פרק 2

ניתוח סטטי ודינמי

ניתוח סטטי

ניתוח סטטי

- ניתוח סטטי עוסק בניתוח התוכנה מתוך הקוד שלה
  - כפי שהוא מופיע בקובץ ההרצה
  - ללא הרצה של התוכנה
- תוכנות שתומכות בניתוח סטטי כוללות
  - דיס-אסמבלרים – מתרגמים את הקוד לאסמבלי
  - דה-קומפיילרים – מתרגמים לשפה עילית, אם ניתן
  - מפענחי מבני קובץ הרצה
    - מפרטים את המבנה של קובץ ה-PE
- גם מועיל שיש
  - Hex dump, strings וכו'

מה זה דיס-אסמבלר?

- דיס-אסמבלר הוא תכנית, שקוראת קובץ הרצה
  - ממירה את תוכן הקובץ משפת מכונה לאסמבלי
  - מנקודת ראות מ״ה והמעבד הוא סתם תכנית
- דיס-אסמבלר צריך
  - להכיר את מבנה קובץ ההרצה (למשל PE של חלונות)
  - לפרש פקודות אסמבלי
  - לקשר כתובות לשמות פונקציות ולשמות משתנים
    - כולל לקישורים אל DLL-ים
  - זיהוי מיקום פקודות המכונה
    - במעבדים בהם גודל פקודת מכונה אינו קבוע זה לא תמיד פשוט
  - לזהות בין קוד למשתנים, ובין סוגי משתנים שונים
    - למשל להדפיס נכון מחרוזות, שלמים, וכו׳

איך פועל דיס-אסמבלר?

- על פניו דיס-אסמבלר מזהה מיקומי פקודות מכונה באופן דומה לזיהוי על ידי המעבד בזמן ריצה
  - כלומר, פקודה מתחילה בבית שאחרי סוף הפקודה הקודמת
  - או, אם יש פקודת jmp, בבית שאליו ה-jmp מפנה

- דיס-אסמבלר בסיסי מפרש פקודות זו אחר זו
  - על פי סדרן בזיכרון
  - ע"י תרגום פקודה אחר פקודה (Linear Sweep)

איך פותח דיס-אסמבלר?

- לאותו רצף של בתים יכולות להיות כמה משמעויות
  - לכן התרגום האוטומטי לא תמיד יזהה את המשמעות הנכונה
  - ישנם אלגוריתמים ושיטות יותר מתקדמות לזיהוי האסמבלי
  - אחת השיטות היא לזהות קפיצות ולקבוע שיעד הקפיצה חייב להכיל קוד אסמבלי תקין (נקרא Recursive Traversal)
    - איזה שיטות נוספות אפשריות?

<table>
<tr><td>...</td><td>...</td><td>E8</td><td>74</td><td>48</td><td>66</td><td>B8</td><td>48</td><td>EB</td><td>E8</td><td>31</td><td>C0</td><td>...</td><td>...</td></tr>
<tr><td colspan="2">...</td><td colspan="5">call eip+0xb8664874</td><td>dec eax</td><td colspan="2">jmp eip-24</td><td colspan="2">xor eax, eax</td><td colspan="2">...</td></tr>
<tr><td colspan="4">...</td><td colspan="2">jz eip+72</td><td colspan="3">mov ax, 0xeb48</td><td colspan="4">call ...</td></tr>
<tr><td colspan="5">...</td><td>dec eax</td><td colspan="3">mov ax, 0xeb48</td><td colspan="4">call ...</td></tr>
</table>

איך פועל דיס-אסמבלר?

- כשיש תערובת של קוד ונתונים באותו section, זה לא תמיד קל
  - אחרת, שם ה-section יכול לעזור
    - מקובל ש-text. מציין קוד, וש-data. מציין משתנים
- דיס-אסמבלר יכול להשתמש במידע נוסף שקיים בקובץ ההרצה, למשל
  - נתונים מכותרות קובץ ההרצה
    - כגון כתובת תחילת התכנית
  - מידע על הספריות הדינמיות (DLL, ko)
    - כתובות הפונקציות מטבלת ה-export
  - מידע שמיועד לדיבגר
    - אם התכנית הודרה עם אופצית דיבגר
    - כולל שמות משתנים ומיקומם בזיכרון, מספרי שורות, שמות פונקציות

The Interactive Disassembler (IDA)

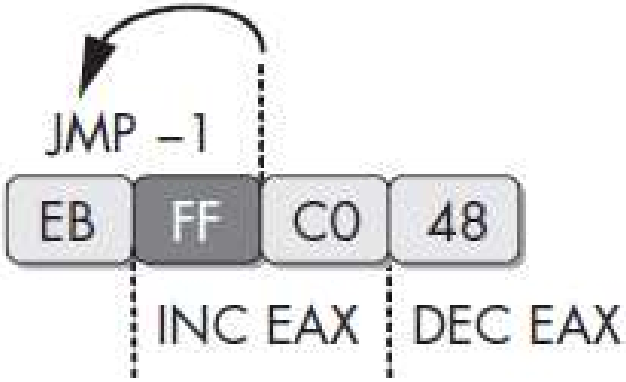
- Ida Pro הוא דיס-אסמבלר אינטראקטיבי
- מיועד ל-RE, עם יכולות דיבוג
- Ida Freeware 5.5 / 7.0
  - גרסה המתאימה לצרכינו
  - תוכנה חופשית
  - הגרסאות האחרות בעלות יכולות נוספות, לא חופשיות
  - אין להשתמש בגרסאות אחרות בקורס זה.
- מאפשרת רישום הערות ומתן שמות לתאי זיכרון
  - למשל משתנים וקוד
  - יכולת מומלצת ביותר
    - מאפשרת לעבוד עם קוד קריא יותר אחרי שזיהינו חלק מרכיביו
    - ואנו נבקש שכך תעשו

אנטי-דיס-אסמבלי

- אנטי-דיס-אסמבלי היא סדרת טכניקות לבלבול דיס-אסמבלר
  - כך שיפרש את הקוד לא נכון
  - או שמקשות עליו לפרש את הקוד

אנטי-דיס-אסמבלי – חפיפת קוד

- מה עושה הקוד הבא?



- זה למעשה (כמעט) פקודת nop בת 4 בתים
  - קפוץ אחורה בית אחד, לקוד חופף (2 בתים : EBFF)
  - הגדל את EAX באחד (2 בתים : FFC0)
  - הקטן את EAX באחד (בית אחד : 48)
    - זה אינו nop בגלל ההשפעה על הדגלים

from Practical Malware Analysis

אנטי-דיס-אסמבלי – ריבוי משמעויות

- מה עושה הקוד הבא?

```
00401328  E8 C7042424   CALL 246417F4
0040132D  3040 00       XOR BYTE PTR DS:[EAX],AL
00401330  E8 A7060000   CALL <JMP.&msvcrt.printf>
00401335  8D7424 1E     LEA ESI,DWORD PTR SS:[ESP+1E]
```

- הוא נראה לא הגיוני – קריאה ל-printf ללא פרמטרים

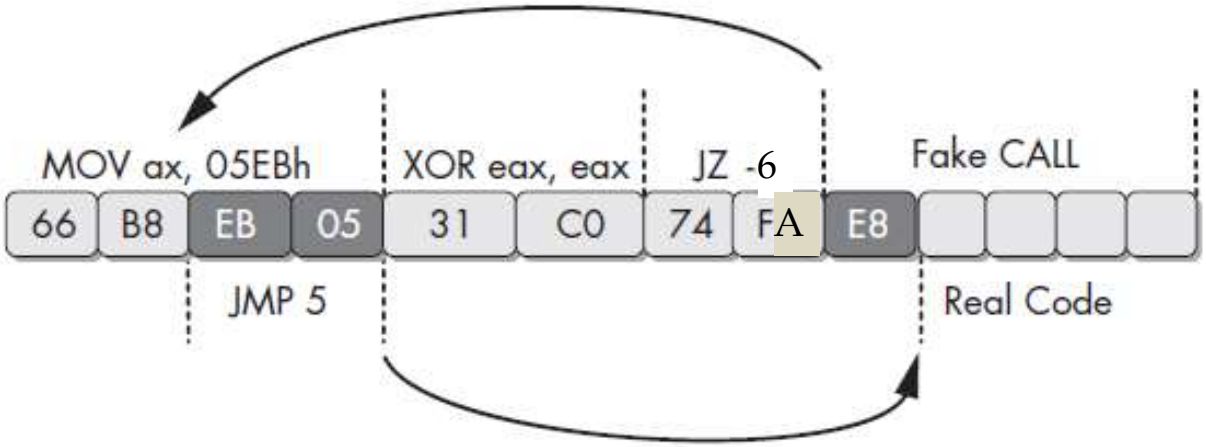
- והקוד הזה?

```
00401329  C70424 24304000  MOV DWORD PTR SS:[ESP],crackV2.00403024  ASCII "Enter the password:"
00401330  E8 A7060000      CALL <JMP.&msvcrt.printf>
00401335  8D7424 1E        LEA ESI,DWORD PTR SS:[ESP+1E]
```

- מדובר באותו הקוד. אבל הפעם ל-printf יש פרמטרים

אנטי-דיס-אסמבלי – חפיפת קוד

- ומה עושה הקוד הזה?



- ה-disassembler לא ידע לתרגם את הפקודות נכון

from Practical Malware Analysis
עם תיקון

אנטי-דיס-אסמבלי – ריבוי משמעויות

- מה עושה הקוד הבא?

```
00401328  E8 C7042424   CALL 246417F4
0040132D  3040 00       XOR BYTE PTR DS:[EAX],AL
00401330  E8 A7060000   CALL <JMP.&msvcrt.printf>
00401335  8D7424 1E     LEA ESI,DWORD PTR SS:[ESP+1E]
```

- הוא נראה לא הגיוני – קריאה ל-printf ללא פרמטרים

- והקוד הזה?

```
00401329  C70424 24304000  MOV DWORD PTR SS:[ESP],crackV2.00403024  ASCII "Enter the password:"
00401330  E8 A7060000      CALL <JMP.&msvcrt.printf>
00401335  8D7424 1E        LEA ESI,DWORD PTR SS:[ESP+1E]
```

- מדובר באותו הקוד. אבל הפעם ל-printf יש פרמטרים

מה תמיד כמה מסובך?

קודי ביניים

- תרגום שפה עילית לקוד ביניים שאינו שפת מכונה
  - קוד הביניים הינו שפה פשוטה
    - בדר"כ ממודלת כמכונת מחסנית אבסטרקטית
    - שקל לפרש אותה ולהריץ אותה, ללא צורך במהדר נוסף
  - בכל מחשב יש תוכנה שמריצה את קוד הביניים
    - באינטרפרטציה, ללא צורך בהידור נוסף
    - בחלק מהמקרים קוד הביניים מתורגם בזמן ריצה לשפת מכונה
      - Just In Time Compiler (JIT)
- שתי דוגמאות עיקריות

שפת קוד ביניים	שפת תכנות	הערות
Java Bytecode	Java	מתוכננת במיוחד ל-Java
CIL (ידועה גם כ-MSIL)	.NET (#C ועוד)	תומכת במגוון שפות

- נתמקד ב-Java Bytecode – בקיצור JB

מבנה בסיסי של שפת Java Bytecode

- שפה מבוססת מחסנית
  - פקודות JB לוקחות פרמטרים מהמחסנית
  - ומחזירות את תוצאתן בראש המחסנית
  - גם המשתנים המקומיים על המחסנית
- ה-opcode תמיד באורך בית אחד
  - יש מקרה בודד של prefix
    - wide – מאפשר אופרנדים בגודל שני בתים בפקודות מסוימות שבדר"כ מקבלות אופרנדים בני בית אחד
- פקודות JB יכולות לקבל מספר קטן של אופרנדים
  - כך שאורך פקודת JB משתנה
  - אופרנדים יכולים להיות בגודל בית אחד או שניים
    - לעיתים גם ארבעה

מבנה בסיסי של שפת Java Bytecode

- JB רץ בתוך מכונה וירטואלית
  - מתוכננת במיוחד להגנה כנגד חולשות
  - למשל, המכונה בודקת חריגה ממערכים
    - מייצרת exception במקרה של חריגה
  - המכונה הווירטואלית מוודאת שקפיצות הן תמיד לתחילת פקודות
    - נבדק על ידי ה-verifier בתחילת ההרצה

- כל כניסה על המחסנית היא בת 32 סיביות
  - שלם, מצביע, וכו'
  - ייצוג של long הוא כשתי כניסות רצופות (64 סיביות)

טכנולוגיית JIT

Just In Time compilation

- הידור קוד בזמן ריצה
- יכולת לבצע אינטרפרטציה לקוד, ולבצע הידור רק לקוד שבשימוש רב
  - וללמוד מהאינטרפרטציה איך כדאי לבצע אופטימיזציות
- מאפשר לוודא שהקוד המהודר אינו מתוכנן לגרום נזק
  - קוד הביניים פשוט יותר לבדיקה
  - הקוד המהודר אינו ניתן לטיפול ע"י תוקף
  - כלומר, נוזקה אינה יכולה להשתלט על הקוד...

מבנה קובץ הרצה ב-Java

- קובץ הרצה jar הינו ארכיון ZIP
  - כלומר ניתן לפתוח אותו עם WinZip
- קובץ jar כולל
  - קובץ ראשי
    - META-INF/MANIFEST.MF
    - מתאר היכן התכנית הראשית, מיקום ספריות, וכו'
  - קבצי מידע כללי
    - author.txt
  - וקבצי class
    - הכוללים Bytecode
    - program.class, lib.class, ...
- ייצוגים פנימיים תמיד ב-big-endian

מבנה קובץ Class

- קובץ class כולל
  - תיאור התוכן הפומבי של המחלקה
  - רשימת קבועים (Constant pool)
    - כוללת את כל הקבועים וסוגם
    - כלומר
      - הגדרות המשתנים (בפרט הפומביים של המחלקה), עם הפניות לטיפוסים
        - ✓ ההפניה היא לאיבר אחר ברשימה
      - מחרוזות
        - ✓ מחרוזות שמופיעות בתכנית
        - ✓ מחרוזות של שמות מחלקות ושמות פונקציות שבשימוש
      - כל הטיפוסים שיש בתכנית
    - מבנה של ענפים של עץ
  - הקוד (bytecode) של כל הפונקציות של המחלקה
    - בשפה דמוית אסמבלי

קצת על המבנה של קובץ Class

- רשימת הקבועים
  - הרשומות מסודרות לפי סדר
    - ללא אינדקס מפורש בפנים
  - כל רשומה מתחילה בבית המתאר טיפוס
    - ומידע באורך קבוע
    - במקרה של UTF-8 – אורך משתנה
      - לאחר הטיפוס, אורך בשני בתים ואז המחרוזת
  - כלומר קל לשנות תוכן של רשומה
    - כולל לשנות אורך
    - בלי צורך לשנות דבר ברשומות אחרות
    - כל עוד לא מוחקים ולא מוסיפים רשומות באמצע

- JB
  - קפיצות נעשות באופן יחסי למיקום תחילת הפקודה
  - שינוי קוד אינו דורש תיקון בקפיצות שאינן קופצות מעל השינוי
  - עדיין עדיף לשנות בלי לשנות אורך

טיפוסים בסיסיים בשפת Java

- להלן רשימת הטיפוסים הבסיסיים בשפת Java וקיצוריהם
  - הקיצורים משמשים בקובץ ה-class

	טיפוס	קיצור
signed	integer	I
	long	L
	short	S
	byte	B
Unicode (16 bit)	character	C
	float	F
	double	D
	boolean	Z
	reference	A

  - למערך יש תוספת "[" לפני האות
  - אין טיפוסי unsigned int/long/byte...

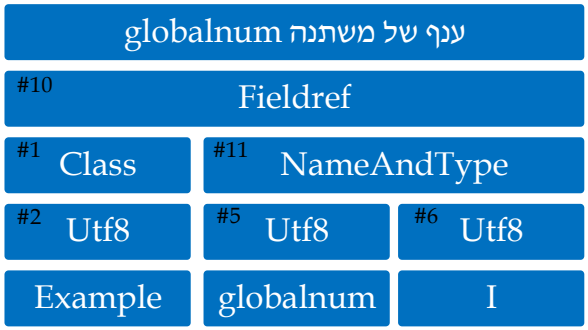
מבנה רשימת הקבועים

- רשימת הקבועים (Constant Pool) היא מערך
  - עם הפניות בצורת ענפים של עץ

- כל כניסה במערך מכילה
  - אינדקס (מאחד עד מספר הקבועים)
  - טיפוס
    - Utf8 : טקסט מקודד ב-UTF-8
      - משמש לכל הטיפוסים האחרים
      - קידוד Utf8 אינו קידוד UTF-8 תקני
        - ✓ NUL מקודד בשני בתים, ותווי UTF-8 ארוכים מקודדים שונה מבתקן
    - String : מחרוזת, בצירוף האינדקס של הטקסט UTF-8 שלה
    - Class : מחלקה, בצירוף האינדקס של הטקסט UTF-8 של השם שלה

מבנה רשימת הקבועים

- NameAndType : שם וטיפוס, עם שני אינדקסים ל-UTF-8, האחד לשם המזהה, והשני לקיצור שם הטיפוס (אות אחת) או טיפוס פונקציה
- Methodref : מתודה, עם אינדקס של מחלקה ואינדקס NameAndType
- Fieldref : שם שדה או שם משתנה, עם אינדקס של מחלקה ואינדקס NameAndType
- Integer
- Float
- Long
- ...



דוגמת תכנית

```
public class Example
{
    public static int globalnum=5;

    public static void main (String args[])
    {
      int num=3+globalnum;

      if(globalnum<10) { num++; };

      System.out.println("The sum is "+num);
    }
}
```

Constant pool

```
#1 = Class              #2                  // Example
#2 = Utf8               Example
#3 = Class              #4                  // java/lang/Object
#4 = Utf8               java/lang/Object
#5 = Utf8               globalnum
#6 = Utf8               I
#7 = Utf8               <clinit>
#8 = Utf8               ()V
#9 = Utf8               Code
#10 = Fieldref          #1.#11              // Example.globalnum:I
#11 = NameAndType       #5:#6               // globalnum:I
#12 = Utf8              LineNumberTable
#13 = Utf8              <init>
#14 = Methodref         #3.#15              // java/lang/Object."<init>":()V
#15 = NameAndType       #13:#8              // "<init>":()V
#16 = Utf8              main
#17 = Utf8              ([Ljava/lang/String;)V
#18 = Fieldref          #19.#21             // java/lang/System.out:Ljava/io/PrintStream;
#19 = Class             #20                 // java/lang/System
#20 = Utf8              java/lang/System
#21 = NameAndType       #22:#23             // out:Ljava/io/PrintStream;
#22 = Utf8              out
#23 = Utf8              Ljava/io/PrintStream;
#24 = Class             #25                 // java/lang/StringBuilder
#25 = Utf8              java/lang/StringBuilder
#26 = String            #27                 // The sum is
#27 = Utf8              The sum is
#28 = Methodref         #24.#29             // java/lang/StringBuilder."<init>":(Ljava/lang/String;)V
#29 = NameAndType       #13:#30             // "<init>":(Ljava/lang/String;)V
#30 = Utf8              (Ljava/lang/String;)V
#31 = Methodref         #24.#32             // java/lang/StringBuilder.append:(I)Ljava/lang/StringBuilder;
#32 = NameAndType       #33:#34             // append:(I)Ljava/lang/StringBuilder;
#33 = Utf8              append
#34 = Utf8              (I)Ljava/lang/StringBuilder;
#35 = Methodref         #24.#36             // java/lang/StringBuilder.toString:()Ljava/lang/String;
#36 = NameAndType       #37:#38             // toString:()Ljava/lang/String;
#37 = Utf8              toString
#38 = Utf8              ()Ljava/lang/String;
#39 = Methodref         #40.#42             // java/io/PrintStream.println:(Ljava/lang/String;)V
#40 = Class             #41                 // java/io/PrintStream
#41 = Utf8              java/io/PrintStream
#42 = NameAndType       #43:#30             // println:(Ljava/lang/String;)V
#43 = Utf8              println
#44 = Utf8              SourceFile
#45 = Utf8              Example.java
```

המשתנה globalnum
מופנה ע"י #10

המחרוזת "The sum is "
מופנה ע"י #26

אינדקס | טיפוס | ערך או הפניה | הערות

פלט של הרצת javap על קובץ המחלקה

דוגמת JB של התכנית

```
static {};                                  // Constructor
  stack=1, locals=0, args_size=0
     0: iconst_5
     1: putstatic      #10                  // Field globalnum:I
     4: return

public static void main(java.lang.String[]);
   stack=4, locals=2, args_size=1
     0: iconst_3                            // Short form for   iconst  3 (single byte instead of 3)
     1: getstatic      #10                  // Field globalnum:I
     4: iadd
     5: istore_1                                    // Short form for   istore  1 (single byte instead of 3)
     6: getstatic      #10                  // Field globalnum:I
     9: bipush         10
    11: if_icmpge      17
    14: iinc           1, 1
    17: getstatic      #18                  // Field java/lang/System.out:Ljava/io/PrintStream;
    20: new            #24                  // class java/lang/StringBuilder
    23: dup
    24: ldc            #26                  // String The sum is
    26: invokespecial  #28                  // Method java/lang/StringBuilder."<init>":(Ljava/lang/String;)V
    29: iload_1
    30: invokevirtual  #31                  // Method java/lang/StringBuilder.append:(I)Ljava/lang/StringBuilder;
    33: invokevirtual  #35                  // Method java/lang/StringBuilder.toString:()Ljava/lang/String;
    36: invokevirtual  #39                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
    39: return
```

כתובת

opcode

אופרנדים

הערות

דיס-אסמבלרים ודה-קומפיילרים

- קיימים מגוון של דיס-אסמבלרים ודה-קומפיילרים
  - הלוקחים קוד JB ומתרגמים לקוד קריא

- למשל
  - דיס-אסמבלרים
    - javap
    - גם גרסאות מסוימות של IDA
  - דה-קומפיילרים
    - JD
    - JAD
    - Krakatau

- בדר"כ מייצרים קוד די טוב

דה-קומפילציה ע"י JD מקובץ ה-Class

```
import java.io.PrintStream;

public class Example
{
  public static int globalnum = 5;

  public static void main(String[] paramArrayOfString)
  {
    int i = 3 + globalnum;

    if (globalnum < 10) i++;

    System.out.println("The sum is " + i);
  }
}
```

Java bytecode לe Obfuscation

- העלמת מחרוזות מרשימת הקבועים
  - למשל יצורן בקוד
- שינוי שמות משתנים ומתודות
- ניקוי נתוני דיבוג
- וכמובן, סיבוך הקוד

- יש מגוון תוכנות obfuscation ייעודיות ל-Java

מודל האבטחה ב-Java

- תמיכה במגוון מבנים מאובטחים ע"י שפת התכנות
  - Strong data typing
  - ניהול זיכרון אוטומטי (ושחרור אוטומטי)
  - אין cast ללא בדיקה
  - אין אריתמטיקה של מצביעים
  - מערכים שמורים בזיכרון הדינמי
    - מוקצים בזמן ריצה (ע"י new), לא על המחסנית

- מכונה וירטואלית שבודקת כל מה שאפשר
  - מגוון בדיקות בזמן טעינת מחלקה
    - בפרט, Bytecode verification
      - למשל, כל הקפיצות למיקומים חוקיים (לא לאמצע פקודות JB)
  - בדיקות בזמן ריצה
    - למשל בדיקות חריגה ממערך
    - בדיקות שימוש במצביע בעל ערך NULL

מודל האבטחה ב-Java

- sandbox
  - מאפשר קביעת הרשאות לקוד שרץ
  - כלומר, לא כל ההרשאות של מריץ הקוד מאופשרות
    - למשל, מניעת גישה לקבצים, גישה לרשת או גישה לשרתים מסוימים ברשת, יציאה מהמכונה הווירטואלית, יצירת תהליך חדש, וכו'
- JIT
  - תוקף לא יכול לשתול קוד כרצונו
- לסיכום, מניעת בעיות אבטחה מקוד עוין על ידי מניעת היכולת של הקוד לבצע פעולות בעייתיות
  - ברמת שפת התכנות, המכונה הווירטואלית, וקריאות לפעולות מסוימות
  - אולם, ההגנות לא מושלמות
    - בשנים האחרונות התגלו מגוון בעיות אבטחה ב-Java שגרמו להפסקת השימוש ב-Java במספר ארכיטקטורות

מודל האבטחה ב-Java

- מה לא נעשה?
  - אין בדיקת integer overflow
- כל שמות המשתנים ברמת המחלקה רשומים ברשימת הקבועים
  - כך שהם נגישים לכל מי שמעוניין לבצע RE
  - בפרט נגישים לדה-קומפיילרים
- ואפילו מספרי השורות כלולים בקובץ...
  - במקור לצרכי דיבוג

הבדלים עיקריים בין JB ל-CIL

- CIL תוכננה על סמך העקרונות של JB

- JB תוכננה במיוחד עבור Java
- CIL תוכננה לתמוך במגוון שפות
  - לכן כוללת מגוון גדול יותר של טיפוסים ומבנים

- המבנה העקרוני של השפות דומה
  - שתיהן מבוססות מחסנית
  - ואפילו עם פקודות דומות רבות

- אבל CIL למדה מהניסיון של JB
  - עם שיפורים
  - קוד הביניים תוכנן ל-JIT
    - JB תוכננה לתמוך באינטרפרטציה וב-JIT

Java באנדרואיד

- באנדרואיד הפעלת Java מורכבת יותר
  - כדי לאפשר את יתרונות Java מצד אחד ופעולה יעילה מצד שני
- עד אנדרואיד 4.4 – שימוש ב-Dalvik
  - אפליקציה נשמרת בקובץ APK שבתוכו נשמר קובץ DEX
    - Dalvik executable
    - קוד ביניים ייעודי שיותר קל לבצע לו JIT
    - נוצר ע"י המפתח מקבצי ה-class וה-JAR בזמן יצירת האפליקציה
  - האפליקציה מורצת כ-JIT ע"י הידור ה-DEX בזמן ריצה
- אנדרואיד 5 – שימוש ב-ART
  - Android run time
  - המרת ה-APK לקובץ הרצה בזמן התקנת האפליקציה ובשדרוג מ"ה
    - אין שינוי במבנה קובץ ה-APK
  - האפליקציה מופעלת מקובץ ההרצה, ללא צורך בשום הידור נוסף
- אנדרואיד 6 – ART משולב JIT
  - כדי לחסוך בזמן התקנה ובזמן שדרוג מ"ה
  - ההמרה לקובץ הרצה נעשית בזמן פנוי של המערכת
  - עד אז מבוצע JIT

ניתוח דינמי

ניתוח דינאמי

- ניתוח דינמי מפעיל את התוכנה הנחקרת, ועוקב אחרי פעולתה
- לצורך כך, תהליך אחר, הנקרא דיבגר, שולט על התהליך הנחקר
  - קובע מתי יעשה מה
  - יכול לעצור ולחדש את הפעולה
  - עוקב ויכול לשנות את כמעט כל משאב של התהליך המדובג
- בשונה מניתוח סטטי, הניתוח נעשה בזמן ריצה
  - ולא על קובץ ההרצה ללא ריצה
  - לכן, ניתן לבחון נתונים שאינם זמינים בניתוח סטטי

מה זה דיבאגר?

- Debugger מאפשר למשתמש
  - הרצת הקוד
    - תוך כדי בדיקת האוגרים והזיכרון של התהליך המדובג
  - שינוי כל אחד מהערכים האלו
    - כולל של הקוד עצמו
  - עצירת ריצת התוכנית (breakpoints) לפי מספר שיטות
    - נרחיב בהמשך
  - לעיתים גם פירוש מבנים מסוימים בזיכרון
    - למשל ניתוח מבנה קובץ ההרצה

כלים נפוצים

- כלים נפוצים בחלונות:
  - Ollydbg
  - Immunity Debugger
  - Windbg

מדבגים תהליך ב-user mode בלבד

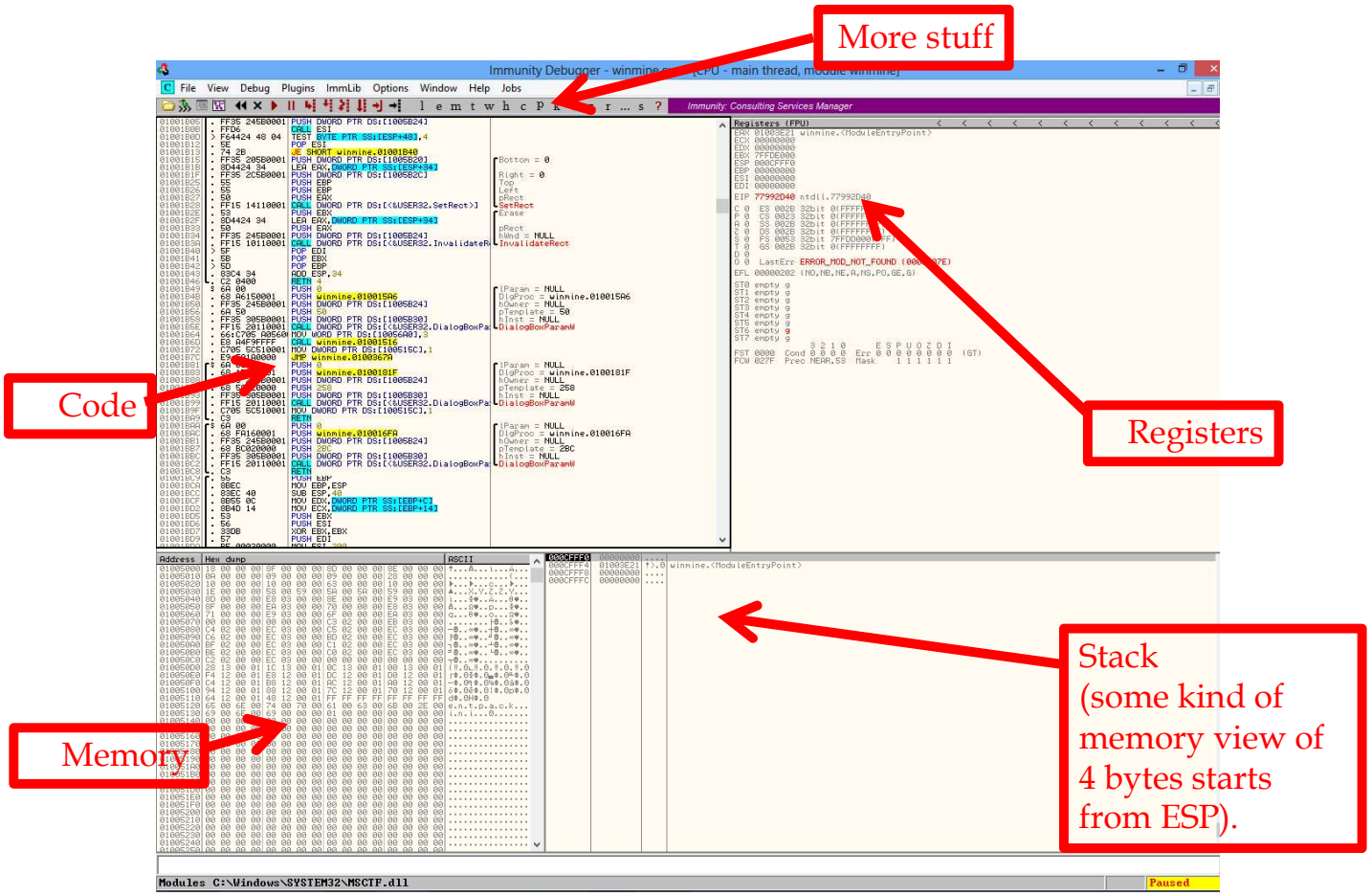
מדבג הכל, הכלי היחידי שמדבג kernel בחלונות

- לינוקס: gdb

דיבוג גרעין מ"ה

- דיבוג גרעין מ"ה (kernel) אינו יכול להיעשות כמו דיבוג תהליך
  - כי הוא לא תהליך
  - אסור לעצור את פעולת הגרעין באמצע ביצוע
    - ואין משמעות להחלפת הקשר של גרעין מ"ה
  - הכלים שמ"ה מספקת לדיבגר לא קיימים במקרה זה
- לכן דיבוג של גרעין מ"ה דורש כלים יעודיים
- לא נעסוק בכך בקורס זה

Ollydbg/IMM


More stuff
Code
Registers
Memory
Stack
(some kind of
memory view of
4 bytes starts
from ESP).

איך פועל דיבאגר?

- דיבגר הוא תהליך, השולט על פעולת תהליך אחר
- מנקודת ראות מ״ה הוא
  - יכול להריץ אותו
  - לעצור את פעולתו
    - כולל לעצור את פעולתו באופן מתוכנן בפקודה מסוימת
  - לשלוט על הזיכרון שלו
    - לקרא ולשנות את תוכן הזיכרון שלו
    - כולל טבלת הדפים שלו
  - לנהל את הטיפול בחריגות של התהליך המדובג
  - לדעת את מצבו ומצב כל הרגיסטרים בכל עצירה
    - ולשנות אותם לצורך המשך הרצה
- המעבד ומ״ה מתוכננים לתמוך בכך

תמיכת המעבד בדיבאג

- דיבגר משתמש בשירותי המעבד על מנת לדבג תהליך
- לשם כך יש
  - פקודות מכונה ופסיקת תוכנה מיוחדות
    - int 3
  - מבחר exceptions יעודיים עבור breakpoints
  - רגיסטרים מיוחדים
  - צורות הפעלה יעודיות
    - למשל single step
  - ושימוש בתכונות אחרות הנתמכות ע״י המעבד
    - למשל זיכרון וירטואלי

תמיכת מ"ה בדיבאגר

- דיבגר משתמש בשירותי מערכת ההפעלה על מנת לדבג תהליך
- מ"ה מתייחסת שונה לתהליך מדובג, למשל
  - מדווחת ל-Debugger על אירועים שונים (events) שרלוונטיים לתהליך המדובג
  - בעת חריגה, מ"ה מעבירה את הטיפול ל-Debugger, ועוצרת את פעולת התוכנית
  - אם התוכנית לא מסוגלת לטפל בחריגה, מ"ה תיתן הזדמנות שנייה ל-Debugger לפני הקרסת התוכנית

- כאשר מדבגים תהליך, הוא עלול להתנהג בצורה שונה
- שיטות Anti-Debugging מנצלות זאת

מבנה הנתונים של דיבוג

- מבנה הנתונים שמ״ה שולחת לתהליך המדבג:

```
typedef struct _DEBUG_EVENT {
    DWORD dwDebugEventCode;     // מציין מה סוג האירוע בגללו נעצרה התכנית
    DWORD dwProcessId;          // process id (pid)
    DWORD dwThreadId;           // thread id in process
    union {                     // מידע נוסף לפי סוג האירוע
        EXCEPTION_DEBUG_INFO      Exception;
                // This includes exception info, e.g., which exception was
                // raised, in case of a page fault: which address failed, etc.
        CREATE_THREAD_DEBUG_INFO  CreateThread;
        CREATE_PROCESS_DEBUG_INFO CreateProcessInfo;
        EXIT_THREAD_DEBUG_INFO    ExitThread;
        EXIT_PROCESS_DEBUG_INFO   ExitProcess;
        LOAD_DLL_DEBUG_INFO       LoadDll;
        UNLOAD_DLL_DEBUG_INFO     UnloadDll;
        OUTPUT_DEBUG_STRING_INFO  DebugString;
        RIP_INFO                  RipInfo;
    } u;
} DEBUG_EVENT, *LPDEBUG_EVENT;
```

דוגמא למבנה של דיבאג

```
// Create the process
main () {
    ...
    CreateProcess (..., DEBUG_PROCESS , ...) ;
    // Or
    DebugActiveProcess( dwProcessId );
    ...
    EnterDebugLoop(...);
    ...
}
```

דוגמא למבנה של דיבאג

```
// Example based on MSDN

void EnterDebugLoop(const LPDEBUG_EVENT DebugEv)
{
   DWORD dwContinueStatus = DBG_CONTINUE;
                                          // exception continuation

   for(;;)
   {
      WaitForDebugEvent(DebugEv, INFINITE);
```

דוגמא למבנה של דיבאג

```
// Process the debugging event code.
    switch (DebugEv->dwDebugEventCode)
    {
      case EXCEPTION_DEBUG_EVENT:
      // Process the exception code. When handling
      // exceptions, remember to set the continuation
      // status parameter (dwContinueStatus). This value
      // is used by the ContinueDebugEvent function.

        switch(DebugEv->u.Exception.ExceptionRecord.ExceptionCode)
        {
```

דוגמא למבנה של דיבאג

```
    case EXCEPTION_ACCESS_VIOLATION:
            // u.Exception.ExceptionRecord.ExceptionInformation is an
            // array that contains the address of the memory that
            // could not be accessed, as well as why (read, write, ...)
      break;
    case EXCEPTION_BREAKPOINT:
      break;
    case EXCEPTION_DATATYPE_MISALIGNMENT:
      break;
    case EXCEPTION_SINGLE_STEP:
      break;
    case DBG_CONTROL_C:
      break;
    default:
      break;
  }
  break;
```

דוגמא למבנה של דיבאג

```
case CREATE_THREAD_DEBUG_EVENT:
   dwContinueStatus = OnCreateThreadDebugEvent(DebugEv);
   break;
case CREATE_PROCESS_DEBUG_EVENT:
   dwContinueStatus = OnCreateProcessDebugEvent(DebugEv);
   break;
case EXIT_THREAD_DEBUG_EVENT:
   dwContinueStatus = OnExitThreadDebugEvent(DebugEv);
   break;
case EXIT_PROCESS_DEBUG_EVENT:
   dwContinueStatus = OnExitProcessDebugEvent(DebugEv);
   break;
case LOAD_DLL_DEBUG_EVENT:
   dwContinueStatus = OnLoadDllDebugEvent(DebugEv);
   break;
```

דוגמא למבנה של דיבאג

```
        case UNLOAD_DLL_DEBUG_EVENT:
          dwContinueStatus = OnUnloadDllDebugEvent(DebugEv);
          break;
        case OUTPUT_DEBUG_STRING_EVENT:
          dwContinueStatus = OnOutputDebugStringEvent(DebugEv);
          break;
        case RIP_EVENT:
          dwContinueStatus = OnRipEvent(DebugEv);
          break;
      }
      // Resume executing the thread that reported the debugging event.
      ContinueDebugEvent(DebugEv->dwProcessId,
                         DebugEv->dwThreadId, dwContinueStatus);
    }
}
```

העברת החריגה לדיבאגר

היישום במערכת ההפעלה

(from reactos.org)

KiDispatchException

```
 if (PreviousMode == KernelMode)  { … }
else
   {
      if (FirstChance)      /* User mode exception, was it first-chance? */
      {
         /* Break into the kernel debugger unless a user mode debugger is present or user mode */
         /* exceptions are ignored, except if this is a debug service which we must always pass to KD */
         if ((!(PsGetCurrentProcess()->DebugPort) &&
             !(KdIgnoreUmExceptions)) | |
             (KdIsThisAKdTrap(ExceptionRecord, &Context, PreviousMode)))
         {
            KiPrepareUserDebugData();          /* Make sure the debugger can access debug directories */
            /* Call the kernel debugger */
          if (KiDebugRoutine(TrapFrame, ExceptionFrame, ExceptionRecord, &Context, PreviousMode, FALSE))
               goto Handled;                          /* Exception was handled */
         }
          /* Forward exception to user mode debugger */
          if (DbgkForwardException(ExceptionRecord, TRUE, FALSE))
               return;
         KiDispatchExceptionToUser()
         __debugbreak();
      }
```

שיטות לעצירת ריצה בדיבאגר

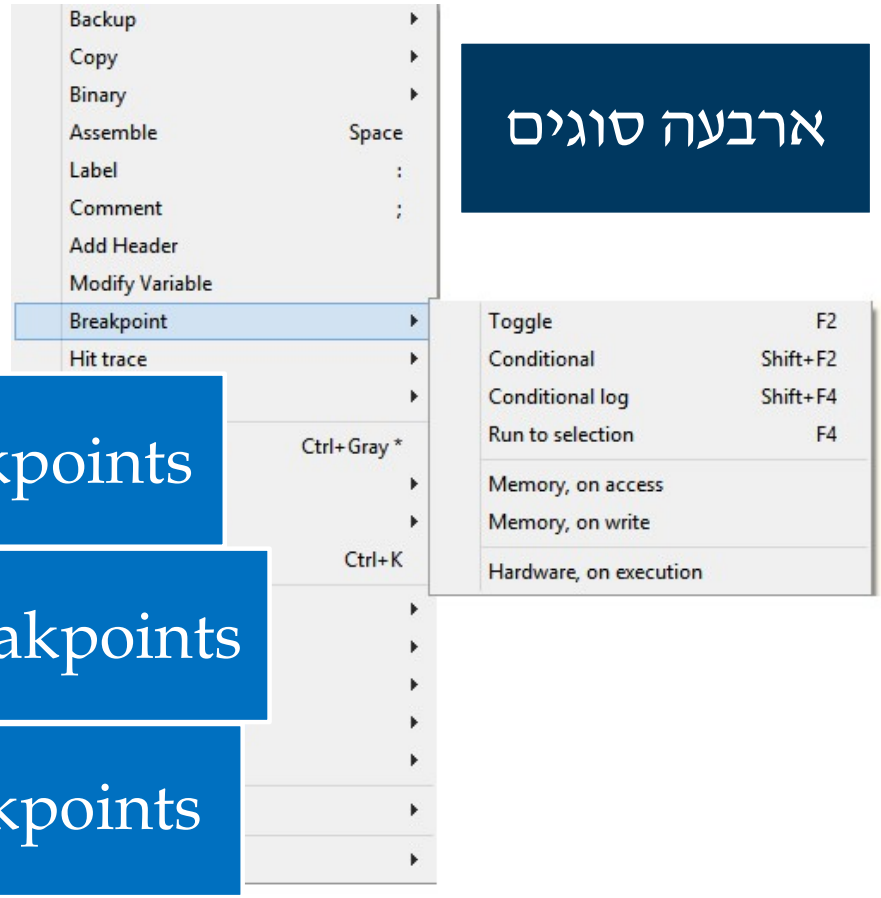
ארבעה סוגים

Single step

Memory breakpoints

Hardware breakpoints

Software breakpoints



שימושים

סוג	עצירה בקריאה מזיכרון	עצירה בכתיבה לזיכרון	עצירה בהרצת קוד
Single Step			☑
Memory BPT	☑	☑	☑
Hardware BPT	☑	☑	☑
Software BPT			☑

Single Step משמש גם ככלי עזר בישומי Memory BPT ו-Software BPT.

Single Step

- המעבד תומך בהפעלת פקודת מכונה בודדת בתהליך מדובג, על פי בקשת הדיבגר
- התהליך מופעל לפקודת מכונה אחת, ועוצר
- התהליך המדבג חוזר לפעולה ומחליט מה לעשות הלאה
  - הוא גם מקבל מידע על המצב של התהליך המדובג
  - למשל אם הייתה חריגה
  - וכמובן ערכי הרגיסטרים, וכו'
- Single Step שימושי לצורך
  - הרצה פקודה בודדת (פקודת Step של הדיבגר)
  - ולשליטה על הפעולה של שירותי דיבגר אחרים
    - בהם הדיבגר צריך לשנות את סביבת ההרצה לפקודת מכונה אחת
      - למשל Memory BPT ו-Software BPT

Memory Breakpoints

- ניתן להגדיר בתים בזיכרון שיגרמו לחריגה אם
  - היה ניסיון לקרוא אותם או לכתוב אליהם
  - היה ניסיון להריץ קוד שכתוב בהם
- ממומש באמצעות שינוי הרשאות הקריאה/כתיבה/ריצה של דף הזיכרון
  - באמצעות שינוי בטבלת הדפים של התהליך
  - גורם ל-EXCEPTION_GUARD_PAGE בזמן ריצה
- ה-breakpoint תקף לכל דף הזיכרון
  - כלומר ל-4096 בתים (תלוי בהגדרות כמובן)
  - הדיבגר צריך לבדוק לאן פקודת המכונה ניגשה בדף הזיכרון
    - ואם זה לא לאחד הבתים אחריהם עוקבים – אז הוא מתעלם
    - לצורך זה הוא מפרש בעצמו את פקודת המכונה בה קרתה החריגה
      - ולהבין מדוע קרתה ובאיזה גישה לזיכרון
        - ✓ מיקום קוד התכנית, קריאה או כתיבה לזיכרון, וכו'
- יתרון : אין הגבלה על מספר ה-memory breakpoints

Memory Breakpoints

- לאחר העצירה, לא ניתן להמשיך ריצה בלי תיקון ההרשאות בטבלת הדפים
- לשם כך מופעל הנוהל הבא
  - תיקון הרשאות הדף
  - הרצת התכנית ב-Single Step
    - כלומר הפעלת פקודת מכונה בודדת
      - זה נתמך בחומרה של המעבד (TF flag)
      - תמיכת המעבד הכרחית! אחרת אי אפשר להבטיח ריצת כל פקודה עם עצירה אחריה
    - יתכן שהתכנית תיעצר שוב באותה פקודת מכונה מסיבה אחרת
      - דורש תיקון כל הסיבות לפני הצלחת הפקודה
  - שיחזור הרשאות הדף לצורך המשך הדיבוג
  - המשך הרצת התכנית עד ה-breakpoint הבא
- כל זה נעשה גם אם עצרנו בגלל גישה לבית בזיכרון באותו הדף שאינו דורש מעקב
  - אבל, בלי שהדיבגר מציג את העצירה הזו למשתמש

Hardware Breakpoints

- ניתן להגדיר מספר מוגבל של בתים בזיכרון שיגרמו לחריגה אם
  - היה ניסיון לקרוא אותם או לכתוב אליהם
  - היה ניסיון להריץ קוד שכתוב בהם
- זה נעשה בעזרת אוגרים מיוחדים (DR0-DR7)
  - המעבד בודק את ערך האוגרים לפני כל גישה לתא בזיכרון
  - חפשו "debug registers" למידע נוסף על האוגרים
- ניתן לבצע את הפעולה רק על BYTE/WORD/DWORD
- חסם של ארבעה hardware breakpoints לכל היותר
- השיטה שקופה לתהליך המדובג
  - וגם יעילה יותר מהאחרות

DR0-DR3	• ארבע כתובות לעצירה
DR4-DR5	• מבוטלים
DR6	• סטטוס
DR7	• הפעלה ושליטה

Software Breakpoints

- מיושמים על ידי שינוי בקוד התכנית
  - שתילת פקודת המכונה int 3 במקום בו רוצים לעצור
  - זהה ל-BPT ב-PDP
- כשמגיעים לנקודת ה-breakpoint, פקודת int 3 מקפיצה חריגה שמועברת ל-Debugger
  - (בהנחה שהתהליך מדובג)
  - כדי להמשיך את הריצה, יש צורך
    - לשחזר את פקודת המכונה המקורית שהייתה שם
    - להריץ ב-single step
    - ואז להחזיר את פקודת int 3
      - כדי שבפעם הבאה שנגיע שוב תהיה עצירה

Software Breakpoints

- למה יש ל-int 3 אופקוד CC בן בית אחד?
  - בעוד ניתן היה להשתמש באופקוד CD 03?
  - כלומר איזה בעיות גורם אופקוד בן שני בתים?

- הדיבגר ממשיך להציג "נכון" את התרגום לאסמבלי
  - למרות ששינה את מקום ה-breakpoint ל-CC
  - כלומר לא מציג את ה-CC
  - אלא את הפקודה המקורית שהיתה שם

int 3 – שני ייצוגים	
CC	CD 03

Software Breakpoints–נקודות

```
EIP->  xor eax,eax
       mov ebx,eax
       push esi
       mov esi , 200
       inc ebx
       xor edi , edi
       cmp edx , esi
       ja label1
       je label2
       mov eax,300
       cmp edx,eax
       sub eax,11
       and eax,FF0
       cmp eax,FF1
```

Software Breakpoints–דוגמא

```
EIP->   xor eax,eax
        mov ebx,eax
        push esi
        mov esi , 200
        inc ebx
        xor edi , edi
        cmp edx , esi
        int 3          // put int 3. remember the overwritten opcodes.
        je label2
        mov eax,300
        cmp edx,eax
        sub eax,11
        and eax,FF0
        cmp eax,FF1
```

Software Breakpoints–נקודות

```
        xor eax,eax
        mov ebx,eax
        push esi
        mov esi , 200
        inc ebx
        xor edi , edi
        cmp edx , esi
EIP->   int 3
        je label2
        mov eax,300
        cmp edx,eax
        sub eax,11
        and eax,FF0
        cmp eax,FF1
```

Exception,
The debugger will handle it.

Before continuing execution, the debugger will restore the original opcode, use single step, and after executing the original opcode will put int 3 back.

Software Breakpoints–נקודות

```
        xor eax,eax
        mov ebx,eax
        push esi
        mov esi , 200
        inc ebx
        xor edi , edi
        cmp edx , esi
EIP->   ja label1
        je label2
        mov eax,300
        cmp edx,eax
        sub eax,11
        and eax,FF0
        cmp eax,FF1
```

Exception,
The debugger will handle it.

Before continuing execution, the debugger will restore the original opcode, use single step, and after executing the original opcode will put int 3 back.

חסרונות של Software Breakpoints

- מתערבים בקוד של תהליך
- לא יעבוד במקרה שתהליך לגיטימי ישנה את הקוד
  - נדיר, אבל קורה למשל ב-NET.
- תהליכים פחות לגיטימיים יכולים לסרוק את קוד התוכנה ולאתר את ה-breakpoint
  - לכן עדיף להשתמש ב-Hardware BP בניתוח נוזקות ותוכנות חשודות אחרות
- בעת ניתוח קוד שמשנה את עצמו, פקודת ה-int 3 עלולה להימחק...
  - למשל כאשר וירוס דחוס פותח את עצמו ומשנה את המקום בו עשינו breakpoint
  - אין בעיה כזו ב-hardware breakpoints – הם יעצרו כשנגיע למקום הזה בקוד החדש

שאלות?


esp, 8
ecx,
ecx
offset
decode
esp,