IAT Hooking

מאת אוריאל מלין (Ratinho)

הקדמה

להבנה מיטבית של המאמר מומלץ לדעת C ואסמבלי ברמה בסיסית, וכן כדאי לעבור לפחות על תחילתו של המאמר של Zerith מגיליון מספר 10.

המטרה שלנו: ליצור "Crack" חיצוני ל-CrackMe שכתבתי

בחודש האחרון כתבתי CrackMe, ופרסמתי אותו באחד מהפורומים בארץ. מטרת ה-CrackMe הייתה לכתוב קובץ כלשהו ש"יגע" ב-CrackMe אך ורק כשהוא נטען לזיכרון ויבצע בו מניפולציה כלשהי, כך שעבור כל סיסמה הפלט שיוחזר יהיה "Good Job!".

ה-CrackMe נמצא בקובץ ה-ZIP המצורף למאמר, בשם: ExternalCrackme.exe (קישור לקובץ ה-ZIP ניתן למצוא בסוף המאמר)

```
00261000  $  55              PUSH EBP
00261001  .  8BEC            MOV EBP,ESP
00261003  .  83EC 24         SUB ESP,24
00261006  .  A1 04D02600     MOV EAX,DWORD PTR DS:[26D004]
0026100B  .  33C5            XOR EAX,EBP
0026100D  .  8945 FC         MOV [LOCAL.1],EAX
00261010  .  33C0            XOR EAX,EAX
00261012  .  68 B0B92600     PUSH external.0026B9B0                  ASCII "Enter password plz:\n"
00261017  .  8845 DC         MOV BYTE PTR SS:[EBP-24],AL
0026101A  .  8945 DD         MOV DWORD PTR SS:[EBP-23],EAX
0026101D  .  8945 E1         MOV DWORD PTR SS:[EBP-1F],EAX
00261020  .  8945 E5         MOV DWORD PTR SS:[EBP-1B],EAX
00261023  .  8945 E9         MOV DWORD PTR SS:[EBP-17],EAX
00261026  .  8945 ED         MOV DWORD PTR SS:[EBP-13],EAX
00261029  .  8945 F1         MOV DWORD PTR SS:[EBP-F],EAX
0026102C  .  8945 F5         MOV DWORD PTR SS:[EBP-B],EAX
0026102F  .  66:8945 F9      MOV WORD PTR SS:[EBP-7],AX
00261033  .  8845 FB         MOV BYTE PTR SS:[EBP-5],AL
00261036  .  E8 F5000000     CALL external.00261130
0026103B  .  6A 1F           PUSH 1F                                 Arg3 = 0000001F
0026103D  .  8D45 DC         LEA EAX,[LOCAL.9]
00261040  .  50              PUSH EAX                                Arg2 = 00361CD0
00261041  .  68 C8B92600     PUSH external.0026B9C8                  Arg1 = 0026B9C8 ASCII "%s"
00261046  .  E8 C6000000     CALL external.00261111                  external.01261111
0026104B  .  83C4 10         ADD ESP,10
0026104E  .  68 CCB92600     PUSH external.0026B9CC                  String2 = "ImUsingExternalSolution"
00261053  .  8D4D DC         LEA ECX,[LOCAL.9]
00261056  .  51              PUSH ECX                                String1 = 00000001 ???
00261057  .  FF15 00A02600   CALL DWORD PTR DS:[<&KERNEL32.lstrcmpA>] lstrcmpA
0026105D  .  85C0            TEST EAX,EAX
0026105F  .v 75 07           JNZ SHORT external.00261068
00261061  .  68 E4B92600     PUSH external.0026B9E4                  ASCII "Good Job!"
00261066  .v EB 05           JMP SHORT external.0026106D
00261068  >  68 F0B92600     PUSH external.0026B9F0                  ASCII "wrong dude!\n"
0026106D  >  E8 BC000000     CALL external.0026112E
```

כפי שניתן לראות, בסך הכל לא מדובר ב-CrackMe מסובך בכלל. המשתמש מתבקש להכניס סיסמה, הסיסמה נקלטת (ע"י scanf()), ונבדקת ע"י lstrcmpA ביחס למחרוזת "ImUsingExternalSolution". במידה והמחרוזות זהות- הפונקציה lstrcmpA תחזיר ב-EAX את הערך "0", ונתקדם לקראת הבדיקה:

```
TEST EAX,EAX
JNZ SHORT 00261068
```

במידה ו-EAX אינו שווה ל-0, כלומר: המחרוזות אינן שוות, לא יידלק דגל ה-zero flag, וה-JNZ יקפיץ ל-"bad boy" אך במידה ו-EAX אכן שווה ל-0 (המחרוזות שוות), נמשיך ל-"Good Job!".

ב-CrackMe הנ"ל ביקשתי ליצור Crack חיצוני בעיקר כדי לאתגר אנשים שיש להם ידע ברמה סבירה ברברסינג, ובנוסף ידע ברמה סבירה בתכנות, אך עדיין לא שילבו בין השניים.

מאמר זה יציג את הפתרון שלי ל-CrackMe הנ"ל.

קצת תיאוריה

כמו שכולנו יודעים (ומי שלא יודע: מוזמן לקרוא את המאמר של יוסף רייסין מגיליון 8 על "קבצי הרצה בתקופות השונות"), כיום קבצי ה-EXE הינם בפורמט Portable Executable או בקיצור PE. אולי לחלקכם הפורמט מוכר מהשימושים השונים שנעשה במידע שבו לצרכי רברסינג, אבל באופן מפתיע, יש לו עוד יעודים ושימושים ☺

כמו שניתן לראות בטבלה כאן, כותר ה-PE מורכב ממספר חלקים. נתמקד ברלוונטיים עבורנו:

מטרתו של כותר ה-PE להוות תאימות למגוון מערכות הפעלה (תוצרת מיקרוסופט כמובן), ולכן **החלק הראשון** מיועד עבור MS-Dos והוא נקרא גם MZ Header (שתי האותיות הראשונות משמשות כ- Magic Number לציון תחילת הקובץ, ונבחרו ע"ש אחד ממפתחי MS-Dos, Mark Zbikowski). תפקידו של חלק זה לבצע פעולות מסוימות במקרה שמערכת ההפעלה שמריצה את הקובץ הינה MS-Dos (כאשר נריץ דרך MS-Dos תוכנית שכתובה עבור Windows, תוצג לנו ההודעה המיתולוגית " This program cannot be run in DOS mode"). השדה הרלוונטי עבור מאמר זה הינו השדה e_lfanew המציין את ההיסט (Offset) מראש הקובץ לתחילת ה-PE Header עצמו.

החלק השני של הכותר, ובעצם עיקר ה-PE Header, שנקרא גם Windows NT information, מכיל מידע עבור ריצה של התוכנית בגרסאות Windows NT (מאז windows 2000 מערכת ההפעלה הביתית של מיקרוסופט מבוססת NT). גם חלק זה מחולק למספר חלקים, אך החלק הרלוונטי למאמר נקרא IMAGE_OPTIONAL_HEADER32.

בחלק זה ישנן הגדרות אופציונאליות לקובץ (ומכאן שמו, למרות שבדרך כלל כמעט כולן מוגדרות), דוגמאת ImageBase ו-EntryPoint (שוודאי מוכרות לרוב הקוראים מעולם הרברסינג). גם כאן, יש חלק אחד שרלוונטי עבורנו, והוא IMAGE_DATA_DIRECTORY (אני מקווה שאתם עוד עוקבים עם הטבלה ☺), שהוא מערך, המכיל 16 תיקיות האחראיות על קוניפיגורציות שונות.

במערך זה נמצאים באינדקס 1: ה-Import Table, ובאינדקס 12: ה-IAT המדוברת, פירושה: Import Address Table.

אז מה יש ב-Import Table?

ה-Import Table הוא מערך עם מידע עבור כל אחד מקבצי ה-DLL שבהם התוכנית תשתמש. עבור כל DLL יש מצביע (Pointer) אל מבנה בגודל 20 בתים (bytes), בשם IMAGE_IMPORT_DESCRIPTOR, המכיל מידע על קובץ ה-DLL.

הערה שתקפה לכל המערכים שאני עומד להזכיר בקרוב, כל עוד לא ציינתי אחרת, היא שסוף המערך מסומן ע"י תא בגודל הרגיל, אך מלא באפסים.

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    union {
        DWORD   Characteristics;
        DWORD   OriginalFirstThunk;
    } DUMMYUNIONNAME;
    DWORD   TimeDateStamp;
    DWORD   ForwarderChain;
    DWORD   Name;
    DWORD   FirstThunk;
} IMAGE_IMPORT_DESCRIPTOR;
```

בשדה Name ישנה כתובת (RVA, כתובת יחסית ל-Image Base) המכילה מחרוזת עם שם ה-DLL. השדות OriginalFirstThunk ו-FirstThunk מכילים את הכתובות (RVA) של ראשי מערכים מסוג:

```
typedef struct _IMAGE_THUNK_DATA32 {
    union {
        DWORD ForwarderString;      // PBYTE
        DWORD Function;             // PDWORD
        DWORD Ordinal;
        DWORD AddressOfData;        // PIMAGE_IMPORT_BY_NAME
    } u1;
} IMAGE_THUNK_DATA32;
```

כאשר OriginalFirstThunk מכיל ב-AddressOfData (כמובן שזה לא משנה איזה שדה בוחרים מכיוון שכולם מאותו טיפוס, ומכיוון שזה Union הם חולקים את אותו הזיכרון, אני מבדיל בין השדות רק כדי שיהיה יותר ברור) את הכתובת של Import Lookup Table, מערך בעל פרטים על הפונקציות אותן אנו רוצים לייבא, המכיל מבנים מסוג:

```
typedef struct _IMAGE_IMPORT_BY_NAME {
    WORD    Hint;
    BYTE    Name[1];
} IMAGE_IMPORT_BY_NAME, *PIMAGE_IMPORT_BY_NAME;
```

כך שבכל רשומה יש שדה בגודל 2 בתים בשם Hint (שדה העוזר ל-loader למצוא את הפונקציה הנוכחית ב-Export Table של ה-DLL), ומיד אחריו שם הפונקציה (שם הפונקציה רק מסומן בתיעוד כמערך עם איבר אחד, אך בפועל יש שם מחרוזת ascii רגילה המסתיימת ב-Null-terminator).

כאן נכנסת לתמונה ה-Import Address Table, או בקיצור IAT. IAT הינה טבלה (או בזיכרון - בסך הכל מערך של תאים בעלי גודל של 4 בתים), שתפקידה לומר לנו באיזה כתובת נמצאת כל פונקציה שייבאנו מה-DLL-ים השונים. כאשר ישנה קריאה לפונקציה מ-DLL, או אפילו לסתם פונקציית Win32api הנמצאת ב-kernel32.dll (למשל), הלינקר בודק מה המיקום (offset) של הפונקציה ב-Import Lookup Table (בזמן ה-Linking, לא בזמן הריצה), ומבצע קריאה לאותו המיקום ב-IAT. הקריאה נראית משהו כמו:

```
CALL DWORD PTR DS:[IAT_BASE+FUNCTUIN_OFFSET]
```

השדה FirstThunk מכיל את הכתובת (RVA) של ה-IAT. גם ב-IAT יש לנו מערך, מערך של IMAGE_THUNK_DATA32.

ב-Image ששמור ב-HD, גם ה-IAT תכיל כתובות של המידע על הפונקציות (ובעצם הערכים במערך של IAT יהיו בדיוק כמו הערכים ב Import Lookup Table, וניגש אליהן דרך השדה AddressOfData. לעומת זאת כאשר ה-loader ממפה את הפונקציות הוא משכתב את ה-IAT, וכעת כל תא מצביע, מהשדה Function, לכתובת (VA, כתובת וירטואלית מלאה) של הפונקציה ב-DLL שנטען לזיכרון.

כדאי להדגיש: ברגע שנשנה את הערך ב-IAT לכתובת של פונקציה אחרת, למעשה נבצע את ה-hook, והתוכנית תקפוץ לפונקציה החדשה שנגדיר במקום למקורית. במקרה שלנו בחרתי לבצע את ה-hook על הפונקציה של השוואת המחרוזות: lstrcmpA.

אז איך עושים את זה?

מה שנעשה זה לכתוב קובץ DLL, שאחראי לבצע hook לתהליך שהוא רץ בתוכו. כמו שראינו כשדיבגנו את הקובץ, התוכנית קוראת ל lstrcmpA ובודקת האם התוצאה שווה ל-0. במידה וכן, נמשיך לפלט הנכון. לכן נבצע את ה-hook שלנו לפונקציה lstrcmpA, ונדאג שבמקומה תקרא פונקציה שתחזיר תמיד 0.

אחרי שנכתוב את קובץ הDLL שאחראי על ה-hook, נשתמש בטכניקה הנקראת DLL Injection, המאפשרת להריץ קוד בתהליך מרוחק. מכיוון שאוראל ארד כבר כתב על DLL Injection בפרוטרוט בגיליון 13 וכן גם בגליון זה ישנה כתבה בנושא, אני אחסוך בהסברים על המזריק, ואסתפק בלצרף את הקוד של המזריק שאני כתבתי (אפרופו המזריק, בגרסא שצרפתי צריך להכניס את ה-PID של הקובץ שרוצים להזריק אליו, בקוד הוספתי בהערה פונקציה, באדיבות vbCrLf, שמקבלת את שם התהליך, ומחזירה את ה-PID שלו).

מתחילים עם ה-DLL:

בטעינת ה-DLL לתהליך נקרא לפונקציה IAThooking (אותה נכתוב עוד מעט)

```
BOOL APIENTRY DllMain(HINSTANCE hInst, DWORD reason, LPVOID reserved)
{
	switch (reason)
	{
	case DLL_PROCESS_ATTACH:
		IAThooking(GetModuleHandleA(NULL),TARGET_FUNCTION,newlstrcmpA);
	}
}
```

ראשית, עלינו להגיע לראש ה-PE, ואם נדייק - לראש ה-MZ. ניתן להגיד לשם בעזרת הפונקציה המובנית ב-win32api, GetModuleHandleA. כאשר נשלח לה כפרמטר NULL, הפונקציה תחזיר את ה- Handle (במקרה שלנו פשוט את הכתובת) לקובץ ה-EXE שיצר את התהליך. כך שהפרמטר הראשון מעביר את הכתובת של תחילת קובץ ה-EXE.

הפרמטר השני מעביר את מאקרו עם שם הפונקציה (למקרה שבעתיד נרצה לבצע hook על פונקציה שונה..), והפרמטר השלישי מעביר מצביע לפונקציה החדשה (כתובת הפונקציה החדשה), לה נרצה לקרוא במקום lstrcmpA.

זה הרגע להזכיר ש-lstrcmpA הינה פונקציית Win32Api, מה שאומר שהיא נקראת בקונבנציית הקריאה stdcall- הפרמטרים נדחפים למחסנית מימין לשמאל, הערך המוחזר מועבר ב-EAX, ואילו תפקידה של הפונקציה (ולא של הפונקציה שקראה לה) לנקות בחזרה את המחסנית (להעיף את הפרמטרים).

אב הטיפוס של הפונקציה נראה כך:

```
int WINAPI lstrcmp(__in  LPCTSTR lpString1,__in  LPCTSTR lpString2);
```

מה שאומר שהפונקציה החדשה שנכתוב צריכה להיות באותו מבנה, על מנת שנשמור על המחסנית נקייה והתוכנית תוכל להמשיך כסדרה. אפשרות מימוש אחת היא:

```
int WINAPI newlstrcmpA(LPCSTR a,LPCSTR b)
{
	MessageBoxA(NULL,"hook called","hehe",MB_OK);
	return 0;
}
```

כמובן שההודעה מוקפצת רק כדי להראות שאכן הופעל ה-hook, וכמו שאמרנו, הפונקציה תחזיר תמיד 0.
אפשרות שניה: ניתן להכריז על הפונקציה כ:

```
__declspec( naked )
```

שאומר במקרה הזה הקומפיילר לא נוגע בפרולוג והאפילוג של הפונקציה (פקודות האסמבלי בתחילת הפונקציה, האחראיות בדרך כלל לבנות מסגרת חדשה במחסנית, עבור המשתנים הלוקאלים של הפונקציה ופקודות האסמבלי בסוף הפונקציה, האחראיות על שחזור המחסנית וניקויה) - אנחנו צריכים לדאוג שהמחסנית תישאר תקינה בחזרה מהפונקציה בעזרת מספר הוראות. במקרה שלנו הפונקציה מקבלת שני מצביעים, כל אחד בגודל 4 בתים. מכיוון שעלינו לדאוג לנקות את הפרמטרים, נצטרך לנקות 8 בתים במחסנית בחזרה. לדוגמא:

```
__declspec( naked ) int newlstrcmpA()
{
	_asm{
		XOR eax,eax;
		RETN 8;
	}
}
```

נמשיך בגוף הפונקציה:

```
bool IAThooking(HMODULE hInstance,LPCSTR targetFunction,PVOID newFunc)
```

נצהיר על מספר משתני עזר:

```
PIMAGE_IMPORT_DESCRIPTOR importedModule;
PIMAGE_THUNK_DATA pFirstThunk,pOriginalFirstThunk;
PIMAGE_IMPORT_BY_NAME pFuncData;
```

כמו שמובן מהשמות, pFirstThunk יצביע על ה-IAT, pOriginalFirstThunk יצביע על ה- Import Lookup Table (נעזר בה על מנת לעבור על כל הפונקציות ולהשוות את שם הפונקציה אם הפונקציה המבוקשת), ואילו pFuncData יצביע כל פעם על המידע של כל רשומה ב-Import Lookup Table. עכשיו נצטרך להכניס לתוך importedModule את ראש מערך ה-Import Table.

```
importedModule=getImportTable(hInstance);
```

הפונקציה מנווטת אל ה-Import Table בדיוק עפ"י התהליך שתיארתי בחלק התיאורטי של המאמר:

```
PIMAGE_IMPORT_DESCRIPTOR getImportTable(HMODULE hInstance)
{
```

הגדרת משתני עזר:

```
	PIMAGE_DOS_HEADER dosHeader;
	IMAGE_OPTIONAL_HEADER optionalHeader;
	PIMAGE_NT_HEADERS ntHeader;
	IMAGE_DATA_DIRECTORY dataDirectory;
```

כותר ה-MZ:

```
	dosHeader=(PIMAGE_DOS_HEADER)hInstance;
```

כותר ה-NT_HEADERS (כמו שאמרנו בשדה e_lfanew שנמצא ב-MZ יש היסט לתחילת ה-PE עצמו, ולכן נחבר את ההיסט עם ה-ImageBase, כתובת הבסיס של תחילת קובץ ה-EXE):

```
ntHeader=(PIMAGE_NT_HEADERS)((PBYTE)dosHeader+dosHeader->e_lfanew);
```

חשוב מאד לבצע casting למבנה dosHeader למצביע של בתים, אחרת בפעולות החיבור הקומפיילר יתייחס להוספה של ההיסט כתזוזה של מספר מבני IMAGE_DOS_HEADER, ולא ככתובת בבתים). מכאן נמשיך ל-Optional_header:

```
optionalHeader=(IMAGE_OPTIONAL_HEADER)(ntHeader->OptionalHeader);
```

מכאן נשאר לנו רק למצוא את ה-dataDirectory המתאים עבור ה-Import Table. במקרה שלנו נצטרך את האינדקס מספר 1:

```
dataDirectory=(IMAGE_DATA_DIRECTORY)(optionalHeader.DataDirectory[IMPORT_TABLE_OFFS
ET]);
```

וכעת להחזיר את הכתובת המלאה של ה-Import Table:

```
return (PIMAGE_IMPORT_DESCRIPTOR)((PBYTE)hInstance + dataDirectory.VirtualAddress);
}
```

כעת, importedModule מצביע לרשומה הראשונה ב-Import Table, נעבור על כל הטבלה, ועבור כל רשומה נעבור על הפונקציות שנמצאות בה (וגם נדפיס אותן ואת שם הDLL, סתם בשביל הכיף), נבצע השוואה עם השם בכל רשומה ב-Import Table Lookup, ובמידה וזהה, נשנה את הערך המתאים ב-IAT, לפונקציה החדשה שלנו:

מעבר על הרשומה כל עוד לא הגענו לסוף (מסומן ע"י אפסים):

```
while(*(WORD*)importedModule!=0)
{
```

הדפסת שם המודול:

```
printf("\n%s - %x:\n---------\n",(char*)((PBYTE)hInstance+importedModule->Name));
 השמות עבורImport Lookup Table ועבור ה IAT:הנוכחיים
pFirstThunk=(PIMAGE_THUNK_DATA)((PBYTE)hInstance+ importedModule->FirstThunk);
pOriginalFirstThunk=(PIMAGE_THUNK_DATA)((PBYTE)hInstance+ importedModule-
>OriginalFirstThunk);
pFuncData=(PIMAGE_IMPORT_BY_NAME)((PBYTE)hInstance+ pOriginalFirstThunk-
>u1.AddressOfData);
```

מעבר על ה-Import Lookup Table ועל ה-IAT (בהתאמה) של כל מודול:

```
	while(*(WORD*)pFirstThunk!=0 && *(WORD*)pOriginalFirstThunk!=0)
	{
```

הדפסת שמות הפונקציות וכתובתן:

```
printf("%X %s\n",pFirstThunk->u1.Function,pFuncData->Name);
```

בדיקה האם הגענו לפונקציה הנכונה (בעיקרון גם כדאי לבדוק האם אנו נמצאים במודול הנכון (במקרה שלנו Kernel32.dll), אך כאן חסכתי זאת משום שאין פה DLL-ים נוספים:

```
if(strcmp(targetFunction,(char*)pFuncData->Name)==0)
{
	printf("Hooking... \n");
```

שליחה לפונקציה (תפורט עוד רגע) שתשכתב את הIAT במיקום שנבחר, עם הכתובת של הפונקציה החדשה:

```
	if(rewriteThunk(pFirstThunk,newFunc))
		printf("Hooked %s successfully :)\n",targetFunction);
}
```

קידום הדסקריפטור ב-Import Lookup Table וב-IAT:

```
		pOriginalFirstThunk++;
pFuncData=(PIMAGE_IMPORT_BY_NAME)((PBYTE)hInstance+pOriginalFirstThunk-
>u1.AddressOfData);
		pFirstThunk++;
	}
```

ולבסוף, קידום הדסקריפטור של המודולים:

```
	importedModule++; //next module (DLL)
}
return true;
```

כמובן שניתן גם לחסוך את כל ההדפסות, ואפילו לעצור את הלולאות ברגע שאיתרנו את המיקום של הפונקציה ב-IAT, כל מה שנתתי כאן זה רק לשם המחשה. וכמובן הפונקציה שאחראית על שכתוב ה-IAT:

```
bool rewriteThunk(PIMAGE_THUNK_DATA pThunk,void* newFunc)
{
```

משתני עזר שישמרו את ההרשאות שיש לדף:

```
	DWORD CurrentProtect;
	DWORD junk;
```

מתן הרשאת כתיבה וקריאה לדף המבוקש:

```
	VirtualProtect(pThunk,4096, PAGE_READWRITE, &CurrentProtect);
```

שמירת הכתובת המקורית (לדוגמא למקרה שנרצה לבצע הוק כללי יותר, שדואג לבצע פעולה מסוימת ולאחר מכן להמשיך את זרימת התוכנית הרגילה):

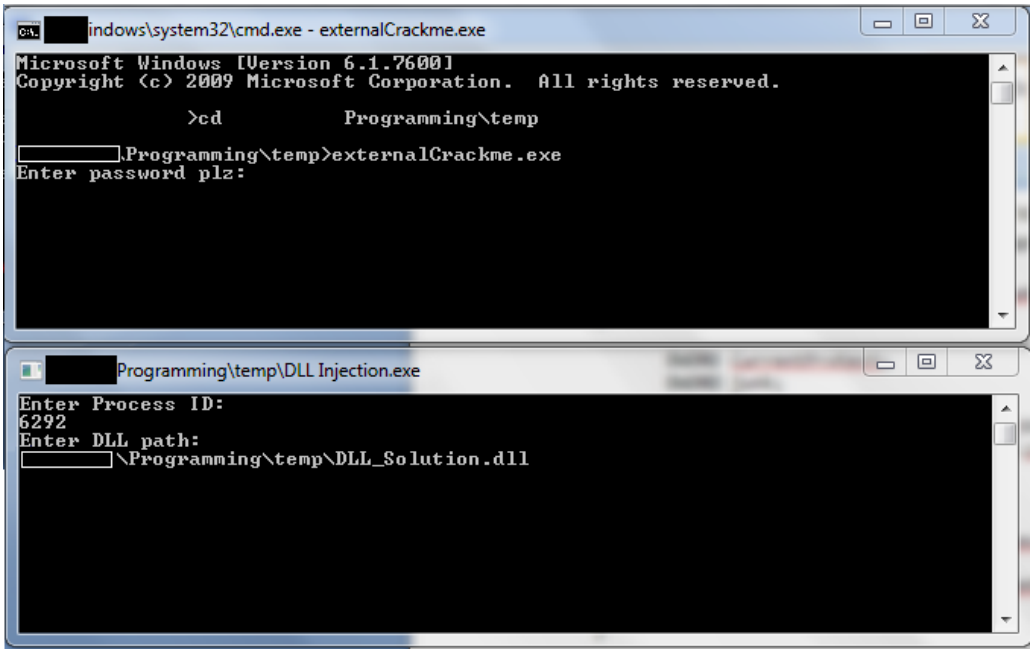
```
	sourceAddr=pThunk->u1.Function;
```

שינוי השדה ב-IAT לכתובת החדשה:

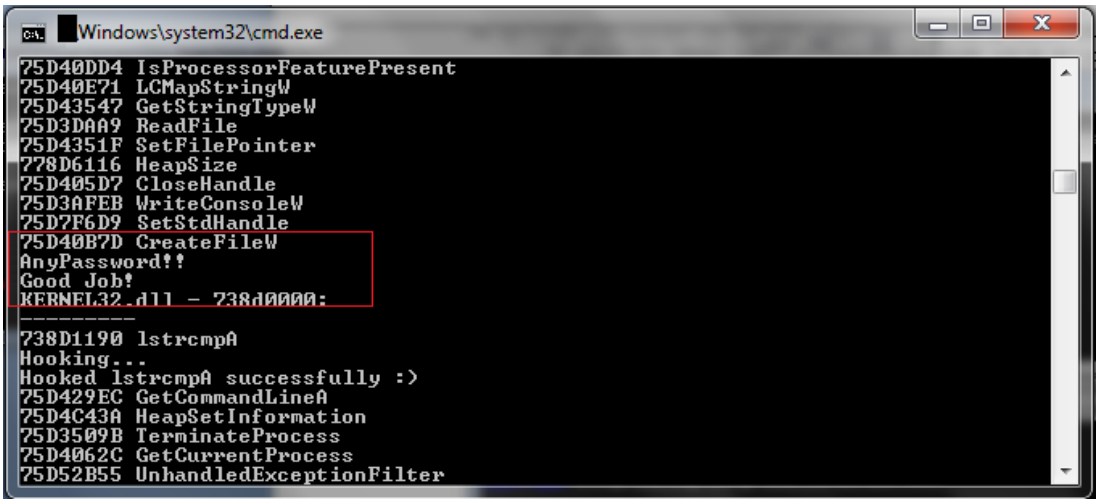
```
        pThunk->u1.Function=(DWORD) newFunc;
```

שחזור ההרשאות הקודמות שהיו בדף:

```
        VirtualProtect(pThunk,4096, CurrentProtect,&junk);
        return true;
}
```



כעת, רק נותר לקמפל את ה-DLL המלא (ראה קוד מצורף), להריץ את התוכנית, להזריק את ה-DLL - והופ, כל סיסמה שנכניס תיתן לנו את הודעות ה-Good Job! המיוחלת:



נלחץ אנטר, יתבצע ההוק, תצא רשימה עם כל הפונקציות, לאחר מכן נכניס את הסיסמה ונלחץ אנטר שנית.

(אגב, בקובץ המצורף הוספתי ביציאה מה-DLL קריאה נוספת לפונקציה של ההוק, הפעם עם הכתובת של הפונקציה המקורית, למקרה שנרצה להחזיר את הערך המקורי שהיה ב-IAT ולהמשיך את התוכנית כרגיל)

סיכום

לדעתי IAT Hooking היא אחת מהשיטות היותר אלגנטיות ויפות לבצע הוקים במערכת. אמנם כמו שאורי כתב במאמר שלו, קל מאד לעלות על השינויים, רק צריך לסרוק את ה-IAT ולבדוק האם יש כתובות מחוץ למרחב הכתובת של ה-DLL, אך הטכניקה גמישה מאד וברגע שממשים פעם ראשונה היא מאפשרת לשנות פונקציות רבות, רק ע"י שינוי שם הפונקציה וכתיבת פונקציה חדשה שתחליף אותה.

אני מקווה שמאמר זה תרם קצת להבנת הטכניקה, בסך הכל לא ראיתי מאמר מפורט כזה בעברית בנושא. אני רוצה להודות לאפיק באותה ההזדמנות שסוף סוף הצליח לשכנע אותי גם לכתוב מאמר, ולא רק לערוך מאמרים אחרים.

את כלל קטעי הקוד והקבצים הבינאריים ניתן להוריד מהקישור הבא:

http://www.digitalwhisper.co.il/files/Zines/0x12/IATHooking.zip

אגב, באותו הפורום קבלתי מספר פתרונות כגון:

- שינוי השורה JNZ SHORT 00261068 ל-0x90 0x90, כלומר nop nop, ולבצע patch לזיכרון התוכנית ולגרום לכך שלעולם לא נקפוץ ל-bad boy.
- Inline hooking ל-lstrcmpA ולגרום ככה שתחזיר תמיד 0.
- פתרון מגניב של Zerith שטוען דרייבר שדואג לבצע את השינויים.
- בגרסא הראשונה, שקומפלה עם ה-c++ runtime, אפילו קבלתי פתרון עם DLL hijacking הגורם ל-scanf() להחזיר את המחרוזת "ImUsingExternalSolution".

את הת'רד המקורי אפשר למצוא כאן.

מקורות

http://www.codeproject.com/KB/system/inject2exe.aspx#PortableExecutablefileformat2

http://www.reverse-engineering.info/SystemInformation/iat.html

http://sandsprite.com/CodeStuff/IAT_Hooking.html

וכמובן, Microsoft MSDN:

http://msdn.microsoft.com/en-us/library/ms123401.aspx