DNS Monitor

Project Documentation

Author: Aleksander Postelga

Date: November 18, 2024

Contents

1 Introduction

The Domain Name System (DNS) is a fundamental component of the Internet infrastructure that translates human-readable domain names into IP addresses and vice versa. Monitoring DNS communication is crucial for network traffic analysis, detecting security threats, and troubleshooting connectivity issues. This project focuses on implementing a tool called `dns-monitor`, which allows users to monitor DNS communication on a selected network interface or process DNS messages from a PCAP file.

2 Introduction

The Domain Name System (DNS) is a critical component of the internet's infrastructure, providing the essential service of translating human-readable domain names into IP addresses that computers use to identify each other on the network.

This document presents the implementation of a DNS monitoring project. This project focuses on implementing a tool called `dns-monitor`, which allows users to monitor DNS communication on a selected network interface or process DNS messages from a PCAP file.

3 Application Design

The primary goal of the project is to provide three main functionalities:

- Display information about DNS messages in a simplified or detailed format.
- Record observed domain names into a file.
- Record translations of domain names to IP addresses into a file.

The application processes only DNS messages transmitted over the UDP protocol. Supported DNS record types include A, AAAA, NS, MX, SOA, CNAME, and SRV. The application is implemented in C using the `libpcap` library for capturing network traffic and the `libresolv` library for working with DNS messages.

3.1 Architecture of the Application

The application is modularly designed, with each file handling a specific functionality:

- `args.c` **and** `args.h`: Handles parsing and validation of command-line arguments. Ensures correct combinations and displays usage instructions when needed.
  - `ProgramArguments`: Structure to hold parsed command-line arguments.
    - `int verbose`: Indicates verbose mode.
    - `char *interface`: The network interface for capturing packets.
    - `char *pcap_file`: The PCAP file for offline processing.
    - `char *domains_file`: File for storing observed domain names.
    - `char *translations_file`: File for storing domain-to-IP translations.
- `dns_capture.c` **and** `dns_capture.h`: Manages DNS packet capturing using `libpcap`, sets up filters for DNS traffic, and processes packets based on their protocol (IPv4/IPv6).
  - `DnsMonitorContext`: This structure holds the runtime state and references to arguments, domain set, translation set, and the packet capture handle.
    - `ProgramArguments *args`: Parsed arguments.
    - `DomainSet domain_set`: Set of observed domain names.
    - `TranslationSet translation_set`: Set of domain-to-IP translations.
    - `pcap_t *pcap_handle`: Handle for live or offline packet capture.
- `domains.c` **and** `domains.h`: Manages observed domain names, ensuring uniqueness and storing them in a file if specified.
  - `DomainSet`: Dynamic structure for managing domain names.
    - `size_t count`: Number of stored domain names.
    - `size_t capacity`: Maximum capacity of the domain array.
    - `char **domains`: Array of domain name strings.

- `translations.c` **and** `translations.h`: Handles domain-to-IP translations, avoiding duplicates, and optionally writes them to a file.
  - `TranslationSet`: Manages translations dynamically.
    - `size_t count`: Number of translations.
    - `size_t capacity`: Maximum capacity of the translation array.
    - `TranslationEntry *entries`: Array of translations.
  - `TranslationEntry`: Represents a single domain-to-IP mapping.
    - `char *domain`: Domain name.
    - `char *ip`: Associated IPv4 or IPv6 address.
- `process_dns_packet.c` **and** `process_dns_packet.h`: Processes DNS messages, extracting information from headers, questions, and resource records.
  - `DnsHeader`: Represents a DNS message header.
    - `uint16_t id`: Transaction ID.
    - `uint16_t flags`: DNS flags and response codes.
    - `uint16_t question_count`: Number of questions.
    - `uint16_t answer_count`: Number of answers.
    - `uint16_t authority_count`: Number of authority records.
    - `uint16_t additional_count`: Number of additional records.
- `print_dns.c` **and** `print_dns.h`: Formats and displays DNS message details, supporting both simplified and verbose output modes.
- `main.c`: The program's entry point, coordinating all modules, initializing resources, and starting packet capturing. It uses the structures described above to manage program state and pass data between components.

4 Implementation Description

4.1 Command-Line Argument Parsing

The program accepts the following arguments:

- `-i <interface>`: Name of the network interface to capture packets from.

- `-p <pcapfile>`: Name of the PCAP file to process stored packets.
- `-v`: Activates verbose mode for detailed output.
- `-d <domainsfile>`: File to store observed domain names.
- `-t <translationsfile>`: File to store translations of domain names to IP addresses.

Argument parsing is implemented using the `getopt` function from the `unistd.h` library. In case of incorrect usage, the program displays help information and terminates with an error. Below are the specific cases where the program will terminate with an error:

- **Missing Required Argument**: If neither `-i <interface>` nor `-p <pcapfile>` is provided, the program terminates with an error.
- **Conflicting Arguments**: If both `-i <interface>` and `-p <pcapfile>` are provided simultaneously, the program terminates with an error.
- **Unknown Option**: If an unrecognized or unsupported option is provided, the program terminates with an error and displays the usage message.
- **Missing Argument for an Option**: If an option that requires an argument (e.g., `-i`, `-p`, `-d`, `-t`) is provided without its corresponding argument, the program also terminates with an error.

In each of these cases, the program displays the usage information by invoking the `print_usage` function and terminates with an appropriate error code:

- `EXIT_FAILURE (1)`: For general errors.
- `BAD_ARGUMENTS (2)`: For invalid or conflicting arguments.

4.2 Program Workflow at Startup

At the beginning of the program execution, several critical steps are performed to set up the environment for DNS monitoring and processing:

1. **Parsing Command-Line Arguments**:

   The program starts by parsing the command-line arguments using the `parse_args` function. This function processes the input parameters and stores them in a `ProgramArguments` structure.

```
int main(int argc, char **argv) {
    ProgramArguments args = parse_args(argc, argv);
}
```

Listing 1: Parsing command-line arguments

2. **Initializing the Monitoring Context**:

   After successfully parsing the arguments, the program initializes the `DnsMonitorContext` structure. This context holds references to the parsed arguments and initializes the domain and translation sets, which are used to store observed domain names and their translations.

```
DnsMonitorContext context;
context.args = &args;

initialize_domain_set(&context.domain_set);
initialize_translation_set(&context.translation_set);
```

Listing 2: Initializing the monitoring context

   The `initialize_domain_set` and `initialize_translation_set` functions allocate memory and set up initial capacities for managing observed domain names and translations.

3. **Loading Existing Domains and Translations**:

   If the user has specified domain (`-d`) or translation (`-t`) files, the program loads existing entries from these files into the respective sets. This was done to avoid writing domains and translations that have already been written to the file.

```
if (args.domains_file) {
    load_domains_from_file(&context.domain_set, args.
        domains_file);
}

if (args.translations_file) {
    load_translations_from_file(&context.
        translation_set, args.translations_file);
}
```

Listing 3: Loading existing domains and translations

4. **Setting Up Signal Handlers**:

   To ensure graceful termination, the program sets up signal handlers for signals such as `SIGINT`, `SIGTERM`, and `SIGQUIT`. These signals are commonly used to interrupt or terminate a running program. The

program uses the `sigaction` function to register a handler function, `handle_signal`, for these signals. This mechanism allows the program to perform necessary cleanup operations before exiting, such as freeing memory and closing resources.

```
struct sigaction sa;
sa.sa_handler = handle_signal; // Assign the handler
    function
sigemptyset(&sa.sa_mask);      // Clear the signal
    mask
sa.sa_flags = 0;               // No special flags

if (sigaction(SIGINT, &sa, NULL) == -1 ||
    sigaction(SIGTERM, &sa, NULL) == -1 ||
    sigaction(SIGQUIT, &sa, NULL) == -1) {
    perror("sigaction");
    exit(EXIT_FAILURE);
}
```

Listing 4: Setting up signal handlers

The `handle_signal` function is a simple handler that sets a global flag, `stop_capture`, indicating that the program should terminate the packet capturing loop. This ensures the program can exit in an orderly fashion rather than being abruptly terminated by the operating system.

```
volatile sig_atomic_t stop_capture = 0;

void handle_signal(int signal) {
    stop_capture = 1; // Notify the program to stop
        the capture loop
}
```

Listing 5: Signal handler function

During the packet capturing loop, the `stop_capture` flag is regularly checked. If the flag is set, the program stops capturing packets, frees allocated memory, and then terminates the program. The implementation of the loop is as follows:

```
while (!stop_capture) {
    int ret = pcap_dispatch(handle, -1,
        dns_packet_handler, (u_char *) context);
    // ...
}
```

Listing 6: Handling signal during the capture loop

This approach ensures that when a termination signal is received, the program exits gracefully by:

- Stopping the capture loop.
- Freeing dynamically allocated memory (e.g., domain and translation sets).
- Closing the pcap handle and freeing filter.

4.3 Packet Capturing

Capturing DNS packets is achieved using the `libpcap` library. The program sets up a filter `"udp port 53"` to capture only UDP packets on port 53, where DNS communication typically occurs. The program initializes and starts capturing packets either from a live network interface or from a PCAP file, based on the arguments provided by the user. The following steps outline the process:

4.3.1 Opening the Capture Source

Depending on the input arguments, the program opens a capture source using either the `pcap_open_live` or `pcap_open_offline` function from the `libpcap` library. The function `pcap_open_live` is used for real-time packet capture from a specified network interface, while `pcap_open_offline` processes stored packets from a PCAP file.

```
pcap_t *handle;
char errbuf[PCAP_ERRBUF_SIZE];

if (context.args->interface) {
    handle = pcap_open_live(context.args->interface, 65536,
        1, 100, errbuf);
} else {
    handle = pcap_open_offline(context.args->pcap_file,
        errbuf);
}

if (handle == NULL) {
    fprintf(stderr, "Error: %s\n", errbuf);
    free_all_resources(&context);
    exit(EXIT_FAILURE);
}

context.pcap_handle = handle;
```

Listing 7: Opening the capture source

If the capture source cannot be opened, the program reports an error and terminates.

4.3.2 Setting Up the Packet Filter

To focus specifically on DNS traffic, the program compiles and applies a Berkeley Packet Filter (BPF) expression. This filter limits the captured packets to only those transmitted over UDP on port 53. The filter compilation is performed using `pcap_compile`, and the result is applied to the capture handle using `pcap_setfilter`.

```
struct bpf_program dns_filter;
char filter_exp[] = "udp port 53";

if (pcap_compile(handle, &dns_filter, filter_exp, 0,
    PCAP_NETMASK_UNKNOWN) == -1) {
    fprintf(stderr, "Error compiling filter: %s\n",
        pcap_geterr(handle));
    pcap_freecode(&fp);
    free_all_resources(context);
    exit(EXIT_FAILURE);
}

if (pcap_setfilter(handle, &dns_filter) == -1) {
    fprintf(stderr, "Error setting filter: %s\n", pcap_geterr
        (handle));
    pcap_freecode(&fp);
    free_all_resources(context);
    exit(EXIT_FAILURE);
}
```

Listing 8: Setting up the packet filter

The compiled filter ensures that only relevant packets are processed, improving efficiency and reducing unnecessary overhead.

4.3.3 Starting the Packet Capture Loop

Once the capture source and filter are configured, the program starts a packet capture loop using `pcap_dispatch`. This loop continuously processes packets, invoking the user-defined callback function `dns_packet_handler` for each captured packet. The loop runs until the program receives a termination signal (`stop_capture` is set) or an error occurs.

```
while (!stop_capture) {
    int ret = pcap_dispatch(handle, -1, dns_packet_handler, (
        u_char *) context);
    if (ret == -1) {
        // Error occurred during packet processing
        fprintf(stderr, "Error: %s\n", pcap_geterr(handle));
        free_all_resources(context);
        pcap_freecode(&fp);
```

```
        exit(EXIT_FAILURE);
    } else if (ret == 0 && pcap_file(handle) != NULL) {
        // End of file reached for offline PCAP processing
        break;
    }
}
```

Listing 9: Starting the packet capture loop

The `dns_packet_handler` function processes each packet to extract DNS message details, as described in the DNS processing section.

4.3.4 Error Handling and Resource Management

If an error occurs during any of these steps, the program logs the error message, frees allocated resources, and terminates gracefully. For example, the BPF filter memory is released using `pcap_freecode`, and the `pcap` handle is closed using `pcap_close`.

```
pcap_freecode(&dns_filter);
pcap_close(handle);
```

Listing 10: Releasing resources

4.4 DNS Message Processing

Processing DNS messages involves several steps:

- **Determining Packet Type**: The program identifies whether the packet is IPv4 or IPv6 by examining the Ethernet header's EtherType field.
- **Extracting Headers**: It extracts the IP and UDP headers accordingly.
- **Handling IPv6 Extension Headers**: For IPv6 packets, extension headers are skipped using a helper function to reach the UDP payload.
- **Parsing DNS Message**: The DNS payload is parsed to extract header fields, question sections, and resource records.

4.4.1 Supported Datalink Types

The program supports only **DLT_EN10MB (Ethernet)** datalink. This is the standard format for Ethernet networks, which are the most common in wired connections. Since most DNS traffic travels over Ethernet, this type is

fully supported. Other datalink types not supported. Adding support for less common types like Wi-Fi would make the program more complex without much practical benefit. However, if needed, the program can be extended in the future to handle other types.

4.4.2 Processing IPv4 Packets

To process IPv4 packets, the program begins by skipping the Ethernet header, which has a fixed size. It then interprets the subsequent bytes as the IP header. Unlike the Ethernet header, the IP header's length is variable due to the potential inclusion of optional fields. To determine its size, the program reads the `ip_hl` field (Internet Header Length), which specifies the header length in 32-bit words. Since each word is 4 bytes, the program multiplies this value by 4 to calculate the actual header size in bytes. This calculation allows the program to correctly locate the start of the UDP header. After processing the fixed-size UDP header, the program extracts the DNS payload.

```
const struct ip *ip_hdr = (struct ip *) (packet + sizeof(
    struct ether_header)); // Skip Ethernet header
int ip_header_length = ip_hdr->ip_hl * 4; // Calculate IP
    header length in bytes
const struct udphdr *udp_hdr = (struct udphdr *) ((u_char *)
    ip_hdr + ip_header_length); // Move to UDP header
const u_char *dns_payload = (const u_char *) udp_hdr + sizeof
    (struct udphdr); // Locate DNS payload
```

Listing 11: Processing IPv4 header

Offset	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Version (4)				IHL				DSCP						ECN		Total Length															
4	32	Identification																Flags			Fragment Offset											
8	64	Time to Live								Protocol								Header Checksum															
12	96	Source address																															
16	128	Destination address																															
20	160	(Options) (if IHL > 5)																															
⋮	⋮																																
56	448																																

Structure of IPv4 header

4.4.3 Processing IPv6 Packets

To process IPv6 packets, the program first skips the Ethernet header, which has a fixed size. Unlike IPv4, the size of the IPv6 header is always fixed at 40 bytes. However, IPv6 packets can include additional extension headers that need to be processed. These extension headers are optional and their presence is indicated by the `Next Header` field in the IPv6 header.

The program uses the `skip_ipv6_extension_headers` function to iteratively traverse any extension headers until it reaches the UDP header. This function reads each extension header, calculates its size, and moves the pointer to the next header in the chain. Once the UDP header is located, the program extracts the fixed-size UDP header and finally processes the DNS payload.

```
const struct ip6_hdr *ip6_hdr = (struct ip6_hdr *) (packet +
    sizeof(struct ether_header)); // Skip Ethernet header
uint8_t next_header = ip6_hdr->ip6_nxt; // Get the Next
    Header field
const u_char *payload = (const u_char *) (ip6_hdr + 1); //
    Move to the payload
const u_char *end_of_packet = packet + header->caplen;
payload = skip_ipv6_extension_headers(payload, &next_header,
    end_of_packet); // Skip extension headers
if (!payload) return; // If no UDP header is found, exit
const struct udphdr *udp_hdr = (const struct udphdr *)
    payload;
const u_char *dns_payload = (const u_char *) udp_hdr + sizeof
    (struct udphdr);
```

Listing 12: Processing IPv6 header

However, not all IPv6 extension headers have the same size, which makes it necessary to dynamically calculate their length based on the type of the header. This is accomplished by examining the specific fields of each header type. Supported extension headers are Hop-by-Hop, Destination, Routing, Fragment extension headers. Other extension headers not supported because they are not typically relevant for DNS monitoring.

Offset	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Version				Traffic class								Flow label																			
4	32	Payload length																Next header								Hop limit						
8	64	Source address																															
12	96																																
16	128																																
20	160																																
24	192	Destination address																															
28	224																																
32	256																																
36	288																																

Structure of IPv6 header

Offset	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Next header								Header extension length								Options and padding															
4	32	Options and padding																															
8	64	Optional: more Options and padding																															
12	96																																
⋮	⋮																																

Hop-by-Hop Options and Destination Options extension header format

Offset	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Next header								Header extension length								Routing type								Segments left							
4	32	Type-specific data																															
8	64	Optional: more type-specific data...																															
12	96																																
⋮	⋮																																

Routing extension header format

Offset	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Next Header								Reserved								Fragment offset													Res		M
4	32	Identification																															

Fragment extension header format

After processing the headers and locating the DNS payload, the program calls the `print_dns_information` (for IPv4) or `print_dns_information_ipv6` (for IPv6) functions. These functions are responsible for extracting the source and destination IP addresses from the IP header and delegating the task of printing detailed DNS packet information to other helper functions.

4.4.4 Parsing DNS Header

The DNS header is parsed by reading fixed-length fields. The program extracts the identifier, flags, and counts for question, answer, authority, and additional sections from the DNS header and stores it in DNS header structure. These values are converted from network byte order to host byte order using the `ntohs` function.

```
DnsHeader dns_header;
dns_header.id = ntohs(*(uint16_t *)(dns_payload));
dns_header.flags = ntohs(*(uint16_t *)(dns_payload + 2));
dns_header.question_count = ntohs(*(uint16_t *)(dns_payload +
    4));
dns_header.answer_count = ntohs(*(uint16_t *)(dns_payload +
   6));
dns_header.authority_count = ntohs(*(uint16_t *)(dns_payload
   + 8));
dns_header.additional_count = ntohs(*(uint16_t *)(dns_payload
    + 10));
```

Listing 13: Parsing DNS header

Offsets		0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
0	0	Transaction ID																Flags															
																		QR	OPCODE				AA	TC	RD	RA	Z				RCODE		
4	32	Number of questions																Number of answers															
8	64	Number of authority RRs																Number of additional RRs															

DNS Header

4.4.5 Parsing DNS Flags

The `flags` field in the DNS header contains important information about the nature and behavior of the DNS message. These flags are parsed using bitwise operations to extract individual bits or groups of bits representing specific flags or codes.

How DNS Flags are Parsed: The flags field is a 16-bit value where specific bits represent different attributes of the DNS message. To extract a particular flag, a bitwise AND operation (`&`) is used with a mask that isolates the relevant bits. The result is then shifted right (`»`) to bring the desired bits to the least significant position for interpretation.

DNS Flags: The program extracts and logs the following flags:

- **QR (Query/Response):** Indicates whether the DNS message is a query (`QR = 0`) or a response (`QR = 1`).
- **Opcode:** Specifies the type of query (e.g., standard query, inverse query).
- **AA (Authoritative Answer):** Indicates whether the response is from an authoritative DNS server.
- **TC (Truncated):** Specifies if the message was truncated due to size limitations.
- **RD (Recursion Desired):** Indicates if recursion is requested by the client.
- **RA (Recursion Available):** Specifies if the server supports recursion.
- **AD (Authenticated Data):** Indicates if the data is authenticated.
- **CD (Checking Disabled):** Specifies if DNSSEC validation is disabled.
- **RCODE (Response Code):** A 4-bit field representing the status of the DNS query (e.g., `NOERROR`, `NXDOMAIN`).

```
printf("Flags: QR=%d, OPCODE=%d, AA=%d, TC=%d, RD=%d, RA=%d,
    AD=%d, CD=%d, RCODE=%d\n",
            is_response, (dns_header->flags & 0x7800) >> 11, (
               dns_header->flags & 0x0400) >> 10,
            (dns_header->flags & 0x0200) >> 9, (dns_header->
               flags & 0x0100) >> 8, (dns_header->flags & 0
               x0080) >> 7,
            (dns_header->flags & 0x0020) >> 5, (dns_header->
               flags & 0x0010) >> 4, dns_header->flags & 0
               x000F);
```

Listing 14: Parsing DNS Flags

Explanation: Each flag is isolated using a bitmask. For example, to extract the `QR` flag (15th bit), the program performs a bitwise AND operation with `0x8000` (which has only the 15th bit set to 1) and then shifts the result 15 bits to the right. Similar operations are applied for other flags.

4.4.6 Processing Question Section

The question section of a DNS message contains queries specifying the information requested by the client. The program processes this section by iterating over each question and extracting the domain name, query type (`qtype`), and query class (`qclass`).

The program supports the following class names: Internet (IN), CSNET (CS), Chaos (CH), and Hesiod (HS). If an unsupported class name is encountered, the program prints UNKNOWN. To obtain the class names, the function `get_dns_class_name` is used.

- **Domain Name Parsing:** The program uses the `process_domain_name` function to parse the domain name, handling DNS name compression with the `dn_expand` function from `libresolv`.
- **Query Type and Class:** After parsing the domain name, the program extracts the query type and query class by reading the next 4 bytes and converting them from network byte order to host byte order using `ntohs`.

```
// Read type and class
uint16_t qtype = ntohs(*(uint16_t*) payload);
uint16_t qclass = ntohs(*(uint16_t*) (payload + 2));
payload += 4;
```

Listing 15: Processing Question Section

- **Recording:** If a domain file is specified, the parsed domain names are stored in the domain set and saved to the file.

Field	Description	Length (octets)
NAME	Name of the requested resource	Variable
TYPE	Type of RR (A, AAAA, MX, TXT, etc.)	2
CLASS	Class code	2

Question section

4.4.7 Processing Resource Record (RR) Section

The resource record (RR) sections in a DNS message—`Answer`, `Authority`, and `Additional`—contain detailed information, such as responses to the query or metadata about the domain. The program processes these sections by iterating over each record and extracting relevant information, including the domain name, record type (`rtype`), class (`rclass`), Time-To-Live (TTL), and resource data (RDATA). The data was parced the same as the for question section but in comparison with question section we also parsing ttl and length of section

```
uint16_t rtype = ntohs(*(uint16_t *) payload);
uint16_t rclass = ntohs(*(uint16_t *) (payload + 2));
uint32_t ttl = ntohl(*(uint32_t *) (payload + 4));
uint16_t rdlength = ntohs(*(uint16_t *) (payload + 8));
payload += 10;
```

Listing 16: Processing Resource Record Section

Field	Description	Length (octets)
NAME	Name of the node to which this record pertains	Variable
TYPE	Type of RR in numeric form (e.g., 15 for MX RRs)	2
CLASS	Class code	2
TTL	Count of seconds that the RR stays valid (The maximum is $2^{31}-1$, which is about 68 years)	4
RDLENGTH	Length of RDATA field (specified in octets)	2
RDATA	Additional RR-specific data	Variable, as per RDLENGTH

Resource record (RR) fields

After parsing the fixed-length fields, the program processes the resource data (RDATA) based on the rtype (record type). Different DNS record types are handled using separate logic to accommodate their unique structures.

4.4.8 Handling Specific DNS Record Types

The program processes supported DNS record types (A, AAAA, NS, MX, SOA, CNAME, SRV) using specialized functions, if the program encountered a type that is not supported, the output will be OTHER. Each record type contains unique data in the RDATA section, which is extracted and handled appropriately based on its structure and purpose.

- **A Record:** Maps a domain name to an IPv4 address. The program processes the RDATA field to extract the 4-byte (32-bit) IPv4 address and converts it to a human-readable string using inet_ntop. If the translation file is provided stores the mapping.

```
// Convert IPv4 to string
inet_ntop(AF_INET, &addr, ip_str, sizeof(ip_str));
add_translation(&context->translation_set,
    domain_name, ip_str, translations_file);
```

- **AAAA Record:** Maps a domain name to an IPv6 address. Similar to the A record, it extracts a 16-byte (128-bit) IPv6 address from the RDATA and converts it to a readable string using inet_ntop. If the translation file is provided stores the mapping.

```
// Convert IPv6 to string
inet_ntop(AF_INET6, &addr6, ip_str, sizeof(ip_str));
```

- **NS and CNAME Records:** NS records indicate authoritative name servers for a domain, while CNAME records specify an alias for a canonical domain name. The program extracts and decodes the domain name in the RDATA field, handling DNS name compression using dn_expand. The extracted domain names are added to the domain set if domain file is provided.

```
int name_len = process_domain_name(packet_start,
    payload, rdata_domain, sizeof(rdata_domain));
if (name_len >= 0) {
    if (context->args->verbose) printf("%s.\n",
        rdata_domain);
    if (context->args->domains_file) {
        add_domain(&context->domain_set, rdata_domain
            , context->args->domains_file);
```

```
    }
}
```

- **MX Record:** Specifies mail exchange servers for a domain, along with their priority. The program extracts the 2-byte (16-bit) preference field and the domain name of the mail exchange server from the `RDATA`. The mail exchange domain is added to the domain set, if the domain file is provided.

```
uint16_t preference = ntohs(*(uint16_t *) payload);
char rdata_domain[256];
int name_len = process_domain_name(packet_start,
   payload + 2, rdata_domain, sizeof(rdata_domain));
if (name_len >= 0) {
    if (context->args->verbose) printf("%d␣%s.\n",
       preference, rdata_domain);
    if (context->args->domains_file) {
        add_domain(&context->domain_set, rdata_domain
           , context->args->domains_file);
    }
}
```

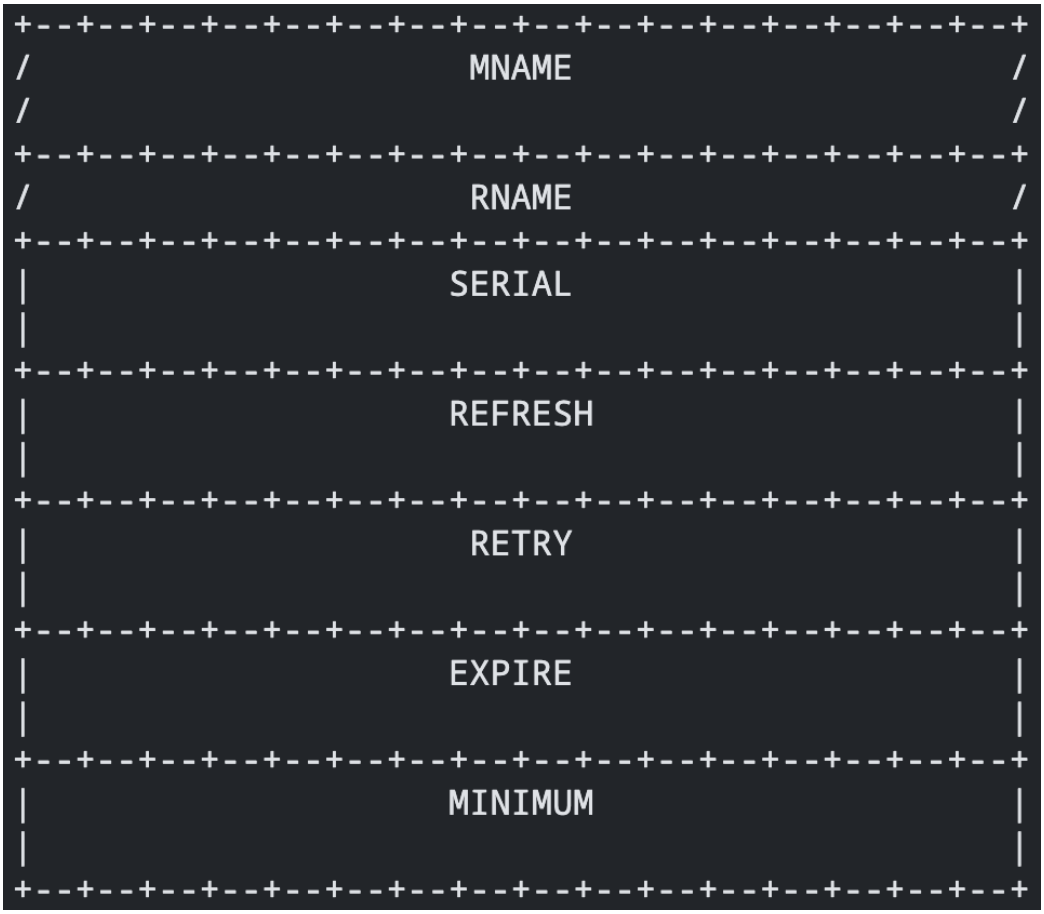
- **SOA Record:** The Start of Authority (SOA) record defines the authoritative information about a domain. It contains details such as the primary name server, the responsible party's email address, and various timers (serial number, refresh interval, retry interval, expiry time, and minimum TTL). The program processes each field sequentially by parsing the `RDATA` using the predefined structure of the SOA record.

  - **Primary Name Server (MNAME):** The first section of the `RDATA` contains the domain name of the primary name server for the zone. This is extracted using the `process_domain_name` function, which handles DNS name compression and added to domain file if specified.
  - **Responsible Party's Email (RNAME):** Following the MNAME, the responsible party's email address (in domain name format) is parsed. The same `process_domain_name` function is used for this field.
  - **Timers:** After the domain name fields, the program reads the following fixed-size fields in order:
    - **Serial Number:** A 32-bit integer representing the version of the zone file.

- **Refresh Interval:** A 32-bit integer specifying how often secondary name servers should check for updates.
- **Retry Interval:** A 32-bit integer indicating the time to wait before retrying failed zone updates.
- **Expiry Time:** A 32-bit integer representing the time a zone will be considered valid without an update.
- **Minimum TTL:** A 32-bit integer specifying the default TTL for resource records.

```
char mname[256], rname[256];
const u_char *rdata_ptr = payload;

// Extract primary name server
int mname_len = process_domain_name(packet_start,
   rdata_ptr, mname, sizeof(mname));
if(context->args->domains_file) {
        add_domain(&context->domain_set, mname, context->
           args->domains_file);
    }
if (mname_len < 0) return;
rdata_ptr += mname_len;

// Extract responsible party's email
int rname_len = process_domain_name(packet_start,
   rdata_ptr, rname, sizeof(rname));
if (rname_len < 0) return;
rdata_ptr += rname_len;

// Verify sufficient data for timers
if (rdata_ptr + 20 > payload + rdlength) return;

// Extract timers
uint32_t serial = ntohl(*(uint32_t *) rdata_ptr);
   rdata_ptr += 4;
uint32_t refresh = ntohl(*(uint32_t *) rdata_ptr);
   rdata_ptr += 4;
uint32_t retry = ntohl(*(uint32_t *) rdata_ptr);
   rdata_ptr += 4;
uint32_t expire = ntohl(*(uint32_t *) rdata_ptr);
   rdata_ptr += 4;
uint32_t minimum = ntohl(*(uint32_t *) rdata_ptr);
```

Listing 17: Processing SOA Record



Structure of the SOA Record

- **SRV Record:** Specifies services for a domain, including priority, weight, port, and target domain. The program extracts:
  - `Priority`: The priority of the target host, lower value means more preferred.
  - `Weight`: A relative weight for records with the same priority, higher value means higher chance of getting picked.
  - `Port`: The port on which the service is running.
  - `Target`: The domain hosting the service.

  The target domain is added to the domain set if domain file is provided.

```
uint16_t priority = ntohs(*(uint16_t *) payload); //
   Extract priority
```

```
uint16_t weight = ntohs(*(uint16_t *) (payload + 2));
    // Extract weight
uint16_t port = ntohs(*(uint16_t *) (payload + 4));
   // Extract port
```

4.5 Output and Storage of Information

4.5.1 Simplified Output

In the simplified output mode, the program prints to the console one line per DNS message, displaying the timestamp, source and destination IP addresses, query or response indicator, and counts of records in each section.

```
printf("%s␣%s␣->␣%s␣(%c␣%d/%d/%d/%d)\n",
       time_str, src_ip, dst_ip, qr,
       dns_header.question_count, dns_header.answer_count,
       dns_header.authority_count, dns_header.
          additional_count);
```

Listing 18: Simplified output example

4.5.2 Verbose Output

In verbose mode (-v), the program prints detailed information about the DNS message, including header fields, flags, and the content of each section to the console.

```
printf("Timestamp:␣%s\n", time_str);
printf("SrcIP:␣%s\n", src_ip);
printf("DstIP:␣%s\n", dst_ip);
printf("SrcPort:␣UDP/%d\n", ntohs(udp_hdr->uh_sport));
printf("DstPort:␣UDP/%d\n", ntohs(udp_hdr->uh_dport));
printf("Identifier:␣0x%X\n", dns_header.id);
printf("Flags:␣QR=%d,␣OPCODE=%d,␣AA=%d,␣TC=%d,␣RD=%d,␣RA=%d,␣
   AD=%d,␣CD=%d,␣RCODE=%d\n",
       is_response, (dns_header.flags & 0x7800) >> 11, (
          dns_header.flags & 0x0400) >> 10,
       (dns_header.flags & 0x0200) >> 9, (dns_header.flags &
          0x0100) >> 8, (dns_header.flags & 0x0080) >> 7,
       (dns_header.flags & 0x0020) >> 5, (dns_header.flags &
          0x0010) >> 4, dns_header.flags & 0x000F);
```

Listing 19: Verbose output example

4.5.3 Recording Domain Names

When the `-d` option is enabled, the program records observed domain names into a specified file while also printing them to the console. To ensure no duplicates are added, the program maintains a dynamic structure, `DomainSet`, to track already recorded domains, which is initialized at startup. Before appending a domain to the file, the program checks if it is already in the set using the `domain_exists` function. If the domain is new, it is added to the set and written to the file.

Domains are saved in real-time, ensuring the file remains up-to-date during program execution. If the file already exists, previously recorded domains are loaded into the set at startup using `load_domains_from_file` to prevent redundant entries.

4.5.4 Recording Translations

When the `-t` option is enabled, the program records translations of domain names to their corresponding IP addresses into a specified file. The implementation is conceptually similar to how domain names are handled using `DomainSet`, but it includes additional logic to handle the domain-to-IP mappings.

Each translation consists of a domain name and its associated IP address. Before a translation is added, the program checks for its existence in the `TranslationSet` using the `translation_exists` function, which ensures that the couple of domain and IP is unique. If the translation is new, it is added to the set and written to the file in the format `<domain> <IP>`.

When the program starts, any existing translations from the specified file are loaded into the `TranslationSet` using `load_translations_from_file`. This prevents redundant entries and ensures consistency between the in-memory set and the file.

5 Usage Instructions

5.1 Compiling the Program

The program can be compiled using the following command:

```
make
```

This command will handle all the necessary steps to compile the source code into an executable file. Additionally, it will organize the output by cre-

ating a dedicated directory where all binary files (except the main executable file) will be stored.

Ensure that the `libpcap` and `libresolv` libraries are installed on your system.

5.2 Running the Program

The program is executed with the following syntax:

```
./dns-monitor (-i <interface> | -p <pcapfile>) [-v] [-d <domainsfile>]
[-t <translationsfile>]
```

5.3 Examples of Usage

- **Basic execution with interface**:

  ```
  ./dns-monitor -i en0
  ```

- **Verbose output with PCAP file**:

  ```
  ./dns-monitor -p capture.pcap -v
  ```

- **Recording domain names and translations**:

  ```
  ./dns-monitor -i eth0 -d domains.txt -t translations.txt
  ```

6 Testing and Test Results

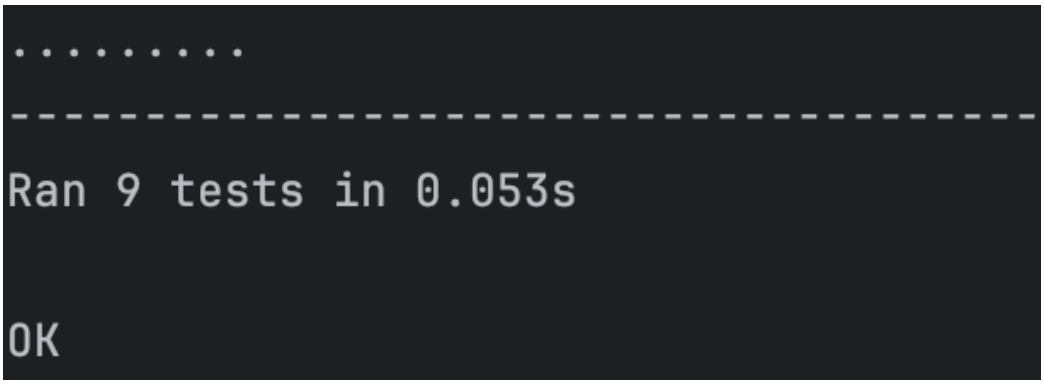
6.1 Argument Testing

To ensure that the program correctly handles command-line arguments, a series of automated tests were developed using Python's `unittest` framework. These tests validate the behavior of the program in various scenarios, focusing on error handling, proper functionality with valid arguments, and ensuring the creation of output files where applicable.

The following tests were conducted:

- **Missing Required Options**: Ensures the program exits with an error if neither `-i` nor `-p` is provided.
- **Conflicting Options**: Ensures the program exits with an error if both `-i` and `-p` are provided simultaneously.
- **Invalid Option**: Verifies the program reports an error when an unsupported option is used.
- **Help Display**: Confirms the program displays the help message and exits successfully when `-h` is provided.
- **Valid PCAP Option**: Ensures the program correctly processes a valid PCAP file with the `-p` option.
- **Verbose Mode with PCAP**: Verifies the program operates correctly in verbose mode (`-v`) with a PCAP file.
- **Combination of Options**: Confirms the program handles valid combinations of options (`-p`, `-v`, `-d`, `-t`) and creates output files.
- **Invalid PCAP File**: Ensures the program reports an error for a nonexistent PCAP file.
- **Invalid Network Interface**: Verifies the program reports an error when an invalid network interface is specified.

The all tests was passed correctly



Argument tests passed

6.2 DNS Record Types Parsing Tests

For testing purposes, a local DNS domain was configured using BIND9. The domain `test.local` was specifically created to facilitate testing by hosting all supported DNS record types on a single server. This setup allowed for convenient querying of various record types without needing to rely on external servers. The records included **A**, **AAAA**, **NS**, **MX**, **SOA**, **CNAME**, and **SRV**.

In addition to testing on the local server, the program was also tested against a public DNS server provided by Google (`8.8.8.8`). This testing ensured that the program could handle DNS messages from real-world environments. Queries for various record types were sent to the Google DNS server, including **A**, **AAAA**, and **MX**, targeting the domain `google.com`. The captured traffic was saved in a PCAP file for analysis.

To simulate DNS traffic for both local and public servers, a series of queries were sent using the `dig` tool. The traffic was captured in PCAP files using `tcpdump`.

The following commands were used during the test:

- To send DNS queries to the local server:

```
dig @127.0.0.1 ns1.test.local A
dig @127.0.0.1 ns1.test.local AAAA
dig @127.0.0.1 test.local NS
dig @127.0.0.1 test.local MX
dig @127.0.0.1 test.local SOA
dig @127.0.0.1 alias.test.local CNAME
dig @127.0.0.1 _service._tcp.test.local SRV
```

- To send DNS queries to the public Google DNS server:

```
dig @8.8.8.8 google.com A
dig @8.8.8.8 google.com AAAA
dig @8.8.8.8 google.com MX
```

- To capture DNS traffic in PCAP files:

```
sudo tcpdump -i lo port 53 -w tests/dns_capture_local.pcap
sudo tcpdump -i any host 8.8.8.8 and port 53
-w tests/dns_real_output.pcap
```

6.2.1 Simple Output Testing

After PCAP files were created containing DNS traffic from various queries, the packets were manually inspected using `Wireshark`.Based on this manual inspection, separate `expected_simple_output` `expected_simple_real_output` files were created for both the local and public DNS server tests.

Local Server Expected Output:

```
2024-11-17 21:49:57 127.0.0.1 -> 127.0.0.1 (Q 1/0/0/1)
2024-11-17 21:49:57 127.0.0.1 -> 127.0.0.1 (R 1/1/0/1)
2024-11-17 21:49:57 127.0.0.1 -> 127.0.0.1 (Q 1/0/0/1)
2024-11-17 21:49:57 127.0.0.1 -> 127.0.0.1 (R 1/1/0/1)
2024-11-17 21:49:57 127.0.0.1 -> 127.0.0.1 (Q 1/0/0/1)
2024-11-17 21:49:57 127.0.0.1 -> 127.0.0.1 (R 1/1/0/3)
2024-11-17 21:49:58 127.0.0.1 -> 127.0.0.1 (Q 1/0/0/1)
2024-11-17 21:49:58 127.0.0.1 -> 127.0.0.1 (R 1/1/0/2)
2024-11-17 21:49:58 127.0.0.1 -> 127.0.0.1 (Q 1/0/0/1)
2024-11-17 21:49:58 127.0.0.1 -> 127.0.0.1 (R 1/1/0/1)
2024-11-17 21:49:58 127.0.0.1 -> 127.0.0.1 (Q 1/0/0/1)
2024-11-17 21:49:58 127.0.0.1 -> 127.0.0.1 (R 1/1/0/1)
2024-11-17 21:49:58 127.0.0.1 -> 127.0.0.1 (Q 1/0/0/1)
2024-11-17 21:49:58 127.0.0.1 -> 127.0.0.1 (R 1/1/0/3)
```

Real DNS Server Expected Output:

```
2024-11-17 21:28:14 192.168.1.104 -> 8.8.8.8 (Q 1/0/0/1)
2024-11-17 21:28:14 8.8.8.8 -> 192.168.1.104 (R 1/1/0/1)
2024-11-17 21:28:14 192.168.1.104 -> 8.8.8.8 (Q 1/0/0/1)
2024-11-17 21:28:14 8.8.8.8 -> 192.168.1.104 (R 1/1/0/1)
2024-11-17 21:28:14 192.168.1.104 -> 8.8.8.8 (Q 1/0/0/1)
2024-11-17 21:28:14 8.8.8.8 -> 192.168.1.104 (R 1/1/0/1)
```

The `dns-monitor` program was then executed in simple output mode using the following commands:

```
./dns-monitor -p tests/dns_capture_local.pcap > tests/actual_simple_output
./dns-monitor -p tests/dns_real_output.pcap > tests/actual_simple_real_output
```

The resulting files, `actual_simple_output_local` and `actual_simple_output_google`, were compared against their respective `expected_simple_output` files using the `diff` command:

```
diff expected_simple_output actual_simple_output
diff expected_simple_real_output actual_simple_real_output
```

No differences were found in either comparison, confirming that the program's simple output mode functions correctly for both local and public DNS servers. The program successfully captured and parsed the DNS packets, providing output in the expected format.

6.2.2 Verbose Output Testing

The verbose output of the program was tested in the same manner as the simple output. PCAP files containing DNS traffic from local and public DNS servers were manually inspected using `Wireshark`, and detailed `expected_verbose_output` and `expected_verbose_real_output` files were created for both scenarios.

The `dns-monitor` program was executed in verbose mode with the following commands:

```
./dns-monitor -v -p tests/dns_capture_local.pcap >
tests/actual_verbose_output
./dns-monitor -v -p tests/dns_real_output.pcap >
tests/actual_verbose_real_output
```

Since the program does not support all DNS record types, unsupported record types were replaced with `UNKNOWN` and unsupported classes were replaced with `OTHER` in the output, as implemented in the program.

Local Server Expected Output:

```
Timestamp: 2024-11-17 21:49:57
SrcIP: 127.0.0.1
DstIP: 127.0.0.1
SrcPort: UDP/45132
DstPort: UDP/53
Identifier: 0xF413
Flags: QR=0, OPCODE=0, AA=0, TC=0, RD=1, RA=0, AD=1, CD=0, RCODE=0

[Question Section]
ns1.test.local. IN A

[Additional Section]
. 0 UNKNOWN OTHER (other data, length 12)
```

```
====================
Timestamp: 2024-11-17 21:49:57
SrcIP: 127.0.0.1
DstIP: 127.0.0.1
SrcPort: UDP/53
DstPort: UDP/45132
Identifier: 0xF413
Flags: QR=1, OPCODE=0, AA=1, TC=0, RD=1, RA=1, AD=0, CD=0, RCODE=0

[Question Section]
ns1.test.local. IN A

[Answer Section]
ns1.test.local. 604800 IN A 127.0.0.1

[Additional Section]
. 0 UNKNOWN OTHER (other data, length 28)
====================
Timestamp: 2024-11-17 21:49:57
SrcIP: 127.0.0.1
DstIP: 127.0.0.1
SrcPort: UDP/42982
DstPort: UDP/53
Identifier: 0xAA54
Flags: QR=0, OPCODE=0, AA=0, TC=0, RD=1, RA=0, AD=1, CD=0, RCODE=0

[Question Section]
ns1.test.local. IN AAAA

[Additional Section]
. 0 UNKNOWN OTHER (other data, length 12)
====================
Timestamp: 2024-11-17 21:49:57
SrcIP: 127.0.0.1
DstIP: 127.0.0.1
SrcPort: UDP/53
DstPort: UDP/42982
Identifier: 0xAA54
Flags: QR=1, OPCODE=0, AA=1, TC=0, RD=1, RA=1, AD=0, CD=0, RCODE=0

[Question Section]
```

```
ns1.test.local. IN AAAA

[Answer Section]
ns1.test.local. 604800 IN AAAA ::1

[Additional Section]
. 0 UNKNOWN OTHER (other data, length 28)
====================
Timestamp: 2024-11-17 21:49:57
SrcIP: 127.0.0.1
DstIP: 127.0.0.1
SrcPort: UDP/50316
DstPort: UDP/53
Identifier: 0x4C7A
Flags: QR=0, OPCODE=0, AA=0, TC=0, RD=1, RA=0, AD=1, CD=0, RCODE=0

[Question Section]
test.local. IN NS

[Additional Section]
. 0 UNKNOWN OTHER (other data, length 12)
====================
Timestamp: 2024-11-17 21:49:57
SrcIP: 127.0.0.1
DstIP: 127.0.0.1
SrcPort: UDP/53
DstPort: UDP/50316
Identifier: 0x4C7A
Flags: QR=1, OPCODE=0, AA=1, TC=0, RD=1, RA=1, AD=0, CD=0, RCODE=0

[Question Section]
test.local. IN NS

[Answer Section]
test.local. 604800 IN NS ns1.test.local.

[Additional Section]
ns1.test.local. 604800 IN A 127.0.0.1
ns1.test.local. 604800 IN AAAA ::1
. 0 UNKNOWN OTHER (other data, length 28)
====================
```

```
Timestamp: 2024-11-17 21:49:58
SrcIP: 127.0.0.1
DstIP: 127.0.0.1
SrcPort: UDP/50073
DstPort: UDP/53
Identifier: 0x6DDC
Flags: QR=0, OPCODE=0, AA=0, TC=0, RD=1, RA=0, AD=1, CD=0, RCODE=0

[Question Section]
test.local. IN MX

[Additional Section]
. 0 UNKNOWN OTHER (other data, length 12)
====================
Timestamp: 2024-11-17 21:49:58
SrcIP: 127.0.0.1
DstIP: 127.0.0.1
SrcPort: UDP/53
DstPort: UDP/50073
Identifier: 0x6DDC
Flags: QR=1, OPCODE=0, AA=1, TC=0, RD=1, RA=1, AD=0, CD=0, RCODE=0

[Question Section]
test.local. IN MX

[Answer Section]
test.local. 604800 IN MX 10 mail.test.local.

[Additional Section]
mail.test.local. 604800 IN A 127.0.0.3
. 0 UNKNOWN OTHER (other data, length 28)
====================
Timestamp: 2024-11-17 21:49:58
SrcIP: 127.0.0.1
DstIP: 127.0.0.1
SrcPort: UDP/34743
DstPort: UDP/53
Identifier: 0x3221
Flags: QR=0, OPCODE=0, AA=0, TC=0, RD=1, RA=0, AD=1, CD=0, RCODE=0

[Question Section]
```

```
test.local. IN SOA

[Additional Section]
. 0 UNKNOWN OTHER (other data, length 12)
====================
Timestamp: 2024-11-17 21:49:58
SrcIP: 127.0.0.1
DstIP: 127.0.0.1
SrcPort: UDP/53
DstPort: UDP/34743
Identifier: 0x3221
Flags: QR=1, OPCODE=0, AA=1, TC=0, RD=1, RA=1, AD=0, CD=0, RCODE=0

[Question Section]
test.local. IN SOA

[Answer Section]
test.local. 604800 IN SOA localhost. root.localhost. 3 604800 86400 2419200

[Additional Section]
. 0 UNKNOWN OTHER (other data, length 28)
====================
Timestamp: 2024-11-17 21:49:58
SrcIP: 127.0.0.1
DstIP: 127.0.0.1
SrcPort: UDP/45277
DstPort: UDP/53
Identifier: 0x27B9
Flags: QR=0, OPCODE=0, AA=0, TC=0, RD=1, RA=0, AD=1, CD=0, RCODE=0

[Question Section]
alias.test.local. IN CNAME

[Additional Section]
. 0 UNKNOWN OTHER (other data, length 12)
====================
Timestamp: 2024-11-17 21:49:58
SrcIP: 127.0.0.1
DstIP: 127.0.0.1
SrcPort: UDP/53
DstPort: UDP/45277
```

```
Identifier: 0x27B9
Flags: QR=1, OPCODE=0, AA=1, TC=0, RD=1, RA=1, AD=0, CD=0, RCODE=0

[Question Section]
alias.test.local. IN CNAME

[Answer Section]
alias.test.local. 604800 IN CNAME www.test.local.

[Additional Section]
. 0 UNKNOWN OTHER (other data, length 28)
====================
Timestamp: 2024-11-17 21:49:58
SrcIP: 127.0.0.1
DstIP: 127.0.0.1
SrcPort: UDP/43058
DstPort: UDP/53
Identifier: 0x9F3C
Flags: QR=0, OPCODE=0, AA=0, TC=0, RD=1, RA=0, AD=1, CD=0, RCODE=0

[Question Section]
_service._tcp.test.local. IN SRV

[Additional Section]
. 0 UNKNOWN OTHER (other data, length 12)
====================
Timestamp: 2024-11-17 21:49:58
SrcIP: 127.0.0.1
DstIP: 127.0.0.1
SrcPort: UDP/53
DstPort: UDP/43058
Identifier: 0x9F3C
Flags: QR=1, OPCODE=0, AA=1, TC=0, RD=1, RA=1, AD=0, CD=0, RCODE=0

[Question Section]
_service._tcp.test.local. IN SRV

[Answer Section]
_service._tcp.test.local. 604800 IN SRV 10 5 80 www.test.local.

[Additional Section]
```

```
www.test.local. 604800 IN A 127.0.0.2
www.test.local. 604800 IN AAAA 2001:db8::1
. 0 UNKNOWN OTHER (other data, length 28)
====================
```

Real DNS Server Expected Output:

```
Timestamp: 2024-11-17 21:28:14
SrcIP: 192.168.1.104
DstIP: 8.8.8.8
SrcPort: UDP/61852
DstPort: UDP/53
Identifier: 0xE9F5
Flags: QR=0, OPCODE=0, AA=0, TC=0, RD=1, RA=0, AD=1, CD=0, RCODE=0

[Question Section]
google.com. IN A

[Additional Section]
. 0 UNKNOWN OTHER (other data, length 0)
====================
Timestamp: 2024-11-17 21:28:14
SrcIP: 8.8.8.8
DstIP: 192.168.1.104
SrcPort: UDP/53
DstPort: UDP/61852
Identifier: 0xE9F5
Flags: QR=1, OPCODE=0, AA=0, TC=0, RD=1, RA=1, AD=0, CD=0, RCODE=0

[Question Section]
google.com. IN A

[Answer Section]
google.com. 178 IN A 142.251.36.78

[Additional Section]
. 0 UNKNOWN OTHER (other data, length 0)
====================
Timestamp: 2024-11-17 21:28:14
SrcIP: 192.168.1.104
DstIP: 8.8.8.8
```

```
SrcPort: UDP/49368
DstPort: UDP/53
Identifier: 0xC662
Flags: QR=0, OPCODE=0, AA=0, TC=0, RD=1, RA=0, AD=1, CD=0, RCODE=0

[Question Section]
google.com. IN AAAA

[Additional Section]
. 0 UNKNOWN OTHER (other data, length 0)
====================
Timestamp: 2024-11-17 21:28:14
SrcIP: 8.8.8.8
DstIP: 192.168.1.104
SrcPort: UDP/53
DstPort: UDP/49368
Identifier: 0xC662
Flags: QR=1, OPCODE=0, AA=0, TC=0, RD=1, RA=1, AD=0, CD=0, RCODE=0

[Question Section]
google.com. IN AAAA

[Answer Section]
google.com. 167 IN AAAA 2a00:1450:4014:80e::200e

[Additional Section]
. 0 UNKNOWN OTHER (other data, length 0)
====================
Timestamp: 2024-11-17 21:28:14
SrcIP: 192.168.1.104
DstIP: 8.8.8.8
SrcPort: UDP/59633
DstPort: UDP/53
Identifier: 0xF57
Flags: QR=0, OPCODE=0, AA=0, TC=0, RD=1, RA=0, AD=1, CD=0, RCODE=0

[Question Section]
google.com. IN MX

[Additional Section]
. 0 UNKNOWN OTHER (other data, length 0)
```

```
====================
Timestamp: 2024-11-17 21:28:14
SrcIP: 8.8.8.8
DstIP: 192.168.1.104
SrcPort: UDP/53
DstPort: UDP/59633
Identifier: 0xF57
Flags: QR=1, OPCODE=0, AA=0, TC=0, RD=1, RA=1, AD=0, CD=0, RCODE=0

[Question Section]
google.com. IN MX

[Answer Section]
google.com. 25 IN MX 10 smtp.google.com.

[Additional Section]
. 0 UNKNOWN OTHER (other data, length 0)
====================
```

The resulting files, `actual_verbose_output` and `actual_verbose_real_output`, were then compared to their respective `expected_verbose_output` and `expected_verbose_real_output` files using the `diff` command:

```
diff expected_verbose_output actual_verbose_output
diff expected_verbose_real_output actual_verbose_real_output
```

No differences were found during the comparison, confirming that the verbose output mode of the program functions correctly.

6.3 Domain Names Logging

The program's functionality to extract and log unique domain names into a file using the `-d` parameter was tested thoroughly. This feature searches for domain names in all DNS messages, including all sections (Question, Answer, Authority, and Additional). Whenever a domain name is encountered, it is stored in the specified file unless it is already present.

For testing this functionality, the same PCAP files that were used in previous tests were utilized again. The `dns-monitor` program was executed with the `-d` option, and the output file was checked for correctness.

6.3.1 Testing that file is created and information is written

The program was executed with the following commands:

```
./dns-monitor -p tests/dns_capture_local.pcap
-d domain_names_local
./dns-monitor -p tests/dns_real_output.pcap
-d domain_names_real
```

Expected Output: The domain names extracted from the DNS traffic were expected to be unique and included all names found in the Question, Answer, Authority, and Additional sections of the DNS messages.

Local Server Expected Output:

```
ns1.test.local
test.local
mail.test.local
localhost
alias.test.local
www.test.local
_service._tcp.test.local
```

Real DNS Server Expected Output:

```
google.com
smtp.google.com
```

Testing Results: The generated output files (`domain_names_local` and `domain_names_real`) were compared with manually created expected output files (`expected_domain_names_local`
and `expected_domain_names_real`) using the `diff` command:

```
diff expected_domain_names_local domain_names_local
diff expected_domain_names_real domain_names_real
```

No differences were found, confirming that the program accurately extracted unique domain names from all DNS messages and logged them into the specified files as expected.

6.3.2 Testing to write information from 2 pcap files into 1 domain file

The program was executed with the following commands:

```
./dns-monitor -p tests/dns_capture_local.pcap
-d tests/domain_names
./dns-monitor -p tests/dns_real_output.pcap
-d tests/domain_names
```

Expected Output: The domain names extracted from the DNS traffic were expected to be unique and included all names found in the Question, Answer, Authority, and Additional sections of the DNS messages from 2 PCAP files.

```
google.com
smtp.google.com
ns1.test.local
test.local
mail.test.local
localhost
alias.test.local
www.test.local
_service._tcp.test.local
```

Testing Results: The output file `domain_names` were compared with manually created expected output file (`expected_domain_names` using the `diff` command:

```
diff expected_domain_names domain_names
```

No differences were found, confirming that the program accurately extracted unique domain names from all DNS messages and all PCAP files and logged them into the 1 file as expected.

6.3.3 Testing to Avoid Duplicate Entries in Domain Names File

To verify that the program does not add duplicate entries to the domain names file, an additional test was conducted. In this test, one domain name was manually removed from the domain names file, and the program was executed again with PCAP files to check whether only missing domain names are added to the file.

Initial Setup: The domain names file (`domain_names`) was manually edited to remove one entry. The updated file looked like this:

```
google.com
smtp.google.com
ns1.test.local
test.local
mail.test.local
localhost
alias.test.local
_service._tcp.test.local
```

The entry `www.test.local` was deliberately removed for this test.

Execution Commands: The program was executed again with the same PCAP files as in previous tests:

```
./dns-monitor -p tests/dns_capture_local.pcap
-d tests/domain_names
./dns-monitor -p tests/dns_real_output.pcap
-d tests/domain_names
```

Expected Output: The program was expected to append only the missing domain name `www.test.local` back to the file while leaving the existing entries unchanged. The expected final content of the file was:

```
google.com
smtp.google.com
ns1.test.local
test.local
mail.test.local
localhost
alias.test.local
_service._tcp.test.local
www.test.local
```

Testing Results: The resulting file (`domain_names`) was compared with the manually created expected output file (`expected_domain_names_final`) using the `diff` command:

```
diff expected_domain_names_final domain_names
```

No differences were found, confirming that the program correctly avoided duplicating entries and only added missing domain names to the file.

6.4 Domain-to-IP Translation Logging

The program's functionality to extract and log domain-to-IP translations into a file using the `-t` parameter was tested thoroughly. This feature searches for **A** and **AAAA** records in the Answer and Additional sections of DNS responses and logs unique translations in the specified file. Whenever a domain-to-IP mapping is encountered, it is stored in the file unless it is already present.

For testing this functionality, the same PCAP files that were used in previous tests were utilized again. The `dns-monitor` program was executed with the `-t` option, and the output file was checked for correctness.

6.4.1 Testing that file is created and information is written

The program was executed with the following commands:

```
./dns-monitor -p tests/dns_capture_local.pcap
-t translations_local
./dns-monitor -p tests/dns_real_output.pcap
-t translations_real
```

Expected Output: The domain-to-IP mappings extracted from the DNS traffic were expected to include unique mappings found in the Answer and Additional sections of the DNS responses.

Local Server Expected Output:

```
ns1.test.local 127.0.0.1
ns1.test.local ::1
mail.test.local 127.0.0.3
www.test.local 127.0.0.2
www.test.local 2001:db8::1
```

Real DNS Server Expected Output:

```
google.com 142.251.36.78
google.com 2a00:1450:4014:80e::200e
```

Testing Results: The generated output files (`translations_local` and `translations_real`) were compared with manually created expected output files (`expected_translations_local`
and `expected_translations_real`) using the `diff` command:

```
diff expected_translations_local translations_local
diff expected_translations_real translations_real
```

No differences were found, confirming that the program accurately extracted domain-to-IP mappings from all DNS responses and logged them into the specified files as expected.

6.4.2 Testing to write information from 2 pcap files into 1 translations file

The program was executed with the following commands:

```
./dns-monitor -p tests/dns_capture_local.pcap
-t tests/translations
./dns-monitor -p tests/dns_real_output.pcap
-t tests/translations
```

Expected Output: The domain-to-IP mappings extracted from the DNS traffic were expected to be unique and included all mappings found in the Answer and Additional sections of the DNS responses from 2 PCAP files.

```
ns1.test.local 127.0.0.1
ns1.test.local ::1
mail.test.local 127.0.0.3
www.test.local 127.0.0.2
www.test.local 2001:db8::1
google.com 142.251.36.78
google.com 2a00:1450:4014:80e::200e
```

Testing Results: The output file `translations` was compared with a manually created expected output file (`expected_translations`) using the `diff` command:

```
diff expected_translations translations
```

No differences were found, confirming that the program accurately extracted unique domain-to-IP mappings from all DNS responses and all PCAP files and logged them into a single file as expected.

6.4.3 Testing to Avoid Duplicate Entries in Translations File

To verify that the program does not add duplicate entries to the translations file, an additional test was conducted. In this test, one domain-to-IP mapping was manually removed from the translations file, and the program was executed again with PCAP files to check whether only missing mappings are added to the file.

Initial Setup: The translations file (`translations`) was manually edited to remove one entry. The updated file looked like this:

```
ns1.test.local 127.0.0.1
ns1.test.local ::1
mail.test.local 127.0.0.3
www.test.local 127.0.0.2
www.test.local 2001:db8::1
google.com 2a00:1450:4014:80e::200e
```

The entry `google.com 142.251.36.78` was deliberately removed for this test.

Execution Commands: The program was executed again with the same PCAP files as in previous tests:

```
./dns-monitor -p tests/dns_capture_local.pcap
-t tests/translations
./dns-monitor -p tests/dns_real_output.pcap
-t tests/translations
```

Expected Output: The program was expected to append only the missing domain-to-IP mapping `google.com 142.251.36.78` back to the file while leaving the existing entries unchanged. The expected final content of the file was:

```
ns1.test.local 127.0.0.1
ns1.test.local ::1
mail.test.local 127.0.0.3
www.test.local 127.0.0.2
www.test.local 2001:db8::1
google.com 2a00:1450:4014:80e::200e
google.com 142.251.36.78
```

Testing Results: The resulting file (`translations`) was compared with the manually created expected output file (`expected_translations_final`) using the `diff` command:

```
diff expected_translations_final translations
```

No differences were found, confirming that the program correctly avoided duplicating entries and only added missing domain-to-IP mappings to the file.

6.5 Stress Test on Interface

To ensure the robustness of the `dns-monitor` program in handling continuous DNS traffic, a stress test was conducted using the `-i` option on a live network interface. The goal of this test was not to validate specific outputs but to confirm that the program could continuously parse DNS packets in real-time under typical usage scenarios, such as web browsing, without crashes or memory issues.

Testing Setup: - The program was executed on the `lo` interface to monitor DNS traffic generated by routine browser activity. - A web browser was used to visit multiple websites, generating DNS queries for various domains. This included visiting popular sites like `google.com`, `youtube.com`, and `github.com`. - No specific output files were generated or compared during this test. The primary focus was on the program's ability to capture and process packets without errors.

Execution Command: The program was run on the live network interface with the following command:

```
sudo ./dns-monitor -i lo -v -d domainsfile -t translationfile
```

The captured DNS traffic was continuously printed to the terminal, demonstrating the program's real-time parsing capabilities.

Valgrind Memory Check: To ensure that the program handles memory correctly during prolonged execution, it was also executed with Valgrind:

```
sudo valgrind ./dns-monitor -i lo -v -d domainsfile
-t translationfile
```

Testing Results: - After CTRL+C interruption the program successfully captured and parsed DNS packets generated by browser activity without any crashes or significant delays. - Valgrind reported no memory leaks or errors during the entire test, confirming the program's memory management is robust under sustained traffic.

Conclusion: The stress test demonstrated that the `dns-monitor` program could reliably monitor real-time DNS traffic on a live interface, even under increased activity from routine web browsing. Combined with the clean Valgrind report, this validates the program's ability to handle live traffic parsing efficiently and without resource leaks.

```
==51121== HEAP SUMMARY:
==51121==     in use at exit: 0 bytes in 0 blocks
==51121==   total heap usage: 502 allocs, 502 frees, 677,254 bytes allocated
==51121==
==51121== All heap blocks were freed -- no leaks are possible
==51121==
==51121== For lists of detected and suppressed errors, rerun with: -s
==51121== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Valgrind result

7 Conclusion

The `dns-monitor` project fulfills all the requirements specified in the assignment. The application provides an effective tool for monitoring DNS communication and offers valuable features for network traffic analysis. Due to its modular design, the application can be easily extended with additional functionalities.

References

[1] *Domain names - implementation and specification* `https://datatracker.ietf.org/doc/html/rfc1035`

[2] *A DNS RR for specifying the location of services (DNS SRV)* `https://datatracker.ietf.org/doc/html/rfc2782`

[3] Tcpdump & Libpcap. `https://www.tcpdump.org`

[4] dn_expand(3) - Linux man page `https://linux.die.net/man/3/dn_expand`

[5] Wikipedia Domain Name System https://en.wikipedia.org/wiki/Domain_Name_System

[6] Wikipedia IPv4 https://en.wikipedia.org/wiki/IPv4

[7] Wikipedia IPv6 packet https://en.wikipedia.org/wiki/IPv6_packet