Intercepting HTTP/2 Packets without Wireshark

abram.id/posts/intercepting-http-2-packets-without-wireshark

March 12, 2020

HTTP/2, what is it?

I never knew HTTP/2 until I bumped into gRPC, which uses HTTP/2 as its Layer 7 protocol. At first, it seems that HTTP/2 is just like another update of the commonly used HTTP/1.1. Yet HTTP/2 is a new HTTP protocol designed to deal with the limitations of HTTP/1.1.

For a bit of context, HTTP/1.1 had plenty of latency and inefficiency issues that made the performance of common internet webpage (that relies heavily on HTTP protocols) extremely hard to optimize. The first time we open a web page, it usually requires requesting a dozen resources from stylesheets, images, JavaScript codes, and other API calls. HTTP/1.1 does this by creating an equal number of TCP connection to get the resources in a parallel fashion. This means when the server is processing and preparing the response, the TCP connection is doing nothing but waiting for the server to give the response. This is very inefficient considering every single TCP connection made is doing nothing for some time. Plus there is always a cost when opening a TCP connection and closing it. So it is very inefficient to use one TCP connection per HTTP request.

HTTP/2 was made to solve some of the problems by enabling TCP to be multiplexed for multiple HTTP requests. So with HTTP/2, we will be opening less number of TCP connections compared to HTTP/1.1. HTTP/2 also enables a TCP connection to be reused for multiple request, no more one TCP connection per HTTP request. These two features will improve the utilization of the TCP connection.

Another main difference of HTTP/2 and HTTP/1.1 is that HTTP/2 is binary, while HTTP/1.1 is textual. On one hand, this gives us the benefit of speed since computers are good with binaries. Yet on the other hand, it is more difficult to debug since humans are not very good with binaries. To add on, what’s more interesting is even the HTTP/2 headers are compressed for performance reasons. These two features increase the complexity to intercept and process HTTP/2 packets from the previous HTTP/1.1 where we could just read the whole payload text.

Aside from features mentioned above, there are plenty of other features of HTTP/2 you can read in the RFC 7540 document.

Intercepting the actual packets

At this time of writing, I haven’t found any way to intercept and decode HTTP/2 packet other than Wireshark. Wireshark is obviously a great tool for network analysis, but at other times, we want to intercept and process the packet built in right onto our applications. In this use case, Wireshark is not a suitable option, so we need to integrate HTTP/2 into existing packet interception library.

To intercept the packets, I will be using Go with Google's gopacket. This stack is my go to choice because Go have the first class support for HTTP and HTTP2 and Gopacket itself is fairly extensible.

From here onwards we'll use the term "frame" to represent the unit of transfer of an HTTP/2 traffic.

Implementing the layers

Since Gopacket doesn't support HTTP/2 as its application layer, we need to tell Gopacket about the structure of HTTP/2 frame using the code below.

```
// Create a layer type and give it a name and a decoder to use.
var LayerTypeHTTP2 = gopacket.RegisterLayerType(12345, gopacket.LayerTypeMetadata{Name: "HTTP2",
Decoder: gopacket.DecodeFunc(decodeHTTP2)})

type HTTP2 struct {
    layers.BaseLayer

    frames []http2.Frame
}

// Implement layer's metadata
func (h HTTP2) LayerType() gopacket.LayerType      { return LayerTypeHTTP2 }
func (h *HTTP2) Payload() []byte                   { return nil }
func (h *HTTP2) CanDecode() gopacket.LayerClass    { return LayerTypeHTTP2 }
func (h *HTTP2) NextLayerType() gopacket.LayerType { return gopacket.LayerTypeZero }

// Implement the decoder function to be used
func decodeHTTP2(data []byte, p gopacket.PacketBuilder) error {
    h := &HTTP2{}
    err := h.DecodeFromBytes(data, p)
    if err != nil {
        return err
    }
    p.AddLayer(h)
    p.SetApplicationLayer(h)
    return nil
}

func (h *HTTP2) Frames() []http2.Frame {
    return h.frames
}

func (h *HTTP2) DecodeFromBytes(data []byte, df gopacket.DecodeFeedback) error {
    var frames []http2.Frame
    frameHeaderLength := uint32(9)
    payloadLength := len(data)

    payloadIdx := 0
    for payloadIdx < payloadLength {
        if payloadIdx+int(frameHeaderLength) > payloadLength {
            return fmt.Errorf("Payload length couldn't contain Frame Headers")
        }

        framePayloadLength := (uint32(data[payloadIdx+0])<<16 | uint32(data[payloadIdx+1])<<8 |
uint32(data[payloadIdx+2]))
        frameLength := int(frameHeaderLength + framePayloadLength)

        rBit := data[payloadIdx+5] >> 7

        if rBit != 0 {
            return fmt.Errorf("R bit is not unset")
        }
        if payloadIdx+frameLength > payloadLength {
            return fmt.Errorf("Payload length couldn't contain Payload with the length mentioned in
Frame Header")
        }

        var framerOutput bytes.Buffer
        r := bytes.NewReader(data[payloadIdx : payloadIdx+frameLength])
        framer := http2.NewFramer(&framerOutput, r)

        frame, err := framer.ReadFrame()
        if err != nil {
            return err
        }
        frames = append(frames, frame)

        payloadIdx += int(frameLength)
    }

    if payloadIdx != payloadLength {
        return fmt.Errorf("Payload length is not equal with the Frame length mentioned in Frame
Header")
    }

    h.BaseLayer = layers.BaseLayer{Contents: data[:len(data)]}
    h.frames = frames
    return nil
}
```

After multiple trials, I found out that `http2.Framer` would get stuck if we give a data that's not a valid HTTP/2 frame format (as depicted below). This means we need to find a way to classify whether the bytes of data is a valid frame or not. RFC 7540 document doesn't mention any way to classify a HTTP/2 frame, so I came up with a currently working solution by checking:

- Is the frame length specified in the frame header the same with the actual payload length?
- Is the R bit is unset?

```
+-----------------------------------------------+
|                 Length (24)                   |
+---------------+---------------+---------------+
|   Type (8)    |   Flags (8)   |
+-+-------------+---------------+-------------------------------+
|R|                 Stream Identifier (31)                      |
+=+=============================================================+
|                   Frame Payload (0...)                      ...
+---------------------------------------------------------------+
```

After we check the validity of the frame, we want to utilize Go's `net/http2` package to decode the frame for us. We do that by creating a new `framer` and pass the data to the `framer` . Next we call the `ReadFrame` function to get the actual HTTP/2 frame.

Intercepting the frames

Next, we’re going to capture the packet and decode the HTTP/2 frames.

```
package main

import (
    "bytes"
    "fmt"
    "log"
    "time"

    "github.com/google/gopacket"
    "github.com/google/gopacket/layers"
    "github.com/google/gopacket/pcap"

    "golang.org/x/net/http2"
)

var (
    device       string        = "lo0"
    snapshot_len int32         = 1024
    promiscuous  bool          = false
    timeout      time.Duration = 900 * time.Millisecond
    filter       string        = "tcp"
)

func main() {
    // Open device: We could also use other options (i.e. Open a .pcap file)
    handle, err := pcap.OpenLive(device, snapshot_len, promiscuous, timeout)
    if err != nil {
        log.Fatal(err)
    }
    log.Printf("Successfully opened live sniffing on %s\n", device)
    defer handle.Close()

    var h2c HTTP2

    // Create a parser to decode our HTTP/2 frame
    parser := gopacket.NewDecodingLayerParser(LayerTypeHTTP2, &h2c)

    // Use the handle as a packet source to process all packets
    source := gopacket.NewPacketSource(handle, handle.LinkType())
    decoded := []gopacket.LayerType{}

    // Process every packet
    for packet := range source.Packets() {
        ipLayer := packet.NetworkLayer()
        if ipLayer == nil {
            log.Println("No IP")
            continue
        }

        // Cast the layer to either IPv4 or IPv6
        ipv4, ipv4Ok := ipLayer.(*layers.IPv4)
        ipv6, ipv6Ok := ipLayer.(*layers.IPv6)
        if !ipv4Ok && !ipv6Ok {
            log.Println("Failed to cast packet to IPv4 or IPv6")
            continue
        }

        tcpLayer := packet.Layer(layers.LayerTypeTCP)
        if tcpLayer == nil {
            log.Println("Not a TCP Packet")
            continue
        }

        tcp, ok := tcpLayer.(*layers.TCP)
        if !ok {
            log.Println("Failed to cast packet to TCP")
            continue
        }

        appLayer := packet.ApplicationLayer()
        if appLayer == nil {
            log.Println("No ApplicationLayer payload")
            continue
        }

        packetData := appLayer.Payload()
        if err := parser.DecodeLayers(packetData, &decoded); err != nil {
            fmt.Printf("Could not decode layers: %v\n", err)
            continue
        }

        fmt.Println("*****************************************************")
        if ipv4Ok {
            fmt.Println("IPv4 SrcIP:        ", ipv4.SrcIP)
            fmt.Println("IPv4 DstIP:        ", ipv4.DstIP)
        } else if ipv6Ok {
            fmt.Println("IPv6 SrcIP:        ", ipv6.SrcIP)
            fmt.Println("IPv6 DstIP:        ", ipv6.DstIP)
        }
        fmt.Println("TCP srcPort:       ", tcp.SrcPort)
        fmt.Println("TCP dstPort:       ", tcp.DstPort)
        fmt.Println("HTTP/2:            ", h2c.frame)
        fmt.Println("*****************************************************")
    }
}
```

Conclusion

Using Go’s native HTTP support and Gopacket, we could build a packet interception program for HTTP/2. For further HTTP/2 header processing, we could also use `net/http2/hpack` package to do the HPACK decoding and encoding.