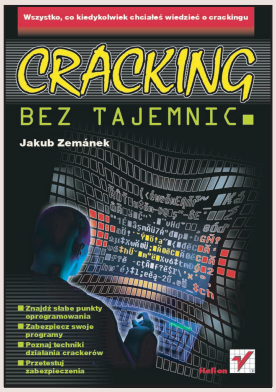
Cracking bez tajemnic



Autor: Jakub Zemanek
Tłumaczenie: Jerzy Kędziera
ISBN: 83-7361-444-3
Tytuł oryginału: Cracking bez tajemstvi
Format: B5, stron: 304

Zabezpieczanie programów i danych jest nierozerwalnie związane ze sposobami omijania lub łamania blokad. Nie można stworzyć skutecznych mechanizmów chroniących programy i dane przez nielegalnym kopiowaniem, nie wiedząc, jak działają ludzie, który te mechanizmy forsują. Oczywiście, niemożliwe jest wprowadzenie zabezpieczenia, które będzie niemożliwe do złamania lub obejścia. Cała sztuka polega na stworzeniu takich sposobów zablokowania dostępu do danych lub kodu programu, których złamanie będzie bardzo czasochłonne. Im dłużej program pozostanie niedostępny dla crackerów, tym dłużej użytkownicy będą kupować wersję legalną.

Książka „Cracking bez tajemnic" opisuje wszystko, co związane jest z crackingiem i tworzeniem zabezpieczeń — od podstawowych zagadnień do bardzo zaawansowanych technik. Zawiera informacje dotyczące prostych algorytmów zabezpieczeń, jak również sposoby tworzenia własnych szyfratorów. Jest przeznaczona zarówno dla programistów, którzy chcą się nauczyć zabezpieczania swoich programów przed crackingiem, ale również dla wszystkich, którzy zamierzają poznać techniki crackowania.

W kolejnych rozdziałach książki znajdziesz:

- Informacje o istniejących metodach zabezpieczania programów i danych
- Metody ochrony przed deasemblacją
- Opisy narzędzi używanych przez crackerów
- Sposoby tworzenia szyfratorów i deszyfratorów PE
- Metody crackowania programów
- Podstawowe informacje dotyczące asemblera

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Rozdział 3.

Ochrona przed deasemblacją

Pojęcie deasemblacji już objaśniałem — chodzi o przekształcenie programu do jego postaci statycznej w asemblerze. Takie postępowanie pomaga zrozumieć logikę niektórych algorytmów i uzyskać ogólną orientację w programie (zwłaszcza w kwestii wyszukiwania łańcuchów i importowanych funkcji oraz miejsc w programie, skąd są wywoływane i stosowane).

Właśnie dzięki możliwości wyszukiwania używanych przez program łańcuchów i importowanych funkcji deasemblery stały się bardzo popularne. Dzięki tym funkcjom cracker często może się dostać do algorytmu zabezpieczającego, bez przeszkód go przestudiować i następnie usunąć. To czyni deasemblery potężną bronią nawet w ręku żółtodzioba.

Używane deasemblery

Za standard w dziedzinie deasemblacji można uznać dwa programy — *W32Dasm* i *IDA*. Oczywiście istnieje jeszcze szereg innych (np. wyśmienity *OllyDbg*), ale te dwa są po prostu najpopularniejsze. Pierwszy z nich jest co prawda przestarzały, jego rozwój został już dawno wstrzymany i mogłoby się wydawać, że praktycznie jest już nie do wykorzystania, jednak tak nie jest. Crackerzy bowiem na tyle go sobie upodobali, że pomimo zakończenia jego rozwoju, kontynuują jego udoskonalanie i wydają najróżniejsze aktualizacje i poprawki. Jest to piękny przykład wykorzystania inżynierii wstecznej do udoskonalenia programu.

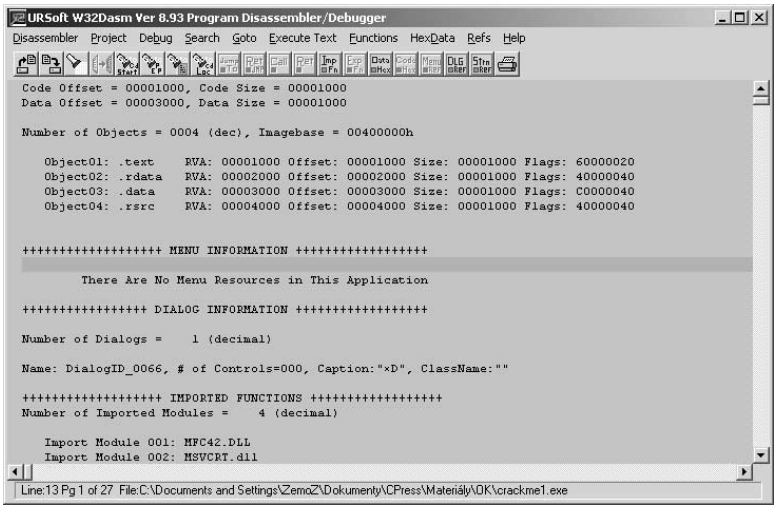
Najróżniejsze aktualizacje i poprawki znajdziesz na załączonej płycie CD.

W32Dasm jest przeznaczony dla wszystkich grup użytkowników. Nie mamy zbyt wielkiego wpływu na przebieg i ustawienia deasemblacji, co rekompensowane jest przez bardzo prostą i intuicyjną obsługę oraz przejrzystość kodu wynikowego. Program oferuje tworzenie odnośników do łańcuchów oraz funkcji importowanych, wyszukiwanie pełnotekstowe i wiele innych funkcji (po wprowadzeniu jednej z poprawek również np. zintegrowany edytor heksadecymalny), które w znacznym stopniu ułatwiają wyszukiwanie i orientację w programie. Obsługa tego narzędzia opisana jest w dalszej części niniejszego rozdziału.

IDA lub *Interactive Disassembler* jest, jak już sama nazwa podpowiada, w pełni konfigurowalnym profesjonalnym narzędziem do deasemblacji. To prawdziwy król deasemblerów, który dzięki swoim możliwościom i funkcjom nie ma konkurencji. Dzięki niesamowitej elastyczności tego programu (wliczając w to możliwość pisania skryptów w języku podobnym do C) można bez problemów poprawnie deasemblować również programy, na których W32Dasm połamie sobie zęby. Procesem deasemblacji można bowiem sterować ręcznie i w wielu aspektach go modyfikować. Te zalety odbijają się nieco na obsłudze, która na początku może się wydawać trochę nieprzejrzysta i skomplikowana. Program przeznaczony jest raczej dla doświadczonych użytkowników i nie jest odpowiedni dla początkujących.

Podstawy użytkowania W32Dasm

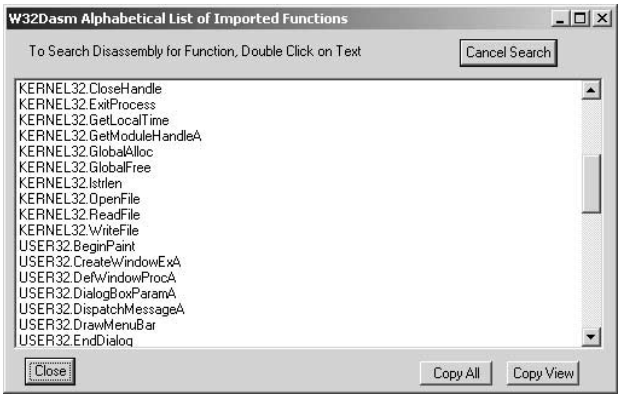
Środowisko pracy programu sprawia bardzo uporządkowane i przejrzyste wrażenie (rysunek 3.1). Najpierw wybierz plik, który chcesz deasemblować (kliknij na pozycję *Open File to Disassemble...* i wybierz plik). Po tym, jak plik zostanie zdeasemblowany (czas tego procesu zależy od wielkości pliku), uaktywnione zostaną kolejne funkcje programu.



Rysunek 3.1. *Środowisko pracy deasemblera W32Dasm*

Naciskając na klawisz *F10* lub pozycję menu *Goto Program Entry Point* przejdziesz do punktu wejścia kodu zdeasemblowanego (na ten temat więcej dowiesz się w części o formacie *PE*). Możesz przejść pod jakikolwiek inny adres naciskając *Shift+F12* lub przycisk *Goto Code Location*.

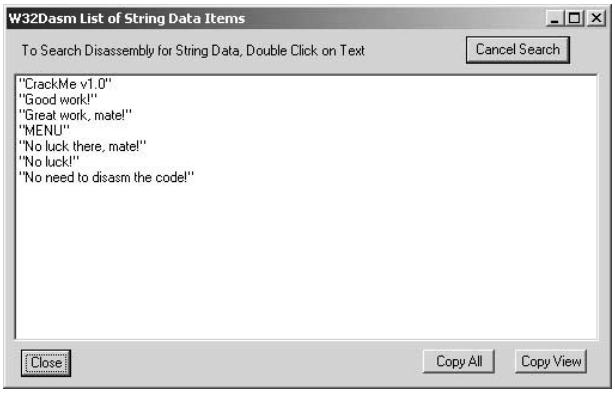
Spis importowanych funkcji wyświetlisz za pomocą przycisku *Imports* (rysunek 3.2), a eksportowanych *Exports*.



Rysunek 3.2. *Zorientowanie się w spisie funkcji importowanych nie jest problemem*

Jeżeli dwukrotnie klikniesz na wybraną funkcję API, przejdziesz do miejsca jej wywołania w kodzie programu. Jeżeli funkcja jest wywoływana wielokrotnie, w kolejne miejsce, gdzie jest wywoływana, przejdziesz powtarzając powyższą czynność.

Odnośniki do łańcuchów wyświetlisz używając przycisku *String Data References*. Często możesz znaleźć ciekawe łańcuchy, które doprowadzą cię w sam środek algorytmu zabezpieczeń (rysunek 3.3). Jeżeli spróbujesz zdeasemblować jakiś shareware'owy program, znajdziesz tam najpewniej komunikat o poprawnie i (lub) niepoprawnie podanych informacjach rejestracyjnych itd.



Rysunek 3.3. *Odnośniki do łańcuchów mogą wiele podpowiedzieć*

Poruszanie się po znalezionych łańcuchach odbywa się tak samo, jak po spisie importowanych funkcji — wystarczy kliknąć dwa razy znaleziony łańcuch aby przejść do kodu programu.

Praca z samym zdeasemblowanym kodem programu i orientacja w nim jest bardzo łatwa. Odnośniki do skoków i instrukcji `CALL` są przejrzyście wkomponowane w kod, tak więc nie ma problemu nawet w przypadku bardziej złożonych algorytmów. Jasnoniebieski pasek oznacza instrukcję, którą właśnie się zajmujesz.

Skoki warunkowe i bezwarunkowe można śledzić naciskając klawisz strzałki w prawo (lub przycisk *Jump To*), tak samo jak wejścia do instrukcji `CALL` (można je wykonać również przyciskiem *Call*). Wrócić do instrukcji `CALL` można naciskając klawisz strzałki w lewo (lub przycisk *Ret*), zaś w przypadku skoków przy pomocy kombinacji klawiszy *Ctrl+<strzałka w lewo>* (lub przycisk *Ret JMP*).

W przypadku większych plików, z którymi często się pracuje a proces deasemblacji trwa dłużej, wygodnie jest zapisać zdeasemblowany kod korzystając z pozycji o nazwie *Save Disassembly File and Create Project File* w menu *Disassembler*. Wczytać plik można następnie z menu *Project*, pozycja *Open Project File*.

To było wyliczenie podstawowych funkcji deasemblera *W32Dasm*. Obsługa jest prosta i bardzo intuicyjna, więc resztę funkcji bardzo szybko przyswoisz sam. Możliwością wykorzystania tego programu również jako debugera nie będziemy się zajmować. Daleko odbiega od jakości SoftICE.

Stosowane algorytmy

Podobnie jak przy antydebugingu, tak w przypadku antydeasemblacji można się zabezpieczać przed specyficznymi funkcjami deasemblera lub przed samą deasemblacją. Jedyną różnicą jest to, że kiedy w antydebugingu program bronił się przed debugingiem aktywnie poprzez detekcję działającego debugera (w książce jednak przedstawię również inne metody stosowane w algorytmach ochrony przed debugingiem, które utrudniają sam debuging — np. w tym rozdziale), w przypadku ochrony przed deasemblacją tak uczynić nie można. Program bowiem, tak samo jak przy edycji kodu programu w edytorze heksadecymalnym, nie jest uruchomiony, więc może się bronić wyłącznie pasywnie. Detekcja aktywna jest co prawda możliwa (np. za pomocą wyszukiwania nazw uruchomionych procesów, nazw okien itd.), ale byłaby zupełnie do niczego (nie bacząc na fakt, że w odróżnieniu od programu SoftICE, deasemblerów i edytorów heksadecymalnych jest całe mnóstwo).

Przed kilku laty, kiedy jeszcze nie było deasemblerów tak rozwiniętych jak dzisiaj, można było zapobiec deasemblacji w prosty sposób. Do kodu programu dodawano algorytmy oraz ciągi instrukcji, których kompilator nigdy by nie wygenerował. Dzięki temu przy próbie dekompilacji (w tym przypadku mówiąc ściślej deasemblacji) dochodziło do błędu lub zapętlenia dekompilatora; na przykład następujące ciągi „mylących” instrukcji były w przeszłości powszechnie stosowane:

```
mov eax,123456h
push eax
ret
```

Ten kod jest w rzeczywistości ekwiwalentem instrukcji `JMP EAX` i ciągle można go pod różnymi postaciami znaleźć w szeregu mechanizmów zabezpieczających.

W dzisiejszych czasach deasemblery są już na coś podobnego przygotowane, tak więc stosowanie tej metody nie zda egzaminu. Nasze starania dlatego skierowane będą w pierwszym rzędzie na to, aby deasemblacja pliku była całkowicie bezprzedmiotowa. Plik będzie można zdeasemblować, ale kod wynikowy będzie bądź bezsensowny, bądź tak nieprzejrzysty, że nieużyteczny do jakiejkolwiek analizy. Tym zajmiemy się jednak za chwilę. Teraz pokażę, jak zabezpieczyć program przed najniebezpieczniejszymi funkcjami deasemblera, którymi są odnośniki do łańcuchów i importowanych funkcji.

Algorytmy podstawowe

Ochrona łańcuchów

Zapobieganie przeszukiwaniu stosowanych przez program łańcuchów jest całkiem łatwe: można je zaszyfrować (zamiast poprawnych łańcuchów pojawią się wówczas wyłącznie bezsensowne znaki, które crackerowi nie ułatwią zadania), zmienić lub zmodyfikować w inny sposób, a najlepiej zaprzestać (jeżeli jest to możliwe) ich stosowania. Sposobów na to, jak zapisać jakiś tekst do pliku, jest przecież bez liku.

Ochrona importowanych funkcji

Deasembler przy generowaniu listy importowanych funkcji wskazujących na poszczególne miejsca w programie korzysta ze specjalnej tabeli, tzw. tabeli importów. O jej dokładnej strukturze, funkcji i zabezpieczeniu dowiesz się więcej z rozdziału o formacie *PE*, z którego zastosowania wywodzi się większość metod ochrony funkcji importowanych. Teraz przedstawię tylko tę najprostszą technikę bazującą na pośrednim wywoływaniu funkcji tak, aby sam kompilator nie umieszczał ich w tabeli importów). W taki sposób wywoływane funkcje nie zostaną wpisane do tabeli importów i deasembler nie zaszereguje ich do swojego spisu.

Ta technika jest wyśmienita w szczególności do wykorzystania z funkcjami API, gdzie biblioteki, z której chcemy w taki sposób „importować", nie trzeba już wgrywać do pamięci.

Najpierw sprawdzimy bazę obrazu (`ImageBase` — patrz rozdział o formacie *PE*) wczytanej biblioteki, gdzie umieszczona jest wymagana funkcja, a następnie określimy jej adres, który następnie już bezpośrednio wywołamy.

Poniższy przykład ustali adres funkcji `CreateFileA` w bibliotece *kernel32.dll*. Jest to naprawdę łatwe:

```
HMODULE Handle = GetModuleHandle("kernel32");  // kernel32 ImageBase
FARPROC Create = GetProcAddress(Handle,"CreateFileA");  // adres funkcji
CreateFileA
```

Teraz już tylko wystarczy funkcję wywołać — np. następująco:

```
__asm
{
    push 0
    push FILE_ATTRIBUTE_NORMAL
    push OPEN_EXISTING
    push 0
    push FILE_SHARE_READ
    push GENERIC_READ
    push wskaźnik_do_nazwy_pliku
    call Create  //CALL CreateFileA
    mov uchwyt_pliku,eax
}
// HANDLE File = CreateFileA("nazwa pliku",GENERIC_READ,FILE_SHARE_READ,NULL,
OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,NULL);
```

Ta technika sama w sobie jest bardzo podatna na atak, ponieważ można za pomocą spisu importowanych funkcji znaleźć i wykorzystać kombinację bardzo dobrze znanych funkcji `GetModuleHandle` i `GetProcAddress`, a ponadto w liście łańcuchów można znaleźć również nazwy bibliotek oraz funkcji, co akurat rzuca się w oczy. Zalecam łączenie tej techniki z najróżniejszymi innymi technikami ochrony importowanych funkcji i łańcuchów.

Możesz się też spotkać z algorytmem, gdzie adres importowanej funkcji określony w jednym miejscu (chociażby z wykorzystaniem tabeli importów lub bez tego) zostaje zachowany, a następnie wywoływany w zupełnie innym miejscu. To tylko jeden z wielu sposobów zapobiegania wyświetleniu przez deasembler importowanej funkcji w miejscu, gdzie naprawdę najmniej tego potrzebujesz.

Praktyczny przykład wykorzystania powyższych metod znajdziesz w jednym z moich przykładów *crackme* w rozdziale dziewiątym.

SMC — kod samomodyfikujący się

Do najczęściej stosowanych sposobów zabezpieczenia oprogramowania, i to nie tylko przed deasemblacją, zaliczany jest *SMC* (ang. *Self Modyfying Code*). Możliwe, że zamieszczenie tej techniki w rozdziale o antydeasemblacji nie będzie się wydawało poprawne, ale miej na względzie to, że liczne techniki antycrackingowe nawzajem się pokrywają i uzupełniają, nie można więc dokładnie ich rozdzielić. Bardziej niż oczywiste jest, że jakaś technika antydeasemblacji „pracuje" również np. jako kod antydebugingowy i *vice versa*. W przypadku SMC obowiązuje to w dwójnasób: utrudnia lub wręcz uniemożliwia edycję kodu źródłowego, debuging i deasemblację.

Za SMC można uznać dwa rodzaje kodu: pierwszy wykorzystuje w pełni dynamiczne generowanie (edycję) kodu programu bezpośrednio podczas pracy programu. Drugi typ wykorzystuje wielopostaciowość zapisu kodu w asemblerze. W celu usystematyzowania, pierwszą z wymienianych możliwości pozwolę sobie nazwać aktywnym, a drugą pasywnym SMC. W żadnym jednak przypadku nie są to oficjalne nazwy.

Pasywny SMC

Spójrz na następujący przykład:

```
:00401000 EB01  jmp 00401003
:00401002 E86641  call  ;parametry instrukcji nie są ważne
```

Zapewne zauważyłeś coś bardzo dziwnego — pierwsza instrukcja `JMP` skacze gdzieś w środek następnej instrukcji `CALL`!? W ten sposób zostaje „przeskoczony" jeden bajt pod adresem `00401002`. Po skoku więc kod w rzeczywistości wygląda następująco:

```
:00401000 EB01  jmp 00401003
:00401002 E8    ;ta niewłaściwa instrukcja zostanie przeskoczona
:00401003 6641  inc eax
```

Na tym prostym przykładzie zrozumiałeś zapewne, o co chodzi w pasywnym SMC i jak ta technika funkcjonuje. Stosując taki sposób zapisu można osiągnąć doskonałe wyniki. Kod staje się bardzo nieprzejrzysty, co w niewyobrażalny wręcz sposób utrudnia nie tylko debugowanie i deasemblację, ale również samą edycję kodu programu. Zrozumienie struktury takiego kodu jest bardzo trudne.

Obawiam się, że nie istnieją chyba ogólne wskazówki tworzenia algorytmów SMC. Zawsze wymaga to określonej ilości czasu, cierpliwości i doświadczenia.

Niech poniższy przykład kodu SMC posłuży za inspirację:

```
:00401000 EB01 jmps 000401003
:00401002 2853BB sub [ebx][-0045],dl
:00401005 7856 js 00040105D
:00401007 3412 xor al,012
:00401009 EB15 jmps 000401020
:0040100B 2881F3214365     sub [ecx][0654321F3],al
:00401011 87EB xchg ebp.ebx
:00401013 02E8 add ch.al
:00401015 6981F35815519590EB04 i mu l eax.d,[ecx][0511558F3],004EB9
:0040101F C2EBEA     retn OEAEB
:00401022 B48B mov ah,08B
:00401024 C3 retn
:00401025 5B pop ebx
```

Czy powiedziałbyś, że jedyne, co ten kod realizuje, to wpisanie wartości 1 do rejestru *EAX*? Zapewne nie, a więc przyjrzyjmy się kolejnym zmianom kodu przy jego realizacji.

Po pierwszym skoku z adresu `00401000` kod zmieni się do następującej postaci:

```
:00401003 push ebx
:00401004 mov ebx,12345678
:00401009 jmp 00401020
:0040100B sub [ecx+654321F3],al
:00401011 xchg ebp, ebx
:00401013 add ch,al
:00401015 imul eax,[ecx+511558F3].04EB9095
:0040101F ret eaeb
:00401022 mov ah,8B
:00401024 ret
:00401025 pop ebx
```

Wykonane zostaną instrukcje pod adresami 00401003 i 00401004 i dojdzie do skoku pod adres 00401020. Po skoku kod wygląda następująco:

```
:00401020 jmp 0040100C
:00401022 mov ah,8B
:00401024 ret
:00401025 pop ebx
```

Wykonany zostanie kolejny skok z powrotem pod adres 0040100C:

```
:0040100C xor ebx,87654321
:00401012 jmp 00401016
:00401014 call 59339182
:00401019 adc eax,EB909551
:0040101E add al,c2
:00401020 jmp 0040100C
:00401022 mov ah,8B
:00401024 ret
:00401025 pop ebx
```

Po wykonaniu instrukcji pod adresem 0040100C dochodzi do kolejnego skoku tym razem pod adres 00401016:

```
:00401016 xor ebx,95511558
:0040101C nop
:0040101D jmp 00401023
:0040101F ret eaeb
:00401022 mov ah.8B
:00401024 ret
:00401025 pop ebx
```

Teraz wreszcie dochodzi do ostatniego skoku pod adresem 0040101D:

```
:00401023 mov eax, ebx
:00401025 pop ebx
```

Jeżeli przepiszemy wszystkie „prawdziwe" instrukcje, otrzymamy następujący kod, z którego już bez problemu zrozumiesz, jak algorytm działa:

```
push ebx   // zachowa EBX
mov ebx,12345678h   // EBX = 12345678h
xor ebx,87654321h   // EBX = 95511559h
xor ebx,95511558h   // EBX = 1
mov eax,ebx   // EAX = EBX = 1
pop ebx   // przywróci EBX
```

Cały powyższy kod jest ekwiwalentem instrukcji MOV EAX,1.

Dzięki zastosowaniu SMC i wybraniu większej liczby instrukcji do wykonania danego zadania (zamiast zwykłego MOV EAX,1) tak prymitywne zadanie, jakim jest wpisanie wartości 1 do rejestru *EAX*, staje się bardzo nieprzejrzystą i skomplikowaną operacją. Zrozumienie struktury takiego kodu jest bardzo trudne. W przypadku dłuższego algorytmu bez debugera wykonanie tego zadania jest wręcz niemożliwe.

Ten przykład miał zilustrować nie tylko implementację SMC w praktyce, ale również fakt, że czasami nie jest złym pomysłem napisanie nawet prostej rzeczy w sposób trochę bardziej złożony. Wynik i powodzenie obu tych technik możesz sam ocenić.

Aktywny SMC

Jednym z najlepszych sposobów ochrony kodu programu jest jego dynamiczne generowanie. Metodę tę już zalecałem na początku pierwszego rozdziału, jednak dokładniej zdefiniowana i opisana jest dopiero tutaj. Moim staraniem było zgrupowanie implementacji SMC w jednym rozdziale — chociaż możliwe, że nie jest to najlepsze miejsce.

Nie chodzi w zasadzie o nic innego niż edycję kodu programu bezpośrednio w trakcie pracy programu. Tak jak wpisujesz dane do zmiennych, rejestrów, itp., możesz spokojnie modyfikować również kod programu. Jedynym problemem jest to, że obszar w pamięci, do którego wpisujemy, musi mieć charakterystykę `writeable`, a więc musi w nim istnieć możliwość zapisu, w przeciwnym razie w programie zaistnieje wyjątek (`STATUS_ACCESS_VIOLATION`). Ten temat jest ściśle związany z formatem *PE*, więc dowiesz się więcej dopiero w poświęconym temu zagadnieniu rozdziale. Tymczasem wystarczy zastosowanie funkcji `VirtualProtect`, która z odpowiednimi parametrami umożliwi nam zapisywanie w uruchamialnym kodzie programu.

Aby zademonstrować prawdziwą moc i możliwości SMC, wszystko pokażę na naprawdę prostym przykładzie. Przyjrzyj się następującemu kodowi — zapewne natychmiast będzie dla ciebie oczywistym efekt jego działania:

```
BOOL A = TRUE;
if (A)
    MessageBox("Kod programu nie został zmodyfikowany",NULL,MB_OK);
else
    MessageBox("Kod programu został z powodzeniem zmodyfikowany",NULL,MB_OK);
```

Konstrukcja z `else` jest logicznie całkowicie zbędna, ponieważ zawsze dojdzie do wyświetlenia komunikatu *Kod programu nie został zmodyfikowany*. W asemblerze kod zapiszemy mniej więcej tak:

```
__asm
{
    mov eax,1
    cmp eax,1
    jnz SMC_
}
    MessageBox("Kod programu nie został zmodyfikowany",NULL,MB_OK);
    return;
SMC_:
    MessageBox("Kod programu został z powodzeniem zmodyfikowany",NULL,MB_OK);
```

Teraz do programu wstawimy wywołanie funkcji `VirtualPortect` i spróbujemy przepisać instrukcję `JNZ` przez instrukcję logicznie odwrotną `JZ`. Tym sposobem odwrócimy logikę całego algorytmu i powinno pojawić się okienko komunikatu `MessageBox` z tekstem *Kod programu został z powodzeniem zmodyfikowany*. Kod wygląda następująco:

```
DWORD Adres,LastProtect;
__asm
{
    mov Adres,offset Przepisz
}
```

```
VirtualProtect ((void*)Adres,10,PAGE_READWRITE,&LastProtect);
__asm
{
    mov ebx,Adres
    mov Byte ptr [ebx],74h   // przepisanie wartości 74h na 75h, tzn. instrukcji
JNZ na JZ
    mov eax,1
    cmp eax,1
}
Przepisz:
        jnz SMC_
}
    MessageBox("Kod programu nie został zmodyfikowany",NULL,MB_OK);
    return;
SMC_:
    MessageBox("Kod programu został z powodzeniem zmodyfikowany",NULL,MB_OK);
```

Wynikiem działania kodu po tej modyfikacji będzie komunikat *Kod programu został z powodzeniem zmodyfikowany*.

Jak widzisz, naprawdę nie chodzi o żadną naukę. Wystarczy drobna modyfikacja w postaci wywołania funkcji API `VirtualProtect` poszerzająca prawa dostępu o możliwość zapisu (`PAGE_READWRITE` — odczyt i zapis) w określonym obszarze pamięci; w naszym przypadku w obszarze, gdzie wczytany jest uruchamialny kod programu.

Jak mówiłem, to naprawdę prymitywny przykład, ale wykorzystanie i potencjał SMC ilustruje wyśmienicie. Jak zawsze zależy tylko od ciebie, jak tę technikę dopracujesz i ile czasu poświęcisz na projekt kodu programu.

Edycja kodu programu w trakcie pracy programu

Najróżniejsze metody zabezpieczania programów przed crackingiem wykorzystują podstawową ideę aktywnego SMC — więc edycja kodu programu w trakcie jego działania należy bezspornie do tych najlepszych i również najczęściej stosowanych. Możliwości są naprawdę nieograniczone — od szyfrowania kodu programu, zmian poszczególnych instrukcji, ich celowego niszczenia, a z drugiej strony aż po wstawianie i przesuwanie wielkich bloków danych, przepisywania całych funkcji itd. Zależy to po prostu wyłącznie od twojej fantazji.

Na kilka możliwości wykorzystania edycji kodu programu w trakcie jego pracy już kilkukrotnie zwracałem uwagę w wielu miejscach tej książki i jeszcze się z nimi spotkasz. Ta część rozdziału miała tylko pokazać, jakie to proste. W rozdziale siódmym omówimy szyfrowanie i deszyfrowanie danych w trakcie pracy programu z wykorzystaniem formatu *PE*.