TLS Project实验报告

1. 实验目的

- 了解TLS协议内容
- 理解TLS连接过程
- 了解TLS如何保证加密通信
- 模拟TLS握手和加密传输过程

2. 实验介绍

基于**TLS1.2**实现的TLS握手过程和加密数据传输过程模拟

2.1 实验环境

- Windows
- python 3.10.13
    - cryptography 41.0.5：加解密算法库

2.2 Manual

首先启动服务端并监听端口，然后启动客户端连接到本地IP`127.0.0.1`和服务端端口

- **Client**:

  运行client：`python ./main_client [options]`

  命令行参数：

```
Options：
-a，服务端地址
-p，服务端端口
-c，选择加密算法套件，格式:[KeyExchange:symmetric:hash]
-E，选择密钥交换算法
-S，选择对称加密算法
-H，选择hash算法
-m，发送到客户端的消息
```

  必须提供的命令行参数：`-a`、`-p`、`-m`，及设置服务端的地址和端口，以及待发送的消息

  可选参数：`-c`、`-E`、`-S`、`-H`，用于选择加密算法套件进行模拟，如果不提供，默认选择的算法套件为：

```
TLS_RSA_WITH_AES_256_CBC_SHA256
```

  即RSA密钥交换算法，AES_256_CBC模式对称加密算法和SHA256消息摘要算法

参数`-c`可一次指定所有算法，格式为**-c KeyExchange:symmetric:hash**

- KeyExchange：密钥交换算法
- symmetric：对称加密算法
- hash：消息摘要算法

其他参数可单独设置对应的算法，未设置的算法为默认值，可设置的算法参数参考实现的加密算法

> 可在class client和class server的supported_cipher_suites属性中定义多个默认的算法
>
> 未在命令行参数选择算法套件时，服务端会根据自身的默认支持算法套件在客户端提供的默认支持算法套件选择第一个匹配项
>
> 可选的默认算法参考TLS_type.py/class CipherSuite

- **Server**:

  运行server：`python ./main_server [options]`

  命令行参数：

```
Options:
-p, 监听端口
-c, 选择加密算法套件，格式:[KeyExchange:symmetric:hash]
-E, 选择密钥交换算法
-S, 选择对称加密算法
-H, 选择hash算法
```

  必须提供的命令行参数：`-a`，及设置服务端监听端口

  可选参数和使用方法与client相同，但要求**server选择的算法套件和client选择的算法套件必须相同**，否则无法协商出一致的算法套件导致连接失败

2.3 项目简介

- `client.py` `main_client.py`

  `client.py`文件中定义了client类，描述了TLS连接和传输信息过程中客户端的执行流程

  `main_client.py`文件是运行客户端的主程序

- `server.py` `main_server.py`

  `server.py`文件中定义了server类，描述了TLS连接和传输信息过程中服务端的执行流程

  `main_server.py`文件是运行服务端的主程序

- `protocol`

  定义了基于TLS1.2的各层协议

  - `handshake_protocol.py`

    定义握手过程中各个握手消息的数据结构，比如ClientHello、ServerHello等

  - `record_protocol.py`

    定义了TLS的记录层协议，用于封装其他上层协议并在TCP网络中传输

    其中`class RecordLayer`是基本的封装结构，用于封装其他上层协议；`wrap()`和`extract()`为对应的打包和解包操作

`class TLSPlaintext`和`class TLSCiphertext`是在TCP网络中传输的字节流结构，分别用于传输明文数据和加密后的数据，其中可以封装多个`RecordLayer`进行传输，比如在服务端回复ClientHello时，将回复的ServerHello、Certificate、ServerHelloDone一同打包并发送给客户端；其中`wrap()`操作用于打包`RecordLayer`，`extract()`操作用于分割字节流数据

> TLS1.2中规定TLSPlaintext和TLSCiphertext封装的内容长度不超过2^{14}字节，但在模拟过程中，消息长度默认不会超过2^{14}字节，因此没有进一步做分片处理

  - `application_data_protocol.py`

    定义了加密信息的数据结构，其中包含加密信息和消息验证码

  - `alert_protocol.py`

    警告协议，因为模拟连接过程的错误消息都通过终端展示，因此并未实现

- `src`

  项目资源文件，主要包含密钥和证书

  - `key`
  - `certificate`

- `util`
  - `crypt.py`
    - 实现各种对称加密和非对称加密算法
    - 主密钥生成和加解密实现
    - 证书生成与验证
    - 数字签名生成与验证
  - `TLS_type.py`

    定义了TLS1.2中的各种类型

  - `utils.py`

    一些工具函数

2.4 实现的加密算法

- 密钥交换算法：
  - RSA：基于RSA的密钥交换算法
  - DHE：基于Diffie-Hellman算法的密钥交换算法

    密钥仅在连接过程中生成，因此具有前向保密性*forward secrecy*
    - DHE_DSS
    - DHE_RSA
  - ECDHE：基于椭圆曲线的DHE算法
    - ECDHE_DSS
    - ECDHE_RSA

> 因为时间有限且只是为了模拟TLS连接过程，椭圆曲线只选择了SECP384R1这1种，客户端和服务端不会再协商具体的椭圆曲线，因此在参数中无需声明
>
> DHE算法生成密钥对的过程可能比较慢

- 签名算法：

签名算法仅实现了**RSA**和**DSA**签名算法，因此上面的密钥交换算法中对应的签名算法只有这两种

- 对称加密算法：
  - AES128_CBC：AES 128位密钥对称加密算法
  - AES256_CBC：AES 256位密钥对称加密算法
  - 3DES_EDE_CBC：3DES对称加密算法
  - RC4_128：RC4 128位密钥对称加密算法

> RC4对称加密算法的可靠性较差，除RC4算法外，其他三个算法均采用CBC模式加密

- 消息摘要算法：
  - MD5
  - SHA1
  - SHA256
  - SHA224
  - SHA384
  - SHA512
- 证书格式
  - RSA_SIGN：RSA密钥签名证书
  - DSA_SIGN：DSA密钥签名证书

> 目前只这两种签名方式的证书类型，证书的hash算法默认为SHA256

2.5 会话密钥生成

参考TLS1.2的会话密钥生成过程，实现了基于**HMAC**(Hash-base Message authentication codes)的会话密钥生成

```
P_hash(secret, seed) = HMAC_hash(secret, A(1) + seed) +
                       HMAC_hash(secret, A(2) + seed) +
                       HMAC_hash(secret, A(3) + seed) + ...

A(0) = seed
A(i) = HMAC_hash(secret, A(i-1))
```

HMAC的生成基于SHA256消息摘要算法以提供足够的安全性

首先，根据选择的对称加密算法确定会话密钥长度，然后以客户端随机数和服务端随机数作为secret，生成预主密钥的HMAC，接着以预主密钥的HMAC作为secret进行循环迭代，直到生成的随机值足够密钥长度，每次迭代生成上一个生成值加上已生成HMAC的个数（增加随机性）的HMAC值（第一个为空），得到下一个HMAC值，最后将所有生成的HMAC拼接在一起并返回密钥长度对应的随机值作为主密钥

```
def generate_master_secret(pre_master_secret, client_random, server_random,
algorithm):
    if algorithm == SymmetricAlgorithm.RC4_128 or \
            algorithm == SymmetricAlgorithm.AES_128_CBC or \
            algorithm == SymmetricAlgorithm.Triple_DES_EDE_CBC:
        length = 16
    elif algorithm == SymmetricAlgorithm.AES_256_CBC:
        length = 32
```

```
    else:
        length = 0

    h1 = hmac.HMAC(client_random + server_random, hashes.SHA256())
    h1.update(pre_master_secret)
    pre_master_secret_hmac = h1.finalize()
    output = [b""]
    counter = 1

    while hashes.SHA256.digest_size * (len(output) - 1) < length:
        h2 = hmac.HMAC(pre_master_secret_hmac, hashes.SHA256())
        h2.update(output[-1])
        h2.update(bytes(counter))
        output.append(h2.finalize())
        counter += 1

    return b"".join(output)[: length]
```

3. 实验过程

3. 1 建立连接

服务端启动后会等待客户端发送握手请求，客户端启动后会向服务端发送ClientHello请求连接

服务端收到客户端消息后判断是ClientHello消息后，双方都通过`handshake()`函数进行下一步的握手操作

3.2 握手过程

客户端与服务端的握手流程如下：

```
Client                                               Server

ClientHello                  -------->
                                                ServerHello
                                                Certificate*
                                          ServerKeyExchange*
                                         CertificateRequest*
                              <--------      ServerHelloDone
Certificate*
ClientKeyExchange
CertificateVerify*
[ChangeCipherSpec]
Finished                     -------->
                                          [ChangeCipherSpec]
                              <--------             Finished
Application Data             <------->     Application Data
```

协商算法套件

1. 客户端发送的ClientHello消息中包含

- 客户端的版本`0x0303`
- 客户端的随机数`client_random`
- 客户端支持的算法套件`cipher_suites`，这里发送的是默认支持套件，连接过程使用的套件可由命令行参数指定
- 会话ID`session_id`为空值，由服务端生成
- 其他信息：`compression_methods`和`extensions`未实现因此为空值

2. 服务端收到ClientHello后，先检测客户端TLS版本是否被支持，然后获取客户端的随机值，会话ID为空因此服务端需要生成一个随机的会话ID

   然后调用`select_cipher_suite()`从客户端支持的算法套件中选择本次会话过程中使用的算法套件；在客户端和服务端的默认支持算法套件`supported_cipher_suites`中，所有套件按推荐使用程度降序排列，因此服务端会选择第一个匹配的算法套件使用

3. 协商出所需信息后，服务端发送ServerHello告知客户端协商结果，包括：
   - 服务端随机数`server_random`
   - 会话ID`session_id`
   - 服务端选择算法套件`selected_cipher_suite`
4. 然后服务端根据选择的密钥交换算法决定是否要发送证书，本项目**支持的密钥交换算法都需要发送证书**

   服务端加载对应签名算法的证书并发送Certificate消息

   > 因为只是模拟过程，客户端和服务端发送的证书都只有一个，所以只需要验证一个证书

5. 根据密钥交换算法选择是否发送ServerKeyExchange消息，**除RSA算法外，其余支持的算法都需要发送**

   RSA密钥交换算法的公钥随Certificate消息中的证书一同发送，客户端验证证书后即可获得RSA公钥

   其余算法都是基于DH密钥交换算法来协商密钥，因此需要通过ServerKeyExchange消息发送DH参数给客户端进行密钥协商
   - DHE算法在每次会话连接过程中生成密钥对和参数，并将参数和服务端公钥发送给客户端，客户端收到参数后根据该参数生成自己的密钥对，通过客户端私钥和收到的服务端公钥利用DH算法协商出预主密钥`pre_master_secret`
   - ECDHE算法和DHE算法协商密钥的过程相同，只是传递的参数是基于椭圆曲线加密算法

   此外，在ServerKeyExchange消息中还要对发送的参数和公钥进行签名，签名数据为客户端随机数+服务端随机数+参数+公钥

6. 默认要求服务端需要验证客户端证书，因此服务端需要发送CertificateRequest消息

   消息中包括：
   - 服务端支持证书类型，这里只支持`RSA_SIGN`和`DSA_SIGN`
   - 支持的签名算法
   - 支持的消息摘要算法，证书消息摘要算法默认为SHA256，因此这里只有SHA256
7. 最后发送ServerHelloDone表示服务端协商消息发送完毕

以上所有服务端发送的消息通过`RecordLayer`层打包为字节流，然后通过`TLSPlaintext`整合为一个TCP包，调用`send_tls_packages`发送到客户端

客户端通过`recv_tls_packages()`收到字节流消息后，先利用`TLSPlaintext`的`extract()`操作将字节流分割为各个消息的字节流，然后调用`RecordLayer`的`extract()`操作将字节流转换为对应的握手消息数据结构

8. 客户端收到ServerHello后，从中获取服务端随机数，会话ID，服务端选择的算法套件，从而确定本次握手采用的加密算法等信息
9. 收到服务端的Certificate消息后，客户端从中提取服务端证书并进行验证，验证成功后提取证书中的公钥，用于后续的签名验证
10. 收到ServerKeyExchange消息后，提取其中的参数和DH公钥，然后验证签名；根据协商确定的密钥交换算法，客户端使用参数生成自己的密钥对
11. 收到CertificateRequest后，客户端提取其中服务端的证书支持信息，调用`select_certificate_type_and_signature_algorithms()`选择客户端发送的证书类型，因为支持证书类型有限，因此最终客户端还是会根据选择的签名算法发送对应的证书
12. 收到ServerHelloDone消息后表示服务端消息结束
13. 客户端根据之前选择的证书类型，加载对应证书并发送Certificate消息到服务端
14. 然后客户端根据协商的密钥交换算法，生成预主密钥，然后通过ClientKeyExchange发送消息给服务端供其生成相同的预主密钥，不同密钥协商算法发送的消息为：
    - RSA：版本信息加上随机生成的46byte随机数，共48byte的随机数作为预主密钥`pre_master_secret`，然后使用RSA私钥加密后发送给服务端
    - DHE、ECDHE：通过ServerKeyExchange提供的参数生成客户端的DH密钥对，利用生成的私钥和收到的服务端DH公钥生成预主密钥，然后将自己的DH公钥发送给服务端
15. 接着客户端需要发送CertificateVerify消息证明其拥有该证书，其中包含的消息为此前所有握手消息的消息摘要(不包括这条消息，通过`handshake_message`记录之前收到和发送的消息)
16. 最后客户端发送ChangeCipherSpec消息表示接下来的消息需要经过会话密钥加密，并根据此前的客户端随机数`client_random`、服务端随机数`server_random`以及预主密钥`pre_master_secret`来生成会话密钥`master_secret`

    然后发送Finished表示客户端握手结束，其中的消息为此前所有握手消息的消息摘要，但该消息摘会经过会话密钥加密发送

以上客户端发送消息同样经过`TLSPlaintext`打包发送

17. 服务端收到客户端的Certificate后，验证其证书并提取其中公钥
18. 收到ClientKeyExchange消息后，获取其中的消息，对不同的密钥协商算法：
    - RSA：利用服务端证书获取的公钥解密获取预主密钥
    - DHE、ECDHE：利用其中的DH公钥和自己的DH私钥协商出预主密钥
19. 收到CertificateVerify后利用服务端保存的握手消息验证签名
20. 收到ChangeCipherSpec和Finished后，利用服务端生成的主密钥解密签名，然后利用服务端的握手消息记录验证签名
21. 验证成功后服务端发送Finished表明握手完成，之后的消息发送都要经过主密钥加密

3.3 加密传输过程

完成握手后，客户端调用`send_encrypt_package()`发送被主密钥加密的消息

待发送的明文消息会先调用`generate_hmac()`生成HMAC消息验证码，签名消息为客户端随机数+服务端随机数+明文长度+明文，然后使用SHA256进行签名并附在明文消息之后，最后对这个消息使用主密钥加密生成密文

生成的密文通过`TLSCiphertext`打包为字节流发送给服务端

服务端收到密文后解包并使用自己的主密钥解密，然后验证HMAC验证收到的消息

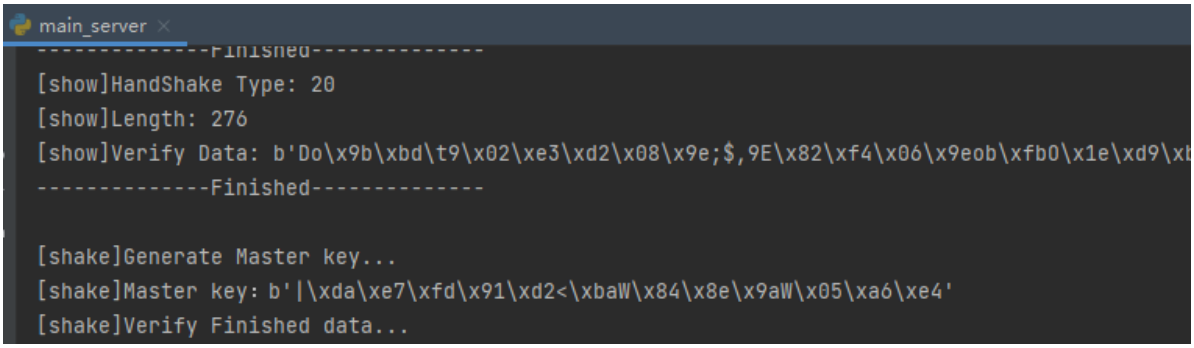
4. 实验结果

示例参数：

服务端：`python ./main_server.py -p 1234 -c ECDHE_RSA:3DES_EDE_CBC:SHA1`

客户端：`python ./main_client.py -a 127.0.0.1 -p 1234 -c ECDHE_RSA:3DES_EDE_CBC:SHA1 -m "finished"`

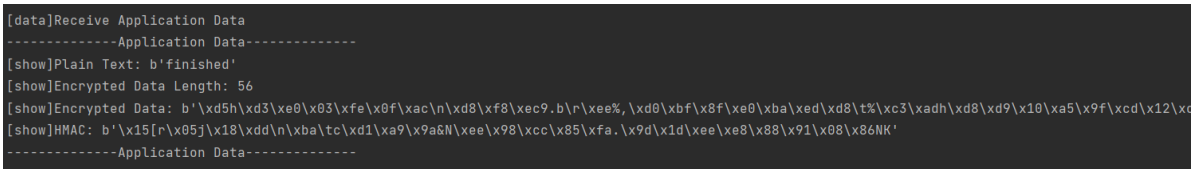
服务端协商生成的主密钥



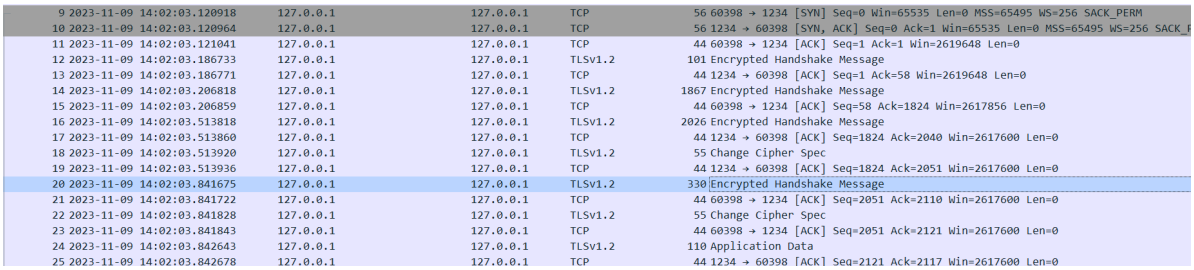
客户端协商生成的主密钥

协商出的密钥完全相同

服务端收到客户端的加密信息并解密：



使用wireshark抓包



wireshark识别出数据包为TLS1.2，但可能因为handshake内部协议实现的比较简单和TLS1.2标准还有差距，或者是多个消息被打包在一起发送导致wireshark没有识别出来，被当作Encrypted Handshake Message

查看第一个TLS数据包，其内容和客户端发送的ClientHello内容相同

```
Transport Layer Security
  TLSv1.2 Record Layer: Handshake Protocol: Encrypted Handshake Message
      Content Type: Handshake (22)
      Version: TLS 1.2 (0x0303)
      Length: 52
      Handshake Protocol: Encrypted Handshake Message
```

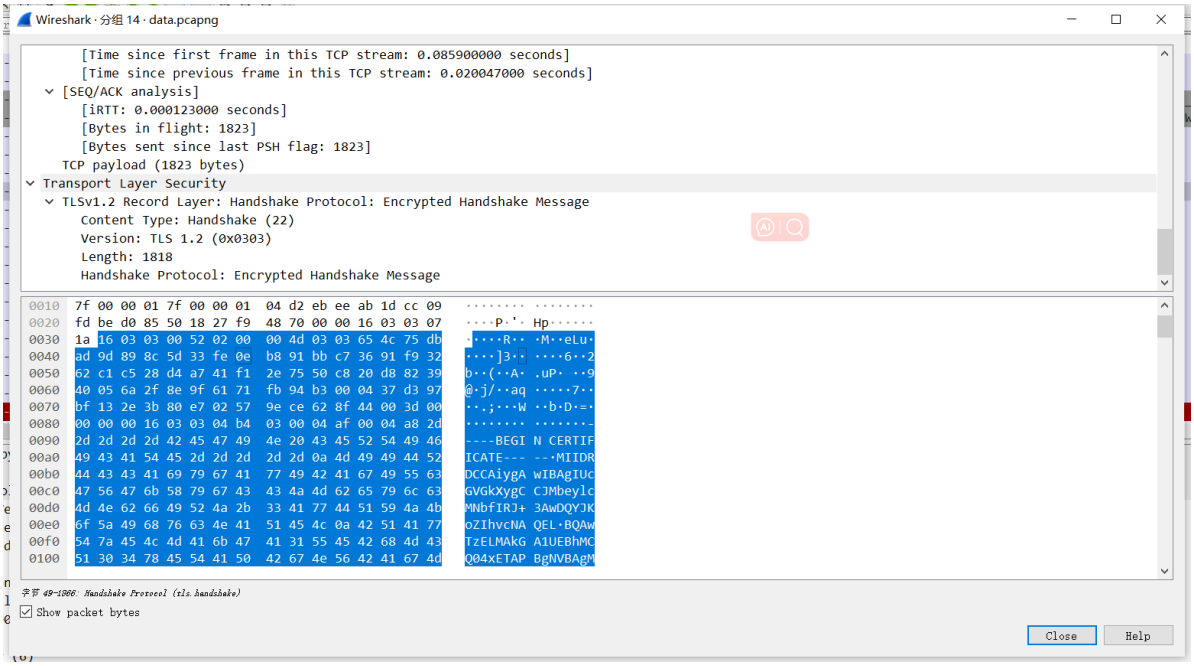
```
0000  02 00 00 00 45 00 00 61  03 7d 40 00 80 06 00 00   ····E··a ·}@·····
0010  7f 00 00 01 7f 00 00 01  eb ee 04 d2 fd be d0 4c   ···· ···· ·····L
0020  ab 1d cc 09 50 18 27 f9  37 9b 00 00 16 03 03 00   ····P·'· 7·····
0030  34 16 03 03 00 34 01 00  00 2f 03 03 65 4c 75 db   4····4·· ·/··eLu·
0040  e7 83 fa f9 3f eb 4b f0  c9 aa 1b 22 bc 3c 61 ce   ····?·K· ···"·<a·
0050  cf 74 44 a1 96 94 27 10  63 be 06 b3 00 00 02 00   ·tD···'· c·····
0060  3d 01 00 00 00                                     =····
```

```
[shake]Send Client Hello

[wrap]Record Content Type: 22
[wrap]Record length: 52
[wrap]Record raw: b'\x16\x03\x03\x004\x01\x00\x00/\x03\x03eLu\xdb\xe7\x83\xfa\xf9?\xebK\xf0\xc9\xaa\x1b"\xbc<a\xce\xcftD\xa1\x96\x94\'\x10c\xbe\x06\xb3\x00\x00

-------------Client Hello-------------
[show]HandShake Type: 1
[show]Version: 0x303
[show]Length: 47
[show]Random: b'eLu\xdb\xe7\x83\xfa\xf9?\xebK\xf0\xc9\xaa\x1b"\xbc<a\xce\xcftD\xa1\x96\x94\'\x10c\xbe\x06\xb3'
[show]Session ID Length: 0
[show]Session ID: None
[show]Cipher Suites Length: 2
[show]Cipher Suites: [61]
[show]Compress Method Length: 1
[show]Compress Methods: [0]
[show]Extensions Length: 0
[show]Extensions: None
-------------Client Hello-------------
```

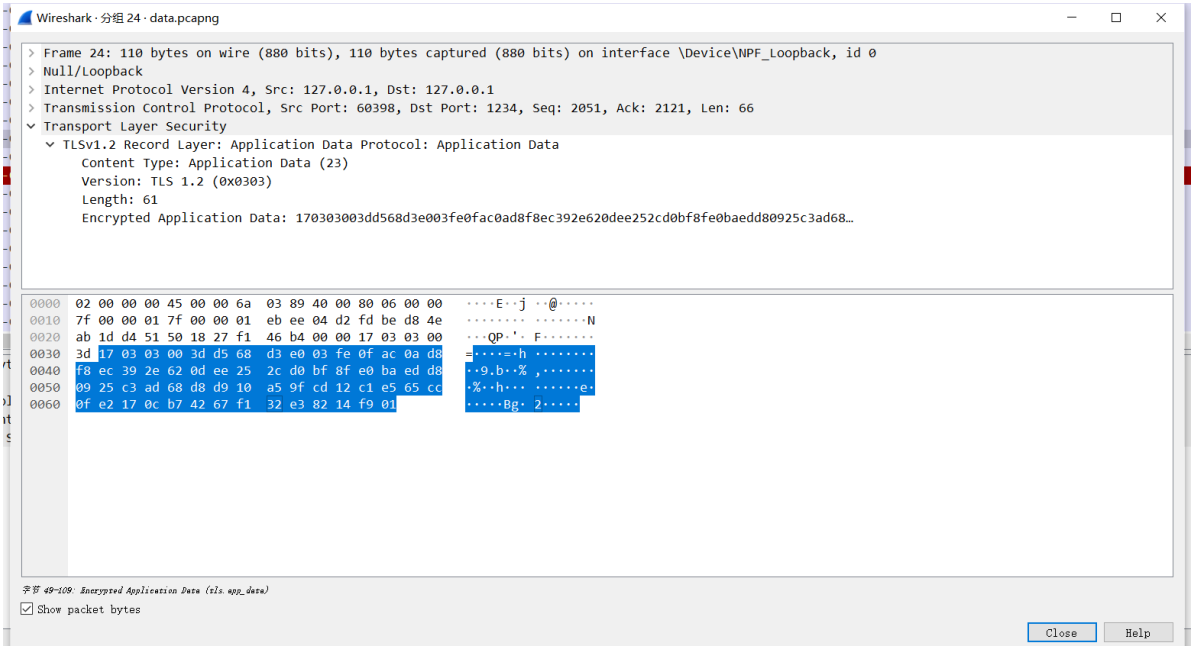
查看第二个TLS数据包，可以明显看出其中包含服务端的证书明文信息



server_rsa_cert.pem | record_protocol.py | application_data_protocol.py | cry

```
-----BEGIN CERTIFICATE-----
MIIDRDCCAiygAwIBAgIUcGVGkXygCCJMbeylcMNbfIRJ+3AwDQYJKoZIhvcNAQEL
BQAwTzELMAkGA1UEBhMCQ04xETAPBgNVBAgMCFNoYW5naGFpMREwDwYDVQQHDAhT
aGFuZ0hhaTEMMAoGA1UECgwDRkRVMQwwCgYDVQQDDANGRFUwHhcNMjMxMTA3MDg1
ODI5WhcNMjMxMTE3MDg1ODI5WjBPMQswCQYDVQQGEwJDTjERMA8GA1UECAwIU2hh
bmdoYWkxETAPBgNVBAcMCFNoYW5nSGFpMQwwCgYDVQQKDANGRFUxDDAKBgNVBAMM
A0ZEVTCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBALK98YSBt/A7esgj
CQNHO0WZCwxMdF/0xkWND5vnhFApT6/Ljnpu73u7PMWNJMBI8HhPphTtP5jXXYvz
0rLQruheFfpFvqJH+qy/Pn/J5CC1aKrpfJw3o657zBLqDMDRQpMrHw/QMzJZBzZW
OGPZUBQBxDvRtXuQtlCTYkeJzQP0exrBHUIj6ucjll9kEuPPjLzLGLENDhNHGYkw
JUQoISsFrLVvlDz1WTqBXfQOTCdp2Hl34t6UpqBQnQH/k05hFLb2wUwTOW/3fFsx
8DtBKLm/+QWwdnp9OoiiQZ1ld+fznTyNNlEstc2jPHoozcT7V3fkvKK3dZ0yonl7
R2I6rbkCAwEAAaMYMBYwFAYDVR0RBA0wC4IJbG9jYWxob3N0MA0GCSqGSIb3DQEB
CwUAA4IBAQAs2ROui0hoTNEoSlt6WQ6lRRFuHv98bGtB6rwUNwaiq1e5xCZggc34
TVXImNclDDMphH/vB7v/J8Q3Sn2K4CEhsVZPzTH2A+1cAiGCCwZK44VkFOewZEgg
RktPXGOBmsfk/1Mi1fE19seLoG5mxHREu1PG8ndDNAb72ZwVzyo1npndDOU4z2+o
gUUAXDkWU5vI6pqlAox9T04zcMU9Gm4wJCSlImrFHhrvRLCpvLDMZ4tfPO11NbJX
jw2y9xYy99SMSMNXdrmcqEfECPBCLmoPrXXtWo0YXJNLx8lCd9d7w1o2Rhzf5wqN
nIw7KGYtyzKBxxtMQ79l4J6nDalvjJ0u
-----END CERTIFICATE-----
```

查看最后一个Application Data包，可以看到其中包含客户端发送给服务端的加密消息




```
[wrap]Record Content Type: 23
[wrap]Record length: 61
[wrap]Record raw: b'\x17\x03\x03\x00=\xd5h\xd3\xe0\x03\xfe\x0f\xac\n\xd8\xf8\xec9.b\r\xee%,\xd0\xbf\x8f\xe0\xba\xed\xd8\t%\xc3\xadh\xd8\xd9\x10\xa5\x9f\xcd\x12\

-------------Application Data-------------
[show]Plain Text: b'finished'
[show]Encrypted Data Length: 56
[show]Encrypted Data: b'\xd5h\xd3\xe0\x03\xfe\x0f\xac\n\xd8\xf8\xec9.b\r\xee%,\xd0\xbf\x8f\xe0\xba\xed\xd8\t%\xc3\xadh\xd8\xd9\x10\xa5\x9f\xcd\x12\xc1\xe5e\xcc\
[show]HMAC: b'\x15[r\x05j\x18\xdd\n\xba\tc\xd1\xa9\x9a&N\xee\x98\xcc\x85\xfa.\x9d\x1d\xee\xe8\x88\x91\x08\x86NK'
-------------Application Data-------------
```

5. Heartbleed Bug

Heartbleed Bug 是OpenSSL在实现TLS协议时出现的一个严重安全漏洞，该漏洞使得攻击者能够从使用TLS加密的通信中读取内存中的数据

漏洞出现在OpenSSLTLS的心跳扩展（Heartbeat Extension）实现。Heartbeat Extension是一个用于维持和测试TLS连接的拓展，发送方发送一方发送一个心跳请求后，接收方必须返回相同的数据以验证连接是否仍然活动

但OpenSSL在实现时没有正确验证发送的心跳请求的长度，攻击者可以发送一个包含恶意构造的心跳请求，请求的数据长度字段长度大于实际数据的长度。接收方在响应时会返回指定长度的数据，但实际上，由于未正确验证长度，它可能返回大于实际请求长度的数据，其中就可能包含敏感数据，比如会话密钥、用户凭证等，敏感数据的泄漏可能导致传输中数据的被解密和通信被篡改

应对 Heartbleed 漏洞的主要措施包括：

- 更新OpenSSL版本
- 如果敏感信息可能已经泄露，考虑重新生成密钥或者及时终止当前会话

6. 参考资料

RFC 5246 - The Transport Layer Security (TLS) Protocol Version 1.2 (ietf.org)

Welcome to pyca/cryptography — Cryptography 41.0.5 documentation

TLS1.2协议设计原理 - 博客园 (cnblogs.com)

https://heartbleed.com/