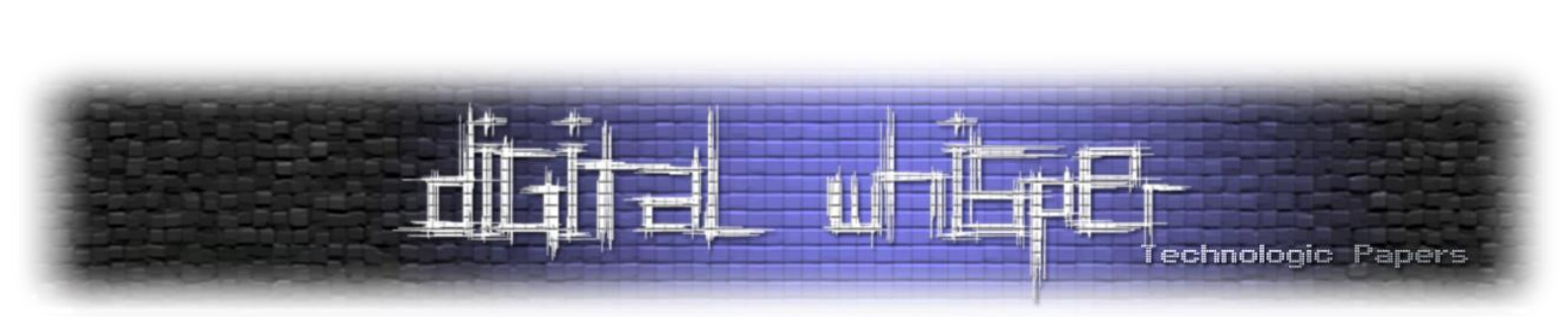


עמוק בקרביים של Windows: שימוש ב-Paging ו-Segmentation כבסיס לאבטחה

מאת גיא פרגל

הקדמה

במאמר זה אציג את הקונספטים המרכזיים: Paging ו-Segmentation. מערכת ההפעלה עושה שימוש בשני הפיצ'רים המעניינים האלו שנמצאים במעבד, בכדי לספק את האבטחה ברמה הבסיסית ביותר, אשר רובנו התרגלנו לקבל אותה כמובנת מאליה.

במהלך המאמר אנו נתמקד במימוש של הקונספטים במעבדי x86 ובמערכת ההפעלה Windows 7 x86, וכמובן שניתן מספר דוגמאות להמחשה באמצעות WinDBG.

לפני שנתחיל, חשוב לציין שהמאמר אינו מיועד למתחילים בתחום, ורצוי ידע מקדים בנושאים - אסמבלי, היכרות עם אכיטקטורת x86, והיכרות עם שפת C.

זו הפעם הראשונה שאני כותב מאמר, לכן אשמח לקבל מכם תיקונים / הערות /רעיונות לשיפור וכו' בתיבת הדוא"ל - 0xGuppy@gmail.com

רקע

אז נתחיל מזה שנתאר מה בדיוק הציפיות שלנו ממערכת הפעלה מודרנית כשאנחנו מדברים על אבטחה. לצורך המאמר, יש לנו שלוש ציפיות מרכזיות:

1. אל תאפשרי לתוכנית לגשת או לערוך מידע של תוכנית אחרת - במיוחד במערכת הפעלה מרובת משימות (Multitasking).
2. אל תאפשרי לתוכנית לגשת או לערוך מידע השייך למערכת ההפעלה.
3. אל תאפשרי לתוכנית לגשת למשאבי חומרה באופן ישיר (כגון: מקלדת, מעכבר, כונן קשיח, משגר טילי נרף וכו').

מערכת ההפעלה לא יכולה לעמוד בשלושת הדרישות האלו לבדה והיא חייבת עזרה של ה-CPU כדי שיבקר, יאכוף ויאפשר את קיומה של האבטחה (אלא אם כן, אתם מדמים CPU כמו ש-JAVA עושה עם JVM). מערכות הפעלה מודרניות כגון Windows ו-Linux עושות שימוש ב-2 פיצ'רים בכדי לממש את האבטחה, אותם נסקור במהלך המאמר - **Paging** ו-**Segmentation**. שניהם יושבים בתוך יחידת ניהול הזיכרון במעבד (MMU).

אנחנו נתחיל מהסבר מקוצר של כל פיצ'ר ולבסוף נחבר בין כל הנקודות ונסביר כיצד הם נותנים מענה לשלושת הצרכים שהצגנו קודם לכן.

מקטעי זיכרון (Segmentation)

את הזיכרון הראשי נוהגים לחלק באינטל למספר מקטעים, או בשמם - סיגמנטים.

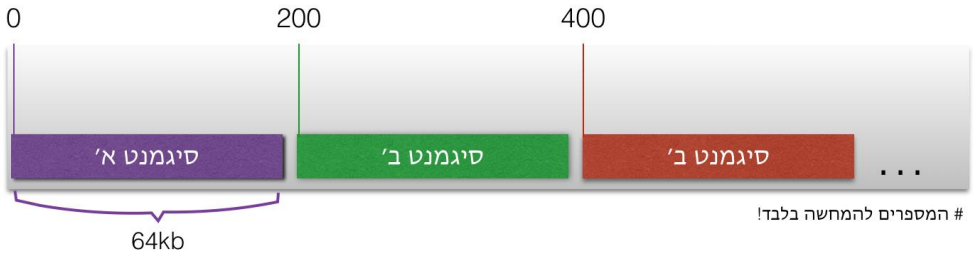
הסיבה לכך כרוכה בהיסטוריה של מעבדי ה-8086 הראשונים אשר הכילו אוגרים בגודל של כ-16 בתים, כשאורך הפקודות היו 8 או 16 בתים בלבד. עובדה זו אפשרה למעבד לגשת רק ל-16^2 מקומות בזכרון, כלומר 64kb.

המהנדסים באינטל רצו לאפשר למעבד לגשת ליותר מקומות בזיכרון, אך עם זאת לא לשנות את גודל האוגרים והפקודות.

כפתרון לבעיה הציגו המהנדסים אוגרים חדשים - אוגרי המקטע (segment registers), אשר תפקידם להצביע על הבלוק של ה-64kb בו אנו מעוניינים להסתכל באותו הרגע. כתובות הזיכרון בהם השתמשנו קודם לכן הפכו ל-offset מאותו המצביע לסגמנט וכך נפתרה הבעיה.

מאז ועד היום הסגמנטציה נשארה והיא תמיד מופעלת במעבדי x86/64 מרגע עליית המחשב.

דוגמא להמחשה:
נגיד וגודל האוגר מאפשר לנו לגשת רק ל-200 כתובות זיכרון. במידה ונרצה לגשת לתא זיכרון בכתובת 250 הנמצא בסגמנט ב' יתבצע החישוב:



כתובת הבסיס של הסגמנט (200) + הכתובת המבוקשת (50) = הכתובת ה"ליניארית" (250).

מקטעי הזיכרון הבסיסיים הינם:

- code segment - בו נמצא קוד התוכנית
- stack segment - האזור בזיכרון המוקצה למחסנית
- data segment - אזור השמור למידע סטאטי (קבועים, תמונות וכו')
- extra segment - המשחק תפקיד זהה ל-ds

פקודות הנוגעות למהלך ריצת התוכנית כמו Jump, ילקחו מהסגמנט בו נמצא קוד התוכנית, פקודות הנוגעות למחסנית כמו Push ילקחו מהסגמנט בו נמצאת המחסנית וכו' (ניתן גם "לכפות" את הסגמנט בו המעבד יעשה שימוש באמצעות far pointers, אבל לא נדון בזה במסגרת המאמר).

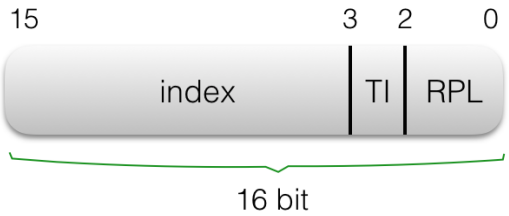
עבור כל מקטע / סגמנט קיים אוגר באורך 16 ביט המייצג אותו. אלו בדיוק אותם אוגרים שהתרגלנו לראות בדיבאגרים:

```
ES  002B    DS  002B
CS  0023    SS  002B
```

היות והסגמנטציה תמיד מופעלת, שיטת הייצוג משתנה בהתאם למצב בו פועל המעבד - Real Mode או Protected Mode. במאמר זה לא נרחיב על Real Mode.

ובכל זאת, למסוקרנים: ב-Real Mode הביטים באוגר המקטע מייצגים את הכתובת הפיזית ממנה מתחיל הסיגמנט. עכשיו יש כמה אפשרויות - אפשר לומר שה-16 ביטים הללו הם ה-"most significant bit" אליהם נחבר את 16 הביטים של ה-offset. כך למעשה נצליח למפות 4GB של זכרון! אך באינטל החליטו שלא לעשות כך, היות וזה ידרוש עוד פינים פיזיים במעבד מה שיגיל עלויות. לכן - השיטה אשר נשארה עד היום היא להכפיל את אוגר המקטע ב-16 וחבר אליו את ה-offset. בדיוק מסיבה זו, לא ניתן לעשות שימוש ביותר מ- 1MB במצב זה.

כשמערכת ההפעלה עלתה, והמעבד נמצא במצב Protected Mode, אוגרי המקטע מהווים למעשה מצביעים לטבלה בה נמצאות רשומות המתארות מקטע:



כל אוגר מקטע (segment selector) מורכב מהפורמט הבא:

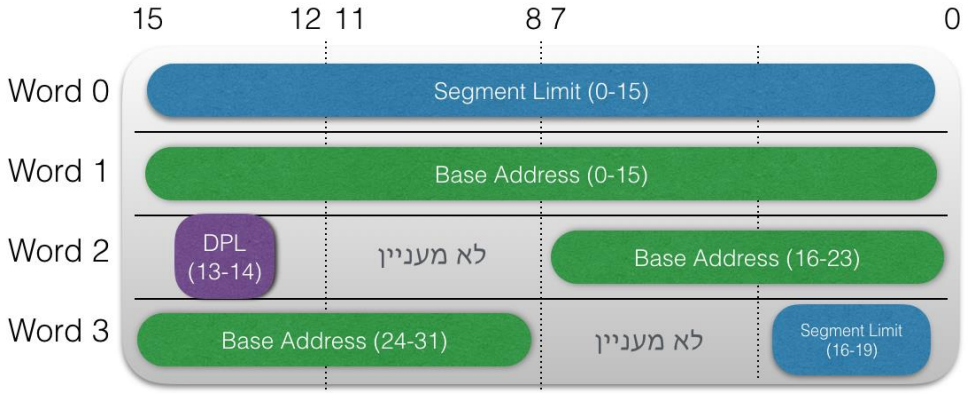
- **RPL (Requested Privilege Level)** - רמת ההרשאות המתבקשת המיוצגת ע"י המספר 0 או 3, כאשר 0 הינה רמת ההרשאות הגבוהה ביותר (kernel Mode), ו-3 היא רמת ההרשאות הנמוכה ביותר (user Mode).

- **index** - הינו מספר הרשומה בטבלה הנקראת בשם - Descriptor Table. קיימות 2 סוגים של טבלאות - הטבלה הגלובאלית - Global Descriptor Table (GDT) והטבלה המקומית Local Descriptor Table (LDT). מהשמות נגזר כי הטבלה הגלובאלית משמשת את כלל התוכניות במערכת בעוד הטבלה המקומית משמשת תוכנית ספציפית.

- **TI** - באיזו טבלה לבחור. 0 עבור GDT ו-1 עבור LDT.

ויש יוצא מן הכלל: הסגמנט cs, המכיל את קוד התוכנית, הינו יוצא דופן ובמקום הערך RPL הוא מכיל את הערך CPL (קיצור של Current Privilege Level). כלומר, רמת ההרשאות של הקוד שרץ כרגע. כעקרון, RPL ו-CPL הינם זהים במהותם (במסגרת המאמר), ולכן נתייחס מעכשיו רק ל-CPL.

LDT איננה רלוונטית לנו כרגע ולכן נתמקד ב-GDT. טבלה זו היא בעצם מערך, בו כל רשומה הינה בת 8 בתים ומייצגת סגמנט אחד. רשומה זו מכונה בשם segment descriptor וזהו המבנה שלה:



נתמקד בערכים העיקריים:

- **Base address** - 32 ביט המייצגים את הכתובת ה"ליניארית" של תחילת הסגמנט.
- **Limit** - 20 ביט המייצגים את גודל הסגמנט.
- **DPL (descriptor privilege level)** - מספר המייצג את ההרשאה הנדרשת בכדי לגשת לסגמנט, כאשר 0 מייצג את רמת ההרשאות הגבוהה ביותר (kernel Mode) ו-3 מייצג את רמת ההרשאות הנמוכה ביותר (user Mode). שימו לב שאופן מימוש זה נכון למעבדי x86/64 בלבד.

אז אחרי שהבנו מה זה סגמנט, למה הוא נוצר, ומה משמעות האוגרים, נשארה רק בעיה אחת לא פתורה: איך המעבד יודע איפה ממוקמת הטבלה GDT?

למעשה בכל CPU/ Core קיים אוגר בשם gdtr, המכיל את הכתובת הליניארית בו נמצא הבייט הראשון בטבלה.

ננסה לעבור את התהליך בעצמנו באמצעות Kernel Debugger. אנו נשתמש בפקודות הבאות ב-WinDBG:

- **r** - תציג לנו את תוכן האוגרים
- **.formats** - נשתמש בפקודה זו כדי לראות את הייצוג הבינארי של ערכים.
- **db** - יציג לנו בלוק זיכרון בכתובת שנבחר.

נתחיל:

1. לצורך הדוגמא נתמקד בסגמנט cs. בואו נציג ונבין את ערכו:

```
kd> r cs
cs=00000008
kd> .formats 8
Evaluate expression:
  Hex:     00000008
  Decimal: 8
  Octal:   00000000010
  Binary:  00000000 00000000 00000000 00001000
  Chars:   ....
  Time:    Thu Jan  1 02:00:08 1970
  Float:   low 1.12104e-044 high 0
  Double:  3.95253e-323
```

רמת ההרשאות של הסגמנט (CPL) - 00. כלומר kernel Mode, שהיא רמת ההרשאות הגבוהה ביותר.

הטבלה בה הסגמנט נמצא (TI) - 0. כלומר הסגמנט נמצא בטבלה - GDT.

האינדקס בטבלת ה-GDT - 1. בדיוק כמו שזה נשמע.

2. כעת נצטרך לאתר איפה ממוקמת טבלת הGDT באמצעות האוגר - GDTR:

```
kd> r gdtr
gdtr=80b95000
```

3. אם אמרנו שכל ערך בטבלה הינו 8 בתים ומייצג Segment Descriptor, בואו נשלוף את הערך אינדקס מספר 1 מהטבלה:

```
kd> db 80b95000
80b95000  00 00 00 00 00 00 00 00-ff ff 00 00 00 9b cf 00  ................
80b95010  ff ff 00 00 00 93 cf 00-ff ff 00 00 00 fa cf 00  ................
80b95020  ff ff 00 00 00 f3 cf 00-ab 20 00 b0 1d 8b 00 80  ......... ......
80b95030  48 37 00 9c 96 93 40 82-ff 0f 00 00 00 f2 40 00  H7....@.......@.
80b95040  ff ff 00 04 00 f2 00 00-00 00 00 00 00 00 00 00  ................
80b95050  68 00 00 70 96 89 00 82-68 00 68 70 96 89 00 82  h..p....h.hp....
80b95060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
80b95070  ff 03 00 50 b9 92 00 80-00 00 00 00 00 00 00 00  ...P............
```

אם נכניס את 8 הבתים למבנה של ה-Segment Descriptor שתיארנו קודם לכן נגלה ש:

- Base Address = 00 00 00 00
- Limit = ff ff f
- DPL = 0

כעת, אחרי שהבנו את כל התהליך, אני מרגיש יותר בנוח לגלות לכם שקיימת פקודה שעושה את כל מה שעשינו הרגע באופן אוטומטי והיא - dg:

```
kd> dg cs
                                    P Si Gr Pr Lo
Sel    Base     Limit     Type     l ze an es ng Flags
---- -------- -------- ---------- - -- -- -- -- --------
0008 00000000 fffffff Code RE Ac 0 Bg Pg P  Nl 00000c9b
```

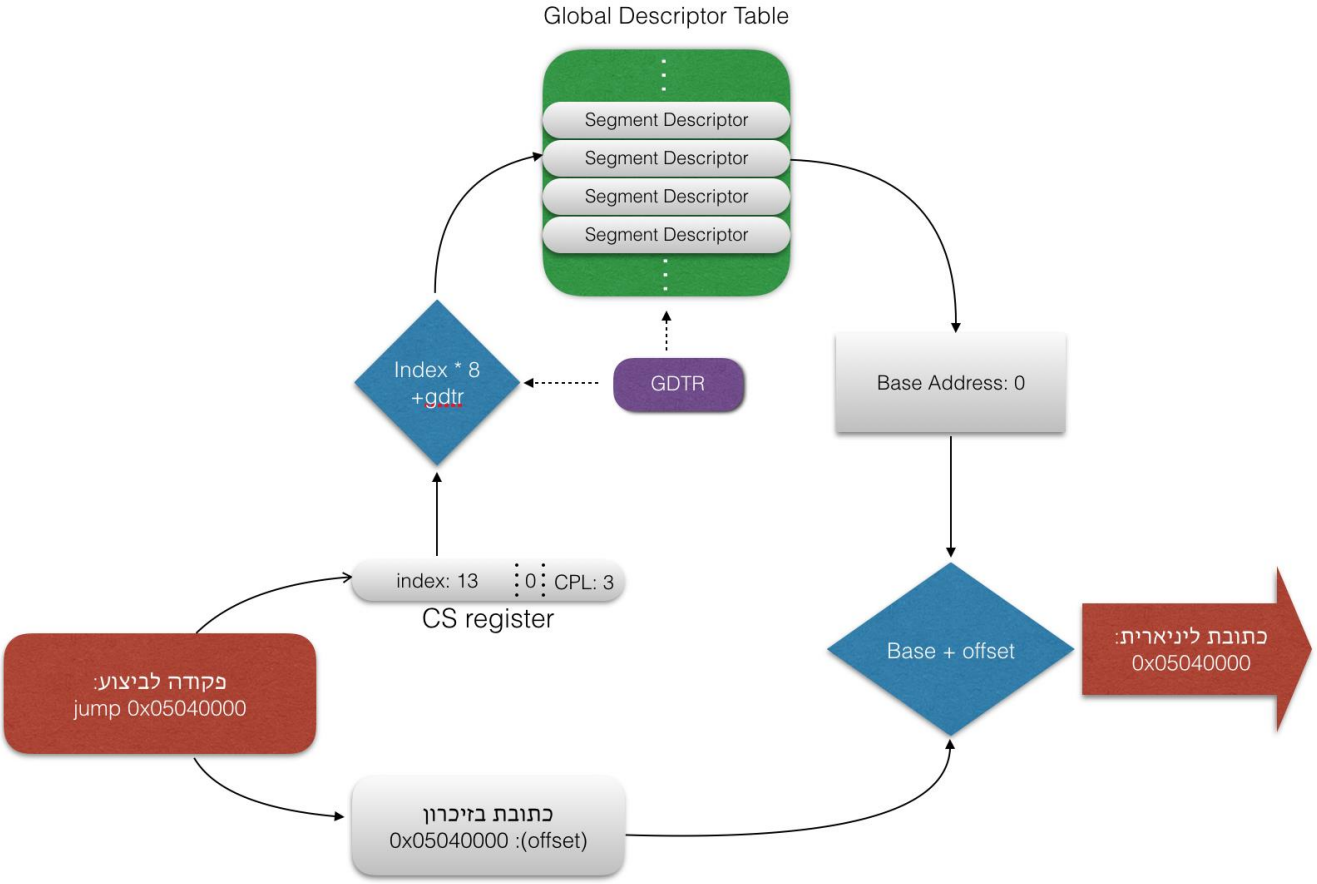
הערה: שימו לב שלקחנו לדוגמא את ה-CS שנמצא באזור זיכרון השמור ל-Kernel. קיים גם סגמנט מקביל ב-User Mode בו הערכים שונים לחלוטין. ממליץ בחום לקחת ה-CS שנותן לכם User-Mode Debugger כמו Olly ולנסות להבין אותו כפי שעשינו כאן בעזרת Kernel Debugger. זה יהיה תרגול מצויין ויעזור לחדד את ההבנה של המאמר.

רגע רגע, אז עברנו את כל התהליך הזה בשביל לראות ש"כתובת הבסיס" היא 0?! התשובה היא: לגמרי כן. זוכרים שהסגמנט נוצר כי בעבר לא היו אוגרים או פקודות הגדולות מ-16 ביט?

אז כיום כמו שאתם יודעים המצב אינו כך, והאוגרים בגודל של 32 ביט ב-x86 ו-64 ביט ב-x64. מה שאומר שהם יכולים לכסות את כל טווח הכתובות גם ללא צורך בסיגמנטים! למצב הזה קוראים באינטל **Flat Memory Model**.

Windows אינה עושה שימוש ב-Segmentation בדרך המסורתית לצורך גישה לזיכרון (base:offset), אלא לשם **אבטחה בלבד**.

התרשים הבא מסכם את כל התהליך:



אחרי שהבנו מהו Segmentation נעבור לסקירה של מנגנון ה-Paging. לאחר מכן, נוכל להבין איך שניהם משלבים כוחות ביחד עם מערכת ההפעלה בכדי לספק לנו את הבסיס לאבטחה.

דפדפוף (Paging)

לפני נתחיל לתאר מה זה בכלל Paging ואיך הוא קשור לאבטחה, נתחיל מסקירה קצרה של איך מתנהלת הגישה לזיכרון ומה הצורך שבגללו המציאו את הפיצ'ר הזה.

כשהמעבד מבקש מלוח האם לגשת לכתובת מסיומת בזיכרון הראשי / RAM הוא עושה זאת באמצעות כתובות פיזיות. כתובות אלו הינן מוחלטות ומייצגות את מספר התא בתוך החומרה. המעבד רשאי לגשת לכל תא שיבחר - בלי תוספות וללא כל בדיקת אבטחה!

תארו לכם מצב בו כל תוכנית שרצה בתוך המעבד הייתה יכולה לבקש מהמעבד לגשת לאיזו כתובת פיזית שהיא חפצה. תוסיפו לזה את העובדה שמדובר במערכת הפעלה מרובת משימות שרצים בה-4 כרטיסיות של Chrome, נגן פלאש, ותוכנית ב-C שנכתבה על ידי מתכנת מתחיל.

שלוש בעיות מרכזיות נוצרות כאן:

4. **יציבות** - כל תוכנית יכולה לגשת לכל כתובת בזיכרון. משמע כל כתיבה לא נכונה לזיכרון תוכל להקריס את המערכת כולה.
5. **אבטחה** - למעשה, כל תוכנית יכולה לגשת לאן שתרצה - בלי הרשאות, בלי בדיקות וללא כל הגנה.
6. **ניצול זיכרון** - תוכנה אחת יכולה לתפוס את כל מרחב הזיכרון הפנוי בחומרה מה שיוכל למנוע ממנו להריץ תוכניות נוספות, או להחזיר אותנו לבעיית היציבות.

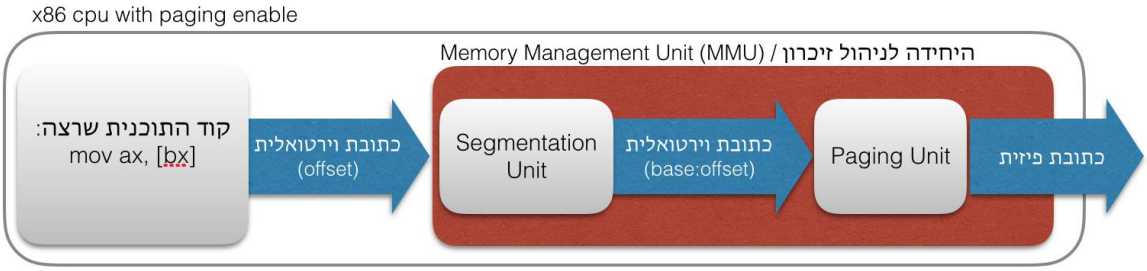
ועכשיו נחזור ל-Paging. בהגדרתה הכללית, Paging הינה שיטה לניהול זיכרון המגדירה סוג חדש של כתובות: **כתובות וירטואליות**. בכתובת אלו יעשה שימוש אך ורק בתוך המעבד, ותפקיד המעבד לבצע את ההמרה בין הכתובות הוירטואליות לכתובות הפיזיות לפני כל גישה לרכיב הזיכרון. חשוב להדגיש - **לכתובות הוירטואליות אין כל משמעות מחוץ למעבד!**

ניתן כדוגמא את קטע קוד הבא:

```
int main ()
{
    int digital = 1000;
    int *pWispher = &digital;
}
```

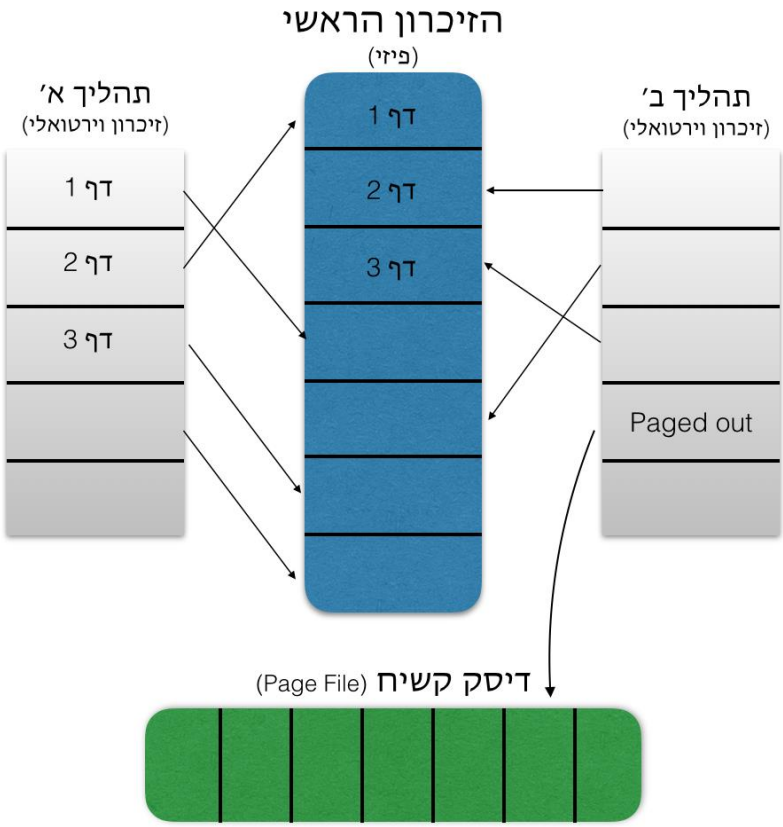
המצביע למעשה מכיל כתובת זיכרון וירטואלית, אותה המעבד מתרגם לכתובת פיזית בה הוא יעשה שימוש כשהו פונה לרכיב הזיכרון.

נעשה סדר בתהליך שעברנו עד כה:



בכדי להיות יותר יעילים, הוחלט לחלק את כל מרחב הזיכרון (הוירטואלי והפיזי) לחלקים הנקראים בשם דפים (Pages). גודל כל דף משתנה בהתאם להגדרתה של מערכת ההפעלה. נכון למאמר שלנו, במערכת הפעלה Windows 7 X86 גודל הדפים נע בין 4kb ל-4MB.

להלן תרשים המפשט את הרעיון:



אז מה בעצם שיטה זו נותנת לנו?

1. אנו יכולים לדמות לכל תוכנית **מרחב זיכרון רציף** משלה! ובכך, למנוע (חומרתית) התנגשויות בין תהליכים ולפשט למתכנת את עניין ניהול הזיכרון.

2. **ניצול יעיל של זיכרון** - ניתן להעביר דפים שלמים בהם לא נעשה כרגע שימוש לכונן הקשיח (לתוך קובץ שנקרא PageFile (שלא נרחיב עליו במסגרת המאמר), ובכך לפנות מקום לתוכניות אחרות. ברגע שהמעבד יזדקק לאותו הדף, הוא יטען אותו חזרה מהדיסק הקשיח. אגב, מסיבה זו נקראת השיטה - דפדפוף.

3. אנו יכולים לתת מאפיינים נוספים לכל דף, כגון - **הרשאות**!

החסרון של Paging נובע מכך שהתרגום לוקח זמן, וכתוצאה מכך נגרמת פגיעה בביצועים. לשם כך המעבד מחזיק רכיב בשם TLB המבצע Caching לתרגומים האחרונים ובכך משפר את זמן התגובה באופן משמעותי.

אז איך Paging מתבצע?

הקישור בין דף הנמצא בזיכרון הפיזי לדף הנמצא בזיכרון הוירטואלי, מתבצע בתוך טבלה הנקראת Page Table. היחידה לניהול זיכרון במעבד עושה שימוש בטבלאות אלו בכדי לבצע את ההמרה בין כתובת וירטואלית לכתובת פיזית.

הטבלאות (יותר נכון להתייחס אליהן כאל עץ בינארי) ממויינות ומחולקת לשתי רמות:

- **Page Directory** - כל רשומה מכילה את הכתובת הפיזית לטבלת הדפים הרלוונטית + דגלים.
- **Page Table** - טבלת הדפים, המכילה עבור כל דף את הכתובת הפיזית בו הוא נמצא + דגלים. מהכתובת הפיזית של הדף נלקח **offset** המייצג את המרווח שבין תחילת הדף לבית המבוקש.

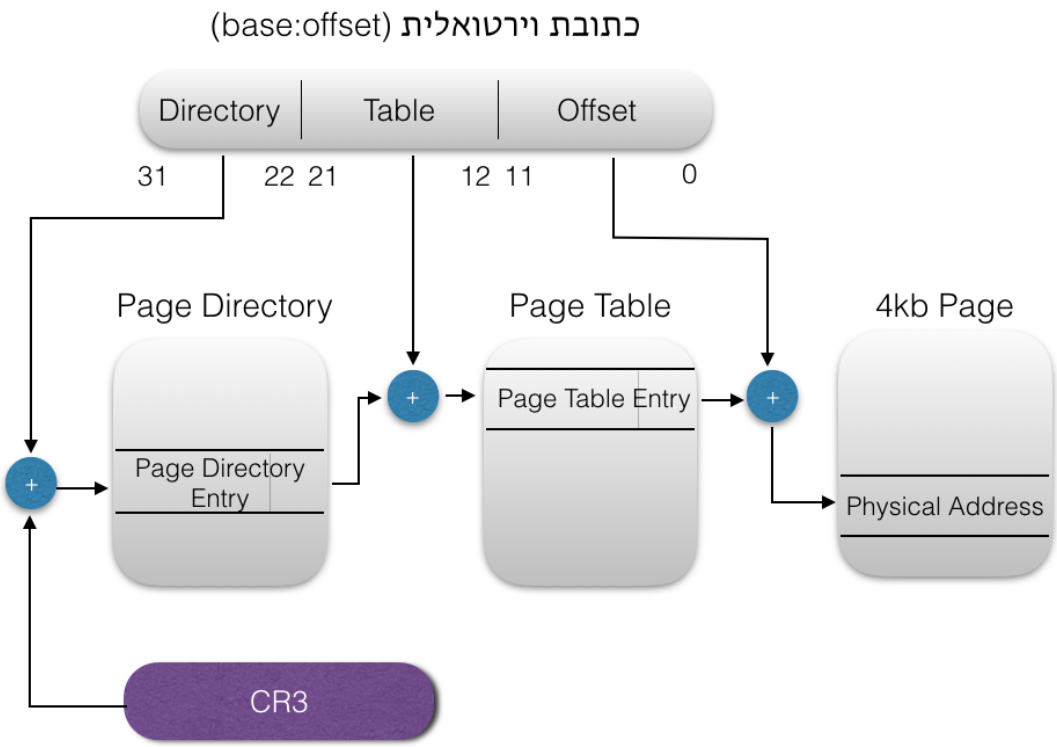
הדגלים שהזכרנו קודם לכן, הנמצאים ב-2 הטבלאות, מכילים מידע נוסף על הדף כגון: גודל, הרשאות וכו'. הדגל בו נתמקד מסומן באות U (User/Supervisor).

הערה: במעבדי IA-32 קיים פיצ'ר בשם PAE אשר תפקידו לאפשר מיפוי של מרחב זיכרון הגדול מ-4GB. זאת באמצעות הוספה של רמה נוספת לטבלאות הנקראת Page Directory Pointer Table. כדי לא לסבך יתר על המידה את העניינים, נתייחס בדוגמאות למצב בו אפשרות זו כבויה. על כן, במידה ואתם אנשים דגולים שמתרגלים מאמרים, שימו לב שאפשרות כבויה (הביט החמישי באוגר הבקרה CR4 שווה ל-0).

ובכל זאת, איך יחידת ניהול הזיכרון יודעת איפה ממוקמות הטבלאות?

בכל CPU/core קיים אוגר הבקרה **CR3**. תפקידו הוא להצביע על הכתובת הפיזית של הרמה הגבוהה ביותר (Page Directory במקרה שלנו).

להלן תרשים המתאר את המבנה של הכתובת הוירטואלית ושיטת התרגום:



כפי שניתן לראות, הכתובת הוירטואלית נחלקת לשלושה חלקים. כל חלק הוא בעצם ה-Offset לטבלה / בלוק הזיכרון הרלוונטי, בדיוק כפי שמתואר בתרשים. כל רשומה מסוג PDE או PTE מיוצגות בפורמט הבא:

לצורך התרגום, נתעלם מהדגלים ונתייחס אך ורק לכתובת הפיזית. רק אל תשכחו אותם כי מיד לאחר מכן נחזור אליהם.

נדגים את תהליך תרגום הכתובות צעד אחר צעד ולפי התרשים באמצעות Kernel Debugger.

נתחיל:

1. נמצא כתובת וירטואלית אותה נרצה לתרגם. נצטרך למצוא כתובת זיכרון שהיא איננה Paged-Out. כלומר, נמצאת בתוך הזיכרון ולא הועברה לדיסק הקשיח. בכדי לעשות זאת, נשתמש בפקודה pcr!.

פקודה זו תציג לנו את מבנה הנתונים המייצג את המעבד הנוכחי. אנו ניקח את כתובת הזיכרון של ה-Thread הנוכחי שרץ באותו הרגע במעבד. כך נבטיח שבסבירות גבוהה דפים אלו יהיו ממופים לזיכרון.

```
kd> !pcr
KPCR for Processor 0 at 82d3ac00:
    Major 1 Minor 1
       NtTib.ExceptionList: 8e6f21bc
           NtTib.StackBase: 00000000
          NtTib.StackLimit: 00000000
         NtTib.SubSystemTib: 801c7000
              NtTib.Version: 00036ef7
         NtTib.UserPointer: 00000001
              NtTib.SelfTib: 7ffda000

                    SelfPcr: 82d3ac00
                       Prcb: 82d3ad20
                       Irql: 0000001f
                        IRR: 00000000
                        IDR: ffffffff
              InterruptMode: 00000000
                        IDT: 80b95400
                        GDT: 80b95000
                        TSS: 801c7000

             CurrentThread: 86b7bd48
                NextThread: 00000000
                IdleThread: 82d44340

                  DpcQueue:
```

2. נמיר את כתובת וירטואלית זו לבסיס ספירה בינארי, לפי הפורמט שתיארנו באיור:

Page Directory (10 MSB)	Page Table (10 Bit)	Offset (12 LSB)
1000011010	1101111011	110101001000

3. נתחיל במציאת הרשומה Page Directory Entry. על מנת לעשות זאת, נמצא את ערכו של האוגר CR3, אשר מייצג את הכתובת הפיזית של תחילת הטבלה (Page Directory). לאחר מכן, נכפיל את

האינדקס בטבלה שקיבלנו בסעיף הקודם ב-4 בכדי לקבל את המספר בבתים (היות והוא מצביע ל-DWORD), ונחבר לו את ערך האוגר CR3:

```
kd> r cr3
cr3=12e6f000
```

1000010111(Bin) -> (0x21a * 0x4) + 0x12e6f000 = 0x**12e6f868**

הפקודה dc! תאפשר לנו לצפות בזיכרון הפיזי:

```
kd> !dc 12e6f868
#12e6f868 3fe88863 3fe89863 3d3c2863 00000000 c..?c..?c(<=....
#12e6f878 3d3c7863 0361a863 0361b863 0361c863 cx<=c.a.c.a.c.a.
#12e6f888 00000000 0361e863 00000000 03621863 ....c.a.....c.b.
#12e6f898 21faa863 19794863 21122863 14c86863 c..!cHy.c(.!ch..
#12e6f8a8 260c6863 28405863 28387863 2c80e863 ch.&cX@(cx8(c..,
#12e6f8b8 28308863 283c9863 2844a863 1294e863 c.0(c.<(c.D(c...
#12e6f8c8 285d5863 27fed863 274d4863 121c6863 cX](c..'cHM'ch..
#12e6f8d8 27a22863 2658a863 25f84863 1118f863 c(.'c.X&cH.%c...
```

מכאן אנו מבינים ש:

Page Directory Entry = 0x3fe88863

4. כפי שאמרנו קודם, ה-12 ביטים התחתונים מייצגים את הדגלים, ולכן נקח את ה-20 העליונים ונחבר אליהם את עשרת הבתים הבאים בכתובת הוירטואלית (המייצגים את ה-Page Table Entry). לא לשכוח להכפיל ב-4 כי מדובר ב-DWORD:

1101111011 (Bin) -> 0x37b

0x3fe88000 + (0x4 * 0x37b) = 0x3fe88dec

נצפה שוב בתוכן הזיכרון:

```
kd> !dc 3fe88dec
#3fe88dec 3db7b963 3db7c963 3db7d963 3db7e963 c..=c..=c..=c..=
#3fe88dfc 3db7f963 3db80963 3db81963 3db82963 c..=c..=c..=c).=
#3fe88e0c 3db83963 3db84963 3db85963 3db86963 c9.=cI.=cY.=ci.=
#3fe88e1c 3db87963 3db88963 3db89963 3db8a963 cy.=c..=c..=c..=
#3fe88e2c 3db8b963 3db8c963 3db8d963 3db8e963 c..=c..=c..=c..=
#3fe88e3c 3db8f963 3db90963 3db91963 3db92963 c..=c..=c..=c).=
#3fe88e4c 3db93963 3db94963 3db95963 3db96963 c9.=cI.=cY.=ci.=
#3fe88e5c 3db97963 3db98963 3db99963 3db9a963 cy.=c..=c..=c..=
```

כלומר:

Page Table Entry = 0x3db7b963

גם כאן, ה-12 ביטים התחתונים מייצגים את הדגלים, לכן ה-20 ביטים העליונים מייצגים את הכתובת הפיזית של הדף בזיכרון! כלומר: 0x3db7b000

5. השלב האחרון שנותר הוא למצוא את הבייט בתוך הדף אליו הכתובת הוירטואלית מתחייסת. בכדי לעשות זאת, נחבר את ה-offset שנמצא בכתובת הוירטואלית (12 ביטים תחתונים) ונחבר אותם לכתובת הפיזית של העמוד. שימו לב שה-offset מיוצג בבתים ולכן הפעם אין צורך להכפיל ב-4!

110101001000 (Bin) -> 0xd48

0x3db7b000 + 0xd48 = **0x3db7bd48**

זהו! למעשה הכתובת הוירטואלית 0x86b7bd48 ממוקמת בכתובת 0x3db7bd48 בזיכרון הפיזי. בכדי להוכיח זאת נשתמש בפקודה dc ללא סימן קריאה, אשר מציגה את תוכנו של הזיכרון הוירטואלי, למול dc! המציגה את התוכן בזיכרון הפיזי:

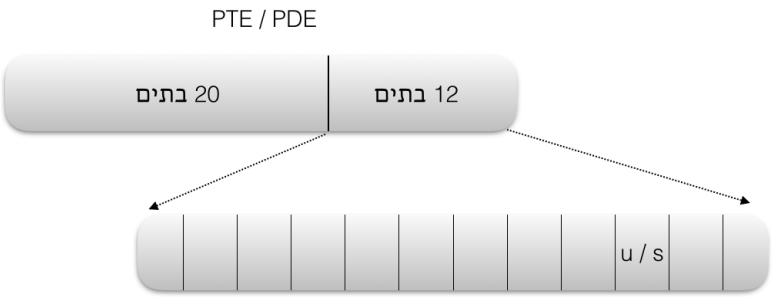
```
kd> !dc 3db7bd48
#3db7bd48 00000006 00000000 86b7bd50 86b7bd50 ........P...P...
#3db7bd58 8315ad09 00000000 00000000 00000000 ................
#3db7bd68 852310e7 00000000 8e6f2ed0 8e6f0000 ..#.......o...o.
#3db7bd78 8e6f2b88 00000000 00000100 00000001 .+o.............
#3db7bd88 86b7bd88 86b7bd88 86b7bd90 86b7bd90 ................
#3db7bd98 8689e8e0 08000000 00000000 00000000 ................
#3db7bda8 00000000 00000519 01000702 00000000 ................
#3db7bdb8 86b7be08 82d3df80 82d3df80 867c2840 ............@(|.
kd> dc 86b7bd48
86b7bd48  00000006 00000000 86b7bd50 86b7bd50  ........P...P...
86b7bd58  8315ad09 00000000 00000000 00000000  ................
86b7bd68  852310e7 00000000 8e6f2ed0 8e6f0000  ..#.......o...o.
86b7bd78  8e6f2b88 00000000 00000100 00000001  .+o.............
86b7bd88  86b7bd88 86b7bd88 86b7bd90 86b7bd90  ................
86b7bd98  8689e8e0 08000000 00000000 00000000  ................
86b7bda8  00000000 00000519 01000702 00000000  ................
86b7bdb8  86b7be08 82d3df80 82d3df80 867c2840  ............@(|.
```

וכמובן שעכשיו, כשאנו יודעים איך התהליך עובד, נגלה לכם שיש פקודה המציגה את ה-Page Table Entry אליה מצביעה הכתובת הוירטואלית באופן אוטומטי, והיא: pte!

```
kd> !pte 86b7bd48
                    VA 86b7bd48
PDE at C0300868            PTE at C021ADEC
contains 3FE88863          contains 3DB7B963
pfn 3fe88 ---DA--KWEV      pfn 3db7b -G-DA--KWEV
```

נותר לנו נושא אחרון וחשוב לפני שאנחנו סוגרים את הפרק של ה-Paging, והוא - לדבר קצת על הדגלים.

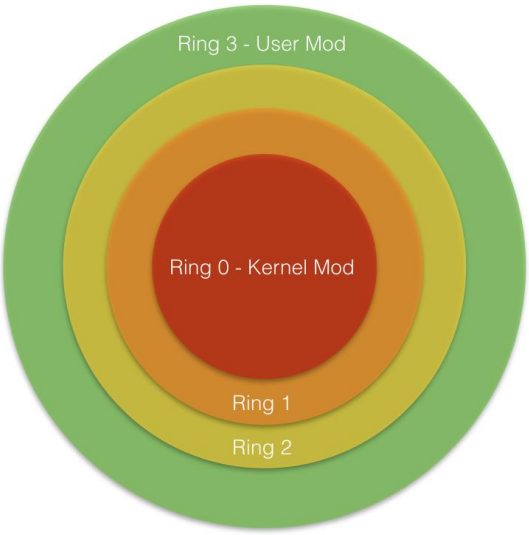
זוכרים שב-PDE וב-PTE התעלמנו מה-12 ביטים הראשונים (השקולים ל-3 ספרות ה-Hex הראשונות)? ובכן דגלים אלו מייצגים את המאפיינים והמצב של כל אחד מהדפים בזיכרון:



הדגל הרלוונטי למאמר שלנו הוא הדגל U/S שמשמעותו User / Supervisor. במידה ודגל זה אינו דלוק, רק קוד ברמת הרשאה גבוהה יכול לגשת אליו.

הכוונה ברמת הרשאה גבוהה היא שה-CPL של ה-Code Segment, עליו דיברנו קודם, שווה ל-0. במידה ולא, המעבד זורק חריגה "Page-Fault Error" ולא מאפשר לגשת לדף.

מגדירים באינטל 4 רמות הרשאה שונות לקוד המכונות בשם: **Protection Rings**. בשיטה זו, קוד יכול לגשת רק למידע ברמת ההרשאה שלו, או מידע ברמת הרשאה נמוכה משלו. מערכות הפעלה מודרניות (כגון לינוקס ו-Windows) עושות שימוש רק ברמות 0 ו-3. כאשר 0 מייצגת את רמת ההרשאות הגבוהה ביותר, בה נמצאת הליבה של מערכת ההפעלה / ה-kernel, ו-3 מייצגת את רמת ההרשאות המוגבלת של המשתמש.



שימו לב: זה לא משנה איזו רמת הרשאות נותנת לך מערכת ההפעלה (Guest, Administrator, System). כל הקוד השייך למשתמש רץ ב-User Mode!

מערכת ההפעלה Windows 7x86 מחלקת את הזיכרון ל-2 חלקים. את החלק העליון (2GB) היא מקצה לליבה (0x80000000 - 0xffffffff) ואת החלק התחתון (2GB) היא מקצה למשתמש (0x00000000-0x7fffffff).

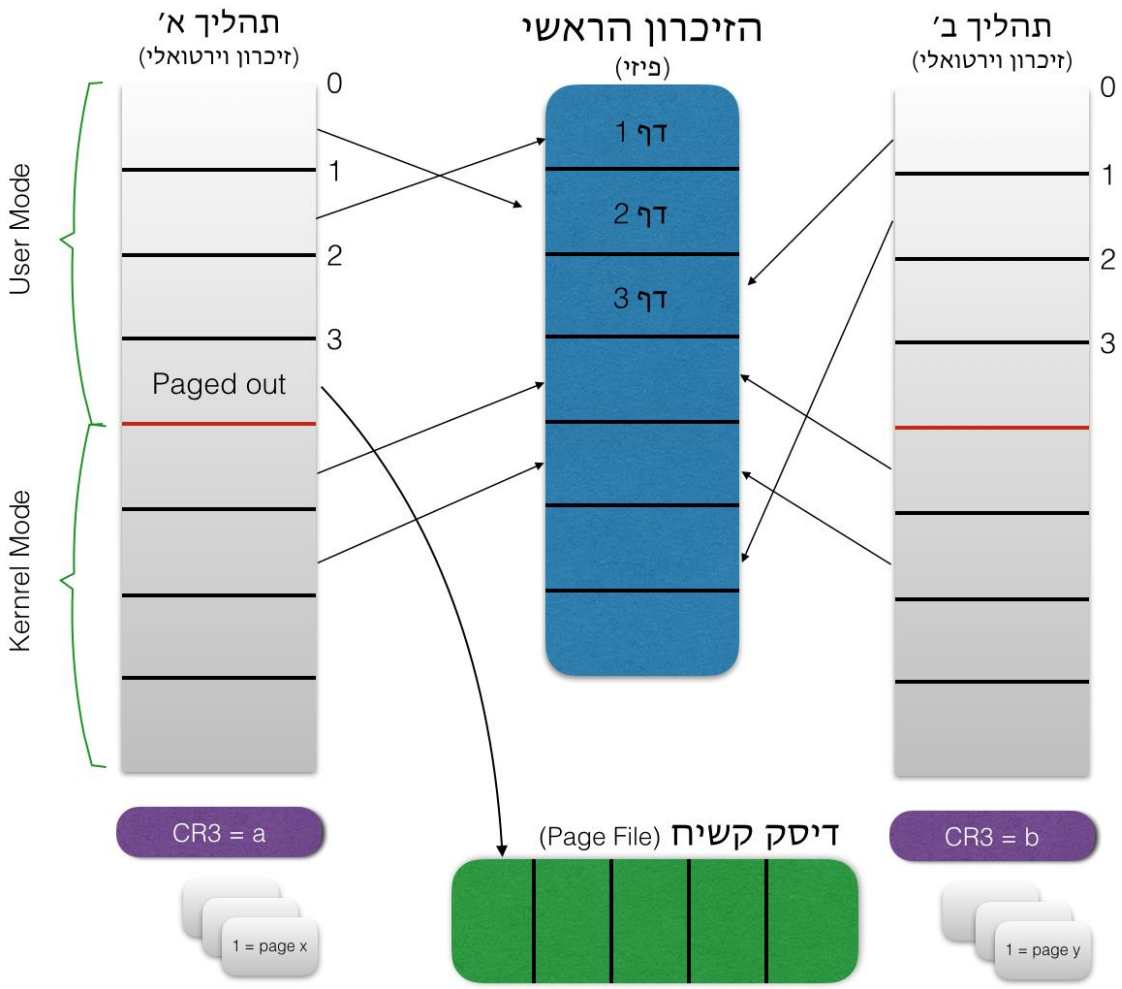
על מנת לבצע זאת, מערכת ההפעלה עושה שימוש בדגל U/S שנמצא ב-PTE וב-PDE שהצגנו קודם לכן. כך יכולה מערכת ההפעלה לצבוע את טווח הכתובות הוירטואליות העליונות ככזה השייך לליבה של מערכת ההפעלה. בדרך זו, כל ניסיון גישה של קוד הרץ ע"י המשתמש לליבה ימנע ע"י המעבד.

המעבד ידע באיזו רמת הרשאות הקוד נמצא לפי ה-CPL של ה-Code Segment. (**למתחכמים שבכם**: לא ניתן לערוך את ה-CPL מ-User Mode)

ובכן, הדבר מבטיח לנו שקוד הרץ ע"י המשתמש לא יוכל לגשת ולערוך מידע השייך למערכת ההפעלה - אך מה לגבי ה-User Mode? מה מבטיח לנו שתהליך אחד לא יפזול לזיכרון של תהליך אחר?

כאן ה-Paging משחק תפקיד מרכזי. מערכת ההפעלה מדמה לכל תהליך מרחב זיכרון וירטואלי שלם כאילו כל הזיכרון שייך רק לו!

כלומר, לכל תהליך יש טבלאות דפים שלו, אשר רק הוא נגיש להם. הדבר מתבטא בכך שלכל תהליך יש ערך שונה באוגר CR3, מה שמונע חומרתית מתהליך מסויים לגשת למרחב הזיכרון של תהליך אחר. על מנת למנוע מצב בו כל תהליך מחזיק העתק של הקרנל כולו, מידע שנמצא בשימוש ברוב התהליכים כגון הקרנל ו-DLL-ים נפוצים, ממופים לאזורים זהים בזיכרון. התרשים הבא ינסה לעשות קצת סדר:



כפי שניתן לראות, לכל תהליך יש את אותן כתובות וירטואליות, אך הן מתורגמות לכתובות פיזיות שונות. זאת מפני שטבלאות הדפים והאוגר CR3 שונים מתהליך לתהליך.

אם נחשוב על זה קצת, נבין שקוד משתמש למעשה מגובל לחלוטין. היות והחומרה נגישה רק מרמת ה-Kernel, קוד משתמש לא יכול בעצם לעשות דבר - לא לכתוב לדיסק קשיח, לא לקבל קלט מהמשתמש, ואפילו לא להציג דברים למסך!

לשם כך יש את ה-WinAPI שהתרגלנו אליהם, המאפשרים לנו באופן מבוקר ובתקווה בטוח, לבצע פעולות בסיסיות - לקבל קלט מהמשתמש, לשמור לקובץ או אפילו להציג חלון. אותם WinAPI בסופו של דבר יודעים להעביר את הבקשה שלנו מה-User-Mode ל-Kernel-Mode, שם מתבצעת רוטינה מוגדרת המבצעת את מבוקשנו. המעבר בין User-Mode ל-Kernel-Mode בדרך כלל מבוצע על ידי הפקודה int או sysenter במעבד, עליהן לא נרחיב במסגרת המאמר.

לסיכום

במסגרת המאמר הגדרנו שלושה צפיות מרכזיות ממערכת ההפעלה, אשר יתנו לנו בסיס מוצק לאבטחה. הצגנו 2 פיצ׳רים מרכזיים: **Paging** ו-**Segmentation** וראינו כיצד Windows 7 עושה בהם שימוש בכדי לענות על ציפיות אלו.

הציפיה הראשונה - לא לאפשר לתהליך לגשת או לערוך מידע של תהליך אחר. ראינו כי המענה לזה ניתן באמצעות שימוש במנגנון ה-Paging, באמצעותו יוצרים הפרדה חומרתית, שלא מאפשרת לזיכרון של תהליך אחד לגשת לזיכרון אחר. (הדרך היחידה לעשות זאת היא רק באמצעות דפים משותפים הממופים ל-2 התהליכים, כגון הקרנל או shared memory).

הציפיה השנייה - אל תאפשר לתהליך לגשת או לערוך מידע השייך למערכת ההפעלה. ראינו כי מערכת ההפעלה עושה שימוש הן ב-Paging והן ב-Segmentation בכדי ליצור הפרדה בין מרחב הזיכרון של קוד המשתמש (User-Mode), לבין מרחב הזיכרון של מערכת ההפעלה (Kernel Mode). ראינו כי תהליך אינו יכול לגשת למרחב הזיכרון של מערכת ההפעלה באופן ישיר.

הציפיה השלישית והאחרונה - אל תאפשר לקוד משתמש לגשת באופן ישיר לחומרה. המענה ניתן גם הוא ברגע שמוגדר כי רק קוד בעל הרשאות גבוהות (Kernel Mode) יכול לגשת אל משאבי החומרה. המשתמש, עושה שימוש ב-API מוגדרים, המאפשרים לו לדבר דרך ה-Kernel באופן, מקובע, מבוקר ובטוח (בשאיפה).

מקורות מידע נוספים

- http://www.intel.com/Assets/en_US/PDF/manual/253668.pdf
- https://en.wikipedia.org/wiki/Global_Descriptor_Table
- https://en.wikipedia.org/wiki/Intel_Memory_Model
- https://en.wikipedia.org/wiki/Memory_segmentation
- https://www.youtube.com/watch?v=nsWklEuhRmM
- http://duartes.org/gustavo/blog/post/memory-translation-and-segmentation/
- http://duartes.org/gustavo/blog/post/cpu-rings-privilege-and-protection/
- https://iambvk.wordpress.com/2007/10/10/notes-on-cpl-dpl-and-rpl-terms/
- https://en.wikipedia.org/wiki/Paging
- http://wiki.osdev.org/Paging