

## Sécurisez vos applications .Net : Partie 2

### Les règles de base

06/06/2004  
Par Elise Dupont

Article paru sur 

L'intégralité des tutoriaux et des codes sources sont disponibles sur <http://www.dotnet-tech.com/tutotiels/>

#### **Droit de diffusion:**

L'ensemble ou partie de ce document ainsi que le code mis à disposition, ne peut être diffusé sur d'autres sites Web sans l'autorisation au préalable de son créateur.

#### **Avant Propos :**

Dans ce deuxième volet sur la sécurité avec .Net, je vais vous expliquer les règles de base d'une application (ASP.Net ou Winform) sécurisée du point de vue du développeur. Ces règles consistent à vous protéger des données en entrée, à limiter les informations en sortie, à se protéger des principales attaques...

Les « grosses » parties telles que l'encryption, l'authentification, l'autorisation ou la sécurisation des communications feront de sujets d'articles plus longs dédiés à chacun de ces thèmes dans cette même série.

Si vous vous êtes renseigné sur le sujet de la sécurité applicative, il est fort possible que cet article ne vous apporte que peu de choses, mais pour les personnes qui ne se sont jamais documentées sur le sujet, il est toujours bon de connaître quelques règles simples et faciles à mettre en place, tout développeur devrait les avoir en tête et les appliquer à chaque développement.

On a souvent tendance à faire l'impasse sur la sécurité car on ne sait pas trop comment s'y prendre, ou alors on estime que c'est une perte de temps (et donc d'argent). Il est

vrai que la sécurité d'une application se définit bien avant le développement, pendant et après le développement, que cela coûte, et « qu'à priori » certaines applications semblent ne pas en avoir besoin (j'ai fréquemment entendu des développeurs intranet dire que la sécurité n'est pas importante en interne pas exemple, alors qu'il est si facile par intranet d'accéder à des données sensibles !). Le but de cet article n'est pas de vous convaincre du bien fondé de la sécurité applicative (et de l'intérêt d'y investir du temps), mais plutôt de vous fournir des astuces rapides, et des réflexes qu'il est bon d'avoir.

Evidement, ces règles ne suffisent pas à rendre votre application invulnérable, c'est pour cela que j'ai décidé d'écrire ces articles en plusieurs parties.

C'est parti !

## Sommaire:

- [1. Les principes de base](#)
  - [2. Adopter le principe des moindres privilèges](#)
  - [3. Ne jamais faire confiance aux données entrées par l'utilisateur](#)
    - [3.1. Les données utilisateur](#)
      - [3.1.1. Validate\(\)](#)
      - [3.1.2. Les Expression régulières](#)
    - [3.2. Les données d'entrée autres que celles de l'utilisateur](#)
  - [4. L'obfuscation n'est pas l'unique protection](#)
  - [5. Réduire la surface d'attaque](#)
    - [5.1. Attaques DoS](#)
    - [5.2. Attaques basées sur les fichiers ou répertoires](#)
    - [5.3. Injections SQL](#)
    - [5.4. Cross-Site scripting](#)
  - [6. Surveiller le comportement de votre application en cas de « plantage »](#)
  - [7. Si vous n'utilisez pas une fonctionnalité, désactivez la](#)
- [Conclusion](#)
- [Annexe : Glossaire](#)

## 1. Les principes de base

Il existe un certain nombre de principes de base à toute application sécurisée, dont :

- adopter le principe des moindres privilèges
- ne jamais faire confiance aux données entrées par l'utilisateur
- ne pas penser que l'obfuscation est la seule action à mener en matière de sécurité
- partir du principe que tout système externe est insécure
- réduire la surface d'attaque
- surveiller le comportement de votre application en cas de « plantage »
- si vous n'utilisez pas une fonctionnalité, désactivez la

Sécuriser une application ne se résume pas à empêcher des intrusions. Dans la sécurité il est aussi question de robustesse de l'application par exemple.

Je vais bien évidemment expliquer chacun de ces points dans les parties qui vont suivre. ([Un glossaire des termes utilisés dans cet article est disponible à la fin](#))

## 2. Adopter le principe des moindres privilèges

Cela consiste à n'autoriser l'application ou les utilisateurs à ne faire que ce qu'ils ont le droit de faire et rien d'autre. Pour les utilisateurs, il s'agira de mettre en place un système d'autorisation (que j'expliquerai dans un futur article), pour une application, il faut limiter les pouvoirs donnés au compte Windows utilisé par l'application.

Ce qui veut dire que dans le cas d'une application Winform, les utilisateurs ne devraient pas avoir besoin d'être authentifié sur la machine en tant qu'administrateur afin d'utiliser l'application. Dans le cas d'une application Web, le compte utilisé (ASPNET en général) ne doit pas avoir des droits plus élevés que nécessaire. Quand vous développez votre application, testez la dans un environnement non administrateur, à la limite accordez des privilèges à certains dossiers si besoin est.

Ce principe évite d'une part des manipulations abusives (telles que supprimer des données nécessaires à l'application par exemple), mais surtout en cas de vulnérabilité de l'application : si l'on pénètre l'application on peut obtenir les droits du compte Windows associé. Moins ce droit est élevé et moins de destruction ou d'actions sont possibles pour le pirate.

### 3. Ne jamais faire confiance aux données entrées par l'utilisateur

Vous le savez : si vous ne vérifiez pas les données en entrée, votre application va très rapidement crasher. Cela rentre aussi dans le cadre de la sécurité. Vous devez donc méticuleusement analyser les données entrées et les vérifier avant de les utiliser afin qu'elles ne créent pas de dommages à votre application. En plus de vérifier si les données sont correctes, il faut aussi vérifier qu'elles ne soient pas nocives.

Premièrement, il faut identifier les données en entrées. Cela peut être :

- les données entrées par l'utilisateur
- les données stockées localement (et donc susceptibles d'être modifiés)
- les données stockées dans l'url

#### 3.1 Les données utilisateur

Je ferai référence indifféremment aux techniques Winform et WebForm dans cet article. Afin de vous y retrouver, je noterai **Wb** pour les WebForms et **Wf** pour les Winforms.

.Net fournit un nombre d'outils qui vous permet de vérifier le contenu des données :

- la propriété « MaxLength » qui vous permet de limiter la taille des données saisies dans les textbox par exemple (Wb/Wf)
- le RequiredFieldValidator qui est un composant qui vous permet de vérifier que le champ X a été renseigné (n'est pas vide) (Wb)
- le CompareValidator qui vous permet de comparer la valeur entrée avec une autre valeur fixée par vous (Wb)
- le RangeValidator qui permet de vérifier que la donnée est entre 2 bornes, pour un nombre par exemple, ou une chaîne de caractère voire une date (Wb)
- Le RegularExpressionValidator qui permet des vérifications très puissantes en utilisant les expressions régulières. (Wb)
- Le CustomValidator qui vous permet de créer votre propre logique de validation (Wb)
- Le ValidationSummary qui est un contrôle qui vous permet de résumer toutes les erreurs trouvées sur la même page web (Wb)
- Le namespace System.Text.RegularExpressions (Wf)

La liste ci-dessus ne suffit pas à protéger votre application. Tout d'abord, il s'agit en général de validations coté client, et si le navigateur (dans le cas d'une application Web) désactive le script coté client, les contrôles ne seront pas fait. Il faut donc vérifier les données coté Client ET Serveur dans le cas d'une application Web.

##### 3.1.1 Validate()

Par ailleurs la validation des contrôles (dans le cas de Validators pour les pages Web) n'est exécutée qu'après le Page\_Load. Dans le cas où vous faites référence à vos contrôles dans la page, si ceux-ci ne contiennent pas des données valides, c'est le plantage. Je vous conseille donc d'utiliser la méthode Validate() afin de vous en assurer, de la façon suivante :

## Visual Basic .NET

```
1: Private Sub Page_Load(ByVal sender As System.Object,  
ByVal e As System.EventArgs)  
Handles MyBase.Load  
2:     If Me.IsPostBack Then  
3:         Me.Page.Validate()  
4:         If Me.Page.IsValid Then  
5:             'utiliser votre controle  
6:             Dim strToto = txtbToto.Text  
7:         Else  
8:             'avertir l'utilisateur que la donnée n'est  
pas valide  
9:         End If  
10:    Else  
11:        'page affichée pour la première fois, pas de  
données à valider  
12:    End If  
13: End Sub
```

## C#

```
1: private void Page_Load(object sender, EventArgs  
e)  
2:     {  
3:         if(IsPostBack)  
4:         {  
5:             this.Validate();  
6:             if(this.IsValid==true)  
7:             {  
8:                 //utiliser votre controle  
9:                 string strToto = txtbToto.Text;  
10:            }  
11:        else  
12:        {  
13:            //avertir l'utilisateur que la donnée n'est  
pas valide  
14:        }  
15:    }  
16:    else  
17:    {  
18:        //page affichée pour la première fois, pas de  
données à valider  
19:    }  
20:    }
```

Il est bon de savoir que par défaut la validation ne s'effectue pas automatiquement et qu'il faut forcer la validation par l'appel de **Page.Validate()**

### 3.1.2 Les Expression régulières

Pour les expressions régulières, il est souvent difficile des les élaborer soit même à moins de n'avoir étudié la question de près, je vous conseille donc le site web <http://regexlib.com/> qui vous donne un bon nombre de patterns tout faits et vous évite de perdre du temps, il faut parfois les adapter.

Pour les pages web, il suffit d'utiliser le contrôle RegularExpressionValidator et d'entrer l'expression régulière, le message d'erreur, et le nom du contrôle à valider comme indiqué dans l'image suivante :



Pour les applications Winform, il vous faut utiliser le namespace System.Text.RegularExpressions de la façon suivante :

#### Visual Basic .NET

```

1:           'vérifier que la donnée est un entier ou un nombre
décimal avec ou sans exposant
2:           If Not Regex.IsMatch(txtbToto.Text,
"^[+-]?([0-9]*\.\?[0-9]+|[0-9]+\.\?[0-9]*) ([eE][+-]?[0-9]+)?$") Then
3:           'avertir l'utilisateur que la donnée n'est pas
valide
4:           Else
5:           'utiliser la donnée
6:           End If

```

#### C#

```

1:           //vérifier que la donnée est un entier ou un nombre
décimal avec ou sans exposant
2:           if (!Regex.IsMatch(txtbToto.Text,
"^[+-]?([0-9]*\.\?[0-9]+|[0-9]+\.\?[0-9]*) ([eE][+-]?[0-9]+)?$"))
3:           {
4:           //avertir l'utilisateur que la donnée n'est pas
valide
5:           }
6:           else
7:           {

```

```
8:         //utiliser la donnée
9:     }
```

### 3.2 Les données d'entrée autres que celles de l'utilisateur

Les applications Web peuvent avoir des entrées de données de sources variées. Vous devez donc ajouter plus de fonctions de validations. Les types de données en entrée non utilisateur sont :

- les paramètres en QueryString (passés dans l'url)
- les cookies
- les variables Serveur
- les fichiers lus et utilisés (valable aussi pour les application Winform)

Le minimum est d'utiliser la méthode `Server.HtmlEncode` pour vérifier qu'aucun contenu ne sera exécuté en tant qu'HTML.

Ainsi, au lieu de faire :

```
1: lblBonjour.Text = "Salut " + Request.QueryString("Utilisateur")
```

Vous devriez faire :

```
1: lblBonjour.Text = "Salut " +
Server.HtmlEncode(Request.QueryString("Utilisateur"))
```

Pour la lecture de fichier par exemple, un certain nombre de vérifications sont à faire. Imaginons que vous récupérez des données depuis un fichier `voiture.txt`. Vous devez vérifier que votre code fonctionne dans le cas où une ligne est vide, que le fichier n'existe pas, que le type de données n'est pas du type attendu (un entier par exemple), que la valeur donnée est réaliste (par exemple -150 km pour le kilométrage d'une voiture n'est pas réaliste) etc...

## 4. L'obfuscation n'est pas l'unique protection

L'obfuscation consiste à transformer une section de code ou un programme, de manière à le rendre totalement incompréhensible à un lecteur humain, même aidé d'outils informatiques. Il existe des outils pour « obscurcir » votre code source.

Cela rends plus difficile le reverse-engineering. Quand on utilise cette technique, les nom des variables et méthodes sont transformés pour donner un code moins facile à lire, et donc dans le cas d'une décompilation permet de rendre difficile la tâche pour la personne qui tente de comprendre ce que le code fait. Visual .Net fournit un utilitaire qui s'appelle `Dotfuscator` par exemple. Attention ! L'obfuscation aide à sécuriser votre application (comme chacune des étapes décrites dans cet article) mais ne fait pas tout. Une fois exécuté, votre code (obscur ou pas) fera les même choses, cela ne change rien aux trous de sécurité.

## 5. Réduire la surface d'attaque

La surface d'attaque définit la vulnérabilité d'une application, d'un serveur. Plus la surface d'attaque est réduite moins il y a de vulnérabilité.

Il existe beaucoup d'attaques. Les plus connues sont le DoS (Denial of Service), les attaques basées sur les fichiers ou répertoires, les injections SQL, les Cross-Site scripting... Voici comment comprendre en quoi elles résident et comment s'en

protéger :

## 5.1 Attaques DoS

Le but d'une attaque DoS est de faire en sorte que l'application ou le serveur ne réponde plus, ne soit plus opérationnel. Elle ne détruit pas vos données mais fait en sorte que les utilisateurs ne puissent plus utiliser votre application. Les attaques DoS sont en général lancées contre des applications qui sont sur le réseau. Les applications Web y sont donc plus vulnérables que les applications Windows.

Il existe plusieurs façon d'obtenir un DoS : en faisant crasher l'application, en faisant monter la mémoire utilisée, en privant l'application du processeur (en inondant le CPU de calculs par exemple), en privant l'application de réseau ou de ressource (taille du disque dur), bref en privant l'application des ressources indispensables à son bon fonctionnement.

Comment s'en protéger ? Les attaques DoS sont sûrement parmi les attaques les plus difficilement parables. Il s'agit de consolider le code avant tout.

- bien définir la gestion d'erreurs
- tester votre code (Unit Testing)
- limiter la taille des données en entrée, et éviter les entrées répétitives
- penser à considérer la stockage des infos sur fichier ou base de données afin de préserver la mémoire (c'est un choix qui n'est pas évident entre sécurité et performance)
- bien nettoyer les objets une fois qu'ils ne sont plus utilisés
- éviter les boucles éternelles (conseil évident :))

Un exemple concret :

Si vous utilisez une collection d'objets avec un `.Add()`, il vaut mieux surveiller les doublons par exemple, dans le cas où un attaquant exécute en boucle le code, il pourrait ajouter à l'infini et faire augmenter de façon impressionnante la mémoire utilisée par votre collection.

## 5.2 Attaques basées sur les fichiers ou répertoires

Imaginons que vous ayez une fonction qui ouvre un fichier en fonction d'un paramètre « `CheminDuFichier` ». Si un attaquant arrive à utiliser cette méthode en passant en paramètre « `..\..\..\..\Windows\notepad.exe` », en fonction des opérations que vous effectuez, on pourrait écrire par-dessus le fichier, le supprimer ou le remplacer. Il faut donc vérifier le chemin passé en paramètre. Une possibilité est d'écrire une méthode qui compare le chemin donné avec le chemin de votre application :

### Visual Basic .NET

```
1:     Private Function CheminValide(ByVal strChemin As
String) As Boolean
2:         Dim strCheminCanonique As String =
Path.GetFullPath(strChemin)
3:         If
Path.GetDirectoryName(strCheminCanonique).ToLower =
Application.StartupPath.ToLower Then
4:             Return True
5:         Else
6:             Return False
7:         End If
8:     End Function
```

### C#

```
1:     private Boolean CheminValide(string strChemin)
2:     {
3:         string strCheminCanonique =
```

```

Path.GetFullPath(strChemin);
4:
if(Path.GetDirectoryName(strCheminCanonique).ToLower ==
Application.StartupPath.ToLower)
5:     {
6:         return true;
7:     }
8:     else
9:     {
10:        return false;
11:    }
12:    }

```

A vous d'adapter le code.

### 5.3 Injections SQL

Supposons que vous exécutiez une requête en fonction d'une entrée rentrée par l'utilisateur (sur un textbox nommé txtbNom) du style :

```
" SELECT * FROM Users WHERE LastName = '" + txtbNom + "'"
```

Si une personne entre dans le controle txtbNom la chaîne suivante :

```
"Hugo' DELETE FROM Users WHERE LastName = 'Baudelaire' "
```

Alors cela supprimera les utilisateurs s'appelant Baudelaire de la table. L'attaquant profite ainsi de la richesse du langage SQL pour détruire des données notamment s'il s'agit d'une base SQL Serveur.

Cela fonctionne aussi sur des select, notamment dans le processus d'authentification, avec un peu d'ingéniosité on pourrait récupérer toute la liste des utilisateurs.

Pire, on pourrait même exécuter des procédures stockées du genre « exec master..xp\_cmdshell 'IISRESET/STOP' »

Vous êtes donc vulnérable dès que vous effectuez une requête SQL en y incorporant des données « étrangères » telles que un utilisateur ou toute autre donnée « non connue », dans les deux cas suivants :

- requête construite dans le code et qui utilise une donnée utilisateur sans vérification : maCommande = new SqlDataAdapter("SELECT \* FROM Users WHERE UserName = '" + txtbLogin.Text + "'",maConnexion)
- procédure stockée avec une unique chaîne pour l'appeler, et qui utilise une donnée utilisateur non filtrée : maCommande = new SqlDataAdapter("SPLogin '" + txtbLogin.Text + "'",maConnexion)

Pour s'en prémunir, il existe différentes méthodes :

- Utiliser une collection de paramètres quand vous construisez votre requête SQL

### Visual Basic .NET

```

1:         Dim maCommande As SqlDataAdapter = New
SqlDataAdapter("SELECT * FROM Users
WHERE UserName = @UserName", maConnexion)
2:         Dim param As SqlParameter =
maCommande.SelectCommand.Parameters.Add("@UserName",
SqlDbType.VarChar, 15)
3:         param.Value = txtbLogin.Text

```

## C#

```
1:      SqlDataAdapter maCommande = new SqlDataAdapter("SELECT * FROM
Users
WHERE UserName = @UserName",maConnexion);
2:      SqlParameter param =
maCommande.SelectCommand.Parameters.Add("@UserName",SqlDbType.VarChar,15);
3:      param.Value = txtbLogin.Text;
```

Si vous utilisez ce système, quel que soit le contenu de l'entrée utilisateur, tout sera exécuté de façon littérale. Un autre avantage d'utiliser cette méthode est que vous pouvez forcer le type et la taille de la donnée ce qui fait une bonne vérification en plus pour les données utilisateur.

- Filtrer les données en entrée en doublant chaque Quote (') rencontré

## Visual Basic .NET

```
1:      Private Function ChaineSQLSecure(ByVal entreeSQL As
String) As String
2:          Return entreeSQL.Replace("'", "'");
3:      End Function
4:      ...
5:      Dim maCommande As SqlDataAdapter = New
SqlDataAdapter("SELECT * FROM Users
WHERE UserName = " + ChaineSQLSecure(txtbLogin) + "'",
maConnexion)
```

## C#

```
1:      private string ChaineSQLSecure(string entreeSQL)
2:      {
3:          return entreeSQL.Replace("'", "'");
4:      }
5:      ...
6:      SqlDataAdapter maCommande = new SqlDataAdapter("SELECT
* FROM Users
WHERE UserName = '" +
7:          ChaineSQLSecure(txtbLogin) + "'",maConnexion);
```

- Si vous utilisez des clauses LIKE il faut étendre l'exemple donné dessus de la façon suivante :

## Visual Basic .NET

```
1:      Private Function ChaineSQLSecure(ByVal entreeSQL As
String) As String
2:          Dim strClean As String = entreeSQL
3:          strClean = strClean.Replace("'", "'")
4:          strClean = strClean.Replace("[", "[ ]")
5:          strClean = strClean.Replace("%", "[%]")
6:          strClean = strClean.Replace("_", "[_]" )
```

```
7:         Return strClean
8:     End Function
```

## C#

```
1:     private string ChainesQLSecure(string entreeSQL)
2:     {
3:         string strClean = entreeSQL;
4:         strClean = strClean.Replace("'", "");
5:         strClean = strClean.Replace("[", "[[]]");
6:         strClean = strClean.Replace("%", "[%]");
7:         strClean = strClean.Replace("_", "[_]");
8:         return strClean;
9:     }
```

Il existe quelques autres astuces qui vous permettent de renforcer la protection anti-Injection SQL :

- limiter la taille des données en entrée. Si le nom d'utilisateur (dans mon exemple mais on pourrait parler d'autres types de données) ne fait que 15 caractères, alors limiter en entrée à 15 caractères.
- Exécuter le code SQL avec le principe des moindres privilèges, cela réduit de façon significative les dommages potentiels qui peuvent être faits. Par exemple en cas d'injection d'une commande « DROP » si le compte SQL utilisé par l'application n'a pas les droits, cette commande ne pourra être exécutée. C'est encore une bonne raison de ne jamais utiliser le compte SQL « sa » pour vos applications
- Quand une erreur de type SQL survient, empêcher la remontée d'erreur jusqu'à l'utilisateur : cela pourrait fournir des informations qui pourraient aider un pirate, il faut toujours filtrer ce genre d'informations et donner un message d'erreur simple sans détail

## 5.4 Cross-Site scripting

Cela affecte les pages web. Si vous autorisez l'utilisation de code HTML sans faire de vérification les impacts peuvent être terribles. Imaginez une page Web avec un textbox pour entrer votre nom d'utilisateur. Si quelqu'un entre le texte suivant :

```
1: <SCRIPT LANGUAGE="Javascript"> alert("J'exécute un script")</SCRIPT>
```

Cela vous donne une idée des possibilités, même si le code donné en exemple ici ne fera rien de mal. Normalement, avec Visual Studio 2003, vous devriez avoir un message d'erreur vous disant qu'une donnée potentiellement dangereuse a été détectée.

Par défaut vous êtes protégés contre ces attaques. Vous pouvez désactiver cette option avec ValidateRequest mais ce n'est pas recommandé.

Première ligne de la page aspx :

```
<%@ Page language="c#" Codebehind="MaPage.aspx.cs"
AutoEventWireup="false" Inherits=" MonApp.MaPage" ValidateRequest="false"
%>
```

L'exemple donné plus haut n'en est qu'un parmi tant d'autres, notamment la redirection d'informations (du type login + mot de passe) vers un autre site web.

Une fois de plus, il est recommandé d'utiliser Server.HtmlEncode et/ou Server.UrlEncode.

Pour plus d'informations je vous conseille les sites web suivants :

<http://www.microsoft.com/technet/security/news/crsstqs.msp>  
<http://www.microsoft.com/technet/security/news/csovery.msp>  
<http://support.microsoft.com/default.aspx?scid=kb:en-us:252985>

## 6. Surveiller le comportement de votre application en cas de « plantage »

Comme je viens de l'expliquer dans les Injection SQL, moins l'utilisateur en sait sur votre application et sa base de données mieux cela vaudra. Il faut donc filtrer les erreurs afin de donner le strict minimum à l'utilisateur en cas de plantage. Dans le cas d'une requête SQL, si les remontées d'erreur ne sont pas correctement gérées, il est fréquent de se retrouver avec des informations sur la base de données, la table concernée par la requête voire sur le compte utilisé.

## 7. Si vous n'utilisez pas une fonctionnalité, désactivez la

Vous pouvez renforcer la sécurité en désactivant des modules ou des composants dont vous n'avez pas besoin. Par exemple, si une application n'utilise pas « l'output caching » alors vous devriez désactiver le module ASP.Net d'output cache. Si une future faille de sécurité est trouvée dans ce module, votre application ne sera pas menacée.

## Conclusion

Voici donc une liste des actions classiques à mener dans vos applications .Net. A elle seules, elles ne vous protégeront pas entièrement, la sécurité est un domaine très large et complexe, mais ces astuces vous prémuniront des principales attaques connues.

Dans cet article, je n'ai pas mentionné le CAS (Code Access Security), qui est un élément important en matière de sécurité .Net. C'est un sujet qui mériterait plus d'une partie dans cet article. Aussi pour le moment je vous invite à aller jeter un coup d'œil aux liens suivants :

<http://www.15seconds.com/issue/040121.htm>

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconcodeaccesssecurity.asp>

[http://www.codeproject.com/dotnet/UB\\_CAS\\_NET.asp](http://www.codeproject.com/dotnet/UB_CAS_NET.asp)

## Annexe : Glossaire

Authentification	Mécanisme qui permet d'identifier de façon sécurisée les utilisateurs de votre application
Autorisation	Mécanisme qui permet d'identifier les actions possibles de chaque utilisateur authentifié
Obfuscation	Opération qui consiste à transformer une section de code ou un programme, de manière à le rendre totalement incompréhensible à un lecteur humain, même aidé d'outils informatiques.
Surface d'attaque	Définit la vulnérabilité d'une application, d'un serveur. Plus la surface d'attaque est réduite moins il y a de vulnérabilité.

Expressions  
régulières

Les expressions régulières (ou regex) peuvent être présentées comme un système très élaboré et très puissant, permettant de : trouver (retrouver ==> coupler ==> assortir) des motifs (pattern ==> profils ==> masques ==> structures) et de traiter (récupérer ==> extraire ==> remplacer) des éléments à l'intérieur d'une chaîne de caractères

**Voir aussi dans la même série :**

- Partie 1 : Sécurisez votre Serveur Windows pour vos applications ASP.Net
- Partie 2 : Les règles de base



L'ensemble ou partie de ce document ainsi que le code mis à disposition, ne peut être diffusé ailleurs sans autorisation préalable

[elise.dupont@europe.com](mailto:elise.dupont@europe.com) - [www.dotnet-tech.com](http://www.dotnet-tech.com) - 2004