Brno University of Technology

Faculty of Information Technology

Network Applications and Network Administration

Tunneling data traffic through DNS queries

Project documentation

Brno November 14, 2022

Maksim Tikhonov

Abstract

DNS tunneling is a type of cyberattack that exploits domain name system protocol in order to pass data through queries. It is considered very powerful type of attack, since DNS traffic is usually considered safe which allows attackers to bypass security gateways.

This project serves more as proof of concept of client-to-server data transfer through DNS queries than an actual implementation of tunneling toolkit.

Contents

1 Introduction

1.1 Domain Name System

DNS is one of the most critical and most basic among all Internet services. It identifies computers that are reachable through the Internet and serves as routing table with human-readable addresses associated with their numerical representations.

1.2 Sending data through DNS tunnel

The idea behind data transfer via DNS queries is that data can be encoded in the `query name` field of query structure in a way described by RFC 1035 [1]: `query name` represents human-readable form of domain name it consists of several labels each separated by the period. Length of entire domain name should not exceed 255 (including null terminator at the end) length of each label should not exceed 63 (w/o null terminator). It's only allowed to use alphanumeric characters and hyphen.

0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5
ID															
QR	Opcode				AA	TC	RD	RA	Z			RC			
Question records count															
Answer records count															
Name server records count															
Name server records count															
Additional records count															
Query name															
Query type															
Query class															

Figure 1: Structure of DNS query

1.3 Project functionality

The project is written in C language and includes two applications: DNS sender (client) and DNS receiver (server).

Server waits for the messages on all addresses on port 53 (DNS port), parses them on receival and stores decoded data given by received query. After data processing is over, it sends DNS answer back to the sender.

Sender, given some data (provided either by file or by standard input), encodes it in DNS query and sends it to the upstream IP address. After query is sent, sender awaits for the server's answer.

Applications support communication with IPv4 protocol on Internet layer and UDP on Transport layer.

2 Implementation

2.1 Argument validation

This subsection describes parsing of command line arguments of each of the two applications.

2.1.1 Client

Client has the following input format:
`dns_sender [-u UPSTREAM_DNS_IP] {BASE_HOST} {DST_FILEPATH} [SRC_FILEPATH]`

List of arguments:

- `UPSTREAM_DNS_IP` is optional; if argument is provided, sender will validate it by the IPv4 regex; if not provided, sender will send data to the address of internal DNS resolver of `systemd-resolved` provided by `resolv.conf`.
- `BASE_HOST` is mandatory; the argument value's length should not exceed 150 characters (it is necessary to leave some space for the actual data). Each label's length should not exceed 63 characters and `base host` should not start with period character.
- `DST_FILEPATH` is mandatory; the length of destination file path should not exceed 4096 characters (maximum length for file path on Linux distributions).
- `SRC_FILEPATH` is optional; if argument is provided, check if file exists; if not provided, sender will read data from standard input.

2.1.2 Server

Server has the following input format:
`dns_receiver {BASE_HOST} {DST_DIRPATH}`

List of arguments:

- `BASE_HOST` is mandatory; parsed in the same fashion as the `base host` argument for the sender.
- `DST_DIRPATH` is mandatory; parsed after the receival of `destination file path` from the client.

2.2 Client pipeline

This subsection describes the communication and data processing from sender's perspective.

2.2.1 Reading loop

After setting all the required parameters and variables and parsing host's base into separate labels, sender starts reading source file until it's `EOF`. For the first couple of queries it sends encoded file path name, then it starts sending encoded data chunks, and, after the end of file is reached, it sends the packet that indicates the end of transfer. Query's identifier is randomly generated in a specific ranges to indicate whether some query contains file name, chunk of data or it signifies the end of file transition. The ranges are 0 to 65513 for data transfers, 65514 to 65534 for file transfers and 65535 for the last chunk. There is also a host's base at the end of every `query name`.

2.2.2 Data transfer

After the creation of DNS packet client sends it to the server and waits for the answer. If the answer is not received within a second, client will be resending the query until it receives an answer.

Upon reaching the end of file, client sends packet that indicates the end of file transfer and waits for the answer that will contain size of resulting file in `rdata` field.

2.3 Server pipeline

This subsection describes the communication and data processing from receiver's perspective.

2.3.1 Receiving loop

After all necessary variables and parameters are set and host's base is parsed into separate labels, receiver starts listening on all addresses on DNS port. Once the data has been received, server decodes it and parses, then it either appends data to the full file path, writes it to the file or completes the transfer of the file; all dependent on DNS header's identifier value.

2.3.2 Data transfer

As soon as received data fragment has been processed, server sends DNS response to confirm the data chunk receival and starts waiting for the next transfer. If there is no transfer within next 2 seconds, then the receiver returns to its initial state and starts waiting for the new file, leaving previous one incomplete.

Upon receiving the packet that denotes the end of file transfer, receiver sends back the size of resulting file and returns to initial state, waiting for the start of transfer of the next file.

2.3.3 Data processing

If base host of `query name` of received data doesn't match the base host given by the argument, the received chunk is ignored.

If the full file path does not exist, server creates it.

2.4 Encoding/Decoding

It is required to only use alphanumeric symbols for query labels, thus it's mandatory to use data encoder/decoder (we may also want to send non-alphanumeric symbols). For this purpose I used `base32` encoding implementation from Google [2] since it only uses capitalised latin characters plus numbers from 2 to 7.

3 Testing

Testing is executed by the `test/test.py` file.

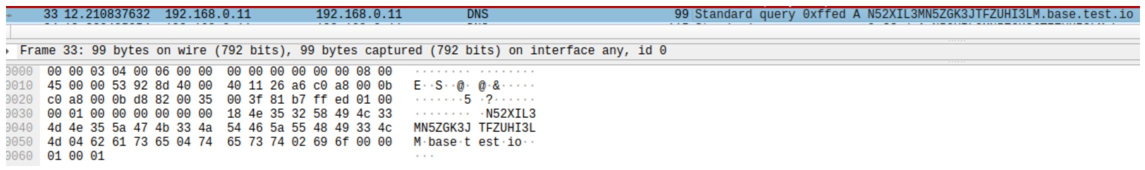
Firstly, it tests applications' response to invalid arguments.

Then it tests it's ability to create file paths that do not exist.

Finally, it executes sequential sending of 10 randomly generated files consisting of 1000 characters from ascii subset (newline plus characters in range of 32 to 127).

4 Efficiency measuring

The data transfer rate was measured by transferring a file with a size of 2279220 bytes (2.2 MB). The transfer, according to wireshark log, lasted for 7.5 seconds (starting at 12.21 finishing at 19.73).

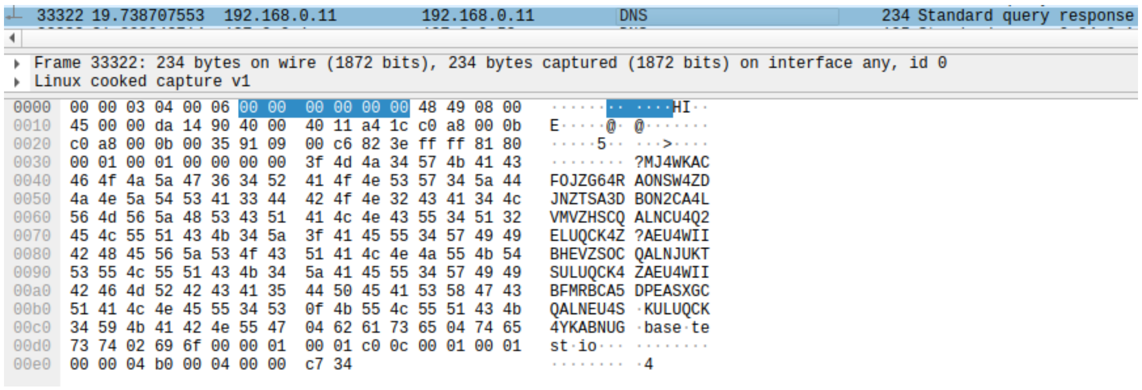


```
33 12.210837632  192.168.0.11          192.168.0.11          DNS          99 Standard query 0xffed A N52XIL3MN5ZGK3JTFZUHI3LM.base.test.io

Frame 33: 99 bytes on wire (792 bits), 99 bytes captured (792 bits) on interface any, id 0

0000  00 00 03 04 00 06 00 00  00 00 00 00 00 00 08 00   ········ ········
0010  45 00 00 53 92 8d 40 00  40 11 26 a6 c0 a8 00 0b   E··S··@· @·&·····
0020  c0 a8 00 0b d8 82 00 35  00 3f 81 b7 ff ed 01 00   ·······5 ·?······
0030  00 01 00 00 00 00 00 00  18 4e 35 32 58 49 4c 33   ········ ·N52XIL3
0040  4d 4e 35 5a 47 4b 33 4a  54 46 5a 55 48 49 33 4c   MN5ZGK3J TFZUHI3L
0050  4d 04 62 61 73 65 04 74  65 73 74 02 69 6f 00 00   M·base·t est·io··
0060  01 00 01                                           ···
```

Figure 2: Packet #33 which starts the data transfer



```
33322 19.738707553  192.168.0.11        192.168.0.11        DNS        234 Standard query response

▸ Frame 33322: 234 bytes on wire (1872 bits), 234 bytes captured (1872 bits) on interface any, id 0
▸ Linux cooked capture v1
0000  00 00 03 04 00 06 00 00  00 00 00 00 48 49 08 00   ········ ····HI··
0010  45 00 00 da 14 90 40 00  40 11 a4 1c c0 a8 00 0b   E·····@· @·······
0020  c0 a8 00 0b 00 35 91 09  00 c6 82 3e ff ff 81 80   ·····5·· ···>····
0030  00 01 00 01 00 00 00 00  3f 4d 4a 34 57 4b 41 43   ········ ?MJ4WKAC
0040  46 4f 4a 5a 47 36 34 52  41 4f 4e 53 57 34 5a 44   FOJZG64R AONSW4ZD
0050  4a 4e 5a 54 53 41 33 44  42 4f 4e 32 43 41 34 4c   JNZTSA3D BON2CA4L
0060  56 4d 56 5a 48 53 43 51  41 4c 4e 43 55 34 51 32   VMVZHSCQ ALNCU4Q2
0070  45 4c 55 51 43 4b 34 5a  3f 41 45 55 34 57 49 49   ELUQCK4Z ?AEU4WII
0080  42 48 45 56 5a 53 4f 43  51 41 4c 4e 4a 55 4b 54   BHEVZSOC QALNJUKT
0090  53 55 4c 55 51 43 4b 34  5a 41 45 55 34 57 49 49   SULUQCK4 ZAEU4WII
00a0  42 46 4d 52 42 43 41 35  44 50 45 41 53 58 47 43   BFMRBCA5 DPEASXGC
00b0  51 41 4c 4e 45 55 34 53  0f 4b 55 4c 55 51 43 4b   QALNEU4S ·KULUQCK
00c0  34 59 4b 41 42 4e 55 47  04 62 61 73 65 04 74 65   4YKABNUG ·base·te
00d0  73 74 02 69 6f 00 00 01  00 01 c0 0c 00 01 00 01   st·io··· ········
00e0  00 00 04 b0 00 04 00 00  c7 34                     ········ ·4
```

Figure 3: Packet #33322 which ends the data transfer

References

[1] P. Mockapetris. Domain names - implementation and specification. RFC 1035, RFC Editor, 11 1987.

[2] M. Gutschke. Base32 implementation. `https://github.com/google/google-authenticator-libpam/blob/master/src/base32.c`, 2010.