Wireshark를 이용한 TCP 분석

제출자: 201421093 김민형

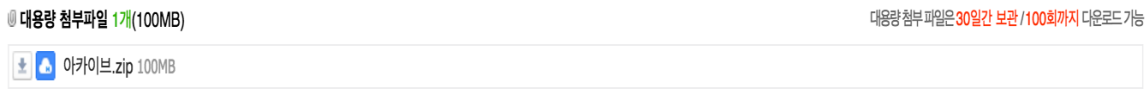
목차

1. 관찰 목표 및 예상 결과

우선 TCP 통신에서의 시작인 connection setup 과정을 알아보려 한다. 이 과정에서는 3-way handshake 를 통한 connection setup 이 이루어질 것이고, 실제로 이것이 이론과 일치하는지, 혹은 그렇지 않은지에 대해 알아볼 계획이다. 그 이후 Client 와 Server 사이에서의 세그먼트 delivery, cumulative ACK, fast retransmission 등 다운로드가 완료될 때까지 일어날 수 있는 일들을 관찰하고 segment 의 구조를 살펴볼 것이다. 마지막으로 connection close 과정에서 4-way handshake 가 발생할 것이고 이에 대해서 관찰할 계획이다.

2. 관찰 방법

네트워크는 아주대학교 기숙사 용지관의 Wi-Fi를 사용하여 진행하였다. 데이터 손실이 일어나서는 안 되는 file download에서의 TCP 분석을 위하여 약 100MB 용량의 사진 압축파일을 네이버 이메일 '나에게 보내기' 기능을 활용하여 서버에 업로드한 후 필자의 노트북에 다시 다운로드 하였다. 여기서 다운로드하는 과정을 약 30초 정도 capture 하였다.



< 사진 1. NAVER 이메일을 통한 파일 다운로드 >

이 과정에서 맥북 터미널에 'nslookup naver.com' 을 입력하여 네이버의 도메인 이름을 가지고 네이버의 IP 주소를 알아내어 'ip.addr' 필터에 적용 시킨다면 각 endpoint 사이의 세그먼트 흐름을 더 쉽게 관찰할 수 있을 것이다.

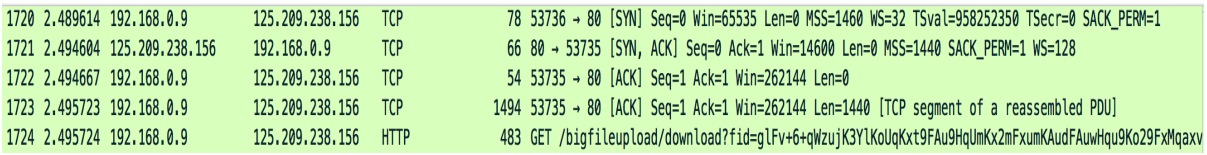
```
gimminhyeong-ui-MacBook-Pro:~ kimminhyung$ nslookup naver.com
Server:         168.126.63.1
Address:        168.126.63.1#53

Non-authoritative answer:
Name:   naver.com
Address: 125.209.222.142
Name:   naver.com
Address: 210.89.164.90
Name:   naver.com
Address: 210.89.160.88
Name:   naver.com
Address: 125.209.222.141
```

< 사진 2. nslookup을 통하여 알아본 NAVER의 IP 주소 >

3. Traffic 분석

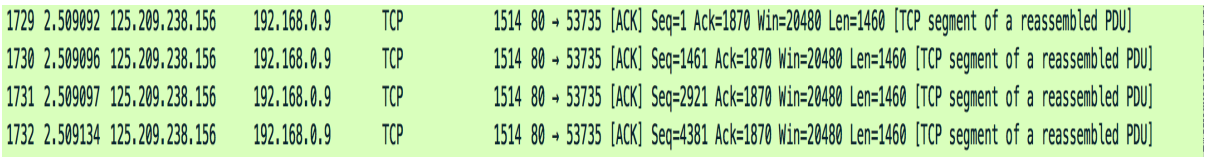
<사진 2>의 결과와 필자의 wireshark에 capture된 여러 IP를 비교한 결과 첫 번째 IP주소인 125.209.222.142와 연결한 것임을 알 수 있었다. 이후 ip.addr==125.209.222.142 필터를 사용하여 필자의 네트워크와의 상호작용을 알아보려 한다.



< 사진 3. 3-way handshake & HTTP file GET >

첫 번째 패킷에서 필자의 네트워크에서 네이버의 네트워크로 SYNbit=1, Seq=0 세그먼트를 보냈다. 이는 서버에 보내는 TCP 연결 요청으로, 이어서 두 번째 패킷에서는 서버로부터 필자의 네트워크로 SYN과 ACK을 동시에 보내온다. 받은 sequence number에 해당하는 숫자의 ACK을 보내는 Go-Back-N과는 다르게 다음에 받아야 할 sequence number에 해당하는 숫자로 ACK을 보내는 hybrid형 방식이다. 이후 클라이언트는 서버가 live 하다는 것을 인식하고 이에 대한 ACK을 보내 서버에게 클라이언트가 live 하다는 것을 입증하여 connection setup을 완료한다. 그 이후 server에 있는 파일을 받아오기 위해 HTTP server file을 GET method를 이용하여 받아온다.

필자는 수업에서 sequence number가 0부터 시작하지 않는다고 배웠다. 그에 대한 이유는 검색을 해보라고 말씀하셨었다. Sequence number가 0부터 시작하지 않는 것은 보안상의 이유로, 다른 단말과의 연결에서의 중간번호로 인식될 수 있으므로 0부터 시작하지 않는다고 한다. 하지만 필자가 capture 한 패킷의 sequence number는 0부터 시작한다. Sample에 올라온 타인의 보고서에도 sequence number는 난수라고 불러도 될 만큼의 예측 불가능한 숫자로부터 시작한다. 그 이유는 생각해보면 간단했다. Sequence number가 0부터 시작하지 않는 이유는 보안이라고 앞서 언급했다. 다른 연결에서 만들어진 sequence number에 중복되지 않기 위함인데, 필자가 내려받은 파일은 필자가 직접 '나에게로 보낸 이메일'에서 업로드한 것이기 때문에 그 누구도 접근할 수 없었고 필자는 최초의 downloader가 된다. 그로 인해 sequence number가 0부터 시작하는 것이 이상하지 않은 것이라고 쉽게 유추할 수 있다.



< 사진 4. 서버에서 넘어오는 파일 segment >

위 사진에서의 패킷들은 모두 1,514byte로 일정하다. 패킷의 헤더가 14byte를 차지하고, 그 상위 계층인 Network layer에서는 IP 헤더가 20byte를 차지한다. 그리고 그 상위 계층인 Transport layer에서는 TCP 헤더가 20byte를 차지한다. 그러므로 Transport layer에서 처리하는 payload는 총

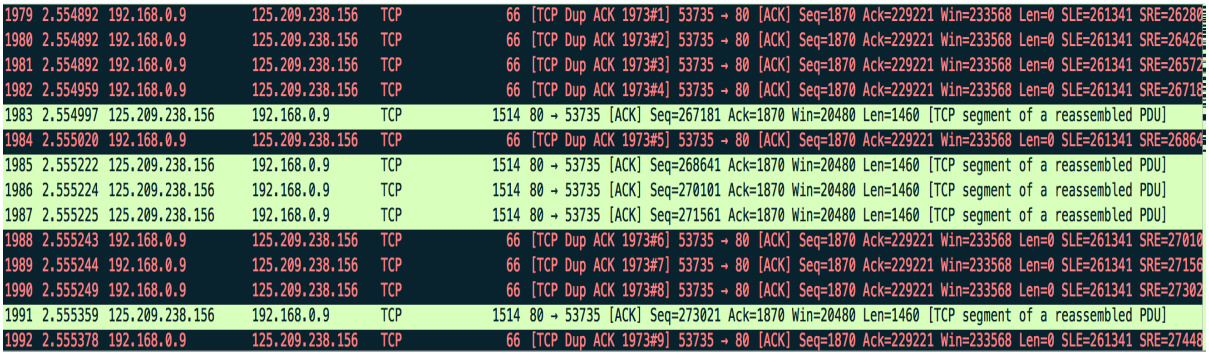
1,460byte이다. 이는 Sequence number의 변화로도 알아볼 수 있는데, 처음 보낸 패킷의 Sequence number는 1이다 그다음은 1,461 → 2,921 → 4,381로 모두 1,460 만큼의 차이를 보인다.

Trace 목록을 내리다보면 TCP Window Update라는 문구를 띤 패킷을 자주 볼 수 있다.

< 사진 5. TCP Window Update >

TCP Window Update 는 더 이상 ACK 을 보내지 않는 ACK 패킷이지만 window size 를 늘린다. Application 이 버퍼에 오랫동안 머물었던 데이터를 읽고 이때 수신된 데이터의 속도를 application 이 consume 하는 속도를 따라잡지 못할 때 window size 를 늘린다. 따라서 TCP 가 데이터를 수신하여 버퍼에 저장할 때와 Application 이 소켓에서 실제로 이 데이터를 읽을 때까지 delay 가 자주 발생하면 window update 가 발생한다.

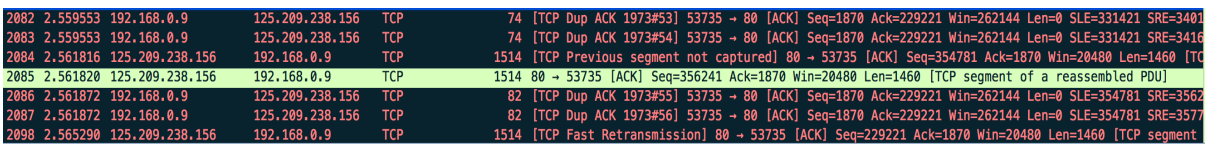
1 차과제에서 알아보았듯 검정색을 띤 패킷은 문제가 생긴 패킷이다. 아래 사진은 Duplicate ACK 의 사례이다.



< 사진 6. Duplicate ACK >

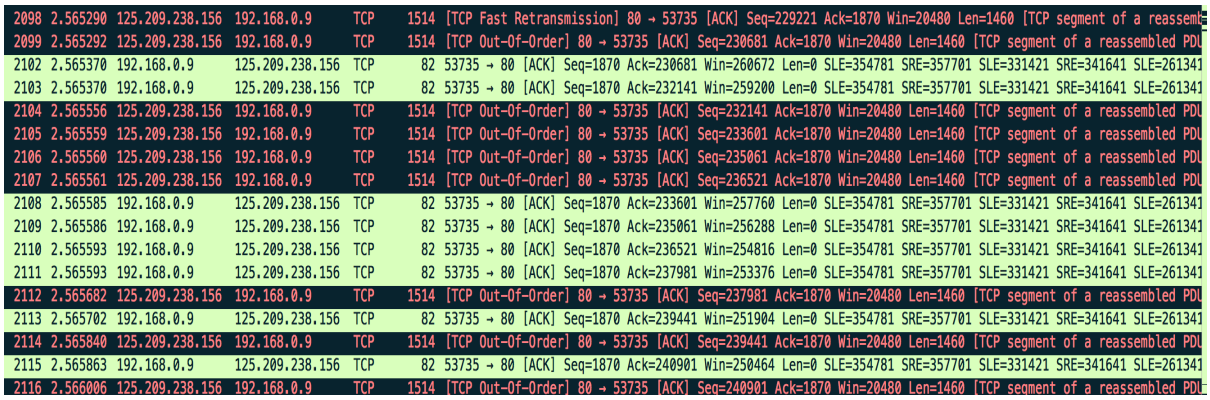
검은색 패킷들을 자세히 보면 Ack number 가 229,221 로 동일하게 바뀌지 않고 계속해서 보내는 것을 알 수 있다. 이것은 TCP 통신이 Cumulative ACK 을 사용한다는 것을 알 수 있게 해준다. 클라이언트는 서버로부터 sequence number = 229,221 에 해당하는 세그먼트를 받지 못했다. 그러므로 그 이후에 정상적으로 받은 세그먼트들과는 무관하게 그저 sequence 를 완성시키기 위해서 서버에게 지속해서 Seq=229,221 에 해당하는 세그먼트를 요청하는 것을 알 수 있다.

우리는 3 번째 duplicate ACK 부터는 데이터가 중간에 유실되었다고 판단하여 fast retransmit 을 한다고 배웠다. 필자는 Seq=229,221 에 해당하는 세그먼트의 retransmission 패킷을 찾아보았다. 아래 사진은 해당 세그먼트에 대한 retransmission 이다.



< 사진 7. TCP Fast Retransmission (Seq=229221) >

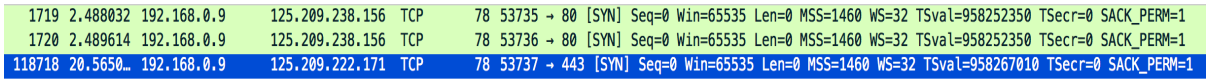
<사진 7>의 마지막 줄을 보면 Seq=229,221에 해당하는 세그먼트를 서버가 클라이언트에게 재전송해준 것을 볼 수 있다. 하지만 그 윗줄을 보면 Seq=229,221에 해당하는 Dup ACK의 번호는 56번이다. 동일한 ACK이 무려 56번이나 일어났다. 필자는 세 번째 Dup ACK부터 반응하여 빠르게 재전송 해줄 것이라고 생각했다. 하지만 네트워크에서의 작업은 굉장히 짧은 시간에 많은 일이 이루어지기 때문에 그리 늦은 반응이었다고 생각하지 않는다. 실제로 관찰한 결과 세 번째 ACK에서부터 fast retransmission이 발생하는 데까지 걸린 시간은 10ms가 채 되지 않는다.



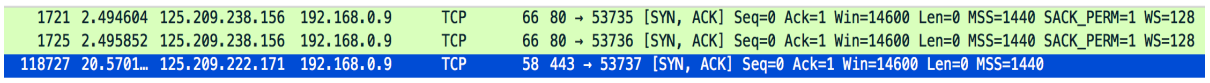
< 사진 8. TCP Out-of-Order >

<사진 8>과 같이 Retransmit이 일어난 직후에 아주 많은 TCP Out-of-Order들이 발생하는 것을 볼 수 있다. 이는 중간에 빠져있던 Sequence number가 들어오면서 재정렬 되는 과정으로 패킷에 대한 오류라고 보기는 어렵다. 그러므로 깊게 다루지 않으려 한다.

TCP flooding attack이란 attacker로부터 서버에게 무수히 많은 SYN 세그먼트를 보내서 서버가 자원할당을 하게 만든 후 서버가 준비된 이후에는 ACK 세그먼트를 보내주지 않아 서버가 할당시켜 놓은 자원들을 block 하는 DoS 공격이다. 필자가 다운로드한 파일은 필자의 개인 이메일 공간이므로 서버 공격을 당했을 것이란 생각은 들지 않지만 다른 사례에도 적용할 기회가 있을 수 있기에 wireshark를 통해서 서버가 SYN attack을 당했는지 알아보려 한다.



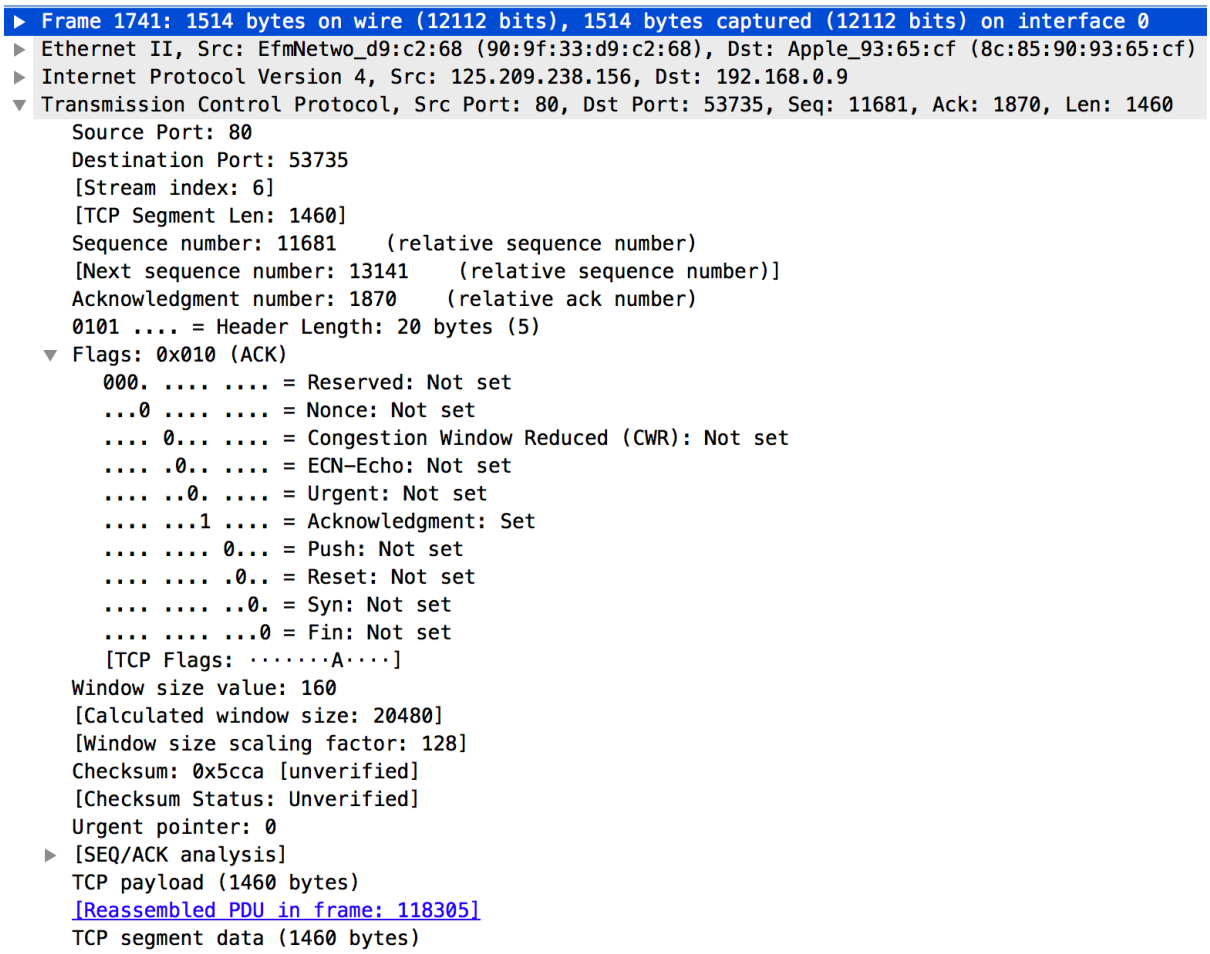
< 사진 9. tcp.flags.syn==1 && tcp.flags.ack==0 필터 적용 >



< 사진 10. tcp.flags.syn==1 && tcp.flags.ack==1 필터 적용 >

<사진 9>는 TCP 연결 요청 세그먼트를 필터하였고 <사진 10>은 서버가 요청에 대한 SYNACK을 보낸 세그먼트를 필터한 것이다. 두 필터를 비교했을 때 세그먼트의 개수가 동일하다면 방화벽이나 서버가 잘 버티고 있다는 뜻이다. 하지만 그렇지 않다면 DoS 공격을 당했다고 볼 수 있다.

이제 필자는 정상적으로 delivey 된 패킷 중 하나를 골라 TCP 헤더 구조를 알아보려 한다. 아래 사진은 TCP 헤더의 구조와 패킷 정보를 보여주고 있다.



< 사진 11. TCP 헤더 구조 >

Source Port는 패킷을 보내는 포트 번호이고 Destination Port는 패킷을 받는 포트 번호이다. 즉 이 패킷은 포트 번호 80에서 포트 번호 53,735로 보내졌다. Sequence number, next Sequence number 그리고 Acknowledgment(ACK) number는 단어 그대로의 뜻을 가지고 있고 뒤에 오는 숫자에 해당한다. 여기서 각 숫자 뒤에 있는 괄호 안에 relative라는 표현이 있는데 상대적이라는 뜻이다. 앞서 필자가 sequence number가 0에서 시작한 이유에 대해서 최초의 downloader라고 유추해냈지만, 필자는 이 단어를 보고 그저 사용자의 가독성을 위해 상대적인 숫자를 보여준 것 일 수도 있다고 생각했다. 그다음으로 헤더의 크기가 20byte라는 것을 알 수 있다.

다음은 flag 정보인데, 각 정보에 할당된 비트가 0이면은 false이고 1이면 true라고 읽을 수 있다. Congestion Window Reduced는 congestion window가 줄어들었는지에 대한 정보이고 0 이기 때문에 false이다. Acknowledgment는 이 패킷이 ACK인지 아닌지에 대한 정보이다. 하지만 TCP trace 목록을 살펴보면 ACK이 아닌 패킷이 있을지에 대한 의문이 생기기도 했다. Push는 buffer에

보관하지 않고 바로 application으로 전달할지에 대한 정보이다. 하지만 일반적으로 잘 사용되지 않는다. Reset은 연결이 비정상적일 때 연결을 끊고 재연결을 하기 위한 정보이다. True일 때 재연결한다. Syn은 연결의 시작을 요청하는 정보이고 Fin은 연결의 종료를 요청하는 정보이다.

Window size value 는 raw 한 윈도우 사이즈로 TCP 헤더로부터 직접 읽힌다. Calculated window size 는 실제 윈도우 사이즈로 congestion window 에 영향을 주는 요인이다. Window size scaling factor 는 그 값이 -1 의 경우에는 unknown 이고, -2 일 경우는 window scaling 이 사용되지 않고 그 외의 양수 값들은 모두 실제 scaling size 인수를 나타낸다. Checksum 은 데이터의 비트가 전달되는 과정에서 변형되었는지 확인할 수 있는 bit 의 합계이다. 마지막으로 TCP 세그먼트 데이터(payload)의 크기는 1,460byte 라는 것을 확인할 수 있다.

이제 마지막으로 connection 이 종료하는 과정을 관찰할 것이다.

```
118309 20.277609    192.168.0.9      125.209.238.156  TCP      54 53735 → 80 [FIN, ACK] Seq=1870 Ack=104418295 Win=1048576 Len=0
118312 20.282852    125.209.238.156  192.168.0.9      TCP      54 80 → 53735 [ACK] Seq=104418295 Ack=1871 Win=20480 Len=0
118715 20.561752    192.168.0.9      125.209.238.156  TCP      54 53736 → 80 [FIN, ACK] Seq=1 Ack=2 Win=262144 Len=0
118719 20.566984    125.209.238.156  192.168.0.9      TCP      54 80 → 53736 [ACK] Seq=2 Ack=2 Win=14720 Len=0
```

< 사진 12. 4-way handshake >

우리가 수업에서 배운 4-way handshake connection close는 아래의 순서를 갖는다고 했다.

1. 클라이언트→서버 (FIN)
2. 서버→클라이언트 (ACK)
3. 서버→클라이언트 (FIN)
4. 클라이언트→서버 (ACK)

하지만 <사진 12>와 같이 필자의 trace 목록 마지막 4개의 패킷은 앞서 말한 순서와 같지 않다. 그리고 세 번째 패킷에서는 뜬금없는 sequence number가 등장하였다. 왜 예상한 결과가 나오지 않았는지에 대해 알아보기 위해 패킷을 더 자세히 보기로 했다.

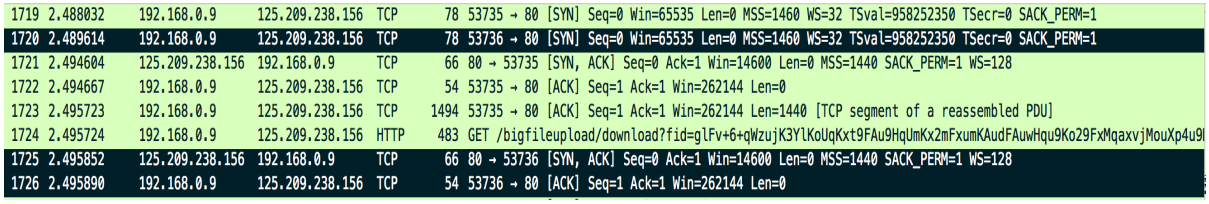
```
▼ Transmission Control Protocol, Src Port: 53735, Dst Port: 80, Seq: 1870, Ack: 104418295, Len: 0
    Source Port: 53735
    Destination Port: 80
    [Stream index: 6]
    [TCP Segment Len: 0]
    Sequence number: 1870    (relative sequence number)
    Acknowledgment number: 104418295    (relative ack number)
    0101 .... = Header Length: 20 bytes (5)
```

< 사진 13. 118309번째(그림 13에서 1번째) 패킷 상세정보 >

```
▼ Transmission Control Protocol, Src Port: 53736, Dst Port: 80, Seq: 1, Ack: 2, Len: 0
    Source Port: 53736
    Destination Port: 80
    [Stream index: 7]
    [TCP Segment Len: 0]
    Sequence number: 1    (relative sequence number)
    Acknowledgment number: 2    (relative ack number)
    0101 .... = Header Length: 20 bytes (5)
```

< 사진 14. 118715번째(그림 13에서 3번째) 패킷 상세정보 >

자세히 보니 각각의 Stream index가 달랐다. Stream index란 간단히 말해 conversation의 개수로 비유하면 좋을 것 같다. 통상 하나의 Stream index에서 오직 한 쌍의 IP만이 통신한다. 하지만 어째서 필자의 trace 목록에는 한 쌍의 IP에서 두 개의 stream index가 나온 것일까? 다시 시작 부분으로 돌아가 보았다.



< 사진 15. 다시 돌아온 trace 목록의 처음 부분 >

다시 돌아가 보니 처음 부분에서 3-way handshake가 두 번 일어난 것을 볼 수 있었다. 보기 쉽게 stream index=7에 해당하는 패킷들을 검은색으로 마킹하였다. 필자가 추측하는 바로는 stream index=7에 해당하는 3-way handshake가 시간차에 의해 stream index=6의 존재를 모른 채 연결을 시도했을것이고, 더 먼저 연결을 요청한 stream index=6을 통해서 file download가 이루어졌다고 생각한다. 그래서 모든 download 과정이 완료되고 나서 두 index는 각각 <사진 12>와 같이 연결 종료 요청을 한 것이다.

하지만 stream index=7의 존재를 알았을 뿐 여전히 필자의 TCP connection closing은 4-way handshake를 만족하지 않는다. 클라이언트는 서버에게 FIN을 요청했고 서버는 그에 대한 ACK을 보냈다. 그리고 FIN_WAIT_2 상태에 놓이게 된 상태다. 이 상태는 양방의 두 번의 통신이 이루어졌기 때문에 네트워크의 문제는 아닌 것으로 판단되며 서버에서 클라이언트로 FIN을 보낼 때 데이터 유실이 생겨 못 받은 것일 수도 있고 서버에서 close를 처리하지 못하는 경우일 수도 있다. 그래서 필자는 FIN_WAIT_2 상태에서 일정 시간이 지나 Timer가 expired 하여 클라이언트 스스로가 close 하게 된 사례라고 생각했다.

4. 새로 알게 된 지식

1차 분석에서 패킷 오류를 뜻하는 검은색으로 분류된 패킷들에 대해서 알아보았지만, 각각 다른 이유를 가진 오류들을 전부 해석하지 못하고 넘어갔던 것이 아쉬웠었다. 그리고 검색을 통해 해석이 이루어졌기 때문에 정확히 이해하기에는 무리가 있었다. 하지만 wireshark가 익숙해지고 또 수업을 듣고 네트워크 구조에 대한 지식을 어느 정도 습득한 이후에 해석하니 훨씬 수월했다. 덕분에 이번 분석에서는 필자의 trace 목록에 있는 모든 종류의 패킷을 빠짐없이 분석할 수 있었다. 그래서 전체적인 흐름의 이해도가 더욱 더 높았던 것 같다. 그리고 새로 알게 된 stream index라는 쓰임도 굉장히 흥미로웠다. Stream index는 여러 TCP/UDP Conversation이 서로 섞이지 않게 분류하려는 의도로 구현되었다. 수업에서 이론을 배울 때는 한 쌍의 연결만을 다루기 때문에 여러 쌍의 연결들을 한 번에 관찰할 수 있는 wireshark를 이용함으로써 새로운 것을 알 수 있었고, 실습의 중요성을 다시 한 번 생각하게 됐다.

5. 관찰 후 소감

전체적인 흐름이나 패킷 분석에서 첫 번째 분석 과제보다 수월하고 매끄러웠다고 생각한다. 이론에서 배운 것들이 그대로 wireshark에 capture 돼서 나오고, 또 예상하지 못한 결과들이 capture 되었더라도 충분히 생각하며 이해할 수 있었다. 그 예로 sequence number가 0부터 시작한 이유와 4-way handshake를 관찰하지 못한 것이 있다. 아직 sequence number가 0부터 시작한 이유에 대해서는 명확하지 않다. 필자가 최초의 downloader여서 그런 것인지 아니면 그저 relative 한 수치에 불과한 것인지 답을 내릴 수 없었다. 하지만 개인적으로 4-way handshake를 직접 관찰하지 못한 것에 대한 아쉬움이 크다. 처음부터 끝까지 패킷 분석을 하면서 관찰이 비교적 매끄럽게 진행되었다고 생각했지만, 마지막 부분에서 예상치 못한 결과가 나와서 애를 먹었다. 처음에 stream index의 존재를 모르는 상태에서는 도저히 이해할 수 없는 구조였기 때문이다. 하지만 이런 예상치 못한 결과 덕분에 새로운 지식을 얻고 생각하는 시간을 가질 수 있었다는것에 큰 의미를 두고 싶다.