**Device Security and Application Deployment**

**(Part 6)**

**Topic 1: Understanding Device Security**

**Topic 2: Deploying applications through a Web Service**

[**http://msdn.microsoft.com/rampup**](http://msdn.microsoft.com/rampup)

**Introduction**

Developing applications for Windows Mobile Devices is similar to developing applications for the desktop, especially if you make use of either Visual Basic.NET or Visual C#. You can use the same development tools to develop your applications, although there are also differences between developing applications for Windows Mobile Devices and developing desktop applications. Depending on the device manufacturer or the mobile operator that sold the device, Windows Mobile devices can have different security models. From a developer’s perspective, device security determines if an application can run on a device and what access to system resources it has. It also has an impact in how you can distribute and install your applications, especially if you want to target a broad range of different devices from different manufacturers of mobile operators.

This article provides information on understanding device security, on how to test different security configurations using Visual Studio 2008’s security manager and how to successfully install your application on a Windows Mobile Device.

**Windows Mobile Device Security**

Different Windows Mobile Devices might have different security configurations installed. Depending on the device manufacturer or the mobile operator, a device might be entirely open to install new applications, a device might prompt the user prior to installing or running an application for the first time on the device or devices might be even locked down completely, preventing the installation of new applications that are not signed with a matching certificate that is available in the device certificate store.

The reason to have different security configurations available has to do with the fact that most modern Windows Mobile devices have phone capabilities and will be used to communicate over a network that is owned by a mobile operator. An important reason for securing Windows Mobile devices is network integrity. Of course mobile operators want to protect their network against malfunctioning software that end users can easily install on a device. That is the reason why it is often not that easy to install each and any application on a Windows Mobile device, even if the application was specifically written to run on Windows Mobile devices.

In order to fulfill the security requirements demanded by mobile operators, Windows Mobile devices can have different security configurations. The exact way security is configured on a Windows Mobile device not only depends on the mobile operator or an enterprise organization, it also depends on the particular Windows Mobile device that is used. Security on Windows Mobile devices is essential and vastly built-in into the Operating System. Different layers of security combined together result in a security configuration that exactly determines if applications can be installed on a device and how they run if they are allowed to run.

**Application Execution Permissions**

Depending on the security configuration of a particular Windows Mobile device, applications might be allowed to run or might be blocked from execution on the device. The following application execution permissions are defined for Windows Mobile Devices:

* **Privileged** – The application can do everything on the device, has full write access to the file system and to the system registry and is also allowed to install certificates that might allow other applications to run on a particular Windows Mobile device.
* **Normal** – The application is restricted in its execution; it cannot call trusted [Win32 APIs](http://msdn.microsoft.com/en-us/library/ms924486.aspx), write to protected areas of the registry, write to system files, or install certificates.
* **Blocked** – The application is not allowed to execute at all.

**Application Certificates**

To determine the application’s execution permission, it is possible to sign your application with a certificate. Windows Mobile devices contain two different certificate stores for this purpose, a privileged certificate store and a normal certificate store. If you sign your application with a certificate from the privileged store, it will get privileged access permissions. An application that is signed with a certificate from the normal store will get normal access permissions. Because of the fact that another party (the OEM or the mobile operator) maintains these certificate stores, you must work with these organizations to get your application signed for a specific party. However, as an ISV you probably want your applications to run on a large range of Windows Mobile devices. Especially with this in mind, the [Mobile2Market](http://www.mobile2market.net/SignUp1.asp)[[1]](#footnote-2) initiative was started through which you can submit your applications for verification and get them signed with the desired certificate. Singing your application through Mobile2Market also means that you or your organization will be identified as the publisher of the application.

**Application Access Levels**

Different access levels determine what an unsigned application is allowed to do on a Windows Mobile device. These different access levels are called **tiers**. Windows Mobile makes use of two different access levels with the following meaning:

* **One-tier security** – A signed application is allowed to execute on a device, it will run with privileged permission, thus having full device access. An unsigned application might be allowed to execute on a device, although this is determined by the security configuration of the device. If an unsigned application is allowed to run it will also run with privileged permission.
* **Two-tier security** – A signed application is allowed to execute on a device, but it will either run with privileged or normal permissions, depending on the type of certificate the application was signed with. An unsigned application might be allowed to execute on a device, although this is determined by the security configuration of the device. If an unsigned application is allowed to run it will always run with normal permission.

**Device Security Configurations**

Now that you know about the different application permissions to run on Windows Mobile devices and about signing your applications with a certificate, one important question is still unanswered. Can your application, especially if the application is unsigned; be installed on a particular Windows Mobile device. There are a number of standard device security configurations defined for Windows Mobile devices, some of which allow users to install and run unsigned applications:

* **Security Off** – All applications will run on the device and all will run with privileged permission. There is no difference between signed and unsigned applications.
* **One-Tier Prompt** - All signed applications will run on the device and all will run with privileged permission. The user decides if unsigned applications are allowed on their device. If the user accepts an unsigned application it will run with privileged permission.
* **Two-Tier Prompt** - All signed applications will run on the device, but the execution permission is determined by the type of certificate the application is signed with (either privileged or normal). The user decides if unsigned applications are allowed on their device. If the user accepts an unsigned application it will run with normal permission.
* **Third-Party Signed** – Only signed applications are allowed to run on the device. Execution permission is determined by the type of certificate the application is signed with (either privileged or normal). In order to get your application signed you have to sign-up to a developer program, like Mobile2Market. Most Windows Mobile devices that are purchased through most mobile operators have Mobile2Market certificates installed in the certificate store.
* **Locked** - Only signed applications are allowed to run on the device. Execution permission is determined by the type of certificate the application is signed with (either privileged or normal).

**Execution Permissions and DLLs**

There is one thing you still need to think about. Just like applications, Dynamic Link Libraries can either be unsigned or signed with a privileged or normal certificate. Since DLLs are used within an application this leads to an interesting question. What will happen if for instance a privileged application calls out to an unsigned DLL that, on its own makes a call to a trusted Win32 API? If the DLL would be promoted to get the same access rights as the calling application this would introduce a security hole. Therefore, the following rules are applied to applications that make calls to DLLs, listing what Access Levels are granted to the DLL:

|  |  |  |  |
| --- | --- | --- | --- |
| **Application / DLL Combination** | **No Security** | **One-Tier Prompt** | **Two-Tier Prompt** |
| Unsigned Application |  |  |  |
| Unsigned DLL | Privileged | Privileged if allowed | Unprivileged if allowed |
| Normal Signed DLL | Privileged | Privileged if allowed | Unprivileged if allowed |
| Privileged Signed DLL | Privileged | Privileged if allowed | Unprivileged if allowed |
| Normal Signed Application |  |  |  |
| Unsigned DLL | Privileged | Privileged | Unprivileged if allowed |
| Normal Signed DLL | Privileged | Privileged | Unprivileged |
| Privileged Signed DLL | Privileged | Privileged | Unprivileged |
| Privileged Signed Application |  |  |  |
| Unsigned DLL | Privileged | Privileged | Not allowed! |
| Normal Signed DLL | Privileged | Privileged | Not allowed! |
| Privileged Signed DLL | Privileged | Privileged | Privileged |

**Testing Your Application under Different Device Security Configurations**

One of the tools that are part of Visual Studio 2008 is the Device Security Manager. It allows you to retrieve current security settings on a Windows Mobile device and on the Device Emulator.

**NOTE**: You can only change security settings on physical devices if the current security configuration of the device allows you to do so. Also, you have to be careful setting a physical device to a Locked security configuration, because this is an irreversible operation. The Device Security Manager will give you a warning about this. Using Device Emulator you can safely experiment with all different security configurations.

Using Device Security Manager you can deploy any of the security configurations that are currently defined and you can import new security configurations as well. You can also get a list of all certificates that are installed in the certificate stores on the connected Windows Mobile Device or Device Emulator.

Figure - Device Security Manager

Device Security Configurations can be defined through XML files, just like the following sample that is available as part of Visual Studio 2008:

<wap-provisioningdoc>
 <characteristic type="**SecurityPolicy**">
 <!-- Allow unsigned apps -->
 <parm name="**4102**" value="**1**" />
 <!-- Prompt -->
 <parm name="**4122**" value="**0**" />
 <!-- 2-tier mode -->
 <parm name="**4123**" value="**0**" />
 </characteristic>
</wap-provisioningdoc>

Testing different Access Levels under different Security Configurations

Now you know how to set different security configurations, you might want to experiment with different security settings. The following code snippet helps you to get started writing a very simple application that calls a trusted Win32 API. If you add this snippet both to an application and to a DLL and code sign both the application and the DLL with different certificates and of course run without a certificate at all, you can test behavior of the application under different security configurations.

 [DllImport("coredll.dll")]

 public extern static void PowerOffSystem();

 private void btnInstructions\_Click(object sender, EventArgs e)

 {

 PowerOffSystem();

 }

If the application calling this API is allowed to call trusted APIs, you will see that Device Emulator will disappear immediately, since it is powered down. If the application is not allowed to call trusted APIs, nothing happens. The API is just ignored.

If on the other hand you are writing to protected areas of the File System or the Registry on the device, such calls will result in an **UnauthorizedAccessException**.

**Application Installation**

Device Security also has its impact on application installation. So far in this article we assumed deploying applications through Visual Studio 2008 and setting security configurations through Device Security Manager. These are great options during application development, but at some time you are ready to ship your application. Of course you want to have a great installation experience for your end users, meaning you have to ship your application either as a CAB file, or perhaps even as a Microsoft Installer (MSI) file. If you decide to package your application in a CAB file, the end user has to copy the CAB file to their Windows Mobile device on which they can use the Explorer to unpack and install the application. If you decide to wrap your CAB file into a Microsoft Installer file, users can install your application from the desktop as long as their Windows Mobile device is connected either through ActiveSync (Windows XP) or through the Windows Mobile Device Center (Windows Vista).

If the device has a strict security configuration, it might not allow unsigned CAB files to be installed on the device. This means that you not only have to think about signing your application, but also about signing your CAB file. To automate this process, the Windows Mobile 6 SDKs contain the **cabsigntool.exe**. This utility runs a command prompt and makes use of an underlying tool being **signtool.exe**. Because signtool.exe is part of Visual Studio 2008, it is recommended to start a Visual Studio 2008 command prompt in order to use signtool.exe without having to setup environment variables yourself to find this tool.

Here is an example of how to use **cabsigntool.exe** from a command prompt:

cabsigntool "C:\Users\UserVS2008\Documents\Visual Studio 2008\Projects\HOL6SampleApp\DVDsMobileCAB\Debug\DVDsMobileCAB.cab" "C:\Users\UserVS2008\Documents\Visual Studio 2008\Projects\HOL6SampleApp\DVDsMobileCAB\Debug\DVDsMobileCAB.cab" -f "C:\Program Files\Windows Mobile 6 SDK\Tools\Security\SDK Development Certificates\SampleUnPrivDeveloper.pfx"

In this example, the CAB file DVDsMobileCab.cab and all of the executables and assemblies it contains are signed with a unprivileged test certificate that can be found in the Windows Mobile 6 SDK. The **cabsigntool.exe** itself can also be found in the Windows Mobile 6 SDK, under the following path: c:\program files\Windows Mobile 6 SDK\Tools\Security. As you can see, entering command line parameters for **cabsigntool.exe** is error prone. Therefore it might be better to create a batch file containing the entire command, especially if you want to run **cabsigntool.exe** more often.

**Updating Applications**

Unlike Desktop Applications that are developed with Visual Studio 2008, Smart Device applications can’t be deployed using ClickOnce technology that allows functionality for self updating applications if newer versions of an application become available. You can create similar functionality yourself, for instance through a Web Service. In this whitepaper you will get a high level overview of how to create an update mechanism that works similar to ClickOnce. The sample is just a starting point, but should give you enough information and ideas to create your own update mechanism for your applications.

In order to be able to update an application automatically, you can for instance make use of a Web Service with methods to return the version number of the latest published build of the application and compare them to the version number of the current build that is installed on the Windows Mobile Device. Since you need to update the running application, you need a separate installer process in order to do that. You can even use the same installer process to initially install the application from scratch.

**Installing a New Application**

To install a brand new auto-updatable application, what you can do is deploy the application through an ASP.NET website. In order to download the application, the user should browse to the Web Site where the application is available for download.

Figure - Web Site that contains the application and downloading the application updater

In figure 2 you see a Web Site where a new application is available for download. By clicking on the link, the user will not install the application itself, but instead they will install an application installer on their device. The application installer will be stored in exactly the same folder where the application will be installed as well. The installer also creates a short cut in the **Program Files** folder on the device with the name of the actual application.

Once the application updater is downloaded, the user can start the application updater by just navigating to **Programs** and click on the short cut that is already available for the application. Since the application itself is not yet available, the application updater will start instead of the real application, as shown in Figure 3.

Figure - Starting the application for the first time

The application updater makes use of a command line argument to determine if it was started by the user or from inside the actual application. The latter is used when updates of the application are available on the website. The following code, which is part of the application updater and runs on a **Form\_Load** event shows how to distinguish between initial installations and update installations:

 private void MainForm\_Load(object sender, EventArgs e)

 {

 if (\_args == null || \_args.Length == 0 || \_args[0].Length == 0)

 {

 \_args = new string[1];

 \_args[0] = initialURL;

 label1.Text = Properties.Resources.InstallString;

 }

 else

 {

 label1.Text = Properties.Resources.UpgradeString;

 }

 }

When the application is initially installed it will update the short cut that is stored in the **Program Files** folder, so the next time the user starts the application from inside the **Program Files** folder, the actual application will start, not the application updater.

Figure - Installing the application and starting it from the application updater

Once the application is installed through the application installer, the application will be started automatically by the application installer as long as the user terminates the installer through **Menu | Exit**. The following code shows how to start the main application from inside the application installer. Note that this code snippet contains hard coded paths, not the best practice and easy to improve, depending on your own requirements:

 private void menuExit\_Click(object sender, EventArgs e)

 {

 if (label1.Text.Equals(Properties.Resources.InstallString) && ! appInstalled)

 {

 Close();

 }

 else

 {

 StartDVDsMobile();

 }

 }

 private void StartDVDsMobile()

 {

 Process theApp = null;

 try

 {

 theApp = Process.Start(@"\Program Files\DVDsMobile\DVDsMobile.exe",

 Process.GetCurrentProcess().Id.ToString());

 }

 catch (Win32Exception exc)

 {

 if (exc.NativeErrorCode == ERROR\_FILE\_NOT\_FOUND)

 {

 MessageBox.Show("DVDsMobile.exe not found.");

 }

 else if (exc.NativeErrorCode == ERROR\_ACCESS\_DENIED)

 {

 MessageBox.Show("No permissions to start DVDsMobile.exe.");

 }

 }

 }

In the above code snippet you can see that the **menuExit\_Click** event handler is used to determine to either automatically start the application or simply terminates. If a new version of the application is installed, the application updater starts the new version of the application, passing it its own process identification, so the application can terminate the application updater. Even though this is not necessary, it will give the end user a better experience, because the end user hardly notices that a new process is started before the current process is terminated.

**Updating the Application**

Each time the user starts the application, it will call out to a Web Service that is hosted on the same site where the application updater was installed from. The Web Service has methods to return the latest version of all assemblies that are part of the application. The application compares its own version numbers with the version numbers returned from the Web Service calls. If there are newer versions available on the hosting website, the user will be prompted to either install the latest version or run the current version. The downside of this approach is that it takes a little longer to start the application. The upside is that a user can always make use of the latest version available. To keep the sample code simple, the application is not checking if a network connection is available, something that should of course be added for a real application. This can be easily done by making use of State & Notification Broker functionality.

Figure - Auto upgrading in action

To download a new version of the application, the application updater visits the Web Site and downloads the latest version through a hidden Web Browser control. Making use of a Web Browser control simplifies downloading a file to the device without needing additional Web Service methods[[2]](#footnote-3). The following code shows you how the application finds out that a new version of itself is available:

 private void MainFormNoTouch\_Load(object sender, EventArgs e)

 {

 this.Text = Properties.ResourcesNoTouch.MainFormNoTouchTitle;

 this.menuItemExit.Text = Properties.ResourcesNoTouch.MenuItemExitText;

 if (\_processID != -1)

 {

 // We have just been updated, so kill our update process.

 Process.GetProcessById(\_processID).CloseMainWindow();

 }

 else

 {

 DVDsMobileUpdateService.DVDsMobileUpdateService updateService =

 new DVDsMobileNoTouch.DVDsMobileUpdateService.DVDsMobileUpdateService();

 string[] latestDVDsMobileVersion = updateService.DVDsMobileVersion().Split(

 new char[] { '.' });

 string[] latestDVDsMobileVersionNoTouch = updateService.DVDsMobileVersionNoTouch().Split(

 new char[] {'.'});

 string[] currentDVDsMobileVersion = \_callingAssembly.Split(new char[] { '.' });

 Version currentDVDsMobileVersionNoTouch =

 Assembly.GetExecutingAssembly().GetName().Version;

 bool update = false;

 update = (Convert.ToInt32(latestDVDsMobileVersion[0]) >

 Convert.ToInt32(currentDVDsMobileVersion[0])) ||

 (Convert.ToInt32(latestDVDsMobileVersion[1]) >

 Convert.ToInt32(currentDVDsMobileVersion[1])) ||

 (Convert.ToInt32(latestDVDsMobileVersionNoTouch[0]) >

 currentDVDsMobileVersionNoTouch.Major) ||

 (Convert.ToInt32(latestDVDsMobileVersionNoTouch[1]) >

 currentDVDsMobileVersionNoTouch.Minor);

 if (update)

 {

 if (MessageBox.Show(

 "A new version of DVDsUpdate is found! Do you want to install the new version?",

 "New version available",

 MessageBoxButtons.YesNo,

 MessageBoxIcon.Question,

 MessageBoxDefaultButton.Button2) == DialogResult.Yes)

 {

 Process.Start(@"\Program Files\DVDsMobile\DVDsMobileAutoUpdater.exe",

 @"http://192.168.2.102:5746/DownloadForm.aspx");

 Application.Exit();

 }

 }

 }

 }

In the above shown code snippet you can see that the Form Load event handler is used to determine if a new version of the application is available. In order to find this out, the application calls two methods on the **DVDsMobileUpdateService** Web Service. The application also reads its own version numbers, if there is a newer version available, the user sees a message box through which they can decide to either install the latest version or continue running the current version. If the user decides to run the latest version, immediately the application updater is started and the application terminates itself. Once a new version of the application is installed on the device, the application updater starts that new version of the application, allowing the user to start working with the application.

# **Additional resources and references**

* [Window Mobile Developer Center](http://msdn.microsoft.com/en-us/windowsmobile/default.aspx)
* [Windows Mobile Device Security Model](http://msdn.microsoft.com/en-us/library/bb416353.aspx)
* [MSDN Webcast: Windows Mobile Device Security](http://www.microsoft.com/events/series/detail/webcastdetails.aspx?seriesid=86&webcastid=5596)
* [Deployment, Setup, Security and You](http://msdn.microsoft.com/en-us/library/bb158571.aspx)
* [Video: How do I Implement Universal Download Capabilities](http://msdn.microsoft.com/en-us/netframework/bb663033.aspx)
* [Video: How do I Create a Desktop Project for Installing CAB Files to a Windows Mobile Device](http://msdn.microsoft.com/en-us/windowsmobile/bb649500.aspx)
1. At the time of writing this white paper, Mobile2Market is about to be replaced by [Windows Mobile Application Readiness](http://msdn.microsoft.com/en-us/windowsmobile/dd569132.aspx). [↑](#footnote-ref-2)
2. The code for the Web Site is not shown in this whitepaper, but you will be able to create this exact installation experience in the Windows Mobile RampUp part 7, where the Hands-On-Lab takes you step-by-step through this entire sample. [↑](#footnote-ref-3)