山东大学 软件 学院

# 安全协议与标准 课程实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：202200201095 | 姓名：杨伟康 | | 班级：22网安 |
| 实验编号：2024-2025-2 | | | |
| 实验题目：安全协议与标准课程实验 | | | |
| 实验学时：16 | | 实验日期： 2025.4.13-5.24 | |

## 实验目的与内容：

|  |
| --- |
| **1. 在 windows 下（使用 visual studio 或其他）或 LINUX 下编译最新的 OPENSSL 密 码安全库（https://www.openssl.org， openssl-1.1.1），生成静态库或动态库。**  **2. 使用 openssl 工具完成如下操作：**  **（1）RSA 2048 位 密钥生成；**  **（2）导出公钥；**  **（3）生成数字证书请求；**  **（4）生成数字证书；**  **（5）生成 pkcs12 格式基于口令保护的标准格式数据包。**  **（6）用 ASN1 编码解析器，显示以上生成数据对象的 ASN1 编码数据。**  **3. 调用函数库，编写代码实现以上工具完成的功能（1-6）。**  **4. 参考分析 openssl 工具包原码的 s\_cilent 和 s\_server 代码部分，自己编写一个 完整的 SSL 客户端和服务器端程序，并能安全通信（配置数字证书），显示执行过程。**  **5.撰写实验报告，要求将以上过程描述并附上源代码（\*.h, \*.c, \*.cpp）, 总结编 程实验心得。** |

## 开发环境：

|  |
| --- |
| 硬件环境： |
| 软件开发环境：  Visual studio2022+OPENSSL3.3.2 |

## 文件提交说明

|  |
| --- |
| 包括**三个**源码文件：generateRSA.cpp实现了使用OPENSSL库实现OPENSSL的部分功能，根据不同命令来调用不同的函数完成功能。  功能如下：  （1）RSA 2048 位 密钥生成；  （2）导出公钥；  （3）生成数字证书请求；  （4）生成数字证书；  （5）生成 pkcs12 格式基于口令保护的标准格式数据包。  （6）用 ASN1 编码解析器，显示以上生成数据对象的 ASN1 编码数据。 |
| Server.cpp和client.cpp分别实现了一个 完整的 SSL 客户端和服务器端程序，并能安全通信（配置数字证书），能够显示执行过程，对大部分关键步骤进行打印输出（参考OPENSSL的s\_cilent 和 s\_server 代码部分）。 |

## 目录：

目录

[安全协议与标准 课程实验报告 1](#_Toc198998985)

[实验目的与内容： 1](#_Toc198998986)

[开发环境： 2](#_Toc198998987)

[文件提交说明 2](#_Toc198998988)

[目录： 2](#_Toc198998989)

[实验具体开发步骤： 3](#_Toc198998990)

[1.在 windows 下（使用 visual studio 或其他）或 LINUX 下编译最新的OPENSSL 密码安全库（https://www.openssl.org， openssl-1.1.1），生成静态库或动态库。 3](#_Toc198998991)

[2. 使用 openssl 工具完成如下操作： 3](#_Toc198998992)

[3. 调用函数库，编写代码实现以上工具完成的功能（1-6）。 9](#_Toc198998993)

[RSA 2048 位 密钥生成 9](#_Toc198998994)

[导出公钥 10](#_Toc198998995)

[生成数字证书请求: 11](#_Toc198998996)

[生成数字证书： 15](#_Toc198998997)

[生成 pkcs12 格式基于口令保护的标准格式数据包。 18](#_Toc198998998)

[简单实现asn1parse的解析器 24](#_Toc198998999)

[4. 参考分析 openssl 工具包原码的 s\_cilent 和 s\_server 代码部分，自己编写一个完整的 SSL 客户端和服务器端程序，并能安全通信（配置数字证书），显示执行过程。 35](#_Toc198999000)

[实现基础的TCP通信（明文传输）​ 37](#_Toc198999001)

[添加OpenSSL安全层 39](#_Toc198999002)

[​客户端验证服务器证书 44](#_Toc198999003)

[服务端验证客户端证书（最终实现双向认证） 48](#_Toc198999004)

[显示详细的执行过程 50](#_Toc198999005)

[编程实验心得 54](#_Toc198999006)

[**1. ​理论与实践的结合​** 54](#_Toc198999007)

[**2. ​工具与代码的双重掌握​** 54](#_Toc198999008)

[**3. ​调试与问题解决能力提升​** 55](#_Toc198999009)

[**4. ​安全通信的实践意义​** 55](#_Toc198999010)

[**5. ​对密码学工程的认知​** 55](#_Toc198999011)

[**6. ​未来学习方向​** 55](#_Toc198999012)

|  |
| --- |
| 温馨提示：可以使用ctrl+f来打开标题界面，点击进行各部分之间的跳转。上面自动生成的目录缩进格式有点问题，导航中的标题界面更加清晰  由于在自己写代码实现功能的部分报告上（**调用函数库，编写代码实现以上工具完成的功能的部分**），我贴出了部分源代码的实现主要功能的函数，所以较长，使用上面的方法可以更好浏览报告，减少源代码太长导致篇幅太长的问题。 |

## 实验具体开发步骤：

### 1.在 windows 下（使用 visual studio 或其他）或 LINUX 下编译最新的OPENSSL 密码安全库（https://www.openssl.org， openssl-1.1.1），生成静态库或动态库。

|  |
| --- |
| 本次实验使用的是Visual studio2022+OPENSSL3.3.2，延续使用上一次密码学原理与实践实验的软件系统，没有进行重新安装最新版3.5.0，经过查阅资料，新版本的内容并不会对实验产生影响，故而继续使用OPENSSL3.3.2 |

### 2. 使用 openssl 工具完成如下操作：

|  |
| --- |
| **（1）RSA 2048 位 密钥生成；**  这里生成RSA密钥是一种公钥加密算法，所以，在开始之前，先回忆一下openssl支持的所有非对称加密算法  命令：  # openssl list -public-key-algorithms    后面省略，可以看到有RSA和DH等比较著名和学习过的算法以及没有接触过的算法，后面我们首先体验一下OPENSSL命令行工具的使用  ●生成私钥 2048为密钥位数  #openssl genrsa -out private.key 2048  生成了2048位的私钥文件  **（2）导出公钥；**  ●从私钥文件中导出公钥  #openssl rsa -in private.key -pubout -out public.key  成功由私钥导出公钥文件  **（3）****生成数字证书请求；**  **数字证书请求用于向 证书颁发机构（CA） 申请数字证书，包含你的公钥和身份信息（如域名、组织名称等）。**  **CSR 本身 不是证书，而是申请证书的中间文件，需由 CA 签名后才能变成正式证书。**  **这里使用私钥生成证书请求CSR**  ●**生成证书请求文件**  **# openssl req -new -key private.key -out request.csr**  **在填写完一系列资料后，就可以生成证书请求文件**  填写资料如下    **（4）****生成数字证书；**  实验用途所以使用自签名：**自签名（Self-Signed，用于测试或内部用途）**  ●直接生成自签名证书（跳过CSR步骤）  #openssl req -x509 -newkey rsa:2048 -keyout private.key -out certificate.crt -days 365  填写完基本信息后生成数字证书（后续不在贴出基本信息）    ●或对已有 CSR 签名（需 CA 私钥）：  #openssl x509 -req -in request.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out certificate.crt -days 365  步骤 1：生成 CA 私钥（ca.key）  #openssl genpkey -algorithm RSA -out ca.key -pkeyopt rsa\_keygen\_bits:2048  步骤 2：生成自签名 CA 根证书（ca.crt）  #openssl req -x509 -new -key ca.key -out ca.crt -days 3650 -subj "/CN=My Root CA"  过程文件和最终文件如下    **（5）****生成 pkcs12 格式基于口令保护的标准格式数据包。**  在 OpenSSL 中，可以使用 pkcs12 命令将证书（.crt）和私钥（.key）打包成 PKCS#12（.p12 或 .pfx） 格式，并使用口令（密码）加密保护。  ●将以下三种文件进行打包：  私钥文件（如 private.key）  证书文件（如 certificate.crt）  CA 证书链文件（可选，如 ca.crt）  #openssl pkcs12 -export \  -inkey private.key \ # 指定私钥文件  -in certificate.crt \ # 指定证书文件  -out bundle.p12 \ # 输出的 PKCS#12 文件名  -name "My Certificate" \ # 别名（可选）  -certfile ca.crt \ # CA 证书链（可选）  -password pass:123456 # 设置保护密码    **（6）用 ASN1 编码解析器，显示以上生成数据对象的 ASN1 编码数据。**  ● 查看 ASN.1 结构  #openssl asn1parse -in bundle.p12 -inform DER    中间是很长的十六进制编码，编码过后为：    **用 ASN1 编码解析器，解析ca.crt**    **用 ASN1 编码解析器，解析private.key**    **用 ASN1 编码解析器，解析certificate.crt** |

### 3. 调用函数库，编写代码实现以上工具完成的功能（1-6）。

#### RSA 2048 位 密钥生成

|  |
| --- |
| 生成2048位的密钥，首先进行定义#define RSA\_KEY\_BITS 2048，保证生成的密钥长度符合要求，下面为生成CTX的长度参数设定函数  然后生成pkey，它是 OpenSSL 的 统一密钥抽象层，可以容纳RSA 密钥（公钥/私钥）  方便以后私钥或者公钥的生成    在有了pkey以后，就可以利用PEM\_write\_PUBKEY(pub\_file, pkey) 和PEM\_write\_PrivateKey(priv\_file, pkey, NULL, NULL, 0, NULL, NULL)分别生成公钥和私钥，但是这里我们要求生成私钥，再利用私钥得到公钥，因此，只使用PEM\_write\_PrivateKey(priv\_file, pkey, NULL, NULL, 0, NULL, NULL)私钥生成函数，公钥使用其他方法得到  生成2048位RSA私钥部分代码如下：  // 生成RSA私钥并保存到文件  int generate\_rsa\_private\_key(const char\* private\_key\_file) {  EVP\_PKEY\_CTX\* ctx = EVP\_PKEY\_CTX\_new\_id(EVP\_PKEY\_RSA, NULL);  if (!ctx || EVP\_PKEY\_keygen\_init(ctx) <= 0) handle\_errors();  if (EVP\_PKEY\_CTX\_set\_rsa\_keygen\_bits(ctx, RSA\_KEY\_BITS) <= 0) handle\_errors();  EVP\_PKEY\* pkey = NULL;  if (EVP\_PKEY\_keygen(ctx, &pkey) <= 0) handle\_errors();  FILE\* priv\_file = fopen(private\_key\_file, "wb");  if (!priv\_file) {  perror("Failed to open private key file");  EVP\_PKEY\_free(pkey);  EVP\_PKEY\_CTX\_free(ctx);  return 0;  }  int success = PEM\_write\_PrivateKey(priv\_file, pkey, NULL, NULL, 0, NULL, NULL);  fclose(priv\_file);  EVP\_PKEY\_free(pkey);  EVP\_PKEY\_CTX\_free(ctx);  return success;  } |

#### 导出公钥

|  |
| --- |
| 在经过上面的函数得到私钥文件后，可以读取文件（PEM\_read\_PrivateKey(priv\_file, NULL, NULL, NULL)），使用EVP\_PKEY\_dup(private\_key);函数，从私钥中提取公钥，然后进行类似上个函数的工作保存公钥，得到公钥的函数如下:  // 从私钥文件提取公钥并保存  int extract\_public\_key(const char\* private\_key\_file, const char\* public\_key\_file) {  FILE\* priv\_file = fopen(private\_key\_file, "rb");  if (!priv\_file) {  perror("Failed to open private key file");  return 0;  }  EVP\_PKEY\* private\_key = PEM\_read\_PrivateKey(priv\_file, NULL, NULL, NULL);  fclose(priv\_file);  if (!private\_key) {  fprintf(stderr, "Failed to read private key\n");  return 0;  }  EVP\_PKEY\* public\_key = EVP\_PKEY\_dup(private\_key);  EVP\_PKEY\_free(private\_key);  if (!public\_key) {  fprintf(stderr, "Failed to extract public key\n");  return 0;  }  FILE\* pub\_file = fopen(public\_key\_file, "wb");  if (!pub\_file) {  perror("Failed to open public key file");  EVP\_PKEY\_free(public\_key);  return 0;  }  int success = PEM\_write\_PUBKEY(pub\_file, public\_key);  fclose(pub\_file);  EVP\_PKEY\_free(public\_key);  return success;  } |

#### 生成数字证书请求:

|  |
| --- |
| 数字证书请求用于向 证书颁发机构（CA） 申请数字证书，包含你的公钥和身份信息（如域名、组织名称等）。  这一步一共涉及两个参数，一个是签名的私钥，另一个是生成的证书请求CSR文件。  首先，我们要读取私钥，作为一个pkey类型的参数，pkey是openssl中的一个 EVP\_PKEY 对象可以同时存储公私钥对。  即使我从私钥文件（如 private.key）加载，OpenSSL 也会自动解析并存储其中的公钥部分，在后面创建证书请求后，需要设置版本和设置公钥，公钥可以直接从pkey中获得。  然后，模仿openssl中创建证书请求时，进行交互询问问题的方式，获得证书请求的各种信息，效果如图  这里采用的是fget和printf的交互进行，部分实现代码如图：    通过读取每行输入的文字，通过检测\n替换\0来给Subject名称（X509\_NAME\* name = X509\_NAME\_new();）填写信息，最终用X509\_REQ\_set\_subject\_name(req, name);将name放入请求中再用私钥签名，完成构建，具体函数的代码如下  // 生成证书请求(CSR)  int generate\_certificate\_request(const char\* private\_key\_file, const char\* csr\_file) {  // 读取私钥  FILE\* priv\_file = fopen(private\_key\_file, "rb");  if (!priv\_file) {  perror("Failed to open private key file");  return 0;  }  //pkey是openssl中的一个 EVP\_PKEY 对象可以同时存储公私钥对。  //即使我从私钥文件（如 private.key）加载，OpenSSL 也会自动解析并存储其中的公钥部分  EVP\_PKEY\* private\_key = PEM\_read\_PrivateKey(priv\_file, NULL, NULL, NULL);  fclose(priv\_file);  if (!private\_key) {  fprintf(stderr, "Failed to read private key\n");  return 0;  }  // 创建证书请求  X509\_REQ\* req = X509\_REQ\_new();  if (!req) {  fprintf(stderr, "Failed to create certificate request\n");  EVP\_PKEY\_free(private\_key);  return 0;  }  // 设置版本  X509\_REQ\_set\_version(req, X509\_REQ\_VERSION\_1);  // 设置公钥  if (!X509\_REQ\_set\_pubkey(req, private\_key)) {  fprintf(stderr, "Failed to set public key in request\n");  X509\_REQ\_free(req);  EVP\_PKEY\_free(private\_key);  return 0;  }  // 创建Subject名称  X509\_NAME\* name = X509\_NAME\_new();  if (!name) {  fprintf(stderr, "Failed to create X509 name\n");  X509\_REQ\_free(req);  EVP\_PKEY\_free(private\_key);  return 0;  }  // 交互式输入Subject信息  char input[256];  printf("You are about to be asked to enter information that will be incorporated\n");  printf("into your certificate request. What you are about to enter is what is called\n");  printf("a Distinguished Name or a DN. There are quite a few fields but you can leave\n");  printf("some blank. For some fields there will be a default value. If you enter '.',\n");  printf("the field will be left blank.\n\n");  printf("Country Name (2 letter code) [AU]: ");  fgets(input, sizeof(input), stdin);  input[strcspn(input, "\n")] = '\0';  if (strlen(input) > 0) {  X509\_NAME\_add\_entry\_by\_txt(name, "C", MBSTRING\_ASC, (unsigned char\*)input, -1, -1, 0);  }  printf("State or Province Name (full name) [Some-State]: ");  fgets(input, sizeof(input), stdin);  input[strcspn(input, "\n")] = '\0';  if (strlen(input) > 0) {  X509\_NAME\_add\_entry\_by\_txt(name, "ST", MBSTRING\_ASC, (unsigned char\*)input, -1, -1, 0);  }  printf("Locality Name (eg, city) []: ");  fgets(input, sizeof(input), stdin);  input[strcspn(input, "\n")] = '\0';  if (strlen(input) > 0) {  X509\_NAME\_add\_entry\_by\_txt(name, "L", MBSTRING\_ASC, (unsigned char\*)input, -1, -1, 0);  }  printf("Organization Name (eg, company) [Internet Widgets Pty Ltd]: ");  fgets(input, sizeof(input), stdin);  input[strcspn(input, "\n")] = '\0';  if (strlen(input) > 0) {  X509\_NAME\_add\_entry\_by\_txt(name, "O", MBSTRING\_ASC, (unsigned char\*)input, -1, -1, 0);  }  printf("Organizational Unit Name (eg, section) []: ");  fgets(input, sizeof(input), stdin);  input[strcspn(input, "\n")] = '\0';  if (strlen(input) > 0) {  X509\_NAME\_add\_entry\_by\_txt(name, "OU", MBSTRING\_ASC, (unsigned char\*)input, -1, -1, 0);  }  printf("Common Name (e.g. server FQDN or YOUR name) []: ");  fgets(input, sizeof(input), stdin);  input[strcspn(input, "\n")] = '\0';  if (strlen(input) > 0) {  X509\_NAME\_add\_entry\_by\_txt(name, "CN", MBSTRING\_ASC, (unsigned char\*)input, -1, -1, 0);  }  printf("Email Address []: ");  fgets(input, sizeof(input), stdin);  input[strcspn(input, "\n")] = '\0';  if (strlen(input) > 0) {  X509\_NAME\_add\_entry\_by\_txt(name, "emailAddress", MBSTRING\_ASC, (unsigned char\*)input, -1, -1, 0);  }  // 设置Subject到请求  X509\_REQ\_set\_subject\_name(req, name);  X509\_NAME\_free(name);  // 签名请求  if (!X509\_REQ\_sign(req, private\_key, EVP\_sha256())) {  fprintf(stderr, "Failed to sign certificate request\n");  X509\_REQ\_free(req);  EVP\_PKEY\_free(private\_key);  return 0;  }  // 写入CSR文件  FILE\* csr\_file\_ptr = fopen(csr\_file, "wb");  if (!csr\_file\_ptr) {  perror("Failed to open CSR file");  X509\_REQ\_free(req);  EVP\_PKEY\_free(private\_key);  return 0;  }  int success = PEM\_write\_X509\_REQ(csr\_file\_ptr, req);  fclose(csr\_file\_ptr);  X509\_REQ\_free(req);  EVP\_PKEY\_free(private\_key);  return success;  } |

#### 生成数字证书：

|  |
| --- |
| 实验用途所以使用自签名，生成数字证书和自签名的逻辑是一样的，不过证书请求改成了x509的数字证书，相较于上一部分的代码，这里我做出了部分改进：  首先，在交互式信息请求那里，由于请求很多，所以会出现代码重复的问题，这里我写了一个函数来简化一下过程    后面直接调用函数就可以了，相较于上一个证书请求的实现更加简洁    其次，在使用程序时，随着编写的函数越来越多，调用时，也会参数越来越多，变得眼花缭乱，我在参数错误时的提醒中，添加了example的示例，来教程序的使用者更好的，进行使用，参数使用错误情况如下：    下面是封装的数字证书函数：    // 生成自签名证书  int generate\_self\_signed\_cert(const char\* key\_file, const char\* cert\_file, int days) {  // 1. 读取已有私钥  FILE\* key\_fp = fopen(key\_file, "rb");  if (!key\_fp) {  perror("Error opening private key file");  return 0;  }  EVP\_PKEY\* pkey = PEM\_read\_PrivateKey(key\_fp, NULL, NULL, NULL);  fclose(key\_fp);  if (!pkey) {  fprintf(stderr, "Error reading private key\n");  return 0;  }  // 2. 创建X509证书结构  X509\* x509 = X509\_new();  if (!x509) {  EVP\_PKEY\_free(pkey);  return 0;  }  // 3. 设置证书版本  X509\_set\_version(x509, X509\_VERSION\_3);  // 4. 设置随机序列号  ASN1\_INTEGER\_set(X509\_get\_serialNumber(x509), 1);  // 5. 设置有效期  X509\_gmtime\_adj(X509\_get\_notBefore(x509), 0);  X509\_gmtime\_adj(X509\_get\_notAfter(x509), days \* 24 \* 3600);  // 6. 设置公钥（从私钥提取）  X509\_set\_pubkey(x509, pkey);  // 7. 交互式输入Subject信息  X509\_NAME\* name = X509\_get\_subject\_name(x509);  printf("\nYou are about to be asked to enter information that will be incorporated\n");  printf("into your certificate request. What you are about to enter is what is called\n");  printf("a Distinguished Name or a DN. There are quite a few fields but you can leave\n");  printf("some blank. For some fields there will be a default value. If you enter '.',\n");  printf("the field will be left blank.\n\n");  char input[256];  read\_input("Country Name (2 letter code) [AU]: ", input, sizeof(input));  if (strlen(input) > 0) {  X509\_NAME\_add\_entry\_by\_txt(name, "C", MBSTRING\_ASC, (unsigned char\*)input, -1, -1, 0);  }  read\_input("State or Province Name (full name) [Some-State]: ", input, sizeof(input));  if (strlen(input) > 0) {  X509\_NAME\_add\_entry\_by\_txt(name, "ST", MBSTRING\_ASC, (unsigned char\*)input, -1, -1, 0);  }  read\_input("Locality Name (eg, city) []: ", input, sizeof(input));  if (strlen(input) > 0) {  X509\_NAME\_add\_entry\_by\_txt(name, "L", MBSTRING\_ASC, (unsigned char\*)input, -1, -1, 0);  }  read\_input("Organization Name (eg, company) [Internet Widgets Pty Ltd]: ", input, sizeof(input));  if (strlen(input) > 0) {  X509\_NAME\_add\_entry\_by\_txt(name, "O", MBSTRING\_ASC, (unsigned char\*)input, -1, -1, 0);  }  read\_input("Organizational Unit Name (eg, section) []: ", input, sizeof(input));  if (strlen(input) > 0) {  X509\_NAME\_add\_entry\_by\_txt(name, "OU", MBSTRING\_ASC, (unsigned char\*)input, -1, -1, 0);  }  read\_input("Common Name (e.g. server FQDN or YOUR name) []: ", input, sizeof(input));  if (strlen(input) > 0) {  X509\_NAME\_add\_entry\_by\_txt(name, "CN", MBSTRING\_ASC, (unsigned char\*)input, -1, -1, 0);  }  read\_input("Email Address []: ", input, sizeof(input));  if (strlen(input) > 0) {  X509\_NAME\_add\_entry\_by\_txt(name, "emailAddress", MBSTRING\_ASC, (unsigned char\*)input, -1, -1, 0);  }  // 8. 设置颁发者信息（自签名证书与Subject相同）  X509\_set\_issuer\_name(x509, name);  // 9. 用私钥签名证书  if (!X509\_sign(x509, pkey, EVP\_sha256())) {  fprintf(stderr, "Error signing certificate\n");  X509\_free(x509);  EVP\_PKEY\_free(pkey);  return 0;  }  // 10. 保存证书  FILE\* cert\_fp = fopen(cert\_file, "wb");  if (!cert\_fp) {  perror("Error opening certificate file");  X509\_free(x509);  EVP\_PKEY\_free(pkey);  return 0;  }  PEM\_write\_X509(cert\_fp, x509);  fclose(cert\_fp);  // 11. 释放资源  X509\_free(x509);  EVP\_PKEY\_free(pkey);  return 1;  } |

#### 生成 pkcs12 格式基于口令保护的标准格式数据包。

|  |
| --- |
| 在 OpenSSL 中，可以使用 pkcs12 命令将证书（.crt）和私钥（.key）打包成 PKCS#12（.p12 或 .pfx） 格式，并使用口令（密码）加密保护。  ●将以下三种文件进行打包：  私钥文件（如 private.key）  证书文件（如 certificate.crt）  CA 证书链文件（可选，如 ca.crt）  这里需要进行加密和解密的分别实现，而且因为参数具有可选项，所以我修改了这次的参数检测，改成了有的参数是必须的，有的参数可以不存在，如加密时的CA证书链文件和解密时的只解密私钥文件，不解密证书文件，下面给出解包时的参数检测，（因为pkcs12打包参数更多，这里放不开，但是原理是相同的）    下面写一下每一个函数的实现流程：  **1. generate\_pkcs12 函数实现流程**   1. **读取私钥**：    * 打开私钥文件（key\_file）。    * 使用 PEM\_read\_PrivateKey 函数读取私钥。    * 关闭私钥文件。    * 如果私钥读取失败，打印错误信息并返回。 2. **读取证书**：    * 打开证书文件（cert\_file）。    * 使用 PEM\_read\_X509 函数读取证书。    * 关闭证书文件。    * 如果证书读取失败，打印错误信息并返回。 3. **读取CA证书链（可选）**：    * 如果提供了CA证书文件（ca\_file），打开该文件。    * 初始化一个空的 STACK\_OF(X509) 结构用于存储CA证书。    * 使用 PEM\_read\_X509 函数循环读取CA证书，并将其推入栈中。    * 关闭CA证书文件。 4. **创建PKCS#12结构**：    * 使用 PKCS12\_create 函数创建一个PKCS#12结构。    * 参数包括密码、友好名称、私钥、证书、CA证书链、加密算法（如TripleDES）和迭代次数。    * 如果创建失败，打印错误信息并释放已分配的资源，然后返回。 5. **写入PKCS#12文件**：    * 打开输出文件（output\_file）。    * 使用 i2d\_PKCS12\_fp 函数将PKCS#12结构写入文件。    * 关闭输出文件。 6. **清理资源**：    * 释放PKCS#12结构、CA证书栈、证书和私钥的内存。 7. **返回结果**：    * 返回写入操作的结果（成功或失败）。   **2. extract\_pkcs12 函数实现流程**  **功能**：从PKCS#12文件中提取私钥和证书，并验证其有效性。  **实现步骤：**   1. **打开PKCS#12文件**：    * 打开PKCS#12文件（p12\_file）。    * 如果打开失败，打印错误信息并返回。 2. **读取PKCS#12结构**：    * 使用 d2i\_PKCS12\_fp 函数从文件中读取PKCS#12结构。    * 关闭文件。    * 如果读取失败，打印错误信息并返回。 3. **验证MAC（密码检查）**：    * 使用 PKCS12\_verify\_mac 函数验证PKCS#12结构的MAC（消息认证码）。    * 如果验证失败，打印错误信息并释放PKCS#12结构，然后返回。 4. **解析PKCS#12结构**：    * 使用 PKCS12\_parse 函数解析PKCS#12结构，提取私钥、证书和CA证书链。    * 如果解析失败，打印错误信息并释放PKCS#12结构，然后返回。 5. **验证密钥对匹配**：    * 使用 X509\_check\_private\_key 函数验证私钥与证书是否匹配。    * 如果不匹配，打印错误信息并释放已分配的资源，然后返回。 6. **输出私钥**：    * 如果提供了私钥输出文件（key\_out），打开该文件。    * 使用 PEM\_write\_PrivateKey 函数将私钥写入文件。    * 关闭文件。    * 如果写入失败，打印错误信息。 7. **输出证书**：    * 如果提供了证书输出文件（cert\_out）且证书存在，打开该文件。    * 使用 PEM\_write\_X509 函数将证书写入文件。    * 关闭文件。    * 如果写入失败，打印错误信息。 8. **清理资源**：    * 释放私钥、证书和PKCS#12结构的内存。 9. **返回结果**：    * 返回操作成功或失败的标志。   具体代码如下：  // 生成 PKCS12 文件  int generate\_pkcs12(const char\* key\_file, const char\* cert\_file,  const char\* ca\_file, const char\* output\_file,  const char\* friendly\_name, const char\* password) {  // 1. 读取私钥  FILE\* key\_fp = fopen(key\_file, "rb");  if (!key\_fp) {  perror("Error opening private key file");  return 0;  }  EVP\_PKEY\* pkey = PEM\_read\_PrivateKey(key\_fp, NULL, NULL, NULL);  fclose(key\_fp);  if (!pkey) {  fprintf(stderr, "Error reading private key\n");  return 0;  }  // 2. 读取证书  FILE\* cert\_fp = fopen(cert\_file, "rb");  if (!cert\_fp) {  perror("Error opening certificate file");  EVP\_PKEY\_free(pkey);  return 0;  }  X509\* cert = PEM\_read\_X509(cert\_fp, NULL, NULL, NULL);  fclose(cert\_fp);  if (!cert) {  fprintf(stderr, "Error reading certificate\n");  EVP\_PKEY\_free(pkey);  return 0;  }  // 3. 读取CA证书链（可选）  STACK\_OF(X509)\* ca\_stack = NULL;  if (ca\_file && strlen(ca\_file) > 0) {  FILE\* ca\_fp = fopen(ca\_file, "rb");  if (!ca\_fp) {  perror("Error opening CA certificate file");  X509\_free(cert);  EVP\_PKEY\_free(pkey);  return 0;  }  ca\_stack = sk\_X509\_new\_null();  X509\* ca\_cert;  while ((ca\_cert = PEM\_read\_X509(ca\_fp, NULL, NULL, NULL))) {  sk\_X509\_push(ca\_stack, ca\_cert);  }  fclose(ca\_fp);  }  // 4. 创建PKCS12结构  PKCS12\* p12 = PKCS12\_create(password, friendly\_name, pkey, cert, ca\_stack,  NID\_pbe\_WithSHA1And3\_Key\_TripleDES\_CBC,  NID\_pbe\_WithSHA1And3\_Key\_TripleDES\_CBC,  PKCS12\_DEFAULT\_ITER, PKCS12\_DEFAULT\_ITER, 0);  if (!p12) {  fprintf(stderr, "Error creating PKCS12 structure\n");  if (ca\_stack) sk\_X509\_pop\_free(ca\_stack, X509\_free);  X509\_free(cert);  EVP\_PKEY\_free(pkey);  return 0;  }  // 5. 写入文件  FILE\* out\_fp = fopen(output\_file, "wb");  if (!out\_fp) {  perror("Error opening output file");  PKCS12\_free(p12);  if (ca\_stack) sk\_X509\_pop\_free(ca\_stack, X509\_free);  X509\_free(cert);  EVP\_PKEY\_free(pkey);  return 0;  }  int ret = i2d\_PKCS12\_fp(out\_fp, p12);  fclose(out\_fp);  // 6. 清理资源  PKCS12\_free(p12);  if (ca\_stack) sk\_X509\_pop\_free(ca\_stack, X509\_free);  X509\_free(cert);  EVP\_PKEY\_free(pkey);  return ret;  }  // PKCS12解包函数  int extract\_pkcs12(const char\* p12\_file, const char\* password,  const char\* key\_out, const char\* cert\_out) {  FILE\* fp = fopen(p12\_file, "rb");  if (!fp) {  perror("Error opening PKCS12 file");  return 0;  }  PKCS12\* p12 = d2i\_PKCS12\_fp(fp, NULL);  fclose(fp);  if (!p12) {  fprintf(stderr, "Error reading PKCS12 structure\n");  return 0;  }  // 验证MAC（密码检查）  if (!PKCS12\_verify\_mac(p12, password, -1)) {  fprintf(stderr, "Password verification failed\n");  PKCS12\_free(p12);  return 0;  }  EVP\_PKEY\* pkey = NULL;  X509\* cert = NULL;  STACK\_OF(X509)\* ca\_stack = NULL;  if (!PKCS12\_parse(p12, password, &pkey, &cert, &ca\_stack)) {  fprintf(stderr, "Failed to parse PKCS12 (wrong password?)\n");  PKCS12\_free(p12);  return 0;  }  // 验证密钥对匹配  if (cert && !X509\_check\_private\_key(cert, pkey)) {  fprintf(stderr, "Certificate does not match private key\n");  EVP\_PKEY\_free(pkey);  X509\_free(cert);  sk\_X509\_pop\_free(ca\_stack, X509\_free);  PKCS12\_free(p12);  return 0;  }  // 输出私钥  int success = 1;  if (key\_out) {  FILE\* key\_fp = fopen(key\_out, "wb");  if (!key\_fp) {  perror("Error opening key output file");  success = 0;  }  else {  success = PEM\_write\_PrivateKey(key\_fp, pkey, NULL, NULL, 0, NULL, NULL);  fclose(key\_fp);  if (!success) fprintf(stderr, "Failed to write private key\n");  }  }  // 输出证书（可选）  if (success && cert\_out && cert) {  FILE\* cert\_fp = fopen(cert\_out, "wb");  if (!cert\_fp) {  perror("Error opening cert output file");  success = 0;  }  else {  success = PEM\_write\_X509(cert\_fp, cert);  fclose(cert\_fp);  if (!success) fprintf(stderr, "Failed to write certificate\n");  }  }  // 清理资源  EVP\_PKEY\_free(pkey);  X509\_free(cert);  sk\_X509\_pop\_free(ca\_stack, X509\_free);  PKCS12\_free(p12);  return success;  } |

#### 简单实现asn1parse的解析器

|  |
| --- |
| 要实现模拟ASN1解析器，就要先了解一下ASN1格式编码，查阅可知：  ASN.1（Abstract Syntax Notation One）是一种数据描述语言，用于定义数据结构。ASN.1 数据通常以 TLV（Tag-Length-Value）格式编码。  Tag：标识数据类型（如 INTEGER、BOOLEAN 等）。  Length：表示数据的长度。  Value：实际的数据内容。  实现思路：  通过不断读取文件内容，解析 ASN.1 数据（PEM 或 DER 格式），递归解析其结构，并输出每个元素的 Tag、Length 和 Value 信息。  根据思路进行设计：  parse\_asn1\_file  ├── 打开文件并创建 BIO 对象  ├── 尝试解析为 PEM 格式（这是要求的三种基本格式）  │ ├── PEM\_read\_bio\_X509 → i2d\_X509 → parse\_asn1  │ ├── PEM\_read\_bio\_PrivateKey → i2d\_PrivateKey → parse\_asn1  │ └── PEM\_read\_bio\_X509\_REQ → i2d\_X509\_REQ → parse\_asn1  ├── 如果不是 PEM 格式，调用 bio\_to\_der，进行暴力解析der  │ └── bio\_to\_der → parse\_asn1  └── 释放资源  上面函数的基本作用是：  **函数之间的关系**  代码主要由以下几个函数组成，它们通过协作实现 ASN.1 数据的解析：  **(1) get\_asn1\_type\_name**   * **功能**： 根据 ASN.1 的 Tag 值，返回对应的类型名称（如 "INTEGER"、"BOOLEAN" 等）。 * **实现**： 使用一个静态表 asn1\_tags 存储 Tag 和类型名称的映射关系，通过遍历表查找匹配的 Tag。 * **作用**： 为解析出的 Tag 提供人类可读的类型名称。   **(2) parse\_asn1**   * **功能**： 递归解析 ASN.1 数据结构。 * **实现**：   + 使用 ASN1\_get\_object 解析 TLV 头部，获取 Tag、Length 和类型（构造类型或基本类型）。   + 如果是构造类型（如 SEQUENCE、SET），递归调用自身解析子元素。   + 如果是基本类型（如 INTEGER、BOOLEAN），根据 Tag 类型解析 Value 内容。 * **关键点**：   + 使用指针操作（const unsigned char\*\* data）递归处理嵌套结构。   + 针对不同类型（如 INTEGER、TIME、STRING 等）提供特定的解析逻辑。 * **作用**： 核心解析逻辑，负责解析 ASN.1 数据的所有层级。   **(3) bio\_to\_der**   * **功能**： 将 BIO（Binary Input/Output）对象中的数据读取为 DER 格式的二进制数据。 * **实现**：   + 使用 BIO\_read 从 BIO 中读取数据。   + 将数据写入内存 BIO（BIO\_s\_mem），然后读取为二进制缓冲区。 * **作用**： 将 PEM 格式的文件数据转换为 DER 格式的二进制数据，便于后续解析。   **(4) parse\_asn1\_file**   * **功能**： 主解析函数，处理文件输入，支持 PEM 和 DER 格式。 * **实现**：   1. 打开文件并创建 BIO 对象。   2. 尝试解析为 PEM 格式的证书（X509）、私钥（PrivateKey）或 CSR（Certificate Signing Request）。      + 使用 PEM\_read\_bio\_\* 函数解析 PEM 数据。      + 使用 i2d\_\* 函数将解析后的数据转换为 DER 格式。   3. 如果文件不是 PEM 格式，则尝试直接读取为二进制数据（DER 格式）。   4. 调用 parse\_asn1 解析 DER 数据。 * **作用**： 文件输入的入口函数，负责处理不同格式的文件并调用解析逻辑。   纠错过程：  这个功能比其他前五个功能的实现都多花费了一些时间，主要是因为这个功能openssl库的封装程度较低，需要自己编写的内容多，而且解析就涉及到各种数据类型，需要针对不同数据类型进行区分，尽量使数据类型完善，防止出现未知类型，导致代码错误。  问题现象：  遇到INTEGER/UTCTIME类型时解析中断  大量UNKNOWN类型识别失败  CSR文件识别率低  关键代码段：  // 初始版本的类型识别表  static const struct {  int tag;  const char\* name;  } asn1\_tags[] = {  {V\_ASN1\_BOOLEAN, "BOOLEAN"},  // ...（仅包含基础类型）  };  2. 改进过程  （1）类型识别扩展  新增20+种ASN.1类型识别：  {V\_ASN1\_ENUMERATED, "ENUMERATED"},  {V\_ASN1\_VISIBLESTRING, "VISIBLESTRING"},  // ...（PKCS#7/PKCS#12相关类型）  （2）指针同步机制  // 改进后的构造类型处理  const unsigned char\* temp\_ptr = p;  parse\_asn1(&temp\_ptr, tag\_len, depth + 1, out);  p = temp\_ptr; // 关键指针同步  （3）CSR识别增强  int is\_csr\_pem(BIO\* bio) {  char buf[256];  BIO\_read(bio, buf, sizeof(buf));  BIO\_seek(bio, 0);  return strstr(buf, "CERTIFICATE REQUEST");  }  3. 最终解决方案  核心改进点：  类型识别表扩展到32种类型  增加PKCS系列特殊类型处理  完善的错误恢复机制  使用示例：  解析生成的证书请求csr代码，可以看到，能够成功分辨不同代码行的位置和类型  而且能够成功解析出value也就是内容，  比如：国家：CN  省份：ShanDong……这些都是我在生成请求时输入的信息，与openssl的asn1parse较为相似，但是肯定没有它的更为准确全面。    特殊情况：  在常规情况下，使用PKCS12打包的带密码的文件无法进行解析，如openssl，就无法解析    而我写的代码在即使是二进制的der的文件下，也能够暴力解析，期待也能够得到一些信息  封装好的4个代码函数如下（有点长300行）：  // ASN.1 类型描述表  static const struct {  int tag;  const char\* name;  } asn1\_tags[] = {  {V\_ASN1\_BOOLEAN, "BOOLEAN"},  {V\_ASN1\_INTEGER, "INTEGER"},  {V\_ASN1\_BIT\_STRING, "BIT STRING"},  {V\_ASN1\_OCTET\_STRING, "OCTET STRING"},  {V\_ASN1\_NULL, "NULL"},  {V\_ASN1\_OBJECT, "OBJECT"},  {V\_ASN1\_UTF8STRING, "UTF8STRING"},  {V\_ASN1\_SEQUENCE, "SEQUENCE"},  {V\_ASN1\_SET, "SET"},  {V\_ASN1\_PRINTABLESTRING, "PRINTABLESTRING"},  {V\_ASN1\_IA5STRING, "IA5STRING"},  {V\_ASN1\_UTCTIME, "UTCTIME"},  {V\_ASN1\_GENERALIZEDTIME, "GENERALIZEDTIME"},  {V\_ASN1\_T61STRING, "T61STRING"},  {V\_ASN1\_IA5STRING, "IA5STRING"},  {V\_ASN1\_UNIVERSALSTRING, "UNIVERSALSTRING"},  {V\_ASN1\_BMPSTRING, "BMPSTRING"},  {V\_ASN1\_VISIBLESTRING, "VISIBLESTRING"},  {V\_ASN1\_ENUMERATED, "ENUMERATED"},  {V\_ASN1\_ENUMERATED, "ENUMERATED"},  {V\_ASN1\_VISIBLESTRING, "VISIBLESTRING"},  {V\_ASN1\_T61STRING, "T61STRING"},  {V\_ASN1\_BMPSTRING, "BMPSTRING"},  {V\_ASN1\_UNIVERSALSTRING,"UNIVERSALSTRING"},  {V\_ASN1\_NUMERICSTRING, "NUMERICSTRING"},  {V\_ASN1\_VIDEOTEXSTRING,"VIDEOTEXSTRING"},  {V\_ASN1\_GRAPHICSTRING, "GRAPHICSTRING"},  {V\_ASN1\_ISO64STRING, "ISO64STRING"},  {V\_ASN1\_GENERALSTRING, "GENERALSTRING"},  {V\_ASN1\_TELETEXSTRING, "TELETEXSTRING"},  {0, NULL}  };  const char\* get\_asn1\_type\_name(int tag) {  for (int i = 0; asn1\_tags[i].name; i++) {  if (asn1\_tags[i].tag == tag) {  return asn1\_tags[i].name;  }  }  return "UNKNOWN";  }  void print\_indent(const unsigned char\* p, const unsigned char\* start, int depth, FILE\* out) {  fprintf(out, "%4ld:d=%d ", (long)(p - start), depth);  for (int i = 0; i < depth; i++) {  fprintf(out, " ");  }  }  // 递归解析ASN.1结构  void parse\_asn1(const unsigned char\*\* data, long length, int depth, FILE\* out) {  const unsigned char\* p = \*data;  const unsigned char\* start = p;  const unsigned char\* end = p + length;  while (p < end && depth < MAX\_DEPTH) {  const unsigned char\* elem\_start = p; // 记录元素起始位置  int tag, klass;  long tag\_len;  int is\_constructed = 0;  // 1. 解析TLV头部  int ret = ASN1\_get\_object(&p, &tag\_len, &tag, &klass, end - p);  is\_constructed = ret & V\_ASN1\_CONSTRUCTED;  long header\_len = p - elem\_start;  const unsigned char\* value\_ptr = p; // Value部分开始位置  // 2. 增强的错误检查  if (ret & 0x80) {  fprintf(out, "%4ld:d=%d [ERROR] Invalid ASN.1 header\n",  (long)(elem\_start - start), depth);  break;  }  if (p + tag\_len > end) {  fprintf(out, "%4ld:d=%d [ERROR] Length overflow (hl=%ld l=%ld remain=%ld)\n",  (long)(elem\_start - start), depth,  header\_len, tag\_len, (long)(end - p));  break;  }  // 3. 打印元素信息  fprintf(out, "%4ld:d=%d hl=%ld l=%4ld ",  (long)(elem\_start - start), depth, header\_len, tag\_len);  // 4. 处理构造类型  if (is\_constructed) {  fprintf(out, "cons: %s\n", get\_asn1\_type\_name(tag));  const unsigned char\* temp\_ptr = p;  parse\_asn1(&temp\_ptr, tag\_len, depth + 1, out);  p = temp\_ptr; // 关键点：更新外层指针位置  }  // 5. 处理基本类型  else {  fprintf(out, "prim: %s", get\_asn1\_type\_name(tag));  // 特殊类型值解析  switch (tag) {  case V\_ASN1\_INTEGER: {  // 简化显示，实际使用ASN1\_INTEGER处理更安全  fprintf(out, ":");  for (long i = 0; i < tag\_len && i < 8; i++) {  fprintf(out, "%02X", value\_ptr[i]);  }  if (tag\_len > 8) fprintf(out, "...");  break;  }  case V\_ASN1\_UTCTIME:  case V\_ASN1\_GENERALIZEDTIME: {  // 简化显示时间内容  fprintf(out, ":%.\*s", (int)tag\_len, value\_ptr);  break;  }  case V\_ASN1\_OBJECT: {  char objbuf[256];  OBJ\_obj2txt(objbuf, sizeof(objbuf), (ASN1\_OBJECT\*)value\_ptr, 1);  fprintf(out, ":%s", objbuf);  break;  }  case V\_ASN1\_BOOLEAN:  fprintf(out, ":%s", \*value\_ptr ? "TRUE" : "FALSE");  break;  case V\_ASN1\_PRINTABLESTRING:  case V\_ASN1\_IA5STRING:  case V\_ASN1\_UTF8STRING:  // 添加长度检查防止越界  if (tag\_len > 0 && value\_ptr[tag\_len - 1] == '\0') {  fprintf(out, ":%s", value\_ptr);  }  else {  fprintf(out, ":%.\*s", (int)tag\_len, value\_ptr);  }  break;  // 其他case语句...  case V\_ASN1\_ENUMERATED: {  fprintf(out, ":");  for (long i = 0; i < tag\_len && i < 8; i++) {  fprintf(out, "%02X", value\_ptr[i]);  }  if (tag\_len > 8) fprintf(out, "...");  break;  }  case V\_ASN1\_VISIBLESTRING:  case V\_ASN1\_T61STRING:  case V\_ASN1\_BMPSTRING:  case V\_ASN1\_UNIVERSALSTRING:  case V\_ASN1\_NUMERICSTRING:  fprintf(out, ":%.\*s", (int)tag\_len, value\_ptr);  break;  case V\_ASN1\_VIDEOTEXSTRING:  case V\_ASN1\_GRAPHICSTRING:  fprintf(out, "[VIDEO/GRAPHIC DATA]");  break;  default:  // 对于完全未知的类型，至少显示HEX值  fprintf(out, "[UNKNOWN-TAG-%d]:", tag);  for (long i = 0; i < (tag\_len > 8 ? 8 : tag\_len); i++) {  fprintf(out, "%02X", value\_ptr[i]);  }  if (tag\_len > 8) fprintf(out, "...");  }  fprintf(out, "\n");  p += tag\_len; // 移动指针到下一个元素  }  }  \*data = p; // 更新外部指针  }  // BIO转DER辅助函数  long bio\_to\_der(BIO\* bio, unsigned char\*\* der) {  unsigned char buf[4096];  long total = 0;  int len;  \*der = NULL;  BIO\* mem = BIO\_new(BIO\_s\_mem());  if (!mem) return -1;  while ((len = BIO\_read(bio, buf, sizeof(buf))) > 0) {  BIO\_write(mem, buf, len);  total += len;  }  \*der = (unsigned char\*)OPENSSL\_malloc(total);  if (!\*der) {  BIO\_free(mem);  return -1;  }  BIO\_read(mem, \*der, total);  BIO\_free(mem);  return total;  }  // 主解析函数  int parse\_asn1\_file(const char\* filename, FILE\* out) {  BIO\* bio = NULL;  X509\* cert = NULL;  EVP\_PKEY\* pkey = NULL;  unsigned char\* der\_data = NULL;  long der\_len = 0;  int ret = 0;  // 1. 尝试读取 PEM 格式文件  bio = BIO\_new\_file(filename, "r");  if (!bio) {  fprintf(stderr, "Error opening file: %s\n", filename);  if (der\_data) OPENSSL\_free(der\_data);  if (bio) BIO\_free(bio);  if (cert) X509\_free(cert);  if (pkey) EVP\_PKEY\_free(pkey);  return ret;  }  // 2. 尝试解析为各种 PEM 格式  if (strstr(filename, ".pem") || strstr(filename, ".key") || strstr(filename, ".crt") || strstr(filename, ".csr")) {  // 尝试作为证书解析  cert = PEM\_read\_bio\_X509(bio, NULL, NULL, NULL);  if (cert) {  fprintf(out, "=== X509 Certificate Structure ===\n");  der\_len = i2d\_X509(cert, &der\_data);  if (der\_data && der\_len > 0) {  const unsigned char\* p = der\_data;  parse\_asn1(&p, der\_len, 0, out);  ret = 1;  }  }  BIO\_seek(bio, 0);  // 尝试作为私钥解析  pkey = PEM\_read\_bio\_PrivateKey(bio, NULL, NULL, NULL);  if (pkey) {  fprintf(out, "=== Private Key Structure ===\n");  der\_len = i2d\_PrivateKey(pkey, &der\_data);  if (der\_data && der\_len > 0) {  const unsigned char\* p = der\_data;  parse\_asn1(&p, der\_len, 0, out);  ret = 1;  }  }  BIO\_seek(bio, 0);  // 尝试作为CSR解析  X509\_REQ\* req = PEM\_read\_bio\_X509\_REQ(bio, NULL, NULL, NULL);  if (req) {  fprintf(out, "=== Certificate Request (CSR) Structure ===\n");  der\_len = i2d\_X509\_REQ(req, &der\_data);  X509\_REQ\_free(req);  if (der\_data && der\_len > 0) {  const unsigned char\* p = der\_data;  parse\_asn1(&p, der\_len, 0, out);  ret = 1;  }  }  BIO\_seek(bio, 0);  }    // 3. 尝试直接读取 DER 格式，不符合上面归类格式，直接暴力解析  else {  BIO\_free(bio);  bio = BIO\_new\_file(filename, "rb");  if (!bio) {  fprintf(stderr, "Error reopening file in binary mode\n");  if (der\_data) OPENSSL\_free(der\_data);  if (bio) BIO\_free(bio);  if (cert) X509\_free(cert);  if (pkey) EVP\_PKEY\_free(pkey);  return ret;  }  der\_len = bio\_to\_der(bio, &der\_data);  if (der\_len <= 0) {  fprintf(stderr, "Unrecognized file format or parsing error\n");  if (der\_data) OPENSSL\_free(der\_data);  if (bio) BIO\_free(bio);  if (cert) X509\_free(cert);  if (pkey) EVP\_PKEY\_free(pkey);  return ret;  }  fprintf(out, "=== Raw ASN.1 Structure ===\n");  if (der\_data && der\_len > 0) {  const unsigned char\* p = der\_data;  parse\_asn1(&p, der\_len, 0, out);  ret = 1;  }  }  if (der\_data) OPENSSL\_free(der\_data);  if (bio) BIO\_free(bio);  if (cert) X509\_free(cert);  if (pkey) EVP\_PKEY\_free(pkey);  return ret;  } |

### 4. 参考分析 openssl 工具包原码的 s\_cilent 和 s\_server 代码部分，自己编写一个完整的 SSL 客户端和服务器端程序，并能安全通信（配置数字证书），显示执行过程。

|  |
| --- |
| **SSL 客户端和服务端通信流程详解**  要理解 SSL 客户端和服务端的通信模式，我们需要明确 ​**客户端-服务器模型**​ 的基本工作原理，以及 SSL/TLS 协议如何在此模型上构建安全通道。以下是详细的流程说明：  SSL/TLS 通信遵循典型的 ​**客户端-服务器模型**​：   1. ​**服务端**​：    * 持续运行在特定端口上监听（如 443）    * 等待客户端连接请求    * 为每个连接创建独立的 SSL 会话    * 处理客户端请求并返回响应 2. ​**客户端**​：    * 主动连接到服务端的 IP 和端口    * 发起 SSL 握手建立安全通道    * 通过加密通道发送请求/接收响应   下面我使用一张**时序图**来解释两者的通信流程    SSL通信流程始于客户端与服务器**建立TCP连接**后，双方通过**SSL/TLS握手协议**协商安全**参数**：  客户端首先发送ClientHello消息声明支持的协议版本和加密套件，服务器回应ServerHello选定具体算法并发送证书供客户端验证；客户端验证证书有效性后生成预主密钥并用服务器公钥加密发送，双方基于预主密钥推导出相同的会话密钥，通过交换Finish消息确认**握手完成**。  此时安全通道正式建立，后续所有应用数据均通过协商的加密算法进行保密传输，形成端到端的加密通信。整个过程通过非对称加密实现密钥交换，对称加密保障数据传输效率，数字证书确保身份真实性，最终实现保密性、完整性和身份认证三大安全目标。 代码构建思路： 根据我上面的浏览，我想创建一个ssl通信的客户端和服务端，我首先要建立一个能够相互通信的tcp套接字，在不安全的情况下，可以通信，完成之后，再逐步添加安全措施如证书和密钥加解密以及握手 |

#### 实现基础的TCP通信（明文传输）​

|  |
| --- |
| **目标**​：确保客户端和服务器能通过TCP套接字建立连接并交换数据。  服务器端伪代码  // 1. 创建TCP套接字  int server\_fd = socket(AF\_INET, SOCK\_STREAM, 0);  // 2. 绑定地址和端口  bind(server\_fd, (struct sockaddr\*)&address, sizeof(address));  // 3. 监听连接  listen(server\_fd, 5);  // 4. 接受客户端连接  int client\_socket = accept(server\_fd, NULL, NULL);  // 5. 读写数据（明文）  char buffer[1024];  read(client\_socket, buffer, sizeof(buffer)); // 接收数据  write(client\_socket, "Hello from server", 17); // 发送数据  客户端伪代码  // 1. 创建TCP套接字  int sock = socket(AF\_INET, SOCK\_STREAM, 0);  // 2. 连接到服务器  connect(sock, (struct sockaddr\*)&serv\_addr, sizeof(serv\_addr));  // 3. 读写数据（明文）  write(sock, "Hello from client", 17); // 发送数据  read(sock, buffer, sizeof(buffer)); // 接收数据  根据上面的思路可以写出一个比较基本的客户端服务端的双端代码，下面是实验基本的TCP连接：  打开服务端    打开客户端    服务端接收到客户端的连接，能够显示连接的IP和端口  使用双端进行一个简短的**对话**    通过WireShark捕获他们的对话信息，可以看到TCP是不安全的直接以明文的形式传输，    图中**TCP负载payload为明文的传输内容123456**    再比如nice to meet you |

#### 添加OpenSSL安全层

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **目标​：在TCP基础上加入SSL/TLS加密，实现安全通信。**  **改造思路：**  我的原始代码是一个简单的 **TCP 服务器**，它使用 socket()、bind()、listen() 和 accept() 进行基本的网络通信。为了增强安全性，我们使用 **OpenSSL** 添加 **SSL/TLS 加密层**，使通信过程变成 **HTTPS（安全 HTTP）** 或 **SMTPS（安全 SMTP）** 等安全协议。 1. 修改思路（对比 TCP 和 SSL/TLS）  | **步骤** | **TCP 服务器** | **SSL/TLS 服务器** | | --- | --- | --- | | **1. 初始化网络库** | WSAStartup() | WSAStartup() + SSL\_library\_init() | | **2. 创建套接字** | socket() | socket() | | **3. 绑定端口** | bind() | bind() | | **4. 监听连接** | listen() | listen() | | **5. 接受连接** | accept() | accept() + SSL\_new() + SSL\_set\_fd() + SSL\_accept() | | **6. 数据收发** | recv() / send() | SSL\_read() / SSL\_write() | | **7. 关闭连接** | closesocket() | SSL\_shutdown() + closesocket() | | **8. 清理资源** | WSACleanup() | SSL\_CTX\_free() + WSACleanup() |  2. OpenSSL 源代码函数（修改依据） (1) 初始化 OpenSSL  void init\_openssl() {  SSL\_load\_error\_strings(); // 加载 SSL 错误信息  OpenSSL\_add\_ssl\_algorithms(); // 初始化 SSL 算法  }   * **作用**：初始化 OpenSSL 库，加载 SSL/TLS 相关的错误信息和加密算法。   (2) 创建 SSL 上下文（SSL\_CTX）  SSL\_CTX\* create\_context() {  const SSL\_METHOD\* method = SSLv23\_server\_method(); // 支持 TLS 1.0/1.1/1.2  SSL\_CTX\* ctx = SSL\_CTX\_new(method); // 创建 SSL 上下文  if (!ctx) {  ERR\_print\_errors\_fp(stderr);  exit(EXIT\_FAILURE);  }  return ctx;  }   * **作用**：   + SSLv23\_server\_method()：支持 **TLS 1.0/1.1/1.2**（兼容性最好）。   + SSL\_CTX\_new()：创建 SSL 上下文，用于管理证书、私钥和加密方式。   (3) 加载证书和私钥  void configure\_context(SSL\_CTX\* ctx) {  // 加载证书（PEM 格式）  if (SSL\_CTX\_use\_certificate\_file(ctx, "server.crt", SSL\_FILETYPE\_PEM) <= 0) {  ERR\_print\_errors\_fp(stderr);  exit(EXIT\_FAILURE);  }  // 加载私钥（PEM 格式）  if (SSL\_CTX\_use\_PrivateKey\_file(ctx, "server.key", SSL\_FILETYPE\_PEM) <= 0) {  ERR\_print\_errors\_fp(stderr);  exit(EXIT\_FAILURE);  }  }   * **作用**：   + SSL\_CTX\_use\_certificate\_file()：加载 **服务器证书**（server.crt）。   + SSL\_CTX\_use\_PrivateKey\_file()：加载 **私钥**（server.key）。   + 如果文件不存在或格式错误，程序会报错退出。   (4) 接受 SSL 连接  // 1. 接受 TCP 连接  client\_socket = accept(server\_socket, (struct sockaddr\*)&client, &client\_len);  // 2. 创建 SSL 对象并绑定到 socket  ssl = SSL\_new(ctx); // 创建 SSL 对象  SSL\_set\_fd(ssl, client\_socket); // 绑定到 socket  // 3. 进行 SSL 握手  if (SSL\_accept(ssl) <= 0) { // 类似 TCP 的 accept()  ERR\_print\_errors\_fp(stderr);  } else {  printf("SSL connection established\n");  }   * **作用**：   + SSL\_new()：创建一个新的 SSL 会话。   + SSL\_set\_fd()：将 SSL 绑定到 TCP socket。   + SSL\_accept()：进行 **SSL/TLS 握手**（类似 TCP 的 accept()）。   (5) 加密通信（SSL\_read / SSL\_write）  // 接收加密数据（替代 recv()）  int recv\_size = SSL\_read(ssl, buffer, BUFFER\_SIZE);  // 发送加密数据（替代 send()）  SSL\_write(ssl, buffer, strlen(buffer));   * **作用**：   + SSL\_read()：解密客户端发来的数据（替代 recv()）。   + SSL\_write()：加密并发送数据（替代 send()）。   (6) 关闭 SSL 连接  SSL\_shutdown(ssl); // 关闭 SSL 连接  SSL\_free(ssl); // 释放 SSL 资源  closesocket(client\_socket); // 关闭 socket   * **作用**：   + SSL\_shutdown()：安全关闭 SSL 连接（发送 close\_notify）。   + SSL\_free()：释放 SSL 对象。  3. 服务端 vs 客户端对比  | **步骤** | **服务端（Server）** | **客户端（Client）** | | --- | --- | --- | | **初始化 OpenSSL** | SSL\_load\_error\_strings() | SSL\_load\_error\_strings() | | **创建 SSL 上下文** | SSL\_CTX\_new(SSLv23\_server\_method()) | SSL\_CTX\_new(SSLv23\_client\_method()) | | **加载证书** | SSL\_CTX\_use\_certificate\_file() | 可选（双向认证时才需要） | | **加载私钥** | SSL\_CTX\_use\_PrivateKey\_file() | 可选（双向认证时才需要） | | **SSL 握手** | SSL\_accept() | SSL\_connect() | | **数据收发** | SSL\_read() / SSL\_write() | SSL\_read() / SSL\_write() | | **关闭连接** | SSL\_shutdown() | SSL\_shutdown() |  4. 修改后的功能  1. **加密通信**：    * 所有数据经过 **AES、RSA 等加密**，防止中间人攻击（MITM）。 2. **身份验证**：    * 服务器必须提供 **证书（server.crt）**，客户端可以验证服务器身份。    * （可选）客户端也可以提供证书，实现 **双向认证**。 3. **数据完整性**：    * SSL/TLS 使用 **HMAC** 确保数据未被篡改。 4. **兼容 HTTPS**：    * 如果客户端是浏览器，可以直接用 https:// 访问。  5. 如何生成证书和私钥****？**** 使用 OpenSSL 生成自签名证书：  openssl req -x509 -newkey rsa:4096 -keyout server.key -out server.crt -days 365 -nodes   * server.key：私钥文件（必须保密！） * server.crt：证书文件（可以公开）   **此时再次进行通信，使用wireshark进行捕获，看是否成功加密，能够看到的是明文还是密文？**  进行简短的通信    **观察捕获到的信息，可以观察到，此时通信进行了加密，负载信息是加密的乱码，防止流量分析，被动攻击，得到通信信息** |

#### ​客户端验证服务器证书

|  |
| --- |
| **要使客户端能够验证服务端证书并打印证书信息，对代码进行以下修改：**   1. **添加证书验证功能** 2. **添加打印证书信息的函数** 3. **修改SSL上下文创建方式以启用验证**   在上一步中虽然能够进行连接，但是客户端没有对服务器的证书进行验证，这容易导致不安全的链接发生，在本部分将为客户端添加认证并且得到证书的一系列信息来保证通信安全。  当前代码相比原始版本增加了完整的SSL/TLS证书验证功能，这是安全通信中至关重要的部分。下面我将详细阐释证书验证的原理、思路及OpenSSL库的调用方式。 证书验证功能的思路和原理基本思路  1. **建立信任链**：验证服务器证书是否由受信任的CA签发 2. **检查证书有效性**：验证证书是否在有效期内 3. **检查证书用途**：验证证书是否可用于服务器身份验证 4. **检查主机名匹配**：验证证书中的主机名是否与实际连接的主机名匹配  验证原理  1. **信任锚(Trust Anchor)**：使用系统默认的CA证书或指定的CA证书作为信任起点 2. **证书链构建**：从终端证书(服务器证书)回溯到根CA证书 3. **签名验证**：使用上级证书的公钥验证下级证书的签名 4. **基本约束检查**：确保CA证书有权限签发下级证书 5. **密钥用途检查**：确保证书可用于TLS服务器认证 6. **有效期检查**：验证证书是否在有效期内  OpenSSL库的调用方式 1. 初始化SSL上下文并设置验证参数  SSL\_CTX\* ctx = SSL\_CTX\_new(TLS\_client\_method());  SSL\_CTX\_set\_verify(ctx, SSL\_VERIFY\_PEER, NULL); // 启用对等证书验证  SSL\_CTX\_set\_verify\_depth(ctx, 4); // 设置最大证书链深度  2. 加载信任的CA证书  // 加载系统默认CA证书  if (!SSL\_CTX\_set\_default\_verify\_paths(ctx)) {  ERR\_print\_errors\_fp(stderr);  exit(EXIT\_FAILURE);  }  // 额外加载指定的CA证书  if (SSL\_CTX\_load\_verify\_locations(ctx, "server.crt", NULL) != 1) {  fprintf(stderr, "Error loading server certificate as trusted CA\n");  ERR\_print\_errors\_fp(stderr);  exit(EXIT\_FAILURE);  }  3. 建立连接后验证证书  // 获取服务器证书  X509\* cert = SSL\_get\_peer\_certificate(ssl);  if (cert) {  print\_certificate\_info(cert); // 打印证书信息    // 验证证书  long verify\_result = SSL\_get\_verify\_result(ssl);  if (verify\_result == X509\_V\_OK) {  printf("Certificate verification successful\n");  } else {  printf("Certificate verification failed: %s (%ld)\n",  X509\_verify\_cert\_error\_string(verify\_result), verify\_result);  }    X509\_free(cert);  }  4. 证书信息打印功能  代码中的print\_certificate\_info()函数展示了如何从X509证书结构中提取各种信息：   * 主题(Subject)信息 * 颁发者(Issuer)信息 * 有效期 * 序列号 * 签名算法 * 公钥信息 * 扩展信息  新增功能的关键点  1. **SSL\_VERIFY\_PEER标志**：启用对服务器证书的验证 2. **证书链验证**：OpenSSL会自动验证整个证书链的有效性 3. **验证结果检查**：通过SSL\_get\_verify\_result()获取详细验证结果 4. **证书信息提取**：通过X509结构体获取证书的详细信息 5. **信任存储管理**：使用系统默认CA证书和自定义CA证书  安全意义 增加的证书验证功能确保了：   1. 通信双方的身份真实性 2. 数据传输的机密性 3. 防止中间人攻击 4. 确保连接到的确实是目标服务器而非假冒服务器  具体实战演示： 打开客户端连接服务器    **当连接建立后会打印成功验证的证书的各种信息**      **在这种情况下，即为验证成功后面可以通信。**  **而当客户端与服务器的证书不匹配，没有通过验证，则连接断开** |

#### 服务端验证客户端证书（最终实现双向认证）

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **开始之前：**由于双向认证需要ca产生的客户端和服务端的证书，就算是产生自签名证书也很麻烦，这里我写了一个一键**产生证书的脚本：**    **为产生证书节省时间，供大家使用。**  **1. 双向认证（mTLS）的内在逻辑**  双向认证（Mutual TLS）要求**客户端和服务端**互相验证对方的证书，确保双方身份可信。其核心逻辑如下：   1. **服务端**：    * 提供自己的证书（server.crt）给客户端。    * **强制要求客户端提供证书**（SSL\_VERIFY\_PEER | SSL\_VERIFY\_FAIL\_IF\_NO\_PEER\_CERT）。    * 使用ca.crt验证客户端证书是否由受信任的CA签发。 2. **客户端**：    * 提供自己的证书（client.crt）给服务端。    * 使用ca.crt验证服务端证书是否由受信任的CA签发。 3. **OpenSSL底层流程**：    * 双方在SSL/TLS握手阶段交换证书。    * 证书链验证（X509\_verify\_cert）确保证书未被篡改且由受信任CA签发。    * 最终建立加密通道。   **2. 代码运行步骤（基于OpenSSL库）** ****（1）服务端（Server）**** // 1. 初始化SSL上下文（强制要求客户端证书）  SSL\_CTX\_set\_verify(ctx, SSL\_VERIFY\_PEER | SSL\_VERIFY\_FAIL\_IF\_NO\_PEER\_CERT, verify\_callback);  // 2. 加载服务端证书和私钥  SSL\_CTX\_use\_certificate\_file(ctx, "server.crt", SSL\_FILETYPE\_PEM);  SSL\_CTX\_use\_PrivateKey\_file(ctx, "server.key", SSL\_FILETYPE\_PEM);  // 3. 加载CA证书（用于验证客户端证书）  SSL\_CTX\_load\_verify\_locations(ctx, "ca.crt", NULL);  // 4. 设置客户端CA列表（可选，用于告知客户端哪些CA受信任）  STACK\_OF(X509\_NAME) \*ca\_list = SSL\_load\_client\_CA\_file("ca.crt");  SSL\_CTX\_set\_client\_CA\_list(ctx, ca\_list);  // 5. 在SSL握手后获取客户端证书  X509 \*client\_cert = SSL\_get\_peer\_certificate(ssl);  if (client\_cert) {  // 打印客户端证书信息  X509\_NAME\_print\_ex\_fp(stdout, X509\_get\_subject\_name(client\_cert), 0, XN\_FLAG\_ONELINE);  X509\_free(client\_cert);  } ****（2）客户端（Client）**** // 1. 初始化SSL上下文（验证服务端证书）  SSL\_CTX\_set\_verify(ctx, SSL\_VERIFY\_PEER, NULL); // 单向认证只需验证服务端  SSL\_CTX\_load\_verify\_locations(ctx, "ca.crt", NULL); // 加载CA证书  // 2. 加载客户端证书和私钥（双向认证才需要）  SSL\_CTX\_use\_certificate\_file(ctx, "client.crt", SSL\_FILETYPE\_PEM);  SSL\_CTX\_use\_PrivateKey\_file(ctx, "client.key", SSL\_FILETYPE\_PEM);  // 3. 在SSL握手后获取服务端证书  X509 \*server\_cert = SSL\_get\_peer\_certificate(ssl);  if (server\_cert) {  // 打印服务端证书信息  X509\_NAME\_print\_ex\_fp(stdout, X509\_get\_subject\_name(server\_cert), 0, XN\_FLAG\_ONELINE);  X509\_free(server\_cert);  }  **3. 双向认证 vs. 单向认证（关键区别）**   | **特性** | **双向认证（mTLS）** | **单向认证（普通TLS）** | | --- | --- | --- | | **证书交换** | 客户端和服务端都提供证书 | 仅服务端提供证书 | | **OpenSSL配置** | `SSL\_VERIFY\_PEER | SSL\_VERIFY\_FAIL\_IF\_NO\_PEER\_CERT` | | **CA证书用途** | ca.crt验证**双方**证书 | ca.crt仅验证**服务端**证书 | | **客户端代码** | 必须加载client.crt和client.key | 不需要客户端证书 | | **典型应用场景** | 银行API、物联网设备认证 | 普通HTTPS网站 |   **4. 关键OpenSSL函数解析**   | **函数** | **作用** | | --- | --- | | SSL\_CTX\_set\_verify() | 设置证书验证模式（单向/双向） | | SSL\_CTX\_load\_verify\_locations() | 加载CA证书（ca.crt）用于验证对方证书 | | SSL\_get\_peer\_certificate() | 获取对方证书（双向认证时才能拿到） | | X509\_verify\_cert() | OpenSSL内部函数，验证证书链是否可信 | | SSL\_CTX\_use\_certificate\_file() | 加载自己的证书（服务端用server.crt，客户端用client.crt） |   **文件配置：**  现在我们要修改成在双方能够互相认证，此时客户端和服务端都要有自己的证书和密钥，每个程序所需如下：   1. **​服务端需要:**    * **server.crt - 服务端证书**    * **server.key - 服务端私钥**    * **ca.crt - CA根证书（用于验证客户端证书）** 2. **​客户端需要:**    * **client.crt - 客户端证书**    * **client.key - 客户端私钥**    * **ca.crt - CA根证书（用于验证服务器证书）**   **桥梁：ca.crt - CA根证书**  **ca.crt 的作用​**   * **​**验证证书链​：确保对方（客户端或服务端）提供的证书是由受信任的CA签发的。 * ​防止中间人攻击​：如果证书不是由ca.crt信任的CA签发，连接会被拒绝。 * ​双向认证的基石​：服务端和客户端都使用同一个ca.crt验证对方的证书。   **这里ca也是模拟实践存在的ca机构**  **程序演示：**  **连接之后打印证书（服务端只打印简短的信息表示成功，客户端由于一次只能连接一个服务端所以打印的服务端证书较为完整）**  **服务端**    **客户端** |

#### 显示详细的执行过程

|  |
| --- |
| **在双端功能完成的差不多的时候，根据流程时序图，在每一步执行过程中都打印步骤，使程序更加清晰可见。**  **时序图：**    这一步只是在各个步骤中加入了printf代码，因此只展示效果图：  **这里我基本上在每一个小步骤上都加上了打印效果，使程序运行逻辑更加清晰。**  **服务端：**      **客户端：** |

## 编程实验心得

|  |
| --- |
| 通过本次安全协议与标准课程的实验，我对OpenSSL密码库的使用和原理有了更深入的理解，尤其是在实际编程中实现了RSA密钥生成、证书管理、PKCS#12打包以及ASN.1解析等核心功能。以下是我的几点心得体会：  **1. ​理论与实践的结合​**   * 课堂上学习的RSA算法、数字证书、PKI等理论知识，在实验中通过OpenSSL库得到了实际验证。例如，通过代码实现密钥对生成时，理解了密钥长度（如2048位）对安全性的直接影响；在证书请求（CSR）生成过程中，明确了Subject字段（国家、组织等）的实际作用。 * OpenSSL的封装程度较高，但某些功能（如ASN.1解析）需要深入底层数据结构，这让我意识到密码学不仅是数学理论，更是工程实践的结合。   **2. ​工具与代码的双重掌握​**   * 实验中既使用了OpenSSL命令行工具（如genrsa、req），也通过编程调用API实现相同功能。对比发现：   + ​**命令行工具**适合快速验证和简单操作，例如生成自签名证书只需一行命令。   + ​**编程实现**更灵活，能定制化流程（如交互式输入Subject信息），但需处理内存管理、错误检查等细节（如EVP\_PKEY\_CTX的释放）。 * 例如，PKCS#12文件的生成与解析中，编程时需要手动处理密码加密、证书链嵌套等逻辑，而命令行工具通过参数即可完成。   **3. ​调试与问题解决能力提升​**   * ​**ASN.1解析**是最具挑战的部分。起初对TLV（Tag-Length-Value）格式不熟悉，导致解析二进制DER文件时出现类型识别错误。通过查阅RFC文档和OpenSSL源码，逐步完善了类型映射表（如区分SEQUENCE和OCTET\_STRING）。 * ​**内存管理**是易错点。例如，未及时释放X509\_REQ结构体会导致内存泄漏，通过Valgrind工具检测并修复了此类问题。 * 错误处理机制也很关键。OpenSSL的许多函数返回0表示失败，但错误原因需通过ERR\_get\_error()获取，这让我养成了检查返回值并打印错误信息的习惯。   **4. ​安全通信的实践意义​**   * 在SSL客户端/服务器实验中，配置数字证书（如自签名证书）和双向验证时，理解了HTTPS通信的底层原理。例如：   + 服务器证书的CN必须与域名匹配，否则客户端会报错。   + 证书链的完整性验证（如中间CA证书）是安全通信的基础。 * 通过抓包分析SSL握手过程（如ClientHello、ServerKeyExchange），直观看到了对称密钥协商的步骤。   **5. ​对密码学工程的认知​**   * OpenSSL的API设计体现了分层思想：高层（如PEM\_write系列函数）简化操作，底层（如ASN.1解析）暴露细节。这让我意识到密码学库需要平衡易用性与灵活性。 * 安全性不仅依赖算法强度，更取决于实现细节。例如，PKCS#12文件的密码保护若迭代次数过低，容易被暴力破解。   **6. ​未来学习方向​**   * 进一步研究OpenSSL的异步I/O和性能优化（如SSL会话复用）。 * 探索更复杂的场景（如OCSP证书状态检查、TLS 1.3的0-RTT特性）。 |