DNS monitor manual

Author: Vojtěch Kuchař - xkucha30

Date: 2024-11-17

Summary

Theory

DNS Architecture

The DNS (Domain Name System) architecture follows a hierarchical, distributed model to translate human-readable domain names into IP addresses. It consists of several key components:

- **DNS Resolver**: The resolver is responsible for initiating the query to find the IP address associated with a domain name.

- **DNS Server**: A server that stores the DNS records for domain names. When a resolver sends a query, the DNS server responds with the appropriate information or forwards the request to another server if it does not have the record.

DNS servers store records that map domain names to IP addresses. These records can be of different types, the most important for the nature of this project are A, AAAA, NS, MX, SOA, CNAME and SRV. Here is a brief description of each:

- **A Record**: Maps a domain name to an IPv4 address.
- **AAAA Record**: Maps a domain name to an IPv6 address.
- **NS (Name Server) Record**: Specifies authoritative name servers for a domain.
- **MX (Mail Exchange) Record**: Directs email messages to the correct mail server for a domain.
- **SOA (Start of Authority) Record**: Contains administrative information about a domain.
- **CNAME (Canonical Name) Record**: Aliases one domain name to another.
- **SRV (Service) Record**: Specifies the location of services in a domain.

More about the individual types of basic DNS records can be found in the RFC 1035 (https://tools.ietf.org/html/rfc1035) document.

The DNS operation process includes:

- **Query**: The process where a client (resolver) sends a request for a domain name to be resolved into an IP address.
- **Response**: The server's answer to the resolver, either with the requested information or a referral to another DNS server.

- **Caching**: To speed up future requests, both resolvers and DNS servers store responses in a local cache for a predefined time (TTL - Time to Live).

In nature of this project, we are focusing on monitoring DNS queries and responses.

Application draft

Projects functionality is divided into several parts:

1. **DNS Monitor**: The main application that captures and analyzes DNS queries and responses in real-time or from PCAP files.
2. **PCAP Parser**: A module that handles the parsing of PCAP files, extracting DNS packets for analysis.
3. **UDP Connection**: A module that establishes a UDP connection for live monitoring of DNS traffic.
4. **Translation**: A module that manages domain translations and IP address mappings. Also handles file operations related to domain and IP address pairs.
5. **Argument Parser**: A module that extracts and stores input arguments for the DNS monitor application, including options for specifying network interfaces, PCAP files, verbose mode, and domain translation files.

Here is a draft of the DNS monitor application dependencies:

Dependency graph

Implementation

This section describes the implementation of the DNS monitor application. The application is designed to capture and analyze DNS queries and responses in real-time and from PCAP files. The application is implemented in C++. The following subsections provide an overview of the application's architecture, key components, and testing procedures.

Source files

dns-monitor.cpp

The entry point of the DNS monitor application. The `run()` method of the DNSMonitor class is called to start the packet capture and DNS packets.

1. **Main Function**:

- `int main(int argc, char *argv[])`: The entry point of the DNS monitor application. It performs the following actions:
  - **DNSMonitor Initialization**: Creates an instance of the `DNSMonitor` class, passing command-line arguments.
  - **Monitoring Execution**: Calls the `run()` method of the `DNSMonitor` class to start the packet capture and DNS query analysis process.

app.cpp

The `app.cpp` file implements the `DNSMonitor` class, which handles DNS traffic monitoring. It processes command-line arguments via argumentParser class and either analyzes a PCAP file or establishes a real-time UDP connection based on the input. Key functions include parsing the PCAP file with `PCAPParser` or setting up a UDP connection for live monitoring, depending on the provided arguments.

1. **DNSMonitor Constructor**:
   - `DNSMonitor::DNSMonitor(int argc, char *argv[])`: Initializes the `DNSMonitor` class and parses the command-line arguments using the `parser.parseArguments` method to store them in the `args` object.
2. **Run Method**:
   - `int DNSMonitor::run()`: Determines whether to handle a PCAP file or establish a real-time UDP connection based on the input arguments:
     - **PCAP File Handling**: If the `-p` flag is provided, it calls `handlePCAPFile` to process the PCAP file.
     - **UDP Connection Handling**: If no PCAP file is specified, it calls `handleUDPConnection` to set up a UDP connection for live monitoring.
3. **Handle PCAP File**:
   - `bool DNSMonitor::handlePCAPFile()`: Processes a PCAP file using the `PCAPParser` class and returns `true` if parsing is successful.
4. **Handle UDP Connection**:
   - `bool DNSMonitor::handleUDPConnection()`: Establishes a UDP connection for live monitoring:
     - **UDP Connection Setup**: Creates and starts the `UDPConnection`. If successful, it returns `true`.

argumentParser.cpp

The `argumentParser.cpp` file implements functionality to extract and store input arguments for the DNS monitor application, including options for specifying network interfaces, PCAP files, verbose mode, and domain translation files.

1. **Parse Arguments**:
   - `void argumentParser::parseArguments(int argc, char *argv[], inputArguments &out)`: This method processes the command-line arguments passed to the application and stores them in

the `inputArguments` structure. It performs the following actions:

- **Help Option**: If `-h` or `--help` is specified, it prints the usage instructions for the program and exits.
- **Interface Argument**: If `-i` is provided, it stores the interface name in `out.interface` and sets `out.i` to `true`.
- **PCAP File Argument**: If `-p` is provided, it stores the PCAP file name in `out.pcapFile` and sets `out.p` to `true`.
- **Verbose Mode**: If `-v` is provided, it sets `out.v` to `true`, enabling verbose mode.
- **Domains File**: If `-d` is provided, it stores the domain names file path in `out.domainsFile` and sets `out.d` to `true`.
- **Translations File**: If `-t` is provided, it stores the translations file path in `out.translationsFile` and sets `out.t` to `true`.
- **Error Handling**: For each argument, the method checks for correct usage and prints an error message and exits if any argument is malformed or unknown.

pcap.cpp

The `pcap.cpp` file implements functionality for opening, parsing, and managing PCAP files, as well as extracting DNS packets from them. It contains methods for handling PCAP file operations and interacting with the `DNSParser` class to process DNS data.

1. **PCAPParser Constructor**:

   - `PCAPParser::PCAPParser(const inputArguments &args)`: Initializes the `PCAPParser` object with the provided input arguments (`args`). It prepares the object by setting the error buffer and initializing the PCAP handle to `nullptr`.

2. **Open File**:

   - `bool PCAPParser::openFile()`: Attempts to open the specified PCAP file in offline mode using `pcap_open_offline`. If the file cannot be opened, an error message is printed and the method returns `false`. If successful, it returns `true`.

3. **Close File**:

   - `void PCAPParser::closeFile()`: Closes the PCAP file handle using `pcap_close` and sets the handle to `nullptr` to ensure that the file is properly closed.

4. **Get DNS Offset**:

   - `int PCAPParser::getDNSOffset(const unsigned char *packet, int length) const`: Determines the offset of the DNS data within a packet by calling the `DNSParser::isDNSPacket` method. This method returns the offset of the DNS header if the packet contains DNS data, or `-1` if it does not.

5. **Parse File**:

- `int PCAPParser::parseFile()`: This method handles the main file parsing process:
  - **File Opening**: It first attempts to open the PCAP file using `openFile()`.
  - **Packet Processing**: It then enters a loop, processing each packet in the PCAP file. For each packet, it checks if the packet contains DNS data by calling `getDNSOffset`. If DNS data is found, it invokes the `DNSParser::parseRawPacket` method to parse and process the packet.
  - **File Closing**: Once all packets have been processed, the PCAP file is closed using `closeFile()`.

translation.cpp

The `translation.cpp` file implements functionality for managing domain translations and performing file operations related to domain and IP address pairs. It provides methods for loading translations, printing them, handling file operations, and ensuring domain names are correctly formatted.

1. **Load Translation**:
   - `void Translation::loadTranslation(std::string domain, std::string ip)`: Adds a new translation (domain to IP address) to the `translations` vector, which stores all domain-IP pairs.
2. **Print Translations**:
   - `void Translation::printTranslations()`: Iterates through the stored translations and writes each domain-IP pair to the file specified by `tfilepath`. If the IP address is empty, the translation is skipped.
3. **Print Domains**:
   - `void Translation::printDomains()`: Iterates through the stored translations and writes each domain name (without the IP address) to the file specified by `dfilepath`.
4. **Open File**:
   - `bool Translation::openFile(std::string openFile)`: Opens the specified file for writing in append mode. If the file cannot be opened, it prints an error message and returns `false`. If successful, it returns `true`.
5. **Close File**:
   - `void Translation::closeFile()`: Closes the currently open file if it is open.
6. **Remove Trailing Dots**:
   - `std::string Translation::removeTrailingDots(const std::string &line)`: Removes any trailing dots from the provided string (typically used for domain names) and returns the cleaned string.
7. **Remove Empty Lines**:
   - `bool Translation::removeEmptyLines(std::string &filePath)`: Reads the file at the given `filePath`, removes empty lines, and overwrites the file with the cleaned content. It also trims trailing dots

from each line. Returns `false` if the file cannot be opened.

8. **Write Line**:

   - `bool Translation::writeLine(const std::string &line, std::string &filePath)`: Writes a line to the specified file at `filePath`, ensuring that the line is properly formatted (removes trailing dots) and does not already exist in the file. Returns `false` if the line does not contain a dot or if the file cannot be opened.

udp.cpp

The `udp.cpp` file implements the `UDPConnection` class, which manages UDP network connections for receiving DNS packets. It includes functionality for creating UDP sockets, processing incoming packets, handling signals for graceful termination, and setting up proper socket communication for both IPv4 and IPv6 addresses.

1. **UDPConnection Constructor**:

   - `UDPConnection::UDPConnection(const inputArguments &args)`: Initializes the `UDPConnection` object, sets the arguments (`args`), and initializes the UDP socket to `-1`. It also sets the global instance pointer `g_udpConnectionInstance` to the current instance of `UDPConnection`.

2. **UDPConnection Destructor**:

   - `UDPConnection::~UDPConnection()`: Closes the UDP socket (if open) and resets the global instance pointer `g_udpConnectionInstance` to `nullptr` upon destruction of the object.

3. **Start Method**:

   - `int UDPConnection::start()`: The main method for managing the UDP connection:
     - **Socket Creation**: First, it calls `createSocket` to initialize the UDP sockets.
     - **Signal Handling**: It sets up a signal handler to allow graceful termination using `setupSignalHandler`.
     - **Packet Processing Loop**: Enters an infinite loop where it waits for incoming packets on both the IPv4 and IPv6 sockets using `select`. If data is available, it processes the packet and checks for DNS content using `DNSParser`.
     - **Packet Parsing**: If the packet contains DNS data, it passes the packet to the `DNSParser::parseRawPacket` method for analysis.

4. **Signal Handling**:

   - `void UDPConnection::handleSignal(int signum)`: Handles termination signals (like SIGINT), gracefully closing the UDP socket and exiting the program.
   - `void UDPConnection::signalHandler(int signum, UDPConnection *instance)`: A helper function to call `handleSignal` using the global instance of `UDPConnection`.
   - `void UDPConnection::setupSignalHandler()`: Sets up a signal handler for SIGINT (Ctrl+C) to ensure the server is terminated gracefully.

5. **Create Socket**:

   - `bool UDPConnection::createSocket()`: Creates two raw UDP sockets—one for IPv4 (`udp_socket`) and one for IPv6 (`udp_socket6`). If either socket creation fails, an error message is printed, and the method returns `false`. Otherwise, it returns `true`.

6. **Global Instance Pointer**:

   - The class uses a global pointer `g_udpConnectionInstance` to refer to the current instance of `UDPConnection`, which is accessed in the signal handler for graceful termination.

dns.cpp

This file handles the parsing and processing of DNS packets. It extracts DNS header and section data from raw packets, including questions, answers, authority, and additional sections. It also prints detailed DNS information depending on the verbosity level.

1. **parse Raw Packet**:

   - `void DNSParser::parseRawPacket(unsigned char *packet, ssize_t size, struct pcap_pkthdr captureHeader, int offset)`: Processes a raw DNS packet, identifies IPv4 or IPv6, parses the DNS header, and extracts section data.

2. **parse DNS Header**:

   - `void DNSParser::parseDNSHeader(const std::vector<uint8_t> &packet, DNSHeader *header)`: Parses the DNS header, extracting fields like ID, flags, and section counts (questions, answers, authorities, and additionals).

3. **parse DNS Packet**:

   - `void DNSParser::parseDNSPacket(const std::vector<uint8_t> &packet, DNSHeader *header, DNSSections *sections)`: Parses the DNS packet to extract and store question, answer, authority, and additional sections.

4. **handle DNS Data**:

   - `void DNSParser::handleDNSData(const std::vector<uint8_t> &packet, DNSHeader *dnsHeader, IPInfo *ipInfo, DNSSections *sections, char *dateTime)`: Handles and prints DNS data, displaying either a summary or detailed packet information based on verbosity settings.

5. **read DomainName**:

   - `std::string DNSParser::readDomainName(const std::vector<uint8_t> &data, size_t &offset)`: Reads a domain name from the packet, handling compression using pointers (RFC 1035).

6. **parse Resource Record**:

- `void DNSParser::parseResourceRecord(const std::vector<uint8_t> &data, size_t &offset)`: Parses a DNS resource record, handling different record types and filling the corresponding `ResourceRecord` structure.

7. **print Sections**:

   - `void DNSParser::printSections(DNSSections *sections, const std::vector<uint8_t> &packet)`: Prints the sections (questions, answers, authorities, additionals) from the DNS packet, respecting verbosity settings.

8. **print Question Section**:

   - `void DNSParser::printQuestionSection(const std::vector<QuestionSection> &questions)`: Prints the question section from the DNS packet.

9. **print Resource Record**:

   - `void DNSParser::printResourceRecord(const ResourceRecord &record, const std::vector<uint8_t> &packet)`: Prints details of a single DNS resource record (name, type, TTL, etc.).

10. **print Bytes**:

    - `void printBytes(const unsigned char *data, int size)`: Prints the raw hexadecimal representation of a data buffer.

11. **DNSParser::ipv6ToString**:

    - `std::string DNSParser::ipv6ToString(const std::vector<uint8_t> &rData)`: Converts an IPv6 address from binary format to its string representation, handling "::" compression.

Header files

app.h

Description Defines the `DNSMonitor` class for DNS monitoring tasks, including handling command-line arguments, processing PCAP files, and establishing UDP connections for real-time DNS monitoring.

Class: `DNSMonitor`

- **Constructor**: Initializes by parsing command-line arguments.
- **Methods**:
  - `run()`
  - `handlePCAPFile()`
  - `handleUDPConnection()`

argumentParser.h

Description

Defines the `inputArguments` structure and the `argumentParser` class for parsing and handling command-line arguments. Provides functionality to extract and store input arguments, such as interface names, PCAP files, and domain translation files.

Struct: `inputArguments`

- `interface`: Name of the interface to listen on.
- `pcapFile`: Name of the PCAP file to process.
- `domainsFile`: Name of the domains file.
- `translationsFile`: Name of the translations file.
- `i`: Flag for interface.
- `p`: Flag for PCAP file.
- `v`: Flag for verbose.
- `d`: Flag for domains file.
- `t`: Flag for translations file.

Class: `argumentParser`

- **Methods**:
  - `parseArguments()`

pcap.h

Description

Defines functionality for opening and parsing PCAP files. Includes methods for extracting DNS packets and managing PCAP file operations.

Class: `PCAPParser`

- **Constructor**: Initializes with the provided input arguments, setting up the error buffer and preparing the PCAP handle.
- **Methods**:
  - `parseFile()`
  - `openFile()`
  - `closeFile()`
  - `getDNSOffset()`

translation.h

Description

Provides functionality for managing domain translations and file operations. Includes methods for loading, printing, and manipulating domain-IP mappings, as well as handling related file I/O tasks.

Struct: `TranslationStruct`

- `domainName`: The domain name to be translated.
- `ip`: The IP address associated with the domain.

Class: `Translation`

- **Constructor**: Initializes with the domain and translation file paths.
- **Methods**:
    - `loadTranslation()`
    - `printDomains()`
    - `printTranslations()`
    - `writeLine()`
    - `removeEmptyLines()`
    - `openFile()`
    - `closeFile()`
    - `removeTrailingDots()`

udp.h

Description

Defines the `UDPConnection` class for managing UDP network connections. Provides methods for setting up, managing, and gracefully shutting down a UDP connection, including signal handling and socket configuration.

Class: `UDPConnection`

- **Constructor**: Initializes the UDP connection using provided input arguments and sets up the global instance pointer.
- **Destructor**: Cleans up by closing the socket if open and resets the global instance pointer.
- **Methods**:
    - `start()`
    - `signalHandler()`
    - `createSocket()`
    - `setupSignalHandler()`
    - `handleSignal()`

dns.h

Description

Defines functionality for handling DNS packets, including parsing, extracting data, and printing details. Structures and classes for managing DNS headers, sections, and resource records are provided, along with utilities for working with raw DNS packet data.

Classes and Functions:

1. **IPInfo**

    - Represents both IPv4 and IPv6 information (source and destination IPs and ports).

2. **DNSHeader**
   - Represents the DNS header, including transaction ID, flags, and section counts (questions, answers, authorities, additional records).
3. **QuestionSection**
   - Represents a DNS query question, including the domain name, query type, and query class.
4. **ResourceRecord**
   - Represents a DNS resource record, containing the record's name, type, class, TTL, and resource data.
5. **DNSSections**
   - Holds sections of a DNS packet: questions, answers, authorities, and additionals.
6. **DNSParser**
   - Main class responsible for parsing raw DNS packet data.
     - Key methods include:
       - `parseRawPacket()`: Entry point for parsing DNS packets.
       - `isDNSPacket()`: Determines if a packet is a DNS packet.
       - `parseDNSHeader()`: Extracts DNS header information.
       - `parseDNSPacket()`: Parses the full DNS packet, filling sections.
       - `parseResourceRecord()`: Parses individual resource records.
       - `printBytes()`: Prints the raw bytes of data in hexadecimal format.
       - `handleDNSData()`: Handles the processed DNS data, including printing sections and records.
7. **Utility Functions**
   - `ipv6ToString()`: Converts raw IPv6 address bytes into human-readable format.
   - `printBytes()`: Prints raw packet bytes in a readable hexadecimal format.
   - `getPacketTimestamp()`: Extracts a timestamp from the packet capture header.

include.h

Description

This header file includes all the necessary libraries and headers used throughout the project. The file also defines macros and global constants.

Included Libraries:

- **Standard C++ Libraries**:
  - `<set>`, `<iostream>`, `<cstring>`, `<string>`, `<sstream>`, `<memory>`, `<fstream>`, `<algorithm>`, `<cctype>`: For basic functionality like string manipulation, I/O, file handling, and utility operations.

- **POSIX Libraries**:
  - `<unistd.h>`, `<stdlib.h>`, `<signal.h>`, `<sys/socket.h>`, `<sys/time.h>`, `<sys/types.h>`: Used for low-level operations such as socket handling, process management, time functions, and system utilities.
- **Networking Libraries**:
  - `<pcap.h>`: For working with packet capture files and network traffic monitoring.
  - `<netinet/in.h>`, `<arpa/inet.h>`, `<netinet/ip.h>`, `<netinet/ip6.h>`, `<netinet/udp.h>`, `<net/if.h>`, `<linux/if_ether.h>`: Provide definitions for handling IP, UDP, and Ethernet protocols, as well as working with IPv4 and IPv6 addresses.
  - `<ifaddrs.h>`: For retrieving network interface information.
  - `<err.h>`: Used for error reporting and handling.
- **Global Constants**:
  - `#define PORT 53`: Defines the default port for DNS, typically used in DNS queries.

Testing

It has been tested on multiple automated test files that are included in the `test` directory. Tests can be run running the `test.py` script. The script will run the program with the test files and compare the output with the expected output. The script will print the results of the tests. Tests can be run like this:

```
make tests
```

Test Files

Test file are located in the `test` directory.