논문 17-42-03-07 The Journal of Korean Institute of Communications and Information Sciences '17-03 Vol.42 No.03
https://doi.org/10.7840/kics.2017.42.3.595

SSLmTCP 핸드쉐이크 : SSL 핸드쉐이크를 포함하는 TCP 3-단계 핸드쉐이크

변 기 석◆, 박 준 철°

SSLmTCP Handshake : Embedding the SSL Handshake into the TCP 3-Way Handshake

Ki-Seok Byun◆, Jun-Cheol Park°

요 약

본 논문에서는 SSL/TLS 핸드쉐이크를 TCP 3-단계 핸드쉐이크에 포함시켜 SSL/TLS 핸드쉐이크 시간을 단축하는 방식을 제안한다. 제안 방식은 기존 TCP에 선택적으로 추가하여 사용할 수 있으며, 적용 시 TCP와 SSL/TLS 핸드쉐이크가 순차적으로 일어나는 대신 겹쳐서 진행되도록 한다. 제안 방식의 프로토타입을 구현하였고, 성능측정을 위한 실험을 진행했다. 실험 결과, 기존 TCP 및 SSL/TLS의 순차적 핸드쉐이크에 비해, 제안 방식은 3.2%부터 14%(클라이언트에서 서버까지의 핑 프로그램에 의한 소요시간이 11.6ms일 때) 수준의 시간 절감을 달성하였다. 클라이언트와 서버 간의 핑 프로그램의 측정된 소요시간이 늘어남에 따라 실행시간 감소율 또한 증가하였는데, 여기서 핑 프로그램의 소요시간은 전파지연 및 큐잉지연이 커짐에 따라 증가하게 된다. 실험 결과는 제안 방식으로 인해 절감된 시간이 핑 프로그램의 측정된 소요시간에 비례하여 커질 것이라는 예측과 잘 부합한다.

Key Words : SSL/TLS Handshake, TCP Handshake, Embedding Handshake, Time Reduction, Linux Kernel Hacking

ABSTRACT

We propose a scheme to reduce the time for the SSL/TLS handshake by embedding it into the TCP 3-way handshake. The scheme can be selectively applied on the standard TCP for making the SSL/TCP handshake happen within the TCP handshake, rather than performing the TCP handshake and SSL/TLS handshake in sequence. We implemented a prototype of the scheme and did some experiments on its performance. Experimental results showed that, compared to the sequential handshakes of the TCP and the SSL/TLS, the time reduction achieved by the scheme varied in the range of 3.2% and 14%(when the elapsed time by the ping program from the client to the server was 11.6ms). The longer the time measured by the ping program, which would grow as the propagation and queuing delays do, the larger the reduction rate. It accords with the supposition that the reduced time due to the scheme will increase in proportion to the amount of the elapsed time measured by the ping program.

◆ First Author : Cryptosystem Development PKI Team, Penta Security Systems, qusrl0715@naver.com, 정회원
° Corresponding Author : Department of Computer Engineering, Hongik University, jcpark@hongik.ac.kr, 종신회원
논문번호 : KICS2016-12-407, Received December 28, 2016; Revised February 14, 2017; Accepted February 15, 2017

Ⅰ. 서 론

웹 트래픽의 보호를 위해 HTTPS(HyperText Transfer Protocol over Secure Sockets Layer)가 널리 사용되고 있다. HTTPS는 웹 통신에 사용되는 HTTP에 SSL(Secure Sockets Layer)를 적용시킴으로써 클라이언트와 서버 간 비밀통신을 가능하게 한다[1]. Alexa 조사에 따르면 2016년 11월, 상위 1,000,000개 웹사이트의 11.6%가 HTTPS를 기본 값(default)으로 채택하고 있으며[2], 2016년 7월, 전체 페이지 불러오기 중 45%가 HTTPS를 사용해 실행됐다[3]. 구글(Google) 등의 기업은 모든 웹 통신에 HTTPS를 사용하자는 HSTS(HTTP Strict Transport Security) 정책을 채택하고 있으며, 향후 HTTPS의 사용 비율은 점점 높아질 것으로 예상된다[4].

HTTPS는 비밀 통신을 위해 SSL을 사용하므로 SSL 핸드쉐이크(handshake) 과정을 거치는데, SSL은 핸드쉐이크를 위해 전송 계층 프로토콜로 TCP를 사용한다. 따라서 SSL 핸드쉐이크 이전에 TCP 핸드쉐이크가 선행되어야 한다. 결과적으로 HTTPS를 사용하기 이전에 TCP 핸드쉐이크 및 SSL 핸드쉐이크 과정이 모두 필요하다. 그러므로 두 가지 핸드쉐이크 과정에 소요되는 시간을 줄이는 것이 사용자의 HTTPS를 통한 웹 서비스의 응답시간 절감에 도움이 된다.

SSL/TLS 사용 시의 실행시간 경감을 위하여, TLS 스냅 스타트(TLS snap start), TLS 서버 인증서 미리 가져오기(Prefetching TLS Server Certificates) 같은 기술들이 제시되었다. 이런 기술들은 과거 통신 내용을 이용하여 SSL 핸드쉐이크 과정을 단축하거나, 클라이언트가 통신을 시도하기 전에 SSL 핸드쉐이크를 미리 실행해 총 실행시간을 줄인다. 이들은 각각 총 실행시간을 줄여줄 수 있어 클라이언트 입장에서의 응답시간 단축 효과를 기대할 수 있지만, 한계점이 존재한다. TLS 스냅 스타트는 처음 접근하는 사이트를 상대로 사용할 수 없으며, TLS 서버 인증서 미리 가져오기는 미리 인증서들을 가져온 사이트 대비 실제 접근하는 사이트의 수가 적으면 시행하기 위한 부하가 높아진다.

본 논문에서는 두 핸드쉐이크 과정의 통합을 통하여 SSL 핸드쉐이크에 소요되는 시간을 줄일 수 있는 SSLmTCP(SSL embedded TCP) 핸드쉐이크를 제안한다. SSLmTCP 핸드쉐이크는 TCP 3-단계(way) 핸드쉐이크 과정에 SSL 핸드쉐이크 과정의 메시지들을 삽입하여 3개의 SSL 핸드쉐이크 메시지가 별도로 전송될 필요가 없어진다는 장점을 가진다. 그 결과로 SSLmTCP 핸드쉐이크를 이용한 통신과정은 서버 입장에서의 메시지 수신 및 처리 과정 단순화와 클라이언트 입장에서의 응답시간 단축 효과를 기대할 수 있다.

Ⅱ. TCP 핸드쉐이크와 SSL 핸드쉐이크

TCP(Transmission Control Protocol)[5,6]는 전송(transport) 계층의 프로토콜로서 세그먼트(segment)가 신뢰성 있게, 순서대로 도착하는 것을 보장하며 HTTP(HyperText Transfer Protocol) 등 다수의 인터넷 응용 프로토콜의 하위 서비스로 널리 사용된다. TCP는 신뢰성 있는 데이터 전달을 위해 전송 데이터가 포함된 세그먼트를 교환하기 전, 서버와 클라이언트 사이에 그림 1과 같은 3-단계 핸드쉐이크 과정을 거친다[7].

SSL은 전송 계층과 응용(application) 계층 사이에 위치하는 통신 규약이며, SSL3.0부터는 TLS(Transport Layer Security)라고도 불린다. TCP/IP 프로토콜 모음은 보안을 고려하여 설계되지 않았기 때문에 도청(eavesdropping), 중간자(man-in-the-middle) 공격 등에 취약하다[8-11]. 이들 공격을 막기 위한 방법 중 하나가 SSL/TLS의 사용이다. SSL은 전송 계층 위에서 송신자와 수신자가 주고받는 메시지를 암호화하여, 통신 내용이 제3자로부터 보호되는 것을 보장하는 일종의 보안 계층이다. SSL/TLS을 사용하면, 클라이언트와 서버가 교환하는 데이터가 암호화되어 전달되기 때문에 공격자는 이 데이터를 탈취하더라도 무슨 내용인지 알 수 없다.

SSL을 사용하기 위해선 그림 2와 같은 핸드쉐이크 과정이 필요하다. 핸드쉐이크에 필요한 정보들은 TCP 세그먼트에서 데이터가 포함되는 영역인 페이로드(payload)에 포함되어 전달된다[12]. 핸드쉐이크는 (Client Hello)를 서버에게 전송하며 시작된다. (Client Hello)에는 클라이언트가 생성한 랜덤 값인 R_C가 포함된다. 대기 중이던 서버는 이를 받고 (Server Hello, Server Certificate)를 전송한다. (Server Hello)에는 서

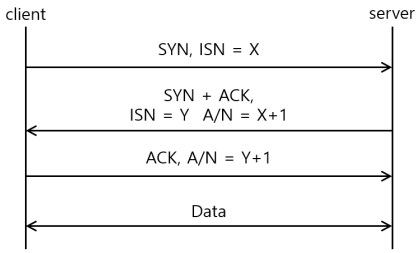

그림 1. TCP 3-단계 핸드쉐이크
Fig. 1. TCP 3-way handshake

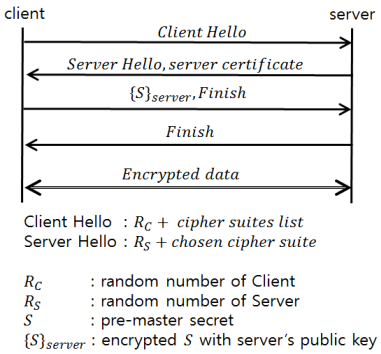

그림 2. SSL 핸드쉐이크
Fig. 2. SSL handshake

버가 생성한 랜덤 값 R_S가 포함된다. 클라이언트는 이를 받은 후 비밀 통신에 사용될 키들을 생성하는 데 필요한 프리-마스터 비밀 값(pre-master secret) S를 생성하고, 이 값을 (Server Certificate)에 저장된 서버의 공개키(public key)로 암호화한 $\{S\}_{server}$를 전송한다. 서버는 전송받은 내용을 자신의 개인키(private key)로 복호화 해 프리-마스터 비밀 값 S를 얻는다. 마지막으로 서버는 클라이언트에게 Finish를 보낸다. 핸드쉐이크 과정이 끝나면 서버와 클라이언트는 R_C, R_S, S를 이용해 서로 공유하는 세션 키(session key)들을 계산해 내고, 이들을 이용해 안전한 통신을 위한 암호화를 진행한다. 여기서의 안전이란 인증(authentication), 무결성(integrity), 기밀성(confidentiality)을 모두 제공함을 의미한다[13].

Ⅲ. 관련 연구

3.1 TLS 스냅 스타트(TLS snap start)

2010년 구글에서 제안한 TLS 스냅 스타트[14]는 Client Hello에 스냅 스타트 확장(snap start extension)을 추가하여, TLS 핸드쉐이크 과정의 서버 응답 메시지를 불필요하게 만든다. 스냅 스타트 핸드쉐이크에서는 클라이언트에서 서버로 단 하나의 메시지만 보내기 때문에, 클라이언트는 서버로부터 랜덤 값(2장의 R_S에 해당)을 받을 수 없다. 서버는 자신이 스스로 랜덤 값을 만드는 대신, 클라이언트가 제안한 20바이트의 랜덤 값(Client Hello에 포함), orbit 및 현재 시각을 이용해 자신의 랜덤 값을 생성한다[14]. 여기서 orbit은 서버가 생성한 8바이트 값으로, 초기 풀 핸드쉐이크를 수행할 때 클라이언트에게 전달된다.

서버는 재생(replay) 공격을 방지하기 위해 클라이언트가 전송한 orbit이 올바른지 확인하고, 클라이언트의 시각을 받아 서버의 현재 시각과 일정 간격 이내인지 확인한다. 또한, 반복 사용인지를 확인하기 위해 클라이언트가 제안한 20바이트의 랜덤 값을 저장하고 있다. 한편, 스냅 스타트를 사용하기 전, 클라이언트는 반드시 서버와 풀 핸드쉐이크 과정을 진행해야 한다. 이 과정 중에 클라이언트는 서버로부터 orbit 등 스냅 스타트 사용에 필요한 정보를 제공받고, 이후 통신부터 스냅 스타트를 사용할 수 있게 된다.

하지만 TLS 스냅 스타트에는 한계점이 존재한다. 인터넷을 이용하다 보면 새로운 사이트에 접근할 때가 많고, 한 번만 접근하게 되는 사이트도 많다. 이런 상황에는 한 번의 풀 핸드쉐이크가 선행되지 않았기 때문에 스냅 스타트를 사용할 수 없게 된다. 스냅 스타트는 핸드쉐이크 과정 중에서 서버의 인증서를 받지 않고, 대신 풀 핸드쉐이크 때 받았던 서버의 인증서를 캐시(cache)해 사용한다. 만일 통신을 시작하기 전에 이 인증서가 만료되면 서버로부터 새로운 인증서를 받아야 하므로 이 경우에도 풀 핸드쉐이크 과정을 실행해야 한다. 본 논문의 제안 방식은 처음 접근하는 사이트와의 통신에도 적용 가능하다.

3.2 TLS 서버 인증서 미리 가져오기

서버 인증서 미리 가져오기[15]는 사용자가 웹 사이트를 이용하는 동안 브라우저의 현재 화면에서 이후 방문 가능성이 있는 웹 사이트들의 인증서 전부를 미리 가져오는 것을 말한다. 그 후 핸드쉐이크에 필요한 계산을 미리 해두어, 사용자가 특정 사이트에 접근하면 즉시 TLS를 이용한 통신을 가능하게 한다.

서버 인증서 미리 가져오기를 위해서는 미리 상당량의 계산을 해 놓아야 한다. 특히 주요 포털 사이트 등에 방문하고 있다면, 현재 화면에서 접근 가능한 사이트가 많아서 추가 부담이 높아지게 된다. 연구결과[15]에 따르면 미리 가져오기 된 사이트 중 20% 정도의 사이트에 실제 접근할 때 추가 부담은 약 5%이고, 10% 정도의 사이트에 실제 접근할 때 추가 부담은 약 10%로 측정되었다.

Ⅳ. SSLmTCP 핸드쉐이크

TCP 3-단계 핸드쉐이크에 SSL 핸드쉐이크를 탑재(embedding)시킴으로써 서버 입장에서의 처리 메시지 수 감축과 클라이언트 입장에서의 응답시간 단축을 목표로 하는 새로운 프로토콜을 소개한다.

4.1 SSLmTCP 핸드쉐이크 개요

SSLmTCP 핸드쉐이크의 핵심 아이디어는 TCP 핸드쉐이크와 SSL 핸드쉐이크를 동시에 진행하는 것이다. 그림 3을 보면, 기존 SSL과 TCP 핸드쉐이크 순차 적용을 사용해 통신할 경우, 클라이언트로부터 서버로의 첫 데이터 전송까지 총 6회(ACK와 Client Hello는 묶어서 전송 가능하므로 1회로 간주)의 메시지(세그먼트를 포함) 전송이 필요함을 알 수 있다. 반면 SSLmTCP 핸드쉐이크는 SYN과 SSL의 메시지 1, SYN+ACK와 SSL의 메시지 2, ACK와 SSL의 메시지 3을 동시에 진행함으로써 서버로부터 클라이언트로의 첫 데이터 전송까지 필요한 메시지 전송 횟수를 4회로 줄인다.

그림 4는 헤더(header)를 포함한 TCP 세그먼트의 구성을 나타낸다. TCP 헤더 필드 중 Flags에는 SYN, ACK, URG, PSH, FIN 등의 1비트의 플래그들이 포함되며, 이들의 조합에 따라 세그먼트의 역할이 정해진다. 예를 들어, SYN 세그먼트는 SYN 플래그만 1, 나머지 플래그는 0으로 설정된다. 수신자는 플래그 값을 확인하여 어떤 역할을 하는 세그먼트인지 알게 되고, 현재 수신자의 TCP 상태에 따라 적절하게 대응한다.

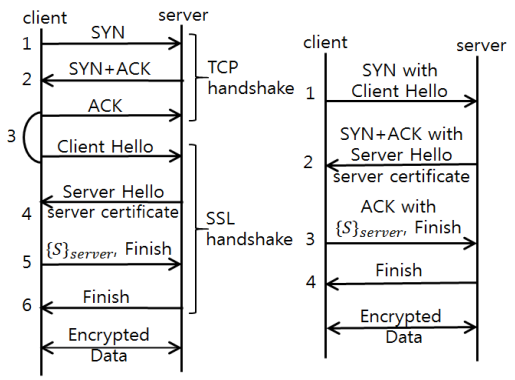

그림 3. SSLmTCP 핸드쉐이크(우측)와 기존 TCP와 SSL 핸드쉐이크 순차 적용의 비교
Fig. 3. Comparing SSLmTCP handshake(right) with the TCP and SSL handshakes in sequence

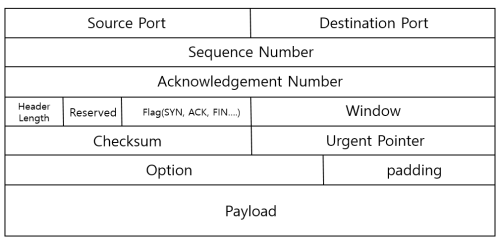

그림 4. TCP 세그먼트
Fig. 4. TCP segment

핸드쉐이크 과정 후 TCP 클라이언트는 ACK 플래그가 지정된 세그먼트에 데이터를 포함시켜 전달할 수 있다. SSLmTCP 핸드쉐이크는 SSL 핸드쉐이크에 필요한 데이터를 전달하기 위해 TCP 세그먼트의 페이로드를 이용한다.

그림 5는 SSLmTCP 핸드쉐이크를 위해 TCP SYN, SYN+ACK, ACK 세그먼트의 페이로드에 각각 어떤 정보가 들어가는지 보여준다. SYN 세그먼트의 페이로드에는 클라이언트의 랜덤 값, Cipher Suites List가 들어간다. Cipher Suites List에는 클라이언트에서 사용 가능한 암호화 알고리즘들 조합의 리스트가 포함된다. SYN+ACK 세그먼트의 페이로드에는 서버의 랜덤 값, Cipher Suites List 중 서버가 선택한 암호화 알고리즘들 조합, 서버의 인증서가 삽입된다. ACK 세그먼트의 페이로드에는 공개키 암호화된 비밀 값과 Change Cipher Spec(이후 협상된 암호화 방식들이 적용됨을 상대방에게 알림)이 삽입된다. 수신자는 삽입된 데이터들을 받아 세션 키들(인증, 기밀성, 무결성 보장을 위한)을 만들고 이들을 데이터 암/복호화에 사용한다.

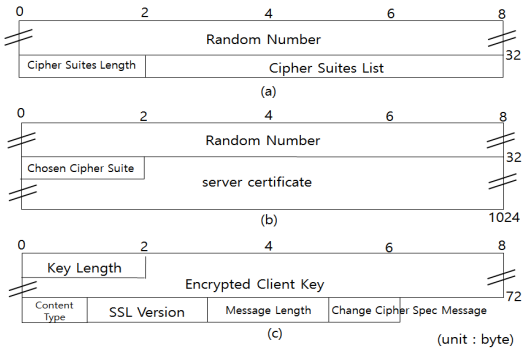

그림 5. (a) SSL 메시지 1을 위한 SYN 페이로드, (b) SSL 메시지 2를 위한 SYN+ACK 페이로드, (c) SSL 메시지 3을 위한 ACK 페이로드
Fig. 5. (a) SYN payload for SSL message 1, (b) SYN + ACK payload for SSL message 2, (c) ACK payload for SSL message 3

4.2 세부 설계

TCP는 상태에 따라 도착하는 세그먼트를 다르게 처리[16]하므로, SSLmTCP 핸드쉐이크는 기존 TCP 관련 커널코드 중, 클라이언트가 SYN-SENT 상태일 때 SYN+ACK 세그먼트를 받는 코드, 서버가 LISTEN 상태일 때 SYN 세그먼트를 받는 코드, 서버가 SYN-RECV 상태일 때 ACK 세그먼트를 받는 코드를 수정한다.

기존 TCP 핸드쉐이크와 SSLmTCP 핸드쉐이크 세

그먼트는 길이에 따라 구분할 수 있다. TCP 핸드쉐이크 세그먼트 크기는 TCP 헤더의 크기인 20바이트이며, 옵션이 추가되는 경우 최대 60바이트가 될 수 있다. SSLmTCP 핸드쉐이크 세그먼트는 기존 TCP 세그먼트에 1024바이트의 페이로드를 붙여 최소 1044바이트부터 최대 1084바이트의 크기를 가진다. 그러므로 TCP 핸드쉐이크와 SSLmTCP 핸드쉐이크 세그먼트는 길이에 따라 구분할 수 있다. 즉, 도착한 TCP 세그먼트의 길이가 1044바이트 이상이면 SSLmTCP 핸드쉐이크로 인식하고, 미만이라면 기존 TCP 핸드쉐이크로 인식할 수 있다. IP의 페이로드에 해당하는 TCP 세그먼트의 길이는 IP 헤더의 Total Length 필드를 통하여 구할 수 있기 때문에, 수신한 메시지의 IP 헤더의 해당 필드에 저장된 값을 확인함으로써 SSLmTCP 핸드쉐이크인지를 쉽게 판별할 수 있다.

SSLmTCP 핸드쉐이크를 적용한 서버 응용 프로그램은 TCP 핸드쉐이크와 SSLmTCP 핸드쉐이크를 따로 구분할 필요 없이 전과 동일한 시스템 콜을 사용하여 TCP 통신 과정을 진행한다. 클라이언트의 TCP 연결 요청 세그먼트가 도착하면 기존 TCP 핸드쉐이크인지 SSLmTCP 핸드쉐이크 인지 구분하여 처리한다. 클라이언트의 TCP 핸드쉐이크는 기존의 connect() 시스템 콜, SSLmTCP 핸드쉐이크는 connect _ssl() 시스템 콜을 각각 이용한다. connect_ssl()은 SSLmTCP 핸드쉐이크를 위해 새롭게 만든 시스템 콜이다. connect_ssl()의 기본 동작은 기존의 TCP와 동일하나, TCP 세그먼트의 페이로드에 SSL 핸드쉐이크를 위한 정보(Client Hello)가 삽입되는 부분이 추가되었다. 아래 표1에서는 TCP 핸드쉐이크와 SSLmTCP 핸드쉐이크의 같은 점 및 주요 차이점을 보여준다.

표 1. TCP 핸드쉐이크와 SSLmTCP 핸드쉐이크의 비교
Table 1. Comparing TCP handshake with SSLmTCP handshake

	TCP handshake	SSLmTCP handshake
role	setup a TCP connection	setup a TCP+SSL connection at once by embedding SSL handshake into TCP handshake
kernel hacking		- distinguish SSLmTCP from TCP handshake - client system call: connect_ssl() - include SSL exchanges in TCP payloads
server system calls	bind(), listen(), accept(), send/recv()	(same)
client system calls	connect(), send()/recv()	connect_ssl(), send()/recv()
header format	standard TCP header format	(same)
TCP payload size	0 byte (no payload)	>= 1024 bytes

4.3 커널 주요 변경 부분 코드

기존의 TCP 핸드쉐이크와 SSLmTCP 핸드쉐이크를 구분하여 위하여 TCP 세그먼트의 길이를 쓰는데, TCP SYN 세그먼트의 구분 처리를 위한 코드 부분은 그림 6과 같다. 다른 세그먼트의 처리 부분도 동일한 로직을 사용한다.

이제 클라이언트가 SSLmTCP 핸드쉐이크를 시작하기 위해 connect_ssl() 시스템 콜을 사용하는데, connect_ssl() 시스템 콜은 기존 connect() 시스템 콜의 모든 과정을 동일하게 진행하되 connect 대신에 sock -> ops -> connect_ssl 함수를 호출한다(그림 7 참조).

이후 기존 TCP 핸드쉐이크와 동일하게 전송 세그먼트를 IP 단계로 전달하는 과정이 진행된다. SSLmTCP

```
switch (sk->sk_state) {
case TCP_CLOSE:
    goto discard;

case TCP_LISTEN:
    ...
    if (th->syn) {
        if (th->fin)
            goto discard;
        if (skb->len >= 1044) {
            if (icsk->icsk_af_ops->conn_request_ssl(sk, skb) < 0)
                return 1;
        }
        else {
            if (icsk->icsk_af_ops->conn_request_ssl(sk, skb) < 0)
                return 1;
        }
    ...
```

그림 6. SYN 세그먼트 처리에서 conn_request_ssl() 와 conn_request() 구분
Fig. 6. Distinguishing conn_request_ssl() from conn_request() in SYN segment processing

```
SYSCALL_DEFINE3(connect_ssl, int, fd,
     struct sockaddr __user *, uservaddr, int, addrlen)
{
    ...
    if (sock->ops->connect_ssl != NULL)
        err = sock->ops->connect_ssl(sock,
                (struct sockaddr *)&address, addrlen,
                    sock->file->f_flags);
    else
        err = sock->ops->connect(sock,
                (struct sockaddr *)&address, addrlen,
                    sock->file->f_flags);
    ...

}
```

그림 7. connect_ssl() 시스템 콜
Fig. 7. connect_ssl() system call

```
skb_put(skb, sizeof(client_hello));

memset(skb->data, 0, sizeof(client_hello));
memcpy(skb->data, client_hello, sizeof(client_hello));
```

그림 8. SYN 페이로드에 CLIENT_HELLO(client_hello) 삽입 코드
Fig. 8. Code for Including CLIENT HELLO(client_hello) in the SYN Payload

를 위해서 세그먼트가 포함된 구조체 skb를 IP 단계로 보내기 전에 그림 8의 코드와 같이 client_hello를 TCP SYN 세그먼트의 페이로드에 삽입한다. TCP SYN-ACK, TCP ACK를 만드는 과정 역시 기존 TCP의 각 세그먼트 생성 과정과 동일하나, skb를 IP 단계로 보내기 전에 client_hello 대신 SSL 핸드쉐이크를 위한 정보들(SYN-ACK 세그먼트에는 server_hello와 server_certificate, ACK 세그먼트에는 pre_master_key)이 각각 삽입된다는 차이가 있다.

V. 실험 및 성능 비교

제안하는 SSLmTCP 핸드쉐이크 프로토콜은 HTTPS를 사용해 브라우저와 통신을 주고받는 웹 사이트에 적용될 경우, TCP 핸드쉐이크와 SSL 핸드쉐이크를 동시에 진행함으로써 교환되는 메시지 개수를 줄일 수 있다. 그 결과로 서버의 메시지 처리 부담이 줄어드는 것은 물론, 기존 방식에 비해 클라이언트에서 서버 응답을 받을 때까지의 대기시간 절감을 가져올 수 있음을 실험결과를 통하여 설명한다.

5.1 환경설정

실험에 사용된 서버, 클라이언트의 컴퓨터 사양은 다음과 같다. 서버머신에는 CPU Intel core i7-2600 3.40GHz, RAM 4GB의 하드웨어에 운영체제 Ubuntu 16.04.1 x86_64, 커널 Linux 4.4.10이 설치되었다. 클라이언트머신에는 CPU Intel core i5-4210U 1.70GHz, RAM 4GB의 하드웨어에 운영체제 Ubuntu 14.04.4 x86_64, 커널 Linux 4.4.10이 설치되었다. Linux 4.4.10 커널을 수정하여 SSLmTCP 핸드쉐이크를 위한 새로운 시스템 콜 connect_ssl()을 추가했고, tcp_transmit_skb, tcp_rcv_state_process 등 TCP 관련 함수를 수정해 tcp_transmit_skb_ssl, tcp_rcv_state_process_ssl 등을 추가로 정의하였다. 이후 새로 정의한 함수들을 사용하여 SSLmTCP 핸드쉐이크 코드를 작성하여 커널에 포함시켰다.

기존의 방법과 SSLmTCP 핸드쉐이크의 성능 차이를 비교하기 위해 서버용 컴퓨터를 특정 위치에 설치해두고, 클라이언트용 컴퓨터의 장소를 옮겨가며 실험하였다. 클라이언트의 각 위치에서 서버와의 연결은 모두 유선 랜을 통해 이루어졌으며, 한 장소에서 걸리는 시간을 10번 씩 측정하여 그 평균치를 도출하였다.

5.2 실험 결과

그림 9는 클라이언트에서 서버로의 핑(ping) 프로그램의 측정 시간(서버-클라이언트 거리에 영향을 받음)에 따른 핸드쉐이크 시간을 비교해서 보여준다. 왼쪽부터 각각의 점은 클라이언트의 위치를 서버와 동일 지점, 6.7km 떨어진 지점, 14km 떨어진 지점, 140km 떨어진 지점, 261km 떨어진 지점에 위치시키고 진행한 실험의 평균 측정치를 나타낸다. X축은 클라이언트가 서버로 핑을 보내어 측정된 시간이며, Y축은 핸드쉐이크 소요시간이다. 핸드쉐이크 소요 시간에는 메시지 송, 수신 시간과 계산과정이 포함되며, 이를 수식으로 나타내면 다음과 같다.

기존 SSL과 TCP 핸드쉐이크 순차 적용

$$X + 6(P + Q) + T \tag{1}$$

SSLmTCP 핸드쉐이크

$$X + 4(P + Q) + T' \tag{2}$$

X : 핸드쉐이크 관련 계산 시간
P : 전파 지연(propagation delay)
Q : 큐잉 지연(queuing delay)
T, T' : 전송 시간(transmission time)

송, 수신 시간은 크게 (전파 지연 + 큐잉 지연 + 전송 시간)으로 구성된다. 이중 전송 시간은 메시지의 크기에 따라 달라진다. 기존 방법과 비교해 SSLmTCP의 핸드쉐이크는 6개의 세그먼트를 4개로 줄여(그림 3 참조), 2번의 TCP 세그먼트 전송을 불필요하게 만든다. 따라서 줄어든 부분의 크기는 TCP 및 IP 헤더만을 포함한 최소 크기의 Ethernet 프레임(frame) 크기인 64바이트의 두 배인 128바이트이고, 이 크기의 데이터 전송 시간 및 이 두 메시지의 전송 시 요구되는 전파 지연과 큐잉 지연의 시간을 포함한 시간이 줄어든다고 볼 수 있다. 전파 지연, 큐잉 지연은 매체의 전송속도, 네트워크의 상태 등에 따라 달라지는데, 핑 프로그램을 통해 측정된 값이 이 둘을 반영한다고 볼 수 있다. 계산 시간에는 공유키를 만들거나, 공개키 암호화 등의 과정이 포함되며 이 중 가장 큰 비중을 차지하는 것이 공개키 암호화 과정이다. 공개키 암호화에는 RSA 4096bit 알고리즘을 사용했으며, 클라이언트에서의 암

호화 과정에 약 10ms, 서버에서의 복호화 과정 약 30ms, 합쳐서 약 40ms가 소요되었다.

클라이언트와 서버 사이가 멀어지거나, 전파 지연이 큰 링크를 통과하는 경우에 메시지 송, 수신 시간이 늘어날 것이다. 따라서 클라이언트와 서버가 같은 위치(동일 Ethernet 스위치를 통해 연결)에 있는 경우의 실험은 기존의 방법과 SSLmTCP 핸드쉐이크의 소요시간이 비슷하지만, 상대적으로 전송 시간이 긴 261km 떨어진 지점에서의 실험에서는 기존의 방법과 SSLmTCP 핸드쉐이크의 소요시간에 큰 차이가 있었다. 서버-클라이언트간 거리는 지도상 직선거리로 측정했으며, 거리가 멀어질수록 핸드쉐이크에 걸리는 시간이 늘어나고, 기존의 방법과 SSLmTCP 핸드쉐이크의 소요시간 차이가 벌어짐을 확인하였다. 한편, 클라이언트가 공개된 WiFi를 통하여 인터넷에 접속되고 서버와 연결되는 실험(클라이언트의 위치는 서버와 16km 떨어진 곳)에서는 실제 데이터 전송을 위한 시간에 비해 네트워크를 발견하고 연결하는 과정까지를 포함한 준비 과정에서 많은 시간이 소모되었다. 그 결과 핑 프로그램의 측정시간이 85.5ms로, 비슷한 거리의 유선 인터넷 연결에 비해 크게 증가했다. 이에 따라 기존 TCP 및 SSL 핸드쉐이크 순차 적용한 결과인 326.6ms에 비해 SSLmTCP 핸드쉐이크를 적용한 결과의 측정치는 236.1ms로, 약 27.7%의 높은 실행시간 감소율을 보였다. 표 2는 서버-클라이언트 사이 핑 프로그램 소요시간에 따른 핸드쉐이크 실행시간 감소율을 보여준다. 기존 TCP및 SSL 핸드쉐이크 순차 적용의 실행시간을 A, SSLmTCP 핸드쉐이크 실행시간을 B라고 할 때, 실행시간 감소율은 $(A-B)/A$의 백분율이다.

이제 제안 방식을 통하여 응답 대기시간이 절감됨을

표 2. 서버-클라이언트 사이 핑 프로그램 소요시간에 따른 실행시간 감소율
Table 2. Reduction rate of measured time according to elapse time by the ping program

Elapsed time by the ping program	0.4	6.0	7.1	8.4	11.6
The TCP and SSL handshakes in sequence	44.3	56.2	66.2	69.1	74.6
SSLmTCP handshake	42.9	52.4	59.6	61.7	64.2
Reduction rate	**3.2%**	**6.8%**	**10.0%**	**10.8%**	**14.0%**

(unit: ms)

보이기 위해 우선 기본적인 HTTPS 요청의 실행 각 단계를 아래 그림 10[17]과 같이 구분하여 살펴본다. 클라이언트 입장에서 요청이 서버로부터 응답받을 때까지의 대기시간은 그림 10의 Receive Time의 직전까지의 시간으로서, DNS Time(name resolution을 위한 시간), Connect Time(TCP 핸드쉐이크 시간), SSL Time(SSL 핸드쉐이크 시간) 및 Wait Time(서버로 GET method의 HTTP 요청을 보내고 기다리는 시간)의 합과 같다. 이러한 대기시간의 구성 요소 중에서 제안하는 SSLmTCP 핸드쉐이크 방식과 TCP 및 SSL 핸드쉐이크의 순차 적용 방식에서 차이가 나는 부분은 Connect Time 및 SSL Time 뿐이고, 나머지 DNS Time과 Wait Time은 방식에 상관없이 같다. 따라서 두 방식의 대기시간의 차이는 핸드쉐이크 소요 시간의 차이와 같게 된다. 제안한 SSLmTCP 핸드쉐이크에서는 그림 10의 Connect(TCP 핸드쉐이크)와 SSL 핸드쉐이크 단계가 합쳐지기 때문에 Connect Time과 SSL Time을 단순 합산한 것보다 적은 시간이 이 부분에서 소요된다. 이에 따라, 그림 9에 서술된 실험 결과를 활용하여, 핑 프로그램 소요시간에 따른 기존 방식 대신 SSLmTCP 핸드쉐이크 방식을 적용했을 때의 대기시간 절감량을 그림 11과 같이 얻을 수 있다.

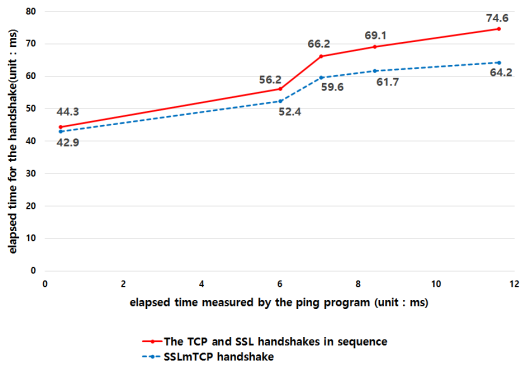

그림 9. 소요시간: SSLmTCP 핸드쉐이크와 기존 TCP와 SSL 핸드쉐이크의 순차 적용 비교
Fig. 9. Elapsed Time: Comparing SSLmTCP Handshake with the TCP and SSL handshakes in sequence

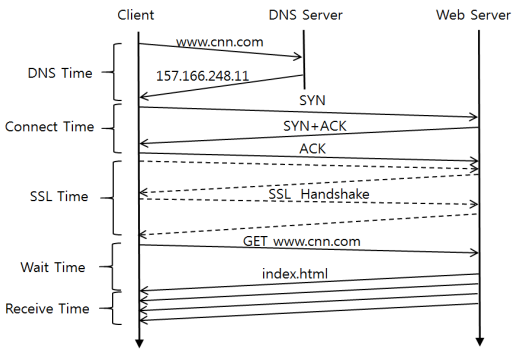

그림 10. 기본 HTTPS 요청 처리 과정 구분 ([17]의 것 다시 그림)
Fig. 10. Basic HTTPS Request Breakdown (adapted from [17])

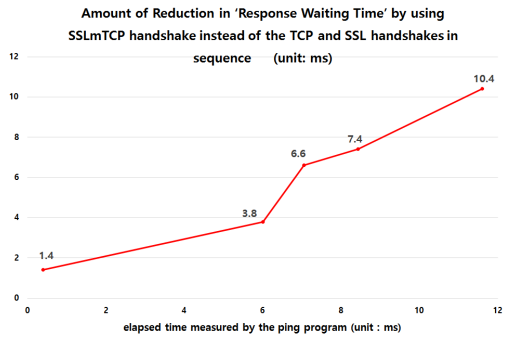

그림 11. TCP와 SSL 핸드쉐이크 순차 적용 대신 SSLmTCP 핸드쉐이크 사용 시의 '응답 대기시간' 절감량
Fig. 11. Amount of Reduction in 'Response Waiting Time' by using SSLmTCP handshake instead of the TCP and SSL handshakes in sequence

5.3 보안성 분석

제안하는 SSLmTCP 핸드쉐이크는 기존의 보안 프로토콜의 취약점을 보완하거나 새로운 보안 기법을 제시하지는 않는다. SSLmTCP 핸드쉐이크는 기존 TCP 핸드쉐이크 과정에 SSL 핸드쉐이크를 삽입함으로써 핸드쉐이크 과정의 시간 절감을 목표로 한다. 제안 방식은 기존 SSL 핸드쉐이크와 동일한 단계를 거치며 동일한 알고리즘을 사용하는 등, SSL 핸드쉐이크 과정이 TCP 핸드쉐이크에 포함되었다는 것을 제외하면, 모든 면에서 SSL 핸드쉐이크를 그대로 흉내 내도록 설계되었다. 따라서 제안하는 SSLmTCP 핸드쉐이크는 보안성의 측면에서 SSL 핸드쉐이크와 동일하다.

Ⅵ. 결 론

대부분의 클라이언트에게 웹 사이트의 반응시간은 서비스 만족도를 결정하는 매우 중요한 요소이다. 만일 HTTP, HTTPS 중 한 가지를 선택할 수 있다면 다수의 클라이언트는 위험에 노출되더라도 대부분 HTTP를 이용한 빠른 통신을 원할 것이다. 이러한 경향은 서버가 HTTPS 사용을 기피하게 되는 요인이 될 수 있으며, 그 결과 사용자의 개인정보 탈취로 이어질 수 있다. 본 논문에서 제시한 SSLmTCP 핸드쉐이크를 이용하면 SSL을 적용한 HTTPS 통신의 응답시간을 줄일 수 있고, 서버 입장에서는 사용자가 느끼는 응답시간이 지연될 것이라는 부담감이 줄어들어 HTTPS 채택에 더욱 적극적이 될 수 있을 것이다. 제안 방식의 실험결과 기존 SSL과 TCP 핸드쉐이크의 순차적용과 비교할 때, 통신 상대방과의 핑 프로그램 측정시간이 0.4ms 일 때 3.2%, 11.6ms일 때 14%의 시간이 줄어드는 것을 보였다. 이를 통해, 제안하는 SSLmTCP 핸드쉐이크를 사용하는 경우 전파지연 및 큐잉지연이 늘어나는 원거리의 TCP 통신에서 더욱 큰 시간 절감을 경험할 수 있을 것이라고 판단한다.

References

[1] E. Rescorla and A.Schiffman, *The Secure HyperText Transfer Protocol*(1999), Retrieved Dec., 20, 2016, from https://tools.ietf.org/html/rfc2660

[2] *HTTPS usage statistics on top websites*(2016), Retrieved Nov., 29, 2016, from statoperator, https://statoperator.com/research/https-usage-statistics-on-top-websites/

[3] Josh Aas, *Enabling HTTP Over SSL*(2016), Retrieved Nov., 29, 2016, from https://letsencrypt.org/2016/06/22/https-progress-june-2016.html

[4] J. Hodges and C. Jackson, *HTTP Strict Transport Security*(2012), Retrieved Dec., 20, 2016, from https://tools.ietf.org/html/rfc6797

[5] T. Socolofsky and C. Kale, *A TCP/IP Tutorial*(1991), Retrieved Dec., 20, 2016, from https://tools.ietf.org/html/rfc1180

[6] W. J. Choi, Ramneek, and W. J. Seok, "Yellow-light TCP: Energy-saving protocol for mobile data transmission," *J. KICS*, vol. 40, no. 03, pp. 478-490, Mar. 2015.

[7] Microsoft, *Explanation of the 3-way Handshake via TCP/IP*, Retrieved Nov., 28, 2016, from https://support.microsoft.com/en-us/kb/172983

[8] A. Freier, P. Karlton, and P. Kocher, *The Secure Sockets Layer*(SSL) Protocol Version 3.0(2001), Retrieved Nov, 21, 2016, from https://tools.ietf.org/html/rfc6101

[9] T. Dierks and C. Allen, *The TLS Protocol Version 1.0(1991)*, Retrieved Nov., 21, 2016, from https://www.ietf.org/rfc/rfc2246.txt

[10] IBM, *Supported SSL and Transport Layer Security protocols*, Retrieved Nov., 26, 2016, from http://www.ibm.com/support/knowledgecenter/ko/ssw_ibm_i_72/rzain/rzainrzaintls.htm

[11] G. T. Park, H. J. Han, and J. H. Lee, "Design

and implementation of lightweight encryption algorithm on OpenSSL," *J. KICS,* vol. 39B, no. 12, pp. 822-830, Dec. 2014.

[12] S. M. Kim, J. S. Park, S. H. Yoon, J. H. Kim, S. O. Choi, and M. S. Kim, "Service identification method for encrypted traffic based on SSL/TLS," *J. KICS,* vol. 40, no. 11, pp. 2160-2168, Nov. 2015.

[13] IBM, *How SSL and TLS provide identification, authentication, confidentiality, and integrity*, Retrieved Nov., 26, 2016, from http://www.ibm.com/support/knowledgecenter/en/SSFKSJ_7.1.0/com.ibm.mq.doc/sy10670_.htm

[14] A. Langley, *Transport Layer Security (TLS) Snap Start*(2010), Retrieved Nov., 26, 2016, from https://tools.ietf.org/html/draft-agl-tls-snapstart-00

[15] E. Stark, LS. Huang, D. Israni, C. Jackson, and D. Boneh, "The case for prefetching and prevalidating TLS server certificate," in *Proc. Netw. and Distrib. Sys. Secur. Symp.* 2012, San Diego, USA, Feb. 2012.

[16] Gordon McKinney, *TCP state Transition Diagram*(2002), Retrieved Dec., 17, 2016, from http://www.cs.northwestern.edu/~agupta/cs340/project2/TCPIP_State_Transition_Diagram.pdf

[17] R. Braud, *Measuring Performance with HTTP Proxies*(2013), Retrieved Feb., 13, 2017, from https://blog.thousandeyes.com/measuring-performance-with-http-proxies

변 기 석 (Ki-Seok Byun)

2015년 2월 : 홍익대학교 컴퓨터공학과 졸업
2017년 2월 : 홍익대학교 컴퓨터공학과 석사
2017년 1월~현재 : 펜타 시큐리티 인턴
<관심분야> 네트워크 보안

박 준 철 (Jun-Cheol Park)

1986년 : 서울대 계산통계학과
1988년 : KAIST 전산학과 석사
1998년 : Univ. of Maryland, College Park, 전산학 박사
현재 : 홍익대학교 컴퓨터공학과 교수
<관심분야> 네트워크/시스템 보안