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:

студент гр. 153502

Бычко В.П.

Проверил:

Лещенко Е.А.

Минск 2024

СОДЕРЖАНИЕ

[Введенеие 3](#_Toc157436533)

[1 Краткие теоретические сведения 4](#_Toc157436534)

[2 Результат выполнения программы 7](#_Toc157436535)

[Приложение А 10](#_Toc157436537)

# ВВЕДЕНИЕ

Цель данной лабораторной работы заключается в изучении теоретических сведений по работе протокола Kerberos и алгоритма DES и разработки программы, реализующее протокол распределения ключей Kerberos, включая процедуру, реализующую Алгоритм DES.

# 1 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 имеющий возможности тонкой настройки под конкретные применения, существует в нескольких версиях. Мы рассмотрим упрощенный механизм аутентификации, реализованный с помощью протокола Kerberos версии 5.

В начале имеются три участника протокола Kerberos: клиент, сервис, центр распределения ключей.

Каждый из участников обладает своим долговременным секретом (ключом). Кроме того, центр распределения ключей обладает секретами всех участников.

Клиент отправляет серверу аутентификации запрос, содержащий: принципал клиента и срок жизни билета.

Сервер аутентификации по полученному принципалу находит в базе Kerberos секрет клиента. Кроме того, для дальнейшего общения с KDC сервер аутентификации случайным образом генерирует сессионный ключ. В итоге в ответ клиенту отправляются два сообщения. Первое сообщение зашифровано с использованием секрета клиента и содержит: сессионный ключ для KDC, метка времени, срок жизни TGT. Второе сообщение (TGT) зашифровано уже с использованием секрета KDC и включает в себя те же самые данные, что и первое сообщение, но вместе с принципалом клиента.

Клиент, приняв ответ, может расшифровать только первое сообщение. Таким образом он получает сессионный ключ для дальнейшего общения с KDC. TGT также сохраняется у клиента в зашифрованном виде.

Теперь, пройдя аутентификацию, клиент желает получить доступ к какому-то сервису. Для этого он отправляет серверу выдачи разрешений запрос, содержащий: принципал сервиса, аутентификатор, состоящий из принципала клиента и метки времени, зашифрованных с использованием извлеченного ранее сессионного ключа для общения с KDC, сохраненный TGT.

Приняв запрос, сервер выдачи разрешений прежде всего выполняет проверку полученных данных. Сначала с использованием секрета KDC сервер расшифровывает TGT и по метке времени со сроком действия убеждается, что TGT не протух.

Далее сервер извлекает сессионный ключ для KDC. Несмотря на то, что указанный ключ был создан в KDC, нужды хранить его в базе Kerberos нет. Действительно, TGT не может быть изменен кем-либо кроме KDC, поэтому полученным из него данным можно доверять.

В случае успешного завершения проверок сервер выдачи разрешений отправляет клиенту ответ, содержащий два сообщения. Первое сообщение зашифровано с использованием сессионного ключа для KDC и содержит: сессионный ключ для общения с сервисом, метка времени, срок жизни TGS билета, принципал сервиса. Второе сообщение (TGS билет) зашифровано с использованием секрета сервиса и включает в себя те же самые данные, что и первое сообщение, а также принципал клиента. Клиент, приняв ответ, может расшифровать только первое сообщение. Таким образом он получает сессионный ключ для дальнейшего общения с сервисом. TGS билет сохраняется у клиента в зашифрованном виде.

Клиент отправляет сервису запрос на получение доступа, содержащий: аутентификатор, состоящий из принципала клиента и метки времени, зашифрованных с использованием извлеченного ранее сессионного ключа для общения с сервисом, сохраненный TGS билет, флаг взаимной аутентификации. Приняв запрос, сервис прежде всего выполняет проверку полученных данных. Сначала с использованием своего секрета сервис расшифровывает TGS и по метке времени со сроком действия убеждается, что TGS не протух. Далее сервис извлекает сессионный ключ.

TGS билет не может быть изменен кем-либо кроме того, кто знает секрет сервиса, а это KDC и сам сервис. Сервис доверяет KDC, таким образом извлеченным из TGS билета данным сервис также может доверять.

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.

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.

Затем 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. 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.

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.

# 2 РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ ПРОГРАММЫ

В результате разработки программы было создано консольное приложение, реализующее протокол распределения ключей Kerberos, включая процедуру, реализующую Алгоритм DES.

На рисунке 2.1 представлена схема работы протокола Kerberos.



Рисунок 2.1 – Схема работы протокола Kerberos

На рисунке 2.2 представлена блок-схема алгоритма шифрования DES.

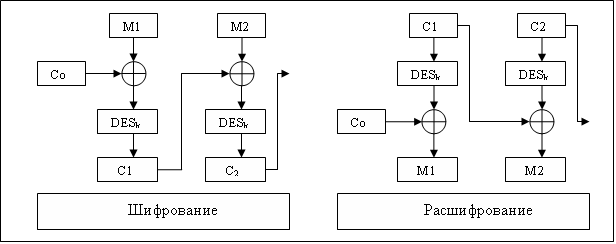


Рисунок 2.2 – Блок-схема алгоритма DES

Результат выполнения программы изображён на рисунке 2.3.

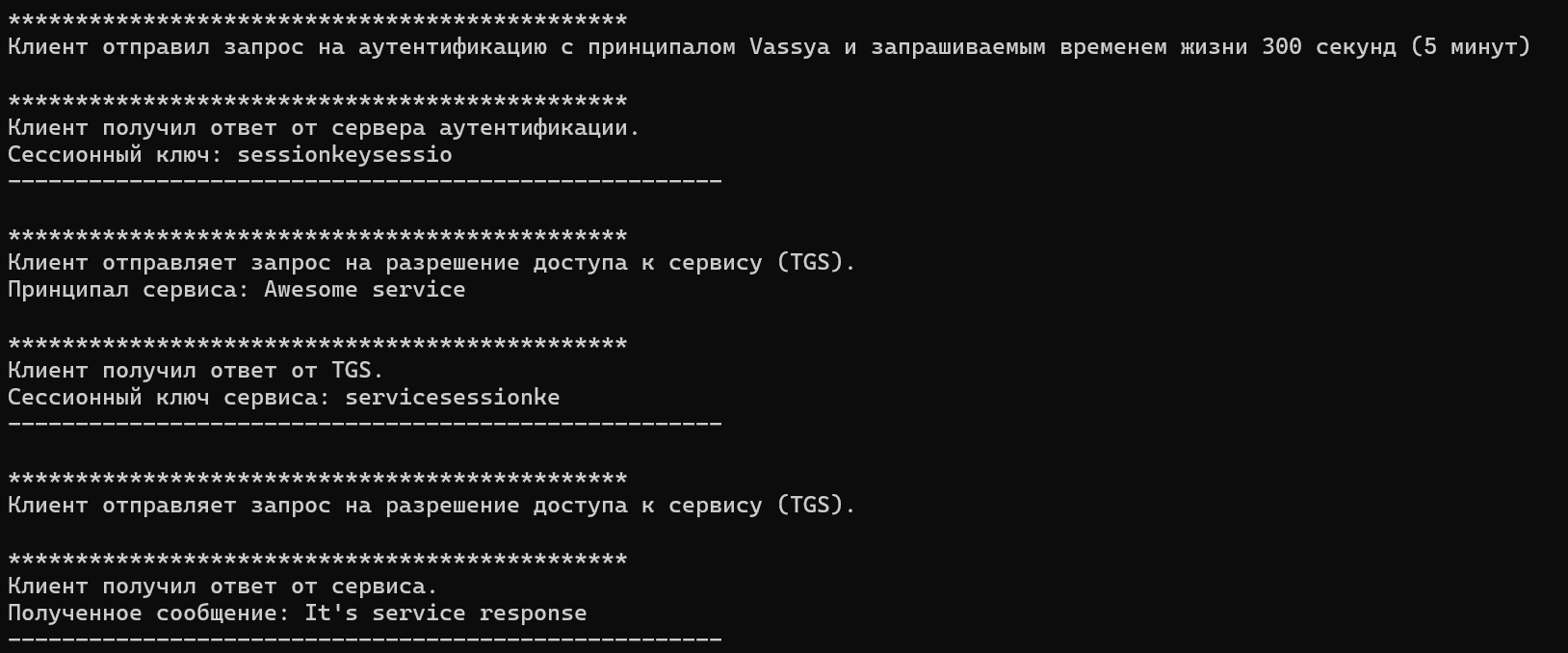


Рисунок 2.3 – Результат программы

# ПРИЛОЖЕНИЕ А

**(обязательное)**

**Исходный код программы**

**DES.cs**

internal class DES

{

#region Constants

private static readonly byte[] \_IP = { 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 };

private static readonly byte[] \_IP\_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 };

#endregion

public static byte[] Encrypt(byte[] data, byte[] key)

{

List<BitArray> keys = GetKeys(key);

List<byte> bigData = new(data);

// Добавляем пробелы до нормального размера

while (bigData.Count % 8 != 0)

{

bigData.Add((byte)' ');

}

List<BitArray> blocks = GetBlocks(bigData.ToArray());

IEnumerable<byte> result = Enumerable.Empty<byte>();

foreach (var block in blocks)

{

// Делаем первоначальную перестановку

BitArray initialPermutation = Permutate(block, \_IP);

BitArray left = new(32);

BitArray right = new(32);

for (int j = 0; j < 32; j++)

{

left[j] = initialPermutation[j];

right[j] = initialPermutation[32 + j];

}

for (int j = 0; j < 16; j++)

{

BitArray newRight = left.Xor(keys[j % keys.Count]);

left = right;

right = newRight;

}

BitArray combined = new(64);

for (int j = 0; j < 32; j++)

{

combined[j] = left[j];

combined[32 + j] = right[j];

}

result = result.Concat(combined.GetBytes());

}

return result.ToArray();

}

public static byte[] Decrypt(byte[] data, byte[] key)

{

List<BitArray> keys = GetKeys(key);

List<BitArray> blocks = GetBlocks(data.ToArray());

IEnumerable<byte> result = Enumerable.Empty<byte>();

foreach (var block in blocks)

{

// Делаем первоначальную перестановку

BitArray initialPermutation = Permutate(block, \_IP\_1);

BitArray left = new(32);

BitArray right = new(32);

for (int j = 0; j < 32; j++)

{

left[j] = initialPermutation[j];

right[j] = initialPermutation[32 + j];

}

for (int j = 0; j < 16; j++)

{

BitArray newRight = left.Xor(keys[j % keys.Count]);

left = right;

right = newRight;

}

BitArray combined = new(64);

for (int j = 0; j < 32; j++)

{

combined[j] = left[j];

combined[32 + j] = right[j];

}

result = result.Concat(combined.GetBytes());

}

return result.ToArray();

}

private static BitArray Permutate(BitArray input, byte[] table)

{

BitArray output = new(table.Length);

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

{

output[i] = input[table[i] - 1];

}

return output;

}

private static List<BitArray> GetKeys(byte[] key)

{

if (key.Length != 16)

throw new ArgumentException("Ключ должен быть равен 16 байт");

List<BitArray> keys = new(16);

for (int i = 0; i < 16; i += 4)

{

keys.Add(new BitArray(key.Skip(i).Take(4).ToArray()));

}

return keys;

}

private static List<BitArray> GetBlocks(byte[] data)

{

List<BitArray> blocks = new(data.Length / 8);

for (int i = 0; i < data.Length; i += 8)

{

blocks.Add(new BitArray(data.Skip(i).Take(8).ToArray()));

}

return blocks;

}

}

**ClientServer.cs**

internal class ClientServer

{

private const string \_key = Configuration.ClientKey;

private readonly string \_login;

private readonly int \_requestedLifeTime;

private readonly string \_servicePrincipal;

public ClientServer(string login)

{

\_login = login;

\_requestedLifeTime = 300;

\_servicePrincipal = "Awesome service";

}

public async Task Listen(CancellationTokenSource cancelTokenSource)

{

UdpClient udpClient = new UdpClient(Configuration.ClientPort);

await SendAuthenticationRequest(udpClient);

var response = await udpClient.ReceiveAsync();

var authResponse = DeserializeResponse<ResponseData<AuthenticationServerResponse>>(response.Buffer);

if (authResponse is null)

{

HandleErrorResponse(cancelTokenSource, "Клиент. Ошибка при получении ответа");

return;

}

if (!authResponse.IsSuccess)

{

cancelTokenSource.Cancel();

await Console.Out.WriteLineAsync("Клиент. Неуспешный ответ при аутентификации: " + authResponse.ErrorMessage);

return;

}

var tgtByKDCkey = authResponse.Data!.TGSEncryptByClientKey;

var tgt = DeserializeTicketGrantingTicket(authResponse.Data.TGSEncryptByClientKey);

var sessionKey = tgt.SessionKey;

Console.WriteLine($"\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n" +

$"Клиент получил ответ от сервера аутентификации." +

$"\nСессионный ключ: {sessionKey}" +

$"\n-----------------------------------------------------");

await SendTGSRequest(udpClient, tgtByKDCkey, sessionKey, cancelTokenSource);

}

private async Task SendAuthenticationRequest(UdpClient udpClient)

{

var authData = new AuthenticationServerRequest(\_login, \_requestedLifeTime);

var authRequest = new ResponseData<AuthenticationServerRequest>() { Data = authData, IsSuccess = true };

var data = Encoding.UTF8.GetBytes(JsonSerializer.Serialize(authRequest));

await udpClient.SendAsync(data, Configuration.AuthEndPoint);

Console.WriteLine("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n" +

$"Клиент отправил запрос на аутентификацию с принципалом {\_login} и запрашиваемым" +

$" временем жизни {\_requestedLifeTime} секунд ({\_requestedLifeTime/60} минут)");

}

private TicketGrantingTicket DeserializeTicketGrantingTicket(byte[] encryptedData)

{

return JsonSerializer.Deserialize<TicketGrantingTicket>(encryptedData.GetJsonString(\_key))!;

}

private async Task SendTGSRequest(UdpClient udpClient, byte[] tgtByKDCkey, string sessionKey, CancellationTokenSource cancelTokenSource)

{

var authenticator = new Authenticator(\_login);

var encryptAuth = JsonSerializer.Serialize(authenticator).GetDesEncryptBytes(sessionKey);

Console.WriteLine("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n" +

"Клиент отправляет запрос на разрешение доступа к сервису (TGS)." +

$"\nПринципал сервиса: {\_servicePrincipal}");

var tgsRequest = new TicketGrantServerRequest(\_servicePrincipal, tgtByKDCkey, encryptAuth);

await udpClient.SendAsync(new ResponseData<TicketGrantServerRequest>() { Data = tgsRequest, IsSuccess = true }.GetBytes(), Configuration.TGServerEndPoint);

var response = await udpClient.ReceiveAsync();

var tgsResponse = DeserializeResponse<ResponseData<TicketGrantServerResponse>>(response.Buffer);

if (tgsResponse is null )

{

HandleErrorResponse(cancelTokenSource, "Клиент. Ошибка при получении ответа");

return;

}

if (!tgsResponse.IsSuccess)

{

cancelTokenSource.Cancel();

await Console.Out.WriteLineAsync("Клиент. Неуспешный ответ при аутентификации: " + tgsResponse.ErrorMessage);

return;

}

var st = DeserializeServiceTicket(tgsResponse.Data!.STEncryptBySessionKey, sessionKey);

var sessionServiceKey = st.ServiceSessionKey;

Console.WriteLine("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n" +

"Клиент получил ответ от TGS." +

$"\nСессионный ключ сервиса: {sessionServiceKey}" +

$"\n-----------------------------------------------------");

await SendServiceRequest(udpClient, sessionServiceKey, tgsResponse.Data!.STEncryptByServiceKey, cancelTokenSource);

}

private ServerRequestTicket DeserializeServiceTicket(byte[] encryptedData, string sessionKey)

{

return JsonSerializer.Deserialize<ServerRequestTicket>(encryptedData.GetJsonString(sessionKey))!;

}

private async Task SendServiceRequest(UdpClient udpClient, string sessinServiceKey, byte[] encryptST, CancellationTokenSource cancelTokenSource)

{

var encryptAuth = JsonSerializer.Serialize(new Authenticator(\_login)).GetDesEncryptBytes(sessinServiceKey);

var serviceRequest = new ApplicationServerRequest(encryptAuth, encryptST);

Console.WriteLine("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n" +

"Клиент отправляет запрос на разрешение доступа к сервису (TGS).");

await udpClient.SendAsync(new ResponseData<ApplicationServerRequest>() { Data = serviceRequest, IsSuccess = true }.GetBytes(), Configuration.ServiceServerEndPoint);

var response = await udpClient.ReceiveAsync();

var appResponse = DeserializeResponse<ResponseData<ApplicationServerResponse>>(response.Buffer);

if (appResponse is null || !appResponse.IsSuccess)

{

await HandleErrorResponseAsync(cancelTokenSource, "Клиент. Ошибка при получении ответа от сервиса");

return;

}

if (!appResponse.IsSuccess)

{

cancelTokenSource.Cancel();

await Console.Out.WriteLineAsync("Клиент. Неуспешный ответ при аутентификации: " + appResponse.ErrorMessage);

return;

}

var message = JsonSerializer.Deserialize<string>(appResponse.Data!.ServiceResEncryptByServiceSessionKey.GetJsonString(sessinServiceKey))!;

await Console.Out.WriteLineAsync("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n" +

"Клиент получил ответ от сервиса." +

$"\nПолученное сообщение: {message}" +

$"\n-----------------------------------------------------");

cancelTokenSource.Cancel();

}

private T? DeserializeResponse<T>(byte[] data)

{

return JsonSerializer.Deserialize<T>(Encoding.UTF8.GetString(data));

}

private async void HandleErrorResponse(CancellationTokenSource cancelTokenSource, string errorMessage)

{

cancelTokenSource.Cancel();

Console.WriteLine(errorMessage);

}

private async Task HandleErrorResponseAsync(CancellationTokenSource cancelTokenSource, string errorMessage)

{

cancelTokenSource.Cancel();

await Console.Out.WriteLineAsync(errorMessage);

}

}