Web (HTTP)

SPARCS SEMINAR 2016-10-12 SAMJO

Today's Topics

- World Wide Web
  - Concept
- HyperText Transfer Protocol 1.1
  - Request + Response
  - Methods, Status Code, Headers, Cookies …
- Transport Layer Security / Secure Socket Layer
- HyperText Transfer Protocol 2.0

Web?

"The **World Wide Web** is an information system of interlinked **hypertext documents** and other digital resources that are accessed via the Internet"

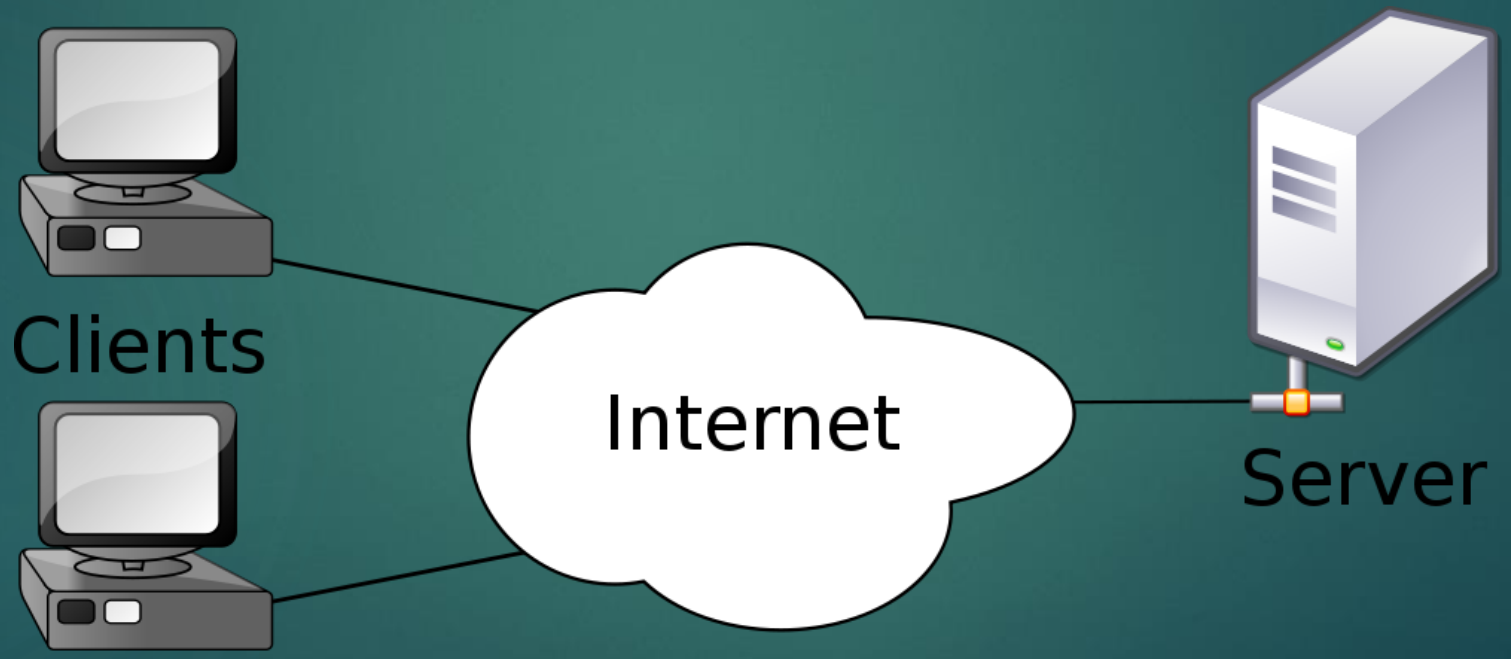
Sir. Tim Berners-Lee (1955-06-08 ~)

History – There is a light!

- 1989: CERN에서 자료공유 목적으로 hypertext라는 개념을 도입
  - "imagine, then, the references in this document all being associated with the network address of the thing to which they referred, so that while reading this document you could skip to them with a click"
- 1990: First web server, web browser (WorldWideWeb), web site (http://line-mode.cern.ch/www/hypertext/WWW/TheProject.html)
- 1991: WWW가 CERN외부에서 최초로 서비스됨

HyperText Transfer Protocol

- "an **application protocol** for distributed, collaborative, hypermedia information systems"
- Located in OSI Layer 7, Port: TCP 80



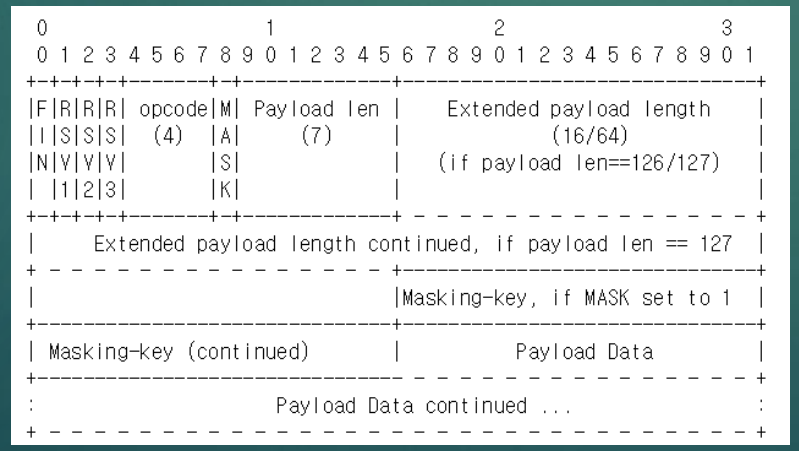
HTTP Basic

- Standards
  - HTTP/1.0 (1996)
  - HTTP/1.1 (1999): improvement in caching, connection optimization…
  - HTTP/2 (2015): compression on http headers (binary!)…
- Request: Client -> Server
- Response: Server -> Client
- Wait! What about Web sockets?
  - No, it is NOT HTTP

Web Socket (PEEK)

- RFC 6455, Enables two-way communication
- Layered over TCP, NOT rely on opening multiple HTTP connections
  - ex: XMLHttpRequest (ajax) / iframes

- URI: (ws|wss)://host:port/path/?query
- Handshake via HTTP/1.1 GET
- Data Frame:




```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
```

HTTP, again

- Managed by W3C (World Wide Web Consortium)
- RFC (Request for Comments) Standard
- IETF (Internet Engineering Task Force)

- -> Syntax Specification (RFC7230)

- 상당히 방대한 분량의 표준이 있음
- 이 세미나에서는 HTTP/1.1과 HTTP/2.0을 다룸

Appendix B. Collected ABNF

```
BWS = OWS

Connection = *( "," OWS ) connection-option *( OWS "," [ OWS
 connection-option ] )

Content-Length = 1*DIGIT

HTTP-message = start-line *( header-field CRLF ) CRLF [ message-body
 ]
HTTP-name = %x48.54.54.50 ; HTTP
HTTP-version = HTTP-name "/" DIGIT "." DIGIT
Host = uri-host [ ":" port ]

OWS = *( SP / HTAB )

RWS = 1*( SP / HTAB )

TE = [ ( "," / t-codings ) *( OWS "," [ OWS t-codings ] ) ]
Trailer = *( "," OWS ) field-name *( OWS "," [ OWS field-name ] )
Transfer-Encoding = *( "," OWS ) transfer-coding *( OWS "," [ OWS
 transfer-coding ] )

URI-reference = <URI-reference, see [RFC3986], Section 4.1>
Upgrade = *( "," OWS ) protocol *( OWS "," [ OWS protocol ] )

Via = *( "," OWS ) ( received-protocol RWS received-by [ RWS comment
 ] ) *( OWS "," [ OWS ( received-protocol RWS received-by [ RWS
 comment ] ) ] )

absolute-URI = <absolute-URI, see [RFC3986], Section 4.3>
absolute-form = absolute-URI
absolute-path = 1*( "/" segment )
asterisk-form = "*"
authority = <authority, see [RFC3986], Section 3.2>
authority-form = authority
```

HTTP Request Structure

[method] [URI] [Version]

[header field]:[header value]*

[body]

GET / HTTP/1.1

Host: www.naver.com

Connection: keep-alive

HTTP Request Method

- **GET**: URL에 해당하는 자료의 요청; 대부분의 페이지 이동
  - Query는 URL 끝에 붙음 (ex: abc.com/?id=sam&pw=abc)
- **POST**: 서버에 자료를 보내는 요청; 대부분의 버튼 클릭
  - Query는 request body에 삽입되어 전송됨
- PUT: 해당 URL에 자료를 저장
- DELETE: 해당 URL에 자료를 삭제
- TRACE, OPTIONS, CONNECT, HEAD

HTTP Response Structure

[version] [status code] [reason message]
[header field]:[header value]*
<CRLF>
[body]

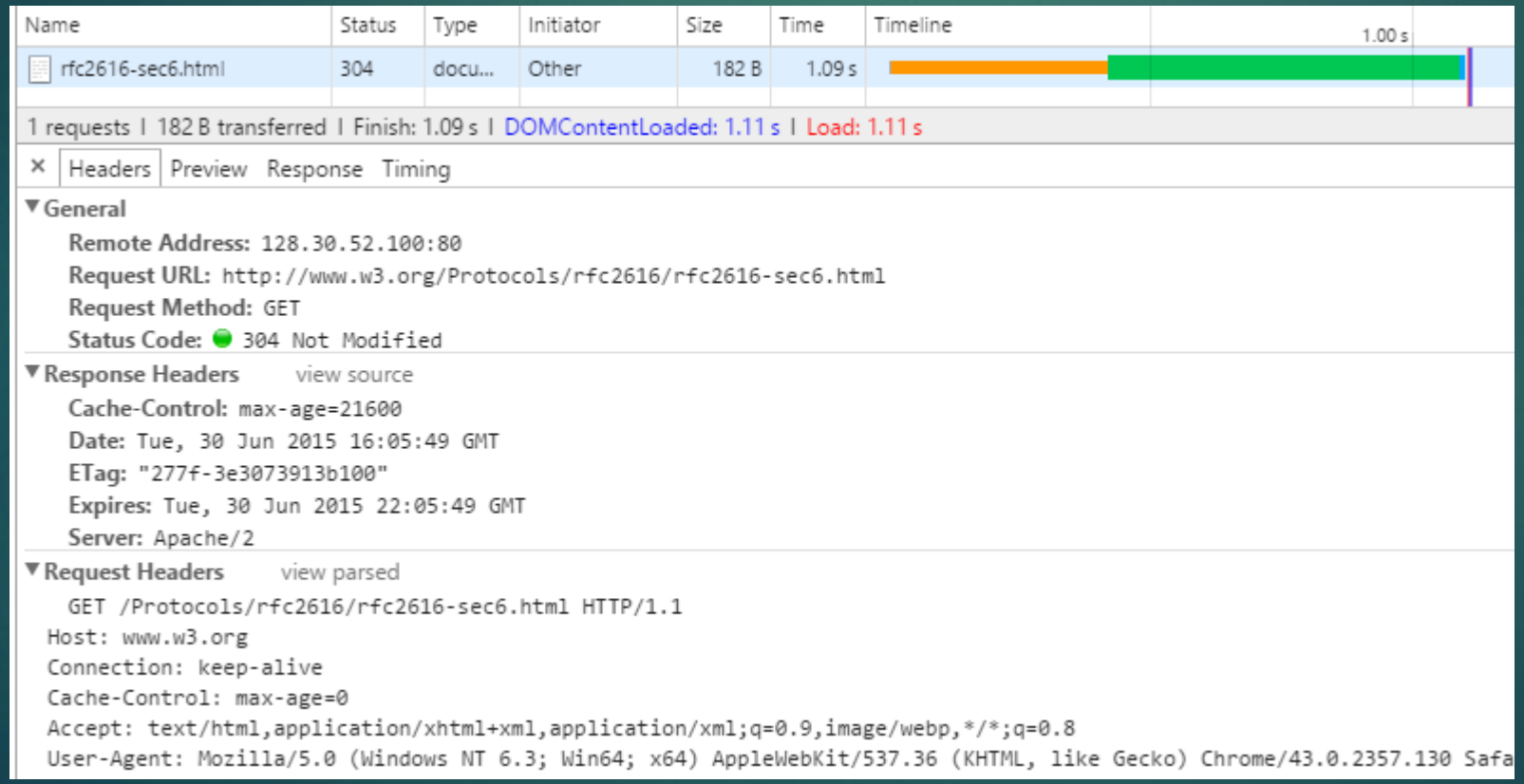
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
<CRLF>
<!doctype html> …

HTTP Status Code

- 1xx: 조건부 응답
- 2xx: 성공
  - 200: 성공
- 3xx: 리다이렉션 완료
  - 301: 영구 이동; 302: 임시 이동
  - 304: 수정되지 않음
- 4xx: 요청 오류
  - 400: 잘못된 요청; 401: 권한 없음; 403: 금지됨; 404: 찾을 수 없음
- 5xx: 서버 오류
  - 500: 내부 오류; 503: 서비스 사용 불가

HTTP Packet

- Chrome에서 F12를 눌러 개발자 도구를 연다
- Network탭을 열고, 웹 페이지를 하나 로드한다.



Headers

- Many… many… many

- Some famous headers
  - Host (Request)
  - Content-Length (Request / Response)
  - Content-Type (Request / Response)
  - User-Agent (Request)
  - Cookie (Request) / Set-Cookie (Response)

Headers – Host, Content-Length

- Host: “specifies the Internet host and port number of the resource being requested”
  - Introduced in HTTP/1.1
  - 1 IP = Multiple Host
  - EX: Host: www.naver.com
- Content-Length: “indicates the size of the entity-body”
  - EX: Content-Length: 1234

Headers – Content-Type

- Content-Type: “indicates the media type of the entity-body”

- text/html, text/plain
- multipart/form-data
- image/jpeg, image/gif, image/png, …
- application/octet-stream, application/xml, …

- Search MIME Type on the Web!

Headers – User-Agent

- Web Browser의 종류를 서버에게 알려주기 위해 만든 header
- 서버는 이 문자열로 사용자의 web browser을 식별할 수 있음
  - **주의: 매우 당연하게 이를 다른 값으로 바꿔서 보낼 수 있음**
- ex: Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/53.0.2785.143 Safari/537.36

Headers – Cookie / Set-Cookie

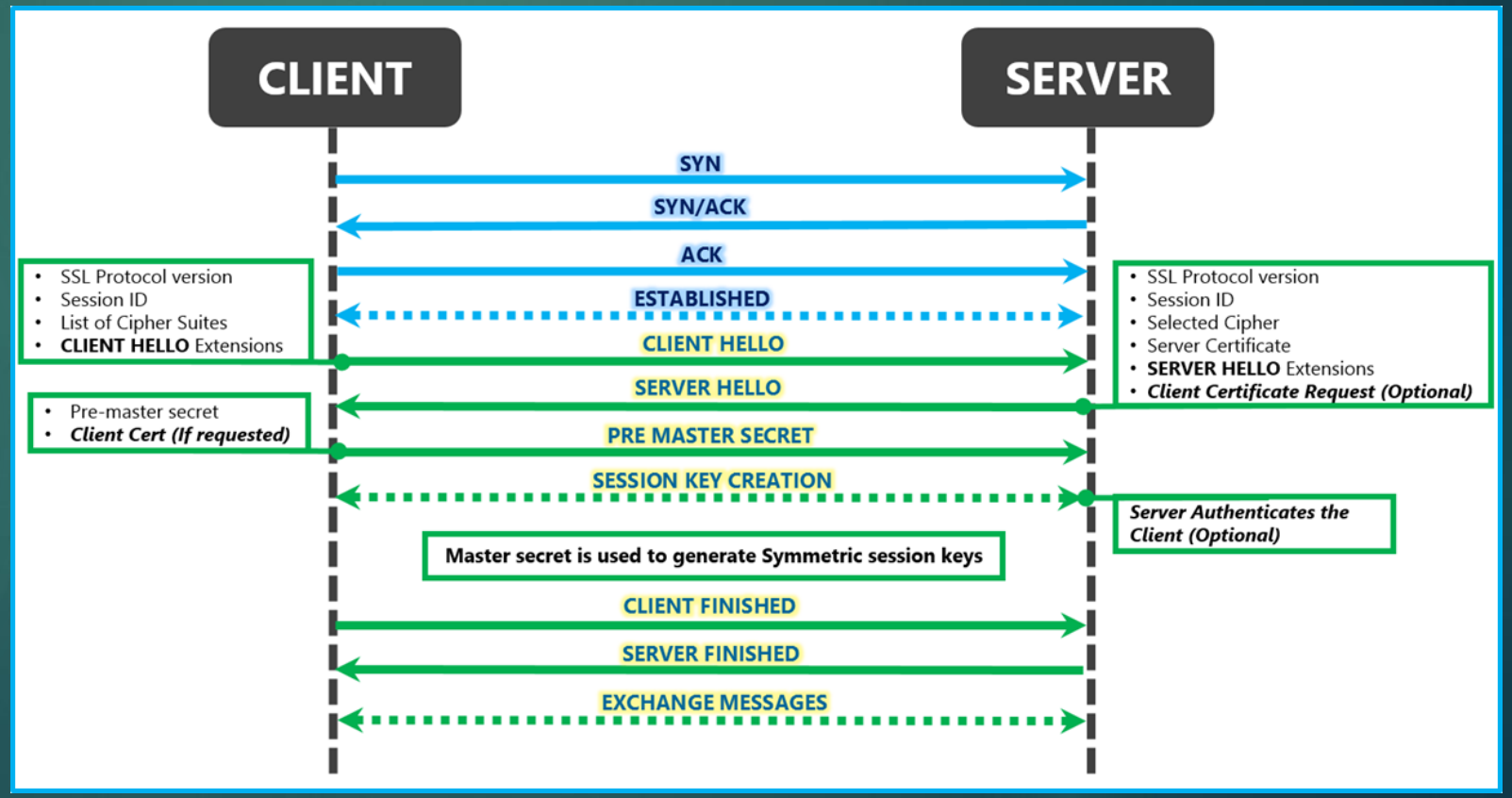
- "a small piece of data sent from a website and **stored in a user's web browser** while the user is browsing that website"
  - 웹 사이트가 사용자의 정보를 기억하는데 사용
  - 로그인, 온라인 쇼핑몰 쇼핑 카트, 광고 등에 사용
  - **사용자의 웹 브라우저에 저장되므로 값을 믿어서는 안됨**

- 1. 웹 사이트가 Response Header의 Set-Cookie라는 항목에 쿠키 정보를 만들어 보냄
- 2. 웹 브라우저는 이를 사용자 컴퓨터에 저장함
- 3. 해당 사이트를 방문할 때마다 Request Header에 넣어서 보냄

TLS / SSL – Add Some Security!

- TLS (Transport Layer Security) / SSL (Secure Socket Layer)
- HTTP를 암호화하자! (OSI Layer: 5 / Port: 443 / Scheme: https)

- Newest: TLSv1.2
- Working: TLSv1.3
- RFC 5246



TLS / SSL – How it works?

- 0. 서버가 인증서를 보내고 client가 이를 검증한다
  - Valid CA, Valid Hostname, Valid Start / End Date

- 1. ???을 사용하여 Client와 Server가 같은 키를 갖는다
  - 1. RSA
  - 2. Diffie-Hellman (Variants: Elliptic Curve, Option: Ephemeral)

- 2. 가지고 있는 같은 키로 ???방식으로 암호화하여 통신한다
  - Crypto: AES-256-CBC, 3DES-CBC, SEED-128-CBC, RC4, …
  - Checksum: SHA1, SHA256, SHA384, MD5, …

TLS / SSL – Cipher Suites

- Protocol – Kx (Key eXchange) – Au (Authenticate) – Enc (Encryption) – MAC (Message Authenticate Code)

- Example: TLS_ECDH_RSA_WITH_AES_128_CBC_SHA256
  - Protocol: TLS
  - Key Exchange: EC오
  - Authenticate: RSA
  - Encryption: AES_128_CBC
  - Integrity: SHA256

TLS / SSL Attacks

- 잘 사용하면 매우 좋으나 잘 사용하는 경우가 별로 없음

- DROWN / POODLE / Heartbleed / Weak Stream Cipher (RC4) / Weak Cert Signature (SHA1) / HTTP Strip (Lack of HTST) / CRIME (TLS Compression) / FREAK (Export Ciphers) / Logjam
  - 기억나는 것만 이 정도

- ECDHE 지원 / AES_128 이상 지원 / SHA-2 이상 지원
- eNULL, aNULL, EXPORT, RC4, MD5는 켜놓으면 안됨

HTTP/2.0

- RFC 7540 (https://http2.github.io)
- Why HTTP/2? HTTP/1.x is TOO SLOW
- Major Changes:
  - Binary
  - Fully Multiplexed
  - Stream Priority
  - Server Push
  - Header Compression
- Cover one-by-one

HTTP/2.0 - Binary

- Hard to read (by human)
- Easy to parse (by computer)
- More compact
- Less error-prone
  - Capitalization
  - Line Endings
  - Spacing
  - Example: 닷지 크롬 / 닷지 웹 (맨 앞에 /r/n 추가)

HTTP/2.0 – Fully Multiplexed

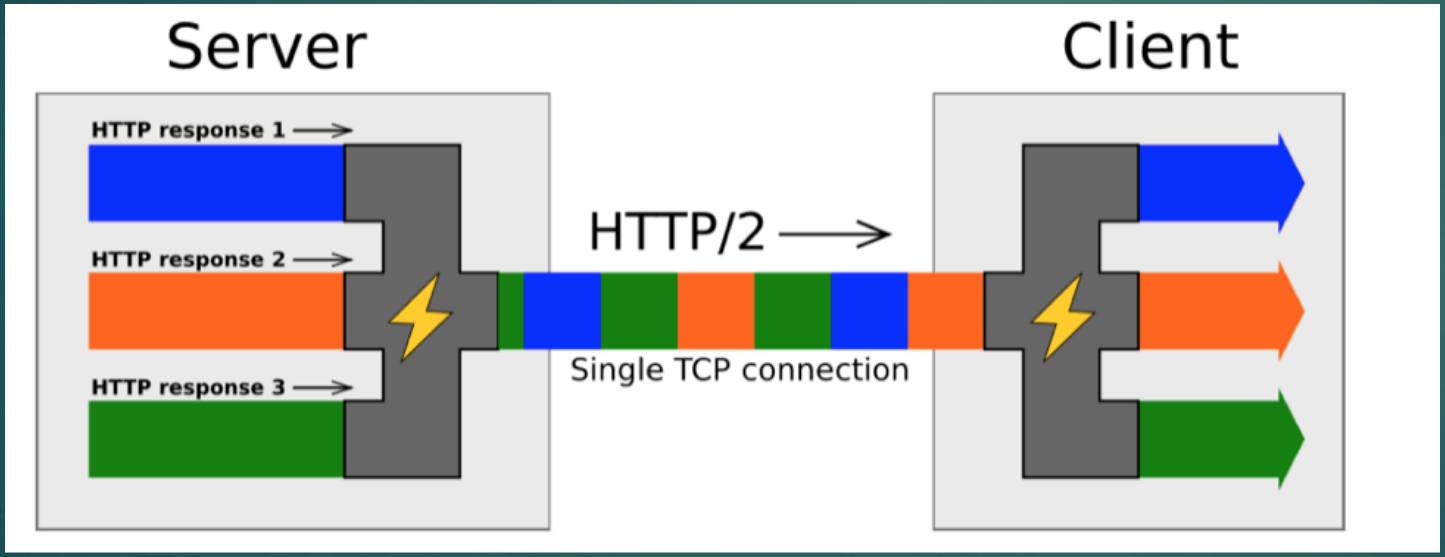
- Suppose an single page: 1 HTML + 20 .pngs
  - H: TCP 3-way handshake

- HTTP/1.0 : H, 1 HTML, H, 1 png, H, 1png, …
- HTTP/1.1 + KeepAlive: H, 1 HTML, 1 png, 1 png, …
- HTTP/1.1 w/ Multi TCP: H, 1 HTML, 7 H, 8 png, 8png, …
- HTTP/1.1 Pipelining : H, 1 HTML, 20 png
  - Problem: hard to implement (ONLY Opera supports + enabled by defaults)
  - + Poorly behaving servers…
- HTTP/2.0 Multiplexing

HTTP/2.0 Multiplexing

- Multiple TCP connections are SLOW (especially on mobile)
  - 3 way handshake and TCP slow-start process

- Frame by Time (source: nginx blog)



HTTP/2.0 Stream Priority

- Suppose a page: 1 HTML, 1 CSS, 2 .pngs

- Scenario 1: Server sent HTML, png, png, CSS
  - Rendering하기 위해서는 CSS가 반드시 필요
  - 마지막 CSS 파일이 도착하기 전까지는 흰 페이지로 표시
- Scenario 2: Server sent HTML, CSS, png, png
  - 일단 이미지가 들어갈 부분만 표시되지 않고 나머지 부분 렌더링 가능
  - 일단 화면에 뭔가 뜨기는 뜸 => 빨리 로딩되는 것처럼 느낌

- Client sent priority to servers!

HTTP/2.0 Server Push

- Warning: NOT related to “push notification”

- HTTP/1.1 Flow:
  - Request an HTML
  - Get the HTML
  - Parse the HTML
  - Request some pngs
  - Get the pngs
  - Render

- HTTP/2.0 Flow (w/ Server Push):
  - Request an HTML
  - Get the HTML and some pngs
  - Parse the HTML
  - Render

HTTP/2.0 Header Compression

- Easy, use the ultimate answer – GZIP (zlib)
- Pass 1. Duplicate string elimination (LZ77)
  - Remove duplicated strings, and reference is inserted.
- Pass 2. Bit reduction (Huffman coding)
  - Simple Huffman coding

CRIME

HTTP/2.0 DOES NOT USE GZIP

TLS/SSL CRIME Attack

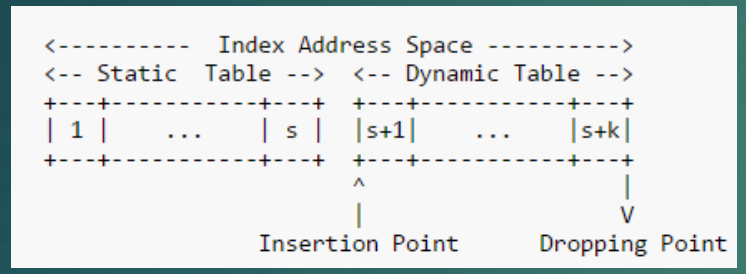
- Compression Ratio Info-leak Made Easy

- Silently add carefully chosen text to the HTTP request
  - via JavaScript by XSS, CSRF, …
- Observe the packet size

- EX: sessionid=abcdefg
  - If an attacker append “wxyz”, nothing changed
  - If an attacker append “abcd”, the packet will be short

HTTP/2.0 HPACK

- HPACK – Hader Compression for HTTP/2 (RFC7541)
  - Table Based (Static / Dynamic) + Huffman Code



- All string header values are encoded using Huffman
- Replace both header name and header value to index
  - Only if possible, and not sensitive

- Follow examples on https://http2.github.io/http2-spec/compression.html#examples

Q&A

THANK YOU!

References

- https://http2.github.io/