Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет   
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина «Информационные сети. Основы безопасности»

|  |  |
| --- | --- |
|  |  |

**ОТЧЕТ**

к лабораторной работе № 2

на тему «Идентификация и аутентификация пользователей. Протокол Kerberos»

Выполнил            И. А. Тиханёнок

Проверил              Е. А. Лещенко

Минск 2024

**СОДЕРЖАНИЕ**

[1 Постановка задачи 3](#_Toc157722973)

[2 Краткие теоретические сведения 4](#_Toc157722974)

[3 Результаты выполнения лабораторной работы 13](#_Toc157722975)

[Выводы 15](#_Toc157722976)

[Список использованных источников 16](#_Toc157722977)

[Приложение А (обязательное) Листинг исходного кода 17](#_Toc157722978)

# 1 ПОСТАНОВКА ЗАДАЧИ

Целью выполнения данной лабораторной работы является создание приложения, реализующего протокол распределения ключей *Kerberos*, включая процедуру, реализующую алгоритм *DES*. В интерфейсе приложения должны быть наглядно представлены исходные данные, данные, передаваемые по сети каждой из сторон, проверки, выполняемые каждый из участников.

# 2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

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

Протокол *Kerberos* достаточно гибкий и имеет возможности тонкой настройки под конкретные применения, а также существует в нескольких версиях. В ходе данной лабораторной работы будет рассмотрен упрощенный механизм аутентификации, реализованные с помощью протокола *Kerberos* версии 5. Схема протокола *Kerberos* представлена на рисунке 2.1.



Рисунок 2.1 – Схема протокола Kerberos

При использовании протокола *Kerberos* нельзя напрямую получить доступ к какому-либо целевому серверу. Чтобы запустить 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 определенному рабочему процессу. К этапам этого процесса относятся:

1 Шаг первый: клиент *С* посылает серверу аутентификации *AS*   
свой идентификатор *с*.

2 Шаг второй: сервер аутентификации *AS*, проверив, что клиент *С* имеется в его базе, возвращает ему билет для доступа к серверу выдачи разрешений и ключ для взаимодействия с сервером выдачи разрешений *TGS*. Вся посылка зашифрована на ключе клиента *С*. Таким образом,   
даже если на первом шаге взаимодействия идентификатор *с* послал не клиент *С*, а нарушитель *Х*, то полученную от *AS* посылку *Х* расшифровать не сможет. Получить доступ к содержимому билета *TGT* не может не только нарушитель, но и клиент *С*, так как билет зашифрован на ключе, который распределили между собой сервер аутентификации и сервер выдачи разрешений.

3 Шаг третий: клиент *С* обращается к серверу выдачи разрешений *TGS*. Он пересылает полученный от *AS* билет, зашифрованный на ключе *Kas\_tgs*,   
и аутентификационный блок, содержащий идентификатор *с* 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 *С*   
и сервером *TGS*. С помощью этого ключа расшифровывается аутентификационный блок. Если метка в блоке совпадает с меткой в билете, это доказывается, 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. Если проверка проходит, и действующая   
в системе политика позволяется клиенту *С* обращаться к клиенту *SS*,   
тогда выполняется следующий шаг.

4 Шаг 4: сервер выдачи разрешений *TGS* посылает клиенту *С* ключ шифрования и билет, необходимые для доступа к серверу *SS*. Структура билета такая же, как на шаге номер 2.

5 Шаг пятый: клиент *С* посылает билет, полученный от сервера выдачи разрешений, и свой аутентификационный блок *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.

6 Шаг шестой: 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.   
Именно поэтому *SS* берет отметку времени из аутентификационного блока *С*, изменяет ее, увеличивая на единицу, шифрует на ключе *Kc\_ss* и возвращает *С*.

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

В алгоритме *Kerberos* могут применяться различные алгоритмы блочного симметричного шифрования. Для целей лабораторной работы был использован алгоритм *DES*.

Одной из наиболее известных криптографических систем с закрытым ключом является *DES* – *Data* *Encryption* *Standard*.

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. На рисунке 2.2 представлена общая схема *DES*.

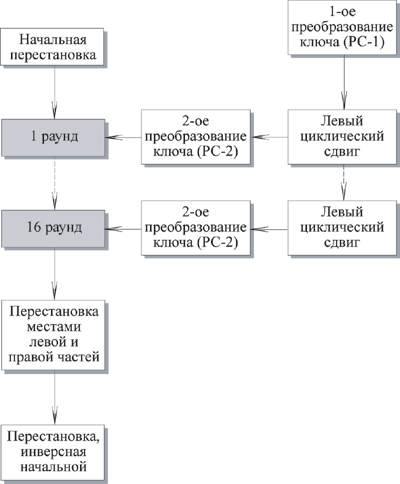


Рисунок 2.2 – Общая схема DES

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. На рисунке 2.3 представлена таблица для начальной перестановки.

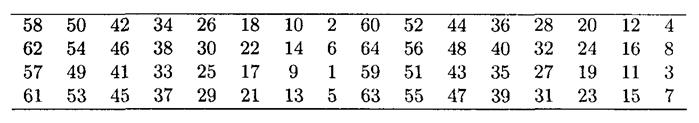


Рисунок 2.3 – Начальная перестановка

На рисунке 2.4 представлена таблица заключительной перестановки.

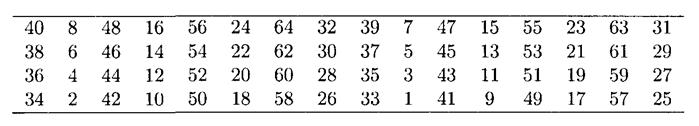


Рисунок 2.4 – Заключительная перестановка

Теперь будет рассмотрена последовательность преобразований, используемую в каждом раунде. На рисунке 2.5 представлен I-ый раунд протокола DES.

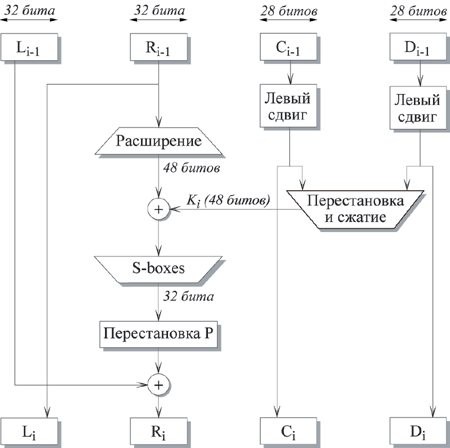


Рисунок 2.5 – *I*-ый раунд *DES*

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:

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*. *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. На рисунке 2.6 представлена   
таблица *S-boxes*.

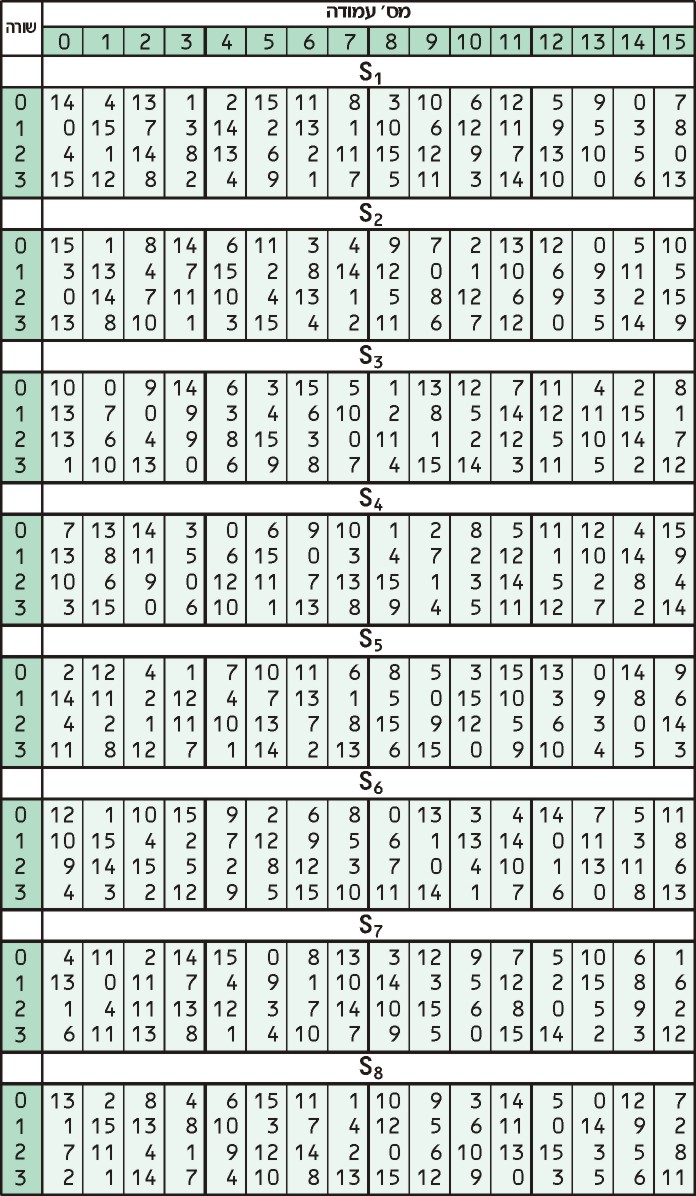


Рисунок 2.6 – *S-boxes*

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*. На рисунке 2.7 представлена таблица *Р*.

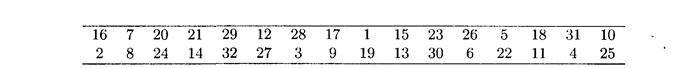


Рисунок 2.7 – Перестановка с помощью *Р*-блоков

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 убираются биты 64, 56, 48, 40, 32, 16, 8), 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 цикла. Таблица сдвигов ключа в зависимости от номера раунда представлена на рисунке 2.9.

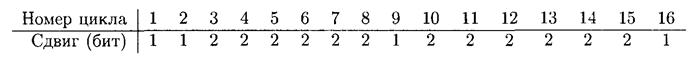


Рисунок 2.9 – Сдвиг ключа в зависимости от номера раунда

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*).

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.

# 3 РЕЗУЛЬТАТЫ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ             РАБОТЫ

В ходе выполнения лабораторной было создано приложение, реализующее протокол распределения ключей *Kerberos*, включая процедуру, реализующую алгоритм *DES*. В интерфейсе приложения наглядно представлены исходные данные протокола, данные, передаваемые по сети каждой из сторон. Результат работы программы представлен на рисунке 3.1.

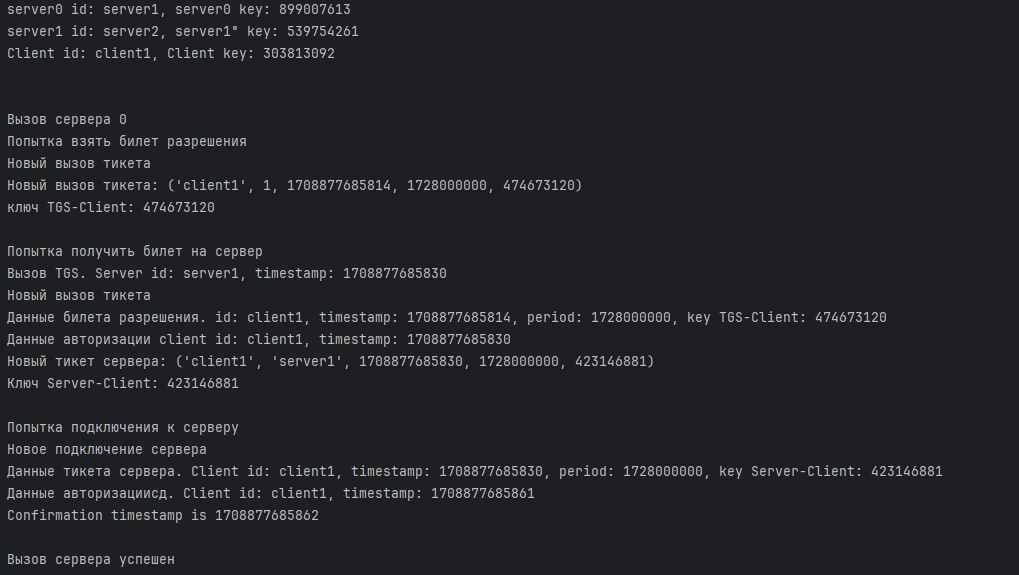


Рисунок 3.1 – Результат работы программы

Таким образом, в ходе данной лабораторной работы было создано приложение, реализующее протокол с использованием алгоритма *DES*.

# ВЫВОДЫ

В ходе лабораторной работы было создано приложение, реализующее протокол распределения ключей Kerberos, включая алгоритм DES.   
В интерфейсе приложения наглядно представлены исходные данные, данные, передаваемые по сети каждой из сторон, и проверки, выполняемые каждым   
из участников.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Что такое Kerberos? [Электронный ресурс]. – Режим доступа: https://www.keepersecurity.com/ru\_RU/resources/glossary/what-is-kerberos/. – Дата доступа: 08.02.2024.

[2] Идентификация и аутентификация. Протокол *Kerberos* [Электронный ресурс]. – Режим доступа: https://intuit.ru/studies/courses/531/387/lecture/8998/. – Дата доступа: 08.02.2024.

[3] Стандарт шифрования *DES* [Электронный ресурс]. – Режим доступа:   
https://kaf401.rloc.ru/Criptfiles/DES.htm/. – Дата доступа: 08.02.2024.

## ПРИЛОЖЕНИЕ А

## (обязательное)

## Листинг исходного кода

Листинг 1 – Программный реализация des.py

# Initial permut matrix for the datas  
PI = [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]  
# Initial permut made on the key  
CP\_1 = [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]  
# Permut applied on shifted key to get Ki+1  
CP\_2 = [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]  
# Expand matrix to get a 48bits matrix of datas to apply the xor with Ki  
E = [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]  
# SBOX  
S\_BOX = [  
 [[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],  
 ]  
]  
# Permut made after each SBox substitution for each round  
P = [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]  
# Final permut for datas after the 16 rounds  
PI\_1 = [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]  
# Matrix that determine the shift for each round of keys  
SHIFT = [1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1]  
def string\_to\_bit\_array(text): # Convert a string into a list of bits  
 array = list()  
 for char in text:  
 binval = binvalue(char, 8) # Get the char value on one byte  
 array.extend([int(x) for x in list(binval)]) # Add the bits to the final list  
 return array  
def bit\_array\_to\_string(array): # Recreate the string from the bit array  
 res = ''.join([chr(int(y, 2)) for y in [''.join([str(x) for x in \_bytes]) for \_bytes in nsplit(array, 8)]])  
 return res  
def binvalue(val, bitsize): # Return the binary value as a string of the given size  
 binval = bin(val)[2:] if isinstance(val, int) else bin(ord(val))[2:]  
 if len(binval) > bitsize:  
 raise "binary value larger than the expected size"  
 while len(binval) < bitsize:  
 binval = "0" + binval # Add as many 0 as needed to get the wanted size  
 return binval  
def nsplit(s, n): # Split a list into sublists of size "n"  
 return [s[k:k + n] for k in range(0, len(s), n)]  
ENCRYPT = 1  
DECRYPT = 0  
class Des():  
 def \_\_init\_\_(self):  
 self.password = None  
 self.text = None  
 self.keys = list()  
 def run(self, key, text, action=ENCRYPT, padding=False):  
 if len(key) < 8:  
 raise "Key Should be 8 bytes long"  
 elif len(key) > 8:  
 key = key[:8] # If key size is above 8bytes, cut to be 8bytes long  
 self.password = key  
 self.text = text  
 if padding and action == ENCRYPT:  
 self.addPadding()  
 elif len(self.text) % 8 != 0: # If not padding specified data size must be multiple of 8 bytes  
 raise "Data size should be multiple of 8"  
 self.generatekeys() # Generate all the keys  
 text\_blocks = nsplit(self.text, 8) # Split the text in blocks of 8 bytes so 64 bits  
 result = list()  
 for block in text\_blocks: # Loop over all the blocks of data  
 block = string\_to\_bit\_array(block) # Convert the block in bit array  
 block = self.permut(block, PI) # Apply the initial permutation  
 g, d = nsplit(block, 32) # g(LEFT), d(RIGHT)  
 tmp = None  
 for i in range(16): # Do the 16 rounds  
 d\_e = self.expand(d, E) # Expand d to match Ki size (48bits)  
 if action == ENCRYPT:  
 tmp = self.xor(self.keys[i], d\_e) # If encrypt use Ki  
 else:  
 tmp = self.xor(self.keys[15 - i], d\_e) # If decrypt start by the last key  
 tmp = self.substitute(tmp) # Method that will apply the SBOXes  
 tmp = self.permut(tmp, P)  
 tmp = self.xor(g, tmp)  
 g = d  
 d = tmp  
 result += self.permut(d + g, PI\_1) # Do the last permut and append the result to result  
 final\_res = bit\_array\_to\_string(result)  
 if padding and action == DECRYPT:  
 return self.removePadding(final\_res) # Remove the padding if decrypt and padding is true  
 else:  
 return final\_res # Return the final string of data ciphered/deciphered  
 def substitute(self, d\_e): # Substitute bytes using SBOX  
 subblocks = nsplit(d\_e, 6) # Split bit array into sublist of 6 bits  
 result = list()  
 for i in range(len(subblocks)): # For all the sublists  
 block = subblocks[i]  
 row = int(str(block[0]) + str(block[5]), 2) # Get the row with the first and last bit  
 column = int(''.join([str(x) for x in block[1:][:-1]]), 2) # Column is the 2,3,4,5th bits  
 val = S\_BOX[i][row][column] # Take the value in the SBOX appropriated for the round (i)  
 bin = binvalue(val, 4) # Convert the value to binary  
 result += [int(x) for x in bin] # And append it to the resulting list  
 return result  
 def permut(self, block, table): # Permut the given block using the given table (so generic method)  
 return [block[x - 1] for x in table]  
 def expand(self, block, table): # Do the exact same thing than permut but for more clarity has been renamed  
 return [block[x - 1] for x in table]  
 def xor(self, t1, t2): # Apply a xor and return the resulting list  
 return [x ^ y for x, y in zip(t1, t2)]  
 def generatekeys(self): # Algorithm that generates all the keys  
 self.keys = []  
 key = string\_to\_bit\_array(self.password)  
 key = self.permut(key, CP\_1) # Apply the initial permut on the key  
 g, d = nsplit(key, 28) # Split it in to (g->LEFT),(d->RIGHT)  
 for i in range(16): # Apply the 16 rounds  
 g, d = self.shift(g, d, SHIFT[i]) # Apply the shift associated with the round (not always 1)  
 tmp = g + d # Merge them  
 self.keys.append(self.permut(tmp, CP\_2)) # Apply the permut to get the Ki  
 def shift(self, g, d, n): # Shift a list of the given value  
 return g[n:] + g[:n], d[n:] + d[:n]  
 def addPadding(self): # Add padding to the datas using PKCS5 spec.  
 pad\_len = 8 - (len(self.text) % 8)  
 self.text += pad\_len \* chr(pad\_len)  
 def removePadding(self, data): # Remove the padding of the plain text (it assume there is padding)  
 pad\_len = ord(data[-1])  
 return data[:-pad\_len]  
 def encrypt(self, key, text, padding=False):  
 return self.run(key, text, ENCRYPT, padding)  
 def decrypt(self, key, text, padding=False):  
 return self.run(key, text, DECRYPT, padding)  
if \_\_name\_\_ == '\_\_main\_\_':  
 key = "secret\_k"  
 text = "Hello wo"  
 d = Des()  
 r = d.encrypt(key, text)  
 r2 = d.decrypt(key, r)  
 print(f"Text: {text}, key {key}")  
 print("Ciphered: %r" % r)  
 print("Deciphered: ", r2)

Листинг 2 – Программная реализация main.py

import tkinter as tk  
from tkinter import messagebox  
import random  
import time  
from des 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 DesEncrypter:  
 def \_\_init\_\_(self):  
 self.logs = ""  
 def encrypt(self, data, key):  
 encrypted = Des().encrypt(key=str(key), text=str(data), padding=True)  
 return encrypted  
 def decrypt(self, data, key):  
 decrypted = Des().decrypt(key=str(key), text=str(data), padding=True)  
 decrypted = make\_tuple(decrypted)  
 return decrypted  
 def log(self, message):  
 self.logs += message + "\n"  
class KeyCreator:  
 @staticmethod  
 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, logger):  
 self.des = DesEncrypter()  
 self.tgs\_id = 1  
 self.key\_tgs = KeyCreator.create\_key()  
 self.logger = logger  
 def get\_permission\_ticket(self, client\_id):  
 self.logger.log('New call ticket:')  
 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)  
 self.logger.log('New call ticket: {}'.format(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  
 self.logger.log('Unknown id client')  
 def get\_server\_ticket(self, permission\_ticket, authority, server\_id):  
 self.logger.log('New call ticket')  
 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]  
 self.logger.log(  
 'Data ----- '  
 '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]  
 self.logger.log('Data for avtomatization ---- '  
 'client id: {}, '  
 'timestamp: {}'.format(auth\_client\_id, auth\_t))  
 if client\_id != auth\_client\_id:  
 self.logger.log('Invalid client')  
 return None  
 if auth\_t < t or auth\_t > t + p:  
 self.logger.log('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)  
 self.logger.log('New server ticket: {}'.format(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  
 @staticmethod  
 def build\_permission\_ticket(client\_id, tgs, t, p, key\_tgs\_c):  
 return client\_id, tgs, t, p, key\_tgs\_c  
 @staticmethod  
 def build\_server\_ticket(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, logger):  
 self.client\_id = client\_id  
 self.client\_key = client\_key  
 self.kdc = kdc  
 self.servers = servers  
 self.des = DesEncrypter()  
 self.permission\_ticket = None  
 self.key\_tgs\_c = None  
 self.logger = logger  
 def make\_server\_call(self, server\_number):  
 self.logger.log('')  
 self.logger.log('')  
 self.logger.log('Call server {}'.format(server\_number))  
 server = self.servers[server\_number]  
 if self.permission\_ticket is None or self.key\_tgs\_c is None:  
 self.logger.log('Attempting to take a 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]  
 self.logger.log('Key TGS-Client: {}'.format(key\_tgs\_c))  
 self.permission\_ticket = permission\_ticket  
 self.key\_tgs\_c = key\_tgs\_c  
 else:  
 self.logger.log('The permission ticket and TGS-Client key are already defined')  
 permission\_ticket = self.permission\_ticket  
 key\_tgs\_c = self.key\_tgs\_c  
 self.logger.log('')  
 self.logger.log('Trying to get a ticket to the server')  
 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]  
 self.logger.log('Key Server-Client: {}'.format(key\_ss\_c))  
 self.logger.log('')  
 self.logger.log('Attempting to connect to the 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:  
 self.logger.log('Server returns incorrect timestamp')  
 return  
 self.logger.log('')  
 self.logger.log('Server call successful')  
 def \_\_call\_tgs(self, permission\_ticket, key\_tgs\_c, server\_id):  
 t = current\_milli\_time()  
 self.logger.log('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, logger):  
 self.server\_id = server\_id  
 self.server\_key = server\_key  
 self.des = DesEncrypter()  
 self.logger = logger  
 def connect(self, server\_ticket, authority):  
 self.logger.log('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]  
 self.logger.log('Data ticket server ---- '  
 'Client id: {}, '  
 'timestamp: {}, '  
 'period: {},'  
 'key Server-Client: {}'.format(client\_id, t, p, key\_ss\_c))  
 if server\_id != self.server\_id:  
 self.logger.log('Unknown server')  
 return None  
 authority = self.des.decrypt(authority, key\_ss\_c)  
 auth\_client\_id = authority[0]  
 auth\_t = authority[1]  
 self.logger.log('Authorization data ---- '  
 'Client id: {}, '  
 'timestamp: {}'.format(auth\_client\_id, auth\_t))  
 if client\_id != auth\_client\_id:  
 self.logger.log('Invalid client')  
 return None  
 if auth\_t < t or auth\_t > t + p:  
 self.logger.log('Ticket is expired')  
 return None  
 confirm\_t = auth\_t + 1  
 self.logger.log('Confirmation timestamp is {}'.format(confirm\_t))  
 encrypted\_confirm\_t = self.des.encrypt(confirm\_t, key\_ss\_c)  
 return encrypted\_confirm\_t  
class Logger:  
 def \_\_init\_\_(self):  
 self.logs = ""  
 def log(self, message):  
 self.logs += message + "\n"  
def initialization\_client(logger):  
 kdc = KDC(logger)  
 server1 = Server(kdc.available\_servers[0], kdc.servers\_keys[0], logger)  
 server2 = Server(kdc.available\_servers[1], kdc.servers\_keys[1], logger)  
 client = Client(kdc.available\_clients[0], kdc.clients\_keys[0], kdc, [server1, server2], logger)  
 logger.log('server0 id: {}, server0 key: {}'.format(server1.server\_id, server1.server\_key))  
 logger.log('server1 id: {}, server1" key: {}'.format(server2.server\_id, server2.server\_key))  
 logger.log('Client id: {}, Client key: {}'.format(client.client\_id, client.client\_key))  
 return client, kdc  
class Application(tk.Tk):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.title("Client Server Communication")  
 self.geometry("400x300")  
 self.logger = Logger()  
 self.client, self.kdc = initialization\_client(self.logger)  
 self.create\_widgets()  
 self.create\_logger\_window()  
 def create\_widgets(self):  
 self.server1\_button = tk.Button(self, text="Call Server 1", command=self.call\_server\_1)  
 self.server1\_button.pack(pady=(50, 10), padx=10, fill="x")  
 self.server2\_button = tk.Button(self, text="Call Server 2", command=self.call\_server\_2)  
 self.server2\_button.pack(pady=(100, 10), padx=10, fill="x")  
 def create\_logger\_window(self):  
 self.logger\_window = tk.Toplevel(self)  
 self.logger\_window.title("Logs")  
 self.logger\_text = tk.Text(self.logger\_window, wrap="word")  
 self.logger\_text.pack(expand=True, fill="both")  
 self.logger\_text.insert(tk.END, self.logger.logs)  
 self.logger\_text.config(state=tk.DISABLED)  
 def update\_logger\_window(self):  
 self.logger\_text.config(state=tk.NORMAL)  
 self.logger\_text.delete("1.0", tk.END)  
 self.logger\_text.insert(tk.END, self.logger.logs)  
 self.logger\_text.config(state=tk.DISABLED)  
 def call\_server\_1(self):  
 self.client.make\_server\_call(0)  
 self.update\_logger\_window()  
 def call\_server\_2(self):  
 self.client.make\_server\_call(1)  
 self.update\_logger\_window()  
if \_\_name\_\_ == "\_\_main\_\_":  
 app = Application()  
 app.mainloop()