Kerberos 算法报告

- 米家龙
- 计算机学院
- 18342075

原理概述

> 客户端需要按顺序分别从如下服务器/进程进行验证：
>
> - AS
> - TGS
> - SS

存在如下密钥：

- K_{Client} 应用于 client 和 AS 之间的**主密钥**
- K_{TGS} 应用于 TGS 和 AS 之间的**对称密钥**
- K_{SS} 应用于 SS 和 TGS 之间的**对称密钥**
- $K_{Client-TGS}$ 应用于 client 和 TGS 之间的**会话密钥**
- $K_{Client-SS}$ 应用于 client 和 SS 之间的**会话密钥**

加密解密函数：

- $E(K, -)$ 使用 K 作为密钥进行加密
- $D(K, -)$ 使用 K 作为密钥进行解密

AS

1. client 向 AS 发送明文消息作为请求（带 clientID ）
2. AS 检查接收到的消息是否符合要求，并且检查 clientID 是否有效：
    - 如果无效：结束会话
    - 如果有效：返回如下两条消息：
        1. 消息 A： $E(K_{Client}, K_{Client-TGS})$
        2. 消息 B： $E(K_{TGS}, <client\ ID, client\ address, validity, K_{Client-TGS}>)$
            - 消息 B 是用 TGS 密钥加密的票据授权票据 TGT，包括客户 ID、客户网络地址、票据有效期、 Client-TGS 会话密钥。（需要注意 client 无法解密消息 B ）
3. Client 收到消息 A 和 B 后，应用 $D(K_{Client}, A)$ 得到 $K_{Client-TGS}$ 用于后续与 TGS 的通信。Client 将凭借消息 B 携带的有效的 TGT 向 TGS 证明其身份。

TGS

1. 申请服务时，Client 向 TGS 发送以下两条消息：
    1. 消息 C： $serviceID, B$
    2. 消息 D： $E(K_{Client-TGS}, <client\ ID, timestamp>)$
        - 消息 D 是用 $K_{Client-TGS}$ 会话密钥加密的认证。
2. TGS 从消息 C 中获得消息 B，应用 D(KTGS , B) 得到 KClient-TGS，再应用 D(KClient-TGS , D) 得到认证内容，并返回给 Client 两条消息：
    1. 消息 E：service ID, ST
        - $ST = E(K_{SS}, <client\ ID, client\ net\ address, validity, K_{Client-SS}>)$ 称为服务票据，包括客户 ID、客户网络地址、票据有效期限、 Client-SS 会话密钥。
    2. 消息 F： $E(K_{Client-TGS}, K_{Client-SS})$ 注意到 Client 无法解密服务票据 ST。

SS

1. Client 应用 $D(K_{Client-TGS}, F)$ 得到 $K_{Client-SS}$，然后向 SS 发出以下两条消息：
    1. 消息 E：由先前步骤得到的 $serviceID, ST$
    2. 消息 G： $E(K_{Client-SS}, <client\ ID, timestamp>)$
        - 消息 G 是用 $K_{Client-SS}$ 会话密钥加密的一个新的认证。
2. SS 应用 $D(K_SS, ST)$ 解密得到 $K_{Client-SS}$，再应用 $D(K_{Client-SS}, G)$ 解密得到认证 G，然后从中提取时间戳 $TS = timestamp$，返回 Client 一条消息 H 作为确认函以确认客户的身份真实，同意向该客户提供服务：

    1. 消息 H：$E(K_{Client-SS}, <client\ ID, TS+1>)$
3. Client 应用 $D(K_{Client-SS}, H)$ 解密消息 H。如果其中的时间戳被正确更新，则 SS 可以信赖，Client 可以向 SS 发送服务请求。
4. 认证过程至此结束，SS 向 Client 客户机提供其所请求的服务

总体结构设计

文件划分如下：

- `src/`
    - `client.c` 客户端源代码
    - `AS.c` AS 源代码
    - `SS.c` SS 源代码
    - `TGS.c` TGS 源代码
    - `des.c` des 加密代码
- `keyClient` 储存 K_{Client} 的文件
- `keySS` 储存 K_{SS} 的文件
- `keyTGS` 储存 K_{TGS} 的文件
- `makefile`

模块分解

> 主要分为三个模块：
>
> - des 相关代码
> - 客户端相关代码
> - 认证服务器/进程相关代码（除了具体步骤之外，其他部分大同小异）

Des 模块

该模块主要负责加密解密，基本沿用原本的 des 代码，部分函数有修改，并且新增部分函数。

Client 模块

主要分成三个部分：

1. 和 AS 通信
2. 和 TGS 通信
3. 和 SS 通信

认证服务模块

需要以下步骤，各个认证模块大体相同:

1. 开启服务器并且监听，循环阻塞等待连接和通信：
    - 等待到连接后：
        1. 继续循环等待有效传输：
        2. 对有效传输进行处理（该部分细节模块直接不同）

数据结构设计

Des 相关

主要应用函数如下：

```
/**
 * 一个完整的解密流程
 * @param encodedStr char* 需要解密的字符串
 * @param encodedStrLen int 需要解密的字符串长度
 * @param decodedStr char* 解密后的字符串
 * @param key char* 解密用的密钥
 * @return 返回解密后的字符串长度
*/
int decodeFull(char *encodedStr, int encodedStrLen, char *decodedStr, char key[17]);

/**
 * 一个完整的加密流程
 * @param srcStr char* 需要加密的字符串
 * @param srcStrLen int 需要加密的字符串长度
 * @param encodedStr char* 加密后的字符串
 * @param key char* 加密用的密钥
 * @return 返回加密后的字符串长度
*/
int encodeFull(char *srcStr, int srcStrLen, char *encodedStr, char key[17]);

/**
 * 将字符串转换为以十六进制显示的字符串
 * char a = \x1\x2\x3\x4\x5\x6\x7\x8
 * 转换为
 * char a1 = \x1\x2\x3\x4
 * char a2 = \x5\x6\x7\x8
 * @param src char* 需要转换的字符串
 * @param srcLen int src 字符串的长度
 * @param dest char* 转换后的字符串
 * @return dest 字符串的长度
*/
int char2intChar(char *src, int srcLen, char *dest);

/**
 * 将以十六进制显示的字符串转换为字符串
 * char a1 = \x1\x2\x3\x4
 * char a2 = \x5\x6\x7\x8
 * 转换为
 * char a = \x1\x2\x3\x4\x5\x6\x7\x8
 * @param src char* 需要转换的字符串
 * @param srcLen int src 字符串的长度
 * @param dest char* 转换后的字符串
 * @return dest 字符串的长度
*/
int intChar2char(char *src, int srcLen, char *dest);
```

> 需要注意的是：由于加密后数据是使用 0~255 的数字来储存进字符中的字符串，并且由于有 0 的存在，因此无法直接通过 `printf()` 函数来进行显示，需要选择将一个字符转换成两个字符的16进制来显示

客户端模块

声明了如下宏和函数：

```c
#define AS_HOST "127.0.0.1"
#define SS_HOST "127.0.0.1"
#define TGS_HOST "127.0.0.1"
#define AS_PORT 23333
#define SS_PORT 23334
#define TGS_PORT 23335

/**
 * 新建 socket
 * @param addr char* 目标 ip
 * @param port int 目标 ip 的端口
 * @return sockaddr_in 结构体
*/
struct sockaddr_in *newSockaddr_in(char *addr, int port);

/**
 * 判断响应是否有效
 * @param response char* 响应字符串
 * @return 0 为有效，1 为无效
*/
int isInvalid(char *response);
```

客户端主函数结构如下：

```c
int main(int argc, char **argv)
{
  if (argc != 2)
  {
    printf("usage: ./client keyClientFile\n");
    return 0;
  }
  else
  {
    // 获取主密钥

    // 定义相关变量

    // AS
    {
      printf("-------AS START-------\n");
      // 新建套接字

      // 尝试连接

      // 发送第一次明文请求

      // 收到响应
      // 判断响应是否有效

      // 消息 A
      // 消息 A 的解密

```

```
        // 收到消息 B
        // 判断响应是否有效

        // 储存消息 B
        printf("--------AS END--------\n\n");
      }

      // TGS
      {
        printf("-------TGS START-------\n");
        // 新建套接字

        // 尝试连接

        // 发送消息 C

        // 收到消息 C
        // 判断响应是否有效

        // 储存 E

        // 生成消息 D
        // 加密消息 D
        // 发送 D

        // 收到响应（消息 F）
        // 判断响应是否无效
        // 解码消息 F
        printf("--------TGS END--------\n\n");
      }

      // SS
      {
        printf("-------TGS START-------\n");
        // 新建套接字

        // 尝试连接

        // 发送消息 E

        // 避免粘包，进行一次无效接受

        // 生成消息 G
        // 加密消息 G
        // 发送消息 G

        // 收到响应（消息 H）
        // 判断响应是否有效
        // 解码消息 H

        // 判断是否认证成功
        if (timestamp == now + 1l && strcmp(clientID_in_H, clientUser) == 0)
        {
          printf("[FIN]: Authentication success\n");
        }
        else
        {
          printf("[ERROR]: Authentication failed\n");
```

```
        }
        printf("--------SS END--------\n\n");
      }

      printf("[END]: clear AS socket\n");

      printf("clear all\n");
      return 0;
    }
  }
```

认证模块

认证模块的基础结构一样，具体如下：

```
  /**
   * 新建 socket
   * @param addr char* 目标 ip
   * @param port int 目标 ip 的端口
   * @return sockaddr_in 结构体
  */
  struct sockaddr_in *newSockaddr_in(char *addr, int port);

  /**
   * 对接收到到的请求进行处理
   * @param clientAddr struct sockaddr_in* 客户端地址
   * @param clientSock int 客户端套接字
  */
  void serve(struct sockaddr_in *clientAddr, int clientSock);

  int main()
  {
    // 申请套接字
    if ((AS_socket = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
      // 报错并退出
    }
    else
    {
      // 生对应地址结构体

      // 进行地址和端口绑定

      // 开始监听

      // 阻塞式接受
      while (1)
      {
        struct sockaddr_in clientAddr; // 客户端
        socklen_t clientAddrSize = sizeof(clientAddr);
        int clientSock;

        // 如果接收到了
        if ((clientSock = accept(AS_socket, (struct sockaddr *)&clientAddr,
  &clientAddrSize)) >= 0)
        {
          // 新建子进程用于处理请求
```

```
          int pid = fork();
          if (pid == 0)
          {
            printf("-------START-------\n");
            serve(&clientAddr, clientSock);
            printf("--------END--------\n\n");
            return 0;
          }
        }
      }
      close(AS_socket);
    }
    return 0;
}

```

而对于不同的认证服务，需要有不用的 `serve` 函数进行处理，如下：

AS

拓展

该模块需要在基础结构上新增如下宏和变量和函数：

```
#define AS_HOST "127.0.0.1"
#define AS_PORT 23333

#define INVALIDID "Invalid ID"
#define INVALIDREQUEST "Invalid request"

const char username[] = "Bob";
const char clientIDList[][200] = {
    "Bob",
    "Alice"}; // 客户端 ID 的列表
const char clientIDListLen = 2;

/**
 * 检查是否含有对应的客户端 ID
 * @param clientID char* 客户端 ID 的字符串
 * @return 0 存在 1 不存在
*/
int checkClientIDList(char *clientID);

/**
 * 随机生成 key_client_TGS
*/
void getKeyCTGS(char res[17]);
```

`serve`

处理请求的 `serve` 函数代码分解如下：

```
void serve(struct sockaddr_in *clientAddr, int clientSock)
{
  char request[1024]; // 请求

  // 显示请求方的相关信息
  printf("from %s: %d\n", inet_ntoa(clientAddr->sin_addr), clientAddr->sin_port);
  int reqLen = read(clientSock, request, 1024);
```

```
8
9     // 循环等待请求
10
11    // 判断请求是否有效
12    // 无效 ID
13    if (reqArgsNum <= 0)
14    {
15      printf("[SEND]: %s\n", INVALIDREQUEST);
16      write(clientSock, INVALIDREQUEST, strlen(INVALIDREQUEST));
17      return;
18    }
19    // ID 不存在
20    else if (checkClientIDList(clientID))
21    {
22      printf("[SEND]: %s\n", INVALIDID);
23      write(clientSock, INVALIDID, strlen(INVALIDID));
24      return;
25    }
26
27    // 如果 ID 有效，需要通过 keyClient 加密暂时生成的会话密钥 key_client_TGS
28
29    // 获取 keyClient
30
31    // 生成 key_client_TGS
32
33    // 生成消息 A
34    // 加密消息 A
35    // 发送消息 A
36
37    // 获取 keyTGS
38
39    // 生成消息 B
40    // 加密消息 B
41    // 发送消息 B
42
43    close(clientSock);
44    return;
45  }
```

TGS

拓展

该模块需要在基础结构上新增如下宏和变量和函数：

```
1   #define TGS_HOST "127.0.0.1"
2   #define TGS_PORT 23335
3
4   #define INVALIDID "Invalid (client or service)ID"
5   #define INVALIDREQUEST "Invalid request"
6   #define INVALIDADDRESS "Invalid address"
7   #define INVALIDTIME "Invalid time"
8
9   const char username[] = "Bob";
10  const char clientIDList[][200] = {
11      "Bob",
12      "Alice"}; // 客户端 ID 的列表
13
```

```
const char serviceIDList[][200] = {
    "testService",
    "service"};
const char clientIDListLen = 2;
const char serviceIDListLen = 2;

/**
 * 检查是否含有对应的客户端 ID
 * @param clientID char* 客户端 ID 的字符串
 * @return 0 存在 1 不存在
*/
int checkClientIDList(char *clientID);

/**
 * 生成 key_client_SS
*/
void getKeyCSS(char res[17]);

/**
 * 检查服务 ID 是否存在
 * @param serviceID char* 服务 ID 字符串
 * @return 0 存在 1 不存在
*/
int checkServiceIDList(char *serviceID);
```

serve

处理请求的 `serve` 函数代码分解如下：

```
void serve(struct sockaddr_in *clientAddr, int clientSock)
{
  char request[1024]; // 请求

  // 获取 key_TGS
  FILE *keyTGSFile = fopen("./keyTGS", "r");
  char keyTGS[17];
  fread(keyTGS, 1, 17, keyTGSFile);

  // 显示请求方的相关信息
  printf("from %s: %d\n", inet_ntoa(clientAddr->sin_addr), clientAddr->sin_port);

  // 循环等待请求
  // 判断请求是否有效

  // 解码消息 B
  // 判断消息 B 是否有效

  // 获取密钥 keySS

  // 生成 key_client_SS

  // 生成 ST
  // 加密 ST
  // 生成消息 E
  // 发送消息 E

  // 接收请求（消息 D）
  // 判断请求有效性
```

```
30
31      // 解密 D
32      // 判断消息 D 的有效性
33
34      // 生成消息 F
35      // 加密消息 F
36      // 发送消息 F
37
38      close(clientSock);
39      fclose(keySSFile);
40      fclose(keyTGSFile);
41      return;
42  }
```

SS

拓展

该模块需要在基础结构上新增如下宏和变量和函数：

```
1   #define SS_HOST "127.0.0.1"
2   #define SS_PORT 23334
3
4   #define INVALIDID "Invalid (client or service)ID"
5   #define INVALIDREQUEST "Invalid request"
6   #define INVALIDADDRESS "Invalid address"
7   #define INVALIDTIME "Invalid time"
8
9   const char username[] = "Bob";
10  const char clientIDList[][200] = {
11      "Bob",
12      "Alice"}; // 客户端 ID 的列表
13
14  const char serviceIDList[][200] = {
15      "testService",
16      "service"};
17  const char clientIDListLen = 2;
18  const char serviceIDListLen = 2;
19
20  /**
21   * 检查是否含有对应的客户端 ID
22   * @param clientID char* 客户端 ID 的字符串
23   * @return 0 存在 1 不存在
24  */
25  int checkClientIDList(char *clientID);
26
27  /**
28   * 生成 key_client_SS
29  */
30  void getKeyCSS(char res[17]);
31
32  /**
33   * 检查是否含有对应的服务 ID
34   * @param serviceID char* 客户端 ID 的字符串
35   * @return 0 存在 1 不存在
36  */
37  int checkServiceIDList(char *serviceID);
```

`serve`

处理请求的 `serve` 函数代码分解如下：

```
void serve(struct sockaddr_in *clientAddr, int clientSock)
{
  char request[1024]; // 请求

  // 显示请求方的相关信息

  // 循环等待请求
  // 判断请求有效性

  // 得到消息 E
  // 获取 key_SS
  // 解密 ST
  // 判断 ST 有效性

  // 避免粘包，进行一次无效发送
  write(clientSock, "RECV G", 1024);

  // 接收请求（消息 G）
  // 判断请求有效性

  // 得到消息 G
  // 解密消息 G
  // 验证消息 G 的有效性

  // 生成消息 H
  // 加密消息 H
  // 发送消息 H

  free(keySSFile);
  close(clientSock);
  return;
}
```

编译运行结果

运行环境

运行环境为 WSL

```
root@LAPTOP-QTCGESHO:/mnt/c/Users/m7811# uname -a
Linux LAPTOP-QTCGESHO 4.4.0-19041-Microsoft #488-Microsoft Mon Sep 01 13:43:00 PST 2020 x86_64 x86_64 x86_64 GNU/Linux

root@LAPTOP-QTCGESHO:/mnt/c/Users/m7811# lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 18.04.5 LTS
Release:        18.04
Codename:       bionic
```

使用 makefile 进行编译运行，makefile 代码如下：

```
# 编译器
GCC := gcc
# 源代码文件
SRC := ./src
# client 代码
CLIENT := client.c
# TGS 代码
TGS := TGS.c
# SS 代码
SS := SS.c
# AS 代码
AS := AS.c
# des 加密
DES := des.c
# 主密钥文件
KEY_CLIENT := ./keyClient

all: client AS SS TGS

client: ${SRC}/${CLIENT} ${SRC}/${DES}
    ${GCC} $< -o $@

TGS: ${SRC}/${TGS} ${SRC}/${DES}
    ${GCC} $< -o $@

AS: ${SRC}/${AS} ${SRC}/${DES}
    ${GCC} $< -o $@

SS: ${SRC}/${SS} ${SRC}/${DES}
    ${GCC} $< -o $@

.PHONY: clean runClient runAS runSS runTGS
clean:
    -@rm client TGS AS SS desTest

runClient: client
    ./$< ${KEY_CLIENT}

runAS: AS
    ./$<

runSS: SS
    ./$<

runTGS: TGS
    ./$<
```

运行结果

> 由于认证过程中能够导致失败的干扰因素比较多，因此只进行一次认证成功的结果展示
>
> 运行该 kerberos 需要开四个终端页面

1. 先运行 AS, TGS, SS 等待请求
2. 使用 `make runClient` 进行客户端认证，得到结果如下图：

client：

```
root@LAPTOP-QTCGESHO:/mnt/d/blog/work/信息安全/004# make runClient
-------AS START-------
[SEND]: authentication Bob
[RECV A]: 91d5af949f490105ab52000f856ccf23c9973733d1de9e52
[MSG A]: b88d29220ec9a4ef
[RECV B]: 7b59266b3cec1ab109ef02eff380fd02a7b99aab861c24f59b72884f9de4286ea9a84d6b9e94613c67face0651c19ebe
--------AS END--------

-------TGS START-------
[SEND C]: testService,7b59266b3cec1ab109ef02eff380fd02a7b99aab861c24f59b72884f9de4286ea9a84d6b9e94613c67face0651c19ebe
[RECV E]: testService,5f778a0ef92ba928e4d92180dde8c5ee9b3f78ee192d6a88fc908a12a47857b38dc787042d5c761aaed8aeddc1ef98eb
[MSG D ]: <Bob,1607862772>
[SEND D ]: 4933c16a65d67de59df0397872bd379362159af38956fb7c
[RECV F]: f55e18916fcd2be3d9a10b830ae55d0162159af38956fb7c
[MSG F]: 04cca0147dab4866
--------TGS END--------

-------TGS START-------
[SEND E]: testService,5f778a0ef92ba928e4d92180dde8c5ee9b3f78ee192d6a88fc908a12a47857b38dc787042d5c761aaed8aeddc1ef98eb
[MSG G]: <Bob,1607862772>
[SEND G]: 33739c90a94a263d196f578d24f6f8d01d362888fcff2159
[RECV H]: 33739c90a94a263df6af7900d40cb85a1d362888fcff2159
[MSG H]: <Bob,1607862773>
[FIN]: Authentication success
--------SS END--------

[END]: clear AS socket
clear all
root@LAPTOP-QTCGESHO:/mnt/d/blog/work/信息安全/004#
```

AS：

```
root@LAPTOP-QTCGESHO:/mnt/d/blog/work/信息安全/004# make runAS
AS bind ip: 127.0.0.1:23333
AS start to listen
-------START-------
from 127.0.0.1: 48928
[RECV]: authentication Bob
[SEND A]: 91d5af949f490105ab52000f856ccf23c9973733d1de9e52
[MSG B]: <Bob,127.0.0.1,1607863672,b88d29220ec9a4ef>
[SEND B ]: 7b59266b3cec1ab109ef02eff380fd02a7b99aab861c24f59b72884f9de4286ea9a84d6b9e946
13c67face0651c19ebe
--------END--------
```

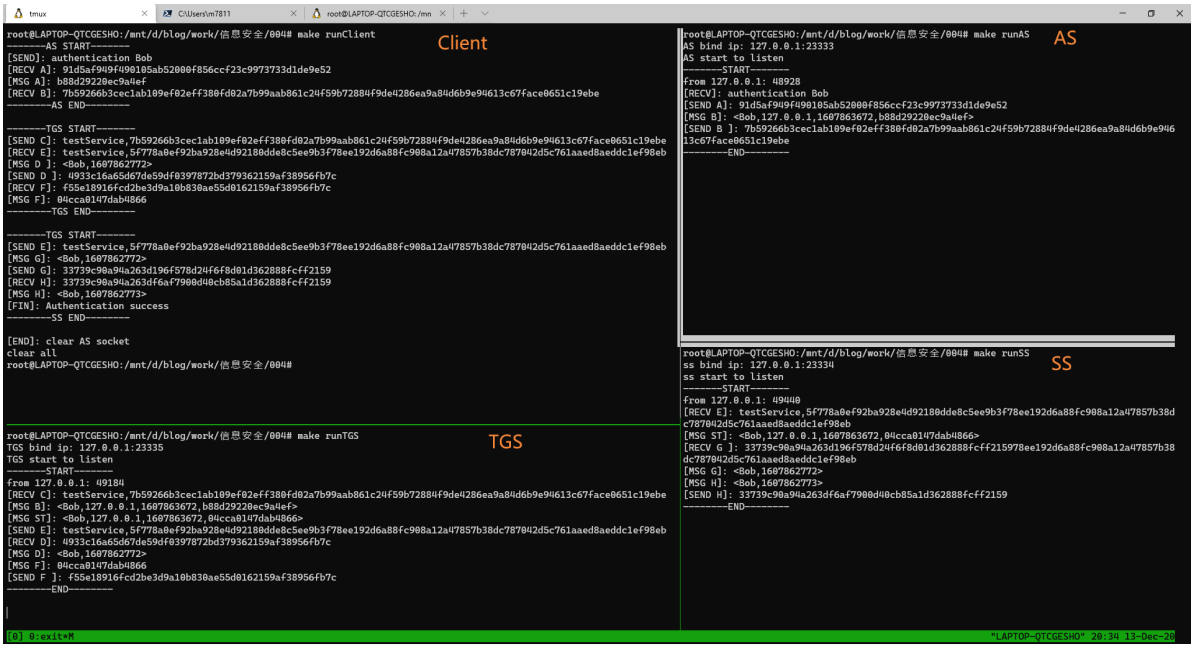
TGS：

```
root@LAPTOP-QTCGESHO:/mnt/d/blog/work/信息安全/004# make runTGS
TGS bind ip: 127.0.0.1:23335
TGS start to listen
-------START-------
from 127.0.0.1: 49184
[RECV C]: testService,7b59266b3cec1ab109ef02eff380fd02a7b99aab861c24f59b72884f9de4286ea9a84d6b9e94613c67face0651c19ebe
[MSG B]: <Bob,127.0.0.1,1607863672,b88d29220ec9a4ef>
[MSG ST]: <Bob,127.0.0.1,1607863672,04cca0147dab4866>
[SEND E]: testService,5f778a0ef92ba928e4d92180dde8c5ee9b3f78ee192d6a88fc908a12a47857b38dc787042d5c761aaed8aeddc1ef98eb
[RECV D]: 4933c16a65d67de59df0397872bd379362159af38956fb7c
[MSG D]: <Bob,1607862772>
[MSG F]: 04cca0147dab4866
[SEND F ]: f55e18916fcd2be3d9a10b830ae55d0162159af38956fb7c
--------END--------
```

SS：

```
root@LAPTOP-QTCGESHO:/mnt/d/blog/work/信息安全/004# make runSS
ss bind ip: 127.0.0.1:23334
ss start to listen
-------START-------
from 127.0.0.1: 49440
[RECV E]: testService,5f778a0ef92ba928e4d92180dde8c5ee9b3f78ee192d6a88fc908a12a47857b38d
c787042d5c761aaed8aeddc1ef98eb
[MSG ST]: <Bob,127.0.0.1,1607863672,04cca0147dab4866>
[RECV G ]: 33739c90a94a263d196f578d24f6f8d01d362888fcff215978ee192d6a88fc908a12a47857b38
dc787042d5c761aaed8aeddc1ef98eb
[MSG G]: <Bob,1607862772>
[MSG H]: <Bob,1607862773>
[SEND H]: 33739c90a94a263df6af7900d40cb85a1d362888fcff2159
--------END--------
```

完整页面截图如下：



C 语言源代码

> 代码量较大，建议直接查看源代码文件

des.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <ctype.h>
#include <string.h>

#define BLOCK64 64          // 01位块长度
#define BLOCK8 9            // 8字节明文块长度，由于字符串限制，必须+1
#define EEXTAND 48          // E-拓展串
#define SUBKEYLEN 48        // 子密钥长度
#define SUBKEYNUM 16        // 子密钥数量
#define KEYLEN 64           // 密钥长度
#define NOCHECKDIGITLEN 56 // 非校验位长度

typedef bool des1_t;
```

```
typedef unsigned char des8_t;

// IP 置换表
const int IP_TABLE[BLOCK64] = {
    58, 50, 42, 34, 26, 18, 10, 2,
    60, 52, 44, 36, 28, 20, 12, 4,
    62, 54, 46, 38, 30, 22, 14, 6,
    64, 56, 48, 40, 32, 24, 16, 8,
    57, 49, 41, 33, 25, 17, 9, 1,
    59, 51, 43, 35, 27, 19, 11, 3,
    61, 53, 45, 37, 29, 21, 13, 5,
    63, 55, 47, 39, 31, 23, 15, 7};

// IP逆 置换表
const int IP_TABLE_REVERSE[BLOCK64] = {
    40, 8, 48, 16, 56, 24, 64, 32,
    39, 7, 47, 15, 55, 23, 63, 31,
    38, 6, 46, 14, 54, 22, 62, 30,
    37, 5, 45, 13, 53, 21, 61, 29,
    36, 4, 44, 12, 52, 20, 60, 28,
    35, 3, 43, 11, 51, 19, 59, 27,
    34, 2, 42, 10, 50, 18, 58, 26,
    33, 1, 41, 9, 49, 17, 57, 25};

// P-置换
const int P_TABLE[BLOCK64 / 2] = {
    16, 7, 20, 21,
    29, 12, 28, 17,
    1, 15, 23, 26,
    5, 18, 31, 10,
    2, 8, 24, 14,
    32, 27, 3, 9,
    19, 13, 30, 6,
    22, 11, 4, 25};

// PC-1 置换表
const int PC_1_TABLE[NOCHECKDIGITLEN] = {
    // C0
    57, 49, 41, 33, 25, 17, 9,
    11, 58, 50, 42, 34, 26, 18,
    10, 2, 59, 51, 43, 35, 27,
    19, 11, 3, 60, 52, 44, 36,

    // D0
    63, 55, 47, 39, 31, 23, 15,
    7, 62, 54, 46, 38, 30, 22,
    14, 6, 61, 53, 45, 37, 29,
    21, 13, 5, 28, 20, 12, 4};

// PC-2 置换表
const int PC_2_TABLE[SUBKEYLEN] = {
    14, 17, 11, 24, 1, 5,
    3, 28, 15, 6, 21, 10,
    23, 19, 12, 4, 26, 8,
    16, 7, 27, 20, 13, 2,

    41, 52, 31, 37, 47, 55,
    30, 40, 51, 45, 33, 48,
```

```
    44, 49, 39, 56, 34, 53,
    46, 42, 50, 36, 29, 32};

// S 盒
const int S_BOX[][BLOCK64] = {
    {14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7,
     0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8,
     4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0,
     15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13},

    {15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10,
     3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5,
     0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15,
     13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9},

    {10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8,
     13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1,
     13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7,
     1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12},

    {7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15,
     13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9,
     10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4,
     3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14},

    {2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9,
     14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6,
     4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14,
     11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3},

    {12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11,
     10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8,
     9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6,
     4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13},

    {4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1,
     13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6,
     1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2,
     6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12},

    {13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7,
     1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2,
     7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8,
     2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11}};

// E-拓规则（比特-选择表）
const int E_EXTAND[SUBKEYLEN] = {
    32, 1, 2, 3, 4, 5,
    4, 5, 6, 7, 8, 9,
    8, 9, 10, 11, 12, 13,
    12, 13, 14, 15, 16, 17,
    16, 17, 18, 19, 20, 21,
    20, 21, 22, 23, 24, 25,
    24, 25, 26, 27, 28, 29,
    28, 29, 30, 31, 32, 1};

des8_t block8[BLOCK8];           // 明文
des8_t encodedBlock8[BLOCK8];    // 加密后的明文
```

```
132   des1_t block64[BLOCK64];          // 二进制明文
133   des1_t encodedBlock64[BLOCK64];   // 加密后的二进制明文
134   des1_t encodingBlock64[BLOCK64]; // 加密中的二进制明文
135   des1_t decodedBlock64[BLOCK64];   // 解密后的二进制明文
136   des8_t decodedBlock8[BLOCK8];     // 解密后的明文
137   des1_t decodingBlock64[BLOCK64]; // 解密中的二进制明文
138
139   char InitKey[KEYLEN / 4 + 1];         // 16进制的输入
140   des1_t Key[BLOCK64];                  // 密钥
141   des1_t Subkey[SUBKEYNUM][SUBKEYLEN]; // 子密钥
142
143   FILE *readFile; // 读取的文件
144
145   /**
146    * 通过密钥生成子密钥，总共生成16个
147    * @param K des1_t* 64位密钥
148   */
149   void getSubkey(des1_t *K);
150
151   /**
152    * 8字节 转换成 64位
153    * @param from des8_t* 源数组
154    * @param to des1_t* 目标数组
155   */
156   void block8ToBlock64(des8_t *from, des1_t *to);
157
158   /**
159    * 64位 转换为 8字节
160    * @param from des1_t* 源数组
161    * @param to des8_t* 目标数组
162   */
163   void block64ToBlock8(des1_t *from, des8_t *to);
164
165   /**
166    * 通过初始获取的密钥进行转换
167   */
168   void getKey();
169
170   /**
171    * 轮函数
172    * @param Ri des1_t*
173    * @param iterationNum int 迭代次数
174    * @return 一个32位数组指针
175   */
176   des1_t *Feistel(des1_t *Ri, int iteraionNum);
177
178   /**
179    * 块加密
180   */
181   void encodeBlock();
182
183   /**
184    * 块解密
185   */
186   void decodeBlock();
187
188   /**
189    * 加密
```

```
*/
int encode(char *srcStr, int srcStrLen, char *encodedStr);

/**
 * 解密
*/
int decode(char *encodedStr, int encodeStrLen, char *decodedStr);

/**
 * 一个完整的解密流程
 * @param encodedStr char* 需要解密的字符串
 * @param encodedStrLen int 需要解密的字符串长度
 * @param decodedStr char* 解密后的字符串
 * @param key char* 解密用的密钥
 * @return 返回解密后的字符串长度
*/
int decodeFull(char *encodedStr, int encodedStrLen, char *decodedStr, char
key[17]);

/**
 * 一个完整的加密流程
 * @param srcStr char* 需要加密的字符串
 * @param srcStrLen int 需要加密的字符串长度
 * @param encodedStr char* 加密后的字符串
 * @param key char* 加密用的密钥
 * @return 返回加密后的字符串长度
*/
int encodeFull(char *srcStr, int srcStrLen, char *encodedStr, char key[17]);

/**
 * 将字符串转换为以十六进制显示的字符串
 * char a = \x1\x2\x3\x4\x5\x6\x7\x8
 * 转换为
 * char a1 = \x1\x2\x3\x4
 * char a2 = \x5\x6\x7\x8
 * @param src char* 需要转换的字符串
 * @param srcLen int src 字符串的长度
 * @param dest char* 转换后的字符串
 * @return dest 字符串的长度
*/
int char2intChar(char *src, int srcLen, char *dest);

/**
 * 将以十六进制显示的字符串转换为字符串
 * char a1 = \x1\x2\x3\x4
 * char a2 = \x5\x6\x7\x8
 * 转换为
 * char a = \x1\x2\x3\x4\x5\x6\x7\x8
 * @param src char* 需要转换的字符串
 * @param srcLen int src 字符串的长度
 * @param dest char* 转换后的字符串
 * @return dest 字符串的长度
*/
int intChar2char(char *src, int srcLen, char *dest);

/**
 * 重置所有数组
*/
```

```
void clearDes()
{
  for (int i = 0; i < 8; i++)
  {
    block8[i] = encodedBlock8[i] = decodedBlock8[i] = 0;
  }

  for (int i = 0; i < 64; i++)
  {
    block64[i] = encodedBlock64[i] = decodedBlock64[i] =
        Key[i] = encodingBlock64[i] = decodingBlock64[i] = 0;
  }

  for (int i = 0; i < 16; i++)
  {
    InitKey[i] = 0;
    for (int j = 0; j < SUBKEYLEN; j++)
    {
      Subkey[i][j] = 0;
    }
  }

  InitKey[16] = 0;
  return;
}

int encodeFull(char *srcStr, int srcStrLen, char *encodedStr, char key[17])
{
  strcpy(InitKey, key);
  getKey();
  getSubkey(Key);
  int res = encode(srcStr, srcStrLen, encodedStr);
  clearDes();
  return res;
}

int decodeFull(char *encodedStr, int encodeStrLen, char *decodedStr, char
key[17])
{
  strcpy(InitKey, key);
  getKey();
  getSubkey(Key);
  int res = decode(encodedStr, encodeStrLen, decodedStr);

  // printf("decodeFull: %s\n", decodedStr);
  clearDes();
  // return &decodedStr;
  return res;
}

int decode(char *encodedStr, int encodeStrLen, char *decodedStr)
{
  int blockNum = encodeStrLen / 8 + (encodeStrLen % 8 == 0 ? 0 : 1);

  for (int t = 0; t < blockNum; t++)
  {
    int len = 0;
    for (int j = 0; j < 8 && (j + t * 8) < encodeStrLen; j++)
```

```
304       {
305         encodedBlock8[j] = encodedStr[j + t * 8];
306         len = j;
307       }
308       encodedBlock8[len + 1] = 0;
309       block8ToBlock64(encodedBlock8, encodedBlock64);
310       decodeBlock();
311       block64ToBlock8(decodedBlock64, decodedBlock8);
312
313       // 去除填充
314       decodedBlock8[8] = 0;
315       int tail = decodedBlock8[7]; // 看末尾那位是否是填充的
316       bool isPadding = true;
317       for (int i = 8 - tail; i < BLOCK8 - 1; i++)
318       {
319         if (decodedBlock8[i] != tail) // 不是填充
320         {
321           isPadding = false;
322           break;
323         }
324       }
325
326       if (isPadding)
327       {
328         decodedBlock8[8 - tail] = 0;
329       }
330
331       for (int i = 0; i < 8; i++)
332       {
333         decodedStr[t * 8 + i] = decodedBlock8[i];
334       }
335
336       // printf("%s\n%s\n", decodedBlock8, decodedStr);
337     }
338
339     // printf("decode end: %s\n", decodedStr);
340
341     return strlen(decodedStr);
342   }
343
344   int encode(char *srcStr, int srcStrLen, char *encodedStr)
345   {
346
347     int blockNum = srcStrLen / 8 + 1;
348
349     bool padding = false; // 判定是否已经补全
350     int len = 0;
351     int count = 0;
352
353     // printf("start to loop\n");
354     for (int t = 0; t < blockNum; t++)
355     {
356       int len = 0;
357       for (int j = 0; j < 8 && (j + t * 8) < srcStrLen; j++)
358       {
359         block8[j] = srcStr[j + t * 8];
360         len = j + 1;
361       }
```

```
      block8[len] = 0;
      // len = fread(block8, 1, 8, readFile);
      // block8[len ] = 0;
      // printf("%s\n", block8);
      if (len < 8)
      {
        for (int i = len; i < 8; i++)
        {
          block8[i] = 8 - len; // 填充
        }
        block8[8] = 0;
        padding = true;
      }
      block8ToBlock64(block8, block64);
      encodeBlock();
      block64ToBlock8(encodedBlock64, encodedBlock8);

      for (int i = 0; i < 8; i++)
      {
        encodedStr[count++] = encodedBlock8[i];
      }
    }

    // 如果刚好输入完成，那么需要补一个块
    if (!padding)
    {
      for (int i = 0; i < 8; i++)
      {
        block8[i] = 0x08;
      }
      block8[8] = 0;
      block8ToBlock64(block8, block64);
      encodeBlock();
      block64ToBlock8(encodedBlock64, encodedBlock8);
      for (int i = 0; i < 8; i++)
      {
        encodedStr[count++] = encodedBlock8[i];
      }
    }
    return blockNum * 8;
  }

  void block8ToBlock64(des8_t *from, des1_t *to)
  {
    for (int i = 0; i < 8; i++)
    {
      des8_t tmp = from[i];
      for (int j = 0; j < 8; j++)
      {
        to[i * 8 + j] = (tmp >> (7 - j)) & 1;
      }
    }
  }

  void block64ToBlock8(des1_t *from, des8_t *to)
  {
    for (int i = 0; i < 8; i++)
    {
```

```
      des8_t tmp = 0;
      for (int j = 0; j < 8; j++)
      {
        tmp = (tmp << 1) + from[i * 8 + j];
      }
      to[i] = tmp;
    }
  }

  void encodeBlock()
  {
    // 初始置换 IP
    for (int i = 0; i < BLOCK64; i++)
    {
      encodingBlock64[i] = block64[IP_TABLE[i] - 1];
    }
    // 16次迭代
    des1_t *Li = encodingBlock64;               // 初始化 L0
    des1_t *Ri = encodingBlock64 + BLOCK64 / 2; // 初始化 R0

    for (int i = 0; i < BLOCK64 / 4; i++)
    {
      des1_t *tmp = Feistel(Ri, i); // 轮函数结果
      des1_t L_tmp, R_tmp;
      for (int j = 0; j < BLOCK64 / 2; j++)
      {
        L_tmp = Ri[j];
        R_tmp = Li[j] ^ tmp[j];

        Li[j] = L_tmp;
        Ri[j] = R_tmp;
      }
    }

    // 交换置换
    for (int i = 0; i < BLOCK64 / 2; i++)
    {
      des1_t tmp = Li[i];
      Li[i] = Ri[i];
      Ri[i] = tmp;
    }
    for (int i = 0; i < BLOCK64; i++)
    {
      encodedBlock64[i] = encodingBlock64[IP_TABLE_REVERSE[i] - 1];
    }
  }

  des1_t *Feistel(des1_t *Ri, int iteraionNum)
  {
    // E 拓展
    des1_t e_extand[48]; // E 拓展结果
    for (int i = 0; i < EEXTAND; i++)
    {
      e_extand[i] = Ri[E_EXTAND[i] - 1];
    }

    des1_t xorList[48]; // 异或的结果
    for (int i = 0; i < EEXTAND; i++)
```

```
478     {
479       xorList[i] = e_extand[i] ^ Subkey[iteraionNum][i];
480     }
481
482     // S 盒压缩
483     des1_t s_box_res[32]; // S 盒压缩结果
484     for (int i = 0; i < 8; i++)
485     {
486       int n = (xorList[i * 6] << 1) + xorList[i * 6 + 5];
                                          // 确定行号
487       int m = (xorList[i * 6 + 1] << 3) + (xorList[i * 6 + 2] << 2) + (xorList[i *
    6 + 3] << 1) + xorList[i * 6 + 4]; // 获取列号
488
489       des8_t res = S_BOX[i][n * BLOCK64 / 4 + m];
490
491       for (int j = 0; j < 4; j++)
492       {
493         s_box_res[i * 4 + j] = (res >> (3 - j)) & 1;
494       }
495     }
496
497     static des1_t p_res[BLOCK64 / 2]; // P 置换的结果
498     for (int i = 0; i < BLOCK64 / 2; i++)
499     {
500       p_res[i] = s_box_res[P_TABLE[i] - 1];
501     }
502
503     return p_res;
504   }
505
506   void getKey()
507   {
508     for (int i = 0; i < 16; i++)
509     {
510       int moveBit = i % 2 == 0 ? 4 : 0;
511       int tmp = InitKey[i] = tolower(InitKey[i]);
512       if (isdigit(tmp))
513       {
514         tmp -= '0';
515         InitKey[i] = tmp;
516       }
517       else
518       {
519         tmp -= ('a' - 10);
520         InitKey[i] = tmp;
521       }
522
523       for (int j = 0; j < 4; j++)
524       {
525         Key[i * 4 + j] = (tmp >> (3 - j)) & 1;
526       }
527     }
528   }
529
530   void getSubkey(des1_t *K)
531   {
532
533     // 进行初始的 PC-1 置换
```

```
534     des1_t CD[NOCHECKDIGITLEN];
535     for (int i = 0; i < NOCHECKDIGITLEN; i++)
536     {
537       CD[i] = K[PC_1_TABLE[i] - 1];
538     }
539 
540     // 循环生成
541     for (int i = 0; i < SUBKEYNUM; i++)
542     {
543 
544       // 进行 LS 操作
545       if (i == 0 || i == 1 || i == 8 || i == 15) // 需要循环左移1个位置
546       {
547         des1_t tmpC = CD[0];                    // 对 C
548         des1_t tmpD = CD[NOCHECKDIGITLEN / 2]; // 对 D
549         for (int j = 0; j < NOCHECKDIGITLEN / 2 - 1; j++)
550         {
551           CD[j] = CD[j + 1];
552           CD[j + NOCHECKDIGITLEN / 2] = CD[j + NOCHECKDIGITLEN / 2 + 1];
553         }
554         CD[NOCHECKDIGITLEN / 2 - 1] = tmpC;
555         CD[NOCHECKDIGITLEN - 1] = tmpD;
556       }
557       else // 否则循环左移2个位置
558       {
559         des1_t tmpC1 = CD[0], tmpC2 = CD[1];
560         des1_t tmpD1 = CD[NOCHECKDIGITLEN / 2], tmpD2 = CD[NOCHECKDIGITLEN / 2 +
    1];
561         for (int j = 0; j < NOCHECKDIGITLEN / 2 - 2; j++)
562         {
563           CD[j] = CD[j + 2];
564           CD[j + NOCHECKDIGITLEN / 2] = CD[j + NOCHECKDIGITLEN / 2 + 2];
565         }
566         CD[NOCHECKDIGITLEN / 2 - 2] = tmpC1;
567         CD[NOCHECKDIGITLEN / 2 - 1] = tmpC2;
568         CD[NOCHECKDIGITLEN - 2] = tmpD1;
569         CD[NOCHECKDIGITLEN - 1] = tmpD2;
570       }
571 
572       // PC-2 压缩置换
573       for (int j = 0; j < SUBKEYLEN; j++)
574       {
575         Subkey[i][j] = CD[PC_2_TABLE[j] - 1];
576       }
577     }
578   }
579 
580   void decodeBlock()
581   {
582     // 初始置换 IP
583     for (int i = 0; i < BLOCK64; i++)
584     {
585       decodingBlock64[i] = encodedBlock64[IP_TABLE[i] - 1];
586     }
587     // 16次迭代
588     des1_t *Li = decodingBlock64;               // 初始化 L0
589     des1_t *Ri = decodingBlock64 + BLOCK64 / 2; // 初始化 R0
590 
```

```
    for (int i = BLOCK64 / 4 - 1; i >= 0; i--)
    {
      des1_t *tmp = Feistel(Ri, i); // 轮函数结果
      des1_t L_tmp, R_tmp;
      for (int j = 0; j < BLOCK64 / 2; j++)
      {
        L_tmp = Ri[j];
        R_tmp = Li[j] ^ tmp[j];

        Li[j] = L_tmp;
        Ri[j] = R_tmp;
      }
    }

    // 交换置换
    for (int i = 0; i < BLOCK64 / 2; i++)
    {
      des1_t tmp = Li[i];
      Li[i] = Ri[i];
      Ri[i] = tmp;
    }

    // 逆置换
    for (int i = 0; i < BLOCK64; i++)
    {
      decodedBlock64[i] = decodingBlock64[IP_TABLE_REVERSE[i] - 1];
    }
  }

  void printF(char *message, int len)
  {
    for (int i = 0; i < len; i++)
    {
      printf("%02x", message[i] & 0xff);
    }
    printf("\n");
  }

  #define getHex(tmp) ((tmp <= 9 && tmp >= 0) ? (tmp + '0') : (tmp - 10 + 'a'))
  #define getHalfChar(tmp) ((tmp <= '9' && tmp >= '0') ? (tmp - '0') : (tmp - 'a'
  + 10))

  int char2intChar(char *src, int srcLen, char *dest)
  {
    int tmp;
    for (int i = 0; i < srcLen; i++)
    {
      tmp = ((src[i] & 0xff) >> 4) & 0x0f;
      dest[i * 2] = getHex(tmp);
      tmp = src[i] & 0x0f;
      dest[i * 2 + 1] = getHex(tmp);
      // printf("%d\t%d\t%02x\t%c\t%c\n", ((src[i] & 0xff) >> 4) & 0x0f, src[i] &
  0x0f, src[i] & 0xff, dest[i * 2], dest[i * 2 + 1]);
    }

    dest[srcLen * 2] = 0;

    return srcLen * 2;
```

```
}

int intChar2char(char *src, int srcLen, char *dest)
{
  for (int i = 0; i < srcLen / 2; i++)
  {
    dest[i] = (getHalfChar(src[i * 2]) << 4) + getHalfChar(src[i * 2 + 1]);
  }

  dest[srcLen / 2] = 0;
  return srcLen / 2;
}
```

client.c

```
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

#include "./des.c"

#define AS_HOST "127.0.0.1"
#define SS_HOST "127.0.0.1"
#define TGS_HOST "127.0.0.1"
#define AS_PORT 23333
#define SS_PORT 23334
#define TGS_PORT 23335

/**
 * 新建 socket
 * @param addr char* 目标 ip
 * @param port int 目标 ip 的端口
 * @return sockaddr_in 结构体
*/
struct sockaddr_in *newSockaddr_in(char *addr, int port);

/**
 * 判断响应是否有效
 * @param response char* 响应字符串
 * @return 0 为有效，1 为无效
*/
int isInvalid(char *response);

struct sockaddr_in *newSockaddr_in(char *addr, int port)
{
  struct sockaddr_in *sockaddr = (struct sockaddr_in *)malloc(sizeof(struct
sockaddr_in));
```

```
    sockaddr->sin_addr.s_addr = inet_addr(addr);
    sockaddr->sin_family = AF_INET;
    sockaddr->sin_port = htons(port);
    return sockaddr;
}

int main(int argc, char **argv)
{
    if (argc != 2)
    {
        printf("usage: ./client keyClientFile\n");
        return 0;
    }
    else
    {
        // 获取主密钥
        FILE *keyClientFile = fopen(argv[1], "r");
        char keyClient[17];
        fread(keyClient, 1, 16, keyClientFile);

        int clientSock; // 客户端套接字

        const char clientUser[] = "Bob";         // 用户 ID
        const char serviceID[] = "testService"; // 服务 ID

        time_t now;

        char request[1024];  // 请求
        char response[1024]; // 响应
        int resLen;          // 响应字符串长度
        char error[1024];    // 错误相关

        char keyCTGS[17]; // key_client_TGS
        char keyCSS[17];  // key_client_SS

        // 消息 A 相关
        char messageA[1024];
        char messageA_encoded[1027];
        char messageA_decoded[1024];
        int messageA_encoded_Len = 0;

        // 消息 B 相关
        char messageB[1024];

        // 消息 C 相关
        char messageC[1024];

        // 消息 D 相关
        char messageD[1024];
        char messageD_encoded[1024];
        char messageD_encoded_transfered[1024];
        int messageD_encoded_len;

        // 消息 E 相关
        char messageE[1024];
        char messageE_encoded[1024];
        char messageE_decoded[1024];
        char messageE_encoded_len = 0;
```

```
100
101       // 消息 F 相关
102       char messageF[1024];
103       char messageF_encoded[1024];
104       char messageF_decoded[1024];
105       char messageF_encoded_len = 0;
106
107       // 消息 G 相关
108       char messageG[1024];
109       char messageG_encoded[1024];
110       char messageG_encoded_transfered[1024];
111       char messageG_encoded_len = 0;
112
113       // 消息 H 相关
114       char messageH[1024];
115       char messageH_encoded[1024];
116       char messageH_decoded[1024];
117       char messageH_encoded_transfered[1024];
118       int messageH_encoded_len = 0;
119
120       // AS
121       {
122         printf("-------AS START-------\n");
123
124         // 新建套接字
125         if ((clientSock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
126         {
127           printf("[ERROR]: client socket failed\n");
128           return 1;
129         }
130
131         // 尝试连接
132         struct sockaddr_in *AS_sockaddr = newSockaddr_in(AS_HOST, AS_PORT);
133         if (connect(clientSock, (struct sockaddr *)AS_sockaddr, sizeof(struct
    sockaddr_in)) < 0)
134         {
135           printf("[ERROR]: connect AS failed\n");
136           return 1;
137         }
138
139         // 发送第一次明文请求
140         sprintf(request, "authentication %s", clientUser);
141         write(clientSock, request, strlen(request));
142         printf("[SEND]: %s\n", request);
143
144         // 收到响应
145         resLen = read(clientSock, response, 1024);
146         response[resLen] = 0;
147
148         // 判断响应是否有效
149         if (isInvalid(response))
150         {
151           close(clientSock);
152           free(AS_sockaddr);
153           return 1;
154         }
155
156         // 消息 A
```

```
157         printf("[RECV A]: %s\n", response);
158         strcpy(messageA, response);
159 
160         // 消息 A 的解密
161         messageA_encoded_Len = intChar2char(messageA, resLen, messageA_encoded);
162         decodeFull(messageA_encoded, messageA_encoded_Len, messageA_decoded,
    keyClient);
163         printf("[MSG A]: %s\n", messageA_decoded);
164         strcpy(keyCTGS, messageA_decoded);
165 
166         // 收到消息 B
167         resLen = read(clientSock, response, 1024);
168         response[resLen] = 0;
169 
170         // 判断响应是否有效
171         if (isInvalid(response))
172         {
173           close(clientSock);
174           free(AS_sockaddr);
175           return 1;
176         }
177 
178         // 储存消息 B
179         strcpy(messageB, response);
180         printf("[RECV B]: %s\n", messageB);
181 
182         close(clientSock);
183         free(AS_sockaddr);
184 
185         printf("--------AS END--------\n\n");
186       }
187 
188       // TGS
189       {
190         printf("-------TGS START-------\n");
191 
192         // 新建套接字
193         if ((clientSock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
194         {
195           printf("[ERROR]: client socket failed\n");
196           return 1;
197         }
198 
199         // 尝试连接
200         struct sockaddr_in *TGS_sockaddr = newSockaddr_in(TGS_HOST, TGS_PORT);
201         if (connect(clientSock, (struct sockaddr *)TGS_sockaddr, sizeof(struct
    sockaddr)))
202         {
203           printf("[ERROR]: connect TGS failed\n");
204           return 1;
205         }
206 
207         // 发送消息 C
208         sprintf(messageC, "%s,%s", serviceID, messageB);
209         printf("[SEND C]: %s\n", messageC);
210         write(clientSock, messageC, strlen(messageC));
211 
212         // 接受 E
```

```
        resLen = read(clientSock, response, 1024);
        response[resLen] = 0;

        // 判断响应是否有效
        if (isInvalid(response))
        {
          close(clientSock);
          free(TGS_sockaddr);
          return 1;
        }

        // 储存 E
        strcpy(messageE, response);
        printf("[RECV E]: %s\n", messageE);

        // 生成消息 D
        now = time(NULL);
        sprintf(messageD, "<%s,%ld>", clientUser, now);
        printf("[MSG D ]: %s\n", messageD);

        // 加密消息 D
        messageD_encoded_len = encodeFull(messageD, strlen(messageD), messageD_encoded, messageA_decoded);
        char2intChar(messageD_encoded, messageD_encoded_len, messageD_encoded_transfered);

        // 发送 D
        write(clientSock, messageD_encoded_transfered, strlen(messageD_encoded_transfered));
        printf("[SEND D ]: %s\n", messageD_encoded_transfered);

        // 收到响应（消息 F）
        resLen = read(clientSock, response, 1024);
        response[resLen] = 0;

        // 判断响应是否无效
        if (isInvalid(response))
        {
          close(clientSock);
          free(TGS_sockaddr);
          return 1;
        }

        // 解码消息 F
        strcpy(messageF, response);
        printf("[RECV F]: %s\n", messageF);
        messageF_encoded_len = intChar2char(messageF, strlen(messageF), messageF_encoded);
        decodeFull(messageF_encoded, messageF_encoded_len, messageF_decoded, keyCTGS);
        printf("[MSG F]: %s\n", messageF_decoded);
        strcpy(keyCSS, messageF_decoded);

        printf("--------TGS END--------\n\n");
      }

      // SS
      {
```

```
266         printf("-------TGS START-------\n");
267 
268         // 新建套接字
269         if ((clientSock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
270         {
271           printf("[ERROR]: client socket failed\n");
272           return 1;
273         }
274 
275         // 尝试连接
276         struct sockaddr_in *SS_sockaddr = newSockaddr_in(SS_HOST, SS_PORT);
277         if (connect(clientSock, (struct sockaddr *)SS_sockaddr, sizeof(struct
    sockaddr)))
278         {
279           printf("[ERROR]: connect SS failed\n");
280           return 1;
281         }
282 
283         // 发送消息 E
284         messageE[strlen(messageE)] = 0;
285         write(clientSock, messageE, strlen(messageE) + 1);
286         printf("[SEND E]: %s\n", messageE);
287 
288         // 避免粘包，进行一次无效接受
289         read(clientSock, response, 1024);
290 
291         // 生成消息 G
292         now = time(NULL);
293         sprintf(messageG, "<%s,%ld>", clientUser, now);
294         printf("[MSG G]: %s\n", messageG);
295 
296         // 加密消息 G
297         messageG_encoded_len = encodeFull(messageG, strlen(messageG),
    messageG_encoded, keyCSS);
298         char2intChar(messageG_encoded, messageG_encoded_len,
    messageG_encoded_transfered);
299 
300         // 发送消息 G
301         write(clientSock, messageG_encoded_transfered,
    strlen(messageG_encoded_transfered));
302         printf("[SEND G]: %s\n", messageG_encoded_transfered);
303 
304         // 收到响应（消息 H）
305         resLen = read(clientSock, response, 1024);
306         response[resLen] = 0;
307 
308         // 判断响应是否有效
309         if (resLen <= 0 || isInvalid(response))
310         {
311           close(clientSock);
312           free(SS_sockaddr);
313           return 1;
314         }
315 
316         strcpy(messageH, response);
317         printf("[RECV H]: %s\n", messageH);
318 
319         // 解码消息 H
```

```
320         messageH_encoded_len = intChar2char(messageH, strlen(messageH),
    messageH_encoded);
321         decodeFull(messageH_encoded, messageH_encoded_len, messageH_decoded,
    keyCSS);
322         printf("[MSG H]: %s\n", messageH_decoded);
323 
324         // 判断是否认证成功
325         time_t timestamp;
326         char clientID_in_H[1024];
327         sscanf(messageH_decoded, "<%[^,],%ld>", clientID_in_H, &timestamp);
328 
329         if (timestamp == now + 1l && strcmp(clientID_in_H, clientUser) == 0)
330         {
331           printf("[FIN]: Authentication success\n");
332         }
333         else
334         {
335           printf("[ERROR]: Authentication failed\n");
336         }
337 
338         printf("--------SS END--------\n\n");
339       }
340 
341       printf("[END]: clear AS socket\n");
342 
343       printf("clear all\n");
344       return 0;
345     }
346   }
347 
348   int isInvalid(char *response)
349   {
350     char error[1024];
351     if (sscanf(response, "Invalid %s", error) != 0)
352     {
353       printf("[ERROR]: %s\n", response);
354       return 1;
355     }
356     return 0;
357   }
```

AS.c

```
1   #include <sys/stat.h>
2   #include <fcntl.h>
3   #include <errno.h>
4   #include <netdb.h>
5   #include <sys/types.h>
6   #include <sys/socket.h>
7   #include <netinet/in.h>
8   #include <arpa/inet.h>
9   #include <stdio.h>
10  #include <string.h>
11  #include <stdlib.h>
12  #include <unistd.h>
13  #include <time.h>
14 
```

```c
#include "des.c"

#define AS_HOST "127.0.0.1"
#define AS_PORT 23333

#define INVALIDID "Invalid ID"
#define INVALIDREQUEST "Invalid request"

const char username[] = "Bob";
const char clientIDList[][200] = {
    "Bob",
    "Alice"}; // 客户端 ID 的列表
const char clientIDListLen = 2;

/**
 * 新建 socket
 * @param addr char* 目标 ip
 * @param port int 目标 ip 的端口
 * @return sockaddr_in 结构体
*/
struct sockaddr_in *newSockaddr_in(char *addr, int port);

/**
 * 对接收到的请求进行处理
 * @param clientAddr struct sockaddr_in* 客户端地址
 * @param clientSock int 客户端套接字
*/
void serve(struct sockaddr_in *clientAddr, int clientSock);

/**
 * 检查是否含有对应的客户端 ID
 * @param clientID char* 客户端 ID 的字符串
 * @return 0 存在 1 不存在
*/
int checkClientIDList(char *clientID);

/**
 * 随机生成 key_client_TGS
*/
void getKeyCTGS(char res[17]);

int main()
{
  // 申请套接字
  int AS_socket;
  if ((AS_socket = socket(AF_INET, SOCK_STREAM, 0)) < 0)
  {
    // 报错并退出
    perror("AS socket");
    return 1;
  }
  else
  {
    // 生对应地址结构体
    struct sockaddr_in *AS_sockaddr = newSockaddr_in(AS_HOST, AS_PORT);

    // 进行地址和端口绑定
```

```
72      if (bind(AS_socket, (struct sockaddr *)AS_sockaddr, sizeof(struct
    sockaddr_in)) < 0)
73      {
74        perror("bind");
75        return 1;
76      }
77      printf("AS bind ip: %s:%d\n", AS_HOST, AS_PORT);
78
79      // 开始监听
80      if (listen(AS_socket, 20) < 0)
81      {
82        perror("listen");
83        return 1;
84      }
85      printf("AS start to listen\n");
86
87      // 阻塞式接受
88      while (1)
89      {
90        struct sockaddr_in clientAddr; // 客户端
91        socklen_t clientAddrSize = sizeof(clientAddr);
92        int clientSock;
93
94        // 如果接收到了
95        if ((clientSock = accept(AS_socket, (struct sockaddr *)&clientAddr,
    &clientAddrSize)) >= 0)
96        {
97          // 新建进程用于处理请求
98          int pid = fork();
99          if (pid == 0)
100         {
101           printf("-------START-------\n");
102           serve(&clientAddr, clientSock);
103           printf("--------END--------\n\n");
104           return 0;
105         }
106       }
107     }
108     close(AS_socket);
109   }
110   return 0;
111 }
112
113 struct sockaddr_in *newSockaddr_in(char *addr, int port)
114 {
115   struct sockaddr_in *sockaddr = (struct sockaddr_in *)malloc(sizeof(struct
    sockaddr_in));
116   sockaddr->sin_addr.s_addr = inet_addr(addr);
117   sockaddr->sin_family = AF_INET;
118   sockaddr->sin_port = htons(port);
119   return sockaddr;
120 }
121
122 void serve(struct sockaddr_in *clientAddr, int clientSock)
123 {
124   char request[1024]; // 缓冲区
125
126   // 显示请求方的相关信息
```

```
127     printf("from %s: %d\n", inet_ntoa(clientAddr->sin_addr), clientAddr-
    >sin_port);
128     int reqLen = read(clientSock, request, 1024);
129 
130     // 循环等待请求
131     while (reqLen <= 0)
132     {
133       reqLen = read(clientSock, request, 1024);
134     }
135     request[reqLen] = 0;
136     printf("[RECV]: %s\n", request);
137 
138     // 判断请求是否有效
139     char clientID[1024];
140     int reqArgsNum = sscanf(request, "authentication %s", clientID);
141 
142     // 无效 ID
143     if (reqArgsNum <= 0)
144     {
145       printf("[SEND]: %s\n", INVALIDREQUEST);
146       write(clientSock, INVALIDREQUEST, strlen(INVALIDREQUEST));
147       return;
148     }
149     // ID 不存在
150     else if (checkClientIDList(clientID))
151     {
152       printf("[SEND]: %s\n", INVALIDID);
153       write(clientSock, INVALIDID, strlen(INVALIDID));
154       return;
155     }
156 
157     // 如果 ID 有效，需要通过 keyClient 加密暂时生成的会话密钥 key_client_TGS
158 
159     // 获取 keyClient
160     FILE *keyClientFile = fopen("./keyClient", "r");
161     char keyClient[17];
162     fread(keyClient, 1, 16, keyClientFile);
163     keyClient[16] = 0;
164 
165     // 生成 key_client_TGS
166     char key_CTGS[17]; // key_CTGS 需要是16位16进制数
167     getKeyCTGS(key_CTGS);
168 
169     // 生成消息 A
170     char messageA[1024];
171 
172     // 加密消息 A
173     encodeFull(key_CTGS, 16, messageA, keyClient);
174     char messageA_transfered[1024];
175     char2intChar(messageA, 24, messageA_transfered);
176 
177     // 发送消息 A
178     write(clientSock, messageA_transfered, strlen(messageA_transfered));
179     printf("[SEND A]: %s\n", messageA_transfered);
180 
181     // 获取 keyTGS
182     FILE *keyTGSFile = fopen("./keyTGS", "r");
183     char keyTGS[17];
```

```
184     fread(keyTGS, 1, 16, keyTGSFile);
185     char messageB[1024], messageB_encoded[1024];
186 
187     // 生成消息B
188     sprintf(messageB, "<%s,%s,%ld,%s>", clientID, inet_ntoa(clientAddr->sin_addr),
    time(NULL) + 900l, key_CTGS);
189     printf("[MSG B]: %s\n", messageB);
190     int messageB_encoded_len = encodeFull(messageB, strlen(messageB),
    messageB_encoded, keyTGS);
191     messageB_encoded[messageB_encoded_len] = 0;
192     char messageB_encoded_transfered[1024];
193     char2intChar(messageB_encoded, messageB_encoded_len,
    messageB_encoded_transfered);
194 
195     // 发送消息 B
196     write(clientSock, messageB_encoded_transfered,
    strlen(messageB_encoded_transfered));
197     printf("[SEND B ]: %s\n", messageB_encoded_transfered);
198 
199     close(clientSock);
200     return;
201   }
202 
203   int checkClientIDList(char *clientID)
204   {
205     for (int i = 0; i < clientIDListLen; i++)
206     {
207       if (!strcmp(clientID, clientIDList[i]))
208       {
209         return 0;
210       }
211     }
212     return 1;
213   }
214 
215   void getKeyCTGS(char res[17])
216   {
217     unsigned char tmp;
218     srand(time(NULL) + 1);
219     for (int i = 0; i < 16; i++)
220     {
221       tmp = rand() % 16;
222       if (tmp >= 10)
223       {
224         res[i] = tmp - 10 + 'a';
225       }
226       else
227       {
228         res[i] = tmp + '0';
229       }
230     }
231     res[16] = 0;
232   }
```

TGS.c

```c
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

#include "des.c"

#define TGS_HOST "127.0.0.1"
#define TGS_PORT 23335

#define INVALIDID "Invalid (client or service)ID"
#define INVALIDREQUEST "Invalid request"
#define INVALIDADDRESS "Invalid address"
#define INVALIDTIME "Invalid time"

const char username[] = "Bob";
const char clientIDList[][200] = {
    "Bob",
    "Alice"}; // 客户端 ID 的列表

const char serviceIDList[][200] = {
    "testService",
    "service"};
const char clientIDListLen = 2;
const char serviceIDListLen = 2;

/**
 * 新建 socket
 * @param addr char* 目标 ip
 * @param port int 目标 ip 的端口
 * @return sockaddr_in 结构体
*/
struct sockaddr_in *newSockaddr_in(char *addr, int port);

/**
 * 对接收到到的东西进行处理
 * @param clientAddr struct sockaddr_in* 客户端地址
 * @param clientSock int 客户端套接字
*/
void serve(struct sockaddr_in *clientAddr, int clientSock);

/**
 * 检查是否含有对应的客户端 ID
 * @param clientID char* 客户端 ID 的字符串
 * @return 0 存在 1 不存在
*/
```

```c
int checkClientIDList(char *clientID);

/**
 * 生成 key_client_SS
*/
void getKeyCSS(char res[17]);

/**
 * 检查服务 ID 是否存在
 * @param serviceID char* 服务 ID 字符串
 * @return 0 存在 1 不存在
*/
int checkServiceIDList(char *serviceID);

int main()
{
  // 申请套接字
  int TGS_socket;
  if ((TGS_socket = socket(AF_INET, SOCK_STREAM, 0)) < 0)
  {
    // 报错并退出
    perror("TGS socket");
    return 1;
  }
  else
  {
    // 生对应地址结构体
    struct sockaddr_in *TGS_sockaddr = newSockaddr_in(TGS_HOST, TGS_PORT);

    // 进行地址和端口绑定
    if (bind(TGS_socket, (struct sockaddr *)TGS_sockaddr, sizeof(struct sockaddr_in)) < 0)
    {
      perror("bind");
      return 1;
    }
    printf("TGS bind ip: %s:%d\n", TGS_HOST, TGS_PORT);

    // 开始监听
    if (listen(TGS_socket, 20) < 0)
    {
      perror("listen");
      return 1;
    }
    printf("TGS start to listen\n");

    // 阻塞式接受
    while (1)
    {
      struct sockaddr_in clientAddr; // 客户端
      socklen_t clientAddrSize = sizeof(clientAddr);
      int clientSock;

      // 如果接收到了
      if ((clientSock = accept(TGS_socket, (struct sockaddr *)&clientAddr, &clientAddrSize)) >= 0)
      {
        // 新建子进程用于处理请求
```

```
112           int pid = fork();
113           if (pid == 0)
114           {
115             printf("-------START-------\n");
116             serve(&clientAddr, clientSock);
117             printf("--------END--------\n\n");
118             return 0;
119           }
120         }
121       }
122       close(TGS_socket);
123     }
124     return 0;
125   }
126
127   struct sockaddr_in *newSockaddr_in(char *addr, int port)
128   {
129     struct sockaddr_in *sockaddr = (struct sockaddr_in *)malloc(sizeof(struct
      sockaddr_in));
130     sockaddr->sin_addr.s_addr = inet_addr(addr);
131     sockaddr->sin_family = AF_INET;
132     sockaddr->sin_port = htons(port);
133     return sockaddr;
134   }
135
136   void serve(struct sockaddr_in *clientAddr, int clientSock)
137   {
138     char request[1024]; // 请求
139
140     // 获取 key_TGS
141     FILE *keyTGSFile = fopen("./keyTGS", "r");
142     char keyTGS[17];
143     fread(keyTGS, 1, 17, keyTGSFile);
144
145     // 显示请求方的相关信息
146     printf("from %s: %d\n", inet_ntoa(clientAddr->sin_addr), clientAddr-
      >sin_port);
147
148     // 循环等待请求
149     int reqLen = read(clientSock, request, 1024);
150     while (reqLen <= 0)
151     {
152       reqLen = read(clientSock, request, 1024);
153     }
154     request[reqLen] = 0;
155
156     char clientID_in_B[1024], serviceID[1024], clientAddr_in_B[1024],
      messageB[1024];
157     time_t validity;
158     int requestArgsNum = sscanf(request, "%[^,],%s", serviceID, messageB);
159
160     // 判断请求是否有效
161     if (requestArgsNum == 0 || strlen(messageB) <= 0)
162     {
163       printf("[SEND]: %s\n", INVALIDREQUEST);
164       write(clientSock, INVALIDREQUEST, strlen(INVALIDREQUEST));
165       return;
166     }
```

```
167     printf("[RECV C]: %s\n", request); // 显示收到的信息
168
169     // 解码消息 B
170     char messageB_encoded[1024]; // 加密后的消息 B
171     char messageB_decoded[1024]; // 解密后的消息 B
172     int messageB_encoded_len = intChar2char(messageB, strlen(messageB),
    messageB_encoded);
173     int messageB_decoded_len = decodeFull(messageB_encoded, messageB_encoded_len,
    messageB_decoded, keyTGS);
174     messageB[messageB_decoded_len] = 0;
175     printf("[MSG B]: %s\n", messageB_decoded);
176
177     // 判断消息 B 是否有效
178     char keyCTGS[17]; // key_client_TGS
179     sscanf(messageB_decoded, "<%[^,],%[^,],%ld, %[^>]>", clientID_in_B,
    clientAddr_in_B, &validity, keyCTGS);
180     char *clientAddrStr = inet_ntoa(clientAddr->sin_addr);
181     if (checkClientIDList(clientID_in_B) || checkServiceIDList(serviceID)) // ID
    无效
182     {
183       printf("[SEND]: %s\n", INVALIDID);
184       write(clientSock, INVALIDID, strlen(INVALIDID));
185       return;
186     }
187     else if (strcmp(clientAddrStr, clientAddr_in_B) != 0) // 地址无效
188     {
189       printf("[SEND]: %s\n", INVALIDADDRESS);
190       write(clientSock, INVALIDADDRESS, strlen(INVALIDADDRESS));
191       return;
192     }
193
194     // 获取密钥 keySS
195     FILE *keySSFile = fopen("./keySS", "r");
196     char keySS[17];
197     fread(keySS, 1, 16, keySSFile);
198
199     // 生成 key_client_SS
200     char keyCSS[17];
201     getKeyCSS(keyCSS);
202
203     // 生成 ST
204     char ST[1024], ST_encoded[1024], ST_encoded_transfered[1024]; // ST 相关
205     int ST_encoded_len = 0;
206     sprintf(ST, "<%s,%s,%ld,%s>", clientID_in_B, clientAddr_in_B, time(NULL) +
    900l, keyCSS);
207     printf("[MSG ST]: %s\n", ST);
208
209     // 加密 ST
210     ST_encoded_len = encodeFull(ST, strlen(ST), ST_encoded, keySS);
211     char2intChar(ST_encoded, ST_encoded_len, ST_encoded_transfered);
212
213     // 生成消息 E
214     char messageE[1024];
215     sprintf(messageE, "%s,%s", serviceID, ST_encoded_transfered);
216
217     // 发送消息 E
218     write(clientSock, messageE, strlen(messageE));
219     printf("[SEND E]: %s\n", messageE);
```

```
220
221     // 接收请求（消息 D）
222     reqLen = read(clientSock, request, 1024);
223     request[reqLen] = 0;
224     // 判断请求有效性
225     if (reqLen <= 0) // 无效请求
226     {
227       printf("[SEND]: %s\n", INVALIDREQUEST);
228       write(clientSock, INVALIDREQUEST, strlen(INVALIDREQUEST));
229       return;
230     }
231     char messageD[1024]; // 消息 D
232     strcpy(messageD, request);
233     printf("[RECV D]: %s\n", messageD);
234
235     // 解密消息 D
236     char messageD_encoded[1024];
                                // 加密后的消息 D
237     char messageD_decoded[1024];
                                // 解密后的消息 D
238     int messageD_encoded_len = intChar2char(messageD, strlen(messageD),
    messageD_encoded);                      // 解密后的消息 D 的长度
239     int messageD_decoded_len = decodeFull(messageD_encoded, messageD_encoded_len,
    messageD_decoded, keyCTGS); // 解密后的消息 D 的长度
240     messageD_decoded[messageD_decoded_len] = 0;
241     printf("[MSG D]: %s\n", messageD_decoded);
242
243     // 判断消息 D 的有效性
244     time_t now = time(NULL), timestamp;
245     char clientID_in_D[1024];
246     sscanf(messageD_decoded, "<%[^,],%ld>", clientID_in_D, &timestamp);
247     if (timestamp < now || timestamp > now + 900l) // 无效时间戳
248     {
249       printf("[SEND]: %s\n", INVALIDTIME);
250       write(clientSock, INVALIDTIME, strlen(INVALIDTIME));
251       return;
252     }
253     else if (strcmp(clientID_in_D, clientID_in_B) != 0) // ID 不一致
254     {
255       printf("[SEND]: %s\n", INVALIDID);
256       write(clientSock, INVALIDID, strlen(INVALIDID));
257       return;
258     }
259
260     // 生成消息 F
261     char messageF[1024]; // 消息 F
262     strcpy(messageF, keyCSS);
263     printf("[MSG F]: %s\n", messageF);
264
265     // 加密消息 F
266     char messageF_encoded[1024];
                  // 加密后的消息 F
267     char messageF_encoded_transfered[1024];
                   // 加密后的转换显示的消息 F
268     int messageF_encoded_len = encodeFull(messageF, strlen(messageF),
    messageF_encoded, keyCTGS); // 加密后的消息 F 的长度
269     char2intChar(messageF_encoded, messageF_encoded_len,
    messageF_encoded_transfered);
```

```
270
271     // 发送消息 F
272     write(clientSock, messageF_encoded_transfered,
    strlen(messageF_encoded_transfered));
273     printf("[SEND F ]: %s\n", messageF_encoded_transfered);
274
275     close(clientSock);
276     fclose(keySSFile);
277     fclose(keyTGSFile);
278     return;
279   }
280
281   int checkClientIDList(char *clientID)
282   {
283     for (int i = 0; i < clientIDListLen; i++)
284     {
285       if (!strcmp(clientID, clientIDList[i]))
286       {
287         return 0;
288       }
289     }
290     return 1;
291   }
292
293   int checkServiceIDList(char *serviceID)
294   {
295     for (int i = 0; i < serviceIDListLen; i++)
296     {
297       if (!strcmp(serviceID, serviceIDList[i]))
298       {
299         return 0;
300       }
301     }
302     return 1;
303   }
304
305   void getKeyCSS(char res[17])
306   {
307     unsigned char tmp;
308     srand(time(NULL) + 1);
309     for (int i = 0; i < 16; i++)
310     {
311       tmp = rand() % 16;
312       if (tmp >= 10)
313       {
314         res[i] = tmp - 10 + 'a';
315       }
316       else
317       {
318         res[i] = tmp + '0';
319       }
320     }
321     res[16] = 0;
322   }
```

SS.c

```c
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

#include "des.c"

#define SS_HOST "127.0.0.1"
#define SS_PORT 23334

#define INVALIDID "Invalid (client or service)ID"
#define INVALIDREQUEST "Invalid request"
#define INVALIDADDRESS "Invalid address"
#define INVALIDTIME "Invalid time"

const char username[] = "Bob";
const char clientIDList[][200] = {
    "Bob",
    "Alice"}; // 客户端 ID 的列表

const char serviceIDList[][200] = {
    "testService",
    "service"};
const char clientIDListLen = 2;
const char serviceIDListLen = 2;

/**
 * 新建 socket
 * @param addr char* 目标 ip
 * @param port int 目标 ip 的端口
 * @return sockaddr_in 结构体
*/
struct sockaddr_in *newSockaddr_in(char *addr, int port);

/**
 * 对接收到到的东西进行处理
 * @param clientAddr struct sockaddr_in* 客户端地址
 * @param clientSock int 客户端套接字
*/
void serve(struct sockaddr_in *clientAddr, int clientSock);

/**
 * 检查是否含有对应的客户端 ID
 * @param clientID char* 客户端 ID 的字符串
 * @return 0 存在 1 不存在
*/
```

```c
int checkClientIDList(char *clientID);

/**
 * 生成 key_client_SS
*/
void getKeyCSS(char res[17]);

/**
 * 检查是否含有对应的服务 ID
 * @param serviceID char* 客户端 ID 的字符串
 * @return 0 存在 1 不存在
*/
int checkServiceIDList(char *serviceID);

int main()
{
  int ss_socket;
  if ((ss_socket = socket(AF_INET, SOCK_STREAM, 0)) < 0)
  {
    perror("ss socket");
    return 1;
  }
  else
  {
    // ss
    struct sockaddr_in *SS_sockaddr = newSockaddr_in(SS_HOST, SS_PORT);

    // 绑定
    if (bind(ss_socket, (struct sockaddr *)SS_sockaddr, sizeof(struct sockaddr_in)) < 0)
    {
      perror("bind");
      return 1;
    }
    printf("ss bind ip: %s:%d\n", SS_HOST, SS_PORT);

    // 开始监听
    if (listen(ss_socket, 20) < 0)
    {
      perror("listen");
      return 1;
    }
    printf("ss start to listen\n");

    // 接收
    while (1)
    {
      struct sockaddr_in clientAddr; // 客户端
      socklen_t clientAddrSize = sizeof(clientAddr);
      int clientSock;

      // 如果接收到了
      if ((clientSock = accept(ss_socket, (struct sockaddr *)&clientAddr, &clientAddrSize)) >= 0)
      {
        int pid = fork();
        if (pid == 0)
        {
```

```
            printf("-------START-------\n");
            serve(&clientAddr, clientSock);
            printf("--------END--------\n\n");
            return 0;
          }
        }
      }
      close(ss_socket);
    }
    return 0;
}

struct sockaddr_in *newSockaddr_in(char *addr, int port)
{
    struct sockaddr_in *sockaddr = (struct sockaddr_in *)malloc(sizeof(struct sockaddr_in));
    sockaddr->sin_addr.s_addr = inet_addr(addr);
    sockaddr->sin_family = AF_INET;
    sockaddr->sin_port = htons(port);
    return sockaddr;
}

void serve(struct sockaddr_in *clientAddr, int clientSock)
{
    char request[1024]; // 请求

    // 显示请求方的相关信息
    printf("from %s: %d\n", inet_ntoa(clientAddr->sin_addr), clientAddr->sin_port);

    // 循环等待请求
    int reqLen = read(clientSock, request, 1024);
    while (reqLen <= 0)
    {
      reqLen = read(clientSock, request, 1024);
    }
    request[reqLen] = 0;

    // 判断请求有效性
    char serviceID[1024]; // 服务 ID
    char ST[1024];        // ST 凭据
    if (sscanf(request, "%[^,],%s", serviceID, ST) <= 0)
    {
      printf("[SEND]: %s\n", INVALIDREQUEST);
      write(clientSock, INVALIDREQUEST, strlen(INVALIDREQUEST));
      return;
    }
    else if (checkServiceIDList(serviceID))
    {
      printf("[SEND]: %s\n", INVALIDID);
      write(clientSock, INVALIDID, strlen(INVALIDID));
      return;
    }

    // 得到消息 E
    char messageE[1024];
    strcpy(messageE, request);
    printf("[RECV E]: %s\n", request); // 显示收到的信息
```

```
168
169     // 获取 key_SS
170     FILE *keySSFile = fopen("./keySS", "r");
171     char keySS[17];
172     fread(keySS, 1, 16, keySSFile);
173
174     // 解密 ST
175     char ST_encoded[1024];
       // 加密后的 ST
176     char ST_decoded[1024];
       // 解密收的 ST
177     int ST_encoded_len = intChar2char(ST, strlen(ST), ST_encoded);
       // 加密后的 ST 长度
178     int ST_decoded_len = decodeFull(ST_encoded, ST_encoded_len, ST_decoded,
    keySS); // 解密后的 ST 长度
179     printf("[MSG ST]: %s\n", ST_decoded);
180
181     // 判断 ST 有效性
182     char clientID_in_E[1024];
183     char clientAddr_in_E[1024];
184     char keyCSS[17];
185     time_t timestamp;
186     int requestArgsNum = sscanf(ST_decoded, "<%[^,],%[^,],%ld,%[^>]>",
    clientID_in_E, clientAddr_in_E, &timestamp, keyCSS); // 读取到的参数个数
187     char *clienAddrStr = inet_ntoa(clientAddr->sin_addr);
                                                  // ST 中的地址
188     time_t now = time(NULL);
                                                 // 当前时间
189
190     if (requestArgsNum <= 0) // 参数不够
191     {
192       printf("[SEND]: %s\n", INVALIDREQUEST);
193       write(clientSock, INVALIDREQUEST, strlen(INVALIDREQUEST));
194       return;
195     }
196     else if (strcmp(clienAddrStr, clientAddr_in_E) != 0) // 地址不对
197     {
198       printf("[SEND]: %s\n", INVALIDADDRESS);
199       write(clientSock, INVALIDADDRESS, strlen(INVALIDADDRESS));
200       return;
201     }
202     else if (timestamp < now) // 时间戳不对
203     {
204       printf("[NOW]: %ld\n", now);
205       printf("[SEND]: %s\n", INVALIDTIME);
206       write(clientSock, INVALIDTIME, strlen(INVALIDTIME));
207       return;
208     }
209
210     // 避免粘包，进行一次无效发送
211     write(clientSock, "RECV G", 1024);
212
213     // 接收请求（消息 G）
214     reqLen = read(clientSock, request, 1024);
215     request[reqLen];
216
217     // 判断请求有效性
218     if (reqLen <= 0)
```

```
219     {
220       printf("[SEND]: %s\n", INVALIDREQUEST);
221       write(clientSock, INVALIDREQUEST, strlen(INVALIDREQUEST));
222       return;
223     }
224 
225     // 得到消息 G
226     char messageG[1024];
227     strcpy(messageG, request);
228     printf("[RECV G ]: %s\n", messageG);
229 
230     // 解密消息 G
231     char messageG_encoded[1024]; // 加密后的消息 G
232     char messageG_decoded[1024]; // 解密后的消息 G
233     int messageG_encoded_len = intChar2char(messageG,
234                                             strlen(messageG),
235                                             messageG_encoded); // 加密后的消息 G 长度
236     int messageG_decoded_len = decodeFull(messageG_encoded,
237                                           messageG_encoded_len,
238                                           messageG_decoded,
239                                           keyCSS); // 解密后的消息 G 长度
240     printf("[MSG G]: %s\n", messageG_decoded);
241 
242     // 验证消息 G 的有效性
243     char clientID_in_G[1024];
244     requestArgsNum = sscanf(messageG_decoded, "<%[^,],%ld>", clientID_in_G,
    &timestamp);
245     if (requestArgsNum <= 0) // 参数不够
246     {
247       printf("[SEND]: %s\n", INVALIDREQUEST);
248       write(clientSock, INVALIDREQUEST, strlen(INVALIDREQUEST));
249       return;
250     }
251     else if (strcmp(clientID_in_E, clientID_in_G) != 0) // 客户端 ID 不对
252     {
253       printf("[SEND]: %s\n", INVALIDID);
254       write(clientSock, INVALIDID, strlen(INVALIDID));
255       return;
256     }
257 
258     // 生成消息 H
259     time_t TS = timestamp + 1l;
260     char messageH[1024];
261     sprintf(messageH, "<%s,%ld>", clientID_in_E, TS);
262     printf("[MSG H]: %s\n", messageH);
263 
264     // 加密消息 H
265     char messageH_encoded[1024];             // 加密后的消息 H
266     char messageH_encoded_transfered[1024]; // 加密后进行转换的消息 H
267     int messageH_encoded_len = encodeFull(messageH,
268                                           strlen(messageH),
269                                           messageH_encoded,
270                                           keyCSS); // 加密后消息 H 的长度
271     char2intChar(messageH_encoded,
272                  messageH_encoded_len,
273                  messageH_encoded_transfered);
274 
275     // 发送消息 H
```

```
276     write(clientSock, messageH_encoded_transfered,
    strlen(messageH_encoded_transfered));
277     printf("[SEND H]: %s\n", messageH_encoded_transfered);
278 
279     free(keySSFile);
280     close(clientSock);
281     return;
282   }
283 
284   int checkClientIDList(char *clientID)
285   {
286     for (int i = 0; i < clientIDListLen; i++)
287     {
288       if (!strcmp(clientID, clientIDList[i]))
289       {
290         return 0;
291       }
292     }
293     return 1;
294   }
295 
296   int checkServiceIDList(char *serviceID)
297   {
298     for (int i = 0; i < serviceIDListLen; i++)
299     {
300       if (!strcmp(serviceID, serviceIDList[i]))
301       {
302         return 0;
303       }
304     }
305     return 1;
306   }
```