> Result<()> {
    // Define the host to listen on
    let host: SocketAddr = "0.0.0.0:12345".parse().unwrap();

    let socket_protocol = if cfg!(target_os = "windows") {
        0
    } else {
        1
    };

    // Create a raw socket
    let sniffer = Socket::new(
        Domain::IPV4,
        Type::RAW,
        Some(Protocol::from(socket_protocol)),
    )?;
    // bind to the public interface
    sniffer.bind(&host.into())?;

    // Read one packet
    let mut buffer: [MaybeUninit<u8>; 65535] = unsafe { MaybeUninit::uninit().assume_init()
    let _ = sniffer.recv_from(&mut buffer)?;
```

```
    let raw_buffer: &[u8] =
        unsafe { std::slice::from_raw_parts(buffer.as_ptr() as *const u8, buffer.len()) };

    if raw_buffer.len() < 20 {
        eprintln!("Invalid packet: too short");
        return Ok(());
    }

    // Create an IP header from the first 20 bytes
    let ip_header = match IP::new(&raw_buffer[..20]) {
        Some(header) => header,
        None => return Ok(())
    };

    println!(
        "Protocol: {} {} -> {}",
        "ICMP",
        ip_header.src_address(),
        ip_header.dst_address()
    );

    println!("Version: {}", ip_header.ver());

    println!(
        "Header Length: {} TTL: {}",
        ip_header.len(),
        ip_header.ttl()
    );

    Ok(())
}
```

Let's put our previously developed code to the test to gain insights into the information extracted from the raw packets traversing the network. We highly recommend conducting this test from a Windows machine to leverage the diverse protocols such as UDP, and TCP, enabling fascinating testing scenarios like opening a web browser. For those confined to Linux, an alternative is to execute the previous ping test to witness the code in action.

```
// The following command will execute the sniffer.
// Set your sudo password below by replacing 'your-passowrd' accordingly

let command = "cd decoding-the-ip-header && cargo build && echo 'your-passowrd' | sudo -S se

if let Err(err) = execute_command(command) {
    eprintln!("Error executing command: {}", err);
```

```
}

// In another terminal or shell window, choose a host to ping, for example: ping google.com
```

```
   Compiling decoding-the-ip-header v0.1.0 (/home/mahmoud/Desktop/Rust Book Dark/dark-web-ru
    Finished dev [unoptimized + debuginfo] target(s) in 0.25s


Protocol: ICMP 142.251.37.238 -> 192.168.1.8
Version: 69
Header Length: 84 TTL: 113
Protocol: ICMP 142.251.37.238 -> 192.168.1.8
Version: 69
Header Length: 84 TTL: 113
Protocol: ICMP 142.251.37.238 -> 192.168.1.8
Version: 69
Header Length: 84 TTL: 113
Protocol: ICMP 142.251.37.238 -> 192.168.1.8
Version: 69
Header Length: 84 TTL: 113
Protocol: ICMP 142.251.37.238 -> 192.168.1.8
Version: 69
Header Length: 84 TTL: 113
```

It's evident that we encounter a limitation here, observing only a response for the ICMP protocol. Yet, for our intended purpose of crafting a host discovery scanner, this limitation is entirely acceptable. Our next course of action involves applying the same decoding techniques previously employed for the IP header to decode the ICMP messages, thereby expanding the capabilities of our network exploration tool.

1.5 ICMP Structure Decoding

In the world of network security and **penetration testing**, understanding the complexities of packet decoding is crucial. Having previously explored the decoding of the IP layer in our packet-sniffing efforts, our next attempt involves deciphering ICMP packets produced by our scanner's transmission of UDP datagrams to closed ports. **ICMP**, or Internet Control Message Protocol, messages possess varying contents, yet maintain three consistent elements: the type, code, and checksum fields. These fields play a pivotal role in conveying the nature of the ICMP message to the receiving host, thereby guiding its proper decoding.

ICMP Structure Within the universe of ICMP messages, our focus narrows down to those with a type value of 3 and a corresponding code value of 3. This specific combination signifies the Destination Unreachable class of ICMP

messages, specifically pinpointing the occurrence of a Port Unreachable error. The structural blueprint of a Destination Unreachable ICMP message is illustrated in the following excerpt that is shamelessly taken from **RFC 792 Page 4**, outlining the composition of its type, code, header, checksum, and additional components:

Destination Unreachable Message

0 1 2 3 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 +-+ | Type | Code | Checksum | +-+ | unused | +-+ | Internet Header + 64 bits of Original Data Datagram | +-+

IP Fields:

Destination Address

```
The source network and address from the original datagram's data.
```

ICMP Fields:

Type

```
3
```

Code

```
0 = net unreachable;

1 = host unreachable;

2 = protocol unreachable;

3 = port unreachable;

4 = fragmentation needed and DF set;

5 = source route failed.
```

Checksum

```
The checksum is the 16-bit ones's complement of the one's
complement sum of the ICMP message starting with the ICMP Type.
For computing the checksum , the checksum field should be zero.
This checksum may be replaced in the future.
```

Internet Header + 64 bits of Data Datagram

```
The internet header plus the first 64 bits of the original
```

[Page 4]

Examining this structure, we discern that the first 8 bits represent the type, followed by the subsequent 8 bits housing the ICMP code. Notably, the originating IP header of the message that triggered the response is encapsulated within the ICMP message, presenting an opportunity for validation.

Integration of ICMP Decoding Now, let's seamlessly integrate the decoding of ICMP packets into our existing packet-sniffing framework. The augmentation of our sniffer encompasses the implementation of an ICMP structure beneath the established IP structure. The following code snippet imports necessary modules and initializes the ICMP structure to facilitate the decoding process when ICMP packets are detected.

```
use std::env;
use std::mem::MaybeUninit;
use std::net::SocketAddr;
use std::net::{IpAddr, Ipv4Addr};


const ICMP_TYPE_CODE_MAP: &[((u8, u8), &str)] = &[
    // taken from ---> https://www.iana.org/assignments/icmp-parameters/icmp-parameters.xht
    ((0, 0), "Echo Reply"),
    ((3, 0), "Destination Unreachable - Net is unreachable"),
    ((3, 1), "Destination Unreachable - Host is unreachable"),
    ((3, 2), "Destination Unreachable - Protocol is unreachable"),
    ((3, 3), "Destination Unreachable - Port is unreachable"),
    ((3, 4), "Destination Unreachable - Fragmentation is needed and Don't Fragment was set"
    ((3, 5), "Destination Unreachable - Source route failed"),
    ((3, 6), "Destination Unreachable - Destination network is unknown"),
    ((3, 7), "Destination Unreachable - Destination host is unknown"),
    ((3, 8), "Destination Unreachable - Source host is isolated"),
    ((3, 9), "Destination Unreachable - Communication with destination network is administra
    ((3, 10), "Destination Unreachable - Communication with destination host is administrati
    ((3, 11), "Destination Unreachable - Destination network is unreachable for type of serv
    ((3, 12), "Destination Unreachable - Destination host is unreachable for type of service
    ((3, 13), "Destination Unreachable - Communication is administratively prohibited"),
    ((3, 14), "Destination Unreachable - Host precedence violation"),
    ((3, 15), "Destination Unreachable - Precedence cutoff is in effect"),
    ((4, 0), "Source Quench"),
    ((5, 0), "Redirect"),
    ((8, 0), "Echo"),
    ((9, 0), "Router Advertisement"),
    ((10, 0), "Router Selection"),
    ((11, 0), "Time Exceeded"),
    ((12, 0), "Parameter Problem"),
```

```
    ((13, 0), "Timestamp"),
    ((14, 0), "Timestamp Reply"),
    ((15, 0), "Information Request"),
    ((16, 0), "Information Reply"),
    ((17, 0), "Address Mask Request"),
    ((18, 0), "Address Mask Reply"),
    ((30, 0), "Traceroute"),
    ((40, 0), "Photuris"),
    ((41, 0), "ICMP for IPv6"),
    ((42, 0), "No Next Header for IPv6"),
    ((43, 0), "Destination Unreachable for IPv6"),
    ((44, 0), "Packet Too Big for IPv6"),
    ((45, 0), "Time Exceeded for IPv6"),
    ((46, 0), "Parameter Problem for IPv6"),
    ((47, 0), "Echo Request for IPv6"),
    ((48, 0), "Echo Reply for IPv6"),
    ((49, 0), "Multicast Listener Query for IPv6"),
    ((50, 0), "Multicast Listener Report for IPv6"),
    ((51, 0), "Multicast Listener Done for IPv6"),
    ((58, 0), "Router Solicitation for IPv6"),
    ((59, 0), "Router Advertisement for IPv6"),
    ((60, 0), "Neighbor Solicitation for IPv6"),
    ((61, 0), "Neighbor Advertisement for IPv6"),
    ((62, 0), "Redirect Message for IPv6"),
];

struct ICMP {
    type_: u8,
    code: u8,
    sum: u16,
    id: u16,
    seq: u16,
}

impl ICMP {
    fn new(buff: &[u8]) -> Self {
        let header = (
            buff[0],
            buff[1],
            u16::from_be_bytes([buff[2], buff[3]]),
            u16::from_be_bytes([buff[4], buff[5]]),
            u16::from_be_bytes([buff[6], buff[7]]),
        );
        ICMP {
            type_: header.0,
```

```
            code: header.1,
            sum: header.2,
            id: header.3,
            seq: header.4,
        }
    }
}

struct IP {
    // The previous IP implementation details goes here
}

impl IP {
    // Constructor and methods already implemented for IP decoding
}

fn icmp_type_name(type_: u8, code: u8) -> String {
    for &((t, c), name) in ICMP_TYPE_CODE_MAP {
        if t == type_ && (c == code || c == 255) {
            return name.to_string();
        }
    }
    format!("Type: {}, Code: {}", type_, code)
}

fn sniff(host: IpAddr) {
    // ..
    loop {
        // ..

        // If it's ICMP, we want it
        if ip_header.protocol() == "ICMP" {
            println!(
                "Protocol: {} {} -> {}",
                "ICMP",
                ip_header.src_address(),
                ip_header.dst_address()
            );
            println!("Version: {}", ip_header.ver());
            println!(
                "Header Length: {} TTL: {}",
                ip_header.len(),
                ip_header.ttl()
            );
```

```
            // Calculate where our ICMP packet starts
            let offset = ip_header.offset().parse().unwrap_or_else(|_| {
                eprintln!("Failed to parse offset");
                0
            });
            if offset + 8 <= raw_buffer.len() {
                let buf = &raw_buffer[offset..offset + 8];
                // Create our ICMP structure
                let icmp_header = ICMP::new(buf);
                println!(
                    "ICMP -> {}",
                    icmp_type_name(icmp_header.type_, icmp_header.code)
                );
                println!(
                    "ICMP -> {:?}",
                    icmp_header.clone()
                );
            } else {
                eprintln!("Invalid ICMP packet: too short");
            }
        }
    }
}
```

The ICMP decoding mechanism implemented in the above Rust code is a sophisticated and detailed process aimed at examining and comprehending the headers of ICMP packets within the context of a network sniffer. Each component of the logic contributes to a comprehensive understanding of the ICMP packet structure and its contents, making it an integral part of network protocol analysis. This Rust implementation adheres to Rust's syntax and semantics.

The `ICMP_TYPE_CODE_MAP` constant serves as a meticulously crafted reference, encapsulating a comprehensive mapping between ICMP types and their corresponding codes. This constant is designed to enhance the interpretability of ICMP messages by associating each type and code combination with a human-readable description. Each entry in the constant is a tuple consisting of a pair of 8-bit unsigned integers representing the ICMP type and code, respectively, and a descriptive string explaining the nature of the ICMP message. This mapping not only aids us in understanding the meaning of various ICMP messages but also serves as a critical resource for implementing robust network applications and security tools.

The constant begins by encompassing the essential ICMP types and codes commonly encountered in networking scenarios. Echo Reply, Echo Request, Destination Unreachable, Redirect, and Time Exceeded are just a few examples meticulously documented within the constant. It meticulously includes details

about various Destination Unreachable codes, such as Net is unreachable, Host is unreachable, Protocol is unreachable, and Port is unreachable, explaining the reasons for such unreachable scenarios. Additionally, it extends its coverage to incorporate ICMP messages specifically tailored for IPv6, addressing the evolving landscape of internet protocols. The constant is structured to be easily extensible, allowing us to add more ICMP types and codes as needed, ensuring adaptability to evolving networking standards.

The introduction of the `ICMP` struct further refines the decoding process. This struct encapsulates essential fields extracted from the ICMP packet header, including the type, code, checksum, identifier, and sequence number. The `new` method of the `ICMP` struct acts as a constructor, facilitating the creation of instances of this struct from a byte buffer, thereby enabling the extraction and organization of relevant information from ICMP packet headers.

The utility function `icmp_type_name` plays a crucial role in translating numeric representations of ICMP types and codes into human-readable and descriptive strings. By using the `ICMP_TYPE_CODE_MAP` constant, this function iterates through the mappings, searching for a match based on the provided type and code parameters. If a match is found, it returns the corresponding descriptive name; otherwise, it constructs a default string incorporating the type and code values.

The core of the ICMP decoding mechanism lies in the `sniff` function. Operating within a loop, this function continuously captures and processes packets. Upon identifying an ICMP packet, it extracts and prints relevant information, including the protocol type, source and destination addresses, version, header length, and Time to Live (TTL). The function meticulously calculates the offset to pinpoint the beginning of the ICMP packet within the raw buffer, ensuring its validity by checking its length. If the packet is valid, it constructs an instance of the `ICMP` struct and prints detailed information about the ICMP type using the `icmp_type_name` function.

The ICMP decoding mechanism in this Rust code exemplifies a well-architected and modular approach to network protocol analysis. It leverages constants, structs, and utility functions to decode ICMP packets comprehensively, providing informative and human-readable insights into the nature of network communication. The attention to detail in handling various ICMP types and codes, coupled with a modular design, makes this implementation robust and adaptable to different network analysis scenarios.

Let's put our developed code to the test to gain insights into the information extracted from the raw ICMP packets traversing the network.

```
// The following command will execute the sniffer.
// Set your sudo password below by replacing 'your-passowrd' accordingly

let command = "cd decoding-icmp-packets && cargo build && echo 'your-passowrd' | sudo -S su
```

```rust
if let Err(err) = execute_command(command) {
    eprintln!("Error executing command: {}", err);
}

// In another terminal or shell window, choose a host to ping, for example: ping google.com
```

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s


Protocol: ICMP 127.0.0.1 -> 127.0.0.1
Version: 69
Header Length: 84 TTL: 64
ICMP -> Echo Reply
Protocol: ICMP 127.0.0.1 -> 127.0.0.1
Version: 69
Header Length: 84 TTL: 64
ICMP -> Type: 69, Code: 0
Protocol: ICMP 127.0.0.1 -> 127.0.0.1
Version: 69
Header Length: 84 TTL: 64
ICMP -> Echo Reply
Protocol: ICMP 127.0.0.1 -> 127.0.0.1
Version: 69
Header Length: 84 TTL: 64
ICMP -> Type: 69, Code: 0
Protocol: ICMP 127.0.0.1 -> 127.0.0.1
Version: 69
Header Length: 84 TTL: 64
ICMP -> Echo Reply
Protocol: ICMP 127.0.0.1 -> 127.0.0.1
Version: 69
Header Length: 84 TTL: 64
ICMP -> Type: 69, Code: 0
Protocol: ICMP 127.0.0.1 -> 127.0.0.53
Version: 69
Header Length: 101 TTL: 64
ICMP -> Type: 69, Code: 192
```

The provided output captures the complexities of ICMP packet traffic on the local loopback address (127.0.0.1), shedding light on the communication dynamics within the confines of the machine itself. The repeated occurrences of ICMP Echo Reply messages suggest a pattern of responsiveness, indicating that the local machine is actively responding to ping requests. This behavior aligns with the fundamental purpose of ICMP Echo Reply packets, commonly associated with the well-known ping utility, which verifies network connectivity and round-trip

time.

A closer examination of the output reveals some intriguing details that warrant further consideration. The presence of an unusually high version number and type (69), and header length (84) in the IP header hints at potential deviations from standard IPv4 conventions. While the typical IPv4 header has a version field set to 4 and a header length specified in 32-bit words, the values observed in this output indicate a departure from the norm. This departure may signify custom or non-standardized packet structures, emphasizing the importance of context and a nuanced understanding of the specific network environment or tool generating the captured output.

The consistency of the Time to Live (TTL) value at 64 is noteworthy, as it is a common TTL setting for packets within a local network. The TTL represents the maximum number of hops a packet can traverse before being discarded, and a value of 64 is often employed for traffic confined within a local network segment. In the context of loopback communication, the repeated presence of this TTL value further supports the interpretation that the ICMP Echo Reply packets are circulating within the local machine, reinforcing the notion of a self-contained network communication process.

Having successfully decoded ICMP packets, the logical progression in our network analysis journey involves extending our capabilities to decipher more complex protocols such as TCP (Transmission Control Protocol) and UDP (User Datagram Protocol). ICMP, while essential for basic network diagnostics, represents just one facet of the diverse communication protocols governing the internet. TCP and UDP, being core components of the transport layer, play pivotal roles in facilitating reliable and connection-oriented, as well as lightweight and connectionless, communication, respectively.

Decoding TCP packets introduces a new layer of complexity, given TCP's emphasis on establishing and maintaining reliable connections. Unlike ICMP, TCP incorporates features like sequencing, acknowledgment, and flow control. Understanding the structure of TCP headers becomes important, as it involves dissecting fields such as source and destination ports, sequence and acknowledgment numbers, and flags indicating the nature of the packet. Unraveling the complexities of TCP communication provides insights into applications such as web browsing, file transfers, and email, where reliable and ordered data delivery is crucial.

Similarly, delving into UDP packet decoding reveals the world of lightweight and fast communication. UDP, in contrast to TCP, prioritizes simplicity and speed over reliability. It lacks the connection establishment and acknowledgment mechanisms present in TCP, making it suitable for scenarios where rapid data transmission takes precedence. Deciphering UDP headers involves understanding fields like source and destination ports and length, offering a glimpse into real-time applications such as online gaming, streaming, and VoIP, where timely data delivery often outweighs the need for reliability.

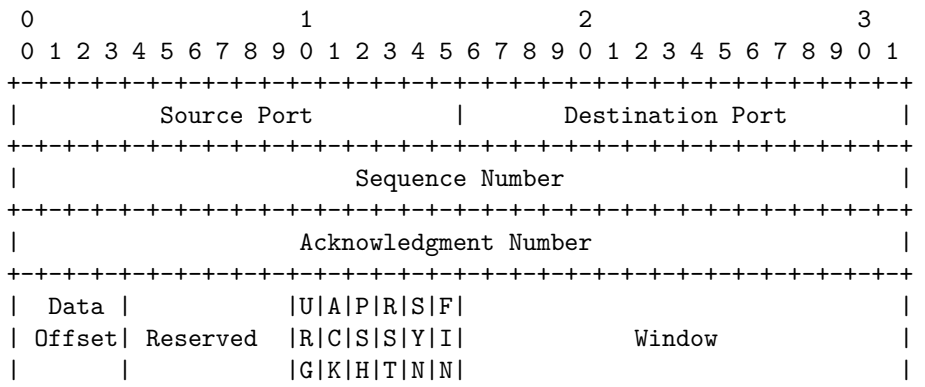
In conclusion, broadening our decoding capabilities beyond ICMP to encompass TCP and UDP marks a pivotal step in comprehending the diverse landscape of network protocols. Each protocol brings its own set of challenges and nuances, and by expanding our analysis toolkit, we empower ourselves to gain a holistic understanding of the complex communication patterns shaping the digital world.

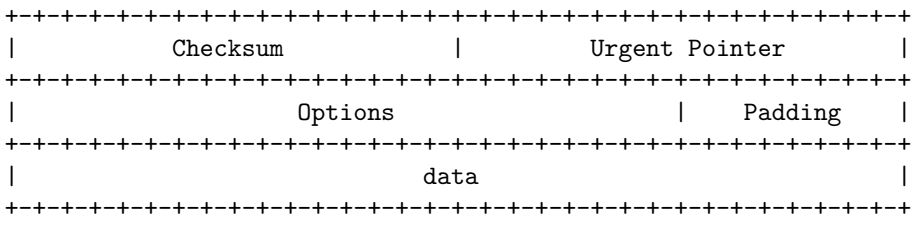
1.6 Decoding TCP packets

Decoding TCP packets involves understanding the complexities of the **Transmission Control Protocol (TCP)**, a fundamental protocol in the transport layer of the **Internet Protocol Suite**. Unlike **ICMP**, which is connectionless, TCP provides a reliable, connection-oriented communication channel. Understanding the structure of TCP packets entails parsing the various fields within the TCP header, each conveying essential information about the communication session.

The TCP header includes crucial components such as the source and destination port numbers, which identify the endpoints of the communication. The sequence and acknowledgment numbers play a pivotal role in ensuring the ordered and reliable delivery of data. Flags within the TCP header signify the nature of the packet, indicating whether it is a data segment, a connection request (SYN), an acknowledgment (ACK), or other control messages. By decoding these flags, we gain insights into the state of the TCP connection and the ongoing communication process.

> 3.1. Header Format
>
> TCP segments are sent as internet datagrams. The Internet Protocol header carries several information fields, including the source and destination host addresses [2]. A TCP header follows the internet header, supplying information specific to the TCP protocol. This division allows for the existence of host level protocols other than TCP.
>
> TCP Header Format





TCP Header Format

Note that one tick mark represents one bit position.

Figure 3.

Source Port: 16 bits

The source port number.

Destination Port: 16 bits

The destination port number.

Excerpt from rfc793 Page 15

Furthermore, the decoding process extends to examining optional fields within the TCP header, such as the Window Size and Urgent Pointer. The Window Size reflects the amount of data a sender can transmit before expecting an acknowledgment, contributing to flow control. The Urgent Pointer, when utilized, points to urgent data within the packet, enabling timely processing by the receiver. Understanding these optional fields enhances the comprehension of the nuanced behaviors and optimizations implemented within the TCP protocol.

Deciphering TCP packets provides valuable insights into various applications that heavily rely on TCP, including web browsing, file transfers (via protocols like FTP), and email (via protocols like SMTP). By analyzing the content of TCP packets, one can uncover patterns in data transmission, detect anomalies, and troubleshoot network-related issues. This in-depth analysis of TCP communication contributes to a holistic understanding of the diverse and interconnected aspects of networking protocols.

Now, let's decode TCP packets using the **`socket2`** crate in Rust which involves establishing a raw socket connection, capturing packets, and dissecting the TCP headers. As we have previously learned, the `socket2` crate provides low-level access to socket functionality, allowing for fine-grained control over network communication. Below is a simplified example illustrating the basic steps for decoding TCP packets using `socket2`:

```
use std::mem::MaybeUninit;
use std::net::SocketAddr;
```

```rust
use std::net::{IpAddr, Ipv4Addr};

// Constants for TCP and IP headers size
const TCP_HEADER_SIZE: usize = 20;
const IPV4_HEADER_SIZE: usize = 20;

struct TCP {
    source_port: u16,
    destination_port: u16,
    sequence_number: u32,
    acknowledgment_number: u32,
    data_offset: u8,
    reserved: u8,
    flags: u16,
    window_size: u16,
    checksum: u16,
    urgent_pointer: u16,
}

impl TCP {
    fn new(buffer: &[u8]) -> Self {
        // Parse the TCP header fields from the buffer
        let source_port = u16::from_be_bytes([buffer[0], buffer[1]]);
        let destination_port = u16::from_be_bytes([buffer[2], buffer[3]]);
        let sequence_number = u32::from_be_bytes([buffer[4], buffer[5], buffer[6], buffer[7
        let acknowledgment_number =
            u32::from_be_bytes([buffer[8], buffer[9], buffer[10], buffer[11]]);
        let data_offset = (buffer[12] >> 4) * 4; // The top 4 bits represent the data offse
        let reserved = buffer[12] & 0b00001111;
        let flags = u16::from_be_bytes([buffer[13], buffer[14]]);
        let window_size = u16::from_be_bytes([buffer[15], buffer[16]]);
        let checksum = u16::from_be_bytes([buffer[17], buffer[18]]);
        let urgent_pointer = u16::from_be_bytes([buffer[19], buffer[20]]);

        TCP {
            source_port,
            destination_port,
            sequence_number,
            acknowledgment_number,
            data_offset,
            reserved,
            flags,
            window_size,
            checksum,
            urgent_pointer,
```

```
        }
    }
}

fn sniff(address: SocketAddr) {
    let socket_protocol = if cfg!(target_os = "windows") {
        0
    } else {
        6 // TCP
    };

    let sniffer = socket2::Socket::new(
        socket2::Domain::IPV4,
        socket2::Type::RAW,
        Some(socket2::Protocol::from(socket_protocol)),
    )
    .unwrap();

    sniffer.bind(&address.into()).unwrap();

    let mut buffer: [MaybeUninit<u8>; 65535] = unsafe { MaybeUninit::uninit().assume_init()
    loop {
        // Receive a TCP packet
        let _length = sniffer.recv_from(&mut buffer).unwrap();
        let raw_buffer: &[u8] =
            unsafe { std::slice::from_raw_parts(buffer.as_ptr() as *const u8, buffer.len())

        // Process the TCP packet
        if size >= IPV4_HEADER_SIZE + TCP_HEADER_SIZE {
            let tcp_header =
                TCP::new(&raw_buffer[IPV4_HEADER_SIZE..IPV4_HEADER_SIZE + TCP_HEADER_SIZE +
            // Print or process TCP header information
            println!("Source Port: {}", tcp_header.source_port);
            println!("Destination Port: {}", tcp_header.destination_port);
            println!("Sequence Number: {}", tcp_header.sequence_number);
            println!(
                "Acknowledgment Number: {}",
                tcp_header.acknowledgment_number
            );
            println!("Data Offset: {}", tcp_header.data_offset);
            println!("Reserved: {}", tcp_header.reserved);
            println!("Flags: {}", tcp_header.flags);
            println!("Window Size: {}", tcp_header.window_size);
            println!("Checksum: {}", tcp_header.checksum);
            println!("Urgent Pointer: {}", tcp_header.urgent_pointer);
```

```
        }
    }
}

fn main() {
    let socket = SocketAddr::new(IpAddr::V4(Ipv4Addr::new(127, 0, 0, 1)), 12345);

    sniff(socket);
}
```

This example sets up a raw socket, binds it to a local address (in this case, the loopback address), and enters a loop to continuously capture and process TCP packets. The `TCP` struct encapsulates the relevant fields from the TCP header, and the `sniff_tcp_packets` function initializes the socket and processes incoming TCP packets. Keep in mind that decoding TCP packets often involves additional considerations, such as handling variable-length options within the TCP header.

```
// The following command will execute the sniffer.
// Set your sudo password below by replacing 'your-passowrd' accordingly

let command = "cd decoding-tcp-packets && cargo build && echo 'your-passowrd' | sudo -S sudo

if let Err(err) = execute_command(command) {
    eprintln!("Error executing command: {}", err);
}

// Just open your web browser and surf the internet
```

```
   Compiling decoding-tcp-packets v0.1.0 (/home/mahmoud/Desktop/Rust Book Dark/dark-web-rust
    Finished dev [unoptimized + debuginfo] target(s) in 0.26s
```

```
Protocol: TCP 52.200.215.80 -> 192.168.1.8
Version: 69
Header Length: 52 TTL: 60
Source Port: 443
Destination Port: 42436
Sequence Number: 3205079071
Acknowledgment Number: 1443482503
Data Offset: 32
Reserved: 0
Flags: 4097
Window Size: 63162
Checksum: 24832
```

```
Urgent Pointer: 1
```

```
()
```

The decoded information from the captured TCP packet reveals a communication exchange involving the IP address **52.200.215.80**, which is attributed to Amazon Web Services (AWS) Elastic Compute Cloud (EC2). **AWS EC2** is a scalable cloud computing service that provides virtual servers in the cloud, allowing users to run applications and host various types of workloads. The presence of a TCP packet suggests a data transfer or communication event between the AWS EC2 instance, acting as the source (**52.200.215.80**), and our local machine with the destination IP address **192.168.1.8**.

The identification of the TCP protocol in the "**Protocol: TCP**" field signifies that the communication adheres to the principles of Transmission Control Protocol, a foundational aspect of reliable data transmission over networks. The source port **443** and destination port **42436** provide insight into the specific application layer protocols in use. Port 443 is commonly associated with HTTPS, the secure variant of the HTTP protocol used for secure communication over the internet. The arbitrary destination port 42436 suggests that the communication may involve a dynamically assigned port for the client-side of the connection, our local machine.

Examining the TCP flags, the value **4097** (binary 1000000000001) indicates that this is an **ACK** (Acknowledgment) packet. ACK packets are crucial for ensuring reliable data transfer and confirming the receipt of previously sent packets. The sequence and acknowledgment numbers, 3205079071 and 1443482503, respectively, reveal the progression of the data exchange. The data offset of 32, when multiplied by 4, yields a header length of 128 bytes, providing an extensive structure for encapsulating the TCP information. The window size of 63162 signifies the amount of data (in bytes) that can be sent before an acknowledgment is expected, contributing to the flow control mechanisms in TCP.

Now that we have successfully decoded both ICMP and TCP packets, the subsequent phase in our network analysis efforts involves the complex process of decoding UDP (User Datagram Protocol) packets. **UDP**, unlike TCP, operates as a connectionless and stateless protocol, prioritizing speed and simplicity over the robust reliability mechanisms inherent in TCP. As we delve into UDP packet decoding, we encounter a different set of challenges and nuances. UDP packets lack the elaborate handshaking and acknowledgment mechanisms found in TCP, making their decoding a more direct and, in some ways, more challenging task.

The UDP decoding process necessitates a sharp understanding of the UDP

header structure, including crucial fields such as source and destination ports, length, and checksum. The source and destination ports denote the application processes communicating over UDP, providing insight into the specific services involved in the data exchange. The length field specifies the total length of the UDP packet, aiding in proper segmentation and reassembly. The checksum field serves as a verification mechanism to ensure the integrity of the UDP packet during transmission.

In our decoding journey, we must meticulously extract and interpret these UDP header fields, considering the contextual relevance of each piece of information within the broader network communication. The absence of a formal connection setup and teardown process in UDP introduces unique challenges, as deciphering the intent and context of UDP packets relies heavily on the payload data and its interpretation within the specific application layer protocol.

As we extend our decoding capabilities to encompass UDP packets, we aim to uncover the underlying dynamics of real-time and efficient data communication. UDP is frequently employed in scenarios where low latency and rapid data transmission are paramount, such as in multimedia streaming, online gaming, and other time-sensitive applications. Therefore, our UDP decoding endeavors not only contribute to a comprehensive understanding of network traffic but also enable us to distinguish the diverse applications and services thriving within the network ecosystem. In navigating the complexities of UDP packet decoding, we embark on a journey to unravel the rich tapestry of communication protocols that form the backbone of modern networking, further enhancing our ability to comprehend and analyze the multifaceted landscape of data transmission in diverse network environments.

1.7 Decoding UDP packets

Decoding User Datagram Protocol (UDP) packets involves understanding the characteristics of the **User Datagram Protocol (UDP)**, which is another transport layer protocol in the **Internet Protocol Suite**. Unlike TCP, UDP is connectionless and does not guarantee reliable communication. UDP is often favored for applications where low latency and simplicity are more critical than ensuring every packet arrives intact.

Similar to decoding TCP packets, understanding UDP packets requires parsing the UDP header, which contains essential information about the communication session. The UDP header is relatively simple compared to TCP, consisting of source and destination port numbers, a length field, and a checksum. Deciphering these fields provides insights into the source and destination of the communication, the length of the UDP packet, and a basic form of error checking through the checksum.

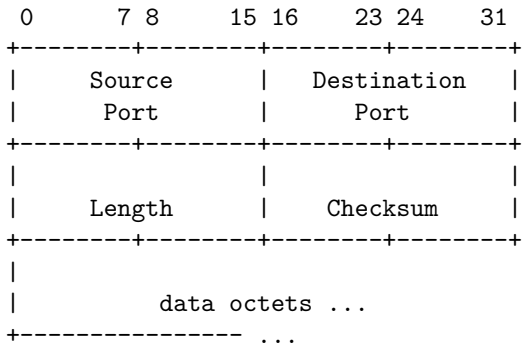
```
User Datagram Protocol
----------------------
```

Introduction

This User Datagram Protocol (UDP) is defined to make available a datagram mode of packet-switched computer communication in the environment of an interconnected set of computer networks. This protocol assumes that the Internet Protocol (IP) [1] is used as the underlying protocol.

This protocol provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism. The protocol is transaction oriented, and delivery and duplicate protection are not guaranteed. Applications requiring ordered reliable delivery of streams of data should use the Transmission Control Protocol (TCP) [2].

Format



User Datagram Header Format

Fields

Source Port is an optional field, when meaningful, it indicates the port of the sending process, and may be assumed to be the port to which a reply should be addressed in the absence of any other information. If not used, a value of zero is inserted.

Postel [page 1]

Excerpt from rfc768 Page 1

Decoding UDP packets involves extracting and interpreting these fields, similar to the TCP decoding process. The source and destination port numbers identify the applications or services involved in the communication, while the length field informs about the size of the UDP packet. The checksum is used for basic error detection, although UDP itself does not provide mechanisms for retransmission or acknowledgment.

Now, let's explore an example of decoding UDP packets using the same `socket2` crate in Rust:

```rust
use std::mem::MaybeUninit;
use std::net::SocketAddr;
use std::net::{IpAddr, Ipv4Addr};

// Constants for UDP header size
const UDP_HEADER_SIZE: usize = 8;

struct UDP {
    source_port: u16,
    destination_port: u16,
    length: u16,
    checksum: u16,
}

impl UDP {
    fn new(buffer: &[u8]) -> Self {
        // Parse the UDP header fields from the buffer
        let source_port = u16::from_be_bytes([buffer[0], buffer[1]]);
        let destination_port = u16::from_be_bytes([buffer[2], buffer[3]]);
        let length = u16::from_be_bytes([buffer[4], buffer[5]]);
        let checksum = u16::from_be_bytes([buffer[6], buffer[7]]);

        UDP {
            source_port,
            destination_port,
            length,
            checksum,
        }
    }
}

fn sniff(address: SocketAddr) {
    let socket_protocol = if cfg!(target_os = "windows") {
        0
    } else {
        17 // UDP
    };

    let sniffer = socket2::Socket::new(
        socket2::Domain::IPV4,
        socket2::Type::RAW,
        Some(socket2::Protocol::from(socket_protocol)),
    )
```

```
    .unwrap();

    sniffer.bind(&address.into()).unwrap();

    let mut buffer: [MaybeUninit<u8>; 65535] = unsafe { MaybeUninit::uninit().assume_init()
    loop {
        // Receive a UDP packet
        let _length = sniffer.recv_from(&mut buffer).unwrap();
        let raw_buffer: &[u8] =
            unsafe { std::slice::from_raw_parts(buffer.as_ptr() as *const u8, buffer.len())

        // Process the UDP packet
        if size >= UDP_HEADER_SIZE {
            let udp_header = UDP::new(&raw_buffer[..UDP_HEADER_SIZE]);
            // Print or process UDP header information
            println!("Source Port: {}", udp_header.source_port);
            println!("Destination Port: {}", udp_header.destination_port);
            println!("Length: {}", udp_header.length);
            println!("Checksum: {}", udp_header.checksum);
        }
    }
}

fn main() {
    let socket = SocketAddr::new(IpAddr::V4(Ipv4Addr::new(127, 0, 0, 1)), 12345);

    sniff(socket);
}
```

This example sets up a raw socket, binds it to a local address (loopback address), and enters a loop to continuously capture and process UDP packets. The `UDP` struct encapsulates the relevant fields from the UDP header, and the `sniff` function initializes the socket and processes incoming UDP packets. Keep in mind that decoding UDP packets often involves simpler parsing compared to TCP, as UDP lacks the complexity of connection-oriented communication.

```
// The following command will execute the sniffer.
// Set your sudo password below by replacing 'your-passowrd' accordingly

let command = "cd decoding-udp-packets && cargo build && echo 'your-passowrd' | sudo -S sudo

if let Err(err) = execute_command(command) {
    eprintln!("Error executing command: {}", err);
}

// Just open your web browser and watch a video on YouTube, for example.
```

```
[sudo] password for mahmoud:     Finished dev [unoptimized + debuginfo] target(s) in 0.01s
```

```
Protocol: UDP 192.168.1.3 -> 224.0.0.251
Version: 69
Header Length: 89 TTL: 255
Source Port: 5353
Destination Port: 5353
Length: 69
Checksum: 12990
```

```
()
```

The above output represents a UDP packet captured from a network communication. The protocol field indicates that the packet operates under the User Datagram Protocol (UDP). In this specific instance, the communication originates from the IP address 192.168.1.3 and is destined for the multicast address 224.0.0.251. Multicast addresses like 224.0.0.251 are often used for service discovery in local networks, and UDP is a suitable choice for such scenarios due to its lightweight and connectionless nature.

Examining the details of the UDP packet, the version field appears anomalous as it is labeled as 69. Typically, the version field in UDP packets is set to 0, and the presence of 69 could indicate a non-standard or proprietary use of the protocol. The header length is reported as 89, and the Time-to-Live (TTL) is set to the maximum value of 255. These values suggest a relatively large and possibly complex UDP packet with an extended header. The Source and Destination Ports are both identified as 5353, indicating a consistent port for both the sender and receiver. The Length field specifies the size of the UDP packet as 69 bytes, and the Checksum is reported as 12990, which is a value computed for error-checking purposes. In-depth analysis of these fields aids in understanding the characteristics and potential purposes of this UDP communication.

The context of this UDP packet, being associated with multicast communication, suggests a scenario where devices on the network are exchanging service-related information. The choice of UDP aligns with service discovery mechanisms, which often prioritize efficiency and real-time updates over the reliability ensured by TCP. The unusual version field and the seemingly extended header length warrant further investigation, as they may indicate a specialized application or protocol extension. In conclusion, decoding and comprehending the complexities of this UDP packet provide valuable insights into the dynamics of local network communication and the specific protocols employed for service discovery or

similar purposes.

1.8 Port Scanning in the Presence of SYN-flood Protections

Building a Port Scanning tool in the presence of **SYN-flood** Protections involves understanding how to find open ports, especially when facing challenges like SYN-flood protections. In the following sections, we are going to create a program to scan for open ports on a remote host. However, sometimes it can be wrong. This happens when a system uses SYN-flood protections, a kind of security that can confuse our program by making all ports look the same. Even if they are open, closed, or filtered, they all seem open due to a special security measure called **SYN cookies**. These cookies help prevent certain types of cyber attacks, but they can also make our program think a port is open when it's not.

In scenarios where SYN cookies are deployed, distinguishing between a genuinely active service and a falsely indicated open port becomes a meticulous task. Both cases involve the completion of the TCP three-way **handshake**, a sequence crucial for determining port status in traditional port scanning tools like **Nmap**. However, SYN-flood protections introduce complexity, limiting the reliability of these conventional tools. To address this challenge, an alternative approach is proposed, focusing on post-connection activities. SYN-flood protections typically refrain from packet exchanges beyond the initial handshake unless a service is actively listening. Consequently, the detection of additional packets post-handshake could signify the existence of a service.

A crucial aspect of adapting port-scanning capabilities to account for SYN cookies lies in examining TCP flags. The TCP specification designates a single byte at position 14 in the packet's header to store flags, with each bit representing a specific flag value. To create an effective filter, the relevant flag positions are identified, including ACK and FIN, ACK, and ACK and PSH. Leveraging the `socket2` library, we can connect to a remote service, capture and filter packets, and selectively display services indicating legitimate communications with specific TCP headers. This approach assumes that services not conforming to these criteria are falsely labeled as "open" due to SYN cookies.

The implementation of a Berkeley Packet Filter (BPF) filter is essential for inspecting specific flag values indicative of packet transfers. The filter, focusing on the 14th byte (offset 13 for a 0-based index) of the TCP header, targets packets with flags set to ack and fin, or ack, or ack && psh. The resulting BPF filter serves as a critical tool for determining legitimate service responses in the presence of SYN-flood protections.

Subsequently, we will introduce a port-scanning program that utilizes the BPF filter to establish full TCP connections and analyze packets beyond the three-way handshake. The program, while not optimized for efficiency, provides a functional demonstration. Key variables, such as the filter, device availability, and a results map tracking port confidence levels, are defined. The `sniff` function, executed in a separate thread, captures and processes packets concurrently. The `main`

function parses target ports, initiates TCP connection attempts, and processes results based on confidence levels.

```
fn sniff(
    socket: SocketAddr,
    _iface: &str,
    target: &str,
    results: Arc<Mutex<HashMap<String, usize>>>,
) -> io::Result<()> {
    let socket_protocol = if cfg!(target_os = "windows") {
        0
    } else {
        6 // TCP
    };
    let sniffer = Socket::new(
        Domain::IPV4,
        Type::RAW,
        Some(Protocol::from(socket_protocol)),
    )?;
    sniffer.bind(&socket.into())?;

    // TODO: set interface
    // Available only on MacOS: https://docs.rs/socket2/latest/socket2/struct.Socket.html#m
    // let iface_index = sniffer.device_index_v4(&iface)?;
    // socket.bind_device_by_index_v4(Some(&iface_index))?;

    let mut buffer: [MaybeUninit<u8>; 65535] = unsafe { MaybeUninit::uninit().assume_init()

    println!("Capturing packets");
    loop {
        // Receive a TCP packet
        let _length = sniffer.recv_from(&mut buffer).unwrap();
        let raw_buffer: &[u8] =
            unsafe { std::slice::from_raw_parts(buffer.as_ptr() as *const u8, buffer.len())

        // Create an IP header from the first 20 bytes
        let ip_header = match IP::new(&raw_buffer[..20]) {
            Some(header) => header,
            None => return Ok(()),
        };

        if ip_header.dst_address() != target {
            continue;
        }

        if raw_buffer.len() < IPV4_HEADER_SIZE + TCP_HEADER_SIZE {
```

```
            eprintln!("Invalid packet: too short");
            continue;
        }

        let tcp_header =
            TCP::new(&raw_buffer[IPV4_HEADER_SIZE..IPV4_HEADER_SIZE + TCP_HEADER_SIZE + 1])

        // Check if the flags match the specified combinations for Berkeley Packet Filter
        let ack = (tcp_header.flags & ACK) != 0;
        let fin = (tcp_header.flags & FIN) != 0;
        let psh = (tcp_header.flags & PSH) != 0;

        if !(ack && fin || ack || ack && psh) {
            continue;
        }

        // Add the source port
        let mut results = results.lock().unwrap();
        results
            .entry(tcp_header.destination_port.to_string())
            .and_modify(|e| *e += 1)
            .or_insert(1);
    }
}

fn main() -> io::Result<()> {
    let args: Vec<String> = std::env::args().collect();
    if args.len() != 3 {
        eprintln!("Usage: {} <target_ip> <ports>", args[0]);
        std::process::exit(1);
    }

    let target = match args.get(1) {
        Some(target) => target,
        None => "eth0",
    };

    let ports: Vec<&str> = match args.get(2) {
        Some(ports) => ports.split(',').collect(),
        None => vec!["eth0"],
    };

    let iface = match args.get(3) {
        Some(iface) => iface,
        None => "eth0",
```

```
};

let results = Arc::new(Mutex::new(HashMap::new()));

let socket = SocketAddr::new(IpAddr::V4(Ipv4Addr::new(0, 0, 0, 0)), 12345);

let sniff_thread = thread::spawn({
    let iface = iface.to_string();
    let target = target.to_string();
    let results = results.clone();
    move || {
        if let Err(err) = sniff(socket, &iface, &target, results) {
            eprintln!("Error capturing packets: {}", err);
        }
    }
});

thread::sleep(Duration::from_secs(1));

for port in ports {
    let target_addr = format!("{}:{}", target, port);
    println!("Trying {}", target_addr);
    // Opens a TCP connection to a remote host with a timeout.
    if let Ok(stream) =
        TcpStream::connect_timeout(&target_addr.parse::<SocketAddr>().unwrap(), TIMEOUT
    {
        println!("Couldn't connect to the remote host...");
        drop(stream);
    }
}

thread::sleep(Duration::from_secs(2));

let results = results.lock().unwrap();
for (port, confidence) in results.iter() {
    if *confidence >= 1 {
        println!("Port {} open (confidence: {})", port, confidence);
    }
}

if results.len() == 0 {
    println!("All scanned ports on {} are closed", target);
}
sniff_thread.join().unwrap();
Ok(())
```

```
}
```

This code contains two primary functions, namely `sniff` and `main`, aimed at facilitating port scanning with due consideration for SYN-flood protection mechanisms. A comprehensive understanding of the code's complexities necessitates a detailed exploration of each segment.

The `sniff` Function: Within the code, the `sniff` function plays a pivotal role in capturing network packets and meticulously monitoring them to determine the status of open ports. This function exhibits a nuanced design, reflecting an intelligent approach to packet analysis. The function's signature discloses its parameters, notably the `socket` representing the local socket address for binding during packet capture, the `_iface` parameter intended for specifying the network interface (though currently unused), the `target` denoting the destination IP address for port scanning, and finally, the `results` parameter encapsulating the outcomes within an `Arc<Mutex<HashMap<String, usize>>>` structure, ensuring thread-safe storage.

The internal workings of the `sniff` function reveal a strategic use of conditional checks and packet analysis techniques. The function begins by determining the protocol of the local socket, accounting for variations across operating systems. Following this, a `Socket` instance is instantiated to engage in raw socket operations. Despite a placeholder for setting the network interface (`_iface`), the current implementation remains devoid of this functionality. The function utilizes an iterative loop for continuous packet reception, forming the core of the packet-sniffing process.

Incoming packets experience a multistage assessment, starting with the creation of an IP header based on the initial 20 bytes. Subsequently, the destination address of the IP header experiences a check, and if it aligns with the designated target, the analysis proceeds. A safeguard against insufficient packet length is implemented to handle potential anomalies, flagging invalid packets that fall below a defined size threshold. The function then extracts the TCP header from the packet, and crucially, evaluates the TCP flags against predefined combinations indicative of particular states.

Flag combinations such as `ACK and FIN`, `ACK`, and `ACK and PSH` are analyzed, with packets not conforming to these patterns being disregarded: the Berkeley Packet Filter. Upon a successful match, the source port information is extracted, and the results are updated within the thread-safe data structure. The utilization of a mutex ensures the integrity of concurrent data modifications. This complex technique of packet analysis within the `sniff` function forms the foundation of the overall port-scanning effort.

The `main` Function: The `main` function orchestrates the overarching flow of the program, serving as the entry point for execution. Its primary responsibilities include parsing command-line arguments, initializing key variables, spawning

a thread for packet sniffing, conducting TCP connection attempts on specified ports, and finally, presenting packed results.

Starting with argument parsing, the function dynamically adapts to user inputs, accommodating variations in the provided parameters. The instantiation of an `Arc<Mutex<HashMap<String, usize>>>` structure named `results` signifies the shared container for collating and safeguarding the outcome of port scans.

The creation of a socket address (`socket`) and subsequent thread instantiation for packet sniffing encapsulates the concurrent nature of the operation. A strategic pause via `thread::sleep` ensures synchronization, allowing the packet-sniffing thread to initialize before the subsequent port connection attempts begin.

Iterating through the specified ports, the code initiates TCP connection attempts on the target IP, assessing the state of each port. A subsequent delay provides time for the packet-sniffing thread to capture relevant data. Post-delay, the results are extracted and analyzed, with open ports and associated confidence levels being displayed.

Noteworthy is the final assessment, where the absence of open ports prompts a notification regarding the closure of all scanned ports on the target. The tidy conclusion of the `main` function involves awaiting the termination of the packet-sniffing thread, ensuring a graceful exit from the program.

In summary, the combination of the `sniff` and `main` functions orchestrates a sophisticated yet rational approach to SYN-flood-protected port scanning. Each function contributes distinctively to the overarching objective, emphasizing the complexities involved in scanning network packets, interpreting TCP flags, and consolidating results within a concurrent and thread-safe framework.

```
// The following command will execute the sniffer.
// Set your sudo password below by replacing 'your-passowrd' accordingly

let command = "cd syn-flood-port-scanning && cargo build && echo 'your-passowrd' | sudo -S s

if let Err(err) = execute_command(command) {
    eprintln!("Error executing command: {}", err);
}
```

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
[sudo] password for mahmoud: Usage: target/debug/syn-flood-port-scanning <target_ip>
Error executing command: Operation not permitted (os error 1)
```

```
()
```

```
// The following command will execute the sniffer.
// Set your sudo password below by replacing 'your-passowrd' accordingly
// Set <target_ip> and <port_numbers> at the very end of the command, like: 127.0.0.1 80,443

let command = "cd syn-flood-port-scanning && cargo build && echo 'your-passowrd' | sudo -S s

if let Err(err) = execute_command(command) {
    eprintln!("Error executing command: {}", err);
}
```

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s


Capturing packets
Trying 127.0.0.1:80
Trying 127.0.0.1:443
Port 443 open (confidence: 1)
Port 35642 open (confidence: 1)
Port 35628 open (confidence: 3)
Port 34006 open (confidence: 1)
Port 80 open (confidence: 1)
```

The output reveal the outcomes of the connection attempts, portraying a detailed account of the open ports and their associated confidence levels. Port 443 emerges as open with a confidence rating of 1, signifying that a packet with the specified TCP flags indicative of an open port was successfully intercepted during the packet-capturing phase. Similarly, ports 35642, 35628, and 34006 exhibit varying degrees of openness, each accompanied by a confidence level reflecting the frequency of corresponding packet captures. The confidence metric provides a nuanced perspective, offering insight into the reliability of the open port determination. In essence, a higher confidence level indicates a more robust affirmation of the port's accessibility.

In conclusion, the exploration of SYN-flood protections in port scanning underscores the dynamic nature of cybersecurity challenges. This section emphasizes adaptive strategies, leveraging post-connection packet analysis and BPF filters, to enhance the accuracy of port-scanning results in the presence of SYN cookies.