Interaction between services and applications of user level in Windows Vista

Author:
Yuri Maxiutenko,
Software Developer of Apriorit Inc.

This article is devoted to the question about working with services and applications in Windows Vista. In particular we'll consider how to start an interactive user-level application from the service and how to organize the data exchange between the service and applications. Solutions are given both for C++ and C#. This article might be useful for those who deal with the task of organizing the interaction between the service and the application on Windows Vista using both managed and native code.

Windows Vista, services and desktop

Before Vista appearance services and user applications in operating systems of Windows family could jointly use session 0. It was possible to easily open windows on the desktop of the current user directly from the service, and also to exchange data between the service and applications by means of window messages. But it became the serious security problem when the whole class of attacks appeared that used windows opened by the services to get access to the services themselves. The mechanism of counteraction to such attacks appeared only in Vista.

In Windows Vista all user logins and logouts are performed in the sessions that differ from the session 0. The possibility of opening windows on the user desktop by the services is very restricted, and if you try to start an application from the service it starts in session 0. Correspondingly if this application is interactive you have to switch to the desktop of session 0. The using of the window messages for data exchange is made considerably harder.

Such security policy is quite defensible. But what if nevertheless you need to start an interactive application on the user desktop from the service? This article describes one of the possible solution variants for this question. Moreover we'll consider several ways of organization of data exchange between services and applications.

Starting interactive applications from the service

As soon as the service and desktop of the current user exist in the different sessions the service will have to “feign” this user to start the interactive applications. To do so we should know the corresponding login name and password or have the LocalSystem account. The second variant is more common so we’ll consider it.

So, we create the service with the LocalSystem account. First we should get the token of the current user. In order to do it we:

1) get the list of all terminal sessions;
2) choose the active session;
3) get token of the user logged to the active session;
4) copy the obtained token.

C++ code

You can see the corresponding code in C++ below.

```
PHANDLE GetCurrentUserToken()
{
    PHANDLE currentToken = 0;
    PHANDLE primaryToken = 0;

    int dwSessionId = 0;
    PHANDLE hUserToken = 0;
    PHANDLE hTokenDup = 0;

    PWTS_SESSION_INFO pSessionInfo = 0;
    DWORD dwCount = 0;

    // Get the list of all terminal sessions
WTSEnumerateSessions(WTS_CURRENT_SERVER_HANDLE, 0, 1, &pSessionInfo,
&dwCount);

    int dataSize = sizeof(WTS_SESSION_INFO);

    // look over obtained list in search of the active session
    for (DWORD i = 0; i < dwCount; ++i)
    {
        WTS_SESSION_INFO si = pSessionInfo[i];
        if (WTSActive == si.State)
        {
            // If the current session is active – store its ID
            dwSessionId = si.SessionId;
            break;
        }
    }

    // Get token of the logged in user by the active session ID
    BOOL bRet = WTSQueryUserToken(dwSessionId, currentToken);
    if (bRet == false)
    {
        return 0;
    }

    bRet = DuplicateTokenEx(currentToken, TOKEN_ASSIGN_PRIMARY |
TOKEN_ALL_ACCESS, 0, SecurityImpersonation, TokenPrimary, primaryToken);
    if (bRet == false)
```

```
    {
        return 0;
    }

    return primaryToken;
}
```

It sould be mentioned that you can use WTSGetActiveConsoleSessionId() function instead of the looking over the whole list. This function returns the ID of the active session. But when I used it for the practical tasks I discovered that this function doesn't always work while the variant with looking through the all sessions always gave the correct result.

If there are no logged in users for the current session then the function WTSQueryUserToken() returns FALSE with error code ERROR_NO_TOKEN. Naturally you can't use the code given below in this case.

After we've got the token we can start an application on behalf of the current user. Pay attention that the rights of the application will correspond to the rights of the current user account and not the LocalSystem account.

The code is given below.

```
BOOL Run(const std::string& processPath, const std::string& arguments)
{
    // Get token of the current user
    PHANDLE primaryToken = GetCurrentUserToken();
    if (primaryToken == 0)
    {
        return FALSE;
    }
    STARTUPINFO StartupInfo;
    PROCESS_INFORMATION processInfo;
    StartupInfo.cb = sizeof(STARTUPINFO);

    SECURITY_ATTRIBUTES Security1;
    SECURITY_ATTRIBUTES Security2;

    std::string command = "\"" + processPath + "\"";
    if (arguments.length() != 0)
    {
        command += " " + arguments;
    }

    void* lpEnvironment = NULL;

    // Get all necessary environment variables of logged in user
    // to pass them to the process
    BOOL resultEnv = CreateEnvironmentBlock(&lpEnvironment, primaryToken,
FALSE);
    if (resultEnv == 0)
    {
        long nError = GetLastError();
    }

    // Start the process on behalf of the current user
    BOOL result = CreateProcessAsUser(primaryToken, 0,
(LPSTR)(command.c_str()), &Security1, &Security2, FALSE, CREATE_NO_WINDOW |
NORMAL_PRIORITY_CLASS | CREATE_UNICODE_ENVIRONMENT, lpEnvironment, 0,
&StartupInfo, &processInfo);
    CloseHandle(primaryToken);
    return result;
```

```
}
```

If the developed software will be used only in Windows Vista and later OSs then you can use CreateProcessWithTokenW() function instead of the CreateProcessAsUser(). It can be called for example in this way:

```
    BOOL result = CreateProcessWithTokenW(primaryToken, LOGON_WITH_PROFILE,
0, (LPSTR)(command.c_str()), CREATE_NO_WINDOW | NORMAL_PRIORITY_CLASS |
CREATE_UNICODE_ENVIRONMENT, lpEnvironment, 0, &StartupInfo, &processInfo);
```

C# code

Let's implement the same functionality in C#. We create the class ProcessStarter that will be used in some subsequent examples. Here I describe only two main methods.

```
public static IntPtr GetCurrentUserToken()
{
    IntPtr currentToken = IntPtr.Zero;
    IntPtr primaryToken = IntPtr.Zero;
    IntPtr WTS_CURRENT_SERVER_HANDLE = IntPtr.Zero;

    int dwSessionId = 0;
    IntPtr hUserToken = IntPtr.Zero;
    IntPtr hTokenDup = IntPtr.Zero;

    IntPtr pSessionInfo = IntPtr.Zero;
    int dwCount = 0;

    WTSEnumerateSessions(WTS_CURRENT_SERVER_HANDLE, 0, 1, ref pSessionInfo,
ref dwCount);

    Int32 dataSize = Marshal.SizeOf(typeof(WTS_SESSION_INFO));

    Int32 current = (int)pSessionInfo;
    for (int i = 0; i < dwCount; i++)
    {
        WTS_SESSION_INFO si =
(WTS_SESSION_INFO)Marshal.PtrToStructure((System.IntPtr)current,
typeof(WTS_SESSION_INFO));
        if (WTS_CONNECTSTATE_CLASS.WTSActive == si.State)
        {
            dwSessionId = si.SessionID;
            break;
        }

        current += dataSize;
    }

    bool bRet = WTSQueryUserToken(dwSessionId, out currentToken);
    if (bRet == false)
    {
        return IntPtr.Zero;
    }

    bRet = DuplicateTokenEx(currentToken, TOKEN_ASSIGN_PRIMARY |
TOKEN_ALL_ACCESS, IntPtr.Zero,
SECURITY_IMPERSONATION_LEVEL.SecurityImpersonation, TOKEN_TYPE.TokenPrimary,
out primaryToken);
```

```
    if (bRet == false)
    {
        return IntPtr.Zero;
    }

    return primaryToken;
}

public void Run()
{
    IntPtr primaryToken = GetCurrentUserToken();
    if (primaryToken == IntPtr.Zero)
    {
        return;
    }
    STARTUPINFO StartupInfo = new STARTUPINFO();
    processInfo_ = new PROCESS_INFORMATION();
    StartupInfo.cb = Marshal.SizeOf(StartupInfo);

    SECURITY_ATTRIBUTES Security1 = new SECURITY_ATTRIBUTES();
    SECURITY_ATTRIBUTES Security2 = new SECURITY_ATTRIBUTES();

    string command = "\"" + processPath_ + "\"";
    if ((arguments_ != null) && (arguments_.Length != 0))
    {
        command += " " + arguments_;
    }

    IntPtr lpEnvironment = IntPtr.Zero;
    bool resultEnv = CreateEnvironmentBlock(out lpEnvironment, primaryToken,
false);
    if (resultEnv != true)
    {
        int nError = GetLastError();
    }

    CreateProcessAsUser(primaryToken, null, command, ref Security1, ref
Security2, false, CREATE_NO_WINDOW | NORMAL_PRIORITY_CLASS |
CREATE_UNICODE_ENVIRONMENT, lpEnvironment, null, ref StartupInfo, out
processInfo_);
    CloseHandle(primaryToken);
}
```

Also we must mention that there is a very good article about the launching of the user-level application from the service with the LocalSystem account privivleges. It is located here: http://www.codeproject.com/KB/vista-security/VistaSessions.aspx

Data exchange between the service and application

It remained only to solve the problem of data exchange between the service and applications. You can use a number of options: sockets, named memory mapped files, RPC and COM. Here we will consider the three easiest ways: text files, events (for C#) and named pipes (for C++).

Text files

One of the simplest solutions is to use text files. When we talk about C# development the most natural is to use XML files.

For example, we must pass some data string from the user-level application to the service. For a start, we must decide where the mediator file should be created. The location must be accessible both for the application and the service.

If the application was started with the permissions of the current logged-in user, the good solution would be to use the “My documents” folder of that user. In this case there will be no access problems from the both sides (as the LocalSystem service has the permissions to access almost everywhere).

So, let’s create the XML-file “sample.xml” in the current user’s “My Documents” folder:

```
using System.Xml;

XmlWriterSettings xmlWriterSettings = new XmlWriterSettings();
// provide the XML declaration
xmlWriterSettings.OmitXmlDeclaration = false;
// write attributes on the new line
xmlWriterSettings.NewLineOnAttributes = true;
// indent elements
xmlWriterSettings.Indent = true;
// get “My Documents” folder path
String myDocumentsPath =
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
String sampleXmlFilePath = Path.Combine(myDocumentsPath,”sample.xml”);
// create the XML file “sample.xml”
sampleXmlWriter = XmlWriter.Create(sampleXmlFilePath, xmlWriterSettings);
```

Now we will create the “SampleElement” element which some useful data would be passed to:

```
sampleXmlWriter.WriteStartElement("SampleElement");
sampleXmlWriter.WriteElementString("Data", "Hello");
```

Let’s finish the file creation:

```
sampleXmlWriter.WriteEndElement();
sampleXmlWriter.Flush();
sampleXmlWriter.Close();
```

And now the service must open that file. To have the access to it the service must get the current user’s “My Documents” folder path. In order to do it we should make an impersonation by means of the operation of getting the token described above:

```
// Get token of the current user
IntPtr currentUserToken = ProcessStarter.GetCurrentUserToken();
// Get user ID by the token
WindowsIdentity currentUserId = new WindowsIdentity(currentUserToken);
// Perform impersonation
WindowsImpersonationContext impersonatedUser =
currentUserId.Impersonate();
// Get path to the ”My Documents”
String myDocumentsPath =
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
// Make everything as it was
impersonatedUser.Undo();
```

Now the service can read the data from the “sample.xml” file:

```
String sampleXmlFilePath = Path.Combine(myDocumentsPath,”sample.xml”);
XmlDocument oXmlDocument = new XmlDocument();
oXmlDocument.Load(sampleXmlFilePath);

XPathNavigator oPathNavigator = oXmlDocument.CreateNavigator();
XPathNodeIterator oNodeIterator =
oPathNavigator.Select("/SampleElement/Data");
oNodeIterator.MoveNext();

String receivedData = oNodeIterator.Current.Value;
```

Data exchange via the text files is very simple for implementation but it has a number of disadvantages. There can be not enough disk space; user can directly meddle in the data record process etc. So let’s consider another ways.

Events

In the trivial case when we need to transmit only information of “yes/no” type (answer on the dialog window question, message if the service should be stopped or not and so on) we can use Events.

Let’s consider an example. The functioning of the some application “sample” should be paused in the certain point until the service gives a command to continue.

The “sample” application is started from the service by means of the already considered class ProcessStarter (in C#):

```
ProcessStarter sampleProcess = new ProcessStarter();
sampleProcess.ProcessName = "sample";
sampleProcess.ProcessPath = @"C:\Base\sample.exe";
sampleProcess.Run();
```

Now we create the global event SampleEvent in that point of the “sample” application where it should stop and wait for the command from the service. We stop the thread until the signal comes:

```
using System.Threading;


EventWaitHandle sampleEventHandle = new EventWaitHandle(false,
EventResetMode.AutoReset, "Global\\SampleEvent");
bool result = sampleEventHandle.WaitOne();
```

We open the global event SampleEvent in that point of the service where it’s necessary to send the command to the application. We set this event to the signal mode:

```
EventWaitHandle handle = EventWaitHandle.OpenExisting("Global\\SampleEvent");
bool setResult = handle.Set();
```

Application gets this signal and continues its functioning.

Named pipes

If we talk about big volumes of data in exchange process we can use named pipe technology. We must mention that the code below is provided in C++ as the classes for working

with the named pipes in C# were introduced only since .NET Framework 3.5. If you want to know how to use these new .NET tools for working with the named pipes you can read, for example, this article:
http://social.msdn.microsoft.com/Forums/en-US/csharpgeneral/thread/23dc2951-8b59-48e4-89fe-d2b435db48c6

Let's suppose that an application periodically needs to send some number of unsigned int type to the service.

In this case we can open the named pipe on the service side and then monitor its state in the separated thread to read and process data when they come. So we create the pipe DataPipe in the service code:

```
HANDLE CreatePipe()
{
	SECURITY_ATTRIBUTES sa;
	sa.lpSecurityDescriptor =
(PSECURITY_DESCRIPTOR)malloc(SECURITY_DESCRIPTOR_MIN_LENGTH);
	if (!InitializeSecurityDescriptor(sa.lpSecurityDescriptor,
SECURITY_DESCRIPTOR_REVISION))
	{
		DWORD er = ::GetLastError();
	}
	if (!SetSecurityDescriptorDacl(sa.lpSecurityDescriptor, TRUE, (PACL)0,
FALSE))
	{
		DWORD er = ::GetLastError();
	}
	sa.nLength = sizeof sa;
	sa.bInheritHandle = TRUE;

	// To know the maximal size of the received data for reading from the
	// pipe buffer

	union maxSize
	{
		UINT   _1;
	};
	HANDLE hPipe = ::CreateNamedPipe((LPSTR)"\\\\.\\pipe\\DataPipe",
PIPE_ACCESS_INBOUND, PIPE_TYPE_MESSAGE | PIPE_READMODE_MESSAGE | PIPE_WAIT,
PIPE_UNLIMITED_INSTANCES, sizeof maxSize, sizeof maxSize,
NMPWAIT_USE_DEFAULT_WAIT, &sa);
	if (hPipe == INVALID_HANDLE_VALUE)
	{
		DWORD dwError = ::GetLastError();
	}
	return hPipe;
}
```

We also create the function to check the thread state and perform reading if required:

```
unsigned int __stdcall ThreadFunction(HANDLE& hPipe)
{
	while (true)
	{
		BOOL  bResult = ::ConnectNamedPipe(hPipe, 0);
		DWORD dwError = GetLastError();

		if (bResult || dwError == ERROR_PIPE_CONNECTED)
		{
			BYTE  buffer[sizeof UINT] = {0};
```

```
                DWORD read = 0;

                UINT   uMessage = 0;

                if (!(::ReadFile(hPipe, &buffer, sizeof UINT, &read, 0)))
                {
                      unsigned int error = GetLastError();
                }
                else
                {
                      uMessage = *((UINT*)&buffer[0]);
                      // The processing of the received data
                }
                ::DisconnectNamedPipe(hPipe);
          }
          else
          {

          }
          ::Sleep(0);
      }
}
```

And finally start the separate thread with ThreadFunction() function:

```
       unsigned int id = 0;
       HANDLE pipeHandle = CreatePipe();
       ::CloseHandle((HANDLE)::_beginthreadex(0,  0,  ThreadFunction,  (void*)
pipeHandle, 0, &id));
```

Now we go to the application side and organize sending of data to the service via named pipe.

```
SendDataToService(UINT message)
{
      HANDLE hPipe = INVALID_HANDLE_VALUE;
      DWORD  dwError = 0;
      while (true)
      {
            hPipe = ::CreateFile((LPSTR)"\\\\.\\pipe\\DataPipe",
GENERIC_WRITE, 0, 0, OPEN_EXISTING, 0, 0);
            dwError = GetLastError();
            if (hPipe != INVALID_HANDLE_VALUE)
            {
                  break;
            }

            // If any error except the ERROR_PIPE_BUSY has occurred,
            // we should return FALSE.
            if (dwError != ERROR_PIPE_BUSY)
            {
                  return FALSE;
            }
            // The named pipe is busy. Let's wait for 20 seconds.
            if (!WaitNamedPipe((LPSTR)"\\\\.\\pipe\\DataPipe", 20000))
            {
                  dwError = GetLastError();
                  return FALSE;
            }
      }
```

```
	DWORD dwRead = 0;
	if (!(WriteFile(hPipe, (LPVOID)&message, sizeof UINT, &dwRead, 0)))
	{
		CloseHandle(hPipe);
		return FALSE;
	}
	CloseHandle(hPipe);
	::Sleep(0);
	return TRUE;
}
```

Conclusion

There is no one and only right solution for the problem of interaction between services and applications in Windows Vista. There are a lot of mechanisms and you should choose the proper one in correspondence with the concrete problem. Unfortunately a lot of variants of such interaction organization were left behind this article scope. The usage of some of such technologies in the terms of C# are discussed, for example, in this article: http://www.codeproject.com/KB/threads/csthreadmsg.aspx

To learn this question deeper and a lot of features of developing for Windows Vista we also recommend the book by Michael Howard, David LeBlanc - Writing Secure Code for Windows Vista (Microsoft Press, 2007).