Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Лабораторная работа № 2

**Идентификация и аутентификация пользователей. Протокол Kerberos**

Выполнил

студент гр. 953505

Кривецкий Р.А.

Проверил

Протько М.И.

Минск, 2023

**Введение**

Целью данной лабораторной было реализовать программные средства протокола распределения ключей Kerberos вместе с процедурой, которая реализует Алгоритм DES.

**Задание**

1) Изучить теоретические сведения.

2) 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 и т.п.); − 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. 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:

* + 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.

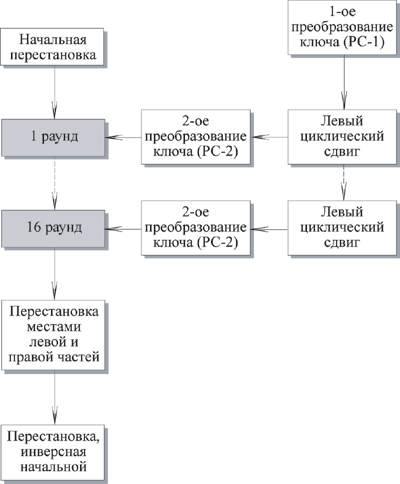


Рисунок 2 Общая схема DES

**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*.

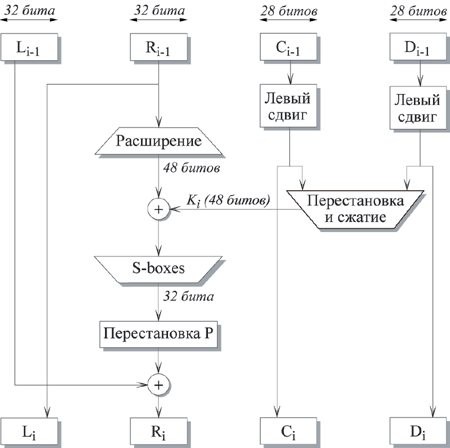


Рисунок 3 - I-ый раунд DES

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.

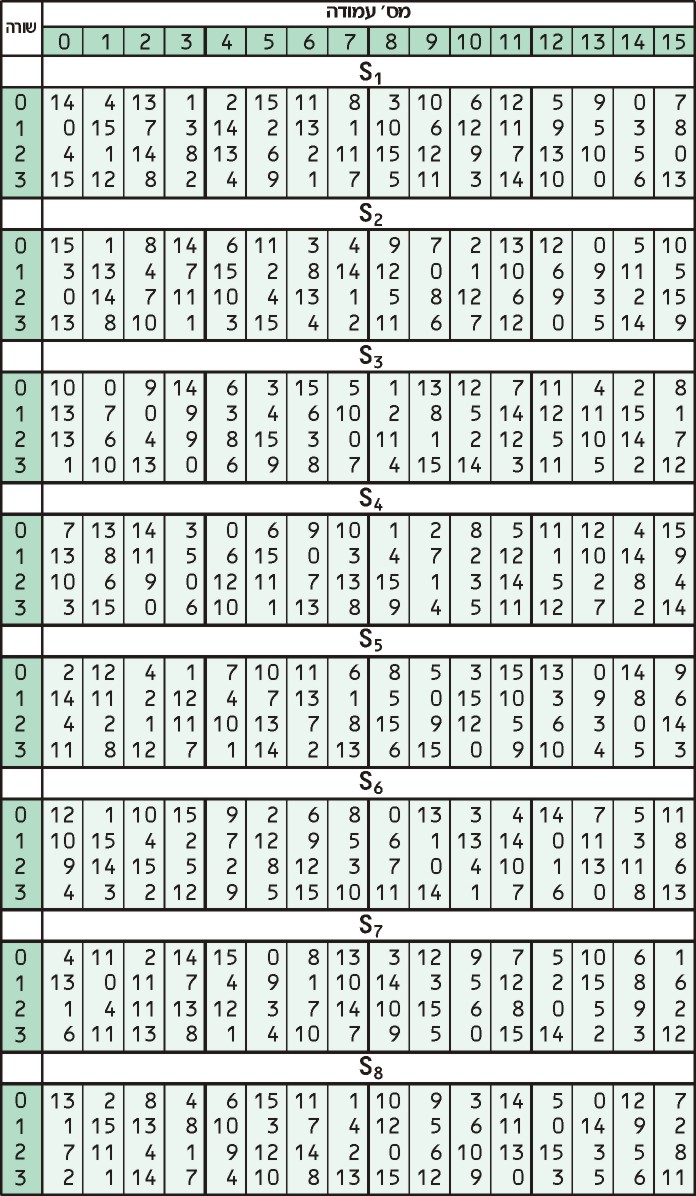


Рисунок 4 - S-boxes

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*.

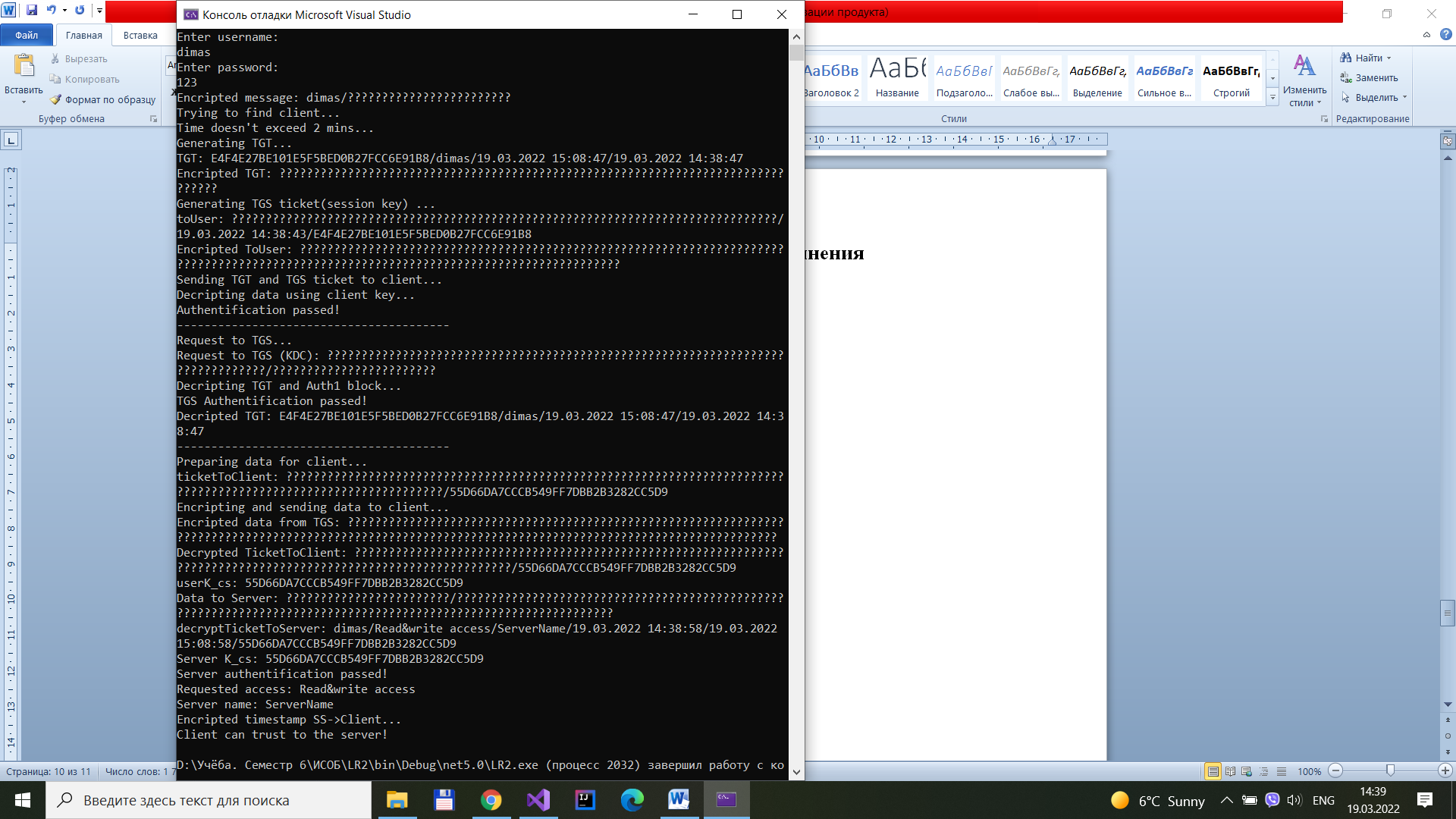
**Создание подключей**

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*.

**Результат выполнения**



**Выводы**

DES был национальным стандартом США в 1977—1980 гг., но в настоящее время DES используется (с ключом длины 56 бит) только для устаревших систем, чаще всего используют его более криптоустойчивый вид (3DES, DESX). 3DES является простой эффективной заменой DES, и сейчас он рассмотрен как стандарт. В ближайшее время DES и Triple DES будут заменены алгоритмом AES (Advanced Encryption Standard — Расширенный Стандарт Шифрования). Kerberos является одним из самых распространенных протоколов аунтефикации. В настоящее время множество ОС поддерживают данный протокол, в число которых входят: Windows 2000 и более поздние версии, которые используют Kerberos как метод аутентификации в домене между участниками, различные UNIX и UNIX подобные ОС (Apple Mac OS X, Red Hat Enterprise Linux 4, FreeBSD, Solaris, AIX, OpenVMS).

**Код программы**

using System;

using System.Collections.Generic;

using System.Security.Cryptography;

using System.Text;

using System.Threading;

namespace LR2

{

public class Program

{

private static readonly string KDCMasterKey = GetHash("masterKey");

private static readonly string serverMasterKey = GetHash("serverMasterKey1");

private static readonly string keyK\_cs = GetHash("keyK\_css");

private static string clientSessionKey;

private static string KDCsessionKey;

private static DateTime ClientTimeStamp { get; set; }

private static readonly Dictionary<string, string> UserList = new()

{

["roman"] = "123",

};

static void Main(string[] args)

{

Console.WriteLine("Enter username:");

string userName = Console.ReadLine();

Console.WriteLine("Enter password:");

string password = Console.ReadLine();

Client clientUser = new(userName, password);

/\*

\* 1)

\* Компьютер пользователя обращается к службе KDC и передает ей

\* имя пользователя, а также текущее время на рабочей станции

\* пользователя, при этом имя пользователя передается в открытом виде,

\* текущее время на рабочей станции пользователя передается в

\* зашифрованном виде и является аутентификатором.

\*/

ClientTimeStamp = DateTime.Now;

var value = new StringBuilder();

value.Append(userName + "/")

.Append(DES.Encrypt(ClientTimeStamp.ToString(), clientUser.PasswordHash));

string message = value.ToString();

Console.WriteLine($"Encripted message: { message }");

/\*

\* 2)

\* Служба KDC ищет пользователя в AS,

\* выявляет мастер ключ пользователя,

\* который основан на пароле пользователя и расшифровывает аутентификатор,

\* т. е. получает время отправки запроса. Разница во времени отправки запроса и

\* текущего времени на контроллере домена не должно превышать определенного значения,

\* установленного политикой протокола Kerberos

\*/

// Authentification server

Console.WriteLine($"Trying to find client...");

Thread.Sleep(2000);

DateTime timeStamp;

Client KDCclient;

var userData = message.Split('/');

if (UserList.TryGetValue(userData[0], out string userPass))

{

KDCclient = new Client(userName: userData[0], password: userPass);

if (DateTime.TryParse(DES.Decipher(userData[1], KDCclient.PasswordHash), out timeStamp))

{

if (timeStamp.AddMinutes(2) > DateTime.Now)

{

Console.WriteLine("Time doesn't exceed 2 mins...");

}

else

{

Console.WriteLine("Time exceeded 2 mins...");

Console.ReadLine();

return;

}

}

else

{

Console.WriteLine("Your password is wrong...");

Console.ReadLine();

return;

}

}

else

{

Console.WriteLine("Client not found...");

Console.ReadLine();

return;

}

/\*

\* 3)

\* Затем KDC создает два объекта:

\* a. ключ сессии, посредством которого будет обеспечиваться зашифрование данных при обмене между клиентом и службой KDC,

\* b. билет на получение билета "Ticket-Granting Ticket" (TGT).

\* TGT включает: вторую копию ключа сессии, имя пользователя,

\* время окончания жизни билета. Билет на получение билета шифруется с использованием собственного мастер ключа службы KDC,

\* который известен только KDC, т. е. TGT может быть расшифрован только самой службой KDC.

\*/

// KDC -> Client (encripted TGT with KDC master key)

Console.WriteLine($"Generating TGT...");

Thread.Sleep(2000);

KDCsessionKey = GetHash(new Random().Next(1000000, 9999999).ToString());

var TGT = new StringBuilder();

TGT.Append(KDCsessionKey + "/")

.Append(KDCclient.UserName + "/")

.Append(DateTime.Now.AddMinutes(30).ToString() + "/")

.Append(DateTime.Now);

string encriptedTGT = DES.Encrypt(TGT.ToString(), KDCMasterKey);

Console.WriteLine($"TGT: {TGT}");

Console.WriteLine($"Encripted TGT: {encriptedTGT}");

/\* 4)

\* Служба KDC зашифровывает аутентификатор пользователя (time stamp)

\* и ключ сессии с помощью ключа клиента.

После этого эти данные отправляются клиенту. \*/

// KDC -> Client (encripted client auth(time)+TGT ticket + session key (encr on pass) with KDC client key)

Console.WriteLine($"Generating TGS ticket(session key) ...");

Thread.Sleep(2000);

var toUser = new StringBuilder();

toUser.Append(encriptedTGT + "/")

.Append(timeStamp + "/")

.Append(KDCsessionKey);

string encryptedToUser = DES.Encrypt(toUser.ToString(), KDCclient.PasswordHash);

Console.WriteLine($"toUser: {toUser}");

Console.WriteLine($"Encripted ToUser: {encryptedToUser}");

/\* 5)

\* Компьютер клиента получает информацию от службы KDC,

\* проверяет аутентификатор,

\* расшифровывает ключ сессии. \*/

Console.WriteLine($"Sending TGT and TGS ticket to client...");

Thread.Sleep(2000);

Console.WriteLine($"Decripting data using client key...");

Thread.Sleep(1000);

string decryptToUser = DES.Decipher(encryptedToUser, clientUser.PasswordHash);

var TGTtimeStampAndSessionKey = decryptToUser.Split('/');

timeStamp = DateTime.Parse(TGTtimeStampAndSessionKey[1]);

if (timeStamp.AddMinutes(2) > DateTime.Now)

{

clientSessionKey = TGTtimeStampAndSessionKey[2];

Console.WriteLine($"Authentification passed!\n{new string('-', 40)}");

}

else

{

Console.WriteLine("Authentification failed!");

Console.ReadLine();

return;

}

/\*

\* 6)

\* Теперь клиент обладает ключом сессии и TGT,

\* что предоставляет возможность безопасного

\* взаимодействия со службой KDC.

\*/

/\*

\* 1)

\* Клиент обращается к TGS.

\* Клиент представляет KDC свой TGT и маркер времени,

\* которые зашифрованы с помощью ключа сессии,

\* известного службе KDC.

\*/

// Client -> TGS

Console.WriteLine($"Request to TGS...");

Thread.Sleep(2000);

var toKDC = new StringBuilder();

toKDC.Append(TGTtimeStampAndSessionKey[0] + "/");

toKDC.Append(DES.Encrypt(DateTime.Now.ToString(), clientSessionKey));

Console.WriteLine($"Request to TGS (KDC): {toKDC}");

/\*

\* 2)

\* KDC расшифровывает TGT, используя свой KDC master key.

\* Маркер времени расшифровывается с помощью session key from TGT.

\* Теперь KDC может подтвердить,

\* что запрос пришел от «правильного» пользователя,

\* т.к. этот пользователь может использовать этот сессионный ключ.

\*/

// Ticket Granting Server

Console.WriteLine($"Decripting TGT and Auth1 block...");

Thread.Sleep(2000);

var toTGSData = toKDC.ToString().Split('/');

var decriptedTGT = DES.Decipher(toTGSData[0], KDCMasterKey);

var tgtData = decriptedTGT.Split('/');

timeStamp = DateTime.Parse(DES.Decipher(toTGSData[1], tgtData[0])); // tgtData[0] - session key from TGT

var tgtTimeStamp = DateTime.Parse(tgtData[3]); //tgtData[3] - time stamp from TGT

if (timeStamp.AddMinutes(2) > tgtTimeStamp) // Timestamp from auth block ~ equals TGT blocks` timestamp

{

Console.WriteLine("TGS Authentification passed!");

Console.WriteLine($"Decripted TGT: {decriptedTGT}\n{new string('-', 40)}");

}

else

{

Console.WriteLine("TGS Authentification failed!");

Console.ReadLine();

return;

}

// TGS -> Client

Console.WriteLine($"Preparing data for client...");

Thread.Sleep(2000);

var ticketToClient = new StringBuilder();

var tgsBlock = new StringBuilder();

timeStamp = DateTime.Now;

tgsBlock.Append(KDCclient.UserName + "/")

.Append("Read&write access" + "/")

.Append("ServerName" + "/")

.Append(timeStamp + "/")

.Append(timeStamp.AddMinutes(30) + "/")

.Append(keyK\_cs); // Key for interaction client with SS

var ticketToServer = DES.Encrypt(tgsBlock.ToString(), serverMasterKey); // Ktgs\_ss

ticketToClient.Append(ticketToServer);

ticketToClient.Append("/" + keyK\_cs);

Console.WriteLine("ticketToClient: " + ticketToClient);

/\*

\* 3)

\* Вся эта структура зашифровывается с помощью сессионного ключа,

\* который стал доступен пользователю при аутентификации.

\* После чего эта информация отправляется клиенту.

\*/

// TGS -> Client

Console.WriteLine($"Encripting and sending data to client...");

Thread.Sleep(2000);

var encryptTicketToClient = DES.Encrypt(ticketToClient.ToString(), KDCsessionKey);

Console.WriteLine("Encripted data from TGS: " + encryptTicketToClient);

/\*

\* 4)

\* Получив билет, клиент расшифровывает его с помощью сессионного ключа,

\* т. е. K\_cs становится доступным клиенту, K\_cs доступен также и серверу.

\* Клиент не может прочитать билет сервера, т. к. он зашифрован на ключе сервера.

\*/

var decryptedToClient = DES.Decipher(encryptTicketToClient, clientSessionKey);

Console.WriteLine($"Decrypted TicketToClient: {decryptedToClient}");

var clientK\_cs = decryptedToClient.Split('/')[1];

Console.WriteLine($"userK\_cs: {clientK\_cs}");

/\*

\* 5)

\* Клиент зашифровывает маркер времени с помощью ключа,

\* K\_cs затем отправляет маркер времени и билет сервера

\* самому серверу, к ресурсам которого пытается получить

\* доступ клиент.

\*/

var toServer = new StringBuilder();

ClientTimeStamp = DateTime.Now;

toServer.Append(DES.Encrypt(ClientTimeStamp.ToString(), clientK\_cs))

.Append("/" + decryptedToClient.Split('/')[0]); // Encripted TGS

Console.WriteLine($"Data to Server: {toServer}");

/\* 6)

\* Получив эту информацию, на первом этапе сервер расшифровывает

\* свой билет, используя свой долговременный ключ.

\* Это предоставляет возможность получить доступ к K\_cs ,

\* с помощью которого будет на втором этапе расшифрован маркер времени,

\* полученный от клиента.\*/

// SS checks that client can be trusted and gets

var toServerData = toServer.ToString().Split("/");

var decryptedTicketToServer = DES.Decipher(toServerData[1], serverMasterKey);

var ticketToServerData = decryptedTicketToServer.Split('/');

Console.WriteLine("decryptTicketToServer: " + decryptedTicketToServer);

var serverK\_cs = ticketToServerData[5];

var tgsTimeStamp = DateTime.Parse(ticketToServerData[3]);

Console.WriteLine("Server K\_cs: " + serverK\_cs);

timeStamp = DateTime.Parse(DES.Decipher(toServerData[0], serverK\_cs));

if (timeStamp.AddMinutes(2) > tgsTimeStamp)

{

Console.WriteLine("Server authentification passed!\n" +

$"Requested access: {ticketToServerData[1]}\n" +

$"Server name: {ticketToServerData[2]}");

}

else

{

Console.WriteLine("Server authentification failed!");

Console.ReadLine();

return;

}

// SS -> Client (Auth2.TimeStamp + 1)

var encriptedServerTimeStamp = DES.Encrypt(timeStamp.AddMinutes(1).ToString(), serverK\_cs);

Console.WriteLine("Encripted timestamp SS->Client...");

// Client checks that SS could be trusted

var decipheredServerTimeStamp = DES.Decipher(encriptedServerTimeStamp, clientK\_cs);

if (ClientTimeStamp.AddMinutes(1).ToString().Equals(decipheredServerTimeStamp))

{

Console.WriteLine("Client can trust to the server!");

}

else

{

Console.WriteLine("Client can't trust to the server!");

Console.ReadLine();

return;

}

}

private static string GetHash(string str)

{

var tmpSource = Encoding.ASCII.GetBytes(str);

var tmpHash = new MD5CryptoServiceProvider().ComputeHash(tmpSource);

var value = new StringBuilder(tmpHash.Length);

for (int i = 0; i < tmpHash.Length; i++)

{

value.Append(tmpHash[i].ToString("X2"));

}

return value.ToString();

}

}

}