# Ochrona programów dla Windows przed crackerami

Jakub Nowak

**Praca programisty zajmującego się tworzeniem aplikacji shareware jest prędzej czy później sabotowana przez crackerów. Często już w dniu publikacji programu w Internecie pojawia się crack lub keygen. Istnieją jednak metody umożliwiające ochronę kodu przed złodziejami.**

Autorzy płatnego oprogramowania często nie potrafią lub nie widzą sensu w zabezpieczaniu swoich dzieł przed złamaniem. Oczywiście nie istnieje idealne zabezpieczenie, które uniemożliwi crackerom stworzenie łaty lub generatora kluczy. Jeśli jednak postaramy się jak najbardziej utrudnić crackerowi jego złodziejskie poczynania, to być może zrezygnuje z naszego programu na rzecz słabiej zabezpieczonego. Zwróćmy więc uwagę na techniki, które sprawią, że nasza aplikacja nie będzie łatwą ofiarą.

## Wykrywanie SoftIce

Cracker właściwie nie może się obyć bez *debuggera*. Dzięki niemu może śledzić kod programu w asemblerze, instrukcja po instrukcji. Istnieje wiele programów o tej funkcjonalności, jednak w środowisku crackerskim największą popularnością cieszy się *SoftIce* firmy NuMega. Jest to debugger, który pracuje w środowisku uprzywilejowanym (*ring 0*).

Aby zabezpieczyć kod przed ewentualnym debugowaniem, można zastosować kilka sztuczek polegających na sprawdzeniu, czy *SoftIce* jest aktualnie zainstalowany w systemie oraz załadowany do pamięci. W razie jego wykrycia możemy zdecydować o zachowaniu naszego programu. Wszystkie przedstawione metody działają bez problemu na Windows 9x, jednak w przypadku innych wersji Windows (ME/NT/XP/2000) niektóre mogą nie spisywać się tak dobrze. Jest to spowodowane zwiększonym poziomem bezpieczeństwa w późniejszych wersjach Windows – niektórych trików nie da się już wykorzystać.

### Z artykułu dowiesz się...

- jak zabezpieczyć własny program przed scrackowaniem,
- jak wykryć obecność debugerów *SoftIce* i *OllyDbg*,
- w jaki sposób szyfrować komunikaty ekranowe,
- poznasz metody korzystania z *dummy opcodes*.

### Powinieneś wiedzieć...

- powinieneś znać Delphi,
- powinieneś znać asemblera,
- powinieneś umieć korzystać z debugerów w Windows.

**Listing 1.** *Wykrycie debuggera przez odnalezienie jego sterowników w pamięci*

```
if
CreateFileA('\\.\SICE', GENERIC_READ or GENERIC_WRITE,
  FILE_SHARE_READ or FILE_SHARE_WRITE, nil, OPEN_EXISTING,
  FILE_ATTRIBUTE_NORMAL, 0) <> INVALID_HANDLE_VALUE
then
  begin
    showmessage('SoftIce wykryty!');
  end;
```

**Listing 2.** *Otworzenie kluczy SoftIce w rejestrze za pomocą funkcji WinAPI RegOpenKeyEx*

```
if
RegOpenKeyEx(HKEY_LOCAL_MACHINE,
  'SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Uninstall\\SoftICE',
  0,KEY_READ,klucz) <> ERROR_SUCCESS
then
  begin
    showmessage('SoftIce wykryty!');
  end;
```

**Listing 3.** *Sprawdzenie wpisu SoftIce w pliku autoexec.bat*

```
var   f: textfile;
      s, pomoc: string;
      l: integer;

begin
assignfile(f, 'c:\autoexec.bat');
reset(f);
while not eof(f) do
  begin
    readln(f, s);
    for l:=1 to length(s) do
    s[l]:=Upcase(s[l]);
    pomoc:=s;
    if pomoc ='C:\PROGRA~1\NUMEGA\SOFTIC~1\WINICE.EXE' then
      begin
        showmessage('SoftIce wykryty!');
      end;
   end;
closefile(f)
end;
```

## W Sieci

- *http://www.pespin.w.interia.pl/* – program *PESpin*,
- *http://www.pelock.com/* – program *PELock*
- *http://www.sysinternals.com/* – programy *Filemon* oraz *Regmon*
- *http://home.t-online.de/home/Ollydbg/* – debuger *OllyDbg*
- http://www.aspack.com/asprotect.html – witryna programu *ASProtect*,
- *http://www.compuware.com/products/devpartner/bounds.htm* – projekt *Bounds Checker*,
- *http://www.siliconrealms.com/* – protektor *Armadillo*,
- *mailto: jakub-nowak@o2.pl* – kontakt z autorem.

Pierwsza i najbardziej popularna metoda odnalezienia *SoftIce* polega na wykryciu jego sterowników – plików *sice.vxd* oraz *ntice.vxd*. Spróbujmy je otworzyć wywołując funkcję *WinAPI* – `CreateFileA` (patrz Listing 1). Jeżeli *SoftIce* (a dokładnie *sice.vxd*) będzie w pamięci, system nie zezwoli na otworzenie sterownika – zostanie zwrócony jego uchwyt. W przeciwnym wypadku funkcja zwróci `INVALID_HANDLE_VALUE`, czyli brak sterownika. Analogicznie możemy postąpić ze sterownikiem *ntice.vxd*.

Inna metoda umożliwiająca wykrycie debugera polega na odnalezieniu jego kluczy w rejestrze systemowym Windows. *SoftIce*, jak każdy inny program, w momencie instalacji dokonuje zmian w rejestrze poprzez dodanie tam swoich wpisów. Modyfikowane klucze znajdują się w drzewie `HKEY_LOCAL_MACHINE`. Są to:

- `SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\SoftICE`,
- `SOFTWARE\NuMega\SoftICE`.

Wystarczy więc, przy użyciu funkcji `RegOpenKeyEx` (patrz Listing 2), otworzyć podany klucz w rejestrze – jeśli nie istnieje, funkcja zwróci wartość `ERROR_SUCCESS`. Tak samo należy postąpić z kluczem `SOFTWARE\NuMega\SoftICE`.

Kolejna, dość prosta technika wykrywania debuggera *SoftIce* polega na odnalezieniu wpisu w pliku *autoexec.bat*. Podczas instalacji *SoftIce* dodaje do tego pliku linijkę wskazującą ścieżkę do *winice.exe* (*C:\PROGRA~1\NUMEGA\SOFTIC~1\WINICE.EXE*), czyli pliku, który załaduje debuger do pamięci. Tę metodę przedstawiono na Listingu 3.

Debugger możemy wykryć stosując także nieco bardziej skomplikowane metody. Jedna z technik polega na wykorzystaniu wyjątków. Możemy je inicjalizować oraz wykorzystywać stosując funkcje API np. `SetUnhandledExceptionFilter` lub `UnhandledExceptionFilter`.

*SoftIce* przechwytuje wszystkie wywołania `INT 3`. Jeżeli rejestr `EBP` ma wartość `BCHK`, a rejestr `EAX` ma

**Listing 4.** *Wywołanie przerwania INT 3*

```
var
Zachowaj         :pointer;
begin
try
asm
  mov   Zachowaj,esp                ; zachowujemy wartość rejestru ESP
  push  offset @dalej               ; wsk. dla SetUnhandledExceptionFilter
  call  SetUnhandledExceptionFilter ; wywołanie wyjatku
  mov   ebp,'BCHK'                  ; załadowanie do EBP wartości 'BCHK'
  mov   eax, 4                      ; do rejestru EAX ładujemy wartość 4
  INT 3                             ; wywołujemy przerwanie INT 3
  Call  ExitProcess                 ; wychodzimy gdy wykryto SoftIce
@dalej:                             ; tu skaczemy gdy nie wykryto SoftIce
  mov   esp, Zachowaj               ; przywracamy oryginalną wartość ESP
  push  offset @koniec
  ret
@koniec:
  ret
end;
except end;
```

**Listing 5.** *Inna metoda wykrycia debugera z zastosowaniem przerwania INT 3*

```
var
Zachowaj         :pointer;
begin
try
asm
  mov   Zachowaj,esp                ; zachowujemy wartość rejestru ESP
  push  offset @dalej               ; wsk. dla SetUnhandledExceptionFilter
  call  SetUnhandledExceptionFilter ; wywołanie wyjatku
  mov   eax, 4                      ; do EAX ładujemy wartość 4
  mov   si, 'FG'                    ; do rejestru SI ładujemy wartość 'FG'
  mov   di, 'JM'                    ; do rejestru DI ładujemy wartość 'JM'
  INT 3                             ; wywołujemy przerwanie INT 3
  Call  ExitProcess                 ; wychodzimy gdy wykryto SoftIce
@dalej:                             ; tu skaczemy gdy nie wykryto SI
  mov   esp, Zachowaj               ; przywracamy oryginalną wartość ESP
  push  offset @koniec
  ret
@koniec:
  ret
end;
except end;
```

wartość 4, to debugger nie zezwala na przypisanie `ExceptionHandler` – zamiast tego zwraca w rejestrze `AL` wartość 0. Jeśli program nie jest uruchamiany za pośrednictwem *SoftIce*, do przechwytywania błędów i kontynuacji od wskazanego adresu stosowany jest `SetUnhandledExceptionFilter`. Jeśli korzystamy z *SoftIce* – nie będzie wywoływany, a to właśnie umożliwi nam wykrycie obecności debuggera.

Funkcję sprawdzającą najwygodniej napisać jako wstawkę asemblera w kodzie *Delphi* (Listing 4). Możemy również tę metodę nieco zmodyfikować, tak aby przerwanie `INT 3` było wywoływane nie z `EAX=4` i `EBP=BCHK`, lecz z rejestrami `SI=FG` oraz `DI=JM` – kod przedstawiono na Listingu 5.

Do wykrywania *SoftIce* można też wykorzystać sposób komunikacji systemu z debugerem. Wywołajmy przerwanie `INT 41` przy rejestrze `EAX=4Fh` (patrz Listing 6). Jeżeli *SoftIce* będzie obecny w systemie, wtedy przejmie on obsługę przerwania i wprowadzi do `EAX` wartość `0F386h`. Wartość ta jest identyfikatorem debuggera w systemie.

W podobny sposób wykryjemy *SoftIce* stosując przerwanie `INT 68h`. Należy je wywołać przy stanie rejestru `AH=43h`. Jeżeli debuggSer będzie w pamięci, to w rejestrze `EAX` zostanie zwrócona wartość `0F386h` (patrz Listing 7).

Jeszcze inna metoda wykrycia *SoftIce* polega na wykorzystaniu IDT (ang. *Interrupt Descriptor Table*). IDT jest tablicą, w której przechowywane są informacje o przerwaniach (patrz też Artykuł Mariusza Burdacha *Proste metody wykrywania debuggerów i środowiska VMware*, *hakin9* 1/2005). Wymaga tego sposób pracy systemu w trybie chronionym.

System Windows tworzy *Interrupt Descriptor Table* dla 255 wektorów przerwań. *SoftIce* wykorzystuje przerwania `INT 1` oraz `INT 3`. Idea tej metody polega na pobraniu z tablicy IDT adresów przerwań `INT 3` oraz `INT 1`, a następnie odjęciu ich wartości od siebie. `INT 3` ma wartość `3115h`, natomiast `INT 1` – `30F7h`. Po odjęciu ich od siebie otrzymamy wartość `1Eh` (patrz Listing 8).

## Co robić po wykryciu debuggera

W pierwszych przykładach pokazywaliśmy komunikat, który informował użytkownika o wykryciu *SoftIce*. W rzeczywistości powinniśmy się tego wystrzegać, gdyż jest to ułatwienie dla crackera próbującego złamać nasz program. Po otrzymaniu takiego komunikatu cracker może poszukać w kodzie stringu mówiącego o wykryciu *SoftIce*, a dzięki niemu szybko odnaleźć i unieszkodliwić zabezpieczenie.

Najlepiej nie informować w ogóle o wykryciu debugera, ale zastosować coś, co może zmylić złodzieja. Dobrą metodą jest załadowanie do odpowiedniej zmiennej wartości:

- 1 – gdy wykryto debuger,
- 0 – gdy debugera nie ma.

Następnie możemy podczas uruchomienia programu, na przykład przy wywołanu przycisku *Rejestra-*

*cja*, sprawdzić wartość tej zmiennej. Jeżeli jest równa 1, to aplikacja kończy działanie bądź przestaje reagować, jeżeli wartość jest równa 0 – program działa normalnie. Przykład znajduje się na Listingu 9.

Gdzie umieszczać nasze wykrywacze *SoftIce*? Najlepiej, by było ich jak najwięcej oraz aby były porozrzucane w różnych miejscach naszego kodu, a nie w jednej części obok siebie. Jeden może uruchamiać się przy starcie programu, inny może się chować pod przyciskiem rejestracji. Nagromadzenie tych funkcji jedna po drugiej tylko ułatwi crackerowi ich znalezienie i unieszkodliwienie.

## Wykrycie OllyDbg

Innym popularnym debuggerem jest *OllyDbg* (patrz Rysunek 1). Pracuje w środowisku okienkowym, więc możemy wykryć go przez napis na belce tytułowej okna. Jest to ciąg *OllyDbg*, a odnajdziemy go dzięki funkcji `FindWindowEx` (patrz Listing 10).

## Filemon i Regmon

Jeżeli w naszym programie wykorzystujemy plik rejestracyjny bądź zapisujemy w rejestrze Windows klucze mówiące o zarejestrowaniu programu, powinniśmy uważać również na programy takie jak *Filemon* oraz *Regmon*. Pierwszy rejestruje wszystkie otwierane pliki, drugi wszystkie wpisy w rejestrze.

Programy te możemy wykryć na dwa sposoby. Jeden to odnalezienie sterownika w pamięci (Listing 11), drugi to wykrycie okna (Listing 12).

W ten sam sposób możemy wykryć program *Regmon*. Wystarczy zamienić w listingach nazwę pliku sterownika na `\.\REGVXD` oraz nazwę okna na `Registry Monitor - Sysinternals: www.siliconrealms.com`.

## Szyfrowanie ciągów znakowych

Większość ważnych komunikatów w naszej aplikacji powinna być zaszyfrowana. Dzięki temu crackerowi będzie trudniej znaleźć punkt zaczepienia, gdyż zamiast na przykład *Nieprawidłowy numer seryjny* zobaczy w kodzie ciąg nic nie znaczących znaków, na przykład *Űüđĺçôâüń&úâěµűřřđçµćđçě˙űě*.

**Listing 6.** *Wykrycie debugera przez przerwanie INT 41h*

```
var
Zachowaj         :pointer;
begin
try
asm
  mov   Zachowaj,esp                ; zachowujemy wartość rejestru ESP
  push  offset @dalej               ; wsk. dla SetUnhandledExceptionFilter
  call  SetUnhandledExceptionFilter ; wywołanie wyjatku
  mov   eax, 4Fh                    ; do EAX ładujemy wartość 4fh
  int   41h                         ; wywołujemy przerwanie INT 41h
  cmp   eax, 0F386h                 ; jeśli równe, wykryto SoftIce
  jnz   @dalej                      ; jeśli nie, debuggera nie ma
  Call  ExitProcess                 ; wychodzimy gdy wykryto SoftIce
@dalej:                             ; tu skaczemy gdy nie wykryto SoftIce
  mov   esp, Zachowaj               ; przywracamy oryginalną wartość ESP
  push  offset @koniec
  ret
@koniec:
  ret
end;
except end;
```

**Listing 7.** *Wykrycie SoftIce za pomocą przerwania int68h*

```
asm
  mov   ah, 43h     ; załadowanie do rejestru AH wartości 43h
  int   68h         ; wywołanie przerwania int68h
  cmp   ax, 0F386h  ; porównanie zawartości rejestru AX z wartością 0F386h
  jnz   @dalej      ; jeśli nie zero (AX <> 0F386h) to nie wykryto SoftIce
  call  ExitProcess ; wyjście z programu
@dalej:             ; kontynuowanie programu
  ret
end;
```

Na Listingu 13 przedstawiamy mały program szyfrujący wybrany ciąg znakowy. Funkcja szyfrująca jest bardzo prosta, używa tylko instrukcji `xor`. Oczywiście można ją ulepszyć, jednak naszym głównym celem jest nieczytelność stringu, a dzięki właściwościom instrukcji `xor` nie będziemy musieli pisać funkcji odwracającej.

Aby wykorzystać funkcję szyfrującą, zastosujmy funkcję do zaszyfrowania napisu (na przykład *Nieprawidłowy numer seryjny*), a rezultat wykorzystajmy w programie (patrz Listing 14). W deasemblerze cracker zamiast stringu *Nieprawidłowy numer seryjny* zobaczy nierzucający się w oczy ciąg *Űüđĺçôâüń&úâěµűřřđçµćđçě˙űě*. W ten sposób powinniśmy szyfrować komunikaty związane z rejestracją lub ochroną programu. Pozostałych lepiej nie szyfrować – to mogłoby wzbudzić podejrzenia crackera.

## Dummy opcodes, czyli nieczytelność kodu

*Dummy opcodes* najprościej można zdefiniować jako bezsensowne, nic nie robiące instrukcje, które tylko zaśmiecają kod programu. Są one jednak doskonałą bronią w walce z crackerem. Jeżeli w naszym kodzie zastosujemy *junki* (bo tak również nazywane są te instrukcje) to podczas debugowania cracker zobaczy kod, który właściwie uniemożliwi mu dokładne zinterpretowanie istotnych instrukcji. Debugowany kod będzie tak bardzo zaśmiecony, że odnalezienie normalnych instrukcji będzie niezwykle trudne i uciążliwe.

Złamanie programu bez usunięcia *junków* jest prawdziwym wyzwaniem. Dzięki temu my zyskujemy na czasie,

**Listing 8.** *Wykrycie SoftIce za pomocą IDT*

```
var
IDT         : integer;
Zachowaj: pointer;
begin
try
asm
  mov   Zachowaj,esp                  ; zachowujemy wartość rejestru ESP
  push  offset @dalej                 ; wsk. dla SetUnhandledExceptionFilter
  call  SetUnhandledExceptionFilter   ; wywołanie wyjątku
  sidt  fword ptr IDT                 ; pobranie IDT
  mov   eax, dword ptr [IDT+2]        ; załadowanie do EAX
  add   eax,8
  mov   ebx, [eax]                    ; EBX = INT1
  add   eax, 16
  mov   eax, [eax]                    ; EAX = INT3
  and   eax, 0ffffh
  and   ebx, 0ffffh
  sub   eax, ebx                      ; odjęcie INT 1 od INT 3
  cmp   eax, 01eh                     ; jeżeli EAX = 01Eh to wykryto SoftIce
  jnz   @dalej                        ; jeżeli EAX <> 0 nie wykryto debuggera
  call  ExitProcess                   ; wyjście z programu
@dalej:                               ; tu skaczemy gdy nie wykryto SoftIce
  mov   esp, Zachowaj                 ; przywracamy oryginalną wartość ESP
  push  offset @koniec
  ret
@koniec:
  ret
end;
except end;
```

**Listing 9.** *Postępowanie w przypadku wykrycia debugera*

```
var zmienna: byte;
procedure sprawdz
begin
if
CreateFileA('\\.\SICE', GENERIC_READ or GENERIC_WRITE,
  FILE_SHARE_READ or FILE_SHARE_WRITE, nil, OPEN_EXISTING,
  FILE_ATTRIBUTE_NORMAL, 0) <> INVALID_HANDLE_VALUE
then
  begin
    zmienna:=1
  end;
end;
{.........tu dalsza część programu.....}
procedure TForm1.rejestracjaClick(Sender: TObject);
begin
     if zmienna=1 then
     ExitProcess(0);
{jeśli zmienna nie równa się 1 to możemy iść dalej}
end;
```

**Listing 10.** *Wykrycie debuggera OllyDbg*

```
if
  FindWindowEx(0,0,0, 'OllyDbg') <> 0
then
  begin
    ExitProcess(0);
  end;
```

a cracker traci na nerwach. Deasembler również źle radzi sobie z *junkami*. Kod jest tak samo nieczytelny. Aby zastosować w naszym programie *dummy opcodes*, należy użyć instrukcji asemblera. Oto przykład:

```
asm
db $EB, $02, $CD, $20
end;
```

Ten przykładowy *junk* był kiedyś stosowany w profesjonalnych *exe-protectorach* (programach zabezpieczających windowsowe pliki wykonywalne PE). W normalnej formie asemblera miałby postać:

```
jmp    $+4
int      20h
```

Instrukcje typu `jmp $(+/- liczba)` powodują przeskok o odpowiednią liczbę bajtów (w przód lub w tył, w zależności od znaku + lub -), przez co *opcody* są źle interpretowane, a kod zagmatwany. Proponujemy samemu zdebugować kod z takimi instrukcjami, aby zrozumieć jak to wygląda w praktyce.

Musimy jednak wiedzieć, gdzie i jak umieścić nasze *junki*. Przede wszystkim należy używać ich w znacznej ilości w kodzie sprawdzającym poprawność numeru seryjnego naszego programu. Poza tym można je umieszczać w komunikatach informujących na przykład o upływającym limicie używania programu, lub przy sprawdzaniu obecności debugera. Przykładowe użycie opcodów przedstawiono na Listingu 15.

Trzeba jednak przyznać, że używanie *junków* w taki sposób nie jest zbyt wygodne. Dlatego też możemy stworzyć plik, np. *dummy.jnk*, zadeklarować tam naszą wstawkę i potem dołączać przed każdą instrukcją nazwę pliku: `{$I dummy.jnk}`.

*Dummy opcodes* to naprawdę dobra metoda na crackera, więc nie powinniśmy ich żałować. Objętość kodu nie zwiększy się znacząco, a ochrona będzie skuteczna. Najlepiej łączyć ze sobą różne *junki* – stworzyć na przykład trzy różne i używać ich na zmianę lub jeden po drugim. Dwa inne przykładowe dummy opcodes:

```
db $EB, $02, $25, $02, $EB, $02, ←
  $17, $02,  $EB, $02, $AC, $F9, ←
  $EB, $02, $F1, $F8
db $E8,$01,$00,$00,$00,$33,$83,$C4,$04
```

Można samemu poeksperymentować wymyślając własne *junki*. Jednak trzeba uważać, gdyż nieumiejętne ich uformowanie może spowodować całkowite zawieszenie programu.

## Rady końcowe

Zabezpieczając swój program przed crackerem, za główne zadanie powinniśmy postawić sobie jego zmylenie. Możemy stosować różne sztuczki. Dobrą metodą jest wstawienie dodatkowego, fałszywego kodu, który byłby odpowiedzialny za rejestrację (oczywiście fałszywą).

Możemy na przykład zastosować długą funkcję sprawdzającą, która w razie złamania wyświetlałaby nieszyfrowany komunikat o poprawnej rejestracji. Cracker myślałby, że złamał program, podczas gdy my jedynie zmienilibyśmy w programie ciąg z `unregistered` na `registered to: xxx`, nie odblokowując jednocześnie funkcji niedostępnych w wersji próbnej.

Inna metoda to wykorzystanie zewnętrznego pliku. Jeżeli w katalogu z aplikacją nie ma odpowiedniego pliku, na przykład *register.dat*, to program skacze do fałszywej procedury sprawdzającej numer seryjny albo od razu zamyka okno rejestracji.

## Uciec przed czasem

Metody, które poznaliśmy pozwolą nam przedłużyć zabezpieczenie naszego programu – cracker będzie musiał spędzić przy nim na pewno trochę więcej czasu. Poza nimi możemy zastosować *exe-protectory*. Jest ich bardzo wiele, na przykład programy *ASProtect* czy *Armadillo*. Bardzo dobrym, darmowym polskim protectorem jest *PE-Spin* (patrz Ramka *W Sieci*). Program, w którym zastosujemy zarówno własne zabezpieczenia jak i *exe-protector*, na pewno ma duże szanse obrony przed potencjalnym crackerem. ■

**Listing 11.** *Wykrycie programu Filemon przez jego sterownik*

```
if
  CreateFileA(' \\.\FILEVXD', GENERIC_READ or GENERIC_WRITE,
  FILE_SHARE_READ or FILE_SHARE_WRITE, nil, OPEN_EXISTING,
  FILE_ATTRIBUTE_NORMAL, 0) <> INVALID_HANDLE_VALUE
then
  begin
    ExitProcess(0);
  end;
```

**Listing 12.** *Wykrycie okna programu Filemon*

```
if
  FindWindowEx(0,0,0, 'File Monitor - Sysinternals: www.sysinternals.com') <> 0
then
  begin
    ExitProcess(0);
  end;
```

**Listing 13.** *Funkcja szyfrująca i pokazująca dany string*

```
function szyfr(text:string):string;
var  t:integer; ch:char; by:byte; tmp:string; pokaz:string;
begin
  for t:=1 to length(text) do
    begin
      by:=ord(text[t]); by:=by xor $2F; by:=by xor $10;
      by:=by xor $AA; ch:=char(by); tmp:=tmp+ch;
    end;
pokaz:=tmp;
showmessage(pokaz);
```

**Listing 14.** *Używanie zaszyfrowanego stringu w komunikacie*

```
if
  Registration = 0
then
  begin
    szyfr('Űüdĺçôâüń&úâëµűřřđçµćđçë˙űë ');
  end;
```

**Listing 15.** *Stosowanie dummy opcodes w kodzie*

```
asm db $EB, $02, $CD, $20 end;
asm db $EB, $02, $CD, $20 end;
asm db $EB, $02, $CD, $20 end;

if Registered = 1
then
  begin
    MessageBox(0,PChar('Thank you for registration!'),
      PChar('Info'),MB_ICONINFORMATION);
  end;

asm db $EB, $02, $CD, $20 end;
asm db $EB, $02, $CD, $20 end;
asm db $EB, $02, $CD, $20 end;
```