Szperając w nagłówkach, czyli wstęp do reverse engineeringu

Wojciech Warpechowski

Tworząc na co dzień programy pod Windows pewnie nie zastanawiasz się, jak wyglądają od środka. Instrukcje języka programowania zamieniane są na bajty, które z kolei owocują kolorowymi okienkami i grafiką a niekiedy nawet wydają dźwięki. Informacje o budowie programów są niezbędnym składnikiem w wielu ważnych dziedzinach życia komputerowego. Z namacalnych przykładów można wymienić chociażby programy do zabezpieczania oprogramowania (tzw. *protectory*), programy kompresujące pliki wykonywalne (tzw. *packery*), programy antywirusowe i wiele innych. Poza tym, wiedza ta jest niezbędna do stosowania technik reverse engineeringu, a tym właśnie zajmiemy się w dalszej części artykułu.

Format pliku wykonywalnego Windows

Programy w systemie Windows są przeważnie plikami PE. PE – skrót od *Portable Executable* – jest formatem wprowadzonym przez firmę Microsoft. Posiada on dosyć przejrzystą budowę. Na początku pliku znajduje się nagłówek MZ i DOS stub następnie nagłówki PE i OPT, Data Directory, tabela sekcji oraz same sekcje i jeszcze parę innych struktur. Przyjrzyjmy się im z bliska.

Dodam tylko, że będę w tym artykule operował wielkościami WORD i DWORD. WORD oznacza 2 bajty, natomiast DWORD – 4 bajty.

Nagłówek MZ i DOS stub

Nagłówek MZ jest pozostawiony w celach kompatybilności z systemem MS-DOS. Zauważ, że przy próbie uruchomienia programu dla Windows w systemie MS-DOS, na ekranie pojawia się tekst: *This program must be run under Win32*. Za wypisanie tego komunikatu i zakończenie działania programu odpowiedzialny jest DOS stub. DOS stub jest małym programikiem znajdującym się w każdym pliku PE od razu za nagłówkiem MZ. Jego kod jest przedstawiony na Listingu 1.

Sam nagłówek MZ ma 40h bajtów (patrz Tabela 1.), jednak tylko 2 pierwsze i 4 ostatnie bajty zawierają interesujące informacje, reszta pól jest przestarzała lub zarezerwowana.

Sygnatura MZ zawsze zawiera bajty 4Dh, 5Ah (czyli litery „M” i „Z” w kodzie ASCII, od jednego z twórców tego nagłówka – Marka Zbikowsky'ego).

Autor od 5 lat zajmuje się problematyką zabezpieczeń oprogramowania. Aktualnie jest studentem 3 roku Informatyki na PJWSTK.
Kontakt z autorem: *s3515@pjwstk.edu.pl*

Tabela 1. *Przegląd struktury nagłówka MZ*

Offset względem początku pliku (hex)	Wielkość	Opis
0	WORD	Sygnatura MZ
…	…	…
3C	DWORD	Offset nagłówka PE

Offset nagłówka PE jest to zwykłe przesunięcie względem początku pliku. Jest to bardzo ważne pole, dzięki niemu system operacyjny uruchamiając program, wie gdzie zaczyna się nagłówek PE.

Nagłówek PE

Przegląd przez strukturę nagłówka PE przedstawia Tabela 2. Cały nagłówek ma 18h bajtów. Jest on bardzo ważny, to właśnie tutaj zapisana jest informacja m.in. o ilości sekcji. Poprawny plik PE powinien mieć na początku tego nagłówka DWORD 50450000 co w kodzie ASCII odpowiada ciągowi „PE” zakończonemu dwoma zerami. Bezpośrednio po nagłówku PE następuje nagłówek OPT.

Nagłówek OPT

Ważne, z naszego punktu widzenia, informacje zawarte w nagłówku OPT (skrót od *OPTIONAL*) przedstawia Tabela 3. Cały nagłówek jest dosyć duży, 60h bajtów.

Na początku tego nagłówka znajduje się WORD opisujący typ programu, dla EXE jest to 010Bh.

Następnie pod offsetem 10h znajdziemy Entry Point (w skrócie EP) programu. Jest to adres pierwszej instrukcji programu, zapisany jako RVA. RVA – skrót od *Relative Virtual Address* – jest to relatywny wirtualny adres, czyli adres w pamięci po odjęciu ImageBase. ImageBase, z kolei, jest to adres bazowy pamięci pod który program jest wgrywany podczas uruchamiania. Jego wartość figuruje pod offsetem 1Ch. Dzięki takiemu zapisowi adresu uzyskujemy bardzo prostą

Tabela 2. *Przegląd struktury nagłówka PE*

Offset względem początku nagłówka PE (hex)	Wielkość	Opis
0	DWORD	Sygnatura nagłówka (np. 50450000 = „PE”,0,0)
…	…	…
6	WORD	Ilość sekcji
…	…	…

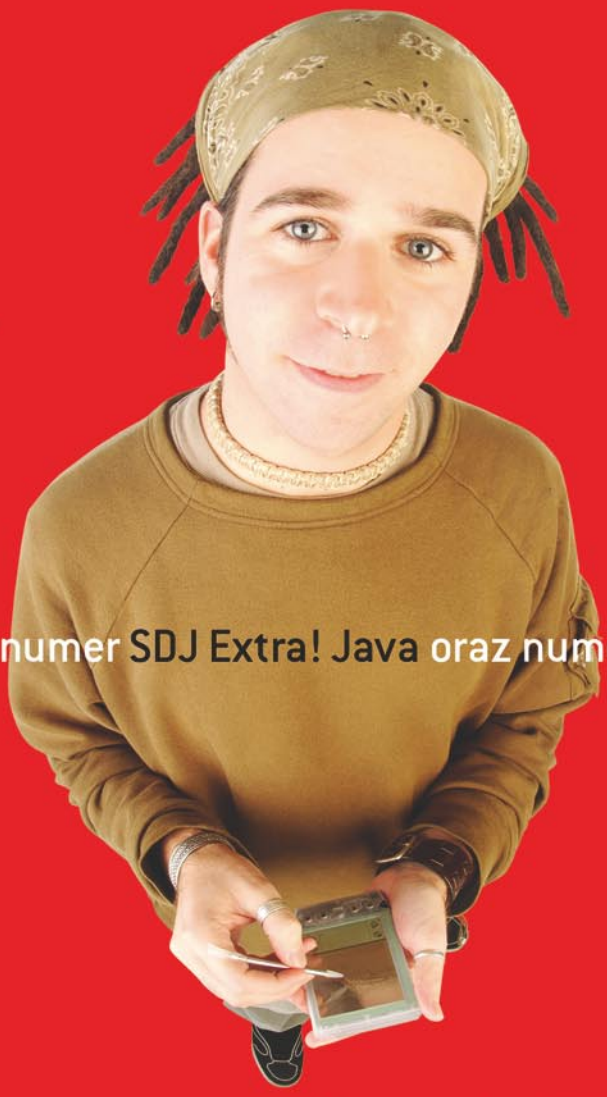
Prenumerata SDJ dla studenta

63% taniej

niż w kiosku

czyli **12** numerów za jedyne **120 zł**

GRATIS!



Dodatkowo otrzymasz numer SDJ Extra! Java oraz numer L+ Extra! Aurox 11.0

Oferta ważna TYLKO do końca listopada.

Roczna prenumerata magazynu Software Developer's Journal:

120 zł dla studentów, oferta ograniczona czasowo

250 zł cena standardowa

Prenumeratę możesz zamówić poprzez sklep internetowy na stronie shop.software.com.pl/student

Zamówienie możesz złożyć również:

- kontaktując się telefonicznie z Działem Prenumeraty **(22) 887-14-44**
- wysyłając fax na numer **(22) 887-10-11**
- za pomocą poczty elektronicznej **pren@software.com.pl**
- wypełniając formularz na stronie internetowej SDJ **www.sdjournal.com**
- poprzez Pocztę Polską i Ruch (prenumerata bez prezentów)

+

Tabela 3. *Przegląd struktury nagłówka OPT*

Offset względem początku nagłówka OPT (hex)	Wielkość	Opis
0	WORD	Typ programu
…	…	…
10	DWORD	Entry Point RVA
…	…	…
1C	DWORD	ImageBase programu
20	DWORD	Wyrównanie w pamięci
24	DWORD	Wyrównanie w pliku
…	…	…
38	DWORD	Wielkość obrazu programu W pamięci
…	…	…

formę „przenośności" (prawdopodobnie stąd „*portable*" w „*Portable Executable*").

Pod offsetem 20h znajduje się DWORD zawierający wielkość wyrównania w pamięci. Jest to bardzo ważne pole, chociażby dlatego, że cały program wgrany do pamięci musi być wielokrotnością tego wyrównania.

Pole zawierające wielkość obrazu programu w pamięci znajduje się pod offsetem 38h. Istnieje również wyrównanie w pliku a jego wartość znajdziemy pod offsetem 24h.

Bezpośrednio po nagłówku OPT następuje struktura zwana Data Directory. Nie będę omawiał jej budowy, gdyż jej znajomość nie jest wymagana do zrozumienia dalszej części artykułu. Powiem jedynie że Data Directory jest wielkości 80h bajtów a bezpośrednio po niej następuje tabela sekcji, której przyjrzymy się dokładniej.

Tabela sekcji

Tabela sekcji składa się z wpisów zawierających informacje o kolejnych sekcjach programu. Ilość wpisów w tabeli określa WORD spod offsetu 6 z nagłówka PE (offset względem początku nagłówka PE). Pojedynczy wpis ma 20h bajtów a jego strukturę przedstawia Tabela 4.

Jak Windows uruchamia programy?

Windows tworzy dla każdej uruchamianej aplikacji osobną przestrzeń adresową. Następnie odczytuje z uruchamianego pliku wartość ImageBase i wgrywa pod ten adres uruchamiany program. Wszystkie sekcje programu, które mają być załadowane do pamięci, umieszczane są pod swoimi adresami wirtualnymi (tzw. VA). Wartości te przechowywane są w pliku w tabeli sekcji. Tabela ta zwiera także wielkość danej sekcji w pamięci. Windows przycina sekcję do tej wielkości lub wypełnia zerami odpowiednią ilość miejsca za sekcją, tak aby zwiększyć jej rozmiar (tzw. *padding*). W ten sposób utworzony został obraz programu w pamięci.

Uruchomienie aplikacji następuje w momencie wykonania jej pierwszej instrukcji. System operacyjny lokalizuje jej adres poprzez odczytanie z pliku wartości Entry Point (EP) i dodanie jej do ImageBase programu.

Po zakończeniu działania aplikacji, Windows zwalnia pamięć zajmowaną przez program.

Tabela 4. *Przegląd struktury wpisu w tabeli sekcji*

Offset względem początku wpisu (hex)	Wielkość	Opis
0	8 bajtów	Nazwa sekcji
8	DWORD	Rozmiar sekcji w pamięci
C	DWORD	RVA sekcji
10	DWORD	Rozmiar sekcji w pliku
14	DWORD	Offset (plikowy) sekcji
…	…	…
1C	DWORD	Flagi sekcji

Nazwa sekcji jest dowolnym ciągiem ośmiu bajtów i nie ma żadnego wpływu na działanie programu, natomiast 4 następne pozycje i pole flag sekcji są bardzo ważne.

Rozmiar sekcji w pamięci określa ilość pamięci zarezerwowanej na daną sekcję, natomiast rozmiar sekcji w pliku określa ilość bajtów przeznaczoną na daną sekcję w pliku programu. Zdziwi Cię zapewne fakt, że rozmiary te są przeważnie różne. Jest tak dlatego, ponieważ obie wielkości są wyrównywane do wielokrotności przyjętego wyrównania w pamięci/pliku, które z kolei też różnią się od siebie. RVA sekcji jest to RVA początku sekcji w pamięci, natomiast offset sekcji jest po prostu miejscem w pliku gdzie zaczyna się dana sekcja.

Flagi sekcji określają jej uprawnienia. Dzięki nim sekcje mogą być do odczytu, zapisu lub wykonania. Ustawienie wszystkich bitów w tym DWORDzie (wartość 0FFFFFFFFh) oznacza nadanie sekcji wszystkich możliwych uprawnień.

Dysponując powyższymi informacjami śmiało możemy przystąpić do budowy programu do zabezpieczania plików PE.

Budujemy protector plików PE

Nasz protector będzie się składał z dwóch części: programu szyfrującego sekcję z kodem oraz programu dodającego do pliku nową sekcję z kodem deszyfrującym oraz modyfikującego wartości w nagłówkach. Nasz protector będzie bardzo prosty, jako szyfru będzie używał xor-owania tekstu jawnego z jednobajtowym, niezmiennym kluczem. Jednak nic nie stoi na przeszkodzie aby poświęcić trochę czasu i zaimplementować jakiś bardziej złożony szyfr.

Szyfrowanie sekcji z kodem

Na początku napiszemy program, nazwijmy go *Prog1*, który:

- otworzy plik *test.exe*,
- sprawdzi czy otwarty plik jest plikiem PE,
- znajdzie sekcję z kodem,

Listing 1. *Standardowy program zawarty w DOS stub*

```
mov   dx, 10h
push   cs
pop   ds
mov   ah, 9h
int   21h
mov   ax, 4C01h
int   21h
db "This program must be run under Win32", 0Dh, 0Ah, "$"
```

Listing 2. *Prog1*

```
#define byte unsigned char
// Stale
#define DELTA_PE_OFFSET 0x3C
#define PE_HEADER_SIZE 0xF8
#define DELTA_NUMBER_OF_SECTIONS 6
#define SECTION_NAME_SIZE 8
#define SECTION_SIZE 0x28
#define DELTA_SECTION_VIRT_SIZE 8
#define DELTA_SECTION_OFFSET 20
// Deklaracje funkcji
void Crypt(byte* pData, int size);
int APIENTRY _tWinMain(
      HINSTANCE hInstance,
      HINSTANCE hPrevInstance,
      LPTSTR    lpCmdLine,
      int       nCmdShow) {
   // Otworz plik
   HANDLE hFile = CreateFile(
      "./test.exe",
      GENERIC_READ | GENERIC_WRITE,
      0,
      NULL,
      OPEN_EXISTING,
      FILE_ATTRIBUTE_NORMAL,
      NULL
   );
   if(hFile==INVALID_HANDLE_VALUE) {
      MessageBox(0, "Nie mogę otworzyć pliku test.exe!",
            "Błąd", MB_ICONERROR);
      return 1;
   }
   // Pobierz offset naglowka PE
   DWORD dwPEOffset, dwTmp;
   SetFilePointer(hFile, DELTA_PE_OFFSET, 0, FILE_BEGIN);
   ReadFile(hFile, &dwPEOffset,
         sizeof(dwPEOffset), &dwTmp, NULL);
   // Sprawdź czy to rzeczywiście plik PE
   WORD wPETest;
   SetFilePointer(hFile, dwPEOffset, 0, FILE_BEGIN);
   ReadFile(hFile, &wPETest, sizeof(wPETest), &dwTmp, NULL);
   if(wPETest!=0x4550) {       // 0x5045 = "PE"
      CloseHandle(hFile);
      MessageBox(0, "To nie jest poprawny plik PE!",
            "Błąd", MB_ICONERROR);
      return 1;
   }
   // Pobierz ilosc sekcji
   WORD wNumberOfSections;
   SetFilePointer(hFile, dwPEOffset+
         DELTA_NUMBER_OF_SECTIONS, 0, FILE_BEGIN);
   ReadFile(hFile, &wNumberOfSections, sizeof(
         wNumberOfSections), &dwTmp, NULL);
   // Znajdź sekcję z kodem (.text lub CODE)
   char SectionName[SECTION_NAME_SIZE+1];
   WORD wSectionCounter = 0;
   do {
      if(wSectionCounter==wNumberOfSections) {
         CloseHandle(hFile);
         return 1;
      }
      ZeroMemory(SectionName, sizeof(SectionName));
      SetFilePointer(
         hFile,
         dwPEOffset + PE_HEADER_SIZE +
               wSectionCounter * SECTION_SIZE,
         0,
         FILE_BEGIN
      );
      ReadFile(hFile, SectionName, SECTION_NAME_SIZE,
            &dwTmp, NULL);
      wSectionCounter++;
   } while(!(strcmpi(SectionName, ".text")==0 ||
         strcmpi(SectionName, "CODE")==0));
   wSectionCounter--;
   // Pobierz wielkość w pamięci sekcji z kodem
   DWORD dwSectionVirtSize;
   SetFilePointer(
      hFile,
      dwPEOffset + PE_HEADER_SIZE + wSectionCounter *
            SECTION_SIZE + DELTA_SECTION_VIRT_SIZE,
      0,
      FILE_BEGIN
   );
   ReadFile(hFile, &dwSectionVirtSize, sizeof(
         dwSectionVirtSize), &dwTmp, NULL);
   // Pobierz offset sekcji z kodem
   DWORD dwSectionOffset;
   SetFilePointer(
      hFile,
      dwPEOffset + PE_HEADER_SIZE + wSectionCounter *
            SECTION_SIZE + DELTA_SECTION_OFFSET,
      0,
      FILE_BEGIN
   );
   ReadFile(hFile, &dwSectionOffset,
         sizeof(dwSectionOffset), &dwTmp, NULL);
   byte* pCode = new byte[dwSectionVirtSize];
   SetFilePointer(hFile, dwSectionOffset, 0, FILE_BEGIN);
   ReadFile(hFile, pCode, dwSectionVirtSize, &dwTmp, NULL);
   Crypt(pCode, dwSectionVirtSize);
   SetFilePointer(hFile, dwSectionOffset, 0, FILE_BEGIN);
   WriteFile(hFile, pCode, dwSectionVirtSize, &dwTmp, NULL);
   delete[] pCode;
   pCode = NULL;
   CloseHandle(hFile);
   hFile = NULL;
   MessageBox(0, "Sekcja z kodem została zaszyfrowana.",
         "OK", MB_ICONINFORMATION);
   return 0;
}
void Crypt(byte* pData, int size) {
   for(int i=0; i<size; i++)
      *(pData+i) ^= 13;
}
```

- zaszyfruje znalezioną sekcję,
- zapisze zmiany w pliku.

Kod programu znajduje się na Listingu 2.

Przeanalizujmy teraz funkcję tWinMain. Na początku program próbuje otworzyć plik *test.exe* ze swojego katalogu. Jeżeli mu się powiedzie, pobiera offset nagłówka PE spod offsetu 3Ch. Następnie sprawdza czy to rzeczywiście jest plik PE. Sprawdzenie polega na teście pierwszego WORD-a z nagłówka PE, który dla plików PE musi zawierać bajty 5045h (czyli "PE" w kodzie ASCII). Jeżeli test się powiedzie, pobierana jest ilość sekcji w pliku. Informacje o tym znajdziemy na szóstym bajcie nagłówka PE. Ilość sekcji będzie nam potrzebna do skonstruowania pętli skanującej tablicę sekcji w poszukiwaniu wpisu o nazwie *.text* lub *CODE*. Mała uwaga praktyczna – wszystkie popularne kompilatory tworzą sekcje z kodem o nazwie *.text* lub *CODE*. Nazwa *.text* jest używana np. w kompilatorze C++ Microsoftu, a *CODE* w Borland Delphi. Niestety każda sekcja (także z kodem) może mieć dowolną nazwę. Ja w przykładowych programach (*Prog1* i *Prog2*) zakładam, że sekcja z kodem ma nazwę *.text* lub *CODE*. W 99 % przypadków to rozwiązanie będzie działać poprawnie. Jednak aby mieć 100 % pewność, należy szukać sekcji także za pomocą charakterystyk. Sekcja z kodem musi mieć w swojej charakterystyce 20h. Najczęściej spotyka się 60000020h.

Po znalezieniu sekcji z kodem musimy pobrać rozmiar tej sekcji w pamięci oraz jej offset plikowy. Następnie rezerwujemy pamięć i wczytujemy sekcję z pliku. Teraz możemy zająć się szyfrowaniem. Zadanie to realizuje funkcja `Crypt`, która xor-uje kolejne bajty kodu z kluczem – bajtem o wartości 13. Zaszyfrowaną sekcję zapisujemy do pliku, nadpisując tym samym jej oryginał. Następnie zwalniamy pamięć i zamykamy uchwyt do pliku.

Uruchom program *Prog1* w katalogu z plikiem *test.exe* (przedtem upewnij się, czy program *test.exe* działa i czy nie ma nałożonego atrybutu *tylko do odczytu*). Jeżeli teraz spróbujesz uruchomić plik *test.exe* Twoim oczom ukaże się komunikat błędu. W tym przypadku oznacza to sukces naszej operacji – kod programu został zaszyfrowany i przy wykonaniu spowodował błąd.

A teraz mała ciekawostka, jeżeli uruchomisz ponownie program *Prog1*, a następnie spróbujesz uruchomić program *test.exe* – program testowy się uruchomi! Spowodowane jest to właściwością stosowanego szyfru. a xor b = c; c xor b = a; Czyli drugie szyfrowanie tym szyfrem (z tym samym kluczem) okazało się być deszyfrowaniem. Program testowy uruchomił się ponieważ został odszyfrowany. Przed uruchomieniem programu *Prog2*, zaszyfruj ponownie plik testowy (czyli uruchom program *Prog1* po raz trzeci).

Naszym kolejnym zadaniem będzie napisanie programu który dopisze tzw. loader do pliku PE.

Konwersja RVA na VA

Większość adresów, w nagłówkach plików PE, przechowywana jest w formie względnych adresów wirtualnych (RVA). Zamiana ich na adresy wirtualne (VA) jest bardzo prosta i ogranicza się do zsumowania RVA i ImageBase (VA = RVA + ImageBase). ImageBase jest bazowym adresem aplikacji – adresem pod który został wgrany program.

Listing 3. *Loader*

```
    pushad                          ; Zachowaj stan rejestrów

    mov     esi, 012345678h             ; ESI = adres danych
                                        ; do deszyfracji
    mov     ecx, 012345678h             ; ECX = wielkość
                                    ; danych do deszyfracji

_go:
    xor     byte ptr [esi+ecx], 0Dh             ; Deszyfruj
    dec     ecx                         ; Zmniejsz licznik o 1
    jnz     _go     ; Jeżeli ECX!=0 to skocz do etykiety _go

    popad                           ; Przywróć stan rejestrów

    push    012345678h                  ; Połóż OEP na stosie
    ret             Pobierz OEP ze stosu i skocz w to miejsce
```

Stworzenie loadera

Loader jest w naszym przypadku kodem umieszczonym w nowej sekcji w pliku PE. Jego zadaniem jest odszyfrowanie kodu właściwego programu przed jego uruchomieniem. Ten szczytny cel możemy osiągnąć przez stworzenie takiego loadera jak na Listingu 3.

Jak widać, loader nie jest skomplikowany. Zbudowany jest z jednej pętli deszyfrującej.

Przyjrzyjmy się jednak dokładniej samej końcówce kodu loadera. Instrukcje `push 012345678h` i `ret` wykorzystaliśmy do wykonania skoku pod adres `012345678h`. Jak to się stało? To proste, instrukcja `push` kładzie na stosie procesora swój argument, w tym przypadku adres `012345678h`, natomiast instrukcja `ret` pobiera ten argument ze stosu i skacze pod pobrany adres.

Zauważ jednak, że w kodzie naszego loadera wszystkie adresy mają wartość `012345678h`, może wydać Ci się to podejrzane… i słusznie. Już spieszę z wyjaśnieniami, otóż kod loadera będzie umieszczany w pliku docelowym przez nasz program, który zaraz napiszemy, a wszystkie wartości `012345678h` zastąpione będą odpowiednimi danymi. Teraz stanowią tylko przykład.

Zatem, mamy już zaszyfrowaną sekcję z kodem i kod loadera który ją nam zdeszyfruje poczym skoczy we wskazany adres. Jedyne czego nam brakuje to program umieszczający loader w zabezpieczanym pliku.

Wszczepianie loadera…

… nie jest rzeczą banalną. Program (*Prog2*), który stworzymy w tym celu musi:

- uzupełnić kod loadera odpowiednimi wartościami,
- utworzyć nową sekcję w zabezpieczanym pliku,
- umieścić w niej kod loadera,
- zmienić niektóre wartości w nagłówkach zabezpieczanego pliku, tak aby system operacyjny zauważył nowododaną sekcję,
- przekierować start programu na początek loadera

Zajmijmy się na początku zebraniem informacji potrzebnych do uzupełnienia wartości w kodzie loadera. Kod przedstawiony jest na Listingu 4.

Listing 4. *Prog2 – zbieranie informacji o pliku PE*

```
#define byte unsigned char
// Stale
#define DELTA_PE_OFFSET 0x3C
#define PE_HEADER_SIZE 0xF8
#define DELTA_NUMBER_OF_SECTIONS 6
#define DELTA_EP 0x28
#define DELTA_IMAGEBASE 0x34
#define DELTA_RVA_ALIGN 0x38
#define DELTA_SIZE_OF_IMAGE 0x50
#define SECTION_NAME_SIZE 8
#define SECTION_SIZE 0x28
#define DELTA_SECTION_VIRT_SIZE 8
#define DELTA_SECTION_FLAGS 0x24
int APIENTRY _tWinMain(
      HINSTANCE hInstance,
      HINSTANCE hPrevInstance,
      LPTSTR    lpCmdLine,
      int       nCmdShow) {
   // Otworz plik
   HANDLE hFile = CreateFile(
      "./test.exe",
      GENERIC_READ | GENERIC_WRITE,
      0,
      NULL,
      OPEN_EXISTING,
      FILE_ATTRIBUTE_NORMAL,
      NULL
   );
   if(hFile==INVALID_HANDLE_VALUE) {
      MessageBox(0, "Nie mogę otworzyć pliku test.exe!",
            "Błąd", MB_ICONERROR);
      return 1;
   }
   // Pobierz offset naglowka PE
   DWORD dwPEOffset, dwTmp;
   SetFilePointer(hFile, DELTA_PE_OFFSET, 0, FILE_BEGIN);
   ReadFile(hFile, &dwPEOffset, sizeof(dwPEOffset),
         &dwTmp, NULL);
   // Sprawdź czy to rzeczywiście plik PE
   WORD wPETest;
   SetFilePointer(hFile, dwPEOffset, 0, FILE_BEGIN);
   ReadFile(hFile, &wPETest, sizeof(wPETest), &dwTmp, NULL);
   if(wPETest!=0x4550) {       // 0x5045 = "PE"
      CloseHandle(hFile);
      MessageBox(0, "To nie jest poprawny plik PE!",
            "Błąd", MB_ICONERROR);
      return 1;
   }
   // Pobierz OEP (Orginal Entry Point - RVA startu
   // programu)
   DWORD dwOEP;
   SetFilePointer(hFile, dwPEOffset+DELTA_EP, 0,
         FILE_BEGIN);
   ReadFile(hFile, &dwOEP, sizeof(dwOEP), &dwTmp, NULL);
   // Pobierz ImageBase
   DWORD dwImageBase;
   SetFilePointer(hFile, dwPEOffset+DELTA_IMAGEBASE,
         0, FILE_BEGIN);
   ReadFile(hFile, &dwImageBase, sizeof(dwImageBase),
         &dwTmp, NULL);
   // Pobierz wielkość wyrównania sekcji w pamięci i w pliku
   DWORD dwRVAAlign, dwFileAlign;
   SetFilePointer(hFile, dwPEOffset+DELTA_RVA_ALIGN,
         0, FILE_BEGIN);
   ReadFile(hFile, &dwRVAAlign, sizeof(dwRVAAlign),
         &dwTmp, NULL);
   ReadFile(hFile, &dwFileAlign, sizeof(dwFileAlign),
         &dwTmp, NULL);
   // Pobierz wielkość obrazu pliku w pamięci
   DWORD dwSizeOfImage;
   SetFilePointer(hFile, dwPEOffset+DELTA_SIZE_OF_IMAGE,
         0, FILE_BEGIN);
   ReadFile(hFile, &dwSizeOfImage, sizeof(dwSizeOfImage),
         &dwTmp, NULL);

   WORD wNumberOfSections;
   SetFilePointer(hFile,
        dwPEOffset+DELTA_NUMBER_OF_SECTIONS, 0, FILE_BEGIN);
   ReadFile(hFile, &wNumberOfSections, sizeof(
         wNumberOfSections), &dwTmp, NULL);
   // Znajdź sekcję z kodem (.text lub CODE)
   char SectionName[SECTION_NAME_SIZE+1];
   WORD wSectionCounter = 0;
   do {
      if(wSectionCounter==wNumberOfSections) {
         CloseHandle(hFile);
         return 1;
      }
      ZeroMemory(SectionName, sizeof(SectionName));
      SetFilePointer(
         hFile,
         dwPEOffset + PE_HEADER_SIZE + wSectionCounter *
               SECTION_SIZE,
         0,
         FILE_BEGIN
       );
      ReadFile(hFile, SectionName, SECTION_NAME_SIZE,
            &dwTmp, NULL);
      wSectionCounter++;
   } while(!(strcmpi(SectionName, ".text")==0 ||
          strcmpi(SectionName, "CODE")==0));
   DWORD dwSectionVirtSize, dwSectionRVA;
   wSectionCounter--;
   SetFilePointer(
      hFile,
      dwPEOffset + PE_HEADER_SIZE + wSectionCounter *
            SECTION_SIZE + DELTA_SECTION_VIRT_SIZE,
      0,
      FILE_BEGIN
   );
   ReadFile(hFile, &dwSectionVirtSize, sizeof(
         dwSectionVirtSize), &dwTmp, NULL);
   ReadFile(hFile, &dwSectionRVA, sizeof(
         dwSectionRVA), &dwTmp, NULL);
```

Listing 5. *Prog2 – utworzenie wpisu opisującego nową sekcję*

```
// Utwórz strukturę opisującą nową sekcję
DWORD dwSDJVirtSize = dwRVAAlign;
DWORD dwSDJFileSize = dwFileAlign;
// Nowa sekcja będzie dopisana na koniec pliku
DWORD dwSDJOffset = GetFileSize(hFile, NULL);
// RVA nowej sekcji to koniec ostatniej sekcji wyrównany
// w pamięci, czyli dotychczasowy rozmiar pliku w pamięci
DWORD dwSDJRVA = dwSizeOfImage;
// Wszystkie uprawnienia
DWORD dwSDJFlags = 0xFFFFFFFF;
byte* pSDJSection = new byte[SECTION_SIZE];
ZeroMemory(pSDJSection, SECTION_SIZE);
// Nazwa sekcji
strcpy((char*)pSDJSection, "SDJ");
// Rozmiar sekcji w pamieci
memcpy(pSDJSection+SECTION_NAME_SIZE, &dwSDJVirtSize, 4);
// RVA
memcpy(pSDJSection+SECTION_NAME_SIZE+4, &dwSDJRVA, 4);
// Rozmiar sekcji w pliku
memcpy(pSDJSection+SECTION_NAME_SIZE+8, &dwSDJFileSize, 4);
// Offset
memcpy(pSDJSection+SECTION_NAME_SIZE+12, &dwSDJOffset, 4);
// Flagi
memcpy(pSDJSection+SECTION_NAME_SIZE+28, &dwSDJFlags, 4);
```

Jak widać kod nie jest skomplikowany, a pobiera nam dużo niezbędnych informacji.

Po otwarciu pliku, pobraniu offsetu nagłówka PE i upewnieniu się, że to poprawny plik PE, program pobiera *Entry Point* programu. W komentarzu w kodzie użyłem nazwy OEP – skrót od *Orginal Entry Point* – aby zaznaczyć, iż jest to oryginalny adres startu programu. Następnie pobierany jest ImageBase, wielkość wyrównania w pamięci/pliku, wielkość obrazu pliku w pamięci oraz ilość sekcji. Potem, tak samo jak w programie *Prog1*, skanujemy tablicę sekcji w poszukiwaniu wpisu o sekcji z kodem (sekcja *.text* lub *CODE*). Po znalezieniu wpisu pobieramy wielkość w pamięci oraz RVA sekcji którą opisuje.

Kolejnym krokiem będzie stworzenie nowego wpisu do tablicy sekcji, opisującego sekcję którą mamy zamiar dodać do pliku. Wpis wykonamy jak na razie tylko w pamięci, zapisywanie do pliku odłożymy na później. Kod przedstawiony jest na Listingu 5.

Program alokuje pamięć potrzebną na strukturę opisującą sekcję pliku PE, następnie wypełnia ją odpowiednimi wartościami. Tutaj należą Ci się pewne wyjaśnienia. Czemu użyłem akurat takich wartości?

Rozmiar sekcji w pamięci wynosi tyle ile wartość wyrównania w pamięci – ponieważ nasz loader jest zaledwie 25-cio bajtowym kodem, na pewno zmieści się w najmniejszym dopuszczalnym rozmiarze, czyli wartości wyrównania w pamięci. To samo dotyczy wielkości sekcji w pliku i wartości wyrównania w pliku.

Offset sekcji w pliku ustawiony jest na aktualny rozmiar pliku, ponieważ nasza sekcja będzie dopisana na jego koniec. Postępując tak mamy pewność, że wartość ta będzie wyrównana do wartości wyrównania w pliku.

RVA naszej sekcji wskazuje na dotychczasową wartość wielkości obrazu programu w pamięci, co jest równoznaczne z adresem końca pliku w pamięci, który z założenia wyrównany jest do wartości wyrównania w pamięci. Flagi dla naszej sekcji ustawiłem na `0FFFFFFFFh`, ponieważ wtedy będziemy mieli na pewno prawo do wykonania jej kodu a to musimy mieć zapewnione.

Nazwa sekcji jest dowolna, zdecydowałem się na SDJ. Jest to skrót od pewnego szacownego źródła wiedzy…

Zajmijmy się teraz dla odmiany czymś prostszym lecz równie niezbędnym – modyfikacją danych w nagłówkach pliku PE. Listing 6 zawiera odpowiedni kod.

Program zapisuje utworzony wcześniej wpis opisujący nową sekcję do tablicy sekcji zaraz za dotychczasową ostatnią sekcją. Następnie zwiększa o 1, wartość ilości sekcji w pliku, oraz ustawia `0FFFFFFFFh` jako flagi, znalezionej wcześniej, sekcji z kodem. Ostatnią modyfikacją jest zwiększenie wartości wielkości obrazu programu w pamięci o wartość wyrównania w pamięci, czyli wielkość nowej sekcji w pamięci. Dzięki temu program (podczas uruchamiania) zostanie wgrany do pamięci w całości – razem z nową sekcją.

Listing 6. *Prog2 – modyfikacja danych w nagłówkach pliku PE*

```
// Zapisz informacje o nowej sekcji w tablicy sekcji
SetFilePointer(
   hFile,
   dwPEOffset + PE_HEADER_SIZE + wNumberOfSections *
         SECTION_SIZE,
   0,
   FILE_BEGIN
);
WriteFile(hFile, pSDJSection, SECTION_SIZE, &dwTmp, NULL);
// Zwiększ ilość sekcji o 1
wNumberOfSections++;
// Zapisz zmodyfikowana ilość sekcji do pliku
SetFilePointer(hFile, dwPEOffset+DELTA_NUMBER_OF_SECTIONS,
      0, FILE_BEGIN);
WriteFile(hFile, &wNumberOfSections, sizeof(
      wNumberOfSections), &dwTmp, NULL);
// Zapisz 0xFFFFFFFF jako flagi sekcji z kodem
// (aby można było do niej pisać)
// Wszystkie uprawnienia
DWORD dwSectionFlags = 0xFFFFFFFF;
SetFilePointer(
   hFile,
   dwPEOffset + PE_HEADER_SIZE + wSectionCounter *
        SECTION_SIZE + DELTA_SECTION_FLAGS,
   0,
   FILE_BEGIN
);
WriteFile(hFile, &dwSectionFlags, sizeof(dwSectionFlags),
      &dwTmp, NULL);
// Zwiększ wielkość obrazu pliku w pamięci o wielkość
// wyrównania sekcji w pamięci
dwSizeOfImage += dwRVAAlign;
// Zapisz zmodyfikowaną wielkość obrazu pliku w pamięci
SetFilePointer(hFile, dwPEOffset+DELTA_SIZE_OF_IMAGE, 0,
      FILE_BEGIN);
WriteFile(hFile, &dwSizeOfImage, sizeof(dwSizeOfImage),
      &dwTmp, NULL);
```

Listing 7. *Prog2 –poprawienie kodu loadera i zapisanie go do pliku*

```
// Przygotuj bufor na kod nowej sekcji
byte* pSDJCode = new byte[dwFileAlign];
/*
Tablica bajtów - code[] - zawiera kod loadera w formie
opcodów (instrukcji procesora, tutaj w formacie
szesnastkowym). Kod ten trzeba jeszcze poprawić -
powstawiać odpowiednie wartości.
*/
byte hex[] = {0x60, \
   0xBE, 0x78, 0x56, 0x34, 0x12, \
   0xB9, 0x78, 0x56, 0x34, 0x12, \
   0x80, 0x34, 0x0E, 0x0D, \
   0x49, \
   0x75, 0xF9, \
   0x61, \
   0x68, 0x78, 0x56, 0x34, 0x12, \
   0xC3
};
// Przepisz kod do bufora i uzupełnij bufor zerami
ZeroMemory(pSDJCode, dwFileAlign);
memcpy(pSDJCode, hex, sizeof(hex));
// Oblicz VA sekcji z kodem i zmniejsz o 1
DWORD dwSectionVA = dwImageBase + dwSectionRVA - 1;
// Zamien OEP RVA na VA
dwOEP += dwImageBase;
// Ustaw odpowiednie dane
memcpy((pSDJCode+2), &dwSectionVA, sizeof(dwSectionVA));
memcpy((pSDJCode+7), &dwSectionVirtSize, sizeof(
      dwSectionVirtSize));
memcpy((pSDJCode+20), &dwOEP, sizeof(dwOEP));
// Zapisz kod w nowej sekcji
SetFilePointer(hFile, 0, 0, FILE_END);
WriteFile(hFile, pSDJCode, dwFileAlign, &dwTmp, NULL);
// Zwolnij pamięć
delete[] pSDJCode;
// Zapisz adres nowej sekcji jako EP
SetFilePointer(hFile, dwPEOffset+DELTA_EP, 0,
      FILE_BEGIN);
WriteFile(hFile, &dwSDJRVA, sizeof(dwSDJRVA),
      &dwTmp, NULL);
// Zamknij plik
CloseHandle(hFile);
hFile = NULL;
MessageBox(0, "Sekcja SDJ została dodana do pliku.
      \nProgram został prosto zabezpieczony ;)", "OK",
      MB_ICONINFORMATION);
return 0;
}
```

Przyszedł w końcu czas na poprawienie wartości w kodzie loadera i zapisaniu go do zabezpieczanego pliku. Kod jest przedstawiony na Listingu 7.

Na początku program alokuje potrzebną ilość pamięci na sekcję, czyści ją zerowymi bajtami oraz kopiuje na jej początek kod loadera. Kod loadera znajduje się w tablicy `hex` zawierającej jego opcody w formacie szesnastkowym.

Następnie program oblicza wirtualny adres (w skrócie VA), czyli adres w pamięci łącznie z ImageBase, sekcji z zaszyfrowanym kodem i dodatkowo zmniejsza go o 1. Dekrementacja adresu spowodowana jest przez algorytm którym posługuje się loader.

Kolejna konwersja z RVA na VA zachodzi dla OEP (czyli Oryginalnego Entry Pointa).

Teraz, mając już te wirtualne adresy, możemy przystąpić do poprawki kodu loadera. W tym celu wstawiamy w odpowiednie miejsca (offsety względem początku kodu loadera: 2, 7 i 20) DWORD-y z odpowiednimi danymi. Najpierw zapisujemy odpowiedni VA sekcji z kodem do deszyfracji, potem wielkość sekcji w pamięci i na koniec adres startu oryginalnego kodu, czyli OEP VA.

Po dokonaniu tych zmian nasz loader jest gotowy do wszczepienia. Zapisujemy go więc razem z całą nową sekcją na koniec zabezpieczanego pliku, zwalniamy pamięć i zamykamy plik.

Nadeszła pora na test generalny… uruchom program *Prog2* w katalogu z wcześniej przygotowanym (zaszyfrowanym) plikiem *test.exe*. Jeżeli nie wystąpiły żadne błędy, program testowy został zabezpieczony. Spróbuj go teraz uruchomić… to działa!

Podsumowanie

Wspólnymi siłami udało nam się stworzyć naprawdę ciekawą rzecz… w pełni działający program z zaszyfrowanym kodem. Pozwól, że przytoczę teraz główne punkty stworzenia zabezpieczonego pliku PE:

- zaszyfrowanie jednej lub więcej sekcji (np. z kodem),
- dodanie loadera (np. jako osobną sekcję),
- po wykonaniu kodu loadera, skok do oryginalnego początku programu,
- modyfikacja odpowiednich danych w nagłówkach pliku

Zachęcam do dalszych eksperymentów, w razie problemów zachęcam do mailowej korespondencji.

Warto zauważyć, że podobnym schematem posługują się packery ale także wirusy plikowe. Dlatego czasem dochodzi do sytuacji, w której oprogramowanie wirusowe traktuje zmodyfikowany plik jak wirusa. Jako przykład może posłużyć bardzo dobry packer FSG i antywirus MSK_VIR.

Więcej informacji o budowie plików PE można znaleźć w Internecie, szczególnie polecam dokumenty:

- *EXE – opis pliku EXE PE v1.1* (format txt) – opis plików PE autorstwa ufcia: *http://www.ufcio.prv.pl/*
- *Portable Executable File Format Compendium V11 By Goppit* (format chm) – opis plików PE autorstwa Goppit'a z grupy ARTeam: *http://cracking.accessroot.com/*
- *Iczelion's PE Tutorials* – seria bardzo dobrych, praktycznych tutoriali autorstwa Iczelion'a: *http://win32asm.cjb.net/.*

Polecam także rzucić okiem na bardzo dobry protector – PESpin, autorstwa *cyberboba*. Program można pobrać z oficjalnej strony: *http://pespin.w.interia.pl/.*

Wspomniany packer – FSG, autorstwa *barta* z grupy *xtreeme*, można ściągnąć ze strony: *http://woodmann.com/bart/xt/.* ■