Chapitre 3
Reverse engineering

1. Introduction

1.1 Présentation

Le reverse engineering (ou rétro-ingénierie en français) consiste à étudier un objet (dans notre cas un malware) pour comprendre son fonctionnement. En informatique, cela se traduit par l'analyse du code machine d'un programme, dans notre cas d'un malware. Étant donné que les malwares ne sont pas diffusés avec leur code source et qu'il n'est pas possible de retrouver les codes des malwares développés en C ou en C++, il est nécessaire de faire appel au reverse engineering pour étudier leur fonctionnement interne. L'analyste étudiera le code assembleur du malware, fonction après fonction. Ce code assembleur est disponible après désassemblage du binaire.

Le code assembleur n'est pas aussi facile à lire que du code source. En effet, c'est un langage bas niveau qui manipule directement les instructions CPU et la mémoire physique.

Nous allons principalement nous intéresser à l'assembleur x86 (32 bits), même si une courte section présentera les principales différences entre le x86 et le x64 (64 bits) d'un point de vue rétro-ingénierie. Aujourd'hui, 80 % des malwares sont compilés en 32 bits, ceci afin de pouvoir impacter le plus de machines possible (les systèmes Windows 64 bits supportent les binaires 32 bits, mais l'inverse n'est pas vrai).

Ce chapitre expliquera comment lire et interpréter l'assembleur, les outils utilisables pour mener l'analyse et des astuces pour la faciliter.

1.2 Législation

Dans de nombreux pays, le reverse engineering est encadré par des lois. De nombreuses utilisations peuvent être faites de cette discipline :

– L'espionnage industriel : certaines sociétés utilisent le reverse engineering pour comprendre les produits développés par les concurrents et voler leur savoir-faire.
– La destruction de protection anticopie : certaines personnes utilisent ces techniques pour permettre de copier des jeux vidéo ou de copier de la musique utilisant un système anticopie (DRM, *Digital Rights Management*).
– L'interopérabilité : des développeurs pratiquent le reverse engineering pour réécrire des logiciels permettant d'utiliser les produits sur des plateformes non supportées par un fabricant (c'est le cas de nombreux drivers sous Linux).

Cette liste n'est bien évidemment pas exhaustive, mais permet de comprendre que cette technique peut être utilisée à bon et à mauvais escient.

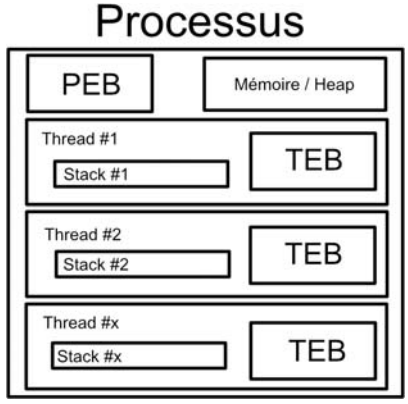
L'utilisation du reverse engineering pour l'analyse de malwares est bien évidemment parfaitement légale. Voici l'extrait de l'article français à ce propos (art. L. 335-3-1 du Code de la propriété intellectuelle) :

« III. - Ces dispositions **ne sont pas applicables aux actes réalisés à des fins de sécurité informatique**, dans les limites des droits prévus par le présent code. »

2. Qu'est-ce qu'un processus Windows ?

2.1 Introduction

Lorsqu'on exécute un processus sous Windows, le système d'exploitation va automatiquement créer un espace mémoire pour celui-ci et un premier thread. Chaque processus en cours d'exécution dispose d'une structure le décrivant, appelée PEB (*Process Environment Block*). Chaque processus dispose d'un ou plusieurs threads. Un thread est un fil d'exécution. Les threads sont exécutés en parallèle. Chaque thread dispose de sa propre pile (*stack*, cf. chapitre Techniques d'obfuscation) et d'une structure le définissant appelée TEB (*Thread Environment Block*). Les threads peuvent accéder à la mémoire du processus. Voici un schéma décrivant un processus :



2.2 Process Environment Block

Il est possible de lire le contenu du PEB d'un processus. Voici un exemple de PEB du processus cmd.exe vu par le débogueur de Microsoft : *WinDbg*. L'adresse mémoire à laquelle se trouve la structure PEB est stockée dans `$PEB` :

```
0:001> r $PEB
$peb=0000000301292000
```

À partir de cette adresse, nous pouvons vérifier le contenu de la structure PEB :

```
0:001> dt _PEB 0000000301292000
ntdll!_PEB
   +0x000 InheritedAddressSpace : 0 ''
   +0x001 ReadImageFileExecOptions : 0 ''
   +0x002 BeingDebugged    : 0x1 ''
   +0x003 BitField         : 0x4 ''
   +0x003 ImageUsesLargePages : 0y0
   +0x003 IsProtectedProcess : 0y0
   +0x003 IsImageDynamicallyRelocated : 0y1
   +0x003 SkipPatchingUser32Forwarders : 0y0
   +0x003 IsPackagedProcess : 0y0
   +0x003 IsAppContainer   : 0y0
   +0x003 IsProtectedProcessLight : 0y0
   +0x003 IsLongPathAwareProcess : 0y0
   +0x004 Padding0         : [4]  ""
   +0x008 Mutant           : 0xffffffff`ffffffff Void
   +0x010 ImageBaseAddress : 0x00007ff7`da850000 Void
   +0x018 Ldr              : 0x00007ffe`98e753c0 _PEB_LDR_DATA
   +0x020 ProcessParameters : 0x000001e6`84de1be0
_RTL_USER_PROCESS_PARAMETERS
   +0x028 SubSystemData    : (null)
   +0x030 ProcessHeap      : 0x000001e6`84de0000 Void
   +0x038 FastPebLock      : 0x00007ffe`98e74fc0
_RTL_CRITICAL_SECTION
[...]
```

Par exemple, à l'offset 0x020 se trouve la structure _RTL_USER_PROCESS_PARAMETERS contenant les paramètres du processus. Elle se trouve à l'adresse 0x000001e6`84de1be0. Voici son contenu :

```
0:001> dt _RTL_USER_PROCESS_PARAMETERS 0x000001e6`84de1be0
ntdll!_RTL_USER_PROCESS_PARAMETERS
  +0x000 MaximumLength    : 0x788
  +0x004 Length           : 0x788
  +0x008 Flags            : 0x6001
  +0x00c DebugFlags       : 0
  +0x010 ConsoleHandle    : 0x00000000`00000050 Void
  +0x018 ConsoleFlags     : 0
  +0x020 StandardInput    : 0x00000000`00000054 Void
  +0x028 StandardOutput   : 0x00000000`00000058 Void
  +0x030 StandardError    : 0x00000000`0000005c Void
  +0x038 CurrentDirectory : _CURDIR
  +0x050 DllPath          : _UNICODE_STRING ""
```

```
+0x060 ImagePathName    : _UNICODE_STRING "C:\WINDOWS\system32\cmd.exe"
+0x070 CommandLine      : _UNICODE_STRING ""C:\WINDOWS\system32\cmd.exe" "
+0x080 Environment      : 0x000001e6`84df6380 Void
```

Comme nous pouvons le voir, la ligne de commande est disponible à l'offset 0x70 de cette structure dans le processus en cours d'exécution.

WinDbg fournit la commande `!PEB`, qui lit et formate tous les éléments pertinents du PEB et les affiche.

2.3 Thread Environment Block

Le même exercice peut être réalisé avec la structure TEB :

```
0:001> r $TEB
$teb=0000000301299000
0:001> dt _TEB 0000000301299000
ntdll!_TEB
   +0x000 NtTib            : _NT_TIB
   +0x038 EnvironmentPointer : (null)
   +0x040 ClientId         : _CLIENT_ID
   +0x050 ActiveRpcHandle  : (null)
   +0x058 ThreadLocalStoragePointer : (null)
   +0x060 ProcessEnvironmentBlock : 0x00000003`01292000 _PEB
   +0x068 LastErrorValue   : 0
   +0x06c CountOfOwnedCriticalSections : 0
   +0x070 CsrClientThread  : (null)
   +0x078 Win32ThreadInfo  : (null)
   +0x080 User32Reserved   : [26] 0
```

Il est intéressant de noter qu'à l'offset 0x60 se trouve l'adresse du PEB vue précédemment. Dans le cas d'un processus 32 bits, l'offset est à 0x30. C'est par ce biais que l'adresse du PEB peut être obtenue par programmation.

3. Assembleur x86

3.1 Registres

Le x86 est une architecture où le processeur utilise principalement des registres 32 bits afin de stocker ses informations. Chaque registre contient un nombre codé sur 32 bits, mais ce nombre peut aussi être vu comme deux nombres de 16 bits ou 4 nombres de 8 bits. Dans un souci de compréhension, nous allons illustrer ce point par un exemple.

Le nombre hexadécimal 0xC0DEBA5E est un entier 32 bits. En effet, il peut être représenté par les 32 bits suivants :

Hexadécimal	C	0	D	E	B	A	5	E
Binaire	1100	0000	1101	1110	1011	1010	0101	1110

Il peut être vu comme deux entiers sur 16 bits : 0xC0DE et 0xBA5E, ou quatre entiers sur 8 bits : 0xC0, 0xDE, 0xBA et 0x5E. Il est important de s'habituer à cette petite gymnastique, car l'assembleur fait souvent un usage abusif de ces différentes représentations.

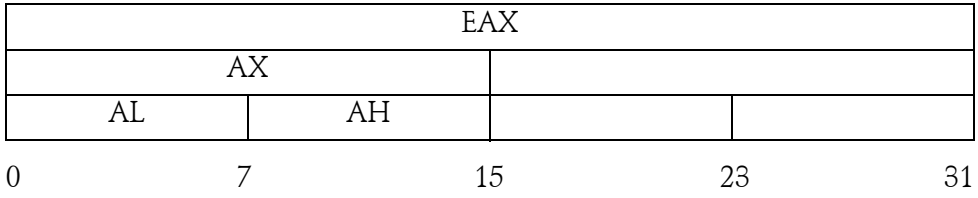
Pour faciliter les explications, on utilise communément des termes spécifiques pour distinguer ces nombres de différentes tailles. Un octet est un nombre sur 8 bits, un mot est un nombre sur 16 bits, soit 2 octets, un double est un nombre sur 32 bits, soit 2 mots ou 4 octets.

Les architectures x86 comportent principalement 16 registres différents classés en cinq types : les registres généraux, les registres d'index, les registres de pointeurs, les registres de segments et le registre de drapeaux (ou *flags*).

Registres généraux

Il existe quatre registres de ce type : EAX, EBX, ECX et EDX.

Ces registres font 32 bits et peuvent être décomposés en sous-registres plus petits. Dans ce cas, leur notation change. Voici un exemple pour EAX :



Les chiffres en bas du chemin correspondent aux bits du registre. Le E présent au debut de chaque registre correspond à *Extended*. Le 32 bits étant une extension du 16 bits, les registres ont gardé les mêmes noms, un E a simplement été ajouté devant chaque nom.

Pour l'anecdote, ces notations barbares proviennent des premières architectures 8 bits qui comportaient quatre registres généraux : A, B, C et D. Avec l'avènement des machines 16 bits, on a utilisé AX, BX, CX et DX, où le X signifie *eXtended*. Chacun de ces registres est décomposé en deux registres de 8 bits : Low pour la partie basse et High pour la partie haute, d'où les notations AL et AH. Lors du passage aux architectures 32 bits, le même mécanisme a été utilisé : on a ajouté le E de *Extended* en préfixe à tous ces noms de registres pour rester cohérent avec les notations précédentes.

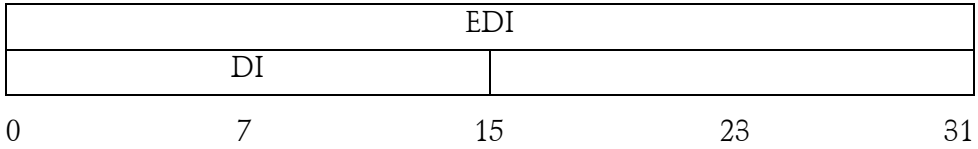
Généralement, ces registres ont une utilisation spécifique. Toutefois, attention : cette utilisation peut être modifiée et elle n'est donc pas garantie.

Le registre EAX est utilisé pour les calculs. Le registre EBX est souvent utilisé pour accéder aux tableaux. Le registre ECX est souvent utilisé comme compteur pour les boucles. EDX est utilisé en complément d'EAX afin de pouvoir stocker davantage d'informations de manière virtuelle.

Registres d'index

Il existe deux registres d'index : EDI et ESI.

Ces registres utilisent 32 bits, mais peuvent être également décomposés en sous-registres. Voici un exemple pour EDI :



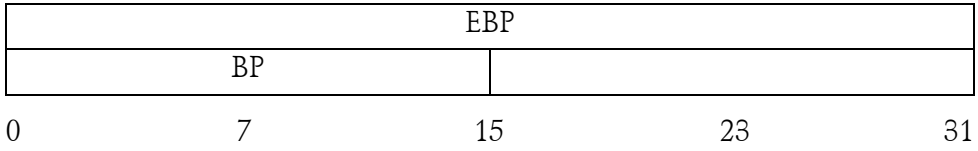
Les chiffres du bas représentent toujours le nombre de bits.

Ces registres sont normalement utilisés pour les manipulations de chaînes de caractères ou les copies mémoire, les D de EDI signifiant « destination » et le « S » de ESI signifiant source. Ainsi, lors d'une copie mémoire, on observera souvent que EDI représente le pointeur sur la zone mémoire de destination et ESI représente le pointeur sur la zone mémoire source. Il faut noter que même si cette convention est souvent respectée par les compilateurs, elle ne l'est pas toujours.

Registres de pointeurs

Il existe trois registres de pointeurs : EBP, ESP et EIP.

Ces registres peuvent également être décomposés en sous-registres. Voici un exemple pour le registre EBP :



Les chiffres représentent les bits du registre. Ces registres sont des registres particuliers. Voici l'usage de chacun :

– Le registre EBP contient une adresse sur la base de la pile, c'est-à-dire la limite entre les arguments et les variables locales. Ce registre permet généralement d'accéder aux données empilées en mémoire sur la pile (*stack*). EBP permet de récupérer des données en mémoire.
– Le registre ESP contient l'adresse courante du haut de la pile. Ce pointeur désigne la zone de la pile où copier les données pour les mettre en mémoire sur la pile (*stack*). ESP permet de stocker des données en mémoire.
– Le registre EIP contient l'adresse des instructions à exécuter.

Ces registres sont utilisés de manière implicite et ne sont généralement pas modifiés par les fonctionnalités du programme.

Registres de segments

Il existe six registres de segments : CS, DS, ES, FS, GS, SS.

Ce sont des registres de 16 bits utilisés pour stocker des valeurs.

Registre de drapeaux

Ce registre de 32 bits est appelé EFLAGS.

Certains drapeaux sont réservés par les fabricants et ne peuvent pas être utilisés, ils ont une valeur par défaut. Voici le schéma du contenu de ce registre avec le nom des drapeaux ou leur valeur réservée :

Bit	Nom du drapeau
0	CF (*Carry Flag*)
1	1
2	PF (*Parity Flag*)
3	0
4	AF (*Auxiliary carry Flag*)
5	0
6	ZF (*Zero Flag*)