Solutions for Chapter 1 – The Internet

June 7, 2023

1 Problems

Problem 1.1 ***Wireshark***
Install Wireshark (`https://www.wireshark.org/`) and have a look at your network traffic. Which protocols from Figure 1.1 did you find? Please argue which protocols cannot be monitored if Wireshark is executed on your client's device. (Hint: Look at Figure 1.2.)

Using Wireshark, you should be able to see all Internet protocols your client is using. These protocols are the same for all transmission hops from the IP layer upwards, but may differ in the TCP/IP link layer. If you are using WLAN to connect to the Internet, WLAN frames will be visible in Wireshark, but no WAN protocols that are used by your home router towards the Internet, or the link layer protocols Internet routers use.

Problem 1.2 ***IPv6***
According to BBC Earth Unplugged (see the Youtube Video), there are about 1,504,000,000,000,000,000,000,000, or 1.504 septillion grains of sand in the Sahara. Could we assign an IPv6 address to each of them?

1.504 septillions roughly correspond to 1.5×2^{80}, since $2^{10} = 1024 \approx 10^3$. So from the 2^{128} IPv6 addresses, we can assign more than $2^{47} \approx 128.000.000.000.000$ IPv6 addresses to each grain of sand in the Sahara.

Problem 1.3 ***UDP vs. TCP***
Suppose you want to stream a compressed audio file (lossless compression). Which transport protocol is better suited here: TCP or UDP?

Real-time audio and video are typically streamed via UDP, because the loss of single packets only minimally impacts audio/video quality. A retransmission of lost packets via TCP would, in this case, introduce a delay of 1 Round-Trip Time (RTT) or more, which would have a more noticeable impact.

However, when audio or video *files* are streamed, TCP can be used since these streams are typically buffered on the client side. In case that *compressed* audio (or video) files are transmitted, TCP must be used, since the loss of a single packet may result in the loss of a complete compression window. (See Section 12.5 for more details on compression.)

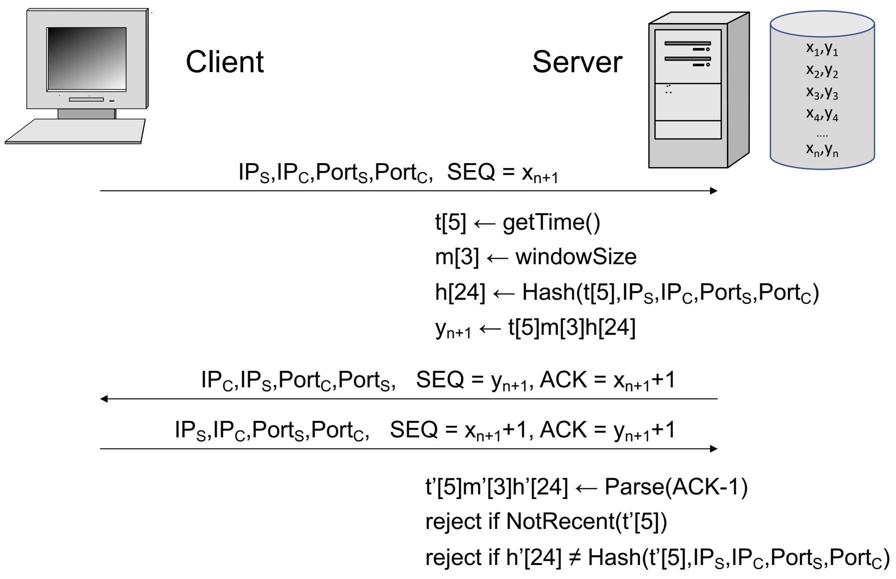
Client
Server
x_1,y_1
x_2,y_2
x_3,y_3
x_4,y_4
....
x_n,y_n
$IP_S,IP_C,Port_S,Port_C$, $SEQ = x_{n+1}$
$t[5] \leftarrow getTime()$
$m[3] \leftarrow windowSize$
$h[24] \leftarrow Hash(t[5],IP_S,IP_C,Port_S,Port_C)$
$y_{n+1} \leftarrow t[5]m[3]h[24]$
$IP_C,IP_S,Port_C,Port_S$, $SEQ = y_{n+1}$, $ACK = x_{n+1}+1$
$IP_S,IP_C,Port_S,Port_C$, $SEQ = x_{n+1}+1$, $ACK = y_{n+1}+1$
$t'[5]m'[3]h'[24] \leftarrow Parse(ACK-1)$
reject if $NotRecent(t'[5])$
reject if $h'[24] \neq Hash(t'[5],IP_S,IP_C,Port_S,Port_C)$

Figure 1: SYN-Cookies.

Problem 1.4 ***HTTPS***
Enter `https://en.wikipedia.org/wiki/Main_Page` *into your web browser and inspect the security information. Can you determine which encryption algorithm was used to retrieve the data from Wikipedia?*

The exact display of this information depends on the web browser you are using, and may be subject to change. Information of the encryption algorithm cannot be extracted from the web site certificate – if the web browser does not display this information, you'll habe to use Wireshark to look into the Server-Hello message which contains the ciphersuite to be used. For more information on this see Chapter 10.

Problem 1.5 ***DoS Attacks***
Have a look at RFC 4987 and try to figure out how SYN-Cookies mitigate SYN-Flooding attacks.

With SYN-Cookies, the server doesn't store any state when its TCP cache is full and thus remains reachable. The basic idea is to send back a stateless pseudorandom value (cf. Section 8.5.2 Photuris) instead of a randomly chosen value in the second message of the TCP handshake.

The process of computing and re-computing SYN-cookies is illustrated in Figure 1: A server whose TCP cache is full receives a TCP connection request from a legitimate client. This client has IP address IP_C and uses local port $Port_C$ to request a TCP connection to $(IP_S, Port_S)$. He sends sequence number x_{n+1} over this socket.

Upon receiving this SYN message, the server performs the following tasks:

- He retrieves a local 5-bit timestamp $t[5]$. This value slowly changes, approximately once for the RTT value of the network.
- He encodes an approximate *Maximum Sequence Size* (MSS) $m[3]$ with 3 bits. If the TCP cache weren't full, the exact value of MSS would be stored there.
- He computes a hash value h over $t[5]$ and the socket $(IP_S, IP_C, Port_S, Port_C)$. He extracts 24 Bits from this hash value and stores them in $h[24]$.
- He computes his sequence number y_{n+1} for the SYN-ACK message as the concatenation of $t[5]$, $m[3]$ and $h[24]$.
- After sending the SYN-ACK message, he erases all these values from memory.

A legitimate client will then answer with the ACK message of the TCP handshake, which contains the values $(SEQ = x_{n+1} + 1, ACK = y_{n+1} + 1)$, and the socket information $(IP_S, IP_C, Port_S, Port_C)$. The server will check this ACK message as follows:

- He parses the bitstring of $ACK - 1$ as $t'[5]m'[3]h'[24]$.
- He checks if $t'[5]$ is a recent value, i.e., if this time value lies within a configured interval before the current time.
- He computes a hash value over $t'[5]$ and the socket $(IP_S, IP_C, Port_S, Port_C)$. He extracts the same 24 Bits from this hash value and compares them to $h'[24]$.
- If this check succeeds, he clears an entry in his TCP cache, stores SEQ, ACK and $m[3]$ there, and waits for incoming TCP bytestreams.

Please note that using a standard hash function without a secret value may not be a good idea, because an attacker can retrieve the actual value of $t[5]$ by trying to connect to the server using a legitimate IP address. He may then be able to compute $h[24]$ from this value and the socket of the spoofed SYN packet. A simple way to mitigate this is to include a secret value in the hash computation which only the server knows.

More information on SYN-Cookies is available at `http://cr.yp.to/syncookies.html` and `https://en.wikipedia.org/wiki/SYN_cookies`.