Beyond good ol' Run key, Part 3

Possible Autostart/start mechanisms that are built-in 'natively' in Windows and also available by means of extra features offered by many applications go beyond typical path locations and registry keys highlighted by popular programs and scripts like Autoruns and SilentRunners. I have covered some of the non-standard persistence techniques in 2 older posts in the series here and here, but as usual – there is always more to write about.

In this post I will cover another batch of less known and possibly 'obscure' technique that could be potentially used for autostart/start purposes. I write 'obscure', because it is not a typical way of doing autostart, but let's be honest – there is nothing really extraordinary about it – just a simple abuse of built-in features in both OS and applications.

CODE-IN-THE-MIDDLE PROXY

Long story short, it is a well known fact that many existing registry entries and files pointing to or containing code can be modified to introduce a code-in-the-middle proxy (DLL, EXE, etc.) that will be executed/loaded first instead of a legitimate entry. The original entries are preserved so that they can be transparently executed/loaded once malware is running. There are many existing examples of this technique already being used e.g. hijacks of Shell Open Command, Image File Execution Options , etc., but it is important to remember that this technique can be extended literally to any registry key or file that is loaded either during autostart or often used by users.

APPLICATION REGISTRATION (APP PATHS) HIJACKING

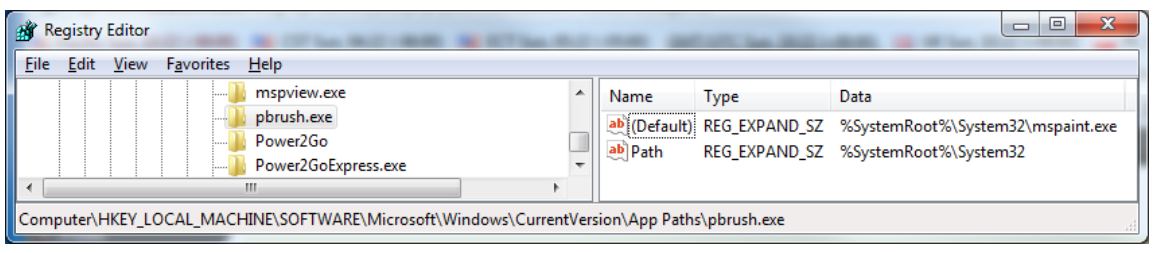
Another proxy technique that could be used to hijack popular applications relies on registry entries stored under the following key:

> HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\App Paths

As per Microsoft:

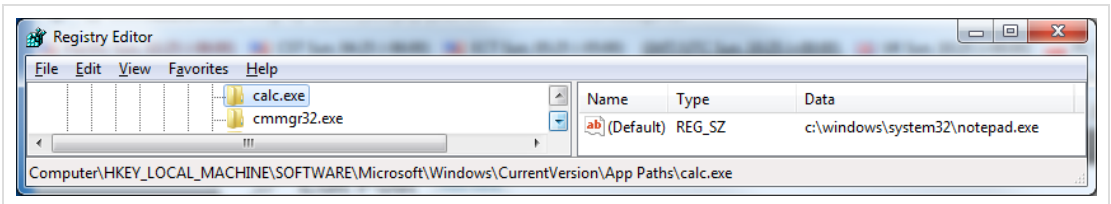
> The entries found under App Paths are used primarily for the following purposes:
>
> - To map an application's executable file name to that file's fully qualified path.
> - To pre-pend information to the PATH environment variable on a per-application, per-process basis.

A legitimate entry that can be found on many newer versions of Windows is shown below:



It is responsible for launching MS Paint program when someone tries to run it using a legacy 'pbrush.exe' name.

One could add a modification for e.g. calc.exe:



From now on, anytime someone tries to run calc.exe manually (e.g. via Start Menu/Run window), Notepad will be launched. It may not be a main persistence mechanism, but could be used for re-infection purposes on systems that have been cleaned up, but not rebuilt.

You can test it (XP needed) by downloading this reg file , then applying it to your Registry and then launching Win+R and typing 'calc' or 'calc.exe' and hitting enter. Note: It doesn't work from command line (a mistake in an older version of this post which I correct here).

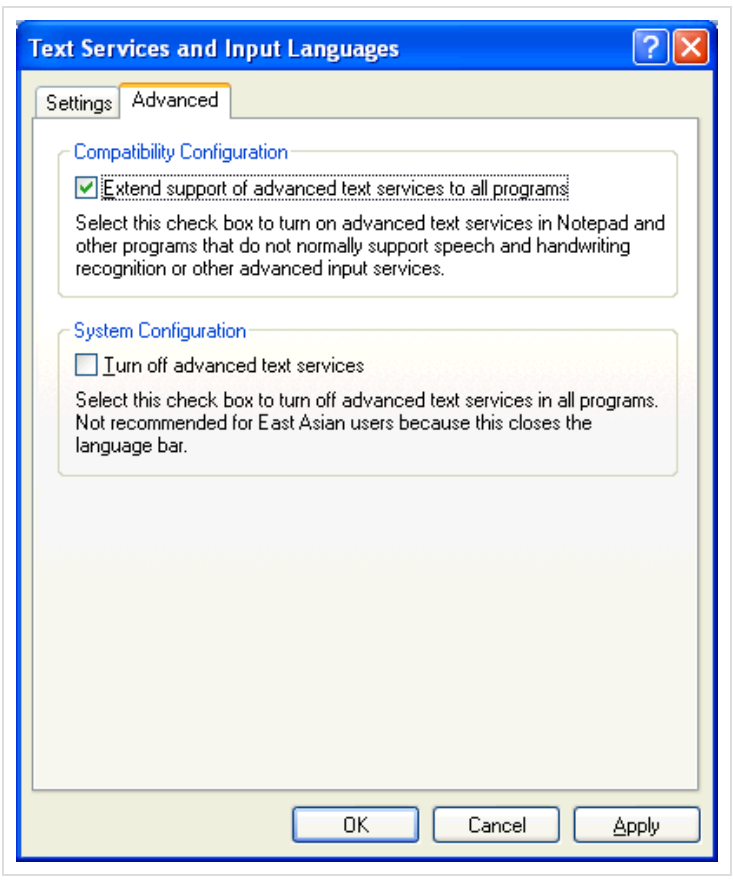
TEXT SERVICES (TSF)

Microsoft defines Text Services as:

> Microsoft Windows Text Services Framework (TSF) is a system service available as a redistributable for Windows 2000. TSF provides a simple and scalable framework for the delivery of advanced text input and natural language technologies. TSF can be enabled in applications, or as a TSF text service. A TSF text service provides multilingual support and delivers text services such as keyboard processors, handwriting recognition, and speech recognition.

From a practical point of view, TSF offers ways to extend available input methods by allowing to install support for languages that are not natively supported by Windows. A good example of such extension is Ekaya – an extension for a Myanmar (Burmese) language.

In order for TSF to work on Windows XP, one has to enable the 'Extended support of advanced text services to all programs':



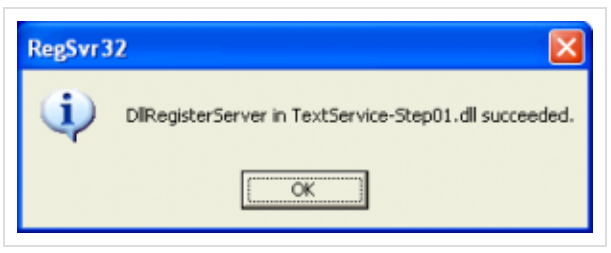
On Windows 7, it is enabled by default (but to install a TSF DLL one requires administrator privileges).

Examples on how to use TSF are provided in Microsoft SDK (look for 'Samples\winui\Input\tsf\TextService' directory). For the purpose of this article, I just picked up the simplest possible example i.e. a project from the 'Samples\winui\Input\tsf\TextService\TextService-Step01' subdirectory and updated it with a trivial cosmetic change – a call to OutputDebugString so that we can observe processes loading and unloading our test DLL.

```
BOOL WINAPI DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID pvRese
{
  TCHAR szFileFullPath[256];
  TCHAR buf[300];
    switch (dwReason)
    {
        case DLL_PROCESS_ATTACH:

            g_hInst = hInstance;
            GetModuleFileName (NULL,szFileFullPath,256);
             _tcscpy (buf, TEXT("TSF DLL loaded: "));
             _tcscat (buf, szFileFullPath);
            OutputDebugString(buf);
            if (!InitializeCriticalSectionAndSpinCount(&g_cs, 0))
                return FALSE;

            break;

        case DLL_PROCESS_DETACH:

            GetModuleFileName (NULL,szFileFullPath,256);
             _tcscpy (buf, TEXT("TSF DLL unloaded: "));
             _tcscat (buf, szFileFullPath);
            OutputDebugString(buf);

            DeleteCriticalSection(&g_cs);

            break;
    }

    return TRUE;
}
```

Once registered with Regsrv32.exe:

```
regsvr32 TextService-Step01.dll
```



the DLL is now active and it will now be loaded to each new process utilizing Text Services (pretty much every single GUI application, including these already running) as can be shown via DebugView from Sysinternals.

Running a few test applications shows the following output in DebugView:

#	Time	Debug Print
0	0.00000000	[3480] TSF DLL loaded: C:\WINDOWS\system32\regsvr32.exe
1	0.94201440	[3480] TSF DLL unloaded: C:\WINDOWS\system32\regsvr32.exe
2	11.72093868	[2624] TSF DLL loaded: C:\WINDOWS\system32\notepad.exe
3	13.40614605	[2624] TSF DLL unloaded: C:\WINDOWS\system32\notepad.exe
4	15.43769360	[1664] TSF DLL loaded: C:\WINDOWS\Explorer.EXE
5	16.22678566	[1664] TSF DLL unloaded: C:\WINDOWS\Explorer.EXE
6	16.27532196	[3296] TSF DLL loaded: C:\WINDOWS\system32\calc.exe
7	18.75527573	[3296] TSF DLL unloaded: C:\WINDOWS\system32\calc.exe
8	25.77335167	[3576] TSF DLL loaded: C:\Program Files\Internet Explorer\IEXPLORE.EXE
9	28.00358963	[3576] TSF DLL unloaded: C:\Program Files\Internet Explorer\IEXPLORE.EXE

Of course, it survives the reboot and is loaded next time user logs on and applications are executed + it works under Windows 7 without any problem:

#	Time	Debug Print
1	0.00000000	[2508] TSF DLL loaded: C:\Windows\SysWOW64\regsvr32.exe
2	2.12864447	[2508] TSF DLL unloaded: C:\Windows\SysWOW64\regsvr32.exe
3	3.85741568	[336] TSF DLL loaded: C:\Users\someguy\Desktop\dv.exe
4	88.90030670	[1380] TSF DLL loaded: c:\windows\SysWOW64\calc.exe
5	90.92218018	[1380] TSF DLL unloaded: c:\windows\SysWOW64\calc.exe
6	98.02786255	[840] TSF DLL loaded: c:\windows\SysWOW64\notepad.exe
7	99.93174744	[840] TSF DLL unloaded: c:\windows\SysWOW64\notepad.exe

You may be wondering if there is any visual indication of the DLL being present on the system.

There is.

If you look at the legitimate software like aforementioned Ekaya – it adds a set of icons to the Language Bar:

and

It can be also seen in Text Services and Input Languages section (you can find it under Regional Settings):

There is no requirement for TSF DLLs to add extra features to the Language Bar, so the Text Services and Input Languages section under Regional Settings is the only place where it is possible to spot the loaded DLL – for our test sample it looks like this:

DLL LOAD ORDER

This is a trick relying on DLL load order – it has been covered on many security blogs in last 2 years so I just mention it for completeness – there are many DLLs that can be 'injected' into a loading process of many popular programs. Two of them: **fxsst.dll** and **ntshrui.dll** have been covered by Nick Harbour from Mandiant in his posts from July 2010 and June 2011.

IIS SERVER EXTENSIONS (ISAPI FILTERS)

In my older post I mentioned plugins and various extensions that can be loaded into various applications. There are really a lot of possibilities here, including multum of popular software, Windows Shell extensions, aforementioned Text services, IME, URL handlers, and so on and so forth. There are also possibilities of writing server environment-specific extensions e.g. ISAPI filters: As per the information on the page

> Every ISAPI filter is contained in a separate DLL that must export two entry-point functions, GetFilterVersionand HttpFilterProc, and optionally export the TerminateFilter function. The metabase property, FilterLoadOrder, contains a list of all filters that IIS loads when the Web service is started.

APPCERTDLLS

This is also a known technique – it has been researched and published by EP_X0FF in 2007 on sysinternals forum. There were a few follow-up posts about it, and a sample code can be found here, here and here. If you are interested you may also read ReactOS code that implements this feature here (search for 'BaseIsProcessAllowed').

Using a slightly modified code from one of the posts, we can build a DLL to demonstrate how it works.

First we need to add a registry key:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Ap
```

then any REG_EXPAND_SZ value pointing to a DLL we have built.

Now we need to restart the system to ensure a system-wide coverage. For testing purposes, it is okay to restart Windows Explorer so that it can refresh its internal program state to include these DLL in a process creation sequence. Or, one can simply launch cmd.exe and then run programs from command line to observe the DLL being loaded into each newly created process:

You may be wondering how it works under 64-bit system. It works pretty well.

In fact, you can register both 32-bit and 64-bit DLLs as a notification on a 64-bit system:

to ensure notifications will be processed for both 64-bit and 32-bit programs:

That's all ! Thanks for reading!

This entry was posted in **Anti-Forensics**, **Autostart (Persistence)**, **Compromise Detection**, **Forensic Analysis**, **Malware Analysis** by **adam**. Bookmark the **permalink**.