11. 보안 소켓 프로그래밍

11.1 TLS/SSL 기초 이론

✅ 개요

TLS(Transport Layer Security)와 SSL(Secure Sockets Layer)는 **네트워크 전송 계층 보안을 위한 프로토콜**이다. 주로 HTTPS, SMTPS, FTPS 등에서 사용되며, **기밀성(Confidentiality)**, **무결성(Integrity)**, **인증(Authentication)**을 보장한다.

> SSL은 3.0까지 발표되었고 이후 TLS로 명칭이 변경됨 (실제로는 TLS 1.0 = SSL 3.1)

✅ TLS의 주요 목적

항목	설명
기밀성	암호화를 통해 중간자(MITM) 공격 방지
무결성	메시지 변조 방지를 위한 MAC 또는 HMAC 사용
인증	공개키 기반 인증서 체계 (CA 기반)로 신원 확인

✅ TLS/SSL 프로토콜 구성

TLS는 다음과 같은 **하위 프로토콜**로 구성된다:

1. **Handshake Protocol**
   - 인증서 교환 및 키 협상
2. **Record Protocol**
   - 실제 응용 데이터 암호화 및 전송
3. **Change Cipher Spec Protocol**
   - 새 암호 알고리즘 적용 시점 통지
4. **Alert Protocol**
   - 에러나 경고 상황 알림

✅ TLS Handshake 과정 (요약 흐름)

1. **ClientHello**
   - 클라이언트가 사용할 수 있는 TLS 버전, 암호 알고리즘 목록, 랜덤 데이터 전송
2. **ServerHello**
   - 서버가 TLS 버전과 암호 알고리즘 선택, 인증서(Certificate) 전달
3. **Key Exchange**

    - RSA 또는 Diffie-Hellman 방식으로 대칭키를 안전하게 교환
4. **Session Key 생성**
    - 양측이 공통의 **대칭키**를 만들어 이후 통신에 사용
5. **Finished 메시지 교환**
    - 암호화 통신 시작 전 마지막 무결성 검증

✅ TLS vs SSL 차이점

항목	SSL 3.0	TLS 1.0~1.3
안전성	알려진 취약점 존재	최신 TLS는 보안성 높음
알고리즘	RC4 등 취약한 암호 사용	AES, ChaCha20, HMAC 등
인증서	X.509	동일
지원 여부	대부분 폐기됨	TLS 1.2/1.3만 사용 권장

✅ TLS 1.3 개선점

항목	설명
핸드셰이크 간소화	왕복 횟수 감소 (1-RTT)
암호 스위트 단순화	불필요한 알고리즘 제거
Perfect Forward Secrecy	모든 키 교환에 임시 키 사용 (e.g. ECDHE)

✅ 인증서와 공개키 암호화

- 서버는 **공인 인증기관(CA)**에서 발급한 **X.509 인증서**를 통해 신뢰를 얻음
- 클라이언트는 인증서에 내장된 **공개키**로 서버의 서명 등을 검증함
- 이후 대칭키를 만들어 실제 데이터 전송은 효율적인 **대칭 암호화**로 수행

✅ 실전 예시 (HTTPS)

1. 브라우저가 `https://example.com` 요청
2. 서버는 TLS 인증서와 Handshake 수행
3. 세션 키 생성 → 브라우저와 암호화된 HTTP 데이터 송수신

✅ 주요 사용 포트

프로토콜	포트	설명
HTTPS	443	웹 보안
SMTPS	465	메일 보안
FTPS	990	파일 전송 보안
TLS over TCP	임의의 포트	응용 계층 직접 연동 가능

✅ 정리 요약

- TLS는 SSL의 후속 보안 프로토콜
- 보안성, 성능, 유연성을 모두 고려한 구조
- 실제 통신은 **대칭키**로 수행하며, 키 교환만 **공개키 기반**

11.2 OpenSSL을 이용한 C 언어 기반 SSL 서버/클라이언트 구현

✅ OpenSSL이란?

OpenSSL은 TLS/SSL 프로토콜을 지원하는 **오픈소스 암호화 라이브러리**다.
TLS 핸드셰이크, 대칭 암호화, 인증서 처리 등을 포함한 전방위 네트워크 보안 기능을 제공한다.

✅ 준비 사항

- Linux 환경 (Ubuntu 기준)
- OpenSSL 개발 라이브러리 설치:

```
sudo apt install libssl-dev
```

- 컴파일 시 `-lssl -lcrypto` 옵션 필요

✅ SSL 서버 코드 예제 (단일 접속)

```
#include <openssl/ssl.h>
#include <openssl/err.h>
#include <netinet/in.h>
#include <unistd.h>
#include <string.h>

#define PORT 4433

int main() {
    SSL_library_init();
    SSL_load_error_strings();
```

```
    OpenSSL_add_ssl_algorithms();

    const SSL_METHOD* method = TLS_server_method();
    SSL_CTX* ctx = SSL_CTX_new(method);

    // 인증서와 키 로드
    SSL_CTX_use_certificate_file(ctx, "server.crt", SSL_FILETYPE_PEM);
    SSL_CTX_use_PrivateKey_file(ctx, "server.key", SSL_FILETYPE_PEM);

    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in addr = {0};
    addr.sin_family = AF_INET;
    addr.sin_port = htons(PORT);
    addr.sin_addr.s_addr = INADDR_ANY;
    bind(sockfd, (struct sockaddr*)&addr, sizeof(addr));
    listen(sockfd, 1);

    int client = accept(sockfd, NULL, NULL);
    SSL* ssl = SSL_new(ctx);
    SSL_set_fd(ssl, client);
    SSL_accept(ssl); // 핸드셰이크

    char buffer[1024] = {0};
    SSL_read(ssl, buffer, sizeof(buffer));
    SSL_write(ssl, "Hello Secure World!\n", 21);

    SSL_shutdown(ssl);
    SSL_free(ssl);
    close(client);
    close(sockfd);
    SSL_CTX_free(ctx);
    EVP_cleanup();
}
```

✅ SSL 클라이언트 코드 예제

```
#include <openssl/ssl.h>
#include <openssl/err.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>

#define PORT 4433

int main() {
    SSL_library_init();
    SSL_load_error_strings();
    OpenSSL_add_ssl_algorithms();

    const SSL_METHOD* method = TLS_client_method();
```

```
    SSL_CTX* ctx = SSL_CTX_new(method);

    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in addr = {0};
    addr.sin_family = AF_INET;
    addr.sin_port = htons(PORT);
    inet_pton(AF_INET, "127.0.0.1", &addr.sin_addr);
    connect(sockfd, (struct sockaddr*)&addr, sizeof(addr));

    SSL* ssl = SSL_new(ctx);
    SSL_set_fd(ssl, sockfd);
    SSL_connect(ssl); // 핸드셰이크

    SSL_write(ssl, "GET /secure HTTP/1.1\r\n\r\n", 26);

    char buffer[1024] = {0};
    SSL_read(ssl, buffer, sizeof(buffer));
    printf("Received: %s\n", buffer);

    SSL_shutdown(ssl);
    SSL_free(ssl);
    close(sockfd);
    SSL_CTX_free(ctx);
    EVP_cleanup();
}
```

✅ 인증서 생성 (자체 서명용 테스트용)

```
openssl req -x509 -nodes -days 365 -newkey rsa:2048 \
  -keyout server.key -out server.crt
```

✅ 주요 함수 설명 요약

함수	설명
`SSL_library_init()`	SSL 함수 초기화
`SSL_CTX_new()`	SSL context 생성
`SSL_new()`	소켓에 대한 SSL 객체 생성
`SSL_set_fd()`	소켓과 SSL 연결
`SSL_accept()`	서버 측 TLS 핸드셰이크 수행
`SSL_connect()`	클라이언트 측 핸드셰이크 수행
`SSL_read()/SSL_write()`	암호화된 송수신
`SSL_shutdown()`	SSL 연결 종료

📌 유의 사항

- TLS 연결은 핸드셰이크 과정에서 오랜 시간이 소요될 수 있으며, 서버 인증서에 따라 `verify` 절차도 필요함
- 실전에서는 **인증서 검증(`SSL_CTX_set_verify`)과 CA 체인 등록**도 수행해야 함

11.3 인증서 발급, 키 교환, 암호화 처리

TLS(Transport Layer Security)에서 안전한 통신을 보장하기 위해 필수적으로 수행되는 세 가지 보안 메커니즘은 다음과 같다.

✅ 1. 인증서 발급 (Certificate Issuance)

▷ 인증서란?

- X.509 형식의 **공개키 + 인증 정보**로 구성된 파일
- 클라이언트는 서버 인증서를 통해 신뢰성과 진위를 검증

▷ 발급 방식

1. **자체 서명(Self-Signed)**
    - 테스트 목적 또는 내부 통신에 사용
    - 외부 CA(인증기관)에 의해 신뢰되지 않음

```
openssl req -x509 -nodes -newkey rsa:2048 \
  -keyout server.key -out server.crt -days 365
```

2. **CA 서명(CA-Signed Certificate)**
    - 정식 인증기관(CA)으로부터 발급
    - 브라우저/OS에 내장된 루트 인증서로 검증됨
    - 절차:
        - 개인키 생성 → CSR 생성 → CA 제출 → 인증서 수신

```
# CSR 생성
openssl req -new -key server.key -out server.csr

# 이후 CSR을 CA에 제출해 .crt 파일 수령
```

✅ 2. 키 교환 (Key Exchange)

TLS에서는 초기 통신 중에 **세션 키(session key)**를 안전하게 교환해야 한다. 이를 위해 다음 두 가지 방식을 사용한다.

① RSA 기반 키 교환 (기본적)

- 클라이언트가 서버의 공개키로 세션 키를 암호화해 전송
- 서버는 개인키로 복호화하여 세션 키 확보
- 단점: Perfect Forward Secrecy 미지원

② Diffie-Hellman(Ephemeral DH 포함)

- **서버와 클라이언트가 키 합의 과정을 통해 세션 키 생성**
- ECDHE(Elliptic Curve Diffie-Hellman Ephemeral) 방식이 현대 TLS에서 널리 사용됨
- 장점: **Perfect Forward Secrecy(PFS)** 지원

✅ 3. 암호화 처리 (Encryption)

TLS 세션 키 확보 이후에는 대칭 키 암호화를 통해 본격적인 데이터 전송이 이루어진다.

암호화 항목	설명
암호화 알고리즘	AES, ChaCha20 등
메시지 인증 (MAC)	HMAC-SHA256 등
압축 (선택적)	TLS 1.3에서는 비활성화 추세
암호 스위트	TLS의 알고리즘 구성 집합. 예: `TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384`

📋 인증서 + 암호화 흐름 요약

1. 클라이언트 → 서버: `ClientHello` (지원 암호 스위트 포함)
2. 서버 → 클라이언트: `ServerHello` + 인증서
3. 클라이언트 → 서버: 키 교환 메시지 (ECDHE/RSA)
4. 양측 → 세션 키 생성 완료 후 암호화 시작
5. 클라이언트 ↔ 서버: 대칭 암호화된 응답 주고받기

🔐 실무 포인트

- 서버에서 `SSL_CTX_use_certificate_file()`과 `SSL_CTX_use_PrivateKey_file()`은 반드시 짝을 이뤄야 함
- 암호화 강도는 OpenSSL의 `openssl ciphers` 명령어로 확인 가능
- TLS 1.3에서는 RSA 인증 기반 방식은 줄고, **ECDHE-ECDSA** 기반이 일반화됨

11.4 인증된 HTTPS 서버 구축

HTTPS 서버는 HTTP 위에 **TLS(또는 SSL)** 계층을 추가해 **암호화된 안전한 웹 통신**을 제공한다.
이 절에서는 C 언어 및 OpenSSL을 기반으로 직접 인증서를 활용한 HTTPS 서버를 구현하고 구성하는 방법을 정리한다.

✅ 1. 전체 흐름 개요

```
클라이언트 브라우저
        ⇅ TLS Handshake
    인증서 검증 + 키 교환
        ⇅ HTTPS 암호화 요청/응답
        ⇅ 파일 전송 / API 응답
HTTPS 서버 (OpenSSL + TCP Socket)
```

✅ 2. 준비 사항

📌 OpenSSL 설치 확인

```
openssl version
# OpenSSL 1.1.x or 3.x
```

📌 서버 인증서 및 키 준비

테스트용 Self-Signed 인증서 생성:

```
openssl req -x509 -nodes -days 365 -newkey rsa:2048 \
  -keyout server.key -out server.crt
```

✅ 3. OpenSSL 기반 HTTPS 서버 코드 예제

📂 파일 구성

- `https_server.c`
- `server.crt` (인증서)
- `server.key` (개인키)

📄 `https_server.c` 주요 구조

```
#include <openssl/ssl.h>
#include <openssl/err.h>
#include <netinet/in.h>
#include <unistd.h>
#include <string.h>

#define PORT 4433

void init_openssl() {
    SSL_load_error_strings();
    OpenSSL_add_ssl_algorithms();
}

SSL_CTX *create_context() {
```

```
    const SSL_METHOD *method = TLS_server_method(); // TLS 1.2/1.3
    SSL_CTX *ctx = SSL_CTX_new(method);
    if (!ctx) {
        perror("Unable to create SSL context");
        exit(EXIT_FAILURE);
    }
    return ctx;
}

void configure_context(SSL_CTX *ctx) {
    SSL_CTX_use_certificate_file(ctx, "server.crt", SSL_FILETYPE_PEM);
    SSL_CTX_use_PrivateKey_file(ctx, "server.key", SSL_FILETYPE_PEM);
}

int main() {
    int sockfd;
    struct sockaddr_in addr;

    init_openssl();
    SSL_CTX *ctx = create_context();
    configure_context(ctx);

    // TCP 소켓 생성
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    addr.sin_family = AF_INET;
    addr.sin_port = htons(PORT);
    addr.sin_addr.s_addr = INADDR_ANY;

    bind(sockfd, (struct sockaddr*)&addr, sizeof(addr));
    listen(sockfd, 1);

    while (1) {
        struct sockaddr_in client;
        uint len = sizeof(client);
        int clientfd = accept(sockfd, (struct sockaddr*)&client, &len);

        SSL *ssl = SSL_new(ctx);
        SSL_set_fd(ssl, clientfd);

        if (SSL_accept(ssl) <= 0) {
            ERR_print_errors_fp(stderr);
        } else {
            const char *response =
                "HTTP/1.1 200 OK\r\n"
                "Content-Type: text/plain\r\n\r\n"
                "Hello over HTTPS!\n";
            SSL_write(ssl, response, strlen(response));
        }

        SSL_shutdown(ssl);
        SSL_free(ssl);
        close(clientfd);
    }
```

```
    close(sockfd);
    SSL_CTX_free(ctx);
    EVP_cleanup();
}
```

✅ 4. 클라이언트 접속 테스트

브라우저에서는 자체 서명 인증서를 신뢰하지 않으므로 **curl**을 사용하는 것이 일반적이다.

```
curl -k https://localhost:4433
# -k: 인증서 검증 무시
```

✅ 5. 실무에서의 HTTPS 서버 배포 포인트

항목	설명
인증서 유효성	Let's Encrypt, GlobalSign 등의 CA 인증서 사용
포트 포워딩	리눅스에서 443 → 사용자 프로세스로 전달 시 `iptables` 사용 가능
성능	OpenSSL + `epoll` 또는 `libevent` 구조로 고도화 가능
보안	TLS 1.2 이상만 허용, HTTP Strict Transport Security(HSTS) 설정 필요

✅ 요약

- HTTPS 서버는 TCP + TLS + HTTP 계층으로 구성됨
- OpenSSL API를 통해 C에서도 TLS 기반 서버 구현이 가능
- 인증서는 반드시 TLS 핸드셰이크에서 사용되며, 신뢰 기반 통신 보장