Bluetooth Proxy Part B

Final report

|  |
| --- |
| Written by |
| Avishay Shasha |
| Itay Zach |

*Be advised that in order to have a fully understanding of this document, one must first read the BT Proxy report of part A.*

# Gratitude and Thanks

* Eli Meirom for the supervision, guidance, tips and mental support during the project.
* Roy Miterany for the equipment and supplies that contributed a great deal for the development process.

# Contents

[1 Abstract 5](#_Toc443154830)

[2 Project Overview 6](#_Toc443154831)

[2.1 Terms And Definitions 6](#_Toc443154832)

[2.2 Overview 6](#_Toc443154833)

[3 Technical Background 8](#_Toc443154834)

[3.1 Bluetooth Discovery 8](#_Toc443154835)

[3.2 Cloud Computing 8](#_Toc443154836)

[4 Architecture 9](#_Toc443154837)

[4.1 Support of devices discovery 9](#_Toc443154838)

[4.1.1 Blocks diagram 9](#_Toc443154839)

[4.1.2 Implementation 9](#_Toc443154840)

[4.2 Cloud server 9](#_Toc443154841)

[4.2.1 Blocks diagram 9](#_Toc443154842)

[4.2.2 Implementation 9](#_Toc443154843)

[5 Issues We Came Across During The Project 10](#_Toc443154844)

[6 Results 11](#_Toc443154845)

[7 Conclusions 12](#_Toc443154846)

[8 References 13](#_Toc443154847)

[8.1 MS Detours 13](#_Toc443154848)

[8.2 Android 13](#_Toc443154849)

[8.3 Windows Sockets 13](#_Toc443154850)

# Table of Figures

[Figure 1: Given system 6](#_Toc444622206)

[Figure 2: BT proxy complete system 6](#_Toc444622207)

[Figure 3: Part B complete system 7](#_Toc444622208)

[Figure 4: Cloud server blocks diagram 13](#_Toc444622209)

[Figure 5: Device discovery support flow 14](#_Toc444622210)

# Abstract

“Bluetooth proxy” project’s goal is to supply a software solution for the BT proximity limitation. In the first part of the project we have supplied a solution that included two sub-systems:

1. A Windows C based application that wraps the original BT application.
2. An Android application that acts as a proxy of the Windows application which translates TCP data from the Windows application and BT data from the BT device.

During the development of the first part, we had two assumptions:

1. We assumed a pre-known MAC address of the BT device.
2. We assumed a fixed IP for the Android device that runs the BT Proxy application.

The system we have developed in part B of “BT Proxy” project omits the need to have those assumptions and supplies a more realistic solution of the BT proximity limitation problem.

# Project Overview

## Terms And Definitions

|  |  |
| --- | --- |
| Term | Definition |
| SDP | Service Discovery Protocol |
| UUID | Universally Unique Identifier |
| Cloud Server | CS |
| Bluetooth Proxy | BTP |
| Bluetooth device | BTD |
| Hooking wrapper | HKW |

Table : Terms and Definitions

## Overview

The given system consists of a BT device and a Windows BT application as depicted in Figure 1.



Figure : Given system

The solution for the proximity limitation problem (in part A of the project) is depicted in Figure 2, which includes:

1. Hooking wrapper that intercepts the Bluetooth Windows API to create a TCP connection.
2. BT Proxy Android application to translate the data from TCP/IP to BT.



Figure : BT proxy complete system

There are two underlying assumptions in the system depicted in Figure 2:

1. The BT device has a known MAC address.
2. The Android device running BT Proxy has a fixed IP.

Hence, we supply two solutions to avoid having these assumptions:

1. Usually, before establishing a BT connection the BT client scans for a nearby BT server name, or server services. The BT Proxy app now supports this kind of discovery procedure by receiving queries from the DLL and sending back the discovered devices data.
2. A cloud always on server that allows the establishing of the TCP link between the DLL and the BT Proxy Android device.

The new system is depicted in Figure 3:



Figure : Part B complete system

# Technical Background

## Bluetooth Discovery

### Devices discovery with BT Winsock API

To facilitate the discovery of Bluetooth devices and services, Windows maps the Bluetooth Service Discovery Protocol (SDP) onto the Windows Sockets namespace interfaces.

The primary functions used for this mapping are the WSALookupServiceBegin, WSALookupServiceNext, and WSALookupServiceEnd functions.

The WSAQUERYSET structure is also used in conjunction with these functions.

It is important to note that the WSA functions are not intended to be used by the SDP solely, and they are rather capable of supporting different kinds of protocols. As a result, not all of the WSAQUERYSET data structures fields are used, and the fields that are used don’t always have a name that is logically related to the actual data that is stored in that field.

1. **WSAQUERYSET data structure**

In case of looking for a BT server name, the following fields are used:

* lpszServiceInstanceName – holds the display name of the device in case LUP\_RETURN\_NAME flag is used when calling WSALookupServiceBegin.
* lpServiceClassId – the GUID (UUID) of the service that is being looked for.
* lpcsaBuffer –holds the found device’s MAC address if LUP\_RETURN\_ADDR flag is used when calling WSALookupServiceBegin.

1. **WSALookupServiceBegin**

Initiates a client query that is constrained by the information contained within a WSAQUERYSET structure. WSALookupServiceBegin only returns a handle, which should be used by subsequent calls to WSALookupServiceNext to get the actual results.

lphLookup holds the desired flags for the service inquiry.

1. **WSALookupServiceNext**

After obtaining a handle from WSALookupServiceBegin, the WSALookupServiceNext is called and supplies the information about a device that was found.

The provider will pass back a WSAQUERYSET structure in the lpqsResults buffer. The client should continue to call this function until it returns WSA\_E\_NO\_MORE, indicating that all of WSAQUERYSET has been returned.

1. **WSALookupServiceEnd**

Called to free the handle after previous calls to WSALookupServiceBegin and WSALookupServiceNext.

### Devices discovery with Android BT API

In order to facilitate a device discovery, one should use the BluetoothAdapter andBroadcastReceiever classes. The BluetoothAdapter represents the local Bluetooth adapter and allows the user to find remote devices either by performing device discovery or by querying the list of paired devices.

In order to start device discovery, the startDiscovery() method of the BluetoothAdapter instance should be called.

Before starting the discovery, the application must register a BroadcastReceiver for an ACTION\_FOUND Intent to get the information about each device that was discovered.

The following snip code demonstrates the use:

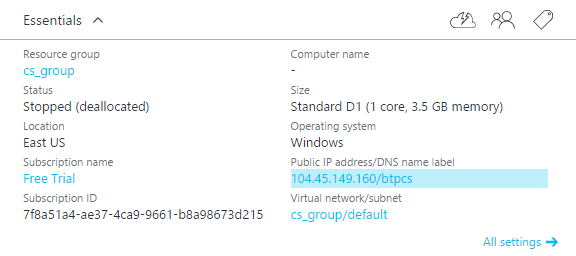


## Cloud server (powered by MS Azure)

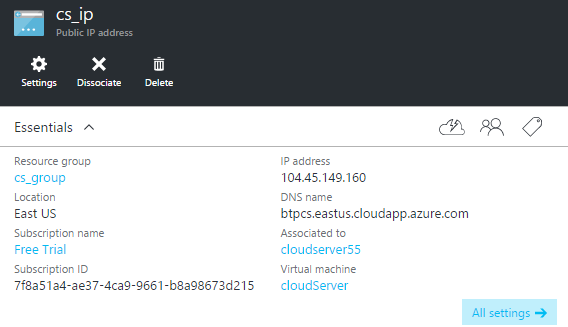
Microsoft Azure allows many cloud services. Virtual Machines (VMs) is among those services.

It is possible to run a Windows 2012 Server VM with native C code. In order to have a functional TCP server, the following steps are necessary (through Azure web portal):

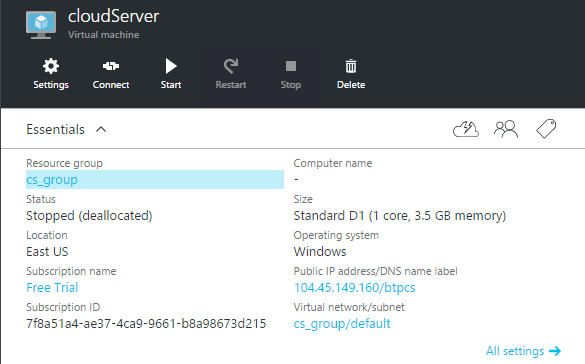
1. Create and start a virtual machine.
2. Create a public IP address.
   1. From the main VM screen, choose the Public IP address:



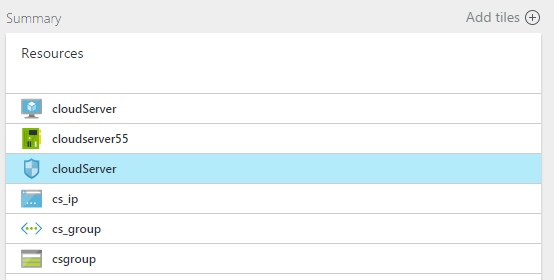
* 1. Add a new Public IP:



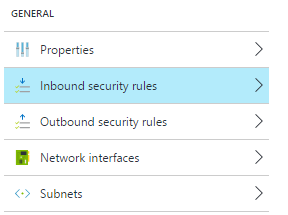
1. Allow the desired port (4020 in BT Proxy CS case)
   1. From the main VM screen, choose Resource group:



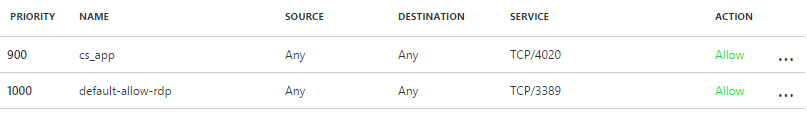
* 1. Choose the security group:



* 1. Add an in-bound security rule:



* 1. Add the new rule with a proper priority:



1. Now, as the security settings are defined, a native TCP C code can be run and accept connections from port 4020.

# Architecture

## 4.1 Cloud server

### Blocks diagram



Figure : Cloud server blocks diagram

### Implementation

In order to solve the fixed IP of the TCP server problem, we decided to implement a TCP server which lies on a cloud service supplied by Microsoft Azure.

The CloudServer (CS) is “always on” waiting for TCP clients to connect, and creates a TCP connection between the HKW and BTP.

In order to identify which client was connected, a pre-defined ID of a client had to be defined. In the small scale system that was implemented, two IDs were selected:

|  |  |
| --- | --- |
| Client | ID |
| Hooking wrapper | “windspc” |
| BT Proxy App | “btproxy” |

Table : Client IDs for CloudServer identification

After a client is connected a communication handler thread is opened to wait for messages from the client (with recv()) and send them to the other side (by send()).

If the other side was not connected yet, an error message is sent back to the source, making it aware that the other side is not connected yet (“msgFromCS\_sendToDstFailed”).

*Note: In case of the HKW, if BTP is not connected yet, HKW will notify the original BT app to handle the connection through a Bluetooth socket. It means, in other words, that the original app must not rely on the TCP connection to the BTP, and just run as originally intended.*

As the connection is established from both sides, the server acts as a tube from HKW and BTP, receiving and sending data from one side to another.

The data that is transferred might be commands, as described in the next session (Support of device discovery), or actual data.

As the original BT app closes the socket as a client, HWK sends the command to BTP. The BTP closes the BT connection to the device, but is still connected to CS, in case the original application runs again.

## Support of devices discovery

### Blocks diagram

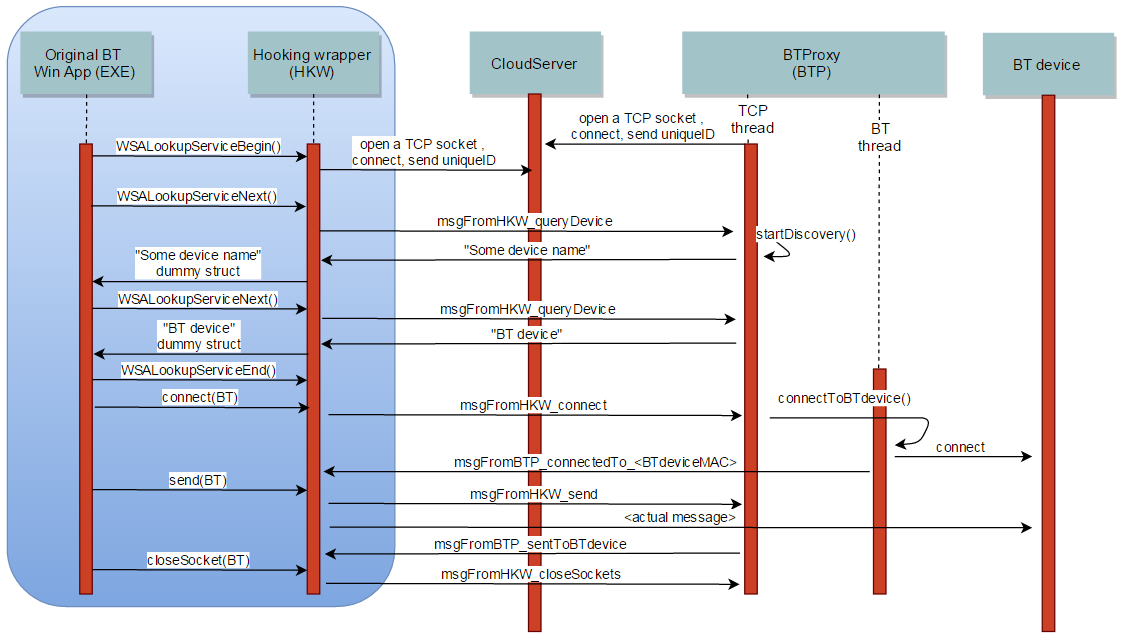


Figure : Device discovery support flow

### Implementation

Devices discovery is supported by a pre-defined protocol that allows the BT Proxy and the Hooking wrapper to communicate and send messages through an established TCP connection.

The CloudServer is always alive and waiting for clients to connect.

When the original BT Win App starts the discovery, the DLL opens a TCP connection to the CloudServer and asks the BTProxy to start the discovery.

With every device that was found by the BT Proxy, its’ name is sent back to the original app, which decides whether to continue the discovery (device was found) or to continue. Note that the DLL informs the original app with the found device by creating a dummy data structure as the WSALookupServiceNext creates in the LPWSAQUERYSET lpqsResults parameter.

After the device is found, the connect method called by the original app initiates the BT Proxy to connect to the BT device, what makes the entire system to act as a tube from the original application to the actual BT device.

After the data is sent, the original application will close its’ socket (which is not really a BT socket), what will make the entire system to shut down, except for the CloudServer that is ready for new connections.

Note:

In case the original BT app desires to connect to several BT devices, HKW identifies the different connections (through different BT sockets) by checking the BT socket in the following manner:

When the device is found remotely by BTP, according to a dummy MAC address that is attached to the found device in HKW, it is possible to save the socket that is opened with this MAC address and save it as a global variable. Using this specific socket number, HKW can identify whether the device is remote or local.

# Results

The following screen captures reflect the fully functional system, as described in figure 5:

|  |  |
| --- | --- |
| 1. CS is on, waiting for clients to connect. Once BTP connects, CS displays the following: | |
| 1. BTP is connected to CS: |  |
| 1. HKW runs the original BT app:     “btdevice” is found by BTP, and the MAC address the original BT app receives is the dummy MAC address (0xAABBCCDDEEFF). Using this MAC address it is possible to identify if the device is remote or local. | |

|  |
| --- |
| 1. The result of this run on BTP is: |
| 1. BT device received the message: |

# Conclusions

## Debugging can affect the functional code

In order to debug the hooked functions from the HKW DLL, a prints to debug txt file were made. During the development of the hook for WSALookupServiceNext(), there was a need to identify the last WSA error in the case of no more BT devices are found in the discovery procedure. If no devices are found, the WSAGetLastError() function should return WSA\_E\_NO\_MORE. But since a new file for debug logs was opened, a different error was returned, ERROR\_ALREADY\_EXISTS. This issue forced a hook for WSAGetLastError() to return the actual error to the original BT app, WSA\_E\_NO\_MORE.

## Use the same compiler for the entire project

When HKW receives a response from a new query to BTP, a dummy WSAQUERYSET structure must be returned to the original BT app.

During the creation of this structure, a new memory allocation of the MAC address and device name must be done in the DLL. Those memory allocations are returned through a pointer that is read in the original BT app.

Since the original BT app and the HKW DLL were compiled with different compliers, an unexplained situation occurred – the pointer pointed to a different address in HKW DLL and original BT app.

The reason for this weird situation was that the older compiler (used to compile the original BT app) approved to use older types than the ones used in HKW DLL, what caused a “shift” of the pointer address.

The solution is to use the same compiler to the entire project.

# References

## MS Detours

[1] API Hooking with MS Detours, CodeProject.com

<http://www.codeproject.com/Articles/30140/API-Hooking-with-MS-Detours>

[2] API Hooking and DLL injection, Infosec Institute,

<http://resources.infosecinstitute.com/api-hooking-and-dll-injection-on-windows/>

## Android

[3] Android Reference Guide,

<http://developer.android.com/reference/packages.html>

## Windows Sockets

[4] Bluetooth Programming with Windows Sockets, Microsoft,

<https://msdn.microsoft.com/en-us/library/windows/desktop/aa362928(v=vs.85).aspx>

[5] Winsock Functions, Microsoft,

<https://msdn.microsoft.com/en-us/library/windows/desktop/ms741394(v=vs.85).aspx>