Before jumping into the Network Extension framework, you need to know a bit of theory about VPN and networking in general. If you are already familiar with networking and VPN, feel free to skip this article. If you are not, this article has a quick overview/refresher on the network protocols, it outlines the problem that VPN is aimed to solve, and then analyzes one of the existing VPN protocols.

Prerequisites

There are no prerequisites for going through this article, but the general knowledge of the networking protocols is going to be a plus. This article makes heavy use of Wireshark. If you are not familiar with this tool, I would suggest installing it (it is free), and inspecting one or two sample captures.

Networking

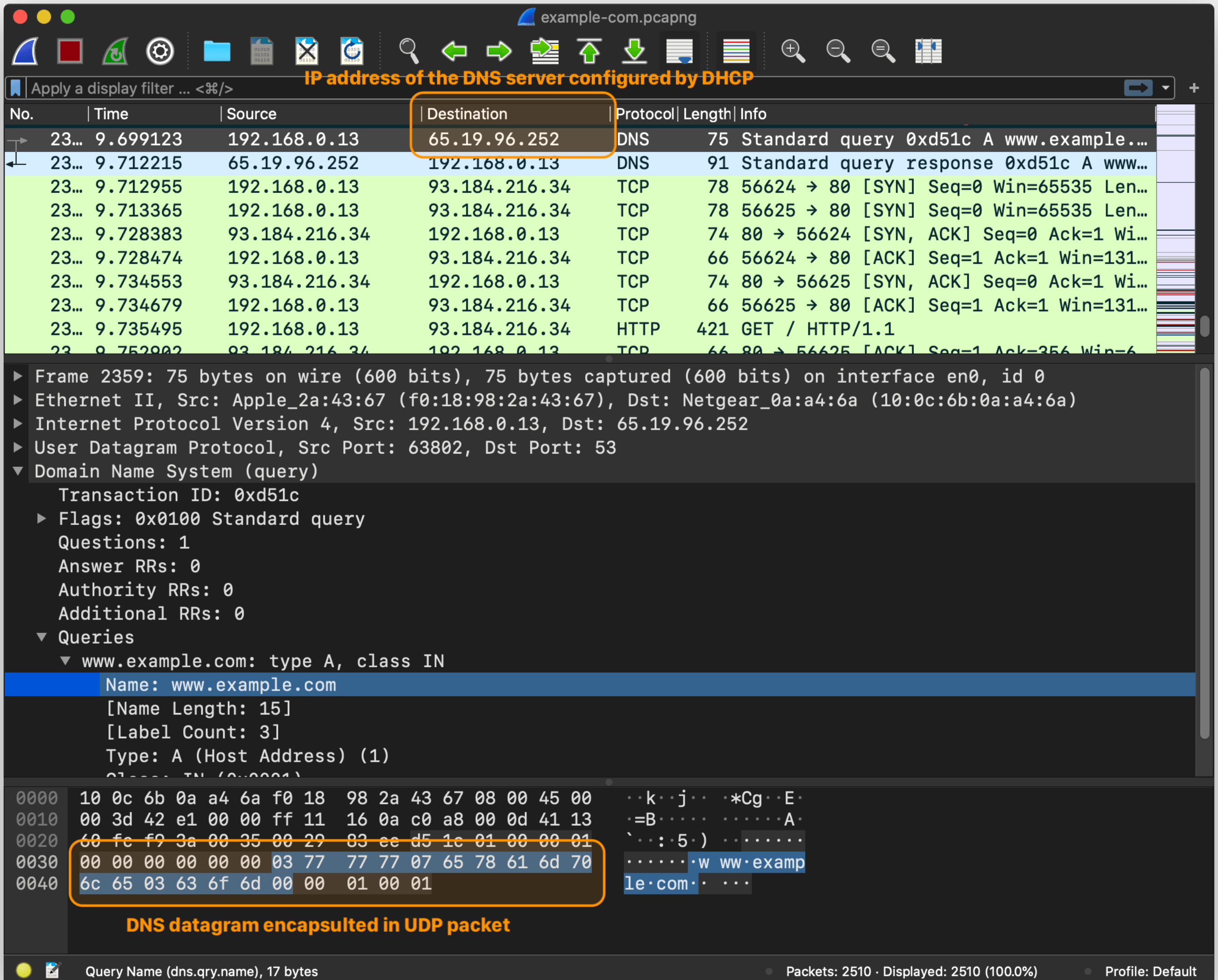
What happens when you open your browser and type in a URL? Let's open Wireshark and find out.

DNS, HTTP (Application Layer)

First, the browser needs to send a DNS request to resolve an IP address for the given domain name, for example `www.example.com`.

DNS Request | DNS Response

The browser sends a DNS query with a domain name `www.example.com`.



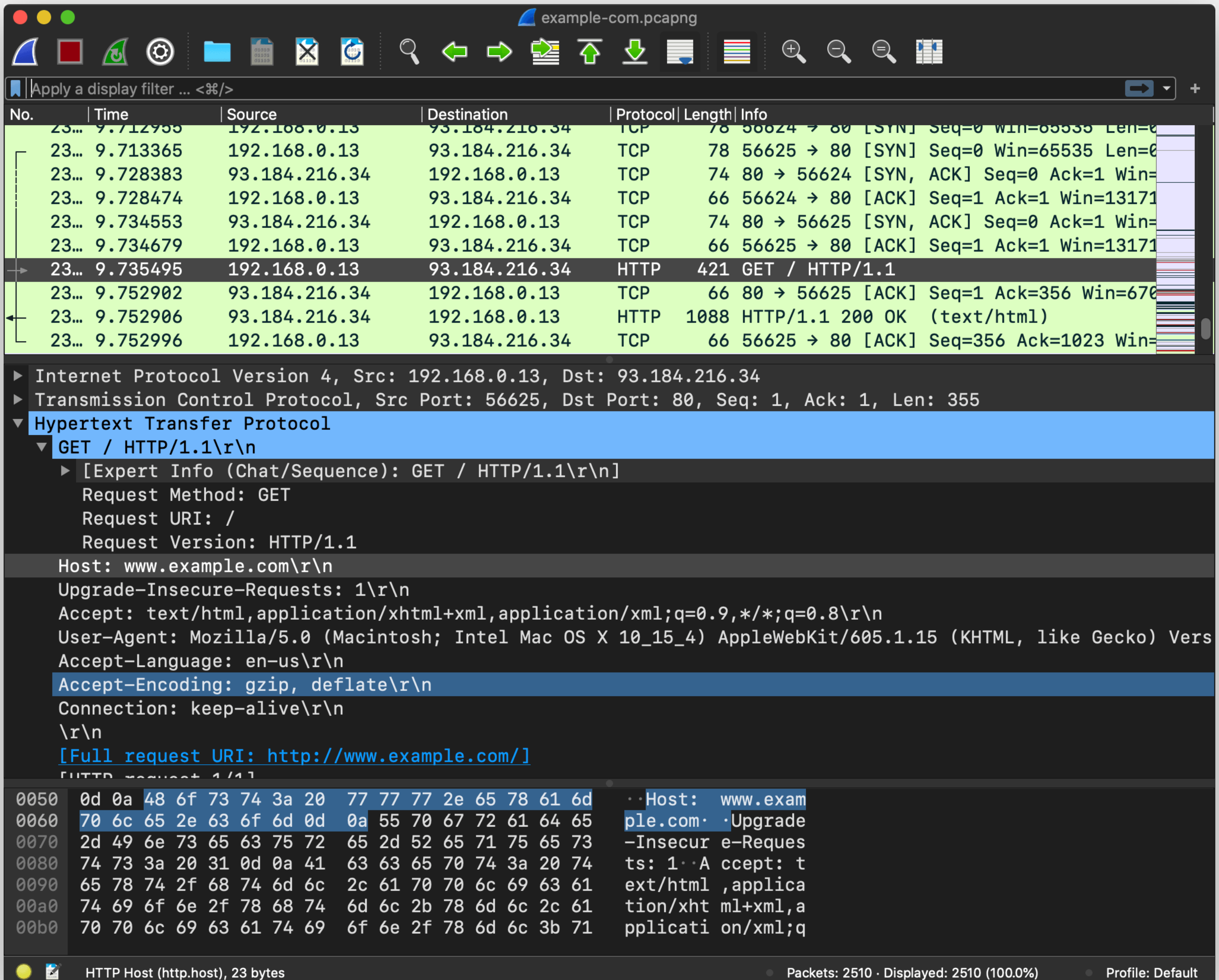
The Domain Name System (DNS) is the phonebook of the Internet. Humans access information through domain names, such as `apple.com` or `nshipster.com`. Browsers interact through IP addresses. DNS translates domain names to IP addresses[1].

> A browser has to know an IP address to send requests to `www.example.com`. How was it able to send the requests to the DNS server? Typically, an IP address (or addresses) of the DNS server is provided to the computer by DHCP when connecting to the router.

Now that the browser has an IP address, it is ready to start sending HTTP requests to `www.example.com (93.184.216.34)`.

HTTP Request | **HTTP Response**

The browser takes the resolved IP address and initiates an HTTP request to fetch the content of the page at the root path (`/`).



Let's now spend too much time talking about the application layer protocols. I'm assuming you are already familiar with HTTP and DNS. The reason I used these two examples is that HTTP typically works on top of TCP[2], and DNS on top of UDP[3].

If you want to learn more, MDN has a great section on HTTP, and Cloudflare – on DNS. If you want to dig a bit deeper, I would recommend reading RFCs. You can find the list of HTTP and DNS RFCs on the respective wiki pages.

UDP, TCP (Transport Layer)

HTTP and DNS solve specific application-level problems. But when a computer receives a message, how does it know which application to deliver it to? This is where *transport layer* protocols, such as User Datagram Protocol (UDP) and Transmission Control Protocol (TCP), come in.

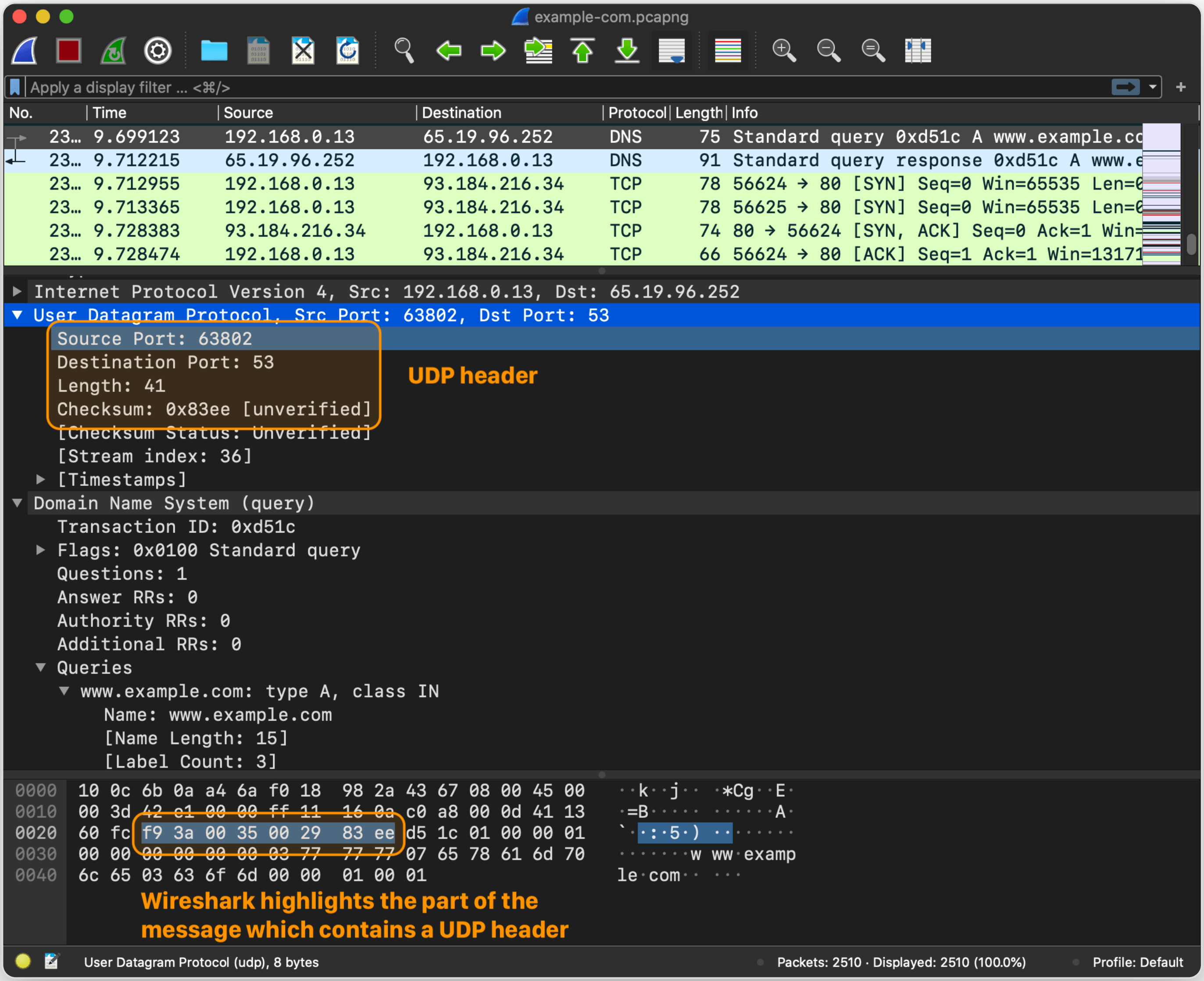
When a server is started, it typically creates a socket, an internal endpoint addressable using a combination of an IP address and a port number (a 16-bit unsigned integer ranging from 0 to 65535), and starts *listening* for messages arriving on this socket.

When a browser needs to send a message, it also needs a socket and creates one using one of the system APIs.

On Apple platforms, there is a variety of APIs for working with ports/sockets, ranging from the Unix sockets, `CFSocket` wrapper and to recently introduced `Network` framework.

Transport layer protocols, such as TCP and UDP, transfer data using protocol data units. For TCP, the PDU is a segment, and a datagram for UDP. Both protocols use a header field for recording the source and destination port number.

The DNS messages are typically sent using UDP (other options exist, even including things like DNS over HTTPS). The UDP datagram header is extremely. simple When a system needs to send a DNS message, it takes the message (the protocol of the message is irrelevant, for DNS it's just raw bytes), and prepends 4 bytes of the UDP datagram header. If we go back to our browser example, here is what the UDP header looks like:

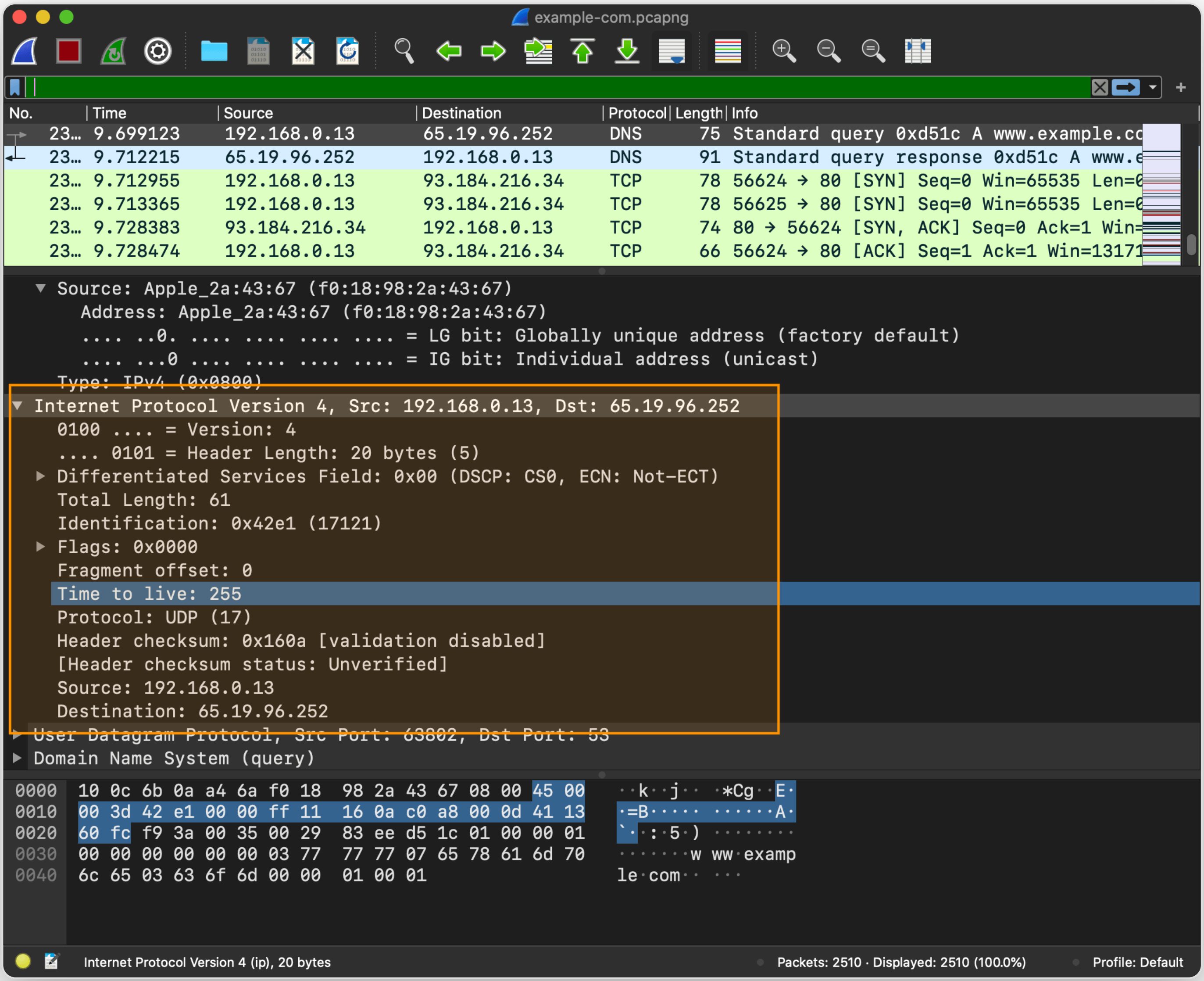


The *source* port is `63802`, which is the port that the browser reserved for itself, and the *destination* port is `53`, which is one of the well-known ports.

Obviously, UDP and TCP are completely different. TCP protocol has its own segment structure. TCP is connection-oriented, it provides reliable, ordered, and error-checked delivery of a stream of octets. UDP, on the other hand, uses a simple connectionless communication model, it's very minimalistic.

IP (Internet Layer)

In the previous section, we established that a socket is addressable using a combination of an IP address and a port number. We know what the port is for. But there are no IP addresses to be found in UDP/TCP headers. To know which computer to send the messages to, you need an IP address and IP protocol. Let's get back to our Wireshark capture to see what information an IP header contains.



All of the information that you saw captured using Wireshark was recorded from my `en0` *network interface* (WiFi).

> **Network interface** is a point of connection between the computer and the network. A computer can have multiple network interfaces: Wi-Fi, Ethernet, etc.
>
> There are *physical* network interfaces, such as `en0` (which is your WiFi module on macOS). You can list all of those using `networksetup -listallhardwareports` command. Each physical interface has an associated Ethernet Address and a hardware device, for example:

```
> networksetup -listallhardwareports

Hardware Port: Wi-Fi
Device: en0
Ethernet Address: f0:18:98:2a:43:67
...
```

And then there are *virtual* network interfaces, such as `lo0` (a loopback interface which computer uses to communicate with itself) and `utun` (which are typically used by VPN, more on this later). You can use `ifconfig` to see all of the network interfaces, including both physical and virtual.

```
> ifconfig

lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 16384
options=1203<RXCSUM,TXCSUM,TXSTATUS,SW_TIMESTAMP
inet 127.0.0.1 netmask 0xff000000
...
```

All of the information that you saw from the captures in this section is also visible to my ISP and to any servers between me and the destination. They saw all my DNS queries, they saw my IP address, and in case of HTTP (unencrypted), they saw all my HTTP communications too[4]. Using this information anyone can identify and track me. And these are the kinds of problems that VPN is designed to solve.

VPN

What is VPN (Virtual Private Network)? Originally, VPN was designed to establish secure connections between a computer and a protected or private corporate network, using an otherwise insecure public network. This typically involved "making" a computer part of the private network so that the computers within the network could address it.

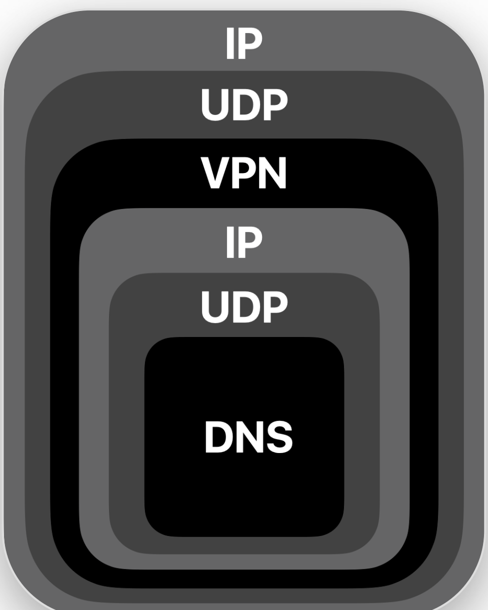
In recent years, VPNs also blown up in popularity in the consumer market as a way to stay secure and anonymous when going online, and also enjoying unrestricted network access by hiding the source of the traffic. Business and personal VPN are typically very different products. I'm going to focus primarily on personal VPN in this series, as it involves slightly less complexity to deal with.

How Does It Works?

The key idea behind VPN is surprisingly simple:

> - Take all of the IP packets[5] coming from the system
> - Encrypt them
> - Encapsulated encrypted data blobs in the VPN packets
> - Send the VPN packets to the VPN server (over UDP/TCP)

The IP packets end up being wrapped in another IP packets, a networking version of a Russian nesting doll.



This is a simple idea, yet it achieves all of our goals. The original IP packets are fully encrypted. You can see neither the application, nor even the internet layer contents (IP addresses). The only thing ISP or other observers are going to see is some blob of encrypted data going between a VPN client and a VPN server.

Now, how do you implement something like this? This is what I'm going to cover in the upcoming articles. It is going to involve direct IP packets manipulations, encryption, virtual network interfaces, and more. Exciting stuff!

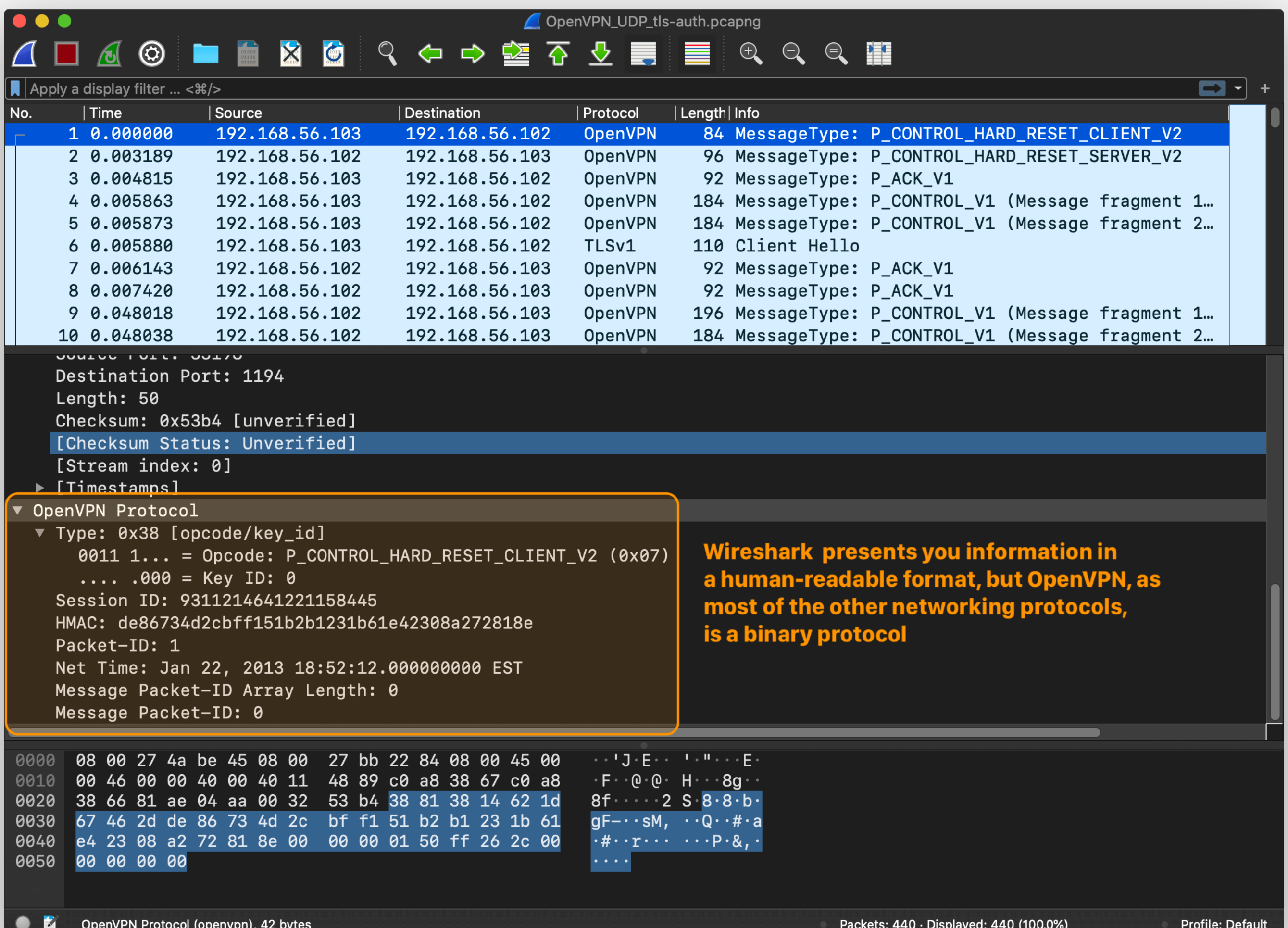
Example: OpenVPN

The idea behind VPN is simple, yet designing and implementing a protocol is far from easy. Before designing a VPN protocol of our own, it is worth exploring an existing VPN protocol, such as OpenVPN.

The description of the protocol is surprisingly short and simple. As a first step, the client **establishes the session** with the server and exchanges cryptographic information.

Reset Client **Reset Server** **Acknowledge**

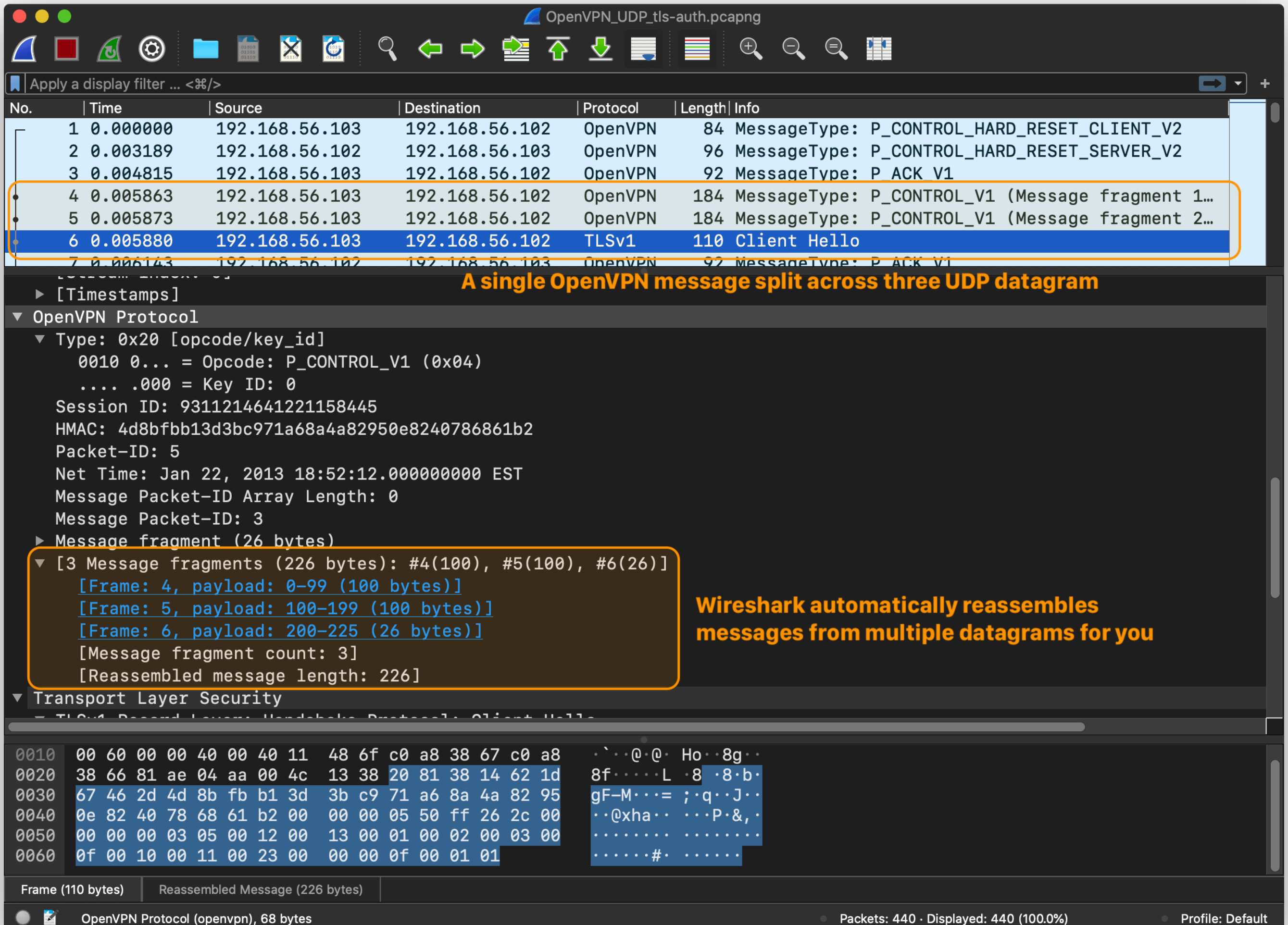
The client sends a *control* (`P_CONTROL`) packet with `P_CONTROL_HARD_RESET_CLIENT_V2` type which prepares a new session.



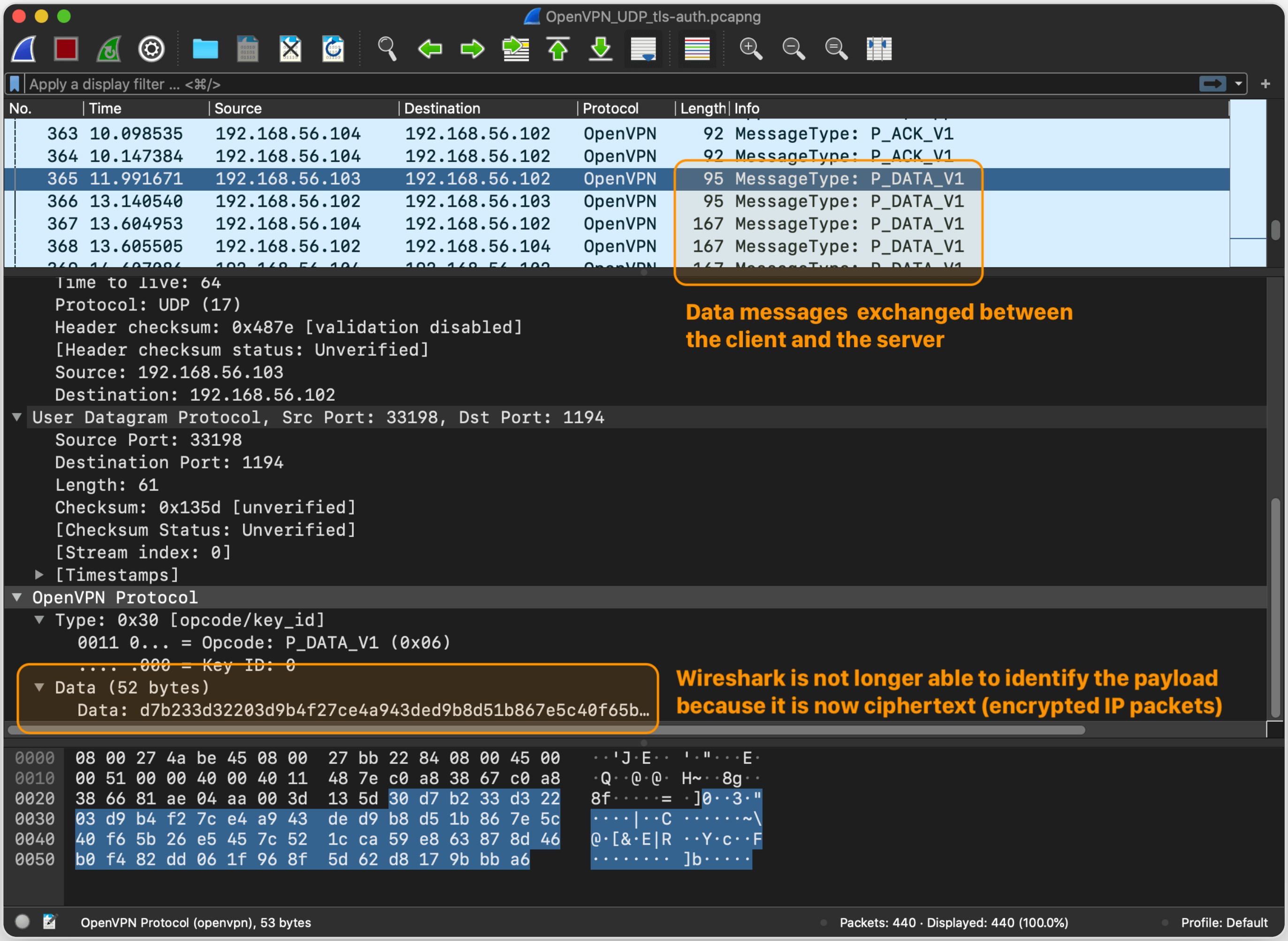
When the session is ready, the client and the server need to exchange cryptographic data to agree on a key. The exchange, called the "handshake" in TLS[6].

Control Message | **TLS Payload**

The client sends another control (`P_CONTROL`) packet, this time with a payload.



I'm not going to go into detail regarding the TLS exchange. The Wireshark session only shows the first messages of the TLS "handshake". The entire handshake requires a few round-trips. The point is that after a few control messages, the client and the server are ready to encrypt and send messages to the server, regardless of which version of TLS and which encryption model was used. And this is what `P_DATA_V1` message type is for – data channel packet containing actual tunnel data ciphertext.



And this wraps up an overview of OpenVPN. There is no need to fully understand it, but the big picture is clear. The main takeaways are:

- OpenVPN is a binary protocol that has multiple message types: control (`P_CONTROL_V1`), acknowledgment (`P_ACK_V1`), and data (`P_DATA_V1`).
- Before a client and a server can start exchanging data (`P_DATA_V1`), they need to establish a session and exchange cryptographic information.
- The messages are sent over UDP[7]. The control messages (`P_CONTROL_V1`) have a delivery guarantee and require acknowledgment (`P_ACK_V1`). The data messages (`P_DATA_V1`) don't require an acknowledgment and don't have a delivery guarantee, the underlying protocol of the original messages is responsible for providing the needed guarantees.

This is enough information to start working on our toy VPN protocol.

Continue Reading »

References

1. Wireshark, **OpenVPN Sample Captures**
2. Cloudflare, **What is DNS?**
3. Mozilla, **HTTP**
4. OpenVPN, **OpenVPN Protocol**

---

1. You could memorize an IPv4 address, but try memorizing `2400:cb00:2048:1::c629:d7a2` (IPv6). ↩
2. HTTP/3 seems to be replacing TCP with QUIC, a new transport layer protocol based on UDP ↩
3. That's not always the case and it depends on your browser. For example, Firefox now uses DNS over HTTPS by default. ↩
4. If you are using HTTPS, you are generally good. Only the destination server is going to be able to read HTTP headers and body of your requests. ↩
5. The packets that are configured to go through the VPN tunnel. A *split* tunnel can be configured, such that only packets with the predefined destination addressed are routed via the tunnel, and the rest go through the normal physical network interfaces. ↩
6. OpenVPN supports two modes: static key (using a pre-shared key) and TLS (using SSL/TLS and certificates for authentication and key exchange). In this example, you see the later. ↩
7. OpenVPN uses UDP by default, but also supports TCP. ↩