**Содержание**:

[1. Введение 3](#__RefHeading___Toc779_1887482898)

[2. Краткие теоретические сведения 4](#__RefHeading___Toc781_1887482898)

[3. Блок-схема алгоритма 13](#__RefHeading___Toc783_1887482898)

[4. Демонстрация работы программы 14](#__RefHeading___Toc785_1887482898)

[5. Вывод 15](#__RefHeading___Toc787_1887482898)

[Приложение 1. Исходный код программы 16](#__RefHeading___Toc789_1887482898)

# **Введение**

Цель лабораторной работы: cоздать приложение, реализующее протокол распределения ключей Kerberos, включая процедуру, реализующую Алгоритм DES.

В интерфейсе приложения должны быть наглядно представлены:

* Исходные данные протокола (модули, ключи, секретные данные и т.п.);
* Данные, передаваемые по сети каждой из сторон;

Проверки, выполняемые каждым из участников.

# **Краткие теоретические сведения**

**Протокол Kerberos**

Протокол Kerberos является одной из реализаций протокола аутентификации с использованием третьей стороны, призванной уменьшить количество сообщений, которыми обмениваются стороны.

Протокол Kerberos, достаточно гибкий и имеющий возможности тонкой настройки под конкретные применения, существует в нескольких версиях. Мы рассмотрим упрощенный механизм аутентификации, реализованный с помощью протокола Kerberos версии 5 (рис. 1):



Рисунок 1 Схема протокола Kerberos

Прежде всего стоит сказать, что при использовании Kerberos нельзя напрямую получить доступ к какому-либо целевому серверу. Чтобы запустить собственно процедуру аутентификации, необходимо обратиться к специальному серверу аутентификации с запросом, содержащим логин пользователя. Если сервер не находит автора запроса в своей базе данных, запрос отклоняется. В противном случае сервер аутентификации работает по следующему рабочему процессу:

**Рабочий этап:**

Пусть клиент C собирается начать взаимодействие с сервером SS (англ. *Service* *Server* - *сервер*, предоставляющий сетевые сервисы). В несколько упрощенном виде, протокол предполагает следующие шаги:

1. **C->AS: {c}.**

Клиент C посылает серверу аутентификации AS свой идентификатор c (идентификатор передается открытым текстом).

1. **AS->C: {{TGT}KAS\_TGS, KC\_TGS}KC,**

где:

* 1. KC - основной ключ C ;
  2. KC\_TGS - ключ, выдаваемый C для доступа к серверу выдачи разрешений *TGS* ;
  3. {TGT} - *Ticket* Granting *Ticket* - билет на доступ к серверу выдачи разрешений

{TGT}={c,*tgs*,t1,p1, KC\_TGS}, где *tgs* - идентификатор сервера выдачи разрешений, t1 - отметка времени, p1 - *период действия* билета.

Запись \{ \cdot \} K_{X}здесь и далее означает, что содержимое фигурных скобок зашифровано на ключе KX (Алгоритм шифрования приводится ниже).

На этом шаге сервер аутентификации AS, проверив, что клиент C имеется в его базе, возвращает ему билет для доступа к серверу выдачи разрешений и ключ для взаимодействия с сервером выдачи разрешений. Вся посылка зашифрована на ключе клиента C. Таким образом, даже если на первом шаге взаимодействия идентификатор с послал не клиент С, а нарушитель X, то полученную от AS посылку X расшифровать не сможет.

Получить доступ к содержимому билета TGT не может не только нарушитель, но и клиент C, т.к. билет зашифрован на ключе, который распределили между собой сервер аутентификации и сервер выдачи разрешений.

1. **C->*****TGS*: {TGT}KAS\_TGS, {Aut1} KC\_TGS, {ID}**

где {Aut1} - аутентификационный блок - Aut1 = {с,t2}, t2 - метка времени; ID - идентификатор запрашиваемого сервиса (в частности, это может быть идентификатор сервера SS ).

Клиент C на этот раз обращается к серверу выдачи разрешений ТGS. Он пересылает полученный от AS билет, зашифрованный на ключе KAS\_TGS, и аутентификационный блок, содержащий идентификатор c и метку времени, показывающую, когда была сформирована посылка. Сервер выдачи разрешений расшифровывает билет TGT и получает из него информацию о том, кому был выдан билет, когда и на какой срок, ключ шифрования, сгенерированный сервером AS для взаимодействия между клиентом C и сервером *TGS*. С помощью этого ключа расшифровывается аутентификационный блок. Если метка в блоке совпадает с меткой в билете, это доказывает, что посылку сгенерировал на самом деле С (ведь только он знал ключ KC\_TGS и мог правильно зашифровать свой идентификатор). Далее делается проверка времени действия билета и времени отправления посылки **3**). Если проверка проходит и действующая в системе политика позволяет клиенту С обращаться к клиенту SS, тогда выполняется шаг **4**).

1. ***TGS*->C: {{*****TGS*}KTGS\_SS,KC\_SS}KC\_TGS,**

где KC\_SS - ключ для взаимодействия C и SS, {*TGS*} - *Ticket* Granting Service - билет для доступа к SS (обратите внимание, что такой же аббревиатурой в описании протокола обозначается и сервер выдачи разрешений). {*TGS*} ={с,ss,t3,p2, KC\_SS }.

Сейчас сервер выдачи разрешений *TGS* посылает клиенту C ключ шифрования и билет, необходимые для доступа к серверу SS. Структура билета такая же, как на шаге 2): идентификатор того, кому выдали билет; идентификатор того, для кого выдали билет; отметка времени; *период действия*; ключ шифрования.

1. **C->SS: {*****TGS*}KTGS\_SS, {Aut2} KC\_SS**

где Aut2={c,t4}.

Клиент C посылает билет, полученный от сервера выдачи разрешений, и свой аутентификационный блок серверу SS, с которым хочет установить сеанс защищенного взаимодействия. Предполагается, что SS уже зарегистрировался в системе и распределил с сервером *TGS* ключ шифрования KTGS\_SS. Имея этот ключ, он может расшифровать билет, получить ключ шифрования KC\_SS и проверить подлинность *отправителя сообщения*.

1. **SS->C: {t4+1}KC\_SS**

Смысл последнего шага заключается в том, что теперь уже SS должен доказать C свою подлинность. Он может сделать это, показав, что правильно расшифровал предыдущее сообщение. Вот поэтому, SS берет отметку времени из аутентификационного блока C, изменяет ее заранее определенным образом (увеличивает на 1), шифрует на ключе KC\_SS и возвращает C.

Если все шаги выполнены правильно и все проверки прошли успешно, то стороны взаимодействия C и SS, во-первых, удостоверились в подлинности друг друга, а во-вторых, получили *ключ* шифрования для защиты сеанса связи - *ключ* KC\_SS.

Нужно отметить, что в процессе сеанса работы клиент проходит шаги 1) и 2) только один раз. Когда нужно получить билет на *доступ* к другому серверу (назовем его SS1 ), клиент С обращается к серверу выдачи разрешений *TGS* с уже имеющимся у него билетом, т.е. протокол выполняется начиная с шага 3).

В алгоритме Kerberos могут применяться различные алгоритмы блочного симметричного шифрования. Для целей настоящей работы будем использовать алгоритм DES:

**Алгоритм DES Основные сведения**

Одной из наиболее известных криптографических систем с закрытым ключом является DES – Data Encryption Standard. Эта система первой получила статус государственного стандарта в области шифрования данных. Она разработана специалистами фирмы IBM и вступила в действие в США 1977 году. Алгоритм DES по-прежнему широко применяется и заслуживает внимания при изучении блочных шифров с закрытым ключом.

Стандарт DES построен на комбинированном использовании перестановки, замены и гаммирования. Шифруемые данные должны быть представлены в двоичном виде.

*DES* является классической *сетью Фейстеля* с двумя ветвями. Данные шифруются 64-битными блоками, используя 56-битный ключ. Алгоритм преобразует за несколько *раундов* 64-битный вход в 64-битный выход. Длина ключа равна 56 битам. Процесс шифрования состоит из четырех этапов. На первом из них выполняется начальная перестановка (*IP*) 64-битного исходного текста (забеливание), во время которой биты переупорядочиваются в соответствии со стандартной таблицей. Следующий этап состоит из 16 *раундов* одной и той же функции, которая использует операции сдвига и подстановки. На третьем этапе левая и правая половины выхода последней (16-й) итерации меняются местами. Наконец, на четвертом этапе выполняется перестановка IP-1 результата, полученного на третьем этапе. Перестановка IP-1 инверсна начальной перестановке.

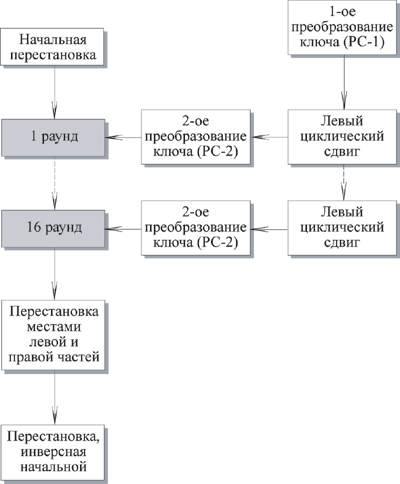


Рисунок 2 Общая схема DES

**Шифрование**

**Начальная перестановка**

Начальная перестановка и ее инверсия определяются стандартной таблицей. Если М- это произвольные 64 бита, то X = IP (M)-переставленные 64 бита. Если применить обратную функцию перестановки Y = IP-1 (X) = IP-1 (IP(M)), то получится первоначальная последовательность бит.

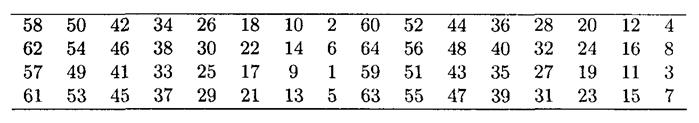


Рисунок 3 - DES. Начальная перестановка

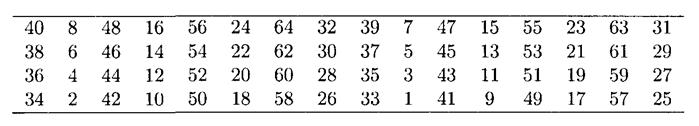


Рисунок 4 - DES. Заключительная перестановка

**Последовательность преобразований отдельного раунда**

Теперь рассмотрим последовательность преобразований, используемую в каждом *раунде*.

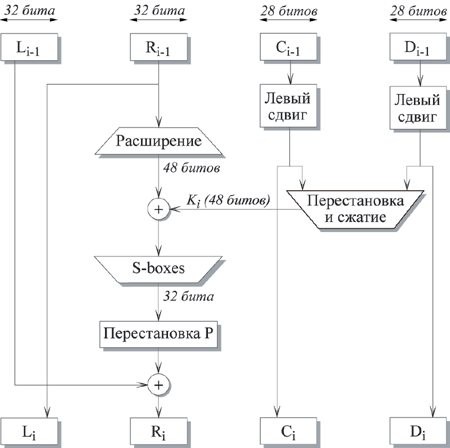


Рисунок 5 - I-ый раунд DES

64-битный входной блок проходит через 16 *раундов*, при этом на каждой итерации получается промежуточное 64-битное значение. Левая и правая части каждого промежуточного значения трактуются как отдельные 32-битные значения, обозначенные *L* и *R*. Каждую итерацию можно описать следующим образом:

Li = Ri-1

Ri = Li-1 F(Ri-1, Ki)

Где обозначает операцию XOR.

Таким образом, выход левой половины *Li* равен входу правой половины *Ri-1*. Выход правой половины *Ri*является результатом применения операции XOR к *Li-1* и функции *F*, зависящей от *Ri-1* и *Ki*.

Рассмотрим функцию *F* более подробно.

*Ri*, которое подается на вход функции *F*, имеет длину 32 бита. Вначале Ri расширяется до 48 бит, используя таблицу, которая определяет перестановку плюс расширение на 16 бит. Расширение происходит следующим образом. 32 бита разбиваются на группы по 4 бита и затем расширяются до 6 бит, присоединяя крайние биты из двух соседних групп. Например, если часть входного сообщения

. . . efgh ijkl mnop . . .

то в результате расширения получается сообщение

. . . defghi hijklm lmnopq . . .

После этого для полученного 48-битного значения выполняется операция XOR с 48-битным *подключом Ki*. Затем полученное 48-битное значение подается на вход функции подстановки, результатом которой является 32-битное значение.

Подстановка состоит из восьми *S-boxes,* каждый из которых на входе получает 6 бит, а на выходе создает 4 бита. Эти преобразования определяются специальными таблицами. Первый и последний биты входного значения *S-box* определяют номер строки в таблице, средние 4 бита определяют номер столбца. Пересечение строки и столбца определяет 4-битный выход. Например, если входом является 011011, то номер строки равен 01 (строка 1) и номер столбца равен 1101 (столбец 13). Значение в строке 1 и столбце 13 равно 5, т.е. выходом является 0101.

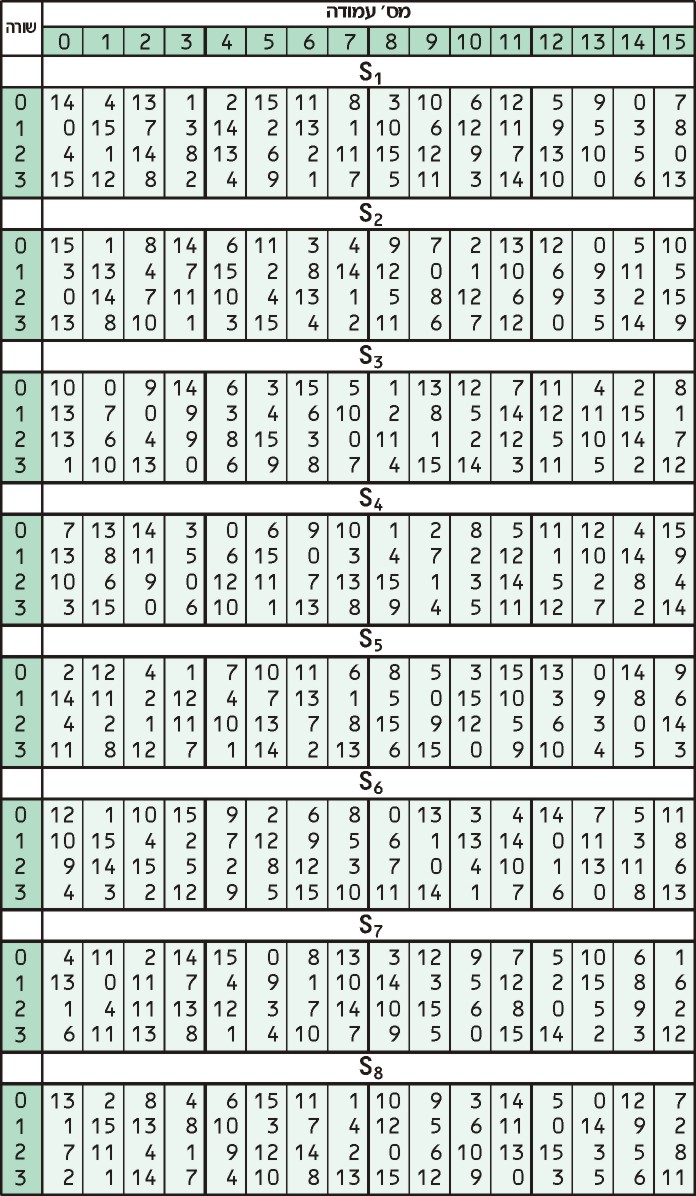


Рисунок 6 - S-boxes

Далее полученное 32-битное значение обрабатывается с помощью перестановки *Р*, целью которой является максимальное переупорядочивание бит, чтобы в следующем *раунде* шифрования с большой вероятностью каждый бит обрабатывался другим *S-box*.

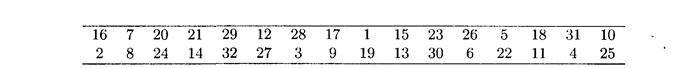


Рисунок 7 - Перестановка с помощью Р-блоков

**Создание подключей**

Ключ для отдельного *раунда Ki*состоит из 48 бит. Ключи *Ki* получаются по следующему алгоритму. Для 56-битного ключа, используемого на входе алгоритма (если используется 64-битный ключ, то, как видно из рис. 5 убираются биты 64, 56, 48, 40, 32, 16, 8) , вначале выполняется перестановка в соответствии с таблицей Permuted Choice 1 (РС-1).

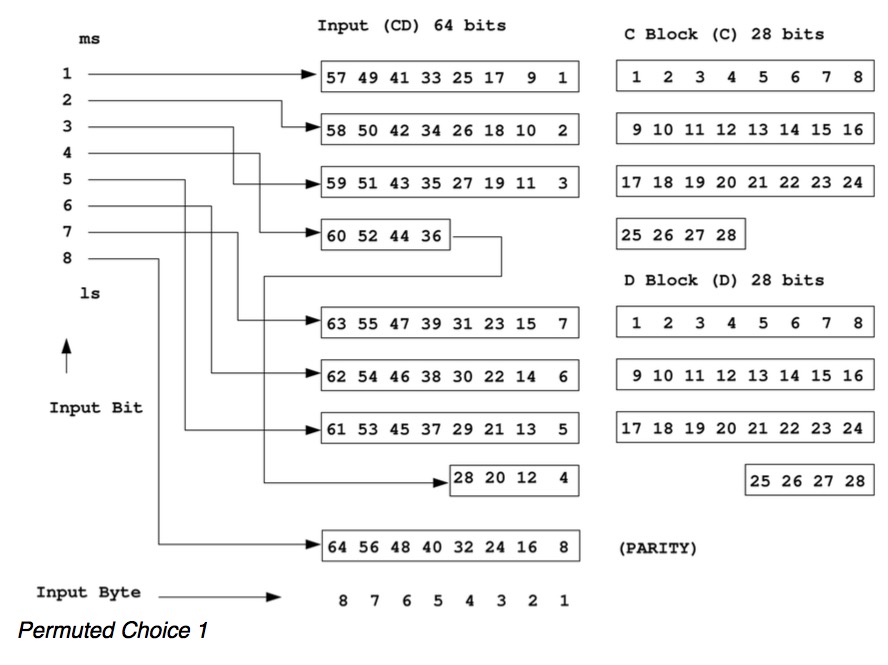


Рисунок 8 - Схема Permuted Choice

Полученный 56-битный ключ разделяется на две 28-битные части, обозначаемые как C0 и D0 соответственно. На каждом *раунде Ci* и *Di*независимо циклически сдвигаются влево на 1 или 2 бита, в зависимости от номера *цикла*.

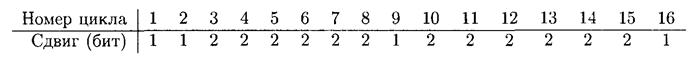


Рисунок 9 - Сдвиг ключа в зависимости от номера цикла

Полученные значения являются входом следующего *раунда*. Они также представляют собой вход в Permuted Choice 2 (РС-2), который создает 48-битное выходное значение, являющееся входом функции *F*(*Ri-1*, *Ki*).

**Дешифрование**

Процесс дешифрования аналогичен процессу шифрования. На входе алгоритма используется зашифрованный текст, но ключи *Ki* используются в обратной последовательности. *K16* используется на первом *раунде*, *K1* используется на последнем *раунде*.

# **Блок-схема алгоритма**

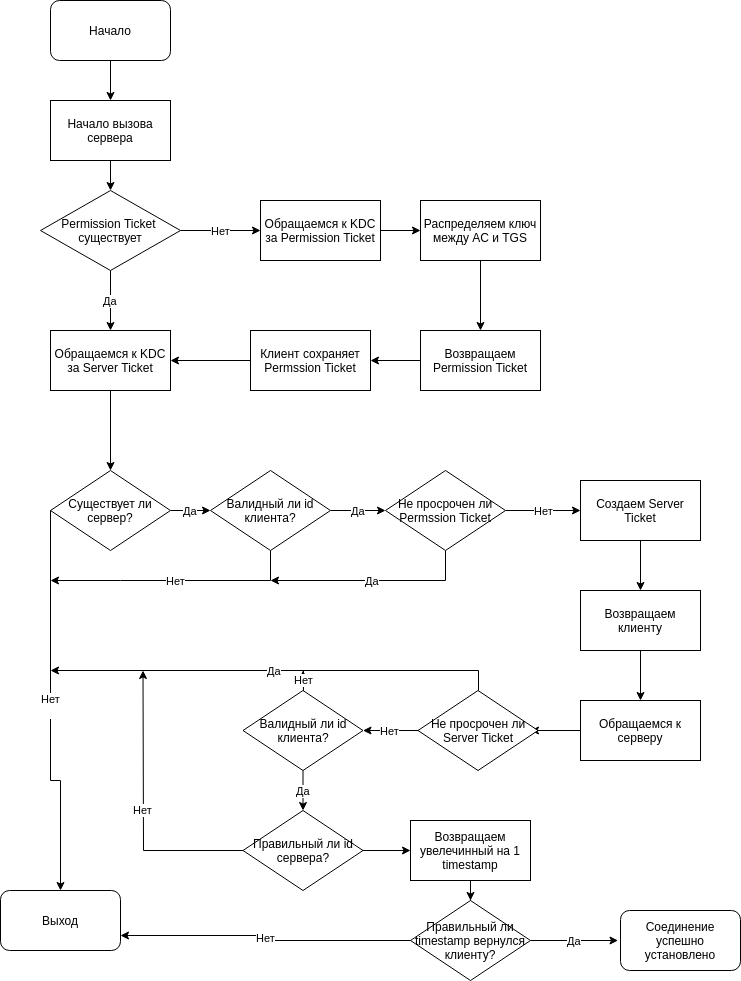


Рисунок 3.1. Блок-схема алгоритма программы

# **Демонстрация работы программы**

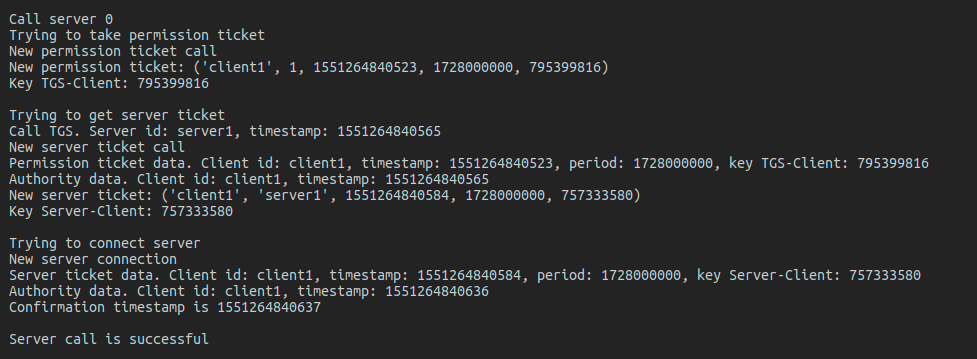


Рисунок 4.1. Общение клиента с первым сервером.

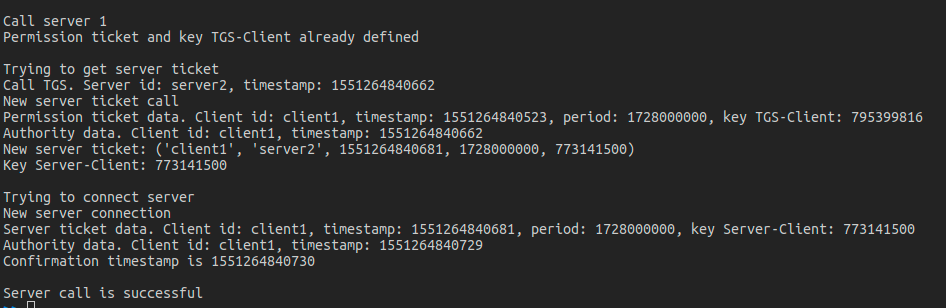


Рисунок 4.2. Общение клиента со вторым сервером

Рисунок 4.3. Первоначальные данные



Рисунок 4.4. Пример DES шифрования и дешифрования ключа

# **Вывод**

В ходе выполнения лабораторной работы были изучены cоздано приложение, реализующее протокол распределения ключей Kerberos, включая процедуру, реализующую Алгоритм DES.

# **Приложение 1. Исходный код программы**

import random

from collections import namedtuple

import time

from pydes import des

from ast import literal\_eval as make\_tuple

current\_milli\_time = lambda: int(round(time.time() \* 1000))

hours\_to\_milli = lambda hour: hour \* 3600 \* 10000

class DesEncrpter:

# Encrypts or decrypts any tuple of data with DES encryption

# Uses pydes library by Robin David (https://github.com/RobinDavid/pydes)

def encrypt(self, data, key):

print()

print('BEFORE encrypt:', data)

encrypted = des().encrypt(str(key), str(data), padding=True)

print('AFTER encrypt:', encrypted)

print()

return encrypted

def decrypt(self, data, key):

print()

print('BEFORE decrypt:', data)

decrypted = des().decrypt(str(key), data, padding=True)

decrypted = make\_tuple(decrypted)

print('AFTER decrypt:', decrypted)

print()

return decrypted

class KeyCreator:

def create\_key():

return random.randint(100000000, 999999999)

class KDC:

available\_clients = ['client1', 'client2']

clients\_keys = [KeyCreator.create\_key(), KeyCreator.create\_key()]

available\_servers = ['server1', 'server2']

servers\_keys = [KeyCreator.create\_key(), KeyCreator.create\_key()]

def \_\_init\_\_(self):

self.des = DesEncrpter()

self.tgs\_id = 1

self.key\_tgs = KeyCreator.create\_key()

def get\_permission\_ticket(self, client\_id):

print('New permission ticket call')

if client\_id in self.available\_clients:

t = current\_milli\_time()

p = hours\_to\_milli(48)

key\_tgs\_c = KeyCreator.create\_key()

ticket = self.build\_permission\_ticket(client\_id, self.tgs\_id, t, p, key\_tgs\_c)

print('New permission ticket:', ticket)

encrypted\_ticket = self.des.encrypt(ticket, self.key\_tgs)

bundle = (encrypted\_ticket, key\_tgs\_c)

index = self.available\_clients.index(client\_id)

client\_key = self.clients\_keys[index]

encrypted\_bundle = self.des.encrypt(bundle, client\_key)

return encrypted\_bundle

print('Unknown client id')

def get\_server\_ticket(self, permission\_ticket, authority, server\_id):

print('New server ticket call')

permission\_ticket = self.des.decrypt(permission\_ticket, self.key\_tgs)

client\_id = permission\_ticket[0]

t = permission\_ticket[2]

p = permission\_ticket[3]

key\_tgs\_c = permission\_ticket[4]

print('Permission ticket data. Client id: {}, timestamp: {}, period: {}, key TGS-Client: {}'.format(client\_id, t, p, key\_tgs\_c))

authority = self.des.decrypt(authority, key\_tgs\_c)

auth\_client\_id = authority[0]

auth\_t = authority[1]

print('Authority data. Client id: {}, timestamp: {}'.format(auth\_client\_id, auth\_t))

if client\_id != auth\_client\_id:

print('Invalid client')

return None

if auth\_t < t or auth\_t > t + p:

pritn('Ticket is expired')

return None

t = current\_milli\_time()

p = hours\_to\_milli(48)

key\_ss\_c = KeyCreator.create\_key()

server\_ticket = self.build\_server\_ticket(client\_id, server\_id, t, p, key\_ss\_c)

print('New server ticket:', server\_ticket)

index = self.available\_servers.index(server\_id)

server\_key = self.servers\_keys[index]

encrypted\_server\_ticket = self.des.encrypt(server\_ticket, server\_key)

bundle = (encrypted\_server\_ticket, key\_ss\_c)

encrypted\_bundle = self.des.encrypt(bundle, key\_tgs\_c)

return encrypted\_bundle

def build\_permission\_ticket(cls, client\_id, tgs, t, p, key\_tgs\_c):

return (client\_id, tgs, t, p, key\_tgs\_c)

def build\_server\_ticket(cls, client\_id, server\_id, t, p, key\_ss\_c):

return (client\_id, server\_id, t, p, key\_ss\_c)

class Client:

def \_\_init\_\_(self, client\_id, client\_key, kdc, servers):

self.client\_id = client\_id

self.client\_key = client\_key

self.kdc = kdc

self.servers = servers

self.des = DesEncrpter()

self.permission\_ticket = None

self.key\_tgs\_c = None

def make\_server\_call(self, server\_number):

print()

print()

print('Call server', server\_number)

server = self.servers[server\_number]

if self.permission\_ticket is None or self.key\_tgs\_c is None:

print('Trying to take permission ticket')

permission\_ticket\_bundle = self.kdc.get\_permission\_ticket(self.client\_id)

if permission\_ticket\_bundle is None:

return

permission\_ticket\_bundle = self.des.decrypt(permission\_ticket\_bundle, self.client\_key)

permission\_ticket = permission\_ticket\_bundle[0]

key\_tgs\_c = permission\_ticket\_bundle[1]

print('Key TGS-Client:', key\_tgs\_c)

self.permission\_ticket = permission\_ticket

self.key\_tgs\_c = key\_tgs\_c

else:

print('Permission ticket and key TGS-Client already defined')

permission\_ticket = self.permission\_ticket

key\_tgs\_c = self.key\_tgs\_c

print()

print('Trying to get server ticket')

bundle = self.\_\_call\_tgs(permission\_ticket, key\_tgs\_c, server.server\_id)

if bundle is None:

return

bundle = self.des.decrypt(bundle, key\_tgs\_c)

server\_ticket = bundle[0]

key\_ss\_c = bundle[1]

print('Key Server-Client:', key\_ss\_c)

print()

print('Trying to connect server')

t = current\_milli\_time()

authority = (self.client\_id, t)

authority\_enctypted = self.des.encrypt(authority, key\_ss\_c)

confirm\_t = server.connect(server\_ticket, authority\_enctypted)

if confirm\_t is None:

return

confirm\_t = self.des.decrypt(confirm\_t, key\_ss\_c)

if confirm\_t != t + 1:

print('Server returns wrong confirmation timestamp')

return

print()

print('Server call is successful')

def \_\_call\_tgs(self, permission\_ticket, key\_tgs\_c, server\_id):

t = current\_milli\_time()

print('Call TGS. Server id: {}, timestamp: {}'.format(server\_id, t))

authority = (self.client\_id, t)

authority\_enctypted = self.des.encrypt(authority, key\_tgs\_c)

bundle = self.kdc.get\_server\_ticket(permission\_ticket, authority\_enctypted, server\_id)

return bundle

class Server:

def \_\_init\_\_(self, server\_id, server\_key):

self.server\_id = server\_id

self.server\_key = server\_key

self.des = DesEncrpter()

def connect(self, server\_ticket, authority):

print('New server connection')

server\_ticket = self.des.decrypt(server\_ticket, self.server\_key)

client\_id = server\_ticket[0]

server\_id = server\_ticket[1]

t = server\_ticket[2]

p = server\_ticket[3]

key\_ss\_c = server\_ticket[4]

print('Server ticket data. Client id: {}, timestamp: {}, period: {}, key Server-Client: {}'.format(client\_id, t, p, key\_ss\_c))

if server\_id != self.server\_id:

print('Wrong server')

return None

authority = self.des.decrypt(authority, key\_ss\_c)

auth\_client\_id = authority[0]

auth\_t = authority[1]

print('Authority data. Client id: {}, timestamp: {}'.format(auth\_client\_id, auth\_t))

if client\_id != auth\_client\_id:

print('Invalid client')

return None

if auth\_t < t or auth\_t > t + p:

pritn('Ticket is expired')

return None

confirm\_t = auth\_t + 1

print('Confirmation timestamp is', confirm\_t)

encrypted\_confirm\_t = self.des.encrypt(confirm\_t, key\_ss\_c)

return encrypted\_confirm\_t

def init():

kdc = KDC()

server1 = Server(kdc.available\_servers[0], kdc.servers\_keys[0])

server2 = Server(kdc.available\_servers[1], kdc.servers\_keys[1])

client = Client(kdc.available\_clients[0], kdc.clients\_keys[0], kdc, [server1, server2])

print('Server 1 id: {}, server 1 key: {}'.format(server1.server\_id, server1.server\_key))

print('Server 2 id: {}, server 2 key: {}'.format(server2.server\_id, server2.server\_key))

print('client id: {}, client key: {}'.format(client.client\_id, client.client\_key))

return client

client = init()

client.make\_server\_call(0)

client.make\_server\_call(1)