Министерство образования Республики Беларусь Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ

И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Отчёт по лабораторной работе №2

По теме “Идентификация и аутентификация пользователей. Протокол Kerberos.”

Выполнила: студентка гр. 053501 Шурко Т.А.

Проверил: ассистент кафедры информатики Протько М. И.

Минск 2023

Содержание

[1. Введение 3](file:///D:\downloads\Lab1.docx#_Toc7030)

[2. Теоретические сведения 3](file:///D:\downloads\Lab1.docx#_Toc7031)

[3. Блок-схема алгоритма 6](file:///D:\downloads\Lab1.docx#_Toc7032)

[4. Тестирование программного продукта 8](file:///D:\downloads\Lab1.docx#_Toc7033)

[5. Вывод 10](file:///D:\downloads\Lab1.docx#_Toc7035)

Примечание. Код программ 11

1. Введение

Целью данной лабораторной работы является изучить теоретические сведения, создать 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);
* Данные, передаваемые по сети каждой из сторон;
* Проверки, выполняемые каждым из участников.

Процесс взаимодействия между сторонами протокола может быть реализован при помощи буферных переменных. Также необходимо выделить каждый из этапов протоколов для того, чтобы его можно было отделить от остальных.

# 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. 2.1):



Рисунок 2.1 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:

* + KC - основной ключ C ;
  + KC\_TGS - ключ, выдаваемый C для доступа к серверу выдачи разрешений *TGS* ;
  + {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.

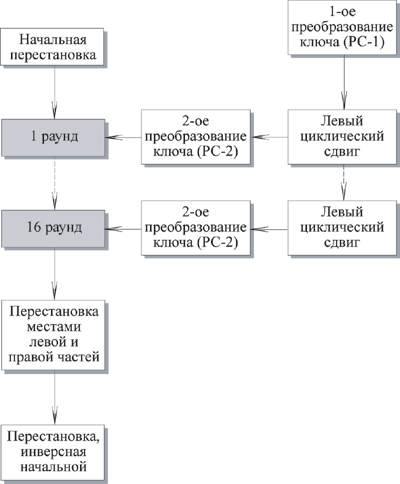


Рисунок 2.2 Общая схема DES

**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.

**Последовательность преобразований отдельного раунда**

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

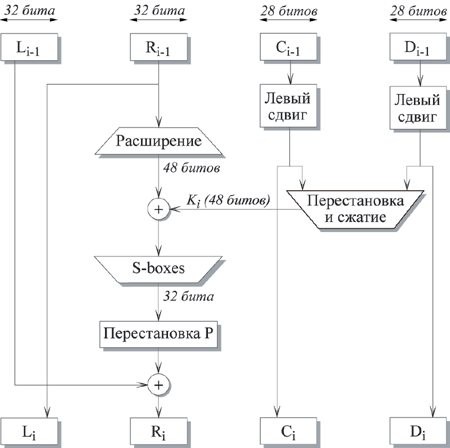


Рисунок 2.3 - I-ый раунд DES

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

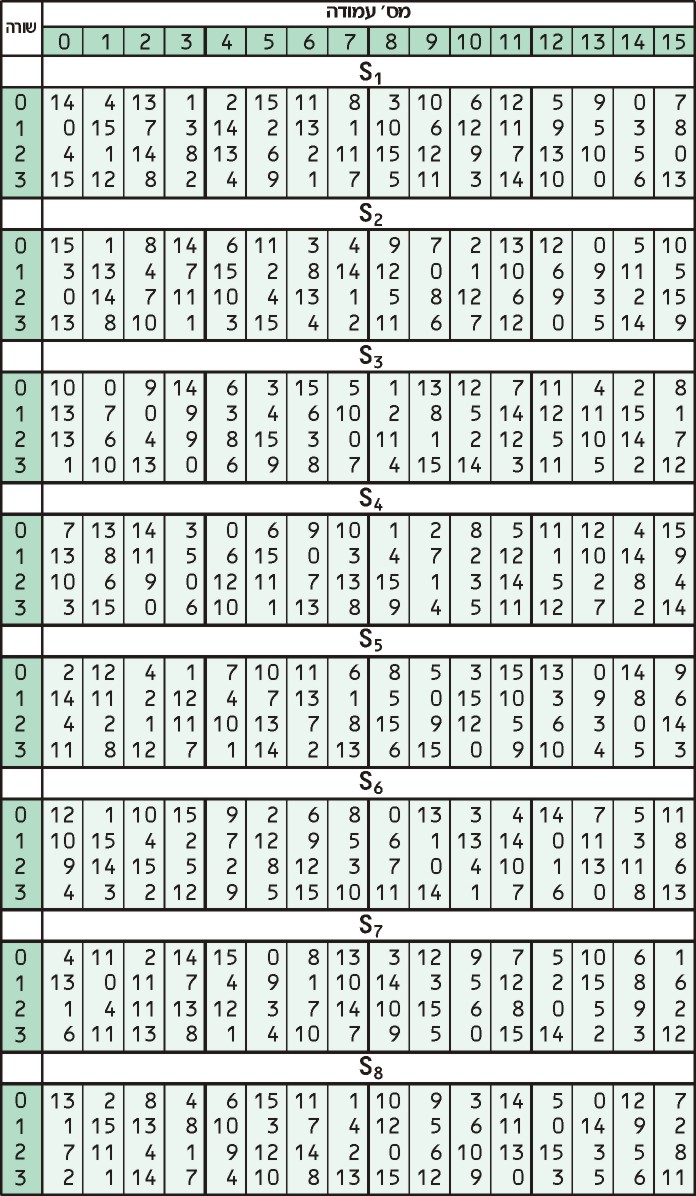
. . . efgh ijkl mnop . . .

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 2.4 - 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*.

**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, вначале выполняется перестановка в соответствии с таблицей Permuted Choice 1 (РС-1). 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 *раунда*. 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

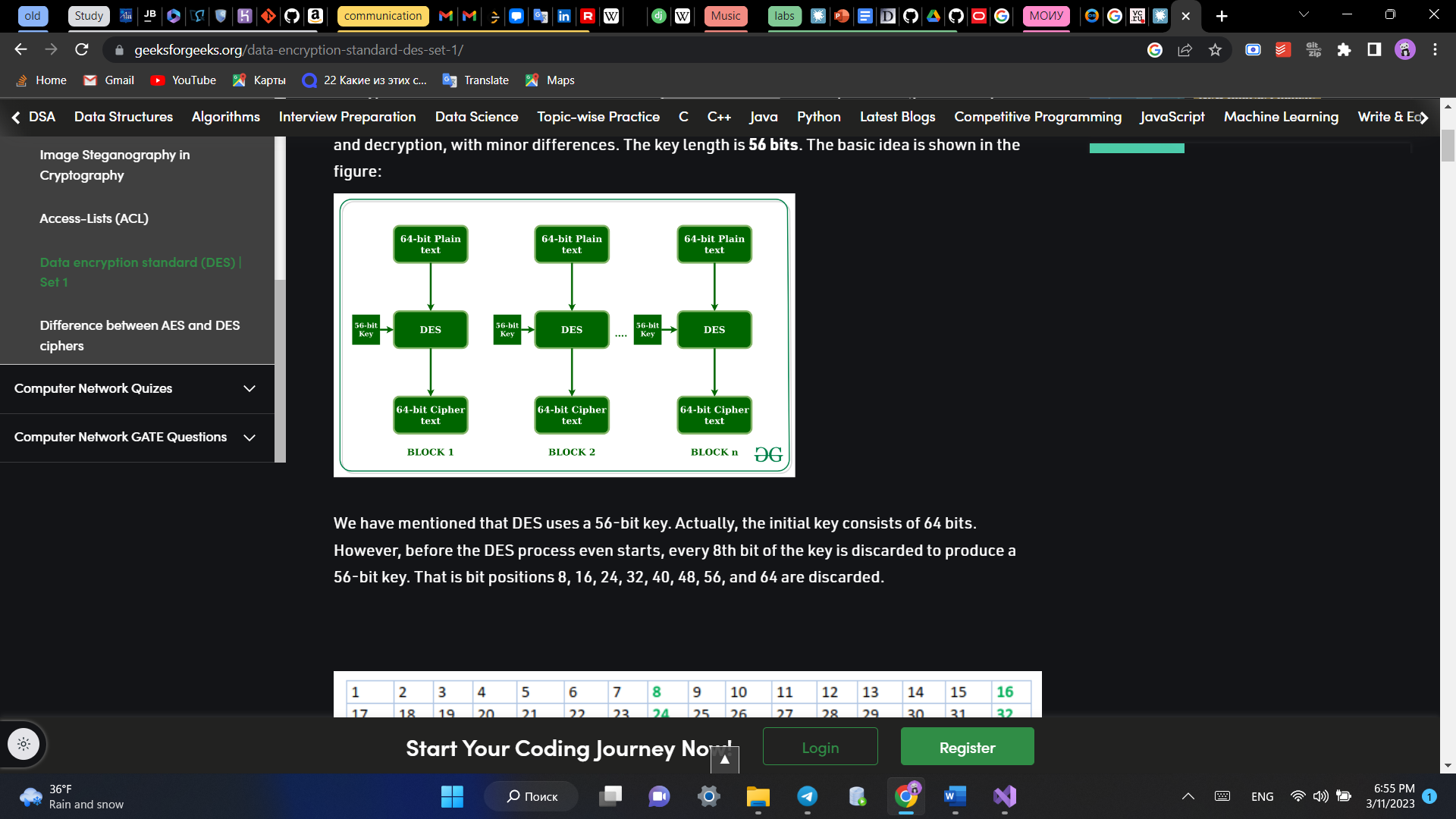


Рис. 3.1 – Основная идея DES

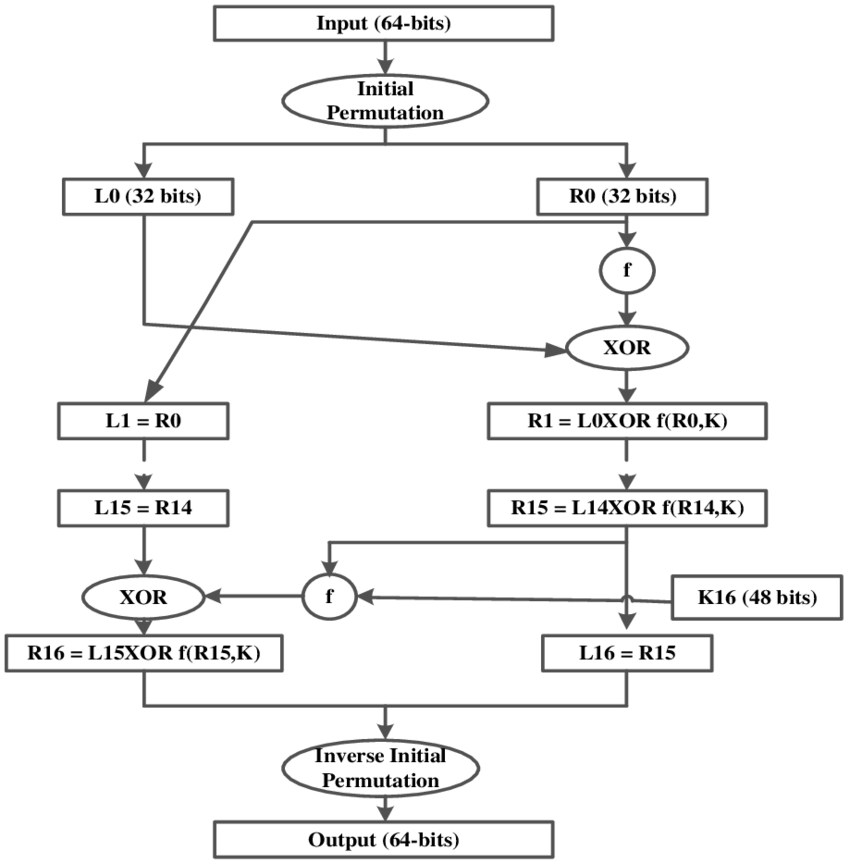
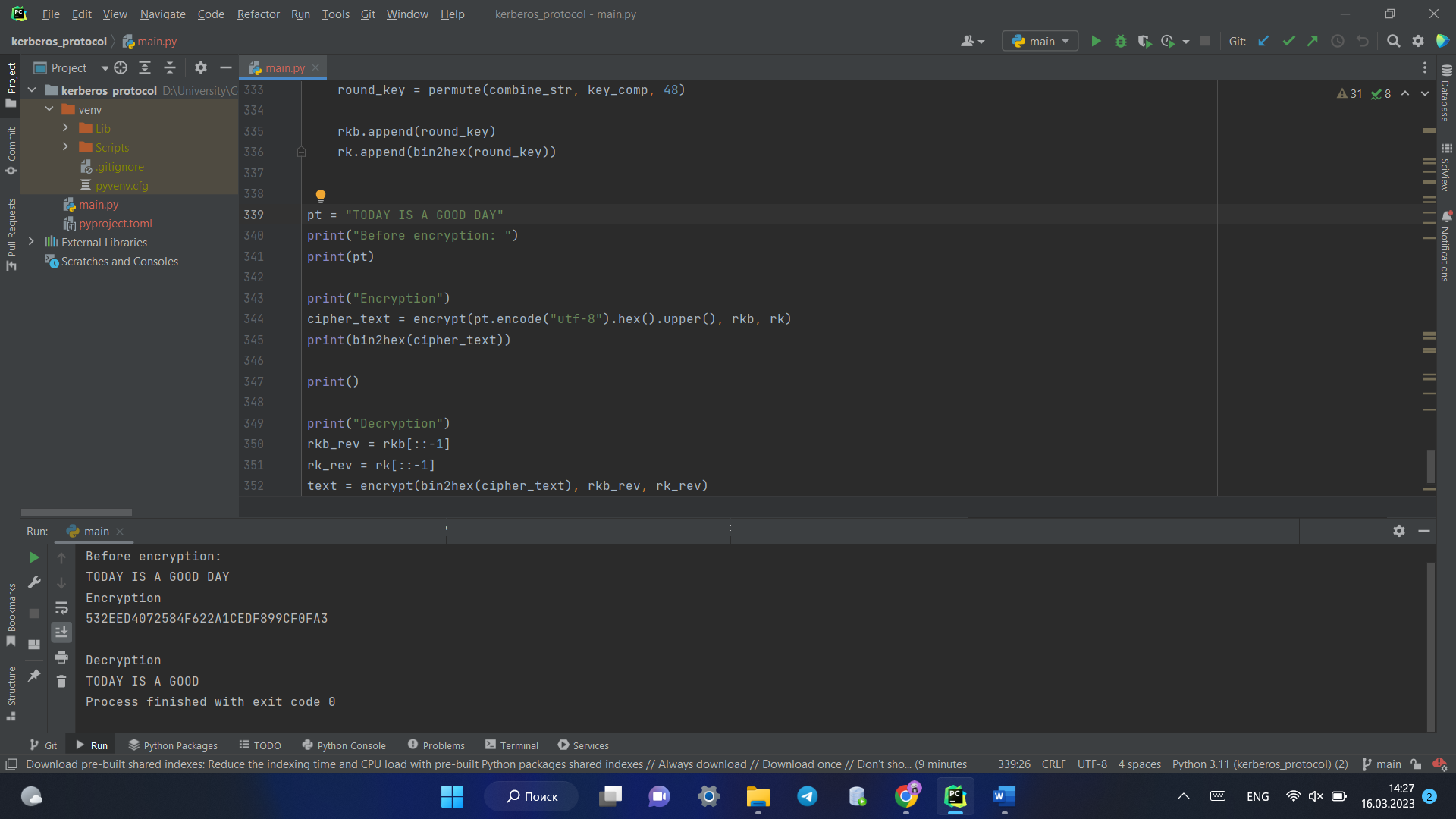


Рис. 3.2 ­– Блок-схема алгоритма DES

# Тестирование программного продукта

# 

# Рис. 4.1 – Результат выполнения программы



# Рис. 4.2 – Результат выполнения программы

# Вывод

В результате выполнения лабораторной работы №2 были изучены теоретические сведения, создано приложение, реализующее протокол распределения ключей Kerberos, включая процедуру, реализующую Алгоритм DES.

Приложение. Текст программ

# Python3 code for the above approach  
  
# Hexadecimal to binary conversion  
  
  
def hex2bin(s):  
 mp = {'0': "0000",  
 '1': "0001",  
 '2': "0010",  
 '3': "0011",  
 '4': "0100",  
 '5': "0101",  
 '6': "0110",  
 '7': "0111",  
 '8': "1000",  
 '9': "1001",  
 'A': "1010",  
 'B': "1011",  
 'C': "1100",  
 'D': "1101",  
 'E': "1110",  
 'F': "1111"}  
 bin = ""  
 for i in range(len(s)):  
 bin = bin + mp[s[i]]  
  
 return bin  
  
  
# Binary to hexadecimal conversion  
  
  
def bin2hex(s):  
 mp = {"0000": '0',  
 "0001": '1',  
 "0010": '2',  
 "0011": '3',  
 "0100": '4',  
 "0101": '5',  
 "0110": '6',  
 "0111": '7',  
 "1000": '8',  
 "1001": '9',  
 "1010": 'A',  
 "1011": 'B',  
 "1100": 'C',  
 "1101": 'D',  
 "1110": 'E',  
 "1111": 'F'}  
  
 hex = ""  
 start = True  
  
 for i in range(0, len(s), 4):  
 ch = s[i:i+4]  
  
 if start and ch == "0000":  
 continue  
 else:  
 start = False  
  
 hex = hex + mp[ch]  
  
 return hex  
  
  
# Binary to decimal conversion  
  
  
def bin2dec(binary):  
 binary1 = binary  
 decimal, i, n = 0, 0, 0  
 while (binary != 0):  
 dec = binary % 10  
 decimal = decimal + dec \* pow(2, i)  
 binary = binary // 10  
 i += 1  
 return decimal  
  
  
# Decimal to binary conversion  
  
  
def dec2bin(num):  
 res = bin(num).replace("0b", "")  
 if (len(res) % 4 != 0):  
 div = len(res) / 4  
 div = int(div)  
 counter = (4 \* (div + 1)) - len(res)  
 for i in range(0, counter):  
 res = '0' + res  
 return res  
  
  
# Permute function to rearrange the bits  
  
  
def permute(k, arr, n):  
 permutation = ""  
 for i in range(0, n):  
 permutation = permutation + k[arr[i] - 1]  
 return permutation  
  
  
# shifting the bits towards left by nth shifts  
  
  
def shift\_left(k, nth\_shifts):  
 s = ""  
 for i in range(nth\_shifts):  
 for j in range(1, len(k)):  
 s = s + k[j]  
 s = s + k[0]  
 k = s  
 s = ""  
 return k  
  
  
# calculating xow of two strings of binary number a and b  
  
  
def xor(a, b):  
 ans = ""  
 for i in range(len(a)):  
 if a[i] == b[i]:  
 ans = ans + "0"  
 else:  
 ans = ans + "1"  
 return ans  
  
  
# Table of Position of 64 bits at initial level: Initial Permutation Table  
initial\_perm = [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]  
  
# Expansion D-box Table  
exp\_d = [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]  
  
# Straight Permutation Table  
per = [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]  
  
# S-box Table  
sbox = [[[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]]]  
  
# Final Permutation Table  
final\_perm = [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]  
  
  
def encrypt(pt, rkb, rk):  
 num\_to\_add = (len(pt) % 64) % 4  
 pt = ("0" \* num\_to\_add + pt)  
  
 pt = hex2bin(pt)  
 cipher\_text = ""  
  
 num = len(pt) // 64  
  
 init = 0  
 while num > 0:  
 # Initial Permutation  
 p1 = permute(pt[init:init+64], initial\_perm, 64)  
  
 # Splitting  
 left = p1[0:32]  
 right = p1[32:64]  
  
 for i in range(0, 16):  
 # Expansion D-box: Expanding the 32 bits data into 48 bits  
 right\_expanded = permute(right, exp\_d, 48)  
  
 # XOR RoundKey[i] and right\_expanded  
 xor\_x = xor(right\_expanded, rkb[i])  
  
 # S-boxex: substituting the value from s-box table by calculating row and column  
 sbox\_str = ""  
 for j in range(0, 8):  
 row = bin2dec(int(xor\_x[j \* 6] + xor\_x[j \* 6 + 5]))  
 col = bin2dec(  
 int(xor\_x[j \* 6 + 1] + xor\_x[j \* 6 + 2] + xor\_x[j \* 6 + 3] + xor\_x[j \* 6 + 4]))  
 val = sbox[j][row][col]  
 sbox\_str = sbox\_str + dec2bin(val)  
  
 # Straight D-box: After substituting rearranging the bits  
 sbox\_str = permute(sbox\_str, per, 32)  
  
 # XOR left and sbox\_str  
 result = xor(left, sbox\_str)  
 left = result  
  
 # Swapper  
 if (i != 15):  
 left, right = right, left  
  
 # Combination  
 combine = left + right  
  
 cipher\_text += str(permute(combine, final\_perm, 64))  
  
 num -= 1  
 init += 64  
  
 return cipher\_text  
  
  
def display\_text(text):  
 if len(text) % 64 == 0:  
 num = len(text) // 64  
 else:  
 num = len(text) // 64 + 1  
  
 index = 0  
  
 while num > 0:  
 print(bytes.fromhex(bin2hex(text[index:index + 64])).decode('utf-8', errors='ignore'), end='')  
 index += 64  
 num -= 1  
  
 return  
  
key = "AABB09182736CCDDAD"  
  
# Key generation  
# --hex to binary  
key = hex2bin(key)  
  
# --parity bit drop table  
keyp = [57, 49, 41, 33, 25, 17, 9,  
 1, 58, 50, 42, 34, 26, 18,  
 10, 2, 59, 51, 43, 35, 27,  
 19, 11, 3, 60, 52, 44, 36,  
 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]  
  
# getting 56 bit key from 64 bit using the parity bits  
key = permute(key, keyp, 56)  
  
# Number of bit shifts  
shift\_table = [1, 1, 2, 2,  
 2, 2, 2, 2,  
 1, 2, 2, 2,  
 2, 2, 2, 1]  
  
# Key- Compression Table : Compression of key from 56 bits to 48 bits  
key\_comp = [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]  
  
# Splitting  
left = key[0:28] # rkb for RoundKeys in binary  
right = key[28:56] # rk for RoundKeys in hexadecimal  
  
rkb = []  
rk = []  
for i in range(0, 16):  
 # Shifting the bits by nth shifts by checking from shift table  
 left = shift\_left(left, shift\_table[i])  
 right = shift\_left(right, shift\_table[i])  
  
 # Combination of left and right string  
 combine\_str = left + right  
  
 # Compression of key from 56 to 48 bits  
 round\_key = permute(combine\_str, key\_comp, 48)  
  
 rkb.append(round\_key)  
 rk.append(bin2hex(round\_key))  
  
  
pt = "HELP ME! PLEASE"  
print(pt)  
  
print("Encryption")  
cipher\_text = encrypt(pt.encode("utf-8").hex().upper(), rkb, rk)  
display\_text(cipher\_text)  
  
print()  
  
print("Decryption")  
rkb\_rev = rkb[::-1]  
rk\_rev = rk[::-1]  
text = encrypt(bin2hex(cipher\_text), rkb\_rev, rk\_rev)  
display\_text(text)