Formation C# – Delphi.NET – Delphi Win32
Développement & Sous-traitance

Introduction à la sécurité sous .NET

L'environnement .NET a été conçu pour répondre aux problèmes de sécurité qui se posent aujourd'hui sous Win32. Si aucun système ne peut garantir l'absolue sécurité, il faut avouer que Win32 n'avait pas été préparé aux attaques des milliers de vers, chevaux de Troie, spams, et autres virus qui gâchent la vie quotidienne de l'utilisateur. Le framework, par construction, pose les bases d'une gestion de la sécurité non plus seulement au niveau des accès machines, ce que Win32 savait faire, mais aussi au niveau du code exécuté, ce qui est nouveau. S'il n'est pas dans le propos de ce livre d'aborder dans ses moindres détails l'aspect sécurité de .NET, il nous a semblé néanmoins nécessaire de vous présenter les grands mécanismes à l'œuvre.

La sécurisation du code

Comme vous le savez maintenant les logiciels compilés pour la plate-forme .NET ne sont pas exécutés simplement par le microprocesseur comme c'était le cas sous Win32. Sur cette plate-forme le code compilé n'est autre que du code machine binaire intégrant des appels à des bibliothèques externes, les API de Windows. Ce dernier n'a aucun contrôle sur le code exécuté, il ne fait que servir les demandes aux API.

Sous .NET le code est avant tout compilé par le CLR, placé en cache, et exécuté sous contrôle du même CLR. C'est pour cette raison qu'on parle de « code managé » ou « code géré » quand on parle de code natif .NET. Comme tout est objet sous .NET, une application est une classe qui contient d'autres classes. Seules les méthodes appelées sont compilées par le JITer et tout code exécuté n'est qu'instanciation de ces classes par le CLR.

Ce contrôle permanent du code autorise la nouvelle plate-forme à procéder à des vérifications beaucoup plus strictes que ne pouvait le faire Win32.

La structure même d'un assemblage permet avant toute exécution des contrôles étendus impossibles à réaliser sous Win32. Par exemple le CLR va pouvoir valider le format du fichier exécutable ce qui permet de rejeter tout code n'ayant pas un format correct et éviter l'exécution d'un fichier détérioré. Mais comme le format PE des assemblages contient aussi des métadonnées, le CRL va pouvoir exercer ses contrôles plus en profondeur, notamment vérifier qu'aucune adresse ne pointe des éléments se trouvant en dehors de l'assemblage. Tout code malicieux ou défectueux est ainsi immédiatement isolé et n'a pas même la possibilité de commencer son exécution. D'ailleurs les métadonnées sont elles-mêmes inspectées avant d'être utilisée pour valider le code.

Lors de la compilation par le JITer, le code est à nouveau contrôlé. Transtypages invalides, taille incohérente de tableaux, utilisation non régulière de pointeurs, tout cela (et bien d'autres choses) est vérifié. Tout code présentant des défaillances potentielles comme des dépassements mémoire est ainsi éliminé. Il n'est pas compilé et ne peut donc pas être exécuté.

Lors de l'exécution d'un code le CLR s'assure aussi de l'intégrité du code notamment grâce à la notion de « nom fort » (*strong name*) c'est-à-dire la signature numérique des assemblages. Un nom fort est constitué à la fois d'une version, d'une culture et d'un cryptage par clé. Lorsqu'un exécutable signé est exécuté par le CLR ce dernier vérifie ainsi que le code est parfaitement intègre et qu'il n'a pas subi d'altération. Les virus qui modifient le code d'un

exécutable en s'insérant dans celui-ci ne peuvent plus accomplir de telles manipulations. Si la modification du code reste possible sur disque, tout code modifié possédant une signature ne passera pas les barrières du CLR, ce dernier comparant à chaque fois la clé de hash calculé à la signature du code avec celle qu'il calcule sur le code avant son exécution.

Enfin, le framework propose une notion de sécurité d'accès au code. Comme ce dernier reste sous contrôle du CLR, il est possible d'attribuer des droits plus ou moins restrictifs à un code exécuté en fonction des droits possédés par l'appelant. Ainsi, si un logiciel ne reçoit pas le droit d'écrire sur disque il lui sera impossible d'outrepasser cette sécurité en appelant un code extérieur qui lui aurait cette permission. Le code appelé n'a pas plus de droit que le code appelant.

Arrivé ici vous comprenez certainement beaucoup mieux pourquoi les techniques d'appel de code Win32 sont considérées dangereuses et pourquoi l'utilisation de sections marquées « unsafe » sont ainsi appelées… Le CLR ne peut en aucun cas exercer les mêmes contrôles sur le code Win32 ou le code .NET non sécurisé. Ces appels, ces portions de code sont autant de failles de sécurité dans lesquelles les nouveaux virus, vers et chevaux de Troie tenteront de pénétrer.

Les deux facettes de la sécurité

Le framework .NET propose deux approches complémentaires pour sécuriser les applications. La première consiste à identifier les utilisateurs, c'est le niveau de sécurité qu'on retrouve sous Win32. L'identification d'un utilisateur permet de savoir qui il est et ce qu'il a le droit de faire ou non. Cette sécurité est dite sécurité basée sur les rôles. La seconde concerne l'identification du code lui-même. Le framework tente de connaître la provenance du code, la société qui l'a écrit et les droits attribués à ce code. Cette sécurité est appelée sécurité d'accès au code (CAS, Code Access Security).

Les rôles

La sécurité basée sur les rôles se rapproche dans son principe des rôles des bases de données SQL. Les rôles sont souvent utilisés dans les applications car ils permettent de façon simple de découper le champ des droits à attribuer. Les rôles peuvent même être calqués sur ceux que les diagrammes de cas d'une analyse UML auront dégagés. Les rôles peuvent être vus comme des « méta utilisateurs ». C'est-à-dire des classes d'utilisateurs. Les droits sont

attribués à une classe précise, un rôle, et ce n'est qu'ensuite qu'on affecte des rôles aux utilisateurs physiques.

Une application qui utilise correctement la gestion des rôles de .NET verra sa sécurité notablement renforcée. Ce n'est pas le logiciel qui teste si l'utilisateur X ou Y a le droit d'effectuer telle ou telle action, c'est le framework qui interdira les utilisateurs non autorisés à effectuer certaines actions. La différence majeure se situe dans le fait que toute sécurité codée dans l'application n'est que du code qui contiendra par force des bogues… Il peut donc y avoir des cas non testés autorisant l'exécution d'actions qui devraient être interdites. En se reposant sur la gestion des rôles du framework ce genre de problème est déporté vers ce dernier dont les failles sont plus facilement identifiées et corrigées qu'un code propriétaire.

Par défaut la sécurité basée sur les rôles repose sur les groupes Windows. Si les utilisateurs n'utilisent pas cette plate-forme le code peut lui-même définir les rôles et stocker les définitions dans une base de données ou dans un conteneur de type Active Directory.

Le contrôle des rôles est utilisé dans les applications Web ce qui permet aux applications de limiter les accès à certaines partie des sites sur la base d'une hiérarchie de rôles clairement définis. Une application Web peut prendre connaissance du rôle de l'utilisateur qu'elle est en train de servir par le biais d'un objet du framework (l'objet `Principal`) qui est accroché à toute requête entrante.

Par souci d'homogénéité avec la sécurité d'accès au code (CAS, voir plus bas) le framework propose des objets de type `PrincipalPermission` qui permettent au CLR de travailler sur les rôles d'une façon similaire à celle des autorisations de type CAS. La sécurité peut être déclarative ou impérative et une application .NET peut prendre connaissance à tout moment du rôle de l'utilisateur connecté pour adapter son comportement (par exemple présence ou absence de certaines entrées de menu selon les droits).

Si la sécurité basée sur les rôles est particulièrement bien adaptée aux applications Web ASP.NET elle est utilisable dans toutes les applications.

PrincipalPermission

Cette classe, du moins ses instances, représente l'identité et le rôle qu'une classe du code doit obtenir pour pouvoir s'exécuter. `PrincipalPermission` peut être utilisé à la fois en mode déclaratif et en mode impératif.

Déclaratif ou impératif

La distinction entre ses deux modes s'effectue sur la façon dont les demandes au système des permissions sont décrites. Le mode impératif consiste à créer des instances des classes de sécurité dans le code même. Le mode déclaratif passe lui par la déclaration d'attributs. Les deux façons d'opérer ont leurs avantages et inconvénients respectifs. Par exemple le contenu d'un attribut, notamment les valeurs des propriétés, est fixé à l'écriture du code alors qu'en instanciant un objet dans le code on peut fixer dynamiquement la valeur des propriétés.

Pour mettre en place une sécurité impérative il suffit de créer une nouvelle instance de `PrincipalPermission` en indiquant l'identité et le rôle qu'on souhaite imposer au code qui suit pour qu'il puisse s'exécuter :

```
// C#
string id = "olivier";
string rôle = "direction";
PrincipalPermission pp = new PrincipalPermission(id,rôle);

// Delphi.NET
Var
id : string = 'olivier';
rôle : string = 'direction';
pp : PrincipalPermission ;
…
pp := PrincipalPermission.Create(id,rôle);
```

Il est possible d'obtenir le même résultat de façon déclarative en utilisant un attribut dans le code :

```
// C#
[PrincipalPermissionAttribute(SecurityAction.Demand,
                              Name = "olivier", Role = "direction")]
```

Lorsque le contrôle de sécurité est rencontré par le CLR à la fois l'identité et le rôle doivent correspondent pour que le code puisse s'exécuter. Il est bien entendu possible de passer une identité « à vide » pour indiquer que seul le rôle doit être pris en compte, ce qui est le mode d'utilisation le plus courant. Il est aussi possible de faire l'inverse, à savoir fixer une identité sans rôle, ce qui permet de contrôler l'identité d'un utilisateur bien précis quel que soit son rôle.

CAS

CAS permet de limiter les accès aux ressources sensibles et donc sécurisées comme les fichiers disque, le réseau, la base de registres. De plus CAS étend son filtrage à l'appel du code non managé ou à l'utilisation de la réflexion. CAS permet un verrouillage du code lui-même quels que soient les droits de l'utilisateur. Le code est lui-même considéré comme un utilisateur qui possède des droits, ce qui est nouveau et offre un niveau de sécurité totalement absent de la plate-forme Win32.

CAS possède lui-même deux facettes, celle qui consiste à tester le code en tant que tel, techniquement, et celle qui consiste à contrôler les droits attribués à ce même code.

Le premier contrôle est effectué par le CRL à l'exécution. Si un code possède un nom fort sa signature est contrôlée. Si le test échoue le code ne sera pas exécuté. Si le code passe ce test (ou qu'il n'est pas signé), le CRL contrôle l'utilisation qu'il fait de la mémoire et vérifie s'il est typé correctement. Les tests effectués à ce niveau peuvent soit rendre le code impossible à exécuter, soit il peut se voire attribuer un niveau de confiance très faible (celui du code non géré). Si un code est classé comme non géré il lui faudra obtenir les droits « full trust » (confiance absolue) pour s'exécuter. Par défaut tout exécutable local sur une machine possède ce niveau d'autorisation (ce qui peut être modifié bien entendu).

Bloquer le code malveillant

CAS se fonde sur un système dit par preuves pour décider des droits qui seront attribués à une application donnée. On peut prendre pour exemple les applications exécutées depuis Internet. La police de sécurité par défaut attribue à un tel code un ensemble de droits dont est exclu la possibilité d'appeler du code non géré. Si cette même application est téléchargée sur le disque local puis exécutée, la police par défaut lui donnera les droits « full trust » ce qui lui permettra d'exécuter des appels à du code non géré.

Service de cryptographie

Le framework offre un large choix d'outils ayant trait à la cryptographie. La signature numérique du code et la vérification de cette signature sont l'une des premières utilisations qui est faite du service de cryptographie. Mais celui-ci met à la disposition du développeur de nombreuses classes incluant :

- Des algorithmes de hash (par exemple MD5) ;
- Cryptage symétrique (à clé secrète partagée) ;
- Cryptage asymétrique (à clé publique partagée).

Sécurité des assemblages

Sous Win32 tout le monde connaît les innombrables problèmes liés à l'utilisation des DLL, ce que certains ont appelé « l'enfer des DLL » (DLL Hell pour nos amis anglo-saxons). Toute personne ayant en effet les droits suffisants peut écraser une DLL par une

version plus ancienne déstabilise par le fait même tous les logiciels s'en servant. Pire, une âme malveillante peut remplacer une DLL par une autre de même nom pour créer une *back door* ou détruire des données. Win32 n'offre absolument aucun moyen sérieux de se prémunir contre ce genre de choses. La difficulté a été reportée sur les installateurs sensés effectuer les contrôles, mais il suffit d'installer Delphi 7 puis Delphi 2005 et de désinstaller le premier pour perdre le BDE par exemple ou le déstabiliser… Et pourtant l'installation d'un environnement comme Delphi est effectuée dans les règles de l'art avec des générateurs d'installation professionnels… Nous ne parlerons même pas des logiciels fournis dans une fichier compressé, sans aucun module d'installation.

Bref, il fallait trouver une parade. Sous .NET les DLL (et les EXE) sont appelées des assemblages, et un assemblage est la plus petite partie déployable d'une application. Un assemblage contient du code, des données (et métadonnées), l'un, l'autre ou les deux. Mais la nouveauté est que les assemblages ont une identité unique (ou peuvent avoir une identité unique). Cette identité est garantie par plusieurs éléments comme un nom de fichier, un numéro de version, une culture et une signature numérique.

Grâce à cette identité, une DLL n'est pas repérable par son seul nom de fichier, celui-ci n'est qu'un élément de son identité unique. Cela permet au CLR de gérer des DLL de même nom physique mais d'identités différentes. Ainsi il est possible de fournir une DLL dans sa version X et la même dans sa version X+1 sans que cela n'interfère. Les logiciels utilisant la version X obtiendront celle-ci du CLR, ceux utilisant la version plus récente l'obtiendront, les deux versions pouvant être chargée simultanément en mémoire, le tout sans aucune collision ni confusion.

Le déploiement d'une DLL peut s'effectuer de deux façons, soit en la fournissant tout simplement dans le répertoire de l'application s'en servant, soit en la déployant dans le GAC (Global Assembly Cache).

La signature numérique

La signature numérique d'une DLL ou d'un exécutable est un moyen simple de s'assurer que l'assemblage ne pourra pas être modifié accidentellement ou volontairement. C'est un élément de sécurité indispensable. Comme nous l'avons vu la signature numérique est aussi un élément permettant de créer une identité unique pour une DLL.

Si .NET propose des solutions pour éviter le fameux enfer des DLL rien ne peut fonctionner si les assemblages n'ont pas une identité unique. Tant qu'une DLL n'est pas signée rien ne permet de garantir qu'une version différente ne la remplacera pas, au risque de voir les mêmes problèmes que ceux de Win32 se reproduire.

Créer l'identité unique d'une DLL est un processus simple qui permet de lui conférer ce qu'on appelle un nom fort. Ce nom est fort car le nom physique est complété par des éléments supplémentaires rendant l'ensemble unique. La signature numérique est la clé de cette unicité. L'annexe 5 présente l'outil `SN.EXE` du framework ainsi que la façon de s'en servir pour signer un assemblage.

La signature en elle-même peut être effectuée de deux façons différentes. Soit en utilisant le linker `AL.EXE` du framework qui possède une option `/keyfile` permettant de spécifier le nom du fichier des clés, soit en renseignant dans le code source du projet les attributs personnalisés suivants :

```
[assembly: AssemblyDelaySign(false)]
[assembly: AssemblyKeyFile('nom du fichier de clé')]
[assembly: AssemblyKeyName('')]
```

La première ligne indique que nous n'utilisons pas le système de signature retardée, la seconde indique le nom du fichier de clés. La troisième ligne trouve son sens lorsque des clés sont installées au niveau du framework et qu'on souhaite signer l'assemblage en utilisant celle-ci plutôt qu'un fichier de signature.

Le processus de signature est assez sophistiqué puisqu'il fonctionne en plusieurs passes en prenant en compte le nom des assemblages référencés par l'assemblage signé, que les clés de hash sont inscrites dans les métadonnées et que tout cela est de nouveau hashé en mélangeant la culture, les informations de version pour au final créer un assemblage signé numériquement. Comme tout système basé sur des clés, si ces dernières sont volées elles pourront servir à fabriquer des assemblages qui pourront passer pour des originaux.

La signature différée

Comme nous l'évoquions plus haut les clés de signatures sont aussi précieuses que les clés d'un coffre-fort. Se les faire voler c'est donner la possibilité de se faire piller, et dans le cas des assemblages de voir des copies passer pour des originaux. Concurrents déloyaux, pirates, virus, employés déçus à l'esprit vengeur, autant d'entités qui déploieront les moyens nécessaires pour voler vos clés…

Comment faire en sorte que les clés soient en la possession uniquement de personne réputées fiables d'autant que la signature est un mécanisme intervenant lors de la compilation (ou de l'édition de lien) ?

C'est pour répondre à cette attente des entreprises que le système de signature différé à été créé.

Les fichiers de clés créés par l'utilitaire `SN` comportent deux sections : la clé privée et la clé publique. Il est possible de séparer les deux clés en utilisant l'utilitaire `SN` de la façon suivante :

```
SN -p e-naxos.snk e-naxos.public.snk
```

Cette commande extraira la clé publique de la paire de clé stockée dans `e-naxos.snk` et placera le résultat dans `e-naxos.public.snk`.

Une fois la clé publique isolée, elle peut être diffusée auprès des développeurs. Les applications et assemblages seront signés à l'aide de cette clé et l'attribut de signature différée (voir plus haut) sera positionné à `true`.

La clé privée reste elle entre les mains d'une personne de confiance, le directeur informatique par exemple.

Les assemblages signés partiellement par cette méthode seront rejetés par le CLR car le nom fort n'est pas valide. Pour que les développeurs puissent les tester il faut lever la vérification toujours par le biais de `SN` :

```
SN -Vr assemblage.en.test.dll
```

Une fois l'assemblage testé et validé techniquement le possesseur de la clé privée pourra le signer définitivement par les commandes suivantes :

```
SN -R assemblage.testé.dll e-naxos.snk
SN -Vu assemblage.testé.dll
```

La première commande utilise le fichier de clés original contenant la clé privée pour compléter le processus de signature, la seconde rétablit les vérifications du CLR (supprimées par `-Vr` pour permettre les tests de l'assemblage) .

Le certificat d'éditeur

La signature numérique d'un assemblage lui confère un nom fort et permet de nombreuses vérifications par le CLR. Mais cette signature ne permet pas de savoir d'où provient le code, de qui il émane, c'est-à-dire quel est son éditeur.

Il existe donc un second procédé de signature, *Signcode*, qui permet de signer un logiciel avec un certificat d'éditeur de logiciel (Software Publisher Certificate ou SPC). Un tel certificat s'obtient d'une autorité de certification tierce comme peuvent l'être Thawte ou Verisign. Ces tiers de certification attribuent des certificats électroniques aux éditeurs de logiciels, ces certificats étant uniques, limités dans le temps et permettant d'identifier la société éditrice de façon exacte.

La signature par certificat est indépendante de la signature par nom fort. De fait toutes les combinaisons sont possibles (signé par certificat mais sans nom fort, nom fort mais pas de certificat, signé par les deux méthodes, aucune signature).

L'achat d'un certificat n'est pas très cher pour une entreprise, par exemple la société Thawte propose des certificats de signature de code pour moins de 200 dollars par an. Avant d'acheter un certificat il est possible de tester le procédé grâce à l'outil `makecert` du framework qui génère des certificats de test conformes à la norme X.509.

Nous n'entrerons pas plus dans les détails de ces opérations qui n'intéresseront que peu de nos lecteurs. Les autres, qu'ils nous en excusent, trouveront des informations complètes dans la littérature technique spécifiquement dédiée à la sécurité sous .NET, ce qui n'est pas le but du présent ouvrage.

Décompilation

Tout logiciel peut être décompilé. Les meilleures protections sous Win32 sont la proie des crackers et tout le monde connaît de drôles de mules capables de fournir les dernières versions des plus grands logiciels pourtant protégés par des systèmes ayant coûtés très chers à leurs éditeurs… Et ce bien que le code soit purement binaire et assez difficile à décompiler et interpréter.

Il est dès lors évident que le code .NET qui n'est que du IL facilement lisible et même traductible en différents langages est une proie rêvée pour toute manœuvre de reverse engineering. À cela Microsoft n'a pas fourni de solution, étonnamment.

Il existe malgré tout des solutions pour protéger le code des applications :

- Déplacer les parties critiques vers des processus serveurs.
- Compiler les assemblages en binaire natif.

- Utiliser un obfuscateur de code.
- Déplacer le code ultra sensible dans des DLL Win32.

Déplacer le code vers les serveurs

Il est évident que tout ce qui peut être placé sur des serveurs bien protégés ne pourra pas être accessible de l'extérieur et donc décompilé. Cette recommandation s'applique surtout aux applications Web mais elle est parfaitement exportable à toutes les autres applications notamment par le biais de .NET Remoting et des services Web.

Toutefois si les applications placées sur les serveurs sont elles-mêmes diffusables (c'est le cas d'un logiciel commercial) cette protection n'est pas suffisante, le client acheteur du logiciel pourra le décompiler. Certains algorithmes peuvent néanmoins être localisés sur des serveurs Internet, les logiciels vendus faisant obligatoire appel à ces processus distants dans le cours de leur fonctionnement normal. Si cela parait encore aujourd'hui très contraignant le lecteur doit savoir que c'est l'avenir du logiciel. Cela fait des années que des firmes comme Microsoft misent sur la généralisation d'un Internet hyper rapide et popularisé pour stopper définitivement tout piratage. En effet, si pour fonctionner Word ou Excel doivent appeler des fonctions situées sur les serveurs de Microsoft plus question de « déplomber » un simple exécutable. Pirater un tel logiciel consistera simplement à copier une coquille vide, une interface sans code ou presque. Ce style de développement sera généralisé lorsque les conditions techniques le permettront, soyez-en certains. D'un autre côté ce mode de développement ouvrira aussi la voie à la location de logiciel. On peut imaginer qu'un jour nous n'achèterons plus de licences pour notre OS, notre traitement de texte, nous payerons avec notre carte bancaire, en ligne, des abonnements annuels, comme pour les anti-virus qui fonctionnent déjà de cette façon. Demain nous louerons un quota de signes tapés annuellement sous Word, les auteurs de livres comme votre serviteur seront alors lourdement taxés !

Compiler les assemblages en binaire natif

Il existe des outils comme Salamander .NET de RemoteSoft qui permettent de compiler en Win32 un code .NET en liant tout ce dont il a besoin, même les fonctions du framework qu'il utilise. Il ne s'agit que d'un début, vraisemblablement de tels outils ont un bel avenir, ne serait-ce que pour diffuser des logiciels .NET en

version Win32 autonomes sans avoir à maintenir deux versions du produit.

Il faut noter que le compilateur `NGen.exe` du framework effectue la même opération mais qu'il ne rend pas le logiciel indépendant de son assemblage d'origine (ni du framework) qui doit toujours être déployé. Le but de `NGen` est d'éviter le temps de compilation des classes de l'application lors des premiers appels qui y sont faits et non de transformer l'assemblage en un fichier binaire Win32 indépendant de .NET.

Utiliser un obfuscateur de code

Nous abordons à l'annexe 5 les moyens d'utiliser un tel outil. En quelques mots un obfuscateur de code est un logiciel qui analyse et transforme un assemblage. Les transformations consistent pour l'essentiel en le brouillage des noms des identificateurs. Ainsi la méthode « `CréerFacture` » deviendra-t-elle « `x2` » et la propriété « `MotDePasse` » se verra traduite par « `_a7z` » (ce ne sont que des exemples fictifs). De fait, même en décompilant le code IL de l'assemblage suivre l'algorithme deviendra très ardu. Ardu ne veut pas dire impossible, sous .NET comme sous Win32 ou autre OS, aucune protection n'est invincible.

Il existe plusieurs obfuscateurs sur le marché, certains gratuits, d'autres payants. Borland fournit Demeanor, à vous de tester et de choisir celui qui convient le mieux à vos besoins.

Appeler du code non géré

Le dernier recours, la dernière cachette possible pour un algorithme très sensible c'est encore une DLL Win32 classique. Ni code IL, ni métadonnées. Rien que du code microprocesseur bien difficile à lire (sauf pour un hacker motivé).

Une telle solution est vraiment à réserver aux cas extrêmes. L'utilisation de code non géré dans une application .NET n'étant vraiment pas une chose à conseiller en dehors d'un contexte bien particulier. Toutefois, si le seul moyen de protéger un code très sensible consiste à le fournir sous forme d'une DLL Win32, et si cela peut éviter de se faire voler une idée révolutionnaire, pourquoi pas.

La sécurité sous ASP.NET

La majorité des sites Web réclame une gestion des accès pour limiter l'utilisation de certaines parties du site. Par exemple une galerie photo est ouverte au public qui peut visiter le site (visualiser le contenu) mais les parties dédiées à la mise à jour des données (les news par exemple) ne pourront être accessible qu'aux seules personnes autorisées. Lorsqu'un site marchant stocke les informations de ses clients ou prospects dans une base de données l'accès à cette base doit lui aussi être strictement contrôlé. La gestion de la sécurité sous ASP.NET est là pour faciliter la résolution des différents problèmes liés à la sécurisation des sites Web (et des services Web).

ASP.NET fonctionne en conjonction avec IIS (Microsoft Internet Information Services), ce couple peut identifier les utilisateurs par différentes méthodes comme le simple login (nom / mot de passe), ces méthodes sont :

- Windows: Basic, digest, ou Integrated Windows Authentication (NTLM[1] ou Kerberos[2]) ;
- Authentification par Microsoft Passport ;
- Authentification par formulaire ;
- Authentification par certificat utilisateur.

ASP.NET contrôle les accès à un site en comparant les authentifications fournies à la sécurité du système de fichier NTFS ou à un fichier XML qui liste les utilisateurs autorisés, les rôles (groupes d'utilisateurs) etc.

Comment fonctionne la sécurité ASP.NET

La sécurisation d'un site Web est une opération essentielle. Plus un site est critique plus elle requiert une planification soigneuse.

ASP.NET utilise à la fois la sécurité de IIS et celle du framework pour aider à la mise en place d'une sécurisation efficace des ressources. Deux fonctions sont fondamentales et sont prises en charge :

---

[1] NTLM (NT LAN Manager) est un protocole d'identification originellement conçu pour sécuriser DCE/RPC. Microsoft l'utilise dans ses systèmes et logiciels (comme IE) comme mécanisme de login

[2] Kerberos est un autre protocole d'authentification dédié aux applications client/serveur – au sens large – qui utilise un mécanisme de cryptographie par clé secrète. Il existe plusieurs implémentations de ce protocole ouvert.

- *Authentification*.
  Cette fonction permet de vérifier si l'utilisateur connecté est bien celui qu'il prétend être. L'application obtient les preuves de l'identité de l'utilisateur par différent moyens à sa guise, le login (nom / mot de passe) étant le plus classique. Si les éléments soumis par l'utilisateur sont validés alors ce dernier est considéré comme authentifié.

- *Autorisation*.
  La gestion des autorisations permet de limiter les droits des utilisateurs en leur accordant ou non des privilèges.

Rappelons que ASP.NET fonctionne sur IIS (au moins sur la plate-forme Windows) et que ce dernier possède déjà sa propre gestion de sécurité qui permet d'octroyer ou refuser des droits par exemple en fonction du Host de l'utilisateur ou de son adresse IP. Ce premier barrage s'ajoute à la sécurité du système de fichier NTFS.

Mais comme ASP.NET est construit sur le framework .NET il bénéfice de tous les services de la plate-forme à la différence des CGI ou ISAPI Win32 qui ne reposent que sur la sécurité IIS. Il est ainsi possible, comme nous l'avons vu plus haut, de renforcer la sécurité des applications Web en mettant en œuvre la sécurité basée sur les rôles et la sécurité d'accès au code (CAS).

ASP.NET et IIS

La figure 23.1 décrit dans ses grandes lignes l'architecture de ASP.NET et les relations entre les divers tiers impliqués.

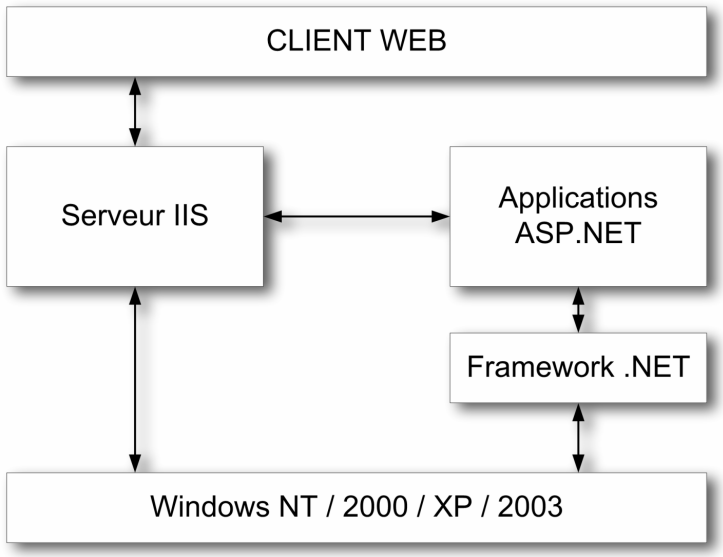

Figure 23.1

Architecture ASP.NET

Tous les clients Web dialoguent avec les applications ASP.NET au travers d'un serveur IIS. Ce dernier décode et optionnellement authentifie les requêtes entrantes. Si le paramétrage de IIS est fait de telle sorte que les requêtes anonymes sont autorisées alors aucune authentification n'est assurée. IIS s'occupe aussi de trouver la ressource demandée par le client (une page html, une application ASP.NET) et si le client est autorisé à y accéder il transfère le contrôle à cette ressource.

Pour rappel on notera que la sécurité sous IIS fonctionne sur la base de comptes NT qui servent à authentifier les utilisateurs. Mais outre la sécurité offerte par IIS, une application ASP.NET bénéfice de la sécurité spécifiquement disponible dans l'environnement ASP.NET. Celle-ci s'ajoute au premier barrage effectué par IIS. Les deux méthodes de sécurisation disponibles sous ASP.NET sont l'authentification par formulaire ou l'identification par Microsoft Passport. Ce dernier cas est de plus en plus rare, les grandes firmes délaissant de plus en plus ce mécanisme qui semble ne plus concerner que les produits et sites Microsoft.

L'authentification par formulaire est l'option qui est donc retenue par la grande majorité des sites Web développés sous ASP.NET. Le principe en est très simple : lorsqu'une ressource protégée est réclamée par un client ASP.NET effectue une vérification pour s'assurer que ce dernier est authentifié. S'il ne l'est pas il est redirigé automatiquement vers un formulaire d'identification (développé comme tout autre formulaire) permettant à l'application de contrôler l'identité et de valider l'autorisation d'accès.

Configuration de la sécurité

La sécurisation des sites Web développés avec ASP.NET est très simple à mettre en œuvre à la grande différence de IIS ou de Windows qui restent de ce point de vue assez obscurs. Il faut dire que Microsoft a fait de gros efforts de simplifications avec le framework .NET.

Ainsi, la configuration de la sécurité sous ASP.NET s'effectue par le biais de deux fichiers XML : `machine.config` et `web.config`. Le premier fichier est installé dans la racine du framework, dans le répertoire `config`. Les paramètres contenus dans ce fichier se propagent à tous les sites ASP.NET installés sur la machine, sauf modification locale. La modification locale des paramètres d'un site ASP.NET est effectuée par le biais du second fichier de configuration, `web.config`, qui doit être placé dans le répertoire même de l'application. Les réglages contenus dans ce fichier se propagent eux aussi à tous les sous répertoires qui eux-mêmes peuvent, ou non, contenir un nouveau fichier `web.config`.

Un fichier `web.config` peut contenir de nombreuses informations utiles à une application ASP.NET comme l'indication du mode de debug ou des paramètres applicatifs comme la chaîne de connexion à la base de données. En dehors de ces informations, le fichier contient une section dédiée à la sécurité, en voici le squelette :

```
<authentication mode="[Windows|Forms|Passport|None]">
    <forms name="[nomFiche]"
       loginUrl="[url]"
       protection="[All|None|Encryption|Validation]"
       path="[chemin]" timeout="[minutes]"
       requireSSL="[true|false]"
       slidingExpiration="[true|false]">
         <credentials passwordFormat="[Clear|MD5|SHA1]">
           <user name="[NomUtilisateur]"
           password="[MotDePasse]"/>
         </credentials>
    </forms>
     <passport redirectUrl="internal"/>
</authentication>
<authorization>
     <allow users="[Liste d'utilisateurs séparés par virgule]"
     roles="[Liste d'utilisateurs séparés par virgule]"/>
     <deny users="[Liste d'utilisateurs séparés par virgule]"
     roles="[Liste d'utilisateurs séparés par virgule]"/>
</authorization>
<identity impersonate ="[true|false]"
  userName="[domain\nom_utilisateur]"
  password="[mot_de_passe_utilisateur]"/>
  <trust level="[Full|High|Medium|Low|Minimal]"
  originUrl=""/>
<securityPolicy>
  <trustLevel name="Full" policyFile="internal"/>
  <trustLevel name="High" policyFile="web_hightrust.config"/>
  <trustLevel name="Medium" policyFile="web_mediumtrust.config"/>
  <trustLevel name="Low" policyFile="web_lowtrust.config"/>
  <trustLevel name="Minimal" policyFile="web_minimaltrust.config"/>
</securityPolicy>
```

Les éléments présentés ci-dessus sont largement documentés dans l'aide Microsoft que nous éviterons ainsi de recopier ici.

Vous noterez que la sécurité décrite dans le fichier `web.config` ne concerne que les applications ASP.NET. Les pages HTML, les images, et toutes les ressources autres que celles d'ASP.NET ne sont pas protégées par ce mécanisme. S'il reste possible de mapper manuellement sous IIS tous ces fichiers vers `Aspnet_isapi.dll` cela ne peut pas s'effectuer automatiquement pour un répertoire entier. Il est donc essentiel de protéger les ressources de ce type par la sécurité IIS ou bien de mettre en place un mécanisme d'indirection passant par une page

ASP.NET (qui peut fournir des ressources placées dans des répertoires non navigables, voire une base de données).

Mécanismes d'identification

Comme nous l'avons vu précédemment il existe plusieurs mécanismes pour sécuriser une application ASP.NET, le plus utilisé étant le mode formulaire. Le second procédé, qui consiste à laisser IIS valider les requêtes s'appelle l'impersonnalisation. Regardons plus en détail comment ces modes de contrôle d'accès fonctionnent.

Le login par formulaire

Dans ce mode c'est l'application ASP.NET elle-même qui au travers d'un formulaire de login va procéder à l'identification de l'utilisateur et lui octroyer ou non le droit d'accès aux ressources protégées. Les informations nécessaires à l'identification de l'utilisateur sont laissées à la discrétion du développeur. Si le strict minimum est habituellement un nom et un mot de passe cela peut être totalement différent d'un site à l'autre ou d'une partie de site à une autre.

On peut supposer par exemple un site totalement libre d'accès offrant une page « liens intéressants » que les utilisateurs du site peuvent remplir eux-mêmes. Tout le site peut être visité sans contrôle mais le concepteur veut s'assurer que le formulaire d'ajout d'un lien ne puisse pas être « vampirisé » par des robots qui procéderaient à des inscriptions automatiques de liens. Pour ce faire nul besoin de login ni de mot de passe, il faut ici seulement vérifier que l'utilisateur est un humain. Dans un tel cas le formulaire d'inscription sera placé dans un sous répertoire du site qui sera protégé par son propre fichier de configuration et par un formulaire d'accès. Ce dernier génèrera une image brouillée d'un nombre ou d'une combinaisons de lettres que l'utilisateur devra recopier. C'est d'ailleurs un procédé classique sur Internet aujourd'hui. De fait, la sécurisation d'un site ASP.NET peut être totale ou partielle et peut être basée sur des procédés aussi différents que le login traditionnel ou la génération d'un cryptogramme à lire et saisir par l'utilisateur.

Comme la sécurité ASP.NET se trouve en seconde position derrière celle de IIS il faut faire attentions aux interférences possibles entre ces deux niveaux de validation des accès. Si la sécurité IIS bloque certains utilisateurs ces derniers n'auront pas même accès au formulaire ASP.NET qui protège le site. C'est pourquoi, sauf cas particulier, la sécurité IIS est paramétrée en mode « accès anonyme » lorsqu'un site ASP.NET est protégé par un formulaire.

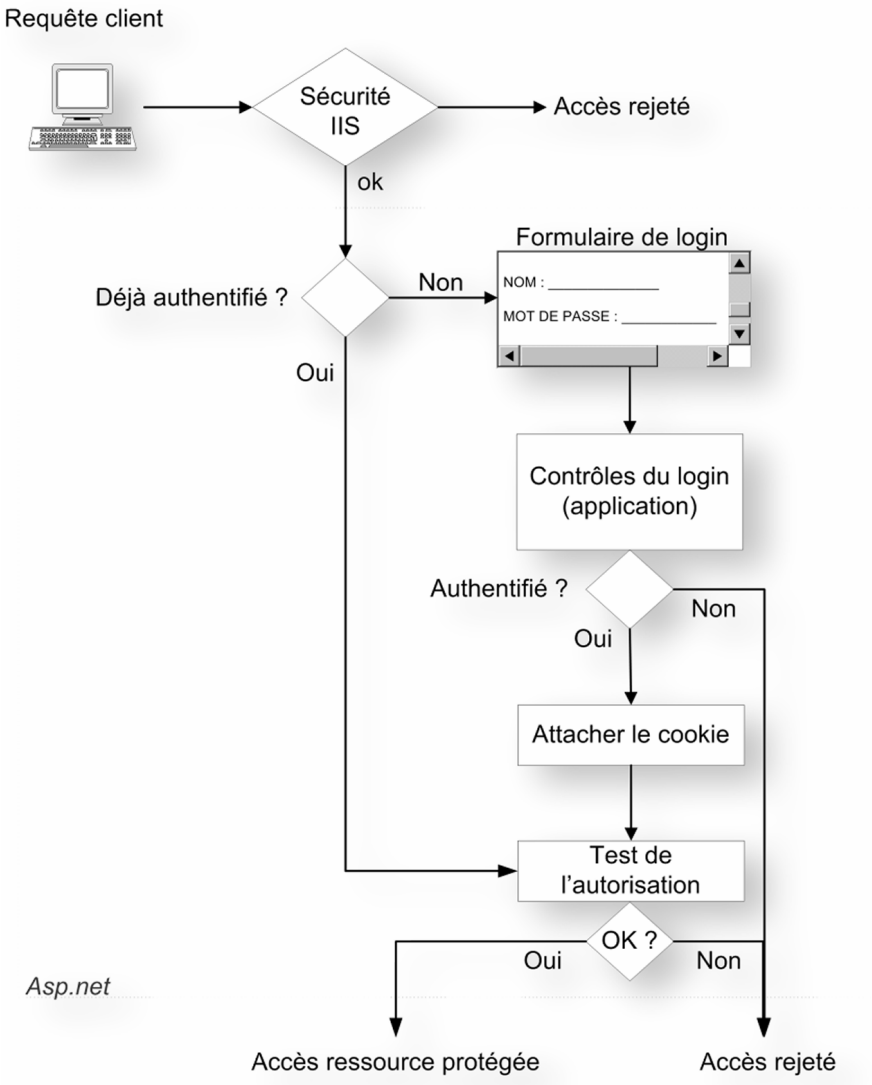

Figure 23.2

Sécurisation par mode formulaire

La séquence logique des événements lors d'une requête client à une ressource protégée suit le schéma proposé figure 23.2.

Le chemin parcouru par la demande du client à une ressource protégée passe en premier lieu par le serveur IIS qui applique ses propres règles de sécurité. Si ce premier barrage rejette la demande une page d'erreur est affichée et la requête de l'utilisateur prend fin.

Si la sécurité IIS laisse passer la requête celle-ci est alors transmise à ASP.NET qui vérifie en premier lieu si le client a déjà été authentifié (présence d'un cookie de sécurité dans cet exemple). Dans l'affirmative ASP.NET poursuit le contrôle en vérifiant si l'autorisation est toujours valide (elle peut avoir expirer par exemple). Si ce test est positif le client obtient la ressource protégée

qu'il a demandé, dans la négative sa requête échoue et l'accès est refusé.

Lorsque ASP.NET détecte que le client ne dispose pas encore d'une autorisation valide il redirige automatiquement le client vers le formulaire de login indiqué dans le fichier web.config. En général se formulaire réclame un couple nom / mot de passe et propose un bouton de validation. Le contrôle des informations saisies se fera généralement dans le gestionnaire d'événement du clic du bouton. Ici c'est le code du formulaire qui décide si les informations saisies sont valides ou non. Dans la négative l'accès est refusé. En général l'utilisateur est redirigé vers la saisie des éléments, un éventuel compteur peut décider au bout de quelques essais infructueux de renvoyer l'utilisateur soit à la page d'accueil du site soit une page d'erreur. Des améliorations peuvent être apportées pour renforcer la sécurité comme « griller » l'IP du client pendant un certain temps ou fournir un dialogue permettant en cas d'oubli du mot de passe de se le faire envoyer par mail.

Si le formulaire valide l'identité du client il utilisera une fonction du framework pour l'indiquer à ASP.NET. Cela se traduira par la création d'un cookie de sécurité (une autre méthode sans cookie existe). Une fois le contrôle rendu à ASP.NET, le flot reprend son cours et le cookie de sécurité va être validé. Ce point précis correspond au point d'entrée lorsque ASP.NET détecte au début de la séquence qu'un cookie de sécurité est déjà présent.

La validation du cookie de sécurité ayant été effectuée ASP.NET décide si l'utilisateur est autorisé ou non à accéder à la ressource protégée.

Le même cycle se répète pour tous les accès à une ressource protégée sur le site ou l'un de ses sous répertoires.

Login par impersonnalisation

Cette méthode repose intégralement sur la sécurité de IIS et de NTFS. La requête du client est validée par IIS qui contrôle si son IP est autorisée ou non et qui applique l'authentification sélectionnée (Basic, Digest ou authentification Windows). Au sortir de ces premiers tests la requête peut être rejetée. Si elle est acceptée IIS appelle l'application ASP.NET en lui passant l'information d'authentification. L'application ASP.NET vérifie si le mode impersonnalisation est autorisé ou non dans le fichier de configuration. Dans l'affirmative elle n'a rien d'autre à gérer, NTFS décidera de donner ou non accès à la ressource demandée.

Lorsque le mode impersonnalisation n'est pas activé dans la configuration de l'application ASP.NET celle-ci fonctionnera avec

les droits accordés au processus IIS. Sous Windows 2000 et XP l'application tournera avec les droits de l'utilisateur ASPNET qui est créé par ASP.NET lors de son installation. Sous Windows 2003 l'application tournera avec l'identité « network account service ».

Le choix entre une gestion des accès par mode formulaire ou par impersonnalisation dépend grandement du type de ressource à protéger et de l'infrastructure qui portera l'application. L'impersonnalisation réclame de pouvoir gérer les droits au niveau de IIS et de Windows. On conçoit bien par exemple qu'une application ASP.NET qui doit être hébergée sur un serveur mutualisé ne pourra en aucun cas exploiter cette méthode et devra gérer sa sécurité par formulaire. De fait l'impersonnalisation est une méthode qu'on réserve principalement aux Intranets, c'est-à-dire à des applications tournant sur une infrastructure totalement sous le contrôle des administrateurs.

Conclusion

La sécurité sous .NET se pose comme une couche au-dessus de la sécurité Windows. D'un côté elle est très simple à mettre en œuvre, de l'autre elle ne fait que rajouter des éléments à une gestion déjà fort complexe qui réclame les compétences d'administrateurs bien formés.

Si nous supposons que le développeur d'application .NET peut se reposer sur la compétence des administrateurs des machines et réseaux sur lesquels il déploiera ses applications alors la gestion de la sécurité avec .NET est finalement assez abordable.

Cet article vous a présenter les grandes lignes de la sécurité sous .NET, n'hésitez pas à vous reporter sur la littérature spécialisée si vous devez concevoir des applications sensibles réclamant une gestion fine des contrôles d'accès.