SYSDREAM

IT Security Services

Analyse d'un *packer* de *trojan* et programmation d'un *unpacker*

Baboon

<baboon@sysdream.com>

© Sysdream - 2008

Introduction

Une technique très utilisées par les antivirus est la détection par signature. Lorsqu'un *malware* possède une ou plusieurs portions de code qui sont toujours identiques, l'antivirus en extrait la séquence d'octets correspondant et s'en sert par la suite pour détecter la présence du *malware*.

Pour lutter contre la détection par signature les *trojans* sont souvent *packés* à l'aide d'un *packer*. Un *packer* est un programme qui compresse , chiffre, protège les exécutables. Pour que les exécutables puissent être lancés, le *packer* leur ajoute une fonction, un loader, qui va décompresser et déchiffrer l'exécutable en mémoire avant de lui rendre la main. Les détections par signatures se faisant sur l'exécutable 'en dur', c'est à dire sur le fichier et non en mémoire lorsque le programme est lancé, le code du *malware* est chiffré et la détection par signature échoue.

Pour analyser ces *malwares* il est nécessaire de les *un-packer*, de reconstruire l'exécutable pour qu'il retrouve sa forme originale et que l'on puisse accéder directement à son code.

La programmation d'un *un-packer* permet d'accélérer l'étape de l'*un-packing* en l'automatisant, il est alors plus rapide d'analyser les variantes d'un *trojan packées* avec le même *packer*.

Pour faire notre *un-packer*, nous allons d'abord analyser le *packer* afin de comprendre son fonctionnement précis, puis nous allons rechercher les points 'sensibles' du *packer*, par exemple l'adresse du saut qui va rendre la main au programme, l'endroit où les *imports* du programme sont résolus, etc., ensuite nous allons rechercher des signatures dans le *packer* qui nous permettrons de retrouver ces points, enfin nous programmerons notre *un-packer* qui utilisera les *debug* APIs de Windows ce qui nous permettra de contrôler l'exécution du *packer* et de *dumper* le processus au bon moment.

Cet article explique...

Comment analyser un *packer* simple.

Comment programmer un *un-packer*.

Ce qu'il faut savoir...

Les bases de l'assembleur x86.

Se servir de OllyDBG (une grande maitrise n'est pas nécessaire).

Programmer en C.

Des notions sur le format PE (*Portable Executable*).

Baboon

Baboon est autodidacte, il pratique le Reverse Engineering depuis maintenant trois ans. Membre de la FAT (*French Assembler Team*), il fréquente plusieurs forums dédiés au Reverse et en stage chez Sysdream lors de la rédaction de cet article. Pour le contacter : baboon@lyua.org ou baboon@sysdream.com

© Sysdream - 2008

Étude du loader

Avant de programmer notre *un-packer* il est nécessaire d'étudier le *loader* (la fonction logée dans l'exécutable qui déchiffra le *malware* au chargement du programme).

Nous allons pour cela d'abord *un-packer* manuellement notre *malware*. Inutile de préciser que lorsque l'on étudie un code malicieux il faut le faire dans une machine virtuelle qui nous permettra en cas d'exécution malencontreuse du *malware* de ne pas avoir à formater notre disque dur, il nous suffira de restaurer l'état de la machine virtuelle avant l'infection grâce à un *snapshot*. Nous ouvrons donc le binaire packé avec OllyDBG et nous pouvons commencer l'analyse.

Anti-Émulation

Le *malware* essaie de freiner et de détecter l'émulation de son code par les antivirus en exécutant 20500 fois l'API TlsFree avec de mauvais arguments et en vérifiant le code d'erreur retourné. Si le code d'erreur est bien celui attendu (ERROR_INVALID_PARAMETER 0x57) alors l'exécution du virus se poursuit, sinon elle s'arrête.

Cet anti-antivirus n'est pas très utile, en effet le code du *packer* n'est pas polymorphique et il est facile d'en extraire une signature qui le détectera à coup sur sans avoir à l'émuler. Ce genre de technique n'est intéressante que dans le cas de *malwares* aboutis, comme protection supplémentaire, seule, la protection que cet anti-émulateur offre est nulle. Enfin lorsqu'un antivirus rencontre un saut conditionnel, il analyse généralement les deux branches du saut.

Déchiffrement

Le *malware* déchiffre ensuite son code qui a été chiffré au moment où il a été *packé*. Chaque octet est chiffré à l'aide d'opération réversibles (multiplication, addition, ou exclusif). Enfin il recherche les adresses des APIs qui vont être utilisées par la fonction qui décompressera le code.

```
004D110A  v EB 07          JMP SHORT 2101f702.004D1113
004D110C    8B45 D0        MOV EAX,DWORD PTR SS:[EBP-30]
004D110F    40             INC EAX
004D1110    8945 D0        MOV DWORD PTR SS:[EBP-30],EAX
004D1113    8B45 E8        MOV EAX,DWORD PTR SS:[EBP-18]
004D1116    8B4D D0        MOV ECX,DWORD PTR SS:[EBP-30]
004D1119    3B48 04        CMP ECX,DWORD PTR DS:[EAX+4]
004D111C  v 73 4F          JNB SHORT 2101f702.004D116D
004D111E    8B45 D4        MOV EAX,DWORD PTR SS:[EBP-2C]
004D1121    0345 D0        ADD EAX,DWORD PTR SS:[EBP-30]
004D1124    0FB600         MOVZX EAX,BYTE PTR DS:[EAX]
004D1127    0FB64D D0      MOVZX ECX,BYTE PTR SS:[EBP-30]
004D112B    6BC9 51        IMUL ECX,ECX,51
004D112E    33C1           XOR EAX,ECX
004D1130    8B4D D4        MOV ECX,DWORD PTR SS:[EBP-2C]
004D1133    034D D0        ADD ECX,DWORD PTR SS:[EBP-30]
004D1136    8801           MOV BYTE PTR DS:[ECX],AL
004D1138    8B45 D4        MOV EAX,DWORD PTR SS:[EBP-2C]      boucle de déchiffrement
004D113B    0345 D0        ADD EAX,DWORD PTR SS:[EBP-30]
004D113E    0FB600         MOVZX EAX,BYTE PTR DS:[EAX]
004D1141    0FB64D EF      MOVZX ECX,BYTE PTR SS:[EBP-11]
004D1145    03C1           ADD EAX,ECX
004D1147    8B4D D4        MOV ECX,DWORD PTR SS:[EBP-2C]
004D114A    034D D0        ADD ECX,DWORD PTR SS:[EBP-30]
004D114D    8801           MOV BYTE PTR DS:[ECX],AL
004D114F    8B45 D4        MOV EAX,DWORD PTR SS:[EBP-2C]
004D1152    0345 D0        ADD EAX,DWORD PTR SS:[EBP-30]
004D1155    0FB600         MOVZX EAX,BYTE PTR DS:[EAX]
004D1158    8B4D F8        MOV ECX,DWORD PTR SS:[EBP-8]
004D115B    0FB649 08      MOVZX ECX,BYTE PTR DS:[ECX+8]
004D115F    33C1           XOR EAX,ECX
004D1161    F7D0           NOT EAX
004D1163    8B4D D4        MOV ECX,DWORD PTR SS:[EBP-2C]
004D1166    034D D0        ADD ECX,DWORD PTR SS:[EBP-30]
004D1169    8801           MOV BYTE PTR DS:[ECX],AL
004D116B  ^ EB 9F          JMP SHORT 2101f702.004D110C
004D116D    8B45 E8        MOV EAX,DWORD PTR SS:[EBP-18]
004D1170    83C0 08        ADD EAX,8                          Ici le code du décompresseur est déchiffré...
004D1173    8945 E8        MOV DWORD PTR SS:[EBP-18],EAX
004D1176  ^ E9 6AFFFFFF    JMP 2101f702.004D10E5
004D117B    8B45 F8        MOV EAX,DWORD PTR SS:[EBP-8]
004D117E    8338 00        CMP DWORD PTR DS:[EAX],0
004D1181  v 0F84 C7000000  JE 2101f702.004D124E
004D1187    8B45 F8        MOV EAX,DWORD PTR SS:[EBP-8]
004D118A    8B00           MOV EAX,DWORD PTR DS:[EAX]
004D118C    0345 E0        ADD EAX,DWORD PTR SS:[EBP-20]
004D118F    8945 CC        MOV DWORD PTR SS:[EBP-34],EAX
EAX=004D1350 (2101f702.004D1350)

Address   Hex dump                 ASCII
00401000  B8 24 0D 4D 00 50 64 FF  ©$.M.Pd
00401008  35 00 00 00 00 64 89 25  5....dë%
00401010  00 00 00 00 33 C0 89 08  ....3└ë■
00401018  50 45 43 6F 6D 70 61 63  PECompac  ) ... et on voit apparaitre le nom du
00401020  74 32 00 C4 A3 9C 96 C5  t2.─ú£û┼      compresseur qui a été utilisé
00401028  C9 21 9E 61 24 D4 28 90  ╔!×a$È(É
```

Décompression et résolution des imports

La compression a été faite grâce a PECompact (on voit le nom du *packer* en mémoire après le déchiffrement), ce *packer* obtient un fort taux de compression , l'exécutable passe de 828Ko avant compression à 127Ko. PECompact met à la disposition du programmeur un système de module qui lui permet d'agir avant ou après la décompression du programme, il est fort probable que le déchiffrement du code et l'anti-antivirus soient intégrés dans un module programmé par le pirate, il est plus simple de développer un module qu'un *packer* de A à Z.

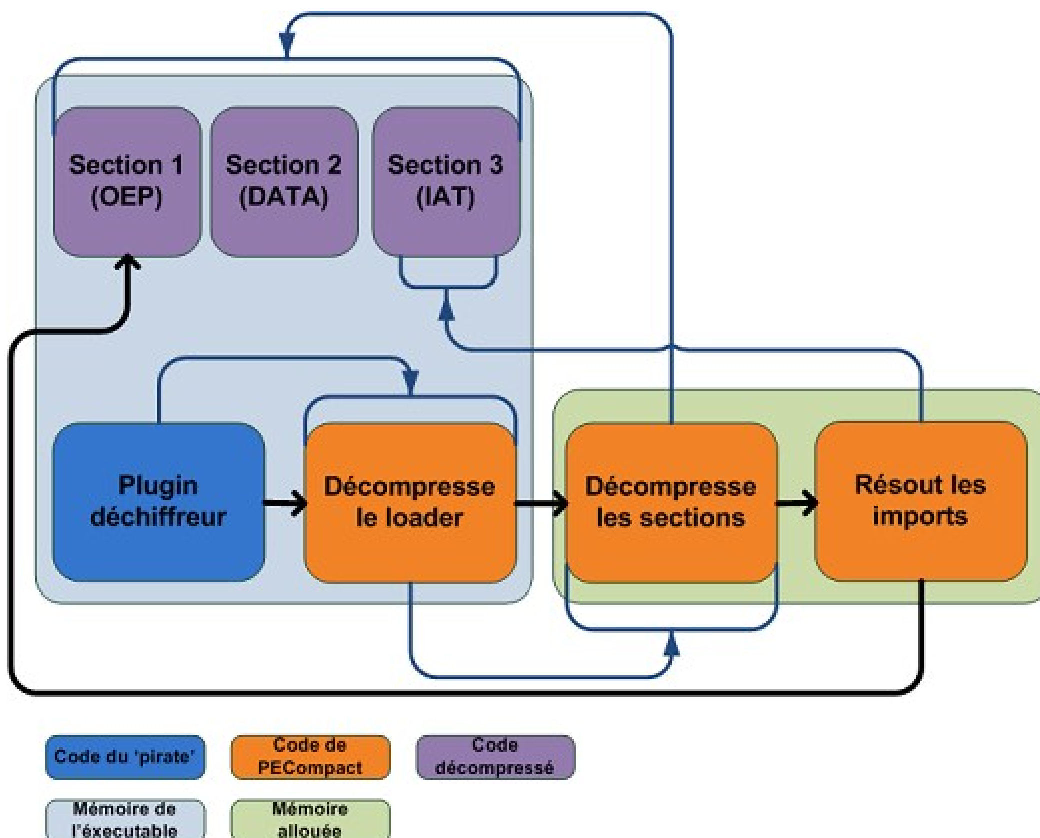
PECompact est un compresseur, pas un *protector*, il est donc facile à *un-packer*. La décompression se déroule selon ce schéma :

© Sysdream - 2008

- 1 – déclenchement d'une exception de type ACCESS_VIOLATION (le processus essaie d'écrire à une adresse nulle),
- 2 – l'exception est gérée par un *handler* précédemment mis en place qui va rediriger le processus sur la fonction de décompression à proprement parler,
- 3 – la fonction de décompression va allouer une page mémoire dans laquelle va être décompressée le code du loader,
- 4 – la fonction décompressée va charger les APIs qui vont être utilisées par le *loader*,
- 5 – un contrôle d'intégrité est effectué sur le code afin de s'assurer qu'il n'a pas été corrompu,
- 5 – les sections de l'application sont décompressées puis copiées a leurs emplacements,
- 6 – les *imports* sont résolus,
- 7 – les droits des sections sont ajustés,
- 8 – le *loader* rend la main au code décompressé.

L'*Import Table* n'est pas détruite par PECompact, elle est juste compressée et les champs FirstChunk supprimés. Généralement les *protector* détruisent l'IT et modifient l'IAT de manière à compliqué *l'unpacking*. Pour résoudre les *imports*, PECompact va parcourir l'*Import Table* décompressée, charger les DLLs nécessaires et rechercher les adresses des APIs dont le programme aura besoin.

Schéma simplifié du fonctionnement du packer :



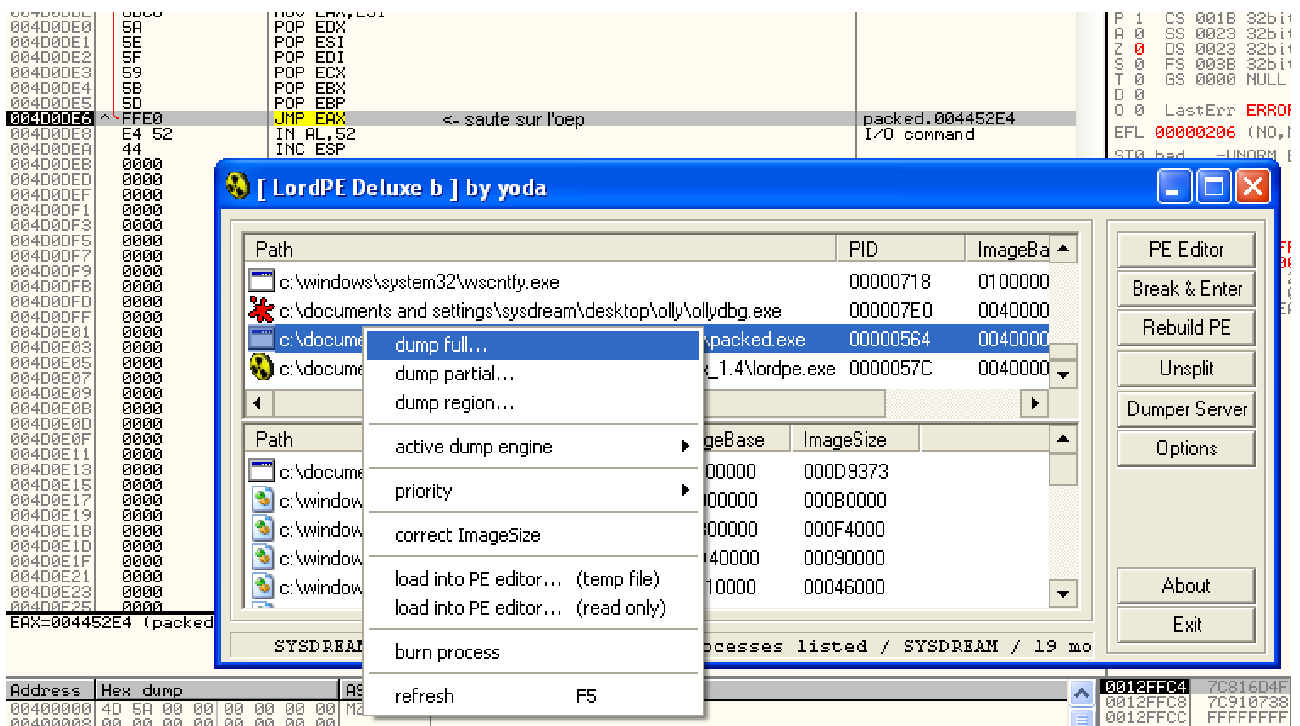
Unpacking manuel

Nous allons maintenant *un-packer* l'exécutable manuellement. Une fois que les sections auront été déchiffrées et décompressées, nous les enregistrerons sur le disque puis nous reconstruirons l'*ImportTable*.

Dumper *l'exécutable en mémoire*

Une fois que le code est décompressé en mémoire, juste avant que le loader du packer rende la main au malware, nous allons *dumper* la mémoire, c'est à dire que nous allons copier en mémoire les *headers* (le DOS *header*, le PE *header* et les Section *headers*) ainsi que les sections et les 'coller' dans un fichier grâce a LordPE.

Nous ouvrons notre binaire *packé* avec OllyDBG, nous traçons jusqu'au JMP EAX qui va sauter sur l'OEP (*Original Entry Point*, le point d'entré original du *malware*, celui qu'il avait avant d'être *packé*) et nous *dumpons* le processus avec LordPE :



*Reconstruire l'***Import Table**

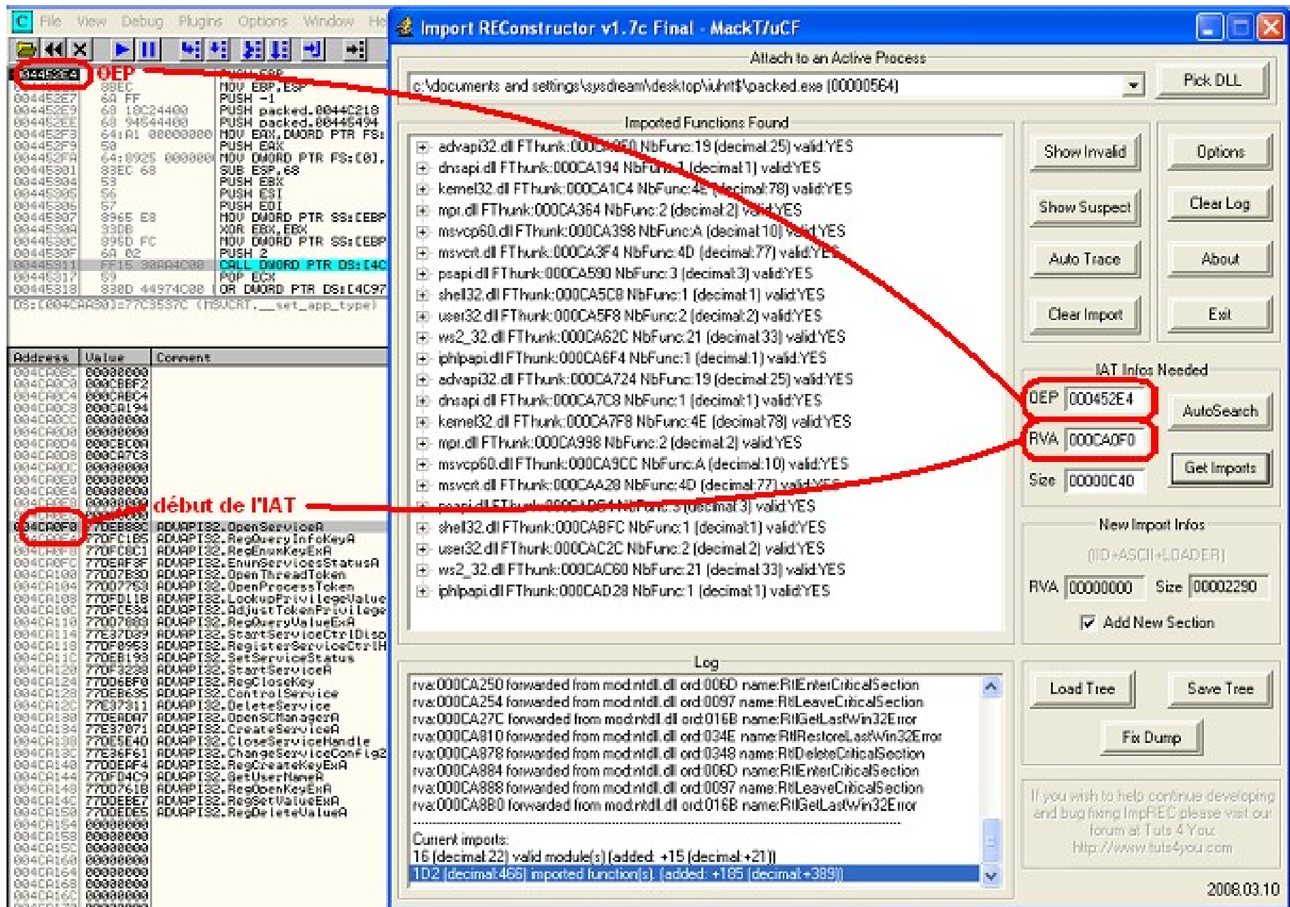
Maintenant que nous avons *dumpé* notre binaire en mémoire, nous devons modifier certains champs du PE et reconstruire l'*Import Table* de notre *malware*.

Pour cela il y a deux possibilités :

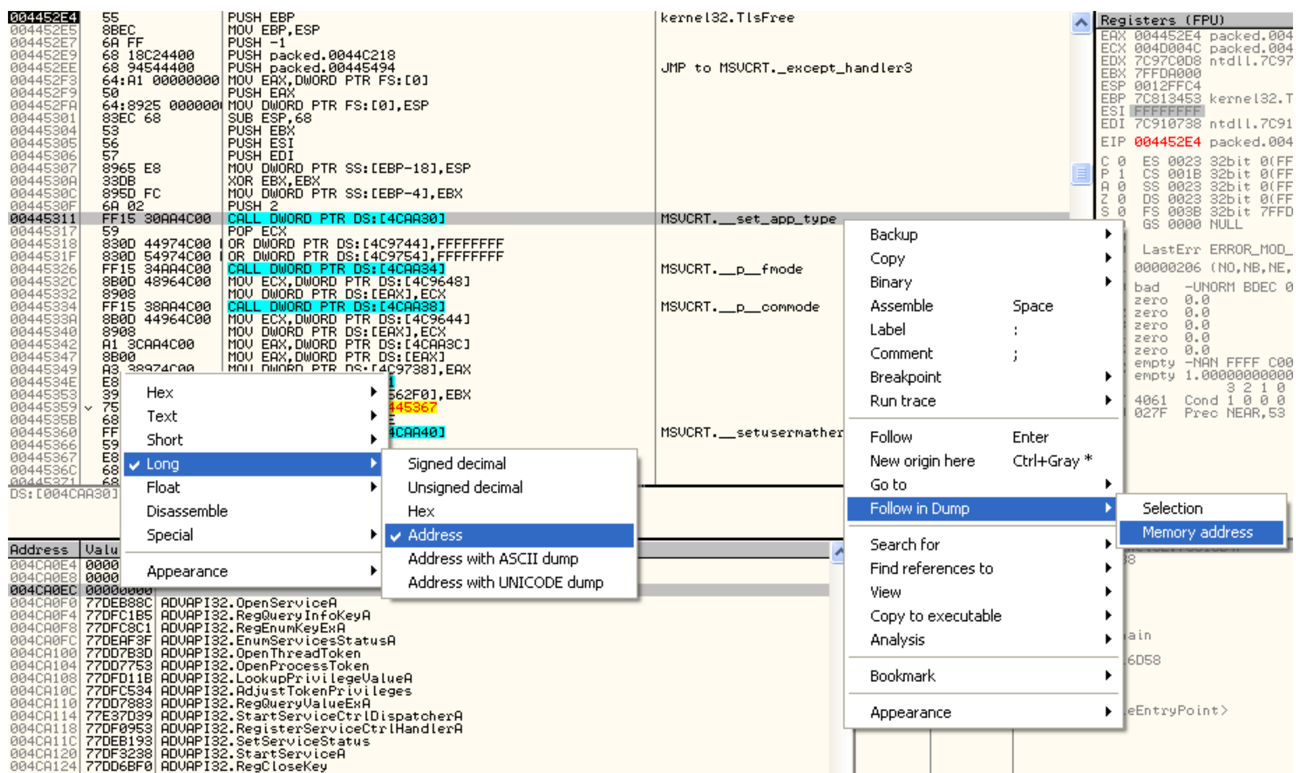
- reconstruire les champs FirstThunk qui ont été détruits par le *packer*.
- utiser ImpRec qui va construire une *Import Table* à partir de l'*Import Address Table* (la table qui contient les adresses des APIs importées).

© Sysdream - 2008

Nous utiliserons la première technique dans notre *un-packer* et la deuxième pour *un-packer* manuellement. ImpRec est très simple d'utilisation, il suffit de lui fournir l'adresse virtuelle de l'IAT (l'adresse que l'on voit sous OllyDBG – l'*Image Base* (0x400000 la plupart du temps)) et sa taille, ImpRec ajoutera une section à l'exécutable dans laquelle il stockera l'IT. ImpRec permet aussi de modifier l'EP (Entry Point) par l'OEP si on lui fournit.



Pour trouver cette IAT, il nous suffit de repérer un call [API], de nous rendre là où est stockée l'adresse de l'API et de remonter en mémoire jusqu'à ce que l'on ne voit plus d'adresse d'API. On peut reprocher a cette méthode de ne pas être très conventionnelle mais elle a le mérite d'être simple, rapide et efficace.



Après reconstruction, nous avons un exécutable décompressé et parfaitement fonctionnel.

Maintenant que nous avons compris comment fonctionnait l'intégralité du loader et que nous avons réussi à *un-packer* manuellement notre binaire, nous allons pouvoir passer à la programmation de notre *un-packer*.

*Programmation de l'*un-packer

Notre *un-packer* va travailler exactement comme nous l'avons fait, il déboguera l'exécutable, récupérera les valeurs utiles, cherchera des signatures, *dumpera* l'exécutable en mémoire et reconstruira l'IAT.
Pour déboguer le processus, nous utiliserons des APIs que Windows met à notre disposition, les *debug* APIs.

Les Debug *APIs*

- 1 – Pour créer un processus en mode *debug*, il suffit d'appeler l'API CreateProcess avec l'argument *dwCreationFlags* égale à DEBUG_PROCESS (0x1)
- 2 – Une fois que le processus a été créé, nous pouvons commencer a recevoir des *events* (des messages) grâce à l'API WaitForDebugEvent, ces *events* nous donnent des informations sur le processus, sur ses *threads*, sur les modules qu'il charge et enfin ils nous renseignent lorsque le processus rencontre une exception ce qui nous permettra plus tard de poser des BPs (des Break Points ou points d'arrêts).
- 3 – Pour contrôler l'exécution du loader nous utiliserons des HBP (Hardware Break Point). Poser un HBP nécessite de modifier les DRs (*Debug Registers*) du *thread* principal, nous utiliserons pour ce faire les APIs GetThreadContext et SetThreadContext

© Sysdream - 2008

```c
<<LISTING lang=C >>
// Listing 1. Créer un processus en mode debug
STARTUPINFOA StartupInfo;
PROCESS_INFORMATION ProcessInfo;
LPCTSTR path = "packed.exe";
StartupInfo.cb = sizeof(STARTUPINFO);
StartupInfo.lpReserved = NULL;
StartupInfo.lpDesktop = NULL;
StartupInfo.lpTitle = NULL;
StartupInfo.dwFlags = NULL;
StartupInfo.cbReserved2 = 0;
StartupInfo.lpReserved2 = NULL;
CreateProcessA(path,NULL,NULL,NULL,FALSE,DEBUG_ONLY_THIS_PROCESS |
DEBUG_PROCESS , NULL , NULL , &StartupInfo , &ProcessInfo)
<</LISTING>>
```

```c
<<LISTING lang=C >>
// Listing 2. Poser un Hardware Break Point
#define OneByteLength   00
#define TwoByteLength   01
#define FourByteLength   3
#define BreakOnExec  0
#define BreakOnWrite  1
#define BreakOnAccess  3
#define GlobalFlag  2
#define LocalFlag  1
#define DR7flag(_size,_type,flag,HBPnum) (((_size<<2 | _type)  << (HBPnum*4 +16)) | (flag <<
(HBPnum*2)))
CONTEXT Context;
DWORD  addrHBP;
DEBUG_EVENT  DbgEvt;
HANDLE hThread;
Context.ContextFlags = CONTEXT_ALL;
GetThreadContext(hThread,&Context);
Context.Dr0 = addrHBP;
// DR7flag est une macro permettant de calculer la
// valeure de DR7 suivant le type de HBP
// que l'on souhaite poser.
Context.Dr7 = DR7flag(OneByteLength,BreakOnExec,GlobalFlag | LocalFlag,0);
SetThreadContext(hThread,&Context);
<</LISTING>>
```

© Sysdream - 2008

<<LISTING lang=C>>

```c
// Listing 3. Attendre une exception SingleStep (indique que le programme a atteint un HBP)
int WaitForSingleStepExc(void)
{
   ContinueDebugEvent(DbgEvt.dwProcessId,DbgEvt.dwThreadId,DBG_CONTINUE);
   while(WaitForDebugEvent(&DbgEvt,INFINITE))
   {
      if (DbgEvt.dwDebugEventCode == EXCEPTION_DEBUG_EVENT)
      {
         if (DbgEvt.u.Exception.ExceptionRecord.ExceptionCode == EXCEPTION_SINGLE_STEP)
            return 1;
         else
            ContinueDebugEvent(DbgEvt.dwProcessId,DbgEvt.dwThreadId,DBG_EXCEPTION_NOT_HAN
DLED);
      }
      else
         ContinueDebugEvent(DbgEvt.dwProcessId,DbgEvt.dwThreadId,DBG_CONTINUE);
   }
   return 0;
}
```

<</LISTING>>

La détection du packer

Il faut être sûr d'avoir affaire au bon *packer* ou notre *un-packer* ne fonctionnera pas et sera susceptible d'exécuter un programme malveillant, en effet, si nous posons des BPs et que nous lançons le processus, il y a très peu de chance pour qu'il break au mêmes adresses et le programme se lancera.

Pour identifier le *packer*, nous utiliserons plusieurs signatures, les signatures sont des suites d'octets invariantes typiques du *packer*, ainsi à l'EP du programme, nous aurons toujours la suite d'instruction PUSH EBP | MOV EBP,ESP | XCHG ESP,EBP | POP EBP ce qui se traduit par la suite d'octet 0x55, 0x8B, 0xEC, 0x87, 0xEC, 0x5D pour savoir si notre exécutable est bien *packé* par le bon *packer*, il nous suffit donc de rechercher cette suite d'octets à son EP, pour plus de précision, nous vérifierons aussi qu'une section *.nah* est présente dans le binaire (cette section est celle qui contient le code du *loader* et son nom est tout le temps le même).

<<LISTING lang=C>>

```c
// Listing 4. Mapper le binaire en mémoire et vérifier qu'il comporte une section .nah
LPCTSTR path = "packed.exe";
HWND hwndDlg;
HANDLE hFile, hMapp;
char* mapping;
PIMAGE_DOS_HEADER pDosHeader;
PIMAGE_NT_HEADERS pPE;
```

```
PIMAGE_SECTION_HEADER pSection;
hFile = CreateFileA(path, GENERIC_READ , FILE_SHARE_READ , NULL , OPEN_EXISTING ,
NULL , NULL);
if (hFile == INVALID_HANDLE_VALUE)
{
   MessageBoxA(hwndDlg,"Echec lors de l'ouverture du fichier", NULL , MB_ICONERROR);
   return 0;
}
hMapp = CreateFileMapping(hFile, NULL , PAGE_READONLY , 0 , 0 , NULL);
mapping = (char *)MapViewOfFile(hMapp,FILE_MAP_READ,0,0,0);
if (mapping == NULL)
{
   MessageBoxA(hwndDlg,"Echec lors du mapping du fichier", NULL , MB_ICONERROR);
   CloseHandle(hMapp);
   CloseHandle(hFile);
   return 0;
}
pDosHeader = (PIMAGE_DOS_HEADER)mapping;
if (pDosHeader->e_magic != 'ZM')
{
   MessageBoxA(hwndDlg,"Dos header non valide", NULL , MB_ICONERROR);
   CloseHandle(hMapp);
   CloseHandle(hFile);
   return 0;
}
pPE = (PIMAGE_NT_HEADERS)(pDosHeader->e_lfanew + mapping);
if (pPE->Signature != 'EP')
{
   MessageBoxA(hwndDlg,"PE header non valide", NULL , MB_ICONERROR);
   CloseHandle(hMapp);
   CloseHandle(hFile);
   return 0;
}
pSection = (PIMAGE_SECTION_HEADER)((PCHAR)pPE + sizeof(IMAGE_FILE_HEADER) + pPE-
>FileHeader.SizeOfOptionalHeader + sizeof(DWORD));
do
{
   if (pSection->VirtualAddress == 0)
   {
      MessageBoxA(hwndDlg,"La signature .nah n'a pas été trouvé dans les noms des sections", NULL ,
MB_ICONINFORMATION);
      CloseHandle(hMapp);
      CloseHandle(hFile);
      return 0;
   }
```

© Sysdream - 2008

```
    if (*(PDWORD)pSection->Name == 'han.')
        break;
    pSection = (PIMAGE_SECTION_HEADER)((PCHAR)pSection +
sizeof(IMAGE_SECTION_HEADER));
}
while (1);
CloseHandle(hMapp);
CloseHandle(hFile);
return mapping;
```

<</LISTING>>

Dumper *l'exécutable*

Nous sommes maintenant sûrs d'avoir affaire au bon *packer*, il nous faut maintenant localiser l'instruction du *loader* après l'exécution de laquelle le *malware* est présent sous sa forme déchiffrée en mémoire alors que les imports ne sont pas encore résolus (lorsque les *imports* sont résolus par le loader, les champs *OriginalFirstChunk* sont écrasés par les adresses des APIs)..
Comme nous l'avons vu, la décompression et la résolution des *imports* se fait dans une zone mémoire allouée, donc à une adresse que l'on ne peut pas déterminer statiquement. Pour localiser la procédure de reconstruction, nous devrons donc analyser dynamiquement le loader via les *debug* API :

- 1 –se rendre à l'EP du binaire,
- 2 – poser un HBP sur l'API VirtualAlloc,
- 3 – attendre que le programme l'atteigne, récupérer la taille de la future page mémoire, poser un HBP sur l'adresse de retour de l'appel,
- 4 –attendre que le processus ai fini l'appel à VirtualAlloc, récupérer sa valeur de retour (donc l'adresse de la page mémoire allouée),
- 5 – à partir de l'adresse de retour de VirtualAlloc, retrouver le JMP EAX qui sautera sur l'OEP ainsi que le CALL EDI qui appellera la fonction de décompression et de résolution des *imports*,
- 6 – poser un HBP sur le CALL EDI,
- 7 – attendre que le processus l'atteigne, la page mémoire allouée contient alors le code du loader,
- 8 –scanner la page mémoire à la recherche de la signature qui indique l'endroit où le loader a finit de décompresser le code et où il s'apprête à résoudre les *imports*,
- 9 – poser un HBP à cette adresse et attendre que le processus l'atteigne,
- 10 – *dumper* l'exécutable en mémoire.

© Sysdream - 2008

<<LISTING lang=C>>

```c
// Listing 5. Arriver à l'endroit où le loader a fini de décompresser le code et avant qu'il ne résolve les pointeurs
// on break à l'EP
Goto((pPE->OptionalHeader.AddressOfEntryPoint + BaseAddress),ProcessInfo.hThread);
// on break sur le VirtualAlloc
hModKern = GetModuleHandleA("Kernel32");
Goto((DWORD)GetProcAddress(hModKern,"VirtualAlloc"),ProcessInfo.hThread);
// on récupère l'adresse de retour du call VirtualAlloc ainsi que la taille de la future page
GetThreadContext(ProcessInfo.hThread,&Context);
ReadProcessMemory(ProcessInfo.hProcess,
(LPCVOID)Context.Esp,&RetnAdd,sizeof(DWORD),&NumberOfBytesRead);
if (NumberOfBytesRead != sizeof(DWORD))
{
   MessageBoxA(hwndDlg,"ReadProcessMemory a échouée", NULL , MB_ICONINFORMATION);
   return TRUE;
}
ReadProcessMemory(ProcessInfo.hProcess,(LPCVOID)
(Context.Esp+8),&AllocSize,sizeof(DWORD),&NumberOfBytesRead);
if (NumberOfBytesRead != sizeof(DWORD))
{
   MessageBoxA(hwndDlg,"ReadProcessMemory a échouée", NULL , MB_ICONINFORMATION);
   return TRUE;
}
// on récupère la valeur de retour de VirtualAlloc
Goto(RetnAdd,ProcessInfo.hThread);
GetThreadContext(ProcessInfo.hThread,&Context);
addrAlloc = Context.Eax;
pJmpToOEP = RetnAdd+0x69;
pCallEdi = RetnAdd+0x45;
//on vérifie des signatures
ReadProcessMemory(ProcessInfo.hProcess,(LPCVOID)
(pJmpToOEP-2),&Sign,sizeof(DWORD),&NumberOfBytesRead);
if (NumberOfBytesRead != sizeof(DWORD))
{
   MessageBoxA(hwndDlg,"ReadProcessMemory a échouée", NULL , MB_ICONINFORMATION);
   return TRUE;
}
if(Sign != 0xE0FF5D5B )
{
   MessageBoxA(hwndDlg,"Signature non conforme", NULL , MB_ICONINFORMATION);
   return TRUE;
}
//on va jusqu'au call qui se chargera de décompresser le code et résoudre les imports
```

© Sysdream - 2008

```
Goto(pCallEdi,ProcessInfo.hThread);
//une fois qu'on est a ce call, la mémoire allouée contient le code du loader
//on peut donc la scanner a la recherche d'une signature qui nous donnera l'adresse où le code est
décompressé et où les imports ne sont pas encore résolus
PackerMem = (PCHAR)VirtualAlloc(NULL,AllocSize,MEM_COMMIT |
MEM_RESERVE,PAGE_READWRITE);
ReadProcessMemory(ProcessInfo.hProcess,
(LPCVOID)addrAlloc,PackerMem,AllocSize,&NumberOfBytesRead);
if (NumberOfBytesRead != AllocSize)
{
   MessageBoxA(hwndDlg,"ReadProcessMemory a échouée", NULL , MB_ICONINFORMATION);
   return TRUE;
}
addrImportResolver = SearchSign(PackerSign,sizeof(PackerSign)-1, PackerMem ,AllocSize);
if (addrImportResolver == 0)
{
   MessageBoxA(hwndDlg,"La recherche de la signature du resolveur d'import a echoué", NULL ,
MB_ICONINFORMATION);
   return TRUE;
}
addrImportResolver += addrAlloc-(DWORD)PackerMem-5;
Goto(addrImportResolver,ProcessInfo.hThread);
// là le processus a fini de décompresser le code il nous reste plus qu'à le dumper
// de plus l'adresse de l'IT se trouve dans Ecx
GetThreadContext(ProcessInfo.hThread,&Context);
addrIT = Context.Ecx – BaseAddress;
// on dump le processus
Dump = DumpProcess(ProcessInfo.hProcess,(char*)BaseAddress);
```

<</LISTING>>

<<LISTING lang=C>>

```
//Listing 6. Exécuter le code jusqu'à une adresse
int Goto(DWORD addr,HANDLE hThread)
{
   CONTEXT Context;
   int RetnValue;

   Context.ContextFlags = CONTEXT_ALL;

   GetThreadContext(hThread,&Context);
   Context.Dr0 = addr;
   Context.Dr7 |= DR7flag(OneByteLength,BreakOnExec,GlobalFlag | LocalFlag,0);
   SetThreadContext(hThread,&Context);
   RetnValue = WaitForSingleStepExc();
```

© Sysdream - 2008

```
    GetThreadContext(hThread,&Context);
    Context.Dr0 = 0;
    Context.Dr7 = 0;
    SetThreadContext(hThread,&Context);
    if (! RetnValue)
        return 0;
    else
        return 1;
}
```

<</LISTING>>

<<LISTING lang=C>>

```
// Listing 7. Rechercher une signature
DWORD SearchSign(char *Sign,DWORD SignSize,char *Mem ,DWORD MemSize)
{
    DWORD i,j,k;
    for (i=0;i<= MemSize; i++)
    {
        k = 0;
        for (j=0;j<SignSize;j++)
            if (Mem[i+j] != Sign[j])
                break;
            else k++;
        if (k==SignSize)
            return (DWORD)(Mem+i);
    }
    return 0;
}
```

<</LISTING>>

Pour *dumper* le programme en mémoire nous allons commencer par *dumper* ses *headers*, d'abord le dos *header*, puis le PE *header* et enfin les section headers, nous obtenons alors la taille du binaire en mémoire en additionnant les champs *VirtualAddress* et *VirtualSize* du dernier section *header*. Une fois que nous avons cette taille, nous pouvons *dumper* l'intégralité du binaire en mémoire.

© Sysdream - 2008

<<LISTING lang=C>>

```c
// Listing 8. Dumper le binaire en mémoire
PCHAR DumpProcess(HANDLE hProcess,char* BaseAddress)
{
   DWORD AllocSize = 0;
   char* Dump;
   IMAGE_DOS_HEADER DosHeader;
   IMAGE_NT_HEADERS PE;
   PIMAGE_SECTION_HEADER pSectionHeaders;
   DWORD i=0;
   DWORD SectionHeadersSize,NumberOfBytesRead;
   ReadProcessMemory(hProcess,BaseAddress,&DosHeader,sizeof(IMAGE_DOS_HEADER),&NumberOf
BytesRead);
   if (NumberOfBytesRead != sizeof(IMAGE_DOS_HEADER))
      return 0;
   ReadProcessMemory(hProcess,BaseAddress +
DosHeader.e_lfanew,&PE,sizeof(IMAGE_NT_HEADERS),&NumberOfBytesRead);
   if (NumberOfBytesRead != sizeof(IMAGE_NT_HEADERS))
      return 0;
   SectionHeadersSize = sizeof(IMAGE_NT_HEADERS)*PE.FileHeader.NumberOfSections;
   pSectionHeaders = (PIMAGE_SECTION_HEADER)malloc(SectionHeadersSize);
   ReadProcessMemory(hProcess,BaseAddress + DosHeader.e_lfanew + sizeof(IMAGE_FILE_HEADER)
+ PE.FileHeader.SizeOfOptionalHeader +
sizeof(DWORD),pSectionHeaders,SectionHeadersSize,&NumberOfBytesRead);
   if (NumberOfBytesRead != SectionHeadersSize)
   {
      free(pSectionHeaders);
      return 0;
   }
   while ((pSectionHeaders+i+1)->VirtualAddress != 0)
      i ++;
   AllocSize = (pSectionHeaders+i)->VirtualAddress + (pSectionHeaders+i)->Misc.VirtualSize;
   free(pSectionHeaders);
   Dump = (char *)VirtualAlloc(NULL,AllocSize,MEM_COMMIT |
MEM_RESERVE,PAGE_READWRITE);
   ReadProcessMemory(hProcess,BaseAddress,Dump,AllocSize,&NumberOfBytesRead);
   if (NumberOfBytesRead != AllocSize)
      return 0;
   return Dump;
}
```

<</LISTING>>

© Sysdream - 2008

Fixer le dump

Pour être totalement fonctionnel, notre *dump* doit être modifié, nous devons reconstruire l'IT qui a été détériorée lors du *packing* – les champs *FirstThunk* ont étés mis à 0 pour optimiser la compression –, il faut aussi réaligner les sections dans le fichier – les tailles des sections et des *headers* stockées sur le disque dur doivent être un multiple du champ *FileAlignment* du PE *Header* –, nous devons remplacer les champs *RawAddress* et *RawSize* dans les section *headers* par leurs nouvelles valeurs (forcément plus grandes étant donné que le binaire était compressé) enfin il faut remplacer la valeur de l'*entry point* par celle de l'*original entry point* et remplacer l'adresse de l'*Import Table* par celle obtenue lors de l'*un-packing*.

<<LISTING lang=C>>

```c
// Listing 9. Fixer le dump
DWORD RebuildDump(char* Dump)
{
   PIMAGE_DOS_HEADER pDosHeader;
   PIMAGE_NT_HEADERS pPE;
   PIMAGE_SECTION_HEADER pSection;
   PIMAGE_IMPORT_DESCRIPTOR IT;
   DWORD curseur,i;

   //il faut d'abord recopier la table des FirstChunk
   pDosHeader = (PIMAGE_DOS_HEADER)Dump;
   pPE = (PIMAGE_NT_HEADERS)(Dump+pDosHeader->e_lfanew);
   pSection = (PIMAGE_SECTION_HEADER)((PCHAR)pPE + sizeof(IMAGE_FILE_HEADER) + pPE->FileHeader.SizeOfOptionalHeader + sizeof(DWORD));
   for (IT = (PIMAGE_IMPORT_DESCRIPTOR)(pPE->OptionalHeader.DataDirectory[1].VirtualAddress + Dump);IT->Characteristics != 0;IT++)
      for (i=0;*(PDWORD)(IT->OriginalFirstThunk + Dump + i) != 0 ; i+=4)
         *(PDWORD)(IT->FirstThunk + Dump + i) = *(PDWORD)(IT->OriginalFirstThunk + Dump + i);

   curseur = pPE->OptionalHeader.SizeOfHeaders;
   //on copie maintenant les sections
   while (pSection->VirtualAddress != 0)
   {
      CopyMemory(Dump+curseur,Dump+pSection->VirtualAddress,pSection->Misc.VirtualSize);
      curseur += pSection->Misc.VirtualSize;
      while(Dump[curseur] == 0)
         curseur --;
      curseur = AlignSize(curseur,pPE->OptionalHeader.FileAlignment);
      pSection->SizeOfRawData = curseur - pSection->PointerToRawData;
      pSection ++;
   }
   return curseur;
}
```

<</LISTING>>

© Sysdream - 2008

Il ne nous reste plus qu'à recopier le *dump* dans un fichier pour avoir notre exécutable *unpacké*.

Conclusion

Le *packer* que nous avons étudié n'est pas très complexe, il n'y a pas de redirection d'API, pas de destruction de l'IT, pas de fonctions redirigées, pas d'*anti-debugger* néanmoins il permet de se familiariser avec les structures des *headers* Windows et il suffira d'ajouter des fonctions à notre *un-packer* pour les *packers* plus complexes. Quoiqu'il en soit, ce *packer* ne protège pas efficacement les *malwares* contre une détection, même par signature, le *packer* n'étant pas polymorphique il est facile d'extraire une signature du module développé par le pirate et donc de détecter tout les *malwares* qui l'utiliseront.

Sur Internet

- *http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx* – Documentation officielle du format PE,
- *http://membres.lycos.fr/w32assembly/ace/iczelion/_pe-tut.ace* – Traduction en français des tutoriaux de Iczelion sur le format PE par Morgatte,
- *http://msdn.microsoft.com/en-us/library/ms679303(VS.85).aspx* – Description des *debug* APIs,
- http://www.bitsum.com/pecompact.shtml – Site officiel du *packer* PECompact,
- *http://pdos.csail.mit.edu/6.828/2005/readings/i386/s12_02.htm* – Description des *Debug Registers*.

© Sysdream - 2008