УДК 004.492.3

Протасов М.В., Мазурець О.В.

Хмельницький національний університет, Україна

МЕТОДИ ПЕРЕХОПЛЕННЯ API В USERMODE НА БАЗІ АРХІТЕКТУРИ X86 ОПЕРАЦІЙНОЇ СИСТЕМИ WINDOWS

В статті виконано аналіз існуючих методів перехоплення системних та прикладних API-функцій та запропоновано новий ефективний метод перехоплення API-функцій.

In this article were analyzed different common existing methods of hooking kernel and application API, and proposed new alternative hook method.

При розробці програмного забезпечення (ПЗ) системного призначення, такого як антивіруси, системи контролю та інші захисні системи, часто виникає потреба в перехопленні системних чи прикладних API-функцій. Вони зазвичай розташовуються в бібліотеках динамічної компоновки (dynamic link library, dll).

Процес передачі керування на бібліотечну функцію при її виклику відбувається у декілька етапів. Для того, щоб функції бібліотеки стали доступними для виклику, бібліотека має бути спроектована в адресний простір процесу.

Існуючі методи перехоплення мають ряд характерних недоліків [1, 2], й *метою статті* встановлено аналіз існуючих методів перехоплення системних та прикладних API-функцій, й визначення за результатами аналізу шляхів підвищення ефективності перехоплення API-функцій.

За ***класичним методом*** процес передачі керування на бібліотечну функцію відбувається при наступних умовах [1]:

- Бібліотека вказана в таблиці імпорту в PE-хедері виконуваного модуля.
- Бібліотека завантажена в АП процесу за допомогою LdrLoadDll або її враппера LoadLibrary.

Exe-модуль являє собою сукупність коду, даних, та хедерів. PE-хедер [1] має структуру, як показана на рисунку 1.

Формат PE-файлу
- Старий MZ-заголовок
- Dos stub-програма (заглушка)
- Файловий PE-заголовок
- Опціональний PE-заголовок
  - Стандартні для COFF поля
  - Windows-специфічні поля
  - Каталоги даних
    - Таблиця експорту
    - Таблиця імпорту
    - Таблиця ресурсів
    - Таблиця обробників виключень
    - Таблиця атрибутів сертифікату
    - Таблиця пересилань (reloc`ів)
    - Таблиця налагоджувальних символів
    - Архітектура (зарезервовано, 0)
    - Глобальний вказівник
    - Таблиця потокових локальних змінних
    - Завантаження структур конфігурації
    - Зв’язний імпорт
    - Таблиця імпортованих адрес
    - Дескриптор відкладеного імпорту
    - CLR-хедер

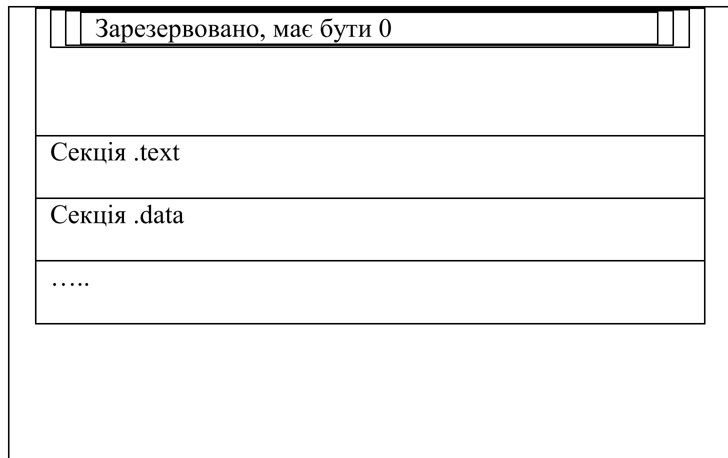

Рис. 1. **Структура PE-заголовку**

У кінці опціонального заголовку міститься каталог структур IMAGE_DATA_DIRECTORY, які виглядають наступним чином [1]:

```
struct IMAGE_DATA_DIRECTORY
   VirtualAddress dd        ?
   Size                     dd        ?
ends
```

Тобто являє собою пари адрес – розмір. З усіх елементів IMAGE_DATA_DIRECTORY найбільш важливою є таблиця імпорту.

Таблиця імпорту складається з наступних полів[1]:

- Таблиця пошуку (ILT).
- Відмітка дати/часу.
- Експедитор ланцюга.
- Адреса строки, яка містить ім’я dll - бібліотеки.
- Таблиця імпортованих адресів (IAT).

Таблиця імпорту має стільки входжень, скільки модулів мають бути імпортовані. Останній запис в таблиці імпорту має містити 0.

З цих полів найбільший інтерес представляють 1, 4 і 5 (таблиця пошуку, ім’я бібліотеки та таблиця імпортованих адресів). Таблиця пошуку представляє бітове поле, яке має наступну структуру [1]:

- Біт 31 – прапор, який задає, як буде відбуватись імпорт. Якщо цей прапор встановлений – імпорт відбувається за ординалом, інакше – за ім’ям.
- Біти 0-15. 16-бітний ординальний номер. Використовується тільки тоді, коли встановлений прапор ординалу.
- Біти 0-30. Якщо прапор ординалу не встановлений – ці біти містять RVA(відносний віртуальний адрес) таблиці підказка/ім’я (hint/name).

Таблиця hint/name складається з [1]:

- 2-байтового поля hint, у якому міститься номер в таблиці вказівників імен з таблиці експорту dll.
- asciiz-стрічка, яка містить ім’я імпортованої функції.
- Поле, яке застосовується для вирівнювання наступного запису на 4-байтну границю.

Таблиця імпортованих адрес (IAT) має таку ж структуру, і до загрузки модуля вони мають бути ідентичні. Останній запис у цих таблицях має містити нулі.

Механізм імпорту діє наступним чином після закінчення роботи, StartUp-APC і ініціалізації Cookies, якщо потік перший, то виконується ініціалізація оточення (кс, хіп, бд, загрузка kernel32.dll, імпорт, etc.). Ініціалізація імпорту відбувається так: виконується парсинг таблиці імпорту, dll, присутні в ній, завантажуються в адресний простір процесу. Таблиця пошуку і таблиця імпортованих адрес мають бути ідентичні. Далі виконується перевірка – як має відбуватись імпорт – по іменам, чи ординалам. Ординал [1]: - номер вказівника в таблиці експортованих адрес в таблиці експорту dll. Цей спосіб рідко використовується, так як в різних версіях dll одні і ті ж функції можуть мати різні ординали. Якщо 31 біт в ILT скинутий – виконується пошук за ім’ям. Спочатку перевіряється 2-байтова підказка, якщо там 0 – виконується парсинг таблиці експорту dll. Після цього IAT заповнюється RVA знайдених ф-цій.

Таким чином, у момент виклику будь-якої API її адреса береться з IAT, у якій до цього моменту вже завантажені адреси функцій. Виходячи з цього, здійснити перехоплення можна наступним чином [1]:

1. Розпарсити таблицю імпорту, і знайти потрібну IAT, у якій знаходиться адреса оригінальної функції.
2. Підмінити адресу на свій обробник функції.
3. Обробити виклик потрібним чином, потім передавши керування на оригінальну функцію (її адресу можна отримати, зберігши адресу перед заміною на свій обробник, або викликавши LdrGetProcedureAddress (або обгортка GetProcAddress).

У цьому методі існують наступні недоліки [3]:

- Модуль може імпортуватись через LoadLibrary/GetProcAddress, і в таблиці імпорту відповідного запису не буде.
- Такий перехват легко задетектувати, зрівнявши адреси в таблицях модуля IAT та таблицях EAT у dll відповідно.

Першу проблему можна обійти, перехопивши GetProcAddress і підмінивши повернуте нею значення на свій обробник. Але це потрібно зробити до моменту динамічного імпортування.

Сплайсинг – один з методів перехоплення функцій шляхом заміни цільової функції з метою процедурної маршрутизації на потрібний обробник [2]. Для того, щоб більш докладно описати цей метод, розглянемо, що з себе представляє процедура на архітектурі x86 з stdcall-конвецією виклику.

Процедура – певна область коду, яка викликається спеціальним чином. На x86 процедура визивається командою call. При цьому у стеку зберігається адреса повернення (наступна за розгалуженням інструкція) і керування передається на процедуру (для виклику в поточному кодовому сегменті, при міжсегментному виклику крім адреси повернення в стеку зберігається селектор кодового сегменту). В стеку можуть зберігатись локальні змінні. Вони адресуються через ebp. Конвеція stdcall потребує збереження ebx, esi, edi та ebp. Тому на початку кожної процедури в стеку зберігається значення ebp, і в нього завантажується вказівник вершини стеку esp і в стеку виділяється місце під локальні змінні. Таким чином, ebp адресує значення попередньої процедури в стеку, вище знаходиться адреса повернення з процедури, а нижче – локальні змінні процедури та її аргументи. Процедура виглядає наступним чином:

```
push	ebp		;збереження попереднього значення ebp
mov	ebp, esp	;завантаження посилання на стековий фрейм в ebp
sub	esp, 40H	;виділення місця під локальні змінні

…		;код процедури

mov	esp, ebp	;повернення стеку
pop	ebp		;відновлення регістру ebp
retn	4		;видалення зі стека 4 параметра процедури
```

Конвеція stdcall передбачає передачу параметрів справа наліво через стек, стек очищає викликана процедура. Для перехоплення функцій, оформлених таким чином, існує декілька технік [2].

Hot patching – один з різновидів сплайсингу, який офіційно підтримується Microsoft [4]. Суть даного методу полягає у наступному: функції, які підтримують hot patching, містять 5-байтовий пролог, та місце перед початком функції, що дозволяє передати керування на свій обробник, не змінюючи фактично коду оригінальної функції:

```
nop
nop
nop
nop
nop
mov	edi, edi
push	ebp
mov	ebp, esp
…		; код функції
```

Для прикладу, початок функції CopyFileA() з бібліотеки kernel32.dll. У дизассемблері IDA вона буде мати вигляд, який показано на рисунку 2.

```
.text:7DD958C8                 db 5 dup(90h)
.text:7DD958CD ; Exported entry 115. CopyFileA
.text:7DD958CD
.text:7DD958CD ; =============== S U B R O U T I N E =======================================
.text:7DD958CD
.text:7DD958CD ; Attributes: bp-based frame
.text:7DD958CD
.text:7DD958CD ; BOOL __stdcall CopyFileA(LPCSTR lpExistingFileName, LPCSTR lpNewFileName, BOOL bFailIfExists)
.text:7DD958CD                 public _CopyFileA@12
.text:7DD958CD _CopyFileA@12   proc near               ; DATA XREF: .text:off_7DE1FA68↓o
.text:7DD958CD
.text:7DD958CD var_10          = byte ptr -10h
.text:7DD958CD var_C           = dword ptr -0Ch
.text:7DD958CD var_8           = byte ptr -8
.text:7DD958CD var_4           = dword ptr -4
.text:7DD958CD lpExistingFileName= dword ptr  8
.text:7DD958CD lpNewFileName   = dword ptr  0Ch
.text:7DD958CD bFailIfExists   = dword ptr  10h
.text:7DD958CD
.text:7DD958CD ; FUNCTION CHUNK AT .text:7DDA2F7E SIZE 00000007 BYTES
.text:7DD958CD
.text:7DD958CD                 mov     edi, edi
.text:7DD958CF                 push    ebp
.text:7DD958D0                 mov     ebp, esp
.text:7DD958D2                 sub     esp, 10h
.text:7DD958D5                 push    [ebp+lpExistingFileName]
.text:7DD958D8                 lea     eax, [ebp+var_8]
```

Рис. 2. **Функція CopyFileA у дизассемблері IDA**

З рисунку 2 видно, що перед початком функції є 5 байт 90h, які на мові асемблера мають мнемоніку nop (No Operation). Інструкція mov edi, edi є фактично 2-байтовим nop-ом, і має код 89 FF або 8B FF (в залежності від того, у яку форму скомпілював її компілятор, у MOV r/m32,r32 чи у MOV r32,r/m32 відповідно). Цього вистачить, щоб її замінити на short jmp, який має форму JMP rel8 (-128/ +127 відносно поточного значення EIP). У 5 вільних байтах, які знаходяться вище функції, можна розмістити ближній перехід (JMP rel32), який займає 5 байт, і здійснює передачу керування у межах поточного кодового сегменту. Код інструкції - E9 cd, де cd – адреса потрібного обробника. Код перехопленої функції буде мати вигляд, показаний на рисунку 3.

Як видно на рисунку 3, на початку прологу функції записане коротке безумовне розгалуження на обробник. Виконати перезапис обробника можна наступним кодом:

```
mov     eax, dword [CopyFileA]  ;адреса оригінального обробника, з IAT
mov     word [eax], 0F9EBH         ;jmp    short $ - 7
lea     ebx, [HookApiProc]
sub     ebx, eax             ;вирахування RVA  обробника
mov     byte [eax - 5], 0E9H         ;опкод jmp
 mov    dword [eax - 4], ebx         ;адреса обробника
```

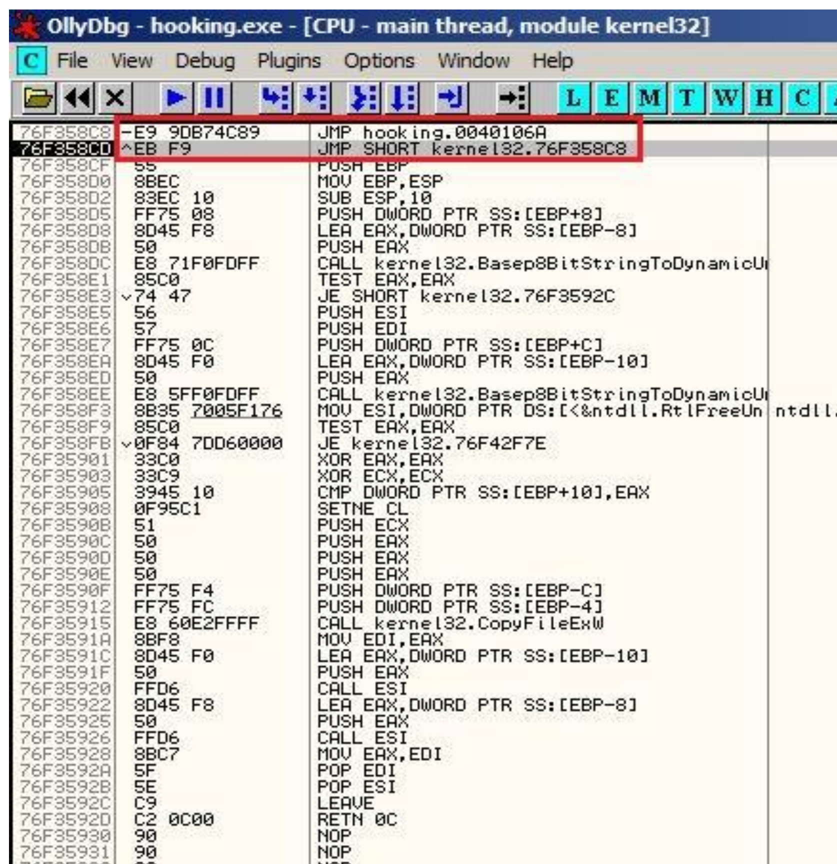

Рис. 3. **Перехоплена Win API CopyFileA у адресному просторі модуля**

Так як CopyFileA приймає 3 параметри, на момент виклику стек буде виглядати, як показано на рисунку 4.

ESP	Адреса повернення до коду, який викликав процедуру
ESP + 4	1 параметр
ESP + 8	2 параметр
ESP + 0CH	3 параметр

Рис. 4. **Стан стеку викликаної процедури**

Якщо потрібно в якомусь випадку заблокувати виконання оригінальної функції, то необхідно знати, скільки їй було передано параметрів, та очистити їх перед передачею керування коду, який викликав процедуру.

У цього методу перехоплення є наступні недоліки [3]:

- Не всі функції оформлені таким чином, деякі не мають прологу, або пролог починається з ініціалізації стекового фрейму (push ebp/mov ebp, esp), в такому разі просто замінити код на розгалуження не вийде, потрібно частину коду зберігати в своєму обробнику (В такому разі це вже не hot patching, а сплайсинг).
- У процесі може бути не один потік, а декілька, і якщо другий потік потрапить у перехоплену функцію до того, як вона буде повністю перехоплена, в результаті чого можуть бути неприємні наслідки.

Щоб цього уникнути, потрібно [3]

1. Отримати зліпок потоків в процесі.
2. Призупинити всі потоки, крім головного, доки повністю не завершиться копіювання коду перехоплення у функцію.
3. Відновити роботу потоків.

При такій реалізації є також недоліки [3]: функція буде перехоплена лише до тих пір, доки існує головний потік, так як код обробника розташовується в ньому. Можливий метод вирішення – обернути реалізацію перехоплення в dll, і зробити наступні кроки:

- Якщо планується перехоплювати функції в системних процесах, потрібно встановити SeDebugPrivilage для потоку, який буде вводити dll.
- Відкрити процес (OpenProcess()) з правами на запис.
- Виділити в адресному просторі процесу пам’ять для dll, яку потрібно ввести.
- Записати код dll (WriteProcessMemory()).
- Запустити код як потік (CreateRemoteThread()).

Перехоплення з побудовою CFG коду програми та стековою маршрутизацією. Як було вище зазначено про сплайсинг, для кожної процедури, оформленої згідно конвенції stdcall, вірні наступні твердження [5]:

- Параметри передаються через стек, очищує його викликана процедура.
- У стеку завжди є 2 чотирьохбайтних слова (dword), які розташовані поряд, попереднє значення ebp і адреса повернення у код, який викликав поточну функцію. Ця область стека називається *базовим стековим фреймом.* Його адресує регістр ebp, і можна описати наступною структурою:

```
struct STACK_FRAME
	LastFrame		dd	?
	ReturnAddress	dd	?
Ends
```

Очевидно, що для процедури, яка буде викликана з даної, буде посилання на поточний стековий фрейм, адреса якого збережена у цій структурі. Таким чином, утворюється *ланцюг стекових фреймів* (stack frame chain, SFC). Для визначення адреси повернення з попередньої процедури потрібно взяти в поточному стековому фреймі посилання на попередній стековий фрейм, і звідти взяти адресу повернення:

```
ReturnAddress = [ebx + STACK_FRAME.LastFrame].ReturnAddress
```

Такий перехід по ланцюгу стекових фреймів називається *бектрейсом* [2] (backtrace). Очевидно, що кожен стековий фрейм має свій номер. Цей номер називається *номером стекового фрейму* (stack frame number, SFN). Процедури, від яких залежить код SFN, виконуються на певному рівні по відношенню між собою. Це рівень залежить від ступеня вкладеності процедур, який називається *рівнем вкладеності* [2] ((nested level, NL):

$$NL = \Delta(SFN(B), SFN(A)),$$

де NL – рівень вкладеності,

A,B – процедури,

SFN() – номер стекового фрейму процедури.

Звідси випливають наступні залежності:

- Одне процедурне розгалуження збільшує NL на 1.
- Вихід у попередню процедуру зменшує NL на 1.
- NL не залежить від змісту стеку, а визначається лише розташуванням процедур в коді.
- SFN залежить від NL, так як при зміні SFN змінюється і NL (при зростанні SFN зпадає NL поточної процедури відносно цільової).
- Локальні змінні поточної процедури розміщені завжди нижче базового стекового фрейму.
- Зміщення N-аргументу процедури вичислюється за наступною формулою:

$$N = sizeof(STACK_FRAME) + 4 * N,$$

де N – номер аргументу.

З цього всього випливає, що знаючи NL та адресу поточного базового стекового фрейму, виконавши бектрейс, можна отримати локальні змінні, аргументи та локальні змінні цільової процедури.

Спосіб перехоплення, при якому виконується підміна адреси повернення на власний обробник у базовому стековому фреймі цільової процедури, SFN якої відомий, називається *стековою маршрутизацією*.

Для виконання стекової маршрутизації необхідні такі компоненти, як конструктор CFG та LDE у перехоплювачі.

CFG (control flow graph) – граф потоку керування, який являє собою певний набір шляхів виконання програми, представлених у вигляді графу [6]. Кожна вершина графу відповідає *базовому блоку* – лінійній області коду, яка не містить розгалужень, і на неї немає розгалужень. Існує 2 виключення:

1. Розгалуження може відбуватись на інструкцію, яка є першою інструкцією базового блоку.
2. Базовий блок завершується інструкцією передачі керування.

Будувати CFG необхідно у тому разі, якщо невідомі NL поточної та SFN цільової процедури.

LDE (length disassembler engine) – дизассемблер довжин, який видає довжину інструкції на подану адресу в АП модуля [7]. Потрібен для визначення початку процедури по адресі повернення на код, якій розміщується після виклику цільової процедури.

Наприклад, відома адреса процедури X() та Y(), X викликає процедуру Y, але не на пряму, а через невідому кількість процедур, тобто виклик відбувається наступним чином:

X() -> A() -> B() -> … -> Y()

Якщо треба визначити, чи виклик процедури Y() відбувся з процедури X(), за умови що адреса та кількість проміжних процедур невідома, то стандартними методами така задача не може бути вирішена (при умові що ресурси обмежені та пошук має відбуватись швидко). Один з варіантів – «сигнатурний» пошук, тобто шукати процедуру у пам’яті за певною послідовністю байт. У цього метода є суттєвий недолік – при зміні реалізації процедури сигнатуру знадобиться міняти, тобто реалізація буде залежати від версії.

Варіант вирішення задачі з використанням CFG, LDE та бектрейсу:

- Описати CFG X() з розкриттям усіх посилань.
- Визначити NL(Y) відносно X.
- Виконати NL ітерацій бектрейсу.
- Перевірити, чи адреса повернення належить A().

Приклад процесу бектрейсу наведено на рисунку 4.

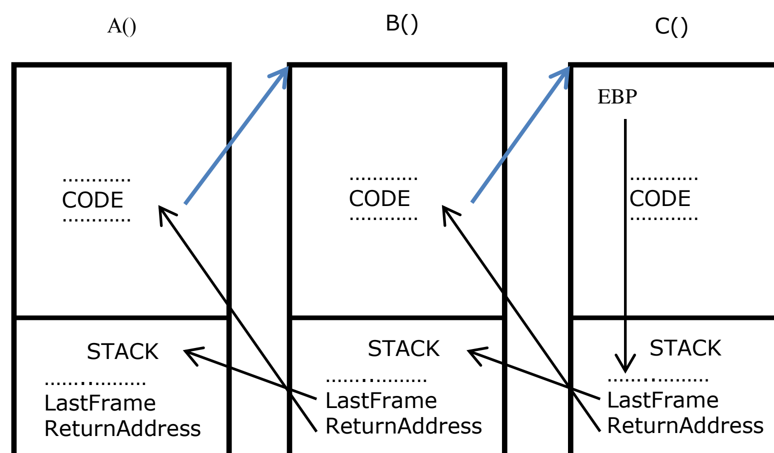

Рис. 4. **Виклики процедур, які належать до однієї SFC**

Трасування [8] – покрокове виконання програми (на відміну від бектрейсу, де прохід фактично передається назад, туди, звідки прийшло керування). Для цього возводиться прапорець TF в регістрі EFLAGS (8 біт), після чого кожна інструкція генерує виключення, і в обробнику виключень можна змінювати контекст потоку.

Варіант перехоплення WinAPI, застосовуючи дану технологію:

1. Знайти глобальну змінну, яка б використовувалась в цільовій процедурі.
2. Визначити NL процедури, в якій відбувається звернення до цієї змінної відносно цільової (можна статично, в WinAPI рідко змінюються процедури, які часто використовуються).
3. Зруйнувати вказівник, щоб він вказував на невірну адресу, зберігши його перед цим для відновлення.
4. У глибині перехоплюваної (цільової) процедури виникне виключення, в обробнику виключень виконати:
   a. Відновити зруйновану глобальну змінну.
   b. Бектрейс до цільової процедури.
   c. Підмінити адресу повернення, зберігши оригінальну в обробнику.
   d. Перезапустити інструкцію, на якій виникло виключення.
5. Коли процедура відпрацює, керування отримає обробник.

При використанні даного підходу є ряд нюансів в залежності від реалізації функції, яку потрібно перехопити.

Основні переваги використання даної технології наступні:

1. Даний вид перехоплення неможливо відслідкувати, так як кодосекції в пам’яті не змінюються.
2. У кожного потоку свій обробник виключень, тому немає ризиків, що іншій потік виконає щось, що ще змінюється в поточному (як при патчах).

Отже, в статті було досліджено сучасний стан інформаційного забезпечення у питанні методів перехоплення коду. Проведено аналіз існуючих методів, та запропоновано новий метод з використанням бектрейсу, графів та стекової маршрутизації. Розглянуто особливості програмних реалізацій розглянутих технологій. Встановлено переваги запропонованого методу перехоплення коду над існуючими.

Література

1. MSDN: Microsoft PE and COFF Specification. - [Електронний ресурс]. Режим доступу: http://msdn.microsoft.com/en-us/windows/hardware/gg463119.aspx.
2. Вікіпедія: Перехват. - [Електронний ресурс]. Режим доступу: http://ru.wikipedia.org/wiki/Перехват_(программирование)
3. Павел Блудов. Как подменить функцию API. - [Електронний ресурс]. Режим доступу: http://www.rsdn.ru/article/qna/baseserv/hookapi.xml#EIC
4. Игорь В. Филимонов. Методы перехвата API-вызовов в Win32. - [Електронний ресурс]. Режим доступу: http://www.rsdn.ru/article/baseserv/apicallsintercepting.xml
5. MSDN: stdcall.- [Електронний ресурс]. Режим доступу: http://msdn.microsoft.com/en-us/library/zxk0tw93.aspx
6. Вікіпедія: Граф потока управления .- [Електронний ресурс]. Режим доступу: http://ru.wikipedia.org/wiki/Граф_потока_управления
7. Вікіпедія: Дизассемблер длин.- [Електронний ресурс]. Режим доступу: http://ru.wikipedia.org/wiki/Дизассемблер_длин
8. Вікіпедія: Трасування.- [Електронний ресурс]. Режим доступу: http://uk.wikipedia.org/wiki/Трасування