Advanced Computer Networks Lab

Dr Sudipta Saha
Associate Professor
Dept of Computer Science & Engineering
Indian Institute of Technology Bhubaneswar

Lab Tutorial - 2

Teaching Assistants: Subham and Lamia

DSSRG: Decentralized Smart Systems Research Group

https://sites.google.com/iitbbs.ac.in/dssrg
Or Google dssrg iitbbs

From Plain TCP to Secure Communication

- Most internet communication uses TLS
- TCP alone provides no security
- HTTPS, secure chat, email → all rely on TLS
- Goal: understand how TLS fits into socket programming

Where does TLS Fit?

Application (Echo, HTTP, Chat)

↓

TLS / SSL (Security Layer)

↓

TCP (Reliable Transport)

↓

IP (Networking)

TCP Socket Programming

`socket()` → create endpoint

`bind()` → attach address (server)

`listen()` → wait for clients

`accept()` → new client socket

`connect()` → client connection

`send()` / `recv()` → data transfer

Demo Program: TCP Echo Server

Unencrypted Communication

- Server receives data
- Server sends back same data
- Client sees echoed message

Minimal code flow

socket → bind → listen → accept

recv → send

Demo Program: TCP Echo Client

Minimal code flow

socket → connect

send → recv

What's Wrong with Plain TCP?

The Security Problem

- No encryption
- No server authentication
- Vulnerable to sniffing
- Vulnerable to MITM attacks

What Is TLS?

TLS = Transport Layer Security

Provides:

- Encryption
- Integrity
- Authentication

Used in HTTPS, secure chat, VPNs

What TLS Does (and Does Not Do)

TLS Does	TLS Does NOT
Encrypt data	Create sockets
Authenticate server	Replace TCP
Negotiate keys	Understand HTTP

TLS Handshake (Big Picture)

- ClientHello
- ServerHello
- Certificate exchange
- Key exchange
- Secure channel established

Visualizing the TLS Handshake

- Client → ClientHello
- Server → Certificate + ServerHello
- Client → Key Exchange
- Secure communication begins

Why We Need a TLS Library

Why Not Plain C?

- C standard library has **no TLS**

TLS is complex:

- Cryptography
- Certificates
- Key exchange

Solution: **OpenSSL**

How TLS Fits Into Socket Code

socket()

connect()

↓

SSL_new()

SSL_connect()

↓

SSL_write() / SSL_read()

Required OpenSSL Header Files

```
#include <openssl/ssl.h>
#include <openssl/err.h>
```

`ssl.h` → TLS functions

`err.h` → error handling

OpenSSL Initialization Functions

```
int SSL_library_init(void);
```

Purpose: Initialize SSL algorithms

Return: 1 on success

```
void SSL_load_error_strings(void);
```

Purpose: Load readable error messages

```
void OpenSSL_add_all_algorithms(void);
```

Purpose: Load crypto algorithms

SSL Context (SSL_CTX)

Creating a TLS context

```
SSL_CTX *SSL_CTX_new(const SSL_METHOD *method);
```

Arguments

- `TLS_client_method()` or `TLS_server_method()`

Return

- Pointer to `SSL_CTX` or `NULL`

What it creates

- A **TLS configuration context** (TLS rulebook for all connections)

SSL Context (SSL_CTX)

Creating a TLS context

```
SSL_CTX *SSL_CTX_new(const SSL_METHOD *method);
```

What it stores

- TLS versions allowed
- Cipher suites
- Verification rules
- Trust settings

Without this:

- TLS has no policy
- No version or cipher negotiation

Certificate Verification

Enable verification

```
void SSL_CTX_set_verify(SSL_CTX *ctx,
int mode,SSL_verify_cb verify_callback);
```

Purpose

- Controls **certificate verification behavior**

Key argument

```
mode = SSL_VERIFY_PEER
```

Means:

"Verify the server's certificate"

Certificate Verification

Enable verification

```
void SSL_CTX_set_verify(SSL_CTX *ctx,
int mode,SSL_verify_cb verify_callback);
```

Why it matters

If this is NOT set:

- TLS still encrypts
- Server identity is NOT verified
- MITM attacks become possible

Encryption without verification = false security

Certificate Verification

Load trusted CAs

```
int SSL_CTX_set_default_verify_paths(SSL_CTX *ctx);
```

What it does

Loads trusted CA certificates from system locations:

- `/etc/ssl/certs`
- OS trust store

Why needed

During handshake:

- Server sends certificate
- Client must verify CA signature
- This function provides **trusted CA public keys**

Creating an SSL Object

```
SSL *SSL_new(SSL_CTX *ctx);
```

Purpose: Create a TLS session

Return: `SSL*` or `NULL`

Creating an SSL Object

```
SSL *SSL_new(SSL_CTX *ctx);
```

What is an SSL object?

An **SSL object represents ONE TLS CONNECTION**

What SSL stores

- Handshake state
- Session keys
- Cipher in use
- Certificate info

Every client connection needs its own SSL object.

Binding TLS to socket

```
int SSL_set_fd(SSL *ssl, int fd);
```

Purpose

Binds TLS to a TCP socket

Return: 1 on success

What it tells OpenSSL

"Use this socket to send & receive TLS records"

TLS Handshake Function

```
int SSL_connect(SSL *ssl);
```

Purpose

- Performs TLS handshake
- Negotiates cipher
- Verifies certificate

Return

- `>0` → success
- `<=0` → error

Complete Model (Before Handshake)

[TLS POLICY]

```
SSL_CTX_new()
SSL_CTX_set_verify()
SSL_CTX_set_default_verify_paths()
```

[TLS SESSION]

```
SSL_new()
```

[TCP BINDING]

```
SSL_set_fd()
```

[HANDSHAKE]

```
SSL_connect()
```

Encrypted I/O Functions

Write encrypted data

```
int SSL_write(SSL *ssl, const void *buf,
int num);
```

Encrypts + sends data

Returns bytes written

Read decrypted data

```
int SSL_read(SSL *ssl, void *buf, int num);
```

Receives + decrypts data

Returns bytes read

Cleanup Functions

```
int SSL_shutdown(SSL *ssl);
void SSL_free(SSL *ssl);
void SSL_CTX_free(SSL_CTX *ctx);
```

Purpose

- Close TLS session
- Free memory
- Avoid leaks

From Echo Server to Secure Echo Server

Plain TCP	TLS
send()	SSL_write()
recv()	SSL_read()
No security	Encrypted

What Are WebSockets?

- WebSockets are a communication protocol
- Used between browsers and servers
- Enable real-time, two-way communication
- Standard web technology

Why WebSockets Are Needed

- HTTP is request–response based
- Connection closes after each request
- Server cannot push data easily
- Inefficient for real-time applications

How WebSockets Solve This

- Single persistent connection
- Full-duplex communication
- Low latency
- Efficient data transfer

Big-picture: What does server program do?

At a high level, the server program:

1. Creates a TCP server on port 8080
2. Waits for a browser to connect
3. Reads an HTTP WebSocket handshake
4. Sends back a correct WebSocket handshake response
5. Keeps the connection open
6. Receives text messages (commands)
7. Executes predefined actions based on those commands

Key Idea of the WebSocket Handshake

- WebSocket connection **starts as HTTP**
- Uses an **HTTP Upgrade mechanism**
- Server explicitly agrees to protocol change
- After handshake → HTTP is no longer used

High-Level Handshake Flow (TCP View)

- TCP connection is established
- Browser sends HTTP request over TCP
- Server reads HTTP data using `recv()`
- Server sends HTTP response using `send()`
- Same TCP connection continues as WebSocket

Browser Handshake Request (What Server Receives)

- Sent as a normal HTTP GET request
- Contains `Upgrade: websocket`
- Contains `Connection: Upgrade`
- Contains a random `Sec-WebSocket-Key`

Accept-Key Computation (Conceptual)

- Extract `Sec-WebSocket-Key` from request
- Append fixed WebSocket GUID
- Compute SHA-1 hash
- Base64 encode the result
- Send as `Sec-WebSocket-Accept`

Mapping Handshake Steps to C Functions

`recv()` → read HTTP handshake request

`strstr()` → locate WebSocket headers

`sscanf()` → extract client key

`SHA1()` → hash key + GUID

`EVP_EncodeBlock()` → Base64 encoding

`send()` → send `101 Switching Protocols`

Server Handshake Response

- HTTP status: `101 Switching Protocols`
- Confirms protocol upgrade
- Includes computed accept key
- Ends HTTP phase

After the Handshake (Protocol Switch)

- Same TCP connection continues
- No more HTTP headers
- Data sent as WebSocket frames
- Application-level messages exchanged