Учреждение образования
«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ»

Кафедра информатики

Отчет по лабораторной работе №2

Идентификация и аутентификация пользователей. Протокол Kerberos

Выполнил:
Студент гр. 953501
Кременевский В.С.

Проверил:
Ассистент кафедры информатики
Протько М.И.

Минск 2022

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):

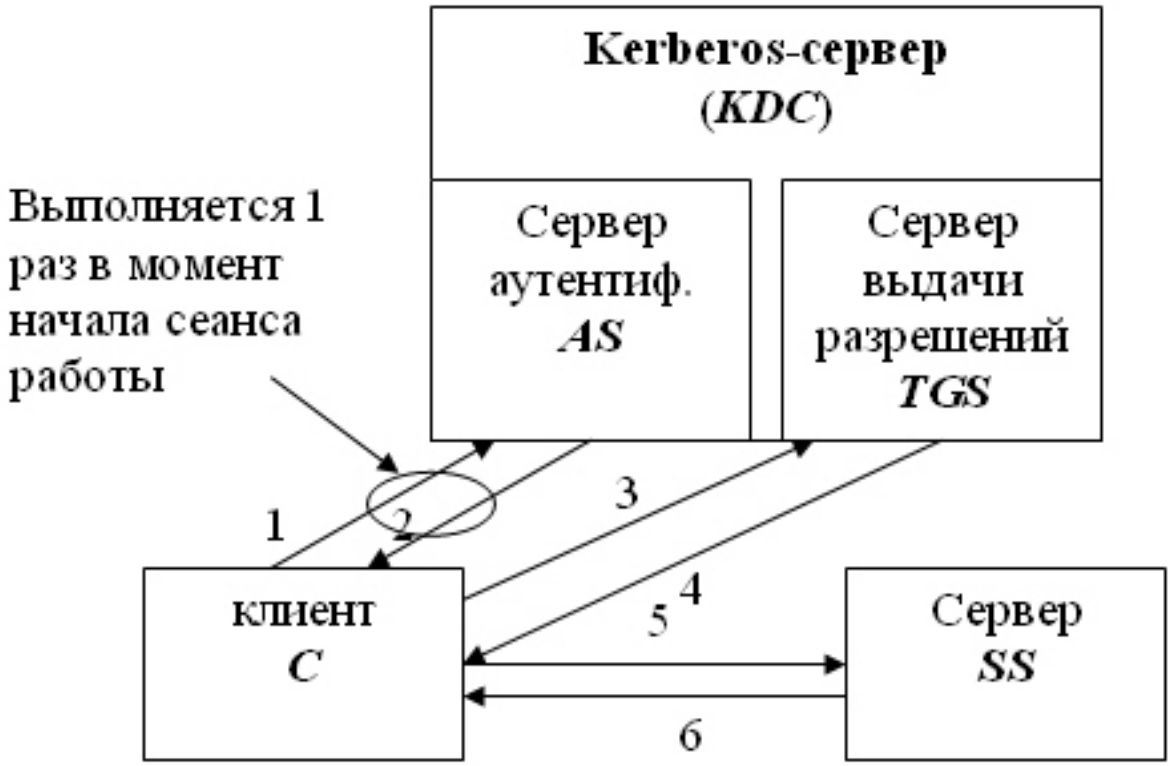

Рисунок 1 Схема протокола Kerberos

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).

2. AS->C: {{TGT}K_{AS_TGS}, K_{C_TGS}}K_C,

где:

- K_C - основной ключ C ;
- K_{C_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*,t_1,p_1, K_{C_TGS}}, где *tgs* - идентификатор сервера выдачи разрешений, t_1 - отметка времени, p_1 - *период действия* билета.

Запись $\{\cdot\}K_X$ здесь и далее означает, что содержимое фигурных скобок зашифровано на ключе K_X (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 c послал не клиент C, 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.

3. C->*TGS*: {TGT}K_{AS_TGS}, {Aut$_1$} K_{C_TGS}, {ID}

где {Aut$_1$} - аутентификационный блок - Aut$_1$ = {c,t_2}, t_2 - метка времени; ID - идентификатор запрашиваемого сервиса (в частности, это может быть идентификатор сервера SS).

Клиент C на этот раз обращается к серверу выдачи разрешений TGS. Он пересылает полученный от AS билет, зашифрованный на ключе K_{AS_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 C (ведь только он знал ключ K_{C_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 C обращаться к клиенту SS, тогда выполняется шаг **4**).

4. *TGS*->C: {{*TGS*}K_{TGS_SS},K_{C_SS}}K_{C_TGS},

где K_{C_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*} ={c,ss,t_3,p_2, K_{C_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.

5. C->SS: {*TGS*}K_{TGS_SS}, {Aut_2} K_{C_SS}

где Aut_2={c,t_4}.

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 K_{TGS_SS}. Имея этот ключ, он может расшифровать билет, получить ключ шифрования K_{C_SS} и проверить подлинность *отправителя сообщения*.

6. SS->C: {t_4+1}K_{C_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 K_{C_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* K_{C_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 C 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 M- 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*.

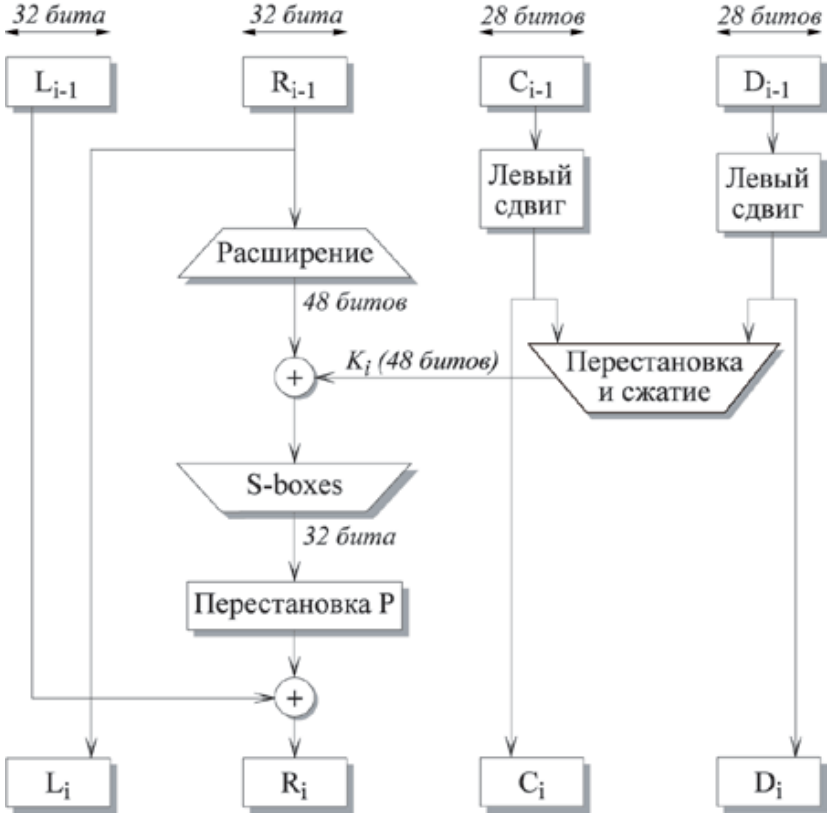

Рисунок 3 - I-ый раунд DES

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:

$L_i = R_{i-1}$

$R_i = L_{i-1} \oplus F(R_{i-1}, K_i)$

Где $\oplus$ обозначает операцию XOR.

Таким образом, выход левой половины L_i равен входу правой половины R_{i-1}. Выход правой половины R_i является результатом применения операции XOR к L_{i-1} и функции F, зависящей от R_{i-1} и K_i.

Рассмотрим функцию F более подробно.

R_i, которое подается на вход функции F, имеет длину 32 бита. Вначале R_i 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* K_i. 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.

שורה	מס' עמודה															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	S_1															
0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
1	0	15	7	3	14	2	13	1	10	6	12	11	9	5	3	8
2	4	1	14	8	13	6	2	11	15	12	9	7	13	10	5	0
3	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13
	S_2															
0	15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
1	3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
2	0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
3	13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9
	S_3															
0	10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
1	13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
2	13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
3	1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12
	S_4															
0	7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
1	13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
2	10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
3	3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14
	S_5															
0	2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
1	14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
2	4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
3	11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3
	S_6															
0	12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
1	10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
2	9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
3	4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13
	S_7															
0	4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
1	13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
2	1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
3	6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12
	S_8															
0	13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
1	1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
2	7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
3	2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

Рисунок 4 - S-boxes

Далее полученное 32-битное значение обрабатывается с помощью перестановки *P*, 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* K_i состоит из 48 бит. Ключи K_i получаются по следующему алгоритму. Для 56-битного ключа, используемого на входе алгоритма, вначале выполняется перестановка в соответствии с таблицей

Permuted Choice 1 (PC-1). Полученный 56-битный ключ разделяется на две 28-битные части, обозначаемые как C_0 и D_0 соответственно. На каждом *раунде* C_i и D_i 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 (PC-2), который создает 48-битное выходное значение, являющееся входом функции $F(R_{i-1}, K_i)$.

Дешифрование

Процесс дешифрования аналогичен процессу шифрования. На входе алгоритма используется зашифрованный текст, но ключи K_i используются в обратной последовательности. K_{16} используется на первом *раунде*, K_1 используется на последнем *раунде*.

Результат выполнения

```
(base) ➜  lr2 python3 kerberos_server.py

Connection address: ('127.0.0.1', 49480)

Sender: ('127.0.0.1', 49480)
Message: 49480

Send to: ('127.0.0.1', 49480)
Message: 14367625799357162146

Sender: ('127.0.0.1', 49480)
Message: 10189375766592573757

Send to: ('127.0.0.1', 49480)
Message: 87055639872476482 42

Stop connection: ('127.0.0.1', 49480)
```

```
(base) ➜  LR2 python3 tcp_server.py
Connection address: ('127.0.0.1', 49481)

Sender: ('127.0.0.1', 49481)
Key: 87055639872476482 42

Sender: ('127.0.0.1', 49481)
Key: 13956083690471814318
```

```
(base) ➜  LR2 python3 client.py
Connection with kerberos server
Get key from kerberos server: 14367625799357162146
Get key from kerberos server: 87055639872476482 42

Access!
```

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.Security.Cryptography;
using System.Text;
using System.Collections.Generic;

namespace ISOB_Kerberos
{
    public class Program
    {
        private static readonly string KDCMasterKey =
GetHash("masterKey");
        private static readonly string serverMasterKey =
GetHash("serverMasterKey");
        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()
        {
            ["anatoli"] = "12345",
        };

        static void Main(string[] args)
        {
            string domainName = "domainName";
            Console.WriteLine("Введите имя:");
            string userName = Console.ReadLine();
            Console.WriteLine("Введите пароль:");
            string password = Console.ReadLine();
            Client clientUser = new(userName, password);
```

```
            var value = new StringBuilder();
            value.Append(userName + "/");
            value.Append(domainName + "/");
            ClientTimeStamp = DateTime.Now;
            value.Append(DES.Encrypt(ClientTimeStamp.ToString(),
clientUser.PasswordHash));
            string message = value.ToString();

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


            Console.WriteLine($"Trying to find client...");
            Thread.Sleep(2000);
            DateTime timeStamp;
            Client KDKclient;

            var userData = message.Split('/');
            if (UserList.TryGetValue(userData[0], out string
userPass))
            {
                KDKclient = new Client(userName: userData[0],
password: userPass);
                timeStamp =
DateTime.Parse(DES.Decipher(userData[2],
KDKclient.PasswordHash));
                if (timeStamp.AddMinutes(2) > DateTime.Now)
                {
                    Console.WriteLine("timeStamp: " +
timeStamp);
                    Console.WriteLine("Time doesn't exceed 2
mins...");
                }
                else
                {
                    Console.WriteLine("Time exceeded 2
mins...");
                    Console.ReadLine();
                    return;
                }
            }
            else
            {
                Console.WriteLine("Client not found...");
                Console.ReadLine();
                return;
            }

            // 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(KDKclient.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}");


            // KDC -> Client (encripted client auth(time)+TGS
ticket(session key) 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(), KDKclient.PasswordHash);
            Console.WriteLine($"toUser: {toUser}");
            Console.WriteLine($"Encripted ToUser:
{encryptedToUser}");

            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;
            }
            // Client -> KDC
            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}");

            // 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(KDKclient.UserName + "/")
                    .Append("Read&write access" + "/")
                    .Append("ServerName" + "/")
                    .Append(timeStamp + "/")
                    .Append(timeStamp.AddMinutes(30) + "/")
                    .Append(keyK_cs);
            var ticketToServer =
DES.Encrypt(tgsBlock.ToString(), serverMasterKey); // Ktgs_ss
            ticketToClient.Append(ticketToServer);
            ticketToClient.Append("/" + keyK_cs);
            Console.WriteLine("ticketToClient: " +
ticketToClient);

            // 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);

            var decryptedToClient =
DES.Decipher(encryptTicketToClient, clientSessionKey);
            Console.WriteLine("Decrypted TicketToClient: " +
decryptedToClient);
            var clientK_cs = decryptedToClient.Split('/')[1];
            Console.WriteLine("userK_cs: " + clientK_cs);

            var toServer = new StringBuilder();
            ClientTimeStamp = DateTime.Now;

toServer.Append(DES.Encrypt(ClientTimeStamp.ToString(),
clientK_cs))
                    .Append("/" + decryptedToClient.Split('/')
[0]); // Encripted TGS

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

            // 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];
            Console.WriteLine("Server K_cs: " + serverK_cs);
```

```
            timeStamp =
DateTime.Parse(DES.Decipher(toServerData[0], serverK_cs));
            if (timeStamp.AddMinutes(2) > DateTime.Now)
            {
                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(decipheredServe
rTimeStamp))
            {
                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();
        }
    }
}
```