

IDŹ DO

PRZYKŁADOWY ROZDZIAŁ



SPIS TREŚCI

KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE

ZAMÓW INFORMACJE
O NOWOŚCIACH

ZAMÓW CENNIK

CZYTELNIA

FRAGMENTY KSIĄŻEK ONLINE

Cracking bez tajemnic

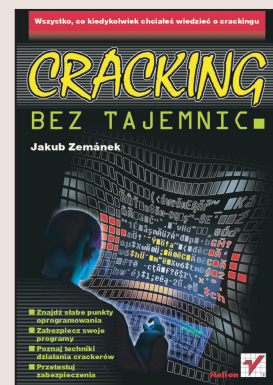
Autor: Jakub Zemanek

Tłumaczenie: Jerzy Kędziera

ISBN: 83-7361-444-3

Tytuł oryginału: [Cracking bez tajemstvi](#)

Format: B5, stron: 304



Zabezpieczanie programów i danych jest nierozdzielnie związane ze sposobami omijania lub łamania blokad. Nie można stworzyć skutecznych mechanizmów chroniących programy i dane przed nielegalnym kopiowaniem, nie wiedząc, jak działają ludzie, którzy 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 crackowaniem 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 crackowaniem, 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 assemblera



Spis treści

Przedmowa	9
Wstęp	11
Czym jest cracking.....	12
Ogólne zasady ochrony przed crackiem.....	13
Zasady Marka.....	13
Kolejne zasady.....	14
Podstawowe programy używane do crackingu.....	15
Czy w ogóle zabezpieczanie oprogramowania ma sens („I tak ktoś to scrackuje”).....	16
Zawartość płyty CD.....	16
Rozdział 1. Istniejące sposoby zabezpieczeń i ich podatność na ataki	17
Szyfrowanie danych.....	17
Programy niepełne.....	18
Podstawowa klasyfikacja dzisiejszych typów zabezpieczeń.....	18
Ograniczenia czasowe.....	19
Kolejne ograniczenia ilościowe.....	22
Numer seryjny.....	22
Plik klucza.....	27
Programy z ograniczoną funkcjonalnością.....	32
Klucz sprzętowy.....	34
Kontrola obecności płyty CD.....	36
Zabezpieczenia przed kopiowaniem płyt CD z danymi.....	39
Fizyczne błędy nośnika CD.....	39
Błędne pliki (Dummy Files).....	39
Płyty CD o pojemności większej niż 74 minuty (Oversized CD).....	40
Błędna wartość TOC (Illegal Table of Contents).....	40
Duże pliki.....	40
Programowe (fikcyjne) błędy i inne ingerencje w proces produkcji płyt CD.....	41
Zabezpieczenia komercyjne.....	41
SafeDisc.....	42
SecuROM.....	43
ProtectCD.....	43
Armadillo (The Armadillo Software Protection System).....	44
ASProtect.....	45
SalesAgent.....	46

VBox	46
Programy napisane w Visual Basicu	46
Porównanie łańcuchów	47
Porównanie zmiennych (Variant data type).....	48
Porównanie zmiennych (Long data type)	48
Przekształcanie typów danych	48
Przemieszczanie danych	49
Matematyka.....	49
Różne	49
Kolejne poważne błędy obecnych zabezpieczeń	49
Rozdział 2. Ochrona przed debugingiem	55
Stosowane debugery	56
Podstawy używania programu SoftICE	56
Konfiguracja programu	56
Podstawowe polecenia, funkcje i obsługa	58
Okna.....	58
Punkty wstrzymania	60
Praca z punktami wstrzymania	63
SEH — Strukturalna obsługa wyjątków	64
Czym jest i do czego służy SEH.....	64
Konstrukcje korzystające z SEH.....	65
Używane algorytmy	66
Algorytmy wykorzystujące funkcję API CreateFileA.....	66
Interfejs BoundsChecker i wykorzystanie przerwania INT 3.....	67
Wykorzystanie przerwania INT 1.....	69
Wykorzystanie przerwania INT 68h.....	71
Wyszukiwanie wartości w rejestrach	72
Wyszukiwanie wartości w pliku autoexec.bat.....	72
Punkty wstrzymania	73
Programowe punkty wstrzymania.....	74
Punkt wstrzymania na przerwanie (BPINT).....	74
Punkt wstrzymania na wykonanie (BPX).....	75
Punkt wstrzymania na dostęp do zakresu pamięci (BPR)	75
Sprzętowe punkty wstrzymania	76
Opis przykładowego programu do detekcji sprzętowych punktów wstrzymania ..	78
Metody zaawansowane	80
Tryby Ring	80
Sposoby przejścia pomiędzy Ring3 i Ring0.....	81
Wykrycie programu SoftICE za pomocą VxDCall	86
Dezaktywacja skrótu klawiaturowego programu SoftICE	88
Kolejne proste możliwości i wykorzystanie SEH	90
Rozdział 3. Ochrona przed deasemblacją	93
Używane deasemblery.....	93
Podstawy użytkowania W32Dasm.....	94
Stosowane algorytmy	96
Algorytmy podstawowe	97
Ochrona łańcuchów	97
Ochrona importowanych funkcji	97
SMC — kod samomodyfikujący się	98
Pasywny SMC	99
Aktywny SMC	101
Edycja kodu programu w trakcie pracy programu.....	102

Rozdział 4. Program FrogsICE i obrona przed nim	103
Podstawy użytkowania programu FrogsICE.....	103
Basic options.....	103
Advanced options	104
Stosowane algorytmy	105
VxDCall funkcji VMM_GetDDBList	105
Wykorzystanie funkcji CreateFileA	107
Rozdział 5. Program ProcDump i obrona przed nim	109
Podstawy użytkowania programu ProcDump	109
Co to jest dumping i do czego służy	112
Używane algorytmy	112
Rozdział 6. Edycja kodu programu.....	115
Metody stosowane przy edycji kodu programu	115
Podstawy użytkowania programu Hiew.....	116
Edycja programu do detekcji SoftICE	117
Stosowane algorytmy	119
Kontrola spójności danych.....	119
Kontrola spójności danych w pliku	120
Kontrola spójności danych w pamięci.....	122
Inne sposoby	126
Rozdział 7. Szyfratory i kompresory PE oraz format PE.....	127
Co to jest format PE pliku?	127
Czym jest szyfrator (kompresor) PE i jak działa.....	128
Jak stworzyć szyfrator (kompresor) PE.....	129
Wady szyfratorów (kompresorów) PE	130
Stosowane szyfratory (kompresory) PE.....	130
ASPack	130
CodeSafe.....	131
NeoLite	131
NFO	131
PE-Compact.....	132
PE-Crypt	132
PE-Shield	133
Petite	133
Shrinker	134
UPX	134
WWPack32.....	135
Format plików PE.....	135
Weryfikacja formatu PE	135
Nagłówek PE	138
Tabela sekcji	140
Nie wiesz, co oznaczają słowa Virtual, Raw i RVA?	141
Tabela importów	142
Tabela eksportów	145
Tworzymy szyfrator PE	146
Dodanie nowej sekcji do pliku.....	147
Przekierowanie strumienia danych	150
Dodanie kodu do nowej sekcji.....	151
Skok z powrotem i zmienne.....	152
Funkcje importowane.....	156
Utworzenie tabeli importów	157
Przetworzenie oryginalnej tabeli importów	160
Użycie importowanej funkcji	164

Obróbka TLS	165
Szyfrowanie	167
Jaki wybrać algorytm szyfrowania	167
Znane algorytmy szyfrowania	167
Co się stanie, gdy ktoś złamie szyfr	169
Co szyfrować, a czego nie	170
Demonstracja prostego szyfrowania w szyfratorze PE	171
Finalna postać stworzonego szyfratora PE	175
Dodatkowe możliwości zabezpieczeń	192
Anti-SoftICE Symbol Loader	192
Kontrola Program Entry Point	193
RSA	193
Przykład zastosowania RSA	196
Podsumowanie szyfratorów i formatu PE	197
Rozdział 8. Kolejne programy stosowane przez crackerów	199
Registry Monitor	199
File Monitor	201
R!SC'S Process Patcher	202
Polecenia w skryptach	203
The Customiser	204
Rozdział 9. Crackujemy	207
Cruehead — CrackMe v1.0	207
Cruehead — CrackMe v2.0	210
Cruehead — CrackMe v3.0	211
CoSH — Crackme1	214
Mexelite — Crackme 4.0	215
Immortal Descendants — Crackme 8	216
Easy Serial	217
Harder Serial	218
Name/Serial	218
Matrix	219
KeyFile	219
NAG	220
Cripple	220
Duelist — Crackme #5	220
Ręczne deszyfrowanie pliku	221
Zmiany bezpośrednio w pamięci	224
tC — CrackMe 9 <id:6>	225
Uzyskanie poprawnego numeru seryjnego	225
Zamiana programu na generator numerów seryjnych	227
tC — CrackMe 10 <id:7>	228
tC — Crackme 13 <id:10>	229
tC — Crackme 20 <id:17>	231
ZemoZ — Matrix CrackMe	234
ZemoZ — CRCMe	237
Edycja programu w edytorze heksadecymalnym	239
Wykorzystanie programu ładującego	242
Rozdział 10. Kolejne informacje o crackingu	245
Wielki wybuch, czyli jak się to wszystko zaczęło	245
Kto poświęca się crackingowi	246
Znani crackerzy i grupy crackerskie	246
+HCU	247
Immortal Descendants	247

Messing in Bytes — MiB	248
Crackers in Action — CiA	248
Phrozen Crew	248
United Cracking Force — UCF	249
Ebola Virus Crew	249
TNT	249
Evidence	249
Da Breaker Crew	250
Ważne miejsca i źródła w internecie	250
Kilka ogólnych rad od crackerów	252
Cracking (Lucifer48)	252
NOP Patching (+ORC)	253
Patching (MisterE)	253
Myśleć jak cracker (rudeboy)	253
Tools (rudeboy)	253
Rozdział 11. Referencje	255
Podstawowe instrukcje asemblera	255
Komunikaty Windows	261
Dostęp do rejestrów	264
Przegląd funkcji programu SoftICE	267
Ustawianie punktów wstrzymania	267
Manipulowanie punktami wstrzymania	268
Wyświetlanie i zmiany w pamięci	268
Wyświetlenie informacji systemowych	268
Polecenia dla portów we-wy	270
Polecenia sterujące przepływem danych	270
Tryby pracy	270
Polecenie wydawane przez użytkownika	270
Użyteczne polecenia	271
Polecenia klawiszy w edytorze liniowym	271
Przewijanie	272
Polecenia do manipulowania oknami	272
Obsługa okien	272
Polecenia dla symboli i źródeł	273
Operatory specjalne	273
Skoki warunkowe, bezwarunkowe i instrukcje set	273
Skoki warunkowe (przejęto z CRC32 Tutorial #7)	273
Skoki bezwarunkowe (przejęto z CRC32 Tutorial #7)	275
Instrukcje SET (AntiMaterie)	276
Algorytm CRC-32	277
Kolejne algorytmy do zastosowania z szyfratorami i kompresorami PE	279
Demonstracja algorytmu szyfrującego	280
Drobne poprawki do ProcDumpa	284
Interfejs BoundsChecker	287
Get ID	287
Set Breakpoint	288
Activate breakpoint	288
Deactivate Lowest Breakpoint	288
Get Breakpoint Status	288
Clear Breakpoint	288
Rozdział 12. Podsumowanie	289
Skorowidz	291

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 deasemblerzy stały się bardzo popularne. Dzięki tym funkcjom cracker często może się dostać do algorytmu zabezpieczającego, bez przeszkód go przestudiować i następnie usunąć. To czyni deasemblerzy potężną bronią nawet w rękę żółtodzioba.

Używane deasemblerzy

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.

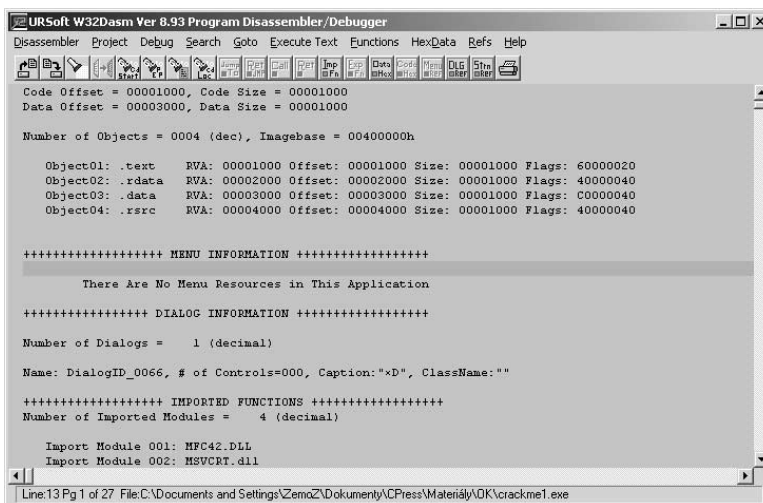
W32Dasm jest przeznaczony dla wszystkich grup użytkowników. Nie mamy zbyt wielkiego wpływu na przebieg i ustawienia deasemblacji, co rekompensowane jest przez bardzo prostą i intuicyjną obsługę oraz przejrzystość kodu wynikowego. Program oferuje tworzenie odnośników do łańcuchów oraz funkcji importowanych, 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 deassemblerów, 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 deasemblovac również programy, w 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 deasemblovac (kliknij na pozycję *Open File to Disassemble...* i wybierz plik). Po tym, jak plik zostanie zdeasemblovany (czas tego procesu zależy od wielkości pliku), uaktywnione zostaną kolejne funkcje programu.

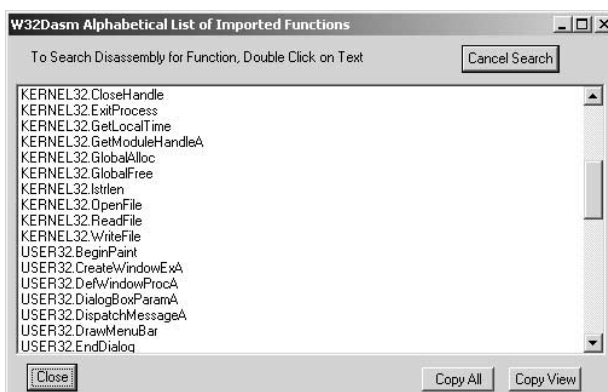
Rysunek 3.1.
Środowisko pracy deasemblera *W32Dasm*



Naciskając na klawisz *F10* lub pozycję menu *Goto Program Entry Point* przejdiesz 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*.

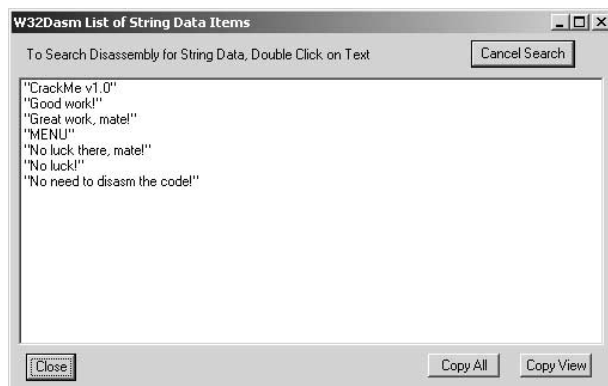
Rysunek 3.2.
Zorientowanie się w spisie funkcji importowanych nie jest problemem



Jeżeli dwukrotnie klikniesz na wybraną funkcję API, przejdiesz do miejsca jej wywołania w kodzie programu. Jeżeli funkcja jest wywoływana wielokrotnie, w kolejne miejsce, gdzie jest wywoływana, przejdiesz 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 zdeasemblovanym kodem programu i orientacja w nim jest bardzo łatwa. Odnośniki do skoków i instrukcji CALL są przejrzysto wkomponowane w kod, tak więc nie ma problemu nawet w przypadku bardziej złożonych algorytmów. Jasno-niebieski 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 jakiegokolwiek 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

Deassembler 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 deassembler 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 wgrzywać 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 (choćby 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 deassembler 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 Modifying 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 assemblerze. 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 niewyobrazalny 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 00401003
:00401002 2853BB sub [ebx][-0045],d1
:00401005 7856 js 0040105D
:00401007 3412 xor al,012
:00401009 EB15 jmps 00401020
:0040100B 2881F3214365 sub [ecx][0654321F3],a1
:00401011 87EB xchg ebp,ebx
:00401013 02E8 add ch,a1
:00401015 6981F35815519590EB04 imul eax,d,[ecx][0511558F3],004EB9
:0040101F C2EBEA retn 0EAEB
: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],a1
:00401011 xchg ebp, ebx
:00401013 add ch,a1
: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 assemblerze 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 `VirtualProtect` 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 kilkakrotnie 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*.