



The Windows Phone Freakshow

Windows Phone App Security for Builders and Breakers

Hack in The Box Conference, Amsterdam 2015 - Research Whitepaper v. 2.0

Luca De Fulgentis – luca@securenetwork.it

✉ info@securenetwork.it

☎ (+39) 02 9177 3041



Italia

PoliHub

Via Giovanni Durando, 39
20158 Milano



United Kingdom

New Bridge Street House
30-34, New Bridge Street
EC4V 6BJ, London

Table of Contents

INTRODUCTION	3
M1 – WEAK SERVER SIDE CONTROLS	4
M2 – INSECURE DATA STORAGE	5
M3 – INSUFFICIENT TRANSPORT LAYER SECURITY	9
M4 – UNINTENDED DATA LEAKAGE	13
M5 – POOR AUTHORIZATION AND AUTHENTICATION	15
M6 – BROKEN CRYPTOGRAPHY	18
M7 – CLIENT SIDE INJECTION	21
M8 – SECURITY DECISIONS VIA UNTRUSTED INPUTS	23
M9 – IMPROPER SESSION HANDLING	26
M10 – LACK OF BINARY PROTECTIONS	28
CONCLUSIONS	29
REFERENCES	30

Introduction

In 2014 we collected samples of insecure code while analyzing 60+ apps targeting Microsoft's Windows Phone platform. The initial set was mostly composed by applications developed with the Silverlight 8.x technology, and was characterized by a 30% of mobile banking apps. Each identified issue has been mapped to the corresponding **Mobile Top Ten Risk** for 2014 entry (Figure 1), and statistics on the initial study have been shared with the OWASP Mobile Project for the definition of the MTT for 2015.

Later we decided to extend our research, developing an automated script that allowed downloading of 160+ AppXs from the US and Italian versions of the Windows Phone Store: the new set of applications has also been analyzed, thus covering security aspects related to apps developed with the Windows Runtime (WinRT) technology.

The so obtained categorized vulnerabilities – our *freaks* – have been elaborated in a second stage of the study, in order to define a *catalog* of insecure API usage, that represents the most complete and accurate public resource on the topic.



Figure 1 – The OWASP Mobile Top 10 Risks for 2014.

M1 – Weak Server Side Controls

This risk is referring to *server-side* security issues and addresses a mobile-platform *agnostic* category of vulnerabilities. Nowadays, back-end systems security can be properly assessed using standard and consolidated testing methodologies (e.g., **OWASP Testing Guide v.4**), and using resources such as the **OWASP Top 10 Web Security Risks** as a driver while exploring the targets security.

Server-side security should always be included in the assessment scope when performing a mobile application penetration test, as a malicious user may not only *directly* attack back-end systems (e.g., web services) to steal confidential data – that, in most of the cases, represent mobile users' data – but also indirectly attack the mobile app while abusing exposed server-side functionalities. A hacked back-end could be used as a *proxy* to attack the user's app, in order to extract locally stored data or inducing the victim into revealing confidential information, combining technical issues with Social Engineering techniques.

During our mobile application penetration testing projects, we noticed that the most relevant security issues involving back-end systems were *logical flaws*, often involving application states "evolution" and *input validation issues*, such as SQL Injection and Cross-Site Scripting (XSS).

It also should be noted that mobile apps source code analysis may reveal a series of details concerning the back-end systems, such as hardcoded URLs pointing to Internet-facing testing or pre-production environments, hardcoded (and "evergreen") session cookies or valid testing accounting credentials: these information may represent precious knowledge for attackers and should never be hardcoded into application code.

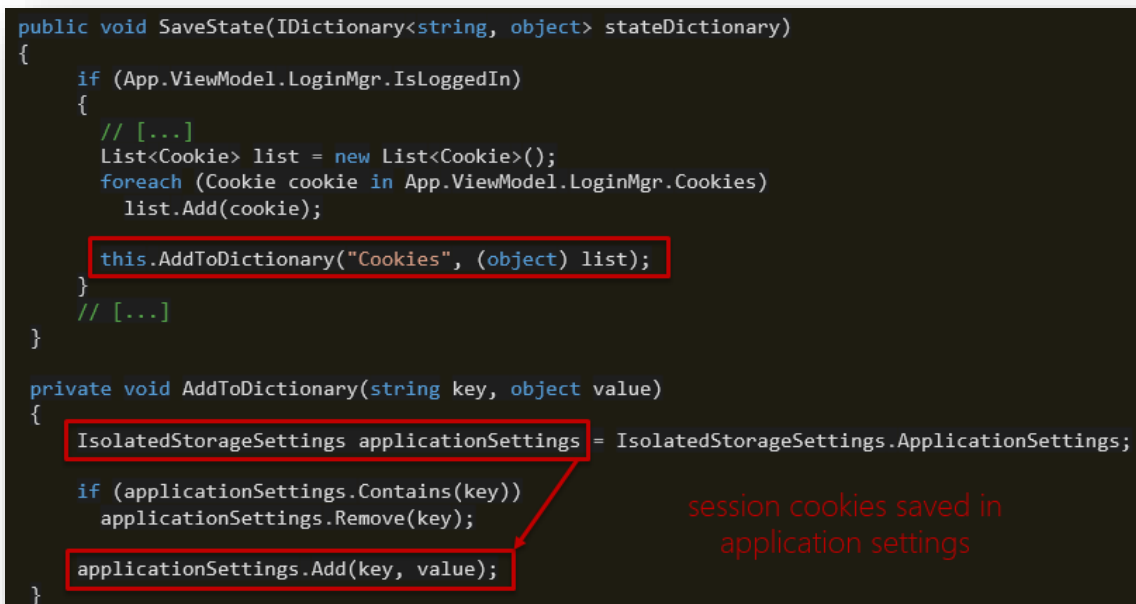
M2 – Insecure Data Storage

The second risk refers to applications that store sensitive or confidential data on devices, without applying any proper encryption techniques. The risk focuses on data confidentiality violation in case of a malicious app capable to access *clear-text* files, or physical attacks performed by users holding a privileged access to the target device file system and the stored information.

A secure mobile app should protect its data, especially if it is required storing:

- Session cookies (Figure 2);
- Account credentials (Figure 3);
- Authentication or authorization tokens;
- User's personal information – e.g., user's home and email addresses;
- Any kind other of sensitive, confidential or private data – e.g., credit card data.

Modern mobile platforms implements built-in disk encryption technologies, but these features do not block attacks at *runtime* where, for example, a malicious app or user hold a privileged access to the device file system that can be abused to steal unencrypted data.



```
public void SaveState(IDictionary<string, object> stateDictionary)
{
    if (App.ViewModel.LoginMgr.IsLoggedIn)
    {
        // [...]
        List<Cookie> list = new List<Cookie>();
        foreach (Cookie cookie in App.ViewModel.LoginMgr.Cookies)
            list.Add(cookie);

        this.AddToDictionary("Cookies", (object) list);
    }
    // [...]
}

private void AddToDictionary(string key, object value)
{
    IsolatedStorageSettings applicationSettings = IsolatedStorageSettings.ApplicationSettings;

    if (applicationSettings.Contains(key))
        applicationSettings.Remove(key);

    applicationSettings.Add(key, value);
}
```

The code snippet illustrates the insecure storage of session cookies. In the `SaveState` method, a list of cookies is created and added to a dictionary with the key "Cookies". This dictionary is then added to the application's settings using `this.AddToDictionary("Cookies", (object) list);`. The `AddToDictionary` method uses `IsolatedStorageSettings.ApplicationSettings` to store the data. A red box highlights the line `applicationSettings.Add(key, value);`, and a red arrow points to it from the text "session cookies saved in application settings".

Figure 2 – Insecure storage of session cookies in the application's sandbox.

The major mobile platforms also implement strong application security models and introduced application *sandboxing*, in order to prevent malicious apps to access other apps local files. It should be noted that sandboxing mechanisms could be defeated by exploiting system's vulnerabilities or *jailbreaking* a device. For these reasons, local data encryption represents a crucial security aspect for mobile apps.

```

private async void DoLogin()
{
    bool? isChecked = this.checkBoxRicordami.IsChecked;
    if ((!isChecked.GetValueOrDefault() ? 0 : (isChecked.HasValue ? 1 : 0)) != 0)
    {
        this.saveCredentials();
    }
    // [...]
    private void saveCredentials()
    {
        if (!(this.textBlockUsername.Text != "") || !(this.textBlockPassword.Password != ""))
            return;

        this.storageSettingsRememberMe.Remove("Username");
        this.storageSettingsRememberMe.Remove("Password");
        this.storageSettingsRememberMe.Remove("isChecked");

        this.storageSettingsRememberMe.Add("Username", this.textBlockUsername.Text);
        this.storageSettingsRememberMe.Add("Password", this.textBlockPassword.Password);
        this.storageSettingsRememberMe.Add("isChecked", true);
        this.storageSettingsRememberMe.Save();
    }
}

```

credentials saved in application setting file

Figure 3 - Insecure storage of account credentials in the app's application settings.

Starting from Windows Phone 8, Microsoft's mobile platform supports BitLocker disk encryption technology (AES 128), which is disabled by default and can only be activated with the Exchange ActiveSync policy **RequiredDeviceEncryption** or MDM policies, thus further increasing the importance of data encryption for this specific platform.

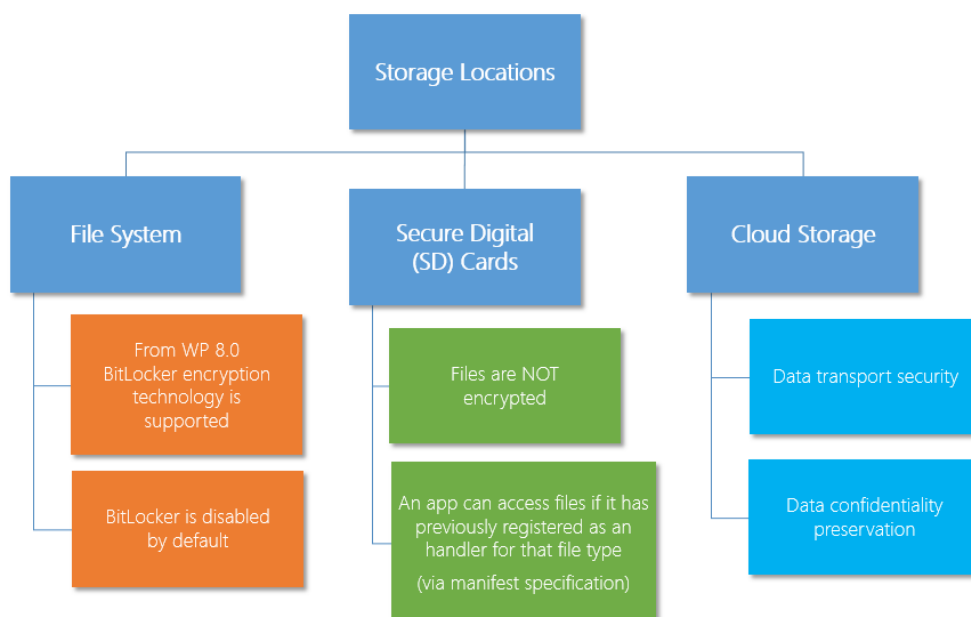


Figure 4 – Storage locations that could be adopted by Windows Phone apps.

Figure 4 summarizes the different storage locations where a Windows Phone app may save its data, while the following tables - divided based on the development technology – detail, for each area, the physical path on the device and the properties used within application source code to refer to the corresponding location.

Locations	Windows Runtime Apps
Local data store	ApplicationData.Current.LocalFolder - URI - ms-appdata:///local/ C:\Data\Users\DefApps\APPDATA\Local\Packages\%packageName%\LocalState
Roaming data store	ApplicationData.Current.RoamingFolder - URI - ms-appdata:///roaming/ C:\Data\Users\DefApps\APPDATA\Local\Packages\%packageName%\RoamingState
Temporary data store	ApplicationData.Current.TemporaryFolder - URI - ms-appdata:///temporary/ C:\Data\Users\DefApps\APPDATA\Local\Packages\%packageName%\TempState
Package installation	Windows.ApplicationModel.Package.Current.InstalledLocation URI: ms-appx:// and ms-appx-web:// C:\Data\SharedData\PhoneTools\AppxLayouts\{GUID}\
Cache data store	ApplicationData.Current.LocalCacheFolder C:\Data\Users\DefApps\APPDATA\Local\Packages\%packageName%\LocalCache
Media Library	KnownFolders.MusicLibrary, KnownFolders.CameraRoll, KnownFolders.PicturesLibrary, KnownFolders.VideosLibrary
SD Card	KnownFolders.RemovableDevices
Local Settings	Windows.Storage.ApplicationData.Current.LocalSettings
Roaming Settings	Windows.Storage.ApplicationData.Current.RoamingSettings
Local and Roaming Settings are saved in C:\Data\Users\DefApps\APPDATA\Local\Packages\%packageName%\Settings\settings.dat, which is a Windows NT registry file (REGF) - and NOT encrypted	

Locations	Silverlight Windows Phone Apps
Application local folder	C:\Data\Users\DefApps\APPDATA\{GUID}\Local\
Application Settings	IsolatedStorageSettings.ApplicationSettings C:\Data\Users\DefApps\APPDATA\{GUID}\Local__ApplicationSetting
Package installation	Windows.ApplicationModel.Package.Current.InstalledLocation C:\Data\Programs\{GUID}\Install\
Cached data	C:\Data\Users\DefApps\APPDATA\{GUID}\INetCache\
Cookies	C:\Data\Users\DefApps\APPDATA\{GUID}\INetCookies\
SD Card	(read only)

The following table could be used as a driver to identify specific methods and properties that refer to the creation or opening of files for several storage locations: it can be adopted to locate code segments capable of storing data, and so evaluate the existence of insecure storage for confidential information.

Locations	Classes, Methods and Properties
Local folders	StorageFolder.CreateFileAsync() StorageFolder.GetFileAsync() StorageFolder.GetFilesAsync() StorageFile.OpenReadAsync() StorageFile.OpenAsync() StorageFile.GetFileFromApplicationUriAsync() StorageFile.GetFileFromPathAsync()
	IsolatedStorageFile.CreateFile() IsolatedStorageFile.OpenFile()
Application or Roaming Settings	IsolatedStorageSettings.ApplicationSettings – property ApplicationData.LocalSettings – property ApplicationData.RoamingSettings – property
SD Card (WP 8.1 only)	KnownFolders.RemovableDevices returns a StorageFolder object that can be sequentially used to read/write data from the SD card
Local database	Identify objects that inherit from System.Data.Linq.DataContext Verify the existence of reserved data stored in the local .sdf file

It should be noted that data stored in local databases, reside in clear-text within the application's sandbox, in files with the ".sdf" extension. These databases should be properly protected by encrypting stored data using OS-provided encryption mechanism, such as the **Data Protection API (DPAPI)** – see M6 for more details.

As a final note, developers should adopt the **Windows.Security.Credentials.PasswordVault** mechanism for sensitive data storage, such as account credentials.

M3 – Insufficient Transport Layer Security

The M3 risk refers to confidentiality and integrity concerning the app-to-endpoint data transmission. Common issues related to transport layer security are:

- **Communication over an unencrypted channel:** clear-text communication (e.g., via HTTP protocol) can be intercepted by malicious users. Moreover, if an attacker is capable to perform a *Man in the Middle* (MitM) attack, he/she can manipulate the traffic sent back to the app, thus potentially triggering additional vulnerabilities, such as *client-side injections* (Figure 5, Figure 6 and Figure 7);
- **Communication over a poorly encrypted channel:** a poorly encrypted channel is based on weak cryptographic algorithms. In these conditions an attacker may decrypt the transmitted data exploiting known algorithms issues;
- **Issues related to digital certificates:** failures in certificates validation or absence of certificate pinning.

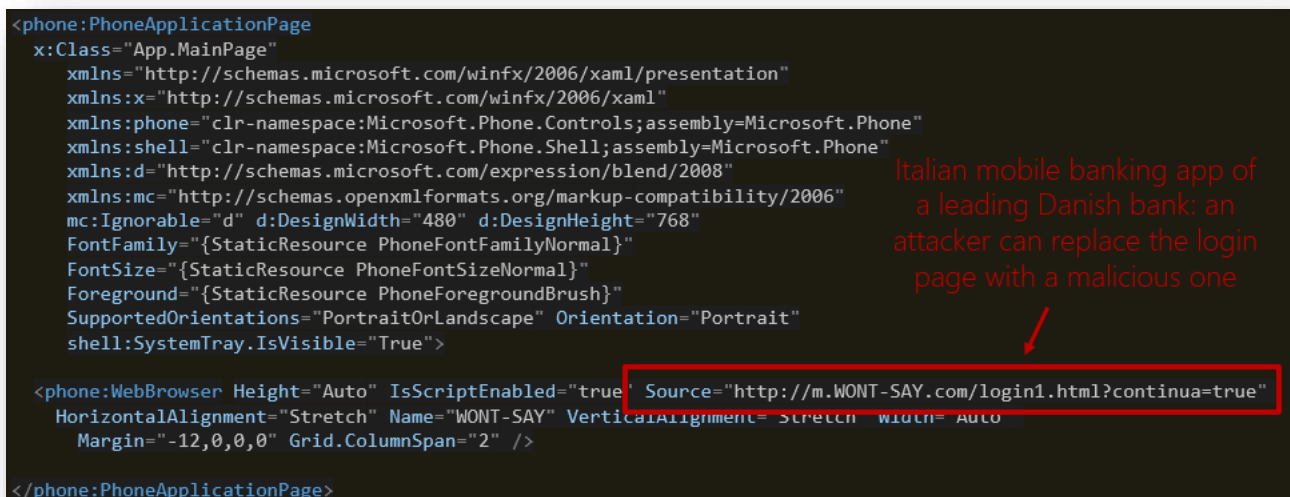


Figure 5 – Loading of remote HTML login page via http.

```

namespace VulnApp
{
    public class MainPage : PhoneApplicationPage
    {
        private Uri home;
        internal WebBrowser Browser;
        [...]

        public MainPage()
        {
            this.InitializeComponent();
            this.home = new Uri("http://vulnerable.com");
            [...]
        }

        [...]
        private void Home_Click(object sender, EventArgs e)
        {
            this.Browser.Navigate(this.home);
        }
    }
}

```

an attacker can manipulate the entire app layout

Figure 6 – A MitM attack allows a malicious user to manipulate the entire app layout.

```

public CordovaView()
{
    this.InitializeComponent();
    if (DesignerProperties.IsInDesignTool)
        return;

    // [...]
    if (this.configHandler.ContentSrc != null)
    {
        this.StartPageUri = !Uri.IsWellFormedUriString(this.configHandler.ContentSrc, UriKind.Absolute) ?
            new Uri(CordovaView.AppRoot + "www/" + this.configHandler.ContentSrc, UriKind.Relative) :
            new Uri(this.configHandler.ContentSrc, UriKind.Absolute);
    }
}

```

www/index.html loads a remote JS via http

```

<!-- ... -->
<link rel="stylesheet" href="style.css" />
<link rel="stylesheet" href="style-icons.css" />
<script type="text/javascript"
src="http://maps.googleapis.com/maps/api/js?v=3.4&key=ABCDE[...]&libraries=places"></script>
<!-- ... -->

```

Figure 7 – Loading of an external JavaScript file via HTTP protocol.

Windows Phone 8.0 *automagically* discards invalid certificates and no public APIs are available to programmatically disable the behavior. Windows Phone 8.1 allows developers to specify errors to be ignored with **HttpBaseProtocolFilter.IgnoreableServerCertificateErrors.Add()**: the method accepts parameters described by the **ChainValidationResult** enumeration, but only a subset of the available values can be specified.

Reference	Ignorable	Not Ignorable
ChainValidationResult (Enumeration)	Expired	Success
	IncompleteChain	Revoked
	WrongUsage	InvalidSignature
	InvalidName	InvalidCertificateAuthorityPolicy
	RevocationInformationMissing	BasicConstraintsError
	RevocationFailure	UnknownCriticalExtension
	Untrusted	OtherErrors

Windows Phone 8.0 platform does not provide any APIs to directly implement pinning mechanisms. Instead, with Windows Phone 8.1 developers can use the **SocketStream.Information** to access **StreamSocketInformation.ServerCertificate**, which allows the retrieval of the remote server's digital certificate: the returned **Certificate** object can then be used to get detailed information on the certificate itself.

The following tables summarize the methods and classes that should be carefully audited to spot vulnerabilities concerning data transmission.

Category	Namespaces	Classes, Methods and Properties	
HTTP	System.Net.Http	HttpClient.DeleteAsync() HttpClient.GetAsync() HttpClient.PostAsync() HttpClient.PutAsync()	HttpClient.GetByteArrayAsync() HttpClient.GetStreamAsync() HttpClient.GetStringAsync() HttpClient.SendAsync()
	Windows.Web.Http	HttpClient.DeleteAsync() HttpClient.GetAsync() HttpClient.PostAsync() HttpClient.PutAsync()	HttpClient.GetStringAsync() HttpClient.SendRequestAsync() HttpClient.GetBufferAsyn() HttpClient.GetInputStreamAsync()
TCP and UDP Sockets	Windows.Networking.Sockets	StreamSocket.ConnectAsync() SocketProtectionLevel.PlainSocket - property StreamSocket.UpgradeToSslAsync() StreamSocketListener - does not support SSL/TLS DatagramSocket.ConnectAsync()	

Category	Namespaces	Classes, Methods and Properties
Web	Microsoft.Phone.Controls	WebBrowser.Navigate() WebBrowser.Source property
	Windows.UI.Xaml.Controls	WebView.Navigate() WebView.Source property
	Microsoft.Phone.Tasks	WebBrowserTask.Uri property
	Windows.System	Launcher.LaunchUriAsync(uri)
WebSocket	Windows.Networking.Sockets	MessageWebSocket.ConnectAsync() – with ws:// uri scheme StreamWebSocket.ConnectAsync() – with ws:// uri scheme
XAML Object Element Usage	-	«Source» property for WebBrowser and WebView «uri» property for WebBrowserTask "NavigateUri" for HyperlinkButton
Push Notifications	Microsoft.Phone.Notification	HttpNotificationChannel(string)
Brutal Approach	-	Grep for Uri() and look at http:// instead of https://
Digital Certificates	Windows.Web.Http.Filters	HttpBaseProtocolFilter.IgnoreableServerCertificateErrors.Add()
Windows.Web.AtomPub, Windows.Networking.BackgroundTransfer, Windows.Web.Syndication classes/methods should be reviewed as well		

It should be noted that in most cases, the use of **https://** instead of **http://** allows developers to properly establish secure encrypted channels.

M4 – Unintended Data Leakage

M4 refers to involuntary data exposure caused by OS or frameworks *side-effects*. In order to properly exploit these issues, an attacker is typically required to hold a privileged access to the target file system or connected network.

Potential sources of information leakage are:

- **System caching:** OS-level caching may result in data leakage if the app does not delete critical information stored in the cache when they are no longer necessary;
- **Application backgrounding:** when an application is backgrounded, data saved or cached by built-in mechanisms - such as session cookies delivered by the back-end system, or previously navigated web pages by a web view - should be deleted to avoid private data theft in case of unauthorized access to the corresponding storage areas (Figure 8);
- **System logging:** the OS may log sensitive data that could be accessed by malicious users or apps on device;
- **Telemetry frameworks, which expose sensitive data:** telemetry frameworks may transmit data in *plain text* - and so sniffed by attackers located on the same network segment - or share reserved information with back-end systems (these system are part of the overall attack surface).

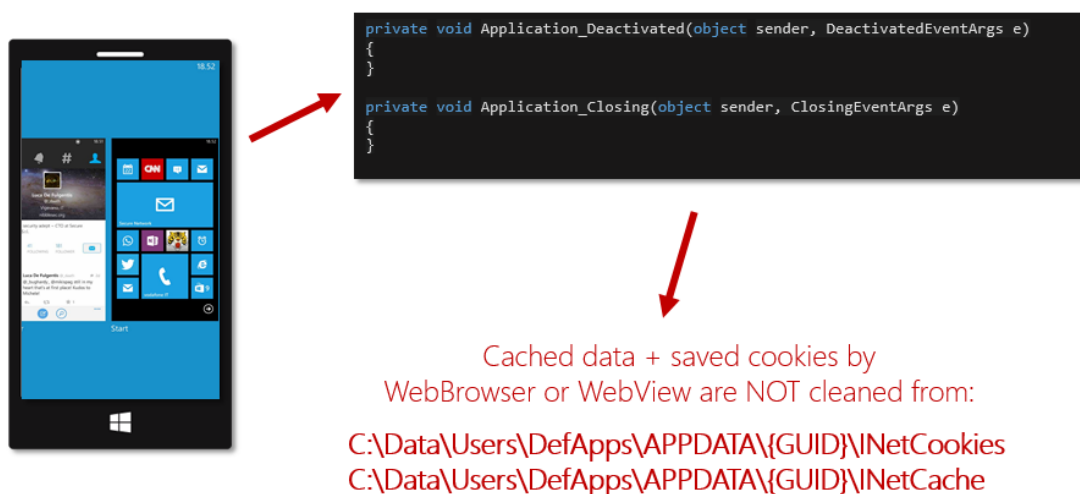


Figure 8 – Cached data are not automatically deleted by the OS on app deactivation or closing.

The following table summarizes a set of methods, related to *application backgrounding*, which should always implement routines to delete cookies and cached, before the interruption of the app execution.

Conditions	Classes, Methods and Properties
Application Backgrounding and Closing	Handler for the Application.Suspending event, typically the OnSuspending() method in App.xaml.cs
	Handler for the Application.Deactivated event, typically the Application_Deactivated() method in App.xaml.cs
	Handler for the Application.Closing event, typically the Application_Closing() method in App.xaml.cs
	Handler for the Application.UnhandledException event, typically the Application_UnhandledException() method in App.xaml.cs
Use of Telemetry Frameworks	HockeyApp, BugSense, etc.
Dump of App Memory	check for encryption keys or passwords stored as «string» (immutable) object (.NET's System.Security.SecureString object is not supported by WP Silverlight Runtime)

Finally, the following table provides strategies to mitigate common data leakage conditions with the Windows Phone platform.

Actions	Classes, Methods or Properties	
Remove cached data on app closing, suspension or deactivation	server-side	Cache-Control: no-store
	client-side	WebBrowserExtensions.ClearInternetCacheAsync() WebBrowser.ClearInternetCacheAsync() WebView - no programmatic way
Remove stored cookies	WebBrowser.ClearCookiesAsync() WebBrowserExtensions.ClearCookie() WebView – use HttpCookieManager.GetCookies() + HttpCookieManager.DeleteCookie()	
Avoid sensitive data disclosure (dump of app's memory)	Use of byte[] array instead of System.String objects, and re-assign bytes when the "secret" is no longer necessary	

M5 – Poor Authorization and Authentication

M5 refers to security *decisions* taken by a mobile app without server-side engagement, thus allowing a malicious user to achieve an unauthorized access to application data or functionalities. Common issues belonging to M5 are:

- **Offline authentication:** offline authentication implies the existence of “secret” information or logic on device, capable to validated user-typed credentials. The solution is prone to binary attacks, because a malicious user can extract the secret and the bypasses the authentication mechanism;
- **Issues related to password complexity:** use of weak password *by design*, such as 4 digits PIN;
- **Absence of anti-guessing or brute forcing mechanisms:** these issues can be related to both app or back-end authentication mechanisms;
- **Predictable authentication/authorization tokens:** generation of authentication or authorization tokens based on predictable user or device information. An attacker may easily forge valid tokens and bypass implemented security mechanisms – both on client or server-side (Figure 9);
- **Authorization issues on apps critical functions/data access:** an application may implement weak authorization mechanisms, which could be exploited to access premium functionalities or steal confidential data (Figure 10 and Figure 11). In the Windows Phone ecosystem, a common issue is represented by the lack of proper client-side authorization mechanisms within the **OnNavigatedTo()** method, which refers to XAML pages that may expose reserved functionalities.

```
public byte[] GetPostData()
{
    string date = DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss", (IFormatProvider) CultureInfo.InvariantCulture);
    return Encoding.UTF8.GetBytes(string.Format("token={0}&&msisdn={1}&InteractiveType=Web&dt={2}",
        this.GetAuthorizationToken(), HttpUtility.UrlEncode(this.par), HttpUtility.UrlEncode(date)));
}

private string GetAuthorizationToken()
{
    string str = MD5Core.GetHashString("RYKn92938339944005kf8fk9ekwdkwud83jud3" + this.par).ToLower();
    Utils.Log("MDE-Token: " + str);
    return str;
}
```

token generation logic can
be easily reproduced

Figure 9 - Client-side generation of an authorization token.

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    base.OnNavigatedTo(e);
    using (IsolatedStorageFile storeForApplication = IsolatedStorageFile.GetUserStoreForApplication())
    {
        this.fileExists = storeForApplication.FileExists("wp contacts backup.zip");
        if (!this.fileExists)
        {
            this.infoTextBlock.Text = "No backup file exists! Please create one before trying to download it.";
        }
        else
        {
            try
            {
                this.server = new HttpServer(2, 65536);
                this.server.Start(new IPEndPoint(IPAddress.Parse("0.0.0.0"), 5656));
                this.server.TextReceived += new EventHandler<HttpDataReceivedEventArgs>(this.server_TextReceived);
                this.infoTextBlock.Text = "http://" + this.server.LocalEndpoint.ToString();
            }
            catch
            {
                this.infoTextBlock.Text = "Unable to start WEB Server. Please check your connectivity settings.";
            }
        }
    }
}
```

contacts backup file is stored in app's sandbox

Figure 10 – Example of contacts backup application affected by an authentication issue.

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    base.OnNavigatedTo(e);
    using (IsolatedStorageFile storeForApplication = IsolatedStorageFile.GetUserStoreForApplication())
    {
        this.fileExists = storeForApplication.FileExists("wp contacts backup.zip");
        if (!this.fileExists)
        {
            this.infoTextBlock.Text = "No backup file exists! Please create one before trying to download it.";
        }
        else
        {
            try
            {
                this.server = new HttpServer(2, 65536);
                this.server.Start(new IPEndPoint(IPAddress.Parse("0.0.0.0"), 5656));
                this.server.TextReceived += new EventHandler<HttpDataReceivedEventArgs>(this.server_TextReceived);
                this.infoTextBlock.Text = "http://" + this.server.LocalEndpoint.ToString();
            }
            catch
            {
                this.infoTextBlock.Text = "Unable to start WEB Server. Please check your connectivity settings.";
            }
        }
    }
}
```

local web server lacks any authentication mechanism

Figure 11 – Contacts backup app does not implement any authentication mechanism on backup download.

The following table highlights a series of Windows Phone APIs commonly adopted to generate authentication/authorization tokens, which should be located within an app and carefully audited.

Identification Data	Namespaces	Classes, Methods and Properties
Device Name	Microsoft.Phone.Info	DeviceStatus.DeviceName
Hardware Identification	Microsoft.Phone.Info	DeviceExtendedProperties.GetValue("DeviceName")
		DeviceExtendedProperties.GetValue("DeviceUniqueId")
Hashing Functions	Windows.Security.Cryptography	SHA1Managed, SHA256Managed, SHA384Managed and SHA512Managed classes (or any other 3° party libraries implementing these functions)
	Windows.Security.Cryptography.Core	HashAlgorithmProvider.OpenAlgorithm()
Geo Location Coordinates	Windows.Devices.Geolocation	Geolocator / Geoposition / Geocoordinate
	System.Device.Location	GeoCoordinateWatcher / GeoPosition / GeoCoordinate

It is a common practice to use the **DeviceExtendedProperties.GetValue("DeviceUniqueId")** as a unique identifier for an app on a device. If an app requires a "real" UUID to identify itself with a remote back-end (identify, not authenticate!) we suggest the use of **HostInformation.PublisherHostId** property instead, because the corresponding string is unique *per device and per publisher*, while the **DeviceExtendedProperties.GetValue("DeviceUniqueId")** is unique *only per device*.

M6 – Broken Cryptography

M6 refers to the risk associated with both the use of weak cryptographic algorithms and processes in the encryption of data stored on device or *in transit*.

The most common issues related to the adoption of *weak cryptography* are:

- **Weak standard cryptographic algorithms:** an app may use cryptographic algorithms that are weak or are affected by intrinsic vulnerabilities, thus allowing an attacker to violate data confidentiality;
- **Custom algorithms:** the use of custom encryption algorithms is normally discouraged in favor of strong community-approved cryptographic solutions;
- **Exotic “encryption” strategies:** for example the use of *encoding* (e.g., serialization) instead of encryption. Encoding is just a data *representation* and not a method to guarantee data confidentiality (Figure 12).

When considering a data *encryption processes*, the most relevant issues are:

- **Hardcoded encryption keys:** if an encryption key is hardcoded in app’s code, an attack against application binaries allows the extraction of the secret, and the decryption of the data (Figure 13);
- **Encryption keys stored with the encrypted data:** an attacker with access to the application sandbox can retrieve both the encrypted data and the key for the decryption;
- **Encryption keys stored in an *unsafe areas*:** keys stored in unsafe areas (e.g., file system’s shared locations) could be easily retrieved by malicious apps or attackers having access to these locations.

The following table summarizes classes and methods that should be identified while assessing mobile app’s source code, in order to spot the existence of issues in the encryption process or in the use of cryptography.

Functions	Namespaces	Classes, Methods and Properties
Hashing	Windows.Security.Cryptography	SHA1Managed, SHA256Managed, SHA384Managed and SHA512Managed classes (or any other 3° party libraries implementing these functions)
	Windows.Security.Cryptography.Core	HashAlgorithmProvider.OpenAlgorithm()
Symmetric Encryption	System.Security.Cryptography.Core	CryptographicEngine.Encrypt() Decrypt()
	System.Security.Cryptography	AesManaged.CreateEncryptor() CreateDecryptor()
Data Encoding	Windows.Security.Cryptography	CryptographicBuffer.[Encode Decode]ToBase64String() CryptographicBuffer.[Encode Decode]ToHexString()
	System.Text	Encoding.UTF8
	System	Convert.ToBase64String() Convert.FromBase64String()
Plenty of third party encryption libraries (e.g., Bouncy Castle for .NET) implement similar algorithms		

```

private void GetUserCompleted(object sender, EventArgs e)
{
    if (e == null)
    {
        // ...
    }
    else
    {
        NetUserCompletedEventArgs completedEventArgs = (NetUserCompletedEventArgs) e;
        byte[] numArray1 = Crypto.encryptString(completedEventArgs.user.username);
        byte[] numArray2 = Crypto.encryptString(completedEventArgs.user.password);
        this.isolatedStorageSettings.StoreValueForKey("Username", (object) numArray1);
        this.isolatedStorageSettings.StoreValueForKey("Password", (object) numArray2);
        CurrentAppConfig.Instance.User = completedEventArgs.user;
        this.storeCurrentUserStoresPreferences(completedEventArgs.user);
    }
}

public class Crypto
{
    public static byte[] encryptString(string input)
    {
        return Encoding.UTF8.GetBytes(input);
    }
}

```

"encrypted" credentials are stored into the sandbox

Figure 12 – Use of serialization instead of an encryption mechanism for credential storage.

```

private static readonly string SaltKey = "*****salt_here*****";

public static string EncryptPlainText(string dataToEncrypt)
{
    AesManaged aesManaged = new AesManaged();
    byte[] bytes1 = new UTF8Encoding().GetBytes(SecurityHelper.SaltKey);

    Rfc2898DeriveBytes rfc2898DeriveBytes = new Rfc2898DeriveBytes(SecurityHelper.SaltKey, bytes1);
    aesManaged.Key = rfc2898DeriveBytes.GetBytes(16);
    aesManaged.IV = rfc2898DeriveBytes.GetBytes(16);
    aesManaged.BlockSize = 128;

    using (MemoryStream memoryStream = new MemoryStream())
    {
        using (CryptoStream cryptoStream = new CryptoStream((Stream) memoryStream,
            aesManaged.CreateEncryptor(), CryptoStreamMode.Write))
        {
            byte[] bytes2 = Encoding.UTF8.GetBytes(dataToEncrypt);
            cryptoStream.Write(bytes2, 0, bytes2.Length);
            cryptoStream.FlushFinalBlock();
            cryptoStream.Close();
            return Convert.ToBase64String(memoryStream.ToArray());
        }
    }
}

```

hardcoded (and same) symmetric encryption key and salt

Figure 13 – Hardcoded symmetric encryption key.

Developers should avoid storing any critical data on device, even if encrypted. However, if this represents a key requirement, the information must be stored adopting *built-in* encryption algorithms. Windows Phone exposes the **Data Protection API** (DPAPI), which guarantees OS-level data encryption. The following table lists the classes and methods that allow developer to safely encrypt data.

Platform	Namespaces	Methods
WP 8.0	System.Security.Cryptography	ProtectedData.Protect() ProtectedData.Unprotect()
WP 8.1	Windows.Security.Cryptography	DataProtectionProvider.ProtectAsync() DataProtectionProvider.UnprotectAsync() DataProtectionProvider.ProtectStreamAsync() DataProtectionProvider.UnprotectStreamAsync()

M7 – Client Side Injection

M7 represents a class of security issues where an interpreter (e.g., a SQL querying engine, a system shell, an XML parser) is *fed* with untrusted data. Untrusted data is used by the vulnerable application to generate a *command*, which is then delivered to the corresponding parser. If no proper validation routines are in place, an attacker may inject special characters and statements to change the *semantic* of the altered command, affecting the confidentiality, integrity or availability of the data handled by the targeted interpreter and underlying system.

Client-side injections are issues that involve interpreters located on the mobile device – the *client side*. The impact of an injection attack depends on the criticality of stored data, and on the specific category of injection.

In order to spot client-side injections it is necessary to map the sources of untrusted data, and then review how these inputs are handled by the underlying parsing routines. Example of untrusted data sources for the Windows Phone platform are:

- **Input from network** – e.g., web responses or any other network communications;
- **Bluetooth or NFC**;
- **Inter Process Communication (IPC) mechanisms** - e.g., via extensions/protocols registration or toast message notifications;
- **Files accessed from SD card** – which is a shared storage area;
- **User typed input** – e.g., via UI, speech to text, camera (e.g., QR code), USB data;

Once identified the sources of potential untrusted data, the following tables can be adopted as a *driver* to address potential unsafe usage of APIs, which may lead to client-side injection vulnerabilities, such as Cross-Site Scripting (XSS - Figure 14), SQL Injections and XML Injections.

Interpreters	Namespaces	Classes, Methods and Properties
HTML/JavaScript	Microsoft.Phone.Controls	WebBrowser.NavigateToString() WebBrowser.InvokeScript() WebBrowser.IsScriptEnabled = true (property)
	Windows.UI.Xaml.Controls	WebView.NavigateToString() WebView.InvokeScript() InvokeScriptAsync() WebView.NavigateToLocalStreamUri() WebView.NavigateWithHttpRequestMessage()
XML	System.Xml.Linq	XDocument.Load()
	System.Xml	XmlReaderSettings.DtdProcessing = DtdProcessing.Parse
XAML	System.Windows.Markup	XamlReader.Load()

Interpreters	Third Parties Libraries	Classes, Methods and Properties
SQL	SQLitePCL	SQLiteConnection.Prepare()
	SQLite-Net-WP8	Query() / Query<T>() / QueryAsync<T>() Execute() / ExecuteAsync() ExecuteScalar<T>() / ExecuteScalarAsync<>() DeferredQuery() / DeferredQuery<T>() FindWithQuery<T>() CreateCommand()
	CSharp-SQLite	SqlCommand.CommandText (property)
	SQLiteWinRT	Database.ExecuteStatementAsync() Database.PrepareStatementAsync()

```
private void ButtonView_Click(object sender, RoutedEventArgs e)
{
    this.ButtonView.IsEnabled = false;
    this.ipTextTemp = this.TextIP.Text.Trim() + "xxxxxxx";
    this.WebBrowser1.NavigateToString('<body bgcolor=black>' +
    "<form action= https://www.REMOTE-SITE.com/path/resource.asp' method=post>" +
    "<input name='iname' value='" + this.TextAdmin.Text.Trim() + "' type='hidden'>" +
    "<input name='pword' value='" + this.PasswordBox1.Password.Trim() +
    "' type=hidden><input name='ip' value='" + this.TextIP.Text.Trim() +
    "' type=hidden><input name='port' value='" + this.TextPort.Text.Trim() +
    "' type=hidden><input name= vers1 value= or1 type= nidden ></form>" +
    "<script>document.forms[0].submit();</script></body>");
}
```

app renders user-controlled data without any validation

Figure 14 – Example of vulnerable usage of the WebBrowser.NavigateToString() method.

Developers should always intend all inputs as *evil* and implement proper input validation routines, adopting *positive validation* strategies, in order to effectively mitigate attacks, such as injections.

Client-side SQL Injection issues can be effectively blocked adopting *parameterized queries* instead of concatenating strings with user-controlled input.

Finally, apps should not enable the DTD parsing features when working with XML documents, because it could be abused to conduct attacks such as **XML External Entity** (XXE).

M8 – Security Decisions via Untrusted Inputs

Inter Process Communication (IPC) mechanisms extend an application's *attack surface* and represent sources for untrusted inputs and unauthorized *actions*. Starting from version 8, Windows Phone provides support to IPC with file and UR associations.

File and URI associations allow the automatic execution of an app when another app launches a registered file type or URI. In order to be executed, an app must register either a file type/extension or an URI scheme in its manifest file, as summarized in the following tables – note that differences exist between WP 8.0 and WP 8.1 platforms.

IPC Mechanism	Supported Platform and Manifest Specifications (WMApManifest.xml)
File Association	<pre><Extensions> <FileTypeAssociation Name="name" TaskID="_default" NavUriFragment="fileToken=%s"> [...] <SupportedFileType> <FileType ContentType="application/sdk">.test1</FileType> <FileType ContentType="application/sdk">.test2</FileType> </SupportedFileTypes> </FileTypeAssociation> </Extensions></pre>
	email attachment, SD cards, website via IE, WebBrowser/WebView, NFC-enabled devices or another app from the Store (Launcher.LaunchFileAsync)
URI Association	<pre><Extensions> <Protocol Name="luca" NavUriFragment="encodedLaunchUri=%s" TaskID="_default" /> </Extensions></pre>
	click here (IE/WebBrowser/WebView), other apps via Launcher.LaunchUriAsync("luca:..") or NFC-enabled devices

IPC Mechanism	Supported Platform and Manifest Specifications (package.appxmanifest)
File Association	<pre><Extensions> <Extension Category="windows.fileTypeAssociation"> <FileTypeAssociation Name="test3"> <DisplayName>My test 3</DisplayName> [...] <SupportedFileTypes> <FileType ContentType="image/jpeg">.test1</FileType> </SupportedFileTypes> </FileTypeAssociation> </Extension> </Extensions></pre>

URI Association	<pre> <Extensions> <Extension Category="windows.protocol"> <Protocol Name="luca" m2:DesiredView="useLess"/> <Logo>images\logo.png</Logo> <DisplayName>My uri has my name</DisplayName> </Extension> </Extensions> </pre>
-----------------	--

A third IPC mechanism is based upon the undocumented **Shell_PostMessageToast** method (**ShellChromeAPI.dll**), which can be abused to perform *Cross Application Navigation Forgery* attacks. The undocumented feature has been identified by cpuguy from XDA, while the term “Cross Application Navigation Forgery” has been coined by Alex Plaskett and Nick Walke from MWR.

Basically a malicious app can use the **Shell_PostMessageToast** method to send a *toast message* that, once tapped by the victim-user, implies the launch of an attacker-controlled XAML page belonging to the target app (Figure 15): an attacker may tamper parameters passed to the **OnNavigatedTo()** and so attack the *code behind* logic.

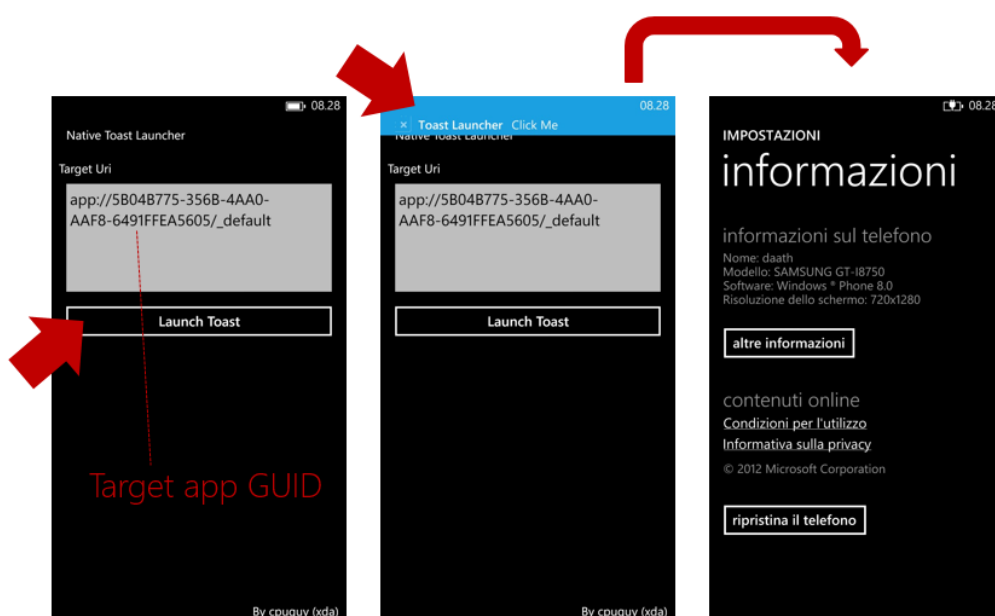


Figure 15 – Demonstration of a Cross Application Navigation Attack using the Native Toast Launcher utility.

While performing a security review of a Windows Phone mobile app, the methods listed in the following table should be carefully audited to identify eventually input validation and authorization issues. In fact, attackers or malicious apps on the target device can trigger their execution under certain circumstances. The impact of the exploitation depends upon the specific implementation and handled data by the targeted methods.

IPC Mechanism	Platform	Namespaces	Classes, Methods and Properties
URI Associations	WP 8.0	System.Windows.Navigation	overridden UriMapperBase.MapUri() method
	WP 8.1	Windows.UI.Xaml.Application	OnActivated() - ActivationKind.Protocol property
File Associations	WP 8.0	System.Windows.Navigation	overridden UriMapperBase.MapUri() method
	WP 8.1	Windows.UI.Xaml.Application	OnFileActivated() method
(Toast Message)	WP 8.0	System.Windows.Navigation	OnNavigatedTo() (NavigationContext.QueryString)

M9 – Improper Session Handling

M9 refers to the existence of insecure app sessions *life cycle*, and embraces issues related to both client and server-side *session handlers*. Common issues belonging to M9 are:

- **Failure to invalidate session on the back-end:** mobile apps logout mechanisms often do not involve server-side sessions invalidation. Instead these functions just invalidate cookies stored on the device (Figure 16);
- **Lack of adequate timeout protection:** back-end systems should force session timeout in order to avoid reuse of stolen cookies. The specific requirement refers to server-side components, even if a session (forced) timeout should be introduced within the app as well;
- **Insecure tokens creation:** client-side generation of authentication or authorization tokens should be avoided because it may rely on predictable information or logic that could be easily reproduced by attackers;
- **Failure to invalidate sessions on app closing or deactivation:** on app closing, backgrounding or suspension, session cookies should be cleaned and invalidated (server-side) to avoid the persistence of confidential data onto the device's file system, thus potentially exposed to data leakage.

```
public void Logout()
{
    if (this.AppState.StCookies != null && this.AppState.StCookies.Count > 0)
    {
        foreach (KeyValuePair<string, System.Net.Cookie> keyValuePair in this.AppState.StCookies)
        {
            System.Net.Cookie cookie = keyValuePair.Value;
            cookie.Expired = true;
            cookie.Discard = true;
        }
    }

    this.AppState.StCookies = (Dictionary<string, System.Net.Cookie>) null;

    ((Frame) this.rootFrame).Navigate(new Uri(ViewList.PreLogin, UriKind.Relative));
}
```

no server-side session cookies
invalidation mechanism is
involved in the logout process

Figure 16 – The method does not involve the back-end in the logout process.

Both developers and security professionals should carefully analyze target apps and identify session handling routines and cookies usage, while ensuring that these mechanisms delete the tokens when they are not needed.

The following is a set of classes that should be carefully audited during code analysis:

- **System.Net** namespace
 - **System.Net.Cookie**
 - **System.Net.CookieCollection**
 - **System.Net.CookieContainer**
 - **System.Net.HttpWebRequest.CookieContainer**
 - **System.Net.HttpWebResponse.Cookies**
- **Windows.Web.Http** namespace (WP 8.1 only)
 - **Windows.Web.Http.HttpCookie**
 - **Windows.Web.Http.HttpCookieCollection**
 - **Windows.Web.Http.HttpCookieManager**

M10 – Lack of Binary Protections

A mobile application should protect itself from *binary attacks*, which have the objective to analyze the application behavior and exchanged data with back-end systems, reverse the app's internals (in order to steal intellectual property) or modify and redistribute the app itself as a malware to fraud users. Moreover, the lack of binary protections may allow a cracker to easily bypass *premium* function protections, thus causing a financial damage to the app's software house. In order to avoid these risks, a mobile application should implement a series of binary protections, such as the following:

- **Certificate pinning:** certificate pinning mechanisms slow down HTTPS traffic analysis, because the attacker is required to *unpin* the certificate, by modifying the target app, before starting communication traffic investigation. Refer to M3 for further technical details on certificate pinning;
- **Code obfuscation:** app bytecode can be trivially decompiled in source code using public available tools such as **.NET Reflector** or **JetBrains dotPeek**. Developers should therefore apply proper bytecode obfuscation techniques in order to *mitigate* the risk of intellectual property theft. It should be noted that even if an app does not implement any reserved business logic, code obfuscation is still suggested in order to further slowdown client analysis from malicious users;
- **Anti-debugging and runtime-tampering detection mechanisms:** anti-debugging and runtime-tampering detection routines usually require the attacker to spend further effort in application analysis, because these mechanisms must be identified and then disabled by modifying both the analysis environment and the app's code. Code obfuscator utilities often implement this kind of mechanisms (e.g., **dotFuscator** and **ConfuserEx**);
- **Code encryption:** Windows Phone Store apps are downloaded as encrypted files and then decrypted during the deployment phase. In these conditions, a privileged access to the device file system allows *clear-text* apps code extraction. Therefore custom app code encryption should be implemented as a further layer of security against reverse engineering. Code obfuscation utilities introduce code obfuscation mechanisms as well.

In order to further increase apps security, it should be configured a specific option that prevents app installation on SD cards (Figure 17). Moreover, The *OWASP Reverse Engineering and Code Modification Prevention Project* provides architectural principles to secure design your apps.

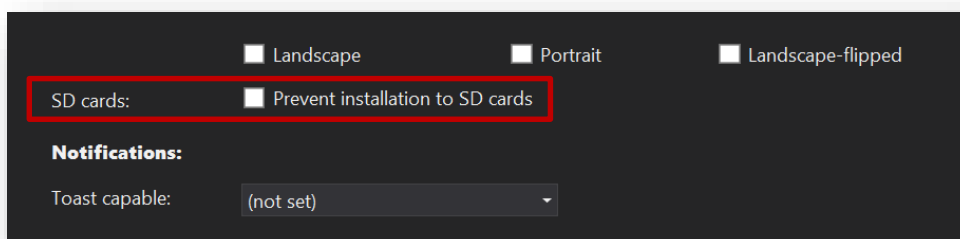


Figure 17 – SD card installation can be disabled via specific app's manifest option.

Conclusions

The paper illustrated our research results concerning the analysis of the security of 200+ applications targeting the Windows Phone platform, developed with both Silverlight and Windows Runtime technologies.

During the investigation, examples of vulnerable code have been collected and mapped to the corresponding MTT 2014 entries. The analysis also allowed the definition and categorization of insecure APIs usage, which represents the most complete and accurate public *catalog* in the genre - so a precious reference for both developers and security professionals.

Finally, it should be noted that substantial part of our study on APIs security should be valid for the Universal Apps targeting Windows Phone 8.1 and Windows 10 for Phone platforms.

References

- **MSDN – API References for Windows Runtime Apps**
 - <https://msdn.microsoft.com/en-us/library/windows/apps/xaml/br211369.aspx>
- **Certificate Pinning in Mobile Applications**
 - <http://www.slideshare.net/iazza/certificate-pinning-in-mobile-applicationsproscons10>
- **Input Validation Cheat Sheet**
 - https://www.owasp.org/index.php/Input_Validation_Cheat_Sheet
- **Native Toast Notification Launcher**
 - <http://forum.xda-developers.com/windows-phone-8/help/qa-native-toast-notification-launcher-t2980873>
- **MWR's Navigating a Sea of Pwn?**
 - https://labs.mwrinfosecurity.com/system/assets/651/original/mwri_wp8_appsec-whitepaper-syscan_2014-03-30.pdf
- **OWASP Mobile Jailbreaking Cheat Sheet**
 - https://www.owasp.org/index.php/Mobile_Jailbreaking_Cheat_Sheet
- **OWASP Reverse Engineering and Code Modification Prevention Project**
 - https://www.owasp.org/index.php/OWASP_Reverse_Engineering_and_Code_Modification_Prevention_Project
- **Windows Phone 8.1 Security Overview**
 - <https://www.microsoft.com/en-us/download/details.aspx?id=42509>
- **Windows Phone 8 Application Security**
 - <http://erpsan.com/wp-content/uploads/2013/06/Windows-Phone-8-application-security-slides.pdf>