Przechwytywanie wywołań funkcji w bibliotekach DLL. API hooking w praktyce.

Autor *Bartosz Wójcik*, ukazał się w magazynie *Programista 3/2013 (10)*

Metody API hookingu dla programistów poprzez wykorzystanie mechanizmu przechwytywania wywołań funkcji w bibliotekach DLL (DLL proxy).

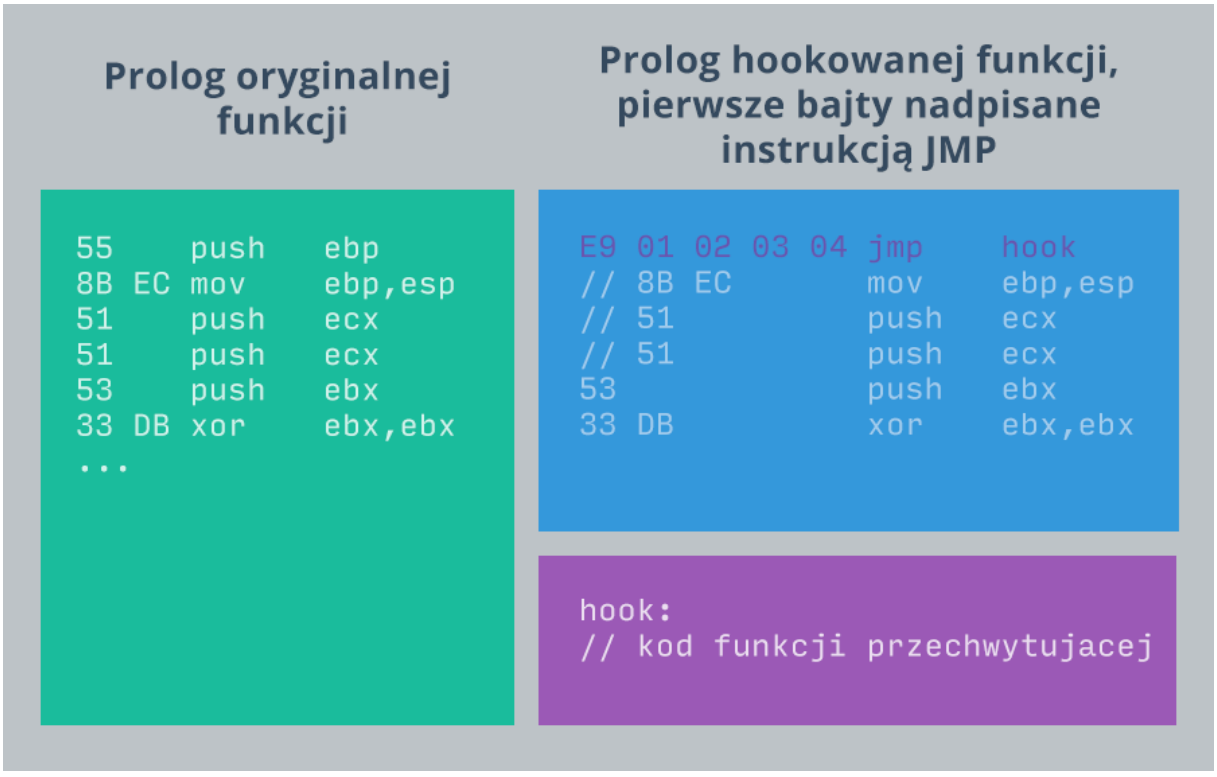
Czasami zdarza się taka sytuacja, że chcemy przechwycić wywołanie jakiejś funkcji w bibliotece dynamicznej DLL, być może odkryliśmy błąd w aplikacji albo chcemy dodać dodatkową funkcjonalność czy też logowanie wywoływanych funkcji i ich parametrów. W normalnych przypadkach mamy dostęp do kodów źródłowych i modyfikacja funkcji wymaga jedynie edycji odpowiedniego pliku źródłowego, lecz czasami nie mamy dostępu do kodów źródłowych biblioteki lub producent ich po prostu nie udostępnia. Co w takich sytuacjach robić? W

poniższym artykule przeczytacie o popularnych rozwiązaniach oraz zaprezentowane zostanie nieco inne podejście do tego tematu.

API Hooking

Najpopularniejszym rozwiązaniem, które zapewne większość z Was zna to tzw. *API Hooking*, czyli technika polegająca na tym, że wywołania funkcji bibliotek można przekierować do swojego kodu. Najbardziej znane biblioteki do tworzenia *hooków* to *Microsoft Detours* (wykorzystywana często przy tworzeniu *hacków* do gier), która jednak w wersji komercyjnej kosztuje bagatela 9,999.95 USD (czyli 31737 PLN!), dla Delphi znajdziemy bibliotekę *madCodeHook*, której koszt wynosi € 349 do zastosowań komercyjnych. Oprócz wymienionych bibliotek istnieje wiele innych i darmowych odpowiedników.

Zasada działania takich bibliotek polega na ingerencji w kod binarny załadowanych bibliotek i ich funkcji, w skrócie przedstawia się to tak, że prolog funkcji (sam początek funkcji w pamięci), której wywołanie chcemy przechwycić, nadpisywany jest przez kod przekierowujący wykonywanie do wskazanej przez nas funkcji (najczęściej jest to instrukcja assemblera `JMP NEAR`, zakodowana hexadecymalnie jako `E9 xx xx xx xx`). Obrazkowo można to przedstawić tak:



Rysunek 1. Funkcja przed i po ustawieniu hooka

Po przekazaniu kontroli do naszej funkcji, zwykle mamy możliwość uruchomić swój kod, wywołać oryginalną funkcję i wrócić do kodu, który wywołał funkcję z biblioteki DLL.

API Hooking może powodować kilka problemów, wiąże się to z budową skompilowanych aplikacji i struktury kodu, problem pojawia się w przypadku, gdy po przechwyceniu wywołania chcemy wywołać oryginalną funkcję (normalnie wpadlibyśmy w nieskończoną pętlę), w takich sytuacjach konieczne jest stworzenie specjalnej konstrukcji kodu, tzw. *trampoliny*, która pomimo założonego *hooka*, pozwala wywołać kod oryginalnej funkcji. Technika *API Hookingu* jest też praktycznie niemożliwa do wykorzystania w przypadku zabezpieczonych bibliotek DLL, gdzie ingerencja czy to w kod biblioteki na dysku czy też w pamięci nie jest możliwa np. gdy wykorzystywana jest jakakolwiek forma sprawdzania sum kontrolnych pliku, regionów pamięci etc.

Klasyczny *API Hooking* też nie nadaje się do przechwytywania *pseudofunkcji* eksportowanych przez dynamiczne biblioteki DLL, chodzi tutaj o eksportowanie wartości, wskaźników klas etc., ponieważ w takich sytuacjach nie ma możliwości stworzenia przekierowania w sensie stworzenia *hooka* pomiędzy oryginalną funkcją, a naszym kodem przechwytującym (bo żadnego kodu po prostu nie ma), pewnym rozwiązaniem jest tutaj modyfikacja tabeli eksportów pliku *PE* (Portable Executable), jednak nie wszystkie biblioteki *hookujące* oferują taką funkcjonalność.

DLL Forwarding

Jednym z bardziej *kreatywnych* i nieco trudniejszych sposobów na przechwycenie wywoływania dowolnych funkcji z bibliotek DLL jest wykorzystanie wewnętrznego mechanizmu systemu Windows o nazwie *DLL Forwarding*, czyli w skrócie *przekazywanie wywołań funkcji DLL*.

Technika ta polega na stworzeniu biblioteki zastępczej tzw. *proxy DLL*, która eksportuje wszystkie funkcje, oferowane przez oryginalną bibliotekę i przekazywanie wszystkich wywołań do oryginalnej biblioteki oprócz tych, które nas interesują. Przekazywanie wywołań funkcji do oryginalnej biblioteki wykorzystuje mało używany mechanizm programu ładującego Windows, który pozwala bibliotekom eksportować funkcje, które faktycznie znajdują się w innych bibliotekach (stąd nazwa *forwarding* – przekazywanie, przekierowanie).

Konwencje wywołań funkcji

Konwencja wywołań funkcji to niskopoziomowy sposób przekazywania parametrów do funkcji i schemat obsługi stosu, w głównej mierze zależy od ustawień kompilatora i w większości języki wysokiego poziomu pozwalają na swobodną zmianę konwencji, czy to na poziomie opcji kompilatora czy też bezpośrednio w kodzie źródłowym poprzez wykorzystanie specjalnych konstrukcji języka. Aby nasza biblioteka pośrednicząca działała sprawnie, funkcje, które przechwytujemy muszą być zapisane w dokładnie takiej samej konwencji jak oryginalne funkcje, muszą być po prostu binarnie kompatybilne, inaczej może zakończyć się to wyjątkiem wskutek np. uszkodzenia wskaźnika stosu etc.

Tabela1. Konwencje wywołań funkcji.

Nazwa	W kodzie C	Parametry	Zwracane wartości	Modyfikowane rejestry	Info
cdecl	cdecl	zapisywane na stosie, stos nie jest korygowany przez funkcję	eax, 8 bajtów: eax:edx	eax, ecx, edx, st(0), st(7), mm0, mm7, xmm0, xmm7	Sposób wywoływania funkcji z bibliotek C, wprowadzony przez Microsoft, wszystkie funkcje systemowe na platformie Linux również używają tego standardu
fastcall	__fastcall	ecx,edx, reszta parametrów przekazywana przez stos	eax, 8 bajtów: eax:edx	eax, ecx, edx, st(0), st(7), mm0, mm7, xmm0, xmm7	Microsoft wprowadził ten standard, ale potem w swoich produktach zmienił go na standard cdecl
watcom	__declspec (wcall)	eax, ebx, ecx, edx	eax, 8 bajtów: eax:edx	eax	Standard wywoływania funkcji wprowadzony przez firmę Watcom w ich kompilatorze C++

stdcall	__stdcall	zapisywane na stosie, stos korygowany przez samą funkcję	eax, 8 bajtów: eax:edx	eax, ecx, edx, st(0) st(7), mm0, mm7, xmm0, xmm7	Domyślny typ wywoływania funkcji API w Windows, w bibliotekach DLL
register	n/a	eax, edx, ecx, reszta na stosie	eax	eax, ecx, edx, st(0), st(7), mm0, mm7, xmm0, xmm7	Metoda wywoływania funkcji w Delphi firmy Borland

Mimo konfigurowalności ustawień, środowiska programowania korzystają z domyślnych ustawień i tak np. dla *Delphi* standardem jest typ *register*, dla większości programów napisanych w C standardem jest *cdecl*.

Funkcje *WinApi* (systemowe Windows) korzystają z mechanizmu *stdcall*, czyli parametry funkcji najpierw zapamiętywane są na stosie, po czym następuje wywołanie funkcji, po wywołaniu funkcji, nie trzeba korygować stosu (*zdejmować* wcześniej zapamiętanych parametrów ze stosu), ponieważ funkcja zrobi to za nas, automatycznie korygując wskaźnik stosu, ciekawostką jest fakt, że kilka funkcji *WinApi* nie korzysta ze sposobu wywoływania *stdcall*, ale z *cdecl*, czyli parametry zapamiętywane są na stosie, po czym wywoływana jest funkcja, ale samo korygowanie stosu musi być wykonane ręcznie. Przykładem takiej funkcji jest `wsprintfA()` z biblioteki systemowej Windows *USER32.dll* (jej odpowiednikiem w bibliotekach C jest `sprintf()`), ten sposób został najprawdopodobniej wprowadzony, ponieważ ww. funkcje nie posiadają stałej ilości parametrów.

Przykład API hookingu

Za nasz przykład posłuży nasza testowa biblioteka *BlackBox.dll***,** która eksportuje tylko dwie, umowne funkcje `Sumuj()` i `Podziel()`, odpowiednio wykonujące dodawanie i dzielenie dwóch liczb. Zakładamy, że posiadamy dokumentację

biblioteki i wiemy jaka jest konwencja wywołań tych funkcji (czyli posiadamy pliki nagłówkowe biblioteki) oraz jakie parametry przyjmują, w innych wypadkach wymagana byłaby już analiza wsteczna kodu (tzw. *reverse engineering*).

Listing 1. Opis funkcji z biblioteki BlackBox.dll

```
// funkcja dodaje dwie liczby, zwraca wartość do
// zmiennej „Wynik”, zwraca TRUE na sukces,
// FALSE w przypadku błędu
BOOL __stdcall Sumuj(int Liczba1, int Liczba2, int * Wynik);


// funkcja dzieli dwie liczby, zwraca wartość do
// zmiennej „Wynik”, zwraca TRUE na sukces,
// FALSE w przypadku błędu
BOOL __stdcall Podziel(int Liczba1, int Liczba2, int * Wynik);
```

W naszym przykładzie funkcja `Podziel()` jest jednak uszkodzona i dzielenie przez zero kończy się zawieszeniem aplikacji (która na nasze nieszczęście nie obsługuje wyłapywania wyjątków), naszym celem będzie naprawienie tej funkcji.

Proxy DLL

Aby naprawić uszkodzoną funkcję w bibliotece *BlackBox.dll*, utworzymy bibliotekę pośredniczącą, która zaimplementuje poprawną funkcję `Podziel()` z obsługą dzielenia przez zero. Implementacja zostanie wykonana w 32 bitowym assemblerze w składni *FASM* (polski kompilator assemblera autorstwa Tomasza Grysztara). Poniżej znajdziecie szkielet takiej przykładowej biblioteki z dokładnymi opisami kolejnych struktur kodu.

Listing 2. Początek naszej biblioteki.

```
;-----------------------------------------------
; deklaracja wyjściowego formatu pliku DLL
;-----------------------------------------------
format PE GUI 4.0 DLL


; nazwa funkcji wejściowej naszej biblioteki
entry DllEntryPoint
```

```
; plik nagłówkowy z definicjami
; i stałymi dla Windows
include '%fasm%\include\win32a.inc'
```

Tutaj zawarte są deklaracje o typie generowanego pliku, w tym miejscu załącza się pliki nagłówkowe oraz deklaruje nazwę funkcji wejściowej, czyli punktu, od którego rozpoczyna się wykonywanie aplikacji lub ładowanie biblioteki DLL.

Listing 3. Sekcja niezainicjalizowanych wartości

```
;-------------------------------------------------
; sekcja z niezainicjalizowanymi wartościami
;-------------------------------------------------
section '.bss' readable writeable

; uchwyt HMODULE oryginalnej biblioteki
      hLibOrg            dd ?
```

Pliki wykonywalne w tym biblioteki DLL dzielą się na sekcje, jedną z nich jest sekcja z niezainicjalizowanymi danymi, która nie zajmuje miejsca na dysku, a jedynie zawiera definicję zmiennych, które w kolejnej fazie programu mogą być zapisane. Nazwy sekcji w plikach wykonywalnych nie mają znaczenia (posiadają jedynie limit 8 znaków), zwykle są to tylko umowne oznaczenia, a przy ich deklaracji konieczne jest określenie praw dostępu (czytanie, zapisywanie, wykonywanie), jednak w przypadku kompilatora *FASM* deklaracja sekcji o nazwie *.bss* powoduje utworzenie sekcji niezainicjalizowanych wartości.

Listing 4. Sekcja z zainicjalizowanymi wartościami.

```
;-------------------------------------------------
; sekcja z zainicjalizowanymi danymi
;-------------------------------------------------
section '.data' data readable writeable

; nazwa oryginalnej biblioteki
      szDllOrg     db 'BlackBox_org.dll',0
```

Tutaj mamy zapisaną nazwę pliku oryginalnej biblioteki, którą przemianowaliśmy na *BlackBox_org.dll* (zapisana jest w kodzie źródłowym jako ciąg znaków *ASCIIz*,

czyli zakończonych zerem), wykorzystana będzie ona w późniejszym kodzie do załadowania tejże biblioteki.

Listing 5. Sekcja z kodem i punktem wejściowym naszej biblioteki.

```
;-------------------------------------------------
; sekcja z kodem naszej biblioteki
;-------------------------------------------------
section '.text' code readable executable

;-------------------------------------------------
; punkt wejściowy biblioteki dynamicznej (DllMain)
;-------------------------------------------------
proc DllEntryPoint hinstDLL, fdwReason, lpvReserved

        mov     eax,[fdwReason]

; komunikat wysyłany po załadowaniu biblioteki DLL
        cmp     eax,DLL_PROCESS_ATTACH
        je      _dll_attach

        jmp     _dll_exit

; biblioteka została właśnie załadowana
_dll_attach:

; pobierz uchwyt oryginalnej biblioteki, może się
; on przydać gdybyśmy chcieli wywołać oryginalne
; funkcje
        push    szDllOrg
        call    [GetModuleHandleA]
        mov     [hLibOrg],eax

; zwróć 1, co oznacza, że powiodła się
; inicjalizacja naszej biblioteki
        mov     eax,1
```

```
_dll_exit:

        ret
```

Sekcja kodu zawiera wszystkie funkcje biblioteki oraz punkt wejściowy (z ang. *entrypoint*), czyli specjalną funkcję, która zostaje wywołana po załadowaniu biblioteki. Sekcja kodu musi być również oznaczona jako *executable*, czyli wykonywalna, w przeciwnym wypadku uruchomienie jej kodu na procesorach z obsługą zapobiegania uruchamiania danych *DEP* (Data Execution Prevention) spowoduje natychmiastowy wyjątek. W kodzie inicjalizującym biblioteki (po otrzymaniu komunikatu *DLL_PROCESS_ATTACH*) korzystamy z nazwy oryginalnej biblioteki i pobieramy jej uchwyt (żeby móc wykorzystać ją np. do wywołania oryginalnych funkcji).

Listing 6. Ochrona przed optymalizacją

```
; wywołaj jakąkolwiek funkcję z oryginalnej
; biblioteki BlackBox_org.dll, bez tego FASM
; usuwa referencje do tej biblioteki i nie
; zostanie ona automatycznie załadowana
        call   dummy
```

Nasza biblioteka wykorzystuje oryginalną bibliotekę, jeśli jednak nie wywołamy z niej żadnej funkcji, kompilator *FASM* usunie do niej referencje (optymalizacja) i nie będzie ona automatycznie załadowana, dlatego tutaj, po instrukcji `ret` - wstawiamy fałszywe wywołanie dowolnej jej funkcji (zadeklarowanej w dalszej części naszej biblioteki).

Listing 7. Poprawna implementacja funkcji Podziel()

```
;-------------------------------------------------
; nasza implementacja funkcji Podziel z poprawionym
; kodem, odpornym na dzielenie przez zero
;-------------------------------------------------
proc Podziel Liczba1, Liczba2, Wynik


; dzielnik, sprawdź czy jest to zero, jeśli tak to
; wyjdź z kodem błędu z funkcji
        mov    ecx,[Liczba2]
```

```
        test   ecx,ecx
        je     PodzielBlad

; załaduj pierwszą liczbę (rejestr EDX rozszerz
; o znak wartości int Liczba1)
        mov    eax,[Liczba1]
        cdq

; wykonaj dzielenie EDX:EAX / ECX, dzielenie
; wykonuje się na parze rejestrów EDX:EAX, które
; traktowane są jako liczba 64 bitowa, wynik
; dzielenia znajdzie się w rejestrze EAX, reszta
; z dzielenia zostanie zapisana w rejestrze EDX
        idiv   ecx

; czy podany jest wskaźnik, gdzie ma być zapisany
; wynik, jeśli brak wskaźnika, wyjdź z kodem błędu
        mov    edx,[Wynik]
        test   edx,edx
        je     PodzielBlad

; zapisz wynik dzielenia pod wskazany adres
        mov    [edx],eax

; wyjdź z kodem TRUE (1)
        mov    eax,1

        jmp    PodzielWyjdz

; błąd dzielenia, zwróć wartość FALSE (0)
PodzielBlad:

        sub    eax,eax

PodzielWyjdz:
```

```
; wyjdź z funkcji dzielenia z ustawionym
; kodem błędu typu BOOL w rejestrze EAX
	ret
endp
```

W naszej implementacji sprawdzane jest dzielenie przez zero i jeśli dzielnik jest zerem, funkcja zwraca kod błędu FALSE, dodatkowo sprawdzany jest wskaźnik do liczby, gdzie ma być zapisany wynik, jeśli wskaźnik jest pusty (NULL), również zostanie zwrócony kod błędu. Należy zwrócić uwagę, żeby funkcja była w takiej samej konwencji wywoływania jak oryginalna funkcja, w naszym przypadku wykorzystana jest konwencja *stdcall*, czyli parametry przekazywane są przez stos, wartość funkcji zwracana w rejestrze EAX i wskaźnik stosu automatycznie jest korygowany po wyjściu z funkcji, kompilator *FASM* robi to automatycznie, generując w pliku wynikowym instrukcję `ret (liczba_parametrów * 4)`, pomimo, że w źródłach jest tylko `ret`.

Listing 8.Tablica importów naszej biblioteki.

```
;-------------------------------------------------
; sekcja z funkcjami wykorzystywanymi przez naszą
; bibliotekę
;-------------------------------------------------
section '.idata' import data readable writeable

; lista bibliotek, których funkcji używamy
library kernel,'KERNEL32.DLL',\
	blackbox, 'BlackBox_org.dll'

; lista funkcji z biblioteki KERNEL32.dll
import kernel,\
	GetModuleHandleA, 'GetModuleHandleA'

; deklarujemy użycie oryginalnej biblioteki, co
; sprawi, że zostanie ona automatycznie
; załadowana
import blackbox,\
```

```
        dummy, 'Podziel'
```

Kompilator *FASM* pozwala nam ręcznie zdefiniować biblioteki i funkcje, z których korzysta nasza biblioteka, oprócz standardowych bibliotek systemowych, umieszczamy tutaj odniesienie do oryginalnej biblioteki *BlackBox_org.dll*, co spowoduje, że system Windows, ładując naszą bibliotekę pośredniczącą, automatycznie załaduje również oryginalną bibliotekę w przestrzeń adresową aplikacji, zaoszczędzi to nam pracy z ręcznym ładowaniem biblioteki przez funkcję `LoadLibraryA()`, a czasami wręcz jest konieczne, dotyczy to bibliotek dynamicznych, które mogą być ładowane jedynie statycznie, przez wpis w tabelę importów aplikacji, najczęściej wykorzystują one mechanizm *TLS* (Thread Local Storage) dla wielowątkowych aplikacji.

Listing 9.Tablica eksportowanych funkcji

```
;-------------------------------------------------
; sekcja z funkcjami eksportowanymi przez naszą
; bibliotekę, musimy tu zadeklarować wszystkie
; funkcje, które znajdują się w oryginalnej
; bibliotece
;-------------------------------------------------
section '.edata' export data readable


; lista eksportowanych funkcji i ich wskaźniki
export 'BlackBox.dll',\
        Sumuj, 'Sumuj',\
        Podziel, 'Podziel'


; nazwa przekazywanej funkcji, najpierw deklaruje
; się nazwę biblioteki, do której ma nastąpić
; przekierowanie, a po kropce deklaruje się nazwę
; docelowej funkcji
        Sumuj db 'BlackBox_org.Sumuj',0
```

W tej sekcji musimy zadeklarować wszystkie funkcje z oryginalnej biblioteki, funkcje które obsługujemy muszą mieć implementacje w kodzie, funkcje, których wywołania chcemy przekierować do oryginalnej biblioteki zapisujemy w specjalnej notacji:

`NazwaDocelowejBiblioteki.NazwaFunkcji`

lub

`NazwaDocelowejBiblioteki.#1`

dla funkcji eksportowanych jedynie przez numer, a nie nazwę. Za całą funkcjonalność odpowiada już wewnętrzny mechanizm systemu Windows, czyli *DLL Forwarding*.

Listing 9.Tablica relokacji

```
;-----------------------------------------------

; sekcja relokacji

;-----------------------------------------------

section '.reloc' fixups data discardable
```

Ostatnią sekcją w naszej bibliotece jest sekcja relokacji, jest ona konieczna do poprawnej funkcjonalności, ponieważ biblioteki dynamiczne DLL mogą być ładowane pod różne adresy bazowe w przestrzeni adresowej procesu i adresy bezwzględne w kodzie biblioteki (np. wskazujące na zmienne globalne) muszą być odpowiednio skorygowane, informacje o tych korektach zapisane są właśnie w sekcji relokacji.

Podsumowanie

Przedstawiona technika może z powodzeniem być wykorzystywana do modyfikacji dowolnych aplikacji, których funkcjonalność zawarta jest w bibliotekach dynamicznych DLL. Posiada ona swoje zalety i wady (w stosunku do np. *API Hookingu*), jednak moim zdaniem oferuje znacznie szersze pole do popisu i w łatwiejszy sposób pozwala zmienić całkowitą funkcjonalność aplikacji. Implementacja tej metody może być również wykonana w językach wysokiego poziomu z odpowiednim wykorzystaniem plików definicji eksportowanych funkcji (DEF).

Źródła

- Microsoft Detours - http://research.microsoft.com/en-us/projects/detours/
- madCodeHook - http://madshi.net/madCodeHookShop.htm
- Różne techniki API Hookingu - http://jbremer.org/x86-api-hooking-demystified/
- Kompilator assemblera FASM - http://flatassembler.net/
- Konwencje wywoływania funkcji - http://msdn.microsoft.com/en-us/library/k2b2ssfy.aspx

- Dynamiczne ładowanie bibliotek i mechanizm TLS Storage - http://support.microsoft.com/kb/118816/en-us
- Data Execution Prevention (DEP) - http://en.wikipedia.org/wiki/Data_Execution_Prevention
- DLL Forwarding - http://msdn.microsoft.com/en-us/library/hyx1zcd3(v=vs.80).aspx

O Autorze

Bartosz Wójcik - autor zajmuje się systemami ochrony oprogramowania przed złamaniem (http://www.pelock.com), zaawansowaną analizą wsteczną kodu (*reverse engineering*), tematy te często porusza na swoim blogu (http://www.secnews.pl), prowadzi również stronę z ogłoszeniami o pracy dla ludzi zajmujących się bezpieczeństwem komputerowym (*hacking*, *pentesting*, *reversing*, *kernel development*) – http://www.compusecjobs.com