目录

一、项目概述

1.1 背景

SSL/TSL 已经成为信息安全的重要基础协议，它能提供通信双方保密性、身份论证和消息完整性需求。

1.2 核心功能

本次实验将完成一个可视化软件配置证书密钥、服务端和客户端连接，并通过 wireshark 抓包分析相关协议

1.3 技术栈

前端技术:

1. Vue
2. Element-plus

后端框架+将前后端架构转化成桌面程序

Pywebview

打包成 exe：

Pyinstaller 打包

其他

1. cryptography 库完成证书密钥的生成

2.ssl、socket 建立连接

1.4 安装部署

运行：

Windows 系统点击 exe 文件

源码运行/打包过程：

打开源代码文件夹

激活虚拟环境：venv\Scripts\activate（requirement.txt 注明所需库，所用虚拟环境也打包在 venv 文件夹中）

打包成 exe：pyinstaller main.spec

也可直接运行 start.py

二、需求分析与界面设计

2.1 功能需求

2.1.1 服务端

1. 证书与密钥管理

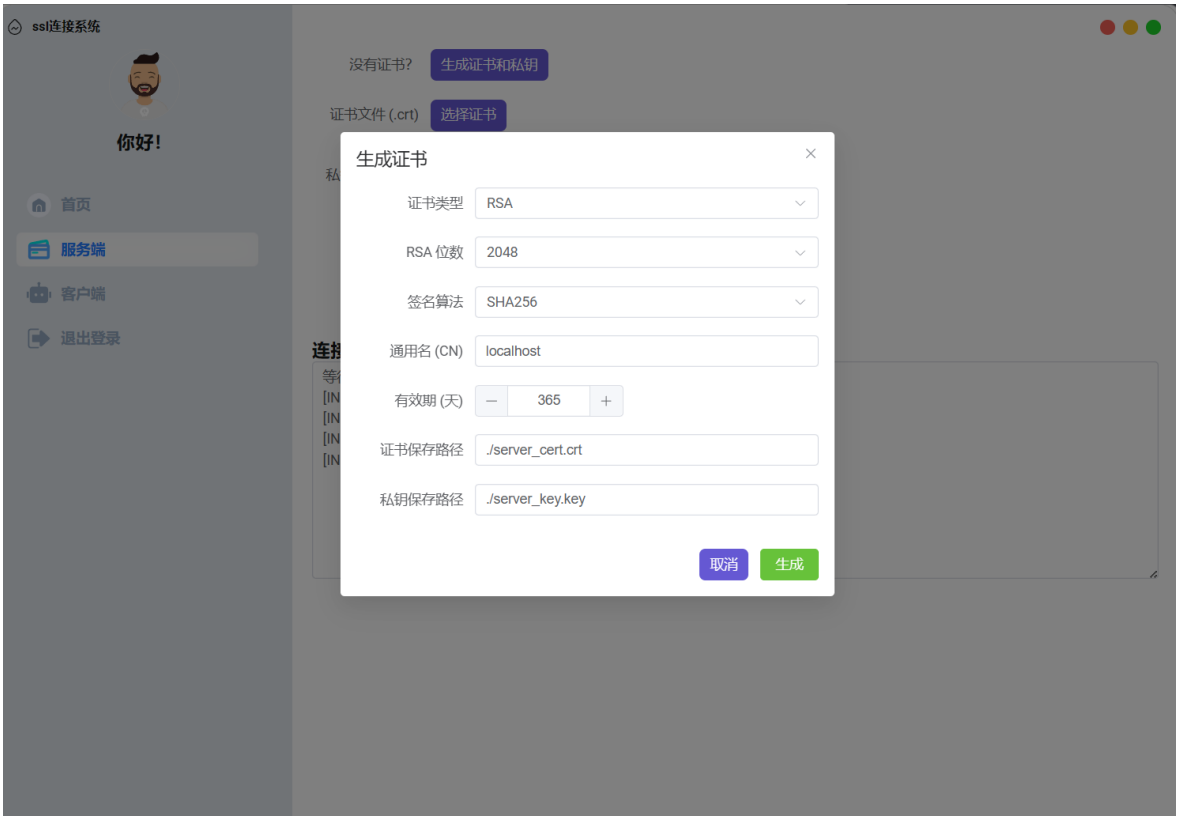
自签名证书生成：支持生成 RSA 或 EC 类型的自签名证书

算法配置：可选择 RSA 密钥长度(2048/3072/4096 位)、EC 曲线类型(secp256r1/secp384r1/secp521r1)、哈希算法(SHA256/SHA384/SHA512)

证书属性配置：支持设置通用名称、有效期(默认 365 天)等证书属性

3. 证书存储：生成的证书和私钥自动保存到指定文件路径

选项获取：提供前端界面需要的下拉菜单选项列表



4. 服务管理

服务启动：配置端口、证书和私钥后启动 TLS 服务

服务停止：安全关闭服务并断开所有客户端连接

5. 运行状态管理：跟踪服务运行状态

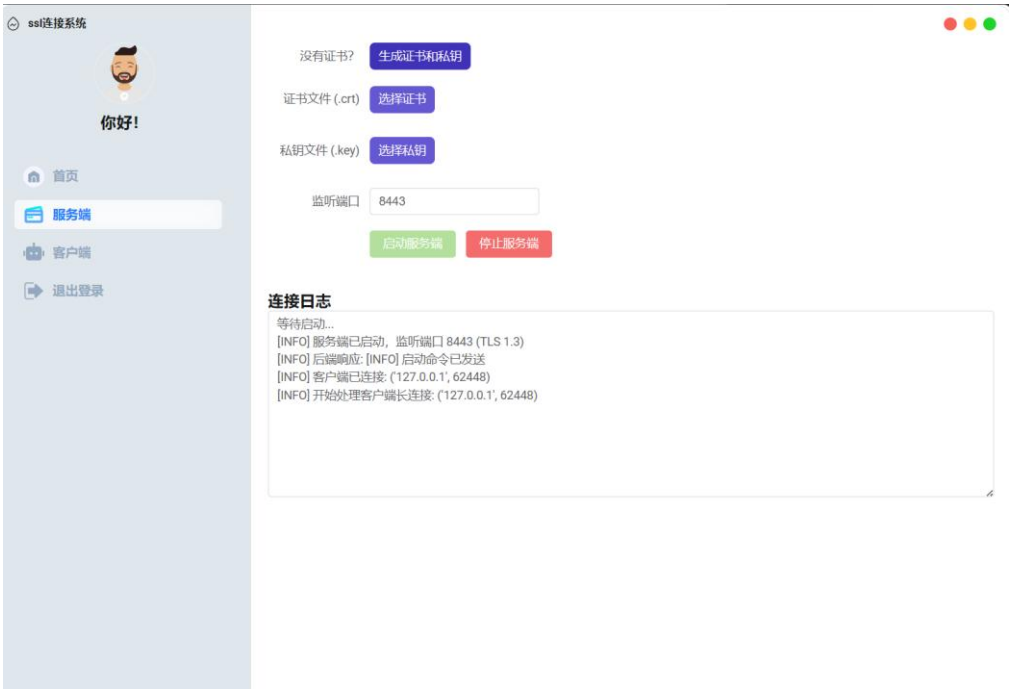
客户端连接处理

长连接支持：支持与多个客户端建立长连接

消息接收与回显：接收客户端消息并进行回显

6. 日志功能

日志记录：记录服务运行状态、连接信息、错误信息等



2.1.2 客户端

1. 连接管理

   TLS 连接建立：连接到指定主机和端口的 TLS 服务端

   连接断开：安全断开与服务端的连接

   连接状态跟踪：维护连接状态标记

2. 证书验证与信息获取

   证书获取：获取服务端提供的证书

   证书解析：解析证书有效期、公钥算法、签名算法等信息

   证书信息展示：将证书信息返回给前端展示

3. 消息通信

   消息发送：向服务端发送文本消息

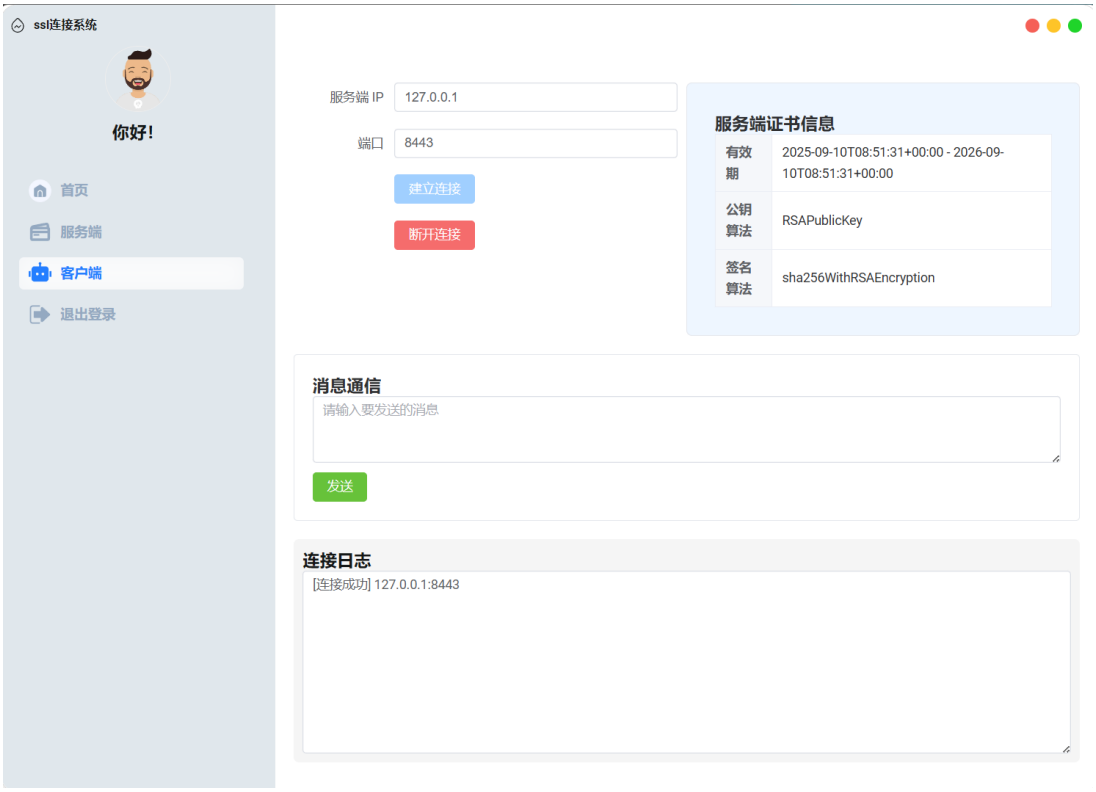
   消息接收：接收服务端回复的消息

4. 日志功能

   日志记录：记录连接状态、发送/接收消息、错误信息等

   日志推送：实时将日志推送到前端界面显示

   日志获取：提供获取完整日志的接口



2.2 非功能需求

• 易用性

提供直观的图形化界面，支持可视化配置、实时监控和信息实时分析，降低运维复杂度。具备详细的日志记录和数据文件生成功能，帮助快速定位问题并进行系统维护。

• 准确性

系统结合先进的机器学习算法，优化检测模型以减少误报和漏报，确保报警信息的准确性和可信度。

• 可扩展性

系统应采用模块化设计，以适应不断变化的网络规模和安全策略。能够自定义 API 接口，便于集成第三方工具或扩展功能，满足未来优化需求。

三、系统设计

3.1 总体设计

本项目按照 MVC 架构，各层如下：

控制器层：serveAPI.py 和 ClientAPI.py 处理路由，调用服务层处理数据实现各个功能

视图层：在 ssl-ui 文件夹中使用 vue 框架构建页面

文件结构如下：

```
Ssl 连接项目/
├── ssl-backend/
│   ├── ClientAPI.py
│   ├── ServerAPI.py
│   ├── requirements.txt
│   ├── start.py
│   └── venv/
└── ssl-ui/
    ├── dist/
    │   ├── …
    │   └── index.html
    ├── index.html
    ├── package-lock.json
    ├── package.json
    ├── src/
    │   ├── App.vue
    │   ├── assets/
    │   ├── components/
    │   ├── main.js
    │   ├── router/
    │   └── views/
    ├── …
```

3.2 作为服务器模块

在此模块，核心功能为生成证书和建立连接

3.2.1 生成证书与密钥

使用 cryptography 库完成证书密钥的生成，并以文件的形式存储，流程如下：

1. 参数解析 ：从传入的 options 字典中提取证书类型、通用名称、有效期、哈希算法等配置参数，若未提供则使用默认值

2.哈希算法选择 ：根据配置选择 SHA256、SHA384 或 SHA512 哈希算法

3.密钥生成 ： 若为 RSA 类型：生成指定密钥长度(2048/3072/4096 位)的 RSA 私钥，若为 EC 类型：根据指定曲线类型(secp256r1/secp384r1/secp521r1)生成 EC 私钥

4. 证书构建 ：

创建证书主体和颁发者信息

设置证书序列号、生效时间和过期时间

5.使用选定的哈希算法和生成的私钥对证书进行签名

6. 证书和私钥存储 ：将生成的证书和私钥以保存到指定路径

7.日志记录与结果返回 ：记录证书生成过程的关键信息，并返回包含证书路径、私钥路径和成功消息的结果

代码：

```
def generate_certificate(self, options: dict):
        cert_type = options.get("cert_type", "RSA")
        common_name = options.get("common_name", "localhost")
        valid_days = int(options.get("valid_days", 365))
        hash_alg = options.get("hash_alg", "SHA256")

        # ---- 选择哈希算法 ----
        hash_algo_map = {
            "SHA256": hashes.SHA256(),
            "SHA384": hashes.SHA384(),
            "SHA512": hashes.SHA512(),
        }
        hash_algo = hash_algo_map.get(hash_alg, hashes.SHA256())

        # ---- 生成密钥 ----
        if cert_type == "RSA":
            key = rsa.generate_private_key(
                public_exponent=65537,
                key_size=int(options.get("rsa_bits", 2048))
            )
        elif cert_type == "EC":
            curve_name = options.get("curve", "secp256r1")
            curve_map = {
                "secp256r1": ec.SECP256R1(),
                "secp384r1": ec.SECP384R1(),
                "secp521r1": ec.SECP521R1(),
```

```
            }
            key = ec.generate_private_key(
                curve_map.get(curve_name, ec.SECP256R1())
            )
        else:
            raise ValueError("Unsupported cert_type")

        # ---- 构建证书 ----
        subject = issuer = x509.Name([
            x509.NameAttribute(NameOID.COUNTRY_NAME, u"CN"),
            x509.NameAttribute(NameOID.STATE_OR_PROVINCE_NAME, u"Shanghai"),
            x509.NameAttribute(NameOID.LOCALITY_NAME, u"Shanghai"),
            x509.NameAttribute(NameOID.ORGANIZATION_NAME, u"MyOrg"),
            x509.NameAttribute(NameOID.COMMON_NAME, common_name),
        ])

        cert = (
            x509.CertificateBuilder()
            .subject_name(subject)
            .issuer_name(issuer)
            .public_key(key.public_key())
            .serial_number(x509.random_serial_number())
            .not_valid_before(datetime.datetime.utcnow())
            .not_valid_after(datetime.datetime.utcnow() +
datetime.timedelta(days=valid_days))
            .add_extension(
                x509.SubjectAlternativeName([x509.DNSName(common_name)]),
                critical=False,
            )
            .sign(private_key=key, algorithm=hash_algo)
        )
```

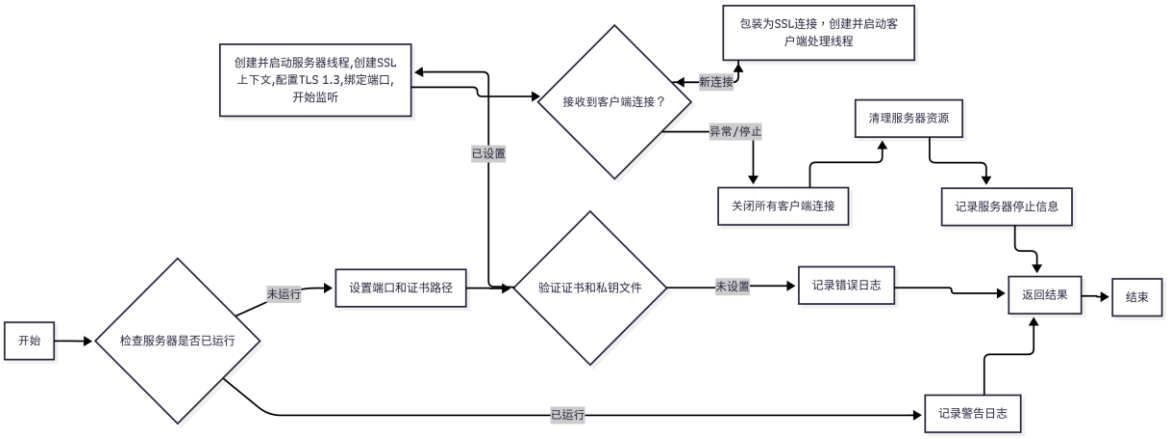
3.1.2 建立长连接

服务端首先要做好准备开启监听，主要流程如下：

- 前置检查：
  - 检查服务器是否已在运行，若已运行则返回警告信息

- 设置端口号、证书文件路径和私钥文件路径
- 验证证书和私钥文件是否已配置，若未配置则返回错误信息

- 服务器启动：
  - 创建独立的服务器线程运行 run_server 函数
  - 在 run_server 函数中创建 SSL 上下文，配置为强制使用 TLS 1.3 协议
  - 加载证书链（证书和私钥）
  - 创建 TCP 监听套接字，设置地址复用选项，绑定到指定端口并开始监听
- 连接处理：
  - 主循环持续接受新的客户端连接
  - 每个新连接被包装为 SSL 连接后添加到客户端集合中
  - 为每个客户端连接创建独立的处理线程，执行 handle_client 函数
- 服务器关闭：
  - 当主循环退出时，关闭所有活跃的客户端连接
  - 清理服务器资源，关闭监听套接字
  - 记录服务器停止信息



代码如下：

```
def run_server():
            try:
                context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
                context.minimum_version = ssl.TLSVersion.TLSv1_3
                 # 强制 TLS1.3
                context.load_cert_chain(certfile=self.cert_file,
keyfile=self.key_file)
                with socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0) as
sock:
                    # 保存原始监听 socket，便于外部 stop 时关闭
```

```
                    self._server_socket = sock
                    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
                    sock.bind(("0.0.0.0", self.port))
                    sock.listen(5)
                    self.running = True

                    while self.running:
                        try:
                            conn, addr = sock.accept()  # 普通 TCP 连接
                                                        #  wrap 成 SSL socket
                            ssl_conn = context.wrap_socket(conn,
server_side=True)

                            self._clients.add(ssl_conn)
                            t = threading.Thread(
                                target=handle_client, args=(ssl_conn, addr),
daemon=True
                            )
                            t.start()

            finally:
                # 主循环退出后，关闭所有客户端连接
                for c in list(self._clients):
                    try:
                        c.shutdown(socket.SHUT_RDWR)
                    try:
                        c.close()
                self._clients.clear()
                # 清理并置空监听对象引用
                try:
                    if self._wrapped_socket:
                        self._wrapped_socket.close()

                try:
                    if self._server_socket:
                        self._server_socket.close()
```

```
            self._wrapped_socket = None
            self._server_socket = None
            self.running = False
```

当监听到客户端连接时，处理流程如下：

- handle_client 函数在独立线程中处理与单个客户端的通信
- 设置短超时以便及时响应服务器关闭信号
- 循环接收客户端发送的数据，解码后记录日志并发送回显消息
- 处理各类异常情况，包括连接超时、网络错误等

```
def handle_client(conn, addr):
        """每个客户端的长连接处理器"""
        try:
            # 设置短超时以便能及时响应服务端关闭
            conn.settimeout(1.0)
            while self.running:
                try:
                    data = conn.recv(4096)
                    if not data:
                        # 客户端关闭连接
                        break
                    try:
                        text = data.decode(errors="ignore")
                    except Exception:
                        text = repr(data)
                    self._log(f"[RECV] {addr} -> {text}")
                    # 简单回显或心跳响应
                    try:
                        conn.sendall(f"服务端收到: {text}".encode("utf-8"))
                    except Exception as e:
                        self._log(f"[ERROR] 发送到客户端失败 {addr}: {e}")
                        break
                except socket.timeout:
                    # 超时循环，继续检查 self.running
                    continue
                except OSError:
```

```
                break
        finally:
            try:
                conn.shutdown(socket.SHUT_RDWR)
            except Exception:
                pass
            try:
                conn.close()
            except Exception:
                pass
            # 从跟踪集合移除
            try:
                self._clients.discard(conn)
            except Exception:
                pass
            self._log(f"[INFO] 客户端已断开: {addr}")
```

3.3 作为客户端模块

在这个模块， 主要职责为与服务端建立连接返回服务端证书信息并向服务端发送消息

3.3.1 建立连接

- 参数解析 ：从传入的 config 字典中提取主机地址和端口号
- SSL 上下文创建与配置 ：
  - 创建默认的 SSL 上下文
  - 禁用主机名检查（ check_hostname = False ）
  - 禁用证书验证（ verify_mode = ssl.CERT_NONE ） 注意：代码中有注释掉的默认严格验证选项，这表明函数设计上预留了严格验证的能力
- 建立连接 ：
  - 首先创建普通的 TCP socket 连接到指定的主机和端口
  - 使用 SSL 上下文将普通 socket 包装为 SSL socket，并指定服务器主机名以支持 SNI (Server Name Indication)
  - 证书获取与解析 ：
- 获取服务端的二进制证书数据
  - 使用 cryptography 库解析 X.509 证书
  - 提取证书有效期、公钥算法和签名算法等关键信息
  - 构建包含证书信息的字典对象
- 状态更新与监听线程启动 ：

- 将连接状态标记为已连接
- 创建并启动独立的守护线程，用于持续监听来自服务器的消息

- 异常处理：
  - 捕获连接过程中可能发生的各类异常
  - 记录错误日志并向上层抛出异常

```
def client_connect(self, config):
        """
        建立 TLS 连接并返回服务端证书信息
        """
        host = config["host"]
        port = int(config["port"])

        try:
            context = ssl.create_default_context()
            context.check_hostname = False
            context.verify_mode = ssl.CERT_NONE
            raw_sock = socket.create_connection((host, port))
            self.ssl_sock = context.wrap_socket(raw_sock,
server_hostname=host)

            # 获取服务端证书
            cert_bin = self.ssl_sock.getpeercert(True)
            cert = x509.load_der_x509_certificate(cert_bin,
default_backend())
            not_before = getattr(cert, "not_valid_before_utc",
cert.not_valid_before)
            not_after = getattr(cert, "not_valid_after_utc",
cert.not_valid_after)
            cert_info = {
                "validity": f"{not_before.isoformat()} -
{not_after.isoformat()}",
                "publicKeyAlgo": cert.public_key().__class__.__name__,
               "signatureAlgo": getattr(cert.signature_algorithm_oid,
"_name", str(cert.signature_algorithm_oid)),
            }
```

```
            self.connected = True
            self.recv_thread = threading.Thread(target=self._recv_loop,
daemon=True)
            self.recv_thread.start()
            return cert_info
```

创建连接后可用 socket 库中的 sendall(message.encode("utf-8"))函数发送消息给服务端。

四、抓包验证

4.1 建立连接过程概述：

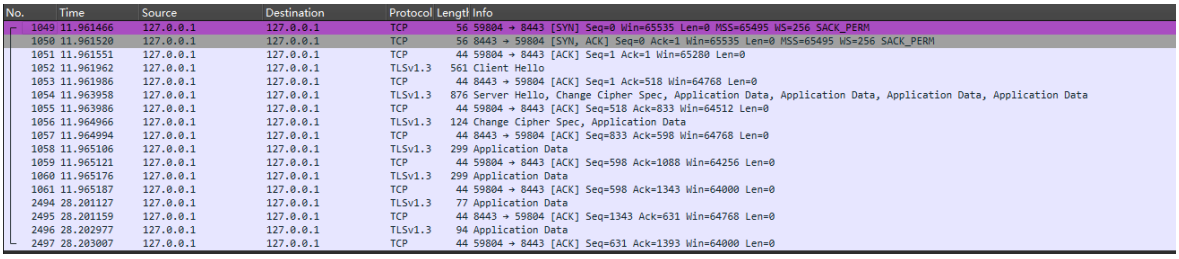
服务端配置：按照课件要求配置如下证书：

服务端在端口 8443 开始运行：

客户端连接：

可以看到证书信息为 P-256 椭圆曲线

使用 wires hark 抓包验证如下：



No.	Time	Source	Destination	Protocol	Lengtl	Info
1049	11.961466	127.0.0.1	127.0.0.1	TCP	56	59804 → 8443 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
1050	11.961520	127.0.0.1	127.0.0.1	TCP	56	8443 → 59804 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
1051	11.961551	127.0.0.1	127.0.0.1	TCP	44	59804 → 8443 [ACK] Seq=1 Ack=1 Win=65280 Len=0
1052	11.961962	127.0.0.1	127.0.0.1	TLSv1.3	561	Client Hello
1053	11.961986	127.0.0.1	127.0.0.1	TCP	44	8443 → 59804 [ACK] Seq=1 Ack=518 Win=64768 Len=0
1054	11.963958	127.0.0.1	127.0.0.1	TLSv1.3	876	Server Hello, Change Cipher Spec, Application Data, Application Data, Application Data, Application Data
1055	11.963986	127.0.0.1	127.0.0.1	TCP	44	59804 → 8443 [ACK] Seq=518 Ack=833 Win=64512 Len=0
1056	11.964966	127.0.0.1	127.0.0.1	TLSv1.3	124	Change Cipher Spec, Application Data
1057	11.964994	127.0.0.1	127.0.0.1	TCP	44	8443 → 59804 [ACK] Seq=833 Ack=598 Win=64768 Len=0
1058	11.965106	127.0.0.1	127.0.0.1	TLSv1.3	299	Application Data
1059	11.965121	127.0.0.1	127.0.0.1	TCP	44	59804 → 8443 [ACK] Seq=598 Ack=1088 Win=64256 Len=0
1060	11.965176	127.0.0.1	127.0.0.1	TLSv1.3	299	Application Data
1061	11.965187	127.0.0.1	127.0.0.1	TCP	44	59804 → 8443 [ACK] Seq=598 Ack=1343 Win=64000 Len=0
2494	28.201127	127.0.0.1	127.0.0.1	TLSv1.3	77	Application Data
2495	28.201159	127.0.0.1	127.0.0.1	TCP	44	8443 → 59804 [ACK] Seq=1343 Ack=631 Win=64768 Len=0
2496	28.202977	127.0.0.1	127.0.0.1	TLSv1.3	94	Application Data
2497	28.203007	127.0.0.1	127.0.0.1	TCP	44	59804 → 8443 [ACK] Seq=631 Ack=1393 Win=64000 Len=0

上图显示 127.0.0.1:59804 → 127.0.0.1:8443 的 TLS 1.3 握手过程，具体分析如下：

TCP 握手阶段

1. 1049 (SYN)

客户端 → 服务端

发起 TCP 连接请求，指定源端口 59804，目标端口 8443。

2. 1050 (SYN, ACK)

服务端 → 客户端

确认收到 SYN，并同步建立连接。

3. 1051 (ACK)

客户端 → 服务端

确认连接建立完成。

TCP 三次握手完成，接下来进入 TLS 握手。

4. 1052 (Client Hello)

客户端 → 服务端

客户端发起 TLS 1.3 握手，提供：

支持的加密套件 (Cipher Suites)

支持的 TLS 版本 (supported_versions，包括 TLS 1.3)

椭圆曲线组 (supported_groups)

密钥交换公钥 (key_share)

签名算法 (signature_algorithms)

5. 1054 (Server Hello, Change Cipher Spec, Encrypted Extensions/Cert 等)

服务端 → 客户端

服务端选择加密参数并回应：

确认使用 TLS 1.3

确认使用的加密套件

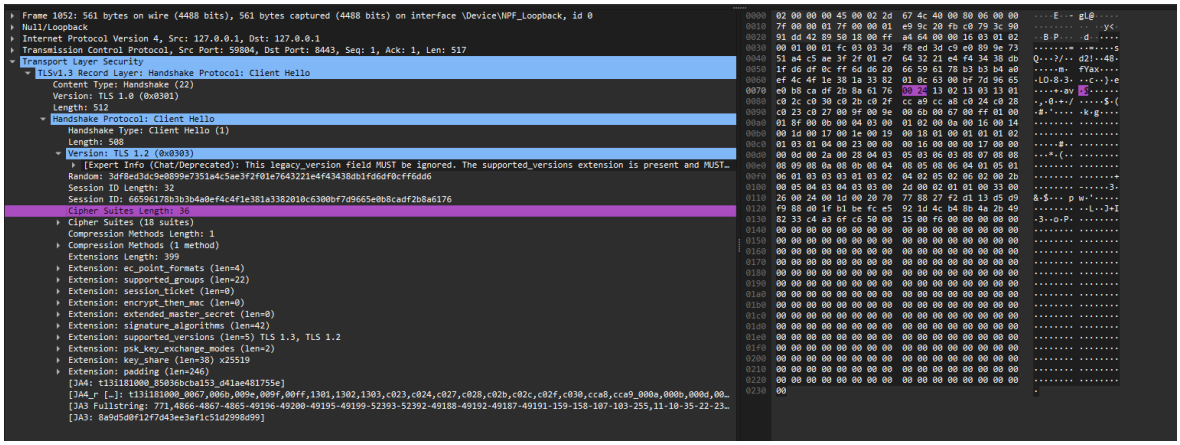
服务端的密钥交换公钥

后续的 Encrypted Extensions、Certificate、Certificate Verify、Finished 已经被加密，Wireshark 中显示为 Application Data。（包括模拟客户端向服务端发送消息）

4.2 建立连接过程具体数据包

4.2.1 client-hello 数据包

1052 数据包如下：

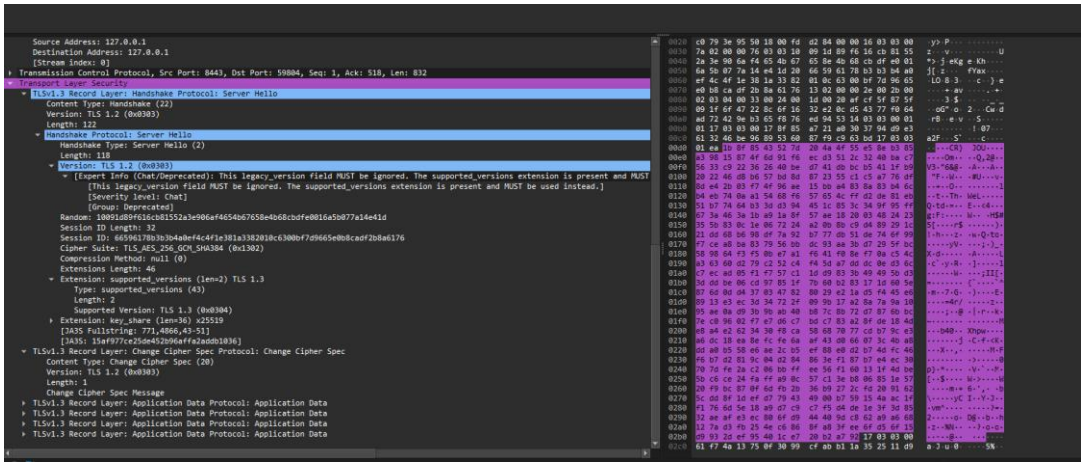


层级	字段	值/说明	字段含义
TLS Record Layer	Content Type	Handshake (22)	表示这是 TLS 握手消息
	Version	TLS 1.0 (0x0301)	记录层版本，占位用，TLS 1.3 并不依赖此字段
	Length	512	后续握手消息的总长度
Handshake Protocol	Handshake Type	Client Hello (1)	握手消息类型，1 表示客户端问候
	Length	508	ClientHello 数据长度
	Version	TLS 1.2 (0x0303)	legacy_version，占位字段；实际用 supported_versions 确认版本
	Random	3df8ed3d...cff6dd6	客户端随机数（32 字节），用于密钥派生
	Session ID Length	32	会话 ID 长度
	Session ID	66596178...a6176	会话 ID，用于恢复会话或新建会话
	Cipher Suites Length	36	加密套件列表长度
Extensions	Cipher Suites	18 个套件	客户端支持的对称加密/认证算法组合

	Compression Methods Length	1	压缩方法列表长度
	Compression Methods	null (0)	不使用压缩（TLS 1.3 必须为 null）
	Extensions Length	399	扩展字段总长度
	ec_point_formats	长度 = 4	支持的椭圆曲线点格式
	supported_groups	长度 = 22	支持的椭圆曲线群（如 x25519, secp256r1）
	session_ticket	长度 = 0	预留字段，用于会话恢复
	encrypt_then_mac	长度 = 0	旧式扩展（TLS 1.3 中不用）
	extended_master_secret	长度 = 0	TLS 1.2 扩展，TLS 1.3 已被替代
	signature_algorithms	长度 = 42	客户端支持的签名算法
	supported_versions	长度 = 5 (TLS 1.3, TLS 1.2)	声明支持的协议版本
	psk_key_exchange_modes	长度 = 2	PSK (预共享密钥) 交换模式
	key_share	长度 = 38，x25519	密钥交换用的公钥参数（DH/ECDHE）
	padding	长度 = 246	填充，用于防止指纹识别
JA3 指纹	JA4	t13i181000_85036bcba153_d41ae481755e	TLS 指纹（扩展版 JA3）
	JA4_r	t13i181000_0067,006b,...	详细指纹，列出 cipher suites 与扩展
	JA3 Fullstring	771,4866-4867-4865-...	ClientHello 的完整特征字符串
	JA3 Hash	8a9d5d0f12f7d43ee3af1c51d2998d99	指纹哈希，用于识别客户端类型

4.2.2 server-hello 数据包

1054 数据包如下：



层级	字段	值/说明	字段含义
TLS Record Layer	Content Type	Handshake (22)	表示这是握手消息
	Version	TLS 1.2 (0x0303)	占位字段（legacy_version），实际版本用扩展字段
	Length	122	握手消息总长度
Handshake Protocol	Handshake Type	Server Hello (2)	握手消息类型：服务端响应
	Length	118	ServerHello 长度
	Version	TLS 1.2 (0x0303)	占位字段，忽略；真实版本在 supported_versions 扩展
	Random	10091d89...14e41d	服务端随机数（32 字节），与客户端随机数一起生成主密钥
	Session ID Length	32	会话 ID 长度
	Session ID	66596178...a6176	回显客户端 Session ID，用于标识会话
	Cipher Suite	TLS_AES_256_GCM_SHA384 (0x1302)	服务端选择的加密套件
	Compression Method	null (0)	必须为 null（TLS 1.3 不支持压缩）
	Extensions Length	46	扩展字段长度
Extensions	supported_versions	长度 = 2 → TLS 1.3 (0x0304)	明确确认使用 TLS 1.3
	key_share	长度 = 36，群组 = x25519，公钥 = afcf5f87...6ed9453	服务端的密钥交换参数（ECDHE 公钥）
JA3S 指纹	JA3S Fullstring	771,4866,43-51	基于 ServerHello 的特征字符串
	JA3S Hash	15af977ce25de452b96affa2addb1036	指纹哈希，用于识别服务端
Change Cipher Spec	Content Type	Change Cipher Spec (20)	兼容 TLS 1.2 的占位消息，TLS 1.3 实际不使用
	Version	TLS 1.2 (0x0303)	legacy 占位
	Length	1	固定长度消息
Application Data (加密)	Opaque Type	Application Data (23)	握手后续消息（加密状态）
	Lengths	23 / 490 / 97 / 69	每个 TLS record 的长度
	Encrypted Application Data	8f85a721a0..., 1b8f85..., …	加密的握手消息（Encrypted Extensions, Certificate, Certificate Verify, Finished）和可能的早期应用数据

五、问题思考

5.1 TLS1.3 比 TLS1.2 更加安全的原因

握手阶段对比：

TLS1.2	TLS1.3
1.ClientHello 客户端支持的版本、密码套件、随机数、扩展等。 2.ServerHello 服务端选择的版本、密码套件，返回随机数。 3.Certificate 服务器证书链。 4.ServerKeyExchange（可选） 包含临时密钥（如 ECDHE 参数）。 5.ServerHelloDone 6.ClientKeyExchange 客户端发出密钥交换数据。 7.ChangeCipherSpec（客户端 → 服务端） 通知**从此之后开始使用加密通信**。 8.Finished（加密发送） 9.ChangeCipherSpec + Finished（服务端 → 客户端）	1.ClientHello 包含支持的版本、扩展、密钥交换参数（ECDHE）、支持的加密套件。 2.ServerHello 确定密码套件，返回密钥交换参数。 **从此开始后续所有消息均加密传输。** 3.EncryptedExtensions（加密） 服务端的一些配置扩展。 4. Certificate（加密） 服务器证书链。 5.CertificateVerify（加密） 服务器证明它拥有私钥。 6. Finished（加密） 客户端 Finished（加密）

从握手过程来看：

TLS1.3 握手阶段更少，因而时间更少从而更安全，TLS1.2 握手至少需要 2-RTT。TLS1.3 优化为 1-RTT（甚至支持 0-RTT 快速恢复连接）

握手加密更彻底：TLS1.2 的大部分握手（证书、密钥交换）都是明文。TLS1.3 在 ServerHello 之后就进入加密状态，防止中间人窃听元数据。

其他因素：

- 算法更强，弱算法移除
  TLS1.2 允许 RC4、3DES、SHA-1 等弱算法。
  TLS1.3 强制使用 AEAD 加密（如 AES-GCM、ChaCha20-Poly1305）和 SHA-256/384。
  明确禁止不安全算法。
- 简化协议，减少攻击面
  TLS1.2 有十几种握手消息和复杂的协商逻辑。
  TLS1.3 移除 ChangeCipherSpec、Renegotiation 等容易出漏洞的机制。