Pouria Salehi katozi s192865

DNS Tunneling - Server side

INFO0010 Introduction to Computer Networking

Université de Liège
Faculté des Sciences Appliquées
Année académique 2022-2023

1 Introduction

This project focuses on the implementation of DNS tunneling, adhering to the RFC1035 protocol. DNS tunneling is a technique that involves transmitting data through DNS queries and responses, allowing covert communication.

2 Architecture

The architecture of the project is designed to operate as a Domain Name System (DNS) server that seamlessly integrates with an external Hypertext Transfer Protocol (HTTP) server. This integration allows the DNS server to dynamically retrieve data from the HTTP server in response to incoming DNS queries, enhancing the server's capabilities. The project has maximum modularity and it contains componenets such as Server, Clienthandler, Header, Query and DNSResponse. Each of this components do their task by followinf RFC 1035 protocal. The architecture is designed to ensure efficient and secure tunneling while adhering to the protocol's specifications.

2.1 Multi-Threading

The server class is implemented using a multi-threading approach, which provides several significant advantages, including :

— Improved Performance : The multi-threaded architecture enables the server to effectively manage numerous incoming requests concurrently, resulting in enhanced response times and overall performance.

— Enhanced Concurrency : Multiple clients can simultaneously initiate tunneling requests, ensuring efficient utilization of resources and mitigating potential bottlenecks.

— Scalability : The architecture is highly scalable, allowing for the creation of additional threads to meet growing tunneling demands.

— Reduced Latency : Concurrent processing minimizes latency for both clients and the server, leading to an improved user experience.

— Flexibility : The modular design facilitates the seamless integration of new threads, enabling the accommodation of future feature enhancements or increased workloads.

```
1   public void start() {
2       try (ServerSocket server = new ServerSocket(this.PORT)) {
3           while (true) {
4               Socket clientSocket = server.accept();
5               SocketAddress address =
                    clientSocket.getRemoteSocketAddress();
6               Clienthandler client = new Clienthandler(clientSocket,
                    domain, address);
7               Thread thread = new Thread(client);
8               thread.start();
9           }
10      } catch (IOException e) {
11          System.out.println(e.getMessage());
12          System.exit(-1);
13      }
14  }
```

2.2 Clienthandle Classe

The ClientHandler class is a Runnable class that manages a connection with one client. The receiveData() method first sets the TCP connection settings, such as the timeout for that specific connection, and gets the InputStream associated with that socket. The method then tries to read as many bytes as specified in the DNS TCP header, as defined by RFC 1035, to make a query with the received data. Finally, the method creates an HTTP instance to manage everything related to the decoded URL. If the HTTP GET request is successful, the method crafts and sends a valid DNS reply with an answer.

TCP connection settings, such as the timeout for that specific connection in order to ensure that the connection will not time out while the method is waiting to read data from the client.

```
    public void receiveData() {
        try {
            clientSocket.setTcpNoDelay(true);
            clientSocket.setSoTimeout(5000);
```

The method then tries to read as many bytes as specified in the DNS TCP header. The DNS TCP header specifies the length of the DNS query, so this ensures that the method reads the entire query.

```
            InputStream inputstream = clientSocket.getInputStream();
            // Read the first 2 byte of DNS header to get the leangth
               of query
            byte[] tmp = new byte[2];
            int Length = inputstream.read(tmp);
            if (Length != 2) {
            //Reporting error and closeing inputstream and client socket
            }
            int queryLength = ((tmp[0] & 0xFF) << 8) + (tmp[1] & 0xFF);
            byte[] queryBuffer = new byte[queryLength];
            int len = 0;
           // Read as many bytes as specified in the DNS TCP header
            while (len < queryLength) {
              len +=inputstream.read(queryBuffer,len,queryLength - len);
              if (len == -1) {
           //Reporting error and closeing inputstream and client socket
                }
            }
```

The method first creates a query and checks for errors. If the query is valid, the method checks the DNS response for issues. If there are no issues, the method sends the response to the client. If there is any issue, the method sends an error message to the client. If the query is valid, the method decodes it and creates an HTTP instance to manage it. The HTTP instance is used to send the query to a DNS server and receive the response. If the HTTP GET request is successful, the method crafts and sends a valid DNS reply with an answer. The DNS reply is crafted according to the RFC 1035 specification.

```
            Query query = new Query(queryLength, dataInput);
            DNSResponse response = new DNSResponse(query);
            if (query.CheckFormat()) {
                response.handleFormatError();
                sendResponse(response);
            } else if (query.RequestRefused(domain)) {
                response.handleRefused();
                sendResponse(response);
            } else {
                HttpGet http = new HttpGet(query.decode());
                http.httpRequest(domain);
                byte[] HTTPresponse = http.SendData();
                response.setData("TXT", HTTPresponse);
                sendResponse(response);
            }
```

Finally, the input stream and client socket are closed. If there are no exceptions, then the method returns. If there are any exceptions, then the method catches them and reports the error.
The run() method is the entry point for the ClientHandler thread. The method first tries to call the receiveData() method. If the receiveData() method throws an exception, the run() method catches the exception and handles it. Finally, the run() method closes any resources that were used by the ClientHandler thread.

```
@Override
    public void run() {
        receiveData();
    }
```

2.3 Header and Query Classes

These classes handle the parsing and creation of DNS header fields and query-specific fields. They facilitate the extraction, modification, and verification of critical DNS-related data. The Query class is particularly adept at assessing query validity, handling DNS errors, and decoding query data.

2.4 HttpGet Classe

The HttpGet class provides the capability to perform HTTP GET requests to an external HTTP server. It prepares, sends, and interprets the HTTP request-response. Additionally, it encodes the HTTP response data using Base64 for integration into DNS responses.

```
   public void httpRequest(String string) {
      HttpURLConnection connection = null;
      try {
         connection = (HttpURLConnection) this.url.openConnection();
         connection.setRequestMethod("GET");
         if (connection.getResponseCode() == HttpURLConnection.HTTP_OK)
            {
            BufferedReader input = new BufferedReader(new
               InputStreamReader(connection.getInputStream()));
            String tmp = new String();
            StringBuffer response = new StringBuffer();
            while ((tmp = input.readLine()) != null) {
               response.append(tmp);
               response.append("\r\n");
            }
        data =
           Base64.getEncoder().encode(response.toString().getBytes());
            input.close();
         }
      } catch (IOException e) {
         e.printStackTrace();
      } finally {
         if (connection != null) {
            connection.disconnect();
         }
      }
   }
```

3 Length Limit

The implementation of DNS tunneling in this project encompasses multiple factors. The HTTP response size encounters an upper threshold, dictated by the RDLENGTH field within the DNS answer section, confining the RDATA portion to a scope of 0 to 65535 bytes. To navigate this restriction, the project suggests adopting multiple answers or responses to transmit fragmented segments of the HTTP response. Furthermore, the inherent constraint on the length of a DNS message, determined by the first two bytes, leads to the consideration of message truncation and recursive queries as prospective remedies. Notably, in light of the constraints posed by IPv4, the RDATA field is capped at 60000 bytes. Consequently, the project explores the concept of tunneling and encapsulation, facilitating the encapsulation of IPv6 packets within IPv4 packets. These strategies collectively empower the project to effectively address diverse limitations and achieve robust DNS tunneling.

4 DNS Tunneling in practice

Sending DNS queries directly to an uncontrolled server presents challenges. The uncontrolled server may not be able to resolve queries from our server, which could render communication impossible. To overcome this, an intermediate DNS server can be employed to translate queries from the controlled server to the uncontrolled server, and vice versa. In this scenario, a practical approach would involve directing the query to the controlled server, which can then establish a connection with the uncontrolled DNS server. However, a potential limitation must be considered : the uncontrolled DNS server might be unable to resolve the specific DNS queries requested by the client. To address this concern, a specialized type of query can be utilized.

5 Limits and Possible Improvements

The current server implementation could be improved by making it more flexible and efficient. One way to do this is to implement the optional point 10 of the assignment. This point allows the maximum length of the HTTP response to be deduced from the variable type short storing RDLENGTH.

Currently, the program fetches and stores the entire HTTP response, even if its length significantly exceeds the maximum. A potential improvement would be to only accept the maximum number of bytes upon receiving the response. This would lead to potential time and space optimization.

Another potential improvement would be to return the TXT records instead of the entire web page. This would be more efficient, as it would only require us to parse the TXT records, rather than the entire web page.

Finally, Storing the query as a byte array instead of a string could improve performance of the program but I am not sure.