DNS Tunneling

Abstract:

We demonstrate a technique to exfiltrate files out of secure networks by tunneling data via Domain Name Service. This technique can further be extended to tunnel all kinds of network data, essentially granting complete (albeit slow) access to the open internet from within a secured network. And the best part is that the password for open access may not be known at all.

Introduction

Primer on DNS

The domain name service refers to a shared, open access directory that maps ASCII character sequences to addresses and the related protocol that allows for access and retrieval of this information.

Most users on the internet can go about their business without even knowing about DNS. However, the internet, will not be as it is, without the domain name service.

We know that each computer connected to the network has some sort of address (although this is a naive simplification, it is one that was applicable for the initial years of internet). Whenever a user wants to access a computer that has made some documents public, he needs to enter the address of that computer. The problem is that these addresses are pretty hard to memorize. Even more so when there are millions of computers connected to the network and a user may frequently want to access a hundred or more computers. In this case, the powers that be came up with a scheme to make it easier for users to access other computers simply by typing in ASCII strings. These strings, called “domains”, are easy to remember and simplify computer addressing.

Thus a user only needs to remember the domain name to access documents on some other computer.

Tunneling

“ Tunneling, also known as "port forwarding," is the transmission of data intended for use only within a private, usually corporate network through a public network in such a way that the routing **node***s in the public network are unaware that the transmission is part of a private network.”* [1]

It may not seem like much, but the concept of tunneling opens up a whole world of possibilities. Ever wondered, how some chinese internet users are able to post comments on YouTube, when YouTube is “officially” blocked in China? By tunneling, of course! They tunnel their traffic in innocuous looking packets going to some remote server in some other country where YouTube works just fine. But, this seemingly “ordinary” server is running a special software. The server strips the incoming traffic of all the encapsulation, and gets the actual packet out. Then it makes requests on behalf of the user back in china. Once it receives the response, it re-encapsulates it and sends it back to Mr. Chow. Mr. Chow does the whole process again on his machine i.e. strip the encapsulation and retrieve the packet. Mr. Chow's computer then gets this data out and displays it. Mr. Chow may not even notice that his data packets have had quite a journey to get to him. For him, it's just ordinary internet (of course, location-aware advertising may break the illusion).

But what are these innocuous looking packets that the Chinese government is willing to pass by

through the Great Firewall? DNS packets, of course!

DNS + Tunneling

Remember when we said everyone who is human and uses the internet needs the DNS. And people of china are also human. So DNS is a natural choice for encapsulating data.

There's only this one little problem. Since everyone needs DNS and people keep browsing the internet all the time, they keep making DNS requests all the time. DNS requests are nothing but requests for the address that is mapped by a certain domain name. To speed up the response of these requests, several computers on the internet, store these responses. So whenever a user makes a query, the request does not need to be sent to the "authoritative server" to retrieve the response. One of these "caching" servers can answer the query too.

The authoritative server for a domain is a computer which is the "authority" on the actual mapping for that domain.

If we look back at tunneling, we know that we need to have some degree of control over the "server" that the user (within a censored network) needs to contact. The server is, of course, outside this network. In case of DNS, this server is the authoritative server.

For a user to access tunnel data via DNS, the user must control the authority server for a particular domain. But therein lies the catch. Assuming that the user is within a network with limited access, how can he access anything outside of the local network? Well, that's where one of the features of DNS is exploited. Since DNS responses are "cached" by the cache servers described above, the cache servers need to have the requisite information in the first place. They too get the information from the authority.

Thus, when a user requests some DNS information, the local DNS cache server (if it does not have the answer), contacts DNS servers that it is connected to. If those servers too do not know the answer, they turn to their DNS peers (and this chain goes on). It's actually a little less peer-to-peer and more hierarchical. But we won't get to that.

So, in effect, the user gets an indirect line to the machine he controls (authority server) and thus can communicate with him. But he is restricted by the protocol of DNS communication. So he must find a way to communicate his requests within the confines of the DNS protocol.

Now, we go on to describe how DNS tunneling can be achieved. An example DNS sendfile program demonstrates a use case of exfiltrating files out of secure networks.

TECHNICAL DETAILS

Nameserver and A records

The first step is to obtain a public IP. Since we should be in control of an authority nameserver[2] for a domain, this nameserver should be publicly accessible. The nameserver itself has a domain name. This domain name is mapped to the public IP we mentioned above. The user looks for public IP of the nameserver in DNS servers higher up in the hierarchy. This is explained by the schematic below:

Fig. 1 Schematic diagram describing the “inverted triangle” DNS hierarchy.

The schematic in Fig. 1 does a fine job of demonstrating the DNS hierarchy. Whenever a node (connected to the internet) wants a DNS request serviced (eg, for domain t1.sahilmn.com), it contacts its local DNS server. After a few hops (given the answer is not cached), it contacts the root DNS server (root DNS server is actually one of 13 worldwide DNS servers). The root server redirects the user to the TLD server (top level domain name server). Examples of TLDs are com, net, org. The TLD server then redirects the user to sahilmn.com name server. This is the node we control.

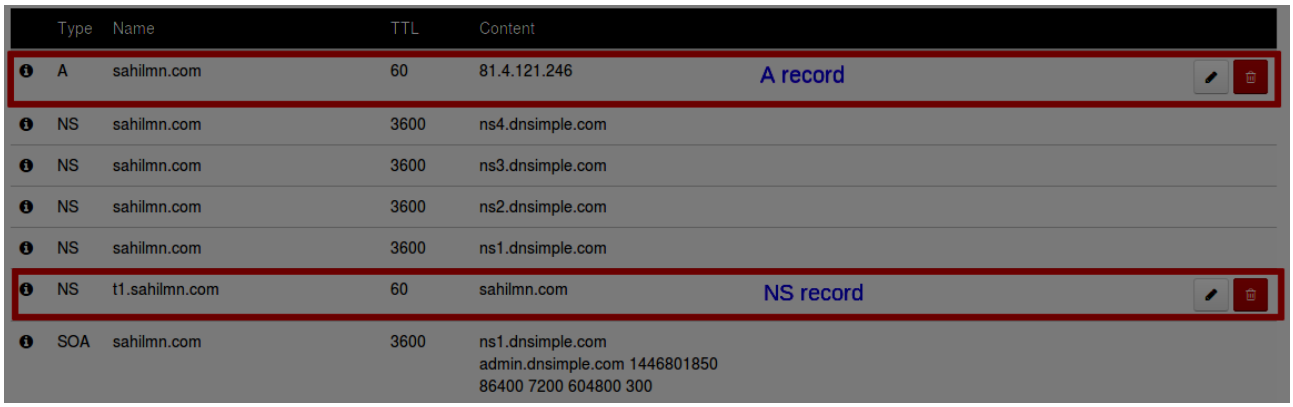
We set this node as authority for all subdomains (like example.sahilmn.com). Thus, if a user wants the address of a computer with domain name like example.sahilmn.com, it must first ask the computer having the domain name sahilmn.com for the address.

We described the domain name service as a sort of distributed directory. Since we control an authority name server, we control a very small part of this directory. We also described that DNS is a mapping from domain name to addresses. Each of these mappings may be called a record.

These mappings may mean different things depending on the “type” of the record. Some of the common record types are 'A' record, 'AAAA' record, 'NS' record, 'TXT' record, etc.

An 'A' record is the most commonly known type of record. It is a mapping from a domain name to an IPv4 address. An 'NS' record is a mapping from a domain name to the address / domain name of its authority server.

To achieve the desired configuration, we add the following records to the DNS directory:



That is:

Name	Type	Content
sahilmn	A	81.4.121.246
t1.sahilmn.com	NS	sahilmn.com

This means that all requests for any address example.t1.sahilmn.com will be redirected to the computer which is mapped by sahilmn.com (here: 81.4.121.246). This is the public IP controlled by the user.

Sending Data

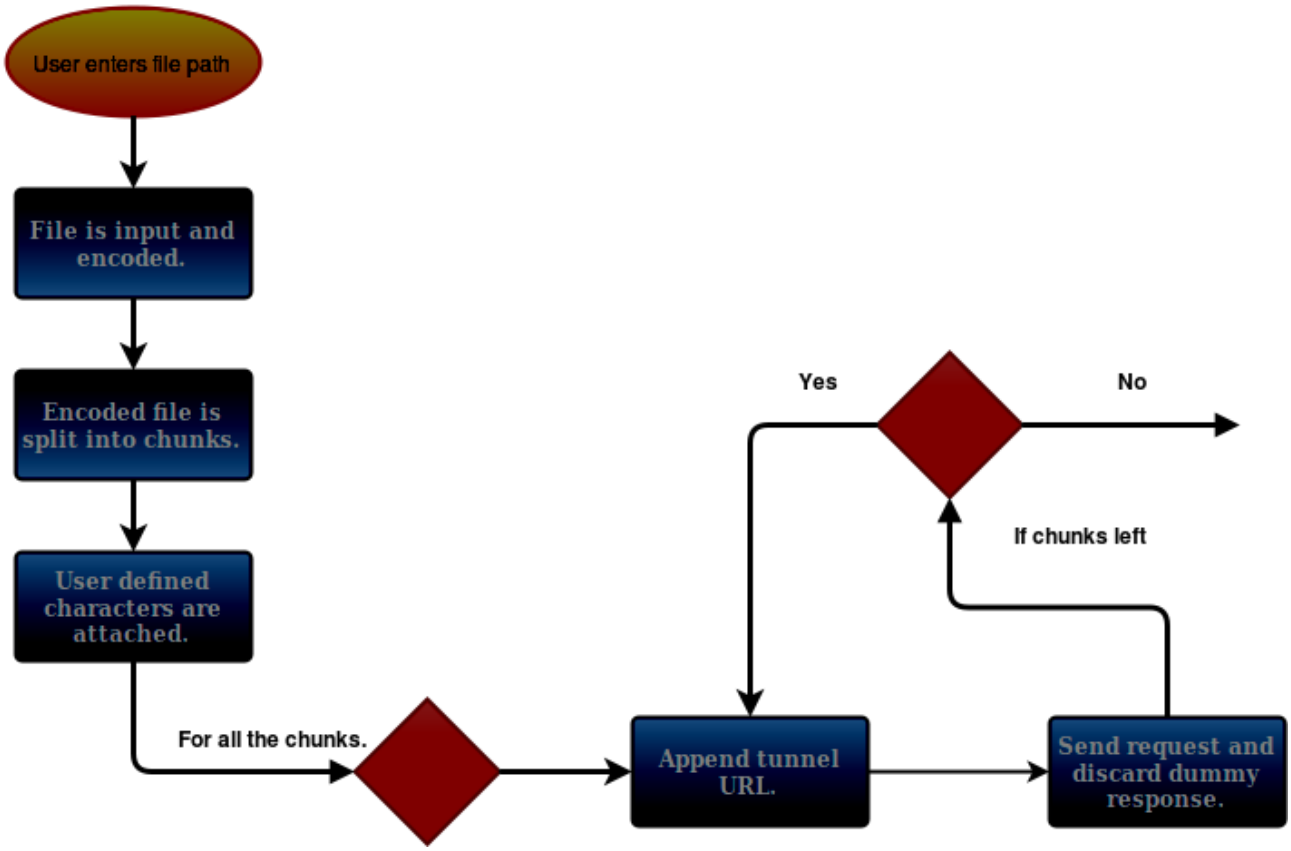
Since we know that the nameserver for all requests of the kind example.t1.sahilmn.com will be sent to 81.4.121.246 for resolution [3], we can program this computer to handle these requests. Moreover, we don't actually need to have any computers mapped to the addresses like example.t1.sahilmn.com. Since, all we are doing is asking for domain name resolution, the nameserver just responds with information that conforms to the DNS protocol. This information however, can be anything. That opens up possibilities for play :).

Since, we don't have any computer mapped to the addresses like example.t1.sahilmn.com, the user (who is controlling the client from within the censored/secured network), needs to know that the information returned by the nameserver is not actionable in the sense that most other DNS responses are. These requests do not necessarily contain any address to which HTTP or any other requests may be sent.

Thus, we can communicate with the server by inserting information in arbitrary domain names and receive DNS responses with useful data. On both sides, the client and the server, we do the necessary encapsulation/decapsulation to get the required data.

We now describe the specific approach taken in our example program.

Client Flow



The client program takes a file path as input and transfers the contents of that file to a remote server controlled by the user. All of this is done by encapsulating the data in multiple DNS requests.

Splitting and Sending Data

Since the file can be large, and DNS packets can carry only a limited amount of data, we must first split the data. Note that our data reaches the nameserver as TXT record requests. The name server extracts the relevant part of the URL section of the request.

The data is sent as part of the URL. Thus, it must have the same character set allowed for a URL. For this, we encode the data using base64 encoding. We use a publicly available library for this.

An example of this encoding is:

```
Input string:  “this is some data”
Output:  “dGhpcyBpcyBzb21lIGRhdGE=”
```

Note that input and output are without quotes.

Once the complete input data is encoded, we split it into chunks of 61 characters.
According to the RFC 1034[4], domain names may not be more than 255 characters and each token label of the domain name (like example in example.t1.sahilmn.com or google in www.google.com) must not be more than 63 characters.

To communicate within the confines of the DNS, we define our own protocol that both the server

and the client understand.

A request/response begins with the character 's'. Here, request/response refers to the decapsulated text at both client and server. If a request/response is split into chunks, it must have 'c' at the beginning and 'c' at the end of the response. The last chunk must have an 'e' at the end of a response.

Thus, an example request (split across 3 DNS requests) is:

```
sFNlcHQuIDIwMDYgCgpGb3IgYWNjZXNzaW5nIHRvIHRoZSBzaWduYXR1cmUgREc.t1.sahilmn.com
cIgeW91IG11c3QgZmlsbCBpbiBhbmQgc2VuZCB1cyBOaGUgZm9sbG93aW5nIGRc.t1.sahilmn.com
cvY3VtZW50OIHdpdGggYSBjZXJOaWZpZWQgSUQgY29weS4gCg==e.t1.sahilmn.com
```

The aforementioned URLs are used to create DNS requests. These DNS request are sent through the cache server, and after a few hops, it reaches the authoritative server. The nameserver extracts the URL, strips the starting and ending characters, concatenates the characters, decodes the resultant string and prints the final output.

DNS Request/Response

The DNS request is created in the client. The server too creates a dummy response for the request.

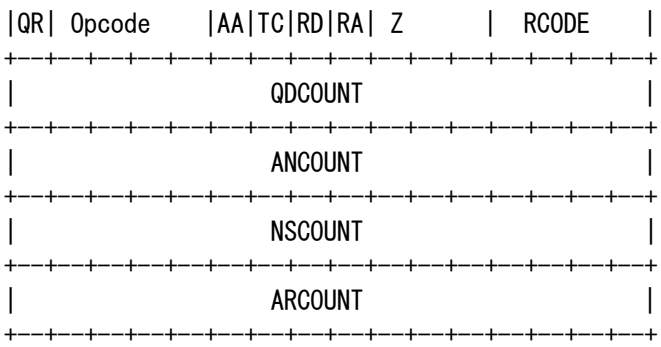
The schematic describes the DNS request/response. Both the request and response have the same format. The difference is in the flags in the header, and the data that is filled in.

The request only has the question filled in.

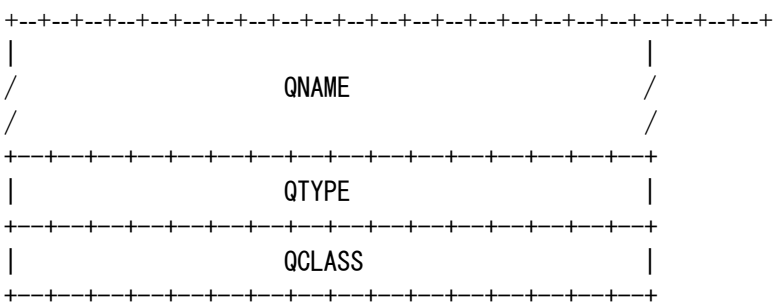
```
+---------------------+
| Header              |
+---------------------+
| Question            | the question for the name server
+---------------------+
| Answer              | RRs answering the question
+---------------------+
| Authority           | RRs pointing toward an authority
+---------------------+
| Additional          | RRs holding additional information
+---------------------+
```

The following schematic describes the DNS header. For the request, we set the QR bit to 0. The QDCOUNT short int is set 1. Rest of the *COUNT short ints are set to 0.

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    ID                         |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```



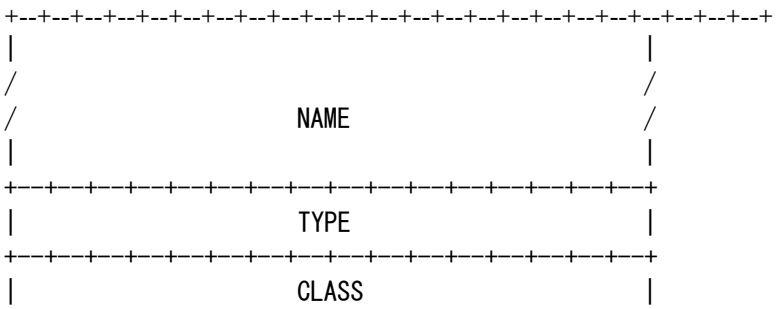
The following schematic describes the question part of the DNS request/response. In the DNS request, we put the URL in number separated form. For example: www.google.com becomes 3www5google3com0. Since, a packet is transmitted as bits, this makes it easier to read it on the other end. For our purposes, we set the QTYPE to TXT (16) and QCLASS to IN.

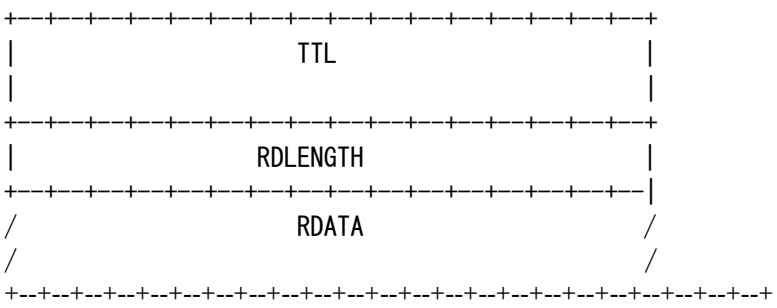


The following schematic describes a resource record. As a response from the server, and additionally for future use as a sort of CURL over DNS, we return a resource record of type TXT. Here, the NAME field contains pointer to the domain name in the “question” part of the response.

Note: we set TTL to 0 so that intermediate DNS cache servers don't cache our request responses. This has two fold benefits. First, a request is always serviced by the authoritative nameserver (unless the packet drops). Secondly, the intermediate DNS cache servers are not flooded with useless information that has to be cached.

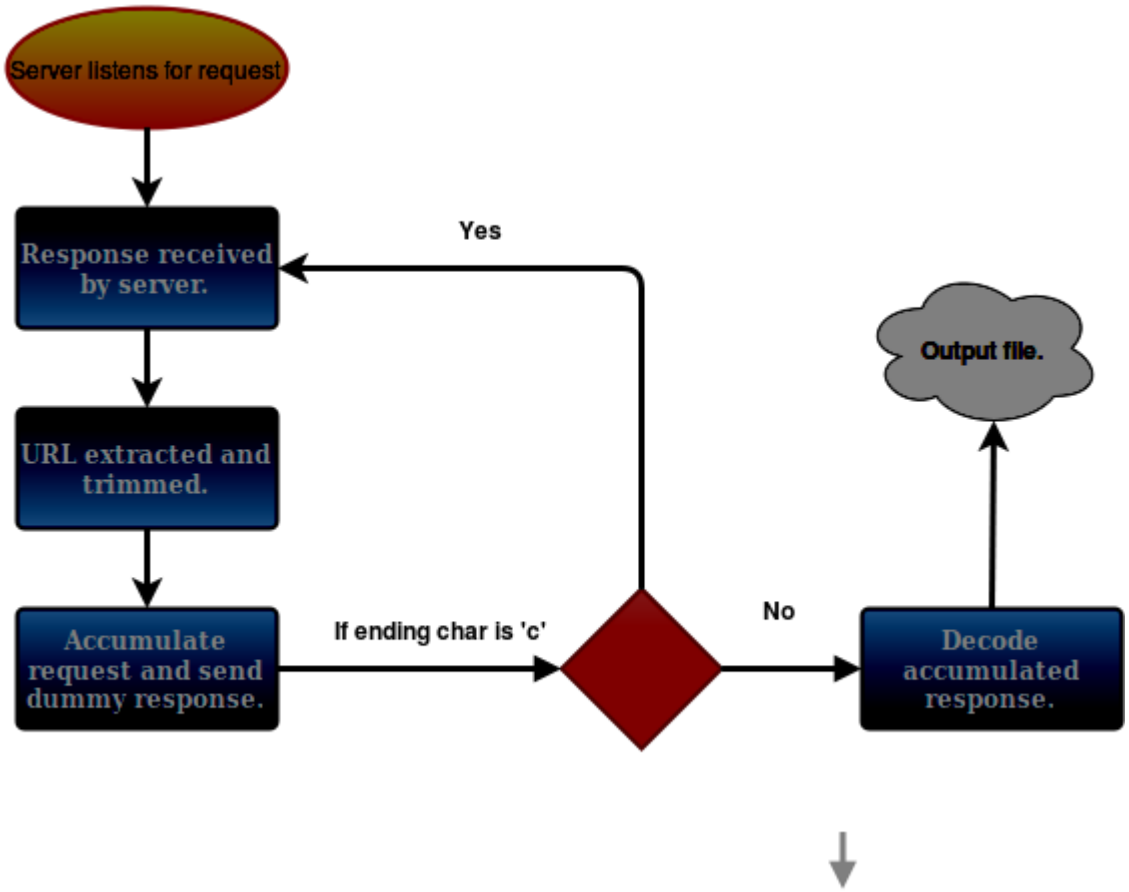
The TYPE short int is set to TXT (16).





Receiving and Assembling Data

The flow of the program at the server side is described by the following schematic:



When the DNS request reaches the server, the server checks if more requests are incoming. Depending on this, the server either continues to receive requests in continuation to the previous one, or resets or prints the accumulated string. Note that the server knows when to stop because the client wraps the string in indicator characters.

On receipt of a DNS request, the server trims the start and end characters. If then appends this string to another accumulator string. Once all the requests are received, the server decodes the received string.

For each request that the server receives, it sends back a dummy response. This is to have something akin to a stop and wait protocol.

Conclusion

The program and the report adequately highlight a method to tunnel internet traffic via DNS requests/responses. This program has been tested on **IITH-GUEST** network and the **Hyderabad Airport** network.

This can be used not only in places where networks are configured with *Captive Portals*, but also wherever a strict firewall is enforced. Most places may be checking for VPN connections but a lot of sysadmins do not restrict access to the local DNS cache server.

Note that this technique has the drawback that your data is unencrypted, and viewable as plain text.

This can be circumvented by using ssh over a DNS tunnel.

For stable package based on the same principle, check out iodine [5].

You can contribute to this program by forking the github branch and issuing a pull request. The package is called beaver [6].

References and footnotes:

[1]http://searchenterprisewan.techtarget.com/definition/tunneling
[2]http://www.dnsknowledge.com/whatis/authoritative-name-server/
[3]Resolution means finding the mapping.
[4]https://tools.ietf.org/html/rfc1034
[5]http://code.kryo.se/iodine/
[6]https://github.com/altairmn/beaver