digital whisper Technologic Papers

Hyper-V Architecture for Security Researcher

מאת דני אודלר

הקדמה

בבסיס הנושא נמצאת הסוגיה איך להריץ שתי מערכות הפעלה נפרדות ומבודדות לחלוטין על מחשב ספציפי. כדי להתמודד עם סוגיה זו משתמשים בשכבת תוכנה, Hypervisor, אשר משמשת סוג של מערכת הפעלה רזה מאד עבור המחיצות (הסבר בהמשך) שנמצאות על המחשב. ה-Hypervisor נתמך חומרתית ע"י הרחבת המעבד (VTx - opcodes חדשים) ומסוגל להקצות ולנהל משאבי חומרה כגון זכרון/מעבד לכל מחיצה בנפרד ובכך להגיע לרמת בידוד מיטבית מבחינת אבטחת מידע.

הרחבות הוירטואליזציה של המעבד מאפשרות את ייחודיות הארכיטקטורה כאשר ניתן להזכיר גם את VT-d שבלעדיו המחיצות היו חשופות להתקפות DMA. לא נפרט כאן מה זה VT-d אבל בגדול זו יכולת להגביל רכיב חומרתי במחשב לכתוב דרך ה-PCIe לדפי זכרון שלא שייכים לו.

בנוסף חשוב לציין את נושא ה-Secure boot או SMM אשר מהווים את שורש המהימנות לפני עליית ה-Hypervisor ואם יש חולשה בהם אזי קשה יהיה לסמוך על השכבה מעליהם (מקווה במאמר אחר לפרט על נושאים אלה).

במאמר זה נתמקד ב-Hypervisor של מיקרוסופט (Hyper-V) אשר נמצא כתשתית בכל גרסאות windows האחרונות. ה-HyperV של מיקרוסופט (Hypervisor type 1) הינו ייחודי בכך שהוא חלק בלתי נפרד מתהליך עליית מערכת ההפעלה ואף נטען לפני הקרנל לעומת מימושים אחרים בהם ה-Hypervisor הוא רכיב שרץ כולו בקרנל.

כיום טכנולוגיות האבטחה החזקות ביותר של מיקרוסופט נשענות על וירטואליזציה אשר בבסיסה הפרדה קשיחה בין אזורי זכרון אשר מקשים מאד על משטח התקיפה. בין היתר ניתן למנות את כל עולם ה-VBS - Secure Kernel, HVCI, CredentialGuard, ApplicationGuard ועוד.

ארכיטקטורה

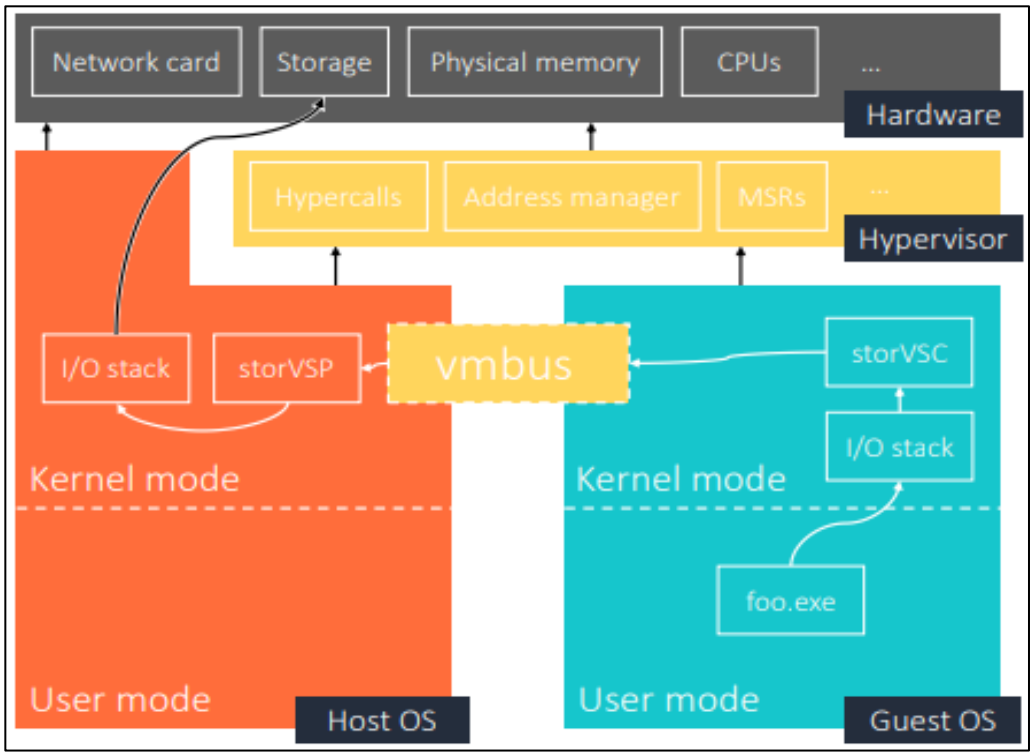
לפני שנתחיל לדבר על הנושא, חשוב שנכיר את המושגים בהם נשתמש במהלך המאמר:

- **מחיצה** - יחידה לוגית על המחשב אשר מיישמת מכונה וירטואלית מבודדת משאר המכונות.
- **מחיצת root** - מתייחס למערכת ההפעלה שמארחת את המכונות הוירטואליות, למעשה מערכת ההפעלה שעולה בעליה.
  - לעניין מאמר זה, **אורח** - מכונה וירטואלית אשר רצה במקביל למערכת ההפעלה המקורית.
  - לעניין מאמר זה, **מארח** - מערכת ההפעלה המקורית שמריצה מכונה וירטואלית אחת או יותר.

  מחיצה תכיל בין היתר זכרון, מעבד וירטואלי ורכיבים פריפריאלים וירטואלים נוספים. המחיצה המארחת מסוגלת לגשת למשאבי חומרה להבדיל ממחיצת אורח אשר לא נגישה לחומרה.
- **VMCall**- המחיצות מדברות עם ה-Hyper-V בעזרת ממשק HyperCall אשר נתמך במעבד (פקודת vmcall). ישנה רשימה די ארוכה של HyperCalls שונים אשר קרנל של מחיצה אורחת או מארחת יכולים לעשות בכדי לתקשר עם ה-Hyper-V, לדוגמא בקשה של מחיצה מארחת ליצור מכונה וירטואלית חדשה, להקצות זכרון עבור מכונה חדשה ועוד. ניתן להקביל את זה למנגנון ה- SystemCall במעבר מיוזר לקרנל. למחיצה מארחת יש יותר HyperCalls מאשר למחיצת אורח.
- **VMEnter** - מצב מעבד אשר קורה בכל פעם כאשר יש מעבר אל מחיצת האורח, בזמן זה נטען כל המידע הנחוץ לריצת המחיצה. המצב קורה בשני מיקרים עיקריים:
  1. בזמן VMLaunch - בפעם הראשונה שהמחיצה האורחת מתחילה לרוץ.
  2. בזמן VMResume- כל פעם נוספת בה המחיצה רצה לאחר ה-VMLaunch.
- **VMExit** - קורה למעשה בכל פעם כשריצת הקוד עוזבת מחיצה מסויימת. ההייפרויזור מכיל מימוש של VmExitHandler אשר בודק את סיבת העזיבה ומכיל מימוש לטיפול העזיבה(אם צריך) למשל אם היתה גישה לדף לא ממופה. גם סיבות עזיבה לגיטימיות(לא שגיאה) מטופלות במסגרת זו כמו טיפול בפסיקות, חריגות, גישה לאוגרי MSR, גישה לאוגרי CR ועוד מצבים שגורמים ליציאה.
- **VMCS** - מבנה של המעבד אשר משמש לשמירת נתונים בעת יציאה/מעבר בין שתי מחיצות. המבנה מכיל מספר אזורים קריטיים ביניהם:
  1. מצב הרגיסטרים (לא רק) של האורח, ישתמשו במידע זה בזמן VMEnter
  2. מצב הרגיסטרים (לא רק) של המארח, ישתמשו במידע זה בזמן VMExit
  3. דגלים אשר מסמנים מה לשמור ב-VmExit
  4. דגלים אשר מסמנים מה לשמור ב-VmEntrer
  5. מידע אשר נותן אינדיקציה מה גרם ל-VmExit למשל ניסיון קריאה של רגיסטר MSR.

מבנה ה-VMCS משמש כארגומנט עיקרי בפעולות מעבד מהותיות כגון VMLAUNCH, VMREAD, VMRESUME, VMWRITE. בנוסף מיבנה זה יכיל גם מצביע לטבלת ה-EPT שבה תשתמש המחיצה הרלוונטית (הסבר בהמשך).

תרשים כתיבה לקובץ במחיצה האורחת:



[כתיבה לקובץ, מקור: Hardening Hyper V Through Offensive Security Research]

תרשים זה ממחיש את המצב הבא:

תהליך user mode במחיצת האורח רוצה לגשת לקובץ שלו על ה"דיסק". הבקשה יורדת לקרנל של האורח בה מתחילה פונקציונליות של גישה לקובץ. מכיוון שהאורח לא יכול לגשת למשאבי חומרה, תהליך הגישה לקובץ יעבור לדרייבר שיסמלץ טיפול בבקשות לחומרה (storVSC). הבקשה תעבור מהאורח למארח דרך פרוטוקול VMBUS לדרייבר קרנלי במארח (storVSP). כעת הבקשה תעבור לדרייבר החומרתי הרגיל שיש לו אכן גישה לחומרה היעודית.

זכרון

כאשר מתייחסים לזכרון פיזי של VM למעשה לא מתייחסים לזכרון פיזי אמיתי מכיוון שה Hypervisor עצמו הוא זה שיחלק את המיפויים בעזרת EPT: Extended page table. למשל דף פיזי מסוים ב-VM ימופה לדף פיזי אמיתי ב-RAM דרך טבלת ה-EPT (הסבר בהמשך).

- **SPA** - system physical address, אלה הדפים הפיזיים ששייכים לזכרון ה-RAM, שייך ל-PFN DB.
- **GPA** - guest physical address, אלה הדפים הפיזיים שרואה מערכת ההפעלה שרצה במחיצה מסויימת.
- **GPADL** - דומה מאד ל-MDL בקרנל של וינדווס. בגדול זה הוא מיבנה אשר מתאר טווח מסוים של זיכרון פיזי ויודע לנעול אותו כדי למנוע את שחרורו או שינוי הרשאותיו. את המיבנה הזה אפשר

להעביר כפרמטר למארח כדי שהוא יבצע פעולות על זכרון זה. בהמשך נראה שימוש במבנה זה בזמן השמשת החולשה.

בידוד של מחיצה

1. **טבלת EPT** - Hyper-V מחזיק ומנהל טבלת דפים נוספת לטבלת הדפים הרגילה שמנהלת מערכת ההפעלה של האורח. הטבלה הזו לוקחת כל דף "פיזי" שמנוהל במחיצה ונותנת לה עוד שיכבת אבסטרקציה בדרך לכתובת הפיזית של הזכרון. כלומר יש לנו מיפוי כפול עבור כל כתובת וירטואלית אצל האורח, כאשר מי שקובע את המיפוי הסופי לדף הפיזי הוא ה-Hyper-V ולכן הוא זה שמסוגל לעשות בידוד בין דפים פיזיים של מחיצות שונות. ה-EPT למעשה מחזיק את המיפוי והרשאות הדף העדכניות והקובעות ולכן נוצר בידול אבטחתי מאד חזק בין שתי מחיצות. בתצורה זו תוקף נמצא ללא כל יכולת לשנות הרשאות של דף כך שיהיה בר הרצה.
2. בפועל מאחורי הקלעים כתובת וירטואלית באורח מתורגמת לכתובת "פיזית"(GPA) ע"י המנגנון הסטנדרטי בעזרת CR3 אשר מכיל כתובת GPA שמצביעה ל-PML4 כשבסוף מגיעים לכתובת GPA אשר מתורגמת בעזרת טבלת EPT לכתובת הפיזית האמיתית-SPA. שימוש ב-EPTP שזהו מצביע לטבלת EPT מתאפשר בזמן שליחת VMCS כאשר נכנסים למחיצת האורח. פירסור טבלת EPT דומה מאד לפירסור טבלת הדפים הרגילה של תהליך בזמן תרגום כתובת וירטואלית לפיזית.
3. מחיצת המארח מסוגלת לנטר או "לתפוס" גישות לחומרה כגון כתיבה לפורטים או MSR (נראה בהמשך שימוש בזה). זו יכולת מפתח עבור מחיצה המארחת(פריוילגיה הגבוהה), כך למשל בטכנולוגית הדגל של מיקרוסופט- VBS (Virtualization Based Security) הקרנל המאובטח (Secure Kernel) יכול "לנטר" שימוש זדוני ברגיסטרים מערכתיים כגון CR0, CR4 ועוד.
4. למעשה כל מחיצה איננה מסוגלת לגשת לזיכרון הפיזי של מחיצה אחרת וגם לא לזיכרון של ה-Hypervisor עצמו. להבדיל ממימושי VM אחרים אשר בהם ה-Hypervisor הוא חלק מהקרנל, במקרה שלנו ה- Hyper-V הוא למעשה קוד אשר רץ במרחב כתובות פיזי נפרד אשר מוקדש רק עבורו.

תהליך עליה עם Hyper-V

1. ה-BIOS מסיים את תפקידו ונותן למודול ה-WinLoad להחליט האם יש צורך בעליית Hyper-V וזאת בהתאם לקונפיגורציות משתמש.
2. אם ההייפרוויזור צריך לעלות, יטען מודול HvLoader אשר יבדוק קונפיגורציות שונות בנוגע להאם מעבד זה יכול להריץ הייפרוויזור ועוד בדיקות נוספות ל"מוכנות טעינה".
3. נטען מודול HVIX64 (מעבדי אינטל) אשר הינו רכיב ליבה באכיטקטורה של Hyper-V. בזמן זה עדיין לא מתחילה ריצת מודול זה, זהו למעשה לב ההייפרוויזור אשר רץ מתחת למערכת ההפעלה (HV Type1).
4. WinLoad ממשיך בקינפוג יכולות נוספות ואז מעביר את הריצה HVIX64.

5. בשלב הזה ההייפרוויסור יצור את המחיצה עם הפריוילגיה הגבוהה, root partition, ואחריה אם צריך מחיצה נוספת, באותה אנלוגיה למשל ב-VBS יווצר קודם VTL1 (עולם מאובטח) ואחריו VTL0 (פחות מאובטח).
6. לאחר יצירת המחיצה תתבצע פקודת מעבד VMLAUNCH אשר תתחיל את ריצת מחיצת ה-root, מתוך המחיצה כעת Winload חוזר לפעילות ויתחיל לעלות את קרנל מערכת ההפעלה.
7. הקרנל ממשיך את תפקידו ומעלה את שאר התהליכים הנחוצים.

ניהול זכרון ותהליכים

כל מחיצה תחזיק למעשה מעבד וירטואלי אשר אחראי על ריצת התהליכים. מכיוון שיכולה להיות יותר ממחיצה אחת ה-Hyper-V ינהל תרחישי context switch בין מעבדים של שתי מחיצות, פעולות אלה נתמכות ע"י מבנה VMCS אשר מכיל בין היתר סטטוס נכחי של המעבד, רגיסטרים ועוד.

כאשר נוצרת מחיצת **מארח** ההייפרוייסור יודע שהיא עם פריוילגיה גבוהה וממפה לתוכה דפי זכרון פיזיים תוך עדכון מתאים של ה-EPT. עם זאת דפים של ההייפרוויסור עצמו לא ימופו לתוך מחיצת המארח וזאת מסיבות אבטחה כדי שלא תהיה לשום מחיצה גישה ישירה אליהם. בטכנולוגיית VBS מחיצת המארח יכולה להחזיק עד ארבע טבלאות EPT וכך משיגה הפרדה עבור כל VTL חדש שיווצר. כיום יש VTL1 שהוא העולם המאובטח.

אתחול הזיכרון של מחיצת **האורח** יתבצע בשני שלבים עיקריים:

1. מחיצת המארח תקצה אוסף של בלוקים של זיכרון (מיוצגים ע"י MDL) ותחזיק אותם ברשימה מקושרת. תהליך זה דומה ב-Windows לבקשת הקצאת זכרון שהינו Reserved.
2. דרייבר VID (תאור בהמשך) במחיצת המארח ידאג לשלוח hypercall ל-HV עם רשימת הבלוקים של הזכרון וכתוצאה מכך ה-HV ימפה את הזכרון לתוך מחיצת האורח ויעדכן בהתאם את טבלת ה-EPT של האורח ה"חדש".

לסיכום, ה-HV יוצר מחיצה עם מרחב זכרון ריק, רק עם טבלת PML4 ריקה, בשלב זה אין זכרון פיזי למחיצה. לאחר מכן מחיצת המארח תבקש מ-HV (שליחת hypercall) ליצור מחיצת אורח חדשה, לאחר מכן ליצור מעבד וירטואלי ולמפות זכרון למחיצה.

משטח תקיפה

ה-Hyper-V עצמו לא מנהל לוגיקות משמעותיות שיכולות להוות בסיס תקיפה, הוא למעשה נותר בכוונה שכבה די רזה על מנת להקטין משטח תקיפה. הלוגיקות המשמעותיות שהוא מתחזק כוללות בין היתר קינפוג ה-EPT, קינפוג חומרה ספציפי, טיפול traps ו-intercepts עבור גישה לפונקציונליות שמצריכה הרשאות גבוהות, טיפול ב-HyperCall שזה למעשה שווה ערך ל-SystemCall שמתבצע מהמחיצה.

מחיצת המארח הינה למעשה היחידה שלא באמת מבודדת לגמרי והיא רצה על המערכת ההפעלה מארחת. מכיוון והיא היחידה שניגשת לחומרה היא למעשה תהווה מתווך בין המחיצות האחרות למשאבי החומרה. אפשר להגיד די בביטחה שרב משטח התקיפה בהיבט של אורח למארח נמצא במחיצת המארח כיוון שהיא מנהלת המון פונקציונליות עבור המחיצות האורחות.

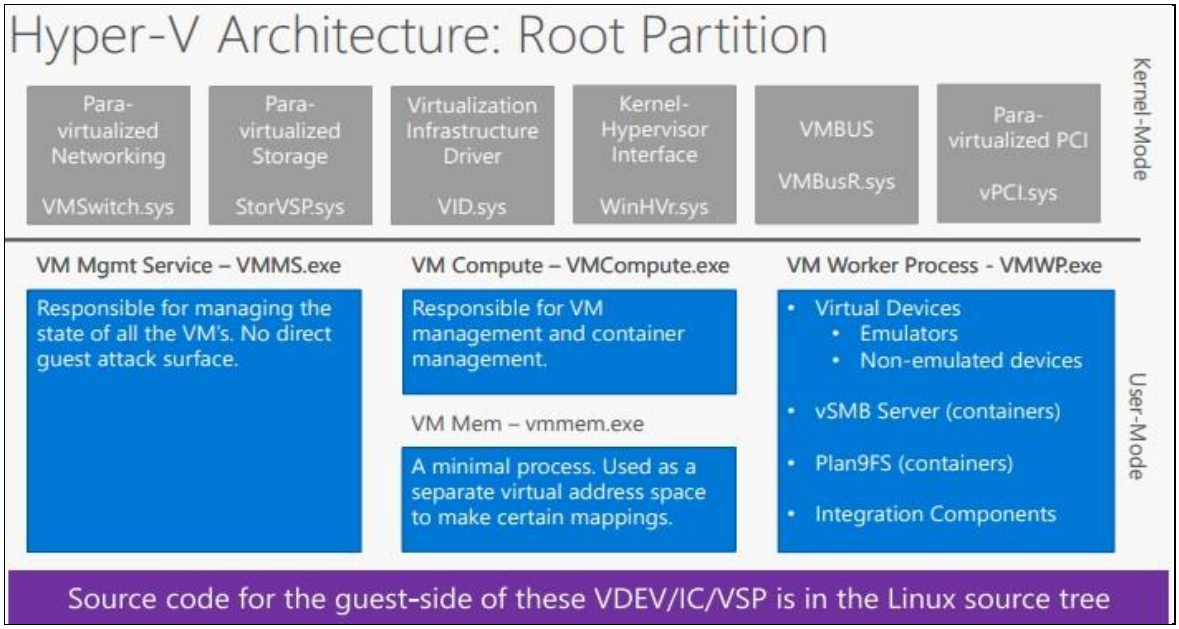
חוליה חלשה - רכיבי המארח

אם נסתכל על איזה רכיבים האורח יכול לתקוף אצל המארח אפשר לדון ברכיבים הבאים:

1. **Virtual device** - זה בדרך כלל התקן מסומלץ (emulated) או ParaVirtualized אשר נטען ל- user mode במחיצה המארחת.
2. **VSP** - virtualization service provider, התקן Para-Virtualized, נטען ל-Kernel mode במחיצה המארחת. הוא מקושר או למעשה מייצג רכיב Virtual device מסעיף 1.
3. **Integration Component** - מבחינת מבט של חוקר סעיף זה דומה מאד לסעיף 1 כאשר האורח יכול לדבר דרך ערוץ תקשורת מול רכיבים מסוג זה.

כמו שכבר הזכרתי למחיצות האורחות אין גישה להתקני חומרה על כן המחיצה המארחת חייבת לספק שירותים מהותיים כגון קריאה/כתיבה למקור אחסון, תקשורת ועוד, אחרת פשוט האורח לא יוכל כלל לתפקד. סעיף 1 מרכיבי המארח נותן פתרון בסיסי לנושאים אלה כאשר הוא מסמלץ רכיב תקשורת, גרפיקה וכו. הבעיה במיקרה זה היא שהתהליכים הינם איטיים מאד כאשר מנסים לסמלץ חומרה ואי אפשר להתפשר על מהירות ביצוע כאשר מנסים לבצע תקשורת או כתיבה של קובץ. כדי להתמודד עם זה משתמשים בסעיף מספר 2 רכיבים Paravirtualized. הרכיבים העיקריים בקטגוריה זו הם: תקשורת, גרפיקה, אחסון ו-PCI.

כמו כן ב-VM דור שתיים יש הרבה פחות רכיבים מסומלצים והרבה יותר רכיבים ParaVirtualized ביחס ל-VM דור אחד.



[רכיבי מחיצה מארחת, מקור: BlackHatUSA - A Dive in to Hyper-V Architecture and Vulnerabilities]

המאמץ בארכיטקטורה הינו לשים כמה שיותר רכיבים ב user mode אלא אם חייבים הרשאות ריצה קרנל או שיקולי יעילות.

User mode מחיצה מארחת

- VMMS.exe מודול (management service) זה אחראי בין היתר על ניהול המצבים של המכונה הוירטואלית ואחראי על יצירת תהליך ה-VMWP. מודול זה לא מתקשר עם המחיצה האורחת.
- VMWP.exe - זהו תהליך (worker process) אשר נוצר עבור כל מחיצת VM. כך למשל קריסה של תהליך זה תשפיע רק על המחיצה הספציפית שמקושרת לתהליך זה. בתהליך זה תימצא פונקציונליות אשר לא רצה במסגרת הקרנל או ה-HV כך למשל רכיבי סעיף 1 וסעיף 3 ימצאו בתהליך זה. לדוגמה, אם רכיב סעיף 1 רוצה לבקש ניטור של גישה לפורט הוא יבקש ממודול ב-VMWP.exe שיבצע זאת, כתוצאה מכך תתבצע קריאה ל-VID.sys (ראה בהמשך) וכתוצאה מכך ירשם callback אשר יקרא בהינתן גישה לפורט זה. מודול זה מדבר הרבה עם הדרייבר vid.sys.
- VMCompute.exe - אחראי על חלק מתהליכי הניהול של קונטיינר ומכונה הוירטואלית.

Kernel mode מחיצה מארחת

- VMSwitch.sys - מאפשר תקשורת מהירה עבור המחיצות המתארחות.
- StorVSP.sys - מרכז דרכו את נושא האחסון. בקשות גישה למקור אחסון אשר מגיעות מהאורח מגיעות לדרייבר זה.
- vPcivsp.sys - משמש עבור שרותי PCI בסיסיים עבור רכיב PCI, למשל למפות פונקציונליות של רכיב PCI לתוך המחיצה האורחת.
- VMBusr.sys - דרייבר הממש ערוץ תקשורת מהיר בין אורח למארח.
- WinHypervisorr.sys - ממשק אשר ברובו עוטף פונקציות HyperCall ומאפשר לדבר עם ה-Hypervisor מהמחיצה המארחת.
- VID.sys- דרייבר עיקרי בארכיטקטורה אשר מתקשר (בעזרת HyperCalls) עם ה-Hypervisor לצרכים מהותיים כגון יצירת VM חדש , הקצאת זכרון עבור יצירת VM חדש, הרשמה של רכיבי יוזר לניטור גישה לפורטים מסויימים ועוד.

טבלת קישור קרנל אורח לקרנל מארח:

	ROOT PARTITION KERNEL	GUEST PARTITION KERNEL
Storage	storvsp.sys	storvsc.sys
Networking	vmswitch.sys	netvsc.sys
Synth 3D Video	synth3dVsp.sys	synth3dVsc.sys
PCI	vpcivsp.sys	vpci.sys

[התאמת רכיבי קרנל כמשטח תקיפה הרצת קוד אורח למארח, מקור: First Steps in Hyper-V Research]

משטח ערוצי תקשורת אורח למארח

תקשורת עם Hypervisor

- HyperCall - מחיצת האורח יכולה לשלוח קריאות מסוג זה ל-Hypervisor, עם הקריאה הזו אפשר להעביר פרמטרים דרך רגיסטרים או לצרף GPADL אשר יכיל תאור מבנה זיכרון שבו יועברו פרמטרים.
- המחיצה האורחת יכולה להשתמש בקבוצה מצומצמת של HyperCalls, למשל TLB Flush ועוד. בגדול משטח המחקר באזור זה די סגור על עצמו וכנראה לא כאן אפשר למצוא דברים פיקנטיים.
- Faults - דף שלא ממופה ב-EPT יחזיר page fault אשר יקרא למנהל החריגות ב-Hypervisor שיטפל בשגיאה. בדרך זו ממומשות גישות MMIO ב-Hypervisor - ה-Hypervisor יכול להבין דרך ה-EPT שמנסים לגשת לדף פיזי שהוא מסומלץ כ-MMIO ולכן כשהוא יטפל ב-fault הזה הוא פשוט יעביר את הבקשה ל-VID.sys כדי שיסיים את העבודה.
- Emulated Instructions - למשל ניסיון גישה לפורטים או פקודות CPUID ינוטרו ע"י ה-Hypervisor והבקשה תעבור הלאה אם זה ב-Hypervisor עצמו או לרכיבי קרנלי כמו ה-VID.sys. כנל אפשר לנטר גישה לרגיסטרים מערכתיים כמו CR4 שאחראי בין היתר על ביטול SMEP.

תקשורת עם קרנל המארח

- VMBUS - ערוץ התקשורת החשוב ביותר בין אורח למארח. מקושר לכל רכיבי הארכיטקטורה בקרנל של המארח. רכיב זה נגיש דרך Kernel Mode Client Library.
- Extended HyperCalls - קריאות אשר ה-Hypervisor מעביר ישירות ל-VID.sys.
- Intercept Handling - כמו שכבר הזכרתי ה-Hypervisor ינטר למשל גישה לפורטים או טיפול בדף לא ממופה או דף עם הרשאות לא מתאימות ויעביר את הבקשה הלאה ל-VID, כאשר ה-VID יחליט מה לעשות עם בקשה זו למשל לסרב גישה או דחייה, למעשה סוג של מכונת קבלת החלטות אשר מתבצעת בקרנל המארח כתוצאה מפעולות האורח.

תקשורת עם User Mode המארח

ברכיבים אלה מתבצעת כל הסימולציה של ההתקנים השונים.

- IO Ports - מודולי user יכולים להירשם לניטור של גישת האורח לפורטים חומרתיים וכתוצאה מכך לסמלץ פונקציונליות חומרתית עבור הבקשה שהתקבלה.
- MMIO - מודולי user יכולים להירשם לניטור גישות לזיכרון פיזי של האורח, זכרון GPA. כך כפי שהוסבר מקודם אפשר לממש גישות MMIO של האורח.
- VMBUS - ערוץ תקשורת מהיר אשר מתפקד ע"י pipes או sockets.

- Aperture - יכולת למפות דף פיזי של אורח (GPA) לתוך זיכרון וירטואלי של VMWP.exe. דבר כזה יאפשר לקרוא/לכתוב לזכרון של האורח, ועל כן נפתח פה חלון בעייתי של סנכרון כי המארח יכול לקרוא מהזכרון ובמקביל האורח יכול לכתוב לזכרון זה.

VMBUS

כמו שכבר צוין כמה פעמים זה הוא פרוטוקל תקשורת מהיר והעיקרי בהחלפת הודעות בין אורח למארח.

הפרוטוקול מממש הודעות שונות (נראה את חלקן במימוש החולשה בהמשך) ושולח הודעות דרך זכרון משותף אשר ממומש כ-Buffer סיקלי (מעגלי). המחיצה המארחת שולחת בקשת הצעה להקמת ערוץ תקשורת, כאשר האורח מאשר את הבקשה הוא שולח את כתובת הזכרון (GPA) שתשמש לתקשורת המשותפת וכתוצאה מכך המארח ימפה כתובות אלה לתוך הזכרון הוירטואלי של עצמו.

כעת שתי המחיצות מסונכרנות על אותו דף פיזי, SPA. בכדי לשלוח הודעה, האורח יכתוב לזכרון שלו ויאותת למארח שיש הודעה ממתינה עבורו, בשלב זה המארח יוכל לקרוא את ההודעה ולטפל בה. הרכיבים השונים בארכיטקטורה לא מדברים "ישירות" על ה-VMBus ולמעשה מתקשרים דרך 3 שכבות אבסטרקציה עיקריות:

1. Kernel mode client library KMCL - משמש עבור VSPs שונים כגון VMSwitch, StorVSP, vPCI אשר נמצאים במרחב הקרנלי. שיכבה זו מבוססת על callbacks רבים אשר נרשמים כאשר יוצרים VMBus ועל כן בהינתן וקרה event מסוים יוטרג callback אשר מטפל ב-event זה. אירוע כזה יכול למשל להיות ארוע איתות "שהגיע הודעה חדשה" ובמקרה זה ה- callback הרלוונטי יגש לזכרון המשותף שהוקצה לשימוש ה-VMBus ויעתיק משם את ההודעה שהגיע לזכרון קרנלי פרטי במארח לפני שהודעה זו תעבור ל-KMCL. הסיבה העיקרית להעתקה זו היא למנוע חלון של time of check time of use שיכול לקרות כאשר מדובר בזכרון משותף.

   במקרה ספציפי כאשר להודעת VMBus מצורף GPADL, יתבצע מיפוי הזכרון לתוך המארח, ולא תתבצע העתקה כדי למנוע time of check time of use, ולכן במיקרה זה המארח יתחיל לטפל בהודעה שהזכרון שלה זמין לכתיבה/קריאה הן באורח והן במארח. נדון בסוגיה זו בתאור החולשה בהמשך.

2. VMBus pipes - משמש עבור רכיבי user mode כגון VMMS.exe, VMCompute.exe. השיטה דומה למקודם, המארח יוזם עם האורח את התחלת ההתקשרות והאורח בתגובה מחזיר pipe סטנדרטי של וינדווס אשר מהווה את בסיס התקשורת. פעולות הכתיבה ל-pipe יכולות להיות לא סינכרוניות כאשר במיקרה זה יקרא סוג של IOCompletionRoutine אשר יטפל בהודעה.
3. VMBus sockets - כמו שכבר הזכרתי אפשר להירשם לניטור (קבלת event) עבור גישה לפורט או מרחב MMIO שנמצא בכתובות של האורח(GPA). הפונקציות שיטפלו בכך הן NotifyIoPortWrite ו-NotifyMMIOWrite אשר אפשר למצוא ייחוס אליהן ב-VMWorker Process.

מאמר מצוין על שימוש VMBus לטובת מחקר חולשות אפשר למצוא במקורות (6).

נתחיל מהסוף: מיקרוסופט מציעה היום קרוב למליון ש"ח עבור מציאת פרצת הרצת קוד מאורח למארח.

בבסיסו של מחקר זה נמצאת ההנחה שיש הפרדה מוחלטת בין מערכת ההפעלה המארחת לבין מערכת ההפעלה האורחת, שזה למעשה לב ליבה של וירטואליזציה. ההפרדה ביניהם היא security boundary יציב וקשיח אשר ממומש הן חומרתית והן תכנתית. תחשבו לרגע על שרת באמזון שמריץ עשרות מכונות וירטואליות של לקוחות שונים, ברור שכל גישה של מכונה וירטואלית לזכרון של מכונה וירטואלית אחרת שרצה על אותו השרת תחשב כבעיית אבטחה חמורה וכמובן גישה מתוך המכונה הוירטואלית לשרת עצמו תהיה בעיה אולי עוד יותר רצינית.

VMSWITCH

כמו שתואר מקודם זה הוא רכיב קרנלי אצל המארח אשר מהווה חלק חשוב בארכיטקטורה ותפקידו לאפשר תקשורת נתונים בעיקר ע"י סימולציה של כרטיס רשת בעזרת פרוטוקול RNDIS.

תהליך איתחול הפרוטוקול בין האורח למארח מתחיל בסדרת הודעות VMBUS אשר מסכימות על הגדרות גירסה וכו' ולאחר מכן האורח יקצה Buffer עבור שליחה ו-Buffer עבור קבלה וישלח למארח הודעה עם צירוף GPADL המתאר את שני הבאפרים שהוקצו. המארח ימפה את ה-GPADL אליו וכך יווצר טווח של זכרון משותף. באפרים אלה הם בנוסף ל-Buffer הסיקלי של ה-VMBUS והם ישמשו לצורכי פרוטוקול התקשורת אשר כולל הודעות RNDIS בקצב גבוה.

כאשר באפרים אלה ממופים אצל המארח מה שקורה הוא שהם מחולקים(חד פעמי) לתתי באפרים, כאשר כל תת Buffer יחזיק את הגבולות שלו - התחלה וסוף, כתוצאה מכך המארח יעדכן את האורח איך הוא חילק את הבאפרים כדי שהמארח יוכל לשים את החבילות בתת Buffer ספציפי שהוא יבחר. האורח מעתיק חבילת RNDIS ל-Buffer השליחה ומיד שולח למארח הודעת VMBUS שתיידע אותו שיש לו חבילה. VMSWITCH יקח חבילה זו ויתחיל לטפל בה לצורכי תקשורת ואם יש לו להחזיר חבילה הוא ישים אותה ב-Buffer הקבלה (מנקודת מבט של האורח) וישלח לו הודעת VMBUS שיש עבורו חבילה.

טיפול בהודעות RNDIS

VMSWITCH ישתמש בכמה threads על מנת לבצע את הטיפול בהודעות. ברמת ה-thread הבודד הפעולות הן די סיריאליות והעיקריות הן - ברגע שיש הודעת חיווי דרך ה-VMBUS שיש חבילה ממתינה יקרא callback שיתחיל לטפל בהודעת RNDIS וכתוצאה ישלח, ב-Buffer הקבלה, חיווי לאורח שהחבילות בטיפול.

תהליך אתחול לא עיקבי

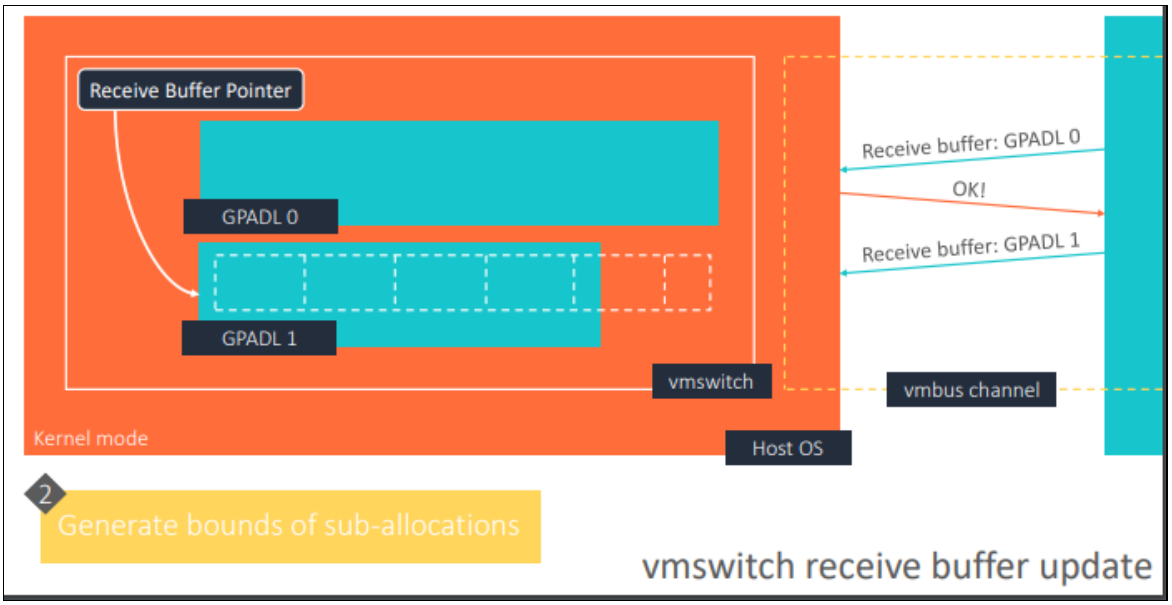
בתהליך איתחול תמיד מעניין לראות מה יקרה אם סדר האתחול ישתבש בצורה כלשהיא שלא ציפו לה והאם אפשר להשתמש עם זה לטובתנו.

כמו שכבר אמרנו האורח מקצה טווח זכרון ושולח את ה-GPADL שלו למארח. מחקירה של הקוד עולה שאין מניעה לשלוח למארח שוב הודעת אתחול עם GPADL אחר, במיקרה כזה המארח יפסיק להשתמש ב-Buffer הקודם וישתמש ב-Buffer החדש, אפשר לחזור על פעולה זו אף יותר מפעם אחת. כרגע נוצר סוג של זכרון ממופה שלא שוחרר אבל אין הרבה מה לעשות עם הנתון הזה.

שלושה שלבים בתהליך האתחול:

1. בהינתן והמארח קיבל Buffer חדש לעבוד איתו, הוא יבצע החלפה של מצביע מה-Buffer הקודם כך שיצביע ל-Buffer הנכחי.
2. ה-Buffer החדש יחולק לתתי באפרים עם גבולות של סוף והתחלה.
3. החלוקה תירשם במבנה אשר ישמש לצרכי הטיפול השוטף ב-Buffer (שליחה/קריאת הודעה מתת באפר)

שלושת הפעולות האלה הן בוודאי לא אטומיות ומתבצעות ללא מנגנון סנכרון/נעילה כלשהו. החולשה כאן נמצאת במצב בו האורח מציע Buffer חדש וכתוצאה מזה המארח מעדכן כך שהמצביע ל-Buffer יתייחס כעת ל-Buffer החדש אך במקביל טרם עודכנו גבולות תתי הבאפרים שמחזיק המארח.



[החלפת Buffer, מקור: Hardening Hyper V Through Offensive Security Research]

כדי לנצל חולשה זו צריך לבקש מהאורח להציע למארח Buffer קבלה חדש קטן יותר, לאחר מכן לגרום להיכנס לתוך חלון ההזדמנויות הצר בין שלב 1 לשלב 2 כך שבדיוק בנקודת זמן זו תתבצע כתיבה ל-Buffer הקבלה החדש. במצב זה תתבצע כתיבה מחוץ לגבולות ה-Buffer (ה-Buffer החדש קטן יותר).

נשמע פשוט, אבל לא ממש, צריך שליטה מלאה ב:

1. המידע שרוצים שיכתב מחוץ לבאפר.
2. להגיע לחלון ההזדמנויות שתואר.
3. מה אני הולך לדרוס מחוץ לבאפר? לדרוס סתם מידע לא יתן לי כלום, צריך שיהיה שם משהו פיקנטי.

המידע שרוצים שיכתב מחוץ לבאפר

צריך לשים לב שכתיבות ל-Buffer הקבלה מתבצעות כהודעות תגובה של המארח, כלומר אי אפשר לשלוח משהוא מהאורח כדי שידרוס את Buffer הקבלה. מהינדוס לאחור של הקוד עלה שאפשר לשלוח הודעות מסוימות אשר יגרמו למארח לשלוח הודעות תגובה שאפשר לשלוט על תוכנן (טוב לנו). הודעות RNDIS_QUERY_MSG נשלחת מהאורח עם ערכים בשליטתנו והודעת החזרה מכילה את אותם הערכים, כסוג של "הודעת אישור". אז יש לנו כעת אפשרות לשלוט בערכים שיכתבו ל-Buffer הקבלה.

להגיע לחלון ההזדמנויות

במבט ראשון אפשר ללכת לכיוון של לשלוח מהאורח המון הודעות אחת אחרי השניה וכתגובה יכתבו המון הודעות ל-Buffer הקבלה כך שמתישהו יהיה אפשר להיכנס בתוך חלון ההזדמנויות של החלפת באפר. שיטה זו לא תעבוד מכיוון שהמארח שולח הודעת סנכרון לאורח לאחר כל כתיבה ל-Buffer ומחכה לאישור האורח על קבלת הודעת הסנכרון. במצה כזה יהיה בלתי אפשרי להגיע לחלון הזדמנויות.

רעיון אחר שעלה הוא האם אפשר לגרום להשהייה בין שליחת הודעה מהאורח לבין שליחת הודעה חזרה של המארח (לתוך Buffer הקבלה)? אם נצליח לגרום להשהייה כזו נוכל בסבירות גבוהה לפתוח את חלון ההזדמנויות כך שנחליף Buffer קבלה ישן בחדש כאשר עדיין לא מתבצעות כתיבות ל-Buffer הקבלה. מחקירת הקוד עלה שאם שולחים הודעות RNDIS_KEEPALIVE_MSG עם גודל שלא מתאים לגדול המוצהר בהודעה, יקרה מצב בו ה- thread ב-VMSWITCH שמטפל בהודעות מתקבלות ידחה את ההודעה ללא טיפול בה וימשיך בטיפול ההודעה הבאה, כל זאת ללא שום כתיבה ל-Buffer הקבלה.

מכאן וקטור הפעולה יהיה כלהלן: האורח ישלח מספר הודעות "לא תקינות" (RNDIS_KEEPALIVE_MSG) ולאחר מכן ישלח הודעה תקינה (RNDIS_QUERY_MSG) . במצב כזה כל ההודעות הלא תקינות ידחו ויגרמו להשהייה בכתיבה ל-Buffer הקבלה, כאשר ההודעה האחרונה תטופל והתוצאה שלה (תוכן בשליטתנו) תיכתב ל-Buffer הקבלה.

ברגע שיש לנו יכולת לשלוט בהשהיית כתיבה ל-Buffer הקבלה, האורח יכול לשלוח למארח Buffer קבלה חדש (קטן יותר) ולגרום לכתיבה מחוץ לגבולות ה-Buffer בעזרת חלון ההזדמנויות שיווצר. אפשר למשל לבחור לשלוח 20 הודעות לא תקינות כאשר במקביל שולחים בקשה להחלפת באפר. אם במיקרה זה ההודעה תכתב בתוך ה-Buffer החדש אזי ההשהייה גדולה מידי וצריך להקטין את כמות ההודעות הלא תקינות. אם ההודעה תכתב בתוך ה-Buffer הישן אזי ההשהייה קטנה מידי וצריך להגדיל את כמות ההודעות הלא תקינות.

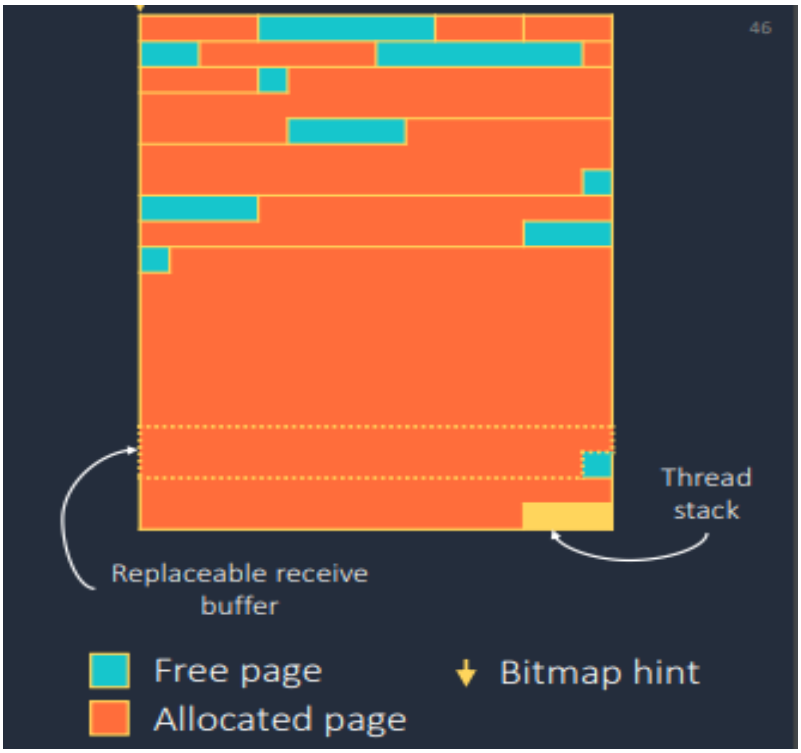
מה אני הולך לדרוס מחוץ לבאפר?

המטרה כעת היא לדרוס איזשהו מיבנה מעניין וכתוצאה מכך נוכל לגרום להרצת קוד. מהמשך המחקר עלה שמיפוי GPADLs לתוך הקרנל והקצאות מחסנית של thread חדש נמצאים באזור זכרון המוגדר System PTE.

הרעיון כעת הוא לרסס את אזור הזיכרון הקרנלי עם GPADLs כך שנסחוט את כל ההקצאות החופשיות באזור זכרון זה ונגיע להקצאות דטרמיניסטיות בדף זיכרון חדש. הריסוס צריך להיות בסדר מסויים ובגדלים מסויימים כדי להשיג את המטרה (לא טריויאלי). בשביל לקבל אפקט כזה נקצה הרבה GPADLs כ-Buffer קבלה(האורח שולט ביכולת ההקצאה ובגודל הבאפר) כך שנסחט את ה-SystemPTE allocator.

במצב זה כל הקצאה נוספת לאזור זה תתרחש בוודאות אחרי הקצאת Buffer הקבלה. מה נוכל להקצות שם?

כמו שכבר נרמז, יש לנו כנראה יכולת להקצות מחסנית חדשה במיקום הזה. הבעיה כאן היא איך לגרום למארח להפעיל thread חדש כדי שתוקצה המחסנית? באותה גירסה היה באג ב-VmSwitch שאיפשר הקצאת thread חדש כי היתה בעיה עם שחרור של Worker Threads שטיפלו בהודעות נכנסות. לכן בשלב מסויים היה אפשר ליצור תהליכון חדש כתלות בהודעות נכנסות. במצב כזה המחסנית תוקצה לאחר Buffer הקבלה ופה זה כבר מתחיל להיות מעניין. להלן מחשה למצב שיווצר:



[מצב זכרון רצוי, מקור: מקור: Hardening Hyper V Through Offensive Security Research]

הבעיה כרגע היא שניתן לדרוס כתובת חזרה במחסנית הרלוונטית אבל ידידנו KASLR די כובל אותנו ולא נדע לאיפה לקפוץ מהכתובת חזרה שנדרוס.

צריך איכשהו להבין איפה אנחנו רצים ולאיזה כתובת מעניינת אפשר לקפוץ. מחקירת הקוד עולה שהודעת nvsp_message שנשלחת לאורח מכילה מבנה עם שדה header ושדה union שמתאים לתכן בגודל משתנה.

במיקרים מסויימים בהודעה שתשלח לאורח, ה-union יכיל רק 8 בטים של מידע מאותחל אבל המיבנה כולו שחוזר כולל את כל ה-union שהוא 40 בטים. במיקרה כזה אנחנו מצליחים להדליף 32 בטים של זכרון מחסנית קרנלי של VMSWITCH.

במידע מודלף זה יש בין היתר כתובת שמצביעה לאיפה נטען המודול של VMSWITCH. כתוצאה מכך ניתן להבין לאיזה כתובות נטען המודול ומרגע זה אפשר להסתמך על כתובות יחסיות כדי למצוא שרשרת ROP.

כעת אפשר לבנות שרשרת ROP אשר תתחיל מדריסת כתובת החזרה של המחסנית שנוצרה בסעיף הקודם.

סיכום

לסיכום ראינו כאן יכולת מוחשית להגיע להרצת קוד מצד האורח לתוך הצד המארח. נעשתה כאן הרבה עבודה של חקירת קוד, הנדסה לאחור והבנת הארכיטקטורה הפנימית של מנגנוני ה-Hyper-V. כל החולשות האלה נסגרו ואף חל שינוי בתכן ההקצאה כך שהקצאות GPADL לא יהיו באותו אזור של הקצאות המחסנית.

אין ספק כי טכנולוגיות האבטחה על בסיס וירטואליזציה של מיקרוסופט מציבות רף חדש במשחק המגן תוקף.

תודה מיוחדת לסער עמר, @amarsaar, מ-MSRC-IL אשר נתן את דגשיו למאמר.

אשמח לשאלות, הערות והארות דרך המייל: danny.odler@gmail.com

מקורות

- https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2018_08_BlackHatUSA/A%20Dive%20in%20to%20Hyper-V%20Architecture%20and%20Vulnerabilities.pdf, **Joe Bialek, Nicola Joeli**
- https://i.blackhat.com/us-18/Thu-August-9/us-18-Rabet-Hardening-Hyper-V-Through-Offensive-Security-Research.pdf, **Jordan Rabet**
- https://msrc-blog.microsoft.com/2018/12/10/first-steps-in-hyper-v-research/, **Saar Amar**
- https://blog.amossys.fr/virtualization-based-security-part1.html
- https://github.com/saaramar/Publications/blob/master/BluehatIL_VBS_meetup/VBS_Internals.pdf
- https://msrc-blog.microsoft.com/2019/01/28/fuzzing-para-virtualized-devices-in-hyper-v/