La rétroconception : application à l'analyse logicielle

Serge Lefranc

Centre d'Électronique de l'Armement
Bruz CEDEX F-35174, France
serge.lefranc@dga.defense.gouv.fr

Résumé Cet article a pour but de présenter la rétroconception et son application à l'analyse logicielle. Pour cela, nous nous attacherons à présenter certains aspects juridiques relatifs à la rétroconception. Nous expliciterons ensuite ses buts afin de nous permettre d'élaborer une méthodologie associée à l'utilisation de certains outils. Après avoir donné quelques exemples de mécanismes de protection et les mesures de contournement correspondantes, nous essayerons d'illustrer cet article par un exemple.

1 Introduction

La rétroconception peut se définir comme l'action d'extraire une connaissance ou un savoir d'une réalisation.

Cette activité est pratiquée depuis très longtemps dans un grand nombre de domaines industriels comme l'automobile ou l'électronique, afin, notamment, de permettre l'accession à un savoir ou une technologie, à moindre frais.

L'informatique n'a pas été épargnée par ce phénomène. Tout possesseur d'un programme possède le code de l'application traduit dans le seul langage que le processeur comprenne, le langage machine. Il suffit ensuite d'avoir les compétences nécessaires pour obtenir un accès de bas niveau aux fonctionnalités du programme.

Historiquement, c'est dans les pays de l'ex Bloc de l'Est et dans certains pays d'Asie que les compétences en rétroconception sont les plus vivaces. N'ayant pas accès de façon simple aux logiciels des pays de l'ex Bloc de l'Ouest, principalement pour cause de restriction, ou d'isolement économique, une forte communauté de rétroconcepteurs s'est développée, d'autant qu'elle était encouragée par l'absence de sanctions et de législation au niveau étatique. C'est donc, tout naturellement, le monde du logiciel payant qui a été le plus victime de cette forme de piratage. Par conséquent, il est normal de trouver une importante communauté de rétroconcepteurs dans le monde Windows.

Ces dernières années ont cependant été marquées par une évolution des tendances. Des compétences en rétroconception ont été nécessaires, et acquises, de la part de la communauté Open Source afin d'identifier les mécanismes de fonctionnement des différents vers qui ont touché le monde du logiciel libre ou d'extraire de l'information des traces binaires retrouvées sur une machine compromise.

Notre exposé va donc tenter, après avoir abordé l'aspect juridique de la rétroconception, de montrer que ses buts sont nombreux et qu'ils dépendent très fortement de l'état d'esprit des personnes qui la pratiquent. Nous verrons ensuite qu'il est difficile de définir une méthodologie de la rétroconception car le spectre de ses objectifs est trop large. Dans le cas le plus simple, typiquement le cracking, nous savons ce que nous cherchons, et dans le cas le plus compliqué, comme la recherche de vulnérabilités, nous n'en n'avons aucune idée. Ceci nous conduira à présenter quelques schémas de protection et les façons de les contourner pour enfin d'illustrer notre présentation par une exemple d'analyse logicielle.

2 Aspects juridiques

Cette partie est basée sur le cours de droit informatique, *l'univers numérique et le droit*, écrit par Maître Gérard HAAS donné à l'École Nationale Supérieure de Techniques Avancées en 2001.

Il convient de ne pas confondre désassemblage et décompilation. Cette dernière transforme le code machine en langage de haut niveau alors que le désassemblage le transforme en langage assembleur.

2.1 Droit d'analyse

L'utilisateur régulier d'un logiciel se voit reconnaître expressément le droit, sans avoir à en demander l'autorisation à l'auteur, d'observer, étudier ou tester le fonctionnement de ce logiciel afin de déterminer les idées et principes qui sont à la base de n'importe quel élément du programme, lorsqu'il effectue des opérations de chargement, d'affichage, de passage, de transmission ou de stockage du programme (Code de la Propriété Intellectuelle art. L.122-6-1-111).

Le droit d'analyse ne peut être exercé que dans le cadre de l'utilisation normale du logiciel. À la différence du droit de décompilation, le droit d'analyse est réservé à l'utilisateur du logiciel, qui ne peut autoriser un tiers à l'exercer à sa place.

2.2 Décompilation

L'utilisateur régulier d'un logiciel peut, sans autorisation de l'auteur, procéder à la reproduction et à la traduction de la forme de son code, c'est à dire sa *décompilation*, lorsque ces opérations sont indispensables pour obtenir les informations nécessaires à l'interopérabilité de ce logiciel avec d'autres logiciels (Code de la Propriété Intellectuelle art L.122-6-1-IV).

Compte tenu des termes de la loi, il semble que, outre la décompilation stricto sensu, le *désassemblage* du logiciel soit permis (Hubert Bitan, 01 Informatique, 17/06/94, p.36). Par ailleurs, la notion de décompilation était inconnue de notre droit, elle a été introduite par la directive européenne du 14 mai 1991.

L'objectif clairement affirmé du législateur est de permettre l'interopérabilité des logiciels, définie par la directive comme la *capacité des programmes d'ordinateurs d'échanger des données et d'utiliser des données qui ont été échangées.*

Le droit de décompiler demeure cependant enfermé dans des conditions très strictes :

- il doit s'agir d'un logiciel créé de façon indépendante et donc qui n'est pas déjà conçu pour être compatible avec d'autres logiciels ;
- seuls sont habilités à procéder à des actes de décompilation les licenciés ou utilisateurs réguliers du logiciel ou les tiers agissant pour leur compte ;
- les informations nécessaires à l'interopérabilité ne sont pas déjà facilement et rapidement accessibles ;
- les actes de décompilation sont limités aux parties du logiciel d'origine nécessaires à l'interopérabilité.

Il convient d'être réservé sur la portée pratique de cette dernière limitation. En effet, il est impossible de savoir, la plupart du temps, où se trouvent les interfaces dans un programme, et la preuve d'un abus s'annonce pour le moins délicate, sauf si le concepteur a donné des indications suffisantes... De plus, les dispositions permettant la décompilation n'autorisent pas pour autant que les informations obtenues :

- soient utilisées à des fins autres que l'interopérabilité des logiciels avec d'autres logiciels ;
- soient communiquées à des tiers sauf si cela est nécessaire pour permettre l'interopérabilité avec le logiciel crée de façon indépendante ;
- soient utilisées pour la mise au point, la production ou la commercialisation d'un logiciel dont l'expression est fondamentalement similaire à celle du logiciel d'origine ou pour tout autre acte portant atteinte au droit d'auteur.

3 Les buts de la rétroconception

Pourquoi avoir besoin de désassembler un programme ? Nous avons vu que légalement, et sous certaines conditions, cette opération est autorisée pour assurer une interoperabilité des applications.

Cependant, ce n'est qu'une facette des raisons motivant le désassemblage. La rétroconception permet également de vérifier la présence de portes cachées ou de vulnérabilités dans un logiciel. Elle peut permettre de passer outre un système de protection, voire même permettre l'accès à certaines techniques propriétaires non documentées.

Toutes ces raisons de pratiquer le désassemblage sont très dépendantes des différents acteurs qui vont la réaliser. En effet, leurs motivations ne vont pas être les mêmes. Et cela va avoir une incidence sur les compétences à mettre en œuvre pour la réaliser. En particulier plus le but de la rétroconception est précis, plus les compétences nécessaires sont simples à acquérir : il est plus simple de chercher quelque chose de connu que d'analyser un programme sans savoir ce que l'on cherche.

3.1 Une entreprise

Pour une entreprise, nous identifions trois raisons principales d'avoir recours à la rétronception :

- assurer l'intéropérabilité entre une application et son parc applicatif ;
- vérifier que le produit ne fait que ce qu'il est censé faire ;
- accéder aux technologies mises au point par des sociétés concurrentes.

Les moyens mis en œuvre pour réaliser ces objectifs sont nombreux car le spectre de difficultés rencontrées est large. Pour assurer l'interopérabilité, les problèmes sont relativement aisés à identifier, même si ils ne sont pas forcément simples à résoudre, alors que pour les deux autres objectifs, il est difficile d'identifier ce qu'il faut chercher. Par exemple, la recherche de vulnérabilités est considérée comme le pire cas dans lequel le rétroconcepteur peut se trouver : il n'a aucune idée de ce qu'il cherche, ni de ce qu'il va trouver.

3.2 Un particulier

La problématique est plus simple pour un particulier. Nous identifions deux raisons qui peuvent le mener à analyser un logiciel :

- enlever les mécanismes de protection lui interdisant l'utilisation de ce logiciel sans s'être acquitté des droits de licence ;
- le modifier afin d'incorporer des fonctionnalités particulières.

Le premier cas de figure fait appel à certaines compétences spécifiques aux méthodes de protections utilisées par les éditeurs de logiciels (nous aborderons, par la suite, quelques techniques utilisées par ces concepteurs pour protéger leurs œuvres), tandis que le deuxième cas va nécessiter des compétences qui dependront des modifications à apporter, cela peut aller du très simple au très compliqué.

3.3 Un groupe de pirates

Pour un groupe de pirates nous identifions également deux objectifs :

- supprimer des différents mécanismes de protection présents sur le logiciel afin de permettre de le distribuer massivement de façon gratuite ;
- rechercher des vulnérabilités afin de les exploiter et de les distribuer.

Les compétences nécéssaires pour réaliser le premier objectif sont très spécifiques. En effet, les schémas de protection évoluant finalement assez peu, les méthodes à employer sont relativement simples. Par contre, le deuxième objectif nécessite des connaissances très poussées en programmation de bas niveau, en architecture des systèmes d'exploitation, voire même en architecture rèseau.

3.4 Une organisation criminelle

L'organisation criminelle poursuit les mêmes buts que le groupe de pirates, mais à l'inverse, elle le fait dans un but lucratif.

3.5 Un organisme étatique

L'organisme étatique a deux principales raisons de faire de la rétroconception :
- vérifier que le logiciel ne possède pas de fonctions cachées ou de vulnérabilités pouvant dégrader la sécurité de l'ensemble du système ;
- rechercher des vulnérabilités afin de les exploiter dans un contexte opérationnel.

On considère que ces objectifs sont ceux qui nécessitent le plus de compétences. En terme de difficultés techniques, ce cas est analogue à celui de l'entrerprise.

4 La méthodologie

Il n'existe pas de méthodologie de la rétroconception ! En effet, si cela était le cas, il existerait des logiciels « clé en main » permettant d'analyser un exécutable.

Nous avons vu que les objectifs à atteindre peuvent être extrêmement variés et nécessiter des compétences très différentes. Cela va du plus simple : enlever une boite de dialogue, au plus compliqué : identifier la présence d'une vulnérabilité et la manière de l'exploiter. Il n'est donc pas possible de décrire une méthodologie de la rétroconception que nous pourrions appliquer à tous les cas de figure.

Cependant, il est quand même possible de dégager un semblant de méthode. Cette esquisse de méthode résulte en fait de l'utilisation de logiciels bien particulier qui vont nous permettre d'accéder à l'information désirée :
- un désassembleur ;
- un débuggeur ;
- des logiciels de surveillance système.

Il est possible de classer ces outils en deux grandes familles : les outils de rétroconception et les outils de surveillance système.

4.1 Les outils de rétroconception

Ils vont nous permettre d'acquérir une vision du fonctionnement interne du programme. Il devient alors possible d'étudier les processus internes du logiciel, de comprendre ses faiblesses et, éventuellement, de les exploiter.

Le désassembleur Il permet d'étudier de façon statique ce que réalise le programme en traduisant l'exécutable en suite d'instructions machine. C'est lui qui va nous permettre de générer le code assembleur correspondant au programme.

Les désassembleurs IDA pro de la société Datarescue [1] ou DASM32 sont couramment utilisés sous Windows.

Le débuggeur Il va nous permettre d'étudier le comportement du programme pendant son exécution. Il va être capable d'en tracer l'exécution pas à pas, de connaître la valeur des registres, de placer des breakpoints (points d'arrêts : lors de la présence d'un breakpoint, le débuggeur fait une pause dans l'exécution de l'application) à certains endroits ou en fonction de certaines conditions.

Il existe deux types de débuggeurs : ceux fonctionnant au niveau applicatif et ceux fonctionnant au niveau système. Pour simplifier, nous pouvons considérer que les débuggeurs de niveau applicatif résident entre l'OS et le programme (ring 3) et que ceux de niveau système résident entre le processeur et l'OS (ces derniers sont plus puissants car ils peuvent débugger au niveau noyau, ring 0). Typiquement, le débuggeur présent dans la suite de développement Visual C++ de Microsoft [2] est de niveau applicatif, et le débuggeur SoftICE proposé par Numega est de niveau système [3].

4.2 Les outils de surveillance

Ce sont des outils qui vont nous permettre de surveiller l'activité du système de fichiers en temps réel, de surveiller l'activité de la base des registres ou l'activité réseau. Cela va permettre d'avoir une information externe au programme et d'identifier les interactions qu'il peut avoir avec son environnement de fonctionnement. Le programme *monitor* de la société Sysinternals [4] permettent de réaliser ces tâches.

5 Quelques protections

Nous venons de voir, qu'hormis l'utilisation d'outils bien spécifiques, il est difficile d'isoler une méthode précise pour faire de la rétroconception d'application. Nous allons détailler les schémas de protection les plus fréquents.

5.1 Les tokens logiciels

C'est l'une des méthodes les plus utilisées de protection logicielle. Ils peuvent prendre différentes formes.

Clé d'enregistrement. Une clé d'enregistrement est présente dans le programme. Lorsque l'utilisateur rentre son numéro de clé, le programme vérifie qu'elle est identique à celle qui est stockée dans le programme.

Serial multiple. Le numéro de série du logiciel est découpé en plusieurs parties ([xx][yy][zz]). Le programme possède un algorithme de vérification du numéro de série qui va vérifier les différentes parties de ce numéro. Cette technique est une amélioration de la précédente car elle permet l'utilisation de différents numéros de série sans avoir à les programmer en dur dans le programme.

Serial/Nom. Le token correspond ici à un couple numéro de série/nom. La vérification utilise le même principe que pour les serials multiples.

Fichier de clé. Le token correspond à un fichier contenant la licence d'utilisation. Elle est stockée dans un fichier sur le disque ou dans la base des registres.

5.2 Les tokens matériels

Étant donné le caractère volatile de l'information numérique et la facilité avec laquelle elle peut être reproduite, il a été envisagé que des solutions à base de tokens matériels pourraient être plus robustes au piratage. Il est en effet plus difficile de dupliquer une clé physique. Elles peuvent prendre plusieurs formes :

Disquette clé. Ce sont des disquettes spéciales dans lesquelles certains secteurs ont été physiquement altérés. Le programme va vérifier la présence des secteurs défectueux.

Dongle. Ce sont des petits périphériques matériels qui s'attachent aux ports d'entrées/sorties de la machine. Les routines de vérification vont demander certaines valeurs aux périphériques connectés à ces ports. Si le token matériel est là, il va détecter le signal électrique et générer une réponse appropriée.

Carte à puce. L'information et/ou les routines de vérification se trouvent dans le processeur de la carte à puce.

5.3 Les NAG screen

Ce sont typiquement des fenêtres *pop-up* qui annoncent à l'utilisateur qu'il peut utiliser le programme mais qu'il n'en possède pas la licence. Elles s'ouvrent à chaque exécution du programme.

5.4 Les limites

Ce sont des limites dans l'utilisation du logiciel. Elles peuvent porter sur le nombre de fois où le programme peut être utilisé, la durée pendant laquelle il peut fonctionner, etc. Le programme ne pourra plus se lancer lorsque la limite sera atteinte.

5.5 Le chiffrement de code

C'est l'une des façons les plus simples de protéger l'exécutable d'un programme. Ce type de techniques est également utilisé par certains virus :

```
mov EAX, count
mov EBX, address
mov ECX, offset
_LOOP:
xor [ECX], EBX
DEC EAX
ja _LOOP
; ensuite on trouve le code et les données
; chiffrées du programme
```

Ainsi, lorsque le programme va être décompilé, l'utilisateur n'aura pas immédiatement accès aux instructions assembleur, il faudra qu'il identifie la ou les routines de chiffrement et qu'il déchiffre l'exécutable. Dans notre exemple, cette routine est un simple XOR, mais c'est souvent plus compliqué.

5.6 Le packing d'exécutable

Cette méthode est utilisée pour compresser le programme tout en lui permettant d'être exécutable en mémoire. Il existe un certain nombre de logiciels permettant de réaliser cette tâche (UPX, Neolite, PKLITE32, ...). Le fonctionnement de ce procédé est assez analogue au chiffrement de code.

5.7 La transformation de code

Le but de cette méthode est de rendre l'exécutable désassemblé incompréhensible et ainsi de ralentir le travail de l'utilisateur en l'empêchant de récupérer facilement de l'information sur le programme.

Il existe différentes façons de rendre cela possible :
- transformation lexicale ;
- transformation de contrôle : on ajoute des instructions opaques.

Par exemple, la suite d'instructions suivante :

```
instruction_a
instruction_b
```

est remplacée par :

```
instruction_a
if (p == true)
 instruction_b
else
 instruction_b
```

Il est possible de rendre la lecture d'un listing assembleur très difficile en utilisant ce type de méthodes. Par contre, l'efficacité du programme en est diminuée.

5.8 Les fonctions anti-débuggeur

Le but du jeu est de rendre l'utilisation d'un débuggeur beaucoup plus compliquée. Pour réaliser cela, il existe deux méthodes principales :
- actions préventives : ce sont des actions effectuées par le programme pour empêcher l'utilisateur de le tracer durant son exécution (masquage des interruptions, modification de l'IVT, ...) ;
- code auto-modifiant (algorithme de chiffrement/déchiffrement, ...).

Pour plus de précision on se réfera à l'article écrit par Inbar RAZ [5].

6 Les mesures de contournements associées

6.1 Le scénario le plus simple

Les schémas de protection que nous venons de présenter sont très simples à mettre en échec s'ils n'incorporent pas des mécanismes de chiffrement/obscurcissement car ils font souvent appel au modèle de protection suivant :

```
result = security_check(condition1, condition2)
if (result == TRUE)
then
 <autorise et execute le programme>
else
 <essaye encore>
```

La `condition1` peut être le numéro de série entré par l'utilisateur et la `condition2` le numéro de série qui est valide. La fonction `security_check` peut aller de la simple comparaison de bit à une requête I/O. Le pseudo code que nous venons d'écrire peut se traduire, en langage assembleur, de la façon suivante :

```
push condition2
push condition1
call security_check
test EAX, EAX
jnz address1 ; adresse du debut du programme
<essaye encore>
```

Le résultat de l'appel à la fonction est stocké dans le registre `EAX`. L'opération `TEST` réalise un *ET* logique. Si le résultat est 0 (faux), l'opération de *ET* logique va positionner le flag `Z` à 1 dans le registre de flags et le saut ne va pas avoir lieu. Si le contenu du registre `EAX` est différent de 0, alors le saut à l'adresse `address1` va avoir lieu.

Ce type de protection est facilement outrepassable en utilisant un débuggeur ou un désassembleur.

6.2 Utilisation d'un débuggeur

En positionnant certains points d'arrêt (par exemple, sur une fonction de comparaison de chaînes de caractères), il va être possible à l'utilisateur :

- de récupérer le serial (stocké en dur dans le programme : `condition2`) ;
- de modifier l'instruction JNZ en JMP, le saut vers le début du programme aura alors toujours lieu ;
- de modifier l'appel à la fonction `security_check` par des NOP, il ne va rien se passer ;
- d'extraire l'algorithme de génération de clés afin de fabriquer un générateur de clés (keygen) dans le cas où la clé n'est pas stockée en dur dans le programme ;

6.3 Utilisation d'un désassembleur

Il n'est pas systématiquement nécessaire de faire appel à un débuggeur, parfois une analyse statique du programme peut se révéler suffisante. En regardant les références à des chaînes de caractères dans le code désassemblé, l'utilisateur peut remonter jusqu'à la routine `security_check` et la « corriger ».

7 Exemple

Notre but est d'essayer de cacher les connexions réseau ouvertes sur une machine de type Windows 2000. Pour cela nous allons modifier le programme `netstat.exe` de Windows car il permet à un utilisateur de connaître l'état des connexions de sa machine.

Nous allons analyser le fonctionnement de ce programme et son comportement lorsqu'une connexion est ouverte sur la machine, puis nous allons essayer d'identifier la ou les fonctions responsables de l'affichage.

Ensuite nous réfléchirons à une façon de modifier le code du programme qui nous permettra de sélectionner les lignes à afficher.

Cette analyse s'inspire en partie du travail effectué par `threat` sur `nethide`.

7.1 Analyse des fonctionnalités du programme

La façon la plus simple de connaître le fonctionnement du programme est tout simplement de commencer par l'utiliser. Pour cela, il suffit de lancer une fenêtre cmd.exe sous Windows et d'exécuter la commande `netstat -an` qui va nous lister toutes les connexions réseau ouvertes sur la machine (figure 1).

Nous réalisons donc qu'à chaque nouvelle connexion ouverte, une nouvelle ligne est affichée.

Nous recherchons donc la ou les fonctions permettant l'affichage d'une chaîne de caractères. Pour cela, nous désassemblons l'exécutable avec IDA Pro et nous regardons les fonctions « intéressantes » de l'API win32 qui sont importées par le programme :

- `FormatMessageA`
- `CharToOemBuffA`
- `fprintf`
- `sprintf`

Toutes ces fonctions ne sont pas indispensables à la réalisation de notre objectif. Afin de déterminer quelles vont être les fonctions à modifier, il est nécessaire de se référer à la définition de chacune d'entre elles dans le guide de développement Microsoft MSDN.

Il apparaît que la fonction `FormatMessageA` est utilisée pour formater une chaîne de caractères. C'est exactement la fonction que nous recherchions.

Désormais, nous avons identifié la fonction importée par l'API win32 qui est responsable de l'affichage et du formatage des informations listées par la commande `netstat -an`.

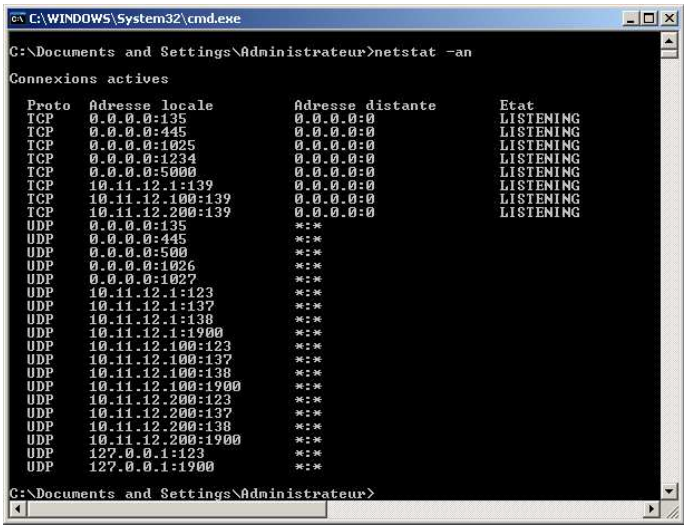

Fig. 1. Liste des connections réseaux ouvertes.

Nous allons maintenant identifier la ou les routines du programme qui font appel à `FormatMessageA`. Pour cela, nous allons utiliser SoftICE et placer un point d'arrêt sur cette fonction. Il est à noter que l'utilisation de SoftICE n'est pas réellement nécessaire car la fonction de formatage de chaînes de caractères n'est apellée que deux fois. Après avoir placé notre point d'arrêt, nous revenons sous Windows et nous relançons la commande `netstat -an`. SoftICE apparaît et nous traçons le programme avec la touche F12 jusqu'à voir les lignes d'affichage dans notre fenêtre `cmd.exe`.

Il apparaît un appel récurrent à la routine se trouvant à l'adresse `001305B`. C'est notre fonction d'affichage !

7.2 Choix d'une fonctionnalité à modifier

Nous savons maintenant que la fonction de formatage des lignes, contenant l'information sur les connexions ouvertes, est appelée par la routine se trouvant à l'adresse `001305B`. C'est cette routine que nous allons analyser afin d'identifier ce qu'il est possible de modifier pour réaliser notre objectif.

Sous IDA Pro, nous allons donc à l'adresse `001305B` afin d'analyser la suite d'instructions assembleur.

```
.text:0100305B ; int __cdecl sub_100305B(int,DWORD dwMessageId,char)
.text:0100305B sub_100305B     proc near         ; CODE XREF: _main+71 p
.text:0100305B                                   ; _main+1C0 p ...
.text:0100305B
.text:0100305B Arguments       = dword ptr -8
```

```
.text:0100305B lpszDst         = dword ptr -4
.text:0100305B arg_0           = dword ptr  8
.text:0100305B dwMessageId     = dword ptr  0Ch
.text:0100305B arg_8           = byte ptr  10h
.text:0100305B
.text:0100305B                 push    ebp
.text:0100305C                 mov     ebp, esp
.text:0100305E                 push    ecx
.text:0100305F                 push    ecx
.text:01003060                 lea     eax, [ebp+arg_8]
.text:01003063                 push    ebx
.text:01003064                 mov     [ebp+Arguments], eax
.text:01003067                 lea     eax, [ebp+Arguments]
.text:0100306A                 push    esi
.text:0100306B                 push    eax
.text:0100306C                 xor     esi, esi
.text:0100306E                 lea     eax, [ebp+lpszDst]
.text:01003071                 push    esi
.text:01003072                 push    eax
.text:01003073                 push    esi
.text:01003074                 push    [ebp+dwMessageId]
.text:01003077                 push    esi
.text:01003078                 push    900h
.text:0100307D                 call    ds:FormatMessageA
.text:01003083                 mov     ebx, eax
.text:01003085                 cmp     ebx, esi
.text:01003087                 jnz     short loc_100308D
.text:01003089                 xor     eax, eax
.text:0100308B                 jmp     short loc_10030E7
.text:0100308D ; ---------------------------------------------------------------
.text:0100308D
.text:0100308D loc_100308D:                    ; CODE XREF: sub_100305B+2C j
.text:0100308D                 mov     eax, ds:_iob
.text:01003092                 cmp     [ebp+arg_0], 2
.text:01003096                 push    edi
.text:01003097                 lea     esi, [eax+40h]
.text:0100309A                 jz      short loc_100309F
.text:0100309C                 lea     esi, [eax+20h]
.text:0100309F
.text:0100309F loc_100309F:                    ; CODE XREF: sub_100305B+3F j
.text:0100309F                 push    8000h
.text:010030A4                 push    dword ptr [esi+10h]
.text:010030A7                 call    ds:_setmode
.text:010030AD                 mov     edi, [ebp+lpszDst]
.text:010030B0                 pop     ecx
```

```
.text:010030B1                  pop     ecx
.text:010030B2                  xor     eax, eax
.text:010030B4                  or      ecx, 0FFFFFFFFh
.text:010030B7                  repne scasb
.text:010030B9                  not     ecx
.text:010030BB                  dec     ecx
.text:010030BC                  push    ecx
.text:010030BD                  push    [ebp+lpszDst]
.text:010030C0                  push    [ebp+lpszDst]
.text:010030C3                  call    ds:CharToOemBuffA
.text:010030C9                  push    [ebp+lpszDst]
.text:010030CC                  push    offset aS
.text:010030D1                  push    esi
.text:010030D2                  call    ds:fprintf
.text:010030D8                  add     esp, 0Ch
.text:010030DB                  push    [ebp+lpszDst]
.text:010030DE                  call    ds:LocalFree
.text:010030E4                  mov     eax, ebx
.text:010030E6                  pop     edi
.text:010030E7
.text:010030E7 loc_10030E7:                             ; CODE XREF: sub_100305B+30 j
.text:010030E7                  pop     esi
.text:010030E8                  pop     ebx
.text:010030E9                  leave
.text:010030EA                  retn
.text:010030EA sub_100305B      endp
```

Après l'appel à la fonction `FormatMessageA` il y a une suite de cinq instructions très intéressantes :

```
.text:01003083                  mov     ebx, eax
```

Cette instruction copie le nombre de caractères écrits dans le buffer dans `EBX` (`EAX` contient la valeur de retour de l'appel à la fonction `FormatMessageA`).

```
.text:01003085                  cmp     ebx, esi
```

Celle-ci regarde si la chaîne de caractères est vide (`ESI` vaut 0).

```
.text:01003087                  jnz     short loc_100308D
```

Si le nombre de caractères de la chaîne n'est pas nul, alors `ZF` = 0 et le flot d'instructions se déplace à l'adresse `100308D` (affichage de la chaîne de caractères).

```
.text:01003089                  xor     eax, eax
```

Sinon le registre `EAX` est mis à 0.

```
.text:0100308B                  jmp     short loc_10030E7
```

Cette dernière instrction déplace le flot d'instructions à la fin du programme (rien à afficher).

7.3 Réalisation de la modification

Nous venons de voir que si il n'y a rien à afficher, nous allons directement à l'adresse de fin du programme (`10030E7`). Si ce n'est pas le cas, nous nous déplaçons à l'adresse `100308D`, soit 4 octets plus loin dans le flot d'instructions.

Afin de modifier les instructions pour que plus rien ne soit affiché par le programme, et cela, même si il y a quelque chose dans la chaîne de caractères, il suffit de modifier l'adresse du saut conditionnel afin qu'il désigne le saut qui amène à la fin du programme. Typiquement, il suffit de modifier :

```
.text:01003087                  jnz     short loc_100308D
```

par

```
.text:01003087                  jnz     short loc_100308B
```

Cela revient à changer la valeur d'un seul octet et le programme n'affichera plus rien (figure 2). Il suffit ensuite d'éditer l'exécutable `netstat.exe` avec un éditeur hexadécimal et de changer l'octet désiré.

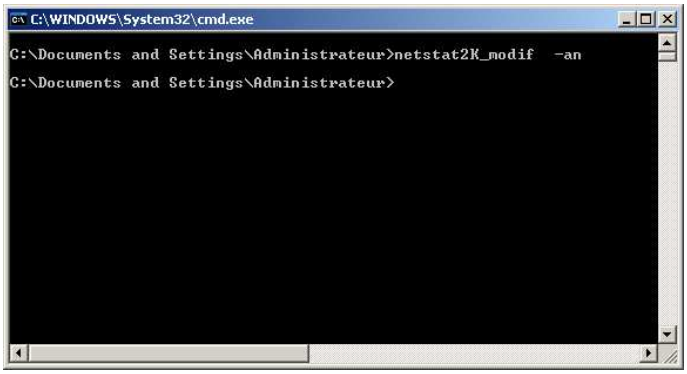

Fig. 2. Affichage du programme netstat modifié

8 Conclusion

Nous venons de montrer que la rétroconception est un moyen efficace d'obtenir de l'information sur un programme exécutable dont nous ne possédons pas les sources.

En modifiant un seul octet du programme, nous avons été capables de modifier son fonctionnement au point de le rendre inefficace.

Pour éviter ce type de manipulations, il existe des mécanismes de protection. Cependant il est illusoire d'espérer mettre au point des techniques réellement incontournables car il y aura toujours un moment où le code machine apparaîtra en clair en mémoire.

9 Remerciements

Je tiens à remercier D.E. pour ses réponses à mes questions, S. Griffe et P. Biondi pour toutes les relectures.

Références

1. IDA Pro, http ://www.datarescue.com
2. Microsoft Visual C++, http ://www.microsoft.com
3. SoftICE, http ://www.numega.com
4. SysInternals, http ://www.sysinternals.com
5. Anti Debugging Tricks, Inbar RAZ, http ://www.bsdg.org/swag/FAQ/0054.PAS.html