COMP 7003
Assignment 2

Design

Shayan Zahedanaraki
A01277365
Feb 2th, 2025

Purpose

- This program accepts 3 arguments from the command line:
    - -c <count>
    - -i <interface>
    - -f <filter>
- It prints the <message> to the console <count> times.

Data Types

Arguments

Purpose: To hold the unparsed command-line argument information

Field	Type	Description
interface	string	The name of the program
count	integer	The number of packets to capture and display
filter	string	The protocol or ethernet type filter captured and displayed packets

Context

Purpose: To hold the arguments, settings, and exit information

Field	Type	Description
arguments	Arguments	The command line arguments
exit_code	integer	The exit code of the program
exit_message	string	The error message to print before exiting

Functions

Function	Description
capture_on_all_interfaces	Sets up threads to begin capturing on all network interfaces with global ip.
capture_packets	Captures packets using Scapy’s sniffer and calls the specified

	callback.
has_global_ip	Checks if the network interface passed to it has a global ip.
interface_is_loopback	Checks if a network interface is a loopback network interface.
packet_callback	Handles captured packets and gives the hex to parse_ethernet_header
parse_ethernet_header	Parses ethernet header, prints it, and calls ethertype handling functions
parse_ip_header	Parses the ip ethertype header, prints it, and calls protocol handling functions
parse_icmp_header	Parses the icmp protocol header, and prints it
parse_tcp_header	Parses the tcp protocol header and prints it
parse_udp_header	Parses the udp protocol header, prints it, and calls parse_dns_header
parse_dns_header	Parses the dns protocol header, prints it
parse_arp_header	Parses the arp ethertype header, prints it
hex_to_mac	Converts

States

State	Description
start	Initial state. Immediately transitions to parse args.
parse args	Parses command-line arguments (interface, filter, count). Transitions to interface decision.
interface decision	Decides which interface to use. If the option is "any", transitions to capture on all interfaces; if "Other", transitions to has global ip other.
has global ip other	Checks if the selected (non-"any") interface has a global IP. If Yes, transitions to capture packets; if No, transitions to done.
capture on all interfaces	Starts threads for capturing packets on all interfaces. Transitions to is loopback.
is loopback	Determines if the interface is loopback. If Yes, loops back to capture on all interfaces; if No, transitions to has global ip.
has global ip	Checks for a global IP on a non-loopback interface. If Yes,

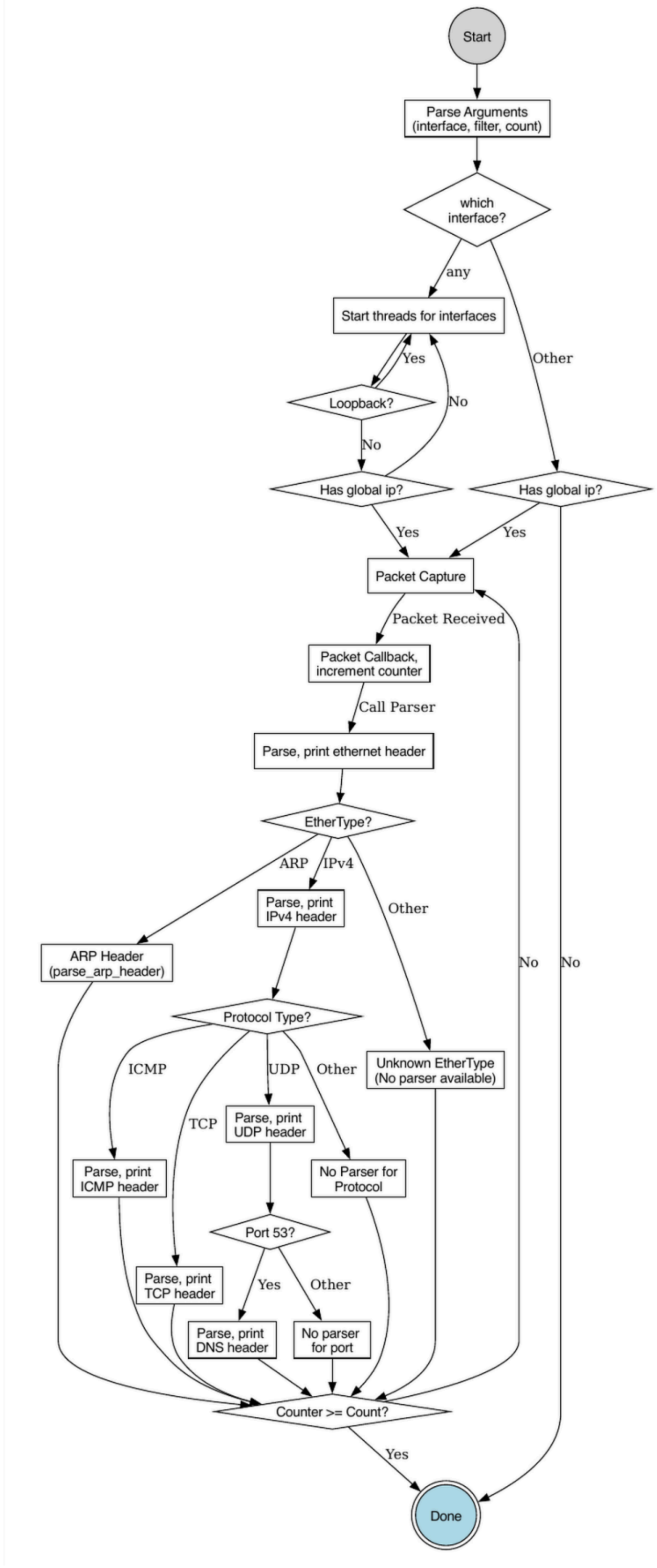
	transitions to capture packets; if No, returns to capture on all interfaces.
capture packets	Captures packets from the network. On receiving a packet, transitions to pkt callback.
pkt callback	Processes the received packet (increments the counter) and calls the parser. Transitions to parse eth.
parse eth	Parses and prints the Ethernet header. Transitions to eth decision.
eth decision	Determines the EtherType. If ARP, transitions to arp; if IPv4, transitions to ipv4; if Other, transitions to unknown.
arp	Parses the ARP header (using `parse arp header`). Transitions to count decision.
ipv4	Parses and prints the IPv4 header. Transitions to ip decision.
unknown ethertype	Handles an unrecognized EtherType (no parser available). Transitions to count decision.
ip decision	For IPv4 packets, decides the protocol type. If ICMP, transitions to icmp; if TCP, transitions to tcp; if UDP, transitions to udp; if Other, transitions to no proto.
parse icmp	Parses and prints the ICMP header. Transitions to count decision.
parse tcp	Parses and prints the TCP header. Transitions to count decision.
parse udp	Parses and prints the UDP header. Transitions to udp decision.
no proto	No parser is available for the specified protocol. Transitions to count decision.
udp decision	For UDP packets, checks if the port is 53. If Yes, transitions to dns; if Other, transitions to no port parse.
parse dns	Parses and prints the DNS header. Transitions to count decision.
no port parse	No parser is available for the given UDP port. Transitions to count decision.
count decision	Checks if the packet counter has reached the specified count. If Yes, transitions to done; if No, returns to capture packets.
done	Final state. Packet capturing and parsing are complete.

State Table

From State	To State	Function
start	parse args	main
parse args	interface decision	main
interface decision	has global ip other	main
has global ip other	capture on all interfaces	has_global_ip
capture on all interfaces	is loopback	capture_on_all_interfaces
is loopback	has global ip	is_loopback
has global ip	capture packets	has_global_ip
capture packets	pkt callback	capture_packets
pkt callback	parse eth	packet_callback
parse eth	eth decision	parse_ethernet_header
eth decision	parse arp	parse_ethernet_header
parse arp	counter decision	parse_arp_header
eth decision	parse ipv4	parse_ethernet_header
parse ipv4	ip decision	parse_ip_header
unknown ethertype	counter decision	parse_ethernet_header
ip decision	unknown protocol	parse_ip_header
ip decision	parse icmp	parse_ip_header
ip decision	parse tcp	parse_ip_header
ip decision	parse udp	parse_ip_header
udp decision	parse dns	parse_udp_header
udp decision	Unknown port	parse_udp_header
parse icmp	counter decision	parse_icmp_header
parse tcp	counter decision	parse_tcp_header
parse udp	port decision	parse_udp_header

port decision	parse dns	parse_udp_header
port decision	no port parse	parse_udp_header
parse dns	counter decision	parse_dns_header
Unknown port	counter decision	parse_udp_header
counter decision	capture packets	capture_packets
counter decision	done	capture_packets
done	done	counter decision

State Transition Diagram



Pseudocode

```
// GLOBAL VARIABLES
global packet_counter ← 0
global counter_lock ← new Lock
global stop_event ← new Event
global_packet_limit ← 0

//----------------------------------------
// MAIN PROGRAM (main.py)
//----------------------------------------
function main():
    // Parse command-line arguments: interface, filter, and count.
    args ← parse_arguments()
    global_packet_limit ← args.count

    // Special case: if filter is "dns", convert it to the proper BPF filter.
    if (args.filter equals "dns" or "DNS"):
    args.filter ← "udp and port 53"

    // Start capturing based on the specified interface.
    if (args.interface.lowercase equals "any"):
    capture_on_all_interfaces(args.filter)
    else:
    if (has_global_ip(args.interface)):
    capture_packets(args.interface, args.filter)
    else:
    print "Error: The specified interface does not have an assigned global IP."

//----------------------------------------
// PACKET CAPTURE FUNCTIONS
//----------------------------------------

// Called for every packet received.
function packet_callback(packet):
    print "Packet callback triggered."
    acquire counter_lock:
    if (packet_counter < global_packet_limit):
    packet_counter ← packet_counter + 1
    print "Captured Packet", packet_counter
    hex_data ← convert packet to hexadecimal string
    // Pass hex data to the Ethernet header parser.
    parse_ethernet_header(hex_data)
    end if
    if (packet_counter ≥ global_packet_limit):
    set stop_event to True
    release counter_lock

// Capture packets on a specific interface.
function capture_packets(interface, capture_filter):
    print "Starting packet capture on", interface, "with filter:", capture_filter
    try:
    sniffer ← initialize Sniffer with:
    interface: interface
    filter: (capture_filter if provided, otherwise capture all packets)
    callback: packet_callback
    store: False
    stop condition: a function that returns stop_event is True

    sniffer.start()
    // Loop until stop_event is set (i.e., desired packet count reached)
    while (stop_event is not set):
    sleep for a short time (e.g., 0.1 seconds)

    if (sniffer is still running):
    sniffer.stop()
    catch KeyboardInterrupt:
    print "Packet capture stopped on", interface
    catch Exception as error:
    print "Error on interface", interface, ":", error
    else:
    print "Packet capture completed on", interface

// Capture packets on all valid interfaces (skipping loopback or those without a global IP).
function capture_on_all_interfaces(capture_filter):
    interfaces ← get list of network interfaces
    for each interface in interfaces:
    if (interface_is_loopback(interface)):
    continue to next interface
    if (has_global_ip(interface) is False):
    continue to next interface

    start a new thread that calls capture_packets(interface, capture_filter)
    add thread to thread list

    try:
    for each thread in thread list:
```

```
    join thread (wait until it finishes)
    catch KeyboardInterrupt:
    print "Packet capture interrupted. Cleaning up..."
    set stop_event to True
    for each thread in thread list:
    join thread

//---------------------------------------
// INTERFACE CHECK FUNCTIONS
//---------------------------------------

function interface_is_loopback(interface):
    addresses ← get network addresses for interface
    for each address in addresses:
    if (address is IPv4 and equals "127.0.0.1") or
    (address is IPv6 and equals "::1"):
    return True
    return False

function has_global_ip(interface):
    addresses ← get network addresses for interface
    for each address in addresses:
    if (address is IPv4 and does not start with "169.254") or
    (address is IPv6 and does not start with "fe80"):
    return True
    return False

//---------------------------------------
// HEADER PARSING FUNCTIONS (packet_parsers.py)
//---------------------------------------

// Parse Ethernet header from hex data.
function parse_ethernet_header(hex_data):
    dest_mac_hex ← substring(hex_data, 0, 12)
    source_mac_hex ← substring(hex_data, 12, 24)
    ether_type ← substring(hex_data, 24, 28)

    dest_mac ← hex_to_mac(dest_mac_hex)
    source_mac ← hex_to_mac(source_mac_hex)

    print "Ethernet Header:"
    print "  Destination MAC:", dest_mac_hex, "|", dest_mac
    print "  Source MAC:", source_mac_hex, "|", source_mac
    print "  EtherType:", ether_type, "|", convert ether_type to integer
    payload ← substring(hex_data, 28, end)

    // Route payload based on EtherType.
    if (ether_type equals "0806"):          // ARP packet
    parse_arp_header(payload)
    else if (ether_type equals "0800"):     // IPv4 packet
    parse_ip_header(payload)
    else:
    print "Unknown EtherType:", ether_type
    print "No parser available for this EtherType."
    return (ether_type, payload)

// Parse IPv4 header from hex data.
function parse_ip_header(hex_data):
    version ← convert substring(hex_data, 0, 1) from hex to integer
    header_length ← (convert substring(hex_data, 1, 2) from hex to integer) * 4
    total_length ← convert substring(hex_data, 4, 8) from hex to integer
    protocol_type ← convert substring(hex_data, 18, 20) from hex to integer
    src_ip ← hex_to_ip(substring(hex_data, 24, 32))
    dst_ip ← hex_to_ip(substring(hex_data, 32, 40))

    flags_frags_offset ← convert substring(hex_data, 12, 16) from hex to integer
    flags ← flags_frags_offset shifted right by 13
    fragment_offset ← flags_frags_offset AND 0x1FFF

    print "IPv4 Header:"
    print "  Version:", version
    print "  Header Length:", header_length, "bytes"
    print "  Total Length:", total_length
    print "  Flags & Fragment Offset:", flags_frags_offset
    print "   Flags:", flags
    print "   Fragment Offset:", fragment_offset
    print "  Protocol:", protocol_type
    print "  Source IP:", src_ip
    print "  Destination IP:", dst_ip

    // Dispatch based on the protocol type.
    if (protocol_type equals 1):
    parse_icmp_header(substring(hex_data, 40, end))
    else if (protocol_type equals 6):
    parse_tcp_header(substring(hex_data, 40, end))
    else if (protocol_type equals 17):
    parse_udp_header(substring(hex_data, 40, end))
    else:
    print "No parser available for this protocol type."

// Parse ICMP header from hex data.
function parse_icmp_header(hex_data):
    icmp_type ← convert substring(hex_data, 0, 2) from hex to integer
    icmp_code ← convert substring(hex_data, 2, 4) from hex to integer
    icmp_checksum ← convert substring(hex_data, 4, 8) from hex to integer
    print "ICMP Header:"
    print "  Type:", icmp_type
    print "  Code:", icmp_code
    print "  Checksum:", icmp_checksum
    print "  Payload (hex):", substring(hex_data, 8, end)

// Parse TCP header from hex data.
function parse_tcp_header(hex_data):
    src_port ← convert substring(hex_data, 0, 4) from hex to integer
    dst_port ← convert substring(hex_data, 4, 8) from hex to integer
    seq_num ← convert substring(hex_data, 8, 16) from hex to integer
    ack_num ← convert substring(hex_data, 16, 24) from hex to integer
    data_offset ← convert substring(hex_data, 24, 25) from hex to integer
    reserved ← convert substring(hex_data, 25, 26) from hex to integer
    flags ← convert substring(hex_data, 26, 28) from hex to integer
    data_offset_bytes ← data_offset * 4
    window ← convert substring(hex_data, 28, 32) from hex to integer
    checksum ← convert substring(hex_data, 32, 36) from hex to integer
    urgent_pointer ← convert substring(hex_data, 36, 40) from hex to integer

    // Extract individual flag bits (e.g., NS, CWR, ECE, etc.) from 'flags'
    print "TCP Header:"
    print "  Source Port:", src_port
    print "  Destination Port:", dst_port
    print "  Sequence Number:", seq_num
    print "  Acknowledgment Number:", ack_num
    print "  Data Offset:", data_offset, "|", data_offset_bytes, "bytes"
    print "  Reserved:", reserved
    print "  Flags:", flags, " (bits detail omitted for brevity)"
    print "  Window Size:", window
    print "  Checksum:", checksum
    print "  Urgent Pointer:", urgent_pointer
    print "  Payload (hex):", substring(hex_data, 40, end)

// Parse UDP header from hex data.
function parse_udp_header(hex_data):
    src_port ← convert substring(hex_data, 0, 4) from hex to integer
    dst_port ← convert substring(hex_data, 4, 8) from hex to integer
    length ← convert substring(hex_data, 8, 12) from hex to integer
    checksum ← convert substring(hex_data, 12, 16) from hex to integer

    print "UDP Header:"
    print "  Source Port:", src_port
    print "  Destination Port:", dst_port
    print "  Length:", length
    print "  Checksum:", checksum
    print "  Payload (hex):", substring(hex_data, 16, end)

    // If the UDP port is 53, assume it's a DNS packet.
    if (src_port equals 53 or dst_port equals 53):
    parse_dns_header(substring(hex_data, 16, end))

// Parse DNS header from hex data.
function parse_dns_header(hex_data):
    id ← convert substring(hex_data, 0, 4) from hex to integer
    flags ← convert substring(hex_data, 4, 8) from hex to integer
    qdcount ← convert substring(hex_data, 8, 12) from hex to integer
    ancount ← convert substring(hex_data, 12, 16) from hex to integer
    nscount ← convert substring(hex_data, 16, 20) from hex to integer
    arcount ← convert substring(hex_data, 20, 24) from hex to integer

    // Decode individual flag bits: qr, opcode, aa, tc, rd, ra, z, rcode.
    print "DNS Header:"
    print "  Transaction ID:", id
    print "  Flags:", flags, " (subfields omitted)"
    print "  Questions:", qdcount
    print "  Answer RRs:", ancount
    print "  Authority RRs:", nscount
    print "  Additional RRs:", arcount
    print "  Payload (hex):", substring(hex_data, 24, end)

// Parse ARP header from hex data.
function parse_arp_header(hex_data):
    hardware_type ← convert substring(hex_data, 0, 4) from hex to integer
    protocol_type ← convert substring(hex_data, 4, 8) from hex to integer
    hardware_size ← convert substring(hex_data, 8, 10) from hex to integer
    protocol_size ← convert substring(hex_data, 10, 12) from hex to integer
    operation ← convert substring(hex_data, 12, 16) from hex to integer
    sender_mac_hex ← substring(hex_data, 16, 28)
    sender_ip_hex ← substring(hex_data, 28, 36)
    target_mac_hex ← substring(hex_data, 36, 48)
    target_ip_hex ← substring(hex_data, 48, 56)

    sender_mac ← hex_to_mac(sender_mac_hex)
    sender_ip ← hex_to_ip(sender_ip_hex)
    target_mac ← hex_to_mac(target_mac_hex)
    target_ip ← hex_to_ip(target_ip_hex)

    print "ARP Header:"
    print "  Hardware Type:", hardware_type
    print "  Protocol Type:", protocol_type
    print "  Hardware Size:", hardware_size
    print "  Protocol Size:", protocol_size
    print "  Operation:", operation
    print "  Sender MAC:", sender_mac_hex, "|", sender_mac
    print "  Sender IP:", sender_ip_hex, "|", sender_ip
    print "  Target MAC:", target_mac_hex, "|", target_mac
    print "  Target IP:", target_ip_hex, "|", target_ip

//----------------------------------------
// HEX FORMATTER FUNCTIONS (hex_formatters.py)
//----------------------------------------

// Convert a hexadecimal string representing a MAC address into colon-separated format.
function hex_to_mac(hex_mac):
    octet_list ← empty list
    for each 2-character segment in hex_mac:
    append segment to octet_list
    mac_string ← join octet_list with ":"
    return mac_string

// Convert a hexadecimal string representing an IP address into dot-separated decimal format.
function hex_to_ip(hex_ip):
    octet_list ← empty list
    for each 2-character segment in hex_ip:
    number ← convert segment from hex to integer
    append number to octet_list
    ip_string ← join octet_list with "."
    return ip_string

//----------------------------------------
// EXECUTION START
//----------------------------------------
if this script is run as the main program:
```

```
call main()
```