PORADNIKI

Reverse Engineering w aplikacjach komputerowych

I. Krótkie wprowadzenie

1. DEFINICJE

Język programowania jest to program, który pozwala nam pisać programy które będą zrozumiane przez komputer. **Aplikacja** jest to dowolny skompilowany program, który jsest złożony za pomocą języka programowania

Reverse Engineering (RE) jest to dekompilacja dowolnej aplikacji, bez względu na język programowania, który był używany do jej stworzenia,aby uzyskać jej kod źródłowy dowolnej jej części.

Reverse engineer może użyć ponownie tego koduw swoich własnych programach lub zmodyfikować już istniejące (już skompilowane) programy dla wykorzystania w inny sposób. Może użyć wiedzy uzyskanej z RE dla poprawy aplikacji programów, znanych również jako bugi. Ale najważniejsze jest to ,że może uzyskać niezwykle użyteczne idee przez obserwowanie jak inni programiści pracują i myślą, zatem poprawiają swoją wiedzę i umiejętności! Oto kilka powodów dla których istnieje obecnie RE a jego zastosowanie zwiększa się każdego roku:

- Osobista edukacja
- Zrozumienie (poprawa) ograniczeń i defektów w narzędziach
- Zrozumienie (poprawa) defektów w obcych produktach
- Tworzenie produktów kompatybilnych z (aby móc pracować) innymi produktami
- Tworzenie produktów kompatybilnych z (aby móc współdzielić dane) innymi produktami
- Nauczenie się zasad jakimi kierują się konkurenci
- Określenie czy inne firmy nie kradną i używają czyjegoś kodu źródłowego

Nie wszystkie z tych rzeczy można uznać za "legalne".Zazwyczaj każdy produkt ma prawa autorskie lub warunki licencyjne.

2. TYPOWE PRZYKŁADY

Pierwsze co przychodzi na myśl kiedy słyszymy RE, to **cracking.** Cracking jest tak stary jak samo programowanie. Crackowanie programu oznacza wyśledzenie i użycie numeru seryjnego lub innej informacji rejestracyjnej, wymaganej dla właściwego działania programu. Dlatego też, jeśli program typu shareware (rozprowadzany darmowo, ale z pewnymi niedogodnościami, takimi jak ograniczone funkcje, nag screeny lub ograniczona funkcjonalność) wymaga poprawnej informacji rejestracyjnej, reverse engineer może dostarczyć tej informacji przez zdekompilowanie określonej części programu

2.1 HACKING

Hakkerzy mają możliwość dostania się do publicznych lub prywatnych serwerów i zmodyfikować ich parametry. Może to brzmieć egzotycznie i raczej dziwnie, ale jest oparte na RE'owaniu systemu operacyjnego i wyszukiwaniu słabości. Rozważmy serwer umiesczony pod adresem http://www.hackme.com/. Kiedy logujemy się na ten serwer przez ftp, telnet, http lub w jakiś inny sposób, możemy łatwo odkryć jaki system operacyjny jest uruchomiony na tym serwerze. Wtedy, reverse engineerujemy moduły bezpieczeństwa tego systemu operacyjnego o szukamy exploitów. Przykładem mogą być serwery Windows. Hacker odwrócił moduł run32.dll i odkrył ,że zmienna, która określa liczbę otwierającą Wiersz Poleceń to bajt (może być od 0 do 255) Dlatego też, jeśli

mógł otworzyć 257 okien wiersza poleceń, można zawiesić system! Ta słabość została załatana wiele lat temu. Miała ona formę "łatek".Za każdym razem , kiedy łatka zostałą stworzona, stare słabości znikają a pojawiają się nowe.

2.1 TELEFONY KOMÓRKOWE

Telefony komókowe oprogramowaniem stoją. Ich menu, funkcjonalności, problemy i funkcje są wynikiem działania oprogramowania, które zwykle jest przechowywane w modułach pamięci. Ponieważ musimu działać na oprogramowaniu możemy wykonać RE na nim i przejść do nieudokumentowanych funkcji i/lub problemów. Weźmy na przykład Nokię 5210. Producent utrzymuje ,że kod bezpieczeństwa jest nie do złamania. Tylko twardy reset możę odblokować ten telefon. Błąd! W dowolnym zablokowanym telefonie wpisz "*3001#12345#". Tajne menu pojawi się i wyświetli wiele ciekawych rzeczy, twój bezpieczny kod. Tym właśnie usługodawca odzyskuje tajny kod jaki zgubiłeś. Super! Ale jak można wykryć tą tajną sekwencjęliczb? Praktycznie to nieskończona liczba kombinacji liczb losowych. To proste. Zrzuć oprogramowanie na dysk komputera. Potem zrób RE oprogramowania i znajdziesz wiele "tajnych" kodów.

2.2 APLIKACJE KOMPUTEROWE

Przyjrzyjmy się grze Saper, która jest dostarczana z każdą wersją Windows. To rzeczywiście prosta gra z niewielkimi możliwościami (i praktycznie bez błędów). Aby w nnią zagrać trzeba wejść w Programy, Akcesoria, Gy i kliknąć Saper. Większość ludzi nie wie ,że gra składa się z dwóch plików. Pliki te są w katalogu instalacyjnym Windowsa (zazwyczaj \Windows lub \Winnt) i są to "Winmine.exe" i "Winmine.ini". Wiemy ,że plik .exe jest plikiem wykonywalnym (lub programem głównym) a plik .ini przechiowuje ustawienia. Spójrzmy bliżej na plik .ini. Wygląda tak:

```
[Minesweeper]
Difficulty=1
Height=16
Width=16
Mines=40
Mark=1
Color=1
Xpos=80
Ypos=76
Time1=999
Time2=999
Time3=999
Name1=Anonymous
Name2=Anonymous
Name3=Anonymous
```

Przejdźmy do lini

Menu=1
Sound=3

Linia menu=1 powoduje ,że znika menu Sapera. Druga linia powoduje pojawianie się dźwięku kiedy wygrywasz (3 różne).

3. WYMAGANIA

Chociaż może to zabrzmi dziwnie dla początkujących, RE jest w rzeczywistości prosty i dużo prostszy niż tworzenie programu. Jeśli ktoś jest programistą, musi wymyślać i tworzyć. Z drugiej

strony, kiedy dekompilujemy program, odczytujemy myśli programisty, które przed nami próbował ukryć.
Nie jest wymagane doświadczenie programistyczne. Jednak, jeśli takowe masz, znacznie ułatwia ono zrozumienie problemu. To co jest konieczne na potrzeby tego zadania to ogólna wiedza o Systemie Operacyjnym Windows. Również połączenia internetowe i konto email, dla uzyskiwania materiałów szkleniowych dystrybuowanych przez Internet.

4. ZAKRES

Naszym głównym celel będzie możliwość RE dowolnej aplikacji komputerowej i możliwość częściowego zrozumienia co dzieje się w programie, Każdy powinien móc wykonać techniki RE i zrealizować pewnne proste zadania. W szczególności skupimy się na:

- Wejściach i wyjściach komputera
- Działaniu OS
- Analizie pliku wykonywalnego
- Asemblacji i dezasemblacji
- Komercyjnych i damowych narzędziach do RE
- Zaawansowanych technikach RE

II. Programowanie Procesora

5. JĘZYKI PROGRAMOWANIA

Jest wiele sposobów na programowanie procesora. Generalnie są trzy generacje języków programowania. Obecnie najpopularniejsze są te trzeciej generacji. Poniżej są przedstawione warianty istniejących języków. Kod maszynowy jest językiem generacji zero ponieważ nie jest językiem!)

Generacja	Język
Pierwsza	Assembler
Druga	Fortran, C, Basic, Pascal,Cobol
Trzecia	Visual C++, Visual Basic, Delphi

Są różne sposoby aby odróżnić języki drugiej a trzeciej generacji. Ważną cechą języków trzeciej generacji jest to ,że obsługują Programowanie Zorientowane Obiektowo (OOP) i stosowanie obiektów. Czyni je to niezwykle elastycznymi i mocnymi, a zatem pozwala to programistom na tworzenie aplikacjio z atrakcyjnym interfejsem szybko i łatwo. Z powyższego wynika ,że assembler jest językiem prymitywnym, dlatego prawie przestarzałym. To nie prawda. Assembler będzie istniał tak długo jak długo będą procesory. Pozwala on na bezpośrednią komunikację z procesorem, a ten pozwala na komunikację ze wszystkimi urządzeniami peryferyjnymi. Wyobraź sobie ,że piszesz program w Fortranie. Kiedy skończysz kod źródłowy, musisz go sklompilować, innymi słowy stworzyć plik wykonywalny, aby system operacyjny mógł wykonać program. Kompilatopr jest programem zewnętrznym, który tłumaczy kod źródłowy, napisany w dowolnym języku (drugiej lub trzeciej generacji) na kod maszynowy. Każdy język używa innego kompilatora ,ale wszystkie programy koniec końców są konwertowane na pliki wykonywalne. Bez względu na język użyty do stworzenia programu, zawsze disasemblujemy plik wykonywalny, tj.konwertujemy kod wykonywalny na obszerny kod asemblerowy. Jedyny problem jest taki ,że asembler jest raczej trudnym językiem i uzależnionym od procesora; dlatego też musimy nauczyć się wiele określonych instrukcji i ,oczywiście, zapoznać się z pojęciami języka programowania asemblera. Generalnie,

jest to bardzo trudne i wymaga wiele czasu i praktyki. Jednakże, łatwiej jest nauczyć się jak "czytać" pewne części zdesamseblowanego kodu i wyodrębnić potrzebne informacje, wtedy skonwertować go do innego języka . Jedynym wyjątkiem od tych zasad się Java (możemy uzyskać kod źródłowy w Javie) i Visual Basic w wersji 2 i 3 (które mają kod źródłowy przechowywany w pliku wykonywalnym, zatem wyodrębnienie jest prostym zadaniem).Poniższa lista pokazuje języki programowania od największej do najmniejszej ilości instrukcji potrzebnych dla punktu funkcyjnego. Obecnie jest tendencja tworzenia języków, które wykonują wiele funkcji w tle i ułatwiają życie programiście. Języki z większą ilością instrukcji na punkt funkcyjny są trudniejsze do nauczenia i zastosowania. Zwróć uwagę na miejsce C++ i Visual Basic. Jeśli mamy program napisany w assemblerze, będziemy potrzebować 53 razy więcej instrukcji niż przy jego stworzeniu za pomocą VBA. Pytanie brzmi , czy wszystko możemy zrobić w VBA? Głupstwem byłoby stosowanie asemblera lub fortran77 do tworzenia wykresu. Jednakże, jeśli masz zamiar mieć bezpośredni dostęp do komórki pamięci zmiennej, wtedy nie możesz zrobić tego innym językiem jak tylko asemblerem.

Język	Ilość instrukcji
Assembler	320
C	125
Fortran77	110
Cobol	90
Smaltalk	80
Lisp	65
C++	50
Oracle(Baza danych)	40
Visual Basic	30
Perl	25
VBA	6

6. ARYTMETYKA PROCESORA

Jedyną rzeczą jaką procesor komputera może zrozumieć jest przełączanie. Mówimy tu o najprostszym sposobie przełączania, tylko z dwoma położeniami: włączony i wyłączony. Kiedy przełącznik jest ustawiony na włączony (lub TRUE) mamy wartość 1. W przeciwnym razie, kiedy przełącznik jest ustawiony na wyłączony (lub FALSE) mamy wartość 0. Taka notacja jest dobra gdyż jest łatwa do zrozumienia. Ale wprowdza pewne problemy. Zobaczmy jak komputer rozumie nasze liczby. Ponieważ ma tylko dwa symbole (1 i 0) do przedstawiania wszystkiego, nie możemy użyć innego systemu liczbowego jak dwójkowy. Aby skonwertować liczbę z binarnej na dziesiętną musimy zrobić co następuje:

$$01101 = 0x2^4 + 1x2^3 + 1x2^2 + 0x2^1 + 1x2^0 = 13$$

Zwróć uwagę,żę wykładniki zaczyna się zliczać od zera od prawej do lewej strony i zwiększane są o 1 dla każdej cyfry. Może to być zakres dla praktycznie dowolnej liczby cyfr. Każdy z przełączników jest bitem. Tak więc łątwo jest zrozumieć co to jest 16 bitów lub 32 bitów. Dla 16 bitowych systemów operacyjnych (takich jak Windows 3.11) największa liczba jak może mieć 16 cyfr , wszystkie ustawione na 1, to 65535. Nawet dla 32 bitowych systemów operacyjnych, największa liczba (ze znakiem) to 2147483647, również jest zbyt mała.
Sztuczką jest tu użycie wykładnika. Dla liczb większych niż 2.14 miliarda, procesor używa liczby 200, któa zajmuje 8 bitów a pozostałe 8 bitów jest używanych dla reszty liczby. Ta sama sztuczka jest wykorzystywana dla przedstawiania liczb rzeczywistych (zmiennoprzecinkowych).

- 21.4 może być zapisane jako .21400 002, gdzie ostatnie trzy cyfry są wykładnikiem potęgi 10. $.214 x 10^2 = 21.4$
- $5.5x10^{199}$ może być zapisane jako .55000 200 (zwróć uwagę ,że zmiennoprzecinkowość nie jest używana, ponieważ pierwsza cyfra to 0 -> 0.55000 200 więc możemy bezpiecznie usunąć 0 sprzed każdej z tych liczb)

Notacja ta nie jest bezpośrednio stosowana w komputerach, ponieważ jak wiesz, komputery rozumieją tylko 0 i 1 . Aby wymusić zrozumienie przez procesor liczby 0.3 musimy ją zadeklarować jako dzielenie:

- 0.3 = 3/10 = 00000011 / 00001010 -> 0.010891... a proceosr nie będzie zdolny wyliczyć równowartości dla 0.3!
- dla 0.375 = 3/8 = 00000011 / 00001000 -> 0.011, to nie problem

Wynik tego zapisu jest taki ,że PC nie może wykonać nawet najłatwiejszego dodawania! Rozważmy coś takiego:

Listing w Basicu

```
Dim i
Dim Sum
For i=1 to 100
Sum=Sum+1
Next i
```

Listing w C/C++

```
Int main()
{
int i;
double sum;
for (i=1;i<100;i++)
sum=sum+1;
return 0;
}
```

Listing w Fortranie

```
DO 50 I=1,100,1
SUM=SUM+1
50 CONTINUE
```

Bez względu na to jakiego języka użyjemy, wynik jest taki sam : **nie 100 !!!** Faktycznie, będzie to liczba bliska 100,taka jak 99.99999283 i jeśli zaokrąglimy tą liczbę (oczekujemy liczby całkowitej) otrzymamy 100.

Bardzo trudno ludziom używać innego systemu liczbowego niż dziesiętny. Jednak, jest jeszcze jeden system, szesnastkowy (heksadecymalny), któy jest bardzo poocny ponieważ jest podzielny przez 8. Liczba 8 jest magiczną liczbą w PC. Bity są dzielone przez 8 (8, 16, 32 i 64). Inne liczby które mogą być przedstawiane przez 8 bitową liczbę to 256 (podzielne przez 8), przez 16 bitów, 65535 (ponownie podzielna przez 8) itd.
Heksadecymalny system liczbowy ma 16 symboli, od 0 do 9 i od A do F. A jest równe 10, B to 11 a F to 15. Dlatego liczba :4

98DC dziesiętnie to $9 \times 16^3 + 8x16^2 + 13x16^1 + 12x16^0 = 39132$

Liczby szesanstkowe są zazwyczaj przedstawiane ze znakiem & przed nimi (Basic) lub z symbolem 0x (C/C++). 0x18 jest szesnastkową liczbą równą 1x16 + 18 = 24 podczas gdy 18 jest dziesiętnym odpowiednikiem 0x12

Oto tablica pozwalająca szybko znajdować liczby heksadecymalne /dziesiętne od 0 do 255.

	0	1	2	3	4	5	6	7	8	9	A	B	C	F	E	F
0	0	1	2	3	4	5	6	7	8	9	19	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

Przykład: 0xD6 = 214 a 102 = 0x66

7. STRUKTURA PAMIĘCI

Jest niezbędne aby dobrze zrozumieć koncepcję pamięci. Są trzy typy pamięci komputerowej: fizyczna tymczasowa (znana również jako RAM), tymczasowa rzeczywista (plik strony pamięci rzeczywistej) i stałą fizyczna (lub magazyn – dysk twardy). Tak więc procesor ma dostęp do swojej tymaczasowej lub stałej pamięci za każdym razem kiedy instrukcja jest wykonywana. Dysk twardy może być rozpatrywany jako duży RAM, która jest stała, w warunkach kiedy system jest resetowany. Jednak zawartość, która jest przechowywana może być modyfikowana lub likwidowana bez żdnego rodzaju ograoniczeń. Dodatkowo do tego, nowoczesne płyty główne mają chipsety EEPROM, które dostarczają ROM dla użytkownika. W tych chipach jest przechowywany program BIOS. Oczywiście zawartość EEPROM może być czasowo zmieniana (za pomocą specjalnych instrukcji). Za każdym razem kiedy aplikacja jest ładowana, zajmuje jakąś przestrzeń w dostępnej pamięci. Jeśli nie ma dość dostępnej pamięci, wtedy aplikacja nie może być załadowana. Przez termin aplikacja rozumiemy program wykonywalny (od systemu operacyjnego do sterowników) Dlaczego jednak nie rozpatrywać tylko pamięci fizycznej jako jedynej dostępnej pamięci. Windows (i inny systemy operacyjne) ma tendencję do znacznego zwiększania dostępnej

pamięci fizycznej przez wykorzystanie wolnej ilości miejsca na dysku twardym. Jest to wykonywane prze system pamięci wirtualnej. Tworzony jest plik nazwany WIN386.SWP, (rezydujący zwykle w katalogu głównym) i jest używany jako rozszerzenie do istniejącej pamięci fizycznej. Pamięć fizyczna może być rozpatrywana jako dysk twardy z super szybkim dostępem, gdzie OS może przechowywać i mieć dostęp do zmiennych i kodu. Dlatego kiedy nasza pamięć fizyczna jest pełna a OS używa dysku twardego , możemy doświadczyć opóźnień w wykonywaniu programu (dysk twardy jest dużo wolniejszy niż pamięć fizyczna) i aktywności dysku twardego bez robienia czegokolwiek (pewne procesy są aktywne w tle nawet jeśli nie używamy komputera).
Jest możliwość określenia rozmiaru dostępne pamięci wirtualnej poprzez panel sterowania.

7.1 ZMIENNE

System operacyjny i aplikacje używają ziennych wewnętrznie jak i imiędzy sobą. Zmienne te różnią się zawartością i typem. Mogą być liczbami (pojedyncze, całkowite, podójne, zmiennoprzecinkowe itd), łańcuchy (pojedynczy znak, długi łańcuch), wartościami boolowskimi i typami zdefiniowanymi przez użytkownika. Ich zawartość różni się (generalnie) wartościami i odnoszą się do różnych rzeczy. Zmienne te są przechowywane w pamięci, która jest zaalokowana dla aplikacji. Windows alokuje 2 GB pamięci dla każdej aplikacji. To nie jest błąd; to 2 GB , chociaż żadna aplioakcja nie zajmuje tyle miejsca. System operacyjny automatycznie alikuje dość miejsca dla tych zmiennych i może ją relokować je na żądanie. Na przykłąd, liczby całkowite zajmują 4 bajty podcza gdy typ long zajmuje 8 bajtów a znak tylko 1. Jesteśmy szczególnie zainteresowani zmiennymi ponieważ wszystkie działania wymagają stosowania zmiennych. W asemblerze, rejestry są używane zamiast zmiennych; logika pozostaje jednak ta sama. Wyobraźmy sobie podpogram porównania. W większości języków programowania taka instrukcja wygląda tak:

```
[C++] IF (A==B) <zrób coś>
else <zrób coś innego>

[Basic] IF A=B then <zrób coś> else <zrób coś innego>
```

W przykładach tych A i B są zmiennymi. Mogą ale nie muszą być tego samego typu. Każdy język definiuje akceptowalne działania (tj. porównuje integer z long)

7.2. ŁAŃCUCHY UNICODE

W systemach Win32 , łańcuchy zmieniają wewnętrzny format. Przez termin "format wewnętrzny" mamy na myśli sposób w jaki System Operacyjny na nich działa. My zajmiemy się łańcuchami Uniocode. Całe wyszukiwanie ASCII dla łańcuchów będzie wykonywane z włączoną opcją wyszukiwania Unicode w edytorze heksadecymalnym (kiedy taka jest dostępna) Różnica między łańcuchami ANSI a Unicode jest taka ,że znak null (00) jest wstawiana po każym znaku. Dlatego ciąg "ABC", szesnastkowo "585960", będzie traktowany jako "580059006000".

7.3 WSKAŹNIKI

Jeśli zdefiniujemy zmienną A i przypiszemy jej wartość 5, wtedy możemy być pewni ,że za każdym razem kiedy pytamy o wartość tej zmiennej, będzie to 5, chyba ,że ją zmienimy. Jedyne czego nie możemy być pewni to położenie tej zmiennej w pamięci. Weźmy na przykład fragment pamięci:

```
##################****##############################
   ^          ^         ^          ^          ^
```

```
0x4990      0x49A0     0x49B0      0x49C0      0x49D0
```

Jeśli założymy ,że zmienna A jest typu integer, możemy być pewni ,że zajmuje 4 bajty w pamięci fizycznej (RAM lub pamięci wirtualnej) Przypuśćmy ,że możemy "zobaczyć" kiedy w pamięci ta zmienna rezyduje. Jeśli mamy 128 MB RAM a zmienna A jest gdzieś tam, możemy mieć wiersz # -jako ilustrację powyższego, gdzie każde # będzie reprezentowało bajt. Pod adresem 0x49A4 znajdziemy zmienną za pierwszym razem kiedy próbujemy ją wyszukać. Jeśli teraz zamkniemy program uruchomimy go ponownie i ustawimy zmienną A na 5 i wyszukamy jej położenia w pamięci fizycznej, odkryjemy ,że to położenie jest zupełnie inne! System operacyjny oczywiście użył tego położenia które zostało zwolnione po zakończeniu programu do innego celu a teraz zaalokował inną przestrzeń pamięci dla naszej aplikacji i dla tej zmiennej! Dlaczego musimy znać położenie zmiennej w pamięci w dowolnym czasie uruchamiania programu? Ponieważ w ten sposób jest możliwe nadpisanie tej wartość lub coś innego w locie!Wyobraź sobie rozgrywkę Quake II. Powoli tracisz siły gdyż dostępna energia to 12. Jest zmienna która przechowuje ilość energii. Jeśli możemy tylko znaleźć położenie gdzie jest 12. możemy przełączyć debugger i zmienić tą wartość na 150, potem wrócić na arenę i pozabijać wszystkich!
Używamy wskaźników dla odzyskiwania położenia zmiennej w pamięci. Wskaźniki istnieją we wszystkich głównych językach programowania , albo udokumentowane albo nieudokumentowane. W C++ używamy symbolu & przed zmienną aby uzyskać jej adres. W Visual Basic używamy nieudokumentowanej funkcji VarPtr aby uzyskać wskaźnik do zmiennej.

III. Anatomia Windows

8. Windows API

Windows zrewolucjonizował komputery osobiste. Przyniósł wielozadaniowość i przetwarzanie wieloprocesowe w naszych komputerach. Można korzystać z Internetu, słuchać MP3 i korzystać z procesora tekstu w tym samym czasie! Przedtem, były ciemne wieki DOS (Disk Operating System), któy był jednozadaniowy .Można było otworzyć tylko jeden program w danym czasie (owszem były programy TSR, ale to inna historia) Więc jeśli chciałeś zagrać w grę a potem napisaćdokument, musiałeś zakończyć grę i uruchomić procesor tekstu. Było oczywiście wiele ograniczeń w urządzeniach sprzętowych jakie mogły byćobsługiwane, możliwościach Internetu, dostępnej pamięci dla programów itd. Windows przeniósł użytkownika bliżej PC i wprowadził otwartą architekturę dla projektantów. Programiści Windows mają teraz wspólne wskazówki jka tworzyć swoje programy. W DOS, każdy program miał inny interfejs użytkownika. Jedne używały myszki inne nie.Przy Windowsie, nie ważne jakiej apliakcji używamy, oczekujemy pewnych funkcji i oczekiwanych zachowań programu. Rozważmy pasek tytułowy dowolnego okna, przyciski, pola wyboru itp. Użytkownik może teraz łatwo sterować dowolną aplikacją okienkową. Ale jak jest możliwe ,że programista może używać tego samego typu przycisków? Windows jest sprzedawany z API (Application Programming Interface), który składa się z setek funkcji dostępnych dla dowolnego programu okienkowego. Większość funkcji API jest kodowanych w DLL (Dynamic Link Libraries) a programista może użyć ich jeśli połączy swój program z tymi DLL'ami. Jednak problemem jest to ,że API zmienia się wraz ze zmanami Windowsa. Nowe funkcje są wprowadzane, błędy usuwane, stare funkcje stają się przestarzałe. Z tego powodu program stworzony pod Windows 95, może nie pracować dobrze np. z Windows ME, nie wspominając o Windows 7.
Skąd więc to zainteresowanie Windows API? Ponieważ wszystkie programy używają funkcji Windows API. Za każdym razem kiedy klikamy przycisk, tekst jest uzyskiwany z pola tekstowego lub okno jest przesuwane, wykonywana jest jakaś funkcja API. W debuggerze możemy usatwić pułapki i przechwycić wykonywanie programu, które zanjduje się między tymi funkcjami.

9. SYSTEM PLIKÓW

Na początku był FAT (znany również jako FAT16) FAT był systemem plików przez DOS, Windows 3.x i Windows 95 w pierwszym wydaniu. Windows 95 wydanie drugie, Windows 98 i Windows 2000 mogą używać FAT32 i FAT16. Windows NT4, Windows 2000 i pozostałe mogą używać NTFS (NT File System). FAT oznacza File Allocation Table. Znajduje się na dysku twardym a pewne informacje, jakie są używane przez system operacyjny określają gdzie na dysku znajduje się określony plik. Plik może zacząć od położenia,potem przerwanie i restart od innego położenia. Plik jest pofragmentowany a kiedy defragmentujemy dysk twardy, składamy wszystkie fragmenty pliku w jedną całość. Aby uzyskać dostęp (do odczytu lub zapisu) do dysku twardego (lub dyskietki, CD-ROM, DVD) programista musi dołączyć Windows API i wykonać ten dostęp poprzez system operacyjny. Jednak, pewne działania (formatowanie niepoprawnych sektorów, nieoznaczone złe sektory itp.)wymagają dostępu bezpośredniego. Jest to raczej proste z assemblerem, pod Win9x i Windows ME, sterownik VWIN32.VXD musi być użyty lub odpowiedni bezpośredni dostęp API dla Windowsa NT i Windows 2000.

10. ANATOMIA PLIKU

Każdy plik, bez wzgędu na zawartość, ma swoj zadanie. Może to być plik wykonywalny, plik medialny (ikona, kursor, obraz, dźwięk, midi itp), plik tekstowy, pkreślony plik aplikacji (taki jak plik Corel Draw, dokument Excel, prezentacja PowerPoint itp) lub coś innego czego potrzebuje użytkownik. Ważne jest i konieczne aby System Operacyjny wiedział jakiego rodzaju aplikacja powinna przetwarzać dany plik. Pojęcie rozszerzenia pliku (część pliku po znaku kropki) zostanie stworzony przez OS dla identyfikacji pliku przez użytkownika. Spójrzmy na plik "dzieci.jpg" Rozszerzenie .jpg inforuje nas ,że musimy ocekiwać pliku obrazu JPEG, któe musi być przetwarzany przez przeglądarkę / edytor obrazu. Co się stanie jeśli zmienimy rozszerzenie z jpg na bmp? Prawda ,że oba są plikami obrazu, ale system operacyjny będzie "sądził" ,że jest to plik jpg. Zrozumie ,że to nie jest plik bitmapy ale JPEG. Rozważmy również to: dwa pliki logo.sys, logos.sys i logow.sys są plikami obrazu (start i zamykanie screenów logo w okienkach) i mają takie same rozszerzenia co msdod.sys, który jest plikem tekstowym . Sprytne programy takie jak ACDSee mogą zidentyfikować ,że logo.sys jest plikem obrazu, podczas gdy msdos.sys nie. Musi być coś więcej. Większość plików ma nagłówki (poza jawnymi plikami ASCII) Nagłówek jest mała częścią która rezyduje na początku każego pliku i zawiera informacje odnoszące się do jej zawartości. Na przykład, każdy wykonywalny zaczyna się od MZ (stary format DOS) i zawiera mały loader, który może działać w DOS. Zatem, jeśli spróbujemy zadziałać z okienkami w DOS, pojawi się komunikat błędu , wskazując ,że "This program cannot be run in DOS mode" i informuje użytkownika ,że powinien uruchomić go w Windows.

10.1. NAGŁÓWEK PLIKU

Format pliku wykonywalnego systemu operacyjnego jest na wiele sposobów odbiciem założeń i zachowań wbudowanych w system operacyjny. Chociaż studiowanie wejść i wyjść formatu pliku wykonywalnego nie jest czymś co zazwyczaj pojawia się najwyżej na liście rzeczy do wykonania programisty, jednak poznanie tego może być bardzo przydatne. Łączenie dynamiczne, zachowanie loadera i zarządzanie pamięcią są trzema przykładami specyfiki systemu operacyjnego, które mogą być wywnioskowane przez studiowanie formatu pliku wykonywalnego. Dal zrozumienia działania jądra systemu Windows, musimyzrozunieć format PE: To proste. I oczywiście musimy zroumieć te jądra gdyż mamy zamiar je reversować. Powszechnie wiadomo ,że pierwszy Windows 32 bitowy , Windows NT jest spadkobiercą VAX VMS i UNIXa. Wiele kluczowych elementów Nt zaprojektowano dla tej platformy przed pojawieniem się Microsoftu. Kiedy nadszedła czas na NT, było naturalne ,że bedzie próbował zminimalizować czas ładowania przez użycie wcześniej napisanych i przetestowanych narzędzi. Format wykonywalny i moduł obiektu jaki produkują te

narzędzia jest nazywany COFF (Common Object File Format). Relatywnie stara natura (w latach komputerowych) COFF może być widziana w ten sposób ,że pewne pola w tych plikach są określone w foramcie ósemkowym. Format COFF był dobrym punktem startowym, ale także koniecznym rozszeniem dla kolejnych , nowowczesych systemów operacyjnych Windows. Wynikiem tej aktualizacji jest format PE (Portable Eecutable) Jest nazywany przenośnym ponieważ wszystkie iplementacje NT na różne platformy (Intel 386, MIPS, Alpha, Power PC itd) używają tego samego formatu wykonywalnego. Są różnice w binarnym kodowaniu instrukcji CPU. Nie możesz uruchomić skompilowanego wykonywalnego PE MIPS'a w systemie Intel. Ważne jest ,że loader systemu operacyjnego i narzędzia programistyczne nie muszą być całkowicie przepisane dla każdego nowego CPU, jaki pojawia się na scenie.

10.2 FORMAT PE

Format PE jest udokumentowany (w ogólnym słowa tego znaczenia) w pliku nagłówkowym WINNT.H, wraz z pewną zdefiniowaną strukturą dla formatuCOFF, format OBJ. W połowie pliku WINNT.H jest sekcja "Image Format". Sekcja ta zaczyna się od małego rarytasu ze starego dobrego formatu DOS MZ i nagłówków formatu NE przed przejściem do nowszych informacji PE. WINNT.H dostarcza definicji surowej struktury danych używanych przez pliki PE, ale zawiera tylko proste podpowiedzi i użyteczne komentarze wyjaśniające co dane struktury i flagi ozanczają. Autor pliku nagłówkowego dla formatu PE jest z pewnością zwolennikiem długich , opisowych nazw wraz z głeboko zagnieżdżonymi strukturami i makrami. Kiedy kodujemy z WINNT.H, nie jest niczym niezwykłym napotkanie wyrażenia takiego jak to :

```
pNTHeader-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_DEBUG].VirtualAddress;
```

Poza czytaniem o plikach PE, chciałbyś również zrzucić jakieś pliki PE aby przyjrzeć się koncepcjom tu prezentowanym. Jeśli używasz narzędzi Microsoft dla Win32, program DUMPBIN z Visual C++ i Win32 SDK mogą przekształcić wyjściowy plik PE lub pliki COFF/OBJ w formę zrozumiałą dla człowieka. DUMPBIN ma nawet sprytną opcję do disasemblacji sekcji kodu w pliku. W świetle zapewnień Microsoft ,że nie można zdisasemblować ich produktów, jest wielce ciekawe ,że można dostarczyć narzędzia, które czyni łatwym disamsblacje ich programów i DLL'i. Jeśli możliwości disamsblacji EXE'ków i OBJ nie byłaby użyteczna, dlaczego Microsoft zaniepokoił się i dodał tą funkcję do DUMPBIN?
Będziemy używać terminu moduł dla oznaczenia kodu ,danych i zasobów pliku wykonywalnego lub DLL'a, które będą ładowane do pamięci. Poza kdoem i danymi, jakich program używa bezpośrednio, moduł jest również złożony z danych wspierających używanych przez Windows dla określenia gdzie kod i dane są umieszczone w pamięci. W Win16 , struktury danych wspierających są w module bazy danych (segment odnoszący się do HMODULE). W Win32 , informacja ta jest przechowywana w nagłówku PE (struktura IMAGE_NT_HEADERS). Najważniejszą rzeczą jaką trzeba wiedzieć o pliakch PE jest to ,że plik wykonywalny na dysku jest bardzo podobny do modułu po załadowaniu Windowsa. Jest tak ponieważ loader Windows nie musi działać wyjątkowo ciężko tworząc proces z pliku dyskowego. Raczej, loader może być łatwy i użyć pliku odwzorowanej pamięci Win32 dla załadowaniu właściwego fragmentu pliku PE do przestrzeni adresowej programu. Abu użyć analogicznej konstrukcji, plik PE jest dom z prefabrykatów: Jest relatywnie kilka fragmentów,a każdyu fragment może być umocowanyw miejcu przy małym wysiłku. I jak łatwo jest podłączyć elektryczność i wodę w prefabrykowanym domu, równie łatwo jest podłączyć plik PE z resztą świata (tzn. Połączyć go z DLL itd). Tak ssało łatwo stosuje się ładowanie DLL'i. Od momentu zładowania modułu .EXE lub .DLL, Windows może skutecznie potraktować go jak inny plik odworowany w pamięci. Jest to wyraźny kontrast do sytuacji w Windowsie 16 bitowym. 16 bitowy loader pliku NE odczytuje w częściach plik i tworzy oddzielne struktury danych dla przedstawienia modułu w pamięci. Kiedy kod lub segment danych musi być załadowany, loader musi zaalokować nowy segment na globalnej stercie, znajduje miejsce gdzie surowe dane są

przechowywane w pliku wytkonywalnym, wyszukuje tego położenia, odczytuje surowe dane i stosuje odpowiednie rozwiązanie. Dodatkowo, kady 16 bitowy moduł jest odpowiedzialny za zapamiętanie wszystkich selektorów aktualnie używanych, czy segment został zrzucony itd.
Dla Win32, cała używana pamięć przez moduł dla kodu, danych, zasobów, tablic importów, tablic eksportu i innych rzeczy jest w jednym ciągłym zakresie liniowej przestrzni adresowej. Wszystko co musimy zrobić w tej sytuacji jest adres gdzie loader odwzorowuje plik wykonywalny do pamięci. Można potem łatwo znaleźć wszystkie różne fragmenty modułu przez postępowanie za wskaźnikami przechowywanymi jako część obrazu. Inny pomysł z jakim musisz się zaznajomićzanim zaczniemy jest Relative Virtual Addres lub RVA. Wiele pól w plikach PE są określane jako RVA. RVA jest po prostu offsetem jakiejś pozycji, w stosunku do miejsca gdzie odworowany jest w pamięci plik. Na przykład, powiedzmy ,że loader Windows odwzorowuje plik PE do pamięci zaczynając od adresu 0x400000 w wirtualnej przestrzeni adresowej. Jeśli pewna tablica w obrazie zaczyna się od adresu 0x401464, RVA tablicy to 0x1464:

(adres wirtualny 0x401464) – (adres bazowy 0x400000) = RVA 0x1464

Aby skonwertować RVA do używalnego wskaźnika do pamięci, po prostu dodaj RVA do adresu bazowego pod który został załadowany moduł. Termi adres bazowy jest innym ważnym pojęciem do zapamiętania. Adres bazowy opisuje adres startowy odwzorowanej pamięci EXE'ka lub DLL'a. Dla wygody Windows używa adresu bazowego modułu jako wywołanie uchwytu modułu (HINSTANCE) . W Win32 wywołanie adresu bazowego modułu HINSTANCE jest conieco mylące, ponieważ termin wywołania uchwytypochodzi z 16 bitowego Windowsa. Każda kopia aplikacji w Win16 pobiera swój własny odzielny segment danych (i przywiązany globalny uchwyt), który odróżnia go od innych kopii tej aplikacji;dlatego ten termin wywołanie uchwytu W Win32, apliakcje nie muszą rozróżniać się jedne od drugiego ponieważ nie współdzielą one tej samej przestrzeni adresowej.Co ważne przy Win32 jest to ,że możesz wywołać GetModuleHandle() dla dowolnego DLL'a , którego używa twój proces i uzyskać wskaźnik którego można użyć dla uzyskania dostępu do komponentów modułu. Przez komponenty, odnosimy się do jego funkcji importowania i eksportowania, jego relokacji, jego sekcji kodu i danych itd. Innym pojęcie z jakim trzeba się zainteresować kiedy zajmujemy się plikami PE i COFF OBJ. Sekcja pliku PE lub pliku COFF OBJ jest mniej więcej odpowiednikiem segmentu lub zasobów w 16 bitowym pliku NE. Sekcja zawiera albo kod albo dane. Niektóre sekcje zawierają kod lub dane, jakie program dekalruje i używa bezpośrednio, podczas gdy inne sekcje danych są tworzone dla ciebie przez linker i bibliotekę, i zawierają informacje niezbędne systemowi operacyjnemu. W opisach formatu PE przez Microsoft, sekcje są traktowane jako obiekty. Termin ten ma również wiele możliwych konfliktów, jednak przyjmijmy wywoływanie sekcji obszarów kodu i danych.

10.3 NAGŁOWEK PE

Pierwszym przystankiem w naszej podróży po formacie PE jest nagłówek PE. Podobnie jak wszystkie formaty plików wykonywalnych Microsoftu, plik PE ma zbiór pól, które definiują jak wygląda cała reszta pliku. Nagłówek PE zawiera niezbędne fragmenty informacji takich jak położenie i rozmiar obszaru kodu i danych, jak system operacyjny ma zamiar wykorzystać plik, i początkowy rozmiar stosu. Podbnie jak przy innych formatach wykonywalnych z Microsoftu, nagłówek PE nie jest na samym początku pliku. Pierwsze sto bajtów typowego pliku PE jest przeznaczone na namiastkęDOS. Ta namiastka jest minimalnym programem DOS, który wyświetla coś podobnego do komunikatu "This program cannot be run in DOS mode". Wskazuje to ,że uruchomiłeś program Win32 w środowisku , który nie osługuje Win32, pokazując komunikat błędu. Kiedy loader Win32 odwzorowuje pamięć pliku PE, pierwszy bajt odwzorowywanego pliku odpowiada pierwszemu bajtowi namiastki DOS. To prawda. Przy każym uruchomionym programie Win32, otrzymujesz komplementarny program DOS załadowany darmowo ! (W Win16, DOS pośredniczący nie jest ładowany do pamięci)

Podobnie jak w innych formatach wykonywalnych Microsoftu, znajdziesz rzeczywisty nagłówek przez wyszukanie jego startowego offsetu, który jest przechowywany w nagłówku DOS. Plik WINNT.H zawiera definicję struktury dla pośredniczącego DOS'a, który czyni bardzo łatwym wyszukiwanie startu nagłówka PE. Pole e_Ifanew jest offsetem relatywnym (lub RVA, jeśli wolisz) do aktualnego nagłówka PE. Aby uzyskać wskaźnik do nagłówka PE w pamięci, po prostu dodaj wartośćpola do obrazu bazowego:

// Zignorujemy rzutowanie typów i konwersji wskaźnika dla jasności...

pNTHeader = dosHeader + dosHeader -> e_Ifanew;

Kiedy masz już wskaźnik do głównego nagłówka PE, zaczyna się rzeczywista zabawa. Główny nagłówek PE jest strukturą typu IMAGE_NT_HEADERS, zdefiniowany w WINNT.H. Struktura IMAGE_NT_HEADERS w pamięci jest tym czego Windows 95 używa jako własnego modułu bazodanowego. Każdy załądowany EXE lub DLL w Windows 95 jest reprezentowany przez strukturę IMAGE_NT_HEADERS. Struktura ta jest złożona z DWORD i dwóch substruktur, i wygląda następująco:

```
DWORD Signature;
IMAGE_FILE_HEADER FileHeader;
IMAGE_OPTIONAL_HEADER OptionalHeader;
```

Pole Signature pokazane jako tekst ASCII to PE\0\0 (po PE występują dwa bajty 0) Jeśli pole e_Ifanew w nagłówku DOS wskazuje na sygnaturę NE przy tym położeniu zamiast sygnatury PE, będziemy pracowali z plikiem NE Win16. Podbnie , LE w polu signature, będzie wskazywało plik Virtual Device Driver (VXD) LX tu będzie oznaczał plik archiwalny Windows 95, OS/2

10.4 NAGŁÓWEK OBRAZU PLIKU

Po sygnaturze DWORD w nagłówku PE, jest struktura typu IMAGE_FILE_HEADER. Pole o tej strukturze zawiera tylko najbardziej podstawowe inforamcje o tym pliku.. Struturata pojawia się niezmodyfikowana ze swojej oryginalnej implementacji COFF. Poza byciem częścią nagłówka PE, pojawia się również na początku COFF OBJ tworzonych przez kompilatory Microsoft Win32. Pola IMAGE_FILE_HEADER

WORD Machine

CPU dla których to pole jest stworzone

CPU	Kod
Intel I386	0x14C
Intel i860	0x14D
MIPS R3000	0x162
MIPS R4000	0x166
DEC Alpha AXP	0x184
Power PC	0x1F0 (little endian)
Motorola 68000	0x268
PA RISC	0x290

WORD NumbersOfSections
Liczba sekcji zawartych w EXE lub OBJ

DWORD TimeDateStamp
Czas i data, kiedy linker (lub kompilator dla pliku OBJ) stworzył ten plik. Pole to przechowuje liczbę sekund od 31 grudniem 1969 od godziny 16.

DWORD PointerToSymbolTable
Offset pliku tablicy symbolu COFF. Pole to jest używane tylko w plikach OBJ i pliki PE z informacją debuggera COFF. Pliki PE obsługują wiele formatów debuggowania,więc debuggery powinny odnosić się do wejści IMAGE_DIRECTORY_ENTRY_DEBUG w katalogu danych.

DWORD NumbersOfSymbols
Liczba symboli w tabeli symboli COFF

WORD SizeOfOptionalHeader
Rozmiar opcjonalnego nagłówka, któryu może wystapić po tej strukturze. W plikach wykonywalnych, jest to rozmiar struktury IMAGE_OPTIONAL_HEADER, która występuje w tej strukturze. W OBJ. Microsoft mówi ,że pole to przypuszczalnie zawsze będzie wynosić 0. Jednak, przy zrzuceniu biblioteki KERNEL32.LIB, OBJ znajdujący się tam ma wartośćniezerową w tym polu.

WORD Characteristics
Flagi , które zawierają użyteczne informacje o pliku. Niektóre ważne pola są opisane poniżę (pozostałe pola są zdefiniowane w WINNT.H)

FLAGA	KOMENATRZ
0x0001	Nie ma relokacji w tym pliku
0x0002	Plik jest obrazem wykonywalnym (tzn.nie OBJ lub LIB)
0x2000	Plik jest dynamicznie łączoną biblioteką , nie programem

10.5 OPCJOANLNY OBRAZ NAGŁÓWKA

Trzecim komponentem nagłówka PE jest struktura typu IMAGE_OPTIONAL_HEADER. Dla plików PE, ta część z pewnością nie jest opcjonalna. Format COFF zezwala na indywidualną impelementację definiowania struktury z dodatkowymi informacjami poza standardowym IMAGE_FILE_HEADER. Wszystkie pola IMAGE_OPTIONAL_HEADER nie są koniecznie krytyczne dla pozanania przez ciebie. Najważniejsze są pola ImageBase i Subsystem. Jeśli chcesz, możesz pominąć opis tych pól.

WORD Magic
Sygnatura WORD identyfikuje stan obrazu pliku. Poniżej zdefiniowano wartości :

FLAGA	OPIS
0x0107	Obraz ROM
0x010B	Zwykły plik wykonywalny (większość plików zawiera tą wartość)

BYTE MajorLinkerVersion
BYTE MinorLinkerVersion

Wersja linkera, która stworzyła ten plik.Liczby te powinny być wyświetlone jako wartości dziesiętne, zamiast wartości szesnastkowych. Typowa wersja linkera to 2.23

DWORD SizeOfCode
Połączenie i zaokrąglenie rozmiaru wszystkich sekcji kodu. Zazwyczaj, większość plików ma tylko jedną sekcje kodu, więc pole to zazwyczaj odpowiada sekcji .text.

DWORD SizeOfInitializedData
Jest to przypuszczalnie całkowity rozmiar wszystkich sekcji, które są złożone z zainicjowanych danych (nie wliczając w to segmentów kodu). Jednakże, nie wydaje się to spójne z rozmiarem sekcji zainicjalizowanych danych w tym pliku.

DWORD SizeOfUninitializedData
Rozmiar sekcji, jakiej loader przeznaczył przestrzeń w przestrzeni adresów wirtualnych, ale która nie obejmuje przestrzeni w pliku dyskowym. Sekcje te nie muszą mieć określonych wartości przy starcie programu, zatem termin dane niezainicjalizowane. Niezainicjalizowane dane zazwyczaj wchodzą w skłąd sekcji wywołania.

DWORD AddressOfEntry
Adres gdzie obraz zaczyna się wykonywać. Jest to RVA i zazwyczaj może być znajdowany w sekcji .text. To pole jest poprawne zarówno w EXE jak i DLL

DWORD BaseOfCode
RVA gdzie zaczyna się sekcja kodu pliku. Sekcje kodu zazwyczaj zaczyna się przed sekcją danych, a po nagłówku PE w pamięci. RVA jest zazwyczaj to 0x1000 w Microsoft Link tworzącym EXE'ki. TLINK32 Borland zazwyczaj ma wartość 0x10000 w tym polu ponieważ domyślnie wyrównuje obiekty do 64KB granicy, zamiast 4KB jak linker Microsoftu.

DWORD BaseOfData
RVA gdzie zaczyna się sekcja danych pliku. Sekcja danych zazwyczaj znajduje się na końcu pamięci, po nagłówku PE i sekcjach kodu.

DWORD ImageBase
Kiedy linker tworzy plik wykonywalny,zakłada,że ten plik będzie odwzorowany w pamięci do określonejlokacji w pamięci. Adres ten jest przechowywany w tym polu. Zakładamy ,że adres ładowania pozwala linkerowi na optymalizację. Jeśli plik rzeczywiście jest odwzorowany w pamięci do tego adresu przez loader, kod nie potrzebuje łatania przed uruchomieniem. Dla DLL'i, domyślnie jest to 0x400000. W Windows 95 adres 0x10000 nie może być używany do ładowania 32 bitowych EXE ponieważ leży wewnątrz adresu liniowego, który jest współdzielony przez wszystkie procesy. Starsze programy , które były linkowane przy założeniu adresu bazowego 0x10000 będą ładowały się dłużej ponieważ loader musi zastosować relokację bazy.

DWORD SectionAlignment
Kiedy odwzorowujemy do pamięci, każda sekcja ma zagwarantowane rozpoczęcie od adresu wirtualnego, któy jest wielokrotnością tej wartości. Z powodów stronicowania, minimalne wyrównanie sekcji to 0x1000, co jest wartością domyślną w linkerze Microsoftu. W Borland C++ TLINK domyślnie jest to 0x10000 (64 KB)

DWORD FileAlignment
W pliku PE, dane surowe , które zawiera każda sekcja zaczynają się od wielokrotności tej wartości. Wartość domyślna to 0x200 bajtów, prawdopodobnie aby zapewnić ,że sekcje zawsze zaczynać się będą od początku sektora dyskowego (który również ma 0x200 bajtów długości). Pole to jest odpowiednikiem rozmiaru wyrównania segmentu/zasobu w plikach NE. W przeciwieństwie do plików NE, pliki PE zazwyczaj nie mają setek sekcji, więc przestrzeń marnowana przy wyrównaniu

sekcji pliku jest zazwyczaj bardzo mała.

WORD Subsystem
Typ subsystemu ,którego ten plik wykonywalny używa dla swojego interfejsu użytkownika. WINNT.H definiuje poniższe wartości:

Subsystem	Wartość	Komentarz
Native	1	Nie wymaga subsystemu (np. Sterownik)
Windows_GUI	2	Uruchamiany w Windows subsystem GUI
Windows_GUI	3	Uruchamiany w Windows subsystem znaków (apliakcja konsolowa)
OS2_GUI	5	Uruchamia w OS/2 subsystem znaków
POSIX_GUI	7	Uruchamia w Posix subsystem znaków

WORD DllCharacteristics
Zbiór flag wskazujących w jakich okolicznościach funkcja inicjująca DLL (np. DllMain()) zostanie wywołana. Wartośćta pojawia sięzawsze ustawiona na 0, mimo to system wywołuje jeszcze funkcję inicjującą DLL dla wszystkich czterech zdarzeń. Poniżej mamy zdefiniowane wartości:

Wartości	Wyjaśnienie
1	Wywołanie kiedy DLL jest najpierw ładowana do przestrzeni adresowej procesu.
2	Wywołanie kiedy wątek się kńczy
4	Wywołanie kiedy wątek się zaczyna
8	Wywołanie kiedy DLL kończy działanie

DWORD SizeOfStackReserve
Ilość wirtualnej pamięci zarezerwowanej dla stsou początkującego wątku. Nie cała pamięć jest wykorzystana. Domyślnie to pole to 0x100000 (1MB) Jeśli określisz 0 jako rozmiar stosu dla CreateThread(), również wątek wynikowy będzie miał stos o takim rozmiarze.

DWORD SizeOfStackCommit
Ilość pamięci, która jest początkowo przeznaczona dla zainicjowania wątku stosu. Domyślnie to pole to 0x1000 bajtów (1 strona) w Microsoft Linkers, podczas gdy TLINK32 ustawia ją na 0x2000 (2 strony)

DWORD SizeOfHeapReserve
Ilość wirtualnej pamięci zarezerwowanej dla procesu inicjalizacji sterty. Ten uchwyt stery jest uzyskiwany przez wywołanie GetProcessHeap(). Nie cała pamięć jest wykorzystywana.

DWORD SizeOfHeapCommit
Ilość pamięci poczkowo przeznaczona dla procesu sterty. Linker domyślnie wstawia 0x1000 bajtów w to pole.

DWORD LoaderFlags
Z WINNT.H, pole to pojawia się w połączeniu z obsługą debuggowania. Poniżej mamy zdefiniowane wartości:

Wartość	Możliwe wyjaśnienia

1	Przywołuje instrukcję punktu przerwania przed startem procesu
2	Przywołuje debugger do procesu po jego załadowaniu

DWORD NumberOfRvaAndSize
Liczba wejść w tablicy DataDirectory. Wartość ta jest zawsze ustawiona na 16 przez bieżące narzędzie.

IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_MUNBEROF_DIRECTORY_ENTRIES]

Struktura IMAGE_DATA_DIRECTORY. Początkowe elementy tablicy zawierają starowe RVA i rozmiar ważnych części pliku wykonywalnego. Niektóre elementy na końcu tablicy są aktualnie nie używane. Pierwszy element tablicy jest zawsze adresem i rozmiarem tablicy eksportu funkcji (jeśli jest) Drugie wejście do tablicy jest adresem i rozmiarem tablicy funkcji importu itd. Po pełną listę zdefiniowanych wejść tablicy, zajrzyj do IMAGE_DIRECTORY_ENTRY_xxx zdefiniowane w WINNT.H. Zadaniem tej tablicy jest zezwolenie loaderowi do szybkiego znajdowania określonych sekcji obrazu (na przykład, tablica funkcji importu), bez konieczności iteracji poprzez każdy obraz sekcji, porównując nazwy. Większość wejśc tablicy opisuje wejścia sekcji danych. Jednak, element IMAGE_DIRECTORY_ENTRY_DEBUG obejmuje tylko małą część bajtów w sekcji .rdata.

10.6 SEKCJA TABLICY

Między nagłówkiem PE a surowymi danymi dla sekcji obrazu leży sekcja tablicy. Zawiera ona informacje o każej sekcji w tym obrazie. Sekcje w obrazie są posortowane wedle ich adresów startowych a nie alfabetycznie. W tym miejscu warto będzie wyjaśnić co jest w tej sekcji. W pliku NE kod programu i dane są przechowywane w różnych segmentach w pliku. Część nagłówka NE jest tablicą struktur, jedna dla każego segmentu używanego przez program. Przechowywane inforamcje zawierają typ segmentu (kod lub dane), jego rozmiar i jego położenie gdziekowliek w pliku. W pliku PE, sekcja tablicy jest analogiczna do tablicy segmentu w pliku NE. W przeciwieństwie do segmentu tablicy pliku NE, sekcja tablicy Penie przechowuje wartości selektora dla każdego kawałka kodu lub danych. Zamiast tego, każde wejście sekcji tablicy przechowuje adres gdzie surowe dane pliku były odwzorowane do pamięci. Chociaż sekcje są analogiczne do 32 bitowych segmentów, w rzeczywistości nie są pojedynczymi segmentami. Zamiast tego, sekcja po prostu odpowiada zakresowi pamięci w przestrzeni adresów wirtualnych procesu. Inny sposób w jaki pliki PE różnią się od plików NE, to to w jaki sposób obsługują dane których twój program nie obsługuje, ale system operacyjny tak. Dwa przykłady to lista DLL'i, jakich używa program wykonywalny i położenie tablicy przygotowania. W pliku NE, zasoby nie są rozważane jako segmenty. Pomimo ,że mają selektory przypisane do nich, informacja o zasobach nie jest przechowywana w segmencie tablicy nagłówka NE. Zamiast tego. Zasoby są przenoszone do oddzielnej tablicy w pobliże końca nagłówka NE. Informacja o funkcjach importowych i eksportowych również nie gwarantują swoich własnych segmentów, ale w zamian jest upchana przy zmaknięciu nagłówka NE. Historia z plikami PE jest inna. Cokolwiek co może być rozważane jako istotne, to to , że kod i dane są przechowywane w pełnoprawnej sekcji. Zatem, informacja o funkcjach importowania jest przechowywana w swojej wlasnej sekcji, ponieważ jest tablica funkcji, które moduł eksportuje. To samo dotyczy relokacji danych. Dowolny kod lub dana, jakie mogą być konieczne aby program lub system operacyjny pobrały swoje własne sekcje. Bezpośednio po nagłówku PE w pamięci znajduje się tablica IMAGE_SECTION_HEADER. Liczba elementów w tej tablicy jest podana w nagłówku PE (pole IMAGE NT HEADER.FileHeader.NumberOfSections). Program PEDUMP podaje dane wyjściowe sekcji tablicy io wszystkich pół sekcji i atrybutów

01 .text VirtSize: 00005AFA VirtAddr: 00001000

raw data offs: 00000400 raw data size: 00005C00
relocation offs: 00000000 relocations: 00000000
line # offs: 00009220 line #'s: 0000020C
characteristics: 60000020
CODE MEM_EXECUTE MEM_READ
02 .bss VirtSize: 00001438 VirtAddr: 00007000
raw data offs: 00000000 raw data size: 00001600
relocation offs: 00000000 relocations: 00000000
line # offs: 00000000 line #'s: 00000000
characteristics: C0000080
UNINITIALIZED_DATA MEM_READ MEM_WRITE
03 .rdata VirtSize: 0000015C VirtAddr: 00009000
raw data offs: 00006000 raw data size: 00000200
relocation offs: 00000000 relocations: 00000000
line # offs: 00000000 line #'s: 00000000
characteristics: 40000040
INITIALIZED_DATA MEM_READ
04 .data VirtSize: 0000239C VirtAddr: 0000A000
raw data offs: 00006200 raw data size: 00002400
relocation offs: 00000000 relocations: 00000000
line # offs: 00000000 line #'s: 00000000
characteristics: C0000048
INITIALIZED_DATA MEM_READ MEM_WRITE
05 .idata VirtSize: 0000033E VirtAddr: 0000D000
raw data offs: 00008600 raw data size: 00000400
relocation offs: 00000000 relocations: 00000000
line # offs: 00000000 line #'s: 00000000
characteristics: C0000040
INITIALIZED DATA MEN_READ MEM_WRITE
06 .reloc VirtSize: 000006CE VirtAddr: 0000E000
raw data offs: 00008A00 raw data size: 00000800
relocation offs: 00000000 relocations: 00000000
line # offs: 00000000 line #'s: 00000000
characteristics: 42000040
INITIALIZED DATA MEM_DISCARDABLE MEM_READ
07 .drectve PhysAddr: 00000000 VirtAddr: 00000000
raw data offs: 000000DC raw data size: 00000026
relocation offs: 00000000 relocations: 00000000
line # offs: 00000080 line #'s: 00000000
characteristics: 00100A00
LNK_INFO LNK_REMOVE
08 .debug$S PhysAddr: 00000026 VirtAddr: 00000000
raw data offs: 00000102 raw data size: 000016D0
relocation offs: 000017D2 relocations: 00000032
line # offs: 00000080 line #'s: 00000000
characteristics: 42100048
INITIALIZED_DATA MEM_DISCARDABLE MEM_READ
09 .data PhysAddr: 000016F6 VirtAddr: 00000000
raw data offs: 000019C6 raw data size: 00000D87
relocation offs: 0000274P relocations: 00000045
line # offs: 00000000 line #'s: 00000000
characteristics: C0480048
INITIALIZED_DATA MEM_READ MEM_WRITE
10 .text PhysAddr: 0000247D VirtAddr: 00000000
raw data offs: 000029FF raw data size: 000010DA
relocation offs: 00003AD9 relocations: 000000E9
line # offs: 000043F3 line #'s: 000000D9

characteristics: 60500020
CODE MEM_EXECUTE MEM_READ
85 .debug$T PhysAddr: 00003557 VirtAddr: 00000000
raw data offs: 00004909 raw data size: 00000030
relocation offs: 00000008 relocations: 00000000
line # offs: 00000000 line #'s: 00000000
characteristics: 42]00048
INITIALIZED_DATA MEM_DISCARDABLE MEM_READ

Każdy IMAGE_SECTION_HEADER jest kompeltną baządanych informacji o jednej sekcji w EXE lub pliku OBJ, i ma następujący format:

BYTE Name[IMAGE_SIZEOF_SHORT_NAME]

Jest to 8 bitowa nazwa ANSI (nie Unicode), która nazywa sekcję. Większość nazw sekcji zaczyna się od **.** (kropki, np. .text), ale nie jest to wymagane, mimo tego w co każe ci wierzyć dokumentacja PE. Możesz nazywać własne sekcje albo dyrektywą segmentu w języku asemblera albo #pragma data_seg i #pragma code_seg w kompilatorze Microsoft C/C++ (Użytkownicy Borland C++ powinni stosować #pragma codeseg) Ważne odnotowania jest to ,że jeśli nazwa sekcji zabiera pełne 8 bajtów, nie ma kończącego bajtu NULL. Jeśli jesteś zapalonym miłośnikiem printf(), możesz użyć "%8s" aby uniknąć kopiowania nazwy łańcucha do innego bufora aby zakończyć go null'em.

union {
DWORD PhysicalAddress
DWORD VirtualSize
} Misc;

Pole to ma różne znaczenia w zależności tego czy występuje w EXE czy OBJ.W EXE, przechowuje wirtualny rozmiar sekcji kodu lub danych. Jest to rozmiar przed zaokrągleniem do najbliższej wielokrotności wyrównania pliku.Pole SizeOfRawData później w strukturze przechowuje wartość tego zaokrąglenia. Co ciekawe, TLINK32 Borlanda odwraca znaczenie tego pola i pola SizeOfRawData, i pojawia się jako poprawny linker. Dla plików OBJ, pole to wskazuje adres fizyczny sekcji. Pierwsza sekcja zaczyna się od adresu 0. Aby znaleźć adres fizyczny kolejnej sekcji, dodajemy wartość SizeOfRawData do fizycznego adresu bieżącej sekcji.

DWORD VirtualAddress
W EXE, pole to przechowuje RVA pdo którym loader może odwzorować sekcję. Dla obliczenia realnego adresu startowego danej sekcji w pamięci, dodaj adres bazowy obrazu do VirtualAddress sekcji przechowywanej w tym polu. W narzędziach Microsoftu, pierwsza sekcja domyślna RVA to 0x1000. W OBJ, pole to nic nie znaczy i jest ustawione na 0

DWORD SizeOfRawData
W EXE pole to zawiera rozmiar sekcji po jej zaokrągleniu do roziaru wyrównania pliku. Na przykład, założmy rozmiar wyrównania pliku na 0x200. Jeśli pole VirtualSize mówi ,że sekcja to 0x35A bajtów długości, pole to powie, że sekcja jest długa na 0x400 bajtów. W OBJ, pole to zawiera dokładny rozmiar sekcji wyemitowany prze kompilator lub asembler. Innymi słowy, dla OBJ jest to odpowiednik pola VirtualSize w EXE.

DWORD PointerToRawData
Pole to jest offsetem pliku gdzie surowe dane dla sekcji mogą być znalezione. Jeśli odwzorowujesz pamięć PE lub pliku COFF (zamiast wynająć system operacyjny do jego załadowania), pole to jest ważniejsze niż pole VirtualAddress. Powoduje to, że w tej sytuacji masz całkowicie liniowe odwzorowanie całego pliku, więc znajdziesz dane dla sekcji pod tym offsetem zamiast pod RVA

określonym w polu VirtualAddress.

DWORD PointerToRelocations
W OBJ , jest to offset pliku do informacji relokacji dla tej sekcji. Informacja relokacji dla każdej sekcji OBJ bezpośednio następuje do surowych danych dla tej sekcji. W EXE, pole to (i pole kolejne) są mało znaczącei są ustawione na 0. Kiedy linker tworzy EXE, rozwiązuje więcej problemów, pozostawiając tylko relokacje adresu bazowego i funkcji importowych rozwiązywanych w czasie ładowania. Informacja o relokacji bazy i funkcjach importowania jest przetrzymywana w bazie relokacji i sekcji funkcji importowania.

DWORD PointerToLinenumbers
Offset tablicy numeru linii. Tablica numeru linii koreluje numery linii pliku źródłowego z adresami gdzie kod generowany dla danej linii może być znaleziony. W nowoczesnych formatach debuggowania, takie jak format CodeView, informacja o numerze linii jest przechowywana jako część informacji debuggowania. W formacie debuggowania COFF jednak, informacja o numerze linii jest konceptualnie odrębna od informacji nazwy / typu symbolicznego. Zazwyczaj, tylko sekcje kodu (np. .text lub CODE) mają numery linii. W plikach EXE numery linii są skupione blisko końca pliku, po sekcji dla surowych danych. W plikach OBJ, tablica muerów linii dla sekcji przychodzi po sekcji surowych danych i tablicy relokacji dla tej sekcji.

WORD NumberOfRelocations
Liczba relokacji w tablicy relokacji dla tej sekcji (poprzednio omówione pole PointerToRelocations). Pole to pojawia się tylko do użycia w plikach OBJ.

WORD NumberOfLinenumbers
Liczba numerów linii w tablicy numerów linii dla tej sekcji (pole PointerToLinenumbers)

DWORD Characteristics
Kiedy większość programistów wywołuje flagi, format COFF/PE odnosi się do charakterystyk. Pole to jest zbiorem flag, które wskazują atrybuty sekcji (kod / data , odczytywanie, zapisywanie itd). Po kompletną listę wszystkich możliwych atrybutówsekcji , zajrzyj do IMAGE_SCN_XXX XXX   #defines w WINNT.H

Zastosowanie flag
Ciekawe jest co jest przechowywane dla każej sekcji. Po pierwsze, zwróć uwagę ,że nie wskazuje atrybutu PRELOAD. Format pliku NE pozwala określić atrybut PRELOAD dla segmentów, które powinny być załadowane bezpośednio w czasie ładowania modułu. Format LX OS/2 2. ma coś podobnego, pozwalającego na określić ,że powino być przeładowanych do 8 stron. Format PE, z drugiej strony , nie ma czegoś takiego.

Flagi COFF	Wyjaśnienie
0x00000020	Sekcja zawiera kod. Zazwyczaj ustawiona w połączeniu z flagą wykonywalną
0x00000040	Sekcja ta zawiera niezainicjowane dane. Prawie wszystkie sekcje z wyjątkiem wykonywalnej i sekcji .bss mają ustawioną tą flagę
0x00000080	Sekcja zawiera niezainicjalizowane dane (np. Sekcja .bss)
0x00000200	Sekcja zawiera komentarze lub inny typ informacji. Typowe

	użycie tej sekcji jest sekcja .drectve tworzona przez kompilator, która zawiera polecenia dla linkera
0x00000800	Zawartość tej sekcji nie powinna być wstawiana do końcowego pliku EXE. Sekcja jest używana przez kompilator / assemler do przekazywania informacji do linkera
0x02000000	Sekcja może być pominięta, ponieważ nie jest konieczna procesowi już załadowanemu. Większość pomijanych sekcji jest w sekcji bazowych relokacji (.reloc)
0x10000000	Ta sekcja jest współdzielona. Kiedy używamy jej z DLL , dana w tej sekcji jest współdzielonaz innymi procesami używającymi DLL. Domyślnie dla sekcji danych jest to nie współdzielone, oznaczając,że każdy proces używający DLL'a uzyskuje swoją odzielną kopię tej sekcji danych. Bardziej technicznie, sekcja współdzielona mówi menadżerowi pamięci aby ustawił odwzorowanie strony dla tej sekcji aby wszystkie procesy używające DLL odnosiły się do tej samej strony fizycznej w pamięci.Aby uczynić sekcję współdzieloną, użyj atrybutu SHARED w czasie linkowania. Na przykład: LINK/SECTION:MYDATA, RWS .... mówi linkerowi że sekcja nazwana MYDATA powinna być odczytywana, zapisywana i współdzielona. Domyślnie segmenty danych Borland C++ mają atrybut wspódzielenia
0x20000000	Sekcja ta jest wykonywalna. Flaga ta jest zazwyczaj ustawiana kiedy ustawiona jest flaga Contains Code (0x00000020)
0x40000000	Sekcja jest do odczytywania. Flaga ta jest prawie zawsze ustawiona dla sekcji w plikach EXE
0x80000000	Sekcja jest do zapisywania. Jeśli ta flaga nie jest ustawiona w sekcji EXE, loader powinien oznaczyć strony odwzorowane w pamięci jako tylko do odczytu lub tylko do wykonania. Zazwyczaj sekcje z tym atrybutem to .data i .bss

Odpowiednik IMAGE_SECTION_HEADER w formacie OS/2 LX nie wskazuje bezpośrednio gdzie może się znajdować sekcja kodu lub danych w pliku. Zamiast tego plik OS/2 LX zawiera stronę przekształcenia tablicowego, która określa atrybuty i położenie w pliku określonego zakresu strony wewnątrz sekcji.. Format PE gwarantuje ,że sekcja danych będzie przechowywana sąsiadująco w pliku. Z tych dwóch formatów, metoda LX może wykazać się większą elastycznością, ale styl PE jest zancząco prostszy i łatwiejszy do pracy. Inną zmianą w formacie PE w stosunku do formatu NE jest to ,że położenie elementów jest przechowywane jako proste offsety DWORD. W formacie NE, położenie prawie wszystkiego było przechowywane jako wartość sektora. Aby znaleźć realny offset pliku, musisz najpierw wyszukać rozmiar jednostki wyrównania w nagłówku NE, i skonwertować go do rozmiaru sektora (zwykle 16lub 512 bajtów). Potem trzeba pomnożyć rozmiar sektora przez określony offset sektora aby uzyskać aktualny offset pliku. Jeśli przypadkiem coś nie jest przechowane jako offset sektora w pliku NE, prawdopodobnie jes przechowywane jako relatywny offset do nagłówka NE. Ponieważ nagłówek NE nie jest na początku pliku, musisz przereciągnąć offset pliku nagłówka NE w kodzie. W przeciwieństwie, pliki PE określają położenie różnych elementów przez zastosowanie prostego offsetu relatywnego tam

gdzie plik był odwzorowany w pamięci. Ogólnie rzecz biorąc, format PE jest dużo łatwiejszy do pracy niż formaty NE , LX czy LE (zakładając ,że możesz używać plików odwzorowanych w pamięci)

10.7 Najczęściej spotykane sekcje

Ponieważ wiemy już co nieco sekcjach i ich położeniach, możey omówić najczęstsze sekcje jakie znajdujemy w plikach EXE i OBJ. Chociaż lista tych sekcji nie oznacza ,że jest kompletnba, ale są to sekcje z jakimi spotykasz się najczęściej. Sekcje są przedstawiane w kolejności ich ważności i częstotliwości ich występowania.

Sekcja .text
Sekcja .text jest tam gdzie kończy się cały kod ogólnego przeznaczenia emiotowany przez kompilator lub asembler. Ponieważ pliki PE uruchamiane są w 32 bitowym trybie i nie są ograniczone do segmentów 16 bitowych, nie ma powodu dzielenia kodu na oddzielne pliki źródłowe w oddzielne sekcje. Zamiast tego , linker łączy wszystkie sekcje .text z różnych OBJ w jedną dużą sekcję .text w EXE. Jeśli używasz kompilatora Borland C++ emitujące on kod do segmentu nazwanego CODE. Zatem pliki PE tworzone przez Borland C++ mają sekcję nazwaną CODE zamiast sekcja .text. Byłem zaskoczony kiedy odkryłem ,że był dodatkowy kod w sekcji .text poza tym jaki stworzyłem kompilatorem albo użyłem z biblioteki. W pliku PE, kiedy wywołasz funkcję w innym module (np. GetMessage() z USER32.DLL), instrukcja CALL wyemitowana przez kompilator nie przeniosła sterowania bezpośrednio do funkcji w DLL'u. Zamiast tego instrukcja wywołania przenosi sterowanie do instrukcji JMP DWORD PTR [XXXXXXXX], która również jest w sekcji .text. Instrukcja JMP skacze pod adres przechowywany w DWORD w sekcji .idata. Sekcja .idata DWORD zawiera rzeczywisty adres punktu wejścia funkcji szystemu operacyjnego. Po chwili zrozumiałem dlaczego wywołania DLL'i implementowane są w ten sposób. Przez przekazywanie wszystkich wywołań do danego DLL'a przez jedno położenie, nie ma dalszej pottzeby aby loader łatał każdą instrukcję, która wywołuje DLL. Każdy loader PE musi wstawić poprawny adres funkcji docelowej do DWORD'a w sekcji .idata. Żadna instrukcja CALL nie musi być łatana. Jest to wyraźnie inne niż pliki NE, gdzie każy segment zawiera listę poprawek które muszą być zastosowane do tego segmentu. Jeśli segment wywołuje daną funkcję DLL 20 razy, loader musi skopiować adres tej funkcji do danego segmentu 20 razy. Minusem metody PE jest to ,że nie można zainicjować zmiennej z prawdziwym adresem funkcji DLL. Na przykład przemyślmy taką sytuację:

```
FARPROC pfnGetMessage = GetMessage;
```

wstawienie adresu GetMessage do zmiennej pfnGetMessage. W Win16 to działa, ale w Win32 już nie. W Win32 zmienna pfnGetMessage kończy przechowywanie adresu JMP DWORD PTR [XXXXXXXX] w sekcji .text, którą omowiłem wcześniej. Jeślic chcielibyśmy wywołać przez funkcję wskaźnik, powinno to zadziałać jak oczekujemy. Jeśli chcemy odczytać bajty na początku GetMessage(), będziemy mieli mniej szczęścia. Po Visual C++ 2/0 wprowadzono nowy zwrot przy wywołaniu funkcji imoprtowych. Jeśli spojrzymy w nagłówki plików systemu z Visual C++ 2.0 (np. WINBASE.H) zobaczymy inne niż w Visual C++ 1.0 nagłówki. W Visual C++ 2.0, prototypy funkcji systemu operacyjnego w systemie DLL'i obejmuje _declspec(dllimport) jako część ich definicji. _declspec(dllimport) okazuje się ,że ma ciekawe efekty przy wywoływaniu funkcji importowych. Kiedy wywołujemy prototypowane funkcje importowane z _declspec(dllimport), kompilator nie generuje wywołania do instrukcji JMP DWORD PTR [XXXXXXXX] gdzieśkolwiek w module. Zamiast tego, kompilator generuje wywołanie funkcji jako CALL DWORD PTR [XXXXXXXX].

Sekcja .data

Podobnie jak .text, jest to domyślna sekcja dla kodu, sekcja .data jest tam gdzie idą zainicjalizowane dane. Dane zainicjalizowane składają się z globalnych i statycznych zmiennych, które są inicjalizowane w czasie kompilacji. Obejmuje również łańcuchy literalne (np. Łańcuch "Hello World" w programie C/C++). Linker łączy wszystkie sekcje .data z plików OBJ i LIB do jednej sekcji .data w EXE. Zmienne lokalne są umieszczone ma stosie wątku i nie zabiera miejsca w sekcjach .data i .bss.

Sekcja DATA
Borland C++ używa nazwy DATA dla swoich domyślnych sekcji danych. Jest to odpowiednik sekcji .data kompilatora Microsoftu.

Sekcja .bss
Sekcja .bss jest tam gdzie są przechowywane niezainicjowane statyczne i globalne zmienne. Linker łączy wszystkie sekcje .bss w plikach OBJ i LIB do jednej sekcji .bss w EXE. W tablicy sekcji, pole RawDataOffset dla sekcji .bss jest ustawione na 0, wskazując ,że ta sekcja nie obejmuje dowolnej przestrzeni w tym pliku. TLINK32 nie emituje sekcji .bss. Zamiast tego, rozszerza rozmiar wirtualny sekcji DATA do stanowiącej część niezainicjowanych danych.

Sekcja .CRT
Sekcja .CRT jest inną zainicjowaną sekcją danych używany przez biblioteki czasu wykonania Microsoft C/C++ (stąd nazwa .CRT). Dane w tej sekcji są używane dla rzeczy takich jak wywołanie konstruktorów klas statycznych C++ zanim main lub WinMain zostanie wywołane.

Sekcja .rsrc
Sekcja .rsrc zawiera zasoby dla tego modułu. W czasach NT, plik .RES dają dane wyjściowe 16 bitowego RC.EXE nie były w formacie jaki linker Microsoftu mógł zrozumieć. Program CVTRES konwertował te pliki .RES do formatu COFF OBJ, umieszczając dane zasobów do sekcji .rsrc wewnątrz OBJ. Linker może potem potraktować zasoby OBJ jaki inny OBJ do połączenia, co oznacza ,że linker nie musi "znać" niczego specjalnego o zasobach. Najnowsze linkery z Microsoft przetwarzają pliki .RES bezpośrednio.

Sekcja .idata
Sekcja .idata zawiera informacje o funkcjach (i danych), które moduł importuje z innych DLL'i. Sekcja ta jest odpowiednikiem modułu tablicy refernecji pliku NE. Kluczowa różnica jest taka ,że każda funkca , jaką importuje plik PE jest specjalnie wylistowany w tej sekcji. Aby znaleźć odpowiednik informacji w pliku NE, musisz przekopać się przez relokacje na końcu surowych danych dla każdego z segmentów.

Sekcja .edata
Sekcja .edata jest to lista funkcji i danych jakie eksportuje plik PE dla stosowania przez inne moduły.Jej odpowiednik NE jest połączeniem wejść tablicy, rezydentnych nazw tablicy i nierezydentnych nazw tablicy. W przeciwieństwie do Win16,rzadko istnieje powód aby eksportować cokolwiek z pliku EXE, więc zazwyczaj widać tylko sekcję .edata w DLL'ach. Wyjatkiem są EXE tworzone przez Borland C++, który pozwala pojawiać się funkcji exportowej (_GetExceptDLLInfo) dla wewnętrznego zastosowania przez bibliotekę czasu wykonania. Kiedy używamy narzędzi Microsoft, dane w sekcji .edata przechodzi do pliku PE poprzez plik .Exp. Linker nie generuje tej informacji sam. Zamiast tego, polega na menadżerze biblioteki (LIN32) dla skanowania plików OBJ i tworzy plik .EXP, któy linker dodaje do swojej listy modułów do połączenia. Tak, to prawda! Te dokuczliwe pliki .EXP są rzeczywiście plikami OBJ z innym rozszerzeniem.

Sekcja .reloc

Sekcja .reloc przetrzymuje tablicę bazowych relokacji. Bazowa relokacja jest poprawką do instrukcji lub wartości zaincjowanej zmiennej; EXE lub DLL potrzebują tej poprawki jeśli loader nie może załadować pliku pod adresem gdzie linker założył ,że będą. Jeśli loader nie może załadować obrazu pod preferowany przez loader adres bazowy, loader ignorujw informację relokacji w tej sekcji. Jeśli chcesz zaryzykować i masz nadzieję ,że loader moze zasze załadować obraz pod załozony adres bazowy możesz użyć opcji /FIXED aby powiedzieć linkerowi aby usunął tą informację. Chociaż może zaoszczędzić miejce w pliku wykonywalnym, możesz również spowodować ,że plik wykonywalny nie będzie działał na platformie Win32. Na przykład,powiedzmy ,że budujesz EXE dla NT a bazą dla EXE to 0x10000. Jeśli powiesz linkerowi tą relokację. EXE nie będzie uruchamiał się pod Windows 95 gdzie adres 0x10000 nie jest dostępny.Ważne do odnotowania jest to ,że instrukcje JMP i CALL generowane przez kompilator używa offsetów relatywnych do instrukcji, zamiast offsetu rzeczywistego w 32 bitowym segmencie. Jeśli obraz potrzebuje być załadowany gdziekowliek indziej niż zakłada adres bazowy linker, ta instrukcja nie musi być zmieniana, ponieważ używają one relatywnego adresowania. W wyniku tego, nie ma zbyt wielu relokacji o jakich możesz myśleć. Relokacje są zazwyczaj potrzebne tylko dla instrukcji, które używają 32 bitowego offsetu dla tych samych danych. Na przykład powiedzmy ,ze masz deklaracje globalnej zmiennej:

```
int i;
int *ptr = &i;
```

Jeśli linker zakłada obraz bazowy 0x10000, adres zmiennej i zakończy się czymś takim 0x12004. W tej pamięci używanej do przetrzymywania wskaźnika ptr, linker wypisze 0x12004, ponieważ jest to adres zmiennej i. Jeśli loader (z jakichś powodów) zdecydował załadować plik pod adresem bazowym 0x70000, adres i będzie to 0x72004. Jednak, wartość wcześniej zainicjowana zmiennej ptr będzie wtedy niepoprawna ponieważ i to teraz 0x60000 bajtów wyżej w pamięci. Sekcja .reloc jest listą miejsc w obrazie gdzie różnice między linkerem z założonym adresem a rzeczywistym adresem ładowania.

Sekcja .tls

Kiedy używasz dyrektywy kompilatora "_declspec(thread)", dana jaką zdefiniowałeś nie idzie do sekcji .data lub .bss. Raczej kopia jej kończy się w sekcji .tls. Sekcja .tls dziedziczy swoją nazwę z terminy thread local storage i jest powiązany z rodziną funkcji TlsAlloc(). Każdy wątek może mieć swój własny zbiór wartości danych statycznych, mimo ,że kod , który używa dane bez względu na to jaki wątek jest wykonywany. Rozważmy program, który ma kilka wątków działających w tym samym zadaniu, i w ten sposób wykonywane przez ten sam kod. Jeśli zadeklarujemy zmienną thread local storage, np:

```
__declspec (thread) int i = 0:
```

//to jest deklaracja zmiennej globalnej

każdy wątek będzie miał przejrzystą własną kopię zmiennej i Możliwe jest również wyraźne zapytanie i użycie thread local storage w czasie wykonywania przez zastosowanie funkcji TlsAlloc, TlsSetValue i TlsGetValue. W większości przypadków, dużo łatwiej zadeklarować własne dane w programie z _declspec (thread) niż alokować pamięć na podstawowym wątku i przechowywać wskaźnik do pamięci w dopasowanym TlsAlloc(). Ważne do odnotowania jest to ,że rzeczywiste bloki pamięci wątku nie są przechowywane w sekcji .tls w czasie wykonania. To znaczy, kiedy przełączamy wątki, menadżer pamięci nie zmienia strony fizycznej pamięci, która jest odwzorowana do modułu sekcji .tls. Zamiast tego , sekcja .tls jest jedynie daną używaną do zainicjowania aktualnego bloku danych wątku.Inicjacja obszaru danych wątku jest wspólnym wysiłkiem między systemem operacyjnym a bibliotekami czau wykonania klompilatora. Jest wymagana dodatkowa dana – katalog TLS -- ,któa jest przechowywana w sekcji .rdata

Sekcja .rdata
Sekcja .rdata jest używana dla przynajmniej czterech rzeczy. Po pierwsze, w EXE tworzonym przez Microsoft Link, sekcja .rdata przechowuje katalog debuggowania. W EXE z TLINK32, katalog debuggowania jest w sekcji nazwanej .debug. Katalog debuggowania jest strukturą tablicy IMAGE_DEBUG_DIRECTORY. Struktury te przechowują informacje o typie, rozmiarze i położeniu różnych typów informacji debuggowania przechowywanych w tym pliku. Mogą pojawić się trzy główne typy informacji debuggowania: CodeView, COFF i FPO. Katalog uruchomieniowy nie jest koniecznie znajdowany na początku sekcji .rdata. Zamiast tego,aby znaleźć początek katalogu uruchomieniowego, musisz użyć RVA w siódmy wejściu (IMAGE_DIRECTORY_ENTRY_DEBUG) katalogu danych. (Katalog danych jest na końcu nagłówka PE części pliku) Aby określić liczbę wejść w katalogu uruchomieniowym Microsoft Link, podziel rozmiar katalogu uruchomieniowego (znajdującego się w polu rozmiaru wejścia katalogu danych) przez rozmiar struktury IMAGE_DEBUG_DIRECTORY. W przeciwieństwie, TLINK32 emituje aktualnie zliczane katalogi uruchomieniowe w polu rozmiaru, nie całkowitą w bajtach. Druga użyteeczna część sekcji .rdata to łańcuch opisowy. Jeśli określis wejście DSCRIPTION w pliku .DEF programu, pojawi się określony łańcuch opisowy w sekcji .rdata. W foramcie NE, łańcuch opisowy jest zawsze pierwszym wejściem nierezydentnej tablicy nazw. Łańcuch opisowy jest zaplanowany do przetrzymywania użytecznych łańcuchów tekstowych opisujących plik. Niestety, nie znalazłem łatwego sposobu znajdowania go. Widziałem pliki PE, które miały łańcuch opisowy przed katalogiem uruchomieniowym, i inne pliki, które miały go po katalogu uruchomieniowym. Nie jestem świadom spójnych metod znajdowania łańcucha opisowego (lub nawet do określenia czy wogóle jest obecny). Trzecim zastosowaniem sekcji .rdata jest użycie GUID'ów w programowaniu OLE. Bilioteka importowa UUID.LIB zawiera zbiór 16 bajtowych GUID'ów, które są używane dla rzeczy takich jak interfejs ID. Te GUID'y kończą w EXE lub DLL sekcję .rdata. Końcowym zastosowaniem sekcji .rdata jest miejsce umieszczania katalogu TLS (Thread Local Storage) Katalog TLS jest specjalną strukturą danych używan a przez bibliotekę czasu wykonania kompilatora dla transparentnego dostarczania thread local storage dla zmiennych zadeklarowanych w kodzie programu.

10.8 Pliki importowania PE
Wcześniej widzieliśmy jak funkcje wywołują zewnętrzne DLL'e nie wywołując bezpośrednio DLL'i. Zamiast tego, instrukcja CALL skacze do instrukcji JMP DWORD PTR [XXXXXXXX] gdzieś w sekcji .text pliku wykonywalnego (lub sekcji .icode jeśli używamy Boorland C++ 4.0) Alternatywnie, if _ declspec(dllimport) było używane w Visual C++, funkcją wywołującą stało się "CALL DWORD PTR [XXXXXXXX]". W takim przypadku, adres którego szuka instrukcja JMP lub CALL jest przechowywana w sekcji .idata. Instrukcja JMP lub CALL prznosi sterowanie do tego adresu, któy jest adresem docelowym. Przed jego załadowaniem do pamięci, informacja przechowywana w sekcji .idata pliku PE zawierająca informacji koniecznych loaderowi określa adres funkcji docelowej i łata je do obrazu wykonywalnego. Po załadowaniu sekcji .idata , zawiera ona wskaźniki do funkcji, które importuje EXE/DLL. Zwróć uwagę ,że wszystkie tablice i struktury jakie omawiałem w tej sekcji są zawarte w sekcji .idata. Sekcja .idata (lub tablica importów) zaczyna się od tablicy IMAGE_IMPORT_DESCRIPTOR. Jest jedna IMAGE_IMPORT_DESCRIPTOR dla każdego DLL'a który PE bezwzględnie łączy. Żadna liczba nie jest konieczna dla wskazywania lioczby struktur w tej tablicy. Zamiast tego, ostatni element tablicy jest wskazywany przez końcowy IMAGE_IMPORT_DESCRIPTOR, który ma pola wypełnione NULL'ami. Format IMAGE_IMPORT_DESCRIPTOR jest następujący:

DWORD Charachteristics/OriginalFirstThunk
Pole to jest offsetem (RVA) tablicy DWORD'ów. Każdy z tych DWORD'ów jest właściwie unią IMAGE_THUNK_DATA. Każdy DWORD IMAGE_THUNK_DATA odpowiada jednej funckji importowanej przez ten EXE/DLL. Jeśli uruchomisz narzędzie BIND, tablica DWORD'ów jest

nieruszana podczas gdy tablica FirstThunk DWORD jest modyfikowana.

DWORD TimeDateStamp
Znaczniki czasu / daty wskazuje kiedy plik został stworzony. Pole to zazwycaj zawiera 0. Jednak, narzędzie BIND Microsoftu aktualizuje to pole znacznikiem czasu / daty DLL'a odnosząc się do IMAGE_IMPORT_DESCRIPTOR

DWORD ForwardedChain
Pole to odnosi się do przekierowania , które wywołuje jedne DLL przekierowując referencje do jednej z jego funkcji do innego DLL'a. Na przykład, w Windows NT, KERNEL32.DLL przekierowuje jakąś z jego funkcji importowych do NTDLL.DLL. Aplikacja może sądzić ,ze jest to wywołanie funkcji w KERNEL32.DLL, ale ale w rzeczywistości skończy wywołanie do NTDLL.DLL. Pole to zawiera indeks do tablicy FirstThunk. Funkcja indeksowana przez to pole będzie przekierowywana do innego DLL'a. Niestety, format w jakim funkcja jest przekierowywana jest dość skąpo opisana przez dokumentację Microsoft.

DWORD Name
Jest to RVA do łańcucha ASCII zakoćzonego zerem zawierającego nazwę imporotwaną DLL'i (np. KERNEL32.DLL lub USER32.DLL)

PIMAGE_THUNK_DATA FirstThunk
Pole to jest offsetem (RVA) do tablicy IMAGE_THUNK_DATA DWORD. W większości przypadków, DWORD jest interpretowany jako wskaźnik do struktury IMAGE_IMPORT_BY_NAME. Jednak możliwy jest również import funkcji przez wartość porządkową. Ważną częścią IMAGE_IMPORT_DESCRIPTOR są nazwy importowych DLL i dwie tablice IMAGE_THUNK_DATA DWORD. Kązdy IMAGE_THUNK_DATA DWORD odpowiada jednej funkcji importu .W pliku EXE, dwie tablice (wskazywane przez pola Characteristic i FirstThunk) uruchamiają równolegle jedna drugą, a kończone są przez wejście wskaźnika NULL na końcu każej tablicy. Dzlaczego są dwie równoległe tablice wskaźników do struktur IMAGE_THUNK_DATA? Pierwsza tablica (wskazywana przez pole Characteristic) jest opuszczona i nigdy nie jest modyfikowana. Czasami jest nazywana tablicą nazw podpowiedzi. Druga tablica (wskazywana przez pole FirstThunk w IMAGE_IMPORT_DESCRIPTOR) jest nadpisywana przez loader PE. Loader iteruje przez każdą IMAGE_THUNK_DATA i znajduje adres funkcji, która odnosi się do niej Loader nadpisuje potem IMAGE_THUNK_DATA DWORD adresem importowanej funkcji. Wcześniej wspomnieliśmy ,ze funkcje CALL i DLL przechodzą przez "JMP DWORD PTR [XXXXXXXX]. Część [XXXXXXXX] odnosi się do jednego z wejść w tablicy FirstThunk. Ponieważ tablica IMAGE_THUNK_DATA, któe jest nadpisywana przez loader w końcu przechowuje adresy wszystkich funkcji imoprtowanych, nazywana jest "Import Address Table" Ponieważ tablica adresów importowanych jest zazwyczaj w sekcji zapisywalnej, relatywnie łatwo jest przechwycić wywołanie EXE , lub stworzyć DLL dla innego DLL'a. Po prostu łatasz właściwe wejscie tablicy adresów importu aby wskazywał żądaną funkcję do przechwycenia. Nie ma potrzeby modyfikowania kodu albo wywołującego albo wywoływanego. Ta możliwość może być bardzo użyteczna.Interesujące jest t o,że w plikach PE tworzonych przez Microsoft, tablica importu nie jest całkowicie syntetyzowany przez linker. Zamiast tego, wszystkie fragmenty konieczne do wywołania funkcji w innym DLL'u rezyduje w bibliotece importu. Kiedy łączysz DLL, menadżer biblioteki (LIB.EXE) skanuje pliki OBJ jakie są łączone i tworzy bibliotekę importu. Bibliteka ta jest inna niż biblioteki importu używane przrez 16 bitowe linkery NE. Biblioteka improtu, którą tworzy 32 bitowy LIB ma sekcję .text i kilka sekcji .idata$ .Sekcja .text w bibliotece importu zawiera JMP DWORD PTR [XXXXXXXX]. Ten fragment ma nazwę przechowywaną w tablicy symboli OBJ. Nazwa tego symbolu jest identyczna do nazwy funkcji eksportowanej przez DLL (np. _ DIspatchMessage@4) .Jedna z sekcji .idata$ w bibliotece importu zawiera DWORD. Inna sekcja .idata$ ma przestrzeń dla "podpowiedzi porządkowych" po których

następuje nazwa importowanej funkcji. Te dwa pola tworzą strukturę IMAGE_IMPORT_BY_NAME. Kiedy później łączysz plik PE, które używają biblioteki importu, sekcje biblioteki importu są dodawane do listy sekcji z twoich OBJ, które linker musi przetwarzać. Ponieważ fragment w biblitece importu ma taką samą nazwę jak funkcje importowane, linker sądzi ,że ten fragment jest rzeczywiście funkcją importowaną, i przygotowuje wywołanie importowanej funckji wskazującej ten fragment. Kawalek ten w bibliotece importu jest szczególnie widoczny jako funkcja iportu. Poza dostarczaniem części fragmnetu kodu funkcji importu, biblioteka importu dostarcza fragmentu sekcji .idata PE (lub tabliocy importu). Te fragmenty pochodzące z różnych sekcji .idata$, które bibliotekarz wstawia do biblioteki importu.. W skrócie, linker nie zna rzeczywiście różnic między funkcjami imporotwanymi a funkcjami które pojawiają się w innym pliku OBJ.

10.9 Pliki eksportowane PE

Przeciwiestwem importowanych funkcji jest eksportowanie funkcji dla zastosowania przez EXE lub inne DLL'e. Plik APE przechowuje informacje o jego funkcjach eksportowanych w sekcji .edata. Generalnie, Microsoft LINK – tworzy pliki PE nie eksportujące niczego, więc nie mają sekcj .edata. EXE'ki TLINK32, z drugiej strony, zazwyczaj eksportują jedne symbol, wię muszą mieć sekcję .edata. Większość DLL'i eksportuje funkcje i ma sekcję .edata. Podstawowym komponentem sekcji .edata (tablica eksportu) są tablicami nazw funkcji, adresy punktów wejścia i wartości prządkowe eksportu. W pliku NE, odpowiednikiem tablicy eksportu są wejścia tablicy, rezydentne nazwy tablicy i nierezydentne nazwy tablicy. W pliku NE, tablice te są przechowywane jako częśc nagłówka NE zamiast w segmentach lub zasobach. Na początku sekcji .edata znajduje się struktura IMAGE_EXPORT_DIRECTORY. Struktura ta następuje po danych wskazywanych przez pola w strukturze IMAGE_EXPORT_DIRECTORY. IMAGE_EXPORT_DIRECTORY wygląda następująco:

DWORD Characteristics

Pole to poajwia się jako nieużywane i zawsze jest ustawione na 0

DWORD TimeDateStamp

Znacznik czasu / daty wskazuje kiedy plik został stworzony

WORD MajorVersion
WORD MinorVersion

Pola te są nieużywane i są usatiwane na 0

DWORD Name

RVA łańcucha ASCII z nazwą tego DLL'a (np. MYDLL.DLL)

DWORD Base

Startowy numer porządkowy eksportu dla funkcji eksportowanej przez ten moduł. Na przykład, jeśli plik eksportuje funkcję z wartością porządkową 10, 11 i 12, pole to będzie zawierać 10.

DWORD NumberOfFunctions

Liczba elementów w tablicy AddressOfFuntion. Wartość ta jest również liczbą funkcji eksportowanych przez ten moduł. Zazwyczaj wartość ta jest taka sama jak pola NumberOfNames, ale mogą się różnić.

DWORD NumberOfNames

Liczba elementów w tablicy AddressOfNames. Wartośćta zawiera liczbę funkcji eksportowanych przez nazwę, która zazwyczaj (ale nie zawsze) pasuje z całkowitą liczbą eksportowanych funkcji.

PDWORD *AddressOfFunctions
Pole to jest RVA i wskazuje na tablicę adresów funkcji. Adresy funkcji są punktami wejści RVA dla każdej eksportowanej funkcji w tym module.

PDWORD *AddressOfNames
Pole to to RVA i wskazuje na tablicę wskaźników łańcucha. Łańcuch zawiera nazwy funkcji eksportowanych przez nazwę z tego modułu.

PDWORD *NumberOfNamesOrdinals
Pole to jest RVA i wskazuje na tablicę WORD'ów. WORD'y są zasadniczo eksportem porządkowym wszystkich funkcji eksportowanych przez nazwę z tego modułu. Jednak nie zapomnij dodać porządkowej liczby startowej określonej w polu Base. Rozkład tablicy eksportu jest do pewnego stopnia nieparzysty. Jak wspomniałem wcześniej, wymagania dla eksportowania funckji są adresem i porządkowym adresem. Opcjonalnie, jeśli eksportujesz funkcję przez nazwę, będzie to nazwa funkcji. Twócy formatu PE nie wstwili wszyskich trzech tych pozycji do struktury i dlatego nie mamy tablicy tych struktur .Zamaist tego musimy szukać różnych kawałków w trzech oddzielnych tablicach. Najważniejsza z tych tablic wskazywana przez IMAGE_EXPORT_DIRECTORY jest tablicą wskazywaną przez pole AddresOfFunction. Jest to balica DWORD'ów, każdy DWORD zawiera adres (RVA) importowanej funkcji. Liczba porządkowa eksportu dla każej eksportowanej funkcji odpowiada jej pozycji w tablicy. Na przykład (zakładając start od 1), adres funkcji z numerem porządkowym 1 będzie miała swój adres w pierwszym elemencie tej tablicy. Funkcja z liczbą porządkową eksportu 2 ma swój adres w drugim elemencie tablicy itd. Są dwie ważne rzezy do zapamiętania o tablicy AddressOfFuntions. Pierwsza to to ,że numer porządkowy eksportu musi być podawana przez wartość w polu Base IMAGE_EXPORT_DIRECTORY. Jeśli pole Base zawiera wartość 10, wtedy pierwszy DWORD w tablicy AddresOfFunctions odpowiada wartości porządkowej eklsportu 10, drugie wejście to 11 itd. Druga rzecz to to ,że liczba porządkowa eksportu może mieć luki. Powiedzmy ,że jawnie wyeksportowałeś dwie funckje w DLL, z wartościami porządkowymi 1 i 3. Chociaż wyeksportowałeś tylko dwie funkcje. Tablica AddresOfFunctions musi zawierać trzy elementy. Wejście w tablicy , które nie odpowiada wyeksportowanej funkcji zawiera wartość 0. Kiedy loader Win32 przygotował wywołanie funkcji, która jest importowana przez wartość porządkową, ma trochę pracy do wykonania. Loader po prostu używa wartości porządkowej funkcji jako indeksu do docelowego modułu tablicy AddresOfFunctions. Oczywiście, loader również musi wziaść pod uwagę, że najniższą wartością porządkową eksportu nie może być 1, i musi modyfikować swoje indeksy właściwie. Najczęściej Win32 EXE i DLL importują funkcje przez nazwę niż wartość. Jest tak kiedy dwie tablice wskazują na strukturę IMAGE_EXPORT_DIRECTORY wchodzącą w grę. Tablice AddressOfNames i AddressOfNameOrdinals istnieją aby zezwalać loaderowi na szybkie znajdowanie wartości porządkowych eksportu odpowiadające danej nazwie funkcji. Tablice AddressOfNames i AddressOfNameOrdinals obie zawierają taką samą liczbę elementów (podan przez pole NumbersOfNames IMAGE_EXPORT_DIRECTORY) Tablica AddressOfNames jest tablicą wskaźników do nazw funkcji, a AddressOfNameOrdinals jest tablicą indeksów do tablicy AddressOfFunction. Zoabczmy jak loader Win32 przygotowuje wywołanie funkcji, która jest importowana przez nazwę. Najpierw, loader wyszukuje łańcuchy wskazane w tablicy AddressOfNames. Powiedzmy ,że znajduje ten łańcuch, który wyszukał w trzecim elemencie. Następnie loader użyje indeksu jaki znalazł szukając odpowiedniego elementu w tablicy AddressOfNameOrdinals (w tym przypadku trzeci element) Tablica ta jest zbiorem WORD'ów, a każdy WORD działa jako indesk do tablicy AddressOfFunctions .Ostatnim krokiem jest pobranie tej wartości w tablicy AddressOfNameOrdinals i użycie jej jako indeksu do tablicy AddressOfFunctions. Nawiasem mówiąc, jeśli zrucisz eksorty z systemu DLL'i (np. KERNEL32.DLL i USER32.DLL), zobaczysz ,że w wielu przypadkach dwie funkcje różnią sie tylko jednym znakiem na końcu nazwy, np. CreateWindowsExA i CreateWindowsExW. Funkcja

zakończona A to funkcja kompatybilna z ASCII (lub ANSI); ta zakończona W to wersja Unicode funkcji. W twoim kodzie, nie określasz wyraźnie jaką funkcję wywołujesz .Zamiast tego,właściwa funkcja jest wybieraza w WINDOWS.H przez preprocesor #ifdef. Oto przykład takiej pracy:

```
#ifdef UNICODE
#define DefWindowProc DefWindowProcW
#else
#define DefWindowProc DefWindowProcA
#endif // !UNICODE
```

IV. Podstawowe pojęcia z assemblacji

11. Rejestry

Możemy potraktować rejestry jako zmienne wewnątrz CPU (Central Processing Unit). Oto kilka rejestrów w komputerze:

AX,BX,CX,DX,CS,DS,ES,SS,SP,BP,SI,DI,Flags i IP

Wszystkie one są 16 bitowe. Możesz potraktować je jak gdyby były one zmienną słowa (lub liczbą całkowitą bez znaku). Jednakże każdy rejestr ma swoje wsłasne zastosowanie.

- AX,BX,CX i DX są rejestrami ogólnego przeznaczenia. Można im przypisać dowolną wartość jaką chces. Oczywiście musisz zmodyfikować je według potrzeb
    - AX zazwyczaj jest nazywany akumulatorem. Większość operacji arytmetycznych jest wykonywanych poprzez AX. Czasami inne rejestry ogólnego przeznaczenia mogą być również wykorzystywane w operacjach arytmetycznych, taki jak DX
    - Rejestr BX jest zwykle nazywany rejestrem bazowym. Jego podstawowym zastosowaniem jest wykonywanie operacji tablicowych. BX jest zazwyczaj stosowany z innymi rejestrami, zwykle z SP wskazującym stos
    - CX jest powszechnie nazywany rejestrem licznika. Rejestr ten używany do operacji zliczania. To dlatego komputer może wykonywać pętle.
    - Rejestr DX jest rejestrem danych

- Rejestry CS, DS, ES i SS są nazywane rejestrami segmentowymi. Możesz używać ich tylko w poprawny sposób.

- CS jest nazywany rjestrem segmentu kodu. Wskazuje segment uruchomionego programu. Nie można modyfikować CS bezpośrednio
- Ds jest rejestrem segmentu danych. Wskazuje segment danych używanych przez uruchomiony program .Możesz wskazywać to gdziekowliek chcesz tak długo jak zawiera żądane dane.
- ES jest nazywany rejestrem dodatkowego segmentu. Zazwyczaj jest używany z DI i użyany jest do spraw wskaźników. Para DS:SI i ES:SI są powszwechnie używane do operacji łańcuchowych
- SS jest nazywany rejestrem segmentu stosu. Wskazuje segment stosu

- Rejestr SI i DI są nazywane rejestrami indeksowymi. Zwykle są używane do przetwarzania tablic lub łańcuchów

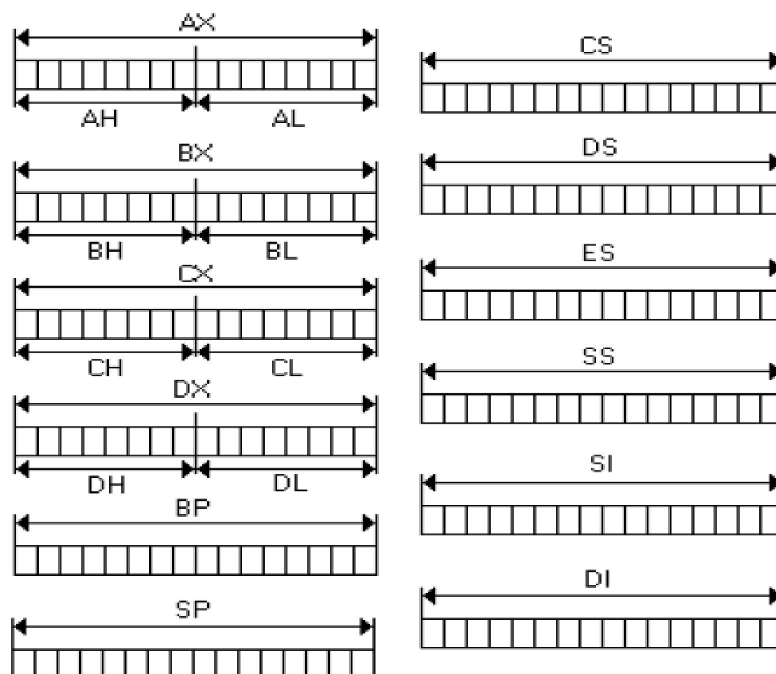
– SI jest nazywany indeksem źródłowym a DI indeksem przeznaczenia. Jak wskazuje nazwa. SI zawsze wskazuje tablicę źródłową a DI zawsze wskazuje miejsce przeznaczenia. Zwykle jest używany do przesuwania bloku danych, takich jak rekordy (lub struktury) i tablice. Rejestry te są stosowane w parzee z DS i ES

- Rejestry BP, SP i IP są nazywane rejestrami wskaźnika

– BP jest wskaźnikiem bazowym, SP wskaźnikiem stosu a IP instrukcją wskaźnika. Zazwyczaj BP jest używany dla zachowania przestrzeni dla zastosowania zmiennych lokalnych. SP jest używany do wskazywania bieżącego stosu. Chociaż SP może być łatwo modyfikowany, musisz być ostrożny. Zrobienie czegoś złego z tym rejestrem możesz spowodować zawieszenie programu. IP oznacza bieżący wskaźnik uruchomionego programu. Zawsze jest w parze z CS i nie jest modyfikalny. Tak więc para CS:IP jest wskaźnikiem wskazującym bieżącą instrukcję uruchomionego programu. Nie masz dostępu do CS lub IP bezpośrednio.

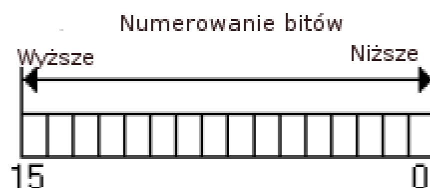
- Rejestr Flags jest używany do przechowywania bieżącego stanu procesora. Ptrzechowuje wartość do której programista nie ma dostępu. Obejmuje to wykrywanie czy ostatnia arytmetyczna operacja dała wynik zero lub może być przepełnionu. Możesz tylko modyfikować flagę ze stosu

Rejestry AX, BX, CX i DX są16 bitowe. Jednak, są one złożone z dwóch mniejszych rejestrów. Na przykład: AX. Wyższe 8 bitów jest nazywane AH a niższe 8 bitów jest nazywane AL. Zarówno AH i AL mogą być dostępne bezpośrednio. Jednak, ponieważrazem składają się na AX. Zmodyfikowanie AH jest modyfikowaniem wyższych 8 bitów AX. Zmodyfikowanie AL jest modyfikwaniem niższych 8 bitów AX



Numerowanie bitów rejestru zaczyna się od niższej części Najniższy bit jest numerowany jako bit 0, najwyższy bit jest numerowany jako bit 15. Więc jest 16 bitów. Dlatego AL zajmuje bity od 0 do

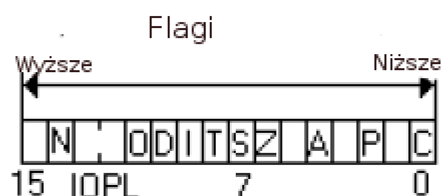
7 AX, AH zajmuje bity od 8 do 15 AX.



Zwróć uwagę ,że od procesorów x386 wprowadzono rozszerzone rejestry. Większośćz tych rejestrów z wyjątkiem rejestrów segmnetowych są poszerzone do 32 bitów. Tak więc rozszerzamy rejestry do EAX, EBX, ECX itd. AX jest tylko niższymi 16 bitami (bit 0 do 15) EAX. BX jest tylko niższymi 16 bitami (bit 0 do 15) EBX itd. Nie ma specjalnego bezpośredniego dostępu do górnych 16 bitów (bit 16 do 31) w rozszerzonym rejestrze. Rejestry segmentowe nie są rozszerzane. Nie ma rejestrów ECS lub EDS.

12. Flags

Flags jest 16 bitowym rejestrem , który zawiera stan procesora. Intel nie dostarcza bezpośedniego dostępu do niego; jest dostępny poprzez stos (poprzez POPF i PUSHF). Jednak, z jakichś powodów, masz potem dostęp do flag używając instrukcji asemblacji SAHF i LAHF. Masz dostęp do każdego atrybutu flagi przez użycie operacji bitowych AND ponieważ każdy stan jest najczęściej przedstawiany przezbit 1. Oto rozkład flag:



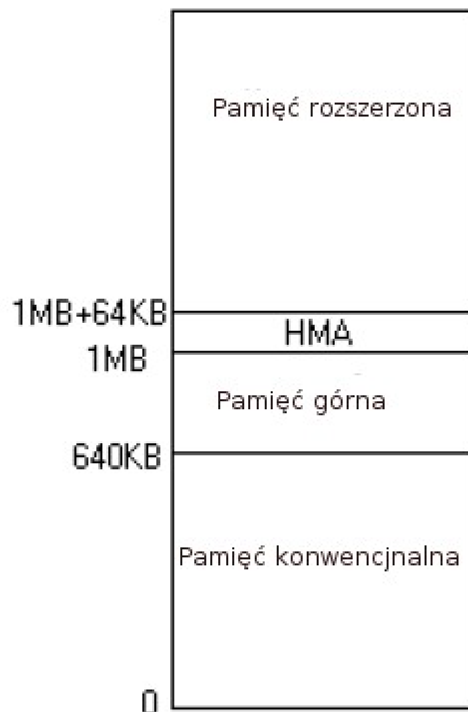
- C oznacza flagę przeniesienia (bit 0). Zmienia się na 1 kiedy ostatnia operacja arytmetyczna, taka jak dodawanie i odejmowanie, ma "przeniesienie" lub "pożyczkę", w przeciwnym razie jest ustawiona na 0. DOS często używa jej dla wskazywania błędów
- P oznacza flagę parzystości (bit 2). Rzadko używana. Będzie ustawiona na 1 jeśli ostatnia operacja (dowolna operacja) w wyniku daje parzystą liczbę bitów 1. Zazwyczaj jest używana w komunikacyjnych sprawach.
- A oznacza flagę pomocniczą (bit 4) Rzadko używana .Ustawiana na 1 w działaniach Binary Coded Decimal (BCD).
- Z oznacza flagę zera (bit 6). Zazwyczaj używana do wykrywania czy ostatnie działanie (dowolne działanie) dało w wyniku zero.
- S oznacza flagę znbaku (bit 7) Często jest używana do wykrywania czy ostatnie działanie dało ujemny wynik. Jest ustawiana na 1 jeśli najwyższy bit (bit 7 w bajcie, lub bit 15 w słowie) ostatniego działania to 1
- T oznacza flagę pułapki (bit 8) Jest używana tylko w debuggerach do włączania funkcji krok – po – kroku.
- I oznacza flagę przerwania (bit 9) Jest używana do przełączania włączania lub wyłączania przerwania. Jeśli bit jest ustawiony (= 1), wtedy przerwania są włączone, w przeciwnym razie są wyłączone. Domyślnie jest włączona.
- D oznacza flagę kierunku (bit 10). Używana dla kierunku opracji łańcuchowych. Jeśli bit jest ustawiony , wtedy wszystkie działania łańcuchowe są wykonywane w tył. W przeciwnym razie w przód. Domyślnie jest ustawiona w przód (= 0).

- O oznacza flagę przepełnienia (bit 11) Jest używana do wykrywania czy ostatania operacja arytmetyczna daje w wyniku przepełnienie czy też nie. Jeśli bit jest ustawiony, wtedy jest przepełnienie.
- IOPL oznacza flagę I/O Privilege Level (bit 12 i 13) Jest używany do oznaczania poziomu uprzywilejowania uruchomionych programów. Jest rzadko używany w w realnym trybie programowania. Flaga ta istnieje w CPU 286 i wyższych.
- N oznacza flagę Nested Task (bit 14) jest flagą w CPU 286 i lepszych. Jest to wykrywanie czy pojawiła się wielozadaniowość (lub wyjątki) Rzadko używana w praktycznym programowaniu.
- Do najczęściej używanych flag należą flagi O,D,I,S,Z i C

Procesory 386 i lepsze mają ulepszone flagi 32 bitowe. Logika jednak pozostaje ta sama.

13. Pamięć

Oczywiście kod programu (kod) jest umieszczony w pamięci. Pamięćjest w rzeczywistości ponumerowana od adresów, zaczynając od 0 ,1, 2 dopóki wszystko nie zostanie odwzorowane. Aby zaadresować dane w pamięci , CPU używa rejestrów. Początkowo CPU miał tylko rejestry 16 bitowe, tak więc maksymalna ilość pamięci. Jaka może być adresowana to 2^{16} = 65536 (64 KB) Jednak, po nadejściu XT, pamięć została rozszerzona do 1 MB. To znaczy, 16 razy większa niż pierwotnie. Niestety, CPU ma rejestry 16 bitowe, które faktycznie nie mogą obsłużyć całej pamięci. Inżynierowie musieli poradzić sobie z tym i wprowadzili technikę nazwaną segmentacją. Oznacza to ,że pamięć jest dzielona praktycznie na kilka obszarów zwanych segmentami.Wraz z nadejściem segmnetacji, pojawiły si,e rejestry segmentowe. Rejestry segmentowe są również 16 bitowe. Ideą segmentacji nie jest podzielenie 1 MB na 16 dokładnych części. Oznacza to ,że rejestry segmentowe są zezwalają na wartości od 0 do 15, i używają tylko 4 bitów. Jeśli numerem segmentu jest 0, wtedy mamy dostęp do pamięci od 0 do 65536. Segment numer 1 pozwala nam na dostęp do pamięci o numerach 16 do 65552. Segment 2 od 32 do 65568 i tak z inkrementacją 16. Jest oczywiste ,że cała 1 MB pamięć jest adresowalna. Dlaczego tak jest?Przez wzgląd na zarządzanie pamięcią systemu operacyjnego. DOS wyrównuje wykonywany kod do najbliższego 16 bajtowego wyrównania. Dostęp do pamięci musi być wykonany przez parę rejestrów. Pierwszym jest rejestr segmentowy a kolejny jest dowolnym rejestrem, zazwyczaj BX, DX, SI lub DI. Para rejestrów zazwyczaj jest zapisywana tak : ES: DI z dwukropkiem między nimi. Para taka jest nazywana parą segment : offset. Tak więc ES:DI oznacza ,że część segmentu jest adresowana przez ES a część offsetowa jest adresowalna przez DI. Jeśli ES zawiera 0 a DI 5, oznacza ,że mamy dostęp do pamięci 5. Jeśli ES:DI = 0001:0005 wtedy rzeczywiście mamy dostęp do rzeczywistego adresu 21 (1*16 + 5 = 21). Tak więc, 0000:0021 i 0001:0005 są w zasadzie tym samym adresem.. Jak prpocesor moze to robić? Para rejestrów segment : offset zawiera adres logiczny. Aktualny adres lub adres absolutny musi być wyliczony z adresu logicznego. Ponieważ przeplot jest zwiększany o 16, wtedy musimy pomnożyć wartośćsegmentu najpierw przez 16 ,potem dodać go do części offsetowej. Zwykle programiści odnoszą się do pamięci 0 do 640 KB jako pamięci konwencjonalnej (dolnej). Obszar powyżej pierwszych 640 KB do 1MB jest nazywana górnym obszarem pamięci (HMA). Powyżej tego punktu mamy rozszerzoną lub powiększoną pamięcią.



Obecnie różnice między rozszerzoną pamięcią a poszerzoną pamięcią nie jest zbyt jasna. W pełni zależy to od sterownika. HIMEM.SYS dostarcza dostępu do rozszerzonej pamięci. EM386.EXE, QEMM, 386MAX, dostarczają dostępu do poszerzonej pamięci. Programiści zazwyczaj wolą pamięćrozszerzoną ponieważ operacje tam wykonywane są szybsze.

14. Stos

Kiedy OS ładuje kod programu do pamięci, określona ilość pamięci jest zarezerwowana aby program uruchamiał się jak należy. Tryb pamięci każego programu zachowuje się inaczej. Jednak jest jedna rzecz: musi być miejsce dla samego kodu, musi by,c, miejsce dla danych i co najważniejsze musi być miejsce na stos. Stos jest jak czasowy obszar do przechowywania rzeczy koniecznych w najbliższej przyszłości (kiedy program jest uruchamiany). Używany jest głównie do przekazywania wartości parametrów do procedur lub funkcji. Czasami, działa również jako czasowa przestrzeń dla alokowania zmiennych lokalnych. Rola stosu jest bardzo ważna. Działą dokładnie jak stos w liście połączonej. Ostatnia pozycja odkładana na stos jest zdejmowana jako pierwsza. Działa tu kocepcja LIFO (Last In First Out). W tej chwili nie musimy znać dokładnego działania stosu. Ważne aby pamiętać o koncepcji LIFO , ale nie jest to lista połączona. Działając ze stosem, potrzeba rezerwacji tak dużo pamięci jak to konieczne dla stosu. Jeśli używamy wielu parametrów w procedurze lub funkcji, musimy zarezerwować duży stos

15. Przerwania

Przerwanie jest tym na co wskazuje nazwa : przerwaniem. Zasadniczo, przerwaniami procesów. Po żądaniu przerwania, procesor zwykle przechowuje tylko CS:IP i stan flag uruchomionego programu, a potem przechodzi do podprogramu przerwania. Po przetworzeniu przerwania , procesor przywraca cały stan poprzedni i przywraca wykonywanie programu. Są trzy rodzaje przerwań: sprzętowe (inne niż CPU), programowe i generowane przez CPU.

Przerwania sprzętowe występują jeśli jedno z urządzeń wewnątrz komputera wymaga pilnego przetwarzania. Opóźnienie procesu może spowodować nieprzewidywalne lub nawet katastrofalne

skutki. Przerwania klawiaturowe są jednym z przykładów. Kiedy naciskasz kalwisz na klawiaturze, generujesz przerwanie. Chip klawiatury zgłasza procesorowi ,że ma znak do wysłania. Możesz sobie wyobrazić jeśli procesor to zignoruje i pójdzie dalej? Klawisz nie zostanie przetworzony.!

Przerwania programowe występują jeśli uruchomiony program wymaga aby program został przerwany i zrobił coś innego. Zwykle jest to oczekiwanie na wprowadzanie danych z klawiatury lub żądania sterowanika graficznego dla zanicjowania ekranu graficznego.

Przerwania generowane przez CPU pojawiają się jeśli procesor wie ,że jest coś złego z kodem uruchomieniowym. Jest to zazwyczaj bezpośrednia ochrona przed zawieszeniem. Jeśli program zawiera instrukcje, których procesor nie zna , procesor przerwie program. Stanie się to również jeśli podzielisz liczbę przez zero (błąd dzielenia przez zero). Przerwania mają wiele zastosowań,i generalnie ułatwiają życie programistą, ponieważ obsługują pewne priorytetowe zdarzenia.

V. Polecenia assemblacji

16. CMP : Porównanie dwóch operandów

16.1 Opis

Porównuje pierwszy operand źródłowy z drugim operandem źródłowym i ustawia stan flag w rejestrze EFLAGS (odnosi się do rozszerzonych Flag) zgodnie z wynikami. Porównanie jest wykonane przez odjęcie drugiego operandu od pierwszego operandu a potem ustawienie stanu flag w ten sam sposób co przy ibstrukcji SUB. Jeśli jako operand jest używana wartość bezpośednia, jest uzupełniana znakiem do długości pierwszego operandu. Instrukcja CMP jest zwykle używana w połączeniu z instrukcją skoku warunkowego (Jcc), przeniesienia warunkowego (CMOVcc) lub SETcc. Kod warunku używanego przez Jcc, CMOVcc i SETcc jest oparty na wyniku instrukcji CMP.

16.1 Działanie

Mamy tu fragment pseudo kodu demonstrującego jak zachowuje się CPU przed wykonaniem instrukcji CMP:

```
temp -> SRC1 – SignExtend(SRC2);
ModifyStatusFlags;
```

17. cc: Skok jeśli warunek jest spełniony

17.1 Opis

Sprawdza stan jedneho lub więcej stanu flag w rejestrze EFLAGS (CF, OF, PF, SF i ZF) i jeśli flagi są w określonym stanie (warunek), wykonuje skok do docelowej instrukcji określonej przez operand przeznaczenia. Kod warunku (cc) jest związany z każdą instrukcją wskazującą warunek jaki ma zostać przetestowany. Jeśli warunek nie jest spełniony, skok nie jest wykonywany a wykonywanie programu przechodzi do instrukcji znajdującej się po instrukcji Jcc. Instrukcja docelowa jesy określona przez relatywny offset (relatywny offset do bieżącej wartości wskaźnika instrukcji jest w rejestrze EIP) Relatywny offset, generalnie jest określony przez etykietę w kodzie asemblerowym, ale na poziomie kodu maszynowego jest kodowany jako 8 lub 32 bitowa wartość bezpośednia ze znakiem, która jest dodawana do wskaźnka instrukcji. Kodowana instrukcja jest najwydajniejsza dla offsetów od -128 do + 127. Jeśli rozmiar operansu atrybutu to 16, górne dwa bajty rejestru EIP są zerowane. Ponieważ określone stany stanu flag mogą czasami być interpretowane na dwa sposoby, są definiowane dwa mnemoniki dla tych samych opkodów. Na przykład, instrukcja JA (jump if above) i instrukcja JNBE (jump if not below or equal) są alternatywnymi mnemonikami dla opkodu 77H. Instrukcja Jcc nie obsługuje dalekich skoków (skok

do innego segmentu kodu) Kiedy cel dla skoku warunkowego jest w innym segmencie, używa przeciwnego warunku niż warunek będącego testowanego przez instrukcję Jcc, a potem uzyskaj dostęp do celu z dalekiego skoku bezwarunkowego (instrukcja JMP) do innego segmentu. Na przykład poniższy warunkowy daleki skok jest niepoprawny:

```
JZ FARLABEL;
```

Aby uzyskać ten daleki skok, użyj tych dwóch instrukcji:

```
JNZ  BEYOND;
JMP FARLABEL;
BEYOND:
```

Instrukcje JECXZ i JCXZ różniąsię od innych instrukcji Jcc ponieważ nie sprawdzają one stanu flag. Zamiast tego sprawdzają zawartość rejestrów ECX i CX, odpowiednio, dla 0. Albo rejestr CX albo ECX są wybierane ze względu na rozmiar adresu atrybutu. Instrukcje te są użyteczne na początku pętli warunkowej, która kończy się instrukcją pętli warunkowej (takiej jak LOOPNE). Zabezpieczają one wejście do pętli kiedy rejestr ECX lub CX są równe 0, które mogą powodować ,że pętla się wykona 2, 32 lub 64K razy, odpowiednio, zaiast zera razy. Wszystkie skoki warunkowe są konwertowane do kodu ładującego do pamięci jedną lub dwie linie, bez względu na adres skoku.

17.2 Działanie

Poniżej mamy fragment pseudo kodu demonstrującego jak zachowuje się CPU przy wykonywaniu polecenia J xx:

```
IF warunek
THEN
EIP ¬ EIP + SignExtend(DEST);
IF OperandSize = 16
THEN
EIP ¬ EIP AND 0000FFFFH;
FI;
FI;
```

18. PUSH: odkładanie słowa lub podwójnego słowa na stos

18.1 Opis

Zmniejsza wskaźnik stosu a potem przechowuje operand źródłowy na szcycie stosu. Atrybut rozmiaru adresu segmentu stosu określa rozmiar wskaźnika stosu (16 lub 32 bity), a atrybut rozmiaru adresu bieżącego segmentu kodu określa o ile wskaźnik stosu jest zmniejszany (2 lub 4 bajty). Na przykład, jeśli atrybuty rozmiaru adresu i operandu sto 32, rejestr ESP (wskaźnik stosu) jest zmniejszany o 4 a jeśli jest to 16, rejestr SP jest zmniejszany o 2 (Flaga B w segmencie stou jest deskryptorem segmsntui określającym atrybut rozmiaru adresu stosu, a flaga D w bieżącym segmencie kodu jest deskryptorem segmentu, razem z refiksem, opisującym atrybut roziaru operandu i również atrybut rozmiaru adresu operandu źródłowego) Odłożenie 16 itowego operandu kiedy atrybut rozmiaru adresu to 32 może kończyć się niewłaściwym wskaźnikiem stosu (to znaczy, wskaźnik stosu nie jest wyrównany do granicy podwójnego słowa). Instrukcja PUSH ESP odkłada wartość rejestru ESP przed wykonaniem instrukcji. Zatem, jeśli instrukcja PUSH używa operandu pamięci w której używany jest rejestr ESP jako rejestr bazowy dla wykliczania adresu operandu, adres efektywny opernadu jest wyliczany przed zmniejszeniem rejestru ESP. W trybie adresu rzeczywistego, jeśli rejestr ESP lub SP jest 1 kiedy instrukcja PUSH została wykonana., procesor zamknie sieze względu na brak miejsca w pamięci. Nie jest generowany żaden wyjątek dla

wskazania tego warunku.

18.2 Działania

Oto fragment pseudo kodu demosntrującego jak CPU zachowuje się wykonując instrukcję PUSH:

```
IF StackAddrSize = 32
THEN
IF OperandSize = 32
THEN
ESP ¬ ESP - 4;
SS:ESP ¬ SRC; (* push doubleword *)
ELSE (* OperandSize = 16*)
ESP ¬ ESP - 2;
SS:ESP ¬ SRC; (* push word *)
FI;
ELSE (* StackAddrSize = 16*)
IF OperandSize = 16
THEN
SP ¬ SP - 2;
SS:SP ¬ SRC; (* push word *)
ELSE (* OperandSize = 32*)
SP ¬ SP - 4;
SS:SP ¬ SRC; (* push doubleword *)
FI;
FI;
```

19 POP: Zdejmowanie wartości ze stosu
19.1 Opis

Ładuje wartość,ze szczytu stosu do położenia określonego przez operand przeznaczenia a potem zwiększa wskaźnik stosu. Operandem przeznaczenia może być reejstr ogólnego przeznaczenia, komórka pamięci lub rejestr segmentowy. Atrybut rozmiaru adresu segmentu stosu określa roziar wskaźnika stosu (16 bitów lub 32 bitów – rozmiar adresu źródłowego) a atrybut rozmairu operandu bieżącego segmentu kodu określa o ile wskaźnik stosu jest zwiększany (2 lub 4 bajty) Na przykład, jeśli te atrybuty rozmiaru adresu i operandu to 32, 32 bitowy rejestr ESP (wskaźnik stosu) jest zwiększany o 4 ,a jeśli jest to 16, 16 bitowy rejestr SP jest zwiększany o 2. Flaga B w deskryptor segmentu segmentu stosu określa atrybut rozmiaru adresu stosu, a flaga D deskryptor segmentu segmentu kodu ,razem z prefiksem, określają atrybut rozmiaru operandu i również atrybut rozmiaru adresu operandu przeznaczenia. Jeśli operand przeznaczenia jest jednym z rejestrów segmentu DS, ES,FS,GS lub SS, wartość ładowana do rejestru musi być poprawnym selektorem segmentu. W trybie ochrony, zdejmowanie selektora segmnentu do rejestru segmentowego automatycznie spowoduje ,ze deskryptor informacji powiązanej z tym selektorem segmentem będzie ładowany do ukrytej części rejestru segmentowego i powoduje ,że selektor i deskryptor informacji będą kontrolować poprawność. Wartość null (0000-0003) może być zdjęta do rejestrów DS, ES, FS lub GS bez spowodowania ogólnego błędu ochrony. Jednak dalsze próby odnoszenia się do segmentu który odpowiada rejestrowi segmentowi ładowanemu wartością null, powoduje ogólny wyjątek ochrony (#GP). W tej sytuacji, nie pojawi się żadne odniesienie do pamięci a zachowana wartość rejestru segmentowego to null. Instrukcja POP nie moze zdejmować wartości do rejestru CS. Dla ładowania wartości do rejestru CS używamy instrukcji RET. Jeśli rejestr ESP jest używany jako rejestr bazowy dla adresowania operandu przeznaczenia w pamięci, instrukcja POP wylicza adres efektywny operandu po zwiększeniu rejestru ESP. Instrukcja POP ESP zwiększa wskaźnik stosu (ESP) zanim dane na starym szczycie stosu zostaną zapisane do miejsca przeznaczenia. Instruckja POP SS blokuje wszystkie przerwania, wliczając w to przerwania NMI, aż do wykonania

kolejnej instrukcji. Akcja ta pozwala na sekwencyjne wykonywanie instrukcji POP SS i MOV ESP, EBP bez niebezpieczeństwa niepoprawnego stosu podczas przerwania 1. Jednak, użycie instrukcji LSS jest preferowaną metodą ładowania rejestrów SS i ESP

19.2 Działanie

Poniżej mamy frament pseudo kodu demosntrującego jak zachowuje się CPU przy wykonywaniu polecenia POP:

```
IF StackAddrSize = 32
THEN
IF OperandSize = 32
THEN
DEST ¬ SS:ESP; (* kopiowanie podwójnego słowa *)
ESP ¬ ESP + 4;
ELSE (* OperandSize = 16*)
DEST ¬ SS:ESP; (* kopiowanie słowa *)
ESP ¬ ESP + 2;
FI;
ELSE (* StackAddrSize = 16* )
IF OperandSize = 16
THEN
DEST ¬ SS:SP; (* kopiowanie słowa *)
SP ¬ SP + 2;
ELSE (* OperandSize = 32 *)
DEST ¬ SS:SP; (* kopiowanie podwójnego słowa  *)
SP ¬ SP + 4;
FI;
FI;
```

Ładowanie rejestru segmentowego w trybie ochrony wynika ze specjalnego sprawdzania i działania ,jak pokazuje poniższy listing. To sprawdzanie jest wykonywane w selektorze segmentu i deskryptora segmentu wskazującego go.

```
IF SS jest ładowany;
THEN
IF selektor segmentu to null
THEN #GP(0);
FI;
IF indeks selektora segmentu jest poza ograniczeniem tablicy deskrytptora
OR segment selector's RPL ¹ CPL
OR segment nie jest zapisywalnym segmentem danych
OR DPL ¹ CPL
THEN #GP(selector);
FI;
IF segment nie jest obecnie zazncz
THEN #SS(selector);
ELSE
SS ¬ segment selector;
SS ¬ segment descriptor;
FI;
FI;
IF DS, ES, FS ub GS jest ładowany selektorem nie zerowym;
THEN
```

IF indeks selektora segmentu jest poza ograniczeniem tablicy deskrytptora
OR segment nie jest danymi lub odczytywalnym segmentem kodu
OR ((segmentjest danymi lub lub niezgodnym segmentem kodu)
AND (both RPL and CPL > DPL))
THEN #GP(selector);
IF segment nie jest obecnie zaznczony
THEN #NP(selector);
ELSE
SegmentRegister ¬ selektor segmentu;
SegmentRegister ¬ deskryptor segmentu;
FI;
FI;
IF DS, ES, FS lub GS jest ładowany zerowym selektorem;
THEN
SegmentRegister ¬ segment selector;
SegmentRegister ¬ segment descriptor;
FI;

NOTKA: w sekwencji instrukcji, które pojedynczo opóniają przerwania , tylko pierwsza instrukcja w tej sekwencji gwarantuje opóźnienie przerwania, ale kolejne instrukcje opóźniania przerwań mogą nie opóźniać przerwań. Zatem w tej sekwencji instrukcji

```
STI
POP SS
POP ESP
```

Przerwania mogą być rozpoznane zanim wykon się sekwencja POP ESP, ponieważSTI również opóźniania przerwania dla jednej instrukcji.

20. AND : Logiczne AND

20.1 Opis

Wykonuje operację bitowego And na operandzie przeznaczenia (pierwszy) i źródłowym (drugi) i przechowuje wynik w lokacji operandu przeznaczenia. Operand źródłowy może być stałą wartością, rejestrem lub komórką pamięci; operand przeznaczenia może być rejestrem lub komórką pamięci. (Jednak dwa operandy pamięci nie mogą być użyte w jednej instrukcji). Każdy bit wyniku instrukcji AND jest 1 jeśli oba odpowiednie bity operandów są jedynkami; w przeciwnym razie jest to 0.

20.2 Działanie i przykład

Pseudo kod :

DEST ¬ DEST AND SRC;

Przykład :
Weźmy liczby całkowite 58 i 167. Poniżej mamy przykład wyniku działania 58 AND 167. Zwróćmy uwagę,że może być to rozszerzone na 32, 64 lub więcej bitowe liczby

1. Konwertujemy liczbę 58 na liczbę binarną : 00111010
2. Konwertujemy liczbę 167 na liczbę binarną : 10100111
3. Porównujemy dwie liczby poziomo. Jeśli dwa bity w tym samym położeniu mają wartość 1, nowa liczba również powinna mieć wartość 1 dla odpowiedniego bitu. Albo 0

00111010

AND 10100111

JEST RÓWNE **00100010** czyli 34.

21. NOT : Negacja uzpełnienia do jednego

21.1 Opis

Wykonuje bitową operację NOT (każda 1 jest zmieniane na 0 a każde 0 jest ustawiane na 1) w operandzie przeznaczenia i przechowuje wynik w komórce operandu przeznaczenia. Operandem przeznaczenia może być rejestr lub komórka pamięci.

21.2 Działanie i przykład

Pseudo kod :

DEST ¬ NOT DEST;

Przykład:
Weźmy liczbę 167. Poniżej mamy ilustrację wyniku działania operacji NOT 167.Zwróćmy uwagę,że może być to rozszerzone na 32, 64 lub więcej bitowe liczby.

1. Konwertujemy liczbę 167 na liczbę binarną : 10100111
2. Odwarcamy wszystkie bity (1 powinno być 0 i vice versa)

NOT 10100111
JEST RÓWNE **01011000** czyli 88. (zauważ ,że jest to 255-167)

22. OR: Logiczne Inclusive OR

22.1 Opis

Wykonuje bitową operację inclusive OR między operandem przeznaczenia (pierwszym) a operandem źródłowym (drugi) a wyniki przechowywany jest w komóce operandu przeznaczenia.Operand źródłowy może być stałą wartością, rejestrem lub komórką pamięci; operand przeznaczenia może być rejestrem lub komórką pamięci. (Jednak dwa operandy pamięci nie mogą być użyte w jednej instrukcji). Każdy bit wyniku instrukcji OR jest 0 jeśli oba odpowiednie bity operandów są zerami; w przeciwnym razie jest to 1

22.2 Działanie i przykład

Pseudo kod:

DEST ¬ DEST OR SRC;

Przykład:

Rozważmy liczby całkowiter 58 i 167. Poniżej mamy przykład wyniku działania 58 OR 167. Zwróćmy uwagę,że może być to rozszerzone na 32, 64 lub więcej bitowe liczby

1. Konwertujemy liczbę 58 na liczbę binarną : 00111010
2. Konwertujemy liczbę 167 na liczbębinarną : 10100111
3. Porównujemy liczby poziomo. Jeśli dwa bity w tym samym położeniu mają wartość 0, nowa liczba powinna mieć również wartość 0 dla odpowiedniego bitu. Inaczej, 1

	00111010
OR	10100111
JEST RÓWNE	**10111111** co oznacza 191.

23. XOR: Logiczne exclusive OR

23.1 Opis

Wyonuje bitową operację exclusive OR (XOR) na operandzie przeznaczenia (pierwszy) i operandzie źródłowym (drugi) a wynik przechowuje w położeniu operandu przeznaczenia. Operand źródłowy może być stałą wartością, rejestrem lub komórką pamięci; operand przeznaczenia może być rejestrem lub komórką pamięci. (Jednak dwa operandy pamięci nie mogą być użyte w jednej instrukcji). Każdy bit wyniku jest 1 jeśli odpowiednie bity operandów są różne; każdy bit jest 0 jeśli odpowiednie ity są takie same.

23.2 Działanie i Przykład

Pseudo kod: DEST ¬ DEST XOR SRC;

Przykład

Rozważmy liczby całkowiter 58 i 167. Poniżej mamy przykład wyniku działania 58 XOR 167. Zwróćmy uwagę,że może być to rozszerzone na 32, 64 lub więcej bitowe liczby

1. Konwertujemy liczbę 58 na liczbę binarną : 00111010
2. Konwertujemy liczbę 167 na liczbębinarną : 10100111
3. Porównujemy liczby poziomo. Jeśli dwa bity w tym samym położeniu są takie same, wynik to 0 w przeciwnym razie 1

	00111010
XOR	10100111
JEST RÓWNE	**10011101** co daje 157

24. Pozostałe instrukcji

24.1 CALL: Wywołanie procedury

Zachowuje informacje linkowania na stosie i skacze do procedury (wywołanie procedury) określonej w operandzie (docelowym) przeznaczenia. Operand docelowy określa adres pierwszej instrukcji w wywołanej procedurze. Opernad ten może być wartością bezpośrednią, rejestrem ogólnego przeznaczenia lub komórką pamięci. Instrukcja ta może być używana do czterech różnych typów wywołań:

- Wywołanie bliskie – wywołanie procedury wewnątrz bieżącego segmentu kodu)segment wskazywany przez rejestr CS), czasmio odnosimy sięndo niego jako wywołania wewnątrz sgmetntowe
- Dalekie wywołanie – wywołanie procedury umieszczonej w innym segmencie niż bieżący segment kodu, czasmi odnosimy się do niego jako wywołania międzysegmentowego.
- Dalekie wywołanie między uprzywilejowanyymi poziomami : dalekie wywołanie procedury w segmencie o innym poziomie uprzywilejowania niż aktualnie wykonywany progrma lub procedura.
- Przełączanie zadań – wywołanie procedury umieszczonej w innym zadaniu.

Ostatnie dwa typy wywołań mogą być wykonywane tylko w trybie chronionym

24.1.1 Wywołanie bliskie
Kiedy wykonujemy bliskie wywołanie, procesor odkłada wartość rejestru EIP (zawierający offset instrukcji występującej po instrukcji CALL) na stos(do użycia jako wskaźnik instrukcji powortu). Procesor potem przeskakuje do adresu w bieżącym segmencie kodu określoneow operandzie docelowym. Operand docelowy określa albo absolutny offset w segmencie kodu (który jest offsetem od początku segmentu kodu) lub offset relatywny (przemieszczenie ze znakiem relatywne do bieżącej wartości wskaźnika instrukcji w rejestrze EIP, który wskazuje instrukcję występującą po instrukcji CALL) Rejestr CS nie jest zmieniony przy bliskim wywołaniu. Dla bliskiego wywołania, offset absolutny jest określany pośrednio w rejestrze ogólnego przeznaczenia lub komórkę pamięci. Atrybut rozmiaru operandu określa rozmiar operandu docelowego (16 lub 32 bity). Offsety absolutne są ładowane bezpośrednio do rejestru EIP. Jeśli atrybut rozmiaru operandu to 16, górne dwa bajty rejestru EIP są zerowane.(Kiedy mamy dostęp do offsetu absolutnego pośrednio używając wskaźnik stosu [ESP] jak rejestru bazowego, wartość bazowa używana jest to wartość ESP bezpośrednio przed wykonaniem instruckji). Offset relatywny jest ogólnie określony jako etykieta w kodzie asemblerowym, ale na poziomie kodu maszynowego, jest kodowany jako wartość bezpośrednia ze znakiem 16 lub 32 bitowa. Wartość ta jest dodawana do wartości rejestru EIP. Podobnie jak z offsetami absoltnymi, atrybut rozmiaru operandu określa rozmiar operandu docelowego (16 lub 32 bity).

24.1.2 Dalekie wywołania w trybie adresu rzeczywistego lub Virtual-8086

Kiedy wykonujemy dalekie wywołanie w trybie adresu rzeczywistego lub Virtual-8086, procesor odkłada bieżącą wartość rejestrów CS i EIP na stosie do wykorzystania jako wskaźnik instrukcji powrotu. Potem wykonuje "dalekie rozwidlenie" do segmentu kodu i offsetu określonych operandem doelowym dla wywołanej procedury. Tu operand docelowy określa absolutny daleki adres albo bezpośrednio ze wskaźnikiem (ptr16:16 lub ptr16:32) albo pośrednio przez komórkę pamięci (m16:16 lub m16:32). W metodzie ze wskaźnikiem, segment i offset wywoływanej procedury jest zakodowany w instrukcji, używając 4 bajtowego (16 bitowy rozmiar operandu) lub 6 bajtowego (32 bitowy rozmiar operandu) bezpośredniego dalekiego adresu. Przy metodzie pośredniej, operand docelowy określa komórka pamięci zawierająca 4 bajtowy (16 bitowy rozmiar operandu) lub 6 bajtowego (32 bitowy rozmiar operandu) dalekiego adresu. Atrybut rozmiaru operandu określa rozmiar offsetu (16 lub 32 bity) w dalekim adresie. Daleki adres jest ładowany bezpośrednio do rejestrów CS i EIP. Jeśli atrybut rozmiaru operandu wynosi 16, górne dwa bajty rejestru EIP są zerowane.

24.1.3 Dalekie wywołania w trybie chronionym

Kiedy procesor działa w trybie chronionym, instrukcja CALL może by wykorzystana w trzech typach dalekiego wywołania:

- Dalekie wywołanie do tego samego poziou uprzywilejowania
- Dalekie wywołanie do różnych poziomów uprzywilejowania (wywołanie między poziomami uprzywilejowania)
- Przełączanie zadań (dalekie wywołanie do innego zadania)

W trybie chronionym, procesor zawsze używa części selektora segmentu dalekiego adresu aby mieć dostęp do odpowiedniego deskryptora w GDT lub LDT. Typ deskryptora (segment kodu, bramka wywołania, bramla zadania lub TSS) i prawa dostępu określają typ operacji wywołania jak będzie wykonana. Jeśli wybrano deskryptor dla segmentu kodu, dalekie wywołanie do segmentu kodu na tym samym poziomie uprzywilejowania będzie wykonane. Jesli wybrany segment kodu jest na

innym poziomie uprzywilejowania a segment kodu jest niezgodny , generowany jest ogólny błąd ochrony. Dalekie wywołanie do tego samego poziomu uprzywilejowania w trybie chronionym jest bardzo podobne do tego w trybie adresu rzeczywistego lub virtual-8086 Operand docelowy określa absolutny daleki adres albo bezpośednio ze wskaźnikiem (ptr16:16) albo pośrednio przez komórkę pamięci (m16:16 lub m16:32). Atrybut rozmiaru operandu określa rozmiar offsetu (16 lub 32 bity) w dalekim adresie. Nowy selektor segmentu kodu i jegio deskryptor są ładowane do rejestru CS,a offset instrukcji jest ładowany do rejestru EIP. Zwróć uwagę ,że bramka wywołania może być również użyta dla wykonania dalekiego wywołania do segmentu kodu na tym samym poziomie uprzywilejowania. Używając tego mechanizmu dostarczamy pośredniego dodatkowego poziomu i jesat to preferowana metoda tworzenia wywołań między 16 a 32 bitowymi segmentami kodu. Kiedy wykonujemy dalekie wywołanie między uprzywilejowanymi poziomami, segment kodu dla procedury będącej wywołaną musi być dostępny przez bramkę wywołania. Selektor segmentu określony przez operand docelowy identyfikuje bramkę wywołania. Tu, ponownie, operand docelowy może określić bramkę wywołania selektora segmentu albo bezpośrednio przez wskaźnik (prt16:16 lub pt16:32) lub pośrednio przez komórkę pamięci (m16:16 lub m16:32). Procesor uzyskuje selektor segmentu dla nowego segmentu kodu a nowy wskaźnik instrukcji (offset) z deskryptora bramki wywołania. Offset z operandu docelowego jest ignorowany kiedy jest używana bramka wywołania. W wywołaniu poziomu uprzywilejowania, procesor przełącza do stosu dla poziomu uprzywilejowania wywołanej procedury. Selektor segmentu na nowego segmentu stosu jest określony w TSS dla aktualnie wykonywanego zadania. Gałąź do nowego segmentu kodu wystąpi po przełączeniu stosu (Zwróć uwagę , kiedy używamy bramki wywołania wykonujemy dalekie wywołanie do segmentu na tym samym poziomie uprzywilejowania). Na nowym, stosie, procesor odkłada selektor segmentu i wskaźnik stosu dla wywołania stosu procedury (opcjonalnie) ustawia parametry ze stosu wywołanej procedury, a selekor segmentu i wskaźnika instrukcji dla segmentu kodu wywołanej procedury. Wartość deskryptora bramki wywołania określa jak wiele parametrów skopiowano na nowy stos. W końcu, procesor skacze do adresu procedury wywołanej wewnątrz nowego segmentu kodu. Wykonanie przełączania zadania instrukcją CALL jest podobna do wykonania wywołania przez bramkę wywołania. Tu operand docelowy określa selektor segmentu brami zadania dla zadania będącego przełączanym (a offset w operandzie docelowym jest ignorowanym) Bramka zadania w zasadzie wskazuje TSS dla tego zadania, który zawiera slektory segmentu do kodu zadania i segmentu stosu. TSS również zawiera wartość EIP dla kolejnej instrukcji, która była wykonywana przed zawieszeniem zadania. Ta wartość wskaźnika instrukcji jest łądowana do rejestru EIP więc zadanie zacznie wykonywanie ponownie od tej kolejnej instrukcji. Instrukcja CALL może również określić selektor segmentu TSS bezpośrednio, co eliminuje pośrdnio bramkę zadania. Zwróć uwagę ,że kiedy wykonujesz przełączanie zadania instrukcją CALL, flaga zagnieżdżonego zadania (NT) jest ustawiona w rejestrze EFLAGS a nowe pole łączenia poprzedniego zadania TSS jest ładowane ze starym selektorem zadania TSS. Kod oczekiwany do zakończenia zagnieżdżonego zadania wykonujemy przez wykonanie instrukcji IRET, która, z powodu ustawienia flagi NT, automatycznie będzie używała poprzedniego łącza zadania do powrotu z wywołanego zadania. Przełączanie zadań instrukcją CALL różni się w tym względzie od instrukcji JMP, która nie ustawia flagi NT i dlatego nie oczekuje instruckji IRET dla zakończenia zadania.

24.1.4 Mieszane wywołania 16 i 32 bitowe

Kiesy robimy dalekie wywołania między 16 a 32 bitowymi segmentami kodu, wywołania powinny być robione przez bramkę wywołania. Jeśli dalekie wywołanie jest z 32 bitowego segmentu kodu do 16 bitowego segmentu kodu, wywołanie powinno pochodzić z pierwszych 64 KB 32 bitowego segmentu kodu. Jest tak ponieważ atrybut rozmiaru opernadu instrukcji jest ustawiony na 16 więc tylko 16 bitowy adres powrotu jest zapisywany. Również wywołanie powinno być tworzone przy użyciu 16 bitowej bramki wywołania więc 16 bitowa wartość będzie odkładana na stos.

24.2 ADD : Dodawanie

Dodaje pierwszy operand (operand przeznaczenia) i drugi operand (operand źródłowy) i przehowuje wynik w operandzie przeznaczenia. Operand przeznaczenia może być rejestrem lub komórką pamięci; operand źródłowy może być wartością bezpośrednią, rejestrem lub komórką pamięci. (Jednak, dwa operandy pamięci nie mogą być użyte w jednej instrukcji) Kiedy wartość bezpośednia jest używana jako operand, jest rozszerzony znakiem do długości formatu operandu przeznaczenia. Instrukcja ADD nie rozpoznaje operandów ze znakiem i bez znaku. Zamiast tego procesor wylicza wynik obu typów danych i ustawia flagi OF i CF wskazujące przeniesienie w wyniku ze znakiem i bez znaku, odpowiednio. Flaga SF wskazuje znak wyniku.

24.3 SUB : Odejmowanie

Odejmuje drugi operand (operand źródłowy) od pierwszego operandu (operand przeznaczenia) i przechowuje wynik w operandzie przeznaczenia. Operandem przeznaczenia może być rejestr lub komórka pamięci; operandem źródłowym może być wartość bezpośrednia, rejestr lub komórka pamięci (Jednak dwa operandy pamięci nie mogą być użyte w jednej instrukcji) Kiedy wartość bezpośrednia jest używana jako operand, jest rozszerzana znakiem do długości foramtu operandu przeznaczenia. Instrukcja SUB nie rozróżnia między operandem ze znakiem czy bez znaku. Zamiast tego, procesor oblicza wynik dla obu typu danych i ustawia flagi OF i CF aby wskazywały pożyczkę w wyniku ze znakiem i bez znaku, odpowiednio. Flaga SF wskazuje znak wyniku ze znakiem.

24.4 MUL : Mnożenie bez znaku

Wykonuje mnożenie bezznakowe dla pierwszego operandu (operand przeznaczenia) i drugiego operandu (operand źródłowy) i przechowuje wynik w operandzie przeznaczenia. Operandem przeznaczenia jest niejawny operand umieszczony w rejestrze AL, AX lub EAX (w zależności od rozmiaru operandu); operand źródłowy jest umieszczony w rejestrze ogólnego przeznaczenia lub komórce pamięci. Wynik jest przechowywny w rejestrze AX, parze rejestrów DX:AX lub EDX:EAX (w zależności od rozmiaru operandu), z wyższym porządkiem bitów iloczynu zawartym w rejestrze AH, DX lub EDX , odpowiednio. Jeśli wyższy porządek bitów iloczynu to 0, flagi CF i OF są zerowane; w przeciwnym razie te flagi są ustawiane.

24.5 DIV : Dzielenie bezznakowe

Dzieli (bezznakową) wartość w rejestrze AX, parze rejestrów DX:AX lub EDX:EAX (dzielna) przez operand źródłowy (dzielnik) i przechowuje wynik w rejestrach AX (AH:AL), DX:AX lub EDX:EAX .Operandem źródłowym może być rejestr ogólnego przeznaczenia lub komórka pamięci. Wyniki niecałkowite są zaokrąglane do zera. Reszta jest zawsze mniejsza niż dzielnik. Przepełnienie jest wskazywane z wyjatkiem #DE (divide error) zamiast flagą CF.

24.6 MOV : Przenoszenie

Kopiuje drugi operand (operand źródłowy) do pierwszego operandu (operand przeznaczenia). Operandem źródłowym może być wartość bezpośrednia, rejestr ogólnego przeznaczenia, rejestr segmentowy lub komórka pamięcię; operandem przeznaczenia może być rejestr ogólnego przeznaczenia, rejestr segmentowy lub komórka pamięci. Oba operandy muszą być tego samego rozmiaru, którym może być bajt, słowo lub podwójne słowo. Instrukcja MOV nie może być używana do ładowania rejestru CS. Próba zrobienia tego da w wyniku tego opkod niewłaściwego wyjątku (#UD). Aby załadować rejestr CS, użyjemy instrukcji dalekiego JMP, CALL lub RET. Jeśli operandem przeznaczenia jest rejestr segmentowy(DS,ES, FS, GS lub SS), operandem źródłowym musi być poprawny selektor segmentu. W trybie chronionym, przeniesienie selektora segmentu do rejestru segmentowego autoamtycznie spowoduje to ,że informacja deskryptora segmentu powiązanego z tym selektorem segmentu będzie załadowana do ukrytej (cień) części rejestru segmentowego. Podczas ładowania tej informacji, selektor segmentu i informacja deskryptora

segmentu jest potwierdzana. Data deskryptora segmentu jest uzyskiwana z wejścia GDT lub LDT dla określonego selektora segmentu. Zerowy selektor segmentu (wartości 0000-0003) może być ładowany do rejestrów DS, ES, FS i GS bez powodowania wyjątku ochrony. Jednak, dowolna kolejna próba odniesienia się segmentu którego odpowiedni rejestr segmentowy jest załądowany wartoscią null powoduje ogólny wyjątek ochrony (#GP) i nie wystąpi żadne odniesienie do pamięci. Załadowanie rejestru SS instrukcją MOV zablokuje wszystkie przerwania dopóki czasu wykonania kolejnej instrukcji . Operacja ta pozwala załadować wskaźnik stosu do rejestru ESP instrukcją (MOV ESP, wartość wskaźnika stosu) przez pojawieniem się przerwania 1. Instrukcja LSS oferuje bardziej wydajną metodę ładowania rejestrów SS i ESP. Kiedy działamy w trybie 32 bitowym i przenosimy dane między rejestrem segmentowym a rejestrem ogólnego przeznaczenia, Procesory Intel Architecture 32 bitowa nbie wymaga użycia 16 bitowego prefiksu rozmiaru operandu (bajt wartością 66H) z tą instrukcją, ale większość asemblerów będzie wstawiała go jeśli jest używana standardowa forma instrukcji (np. MOV DS, AX). Procesor wykona tą instrukcję poprawnie, ale zazwyczaj będzie wymagała dodatkowego cyklu zegarowego. Większość asemblerów używając formy instrukcji MOV DS, EAX będzie unikała tego niepotrzebnego prefiksu 66H. Kiedy procesor wykonuje instrukcję z 32 bitowym rejestrem ogólnego przeznaczenia, zakłada ,że 16 najmniej znaczących bitów rejestru ogólnego przeznaczenia są operandami przeznaczenia lub źródłowym. Jeśli rejestr jest operandem przeznaczenia, wartość wynikowa w dwóch wyższych bajtach rejestru jest implementowany zależnie.

VI. SoftIce dla Windows

25. Instalacja SoftIce

Kliknij podwójnie plik instalacyjny. Zainicjuje się standardowy proces instalacji. (InstalShield). Kliknij Next. Przeczytaj i zatwierdź warunki licencji. W kolejnym okienku dialogowym wpisz numer seryjny produktu. Kliknij Next. Pozostawiamy domyślną instalację. Jeśli z jakichś powodów chcesz zmodyfikować katalog instalacji, możesz to zrobić ale wiedz, że podana ścieżka absolutna w tym opisie musi wskazywać katalog jakiego użyłeś do instalacji. Kliknij Next. Zaznacz wszystkie cztery opcje instalacji. Możesz przeskoczyć pliki demonstracyjne jeśli chcesz. Pełna instalacja zajmie w przybliżeniu 13 MB miejsca na dysku. Kliknij Next. Jeśli jesteś całkowicie pewny co do swojej karty graficznej, po przetestowaniu jej z powodzeniem, pozostaw domyślny adapter VGA. Zwróć uwagę ,że mądrze jest nacisnąć przycisk Test nawet jeśli pozostawiłeś zaznaczony domyślny adapter VGA ,Monochromatyczna karta / monitor powinny być odznaczone, podczas gdy Universal Video Driver powinien pozostać zaznaczony. Kliknij Next.. Nie zaznaczaj myszki jeszcze; robimy to później. Jednak, jeśli wybierzesz sterownik myszki, upewnij się,że dokonałeś dobrego wyboru. Kliknij Next. Teraz proces instalacyjny wymaga zezwolenia na modyfikację pliku autoexec.bat. Powinieneś udzielić zezwolenia przez zaznaczenie pierwszej opcji. SoftIce wstawi parę ważnych linii w pliku autoexec.bat aby SoftIce mógł ładować się przy starcie systemu. Kliknij dwa razy Next. Instalator skopiuje wszystkie koniewczne (i zaznaczone) pliki na dysk. Po tym procedura jest zakończona, zostaniesz poproszony o rejestrację SoftIce. Wybierz ostatnią opcję (rejestruj później) i naciśnij Next. Może pojawić się prośba o zainstalowanie Adobe Acrobata. Jeśli nie masz tego programu, musisz go mieć aby przeczytać podręcznik SoftIce. Jednakże ,możesz przeskoczyć ten komunikat przez naciśnięcie OK. Musisz zrestartować system. Używamy Ctrl + D aby uruchamiać SoftIce

26. Konfiguracja SoftIce

Przejdź do programów i zobacz nową grupę programów nazwaną Numega SoftIce. Wewnątrz niej znajdź dwie opcje setupu, jedną dla wyświetlania adapteru i jedną dla myszki. Możesz zmienić ustawienia myszki jeśli nie działa ale nie zmieniaj nic w ustawieniach wyświetlania jeśli działają. SoftIce pojawiać się będzie w małym okienku. Są również 4 pliki pomocy, standardowy plik html

read me, plik pomocy i dwa pliki pdf. Wszystko to jest użyteczne, zwłaszcza jeśli wystąpią nieoczekiwane problemy. W końcu, jest tam apliakcja nazwana Numega SoftIce Loader. Nie ładuje ona SoftIce, nazwa może być myląca. Edytuje ona graficzny interfejs pliku inicjalizacyjnego (.ini) używanego do przechowywania uustawień SoftIce. Znajdź plik nazwany winice.dat umieszczony w katalogu instalacyjnym SoftIce. Otwórz go w Notatniku. Plik ten przechowuje ustwienia adaptacyjne jakie SoftIce odczytuje za każdym uruchomieniem

26.1 Zmiana rozmiaru Paneli

Otwórz SoftIce (naciśnij Ctrl + D). Wpisz lines .Na dole linii zobaczysz linię pomocy,wyjaśniającą co robi dana komenda. Naciśnij spację.Wtedy zobaczysz parametry jakie akceptuje wiersz poleceń. Ja używam 60. Poeksperymentuj ze wszysytkimi liczbami jaki zobaczysz. Potem, jeśli chcesz dokonać zmian (pozostaw jakiśczas uruchomiony SoftIce), przejdźdo Notatnika, gdzie znajduje się otwarty plik winice.dat i zapisz ustawienia:

INIT="X;" is replaced by **INIT="lines 60;"**

Zapisz zmiany w notatniku. Aby zmienić liczbę kolumn w oknie, proces jest taki sam, ale użyte polecenie to width. Tak więc, wpisz width a SoftIce powie ci jaką ma szerokość. Naciśnij spację a uzyskasz zakres ustawień jakie możesz użyć. Ja używam 80. Ustawienia musisz wstawić do pliku winice.dat jak "width = 80;" podobnie z wierszem poleceń:

INIT = "lines 60; width=80;"

26.2 Panele

SoftIce ma wiele paneli, które są konfigurowalne. To znaczy, możesz ukryć lub odkryć każy z nich i oczywiście zmienić jego rozmiar, z lub bez pomicy myszki. Będziemy używali poleceń klawiaturowych.

Polecenie	Wyjaśnienie
WC [liczba]	Jeśli liczba jest określona, przełączza okno kodu. Jeśli liczba jest obecna, ustawia linie okna kodu równe tej liczbie (Zalecane: 25, jest to bardzo ważne okno!)
WR	Przełącza okno rejestrów któe są na szczycie części okna. Zalecane jest aby okno rejestrów było zawsze widoczne.
WD [liczba]	Liczba zachowuje się jak w poleceniu WC. Przełącza okno danych, które może być użyte jako edytor szesnastkowy. Możesz chcieć zamknąć to okno i zwolnić trochę miejsca.
WF	Przełącza okno wskaźnika stosu zmiennoprzecinkowego. Nie będziemy używać tego okna.
WL	Przełącza lokalne okno. Nie będziemy go używać
WS	Przełącza okno stosu.Nie będziemy go używać
WW	Przełącza okno strażnika. Ustawia straż poleceniem watch <adres>. Jest bardzo użyteczne i będzie potrzebne w przyszłości.

Aby dokonywać zmian, przechowuj te ustawienia w pliku winice.dat. Oto moja propozycja:

INIT="lines 60;code on;wd 13;wc 25;wr;ww 6; faults off;"

26.3 Inne użyteczne ustawienia

Polecenie code włącza i wyłącza kolumnę kodu w oknie disasemblacji.Okno kodu jest drugą kolumną od lewej do prawej i zawiera okody funkcji, które sązdisasemblowane. Jeśli jest wyłączone są tylko trzy kolumny zamiast czterech. Przyjęło się ustawienie faults off jako domyślnego ustawienia. Wymusza to na SoftIce brak wyskakiwanie za każdym razem keidy aplikacja okienkowa się zawiesza. Poza tym nie chcemy debuggować wszystkiego! Konieczne jest ustawienie faults on kiedy debuggujemy wsłasną aplikację ponieważ w przypadku błędu, SoftIce się nie pojawi kiedy błędy są wyłączone. Jeśli Ctrl+D jest niepręczne dla ciebie, możesz zawsze użyć polecenia altkey do zmiany. Składnia ALTKEY to klawisz CTRL lub ALT. Na przykład altkey alt F zastąpi skrót CTRL+D na ALT+F .Pewnie zauważyłeś ,że kursor górny i dolny są używane do nawigowania przez poprzednie polecenia w panelu poleceń. Również zauważ ,że wpisanie cls powala wyczyścić panel SoftIce. Ustawienie mouse x, ustawia szybkość myszki od 0 (najwolniejsza) do 3 (najszybsza). Wszystkie te ustaawienia (i inne) mogą być zapisane w pliku winice.dat, i będą przywracane przy każdej inicjalizacji SoftIce:

```
INIT="lines 60;code on;wd 13;wc 25;ww 6;wl;dex 1 ss:esp;faults off;"
INIT="altkey ctrl d;watch es:di;watch eax;watch *es:di;set mouse 3;cls;X;"
```

26.4 OknoSoftice

Pponieważ wszystko jest zamrażane, nie jest łatwo pobrać zrzut ekranowy okna SoftIce. Sposobem może być zamrożenie czegoś w tle, wywołanie SoftIce i zamknięcie go. SoftIce pozostanie zamrożonyw oknie, ponieważ odmalowanie jest niedostępne dla zamrożonej aplikacji. Wtedy, można prawdopodobnie zrobić zrzut ekranowy (lub nie , ponieważ SoftIce uwa różnych sterowników monitora)

```
+---------------------------------------------------------------------------+
| EAX=...     This is the registers window, activate it                     |
| EDI=...     by typing wr                                                  |
+---------------------------------------------------------------------------+
| es:di=...   This is the watch window, activate it by typing ww.  To watch |
|             something, type watch bla, eg watch eax                       |
+---------------------------------------------------------------------------+
| xxxx:yyyyyyyy 01 02 03 04 05 06 07 08-09 10 11 12 13 14 15 16 ...............|
| xxxx:yyyyyyyy 01 02 03 04 05 06 07 08-09 10 11 12 13 14 15 16 ...............|
|                                                                           |
| This is the data window, activate it by typing wd.  It is here that you can |
| see what's stored in memory.                                              |
+---------------------------------------------------------------------------+
| 0028:C000A010  7902             JNS     C000A014                (JUMP  )|
| 0028:C000A012  33C0             XOR     EAX,EAX                           |
| 0028:C000A014  83E81F           SUB     EAX,1F                            |
|                                                                           |
| This is the code window, where the assembly commands the target is executing |
| are displayed.  Activate this window by typing wc.                        |
+---------------------------------------------------------------------------+
|:< type your commands here >                                               |
|                                                                           |
| Well this is where you type your commands                                 |
+---------------------------------------------------------------------------+
```

26.5 Symbole

Symbole, są funkcjami DLL, które są importowane w SoftIce, więc program zna jakie parametry są pobierane jako argumenty i jak są zwracane (jeśli coś jest zwracane) Symbole są ładowane w pliku winice.dat, na końcu pliku ze średnikiem przed nimi. Usunięcie tych średników jest wymagane aby zaimportować symbole i ustawić punkty przerwania.

```
;EXP=c:\windows\system\kernel32.dll
;EXP=c:\windows\system\user32.dll
;EXP=c:\windows\system\gdi32.dll
;EXP=c:\windows\system\comdlg32.dll
;EXP=c:\windows\system\shell32.dll
;EXP=c:\windows\system\advapi32.dll
;EXP=c:\windows\system\shell232.dll
;EXP=c:\windows\system\comctl32.dll
```

Proszę zwrócić uwagę ,że chociaż byłoby miło zatrzymać wszystkie moduły (moduły okienek i programy dll'i), jest to ważne ponieważ zajmuja znyt wiele zasobów. Więc, założymy punkty przerwań tylko na "popularne" moduły takie jak kernel32.dll, user32.dll, comdlg32.dl i advapi32.dll. Możesz oczywiście założyć punkty przerwania wszystkich domyślnych wejść w module winice.dat SoftIce. Jest inny punkt jakim ,musimy się zająć, to ścieżka do modułu. Musi to być absolutna ścieżka dostępu. Więc jeśli katalogiem instalacyjnym Windows jest C:\WINNT, prawdopodobnie kernel32.dll znajduje się w C:\Winnt\system32\kernel32.dll. Jeśli nie jesteś pewny, możesz użyć funkcji Search for Files aby wyszukać moduł na którym chcesz założyć punkt przerwania.

27. Punkty przerwania

Poniżej mamy wylistowane wszystkie popularne punkty przerwań z poleceniami. Oczywiście, jest wiele skrótów i poleceń związanych z punktami przerwań. Najlepiej zajrzeć jest do podręcznika SoftIce po więcej informacji. Zanim przejdziesz do zakładania breakpointów, upewnij się,że dobrze zrozumiałeś co to sąsymbole i w jakich DLL'ach rezydują fnkcje na jakich chcesz założyć breakpoint. Użycie polecenia bpx ustawia breakpoint na wykonanie. Polecenie to pobiera jako argument tylko nazwę funkcji. Na wykonaniu oznacza ,że SotIce pojawi się kiedy to polecenie jest wywołane. Wtedy możesz łatwo nawigować do funkcji wywołującej. Zwróć uwagę ,że aby ustwić brekapoint, musisz mieć zaimportowane symbole. Jeśli nie znasz nazwy funkcji na jakiej chcesz ustawić breakpoint, zajrzyj do platformy Microsoft SDK dla zaktualizowania listy. Niektóre funkcje są określone przez system operacyjny. Wszystkie punkty przewrań są przechowywane na liście breakpointów. Aby zobaczyć pozycje na liście breakpointów , wpisz bl w wierszu poleceń. Pierwsza kolumna wskazuje indeks punktu przerwania, który zaczyna zliczanie od 0. Dlatego te, drugi indeks to 1, trzeci 2 itd. Indeks jest bardzo użyteczny kiedy chcesz edytować punkt przerwania. Aby wyedytować istniejący breakpoint skorzystaj z instrukcji bpe <indeks>. Spójrzmy na kilka przykładów:

bpx getdlgitemtetexta

- Tu ustawiamy breakpoint na wykonanie funkcji "getdlgitemtexta". SoftIce potrzebuje symboli z modułu user32.dll, ponieważ ta funkcja jest częścią modułu user32.dll.

bl

- Pobieramy listę breakpointów. Zwrócć uwagę,że SoftIce zaczyna listowanie breakpointów po ich indeksach (00), typie (BPX = na wykonaniu) i module!nazwie funkcji tj. USER32.DLL!GetDlgItemTextA. SoftIce automatycznie rozpoznaje ,że ta funkcja należy do user32.dll Jeśli nie możesz znaleźć właściwego modułu, wtedy musimy albo edytować winice.dat albo użyć innnej nazwy funkcji.

bpe 0

- Decydujemy ,czy ten breakpoint jest niepoprawny, więc chcemy go edytować. SoftIce przejdzie do trybu edycji, z bieżącą wartością breakpoinra w pamięci – 0

bp getdlgitemtetextw

- Zastępuje getdlgitemtetexta na getdlgitemtetextw .Aby zweryfikować zmiany wszystko co musimy zrobić to użyć bl.

Dla usunięcia wszystkich breakpointów jakie ustawiłeś, musisz użyć polecenia bc (breakpoint clear) Jeśli używana jest (*) jako argument, wszystkie breakpointy będą wyczyszczone. Musimy wskazać liczbę breakpointów jaką życzysz sobie usunąć (użyj bl dla uzyskania tej liczby) Możemy odkryć ,że wszystkie breakpointy w pamięci nie są użyteczne. Dlatego możemy chcieć zablokować wszystkie z nich bez ich wymazywania. Jeśli wymażemy je, wszystkie które są konieczne będą zablokowane. Dla tego celu użyjemy poleceń bd i be, które mają ten same argumenty z bc. Jeśli używana jest (*), wszystkie breakpointy będą zablokowane lu włączone odpowiednio, w przeciwnym razie breakpoint który odpowiada temu indeksowi będzie blokowany lub włączany.Zwróć uwagę ,że kiedy listujemy breakpointy (bl), zablokowane breakpointy są listowane z * po ich indeksach.

Polecenia	Wyjaśnienia
bpx <nazwa>	Polecenie ustawia breakpoint na wykonanie. Nie możesz ustawić tego samego breakpointu dwa razay. Pojawi się błąd (zduplikowany breakpoint)
bl	Listuje wszystkie punkty przerwanń
bpe <indeks>	Edytuje breakpoint ,który już jest zdefiniowany. Abyb zobaczyć jaki jest indeks punktu przerwania który cię intersuje, użyj bl
bc * lub <indeks>	Zeruje jeden lub wszystkie breakpointy
bd * lub <indeks>	Wyłącza jeden lub wszystkie breakpointy
be * lub <index>	Włącza jeden lub wszystkie breakpointy
bh	Historia breakpointów

Czasami jest nie dość ustawionych breakpointów tylko na wykonanie. SoftIce jest nadzwyczaj wszechstronny i praktyczny w ustaawianiu breakpointów na innych rzeczach i działa w trybie pamięci lub trybie sprzętowym. Jest tak dlatego ,że jest bardzo użyteczny kiedy debuggujemy sterowniki lub Windows

- Bmsg breakpoint na komunikatach Windows
- Bpint breakpoint na przerwaniach
- Bpio breakpoint na I/O
- Bpm breakpoint na dostępie do pamięci (może być bpm, bpmb, bpmd, bpmw)
- Bpr breakpoint na zakresie pamięci (bardzo, bardzo użyteczny)

Można również znaleźć bpt (szablon) bstat (statystyka) , ale te polecenia wychodzą poza zakres tego tekstu.

28 Użyteczne funkcje

GetDlgTextItemA jest wywoływana za każdym razem kiedy jest pobierany łańcuch z pola tekstowego. Dlatego też,polegając na fakcie ,że za każdym razem wpisujemy coś w polu tekstowym a program róbuje pobrać te dane tam zawarte,wykonywana będzie ta funkcja.

Hmemcpy jest lepsza niż poprzednia,ponieważ funkcja ta zawsze będzie wywoływana. Załóżmy ,że programista omija getdlgitemtetexta przez użycie swoich własnych kontrolek i metod. Ponieważ łańcuch będzie przechowywany w zmiennej która rezyduje w pamięci, hmemcpy zostanie wywołana.

29. Nawigacja w SoftIce

Po uistawieniu breakpointów i pojawieniu sięSoftIce'a, wszystko co musimy to odkryć kro wywołuje funkcję na jakiej chcemy założyć breakpoint. To znaczy, funkcja rezyduje w module, którego symbole są importowane w SoftIce. Kiedy SoftIce wykrywa tą funkcję, oznacza to ,że jesteśw module , który implementuje tą funkcję i oczywiście nie jesteś nią zainteresowany. Pierwsza rzecz jaką musisz zrobić to wrócić do miejsca wywołującego. Jest to wykonane przez naciśnięcie kalwisza F11 (skok do programu wywołującego) .Potem możesz zajrzeć do poniższej tabelki aby zacząć nawigować po kodzie.

Skrót	Wyjaśnienie
F3	Przełącza między różnymi panelami SoftIce jeśli to konieczne
F4	Pokazuje ekran programu
F7	Wykonanie w miejscu kursora
F8	Wykonanie pojedynczego kroku
F10	Przechodzenie
F11	Skok do programu wywołującego, idziemy do funkcji, która wywołała punkt przerwania.
F12	Powrót z procedury wywołania

VII. Edytor HACKMAN

Hackman jest to edytor szesnastkowy i disasembler. Jego zadaniem jest ułatwienie eksportu i modyfikacji części kodu źródłowego dowolnego programu wykonywalnego. Oczywiście ,może być użyty jako sam edytor heksadecymalny. Poniżej opiszę najciekawsze dostępne funkcje.

30. Manipulacja łańcuchami

Tą funkcją możesz przeszukiwać zarówno 16 jak i 32 bitowe pliki dla łańcuchów. Łańcuch może być dowolną etykietę jaka pojawia się w menu lub statyczne lub dynamiczne napisy wewnątrz programu. Weźmy na przykład Windows Explorera. Etykiety są nazwami menu (jak Plik) ale również dynamicznymi menu. Spróbuj kliknąć na pliku .exe i pliku .dll. Zwróćmy uwagę,że menu zmienia się w pop-up menu, które się pojawia i zawsze istnieją (jak Właściwości, Usuń i Zmień nazwę) Menu statyczne są tymi ,które zawsze się pojawia, Dynamiczne są to te które się pojawiają albo nie. Jeśli spróbujesz usunąć plik, pojawia się okno dialogowe prosząc cię o potwierdzenie. Tekst w oknie dialogowym jest zawsze zasobem łańcuchowym. Prawie każda apliakcja ma zasób łańcuchowy. Pierwszym krokiem w wyszukiwaniu ich jest identyfikacja czy program jest 16 lub 32 bitowy. Różnica między nimi jest taka ,że programy 32 bitowe używają łańcuchów Unicode. Ilustruje to poniższy przykład:

* Etykieta "Text" jest równa 5465787400 w standardowym ANSI (16-bit)
* Etykieta "Text" jest równa 540065007800740000000 w Unicode

Różnica jest taka ,że między dwoma literami pojawia się znak null (00). Wszystko co musisz zrobić ro sprawdzić wyszukiwanie Unicode za każdtm razem kiedy chcesz znaleźć łańcuch. (Polecenie Edit -> Find) Hackman automatycznie potraktuje dowolny łańcuch jako ANSI jeśli ta opcja nie jest zaznaczona i jako Unicode jeśli jest zaznaczona. Kiedy znajdziesz łańcuch jaki chcesz

zmodyfikować, pamiętaj: możesz zastąpić go dowolnym łańcuchem równym lub krószym ale nie możesz łatwo wstawić znaki. Powiedzmy ,że łańcuch "Text" musi być zastąpiony przez "Test". Jeśli 32 bitowy,zaznacz Unicode i znajdź go. Potem przejdź do "x" i zamień na "s". Naciśnij przycisk Save z menu Write aby zapisa,c zmiany. Teraz powiedzmy, że chcemy przyciąć tą etykietę. Zamiast "Tekst" chcesz etykietę "ekst" Przejd do T i wpisz e, potem x a potem t. Przejdź do części heksadecymalnej, na liczbę 74 i zmień na 00. To przytnie literowe słowo do słowa 3 literowego:

* “Text”jest zainicjowany 54006500780074000000
* Docelowyt “ext” to będzie 65007800740000000000

W przypadku gdzie działamy z plikami 16 bitowymi wszystko co musimy zrobić to zastąpić wartość szesanstkową znaku przycinając 00. Cały proces moze bbyć dalej uproszczony jeśli umieścimy znak null (00) w dowolnym znaku i zignorujemy resztę:

* “Text” jest inicjalizowany 54006500780074000000
* 65000000740065000000 będzie wyświetlało “e” ponieważ po “e” mamy null.

Musimy być bardzo ostrożni kiedy działamy z menu i innymi łańcuchami, które obsługują akceleratory. Akcelerator jest używany jako skrót klawiaturowy, połżczony z Alt. Weźmyza przykład Windows Explorer. Przy przytrzymaniu Alt naciśnięciu H a potem A. Pojawi się pole About. Litera która odpowiada za menu, przycisk i inne aktywne elemnty pojawią się z podkreśleniem.Zwróc uwagę na podkreśloną literęH w menu Windows Eplorera. Jest stuczka z umieszczeniem akceleratora w zasobach, nawet jeśli akcelerator ie istnieje. Lub możesz ususnąć jeden jeśli już istnieje! Akcelerator pojawia się ze znakiem ampersandu (&) przed literą. Np. Help pojawi się jako &Help a Format z podkreślonym "o" będzie to F&ormat. Zastąpienie F&ormat &Format zastąpi akcelerator "o" na "f"

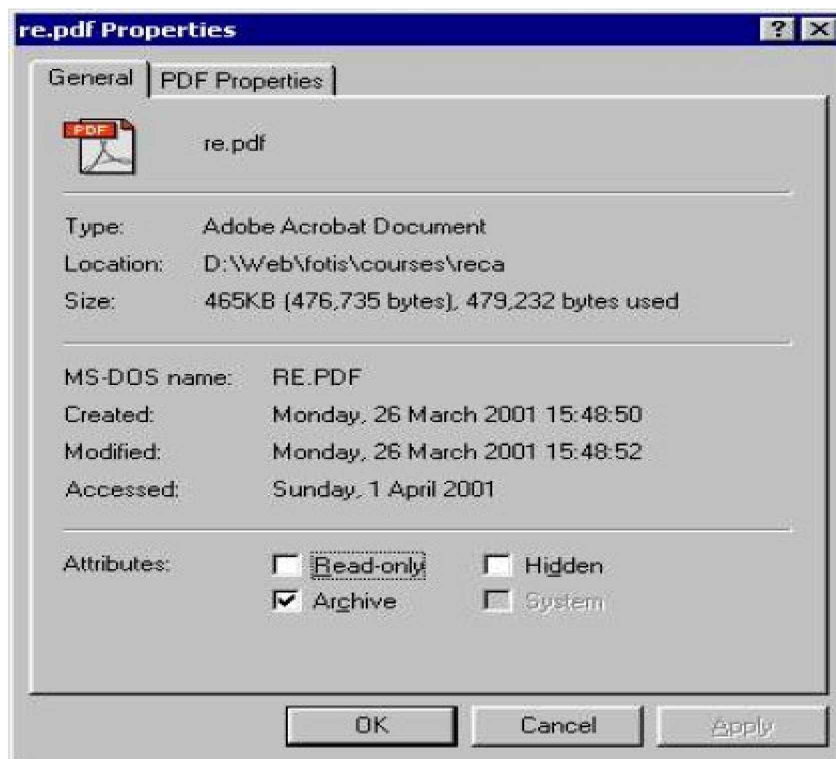
31. Znacznik wersji

Znacznik wersji zazwyczaj istnieje w plikach wykonywalnych i DLL'ach. Aby zobaczyć znacznk wersji pliku , wszystko co musisz zrobić to kliknąć go, potem wybrać Właściwości i klinąć zakładkę Version. Jeśli zakładka Version nie istnieje, wtedy nie ma znacznika wersji.

Aby zmienić znaznik wersji jest łatwe; przejdź do menu Write, Special a potem idź do Version Changer. Jeśli ma już otwarty znacznik wersji, rozszerzenie (Version Changer) automatycznie pozwoli ci edytować wersję łańcucha. Może być to zrobione ręcznie, wyszukaj łańcuchy i zastąp je, ale ten sposób jest szybszy.

32. Zancznik daty

Znacznik daty istniej zazwyczaj dla każdego pliku. Chociaż data io czas utworzenia i ostatniej modyfikacji daty i czasu są logicznie zmanipulowane, czas ostatniego dostępu nie. I to zdarzyło soę ponieważ kiedy klioknąłeś plik i wybrałeś właściwości, faktycznie uzyskałeś do niego dostęp, i dlaytego znacznik ponownie się zmienił. Możesz zmienić znacznik daty dowolnego pliku Hackmanem. Po otwarciu pliku w Hackmanie, przejdź do Write, potem Special a potem wybierz Modify dates



33. Zasoby ikon

Aby wyodrębnić ikonę zasobów z pliku, użyj Icon Xtractor (z menu Write wybierz Icon Xtract) Zastąp ikonę w pliku,, wyodrębnij najpierw ikonę jaką chcesz zastąpić. Potem, otwórz ją w Hackmanie (wyodrębniony plik) i wybierz część z niego (kliknij dwukrotnie dla wyboru, powiedzmy 15, 16 bajtów). Naciśnij Copy. Przejdź do pliku docelowego a potem naciśnij Find. Wklej bajty do znalezionego pola tekstowego. Znajdziesz kod źródłowy ikony w pliku docelowym. Będzie długi na 766 bajtów, jeśli nie jesteś pewny, sprawdź długość wyodrębnionego pliku. Nadpisz

tą informację (usuń ją a potem użyj wstawionego pliku aby wstawić nową ikonędo pliku docelowego)

34. Inne narzędzia

Inne użyteczne narzędzia w pakiecie Hackmana:

- Automatic patch: tworzy dowolne zmiany jakie chcesz, potem używamy Make Patch do stworzenia łatki programu
- Dsiassembler: nie potrzebny opis!
- Decrypt / Encrypt : mocny (do 128 bitów) podprogram kryptograficzny
- Export: eksportuje i filtruje część lub cały kod źródłowy w Javie, C++, Text, HTML, Quickbasic i innych popularnych formatów. Polecenia są dostępne zarówno dla schowka (Copy AS) i bufora pliku (File | Eport)
- Online libraries: naciśnij F1 lub z menu Help wybierz książki online