Windows RunTime
QuarksLab

13 juin 2012

Sébastien Renaud
srenaud@quarkslab.com
Kévin Szkudłapski
kszkudlapski@quarkslab.fr

QuarksLab
71 Avenue des Ternes
75017 Paris
FRANCE

Table des matières

0.1 Introduction

Windows 8 est le nom de code actuel de la prochaine version du système d'exploitation de Microsoft Windows faisant suite à Windows 7. Ce système à venir comporte de nombreux changements par rapport aux versions précédentes. En particulier, il ajoute la prise en charge des microprocesseurs ARM en plus des architectures précédemment prises en compte que sont IA-32 et x86-64 [1]. Une nouvelle interface, nommée « Metro » [2], a été ajoutée et a été spécialement conçue pour les entrées utilisateur via un écran tactile et/ou un stylet, en plus de périphériques d'entrée plus usuels comme la souris et le clavier.

Cette nouvelle interface repose entièrement sur un moteur d'exécution nommé « Microsoft Windows Runtime » [3].

Ainsi nous présenterons dans ce document ce nouveau moteur « Microsoft Windows Runtime » construit pour le prochain système d'exploitation Microsoft Windows 8 ainsi que certains de ses principes de sécurité sous-jacents.

Cette présentation est le résultat de recherches menées sur deux versions du système nommée respectivement « Windows 8 Developper Preview » et « Windows 8 Consumer Preview ». Au moment d'écrire ces lignes, Microsoft n'a pas encore annoncé une date de sortie précise pour le système d'exploitation Windows 8, mais certains rapports indiquent que le système serait prévu pour une disponibilité générale avant la fin de l'année 2012.

Le « Microsoft Windows Runtime » a été révélé lors de la conférence Microsoft Build le 12 Septembre 2011 et est bien sûr inclus de fait dans les deux versions susnommées puisqu'il en est un constituant fondamental.

Dans une première partie nous allons détailler ce qu'est ce nouveau moteur d'exécution de Windows, dans un sens assez large.

Ensuite, nous discuterons de la manière dont il est possible pour les programmeurs de cibler le nouveau « Windows Runtime » et ce qui constitue ce moteur d'exécution d'un point de vue haut niveau.

Afin d'aller plus loin, dans le chapitre suivant, nous verrons comment le « Windows Runtime », fonctionne – en mode utilisateur – à partir d'un point de vue interne : comment un programme construit au-dessus du « Windows Runtime » est démarré, quels sont les mécanismes qui conduisent à son exécution et comment ce type de processus est sécurisé. Nous parlerons aussi de la « Boutique d'application Windows » (Windows Store) qui est la pièce maîtresse dans le modèle de distribution des applications utilisant le « Windows Runtime ».

Enfin, nous inspecterons l'une des caractéristiques fondamentales du « Windows Runtime » : son bac à sable (*sandbox*). Dans cette partie nous verrons ainsi comment cette *sandbox* est construite au niveau du noyau Windows, la façon dont elle interagit avec le système et quelles sont ses principales caractéristiques. Pour ce faire, nous comparerons le modèle de *sandbox* du « Windows Runtime » avec celle, plus connue, du navigateur Chrome et pourrons voire ainsi quelles sont les différences, les avantages et les inconvénients possibles de chacun de ces modèles.

0.2 Windows Runtime - Présentation

Le « Microsoft Windows Runtime », ou de manière plus commune et plus courte « WinRT », est le nouveau modèle de programmation et framework de programmation par Microsoft qui constitue le cœur des nouvelles applications de type Metro (aussi connu sous le nom « d'application immersives » ou *immersives apps*) dans le prochain système d'exploitation Windows 8. En effet, ce système d'exploitation a une double disposition et vise à la fois le marché des tablettes et le plus habituel environnement complet de bureau Windows sur PC.

WinRT ajoute des fonctionnalités de sécurité pour les applications conçues pour être sûres et pour s'exécuter en environnement contraint et isolé via une *sandbox*. WinRT apporte des fonctionnalités sécurisées à ce type d'application comme le stockage isolé, l'installation en dossier unique et la nécessité d'obtenir le consentement des utilisateurs afin d'accéder à des fonctionnalités « extérieures » (système de fichiers, réseau, caméra, microphone, etc.).

Bien que Microsoft indique clairement que WinRT n'est pas un juste un autre niveau d'abstraction et est conçu comme un remplacement du sous-système Win32, on peut néanmoins remarquer et dire de manière générale que les fondements de WinRT reposent en partie sur l'interface COM (*Component Object Model*), plus précisément une version étendue de COM, les binaires propres au framework WinRT et l'API Win32. De ce fait, WinRT est une couche applicative non managée native — programmée en C ++ — et son API est donc purement orientée objet. Cette API peut être utilisée aussi bien à partir de langages natifs qu'au travers de langages managés.

En conséquence Les développeurs peuvent écrire des applications ciblant WinRT pour Windows 8 en utilisant une variété de langages, y compris C ou C++ (avec WRL : *Windows Runtime Library* qui utilise la méta-programmation avec modèles) ou avec une version étendue de C++ appelée C++/CX (*C++ with Component Extensions*), des langages managés reposant sur le *Common Infrastructure Language* (par exemple C# et VB.NET) et enfin JavaScript.

0.3 Modèle de programmation

Ce chapitre vise à présenter la surface du Windows Runtime: comment les applications ciblant WinRT sont construites, ce qui les constitue et quelles sont les caractéristiques de sécurité apportées par le framework et le système d'exploitation.

0.3.1 Bibliothèques

Tout d'abord, nous parlerons ici de ce qui compose le cœur de WinRT — toujours au niveau utilisateur mais à bas niveau —, notamment au travers des différentes bibliothèques dynamiques qui constituent le framework dont voici un aperçu :

- Bibliothèques système usuelles
  - Système : ntdll.dll ; kernel32.dll ; kernelbase.dll ; user32.dll ; gdi32.dll ; advapi32.dll ; rpcrt4.dll ; etc.
  - runtime : msvcrt.dll ; msvcr110.dll ; msvcp110.dll ; etc.
- Bibliothèques cryptographiques
  - cryptbase.dll ; cryptsp.dll ; bcryptprimitives.dll ; etc.
- Bibliothèques COM
  - Notamment combase.dll
- Bibliothèques propres à WinRT
  - vccorelib110.dll [MS VC WinRT Core Library]
  - %systemroot%“system32“Windows.*.dll
  - twinapi.dll ; twinui.dll
- DirectX
  - DirectX 2D : d2dxx.dll
  - DirectX 3D : d3dxx.dll
- Langage managés seulement
  - CLR Host DLL : clrhost.dll [In Proc server for managed servers in the Windows Runtime]
  - CLR base components : clr.dll; clrjit.dll ; mscoree.dll ; mscorlib.dll ; etc.
  - Assemblies requis par le programme, par ex. : System.IO.dll ; System.Net.Primitives.dll ; System.Threading.dll ; etc.

On notera la présence évidente des bibliothèques dynamiques propres à l'API Win32. Les bibliothèques cryptographiques sont nécessaires afin de vérifier la signature de l'application et de ses fichiers. Les bibliothèques du *framework* WinRT sont rangées par espaces de noms (*namespace*), ces derniers étant utilisés lors de la programmation d'une application. Par exemple la bibliothèque Windows.Networking.dll correspond à l'espace de nom `Windows::Networking` : c'est-à-dire que la bibliothèque comprend toutes les classes, méthodes et fonctions propres à l'espace de nom correspondant [1]. Tous les programmes basés sur l'interface Metro utilisent DirectX pour leur rendu graphique, que le programme utilise explicitement DirectX ou non, ou encore qu'il soit managé ou pas. Les langages managés quant à eux utilisent bien sûr en sous-main les bibliothèques dynamiques propres au *framework* .NET. Finalement on mentionnera la présence de la bibliothèque *combase.dll* : les applications WinRT sont intimement dépendantes de la technologie COM (en réalité une version étendue de COM). Cette bibliothèque est la pierre angulaire de la machinerie COM sous-jacente.

0.3.2 Exemple de programme

Comme indiqué précédemment de nombreux langages de programmation, natifs ou managés, peuvent être utilisés pour programmer dans le cadre de WinRT. Nous montrerons ici un exemple en langage natif utilisant C++/CX et disposant d'une interface programmée en XAML. Notez que l'utilisation de XAML n'est pas obligatoire bien que simplifiant grandement la composition d'une interface graphique.

Pour rappel XAML (*Extensible Application Markup Language*) est langage basé sur XML [4]. Ce langage peut être utilisé comme langage de balisage pour interface utilisateur, permettant de définir des éléments de l'interface, des liaisons de données, des événements, etc.

Un aperçu du rendu de notre application de test est visble en figure 1.

1. Notez toutefois qu'il n'y a pas forcément une bijection entre un espace de nom et une bibliothèque dynamique

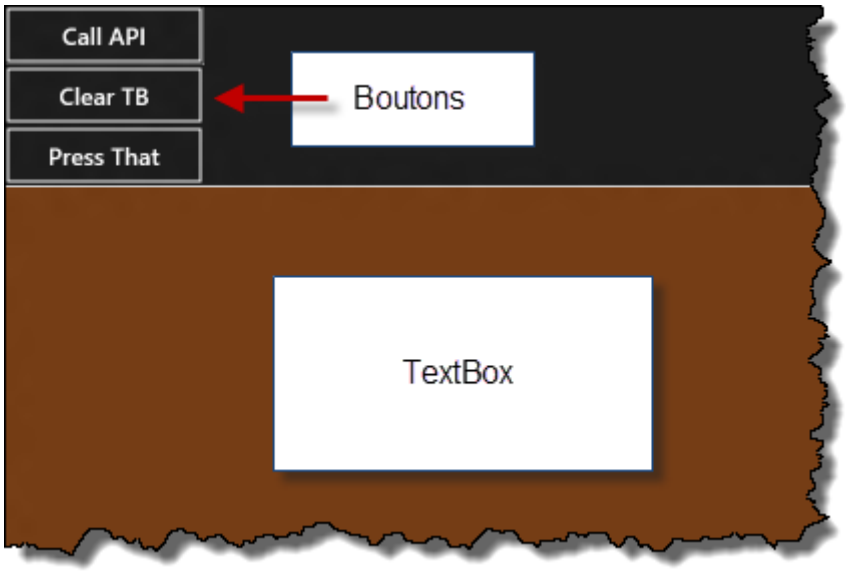

FIGURE 1 – Rendu d'une interface graphique en XAML

Le fichier XAML utilisé pour la constitution de cette même interface :

```
1  <Page
2      x:Class="TestApp.BlankPage"
3      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
4      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
5      xmlns:local="using:TestApp"
6      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
7      xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
8      mc:Ignorable="d">
9
10     <Grid Background="{StaticResource␣ApplicationPageBackgroundBrush}">
11         <StackPanel>
12             <Button Content="Call␣API" FontSize="24" Width="200" Height="60" Click="CallApi_Click" />
13             <Button Content="Clear␣TB" FontSize="24" Width="200" Height="60" Click="ClearButton_Click"
                    />
14             <Button Content="Press␣That" FontSize="24" Width="200" Height="60" Click="
                   PressThatButton_Click" />
15             <TextBox x:Name="textbl1" FontSize="24" Background="SaddleBrown" ScrollViewer.
                   VerticalScrollBarVisibility="Auto"
16                       ScrollViewer.HorizontalScrollBarVisibility="Auto" AcceptsReturn="True" Height="
                           649"/>
17         </StackPanel>
18     </Grid>
19 </Page>
```

Notre application est on ne peut plus simple : une grille contenant un *StackPanel* (conteneur où les éléments sont empilés les uns sur les autres) qui lui même contient 3 boutons et une *TextBox* (boîte de texte). Notez ci-dessous l'attribut `Click` de l'élément `Button` dont la valeur est `CallApi_Click` :

Listing 1– Attribut Click de l'élément Button

```
1  <Button Content="Call␣API" FontSize="24" Width="200" Height="60" Click="CallApi_Click" />
```

Ci-dessous la définition de la classe correspondante au XAML :

Listing 2– Attribut Click de l'élément Button

```
1  //
2  // BlankPage.xaml.h
3  // Declaration of the BlankPage.xaml class.
4  //
5
6  #pragma once
7
8  #include "pch.h"
9  #include "BlankPage.g.h"
```

```
10
11  namespace TestApp
12  {
13          /// <summary>
14          /// An empty page that can be used on its own or navigated to within a Frame.
15          /// </summary>
16          public ref class BlankPage sealed
17          {
18          public:
19                  BlankPage();
20
21          protected:
22                  virtual void OnNavigatedTo(Windows::UI::Xaml::Navigation::NavigationEventArgs^ e)
                        override;
23
24          private:
25                  void CallApi_Click(Platform::Object^ sender, Windows::UI::Xaml::RoutedEventArgs^ e);
26                  void ClearButton_Click(Platform::Object^ sender, Windows::UI::Xaml::RoutedEventArgs^ e
                        );
27                  void PressThatButton_Click(Platform::Object^ sender, Windows::UI::Xaml::
                        RoutedEventArgs^ e);
28                  void LauchBrowser(Windows::Foundation::Uri^ uri);
29          };
30  }
```

Finalement la mise en œuvre de la fonction `CallApi_Click()` :

Listing 3– Fonction CallApi_Click()

```
1  // fichier BlankPage.xaml.cpp
2
3  void BlankPage::CallApi_Click(Platform::Object^ sender, Windows::UI::Xaml::RoutedEventArgs^ e)
4  {
5    textbl1->Text = "[*] Hello World !\n";
6  }
```

Lors de la compilation d'un tel programme, Visual Studio génère un nombre important de fichiers `*.cpp` et `*.h` (notamment concernant l'interface graphique) en sous-main afin d'alléger le travail du développeur.

0.3.3 WinRT & C++/CX

Le système de type WinRT s'appuie fortement sur les composants WinRT, qui sont des objets COM mettant en œuvre un ensemble spécifique d'interfaces de programmation et adhérant à une certaine ABI (*Application Binary Interface*). Tous les objets WinRT (on parle aussi de composants) doivent implémenter l'interface `IInspectable` [5] et dériver, de manière ultime, de `Platform::Object` [6].

l'objet (classe WinRT) `FileInformation`, présent dans l'espace de nom `Windows::Storage::BulkAccess` [7] est présenté en figure 2.

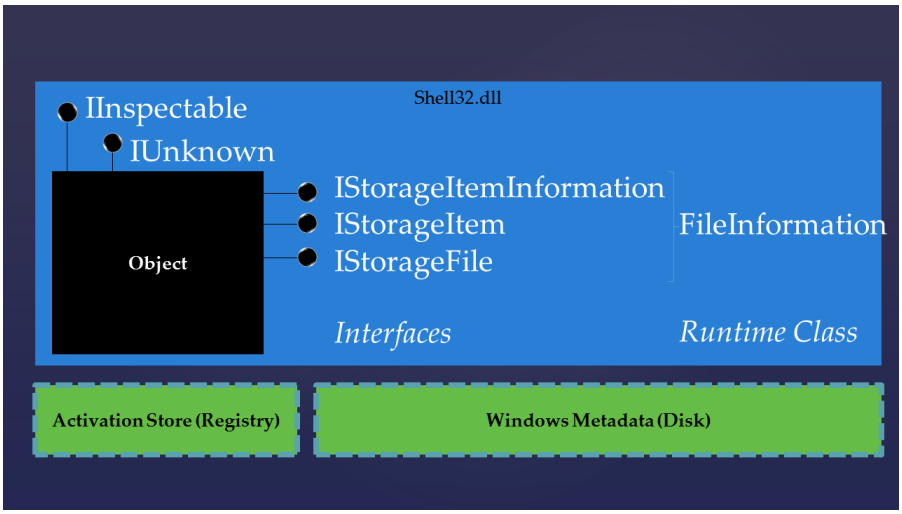

FIGURE 2 – Objet `FileInformation`

Ci-dessous l'interface `IInspectable` :

Listing 4– Interface `IInspectable`

```
1   // extrait de <inspectable.h>
2
3       MIDL_INTERFACE("AF86E2E0-B12D-4c6a-9C5A-D7AA65101E90")
4       IInspectable : public IUnknown
5       {
6       public:
7           virtual HRESULT STDMETHODCALLTYPE GetIids(
8               /* [out] */ __RPC__out ULONG *iidCount,
9               /* [size_is][size_is][out] */ __RPC__deref_out_ecount_full_opt(*iidCount) IID **iids) = 0;
10
11          virtual HRESULT STDMETHODCALLTYPE GetRuntimeClassName(
12              /* [out] */ __RPC__deref_out_opt HSTRING *className) = 0;
13
14          virtual HRESULT STDMETHODCALLTYPE GetTrustLevel(
15              /* [out] */ __RPC__out TrustLevel *trustLevel) = 0;
16
17      };
```

Un composant WinRT implémente donc l'interface IInspectable, qui dérive elle-même de `IUnknown` (notez toutefois que les composants WinRT, à la différence des composants COM, ne mettent pas en œuvre l'interface `IDispatch`). L'interface `IInspectable` propose trois méthodes :

- `GetIids()` : renvoie les interfaces implémentées par le composant ;
- `GetRuntimeClassName()` : renvoie le nom entièrement qualifié du composant ;
- `GetTrustLevel()` : renvoie le niveau de confiance du composant.

L'interface `IInspectable` est utilisée pour les liaisons dynamiques, notamment à partir de JavaScript. Dans le cadre d'une référence à un composant WinRT depuis JavaScript, l'interpréteur JavaScript peut alors appeler la méthode `IInspectable::GetRuntimeClassName()` afin d'obtenir le nom de classe entièrement qualifié (*fully qualified name*) du composant WinRT. En utilisant l'espace de noms, l'interpréteur peut alors demander au système de charger le fichier de métadonnées (fichier `*.winmd`) décrivant le composant, et à partir de ces métadonnées déterminer comment utiliser les interfaces proposées (mises en œuvre) par le composant.

Lorsque l'on accède à un composant WinRT depuis C++ ou un langage basé sur CLI (langages .NET), l'interface `IInspectable` n'est alors pas strictement nécessaire dans le sens où `IUnknown::QueryInterface` est suffisant. En fait, l'interopérabilité entre .NET et WinRT s'appuie sur une interopérabilité COM simple - un RCW (*Runtime Callable Wrapper*) est créé par le CLR permettant ainsi de gérer la durée de vie du composant WinRT sous-jacent. Dans le cas des liaisons dynamiques entre C++ et WinRT le processus est un peu plus complexe, et c'est ici que les extensions du langage (le « CX » de C++/CX) entrent en jeu.

C++/CX (*C++ with **C**omponent E**x**tension*) [8] n'est en soi pas un nouveau langage mais simplement C++ avec des extensions. C++/CX est en réalité un « sucre syntaxique » qui cache une mise en œuvre de niveau inférieur autour de COM. Bien qu'ayant une syntaxe ressemblant fortement à celle de C++/CLI (ce dernier étant un langage managé), C++/CX est purement un langage natif. Les extensions du langage C++ pour WinRT sont activées par le commutateur `/ZW` du compilateur Microsoft.

Les extensions permettent de faire la correspondance entre la syntaxe C++ WinRT et des patterns C++ standardisés comme les constructeurs, destructeurs, méthodes de classes, exceptions, etc. Elles permettent aussi de cacher la mise en œuvre sous-jacente de COM autour de l'activation d'une classe COM (au travers de la fonction `RoActivateInstance` [9]) lors de l'appel à un constructeur ou encore de cacher la gestion automatique des références.

La création d'un composant (implémentant donc `IInspectable`, `IUnknown` et dérivant de la classe `Platform::Object`) se fait en utilisant les mots clés `ref class` [10] comme dans l'exemple suivant :

Listing 5– Déclaration d'un composant WinRT

```
1   // WinRTComponent.h - édclaration du composant WinRTComponent
2   #pragma once
3   namespace CalcWinRTComponent
4   {
5       public ref class WinRTComponent sealed
6       {
7       public:
8           WinRTComponent();
9       int Mul(int x, int y);
10      int Add(int x, int y);
11      int Div(int x, int y);
```

```
12      };
13  }
```

Ci-dessous la mise en œuvre du composant :

Listing 6– Définition d'un composant WinRT

```
1  // WinRTComponent.cpp - éDfinition du composant [mise en oeuvre]
2  #include "pch.h"
3  #include "WinRTComponent.h"
4
5  using namespace CalcWinRTComponent;
6  using namespace Platform;
7
8  WinRTComponent::WinRTComponent(){}
9
10 int WinRTComponent::Mul(int x, int y){ return x*y;}
11 int WinRTComponent::Add(int x, int y){ return x+y;}
12 int WinRTComponent::Div(int x, int y){ return x/y;}
```

Le résultat est une classe C++ mettant en œuvre des interfaces COM de manière transparente.

C++/CX dispose d'un mot clé contextuel pour créer des objets WinRT : `ref new` [11]. Ce mot clé renvoie donc un *handle* d'objet WinRT désigné par une accent circonflexe (« ^ » appelé *hat*), qui est de manière conceptuelle semblable à un pointeur, la différence majeure étant que les instances ainsi créées sont des références comptées sur des objets COM : il n'est donc pas nécessaire au programmeur de s'occuper du cycle de vie d'un objet pris en charge par WinRT.

Voici un exemple de code :

Listing 7– Exemple de référence en C++/CX

```
1  void BlankPage::TestComponent(void)
2  {
3      // instanciation du composant
4    CalcWinRTComponent::WinRTComponent^ calcComp = ref new WinRTComponent();
5
6      // appel de émthode Add()
7    auto num = calcComp->Add(1337, 42);
8
9      // affichage du érsultat dans une boite de texte
10   this->textbl1->Text = num.ToString();
11
12     // élibration automatique de l'objet "calcComp",
13     //    plus aucune éérfrence né'tant éutilise.
14 }
```

Un *hat* est donc un pointeur sur un pointeur vers un tableau de pointeurs de fonctions. Il est possible d'inspecter précisément l'agencement d'une classe en mémoire en utilisant le commutateur de compilateur `/d1reportSingleClassLayout` :

```
class WinRTComponent size(48):
+---
| +--- (base class __IWinRTComponentPublicNonVirtuals)
| | +--- (base class Object)
 0 | | | {vfptr}
| | +---
| +---
| +--- (base class Object)
 8 | | {vfptr}
| +---
| +--- (base class IWeakReferenceSource)
| | +--- (base class __abi_IUnknown)
16 | | | {vfptr}
| | +---
| | +--- (base class Object)
24 | | | {vfptr}
| | +---
| +---
```

```
32 | __abi_FTMWeakRefData __abi_reference_count
+---
```

À la base de notre classe on trouve donc une classe `__IWinRTComponentPublicNonVirtuals` qui, comme son nom l'indique, comprend tous les membres publics et non virtuels de la classe :

```
WinRTComponent::$vftable@__IWinRTComponentPublicNonVirtuals@:
 | &WinRTComponent_meta
 |  0
 0 | &WinRTComponent::__abi_QueryInterface
 1 | &WinRTComponent::__abi_AddRef
 2 | &WinRTComponent::__abi_Release
 3 | &WinRTComponent::__abi_GetIids
 4 | &WinRTComponent::__abi_GetRuntimeClassName
 5 | &WinRTComponent::__abi_GetTrustLevel
 6 | &WinRTComponent::__abi_CalcWinRTComponent___IWinRTComponentPublicNonVirtuals____abi_Mul
 7 | &WinRTComponent::__abi_CalcWinRTComponent___IWinRTComponentPublicNonVirtuals____abi_Add
 8 | &WinRTComponent::__abi_CalcWinRTComponent___IWinRTComponentPublicNonVirtuals____abi_Div
 9 | &WinRTComponent::Mul
10 | &WinRTComponent::Add
11 | &WinRTComponent::Div
```

On retrouve bien l'interface `IUnknown` à la base de la classe (méthodes : `QueryInterface`, `AddRef` et `Release`) suivie des méthodes de l'interface `IInspectable`, déjà discutées, et enfin les méthodes propres à notre classe (`Mul`, `Add` et `Div` dans notre exemple) mais scindées en deux groupes.

Le premier groupe (voir ci-dessus : index 6, 7 et 8), dont les noms de méthodes commencent avec le suffixe `__abi`, comprend les méthodes utilisant l'ABI de WinRT. Ces méthodes prennent un pointeur `this*` en premier argument et un pointeur vers un entier en dernier argument, comme le montre le code ci-dessous :

Listing 8– Méthode Add - ABI

```
1  WinRTComponent::__abi_CalcWinRTComponent___
2      IWinRTComponentPublicNonVirtuals____abi_Add(
3          CalcWinRTComponent::WinRTComponent *this, int x, int y,
4              int *__abi_returnValue) {
5
6      *__abi_returnValue = x + y;
7
8      return 0;
9  }
```

Le dernier groupe de méthodes (index 9, 10, 11) comprend des méthodes simples n'utilisant pas une ABI particulière (méthodes proches du C si on excepte le pointeur `this*` passé en premier argument). Ci-dessous la méthode `Add` à l'index 10 de la `vtable`:

Listing 9– Méthode Add - Non ABI

```
1  WinRTComponent::Add(CalcWinRTComponent::WinRTComponent *this, int x, int y) {
2      return x + y;
3  }
```

Il est donc techniquement possible d'appeler une fonction directement via la table de pointeurs de fonctions (`vfptr`) comme le montre l'exemple ci-dessous :

Listing 10– Méthode Add - Non ABI

```
1  void BlankPage::TestComponent2(void)
2  {
3    auto calcComp = ref new WinRTComponent();
4
5    typedef int (__stdcall* PfnAdd)(void*, int, int);
6
7    auto pFunc = (*reinterpret_cast<PfnAdd**>(calcComp))[10];
8
9    auto num = pFunc(nullptr, 1337, 42);
10
11   textbl1->Text = num.ToString();
12 }
```

Une vue de l'instance d'objet (object `CalcWinRTComponent::WinRTComponent`) via l'inspecteur d'objets de Visual Studio est présentée en figure 3.

Reprenons cette fois-ci notre exemple du listing 7 et voyons comment cela ce traduit en langage assembleur `x86-64` :

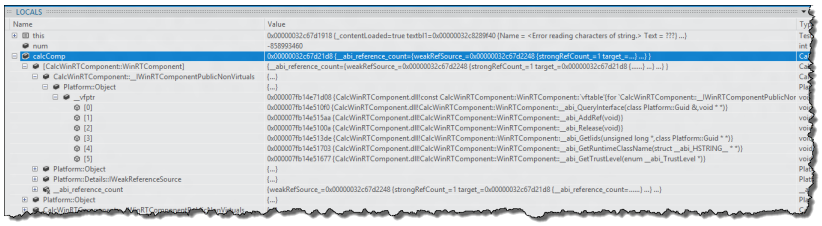

FIGURE 3 – Vue de `CalcWinRTComponent::WinRTComponent` par Visual Studio

Listing 11– Méthode TestComponent en assembleur

```
1  ; void __cdecl TestApp__BlankPage__TestComponent(TestApp::BlankPage *this)
2  ?TestComponent@BlankPage@TestApp@@AE$AAAXXZ proc near.
3
4  unk             = qword ptr -28h
5  __abi_return_value= qword ptr  10h
6  calcComp        = qword ptr  18h
7  str_int         = qword ptr  20h
8
9     push    rbx
10    push    rsi
11    push    rdi
12    sub     rsp, 30h
13    mov     [rsp+48h+unk], 0FFFFFFFFFFFFFFFEh
14    mov     rsi, rcx        ; rsi = rcx = this*
15    mov     [rsp+48h+__abi_return_value], 0
16    call    ??0WinRTComponent@CalcWinRTComponent@@QE$AAA@XZ
17    mov     rdi, rax
18    mov     [rsp+48h+calcComp], rax
19    mov     r10, [rax]      ; r10 = __IWinRTComponentPublicNonVirtuals vtable
20    lea     r9, [rsp+48h+__abi_return_value] ; return value
21    mov     edx, 0n1337     ; x
22    mov     r8d, 0n42       ; y
23    mov     rcx, rax        ; this*
24    call    qword ptr [r10+38h] ; index #7 in vtable (ABI Add) : 0x38 / sizeof(void*) == 7
25    test    eax, eax
26    jns     short @@NoException
27    mov     ecx, eax        ; hr
28    call    __abi_WinRTraiseException(long)
29    int     3               ; Trap to Debugger
30
31 @@NoException:
32    mov     eax, dword ptr [rsp+48h+__abi_return_value]
33    mov     dword ptr [rsp+48h+__abi_return_value], eax
34    lea     rcx, [rsp+48h+__abi_return_value]
35    call    cs:default::int32::ToString(void)
36    mov     rbx, rax        ; rbx = rax = String^
37    mov     [rsp+48h+str_int], rax
38    mov     rdx, rax        ; __param0
39    mov     rcx, [rsi+1B0h] ; this
40    call    ?set@?QITextBox@Controls@Xaml@UI@Windows@@Text@TextBox
41    nop
42    mov     rcx, rbx
43    call    WindowsDeleteString_0 ; delete String^
44    nop
45    mov     rax, [rdi]
46    mov     rcx, rdi
47    add     rsp, 30h
48    pop     rdi
49    pop     rsi
50    pop     rbx
51    jmp     qword ptr [rax+10h]
52 ?TestComponent@BlankPage@TestApp@@AE$AAAXXZ endp
```

Notez les différents appels présents dans le code ci-dessus :

1. Appel du construteur du composant (ligne 16);
2. Appel de la méthode `Add` (ligne 24) [paramètres passés par registres lignes 20-23];
3. Appel de la méthode `int32::ToString` (ligne 35);
4. Appel de la propriété `Text` de `TextBox` (ligne 40);
5. Appel du destructeur de la chaine créée via `int32::ToString` (ligne 43);
6. Appel du destructeur du composant, via `IUnknown::Release` (ligne 51).

Activation d'un composant

La création d'un objet WinRT en utilisant le mot clé `ref new` ne fait pas que créer un objet sur le tas. L'objet ainsi créé est un objet COM et sous WinRT ces objets sont créés par l'intermédiaire de ce que l'on nomme une « usine d'activation » (*activation factory*) qui met en œuvre l'interface `IActivationFactory` [12] disposant de la méthode `ActivateInstance` :

Listing 12– Interface `IInspectable`

```
1   // extrait de <Activation.h>
2
3       MIDL_INTERFACE("00000035-0000-0000-C000-000000000046")
4       IActivationFactory : public IInspectable
5       {
6       public:
7           virtual HRESULT STDMETHODCALLTYPE ActivateInstance(
8               /* [out] */ __RPC__deref_out_opt IInspectable **instance) = 0;
9
10      };
```

L'activation d'une classe permet d'enregistrer celle-ci dans le magasin d'activation de WinRT (*WinRT activation store*), permettant ainsi de mettre à disposition d'une application WinRT une instance de la classe.

WinRT comprend notamment des fonctions, nommément `RoActivateInstance` et `RoGetActivationFactory`, qui permettent d'obtenir l'*activation factory* pour une classe WinRT spécifique. Dans le cadre de la mise en œuvre d'une classe WinRT, il est nécessaire de pourvoir la *factory* adéquate (qui elle-même met en œuvre `IActivationFactory`). Cela dit, ceci n'est pas vraiment nécessaire dans la grande majorité des cas puisque le compilateur va générer toute la machinerie nécessaire lorsque C++/CX est utilisé.

Donc, une fois que l'objet `IActivationFactory` est obtenu, la méthode `ActivateInstance` est appelée sur ce même objet. Si l'appel est réussi, un pointeur vers un objet `IInspectable` est renvoyé (par l'intermédiaire du paramètre de sortie). `IInspectable` est à WinRT ce que `IUnknown` était à une approche COM plus « traditionnelle ».

En pratique l'activation d'un composant WinRT se fait au travers du constructeur de celui-ci (voir ligne 16 du listing assembleur ci-dessus qui est le constructeur du composant). L'activation est présentée de manière générale ci-dessous :

1. L'application demande la création de l'objet (via le constructeur de l'objet) ;
2. Le nom de l'objet est passé à `RoActivateInstance` ;
3. La bibliothèque dynamique correspondante est recherchée grâce au catalogue de composants (situé dans la base de registre) ;
4. La bibliothèque dynamique est chargée ;
5. Appel de la fonction `DllGetActivationFactory` mise en œuvre par la bibliothèque dynamique du composant ;
6. La fonction précédente appelle `GetActivationFactory` ;
7. L'objet est créé ;
8. L'interface `IInspectable` est renvoyée ;
9. (langage nécessitant une projection) La projection crée un `wrapper` en utilisant les méta-données du fichier winmd ;
10. (langage nécessitant une projection) L'objet est lié à la projection ;
11. (langage nécessitant une projection) Le `wrapper` est retourné à l'application.

Ci-dessous la commande `wt` (*Watch and Trace*) effectuée depuis un *debugger* sur le constructeur de notre composant exemple (`CalcWinRTComponent::WinRTComponent`). Cette commande permet de voir tous les appels de fonctions exécutés dans une fonction spécifique. Notez que la profondeur n'est ici que de deux niveaux (option `-l2`) : sans spécification de niveau la *trace* comprend en effet plus de 3800 lignes:

```
0:004> wt -l2 -or -oR
   17     0 [  0] TestApp!CalcWinRTComponent::WinRTComponent::WinRTComponent
   14     0 [  1]   vccorlib110!GetActivationFactoryByPCWSTR
   66     0 [  2]     vccorlib110!__getActivationFactoryByHSTRING rax = 0
   20    66 [  1]   vccorlib110!GetActivationFactoryByPCWSTR rax = 0
   25    86 [  0] TestApp!CalcWinRTComponent::WinRTComponent::WinRTComponent
    8     0 [  1]   CalcWinRTComponent!CalcWinRTComponent::__WinRTComponentActivationFactory::
                      __abi_AddRef = unsigned long 4
```

```
31    94 [  0] TestApp!CalcWinRTComponent::WinRTComponent::WinRTComponent
 7     0 [  1]   CalcWinRTComponent!CalcWinRTComponent::__WinRTComponentActivationFactory::
                     [Platform::Details::IActivationFactory]::
                         __abi_Platform_Details_IActivationFactory____abi_ActivateInstance
10     0 [  2]     CalcWinRTComponent!CalcWinRTComponent::__WinRTComponentActivationFactory::
                     [Platform::Details::IActivationFactory]::ActivateInstance
                         = class Platform::Object * 0x000000b6'39474328
13    10 [  1]   CalcWinRTComponent!CalcWinRTComponent::__WinRTComponentActivationFactory::
                     [Platform::Details::IActivationFactory]::
                         __abi_Platform_Details_IActivationFactory____abi_ActivateInstance
                             = long 0n0
39   117 [  0] TestApp!CalcWinRTComponent::WinRTComponent::WinRTComponent
 8     0 [  1]   CalcWinRTComponent!CalcWinRTComponent::WinRTComponent::__abi_AddRef
                     = unsigned long 2
44   125 [  0] TestApp!CalcWinRTComponent::WinRTComponent::WinRTComponent
 6     0 [  1]   CalcWinRTComponent!CalcWinRTComponent::WinRTComponent::__abi_Release
17     0 [  2]     CalcWinRTComponent!__abi_FTMWeakRefData::Decrement = long 0n1
13    17 [  1]   CalcWinRTComponent!CalcWinRTComponent::WinRTComponent::__abi_Release
                     = unsigned long 1
53   155 [  0] TestApp!CalcWinRTComponent::WinRTComponent::WinRTComponent
27     0 [  1]   CalcWinRTComponent!CalcWinRTComponent::WinRTComponent::__abi_QueryInterface
                     = long 0n0
62   182 [  0] TestApp!CalcWinRTComponent::WinRTComponent::WinRTComponent
 8     0 [  1]   CalcWinRTComponent!CalcWinRTComponent::WinRTComponent::__abi_AddRef
                     = unsigned long 3
68   190 [  0] TestApp!CalcWinRTComponent::WinRTComponent::WinRTComponent
 6     0 [  1]   CalcWinRTComponent!CalcWinRTComponent::WinRTComponent::__abi_Release
17     0 [  2]     CalcWinRTComponent!__abi_FTMWeakRefData::Decrement = long 0n2
13    17 [  1]   CalcWinRTComponent!CalcWinRTComponent::WinRTComponent::__abi_Release
                     = unsigned long 2
74   220 [  0] TestApp!CalcWinRTComponent::WinRTComponent::WinRTComponent
 6     0 [  1]   CalcWinRTComponent!CalcWinRTComponent::WinRTComponent::__abi_Release
17     0 [  2]     CalcWinRTComponent!__abi_FTMWeakRefData::Decrement = long 0n1
13    17 [  1]   CalcWinRTComponent!CalcWinRTComponent::WinRTComponent::__abi_Release
                     = unsigned long 1
78   250 [  0] TestApp!CalcWinRTComponent::WinRTComponent::WinRTComponent
 6     0 [  1]   CalcWinRTComponent!CalcWinRTComponent::__WinRTComponentActivationFactory::
                     __abi_Release
17     0 [  2]     CalcWinRTComponent!__abi_FTMWeakRefData::Decrement = long 0n3
14    17 [  1]   CalcWinRTComponent!CalcWinRTComponent::__WinRTComponentActivationFactory::
                     __abi_Release = unsigned long 3
```

Notez finalement qu'il est bien sûr possible de cibler le *framework* WinRT en C, au prix style de programmation fastidieux et très « bavard » du fait des très nombreux appels relatifs aux méthodes COM, appels qui sont de fait cachés par l'utilisation de C++/CX. La bibliothèque WRL (*Windows Runtime Library*) peut être utilisée pour cibler WinRT dans une application sans toutefois utiliser C++/CX mais sans non plus mettre en œuvre toute la « plomberie » nécessaire à bas niveau.

Conteneur d'application & Binaire WinRT

Windows 8 introduit le concept d'une nouvelle sandbox de sécurité, appelé *AppContainer* (conteneur d'application), qui offre des autorisations de sécurité plus fines que sur les systèmes précédents. Ce conteneur permet notamment de bloquer les accès en écriture et/ou lecture au système de fichier ainsi que l'utilisation d'autres ressources du système (microphone, caméra, Internet, etc.). C'est l'*AppContainer* qui contribue à s'assurer que l'application n'a pas accès à des capacités qu'elle n'aurait pas préalablement déclarées et/ou qui n'auraient pas été accordées à l'utilisateur. Ce sujet est abordé plus en détail au chapitre « SandBoxes » 0.4.

Les fichiers exécutables ciblant WinRT restent des fichiers au format PE (*Portable Executable*), on notera que ce format n'a pas été altéré pour prendre en charge WinRT. Il est néanmoins possible de savoir si une application est un `AppContainer` (application utilisant la sandbox de WinRT) en testant le champ `DllCharacteristics` de la structure `_IMAGE_OPTIONAL_HEADER` situé dans l'en-tête PE avec le masque suivant :

Listing 13– DllCharacteristics - AppContainer

```
1  // extrait de winnt.h - BUILD Version: 0084
2  #define IMAGE_DLLCHARACTERISTICS_APPCONTAINER 0x1000     // Image should execute in an AppContainer
```

La valeur du champ `DllCharacteristics` par défaut pour une application WinRT est de 0x9160 ce qui correspond aux champs suivants :

```
// Image can handle a high entropy 64-bit virtual address space.
IMAGE_DLLCHARACTERISTICS_HIGH_ENTROPY_VA 0x0020

// DLL can move.
IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE 0x0040

// Image is NX compatible
IMAGE_DLLCHARACTERISTICS_NX_COMPAT 0x0100
```

```
// Image should execute in an AppContainer
IMAGE_DLLCHARACTERISTICS_APPCONTAINER 0x1000

//  Application is Remote Desktop Services aware
IMAGE_DLLCHARACTERISTICS_TERMINAL_SERVER_AWARE 0x8000
```

Notez que les fonctionnalités de sécurité introduites par le compilateur et le système telles que `SAFESEH` (gestionnaire d'exceptions sécurisé), `DYNAMICBASE` (adresse de base du module aléatoire) et `NXCOMPAT` (données non exécutables) sont des caractéristiques obligatoires pour les applications ciblant WinRT qui sont amenées à être partagées publiquement. On notera l'apparition des options de compilation suivantes propres à Visual Studio 11 et WinRT:

- /ZW : le programme cible WinRT ;
- /APPCONTAINER : l'application s'exécute dans un *AppContainer* ;
- /WINMDFILE : produit un fichier de méta-données ;
- /WINMDDELAYSIGN : permet de signer partiellement le fichier de méta-données ;
- /WINMDKEYCONTAINER : permet de spécifier un conteneur de clé pour signer le fichier de méta-données ;
- /WINMDKEYFILE : permet de spécifier une clé ou une paire de clés pour signer le fichier de méta-données.

Notons finalement qu'il est impossible de démarrer une application WinRT en double-cliquant sur celle-ci [2], des vérifications internes mises en place au niveau du kernel empêchent ce type de démarrage. Il est donc nécessaire de passer par le bureau Metro pour démarrer une application ciblant WinRT.

Projection

Les « projections » WinRT sont des liaisons entre le cœur bas niveau de WinRT et les langages situés à un niveau supérieur. De manière concrète une « projection » WinRT est assimilable au terme de *binding*.

Les projections WinRT dénotent la façon dont les API Windows Runtime sont exposées pour chaque langage prenant en charge WinRT. Cela peut se faire au moment même de la compilation (comme en C++) ou à l'exécution (comme en JavaScript) ou une combinaison des deux comme en C#. Chaque langage décide ainsi, au mieux, de la façon dont il présente l'API WinRT à ses consommateurs. La plupart du temps il s'agit d'une exposition directe, mais d'autres fois ce sont des wrappers ou redirections qui peuvent entrer en action.

Lors de la mise en œuvre d'un composant en C++ ou dans un langage. NET, son API sera décrite dans un fichier `*.winmd` [13]. On peut alors utiliser ce composant à partir des trois environnements possibles (natif, JavaScript et. NET). Même en C++ l'utilisateur n'est alors jamais exposé à la sous-couche COM, l'utilisation de COM se cachant derrière les outils de projection. Le programmeur utilise alors ce qui ressemble à une API C++ orientée objet.

Pour pouvoir utiliser les différentes constructions de WinRT, la plate-forme sous-jacente définit un ensemble de types de bases et leurs correspondances (*mapping*) vers différents langages. Par exemple, aux *collections* de WinRT correspondent des constructions qui sont propres à chaque langage.

Dans le cas d'un composant natif, la sortie – des outils de compilation et d'édition de lien – inclut une `*.dll` et un fichier `*.winmd` (et un fichier `*.pdb` si nécessaire). Dans le cas du composant géré (*managé*), la sortie est soit une `*.dll` soit un fichier `*.winmd` (plus le fichier `*.pdb` associé), en fonction du type de sortie tel que défini dans les propriétés du projet. Si le type de projet demandé est une bibliothèque de classe, la sortie est une `*.dll`, ce qui est suffisant si le composant est censé être utilisé à partir d'un langage managé. Toutefois, si le composant doit être utilisé à partir de C++/CX ou Javascript, le projet doit alors être configuré pour utiliser des fichiers `*.winmd`. Dans ce cas le fichier `*.dll` est remplacé par un fichier `*.winmd`, qui contient, au moins dans sa version actuelle, à la fois les méta-données et l'application.

Ces projections, donc mises en œuvre par le biais des fichiers `*.winmd`, sont au format ECMA-335 [14] et entièrement dé-compilables.

La figure4 montre notre composant de démonstration tel qu'il est vu par une application C#.

La même vue avec le composant dé-compilé :

2. Cela reste possible à l'aide d'un *debugger kernel* en circonvenant la vérification: dans ce cas précis le processus est lancé en *medium integrity*, en dehors de la sandbox

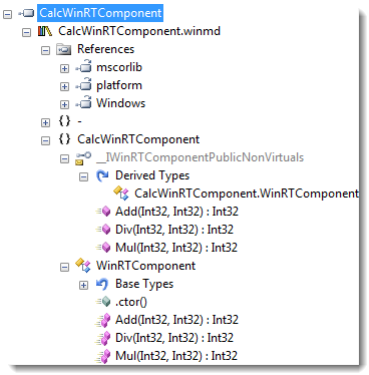

FIGURE 4 – Vision par une application C# du composant `CalcWinRTComponent`

Listing 14– Composant dé-compilé

```
1  namespace CalcWinRTComponent
2  {
3      internal interface __IWinRTComponentPublicNonVirtuals
4      {
5          // Methods
6          int Add(int x, int y);
7          int Div(int x, int y);
8          int Mul(int x, int y);
9      }
10
11     public sealed class WinRTComponent : __IWinRTComponentPublicNonVirtuals
12     {
13         // Methods
14         [MethodImpl(0, MethodCodeType=MethodCodeType.Runtime)]
15         public WinRTComponent();
16         [MethodImpl(0, MethodCodeType=MethodCodeType.Runtime)]
17         public sealed override int Add(int x, int y);
18         [MethodImpl(0, MethodCodeType=MethodCodeType.Runtime)]
19         public sealed override int Div(int x, int y);
20         [MethodImpl(0, MethodCodeType=MethodCodeType.Runtime)]
21         public sealed override int Mul(int x, int y);
22     }
23 }
```

Le même fichier dé-compilé via le programme `ILDASM` est visble en figure 5.

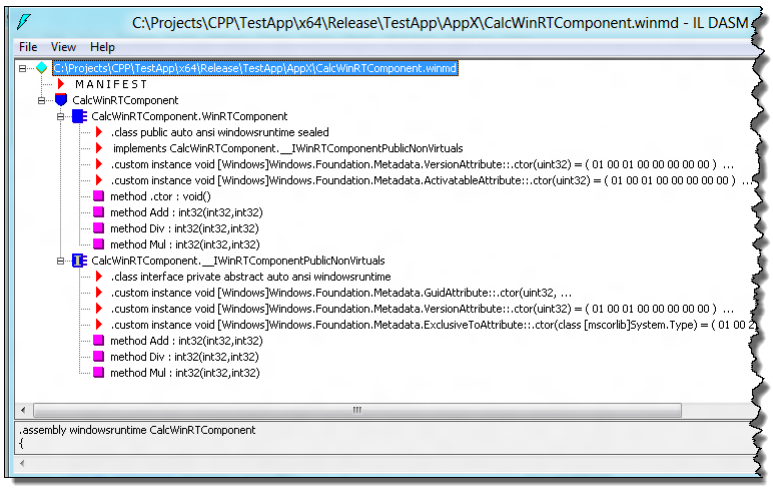

FIGURE 5 – Composant `CalcWinRTComponent` décompilé avec `ILDASM`

0.3.4 *Package* d'application

Les applications WinRT sont chacune construites autour de ce qu'on appelle un « *Package* » [26], « *App Package* » ou encore « *AppPackage* » (aussi connu dans nos contrées sous le dénominatif de progiciel) qui est l'unité de déploiement d'une application Metro (utilisant donc WinRT). Une fois le code de l'application mis en œuvre, la création et la distribution d'un *AppPackage* est entièrement automatisée sous Visual Studio 11 (menu "Project" –¿ "Store" –¿ "Create App Package").

De manière schématique, les étapes de la création d'un *AppPackage* sont les suivantes :

- Inclusions des fichiers binaires exécutables ;
  - Exécutable principal (*.exe) ;
  - Composants externes (*.dll) ;
  - Fichiers méta-données nécessaires aux projections (*.winmd).
- Fichier interface graphique, si requis (*.xaml) ;
- Fichier Manifest (*.xml) ;
- Resources
  - Ressources textuelles (*.xml) ;
  - Resources compilées [ex : support multilingue] (*.pri) ;
  - Fichiers images (dossier ."Assets).
- Signature d'application (*.p7x).

Une fois ces ressources rassemblées, le tout est regroupé dans un fichier `*.appx` qui est un fichier répondant au format `zip`. Un autre *package* (`*.appxsym`, toujours au format `zip`) est créé, renfermant les fichiers `*.pdb` utiles en cas de crash afin de générer un rapport (seulement si l'inclusion de ces fichiers a été demandée lors de la création du *package*).

Les *packages* susmentionnés sont finalement regroupés dans un *package* `*.appxupload` (format `zip`). Ce dernier est ensuite signé à l'aide d'un certificat (`*.cer`). Notez qu'il est possible de combiner à l'intérieur d'un même *AppPackage* jusqu'à trois versions du même programme ciblant des plate-formes différentes : x86, x86-64 et ARM. Le package ainsi généré n'est dépendant que du framework .NET dans le cas d'un programme managé ou des runtimes C++ dans le cas d'un programme natif, les deux étant déjà pré-installés sur Windows 8.

L'*upload* de l'application vers le Windows Store se fait via Visual Studio 11. L'installation locale est possible via Visual Studio 11 ou par l'intermédiaire un script *shell*, mais uniquement si la machine dispose d'une licence développeur. En pratique il n'existe pas d'installateur (type `*.msi` ou `*.exe`) d'*AppPackage*, mais en sous-main l'installateur courant de Windows (`msiexec.exe`) est utilisé pour installer le *package*, enregistrer le certificat et créer toutes les clés de registres requises.

L'installation via le Windows Store ne requiert aucun droit particulier, les applications étant enregistrées et créées pour chacun des utilisateurs (les clés de registres associées à l'application sont en effet créées dans la partie privée et relative à chaque utilisateur : `HKEY_CURRENT_USER` ou `HKCU`).

L'installation est en partie opérée grâce au fichier `AppxManifest.xml` qui vient avec l'application, notamment grâce au contenu de la balise `<Application>` :

Listing 15– Fichier Manifest

```
<?xml version="1.0" encoding="utf-8"?>
<Package xmlns="http://schemas.microsoft.com/appx/2010/manifest">
  <Identity Name="TesAppPackage-da8b54b0-f34d-4ef4-821d" Publisher="CN=Seb" Version="1.0.0.0"
        ProcessorArchitecture="x64" />
  <Properties>
    <DisplayName>TestApp</DisplayName>
    <PublisherDisplayName>Seb</PublisherDisplayName>
    <Logo>Assets\StoreLogo.png</Logo>
    <Description>TestApp</Description>
  </Properties>
  <!-- ** SNIP ** -->
  <Applications>
    <Application Id="App" Executable="TestApp.exe" EntryPoint="TestApp.App">
      <!-- ** SNIP ** -->
      <Extensions>
        <Extension Category="windows.fileSavePicker">
          <FileSavePicker>
            <SupportedFileTypes>
              <FileType>.txt</FileType>
            </SupportedFileTypes>
          </FileSavePicker>
        </Extension>
      </Extensions>
```

```
24        </Application>
25      </Applications>
```

Les propriétés déclarées dans le fichier XML sont alors inscrites dans la base de registre durant l'installation :

La clé de registre `HKCU\Software\Classes\ActivatableClasses\Package` contient la liste de tous les *AppPackage* enregistrés pour l'utilisateur courant. En figure 6 un exemple avec notre *package* de test.

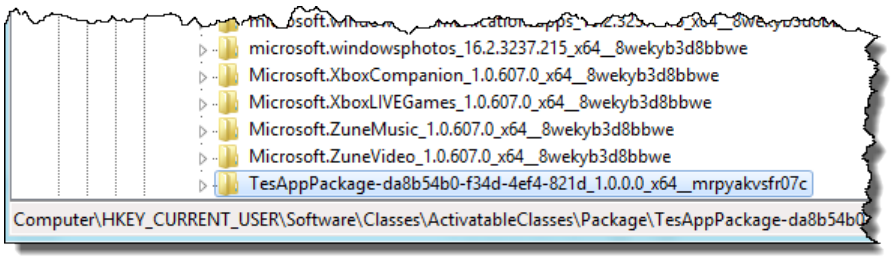

FIGURE 6 – *AppPackage* dans la base des registres

Une vue des sous-clés (et de leur valeurs) provenant de notre *package* de test est présentée en figure 7.

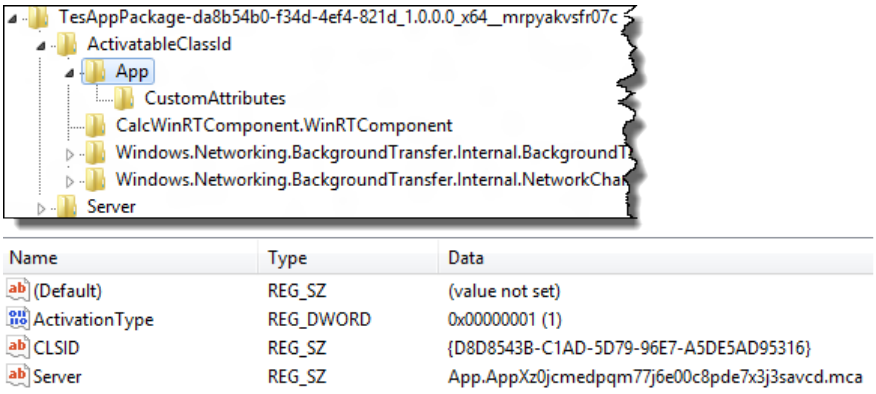

FIGURE 7 – Valeurs de *AppPackage* dans la base des registres

Les applications Metro utilisent des contrats et des extensions [15] pour déclarer les interactions, avec d'autres applications et Windows, qu'elles peuvent prendre en charge. Ces applications doivent inclure les déclarations nécessaires dans le manifeste du *package* et mettre en œuvre certains appels vers les API du Windows Runtime pour communiquer avec Windows et les applications participants à d'autres contrats.

Un schéma de présentation des extensions ets visble en figure 8.

Ci-dessous une partie d'un fichier de *Manifest* d'application où est déclarée une extension :

Listing 16– Fichier Manifest - déclaration d'extension

```
1       <!-- ** SNIP ** -->
2       <Extensions>
3         <Extension Category="windows.fileSavePicker">
4           <FileSavePicker>
5             <SupportedFileTypes>
6               <FileType>.txt</FileType>
7             </SupportedFileTypes>
8           </FileSavePicker>
9         </Extension>
10      </Extensions>
```

Un contrat peut être vu comme un accord entre Windows et une ou plusieurs applications. Les contrats définissent les exigences auxquelles les applications doivent répondre pour participer à ces interactions.

Dans l'exemple ci-dessus l'application implémente le contrat `FileSavePicker` qui permet à une application de sauvegarder des fichiers. L'intérêt ici est que d'autres applications peuvent utiliser cette application (proposant donc le contrat `FileSavePicker`) pour sauvegarder leurs propres fichiers. Par exemple, si cette application

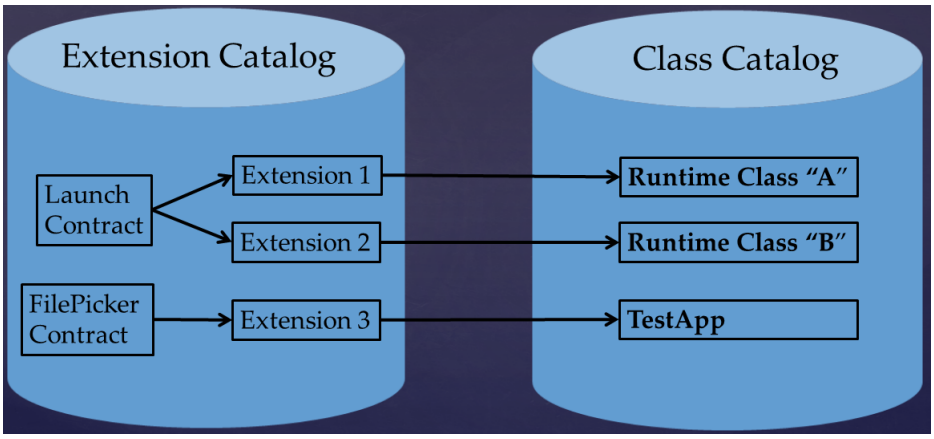

FIGURE 8 – Extensions

propose de sauvegarder des fichiers sur Internet (pour utiliser un mot à la mode, utilisons le vocable *cloud*), il est alors possible pour d'autres applications, lorsqu'elles vont vouloir sauvegarder leurs propres fichiers, d'utiliser cette application pour sauvegarder leurs fichiers sur internet, et ce de manière transparente : lorsqu'une application voudra sauvegarder un fichier, une boite de dialogue permettant de sauvegarder le fichier apparaît et il suffit de choisir l'application permettant de sauvegarder sur internet pour que celle-ci soit appelée et fasse ce qui lui est demandé.

Il existe d'autres types de contrats, le plus important d'entre eux étant le contrat `Launch` qui permet de démarrer une application. Ce contrat est utilisé lorsqu'une application est démarrée depuis `explorer` mais aussi depuis une autre application Metro.

Les contrats disponibles sont visibles depuis la clé de registre suivante :

– `HKEY_CURRENT_USER\Software\Classes\Extensions`

La figure 9 montre une vue des contrats dans la base registre.

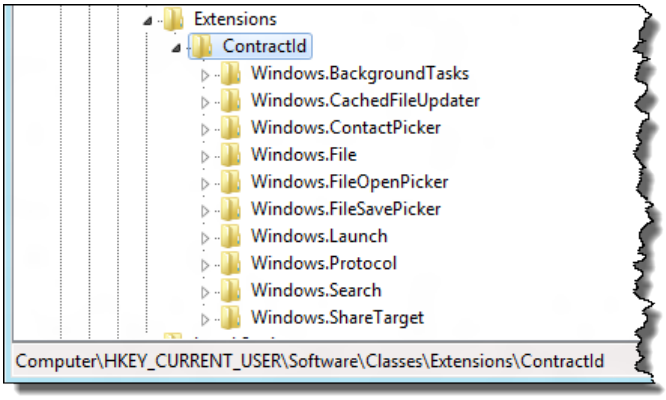

FIGURE 9 – Listes des contracts dans la base de registre

Notez que les contrats sont donc disponibles par utilisateur grâce à l'utilisation de la clé `HKCU`. Sous chaque contrat on retrouve la liste des applications mettant en œuvre ce contrat, comme montré en exemple dans la figure 10 où notre application de test met en œuvre le contrat `FileSavePicker`.

0.3.5 Capacités

Les *packages* d'applications WinRT utilisent des « capacités » [16] (*capabilities*) qui définissent les privilèges accordés aux applications, c'est-à-dire ce à quoi l'application peut accéder, par exemple des ressources comme : microphone, webcam, Internet, système de fichiers du système d'exploitation, etc.

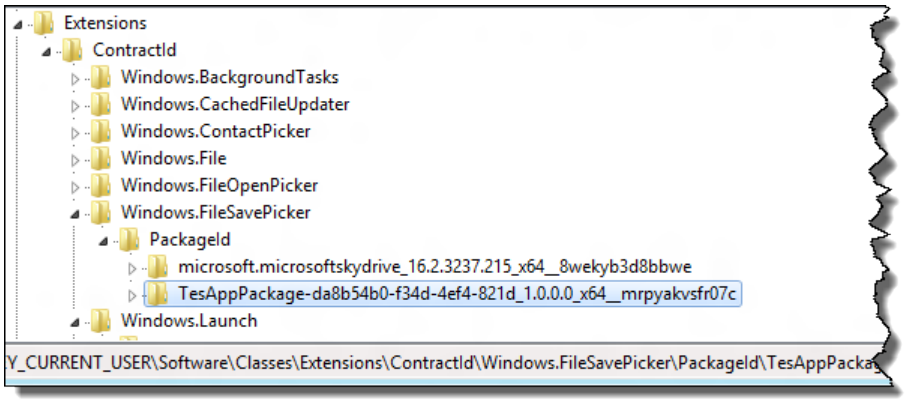

FIGURE 10 – Contrat `FileSavePicker`

Les capacités sont d'abord définies dans le fichier `manifest` utilisé lors de l'installation de l'application. Ci-dessous un exemple provenant de notre application de test :

Listing 17– Fichier Manifest - Capacités

```
1    <Capabilities>
2      <Capability Name="picturesLibrary" />
3      <Capability Name="internetClient" />
4    </Capabilities>
```

Dans cet exemple l'application peut accéder à Internet et au système de fichiers, mais dans ce dernier cas uniquement au dossier `pictures` qui, rappelons-le, est un dossier privé réservé à l'utilisateur courant.

Les `capabilities` sont inscrites dans la base de registre via des descripteurs de sécurité (`SID`) dans la clé suivante :

– `HKEY_CURRENT_USER\Software\Classes\LocalSettings\Software\Microsoft\Windows\CurrentVersion\AppModel\Repository\Packages\<PackageName>`

On peut voir, dans la figure 11 les clés `CapabilitySids` et `PackageSid` concernant respectivement les descripteurs de sécurité inhérents aux capacités et au `package` d'application.

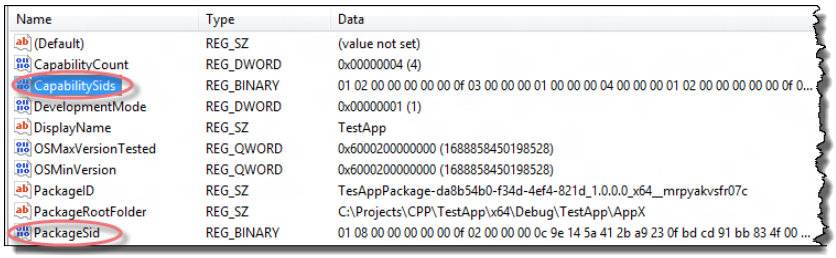

FIGURE 11 – Capacités dans la base des registres

Chacune des capacités d'une application, définies dans le fichier `manifest.xml` utilisé lors de l'installation, est inscrite dans la base de registre à un endroit propre à l'application (et propre à l'utilisateur). Chacune de ces capacités est alors exprimée via un descripteur de sécurité (`SID`) connu [17].

0.3.6 Démarrage d'une application WinRT

Le démarrage d'une application se fait via sa *tuile* (*tile*) depuis le bureau Metro (*Metro Desktop*), ce dernier étant en fait une extension de l'explorateur Windows (explorer.exe). C'est donc depuis explorer.exe que le programme est lancé, via le *shell* notamment depuis la fonction `CShellTask::TT_Run`. Ci-dessous une *stack trace* du début du lancement d'un programme WinRT :

...

```
twinui.dll CRunnableTask::Run
SHELL32.dll CShellTask::TT_Run
SHELL32.dll CShellTaskThread::ThreadProc
SHELL32.dll CShellTaskThread::s_ThreadProc
SHCORE.dll ExecuteWorkItemThreadProc
...
```

De là, `explorer` va vérifier que l'application met bien en œuvre le contrat `Launch`. Si tel est le cas, toutes les clés de registres sont lues et l'application est activée. Notez dans la *stack trace* ci-dessous l'utilisation du *class catalog* et de l'*extension catalog* :

```
...
combase.dll CComCatalog::GetRuntimeClassInfo
combase.dll WinRTGetRuntimeClassInfo
combase.dll CActivationStore::GetActivatableClassRegistration
combase.dll GetExtensionRegistrationByIndex
combase.dll CExtensionRegistrationsIterator::AdvanceToNextValidRegistration
combase.dll CExtensionRegistrationsIterator::FinalConstruct
combase.dll CExtensionCatalog::QueryCatalogByPackageFamilyName
twinui.dll CApplicationActivationManager::ActivateApplicationForContractByAppId
twinui.dll AddAltTextFromImageElements
twinui.dll ActivateApplicationForLaunch
twinui.dll shell::TaskScheduler::CSimpleRunnableTaskParam
twinui.dll shell::TaskScheduler::CSimpleRunnableTaskParam
twinui.dll CRunnableTask::Run
SHELL32.dll CShellTask::TT_Run
...
```

Ci-dessous l'activation utilisant la classe `ActivationStore` :

```
...
combase.dll ActivationStore::ActivatableClassIdEntry::Initialize
combase.dll CWinRTActivationStoreCatalog::GetRuntimeClassInfo
combase.dll CComCatalog::GetRuntimeClassInfoFromPackageScopedCatalog
combase.dll CComCatalog::GetRuntimeClassInfo
combase.dll WinRTGetRuntimeClassInfo
combase.dll CExtensionRegistration::Activate
twinui.dll CApplicationActivationManager::_ActivateExtensionForContractHelper
twinui.dll CApplicationActivationManager::ActivateApplicationForContractByAppId
twinui.dll AddAltTextFromImageElements
twinui.dll ActivateApplicationForLaunch
twinui.dll shell::TaskScheduler::CSimpleRunnableTaskParam
twinui.dll shell::TaskScheduler::CSimpleRunnableTaskParam
twinui.dll CRunnableTask::Run
SHELL32.dll CShellTask::TT_Run
...
```

Ensuite les ressources sont lues et le *splash screen* est affiché.

Une requête RPC (`Remote Procedure Call`) — conduite en sous-main sur le mécanisme de transport ALPC[3] — est alors envoyée au processus svchost.exe qui détient le service RPCSS (`Remote Procedure Call Service`). Ce service est notamment chargé de gérer l'activation de certains objets COM et d'assurer la communication entre les objets COM distants (communication inter-processus) sur l'ordinateur local ou via le réseau. Du fait de la surface d'attaque possible sur ce service, celui-ci fonctionne avec des privilèges restreints (`NT Authority\Network Service`).

Le service RPCSS envoie alors une requête RPC au service `DCOM Server Process Launcher`, qui fonctionne cette fois avec le privilège le plus haut possible, c'est-à-dire le privilège `NT Authority\SYSTEM`. Le but de ce service est de démarrer des applications, notamment via l'*impersonation*, c'est-à-dire en utilisant les privilèges d'un utilisateur du système ou des privilèges spéciaux. Ci-dessous la *stack trace* menant au démarrage du processus WinRT depuis le service `DCOM Server Process Launcher` :

```
>kb
KERNELBASE!CreateProcessAsUserW
rpcss!CClsidData::PrivilegedLaunchRunAsServer+0x2886
rpcss!_LaunchRunAsServer+0xb3
RPCRT4!Invoke+0x2a
RPCRT4!NdrStubCall2+0x333
RPCRT4!NdrServerCall2+0x19
RPCRT4!DispatchToStubInCNoAvrf+0x55
RPCRT4!RPC_INTERFACE::DispatchToStubWorker+0x1dd
RPCRT4!RPC_INTERFACE::BeginNullManagerCall+0x17
RPCRT4!LRPC_SCALL::DispatchRequest+0x25e
RPCRT4!LRPC_SCALL::QueueOrDispatchCall+0x111
RPCRT4!LRPC_SCALL::HandleRequest+0x376
RPCRT4!LRPC_SASSOCIATION::HandleRequest+0x1a2
RPCRT4!LRPC_ADDRESS::HandleRequest+0xe4
RPCRT4!LRPC_ADDRESS::ProcessIO+0x554
RPCRT4!LrpcServerIoHandler+0x16
RPCRT4!LrpcIoComplete+0x16
ntdll!TppAlpcpExecuteCallback+0x15e
```

3. Pas de réel consensus sur l'acronyme : `Advanced / Asynchronous [Local Procedure Call] [Local Inter-Process Communication]`

```
ntdll!TppWorkerThread+0x384
KERNEL32!BaseThreadInitThunk+0xe
ntdll!__RtlUserThreadStart+0x4a
ntdll!_RtlUserThreadStart+0x1c
```

Comme on peut le voir ci-dessus l'application WinRT est lancée via l'API win32 `KERNELBASE!CreateProcessAsUserW`. Durant la création du processus au travers de la fonction `CreateProcessAsUserW` un appel particulier est exécuté, comme montré ci-dessous :

```
0:005> kb
ntdll!NtCreateLowBoxToken
KERNELBASE!BasepCreateLowBox+0x142
KERNELBASE!CreateProcessInternalW+0x1924
KERNELBASE!CreateProcessAsUserW+0x2d
rpcss!CClsidData::PrivilegedLaunchRunAsServer+0x542
rpcss!_LaunchRunAsServer+0xb9
...
```

L'appel à la fonction `NtCreateLowBoxToken` du module `ntdll` est un appel majeur dans la création de la sandbox WinRT. Cet appel sera discuté en détail dans un prochain chapitre.

0.3.7 Microsoft Windows Store

Le magasin d'applications Microsoft Windows (*Windows Store*) est une plate-forme de distribution numérique à venir et est la plate-forme de fait pour les applications Metro / WinRT. De manière concise, le Windows Store est une pièce centrale pour les applications Metro et incidemment toutes les applications développées pour WinRT : les développeurs passeront nécessairement par cette plate-forme pour proposer leurs applications au monde parce qu'il n'existe pas de concept d'installation locale pour les applications WinRT. Le modèle de distribution des applications WinRT est donc centré exclusivement sur le Windows Store.

Les applications allant sur le Windows Store doivent être signées et devront nécessairement passer par un processus de vérification avant d'être pleinement disponibles dans le magasin d'applications. Microsoft propose ainsi un outil (à savoir le « Windows App Certification Kit ») destiné aux développeurs afin de vérifier localement sur leur machine de développement si leurs applications passeront le processus de certification distant (les vérifications locales et distantes sont, d'après Microsoft, basées sur le même outil).

Les applications WinRT sont contraintes d'utiliser le framework WinRT ou un ensemble restreint de l'API de COM et/ou Win32 [25]. Il est tout de même possible de contourner cette restriction et d'appeler des API (ou même d'obtenir des structures internes de bas niveau du système d'exploitation) qui ne sont habituellement pas autorisées.

Du fait que les applications WinRT natives sont, par essence, compilées avec le jeu d'instructions du processeur, il est difficile d'interdire totalement l'utilisation d'API faite par des biais détournés. Il est ainsi possible d'utiliser des API interdites depuis une application WinRT en utilisant une technique très employée dans les *shellcodes* qui consiste d'abord à récupérer un pointeur vers une structure interne du système d'exploitation nommée `PEB` (*Process Environment block*).

Cette structure contient un pointeur vers une autre structure nommée TEB (*Thread Environment block*, une structure par *thread* du processus). De cette structure il est possible de récupérer les noms et les adresses de base de tous les modules (binaire principal et bibliothèques dynamiques) chargés dans l'espace d'adressage du processus.

Connaissant le nom et l'adresse de base d'un module il est alors possible de *parser* celui-ci au format *Portable Executable*, d'aller chercher la table d'export (qui est, de manière schématique, une table de noms de fonctions et de pointeurs associés à ces mêmes fonctions). Une fois que le pointeur d'une fonction déterminée est obtenu, il devient alors possible d'appeler cette dernière.

Notez que tout le processus décrit est dynamique, ne fait appel à aucune fonction en particulier et nécessite tout au plus une centaine de lignes en C.

Il semble que le processus de vérification des applications en local ne fasse cette vérification que de manière statique sur les binaires vérifiés (vérification de la table d'import des fonctions) et non dynamiquement, ce qui ne permet pas de capter / d'appréhender la résolution dynamique des fonctions telle que présentée ci-dessus.

Il nous a été possible de démarrer un *shell* local sur une machine en utilisant la technique sus-mentionnée puis de faire vérifier l'application qui démarrait ce *shell* avant un hypothétique téléchargement sur le Windows Store : le résultat est que l'application passe le contrôle, notamment celui chargé de vérifier qu'aucune API interdite (comme celle qui permet de démarrer un processus) n'est utilisée.

Tout appel de fonction, même si cet appel n'est en théorie pas permis, reste soumis à la *sandbox* de WinRT. Notez qu'un programme démarré via un processus WinRT par l'intermédiaire de la fonction `CreateProcess()` (ou toute fonction similaire) est limité dans son champ d'action puisque le processus ainsi démarré hérite du jeton d'accès restreint du processus WinRT parent.

Microsoft, contacté par nos soins, a admis qu'il existait un problème du côté de la vérification en local des applications mais que des vérifications supplémentaires étaient effectuées sur le Windows Store lui-même avant qu'une application ne soit mise à disposition du public. Nous ignorons toutefois quelle est la nature de ces vérifications.

0.4 Sandboxes

Ce chapitre va détailler l'implémentation de deux modèles de *sandbox* (bac à sable) dans l'environnement Windows. Le premier est celui de Chrome qui permet d'isoler un onglet assurant le rendu du code HTML du navigateur[24], limitant ainsi l'accès aux ressources en cas de compromission. Le second est celui utilisé par Microsoft pour isoler les applications *WinRT* des autres applications. L'intérêt est double car ce mécanisme permet de limiter l'accès au système en cas de compromission (comme pour Chrome) mais également pour une utilisation restreinte lorsque Windows fonctionne sur une tablette (comme avec un iPad).

Bien que l'utilisation et l'implémentation diffèrent, on retrouve un concept unique dans les *sandboxes* : un processus dit *broker* qui possède des droits ¡¡ classiques ¿¿ et les processus *sandbox*-és qui ont des droits très restreints. À chaque action nécessitant des droits particuliers (par exemple, ouvrir un fichier sur un répertoire particulier), la *sandbox* va effectuer une requête auprès du processus *broker*, si cette demande est acceptée, il va alors effectuer l'action et lui renvoyer un identifiant permettant à la *sandbox* d'effectuer d'autres requêtes avec ou sans le *broker*.

La comparaison des *sandboxes* sera effectuée parmi les critères suivants : *broker*, droit des *sandbox*es, communication entre *broker* et *sandbox* et gestion des droits.

0.4.1 Sécurité de Windows

Avant de voir en détail l'implémentation d'une *sandbox* sous Windows, il est important de connaître les concepts de sécurité offerts par Windows. Si avec des systèmes d'exploitations basés sur UNIX tout est fichier, avec Windows tout est ¡¡ objet ¿¿. Un objet permet de représenter une ressource (comme un fichier, un répertoire, une entrée dans la base des registres, un processus ...), et peut être manipulé par l'espace utilisateur à l'aide d'un `HANDLE`[4]. Les objets sont représentés dans le noyau à l'aide de la structure `_OBJECT_HEADER`, le champ qui nous intéresse ici est le `SecurityDescriptor` qui est du type `_SECURITY_DESCRIPTOR`[18]. Cette structure contient entre autre : un SACL (*System Access Control List*), qui permet de *log*guer des accès à un objet, et un DACL (*Discretionary Access Control List*), qui permet d'identifier les droits d'accès d'un objet en fonction de l'émetteur. Ces deux structures (*SACL* et *DACL*) sont du type `_ACL`[19] qui est un vecteur d'un autre type de structure nommé `_ACE`. Les `_ACE`s pouvant avoir différents rôles, il existe une pléthore de sous-structures. Les seules qui nous intéressent dans ce cas sont `_ACCESS_ALLOWED_ACE` et `_ACCESS_ALLOWED_ACE`, permettant respectivement d'autoriser ou de d'interdire un accès.

Listing 18– Structures *ACE*

```
1  typedef struct _ACE_HEADER {
2      UCHAR AceType;
3      UCHAR AceFlags;
4      USHORT AceSize;
5  } ACE_HEADER;
6
7  typedef struct _ACCESS_ALLOWED_ACE {
8      ACE_HEADER Header;
9      ACCESS_MASK Mask;
10     ULONG SidStart;
11 } ACCESS_ALLOWED_ACE;
12
13 typedef struct _ACCESS_DENIED_ACE {
14     ACE_HEADER Header;
15     ACCESS_MASK Mask;
16     ULONG SidStart;
17 } ACCESS_DENIED_ACE;
```

Le champ `Header` contient les informations de base d'un *ACE* et permet de différencier les types ; le champ `AccessMask` définit le type d'accès ; enfin le champ `SidStart` est un identifiant de sécurité (*Security Identifier* ou *SID*)[20] et peut correspondre à un utilisateur, un groupe, une machine, un droit particulier (utilisé par un processus), une session... Pour accéder à un objet, l'émetteur a besoin d'un autre objet nommé jeton d'accès (*Access Token*)[21]. Cet objet contient beaucoup d'informations comme le *SID* : de la session, du compte utilisateur, des groupes auxquels il appartient... Et comme tout objet, il contient également une *ACL* permettant de limiter son accès vis-à-vis d'un autre jeton d'accès.

Les demandes d'accès à un objet sécurisé passent par l'appel système `NtAccessCheck` et fonctionnent de la manière présentée en figure 12. Lors de la recherche de l'*ACE*, la fonction `NtAccessCheck` (décrite dans la

4. Certains `HANDLE`s sont en réalité des pointeurs sur fonction dans l'espace mémoire utilisateur.

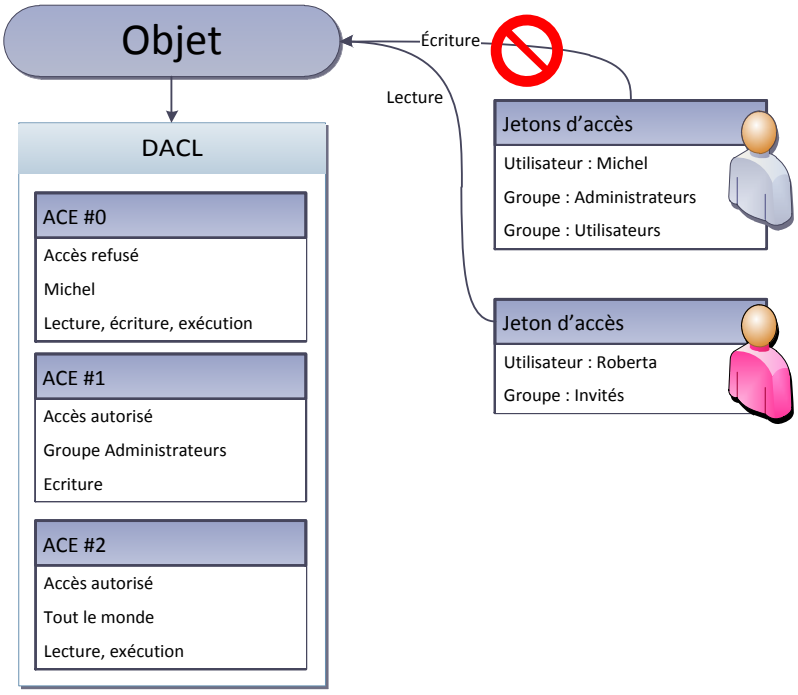

FIGURE 12 – Fonctionnement interne de NtAccessCheck

figure 12) parcourt les *ACE*s dans l'ordre du tableau : si un *ACE* correspond à un *SID*, la fonction s'arrête et renvoie le résultat (accès accepté ou refusé) ; si aucun *ACE* n'est trouvé, `NtAccessCheck` renvoie *STATUS_ACCESS_DENIED* pour signaler que l'accès est refusé. Enfin, si un objet ne possède pas de *DACL*[5] (nul, et non vide !), l'accès est toujours autorisé.

Comme nous l'avons vu, les objets contiennent des droits. Si nous souhaitons que notre *sandbox* n'ait pas le droit de manipuler ces objets, nous avons tout intérêt à limiter le jeton d'accès de nos *sandboxes*. Pour simplifier cette tâche, Microsoft a introduit le concept de jeton d'accès restreint (*restricted access token*), dont le fonctionnement est contraire à celui vu précédemment : `NtAccessCheck` va vérifier de la même façon si une entrée correspond au jeton, à la différence que l'accès sera refusé si le SID et l'ACE correspondent.

Les *sandboxes* s'exécutant sur un système Windows Vista ou supérieur peuvent utiliser les niveaux d'intégrités (*integrity level*)[22]. Bien qu'il existe quatre niveaux prédéfinis : *Low Mandatory Level* ; *Medium Mandatory Level* ; *High Mandatory Level* et *System Mandatory Level*, la plupart des processus utilisent le niveau *Medium Mandatory Level*. Dans le cas d'une *sandbox*, les processus isolés utiliseront le niveau *Low Mandatory Level*, permettant ainsi d'autoriser l'écriture uniquement sur le répertoire ¡¡ `%UserProfile%\AppData\LocalLow` ¿¿ et dans la base registre ¡¡ `HKEY_CURRENT_USE\Software\AppDataLow` ¿¿. Les différences de niveaux peuvent également permettre d'utiliser le UIPI (*User Interface Privilege Isolation*). Cette fonctionnalité permet de limiter le type de message qui peut être envoyé (en particulier ceux liés à l'écriture et à l'exécution) si l'émetteur du message a un niveau inférieur au destinataire, cette sécurité vise à protéger contre l'attaque *shatter*. Il est à noter que contrairement au mode *seccomp* du *kernel* Linux, ce mécanisme n'interdit pas au processus en *Low Mandatory Level* d'utiliser les appels systèmes, seul son jeton d'accès limitera son périmètre d'action.

0.4.2 Le modèle de Chrome

Broker

Le *broker* est simplement le processus *chrome.exe* et représente le processus père des *sandboxes*. En cas de mort, les *sandboxes* sont automatiquement détruites.

5. C'est le cas sur les partitions de type *FAT*

Droit de la *sandbox*

Chrome utilise le jeton d'accès le plus restrictif possible sous Windows, à savoir :

SID	Flags
Logon SID	obligatoire
Tout le monde	obligatoire, interdit
RESTRICTED SID	obligatoire, restreint
Intégrité basse	intégrité

Le jeton d'accès ne possède aucun privilège. Il est à noter que Chrome limite l'accès à l'utilisation des ressources de la machine avec les *Job*s et simule un *UIPI* en créant un nouveau bureau [6].

Communication

Il existe deux mécanismes différents pour communiquer entre la *sandbox* et le broker : le *named pipe* (tuyau nommé) et un système qui utilise une *shared memory* (mémoire partagée). Nous nous focaliserons uniquement sur la dernière méthode (représentée par la figure 13) car elle est utilisée pour les accès au ressource système. Une des subtilités de l'implémentation de la *sandbox* de Chrome est d'utiliser une technique de *hooking* [7] (*interception*) pour remplacer les fonctions Windows vers ses propres routines qui vont communiquer avec le broker. L'utilité de ce mécanisme est de mettre en *sandbox* des modules dont les sources ne sont pas disponibles, en aucun cas cela permet de renforcer la sécurité.

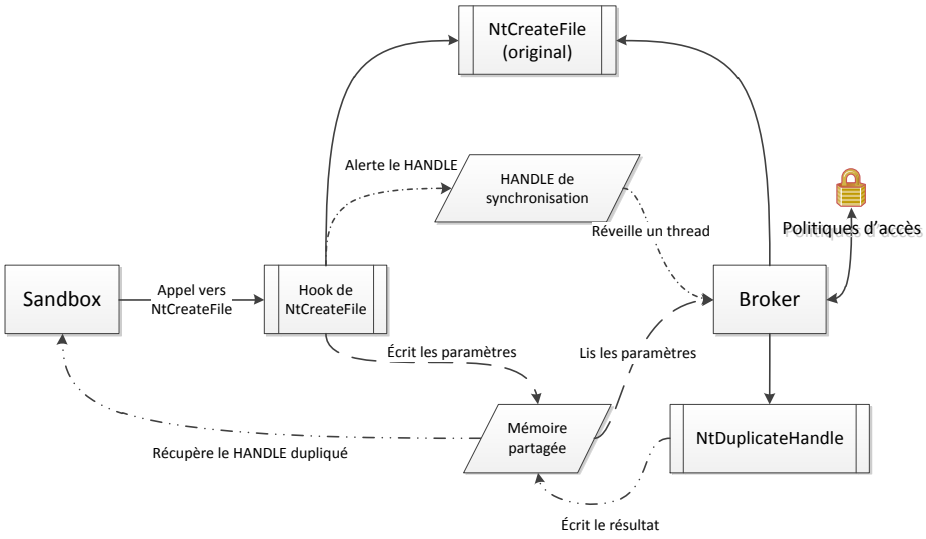

FIGURE 13 – Vue d'ensemble de la sandbox Chrome

La *sandbox*, en essayant d'appeler `NtCreateFile`, va appeler `TargetNtCreateFile` qui va s'occuper de traiter la demande d'ouverture du fichier. Cette fonction va commencer par essayer d'ouvrir le fichier avec les droits de la *sandbox*, si l'opération réussit, la fonction va directement renvoyer l'*HANDLE*. Si non les paramètres vont être écrits dans la mémoire partagée et le *HANDLE* de synchronisation, que les processus ont en commun, va être signalé à l'aide de la fonction `SignalObjectAndWait`. Le *thread* du *broker* va alors vérifier si la *sandbox* a bien le droit d'accéder au fichier. Si tel est le cas, la fonction `NtCreateFile` sera appelée de nouveau mais avec les droits du broker. Si la fonction réussit, le *HANDLE* va être dupliqué pour le contexte processus de la *sandbox* avec `DuplicateHandle`, et le résultat sera écrit dans la mémoire partagée. Dès que le broker aura terminé son traitement, il va notifier à la *sandbox* que l'opération s'est terminée en utilisant la fonction `SetEvent`. La *sandbox* pourra consulter la mémoire partagée pour obtenir le résultat de sa requête.

6. Le périmètre des messages Windows sont limités au bureau
7. en modifiant les adresses dans *EAT* (Export Address Table) pour l'API classique et un *inline hook* pour les appels systèmes

Gestion des droits

Les développeurs de Chrome ont décidé de refaire un système de politique d'accès permettant une configuration très fine. L'ajout d'une règle utilise la méthode `PolicyBase::AddRule` qui nécessite le type de sous-système (fichier, processus, tuyau nommé...), l'action qui est dépendant au sous-système (accès autorisé vers un fichier, création d'un processus...) et un *pattern*. Voici un exemple extrait de la documentation officielle permettant l'accès en lecture aux fichiers de *dump* commençant par « d » et ayant l'extension *.dmp* dans le répertoire `C:\temp` :

```
AddRule(SUBSYS_FILES, FILES_ALLOW_READONLY, L"c:\temp\app_log\d*.dmp");
```

0.4.3 Le modèle WinRT

Contrairement à son homologue Google, Microsoft a les pleins pouvoirs sur les sources de son système d'exploitation. Cette différence lui permet de modifier de façon intrusive la sécurité. Cela explique le changement majeur de la structure `_TOKEN` qui représente un jeton d'accès. Ces ajouts sont :

- un pointeur vers un *SID* du paquet (*Package*) ;
- un tableau de *SID* contenant les capacités (*Capabilities*) ;
- une structure contenant le numéro de la *lowbox* (*LowboxNumberEntry*) ;
- une structure contenant les *HANDLE*s de la *lowbox* (*LowboxHandlesEntry*).

Broker

Le processus broker est en réalité une interface COM contenue dans l'exécutable *RuntimeBroker.exe*. Il est lancé automatiquement par *svchost.exe* en cas de requête d'une *sandbox*. En cas de mort, le processus se relancera automatiquement, sans détruire les processus WinRT. Il est à signaler que certaines applications jouissent de leurs *brokers* personnalisés, c'est le cas pour *Remote Desktop* qui utilise l'exécutable wkspbroker.exe (RemoteApp and Desktop Connection Runtime Broker). Le *broker* WinRT utilise l'*impersonation*[23] (ou en français ¡¡ imitation ¿¿), ce procédé permet à un *thread* d'utiliser le jeton d'accès de la *sandbox* et de tester l'accès à des objets dans un autre contexte de sécurité. La fonction utilisée dans un contexte *COM* est `CoImpersonateClient` qui va permettre de prendre l'identité sécuritaire du client (ici, la *sandbox*).

Droit de la *sandbox*

Le jeton d'accès de la *sandbox* est généré à partir du nouvel appel système `NtCreateLowBoxToken`[8]. Cette fonction effectue les actions suivantes :

- duplique le jeton d'accès spécifié en paramètre ;
- réduit les droits du jeton d'accès à *SeChangeNotifyPrivilege* et *SeIncreaseWorkingSetPrivilege* ;
- passe le niveau d'intégrité du jeton d'accès à bas (*low*);
- initialise les champs : paquet, capacités, numéro, *HANDLE*s, du jeton d'accès ;
- modifie les droits d'accès vers le jeton à : lui-même avec *TOKEN_ALL_ACCESS* et Administrateurs avec *TOKEN_QUERY*.

Le jeton d'accès de la *sandbox* contient les éléments suivants :

SID	Flags
Logon SID	obligatoire
Tout le monde	obligatoire
SID du paquet	containeur d'application
SID des capacités (si configurés)	capacité
intégrité basse	intégrité

Les privilèges accordés sont :

- SeChangeNotifyPrivilege (activé par défaut)

8. NTSTATUS NtCreateLowBoxToken(PHANDLE pLowBoxTokenHandle, HANDLE hProcessToken, ACCESS_MASK AccessMask, POBJECT_ATTRIBUTES pObjAttr, PSID pPackageSid, ULONG CapabilitySidCount, PSID_AND_ATTRIBUTES pCapabilitiesSid, ULONG HandleArrayCount, PHANDLE HandleArray)

- SeIncreaseWorkingSetPrivilege (désactivé)
- SeShutdownPrivilege (désactivé)
- SeTimeZonePrivilege (désactivé)
- SeUndockPrivilege (désactivé)

Il existe une petite subtilité qui fait que ces privilèges ne sont pas ¡¡ exploitables ¿¿ par l'application WinRT, bien qu'elle soit en mesure d'ouvrir un *HANDLE* vers le jeton d'accès avec le droit `TOKEN_ADJUST_PRIVILEGES` (qui permet de réactiver un privilège), il semblerait [9] que la fonction système `SepAdjustPrivileges` utilise la fonction `RtlSidDominates` pour savoir si le jeton qui émet la requête a un niveau d'intégrité supérieur à bas (*low*) avant de réactiver un droit. Si ce n'est pas le cas, la demande ne sera pas traitée. Comme toutes les *sandboxes* sont en intégrité basse, ces privilèges ne sont pas réactivables.

Communication

La communication entre le broker et la *sandbox* (représentée par la figure 14) repose sur une technologie ancienne de Microsoft, le *COM*. L'avantage de ce choix réside dans le fait qu'un objet COM peut s'exécuter soit en local soit dans un autre processus ou même sur un autre ordinateur. Dans la nouvelle API de WinRT, la plupart des méthodes possédent le suffixe *Async*, ce sont ces méthodes qui vont, en interne, appeler la fonction `ObjectStubless`. Cette fonction est responsable de l'envoi de la requête vers le broker. Pour réaliser cette tâche, elle va *marshall*iser l'objet représentant la requête (par exemple `__FIAsyncOperation_1_Windows_CStorage_CStorageFile` [10]) avec `Ndr(64)pClientMarshal`, utiliser la fonction `NdrExtpProxySendReceive` qui va appeler `NtAlpcSendWaitReceivePort` pour communiquer avec le broker en utilisant un port ALPC [11]. Une fois la réponse obtenue et *unmarshall*isée, et si la fonction renvoie un paramètre, la routine présente dans `IAsyncAction::Completed` sera appelée.

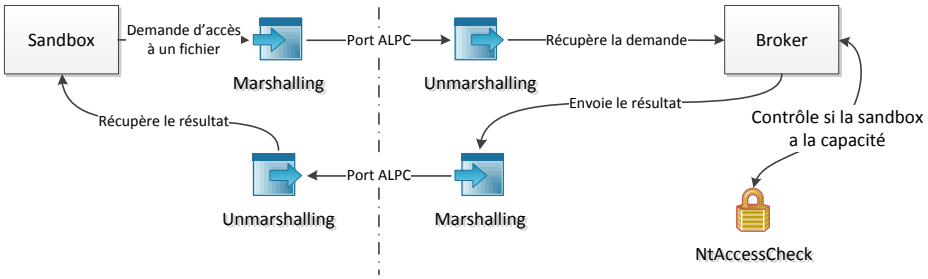

FIGURE 14 – Vue d'ensemble de la sandbox WinRT

Gestion des droits

Les droits sont organisés sous forme de capacités.

SID	Nom
S-1-15-3-1	Your Internet connection
S-1-15-3-2	Your Internet connection, including incoming connections from the Internet
S-1-15-3-3	A home or work network
S-1-15-3-4	Your pictures library
S-1-15-3-5	Your videos library
S-1-15-3-6	Your music library
S-1-15-3-7	Your documents library
S-1-15-3-8	Your Windows credentials
S-1-15-3-9	Software and hardware certificates or a smart card
S-1-15-3-10	Removable storage

9. Ce comportement n'est pas décrit sur MSDN
10. La routine traitant cette requête se situe dans shell32.dll
11. Pas de réel consensus sur l'acronyme : `Advanced / Asynchronous [Local Procedure Call] [Local Inter-Process Communication]`

Lors de la réception d'une requête par le *broker*, ce dernier va prendre l'identité sécuritaire (*impersonate*) de la *sandbox* à l'aide de la fonction `CoImpersonateClient`, puis il va récupérer son jeton d'accès pour ensuite appeler la fonction `RtlCheckTokenCapability` avec en paramètres : le jeton d'accès, le *SID* de la capacité et un pointeur vers un booléen spécifiant si la *sandbox* a accès ou non à cette capacité. La fonction qui vérifie les accès va alors construire un descripteur de sécurité avec le SID de l'application et celui de la capacité, pour ensuite appeler `NtAccessCheck` avec le jeton d'accès de la *sandbox*. Si la fonction réussit, le *broker* effectue l'action demandée et communique le résultat à la *sandbox*, dans le cas contraire, la *sandbox* recevra une erreur.

Il est important de savoir que les *declarations* peuvent également limiter les accès aux capacités. Par exemple, chaque application doit préciser l'extension supportée, si l'extension n'est pas explicitement déclarée le broker refusera l'accès. On peut souligner que certaines extensions sont réservées au système, comme les *.dll*.

0.4.4 Confrontation des modèles de *sandboxes*

Pour mieux comparer les deux modèles de *sandboxes*, nous allons réutiliser les critères vus précédemment.

Broker

Le broker de WinRT semble plus robuste dans son indépendance vis-à-vis des *sandboxes*. Si le broker meurt de manière inopinée, il sera relancé automatiquement et les *sandboxes* seront toujours actives. Sous Chrome, si le processus *broker* meurt, ces processus enfants (ici, les *sandboxes*) mourront avec lui.

Droit des *sandboxes*

Le design de Google offre une restriction plus forte. Lors de sa création, la *sandbox* n'a pratiquement accès à rien et les ressources du système lui sont limitées. Dans le design de Microsoft, la *sandbox* possède un jeton d'accès pour manipuler son propre répertoire et ne possède aucune restriction sur les ressources du système. Ce choix peut permettre à une application WinRT malvaillante de créer une multitude de thread (en utilisant la technique évoquée auparavant) pour bloquer tout le système.

Communication

La méthode employée par Chrome est particulièrement intéressante en terme de performance : peu de données sont échangées et les échanges sont réalisés sur une mémoire partagée. En revanche, comme elle repose sur des *hook*, les fonctions sont intrinséquement synchrones. Microsoft a fait le choix d'imposer un modèle asynchrone du fait que les applications peuvent être suspendues, pour éviter les problèmes des applications bloquées et de la lenteur potentielle de l'API win32. Ce modèle semble plus adapté dans un environnement *sandbox*-é, car il est possible que la communication avec le broker soit rompue.

Gestion des droits

Le fonctionnement de la *sandbox* Microsoft repose sur une API déjà éprouvée. Bien qu'elle permette un réglage moins fin, elle reste plus simple à configurer et est visuellement plus simple à observer (présence des SIDs dans le jeton d'accès).

0.5 Conclusion

Le concept de WinRT dans son entièreté est certainement un paradigme très fort pour Microsoft et ce paradigme sera probablement amené à se développer dans l'avenir – peut-être même comme un remplacement possible d'une partie l'API Win32 et de son sous-système.

Via son Windows Store et ses applications *packagées*, Microsoft suit ses concurrents – notamment Apple et Google – sur le marché des applications faciles à installer, à utiliser et prêtes a l'emploi.

De part son nouvel environnement, Microsoft continue de mettre l'accent sur la sécurité. La mise en place des niveaux d'intégrité et des capacités nous semble offrir une *sandbox* convenable et, d'après nos recherches, sans défauts majeurs puisque s'appuyant sur des éléments éprouvés du système d'exploitation. Cela ne veut toutefois pas dire que la *sandbox* (ou le modèle de sécurité entier de WinRT) n'est pas faillible. Le *broker* ou le tunnel de communication entre l'application et le *broker* peuvent être des talons d'Achille.

On constate également une mise en avant de la fiabilité des exécutables avec le « tout asynchrone ». Avec cette nouvelle interface Microsoft vise inévitablement le marché des tablettes, à l'instar d'Apple, et tout porte à croire (on pensera ici notamment aux concepts de *sandbox*, `UEFI` et son *secure boot*, ...) que les systèmes d'exploitation Windows cloisonneront de plus en plus leur environnement et qu'un accès limité sera offert aux utilisateurs de ces systèmes. Il ne serait pas étonnant de voir dans un futur proche des *jailbreaks*[12] pour Windows 8.

12. Échappement de *sandbox*

Bibliographie

[1] http://www.microsoft.com/presspass/press/2011/jan11/01-05SOCsupport.mspx

[2] http://msdn.microsoft.com/en-us/library/windows/apps/hh464920.aspx

[3] http://msdn.microsoft.com/en-us/library/windows/apps/br211377.aspx

[4] http://en.wikipedia.org/wiki/Extensible_Application_Markup_Language

[5] http://msdn.microsoft.com/en-us/library/br205821(v=vs.85).aspx

[6] http://msdn.microsoft.com/en-us/library/windows/apps/hh748265(v=vs.110).aspx

[7] http://msdn.microsoft.com/en-us/library/windows/apps/windows.storage.bulkaccess.fileinformation

[8] http://msdn.microsoft.com/en-us/library/windows/apps/hh699871(v=vs.110).aspx

[9] http://msdn.microsoft.com/en-us/library/br224646(v=vs.85).aspx

[10] http://msdn.microsoft.com/en-us/library/windows/apps/hh454071(v=vs.110).aspx

[11] http://msdn.microsoft.com/en-us/library/te3ecsc8(v=vs.110).aspx

[12] http://msdn.microsoft.com/en-us/library/br205779(v=vs.85).aspx

[13] http://msdn.microsoft.com/en-us/library/windows/apps/hh755822(v=vs.110).aspx

[14] http://www.ecma-international.org/publications/standards/Ecma-335.htm

[15] http://msdn.microsoft.com/en-us/library/windows/apps/hh464906(v=VS.85).aspx

[16] http://msdn.microsoft.com/en-us/library/windows/apps/hh464936.aspx

[17] http://msdn.microsoft.com/en-us/library/windows/desktop/hh448474(v=vs.85).aspx

[18] http://msdn.microsoft.com/en-us/library/windows/desktop/aa379563(v=vs.85).aspx

[19] http://msdn.microsoft.com/en-us/library/windows/desktop/aa374872(v=vs.85).aspx

[20] http://msdn.microsoft.com/en-us/library/windows/desktop/aa379571(v=vs.85).aspx

[21] http://msdn.microsoft.com/en-us/library/windows/desktop/aa374909(v=vs.85).aspx

[22] http://msdn.microsoft.com/en-us/library/bb625963.aspx

[23] http://msdn.microsoft.com/en-us/library/windows/desktop/ms691341(v=vs.85).aspx

[24] http://www.chromium.org/developpers/design-documents/sandbox

[25] http://msdn.microsoft.com/en-us/library/windows/apps/br205757.aspx

[26] http://msdn.microsoft.com/en-us/library/windows/apps/hh464929.aspx


QUARKS
LAB
INNOVATIVE
SECURITY