Dodawanie funkcji do gotowych plików binarnych

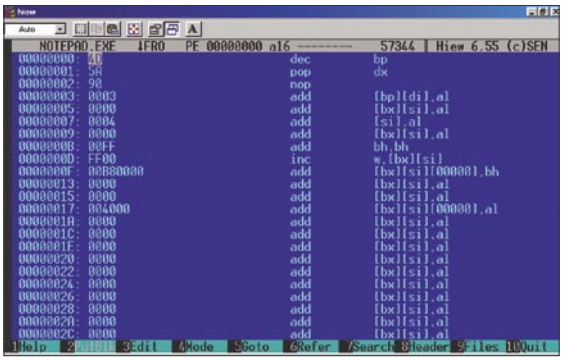
Jakub Nowak

Czym jest *reverse-engineering*? W świecie komputerów to nic innego jak dokładne badanie programów komputerowych, dokładne poznawanie ich struktury oraz zastosowania poprzez śledzenie oraz dogłębną analizę kodu. Ma ono wiele odmian oraz zastosowań w praktyce. *Reversingiem* często nazywane jest np. odwrócenie danej funkcji programu, odwrócenie danego algorytmu (np. kryptograficznego). Poprzez *reverse engineering* możemy również rozumieć pewną modyfikację już istniejącego kodu w postaci pliku binarnego np. poprzez dodanie do niego pewnych nowych funkcji bądź też usunięcie innych już istniejących. O tym właśnie będzie traktował ten artykuł. Postaramy się dodać pewną funkcję do istniejącego już pliku EXE.

Być może wydaje się to bezcelowe, aby dodawać nowe funkcje do już istniejącego, skompilowanego kodu. Aby usprawnić program wystarczyłoby dopisać przecież odpowiedni kod w źródłach danego programu a następnie skompilować go ponownie. Co jednak zrobić, gdy nie posiadamy źródeł danego programu? Gdy posiadamy tylko gotowy plik wykonywalny? Otóż wtedy niezbędna jest ingerencja w już gotowy kod. W jaki sposób to zrobić? W jaki sposób móc modyfikować skompilowany już kod w postaci gotowego pliku EXE? Tutaj przyda nam się znajomość języka *Asembler.* Jak wiadomo, każdy kompilator czy to języka *C++* czy to np. *Delphi* tworząc plik, zamienia istniejące instrukcje danego języka na język niskopoziomowy czyli asembler. Instrukcje asemblera są prezentowane w postaci heksadecymalnej, a te z kolei przekształcane są na postać binarną czyli zero-jedynkową. W takiej właśnie postaci instrukcje interpretowane są przez procesor. Każdy gotowy plik wykonywalny możemy zdesasemblować lub też zdebugować, dzięki czemu będziemy mogli ujrzeć jego kod w postaci instrukcji asemblera. W tym języku również możemy dodać do pliku odpowiednie funkcje.

Aby móc dodać cokolwiek do istniejącego już programu niezbędna jest znajomość języka asembler w stopniu chociaż podstawowym. Ponadto modyfikowany będzie plik wykonywalny o rozszerzeniu EXE, więc niezwykle przydatna jest również znajomość budowy plików w formacie PE.

Autor jest studentem. Pasjonuje się *reverse-engineeringiem* oraz technikami zabezpieczeń w programach. Jest autorem artykułu „Ochrona programów dla Windows przed crackerami" dla magazynu Hackin9
Kontakt z autorem: *jakub-nowak@o2.pl*



Rysunek 1. *Nasz notatnik w hexedytorze*

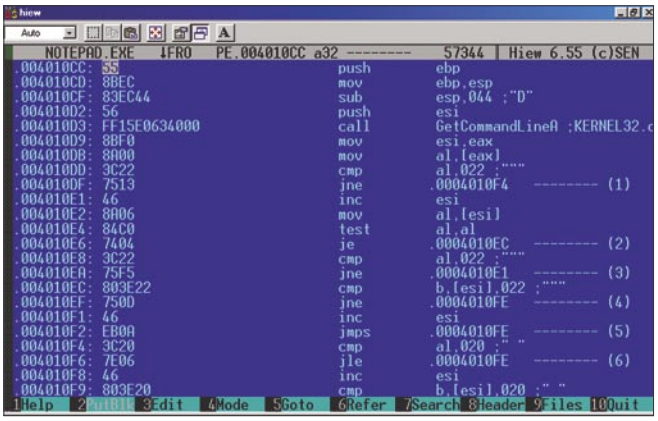
Krótki opis formatu PE

Czym jest format PE? Nazwa PE to skrót od *Portable Executable*, i określa ona sposób kodowania plików wykonywalnych, zaprezentowany przez firmę Microsoft dla systemów operacyjnych serii Win32. Format pliku PE w systemie Windows po raz pierwszy został przedstawiony wraz z systemem operacyjnym Windows NT, w wersji 3.1. Specyfikacja formatu PE pochodzi z systemu UNIX i jest znana jako *COFF* (*Common Object File Format*). Znaczenie *Portable Executable*, czyli przenośny plik wykonywalny, oznacza kompatybilność między platformami x86, MIPS, Alpha. Każda z tych architektur posiada różne kody instrukcji, ale najistotniejszy okazuje się tutaj fakt, że programy ładujące oraz ich 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.

Struktura pliku PE zorganizowana jest jako ciągły strumień danych, plik rozpoczyna się od nagłówka MS-DOS, kodu real-mode, który wykonywany jest w trybie MS-DOS i najczęściej informuje, że aplikacja została napisana dla systemu Win32. Po tym kodzie, zapisany jest nagłówek PE, następnie opcjonalny nagłówek PE oraz tabela zawierająca nagłówki wszystkich sekcji pliku EXE. Po tabeli nagłówków sekcji, zapisana jest struktura zawierająca dodatkowe informacje, między innymi o położeniu tabeli relokacji, tabeli importów itp., po tej strukturze zapisane są sekcje, które mogą zawierać kod, dane programu, dane o importowanych funkcjach, relokacje, zasoby itp.

Struktura pliku PE opiera się zasadniczo na nagłówkach, które stanowią o całej strukturze pliku PE.

Oczywiście jest to tylko pobieżny opis specyfikacji PE. Dokładne opisanie tego formatu byłoby zadaniem na osobny artykuł z uwagi na obszerność. Polecam dokładniejszą lekturę tego formatu w internecie (chociażby na stronach Microsoft).



Rysunek 2. *Entry Point notatnika*

Modyfikacja pliku notepad.exe

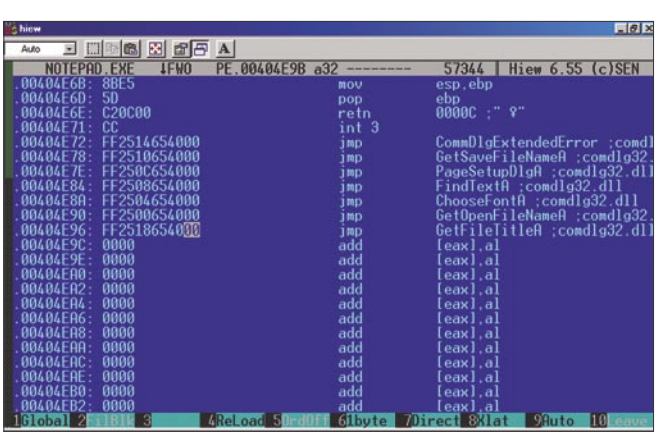
Tyle teorii. Pora na praktykę. Co więc będzie naszym celem? Otóż spróbujemy zmodyfikować plik *notepad.exe* (windowsowy notatnik) tak, aby przed jego uruchomieniem ukazał się nam *MessageBox*. Może nam mówić np. o tym, że to my dokonaliśmy zmian w tymże pliku.

Jakie będą nam potrzebne narzędzia? Do zmian jakie chcemy dokonać w zupełności wystarczy zwykły hexedytor. Na potrzeby tego artykułu będę korzystał z hexedytora HIEW. Jest idealny do naszego zadania, możemy w nim widzieć kod w postaci instrukcji asemblera, dopisać nasz własny oraz zapisać zmiany. Ponadto przydatny może być jakiś analizator plików PE. Doskonałym narzędziem do tego jest LordPE. Nie należy oczywiście zapominać o disasemblerze. Będzie on nam potrzebny do szybkiego poruszania się w kodzie oraz znajdowania odpowiednich wywołań danych funkcji. Użyjemy do tego disasemblera o nazwie Win32Dasm.

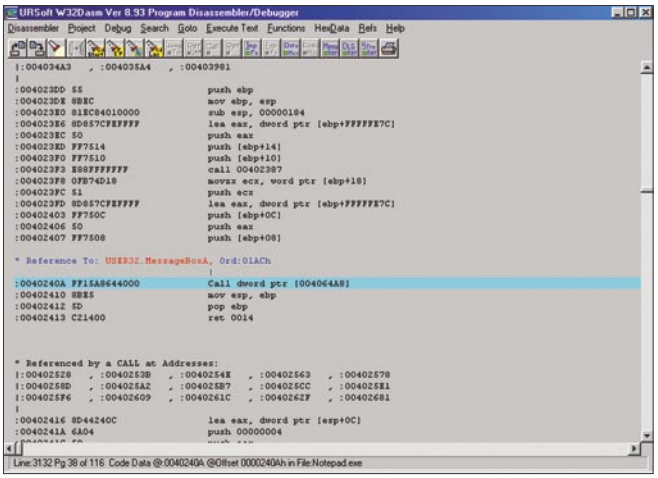
Spróbujmy otworzyć więc nasz notatnik w HIEW. Widok jaki ujrzymy prezentuje Rysunek 1.

Ustalenie Entry Point programu

Mamy więc załadowany plik w hexedytorze. Pierwszą czynnością jaką powinniśmy teraz wykonać, jest dostanie się do tzw. *Entry Point* programu czyli miejsca, w którym zaczyna się wykonywanie kodu programu. Inaczej mówiąc *Entry Point* to po prostu miejsce (adres) wykonania pierwszej instrukcji programu. Aby się do niego dostać należy w HIEW wcisnąć klawisz [*F8]* na klawiaturze (pokaże nam się wtedy cała specyfikacja załadowanego w hexedytorze pliku) a następnie klawisz [*F5]*. Znajdziemy się w miejscu, które ukazuje Rysunek 2.



Rysunek 3. *Nieużywane bajty w kodzie programu*



Rysunek 4. *Funkcja MessageBoxA*

To tutaj startuje program, to tutaj rozpoczyna on wykonywanie swojego kodu. Jak widać jest to *offset* 4010CC. *Offset* to nic innego jak rzeczywista odległość danego bajtu od początku pliku wyrażona w bajtach. Inaczej mówiąc określa fizyczne miejsce bajtu w danym pliku.

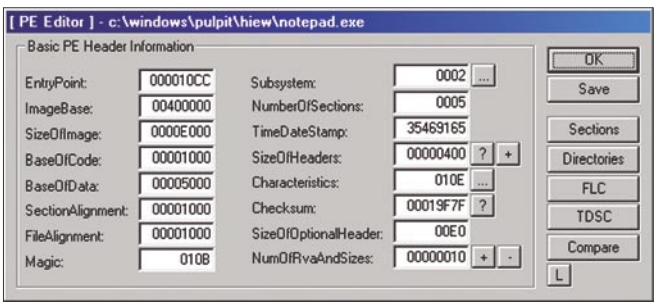
Wiemy już zatem gdzie startuje nasz program. Wartość ta jest nam niezbędna, ponieważ naszym celem jest dodanie *MessageBoxA* na samym starcie programu, stąd musimy wiedzieć gdzie ma on swój początek, aby później móc odpowiednio zmodyfikować kod.

Ustalenie wolnych bajtów na dodanie kodu

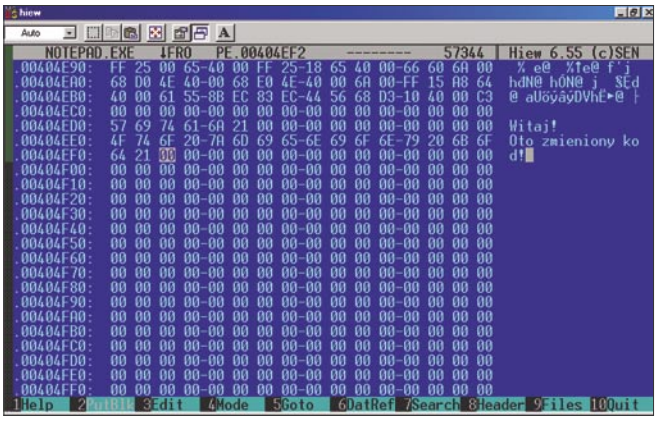
Naszym następnym celem jest poszukanie miejsca, w którym będziemy mogli dodać nasz kod. Musimy więc poszukać bajty, które nie są wykorzystywane w programie. Nieużywane bajty są odznaczone w kodzie jako 0. Jak wiadomo każda instrukcja asemblera ma swoją postać w hexach.

Gdzie są więc takie nieużywane bajty? Przeważnie znajdują się pod koniec odpowiednich sekcji pliku PE. Wystarczy jednak, że w HIEW wyszukamy je za pomocą funkcji „Search". Wciskamy klawisz [*F7]* i wpisujemy długi ciąg zer. Im więcej wpiszemy tym większe prawdopodobieństwo na znalezienie odpowiedniego miejsca, gdzie znajduje się naprawdę sporo nieużywanych bajtów. Ja wpisałem ciąg zer do końca i znalazłem się w miejscu, które wskazuje Rysunek 3.

Nasze wolne miejsce zaczyna się pod *offsetem* 404E9C. Jak widać znajdujemy się w sytuacji komfortowej, ponieważ wolnego miejsca jest bardzo dużo co gwarantuje nam swobodę w dopisywaniu naszego nowego kodu. Przy okazji można tutaj wspomnieć, iż każdy program posiada pewne nieużywane bajty. Raz jest ich mniej, raz jest ich więcej, ale fakt faktem występują one zawsze w każdym programie, niezależ-



Rysunek 5. *Informacje o nagłówku notepad.exe*



Rysunek 4. *Teksty w postaci hexadymecalnej*

nie od kompilatora. Nieważne jest czy program napisany był w *C++*, *Delphi*, czy nawet w języku niskopoziomowym jakim jest *Asembler*. Oczywiście w programie napisanym w języku asembler będzie ich o wiele wiele mniej niż np. w Delphi. Wszystko uwarunkowane jest od kompilatora oraz pewnej struktury programu.

Wyszukanie wywołania funkcji MessageBoxA w programie

Mamy więc miejsce startu programu, mamy również miejsce, w którym możemy dopisać nasz kod. Naszym następnym celem jest wyszukanie w programie wywołania funkcji API *MessageBoxA*. To właśnie tą funkcję chcemy dopisać, więc musimy odnaleźć jakieś jej wywołanie w kodzie. Jest to nam potrzebne do spisania odpowiednich bajtów, które reprezentują wywołanie *MessageBoxA*. Innymi słowy musimy dowiedzieć się jak funkcja ta wygląda w postaci trybu szesnastkowego.

Wyszukiwanie funkcji API w programie (o ile nie jest on spakowany *packerem* bądź też zabezpieczony *protectorem*) jest bardzo łatwe przy pomocy dowolnego disasemblera. My użyjemy do tego programu o nazwie Win32Dasm. Jest to już dość stary jak na dzisiejsze czasy disasembler (obecnie bardzo popularnym środowiskiem do disasemblacji programów jest IDA), jednakże do naszych celów będzie doskonały ze względu na swoją prostotę obsługi.

Tabela 1. *Ogólna struktura pliku PE*

Nagłówek MS-DOS
MS-DOS stub
Nagłówek PE
Opcjonalny nagłówek PE
PE Data Directory
Nagłówek sekcji 1
Nagłówek sekcji 2
Nagłówek sekcji 3
...
Nagłówek sekcji n
Dane sekcji 1
Dane sekcji 2
Dane sekcji 3
...
Dane sekcji n

Listing 1. *Entry Point notatnika*

```
004010CC      55                push ebp
004010CD      8BEC              mov ebp, esp
004010CF      83EC44            sub esp, 044
004010D2      56                push esi
004010D3      FF15E0634000      call GetCommandLineA
```

Otwórzmy więc nasz notatnik w disasemblerze. Po załadowaniu wybierzmy ikonkę importów i wyszukajmy naszego *MessageBoxA*. Znajdziemy go w bibliotece *User32.dll*. Klikamy na niego dwa razy i ujrzymy kod który prezentuje Rysunek 4.

Podświetlony CALL to jest właśnie nasz *MessageBox*. Jak widać obok po lewej stronie znajduje się jego adres oraz to co najbardziej nas interesuje czyli jego postać hexadymecala. Widać wyraźnie, iż opkod (numer instrukcji assemblera wyrażony liczbą w systemie szesnastkowym) funkcji *MessageBoxA* wynosi FF15A8644000. Jest to ważne dla naszych dalszych działań. Gdy mamy już wszystkie potrzebne dane, czyli *Entry Point* programu, adres gdzie znajdują się nieużywane bajty oraz opkod funkcji *MessageBoxA* możemy przystąpić do modyfikacji kodu.

Zamiana oryginalnego Entry Pointu na wolne miejsce w kodzie

Jak wiemy celem jest dodanie naszego *MessageBoxA* na samym starcie programu. Musimy więc przekierunkować oryginalny *Entry Point* który znajduje się pod *offsetem* 4010CC na *offset* 404E9C czyli miejsce, gdzie rozpoczynają się nieużywane bajty. Można to zrobić na dwa sposoby.

Podmiana Entry Pointu w edytorze PE

Pierwszym sposobem jest zmiana *Entry Pointu* przez odpowiedni program do plików PE. Jednym z najpopularniejszych i najstabilniejszych jest LordPE. Ma on wiele zastosowań, ale nas najbardziej interesuje PE Editor, dzięki któremu możemy dowiedzieć się wszystkich potrzebnych informacji o nagłówku PE w danym pliku. Otwieramy więc nasz notatnik w LordPE, a w nim korzystamy z funkcji PE Editor. Informacje o nagłówku naszego pliku obrazuje Rysunek 5. Naszym celem jest zmienienie *Entry Pointu* na *offset* do naszych nieużywanych bajtów. W polu *Entry Point* zmieniamy więc wartość 10CC na nasze 4E9C. Zachowujemy zmiany i wciskamy OK. W tym momencie przenieśliśmy *Entry Point* do naszego miejsca.

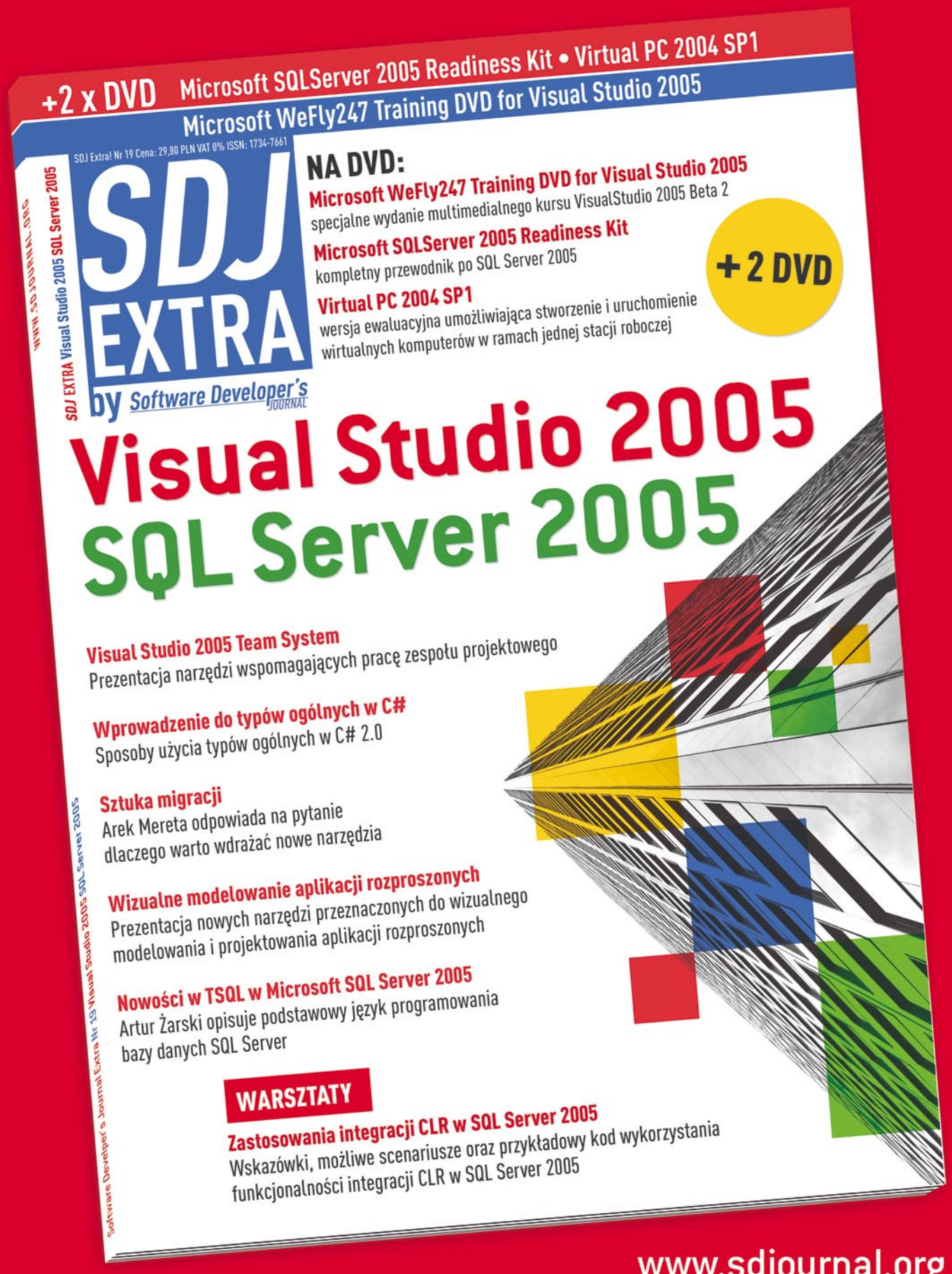
Zamiana Entry Pointu poprzez nadpisanie pierwszych bajtów

Druga metoda zamiany *Entry Pointu* jest trochę bardziej skomplikowana oraz co ciekawe bardzo często wykorzystywana przez różnego rodzaju wirusy. Mianowicie polega ona

Listing. 2 *Przerobiony kod Entry Pointu*

```
004010CC      B89C4E4000        mov eax, 000404E9C
004010D1      FFE0              jmp eax
004010D3      FF15E0634000      call GetCommandLineA
```

Już w sprzedaży



www.sdjournal.org

Pismo dostępne także w sklepie www.shop.software.com.pl

Listing 3. *Struktura wywołania funkcji MessageBox*

```
int MessageBox(
    HWND hWnd,
    LPCTSTR lpText,
    LPCTSTR lpCaption,
    UINT uType
);
```

na nadpisaniu pierwszych bajtów naszym kodem, który skacze do żądanego miejsca. Następnie w miejscu do którego skoczyliśmy wykonujemy naszą funkcję, przywracamy nadpisane bajty oraz powracamy do oryginalnego początku programu. Załadujmy więc jeszcze raz do HIEW nasz notatnik oraz wyszukajmy ponownie jego *Entry Point*. Jego kod obrazuje Listing 1.

Offset 4010CC jest miejscem startu naszego programu. Musimy więc w tym miejscu zastąpić oryginalny kod naszym, który skoczy pod *offset* 404E9C czyli do naszych pustych bajtów. Zamienimy więc pierwsze bajty z Listingu 1 na nasz kod, który wykona skok. Nachodzimy więc w HIEW na pierwszą instrukcję spod *offsetu* 4010CC i wciskamy klawisz [*F3]* a następnie [*F2]* na klawiaturze. Dzięki temu możemy zastąpić oryginalny kod naszym (oczywiście w postaci mnemoników asemblera). Przerobiony kod prezentuje Listing 2.

Porównując Listingi 1 oraz 2 zauważymy, iż cztery oryginalne instrukcje zostały zastąpione przez dwie nowe, które powodują skok do nieużywanych bajtów. Metoda nadpisywania oryginalnych bajtów w celu umieszczenia kodu skaczącego do wolnej przestrzeni jest właściwie metodą niezbędną przy wszelkich modyfikacjach kodu. W przypadku zmiany *Entry Pointu* nie musimy ingerować w kod (metoda pierwsza), jednakże gdybyśmy chcieli wykonać skok do nieużywanych bajtów gdzieś w środku programu metoda druga jest wręcz niezbędna i po prostu konieczna.

Przy zmianie *Entry Pointu* poprzez nadpisanie kodu należy pamiętać, iż jak to jest wspomniane wcześniej, metoda ta jest bardzo często stosowana przez różnego rodzaju wirusy, więc bardzo prawdopodobne jest, iż po takiej zmianie niektóre programy antywirusowe mogą (choć nie muszą) całkowicie mylnie uznać nasz zmodyfikowany program za wirus.

Dopisywanie kodu odpowiedzialnego za MessageBoxA

Dzięki zmianie *Entry Pointu* (czy to metodą pierwszą czy to metodą drugą) znajdziemy się pod *offsetem* 4049EC. Mo-

Listing 4. *Wywołanie MessageBoxA w postaci instrukcji asemblera*

```
push    0               ; styl okna (0 oznacza sam przycisk
                        ; OK)
push    offset belka    ; adres tekstu na belce okna
push    offset tekst    ; adres tekstu głównego
push    0               ; uchwyt okna nadrzędnego, 0 = brak
call      MessageBoxA   ; wywołanie naszego MessageBoxA
```

Listing 5. *Końcowy kod dla zmiany Entry Pointu przez LordPE*

```
pusha               ; odkładamy na stos wszystkie rejestry
                    ; przed naszymi zmianami
push    0           ; styl okna (0 oznacza sam przycisk OK)
push    404ED0      ; adres tekstu na belce okna
push    404EE0      ; adres tekstu głównego
push    0           ; uchwyt okna nadrzędnego, 0 = brak
call    MessageBoxA ; wywołanie naszego MessageBoxA
                    ; (zapisujemy w postaci hexów)
popad               ; przywrócenie rejestrów odłożonych
                    ; wcześniej przez instrukcje pusha
push    4010CC      ; skok do oryginalnego Entry Pointu
retn                ; zwrócenie adresu do EIP i powrót
                    ; w wskazane miejsce
```

żemy zabrać się zatem za dopisywanie naszego kodu, który odpowiedzialny będzie za wyświetlenie *MessageBoxA*. Przypomnijmy więc jak wygląda wywołanie tejże funkcji. Największą kopalnią wiedzy w internecie na temat WinAPI czy też innych rzeczy dotyczących programowania na platformę Windows jest MSDN. Z niego też pochodzi następujący opis funkcji *MessageBox*, który zaprezentowany jest w Listingu 3.

HWND hWnd to nic innego jak uchwyt okna nadrzędnego. *LPCTSTR lp Text* to *offset* do naszego tekstu jaki chcemy pokazać. *LPCTSTR lpCaption* to *offset* do tekstu, który będzie znajdował się na belce naszego *MessageBoxA*. *UINT uType* oznacza jaki styl ma mieć nasz *MessageBox*. My skorzystamy ze zwykłego okna z jednym przyciskiem OK. Jest to domyślna postać okna i wystarczy, że przypiszemy tutaj wartość 0. Wiemy już więc jak wygląda schemat naszego *MessageBoxA*. Teraz zapiszemy to w postaci asemblera. Prezentuje to Listing 4.

Listing 6. *Końcowy kod dla zmiany Entry Pointu przez nadpisanie bajtów*

```
pusha               ; odkładamy na stos wszystkie rejestry
                    ; przed naszymi zmianami
push    0           ; styl okna (0 oznacza sam przycisk OK)
push    404ED0      ; adres tekstu na belce okna
push    404EE0      ; adres tekstu głównego
push    0           ; uchwyt okna nadrzędnego, 0 = brak
call    MessageBoxA ; wywołanie naszego MessageBoxA
                    ; (zapisujemy w postaci hexów)
popad               ; przywrócenie rejestrów odłożonych
                    ; wcześniej przez instrukcje pusha
push    ebp         ;
mov     ebp, esp    ; przywrócenie nadpisanych na początku
                    ; programu instrukcji
sub     esp, 44     ;
push    esi         ;
push    4010D3      ; adres miejsca do którego musimy
                    ; wrócić
retn                ; zwrócenie adresu do EIP i powrót
                    ; w wskazane miejsce
```

Instrukcje `push` odkładają odpowiednie parametry oraz wartości na stos, które są potrzebne do wywołania *MessageBoxA*, instrukcja `call` natomiast wywołuje samo okno. Mamy więc kod wywołujący nasze okienko, jednakże nie mamy tekstu, który ma nam owo okienko pokazać. Potrzebny nam będzie tekst na belce okna oraz tekst główny czyli w środku okienka. Pytanie tylko gdzie umieścić te teksty. Oczywiście musimy zrobić to pod *offsetami*, w których znajdują się wolne bajty. Ponadto musimy znać wartości w hexach naszych napisów. My użyjemy tekstu *„Witaj!”* na belce okna oraz *„Oto zmieniony kod!”* w treści boxa. Tekst *„Witaj!”* w postaci hexadymecalnej ma postać 576974616A21, natomiast tekst *„Oto zmieniony kod!”* prezentuje się jako 4F746F207A6D69656E696F6E79206B6F6421. Wolne miejsce dla kodu naszego boxa jak wiadomo rozpoczyna się pod *offsetem* 4049EC. My poszukamy jeszcze niżej trochę miejsca dla naszych napisów. Pod *offsetem* 404ED0 możemy wpisać hexy dla tekstu *„Witaj!”*, pod offsetem 404EE0 natomiast hexy dla *„Oto zmieniony kod!”*. Całość prezentuje Rysunek 5.

Mamy już więc w programie odpowiednie teksty pod odpowiednimi *offsetami*. Możemy zatem w końcu wpisać nasz kod *MessageBoxA* rozpoczynając od *offsetu* 4049EC. Przy pisaniu kodu należy pamiętać, aby instrukcję `call MessageBoxA` wpisać jako ciąg hexów, które wcześniej odnaleźliśmy. Pozostały kod można wpisywać jako mnemoniki asemblera. Listing 5 prezentuje kod, który musimy wpisać, gdy zmienialiśmy *Entry Point* poprzez LordPE, Listing 6 natomiast kod, gdy nadpisywaliśmy bajty. Należy jeszcze zwrócić uwagę, iż w obu listingach różne są adresy powrotu do oryginalnego *Entry Pointu*. Jest to spowodowane tym, że w przypadku nadpisania bajtów zmienia się struktura oryginalnego kodu, przez co zmienia się również miejsce do którego musimy powrócić.

Gotowe. Po zapisaniu zmian i uruchomieniu ponownie notatnika wyskoczy nam nasz MessageBox!

Pozostała nam jednak jeszcze jedna kwestia do omówienia. Mianowicie, aby dodać nasz *MessageBox* na początek programu skorzystaliśmy z tego, iż funkcja ta była już zaimplementowana w programie tyle tylko, że w innym miejscu. Dzięki temu wystarczyło, iż spisaliśmy reprezentacje heksadecymalną *MessageBoxA*, aby móc go swobodnie użyć do naszych celów. Powstaje jednak pytanie co zrobić w przypadku, gdy program, który chcemy modyfikować nie importuje funkcji (odpowiedniego API) które jest nam niezbędne?

Listing 7. *Importowanie funkcji API*

```
push  offset biblioteka  ; offset do nazwy biblioteki,
                         ; w której jest potrzebna nam API
                         ; (np. User32.dll)
call  LoadLibraryA       ; wywołanie LoadLibraryA,
                         ; w rejestrze EAX znajduje sie
                         ; teraz uchwyt potrzebny dla
                         ; funkcji GetProcAddress
push  offset API         ; offset do potrzebnej nam API
                         ; np. MessageBoxA
push  eax                ; uchwyt z LoadLibraryA
call  GetProcAddress     ; wywołanie GetProcAddress,
                         ; w rejestrze EAX znajduje się
                         ; teraz potrzebna nam API
```

W sieci:

- *http://www.softpedia.com/get/Programming/File-Editors/LordPE.shtml*
- *http://www.softpedia.com/get/Programming/File-Editors/Hiew.shtml*
- *http://www.softpedia.com/get/Programming/Debuggers-Decompilers-Dissasemblers/WDASM.shtml*
- *http://msdn.microsoft.com/*
- *http://www.reteam.org/tools/tf23.zip*

Tutaj sprawa się trochę komplikuje, gdyż musimy sami zaimportować potrzebną nam funkcję do programu..

Importowanie nowych funkcji do programu

Aby zaimportować jakąkolwiek funkcję API musimy znać dwie rzeczy. Po pierwsze to oczywiście nazwa danego API (np. `MessageBoxA`) oraz po drugie bibliotekę z jakiej będzie ona importowana. W przypadku `MessageBoxA` jest to biblioteka `User32.dll`. Aby sprawdzić, która API znajduje się w której bibliotece można skorzystać z jakiegoś opisu funkcji WinAPI, których jest bardzo dużo w sieci. Gdy już znamy te dwie podstawowe kwestie musimy zapoznać się bliżej z dwiema funkcjami, dzięki którym „wszczepimy” naszą API do programu. Pierwsza to `LoadLibraryA` druga to `GetProcAddress`. Te dwie funkcje importowane są w prawie każdym programie. Dzięki nim możemy załadować naszą nową API do programu. Strukturę obu funkcji możemy sprawdzić sobie na stronach MSDN. Listing 7 przedstawia obie funkcje zaimplementowane już w postaci języka asembler.

Oczywiście pod *offsetami* musimy wpisać adres do nazwy odpowiedniej biblioteki oraz odpowiedniej API czyli w naszym przypadku `User32.dll` oraz `MessageBoxA`. Po wykonaniu instrukcji z Listingu 7 w rejestrze `EAX` znajduje się potrzebna nam API. Aby ją wywołać wystarczy użyć później w kodzie instrukcji `call eax`, gdyż `GetProcAddress` właśnie do `EAX` zwraca wyszukaną funkcję.

Trzeba przyznać jednak, iż ręczne importowanie funkcji nie jest zbyt wygodne. Poza tym zdarza się (choć niezmiernie rzadko), iż w programie nie jest zaimportowana nawet funkcja `LoadLibraryA` (jedyne pewne funkcje, które znajdują się zawsze w każdym programie to `GetModuleHandleA` oraz `GetProcAddress`). Wtedy nawet powyższy kod na niewiele nam się zdaje. W takiej sytuacji mamy do wybory dwie drogi. Albo napiszemy kod, który załaduje nam `LoadLibraryA` albo skorzystamy z gotowego programiku dołączającego wybrane funkcje API. Jako że napisanie kodu, który ładowałby `LoadLibraryA` jest dość skomplikowane i wymaga bardzo dobrej wiedzy z zakresu asemblera skorzystamy z programu, który bezpośrednio może zaimportować potrzebną nam API. Możemy to zrobić dzięki programowi `IIDKing`. Jest on bardzo pomocny i o wiele wygodniejszy niż ręczne importowanie funkcji w przypadku ich braku w programie.

Podsumowując artykuł należałoby stwierdzić, iż modyfikacja plików binarnych nie należy do najłatwiejszych zadań. Wystarczy jednak odrobina samozaparcia oraz pewnej wiedzy z zakresu asemblera i plików PE, aby swobodnie poruszać się po kodzie programu oraz go modyfikować według naszych potrzeb. ■