Technologic Papers

(Native) Hypervisor from Scratch- חלק א'

מאת אמיר ישורון

הקדמה

ברוכים הבאים לחלק הראשון מתוך שניים בסדרת מאמרים העוסקת בכתיבת native hypervisor. ייתכן וכרגע זה נשמע כמו סינית עבורכם, אך יש לנו עוד דרך ארוכה לעבוד שבסופה תהפכו למומחים בתחום. אתחיל מרקע קצר מאוד על מנגנון הוירטואליזציה.

בשנים האחרונות השימוש בוירטואליזציה לצרכי הגנה על מערכות הפך נפוץ (לדוגמא Hyper-V של Microsoft), ומספר חוקרי האבטחה העוסקים בתחום גדל משמעותית. על רקע העליה הזאת, אם תנסו לחפש מדריכים על כתיבת hypervisor-ים (אל תיבהלו מהמילה הזאת, עוד נרחיב עליה ממש בקרוב) אתם תמצאו כאלו ללא יותר מדי מאמץ. מה ההבדל בין כל אלו לבין מה שאני עושה כאן? גם על זה נדבר. ראשית נגדיר כמה מושגים שיש להכיר לפני שממשיכים הלאה.

מהי וירטואליזציה

כמו שהמושג רומז לנו - הכוונה היא להפוך משהו פיזי לוירטואלי. ב"עולם המחשבים", המינוח המודרני מתכוון לוירטואליזציה של החומרה, כלומר לסמלץ את פעולת החומרה מול מערכת ההפעלה. לגוף שאחראי על הפעולה הזאת קוראים hypervisor.

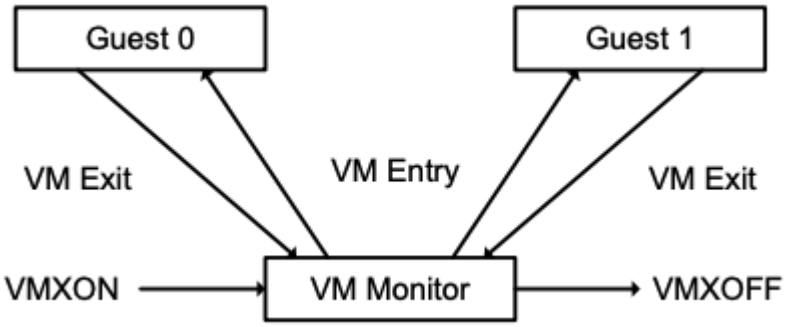
הרכיב החומרתי שמספק את התמיכה בוירטואליזציה הוא המעבד. אי שם ב־2005 ו־2006, יצרניות המעבדים Intel ו־AMD הכריזו על תמיכה בטכנולוגיות חדשות: VT-x ו־AMD-V, שמאפשרות את הקסם הזה שאנחנו נדבר עליו היום. במאמר הזה אנחנו נתמקד בוירטואליזציה על גבי מעבדי Intel באמצעות טכנולוגיית VT-x, אך הרעיונות הכללים נכונים גם בנוגע למעבדי AMD ו־ARM (אם כי המימוש יהיה שונה).

זה נחמד מאוד... אבל תכל'ס, מה היכולות ש־**hypervisor** מביא איתו?

בסופו של דבר, ל־hypervisor יש את היכולת לגרום ל"עצירה רגעית" של מערכת ההפעלה כאשר מתרחשות פעולות מוגדרות מראש. לעצירה הרגעית הזאת אנחנו קוראים vm-exit. כאשר מתרחש vm-exit, השליטה עוברת ל־hypervisor. ה־hypervisor יכול לבדוק מה הסיבה שגרמה ל־vm-exit: האם הייתה גישה למקום ש"סימנו" אותו בזיכרון? האם רצה פקודת אסמבלי "אסורה" שגרמה ל־vm-exit? האם התרחש page-fault, וביקשנו מה־hypervisor לגרום ל־vm-exit כאשר מתבצע page-fault?

כמובן שיש עוד המון סיבות שיכולות לגרום ל־vm-exit. לכל סיבה ה־hypervisor יכול להגיב באופן שונה. בין היכולות של ה־hypervisor זה לשנות את האוגרים (registers) של מערכת ההפעלה, לכתוב למרחב הכתובות של מערכת ההפעלה, לחסום קריאה, כתיבה והרצה מ־page-ים של מערכת ההפעלה, ולמעשה - יש ביכולתו לעשות הכל.

לאחר שה־hypervisor מטפל בעצירה הרגעית, הוא מחזיר את השליטה למערכת ההפעלה. לאירוע של החזרת השליטה למערכת ההפעלה קוראים vm-entry. כך הדברים נראים ממעוף הציפור:



[קרדיט: Intel Developer Manual Volume 3C]

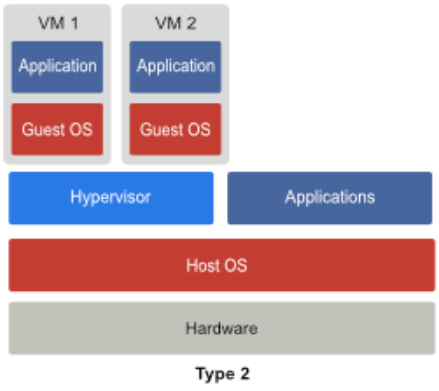
- המונח Guest מתייחס למערכת/מערכות ההפעלה המותקנת על המחשב.
- את ה־hypervisor מכנים לעיתים גם Virtual Machine Monitor, או בקיצור VMM.

במאמר הזה אני אציג את סוגי ה־hypervisor-ים השונים ונתמקד בכתיבת native hypervisor, עליו אפרט ממש עוד רגע. אז למה אנחנו מחכים? בואו נצא לדרך!

סוגים שונים של Hypervisor-ים

Hypervisor Type 2

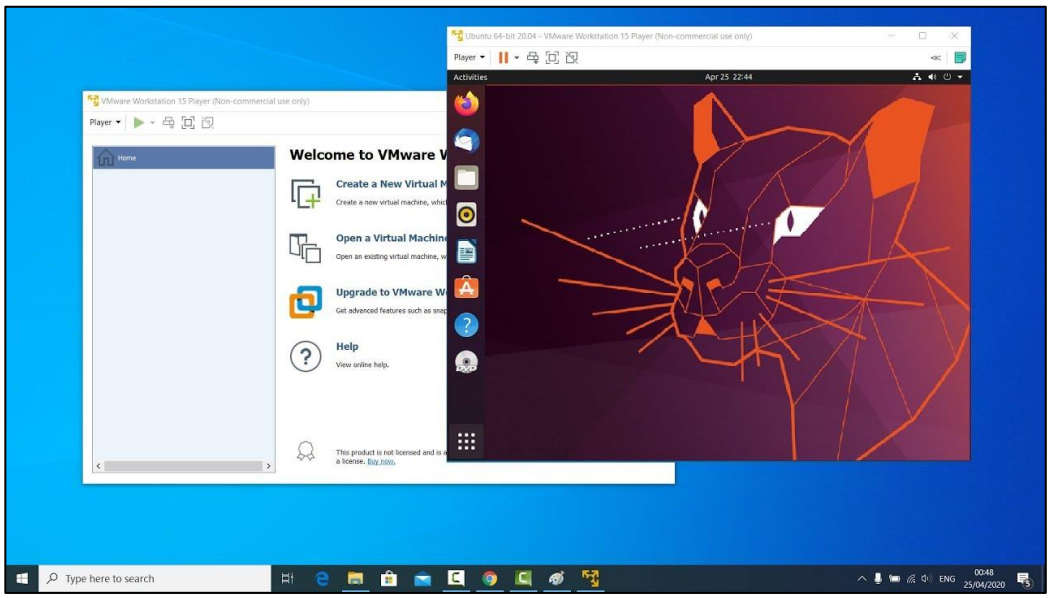
אני מאמין ש־99% מהקוראים של המאמר הריצו בעבר הפצה כלשהי של Linux על מחשב ה־Windows שלהם, או לחילופין התקינו גרסא כלשהי של Windows על גבי מחשב ה־Linux שלהם באמצעות מכונה וירטואלית. זו היא דוגמא מצויינת לשימוש בוירטואליזציה. אם נחלק לשכבות את התרחיש הזה שתיארתי. זה יראה כך:



[קרדיט: vembu.com]

למטה, בשכבה התחתונה, נמצאת החומרה הפיזית. מעליה יושבת מערכת ההפעלה, ועליה רץ הגוף שאחראי על סמלוץ החומרה כלפי מערכות ההפעלה שרצות כמכונות וירטואליות. בתרחיש זה, ניתן לחשוב על ה־hypervisor כ"אפליקציה" שרצה על מערכת ההפעלה הראשית.

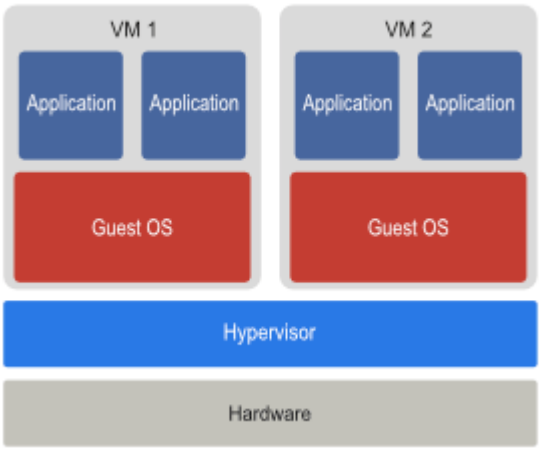
למערכת ההפעלה שמותקנת על המחשב ומדברת ישירות עם החומרה קוראים Host OS (מארחת), ולמערכות ההפעלה שמותקנות על גבי ה־hypervisor אנחנו קוראים Guest OS. כל גישה לחומרה של ה־Guest OS עוברת דרך ה־Host OS. מבחינת מינוחים, ל־hypervisor שמותקן על גבי מערכת הפעלה כלשהי, אנחנו קוראים hypervisor type 2, וכך למעשה נראה הדבר בפועל:



[קרדיט: youtube.com]

Hypervisor type 1 (Native)

אחרי שהצגנו את הסוג שרובנו מכירים וכנראה יצא לנו גם להתעסק איתו, נציג את הסוג שבשבילו התכנסנו היום - Native hypervisor. בתרשים שלמעלה ראינו שה־hypervisor למעשה מותקן על מערכת ההפעלה, ולכן הוא רשאי להשתמש בכל הכלים שהיא מציעה. האם כל ה־hypervisor-ים מוכרחים להיות מותקנים על מערכת ההפעלה? התשובה היא לא. בואו נסתכל על סוג נוסף:



[קרדיט: vembu.com]

מה אנחנו רואים כאן? לאיפה נעלם ה־Host OS?

בתרשים הזה ה־hypervisor הוא בעצם בתפקיד ה־host, ועל גביו מותקנות מערכות ההפעלה השונות, אשר מכונות Guest OS (במסגרת המימוש שיוצג לאורך המאמר אנחנו נתמוך ב־Guest OS יחיד).

השימוש בסוג זה של hypervisors יכול להיות:

- במקרה של יותר מ־Guest OS יחיד - ניצול אפקטיבי יותר של החומרה על מנת להריץ מספר מערכות הפעלה במקביל, כמו למשל על שרתי ה־EC2 של AWS.
- במקרה של Guest OS יחיד - הגנה על מערכת ההפעלה באמצעות מנגנון שלמערכת ההפעלה אין גישה אליו, כלומר הוא שקוף לחלוטין ויש לו שליטה מלאה על כל מה שקורה. מדובר בכוח מאוד גדול שלא ניתן להשיג אפילו באמצעות קוד שרץ ב־kernel של מערכת ההפעלה.

לשם מה התכנסנו

כמו שהזכרתי קודם, אם תנסו לחפש בגוגל מדריכים לכתיבת hypervisors - כנראה שתמצאו כמה כאלו, וחלקם אף איכותיים ומוסברים היטב. ההבדל בין מה שאני מציג כאן היום לבין המדריכים האלו הוא ה־type של ה־hypervisor.

כל המדריכים שקראתי מסבירים כיצד לכתוב hypervisor שמותקן על גבי מערכת ההפעלה כ־driver, ואני מוכרח לומר שחלקם מסבירים את הנושא באופן פנטסטי. עם זאת, לא יצא לי להתקל אף לא במדריך אחד שמסביר מההתחלה ועד הסוף כיצד לכתוב native hypervisor שיהיה בתפקיד ה־host, ויארח את מערכת ההפעלה שמותקנת עליו כ־guest. החלטתי לנצל את הזמן שיש לי לפני הגיוס, ולכתוב אחד בעצמי.

הרקע הנדרש

אנחנו נצטרך לא מעט ידע קודם לפני שניגש לכתוב את ה־hypervisor שלנו, ואני הולך לכסות 80% עד 90% ממנו כאן במאמר. מנגד, יש מספר דברים שאני לא הולך לכסות, והם: היכרות מעמיקה עם השפה C, הכרות עם x86 assembly, איך לקמפל (ידנית) קוד C באמצעות gcc, איך "לקמפל" קוד Assembly (באמצעות nasm), איך ללנקג' קבצי object (באמצעות ld), היכרות בסיסית עם קונספטים של מערכות הפעלה (כמו מנגנוני ה־paging ו־segmentation למשל). כאן יש מאמר מומלץ שמסביר בהרחבה על מנגנון ה־Paging.

המטרה והכלים העומדים לרשותנו

המטרה: כשאנחנו מתיישבים מול המקלדת שלנו ומתחילים לכתוב קוד, אנחנו תמיד יודעים מה התוצר שאנחנו מכוונים אליו. אם אנחנו יושבים לכתוב תוכנית כלשהי עבור מערכות Windows, אנחנו מצפים שבסופו של יום יהיה לנו קובץ exe; אם אנחנו כותבים דרייבר במערכות Windows, אנחנו מעוניינים בקובץ sys; אם אנחנו מחליטים לכתוב ספריה דינמית, אנחנו מעוניינים בקובץ dll. טוב, אני מאמין שהבנתם את הרעיון.

במקרה שלנו, התוצר יהיה קובץ iso אותו נוכל לצרוב ל־USB ולחבר למחשב פיזי או לחילופין לעשות לו mount דרך VMware למערכת הפעלה כלשהי. אל דאגה, נסביר בקרוב איך אפשר ליצור קבצי iso.

הכלים: סביבת הפיתוח תהיה מערכת linux סטנדרטית (אני עבדתי מעל fedora ו־ubuntu). נשתמש ב־gcc בתור קומפיילר, ב־nasm בתור אסמבלר, ב־ld כלינקר, וב־grub כ־bootloader (לא להיבהל, עוד נרחיב עליו).

סביבת הבדיקה: אנחנו נבדוק את ה־hypervisor באמצעות VMware שמריץ Windows 10 בגרסאת 64bit. כלומר, אנחנו נדאג שה־hypervisor שלנו יעלה לפני Windows, והוא ידאג כבר להעלות את מערכת ההפעלה כשהוא יסיים לאתחל את עצמו. מה שאנחנו צריכים בתור התחלה זה רק להעלות את הקוד שלנו לפני מערכת ההפעלה עצמה.

שימו לב כי מבחינה תיאורטית hypervisor שהוא native מסוגל להריץ כל מערכת הפעלה מעליו, והקוד שאנחנו רושמים לא מותאם בהכרח ל־Windows. בפועל, כל מערכת הפעלה צריכה טיפול מעט שונה וקונפיגורציות מעט אחרות. 99% ממה שאכתוב כאן זהה עבור כל מערכות ההפעלה.

דגשים והערות

- כפי שרשמתי למעלה, המאמר יכסה גם נושאים שלא בהכרח קשורים באופן ישיר לוירטואליזציה, כמו מנגנונים שונים וקונספטים כלליים של מערכות הפעלה. תרגישו חופשי לדלג על קטעים שלא רלוונטים עבורכם, אם אתם כבר מכירים את החומר.
- ה־hypervisor נכתב ללא תמיכה במערכות UEFI, אלא רק ב־Legacy BIOS. ייתכן ובעתיד יורחב לתמוך גם ב־UEFI - מוזמנים לעקוב ב־github.

יאללה, בואו נתחיל!

המעבד

מצבים שונים של המעבד:

כמו שהזכרתי קודם, על מנגנון הוירטואליזציה אחראי המעבד. לכן, לפני שאנחנו ניגשים לכתוב קוד שמשפיע על מצב המעבד ומשתמש ב"פיצ'רים" שהוא מציע, עלינו להכיר היטב את אופן הפעולה שלו. למעבד יש מספר מצבים בהם הוא יכול להיות מצוי, כאשר בכל מצב פקודות האסמבלי בפועל יבוצעו באופן שונה. כעת נסביר מעט על המצבים השונים של המעבד, ומתי הוא נמצא בכל מצב. שימו לב כי לא ארחיב על 8086-virtual מכיוון שמצב זה של המעבד לא רלוונטי לכתיבת ה־hypervisor.

Real Address Mode

- במצב זה המעבד מפרש את כל הכתובות ככתובות פיזיות, ואין חלוקה למרחב כתובות וירטואלי.
- ניתן להשתמש באוגרים בגודל 16bit ו־32bit בלבד, כלומר גם אם המעבד תומך ב־64bit, לא ניתן להשתמש ב־rax, rbx וכו׳.
- הכתובות מוגבלות ל־20 ביטים בלבד. האוגרים ax, bx, cx וכו׳ הם בגודל 16 ביטים, אך באמצעות segmentation ניתן להגיע לכתובות בגודל 20 bit-ים.
- ניתן להשתמש בפסיקות (interrupt-ים) של ה־BIOS. למי שלא מכיר, נרחיב על חלק מהם ממש בקרוב.
- על מנת לשמור על תאימות לאחור, זה הוא המצב ההתחלתי שכל יחידת עיבוד נמצאת בה כשהיא עולה לראשונה.
- אין תמיכה Privilege Levels.

המעבד נמצא במצב זה אם הביט במקום ה־0 באוגר CR0 כבוי. בהתאם למה שהוזכר למעלה, ביט זה כבוי כאשר כל ליבה מתחילה לראשונה להריץ קוד על מנת לשמור על backwards compatibility.

Protected Mode

- יש תמיכה במרחב כתובות וירטואלי ו־Paging. האוגר CR3 יכיל את הכתובת הפיזית של ה־page table.
- במצב זה ניתן להשתמש באוגרים בגודל 32bit.
- הכתובות מוגבלות ל־32bit, ויש שימוש ב־segmentation.
- לא ניתן להשתמש בפסיקות של ה־BIOS.
- יש תמיכה ב־Privilege Levels.

המעבד נמצא במצב זה אם הביט במקום ה־0 באוגר CR0 דולק. כמו כן, paging מופעל אם הביט במקום ה־31 באוגר CR0 דולק.

על מנת לעבור מ־real mode ל־protected mode, ראשית נדליק את הביט ה־0 ב־CR0. לאחר מכן נבצע far jump לקוד של 32bit, ושם נפעיל את ה־paging.

זה נראה כך:

```
[BITS 16]
mov eax, cr0
or eax, 1
mov cr0, eax
jmp <32_BIT_SELECTOR>:EnableProtectedMode

[BITS 32]
EnableProtectedMode:
    mov eax, cr0
    or eax, (1 << 31)
    mov cr0, eax
```

Long Mode, מכונה גם **IA32e** או **64bit Mode**

- יש תמיכה במרחב כתובות וירטואלי, ב־Paging והשימוש ב־PAE (Physical Address Extension) הוא חובה.
- במצב זה ניתן ואף רצוי להשתמש באוגרים בגודל 64bit.
- הכתובות מוגבלות למקסימום שהמעבד יכול לתמוך בו. מבחינת זיכרון וירטואלי, יש לנו basically אינסוף זיכרון.
- אין שימוש ב־Segmentation, אך עדיין יש שימוש ב־segment descriptors.
- לא ניתן להשתמש בפסיקות של ה־BIOS.
- יש תמיכה ב־Privilege Levels.

המעבד נמצא במצב זה אם:

- כל תנאי Protected Mode מתקיימים.
- המעבד משתמש ב־PAE paging. ניתן להפעיל קונפיגורציה זו באמצעות הדלקת הביט החמישי של CR4.
- הביט השמיני ב־EFER MSR דולק. אפשר להתייחס ל־EFER MSR בתור אוגר קונפיגורציה נוסף. מומלץ לקרוא על הפקודות rdmsr ו־wrmsr לפני קריאת קטע הקוד הבא.

***הערה**: למצב זה קיים גם מצב ביניים המכונה compatibility mode, אשר מאפשר להריץ אפליקציות של 32bit למרות שהמערכת במצב של 64bit. כדי להגיע מ־compatibility mode ל־Long Mode ה"אמיתי", יש לטעון ל־selector CS ערך ב־descriptor table אשר הביט ה־51 בו דולק. ביט זה מעיד על כך שמקטע הקוד הנוכחי מכיל פקודות של x64.*

בקוד שלפנינו, הטעינה הזו מתבצעת בפקודת ה־jmp:

```
mov eax, cr0
and eax, ~(1 << 31) ; Mode changing requires disabling paging
temporarily
mov cr0, eax
mov eax, cr4
or eax, (1 << 5)
mov cr4, eax
mov eax, 0x17000 ; Set cr3 with PML4[0], will be explained later
mov cr3, eax
mov ecx, EFER_MSR
rdmsr
or eax, (1 << 8)
wrmsr
mov eax, cr0
or eax, (1 << 31)
mov cr0, eax
jmp <64_BIT_SELECTOR>:CompatibilityTo64

[BITS 64]
CompatibilityTo64:
    ; 64bit code goes here
```

ה־Bootloader

כעת, שאנחנו כבר הפכנו מומחים ואנחנו מכירים (כמעט) את כל מצבי המעבד, ניגש לעבודה. ישבנו מול מקלדת ועכבר, פתחנו repository חדש ב־github, חלקנו כבר מכניסים לקורות חיים שאנחנו מומחי וירטואליזציה. אחרי שעשינו את כל זה, איך מתקדמים?

הדבר הראשון שצריך לעשות - הוא להעלות את ה־hypervisor לפני מערכת ההפעלה. כלומר, ברצוננו להריץ קוד ישירות "על החומרה" לפני שמערכת ההפעלה עולה. בשביל לעשות את זה, נצטרך לכתוב קוד שיטען את ה־iso שלנו לזיכרון כאשר המחשב נדלק או לחילופין להשתמש בכלי מוכן שיכול לעשות את זה עבורנו. אנחנו נבחר באופציה השניה, מאחר וזה יחסוך לנו לא מעט עבודה וקוד.

ה־Bootloader בו נשתמש הוא grub, כלי חינמי במסגרת פרויקט GNU. ליתר דיוק, נשתמש ב־multiboot2 שהיא אחת השיטות ש־grub מציעה להעלאת המערכת.

בניגוד למה שכנראה הרוב המבוהל סובר, האופן שבו ה־bootloader פועל הוא מאוד פשוט: בהינתן קובץ elf כלשהו, ובו הקוד (text segment) נמצא בהיסט XYZ, אנחנו נבקש מה־bootloader להעלות את הקוד הזה לכתובת XYZ בזיכרון של המחשב. כמו כן, נציין מה הפונקציה המשמשת כ־entrypoint של המערכת, וזה הכל!

Multiboot2 תעלה עבורנו את המערכת עם הקוד בכתובת XYZ שנבקש ממנה, ותדאג שה־instruction pointer יצביע לתחילת ה־entrypoint שלנו. שימו לב ש־multiboot2 תעלה את המערכת ב־protected mode ללא paging. תוכלו לקרוא על השיטה בה multiboot2 פועלת בקישור הזה.

במסגרת השימוש ב־multiboot2, תפקידנו הוא להכין את הקרקע. כלומר, נצטרך לדאוג לקונפיגורציות בסיסיות בהתאם לקישור הנ"ל:

```
multiboot2_header_start:
    dd 0xE85250D6 ; magic field, DWORD
    dd 0          ; architecture - i386 protected mode, DWORD
    dd multiboot2_header_end - multiboot2_header_start ; header length, DWORD
    dd 0x100000000 - (0xE85250D6 + (multiboot2_header_end -
multiboot2_header_start) + 0) ; checksum, DWORD
    multiboot2_address_tag_start:
        dw 2 ; type, WORD
        dw 0 ; flags, WORD
        dd multiboot2_address_tag_end - multiboot2_address_tag_start ; dize,
DWORD
        dd CODE_BEGIN_ADDRESS
        dd -1 ; data segment is present to the end of the imgae
        dq 0  ; bss
    multiboot2_address_tag_end:
    multiboot2_entry_address_tag_start:
        dw 3      ; type, WORD
        dw 0      ; flags, WORD
        dd multiboot2_entry_address_tag_end - multiboot2_entry_address_tag_start
; size, DWORD
        dd _start ; entrypoint, DWORD
    multiboot2_entry_address_tag_end:
        dd 0
        dd 0
        dw 8
multiboot2_header_end:
```

נעבור על הקונפיגורציות לפי הסדר:

מקטע ראשון:

- שדה ה־magic: מדובר בערך מיוחד אשר ה־bootloader מחפש בקובץ ה־elf שאנחנו מספקים לו על מנת לאתר את תחילת אזור הקונפיגורציות. אין לשנות ערך זה.
- שדה ה־architecture: מכיוון ש־mutiboot2 נועד לתמוך בכמה ארכיטקטורות, אנחנו משתמשים בשדה זה על מנת לציין את הארכיטקטורה עליה אנחנו עובדים. הערך 0 מסמל Protected Mode במעבדי Intel.
- שדה האורך: מציין את אורך מקטע הקונפיגורציות.
- שדה ה־checksum: נועד על מנת להבטיח שה־magic לעיל אכן מצביע לתחילת אזור הקונפיגורציות. תוכלו לקרוא עוד על ה־checksum וכיצד הוא פועל בקישור למעלה.

מקטע שני:

- שדה ה־type: מציין את סוג המקטע. במקרה זה, מדובר במקטע שמציין היכן הקוד שלנו נמצא ב־ elf file שעל בסיסו בנינו את ה־iso.
- שדה ה־flags: חסר חשיבות, אך חובה לציין אותו.
- שדה האורך: מציין את אורך המקטע הנוכחי.
- שדה ה־address: מציין את הכתובת בה הקוד שלנו נמצא ב־elf file שעל בסיסו נבנה את ה־iso. איך אפשר לדעת מה היא הכתובת הזאת? מוזמנים לקרוא על linker scripts ולהתרשם מהקובץ שאני כתבתי, שאיתו אנחנו מלנקג׳ים את קבצי ה־object שלנו.

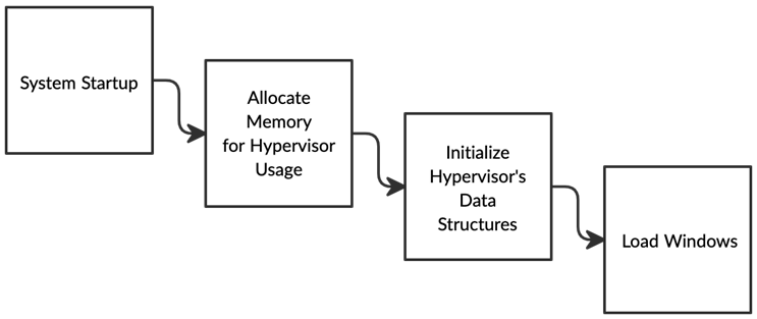
על המקטע השלישי לא ארחיב יותר מדי - אומר רק שהוא מציין היכן נמצא ה־entrypoint של המערכת. במקרה שלנו, ה־entrypoint הוא היכן שהפונקציה start_ נמצאת בזיכרון.

המקטע הרביעי הוא מקטע אשר תפקידו לסמל את סוף אזור הקונפיגורציות. כל הערכים בו הם ערכים חסרי חשיבות עבורנו והם ממוקמים שם כי כך נצטווינו לעשות על ידי הכותבים של grub.

טוב, בואו נעצור ונבין איפה אנחנו נמצאים. לאחר שניצור iso המכיל את הקוד הזה, אפשר לומר שהצלחנו להעלות את המערכת כאשר הפונקציה הראשונה לרוץ היא start_, ואת הפונקציה הזאת אנחנו נכתוב. במילים אחרות, המחשב בשליטתנו!

כל מה שצריך לעשות על מנת ליצור קובץ bootable iso זה לכתוב קובץ grub.cfg מתאים. לא אעבור כאן על איך כותבים קובץ שכזה, אך באתר הרשמי של פרייקט gnu, ב־manual של grub תוכלו למצוא את ההסבר המלא.

לא אשקר - יש עוד המון עבודה, אך תמיד אומרים שעם הנצח לא מפחד מדרך ארוכה. כדי להקל עלינו קצת בכל התהליך, יצרתי תרשים קטן שינחה אותנו. התרשים מתאר בכלליות את השלבים הכרוכים בהעלאת המערכת, ואנחנו נפרק כל שלב לתתי-שלבים בהמשך הדרך.



למעשה, אנחנו כמעט יכולים לסמן V על השלב הראשון. באמצעות grub המערכת עלתה ב־ protected mode, וקוד האסמבלי שהצגתי קודם לכן יביא אותנו ל־long mode. נותר רק להציג את התמיכה ב־ paging (זכרו - לא ניתן להיות ב־long mode כאשר paging לא פועל) ואנחנו מסודרים.

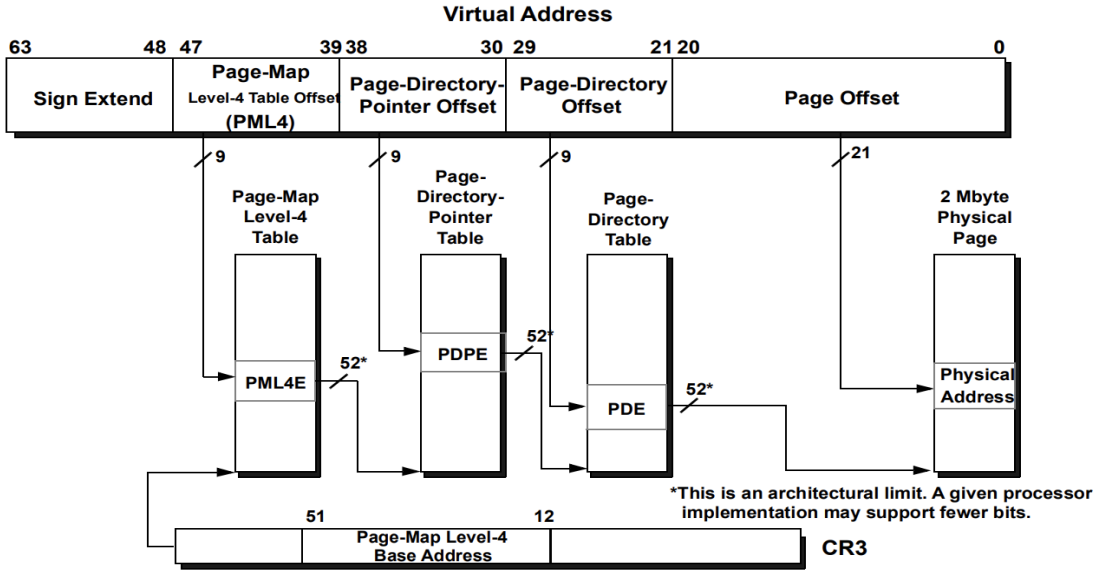
Paging

כחלק מאתחול המערכת, נצטרך לדאוג לאתחל את ה־page tables על מנת להיכנס ל־long mode. אני אגיד כבר עכשיו - ה־hypervisor מממש את מנגנון ה־paging רק מכיוון שהוא מוכרח לעשות זאת. אנחנו לא הולכים להתעסק עם process־ים ו־thread־ים. על כל ליבה של המעבד ה־hypervisor רץ כ־process יחיד ו־thread יחיד, כך שאנחנו לא באמת מתכוונים לנצל את החלוקה לזיכרון וירטואלי.

יתר על כן, אנחנו נשתמש ב־page table שעושה מיפוי בהיסט 0 - כלומר כל כתובת וירטואלית תתורגם בסופו של יום לאותה הכתובת הפיזית בזיכרון (הכתובת הוירטואלית 0x1000 תפנה אותנו לכתובת הפיזית 0x1000). מכיוון שה־hypervisor הוא process יחיד, וה־page tables נשמרים פעם אחת בלבד לזיכרון, אנחנו לא נצטרך לטעון את הדפים לדיסק ולממש מנגנון של page faults.

כל ה־page tables יהיו בזיכרון באופן תמידי. לכן, אנחנו נשתמש בדפים של 2MB ולא של 4KB, וזאת על מנת לחסוך בזיכרון שה־hypervisor משתמש בו, היות וחלוקה לדפים של 4KB מגדילה משמעותית את כמות הזיכרון שה־hypervisor ישתמש בו (זכרו שהטבלאות עצמן גם הן נמצאות בזיכרון, וללא מימוש מנגנון swap in/out - טבלאות המיפוי נמצאת בזיכרון באופן תמידי).

בואו נסתכל כיצד מתבצע תרגום של כתובת וירטואלית לפיזית, כאשר משתמשים בדפים של 2MB:



[קרדיט: AMD64 developer manual, Volume 2]

שימו לב שלמעשה אין שימוש ב־page tables, אלא ה־entries ב־page directory מצביעים ישירות אל דפים בגודל 2MB בזיכרון. אתם ודאי תוהים כיצד כל entry ב־page directory "יודע" האם הוא מצביע ל־page table רגיל או שמא ל־page בגודל 2MB.

התשובה לשאלה הזאת נמצאת בביט מספר 7 ב־page directory entry - במידה והוא דולק, ה־entry הנוכחי מצביע לדף בגודל 2MB. במידה והוא כבוי, ה־entry הנוכחי מצביע על page table המכיל entries שהם מצביעים לדפים בגודל 4KB.

לפני שנקפוץ לפונקצית C שתתחיל בתהליך אתחול ה־hypervisor, נציג את הקוד שמכין את טבלאות ה־paging בהם ה־hypervisor משתמש בעת אתחול המערכת.

בקוד שלפנינו אנחנו יכולים לראות שבכתובת 0x17000 (שנבחרה באופן שרירותי) מתחיל ה־PML4. מכיוון שכל entry בטבלה הזו ממפה 512GB של זיכרון פיזי (PML4), נסתפק ב־entry אחד בלבד שמצביע ל־Page directory pointer table שנמצא בכתובת 0x18000.

כל entry ב־page directory pointer table ממפה 1GB של זיכרון פיזי, לכן לא נצטרך באמת לבצע מיפוי עבור כל 512 ה־entries בטבלה (אלא אם יש לכם 512GB של RAM במחשב), אלא נסתפק במיפוי לפי כמות ה־RAM במחשב שלנו. המשתנה COMPUTER_MEM_SIZE מחזיק בתוכו את כמות ה־RAM ב־GB שיש במחשב. כל ערך מתחת ל־513 לא יצריך שינוי של אף לא שורת קוד אחת.

(הערה ביניים: קיצרתי כאן קצת על מנת להתמקד בעיקר ולא בטפל. למעשה, יש אתחול של paging עוד לפני שקפצנו ל־long mode, ומה שמוצג כאן זה לא המימוש המלא. מי שמעוניין במימוש המלא, יכול להתרשם ממנו ב־github למטה).

```
MovQwordToAddressLittleEndian 0x17000, 0x0, 0x18003
mov eax, 0x19000
mov edi, 0x18000
mov ecx, COMPUTER_MEM_SIZE ; If we have 16GB of RAM, this value will be
16
.setup_pdpt:
    mov edx, eax
    or edx, 3
    mov dword [edi], edx
    mov dword [edi + 4], 0
    add eax, 0x1000
    add edi, 8
    loop .setup_pdpt

mov ecx, COMPUTER_MEM_SIZE
shl ecx, 9                      ; multiply by 512
xor eax, eax                    ; start address is 0
mov ebx, ((1 << 7) | (1 << 1) | 1) ; bit 7 ON indicates a large page
mov edi, 0x19000
.setup_pds:
    or eax, ebx
    mov dword [edi], eax
    mov dword [edi + 4], 0
    add edi, 8
    add eax, LARGE_PAGE_SIZE
    loop .setup_pds
```

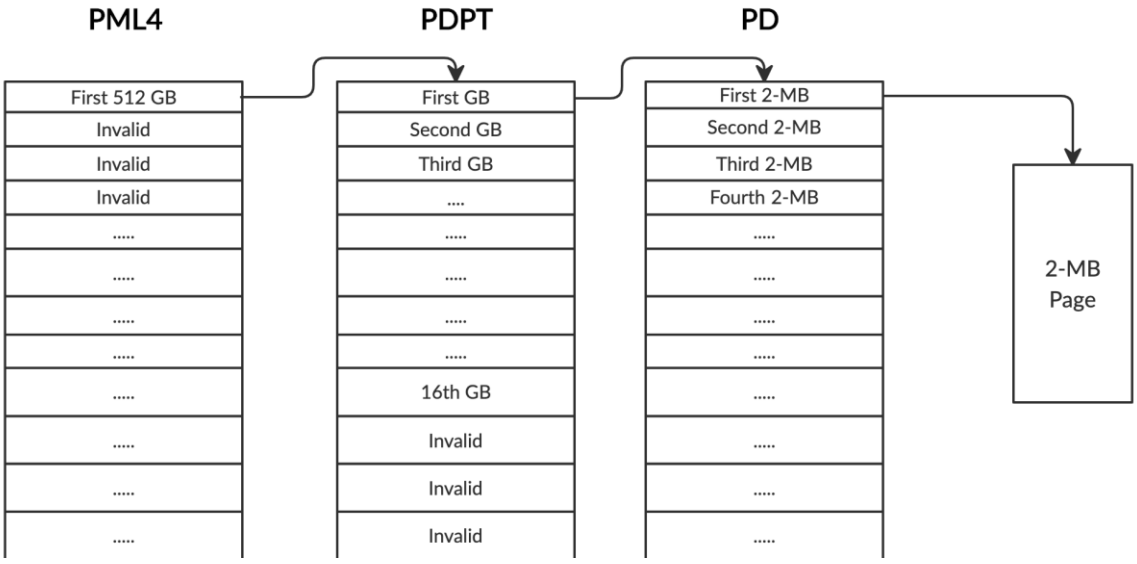
זיכרון וירטואלי - אולי בכל זאת?

כמו שאמרתי, לבצע מיפוי ישיר בין הכתובת הוירטואלית 0x1000 לכתובת הפיזית 0x1000 לחלוטין עובד וגם עונה על כל הצרכים שלנו. אבל בכל זאת, על מנת להוכיח לעצמנו שאנחנו מבינים כיצד עובד הזיכרון הוירטואלי ומנגנון ה־paging - אנחנו יכולים לבצע מיפוי בהיסט X.

למה הכוונה? אנחנו יכולים להחליט שאנחנו ממפים את הזיכרון הפיזי באופן לינארי, כך שכל כתובת וירטואלית X+Y תפנה לכתובת הפיזית Y.

לצורך הדוגמא, נניח שאנחנו מבצעים מיפוי בהיסט 0xFFFF000000000000. במצב זה, הכתובת הפיזית 0x1000 תמופה לכתובת הוירטואלית 0xFFFF000000001000.

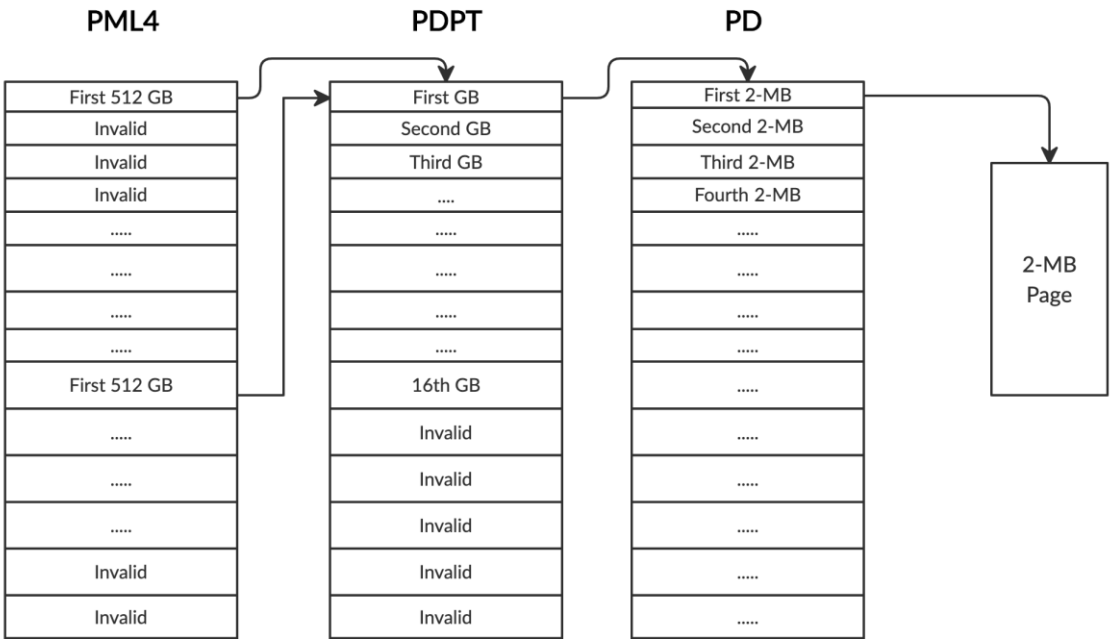
על מנת ליצור מיפוי שכזה, נסתכל על טבלאות המיפוי כפי שהן נראות כרגע:



הקוד למעלה יוצר entry בודד ב־PML4 שימפה את 512GB הראשונים. שאר הערכים ב־PML4 אינם ואלידיים (valid).

נוכל לנצל את העובדה הזאת, ולהחליט שאחד ה־entries בטבלה הזאת יבצע מיפוי לאותו ה־PDPT שמצביע עליו ה־entry הראשון.

זה בעצם יראה כך:



אם כך, מה הוא ה־offset הוירטואלי? כלומר מה הוא ההיסט X שיש להוסיף לכתובת פיזית Y על מנת להגיע לכתובת הוירטואלית הנכונה?

התשובה לשאלה הזאת תלויה בתא בו בחרנו ב־PML4. נניח שבחרנו את התא ה־200 שיבצע את המיפוי הכפול (כלומר התא ה־200 מצביע גם הוא ל־512GB הראשונים, בדומה לשרטוט למעלה). על פי השרטוט שצירפתי מה־manual של AMD, אנחנו יודעים שביטים 39 עד 47 בכל כתובת וירטואלית מוקצים לאינדקס ב־PML4.

לכן, נצטרך לתרגם את 200 לבינארי, למלא בביטים 39 עד 47 את הערך שנקבל, וכל שאר הביטים בכתובת הוירטואלית יהיו אפסים. אם נעשה זאת, נקבל את הערך 0x640000000000. כלומר לכל כתובת פיזית X, הכתובת ה"וירטואלית" המתאימה לו היא X + 0x640000000000.

טוב די, סבלנו מספיק. אחרי שיש לנו את כל זה, הגיע הזמן לקרוא לפונקצית C ושם יתחיל כל האקשן. מעתה והלאה, רוב העבודה תיעשה ב־C, אם כי נצטרך לחזור לאסמבלי כמה פעמים נוספות. אסביר בהמשך בדיוק לאילו צרכים.

מבני הנתונים של ה־Hypervisor

כל מי שעקב עד כאן יכול לטפוח לעצמו על השכם. ללא קשר ל־hypervisor או לוירטואליזציה בכלל - אתם עכשיו יודעים להעלות את המחשב ולהריץ את הקוד שלכם ב־long mode. זה אומר שאנחנו יכולים להתחיל לעבוד על השלב הבא בתרשים, שהוא הקצאת זיכרון לשימוש ה־hypervisor.

רגע לפני ההקצאה, בואו נדבר על מבני הנתונים שה־hypervisor משתמש בהם, ועל האופן בו הוא פועל. לפני כמה עמודים זרקתי מילה על זה שה־hypervisor רץ כ־process יחיד עם thread יחיד על כל יחידת עיבוד (ליבה). למעשה, אנחנו צריכים לשנות את החשיבה שלנו להיות "מכוונת מעבד". למה אני מתכוון? ובכן, לכל יחידת עיבוד יש סט רגיסטרים משלה, מה שאומר שלכל ליבה יש state משלה. ה־state הזה כולל בין היתר page tables שונים לכל ליבה, global descriptor table ייחודי, extended page tables ייחודיים (לא לדאוג, יהיה לנו עוד פרק שלם בהמשך שיעסוק בזה) והכי חשוב - מבני נתונים ייחודיים שנדרשים לניהול מנגנון הוירטואליזציה על ידי המעבד.

Basically, מדובר באזורים בזיכרון שאנחנו אומרים למעבד "קח, אתה מנהל אותם ומשתמש בהם, אני רק קורא וכותב אליהם דרך ממשק שאתה נותן לי". האזורים האלו כוללים את ה־VMXON region ואת ה־ VMCS region. על הראשון אכתוב בקצרה בהמשך, ועל השני אנחנו נתעכב עוד לא מעט. כרגע אני רק רוצה להציג את מבני הנתונים האלו as is, אסביר על מה שאנחנו מכירים יודעים, ומה שאנחנו לא יודעים - עוד נדון עליו. הסירו דאגה מלבכם.

SINGLE_CPU_DATA: מבנה נתונים לשמירת מידע שרלוונטי לכל ליבה בנפרד (לכל ליבה יש עותק ייחודי של ה־struct הזה):

```
typedef struct _SINGLE_CPU_DATA
{
   BYTE vmxon[PAGE_SIZE];
   BYTE vmcs[PAGE_SIZE];
   BYTE msrBitmaps[PAGE_SIZE];
   BYTE stack[STACK_SIZE];
   QWORD pageMapLevel4s[ARRAY_PAGE_SIZE]; // cr3
   QWORD pageDirectoryPointerTables[ARRAY_PAGE_SIZE]; // 1 PML entry = 512GB,
enough for the HV
   QWORD pageDirectories[ARRAY_PAGE_SIZE * ARRAY_PAGE_SIZE]; // HV will use 2MB
pages
   QWORD eptPageMapLevel4s[ARRAY_PAGE_SIZE];
   QWORD eptPageDirectoryPointerTables[ARRAY_PAGE_SIZE];
   /* Each 512 entries here are referenced by an entry from the previous table.
       Each entry in the previous table maps 1GB of memory */
   QWORD eptPageDirectories[ARRAY_PAGE_SIZE * COMPUTER_MEM_SIZE];
   QWORD eptPageTables[ARRAY_PAGE_SIZE * ARRAY_PAGE_SIZE * COMPUTER_MEM_SIZE];
   BYTE coreIdentifier;
   QWORD gdt[0xff];
   PSHARED_CPU_DATA sharedData;
} SINGLE_CPU_DATA, *PSINGLE_CPU_DATA;
```

- **vmxon** - מבנה נתונים המנוהל על ידי יחידת העיבוד הנוכחית, ואנחנו מעבירים למעבד את הכתובת הפיזית שלו כאשר נבצע "הדלקה" של ה־hypervisor (יפורט בהמשך).
- **vmcs** - מבנה נתונים המנוהל על ידינו ומשמש את המעבד לניהול מנגנון הוירטואליזציה. בהמשך נדבר עליו בהרחבה (בלשון המעטה).
- **msrBitmaps** - נרחיב עליו בחלק 2 של המאמר.
- **stack** - לכל ליבה יש stack ייחודי משלה, וזה הוא בדיוק הערך שיהיה ב־rsp כאשר המערכת רצה ב־context של ה־hypervisor.
- **pageMapLevel4s** - טבלת ה־PML4 עבור הליבה הנוכחית. כפי שפירטתי למעלה, רק השדה הראשון בטבלה זו נמצא בשימוש.
- **pageDirectoryPointerTables** - טבלאות ה־Page Directory Pointer Tables עבור הליבה הנוכחית. כפי שפירטתי למעלה, רק 16 שדות יהיו בשימוש כאן, אם יש לנו 16GB של RAM במחשב.
- **pageDirectories** - טבלאות ה־Page Directories המשמשות את הליבה הנוכחית. כל entry בטבלה שכזו מצביע ישירות ל־page בזיכרון בגודל 2MB.
- **eptXXXXXXX** - נרחיב על משתנים אלו לקראת סוף החלק הראשון של המאמר.
- **coreIdentifier** - לכל ליבה במערכת יש מזהה ייחודי שהיא מקבלת בזמן עליית המערכת כאשר ה־BIOS מבצע ואלידציה לרכיבי החומרה השונים. בהמשך נראה כיצד ניתן למצוא את המזהה הייחודי של כל ליבה.
- **gdt** - ה־Global Descripter Table המשוייך לליבה הנוכחית.
- **sharedData** - מצביע למבנה נתונים המאחסן מידע המשותף לכלל הליבות במערכת. כל ליבה שומרת מצביע למבנה נתונים זה.

SHARED_CPU_DATA: מבנה נתונים גלובאלי לשמירת מידע שזהה עבור כל הליבות המערכת:

```
typedef struct _SHARED_CPU_DATA
{
   struct _SINGLE_CPU_DATA* cpuData[MAX_CORES];
   struct _CURRENT_GUEST_STATE* currentState[MAX_CORES];
   E820_LIST_ENTRY validRam[E820_OUTPUT_MAX_ENTRIES];
   BYTE validRamCount;
   E820_LIST_ENTRY allRam[E820_OUTPUT_MAX_ENTRIES];
   BYTE memoryRangesCount;
   BYTE numberOfCores;
   BYTE_PTR hypervisorBase;
   QWORD physicalHypervisorBase;
   QWORD hypervisorBaseSize;
   BYTE_PTR codeBase;
   QWORD physicalCodeBase;
   QWORD codeBaseSize;
   QWORD int15Segment;
   QWORD int15Offset;
} SHARED_CPU_DATA, *PSHARED_CPU_DATA;
```

- **cpuData** - מערך של מצביעים ששומר את הכתובות של המבנים המחזיקים את המידע הרלוונטי לליבה ספיציפית.
- **currentState** - מערך של מצביעים ששומר את הכתובות של המבנים המחזיקים את המידע הרלוונטי בעת התרחשות vm-exit. נסביר עליהם בעוד רגע.
- **validRam** - רשימה של טווחי זיכרון שמישים. בהמשך ארחיב על טווחי הזיכרון ואיך אפשר לקבל את הרשימה שלהם, ואסביר כמובן למה חשוב לנו לשמור את מידע זה.
- **allRam** - רשימה של כל טווחי הזיכרון המערכת - אלו ששימישים ואלו שאינם שמישים. רשימה זו בהכרח מכילה גם את הרשימה הקודמת.
- **numberOfCores** - משתנה המחזיק את מספר הליבות במערכת.
- **hypervisorBase** - מחזיק את הכתובת הוירטואלית שבה ה־hypervisor מאחסן את הנתונים שלו.
- **physicalHypervisorBase** - מחזיק את הכתובת הפיזית שבה ה־hypervisor מאחסן את הנתונים שלו.
- **codeBase** - מחזיק את הכתובת הוירטואלית שבה נמצא הקוד של ה־hypervisor.
- **physicalCodeBase** - מחזיק את הכתובת הפיזית שבה נמצא הקוד של ה־hypervisor.
- **int15Segment** - נדבר עליו בהמשך.
- **int15Offset** - כנ"ל.

CURRENT_GUEST_STATE: מבנה נתונים לשמירת המידע הרלוונטי בעת ביצוע vm-exit:

```
typedef struct _CURRENT_GUEST_STATE
{
   REGISTERS guestRegisters;
   PSINGLE_CPU_DATA currentCPU;
} CURRENT_GUEST_STATE, *PCURRENT_GUEST_STATE;
```

וכך נראה **REGISTERS**:

```
typedef struct _REGISTERS
{
   QWORD rax;
   QWORD rbx;
   QWORD rcx;
   QWORD rdx;
   QWORD rsi;
   QWORD rdi;
   QWORD rbp;
   QWORD r8;
   QWORD r9;
   QWORD r10;
   QWORD r11;
   QWORD r12;
   QWORD r13;
   QWORD r14;
   QWORD r15;
   QWORD rsp;
   QWORD rip;
}REGISTERS, *PREGISTERS;
```

- **guestRegisters** - מדובר במשתנה שתפקידו לשמור את כל ה־register-ים של ה־guest ברגע שהתרחש vm-exit. כלומר, בכל פעם שהתרחש vm-exit, הדבר הראשון שנעשה הוא לשמור את האוגרים במבנה הזה. כמו כן, בכל פעם שאנחנו נחזור ל־guest ונבצע vm-entry, אנחנו נדאג לקחת את הערכים מבמבנה הזה ולהעביר אותם לרגיסטרים, כדי שה־guest יעודכן בערכים ששינינו. בהמשך נראה כיצד בדיוק מתבצע תהליך הטעינה.

- **currentCPU** - משתנה זה מצביע למבנה מסוג SINGLE_CPU_DATA המשוייך לליבה הנוכחית עליה הקוד רץ. כמו שהסברתי קודם, יש מופע "נפרד" של ה־hypervisor על כל ליבה של המעבד, והמשתנה הזה אחראי להצביע על הליבה הנוכחית בה אנחנו רצים.

הקצאת זיכרון עבור מבני הנתונים של ה־hypervisor

מה שאני עושה עכשיו, הוא לדמיין אתכם שואלים - "אבל רגע, למה הכוונה 'הקצאת זיכרון'? הרי אין פה שום malloc, new, realloc, calloc או כל דבר אחר. המערכת הרגע עלתה, ואין לנו שום ספרייה או קוד להשתמש בו". הדבר הבא שאני עושה הוא להחמיא לכם על השאלה הנכונה והחכמה ששאלתם, ואז אני עונה: מערכת ההפעלה וה־hypervisor שלנו יושבים על אותה החומרה, ובסופו של דבר משתפים את ה־RAM. אנחנו צריכים לדאוג שכל מבני הנתונים וגם הקוד של ה־hypervisor יהיו נסתרים ממערכת ההפעלה, כדי שהיא לא תדרוס אותם חלילה.

ציינתם גם שאין לנו איך להקצות זיכרון, מכיוון שהקוד שלנו הוא היחיד שקיים כרגע על המחשב. זה נכון, אבל רק חלקית. כדי לענות על השאלה ששאלתם, הגיע הזמן לדבר על ה־BIOS.

הערה קטנה לפני כן: מי שכבר נכנס לעניינים, הבין שגודל ההקצאה הוא פונקציה של מספר הליבות (כלומר, ככל שמספר הליבות במעבד גדול יותר - כך נצטרך להקצות יותר זיכרון, שהרי יש צורך לשמור עותק של SINGLE_CPU_DATA עבור כל ליבה בנפרד). בהמשך אני מסביר בדיוק איך לזהות את מספר הליבות במחשב על מנת לדעת כמה מקום להקצות.

ה־BIOS

אני מניח שכמעט כולם שמעו על המושג הזה בעבר. משמעותו Basic Input Output System, והוא מנגנון חומרה-תוכנה המשתמש לאתחול המחשב עוד לפני שמערכת הפעלה או כל קוד אחר רץ מתחיל לרוץ. מעבר לאתחול רכיבי החומרה, ה־BIOS דואג לאחסן באזורים ידועים מראש בזיכרון מידע אודות רכיבי החומרה (למשל - כמה ליבות יש למעבד המותקן על המחשב).

כמו כן, הוא מספק אוסף של פסיקות (Interrupt-ים) אשר יכולות לשמש את מערכת ההפעלה לביצוע מגוון פעולות שונות. פסיקות אלו זמינות ב־real address mode בלבד. במהלך המאמר אציג שתי פסיקות שה־hypervisor עושה בהן שימוש. עתה נדבר על אחת מהן.

Int 15h

זו היא הפסיקה הראשונה שה־hypervisor יעשה בה שימוש. פסיקה זו משמשת לביצוע של מגוון פעולות, והפעולה שתתבצע נקבעת על פי הערך הנקבע באוגר eax בעת ביצוע הקריאה ל־int 15h. מבין על הערכים האפשריים בעת ביצוע הקריאה, אותנו מעניין הערך 0xE820. הערך הזה מבקש מה־BIOS לקבל את מפת הזיכרון הפנוי כפי שהוא מגדיר אותה. אני מניח שחלקכם לא מבינים, ושואלים למה בכלל צריך את הפסיקה הזאת? הרי כשהמחשב עולה, כל הזיכרון מוגדר כפנוי, לא? אז זהו, שלא. ה־BIOS מגדיר חמישה סוגים שונים של זיכרון:

- זיכרון שמיש
- זיכרון שמור - לא שמיש
- זיכרון ACPI reclaimable
- זיכרון ACPI NVS
- זיכרון פגום

מה שמעניין אותנו הוא הסוג הראשון בלבד - זיכרון שמיש. אנחנו נבקש מה־BIOS את מפת הזיכרון במערכת ונסרוק את הרשימה שתתקבל. מתוך הרשימה הזאת, נמצא טווח זיכרון שמיש גדול מספיק שיכול להכיל את כל מבני הנתונים שה־hypervisor מחזיק.

מהגודל של טווח הזיכרון הזה נדאג להחסיר את הגודל של של כלל מבני הנתונים שלנו, ונשמור את הכתובת בה נתחיל לאחסן את אותם מבני הנתונים. מאוחר יותר, כשנעלה את מערכת ההפעלה והיא בתורה תבקש את מפת הזיכרון מה־BIOS, ה־hypervisor ידאג לספק למערכת ההפעלה תשובה "מזויפת" - הוא יחזיר את מפת הזיכרון שאנחנו ערכנו.

מערכת ההפעלה לא תשאל יותר מדי שאלות, ומכיוון שהזיכרון שלנו לא מסומן כשמיש (ליתר דיוק, הוא לא מופיע כלל במפת הזיכרון) היא לא תבצע לו מיפוי - וכך בעצם יצרנו לעצמנו אזור שלם בזיכרון שהוא רק שלנו. מערכת ההפעלה לא תעז להתעסק עם ה־ BIOS, שהוא ה־bad boy של השכונה. היא תציית לתשובה שהוא העביר לה ותבצע מיפוי אך ורק לזיכרון שמוגדר כ־valid-י.

בהתאם למה שרשמתי פה, אנחנו מבינים בעת עליית מערכת ההפעלה נצטרך לדאוג למנגנון hooking של הפסיקה הזאת (int 15h). נדבר על כך מאוחר יותר בחלק 2, לאחר שנאתחל את כלל מבני הנתונים. זאת תהיה דוגמא מצוינת לכוח של הוירטואליזציה. (איזו התרגשות!)

שימוש ב־int 15h

עכשיו כשאנחנו יודעים מה הפסיקה עושה ולמה אנחנו רוצים להשתמש בה, נותר רק להבין איך היא פועלת. בואו נצא לדרך.

קודם לכן הזכרתי שהפסיקה הזאת מחזירה רשימה של טווחי זיכרון, כאשר כל טווח זיכרון ממופה לאחת מחמשת סוגי הזיכרון למעלה. על מנת לקבל את הרשימה הזאת, יש לקרוא לפסיקה שוב ושוב עד שהיא תגיד לנו "חבר, זהו. נתתי לך את טווח הזיכרון האחרון. תפסיק להציק לי".

בתור התחלה, נשים באוגר eax את הערך 0xE820; נכוון את es:di לכתובת של ה־buffer אליו הפסיקה תכתוב את מפת הזיכרון; נשים באוגר edx את הערך 0x534D4150; נשים באוגר ecx את הערך 24; נבצע int 15h. אם הקריאה הצליחה, באוגר edx ישאר הערך 0x534D4150 וה־carry flag יהיה 0.

לאחר הקריאה הראשונה, נבצע הלולאה את הפסיקה שוב ושוב, ובכל פעם נשים באוגר eax את הערך 0xE820; נשים באוגר ecx את הערך 24; נקדם את di בגודל כל איבר ברשימה (24); לא נשנה את הערך שב־ebx בשום פנים ואופן. למעשה, ebx מתפקד בתור אינדקס ברשימה (ה־BIOS יחזיר לנו את הערך במקום ה־ebx). ואיך נדע שהגענו לסוף הרשימה? במידה והגענו לסוף, אחד מההבאים יקרה: הפסיקה תחזור עם carry flag דולק, או ש־ebx יתאפס. אנחנו נבדוק את כל אחד מהתנאים הללו לפני שנבצע שוב את הפסיקה. את הקוד אני לא אצרף כאן, אבל אתם כמובן מוזמנים להיכנס לאתר osdev, שם יש מימוש מלא לכל התהליך הזה שתיארתי.

אני הרצתי את כל התהליך והדפסתי (רגע, מה?! להדפיס?! אבל אמרת אין פונקציה שמאפשרת הדפסה! אין ספריות, אין מערכת הפעלה, איך עשית את זה?? התשובה לשאלה נמצאת כאן) את התוצאה. אני מצרף את 5 ה־entries הראשונים ברשימה, כפי שהתקבלו במכונה וירטואלית בה הרצתי את הפסיקה:

```
######### 0
Base address: 0000000000000000
Length: 000000000009F400
Type: 1
######### 1
Base address: 000000000009F400
Length: 0000000000000C00
Type: 2
######### 2
Base address: 00000000000CE000
Length: 0000000000002000
Type: 2
######### 3
Base address: 00000000000DC000
Length: 0000000000024000
Type: 2
######### 4
Base address: 0000000000100000
Length: 00000000BFDE0000
Type: 1
######### 5
Base address: 00000000BFEE0000
Length: 000000000001F000
Type: 3
```

עכשיו, אחרי שיש לנו את מפת הזיכרון, אנחנו נמצא לנו טווח זיכרון ונשתלט על חלק ממנו.

נתחיל לסרוק את הרשימה שהחזיר לנו E820 ונעצור כאשר מצאנו טווח זיכרון מספיק גדול שיכול לאכלס את כל הזיכרון שה־hypervisor צריך:

```
STATUS AllocateMemoryUsingMemoryMap
   (IN PE820_LIST_ENTRY memoryMap, IN DWORD memoryRegionsCount, IN QWORD
allocationSize, OUT BYTE_PTR* address)
{
   QWORD alignedAllocationSize = ALIGN_UP(allocationSize, LARGE_PAGE_SIZE);
   INT upperIdx = NEG_INF; // high addresses are more rarely to be in use on the
computer startup
   for(QWORD i = 0; i < memoryRegionsCount; i++)
   {
       if(memoryMap[i].type != E820_USABLE_REGION)
           continue;
       if(memoryMap[i].length <= allocationSize)
           continue;
       if((INT)i > upperIdx)
           upperIdx = i;
   }
   if(upperIdx == NEG_INF)
       return STATUS_NO_MEM_AVAILABLE;

   QWORD unalignedCountBase = memoryMap[upperIdx].baseAddress % PAGE_SIZE;
   QWORD unalignedCountLength = memoryMap[upperIdx].length % PAGE_SIZE;
   memoryMap[upperIdx].length -= (alignedAllocationSize + unalignedCountBase +
unalignedCountLength);
   *address = memoryMap[upperIdx].baseAddress + memoryMap[upperIdx].length;
   return STATUS_SUCCESS;
}
```

שימו לב שאנחנו סורקים את הרשימה גם לאחר שנמצא מקטע כזה, במטרה למצוא את המקטע האחרון שמתאים מבחינת הגודל.

מדוע אנחנו עושים זאת? ובכן, כאשר מערכת ההפעלה עולה, עוד לפני שמנגנון ה־paging מופעל ולפני שהיא מבצעת בכלל את E820, היא מתחילה להריץ קוד בכתובות נמוכות במערכת (פשוט מאוד מכיוון שהיא מוגבלת ל־20 ביטים בלבד ב־real mode).

הקוד הזה דואג לאתחול בסיסי של המערכת ואנחנו חייבים לדאוג שגם בשלב זה מערכת ההפעלה לא תדרוס המידע שלנו. אם מראש ניגש לכתובות הגבוהות ונדאג להשתמש בהן, אנחנו יכולים להרגיע את הדאגות בראשנו מכיוון שלמערכת ההפעלה אין גישה לכתובות האלו בעת עליית המערכת.

עוד משהו קטן שאנחנו עושים זה לוודא שה־base address שלנו הוא page aligned - כלומר הוא כפולה של 0x1000. זה נובע בעיקר מסיבות טכניות שלא אפרט עליהן עכשיו.

אתחול ה־Hypervisor

אוקיי, אנחנו כבר בשלב השלישי בתהליך. הגיע הזמן לאתחל את מבני הנתונים של ה־hypervisor ואת ה־hypervisor עצמו. הדבר הראשון שאני אעשה זה לעשות את הקישורים הפשוטים מאוד בין SINGLE_CPU_DATA ו־SHARED_CPU_DATA, כך שלכל ליבה יהיה מצביע ל־SHARED_CPU_DATA דרך ה־SINGLE_CPU_DATA שמשויך לה.

כמו כן, נבצע קישור בין CURRENT_GUEST_STATE ל־SINGLE_CPU_DATA, כך שכל מופע של CURRENT_GUEST_STATE יצביע ל־SINGLE_CPU_DATA.

הדבר האחרון שאנחנו דואגים לו הוא לאכלס את המשתנה coreIdentifier שנמצא במבנה המיועד לכל ליבה של המעבד. כאשר המערכת עולה, כל ליבה מקבלת ID ייחודי שיזהה אותה. באופן default-י, ה־IDs מתחילים מ־0 עם קפיצות של 1. בעוד רגע אציג דרך לזיהוי מספר הליבות והמזהים היחודיים:

```
for(BYTE i = 0; i < numberOfCores; i++)
{
    sharedData->cpuData[i] = hypervisorBase
      + ALIGN_UP(sizeof(SHARED_CPU_DATA), PAGE_SIZE)
      + i * ALIGN_UP(sizeof(SINGLE_CPU_DATA), PAGE_SIZE);

    sharedData->currentState[i] = hypervisorBase
      + ALIGN_UP(sizeof(SHARED_CPU_DATA), PAGE_SIZE)
      + numberOfCores * ALIGN_UP(sizeof(SINGLE_CPU_DATA), PAGE_SIZE)
      + i * ALIGN_UP(sizeof(CURRENT_GUEST_STATE), PAGE_SIZE);

    sharedData->currentState[i]->currentCPU = sharedData->cpuData[i];
    sharedData->cpuData[i]->sharedData = sharedData;
    sharedData->cpuData[i]->coreIdentifier = processorIdentifires[i];
}
```

זיהוי מידע אודות רכיבי החומרה

קודם לכן אמרתי שנצטרך לדעת מה הוא מספר הליבות המדויק שיש במחשב על מנת לקבוע את הגודל הכולל של ההקצאה. כעת נלמד בדיוק איך למצוא את מספר הליבות, וגם את המזהה הייחודי של כל ליבה.

ה־RSDP וה־RSDT

ה־(Root System Descriptor Pointer) RSDP הוא מצביע למבנה נתונים שמאכלס טבלאות חשובות אודות רכיבי החומרה. למבנה הנתונים הזה קוראים (Root System Descriptor Table) RSDT והוא מאכלס בין היתר טבלה שמכילה מידע אודות המעבד.

ה־RSDP נראה כך, וסימנתי את השדה שמעניין אותנו:

```
struct RSDPDescriptor {
 char Signature[8];
 uint8_t Checksum;
 char OEMID[6];
 uint8_t Revision;
 uint32_t RsdtAddress;
} __attribute__ ((packed));
```

[צילום: osdev]

הגורם שאחראי על אתחול ה־RSDP בעל עליית המערכת הוא ה־BIOS, כמובן. הוא ממוקם באחד משני מקומות:

- ב־1KB הראשונים של ה־EBDA (Extended BIOS Data Area). בכתובת הסטטית 0x40E יש מצביע לאיזור זה.
- במרחב הכתובות שבין 0xE0000 ו־0xFFFFF.

על מנת לזהות את המבנה הזה, נסרוק את מרחבי הכתובות הללו ונחפש את התבנית "RSD PTR " (שימו לב לרווח בסוף).

```
STATUS FindRSDT(OUT BYTE PTR* address, OUT QWORD PTR type)
{
   CHAR pattern[] = "RSD PTR ";
   BYTE_PTR ebdaAddress = (*(WORD_PTR)EBDA_POINTER_ADDRESS) >> 4;
   BYTE_PTR rsdpBaseAddress;

   for(QWORD i = 0; i < 0x1024; i += 16)
   {
       if(!CompareMemory(ebdaAddress + i, pattern, 8))
       {
           rsdpBaseAddress = (ebdaAddress + i);
           goto RSDPFound;
       }
   }
   for(QWORD start = 0xE0000; start < 0xFFFFF; start += 16)
   {
       if(!CompareMemory(start, pattern, 8))
       {
           rsdpBaseAddress = start;
           goto RSDPFound;
       }
   }
   return STATUS_RSDP_NOT_FOUND;
RSDPFound:
   ...
   ...
   ...
}
```

לאחר שמצאנו את ה־RSDP ודרכו הגענו ל־RSDT, הגיע הזמן לקבל מידע אודות רכיבי החומרה. מבלי להיכנס ליותר מדי פרטים, אחת הטבלאות ב־RSDT נקראת MADT, קיצור של Multiple APIC Description Table. זאת הטבלה שמעניינת אותנו, והיא מכילה בדיוק את המידע שחשוב לנו - מספר ליבות ומזהים ייחודיים עבור כל ליבה.

למי שמעוניין במבנה הטבלה, ניתן לקרוא עוד בקישור הזה. מה שאני עשיתי הוא לקחת את המידע שנמצא שם ולהכניס למשתנה coreIdentifier שהצגתי למעלה. עתה כל מבנה מסוג SINGLE_CPU_DATA מכיל את המזהה הייחודי של כל ליבה.

אתחול מנגנון הוירטואליזציה

אחרי שהכנו את כל מה שדרוש וכל הקרקע כבר מוכנה, הגיע הזמן "להפעיל" את ה־hypervisor. מי שיגלול למעלה לתרשים הראשון ששמתי מתוך ה־manual של Intel יראה שתי פקודות אסמבלי שרשומות שם: VMXON ו־VMXOFF. הפקודות האלה אחראיות על "להדליק" ו־"לכבות" את ה־hypervisor. לפני שנריץ את הפקודה VMXON ונדליק את ה־hypervisor, אנחנו חייבים לוודא שכל דרישות הקדם של הפקודה VMXON נענו.

ראשית, עלינו ליצור אזור בזיכרון בגודל Page אחד (0x1000) המשמש את המעבד לשימושים פנימיים שלו. לאזור זה קוראים VMXON region, והוא ייחודי לכל ליבה. לכן, נשמור אותו במבנה הנתונים שמאכלס את המידע של ליבה יחידה (SINGLE_CPU_DATA). ארבעת הבתים הראשונים של אזור זה צריכים להיות זהים לארבעת הבתים הראשונים של ה־MSR בשם IA32_VMX_BASIC. נאתחל זאת כך:

```
*(DWORD_PTR)(cpuData->vmxon) = (DWORD)__readmsr(MSR_IA32_VMX_BASIC);
```

שנית - מנגנון הוירטואליזציה דורש ביטים מסוימים דוקלים ב־CR0 וב־CR4. איך נדע אילו ביתים הוא דורש שיהיו כ־1 לפני הפעלת ה־hypervisor? באמצעות MSR בשם IA32_VMX_CR0/4_FIXED0. באופן דומה, הביטים הנדרשים להיות 0 מדווחים ע"י IA32_VMX_CR0/4_FIXED1:

```
__writecr0((__readcr0() | CR0_NE_ENABLED | __readmsr(MSR_IA32_VMX_CR0_FIXED0))
           & __readmsr(MSR_IA32_VMX_CR0_FIXED1));
__writecr4((__readcr4() | CR4_VMX_ENABLED | __readmsr(MSR_IA32_VMX_CR4_FIXED0))
           & __readmsr(MSR_IA32_VMX_CR4_FIXED1));
```

שימו לב שב־CR4 הביט המשויך להפעלת מנגנון הוירטואליזציה חייב להיות דולק (CR4_VMX_ENABLES), לכן דאגתי להפעיל אותו באופן מפורש.

זהו, הגיע הזמן להפעיל את ה־hypervisor. הפרמטר של הפקודה vmxon שאחראית על הפעלת המנגנון הוא הכתובת הפיזית של ה־vmxon region:

```
__vmxon(VirtualToPhysical(cpuData->vmxon));
```

בום! בזה הרגע, ממש עכשיו, תוך כדי שאתם קוראים את שורות אלו - הקוד ממשיך לרוץ "בתוך" ה־hypervisor. כלומר, המצב כרגע הוא כאילו התרחש vm-exit וה־hypervisor נכנס לפעולה. בעצם לאחר הרצת השורה הזאת, אנחנו רצים ב־context של ה־hypervisor (בשפה המקצועית: vmx-root operation).

הדבר הבא לעשות הוא לגשת ולקנפג (מהמילה configure) את ה־hypervisor ולספר לו על דברים חשובים כמו מה הכתובת של הפקודה הראשונה שצריכה לרוץ כאשר מתרחש vm-exit, מה הכתובת של ה־stack pointer כאשר מתרחש vm-exit, מה הם ה־selectors עבור ה־host, ה־GDT שלו ועוד אלף ואחד דברים. כדי לעשות את זה, נעבור לדבר על הגוף האחראי על החזקת ותחזוקת הקונפיגורציות האלו.

ה־VMCS

תחזיקו חזק, כי אנחנו הגענו לחלק החשוב ביותר וללב ליבה של הוירטואליזציה. אנחנו מתחילים לדבר על ה־Virtual Machine Control Structure. זה הוא מבנה נתונים שמחולק לוגית ל־3 אזורים, והוא מחזיק את כל המידע הרלוונטי למעבד בנוגע לה:

- ערכים הנקבעים בעת התרחשות vm-exit: מאחסן מידע המשמש את המעבד בעת התרחשות vm-exit. בין היתר הוא מדווח על איזה קוד צריך לרוץ בעת תרחיש כזה, איפה ה־stack צריך להיות ממוקם, היכן ממוקם ה־PML4 של ה־hypervisor ועוד.

- ערכים הנקבעים בעת התרחשות vm-entry: כאשר מתבצע אירוע של חזרה ל־guest (אותו אנו מכנים vm-entry), נצטרך לדעת מה הוא הערך החדש של ה־instruction pointer, של ה־ stack pointer, אוגר הדגלים (flags) ועוד ערכים נוספים.

- אופן הפעולה של ה־hypervisor: אילו פעולות צריכות לגרום ל־vm-exit? האם page-fault אמור לגרום ל־vm-exit? האם פסיקות debug אמורות לגרום ל־vm-exit? האם מותר לגשת לכתובת הפיזית xFFFF0123000012450 ולהריץ שם קוד? האם מותר לקרוא משם? לכתוב לשם? האם מותר לכתוב ל־MSR שמספרו x1780? האם מותר לקרוא ממנו? כל זאת ועוד מדווח ב־VMCS.

כדי לבצע "היכרות" או "שידוך" בין המעבד ל־VMCS, אנחנו נצטרך לשמור אזור בזיכרון בגודל page בודד שישמש לאחסון ה־VMCS (בדומה ל־VMXON). אנחנו נאחסן אותו במבנה הנתונים ששומר state של ליבה בודדת, מכיוון שכל ליבה יכולה להיות במצב שונה לחלוטין מאחרת. כמו כן, שוב בדומה ל־VMXON, ארבעת הבתים הראשונים צריכים להיות זהים ל־MSR ה־IA32_VMX_BASIC. אתחול ה־VMCS ורג'יסטרציה (register) שלו נראים כך:

```
*(DWORD_PTR)(cpuData->vmcs) = (DWORD)__readmsr(MSR_IA32_VMX_BASIC);
// VMCS status will now turn to cleared
__vmclear(VirtualToPhysical(cpuData->vmcs));
// Current & active
__vmptrld(VirtualToPhysical(cpuData->vmcs));
```

כתיבה וקריאה ל־VMCS

קריאה וכתיבה ל־VMCS לא תיעשנה באמצעות גישה מפורשת לזיכרון. כלומר אנחנו לא נכתוב לאזור זה באופן ישיר. אנחנו נשתמש בממשק של המעבד לכתיבה וקריאת ערכים ל־VMCS. כתיבת ערכים נעשית באמצעות הפקודה vmwrite, וקריאת ערכים נעשית באמצעות הפקודה vmread. בעוד רגע נראה מספר דוגמאות לכתיבה ל־VMCS.

אתחול ה־VMCS

לפני שנתחיל באתחול, אני רוצה לציין שלמרות שה־section הזה הולך להיות ארוך, זה הוא עדיין רק אתחול חלקי. על אתחול חלק מהשדות אני דילגתי מכיוון שהם פחות מעניינים או טריוויאלים. אל דאגה, תהיו בטוחים שאת החלקים המעניינים והקריטיים להבנה דאגתי לכלול כאן.

קדימה, בואו נסתכל על תרשים השדות ב־VMCS ונדבר על המשמעות של חלק מהם:

GUEST STATE AREA				
CR0	CR3			CR4
DR7				
RSP	RIP			RFLAGS
CS	Selector	Base Address	Segment Limit	Access Right
SS	Selector	Base Address	Segment Limit	Access Right
DS	Selector	Base Address	Segment Limit	Access Right
ES	Selector	Base Address	Segment Limit	Access Right
FS	Selector	Base Address	Segment Limit	Access Right
GS	Selector	Base Address	Segment Limit	Access Right
LDTR	Selector	Base Address	Segment Limit	Access Right
TR	Selector	Base Address	Segment Limit	Access Right
GDTR	Selector	Base Address	Segment Limit	Access Right
IDTR	Selector	Base Address	Segment Limit	Access Right
IA32_DEBUGCTL	IA32_SYSENTER_CS	IA32_SYSENTER_ESP	IA32_SYSENTER_EIP	
IA32_PERF_GLOBAL_CTRL	IA32_PAT	IA32_EFER	IA32_BNDCFGS	
SMBASE				
Activity state	Interruptibility state			
Pending debug exceptions				
VMCS link pointer				
VMX-preemption timer value				
Page-directory-pointer-table entries	PDPTE0	PDPTE1	PDPTE2	PDPTE3
Guest interrupt status				
PML index				
HOST STATE AREA				
CR0	CR3			CR4
RSP	RIP			
CS	Selector			
SS	Selector			
DS	Selector			
ES	Selector			
FS	Selector	Base Address		
GS	Selector	Base Address		
TR	Selector	Base Address		
GDTR	Base Address			
IDTR	Base Address			
IA32_SYSENTER_CS	IA32_SYSENTER_ESP		IA32_SYSENTER_EIP	
IA32_PERF_GLOBAL_CTRL	IA32_PAT		IA32_EFER	

Guest state area

אל אזור זה של ה־VMCS נטענים ערכים כאשר מתרחש vm-exit. למשל, אל התא המתאים ל־CR0 יטען הערך באוגר CR0 כאשר מתרחש vm-exit. ל־RIP במבנה זה יטען הערך של RIP כאשר מתרחש vm-exit. כמו כן, ערכים אלו נטענים חזרה ל־guest כאשר מתבצעת הפעולה של חזרה ל־guest, מה שכינינו קודם vm-entry.

נתחיל באתחול השדות של ה־guest. אתם ודאי זוכרים שעכשיו אנחנו כבר רצים ב־context של ה־hypervisor, משום שהרצנו את הפקודה vmxon. היינו רוצים שלאחר שנסיים לקנפג (או בעברית - לתצר) את ה־VMCS, אנחנו נחזור לרוץ ב־context של ה־guest ולא של ה־hypervisor (מכיוון שבהמשך נעלה את מערכת ההפעלה, והיא חייבת לעלות כ־guest. היא לא יכולה לעלות ב־context של ה־hypervisor). לכן, נאתחל את כל ה־Guest State Area להכיל בדיוק את אותם הערכים שכרגע יש על ה־host, כדי שבסוף האתחול הכל ימשיך לרוץ מאותה הנקודה. נתחיל כך:

```
__vmwrite(GUEST_CR0, __readcr0());
__vmwrite(GUEST_CR3, __readcr3());
__vmwrite(GUEST_CR4, __readcr4());
GDT gdt;
IDT idt;
GetGDT(&gdt);
GetIDT(&idt);
__vmwrite(GUEST_GDTR_BASE, gdt.address);
__vmwrite(GUEST_GDTR_LIMIT, gdt.limit);
__vmwrite(GUEST_IDTR_BASE, idt.address);
__vmwrite(GUEST_IDTR_LIMIT, idt.limit);
__vmwrite(GUEST_LDTR_BASE, 0);
__vmwrite(GUEST_LDTR_LIMIT, 0xff);
__vmwrite(GUEST_LDTR_AR_BYTES, VMCS_SELECTOR_UNUSABLE);
```

אנחנו שומרים על ה־GDT הנוכחי (אותו האחד שהכנו עוד בעת עליית המערכת), IDT הנוכחי, ערכי ה־Control registers הנוכחיים ואנחנו מסמנים את ה־(LDTR (Local Descriptor Table Refister כלא שמיש, משום שלא יצרנו אחד כזה, ואנחנו גם לא נצטרך.

בנוסף, אנחנו צריכים לדאוג לערכי ה־selector-ים השונים במערכת. אני אדגים כיצד מאתחלים את ה־selector CS, ושאר ה־selector-ים מאותחלים באופן דומה אך עם ערכים שונים:

```
// CS related data
    __vmwrite(GUEST_CS_SELECTOR, GetCS() & 0xf8);
    __vmwrite(GUEST_CS_BASE, 0);
    __vmwrite(GUEST_CS_LIMIT, 0xFFFFFfff);
    __vmwrite(GUEST_CS_AR_BYTES, 0xa09b);
```

שימו לב שב־VMCS אנחנו מציינים מה הוא הערך של כל selector באופן שונה מהרגיל. כאן כל selector מחולק פיזית לחלקים אליהם הוא מחולק לוגית ב־GDT (שדה ה־limit, שדה ה־base, ושדה ה־ access rights).

בערך ה־selector אנחנו שמים את מה שיש כרגע ב־CS, ודואגים לאפס את שלושת הביטים הימניים (היזכרו במבנה ה־selector CS על מנת להבין מדוע אנחנו עושים זאת). בשדה ה־base אנחנו שמים 0, וב־limit את הערך המקסימלי האפשרי, שהוא 0xFFFFFfff (זכרו שב־long mode לערך זה אין חשיבות כלל). שדה ה־access rights הוא החשוב ביותר, ואנחנו מאתחלים אותו בהתאם לערכים שיש ב־descriptor המקורי.

שני השדות החשובים ביותר שנותר לטפל בהם הם GUEST_RIP ו־GUEST_RSP. הערך ב־VMCS בשדה ה־GUEST_RIP הוא הכתובת של הפקודה שתתבצע כאשר נחזור לרוץ בתור guest, כאשר יתבצע vm-entry. באופן דומה, GUEST_RSP הוא הכתובת של ה־stack אליו יצביע האוגר rsp כאשר נחזור לרוץ בתור guest. נאתחל אותם כך:

```
__vmwrite(GUEST_RIP, VmmToVm);
__vmwrite(GUEST_RSP, 0);
```

הערך של VmmToVm (קיצור של Virtual machine monitor TO virtual machine) הוא מצביע לפונקציית אסמבלי שתוצג בסיום אתחול ה־VMCS. באותה הפונקציה אנחנו נדאג גם לבצע השמה ל־GUEST_RSP, שכרגע אנחנו מאחסנים בו 0.

בנקודת הזמן הנוכחית סיימנו לאתחל את ה־guest state area ואנחנו יכולים להמשיך לאתחול ה־host state area.

Host State Area

אזור זה של ה־VMCS מחזיק את הערכים אשר יטענו בעת התרחשות ה־vm-exit הבא. ערכים אלו רלוונטים ל־host, והם מחזיקים את ה־GDT של ה־hypervisor, את ה־stack pointer שלו, את כל ה־control registers ועוד.

נתחיל באתחול מאוד פשוט של CR0 ו־CR4:

```
__vmwrite(HOST_CR0, __readcr0());
__vmwrite(HOST_CR4, __readcr4() | CR4_OSXSAVE | (1 << 9));
```

באוגר CR4 אנחנו דואגים להפעיל את הביטים שהתנ"ך (Intel Manuals) דורש מאיתנו להדליק ב־host של המערכת. אין יותר מדי מה לפרט עליהם.

נעבור לאתחול מנגנון ה־Paging ולביצוע השמה ל־CR3. חלקכם חווים דז'ה וו רציני, ותוהים לעצמכם מדוע צריך לבצע אתחול נוסף למנגנון ה־Paging אם לפני כמה עמודים חפרתי לכם לא מעט על זה.

התשובה לתהייה הזאת היא פשוטה מאוד. ה־paging שעכשיו אנחנו נגדיר הולך לשמש את ה־hypervisor לאורך כל "משך חייו", ואילו ה־paging שהגדרנו קודם לכן משמש אותנו רק באתחול המערכת. היזכרו החלטנו באופן שרירותי לחלוטין להשתמש בכתובת 0x17000 ל־PML4, מבלי שהבטיחו לנו שהכתובת הזאת פנויה. מה שאנחנו עכשיו עושים זה להגדיר מנגנון Paging "חדש", שהוא חלק ממבני הנתונים של ה־hypervisor.

כפי שציינתי קודם, מבני הנתונים של ה־hypervisor נסתרים לחלוטין ממערכת ההפעלה, וזו היא הסיבה שאנחנו צריכים ליצור טבלאות מיפוי חדשות, שיעבדו בדיוק על אותו העיקרון שטבלאות המיפוי הנוכחיות עובדות. אתם יכולים להתנחם בזה שהאתחול לא יעשה באסמבלי, אלא הפעם ב־C.

האתחול נראה כך:

```
__vmwrite(HOST_CR3, InitializeHypervisorPaging(cpuData));
```

והפונקציה InitializeHypervisorPaging פשוטה מאוד, וכתובה באופן הבא:

```
QWORD InitializeHypervisorPaging(IN PSINGLE CPU DATA cpuData)
{
   cpuData->pageMapLevel4s[0] = VirtualToPhysical(cpuData->pageDirectoryPointerTables) |
PAGE_PRESENT | PAGE_RW;
   cpuData->pageMapLevel4s[200] = VirtualToPhysical(cpuData->pageDirectoryPointerTables) |
PAGE PRESENT | PAGE RW;
   for(QWORD i = 0; i < COMPUTER_MEM_SIZE; i++)
       cpuData->pageDirectoryPointerTables[i] =
           VirtualToPhysical(&(cpuData->pageDirectories[i * ARRAY_PAGE_SIZE])) |
PAGE_PRESENT | PAGE_RW;

   QWORD physicalAddress = 0;
   for(QWORD i = 0; i < COMPUTER_MEM_SIZE * ARRAY_PAGE_SIZE;
       i++, physicalAddress += LARGE_PAGE_SIZE)
       cpuData->pageDirectories[i] = physicalAddress | PAGE_LARGE_PAGE | PAGE_RW |
PAGE_PRESENT;
   return VirtualToPhysical(cpuData->pageMapLevel4s);
}
```

מה שאנחנו עושים בהתחלה (שורה 1) זה לקשר בין ה־PML4 ל־PDPT (ה־PML4 באופן מעשי יסתפק בתא אחד בלבד, הסברתי על זה קודם). לאחר מכן (שורה 2) אנחנו מבצעים את אותו הקישור בדיוק עבור התא ה־200 ב־PML4, כדי ליצור מיפוי בהיסט X, כפי שתיארתי בהסבר על paging.

הלולאה הראשונה אחראית על קישור של ה־PDPTs ל־PDs. הערך של ARRAY_PAGE_SIZE הוא 512, שזה מספר הרשומות בכל table כאשר PAE פועל. אם במחשב יש 16GB של זיכרון RAM, אזי הלולאה תקשר את 16 ה־entries הראשונים ב־PDPT ל־16 טבלאות PDs, שכל אחת ממפה 1GB של זיכרון.

הלולאה השנייה אחראית על קישור כל איבר ב־PD לכתובת הפיזית אליה הוא מצביע (זכרו - אנחנו משתמשים ב־page-ים בגודל של 2MB, ולכן ה־PDs לא מצביעים על page tables, אלא ישירות על הזיכרון הפיזי.)

ערך החזרה של הפונקציה הוא כמובן הכתובת הפיזית של PML4, מה שנכנס ישירות ל־CR3, כפי שראינו למעלה.

הדבר הבא שנאתחל זה את HOST_RIP ו־HOST_RSP. ערכים אלו שומרים את הכתובות של rip ו־rsp בהתאמה כאשר יתרחש vm-exit. יש שיגידו שאלו ההערכים החשובים ביותר ב־VMCS.

```
    __vmwrite(HOST_RIP, HandleVmExit);
    __vmwrite(HOST_RSP, cpuData->stack + sizeof(cpuData->stack));
```

מכיוון שה־stack כפי שאנחנו מכירים אותו גדל "מלמעלה למטה", אנחנו דואגים שהערך של HOST_RSP יהיה הכתובת בה ה־stack "נגמר". כמו כן, אנחנו משתמשים כאן בכתובת הוירטואלית ולא הפיזית, מכיוון שגם rip וגם rsp מתייחסים לכתובות וירטואליות.

הפונקציה HandleVmExit, כמו שאתם יכולים לנחש, היא פונקציית אסמבלי שמטפלת ב־vm-exits. אני אציג אותה ממש בעוד רגע, כשנסיים לקנפג את ה־host state area.

נעבור ל־fs ו־gs. ב־VMCS יש לנו שני שדות בשם HOST_FS_BASE ו־HOST_GS_BASE. ערכים אלו נטענים לאוגרים fs ו־gs בהתאמה כאשר מתרחש vm-exit. אוגרים אלו הם אוגרים שאין להם שימוש חומרתי - כלומר התוכנה (מערכת ההפעלה וה־hypervisor) עושים בהם שימוש והם אחראים על ניהול הערכים האלו.

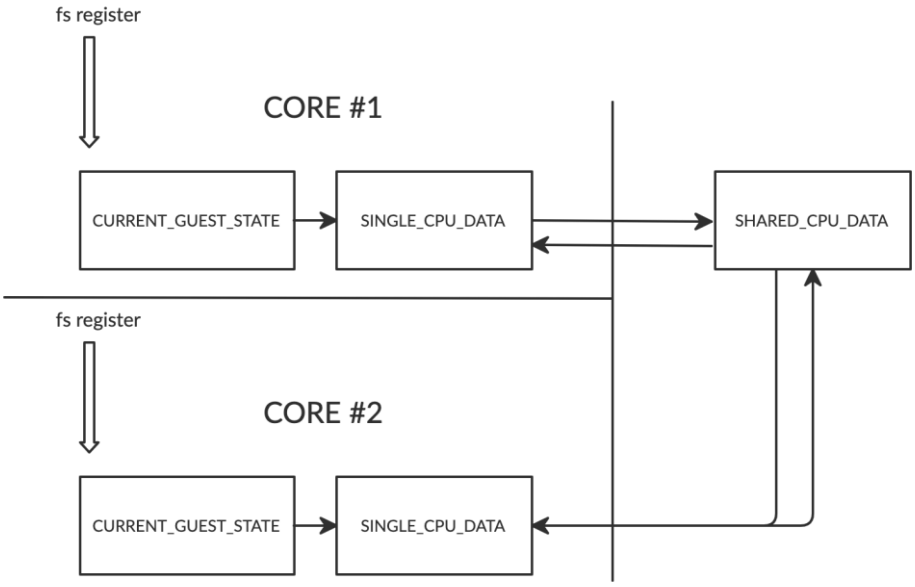
מה השימוש שה־hypervisor עושה בהם? בואו נראה:

```
__vmwrite(HOST_FS_BASE, cpuData->sharedData->currentState[cpuData-
>coreIdentifier]);
__vmwrite(HOST_GS_BASE, 0);
```

אתם יכולים לנחש שב־gs אין כל שימוש. מי שניחש כך בהחלט צדק. מה עם fs? למה הוא משמש? ובכן, הוא משמש לאחסון מבנה הנתונים הרלוונטי לכל ליבה כאשר מתרחש vm-exit. המשתנה currentState שומר מערך של מצביעים מסוג CURRENT_GUEST_STATE.

מבנה זה, כפי שכבר תואר, שומר את הרגיסטרים של ה־guest בכל פעם שמתרחש vm-exit, ובנוסף יש לו מצביע ל־SINGLE_CPU_DATA.

את הקשר בין מבני הנתונים השונים ניתן לראות בדיאגרמה הבאה:



מה עם שמירת הרגיסטרים למשתנה guestRegisters? כלומר, מי דואג לכך שבכל פעם שיתבצע vm-exit, הרגיסטרים ישמרו שם? האם מדובר במנגנון מובנה של הוירטואליזציה?

לצערנו התשובה היא שלילית. אנחנו נעשה זאת ידנית בכל פעם שיתרחש vm-exit, ממש כך:

```
HandleVmExit:
    mov qword [fs:0], rax
    mov qword [fs:8], rbx
    mov qword [fs:16], rcx
    mov qword [fs:24], rdx
    mov qword [fs:32], rsi
    mov qword [fs:40], rdi
    mov qword [fs:48], rbp
    mov qword [fs:56], r8
    mov qword [fs:64], r9
    mov qword [fs:72], r10
    mov qword [fs:80], r11
    mov qword [fs:88], r12
    mov qword [fs:96], r13
    mov qword [fs:104], r14
    mov qword [fs:112], r15
    mov rax, 0x0000681c    ; Geust RSP
    vmread rbx, rax
    mov qword [fs:120], rbx
    mov rax, 0x0000681e    ; Geust RIP
    vmread rbx, rax
    mov qword [fs:128], rbx

    ; Run handler function
    call HandleVmExitEx
```

השגרה הזו נקראת בכל פעם שמתרחש vm-exit. הכתובת של השגרה היא למעשה הערך שיש לנו ב־HOST_RIP.

הדבר הראשון שאנחנו עושים הוא לקחת את האוגרים as is ולאחסן אותם ב־CURRENT_GUEST_STATE שאליו מצביע fs. מכיוון שהאוגרים מאוחסנים בתחילת המבנה (למעלה ניתן לראות כי האיבר הראשון במבנה זה הוא מבנה מסוג REGISTERS), אנחנו מתחילים להעביר את האוגרים לפי הסדר ל־fs עם קפיצה של 8 בכל פעם (גודל כל רגיסטר). היחידים שחריגים בעין הם rip ו־rsp.

אם תחשבו על זה, אי אפשר לעשות פשוט mov עם rip, מכיוון ש־rip מצביע על הקוד שרץ עכשיו, ולא מה שרץ ברגע שהתרחש vm-exit (התעלמו לרגע מהעובדה שטכנית אי אפשר לבצע mov עם rip). באופן דומה rsp מצביע על ה־stack של ה־hypervisor ולא של ה־guest.

אם אתם זוכרים - אנחנו דיברנו לפני כמה דקות על ה־guest state area. לשם נטענים ערכים שרלוונטים ל־guest כאשר מתרחש vm-exit, והם נטענים חזרה ל־guest ב־vm-entry. לכן, אנחנו ניקח את הערכים של rip ו־rsp מה־guest state area ב־VMCS.

Exit Reasons וטיפול ב־**Vm-Exits**

למעלה אתם יכולים לראות את השורה call HandleVmExitEx. הפונקציה HandleVmExitEx היא פונקציית C שבעצם אחראית לטפל ב־vm-exit. אני אחשוף את השלד שלה עכשיו, ובהמשך הדרך נרחיב אותה:

```
VOID HandleVmExitEx()
{
    QWORD exitReason = vmread(VM_EXIT_REASON);

    switch(exitReason & 0xffff) // 0..15, Intel SDM 26.7
    {
        case EXIT_REASON_EPT_VIOLATION:
            // ...
            break;
        case EXIT_REASON_INVALID_GUEST_STATE:
            // ...
            break;
        case EXIT_REASON_EXCEPTION_NMI:
            // ...
            break;
        case EXIT_REASON_INIT:
            // ...
            break;
        case EXIT_REASON_MSR_LOADING:
            // ...
            break;
        case EXIT_REASON_MCE_DURING_VMENTRY:
            // ...
            break;
        default:
            Print("No match found, exit reason %d\n", exitReason);
            ASSERT(FALSE);
    }
}
```

כמו שאתם יכולים להבין - ה־VMCS מספק לנו שדה(ות) שמספרים לנו על הסיבה שגרמה ל־vm-exit. לכל סיבה כזאת יש exit reason המשוייך לה (אלו הם ערכים מוגדרים מראש). אנחנו קוראים מה־VMCS את הערך של השדה ובוחנים אותו.

בהתאם לערך שלו אנחנו נטפל ב־vm-exit, כאשר אופן הטיפול בכל סיבה יכול להיות שונה. אם ה־hypervisor נתקל ב־exit reason שהוא לא ציפה לו (default), הוא מדפיס הודעת שגיאה ונכנס ללופ אינסופי שתוקע את המערכת (ASSERT).

יש לנו מספר גדול מאוד של סיבות היכולות לגרום ל־vm-exit, ואני לא הולך לעסוק בכל הסיבות האפשריות במסגרת המאמר. בקרוב מאוד אדבר על כמה סיבות נבחרות, ואתם יכולים להתרשם ב־github כיצד ניתן לטפל ב־exit reasons נוספים שאני לא אדבר עליהם כאן.

Control fields & VM-Execution controls

ה־VMCS שלנו מורכב ממקטעים שעל שניים מהם כבר עברנו. דיברנו גם על שני שדות חשובים מאוד, HOST_RIP ו־HOST_RSP, ששניהם חלק מה־host state area. אם טיפלנו כבר במקטע של ה־host וגם של ה־guest, אז על מה כבר נשאר לנו כבר לדבר? שאלה מצוינת! שימו לב שבאף מקום לא דאגנו לציין כיצד ה־hypervisor למעשה יעבוד. למה הכוונה? לא הגדרנו באף מקום על אילו פעולות אנחנו רוצים לגרום ל"עצירה רגעית" של המערכת ולהעביר את השליטה ל־hypervisor (כלומר, לגרום ל־vm-exit). המקום לעשות זאת הוא בדיוק ב־section הזה של ה־VMCS, והוא נראה כך:

CONTROL FIELDS				
Pin-Based VM-Execution Controls	External-interrupt exiting	NMI exiting	Virtual NMIs	
	Activate VMX-preemption timer	Process posted interrupts		
Primary processor-based VM-execution controls	Interrupt-window exiting	Use TSC offsetting		
	HLT exiting	INVLPG exiting	MWAIT exiting	RDPMC exiting
	RDTSC exiting	CR3-load exiting	CR3-store exiting	CR8-load exiting
	CR8-store exiting	Use TPR shadow	NMI-window exiting	MOV-DR exiting
	Unconditional I/O exiting	Use I/O bitmaps	Monitor trap flag	Use MSR bitmaps
	MONITOR exiting	PAUSE exiting	Activate secondary controls	
Secondary processor-based VM-execution controls	Virtualize APIC accesses	Enable EPT	Descriptor-table exiting	Enable RDTSCP
	Virtualize x2APIC mode	Enable VPID	WBINVD exiting	Unrestricted guest
	APIC-register virtualization	Virtual-interrupt delivery	PAUSE-loop exiting	
	RDRAND exiting	Enable INVPCID	Enable VM functions	VMCS shadowing
	Enable ENCLS exiting	RDSEED exiting	Enable PML	EPT-violation #VE
	Conceal VMX non-root operation from Intel PT	Enable XSAVES/XRSTORS		
	Mode-based execute control for EPT	Use TSC scaling		
Exception Bitmap	I/O-Bitmap Addresses	TSC-offset		
Guest/Host Masks for CR0	Guest/Host Masks for CR4	Read Shadows for CR0	Read Shadows for CR4	
CR3-target value 0	CR3-target value 1	CR3-target value 2	CR3-target value 3	CR3-target count
APIC Virtualization	APIC-access address	Virtual-APIC address	TPR threshold	
	EOI-exit bitmap 0	EOI-exit bitmap 1	EOI-exit bitmap 2	EOI-exit bitmap 3
	Posted-interrupt notification vector	Posted-interrupt descriptor address		
Read bitmap for low MSRs	Read bitmap for high MSRs	Write bitmap for low MSRs	Write bitmap for low MSRs	
Executive-VMCS Pointer	Extended-Page-Table Pointer	Virtual-Processor Identifier		
PLE_Gap	PLE_Window	VM-function controls	VMREAD bitmap	VMWRITE bitmap
ENCLS-exiting bitmap	PML address			
Virtualization-exception information address	EPTP index	XSS-exiting bitmap		
VM-EXIT CONTROL FIELDS				
VM-Exit Controls	Save debug controls	Host address space size	Load IA32_PERF_GLOBAL_CTRL	
	Acknowledge interrupt on exit	Save IA32_PAT	Load IA32_PAT	Save IA32_EFER	Load IA32_EFER
	Save VMX preemption timer value	Clear IA32_BNDCFGS	Conceal VM exits from Intel PT	
VM-Exit Controls for MSRs	VM-exit MSR-store count	VM-exit MSR-store address		
	VM-exit MSR-load count	VM-exit MSR-load address		
VM-EXIT INFORMATION FIELDS				
Basic VM-Exit Information	Exit reason	Exit qualification		
	Guest-linear address	Guest-physical address		
VM Exits Due to Vectored Events	VM-exit interruption information	VM-exit interruption error code		
VM Exits That Occur During Event Delivery	IDT-vectoring information	IDT-vectoring error code		
VM Exits Due to Instruction Execution	VM-exit instruction length	VM-exit instruction information		
	I/O RCX	I/O RSI	I/O RDI	I/O RIP
VM-instruction error field				

- Natural-Width fields.
- 16-bits fields.
- 32-bits fields.
- 64-bits fields.

CopyLeft 2017, @Noteworthy (Intel Manuel of July 2017)

כמו שאתם רואים מדובר במקטע עצום ולכן אני לא הולך לכסות את כולו. אנחנו נעסוק רק בקונפיגורציות נבחרות, ואני בטוח שברגע שתסיימו לקרוא את המאמר אתם תלכו לקרוא עוד ב־manuals של intel על מה שנראה לכם מספיק מעניין.

הדבר הראשון שאסביר עליו הוא על השילוש הקדוש שכולל את Pin-Based VM-Execution Controls (השדה הראשון), Primary processor based VM-execution controls (השדה השני) ו־ Secondary processor based VM-execution controls (השדה השלישי).

Primary processor based VM-execution controls

במקטע זה של ה־VMCS נשמר מידע אודות טיפול בפעולות אסינכרוניות. למעשה, כמעט כל השדות פה מתייחסים לאופן הטיפול בפסיקות. תוכלו לראות בטבלה כאן את מבנה שדה זה. ה־hypervisor הבסיסי שלנו לא עושה בשדה זה כל שימוש, ועל כן לא ארחיב עליו כאן.

Primary processor based VM-execution controls

במקטע זה של ה־VMCS נשמר מידע אודות הפעלת פונקציונליות של ה־hypervisor בהתאם לפעולות המתבצעות ב־guest. על פי ה־manuals של intel, כאשר הביט המקושר אל פעולה מסוימת דולק, אזי הפעולה תגרום ל־vm-exit כאשר היא תתבצע ב־guest. הטבלה המתארת את ה־layout של שדה זה נמצאת כאן ו־כאן. מעיון מטבלה, תוכלו להתרשם שאם ביט 7 דולק אזי יתרחש vm-exit כאשר הפקודה HLT תתבצע על ידי ה־guest. כמו כן, ביטים 15 ו־16 קובעים האם יתבצע vm-exit כאשר ה־guest מבצע טעינה והשמה בהתאמה לאוגר CR3. בחלק 2 של המאמר אציג את אחד השדות שה־hypervisor שלנו עושה בהם שימוש.

Secondary processor based VM-execution controls

מקטע זה של ה־VMCS הוא basically המשך של המקטע הקודם, וגם הוא מתאר מה ה־guest רשאי לעשות עם התערבות ה־hypervisor, ומה הוא רשאי לעשות ללא ההתערבות של ה־hypervisor. תוכלו לראות את המבנה שלו כאן. כבר בהמשך החלק הנוכחי של המאמר אני אציג שימוש באחד השדות אשר נמצאים פה.

עכשיו אחרי שדיברנו על השילוש הקדוש, הגיע הזמן לדבר על שדות נוספים שמגדירים ל־hypervisor על מה בדיוק יתבצע vm-exit, ומה מותר או אסור ל־guest לבצע.

Exception Bitmap

מדובר בתא ב־VMCS באורך 4 בתים הקובע עבור כל exception מהרשימה הבאה, האם הוא גורם ל־vm-exit בעת התרחשותו:

Divide by zero error, Debug, Non-maskable interrupt, Breakpoint, Overflow, Bound Range Exceeded, Invalid Opcode, Device Not Available, Double Fault, Coprocessor Segment Overrun, Invalid TSS, Segment Not Present, Stack-Segment Fault, General Protection Fault, Page Fault, x87 Floating-Point Exception, Alignment Check, Machine Check, SIMD Floating-Point Exception, Virtualization Exception, Security Exception, Triple Fault

אם הביט המקושר ל־exception מסוים דולק בעת התרחשותו, הדבר יגרום ל־vm-exit. אם הוא איננו דולק, הטיפול ב־exception יעשה על ידי ה־ guest על פי ה־IDT שלו, ללא התערבות של ה־host.

דוגמא לשימוש:

נניח וברצוננו לגרום ל־vm-exit בכל פעם שמתרחש page-fault ב־guest, ולבצע בדיקה כלשהי על הכתובות שגרמה ל־page-fault. האתחול יראה כך:

```
__vmwrite(EXCEPTION_BITMAP, vmread(EXCEPTION_BITMAP) | (1 <<
PAGE_FAULT_VECTOR_NUMBER));
```

אנחנו ראשית קוראים את הערך הנוכחי שיש ב־EXCEPTION_BITMAP ולאחר מכן מדליקים את הביט המקושר ל־page faults.

מעתה והלאה, כל page-fault ב־guest יגרום ל־vm-exit. מה הוא ה־exit reason המשוייך ל־page fault אתם תוהים? ובכן, אין exit reason לכל exception בפני עצמו. יש exit reaon "כללי" עבור exception-ים שהוא EXIT_REASON_EXCEPTION_NMI.

על מנת לדעת מה הוא בדיוק ה־exception שאירע, נעזר בשדה נוסף ב־VMCS שמספר לנו מה הוא סוג ה־exception, שהוא VM_EXIT_INTR_INFO. שמונת הביטים הימניים יגידו לנו מה הוא בדיוק ה־ vector number המשוייך ל־exception שקרה

למשל, זיהוי page faults על ידי ה־host נעשה באופן הבא (המקטע שלהלן הוא חלק מהפונקציה HandleVmExitEx, לכן רשמתי רק את ה־case הרלוונטי):

```
case EXIT_REASON_EXCEPTION_NMI:
   QWORD vectorNumber = vmread(VM_EXIT_INTR_INFO) & 0xff;
   if (vectorNumber != PAGE_FAULT_VECTOR_NUMBER)
       break;
   QWORD virtualAddress = vmread(EXIT_QUALIFICATION);
   //
   // perform some checks on the faulting address
   //
   if(shouldInjectPageFaultBackToGuest)
   {
       __writecr2(virtualAddress);
       InjectExceptionToGuest(PAGE_FAULT_VECTOR_NUMBER);
   }
   break;
```

ראשית אנחנו בודקים מה הוא ה־exception על פי שדה ה־VM_EXIT_INTR_INFO שנמצא ב־VMCS. אם ה־vector number שונה מה־vector המשוייך ל־page fault, אזי לא מדובר ב־page faults ואנחנו מפסיקים את הבדיקה.

אם אכן מדובר ב־page fault, אנחנו קוראים את שדה ה־EXIT QUALIFICATION שב־VMCS. שדה זה מכיל את הכתובת הוירטואלית שגרמה ל־page fault. לאחר מכן נבצע בדיקה כלשהי, ובהתאם לבדיקה שנבצע נחליט האם אנחנו אכן מעוניינים שה־page fault יתרחש או לא.

אם אנחנו רוצים לבטל אותו, פשוט "לא נעשה כלום". מדוע? משום שאם התרחש vm-exit כתוצאה מ־exception על פי ה־exception bitmap, אזי ה־exception לא יטופל על ידי ה־guest באופן דיפולטיבי. על מנת לבצע delivering של ה־exception ל־guest נצטרך להזריק אותו באופן ידני.

במקרה של page fault, נכתוב לאוגר CR2 את הכתובת שגרמה ל־page fault (ניתן לקרוא על אוגר זה ומטרתו כאן), ונזריק page fault ל־guest (אני לא אתעכב על איך בדיוק מתבצעת ההזרקה, מוזמנים להסתכל ב־github על מנת לראות כיצד זה ממומש, או לקרוא ב־Part 3C של התנ"ך).

EPT Enabled & EPT Pointer

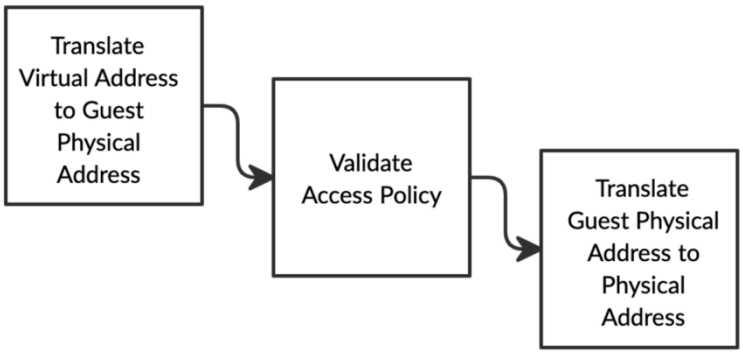
לפני שאפצח בהסבר, לפי דעתי מדובר באחד הדברים היפים ביותר והחזקים ביותר שמנגנון הוירטואליזציה מציע.

תעצמו לרגע עיניים, קחו נשימה עמוקה, ודמיינו לעצמכם את הסיטואציה הבאה: process כלשהו ב־guest ניגש לכתובת וירטואלית X. הכתובת הוירטואלית X עברה תרגום לכתובת פיזית Y באמצעות מנגנון ה־paging של מערכת ההפעלה. הכל טוב ויפה.

עכשיו דמיינו לכם איזה כוח היה לכם בידיים אם הייתם יכולים לקבוע שהכתובת הפיזית Y היא לא באמת Y. כלומר, הייתם קובעים שהכתובת הפיזית Y בכלל מפנה לכתובת הפיזית Z. יותר מזה - דמיינו לעצמכם שהייתם יכולים לקבל דיווח על כך שה־guest ניסה לגשת לכתובת הפיזית Y ולדעת בדיוק מה הוא ניסה לעשות - האם ניסה לקרוא? האם ניסה לכתוב? האם ניסה לבצע הרצה?

וואו. עכשיו אמור להיות הרגע בו אתם פוקחים את העיניים ואומרים לעצמכם "איזה חלום... אם רק זה היה אמיתי...". טוב - זה הוא יום המזל שלכם, כי זה אמיתי לגמרי. מנגנון הוירטואליזציה מביא איתו כלי חדש שנקרא Extended Page Table, או בקיצור EPT. בעזרת מנגנון זה כל physical address הופך להיות guest physical address.

כלומר, הכתובת הפיזית שה־guest מנסה לגשת אליה היא לא באמת כתובת פיזית אמיתית, אלא זו היא כתובת וירטואלית שהולכת לעבור תרגום נוסף ברמת ה־**hypervisor** לכתובת פיזית אמיתית. אם אנחנו מדברים על guest שנמצא ב־protected mode עם paging, כל כתובת וירטואלית עוברת את התהליך הבא:

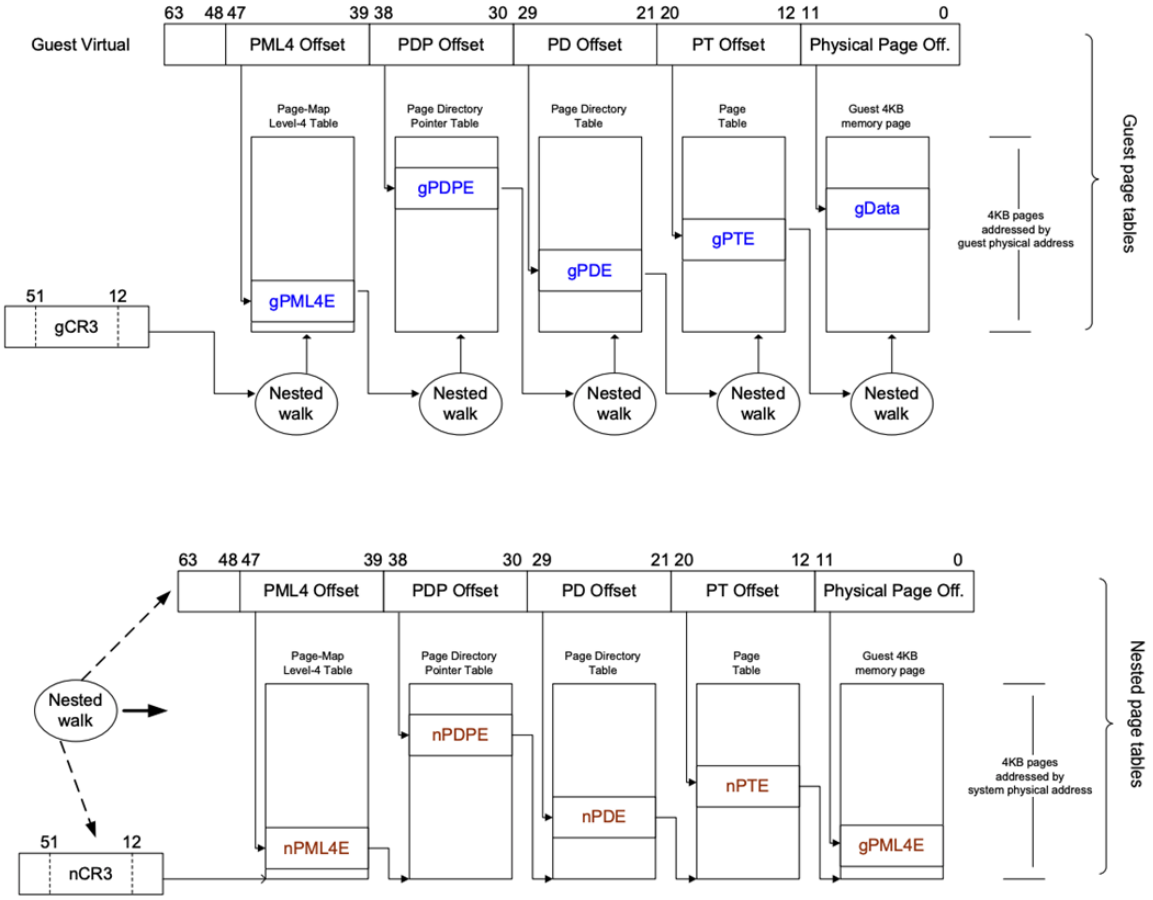


כמו שאתם רואים, לפני שהכתובת הפיזית ה"מזויפת" מתורגמת לכתובת הפיזית האמיתית, המנגנון של EPT מוודא שהפעולה שה־guest ניסה לבצע היא valid-ית.

אם הוא ניסה לבצע הרצה של קוד (למשל) על page שה־hypervisor סימן ככזה שלא ניתן להריץ בו קוד - אזי יתבצע vm-exit עם exit reason של EPT_VIOLATION. כמובן שבהמשך אראה דוגמא לשימוש של המנגנון.

רגע רגע רגע... זה אומר שעכשיו תהליך התרגום של כתובת וירטואלית לפיזית הוא איטי יותר, כי יש שכבה נוספת של תרגום? כן, טוב, לא באמת חשבתם שתקבלו את הכוח הזה מבלי לשלם אפילו קצת, נכון?

למעשה, כך נראה תרגום של כתובת וירטואלית לפיזית כאשר מנגנון ה־EPT פועל:



[קרדיט: AMD-V Nested Paging Guide]

מה שאתם רואים למעלה זה את טבלאות המיפוי שאנחנו רגילים לראות, כאשר בכל מעבר מטבלה לטבלה אנחנו ניגשים ל־Extended Page Table (מה שמכונה Nested Walk) על מנת לתרגם את הכתובת הפיזית ה"מזויפת" (בה ה־guest משתמש) של כל Page Table לכתובת הפיזית האמיתית שהוא נמצא בה (כלומר לתרגם כל guest physical address ל־physical address).

ה־hypervisor הוא זה שאחראי על ניהול מנגנון ה־EPT ול־guest אין שום גישה אליו. כלומר מבחינתו הוא באמת ניגש לכתובת הפיזית אליה הוא ניסה לגשת.

עכשיו, שאנחנו מבינים את הרעיון, בואו ניגש למימוש, ולאחר מכן נראה דוגמא לשימוש. ראשית, השימוש ב־EPT איננו חובה וה־hypervisor באופן דיפולטיבי איננו מפעיל את המנגנון הזה.

אם תסתכלו למעלה בתרשים של ה־VMCS, תחת ה־section של Secondary processor-based VM-Execution controls, אתם יכולים לראות שיש ביט אחד המשויך ל־Enable EPT.

הדבר הראשון שנצטרך לעשות כמובן הוא להדליק את הביט הזה, וזה נעשה כך:

```
__vmwrite(SECONDARY_VM_EXEC_CONTROLS, vmread(SECONDARY_VM_EXEC_CONTROLS)
| CPU_BASED_CTL2_ENABLE_EPT);
```

אנחנו בעצם קוראים את הערך הנוכחי שיש בשדה זה, ומדליקים גם את הביט המקושר ל־EPT. מעתה והלאה, כל גישה של ה־guest לכתובת כלשהי על גבי הליבה הנוכחית תיעשה בהתאם למנגנון שהצגתי למעלה.

לאחר שהדלקנו את הביט הזה, נצטרך להזין בשדה EPT_POINTER את הכתובת הפיזית של ה־ EPT PML4 (כלומר 4 Extended Page Table - Page Map Level).

שדה זה מזכיר לנו בדיוק את התפקיד של CR3 במנגנון ה־paging הרגיל. זה נראה כך:

```
__vmwrite(EPT_POINTER, InitializeExtendedPageTable(cpuData));
```

וכמובן, הפונקציה InitializeExtendedPageTable מאתחלת את ה־EPT. לפני שנציג את הפונקציה, נדבר על מבנה טבלאות ה־EPT שדומה מאוד למבנה הטבלאות הרגיל.

ראשית, נראה את המבנה של EPT_POINTER המוחזר מפונקציית האתחול:

Table 24-8. Format of Extended-Page-Table Pointer

Bit Position(s)	Field
2:0	EPT paging-structure memory type (see Section 28.2.6): 0 = Uncacheable (UC) 6 = Write-back (WB) Other values are reserved.[1]
5:3	This value is 1 less than the EPT page-walk length (see Section 28.2.2)
6	Setting this control to 1 enables accessed and dirty flags for EPT (see Section 28.2.4)[2]
11:7	Reserved
N-1:12	Bits N-1:12 of the physical address of the 4-KByte aligned EPT PML4 table[3]
63:N	Reserved

[קרדיט: Intel Developer Manual 3C]

שלושת הביטים הראשונים מציינים את ה־type של ה־memory, כלומר האם ניתן לעשות לו cache או לא. אין סיבה כרגע שלא נאפשר caching, אז נבחר את הערך 6 (ראו מימוש בהמשך). כמו כן, שלושת הביטים הבאים, באינדקסים 3 עד 5 מציינים את האורך של ה־page walk פחות 1.

מכיוון שיש לנו 4 טבלאות מיפוי (PML4, PDPT, PD, PT), ה־page walk length הוא 4, ומכאן שהערך בשדה זה הוא 3.

ביטים 12 עד הסוף יהוו את הכתובת הפיזית של תחילת ה־EPT PML4.

שנית, נציג את מבנה ה־PML4/PDPT/PD/PT שדומים מאוד ב־EPT (מבחינתנו יהיה להם את אותו המבנה, ונתעלם מההבדל הדק ביניהם):

Table 28-1. Format of an EPT PML4 Entry (PML4E) that References an EPT Page-Directory-Pointer Table

Bit Position(s)	Contents
0	Read access; indicates whether reads are allowed from the 512-GByte region controlled by this entry
1	Write access; indicates whether writes are allowed from the 512-GByte region controlled by this entry
2	If the "mode-based execute control for EPT" VM-execution control is 0, execute access; indicates whether instruction fetches are allowed from the 512-GByte region controlled by this entry If that control is 1, execute access for supervisor-mode linear addresses; indicates whether instruction fetches are allowed from supervisor-mode linear addresses in the 512-GByte region controlled by this entry
7:3	Reserved (must be 0)
8	If bit 6 of EPTP is 1, accessed flag for EPT; indicates whether software has accessed the 512-GByte region controlled by this entry (see Section 28.2.4). Ignored if bit 6 of EPTP is 0
9	Ignored
10	Execute access for user-mode linear addresses. If the "mode-based execute control for EPT" VM-execution control is 1, indicates whether instruction fetches are allowed from user-mode linear addresses in the 512-GByte region controlled by this entry. If that control is 0, this bit is ignored.
11	Ignored
(N-1):12	Physical address of 4-KByte aligned EPT page-directory-pointer table referenced by this entry[1]
51:N	Reserved (must be 0)
63:52	Ignored

[קרדיט: Intel Developer Manual 3C]

הביטים המעניינים ביותר מבחינה אפליקטיבית הם ביטים 0 עד 2 ו־10. הביטים 0 עד 2 מציינים האם מותר לבצע קריאה, כתיבה והרצה (בהתאמה) מה־region עליו מצביע ה־page הנוכחי, כאשר גודל ה־region נקבע בהתאם לטבלה בה אנחנו נמצאים.

ביט 10 מציין האם להתייחס לביט 2 בתור הרשאת הרצה כפי שתיארתי, או שמא לתת לו "טוויסט" קטן. במקרה שביט 10 דולק, ביט 2 יקבע האם קוד שרץ ב־user mode רשאי להריץ פקודות מה־region הנוכחי. במידה וביט 10 כבוי, ביט 2 מסמל האם קיימת הרשאת הרצה מה־region הנוכחי, ללא תלות ב־CPL.

עכשיו, שאנחנו כבר מכירים את מבנה טבלאות ה־EPT, נוכל לממש את טבלאות המיפוי:

```
QWORD InitializeExtendedPageTable(IN PSINGLE_CPU_DATA cpuData)
{
   QWORD eptp = VirtualToPhysical(cpuData->eptPageMapLevel4s) | EPT_PAGE_WALK_LENGTH |
EPT_POINTER_WB;
   cpuData->eptPageMapLevel4s[0] = VirtualToPhysical(cpuData-
>eptPageDirectoryPointerTables) | EPT_RWX;
   for(QWORD i = 0; i < COMPUTER_MEM_SIZE; i++)
       cpuData->eptPageDirectoryPointerTables[i] = VirtualToPhysical(&(cpuData-
>eptPageDirectories[i * ARRAY_PAGE_SIZE]))
           | EPT_RWX;
   for(QWORD i = 0; i < COMPUTER_MEM_SIZE * ARRAY_PAGE_SIZE; i++)
       cpuData->eptPageDirectories[i] = VirtualToPhysical(&(cpuData->eptPageTables[i *
ARRAY_PAGE_SIZE])) | EPT_RWX;
   QWORD physicalAddress = 0;
   for(QWORD i = 0; i < COMPUTER_MEM_SIZE * ARRAY_PAGE_SIZE * ARRAY_PAGE_SIZE; i++,
physicalAddress += PAGE_SIZE)
       cpuData->eptPageTables[i] = physicalAddress | EPT_RWX | (EPT_POINTER_WB << 3);

   return eptp;
}
```

דבר ראשון - בואו נרגע, אין ממה לחשוש! אני הולך להסביר על כל שורה ושורה בנפרד.

נתחיל:

מכיוון שכל יחידת עיבוד מחזיקה VMCS ייחודי לה, וה־EPT משוייך ל־VMCS, בחרתי לממש EPT נפרד לכל יחידת עיבוד. לכן הכתובת של ה־EPT תמצא במבנה הנתונים אשר רלוונטי לכל ליבה (SINGLE_CPU_DATA). זה הוא גם הפרמטר שהפונקציה מקבלת.

ראשית אנחנו ממירים את הכתובת הוירטואלית של ה־PML4 לכתובת פיזית (שורה 1). כמו כן, אנחנו מעדכנים 2 דברים (באמצעות bitwise or): את ה־page walk length ואת ה־caching policy, כפי שכבר תואר.

הסברנו כבר מדוע ה־page walk length הוא 3, לכן כל שנותר הוא להזיז את הערך 3 למקום המתאים (ביטים 3 עד 5). נעשה זאת באמצעות shift שמאלה שמביא את המספר למקום הנדרש. ה־ caching policy שנבחר בו הוא write back. הערך של EPT_POINTER_WB הוא 6, עם shift שמאלה שמביא את המספר למקום הנדרש.

באופן דומה לטבלאות ה־PML4 במסגרת מנגנון ה־paging ה"רגיל", אנחנו נשתמש אך ורק בתא אחד מכיוון שהוא ממפה 512GB של זיכרון. לכן אנחנו ניגשים ישירות ל־PML4 בתא ה־0 ומכוונים אותו לטבלה אחת שמכילה 512 רשומות של PDPTs (שורה 2). הדבר הנוסף שנעשה הוא לאפשר קריאה, כתיבה והרצה באמצעות ביצוע bitwise or עם הערך 7 (111 בבינארית).

הלאה, הבנו כבר שה־EPT מתייחס לזיכרון פיזי, לכן מספיק לאתחל רשומות ב־PDPT בהתאם לכמות ה־RAM שיש במחשב (שנקבע על ידי COMPUTER_MEM_SIZE). נקשר כל איבר ב־PDPT ל־512 רשומות ב־PDs. באופן דומה מקשרים גם כל איבר ב־PD ל־512 רשומות ב־PTs.

מה שקורה החל מכאן כבר מאוד דומה לאתחול מנגנון ה־paging הרגיל - אנחנו מתחילים מכתובת פיזית 0, ובקפיצות של 0x1000 (גודל של page בודד) אנחנו ממפים את הזיכרון באופן לינארי (כך שהכתובת הפיזית X של ה־guest באמת תפנה לכתובת הפיזית X). הערך המוחזר מהפונקציה הוא כמובן המצביע שיצרנו בהתחלה.

עכשיו אחרי שאתחלנו את ה־EPT, הגיע הזמן לדבר על דוגמא לשימוש:

הדוגמא לשימוש שאני אתן לא תכלול את כל הקוד שרלוונטי אליה כאן מכיוון שהוא מאוד ארוך. עם זאת, הוא ממומש במלואו ב־github (קוד רלוונטי: #1, #2, #3) לכל מי שמעוניין לקרוא אותו ולרדת לפרטים הקטנים.

למי מכם שבקיא ב־internals של מערכות Windows בטח יודע כמה קל זה לעשות hook ל־system calls במערכות 32 ביט. כל מה שצריך לעשות זה פשוט לשכתב את ה־SSDT ולשנות את הכתובות בטבלאות

המערכת. עם זאת, רמת הקושי עולה כשמדברים על 64 ביט עם כניסתו של מנגנון ה־KPP (בשמו המלא Kernel Patch Protection, או Patchguard). לא ניתן לבצע מניפולציות על מבני נתונים מסוימים של ה־kernel, וביניהם ה־SSDT.

מעבר לכך, לא ניתן גם לבצע שינוי קוד (כלומר לדרוס את הפקודות הקיימות באחרות, כמו jmp לכתובת בזיכרון). כאשר Patchguard מזהה שינוי כלשהו במבני הנתונים של המערכת או בקוד ה־kernel-י, הוא פשוט מקריס את המערכת.

למרות הקושי הגדול ש־Patchguard מכניס, הוא ממומש בצורה פשוטה להפליא. אחת לכמה זמן הוא סורק את מרחב הזיכרון של ה־kernel (חלקים בכל פעם) והוא מחפש אחר שינויים. כלומר, הוא פשוט מבצע קריאה של קוד ומבני נתונים ומחפש אחר שינויים.

(הערת אגב: מנגנון ה־Patchguard עושה עוד דברים רבים ומגן מפני מודיפיקציות רבות אחרות. כל מה שרשמתי נכון ספיציפית למקרה בו אנו דנים עכשיו, של hook בעזרת שינוי קוד/שינוי ה־SSDT).

ואיך בדיוק ה־EPT נכנס לתמונה? כיצד הוא יכול לעזור לנו?

ובכן, ה־hypervisor יכול לקבוע שאסורה הרשאת כתיבה וקריאה למקומות מסוימים בזיכרון. כאשר תתבצע קריאה מכתובות מסוימות שמסומנת כאסורה לקריאה, יתרחש vm-exit עם exit reason של EPT_VIOLATION. באמצעות שדה ב־VMCS שנקרא GUEST_PHYSICAL_ADDRESS, נוכל לדעת באיזו כתובת בדיוק התרחש ה־EPT_VIOLATION.

אתם בטח אומרים לעצמכם "טוב... אז?", נניח ואני מעוניין לשים hook על NtOpenProcess לאחר שכבר מצאתי את הכתובת הוירטואלית ב־kernel שבה יושבת הפונקציה. אנחנו יכולים לשנות את הקוד כך שיגרום ל־vm-exit בכניסה לפונקציה - למשל, להחליף את הפקודה הראשונה ב־int 3, ובאמצעות ה־EXCEPTION_BITMAP נבקש לצאת בכל פעם שתתרחש פסיקת Debug. שינוי הקוד של ה־system call הוא פשוט מאוד, מכיוון של־hypervisor יש הרשאת כתיבה לכל מקום, כולל לכתובת ה־kernel-יות של ה־guest.

הקושי הגדול הוא בשאלה כיצד נמנע מ־patchguard "לעלות עלינו"? כאן מתחיל האקשן; אנחנו נסמן את ה־page ששונה (עבר שינוי) כלא קריא, אך כזה שניתן לבצע ממנו הרצה. כך למעשה יתרחש vm-exit בכל פעם שתהיה קריאה מה־page עם exit reason של EPT_VIOLATION. כאשר מתבצעת הרצה של NtOpenProcess, יתבצע vm-exit בעקבות פסיקת ה־debug ששתלנו בתחילת הקוד (EXIT_REASON_EXCEPTION_NMI עם VM_EXIT_INFT_INFO המעיד על פסיקת debug, בדומה ל־ page faults עליהם דנו קודם).

כאשר יתבצע vm-exit כתוצאה מגישה של patchguard לקוד, אנחנו נבצע אמולציה (emulation) של פקודת הקריאה ונחזור חזרה ל־guest. כאשר יתרחש vm-exit כתוצאה מהרצה של ה־system call, נבצע מניפולציות על הפרמטרים שלו, בהתאם לרצוננו, ולאחר מכן נחזיר את השליטה ל־guest.

נציג את האלגוריתם במלואו:

בעת עליית המערכת, בצע:

1. מצא את מיקום ה־SSDT ואת הכתובות הפיזיות של ה־system calls עליהם מעוניינים לעשות hook
2. החלף את הפקודה הראשונה ב־INT3
3. הפעל את הביט המקושר ל־debug exceptions ב־exceptions bitmap (*)
4. ב־EPT, סמן את ה־page בו נמצא הקוד של ה־system call כלא קריא (מותרת הרצה ממנו בלבד)

בעת ריצת המערכת:

- אם התרחש vm-exit כתוצאה מ־EPT violation על הכתובת הפיזית של system call, סימן ש־KPP ניסה לקרוא מה־page. נבדוק מה הפעולה שהוא ביצע, ונשנה את הרגיסטרים בהתאם לפקודה שהוא ניסה לבצע, כך שהוא לא יהיה מודע לשינוי (קוד).
- אם התרחש vm-exit כתוצאה מפסיקת debug והכתובת עליה מצביע rip היא כתובת של system call, סימן שנקרא אחד ה־system calls שעליהם ביצענו hook. נבצע מניפולציה כלשהי ונחזור חזרה ל־guest.

(*) - הפעלת הביט תראה כך:

```
__vmwrite(EXCEPTION_BITMAP, vmread(EXCEPTION_BITMAP) | (1 << INT_BREAKPOINT));
```

סיכום ביניים

לפי הכותרת אתם כבר מבינים שהגיע הזמן שנעצור פה, לפחות לעכשיו. כן כן, אני יודע שאתם מתים להמשיך לקרוא עוד וכל מה שתעשו במהלך החודש הזה זה לחכות בציפייה לחלק הבא שיצא בסוף דצמבר. אין מה לעשות, כולנו נצטרך לעבוד על דחיית הסיפוקים שלנו.

ועכשיו ברצינות, בואו נסכם מה עשינו עד כה, ונדבר על מה שנותר לעבדכם הנאמן להציג בחודש הבא:

בחלק זה של המאמר עברנו על:

- העלאת המחשב עוד לפני מערכת ההפעלה והרצה של קוד ישירות על החומרה
- שימוש בסיסי באחת הפסיקות של ה־BIOS המתארת את מצב הזיכרון במחשב
- הקצאת מקום עבור מבני הנתונים בהם ה־hypervisor יעשה שימוש
- אתחול כמעט מלא של ה־VMCS (נותר לא עוד הרבה, אני מבטיח!)

נותר לנו לדבר על:

- סיום אתחול ה־VMCS
- הפעלת ה־hypervisor על כל הליבות של המעבד (כלומר, כיצד ניתן להריץ קוד על ליבות שונות של המעבד)
- ביצוע hook על הפסיקה int 15h, כך שמערכת ההפעלה תקבל את מפת הזיכרון שה־hypervisor זייף
- העלאת מערכת ההפעלה על גבי ה־hypervisor שלנו

בקיצור, כל הכיף עוד לפנינו!

(אגב, המימוש המלא של כל מה שהצגתי כאן נמצא ב־repository של הפרויקט ב־github)

קצת עליי

שמי אמיר, בן 19 עם כמה חודשים אחדים לפני גיוס. במהלך שני סגרי הקורונה שעברנו חיפשתי איך להעביר את הזמן, ומצאתי את התשובה ב־manuals של Intel - כמו כל אדם בריא כמובן.

מוזמנים לשלוח אליי שאלות במייל: amiryeshooroon@gmail.com.

קרדיטים

- האתר בו פורסמה סדרת המאמרים שבהשראתה נקבע שם המאמר: https://rayanfam.com
- Intel Manuals: https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html
- OSDev: https://wiki.osdev.org/Expanded_Main_Page
- GNU Linker Scripts: https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_chapter/ld_3.html
- GNU Grub: https://www.gnu.org/software/grub/manual/grub/grub.html
- האתר בעזרתו הוכנו השרטוטים: https://creately.com