using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Diagnostics;

using System.Linq;

using System.ServiceProcess;

using System.Text;

using System.Threading.Tasks;

using System.Net.Sockets; //used for TcpClient class

using System.Net; //used for IPEndPoint class

using System.Threading;

using System.IO;

using KeyloggerCommunications;

using System.ServiceModel;

using System.Net.NetworkInformation;

using System.Configuration;

namespace WindowsServiceTracker

{

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* This class is where the majority of the work of the service is done.

\* Currently the only functionality not done exclusively in this service

\* is the keylogging. Keylogging is done in the WTKL project in the

\* SystemTrayKeylogger.cs file.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public partial class Tracker : ServiceBase, KeyloggerCommInterface

{

//Constants

//127.0.0.1 = 0x0100007F because of network byte order

public const byte KEYLOG\_ON = 134;

public const byte KEYLOG\_OFF = 135;

public const byte TRACE\_ROUTE = 131;

public const byte KEYLOG = 130;

public const byte NOT\_STOLEN = 133;

public const byte STOLEN = 132;

public const byte NO\_OP = 255;

private const string ERROR\_LOG\_NAME = "TrackerErrorLog";

private const string ERROR\_LOG\_MACHINE = "TrackerComputer";

private const string ERROR\_LOG\_SOURCE = "WindowsServiceTracker";

public const int CHECKIN\_WAIT\_TIME = 60000; //todo increase to ~5min after debugging

private const string REG\_PATH = "Software\\SOT\\WindowsServiceTracker";

private const string REG\_FIELD\_STOLEN = "Stolen";

private const string REG\_FIELD\_MAC = "MAC";

private const string REG\_VAL\_FALSE = "false";

private const string REG\_VAL\_TRUE = "true";

private const string ID\_FILE = "ID.txt";

//Variables

private String ipAddressString = "127.0.0.1";

private String domainName = "";

private int port = 10011;

private IPAddress ipAddress = new IPAddress(0x0100007F);// = 0x0100007F; //default to local host

private volatile IPAddress[] ipList;

private ChannelFactory<KeyloggerCommInterface> pipeFactory = new ChannelFactory<KeyloggerCommInterface>(

new NetNamedPipeBinding(), new EndpointAddress("net.pipe://localhost/PipeKeylogger"));

private KeyloggerCommInterface pipeProxy;

private Thread tcpThread;

private volatile Thread retryStartKeylogger;

private volatile bool connectionKeepAlive = true;

private volatile bool tcpKeepAlive = true;

private volatile string macAddress = null;

private volatile String keyLogFilePath;

// Variables in this block are intended to be used only with the thread

// maintaining the tcp connection. They are not thread safe and should

// not be used by other threads without making them volatile.

private IPEndPoint tcpIpPort;

private NetworkStream tcpStream;

private TcpClient tcp;

private static bool reportedStolen = false;

private int pingFrequency = 30000;

private int keylogFrequency = 20000; //todo up time on final release

private Stopwatch pingStopwatch = new Stopwatch();

private Stopwatch keylogStopwatch = new Stopwatch();

private int currentIP = 0;

/\* Constructor for the service. Currently only creates an event log source

\* that is used to output errors with in the windows event logs.

\*/

public Tracker()

{

InitializeComponent();

//Creates the error log source if it doesn't already exist

if (!EventLog.SourceExists(ERROR\_LOG\_SOURCE))

{

EventLog.CreateEventSource(ERROR\_LOG\_SOURCE, ERROR\_LOG\_NAME);

}

}

/\* This method is the first method to be ran when the service starts running. For pretty

\* much all intents and purposes this is simply the main method.

\*/

protected override void OnStart(string[] args)

{

//Use the following line to launch an instance of visual studio to debug

//with. You can also just run the service and then attach the debugger

//to the process.

//System.Diagnostics.Debugger.Launch();

//Keep the service running for 15 seconds

Thread.Sleep(30000); //todo remove for final release

//Sets the current directory to where the WindowsServiceTracker.exe is located rather

//than some Windows folder that I couldn't seem to locate

System.IO.Directory.SetCurrentDirectory(System.AppDomain.CurrentDomain.BaseDirectory);

// get whether the laptop is reported stolen from the registry

reportedStolen = getStolenRegVal();

// try to read settings before using default values

try

{

ipAddressString = Properties.Settings.Default.ServerIP;

port = Convert.ToInt32(Properties.Settings.Default.ServerPort);

domainName = Properties.Settings.Default.ServerDomain;

}

catch (ConfigurationException e)

{}

catch (Exception e)

{}

// get list of IPs from domain name

if (domainName.Length > 0)

{

try

{

ipList = Dns.GetHostAddresses(domainName);

}

catch

{ }

}

//convert string IP to long and use if domain name failed

if (ipList == null || ipList.Length == 0)

{

try

{

ipAddress = IPAddress.Parse(ipAddressString);

ipList = new IPAddress[1];

ipList[0] = ipAddress;

}

catch (Exception)

{ }

}

CreateOpenPipe();

keyLogFilePath = GetKeylogFilePath();

if (reportedStolen)

{

StartKeylogger();

}

macAddress = getMacAddress();

createIDFile();

//StartKeylogger(); //todo remove after debugging

tcpThread = new Thread(this.IpConnectionThread);

tcpThread.Start();

}

/\*This method runs immediately before the service stops and shuts down. So all writing to

\* config/settings files and closing connections should be done here.

\*/

protected override void OnStop()

{

StopKeylogger();

tcpDisconnect();

tcpKeepAlive = false;

connectionKeepAlive = false;

if (tcpThread != null && tcpThread.IsAlive)

{

tcpThread.Join();

}

}

/\*Creates the pipe over which keylogger functions can be called. Functions are called

\* using pipeProxy.FunctionName();

\*/

private void CreateOpenPipe()

{

pipeProxy = pipeFactory.CreateChannel();

}

//Starts the keylogger

public bool StartKeylogger()

{

if (CheckIfRunning())

{

return pipeProxy.StartKeylogger();

}

if (retryStartKeylogger == null || !retryStartKeylogger.IsAlive/\*retryStartKeylogger.ThreadState != System.Threading.ThreadState.Running\*/)

{

//WriteEventLogEntry("new keylogger thread");

retryStartKeylogger = new Thread(this.retryStartKeyloggerThread);

retryStartKeylogger.Start();

}

return false;

}

//Stops the keylogger

public bool StopKeylogger()

{

if (CheckIfRunning())

{

return pipeProxy.StopKeylogger();

}

return false;

}

/\*

\* This method is used as a thread which is launched if the keylogger is called to start

\* when the keylogger program is not currently running.

\*/

private void retryStartKeyloggerThread()

{

while (!StartKeylogger() && reportedStolen)

{

Thread.Sleep(60000);

}

}

/\*

\* Get location of keylog file

\*/

public String GetKeylogFilePath()

{

if (CheckIfRunning())

{

return pipeProxy.GetKeylogFilePath();

}

return String.Empty;

}

//Checks to see if the keylogger program is running

public bool CheckIfRunning()

{

try

{

return pipeProxy.CheckIfRunning();

}

catch (Exception e)

{

CreateOpenPipe();

return false;

}

}

/\* This method writes to the windows event logs with an "Information" event

\* type. All you have to do for it to work is call the method with a string

\* as the argument and it will write an event for you. Useful for error/bug

\* output

\*/

private void WriteEventLogEntry(string eventLogInput)

{

//Write to the Windows Event Logs, shows up under Windows Logs --> Application

EventLog.WriteEntry(ERROR\_LOG\_SOURCE, eventLogInput, EventLogEntryType.Information);

}

/\* Tries to connect to the server and sleeps for a given amount of time

\* if the server reports it is not stolen.

\*/

private void IpConnectionThread()

{

while (connectionKeepAlive)

{

MaintainServerConnection();

if (!reportedStolen && connectionKeepAlive)

{

long timeWaited = 0;

while (connectionKeepAlive && timeWaited < CHECKIN\_WAIT\_TIME)

{

Thread.Sleep(5000);

timeWaited += 5000;

}

}

}

}

/\* This method is used to create a thread that will constantly try to connect

\* to the server while it is active. When a connection is established, the

\* MAC address is immediately sent to the server and it waits for commands.

\*/

private void MaintainServerConnection()

{

tcpKeepAlive = true;

pingStopwatch.Restart();

keylogStopwatch.Restart();

int maxwaitBetweenConnects = 60;

int waitToConnect = 0;

int bufferSize = 1;

byte[] buffer = new byte[bufferSize];

while (tcpKeepAlive && connectionKeepAlive)

{

if (tcp == null || !tcp.Connected)

{

try

{

bool connected = tcpConnect();

if (connected)

{

waitToConnect = 0;

getTcpStream();

SendStdMsg(macAddress, true);

}

else

{

Thread.Sleep(waitToConnect \* 1000);

if (waitToConnect < maxwaitBetweenConnects)

{

waitToConnect += 5;

}

}

}

catch (Exception)

{

Thread.Sleep(waitToConnect \* 1000);

if (waitToConnect < maxwaitBetweenConnects)

{

waitToConnect += 5;

}

}

}

else

{

if (tcpStream == null || !tcpStream.CanRead)

{

getTcpStream();

}

else

{

try

{

int bytesRead;

bytesRead = tcpStream.Read(buffer, 0, bufferSize);

if (bytesRead == 0) //if 0, connection died. ping works to keep sending bytes to connection status

{

tcp = null;

tcpStream = null;

}

else {

switch (buffer[0])

{

case KEYLOG\_ON:

StartKeylogger();

break;

case KEYLOG\_OFF:

StopKeylogger();

break;

case TRACE\_ROUTE:

SendStdMsg(TRACE\_ROUTE, traceRoute(ipAddressString), true);

break;

case KEYLOG:

sendKeylog();

break;

case NOT\_STOLEN:

tcpKeepAlive = false;

reportedStolen = false;

writeStolenRegVal(reportedStolen);

StopKeylogger();

break;

case STOLEN:

reportedStolen = true;

writeStolenRegVal(reportedStolen);

StartKeylogger();

break;

default:

break;

}

}

}

catch (Exception)

{}

//ping server

if (pingStopwatch.ElapsedMilliseconds > pingFrequency)

{

SendStdMsg(NO\_OP, "", true);

pingStopwatch.Restart();

}

//send keylog

if (reportedStolen && keylogStopwatch.ElapsedMilliseconds > keylogFrequency)

{

sendKeylog();

keylogStopwatch.Restart();

}

}

}

}

tcp.Close();

}

/\* Creates a new connection with the server.

\*/

private bool tcpConnect()

{

int ipAddressesTried = 0;

do

{

tcpIpPort = new IPEndPoint(ipList[currentIP], port);

tcp = new TcpClient();

try {

tcp.Connect(tcpIpPort);

}

catch{}

if (!tcp.Connected)

{

currentIP++;

ipAddressesTried++;

currentIP %= ipList.Length;

return false;

}

} while (!tcp.Connected && ipAddressesTried < ipList.Length);

return true;

}

/\* Closes the TCP connection with the server

\*/

private bool tcpDisconnect()

{

try

{

tcpStream.Close();

}

catch (NullReferenceException)

{ }

return true;

}

/\* Attempts to get get the NetworkStream from the tcp connection and

\* assign it to the tcpStream variable.

\*/

private bool getTcpStream()

{

try

{

tcpStream = tcp.GetStream();

tcpStream.ReadTimeout = 5000;

}

catch (InvalidOperationException)

{

return false;

}

return true;

}

/\* Writes a message to the tcp connection. The format is

\* <opcode><message><newline>

\* A msg that is null or empty will leave it out.

\* If newLine is false, the newline will not be included at the end of the msg.

\*/

private bool SendStdMsg(bool includeOpcode, byte opcode, byte[] msg, bool newLine)

{

int msgSize = 0;

int offset = 0;

byte[] newLineBytes = Encoding.UTF8.GetBytes("\n");

byte[] combinedMsg;

try

{

//opcode

if (includeOpcode)

{

msgSize += 1;

}

if (msg != null && msg.Length != 0)

{

msgSize += msg.Length;

}

if (newLine == true)

{

msgSize += newLineBytes.Length;

}

// incase there is nothing to send

if (msgSize == 0)

{

return true;

}

combinedMsg = new byte[msgSize];

// assemble message into single array to be sent as a single unit

if (includeOpcode)

{

combinedMsg[offset] = opcode;

offset += 1;

}

if (msg != null && msg.Length != 0)

{

msg.CopyTo(combinedMsg, offset);

offset += msg.Length;

}

if (newLine == true)

{

newLineBytes.CopyTo(combinedMsg, offset);

offset += newLineBytes.Length;

}

tcpStream.Write(combinedMsg, 0, combinedMsg.Length);

}

catch (Exception)

{

return false;

}

return true;

}

/\* Writes a message to the tcp connection. The format is

\* <opcode><message><newline>

\* A msg that is null or empty will leave it out.

\* If newLine is false, the newline will not be included at the end of the msg.

\*/

private bool SendStdMsg(byte opcode, byte[] msg, bool newLine) // todo first message to fail isn't detected, maybe send empty message first?

{

return SendStdMsg(true, opcode, msg, newLine);

}

/\* Convenience method to send a string message with optional newline

\* on the end. No opcode is included at the beginning of the message.

\*/

private bool SendStdMsg(String msg, bool newLine)

{

byte[] byteMsg = null;

if (msg == null || msg.Length == 0)

{

return SendStdMsg(false, NO\_OP, byteMsg, newLine);

}

else

{

byteMsg = Encoding.UTF8.GetBytes(msg);

return SendStdMsg(false, NO\_OP, byteMsg, newLine);

}

}

/\* Convenience method that converts a string message to a byte[] array

\* before using sendStdMsg(byte opcode, byte[] msg) to send it.

\*/

private bool SendStdMsg(byte opcode, String msg, bool newLine)

{

byte[] byteMsg = null;

if (msg == null || msg.Length == 0)

{

return SendStdMsg(opcode, byteMsg, newLine);

}

else

{

byteMsg = Encoding.UTF8.GetBytes(msg);

return SendStdMsg(opcode, byteMsg, newLine);

}

}

/\* Performs a traceroute to the given address. Returns a string of

\* IP addresses delimited by '~'

\*/

private String traceRoute(String address)

{

String ipString = String.Empty;

IEnumerable<IPAddress> ipList = IP.getTraceRoute(address);

foreach (IPAddress nodeAddress in ipList)

{

ipString += nodeAddress + "~";

}

try

{

ipString = ipString.Remove(ipString.Length - 1);

}

catch (ArgumentOutOfRangeException)

{

}

return ipString;

}

/\* sends the contents of the keylog file to the server

\* and deletes it. If unable to finish, it should store

\* the remaining contents in a file file and attempt

\* to send it first next time before sending the active

\* file.

\*/

private bool sendKeylog()

{

bool successfulFileOpen = true;

StreamReader log = null;

String tempFile = "tempFile.txt";

String storedFile = "storedFile.txt";

int readSize = 1024;

char[] buffer = new char[readSize];

int bytesRead;

byte[] msg;

bool storedFileExists = false;

bool sentAllContent = false;

// see if an unsent file exists

try

{

File.Move(storedFile, tempFile);

storedFileExists = true;

}

catch (Exception)

{ }

// if there is not an unsent file, grab the active one

if (!storedFileExists)

{

try

{

File.Move(keyLogFilePath, tempFile);

}

catch (Exception)

{

return true;

}

}

try

{

log = new StreamReader(tempFile);

// the nested try block is so that when there is no keylog file,

// it still sends the opcode and newline char

// previous comment changed. not sending messages if there

// is nothing to send

//todo revisit the nested try blocks reguarding the change

try

{

bool lastSendSuccessful = true;

while (!log.EndOfStream && lastSendSuccessful)

{

bytesRead = log.Read(buffer, 0, readSize);

msg = Encoding.UTF8.GetBytes(buffer, 0, bytesRead);

if (msg.Length > 0)

{

lastSendSuccessful = SendStdMsg(KEYLOG, msg, true);

Thread.Sleep(1000);

}

else // change noted above

{

lastSendSuccessful = true;

}

}

sentAllContent = lastSendSuccessful;

}

catch (Exception)

{ }

if (!sentAllContent)

{

try

{

StreamWriter store = new StreamWriter(storedFile);

store.Write(buffer, 0, buffer.Length);

while (!log.EndOfStream)

{

store.Write(log.Read());

}

store.Close();

}

catch (Exception)

{ }

}

}

catch (Exception)

{

successfulFileOpen = false;

}

try

{

log.Close();

File.Delete(tempFile);

if (successfulFileOpen && sentAllContent)

{

// if we sent an old file, send new one now

if (storedFileExists && sentAllContent)

{

return sendKeylog(); //todo recursion on huge keylog could be a problem

}

}

}

catch (Exception)

{

//return false;

}

return sentAllContent;

}

/\* Retrieves the RegistreyKey and returns it. Contains persistent information for the service.

\*/

private Microsoft.Win32.RegistryKey getStolenRegKey()

{

Microsoft.Win32.RegistryKey stolenKey = Microsoft.Win32.Registry.LocalMachine.OpenSubKey(REG\_PATH, true);

if (stolenKey == null)

{

stolenKey = Microsoft.Win32.Registry.LocalMachine.CreateSubKey(REG\_PATH);

}

return stolenKey;

}

/\* Gets the value from the registry that indicates whether the device is stolen.

\*/

private bool getStolenRegVal()

{

string stolen = "";

Microsoft.Win32.RegistryKey stolenKey = getStolenRegKey();

try

{

stolen = stolenKey.GetValue(REG\_FIELD\_STOLEN).ToString();

} catch (NullReferenceException)

{

stolenKey.SetValue(REG\_FIELD\_STOLEN, REG\_VAL\_FALSE);

}

if (stolen == REG\_VAL\_TRUE)

{

return true;

}

else

{

return false;

}

}

/\* Writes either true or false to the registry to store the stolen state

\* of the device for persistence through turning on and off.

\*/

private void writeStolenRegVal(bool stolen)

{

Microsoft.Win32.RegistryKey stolenKey = getStolenRegKey();

if (stolen)

{

stolenKey.SetValue(REG\_FIELD\_STOLEN, REG\_VAL\_TRUE);

}

else

{

stolenKey.SetValue(REG\_FIELD\_STOLEN, REG\_VAL\_FALSE);

}

}

/\* Attempts to get the MAC address from the registry. If unsuccesful, gets it from

\* the hardware and stores that in the registry;

\*/

private string getMacAddress()

{

string mac = "";

Microsoft.Win32.RegistryKey stolenKey = getStolenRegKey();

try

{

mac = stolenKey.GetValue(REG\_FIELD\_MAC).ToString();

}

catch (NullReferenceException)

{

stolenKey.SetValue(REG\_FIELD\_STOLEN, "");

}

if (mac == "")

{

mac = getMacAddressFromHardware();

stolenKey.SetValue(REG\_FIELD\_MAC, mac);

return mac;

}

else

{

return mac;

}

}

/\* Gets the MAC address of the laptop. The method loops through all existing network

\* adapters looking for an ethernet adapter, if one is found then it is immediately

\* returned. If not, then it looks for the first WiFi adapter in the list. If it finds

\* a wifi adapter it will continue looping to prioritize for ethernet. If neither WiFi

\* nor Ethernet is found then the MAC address of the active adapter is used.

\*/

private string getMacAddressFromHardware()

{

string mac = string.Empty;

bool keepUnlessEthernet = false;

foreach (NetworkInterface nic in NetworkInterface.GetAllNetworkInterfaces())

{

if (nic.NetworkInterfaceType == NetworkInterfaceType.Ethernet)

{

return nic.GetPhysicalAddress().ToString();

}

else if (!keepUnlessEthernet && nic.NetworkInterfaceType == NetworkInterfaceType.Wireless80211)

{

mac = nic.GetPhysicalAddress().ToString();

keepUnlessEthernet = true;

}

else if (mac == string.Empty && nic.OperationalStatus == OperationalStatus.Up)

{

mac = nic.GetPhysicalAddress().ToString();

}

}

return mac;

}

/\* Creates a file with the ID (MAC adress) that the service is using

\* to identify itself

\*/

private void createIDFile()

{

string tempID = "tempID";

try

{

StreamWriter idOut = new StreamWriter(tempID);

idOut.Write(macAddress);

idOut.Close();

File.Move(tempID, ID\_FILE);

}

catch

{

try

{

File.Delete(tempID);

}

catch

{ }

}

}

}

}