Dos Attack in Internet of Things

Software: NetSim Standard v13.0 (32/64 bit), Visual Studio 2019

Project Download Link:

https://github.com/NetSim-TETCOS/DOS_Attack_in_IoT_v13.0/archive/refs/heads/main.zip

Follow the instructions specified in the following link to download and setup the Project in NetSim:

https://support.tetcos.com/en/support/solutions/articles/14000128666-downloading-and-setting-up-netsim-file-exchange-projects

Introduction

A Denial of Service (DoS) attack is an attempt to make a system unavailable to the intended user(s), such as preventing access to a website. A successful DoS attack consumes all available network or system resources, usually resulting in a slowdown or server crash. Whenever multiple sources are coordinating in the DoS attack, it becomes known as a DDoS (Distributed Denial of Service) attack. Standard DDoS Attack types:

- SYN Flood
- UDP Flood
- SMBLoris
- ICMP Flood
- HTTP GET Flood

SYN Flood

TCP SYN floods are DoS attacks that attempt to flood the DNS server with new TCP connection requests. Normally, a client initiates a TCP connection through a three-way handshake of messages:

- The client requests a connection by sending a SYN (synchronize) message to the server.
- The server acknowledges the request by sending SYN-ACK back to the client.
- The client answers with a responding ACK, establishing the connection.

This triple exchange is the foundation for every connection established using the Transmission Control Protocol (TCP). A SYN Flood is one of the most common forms of DDoS attacks. It occurs when an attacker sends a succession of TCP Synchronize (SYN) requests to the target in an attempt to consume enough resources to make the server unavailable for legitimate users. This works because a SYN request opens network communication between a prospective client and the target server. When the server receives a SYN request, it responds acknowledging the request and holds the communication open while it waits for the client to acknowledge the open connection. However, in a successful SYN Flood, the client acknowledgment never arrives, thus consuming the server's resources until the connection times out. A large number of incoming SYN requests to the target server exhausts all available server resources and results in a successful DoS attack. Before implementing this project in NetSim, users have to understand the steps given below:

TCP Log file

- Users need to understand the TCP log file which will get created in the temp path of NetSim <Windows Temp Folder>/NetSim>
- The TCP Log file is usually a very large file and hence is disabled by default in NetSim.
- To enable logging, go to TCP.c inside the TCP project and change the function bool isTCPlog() to return true instead of false.

At malicious node

Create a new timer event called SYN_FLOOD in TCP for sending TCP_SYN packets that should be triggered for every 1000 microseconds. This will create and send the TCP_SYN packet for every 1000 microseconds. SYN request opens network communication between a client and the target.

At Target node

When the target receives a SYN request, it responds acknowledging the request and holds the communication open while it waits for the client to acknowledge the open connection. If a SYN packet arrives at Receiver, it should reply with a SYN_ACK packet. For this SYN_ACK packet, add a processing time of 2000 microseconds in Ethernet Physical Out. This delays the arrival of SYN_ACK at source node. During this delay, another SYN packet will get created at the malicious node. A large number of incoming SYN requests to the target exhausts all available server resources and results in a successful DoS attack **SYN_FLOOD in NetSim**:

C functions for the SYN_FLOOD attack

To implement this project in NetSim, we have created SYN_FLOOD.c file inside TCP project. The file contains the following functions:

- int is_malicious_node(); //This function is used to check the node is malicious node or not.
- int socket_creation(); //This function is used to create a new socket and update the socket parameters.
- static void send_syn_packet(PNETSIM_SOCKET s); //This function is used to create and send SYN packet to the network layer.
- void syn_flood(); //This function is used to check whether the socket is present or not and also adds a timer event called SYN_FLOOD (triggers for every 1000µs)

Steps to simulate the attack

1. Open the Source codes in Visual Studio by going to Your work-> Workspace Options and Clicking on Open code button in NetSim Home Screen window.
2. In Visual Studio, under the **TCP** project in the solution explorer, a **SYN_FLOOD.c** file is added as part of this project.
3. Right click on the solution in the solution explorer and select Rebuild. (Note: first rebuild the TCP project and then rebuild the Ethernet project).

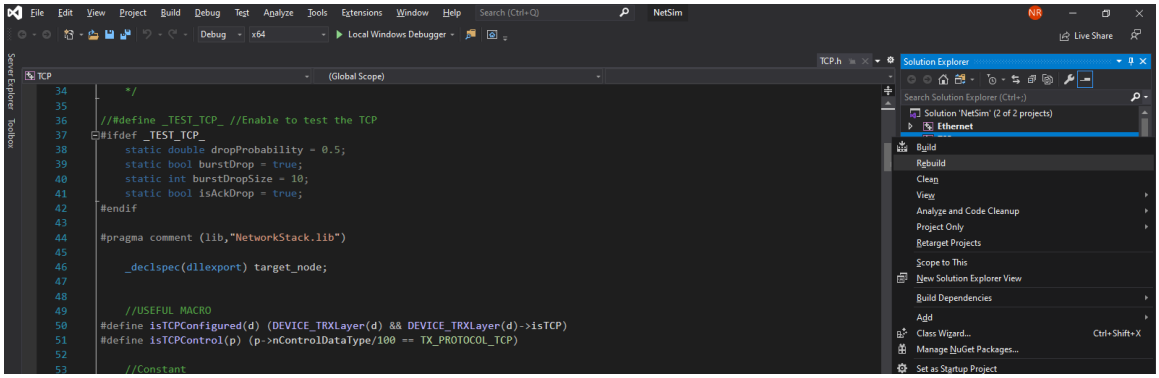


Figure 1: Screen shot of NetSim project source code in Visual Studio

4. Upon successful build modified libTCP.dll and libEthernet.dll file gets automatically updated in the directory containing NetSim binaries.

Running Simulations. Case 1: Without an attacker (malicious nodes)

1. The DOS_Attack_IoT_Workspace comes with a sample configuration that is already saved. To open this example, go to Your work and click on the DOS_Attack_Example_Case_1 from the list of experiments.
2. The saved network scenario consisting of 2 sensors, 1 6LOWPAN Gateway, 1 router, and 1 wired node in the grid environment forming a IoT Network. Traffic is configured from sensor node to the Wired Node.

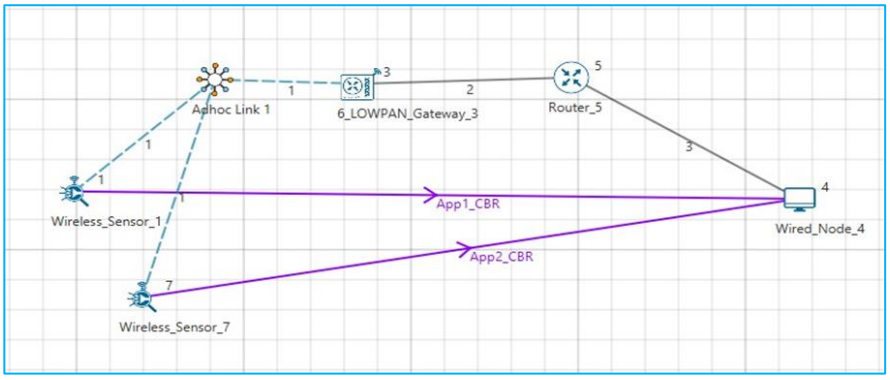


Figure 2: Scenario showing wireless sensor nodes communicating with the server in NetSim

3. Help Open-Source code

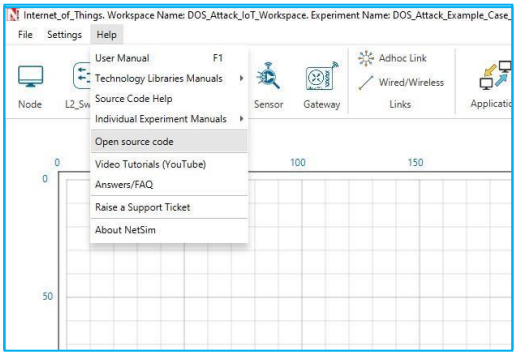


Figure 3: Open-source code in one click

4. In TCP.h set **NUMBEROFMALICIOUSNODE** as 1.
5. In SYN_FLOOD.c set **malicious node** as 0**.**
6. Right click on the solution in the solution explorer and select Rebuild. (Note: first rebuild the TCP project and then rebuild the Ethernet project).
7. Upon successful build modified libTCP.dll and libEthernet.dll file gets automatically updated in the directory containing NetSim binaries.
8. Run the simulation for 100 seconds.

Case-2: With one Malicious Node

1. The DOS_Attack_IoT_Workspace comes with a sample configuration that is already saved. To open this example, go to Your work and click on the DOS_Attack_Example_Case_2 from the list of experiments.
2. The saved network scenario consisting of 3 sensors, 1 6LOWPAN Gateway, 1 router, and 1 wired node in the grid environment forming a IoT Network. Traffic is configured from sensor node to the Wired Node.

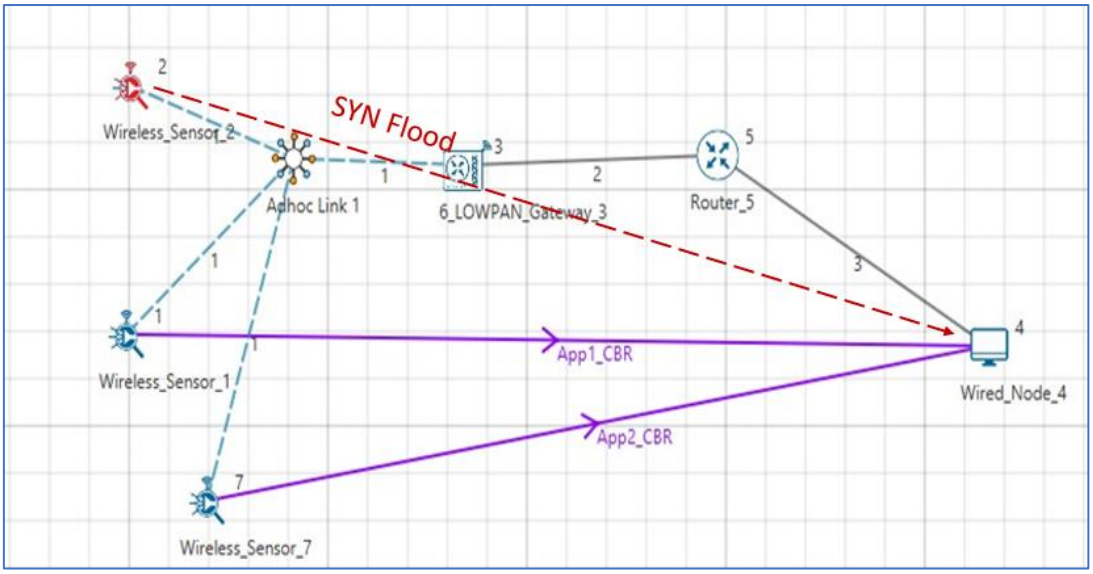


Figure 4: A malicious node initiates a SYN-FLOOD attack

3. Help Open-Source code
4. In TCP.h set **NUMBEROFMALICIOUSNODE** as 1.
5. In SYN_FLOOD.c set malicious node as 2.
6. Right click on the solution in the solution explorer and select Rebuild. (Note: first rebuild the TCP project and then rebuild the Ethernet project).
7. Upon successful build modified libTCP.dll and libEthernet.dll file gets automatically updated in the directory containing NetSim binaries.
8. Run the simulation for 100 seconds.

Case-3: With two Malicious Node

1. The DOS_Attack_IoT_Workspace comes with a sample configuration that is already saved. To open this example, go to Your work and click on the DOS_Attack_Example_Case_3 from the list of experiments.
2. The saved network scenario consisting of 4 sensors, 1 6LOWPAN Gateway, 1 router, and 1 wired node in the grid environment forming a IoT Network. Traffic is configured from sensor node to the Wired Node.

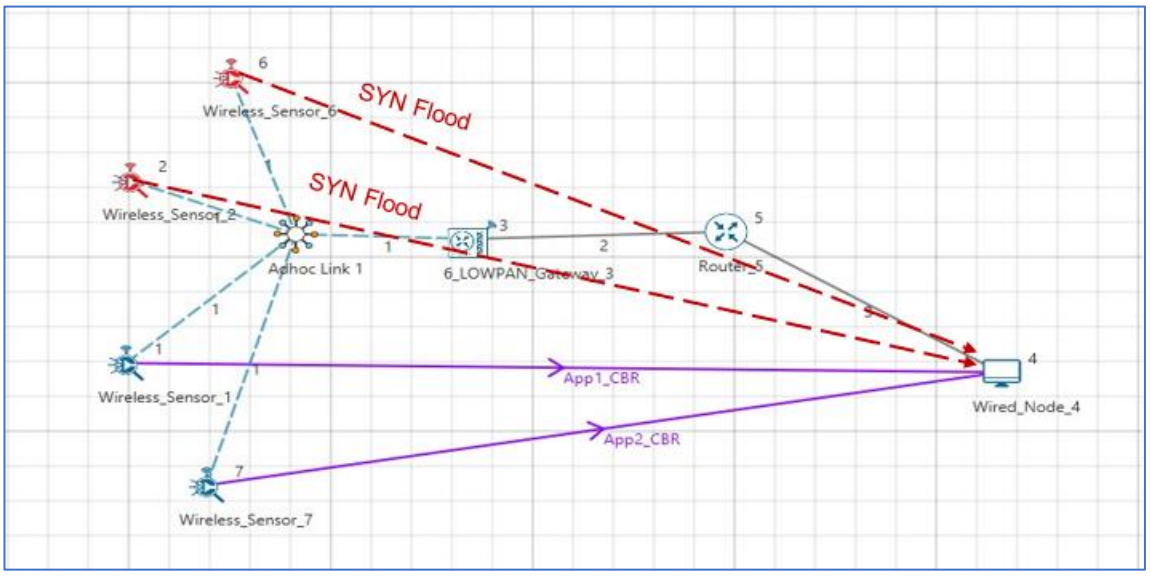


Figure 5: Now two malicious nodes are involved the SYN-FLOOD attack

3. Help Open-Source code.
4. In TCP.h set **NUMBEROFMALICIOUSNODE** as 2.
5. In SYN_FLOOD.c set **malicious node** as 2, 6**.**
6. Right click on the solution in the solution explorer and select Rebuild. (Note: first rebuild the TCP project and then rebuild the Ethernet project).
7. Upon successful build modified libTCP.dll and libEthernet.dll file gets automatically updated in the directory containing NetSim binaries.
8. Run the simulation for 100 seconds.

Results and discussion

After simulation open metrics window and observe the throughput.

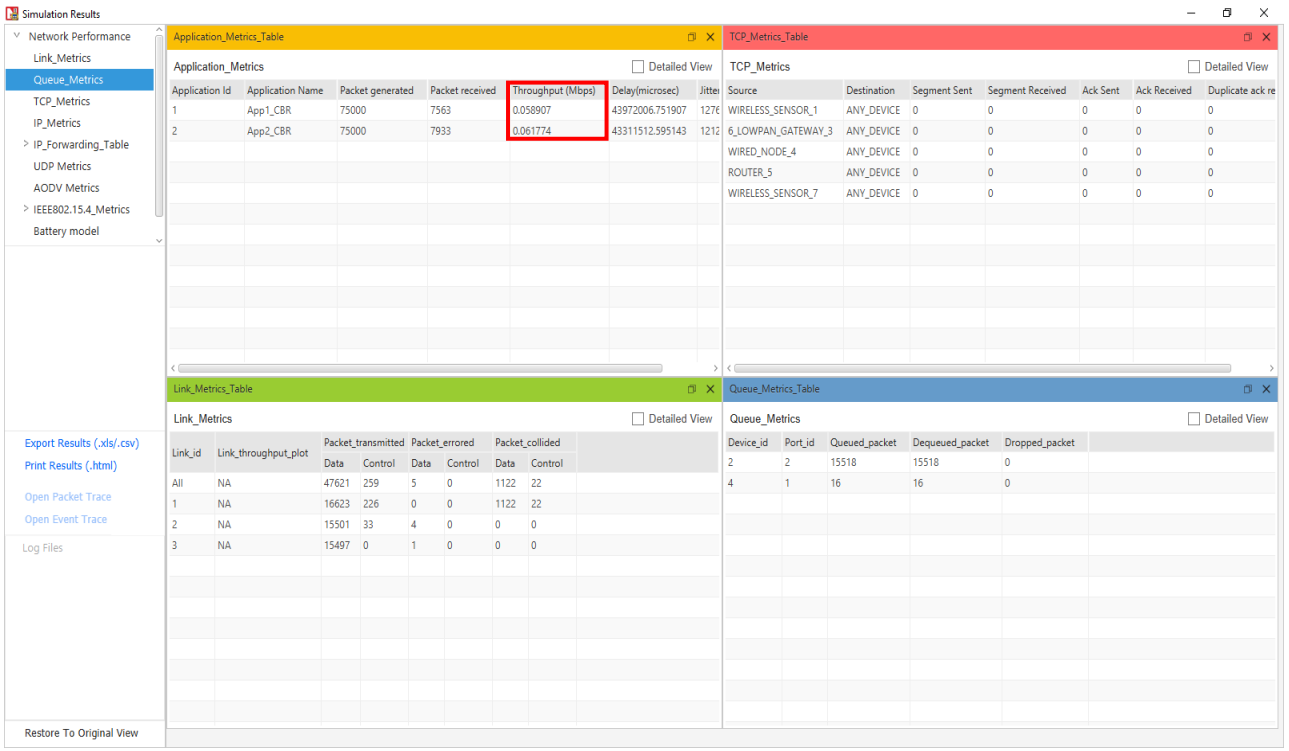
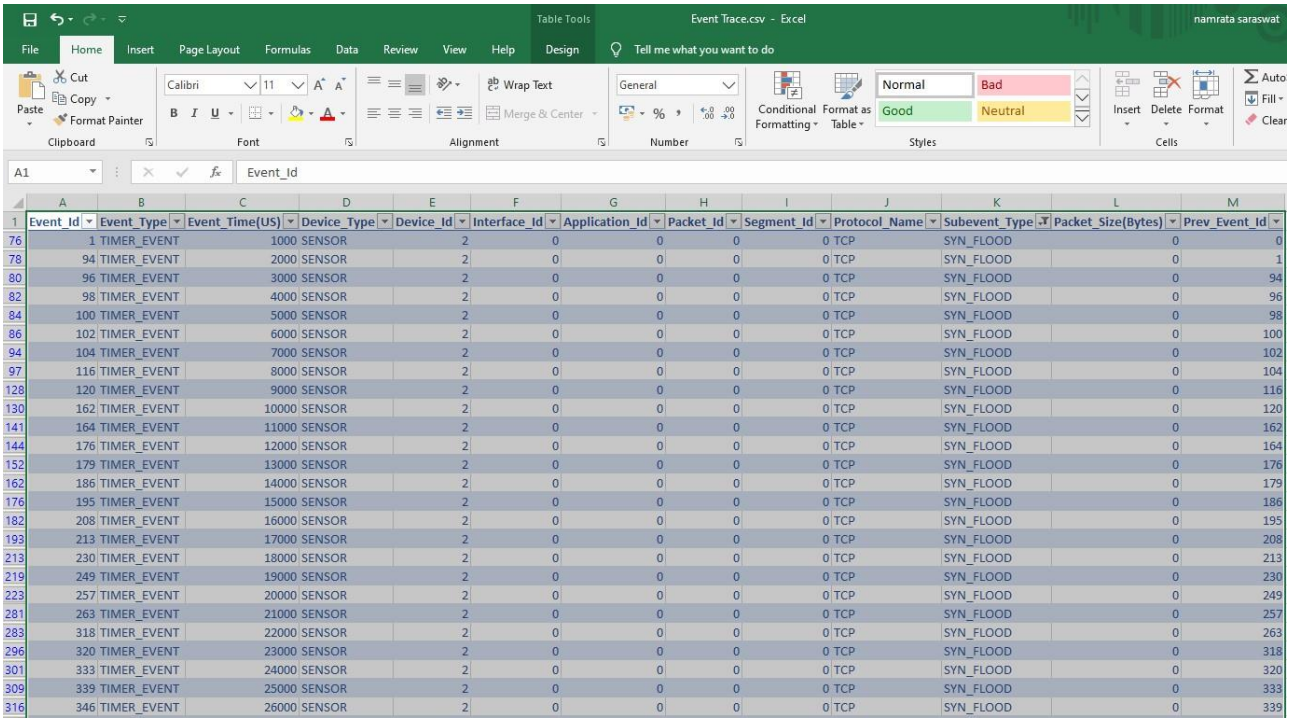


Figure 6: NetSim results dashboard with throughput highlighted

Go to the result window open Event trace, user can find out the SYN_FLOOD packets via filtering subevent type as SYN_FLOOD.



Event Trace.csv - Excel

Event_Id	Event_Type	Event_Time(US)	Device_Type	Device_Id	Interface_Id	Application_Id	Packet_Id	Segment_Id	Protocol_Name	Subevent_Type	Packet_Size(Bytes)	Prev_Event_Id
1	TIMER_EVENT	1000	SENSOR	2	0	0	0	0	TCP	SYN_FLOOD	0	0
94	TIMER_EVENT	2000	SENSOR	2	0	0	0	0	TCP	SYN_FLOOD	0	1
96	TIMER_EVENT	3000	SENSOR	2	0	0	0	0	TCP	SYN_FLOOD	0	94
98	TIMER_EVENT	4000	SENSOR	2	0	0	0	0	TCP	SYN_FLOOD	0	96
100	TIMER_EVENT	5000	SENSOR	2	0	0	0	0	TCP	SYN_FLOOD	0	98
102	TIMER_EVENT	6000	SENSOR	2	0	0	0	0	TCP	SYN_FLOOD	0	100
104	TIMER_EVENT	7000	SENSOR	2	0	0	0	0	TCP	SYN_FLOOD	0	102
116	TIMER_EVENT	8000	SENSOR	2	0	0	0	0	TCP	SYN_FLOOD	0	104
120	TIMER_EVENT	9000	SENSOR	2	0	0	0	0	TCP	SYN_FLOOD	0	116
162	TIMER_EVENT	10000	SENSOR	2	0	0	0	0	TCP	SYN_FLOOD	0	120
164	TIMER_EVENT	11000	SENSOR	2	0	0	0	0	TCP	SYN_FLOOD	0	162
176	TIMER_EVENT	12000	SENSOR	2	0	0	0	0	TCP	SYN_FLOOD	0	164
179	TIMER_EVENT	13000	SENSOR	2	0	0	0	0	TCP	SYN_FLOOD	0	176
186	TIMER_EVENT	14000	SENSOR	2	0	0	0	0	TCP	SYN_FLOOD	0	179
195	TIMER_EVENT	15000	SENSOR	2	0	0	0	0	TCP	SYN_FLOOD	0	186
208	TIMER_EVENT	16000	SENSOR	2	0	0	0	0	TCP	SYN_FLOOD	0	195
213	TIMER_EVENT	17000	SENSOR	2	0	0	0	0	TCP	SYN_FLOOD	0	208
230	TIMER_EVENT	18000	SENSOR	2	0	0	0	0	TCP	SYN_FLOOD	0	213
249	TIMER_EVENT	19000	SENSOR	2	0	0	0	0	TCP	SYN_FLOOD	0	230
257	TIMER_EVENT	20000	SENSOR	2	0	0	0	0	TCP	SYN_FLOOD	0	249
263	TIMER_EVENT	21000	SENSOR	2	0	0	0	0	TCP	SYN_FLOOD	0	257
318	TIMER_EVENT	22000	SENSOR	2	0	0	0	0	TCP	SYN_FLOOD	0	263
320	TIMER_EVENT	23000	SENSOR	2	0	0	0	0	TCP	SYN_FLOOD	0	318
333	TIMER_EVENT	24000	SENSOR	2	0	0	0	0	TCP	SYN_FLOOD	0	320
339	TIMER_EVENT	25000	SENSOR	2	0	0	0	0	TCP	SYN_FLOOD	0	333
346	TIMER_EVENT	26000	SENSOR	2	0	0	0	0	TCP	SYN_FLOOD	0	339

Figure 7: NetSim Event trace with filtered applied to SUB-EVENT-TYPE column. It shows all the SYN_FLOOD Packets and one can notice the 1000 µs inter-packet arrival times in Column C

Case 1 shows the results when there is no attack. The two user applications, attain a throughput of about 0.06 Mbps. In the table we see the throughput of for these two applications falling as we increase the number of attack nodes. This is because the server's resources are being used up in handling the SYN-FLOOD packets and the server is unable to sustain packet transmissions for the regular applications. In this example, with a co-ordinated attack involving 4 systems the throughputs are down 70%.

	Throughput_APP1 (Mbps)	Throughput_APP2(Mbps)
Case-1: Malicious Node =0	0.06	0.06
Case-2: Malicious Node =1	0.05	0.05
Case-3: Malicious Node =2	0.04	0.04

Table 1: Throughputs seen by the user applications. The first row is the throughput when there is no attack. In other samples show the fall in throughputs as the number of attacker systems are increased

Users can similarly run DOS attack simulations on their own networks and analyse its impact on throughput and latency.

Appendix: NetSim source code modifications

Changes to fn_NetSim_TCP_Trace(), in TCP.c file, within TCP project

```
/* This is used to add the SYN_FLOOD sub-events in Event Trace file */

_declspec (dllexport) char *fn_NetSim_TCP_Trace(int nSubEvent)
{
if (nSubEvent == SYN_FLOOD)
        return "SYN_FLOOD";
return (GetStringTCP_Subevent(nSubEvent));
}
```

Changes to fn_NetSim_TCP_HandleTimer(), in TCP.c file, within TCP project

```
/* This is used to call the syn_flood() function periodically */

static int fn_NetSim_TCP_HandleTimer()
{
switch (pstruEventDetails->nSubEventType)
{
case SYN_FLOOD:
        syn_flood();
        break;
case TCP_RTO_TIMEOUT:
        handle_rto_timer();
        break;
```

Changes to fn_NetSim_TCP_Init(), in TCP.c file, within TCP project

```
/* This is used to register the first SYN_FLOOD event */

_declspec (dllexport) int fn_NetSim_TCP_Init(struct stru_NetSim_Network *NETWORK_Formal,
                                             NetSim_EVENTDETAILS *pstruEventDetails_Formal,
                                             char *pszAppPath_Formal,
                                             char *pszWritePath_Formal,
                                             int nVersion_Type,
                                             void **fnPointer)
{
fn_NetSim_TCP_Init_F(NETWORK_Formal,
                     pstruEventDetails_Formal,
                     pszAppPath_Formal,
                     pszWritePath_Formal,
                     nVersion_Type,
                     fnPointer);
NetSim_EVENTDETAILS pevent;
memcpy(&pevent, pstruEventDetails, sizeof pevent);

for (int i = 0; i < NETWORK->nDeviceCount; i++)
{
        if (is_malicious_node(i + 1))
        {
                pevent.nDeviceId = i + 1;
                pevent.dEventTime += 1000;
                pevent.nEventType = TIMER_EVENT;
                pevent.nSubEventType = SYN_FLOOD;
                pevent.nProtocolId = TX_PROTOCOL_TCP;
```

```
                fnpAddEvent(&pevent);

        }
}
return 0;
}
```

Changes to add_timeout_event() in RTO.c file, within TCP project

```
/* This is used to avoid RTO timeouts for malicious nodes */

void add_timeout_event(PNETSIM_SOCKET s,
                       NetSim_PACKET* packet)
{
NetSim_PACKET* p = fn_NetSim_Packet_CopyPacket(packet);
add_packet_to_queue(&s->tcb->retransmissionQueue, p, pstruEventDetails->dEventTime);
NetSim_EVENTDETAILS pevent;
memcpy(&pevent, pstruEventDetails, sizeof pevent);
pevent.dEventTime += TCP_RTO(s->tcb);
pevent.dPacketSize = packet->pstruTransportData->dPacketSize;
pevent.nEventType = TIMER_EVENT;
pevent.nPacketId = packet->nPacketId;
if (packet->pstruAppData)
{
        pevent.nApplicationId = packet->pstruAppData->nApplicationId;
        pevent.nSegmentId = packet->pstruAppData->nSegmentId;
}
else
        pevent.nSegmentId = 0;
if (!is_malicious_node(pevent.nDeviceId))
{
        pevent.nProtocolId = TX_PROTOCOL_TCP;
        pevent.pPacket = fn_NetSim_Packet_CopyPacket(p);
        pevent.szOtherDetails = NULL;
        pevent.nSubEventType = TCP_RTO_TIMEOUT;
        fnpAddEvent(&pevent);
        print_tcp_log("Adding RTO Timer at %0.1lf", pevent.dEventTime);
}
}
```

Changes to TCP.h file, within TCP project

```
/* This is used to define the number of malicious nodes */

#pragma comment (lib,"NetworkStack.lib")
        _declspec(dllexport) target_node;
        //USEFUL MACRO
#define isTCPConfigured(d) (DEVICE_TRXLayer(d) && DEVICE_TRXLayer(d)->isTCP)
#define isTCPControl(p) (p->nControlDataType/100 == TX_PROTOCOL_TCP)

        //Constant
#define TCP_DupThresh         3
#define NUMBEROFMALICIOUSNODE 2
```

Addition of SYN_flood.c file, within TCP project

```
/* This is used to define the malicious node ID's and the target node ID */
/* This has functions defined for SYN flood attack*/
```

```
#include "main.h"
#include "TCP.h"
#include "List.h"
#include "TCP_Header.h"
#include "TCP_Enum.h"

int malicious_node[NUMBEROFMALICIOUSNODE] = {2,6};
static void send_syn_packet(PNETSIM_SOCKET s);
//static PNETSIM_SOCKET socket_creation();
int target_node = 4;
PNETSIM_SOCKET get_Remotesocket(NETSIM_ID d, PSOCKETADDRESS addr);
static PSOCKETADDRESS sockAddr = NULL;

int is_malicious_node(NETSIM_ID devid){}
void syn_flood(){}
static void send_syn_packet(PNETSIM_SOCKET s){}
int socket_creation(){}
```

Changes to TCP_Enum.h file, within TCP project

```
/* This is used to a new SYN_FLOOD subevent in TCP_Subevent */

#include "EnumString.h"

BEGIN_ENUM(TCP_Subevent)
{
DECL_ENUM_ELEMENT_WITH_VAL(TCP_RTO_TIMEOUT, TX_PROTOCOL_TCP * 100),
DECL_ENUM_ELEMENT(TCP_TIME_WAIT_TIMEOUT),
DECL_ENUM_ELEMENT(SYN_FLOOD),
}
```

Changes to Ethernet.h file, within ETHERNET project

```
/* This is used to define processing time for syn_flood packets */

#ifndef _NETSIM_ETHERNET_H_
#define _NETSIM_ETHERNET_H_
#ifdef  __cplusplus
extern "C" {
#endif

#pragma comment(lib,"NetworkStack.lib")
#pragma comment(lib,"Metrics.lib")
#pragma comment (lib,"libTCP.lib")
#define isETHConfigured(d,i) (DEVICE_MACLAYER(d,i)->nMacProtocolId ==
MAC_PROTOCOL_IEEE802_3)
//Global variable
PNETSIM_MACADDRESS multicastSPTMAC;

#define ETH_IFG 0.960 //Micro sec
#define Processing_TIME 1000
```

Changes to fn_NetSim_Ethernet_HandlePhyOut() in Ethernet_Phy.c file, within ETHERNET project

```
/* This is used to add processing delay for TCP SYN packets */

/* This is used to add processing delay for TCP SYN packets */

double start;
if (pstruEventDetails->nDeviceId == target_node && (packet->nControlDataType == 40102 || packet-
>nControlDataType == 40105))
{
	if (phy->lastPacketEndTime + phy->IFG <= pstruEventDetails->dEventTime)
		start = pstruEventDetails->dEventTime + Processing_TIME;
	else
		start = phy->lastPacketEndTime + phy->IFG + Processing_TIME;
}
else
{
	if (phy->lastPacketEndTime + phy->IFG <= pstruEventDetails->dEventTime)
		start = pstruEventDetails->dEventTime;
	else
		start = phy->lastPacketEndTime + phy->IFG;
}
```

TCP Project Properties:

- Right click on TCP project and select Properties.
- In Linker section go to Advanced
- The import library value has been updated for 32-bit and 64-bit source code settings.
    - 32-bit as ***..\lib\lib$(TargetName).lib***
    - 64-bit as ***..\lib_x64\lib$(TargetName).lib***

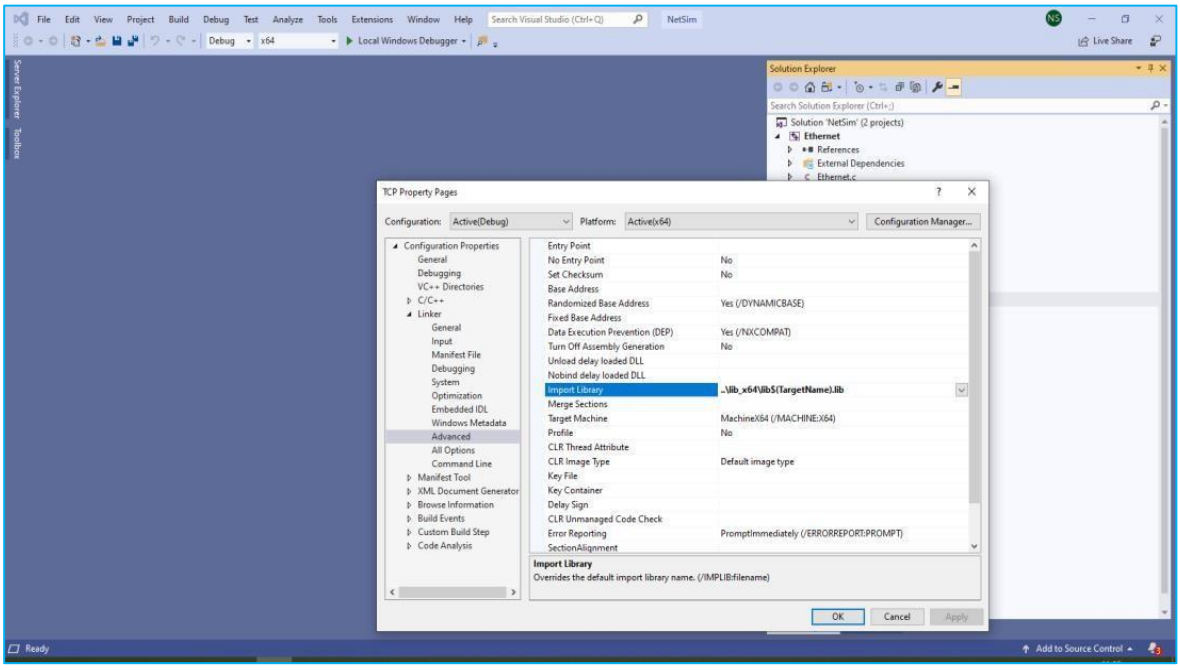


Fig 8: Visual Studio project settings