HAL
open science

On Robust Covert Channels Inside DNS

Lucas Nussbaum, Pierre Neyron, Olivier Richard

To cite this version:

Lucas Nussbaum, Pierre Neyron, Olivier Richard. On Robust Covert Channels Inside DNS. 24th IFIP International Security Conference, May 2009, Pafos, Cyprus. inria-00425616

HAL Id: inria-00425616

https://inria.hal.science/inria-00425616

Submitted on 22 Oct 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

On Robust Covert Channels Inside DNS

Lucas Nussbaum, Pierre Neyron and Olivier Richard

Abstract Covert channels inside DNS allow evasion of networks which only provide a restricted access to the Internet. By encapsulating data inside DNS requests and replies exchanged with a server located outside the restricted network, several existing implementations provide either an IP over DNS tunnel, or a socket-like service (TCP over DNS). This paper contributes a detailed overview of the challenges faced by the design of such tunnels, and describes the existing implementations. Then, it introduces TUNS, our prototype of an IP over DNS tunnel, focused on simplicity and protocol compliance. Comparison of TUNS and the other implementations showed that this approach is successful: TUNS works on all the networks we tested, and provides reasonable performance despite its use of less efficient encapsulation techniques, especially when facing degraded network conditions.

1 Introduction

Nowadays, more and more networks only allow limited access to the Internet (intranets of companies, wireless networks in hotels, censored Internet access in some countries ...). As a result, many people have tried to leverage an unfiltered protocol to get a full access to the Internet, by establishing a communication channel to another system on the Internet. It has been shown that it is possible to hide data into IP and TCP headers [12], but also using protocols such as ICMP [11], HTTP and HTTPS [8, 13], or even IPv6 [9], for example.

Lucas Nussbaum
LIP, ENS Lyon, e-mail: lucas.nussbaum@ens-lyon.fr

Pierre Neyron
INRIA, e-mail: pierre.neyron@imag.fr

Olivier Richard
Laboratoire d'Informatique de Grenoble, e-mail: olivier.richard@imag.fr

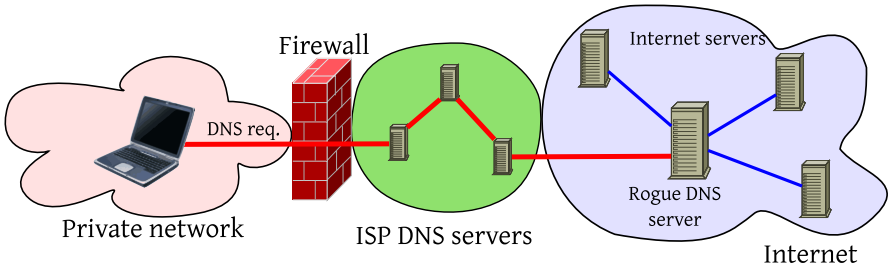

Fig. 1 General principle of covert channels inside DNS

In this article, we focus on covert channels using DNS. The DNS protocol is interesting for covert channels, because of its omnipresence: it is indeed difficult to provide an Internet access without providing access to a DNS service (one case is however possible, with configurations providing Web access only and where DNS resolution can then only be required on the HTTP proxy machine, but not on the end-clients). Furthermore, on networks where authentication or payment is required for users to get granted an access to Internet (usually using a *captive portal* [10]), DNS servers cannot return incorrect results until the user authenticates: if the DNS servers would return incorrect results to the users, the users' applications (web browser, for example) could cache the wrong result, and re-use it after authentication, preventing the user from connecting to some hosts even after authentication. As a result, networks that only allow Internet access after authentication on a captive portal, like those found in airports or hotels, allow full DNS access even before the user has logged in or paid the connection fee.

But encapsulating information in DNS packets raises a number of interesting challenges, since the DNS protocol entails a lot of restrictions. It restricts the size of packets and DNS records, leading the existing implementations of DNS tunnels to make compromises between protocol compliance and performance, making most of them easily detectable and then filtered. Those compromises are difficult to choose, because experiments need to happen on a lot of real networks to confirm that a given choice doesn't break the tunnel on some networks.

In the remainder of this article, we present an overview of IP over DNS tunnels (Section 2), and describe the existing implementations (Section 3). Then, we introduce TUNS, our implementation of an IP over DNS tunnel (Section 4), and evaluate it together with the other existing implementations (Section 5).

2 Overview

Figure 1 describes the general principle of covert channels inside DNS tunnels. The client is located in a network where communications to the outside world are going through a firewall. To communicate with servers on the Internet, the client encap-

sulates data in DNS queries related to the domain delegated to a rogue DNS server located on the Internet. Those queries are sent to the local network's DNS server (direct communication to other DNS servers is usually firewalled), then travel through the ISP's DNS infrastructure, and finally reach the rogue DNS server. Once there, the rogue DNS server decodes the data, and sends it to its target destination, as if it originated from the rogue server itself.

The return path works in a similar way: since the data was sent from the rogue DNS server, replies from target servers return back to the rogue server, which then encapsulates the data in DNS replies sent back to the initial client. However, since those DNS replies can only be sent in reply to DNS queries sent by the client, the client must keep polling the server for data.

Since DNS queries and replies travel through the ISP's DNS infrastructure, they must not differ too much from normal DNS packets. If they are not RFC-compliant or too easy to detect, they will be filtered. For example, data sent from the client to the server is usually encoded in the name being queried, using Base32 (5 bits of information per character) or Base64 (6 bits per character) encodings. But DNS only allows 63 different characters (`[a-z][A-Z][0-9]-`), forcing implementations that choose to use Base64 to add a non-compliant character to this set. Another problem is that Base64 is case-sensitive, while DNS allows servers to change the queries' case (RFC 1035 [5], section 2.3.3: *When data enters the domain system, its original case should be preserved whenever possible.*).

To deal with those constraints, the existing implementations make various compromises, which we describe in the following section.

3 Existing Implementations

There are several existing implementations of covert channels using DNS, which can be divided into two categories:

- covert channels that provide an IP over DNS tunnel (that allow transmitting IP packets through the communication channel) ;
- covert channels that provide a single TCP-like communication channel, allowing the establishment of an SSH connection (or any other kind of TCP connection) through it.

3.1 IP over DNS Tunnels

IP over DNS tunnels generally use a `tun` (level 3) or `tap` (level 2) device, allowing the user to route packets to that interface. Their use is transparent for applications.

NSTX [3] is the oldest of such implementations. To encode data into queries, it uses a non-compliant Base64 encoding (using "_" in addition to the 63 characters allowed by the DNS RFC). Replies are encoded into TXT records.

Iodine [2] is a more recent project. It uses either Base32 or a non-compliant Base64 encoding to encode the data (chosen via a configuration option). Replies are sent using NULL records. NULL records are described in RFC 1035 [5] section 3.3.10 as a container for any data, up to 65535 bytes long. It is used as a placeholder in some experimental DNS extensions.

Additionally, Iodine uses EDNS0 [6], a DNS extension that allows using DNS packets longer than the 512-byte limit initially chosen in RFC 1035.

Both NSTX and Iodine split IP packets into several DNS packets, send them separately, then reassemble the IP packets at the other endpoint (in a way similar to IP fragmentation).

3.2 TCP over DNS Tunnels

The second category of tunnels only provides a single TCP connection. The user generally establishes an SSH connection, then uses SSH's *port forwarding* and *SOCKS proxy* features.

The main drawback of those solutions is that they must provide a reliable communication channel over an unreliable protocol, and thus deal with losses, retransmissions, reordering and duplication of DNS packets.

OzymanDNS [4] is the most widely known implementation. It uses Base32 to encode queries, TXT records for replies, and the EDNS0 extension. During our tests, it proved unstable, crashing frequently.

dns2tcp [1] is a more recent implementation. It uses TXT records, and a non-compliant Base64 encoding (use of '/' in addition to the 63 characters allowed in DNS).

3.3 Conclusion

We tested those four implementations on a dozen of different DNS infrastructures (various french DSL providers, academic networks, public hotspots in hotels, airports, train stations, ...). The four implementations we tested failed to work on a majority of networks. This was expected, since:

- NSTX and dns2tcp can be blocked by forbidding non-compliant names in queries (both of them use DNS names with additional characters).
- All the implementations can be blocked by not serving queries for rarely used DNS records (TXT, NULL) or extensions (EDNS0). This is not an option on most networks, because it would break existing protocols (TXT records are used by the SPF anti-spam system, for example). But it is an acceptable option for commercial hotspots, where the user is less likely to expect a complete access to all Internet features.

4 TUNS

Since the four existing implementations failed to work on a majority of networks, we wrote our own prototype, named TUNS. We aimed at using only standard and widely used features of DNS, so TUNS' packets would be harder to filter in firewalls.

TUNS is an IP over DNS tunnel, written in Ruby, and available under the GNU GPL[1]. Contrary to other solutions, which use TXT or NULL records, rarely used for legitimate reasons, TUNS only uses CNAME records. It encodes the IP packets using a Base32 encoding (Figure 2). Unlike NSTX and Iodine, TUNS doesn't split IP packets into several smaller DNS packets: instead, the MTU of the tunnel's interface is reduced to a much smaller value (140 bytes by default), and the operating system is responsible for splitting IP packets using IP fragmentation. This removes the need to implement a state-machine to retransmit lost DNS packets, and increases the reliability of the tunnel in case of packet loss, but reduces the amount of useful information that can be transmitted, since the IP headers are repeated in each DNS packet.

When there is data to transmit on the client side, they are immediately encapsulated into a DNS query, and sent to the server. To receive data from the server, the client polls it on a regular basis with short DNS queries. If the server has data that must be sent to the client, it answers the client's query immediately. If it doesn't have data to send to the client, it waits for a small amount of time before sending an empty reply. If data to be transmitted arrives during this waiting delay, it is transmitted immediately. This allows reducing latency in interactive communications (such as SSH sessions for instance).

Another problem that is addressed in TUNS is the fact that DNS infrastructures sometimes send duplicated queries (probably, in a selfish way, to increase their chances to get a reply despite packet loss): a single query from a client can be duplicated by an intermediate server, and the final server will receive that query twice. In that case, the DNS server must return the same reply to both queries, otherwise an IP packet will be lost. A cache was added in TUNS to allow that.

We tested TUNS on a wide range of networks (including those where we previously tested the other implementations), and TUNS always worked properly.

5 Performance Evaluation

We evaluated the performance of NSTX 1.1 beta6, Iodine 0.4.1 and TUNS 0.9.2. To be able to compare them using a wide range of network conditions, we used network emulation (our experimental setup is described in figure 3). The experiments were performed on 3 nodes of the Grid'5000 platform (quad-Opteron 2.2 GHz, 4 GB of RAM, connected to Gigabit Ethernet). The nodes are running Linux 2.6.22 com-

[1] TUNS can be downloaded from `http://www-id.imag.fr/~nussbaum/tuns.php`

The client sends a data packet to the server:

```
Domain Name System (query)
  dIUAAAVAAABAAAQABJ5K4BKBVAHAKQNICBAAAOS5TD4ASKPSQIJEM7VABAAEASC.
  MRTGQ2TMNY0.domain.tld: type CNAME, class IN
```

The client sends a short query that the server will use to send a reply:

```
Domain Name System (query)
  r882.domain.tld: type CNAME, class IN
```

The server acknowledges the data that was sent. In its reply, it indicates the size of the server-side queue (l**4**.domain.tld, so 4 packets), so the client can send more requests for data:

```
Domain Name System (response)
 Queries
  dIUAAAVAAABAAAQABJ5K4BKBVAHAKQNICBAAAOS5TD4ASKPSQIJEM7VABAAEASC.
  MRTGQ2TMNY0.domain.tld: type CNAME, class IN
 Answers
  dIUA[..]0.domain.tld: type CNAME, class IN, cname l4.domain.tld
```

The server sends a reply containing data to the client:

```
Domain Name System (response)
 Queries
  r882.domain.tld: type CNAME, class IN
 Answers
  r882.domain.tld: type CNAME, class IN, cname dIUAAAVCWIUAAAQABH
  VCY2DMO2HQ7EAQSEIZEEUTCOKBJFIVSYLJOF4YDC.MRTGQ2TMNY0.domain.tld
```

Later, to another request for data, the server replies that it doesn't have any data to send:

```
Domain Name System (response)
 Queries
  r993.domain.tld: type CNAME, class IN
 Answers
  r993.domain.tld: type CNAME, class IN, cname dzero.domain.tld
```

Fig. 2 Content of DNS packets exchanged between a TUNS client and server, as seen with the `wireshark` network analyzer. Some packets have been shortened for space reasons.

piled for x86, not x86-64, to allow network emulation to benefit from the support for high-frequency timers, which didn't exist for x86-64 in Linux 2.6.22. Emulation settings are applied using the TC (Traffic Control) subsystem of Linux when the packets exit the emulator node, both when travelling from the client to the server, and on their way back from the server. Latency measurements were performed using `ping`, with one measurement per second. Bandwidth measurements were done using `iperf`. For all experiments, the network bandwidth was limited to 1 Mbps, to reproduce conditions found on slow wireless networks. This bandwidth limitation was confirmed to be realistic by measuring the available bandwidth on a few wireless networks.

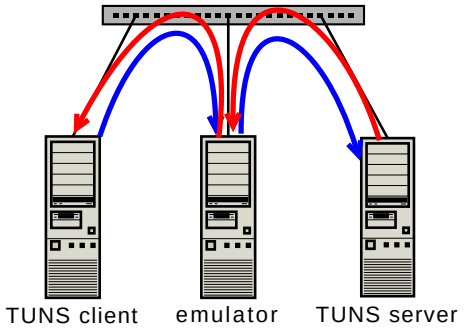

Fig. 3 Experimental setup.

5.1 Influence of Latency

Our first set of experiments focus on determining how the various solutions react to high-latency situations. Such situations are frequent with such tools, for example when the client connects to a server located overseas. Figure 4 shows the perceived latency, when the underlying network latency between the client and server ranges from 0 ms to 100 ms (so the maximum Round Trip Time is 200 ms). All solutions perform in similar ways in that case, but TUNS and Iodine exhibit a small, constant overhead compared to NSTX.

Figure 5 shows the measured upload bandwidth. Iodine gives the best results, while NSTX is slower than Iodine, especially when the latency is relatively low. TUNS is significantly slower than the other implementations. There are several reasons for this:

- TUNS uses Base32 encoding, while NSTX and Iodine use Base64, which is more efficient, but not RFC-compliant.
- NSTX and Iodine split IP packets, while TUNS relies on IP fragmentation. This causes the IP headers to be encoded in each packet, leaving less space for the rest of the data.
- NSTX and Iodine are written in C, while TUNS is written in Ruby. The use of an interpreted scripting language clearly increases the processing overhead. After discovering the performance problems of our implementation, we profiled TUNS using Ruby's profiler, and the Ruby DNS library we used proved to be a major bottleneck. The development version of Ruby (Ruby 1.9) improves performance slightly.

Figure 6 describes the tunnel latency using *pings* initiated from the server. It shows the efficiency of the polling method used by the tunnel. While NSTX and TUNS provide similar performance, Iodine's performance is much lower. Further investigations show that Iodine responds immediately to polling requests, even if it doesn't have anything to send. Instead, in that case, NSTX and TUNS wait for a while. If data to be sent to the client arrives during that waiting period, it can then

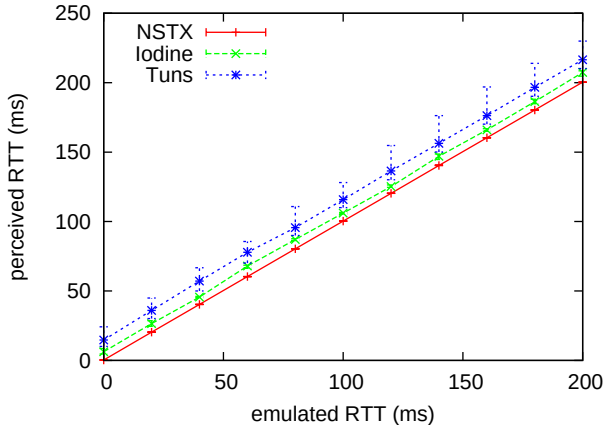

Fig. 4 Perceived latency using *pings* initiated on the client side. Vertical bars indicate the minimum and maximum values over 20 measurements.

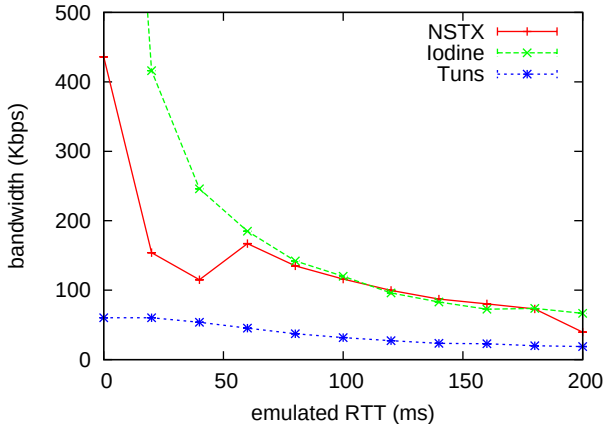

Fig. 5 Upload bandwidth (client to server).

be sent immediately. This optimization has a drawback: if the server takes too much time to reply, an intermediate DNS server (part of the ISP's infrastructure, for instance) might report a failure. In TUNS, the duration which the server is authorized to wait is configurable from the client's side, to adjust to different DNS infrastructures.

As seen in figure 7, the download bandwidth (server to client) is similar to the upload bandwidth (Figure 5). During all our bandwidth measurements, NSTX provided more variable performance than TUNS and Iodine.

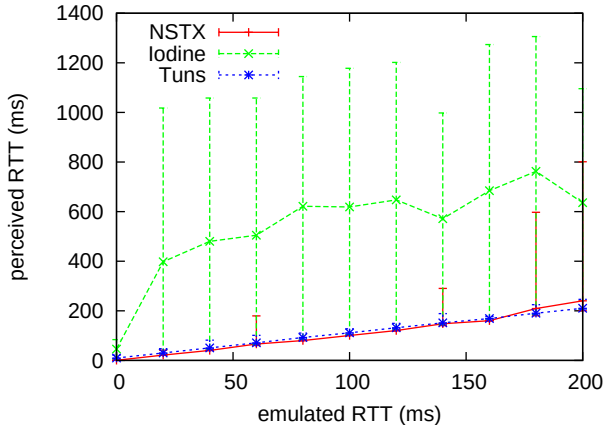

Fig. 6 Perceived latency using *pings* initiated on the server side.

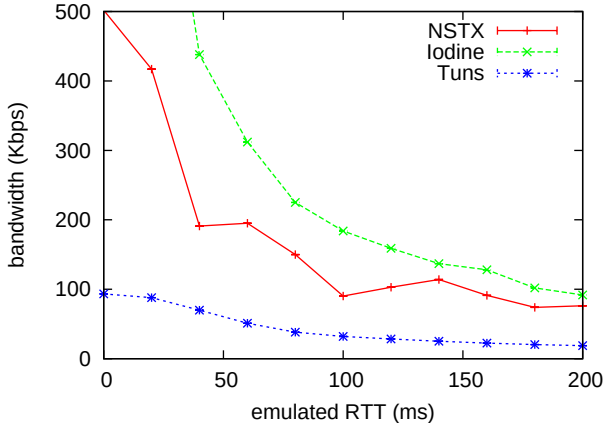

Fig. 7 Download bandwidth (server to client).

5.2 Influence of Degraded Network Conditions

After this first set of measurements, we focused on how the various implementations would perform in degraded network conditions. We emulated a network with 5% of packet loss (uniformly distributed), and latency varied of 10 ms around the value we defined (following normal distribution), causing packets reordering.

Results (Figure 8) show that, while the latency is mostly unaffected, the bandwidth is clearly penalized by such conditions. While TUNS was clearly the slowest implementation in *perfect* network conditions, it now outperforms NSTX.

This is very likely to be caused by the fact that Iodine and NSTX split the IP packets into several DNS packets: when a DNS packet is lost, Iodine and NSTX

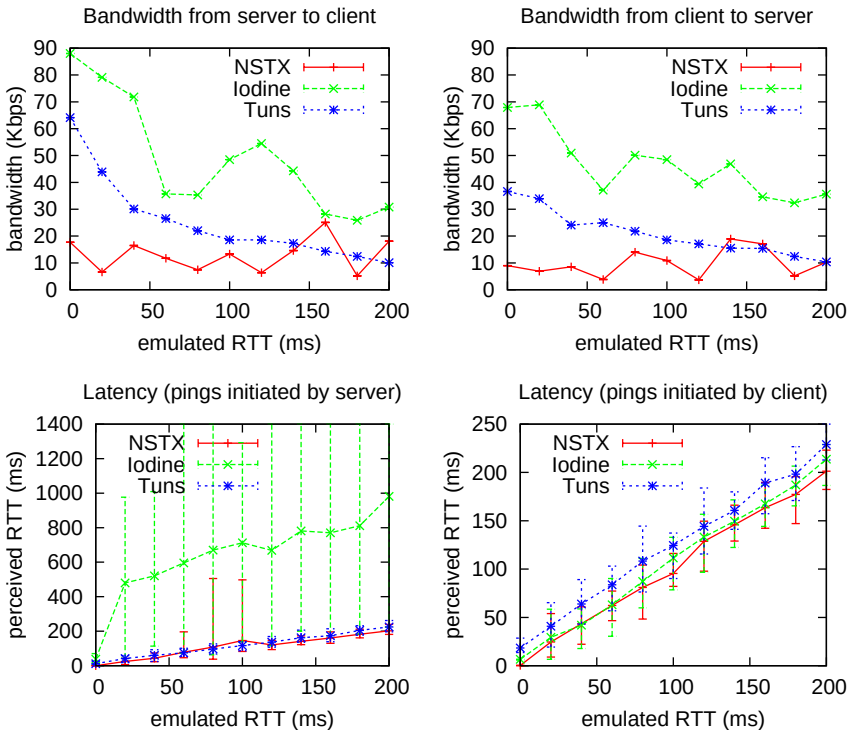

Fig. 8 Measurements on a network emulating packet loss and reordering.

must take care of retransmitting it, or must discard the other DNS packets split from the same IP packet, and wait for that IP packet to be retransmitted.

6 Changing Tunnel Parameters

While TUNS makes a compromise with efficiency to be able to function properly on more networks than the other implementations, there are cases where this compromise is not necessary. It is interesting to be able to adjust the tunnel's parameters, to match what the network will tolerate. However, this configuration change must be done remotely, from the client-side: the user might lock himself out while trying different parameters if that's not possible.

In TUNS, the client can send special DNS requests to change the configuration of the server. Currently, the following parameters can be changed:

- MTU of the tunnel interface: some DNS infrastructure allow sending larger DNS packets (more than 512 bytes). That allows increasing the length of the IP packets going through the tunnel, thus increasing the efficiency of the tunnel.
- Delay during which the server can keep a request before answering it: as explained in section 5.1, this affects the efficiency of the polling mechanism.

In the future, TUNS could be modified to allow other parameters to be changed as well:

- Encoding used for queries and replies (allow switching from Base32 to Base64 if the network allows it) ;
- Type of requests used (switch from CNAME to TXT or NULL) ;
- Allow enabling EDNS0, and DNS queries over TCP.

7 Future Work

In addition to the change of other tunnel parameters, several other improvements could be investigated for TUNS.

First, it would be interesting if TUNS could automatically adapt to a network: it could infer the optimal polling frequency for a path, and detect which countermeasures/filtering a network does to choose the optimal settings for encoding.

To increase the bandwidth of the tunnel, focus should be put on increasing the amount of useful data transferred in each DNS packet. Mechanisms like headers compression [7] could be use to decrease the overhead of using short IP packets. However, to be able to reduce the number of DNS packets, and not only the size of independent DNS packets, such mechanisms should be used before IP fragmentation happens, so it would have to be done in the kernel. A simple way to reach this goal could be to use PPP, instead of simply encapsulating IP packets.

Working on the encoding scheme could also bring interesting results. While the original DNS RFC doesn't allow the "_" character, this character is used in several DNS extensions (for example, it's commonly used in DNS SRV records). During our experiments, we encountered both networks that dropped queries containing the "_", and networks that allowed queries containing "_" for all records (not only SRV). Allowing Base64 using "_" as an option is a fairly conservative choice.

Another option is to add an escape mechanism to do Base64 using only the 63 valid characters. The problem with this is that it will cause the packet length to vary. In our experiments with changing the tunnel's MTU (which causes the DNS packets to increase or decrease size), we discovered that many DNS infrastructures were very strict on the packet length they allowed, which could cause some IP packets to never be transmitted through the tunnel, causing some connections (e.g TCP connections) to hang. A workaround could be to allow several escape mechanisms to co-exist and be chosen on a per-packet basis: the implementation could then choose the escape scheme that produces the shortest DNS packet, for each IP packet it tries to transmit.

8 Conclusion

The number of existing implementations proves it: the idea of using IP over DNS tunnels is not new. However, the number of existing implementations also shows that none of the implementations bring a definitive answer to this problem, due to the number of challenges that such tunnels need to overcome. This paper provides a detailed exploration of those challenges.

Specifically, TUNS proposes interesting solutions to address those challenges. It favors a simple design, and stays within the boundaries fixed by the DNS protocol's specification. This proved successful: TUNS is the only tunnel that worked on all the real networks we could try. TUNS also achieves reasonable performance compared to the other solutions, especially when facing bad network conditions, which are frequent with wireless hotspots.

Finally, TUNS demonstrates that it is possible to achieve reasonable performance without resorting to obscure DNS features or non-compliant behaviour. From a network administrator point of view, it seems difficult to block TUNS without also blocking legitimate traffic: the only solution left is to reduce the bandwidth of the covert channel by using traffic shaping techniques (to rate-limit the DNS queries), thus making the channel mostly useless.

References

1. Dns2tcp. `http://hsc.fr/ressources/outils/dns2tcp/`
2. Iodine. `http://code.kryo.se/iodine/`
3. Nstx. `http://thomer.com/howtos/nstx.html`
4. Ozymandns. `http://www.doxpara.com/`
5. RFC 1035: Domain names - implementation and specification
6. RFC 2671: Extension mechanisms for DNS (EDNS0)
7. RFC 3095: ROHC framework and four profiles: RTP, UDP, ESP, and uncompressed
8. Llamas, D., Allison, C., Miller, A.: Covert channels in internet protocols: A survey. In: 6th Annual Postgraduate Symposium about the Convergence of Telecommunications, Networking and Broadcasting (2005)
9. Lucena, N., Lewandowski, G., Chapin, S.: Covert channels in IPv6. Lecture Notes in Computer Science **3856** (2007)
10. Mejia-Nogales, J.L., Vidal-Beltran, S., Lopez-Bonilla, J.L.: Design and implementation of a secure access system to information resources for ieee802.11 wireless networks. In: CERMA '06: Proceedings of the Electronics, Robotics and Automotive Mechanics Conference (CERMA'06) (2006)
11. Ray, B., Mishra, S.: Secure and reliable covert channel. In: CSIIRW '08: Proceedings of the 4th annual workshop on Cyber security and informaiton intelligence research (2008)
12. Rowland, C.H.: Covert channels in the TCP/IP protocol suite. First Monday **2**(5) (1997)
13. Zander, S., Armitage, G., Branch, P.: Covert channels and countermeasures in computer network protocols. IEEE Communications Magazine **45**(12) (2007)