Digital Whisper
Technologic Papers

מערכות הפעלה - שימוש ב-IDA למחקר בסיסי של תוכנות

מאת ברק גונן, המרכז לחינוך סייבר

הקדמה

המאמר הבא הינו הפרק השני מתוך ספר לימוד חדש שעתיד לצאת בספטמבר הקרוב כחלק מפעילות המו"פ של המרכז לחינוך סייבר של קרן רש"י. הספר נכתב ע"י ברק גונן והוא יונגש, יחד עם חומרים נוספים של המרכז לחינוך סייבר, בחינם ולשימוש הכלל. הספר יעסוק בחקר מערכות הפעלה עם דגש עם תחום הסייבר וההנדסה לאחור: עבודה עם קבצים בינארים, תהליכים, זכרונות, קמפול, עבודה עם דיסאסמבלר, מבנה מערכת הקבצים, שינוי התנהגות של תוכנה, הזרקת DLL-ים, שפות סקריפטינג, ומעט על נוזקות - שווה לחכות!

מטרת הפרק

בפרק זה נבצע "יישור קו" של נושאים בסיסיים בפעולת המחשב. לאחר שניזכר כיצד עובד המעבד, נענה על השאלה ממה מורכבות תוכנות מחשב? כדי לענות על שאלה זו, ננקוט בגישה מחקרית. נקמפל קטעי קוד שונים ונראה היכן מתבצעת השמירה בזיכרון. הקוד ייכתב בשפת C, קימפול התוכניות יתבצע באמצעות Visual Studio ואילו המחקר יתבצע בעזרת כלי בשם IDA. מטרה עקיפה של הפרק היא לימוד שפת C תוך כדי התנסות. הפרק אינו מדריך לשפת C ואם אינכם מכירים את השפה סביר שתצטרכו ללמוד דברים מסויימים בכוחות עצמכם, אך הפרק יעניק לכם כמה קטעי קוד בסיסיים ופשוטים שתוכלו לחקור ולהתקדם באמצעותם. ההיכרות עם IDA תפתח לנו צוהר לעולם ה-Reverse Engineering המרתק, ומי שיאהב את הטעימה מיכולת המחקר שהתוכנה מציעה יוכל להעמיק באופן עצמאי.

כיצד עובד המעבד?

המעבד מריץ פקודות בשפת מכונה. שפת מכונה היא שפה של אחדות ואפסים והיא קשורה באופן הדוק למעבד הספציפי שאנחנו עובדים איתו. לדוגמה, הפקודה "העתק את X לתוך Y" תיכתב באופן אחד במעבדים מתוצרת אינטל ובאופן אחר במעבדים מתוצרת AMD. לשם המחשה, פקודת העתקה במעבד אינטל עשויה להיות "10100000" ואילו פקודת העתקה ב-AMD עשויה להיות "10001000". נדגיש כי יש מגוון רב של פקודות העתקה, והביטים משתנים לפי סוג המקור והיעד, אך העקרון הוא שלכל מעבד יש פקודות משלו ואין תאימות בין המעבדים השונים. למעשה, גם בין משפחות מעבדים שונים של אותו יצרן

יש הבדלים. לדוגמה במעבדי אינטל יש הבדלים בין פקודות שמיועדות למשפחת מעבדי ה-32 ביט לעומת מעבדי ה-64 ביט.

בדרך כלל ישנה תאימות לאחור, כלומר מעבד של 64 ביט יודע להריץ גם פקודות של 32 ביט, אך ההיפך אינו מתקיים מעבד 32 ביט לא יודע להריץ פקודות של 64 ביט, גם אם הן של אותו יצרן מעבד.

<u>תזכורת: מה משמעות מעבד 32 למול 64 ביט?</u>

לכל מעבד יש חלקי חומרה הנקראים רגיסטרים ומשמשים לביצוע כל פעולות החישוב, הגישה לזיכרון והרצת התוכנית. מספר הרגיסטרים שונה בין דורות שונים של מעבדים, וגם גודל הרגיסטרים שונה בין דורות המעבדים. כאשר אומרים "מעבד 32 ביט" מתכוונים לכך שגודל הרגיסטרים שלו הוא 32 ביט. למעבד 64 ביט יש, כצפוי, 64 ביט בכל רגיסטר. מה משמעות הדבר?

רגיסטר גדול יותר יכול להחליף יותר מידע עם הזיכרון בכל גישה לזיכרון. נניח שיש לנו זיכרון בגודל 16 בתים (בית= 8 ביט) ואנחנו מעוניינים להעתיק אותו למקום אחר בזיכרון. רגיסטר של 32 ביט מסוגל לטפל ב-4 בתים של זיכרון בכל פעם, לכן נצטרך לגשת אל הזיכרון 8 פעמים. מתוכן 4 פעמים יהיו קריאות מהזכרון ממנו מעתיקים, ו-4 יהיו גישות לזיכרון אליו מעתיקים. לעומת זאת, רגיסטר של 64 ביט יוכל לבצע את אותה עבודה בחצי מכמות הגישות לזיכרון וכך לחסוך לנו זמן רב.

הבדל משמעותי נוסף הוא שינוי בגודל פס הכתובות (Address Bus). למעבד 32 ביט יש פס כתובות בגודל 32 ביט. כלומר ניתן לייצג 2 בחזקת 32 כתובות שונות, או במילים פשוטות, בערך 4 ג'יגה בתים. המשמעות היא שגם אם יש למחשב שלנו יותר מ-4 ג'יגה זיכרון, לא ניתן יהיה לנצל אותו. המעבד פשוט לא יודע איך לגשת לזיכרון שאי אפשר לייצג את הכתובת שלו בפס הכתובות. בעבר 4 ג'יגה נחשב דמיוני, אולם כיום קשה לרכוש מחשב שיש בו רק 4 ג'יגה זיכרון RAM, שלא לדבר על דיסק קשיח.

חשוב להדגיש שכדי לעבוד עם כתובות של 64 ביט, לא די בכך שהמעבד יתמוך בכך. יש צורך שגם מערכת ההפעלה תתמוך בכך. לכן למרבית מערכות ההפעלה יש גרסאות 32 ו-64 ביט. מה בדיוק צריכה לעשות מערכת ההפעלה כדי לתמוך ב-64 ביט? על כך נלמד בפרק שעוסק בזיכרון.

כיצד עובד המעבד - המשך

כאשר אנחנו כותבים קוד בשפה עילית, כגון C, איננו צריכים להיות מוטרדים לגבי סוג המעבד עליו תרוץ התוכנית שלנו. הקוד בשפת C נראה אותו דבר על סוגי מעבדים שונים. אולם, ברגע שנריץ על הקוד אסמבלר שיהפוך אותו לשפת אסמבלי, נצטרך לבחור לאיזה סוג מעבד האסמבלי יהיה נכון. בשפת אסמבלי ובשפת מכונה כל קוד חייב להתאים למעבד ספציפי.

המעבד מריץ רצפים של פקודות בשפת מכונה, המונח הלועזי הוא Instruction. יכול להיות Instruction להעתקה של ערך לתוך רגיסטר, Instruction לחיבור של שני רגיסטרים וכו'. הנה כך נראות מספר Instructions בשפת אסמבלי שנכתבו עבור מעבד אינטל 32 ביט:

```
mov     eax, 2
mov     ebx, 3
add     eax, ebx
```

בזיכרון של המחשב הפקודות הללו לא שמורות כפי שהן כאן, אלא שמור התרגום שלהן לשפת מכונה. אך כאשר נתבונן בזיכרון המחשב לא נראה רצף של אחדות ואפסים אלא ספרות הקסדצימליות.

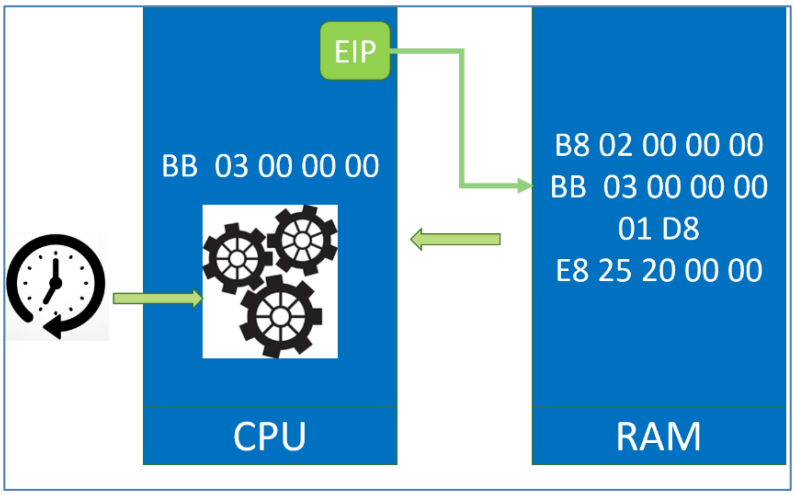
לנו בני האדם קשה לעבוד עם מספרים בינאריים, קל מאד להתבלבל. נסו להקריא ולזכור את רצף המספרים הבא: "1001111111100011". לא נעים! מצד שני, אם נתרגם את אותו רצף לספרות הקסדצימליות הוא ייכתב כך: "9FE3". נכון עכשיו זה ברור? טוב, לא בדיוק, עדיין מדובר בשפת מכונה, אבל כעת יותר קצר לכתוב את הפקודה ולזכור אותה. לכן כל כלי דיבוג שמציג לנו את הזיכרון של המחשב יציג זאת לא בבינארי אלא בהקסצימלי.

בשפת מכונה, הפקודות שרשמנו למעלה ייראו כך:

```
B8  02  00  00  00
BB  03  00  00  00
01  D8
```

בזמן ריצת התוכנית, הפקודות הללו מועתקות מהדיסק הקשיח אל זיכרון מהיר יותר שנקרא RAM. המעבד מכיל רגיסטר שנקרא EIP, קיצור של Extended Instruction Pointer, והוא זה שמצביע על הפקודה הבאה שיש להריץ. בכל פעם שהשעון של המעבד מתקדם, ה-EIP עובד להצביע על הפקודה הבאה.

עד אז המעבד חייב להשלים את ביצוע הפקודה הנוכחית:



כדי להבין יותר לעומק איך מורכבות תוכנות מחשב, נחקור כיצד קוד שאנחנו כותבים שמור כשפת מכונה. כדי לעשות זאת נצטרך לרכוש ידע בכלי מתאים למשימה. לפני הכל, קיראו את הנספח "מדריך בסיסי לעבודה עם IDA". לאחר שתשלטו בבסיס של התוכנה, המשיכו לביצוע המשימות הבאות.

<u>מחקר אזורי הזיכרון השונים של תוכנה</u>

את החלק הזה נלמד באמצעות משימות מחקר קטנות. בכל משימה כזו ניקח קטע קוד קטן ונבדוק כיצד נשמר בזיכרון דבר מה שמעניין אותנו. מה מעניין אותנו? אנחנו רוצים לדעת היכן בזיכרון נשמר קוד של תוכנה שאנחנו מריצים. לתוכנה יש כל מיני חלקים:

- פקודות
- קבועים
- משתנים גלובליים
- משתנים מקומיים
- אזורי זיכרון שמוקצים תוך כדי ריצה (פקודת malloc)
- פונקציות שאנחנו מייבאים בזמן ריצה (מתוך קבצים מסוג DLL, נלמד עליהם בהמשך)

לכן בכל פעם ניקח תוכנה שכתבנו וכוללת את אחד הדברים האלה ונבדוק היכן מוקצה הזיכרון.

דגש חשוב להמשך: אנחנו חוקרים כעת את הקוד כפי שהוא שמור בקובץ ההרצה שיצרנו, קובץ מסוג EXE. קובץ ה-EXE שמור על הדיסק הקשיח ולא טעון אל ה-RAM. בתהליך הטעינה ל-RAM יש דברים רבים שמשתנים, ונעמוד עליהם בהמשך. כרגע מה שחשוב לזכור הוא שאנחנו צופים בתמונת מצב של אזורי הזיכרון כפי שהם מוגדרים על הדיסק הקשיח, לא בתמונת הזיכרון בתוך ה-RAM.

שלב א'- פקודות

השתמשו בתוכנית hello.exe שיצרתם עבור נספח לימוד ה-IDA. היכן נשמרות הפקודות עצמן? כפי שאתם רואים, באזור זיכרון שנקרא text. מה אפשר לדעת על אזור זיכרון זה?

כשאתם בתצוגת קוד (לא תצוגה גרפית) גללו למעלה עד שתגיעו לתחילת אזור ה-text. יופיע לפניכם טקסט דומה לזה:

```
; Section 2. (virtual address 00002000)
; Virtual size                 : 00000019 (     25.)
; Section size in file         : 00000200 (    512.)
; Offset to raw data for section: 00000600
; Flags 60000020: Text Executable Readable
; Alignment     : default

; Segment type: Pure code
; Segment permissions: Read/Execute
_text segment para public 'CODE' use32
assume cs:_text
;org 402000h
assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing
```

על מה אנחנו מסתכלים? IDA לוקחת את ההגדרות של אזורי הזיכרון מתוך ההגדרות השמורות בקובץ ה-exe. התוכנה מציגה לנו בצורה נוחה לקריאה את משמעות המידע שנמצא ב-exe. השורה שמעניינת אותנו היא היכן שכתוב "Flags 60000020: Text Executable Readable". הפירוש של השורה ניתן מיד אחר כך בהערות האפורות- זהו אזור זיכרון שהוא "Pure Code" ויש לו הרשאות של קריאה והרצה.

שלב ב' - קבועים

חיקרו בעצמכם היכן שמורה המחרוזת "Hello!" שהגדרתם.

רק לאחר מכן עיברו על ההסבר הבא ובידקו שהגעתם לאותה תוצאה.

בתוך ה-main שלנו, נתמקד בשתי שורות. הראשונה, push offset aHello. שורה זו דוחפת לתוך המחסנית את הכתובת בזכרון של מחרוזת Hello. אם נקליק על aHello נופנה לכתובת של המחרוזת בזכרון:

```
.rdata:00416B92                 align 4
.rdata:00416B94 aHello          db 'Hello!',0Ah,0       ; DATA XREF: sub_412320+1E↑o
.rdata:00416B9C aStackMemoryCor db 'Stack memory corruption',0
```

ניתן לראות שהמחרוזת מוגדרת בכתובת B94416, ולאחריה מוגדר התו Ah0 (כזכור מלימודי האסמבלי, ירידת שורה) ולאחריה התו 0 (כזכור, מציין סיום המחרוזת לטובת פונקציות הדפסה).

אך מהו rdata? אם נגלול למעלה את הזיכרון, נגיע אל תחילת החלק בזיכרון בו מוגדר rdata:

```
.rdata:00416000 ; Section 3. (virtual address 00016000)
.rdata:00416000 ; Virtual size                  : 00001FE9 (   8169.)
.rdata:00416000 ; Section size in file          : 00002000 (   8192.)
.rdata:00416000 ; Offset to raw data for section: 00005200
.rdata:00416000 ; Flags 40000040: Data Readable
.rdata:00416000 ; Alignment     : default
.rdata:00416000 ; ===================================================
.rdata:00416000
.rdata:00416000 ; Segment type: Pure data
.rdata:00416000 ; Segment permissions: Read
.rdata:00416000 _rdata          segment para public 'DATA' use32
.rdata:00416000                 assume cs:_rdata
```

רואים ש-rdata הוא אזור בזיכרון בעל האפיון "Data Readable". ההרשאות של אזור זה בזכרון הן "Read". אי אפשר לכתוב לאזור זה בזיכרון. המשמעות היא שהמחרוזת Hello שהגדרנו בקוד היא קבועה, נשמרת במקום שמיועד לקריאה בלבד. ואכן בהמשך נראה שאילו היינו רוצים שהמחרוזת שלנו תוכל להשתנות היה עלינו להגדיר אותה בצורה שונה.

שלב ג' - משתנים גלובליים

הגדירו אוסף של משתנים גלובליים מסוגים שונים וצפו במקום שבו הם נשמרים בזיכרון.

```
// global variables
char      my_char = 'b'
short     my_short = 4000;
int       my_int = 10;
long      my_long = 65535;
double    my_double = 4.32;
```

שימו לב, שכדי שהמשתנים הללו יופיעו בקוד ה-exe, קימפול התוכנית חייב להיות עבור גרסת debug. כאשר מקמפלים עבור גרסת release, הקומפיילר נוטה להעיף חלקי קוד שאין בהם שימוש.

אם ביצעתם את הדברים נכון, מצאתם את המשתנים הללו שמורים באזור זיכרון שנקרא data. מה ההבדל בינו לבין אזור הזכרון rdata, ששימש לשמירת הקבוע "Hello!"?

```
.data:00403000 ; Flags C0000040: Data Readable Writable
.data:00403000 ; Alignment     : default
.data:00403000 ; ===================================================
.data:00403000
.data:00403000 ; Segment type: Pure data
.data:00403000 ; Segment permissions: Read/Write
```

כפי שרואים, data הוא בעל הרשאות כתיבה וקריאה, לא רק קריאה. זה הגיוני- אחרי הכל הגדרנו משתנים, לא קבועים.

שלב ד' - משתנים מקומיים

כעת נגדיר פונקציה ובתוכה נשים מספר משתנים מקומיים. חשוב להגדיר פונקציה שמקבלת פרמטרים, כדי שנוכל לראות גם היכן נשמרים הפרמטרים שמועברים לפונקציה. לשלב הזה, וכן לשלב הבא במחקר, מומלץ להשתמש בקוד הבא:

```
int mult(int);
int max_num = 10;
int *buf = (int*)malloc(max_num * sizeof(int));

int main()
{
    for (int i = 0; i < max_num; i++) {
        buf[i] = mult(i);;
        printf("%d\n", buf[i]);
    }
    return 0;
}

int mult(int num)
{
    static int my_num = 5;
    int result = num*2 + my_num;
    return result;
}
```

לפני שתמשיכו לקרוא, חפשו בעצמכם וענו על השאלות הבאות:

- באיזה אזור זיכרון שמורים המשתנים המקומיים שהגדרנו (i, result)?
- באיזה אזור זיכרון נמצא הפרמטר num שמועבר לפונקציה mult?
- כיצד IDA קוראת לפרמטר num? וכיצד IDA מציין שמות של משתנים מקומיים?
- מהו המיקום של הכתובות בזיכרון של המשתנים המקומיים לעומת הפרמטרים? האם זה תמיד חייב להיות כך?

תשובה:

בוודאי מצאתם את כל המשתנים הללו שמורים באזור זיכרון שנקרא stack, המחסנית.

המשתנים שנשלחים לפונקציה נקראים args, והמשתנים בתוך הפונקציה נקראים vars. שימו לב לכתובות בזיכרון של args לעומת ה-vars. כזכור לנו מלימודי האסמבלי, ה-args נדחפים למחסנית בכתובות גבוהות בזיכרון, לפני הכניסה לפונקציה. כאשר הפונקציה מגדירה משתנים מקומיים הם תמיד יהיו בכתובות נמוכות יותר מאשר הפרמטרים שהפונקציה מקבלת.

שלב ה' - סוגי משתנים נוספים

ישנו בקוד הדוגמה משתנה מקומי שהוא static. כזכור, משתנים אלו מיוחדים בכך שערכם נשמר בין קריאות לפונקציה. היכן הייתם מצפים שמשתנה static יישמר?

לבסוף, נסו להבין באיזו כתובת נמצא המערך בגודל עשרה intים שמוגדר על ידי פקודת ה-malloc. דעו מראש, שתצליחו למצוא רק את המיקום בזיכרון של המצביע buf, לא של המערך עצמו. בהמשך נבין מדוע.

תשובה:

משתנה מסוג static אינו יכול להשמר על ה-stack כמו משתנים מקומיים אחרים, מכיוון שהמידע על ה-stack לא נשמר בין קריאות לפונקציה. לכן הדרך היחידה היא לשמור אותו במקום שבו נשמרים משתנים גלובליים, ה-data.

פקודת ה-malloc מקצה זיכרון מתוך ה-Heap. זהו אזור זיכרון שמיועד להקצאות זיכרון בזמן ריצת התוכנית. כדי לדבר על מיקום ה-Heap בזיכרון נפריד בין המצביע על אזור הזיכרון (המשתנה buf) לבין אזור הזיכרון שהוקצה (כלומר הכתובת בזיכרון שהמצביע מצביע עליה).

המצביע לאזור הזיכרון שהגדרנו נמצא באזור ה-data, כיוון שהגדרנו אותו גלובלי, אך הוא גם יכל להיות על המחסנית אם היינו מבצעים את ה-malloc מתוך קוד של פונקציה כלשהי. אך הערך ש-buf מצביע עליו הוא כתובת שנמצאת בתוך אזור זיכרון אחר. זיכרון זה מוקצה בזמן ריצה, לכן לא נוכל לראות אותו ולבדוק את הכתובות שלו באמצעות IDA. זאת כיוון ש-IDA הוא כלי לניתוח סטטי של קוד (ניתוח קוד לא בזמן ריצה). ישנם כלים אחרים, שמאפשרים ניתוח דינמי (בזמן ריצה), כגון x96dbg או WinDbg, אך הם מחוץ להיקף של ספר זה. מה שכן נוכל לעשות, הוא להדפיס את הכתובות של המשתנים בזמן הריצה. בצעו עריכה לקוד של main:

```
int main()
{
    int i;
    for (i = 0; i < max_num; i++) {
        buf[i] = mult(i);;
        printf("%d\n", buf[i]);
    }
    printf("Address of local variable i: %x\n", &i);
    printf("Address of global variable max_num: %x\n", &max_num);
    printf("Address of pointer to array: %x\n", &buf);
    printf("Address of array on heap: %x\n", buf);
    printf("value of 1st element in array: %x\n", *buf);
    return 0;
}
```

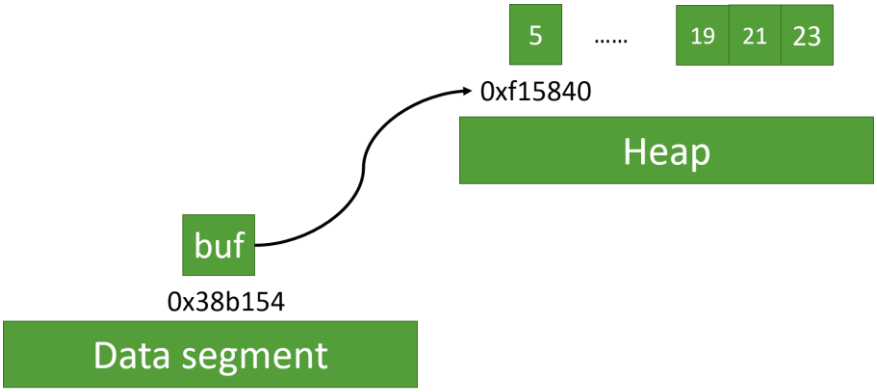
בשלושת ההדפסות הראשונות אנחנו מדפיסים את הכתובות שניתנו למשתנים i, max_num, buf. הכתובת של המשתנה buf, שרשומה בתור &buf, היא כתובת שמאחסנת מצביע לזיכרון, ולא את המערך

עצמו. הכתובת של המערך נמצאת במצביע לזיכרון, כלומר בתוך buf. לטובת שלמות הדוגמה, אפשר לראות גם כיצד ניתן להוציא את הערך הראשון מתוך המערך, וזאת באמצעות buf*. פעולת ה-*, שנקראת dereferencing, ניגשת אל המקום בזיכרון שהמשתנה מצביע עליו.

אם תריצו את התוכנית, תוכלו לראות את הכתובות שהוקצו בזיכרון. כמובן שהכתובות עשויות להשתנות בין הרצה להרצה, כיוון שה-Stack וה-Heap מוקצים ב-RAM בכתובות חדשות כל פעם שהתוכנית נטענת ל-RAM. הרצה לדוגמה:

```
Address of local variable i: cffa44
Address of global variable max_num: 38b000
Address of pointer to array: 38b154
Address of array on heap: f15840
value of 1st element in array: 5
```

נמחיש כיצד המשתנה buf מצביע על הזיכרון שמוקצה ב-Heap:



מה ניתן להסיק מהדמיון והשוני בין הכתובות שמודפסות בזמן ריצה?

המשתנים max_num ו-buf הם משתנים גלובליים. אפשר לראות שאכן הכתובות שלהן בזיכרון דומות למדי- שתיהן מהצורה x38bxxx0. המשתנה i נמצא על המחסנית ולכן המיקום שלו בזיכרון xcffa440, ניכר שנמצא באזור זיכרון אחר. המשתנה buf מצביע על הזיכרון שהקצינו עם malloc ואשר נמצא בכתובת xf158400, השוני בינה לבין הכתובות שנמצאות על ה-data segment ועל ה-stack הוא בולט ומעיד על כך שמדובר באזור זיכרון אחר, שהוא כמובן ה-Heap.

שלב ו' - BSS

אזור ה-BSS משמש להקצאה של זיכרון עבור משתנים לא מאותחלים. נמחיש באמצעות דוגמת קוד. הקוד הבא מגדיר שני מערכים- המערך הראשון אינו מאותחל והמערך השני מאותחל:

```
#include <stdio.h>
#pragma bss_seg("bss")

char uninit_str[10000000];
char init_str[2] = {'A',0};

int main()
{
    uninit_str[0] = 'H';
    uninit_str[1] = 'i';
    uninit_str[2] = 0;
    printf("%s\n%s\n", uninit_str, init_str);
    return 0;
}
```

על סמך מה שלמדנו עד כה, שני המערכים אמורים להיות מוגדרים כמשתנים גלובליים. העניין הוא שאנו מקצים משתנה גלובלי בגודל עצום, 10 מליון בתים, אך לא מכניסים לתוכו ערכים. אין סיבה שקובץ ה-exe שיווצר יכלול 10 מליון בתים שאינם מכילים מידע. הרבה יותר חסכוני לשמור באזור זיכרון מיוחד את המידע על כך שישנו משתנה בגודל 10 מליון בתים, ולהקצות את הזיכרון עצמו רק בזמן הריצה, כאשר צריכים אותו.

הפקודה pragma bss_seg מנחה את הקומפיילר לאסוף את כל המשתנים הלא מאותחלים לתוך אזור זיכרון מיוחד. אם נחקור את קובץ ההרצה באמצעות IDA נראה את ההבדלים בין הכתובות בזיכרון של uninit_str לעומת init_str:

```
push    offset unk_F8C000
push    offset byte_417000
push    offset aSS        ; "%s\n%s\n"
```

ראשי התיבות של BSS מייצגים Block Started by Symbol, אולם בגלל החסכוניות של ה-BSS לעיתים מפרשים את ראשי התיבות כ- Better Save Space ☺

המשתנה uninit_str נמצא בתוך ה-bss, בכתובת x4170000. המשתנה init_str נמצא באזור ה-data וכתובתו xf8c0000. בשורה התחתונה, מה שראינו הוא שלמרות שגם המערך המאותחל וגם המערך שאינו מאותחל הם משתנים גלובליים, הם אינם מוגדרים באותו אזור זיכרון. איך דבר זה מסייע למטרה שלנו, חיסכון בגודלו של קובץ ה-exe? אזור הזיכרון של ה-BSS לא באמת נזקק למקום בקובץ ה-EXE, מכיוון שאין בו מידע. הדבר היחיד שנשמר הוא גודל האזור הזה. כך, קובץ ה-exe נשאר קומפקטי. רק עם הרצת התוכנה מתבצעת הקצאה של מקום בזיכרון עבור המשתנים הלא מאותחלים, בתוך אזור ה-data.

אזורי זיכרון נוספים שלא חקרנו עד כה

קובץ exe עשוי להכיל הגדרות של אזורי זיכרון נוספים. חלק מאזורים מוגדרים על ידי מייקרוסופט לצורך דיבוג או שימושים אחרים. אזור ה-rsrc כולל משאבים שהתוכנה שלנו משתמשת בהם (כפי שאתם זוכרים, בפרק הקודם ראינו איך ליצור משאב אייקון לתוכנה).

מהו idata? על סמך הידע הנוכחי שלנו קשה להסביר את הדברים לעומק. אך היו בטוחים שהדברים יתבררו בהמשך. מבלי להתעכב על יותר מדי פרטים, כאשר אנחנו מתכנתים, כמעט תמיד נרצה להשתמש בפונקציות מתוך ספריות קוד שמיובאות לקוד שלנו בזמן ריצה. לספריות קוד אלו קוראים DLL. לדוגמה רבות מהפונקציות של מערכת ההפעלה מיובאות לקוד שלנו בזמן ריצה באמצעות DLL שנקרא Kernel32.DLL. בהמשך ישנו פרק מיוחד שמדבר על DLLים. אזור ה-idata משמש לצורך שמירת הכתובות של הפונקציות שאנחנו מייבאים, אך אזור זה אינו קיים בקובץ ה-exe מכיוון שהכתובות של הפונקציות החיצוניות עדיין לא ידועות והן נקבעות רק בזמן הטעינה של התוכנית ל-RAM. בהמשך ישנו פרק מיוחד שמוקדש לטעינה של תוכניות ל-RAM. נבין טוב יותר בעתיד.

מהו reloc? גם נושא זה מעט מורכב להסבר כרגע. בתהליך הטעינה של תוכנה ל-RAM נדרש לעיתים לשנות כתובות של פונקציות ומשתנים בזיכרון (דמיינו מצב שבו תוכנית נטענת ל-RAM לכתובות שונות ממה שהיא ציפתה. אם ההמחשה לא ברורה- נבין בדיוק בהמשך).

סיכום אזורי הזיכרון

חיזרו אל תוכנת ה-IDA והקישו על Shift+F7. פעולה זו תעביר אתכם אל מסך שבו מרוכזים כל אזורי הזיכרון שבקוד. הגודל של אזורי הזיכרון צפוי להשתנות בין תוכנית לתוכנית, אולם המבנה של הטבלה נשאר דומה וצפוי להראות כך:

Name	Start	End	R	W	X
.textbss	0000000000401000	0000000000411000	R	W	X
.text	0000000000411000	0000000000417000	R	.	X
bss	0000000000417000	0000000000F89000	R	W	.
.rdata	0000000000F89000	0000000000F8C000	R	.	.
.data	0000000000F8C000	0000000000F8D000	R	W	.
.idata	0000000000F8D000	0000000000F8D1BC	R	.	.

שם הסגמנט

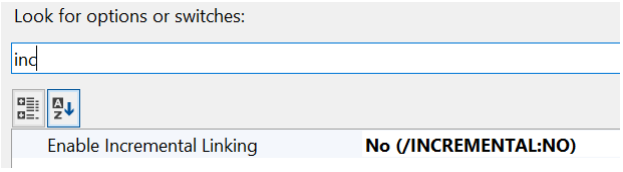
כתובת התחלה ב-RAM

כתובת סיום ב-RAM

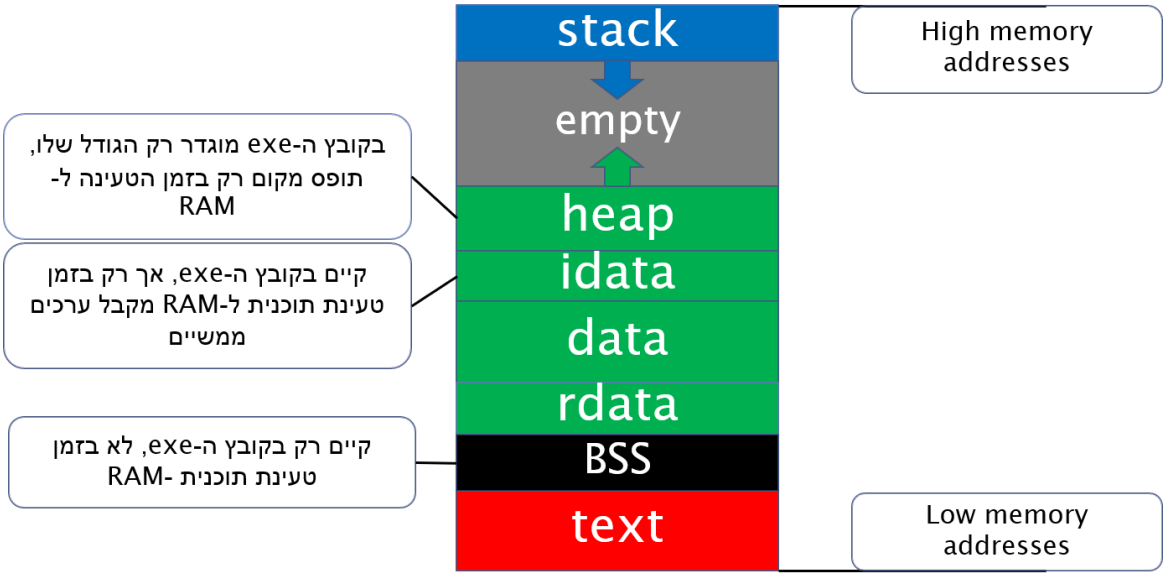
הרשאות (קריאה R, כתיבה W, הרצה X)

אפשר לראות את שמות אזורי הזיכרון, כתובת ההתחלה והסיום שלהם ומהן ההרשאות שלהם. כפי שראינו, ישנן 3 הרשאות – קריאה, כתיבה והרצה. אזור זיכרון יכול לבחור כל צירוף מבין שלושת ההרשאות הללו. כל אזור זיכרון מתחיל בכתובת שהיא מיד לאחר הכתובת האחרונה של אזור הזיכרון שלפניו.

אזור הזיכרון היחיד שלא דנו בו הוא textbss. זהו אזור זיכרון שמשמש את Visual Studio בתהליך ה-linking של התוכנה (חלק מתהליך הפיכת התוכנה לקובץ בשפת מכונה, נלמד עליו בהמשך). אפשר לבטל אותו אם מבטלים את אופציית ה-Incremental Linking בהגדרות הפרוייקט:



נסדר את מבנה אזורי הזיכרון בצורה גרפית:



זהו המבנה הקלאסי מספרי הלימוד של אזורי זיכרון של תוכנה, אך מצאנו אותו לבד באמצעות הכלים שלמדנו ☺

חשוב להבין, ונדון בכך בפירוט בהמשך, שבזמן הטעינה של קובץ EXE ל-RAM משתנים דברים רבים. יש אזור זיכרון ש"נעלם", ה-BSS. יש אזור זיכרון ש"נוצר", ה-Heap. ויש אזור זיכרון שרק בזמן הטעינה ל-RAM מקבל ערכים בעלי משמעות, ה-idata. השינויים בין המצב שקיים על קובץ ה-EXE לבין המצב ב-RAM מפורטים בהערות לצד אזורי הזיכרון שמצאנו.

לסיכום, אנחנו רואים שכל תוכנית בנויה מאזורי זיכרון שונים, אשר לכל אזור יש הרשאות שונות (כתיבה, קריאה, הרצה). אזורי הזיכרון צמודים זה לזה וניתן לבנות "מפה" של הזיכרון. אולם שאלות רבות נותרו עדיין ללא מענה- איך נראה הזיכרון כאשר יותר מתוכנה אחת רצה? האם שתי תוכנות יכולות לגשת לאותו זיכרון? והיכן הזיכרון ששייך למערכת ההפעלה? על שאלות אלו ורבות אחרות נענה בהמשך הלימוד.

מצויידים בידע הזה אנחנו יכולים להתחיל להעמיק את הידע שלנו אל תוך מערכת ההפעלה.

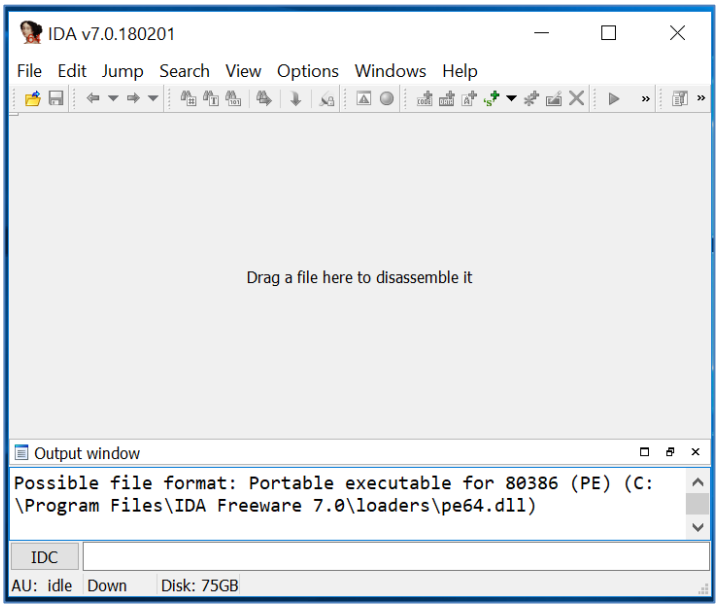
נספח - מדריך בסיסי לעבודה עם IDA

השם IDA הוא שם של כלי מסחרי, קיצור של Interactive DisAssembly. דיסאסמבלי הוא כינוי כללי לתוכנה שיודעת לקחת קוד בשפת מכונה ולהציג אותו בשפת אסמבלי. תוכנת IDA היא מהכלים הנפוצים ביותר בתעשיה ולמרות שהיא עולה כסף רב, לעיתים עשרות אלפי דולרים לרשיון, אפשר להוריד חינם גרסה שלה שמאפשרת לנו את כל מה שאנחנו צריכים בשלב זה.

בחרו את הגרסה העדכנית ביותר של IDA אשר כתוב לידה freeware (המנעו מלהתקין demo בעל תכונות מלאות, אך עם רשיון לזמן מוגבל).

https://www.hex-rays.com/products/ida/support/download.shtml

לאחר שסיימנו את ההתקנה נעלה את IDA:



ה-Disassembly הראשון שלנו

נוריד את הקובץ add.exe מהלינק הבא:

https://data.cyber.org.il/os/add.exe

נגרור את קובץ ה-exe אל החלון (או נשתמש בטאב file->open).

במסך הבא נאמר ל-IDA לטעון את הקובץ שלנו כקובץ PE, קיצור של Portable Executable, הפורמט הנפוץ לקבצי הרצה.

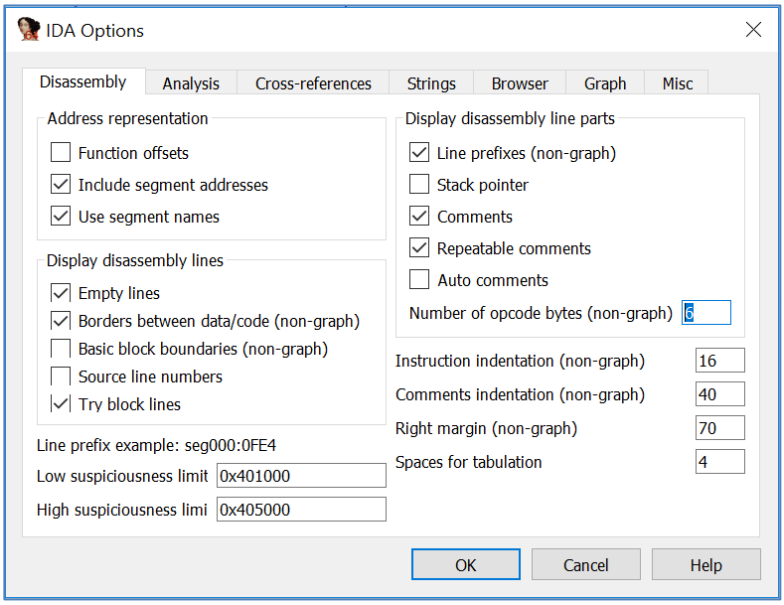
והנה, IDA הצליחה לקחת את הקוד שלנו בשפת מכונה ולתרגם אותו בחזרה לקוד בשפת אסמבלי. אפשר לזהות את אותן הפקודות שכתבנו בקוד המקור:

```
public start
start proc near
mov     eax, 2
mov     ebx, 3
add     eax, ebx
call    sub_404036
push    0               ; uExitCode
call    ds:ExitProcess
start endp
```

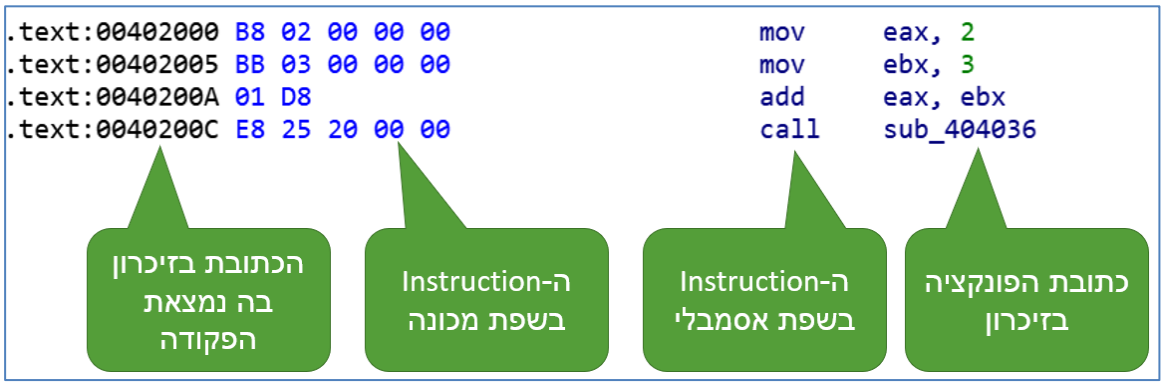
כזכור, המטרה שלנו היתה לצפות בקוד בשפת המכונה. לכן נלחץ על מקש הרווח. מקש הרווח מעביר את התצוגה בין מצב גרפי למצב קוד:

```
.text:00402000                  public start
.text:00402000 start            proc near
.text:00402000                  mov     eax, 2
.text:00402005                  mov     ebx, 3
.text:0040200A                  add     eax, ebx
.text:0040200C                  call    sub_404036
.text:00402011                  push    0               ; uExitCode
.text:00402013                  call    ds:ExitProcess
.text:00402013 start            endp
```

מיד נבין מה אנחנו רואים כאן. ישנו רק דבר אחד נוסף שנרצה לעשות. הכנסו לתפריט Options ובחרו בתוכו את האפשרות General. יופיע לפניכם המסך הבא, בתוכו שנו את הערך של Number of opcode bytes שבמקום שיהיה 0 יהיה 6, כמו בדוגמה הבאה:



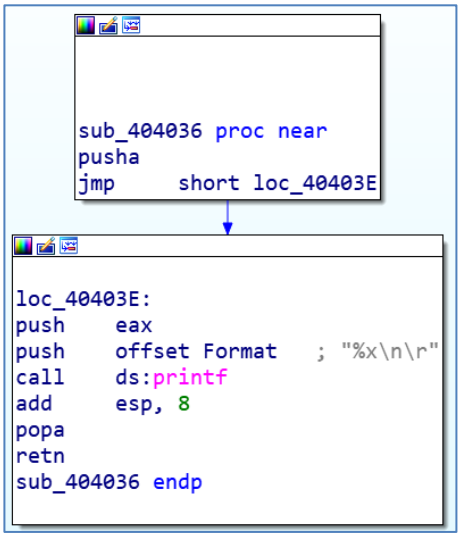
כעת התצוגה היא כפי שרצינו. מה אנחנו רואים?



בצד שמאל, היכן שכתוב text ולידו מספר, זוהי הכתובת בזיכרון בה נמצאת הפקודה. לאחר מכן, אפשר לראות את הפקודות בשפת מכונה ממש (הידד! ☺). שימו לב לדבר הבא: הפקודה הראשונה, המתחילה ברצף B802, גודלה בדיוק 5 בתים. זאת מכיוון שכל ספרה הקסדצימלית מייצגת 4 ביטים, ולכן שתי ספרות הקס שוות לבית אחד. מהו ההפרש בין הכתובת של הפקודה השניה לפקודה הראשונה?

402005 פחות 402000 שווה אכן 5 בתים. כך אנחנו רואים שכל פקודה נמצאת בזיכרון בדיוק לאחר הפקודה הקודמת.

שתי העמודות הבאות נראות בדיוק כמו הקוד בשפת אסמבלי. בקצה השורה האחרונה יש משהו חדש- פקודה שנקראת "call" ולאחריה המילים "sub" ואז רצף ספרות. פקודה זו היא ,כזכור מלימודי האסמבלי, קריאה לפרוצדורה. רצף המספרים הוא בדיוק הכתובת שבה הפרוצדורה שמורה בזיכרון. כדי לראות זאת, לחצו שוב על מקש הרווח כדי לעבור למוד גרפי, הקליקו על התווית "sub_404036" (ייתכן שאצלכם שם התווית יהיה שונה מכיוון שהפרוצדורה נמצאת במקום שונה בזיכרון). כעת קוד הפרוצדורה נמצא לפניכם:



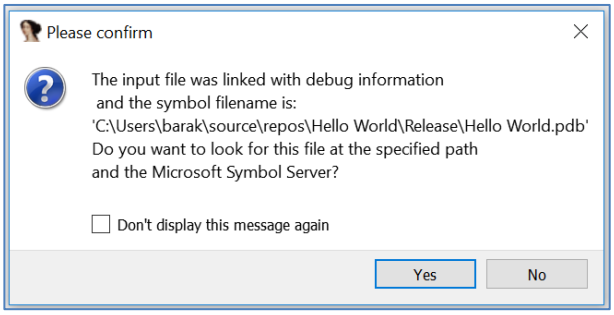
אפשר לראות שבתוך הקוד יש פקודת קפיצה, המיוצגת גרפית על ידי חץ, וכמו כן יש קריאה ל-printf. ואכן, מטרת הקוד הזה היא להדפיס את הערך ששמור בתוך הרגיסטר eax.

מבצעים Disassembly לקוד של עצמנו

בשלב הראשון ניצור קובץ הרצה מסוג EXE. קמפלו את הקוד הבא ב-visual studio, במצב **debug** (חשוב - לא release, הדבר ישנה את המשך התהליך):

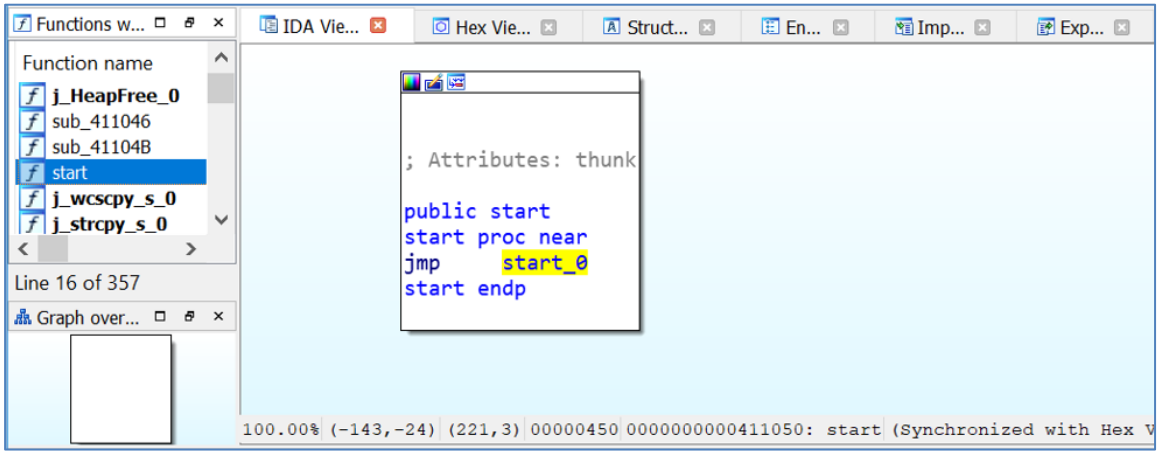
```
#include <stdio.h>

int main()
{
    printf("Hello!");
    return 0;
}
```

לאחר שנטען את קובץ ה-EXE לתוך IDA נקבל הודעה דומה להודעה הבאה:



כעת נבחר באפשרות "**No**". הבחירה הזו משמעותית- IDA מזהה שיש קובץ מסוג pdb עם מידע מהקומפיילר (שכולל לדוגמה שמות של פונקציות). כאשר נקבל קוד שקומפל על ידי מישהו אחר, לא סביר שיהיה לנו קובץ pdb שייקל עלינו, לכן נבחר No.

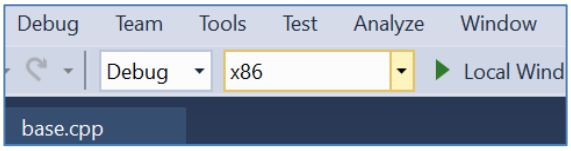
הפתעה! אם ציפינו לכך שריצת התוכנית תחל ב-main, הדבר רחוק מכך. לפני שרץ main מתרחשים דברים רבים. אנחנו נמצאים כרגע במקום שנקרא start. ה-start היא נקודה חשובה מאד, משום שמשם מתחילה ריצת התוכנית שלנו. אם נרצה להגיע אליו שוב- מצד שמאל יש את שמות כל הפונקציות בקוד שלנו, לחיצה כפולה על start תחזיר אותנו לשם. מה גורם לכך שהתכנית מתחילה דווקא מ-start, ואיך אפשר לשנות את זה?התשובה לכך קשורה לדרך שבה קובץ הרצה מאורגן, נלמד על כך בפרק "פורמט PE", אך בינתיים נסתפק בתשובה זו.



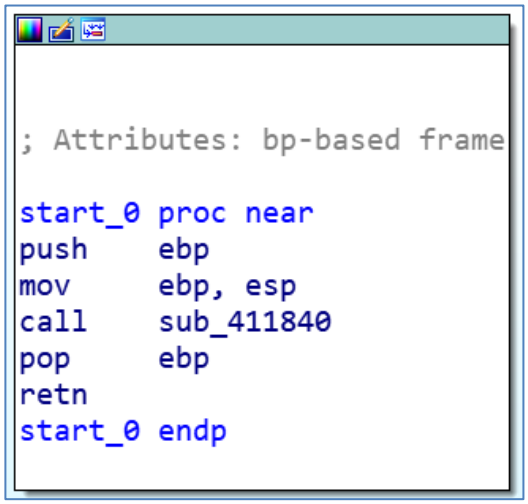
מציאת ה-main

משימתנו הראשונה היא למצוא את ה-main שלנו. לפנינו פקודת אסמבלי, שהיא הפקודה הראשונה שהמעבד מריץ כאשר הוא מריץ את התכנית שלנו, הפקודה jmp start_0. פקודה זו גורמת לקפיצה של הקוד אל עבר המקום בזיכרון שמסומן על ידי התווית start_0. ליתר דיוק, הקפיצה היא שינוי בערכו של הרגיסטר eip, קיצור של Extended Instruction Pointer. רגיסטר זה הוא ההרחבה ל-32 ביט של הרגיסטר IP המוכר לנו מעולם ה-16 ביט (למי שלמד מספר האסמבלי של המרכז לחינוך סייבר).

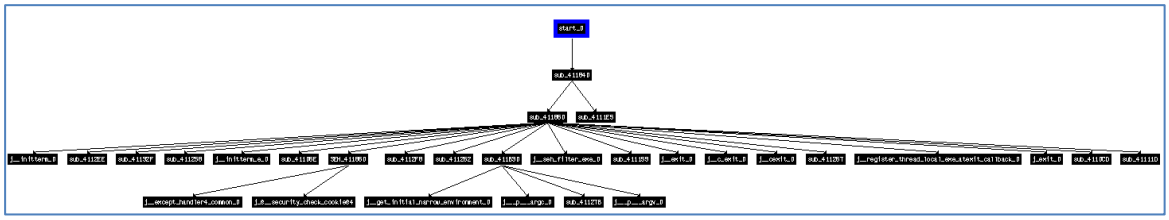
נזכיר, שאנו נמצאים כרגע בעולם ה-32 ביט משום שהגדרות הפרוייקט שלנו ב-Visual Studio היו לקמפל לקוד שתומך ב-32 ביט, שנקרא x86. באותו מסך היינו יכולים לבחור באפשרות x64 ולקבל תמיכה ב-64 ביט:



נחזור לפקודת ה-jmp שלנו: לידה רשומה כתובת בזיכרון. אם נקליק עליה נגיע לכתובת זו בזיכרון. ננסה זאת:



הגענו לכתובת start_0, יש אחריה מספר פקודות. איך כל זה עוזר לנו להגיע ל-main? אם נעשה קליק ימני ונבחר "Xrefs graph from"נקבל את גרף הקריאות לפונקציות בתוכנית:



במגבלות העמוד אי אפשר להראות את שמות הפונקציות, אבל זו לא באמת מגבלה. אפשר לראות שהכל מתחיל למעלה, ב-start שמוקף בכחול. Start קורא לפונקציה, שקוראת לשתי פונקציות והן כבר קוראות לכמות די נכבדה של פונקציות. היכן שהוא בגרף הזה יש את פונקציית ה-main שלנו, אבל היא אינה קרויה main. אם כך, חישבו- איך נמצא אותה?

דרך אחת אפשרית היא היא לעבור על כל הפונקציות בצד שמאל של IDA, היכן שראינו שנמצא start, ולהקליק עליהן אחת אחת עד שנמצא קוד שנראה כמו main. זה אפשרי אבל ארוך ומתיש.

אפשרות אחרת היא לחשוב היכן נכון לחפש את main? באיזה מקום בקוד צפויה להיות קריאה ל-main?

אפשר לצפות שהקריאה תהיה לאחר קריאות לפונקציות של אתחולים ובדיקות, ושאחרי ה-main תיקרא רק פונקציה של יציאה מהתוכנית, משהו שיכלול את השם exit בשם הפונקציה. לכן מה שנעשה זה נתקדם עם הקריאות לפונקציות, בכל פעם שיש לנו הסתעפות בקריאות (פונקציה שקוראת ליותר מפונקציה אחת) נבחר להתקדם עם הפונקציה האחרונה שנקראת, כל עוד היא אינה כוללת בתוכה את השם "exit".

אם כך, נתחיל. הפקודה הראשונה שיש לנו שקוראת לפונקציה היא call sub_411840. כזכור call היא קריאה לפונקציה ואחריה מגיעה תווית שהיא הכתובת של הפונקציה בזכרון. במקרה זה IDA אומר לנו שהכתובת של הפונקציה בזכרון היא 411840.

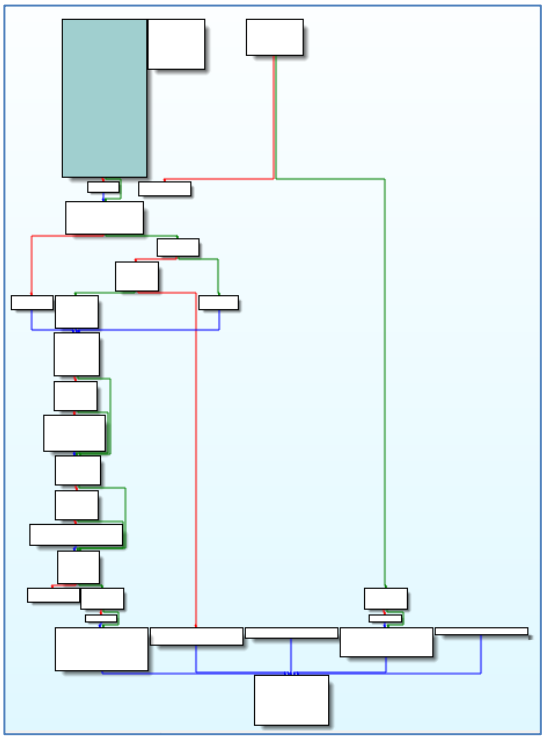
הערה חשובה: הכתובת לא חייבת להיות 411840 כמו בדוגמה. זאת מכיוון שבכל קומפילציה הקומפיילר יכול לבחור כתובת שונה. כמו כן יכול להיות שאצלכם מותקנת גרסה שונה של הקומפיילר. לכן צפוי שיהיו הבדלים מסויימים בקוד זהה שקומפל על ידי שני מחשבים שונים.

נקליק על שם הפונקציה ונעבור אליה:

```
; Attributes: bp-based frame

sub_411840 proc near
push    ebp
mov     ebp, esp
call    sub_4111E5
call    sub_411860
pop     ebp
retn
sub_411840 endp
```

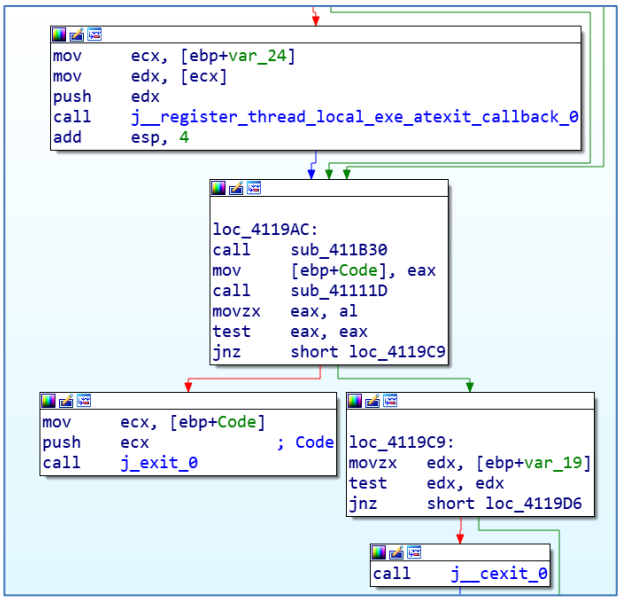
לפנינו הפונקציה שנמצאת בזכרון בכתובת 411840, כפי שניתן לראות בשורה הראשונה. בתוכה יש שתי קריאות לפונקציות. בהתאם לשיטה שלנו, נבחר בנתיב של הפונקציה האחרונה, ונקליק על sub_411860.

נגיע למסך הבא (שכאן מוצג ב-zoom out מקסימלי עקב היקף הקוד הרב):



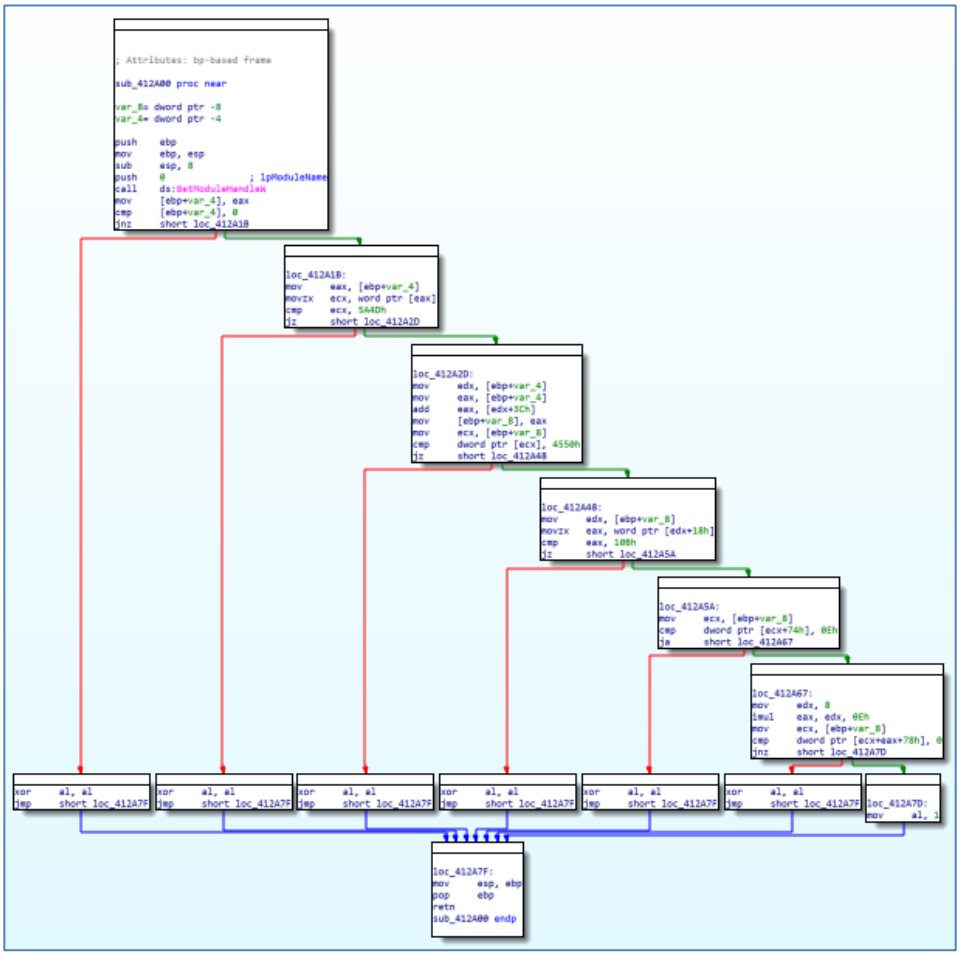
זה אכן נראה לא מעט קוד, אבל אין מה לחשוש- אנחנו לא צריכים לפענח את כולו. רק למצוא את מה שאנחנו מחפשים. כעת נשתמש בהנחת העבודה שלנו שהקריאה ל-main תהיה אחרונה לפני ה-exit. לכן במקום לקרוא את הקוד מלמעלה למטה, נקרא אותו מלמטה למעלה ונחפש פונקציה שנקראת לפני ה-exit.

ואמנם, אם נתמקד בקטע הקוד שנמצא בחלק השמאלי התחתון של הגרף שלנו, נמצא קריאות לפונקציות שמתחילות בשם j_exit:



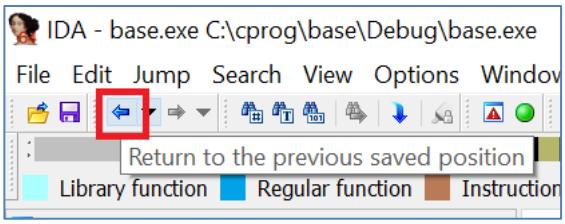
הפונקציה שנקראת אחרונה היא sub_41111D. נקליק עליה ונקפוץ יחד עם ה-jmp אל כתובת A00412.

כעת נקבל קוד שנראה כך:



צורה כזו של קוד אופיינית לאוסף של תנאי if שיש ביניהם and ולאחר מכן else. רק אם כל התנאים מתקיימים אז מתבצע משהו, ואם אחד מהם אינו מתקיים אז קורה דבר מה אחר. אפשר לראות שכל תנאי מוביל אותנו במסלול הירוק אל התנאי הבא, או במסלול האדום הישר אל ה-else. בכל אופן, זה לא נראה כמו ה-main שלנו.

המסקנה היא שהגענו רחוק מדי בקוד שלנו ואנחנו נמצאים בחלק שבו ה-main כבר סיים לרוץ, לכן צריך לחזור אחורה. כדי לחזור אחורה, נשתמש בלחצן החזרה אחורה של IDA, המודגש פה באדום:



נחזור עד לקטע הקוד הבא:

```
loc_4119AC:
call    sub_411B30
mov     [ebp+Code], eax
call    sub_41111D
movzx   eax, al
test    eax, eax
jnz     short loc_4119C9
```

הפעם נבחר ב-sub_411B30. הקלקה עליו תוביל אותנו אל הקוד הבא:

```
; Attributes: bp-based frame

sub_411B30 proc near
push    ebp
mov     ebp, esp
call    j__get_initial_narrow_environment_0
push    eax
call    j___p___argv_0
mov     eax, [eax]
push    eax
call    j___p___argc_0
mov     ecx, [eax]
push    ecx
call    sub_41127B
add     esp, 0Ch
pop     ebp
retn
sub_411B30 endp
```

נפלא. לפנינו קוד שבו רואים קריאה לפונקציה של קבלת משתני סביבה ולאחריה קריאות לפונקציות שמקבלות את argv ו-argc.

הפרמטר argv מחזיק את הערכים שהמשתמש ביקש להעביר לפונקציית ה-main ואילו argc מחזיק את כמות הערכים. אלו דברים שצריכים להתרחש רגע לפני הקריאה ל-main. מיד אחרי השגת הפרמטרים ומשתני הסביבה יש קריאה ל-sub_41127B. לחיצה עליה ודילוג על ה-jmp שבדרך יובילו אותנו אל הקוד הבא:

```
; Attributes: bp-based frame

sub_412320 proc near

var_C0= byte ptr -0C0h

push    ebp
mov     ebp, esp
sub     esp, 0C0h
push    ebx
push    esi
push    edi
lea     edi, [ebp+var_C0]
mov     ecx, 30h
mov     eax, 0CCCCCCCCh
rep stosd
push    offset aHello   ; "Hello!\n"
call    sub_411366
add     esp, 4
xor     eax, eax
pop     edi
pop     esi
pop     ebx
add     esp, 0C0h
cmp     ebp, esp
call    sub_41134D
mov     esp, ebp
pop     ebp
retn
sub_412320 endp
```

זהו, סיימנו את הבסיס שיאפשר לנו לקחת קובץ הרצה, לפתוח אותו ב-IDA, למצוא את ה-main שלו ולהתבונן בקוד האסמבלי שלו.

הדרך הקלה

לאחר שלמדנו כיצד למצוא לבד את ה-main של פונקציה, אפשר לגלות לכם שבמקרים מסויימים אפשר לעבוד בדרך הקלה.

קמפלו את הקוד שלכם במצב Release. במסך הבחירה הבא, בחרו "Yes" על מנת להעלות את קובץ ה-pdb. כיוון שכעת IDA יודע שיש פונקציה בשם main ומה הכתובת שלה, הוא מביא אותנו ישירות ל-main בלי צורך לעבור דרך start. כמו כן, IDA מזהה באמצעות קובץ ה-pdb שהכתובת של הפונקציה שאנחנו קוראים לה היא בעצם printf ולכן מקל עלינו. כעת מסך הפתיחה נראה כך:

```
; int __cdecl main(int argc, const char **argv, const char **envp)
main proc near

argc= dword ptr  4
argv= dword ptr  8
envp= dword ptr  0Ch

push    offset aHello   ; "Hello!\n"
call    printf
add     esp, 4
xor     eax, eax
retn
main endp
```

מצוידים בכלים שקיבלתם, אתם יכולים להמשיך ולחקור קוד. מומלץ להמשיך לקמפל קוד בשפת C ולראות איך התוצאה נראית בזיכרון המחשב.

המשך מחקר מוצלח :)

תיקונים והצעות לשיפור

יש לכם הערות לגבי חומר הלימוד? מצאתם טעות כלשהי או שאתם רוצים להציע משהו עבור גרסאות עתידיות של ספר הלימוד? מוזמנים לשלוח מייל ל-contact@cyber.org.il.

לימוד פורה ומהנה!

ברק גונן

barakg@cyber.org.il

המרכז לחינוך סייבר, 2020