

Obfuscation : protection du code source contre le reverse engineering

par [Hugo Etiévant](#)

Date de publication : 08/10/2006

Dernière mise à jour : 14/10/2006

Ce document présente les techniques et les limites de l'obfuscation du code source à des fins de protection contre le reverse engineering. On parle aussi d'assombrissement destiné à rendre le code impénétrable.

I - Introduction

- A - Les dangers du reverse engineering
- B - Les techniques de protection
- C - Pourquoi l'obfuscation ?

II - Les techniques de l'obfuscation

A - Le style

- 1 - Transformation des identifiants
- 2 - Suppression des commentaires
- 3 - Suppression du style de codage
- 4 - Suppression des API inutilisées
- 5 - Suppression des instructions de débogage et de compilation

B - Les données

- 1 - Cryptage des chaînes
- 2 - Transtypage
- 3 - Réordonnement des tableaux
- 4 - Modification de la visibilité des variables
- 5 - La mare aux variables

C - La structure de l'application

- 1 - Le modèle de classes
- 2 - Les structures de contrôle

III - L'efficacité de l'obfuscation

IV - Les limites de l'obfuscation

V - Conclusion

VI - Voir aussi

VI - Remerciements

I - Introduction

La sécurité et la propriété intellectuelle sont deux priorités des éditeurs de logiciels. La sécurité d'une application est nécessaire pour sa large diffusion. La propriété intellectuelle garantit la pérennité des recettes et justifie les investissements réalisés pour son développement. Mais quid du secret industriel ?

A - Les dangers du reverse engineering

Le **reverse engineering** est une technique permettant de reconstituer le code source d'une application à partir de sa forme compilée telle que livrée à ses clients par un éditeur. La possession du code source permet de connaître le fonctionnement précis d'une application.

Un concurrent peut grâce à cette technique connaître les **algorithmes** utilisés par son concurrent et lui voler ses secrets. Outre les algorithmes, les identifiants de connexion à des ressources (bases de données, annuaires...) sont là aussi accessibles, d'où le risque de vol de mot de passe et d'accès frauduleux à des ressources protégées.

B - Les techniques de protection

Face aux risques liés au reverse engineering quelles techniques permettent de cacher le code source d'une application ?

Il en existe quatre :

- l'**obfuscation** transforme le code source avant **compilation** de manière à le rendre illisible pour l'être humain
- le **chiffrement** assure la **confidentialité** totale du code source tant que l'algorithme de **chiffrement** n'a pas été cassé et que la clé n'a pu être trouvée par **force brute**
- l'**exécution de code distant** permet de ne livrer aux clients qu'une partie de l'application, les portions sensibles sont conservées sur un serveur distant protégé sur lequel elles s'exécutent
- le **code natif protégé** est un code compilé pour une architecture matérielle très spécifique, rendant difficile l'utilisation d'un **décompilateur** adapté

Il sera discuté dans cet article de la technique par **obfuscation**.

C - Pourquoi l'obfuscation ?

La technique de l'obfuscation a de nombreux atouts, elle est :

- la moins coûteuse en terme financier,
- la plus facile à mettre en place en terme d'architecture,
- totalement transparente pour l'utilisateur,
- totalement transparente pour les développeurs.

Une application obfusquée n'a rien de différent d'une application normale sauf que son code compilé ne permet plus de retrouver facilement le code source originel même avec un décompilateur évolué.

Il existe évidemment des outils qui essaient de réaliser l'opération inverse de l'obfuscation, c'est ainsi que la guerre des obfusateurs / désobfusateurs se mène actuellement. Le choix d'un outil complexe et à jour permet de conserver le secret de son code source durant un laps de temps suffisant pour un éditeur pour garder l'avantage sur ses concurrents malveillants.

II - Les techniques de l'obfuscation

Nous allons voir dans cette partie les principales techniques d'obfuscation du code. Il existe une panoplie impressionnante de techniques, aussi, cet article n'a pas pour but d'en dresser une liste exhaustive, mais de présenter les principaux axes de travail des créateurs de désobfuscatteurs.

Des nouvelles techniques d'obfuscation sont régulièrement inventées, avec pour corollaire, l'ajout des contres mesures adéquates dans les désobfuscatteurs...

A - Le style

L'aspect général du code source d'une application est obfusqué de ces manières ci :

- renommage de tous les identifiants
- suppression des commentaires
- élagage des déclarations d'API non utilisées
- suppression des règles de style

1 - Transformation des identifiants

Historiquement, c'est la première technique à avoir été utilisée en obfuscation. Elle consiste à renommer tous les identifiants du code : variables, constantes, classes, méthodes, attributs. Les étudiants apprennent en tout premier lieu à nommer de façon explicite les variables utilisées afin de faciliter la compréhension du code. Or obfusquer, c'est rendre incompréhensible le code ! Il faut donc donner des noms abscons à ces variables et autres entités du programme.

Le renommage des identifiants est appelé *refactoring*, ce n'est pas une tâche évidente pour un outil de refactoriser le code, car tous les modules, même lointains, susceptibles de référencer telle ou telle fonction ou variable doivent être modifiés afin d'utiliser le nouvel identifiant. Ainsi, pour chaque entité refactorisée, le source entier doit être analysé et modifié si nécessaire.

Mais quel nouvel identifiant donner ? Il existe plusieurs méthodes possibles :

- **méthode aléatoire** : une chaîne de caractère aléatoire unique est générée pour chaque identifiant à obfusquer
- **méthode *Overload Induction*** : la chaîne la plus simple est donnée à chaque identifiant, par exemple : a(), puis b(), ... aa()... pour les fonctions.

Cette méthode a pour particularité de tenter de donner à un maximum d'entité le même identifiant en exploitant leur portée et la surcharge. Elle a plusieurs atouts : la réduction de la taille du code, la réduction de l'encombrement de la taille du tas d'appel lors de l'exécution du code.

- **méthode d'invisibilité** : une chaîne de caractère comportant des caractères spéciaux interdits par le langage et les principaux décompilateurs est générée pour chaque identifiant

Exemple de code Java non obfusqué

```
public synchronized void put(int key, Employee value) {
    Integer I = new Integer(key);
    super.put(I, (Object) value);
}
```

Exemple obfusqué par la méthode aléatoire

```
public synchronized void yrwla35rn3z22jd0sci9(int sbhc8wduotn7gkbr8pq6, k0j9y980ekqci18t09ju
78nrx59777f4io0qpg7t) {
    Integer f841593p5rh12zf88285 = new Integer(sbhc8wduotn7gkbr8pq6);
    super.yrwla35rn3z22jd0sci9(f841593p5rh12zf88285, (Object) 78nrx59777f4io0qpg7t);
}
```

Exemple obfusqué par la méthode Overload Induction

```
public synchronized void a(int a, b c) {
    Integer d = new Integer(a);
    super.a(d, (Object) c);
}
```

Exemple obfusqué par la méthode d'invisibilité

```
public synchronized void #~a(int @b, f# a~) {
    Integer #~b = new Integer(@b);
    super.#~a(#~b, (Object) a~);
}
```

2 - Suppression des commentaires

Tout plan qualité d'un projet de développement logiciel impose l'usage des commentaires dans le code source et spécifient dans le détail la manière de les écrire. Par exemple, en Java, il existe une norme précise : la JavaDoc. Un programme bien commenté contient plus de 50% de commentaires. Les commentaires sont un instrument indispensable pour la **maintenance** car ils donnent des indications explicites essentielles pour la compréhension du code.

L'obfuscation suppose la suppression systématique de tous commentaires.

Exemple de code Java commenté selon la norme JavaDoc

```
/**
 * Recherche d'une agence bancaire auprès de la banque
 * en fonction de son numéro.
 * @param number numéro de l'agence
 * @return référence vers une agence
 * @exception pour tout problème
 * @since 2
 */
public Branch findBranch(String number) throws ProblemException {
    Branch b = null;
    try {
        // Extraction de la référence vers l'agence
        if ( (b = (Branch) (entries.get(number))) != null ) {
            System.out.println("the branch " + number + " found");
        }
        // levée d'une exception si non trouvée
        else {
            throw new ProblemException("BankImpl.findBranch", "no branch found !");
        }
    } catch (ProblemException e) {
        System.out.println("BankImpl.findBranch : error in finding - " + e.getMessage());
    }
    return b;
}
```

Exemple de code Java décommenté

```
public Branch findBranch(String number) throws ProblemException {
    Branch b = null;
```

Exemple de code Java décommenté

```

try {
    if ( (b = (Branch) (entries.get(number))) != null ) {
        System.out.println("the branch " + number + " found");
    }
    else {
        throw new ProblemException("BankImpl.findBranch", "no branch found !");
    }
} catch(ProblemException e) {
    System.out.println("BankImpl.findBranch : error in finding - " + e.getMessage());
}

return b;
}

```

3 - Suppression du style de codage

Les conventions de codage d'une application sont une autre spécification du plan qualité de tout bon projet de développement soucieux de la gestion de la qualité dont la maintenabilité est un critère.

Ces [conventions de codage](#), appelées également *style de codage* sont l'ensemble des règles strictes auxquelles doivent se plier les développeurs d'une application afin de s'assurer de l'uniformité du code source. Ce style permet de garantir une lisibilité assurée pour une maintenance efficace.

Un style décrit la manière d'[indenter](#) le code, d'écrire les délimiteurs de blocs de code, de nommer les variables...

Exemple de code Java respectant une convention de style

```

package bankServices;

import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;

import java.util.HashMap;

public class BankImpl extends BankPOA {
    private HashMap entries;

    public void unregisterBranch(String number) throws ProblemException {
        try {
            Branch b = (Branch) (entries.get(number));

            if(b != null) {
                entries.remove(number);
                System.out.println("the branch " + number + " is deleted");
                return;
            } else {
                throw new ProblemException("BankImpl.unregisterBranch","no branch
reference found !");
            }
        } catch(ProblemException e) {
            System.out.println("BankImpl.unregisterBranch : error in unregistering - " +
e.getMessage());
        }
    }
}

```

Exemple de code Java dont le style a été détruit

```

package bankServices;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;

```

Exemple de code Java dont le style a été détruit

```
import org.omg.PortableServer.POA;
import java.util.HashMap;
public class BankImpl extends BankPOA{private HashMap entries;public void unregisterBranch(String
number) throws ProblemException{
try{Branch
b=(Branch)(entries.get(number));if(b!=null){entries.remove(number);System.out.println("the branch
"+number+" is deleted");
return;}else{throw new ProblemException("BankImpl.unregisterBranch","no branch reference found!");
}}catch(ProblemException e){System.out.println("BankImpl.unregisterBranch : error in unregistering -
" + e.getMessage());}}
```

4 - Suppression des API inutilisées

Les noms des bibliothèques importées (Java, PHP) ou incluses (C, C++) au code sont susceptibles de fournir des indications sur les algorithmes employés dans le source. Aussi, cacher celles qui ne sont pas utilisées permet de réduire le nombre d'indices à disposition des indiscrets.

Cette fonctionnalité est également présente dans les IDE sous la forme d'une fonctionnalité appelée (clic droit, puis :) "Source > Organize imports" dans Eclipse, par exemple.

Exemple de code Java dont les imports sont nombreux

```
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;
import java.util.HashMap;
```

Exemple de code Java dont les imports ont été réduits

```
import org.omg.CORBA.ORB;
import org.omg.PortableServer.POA;
import java.util.HashMap;
```

5 - Suppression des instructions de débogage et de compilation

Les instructions de compilation peuvent également être retirées du code dans certains cas. Ceci est spécifique au langages pour lesquels l'obfuscateur travaille non pas sur le code source mais sur un **bytecode** (Java), un code managé (.NET), ou tout code précompilé qui n'a plus besoin de telles instructions.

B - Les données

Les données d'initialisation de variables, les valeurs de constantes et le contenu des tableaux donnent eux aussi des informations utiles aux espions. Ainsi les identifiants de connexions à des bases de données distantes peuvent être exploités.

1 - Cryptage des chaînes

Le cryptage des chaînes de caractères permet de cacher leur valeur. La clé de décryptage et la fonction de décryptage seront noyées dans le code.

Cryptage d'une adresse email en JavaScript

```
<script type="text/javascript">
<!--
    Ch=new Array(4);
    Res=new Array(4);
    Ch[0]='le_club_des_developpeur';
    Ch[1]='ÛÆËÏää';
```

Cryptage d'une adresse email en JavaScript

```

Ch[2]='-xÀÇÍØÖÉÓ';
Ch[3]='ÐÈÖÈðÀÖÏËß;ÃÖÓ';
for(y=1;y<4;y++){
  Res[y]="";
  for(x=0;x<Ch[y].length;x++){
    Res[y]+=String.fromCharCode(Ch[y].charCodeAt(x)-Ch[0].charCodeAt(x));
  }
  document.write('<a
href="'+Res[1]+':webmaster'+Res[2]+'-'+Res[3]+'>webmaster'+Res[2]+'-'+Res[3]+'</a>');
//-->
</script>

```

2 - Transtypage

La transformation de types, l'utilisation d'un codage particulier permet d'induire en erreur les désobfuscateurs.

Par exemple, utiliser en Java l'objet Integer au lieu du type primitif augmente la taille et la complexité du code à analyser. Ou encore, remplacer une variable par une combinaison d'autres variables participe à la complexification du code.

3 - Réordonnement des tableaux

Certains tableaux de valeurs renseignent sur les traitements dans lesquels ils interviennent. C'est le cas des listes de pays, de taux de conversions monétaires, etc.

Le réordonnement aléatoire des tableaux supprime l'indice à la compréhension de son contenu que fournissait le tri (alphabétique ou autre).

Attention cependant, à rajouter les fonctions de tri des tableaux lorsque leur manipulation pose pour prédicat une forme de tri (exemple de la recherche dichotomique).

4 - Modification de la visibilité des variables

La visibilité des variables donne un indice sur leur usage et leur signification. Passer toutes les variables locales en globales rend la tâche d'analyse plus difficile, surtout si elles portent le même nom : une même entité sera utilisée pour différents types de traitement, rendant difficile la détermination de son rôle exact.

5 - La mare aux variables

D'autres techniques sur les variables existent : la création d'alias, les dépendances entre variables...

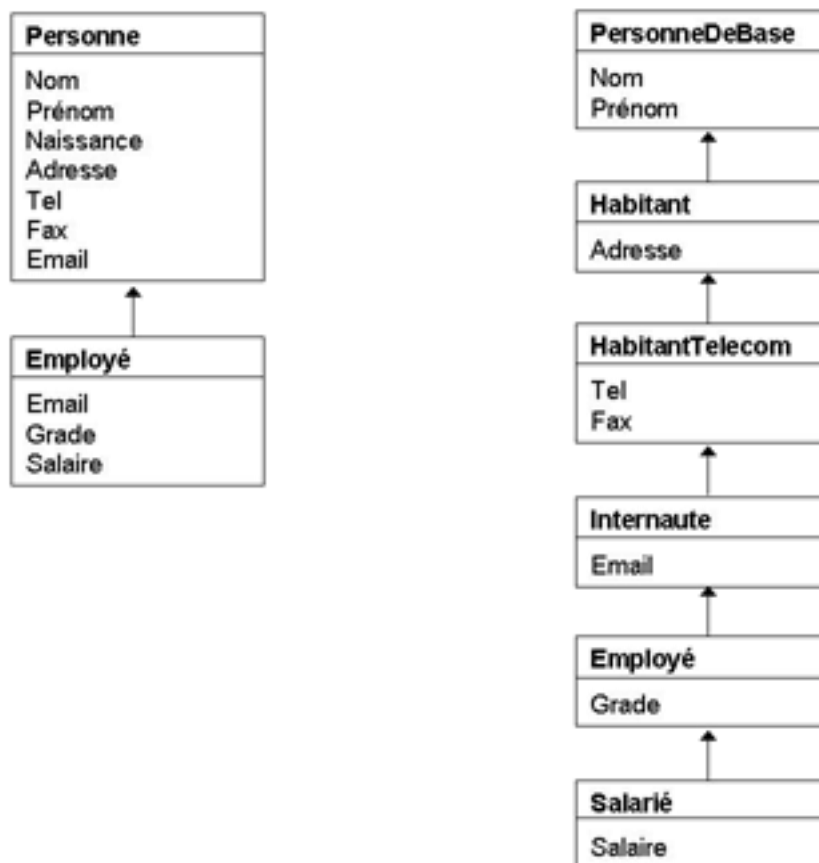
La création d'alias, c'est-à-dire de multiples variables pointant toutes vers la même valeur (merci les pointeurs) augmente considérablement le temps de calcul d'un désobfuscateur et lui cache une partie de la logique entourant les traitements de la valeur pointée.

C - La structure de l'application

C'est véritablement par la modification de la structure de l'application que l'obfuscation réussit le mieux.

1 - Le modèle de classes

Le **modèle de classes** de l'application peut être profondément remanié (scinder des classes, en regrouper d'autres) afin d'en obscurcir la logique. Alourdir le graphe d'héritage augmentera de façon prohibitive le temps d'exécution du désobfusicateur.



Grappe d'héritage alourdi

De la même manière, les **structures de données**, qui généralement respectent des **patrons de conception** sont transformés en d'autres patrons de conception moins connus. L'inconvénient de cette technique est la perte des bénéfices tirés des patrons initialement choisis.

2 - Les structures de contrôle

Un nombre quasi illimité de techniques permet de tromper l'analyse du code source :

- l'ajout de fausses structures conditionnelles dont le résultat est toujours vrai mais destiné à faire dire au désobfusicateur que le traitement qui suit est conditionnel et qu'il dépend de l'état de telle ou telle variable
- l'augmentation artificielle de la taille du code par ajout de méthodes et de classes inutiles
- l'insertion de code mort (instructions n'impactant pas les traitements ni les données de l'application), mais il en résulte un **temps de réponse** supplémentaire de l'application

Exemple de code mort dans une méthode C#

```

private void InitializeComponent()
{
    this.button_crypter = new System.Windows.Forms.Button();
    this.NeRienFaire("APXS"); // code mort
}
  
```

Exemple de code mort dans une méthode C#

```

this.button_decrypter = new System.Windows.Forms.Button();
this.FaireRienDuTout(new System.Windows.Forms.Button()); // code mort
this.button1 = new System.Windows.Forms.Button();
this.EncoreRien(50, -10); // code mort
this.SuspendLayout();
this.AutreRien(); // code mort
}

```

- l'augmentation de la quantité de tests et de structures de contrôle

Boucle à obfusquer en C

```

for(int i=1; i<=n; i++) {
    for(int j=1; j<=n; j++) {
        tab[i,j] = fct(i,j);
    }
}

```

Boucle obfusquée en C

```

for(int I=1; I<=n; I+64) {
    for(int J=1; J<=n; J+64) {
        for(int i=I; i<=min(I+63,n); i++) {
            for(int j=J; j<=min(J+63,n); j++) {
                tab[i,j] = fct(i,j);
            }
        }
    }
}

```

- la modification de l'ordre des expressions booléennes
- l'augmentation la quantité d'arguments des méthodes en leur passant des arguments fantoches
- l'augmentation du niveau d'imbrication du code par ajout de structures inutiles
- l'augmentation de la complexité des structures de données (augmentation du nombre de dimensions d'un tableau...)
- la complexification des conditions de boucle sans pour autant modifier le nombre de tours
- la permutation de code : une portion de code compilée peut être remplacée par une autre qui devra être interprété par un interpréteur fourni par l'obfuscateur, mais cette technique est coûteuse en ressources et s'oppose à la diffusion du logiciel mais fournit un haut niveau de protection
- le rajout d'opérandes redondants dans les expressions arithmétiques
- la parallélisation de code non pas pour augmenter les performances mais pour obscurcir le graphe de contrôle, l'analyse statique par les obfuscateurs des tâches parallélisées est très difficile ! Une section de code séquentielle peut être facilement parallélisée si elle ne contient pas de dépendance de données. Par exemple, en Java du code peut être parallélisé avec les threads, s'il y a dépendances de données, des synchronisations peuvent être faites.

Exemple de code Java à paralléliser

```

void methode() {
    // instruction 1
    // instruction 2
    // instruction 3
}

```

Exemple de code parallélisé en Java

```

// 1er thread
class methodeA extends Thread {
    methodeA() {
    }

    public void run() {
        // instruction 1
        // instruction 2
    }
}

```

Exemple de code parallélisé en Java

```
// 2nd thread
class methodeB extends Thread {
    methodeB() {
    }

    public void run() {
        // instruction 3
    }
}
void methodeC() {
    methodeA mA = new methodeA();
    mA.run();
    methodeB mB = new methodeB();
    mB.run();
}
```

- déplacement du code de classes en classes
- explosion de méthodes en plusieurs autres

Fonction PHP à exploser

```
function count($id) {
    $fichier=$id.".cpt";
    if(!file_exists($fichier)) {
        $fp=fopen($fichier,"w");
        fputs($fp,"0");
        fclose($fp);
    }
    $fp=fopen($fichier,"r+");
    $hits=fgets($fp,10);
    $hits++;
    fseek($fp,0);
    fputs($fp,$hits);
    fclose($fp);
}
```

Fonction PHP explosée en plusieurs autres

```
function count($id) {
    $fichier=$id.".cpt";
    create($fichier);
    write($fichier);
}
function create($fichier) {
    if(!file_exists($fichier)) {
        $fp=fopen($fichier,"w");
        fputs($fp,"0");
        fclose($fp);
    }
}
function write($fichier) {
    $fp=open($fichier,"r+");
    $hits=fgets($fp,10);
    $hits = calculate($hits);
    fseek($fp,0);
    fputs($fp,$hits);
    fclose($fp);
}
function open($fichier, $opt) {
    $fp=fopen($fichier, $opt);
    return $fp;
}
function calculate($hits) {
    return $hits++;
}
```

- fusion de portions de code distantes dans une unique méthode
- clonage de méthodes, les appels à la méthode originale étant remplacés par un appel vers l'une des méthodes clones

Exemple de méthode à cloner

```
public Graphe Ajouterlesommet(Object s) {
    if(!Recherchersommet(s)) {
        Cellgraphe c = new Cellgraphe(s);
        listcell.Insererapres(c);
    }
    return this;
}
```

Exemple de méthodes clones

```
// clone n°1
public Graphe AjouterUnSommet(Object sommet) {
    if(!Recherchersommet(sommet)) {
        Cellgraphe cell = new Cellgraphe(sommet);
        listcell.Insererapres(cell);
    }
    return this;
}
// clone n°2
public Graphe AjouterSommet(Object s) {
    if(Recherchersommet(s) != 0) {
        Cellgraphe c = new Cellgraphe(s);
        listcell.Insererapres(c);
    }
    return this;
}
// clone n°3
public Graphe Ajouterlsommet(Object obj) {
    if(!Recherchersommet(obj))
        Cellgraphe c = new Cellgraphe(obj);
    if(!Recherchersommet(obj))
        listcell.Insererapres(c);
    return this;
}
```

- relocalisation par dispersion des éléments dont la logique a voulu que le programmeur les regroupe
- etc.

III - L'efficacité de l'obfuscation

L'efficacité de l'obfuscation se mesure en effort de programmation du débogueur, en temps d'exécution et en niveaux de ressources nécessaires au débogueur pour produire un code source lisible.

Ainsi, le challenge de l'obfuscation est similaire à celui du chiffrement : il s'agit de produire un code suffisamment obfusqué pour rendre toute tentative d'analyse vaine du fait des temps de calculs incroyablement longs et de la complexité des traitements à réaliser. Mais l'évolution technologique est telle que les capacités de calculs des ordinateurs évoluent rapidement, la durée de la protection est courte.

IV - Les limites de l'obfuscation

L'obfuscation d'une application ne permet plus de réaliser les opérations réflexives sur le code générique : les [API](#) de [réflexion](#) ne fonctionneront plus car le diagramme de classe sera changé de façon aléatoire par l'obfuscateur.

Les opérations de débogage et la réalisation de traces ne pourront plus être proposées sur la version commerciale de l'application. Privant alors le support d'outils importants pour l'aide aux utilisateurs.

La sécurité par l'opacité est un mythe ! L'obfuscation protège le code source contre la piraterie intellectuelle durant un temps assez court. Mais elle ne protège pas des pirates voulant exploiter les failles de sécurité de l'application.

L'augmentation de la complexité algorithmique et la modification des structures de données augmentent le temps d'exécution. Les patrons de conceptions choisis par le développeur disparaissent et peuvent être remplacés par d'autres moins efficaces.

La qualité (algorithmique) du code source baisse considérablement et peut être un frein à la certification par des organismes tiers.

L'appel à des API externes (notamment en Java) fait par le nom ne peut PAS être obfusqué, et donnent alors des indices aux pirates.

Les désobfuscateurs procèdent à l'analyse statistique des méthodes d'obfuscation des obfuscateurs disponibles sur le marché en leur présentant des sources à obfusquer et dont ils analysent le résultat après obfuscation. Ce travail facilite alors la tâche de désobfuscation. Il faudra alors avoir recours à des obfuscateurs faits maisons mais probablement moins efficaces que ceux du marché. D'où le dilemme...

V - Conclusion

Cet article a fait le tour des techniques d'obfuscations et vous en a présenté les avantages et inconvénients. Tout comme dans le domaine de la cryptographie ou des DRM, la bataille entre obfuscateurs et désobfuscateurs fait rage. Sans cesse des parades à l'obfuscation seront trouvées et obligeront les éditeurs d'obfuscateurs à mettre à niveau leurs outils.

L'obfuscation ne peut être vue comme une solution durable pour la protection de la propriété intellectuelle d'autant que la qualité et les performances d'une application obfusquée vont être sérieusement dégradées. Il faut donc être très prudent dans son usage et lui préférer une réflexion sur les méthodes commerciales et le positionnement stratégique de son entreprise et de ses applications afin de défendre au mieux ses intérêts économiques.

VI - Voir aussi

Pour tous ceux qui souhaitent en savoir d'avantage sur les algorithmes utilisés par les obfuscateurs, je recommande cet excellent article :

- [A Taxonomy of Obfuscating Transformations](#) de Christian Collberg, Clark Thomborson et Douglas Low de l'Université d'Auckland

Vous trouverez sur [Developpez.com](#) d'autres ressources sur des obfuscateurs connus :

- .NET : [Protégez, optimisez et contrôlez votre code avec {smartassembly}](#) de Louis-Guillaume MORAND
- .NET : [Protection, optimisation et déploiement de code .NET grâce à XenoCode 2005](#) de Thomas LEBRUN
- J2ME : [Antenna](#) de christopheJ

VI - Remerciements

Je tiens à remercier [Eric Reboisson](#), [gorgonite](#) et [Webman](#) pour leurs corrections.