digital whisper
Technologic Papers

Portable Executable

מאת Spl0it

הקדמה - מה זה PE?

ויקיפדיה: "PE (קיצור של Portable Executable) הוא פורמט שפותח ע"י Microsoft עבור קבצי ריצה, קבצי אובייקט, ספריות קישור-דינמי (DLL), קבצי פונטים (FON) ועוד אשר משומשים בגרסאות ה-32 וה-64 ביט של מערכות המשתמשות במערכת ההפעלה Windows. PE הוא מבנה נתונים אשר מקבץ את המידע ההכרחי בשביל שה-Loader של Windows יצליח לנהל את הקוד בזמן ריצה".

סוגי הקבצים הנפוצים ביותר המשתמשים בפורמט PE:

- exe - קובץ ריצה
- dll - ספריית קישור-דינמי
- sys/drv - קובץ מערכת (דרייבר לקרנל)
- ocx - קובץ שליטה ב-ActiveX
- cpl - לוח בקרה
- scr - שומר מסך

הערה: לקבצי lib. (ספריות סטטיות) יש פורמט שונה, לא PE.

מערכת ההפעלה Windows עושה שימוש בקבועים הנ"ל כדי לייצג גדלים של משתנים:

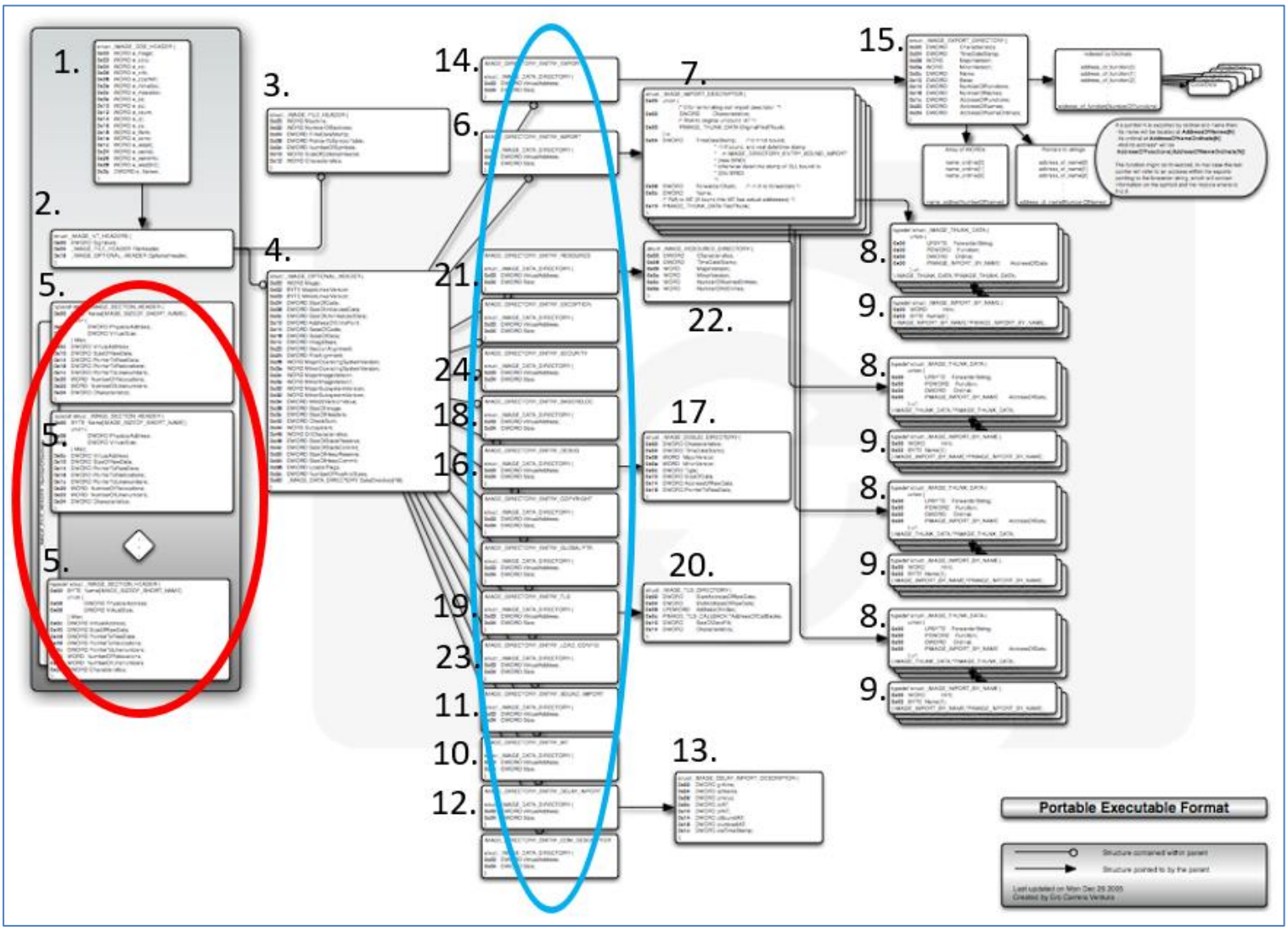
גודל	טיפוס
בית 1	CHAR (Character)
2 בתים	WORD
2 בתים	SHORT (Short Integer)
4 בתים	DWORD (Double Word)
4 בתים	LONG (Long Integer)
8 בתים	QWORD (Quad Word)
8 בתים	LONGLONG

כלים לחקירת ה-PE:

- PEView - לטובת הסתכלות על ה-PE של קבצים בפורמט זה

- CFF Explorer - אותו דבר, אך עם פיצ'רים נוספים כגון עריכת ה-PE בהקסדצימלי והמרת הקובץ לשפת אסמבלי
- WinDbg - עבור ניפוי שגיאות (Debugging) בסיסי

פורמט ה-PE נראה כך (התמונה ממוספרת כדי שההסברים בהמשך המאמר יהיו ברורים יותר. לתמונה "נקייה" יותר, לחצו כאן):



DOS-Header

המבנה הראשון, הנמצא ב-Offset 0x0, נקרא **DOS-Header** והוא נראה כך (מספר 1 בתמונת פורמט ה-PE):

```
struct _IMAGE_DOS_HEADER {
0x00  WORD e_magic;
0x02  WORD e_cblp;
0x04  WORD e_cp;
0x06  WORD e_crlc;
0x08  WORD e_cparhdr;
0x0a  WORD e_minalloc;
0x0c  WORD e_maxalloc;
0x0e  WORD e_ss;
0x10  WORD e_sp;
0x12  WORD e_csum;
0x14  WORD e_ip;
0x16  WORD e_cs;
0x18  WORD e_lfarlc;
0x1a  WORD e_ovno;
0x1c  WORD e_res[4];
0x24  WORD e_oemid;
0x26  WORD e_oeminfo;
0x28  WORD e_res2[10];
0x3c  DWORD e_lfanew;
};
```

```
typedef struct _IMAGE_DOS_HEADER {      // DOS .EXE header
    WORD    e_magic;                    // Magic number
    WORD    e_cblp;                     // Bytes on last page of file
    WORD    e_cp;                       // Pages in file
    WORD    e_crlc;                     // Relocations
    WORD    e_cparhdr;                  // Size of header in paragraphs
    WORD    e_minalloc;                 // Minimum extra paragraphs needed
    WORD    e_maxalloc;                 // Maximum extra paragraphs needed
    WORD    e_ss;                       // Initial (relative) SS value
    WORD    e_sp;                       // Initial SP value
    WORD    e_csum;                     // Checksum
    WORD    e_ip;                       // Initial IP value
    WORD    e_cs;                       // Initial (relative) CS value
    WORD    e_lfarlc;                   // File address of relocation table
    WORD    e_ovno;                     // Overlay number
    WORD    e_res[4];                   // Reserved words
    WORD    e_oemid;                    // OEM identifier (for e_oeminfo)
    WORD    e_oeminfo;                  // OEM information; e_oemid specific
    WORD    e_res2[10];                 // Reserved words
    LONG    e_lfanew;                   // File address of new exe header
  } IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

הערה להמשך המאמר: שדות המסומנים ב**כחול** הם שדות שחשובים לנו. לא אוכל לכסות את כלל השדות במאמר זה, לכן אכסה רק את השדות החשובים.

אכפת לנו בעצם רק משני ערכים: הערך בשדה **e_magic** והערך בשדה **e_lfanew** (כל השאר אלו דברים שקשורים ל-DOS).

e_magic אומר לנו באיזה סוג קובץ מדובר והערך בשדה **e_lfanew** הוא ה-Offset למבנה נתונים הבא (ה-PE Header הבא). הערך של **e_magic** יהיה 'MZ', שזה בעצם מארק זביקובסקי (Mark Zbikowski), הבחור שפיתח את MS-DOS.

דרך אגב, עבור רוב התוכנות ב-Windows, ה-DOS header כוללת פיסת תוכנית של DOS אשר מדפיסה את הפלט: "This program cannot be run in DOS mode" (תוכנית זאת איננה יכולה לרוץ ב-DOS). לדוגמא, אם מישהו ינסה להריץ את פנקס הרשימות (notepad.exe) בתוך DOS, הפלט שיוחזר יהיה "This program cannot be run in DOS mode".

NT-Header

אחרי ה-DOS-Header, אנו מגיעים ל-**NT-Header**. הוא נראה כך (מספר 2 בתמונת פורמט ה-PE):

```
struct _IMAGE_NT_HEADERS {
0x00  DWORD Signature;
0x04  _IMAGE_FILE_HEADER FileHeader;
0x18  _IMAGE_OPTIONAL_HEADER OptionalHeader;
};
```

```
typedef struct _IMAGE_NT_HEADERS {
    DWORD Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER32 OptionalHeader;
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

זהו מבנה נתונים אשר טמועים בתוכו 2 מבני נתונים נוספים, ושמותיהם **FILE_HEADER** ו-**OPTIONAL_HEADER**.

Signature יהיה שווה לערך 0x00004550 (המחרוזת "PE" בייצוג ASCII) בתוך DWORD. אחרת, הוא יחזיק ב-2 struct-ים אחרים אשר טמועים ב-PE.

FILE_HEADER

לאחר מכן, נמצא השדה **FILE_HEADER**. שדה זה הוא מבנה נתונים אשר מוטבע בתוך ה-NT-Header ונראה כך (מספר 3 בתמונת פורמט ה-PE):

```
struct _IMAGE_FILE_HEADER {
0x00  WORD Machine;
0x02  WORD NumberOfSections;
0x04  DWORD TimeDateStamp;
0x08  DWORD PointerToSymbolTable;
0x0c  DWORD NumberOfSymbols;
0x10  WORD SizeOfOptionalHeader;
0x12  WORD Characteristics;
};
```

```
typedef struct _IMAGE_FILE_HEADER {
    WORD    Machine;
    WORD    NumberOfSections;
    DWORD   TimeDateStamp;
    DWORD   PointerToSymbolTable;
    DWORD   NumberOfSymbols;
    WORD    SizeOfOptionalHeader;
    WORD    Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

Machine הוא הערך שקובע את ארכיטקטורת המעבד שהתוכנה אמורה לעבוד לפיה. ערך זה הוא האינדיקציה הראשונית שלנו בנוגע להאם מדובר בקובץ מסוג 32 או 64 ביט. אם הערך הוא **014C**, מדובר בבינארי מסוג x86 (כתוב באסמבלי מסוג 32 ביט), הידוע בשמו **PE32**. אם הערך הוא **8664**, מדובר בבינארי מסוג x86-x64 (בינארי מסוג AMD64, כתוב באסמבלי מסוג 64 ביט), הידוע בשמו **PE32+**.

NumberOfSections הוא כמות ה-Section Header-ים הקיימים (מסומנים בעיגול **אדום** בתמונת פורמט ה-PE).

TimeDateStamp הוא ערך המצייג תאריך ב-Unix (Unix timestamp, כמות השניות שעברו מאז epoc, כאשר epoc הוא 00:00:00 ב-1 בינואר, 1970) וערך זה נקבע בזמן קישור (At link time). שדה זה אומר לנו מתי הקובץ קומפל. ערך זה משומש הרבה בחקירת נוזקות, אך תקחו בחשבון שהתוקף יכול לשנות את ערך זה, לכן אי אפשר לסמוך על אמינותו.

Characteristics מכיל הגדרות לקובץ, כגון:

```
#define IMAGE_FILE_EXECUTABLE_IMAGE          0x0002
// File is executable  (i.e. no unresolved external references).
#define IMAGE_FILE_LINE_NUMS_STRIPPED        0x0004
// Line numbers stripped from file.
#define IMAGE_FILE_LARGE_ADDRESS_AWARE       0x0020
// App can handle >2gb addresses
#define IMAGE_FILE_32BIT_MACHINE             0x0100
// 32 bit word machine.
#define IMAGE_FILE_SYSTEM                    0x1000
// System File.
#define IMAGE_FILE_DLL                       0x2000
// File is a DLL.
```

לדוגמא, אם ערכו של שדה זה יהיה 0x2002 אז קובץ זה הוא DLL וגם קובץ ריצה, זאת כתוצאה מהחישוב של 0x0002+0x2000.

מבנה הנתונים השני המוטמע בתוך ה-NT-Header הוא **OPTIONAL_HEADER** והוא נראה כך (מספר 4 בתמונת פורמט ה-PE):

```
struct _IMAGE_OPTIONAL_HEADER {
0x00  WORD Magic;
0x02  BYTE MajorLinkerVersion;
0x03  BYTE MinorLinkerVersion;
0x04  DWORD SizeOfCode;
0x08  DWORD SizeOfInitializedData;
0x0c  DWORD SizeOfUninitializedData;
0x10  DWORD AddressOfEntryPoint;
0x14  DWORD BaseOfCode;
0x18  DWORD BaseOfData;
0x1c  DWORD ImageBase;
0x20  DWORD SectionAlignment;
0x24  DWORD FileAlignment;
0x28  WORD MajorOperatingSystemVersion;
0x2a  WORD MinorOperatingSystemVersion;
0x2c  WORD MajorImageVersion;
0x2e  WORD MinorImageVersion;
0x30  WORD MajorSubsystemVersion;
0x32  WORD MinorSubsystemVersion;
0x34  DWORD Win32VersionValue;
0x38  DWORD SizeOfImage;
0x3c  DWORD SizeOfHeaders;
0x40  DWORD CheckSum;
0x44  WORD Subsystem;
0x46  WORD DllCharacteristics;
0x48  DWORD SizeOfStackReserve;
0x4c  DWORD SizeOfStackCommit;
0x50  DWORD SizeOfHeapReserve;
0x54  DWORD SizeOfHeapCommit;
0x58  DWORD LoaderFlags;
0x5c  DWORD NumberOfRvaAndSizes;
0x60  _IMAGE_DATA_DIRECTORY DataDirectory[16];
};
```

גרסאת 64 ביט:

```
typedef struct _IMAGE_OPTIONAL_HEADER64 {
    WORD        Magic;
    BYTE        MajorLinkerVersion;
    BYTE        MinorLinkerVersion;
    DWORD       SizeOfCode;
    DWORD       SizeOfInitializedData;
    DWORD       SizeOfUninitializedData;
    DWORD       AddressOfEntryPoint;
    DWORD       BaseOfCode;
    ULONGLONG   ImageBase;
    DWORD       SectionAlignment;
    DWORD       FileAlignment;
    WORD        MajorOperatingSystemVersion;
    WORD        MinorOperatingSystemVersion;
    WORD        MajorImageVersion;
    WORD        MinorImageVersion;
    WORD        MajorSubsystemVersion;
    WORD        MinorSubsystemVersion;
    DWORD       Win32VersionValue;
    DWORD       SizeOfImage;
    DWORD       SizeOfHeaders;
    DWORD       CheckSum;
    WORD        Subsystem;
    WORD        DllCharacteristics;
    ULONGLONG   SizeOfStackReserve;
    ULONGLONG   SizeOfStackCommit;
    ULONGLONG   SizeOfHeapReserve;
    ULONGLONG   SizeOfHeapCommit;
    DWORD       LoaderFlags;
    DWORD       NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_
                                    DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER64, *PIMAGE_OPTIONAL_HEADER64;
```

גרסאת 32 ביט:

```
typedef struct _IMAGE_OPTIONAL_HEADER {
    WORD    Magic;
    BYTE    MajorLinkerVersion;
    BYTE    MinorLinkerVersion;
    DWORD   SizeOfCode;
    DWORD   SizeOfInitializedData;
    DWORD   SizeOfUninitializedData;
    DWORD   AddressOfEntryPoint;
    DWORD   BaseOfCode;
    DWORD   BaseOfData;
    DWORD   ImageBase;
    DWORD   SectionAlignment;
    DWORD   FileAlignment;
    WORD    MajorOperatingSystemVersion;
    WORD    MinorOperatingSystemVersion;
    WORD    MajorImageVersion;
    WORD    MinorImageVersion;
    WORD    MajorSubsystemVersion;
    WORD    MinorSubsystemVersion;
    DWORD   Win32VersionValue;
    DWORD   SizeOfImage;
    DWORD   SizeOfHeaders;
    DWORD   CheckSum;
    WORD    Subsystem;
    WORD    DllCharacteristics;
    DWORD   SizeOfStackReserve;
    DWORD   SizeOfStackCommit;
    DWORD   SizeOfHeapReserve;
    DWORD   SizeOfHeapCommit;
    DWORD   LoaderFlags;
    DWORD   NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_
                NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

אני יודע שקוראים לשדה זה Optional Header, אך שלא תטעו, שדה זה לא אופציונלי בכלל והוא אפילו שדה חובה.

Magic משומש ע"י ה-Loader של מערכת ההפעלה כדי לקבוע האם להתייחס לקובץ זה בתור קובץ מסוג 32 או 64 ביט. לכן, **Magic** הוא הערך שקובע לנו את סוג הקובץ. ההבדל בינו לבין שדה ה-**Machine** הוא ש-**Machine** מציין את ארכיטקטורת המעבד ונותן לנו אינדיקציה ראשונית בלבד בנוגע להאם מדובר בקובץ מסוג 32 או 64 ביט.

שדה ה-**Magic** באמת קובע באיזה סוג קובץ מדובר. אם נשנה את הערך בשדה **Machine** התוכנה תעבוד כרגיל, אך אם נשנה את שדה ה-**Magic** התוכנה תקרוס, כתוצאה מההבדלים בגרסאות ה-32 וה-64 ביט של ה-**OPTIONAL_HEADER**. מכאן אנו מסיקים שהשדות ב-**OPTIONAL_HEADER** ישונו בהתאם לערך בשדה **Magic**. אם הערך בשדה **Magic** הוא **0x10B** מדובר בקובץ מסוג 32 ביט (PE32) וה-**OPTIONAL_HEADER** יהיה בגרסאת 32 ביט. אם הערך הוא **0x20B** מדובר בקובץ מסוג 64 ביט (PE32+) וה-**OPTIONAL_HEADER** יהיה בגרסאת 64 ביט.

AddressOfEntryPoint מציין את ה-RVA (Relative Virtual Address, או RVA, מצביע למקום בזיכרון) למקום שבו ה-Loader יתחיל להריץ את הקוד ברגע שהוא יסיים להעלות את הקובץ לזיכרון. לכן, זהו המקום הרישמי בזיכרון שבו הקוד מתחיל (אל תניחו שהוא מצביע להתחלה של פונקציית ()main).

SizeOfImage הוא הכמות הזיכרון שחייבת להיות שמורה מראש על-מנת להעלות את הקובץ לזיכרון (בעצם, הוא הגודל הכולל של הקובץ לאחר שהוא עבר תהליך של מיפוי לזיכרון). ה-Loader של מערכת ההפעלה מסתכל על הערך בשדה זה, מקצה את אותה כמות של זיכרון ולאחר מכן ממפה את חלקי הקובץ לתוך מרחב זיכרון זה. הערך שלו הוא תוצאה של חישוב השדות אלו של האגף (Section, נדבר על זה בהמשך) האחרון:

SECTION_HEADER.Misc.VirtualSize + SECTION_HEADER.VirtualAddress, אתם תבינו איך עשיתי את חישוב זה מאוחר יותר, כשנגיע ל-Section Headers.

SectionAlignment אומר בעצם שאגפים צריכים להיות מיושרים בכפולות של ערך זה. לדוגמא, אם הערך של שדה זה הוא 0x1000, אז היינו מצפים לראות אגפים מתחילים בכתובות 0x1000, 0x2000, 0x5000 וכו'. נרצה לדעת את הערך בשדה **SectionAlignment** כאשר יהיה לנו קוד ו/או נתונים היושבים בדיסק ונרצה לדעת באיזה כפולות של כתובות הם ימופו לזיכרון. המטרה של שדה זה הוא להגדיר ל-Loader של מערכת ההפעלה שהוא צריך למפות את הנתונים שלו בכפולות של אותו ערך. לכן, **SectionAlignment** הוא בעצם דרך ליישור האגפים שמופו כבר אל תוך הקובץ, אל תוך דיסק.

FileAlignment אומר שנתונים יכתבו לתוך קובץ בחתיכות שגודלן לא קטן מערך זה. לדוגמא, אם הערך בשדה **FileAlignment** יהיה 0x200 ויש לנו אגף באורך של 10 בתים, אנחנו נצטרך "לרפד" את המרחב בין 0xA (10) עד 0x200 (512), כדי שגודל האגף לא יהיה קטן מערך זה. הערכים הנפוצים ביותר לשדה זה הם 0x200 (512, הגודל של סקטור בכונן קשיח) ו-0x80 (הגודל של סקטור בדיסק און-קי). לכן, **FileAlignment** הוא בעצם דרך ליישור נתונים בדיסק.

ImageBase מציין את הכתובת הוירטואלית הרצויה לתחילת הקובץ, כאשר הקובץ ימופה לזיכרון. בעצם, **ImageBase** הוא הדרך של הקובץ להגיד: "אני רוצה להיות ממוקם בכתובת זו בזיכרון כאשר אני יעבור תהליך של מיפוי לזיכרון".

DLLCharacteristics מציין חלק מהאופציות האבטחתיות החשובות כמו ASLR והגדרת אזורים בזיכרון בתור אזורים שאינם ניתנים לריצה (non-executable) עבור ה-Loader. אופציות אלו ישפיעו לא רק על קבצי DLL, אלא גם על קבצי exe. וכו' (אולי חלקכם חשבו אחרת בגלל שם השדה). חלק מהאופציות הן:

```
#define IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE     0x0040 // DLL can move.
#define IMAGE_DLLCHARACTERISTICS_FORCE_INTEGRITY 0x0080 // Code Integrity Image
#define IMAGE_DLLCHARACTERISTICS_NX_COMPAT        0x0100 //Image is NX compatible
#define IMAGE_DLLCHARACTERISTICS_NO_SEH           0x0400 // Image does not use
SEH.  No SE handler may reside in this image
```

IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE יוגדר כאשר הקובץ יקושר (Will be linked) עם האופציה /DYNAMICBASE (ב-IDE-ים כמו Virtual Studio, ניתן לבחור אופציות קישור (Linker options) שיגדירו את התנהגות הקובץ בזמן ריצה ו/או טעינה. כדי ללמוד איך לעשות זאת, לחצו כאן). אופציה זאת אומרת למערכת ההפעלה שקובץ זה תומך ב-ASLR (Address Space Layout Randomization).

אופציית הקישור /FIXED חייבת להיות מוגדרת כ-"NO" בשביל קבצי exe., אחרת הקובץ לא יקבל את המידע על המיקום החדש של הנתונים (Relocation Information, נדבר על זה בהמשך). במילים אחרות, אנחנו אומרים ל-Linker: "הקוד שלי תומך בכך שיזיזו אותו בזיכרון". כאשר ניצור קובץ exe. ולא נגדיר את אופציה זו, אנחנו אומרים למערכת ההפעלה: "אל תזיז את הקוד שלי בזיכרון". אם אני לא טועה, אם לא נגדיר את אופציה זו ומרחב זיכרון זה יהיה תפוס ע"י תוכנית אחרת, התוכנית שלנו תקרוס.

IMAGE_DLLCHARACTERISTICS_FORCE_INTEGRITY אומר בעצם לבדוק בזמן הטעינה (At load time) האם ה-Hash הדיגיטלי החתום (Digitally signed hash) שלנו תואם למקור. במילים אחרות, אל תטען את הקובץ אלא אם כן יש לו חתימה דיגיטלית (נדבר על זה בהמשך) מצורפת והיא תואמת לחתימה המקורית.

IMAGE_DLLCHARACTERISTICS_NX_COMPAT יוגדר כאשר הקובץ יקושר עם האופציה /NXCOMPAT. אופציה זאת אומרת ל-Loader שה-Image תומך ב-Data Execution Prevention (DEP) ושלאגפים שאינם ניתנים לריצה אמור להיות את הדגל (Flag) NX מוגדר בזיכרון. במילים אחרות, קוד זה תומך בהגדרת חלקים מסויימים של אזורי נתונים בתור אזורים שאינם ניתנים לריצה. לדוגמא, האזורים בזיכרון Stack, Heap ו-Data הם אזורים שאינם ניתנים לריצה (אם לא הבנתם את אחד מהמושגים, מומלץ בחום לבדוק אותו בגוגל ☺).

IMAGE_DLLCHARACTERISTICS_NO_SEH אומר שהקובץ זה לעולם לא ישתמש בטיפול החריגות המובנה (Structured exception handling) ולכן שום Handler ברירת מחדל של שגיאות צריך להיווצר (בגלל שבהיעדר אופציות נוספות, ה-SEH Handler הוא בעל חולשות פוטנציאליות להתקפה). לכן, אופציה

זו אומרת שאם התוכנית נתקלת בחריגה כלשהי (Exception, כמו Stack-Overflow), על מערכת ההפעלה להפסיק את ריצת התוכנית.

השדה האחרון של ה-**OPTIONAL_HEADER** נקרא:

DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES]

זהו בעצם מערך של 16 פויינטרים (רשמית 16, אבל רק ה-14/15 הראשונים באמת משומשים) לכל המבני נתונים האחרים אשר נדבר עליהם בהמשך (שדה זה מחזיק פויינטרים לכל השדות המוקפים ב**תכלת** בתמונת פורמט ה-PE).

אתם בטח שואלים את עצמכם (או שלא, אני לא יכול לדעת), למה דווקא 16 פויינטרים? התשובה היא מכיוון שזה הוגדר כך בספרייה winnt.h (#define IMAGE_NUMBEROF_DIRECTORY_ENTRIES 16).

סוג המשתנה של **DataDirectory[16]** הוא struct הנקרא **IMAGE_DATA_DIRECTORY**. ה-struct נראה כך:

```
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD   VirtualAddress;
    DWORD   Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

VirtualAddress הוא RVA למבנה נתונים אחר. מבנה נתונים זה הוא בגודל **Size**.

אגפים (Sections)

אחרי שסיימנו לדבר על ה-**OPTIONAL_HEADER**, בואו נדבר על אגפים (ל-Section אין ממש תרגום בעברית, אז נקרא ל-Section אגף במהלך המאמר).

אגפים הם קבוצה של חלקי קוד או נתונים אשר יש להם את אותה מטרה או אמורות להיות להם את אותן הרשאות בזיכרון. מטרת האגפים היא לסדר חתיכות של נתונים כדי להגיד למערכת ההפעלה שיש לאותן חתיכות של נתונים את אותן ההרשאות. לדוגמא, אם יש לנו משתנה גלובלי בקוד שלנו, יכול להיות שהוא הוגדר בתוך אגף עם הרשאות קריאה וכתיבה, או אם יש לנו משתנה מסוג מחרוזת, יכול להיות שהוא הוגדר בתוך אגף עם הרשאות קריאה בלבד וכו'.

שמות האגפים הנפוצים ביותר הם:

- **.text** - המיקום שבו הקוד האמיתי נמצא, הקוד אשר אמור אף פעם לא לזלוג מהזיכרון אל הדיסק, אפילו אם נגמר לנו הזיכרון.
- **.data** - נתונים עם הרשאות של קריאה וכתיבה (משתנים מאותחלים גלובליים וסטאטיים).
- **.rdata** - נתונים עם הרשאות קריאה בלבד (מחרוזות).

- **.bss** - הראשי תיבות של BSS הן Block Started by Symbol או Block Storage Segment או Block Storage Start, תלוי את מי שואלים. האגף הזה מכיל את כל המשתנים הגלובליים והסטאטיים אשר מאותחלים ל-0 או שאין להם אתחול מפורש בקוד המקור. הגודל של אגף זה יהיה 0 בדיסק (כדי לשמור מקום בדיסק), אבל גודלו לא יהיה 0 בזיכרון (מכיוון שאנחנו עדיין צריכים להשתמש במשתנים אלו). זאת הסיבה שגודל הקובץ הוא קטן יותר מגודל הזיכרון שהוקצב לקובץ. מכיוון שאנחנו רוצים שמערכת ההפעלה תקציב מקום בזיכרון, כי יהיו משתנים גלובליים או סטאטיים בסוף, אבל אנחנו לא צריכים שאיזה שהוא ערך מיוחד יהיה מאותחל לאותם משתנים. לכן, גודל הקובץ בזיכרון יהיה גדול יותר מגודל הקובץ בדיסק. בפועל, זה נראה כי אגף ה-bss. מתמזג לתוך אגף ה-data. באמצעות ה-Linker.
- **.idata** - מכיל את טבלת הכתובות המיובאות (IAT, Import Address Table, נדבר על זה בהמשך). לדוגמא, אגף זה יכיל את הרשימה של כל הקבצים שאנחנו רוצים לייבא נתונים מהם (למשל פונקציות). בפועל, זה נראה כי אגף זה מתמזג לתוך אגף ה-text. או לתוך אגף ה-rdata...
- **.edata** - באגף זה יהיו לדוגמא כל הפונקציות אשר מפתח הקוד רוצה לייצא כדי שמפתחים אחרים יוכלו להשתמש באותן פונקציות, בתוך הקוד שלו. במילים אחרות, מטרת אגף זה הוא לייצא נתונים.
- **PAGE*** - קוד/נתונים אשר מותר להוציא מהזיכרון לדיסק אם אנחנו קצרים בזיכרון (נראה כי אגף זה נמצא בשימוש בעיקר אצל דרייברים של ליבת מערכת ההפעלה (Kernel drivers)).

.reloc - באגף זה יהיה מידע על המיקומים החדשים של הנתונים (Relocation Information) לטובת שינוי של כתובת המוטבעות בקוד אשר מניחות שהקוד הועלה ב-**ImageBase** הרצוי. המטרה של אגף זה היא לייצר Image לקובץ ריצה אשר יכול להיות ממוקם בצורה רנדומלית בזמן העלאה (At load time) בעזרת שימוש ב-ASLR. לדוגמא, ה-**ImageBase** הרצוי הוא 0x1000 אך הוא השתנה ל-0x2000 בגלל ה-ASLR. בקוד שלנו יש כתובות למשתנים ונתונים מסויימים אשר מבחינתם ה-**ImageBase** הוא 0x1000 ולכן הכתובות הן חושבות שתחילת התוכנית תהיה בכתובת 0x1000 בזיכרון. מכיוון שתחילת התוכנית תהיה בכתובת 0x2000 צריך להוסיף לכל הכתובות שבאגף זה 0x1000 (מכיוון ש: 0x2000-0x1000=0x1000) על מנת שהכתובות יהיו נכונות. לדוגמא, אם לאחת מהפונקציות באגף זה תהיה את הכתובת 0x1300 אז הכתובת תשתנה ל-0x2300 כדי שהכתובת תהיה רלוונטית ל-**ImageBase** החדש.

- **.rsrc** - משאבים, כמו אייקונים ועוד קבצים כלשהם. לאגף זה יש מבנה נתונים שמארגן את המשאבים כמעט כמו מערכת קבצים (File System).
- **.pdata** - חלק מהתוכנות משתמשות במבנה נתונים מסוג PDATA על מנת לסייע ב-Stack trace בזמן ריצה. מבנה נתונים זה עוזר בניפוי שגיאות (Debugging) ובעיבוד חריגות (Exception processing). אגף זה מכיל נתונים על ניהול חריגות (Exception handling) בגרסאת 64 ביט.

לכל אגף יש את ה-**SECTION_HEADER** שלו. לכן, ישר אחרי ה-**OPTIONAL_HEADER**, אנחנו נמצא את ה-**SECTION_HEADER** (אפילו ביט אחד בין שני אלה ומערכת ההפעלה תתבלבל). **SECTION_HEADER** נראה כך (מספר 5 בתמונת פורמט ה-PE):

```
typedef struct _IMAGE_SECTION_HEADER {
0x00  BYTE  Name[IMAGE_SIZEOF_SHORT_NAME];
      union {
0x08          DWORD PhysicalAddress;
0x08          DWORD VirtualSize;
      } Misc;
0x0c  DWORD VirtualAddress;
0x10  DWORD SizeOfRawData;
0x14  DWORD PointerToRawData;
0x18  DWORD PointerToRelocations;
0x1c  DWORD PointerToLinenumbers;
0x20  WORD  NumberOfRelocations;
0x22  WORD  NumberOfLinenumbers;
0x24  DWORD Characteristics;
};
```

```
#define IMAGE_SIZEOF_SHORT_NAME    8

typedef struct _IMAGE_SECTION_HEADER {
    BYTE    Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
            DWORD   PhysicalAddress;
            DWORD   VirtualSize;
    } Misc;
    DWORD   VirtualAddress;
    DWORD   SizeOfRawData;
    DWORD   PointerToRawData;
    DWORD   PointerToRelocations;
    DWORD   PointerToLinenumbers;
    WORD    NumberOfRelocations;
    WORD    NumberOfLinenumbers;
    DWORD   Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

Name[8] הוא מערך של תווי ASCII. מערך זה לאו דווקא יסתיים ב-null. לכן, אם אנחנו ננסה לנתח קובץ PE בעצמנו, אנחנו נצטרך להיות מודעים לכך. שדה זה הוא בשביל בני אדם כמונו (ובפוטנציאל גם ל-Linker). זאת אומרת שאלו רק 8 בתים שהוקצבו לנו ואנחנו יכולים להכניס למערך זה אילו תווים שאנחנו רוצים.

VirtualAddress הוא ה-RVA של האגף ביחס ל-**OptionalHeader.ImageBase**. במילים אחרות, **VirtualAddress** הוא המקום שבו אגף זה ימופה בזיכרון. לכן, אגף זה יתחיל בכתובת שנמצאת בערך של השדה **VirtualAddress** בזיכרון.

Misc.VirtualSize הוא הגודל של אגף זה בזיכרון. מכאן נובע ש-**VirtualAddress** היא הכתובת של תחילת האגף בזיכרון והאגף יהיה בגודל **Misc.VirtualSize** בתים.

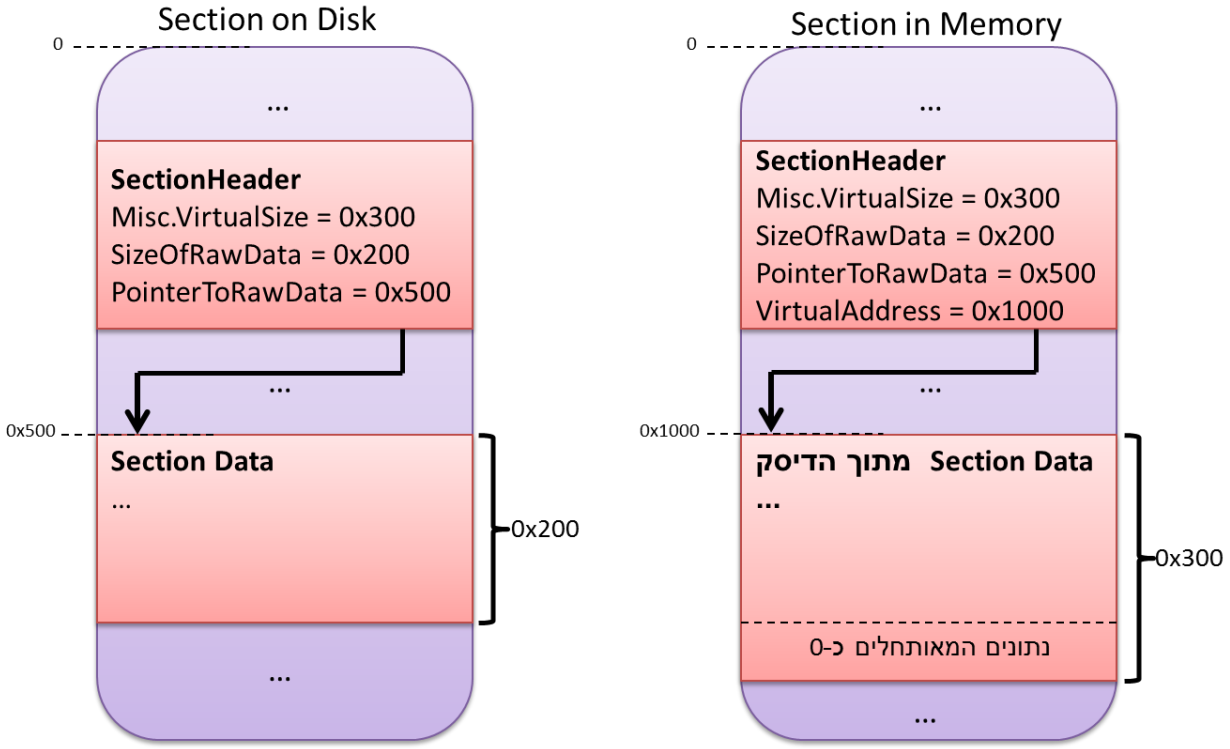
PointerToRawData הוא ה-Offset היחסי (Relative offset) מתחילת הקובץ אשר אומר לנו איפה נתוני האגף יהיו מאוחסנים. במילים אחרות, **PointerToRawData** הוא המיקום של אגף זה בדיסק. לכן, אגף זה יתחיל בכתובת שנמצאת בערך של השדה **PointerToRawData** בקובץ.

SizeOfRawData הוא הגודל של אגף זה בקובץ. מכאן נובע ש-**PointerToRawData** הוא הכתובת של תחילת האגף בקובץ והאגף יהיה בגודל **SizeOfRawData** בתים.

יש קשר מעניין בין **Misc.VirtualSize** לבין **SizeOfRawData**. לפעמים אחד מהם גדול יותר ולפעמים ההפך. למה ש-**Misc.VirtualSize** יהיה גדול יותר מ-**SizeOfRawData**? זה מעיד על כך שהאגף הקצה יותר מקום בזיכרון מאשר כמות הנתונים הנכתבו לדיסק. כדי להמחיש את דוגמא זאת, תחשבו על אגף ה-bss. לרגע. אגף זה צורך מקום בזיכרון בשביל משתנים. משתנים אלו לא מאותחלים, זאת הסיבה שהם לא חייבים לצרוך מקום בדיסק, אבל חייבים לצרוך מקום בזיכרון לטובת שימוש עתידי בהם. כתוצאה מכך,

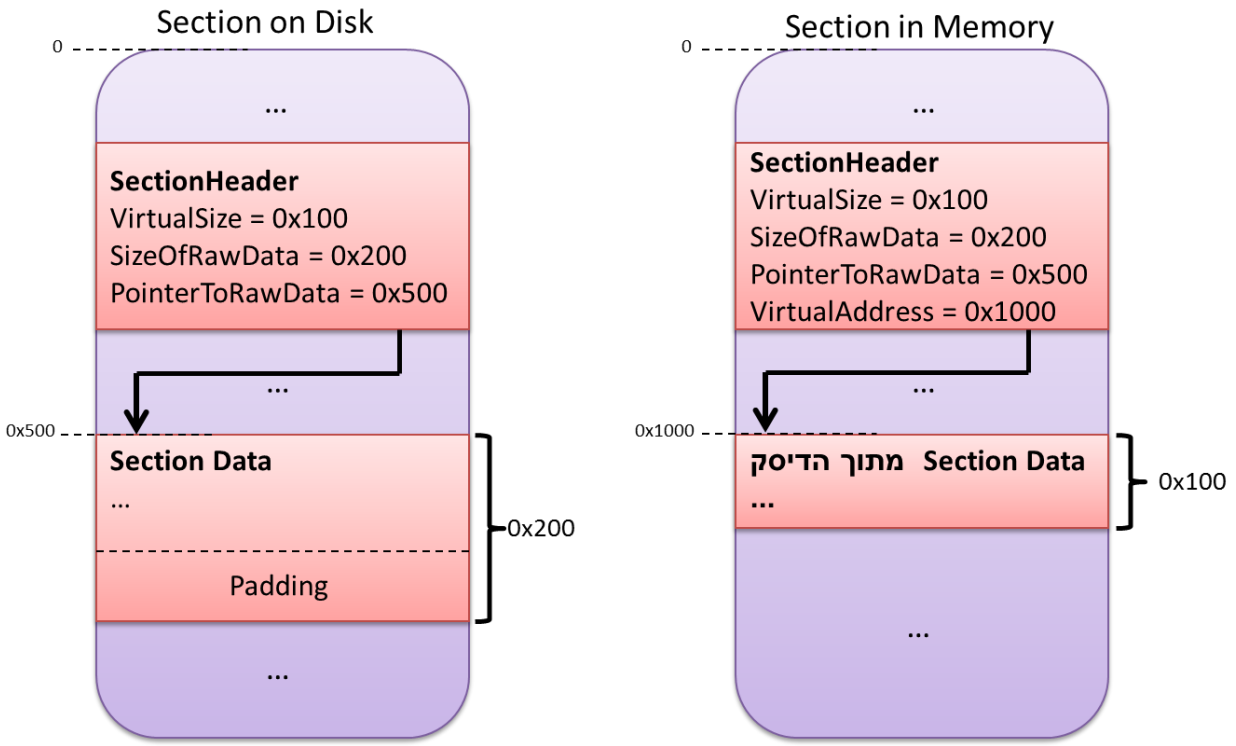
ה-Loader יכול פשוט לתת חלק של זיכרון לטובת אחסון משתנים אלו, באמצעות הקצאת כמות זיכרון בגודל **Misc.VirtualSize**. לכן, גודל הקובץ יהיה קטן יותר.

דוגמא למקרה שבו **Misc.VirtualSize** **גדול יותר** מ-**SizeOfRawData**:



כפי שניתן לראות מתמונה זאת, בדיסק הכתובת ההתחלתית של נתוני האגף (Section Data) היא 0x500 (כפי שכתוב ב-**PointerToRawData**) וגודלם הוא 0x200 (כפי שכתוב ב-**SizeOfRawData**). בזיכרון, הכתובת ההתחלתית של נתוני האגף היא 0x1000 (כפי שכתוב ב-**VirtualAddress**) וגודלם הוא 0x300 (כפי שכתוב ב-**Misc.VirtualSize**). לכן, לנתוני האגף יהיו 0x200 בתים אשר יועברו מהדיסק אל הזיכרון, ושאר 0x100 הבתים (0x300-0x200=0x100) יהיו נתונים המאותחלים כ-0 וגם הם יכתבו לזיכרון.

דוגמא למקרה שבו **Misc.VirtualSize** **קטן יותר** מ-**SizeOfRawData**:



דוגמא זאת נגרמה כתוצאה מהשדה **OPTIONAL_HEADER.FileAlignment**. לפי הדוגמא, הערך של **FileAlignment** הוא 0x200 ויש לנו כמות נתונים בגודל של 0x100 בתים. לכן, ה-Linker יצטרך לכתוב לקובץ 0x100 בתים של נתונים ולאחר מכן להוסיף עוד 0x100 בתים של "ריפוד" (Padding). כאשר **Misc.VirtualSize** קטן יותר מ-**SizeOfRawData**, ה-Loader מנסה להגיד: "אוקיי, אני רואה שבפועל אני צריך להקצות 0x100 בתים לזיכרון ולקרוא 0x100 בתים של נתונים מתוך הדיסק".

Characteristics נותן לנו מידע על האגף, לדוגמא:

```
#define IMAGE_SCN_CNT_CODE                0x00000020 // Section contains code.
#define IMAGE_SCN_CNT_INITIALIZED_DATA    0x00000040 // Section contains
                                                        initialized data.
#define IMAGE_SCN_CNT_UNINITIALIZED_DATA 0x00000080 // Section contains
                                                        uninitialized data.
#define IMAGE_SCN_MEM_DISCARDABLE         0x02000000 // Do not cache this section
#define IMAGE_SCN_MEM_NOT_CACHED          0x04000000 // Section can be discarded.
#define IMAGE_SCN_MEM_NOT_PAGED           0x08000000 // Section is not pageable.
#define IMAGE_SCN_MEM_SHARED              0x10000000 // Section is shareable.
#define IMAGE_SCN_MEM_EXECUTE             0x20000000 // Section is executable.
#define IMAGE_SCN_MEM_READ                0x40000000 // Section is readable.
#define IMAGE_SCN_MEM_WRITE               0x80000000 // Section is writeable.
```

אגב, אם תהיתם מה לגבי שאר השדות של ה-SECTION_HEADER (**PointerToRelocations**, **NumberOfRelocations**, **PointerToLinenumbers** ו-**NumberOfLinenumbers**), כיום לא נעשה בהם שימוש.

ייבוא מה-PE (PE Imports)

לפני שנדבר על ייבוא מה-PE, אנחנו נצטרך לדון על קישור סטאטי ודינאמי (Static Linking VS Dynamic Linking) וההבדל בינהם. כשאנחנו משתמשים ב**קישור סטאטי**, אנחנו כוללים עותק של כל הפונקציות עזר שאנו משתמשים בהם בתוך הקובץ שיצרנו ויוצרים קובץ בינארי גדול ועצמאי (קובץ exe. לדוגמא) אשר לא תלוי בקבצים אחרים על-מנת לרוץ. כשאנחנו משתמשים ב**קישור דינאמי**, אנחנו מציבים פויינטרים לפונקציות בתוך ספריות הנמצאות מחוץ לקובץ, בזמן ריצה. זאת אומרת שאנחנו ניצור קובץ בינארי קטן יותר אשר תלוי בקבצים אחרים (אשר כוללים חלק מהפונקציות של הקובץ שלנו) כדי לרוץ.

קובץ ריצה אשר השתמשו בקישור סטאטי בשביל ליצור אותו הוא "מנופח" יותר לעומת קובץ ריצה אשר השתמשו בקישור דינאמי בשביל ליצור אותו. מצד אחד, הוא עצמאי ולא תלוי באף קובץ. מצד שני, Patch-ים או תיקונים לסיפריות לא יחולו על קבצים שהשתמשו בקישור סטאטי עד שיקשרו אותם מחדש (Re-link) וזה אומר שיכול להיות באותם קבצים חולשות בקוד הרבה לאחר שהוציא Patch לחולשות אלו.

דבר זה שקוף למתכנת, אבל איזה אסמבלי הקומפיילר מייצר כאשר אנחנו קוראים לפונקציות מיובאות מספריות אחרות, כמו printf()? נשתמש בקוד HelloWorld.c (קוד שכל מה שהוא עושה זה להדפיס "Hello World") כדי להבין איך הפונקציה printf("Hello World!\n") מיובאת:

```
004113BE 8B F4              mov         esi,esp
004113C0 68 3C 57 41 00     push        41573Ch
004113C5 FF 15 BC 82 41 00  call        dword ptr ds:[004182BCh]
```

כפי שאנחנו רואים באסמבלי, הכתובת **המודגשת** מצביעה על טבלת הכתובות המיובאות (Import Address Table, או IAT) והקוד קורא לפונקציה printf() מתוך ה-IAT ולאחר מכן מריץ אותה.

זוכרים את השדה **DataDirectory[16]** שנמצא בתוך ה-**OPTIONAL_HEADER**? אם לא, כדאי לכם, מכיוון שאני הולך להזכיר את שדה זה הרבה בהמשך המאמר. אז הערך במקום (אינדקס) 1 במערך, ב-**DataDirectory[1]**, יש מבנה נתונים הנקרא DIRECTORY_ENTRY_IMPORT. הוא נראה כך (מספר 6 בתמונת פורמט ה-PE):

```
IMAGE_DIRECTORY_ENTRY_IMPORT

struct _IMAGE_DATA_DIRECTORY {
0x00   DWORD VirtualAddress;
0x04   DWORD Size;
};
```

בנוסף לכך, תזכרו שכל הערכים במערך זה זהים, בכולם יהיה כתובת וירטואלית (RVA) וגודל (ע"פ ה-Struct בשם **IMAGE_DATA_DIRECTORY** שראינו מספר עמודים קודם לכאן).

השדה הראשון של DIRECTORY_ENTRY_IMPORT הוא RVA למבנה הנתונים אשר נמצא בו את נתוני הייבוא (Import Information) והשדה השני הוא הגודל של אותו מבנה נתונים. במקרה הזה, ה-RVA מצביע למערך של כמה מבני נתונים מאותו סוג, הנקראים **IMPORT_DESCRIPTOR** (מספר 7 בתמונת פורמט ה-PE):

```
struct _IMAGE_IMPORT_DESCRIPTOR {
0x00  union {
            /* 0 for terminating null import descriptor  */
0x00        DWORD      Characteristics;
            /* RVA to original unbound IAT */
0x00        PIMAGE_THUNK_DATA OriginalFirstThunk;
      } u;
0x04  DWORD      TimeDateStamp;     /* 0 if not bound,
                                     * -1 if bound, and real date\time stamp
                                     *   in IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT
                                     * (new BIND)
                                     * otherwise date/time stamp of DLL bound to
                                     * (Old BIND)
                                     */
0x08  DWORD      ForwarderChain;    /* -1 if no forwarders */
0x0c  DWORD      Name;
      /* RVA to IAT (if bound this IAT has actual addresses) */
0x10  PIMAGE_THUNK_DATA FirstThunk;
};
```

אני חושב שהם התכוונו ל"INT"

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    union {
        DWORD   Characteristics;            // 0 for terminating null import descriptor
        DWORD   OriginalFirstThunk;         // RVA to original unbound IAT (PIMAGE_THUNK_DATA)
    };
    DWORD   TimeDateStamp;                  // 0 if not bound,
                                            // -1 if bound, and real date\time stamp
                                            //    in IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT (new
    BIND)
                                            // O.W. date/time stamp of DLL bound to (Old BIND)

    DWORD   ForwarderChain;                 // -1 if no forwarders
    DWORD   Name;
    DWORD   FirstThunk;                     // RVA to IAT (if bound this IAT has actual
    addresses)
} IMAGE_IMPORT_DESCRIPTOR;
```

יהיה לנו IMPORT_DESCRIPTOR אחד עבור כל קובץ שנייבא ממנו. אנחנו נקרא למערך זה של ה-IMPORT_DESCRIPTOR-ים בשם **Import Descriptor Table**, או **Import Directory**. מערך זה יגמר ב-null, זאת אומרת שה-IMPORT_DESCRIPTOR האחרון יהיה באותו גודל כמו השאר, אבל יכיל רק אפסים ולכן זהו מערך של נתונים הנגמר ב-null (A null-terminated array of data structures).

OriginalFirstThunk הוא RVA המצביע לטבלת השמות המיובאות (Import Name Table, או INT). INT הוא מערך והערכים שלו מצביעים לשמות/פונקציות שאנחנו נייבא מהקובץ. סוג הערכים במערך הוא Struct בשם IMAGE_THUNK_DATA (נדבר עליו אחר כך). קוראים לשדה **OriginalFirstThunk** כך מכיוון שה-INT הוא מערך של Struct-ים מסוג IMAGE_THUNK_DATA. לכן, שדה זה של ה-IMPORT_DESCRIPTORמצביע לערך הראשון של ה-INT.

FirstThunk הוא כמו **OriginalFirstThunk**, חוץ מהעובדה שהוא לא RVA שמצביע ל-INT, הוא RVA שמצביע לטבלת הכתובות המיובאות (Import Address Table, או IAT). IAT הוא מערך והערכים שלו מצביעים לכתובות של הפונקציות שאנחנו נייבא מהקובץ. כתובות אלו תואמות לשמות/פונקציות של ה-INT. ה-IAT הוא גם מערך של מבני נתונים מסוג IMAGE_THUNK_DATA.

Name הוא רק RVA המצביע לשם של הקובץ שאנחנו מייבאים ממנו דברים. לדוגמא, hal.dll, ntdll.dll וכו'.

OriginalFirstThunk ו-**FirstThunk** מצביעים למערך של Struct-ים בשם **IMAGE_THUNK_DATA**. ה-Struct נראה כך (מספר 8 בתמונת פורמט ה-PE):

```
typedef struct _IMAGE_THUNK_DATA {
        union {
0x00            LPBYTE    ForwarderString;
0x00            PDWORD    Function;
0x00            DWORD     Ordinal;
0x00            PIMAGE_IMPORT_BY_NAME   AddressOfData;
        } u1;
} IMAGE_THUNK_DATA,*PIMAGE_THUNK_DATA;
```

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

מכיוון ש-IMAGE_THUNK_DATA מכיל Union, הוא יכול להיות רק את אחד מהשדות **ForwarderString**, **Function**, **Ordinal** או **AddressOfData**.

לנו חשובים רק 2 מקרים. במקרה הראשון זה **Function**, שזה בעצם פויינטר אל DWORD. בפועל, זאת הכתובת של הפונקציה שייבאנו. במקרה השני זה **AddressOfData**. אם IMAGE_THUNK_DATA הוא ה-IAT, **AddressOfData** יהיה הכתובת לפונקציה שייבאנו. אם IMAGE_THUNK_DATA הוא ה-INT, זה יהיה פויינטר למבנה נתונים אחר שבו יהיה מספר ואת שם הפונקציה שייבאנו (בגלל מבנה נתונים בשם IMAGE_IMPORT_BY_NAME, עליו נדבר בהמשך). ה-IAT וה-INT הם מבני נתונים מסוג IMAGE_THUNK_DATA32 והם מפורשים בתור מצביעים למבני נתונים בשם IMAGE_IMPORT_BY_NAME. בסופו של דבר, הם **AddressOfData**.

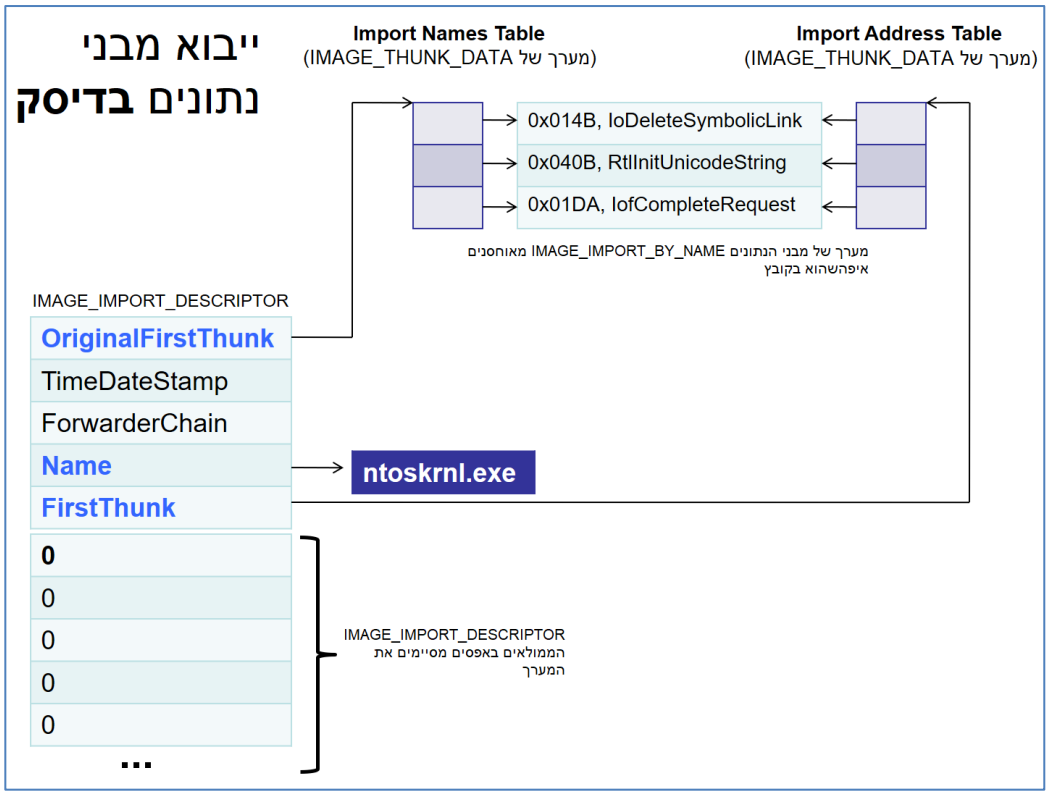
בפועל, **AddressOfData** הוא RVA למבנה הנתונים **IMAGE_IMPORT_BY_NAME**. מבנה הנתונים IMAGE_IMPORT_BY_NAME נראה כך (מספר 9 בתמונת פורמט ה-PE):

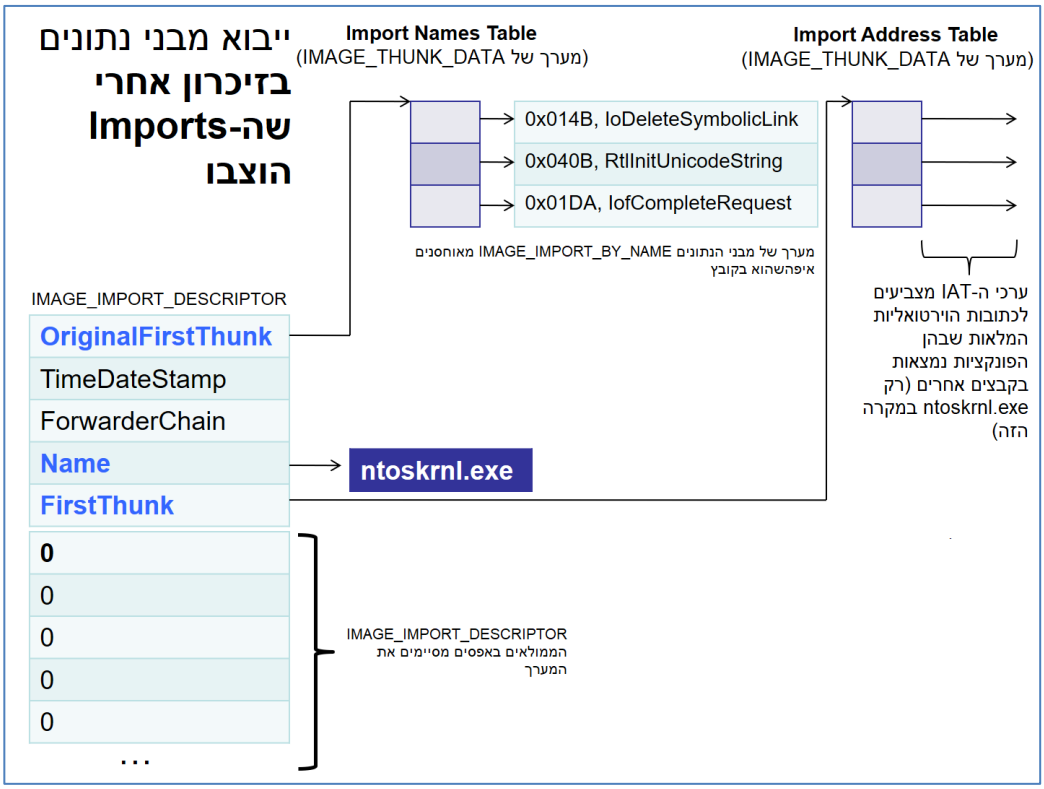
```
typedef struct _IMAGE_IMPORT_BY_NAME {
0x00  WORD        Hint;
0x02  BYTE Name[1];
} IMAGE_IMPORT_BY_NAME,*PIMAGE_IMPORT_BY_NAME;
```

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

Hint מציין מספר אפשרי לפונקצייה מיובאת. נדבר על זה בהמשך כאשר נדבר על ייצוא (Exports), אבל בקצרה זאת דרך לחפש פונקציה באמצעות אינדקס, במקום באמצעות שם. **Name** מצד שני, זאת דרך לחפש פונקצייה באמצעות שם. שדה זה לא באורך של בית אחד, הוא מחרוזת ASCII אשר נגמרת ב-null אשר עוקבת אחרי ה-**Hint**. הערך של **Name** יכול להיות גם null.

בדיסק, ה-INT וה-IAT יצביעו למערך של IMAGE_IMPORT_BY_NAME, אבל בזיכרון, IAT יצביע על הכתובות בפועל, כמו שהוא צריך:





בסוף, בתוך **DataDirectory[16]** יש "קיצור דרך" ל-IAT (לבסיס עצמו של ה-IAT) ב-**DataDirectory[12]**והוא נקרא **IMAGE_DIRECTORY_ENTRY_IAT**. הוא נראה כך (מספר 10 בתמונת פורמט ה-PE):

```
IMAGE_DIRECTORY_ENTRY_IAT

struct _IMAGE_DATA_DIRECTORY {
0x00   DWORD VirtualAddress;
0x04   DWORD Size;
};
```

ה-RVA (**VirtualAddress**) מצביע ישירות להתחלה של ה-IAT וגודל ה-IAT הוא **Size**.

עד כה דיברנו על ייבואים נורמליים. הנושא הבא שנדבר עליו הוא **Bound Imports**. Import Binding זאת פעולה שנועדה לטובת אופטימיצה של מהירות. הכתובות של הפונקציה יוצבו בזמן קישור (At link time) ויכנסו לתוך ה-IAT. תהליך זה יעשה אך ורק בגרסאות ספציפיות של הקובץ. אם הקובץ ישתנה אז כל ערכי ה-IAT יהיו לא נכונים, אבל זה רק אומר שאנחנו נצטרך להציב ערכים חדשים, אז זה לא כל כך נורא מאשר שלא היינו משתמשים בזה מלכלתחילה. במילים אחרות, Bound Imports הוא בעצם מילוי מראש של ה-IAT עם הכתובות הוירטואליות שהקומפיילר חושב שהן אמורות להיות (אחרי חישובים). אם הקומפיילר טעה, מערכת ההפעלה תתקן את זה, אבל אם הוא צדק אז יש לנו כאן אופטימיצה של מהירות, מכיוון שמערכת ההפעלה לא הייתה צריכה לחפש בטבלת הייצוא (Export Table, נדבר על זה אחר כך) אחר מחרוזת מסויימת (את שם הפונקציה).

בתוך ה-IMPORT_DESCRIPTOR, הערך בשדה **TimeDataStamp** הוא בדרך כלל "0", אבל עבור Bound Imports הערך שלו יהיה "1-". טבלת ה-Bound Imports תהיה נפרדת מטבלת הייבואים (Import-ים) הרגילים. אם הערך בשדה הוא "1-", ה-Loader של מערכת ההפעלה יבדוק אם הכתובות שהקומפיילר מילא אלו הכתובות המדוייקות.

הטבלה הבאה שנדבר עליה תקבע אם כתובות אלו מדוייקות או לא. הכתובות של הפונקציות אמורות להיות נכונות, אלא אם כן הקובץ עודכן לגרסא חדשה יותר. אם הוא עודכן, יכול להיות שהפונקציות שלו הוזזו לכתובות אחרות. לכן, אנחנו חייבים לשמור מידע על גרסאת הקובץ. אנחנו נשתמש בטבלה הבאה כדי לבדוק האם הקובץ נשאר באותה גרסא. אם הקובץ באותה גרסא אז הכתובות של הפונקציות יהיו נכונות ולא יהיה צורך במילוי מחדש של הכתובות בטבלה.

נחזור כעת אל **DataDirectory[16]**, ה-RVA של **DataDirectory[11]** הולך להצביע אל מערך של מבני נתונים בשם **IMAGE_BOUND_IMPORT_DESCRIPTOR**. **DataDirectory[11]** נראה כך (מספר 11 בתמונת פורמט ה-PE):

```
IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT

struct _IMAGE_DATA_DIRECTORY {
0x00   DWORD VirtualAddress;
0x04   DWORD Size;
};
```

המערך של IMAGE_BOUND_IMPORT_DESCRIPTOR-ים יסתיים ב-IMAGE_BOUND_IMPORT_DESCRIPTOR מלא באפסים (כמו שנעשה ב-IMAGE_IMPORT_DESCRIPTOR):

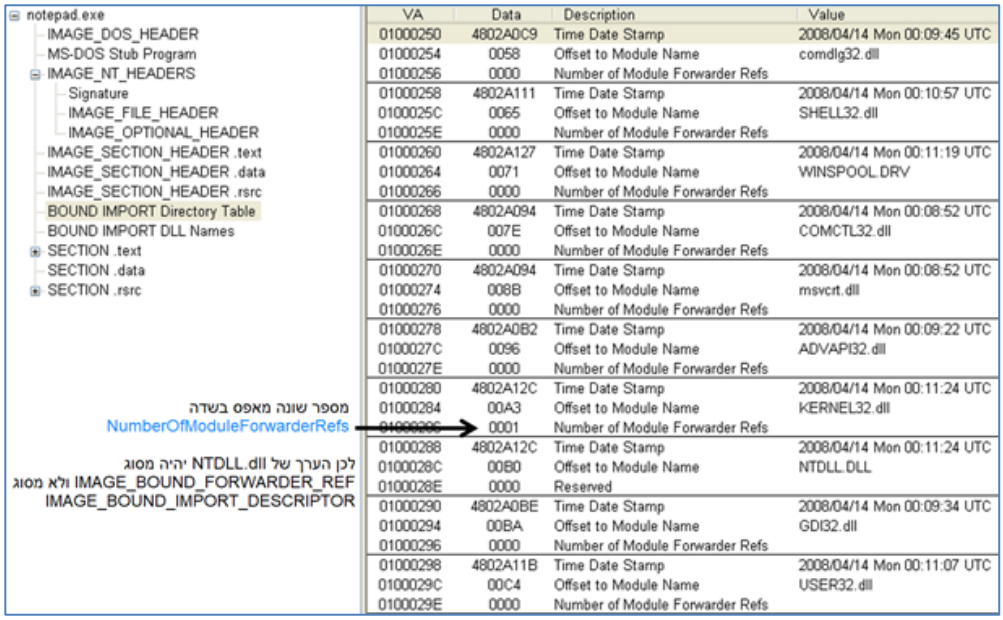
```
typedef struct _IMAGE_BOUND_IMPORT_DESCRIPTOR {
    DWORD    TimeDateStamp;
    WORD     OffsetModuleName;
    WORD     NumberOfModuleForwarderRefs;
// Array of zero or more IMAGE_BOUND_FORWARDER_REF follows
} IMAGE_BOUND_IMPORT_DESCRIPTOR, *PIMAGE_BOUND_IMPORT_DESCRIPTOR;

typedef struct _IMAGE_BOUND_FORWARDER_REF {
    DWORD    TimeDateStamp;
    WORD     OffsetModuleName;
    WORD     Reserved;
} IMAGE_BOUND_FORWARDER_REF, *PIMAGE_BOUND_FORWARDER_REF;
```

TimeDateStamp הוא הערך מנתוני הייצוא (Exports Information) של הקובץ שאנחנו מייבאים ממנו. בעצם, זאת גרסאת הקובץ שייבאנו והערך של השדה זה הוא הזמן שהקובץ עודכן כאשר הקובץ קומפל.

OffsetModuleName הוא ה-Offset של תחילת ה-IMAGE_BOUND_IMPORT_DESCRIPTOR הראשון. ערכו יהיה שמו של הקובץ המיובא, לדוגמא SHELL32.dll, KERNEL32.dll וכו'.

המערך של IMAGE_BOUND_IMPORT_DESCRIPTOR ב"פנקס רשימות" (Notepad.exe):



ASLR הופך את תהליך ה-Import Binding לחסר תועלת. מכיוון שאם ה-ASLR יעשה את העבודה שלו, הכתובות שיווצרו כתוצאה מתהליך זה יהיו לא נכונות רוב הזמן, כי התוכנית לא תיטען באותו בסיס ולכן כל הכתובות יזוזו. אז אנחנו נהיה חייבים להציב כתובות חדשות בזמן העלאה (At load time) בכל מקרה, וזאת הסיבה שהזמן שהשקענו כדי לבדוק את גרסאת הקובץ הוא חסר תועלת, אז פשוט לא נשמש ב-Import Binding כאשר אנחנו משתמשים ב-ASLR.

DLL-ים מעוכבים טעינה (Delay Loaded DLLs)

הנושא הבא שלנו הם DLL-ים מעוכבים טעינה. הכוונה היא שספריות לא יוטענו למרחב הזיכרון עד הפעם הראשונה שישתמשו בהם, לדוגמא כאשר קוד יקרא לפונקציה בתוך SHELL32.dll. זה דווקא יכול להיות דבר טוב לקוד, לדוגמא, כאשר אנחנו צריכים לייבא קובץ DLL גדול מאוד אשר ייקח הרבה מאוד זיכרון ואנחנו קוראים לו רק פעם אחת כאשר מופע (Event) ספציפי קורה. זה לא יעיל לייבא את כל ה-DLL עד לרגע שאנחנו נשתמש בו, מכיוון שהוא תופס מלא זיכרון. במילים אחרות, נייבא את ה-DLL רק כשנצטרך אותו. כאשר נשתמש ב-DLL-ים מעוכבים טעינה, נייצר בעצם עוד נתונים נפרדים מה-DLL-ים הנטענים בצורה רגילה (Normal DLL loading), כדי לתמוך בעיכוב הטעינה.

בחזרה ל-**DataDirectory[16]**, ה-RVA של **DataDirectory[13]** הולך להצביע אל מבנה נתונים אחר, אשר נקרא IMAGE_DELAY_IMPORT_DESCRIPTOR. **DataDirectory[13]** נראה כך (מספר 12 בתמונת פורמט ה-PE):

```
IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT

struct _IMAGE_DATA_DIRECTORY {
0x00   DWORD VirtualAddress;
0x04   DWORD Size;
};
```

IMAGE_DELAY_IMPORT_DESCRIPTOR נראה כך (מספר 13 בתמונת פורמט ה-PE):

```
struct _IMAGE_DELAY_IMPORT_DESCRIPTOR {
0x00   DWORD grAttrs;
0x04   DWORD szName;
0x08   DWORD phmod;
0x0c   DWORD pIAT;
0x10   DWORD pINT;
0x14   DWORD pBoundIAT;
0x18   DWORD pUnloadIAT;
0x1c   DWORD dwTimeStamp;
};
```

```
typedef struct ImgDelayDescr {
DWORD    grAttrs;       // attributes
RVA      rvaDLLName;    // RVA to dll name
RVA      rvaHmod;       // RVA of module handle
RVA      rvaIAT;        // RVA of the IAT
RVA      rvaINT;        // RVA of the INT
RVA      rvaBoundIAT;   // RVA of the optional bound IAT
RVA      rvaUnloadIAT;  // RVA of optional copy of original IAT
DWORD    dwTimeStamp;   // 0 if not bound,
                        // O.W. date/time stamp of DLL bound to (Old BIND)
} ImgDelayDescr, * PImgDelayDescr;
```

rvaDLLName הוא השם של ה-DLL שנייבא ממנו את הפונקציות, לדוגמא gdiplus.dll, SHELL32.dll וכו'.

rvaIAT מצביע אל IAT נפרד עבור פונקציות מעוכבות טעינה (Delay Load IAT) בלבד, זהו ה-IAT שמעניין אותנו באמת. ראשית לכל, ה-IAT עבור המעוכבי טעינה מחזיק כתובות וירטואליות של **Stub code**. Stub code הוא בעצם פויינטר לקוד בתוך הקובץ עצמו. אז בפעם הראשונה שנקרא לפונקציה מעוכבת טעינה,

היא קודם כל תקרא ל-Stub code. אם נחוץ, ה-Stub code טוען את המודל שמכיל את הפונקציה שאנחנו רוצים לקרוא לה. לאחר מכן, הוא מציב את הכתובת של הפונקציה בתוך מודל זה. הוא ממלא את הכתובות לתוך ה-IAT עבור המעוכבי טעינה ואז קורא לפונקציה הרצויה. בפעם השניה שהקוד יקרא לפונקציה, הקוד יעקוף את התהליך שדיברנו עליו עכשיו וילך ישר אל הפונקציה הרצויה. אכפת לנו מה-**rvaIAT**מכיוון שהוא מצביע ל-IAT הנפרד שבו כתובות מתמלאות בעת הצורך.

לדוגמא, בתוך mspaint.exe הקוד קורא לפונקציה בשם()DrawThemeBackground מהכתובת 0x103e6c4. כתובת זאת לא תוביל אותנו ישר לפונקציה, היא תוביל אותנו ל-IAT עבור המעוכבי טעינה. בפועל הכתובת 0x103e6c4 מכילה את כתובת בפני עצמה, את הכתובת 0x1035425. כתובת זאת היא בעצם Stub code. ה-Stub code טוען את ה-DLL שבו הפונקציה נמצאת, משיג את ה-VA (כמו RVA, רק שמוסיפים את הערך בשדה **ImageBase**. אפשר לרשום AVA, Absolute Virtual Address או VA) של הפונקציה (נגיד שה-VA הוא 5ad72bef), שם אותו במקום 0x1035425 ומריץ את הפונקציה. בפעם הבאה שהקוד יקרא לפונקציה, הוא ילך קודם ל-IAT עבור מעוכבי טעינה בכתובת 0x103e6c4, כתובת זאת תכיל את הכתובת 5ad72bef (במקום 0x1035425), וכפי שהבנו, כתובת זאת היא הכתובת האמיתית של הפונקציה, ויריץ אותה. מכאן אנו מסיקים שהקוד חיפש את הכתובת של הפונקציה פעם אחת, שמר אותה בתוך ה-IAT עבור מעוכבי טעינה כדי שמתי שהקוד יקרא לפונקציה שוב, הקוד ילך ישירות לפונקציה מתוך ה-IAT עבור המעוכבי טעינה.

rvaINT מצביע אל INT נפרד עבור פונקציות מעוכבות טעינה בלבד. אכפת לנו מה-**rvaINT** מכיוון שהוא מצביע ל-INT הנפרד שבו שמות/פונקציות מתמלאות בעת הצורך.

קיימת דרך אחרת לייבא דברים אשר קשורים ל-DLL-ים המעוכבים טעינה, אשר נקראת **ייבוא בזמן ריצה (Runtime Import).** נוזקות משתמשות בסוג זה של ייבוא הרבה. תהליך זה הוא ה"מאחורי הקלעים" של עיכובי טעינה. הוא משתמש בשתי פונקציות אלו של Windows: ()LoadLibrary ו-()GetProcAddress .

()LoadLibrary יכולה להקרא כדי לטעון DLL באופן דינאמי לתוך מרחב זיכרון של תהליך. אנחנו נותנים לפונקציה את השם של ה-DLL והיא מחזירה את כתובת הבסיס (Base Address) של המקום שבו ה-DLL נטען.

()GetProcAddress נותן לנו את הכתובת של הפונקציה שצויינה ע"י שם או ע"י מספר סידורי (Ordinal, נדבר עליו בהמשך). ניתן להשתמש בכתובת זאת בתור פויינטר לפונקציה.

זוכרים שדיברנו על DLL-ים מעוכבים טעינה, וה-Linker "איכשהו" טען את ה-DLL ומצא את הכתובת של הפונקציה? הוא בעצם השתמש ב-()LoadLibrary וב-()GetProcAddress.

בפונקציות אלו משתמשים מלא בנוזקות כדי שלא יהיה ניתן לדעת באילו פונקציות הנוזקה השתמשה בדרך הפשוטה, באמצעות הסתכלות על ה-INT, מכיוון שמפתח הנוזקה רוצה להסתיר את הפונקציות שבו הוא משתמש. לכן, הנוזקה תכיל את כל השמות של הספריות המיובאות ושל הפונקציות מעורבבות בנתונים של הקוד, ולאחר מכן הנוזקה תניח אותם במקום הנכון ותציב אותם בצורה דינאמית לפני שהיא קוראת לפונקציות המיובאות. במילים אחרות, הנוזקה רוצה להסתיר את שמות הפונקציות שלה, לכן היא תשתמש ב-2 פונקציות אלו כדי לגרום לשמות הפונקציה שלה להיות יותר קשות לגילוי מאשר פשוט להסתכל על ה-INT.

ייצוא (Exports)

ציינתי את טבלת הייצוא וייצוא בכללי מספר פעמים במהלך מאמר זה. בשביל שספרייה תהיה שימושית, קודים אחרים ירצו להשתמש בפונקציות שלה. לכן, פונקציות אלו יהיו חייבות להיות בעלות היכולת לייבא את עצמן לקודים אחרים.

אנחנו יכולים לחשוב על טבלת הייצוא (Export Table) בתור רשימה גדולה שממפה RVA לתוך מחרוזת מסויימת.

יש שתי אופציות לייצוא פונקציות ונתונים: ייצוא באמצעות שם (כאשר למתכנת יש אפילו את האופציה לקרוא לשם המיוצא בשם אחר מהשם האמיתי שלו) או ייצוא באמצעות מספר סידורי (Ordinal).

מספר סידורי הוא רק אינדקס ואם פונקציה מיוצאת באמצעות מספר סידורי, אפשר לייבא אותה רק באמצעות מספר סידורי. אפילו שלייצא באמצעות מספר סידורי חוסך מקום, עצם העובדה שלא קיימת מחרוזת נוספת בשביל שם כלשהו ולא יוקדש זמן לטובת חיפוש המחרוזת, גורמת לעוד יותר עבודה עבור המתכנת אשר רוצה לייבא את מה שמיוצא. אבל זאת גם דרך להפוך את ה-API ליותר פרטי (ללא תיעוד). לדוגמא, אני רוצה להשתמש בפונקצייה מספר 10 מהקובץ kernel32.dll בגלל שאני יודע מה עושה פונקצייה מספר 10 ואני רוצה לחסוך זמן ומקום. זה סוג של מנגנון של אופטימציית מהירות. החיסרון בייבוא באמצעות מספר סידורי הוא שאם המספר הסידורי משתנה, האפליקציה שלנו לא תעבוד כמתוכנן. לכן, למפתח אשר מייצא באמצעות מספר סידורי יש תמריץ לא לשנות אותו אלא אם כן הוא רוצה שתוכנות שישתמשו במספר סידורי זה לא יפעלו כראוי, לדוגמא כדי לכפות על API שלא אושר את הפסקת השימוש בו.

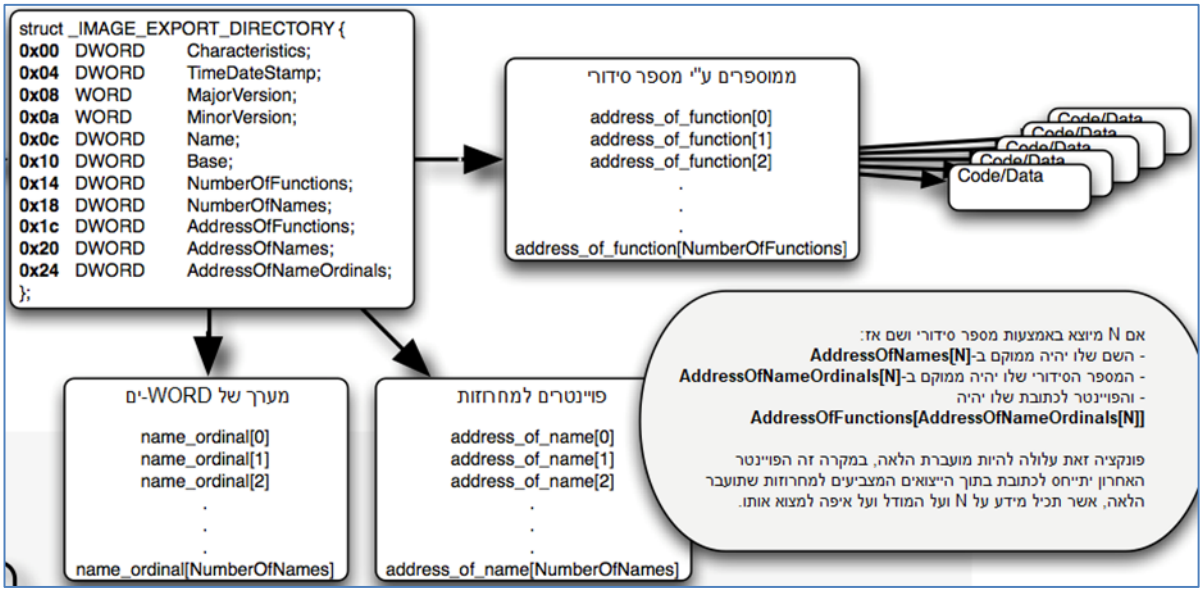
כדי להשיג את טבלת הכתובות המיוצאות (Export Address Table או EAT), אנחנו נפנה אל הערך בשדה **DataDirectory[0]**. ה-RVA של **DataDirectory[0]** מצביע אל ה-EAT. **DataDirectory[0]** נקרא IMAGE_DIRECTORY_ENTRY_POINT ונראה כך (מספר 14 בתמונת פורמט ה-PE):

```
IMAGE_DIRECTORY_ENTRY_EXPORT

struct _IMAGE_DATA_DIRECTORY {
0x00   DWORD VirtualAddress;
0x04   DWORD Size;
};
```

ה-RVA מצביע אל מבנה נתונים אשר נקרא **IMAGE_EXPORT_DIRECTORY** אשר מצביע אל 3 רשימות.

הוא נראה כך (מספר 15 בתמונת פורמט ה-PE):




```
typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD   Characteristics;
    DWORD   TimeDateStamp;
    WORD    MajorVersion;
    WORD    MinorVersion;
    DWORD   Name;
    DWORD   Base;
    DWORD   NumberOfFunctions;
    DWORD   NumberOfNames;
    DWORD   AddressOfFunctions;     // RVA from base of image
    DWORD   AddressOfNames;         // RVA from base of image
    DWORD   AddressOfNameOrdinals;  // RVA from base of image
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

אנו מדברים על מודל ספציפי אשר מייצא קבוצה של פונקציות. זאת הסיבה שיש לנו IMAGE_EXPORT_DIRECTORY אחד לכל קובץ.

השדה **TimeDateStamp** הרשום כאן הוא השדה שנבדק בפועל מול ה-Loader כאשר הוא מנסה לקבוע האם ה-Bound Imports אינם מעודכנים למשל. שדה זה יכול להיות שונה מהשדה **TimeDateStamp** שנמצא ב-**FILE_HEADER**. כנראה (לא הצלחתי לוודא את זה), ה-Linker מעדכן את שדה זה רק אם יש שינויים חשובים ל-RVA-ים או לפונקציות המיוצאות. בדרך זאת, "גרסאת" ה-**TimeDateStamp** יכולה להישאר מותאמת לאחור (Backward Compatible) ככל האפשר.

NumberOfFunctions יהיה שונה מ-**NumberOfNames** כאשר הקובץ ייצא חלק מהפונקציות באמצעות מספר סידורי. במילים אחרות, בגלל שייבוא באמצעות מספר סידורי מותר, יכול להיות לנו יותר פונקציות

מאשר שמות. לכן, לחלק מהפונקציות לא נוכל לקרוא באמצעות שם, אלא רק באמצעות מספר סידורי. אם נדע את מספר השמות, כאשר נחפש אחר ייבוא באמצעות שם, ה-Loader יוכל לעשות חיפוש בינארי.

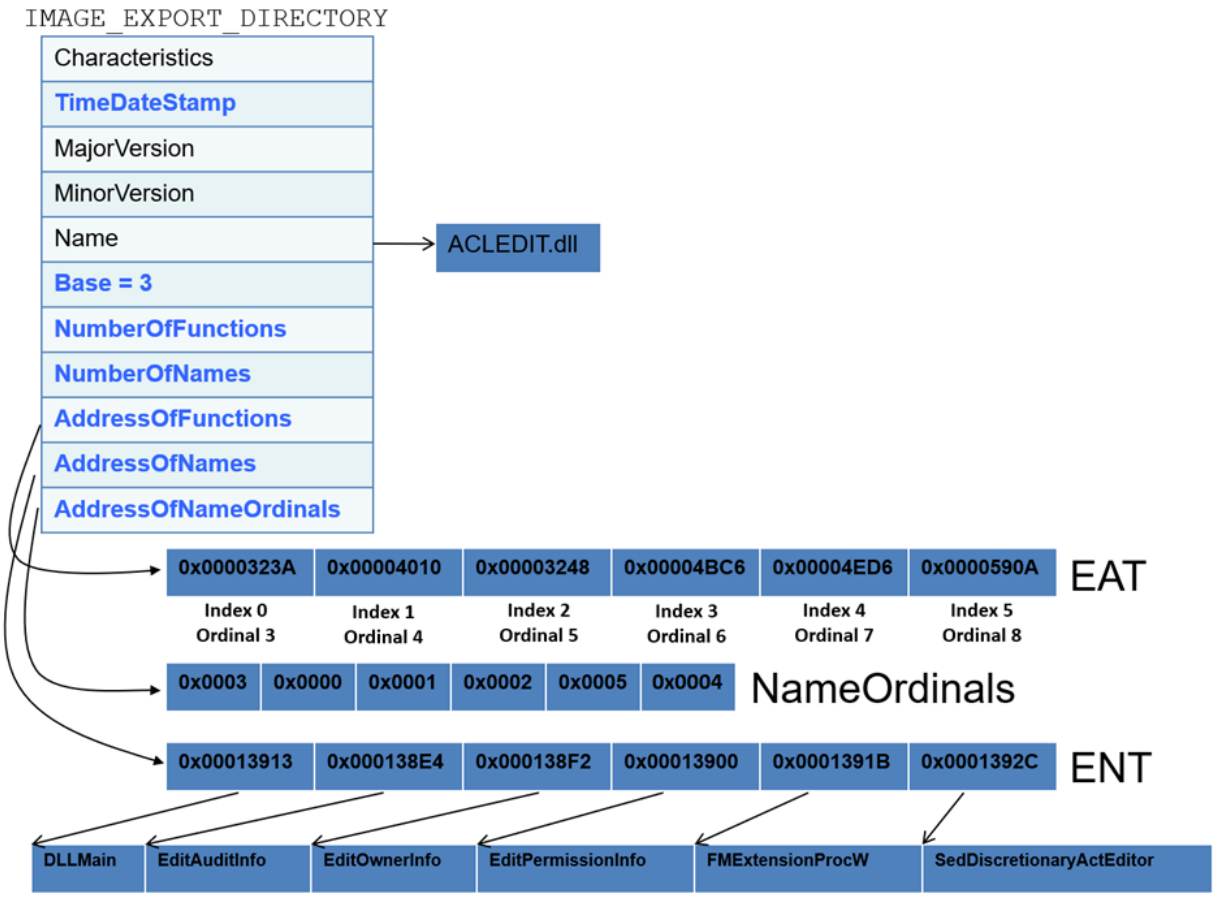
Base הוא המספר שנחסיר מהמספר הסידורי בשביל להגיע ל-Offset באינדקס 0 בתוך מערך של **AddressOfFunctions** (נדבר עליו אחר כך). הערך הברירת מחדל של המספר הסידורי הוא "1", לכן לרוב הוא יהיה "1". אולם שהמספר הסידורי יכול להתחיל ב-10 אם המתכנת בוחר לעשות כך. במילים אחרות, **Base** האינדקס ההתחלתי של המספרים הסידוריים.

AddressOfFunctions הוא RVA המצביע אל תחילת המערך המחזיק RVA-ים בגודל DWORD אשר מצביעים לתחילת הפונקציות המיוצאות. המערך שמצביעים אליו צריך להיות עם מספר של **NumberOfFunctions** ערכים. **AddressOfFunctions** יצביע אל ה-EAT. שדה זה הוא "מקביל" לשדה **FirstThunk** בתוך ה-IMPORT_DESCRIPTOR המצביע אל ה-IAT.

AddressOfNames הוא RVA המצביע אל תחילת המערך המחזיק RVA-ים בגודל DWORD אשר מצביעים למחרוזות המייצגות את שמות הפונקציות. המערך שמצביעים אליו צריך להיות בן מספר של **NumberOfNames** ערכים. **AddressOfFunctions** יצביע אל ה-ENT. שדה זה הוא "מקביל" לשדה **OriginalFirstThunk** בתוך ה-IMPORT_DESCRIPTOR המצביע אל ה-INT.

AddressOfNameOrdinals הוא RVA המצביע אל תחילת המערך המחזיק מספרים סידוריים בגודל של WORD (16 ביט). הערכים במערך זה מתחילים באינדקס 0 ולכן הם לא מושפעים מה-**Base**. מערך זה "מתרגם" מה-ENT ל-EAT. לדוגמא, אם אנחנו רוצים לייצא פונקציה ספציפית ואנחנו קוראים לה לפי שם והכתובת של פונקציה זאת ב-ENT היא באינדקס מספר 2, אז הערך באינדקס 2 בתוך טבלת המספרים הסידורים, יהיה האינדקס של הפונקציה ב-EAT. אנחנו צריכים את "מתרגם" זה בגלל שהכתובות ב-EATלא באותו סדר כמו הכתובות ב-ENT. לכן אנחנו נשתמש בטבלת המספרים הסידוריים כדי להשיג את האינדקס של כתובות הפונקציות מה-EAT.

תמונה זאת מדגימה את IMAGE_EXPORT_DIRECTORY:



קיים מקרה קצה אחד של ייצוא הנקרא **ייצוא העברתי (Forward Exports)**. זאת הסיבה שיש לנו Bound Imports מסובכים יותר. ייצוא העברתי הוא בעצם האופציה להעביר את הטיפול בפונקציה ממודל אחד לאחר. זה יכול לדוגמא להיות בשימוש אם הקוד אורגן מחדש כדי להזיז את הפונקציה למודל אחר כשאנחנו רוצים לשמור על תאימות לאחור.

כפי שראינו, בדרך כלל **AddressOfFunctions** מצביע אל מערך של RVA-ים אשר מצביעים אל קוד. אולם, אם RVA בתוך המערך של ה-RVA-ים מצביע אל אגף הייצוא, ה-RVA בפועל יצביע אל מחרוזת בפורמט DllToForwardTo.FunctionName. במילים אחרות, מחוץ לתחום של נתוני הייצוא, יהיה לנו RVA שבמקום להצביע אל פונקציה, הוא יצביע אל מחרוזת אשר אומרת: "פונקציה זאת הושתלה ב-DLL אחד (לדוגמא ACLEDIT.EditAuditInfo).

Stuxnet לדוגמא (אם אתם לא יודעים מה זה, Google it ☺) השתמשה ב-DLL זדוני. בתוך DLL זה היא לכדה או יישמה מחדש את חלק מהפונקציות מה-DLL המקורי ובשביל שאר הפונקציות, שמבחינתה לא מעניינות, היא השתמשה בייצוא העברתי ל-DLL המקורי כדי שהם יעבדו כרגיל. רק כדי להשלים את מה שלא הסברתי קודם, עכשיו שלמדנו על ייצוא העברתי של פונקציות, המטרה של **NumberOfModuleForwarderRefs** (בתוך IMAGE_BOUND_IMPORT_DESCRIPTOR) וגם של

IMAGE_BOUND_FORWARDER_REF הוא שכאשר ה-Linker ינסה לאמת שאף אחד מה- Bound Importsשונו, הוא יצטרך לוודא שאף אחד מהגירסאות (**TimeDateStamps**) של המודלים שייבאנו שונו. לכן, אם מודל קושר למודל אחר שמשתמש בייצוא העברתי למודלים אחרים, מודלים אלו חייבים להיבדק גם כן. לפרוטוקול, בערך של **NumberOfModuleForwarderRefs** יוצב ערך מספרי לא אפסי כשאנחנו נייבא מקובץ אשר משתמש בייצואים העברתיים במקום כלשהו.

ספריית ניפוי השגיאות (Debug Directory)

בחזרה אל **DataDirectory[16]**, ה-RVA של **DataDirectory[6]** הולך להצביע אל מבנה נתונים הנקרא IMAGE_DEBUG_DIRECTORY. **DataDirectory[6]** נקרא IMAGE_DIRECTORY_ENTRY_DEBUG ונראה כך (מספר 16 בתמונת פורמט ה-PE):

```
IMAGE_DIRECTORY_ENTRY_DEBUG

struct _IMAGE_DATA_DIRECTORY {
0x00   DWORD VirtualAddress;
0x04   DWORD Size;
};
```

IMAGE_DEBUG_DIRECTORY נראה כך (מספר 17 בתמונת פורמט ה-PE):

```
struct _IMAGE_DEBUG_DIRECTORY {
0x00   DWORD Characteristics;
0x04   DWORD TimeDateStamp;
0x08   WORD  MajorVersion;
0x0a   WORD  MinorVersion;
0x0c   DWORD Type;
0x10   DWORD SizeOfData;
0x14   DWORD AddressOfRawData;
0x18   DWORD PointerToRawData;
};
```

```
typedef struct _IMAGE_DEBUG_DIRECTORY {
    DWORD   Characteristics;
    DWORD   TimeDateStamp;
    WORD    MajorVersion;
    WORD    MinorVersion;
    DWORD   Type;
    DWORD   SizeOfData;
    DWORD   AddressOfRawData;
    DWORD   PointerToRawData;
} IMAGE_DEBUG_DIRECTORY, *PIMAGE_DEBUG_DIRECTORY;

#define IMAGE_DEBUG_TYPE_UNKNOWN          0
#define IMAGE_DEBUG_TYPE_COFF             1
#define IMAGE_DEBUG_TYPE_CODEVIEW         2
#define IMAGE_DEBUG_TYPE_FPO              3
#define IMAGE_DEBUG_TYPE_MISC             4
#define IMAGE_DEBUG_TYPE_EXCEPTION        5
#define IMAGE_DEBUG_TYPE_FIXUP            6
#define IMAGE_DEBUG_TYPE_OMAP_TO_SRC      7
#define IMAGE_DEBUG_TYPE_OMAP_FROM_SRC    8
#define IMAGE_DEBUG_TYPE_BORLAND          9
#define IMAGE_DEBUG_TYPE_RESERVED10       10
#define IMAGE_DEBUG_TYPE_CLSID            11
```

שדה **TimeDateStamp** זה הוא ה-**TimeDateStamp** השלישי שאכפת לנו עבור חקירת המטרות הזדוניות של נוזקות וכו'. מטרתו היא זהה למטרות של השניים האחרים. **TimeDateStamp** ישתנה כאשר נתוני ניפוי השגיאות (Debug Information) ישתנו. אני כמעט בטוח שהערך שלו יהיה זהה לערך של ה-**TimeDateStamp**ב-**FILE_HEADER**.

Type יציין מבנה מסויים אשר נדבר עליו בהמשך המאמר. יש 11 סוגים שונים של **Type**-ים (הם כלולים בתמונה שלמעלה, איפה שכל ה-#define-ים). במקרה שלנו, ערכו של השדה יהיה **IMAGE_DEBUG_TYPE_CODEVIEW**. הערך היחיד שאכפת לנו ממנו הוא **IMAGE_DEBUG_TYPE_CODEVIEW** מכיוון שזה הפורמט הנפוץ ביותר אשר מצביע אל מבנה הנתונים המחזיק את הנתיב (Path) אל קובץ ה-pdb. אשר מחזיק את נתוני הניפוי שגיאות.

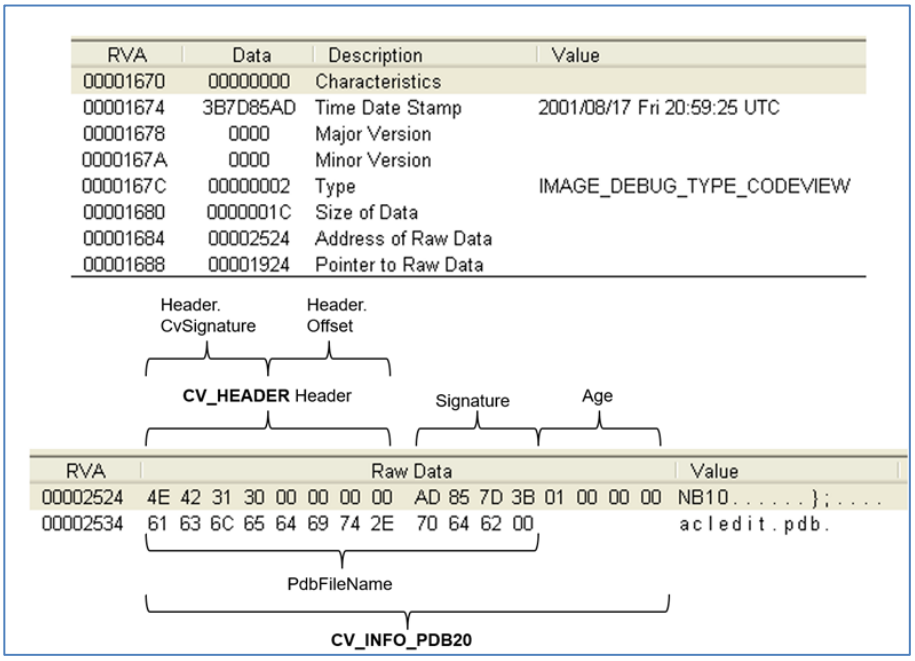
למטרות שלנו, ערכו של **Type** תמיד יהיה 2 מכיוון שהוא הוגדר ככה (**#define** "**IMAGE_DEBUG_TYPE_CODEVIEW 2**"). ומכיוון שבערך זה Microsoft משתמשת לטובת בניית נתוני הניפוי שגיאות שלה.

SizeOfData יציין את הגודל של המבנה ב-**Type**.

AddressOfRawData הוא RVA לנתוני הניפוי שגיאות. במילים אחרות, המיקום של נתוני ניפוי השגיאות, בזיכרון.

PointerToRawData הוא ה-Offset של הקובץ אל נתוני ניפוי השגיאות. במילים אחרות, המיקום של נתוני ניפוי השגיאות, בקובץ. **AddressOfRawData** ו-**PointerToRawData** יהיו באותו גודל בגלל שאין לנו דברים כמו "ריפוד" כאן וכו'.

לפרוטוקול, אנחנו יכולים להשיג את שם הקובץ מסוג pdb. באמצעות חיפוש כתובות מתוך השדה **PointerToRawData**, בתוך הקובץ. ברגע שמצאנו אותו, אנחנו נוכל לדעת אולי מי כתב את הקובץ (משתמשים בזה הרבה בניתוח נוזקות). לדוגמא, אם אנחנו מנתחים נוזקה והנתיב לקובץ ה-pdb. הוא "e:\gh0st\server\sys\i386\RESSDT.pdb", ולאחר מכן עוד נוזקה עם נתיב באותו שם או שם דומה, אנחנו נדע בוודאות שאת נוזקה זאת כתב אותו אדם.



שינויי כתובות מחדש (Relocations)

ה-RVA של **DataDirectory[5]** הולך להצביע על נתוני הכתובות החדשות (Relocation Information). **DataDirectory[5]** נקרא IMAGE_DIRECTORY_ENTRY_BASERELOC ונראה כך (מספר 18 בתמונת פורמט ה-PE):

```
IMAGE_DIRECTORY_ENTRY_BASERELOC

struct _IMAGE_DATA_DIRECTORY {
0x00   DWORD VirtualAddress;
0x04   DWORD Size;
};
```

אני אסביר מה הכוונה בנתוני הכתובת החדשות באמצעות דוגמא. תחשבו על מקרה שבו קובץ רוצה להיות ממוקם בכתובת ההתחלתית 0x1000000 והוא נטען בכתובת 0x2000000. לכן, אנחנו חייבים לסדר מחדש את הכתובות של כל הקבועים. מערכת ההפעלה תשתמש בנתוני הכתובות החדשות כדי לחפש את קבועים אלו בשביל לסדר אותם מחדש. נתוני הכתובות החדשות יהיו בעצם רשימה של כל הקבועים אשר נצטרך להזיז אותם בזיכרון במידה ויקרה מקרה כמו בדוגמא. נתוני הכתובות החדשות בדרך כלל ממוקמות באגף reloc..

את הדבר שאני הולך להציג לכם עכשיו לא תמצאו בתמונת פורמט ה-PE, אבל IMAGE_DIRECTORY_ENTRY_BASERELOC מצביע אל מערך של מבני נתונים מסוג **IMAGE_BASE_RELOCATION**.

```
typedef struct _IMAGE_BASE_RELOCATION {
    DWORD   VirtualAddress;
    DWORD   SizeOfBlock;
//  WORD    TypeOffset[1];
} IMAGE_BASE_RELOCATION;
```

VirtualAddress מציין את הכתובת המיושרת אשר כתובות היעד יהיו רלוונטיות אליו. במילים אחרות, השדה **VirtualAddress** אומר שמבנה הנתונים IMAGE_DIRECTORY_ENTRY_BASERELOC יתאים את עצמו לתחום מסויים של RVA-ים (זה יהיה רק טווח של 0x1000) וזה אומר שכל הנתונים (במקרה זה, רשימה של כל הדברים שאנחנו צריכים לסדר בתוך מרחב זיכרון זה) הקשורים לזה יהיו רלוונטים מכתובת 0x1000 לכתובת 0x2000. מבנה הנתונים הבא יהיה רלוונטי מכתובת 0x2000 לכתובת 0x3000 וכך הלאה.

SizeOfBlock הוא הגודל של IMAGE_BASE_RELOCATION עצמו + כל נתוני כתובות היעד. במילים אחרות, מהו הגודל של רשימת הדברים שנצטרך לסדר אותם במרחב כתובת זה.

הרשימה הבאה לאחר **SizeOfBlock** היא מספר משתנים של כתובות יעד בגודל WORD. בעצם, קבוצה של WORD-ים עם הכתובות המדוייקות של הקבועים שנצטרך לסדר אותם, לאחר חישוב.

מספר זה יכול להיות מחושב כך (משמאל לימין):

הגודל של WORD / (הגודל של IMAGE_BASE_RELOCATION - **SizeOfBlock**)

ארבעת הביטים העליונים מתוך 16 הביטים של כתובת היעד מציינים את סוגו. 12 הביטים התחתוניים מציינים את ה-Offset, אשר ישומש בצורה שונה בהתאם לסוג. הסוגים הם:

```
#define IMAGE_REL_BASED_ABSOLUTE              0
#define IMAGE_REL_BASED_HIGH                  1
#define IMAGE_REL_BASED_LOW                   2
#define IMAGE_REL_BASED_HIGHLOW               3
#define IMAGE_REL_BASED_HIGHADJ               4
#define IMAGE_REL_BASED_MIPS_JMPADDR          5
#define IMAGE_REL_BASED_MIPS_JMPADDR16        9
#define IMAGE_REL_BASED_IA64_IMM64            9
#define IMAGE_REL_BASED_DIR64                 10
```

אכפת לנו רק מהסוג **IMAGE_REL_BASED_HIGHLOW**, אשר משומש כאשר ה-RVA של הכתובת החדשה צויין באמצעות **VirtualAddress** + 12 הביטים התחתוניים.

בואו נפתח את PEView:

00021690	00003000	RVA of Block	
00021694	0000003C	Size of Block	
00021698	32FB	Type RVA	000032FB IMAGE_REL_BASED_HIGHLOW
0002169A	3307	Type RVA	00003307 IMAGE_REL_BASED_HIGHLOW
0002169C	334A	Type RVA	0000334A IMAGE_REL_BASED_HIGHLOW

בתמונה למעלה אם הקובץ ימוקם מחדש, ה-Loader יקח את כתובת היעד 0x32FB. מכיוון שארבעת הביטים העליונים הם 0x3 = **IMAGE_REL_BASED_HIGHLOW**. 12 הביטים התחתוניים הם 0x2FB. בהינתן הסוג, אנחנו נעשה את החישוב (0x3000) **VirtualAddress** ועוד 12 הביטים התחתוניים (0x2FB) והתוצאה 0x32FB תהיה ה-RVA של המיקום אשר צריך להיות מתוקן במידה והקובץ ימוקם מחדש.

לאחר מכן ה-Loader פשוט יוסיף את המרחק בין הכתובת הרצוייה לטעינת הקובץ לבין כתובת הטעינה האמיתית ופשוט יוסיף את מרחב זה לנתונים בתוך ה-RVA 0x32FB.

Thread Local Storage (TLS)

לפני שנמשיך, אם אתם לא יודעים מה זה "Thread", Google it ☺

אז כפי שאנחנו יודעים, כל Thread יכול לראות את אותם משתנים גלובלים, אבל פעולה כלשהי חייבת להיעשות כדי לוודא שהם לא נתקלים במצב שבו Thread 2-ים מנסים לגשת ולשנות את חלק מהמשתנים בצורה אשר תפריע לריצה של השני באמצעות הריסת הציפיות שלו.

לכן, רצוי לפעמים שיהיה לנו משתנים (חוץ ממשתנים לוקאלים הממוקמים ב-Stack) אשר נגישים רק ל-Thread אחד, מכיוון שאם Thread-ים מרובים ינסו לשנות את אותו משתנה גלובלי הם יכולים להרוס את המשתנה אחד לשני. במילים אחרות, אנחנו נרצה שהמשתנה שלנו יהיה לוקאלי וגם יוכל להיות נגיש ל-Threadשלנו ואנחנו נשתמש בו בצורה נפרדת משאר ה-Thread-ים.

Thread Local Storage (TLS) הוא מנגנון אשר Microsoft סיפקו לנו במפרט של ה-PE כדי לתמוך במטרה זאת. הם תומכים בנתונים רגילים וכן ב-Callback Functions, אשר יכולים לאתחל/להרוס נתונים בתהליך היצירה/ההריסה של Thread. נתוני ה-TLS מאוחסנים בדרך כלל באגף tls... ה-RVA של **DataDirectory[9]** הולך להצביע אל מבנה נתונים בשם **IMAGE_TLS_DIRECTORY**.

DataDirectory[9] נקרא IMAGE_DIRECTORY_ENTRY_TLS ונראה כך (מספר 19 בתמונת פורמט ה-PE):

```
IMAGE_DIRECTORY_ENTRY_TLS

struct _IMAGE_DATA_DIRECTORY {
0x00   DWORD VirtualAddress;
0x04   DWORD Size;
};
```

IMAGE_TLS_DIRECTORY נראה כך (מספר 20 בתמונת פורמט ה-PE):

```
struct _IMAGE_TLS_DIRECTORY {
0x00  DWORD      StartAddressOfRawData;
0x04  DWORD      EndAddressOfRawData;
0x08  LPDWORD    AddressOfIndex;
0x0c  PIMAGE_TLS_CALLBACK *AddressOfCallBacks;
0x10  DWORD      SizeOfZeroFill;
0x14  DWORD      Characteristics;
};
```

```
typedef struct _IMAGE_TLS_DIRECTORY32 {
    DWORD    StartAddressOfRawData;
    DWORD    EndAddressOfRawData;
    DWORD    AddressOfIndex;
    DWORD    AddressOfCallBacks;
    DWORD    SizeOfZeroFill;
    DWORD    Characteristics;
} IMAGE_TLS_DIRECTORY32;
```

StartAddressOfRawData הוא ה-VA שבו הנתוני ה-TLS מתחילים.

EndAddressOfRawData הוא ה-VA שבו הנתונים ה-TLS מסתיימים.

AddressOfCallbacks הוא VA אשר מצביע אל מערך של פויינטרים לפונקציות מסוג PIMAGE_TLS_CALLBACK. **AddressOfCallbacks** יהיה מערך של VA-ים לפונקציות אשר אנחנו נרצה

לקרוא כאשר ה-Thread החדש שלנו יתחיל לרוץ. פונקציות אלו יתחילו כאשר ה-Thread יתחיל לרוץ לטובת אתחול נתונים או לטובת דברים אחרים.

SizeOfZeroFill הוא הגודל של החלק הלא מאותחל של מבנה ה-TLS, חלק זה מלא באפסים. שדה זה מעניין מכיוון שהוא כמו החלק עם האפסים באגף הbss., הנעוץ אחרי נתוני ה-TLS.

פונקציות Callbacks ירוצו **לפני** מה שנמצא בכתובת שבתוך **OPTIONAL_HEADER.AddressOfEntryPoint**. חשוב לדעת את זה מכיוון שכאשר אנחנו ננתח קובץ בינארי באופן סטאטי מההתחלה אנחנו נרצה לדעת שפונקציות אלו ירוצו לפני הכתובת המוגדרת כ"התחלת הבינארי" ויכולות לעשות דברים, כמו לתקשר עם שרת, לפני מה שמוגדר בכתובת **OPTIONAL_HEADER.AddressOfEntryPoint**.

משאבים (Resources)

ה-RVA של **DataDirectory[2]** הולך להצביע אל מבנה נתונים בשם **IMAGE_RESOURCE_DIRECTORY**. **DataDirectory[2]** נקרא IMAGE_DIRECTORY_ENTRY_RESOURCE ונראה כך (מספר 21 בתמונת פורמט ה-PE):

```
IMAGE_DIRECTORY_ENTRY_RESOURCE

struct _IMAGE_DATA_DIRECTORY {
0x00   DWORD VirtualAddress;
0x04   DWORD Size;
};
```

IMAGE_RESOURCE_DIRECTORY נראה כך (מספר 22 בתמונת פורמט ה-PE):

```
struct _IMAGE_RESOURCE_DIRECTORY {
0x00  DWORD   Characteristics;
0x04  DWORD   TimeDateStamp;
0x08  WORD    MajorVersion;
0x0a  WORD    MinorVersion;
0x0c  WORD    NumberOfNamedEntries;
0x0e  WORD    NumberOfIdEntries;
};
```

```
typedef struct _IMAGE_RESOURCE_DIRECTORY {
    DWORD   Characteristics;
    DWORD   TimeDateStamp;
    WORD    MajorVersion;
    WORD    MinorVersion;
    WORD    NumberOfNamedEntries;
    WORD    NumberOfIdEntries;
} IMAGE_RESOURCE_DIRECTORY,
```

דיברנו על משאבים כאן. המשאבים בדרך כלל מאוחסנים באגף rsrc., כמו שציינתי בעבר. לפרוטוקול, במקרה של Stuxnet, בתוך אגף ה-rsrc. היו את כל ה-Exploit-ים ואת כל ה-DLL-ים שהנוזקה רצתה להזריק. המשאבים יכולים לפעמים להזדהות באמצעות שם אמיתי ולפעמים להזדהות באמצעות מספר מזהה, אבל לא באמצעות שניהם.

NumberOfNamedEntries הוא מספר הייצואים אשר מוגדרים באמצעות שם ו- **NumberOfIdEntires** הוא מספר הייצואים אשר מוגדרים באמצעות מספר מזהה.

לאחר IMAGE_RESOURCE_DIRECTORY, יהיה מערך של **NumberOfIdEntries** + **NumberOfNamedEntries** ערכים אשר יכילו Struct-ים מסוג IMAGE_RESOURCE_DIRECTORY_ENTRY (אם ערכי השמות בהתחלה ואחריו ערכי המספרים המזהים). את **IMAGE_RESOURCE_DIRECTORY_ENTRY** לא רואים בתמונת פורמט ה-PE, אבל הוא נראה כך:

```
typedef struct _IMAGE_RESOURCE_DIRECTORY_ENTRY {
    union {
            struct {
                  DWORD NameOffset:31;
                  DWORD NameIsString:1;
            };
            DWORD   Name;
            WORD    Id;
    };
    union {
            DWORD   OffsetToData;
            struct {
                  DWORD   OffsetToDirectory:31;
                  DWORD   DataIsDirectory:1;
            };
    };
} IMAGE_RESOURCE_DIRECTORY_ENTRY;
```

זה פשוט יותר ממה שזה נראה. הנתונים אשר נמצאים בימין הקיצוני בבינארי (The least significant data) תמיד יהיו רשומים ראשונים במבני נתונים של שפת C. NameOffset נמצא בבינארי ב-31 הביטים הימניים ביותר (Least significant 31 bits), וביט 1 של NameIsString ישאר הביט השמאלי ביותר (Most-Significant Bit, או MSB, הביט השמאלי ביותר, לדוגמא ברצף הביטים **0**1100110 ה-MSB מודגש). אם ה-MSB של ה-DWORD הראשון (במקרה זה NameIsString) יוגדר כ-1 (100000 בייצוג הקסדצימלי זה 80, אז המספר הראשון בערך זה יהיה 8), זה אומר ש-31 הביטים הנמוכים יותר (במקרה זה NameOffset) הם Offset למחרוזת שמייצגת את שם המשאב (ומיוצגת בתור מחרוזת Wide character Pascal. לכן, במקום שהמחרוזת תסתיים ב-null היא תתחיל עם גודל אשר מציין את מספר התווים העוקבים). במילים אחרות, אם ה-MSB יוגדר כ-1, יתייחסו אליו בתור RVA בגודל DWORD למחרוזת, וזה בעצם שם הדבר המיוצא (DWORD Name).

אם ה-MSB יוגדר כ-0, יתייחסו אליו בתור מספר מזהה בגודל WORD. אם ה-MSB של ה-DWORD השני יוגדר כ-1 (במקרה זה DataIsDirectory) זה אומר ש-31 הביטים הנמוכים יותר (במקרה זה OffsetToDirectory), יהיו Offset למבנה נתונים אחר מסוג IMAGE_RESOURCE_DIRECTORY. אם ה-MSB יוגדר כ-0 זה אומר שהוא Offset לנתונים בפועל. כדי לפשט את כל מה שאמרתי עכשיו, אלו בעצם רק 2 דרכים שונות להסתכל על הנתונים. ה-DWORD הראשון הוא מחרוזת או מספר מזהה וה-DWORD השני הוא פויינטר לספרייה או פויינטר לנתונים.

תצורת הטעינה (Load Configuration)

נושא זה רלוונטי לאבטחת מידע. ה-RVA של **DataDirectory[10]** הולך להצביע אל מבנה נתונים בשם LOAD_CONFIG_DIRECTORY. **DataDirectory[10]** נקרא IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG ונראה כך (מספר 23 בתמונת פורמט ה-PE):

```
IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG

struct _IMAGE_DATA_DIRECTORY {
0x00   DWORD VirtualAddress;
0x04   DWORD Size;
};
```

IMAGE_LOAD_CONFIG_DIRECTORY נראה כך:

32 ביט:

```
typedef struct {
    DWORD    Size;
    DWORD    TimeDateStamp;
    WORD     MajorVersion;
    WORD     MinorVersion;
    DWORD    GlobalFlagsClear;
    DWORD    GlobalFlagsSet;
    DWORD    CriticalSectionDefaultTimeout;
    DWORD    DeCommitFreeBlockThreshold;
    DWORD    DeCommitTotalFreeThreshold;
    DWORD    LockPrefixTable;
    DWORD    MaximumAllocationSize;
    DWORD    VirtualMemoryThreshold;
    DWORD    ProcessHeapFlags;
    DWORD    ProcessAffinityMask;
    WORD     CSDVersion;
    WORD     Reserved1;
    DWORD    EditList;
    DWORD    SecurityCookie;
    DWORD    SEHandlerTable;
    DWORD    SEHandlerCount;
} IMAGE_LOAD_CONFIG_DIRECTORY32
```

64 ביט:

```
typedef struct {
    DWORD       Size;
    DWORD       TimeDateStamp;
    WORD        MajorVersion;
    WORD        MinorVersion;
    DWORD       GlobalFlagsClear;
    DWORD       GlobalFlagsSet;
    DWORD       CriticalSectionDefaultTimeout;
    ULONGLONG   DeCommitFreeBlockThreshold;
    ULONGLONG   DeCommitTotalFreeThreshold;
    ULONGLONG   LockPrefixTable;
    ULONGLONG   MaximumAllocationSize;
    ULONGLONG   VirtualMemoryThreshold;
    ULONGLONG   ProcessAffinityMask;
    DWORD       ProcessHeapFlags;
    WORD        CSDVersion;
    WORD        Reserved1;
    ULONGLONG   EditList;
    ULONGLONG   SecurityCookie;
    ULONGLONG   SEHandlerTable;
    ULONGLONG   SEHandlerCount;
} IMAGE_LOAD_CONFIG_DIRECTORY64,
*PIMAGE_LOAD_CONFIG_DIRECTORY64;
```

SecurityCookie הוא VA אשר מצביע אל המיקום שבו מחסנית ה-Cookie אשר משומשת עם הדגל (Flag) GS/ תאוחסן.

<u>הערה:</u> בשביל מה שאני הולך להסביר עכשיו אתם תצטרכו לדעת קצת אסמבלי x86.

דגל ה-GS/ אומר שהקומפיילר ישים ערך רנדומלי בין המשתנים הלוקאלים שלנו לבין ה-Registersהשמורים שלנו. לכן, אם התוקף ידרוס את המשתנים הלוקאלים שלנו בשביל להשיג את ה-EIP Register, לפני ריצת הקוד, מערכת ההפעלה תבדוק אם ה-Coockie "נשפך" (Was overflowed)

ושונה באמצעות השוואה עם אותו ערך רנדומלי אשר ימוקם בסוף הקוד. אם הוא שונה, מערכת ההפעלה תדע שבוצעה מתקפת Buffer Overflow ותפסיק את ריצת הקוד.

SEHandlerTable הוא VA אשר מצביע אל טבלת ה-RVA-ים אשר מציינים את הפונקציות היחידות לטיפול בחריגים (Exception Handlers) אשר ניתן להשתמש בהן עם ה-(SEH) Structured Exception Handler. SEH הוא פויינטר של פונקצייה למבנה של חריגות אשר Windows משתמש כדי להתמודד עם אירועים מסויימים. במילים אחרות, אם בעיה נגרמת, תריץ את הפונקציה אשר תטפל בחריגה זו בהתאם. התוקף יכול לכתוב מחדש נתונים נוספים בתוך המחסנית. התחליף של הפויינטרים למטפלים בחריגות אלו נגרמת כתוצאה מאופציית הקישור SAFESEH/. **SEHandlerTable** הוא פויינטר לטבלה אשר תוכנתה מראש לתוך הבינארי עם טיפולי חריגות שעלולים להיקרא לטובת השוואת הכתובות של טיפולי החריגות עם ה-SEH.

SEHandlerCount הוא מספר הערכים במערך אשר SEHandlerTable מצביע אליו.

Directory Entry Security

הנושא האחרון שלנו למאמר זה הוא **Directory Entry Security**. משתמשים בזה לטובת השוואה בין תעודת הקוד (Code Certificate) לתעודת הקוד שאמורה להיות. אחרת, הקוד לא ירוץ. ה-RVA של **DataDirectory[4]** הולך להצביע אל תעודה דיגיטלית (Digital Certificate), במידה וקיימת חתימה דיגיטלית המוטבעת בתוך הקובץ. **DataDirectory[4]** נקרא IMAGE_DIRECTORY_ENTRY_SECURITY ונראה כך (מספר 24 בתמונת פורמט ה-PE):

```
IMAGE_DIRECTORY_ENTRY_SECURITY

struct _IMAGE_DATA_DIRECTORY {
0x00   DWORD VirtualAddress;
0x04   DWORD Size;
};
```

Authenticode היא מילת המפתח של Microsoft עבור הפעולה של לחתום דיגיטלית על קבצים. זוכרים מה שאמרנו שיש את IMAGE_DLLCHARACTERISTICS_FORCE_INTEGRITY אשר אומר ל-Loader לבצע בדיקת חתימות דיגיטליות לפני שהוא נותן לקוד לרוץ? אז בעצם זה אומר לערך בתוך ה-RVA של **DataDirectory[4]** לבדוק את החתימה הדיגיטלית.

לסיכום

זה היה מאמר בנושא פורמט ה-Portable Executable. כיסינו את רוב הנושאים החשובים הנוגעים ל-PE במאמר זה וגם קצת נושאים אחרים. נושא זה הוא הנושא העיקרי שצריך ללמוד לפני שאתם לומדים כיצד לכתוב נוזקות (זה וגם אסמבלי x86 ודיבאגינג). אם את/ה מפתח/ת נוזקות מתחיל/ה, מה שלמדת עכשיו, ישרת אותך בהמשך המסע שלך.

אולי שאלתם את עצמכם מה בנוגע לשאר הנושאים שלא כיסיתי? קודם כל, רוב הערכים בתוך DataDirectory[] לא רלוונטים ל-x86 ולכן לא כיסיתי אותם. אם אתם מעוניינים בשאר הערכים של ה-DataDirectory[] או רוצים לדעת עוד על השדות אשר לא סומנו ב**כחול**, אתם תמיד יכולים להשתמש ב-MSDN, יש להם את התיעוד של כל מבנה ה-PE. כיסיתי רק מה שאני חושב לנכון ולא את הכל, מכיוון שאם הייתי עושה אחרת, מאמר זה היה הרבה יותר ארוך ממה שהוא עכשיו.
אני יודע שחלק מהמילים במאמר זה באנגלית ואני מתנצל על כך. תחילה כתבתי את מאמר זה באנגלית וכשתרגמתי אותו לא הצלחתי למצוא תרגום נורמלי למילים אלו בעברית ולשמור על הקשר המשפט תקין ומובן. אם יש לכם שאלות כלשהן או הצעות לתיקונים למאמר (כי אף אחד לא מושלם), מוזמנים ליצור איתי קשר במייל TheSpl0itBlog@gmail.com.

בברכה,

Spl0it

ביבליוגרפיה

תודה ענקית לזינו קובך (Xeno Kovah), יוצר הקורס "The Life of Binaries". מאמר זה מבוסס בגסות על קורס זה.

1. http://opensecuritytraining.info/LifeOfBinaries.html
2. https://msdn.microsoft.com/en-us/library/windows/desktop/ms680547(v=vs.85).aspx
3. https://msdn.microsoft.com/en-us/library/ms809762.aspx
4. https://en.wikipedia.org/wiki/Portable_Executable
5. http://www.csn.ul.ie/~caolan/publink/winresdump/winresdump/doc/pefile2.html
6. https://en.wikibooks.org/wiki/X86_Disassembly/Windows_Executable_Files
7. https://www.opswat.com/blog/closer-look-portable-executable-information
8. http://www.openrce.org/reference_library/files/reference/PE%20Format.pdf