Digital Whisper Technologic Papers

Hypervisor from Scratch (Native) - חלק ב'

מאת אמיר ישורון

הקדמה

ברוכים הבאים לחלק השני והאחרון בסדרת מאמרים העוסקת בכתיבת native hypervisor. ראשית, למי שלא קרא את החלק הקודם - מומלץ מאוד לגשת ולקרוא אותו בטרם קריאת חלק זה, מכיוון שמרבית מהנושאים כאן הם המשך ישיר של מה שעסקתי בו בחלק הקודם.

בחלק זה נעסוק במספר נושאים, בין היתר נדבר על סיום אתחול ה־VMCS, תקשורת בין יחידות עיבוד, אתחול ה־hypervisor מעל כל ליבות המעבד, ביצוע hook לפסיקות BIOS וכן העלאת מערכת ההפעלה.

קדימה, בואו נצא לדרך!

אתחול ה־VMCS - המשך

את החלק הקודם של המאמר סיימנו בהסבר מקיף על מנגנון ה־EPT, והראנו דוגמא לשימוש שכללה ביצוע hook-ים ל־system calls במערכות 64 ביט של Windows, תוך עקיפת מנגנון ה־patchguard.

עתה, נציג את סיום האתחול של ה־VMCS ונרחיב על פיצ׳רים נוספים המאפשרים לנו לשלוט ולהשפיע באופן ישיר על האופן בו ה־guest רץ.

Unresricted Guest Mode

כאשר דיברנו על המצבים השונים בהם המעבד שלנו יכול לרוץ, הסברנו שכאשר המחשב עולה, המעבד עולה במצב של real address mode על מנת לשמור על תאימות לאחור. לכן, כל מערכות ההפעלה מתחילות להריץ את הקוד הראשוני במצב זה. כאשר נסיים את אתחול ה־hypervisor ונרצה להעלות את מערכת ההפעלה, נצטרך להיות מצויים ב־guest state שרץ ב־real address mode.

למה אני מציג לכם את זה בכלל כ"בעיה"? האם יש כאן משהו שעלול לסבך אותנו בצרות? התשובה לצערנו הרב היא כן, אך הפתרון לבעיה הוא פשוט מאוד. מנגנון הוירטואליזציה באופן default-י איננו מאפשר ל־guest להיות מצוי ב־real address mode, אלא הוא חייב להיות ב־protected mode עם paging או long mode. אם כך, איך אפשר להעלות את מערכת ההפעלה, שצריכה לעלות ב־ real address mode? יש לנו שני פתרונות אפשריים:

- נפנה למייקרוסופט ונבקש מהם לספק לנו image של Windows, ששם המערכת עולה ב־ protected mode או ישירות ב־long mode. ברגע שיהיה לנו את ה־image, ה־guest לא יצטרך לרוץ ב־ real address mode כלל.
- נשתמש ב־unrestricted guest mode - פיצ'ר חדש שנוסף למנגנון הוירטואליזציה בשנת 2010, המאפשר ל־guest לרוץ במצב של real address mode.

אין ספק שמדובר בהתלבטות קשה מאוד, והבחירה היא בהחלט לא קלה, אך כרגע הוחלט שנבחר שנבחר באופציה השניה מבין השניים.

ראשית, על מנת להפעיל מצב זה השימוש ב־EPT הוא חובה. נביא ציטוט מהספר האהוב עלינו:

If either the "unrestricted guest" VM-execution control or the "mode-based execute control for EPT" VM execution control is 1, the "enable EPT" VM-execution control must also be 1.

נפעיל את הביט המקושר אליו ב־Secondary Processor Based Vm-Execution 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	

ובקוד זה נראה כך:

```
__vmwrite(SECONDARY_VM_EXEC_CONTROL, AdjustControls(CPU_BASED_CTL2_ENABLE_EPT |
CPU_BASED_CTL2_UNRESTRICTED_GUEST, MSR_IA32_VMX_PROCBASED_CTLS2));
```

כמו שאתם מבינים, אנחנו מבצעים כתיבה לשדה SECONDART_VM_EXEC_CONTROLS, והערך שם הוא הערך שהפונקציה AdjustControls מחזירה. "מה הפונקציה הזו עושה?", אתם ודאי תוהים. ובכן, intel מציינת כי ישנם ערכים ב־SECONDART_VM_EXEC_CONTROLS שחייבים להיות דולקים, וחלק אחר שחייבים להיות כבויים. ה־MSR המדווח על ערכים אלו הוא MSR_IA32_VMX_PROCBASED_CTLS2. לכן, מה שאנחנו עושים בפונקציה AdjustControls זה לקחת את הערך ב־MSR הזה, ומוסיפים עליו את CPU_BASED_CTL2_ENABLE_EPT ו־CPU_BASED_CTL2_UNRESTRICTED_GUEST.

מעתה והלאה אנחנו רשאים להיכנס ל־real address mode בתור guests ללא חשש.

MSR bitmap

יצא לנו לדבר לא מעט על ה־MSRs במהלך המאמר הקודם והנוכחי. למעשה מדובר באוסף של ערכים אשר משמשים את המעבד לביצוע של פעולות מסוימות; את חלק מהערכים המעבד קובע בעצמו, ואילו חלק אחר אנחנו קובעים. בגלל חשיבותם הגדולה, מנגנון הוירטואליזציה מאפשר לבצע בקרה לפעולות של כתיבה וקריאה לכל ה־MSRs במערכת (שזה בעצם hook לפעולות wrmsr ו־rdmsr). כלומר, ביכולתנו לגרום ל־vm-exit יזום כאשר מתבצעת הפעולה wrmsr וגם rdmsr. שליטה במנגנון זה נעשית על ידי שימוש ב־MSR bitmaps. בפועל, מדובר בוקטור באורך page בזיכרון ובו לכל MSR מקושרים 2 ביטים: ביט אחד קובע האם בקריאת ערכים מה־MSR המסוים יתבצע vm-exit, וביט נוסף הקובע האם בקריאת בעת כתיבת ערכים ל־MSR המסוים יתבצע vm-exit. ב-SINGLE_CPU_DATA, דאגנו להצהיר עליו כך:

```
BYTE msrBitmaps[PAGE_SIZE];
```

ה־bitmaps מחולק(ים) באופן הבא:

- 1024 הבתים הראשונים קובעים האם בעת קריאת ערכים מ־MSRs בתחום שבין 0x0 ל־0x1fff יתבצע vm-exit.
- 1024 הבתים שלאחר מכן קובעים האם בעת קריאת ערכים מ־MSRs בתחום שבין 0xc0000000 ל־0xc0001fff יתבצע vm-exit.
- 1024 הבתים שלאחר מכן קובעים האם בעת כתיבת ערכים ל־MSRs בתחום שבין 0x0 ל־0x1fff יתבצע vm-exit.
- 1024 הבתים שלאחר מכן קובעים האם בעת כתיבת ערכים ל־MSRs בתחום שבין 0xc0000000 ל־0xc0001fff יתבצע vm-exit.

נציג עתה פונקציה המקבלת מספר MSR ושני משתנים בוליאנים - האחד קובע האם יש לגרום ל־vm-exit בקריאה, והשני קובע האם יש לגרום ל־vm-exit בכתיבה.

```
STATUS UpdateMsrAccessPolicy(IN PSINGLE_CPU_DATA data, IN QWORD msrNumber, IN
BOOL read, IN BOOL write)
{
   BYTE range;
   QWORD msrReadIdx, msrWriteIdx;
   BYTE_PTR bitmap;
   if(!IsMsrValid(msrNumber, &range))
       return STATUS_INVALID_MSR;
   msrReadIdx = (range == MSR_RANGE_FIRST) ? msrNumber / 8 : (msrNumber -
0xc0000000) / 8 + 1024;
   msrWriteIdx = (range == MSR_RANGE_FIRST) ? msrNumber / 8 + 2048 : (msrNumber
- 0xc0000000) / 8 + 3072;
   bitmap = data->msrBitmaps;
   if(read)
       bitmap[msrReadIdx] |= (1 << (msrNumber % 8));
   else
       bitmap[msrReadIdx] &= ~(1 << (msrNumber % 8));
   if(write)
       bitmap[msrWriteIdx] |= (1 << (msrNumber % 8));
   else
       bitmap[msrWriteIdx] &= ~(1 << (msrNumber % 8));
   return STATUS_SUCCESS;
}
```

ראשית, מתבצעת בדיקה האם ה־MSR הוא valid-י. במקרה שלא, יוחזר status המסמל שגיאה. כמו כן, הפונקציה IsMsrValid קובעת באיזה תחום נמצא ה־msrNumber שהתקבל. לאחר מכן אנחנו קובעים את האינדקס לקריאה וכתיבה (שימו לב כי לכל MSR מוקצה ביט אחד ולא בית), ומדליקים/מכבים ערך זה ב־bitmaps, בהתאם לערך הבוליאני שהתקבל כפרמטר לפונקציה.

הדבר הנוסף שנצטרך לדאוג לו הוא לקשר את ה־bitmaps ל־VMCS המשוייך לליבה הנוכחית. הקישור של ה־bitmaps והפעלה שלו מתבצעים כך:

```
__vmwrite(CPU_BASED_VM_EXEC_CONTROL,
AdjustControls(CPU_BASED_ACTIVATE_MSR_BITMAP |
CPU_BASED_ACTIVATE_SECONDARY_CONTROLS, MSR_IA32_VMX_PROCBASED_CTLS));
__vmwrite(MSR_BITMAP, VirtualToPhysical(cpuData->msrBitmaps));
```

השורה הראשונה מפעילה את הביט המקושר ל־MSR bitmap ב־CPU_BASED_VM_EXRC_CONTROL (ראו תמונה למטה). השורה השניה אחראית על שיוך המשתנה msrBitmaps ל־VMCS הנוכחי.

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	

לאחר שהצגנו את המנגנון ואת אופן השימוש בו, נראה כמובן דוגמא לשימוש:

בחלק הקודם הצגתי שיטה המאפשרת לבצע hook ל־system calls במערכות 64bit של Windows. עתה, נציג שיטה פשוטה יותר, אשר פועלת במערכות 32bit, וכמובן שגם המימוש שלה יהיה פשוט יותר. במערכות 32bit בלבד, כאשר מערכת ההפעלה עולה היא מאחסנת ב־MSR ששמו IA32_SYSENTER_EIP_MSR את הכתובת של ה־system call dispatcher אשר מעביר את הקריאות ל־system calls השונים.

במערכות Windows, קריאה ל־NtWriteFile (למשל) ראשית מגיעה ל־ntdll, שם יש את הקוד הבא:

```
ntdll!NtWriteFile:
    mov eax, 7h
    mov edx, offset SharedUserData!SystemCallStub
    call dword ptr [edx]
    ret 24h
```

פקודת ה־call מבצעת קריאה לקוד הבא (הנמצא גם הוא ב־ntdll):

```
Ntdll!KiFastSystemCall:
    mov edx, esp
    sysenter
```

הפקודה sysenter משגרת אותנו ל־kernel mode, כאשר לבסוף נגיע לקוד של ה־system call ב־ntoskrnl.exe, לאחר כמובן שה־dispatcher יפנה אותנו לשם.

איך זה קשור ל־MSR bitmap? איך הכלי הזה בדיוק זה יכול לעזור לנו על מנת לבצע hook על system calls? ובכן, כדי להבין את התשובה לשאלה הזו חשוב להבין כיצד הפקודה sysenter עובדת. נצטט מההסבר על הפקודה sysenter באתר c9x:

When the SYSENTER instruction is executed, it loads the instruction pointer from the IA32_SYSENTER_EIP_MSR into the EIP register

אנחנו מבינים שהערך של IA32_SYSENTER_EIP_MSR נכנס ל־EIP בעת ביצוע הפקודה sysenter. לכן, אם נוכל לשתול שם ערך לא valid-י כלשהו (למשל 0x93939393), אזי יתרחש page-fault בכל פעם שמערכת ההפעלה תריץ את הפקודה sysenter. אם ה־hypervisor יתפוס את ה־page-faults השונים (זוכרים את ה־EXCEPTION_BITMAP?), הוא יכול לבדוק את הכתובת שעליה אירע ה־page fault. אם הכתובת היא 0x93939393 - הרי שהורצה הפקודה sysenter. אם הערך היא כל דבר אחר - יש להזריק page-fault על הכתובת הזו חזקה ל־guest. נציג עתה אלגוריתם מלא:

בעת אתחול המערכת, בצע:

- הפעל את הביט המקושר לכתיבה לערך של IA32_SYSENTER_EIP_MSR ב־MSR bitmap.
- הפעל את הביט המקושר ל־page-fault שב־EXCEPTION_BITMAP ב־VMCS.

בעת אתחול מערכת ההפעלה:

- כאשר התבצע vm-exit בגלל כתיבה ל־MSR זה, שמור את הערך שמערכת ההפעלה ניסתה לכתוב במבני הנתונים של ה־hypervisor. כמו כן, כתוב ל־IA32_SYSENTER_EIP_MSR ערך זבל כלשהו (0x93939393). לאחר מכן כבה את הביט המקושר לכתיבה לערך של ל־IA32_SYSENTER_EIP_MSR ב־MSR bitmap.

בעת ריצת מערכת ההפעלה:

- אם התרחש vm-exit כתוצאה מ־page-fault, בדוק את הערך שעליו התבצע ה־fault (כיצד? בדקו!).
- במידה והכתובת היא 0x93939393, הרי שמדובר ב־system call. הרץ קוד כלשהו שיטפל ב־hook ולאחר מכן עדכן את EIP להיות הערך המקורי של IA32_SYSENTER_EIP_MSR, אשר שמור במבני הנתונים של ה־hypervisor.
- אחרת, לא מדובר ב־system call ויש להזריק את ה־page-fault חזרה ל־.guest

כך למעשה ביצענו hook על כל ה־system calls במערכת, כולל אלו של win32k! (שגם מבצעים קריאה ל־sysenter)

חזרה ל־guest state

לאחר שסיימנו להריץ את הקוד שמאתחל את ה־VMCS, יש לזכור שאנחנו עדיין רצים בתור ה־host, כלומר הקוד כרגע רץ ב־context של ה־hypervisor. הרגע בו נכנסנו ל־context של ה־hypervisor היה כאשר הרצנו את הפקודה vmxon.

בחלק הקודם הזכרתי שאת מערכת ההפעלה נהיה חייבים להעלות בתור guest, כלומר עלינו לצאת מה־context הנוכחי בו אנחנו רצים. לכן, עתה נציג את הקוד שאחראי על המעבר חזרה ל־guest state:

```
if(SetupCompleteBackToGuestState() != STATUS_SUCCESS)
{
  // Should never arrive here
  Print("FLAGS: %8, instruction error: %8\n", __readflags(), vmread(VM_INSTRUCTION_ERROR));
  ASSERT(FALSE);
}
Print("Done initialization on core #%d\n", cpuData->coreIdentifier);
```

Hypervisor State

Guest State

אלו השורות שלמעשה חותמות את פונקציית האתחול של ה־hypervisor (ששמה הוא InitializeSingleHypervisor. מדוע זה שמה? נגלה בהמשך). נסביר את הקוד:

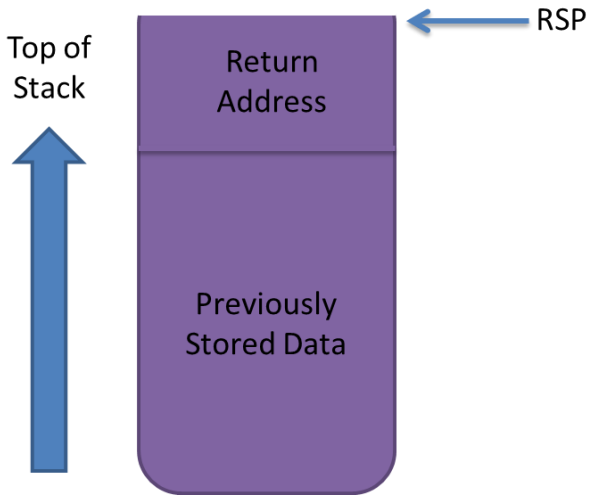
במידה והחזרה ל־guest נכשלת, אנחנו מדפיסים את ה־flags שיעזור לנו להבין מה הבעיה (על פי ה־manuals, כמובן). כמו כן, הערך של VM_INSTRUCTION_ERROR ב־VMCS יעזור לנו גם כן במצב זה. לאחר מכן נכנס ללולאה אינסופית (ASSERT) והמחשב בעצם יתקע.

אם כך, מה הוא הקסם הקורה בפונקציה SetupCompleteBackToGuestState? מה בדיוק רץ שם? בואו נסתכל יחד:

```
SetupCompleteBackToGuestState:
   mov rax, 0x0000681c
   vmwrite rax, rsp
   vmlaunch
```

אנחנו מעבירים ל־rax את הערך ב־VMCS של GUEST_RSP (כלומר, המקום ב־VMCS אליו נכתב הערך של rsp של ה־guest ברגע שיתבצע vm-entry). לאחר מכן אנחנו כותבים לשדה הזה את הערך הנוכחי של rsp. מדוע אנחנו עושים זאת? ובכן, בזכות השורה הזו ברגע שיתרחש vm-entry, ה־stack של ה־guest יהיה בדיוק אותו ה־stack שיש כרגע ב־host.

מה הערכים שיושבים שם? בואו נסתכל על השרטוט הבא, תוך כדי שנזכר כיצד מנוהלות קריאות לפונקציות ב־x86:



מכיוון שכתובת החזרה תמיד מאוחסנת ב־stack, אנחנו מבינים שברגע שתבוצע הפקודה ret, המעבד יקח את כתובת החזרה מה־stack ויחזור להריץ את הפקודה הבאה שאחרי הקריאה לפונקציה SetupCompleteBackToGuestState.

לאחר כתיבת הערך הנוכחי של rsp ל־VMCS, אנחנו מריצים את הפקודה vmlaunch - שלמעשה גורמת ל־vm-entry, כלומר היא גורמת לטעינת כל הערכים ב־VMCS למקומות המתאימים (הערך ב־GUEST_CR0 יכנס ל־cr0, הערך ב־GUEST_RSP יכנס ל־rsp, הערך ב־GUEST_CR3 יכנס ל־cr3 וכו'). בין היתר, הערך שבשדה HOST_RIP יכנס לאוגר rip, והמעבד ימשיך להריץ קוד מהכתובת שמופיעה שם. נזכיר מה כתבנו לשדה זה בחלק הקודם של המאמר:

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

וכיצד נראית הפונקציה VmmToVm? חלקכם ודאי כבר הבינו, שיש בה פקודה אחת בלבד (טוב, כמעט):

```
VmmToVm:
   mov rax, STATUS_SUCCESS
   ret
```

מכיוון ש־rax תמיד יכיל את ערך החזרה מהפונקציה, אנחנו שמים שם STATUS_SUCCESS (שזה למעשה 0). הדבר הבא שנעשה הוא להריץ את הפקודה ret, שכפי שהסברנו תגרום למעבד לחזור להריץ קוד בכתובת שכרגע נמצאת בראש ה־stack, שזה בדיוק שורה אחת אחרי הקריאה לפונקציה SetupCompleteBackToGuestState.

בום! השורה הבאה שתרוץ היא השורה שעליה מצביע החץ האדום, שלצידו הכיתוב Guest State, וזאת לאחר שהתבצע vm-entry (כלומר עברנו מה־context של ה־hypervisor ל־guest).

תקשורת בין יחידות עיבוד

אוקיי, אז סיימנו את אתחול ה־VMCS. החלק הקשה ללא ספק מאחורינו. עכשיו אנחנו נרגשים להמשיך הלאה ולהעלות את המערכת, כי הרי סיימנו הכל! רגע... אולי לא. חכם זקן אמר פעם "על כל ליבה של המעבד ה־hypervisor רץ כ־process יחיד ו־thread יחיד" (רמז: זה הוא אותו האחד הכותב את שורות אלו). למה הכוונה בדיוק במשפט הזה? אסביר; כל הקוד שהרצנו עד עתה שדאג לאתחל את ה־VMCS רץ מעל ליבה מסוימת של המעבד וה־hypervisor מוכן לשימוש מעל הליבה הזאת בלבד. השאלה המתבקשת היא מה עם שאר יחידות העיבוד?

ובכן, יש לי חדשות טובות וחדשות רעות. החדשות הרעות הן שצריך לעשות את כל התהליך המפרך הזה עבור כל הליבות במערכת. החדשות הטובות הן שהקוד כבר כתוב, רק צריך למצוא דרך להריץ אותו על כל הליבות האחרות. כאן נכנס לתמונה מנגנון ה־APIC. בואו נדבר עליו קצת.

ראשי התיבות של APIC הן Advanced Programmable Interrupt Controller - מדובר במנגנון המאפשר לשלוח Interrupts בין ליבות שונות של המעבד, מה שמכונה Inter Processor Interrupt. למעשה כאשר המחשב שלנו עולה, אחת הליבות מקבלת את התואר הנחשק אליו כל ליבה שואפת במהלך חייה, הלא הוא ה־BSP, קיצור של Bootstrap Processor. הליבה הזו אחראית על הרצת הקוד של ה־Bootloader, והיא אחראית גם על "הפעלת" שאר הליבות במערכת (באמצעות מנגנון ה־APIC כמובן), שכן בעת עליית המחשב רק ליבה אחת באמת מריצה קוד.

אם כך, אנחנו מבינים שעלינו להשתמש בכלי הזה שהצגתי על מנת להריץ את פונקציית האתחול של ה־hypervisor על שאר הליבות במערכת. לאחר הרצת פונקציית האתחול, נריץ את הפקודה hlt על מנת להחזיר את הליבה למצב "שינה", עד שיתקבל ה־Interrupt הבא שיחזיר אותה לחיים.

איך שולחים Interrupt-ים בין הליבות השונות אתם שואלים? בואו נצלול ל־manuals ונצא עם תשובה לשאלה הזו. סתם, סתם! אל תברחו... אני פשוט אציג כאן את הדרך המלאה, ואחסוך מכם את הקריאה המפרכת.

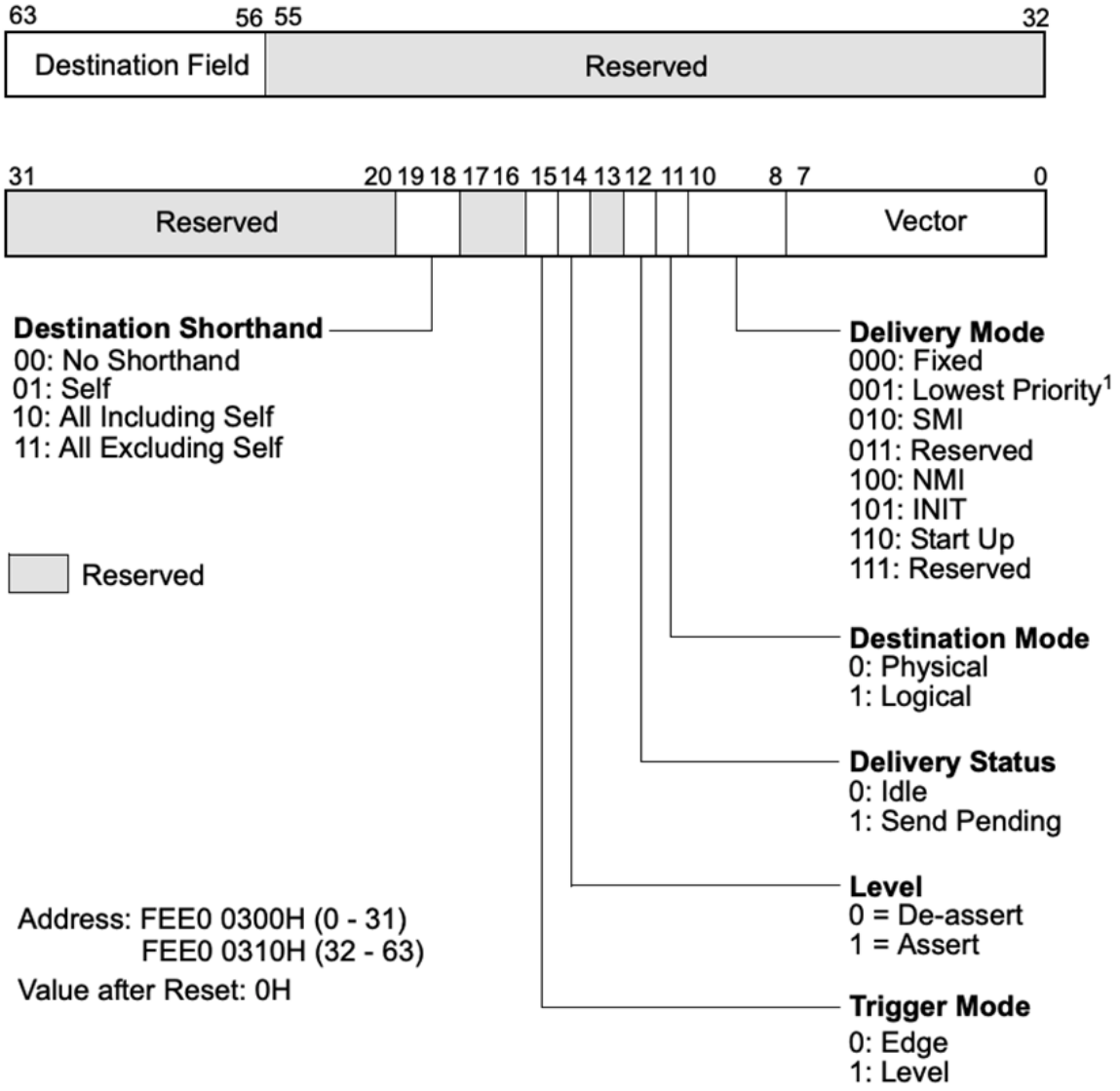
ICR - Interrupt Command Register

כאן טמון הקסם הגדול. מדובר ב"אוגר" בגודל 8 בתים אשר אחראי להעברת IPIs - Inter Processor Interrupts בין הליבות השונות של המערכת. איך הדבר הזה בנוי? ובכן, יש שתי גרסאות של מנגנון ה־APIC, ובכל אחת מהן ה־ICR עובד באופן מעט שונה.

Standard APIC

אני כבר אגיד שזה הוא לא המודל בו אנחנו נשתמש, אבל חשוב להכיר אותו לפני שנעבור למודל החדש יותר (x2APIC). במודל הסטנדרטי (הישן יותר) של מנגנון ה־APIC, "אוגר" ה־ICR הוא בגודל 8 בתים והוא נמצא במקום קבוע בזיכרון - 4 הבתים הראשונים בכתובת הפיזית 0xFFE00300, ו־4 הבתים האחרונים בכתובת 0xFFE00310. כאשר מתבצעת פעולת הכתיבה ל־4 הבתים התחתונים הפסיקה "משוגרת" לליבה בה אנחנו חפצים.

ה־layout של האוגר המיוחד הזה נראה כך:



[קרדיט: Intel SDM, Volume 3A]

נסביר על השדות פה, ונגיד מה רלוונטי לשימוש שלנו:

- Vector - ה־vector number של הפסיקה אותה אנחנו מעוניינים לשגר. למרות ההגדרה הפשוטה של השדה, יש שני מקרים יוצאי דופן הרלוונטים לנו - אם סוג הפסיקה שאנחנו מבצעים היא פסיקת INIT (חייבת להתבצע לפני שיגור של פסיקות אחרות) - אזי הערך כאן צריך להיות 0. אם סוג הפסיקה היא פסיקת SIPI (קיצור של Start up Inter Processor Interrupt, נרחיב עליה בעוד רגע) - אזי הערך כאן צריך להיות מספר ה־page בו נמצא הקוד שצריך לרוץ. כלומר, אם אני רוצה שהליבה שאני שולח אליה את ה־IPI תתחיל להריץ את הקוד שיש בכתובת 0x8000, נשים ב־vector את הערך 0x8 ונשגר SIPI.
- Delivery Mode - מסמל את סוג הפסיקה שיש להריץ. אנחנו נשתמש ב־INIT ו־Start up Interrupt על מנת לאתחל את ה־hypervisor על כל הליבות של המעבד.INIT - כאשר ליבה מקבלת פסיקה מסוג INIT, היא נכנסת למצב של Wait-For-Sipi. הפירוש של SIPI כפי שכבר רשמתי הוא Start up Inter Processor Interrupt. כלומר לאחר קבלת INIT, הליבה מחכה לפסיקה מסוג SIPI שתועבר אליה.

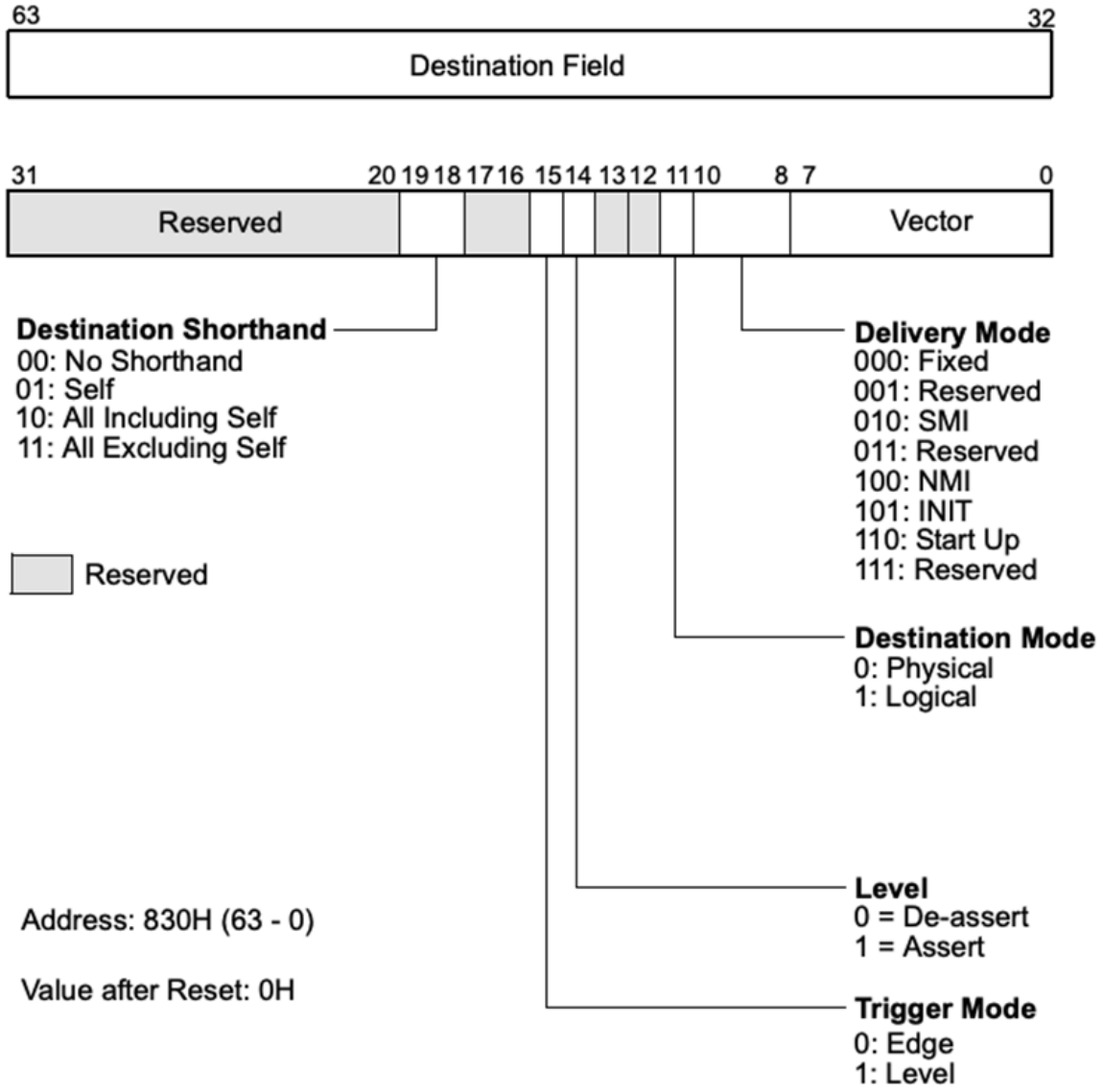
  SIPI - בהתאם לכתוב למעלה, מדובר בפסיקה שלאחר קבלתה הליבה מתחילה להריץ קוד בכתובת 0x1000 * Vector (כלומר הערך בשדה ה־vector כפול 0x1000, כגודל כל page). לכן, מה שאנחנו נעשה יהיה לכתוב לכתובת כלשהי בזיכרון את הקוד שאנחנו מעוניינים שהליבה תריץ, נחלק אותה ב־0x1000, ואת התוצאה נשמור בשדה ה־vector.
- Destination Mode - אנחנו נשתמש בערך 0 עבור physical. המשמעות של ערך זה קובעת שבשדה ה־Destination Field יהיה את המזהה הייחודי שיש לכל ליבה, בהתאם למה שנקבע על ידי ה־BIOS בעת עליית המערכת (זוכרים את coreIdentifier שנמצא ב־SINGLE_CPU_DATA?).
- Destination Field - בהתאם למה שרשמתי ב־Destination Mode, שדה זה יכיל את המזהה הייחודי שיש לכל ליבה.

על שאר השדות אני לא אפרט, מכיוון שהם פחות קריטיים להבנה כאן.

x2APIC

עם הזמן נוצר הצורך להביא חלופה למנגנון ה־APIC הקיים ולהחליף אותו במנגנון קצת יותר מתקדם, אך עם זאת לשמור על דמיון מקסימלי ל־APIC המקורי. מה גם שכתיבה לכתובת 0xFFE00300 על מנת לגרום להרצה של פסיקה עלולה להיתפס כפעולה פרימיטיבית ולא מובנת.

גבירותיי ורבותי - הכירו את x2APIC! מדובר במנגנון כמעט זהה לזה של ה־APIC, ומבחינתנו (כלומר, במסגרתה שימוש בכתיבת ה־hypervisor) ההבדל יהיה רק באופן שבו ה־ICR בנוי. בואו נביט על המבנה החדש שלו:



[קרדיט: Intel SDM, Volume 3A]

כמו שאתם יכולים לראות, השינוי כמעט ולא ישפיע עלינו, והוא בא לידי ביטוי בעובדה שכל ארבעת הבתים העליונים מוקדשים ל־Destination Field, שזה הוא המזהה הייחודי של כל ליבה.

כיצד מתבצעת הכתיבה ל־ICR ב־x2APIC אתם שואלים?

הכתיבה מתבצעת באמצעות כתיבת ערכים ל־MSR שמספרו 0x830. כלומר, אנחנו נשתמש ב־Intrinsic של writemsr על מנת לשגר IPI-ים. ללא ספק מדובר בדרך מעט יותר נחמדה מאשר כתיבה ישירה לזיכרון. יאללה, בואו נציג את המימוש!

ראשית, נוודא שאפשר להשתמש ב־x2APIC. אם המנגנון איננו זמין - נבצע assertion. אם הוא אכן זמין, נפעיל אותו. זה נראה כך:

```
STATUS DetectX2APICAvailability()
{
   QWORD tmp, ecx;
   __cpuid(1, 0, &tmp, &tmp, &ecx, &tmp);

   return (ecx & CPUID_2XAPIC_AVAILABLE) ? STATUS_SUCCESS :
STATUS_2XAPIC_NOT_AVAILABLE;
}

VOID EnableX2APIC()
{
   ASSERT(DetectX2APICAvailability() == STATUS_SUCCESS);
   __writemsr(MSR_IA32_APIC_BASE, __readmsr(MSR_IA32_APIC_BASE) | (1 << 10) | (1
<< 11));
}
```

את זמינות ה־x2APIC אנחנו בודקים באמצעות cpuid עם main leaf של 1 ו־sub-leaf של 0. באמצעות הביט המשוייך לזמינות APIC2x באוגר ecx (הביט ה־21) אנחנו יכולים לדעת האם הוא אכן זמין או לא. לאחר מכן, אנחנו מפעילים את ביט 10 ו־11 ב־MSR ה־MSR_IA32_APIC_BASE. החל מרגע זה, השימוש ב־Standard APIC אסור, והוא עלול לגרום ל־General Protection Exception.

שליחת IPI תיעשה באופן הבא:

```
VOID X2APICIssueIPI(IN QWORD destination, IN QWORD vector, IN QWORD
deliveryMode, IN QWORD lvl)
{
   QWORD ipi = (destenation << APIC_DESTINAION_BIT_OFFSET) | deliveryMode |
vector | lvl;
   __writemsr(MSR_IA32_X2APIC_ICR, ipi);
}
```

כמו שאתם יכולים לנחש - הערך של MSR_IA32_X2APIC_ICR הוא בדיוק 0x830, בהתאם לתרשים שראינו למעלה מה־manual של Intel.

עתה, כל מה שנותר לעשות הוא לעבור ליבה-ליבה ולשגר אליה IPI שיגרום לאתחול ה־hypervisor שלנו על הליבה הזו. לאחר שהתהליך הזה יסתיים, אנחנו יכולים להכריז על כך שה־hypervisor שלנו מותקן כהלכה על כל ליבות המעבד, ונהיה קרובים יותר מתמיד להעלאת מערכת ההפעלה.

הרצת פונקציית האתחול על כל הליבות נעשית כך:

```
STATUS ActivateHypervisorOnProcessor(IN QWORD processorId, IN PSINGLE_CPU_DATA
cpuData)
{
   CopyMemory(APIC_FUNC_ADDRESS, ApicStart, ApicEnd - ApicStart);
   X2APICIssueIPI(processorId, 0, APIC_INIT_INTERRUPT, APIC_LEVEL_DEASSERT);
   Sleep(50);
   *(QWORD_PTR)CPU_DATA_ADDRESS = cpuData;
   *(QWORD_PTR)(CPU_DATA_ADDRESS + sizeof(QWORD)) = InitializeSingleHypervisor;
   *(BYTE_PTR)SEMAPHORE_LOCATION = 0;
   X2APICIssueIPI(processorId, APIC_FUNC_ADDRESS / PAGE_SIZE,
APIC_SIPI_INTERRUPT, APIC_LEVEL_ASSERT);
   Sleep(50);
   for(QWORD i = 0; i < 0x11ffffffffffffffULL && (*(BYTE_PTR)SEMAPHORE_LOCATION
== 0); i++);
   return *(BYTE_PTR)SEMAPHORE_LOCATION == 0 ? STATUS_APIC_SIPI_FAILED :
STATUS_SUCCESS;
}
```

אין מה להילחץ כלל. נסתכל על הקוד הזה יחד ונפרש אותו:

תחילה, אנחנו מעתיקים את הקוד שלנו לכתובת מסוימת בזיכרון, שהערך שלה מוגדר על ידי APIC_FUNC_ADDRESS. תוכלו להתרשם מהקוד שאנחנו מעתיקים כאן (הוא מעט ארוך, לכן בחרתי שלא להציג אותו כאן). לאחר מכן, אנחנו מריצים INIT על הליבה שאותה אנו מאתחלים. זה מתבצע באמצעות הפונקציה שהצגנו למעלה.

הצעד הבא הוא להעביר את הפרמטרים הרלוונטיים לליבה עליה הקוד ירוץ - האחד זה את המבנה SINGLE_CPU_DATA השייך אליה, והשני הוא את הכתובת של הפונקציה InitializeSingleHypervisor (שעושה את האתחול ל־VMCS, מההתחלה ועד הסוף).

אנחנו גם יוצרים לנו מעין lock פרימיטיבי, שהליבה הנוכחית מגדירה כ־0, והליבה שאליה אנחנו שולחים IPI תדאג לשנות את הערך הזה להיות 1 כאשר היא מסיימת את האתחול. כך למעשה הליבה הנוכחית נכנסת ללופ עד אשר האתחול יסתיים.

כמו שאתם ודאי מנחשים - השגרה הזו תיקרא שוב ושוב עבור כל הליבות במערכת, ממש כך:

```
for(QWORD i = 1; i < numberOfCores; i++)
   ASSERT(ActivateHypervisorOnProcessor(processorIdentifires[i], sharedData-
>cpuData[i]) == STATUS_SUCCESS);
   // Initialize hypervisor on BSP
   InitializeSingleHypervisor(sharedData->cpuData[0]);
```

אנחנו מבצעים אתחול עבור כל הליבות במערכת, פרט לליבה הנוכחית (ה־BSP). לאחר שה־BSP דואג לשגר IPI שיאתחל את כל הליבות במערכת, הוא מבצע את האתחול בעצמו על ידי קריאה ישירה ל־InitializeSingleHypervisor.

זיוף מפת הזיכרון

הגיע הזמן להשתמש לראשונה בכוח שמקנה לנו מנגנון הוירטואליזציה. כמו שתיארתי בחלק הראשון של המאמר, נרצה נבצע hook על הפסיקה שמעבירה את מפת הזיכרון למערכת ההפעלה, ונעביר לה את מפת הזיכרון שזייפנו בחלק הקודם (זוכרים שהסתרנו את האזור בזיכרון בו ה־hypervisor עושה שימוש?). על מנת להבין כיצד לעשות את זה, נצטרך להיות בקיאים באופן בו פסיקות ה־BIOS עובדות ב־real address mode. אסביר בקצרה על האופן בו מנוהלות פסיקות אלו.

כאשר המחשב עולה לראשונה, ה־BIOS כמו שראינו בחלק הקודם של המאמר מספק מערכת input-output מינימלית, והוא גם מספק תשתית מינימלית לקבלת מידע אודות רכיבי החומרה במערכת. מעבר לשני תפקידים חשובים אלו, ה־BIOS יוצר IVT - קיצור של Interrupt Vector Table שמתחיל בכתובת הפיזית 0 (כן כן, ממש בביט הראשון של ה־RAM).

ה־IVT בנוי כך:

IVT Offset	INT #	Description
0x0000	0x00	Divide by 0
0x0004	0x01	Reserved
0x0008	0x02	NMI Interrupt
0x000C	0x03	Breakpoint (INT3)
0x0010	0x04	Overflow (INTO)
0x0014	0x05	Bounds range exceeded (BOUND)
0x0018	0x06	Invalid opcode (UD2)
0x001C	0x07	Device not available (WAIT/FWAIT)
0x0020	0x08	Double fault
0x0024	0x09	Coprocessor segment overrun
0x0028	0x0A	Invalid TSS
0x002C	0x0B	Segment not present
0x0030	0x0C	Stack-segment fault
0x0034	0x0D	General protection fault
0x0038	0x0E	Page fault
0x003C	0x0F	Reserved
0x0040	0x10	x87 FPU error
0x0044	0x11	Alignment check
0x0048	0x12	Machine check
0x004C	0x13	SIMD Floating-Point Exception
0x00xx	0x14-0x1F	Reserved
0x0xxx	0x20-0xFF	User definable

[קרדיט: osdev]

כל הפנייה לפסיקה תופסת 4 בתים בזיכרון, וכאשר מתבצעת הפקודה int X, המעבד שלנו קופץ לקוד שהטבלה הזו מורה לו (כל האמור כאן נכון בתנאי שהמעבד נמצא ב־real address mode והוגדר לו ה־IVT כהלכה).

אחד הדברים הראשונים שה־HAL (קיצור של Hardware Abstraction Layer) של Windows עושה בעת עליית המערכת זה לבצע int 0x15. הפסיקה הזו אמנם איננה מופיעה בטבלה הנ"ל, אך היא הפכה לסטנדרט ברמה כזו שה־kernel של מערכות הפעלה אשר רצות מעל BIOS משתמשות בו.

לפני שנחשוב יחד על איך לבצע את ה־hook על int 0x15, נצטרך להבין כיצד נראית החלוקה הפנימית בכל תא של ה־IVT (כלומר, כיצד בנויים ארבעת הבתים של כל מופע ב־IVT). בפועל, אין שם הפנייה לכתובת ישירה בזיכרון, אלא יש חלוקה פנימית ל־2 חלקים: segment ו־offset:

```
+-----------+-----------+
|  Segment  |  Offset   |
+-----------+-----------+
4           2           0
```

[קרדיט: osdev]

כלומר, המעבד יבצע קפיצה לכתובת: segment * 0x10 + offset (כך פועל מנגנון ה־segmentation ב־real mode).

עכשיו שאנחנו כבר מבינים כיצד בנוי כל תא ותא ב־IVT, נרצה לשנות אותו כך שתתבצע "קפיצה" ל־hypervisor שלנו שיספק את התשובה המזוייפת. כלומר, אנחנו רוצים לגרום ל־vm-exit באופן יזום, כך שבפונקציה HandleVmExitEx נדע לזהות מתי בדיוק התבצע vm-exit. כיצד ניתן לגרום לדבר כזה? למזלנו, קיימת פקודה הנקראת vmcall הפועלת באופן דומה מאוד לפקודה syscall, אשר אחראית לביצוע המעבר מ־user mode ל־kernel mode.

הפקודה vmcall גורמת ל־vm-exit, עם exit reason של EXIT_REASON_VMCALL. כאשר נזהה את exit reason זה ב־hypervisor, נבדוק מהיכן הגיעה הקריאה (כלומר, האם האוגר ip מצביע על הכתובת של ה־hook שלנו). אם כן, סימן שהתבצעה קריאה כלשהי ל־int 0x15. שימו לב שבשלב זה עדיין לא בטוח שמערכת ההפעלה ביקשה את מפת הזיכרון, אלא ייתכן שהיא ביצעה int 0x15 במטרה אחרת לחלוטין.

לכן, נבדוק את הערך של eax; אם הערך ב־eax הוא 0xE820, סימן שמערכת ההפעלה אכן ניסתה לקבל את מפת הזיכרון מה־BIOS. אנחנו נספק לה את המפה המזוייפת ונחזור חזרה ל־guest. אם הערך ב־eax הוא לא 0xE820, פשוט נחזיר את ה־guest לכתובת האמיתית של int 0x15 ונתן לקוד הפסיקה המקורי לטפל בקריאה, ללא התערבות נוספת של ה־hypervisor.

ראשית, בואו נערוך את ה־IVT כרצוננו:

```
STATUS SetupE820Hook(IN PSHARED_CPU_DATA sharedData)
{
    DWORD_PTR ivtAddress = PhysicalToVirtual(0);
    QWORD segment = (ivtAddress[0x15] >> 16) & 0xffffULL;
    QWORD offset = ivtAddress[0x15] & 0xffffULL;
    BYTE vmcall[] = { 0x0f, 0x01, 0xc1 };
    CopyMemory(E820_VMCALL_GATE, vmcall, 3);
    ivtAddress[0x15] = E820_VMCALL_GATE;
    sharedData->int15Offset = offset;
    sharedData->int15Segment = (segment << 4);
    return STATUS_SUCCESS;
}
```

קודם כל אנחנו שומרים את הערכים הנוכחיים ב־IVT, על מנת להפנות לשם פסיקות 0x15 עם ערך ב־eax שהוא לא 0xE820. מיד לאחר מכן אנחנו כותבים לאיזושהי כתובת בזיכרון (שערכה שווה לערך הקבוע E820_GATE_CALL. בפועל, הערך שלו הוא 0xfffc - אין לכך שום סיבה מיוחדת. היה אפשר לבחור בכל כתובת שהיא לא reserved על ידי ה־BIOS) את ה־opcode של vmcall, ומשנים את הערך של ה־IVT במקום ה־0x15 להצביע לכתובת בה שתלנו את פקודת ה־vmcall (שימו לב שה־segment הוא 0, מכיוון שהכתובת נכנסת ב־16 ביטים, ואין צורך ב־segment).

עתה, נראה את הקוד ב־HandleVmExitEx שתטפל במקרה של EXIT_REASON_VMCALL:

```
case EXIT_REASON_VMCALL:
{
        HandleVmCall(guestState);
        break;
}
```

וכמובן, מה שמעניין אותנו נמצא בפונקציה HandleVmCall:

```
STATUS HandleVmCall(IN PCURRENT_GUEST_STATE data)
{
    PREGISTERS regs = &data->guestRegisters;
    if(regs->rip == E820_VMCALL_GATE)
    {
        if(regs->rax == 0xE820)
        {
            BOOL carrySet = FALSE;
            if(regs->rbx >= data->currentCPU->sharedData->memoryRangesCount)
            {
                carrySet = TRUE;
                goto EmulateIRET;
            }
            regs->rax = E820_MAGIC;
            regs->rcx = (regs->rcx & ~(0xffULL)) | 20;
            CopyMemory(vmread(GUEST_ES_BASE) + (regs->rdi & 0xffffULL),
                &(data->currentCPU->sharedData->allRam[regs->rbx++]), sizeof(E820_LIST_ENTRY));
            carrySet = FALSE;
            if(regs->rbx == data->currentCPU->sharedData->memoryRangesCount)
                regs->rbx = 0;
EmulateIRET:
            regs->rip = (DWORD)(*(DWORD_PTR)(vmread(GUEST_SS_BASE) + regs->rsp));
            WORD csValue = *(WORD_PTR)(vmread(GUEST_SS_BASE) + regs->rsp + 2);
            WORD flags = *(WORD_PTR)(vmread(GUEST_SS_BASE) + regs->rsp + 4);
            if(carrySet)
```

```
                flags |= RFLAGS_CARRY;
            else
                flags &= ~(RFLAGS_CARRY);
            __vmwrite(GUEST_CS_BASE, csValue << 4);
            __vmwrite(GUEST_CS_SELECTOR, csValue);
            regs->rsp += 6;
            return STATUS_SUCCESS;
        }
        regs->rip = data->currentCPU->sharedData->int15Offset;
        __vmwrite(GUEST_CS_BASE, csValue << 4);
        __vmwrite(GUEST_RFLAGS, flags);
        __vmwrite(GUEST_CS_SELECTOR, csValue);

        return STATUS_SUCCESS;
    }

    return STATUS_UNKNOWN_VMCALL;
}
```

ראשית מתבצעת בדיקה על הערך שבאוגר eax. אם הוא לא 0xE820, אנחנו לא מטפלים בה. אם הערך הוא אכן 0xE820, אז מתחיל כל האקשן; מי מכם שקרא את החלק הקודם ודאי זוכר כיצד עובדת הפסיקה int 0x15, וכיצד היא מחזירה את התשובה. מה שאנחנו עושים כאן למעשה זה לסמלץ את פעולת הפסיקה, ובכל פעם אנחנו מחזירים את האיבר הבא מה־memoryRange ששמרנו בצד (גם כן, בחלק הקודם של המאמר). לאחר שאנחנו מטפלים בתוכן הפסיקה, יש להחזיר את ה־guest לרוץ לנקודה שלפני הקריאה. כדי לעשות זאת, אנחנו צריכים לסמלץ את הפקודה iret, אשר נמצאת בסוף כל פסיקת real mode-ית. בפועל הפקודה הזו מוציאה מה־stack את הערך החדש של ip, את הערך של sp וגם של אוגר הדגלים. בשורות שלאחר ה־label שנקרא "EmulateIRET" זה בדיוק מה שאנחנו עושים - מוציאים את הערכים המתאימים מה־stack ומבצעים השמה לאוגרים המתאימים.

טעינת מערכת ההפעלה

הגענו לאבן הדרך האחרונה שלנו. לפני שניגש ונדבר על טעינת מערכת ההפעלה מהדיסק, נעשה סיכום של הדברים שעשינו עד כה:

- אתחלנו את ה־hypervisor על כל ליבות המעבד, וכל הליבות שהן לא ה־BSP "ישנות" ומחכות לאתחול ע"י מערכת ההפעלה. כרגע על כל הליבות אנחנו נמצאים ב־guest state, וכשה־HAL של Windows יבצע אתחול, הוא ירוץ מעל ה־hypervisor שאותחל כבר על כל הליבות, באמצעות מנגנון ה־APIC.
- שתלנו hook על הפסיקה int 0x15, כך שה־hypervisor יספק את מפת הזיכרון המזוייפת לכל מי שמבצע שימוש בפסיקה.
- דאגנו לקיום ואתחול מבני הנתונים השונים ההכרחיים לריצה תקינה של מנגנון הוירטואליזציה.

עתה הגיע הזמן לטעון מערכת הפעלה שתרוץ מעל ה־hypervisor שלנו. ממש התרגשות שיא!

BIOS vs. UEFI

בחלק הראשון של המאמר, רשמתי הערה קטנה בתחילתו שה־hypervisor נכתב עם תמיכה במערכות BIOS בלבד, וללא תמיכה ב־UEFI. אני לא הולך להיכנס לעומק ולעמוד על ההבדלים בין השניים, אבל בכל זאת בכמה מילים אנסה להסביר למה הכוונה.

עד ל־2005-2006 ה־firmware הנפוץ ביותר בשוק ה־PCs היה לא אחר מה־BIOS. כמו שרשמתי בחלק הקודם, מדובר ברכיב חומרה-תוכנה שדואג לאתחול בסיסי ומינימלי של רכיבי החומרה במחשב, והוא כולל גם ממשק input ו־output. ה־BIOS החל להיחשב ככלי "פרימיטיבי" כבר ב־2005-2006, ובתקופה זו יצרניות החומרה החלו למכור את מוצריהם עם UEFI במקום ה־BIOS שהפך ל־Legacy. מבחינתו, ההבדל העיקרי בין השניים הוא באופן בו "מעלים" את מערכת ההפעלה מהדיסק ל־RAM (כמובן שישנם הבדלים רבים נוספים, אך זה מה שמעניין אותנו בנקודת הזמן הנוכחית). בואו נבין איך בדיוק ה־BIOS טוען עבורנו את מערכת ההפעלה לדיסק, ואנחנו נטען את Windows ממש באותו האופן.

MBR

ה־MBR הוא למעשה הסקטור הראשון בדיסק שהוא BIOS bootable, וראשי התיבות הם קיצור של Master Boot Record. בכל דיסק שהוא bootable (כלומר, אפשר לבחור אותו בתפריט ה־boot של ה־BIOS), הסקטור הראשון מכיל 512 בתים המחולקים באופן הבא (שימו לב כי יש מספר גרסאות שונות ל־MBR, אנחנו נדון כאן ב־MBR הקלאסי):

- **446 הבתים הראשונים**: בעצם מדובר בקוד שירוץ כאשר נלחץ על כפתור ההדלקה של המחשב. ה־BIOS טוען את הקוד הזה לזיכרון ומפנה את האוגר ip לכתובת אליה הוא טען את בתים אלו.
- **16 * 4 בתים הבאים**: כאן נמצא המידע על (לכל היותר) 4 partition entries בדיסק הנוכחי. אני לא אפרט עליהם מכיוון שהם פחות רלוונטים לנו, אבל חשוב לדעת שהם חלק מה־MBR.
- **2 בתים אחרונים**: בשני הבתים האחרונים של ה־MBR יש signature שתפקידו לעזור ל־BIOS להבין כי הדיסק הנוכחי הוא bootable. ערך החתימה הוא 0xAA55.

כפי שהזכרתי תוך כדי ההסבר, תפקיד ה־BIOS הוא מאוד פשוט. כל שעליו לעשות הוא לטעון את ה־MBR מהדיסק שעליו יושבת מערכת ההפעלה, ולמקם אותו ב־RAM. בפועל, ה־BIOS יטען את ה־MBR תמיד לאותה הכתובת, והיא 0x7c00. אין משמעות אמיתית לבחירה בכתובת הזו, אלא זה נובע בעיקר מסיבות היסטוריות. מוזמנים לקרוא על זה כאן.

אם כך, על שעלינו לעשות הוא למצוא את הדיסק עליו מותקנת מערכת ההפעלה, לטעון משם את הסקטור הראשון, ולהעתיק אותו לכתובת (הפיזית) 0x7c00. נשמע פשוט מאוד; בואו נצא לדרך!

קריאת נתונים מהדיסק

סיכוי סביר בהחלט שכמעט כולנו מעולם לא קראנו באופן ישיר מהדיסק סקטורים. אמנם עבודה עם קבצים זה משהו נפוץ מאוד וגם הוא מערב קריאה של נתונים מהדיסק, אך סביר להניח שמעולם לא התבקשתם לטעון סקטור ספיציפי מהדיסק לזיכרון. לשמחתנו הרבה, ה־BIOS דואג לנו מהבחינה הזו. קיימת פסיקה אשר מקבלת מספר (אינדקס) דיסק, סקטור לקריאה, כתובת יעד בזיכרון והיא עושה עבורנו את הקסם של קריאה מהדיסק. כל שעלינו לעשות הוא להשתמש בפסיקה הזו.

int 0x12

הפסיקה הזו משמשת אותנו בדיוק עבור מה שתיארתי למעלה. שמה המלא הוא Extended Read Sectors From Drive. היא מקבלת כפרמטר DAP, שזה קיצור של Disk Address Packet. ה־DAP נראה כך:

Registers	Description
AH	42h = function number for extended read
DL	drive index (e.g. 1st HDD = 80h)
DS:SI	segment:offset pointer to the DAP, see below

DAP : Disk Address Packet

offset range	size	description
00h	1 byte	size of DAP (set this to 10h)
01h	1 byte	unused, should be zero
02h..03h	2 bytes	number of sectors to be read, (some Phoenix BIOSes are limited to a maximum of 127 sectors)
04h..07h	4 bytes	segment:offset pointer to the memory buffer to which sectors will be transferred (note that x86 is little-endian: if declaring the segment and offset separately, the offset must be declared before the segment)
08h..0Fh	8 bytes	absolute number of the start of the sectors to be read (1st sector of drive has number 0) using logical block addressing (note that the lower half comes before the upper half)[9]

[קרדיט: ויקיפדיה]

הבית הראשון ה־DAP מכיל את הגודל שלו, שהוא תמיד 16 בתים. הבית השני מיועד לשימוש ע״י ה־BIOS בלבד, וערכו צריך להיות מאותחל ל־0. שני הבתים הבאים מכילים את מספר הסקטורים לקריאה. מכיוון שאנחנו קוראים סקטור יחיד, ערך זה יהיה 1. ארבעת הבתים הבאים מכילים את כתובת היעד של הסקטור אותו נקרא מהדיסק. בערך זה נשים כתובת של buffer כלשהו, שיועתק לכתובת 0x7c00 מאוחר יותר. שמונה הבתים האחרונים מכילים את האינדקס בדיסק שממנו תתחיל קריאת הסקטורים. מכיוון שה־MBR נמצא בסקטור 0 של כל bootable device, ערך זה יהיה 0.

לאחר סיום אתחול ה־DAP, נבצע מעבר של המערכת ל־real mode, נכוון את DS:SI להצביע אל ה־DAP שלנו, נשים ב־AH את הערך 0x42 ונריץ int 0x12. האתחול נראה כך:

```
VOID ReadFirstSectorToRam(IN BYTE diskIndex, OUT BYTE_PTR* address)
{
    PDISK_ADDRESS_PACKET packet = DAP_ADDRESS;
    packet->size = 0x10;
    packet->reserved = 0;
    packet->count = 1;
    packet->offset = FIRST_SECTOR_DEST;
    packet->segment = 0;
    packet->sectorNumberLowPart = 0;
    packet->sectorNumberHighPart = 0;
    *(BYTE_PTR)(DAP_ADDRESS + 0x10) = diskIndex;
    EnterRealModeRunFunction(DISK_READER);
    *address = FIRST_SECTOR_DEST
}
```

כמו שאתם רואים, יצרתי מצביע ל־DAP בכתובת השמורה ב־DAP_ADDRESS. הערך שיש שם הוא 0x4000. למה? כובע! (ועכשיו ברצינות - אין משמעות מיוחדת לכתובת הזו, פשוט בחרתי כתובת נמוכה ע"מ שנוכל להצביע אליה ב־real mode). בכתובת שאחרי ה־DAP שמרתי את הערך של disk index כדי שנוכל להעביר אותו ל־DL לפני הקריאה. הקריאה תתבצע כך:

```
[BITS 16]
DiskReader:
    xor eax, eax
    mov si, DAP_ADDRESS
    mov ah, 0x42
    mov dl, byte [DAP_ADDRESS + 0x10]
    xor bx, bx
    int 13h
```

פשוט מאוד, לא ככה?

עכשיו כשנחזור חזרה לפונקציה ReadFirstSrctorToRam, הסקטור הראשון של הדיסק המשוייך ל־ disk index ימצא בכתובת FIRST_SECTOR_DEST. כל שעלינו לבדוק הוא האם מדובר בדיסק שהוא bootable, כלומר לבדוק האם הוא מכיל את ה־signature 0xAA55 בסופו. אם זה אכן המצב, נעתיק את התוצאה ל־0x7c00, נעבור ל־real mode, נבצע far jump לכתובת הזו - והופ, טענו את Windows בהצלחה!

```
VOID LoadMBRToEntryPoint()
{
    PMBR sectorAddress;
    for(BYTE diskIdx = 0x80; diskIdx < 0xff; diskIdx++)
    {
        ReadFirstSectorToRam(diskIdx, &sectorAddress);
        if(sectorAddress->bootCode == BOOTABLE_SIGNATURE)
        {
            Print("An MBR disk was found at disk #%d\n", diskIdx);
            CopyMemory(MBR_ADDRESS, sectorAddress, MBR_SIZE);
            *(BYTE_PTR)(WINDOWS_DISK_INDEX) = diskIdx;
            break;
        }
    }
    // 0x7c00 now contains the MBR
}
```

אנחנו מתחילים לסרוק את הדיסקים החל מאינדקס 0x80, שהוא האינדקס של הדיסק הראשון. בכל פעם אנחנו בודקים האם שני הבתים האחרונים מעידים על כך שהדיסק הזה הוא bootable או לא. אם כן, אנחנו מבצעים העתקה של הסקטור ל־MBR_ADDRESS (שהערך בו כמובן הוא 0x7c00) ונאחסן במקום זמני בזיכרון את האינדקס של הדיסק בו נמצא ה־MBR (וזאת כדי לאחסן את הערך ב־DL רגע לפני ביצוע far jump ל־0x7c00).

כך נראה הקוד שמבצע את הפינאלה:

```
HandleControlToWindows:
    mov dl, [WINDOWS_DISK_INDEX]
    mov ax, 0
    mov ds, ax
    mov es, ax
    mov ss, ax
    mov fs, ax
    mov gs, ax
    jmp 0:0x7c00
```

אנחנו מאחסנים ב־DL את הערך ששמרנו בכתובת המוצבעת על ידי WINDOWS_DISK_INDEX (שזה הוא האידקס של הדיסק בו נמצא ה־MBR של Windows), וזאת כחלק מהדרך בה ה־BIOS טוען MBR של דיסק כלשהו לזיכרון.

זה הכל! זה הוא רגע מאוד מרגש, שכן הצלחנו להעלות את מערכת ההפעלה שתרוץ מעל ה־hypervisor שלנו!

אחרית דבר

זהו, אפשר לומר שסיימנו כאן. מערכת ההפעלה עלתה (או יותר נכון הועלתה על ידנו), ה־hypervisor שלנו פעיל על כל הליבות, ולמעשה כאן מתחיל החלק הכיף הכולל טיפול ב־vm-exits ושימוש בכלים שהוירטואליזציה מציעה (משחקים עם EPT, ביצוע hook-ים מעניינים ומה לא). את זאת ועוד אני לא הולך לכסות במאמר, אלא כל שנותר לי הוא לאחל לכם בהצלחה, ולקוות שתיגשו אתם ותנסו לעשות זאת בכוחות עצמכם. מקווה שנהנתם לפחות כמוני, ובעיקר אני מקווה שהכלים שרכשנו יחד ישמשו אתכם ויהוו בסיס מוצק לעבודה עם מנגנון הוירטואליזציה.

(אגב, המימוש המלא של כל מה שהצגתי כאן נמצא ב־repository של הפרויקט ב־github, יחד עם דברים נוספים שלא עסקנו בהם)

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 - Creately:

https://creately.com/