BLUETOOTH PROXY
FINAL REPORT

Written by
Avishay Shasha
Itay Zach

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
2 Project Overview 6
2.1 Terms And Definitions 6
2.2 Overview 6
3 Technical Background 8
3.1 Function/API Hooking (or: Interception) 8
3.2 Microsoft Detours 8
3.2.1 DLL Injection 9
3.3 Windows Socket API 11
4 Architecture 12
4.1 Hooking Wrapper 12
4.1.1 Blocks Diagram 12
4.1.2 Implementation 12
4.1.3 Injector 12
4.1.4 Hooked WSA Functions 13
4.2 Bluetooth Proxy Application 15
4.2.1 Blocks Diagram 15
4.2.2 Implementation 15
5 Issues We Came Across During The Project 16
5.1 Sockets and OS 16
6 Results 17
7 Conclusions 19
7.1 Learn by “getting dirty” 19
7.2 Have a plan 19
7.3 Divide and conquer 19
8 References 20
8.1 MS Detours 20
8.2 Android 20
8.3 Windows Sockets 20

TABLE OF FIGURES

Figure 1 - Given system 6
Figure 2 - BT proxy complete system 7
Figure 3 - Hooking wrapper blocks diagram 12
Figure 4 - BT Proxy app blocks diagram 15

1 ABSTRACT

Today’s world is all about communication and instant activities.

Bluetooth is a well-known protocol many devices and application are based on. Bluetooth has many advantages that allow almost every day usage – data can be transferred between two stand-alone devices with no need of a mainframe or a large system.

But the main problem with Bluetooth is the short distance limitation. So what if you could run an already existing application without the distance limitation?

The “Bluetooth proxy” aims to solve this problem by taking over the Windows Bluetooth API functions and send the data over TCP/IP to an Android device that will be used as the Bluetooth Proxy.

2 PROJECT OVERVIEW

2.1 TERMS AND DEFINITIONS

Term	Definition
WSA	Windows Socket API
BT	Bluetooth
DLL	Dynamic Link Library

2.2 OVERVIEW

Bluetooth Proxy is meant to take control over a given Bluetooth system BT functions in order to transmit the data over TCP/IP.

Let us say a doctor has to configure a BT based hearing device for the patient, the hearing device (and the patient) have to come to the doctor's clinic for the configuration to be done.

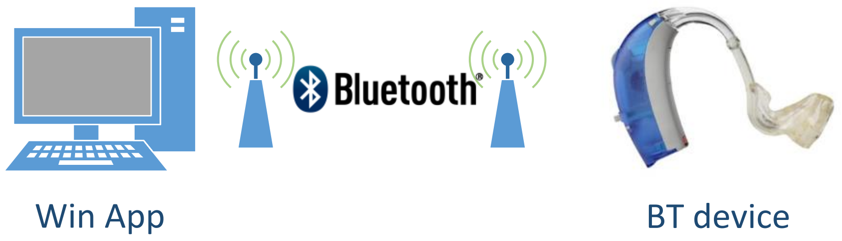


Figure 1 - Given system

The idea is to expand the given system to the following:

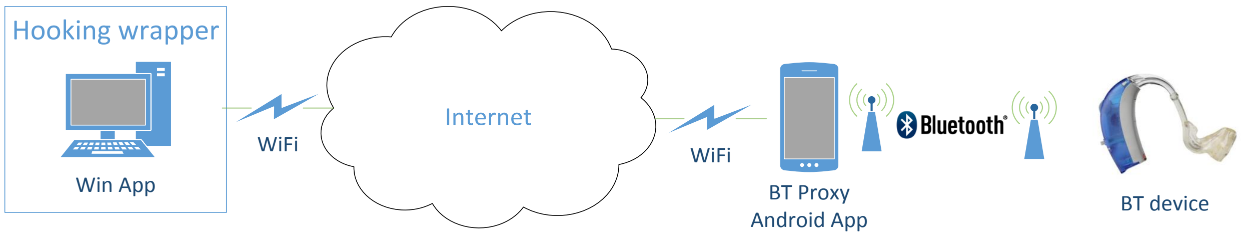


Figure 2 - BT proxy complete system

The system consists of two major building blocks:

- Hooking wrapper using Microsoft Detours
- BT Proxy Android Application

3 TECHNICAL BACKGROUND

3.1 FUNCTION/API HOOKING (OR: INTERCEPTION)

Function (and in particular API function) hooking is a technique that allows a user-defined function to be executed instead of the original function. The traditional way to implement this technique is by overwriting the address of the original function with the user-defined function address.

It is not quite easy to implement the API hooking in the manner described since we have to take into account the JMP addresses, memory allocation and so on. That is why Microsoft was kind enough to create a library that wraps up the entire flow into several easy to use functions, the Microsoft Detours Library.

3.2 MICROSOFT DETOURS

From the MS Detours overview:

> "Detours is a library for intercepting arbitrary Win32 binary functions on x86 machines. Interception code is applied dynamically at runtime. Detours replaces the first few instructions of the target function with an unconditional jump to the user-provided detour function. Instructions from the target function are placed in a trampoline. The address of the trampoline is placed in a target pointer. The detour function can either replace the target function, or extend its semantics by invoking the target function as a subroutine through the target pointer to the trampoline."

There are several methods that MS Detours offers. In this project we used the DLL injection functions that MS Detours provides.

3.2.1 DLL Injection

In order to inject a DLL that holds the user-defined functions, the user must create an executable application that calls the original executable with the DLL that he wants to inject.

The following code snip describes the injection of user defined functions implemented in a DLL (located at `DLLPath`) into the original application (located at `ExePath`). The injection is done using `DetourCreateProcessWithDllEx(...)` function.

```
// ================================================================
// injector.cpp
// ================================================================
int main(int argc, char *argv[])
{
	STARTUPINFO si;
	PROCESS_INFORMATION pi;
	char* ExePath = "..."; // original application path
	char* DllPath = "..."; // dll with user defined functions path

	ZeroMemory(&si, sizeof(si));
	ZeroMemory(&pi, sizeof(pi));
	si.cb = sizeof(si);
	si.dwFlags = STARTF_USESHOWWINDOW;
	si.wShowWindow = SW_SHOW;

	if (!DetourCreateProcessWithDllEx(NULL,  ExePath, NULL, NULL,
			TRUE, CREATE_DEFAULT_ERROR_MODE | CREATE_SUSPENDED,
			NULL, NULL, &si, &pi, DllPath, NULL))
	{
		return EXIT_FAIL; // Detour failed
	}

	ResumeThread(pi.hThread);
	WaitForSingleObject(pi.hProcess, INFINITE);

	CloseHandle(&si);
	CloseHandle(&pi);
	return EXIT_SUCCESS;
}
```

Alongside with the executable above, the following code snip describes how to hook a single API function, `MessageBoxW(...)`. The user-defined function, `MyMessageBox(...)` is implemented in the DLL, and its' signature has to remain exactly as the original function signature.

The `DetourAttach(ppPointer,pDetour)` function (called during DLL_PROCESS_ATTACH) attaches a detour (MyMessageBox) to the target function (pMessageBox). Same goes for the `DetourDetach(...)` (called during DLL_PROCESS_DETACH).

```
// ==================================================================
// dllmain.cpp
// ==================================================================
// original function
int (WINAPI *pMessageBox)(_In_opt_ HWND hWnd, _In_opt_ LPCTSTR
lpText,_In_opt_ LPCTSTR lpCaption, _In_ UINT uType) = MessageBoxW;

// user-defined function
int WINAPI MyMessageBox(_In_opt_ HWND hWnd, _In_opt_ LPCTSTR lpText,
_In_opt_ LPCTSTR lpCaption, _In_ UINT uType) {
            // User implementation goes here.
            // Usually the original function is called after the user
            // code is finished
            return pMessageBox(hWnd, lpText, lpCaption, uType);
}

BOOL WINAPI DllMain(HINSTANCE hinst, DWORD dwReason, LPVOID reserved)
{
      if (DetourIsHelperProcess()) {
            return TRUE;
      }

      if (dwReason == DLL_PROCESS_ATTACH) {
            DetourTransactionBegin();
            DetourUpdateThread(GetCurrentThread());
            DetourAttach(&(PVOID&)pMessageBox, MyMessageBox);
            DetourTransactionCommit();
      }
      else if (dwReason == DLL_PROCESS_DETACH) {
            DetourTransactionBegin();
            DetourUpdateThread(GetCurrentThread());
            DetourDetach(&(PVOID&)pMessageBox, MyMessageBox);
            DetourTransactionCommit();
      }
      return TRUE;
}
```

3.3 WINDOWS SOCKET API

Windows Socket API allows different protocols (e.g. TCP, BT, etc.) to be used with the same API functions. In a Server-Client system, a Server will open a socket using `socket()` and will wait for clients to connect using `accept()`, while the client will connect to the server using `connect()` function, and will probably send data using `send()` function.

Regardless of the protocol being used, data will be transmitted and received with the exact same functions, but with a different set of arguments.

4 ARCHITECTURE

4.1 HOOKING WRAPPER

4.1.1 Blocks Diagram

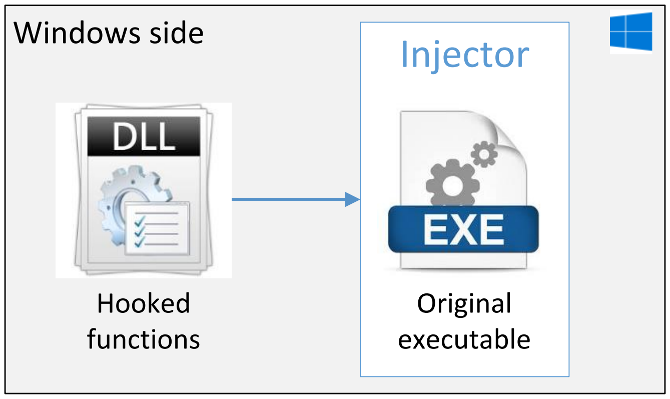


Figure 3 - Hooking wrapper (relies on Windows side) blocks diagram

4.1.2 Implementation

In the Windows Side we implemented an injector that injects a DLL containing the hooked WSA functions to the original executable.

4.1.3 Injector

The implementation of the injector is similar to the implementation described in DLL injection section.

4.1.4 Hooked WSA Functions

1. **MySocket**

```
SOCKET WSAAPI MySocket(
  _In_ int af,        // Address Family (INET/BTH/etc.)
  _In_ int type,
  _In_ int protocol
);
```

MySocket opens two sockets using the original WSA socket, a **TCP** socket and a **BT** socket. The reason we opened two sockets is that in the following connect, if the BT connect fails using the BT socket (due to distance limitation or other error), a connect with the TCP socket will be used.
MySocket returns the **BT** socket number, since the BT original application placed the call to Socket().

2. **MyConnect**

```
int MyConnect(
  _In_ SOCKET                s,      // The unconnected opened socket
  _In_ const struct sockaddr *name,  // A pointer to the sockaddr to which
                                     // the connection should be established
  _In_ int                   namelen// The length [bytes] of name
);
```

At first, MyConnect tries to connect to the opened BT socket. If it fails, MyConnect connects to the opened TCP socket using **the fixed IP and Port** of the BTProxy Android phone.
MyConnect returns the BT connect result if it succeeded, or the TCP connect result if the BT connect failed.

3. MySend

```
int MySend(
  _In_        SOCKET s,
  _In_ const char   *buf,
  _In_        int    len,
  _In_        int    flags
);
```

If the BT connect succeeded, the data (buf) will be sent to the BT socket using the original send WSA function. If the BT connect failed, MySend will use the TCP socket to send the data.
Note that buf, len and flags are used as they were received from the original BT application.
MySend returns the send WSA result.

4. MyClosesocket

```
int MyClosesocket(
  _In_ SOCKET s
);
```

MyClosesockets closes both the TCP and the BT opened sockets.

4.2 BLUETOOTH PROXY APPLICATION

4.2.1 Blocks Diagram

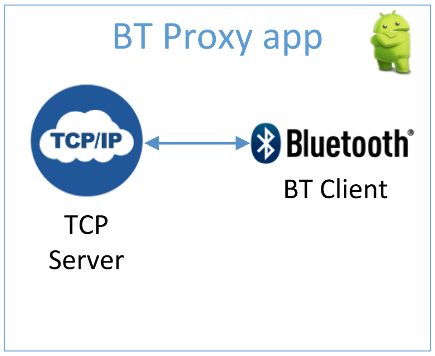


Figure 4 - BT Proxy app blocks diagram

4.2.2 Implementation

The BT Proxy app purpose is to act as a tube between the given Windows application and the BT device. The communication from the Hooking Wrapper is done by connecting as a client in the Windows side to a TCP server in the BT Proxy side.

After the TCP Server receives a TCP packet, it transmits it as a BT packet through the BT client. The packet will be received at the BT original application as it should function as a BT Server.

5 ISSUES WE CAME ACROSS DURING THE PROJECT

5.1 SOCKETS AND OS

During the Hooking development, we came across an interesting bug. We didn’t close the TCP socket that was opened in the hooked function, and since sockets are controlled by the OS, whenever a new TCP socket to the same server IP was opened, the OS actually used the already opened socket. The TCP server (relies in the Android BTProxyApp) kept waiting for a new TCP connection, that was never opened.

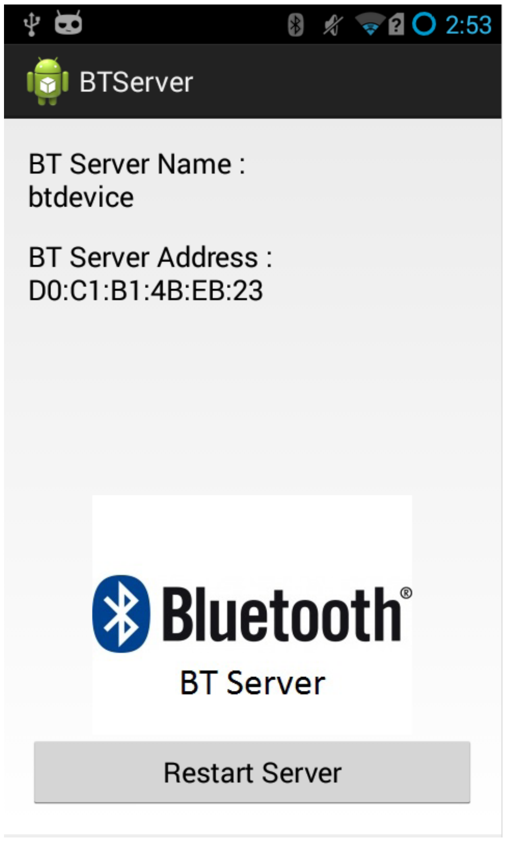
This bug made us realize how sockets are controlled by the OS.

6 RESULTS

To verify the fully flow implementation, we have built a "dummy" application that sends a String from the Windows side to the BT device.

The following screen shots show the verification environment:

1. The BT device is turned on

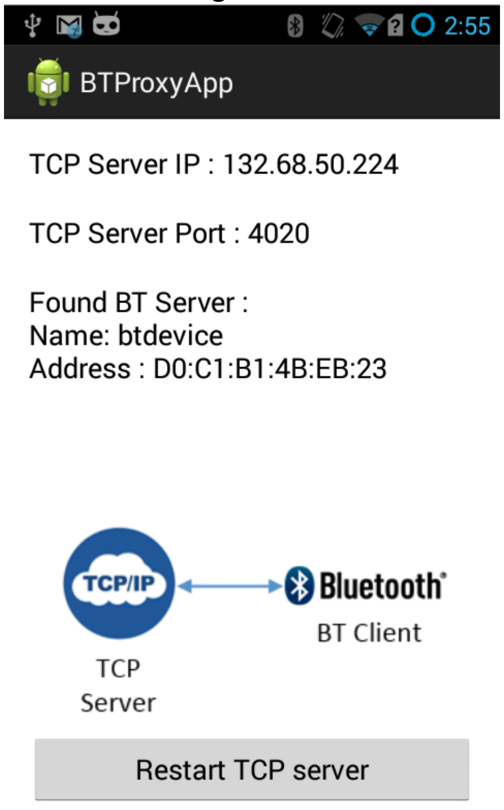


BTServer

BT Server Name :
btdevice

BT Server Address :
D0:C1:B1:4B:EB:23

2. The BT proxy is running, looking for the BT device according to its' MAC address



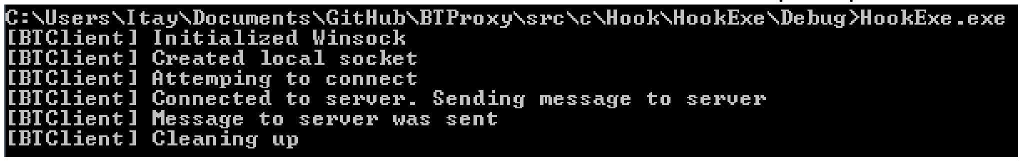
BTProxyApp

TCP Server IP : 132.68.50.224

TCP Server Port : 4020

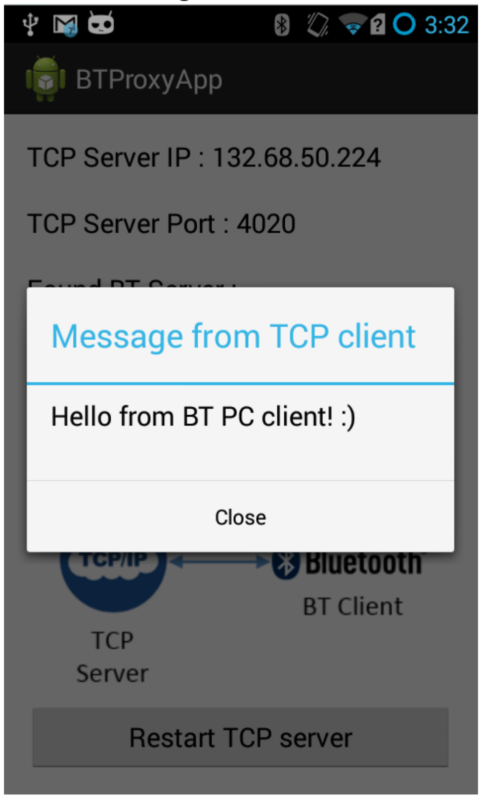
Found BT Server :
Name: btdevice
Address : D0:C1:B1:4B:EB:23

3. The Hook executable is called from the windows command prompt:



```
C:\Users\Itay\Documents\GitHub\BTProxy\src\c\Hook\HookExe\Debug>HookExe.exe
[BTClient] Initialized Winsock
[BTClient] Created local socket
[BTClient] Attemping to connect
[BTClient] Connected to server. Sending message to server
[BTClient] Message to server was sent
[BTClient] Cleaning up
```

4. The message that is sent from the Windows BT application is shown in the BT device:



BTProxyApp

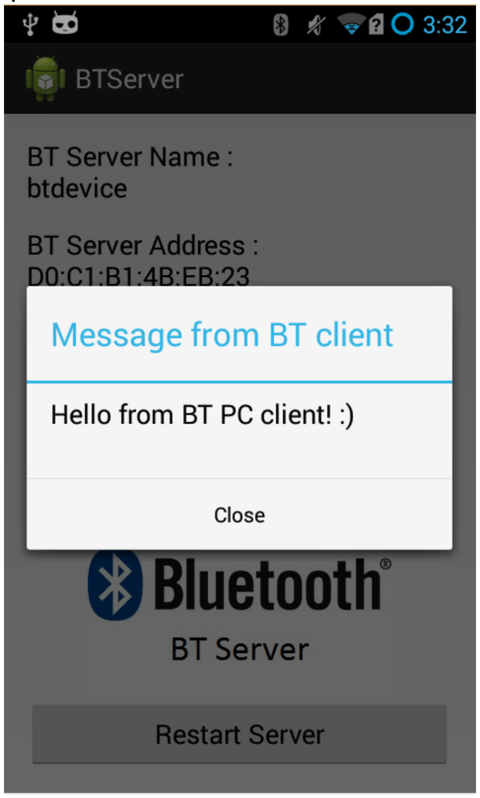
TCP Server IP : 132.68.50.224

TCP Server Port : 4020

Message from TCP client

Hello from BT PC client! :)

Close



BTServer

BT Server Name : btdevice

BT Server Address : D0:C1:B1:4B:EB:23

Message from BT client

Hello from BT PC client! :)

Close

7 CONCLUSIONS

7.1 LEARN BY “GETTING DIRTY”

Especially when it comes to coding, this project showed us that getting your hands dirty is the best way to learn. That is why we first tried detouring a simple Windows API function, the MessageBox, as explained in the DLL injection technical background.

7.2 HAVE A PLAN

One of the most important lessons we’ve learned is that everything has to be planned, and one cannot be too enthusiastic to start coding.

First, the system design has to be planned. The design must be divided into logical blocks and the interfaces have to be defined.

After the architecture is figured out, you have to plan the schedule. Estimations of how long will it take to both write and more importantly verify every logical block have to be made. One must also consider the integration procedure that might take even long than imagined.

7.3 DIVIDE AND CONQUER

The very basics of good engineering is to divide the big problem into several small ones that you can maintain and solve easily. Once every small problem is solved and contained separately, integrating everything should become easier.

8 REFERENCES

8.1 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/

8.2 ANDROID

[3] Android Reference Guide,

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

8.3 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