Project 4: IP over DNS

Due: 11:59 PM, Dec 12, 2017

Contents

1 Introduction

In this final project you will have a little fun providing full IP connectivity when all you have is the ability to send recursive DNS queries through a local DNS resolver. You are allowed, but not required, to work in pairs for this project.

You will learn a number of things in this project, including virtual interfaces, data encoding, tunneling, and network address translation. You will also learn to identify vulnerabilities in a network setup, such as a misconfigured DNS server. *Note that gaining access to network resources you are not authorized to access may be illegal. We do not encourage you to do this, nor will be responsible for any consequences if you do this.*

2 Components

The idea here is to trick a DNS server to send recursive queries on your behalf to a DNS server you control. You can encode any outgoing data you want in a prefix of the DNS query you send, and the DNS server can encode any data it wants in the DNS response. One place in which you can encode your data is in the TXT record, but you may also be able to get away with using a NULL resource record.

Of course, the devil is in the details, and there are several pieces to get this working. Let's call the machine behind the restricted connection the client, and the DNS server you control the server. The server is going to be your gateway to the Internet.

The main piece of the solution is a *tunnel* between the client and the server. For both of them, the tunnel will look like a point-to-point link that sends and receives IP packets. It just so happens that the bits are conveyed inside of DNS packets.

There are two main parts of the solution: the first is to establish the tunnel, so that the packets can flow between the client and the server. The second part is to use the tunnel to provide Internet access to the client through the tunnel.

2.1 Creating the tunnel

In this part you create two interfaces at either end of the tunnel and get IP packets to go between the client and the server.

You need to have a way to get IP packets leaving the client to be sent through the tunnel, rather than directly through the network interface in the client. Rather than writing a kernel module to do this, there is a better way: to create a *virtual interface*. A virtual interface, also called a TUN/TAP device, looks to the IP stack just like a physical interface (e.g., `eth0`), but the packets sent to this interface are delivered *to a user-space program*. Likewise, when this user-space program writes packets to the virtual interface, they are delivered to the IP stack in the kernel, as if they had come from the network.

The difference between a TUN and a TAP interface is that the TUN interface works at the IP layer: it sees raw IP packets, while the TAP interface sees raw Ethernet frames. Since you are tunneling IP packets, you can use a TUN interface for this.

You will create TUN interfaces both at the client and at the server, and you should give them private addresses (from RFC 1918). For example, you could give the client's interface the address 10.10.10.1 and the server's interface the address 10.10.10.2. You should then configure routes at the client and at the server so that IP packets sent to the address of the other side go through the corresponding TUN interfaces. This is the minimum you have to do to establish the communication, and you should be able to ping one side from the other, and vice-versa. The `ip(8)` command is your friend here, for setting up the routing tables. Look at the Resources section below for pointers on how to create the TUN virtual interfaces.

Once you have the interfaces and the basic routing working, you need to write the two main pieces of the solution: the programs which sit at either end of the tunnel.

At the client, you'll write a program that will read packets written to the TUN interface and send them as DNS queries to the local resolver (which your machine probably got from DHCP). The client opens two file descriptors: one to the TUN device, and one binding to UDP port 53 at the local DNS resolver. This program will also receive IP packets encapsulated in DNS responses from the server, decode them, and write them back to the TUN device so the kernel receives them.

How do your DNS packets reach your server? Your DNS server will be running on a virtual machine, which we have configured as the name server for a subdomain of `cs168.fun`, such as `a.cs168.fun`. Any query that you send to a subdomain of `a.cs168.fun` will be forwarded to your DNS server. We will provide the actual names and addresses of your virtual machine and domain by email.

The DNS server running on your virtual machine will be the other end of the tunnel. It has to decapsulate the IP packets encoded in the DNS queries, and inject them into the kernel, as if they had come *from* the IP address of the other end of the tunnel and *to* the IP address of the server's TUN interface. You should also configure the routes at the server to make sure you can send packets

to the client's TUN interface's address.

Again, to test this part, you should be able to ping one side from the other. If the addresses you used are the addresses above (which they don't have to be), you should be able to do:

`client> ping 10.10.10.2`

and

`server> ping 10.10.10.1`,

and have them both work.

Note that it is tricky to get the server → client version working, as only the client can initiate communication by sending a DNS request.

2.2 Using the tunnel for Internet access

Now that you have an IP link between the client and the server, you have to arrange it so that the client can access the Internet using the server's connection.

There are different ways of doing this, with different tradeoffs and capabilities. You have to do at least one of them. Your solution must work for HTTP and for another application protocol of your choosing.

Basic solution: One suggestion is to use SSH as a SOCKS 5 Proxy (RFC1928), which allows you to forward any TCP or UDP protocol via the proxy. This solution is simple, but not entirely transparent: you need to tell your application to send its traffic to the proxy, rather than to the final destination. See the man page for SSH on how to set it up as a SOCKS proxy.

General solution: A more general and transparent solution is to have the server forward any IP packet it receives from the tunnel to the Internet, and to relay the response packets back into the tunnel.

One detail is that you would have to rewrite the SRC IP address of the packets to be the public IP address of the server, and send the packet to the destination using a RAW IP socket. You would then have to find a way of determining which IP packets coming into the public IP address of the server to forward back to the tunnel.

You might recognize this rewriting as the job of a Network Address Translation gateway. Fortunately, you don't need to do this, as Linux already has a full implementation of NAT, which you can configure with the `iptables` command.

2.3 Summary

The main pieces of the solution, then, are:

a. Setting up a TUN virtual interface at the client

b. Configuring ip routes at the client to use the virtual interface

c. Writing the client end of the tunnel

d. Writing the server end of the tunnel

e. Setting up a TUN virtual interface at the server

f. Setting up a forwarding mechanism, either through a proxy or via a NAT

3 What to hand in

You must hand in all of the source code that you use for the client and the server portions. You must also create two scripts, one to be run at the client, and another to be run at the server, that does all of the configuration and starts all of the programs necessary for your tunnel to work. Optionally, you should also provide scripts to shut down the programs, and revert the configurations to what they were before. The configurations will involve creating the TUN interfaces, setting up routing tables, and proxies, if necessary.

Your hand-in must have a README file describing the steps to compile and run your client and server, and instructions on how to configure application(s) at the client to use the tunnel (if you use the proxy solution). For example, you might require that a Web browser be configured with a SOCKS proxy running on localhost:5000.

The README file MUST also briefly describe the main points of your solution, including, but not necessarily limited to: how you configure the routes, how you encode data, and any details of the protocol between the client and the server.

Your programs must be parameterizable, so that they will work if your DNS server is configured on another domain.

4 Challenges and Design Decisions

While we have given you a high-level description of what to do, there are many details that you have to figure out.

Language You are free to use any programming language to implement the different components of the system, provided that it works end to end.

Third party code You are allowed to use third party libraries to create and set up tunnels, and to create custom DNS packets. You may use third-party or library code to encode and decode the data into the DNS packets. You MUST however implement the client and the server ends of the tunnel.

Limits RFC 1035 describes the DNS packet format, and imposes a number of limits that are relevant to you:

- A UDP DNS packet cannot exceed 512 bytes

- A DNS name in a query cannot exceed 255 bytes
- Each label in a DNS name cannot exceed 63 bytes

Check the RFC for the format and limits of the TXT and NULL resource records.

Encoding You have to decide how to encode the IP packets into DNS queries and responses. One possibility is to use the URL-safe version of base64 encoding, which can encode 3 bytes into 4 characters from the set [A..Za..z0..1-_]. This encoding is described in RFC 4648. You don't have to use it, as your client only has to talk to your server. If you wanted, you could even compress the data before sending it, to increase the bandwidth.

Asymmetry The communication that you can implement is asymmetric: only the client can initiate communication with the server, by sending a DNS query. It is up to you to provide a way for the communication to seem as close as possible to being bidirectional.

Fragmentation You can implement some mechanism to deal with IP packets larger than your DNS packets can carry, or you can avoid fragmentation by appropriately setting the MTU on the links you create.

Architecture By now you should be comfortable writing network servers that process and forward packets. For example, will you usea a thread- or event-based design?

5 Resources

This project will require some network layer setup that is not permitted on department-managed machines. You should configure to run your DNS server on the provided AWS instance, and run your client on any Linux machine (your laptop) which you have sudo privilege. There are many text resources online that will help you in this project. Note that you must write the client and the server portions of this project yourself (see **Third party code** above).

Here are some pointers:

- Look at the man page for `ip(8)`. You will use it to create and configure your routes. In newer versions of Linux you can even use it to create your virtual interfaces.
- Libnet (`http://packetfactory.openwall.net/projects/libnet/`) and firedns (libfiredns-dev on Debian/Ubuntu) are two libraries that allow you to construct and parse DNS packets.
- Vtun (`http://vtun.sourceforge.net/tun/index.html`) provides a TUN/TAP driver for Linux, and has some documentation. There are TUN/TAP drivers for other operating systems as well.
- This page (`http://backreference.org/2010/03/26/tuntap-interface-tutorial/`) has a very good tutorial on TUN/TAP.

Lastly, the following RFCs are very relevant:

- RFC 1918 - Address Allocation for Private Internets
- RFC 1928 - SOCKS Protocol Version 5
- RFC 1035 - Domain names – implementation and specification
- RFC 4648 - The Base16, Base32, and Base64 Data Encodings

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS168 document by filling out the anonymous feedback form:

`https://piazza.com/brown/fall2017/cs168`.