VIRTUALIZATION

INTRODUCTION

וירטואליזציה הינה דימוי תשתיות חומרה אמיתיות באמצעות הקצאת משאבי חומרה מתוך תשתית חומרה רחבה יותר.
חשוב לשים לב שוירטואליזציה יכולה להתבצע ברמת האפליקציה (JVM) וברמת מערכת ההפעלה (VMware, Hypervisor),
אנו נתייחס לוירטואליזציה באזורי מערכת ההפעלה.

בסוף, וירטואליזציה מספקת שכבת אבסטרקציה בין החומרה לבין מערכת ההפעלה, ומאפשרת לעשות דברים די מגניבים:

- הרצת מכונות וירטואליות בבית לצורכי מחקר (VM).
- שרת ארגוני שמריץ המון מכונות על מנת לנצל את משאבי החומרה ולחסוך בעלות (ESX).
- יכולת ניהול, גמישות, ושחזור של משאבים באופן פשוט ויעיל.
- קבלת נוטיפיקציה על אירועי מערכת לבחירתנו (hypervisor).
- **אבטחה (VBS) !**

HYPERVISOR/VMM

תכנה לניהול מכונות וירטואליות, אשר יוצרת הדמיה של משאבי החומרה למכונות (מעבד, זיכרון, פסיקות I/O).

HYPERVISOR TYPES

נהוג לחלק את סוגי ה Hypervisors לשלוש משפחות עיקריות:

1. **bare-metal** רץ על החומרה האמיתית ("על הברזלים") ומריץ מעליו המון VMs - חוות שרתים (ESX, MS Hyper-V).
2. **hosted** רצים מעל מערכת ההפעלה כמו כל תכנה אחרת, אך מספק שירות שמסוגל להריץ VM במחשב שלנו (VMware Workstation, KVM).
3. **host-only** משתמשים ביכולות הוירטואליזציה על מנת להריץ את המערכת הקיימת כ VM, במטרה לקבל נוטיפיקציה על אירועים מאוד low-level-ים, למשל כמו גישה לאוגר רגיש \ אזור זיכרון.
יכול לשמש גם למטרות פחות נחמדות (BluePill)

IMPLEMENTATION

POPEK AND GOLDBERG VIRTUALIZATION REQUIREMENTS

- **אמינות** - ה guest חייב לרוץ בדיוק כפי שהיה רץ אילו היינו קונים מחשב שמריץ אותו ישירות על החומרה.
- **ביצועים** - ניתן להבין שהרצת ה Guest תהיה איטית יותר, אך חשוב לזכור שבסוף אם זה איטי - לא ישתמשו בזה.
- **אבטחה** - חשוב לדאוג שה Guest לא יוכל להשפיע על ה host או guest אחרים כרצונו.

IMPLEMENTATIONS METHODS

EMULATION

נכתוב תכנית אימולצייה אשר מקבלת אסמבלי של ארכיטקטורה כלשהי, עוברת פקודה פקודה מתרגמת אותה לאכיטקטורה מעליה היא רצה ומבצעת אותה (תוך כדי עיבוד של פקודות מותרות\אסורות וכו'). היתרונות הן שיצרנו כאן תכנית מדהימה שמספקת אבסטרקציה ובעצם נותנת להריץ כל ארכיטקטורה X מעל ארכיטקטורה Y (במידה ונתמך), החיסרון הגדול הוא שזה פשוט איטי באופן בלתי סביל.

TRAP-AND-EMULATE

לאחר מכן, הבינו שאי אפשר לעבד כל opcode לפני שמריצים אותו, ולכן הגישה המבוקשת היא "תן לקוד לרוץ - מה כבר יקרה ?!". הבעיה היא שיש פקודות ASM שלא היינו רוצים לתת ל guest להריץ (מבעיות אבטחה, ארגון):

- **Sensitive Instructions** - שליטה במשאבי החומרה.
- **Privileged Instructions** - פקודות שהיינו רוצים שרק ring 0 יוכל לבצע.

נניח שאנחנו מניחים שהפקודות הבעייתות לנו מבחינת וירטואליזציה הן אותן פקודות לעיל, ובנוסף בהנחה שהמעבד יודיע לי מספיק טוב שאני מריץ פקודה שלא הייתי אמור (general protection exception) - השיטה אומרת בואו ניתן לקוד פשוט לרוץ במצב unprivileged, ובמידה והרצנו פקודה בעייתית יקפוץ trap שה VMM יידע לטפל בו.
היתרונות הן שכל הפקודות הנורמטיביות ירוצו מעל החומרה באופן ישיר. החיסרון הוא שעדיין פקודות בעייתיות יהיו יותר איטיות מבאופן רגיל. כל זה טוב ויפה בארכטיקטורה דמיונית, אך כשמסתכלים פרקטית על x86 עולות בעיות:

1. **Non-Privileged Sensitive Instructions** - ישנן פקודות ב x86 שלא עומדות בהנחות העבודה שלנו:
   a. פקודות שקשורות ל interrupt flags שירוצו כ unprivileged, יתנהגו כ NOP במקום להעלות exception.
   b. ישנן פקודות unprivileged שמשתמשות בעקיפין במשאבי חומרה שה VMM היה רוצה לשנות.
   למשל LAR ניגש ל GDT, המעבד ייגש ל GDT של ה VMM במקום לשל ה guest OS.
   c. פקודות push %cs שמעתיקות את ה CPL, עלולות להוות בעיה כיוון שv guest OS יצפה לראות שם 0, אבל הוא בעצם יראה שם ring שהוא לא 0.
   d. פקודות קריאה בלבד של משאבי חומרה שבעצם יקראו את של ה VMM במקום את של ה guest OS.

Group	Instructions
Access to interrupt flag	`pushf, popf, iret`
Visibility into segment descriptors	`lar, verr, verw, lsl`
Segment manipulation instructions	`pop <seg>, push <seg>, mov <seg>`
Read-only access to privileged state	`sgdt, sldt, sidt, smsw`
Interrupt and gate instructions	`fcall, longjump, retfar, str, int <n>`

2. **Ring Compression** - כשקבענו שה guest OS ירוץ כ unprivileged, הכוונה היא לא ב 0 ring כלומר 1/2/3.
   x86 paging למשל לא מבדיל בין 0/1/2 ring, וגם ב x64 יש מגבלות, לכן ה guest OS חייב לרוץ ב 3 ring (UM).

3. **Address Space Compression** - בסופו של דבר יש מבני נתונים שנשמרים עבור המעברים בין ה guest OS לבין ה VMM ולכן מבוזבז זיכרון ל guest OS (יתרה מכך, יש מימושים שה VMM נמצא ב Address Space של ה VM). בנוסף לכך, חשוב לוודא שה guest לא יכול לגשת לאזורים האלה בזיכרון, שכן אחרת זה מעלה בעיות אבטחה.

עם זאת, בגלל חשיבות הוירטואליזציה x86 שופר - לא באופן ש trap-and-emulate יעבוד, אך בשיטות עקיפות.

FULL VIRTUALIZATION

גישה שבאה לפתור את הרצת הפקודות הבעייתיות בעזרת שיטת **Binary Translation (BT) + Direct Execution (DE)**.
BT מוצאת את הפקודות הבעייתיות באסמבלי (סטטית או דינמית) ומחליפה אותן בפקודות שקולות אך קצת שונות \ ב trap ל VMM אשר יאמלץ פקודה שקולה, את הפקודות הרגילות מריצה כמו שהן. סה"כ פותר את הבעיה, אך יש איטיות שעדיין קיימת בהעברת המושכות לאמלוץ של חלק מהפקודות הבעייתיות + האמלוץ עצמו.

איך BT באמת מחליף את הפקודות הבעייתיות ?

- **סטטית** - ממש מתרגם מראש בינארים להרצה לבינארים חדשים שניתן להריץ (עלול לפספס).
- **דינמית** - שמירת ה X פקודות הבאות לביצוע ב translation cache, והחלפתן בזמן ריצה לפני ביצוע.

PARAVIRTUALIZATION (PV)

שיטה שאומרת שהרבה בעיות שנוצרות רק בגלל שמ"ה לא מודעת שהיא רצה מעל VMM, לכן היא מציעה API של פקודות בעייתיות להרצה ל VMM. כמובן שהיתרון שלה הוא מהירות - הכל רץ כמו שהוא והקריאות API מהירות מאוד. עם זאת, איבדנו הרבה ממה שרצינו להשיג - guest OS יודעת שהיא רצה מעל ה VMM, והכי גרוע שצריך לשנות את ה source code של ה guest OS שזה לא מובן מאליו ולא תמיד אפשרי.

HARDWARE ASSISTED VIRTUALIZATION (HVM)

Intel ו AMD הבינו את הבעיה והצורך ובאזור 2006, הוציאו מעבדים AMD-V Intel-VT-x אשר פותרים את שורש הבעיה. המעבדים החדשים בעצם תומכים בסט פקודות חדש שרק ה VMM יכול להריץ (כמו ring -1), אשר תומכים חומרתית ביכולת של ה VMM לקבל את הריצה כאשר מורצת פקודה "בעייתית".

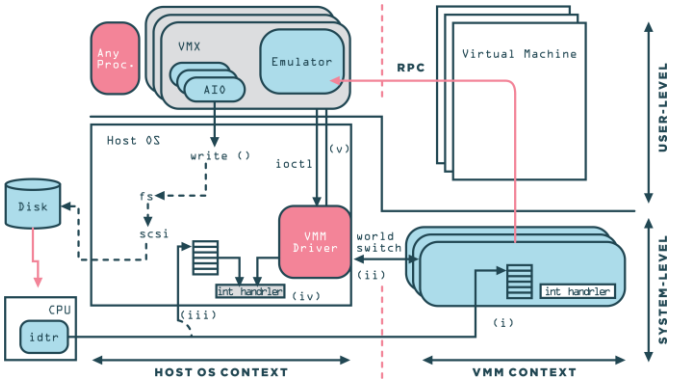
כיום, זו השיטה המקובלת לעשות וירטואליזציה, עם זאת משתמשים בשילובים עם טכניקות נוספות על מנת לעשות אופטימיזציות נוספות, אך טכנולוגיית intel-VT-x כלשעצמה מספקת כלים להתמודדות עם הבעיה.

VMWARE WORKSTATION (X86) BEFORE HVM

נציג בגדול את הפתרונות של VMWare להרצת VM מעל ה Host OS, ללא יכולות וירטואליזציה חומרתיות.

ARCHITECTURE

תהליך UM ראשי שמנהל את כל הויטואליזציה של ה VMs, תהליכי בן בשם vmx עבור כל VM שמריצים, וכמובן דרייבר קרנלי VMM שמטפל בחלקים ה Low-level-י של הוירטואליזציה.

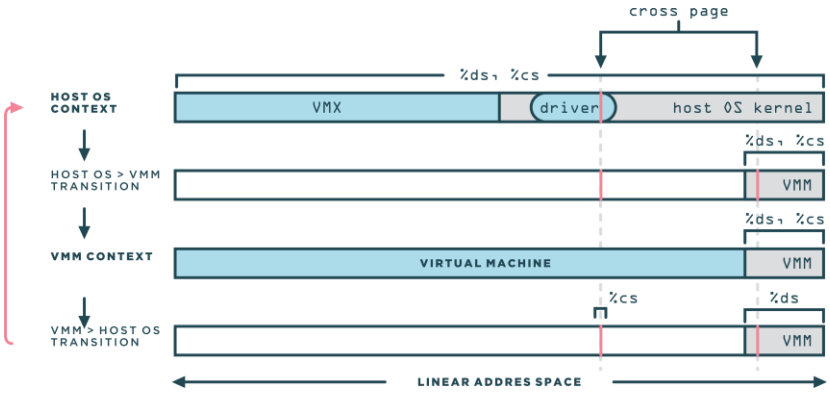


VMWare Workstation Architecture

HOST OS VS. VMM CONTEXT

בכל רגע נתון מ"ה יכולה להימצא ב VMM context \ host OS context \ world switch, כאשר לכל context יש את ה"עולם" שלו (Virtual Address Space, IDT, Stack, etc.).

חשוב לשים לב שמבחינת מ"ה ה VMM driver הוא מודול קרנלי רגיל, אך בפועל הוא מממש את כל הפרדת העולמות הזאת.



world-switch process

נקצה cross-page ונשמור בו את הקוד שאחראי ל world-switch ואת מבני הנתונים שיש לשמור בין ה contexts:

```
pushf
pop     eax
mov     [ebp-3Ch], eax  ; save eflags
cli                     ; clear interrupts
mov     eax, cr0
mov     [ebp-1Ch], eax  ; save cr0
mov     eax, cr2
mov     [ebp-18h], eax  ; save cr2
mov     eax, cr4
mov     [ebp-20h], eax  ; save cr4
mov     eax, dr0
mov     [ebp-38h], eax  ; save dr0
mov     eax, dr1
mov     [ebp-34h], eax  ; save dr1
mov     eax, dr2
mov     [ebp-30h], eax  ; save dr2
mov     eax, dr3
mov     [ebp-2Ch], eax  ; save dr3
mov     eax, dr6
mov     [ebp-28h], eax  ; save dr6
mov     eax, dr7
mov     [ebp-24h], eax  ; save dr7
sgdt    fword ptr [ebp-10h] ; save gdtr
sldt    word ptr [ebp-2] ; save ldtr
```

```
mov     eax, [ebp-18h]
mov     cr2, eax        ; restore cr2
mov     eax, [ebp-20h]
mov     cr4, eax        ; restore cr4
mov     eax, [ebp-38h]
mov     dr0, eax        ; restore dr0
mov     eax, [ebp-34h]
mov     dr1, eax        ; restore dr1
mov     eax, [ebp-30h]
mov     dr2, eax        ; restore dr2
mov     eax, [ebp-2Ch]
mov     dr3, eax        ; restore dr3
mov     eax, [ebp-28h]
mov     dr6, eax        ; restore dr6
mov     eax, [ebp-24h]
mov     dr7, eax        ; restore dr7
lldt    word ptr [ebp-2] ; restore ldtr
cmp     dword ptr [edi+270h], 65h
jnz     loc_10002CB0
mov     eax, [edi+274h]
lea     ecx, [eax-20h]  ; switch 224 cases
cmp     ecx, 0DFh
ja      loc_10002C91    ; jumptable 100026B5
jmp     ds:off_10002CC0[ecx*4] ; switch jump
```

cross-page context

לאחר מכן, כאשר נרצה לעבור מ context אחד לאחר, נקרא לקוד שב cross-page אשר שומר את כל הקונפיגורציה הנוכחית, נמפה את ה cross-page גם לזיכרון של ה context האחר, ונטען את ה context האחר כפי שהיה שמור ב cross-page.

EXTERNAL INETRRUPTS

כאשר device חיצוני יעלה Interrupt ב VMM context, רץ קוד interrupt handler של ה VMM שבעצם עושה world switch (שגורם להחלפת IDT) ולאחר מכן פשוט מריץ int <vector> בהתאמה שה host OS יטפל בו כפי שהוא יודע, כשהטיפול מסתיים שהריצה חוזרת ל VMM driver והוא בתורו מחזיר את הריצה ל UM VMX process המתאים על מנת לאפשר scheduling.

I/O REQUESTS

עבור Virtual I/O requests שה VMM מעוניין לבצע עבור VM ספציפי, ה VMM משתמש ב RPC ל vmx process המתאים אשר מנהל threads שאחראים על כל הבקשות. לאחר קבלת ה RPC, ה vmx בתורו יעשה syscall בהתאמה.

VMM

VIRTUALIZATION METHOD

נקטו בסוג של גישת Full Virtualization (שהצגנו לעיל), בגדול משתמשים בשיטת DE על הרצת קוד UM ישירות ב ring 3, ואילו בשיטת BT עבור קוד KM של ה guest שירוץ ב ring 1.

VMM PROTECTION

נקטו בשיטה בה ה VMM נמצא בזיכרון הקרנלי של ה VM אותו מריצים (ב 4MB העליונים), מה שמשפר ביצועים אך מעלה בעיית אבטחה שבה ה VM יכול לגשת לזיכרון של ה VMM כרצונו. עבור הרצה של user-mode ב guest מובטח לנו כבר שלא יוכלו לגשת לדפים קרנליים בעזרת ה User/Supervised flag. עבור kernel-mode בשילוב עם ה BT, נוצרה בעיה ששינינו קוד "בעייתי" לקוד שניגש לחלקים בזיכרון של ה VMM. לכן, שינו את ה segment descriptor בכך שהקטינו את גודלם ב 4MB כך שיצביעו עד התחלת ה VMM (לא כולל). בנוסף לכך, שינו את segment gs שיצביע על תחילת ה VMM והשתמשו ב BT על מנת להחליף כל פקודה בעייתית לגישה ל VMM בעזרת gs, וכל גישה אחרת עם gs הוחלפה ב fs.

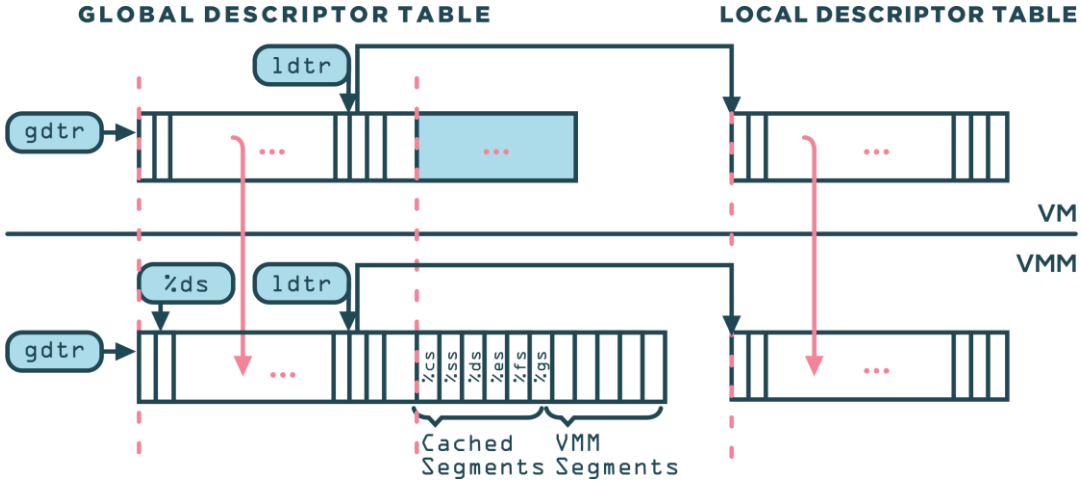
MEMORY VIRTUALIZATION

האתגר שעומד בפנינו הוא לגרום ל VM לחשוב שיש לו זיכרון משלו, ושפקודות כמו mov מזיכרון יעבדו אוטומטית.
הגישה שנקטו בה היא **shadow page tables**, בשיטה זו אנו מנהלים בעצם page table והעתק שלו ב VMM.
ה VMM לוקח את כל ה page directory ב cr3 ומשנה את כל אזורי הזיכרון שלו ל read-only. לאחר מכן, כאשר ה guest רוצה להקצות זיכרון לדוגמא או כל דבר אחר, זה מתבטא בכתיבה ל page tables, מה שיגרור interrupt שה VMM יקבל ויטפל בו. ה VMM כשלעצמו שומר "העתק" של ה guest page tables ומעדכן אותו בכל פעם שהוא מקבל Interrupt שהיה כתיבה לאזור של ה guest page tables, בנוסף לכך הוא מקצה זיכרון אמיתי אצלו ומעדכן אצלו ואצל ה guest את המיפוי החדש בין ה guest VA לבין ה Machine PA.

בסופו של דבר השיטה צורכת יותר זיכרון (shadow+), היא הופכת את העסק לאיטי יותר כיוון שהיא גורמת להמון page faults שמתבטאים בסוף ב VM-exit והרצת קוד של ה VMM + איפוס ה TLB בכל מעבר בין VMs. בין היתר בעיה זו היא אחת מהסיבות העיקריות למעבר בתמיכה חומרתית לפתרון הבעיה (SLAT\EPT).

SEGMENTS DESCRIPTORS VIRTUALIZATION

הגישה שנקטו בה היא **shadow segments descriptors**, כאשר מחלקים את הסגמנטים לשלוש קבוצות שונות: רק את אלה שה VMM משתמש בהם לשימוש פנימי (vmm), את אלה של ה vCPU – fs, cs, gs, ... (cached), כל השאר (shadow).



Segments Descriptors

דואגים לכך שנקבל נוטיפיקציה על השינוי של ה segment descriptors, ונגרום לדברים הבאים לקרות:

- הקטנת הגודל של ה shadow, cached שלא יוכלו לגשת ל 4MB האחרונים של ה VMM.
- שינוי ה DPL של ה shadow, cached עם DPL=0 ל 1, על מנת שנוכל להריץ קוד שנוצר מה BT.
- שינוי ה fs שיצביע על תחילת הקוד של ה VMM (כדי שרק לו יהיה גישה אליו).
- ה BT אחראי לדאוג שהפקודות לא יגרמו ל VMM segments ו fs להיטען, על מנת למנוע גישה ל VMM. למשל כל גישה ל fs תוחלף בגישה ל gs.

INTEL-VT-X

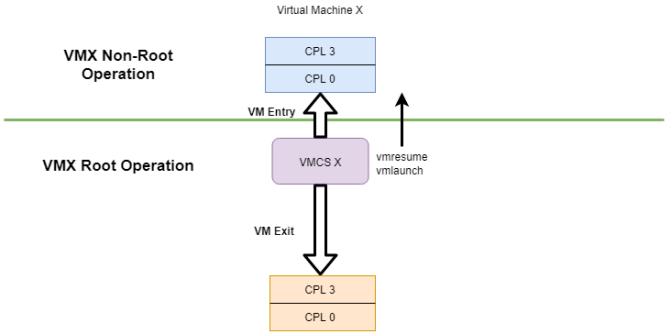
תמיכה חומרתית של אינטל בוירטואליזציה של המעבד Processor emulation, בין היתר היכולת להגדיר אירועים שיגרמו ל VMM לקבל את השליטה, לעצור את הריצה של ה guest לאחר זמן מוגדר, פתרונות לוירטואליזציה של זיכרון וכו'...
לאינטל יש פתרונות גם ל Network\PCI emulation שלא נכנס אליהם.

VM EXTENSION (VMX) OPCODES

פקודות חדשות שהמעבד תומך בהן שמקלות משמעותית את החיים בוירטואליזציה (ניתן להריץ את הפקודות רק כשרצים ב VMX operation, אחרת יקפוץ invalid opcode exception). כאשר רצים ב VMX operation ניתן לרוץ בשני מצבים:

- VMX root operation - כאשר ה VMM רץ, יכולות של ring 0 + פקודות וירטואליזציה privileged (כמו ring -1).
- VMX non-root operation - כאשר ה VM רץ, בעל יכולת להריץ פקודות וירטואליזציה non-privileged.

המעבר בין שני המצבים הללו נעשה באמצעות VMX transition, שמורכב מפעולות VM Exit ו VM Entry מאחורי הקלעים.



VMCS-MAINTENANCE INSTRUCTIONS

VMX OPCODE	PURPOSE
VMXON	Enter VMX root operation, using memory-operand
VMPTRLD	Load source operand to current VMCS
VMPTRST	Store current VMCS into destination operand
VMCLEAR	Clear the current VMCS
VMREAD	Read current VMCS component to destination operand
VMWRITE	Write current VMCS component from source operand
VMLAUNCH	Launch a VM managed by the current VMCS
VMRESUME	Resume a VM managed by the current VMCS
VMOFF	Processor leave VMX operation

GUEST INSTRUCTIONS

VMX OPCODE	PURPOSE
VMCALL	Call the VMM for service (VM-Exit)
VMFUNC	Invoke VM function in VMM (No VM-Exit)

VMX-SPECIFIC TLB MANAGEMENT INSTRUCTIONS

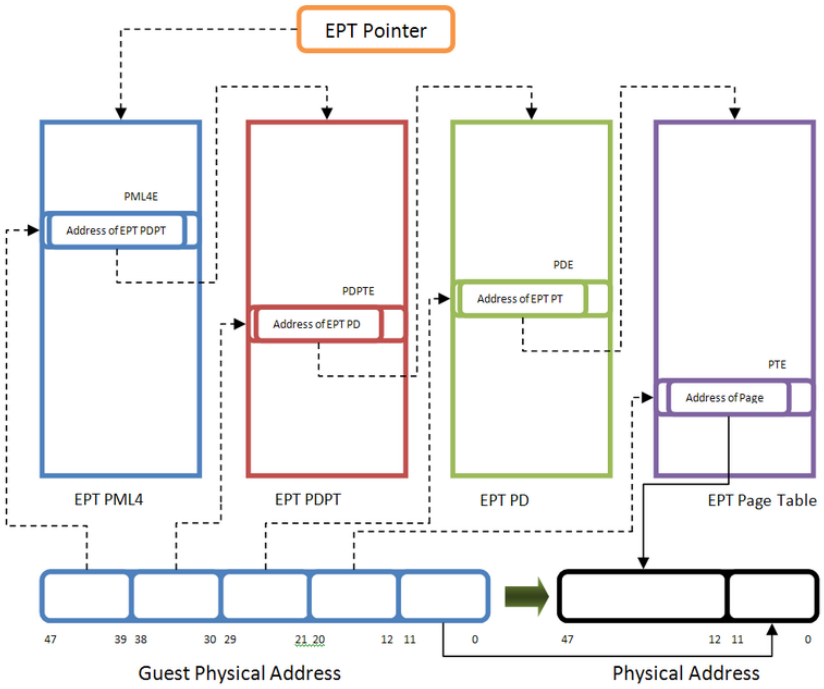
VMX OPCODE	PURPOSE
INVEPT	Invalidate cached EPT mappings in the processor
INVVPID	Invalidate cached based on the Virtual Processor ID

MEMORY VIRTUALIZATION

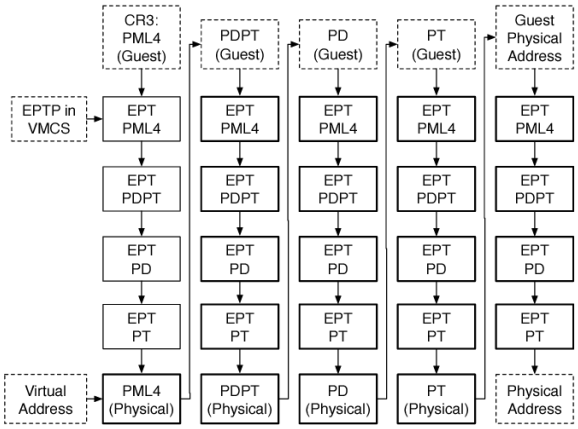
EXTENDED PAGE TABLES (EPT\SLAT)

כפי שראינו בעבר, התמודדנו עם בעיית ה memory virtualization בעזרת shadow page tables שמסבכים את החיים וגורמים ל overhead די גדול, לכן הוסיפו את התמיכה החומרתית על מנת להתמודד עם הבעיה באופן היעיל והנוח ביותר Second Level Address Translation.

בסופו של דבר כל מה שעשו זה להוסיף תמיכה חומרתית בשכבה נוספת של paging. כלומר ה guest OS ינהל את ה paging שלו כפי שהוא יודע וימפה gVA->gPA בעזרת cr3, ובנוסף לכך ה VMM ינהל paging שימפה gPA->mPA בעזרת אתחול האוגר EPTP. כאשר המעבד במצב non-root operation הוא יבצע תרגום בעזרת cr3 ולאחר מכן תרגום נוסף ע"י EPTP.



בעזרת התמיכה החומרתית נפתרה הבעיה שאנחנו לא עושים VM Exit בכל שינוי של ה paging ולא מתחזקים shadow page table, אך כעת במידה וה VA לא נמצאה ב TLB שלב תרגום הכתובות ארוך פי כמה וכמה (תרגום של CR3, ובנוסף תרגום **תכנתי** של EPTP שכולל המון תרגומי VA->PA בעצמו), ובנוסף אנחנו עדיין עושים flush ל TLB כאשר אנו מחליפים VM.



כתוצאה מכך, הרחיבו את ה TLB עם עוד רשומה שנקראת Virtual Processor ID (VPID), כאשר לכל vCPU של VM יש ID מיוחד לו (VPID). כעת, כאשר נרצה לתרגם VA בעזרת EPT ראשית נבדוק האם TLB[VPID, VA] קיים ובמידה ולא נבצע את תהליך ה paging. בכך, כאשר אנו ב VMX transition אין סיבה לעשות flush ל TLB והתקורה נחסכת מאיתנו מה שמשפר את הביצועים משמעותית. (הפקודה INVEPT מאפסת את כל ה TLB, בעוד ש INVVPID מאפסת רשומות של ה vCPU הרלוונטי).

MEMORY TYPE RANGE REGISTER (MTRR)

כידוע, לא כל ה RAM במלואו משמש כזיכרון פיזי קלאסי, אלא ה BIOS מקצה חלקים נבחרים ממנו לטובת MMIO.

```
    7) UC [0000000000000000 - 0000000000000000] (       0 kb)
Physical memory ranges (8)
    0)    [0000000000001000 - 00000000000A0000] (     636 kb)
    1)    [0000000000100000 - 0000000000353000] (    2380 kb)
    2)    [00000000003DA000 - 000000000BAF4000] (  187496 kb)
    3)    [000000000BB12000 - 000000000CFA9000] (   21084 kb)
    4)    [000000000CFB2000 - 000000000CFC5000] (      76 kb)
    5)    [000000000CFCB000 - 000000000CFDB000] (      64 kb)
    6)    [000000000CFE0000 - 000000000DEF9000] (   15460 kb)
    7)    [000000000DF89000 - 0000000080000000] ( 1868252 kb)
```

השאלה שנשאלת היא איך המעבד מודע ל RAM Regions, ומה המאפיינים של כל אזור זיכרון (start, end, cache policy) ? התשובה היא, כמובן, MTRR ! אוסף אוגרים ייעודים שמטרתם להגדיר ע"י ה OS & BIOS את האזורים הללו. (WinDbg !mtrr)

Table 11-8. Memory Types That Can Be Encoded in MTRRs

Memory Type and Mnemonic	Encoding in MTRR
Uncacheable (UC)	00H
Write Combining (WC)	01H
Reserved*	02H
Reserved*	03H
Write-through (WT)	04H
Write-protected (WP)	05H
Writeback (WB)	06H
Reserved*	7H through FFH

cache policy types

על מנת לגשת לאוגרים האלה יש לבדוק שהם supported & enabled ע"י CPUID & IA32_MTRR_DEF_TYPE MSR.
בנוסף לכך, ניתן לדרוס את ה cache policy בעזרת Page Attributes Tables (PAT) עבור pages אבל לא נכנס לזה פה.

למה זה מעניין אותנו ? כיוון שבעזרת ה EPT ניתן למפות gPA->hPA, ועל מנת ליצור ולהקצות מיפוי שכזה יש להבין מהן הכתובות הפיזיות הקיימות שממופות ומה ה cache policy שלהן. נניח עבור hypervisor type 3, נרצה לעשות Identity mapping, כלומר לגרום לכך שכל gPA פשוט ימופה לעצמו או במילים אחרות gPA=hPA.

VMXON REGION

אזור זיכרון שמפתח ה VMM אחראי לאלקץ עבור כל מעבד אמיתי ולהעביר את הכתובת הפיזית לפקודה VMXON כפרמטר. האזור זיכרון מיועד לניהול ה VMX operations בין ה VMXON ל VMXOFF. רק המעבד רשאי לגשת לאזור זה ואין לגשת אליו (גישה אליו תוביל להתנהגות לא מוגדרת). עקרונית, אין לנו כל כך סיבה להתעסק איתו יותר מדי מה גם שלטענתם הוא משתנה מאוד בין גרסה לגרסה של VT-x (המבנה הכללי שלו דומה ל VMCS, ה data פחות).

Format of the VMXON Region

Byte Offset	Contents
0	Bits 31:0 VMCS revision identifier
4	VMXON data (implementation-specific format)

VM CONTROL STRUCTURE (VMCS)

מבנה נתונים (4KB aligned) שנשמר בזיכרון עבור כל VM (ליתר דיוק עבור כל virtual CPU) ומנוהל ע"י ה VMM. נוצר על מנת לשמר מצב בין החלופת VM-ים, כאשר יש transition ה VMCS הנוכחי מכיל את כל המידע הדרוש על מנת לבצע אותו - של ה VM, של ה VMM וכו' (בכל רגע נתון המעבד עובד עם ה VMCS הנוכחי והיחיד, ניתן לשנות אותו ע"י הפקודות VMCLEAR או VMPTRLD אך יש לשים לב שהמעבד מצפה לקבל את הכתובת הפיזית של המבנה).
זה מבנה הקסם של כל הוירטואליזציה שלנו ! פה נוכל לאתחל ולקנפג את כל הפיצ'רים המגניבים של Intel ל VM שלנו.

VMCS PARTS

ה VMCS מחולק לשישה חלקים עיקריים:

- **Guest State Area** – מצב המעבד של ה guest, נשמר ב VM Exit ונטען ב VM Entry.
  - Guest registers, EPT control, segment descriptors, rflags, etc.
- **Host State Area** – מצב המעבד של ה host, נשמר ב VM Entry, ונטען ב VM Exit.
  - Registers, segment descriptors, MSRs, etc.
- **VM Execution Control Fields** – שדות שליטה על מעבד ה VM - קובע מה יגרור VM Exit.
  - MSRs, Exception bitmasks, registers, RPTP, VPID, APIC, etc.
- **VM Exit Control Fields** – שדות שליטה על ה VM Exit.
  - Debug controls, MSRs, etc.
- **VM Entry Control Fields** – שדות שליטה על ה VM Entry.
  - pretty same as VM exit control fields + event injection
- **VM Exit Information Fields** – שדות מידע לקריאה בלבד על מנת לקבל מידע על סיבת ה VM Exit.
  - Exit reason, error codes, opcode execution exits, etc.

MODEL SPECIFIC REGISTERS (MSRS)

אוגרים אשר משמשים כקונפיגורציה של מ"ה אל מול החומרה (בדומה ל control registers). ניתן לחשוב על זה כמערך של אוגרים פנימיים (פשוט בלי שמות מוחצנים) אשר ניתן לגשת אליו לאינדקס מסוים בעזרת הפקודות rdmsr, wrmsr מ ring 0.
בנוסף, תחת תנאים מסוימים ניתן לגשת ל MSRs שהם non-privileged גם מ ring 3 למשל בעזרת rdtsc.
בין היתר הם מעניינים אותנו כיוון שיש MSRs שלמשל מסמנים על פיצ'רים דלוקים, ובנוסף MSRs ספציפיים של VMX.
לדוגמה, הפקודה VMXON מושפעת משלושת הביטים הראשונים של IA32_FEATURE_CONTROL MSR.
נהוג לסמן כל אוגר כזה בתור האינדקס שלו ועם שם מוצמד שמתחיל ב IA32_*.

VMCS BITS

פירוט של כל ביט במבנה. ניתן לשנות\לקרוא את המבנה כאשר הוא טעון בעזרתה VMWRITE, VMREAD.

Virtual Machine Control Structure

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	

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.

COOL FEATURES

על ידי שליטה של ה VMM בשדות ה VMCS, intel מספקים לנו יכולות רבות בתור hypervisor שמריץ את ה VM על המעבד.

- **EPTP -** כפי שכבר ראינו, ניתן להשתמש (לא חייב) בשכבה נוספת של תרגום כתובות על מנת להתמודד מיפוי הכתובות הפיזיות של ה guest לכתובות הפיזיות של ה host. על הדרך מספק המון יכולות אבטחה ודברים מגניבים שניתן לעשות בעזרת משחק עם ה page table כפי שנראה בתרגילים.
- **VMCALL -** יכולת של ה VM ליזום VM-Exit עם פרמטרים כלשהם, בעצם בקשת service מה VMM ב root mode.
- **CR Access** – יכולת של קבלת נוטיפיקציה דרך VM-Exit כאשר ניגשים ל control registers, יכול מאוד לעזור גם בהקשרי אבטחה למשל אם מישהו משחק עם CR4.
- **MSR bitmap -** היכולת להגדיר עבור אילו אוגרי MSR יגרר VM-Exit כאשר ניגשים אליהם ב VM לקריאה או כתיבה.
- **Event Injection -** היכולת של ה hypervisor להזריק interrupts + exceptions ל VM למשל int 3.
- **Exception Bitmap -** היכולת להגדיר עבור אילו vectored events (just exceptions) יקפוץ VM-Exit.
- **MTF -** Monitor Trap Flag, היכולת של ה hypervisor להריץ opcode יחיד ב VM, ולאחר מכן לקבל את השליטה חזרה אליו בעזרת VM-Exit כמובן.
- **וכו' וכו' -** את השאר תקראו ב manual !

התחלה טובה תהיה להסתכל על Definitions Of Primary\Secondary Processor-Based VM-Execution Controls.

VIRTUALIZATION BASED SECURITY (VBS)

כפי שראינו בפרק הוירטואליזציה, intel הוסיפו תמיכה חומרתית בוירטואליזציה שבין היתר מספקת תמיכה חומרתית בהמון פיצ'רים מאוד מגניבים כמו SLAT/opcodes notification/registers notification/memory isolation. כאשר מוסיפים תמיכות חומרתיות זה פותח דלתות לכוח שלא היה בעבר - Microsoft לקחו את הלימונים (VT-x) ועשו מהם לימונדה (עוד אבטחה !).

אבטחה מבוססת וירטואליזציה משתמשת ב Hyper-V של windows על מנת ליצור אזורים מבודדים של זיכרון ממערכת ההפעלה הסטנדרטית. בכך, גם אם נמצאה חולשה במ"ה הפתרונות האבטחתיים החדשים מקשים משמעותית על השמשתן: הרצת קוד malware בקרנל, גישה למשאבים רגישים (user credentials) וכו'

MS HYPER-V ARCHITECTURE

Hyper-V הוא hypervisor type-1, כלומר שאנו מתקינים אותו אנחנו לא באמת מתקינים אותו ב Host OS, אלה הוא רץ מעל החומרה וה host OS מעליו (ועוד כמה מ"ה). כמובן שה Hypervisor אחראי לשליטה ול scheduling בין המכונות השונות.

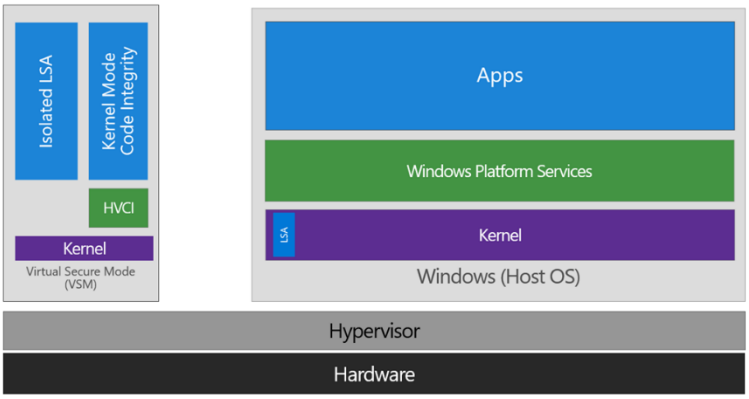
VIRTUAL SECURE MODE (VSM)

פיצ'ר אבטחתי המשתמש ב Hyper-V + SLAT על מנת ליצור אוסף Virtual Trust Levels - VTLs מבודדים אחד מהשני, כלומר כאשר רצים ב VTL כלשהו לא יהיה ניתן לגשת ל VTL אחר (אלא אם כן Hyper-V מאפשר זאת בדרך כלשהי) ללא תלות ב ring הנוכחי. בדרך זו ניתן יהיה לבודד את התהליכים הקריטיים והזיכרון שלהם במטרה להגן מתוכנות זדוניות קרנליות.
הקבלה נחמדה תהיה לחשוב על ציר ה CPL של rings כציר מאונך לציר ה ה VTLs (לא תלויים אחד בשני).

כפי שנראה בהמשך, כל VTL הוא בעצם מכונה נוספת מעל Hyper-V, כאשר יש בידוד בגישות לזיכרון בין VTLs.
כעת נממש את כל הפעולות הרגישות במכונה המבודדת שלנו:

- Local Security Authority (LSA)
- Kernel Mode Code Integrity (KMCI)
- Hypervisor Code Integrity (HVCI)

השאלה שנשאלת היא - אם אי אפשר לגשת מ VTL אחד לשני, איך נוכל לתמוך בפונקציונליות הנדרשת ?
התשובה היא שמ"ה יממשו מנגנון תקשורתי לשאילתות הכרחיות בהסכמת ה Hyper-V, בכך בעצם נקטין משמעותית את הסיכוי שקוד זדוני בקרנל יוכל לגנוב מידע רגיש כמו password hash למשל (מהסיבה הפשוטה שהוא מאוחסן ב VTL הרגיש).



DEVICE GUARD

אוסף מנגנונים שמטרתן למנוע הרצת קוד של malwares על ידי יכולת לאפשר רק לקוד שמוכר כלגיטימי לרוץ.

CONFIGURABLE CODE INTEGRITY (CCI)

מבטיח שרק קוד מהימן ירוץ מה boot loader והלאה, למשתמש יש את היכולת להגדיר Policy להרצת אפליקציות מורשות בלבד. המנגנון שאחראי על הבדיקות בקרנל הוא KMCI, ומהצד ה UM הוא UMCI.

VSM PROTECTED CODE INTEGRITY

ממקסם את ההגנה על מנגנוני ה KMCI וה HVCI מ host OS ring 0 בכך שמעביר את המימוש שלהם ל VTL0.

PLATFORM AND UEFI SECURE BOOT

מבטיח שכל הבינארים הראשונים שעולים לפני המנגנונים הקודמים (boot loader + UEFI firmware) חתומים ולגיטימים.
שימו לב שיש דרישה ל UEFI firmware ולתמיכה ב Secure Boot option.

CREDENTIAL GUARD

כידוע, lsass הינו התהליך אשר מנהל את כל אכיפת מדיניות האבטחה בווינדוס, בין היתר אחראי על אימות משתמש, שינוי סיסמא, תיעוד אירועי אבטחה וכו'. בתכלס אחראי על כל עניין ה credentials ולכן הוא תהליך מעניין בעיני תוקפים.
כחלק מ VSM מטרת Credential Guard היא לבודד ולהגן על מידע רגיש של המשתמש והמערכת בהנחה שקוד זדוני כבר רץ במערכת, ובכך ממזער את הרלוונטיות של התקפות כמו Pass The Hash, Mimikatz על סיסמאות מפתחות ועוד...

על מנת לפתור את הבעיה הוחלט לבודד את lsass ולהעביר אותו ל VTL0 על מנת להגן מגישה לזיכרון רגיש גם ב ring 0.
אך על מנת לשמור על legacy & backwards compatibility (איך לא ?), הוחלט להשאיר את lsass ב host OS כך שקריאות ל LSA עדיין יעבדו ומאחורי הקלעים פשוט ישמשו כ proxy לתקשורת עם VTL0.
מהצד של VTL0 ממומש trustlet ייעודי lsaiso - isolated LSA, שתפקידו לשמור ולממש את הלוגיקה האמיתית של lsass.