DECOMPRESSER UN EXE COMPACTE/COMPRESSE AVEC UN DEBOGUEUR

Vous êtes déjà tombés sur un programme qui ne se désassemble pas ? Ou bien alors sans aucune "String Data Référence" (SDR) ? Eh bien c'est le signe que le programme est compressé, chiffré ou compacté ! Voici comment s'en sortir dans un cas simple, en utilisant un vieux debogueur : SoftIce.

Nous allons apprendre à décompresser un programme qui est passé au compresseur et ce, à la main (avec un débogueur) pour pouvoir désassembler le code, le lire et tenter de récupérer les SDR. Ce n'est pas gagné pour les SDR, mais vous obtiendrez un listing décompacté, donc lisible. Vous allez voir que dans le cas d'une simple compression, c'est facile. Ce n'est pas le cas lorsque le programme a été packé, car il comporte des protections supplémentaires.

Au programme :

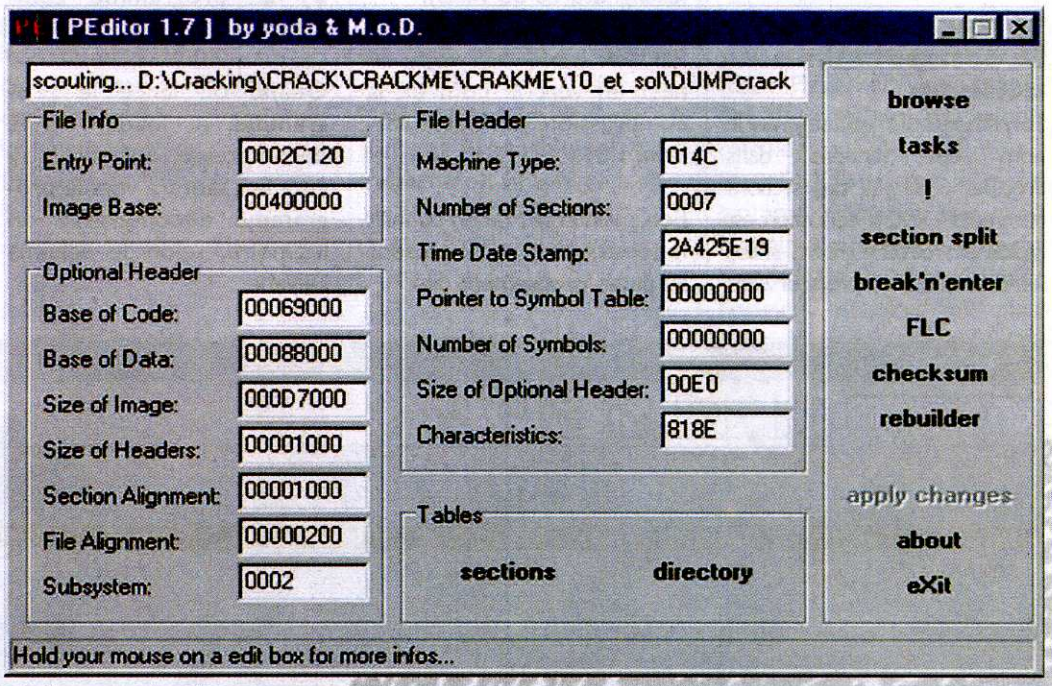
1) Rappel rapide du principe de fonctionnement des programmes compressés/compactés,

2) La décompression proprement dite : manuellement avec un débogueur.

Les outils : un éditeur de PE et un débogueur (celui de votre choix).

Avant de vous atteler à la tâche qui consiste à dumper un programme manuellement, il vous faut quelques indications :

Les informations importantes sont l'Entry Point, l'Image Base, ainsi que les caractéristiques des sections.



1. PEditor : nous donne l'entry point

Comme indiqué par un éditeur de PE, chaque section possède ses propres caractéristiques. Ce qui nous intéresse, outre l'Image base et l'Entry point (cf. prise écran 1), c'est la caractéristique de la section (cf. prise écran 2) où se trouve l'Entry point. Les caractéristiques servent à dire si l'on peut exécuter, lire ou écrire sur chaque section, d'où leur importance. Une section dont la caractéristique est à C0000040 ne peut pas être désassemblée. Il faut la mettre à E0000020 sinon votre débogueur risque de ne pas breaker sur l'Entry Point. Remettez la section à E0000020 si besoin est.

Maintenant, notre programme pourra être désassemblé. Sauf qu'il n'y a pas les SDR et que le code est compressé, donc illisible… C'est maintenant que l'on va s'occuper de dumper notre cible afin de retrouver un listing lisible au désassemblage et, si possible, les SDR.

Nous savons que Procdump

Section Table Viewer

Section	Virtual Size	Virtual Offset	Raw Size	Raw Offset	Characteristics
.text	0002C000	00001000	0002B200	00000400	E0000020
.text0	00002000	0002D000	00000C00	0002B600	C0000020
.text1	00003000	0002F000	00000600	0002C200	C0000020
.text2	00005000	00032000	00000200	0002C800	C0000020
.text3	00050000	00037000	00000800	0002CA00	C0000020
.text4	00002000	00087000	00000C00	0002D200	C0000020
.fs	0004E000	00089000	0004DC00	0002DE00	40000040

Do a right mouse click on a sectionname for more options...

2. Les sections du programme

reconnaît une bonne partie des compresseurs/compacteurs du marché et est capable de les décompresser convenablement. Ce qui nous intéresse est le fonctionnement d'un dump. Nous ferons donc ce dump manuellement.

1/ Recherche de la compression utilisée

1e méthode : On peut regarder le nom des sections dans ProcDump. S'il y a des sections nommées UPX, la déduction est rapide. On retrouve généralement l'empreinte du compresseur dans le titre de certaines sections.
2e méthode : Sinon on prend un analyseur de fichier pour connaître la compression utilisée (PeID, StudPE, etc.). De même, un éditeur hexadécimal affichera dans la marge des empreintes (un peu comme les signatures de virus).

Par exemple on voit en haut en vert le nom de la compression, UPX, et la version, 1.01.

Bon, maintenant qu'on connaît le compresseur utilisé, passons au dump proprement dit.

2/ Le fonctionnement des programmes compressés

Un programme compressé est structuré de la manière suivante :

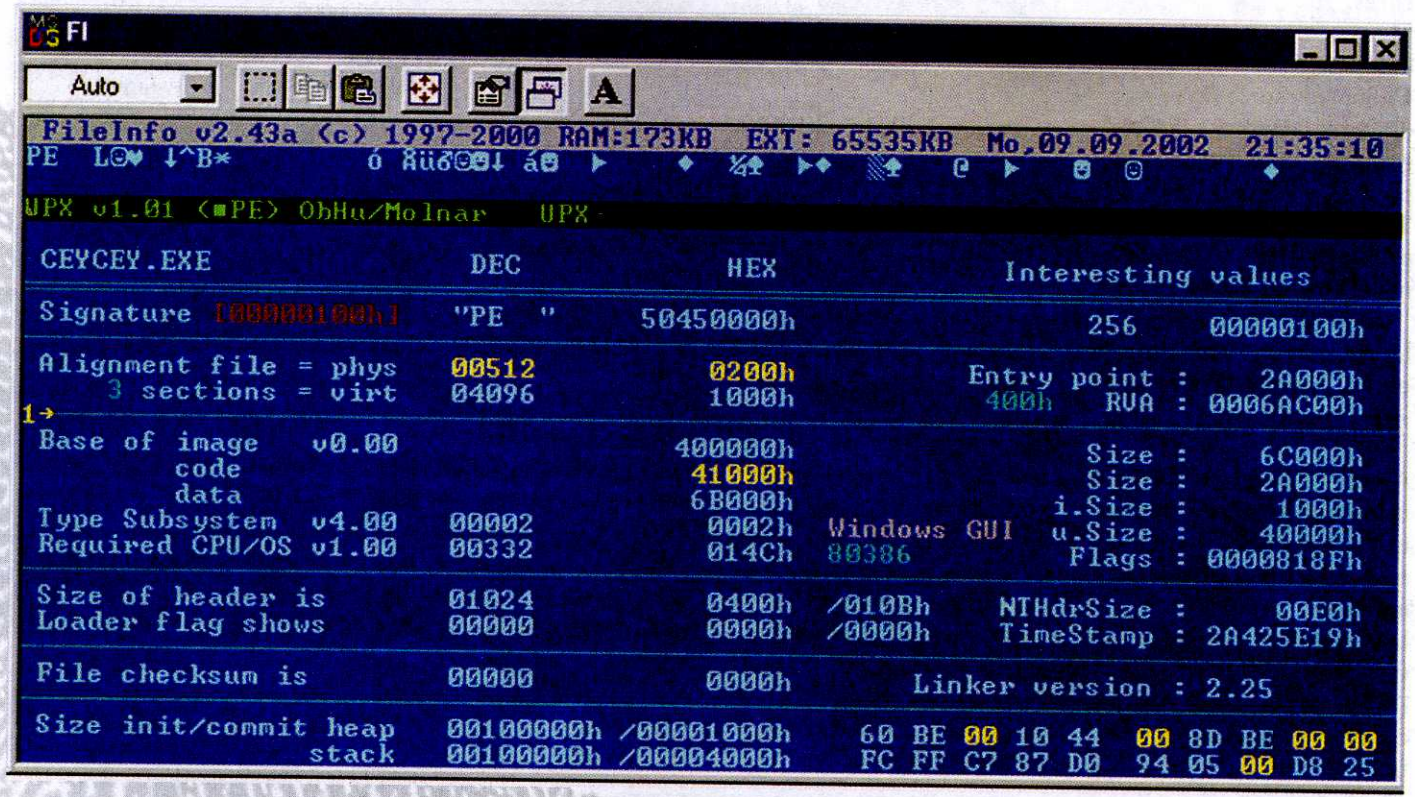
1) Les infos préliminaires et le PE,
2) Le "loader" (la routine de décompression),
3) Le programme compressé.

Quand on lance le programme, le loader va le décompresser puis, dès qu'il aura fini, sautera vers le programme décompressé en mémoire, selon le schéma suivant :

Adresse **001** : le loader décompresse le programme à l'adresse 401000,
002 : le loader continue à décompresser en 401001,
[...]
[...]
99 : le loader saute vers le programme décompressé en mémoire à l'adresse 401000,
100 : c'est la ligne 100 sur le disque dur mais on est en 401000 en mémoire au début du programme qui est maintenant décompressé.
Le binaire suit quant à lui ce schéma :
- info / PE,
- loader qui décompresse l'exe puis saute sur ce dernier (décompressé),
- le proggy décompressé démarre.

3/ LE DUMP (LE SEUL TRAVAIL REEL)

Explication théorique
Nous interviendrons juste au moment où la cible saute sur le prog décompressé en mémoire vive (à la fin de la



routine de décompression).
a) Nous allons modifier la fin du loader afin de faire boucler le programme en mémoire vive.
b) Puis nous allons faire un dump (enregistrer une image de la mémoire vive) du programme.
c) Pour finir, nous allons modifier l'Entry Point (EP) pour commencer le programme après le loader. En effet, l'EP pointe sur le loader (la routine de décompression) et si le loader s'exécute, ça va décompresser le programme qui est déjà décompressé et cela va re-modifier l'exe qui va planter. Il faut donc indiquer comme adresse pour l'EP le début du programme juste après la routine de décompression.
Comment faire ? Nous avons besoin de deux programmes : Débogueur : SOFT-ICE ou OllyDBG pour tracer le programme, la routine de décompression et ainsi trouver l'adresse de l'EP du proggy décompressé.
Editeur de PE : PROCDUMP pour analyser le PE, ou autre...

PLAN DE TRAVAIL

1/ Tracer la routine de décompression jusque au saut du proggy décompressé.
2/ Noter l'adresse du début du proggy décompressé.
3/ Faire boucler le proggy en mémoire vive.
4/ Faire un dump avec ProcDump.
5/ Fermer le programme qui tourne en boucle avec ProcDump.
6/ Modifier l'Entry Point du programme dumpé et mettre l'EP noté en 2 pour rendre le programme exécutable.

Remarque : il existe plusieurs débogueurs. Chacun a ses raccourcis mais ils ont tous les mêmes fonctionnalités principales : BreakPoint, modification du programme en mémoire vive, Step into, Step over, run. Donc, quel que soit le débogueur utilisé, la théorie permet de comprendre comment dumper un programme. Ce qui suit est donc adaptable à votre débogueur favori.

Explication pratique

Nous allons donc lancer notre débogueur et charger notre programme. Si le débogueur ne breake pas, il faut soit modifier la configuration de votre débogueur (OllyDBG permet de tracer un programme depuis le début de la routine de décompression), soit modifier la caractéristique de la section contenant l'EP. On met la caractéristique de la section correspondante à E0000020.
Maintenant notre débogueur break dès la première ligne du programme.
Notre travail commence. La routine de décompression a le schéma suivant :

```
PUSHAD
;[routine de décompression]
;[...]
POPAD
JMP 00401000
;(prog décompressé)
```

Il y a des variantes :

```
PUSHAD
;[...]
POPAD
JMP EAX
; la lecture du registre EAX vous donne l'adresse du proggy).
```

Avec ASPACK on a :

```
POPAD
JMP     xxxxxxxx
CALL    xxxxxxxx
POP     SS
RET
```

Le PUSH EAX est caché dans le CALL et le RET fait un retour sur l'adresse contenue dans EAX qui est l'EP du proggy décompressé.

Entre le PUSHAD et le POPAD, il y pas mal de chemin à tracer avec le débogueur. Ainsi, dès le break, nous allons tomber sur un PUSHAD, qui est le début de la routine de décompression. Alors on commence à tracer la cible. Quand il y a des boucles, posez des BreakPoint afin de sortir rapidement des routines. Quand on connaît le compresseur utilisé, on peut chercher le POPAD / JMP XXXX et mettre directement un BreakPoint sur le POPAD. Cela permet de passer toute la routine de décompression . C'est un cas facile (UPX), mais avec des packers, ce n'est pas aussi aisé. De plus le loader peut contenir des protections, ce qui peut obliger à tracer la routine dans son intégralité. Mais passons, les packers ne sont pas au programme. À un moment, on va arriver à :

```
JMP XXXXXXX
CALL XXXXXX
POPAD
JMP 004578F4
;(l'adresse chez moi du prog décompressé en mémoire)
```

Là vous notez l'adresse du JMP (après le POPAD), et pour arriver au JMP 004578F4, vous posez un BPX sur le POPAD (en ayant pensé a supprimer tout les BPX posés précédemment), faites avancer le programme et vous tomberez sur le POPAD.Vous enlevez le BPX, vous vous positionnez sur la ligne du JMP, et là vous écrivez dans le Soft. Transformez le saut après POPAD par JMP EIP ce qui fait boucler le programme (pour Soft-Ice) ou faites un saut qui fait boucler le proggy sur lui même en mémoire vive avec OllyDBG. Bon, vous avez noté l'adresse du proggy décompressé en mémoire, vous avez fait boucler l'exe, maintenant vous lancez ProcDump, vous faites un "refresh list" dans la fenêtre des task, vous cherchez votre prog et d'un clic droit vous faites un DUMP FULL. Vous enregistrez le prog sous un nouveau nom (par exemple DUMP.exe) puis vous faites un KILL TASK pour arrêter le prog qui tourne en boucle (ne pas oublier).
Le programme est dumpé mais n'est pas exécutable. Il faut modifier l'entry point comme suit : Nouveau EP = EP – Image BASE. (EP est celui que vous avez noté). Ex : 004578F4 – 00400000=578F4

Vous ouvrez ProcDump, vous choisissez votre prog (DUMP.exe) et vous modifiez l'EntryPoint par la valeur calculée.
Et voilà ! Vous avez un programme décompressé exécutable. Comparez la taille de l'original et du prog dumpé, l'original est trois fois plus petit. Maintenant, vous pouvez le désassembler tranquillement avec WDASM; le listing est lisible. Il vous manquera sûrement les imports, mais ce sera pour plus tard.
Entraînez-vous en téléchargeant des compresseurs puis des packeurs : UPX, Aspack, Petite..., compressez le notepad.exe et tentez de le dumper. Ou cherchez des crackme packés, ils sont légions.

Mille mercis à Fravia, Orc, Frog's Print, Christal, Mr Philex, TaMaMBoLo, Shmeita (et toute la Shmeitcorp), Iciteam, shoud, Mr Horse et ceux qui m'ont aidé dans mon apprentissage... perpétuel.

Silkscalp@gmail.com

CODEZ VOTRE PACKER

Après vous avoir expliqué comment désassembler un programme packé, nous allons nous attarder sur le fonctionnement interne d'un packer simple. Il y a non seulement le cryptage et la compression, mais surtout tout ce qui concerne le format PE. Une bonne occasion de s'y plonger.

Le packing est une méthode anti-cracking assez répandue. Les packers les plus célèbres sont ASProtect, Petite, UPX, et j'en oublie. Il y a peu de temps de cela, je m'intéressais au fonctionnement de ces packers, avec un projet en tête, celui d'en réaliser un. Je me suis attelé à la tâche, et j'ai réussi à coder un petit packer très simple, fonctionnel et léger, j'ai nommé Kassandra. Dans cet article, je vais expliquer ce qu'est un packer, son fonctionnement, ses utilisations, puis nous passerons à la programmation proprement dite. Je conclurais sur les éventails de possibilités d'amélioration de ce petit packer.

1 - Fonctionnement d'un packer

Un packer est un logiciel qui permet de réduire la taille d'un exécutable, tout en laissant cet exécutable fonctionnel. Ce genre d'outil peut être pratique pour gagner de la place, et permet donc également de gagner du temps lors d'un téléchargement par exemple. Il permet aussi de retarder d'éventuelles attaques de reverse engineering. Pour cela, il agit au sein même du programme ciblé : il fait en sorte que le code composant ce programme n'apparaisse pas de la même manière en dead-listing (c'est à dire dans un listing ASM généré par un désassembleur) qu'en mémoire. Ce qui complique assez les choses, comme nous le verrons plus loin, et nous amène à nous poser plusieurs questions : comment le packer fait-il pour rendre le code désassemblé " illisible " ? Et comment fait-il pour que ce même programme puisse quand même s'exécuter correctement ?
Le principe de fonctionnement d'un packer n'est pas très compliqué à comprendre. Dans un premier temps, il modifie le code de l'exécutable de manière à le rendre incompréhensible lors d'un désassemblage, ce qui peut être réalisé par un cryptage et par une compression du code. Il s'arrange ensuite pour qu'à l'exécution du programme, un bout de code se charge de remettre le code modifié par le packer en état, pour qu'il puisse être correctement exécuté. Tout cela est fait de manière transparente, et l'utilisateur ne peut pas déceler la présence d'un packer à l'œil nu.

Avant de rentrer dans des détails un peu plus techniques, je tiens à rappeler comment Windows lance des exécutables. Sous Windows, tout exécutable est au format PE (Portable Executable). C'est un format de fichier respectant une structure stricte et définie. Windows possède un loader qui est appelé à chaque fois que vous lancez un programme. Ce loader va charger un exécutable stocké sur disque en mémoire pour ensuite l'exécuter, c'est le mappage. C'est une étape cruciale du processus de chargement d'un exécutable PE. L'exécutable contient des informations sur la manière dont il doit être mappé en mémoire, que le loader de Windows suit à la lettre. Une fois le programme exécutable chargé en mémoire, le loader lui passe la main, et le programme chargé se lance.

Le packer va modifier l'exécutable afin de rendre son code incompréhensible. Il va lire le code machine contenu dans le programme, le crypter puis le compresser afin de réduire la taille de l'exécutable et ajoutera ensuite un bout de code qui sera exécuté au lancement du programme. Ce bout de code remettra le code exécutable crypté et compressé du programme en état, afin que celui-ci puisse être exécuté. C'est ce qu'on appelle un loader, car il aura le même objectif que celui de Windows : faire en sorte que le code chargé se lance. Bien sûr, le loader du packer sera lancé après le loader de Windows, ce qui aura pour conséquence de dérouter totalement un désassembleur/débuggueur, celui-ci affichant juste le code chargé par le loader de Windows ???. Le désassembleur n'affichera que le code crypté et compressé, ainsi que le code du loader installé par le packer. Le code " actif " du programme packé est inaccessible directement par un désassemblage. Pour le retrouver, il va falloir ruser. Au final, on obtient un exécutable plus petit que l'original, et qui retarde les attaques par reverse engineering. Pour exemple, voire les deux captures d'écran qui montrent le code d'un exécutable et son équivalent packé.

```
00612A64  55          PUSH EBP
00612A65  8BEC        MOV EBP,ESP
00612A67  83C4 F0     ADD ESP,-10
00612A6A  B8 04236100 MOV EAX,DEVPHP.00612304
00612A6F  E8 0C47DFFF CALL DEVPHP.00407180
00612A74  A1 A4726100 MOV EAX,DWORD PTR DS:[6172A4]
00612A79  8B00        MOV EAX,DWORD PTR DS:[EAX]
00612A7B  E8 B051E7FF CALL DEVPHP.00487C30
00612A80  A1 A4726100 MOV EAX,DWORD PTR DS:[6172A4]
00612A85  8B00        MOV EAX,DWORD PTR DS:[EAX]
```

R D'EXÉCUTABLES

```
00684125| DBCE          | FCMOVNE ST,ST(6)
00684127| 9D            | POPFD
00684128| DCD8          | FCOMP EAX
0068412A| DD9B B7B8D7B9 | FSTP QWORD PTR DS:[EBX+B9D7B8B7]
00684130| B1 B2         | MOV CL,0B2
00684132| BC D0D0CFCD   | MOV ESP,CDCFD0D0
00684137| CC            | INT3
00684138| CC            | INT3
00684139| CB            | RETF
0068413A| C9            | LEAVE
```

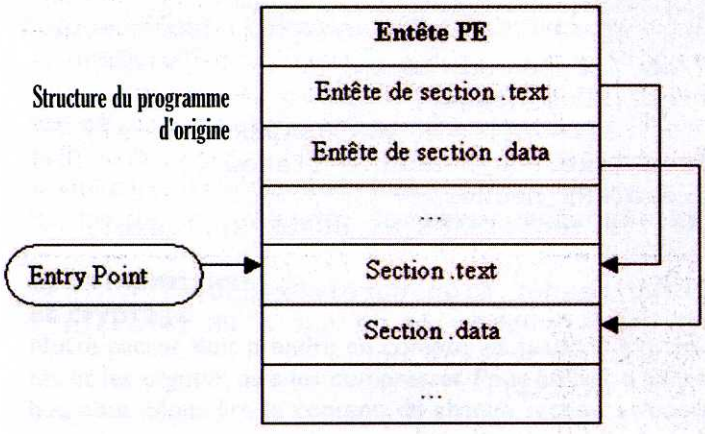
```
Illegal use of regist

Far return
```

2 - Conception du packer Kassandra

Pour concevoir mon packer, j'ai dû d'abord établir son fonctionnement. Il doit pouvoir faire trois choses :

- ajouter du code dans un exécutable,
- crypter et compresser le contenu d'un exécutable,
- assurer la fonctionnalité de l'exécutable packé.

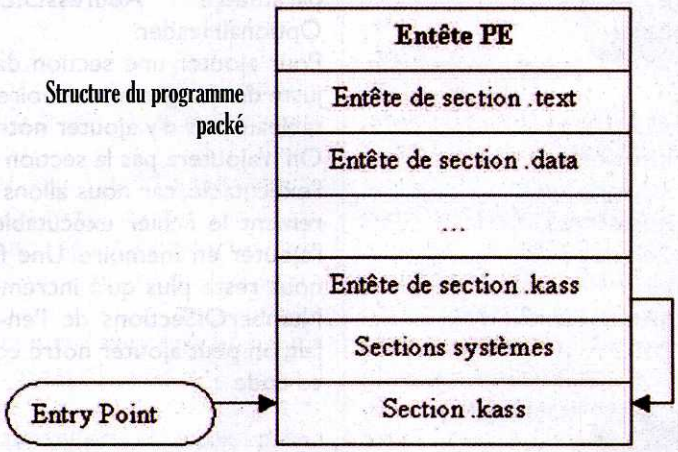
Le packer va d'abord analyser l'exécutable, puis compresser l'ensemble des sections présentes dans celui-ci, sauf les sections critiques (la section contenant la table des imports, celle contenant les ressources et celle contenant les relocations). Il réécrit ensuite un exécutable fonctionnel, en transformant les sections physiques en sections virtuelles : ces sections n'auront plus de contenu sur disque chargé dans leur espace mémoire lors du chargement du programme, mais seront justes mappées par le loader de Windows. Notre loader se chargera de décompresser leur contenu en mémoire, et de rendre l'exécutable fonctionnel. Le packer va donc devoir ajouter du code.



a) Ajout de code dans un exécutable

Pour ajouter du code dans un exécutable, on va devoir ajouter une section. Pour cela, il faut comprendre le fonctionnement du format PE dans son intégralité. Vous trouverez des informations supplémentaires, notamment au sujet des tables d'imports, dans le Manuel 11 (API hijacking).

Voici une structure d'un fichier exécutable :



DOS Header (MZ)PE Header (PE)En-tête de section .textEn-tête de section .dataEn-tête de section .rsrcSection .textSection .dataSection .rsrc

L'en-tête DOS nous importe peu. C'est surtout l'en-tête du format PE qui contient les informations utiles à la compréhension du format. L'en-tête PE est défini par la structure IMAGE_NT_HEADERS définie dans winnt.h. Voici son prototype :

```
typedef struct
_IMAGE_NT_HEADERS {
 DWORD Signature;
 IMAGE_FILE_HEADER
   FileHeader;
 IMAGE_OPTIONAL_HEADER
   OptionalHeader;
} IMAGE_NT_HEADERS,
  *PIMAGE_NT_HEADERS;
```

Cet en-tête est constitué de deux autres structures et d'une signature, qui doit valoir " PE\0 ". Le premier entête FileHeader contient les informations relatives au stockage sur fichier, et est caractérisé par la structure de type IMAGE_FILE_HEADER définie comme suit :

```
typedef struct
_IMAGE_FILE_HEADER
{
//informations sur
  la machine
 WORD Machine;
//nombre de sections
  de l'exécutable
 WORD NumberOfSections;
 DWORD TimeDateStamp;
//Adresse de la
table des symboles
 DWORD
   PointerToSymbolTable;
//Nombre de symboles
 DWORD NumberOfSymbols;
//Taille de
  l'OptionalHeader
 WORD
 SizeOfOptionalHeader;
 WORD
   Characteristics;
} IMAGE_FILE_HEADER,
 *PIMAGE_FILE_HEADER;
```

Quand le loader de Windows charge le fichier exécutable, il récupère le nombre de sections à mapper grâce au paramètre NumberOfSections de l'entête FileHeader. Le loader va

ensuite analyser l'en-tête OptionalHeader qui, lui, contient des informations sur la structure de l'exécutable, et notamment sur la manière dont il doit être monté en mémoire.

```
typedef struct _IMAGE_OPTIONAL_HEADER {
      WORD Magic;
      BYTE MajorLinkerVersion;
      BYTE MinorLinkerVersion;
      DWORD SizeOfCode;
      DWORD SizeOfInitializedData;
      DWORD SizeOfUninitializedData;
      DWORD AddressOfEntryPoint;
      DWORD BaseOfCode;
      DWORD BaseOfData;
      DWORD ImageBase;
      DWORD SectionAlignment;
      DWORD FileAlignment;
      WORD MajorOperatingSystemVersion;
      WORD MinorOperatingSystemVersion;
      WORD MajorImageVersion;
      WORD MinorImageVersion;
      WORD MajorSubsystemVersion;
      WORD MinorSubsystemVersion;
      DWORD Reserved1;
      DWORD SizeOfImage;
      DWORD SizeOfHeaders;
      DWORD CheckSum;
      WORD Subsystem;
      WORD DllCharacteristics;
      DWORD SizeOfStackReserve;
      DWORD SizeOfStackCommit;
      DWORD SizeOfHeapReserve;
      DWORD SizeOfHeapCommit;
      DWORD LoaderFlags;
      DWORD NumberOfRvaAndSizes;
      IMAGE_DATA_DIRECTORY
DataDirectory[
  IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER,
  *PIMAGE_OPTIONAL_HEADER;
```

Cet en-tête est très important pour le loader. C'est lui qui le renseigne sur la taille totale de mémoire à allouer pour y mapper l'exécutable, et c'est encore lui qui lui fournit les RVAs (Relatives Virtual Address) et tailles des différentes sections à mapper (grâce au tableau DataDirectory).
Suit ensuite ce qu'on appelle la table des sections, qui est une suite d'en-têtes de sections définissant la manière de les mapper en mémoire, ainsi que leurs caractéristiques. Un en-tête de section est défini par une structure IMAGE_SECTION_HEADER :

```
typedef struct _IMAGE_SECTION_HEADER {
 BYTE Name[IMAGE_SIZEOF_SHORT_NAME];
 union {
          DWORD PhysicalAddress;
          DWORD VirtualSize;
 } Misc;
 DWORD VirtualAddress;
 DWORD SizeOfRawData;
 DWORD PointerToRawData;
 DWORD PointerToRelocations;
 DWORD PointerToLinenumbers;
 WORD NumberOfRelocations;
 WORD NumberOfLinenumbers;
 DWORD Characteristics;
} IMAGE_SECTION_HEADER,
  *PIMAGE_SECTION_HEADER;
```

Le paramètre Name indique au loader de Windows le nom de la section à mapper. Ce paramètre peut contenir n'importe quelle chaîne, c'est juste histoire de nommer la section. Le second paramètre est généralement la VirtualSize, c'est à dire la taille virtuelle de la section, telle qu'elle sera mappée. Le loader de Windows va donc mettre à la RVA donnée par le paramètre VirtualAddress les données stockées dans le fichier à l'offset PointerToRawData, de taille SizeOfRawData, en allouant une zone mémoire de VirtualSize octets. Le paramètre SizeOfRawData doit être un multiple de FileAlignment, et le paramètre VirtualAddress un multiple de SectionAlignment (donnée par l'en-tête OptionalHeader). Une fois cela fait, le loader passe la main au programme en jumpant sur son EntryPoint, donné par le paramètre AddressOfEntryPoint de l'en-tête OptionalHeader.
Pour ajouter une section dans un exécutable, il nous suffit juste de charger en mémoire les sections existantes, dans un tableau, puis d'y ajouter notre section que l'on paramètrera. On n'ajoutera pas la section directement en réécrivant dans l'exécutable, car nous allons par la suite reconstruire entièrement le fichier exécutable. Nous nous contenterons de l'ajouter en mémoire. Une fois la section paramétrée, il ne nous reste plus qu'à incrémenter d'une unité le paramètre NumberOfSections de l'en-tête FileHeader. Une fois cela fait, on peut ajouter notre code. Nous réalisons ceci grâce à ce code :

```
//on déclare une structure DOS_HEADER :
IMAGE_DOS_HEADER Dos_header;
//ainsi qu'une structure PE
IMAGE_NT_HEADERS Nt_header;
//et un pointeur sur tableau de sections
IMAGE_SECTION_HEADER *tab_sections;

[...]

//on lit les en-têtes
fread(&Dos_header,
      sizeof(IMAGE_DOS_HEADER),1,exe);
fseek(exe,Dos_header.e_lfanew,0);
fread(&Nt_header,
      sizeof(IMAGE_NT_HEADERS),1,exe);

if(Nt_header.Signature!=0x4550) {
 printf("<ERROR> %s is not a PE file !\n"
        ,argv[1+nb_opt]);
      exit(1);
}
```

```
//puis les descripteurs de sections
nb_sections=
  Nt_header.FileHeader.NumberOfSections;

//on alloue :
tab_sections=
      malloc(sizeof(IMAGE_SECTION_HEADER)
            *(nb_sections+1));
tab_loading=malloc(sizeof(LOADER_DATA)
                  *(nb_sections+1));
printf("Starting packing of %s\n",
      argv[nb_opt+1]);

if(verbose==1){
 printf("\nSections already present :\n");
 printf("--Name--    ---RVA-- --ROff-- -
      -Rsize- --Vsize- --Flags---\n");
}

for(a=0;a<nb_sections;a++) {
 fread(&tab_sections[a],
      sizeof(IMAGE_SECTION_HEADER),
      1,exe);
}
[...]

//on ajoute ensuite notre section .kass
strncpy(tab_sections[nb_sections].Name,
      ".kass",8);
tab_sections[nb_sections].Misc.VirtualSize=
      RoundAddress(0,CodeBase);
tab_sections[nb_sections].VirtualAddress=
      FindNextRVA();

//on ne connaît pas encore la taille :
tab_sections[nb_sections].SizeOfRawData=
   0;
tab_sections[nb_sections].PointerToRelocations=
   0;
tab_sections[nb_sections].PointerToLinenumbers=
   0;
tab_sections[nb_sections].NumberOfRelocations=
   0;
tab_sections[nb_sections].NumberOfLinenumbers=
   0;
tab_sections[nb_sections].Characteristics=
   0xC0000020;

nb_sections+=1;
[...]
//on met à jour les headers
Nt_header.FileHeader.NumberOfSections+=1;
```

b) Compression et cryptage

Notre packer doit prendre en compte les sections existantes et les crypter, puis les compresser. Pour arriver à notre but, nous allons lire le contenu de chaque section et compresser ces sections dans un espace mémoire alloué spécialement. On cryptera ensuite le résultat de ces compressions, de manière à rendre le code illisible. Pour la compression, on utilise un algorithme appelé RLE (pour Run Length Encoding) qui n'est pas vraiment performant, mais sera utile. Pour le cryptage, j'ai opté pour un simple xor, pas plus compliqué. Je ne compte pas faire un packer digne d'upx ou autre, mais simplement créer un petit packer fonctionnel. Dans l'idéal, il est conseillé de compresser avant de crypter, afin de supprimer des motifs ou des indices qui pourraient aider le cryptanaliste. Cependant, vu que la clé est de toute façon présente dans l'exécutable, et que l'algorithme de cryptage n'est pas exceptionnel, cela n'a pas d'importance ici.

Voici le code en C se chargeant de la compression. Il est un peu complexe à comprendre, c'est pourquoi je l'explicite par la suite.

```
// CompressionRLE ]*******************
*
int RLE(char *buffer,char *output,
        long taille){
 long pos_in,pos_out,count; //divers
 char car[2];    //tableau de deux caractères
 char temp[2];

 pos_in=1;
 pos_out=0;
 car[0]=buffer[0];
 car[1]=buffer[1];

 while(pos_in<(taille+1)) {

  if(car[0]==car[1]) { // deux chars égaux
     count=0;
      //pos_in++;
      output[pos_out++]=car[0];
      while(car[0]==car[1]
            && (pos_in+count)<taille) {
       count++;
       if(count<255) { //255 ensemble max
         car[1]=buffer[pos_in+count];
       }
       else {
         output[pos_out++]=car[0];
         output[pos_out++]=count;
         pos_in+=count;
         count=1;
       }
      }

      if(count!=0) {
        output[pos_out++]=car[0];
        output[pos_out++]=count;
        pos_in+=count;
        //on lit 2 cars
        car[0]=buffer[pos_in];
        car[1]=buffer[++pos_in];
      }
```

```
if(RS_id>=0)
{
  original_RS=
   tab_sections[RS_id].PointerToRawData;
  tab_sections[RS_id].PointerToRawData=
   Start;

  Start=
    RoundAddress(Start +
                 tab_sections[RS_id]
                 .SizeOfRawData,
                 FileBase);
}
if(RE_id>=0)
{
 original_RE=
  tab_sections[RE_id].PointerToRawData;

  tab_sections[RE_id].PointerToRawData=
   Start;
  Start=
   RoundAddress(Start+
                tab_sections[RE_id]
                .SizeOfRawData,FileBase);
}
[...]
//on reparamètre toutes les sections en
virtuel
for(a=0;a<nb_sections-1;a++)
{
  if(a!=IT_id && a!=RS_id && a!=RE_id){
  // propriétés
  tab_sections[a].Characteristics=
     0xC0000040;
  // aucune donnée provenant du fichier
  tab_sections[a].SizeOfRawData=0;
  tab_sections[a].PointerToRawData=0;
  }
  else
  {
  tab_sections[a].PointerToRawData+=
     padding; //si décalage alors on
              décale
  }
}
```

Le packer définit les sections comme étant virtuelles en paramétrant leur taille physique et le pointeur sur données à zéro. On évite bien sûr de toucher aux sections critiques. Une fois cela fait, nous sommes prêts pour reconstruire l'exécutable, qui deviendra l'exécutable packé. Il ne nous manque qu'une seule chose : le loader. Le loader est constitué comme ceci :

DécrypteurDécompresseur de la section .text...Décompresseur de la n-ème section Jumper

Le décrypteur va simplement décrypter les données avant que le reste du code ne les décompresse. Voici le code assembleur employé :

```
PUSHAD
XOR EAX,EAX
XOR EBX,EBX
MOV ECX,StartOfData
Loop:
MOV AL,BYTE PTR DS:[ECX+EBX]
XOR EAX,EBX
XOR EAX,Clef
MOV BYTE PTR DS:[ECX+EBX],AL
INC EBX
CMP EBX,Taille
JNZ Loop
```

Le paramètre StartOfData correspond à l'adresse de la section .kass où les données compressées et cryptées ont été écrites. Le paramètre Clef est simplement la clef de cryptage utilisée par le packer pour crypter les données, qui doit être renseigné afin que le loader puisse décrypter convenablement les données. Le paramètre taille quant à lui précise la taille des données à décrypter. L'instruction PUSHAD au tout début du code permet de sauvegarder l'état des registres de pile notamment, afin de pouvoir les restituer juste avant de sauter sur la première instruction du programme original. Le décrypteur est suivi d'une série de décompresseurs, qui vont se charger de mapper le contenu réel de chaque section dans l'espace mémoire. Le code assembleur contient entre autres un décompresseur RLE.

```
MOV EAX,StartOffset
MOV ECX,EndOffset
MOV ESI, VirtualAddress
MOV DL,BYTE PTR DS:[EAX]

ReadCar:
INC EAX
MOV BL,BYTE PTR DS:[EAX]
CMP BL,DL
JNZ SingleCar
MOV BYTE PTR DS:[ESI],BL
INC EAX
MOV BL,BYTE PTR DS:[EAX]
NOP

RepeatCar:
CMP BL,1
JE NextCar
INC ESI
MOV BYTE PTR DS:[ESI],DL
DEC BL
JMP RepeatCar

SingleCar:
MOV BYTE PTR DS:[ESI],DL
MOV DL,BL

NextCar:
INC ESI
CMP EAX,ECX
JL SHORT ReadCar
```

Chaque décompresseur prend en paramètre trois informations : StartOffset, qui correspond à l'adresse où se trouvent les données à décompresser, EndOffset, qui correspond à l'adresse où se terminent les données à décompresser et enfin VirtualAddress, qui correspond à l'adresse virtuelle où seront mappées les données. Le reste du code étant le module de décompression. Pour chaque section compressée, un module est créé.

Le loader se termine avec le jumper, qui repasse la main au programme original :

```
POPAD
MOV EAX,OEP
JMP EAX
```

Le jumper prend juste un paramètre : l'OEP (Original Entry Point). L'instruction POPAD permet de réinitialiser les registres.

On doit faire en sorte que ce soit le code de notre section qui soit exécuté en premier, ce qui est réglé en modifiant le paramètre AddressOfEntryPoint de l'OptionalHeader par ce petit bout de code :

```
//mise à jour des en-têtes
Nt_header.OptionalHeader
  .AddressOfEntryPoint =
    tab_sections[nb_sections -1]
      .VirtualAddress;
```

c) Réécriture de l'exécutable

Il ne nous reste plus qu'à réécrire le fichier exécutable :

```
//********************
// Réécriture de l'exécutable
//********************
result=fopen(argv[nb_opt+2],"w+b");
if(result) {
  //on écrit les en-têtes nt et dos
  fwrite(&Dos_header,
         sizeof(IMAGE_DOS_HEADER),
         1,result);
  fseek(result,Dos_header.e_lfanew,0);

  fwrite(&Nt_header,
         sizeof(IMAGE_NT_HEADERS),
         1,result);

}

//on écrit ensuite les en-têtes des
  sections
  for(a=0;a<nb_sections;a++)
    fwrite(&tab_sections[a],
           sizeof(IMAGE_SECTION_HEADER),
           1,result);
```

```
// sections critiques
if(IT_id>=0) {
  buffer= malloc(
    tab_sections[IT_id].SizeOfRawData
);

  fseek(exe,original_IT,0);
  fseek(result,
        tab_sections[IT_id].PointerToRawData,
        0);
  fread(buffer,
        tab_sections[IT_id].SizeOfRawData,
        1,exe);
  fwrite(buffer,
         tab_sections[IT_id].SizeOfRawData,
         1,result);
  free((void *)buffer);
}

if(RS_id>=0) {
 buffer= malloc(
  tab_sections[RS_id].SizeOfRawData
);
 fseek(exe,original_RS,0);
 fseek(result,
  tab_sections[RS_id].PointerToRawData,
  0);

 fread(buffer,
  tab_sections[RS_id].SizeOfRawData,
  1,exe);

 fwrite(buffer,
        tab_sections[RS_id].SizeOfRawData,
        1,result);

 free((void *)buffer);
}

if(RE_id>=0) {
  buffer= malloc(
    tab_sections[RE_id].SizeOfRawData
);

  fseek(exe,original_RE,0);
  fseek(result,
    tab_sections[RE_id].PointerToRawData
    ,0);
  fread(buffer,
    tab_sections[RE_id].SizeOfRawData,1
    ,exe);
  fwrite(buffer,
    tab_sections[RE_id].SizeOfRawData,1
    ,result);

  free((void *)buffer);
}
```

```
fseek(result,
  tab_sections[nb_sections-1].PointerToRawData,
  0);

//on écrit le loader
fwrite(sreg,1,1,result);

//on écrit le décrypteur
  (plusieurs parties)
fwrite(cypher1,5,1,result);

Start=
    loader_size * nb_comp_sections
    + jumper_size + cypher_size
    + tab_sections[nb_sections-1]
        .VirtualAddress
    + Nt_header.OptionalHeader.ImageBase;

fwrite(&Start,sizeof(long),1,result);
fwrite(cypher2,7,1,result);
fwrite(&clef,1,1,result);
fwrite(cypher3,6,1,result);
fwrite(&compressed,
       sizeof(long),1,result);
fwrite(cypher4,2,1,result);


//on écrit un décompresseur par section
for(a=0;a<nb_sections-1;a++) {

  if(a!=IT_id && a!=RS_id && a!=RE_id){
    fwrite(loader1,1,1,result);

    tab_loading[a].StartOffset +=
        Nt_header.OptionalHeader.ImageBase
       + loader_size*nb_comp_sections
       + jumper_size+cypher_size;
    fwrite(&tab_loading[a].StartOffset,
        sizeof(long),1,result);
    fwrite(loader2,1,1,result);

    tab_loading[a].EndOffset+=
     Nt_header.OptionalHeader.ImageBase
     + loader_size*nb_comp_sections +
     jumper_size+cypher_size;
    fwrite(&tab_loading[a].EndOffset,
      sizeof(long),1,result);
    fwrite(loader3,1,1,result);

    tab_loading[a].VirtualAddress +=
     Nt_header.OptionalHeader.ImageBase;
    fwrite(&tab_loading[a].VirtualAddress,
         sizeof(long),1,result);

    fwrite(loader4,36,1,result);
  }
}

//et le jumper
fwrite(jumper1,2,1,result);
fwrite(&OEP,sizeof(long),1,result);
fwrite(jumper2,2,1,result);

//puis on écrit les données compressées
fwrite(memory_map,compressed,1,result
);
//et on met du padding
for(a=ftell(result);
   a<tab_sections[nb_sections-1]
       .SizeOfRawData
     + tab_sections[nb_sections-1]
       .PointerToRawData;a++)
 {
  fwrite("\x00",1,1,result);
 }
```

Les codes correspondant au loader, décrypteur et décompresseur sont découpés, afin de pouvoir stocker facilement des adresses 32 bits. On obtient au final un exécutable fonctionnel, qui se décrypte et se décompresse tout seul comme un grand. Alors, c'était si compliqué ?

3 - Exemple d'utilisation

Nous allons essayer notre packer sur un programme de Windows, j'ai nommé le bloc-note ?. Pour packer, c'est relativement simple. Il suffit de taper cette commande sous DOS : kassandra –v notepad.exe notepad_packed.exe

Voici ce que nous affiche le programme.

```
Kassandra Study Packer :: copyleft Virtualabs 2004 ::
Starting packing of notepad.exe

Sections already present :
--Name--    ---RVA-- --ROff-- --Rsize- -
-Vsize- --Flags---
 .text : 1000 1000 4000 3efc [60000020]
 .data : 5000 5000 1000 83c [c0000040]
 .idata :6000 6000 1000 e22 [40000040]
 .rsrc : 7000 7000 6000 6000 [40000040]
 .reloc : d000 d000 1000 aae [42000040]

Original Entry Point : 4010cc
Size of Image : e000
Import Table found : .idata (2)
Resources Table found : .rsrc (3)
Import Table starts at 1000 and ends at
2000
Resource Table starts at 2000 and ends
at 8000
Section .kass added at offset 8000
Compression done
Encryption done
notepad.exe was correctly packed to
notepad2.exe
```

On peut voir que le packer a détecté toutes les sections, ainsi que l'entry point et la taille de l'image. Il affiche aussi les informations concernant les sections critiques (.idata et

.rsrc). L'ajout de notre section .kass s'est effectué sans encombre, ainsi que l'opération de compression et de cryptage. Regardons désormais la taille du nouvel exécutable : 53 248 octets. Alors que l'original en faisait 57 344 ! On a gagné en place, et si on lance l'exécutable, on peut voir qu'il marche impeccablement.

4 - Améliorations possibles

Le packer que je vous présente ici est rudimentaire, et le code en C et en ASM est sûrement optimisable. J'ai juste évité un point, qui fait aussi l'attrait des packers : l'anti-dumping. L'anti-dumping est l'ensemble des méthodes employées par un programme packé pour empêcher un dump mémoire. Il y a plusieurs techniques connues : effacement du PE header, destruction de l'IAT, obfuscations diverses… Je ne m'étendrais pas dessus, vu que c'est la première version de mon packer. Je vais quand même m'arrêter sur les techniques de destruction d'IAT, car elles valent le coup d'être précisées.

L'IAT est la table permettant au programme d'appeler des fonctions externes. Cette table est complétée au lancement du programme par le loader de Windows, qui inscrit pour chaque référence à une fonction d'une librairie externe l'adresse mémoire de cette fonction. C'est pour cela que la première version de Kassandra ne touchait pas la section contenant cette table. Cela dit, il y a moyen d'empêcher le fonctionnement d'un dump : la destruction de l'IAT. Si on compresse comme les autres sections la section contenant la table des imports, le loader de Windows ne pourra plus la compléter, et lors d'un désassemblage, elle sera incomplète. Un bon point pour nous : lors d'un dump, cette table ne sera pas fonctionnelle. Par contre on ne peut pas directement lancer le programme si l'IAT n'est pas correcte : on va devoir inclure un module de correction de l'IAT dans notre loader, qui ira la modifier directement en mémoire.
Des variantes sont alors applicables : au lieu de mettre l'adresse de la fonction référencée, on peut ajouter dans notre section une fonction émulant la fonction appelée, ou encore protégeant l'adresse réelle par cryptage interposé. De toute manière, il faut empêcher un cracker de trouver l'adresse réelle de la fonction. Quand on fait du manual unpacking, c'est l'étape la plus embêtante. La reconstruction de l'IAT peut être automatisée, par exemple avec ImpRec. Mais quand on utilise du cryptage, ça devient plus complexe pour l'attaquant. Bref, il existe une panoplie de techniques possibles.

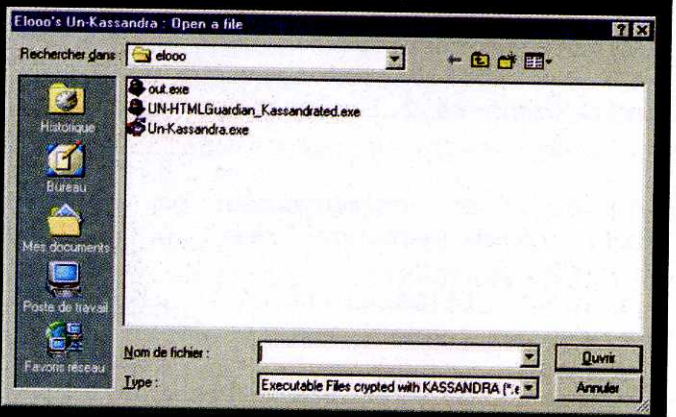
Autre point, j'ai effectué bon nombre de tests sur ce packer, mais il n'est pas exempt de bugs. Si un programme packé plante, c'est sûrement dû au packer.

Conclusion

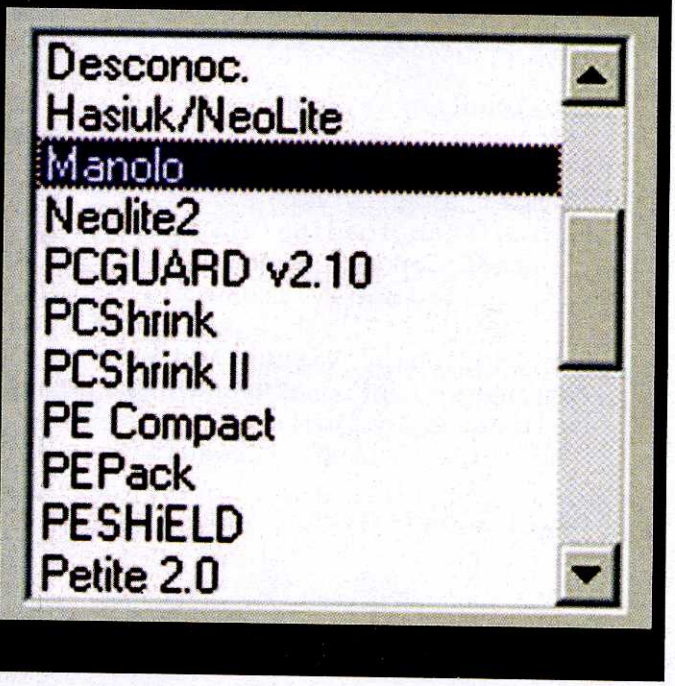
Le codage d'un packer prend pas mal de temps, mais reste toutefois à la portée de tous. Cependant, je tiens à mettre quand même un bémol : le format PE est parfois obscur, et sa découverte est toujours fastidieuse. Mais ceux qui le connaissent pourront sans problème implémenter de nouvelles fonctions et méthodes. Je ne donne qu'une base de départ, un code qui m'a pris quelque temps à réaliser. Il n'est d'ailleurs pas exempt de bugs, donc si vous en trouvez n'hésitez pas à m'en faire part. Pour conclure, je vous conseille d'aller visiter ces quelques pages :
http://Kharneth.free.fr/_Tutorials_/Yoda's_crypter.html [analyse d'un crypteur d'exécutable]
http://unpacking.free.fr [Manual Unpacking et packing en général]
Je remercie Kharneth pour son aide précieuse, sa patience et son humour, b0l0k pour les améliorations apportées au code, ma puce qui me supporte tous les jours, et tous ceux que j'oublie…

By virtualabs

procdump.png :
Le packer de cet article ne met en oeuvre aucune protection particulière. Il existent de nombreux autres packers, plus ou moins efficaces. On peut voir dans cette capture un extrait de la liste des packers connu et réversibles grâce à l'utilitaire ProcDump.



Depuis la première publication de cet article, Elooo a conçu le unpacker qui va avec. Dispo à cette adresse : http://elooo.fff.free.fr/tools/Un-Kassandra.zip.



```
        else {
            output[pos_out++]=car[0];
            car[0]=car[1];
            car[1]=buffer[++pos_in];
        }
    }

  //output[pos_out]=0;
  return pos_out+1;
}

//*********************************
//fonction de cryptage de tampon
//*********************************
int Crypte(char *tampon, long taille,
           char clef) {
    long a;
    for(a=0;a<taille;a++) {
        tampon[a]=tampon[a]^a;
        tampon[a]=tampon[a]^clef;
    }
    return 1;
}

//*********************************

[...]

b=0;
nb_comp_sections=0;

// pour chaque section
for(a=0;a<nb_sections-1;a++) {

 // S'il n'y a pas de section critique
 if(a!=IT_id && a!=RE_id && a!=RS_id) {
  nb_comp_sections+=1;

  // on alloue
  buffer=malloc(
    tab_sections[a].SizeOfRawData)
;
  // on lit la section
  fseek(exe,
   tab_sections[a].PointerToRawData,0);

  fread(buffer,
   tab_sections[a].SizeOfRawData,1,exe
);
  // on compresse en mémoire
  OSize=RLE(buffer,memory_map+b,
            tab_sections[a].SizeOfRawData);
  OSize-=1;
  tab_loading[a].StartOffset=
            b + tab_sections[nb_sections
            -1].VirtualAddress;
  tab_loading[a].EndOffset=
    tab_loading[a].StartOffset+OSize-1;
  tab_loading[a].VirtualAddress=
    tab_sections[a].VirtualAddress;
  b+=OSize;
  free((void *)buffer);
 }
}
compressed=b;
printf("Compression done\n");

//on crypte les données compressées
Crypte((char *)memory_map,mem_size,clef);
printf("Encryption done\n");
```

Le principe de la compression RLE est simple. Si un code se répète, on compte le nombre de fois qu'il apparaît et on inscrit ce nombre. Par exemple, si on a la chaîne aaabbbbbaaaaa, ça donne : a3b5a5. Une variante est employée ici. Pour distinguer un code répété, on l'inscrit deux fois, et on le fait suivre de son nombre d'apparitions. Toutefois, vu que ce nombre d'apparitions est stocké sur un octet, nous sommes limités à 256 apparitions. En cas de dépassement, on doit noter une autre fois le caractère répété suivi du nombre de fois qu'il doit apparaître, et ce jusqu'à ce que le compteur soit inférieur à 256. Cela peut donner des fois : a[255]a[3] si la lettre apparaît 258 fois successivement. Pour le cryptage xor, c'est vraiment sommaire ???.

Notez bien que l'on ne crypte pas et que l'on ne compresse pas toutes les sections ! Nous devons faire attention à trois sections principales, que le loader doit obligatoirement trouver si elles sont présentes dans le DataDirectory : la section des imports, celle des ressources (pour que l'icône associée ua programme apparaisse), et celle des relocations. Ces trois sections sont définies respectivement dans le DataDirectory par les index 1,2 et 9.

Le packer reconstruit ensuite l'exécutable, en recalculant les offsets des sections à garder, et en modifiant les caractéristiques des sections compressées. Ces sections deviendront purement virtuelles, et n'auront plus de contenu associé dans le fichier exécutable. Ce code permet de réécrire le début de l'exécutable, et de recalculer les sections :

```
//on reparamètre les pointeurs sur data
de idata et rsrc et reloc

if(IT_id>=0)
{
 original_IT=
   tab_sections[IT_id].PointerToRawData;

 tab_sections[IT_id].PointerToRawData=
   Start;
 Start=
   RoundAddress(Start
                +tab_sections[IT_id]
                .SizeOfRawData,FileBase);
}
```

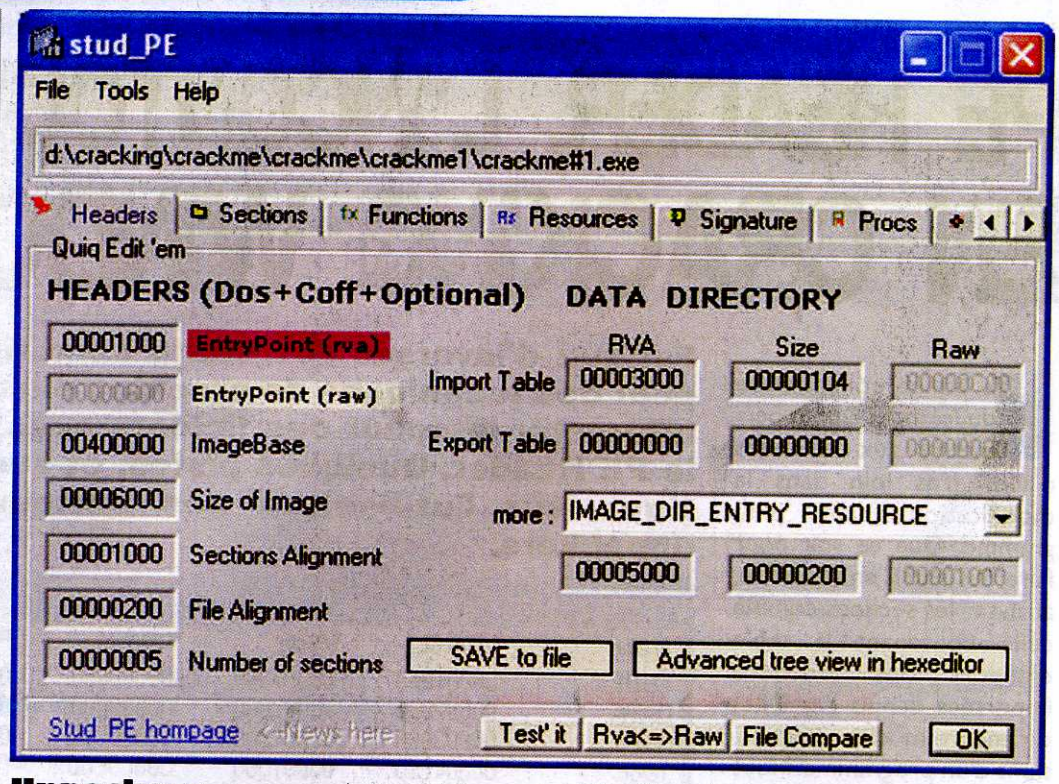
WILD

l'article précédant. Vous vous rappelez sans doute que nous cherchions un endroit à modifier pour compenser les différences de la somme de contrôle. Les fins de section sont toutes trouvées. C'est également un endroit commode pour rajouter une petite portion du code (pour modifier le programme lors de son exécution, par exemple), sur lequel on peut se brancher depuis l'intérieur du programme.

Informations générales

La deuxième prise d'écran présente ce qu'un éditeur de PE classique montre en premier, c'est-à-dire des infos comme l'image base, l'EntryPoint, la size of image, le header size (qui en général est toujours de 400h). Ici on peut relever des information comme le RVA (valeur 1000, en rouge) qui signifie Relative Virtual Adresse , l'ImageBase ,l'entryPoint, le nombre de section (5) et d'autre informations. Pour vous donner un exemple : pourquoi la majorité des programme non packé commence en 401000 (ou du moins leur code, data comprises) quand on les désassemble ?

En fait, cela vient du PE header. Le PE loader charge l'exe en mémoire grâce au infos données par le PE et ici le PE indique que la plage mémoire réservée au programme commence en 400000 (l'Image base comme par hasard) , additionné a l'EntryPoint (1000) : 400000 + 1000 = 401000 soit l'adresse du début du programme quand on le désassemble. Ce n'ai pas un hasard et ces valeurs sont très importantes pour arriver a une bonne décompression d'un programme compressé.

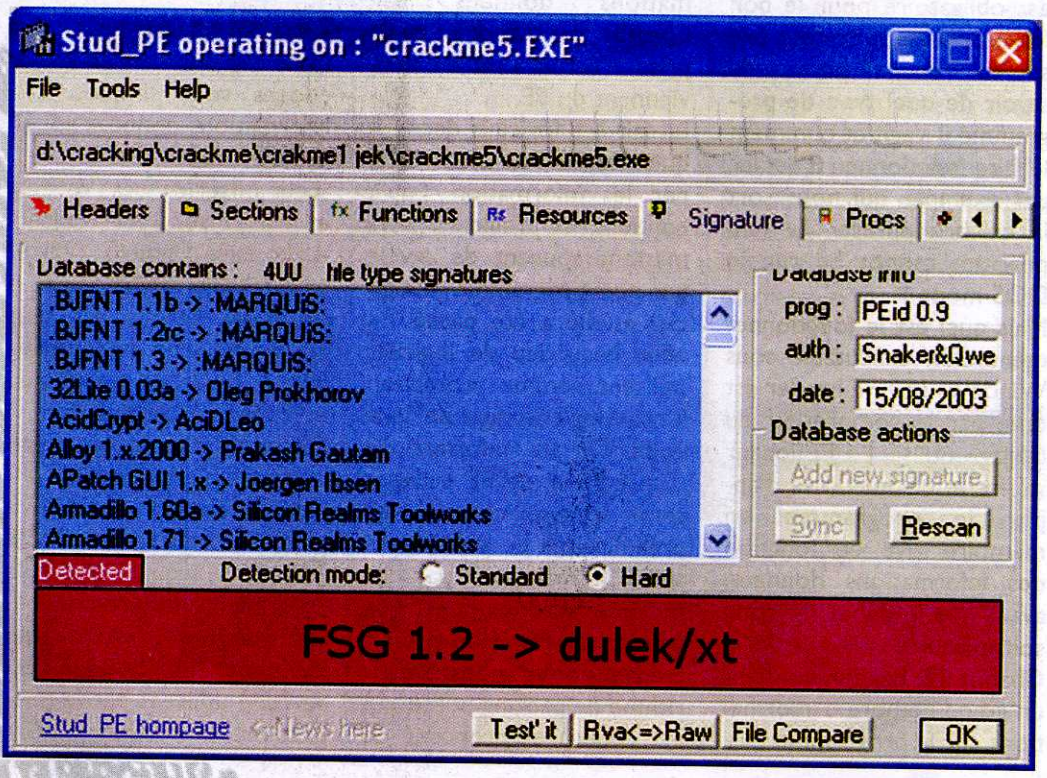


Unpackers

Ici on voit que l'éditeur de PE peut détecter le type de compression utilisé pour protéger le programme (parmi un grand nombre d'algorithmes), présentement FSG1.2 par dulek/xt. Pe-scan ou StudPE en sont capable (même si pe-scan donne juste la compression sans faire éditeur de PE). C'est aussi le cas de ProcDump, qui permet également d'ajouter de nouveaux format de compression, grâce à un langage de script. En effet, lors qu'un nouvelle technique est rencontrée, le cracker doit décompresser le programme à la main (avec un débugger pour les plus simples). Le langage de script permet d'automatiser le processus, au cas où cette technique est à nouveau rencontrée.

Nous n'avons fait que survoler les possibilités offertes par la manipulation du PE header. À vous de les expérimenter.

By : silkscalp



PE HEADER, L'INCONTOURNABLE EN CRACKING WINDOWS

Fatigué d'inverser des sauts conditionnels ? En route pour des techniques de cracking plus avancées, faisant intervenir le coeur des fichiers exécutables Windows : le PE Header. Quelques pistes, et surtout un guide pratique sur l'utilisation des éditeurs PE, et autres unpackers.

Un bonne connaissance du format PE peut permettre aux reversers d'aller très loin dans la modification d'un programme. Que ce soit pour faire des patch en mémoire, analyser les sections comme celle contenant la table d'import (les fonctions importées depuis des DLL) ou encore modifier des sections d'un programme. Nous allons dans cet article essayer de comprendre ce qu'est le PE, quelles sont les informations qu'il fournit et en quoi ces informations peuvent être utiles à un cracker.

Le sigle PE signifie Portable Exécutable. C'est une entête reconnue par toute les plateformes Windows. Le PE est obligatoire pour le bon fonctionnement de tout exe sur Windows : il permet de savoir de quel type de programme il s'agit et comment le faire fonctionner. C'est lui, par exemple, qui nous sort la célèbre phrase : « This program cannot be run in DOS mode ».

Mais quel est le fonctionnement d'un exécutable sous Windows ? Le PE header est crée par le compilateur du langage utilisé. Le PE loader va lire le PE, puis il va mapper le programme en mémoire vive en fonction des informations données, avant que le début du code soit exécuté.

Chaque PE header contient un grand nombre d'informations sur le programme qu'il décrit, dont je vais, dans cette introduction, parler du principal. Pour vous montrer la mesure des informations contenues par le PE header, sachez que toutes les informations données par LordPE ou StudPE (voir prise d'écran 1 et 2) proviennent du PE.

En fait ces logiciels décortiquent le PE pour nous donner les information qu'il contient, et d'ailleurs permettent souvent de savoir par quel programme le logiciel étudié a été packé. En effet, beaucoup de logiciels utilisent comme protection le packing/cryptage de leur code. Or les informations d'un éditeur de PE peuvent servir à passer ces mesures préliminaires de protéction, soit à la main, soit par des moyens automatiques. Les éditeur de PE élaboré vont trouver pour nous l'algorithme de packing utilisé (s'il est connu) et proposer de décompresser le programme.

Je vais vous montrer avec quelques prises d'écrans les informations que peut donner un éditeur de PE.

[Section Table]

Name	VOffset	VSize	ROffset	RSize	Flags
CODE	00001000	00001000	00000600	00000200	60000020
DATA	00002000	00001000	00000800	00000400	C0000040
.idata	00003000	00001000	00000C00	00000200	C0000040
.reloc	00004000	00001000	00000E00	00000200	50000040
.rsrc	00005000	00001000	00001000	00000200	50000040

Les séctions

Dans cette prise d'écran les choses sont simples. Les informations données sont :

-Le nom des sections

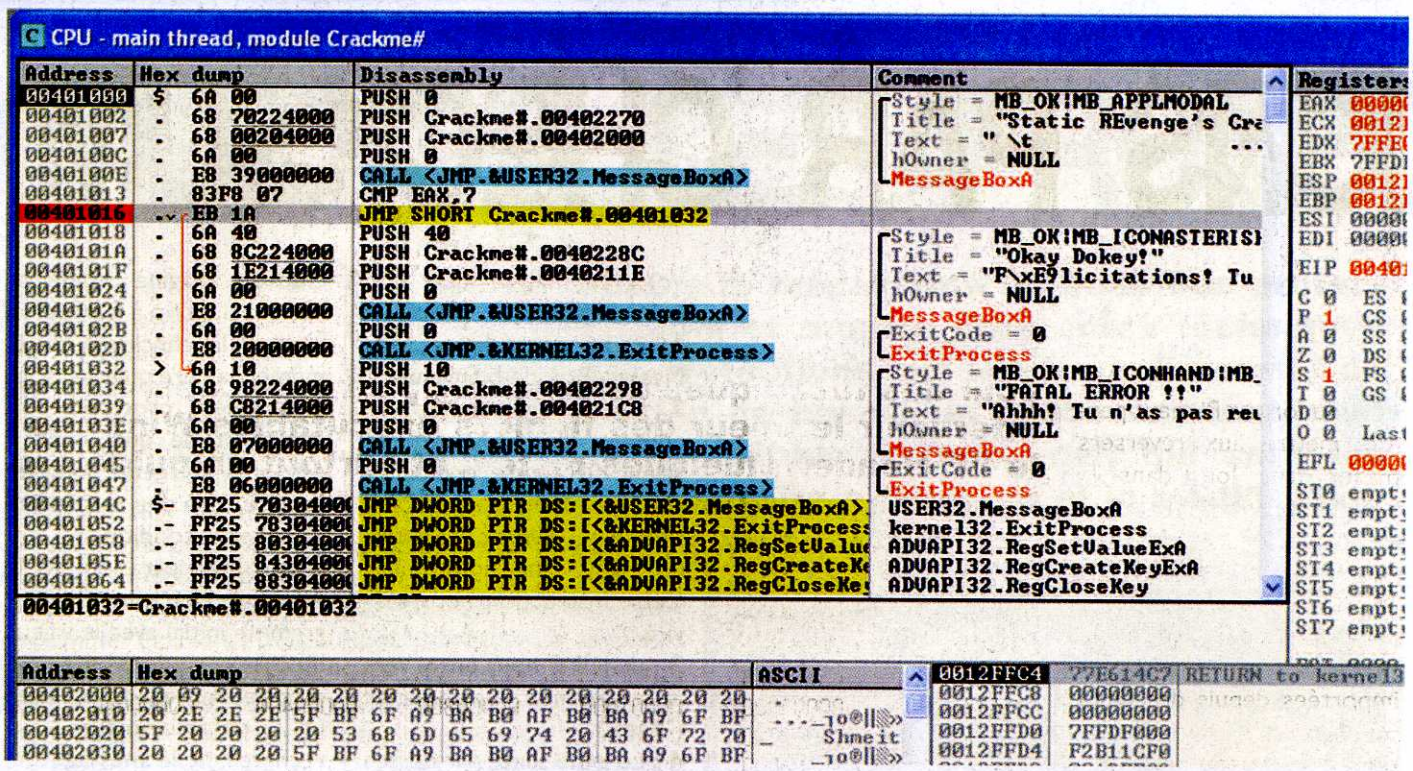
-Leur offset Virtuel (Voffset) et leur Offset Réel (Roffset)

-Leur taille virtuelle (VSize) et leur taille réelle (Rsize)

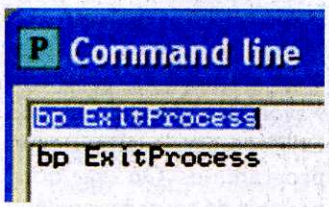
-Et aussi, ce qui est très important, leur caractéristiques : 'readable', 'writtable', et 'contain initialized data'.

D'abord,il faut savoir qu'un exécutable est séparé en différentes sections. Les principales sections sont : CODE (ou .TEXT) qui contient le code même du programme, la section .DATA qui contient les donnée ou variable utilisées par le programme, la section .IDATA qui contient les informations sur les fonctions et les DLL utilisées par le programme. C'est la section d'Import. Il y a encore la section .EDATA qui contient les fonction Exportées par le programme ; la section .rsrc qui contient les ressources utilisées par le programme (icones, chaines traduisibles, etc.). Toutes les informations sur la séparation des section et leurs taille est contenu dans le PE header.

Sachant que dans la plupart des cas, la taille réservée est plus grande que la taille réelle (la taille réservée est arrondie à 256 près), il reste presque toujours de la place a la fin des section pour qu'un crackeur puisse faire des chose pas très catholique. Nous verrons ces possibilités, en détail, dans un prochain Manuel, mais sachez déjà que cela peut permettre de passer la protection par checksum de



On peut poser un breakpoint avec la touche F2, directement sur la ligne qui nous intéresse.



Dans le menu plugin, l'option Command-line permet aussi de mettre des bp sur des API. Dans notre cas, on fera « bp ExitProcess ».

Comme ça dès que ExitProcess est appelé on tombe sur la bonne ligne dans le code, juste après le JUMP(ligne grisée). On a plus qu'a poser un breakpoint avant ExitProcess avec F2 sur le call checksum puis faire un 'trace into' pour arriver à la routine du checksum (en 401018). Je rappelle que dans la capture il n'est pas présent, mais c'est le processus théorique qui compte.

Deuxième méthode

Une deuxième méthode un peu plus raffinée serai de faire en sorte que la valeur du checksum reste la même. Puisqu'on a modifié le saut (en remplaçant EB1A par EB00), on a 1A en moins dans la valeur du Checksum. Il faut donc que dans les opcode on rajoute un total de 1A. Et c'est la que la comparaison située avant le JUMP va nous servir. On va remplacer CMP 07 par CMP 21(21= 07 + 1A), comme ça valeur du checksum reste identique. On aurais pu rajouter ailleurs 1A mais cet endroit est pratique car le résultat de la comparaison n'est pas pris en compte.

Il faut faire preuve d'astuce pour que les modifications que l'on réalise pour compenser ne modifient pas le comportement du programme. De plus il ne faut pas oublier que le checksum ne se fait pas forcement sur tout le programme. Si le checksum se fait depuis l'EntryPoint du prog (00401000) jusqu'au JUMP 1A compris (00401016), on ne peut modifier un opcode de la 3eme MessageBox. Dans ce cas, il est judicieux de modifier l'opcode de la comparaison (adresse 00401013) située juste avant le JUMP.

On peut faire cette modification directement avec l'éditeur hexadécimal de Ollydbg. On voit dans cette dernière capture que 07 a été remplacé par 21.

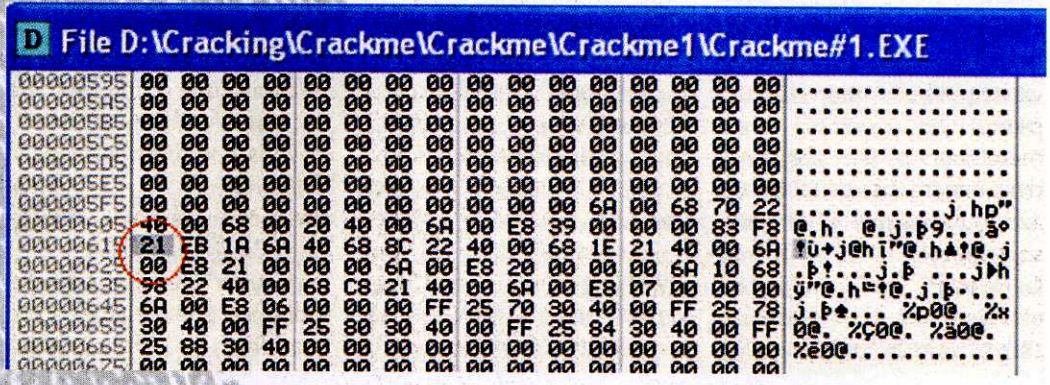
Après nous être familiarisé avec les checksums, nous verrons dans l'article suivant des techniques plus avancées pour les contourner.

Greet'z to my friends,les crackeurs et les teams qui font vivre la scèn francaise de cracking : Iciteam qui n'est plus (au dernières nouvelles, désolés les gars mais on se recroisera sur le net), N-Gen, toute la Shmeit Corp, Christal, TeeJi, Tamambolo, Lutin noir et les autres. Sans oublier Capashen, CrashFr, JeK et JiJi pour ce qu'il m'apprend 3 jours par semaines, et a tout les autres que je n'ai pas la place de citer.

OllyDbg : www.ollydbg.de

Entraînez-vous sur un autre crackme : demandez sur http://forum.thehackademy.net ou cherchez sur Google.

By : Silkscalp



CTIONS PAR CHECKSUM

```
// Program Entry Point
addresse        opcode Code ASM
:00401000       6A00    push 00000000
:00401002       6870224000      push 00402270
* Possible StringData Ref from Data Obj ->" "
:00401007       6800204000      push 00402000
:0040100C       6A00    push 00000000
* Reference To: USER32.MessageBoxA, Ord:0000h
:0040100E       E839000000      Call 0040104C
// C'était la MessageBoxA de bienvenue

:00401013       83F807  cmp eax, 00000007
:00401016       EB1A    jmp 00401032
// Tout ce joue dans ce JUMP vers 00401032, qui
correspond à la vilaine MessageBox disant qu'on a
raté ?

// GOODBOY : Si on arrive ici on a réussi
   (00 40 10 18)
// (NOTE : On va considerer qu'avant ce qui
suit?// pourrait se trouver un call vers la rou-
tine du ?// checksum)
:00401018       6A40    push 00000040
* Possible StringData Ref  ->"Okay Dokey!"
:0040101A       688C224000      push 0040228C
// une MessageBoxA qui nous dit : GoodBoy
* Possible StringData Ref from Data Obj ->"F"
:0040101F       681E214000      push 0040211E
:00401024       6A00    push 00000000
* Reference To: USER32.MessageBoxA, Ord:0000h
:00401026       E821000000      Call 0040104C
:0040102B       6A00    push 00000000
* Reference To: KERNEL32.ExitProcess, Ord:0000h
:0040102D       E820000000      Call 00401052
* Referenced by a (U)nconditional or (C)onditio-
nal Jump at Address: 00401016(U)
:00401032       6A10    push 00000010
// BADBOY : L'adresse du JMP vu plus haut (00401032)
* Possible StringData Ref  ->"FATAL ERROR !!"
:00401034       6898224000      push 00402298
* Possible StringData Ref from Data Obj ->"Ahhh!
Tu n'as pas reussis "
:00401039       68C8214000      push 004021C8
:0040103E       6A00    push 00000000
* Reference To: USER32.MessageBoxA, Ord:0000
:00401040       E807000000      Call 0040104C
// La MessageBox disant qu'on est pas bon

:00401045       6A00    push 00000000
* Reference To: KERNEL32.ExitProcess, Ord:0000h
:00401047       E806000000      Call 00401052
```

Voici l'amnalyse que l'on peut en faire :

1 : le prog lance une MessageBox de bienvenue

2 : le prog compare le contenu de EAX avec la valeur 07 (cette comparaison est inutile car elle est suivi d'un JUMP, c'est-à-dire un saut inconditionnel). Cette instruction nous servira.

3 : un JUMP nous envoi vers la troisième MessageBox ("Raté")

4 : Le jump saute par dessus la partie qui annonce que le crackme a été réussi. C'est aussi ici que le Call hypothétique du checksum vérifierait l'intégrité des données. Si le code a été modifié, on ferait un appel a ExitProcess, sinon on continuerait.

Je placerait ici le teste de checksum, après la première protection (inutile de donner prématurément des indices).

4 : le prog lance la 3eme MessageBoxA déclarant qu'on a pas réussi a cracker ce crackme.

5 : l'API ExitProcess quitte le programme

Pour réussir ce crackme, il suffit de modifier le jmp 00401032, par exemple par jmp 00401018 (EB1A -> EB00). Mais cette modification serait détecté par la vérification de checksum (il y aurait une différence de 26, soit 1A en hexadécimal). Voyons comment faire pour éviter ça.

Première méthode (classique)

Lorsqu'un message s'affiche indiquant que le fichier a été modifié ou vérolé, un crackers imagine immédiatement si qu'il a à faire : il va cracker le checksum de la même manière qu'il a cracké la première protection, en cherchant de quel endroit est ouverte cette fenêtre. Ensuite, il n'a plus qu'à inverser le saut conditionnel suivant la comparaison de la valeur du checksum du programme initial avec la valeur du checksum du programme modifié. Par exemple : avec un désassembleur en recherchant le message qui s'affiche dans les string data reference. Ou avec un débogueur, en plaçant un breakpoint sur MessageBoxA.

Même si aucun message n'est affiché, il va forcément y avoir un appel à la fonction ExitProcess pour quitter le programme. On va donc pouvoir de toute façon dénicher l'endroit du programme qui compare les checksums. En posant un breakpoint (bp) sur ExitProcess, on va pouvoir très facilement remonter le flot d'exécution.

Ensuite, on peut désactiver la protection de différentes manières, par exemple :

–en supprimant le call appelant la fonction de checksum.

–en inversant le saut conditionnel suivant la comparaison.

–en remplaçant la valeur comparée par la nouvelle valeur de checksum (que le programme aura calculé pour nous, et qui se trouvera dans le registre qui fait l'objet de la comparaison)

Avec ollydbg cela donnerait :

NEWBIE

CRACKER LES PROTEC

Cet article rappèle le mécanisme de base des protections par checksum et montre différentes techniques pour passer ces protections sur Windows. Il permet de révisier l'utilisation de base d'un désassembleur, d'un débugger et d'un éditeur hexadécimal, qui sont les outils de base nécessaire à l'examen et la modification d'un programme binaire.

Lorsqu'on a compris le fonctionnement d'un programme (ou d'une partie qui nous intéressait), on peut aussi être tenté d'en modifier le comportement. Les crackers, surtout, voudront supprimer les appels aux routines de protections. D'autres voudront plutôt ajouter des options à un programme closed-source, par besoin, par défi ou pour tricher à un jeu vidéo. Dans certains cas, il est même nécessaire de modifier le programme pour faciliter la phase d'analyse (enlever les pièges anti-debugger, par exemple).

Nous allons pour l'instant nous intéresser au premier cas, parce qu'il est plus simple à illustrer. Nous allons donc nous mettre dans la peau du cracker qui veut supprimer une vérification de numéro de série. Dans les cas les plus courants, cette supression se résume à inverser un saut conditionnel, ce qui aura pour effet d'accepter tous les serials (sauf ceux qui sont valides). La tâche est moins triviale si on a affaire à une deuxième protection, qui, à l'aide d'un checksum, tente d'empêcher cette inversion (et tout autre modification du code machine).

Rappel sur les checksums

Il y a deux types de vérification pour l'intégrité des données. Les checksums (sommes de contrôle) et les signatures. Les premiers sont utilisés pour détecter les erreurs (de transmission, par exemple), les secondes pour empêcher la falsification (voir la rubrique Linux, sur l'intégrité des fichiers). Malheureusement, les développeurs de protections ne font pas toujours la différence et utilisent souvent des checksums (CRC16/32, XOR, ou d'autres versions personnelles) là où ils devraient utiliser une signature (MD5, ou d'autres algorithmes difficiles à tromper). Dans cet article, je vais considérer le cas, assez courant hélas, de la somme de contrôle calculée en faisant la somme des opcodes des instructions.

Par exemple un Checksum fait sur ces 2 instructions :

```
Adresse     opcode      asm
401000      83F805      CMP EAX,05
401003      7505  JNE 401008
```

Si l'on fait un checksum sur ces 2 instructions, on a comme résultat 0x1FA (= 83+F8+05+75+05). Si l'on remplace le JNE par un JN, on aura une autre somme : 0x1F9 (=83+F8+05+74+05).

Il faut voir aussi qu'avec une somme, un modification peut en compenser une autre ; nous y reviendrons.

Reconnaître une protection par checksum

Il y a plusieurs manière de détourner un checksum. Si la somme d'un checksum n'est pas valable (=ne correspond pas a la valeur du checksum de l'exe initial) il peut se passer plusieurs résultats lors du lancement du programme :

1.Soit il affiche carrément un texte du style « fichier modifié » ou « fichier infecté par un virus » .

2.Soit il affiche une erreur au lancement (généralement c'est une MessageBox qui indique l'erreur).

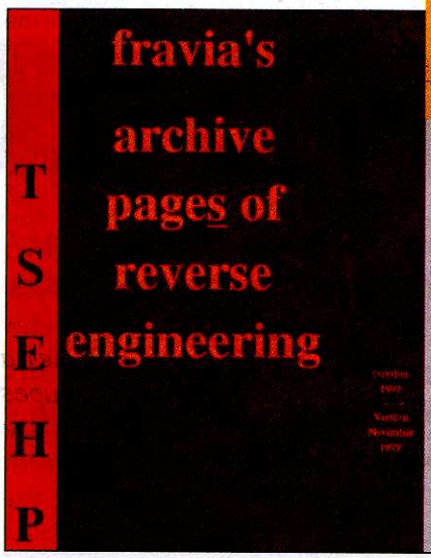
3.Soit le programme ne se lance pas sans rien indiquer.

Les 2 et 3 ne sont pas des indication fiable a 100% de la présence d'une protection par checksum. Dans tous les cas, le prog finira forcément par la fonction ExitProcess.

Pratique

Prenons l'exemple suivant : un crackme de Static REvenge's de la mythique Shmeit Corp (grâce leur soit loué, car c'est par leurs célèbres mémentos que j'ai appris le cracking et priez pour qu'ils n'arrêtent pas car ils sont vraiment bon et forment avec d'autres les fondements de la scène française). Je vous présente le code désassemblé d'une partie d'un crackme de Static Revenge's puis je l'expliquerai. Je vous préviens tout de suite, il n'y a pas de checksum dans ce crackme, mais on va imaginer ce qui se passerait s'il y en avait un pour nous compliquer la vie.

LES BONNES ADRESSES



Software reverse engineering and web survival arguments

Software protection techniques and tools

-

Anonymity on the Web: stalking, enemy tracking and other techniques

-

Reality cracking and anti-advertisement techniques and tools

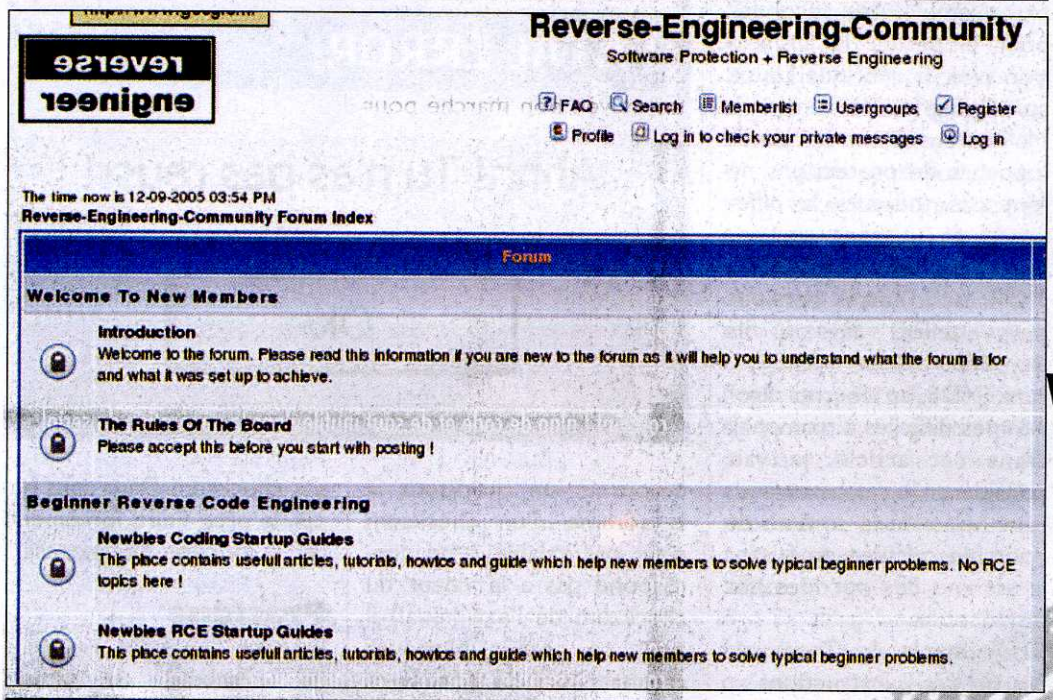
-

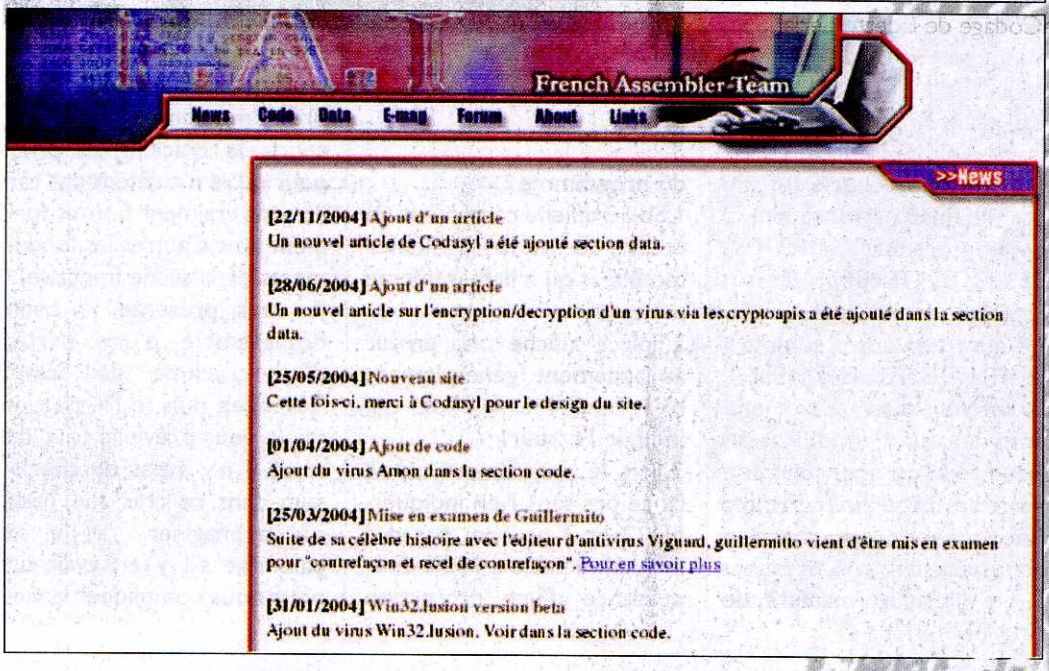
How to search the web: combing, klebing and other strategies

-

cookies removal; bots and web-spiders trapping; anti-MicroSoft and anti-Netscape scripts and tricks; tools for software reverse engineering; cgi-cracking; user self-defense; corporate survival counter measures; home-pages capering and password cracking techniques; commercial stunt sites busting; Java applets reversing; vxd monitoring; steganographical and cryptological reversing matters; Javascript based site protection and deprotection techniques; email patterns reversing

http://community.reverse-engineering.net/
http://zor.gulli.com/
http://neitsabes.free.fr
http://www.newffr.com/
http://fat.next-touch.com/main.htm
http://zor.org/fighting
http://protools.cjb.net
http://ollydbg.de
http://acrigs.com/FRAVIA/FRAVIA_index.htm
http://www.xfromfrance.net





```
SI NbCarac >4 FAIRE
   Pour I de 1 jusqu'à NbCarac
      CaracEtudié <- prendre caractère n° I
                     du nom
      // (I=1er caractère du nom : S,
         puis I=2ème caractère...)
      ValeurCarac <- Valeur NUMERIQUE
                     ASCII(CaracEtudié)
      // (la valeur numérique du carac-
         tère ascii. Ex S = 83)
      ValeurCarac <- ValeurCarac -
                     CompteurInverse
     // (Donc pour S :
        S = 83, 83 - 9 = 74)
      CaracEtudie <-  VALEUR ASCII de
                      (ValeurCarac)
      //(Donc ici c'est Valeur ascii de
        74 : 74 = " R ")
      Serial <- Serial plus CaracEtudié
      // SERIAL = vide plus " R ", donc
         SERIAL = " R " au premier tour.
      CompteurInverse <- CompteurInverse -1
      // 9 = 9-1 donc 8 pour le pro-
         chain tour de boucle.

   FIN Pour

   Afficher Nom
   Afficher Serial

SINON
Afficher " Pas assez de caractères au nom... "
   FIN SI
FIN
```

Pour concrétiser un peu cet algo, on va prendre un jeu d'essai comme TOTO45.
Le nom fait six lettres, donc remplit le critère de longueur minimum. On va voir ce que donne comme résultat l'algo écrit ci-dessus, appliqué au nom TOTO45 :

```
LettrecompteurCompteurinverseCodage de la lettreSerial
obtenuCalculT16NNAsciiNUM(T)=84-
6=78.AsciiALPHA(78)=N O25JN & J donne
NJAsciiNUM(T)=79-5=74.AsciiALPHA(74)=J
T34PNJ & P donne NJPAsciiNUM(T)=84-
4=80.AsciiALPHA(80)=P O43LNJP & L donne
NJPLAsciiNUM(T)=79-3=76.AsciiALPHA(76)=L
4522NJPL2AsciiNUM(T)=52-
2=50.AsciiALPHA(50)=2
5614NJPL24AsciiNUM(T)=53-
1=52.AsciiALPHA(52)=4
Le serial correspondant à " TOTO45 "
est donc " NJPL24 ".
```

Conclusion

Il s'agissait ici d'un exemple simple, un cas d'école, facile à reverser. Le principe de la réalisation d'un keygen est cependant toujours le même, à cela près que les différentes étapes peuvent être rendues plus difficiles (voir encadré pour quelques méthodes qui rendent la vie plus dure au cracker). Ce qu'il faut comprendre, c'est que comme l'algorithme est présent, d'une manière ou d'une autre, dans le programme, on peut le récupérer et le détourner. On peut pallier à ce problème en utilisant des fonctions irréversibles ou des algorithmes cryptographiques asymétriques (la clé publique étant dans le programme, et la privée chez l'éditeur).
Cependant on pourra difficilement empêcher les gens de s'échanger des couples nom/serial valides. C'est là tout le problème des protections par numéro de série : c'est un moyen bon marché pour l'éditeur de distribuer des licences, mais par là même une protection facile à abuser pour n'importe qui.

Greet'z to the french crackers et tous ceux qui écrivent des tutoriaux dans le but de transmettre leur connaissance... : la schmeitcorp, FFF... et à tous les autres groupes que je n'ai pas la place de citer mais qui font vivre la scène française par leur présence...

By SILKSCALP

Quelques protections possibles

1) Il est évident que plus l'algorithme utilisé pour générer le serial est compliqué, plus la tâche du cracker sera ardue. Je vous conseille donc, pour les programmes que vous voulez protéger, de faire des algo de génération de serials compliqués.
2) Vous avez aussi la possibilité d'essayer de perdre le cracker en entrecoupant l'algo de code et de saut inutile dans la génération du serial.
3) Vous pouvez utiliser des packer/crypteur afin d'essayer d'empêcher une décompilation rapide et complète du programme. L'utilisation des méthodes de détection de débogueur/désassembleur (parfois intégrée dans les packer/crypter) est intéressante : si un débogueur est lancé sur l'ordi, le programme s'arrête. L'utilisation du débogueur est presque obligatoire pour faire un keygen, quand le code devient compliqué (à moins d'arriver à repérer la routine dans un code désassemblé et de très bien comprendre l'assembleur).

Surtout ne jamais utiliser de chaîne fixe pour stocker le serial : le serial est stocké dans une variable initialisée au début du programme. Dans ce cas, le serial apparaîtra dans les string data reference. L'idéal étant que le serial soit généré à la volée.

4) On peut essayer de perdre le cracker dans un algo et l'envoyer dans une boucle infinie. Avec ollydgb (ring3) cela va juste faire perdre un peu de temps (+ le temps de la compréhension), mais avec softice (ring 0) la boucle infinie nous impose un reboot de l'ordi car on ne pourra reprendre la main pour fermer le débogueur, et donc nous impose d'autres techniques de débogage.
5) On peut utiliser des tests pour voir si les registres de débogage sont utilisés (DR0.) et, si c'est le cas, un débogueur tourne. Il faut donc utiliser une manière différente de déboguer.
6) On peut même utiliser l'obfuscation, mais c'est un sujet très compliqué qui est à traiter séparément et mérite largement un ou plusieurs articles...

```
00401143  . E8 42000000   CALL <JMP.&USER32.GetDlgItemTextA>      GetDlgItemTextA
00401148  . B8 80304000   MOV EAX,AD_CM%2.00403080
0040114D  . BB 80324000   MOV EBX,AD_CM%2.00403280
00401152  . 8BCE          MOV ECX,ESI
00401154  > 8A10          MOV DL,BYTE PTR DS:[EAX]
00401156  . 2AD1          SUB DL,CL
00401158  . 3813          CMP BYTE PTR DS:[EBX],DL
0040115A  .v75 18         JNZ SHORT AD_CM%2.00401174
0040115C  . 40            INC EAX
0040115D  . 43            INC EBX
0040115E  .^E2 F4         LOOPD SHORT AD_CM%2.00401154
00401160  . 6A 40         PUSH 40                                 Style = MB_OK|MB_ICONASTERISK|MB_APPLMODAL
00401162  . 68 12304000   PUSH AD_CM%2.00403012                    Title = "ArturDents CrackMe#2"
00401167  . 68 27304000   PUSH AD_CM%2.00403027                    Text = "Yeah, you did it!"
0040116C  . FF75 08       PUSH DWORD PTR SS:[EBP+8]               hOwner
0040116F  . E8 1C000000   CALL <JMP.&USER32.MessageBoxA>          MessageBoxA
00401174  > C9            LEAVE
00401175  . C2 0400       RETN 4
```

Breakpoint un peu avant " Yeah You Did It ".

Nous pourrions analyser Gen en regardant le code ASM, mais le débogage du programme nous permet de mieux comprendre comment la routine de génération du serial fonctionne. Cette routine est très courte : sept lignes. Elle génère, grâce à une boucle, chaque caractère du serial en fonction des caractères du nom. Je vais expliquer comment le serial est généré à partir des quelques premiers caractères du nom, sachant que le processus est le même pour les caractères suivants.

4) Explication de la routine de génération du serial

Considérons que nous avons passé la vérification de la taille de notre nom et donc qu'il fait au moins cinq caractères. Je prendrais comme nom " silkscalp " et comme serial " 123456 ". J'utilise OllyDbg qui m'a permis de poser un point d'arrêt à l'adresse mémoire 00401154, un peu avant le " Yeah You Did It ".

Grâce au débogueur, nous allons rapidement analyser le code et surtout, nous verrons ce que contiennent ces variables (en fait, nous confirmerons l'analyse du code qui a été grandement facilitée par l'usage des API et notre connaissance de leur prototype).

Déjà, les trois lignes précédant mon break-point mettent (voir illustration) :

Le nom dans EAX. En effet, dans la partie des registres, on voit marqué 00403080 (adresse mémoire) " silkscalp ".

Le serial dans EBX (123456). Pareil, mais l'adresse est 00403280.

Le nombre de caractères de notre nom dans ECX (silkscalp = 9 caractères).

Maintenant, on se trouve sur la ligne où j'ai mis mon break-point. Vu qu'ECX sert de compteur, la boucle va s'arrêter au dernier caractère de notre nom, ou lors d'un saut s'il y a une erreur dans le serial. Le compteur décrémente de 1 par tour de boucle jusqu'à arriver à zéro. Donc, ECX va partir de 9 et diminuer de 1 par tour de boucle (c'est l'équivalent d'une boucle " for(i=9 ; i>0 ; i--) ").

00401154 : met la value ascii du 1er caractère de mon nom dans DL, la partie basse d'EDX (cela suffit à traiter un caractère sur 8 bits, comme le s de " silkscalp " - 73 en ascii). " BYTE " sert à indiquer qu'on ne copie que les 8 premiers bits d'EAX dans DL.

00401156 : on soustrait de DL la valeur du compteur (9 au premier tour de boucle, 8 au deuxième, puis 7...). DL contient donc maintenant 6A (73-9=6A).

00401158 : le programme compare DL (6A) avec la valeur numérique du premier caractère du serial contenu dans EBX. Soit 31 pour le 1 de " 123456 ".

0040115A : si le résultat n'est pas égal, on est éjecté du programme, sinon, on continue.

Les trois lignes suivantes vont incrémenter EAX et EBX de 1 pour passer aux caractères suivants. Puis ECX va diminuer de 1 avant de boucler pour revenir à 00401154.

On peut donc dès maintenant en déduire un petit algorithme retraçant la routine telle qu'elle se déroule ci-dessus. Je prends comme nom " SILKSCALP ".

```
Registers (FPU)
EAX 00403080 ASCII "silkscalp"
ECX 00000009
EDX 00140608
EBX 00403280 ASCII "123456"
ESP 0012FCD4
EBP 0012FCD4
ESI 00000009
EDI 0012FD54

EIP 00401154 AD_CM%2.00401154
```

Contenu des registres au début de Gen (EAX=nom, EBX=serial).

```
PROGRAMME GEN_SERIAL
DEBUT

VAR :

      Nom :  Chaîne de caractères
      Serial : Chaîne de caractères
      NbCarac : entier
      SerialFinal : Chaîne de caractères
      CaracEtudié : Chaîne de caractères
      ValeurCarac : entier
      CompteurInverse :entier
      I : entier


      Serial <- ""
      NbCarac <- longueur du nom
        //longueur : silkscalp=9 carac)

      CompteurInverse <- NbCarac
        // donc 9 dans cet exemple
```

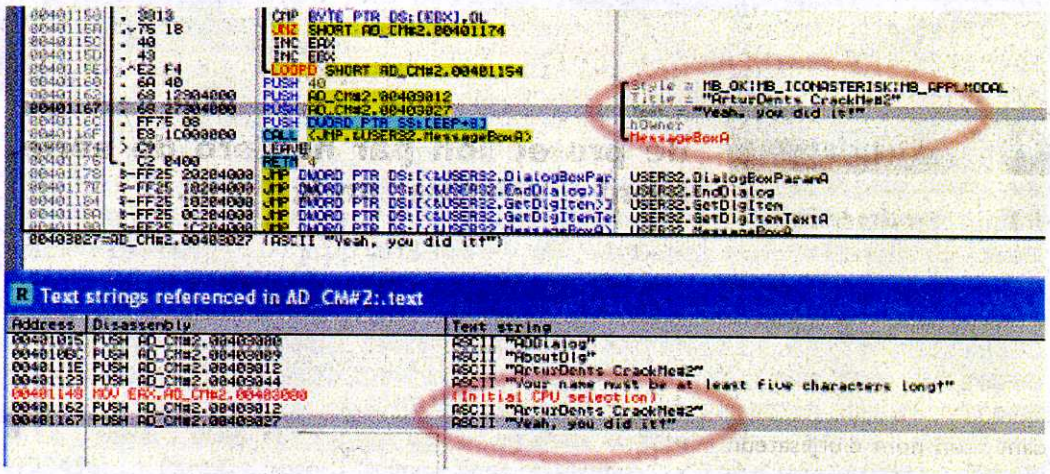
Si on regarde quels sont les arguments que l'on passe à cette fonction dans la documentation de Microsoft, ou même APICRACK.hlp de " +Sync ", on s'aperçoit que :
- le PUSH 0BB8 correspond au numéro du control traité par la fonction.
- 00403080 correspond à la zone mémoire où sera stocké notre nom.
- SS :EBP+8 contient logiquement le handle de la fenêtre.
- Et surtout 14 (20d), qui correspond au nombre de caractères maximum traité par la fonction.

(Il faudra en tenir compte dans notre keygen.)

```
00401113   |. 8BF0            MOV ESI,EAX
//-------- Puis ce nombre est copié d'EAX vers ESI
00401115   |. 8D01            LEA EAX,DWORD PTR DS:[ECX]
00401117   |. 83FE 05         CMP ESI,5
//-------- Ici, le prog compare ESI avec 5, sachant qu'ESI (le nombre de caractères composant notre nom, si vous avez suivi jusque là).
0040111A   |. 7D 18           JGE SHORT AD_CM#2.00401134
//-------- Si notre nom fait plus de cinq caractères, on continue la routine, sinon on est viré du programme, après l'affichage d'une MessageBoxA. Si on satisfait la condition, on saute en 00401134 qui va être le début de la routine de génération du serial.
0040111C   |. 6A 40           PUSH 40
0040111E   |. 68 12304000     PUSH AD_CM#2.00403012  |Title = "ArturDents CrackMe#2"
00401123   |. 68 44304000     PUSH AD_CM#2.00403044  |Text = "Your name must be...!"
00401128   |. FF75 08         PUSH DWORD PTR SS:[EBP+8]   |hOwner
0040112B   |. E8 60000000     CALL <JMP.&USER32.MessageBoxA>
[...]
//-------- On arrive ici à la fin du cas où notre nom fait au moins cinq caractères. Passons aux choses sérieuses.
00401134   |> 6A 14           PUSH 14
00401136   |. 68 80324000     PUSH AD_CM#2.00403280
0040113B   |. 68 B90B0000     PUSH 0BB9
00401140   |. FF75 08         PUSH DWORD PTR SS:[EBP+8]
00401143   |. E8 42000000     CALL <JMP.&USER32.GetDlgItemTextA>
00401148   |. B8 80304000     MOV EAX,AD_CM#2.00403080
//-------- L'API a le même prototype que donné plus haut.
      On note que le serial est sauvegardé dans 00403280.
0040114D   |. BB 80324000     MOV EBX,AD_CM#2.00403280
//-------- Mets l'adresse du serial dans EBX.
00401152   |. 8BCE            MOV ECX,ESI
//-------- Le nombre de caractères de notre nom dans ECX (je rappelle que ECX sert de compteur pour les boucles).Gen commence réellement ici. Nous allons la développer plus tard, grâce au débogueu
r.
00401154   |> 8A10            /MOV DL,BYTE PTR DS:[EAX]
00401156   |. 2AD1            |SUB DL,CL
00401158   |. 3813            |CMP BYTE PTR DS:[EBX],DL
0040115A   |. 75 18           |JNZ SHORT AD_CM#2.00401174
0040115C   |. 40              |INC EAX
0040115D   |. 43              |INC EBX
0040115E   |.^E2 F4           \LOOPD SHORT AD_CM#2.00401154
//-------- On passe par les lignes qui suivent uniquement si notre serial est correct (sinon on saute cette partie)
.
00401160   |. 6A 40            PUSH 40
00401162   |. 68 12304000     PUSH AD_CM#2.00403   |Title = "ArturDents CrackMe#2"
00401167   |. 68 27304000     PUSH AD_CM#2.00403      |Text = "Yeah, you did it!"
0040116C   |. FF75 08          PUSH DWORD PTR SS:[EBP+8]
0040116F   |. E8 1C000000     CALL <JMP.&USER32.MessageBoxA>
//-------- Si on est passé par ici, c'est que l'on a un bon serial.
00401174   |> C9              LEAVE
00401175   \. C2 0400          RETN 4
Fin du programme.
```

ROTECTIONS PAR SERIAL

Nous vérifierons aussi que le programme n'est pas protégé par un cryptage ou une compression à l'aide de StudPE (onglet signature). Celui-ci nous indique que le programme a été fait avec MASM/TASM, donc écrit en asm pur, et n'a pas été protégé particulièrement. Puis, grâce a cet excellent programme, nous pouvons voir toutes les fonctions importées depuis des DLL.



OllyDbg, recherche de la string reference " Yea You did it ".

3) Recherche de la routine de génération du serial

Nous allons utiliser OllyDbg pour tracer le crackme et trouver la routine qui génère le serial. À partir de maintenant, nous appellerons cette routine " Gen ". Elle génère un serial valide en fonction du nom et le compare avec notre serial, cette comparaison se faisant caractère par caractère.

Pour trouver cette routine, on lance OllyDbg puis on charge le crackme (AD_CM#2.exe, disponible sur le site du Journal). Ensuite, on va chercher dans les " string data reference " s'il n'y a pas une indication qui pourrait nous aider à situer Gen. Pour cela, on fait un clic droit dans la fenêtre principale et un " Search for ", puis, dans le menu, on choisit " All Referenced Text String " (c'est l'équivalent des " string data reference " de Wdasm). Et en effet, on trouve une référence " Yeah you did it ! " (qui signifie que l'on a réussi). En double-cliquant, on se trouve directement dans le code du programme, sur la ligne correspondante. On aurait aussi pu poser un break point sur GetDlgItemTextA, car StudPE nous a indiqué que cette API était utilisée par le programme, et tracer le programme à la recherche de la routine de génération du serial. Cette méthode est plus efficace, mais lorsqu'on trouve une chaîne de caractères disant " Yeah You did It " dans un crackme, on peut légitimement supposer que cette chaîne sera affichée en cas de réussite.

Si un message indique que l'on a réussi, logiquement, Gen se trouve peu avant. Donc on va remonter un peu dans le code et regarder les API appelées.

Dans l'ordre d'apparition (ordre linéaire par rapport au code) :

- Il y a un GetDlgItemTextA qui récupère notre nom à partir du contrôle de zone de texte et stocke notre nom dans une adresse mémoire. On peut voir le numéro de ce contrôle grâce au numéro donné par OllyDbg : Control Id (ici BB8 ou 3000 en décimal).
- Le nombre de caractères de notre nom est comparé avec 5 ; une MessageBoxA apparaît si la taille est incorrecte.

- Une boucle commençant en 00401154 et finissant en 0040115, qui est le c?ur de notre routine Gen.
- Un autre appel à GetDlgItemTextA, qui lui récupère notre serial entré. On peut le voir grâce au Control Id donné qui est le suivant de celui de saisie du nom (ici BB9 ou 3001).
- Une MessageBoxA indiquant la réussite.
- Un ExitProcess manuel qui est exécuté quel que soit le résultat.

Voici toute cette portion de code commentée afin de faire comprendre au mieux ce qui pourrait encore paraître obscur à certains.

```
ADRESSE   CODE HEXA          CODE ASM

004010FC  /$ 55              PUSH EBP
004010FD  |. 8BEC            MOV EBP,ESP

//--------- Préparation de la pile
004010FF  |. 6A 14           PUSH 14
00401101  |. 68 80304000     PUSH
                          AD_CM#2.00403080
00401106  |. 68 B80B0000     PUSH 0BB8
0040110B  |. FF75 08         PUSH DWORD
                          PTR SS:[EBP+8]
0040110E  |. E8 77000000     CALL
          <JMP.&USER32.GetDlgItemTextA>

//--------- La valeur de retour de
GetDlgItemTextA est le nombre de lettres
de notre nom (dans EAX).
```

W I L D

LA FAIBLESSE DES PR

Prérequis : niveau newbie en reverse et quelques connaissances de base en ASM.

Le cracking est un excellent moyen d'aborder par la pratique les outils de deboguage et le langage assembleur. Dans cet article, nous nous intéressons à un exemple simple de protection par numéro de série, et montrons comment en dériver un keygen.

1) Introduction

Définition : un keygen, ou keygenerator, est un petit programme qui utilise la routine de génération de serial du programme, afin de générer le serial correspondant à un nom d'utilisateur. Un keygen est donc spécifique à un programme.

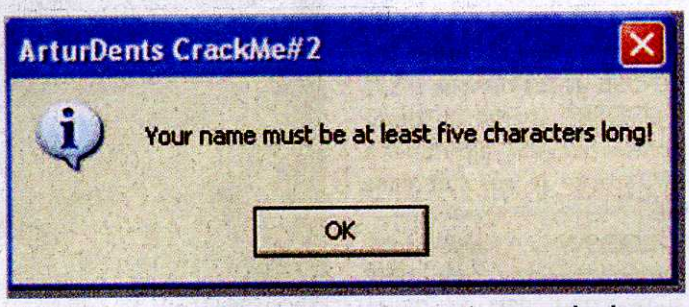
Nous allons voir comment un cracker peut réaliser un KeyGen, afin de comprendre comment s'en protéger. Pour cela, nous prendrons un exemple simple mais concret : un crackme.

Dans une première partie, nous ferons l'analyse du crackme, et en particulier de la routine de génération du serial. Puis, dans une deuxième partie, grâce à cette analyse, nous allons coder un KeyGen.

Outils :

Un désassembleurs (optionnel) : wdasm 8.93 (et ses patchs) ou bien IDA PRO, qui est beaucoup plus puissant selon moi, car il permet de renommer toutes les fonctions et les variables dans le programme désassemblé (ce qui permet une meilleure compréhension, et même de " s'approprier le programme "). Ces possibilités ne sont pas présentes dans Wdasm (Ref Misc 14, article sur IDA).

Un débogueur : j'utiliserais OllyDbg 1.10 avec des plugins afin de tracer la routine de génération du serial.



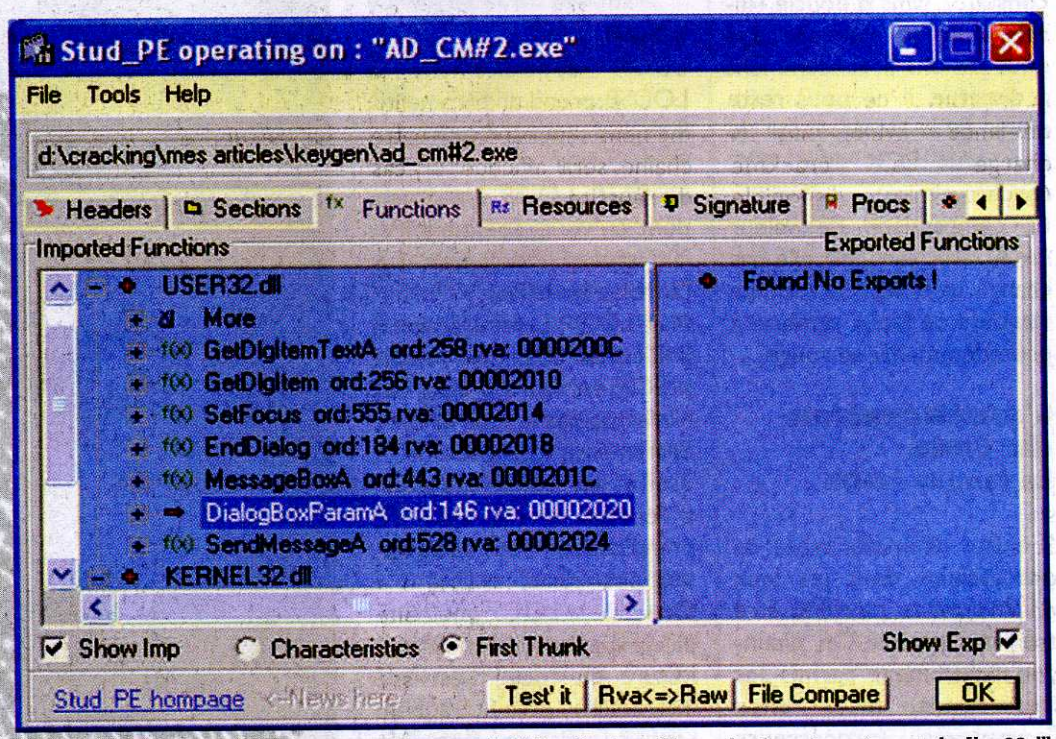
Ici, l'indication importante est qu'il faut mettre au moins 5 caractères dans le nom.

Visual Basic (ou tout langage que vous maîtrisez afin de créer le crackme). J'ai choisi VB car il permet de développer très rapidement l'interface graphique et le code est simple à écrire.

Un éditeur de PE pour la prise d'informations préalable. Je conseillerai StudPE ou LordPe (mais d'autres sont très performants). Actuellement, ils sont parmi les meilleurs, car ils permettent d'avoir accès à toutes les information du PE, de faire des dump et de rechercher si le programme est protégé par une compression/cryptage.

2) Analyse générale du crackme

Pour commencer l'analyse, nous allons lancer la cible et noter toutes les indications potentielles que nous apercevons lors des tests : mettre un serial sans nom, mettre un nom d'un caractère, ne pas mettre de serial, etc. Ce, pour voir si une MessageBoxA apparaît et donne quelques indications.



On peut ainsi voir parmi les fonctions importées : GetDlgItemTextA et MessageboxA, qui proviennent de User32.dll, ainsi qu'ExitProcess de Kernel32.dll.

AINES DE CARACTERES

```
mov eax, ecx
ret
```

Le code initial (en bleu) allégé de son prologue récupère l'adresse passée en argument par ESP en la stockant dans ESI. Pour optimisation, on initialise le compteur ECX à zéro (taille initiale de la chaîne) à l'aide d'une soustraction.
La boucle (en violet) commence par l'instruction spéciale LODSB. LODSB (B pour Byte) prend un seul octet à l'adresse enregistrée dans ESI et met le résultat dans la partie adéquate du registre EAX :AL. Après quoi, ESI est incrémenté d'un octet.
LODSB nécessite donc qu'un registre soit initialisé correctement (ESI) et effectue deux actions : la récupération d'un octet, l'incrémentation d'un registre. Comme vous le constatez, dans la boucle, l'incrémentation manuelle que nous faisions précédemment à disparue. Il ne nous reste alors qu'à incrémenter le compteur ECX avec INC (code optimisé).
La comparaison de valeur (de AL avec zéro) n'a, elle, pas changé. La boucle se termine toujours de façon similaire à précédemment (en rouge).

L'amélioration du code

La famille LODS

Il existe différents types de processeurs dont les jeux d'instructions peuvent être très changeants. On distingue notamment parmis les instructions celles dites "complexes" (processeurs CISC) et les instructions "simples" (processeurs RISC). Les processeurs Intel ont un jeu d'instructions à la fois simples (MOV, PUSH, POP, etc.) et complexes (LODSB et LOOP en sont deux exemples).

Voici un rappel utile sur LODSB et ses dérivés :
- LODSB : charge un octet pointé par ESI, ajoute 1 à ESI, stocke le résultat dans AL.
- LODSW : charge deux octets pointés par ESI, ajoute 2 à ESI, stocke le résultat dans AX.
- LODSD : charge quatre octets pointés par ESI, ajoute 4 à ESI, stocke le résultat dans EAX.

Grâce à ces instructions, on peut non seulement parcourir des chaînes mais également des tableaux de nombres.

LOOP

LOOP, contrairement à LODSB, prend un paramètre. Il s'agit d'une adresse (représentée dans votre code par un label). LOOP fonctionne donc en visuel comme JMPet fait sauter l'exécution sur l'adresse spécifiée.
Mais LOOP, contrairement à JMP, est conditionnel : il vérifie si ECX est égal à 0 et, si oui, n'effectue pas le saut. Sinon, il décrémente le registre ECX de 1 et fait sauter l'exécution à l'endroit spécifié. ECX ne prend pas son nom de compteur pour rien...
On utilisera LOOP pour simplifier la programmation et la lecture du code. C'est une instruction utile lorsqu'on souhaite effectuer un nombre prédéfini d'actions.

```
;; Un implémenta-
   tion de strcmp
   en assembleur
   (C) Microsoft
   1985-1997
   en entrée :
      ecx et edx
   en sortie : ax

strcmp  proc
 ; edx = src
 mov edx,[esp + 4]
 ; ecx = dst
 mov ecx,[esp + 8]

 test     edx,3
 jnz short dopartial

dodwords:
 mov eax,[edx]
 cmp      al,[ecx]
 jne short donene

 or       al,al
 jz short doneeq
 cmp ah,[ecx + 1]
 jne short donene

 or       ah,ah
 jz short doneeq

 shr      eax,16
 cmp al,[ecx + 2]
 jne short donene

 or       al,al
 jz short doneeq

 cmp ah,[ecx + 3]
 jne short donene

 add      ecx,4
 add      edx,4
 or       ah,ah
 jnz short dodwords
doneeq:
 xor      eax,eax
 ret
donene:
 sbb      eax,eax
 sal      eax,1
 inc      eax
 ret

dopartial:
 test     edx,1
 jz short doword

 mov      al,[edx]
 inc      edx
 cmp      al,[ecx]
 jne short donene

 inc      ecx
 or       al,al
 jz short doneeq

 test     edx,2
 jz short dodwords

doword:
 mov      ax,[edx]
 add      edx,2
 cmp      al,[ecx]
 jne short donene

 or       al,al
 jz short doneeq

 cmp ah,[ecx + 1]
 jne short donene

 or       ah,ah
 jz short doneeq

 add      ecx,2
 jmp short dodwords

strcmp  endp

        end
```

MANIPULATION DE CHA

Les chaînes de caractères interviennent dans pratiquement tous les éléments visibles d'une interface graphique ou non. Cet article montre comment on les manipule en assembleur.

Une application concrète

strlen() est une fonction de la librairie C qui prend en argument l'adresse d'une chaîne de caractères et retourne sa taille. À quoi bon utiliser la fonction strlen() si vous pouvez recoder vous-même cette fonction en assembleur ;-) ? En réalité c'est un exercice instructif que de recoder des fonctions de ce type.

La mesure de la longueur d'une chaîne de caractères est une procédure simple. Sachant que toute chaîne de caractères est terminée par un octet nul (un octet égal à zéro), il suffit de la parcourir octet par octet et de compter le nombre d'octets parcourus jusqu'à l'octet nul terminant la chaîne.

N'essayez pas de réécrire ce code directement ; il s'agit d'une fonction d'un programme et non un programme complet.

Syntaxe AT&TSyntaxe NasmSyntaxe Masmmy_strlen:

```
push %ebp
mov %esp, %ebp
mov 8(%ebp), %esi
mov $0, %ecx
_loop:
mov (%esi), %al
cmp $0, %al
je _end
add $1, %ecx
add $1, %esi
jmp _loop
_end:
mov %ecx, %eax
mov %ebp, %esp
pop %ebp
retmy_strlen:
push ebp
mov ebp, esp
mov esi, [ebp + 8]
mov ecx, 0
.loop:
mov al, [esi]
cmp al, 0
je .end
add ecx, 1
add esi, 1
jmp .loop

.end:
mov eax, ecx
mov esp, ebp
pop ebp
retmy_strlen:
push ebp
mov ebp, esp
mov esi, [ebp + 8]
mov ecx, 0
boucle:
mov al, [esi]
cmp al, 0
je fin
add ecx, 1
add esi, 1
jmp boucle

fin:
mov eax, ecx
mov esp, ebp
pop ebp
ret
;(Sous Masm, "loop" et "end" sont des mots réservés, on utilisera des synonymes.)
```

La première partie du code (en bleu) sert à initialiser les données nécessaires à la procédure de comptage. Après le prologue "mov 8(%ebp), %esi" récupère l'adresse de la chaîne de caractères dans le registre ESI. Ensuite le registre ECX (le compteur) est mis à zéro. Les deux premières instructions de la boucle (violet) récupèrent le premier octet de la chaîne (notez qu'on ne récupère qu'un seul octet) et le comparent à 0. Si l'octet est nul, alors on saute à la fin de la fonction, autrement on incrémente ECX (le compteur) puis ESI (contenant l'adresse de la chaîne, on va donc un octet plus loin dans la chaîne). Après quoi, on revient au début de la boucle.

À la fin de la boucle, le résultat dans ECX est transféré vers EAX puis on effectue l'épilogue et on retourne.

L'amélioration du code

Le code présenté jusque là est assez fruste. Il est possible d'alléger sa taille et sa lisibilité moyennant quelques techniques de programmation et de nouvelles instructions.

Par exemple, pourquoi utiliser un prologue et un épilogue alors que nous ne touchons pas à la pile autrement que pour récupérer un argument ? C'est certes une bonne habitude, mais on peut occasionnellement s'en passer.

De même, si l'on modifie quelques instructions, on pourrait alléger la taille du code compilé et sa vitesse d'exécution (qui a son importance lorsque le code fait des boucles). C'est le cas ici des initialisations des compteurs et de leur utilisation : "add $1, %ecx", par exemple, est long à écrire et prend plus de place une fois compilé.

C'est également l'occasion de découvrir une instruction complexe : LODSB.

Syntaxe AT&TSyntaxe NasmSyntaxe Masmmy_strlen:

```
mov 4(%esp), %esi
sub %ecx, %ecx
_loop:
lodsb
cmp $0, %al
je _end
inc %ecx
jmp _loop

_end:
mov %ecx, %eax
retmy_strlen:
mov esi, [esp + 4]
sub ecx, ecx
.loop:
lodsb
cmp al, 0
je .end
inc ecx
jmp .loop

.end:
mov eax, ecx
retmy_strlen:
mov esi, [esp + 4]
sub ecx, ecx
boucle:
lodsb
cmp al, 0
je fin
inc ecx
jmp boucle
fin:
```

GESTION DES CONDITIONS

Les sauts conditionnels sont la base de tout algorithme bas niveau. Cette courte introduction sur leur utilisation en assembleur est nécessaire pour la suite.

L'initiation

En assembleur, l'évaluation de conditions ne se fait pas de la même façon que dans les autres langages à l'aide de mots clés tels "if... else...", ces mots-clefs n'étant d'ailleurs qu'une représentation simplifiée de ce qui se passe en assembleur. Ce que l'on évalue, ce sont les registres. On effectue une comparaison entre deux valeurs (en registre ou en mémoire, mais jamais de mémoire à mémoire) et du résultat de cette comparaison va dépendre la suite de l'exécution du programme.

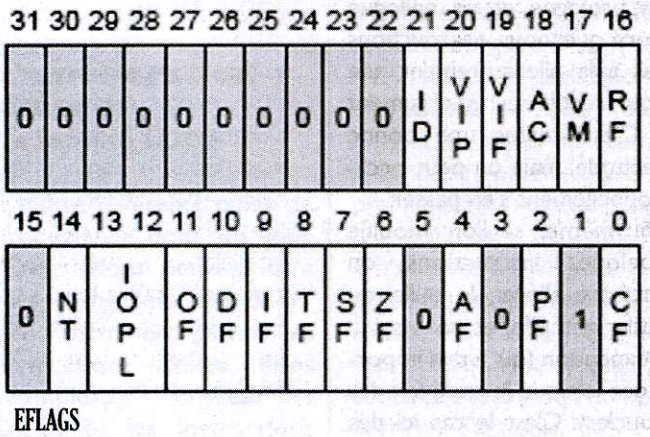
Comparer deux registres, c'est les soustraire l'un à l'autre et analyser le résultat. Par exemple si l'on compare EAX avec une donnée en mémoire, alors on regardera si le résultat est inférieur, égal, ou supérieur à zéro.

Prenez le registre EFLAGS. Les bits qui le composent représente l'état du CPU (ou du processus actuellement exécuté). L'un d'eux (ZF) sertà indiquer si le résultat du dernier calcul est égal ou non à zéro. C'est notamment sur ce flag (et également SF) que s'appuient les instructions de gestion des conditions.

CMP est la principale instruction permettant de faire des comparaisons, et l'ensemble des instructions de saut dites "Jcc" sont celles permettant à la fois d'évaluer le résultat de la comparaison et de rediriger l'exécution du programme à un endroit voulu. "Jcc" n'est pas une instruction en soi comme le serait JMP, c'est un diminutif désignant l'ensemble des instructions de sauts conditionnels ("Jump if Condition is Met").

Un exemple

Les instructions de sauts conditionnels sont nombreuses. Avant d'en aborder les principales, un exemple de code exploitant une instruction de saut conditionnel devrait vous éclairer sur la façon de les utiliser.



EFLAGS

Pour bien réaliser un saut conditionnel on doit effectuer une comparaison et, immédiatement après, réaliser le saut :

```
;Syntaxe AT&T
boucle:
 mov $7777, %eax
 cmp $7778, %eax
 jb boucle
      ...
 ret  ...

;Syntaxe Intel
boucle:
 mov eax, 7777
 cmp eax, 7778
 jb boucle
      ...
 ret
```

Dans cet exemple, la comparaison de EAX avec 7778 par CMP impose au processeur le calcul : EAX – 7778. Le résultat étant inférieur à 0, cela signifie que EAX est inférieur à 7778 (évidemment, nous avons mis la valeur 7777 dedans). L'instruction JB sert justement à évaluer le résultat. JB (Jump if Below) fait boucler le programme à l'infini en sautant à l'endroit spécifié par "boucle". Dans le cas présent, le programme fait une boucle infinie parce que 7777 est inférieur à 7778. Si l'on avait mis une valeur supérieure à 7778 dans le registre EAX, l'exécution se serait poursuivie après JB sans réalisation de saut.
Attention à la syntaxe d'écriture avec CMP ! CMP soustrait (en syntaxe Intel) le second opérande du premier. Notez également qu'on ne peut mettre une valeur immédiate (7778 dans le cas présent) qu'en seconde opérande (à droite). Des erreurs peuvent survenir lorsqu'on ne prend pas en compte ces deux points.

Les principales instructions de saut Jcc

Voici les principales instructions de saut sur conditions :

- JA : Jump if Above (saut si supérieur),
- JNA : Jump if Not Above (saut si non supérieur),
- JAE : Jump if Above or Equal (saut si supérieur ou égal),
- JB : Jump if Below(saut si inférieur),
- JNB : Jump if Not Below (saut si non inférieur),
- JBE : Jump if Below or Equal (saut si inférieur ou égal),
- JE : Jump if Equal (saut si égal),
- JNE : Jump if Not Equal (saut si non égal).

Il existe plus d'une cinquantaine de variétés de saut, mais leurs fonctions se recoupent. Faire un saut JNAE (Jump if Not Above or Equal - " saut si non supérieur ou égal à") revient à faire un saut JB (Jump if Below - "saut si inférieur à").

WILD

LES LIVRES SUR L'ASSEMBLEUR

Il existe de nombreux ouvrages numériques et papier sur le langage assembleur, mais il en est que vous vous devez absolument de connaître. Il s'agit des livres du constructeur Intel... Cet article reprend des aspects capitaux de la programmation en langage assembleur.

Tous les constructeurs de matériel fournissent en abondance une documentation gratuite et détaillée sur le fonctionnement de leurs composants et leurs vecteurs de programmation. Il est dans l'intérêt des fabricants que tout individu ou société puisse librement et indépendamment de toute contrainte développer des applications et des supports logiciels (drivers) pour un produit particulier. Le matériel se vend très souvent grâce aux logiciels qui suivent derrière, on l'avait bien compris depuis les années 80 (voir le film Pirates de la Sillicon Valley). Ainsi Windows n'est-il qu'un produit spécifique aux processeurs Intel, ce qui n'est pas sans conséquences sur les ventes de ce dernier. Grâce aux efforts du géant américain – et nous devons le reconnaître - ces livres sont un must-have. Ils sont bien structurés et clairs pour des personnes de niveau moyen, ce qui est rare pour des documentations s'adressant à des ingénieurs.

Auparavant envoyés gratuitement par la poste, les documents sont aujourd'hui disponibles sur le site officiel du constructeur, à l'adresse http://www.intel.com/design/pentium4/manuals/index_new.htm. Des documentations spécifiques aux autres familles de processeurs que l'IA-32 constituent également des ouvrages particuliers.

LDS/LES/LFS/LGS/LSS—Load Far Pointer

Opcode	Instruction	Description
C5 /r	LDS r16,m 16:16	Load DS:r16 with far pointer from memory.
C5 /r	LDS r32,m 16:32	Load DS:r32 with far pointer from memory.
0F B2 /r	LSS r16,m 16:16	Load SS:r16 with far pointer from memory.
0F B2 /r	LSS r32,m 16:32	Load SS:r32 with far pointer from memory.
C4 /r	LES r16,m 16:16	Load ES:r16 with far pointer from memory.
C4 /r	LES r32,m 16:32	Load ES:r32 with far pointer from memory.
0F B4 /r	LFS r16,m 16:16	Load FS:r16 with far pointer from memory.
0F B4 /r	LFS r32,m 16:32	Load FS:r32 with far pointer from memory.
0F B5 /r	LGS r16,m 16:16	Load GS:r16 with far pointer from memory.
0F B5 /r	LGS r32,m 16:32	Load GS:r32 with far pointer from memory.

Description

Loads a far pointer (segment selector and offset) from the second operand (source operand) into

Extrait : description d'une instruction dans le Vol 2A d'Intel.

- Le volume un de la collection « *IA-32 Intel Architecture Software Developer's Manual* » s'intitule Basic Architecture. Cet ouvrage est essentiel. Il aborde toutes les caractéristiques du fonctionnement du processeur lorsque celui-ci doit exécuter un programme. On y aborde le fonctionnement des registres, de la stack (pile), des formats de données, des instructions, etc. Mais ce volume ne décrit pas les instructions du processeur, il ne fait que les présenter.

- Le deux volumes suivants (*Vol 2A* et *Vol2B*) sont les ouvrages décrivant exhaustivement l'ensemble des instructions qu'exécute le processeur. Le volume 2A décrit toutes les instructions de A à M, et le volume 2B les instructions de N à Z.

- Le dernier volume, *System Programming Guide* décrit l'environnement d'exécution des systèmes d'exploitation, la façon de programmer avec les registres spécifiques du processeur et s'adresse aux développeurs d'OS. Il faut déjà savoir programmer en assembleur pour se pencher sur ce dernier ouvrage. C'est écrit en anglais, heureusement non littéraire, juste technique (tentez de lire l'encadré exemple pour vous faire une idée).

Vous voulez savoir s'il est possible d'en apprendre autant avec de la documentation sur Internet ou dans la littérature francophone ? Non, mais en dehors des ouvrages spécifiques sur les processeurs d'Intel, il existe des livres sur la programmation assembleur qui sont spécifiques aux systèmes d'exploitation. Il existe de bons recueils sur la programmation en assembleur pour Windows, en voici un pour démarrer :

- *Assembleur x86* par Kip Irvine aux éditions CampusPress (en français).

Vous vous demandez si la lecture de ces ouvrages (au moins du premier) suffit à permettre de se mettre immédiatement à l'assembleur ? Malheureusement, non. On ne vous l'avait pas dit, personne ne vous l'avait dit, mais connaître le langage assembleur ne suffit pas pour programmer en assembleur.

Edito

Dans ce numéro nous traitons du Cracking et du Reversing. Pourquoi ? Quel est l'intérêt de savoir cracker et mieux, savoir reverser ? Si vous savez ce qu'est le Cracking (en gros : modifier un programme afin de ne pas avoir à le payer pour l'utiliser), savez vous vraiment ce qu'est le Reversing ?

Si le Cracking - qui n'est qu'un sous-domaine minime du Reversing - n'apporte pas grand chose du point de vue de la compréhension de votre OS, le Reversing consiste à « retourner » un programme afin de l'analyser. Cela peut rendre de multiples besoins : on peut modifier les fonctionnalités d'un programme, on peut analyser le fonctionnement d'un programme dont on n'a pas les sources et surtout, le Reversing permet une meilleur compréhension de votre système d'exploitation. En faisant du reversing vous comprendrez mieux le fonctionnement d'un exécutable, les relations qui existent entre cet exécutable, les DLL et Windows (le reversing peut aussi s'appliquer à Linux).

En poursuivant cette logique, on arrive à des points très importants : le Reversing permet d'analyser le fonctionnement de programmes propriétaires, d'analyser un programme suspect et donc (le rapport est implicite) de pouvoir appréhender des points stratégiques de la sécurité informatique comme l'injection de processus, le détournement d'API (API Hooking), la création de RootKit et beaucoup d'autre domaines très important pour la sécurité. Imaginez en effet les dommages causés à votre firewall si l'on change des règles essentielles à la sécurité de votre réseau ou si l'on fait disparaître les alertes de votre antivirus.

Ceci dit pour arriver à un niveau correct, voire bon en Reversing, commencer par le Cracking est l'approche la plus commune et la plus pédagogique du point de vue de l'évolution des connaissances.

La plupart des groupes qui font du Reversing ont commencé en faisant du Cracking. Et cela en partie parce qu'être membre d'une équipe permet de partager ses connaissances, chacun alors bénéficie de l'apport des un et des autres. L'aspect illégal de ces activités n'est somme toute très secondaire pour ces personnes.

Nous vous rappelons que cracker un logiciel est illégal en France et dans la plupart des pays, même si on ne s'en sert pas. Si c'est pour s'en servir sans payer, c'est encore pire : utilisez des logiciels libres, plutôt !

Demandez les codes source complets sur notre forum Cracking :
http://forum.thehackademy.net

Sommaire

Partie 1 : Introduction

Apprendre l'assembleur **p.4**

La faiblesse des protections par serial **p.8**

Liens utiles p.13Cracker les protections par checksum **p.14**

PE Header, l'incontournable en cracking Windows **p.17**

Dumper un exe avec SoftICE **p.19**

Partie 2 : Reversing

Programmer un packer **p.22**

Patcher un programme sans les sources **p.31**

Déjouer IDA/OllyDBG/WDasm **p.34**

Injex : L'arme absolue du reverser **p.44**

Partie 3 : Au delà du reversing

L'esentiel de l'API Hooking **p.52**

Analyser un virus Windows sur Linux **p.56**

HACKADEMY HORS SERIE

est édité par DMP,
26 bis rue Jeanne d'Arc 94160 St Mandé

Rédaction : Silkscalp
Conception Graphique : Weel
Couverture : Lechatkitu
ISSN en cours
numéro de comission paritaire en cours
Directeur de publication : Olivier Spinelli

Imprimé en France par roto champagne

© DMP

www.thehackademy.net

HACKADEMY HORS SERIE #2

CRACKING
REVERSE ENGINEERING

L 12890 - 2 H - F: 5,00 € - RD

Janvier - Février 2006 5 Euro

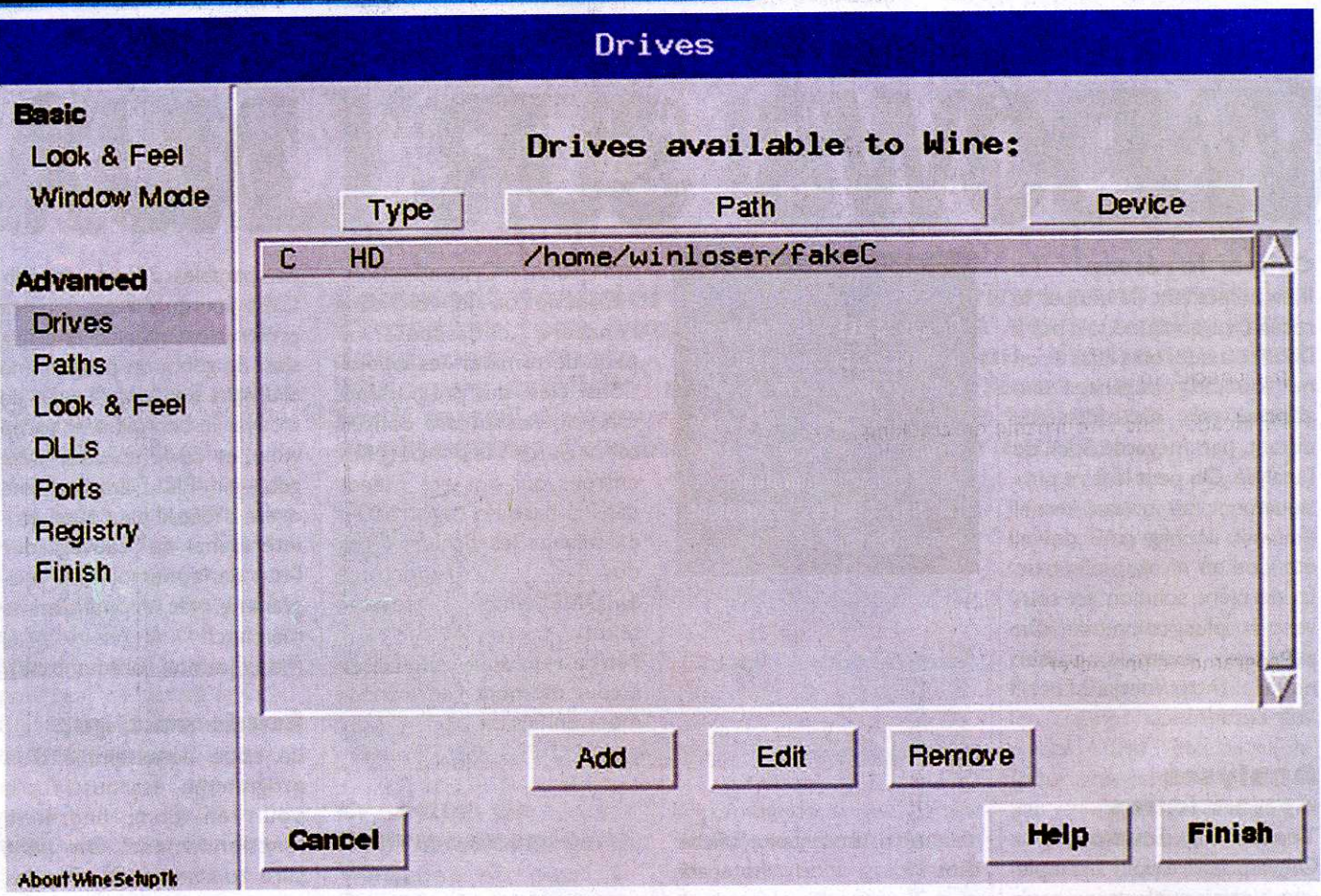
Tout sur le cracking, reversing et plus si affinité

Serials, checksums, PE Header, dumpers, packers, patcher sans les sources

Utiliser et empêcher d'utiliser

IDA/OllyDBG/WDasm

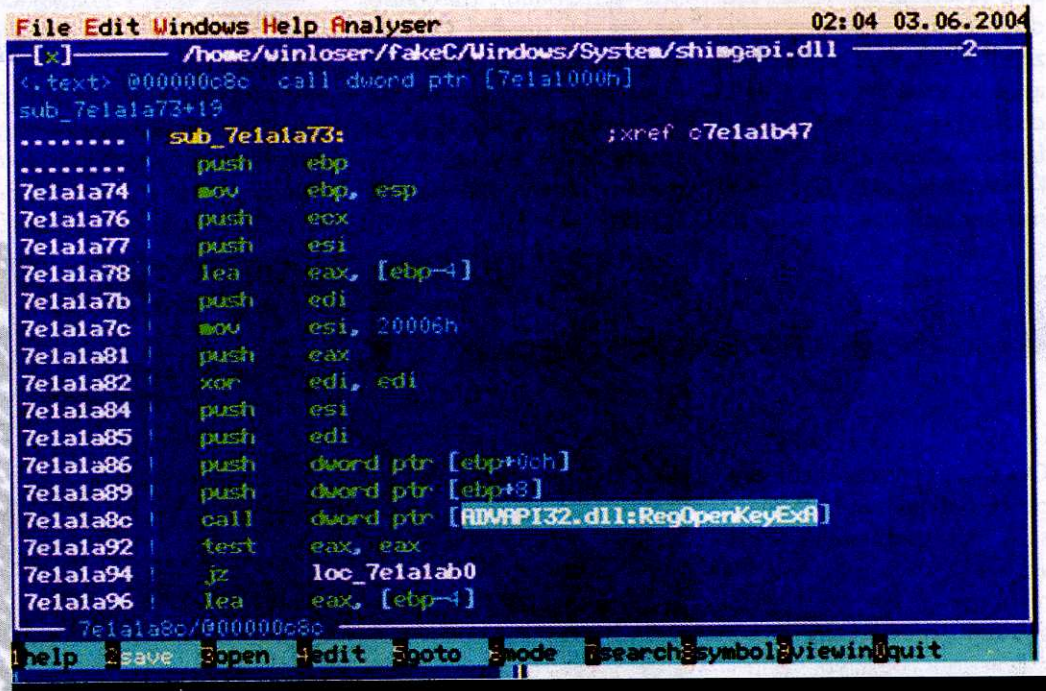
Quelle influence sur la sécu informatique ?



désassemblée qui fait appel à RegOpenKeyExA.

Ces méthodes ont leurs limites. Il est parfois nécessaire d'avoir un vrai système Windows pour que le programme à étudier fonctionne correctement (techniques anti-débogue, incompatibilités de wine). Mais le point de vue que donne Linux sur les virus Windows est presque toujours un avantage.

By dvrasp



Autres outils

Les outils de reversing sur Linux sont nombreux et en général installés par défaut. On peut citer, dans le genre sobre : gdb, strace, ltrace, objdump, strings, hexdump, etc. Allez voir leur man pages.Dans le genre plus poussé, on peut s'intéresser à des outils plus puissants comme fenris ou elfsh. Une liste assez complète est disponible ici :http://lcamtuf.coredump.cx/fenris/other.txt

Magazines
chat
forum

thehackademy.net