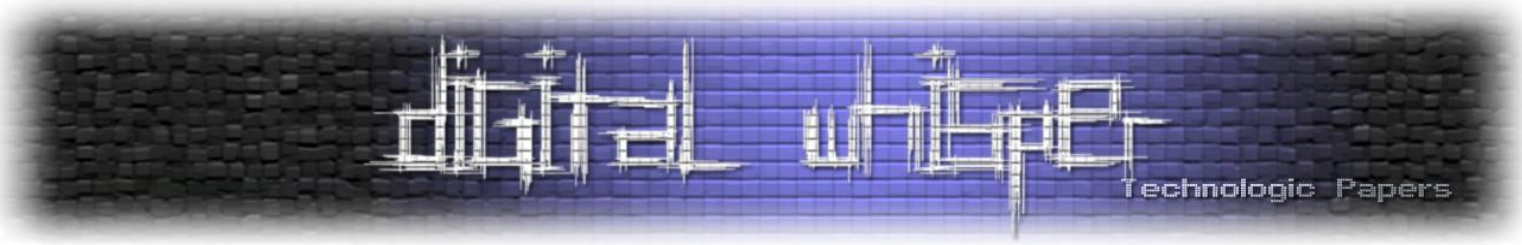


אבטחה מבוססת וירטואליזציה – Virtualization Based Security

מאת ליאל חנוכוב

הקדמה

רבים אינם מכירים את ארכיטקטורת מערכת ההפעלה Windows המודרנית ומניחים שעדיין מדובר בקרנל המונוליטי ntoskrnl.exe לניהול מערכת ההפעלה. מאז Windows 10, מייקרוסופט שינתה באופן מהפכני את מודל האבטחה של Windows באופן שמשפיע על רמת האמון והתממשקות שיש בין הקרנל ntoskrnl.exe לחומרה. במרכז המודל החדש עומד הקונספט "אבטחה מבוססת וירטואליזציה", טכנולוגיה שמשתמשת ב-Hypervisor Hyper-V כדי לחלק את המערכת לסביבות מבודדות שנקראות Virtual Trust Levels (VTLs), כשלכל אחת זיכרון והרשאות משלה. מה שבעבר חשבנו שרץ ב-Ring 0 (שכבת ההרשאות הכי נמוכה של המעבד) רץ ב-VTL 0 כשמעליו ב-VTL 1 קיים קרנל נפרד שנקרא Secure Kernel עם הרשאות גבוהות בהרבה שקרנל ה-NT (להלן NT) לא יכול לגשת אליו ככל שירצה. בין שני העולמות מתווך ה-Hypervisor שמהווה את הישות הכל-יכולה במערכת, הוא יכול לגשת לכל כתובת בזיכרון ולהשפיע על כל רכיב, והוא זה שאוכף את ההפרדה בין ה-VTLs באמצעות Second-Level Address Translation (SLAT). ההשפעה של זה במציאות היא שכל קרנל NT הוא אורח במערכת ההפעלה שמדבר ל-Hypervisor שמחליט מה הוא מורשה ולא מורשה לראות.

במאמר זה אציג את המבנה והיישום של אבטחה מבוססת וירטואליזציה, אראה איך המעבר מקרנל ה-NT לקרנל הבטוח (Secure Kernel) ממומש, איך יישומים ממומשים ב-Isolated User Mode (IUM) ולאורך הדרך אציג כלי שכתבתי כדי לאפשר לנו לדבג יישומים אלה.

קריאה מהנה,

ליאל

הגדרת הסביבה

כדי לעקוב אחרי הפעולות שאראה ושיהיו לכם את הגרסאות הנכונות של התוכנות, נשתמש במערכת ההפעלה של האורח שירוץ על גבי Hyper-V (תוכלו להשתמש גם ב-VMWare):

```
Microsoft Windows 11 Version 24H2 (OS Build 26100.7623)
```

חשוב יהיה להפעיל את התוכנה Memory Integrity כדי שאבטחה מבוססת וירטואליזציה (VBS) תהיה מופעלת ולבדוק האם המעבד שלכם תומך בוירטואליזציה. אם אתם משתמשים ב-Hyper-V כדי לנהל את המכונות הוירטואליות שלכם, תוודאו שאתם מאפשרים nested virtualization עם הפקודה:

```
Set-VMProcessor -VMName <VMName> -ExposeVirtualizationExtensions $true
```

ומאפשרים את ה-Guest Integration Services כדי שההתממשקות שלכם עם המכונה הוירטואלית תהיה חלקה ותאפשר שיתוף קבצים והעתקה-הדבקה. כדי לדבג את הקרנל באורח אפתח את ה-Command Prompt בתור מנהל ונבצע את הפקודות הבאות:

```
bcdedit /debug on
bcdedit /dbgsettings net hostip:172.23.128.1 port:50020 key:1.2.3.4
```

ובשביל לדבג את ה-Hypervisor:

```
bcdedit /Hypervisordebug on
bcdedit /Hypervisorsettings net hostip:172.23.128.1 port:50030 key:5.6.7.8
Set-VMProcessor -VMName <VMName> -ExposeVirtualizationExtensions $true
```

שימו לב לעדכן את כתובת ה-IP בהתאם לסביבה שלכם. ב-Hyper-V בכל פעם שמכבים או מחליפים את ה-Network Adapter, כתובת ה-IP של מכונת האורחת משתנה יחד עם זו של המארח עבור אותו vSwitch וזה כולל גם כיבוי של המחשב המארח.

נעשה כמה התאמות בהמשך בשביל לדבג את הקרנל הבטוח (Secure Kernel), אך בינתיים זו הקונפיגורציה המומלצת והמינימליסטית בשביל לדבג את הקרנל וה-Hypervisor. אשתמש ב-WinDbg Preview כדי לדבג את שניהם.

מהו Hypervisor?

ה-Hypervisor מאפשר להפעיל כמה מערכות הפעלה על אותו מחשב פיזי, כך שכל אחת מהן חושבת שהיא פועלת על חומרה משלה. הוא מנהל את המשאבים של המחשב כמו זיכרון, מעבד ואחסון, ומחלק אותם בין המכונות הוירטואליות בצורה מסודרת. בזכות זה, אפשר להריץ סביבות עבודה שונות במקביל, לבדוק תוכנות בלי לסכן את המארח, ולנצל טוב יותר את כוח המחשוב. יש שני סוגים של Hypervisor: סוג 1 שפועל ישירות על החומרה וסוג 2, שפועל מעל מערכת הפעלה מסוימת, לכל אחד מהם יתרונות משלו. יש לנו מספר דרכים לתקשר עם ה-Hypervisor ולהעביר דרכו או אליו מידע:

- Hypercalls – הדרך הישירה והמוכרת ביותר. האורח קורא לפונקציות שה-Hypervisor מספר, בדומה ל-syscalls, רק שהפעם הקריאה היא מאורח ל-Hypervisor שמתחתיה.
- Hyper-V – Synthetic MSRs מגדיר משפחה שלמה של אוגרי MSR שמשמשים כממשק שליטה ובקרה. לדוגמא: HV_X64_MSR_GUEST_OS_ID שמשמש לזיהוי מערכת ההפעלה. בעזרת Synthetic MSRs האורח יכול להגדיר את ה-Synthetic Interrupt Controller (SynIC) ש-Hyper-V חושף לאורחים מוארים (enlightened guests), מה שעוזר לנו ליצור התקנים וירטואליים שיאפשרו לנו לתקשר בין ה-Hypervisor לאורח ללא הפעולות היקרות של מעבר הקשר יקר אל ה-Hypervisor, או לבצע אמולציה להתקן חומרתי, בהמשך אסביר על SynIC.
- VM exit – אירוע שמתאר קריאה שתגרום לנו לצאת מהמכונה הוירטואלית (Virtual Machine exit). גם בלי לקרוא ל-hypercall מפורש, פעולות מסוימות של האורח יכולות לגרום ל-VM exit, מ-EPT violations ועד כתיבה ל-Control Registers. נוכל לגרום ל-VM exit עם כל פעולות ה-VMX שמבוצעות על ידי האורח כגון VMCALL, VMLAUNCH, VMRESUME ששמותיהן נותנים לנו רמז על מה שהן עושות.

לא ארחיב כאן על Hypervisors לעומק, אך למי שמעוניין להתעמק אמליץ על המדריך של סטושי טנדה לכתיבת Hypervisor בשפת ראסט, ועל התוסף ל-WinDbg hvext.

Synthetic Interrupt Controller (SynIC)

בחומרה אמיתית, למעבד אין מידע על אירועים כמו מתי לחצנו על כפתור, קיבלנו פקטה בעזרת האינטרנט או שעון שסיים לספור, לכן, הציגו את ה-interrupt controller, שיקבל interrupts מהרכיבים השונים ולפי כך יעביר אותם למעבד, שבהתאם יבצע את העבודה המתאימה.

Hypervisor שיריץ אורח חייב להפוך את כל זה לוירטואלי, למעבדים מודרניים יש תכונות כמו APIC Virtualization, posted interrupts וכדומה, שמורידים מעלות הטיפול ב-interrupts ביחד עם Local Advanced Programmable Interrupt Controller (Local APIC), שאחראי לקבל את ה-interrupts ולטפל בהם, אך בכל זאת זה עדיין ממשק חומרתי שנועד לתקשר עם רכיבים פיזיים.

לכן ב-Hyper-V הוגדר מבנה חדש שנועד לשנות את הדרך שבה אורח ו-Hypervisor מתקשרים והיא נקראת the Synthetic Interrupt Controller (SynIC). SynIC הוא ממשק פרה־וירטואלי לניהול הודעות ואירועים אסינכרוניים שעובד ביחד עם LAPIC, לא מחליף אותו.

- החלק הבא מאוד מסובך והיה לי מורכב להסביר אותו בעברית ועוד בפחות מ-5 עמודים, אז מוזמנים לדלג עליו ומי שרוצה יכול לראות את ההתנהגות מלמעלה של הדברים.

כל מעבד וירטואלי (Virtual Processor) שמשתמש ב-SynIC יקבל:

- 16 Synthetic Interrupt Sources (SINT0-SINT15) – ערוצים לוגיים שיעבירו מידע על interrupts, לדוגמא VMBUS בדרך כלל משתמש ב-SINT2.
- The Synthetic Interrupt Message Page (SIMP) – עמוד של 4KB שימופה לזיכרון הפיזי של האורח ויחולק ל-16 עמודות של 256 בתים – עמודה לכל SINT.
- The Synthetic Interrupt Event Flag Page (SIEFP) – עוד עמוד של 4KB שגם כן יחולק ל-16 עמודות של 256 בתים כשניתן לכל SINT עמודה, עמוד זה ישומש להעברת הדגלים על ה-SINT הספציפי, כל ביט הוא דגל שמשמעותו לדוגמא "אירוע X קרה ל-Y SINT" כש-X יהיה מיוצג על ידי ביט ה-index בעמוד הנוכחי (2047 ביטים בחלוקה זו), ו-Y יהיה מספר ה-SINT.
- 4 טיימרים סינטטיים (Synthetic timers) – מיוצגים על ידי STIMER0-STIMER3, לאחר תפוגה של טיימר נעביר את ההודעה HVMSG_TIMER_EXPIRED ל-SIMP.
- Configuration MSRs – מה שיכתוב את ה-traps ל-Hypervisor ומהם נקרא ונכתוב את המצב הנוכחי.

השלבים להפעלת SynIC באורח ייראו להלן:

1. האורח יקרא את CPUID ב-leaf 0x40000003 כדי לאשר שאכן SynIC זמין.
2. נאלקץ עמוד של 4KB בזיכרון האורח ונכתוב את הכתובת שלו ל-HV_X64_MSR_SIMP.
3. נעשה אותו דבר כמו שלב 2 בשביל HV_X64_MSR_SIEFP
4. עבור כל SINT שהאורח מתכנן להשתמש בו, נכתוב HV_X64_MSR_SINTx עם ה-vector, mask, ו-auto-EOI כפי שנרצה.
5. נגדיר את ה-Interrupt Descriptor Table ואת ה-Interrupt Service Routine כדי שנדע מה לעשות עם interrupts שנקבל.
6. נגדיר את ביט 0 של HV_X64_MSR_SCONTROL כדי לאפשר SynIC במעבד הוירטואלי הנוכחי.

הפונקציה ב-securekernel.exe שאחראית על אתחול של SynIC בקרנל הבטוח היא ShvlpInitializeSynic:

```
__int64 __fastcall ShvlpInitializeSynic(__int64 Context)
{
  unsigned __int64 HV_X64_MSR_SIMP; // rbx
  char v3; // di

  // Configure SINT0 (HV_X64_MSR_SINT0 = 0x40000090).
  //   vector  = 0xF0
  //   Masked  = 0  (unmasked)
  //   AutoEoi = 1  (bit 17)
  // This is the vector the synthetic timer below will raise.
  __writemsr(0x40000090u, 0x200F0u);
  // Arm a synthetic timer that delivers through SINT0.
  ShvlpEnableSyntheticTimer(1073741968, 0);
  // Read current SIMP (HV_X64_MSR_SIMP = 0x40000083).
  // The hypervisor pre-populates this with a default message-page GPFN before the SK initializes.
  // The low 12 bits hold the control flags (bit 0 = Enable).
  HV_X64_MSR_SIMP = __readmsr(0x40000083u);
  if ( (ShvlpFlags & 1) != 0 )
  {
    v3 = 0;
  }
  else
  {
    // Bit 0 clear -> override with the SK's own message page
    v3 = 1;
    HV_X64_MSR_SIMP = (ShvlpSynicMessagePfn << 12) | HV_X64_MSR_SIMP & 0xFFF;
  }
  // Map the chosen GPFN into the boot address space at [a1+0x18]
  SkmmMapBootPage(*(Context + 24), HV_X64_MSR_SIMP >> 12, 2);
  // Extract PFN from PTE, claim the backing physical page.
  if ( !v3 && !SkmmClaimMappedPage(*(Context + 24), 0) )
    return 0xC0000043LL;                    // STATUS_SHARING_VIOLATION
  // Commit SIMP with Enable (bit 0) set, activating message delivery.
  __writemsr(0x40000083u, HV_X64_MSR_SIMP | 1); // Enable the SIMP, preserving the hypervisor-assigned message page.
  return 0;
}
```

לאחר העמוד המורכב הזה, אני מקווה שלפחות הבנתם מה השימוש שלנו עם Synthetic MSRs ביחד עם ה-Hypervisor.

Virtual Machine Control Structure (VMCS)

ה-VMCS הוא מבנה נתונים שמנוהל על ידי ה-VMM לכל vCPU (Virtual CPU) והוא תחת אחריות ה-Virtual Machine Manager (VMM) כשבכל שינוי בהקשר ההרצה בין מכונות וירטואליות שונות, ה-VMM טוען VMCS מתאים ויגדיר את המצב הנוכחי של המעבדים הוירטואלים של אותה מכונה.

ה-VMCS כולל שש קבוצות לוגיות:

- Guest-state area: כשיהיה לנו VM exit ממכונה וירטואלית, מצב המעבד יישמר באיזור זה.
- Host-state area: כשנצא ממכונה וירטואלית בעזרת VM exit המערכת המארחת תקח שליטה על המעבד, מה שישנה את מצב האוגרים המאוחסנים באזור זה ויחזיר את הקשר ההרצה של המעבד מהאורח למארח. ולאחר מכן נמשיך לבצע את הפעולות ש-rip מצביע עליהן.
- VM-execution control fields: שדות שיישלטו בפעולות המעבד במצב VMX non-root operation.
- VM-exit control fields: שדות ששולטים ב-VM exits.
- VM-entry control fields: שדות ששולטים ב-VM entries.
- VM exit information area: איזור לקריאה בלבד ומציג את מספר השגיאות שקרו לאחר שפעולת vmx נכשלה.

למידע נוסף על Intel VT-x מוזמנים לקרוא את הבלוג הבא על יסודות הטכנולוגיה וכמובן את המדריך של אינטל.

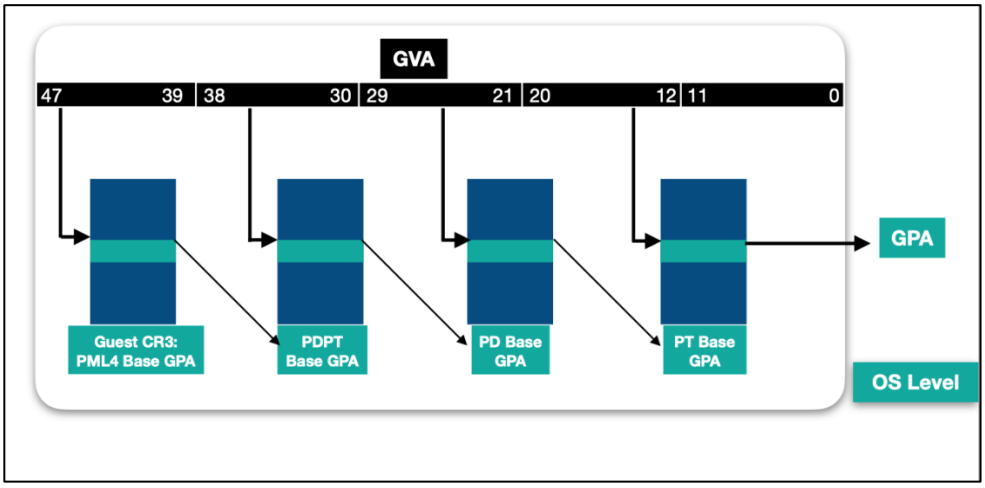
VTL – Virtual Trust Level

Virtual Trust Level (VTL) היא יחידת הרצה שמקושרת עם מעבד וירטואלי (Virtual Processor). ב-Windows יש לנו 2 רמות אמון וירטואליות, VTL 0 ו-VTL 1 כשב-VTL 0 ירוץ קרנל ה-NT וב-VTL 1 ירוץ הקרנל הבטוח, באופן כללי Hyper-V תומך בכעד 16 רמות אמון וירטואליות.

Second Level Address Translation (SLAT)

אז אחרי שהבנו את הבסיס לוירטואליזציה, ניגע במושג האחרון שיסדר את המחשבה לפני צלילה למחקר. כתובות זיכרון וירטואליות במעבד מהוות אבסטרקציה לכתובות פיזיות, במקום שכל תוכנה תוכל לכתוב לכתובת פיזית כיום לכל תוכנה יש את מרחב הזיכרון הוירטואלי שלה. שני תהליכים יכולים לגשת לאותה כתובת וירטואלית שמצביעות לכתובות פיזיות אחרות. דבר זה אפשרי הודות לתרגום הכתובות במעבד. שבו כל כתובת וירטואלית היא שדה ביטים המקודד היסטים בטבלאות המשמשות לתרגום כתובות במהלך page-table walk (מעבר טבלת דפים). כל רשומה הנקראת בטבלה מפנה לכתובת הפיזית של בסיס הטבלה הבאה. כאשר מגיעים לטבלה האחרונה, הרשומה הנקראת Page Table Entry (PTE) שמכילה את מספר המסגרת של הדף הפיזי המתקבל ואת זכויות הגישה אליו.

מעבר הדפים (page walk) מתחיל מטבלת PML4, שכתובתה מצוינת על ידי האוגר CR3 בהקשר של התהליך המנסה לבצע את הגישה.



וירטואליזציית חומרה מציגה שכבה נוספת בתהליך תרגום הכתובות, על ידי הוספת רמה חדשה בהיררכיית הטבלאות. תכונה זו של וירטואליזציית חומרה, הנקראת Second Level Address Translation (SLAT) או nested-paging, מאפשרת להפוך לוירטואלי את הזיכרון הפיזי של המכונה האורחת. היא מתבססת על מנגנון טבלת הדפים המורחבת (EPT) שמגדיר 4 מרחבי כתובות שונים:

- Guest Virtual Addresses (GVA) - כתובות וירטואליות כפי שהן נראות על ידי התהליכים הרצים במכונה הוירטואלית.
- Guest Physical Addresses (GPA) - כתובות פיזיות כפי שהן נראות על ידי מערכת ההפעלה האורחת כזיכרון פיזי.
- System Virtual Addresses (SVA) - כתובות וירטואליות כפי שהן נראות על ידי ה-Hypervisor.
- System Physical Addresses (SPA) - כתובות פיזיות המצביעות אל הזיכרון הפיזי בחומרה.

ולפי כך נחלק את מרחבי הזיכרון של ה-Hypervisor ושל האורח, כך שהאורח לא יוכל לקרוא את מרחב הזיכרון של ה-Hypervisor.

המעבר בין העולמות

עכשיו שעברנו על המושגים, הקונספטים ויש לנו בסיס להבנת המאמר, נוכל להבין את החלקים המעניינים. כפי שציינתי בהקדמה, כל קרנל Windows מודרני רץ בתור אורח על המחשב הפיזי, למה זה בעצם?

עם ההצגה של אבטחה מבוססת וירטואליזציה (נקרא למושג VBS מעכשיו), אנחנו מחלקים את מערכת ההפעלה לעולם הרגיל ולעולם הבטוח, כשהחלקים הרגישים של מערכת ההפעלה כגון TPM ונתונים ביומטריים יישמרו במרחב הזיכרון של העולם הבטוח. תכונה עיקרית של VBS היא Memory Integrity שבה בעצם ה-Hypervisor לא נותן לעמוד זיכרון להיות כתיב ובעל יכולת הרצה באותו זמן, אפשר לחשוב על זה כהרצת פעולת XOR על הדגלים W (Writeable) ו-X (Executable), בגלל ש-0 VTL לא יכול לשנות רשומות ב-SLAT, הבעלים של ה-SLAT של VTL 0 הוא VTL-1 כשהקרנל הבטוח מנהל זאת דרך הפונקציה HvCallModifyVtlProtectionMask שנקראת על ידי ה-hypercall 0x000c, מצורף ההסבר מה-TLFS של מייקרוסופט:

15.15.2 **HvModifyVtlProtectionMask**

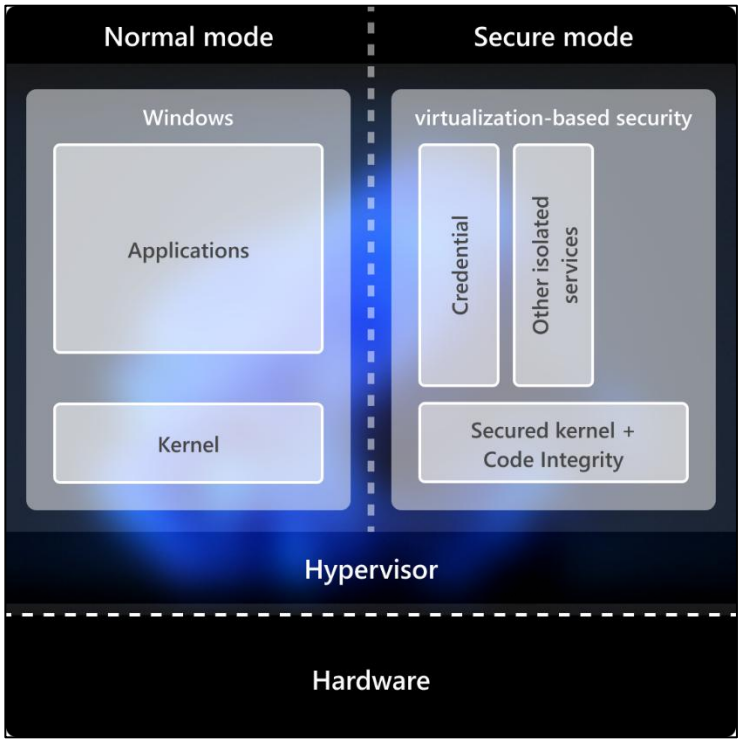
The HvModifyVtlProtectionMask hypercall modifies the VTL protections applied to an existing set of GPA pages.

Wrapper Interface

```
HV_STATUS
HvModifyVtlProtectionMask (
    _in HV_PARTITION_ID         TargetPartitionId,
    _in HV_MAP_GPA_FLAGS        MapFlags,
    _in HV_INPUT_VTL            TargetVtl,
    _in_ecount(PageCount)       HV_GPA_PAGE_NUMBER GpaPageList
);
```

ההסבר מציין כי VTL יכול לשים הגנות רק על VTL ברמה נמוכה ממנו, וכל ניסיון לשנות הגנות על טווחי זיכרון שלא נמצאים ב-RAM יחזירו HV_STATUS_INVALID_PARAMETER.

כפי שניתן לראות בדיאגרמה הבאה אין תקשורת ישירה ביניהם, אז התקשורת היחידה שיכולה להיות היא באמצעות ה-Hyper-V Hypervisor.



ה-Hypervisor מיוצג על ידי הקובץ hvix64.exe במחשבים שמבוססים על מעבדי Intel, hvax64.exe בשביל AMD ו-hvaa64.exe ב-ARM. תכונות הוירטואליזציה ממומשות באופן שונה בכל ארכיטקטורה, מה שב-Intel יהיה Intel VT-x שיביא לנו את ה-Virtual Machine Control Structure (VMCS) וה-SLAT, ב-AMD יהיה AMD-V שיביא לנו את ה-Virtual Machine Control Block (VMCB) וב-ARM המונח יקרא ARM Virtualization Extension ככה שכל גרסה של ה-Hypervisor שונה מהאחרת. אנחנו נתמקד ב-Hypervisor hvix64.exe, שמיישם את ה-Intel VT-x.

בשביל להתחיל להבין איך VTL 0 מתקשר עם VTL 1 נפתח את הקובץ ntoskrnl.exe (שלהבדיל מ-hvix64.exe כן יהיה דומה בין ארכיטקטורות) ונשים לב לפונקציה VslpEnterIumSecureMode, שלפי שמה נוכל להסיק שמתבצעת תקשורת כלשהי עם ה-Isolated User Mode (IUM) בקרנל הבטוח. שימו לב שפונקציות שיפעילו בקשות בטוחות (Secure Calls) יתחילו עם הקידומת Vsl. נראה בפרולוג הפונקציה בדיקה אם Virtual Secure Mode (VSM)מופעל:

```
if ( VslVsmEnabled )
{
  CurrentIrql = KeGetCurrentIrql();
  EntryIrql = CurrentIrql;
  __writecr8(0xFu);
  if ( KiIrqlFlags )
    KiRaiseIrqlProcessIrqlFlags(CurrentIrql, 15);
}
```

אם נסתכל על ההפניות הצולבות של הפונקציה, נראה שלושה מסלולים שניתן לקחת:

1. בקשה סינכרונית מקרנל ה-NT לקרנל הבטוח.
2. בקשה אסינכרונית מהקרנל הבטוח לקרנל ה-NT, מכיוון שהקרנל הבטוח לא מתעסק עם פעולות קלט ופלט וקריאת I/O, הוא צריך להשתמש בפונקציות מקרנל ה-NT (אפשר לראות זאת גם בטיפול ב-HvCallVtlReturn hypercall שבה נקבל את הסטטוס/ערך בחזרה מהקרנל הבטוח לאחר הבקשה ששלחנו).
3. בקשות מתהליכים ב-Isolated User Mode לקרנל הבטוח.

נאמר לנו בספר Windows Internals 7th edition part 2 שהארגומנט הראשון יהיה אובייקט SKCALL בגודל 104 (0x68) בתים שמטרתו להעביר את הפעולה שנרצה לבצע ויכיל שדות כמו:

- Secure Handle – ה-Handle לתהליך שה-Thread ירוץ תחתיו.
- MDL – מצביע ל-MDL שיכיל מידע רלוונטי לאובייקט.
- CID – ערך שמתאר את ה-Process ID (PID) וה-Thread ID (TID).

עכשיו אנחנו יודעים שהפונקציה מקבלת SKCALL בפרמטר הראשון ו-Secure Service Call Number (SSCN) בפרמטר השני. הפרמטר הראשון יהיה הפעולה שנרצה לבצע, לאחר הצלבה של כל הקריאות לפונקציה נסיק את הקודים הבאים של הפעולות:

- 0 – המשכת הרצה של Secure Thread או כניסה מחדש אליו לאחר קריאה רגילה (מקרנל ה-NT לבטוח).
- 1 – אם ה-thread הנוכחי רץ בהקשר של enclave, הקרנל הבטוח מסמן לנו לצאת מלולאת ה-dispatch בלי לגעת ב-IRQL.
- 2 – קריאות לשירותים שיפעילו את הפונקציה IumInvokeSecureService בקובץ של הקרנל הבטוח securekernel.exe ב-VTL 1.
- 3 – ניקוי ה-Translation Lookaside Buffer (TLB).

עכשיו אנחנו נראה שהפונקציה HvlSwitchToVsmVtl1 אחראית על שליחת ה-Hypercall המתאים לקרנל וגם לקבלת התשובה ממנו, שיכולה לכלול בקשה משירות שקרנל ה-NT מספק בעזרת הפונקציה PsDispatchIumService:

```
  while ( 1 )
  {
LABEL_38:
    if ( (BYTE4(xmmword_140FC5B50) & 8) != 0 )
    {
      EtwTraceEnterVtl1(v5, Sscn);
      // Round-trip into the SK.
      HvlSwitchToVsmVtl1(0, SkCall, SecureThreadSaved);
      Status = *(SkCall + 8);
      EtwTraceExitVtl1(v5, Sscn);
    }
    else
    {
      HvlSwitchToVsmVtl1(0, SkCall, SecureThreadSaved);
      Status = *(SkCall + 8);
    }
    // Return-reason: 6=done, 1=early exit, 0/5=PsDispatchIumService,
    // 2=IUM syscall (UM), 3=IUM syscall (KM); bit 7 = SK-requested int 3.
    v18 = *(SkCall + 1);
    if ( v18 < 0 )
    {
      // if bit 7 of the return-reason byte is set,
      // NT executes an int 3 which will break into the debugger.
      __debugbreak();
      *(SkCall + 1) &= ~0x80u;
      v18 = *(SkCall + 1);
    }
```

הפונקציה PsDispatchIumService משמשת כנקודת הכניסה שלנו ל-Normal Call ותרוץ כשהקרנל הבטוח יחזיר סטטוס 0 או 5 לפי אובייקט ה-SKCALL. הפונקציה תנהל את הקריאות מהקרנל הבטוח לקרנל ה-NT:

```
    switch ( *(SkCall + 1) )
    {
      case 0:
LABEL_53:
        PsDispatchIumService(SkCall, v21, v18, v19);// Reason 0/5: NT service callback the SK needed.
        break;
      case 2:
        if ( !CurrentThread->PreviousMode )     // Reason 2 from KernelMode -> STATUS_INVALID_DEVICE_REQUEST.
        {
          *(SkCall + 8) = 0xFFFFFFFFC0000030uLL;
          break;
        }
```

בפונקציה HvlSwitchToVsmVtl1 נראה שאין שום קריאה לפעולה כמו vmcall בשביל לקרוא ל-hypercalls המתאימים, הקטע היחיד שיכול להביא אותנו לשם הוא קריאה לאיזור בזיכרון עם המצביע HvlpVsmVtlCallVa שהקריאה אליו נראית כך:

```
result = (*&HvlpVsmVtlCallVa)(a1, SKCALL, KeGetCurrentIrql(), SECURE_THREAD)
```

אך כשנסתכל בזיכרון נראה שהאיזור אינו מכיל דבר, זה מכיוון שמדובר באיזור ריק שימולא בזמן הרצה לאחר שה-Hypervisor יאותחל.

```
CFGRO:0000000141201860 HvlpVsmVtlCallVa dq 0                    ; DATA XREF: VslpEnterIumSecureMode+24↑r
CFGRO:0000000141201860                                          ; VslGetNestedPageProtectionFlags+2E↑r ...
```

```
1: kd> dq nt!HvlpVsmVtlCallVa L1
fffff800`de401860  fffff800`6c60000f
1: kd> u fffff800`6c60000f
fffff800`6c60000f 488bc1          mov     rax,rcx
fffff800`6c600012 48c7c111000000  mov     rcx,11h
fffff800`6c600019 0f01c1          vmcall
fffff800`6c60001c c3              ret
fffff800`6c60001d 8bc8            mov     ecx,eax
fffff800`6c60001f b812000000      mov     eax,12h
fffff800`6c600024 0f01c1          vmcall
fffff800`6c600027 c3              ret
```

בדיבוג דינאמי של הקרנל נראה שמדובר בקטע קוד קצר שמכיל שתי פונקציות, אחת תשלח vmcall עם הערך 0x11 (17) והשנייה 0x12 (18).

בקטע Appendix A: Hypercall Code Reference של התיעוד של מייקרוסופט נוכל לראות שקוד hypercall 0x11 יהיה שייך לפונקציה HvCallVtlCall, שתשמש אותנו לשליחת הבקשה הבטוחה (Secure Call), ו-0x12 ל-HvCallVtlReturn, שתשמש אותנו לקבל הערך/סטטוס הבקשה.

בוחנים את ה-Hypervisor

עכשיו שהבנו מאיפה אנחנו שולחים את קריאת ה-vmcall, נותר לנו לראות איפה ה-VM exit שלנו. VM exit הוא שם לאירוע שקורה כשהמעבד מחליף מפעולת VMX non-root לפעולת VMX root, שזה בעצם אומר שאנחנו מעבירים את ההרצה ל-Hypervisor.

נפתח את הקובץ hvix64.exe בדיסאסמבלר המועדף שלנו ונראה קובץ של 2MB שאין לנו סימבולים בשבילו, וכשזה המצב, הדבר הכי טוב הוא לראות מחקרים ומאמרים קודמים על הקובץ, למרבה מזלנו יש את Saar Amar שכתב על כמה שיטות שיכולות לעזור לנו עם המחקר, אחת מהם היא לבצע הבדל בין בינארים ולשלב ביניהם. הבדל בין בינארים היא טכניקה נפוצה שחוקרי חולשות משתמשים בה בעיקר כדי לראות באיזה חלק של מערכת תוקנה חולשה לאחר שהתפרסמה, בעזרת השוואה של הגרסה הישנה עם החדשה נראה רק את מה שהשתנה ובכך נוכל לסנן הרבה רעש ולראות באיזה איזור בוצע התיקון.

במקרה שלנו לא אכפת לנו מפונקציה ספציפית, אנו רוצים לראות אילו פונקציות שוות במרבית האחוזים לאלו שיש לנו. לדוגמה, ניתן לראות שללא סימבולים כל הפונקציות יקראו sub_xxxxxxxx, דבר שלא אומר לנו הרבהף, אך אם נבצע הבדל בינארי בין hvix64.exe ל-ntoskrnl.exe נוכל לדעת מה השם של חלק מהפונקציות ולשנות את שמן ואת שמות המשתנים שבתוכן, אפשר לראות שקיבלנו די הרבה פונקציות דומות מקובץ מערכת אחד, אני השתמשתי בתוכנה BinDiff ל-IDA:

Matched Functions

Similarity	Confidence	Change	EA Primary	Name Primary	EA Secondary	Name Secondary	Con
1.00	0.99	-------	FFFFF8000...	__GSHandlerCheck	000000014...	__GSHandlerCheck	
1.00	0.99	-------	FFFFF8000...	sub_FFFFF8000022DC70	000000014...	SymCryptWipeAsm	
1.00	0.99	-------	FFFFF8000...	sub_FFFFF8000022E060	000000014...	SymCryptSha256Append	
1.00	0.99	-------	FFFFF8000...	sub_FFFFF800002512AC	000000014...	KdpQuickMoveMemory	
1.00	0.99	-------	FFFFF8000...	sub_FFFFF80000321EB0	000000014...	Uart16550LegacyInitializePort	
1.00	0.99	-------	FFFFF8000...	sub_FFFFF800003220C0	000000014...	Uart16550SetBaud	
1.00	0.99	-------	FFFFF8000...	sub_FFFFF800003222DC	000000014...	UartpSetAccess	
1.00	0.99	-------	FFFFF8000...	sub_FFFFF80000322588	000000014...	IaLpssPciSetPower	
1.00	0.99	-------	FFFFF8000...	sub_FFFFF800003227D0	000000014...	IaLpssSetPowerD0	
1.00	0.99	----...	FFFFF8000...	sub_FFFFF80000322880	000000014...	IaLpssSetPowerD3	
1.00	0.99	-------	FFFFF8000...	sub_FFFFF80000322B30	000000014...	SpiMax311GetByte	
1.00	0.99	----...	FFFFF8000...	sub_FFFFF80000322E74	000000014...	SpiSend16	
1.00	0.99	-------	FFFFF8000...	sub_FFFFF800003A57A0	000000014...	HvcallpExtendedFastHypercallWithO...	
1.00	0.99	-------	FFFFF8000...	sub_FFFFF800003B2380	000000014...	memmove	
1.00	0.99	-------	FFFFF8000...	sub_FFFFF80000230C50	000000014...	SymCryptSha256AppendBlocks_xm...	
1.00	0.99	----...	FFFFF8000...	sub_FFFFF80000321F50	000000014...	Uart16550PutByte	
1.00	0.99	----...	FFFFF8000...	sub_FFFFF80000321CE0	000000014...	Uart16550GetByte	
1.00	0.99	-------	FFFFF8000...	sub_FFFFF800003B2750	000000014...	memcmp	
1.00	0.99	----...	FFFFF8000...	sub_FFFFF80000322080	000000014...	Uart16550RxReady	
1.00	0.99	----...	FFFFF8000...	sub_FFFFF8000022DFCC	000000014...	SymCryptFdefRawMul	
1.00	0.99	-------	FFFFF8000...	sub_FFFFF8000022E200	000000014...	SymCryptSha256AppendBlocks_shani	
1.00	0.99	-------	FFFFF8000...	sub_FFFFF800002A18F8	000000014...	PopInterruptSteeringEnabled	
1.00	0.99	-------	FFFFF8000...	sub_FFFFF8000032CCD8	000000014...	MmHasImageBeenImportOptimized	
1.00	0.99	-------	FFFFF8000...	sub_FFFFF80000322C90	000000014...	SpiMax311RxReady	
1.00	0.99	-------	FFFFF8000...	sub_FFFFF8000021E4E4	000000014...	HalpTimerGetInternalData	
1.00	0.99	-------	FFFFF8000...	sub_FFFFF80000322530	000000014...	IaLpssInitializePort	
1.00	0.99	-------	FFFFF8000...	sub_FFFFF80000322AF8	000000014...	SpiMax311BufferRxData	
1.00	0.99	----...	FFFFF8000...	sub_FFFFF800003229DC	000000014...	SpiInit	
1.00	0.99	-------	FFFFF8000...	sub_FFFFF80000322C00	000000014...	SpiMax311PutByte	
1.00	0.99	----...	FFFFF8000...	sub_FFFFF800003220E0	000000014...	Uart16550SetBaudCommon	
1.00	0.99	-------	FFFFF8000...	sub_FFFFF8000022D91C	000000014...	__GSHandlerCheckCommon	
1.00	0.99	----...	FFFFF8000...	sub_FFFFF80000322720	000000014...	IaLpssReadCmdStatus	
1.00	0.99	----...	FFFFF8000...	sub_FFFFF80000322760	000000014...	IaLpssReadPmcsr	
1.00	0.99	----...	FFFFF8000...	sub_FFFFF8000032293C	000000014...	IaLpssWriteCmdStatus	
1.00	0.99	----...	FFFFF8000...	sub_FFFFF80000322978	000000014...	IaLpssWritePmcsr	
1.00	0.99	-------	FFFFF8000...	sub_FFFFF8000022E5C4	000000014...	SymCryptSha256AppendBlocks_ul1	
1.00	0.99	-------	FFFFF8000...	sub_FFFFF800003A5BC0	000000014...	KiEncls	
1.00	0.99	----...	FFFFF8000...	sub_FFFFF8000022DBE4	000000014...	SymCryptInitEnvCommon	
1.00	0.99	----...	FFFFF8000...	sub_FFFFF80000321DF4	000000014...	Uart16550InitializePortCommon	
1.00	0.99	-------	FFFFF8000...	sub_FFFFF80000322E3C	000000014...	SpiMax311TxEmpty	
1.00	0.98	-------	FFFFF8000...	sub_FFFFF8000021E920	000000014...	HalpCallEfiGetTime	
1.00	0.98	-------	FFFFF8000...	sub_FFFFF8000022D998	000000014...	SymCryptCpuidExFuncEnvWindows...	
1.00	0.98	-------	FFFFF8000...	sub_FFFFF8000022DC40	000000014...	SymCryptWipe	
1.00	0.98	-------	FFFFF8000...	sub_FFFFF8000022E7EC	000000014...	SymCryptSha256Init	

אני בחרתי לחבר ל-hvix64.exe כל פונקציה עם דמיון של 0.99 ומעלה וביטחון של 0.98 ומעלה, אך מניסיון גם פונקציות עם 0.95 דמיון היו שוות לפעמים אז שווה לנסות. לאחר שהשוותי מול מספיק קבצי מערכת קיבלתי די הרבה פונקציות, והדבר שעשיתי לאחר מכן הוא להריץ את סקריפט הפייתון שיצר gerhart01 ולשדרג אותו ל-IDA 9.2. הסקריפט זמין פה.

חיפוש ה-VM exit handler

יש מספר דרכים מתועדות על איך למצוא את הפונקציה שתטפל באירועי VM exit אך לי היה הרבה מזל, חיפשתי "VM" לאורך כל המחרוזות של הקובץ ומחרוזת אחת צצה לעיני, "[%d] MinimalLoop VMX_EXIT_REASON_INIT_INTR. Rebooting the system". למחרוזת רק קריאה אחת במה שנראה כמו הפונקציה שמנהלת אירועי VM exit, עקב הגישה הישירה לאוגרים ובדיקה של קודים התואמים סיבות של אירועי VM exit. ללא המחרוזת הזו היינו יכולים לחפש פעולות vmresume, כי לאחר יציאה מהמכונה הוירטואלית אנחנו צריכים להמשיך אותה (resume). נסתכל על הקובץ vmx.h של Alex Ionescu ונראה את הקודים המתאימים למה שהפונקציה בודקת, לדוגמה הערך 0x12 (18) ייצג את הסיבה ליציאה VMX_EXIT_REASON_VMCALL שתראה לנו איזו פונקציה אחראית על ניהול hypercalls ובדיקתן.

```
    if ( exitReason != 1 )
    {
      switch ( exitReason )
      {
        case VMX_EXIT_REASON_INIT_SIGNAL:
          sub_24C9C0("[%d] MinimalLoop VMX_EXIT_REASON_INIT_INTR. Rebooting the system\n", *(self + 160));
          v4 = 3099;
          goto LABEL_244;
        case VMX_EXIT_REASON_EXECUTE_CPUID:
          _RAX = **v76;
          __asm { cpuid }
          v66 = _RAX;
          if ( **v76 == 1073741828 )
            v66 = _RAX & 0xFFFFEFFF | ((qword_B1CE0 & 1) << 12);
          v7 = v76;
          **v76 = v66;
          *(*v7 + 3) = _RBX;
          *(v7 + 32) |= 0x80u;
          *(*v7 + 1) = _RCX;
          *(*v7 + 2) = _RDX;
LABEL_89:
          v5 = v73;
          goto LABEL_213;
        case VMX_EXIT_REASON_EXECUTE_INVLPG:
          goto LABEL_213;
        case VMX_EXIT_REASON_EXECUTE_VMCALL:
          v60 = HypercallHandler(v5, &v77);
          v10 = 2;
```

סקירת ה-Hypercall HvCallVtlCall

לאחר שניכנס לפונקציה HypercallHandler נראה קטע switch שייבדוק את קוד ה-hypercall שניתן ויפעיל את הפונקציה המתאימה, כפי שציינתי קודם בעזרת ה-TLFS נוכל לראות איפה הפונקציות שאחראיות על HvCallVtlCall ו-HvCallVtlReturn. לפני שנסתכל על הפונקציה חשוב שנבין של-hypercalls ב-Hypervisor יש 3 calling conventions:

1. Standard – מקבל GPA (Guest Physical Address) של פרמטפר קלט דרך האוגר rdx ופלט דרך r8.
2. Fast – מקבל שתי ארגומנטים של קלט שמועברים באוגרים rdx ו-r8.
3. Extended Fast – משתמשת בשישה אוגרי XMM כדי לתמוך בבלוק קלט של עד 112 בתים.

אם תשימו לב ותחזרו לאיזור בזיכרון שנקרא HvlpVsmVtlCallVa, תראו שכשאנחנו קוראים לפעולת ה-vmcall אנחנו לא קוראים לה באף אחת מהדרכים האלו, אנחנו מעביר לה רק את הקוד שאנחנו רוצים וזהו.

```
fffff800`6c60000f 488bc1          mov     rax,rcx
fffff800`6c600012 48c7c111000000  mov     rcx,11h
fffff800`6c600019 0f01c1          vmcall
fffff800`6c60001c c3              ret
```

אם נשווה זאת לבקשה רגילה ב-Application Binary Interface (ABI) של Hyper-V, נראה ש-rcx יפוצל למבנה של מספר חלקים. ביטים 0-15 יהיו קוד הקריאה, ביט 16 הוא דגל ה-fast, ביטים 17-26 יעבירו את ה-variable header, ביטים 32-43 את ה-rep count וכך הלאה עם GPA של קלט ופלט ב-rdx ו-r8. מה שמראה לנו ש-HvCallVtlCall ו-HvCallVtlReturn לא עוקבות אחר ה-ABI של hypercall ב-Hyper-V. אם נחזור ל-TLFS נראה שב-HvCallVtlCall ה-Hypervisor שומר על המצב של כל האוגרים רבי תכלית ואוגרי ה-XMM. וזה בדיוק איך שארבעת הארגומנטים ב-VslpEnterIumSecureMode עוברים את כל הדרך עד ל-Hypervisor.

```
    switch ( HypercallCode )
    {
      case HvCallInvokeHypervisorDebugger:
        v12 = HvCallInvokeHypervisorDebugger;
        v9 = 16;
        goto LABEL_38;
      case HvCallVtlCall:
        if ( !sub_236DDC() )
          goto LABEL_31;
        SecureCallResult = SecureCallHandler;
        break;
      case HvCallVtlReturn:
        if ( !sub_236DDC() )
        {
LABEL_31:
          v10 = 2;
          goto LABEL_18;
        }
        SecureCallResult = HvCallVtlReturn;
        break;
      default:
        switch ( HypercallCode )
        {
          case HvCallPostDebugData:
            v12 = HvCallPostDebugData;
```

לפני שנמשיך, כדי להבין את הקרנל הבטוח, יהיה טוב להבין את מבנה ה-Virtual Processor. אובייקט ה-Virtual Processor הוא אבסטרקציה של ה-Hypervisor ל-partition מסוים.

לאחר מכן ה-Hypervisor מנהל בעצמו את הקישור בין מעבדים וירטואלים (Virtual Processors) למעבדים הפיזיים הלוגים (מעבד לוגי ב-VMX הוא יחידת הרצה חומרתית כפי שנראית על ידי מערכת ההפעלה או ה-Hypervisor). נסתכל על הבלוג של סער ונראה את הציטוט הבא:

> *"You will probably notice accesses to different structures pointed by the primary gs structure. Those structures signify the current state (e.g. the current running partition, the current virtual processor, etc.). For instance, most hypercalls check if the caller has permissions to perform the hypercall, by testing permissions flags in the gs:CurrentPartition structure."*

מה שאנחנו לומדים מכך זה שאובייקט ה-Virtual Processor ביחד עם ה-current partition וה-privilege mask נמצאים בהיסט מסוים מהכתובת שבאוגר gs. ב-HvCallPostDebugData hypercall נראה הפנייה ל-gs:360h עם bitmask בהיסט 0x1b0, מכיוון שראיתי את התבנית הזו בהרבה hypercalls נוכל להניח ש-gs:360h יהיה ה-"current partition" ו-0x2b יהיה ה-privilege mask, ספציפית ביט הדיבוג כפי שנראה ב-enum HV_PARTITION_PRIVILEGE_MASK.

```
HvCallPostDebugData proc near           ; DATA XREF: .rdata:00000000000029D8↑o
                                        ; HypercallHandler:loc_3A33B7↓o

var_18          = qword ptr -18h

                sub     rsp, 38h
                mov     r9, gs:360h
                mov     r10, rdx
                mov     r8, rcx
                bt      qword ptr [r9+1B0h], 2Bh ; '+'
                jnb     short loc_28D52A
                cmp     cs:byte_6B040, 0
                jz      short loc_28D52A
                call    sub_251AC0
                test    al, al
                jz      short loc_28D52A
```

עכשיו מה שנשאר לנו הוא ה-Virtual Processor, אנחנו צריכים אותו מכיוון שמה שמייחד קריאה ל-VTL מכל vmcall נורמלי, ה-VMCS, ה-VtlMask, הם כולם ייחודיים ל-Virtual Processor לא ל-partition. התחלתי לראות פניות לאוגר gs, ובעזרת המאמר של Quarkslab ראיתי שאנחנו טוענים אותו רק שתי קריאות מעל מנהל ה-VM exit.

```
; __int64 sub_343D90()
sub_343D90      proc near               ; CODE XREF: sub_23F040:loc_23F4B2↑p

arg_0           = qword ptr  8

                mov     [rsp+arg_0], rbx
                push    rdi
                sub     rsp, 20h
                mov     rbx, gs:0
                mov     rdi, [rbx+368h] -> Load VirtualProcessor Object
                test    rdi, rdi
                jnz     short loc_343DB2

loc_343DAF:                             ; CODE XREF: sub_343D90+20↓j
                hlt
                jmp     short loc_343DAF

loc_343DB2:                             ; CODE XREF: sub_343D90+1D↑j
                xor     ecx, ecx
                call    sub_343E40
                lea     rcx, [rdi+0EC0h] -> Pass VirtualProcessor + 0xEC0 to
sub_3326D8 which later on passes it as the first argument to VMExitHandler
                xor     r8d, r8d
                mov     rdx, rbx
                call    sub_3326D8
                mov     rbx, [rsp+28h+arg_0]
                add     rsp, 20h
                pop     rdi
                retn
sub_343D90      endp
```

ה-Secure Call Handler

הפונקציה שתנהל את הבקשה לעבור ל־VTL 1 היא SecureCallHandler. היא תוציא קודם כל אובייקט מ-VirtualProcessor + 0x3c0, ולאחר מכן אובייקט נוסף מהיסט 0x14. כשעוברים ל-VTL חדש, ה-Hypervisor לא רק מתעד זאת ב-Virtual Processor, אלא גם מתחיל להריץ את הפעולות בהקשר של אותו VTL – עם VMCS שונה, SLAT שונה ומצב אוגרים שונה. Hyper-V מנהל את ה"Current VTL" דרך מבנה ה-Virtual Processor ובגרסה הזו של Hyper-V השדה "Current VTL" מנוהל דרך המעבד הוירטואלי הנוכחי בהיסט 0x3c0 – המצביע לתוך מערך ה-VtlArray של המעבד הוירטואלי, שתמיד מצביע לאיבר הפעיל כרגע. בנוסף השדה VtlNumber בהיסט 0x14 בתוך האובייקט הזה מציין איזה VTL הוא מייצג.

```
void __usercall SecureCallHandler(
    _VIRTUAL_PROCESSOR *VirtualProcessor,
    __int64 SecureCallReady
)
{
    int currentVtl;          // eax
    bool isVtlInitialized;   // zf
    int mask;                // esi

    currentVtl = 1 << VirtualProcessor->CurrentVtl->VtlNumber;

    isVtlInitialized = !_BitScanForward(
        &mask,
        VirtualProcessor->VtlMask & ~(currentVtl | (currentVtl - 1))
    );

    if (!isVtlInitialized && !SecureCallReady)
    {
        FixupVtl0RipToNextInstruction(
            VirtualProcessor->VmExitInstructionLen
        );

        SetupVtlTransition(VirtualProcessor, mask);
        FinishTransition(VirtualProcessor, mask, 1LL);
    }
}
```

במקרה של Virtual Processor + 0x3c0 ספציפית, זהו ה-VTL שבו המעבד נמצא כרגע. במקרה הזה ה-decompiler של IDA השתבש לי ולכן ערכתי את הפסאודו-קוד בעצמי.

ה-VtlMask עוקב אחר אילו VTLים אותחלו – כלומר ה-bitmask שבה כל ביט מייצג את מצב האתחול של VTL מסוים. כפי שמוצג, מבנה ה-Virtual Processor מחזיק שני שדות קשורים: ה-VTL הפעיל כרגע, ומערך שמכיל רשומה עבור כל VTL אפשרי במערכת.

ברגע שה-SecureCallHandler קובע שהקריאה רשאית להמשיך, הפעולה הראשונה שלו היא לקדם את ה-instruction pointer של ה-VTL הנוכחי. כדי להבין למה זה דווקא הצעד הראשון, צריך לזכור ש-Virtual Secure Mode (VSM) מציג שני מבני VMCS נפרדים: אחד שייך ל-VTL 0 והשני ל-VTL 1. במקרה שלנו VTL 0 הוא ה-VTL הנוכחי. הקונבנציה הסטנדרטית לטיפול ב-VM exit היא לקדם את ה-instruction pointer של ה-guest מעבר להוראה שגרמה ליציאה (exit):

```
VirtualProcessor->VtlMask |= 1 << targetVtl;
VirtualProcessor->CurrentVtl = VirtualProcessor->VtlArray[targetVtl];
```

ובכך כשה-Hypervisor מסיים את עבודתו ומבצע VM entry האורח ממשיך מההוראה שאחריה. הקידום הזה חייב לקרות לפני המעבר ל-VTL 1, אחרת כש-VTL 0 יחזור לרוץ הוא יבצע את אותה הוראת vmcall שוב ושוב. העדכון מתבצע דרך ה-Enlightened VMCS, או ישירות מול ה-VMCS באמצעות ההוראות vmread ו-vmwrite.

```
void __fastcall SetupVtlTransition(
    _VIRTUAL_PROCESSOR *VirtualProcessor,
    unsigned __int8 TargetVtl
)
{
    __int64 self;         // rsi
    __int64 currentVtl;   // r8

    self = __readgsqword(0);
    currentVtl = VirtualProcessor->CurrentVtl->VtlNumber;

    if (currentVtl != TargetVtl)
    {
        if (byte_FFFFF800000785E0)
        {
            if ((dword_FFFFF800000785C8 & 0x2000) != 0)
                sub_FFFFF8000025E15C(
                    0x1D4D,
                    currentVtl | (TargetVtl << 16)
                );
        }

        sub_FFFFF800002AF304(self, VirtualProcessor);
        PerformVtlTransition(self, VirtualProcessor, TargetVtl);
    }
}
```

לאחר תיקון ה-instruction pointer של VTL 0, ההרצה עוברת ללוגיקת המעבר ל-VTL 1. אחת הבדיקות המקדימות היא לוודא שה-VTL הייעודי שונה מה-VTL הנוכחי.

וכאן אנחנו מעדכנים את נתוני ה-VTL כך שיכילו את המצב החדש של VTL 1, ומעדכנים גם את מצב ה-Virtual Processor הנוכחי כדי שיידע שהוא נמצא כעת תחת VTL 1.

```
void __fastcall PerformVtlTransition(__int64 Self, _VIRTUAL_PROCESSOR
*VirtualProcessor, unsigned __int8 TargetVtl)
{

    //
    // Get the new VTL 1 we target
    //
    newVtlData = VirtualProcessor->VtlArray[TargetVtl];

    //
    // Update the current Virtual Processor state to VTL 1
    //
    VirtualProcessor->CurrentVtlNumber = TargetVtl;

    //
    // Update the current VTL data for the current processor
    //
    VirtualProcessor->CurrentVtl = newVtlData;
}
```

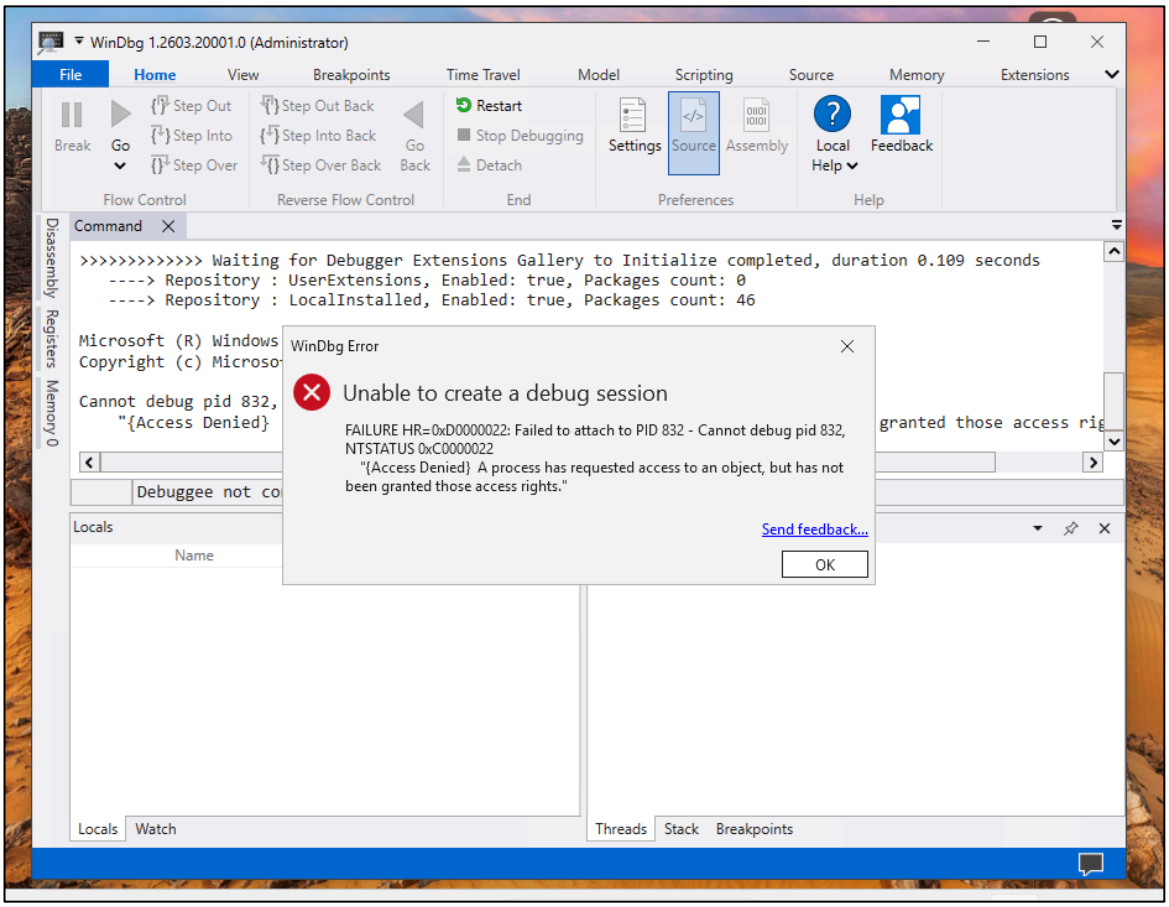
עכשיו כשסידרנו את כל השדות, נעבור להחלפת ה-VMCS. יש להחליף את ה-VMCS הפעיל של ה- Virtual Processor במבנה ששייך ל-VTL 1. נוכל לראות זאת בפונקציה TransitionToNewVtl. ה- VTL הנכנס מתואר במה שנקרא לו private VTL data. הרלוונטיויות של המבנה הזה היא שהוא מחזיק מצביע ל-VMCS היעד. ברגע שהמצביע הזה נגיש, ההחלפה מתבצעת: בפלטפורמות לא "מוארות" (enlightened), הפקודה vmptrld מתבצעת מול הכתובת הפיזית של ה-VMCS ובמערכות "מוארות", ה-VMCS נטען לפי הכתובת הוירטואלית שלו.

```
void __fastcall TransitionToNewVtl(__int64 Self, _VTL_PRIVATE_DATA
*PrivateVtlData)
{
    _HV_VMX_ENLIGHTENED_VMCS *enlightenedVmcs; // rdx
    unsigned __int64 v6; // r8
    unsigned __int64 v7; // r8
    unsigned __int64 v8; // r8
    __int64 v9; // rax
    _VTL_VMCS_DATA *VtlVmcsData; // rax
    unsigned __int64 vtlVmcsPhysAddr; // rcx
    unsigned __int64 self; // rax
    __int64 v13; // [rsp+38h] [rbp+10h]

    PrivateVtlData->VtlVmcsData->Unknown = 0;
    _RCX = PrivateVtlData->VtlVmcsData;
    enlightenedVmcs = _RCX->VtlVmcsEnlightenedAddress;

    if (enlightenedVmcs)
    {
        //
        // Do we use enlightenments?
        //
        if ((dword_FFFFF800000AECB0 & 1) != 0)
        {
            vtlVmcsPhysAddr = _RCX->VtlVmcsPhysicalAddress;
            enlightenedVmcs->SyntheticControls = 1;
            self = __readgsqword(0);

            //
            // Update the current VMCS to that of VTL 1
            //
            *(self + 0x2C680) = enlightenedVmcs;
            *(*(self + 0x2C4C8) + 0x30LL) = vtlVmcsPhysAddr;
        }
    }
    else
    {
        __asm { vmptrld qword ptr [rcx+188h] }
    }
}
```

Isolated User Mode (IUM)

עכשיו שהבנו איך נעבור מ-VTL 0 ל-VTL 1, נסתכל על שכבת ה-User Mode של ה-Secure Kernel שנקראת Isolated User Mode (IUM), בשכבה זו תהליכים מתקשרים עם קרנל ה-NT דרך system calls. תהליכים נפוצים בשכבה זו כוללים את Lsalso.exe ו-vmsp.exe. אפילו עם הרשאות ברמת הקרנל ב-VTL 0, זה בלתי אפשרי לשנות את הזיכרון של VTL 1. העיצוב הזה מגן נגד מתקפות ברמת הקרנל, ומגן על מידע חשוב כמו סיסמאות משתמשים, מפתחות הצפנה של BitLocker ואפילו ביומטריה.

ה-Secure Kernel הוא האחראי להעברת ה-System Calls של תהליכים ב-IUM. תודה לירדן שפיר שכתבה מדריך על איך להשתמש ב-LiveCloudKd ולדבג את ה-Secure Kernel ול-gerhart שכתב את הכלי LiveCloudKd. אבל עלתה בי המחשבה, איך זה שאי אפשר לדבג תהליכים שרצים ב-Isolated User Mode? לאחר שאפתח את הדיבגר WinDbg בתור מנהל, וננסה לדבג איתו את התהליך Lsalso.exe, נקבל את השגיאה הבאה:



מה שמראה שגם למרות שאנחנו מנהלים, אנחנו לא יכולים לדבג את התהליך.

דיבוג תהליכים ב-Secure Kernel

בשימוש בגרסה החדשה ביותר של LiveCloudKd, מצאתי דרך לדבג תהליכים שרצים ב-Isolated User Mode בעזרת השימוש במכונה וירטואלית. העיקרון חבוי בכך שה-Virtual Secure Mode במכונה הוירטואלית מבודד אזורי זיכרון של VTL-ים שונים. במכונה המארחת עדיין אפשר להשתמש ב-hypercalls ברמת הקרנל או ב-API של הדרייבר winhv.sys, כדי לגשת לזיכרון הפיזי הליניארי של המכונה הוירטואלית.

LiveCloudKd מספק דרייבר חתום בשם hvmm.sys, שחושף את היכולות האלה דרך API ברמה האפליקטיבית – כלומר, זיכרון שמוקצה ל-VTL 1 בתוך מכונה וירטואלית עדיין נגיש מהמכונה המארחת באמצעות כתובות פיזיות. כדי לדבג את תהליכי IUM, נצטרך לשנות את הפונקציה SkpsIsProcessDebuggingEnabled שנמצאת ב-securekernel.exe בזמן הרצה. בגרסה החדשה של ה-Secure Kernel הפונקציה הזאת מוסתרת בתוך הפונקציה IumInvokeSecureService, שאליה יגיעו הבקשות שלנו לשירותים השונים של ה-Secure Kernel. נוכל לראות שהפונקציה SkpsIsProcessDebuggingEnabled מעניקה את ההרשאות לדבג תהליכים – אך את שינוי הסטטוס בפועל מבצעת פונקציית האחות שלה, SkpsEnableDebugging.

```
__int64 __fastcall SkpsEnableDebugging(__int64 PROCESS, char a2)
{
  __int64 v3; // rdi
  __int64 v5; // [rsp+30h] [rbp+8h]

  if ( a2 )
  {
    _InterlockedOr(PROCESS, 0x40u);             // pending-attach flag
    _InterlockedOr(PROCESS, 0x10u);             // debugging-enabled flag
  }
  else
  {
    _InterlockedAnd(PROCESS, 0xFFFFFFEF);       // clear debugging-enabled
  }
  v3 = *(PROCESS + 144);
  v5 = SkiAttachProcess(PROCESS);
  *(v3 + 2) = a2 != 0;
  return SkiAttachProcess(v5);
}
```

אסטרטגיית הדיבוג של ה-Secure Kernel נשמרת ב-Image strategy configuration שלו. שינוי ישיר של הקונפיגורציה או הקובץ ייגרום לאימות החתימה של securekernel.exe להיכשל, מה שיוביל למסך כחול של המכונה הוירטואלית :(

אבל אם נוכל לשנות בזמן ריצה את הבדיקה ולדרוס את ענף ה-else ב-NOP-ים, זה מספיק כדי לתת ל-SkpsEnableDebugging לרוץ ללא תנאי. הטכניקה נשענת על שלוש פיסות מידע שאנחנו נצטרך למצוא ממכונת המארח:

1. ה-Guest Physical Address (GPA) של העמוד שמכיל את הבתים שנרצה לשנות, נמצא זאת בעזרת סריקה של הזיכרון הפיזי של האורח בשביל החתימה.
2. ה-Guest Virtual Address (GVA) וערך האוגר CR3 המתאים, אשר משוחזרים באמצעות לכידת ה-vCPU בעמוד וקריאת האוגרים שלו באמצעות HvGetVpRegisters.
3. המיפוי מ-GVA ל-GPA עבור המיקום של SkpsIsProcessDebuggingEnabled, אשר נגזר על ידי מעבר על טבלאות הדפים החל ממערך ה-CR3 ששוחזר.

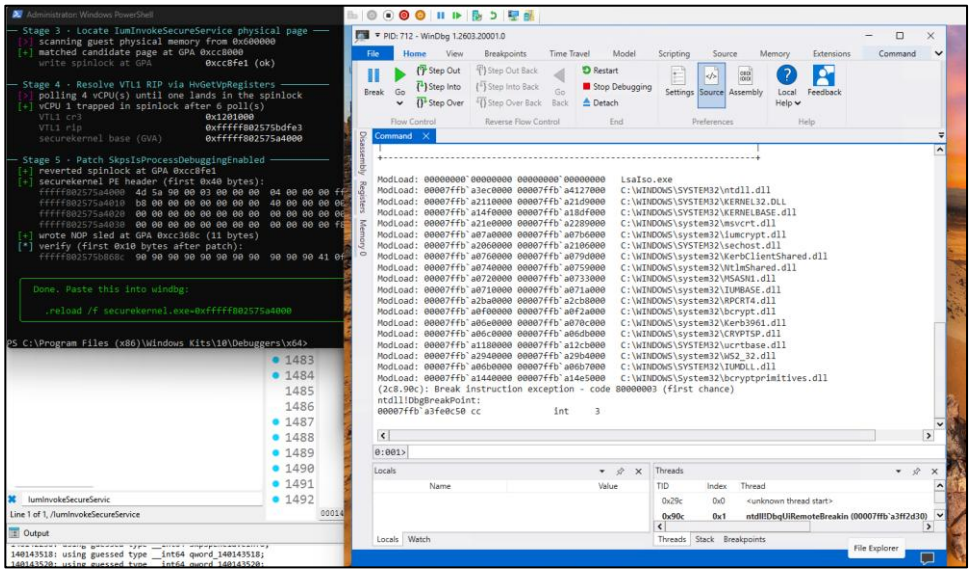
הפונקציה של securekernel.exe שמצאתי שמופעלת בתדירות הגבוהה ביותר, מה שמבטיח לנו שלא נחכה עד שהיא תקרא הייתה IumInvokeSecureService, שנחשבת לאחת הפונקציות הגדולות ביותר בקובץ ולכן אחת הקלות ביותר לזיהוי באמצעות חתימה. באמצעות SharpDisasm מצאתי את הוראת ה-ret שלה שמופיעה בסוף הפונקציה ולאחר מכן סרקתי את הזיכרון הפיזי של האורח בשביל תבנית הבייטים עד שמצאתי עמוד תואם שבו רשמתי חמישה בתים:

```
f3 90                   pause
eb fc                   jmp     0x0
c3                      ret
```

שינוי הזיכרון לא גורם למסך כחול במכונה הוירטואלית. בחיפוש בין כל ה-RIPs של המעבדים הוירטואלים בעזרת ה-hypercall HvGetVirtualProcessorRegisters. אם הבתים התחתונים של RIP תואמים ל-12 הבתים התחתונים של ההוראה הזו, אנחנו נשיג את ה-GVA וה-GPA המתאים, לאחר מכן נחשב את כתובת הבסיס של CR3 של ה-page table וה-page directory (גם דרך hypercall). בחיפוש המחרוזת "SkpsIsProcessDebuggingEnabled" בעזרת SharpDisasm, נוכל לקבל את הכתובת הרלטיבית של הפונקציה הסמויה, בהוספת כתובת רלטיבית זו לכתובת הבסיס ייתן לנו את ה-GVA של בלוק ה-if שאותו אנחנו רוצים לנטרל, ובמעבר על ה-page tables מ-CR3 יהפכו את זה ל-GPA. נרשום מספר NOP בכתובת הפיזית כדי לשכתב את מסלול ההרצה, ובכך להרשות ל-SkpsEnableDebugging לרוץ ללא בעיות, ולהרשות לנו לדבג את התהליך ב-IUM.

```
        // After the patch the 7-byte access-denied block is gone, so the
        // `else` is unreachable — both paths now fall into the if-body
        if ( v67 )
        {
LABEL_185:
          SkpsEnableDebugging(v574, a1[16]);
          inited = 0;                           // STATUS_SUCCESS
        }
        else
        {
          inited = 0xC0000022;                  // STATUS_ACCESS_DENIED
        }
```

עכשיו באותה מכונה עם אותם צעדים שניסינו בפעם הראשונה, נחבר את הדבגר WinDbg לתהליך Lsalso.exe נקבל את המראה הבא שמעיד על הצלחתינו!



הפרויקט מפורסם כפרויקט קוד פתוח בגיטהאב - https://github.com/ReverseWarrior/IUM-Debugger

אני תמיד שמח להצעות לבניית כלים ייחודיים כמו זה, ואולי אפילו להוסיף לו פיצ'רים חדשים :)

סיכום

עברנו על קונספטים מורכבים ובסיסיים כאחד בעולם הוירטואליזציה, מהבסיס של Hypervisor, עברנו לראות מקרוב את התוכנה Hyper-V שעל גביה מבוססת תעשייה שלמה של ענן. הבנו איך 2 עולמות, הרגיל והבטוח מתקשרים זה עם זה, ואילו קטעי קוד מעורבים בתהליך. לאחר מכן נכנסו לעולם של VTL 1 וראינו איך אפשר לשנות בזמן אמת את מערכת ההפעלה בכך שתתן לנו את הפיצ'רים שאנחנו רוצים.

בנוסף תוכלו למצוא את המאמר בבלוג שלי באנגלית reversewarrior.github.io

על המחבר

אני ליאל חנוכוב, חוקר חולשות בעל תשוקה לטכנולוגיות שמעורבות בחיי היום-יום שלנו כדי להפוך את העולם לבטוח יותר, תחומי המחקר שלי כוללים בין היתר אבטחה של מערכות הפעלה ומערכות האינטרנט של הדברים (IoT). אשמח לענות על שאלות ולקבל משוב! זמין ב-X ובלינקדאין :)

מקורות מידע

- :Microsoft Learn — Hypervisor Top-Level Functional Specification (TLFS) https://learn.microsoft.com/en-us/virtualization/hyper-v-on-windows/tlfs/tlfs
- Microsoft Learn — HV_PARTITION_PRIVILEGE_MASK: https://learn.microsoft.com/en-us/virtualization/hyper-v-on-windows/tlfs/datatypes/hv_partition_privilege_mask
- Microsoft Learn — VBS Enclaves: https://learn.microsoft.com/en-us/windows/win32/trusted-execution/vbs-enclaves
- Microsoft Learn — Enable Memory Integrity / VBS: https://learn.microsoft.com/en-us/windows/security/hardware-security/enable-virtualization-based-protection-of-code-integrity
- Saar Amar — First Steps in Hyper-V Research (MSRC blog): https://www.microsoft.com/en-us/msrc/blog/2018/12/first-steps-in-hyper-v-research
- Quarkslab — A Virtual Journey: From Hardware Virtualization to Hyper-V's Virtual Trust Levels: https://blog.quarkslab.com/a-virtual-journey-from-hardware-virtualization-to-hyper-vs-virtual-trust-levels.html
- Yarden Shafir — Secure Kernel Research with LiveCloudKd: https://windows-/internals.com/secure-kernel-research-with-livecloudkd
- gerhart01 — LiveCloudKd: https://github.com/gerhart01/LiveCloudKd
- tandasat — hvext (WinDbg extension): https://github.com/tandasat/hvext
- :ionescu007 — SimpleVisor (Intel VMX header vmx.h) https://github.com/ionescu007/SimpleVisor/blob/master/vmx.h
- justinstenning — SharpDisasm: https://github.com/justinstenning/SharpDisasm
- :ReverseWarrior — Hypervisors-Scripts (IDA 9.2 update) https://github.com/ReverseWarrior/Hypervisors-Scripts
- ReverseWarrior — IUM-Debugger: https://github.com/ReverseWarrior/IUM-Debugger
- .Windows Internals 7th Edition, Part 2 — Pavel Yosifovich, Mark E. Russinovich, David A Solomon, Alex Ionescu
- Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C: System Programming Guide