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 №3

на тему

**АТАКИ ПРИ УСТАНОВКЕ TCP-СОЕДИНЕНИЯ И ПРОТОКОЛОВ ПРИКЛАДНОГО УРОВНЯ**

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 9](#_Toc157436535)

[Приложение А 13](#_Toc157436537)

# ВВЕДЕНИЕ

Цель данной лабораторной работы заключается в изучении теоретических сведений по работе адресации в сети Internet, физического, канального и транспортного уровней, а также разработке программы, реализующее атаки на протокол при установке TCP-соединения. В интерфейсе приложения необходимо наглядно продемонстрировать передаваемые данные, исходные данные протокола и проверки каждой из сторон.

# 1 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Стек протоколов TCP/IP (Transmission Control Protocol/Internet Protocol, протокол управления передачей/протокол интернета) – сетевая модель, описывающая процесс передачи цифровых данных. модель TCP/IP разделена на уровни, как и OSI, но отличие двух моделей в количестве уровней. Документами, определяющими сертификацию модели, являются RFC 1122 и RFC1123. Эти стандарты описывают четыре уровня абстракции модели TCP/IP: прикладной, транспортный, межсетевой и канальный (см. рисунок 1.1).

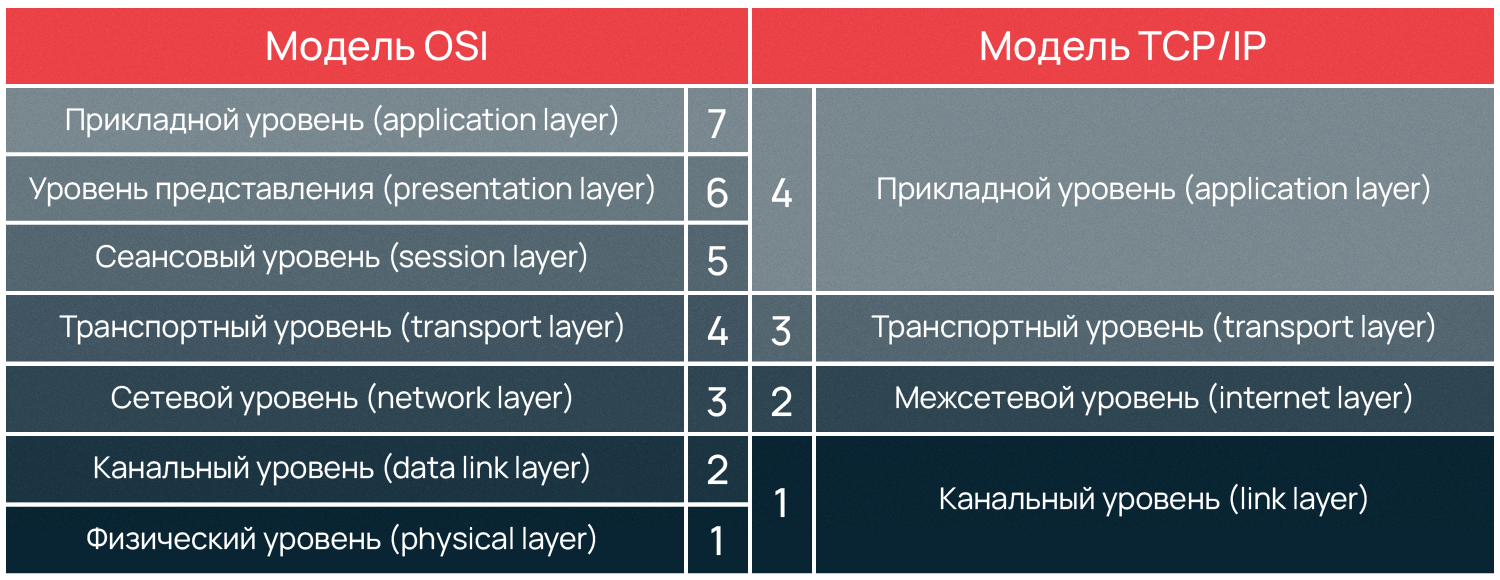


Рисунок 1.1 – Сравнение модели OSI и TCP/IP

Предназначение канального уровня – дать описание тому, как происходит обмен информацией на уровне сетевых устройств, определить, как информация будет передаваться от одного устройства к другому. Информация здесь кодируется, делится на пакеты и отправляется по нужному каналу, т.е. среде передачи. Этот уровень также вычисляет максимальное расстояние, на которое пакеты возможно передать, частоту сигнала, задержку ответа и т.д. Все это – физические свойства среды передачи информации. На канальном уровне самым распространенным протоколом является Ethernet

Глобальная сеть интернет состоит из множества локальных сетей, взаимодействующих между собой. Межсетевой уровень используется, чтобы описать обеспечение такого взаимодействия. Межсетевое взаимодействие – это основной принцип построения интернета. Локальные сети по всему миру объединены в глобальную, а передачу данных между этими сетями осуществляют магистральные и пограничные маршрутизаторы. Именно на межсетевом уровне функционирует протокол IP, позволивший объединить разные сети в глобальную. Как и протокол TCP, он дал название модели.

Постоянные резиденты транспортного уровня — протоколы TCP и UDP, они занимаются доставкой информации.

TCP (протокол управления передачей) – надежный, он обеспечивает передачу информации, проверяя дошла ли она, насколько полным является объем полученной информации и т.д. TCP дает возможность двум конечным устройствам производить обмен пакетами через предварительно установленное соединение. Он предоставляет услугу для приложений, повторно запрашивает потерянную информацию, устраняет дублирующие пакеты, регулируя загруженность сети. TCP гарантирует получение и сборку информации у адресата в правильном порядке.

В заголовке TCP содержатся следующие основные поля:

– Source port (16 бит): порт источника. Порт хоста, от которого исходит запрос.

– Destination port (16 бит): порт назначения. Порт хоста, куда направляется запрос.

– Sequence number, SYN (32 бита): порядковый номер. Позволяет контролировать порядок сообщений. Каждая конечная точка (как порт источника, так и порт назначения) будут поддерживать свой уникальный порядковый номер для отправляемых сообщений. При установлении соединения TCP (используется сообщение с установленным флагом SYN) в качестве изначального порядкового номера будет сгенерировано случайное число.

– Acknowledgement number, ACK (32 бита): номер подтверждения. Когда сообщение содержит флаг ACK, то значение в номере подтверждения должно соответствовать следующему порядковому номеру (SYN), которое отправитель сообщения с флагом ACK ожидает получить от передающей системы. Таким образом, отправка одного номера подтверждения способна подтвердить получение всех байтов с информацией, полученных до этого.

– ACK (1 бит). Устанавливается, когда пакет содержит значение номера подтверждения в поле подтверждения. Все пакеты после стартового пакета SYN будут иметь установленный флаг ACK.

– RST (1 бит): сброс данного соединения. Отправкой пакета RST одна из сторон сообщает о немедленном разрыве соединения. При этом соединение обрывается, а буфер очищается. Самые распространенные причины отправки пакета с установленным флагом RST — ответ на пакет, полученный для закрытого сокета; пользователь сам прервал соединение (например, закрыв браузер, не дожидаясь ответа); соединение не было нормально закрыто, но находится в неактивном состоянии некоторое время.

– SYN (1 бит). Начинает соединение и синхронизирует порядковые номера. Первый пакет, отправленный с каждой стороны, должен в обязательном порядке иметь установленным этот флаг.

– FIN (1 бит). Одна из конечных точек отправляет пакет с установленным флагом FIN для другой конечной точки, чтобы сообщить, что все пакеты были отправлены, и соединение пора завершить.

– Window size (16 бит): размер окна приема. В нем указывается количество байт данных, считая от последнего номера подтверждения, которые готов принять отправитель данного пакета. Другими словами, отправитель данного пакета в этом поле сообщает другой стороне, каким доступным на данный момент размером буфера приема данных он располагает.

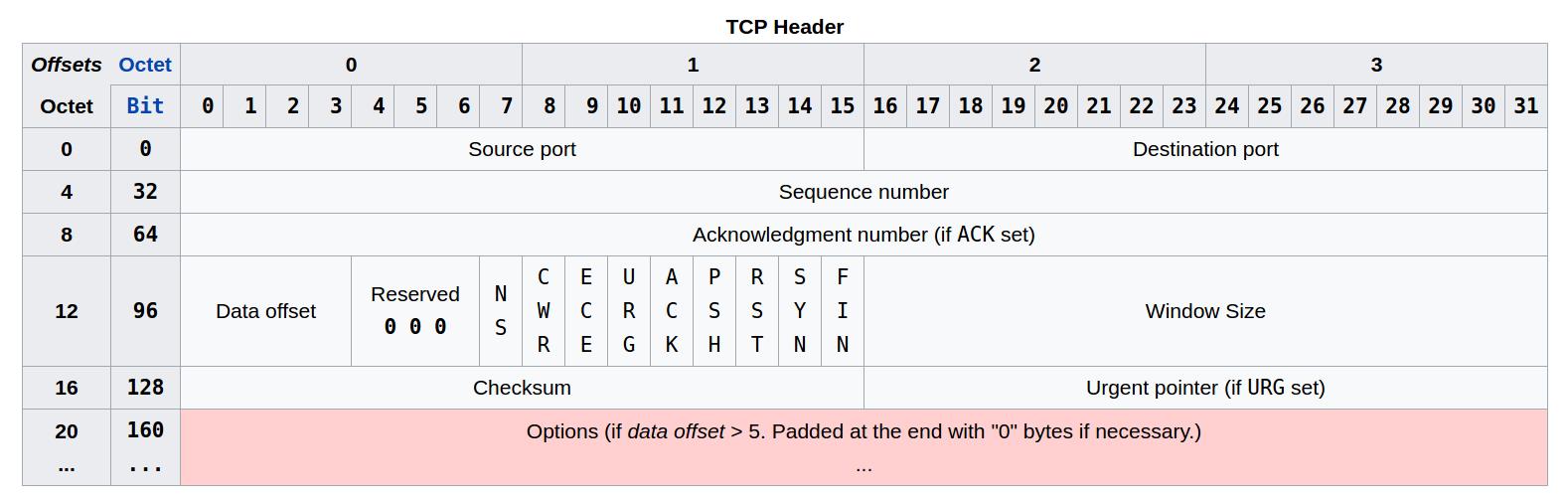


Рисунок 1.2 – Заголовок TCP

Перед тем, как данные могут быть переданы между двумя узлами, в TCP, в отличие от UDP, предусмотрена стадия установки соединения. Также после того, как все данные были переданы, наступает стадия завершения соединения. Таким образом, осуществление каждого TCP-соединения можно условно разделить на три фазы: инициализация соединения, загрузка данных, завершение соединения.

Установка соединения осуществляется с помощью, так называемого трехстороннего рукопожатия TCP (см. рисунок 1.3). Инициатором соединения может выступать любая сторона.

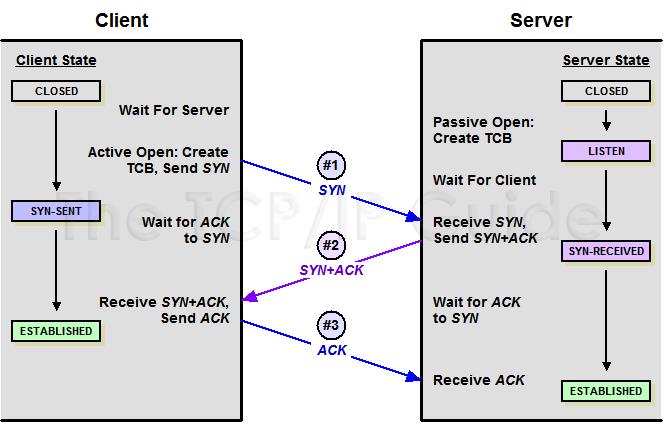


Рисунок 1.3 – Трехстороннее рукопожатие TCP

После инициализации соединения полезная нагрузка будет перемещаться в обоих направлениях TCP-соединения. Все пакеты в обязательном порядке будут содержать установленный флаг ACK. Другие флаги, такие как, например, PSH или URG, могут быть, а могут и не быть установленными.

При нормальном завершении TCP-соединения в большинстве случаев инициализируется процедура, называемая двухсторонним рукопожатием, в ходе которой каждая сторона закрывает свой конец виртуального канала и освобождает все задействованные ресурсы (см. рисунок 1.4). Обычно эта фаза начинается с того, что один из задействованных процессов приложения сигнализирует своему уровню TCP, что сеанс связи больше не нужен. Со стороны этого устройства отправляется сообщение с установленным флагом FIN (отметим, что этот пакет не обязательно должен быть пустым, он также может содержать полезную нагрузку), чтобы сообщить другому устройству о своем желании завершить открытое соединение. Затем получение этого сообщения подтверждается (сообщение от отвечающего устройства с установленным флагом ACK, говорящем о получении сообщения FIN). Когда отвечающее устройство готово, оно также отправляет сообщение с установленным флагом FIN, и, после получения в ответ подтверждающего получение сообщения с установленным флагом ACK или ожидания определенного периода времени, предусмотренного для получения ACK, сеанс полностью закрывается.

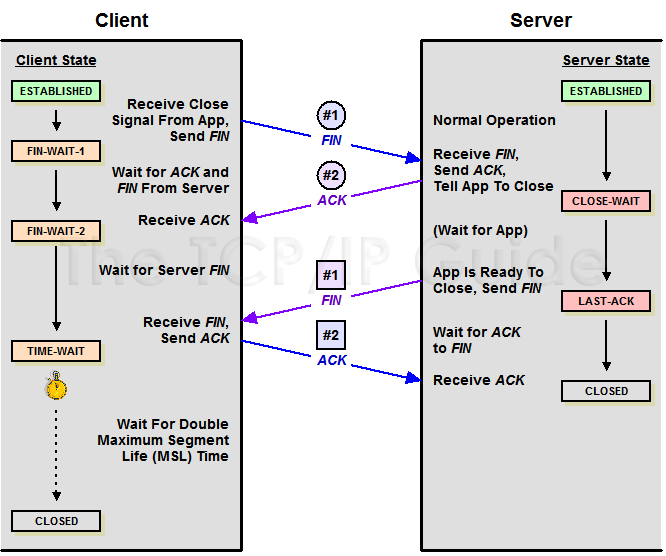


Рисунок 1.4 – Завершение TCP-соединения

# 2 РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ ПРОГРАММЫ

В качестве атак на TCP-протокол были выбраны атака TCP reset вслепую и SYN Flood.

Если атакующий имеет возможность перехвата трафика, которым обмениваются его жертвы, то может считывать порядковые и подтверждающие номера TCP-пакетов жертв. Он может использовать эту информацию для того, чтобы выбирать, какие порядковые номера давать своим фальшивым сегментам RST. Однако если атакующий не может перехватывать трафик жертв, то не будет знать, какие порядковые номера вставлять. Но он всё равно может передавать любое количество сегментов RST с любым количеством разных порядковых номеров, надеясь, что один из них окажется верным. Такая атака называется атакой TCP reset вслепую. В первоначальной версии протокола TCP атакующему достаточно было только подобрать порядковый номер RST в пределах TCP-окна получателя. Это слишком упрощало успешные атаки вслепую, так как для почти гарантированного успеха атакующему достаточно было просто отправить несколько десятков тысяч сегментов. Чтобы противостоять этому, правило, заставлявшее получателя принимать сегмент RST, заменили на строгий критерий. Благодаря новым правилам для осуществления атак TCP reset вслепую нужно отправлять миллионы сегментов, что делает их практически нереализуемыми.

Атака SYN Flood реализуется следующим образом. Клиент генерирует SYN-пакет, запрашивая новую сессию у сервера. Поскольку TCP-сессия открыта (алгоритм «трехэтапного рукопожатия TCP» исполнен), хост будет отслеживать и обрабатывать каждую пользовательскую сессию, пока она не будет закрыта. Во время SYN Flood атакуемый сервер с большой скоростью получает поддельные SYN-запросы, содержащие поддельный IP-адрес источника. SYN Flood поражает сервер, занимая всю память таблицы соединений, обычно используемую для хранения и обработки входящих пакетов. Это вызывает критическое падение производительности и, как итог, отказ в работе сервера.

На рисунке 2.1 представлен корректный результат работы программы, если хакер не атакует соединение.

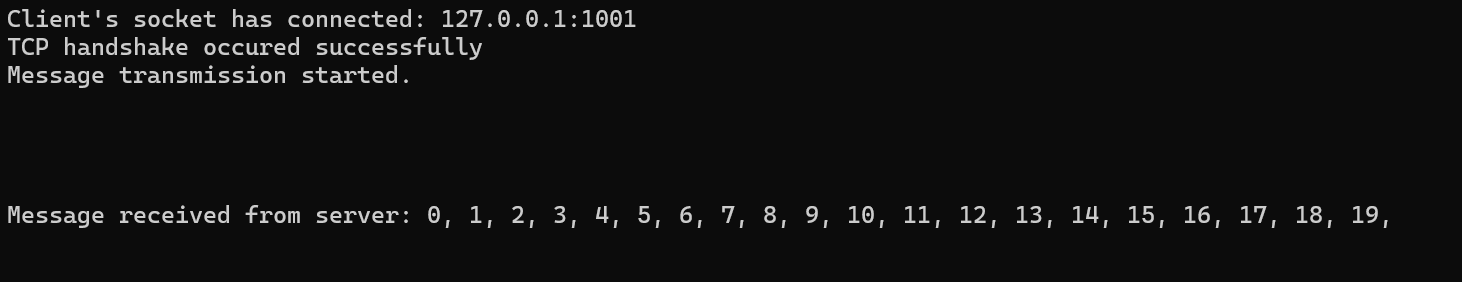


Рисунок 2.1 – Корректный результат программы

На рисунке 2.2 представлен результат программы, когда хакер реализует атаку TCP reset. Соответственно, хакер отправил пакет с установленным флагом RST и измененными портами, чтоб сервер разорвал соединение с клиентом.

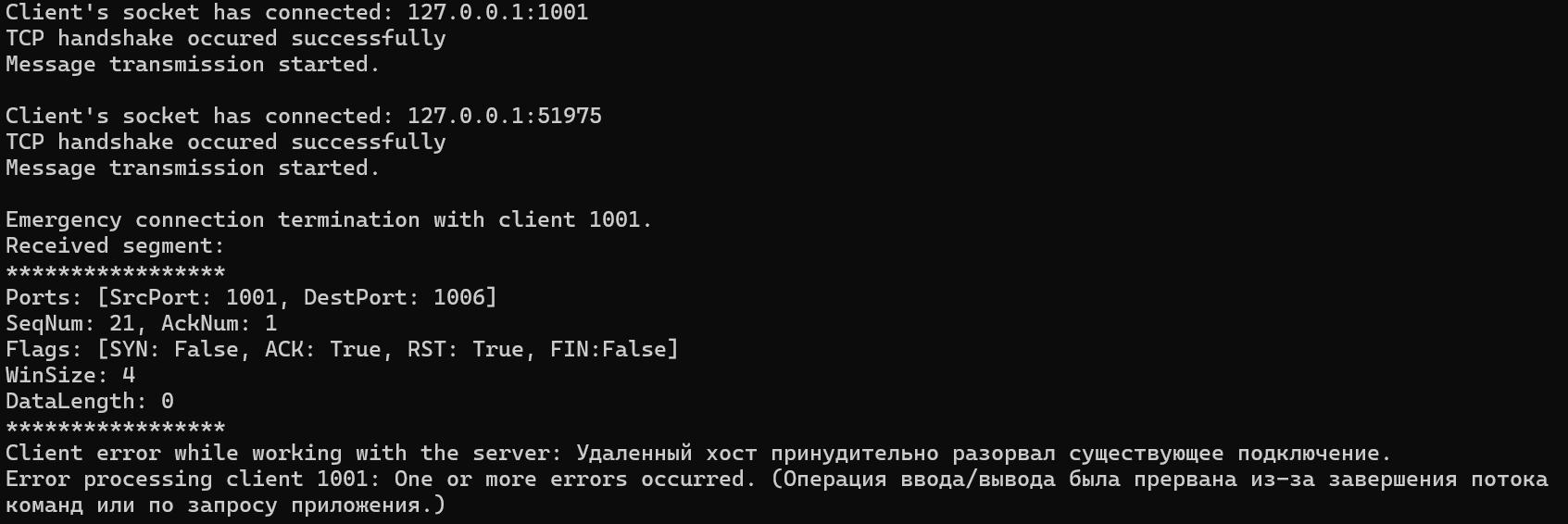


Рисунок 2.2 – Атака TCP reset

На рисунке 2.3 представлен результат программы, когда хакер реализует атаку TCP reset. Соответственно, хакер параллельно отправляет несколько запросов с флагом SYN, но не подтверждает дальнейшего подключения. Т.к. у сервера установлено максимальное время ожидания ответа, то по истечению этого времени сервер разрывает соединение.

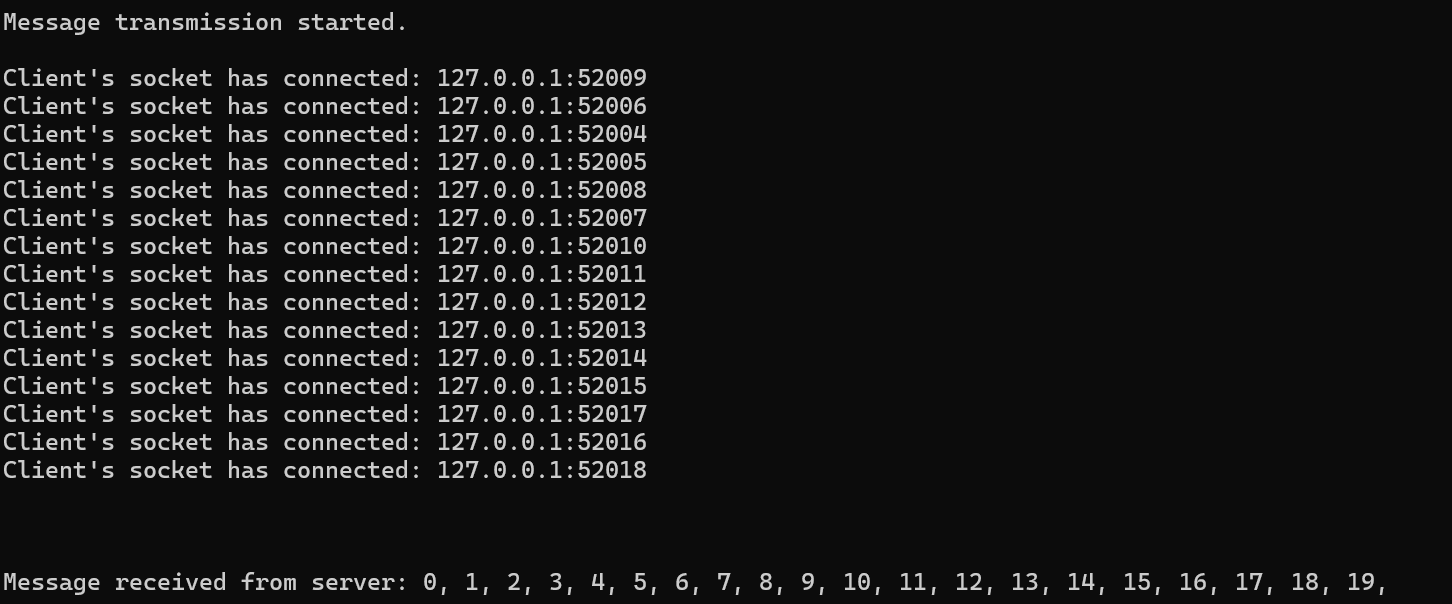


Рисунок 2.3 – Результат атаки SYN Flood

# ПРИЛОЖЕНИЕ А

**(обязательное)**

**Исходный код программы**

**NetworkSegment.cs**

namespace Lab3

{

public class NetworkSegment

{

private int \_sourcePort;

private int \_destinationPort;

private uint \_seqNumber;

private uint \_ackNumber;

private bool \_isSYN;

private bool \_isACK;

private bool \_isRST;

private bool \_isFIN;

private ushort \_winSize;

private byte[] \_payload = [];

public int SourcePort

{

get { return \_sourcePort; }

set { \_sourcePort = value; }

}

public int DestinationPort

{

get { return \_destinationPort; }

set { \_destinationPort = value; }

}

public uint SequenceNumber

{

get { return \_seqNumber; }

set { \_seqNumber = value; }

}

public uint AcknowledgmentNumber

{

get { return \_ackNumber; }

set { \_ackNumber = value; }

}

public bool SYNFlag

{

get { return \_isSYN; }

set { \_isSYN = value; }

}

public bool ACKFlag

{

get { return \_isACK; }

set { \_isACK = value; }

}

public bool RSTFlag

{

get { return \_isRST; }

set { \_isRST = value; }

}

public bool FINFlag

{

get { return \_isFIN; }

set { \_isFIN = value; }

}

public ushort WindowSize

{

get { return \_winSize; }

set { \_winSize = value; }

}

public byte[] Data

{

get { return \_payload; }

set { \_payload = value; }

}

public static NetworkSegment CreateEmptySegment(ushort window, int sourcePort, int destPort, uint seqNum, uint ackNum, bool ack = false, bool syn = false, bool rst = false, bool fin = false)

{

return new NetworkSegment

{

SourcePort = sourcePort,

DestinationPort = destPort,

SequenceNumber = seqNum,

AcknowledgmentNumber = ackNum,

SYNFlag = syn,

ACKFlag = ack,

RSTFlag = rst,

FINFlag = fin,

WindowSize = window,

};

}

public static IEnumerable<NetworkSegment> GetSegments(byte[] data, ushort windowsSize, int sourcePort, int destinationPort, uint seqNum, uint ackNum)

{

List<NetworkSegment> segments = new();

int id = 0;

do

{

segments.Add(new NetworkSegment()

{

SourcePort = sourcePort,

DestinationPort = destinationPort,

SequenceNumber = seqNum,

AcknowledgmentNumber = ackNum,

WindowSize = windowsSize,

Data = data[id..(id + windowsSize < data.Length ? id + windowsSize : data.Length)],

});

seqNum += windowsSize;

}

while ((id += windowsSize) < data.Length);

return segments;

}

public override string ToString()

{

StringBuilder sb = new StringBuilder();

sb.AppendFormat("\n" +

"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n" +

"Ports: [SrcPort: {0}, DestPort: {1}]" +

"\nSeqNum: {2}, AckNum: {3}\n" +

"Flags: [SYN: {4}, ACK: {5}, RST: {6}, FIN:{7}]\nWinSize: {8}\nDataLength: {9}\n" +

"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*",

SourcePort, DestinationPort, SequenceNumber, AcknowledgmentNumber, SYNFlag, ACKFlag, RSTFlag, FINFlag, WindowSize, Data.Length);

return sb.ToString();

}

}

}

**Program.cs**

CancellationTokenSource cancelTokenSource = new CancellationTokenSource();

CancellationToken token = cancelTokenSource.Token;

Client client = new Client();

NetworkNode server = new NetworkNode();

CyberIntruder hacker = new CyberIntruder();

try

{

Task.WaitAll([server.Listen(token),

hacker.ConnectToServer(NetworkNode.ServerIP, token),

client.ConnectToServer(NetworkNode.ServerIP, cancelTokenSource)]);

Thread.Sleep(500);

}

catch (AggregateException ex) when (ex.InnerException is TaskCanceledException)

{ }

**NetworkNode.cs**

namespace Lab3

{

internal class NetworkNode

{

private const int \_port = 1006;

private static IPAddress \_localIPAddress = IPAddress.Parse("127.0.0.1");

private static IPEndPoint \_localEndPoint = new IPEndPoint(\_localIPAddress, \_port);

public static IPEndPoint ServerIP { get => \_localEndPoint; }

private readonly Dictionary<int, Socket> \_clients = [];

public async Task Listen(CancellationToken token)

{

var listenerSocket = ConfigureSocket();

try

{

await SocketListening(listenerSocket, token);

}

catch (OperationCanceledException)

{ }

catch (Exception ex)

{

Console.WriteLine($"Ошибка: {ex.Message}");

}

finally

{

listenerSocket.Close();

listenerSocket.Dispose();

}

}

private Socket ConfigureSocket()

{

Socket listenerSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

listenerSocket.ReceiveTimeout = 1000;

listenerSocket.Bind(\_localEndPoint);

listenerSocket.Listen(10);

return listenerSocket;

}

private async Task SocketListening(Socket listenerSocket, CancellationToken token)

{

while (true)

{

if (token.IsCancellationRequested)

break;

Socket clientSocket = await listenerSocket.AcceptAsync(token);

Console.WriteLine($"Client's socket has connected: {clientSocket.RemoteEndPoint}");

\_clients.Add((clientSocket.RemoteEndPoint as IPEndPoint).Port, clientSocket);

new Thread(() => ProcessClientConnection((clientSocket.RemoteEndPoint as IPEndPoint).Port, token)).Start();

}

}

private void ProcessClientConnection(object clientIdentifier, CancellationToken token)

{

int clientPort = (int)clientIdentifier;

try

{

uint ackNumber, sequenceNumber = 0;

ushort windowSize;

NetworkSegment initialSegment = ReadSegment(clientPort, token).Result;

if (!initialSegment.SYNFlag || initialSegment.Data.Length > 0)

throw new Exception($"Invalid initial connection segment from client {clientPort}: {initialSegment}");

ackNumber = initialSegment.SequenceNumber + 1;

windowSize = initialSegment.WindowSize;

NetworkSegment synAckSegment = NetworkSegment.CreateEmptySegment(windowSize, \_port, clientPort, sequenceNumber, ackNumber, syn: true, ack: true);

SendSegment(clientPort, synAckSegment);

NetworkSegment handshakeConfirmationSegment = ReadSegment(clientPort, token).Result;

if (!handshakeConfirmationSegment.ACKFlag || handshakeConfirmationSegment.Data.Length > 0)

throw new Exception("Invalid connection confirmation segment from client.");

Console.WriteLine("TCP handshake occured successfully\nMessage transmission started.\n");

StringBuilder messageBuilder = new StringBuilder();

for (int i = 0; i < 20; i++)

{

messageBuilder.Append(i).Append(", ");

}

byte[] messageData = messageBuilder.ToString().ToUtf32Bytes();

List<NetworkSegment> messageSegments = NetworkSegment.GetSegments(messageData, windowSize, \_port, clientPort, sequenceNumber, ackNumber).ToList();

foreach (NetworkSegment segmentToSend in messageSegments)

{

Thread.Sleep(10);

SendSegment(clientPort, segmentToSend);

NetworkSegment acknowledgmentSegment = ReadSegment(clientPort, token).Result;

if (!acknowledgmentSegment.ACKFlag || acknowledgmentSegment.Data.Length > 0)

throw new Exception("Incorrect acknowledgment segment received from client.");

if (acknowledgmentSegment.RSTFlag)

{

Console.WriteLine($"Emergency connection termination with client {acknowledgmentSegment.SourcePort}.\nReceived segment: {acknowledgmentSegment}");

\_clients[acknowledgmentSegment.SourcePort].Close();

return;

}

if (segmentToSend.SequenceNumber + windowSize != acknowledgmentSegment.AcknowledgmentNumber)

throw new Exception("Client acknowledged an incorrect set of data.");

}

SendSegment(clientPort, NetworkSegment.CreateEmptySegment(windowSize, \_port, clientPort, sequenceNumber, ackNumber, fin: true));

}

catch (AggregateException ex) when (ex.InnerException is TaskCanceledException)

{ }

catch (Exception ex)

{

Console.WriteLine($"Error processing client {clientPort}: {ex.Message}");

}

finally

{

if (\_clients[clientPort].Connected)

\_clients[clientPort].Close();

}

}

private void SendSegment(int port, NetworkSegment segment)

{

byte[] responseBuffer = segment.ToJsonBytes();

\_clients[port].Send(responseBuffer);

}

public async Task<NetworkSegment> ReadSegment(int port, CancellationToken token)

{

byte[] messageBuffer = new byte[4096];

List<byte> res = new();

int bytesRead = await \_clients[port].ReceiveAsync(messageBuffer, token);

res.AddRange(messageBuffer[0..bytesRead]);

return res.ToArray().DeserializeTcpSegment();

}

}

}

**Client.cs**

namespace Lab3

{

internal class Client

{

private readonly IPEndPoint \_ip = new(IPAddress.Parse("127.0.0.1"), Constants.ClientPort);

public IPEndPoint ClientIP { get => \_ip; }

public Task ConnectToServer(IPEndPoint serverIP, CancellationTokenSource cancelTokenSource)

{

Socket clientSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

clientSocket.Bind(\_ip);

try

{

clientSocket.Connect(serverIP);

SendSegment(clientSocket, NetworkSegment.CreateEmptySegment(4, Constants.ClientPort, serverIP.Port, 0, 0, syn: true));

NetworkSegment segment = ReadSegment(clientSocket);

if (!segment.SYNFlag || !segment.ACKFlag)

throw new Exception("Incorrect initial segment from server");

SendSegment(clientSocket, NetworkSegment.CreateEmptySegment(4, Constants.ClientPort, serverIP.Port, 1, 1, ack: true));

StringBuilder serverMessage = new();

segment = ReadSegment(clientSocket);

do

{

serverMessage.Append(segment.Data.Utf32BytesToString());

Thread.Sleep(10);

SendSegment(clientSocket, NetworkSegment.CreateEmptySegment(4, Constants.ClientPort, serverIP.Port, 1, segment.SequenceNumber + (uint)segment.Data.Length, ack: true));

segment = ReadSegment(clientSocket);

}

while (!segment.FINFlag);

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

"Message received from server: " + serverMessage +

"\n\n\n");

}

catch (Exception ex)

{

Console.WriteLine($"Client error while working with the server: {ex.Message}");

}

finally

{

if (clientSocket.Connected)

clientSocket.Disconnect(false);

clientSocket.Close();

}

cancelTokenSource.Cancel();

return Task.CompletedTask;

}

private void SendSegment(Socket socket, NetworkSegment segment)

{

byte[] responseBuffer = segment.ToJsonBytes();

socket.Send(responseBuffer);

}

public NetworkSegment ReadSegment(Socket socket)

{

byte[] messageBuffer = new byte[4096];

List<byte> res = new();

int bytesRead = socket.Receive(messageBuffer);

res.AddRange(messageBuffer[0..bytesRead]);

return res.ToArray().DeserializeTcpSegment();

}

}

}

**CyberIntruder.cs**

namespace Lab3

{

internal class CyberIntruder

{

public async Task ConnectToServer(IPEndPoint clientIP, CancellationToken token)

{

await Task.Delay(500);

SynFloodAttack();

//ResetAttack(token);

}

private void ResetAttack(CancellationToken token)

{

Socket clientSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

try

{

clientSocket.Connect(NetworkNode.ServerIP);

SendSegment(clientSocket, NetworkSegment.CreateEmptySegment(4, Constants.ClientPort, NetworkNode.ServerIP.Port, 0, 0, syn: true));

NetworkSegment segment = ReadSegment(clientSocket);

if (!segment.SYNFlag || !segment.ACKFlag)

throw new Exception("Incorrect first packer from server");

SendSegment(clientSocket, NetworkSegment.CreateEmptySegment(4, Constants.ClientPort, NetworkNode.ServerIP.Port, 1, 1, ack: true));

for(int i = 5; i < 100; i++)

{

if (token.IsCancellationRequested)

break;

Thread.Sleep(100);

try

{

SendSegment(clientSocket, NetworkSegment.CreateEmptySegment(4, Constants.ClientPort, NetworkNode.ServerIP.Port, (uint)i \* 4 + 1, 1, rst: true, ack: true));

}

catch { }

}

}

catch (Exception ex)

{

Console.WriteLine($"Error occured while intruder worked with server: {ex.Message}");

}

finally

{

if (clientSocket.Connected)

clientSocket.Close();

}

}

private void SynFloodAttack()

{

Parallel.For(5, 20, (int i) =>

{

try

{

Socket socket = new(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

socket.Connect(NetworkNode.ServerIP);

SendSegment(socket, NetworkSegment.CreateEmptySegment(4, Constants.ClientPort, 1000 + i, 0, 0, syn: true));

}

catch { }

});

}

private void SendSegment(Socket socket, NetworkSegment segment)

{

byte[] responseBuffer = segment.ToJsonBytes();

socket.Send(responseBuffer);

}

public NetworkSegment ReadSegment(Socket socket)

{

byte[] messageBuffer = new byte[4096];

List<byte> res = new();

int bytesRead;

bytesRead = socket.Receive(messageBuffer);

res.AddRange(messageBuffer[0..bytesRead]);

return res.ToArray().DeserializeTcpSegment();

}

}

}