# Audit d'applications .NET
# Le cas Microsoft OCS 2007 (R1 et R2)

Nicolas Ruff
nicolas.ruff(@)eads.net

EADS Innovation Works

## 1 Disclaimer

« .NET » est une technologie Microsoft dont les détails d'implémentation sont assez peu documentés, et dont les spécifications évoluent rapidement.

C'est pourquoi le lecteur avisé qui relèverait quelque coquille dans le présent document, ou qui aurait des compléments d'information à apporter, est prié de prendre contact avec moi. Il sera assuré du meilleur accueil, car je désespère de rencontrer des gens manifestant un intérêt profond pour cette technologie. Je ne parle pas de développer des interfaces WPF exotiques en dupliquant des codes trouvés sur Internet, mais bien de comprendre le *bytecode* .NET et l'implémentation de la machine virtuelle sous-jacente...

## 2 L'audit, pourquoi ?

Avant de rentrer dans le vif du sujet, il faut commencer par se poser la question préliminaire : « pourquoi auditer une application telle que Microsoft OCS 2007 » ? Cette interrogation est légitime à plusieurs points de vue :

- Compte-tenu de la taille et de la complexité de cette application, l'audit en « boite noire » représente un effort important qu'un client final n'est pas forcément prêt à consentir dans le cadre d'un simple déploiement.
- On peut supposer que Microsoft a déjà élagué les bogues les plus triviaux, dans le cadre de son processus SDL [1]. L'audit n'en sera que plus complexe.

1. Security Development Lifecycle : http://www.microsoft.com/security/sdl/

– Enfin, si des bogues critiques sont trouvés malgré tout, leur correction ne sera possible que par Microsoft puisque nous sommes en présence d'une application commerciale *Closed Source*.

Le cas de l'audit offensif (c'est-à-dire visant à trouver des bogues dans les applications utilisées par la concurrence) est laissé à part, même si l'affaire « Aurora » démontre que ce type d'audit se pratique [2].

Certains clients ont une politique de sécurité stricte - voire sont soumis à des réglementations ou des standards - qui les astreignent à auditer toute nouvelle application mise en production. Ce type de contrainte mal comprise peut toutefois rapidement tourner au « scan Nessus [3] », c'est-à-dire à la recherche de vulnérabilités connues dans des produits qui ne le sont pas encore.

Faire pratiquer un audit de sécurité sur une application telle que Microsoft OCS 2007 (ou toute autre application de même ampleur pour laquelle la littérature sécurité reste vierge) présente à mon avis de multiples intérêts :

1. Identifier les briques logicielles de base

   Aucune application d'envergure n'est conçue *ex nihilo* aujourd'hui. De nombreux composants logiciels sont réutilisés, comme des librairies Open Source (ex. zlib, libpng), des librairies Closed Source (ex. Autonomy KeyView, source de tant de vulnérabilités dans les produits Lotus Notes, BlackBerry et Symantec), voire des échafaudages complexes (ex. Sun JRE, Tomcat).

   Connaitre les composants réutilisés permet d'affiner son exposition au risque. Il n'est pas rare de voir des éditeurs livrer une version complète de la JVM Sun avec leur produit, sans jamais alerter leurs clients sur les mises à jour de sécurité proposées par Sun, ni même supporter officiellement la migration vers une JVM plus récente, par exemple.

2. Evaluer les vecteurs de compromission

   Une application écrite en langage C risque d'être beaucoup plus vulnérable aux failles de type *Buffer Overflow* que la même application écrite en langage C# ou Java.

2. http://siblog.mcafee.com/cto/source-code-repositories-targeted-in-operation-aurora/
3. Nessus étant par ailleurs un très bon outil d'audit !

Une stratégie de défense en profondeur consistera dans le premier cas à activer DEP sur le serveur hôte, tandis que dans l'autre ce seront les options de configuration de la machine virtuelle (CLR ou JRE) qui seront affinées.

3. Comprendre le fonctionnement interne de l'application

   Il est regrettable de constater que les intégrateurs, les consultants experts, et même les partenaires « Gold » d'une solution logicielle complexe la maitrise rarement au-delà des interfaces graphiques et de la base de connaissance des bogues connus.

   L'audit sécurité permet d'acquérir une connaissance intime des mécanismes audités, ce qui permet (dans une certaine mesure) d'anticiper les problèmes plutôt que de les subir. Que va-t-il se passer à l'expiration d'un certificat ? Où sont journalisées les traces de débogage de l'application ? Les protocoles propriétaires utilisés traversent-ils les mécanismes de NAT ? Toutes ces questions trouvent souvent leur réponse lors de l'analyse de l'implémentation technique.

   Par ailleurs il faut noter que de plus en plus d'éditeurs se tournent vers la technologie « .NET », pour différentes raisons telles que :

   – La rapidité et la facilité de développement ;
   – La disponibilité en masse de compétences .NET chez les développeurs ;
   – La bonne pénétration du *runtime* .NET sur les postes client, qui assure aux applications .NET la compatibilité avec le parc existant (Windows XP) et futur (Windows Seven et au-delà).

   ... et probablement plein d'autres bonnes raisons techniques qui m'échappent (ex. déploiement ClickOnce, compatibilité 64 bits, etc.).

   Dès lors, acquérir des outils et des méthodologies de travail en environnement .NET est forcément un pari d'avenir. Bonne lecture !

# 3 Introduction à .NET

## 3.1 Présentation générale

Microsoft investit énormément dans la technologie « .NET », au point d'envisager de remplacer le noyau Windows actuel par un mi-

cronoyau basé sur .NET (projet Midori [4] basé sur Singularity [5], qui pourrait devenir Windows 8).

Cette technologie est d'ores et déjà critique pour Microsoft, puisqu'elle est au cœur de plusieurs applications phares comme SilverLight (présenté comme le concurrent de Flash) ou Azure (la plateforme de Cloud Computing), pour ne citer qu'eux.

Le Framework .NET se compose de plusieurs éléments :

1. La spécification d'un *bytecode* : *Common Language Infrastructure* (CLI).
2. Une machine virtuelle, permettant d'exécuter ce *bytecode* sur les systèmes Windows : *Common Language Runtime* (CLR). Il existe également des implémentations Open Source, pouvant exécuter du *bytecode* .NET sur d'autres systèmes d'exploitation, comme le projet Mono.
3. Des librairies de base : *Base Class Library* (BCL). Ces librairies doivent être disponibles dans tous les environnements d'exécution .NET.
4. Des librairies additionnelles : *Framework Class Library* (FCL).

De nombreux langages peuvent être compilés en *bytecode* .NET, dont les plus connus sont C#, VB.NET et ASP.NET – mais on peut citer également J# (un clone de Java dont nous reparlerons), F# (un langage fonctionnel), IronPython ou... COBOL.NET. Il est même possible d'utiliser le langage C++ – on parle alors de C++/CLI (anciennement *Managed C++*).

Les éléments essentiels sont standardisés ECMA et ISO [6] :
- ECMA-334 pour le langage C#
- ECMA-335 et ISO/IEC 23271 :2006 pour CLI et BCL
- ECMA-372 pour le langage C++/CLI

## 3.2 Versions

Initialement, .NET était destiné à contrer Java, dont Microsoft a été privé d'usage suite à un procès retentissant avec Sun [7]. C'est

4. http://fr.wikipedia.org/wiki/Midori_syst%C3%A8me_d'exploitation)
5. http://fr.wikipedia.org/wiki/Singularity
6. http://msdn.microsoft.com/en-us/netframework/aa569283.aspx
7. http://en.wikipedia.org/wiki/Microsoft_Java_Virtual_Machine#Sun_vs._Microsoft

ainsi que la version 1.0 du Framework .NET (rapidement suivie de la version 1.1) a vu le jour.

Une version 2.0 est ensuite apparue, qui est aujourd'hui (de mon expérience) la version la plus utilisée pour diverses raisons : installation facile sous Windows XP, légèreté du Framework, importante liste de compilateurs supportés, etc.

Les versions du Framework suivent les versions de Visual Studio, tout en conservant une compatibilité ascendante :

- Visual Studio 2002 -> Framework 1.0
- Visual Studio 2003 -> Framework 1.1
- Visual Studio 2005 -> Framework 2.0
- Visual Studio 2008 -> Framework 3.5
- Visual Studio 2010 -> Framework 4.0

# 4 Sécurité du bytecode .NET

## 4.1 Avantages...

Les langages générant du code .NET (tels que C#) présentent des avantages certains pour la sécurité des développements par rapport aux langages traditionnels C/C++. En cela ils sont comparables aux autres langages modernes (tels que Java, Python, Ruby, etc.).

L'objet n'est pas d'établir ici une liste exhaustive des avantages de .NET, d'autant que cette liste pourrait prêter à discussion. Voici toutefois quelques points clés :

- L'apport le plus évident est le typage fort des données, y compris au niveau du *bytecode* (donc à l'exécution), ce qui évite les manipulations hasardeuses de pointeurs ou les « *buffer overflows* ».
- Le développeur n'a plus à se soucier des allocations mémoire grâce à la présence d'un *garbage collector*, ce qui évite les problèmes de type « *double free* » ou les références d'objets invalides.
- Le développeur dispose en standard d'une librairie réputée « sûre », ce qui évite les implémentations hasardeuses - comme dans le domaine de la génération d'aléa ou de la cryptographie.
- La machine virtuelle peut également appliquer une politique de sécurité à l'exécution, comme par exemple limiter l'accès

aux fichiers locaux ou les accès réseau. Cette politique est configurable localement par l'administrateur.
- Il existe un mécanisme de signature de code, pouvant être utilisé dans la politique de sécurité de la machine virtuelle.

La plupart des applications .NET remplacent des applications Windows « traditionnelles » et s'exécutent donc avec les droits complets de l'utilisateur sur le système hôte. La politique de sécurité de la machine virtuelle est rarement mise en œuvre, sauf si une application demande explicitement à s'exécuter avec des droits restreints.

Toutefois il existe des cas où du code .NET inconnu peut être exécuté sur une machine tierce : c'est le cas des applications SilverLight ou Azure par exemple. Dans ce cas, une politique de sécurité contraignante est appliquée par le système hôte.

## 4.2 ... et inconvénients

Pour commencer, il faut signaler que .NET ne protège évidemment pas contre les failles logiques (ex. injection SQL ou *backdoor*), même si le *bytecode* .NET est beaucoup plus facilement vérifiable (soit par analyse statique, soit par analyse dynamique) que du code C ou de l'assembleur x86, du fait qu'il conserve les types de données.

Ensuite il faut signaler que le code .NET autorise l'appel à des librairies du système (via P/Invoke[8]) ou des objets COM (via System.Runtime.InteropServices). Toute faille présente dans ces librairies pourra être déclenchée « classiquement » depuis un code .NET.

Pour parler de choses plus spécifiques à .NET, il faut savoir que .NET autorise malgré tout la manipulation de pointeurs dans les blocs de code marqués comme *unsafe*[9] en C# (et compilés avec l'option /unsafe). Le *bytecode* généré aura toutefois l'attribut « non vérifiable », ce qui ne devrait pas lui permettre de s'exécuter dans n'importe quel contexte de sécurité.

On jettera un voile pudique sur le mot clé *stackalloc*, qui permet d'allouer de l'espace dans la pile pour une variable locale. Ce mot clé n'est utilisable que dans le contexte *unsafe* vu précédemment.

Enfin, comme dans tout logiciel de taille conséquence, développé sans le support d'une preuve mathématique, il existe des bogues

8. http://fr.wikipedia.org/wiki/P/Invoke
9. http://msdn.microsoft.com/en-us/library/chfa2zb8%28VS.71%29.aspx

d'implémentation dans le Framework lui-même (qui vont être présentés ci-après).

**Historique des failles dans le Framework .NET** La JVM fournie par Sun est coutumière des failles de sécurité (le site Secunia en recense 112 uniquement pour la version 1.6, à la date de rédaction de ce document [10]).

La faille la plus notable dans la JVM ces dernières années est connue sous le nom de « util.calendar() » [11]. Elle a gagné ses galons médiatiques lors du concours « Pwn to Own », organisé chaque année lors de la conférence CanSecWest.

Comparativement, le Framework .NET semble relativement épargné puisqu'à la date de rédaction de cet article, seuls 5 bulletins de sécurité affectent le Framework .NET en version 2.0 :

- MS06-033 et MS06-056 concernent une fuite d'information et un XSS dans ASP.NET
- MS07-040 et MS09-061 corrigent plusieurs « vraies » failles d'évasion de la machine virtuelle
- MS08-052 et MS09-062 affectent GDI+, une librairie de support utilisée (entre autres) par .NET
- MS09-036 est un déni de service sur ASP.NET

Faut-il en conclure que le Framework .NET est plus sûr que Java ? Certainement pas, pour les raisons suivantes :

- Compte-tenu du taux de pénétration de la machine virtuelle Java et de la possibilité d'instancier sans confirmation des applets Java dans le navigateur, la technologie Java a fait l'objet de beaucoup plus de recherches de la part des attaquants potentiels.
- En parcourant les sites http://social.msdn.microsoft.com/Forums/ et http://connect.microsoft.com/, on peut se rendre compte que l'impact sécurité des bogues identifiés dans le Framework .NET est rarement qualifié. Seuls les chercheurs prenant contact directement avec le MSRC [12] ont une chance de voir leurs découvertes qualifiées de « faille de sécurité ». C'est également ce que déplore l'auteur du logiciel IKVM (par

10. http://secunia.com/advisories/product/12878/?task=advisories
11. http://blog.cr0.org/2009/05/write-once-own-everyone.html
12. http://www.microsoft.com/Security/msrc/default.aspx

ailleurs crédité pour plusieurs failles de sécurité dans le Framework .NET) sur son blog[13].

**Typologie des failles** De par leur complexité, aucune faille affectant le Framework .NET ne ressemble à une autre.

Les composants affectés par les bulletins publiés sont les suivants :

- Les librairies de support (écrites en code non managé de type C/C++), et plus particulièrement la librairie graphique GDI+.
- Le chargeur de fichiers au format PE (MS07-040 / CVE-2007-0041).
- Le compilateur JIT (MS07-040 / CVE-2007-0043).
- La logique même du Framework (MS09-061 – voir ci-dessous).

Devant une telle variété de problèmes, difficile de prévoir où frappera la foudre la prochaine fois.

**Exploitation de failles dans le Framework .NET** Attardons-nous un instant sur la faille MS09-061, puisque son auteur a souhaité publier tous les détails techniques sur son blog[14]. Il s'agit d'une faille d'évasion du Framework .NET (i.e. exécution de code natif x86) depuis une *assembly* qui s'exécute dans un contexte de sécurité restreint (ex. application SilverLight).

Pour commencer, l'auteur commence par décrire dans un autre billet[15] une méthode d'exécution de code x86 depuis une *assembly* provenant d'un contexte de sécurité non restreint (*Full Trust Assembly*).

Il s'agit d'utiliser le mot clé StructLayout[16] pour créer une union entre un objet et un tableau d'entiers, ce qui permet d'accéder à la même zone de mémoire via l'un ou l'autre point d'entrée. Il suffit alors de créer un objet de type *delegate*[17] (l'équivalent d'un pointeur de fonction en code managé) et de l'invoquer pour pouvoir exécuter du code natif x86 qui aura préalablement été écrit en mémoire.

13. http://weblog.ikvm.net/
14. http://weblog.ikvm.net/PermaLink.aspx?guid=d1c6348b-acb9-4997-82b0-10a85d70e22a
15. http://weblog.ikvm.net/PermaLink.aspx?guid=3cc8beef-3424-488d-8429-50e244f15ccc
16. http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.structlayoutattribute.aspx
17. http://msdn.microsoft.com/fr-fr/library/900fyy8e.aspx

Le mot clé StructLayout est utilisé dans le cadre de l'interopérabilité avec du code non managé, et permet de préparer les structures de données attendues par les API natives. On notera à la lecture des forums Microsoft susmentionnés que ce mot clé fût la cause de nombreux bogues pas toujours très clairs... Toutefois cette construction n'est autorisée que pour le code exécuté dans un contexte de sécurité non restreint.

Pour en revenir à MS09-061, la faille consiste en une vérification de type qui a été commentée dans le code du Framework .NET 2.0 (ainsi que dans sa version Open Source baptisée ROTOR [18]). Cette absence de vérification permet de combiner deux *delegate* de type différent, afin d'aboutir au même résultat que celui obtenu par la combinaison StructLayout/union, mais dans un contexte de sécurité restreint cette fois-ci.

La beauté de cette faille, c'est qu'elle est probablement la première pour laquelle un code d'exploitation en *bytecode* .NET a été publié...

### 4.3 Où sont les failles ?

Pour conclure sur la sécurité intrinsèque du Framework .NET, il faut noter que c'est un sujet à la fois extrêmement technique, passionnant, et quasiment vierge.

Il existe quelques failles connues permettant de détourner le flot d'exécution d'une application .NET lors du traitement d'une donnée fournie par l'utilisateur : ces failles affectent ASP.NET ou la librairie graphique GDI+. Les scénarios d'attaque sont donc limités, d'autant qu'aucun code d'exploitation public n'est disponible.

Toutes les autres failles connues nécessitent au préalable de pouvoir exécuter du code .NET sur la cible, ce qui limite leur impact au cas du *plug-in* SilverLight ou de l'hébergement de pages ASP.NET.

Lors de l'audit d'une application .NET, les failles sont donc plutôt à chercher :

- Dans la logique de l'application (ex. *backdoor*, injection SQL, etc.) ;

18. http://www.koders.com/csharp/fid0CEAECF1A5FE5FD63AD9A545B67380CA53D5CFFD.aspx?s=idef:system#L256

- Dans la mauvaise utilisation ou la réimplémentation hasardeuse des primitives sensibles (ex. chiffrement, génération d'aléa) ;
- Ou dans ses interfaces avec du code « non managé » (de type C/C++).

Passons désormais aux techniques permettant de mener à bien un tel audit.

# 5 Méthodes et outils d'audit

## 5.1 Code natif

Pour commencer, il faut signaler que le *bytecode* .NET est toujours compilé en code x86 au moment de l'exécution. Ce mécanisme de compilation à la demande est appelé JIT (*Just-In-Time*).

Il est donc possible d'analyser une application .NET avec un bon vieux débogueur comme OllyDbg ou SoftIce :).

Cette méthode est toutefois extrêmement fastidieuse, car les piles d'appel aux API natives sont extrêmement longues, et la plupart des débogueurs ne sont pas capables d'interpréter les données de typage (à l'exception notable de PEBrowse Debugger [19]).

Pour les librairies les plus utilisées (comme les librairies système), celles-ci sont précompilées et placées dans le *Global Assembly Cache* (GAC) natif, avec le suffixe « .ni.dll » (ni = *native image*), qui se trouve dans le répertoire :

« %windir%\assembly\NativeImages* ».

Le contenu du GAC peut être consulté avec la commande GACUTIL [20].

Il est également possible de compiler en code natif n'importe quelle *assembly*, en utilisant la commande NGEN [21] (*Native Image Generator*).

On notera que la technique proposée par l'outil « .NET Sploit » [22] pour réaliser un *rootkit* .NET consiste à modifier les *assemblies* placées

19. http://www.smidgeonsoft.prohosting.com/pebrowse-pro-interactive-debugger.html
20. http://msdn.microsoft.com/fr-fr/library/ex0ss12c{%}28VS.80{%}29.aspx
21. http://msdn.microsoft.com/fr-fr/library/6t9t5wcf%28VS.80%29.aspx
22. http://www.applicationsecurity.co.il/english/NETFrameworkRootkits/tabid/161/Default.aspx

dans le GAC, dont la signature est vérifiée à l'installation mais pas à l'exécution.

### 5.2 Méthodes statiques

**Désassemblage** Le *bytecode* .NET contenu dans une *assembly* est entièrement modifiable à l'aide des outils ILDASM [23] et ILASM [24] fournis par Microsoft.

ILDASM permet de générer un listing assembleur (au format IL) correspondant à l'*assembly* fournie en entrée. Ce fichier peut être modifié à loisir dans un éditeur de texte, puis recompilé avec ILASM. Si aucune modification n'a été apportée, le fichier binaire généré en sortie sera strictement identique au fichier binaire fourni en entrée, la compilation en *bytecode* ne perdant aucune information sur la sémantique du programme.

Les seules différences qui peuvent apparaitre sont liées aux données externes embarquées dans l'*assembly*, comme le manifeste, les ressources, etc. ainsi qu'un comportement différent du compilateur utilisé, si les versions et les options de compilation ne sont pas strictement identiques.

Il n'existe pas de méthode pour se protéger contre la modification d'une *assembly*, à l'exception de :

- La signature de code. Il sera alors impossible d'exécuter une *assembly*, même légèrement modifiée, dans le même contexte de sécurité.
- L'obfuscation de code. Il faut noter que l'obfuscateur livré avec Visual Studio (*Dotfuscator Community Edition*) utilise un renommage alphabétique des variables, n'ayant pas d'autre effet que de complexifier la compréhension du programme par un humain. Il existe d'autres obfuscateurs sur le marché, renommant les variables avec des caractères non imprimables, ce qui ne permet plus de manipuler le fichier IL facilement.

La modification d'une *assembly* permet de l'instrumenter à loisir. La contrainte pour l'analyste est qu'il doit arrêter puis redémarrer son application à chaque modification, puisqu'il modifie les fichiers sur disque et non en mémoire.

23. http://msdn.microsoft.com/fr-fr/library/f7dy01k1%28VS.80%29.aspx
24. http://msdn.microsoft.com/fr-fr/library/496e4ekx%28VS.80%29.aspx

**Décompilation** La décompilation de *bytecode* .NET non obfusqué fonctionne extrêmement bien, puisque le *bytecode* embarque toutes les informations sémantiques issues du code source. Il est même possible de décompiler un programme dans un langage différent de la source d'origine ! (Modulo les constructions spécifiques à chaque langage – les développeurs Visual Basic ayant une forte appétence pour le GOTO par exemple).

Le meilleur outil gratuit pour cette tâche est Reflector.NET [25], conçu par un employé Microsoft (on n'est jamais aussi bien servi que par soi-même). Il est extensible par un mécanisme de plug-ins, et bénéficie d'un soutien très actif de la communauté. Une version commerciale, intégrée à Visual Studio et offrant en plus le débogage, est aujourd'hui proposée par la société Red Gate.

Il existe également d'autres outils commerciaux (comme ceux de la société 9Rays [26]). La décompilation n'étant pas une activité commercialement rémunératrice, les éditeurs de décompilateurs ont en général un produit de protection logicielle à vendre. Il est assez amusant de constater que le décompilateur de la société X refuse de décompiler les applications protégées par ses outils, mais fonctionne très bien sur ceux de la société Y :).

**Phoenix Framework** Le Phoenix Framework est un environnement de manipulation du *bytecode* .NET issu de Microsoft Research [27]. Ce projet est toutefois suffisamment abouti pour que Microsoft envisage de l'intégrer dans Visual Studio 2010.

Il est impossible de présenter le Phoenix Framework en quelques lignes. Pour qui souhaite réaliser de l'analyse statique ou de la manipulation d'*assemblies*, il est vivement recommandé de suivre les 3 jours de tutoriels disponibles sur Microsoft Connect [28].

Il est à noter que l'outil d'analyse statique Cthulhu développé par Matt Miller et présenté à ToorCon 2007 repose sur le Phoenix Framework.

---

25. http://www.red-gate.com/products/reflector/
26. http://www.9rays.net/
27. http://research.microsoft.com/en-us/collaboration/focus/cs/phoenix.aspx
28. https://connect.microsoft.com/Phoenix

### 5.3 Méthodes dynamiques

**WinDbg** WinDbg (le débogueur Microsoft [29]) présente de nombreux avantages sur ses concurrents. Pour ce qui est du code .NET, il est capable d'exploiter les informations de typage contenues dans les *assemblies*.

Cette fonctionnalité est implémentée dans une extension fournie avec chaque version du Framework .NET et judicieusement nommée « SOS.DLL ». Il suffit de charger la version correspondant à la version du Framework utilisée par l'application :

```
0:004> .loadby sos.dll mscorwks
```

De nombreuses fonctions de support .NET sont désormais disponibles :

```
0:004> !help
-------------------------------------------------------------------------------

SOS is a debugger extension DLL designed to aid in the
    debugging of
managed programs. Functions are listed by category, then
    roughly in
order of importance. Shortcut names for popular functions are
    listed
in parenthesis.

Type "!help <functionname>" for detailed info on that function.

Object Inspection                  Examining code and stacks
------------------------------
    ------------------------------
DumpObj (do)                       Threads
DumpArray (da)                     CLRStack
DumpStackObjects (dso)             IP2MD
DumpHeap                           U
DumpVC                             DumpStack
GCRoot                             EEStack
ObjSize                            GCInfo
```

Les avantages de WinDbg sont nombreux :
- Débogage symbolique ;
- Visibilité sur le *bytecode* et sur le code natif (par exemple en cas d'intégration de code non managé, du type composant COM ou P/Invoke).
- Possibilité de modifier le comportement de l'application dynamiquement (modification des données ou du code JIT-compilé).

29. http://www.microsoft.com/whdc/DevTools/Debugging/default.mspx

L'interface de WinDbg reste toutefois assez confuse visuellement, et le passage a l'échelle sur de grosses applications reste problématique... comme nous le verrons par la suite dans l'application de cet outil à Microsoft OCS 2007.

Pour la complétude des références, on peut également mentionner l'extension SOSEx.dll qui ajoute quelques commandes fort utiles, et dont la version 2 est disponible ici [30].

**API de débogage** Le Framework .NET fourni par Microsoft expose une API de débogage très riche sous forme d'interfaces COM, préfixées par ICorDebug* [31].

L'implémentation d'un débogueur sur la base de ces interfaces n'est pas une partie de plaisir. Il existe toutefois un débogueur Open Source fourni par Microsoft qui peut servir de base de code : MDbg [32]. Ce débogueur est rustique (interface en ligne de commande), mais il peut s'intégrer avec IronPython – ce qui lui confère des propriétés intéressantes.

**API de profilage** Le Framework .NET expose une API de profilage permettant à une application tierce d'être notifiée de tout évènement interne au Framework (ex. instanciation d'une classe, compilation JIT, etc.). Elle est exposée sous forme d'interfaces COM, préfixées par ICorProfiler* [33].

Cette API peut également être utilisée pour remplacer le *byte-code* à l'intérieur d'une fonction qui n'a pas encore été JIT-compilée, grâce à l'API SetILFunctionBody(). Toutefois l'une des contraintes majeures de cette technique est que le profileur doit être lui-même un composant COM, ce qui allonge et complexifie les développements. Il reste possible de partir d'un code existant, comme celui de l'outil Logger [34].

Il existe également des implémentations *Closed Source*, comme dotTrace [35] ou Foundstone .NETMon [36].

30. http://www.stevestechspot.com/SOSEXV2NowAvailable.aspx
31. http://msdn.microsoft.com/en-us/library/ms230588.aspx
32. http://blogs.msdn.com/jmstall/archive/2005/11/08/mdbg_linkfest.aspx
33. http://msdn.microsoft.com/fr-fr/library/ms233177.aspx
34. http://www.codeproject.com/KB/cs/IL_Rewriting.aspx
35. http://www.jetbrains.com/profiler/
36. http://www.foundstone.com/us/resources/proddesc/netmon.htm

Pour quiconque souhaite se lancer dans la réécriture dynamique de code en utilisant l'API de profilage, un excellent article est disponible ici [37].

**Réflexion et méthodes dynamiques** Le Framework .NET supporte la réflexion à travers les classes disponibles dans l'espace de noms « System.Reflection ».

La réflexion permet d'avoir accès depuis du code .NET à l'ensemble des informations disponibles sur une *assembly* ou une classe : types, variables, méthodes (y compris le *bytecode*), évènements, métadonnées et attributs – le tout y compris sur des types privés.

L'espace de noms « System.Reflection.Emit » offre une possibilité supplémentaire : générer dynamiquement des *assemblies*, des classes ou des méthodes, le tout en utilisant des mnémoniques puisqu'un assembleur IL est gracieusement fourni par la classe « ILGenerator ».

Muni de ces primitives, il est très simple de réécrire un outil semblable à Reflector – sauf la décompilation qui représente le gros du travail. Il convient juste de prendre garde à intercepter correctement l'évènement « AssemblyResolve », car l'API standard va descendre à travers toutes les *assemblies* requises pour l'analyse et potentiellement échouer sur une *assembly* introuvable.

Il reste toutefois une question en suspens : est-il possible de remplacer une classe ou une méthode existante par du code généré à la volée ? La réponse est malheureusement non (à ma connaissance), car l'API n'autorise pas le remplacement en mémoire du code issu d'une *assembly* sur disque. Seules les méthodes générées dynamiquement peuvent être remplacées, grâce à la classe « MethodRental ».

On notera qu'il est possible de sauvegarder sur disque les *assemblies* générées dynamiquement, grâce à la classe « AssemblyBuilder ». Ces *assemblies* ne peuvent alors plus être modifiées dynamiquement, conformément à la limitation évoquée précédemment.

**Use the source, Luke** Des codes source équivalents aux Frameworks 1.0 et 2.0 sont disponibles sous le nom de SSCLI – *Shared Source Common Language Infrastructure* – aussi appelé « projet ROTOR » chez Microsoft. Le code disponible permet de se faire une

37. http://msdn.microsoft.com/en-us/magazine/cc188743.aspx

bonne idée du fonctionnement interne du Framework, tout en étant non supporté, délicat à compiler, et potentiellement différent de la version maintenue en interne par Microsoft.

Le code source du Framework 3.5 et des librairies afférentes est plus largement ouvert [38], mais sous forme de symboles de débogage. Il ne semble pas faisable de reconstruire tout ou partie du Framework 3.5 à partir des fichiers téléchargeables sur le site de Microsoft (ou du moins personne ne s'y est aventuré à ma connaissance).

**Les cas simples** Les techniques déroulées précédemment se veulent génériques. Il ne faut toutefois pas oublier les cas simples, qui ne nécessitent pas une telle artillerie.

Par exemple, pour les méthodes sans effet de bord, il est beaucoup plus rapide d'instancier la classe dans un nouveau projet C#, voir dans IronPython, pour étudier son fonctionnement.

Il est également très simple d'être notifié de tout évènement (*Event*). En effet n'importe quel objet générant des évènements (ex. un bouton, une case à cocher, etc.) offre la possibilité d'ajouter dynamiquement des *Event Handlers* [39].

# 6 Application : Microsoft OCS 2007

## 6.1 Introduction

**Présentation du produit** Le produit Microsoft Office Communications Server (OCS) est un produit de communication unifiée (c'est-à-dire offrant des fonctions de VoIP, messagerie instantanée, réunions virtuelles, etc.).

Il s'agit d'un produit important dans le portfolio Microsoft, car le marché pour les communications unifiées est en pleine expansion, tiré par l'essor de la VoIP.

Une partie du produit OCS est issue du rachat de la société PlaceWare en 2003 (cette société ayant elle-même été fondée en 1996 par des anciens de Xerox). Il s'est d'abord appelé Live Communications Server 2003 puis 2005, avant de prendre son nom actuel.

38. http://referencesource.microsoft.com/netframework.aspx
39. http://msdn.microsoft.com/en-us/library/dfty2w4e.aspx

Le produit initial offrait des fonctions de conférence en ligne. Il était entièrement écrit en Java et n'était pas destiné à être déployé chez les utilisateurs. On peut donc imaginer que l'intégration ne s'est pas faite sans peine, et que des failles de sécurité ont pu perdurer (d'autant que la sécurité des applications était un domaine balbutiant en 1996).

**Distinction importante** Il existe deux versions du produit OCS 2007 : la version initiale (sortie en 2007) et la version dite « R2 » (sortie en 2009).

Bien que ces deux produits semblent très similaires (ils sont d'ailleurs fonctionnellement assez proches), sous le capot il s'avère que de nombreuses parties du code ont été réécrites. En conséquence il sera nécessaire dans la suite du document de préciser si les techniques utilisées s'appliquent à la version « R1 » ou « R2 » (ou les deux).

**Pourquoi OCS ?** Le produit OCS fait partie de ces monstres rarement audités (comme SAP, SharePoint et tant d'autres), car les chercheurs en sécurité sont astreints à des cycles de publication rapides pour continuer à exister dans le *Security Circus*.

Il n'existe aucune étude publique sur la sécurité de ce produit, et très peu de failles ont été publiées par des tiers.

Etant une application massivement « .NET », OCS se prête toutefois bien à la mise en œuvre de l'ensemble des techniques décrites préalablement.

### 6.2 Installation du produit

Une version d'évaluation du produit est mise à disposition gracieusement sur le site de Microsoft [40]. On peut toutefois signaler que l'installation d'OCS « R1 » n'est pas une partie de plaisir, de nombreux composants (ex. DNS, SQL Server, etc.) devant être installés et configurés manuellement. Les choses se sont améliorées avec OCS « R2 », dont l'installation « Standard » peut être effectuée en quelques clics sur un seul serveur physique.

40. http://technet.microsoft.com/en-us/evalcenter/bb684921.aspx

Après installation, nous sommes en face d'un monstre : 180 Mo de binaires dans le répertoire d'installation, et 40 Mo dans le répertoire partagé « Common Files » (pour la version « R2 »).

Puisqu'il faut bien commencer quelque part, nous nous focaliserons dans la suite sur la fonction de *Web Conferencing* (implémentée dans le répertoire éponyme). En effet, cette fonction a la propriété intéressante de pouvoir accepter des invités anonymes en provenance d'Internet. Sa sécurité est donc essentielle pour celle du produit.

### 6.3 Instrumentation statique

Il s'avère que l'application est capable de générer une quantité impressionnante de traces applicatives – quasiment toutes les méthodes peuvent être tracées.

Les outils OCSLogger [41] / OCSTracer [42] peuvent être utilisés pour gérer ces traces applicatives.

Dans ces conditions, une instrumentation statique additionnelle de l'application n'est pas utile.

On notera que le débogueur WinDbg supporte la journalisation ETW via l'extension **!wmitrace**. Mais il reste plus convivial d'utiliser l'outil **OCSLogger.exe** que d'extraire les GUID des traces « à la main »...

### 6.4 Décompilation

L'*assembly* principale et toutes ses dépendances se chargent correctement dans l'outil Reflector. Aucune obfuscation ne semble appliquée sur le code. A ce stade on peut donc espérer régénérer un code C# compilable.

La décompilation complète de l'application offre des avantages considérables pour l'analyste, car il peut ensuite utiliser toutes les fonctions disponibles dans l'environnement de développement Visual Studio (références croisées, exécution pas-à-pas, inspection des variables à l'exécution, etc.) pour appréhender plus rapidement le fonctionnement du logiciel.

41. http://technet.microsoft.com/en-us/library/bb894487.aspx
42. http://msdn.microsoft.com/en-us/library/bb857283.aspx

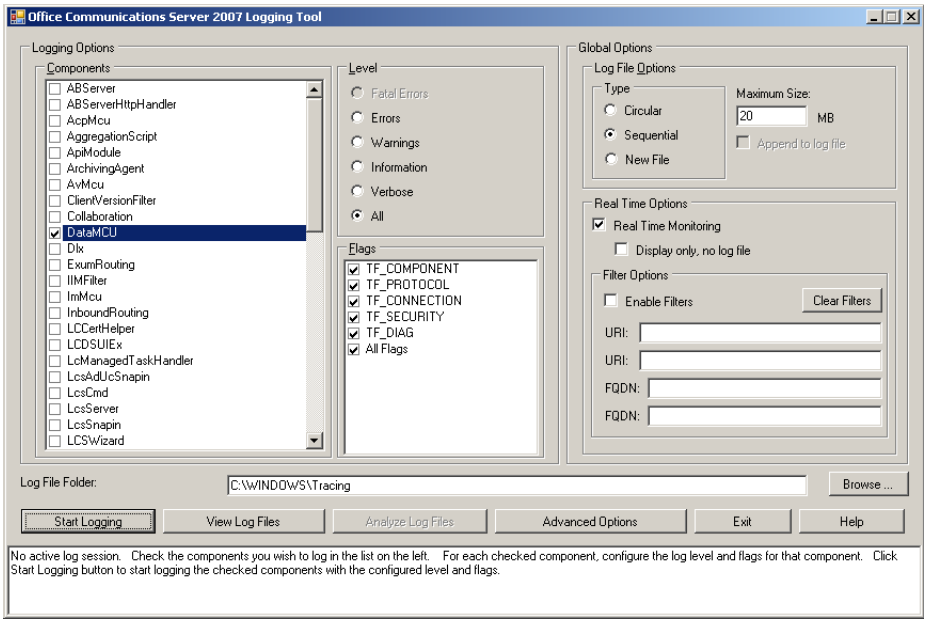


**FIGURE 1.** L'outil OCSLogger.exe

Toutefois il existe quelques erreurs de syntaxe dans le code produit (imputables à l'outil et faciles à corriger), ainsi que plusieurs points durs (dans la version R1), décrits ci-après.

**Fonctions de trace** Le code de génération des traces semble issu d'un outil automatique. Le nom WPP (utilisé en interne) laisse à penser qu'un préprocesseur similaire à celui disponible pour les pilotes en mode noyau [43] a été appliqué sur le code. Il sera toutefois plus simple par la suite de supprimer purement et simplement ces traces, plutôt que de réimplémenter le préprocesseur (a priori non disponible publiquement à la date de rédaction de ce document).

Les traces sont générées au format « .ETL », mais peuvent être converties en texte. Les fichiers d'évènements (indispensables à l'exploitation des fichiers « .ETL ») ne sont pas fournis au format texte standard « .TMF », mais dans un format binaire « .TMX » apparemment spécifié pour l'occasion, ce qui conforte l'hypothèse précédente de code spécifique.

43. http://msdn.microsoft.com/en-us/library/ms793164.aspx

Enfin on notera également que l'implémentation en code managé des fonctions de trace est marquée comme *unsafe*. On peut toutefois supposer que ce code généré automatiquement a été correctement revu et ne va pas induire de failles dans l'application...

**Assemblies J#** J#[44] est un langage très proche de Java (Microsoft n'ayant toutefois pas le droit d'implémenter la spécification Java « officielle » depuis la perte de son procès avec Sun), compilé en *bytecode* .NET.

J# a été conçu en 2002 par Microsoft comme une technologie de transition, devant permettre aux développeurs Java de migrer « en douceur » vers .NET. Le développement a été entièrement réalisé en Inde (Hyderabad). J# n'est pas promis à un grand avenir car il n'est plus supporté à partir de Visual Studio 2008.

Toutefois il s'est avéré que J# est une technologie clé pour Microsoft OCS, puisqu'une partie du code Java de l'application PlaceWare d'origine n'a pas encore été migré. C'est probablement l'une des raisons qui ont motivé Microsoft à publier une version « 64 bits » des librairies de support J# en 2007.

Il est facile d'identifier les *assemblies* écrites en J# dans l'application OCS 2007 R1, puisque ces *assemblies* sont liées aux librairies « vjscor.dll » ou « vjslib.dll ». Ce sont (pour la fonction *Web Conferencing*) :

```
Microsoft.RTC.Server.DataMCU.Application.dll
Microsoft.RTC.Server.DataMCU.Application.Shared.dll
Microsoft.RTC.Server.DataMCU.AppSharing.dll
```

Du fait de l'aspect confidentiel de la technologie J#, il n'existe pas à proprement parler de bon décompilateur pour ce langage (y compris dans les outils commerciaux que j'ai pu tester – c'est-à-dire pour lesquels une version d'évaluation est disponible). Il est peu probable qu'un tel outil apparaisse à l'avenir.

Quant à la traduction du *bytecode* en C#, elle ne produit pas un résultat exploitable (i.e. recompilable) à cause de toutes les astuces que Microsoft a dû déployer pour faire entrer du code Java sur la plate-forme .NET. Les différences conceptuelles entre les langages

44. http://en.wikipedia.org/wiki/J_sharp

J#/Java et C# sont en effet considérables [45]. Rien que la classe de base dont dérivent toutes les autres (*object*) est différente.

On notera qu'une partie des classes J# appartient à l'espace de nommage « com.netopia.* ». Ceci laisse à penser que le protocole de partage d'écran de la fonction *Web Conferencing* est celui du produit Timbuktu (anciennement Netopia, désormais Motorola).

**Getters/setters** Le code OCS 2007 R2 semble utiliser massivement la construction simplifiée « { get ; set ; } » pour exposer les propriétés des classes.

Le langage C# évolue rapidement, et cette construction n'est pas encore supportée par les décompilateurs existants (à la date de rédaction de cet article). On peut toutefois supposer que le problème sera rapidement résolu. De plus les séquences de code correspondantes sont facilement identifiables et factorisables.

**Signature de code** Il s'avère que toutes les *assemblies* produites par Microsoft ont été signées avec une clé appartenant à Microsoft. La signature obtenue permet d'identifier chaque version de chaque *assembly* de manière unique (c'est le mécanisme du *Strong Name* [46]).

Lors de l'édition de liens, le *Strong Name* des dépendances est intégrée aux *assemblies* via le Manifeste de l'application. En cas de modification/recompilation d'une *assembly*, il est donc nécessaire d'éditer le Manifeste de toutes les *assemblies* qui en dépendent.

Le Manifeste peut être manipulé à l'aide de l'outil MT.EXE, fourni dans Visual Studio. Mais cette tâche est relativement fastidieuse vu la quantité d'*assemblies* impliquées...

Une autre solution consiste à utiliser la commande du SDK .NET « SN.EXE ». L'option « -Vr » permet de désactiver la vérification du *Strong Name* pour une *assembly* donnée.

Une solution plus radicale consiste à supprimer toute forme de signature sur tous les exécutables. Un utilitaire tel que SNSRemover [47] permet d'automatiser l'opération.

À noter que ces solutions ne répondent pas au problème du développeur spécifiant explicitement une vérification de signature

45. http://en.wikipedia.org/wiki/Comparison_of_Java_and_C_Sharp
46. http://en.wikipedia.org/wiki/Strong_key
47. http://www.ntcore.com/download.php

au chargement d'une classe, comme c'est le cas par exemple dans l'*assembly* **DataMcuSvc**, méthode **Microsoft.Rtc.Server.DataMCU.StartServer()** :

```
string appClassName = string.Format("placeware.apps.aud.
    AuditoriumApplication,
   Microsoft.Rtc.Server.DataMCU.Application, Version={0},
   Culture=neutral, PublicKeyToken=31bf3856ad364e35", str5);
```

**Résultat obtenu** A titre d'exemple, voici le code « brut » obtenu après décompilation du point d'entrée `Microsoft.Rtc.Server.DataMCU.LMProgram::Main` :

```
private static void Main(string[] args)
{
    if ((args.Length == 1) && (args[0] == "-noservice")) {
        try {
            bool flag = AllocConsole();
            Console.Write("Data MCU is initializing... ");
            ServiceWorker worker = new ServiceWorker();
            worker.StartServer(args);
            Console.WriteLine("done.\nEnter q to exit.");
            while (!Console.ReadLine().Equals("q") { }
            Console.Write("Stopping....");
            worker.StopServer();
            Console.WriteLine("Stopped.");
            Console.WriteLine("Hit enter to close this window
                and exit the process.");
            Console.ReadLine();
            if (flag)
                FreeConsole();
            Environment.Exit(0);
        } catch (Exception exception) {
            Console.WriteLine("Exception terminated DataMCU.\
                nType={0}\nMessage={1}\nStack={2}", exception.
                GetType().FullName, exception.Message,
                exception.StackTrace);
            Environment.Exit(Marshal.GetHRForException(
                exception));
        }
    } else {
        try {
            ServiceBase.Run(new LMService());
            if ((Trace.traceProvider.Level >= 5) && ((Trace.
                traceProvider.Flags & 1) != 0))
                WPP_df782f688133deb7f16baab168b61264.WPP_NOARGS
                    (10);
            Environment.Exit(0);
        } catch (Exception exception2)
        {
            if ((Trace.traceProvider.Level >= 2) && ((Trace.
                traceProvider.Flags & 1) != 0))
```

```
                    WPP_df782f688133deb7f16baab168b61264.WPP_sss
                        (11, TraceProvider.MakeStringArg(exception2
                        .GetType().FullName), TraceProvider.
                        MakeStringArg(exception2.Message),
                        TraceProvider.MakeStringArg(exception2.
                        StackTrace));
                Environment.Exit(Marshal.GetHRForException(
                    exception2));
            }
        }
    }
```

On identifie rapidement le code de génération des traces (à supprimer avant recompilation), ainsi qu'un argument de ligne de commande possible : « `-noconsole` ».

### 6.5 Analyse dynamique

Nous prendrons l'exemple de l'ouverture du port TCP/8057 par le composant **DataMCUSvc.exe**. L'objectif est de retrouver le code responsable de cette opération en utilisant des techniques d'analyse dynamique.

**Code non managé** On peut raisonnablement supposer que l'ouverture du port en écoute va utiliser l'API native **ws2_32!bind()**. Il suffit donc de positionner un point d'arrêt logiciel à l'aide du débogueur WinDbg, en utilisant la commande suivante :

**bp ws2_32!bind**

Il s'avère que de nombreux appels à **bind()** sont effectués au lancement de l'application (dans l'exemple ci-dessous, un appel RPC). Il serait plus judicieux de positionner un point d'arrêt conditionnel sur le port 8057. Mais dans tous les cas, la pile d'appel est conséquente...

```
Breakpoint 0 hit
WS2_32!bind:
71c06e49 8bff            mov     edi,edi
0:000> kv
ChildEBP RetAddr  Args to Child
0012e5cc 77c8a528 000003dc 0012e6ac 00000010 WS2_32!bind (FPO: [Non-Fpo])
0012e6cc 77c8a725 03ee75d4 0012e74c 00000005 RPCRT4!WS_Open+0x27c (FPO: [Non-Fpo])
0012e800 77c8a7eb 03ee75d4 03ee5f00 00000087 RPCRT4!TCPOrHTTP_Open+0x1fc (FPO: [Non-Fpo])
0012e838 77c5899c 03ee75d4 03ee5ec8 03ee5f00 RPCRT4!TCP_Open+0x5c (FPO: [Non-Fpo])
0012e880 77c5b2cc 00000000 03ee5ec8 03ee5f00 RPCRT4!OSF_CCONNECTION::TransOpen+0x5e (FPO: [Non-Fpo])
0012e8e4 77c5b1b8 03ee5f48 000927c0 00000000 RPCRT4!OSF_CCONNECTION::OpenConnectionAndBind+0xbe (FPO:
    [Non-Fpo])
0012e928 77c5b3f5 00000000 0012e9d8 03ee5f80 RPCRT4!OSF_CCALL::BindToServer+0xe3 (FPO: [Non-Fpo])
0012e940 77c6245d 0012ea40 00000000 00000000 RPCRT4!OSF_BINDING_HANDLE::InitCCallWithAssociation+0x5c
    (FPO: [Non-Fpo])
0012e9b8 77c624a0 0012e9d8 0012ea40 0012e9dc RPCRT4!OSF_BINDING_HANDLE::AllocateCCall+0x497 (FPO: [Non
    -Fpo])
0012e9e8 77c71122 00000000 0012ea6c 00000001 RPCRT4!OSF_BINDING_HANDLE::NegotiateTransferSyntax+0x28 (
    FPO: [Non-Fpo])
0012ea00 77c707f5 0012ea40 00000000 0012ea20 RPCRT4!I_RpcGetBufferWithObject+0x5b (FPO: [Non-Fpo])
0012ea10 77c72b64 0012ea40 0012ee28 0012ee0c RPCRT4!I_RpcGetBuffer+0xf (FPO: [Non-Fpo])
0012ea20 77ce2125 0012ea6c 000000db 03ee5f48 RPCRT4!NdrGetBuffer+0x2e (FPO: [Non-Fpo])
0012ee0c 77c80968 77c593e0 77c84e06 0012ee28 RPCRT4!NdrClientCall2+0x197 (FPO: [Non-Fpo])
0012ee20 77c80943 03ee5f48 03edfbf0 03ee6070 RPCRT4!ept_map+0x1b (FPO: [Non-Fpo])
0012eedc 77c854fc 03edfbf0 766f214c 766f2160 RPCRT4!EpResolveEndpoint+0x247 (FPO: [Non-Fpo])
0012ef18 77c893b2 766f2148 03edfbf0 03edfc10 RPCRT4!DCE_BINDING::ResolveEndpointWithEpMapper+0x46 (FPO
    : [Non-Fpo])
0012ef4c 77c88cfa 766f2148 000927c0 00000001 RPCRT4!OSF_BINDING_HANDLE::ResolveBindingWorker+0x50 (FPO
    : [Non-Fpo])
0012ef68 77c7f435 766f2148 00000000 0012efb8 RPCRT4!OSF_BINDING_HANDLE::ResolveBinding+0x5c (FPO: [Non
    -Fpo])
0012ef78 766f5114 03edfbe0 766f2148 00000000 RPCRT4!RpcEpResolveBinding+0x3c (FPO: [Non-Fpo])
[...]
```

**Listing 1.1.** Pile d'appel partielle lors du premier appel à bind() (vue du code non managé)

```
0:000> !CLRStack
OS Thread Id: 0x244 (0)
ESP       EIP
0012f25c 71c06e49 [NDirectMethodFrameSlim: 0012f25c] Microsoft.Rtc.Internal.Wmi.WmiConsumer.
    GetComputerObjectName(Int32, System.Text.StringBuilder, UInt64 ByRef)
0012f270 011670dc Microsoft.Rtc.Internal.Wmi.WmiConsumer.get_MachineDn()
0012f288 01166e35 Microsoft.Rtc.Internal.Wmi.WmiConsumer.get_Msft_SipMcuSetting()
0012f2cc 01166bb4 Microsoft.Rtc.Internal.Wmi.WmiConsumer.get_Msft_SipMcuFactorySetting()
0012f300 0116690c Microsoft.Rtc.Internal.Wmi.WmiConsumer.get_PoolDn()
0012f320 01166702 Microsoft.Rtc.Internal.Wmi.WmiConsumer.get_PoolInstance()
0012f354 01166568 Microsoft.Rtc.Internal.Wmi.WmiConsumer.get_Backend()
0012f38c 01163f47 Microsoft.Rtc.Internal.Wmi.WmiConsumer.GetInitialSettings(Microsoft.Rtc.Internal.Wmi
    .WmiConsumerClassEntry)
0012f3c8 01163968 Microsoft.Rtc.Internal.Wmi.WmiConsumer.Start()
0012f3f4 0116122c Microsoft.Rtc.Server.DataMCU.Configuration.ServerConfiguration.StartWmiConsumer()
0012f404 011610a6 Microsoft.Rtc.Server.DataMCU.Configuration.ServerConfiguration.Initialize()
0012f408 01160556 Microsoft.Rtc.Server.DataMCU.ServiceWorker.StartServer(System.String[])
0012f454 01160264 Microsoft.Rtc.Server.DataMCU.LMProgram.Main(System.String[])
0012f69c 79e88f63 [GCFrame: 0012f69c]
```

**Listing 1.2.** Pile d'appel lors du premier appel à bind() (vue du code managé)

La commande **kv** donne la pile d'appel non managée du thread courant. La commande **!CLRStack** donne la pile d'appel managée du thread courant. La commande **!DumpStack** permet d'obtenir la pile d'appel complète, incluant le code managé et le code non managé. Cette pile n'est pas reproduite ici car elle occupe plusieurs pages...

Il est également possible de spécifier un thread quelconque à ces deux commandes, à l'aide de la syntaxe suivante :

```
0:024> !threads
ThreadCount: 12
UnstartedThread: 0
```

```
BackgroundThread: 7
PendingThread: 0
DeadThread: 0
Hosted Runtime: no
                                    PreEmptive   GC Alloc           Lock
       ID OSID ThreadOBJ    State     GC       Context        Domain   Count APT Exception
   0    1  e24 001818b0      a020 Enabled  00000000:00000000 0014c9e0     1 MTA
   2    2  b00 0018b780      b220 Enabled  00000000:00000000 0014c9e0     0 MTA (Finalizer)
   7    3  93c 001fd4c0   200b020 Enabled  00000000:00000000 0014c9e0     0 MTA
  11    4  884 03f2bb90    80a220 Enabled  00000000:00000000 0014c9e0     0 MTA (Threadpool Completion
       Port)
  12    5  110 03f0e9a0   880b220 Enabled  00000000:00000000 0014c9e0     0 MTA (Threadpool Completion
       Port)
  14    6  ed4 03f1d808   200b220 Enabled  00000000:00000000 0014c9e0     0 MTA
  21    7  df8 03f43718   200b020 Enabled  00000000:00000000 0014c9e0     0 MTA
  15    8  4e0 03f80c78   200b020 Enabled  00000000:00000000 0014c9e0     0 MTA
  16    9  914 03f81840      7020 Enabled  00000000:00000000 0014c9e0     0 STA
  19    c  cd4 00231d30   180b220 Enabled  00000000:00000000 0014c9e0     0 MTA (Threadpool Worker)
  20    b  14c 03eccac0    800220 Enabled  00000000:00000000 0014c9e0     0 Ukn (Threadpool Completion
       Port)
  22    a  380 03f66b10   200b220 Enabled  00000000:00000000 0014c9e0     1 MTA
```

**Listing 1.3.** Lister les threads managés

```
0:024> ~16e !clrstack
OS Thread Id: 0x914 (16)
ESP       EIP
05b9f618 7c82ed54 [NDirectMethodFrameStandalone: 05b9f618] com.ms.vjsharp.windowing.win32.
    UnsafeWin32Calls.intGetMessage<PInvokeHelper>vjsnativ(MSGHelper, Int32, Int32, Int32)
05b9f630 6cdc0428 com.ms.vjsharp.windowing.win32.UnsafeWin32Calls.intGetMessage(com.ms.vjsharp.win32.
    MSG, Int32, Int32, Int32)
05b9f64c 6ceca03e com.ms.vjsharp.windowing.win32.Win32Toolkit.run()
05b9f678 6ce3a1d0 java.lang.Thread.run()
05b9f6a4 03c132de [MulticastFrame: 05b9f6a4] System.Threading.ThreadStart.Invoke()
05b9f6b4 793d7a7b System.Threading.ThreadHelper.ThreadStart_Context(System.Object)
```

```
05b9f6bc 793683dd System.Threading.ExecutionContext.Run(System.Threading.ExecutionContext, System.
    Threading.ContextCallback, System.Object)
05b9f6d4 793d7b5c System.Threading.ThreadHelper.ThreadStart()
05b9f8f8 79e88f63 [GCFrame: 05b9f8f8]
```

**Listing 1.4.** Pile d'appel du thread managé n ° 16

**Code managé** Dans cet exemple, on fait l'hypothèse raisonnable que l'API **System.Net.Sockets.Bind()** va être utilisée par le code managé. Nous allons donc placer un pont d'arrêt directement dans le code managé.

Tout d'abord il convient de s'assurer que le Framework .NET est bien chargé en mémoire. Avec le débogueur WinDbg, il suffit de demander à interrompre l'exécution lors du chargement de la librairie « mscorwks.dll ».

```
sxe ld mscorwks.dll
```

Il est alors possible de charger l'extension de débogage adaptée à la version du Framework .NET utilisée :

```
.loadby sos mscorwks
```

Grâce à cette extension, il est possible de mettre des points d'arrêt sur le code managé : `!bpmd DataMCUSvc.exe Microsoft.Rtc.Server.DataMCU.LMProgram.Main`

Il faut mentionner que la commande **!Name2EE** permet d'identifier le fichier contenant une méthode donnée, afin de positionner les points d'arrêt adéquats.

```
0:000> !Name2EE *!Microsoft.Rtc.Internal.Wmi.WmiConsumer.
    GetComputerObjectName
Module: 790c2000 (mscorlib.dll)
-------------------------------------
Module: 009223b4 (sortkey.nlp)
-------------------------------------
Module: 00922044 (sorttbls.nlp)
-------------------------------------
Module: 00902c14 (DataMCUSvc.exe)
-------------------------------------
Module: 67a30000 (System.ServiceProcess.dll)
-------------------------------------
Module: 7a714000 (System.dll)
-------------------------------------
Module: 00903f2c (Microsoft.Rtc.Server.DataMCUTools.dll)
-------------------------------------
Module: 00905218 (Microsoft.Rtc.Server.DataMCU.Hosting.Runtime.
    dll)
-------------------------------------
Module: 00907ebc (Microsoft.Rtc.Server.McuInfrastructure.dll)
Token: 0x06000166
MethodDesc: <not loaded yet>
Name: Microsoft.Rtc.Internal.Wmi.WmiConsumer.
    GetComputerObjectName
Not JITTED yet.
-------------------------------------
Module: 01212390 (LcWmiConsumer.Managed.dll)
Token: 0x06000039
MethodDesc: 01212c68
Name: Microsoft.Rtc.Internal.Wmi.WmiConsumer.
    GetComputerObjectName(Int32, System.Text.StringBuilder,
    UInt64 ByRef)
Not JITTED yet. Use !bpmd -md 01212c68 to break on run.
-------------------------------------
Module: 67580000 (System.Management.dll)
```

**Listing 1.5.** Utilisation de la commande Name2EE

**Résultat obtenu** La pile d'appel à **ws2_32!bind()** lors de l'ouverture du port TCP/8057 est la suivante.

```
Breakpoint 1 hit
WS2_32!bind:
71c06e49 8bff            mov     edi,edi

0:000> !clrstack
OS Thread Id: 0xbe0 (0)
ESP       EIP
0012f2cc 71c06e49 [ComPlusMethodFrameGeneric: 0012f2cc]
    Microsoft.Rtc.Server.DataMCU.Transport.Interop.
    TransportFactory.ListenOn(System.String)
0012f2dc 040df559 Microsoft.Rtc.Server.DataMCU.Messaging.
    MessageConnectionAcceptor..ctor(System.String)
```

```
0012f2ec 040dee59 Microsoft.Rtc.Server.DataMCU.Hosting.
    ApplicationServices.LdmApplication.Initialize()
0012f324 040dd14f placeware.apps.aud.AuditoriumApplication.
    Initialize()
0012f330 040dd0e2 Microsoft.Rtc.Server.DataMCU.Hosting.
    ApplicationServices.Application.InitializeInternal(System.
    Collections.Generic.IDictionary`2<System.String,System.
    String>)
0012f33c 040dcca3 Microsoft.Rtc.Server.DataMCU.Hosting.
    ApplicationServices.Application.InitializeApplication(
    System.Type, System.Collections.Generic.IDictionary`2<
    System.String,System.String>)
0012f348 0116d243 Microsoft.Rtc.Server.DataMCU.Hosting.Runtime.
    ApplicationController..ctor(System.Collections.Generic.
    IDictionary`2<System.String,System.String>, System.String,
    System.String, Byte[], Byte[], System.String, System.String
    , Microsoft.Rtc.Server.DataMCU.Hosting.Runtime.
    IServiceWorker)
0012f408 011607c0 Microsoft.Rtc.Server.DataMCU.ServiceWorker.
    StartServer(System.String[])
0012f454 01160264 Microsoft.Rtc.Server.DataMCU.LMProgram.Main(
    System.String[])
0012f69c 79e88f63 [GCFrame: 0012f69c]

0:000> dw poi(esp+8)
03edf688  0002 791f 0000 0000 0000 0000 0000 0000
```

Le deuxième paramètre pointe sur une structure **sockaddr_in**. Le deuxième entier 16 bits de cette structure, lu en « *network order* », correspond bien au port $0x1f79 = 8057$. Ce port est donc ouvert par le constructeur de la classe `Microsoft.Rtc.Server.DataMCU.Messaging.MessageConnectionAcceptor`

## 6.6 Test unitaire

Comme toute *assembly* .NET, chaque composant du produit OCS 2007 peut être utilisé de manière autonome, soit dans un nouveau projet Visual Studio, soit directement en ligne de commande à l'aide d'outils comme IronPython [48].

Cette méthode permet de vérifier de manière unitaire les fonctionnalités d'une classe, comme dans l'exemple ci-dessous (les commandes sont préfixées par « >>> »).

```
>>> import clr
>>> clr.AddReference("Microsoft.RTC.Server.DataMCU.Application.
    Shared.dll")
>>> import placeware.io.PWPath
```

48. http://www.codeplex.com/Wiki/View.aspx?ProjectName=IronPython

```
>>> from System import Array
>>> a = Array[str]("")
>>> placeware.io.PWPath.main(a)
Testing ...
checkPWPathSyntax() - true
convertToUnixSyntax() - /awm/vol/vol-01/eng/work/foo/bar.html
convertToWindowsSyntax() - \\awm\vol\vol-01\eng\work\foo\bar.
    html
getPWPath() - awm:/vol/vol-01/eng/work/foo/bar.html
getUnixPath() - /awm/vol/vol-01/eng/work/foo/bar.html
getWindowsPath() - \\awm\vol\vol-01\eng\work\foo\bar.html
>>> b = Array[str]([\og{}c:\\windows\\notepad.exe\fg{}])
>>> placeware.io.PWPath.main(b)
Testing ...
checkPWPathSyntax() - false
convertToUnixSyntax() - null
convertToWindowsSyntax() - null
getPWPath() - c:\windows\notepad.exe
getUnixPath() - null
getWindowsPath() - null
```

**Listing 1.6.** Test de la classe placeware.io.PWPath avec IronPython

# 7 Exemple de résultats

Il est impossible de reproduire ici les résultats complets de l'audit applicatif mené sur OCS 2007 R1 et R2, d'autant que ces audits ont été réalisés dans un cadre commercial.

Voici toutefois deux exemples qui démontrent que les méthodes présentées sont suffisantes pour obtenir des réponses complètes sur tous les points clés affectant la sécurité du produit.

## 7.1 Protocole PlaceWare

**Question posée** Lors de la connexion à un meeting, les premiers échanges réseau (incluant l'authentification utilisateur) s'effectuent en protocole SIP.

Toutefois après avoir récupéré plusieurs paramètres de configuration dans la réponse SIP, le client Live Meeting se reconnecte au port TCP/8057 du serveur, sur lequel une communication dans un protocole propriétaire non documenté est établie.

La question essentielle qui se pose alors est : « comment l'authentification utilisateur est-elle propagée entre ces deux connexions ? »

**Aperçu du protocole** Voici les premiers échanges applicatifs entre client et serveur sur le port TCP/8057.

```
C>SV3.1(20)  application_data
-----------------------------------------------------------------
pw2

C>SV3.1(2580)  application_data
-----------------------------------------------------------------
00 00 00 00 00 00 00 20 37 30 30 30 30 30 30 30    ....... 70000000
30 30 30 30 30 30 30 38 36 44 44 35 38 34 41    0000000086DD584A
46 32 31 46 30 32 37 41 04 00 00 00 00 16 00 00    F21F027A ........
00 0b 00 01 87 00 f1 86 d1 ce 71 ef a6 16 00 00    .........q ......
00 2e 00 02 00 1e 61 7c 49 1b 28 32 1c 16 fa f2    ......a|I .(2....
[...]
```

Le client commence par envoyer la chaine « pw2\0 », sans doute pour indiquer qu'il parle le protocole « PlaceWare 2 ». Il envoie ensuite une longue séquence binaire, incluant une chaine de caractères hexadécimale. Il s'avère que cette chaine a été récupérée à la fin de l'échange SIP, dans un paramètre dénommé « sAuthID ».

Il va falloir désormais comprendre comment est généré ce paramètre « sAuthID », en utilisant les techniques précédentes.

**Recherche du point d'entrée** Lors de la connexion d'un client sur le port TCP/8057, la méthode suivante est appelée :

```
Microsoft.Rtc.Server.DataMCU.Messaging.
    MessageConnectionAcceptor.HandleTransportConnection
```

**Listing 1.7.** Instanciation de la classe TransportFactory

En effet cette méthode est définie en tant que fonction de « callback » au niveau de la classe **TransportFactory** :

```
public MessageConnectionAcceptor(string listenerUrl)
{
    this.m_trustedServers = new List<string>();
    this.listener = ((TransportFactory) new
        TransportFactoryClass()).ListenOn(listenerUrl);
    this.localUrl = this.listener.Url;
    this.acceptCallback = new AcceptCallback(this);
}
```

```
    public void Callback(ITransportAsyncResult ar)
    {
        this.acceptor.HandleTransportConnection(ar);
    }
```

Pour s'en assurer, il suffit de mettre un point d'arrêt sur cette méthode :

```
0:006> !Name2EE Microsoft.RTC.Server.DataMCU.Messaging.dll!Microsoft.Rtc.Server.DataMCU.Messaging.
    MessageConnectionAcceptor.HandleTransportConnection
Module: 03c86a34 (Microsoft.Rtc.Server.DataMCU.Messaging.dll)
Token: 0x06000185
MethodDesc: 03c8a810
Name: Microsoft.Rtc.Server.DataMCU.Messaging.MessageConnectionAcceptor.HandleTransportConnection(
    Microsoft.Rtc.Server.DataMCU.Transport.Interop.ITransportAsyncResult)
Not JITTED yet. Use !bpmd -md 03c8a810 to break on run.
0:006> !bpmd -md 03c8a810
MethodDesc = 03c8a810
Adding pending breakpoints...
```

**Listing 1.8.** Mise en place d'un point d'arrêts sur le début du traitement d'une nouvelle connexion

De fil en aiguille, on remonte la pile d'appel suivante :

– `Microsoft.Rtc.Server.DataMCU.Messaging.MessageConnectionAcceptor.HandleNewConnection`

– `Microsoft.Rtc.Server.DataMCU.Messaging.MessageConnectionAcceptor+ConnectionVerificationContext`

– `Microsoft.Rtc.Server.DataMCU.Messaging.RecordConnection`

Cette dernière classe est extrêmement importante dans le traitement des messages. L'essentiel de notre analyse se concentrera dessus.

**Classe RecordConnection** Cette classe contient les types et méthodes suivantes :

- Initialisation de la signature « pw2 »

```
static RecordConnection()
{
    signature = new byte[] { 0x70, 0x77, 50, 0 };
    defaultReadBufferSize = 0x200;
}
```

- Définition des messages du protocole PlaceWare

```
private enum FrameCode : byte
{
    Authentication = 0x55,
    BreakChannel = 6,
    CloseChannel = 0,
    DataRecord = 0x16,
    NoCode = 0xff,
    OpenChannel = 0x37,
    SetChannel = 4,
    Signature = 0x56
}
```

  A la lecture de la méthode **ReadFrames()** on se rend compte que le message « 5 » est également supporté et correspond à la méthode **Abort()**.

- Boucle de traitement des messages La boucle de traitement des messages proprement dite est la méthode **ReadFrames()**. La logique de cette machine à états est la suivante.

  1. Etat **FrameCode.Signature** La méthode **ReadSignature()** est en charge de vérifier les 4 octets de signature vus précédemment.
  2. Etat **FrameCode.Authentication** La méthode **ReadAuthenticationKey()** consomme 4 octets obligatoirement nuls. Il s'agit probablement d'un reliquat de la version précédente du protocole.

     La méthode **ReadRecordLengthAndBody()** consomme ensuite 4 octets représentant la taille des données à venir (octets de poids fort en premier). Cette taille doit être inférieure ou égale à la constante **maxLength**, soit 0x8000. Les données restantes sont copiées dans la variable **body** puis copiées à nouveau dans la variable **destinationArray**.

Il existe une possibilité d'erreur de manipulation d'entiers à cette étape (classe d'erreur appelée « *integer overflow* « / » *integer underflow* »). Il est donc nécessaire de vérifier que tous les types manipulés sont non signés (en l'occurrence de type **UInt32**) et qu'ils ne font pas l'objet d'arithmétique hasardeuse.

```
private bool ReadRecordLengthAndBody(uint maxLength,
    out BufferView body)
{
    uint CS$1$0000;
    bool chArray = false;
    body = null;
    this.VerifyIsIoThread();
    if (this.ReadUInt32(out CS$1$0000)) {
        ...
        if (CS$1$0000 > maxLength) {
        ...
```

La variable **body** est passée à la méthode **InvokeKeyHandlerCallback()**, qui appelle dans l'ordre **HandleKeyReceived()** puis **OnVerifyKey()** puis **KeyVerifier()**.
Cette dernière méthode est en charge de la vérification effective de la clé (« sAuthId »), elle est implémentée dans la classe `Microsoft.Rtc.Server.DataMCU.Hosting.ApplicationServices.LdmApplication`

**Vérification de la clé (sAuthId)** Comme vu précédemment, la connexion d'un client Live Meeting s'effectue en deux étapes.

La première étape est une connexion SIP permettant d'authentifier l'utilisateur, et de récupérer les paramètres du meeting au format XML.

La deuxième étape est une connexion selon un protocole binaire propriétaire, appelé PlaceWare. Le seul élément d'authentification de cette connexion semble être un jeton transmis dans le fichier XML sous le nom de « sAuthId ».

La manipulation de ce jeton est donc un élément clé de la sécurité du protocole PlaceWare. Or ce jeton correspond à la clé passée à la méthode **KeyVerifier()**.

**KeyVerifier()** fait simplement appel à la méthode **Redeem()** de la classe **TicketManager**. Les opérations effectuées par cette méthode sont les suivantes :

- Vérification de taille : le ticket doit avoir une taille de 32 octets exactement.
- Le ticket doit être composé de caractères hexadécimaux uniquement, qui sont ensuite décodés par la classe **HexEncoder**.
- Les 16 octets binaires obtenus après décodage sont passés à la méthode **RedeemInternal()**.
- Le ticket est composé de deux parties indépendantes : les 8 premiers octets sont stockés dans la variable **key**, tandis que les 8 derniers octets sont stockés dans la variable **num2**.

**key** correspond en fait à un index (généré de manière incrémentale) dans un objet de type dictionnaire, où sont stockés les tickets valides.

Si le ticket n'est pas trouvé par la méthode **TryGetValue()**, la fonction retourne immédiatement **null**. Dans le cas contraire le ticket est retiré du dictionnaire.

Une vérification supplémentaire est effectuée : **num2** doit être égal à la valeur **Secret** du ticket. Cette valeur est générée aléatoirement à la création du ticket par la primitive (sûre) `System.Security.Cryptography.RandomNumberGenerator`.

Si toutes ces vérifications réussissent, le contexte stocké dans le ticket est retourné.

Pour finir, les tickets ont une durée de vie de 2 minutes par défaut à la création.

**Générateur de tickets** Le constructeur de la classe **TicketManager** est donné ci-dessous.

```
public TicketManager(TimeSpan ticketExpiry)
{
    this.tickets = new Dictionary<ulong, Ticket>();
    this.ticketExpiry = ticketExpiry;
    this.randomGenerator = RandomNumberGenerator.Create();
}
```

**ticketExpiry** est une constante définie à 2 minutes.

La génération des tickets individuels repose sur la méthode **GenerateInternal()**, dont le cœur est le suivant :

```
lock (this)
{
   ulong num;
```

```
        this.randomGenerator.GetBytes(data);
        workItem.Secret = BitConverter.ToUInt64(data, 0);
        this.nextTicketId = (num = this.nextTicketId) + ((ulong) 1L)
            ;
        workItem.TicketId = num;
        ThreadScheduler.GetScheduler().Schedule(workItem, DateTime.
            UtcNow + this.ticketExpiry);
        this.tickets[workItem.TicketId] = workItem;
    }
```

**Sécurité du schéma d'authentification** A la lumière du processus précédent, il est possible de reconstruire le schéma de passage d'authentification entre le protocole SIP et le protocole PlaceWare.

1. Le client s'authentifie via le protocole SIP.
2. Le serveur SIP génère un ticket d'authentification, contenant un index séquentiel, un élément aléatoire de 8 octets et une durée de vie de 2 minutes.
3. Le serveur SIP transmet le ticket au client dans le champ « sAuthId ».
4. Le client a 2 minutes pour se reconnecter en protocole PlaceWare et présenter son ticket.
5. Le ticket est détruit à la première tentative d'authentification (réussie ou non).

Ce schéma semble plutôt robuste. La seule faille envisagée est la destruction des tickets en cours de validité par un attaquant malveillant, compte-tenu du fait que les numéros de ticket sont générés de manière incrémentale (donc relativement faciles à prédire). Pour agir, l'attaquant doit envoyer sa demande d'authentification entre la connexion SIP et la connexion PlaceWare du client légitime, ce qui laisse une fenêtre de tir très étroite.

### 7.2 Génération d'aléa

**Question posée** La génération d'aléa est toujours un point chaud pour la sécurité des applications (ex. génération de clés de chiffrement, de cookies de session, etc.). Il est en général vital que l'aléa généré ne soit pas prédictible par un attaquant, même s'il a eu accès à plusieurs valeurs antérieures du générateur.

La question qui se pose alors est la suivante : « quels sont les générateurs d'aléa utilisés par OCS, et à quoi servent-ils ? »

**Réponse** En combinant des techniques d'analyse statique (références croisées) et d'analyse dynamique (points d'arrêt), il est possible d'identifier que les générateurs d'aléa suivants sont utilisés dans la version OCS 2007 « R1 » :

- java.security.SecureRandom
- java.util.Random
- System.Random

« java.util.Random » est un simple *wrapper* de la classe « System.Random ».

L'implémentation de « System.Random » est basée sur l'algorithme soustractif de Donald E. Knuth. La documentation Microsoft indique que cette implémentation n'est pas forcément sûre [49] :

*« To generate a cryptographically secure random number suitable for creating a random password, for example, use a class derived from System.Security.Cryptography.RandomNumberGenerator such as System.Security.Cryptography.RNGCryptoServiceProvider. »*

De plus la classe « System.Random » est toujours instanciée par la construction suivante : « x = new Random() ». En l'absence de paramètre fourni au constructeur, la graine d'initialisation par défaut est « System.Environment.TickCount », donc le nombre de millisecondes écoulées depuis le dernier redémarrage du système.

Au final un générateur d'aléa qu'on peut considérer comme « non sûr » est donc utilisé dans toutes les classes suivantes :

- placeware.apps.aud.AudienceS
- placeware.apps.aud.SlideFiles - en particulier les méthodes createName() et createRandom()
- placeware.apps.aud.SlideViewerS
- placeware.apps.blobparts.BlobManagerS
- placeware.security.RandomString
- placeware.util.PWTime
- Microsoft.Rtc.Internal.Sip.SipDialog
- Microsoft.Rtc.Internal.Sip.ConnectionControlModule
- placeware.security.RandomString
- placeware.apps.aud.policy

Prenons l'exemple de la méthode « createRandom() » issue de la classe « placeware.apps.aud.SlideFiles », dont le code est le suivant.

49. http://msdn2.microsoft.com/en-us/library/system.random.aspx

```
public virtual string createRandom(string extension, string why
    )
{
    string key;
    do {
        long lnum = (rand.nextLong() & 0x7fffffffffff) | 0
            x800000000000;
        key = new StringBuffer().append("x").append(Long.
            toHexString(lnum)).append(extension).ToString();
    } while (((SlideFileInfo) this.counts.get(key)) != null);
    return key;
}
```

Dans le cas où des supports sont échangés lors d'un *meeting*, le nom des fichiers tels qu'ils sont créés sur le serveur Web de partage est donc issu de la concaténation des éléments « x », « rand » et « extension », où « rand » provient du générateur d'aléa « non sûr ».

Ces fichiers sont par la suite laissés en accès libre sur le serveur Web pendant une durée de conservation qui est de 14 jours par défaut. Même en l'absence de *Directory Browsing* sur le serveur Web, il est donc envisageable qu'un attaquant puisse récupérer ces fichiers en devinant leur nom.

La suite de l'étude a toutefois montré que ces fichiers étaient chiffrés en AES avec une clé de *meeting* temporaire issue d'un générateur d'aléa « sûr ».

## 8 Conclusion

De mon expérience, l'audit applicatif a supplanté l'audit de systèmes et de réseaux dans les besoins exprimés par les clients.

Malheureusement, plusieurs facteurs contribuent à une inflation démesurée de la taille et de la complexité des applications industrielles : environnements de développement et librairies toujours plus riches, empilement de couches logicielles au fur et à mesure des évolutions, accroissement de la puissance des machines, etc.

Dans ces conditions, l'audit applicatif en « boite noire » devient un exercice complexe, alors que les éditeurs ne maitrisent parfois plus eux-mêmes les mécanismes internes de leurs produits.

Fort heureusement, la richesse sémantique du *bytecode* .NET permet de disposer d'outils et de méthodes d'audit en « boite noire » efficaces, comme cet article tend à le démontrer sur un monstre de complexité : le produit Microsoft OCS 2007.

Compte-tenu du nombre de nouveaux développements réalisés sur la plateforme .NET, le développement d'outils et la montée en compétences sur le sujet s'avère être un investissement d'avenir. Il resterait à fédérer une communauté de gens intéressés par .NET et souhaitant partager le fruit de leurs recherches, comme cela est déjà le cas dans le domaine des applications « natives » (Win32/x86).