The New Processes of Windows

מאת רז עומריי

הקדמה

עם יציאתה של Windows 10, נוספו מספר פיצ'רים חדשים ומשמעותיים למערכת ההפעלה. חלק מפיצ'רים אלו מתבססים על סוג חדש של תהליכים הקיים ב-Windows, השונה מהדרך שבה עובדים התהליכים שהיו עד כה, ה-NT Processes. לתהליכים מסוג זה קוראים Minimal & Pico Processes.

Pico Processes הינם סוג חדש של תהליכים ל-Windows, שנולד כחלק מפרוייקט שבוצע על ידי ה-MSR (Micorsoft Research): Drawbridge. מטרת הפרוייקט הייתה להריץ יישומים בסביבה מבודדת, ללא תלות במערכת ההפעלה (למשל, להריץ יישומים של Windows XP על גבי Windows 7/8/10). לרוב, ריצה של יישומים מסוג זה הייתה מתבצעת על מכונה וירטואלית (VM). אך, המטרה של ה-MSR הייתה להריץ את היישומים הללו על גבי תהליך של מערכת ההפעלה המארחת (ה-host OS). ומכאן, נולד הצורך בסוג חדש של תהליכים למערכת ההפעלה: ה-Pico Processes, שמתבסס על סוג נוסף של תהליכים הנקרא Minimal Processes, כפי שנראה בהמשך.

מאמר זה יסקור מהם סוגי התהליכים החדשים שקיימים ב-Windows, ובמה הם שונים מהתהליכים שהיו קיימים עד כה. בנוסף, במאמר יוצגו מספר שימושים שונים לסוגי התהליכים שנוצרו: ה-Memory Compression, קונספט חדש על מנת להשתמש ביותר זיכרון ולהפחית את הכתיבה לדיסק, וה-WSL (Windows Subsystem for Linux), שמאפשר הרצה של פקודות והרצה של קבצי ELF מתוך Windows.

אך לפני שנתחיל להסביר על תהליכים ושימושים הללו נדון בתהליכים הנורמטיביים שיש ב-Windows.

מהו תהליך?

תהליך הינו מופע שאחראי להכלה של קבוצת משאבים שמשתמשים בהם בעת הרצה של תוכנית מסוימת (כאשר תוכנית היא רצף של פקודות להרצה). למעשה, תהליך מכיל מופע של התכנית ועוד משאבים שנחוצים לריצת התכנית, לדוגמה DLLs (Dynamic Link Libraries).

כל תהליך נורמטיבי בWindows, כלומר NT Process, מכיל את הדברים הבאים:

- מרחב כתובות וירטואלי פרטי ב-user-mode - קטע בזיכרון שהתהליך יכול להשתמש בו.
- קובץ הרצה - מכיל את הקוד ההתחלתי להרצה, וממופה לתוך מרחב הכתובת של התהליך.
- טבלת של handles - טבלה המכילה גישה למשאבים ואובייקטים שמוחזקים על ידי, וזמינים לכל ה-thread-ים של התהליך (למשל, קבצים).
- Security Context - זהו סימן של כל תהליך שנועד לבדיקה בעת גישה למשאבים משותפים (האם לתהליך יש מספיק הרשאות לגשת למשאב מסויים או לא).
- Threads - לכל תהליך חייב להיות thread אחד לפחות על מנת להריץ את הקוד. יש לשים לב כי מה שמריץ את הקוד הוא ה-thread ולא התהליך עצמו.

בעת יצירת תהליך ב-Windows, בנוסף ליצירה של הדברים לעיל, מתחוללים מספר תהליכים נוספים:

- יצירת ה-EPROCESS וה-PEB - אלו אובייקטים שמייצגים את התהליך ומכילים מידע על התהליך. ה-EPROCESS (Executive Process Object) הינו אובייקט שמייצג את התהליך ב-kernel, נמצא במרחב הכתובות של ה-kernel ומוחזק על ידי ה-Object Manager. לעומת זאת, ה-PEB (Process Enviorment Block) הינו מבנה נתונים המכיל מידע על התהליך במרחב הכתובות של התהליך עצמו (ב-user mode).
- יצירת ה-ETHREAD וה-TEB - אלו אובייקטים המתארים את מצבו הנוכחי של thread מסוים ומכילים מידע על thread מסוים בתהליך. בדומה ל-EPROCESS, ה-ETHREAD (Thread Object Executive) נמצא גם הוא במרחב הכתובות של הקרנל ומוחזק על ידי ה-Object Manager, בעוד שה-TEB (Thread Environment Block) נמצא בתוך מרחב הכתובת של התהליך עצמו (ב-user mode).
- יצירת ה-KUSER_SHARED_DATA - זהו מבנה נתונים של הקרנל שמכיל מידע רב על מערכת ההפעלה, כגון הנתיב המלא לתקייה של Windows, גרסת מערכת ההפעלה, האם מופעל דיבאגר על הקרנל ועוד. בכל תהליך, מבנה נתונים זה נמצא במרחב הכתובות ב-user space, בכתובת 0x7ffe0000, וניתן לקריאה בלבד.
- מיפוי של ה-ntdll.dll - ה-ntdll.dll הינו מודול ב-user mode שמהווה שכבת תקשורת לתהליכים עם ה-kernel. ה-ntdll.dll אחראי לביצוע פעולה מסויימת על ידי קריאה ל-System Call הנדרש, ובכך מתקשר עם הקרנל. פרט לפונקציות אשר אחראיות לביצוע System Calls שונים (למשל NtCreateFile), ה-ntdll.dll מכיל פונקציות שנחוצות עבור תהליכים שאין להם Subsystem מסוים (קרי, Native Subsystem), כמו strcpy, pow וכו'.

Minimal Processes

לפני שנבין מהם ה-pico processes נצטרך להבין מהם ה-minimal processes: החל מ-Windows 8.1 הופיע סוג חדש של תהליכים הנקרא minimal processes. אולם, תהליכים אלו הופיעו באופן רשמי רק ב-Windows 10.

Minimal processes הינם תהליכים בעלי מרחב כתובות ריק. בשל כך הדברים הבאים מתחוללים:

- לא נטען קובץ ההרצה ואף DLL לא ממופה לתוך התהליך (לרבות ה-ntdll.dll).
- לא נוצר שום thread התחלתי להרצת קוד.
- לא נוצר שום מבנה נתונים שנוצר ביצירת התהליך: ה-PEB, ה-TEB לכל thread, ובנוסף גם ה-KUSER_SHARED_DATA.

<u>הערה</u>: ל-minimal process בדומה לתהליך רגיל, יש שם, תהליך "אבא", ו-Security Context.

כדי ליצור minimal process, יש לקרוא לפונקציה NtCreateProcessEx עם flag ספיציפי (0x800) וזאת בתנאי שהקריאה מתבצעת מתוך ה-kernel mode. ומכאן, ניתן להבין כי אין אפשרות ליצור minimal processes ב-user mode ולהריץ מתוכן קוד, אך ניתן להריץ קוד מסוים באמצעות thread-ים הנוצרים ב-kernel-mode. כל ה-thread-ים הנמצאים בתוך minimal process נקראים minimal threads.

ניתן להבחין בתהליכים מסוג זה באמצעות ה-Minimal flag הנמצא ב-EPROCESS הקשור לאותו תהליך. כאשר flag זה מוגדר, אין אפשרות להקצות זיכרון ל-thread ב-user-mode, כמו ליצור TEB או stack בשבילו.

ל-minimal processes קיימים מספר שימושים שונים:

<u>Memory Compression</u>: תהליך זה אחראי לדחיסה של הזיכרון הפיזי, ומחזיק בזיכרון הדחוס במרחב הכתובות שלו ב-user mode. תהליך זה קיים על מנת להגדיל את הזיכרון הפיזי הפנוי, ובכך להפחית את הכתיבה לדיסק. התהליך מנוהל על ידי רכיב הנקרא Store Manager.

<u>Secure System</u>: תהליך זה מייצג את מרחב הכתובות של ה-kernel שנמצא ב-VLT (Virtual Trust Level), אשר מנוהל על ידי ה-VBS (Virtualization Based Security). ה-VBS הינו פיצ'ר חדש של Windows 10, אשר נועד להגן על מערכת ההפעלה מפני קוד או חולשה שהמערכת לא יכולה להגן עליה, כמו דרייבר צד-שלישי שיכול להכיל קוד זדוני/חולשה שיכולה להוביל להרצה של קוד שכזה, ובכך לגרום להתנהגות לא רצויה במערכת. מנגנון זה נעשה באמצעות בידוד של תהליכים קריטים במערכת (הן ב-user space והן ב-kernel space), וחלוקה של המערכת לשתי שכבות הנקראות VTLs, כאשר האזורים הקריטיים נמצאים ב-VTL 1. לשאר הרכיבים הנמצאים ב-VTL 0, אין אפשרות להשפיע על הרכיבים הנמצאים ב-VTL 1.

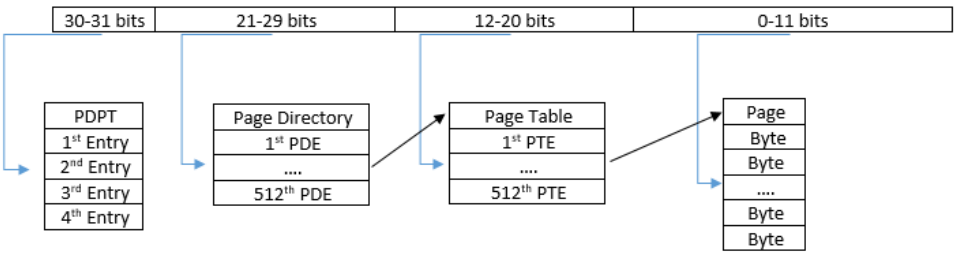
הרחבה - Memory Compression

כאמור, Memory Compression הינו פיצ'ר חדש שפורסם לראשונה ב-Windows 10, שמטרתו היא לצמצם את הכתיבה לדיסק על ידי תוכניות שונות. על מנת לצמצם את הכתיבה לדיסק, מתבצעת דחיסה של Page-ים בזיכרון הפיזי, וכתוצאה מכך מתפנה מקום נוסף לכתיבה ב-RAM. לפני שנבין כיצד תהליך זה עובד, עלינו להבין כיצד הזיכרון מנוהל לכל תהליך במערכת ההפעלה.

קצת על Memory Management

כפי שהבנו מהחלקים הקודמים במאמר, לכל תהליך יש מרחב כתובות וירטואלי פרטי משלו. בדרך כלל, לתהליכים במערכת 32 ביט, מרחב הכתובות בדרך כלל משתרע לגודל של 4 GB, ובמערכת 64 ביט מגיע לגודל של 256 TB, כאשר חצי ממרחב הכתובות משמש עבור מערכת ההפעלה עצמה (2 GB ו-128 TB בהתאמה). הזיכרון מנוהל על ידי רכיב שנמצא ב-executive (הרכיב הכי גדול בו) הנקרא Memory Manager ובאמצעות חלוקה של הזיכרון לבלוקים של 4 KB (בדרך כלל), הנקראים Page-ים. בעת גישה לזיכרון הוירטואלי מתבצע תרגום של הכתובת הוירטואלית למיקום שלו ב-RAM. מיפוי הכתובות ותרגומן לזיכרון הפיזי נעשה על ידי ה-MMU (Memory Management Unit) שממוקמת בדרך כלל במעבד. על מנת לפשט את התהליך, נדגים אותו על מערכת 32 ביט.

כפי שניתן לראות בתרשים הבא, הכתובת הוירטואלית מחולקת ל-4 חלקים:



1. שני הביטים השמאליים (30-31) מתארים את האינדקס של הכניסה ב-Page Directory Pointer Table (PDPT). ה-PDPT הוא מבנה ייחודי לכל תהליך המכיל 4 כניסות, שממנו מתחיל תהליך התרגום. הכתובת הפיזית שלו ידועה בתוך KPROCESS ובתוך register מיוחד של המעבד CR3, שטוען את הכתובת מתוך ה-KPROCESS בעת גישה לזיכרון וירטואלי על ידי thread מסוים. הכניסה ב-PDPT מכילה את הכתובת הפיזית של Page Directory, ונקראת PDPE.
2. תשעת הביטים לאחר מכן (21-29) מתארים את הכניסה בתוך ה-Page Directory מתוך 512 הכניסות האפשריות. כל כניסה נקראת Page Direcotry Entry (PDE), והיא מצביעה על הכתובת הפיזית של Page Table.

3. תשעת הביטים לאחר מכן (12-20) מתארים את הכניסה בתוך ה-Page Directory מתוך 512 הכניסות האפשריות. כל כניסה נקראת Page Table Entry (PTE), והיא מצביעה על הכתובת הפיזית של ה-Page הרצוי.
4. שנים עשר הביטים שנותרו (0-11) מתארים את ה-offset בתוך ה-Page לבית מסויים, ובכך נקבל לבסוף את הכתובת הפיזית בו של בית מסויים אליו ניסינו לגשת מתוך הזיכרון הוירטואלי הנמצא במרחב הכתובות של התהליך.

במהלך תהליך התרגום הנ"ל, יכול להיות כי אחת מהכניסות (ה-PDE או ה-PTE) לא תהיה תקינה. ניתן לבדוק אם הכתובת שנמצאת ב-PDE או ב-PTE תקינה על ידי הביט הראשון, הנקרא valid bit, שערכו יהיה 0 במידה והכתובת לא תקינה. במקרה כזה, המעבד יזרוק שגיאה אשר תטופל על ידי ה- Memory Manager, הנקראת Page Fault. Page Fault יכול להיגרם במצב בו אין הרשאות לגישה לאותו Page, כאשר ה-Page לא קיים ב-RAM וכו'. אנו נדון במצב שיש Page Fault כאשר ה-Page לא נמצא בזיכרון הפיזי של אותו תהליך שאליו אנחנו ניגשים.

אז למה בכלל שיהיה קיים זיכרון לא על ה-RAM? ובכן, גודל מרחב הכתובות הוירטואלי גדול יותר מגודל הזיכרון אשר ניתן לשימוש ב-RAM. פרט לכך, ה-NT Kernel תומכת בגודל מסויים של זיכרון פיזי אשר ניתן להשתמש בו. על מנת להרחיב את ה-RAM, ה-Memory Manager שומר חלק מהזיכרון על הדיסק בתוך Page File. כאשר אין מספיק זיכרון פנוי ב-RAM, ה-Memory Manager יכול להעביר Page-ים (בדרך כלל בכאלו שלא נעשה בהם שימוש רב) לתוך ה-Page File, על מנת ליצור מקום ב-RAM. אולם מנגנון זה דורש גישה לדיסק בעת Page Fault, שכן במצב כזה תתבצע קריאה מתוך ה-Page File וה-Page יטען לתוך ה-RAM בחזרה - דבר הפוגע בביצועים של התהליך. ב-Windows 10 נוסף למנגנון זה, מנגנון ה-Memory Compression, שדוחס את הזיכרון ושומר אותו ב-RAM במקום לכתוב אותו אל הדיסק.

ניהול הזיכרון הפיזי

הזיכרון הפיזי בWindows מנוהל ומתואר על ידי ה-Page Frame Number Database או בקיצור ה- PFN Database, אשר שומר את המצב של כל Page בזיכרון הפיזי. זאת, בנוסף ל-Working Sets ששומרים את המידע על ה-Page-ים של תהליך מסויים בתוך הזיכרון הפיזי.

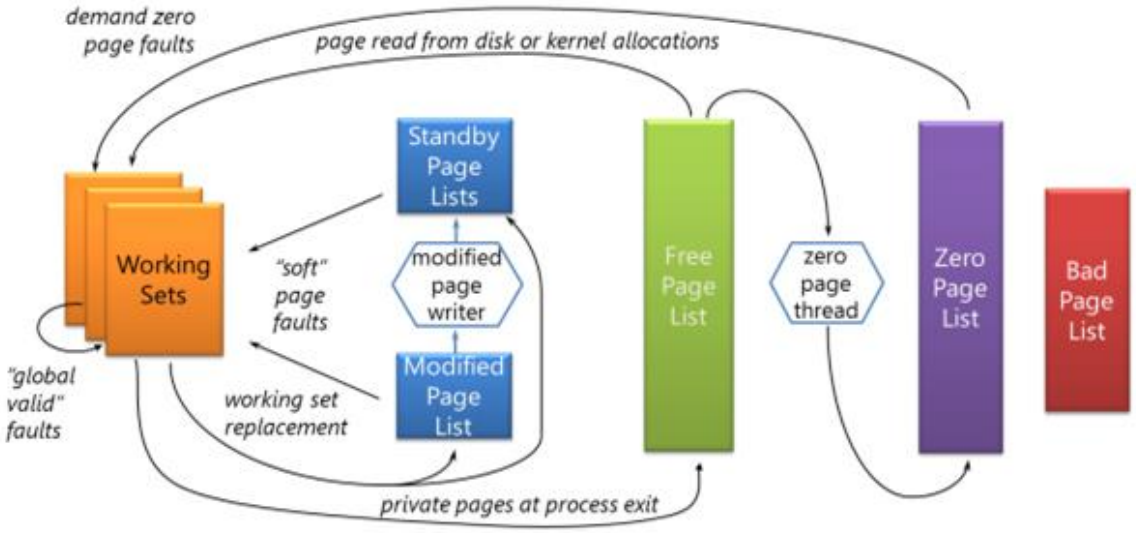
קיימים מספר מצבים אשר יכולים להיות ל-Page, והם:

- Active/Valid - זהו Page שמוחזק על ידי Working Set של תהליך מסויים, או על ידי הקרנל (כמו nonpaged kernel page), וקיים לו PTE תקין שמצביע אליו (ה-valid bit שווה ל-1).
- Modified - זהו Page שהוסר מה-Working Set של תהליך מסויים, אבל כן היה בו שימוש (נכתב לתוכו מידע), ועדין המידע שבתוכו לא נכתב לדיסק.
- Standby - זהו Page שהוסר מה-Working Set של תהליך מסוים, אך כן היה בו שימוש. בשונה מ-Modified Pages, Page-ים אלו כן נכתבו לדיסק.

- Free - זהו Page שאין בו יותר שימוש, אך כתוב בתוכו עדיין מידע מלפני ששוחרר. אין לתת את ה-Page-ים האלו בעת הקצאה של Page חדש על ידי ה-user (למשל בעת ביצוע VirtualAlloc), בשל העובדה שכתוב שם מידע מתהליך אחר. אולם, יש שימוש ב-Page-ים אלו בעת הקצאות של Page-ים בקרנל, או בעת הקצאות של Page עבור טעינת קובץ. בשאר המקרים, יש לדאוג כי ה-Page המוקצה יהיה Zerored Page.
- Zeroed - זהו Page שאין בו שימוש, אך בשונה מה-Free Pages, כולו מכיל אפסים. זאת נעשה על ידי thread הנקרא Zero Page Thread, שלוקח Free Pages ומאפס את כל הביטים בתוכם. פרט לכך, יכולים להיווצר מקרים בהם ה-Page כבר מכיל כולו אפסים, ולכן יהיה במצב זה ולא במצב של Free.

<u>הערה</u>: קיימים עוד מצבים ל-Page-ים, כמו Bad, Modified no-write וכו', שלא תוארו על מנת לפשט את חלק זה.

לכל המצבים הללו, ה-Page-ים מסודרים ברשימות מקושרות, על מנת שיהיה יותר קל ל-Memory Manager לאתר Page-ים במצב מסוים. התרשים הבא מתאר את הרשימות הללו:



[מקור: https://blogs.msdn.microsoft.com/tims/2010/10/29/pdc10-mysteries-of-windows-memory-management-revealed-part-two]

במצב בו אין מספיק זיכרון לתהליך מסויים, ה-Memory Manager מעביר Page-ים מה-Working Set לתוך ה-Modified List, או אל ה-Standby List, במידה ולא נכתב לאותו Page מידע. במצב בו ה-Modified List גדולה מידי וה-Memory Manager מחליט שיש להפחית אותה, ה-Modified Page Writer מתחיל לכתוב Page-ים מה-Modified List לתוך הדיסק, וה-Page-ים יימצאו לאחר מכן ב-Standby List כ-cache. אולם, ה-Standby List נחשבת כחלק מהזיכרון הזמין שתהליכים יכולים להשתמש. כך למשל, במידה וה-Free/Zero Lists ללא משאבים לספק Page-ים, ה-Memory Manager ישתמש ב-Page מתוך ה-Standby List ויאפס אותו. במידה ויתבצע Page Fault עבור מידע שנמצא ב-Modified/Standby Lists, ה-Page יועבר חזרה אל ה-Working Set (ומכאן Soft Page Fault כי לא הזדקקנו לדיסק). אך, במידה

והמידע המבוקש נמצא בדיסק (למשל Page שנכתב לדיסק והוסר מתוך ה-Standby List), תתבצע Hard Page Fault ויוקצה Page חדש מתוך ה-Free List (בדרך כלל) ששם יוכל המידע שהיה קיים עד עתה בדיסק.

כך היה עד לפני קיומו של ה-Memory Compression. כעת, ל-Memory Manager קיימת האפשרות להעביר את המידע מתוך ה-Modified List אל ה-Working Set של ה-Memory Compression, במקום לכתוב בדיסק. חשוב לציין כי המידע הדחוס נשמר בתוך Working Set של תהליך, ולכן כל התהליך על אותם Page-ים דחוסים יכול להתבצע כפי שמתבצע על ה-Page-ים של תהליכים אחרים. כלומר, המידע הדחוס יכול לעבור אל ה-Modified List ובמידת הצורך להיכתב לדיסק ולהגיע ל-Standby List.

כיצד ה-Memory Compression פועל?

כאשר ה-Memory Manager יחליט שאין מספיק זיכרון פנוי ב-RAM, יוכל כעת לדחוס Page-ים (שבדרך כלל השימוש בהם היה נמוך) ולשמור את המידע כדחוס, במקום לכתוב את המידע לדיסק ישירות. במידה ויהיה צורך באותו Page, יתבצע התהליך ההפוך שלאחריו יוכל התהליך באותו Page.

כדי לבצע את תהליך זה, ה-Memory Manager משתמש ברכיב הנקרא Store Manager. ה-Store Manager אחראי לביצוע הדחיסה של ה-Page-ים ושמירה של המידע והדחוס יחד עם הכתובת הוירוטאלית שלו, ואף אחראי לאינטראקציה עם ה-Memory Management.

ה-Store Manager מכיל Stores, ששם נשמר כל המידע הדחוס. קיימת Store לכל התהליכים שאינם אפליקציות UWP (Universal Windows Platform) הנקראת System Store, ונוצר על ידי ה-Superfetch בעת טעינת המערכת. לעומת זאת, אפליקציות ה-UWP מתקשרות בעת יצירתן עם ה-Superfetch על מנת ליצור Store משלהן. ה-Stores נמצאות בתוך ה-Working Set של ה-Memory Compression בתוך ה-user space של התהליך, משום שזהו תהליך מינימלי ויכולה מערכת ההפעלה להשתמש במרחב הכתובות שנמצא ב-user mode כרצונה.

לכל Store יש מספר רכיבים שבתוכם מאחסן את המידע על ה-Page-ים ואת המידע שבתוכם לאחר הדחיסה. ראשית, כל Store מכיל עץ האחראי לשמירת המידע על ה-Page-ים, כאשר כל חוליה בו מצביעה על המקום שבו נמצא המידע הדחוס של אותו Page. אותו מידע דחוס נשמר במערך של Regions, כאשר כל region בעל ערך קבוע שניתן לקנפג (כברירת מחדל: 128 kb). המידע עצמו של ה-Page נשמר בחתיכות של 16 ביט. לאחר יצירה של Store, אין בה אף לא region אחד, ולכן ה-Store Manager מבצע אלוקציות ודיאלוקציות במידת הצורך.

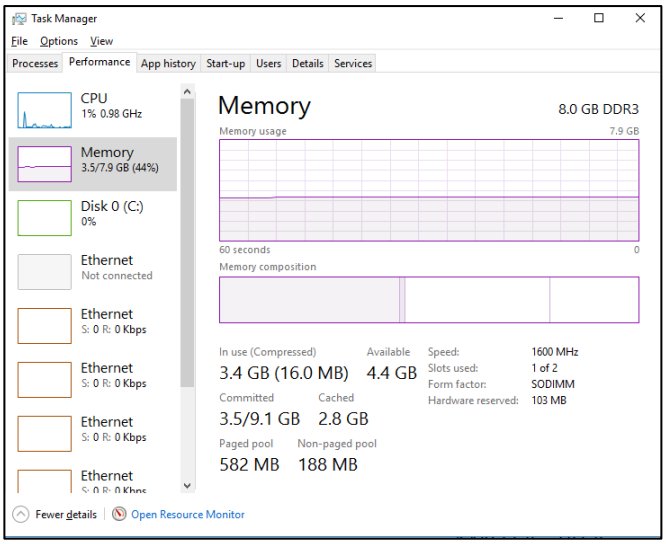
בדומה למידע על ה-Page-ים, המידע על ה-regions נמצא אף הוא בעץ. בעת הוספה של Page דחוס והוצאה שלו, מעודכן המידע בעצים, ונבדק האם אין מספיק ב-region ל-Page או האם ה-region ריק

בהתאמה, שכן כך ה-Store Manager יכול לבצע הקצאות או שיחרורים של הזיכרון של ה-regions. חשוב להדגיש כי עד שה-region לא ריק לגמרי, לא יתבצע שחרור של הזיכרון. כמו כן, הוספה של Page תתבצע ב-thread עם low-priority (7), וביצוע decompression ל-Page תתבצע ב-parallel, שכן יש צורך באותו Page בתהליך מסויים.

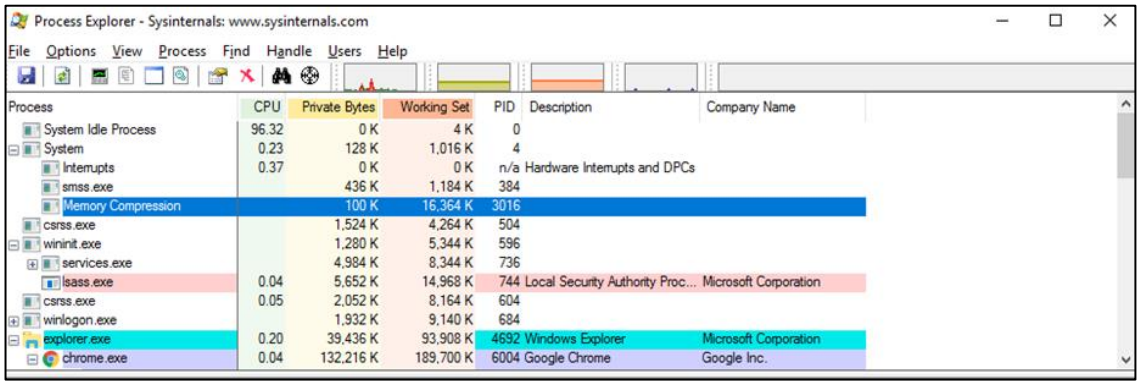
ה-Memory Compression כתהליך מינימלי

בגרסאות הראשונות של Windows 10, ה-Memory Compression היה חלק ממרחב הכתובות של התהליך System. אולם, דבר זה היה נראה תמוה, שכן בעקבות כך היה נראה כי התהליך System צורך הרבה זיכרון, אך בפועל היה שומר את ה-Page-ים הדחוסים. לכן, הוחלט להפריד את תהליך זה מה-System, וכיום מהווה תהליך מינימלי. אמנם לא ניתן לראות את תהליך זה בתוך ה-Task Manager, אך ניתן לראות אותו מתוך הכלי Process Explorer. מתוך ה-Task Manager ניתן לראות את כמות הזיכרון הדחוס במערכת.

ה-Task Manager הינו כלי שמאפשר לראות את רוב התהליכים אשר רצים במערכת, ומידע נוסף על המערכת כמו השימוש ב-CPU, ב-RAM, בדיסק וכו'. על מנת לראות את כמות הזיכרון הדחוס במערכת יש לגשת את הטאב Performance, וללחוץ על החלק שמתאר את הזיכרון (Memory). כעת תוכלו לראות בחלון את השימוש של המערכת ב-RAM, את כמות הזיכרון הפיזי שזמינה במערכת וכו'. ליד כמות הזיכרון שנמצאת בשימוש במערכת, יימצא גודל הזיכרון הדחוס בסוגריים.

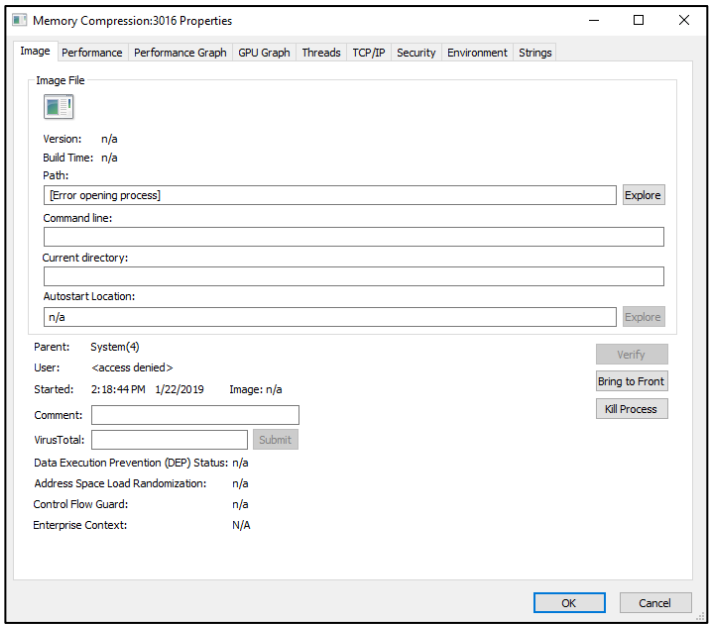


Process Explorer הינו כלי מתוך אוסף של כלים הנקרא SysInternals. כלי זה מאפשר לקבל מידע על כל התהליכים שקיימים במערכת, ואינפורמציה רחבה יותר ממה שמספק ה-Task Manager. בנוסף, ב-Process Explorer ניתן לראות את התהליך של Memory Compression ואת מרכיביו השונים.



ניתן לראות כי תהליך ה-Memory Compression הוא child process של system, מה שמצביע שזהו תהליך של המערכת. בנוסף לכך, ניתן להבחין כי ה-Working Set, שזהו גודל הזיכרון הפיזקלי שמשתמש התהליך, שווה בגודלו לכמות הזיכרון הדחוס במערכת שראינו ב-Task Manager. אם נלחץ התהליך נוכל לקבל מידע מפורט יותר על תהליך זה.

נוכל לראות כי אין Image file מסויים על תהליך זה, כלומר לא מופה שום קובץ executable לתוך תהליך זה - דבר המרמז לנו על היותו של התהליך תהליך מינמלי. אם נסתכל על הטאב של ה-Threads נראה כי ה-Threads מריצים את הפונקציה KeQueryNodeActiveAffinity. זוהי אינה הפונקציה שרצה כעת על ה-thread-ים בתהליך זה, אך ל-Process Explorer אין את ה-Symbols עבור תהליך זה. לכן, ניתן להשתמש בכלים אחרים, כמו Local Kernel Debugger, על מנת לראות את הפונקציות הללו. ההסבר על ה-kernel debugger המקומי נמצא בפסקה הבאה, אך אדגים שם גם את ה-thread-ים עצמם.



על מנת להראות שתהליך זה הינו באמת תהליך מינימלי, עלינו להראות שהפלאג Minimal אשר נמצא ב-EPROCESS הוא 1. על מנת לבחון את ה-EPROCESS נשתמש ב-Local Kernel Debugger. על מנת להריץ את ה-kernel debugger נשתמש ב-windbg, debugger ל-Windows, עם הפלאג kl. יש להריץ את ה-

windbg עם הרשאות Adminstrator. על מנת לתמוך ב-kernel debugging יש לעקוב אחרי המאמר הבא מתוך ה-MSDN.

נשתמש בפקודה:

```
!process cid 0
```

כאשר ה-cid הינו קיצור ל-CLIEND_ID, אשר מכיל HANDLE-ים ל-process ול-thread של אותו תהליך. במקרה שלנו, הערך שלו שווה ל-PID של התהליך, שאותו ניתן לקבל מתוך Process Explorer.

הוספת ה-0 בסוף זה על מנת לקבל תקציר של המידע של אותו תהליך:

```
lkd> !process bc8 0
Searching for Process with Cid == bc8
PROCESS ffffbc0cf57fc040
    SessionId: none  Cid: 0bc8    Peb: 00000000  ParentCid: 0004
    DirBase: 5c422002  ObjectTable: ffffcc085e889b40  HandleCount: <Data
Not Accessible>
    Image: MemCompression
```

ללא הפרמטר 0, נוכל לקבל מידע מורחב על התהליך שכן יספק לנו גם מידע על ה-thread-ים של התכנית עם שמות הפונקציות שרצות על ידי אותו thread-ים.

```
lkd> !process bc8
Searching for Process with Cid == bc8
PROCESS ffffbc0cf57fc040



THREAD ffffbc0cf57fe080  Cid 0bc8.0bd0  Teb: 0000000000000000
Win32Thread: 0000000000000000 WAIT: (Executive) KernelMode Non-Alertable
ffffbc0cf57ed7c0  NotificationEvent Not impersonating
        Owning Process            ffffbc0cf57fc040       Image:
MemCompression
        Attached Process          N/A            Image:         N/A
        Wait Start TickCount      1689535        Ticks: 6292
(0:00:01:38.312)
        Context Switch Count      31182          IdealProcessor: 0
        UserTime                  00:00:00.000
        KernelTime                00:00:00.078
        Win32 Start Address nt!SmKmStoreHelperWorker
(0xfffff8009c8fa0ec)



THREAD ffffbc0cf2cbe080  Cid 0bc8.203c  Teb: 0000000000000000
Win32Thread: 0000000000000000 WAIT: (Executive) KernelMode Non-Alertable
ffffbc0cfbc1f8d8  NotificationEvent ffffbc0cfbc1f8c0  NotificationEvent
Not impersonating
        Owning Process            ffffbc0cf57fc040       Image:
MemCompression
        Attached Process          N/A            Image:         N/A
        Wait Start TickCount      650399         Ticks: 1046000
(0:04:32:23.750)
        Context Switch Count      131            IdealProcessor: 1
        UserTime                  00:00:00.000
```

```
        KernelTime                00:00:00.000
        Win32 Start Address nt!SMKM_STORE<SM_TRAITS>::SmStReadThread
(0xfffff8009c8fb2a0)
```

ניתן לראות כי שתי הפונקציות שרצות על ידי ה-thread-ים הם SmKmStoreHelperWorker ו-SmStReadThread, אשר חלק מה-Store Manager ומנהלות את ה-Memory Compression.

יתרה מכך, נוכל לראות כי ה-Parent Id הינו 4, שזהו ה-PID הקבוע של התהליך System, כפי שראינו ב-Process Explorer. בנוסף לכך, נוכל לראות כי ה-PEB מצביע ל-NULL, ואף לכל thread (מהרצת הפקודה השניה) ה-TEB מצביע ל-NULL. דבר זה מרמז על היותו של התהליך מינימלי. בנוסף לכך, בשורה הראשונה של הפלט נוכל להבחין בכתובת ffff8f05b79942c0: זוהי הכתובת של האובייקט המייצג את התהליך בקרנל - ה-EPROCESS.

על מנת לקבל את המידע הנמצא ב-EPROCESS, נשתמש בפקודה dt nt!_EPROCESS addr כאשר addr זוהי הכתובת של ה-EPROCESS.

```
lkd> dt nt!_EPROCESS ffff9b080d86b040
   +0x000 Pcb              : _KPROCESS
   +0x2d8 ProcessLock      : _EX_PUSH_LOCK
   +0x2e0 RundownProtect   : _EX_RUNDOWN_REF
   +0x2e8 UniqueProcessId  : 0x00000000`00000bc8 Void

   +0x6cc Flags3           : 0x800001
   +0x6cc Minimal          : 0y1
   +0x6cc ReplacingPageRoot : 0y0
   +0x6cc DisableNonSystemFonts : 0y0

   +0x710 PicoContext      : (null)
```

נוכל להבחין כי בכתובת 0x6cc, התכונה Flags3, אשר נוספה ב-Windows 8.1, מכילה בביט הראשון שלה את הערך 1, שזהו הביט שמציין את היותו של התהליך מינימלי. Windbg נותן לנו פירוט על כל אחד מחלקיו של Flags3 ומציין כי Minimal שווה ל-0x1, מה שאומר שהתהליך מינימלי. פרט לכך, ניתן להבחין כי התכונה PicoContext שווה ל-null, מה שמבחין בין תהליך מינימלי ל-Pico Process.

Pico Processes & Providers

Pico Process הוא תהליך מינימלי שמחובר אל kernel-mode driver. דרייבר מסוג זה נקרא Pico Provider ותפקידו לשלוט בריצה שתתבצע בתוך התהליך. למעשה, השליטה שמסופקת ל-Pico Provider רבה כל כך, שמסוגל הדרייבר לחקות את ההתנהגות של מערכת הפעלה שונה, ובכך להריץ תוכניות שמיועדות למערכות הפעלה אחרות, מבלי שהתוכניות יהיו מודעות שהן רצות על Windows כלל. ה-Pico Provider אחראי ליצירת והריגת Pico Process ו-Pico Threads, טיפול כאשר Pico Process מבצע System calls, טיפול בחריגות ועוד. בדומה לתהליך מינימלי, pico process בעל מרחב כתובות ריק, ולכן לא נוצר בו PEB, TEB, KUSER_SHARED_DATA, ולא ממופים אליו DLLs וקובץ להרצה. כדי להבחין בין pico process ל-minimal process, התכונה PicoContext באובייקטים EPROCESS ו-ETHREAD שתהיה שונה מ-NULL כאשר התהליך הינו Pico Process. ה-threads של Pico process נקראים pico threads.

על מנת לתמוך במנגנון הזה, על דרייבר להירשם כ-Pico Provider באמצעות הפונקציה PsRegisterPicoProvider. לאחר ההירשמות כ-Pico Provider, הדרייבר מקבל רשימת פוינטרים לפונקציות שמאפשר לו לנהל Pico Processes:

- PspCreatePicoProcess, PspCreatePicoThread - יצירה של Pico Process ו-Pico Thread.
- PspTerminateThreadByPointer, PspTerminatePicoProcess - סגירה של Pico Process ו-Pico Thread.
- PsSuspendThread, PsResumeThread - השהייה והמשך של Thread.
- PspGetPicoProcessContext, PspGetPicoThreadContext - הגדרה וקבלה של הערך שבשדה Pico Context שנמצא באובייקטים EPROCESS ו-ETHREAD.
- PspSetPicoThreadDescriptorBase - השמה של ערכים ב-FS registers ו-GS, שבדרך כלל מצביעים למבני נתונים שמכילים מידע על thread מסוים (כמו TEB).

בנוסף לכך, שולח ה-Pico Provider אל ה-kernel קבוצת מצביעים לפונקציות שמהוות callbacks לאירועים שונים שיכולים להתבצע ב-Pico Process/Thread. ה-callbacks הם כאשר: Pico Thread מבצע system call, נזרקת שגיאה מתוך Pico Thread, יש בקשה לשם של Pico Process, Event Tracing for Windows מבקש את ה-stack trace של Pico Process, יש ניסיון לפתיחת Handle ל-Pico Process/Thread, יש ניסיון לסגירה של התהליך או thread, או כאשר התהליך או thread מסוים נסגר באופן בלתי צפוי. השימוש ב-API שמספק הרשמה של הדרייבר כ-Pico Provider ומספק את הטיפול ב-Pico Processes ו-Pico Threads, זמין כאשר PspPicoRegistrationDisabled הוא FALSE. שדה זה הופך ל-TRUE לפני שה-boot drivers נטענים. לכן, על הדרייבר להיטען לפני כל שאר הדרייברים, וזאת יכול להיות להתבצע רק אם הדרייבר חתום על ידי Microsoft Signer Certificate. כיום, השימוש היחידי ב-Pico Processes על ידי Windows הוא ה-WSL (Windows Subsystem for Linux), שמאפשר ריצה של פקודות bash והרצה של קבצי ELF שהדרייברים האחראים עליו הם Lxss.sys ו-LxssCore.sys.

Windows Subsystem for Linux

ה-WSL (Windows Subsystem for Linux) הינו פיצ'ר חדש שנחשף בעדכון של Windows 10, וזמין החל מהגרסה 1607. ה-WSL מספק סביבה שמסוגלת להריץ קבצי ELF ללא צורך בקומפילציה מחדש. באמצעות המנגנון החדש שנבנה כתוצאה מהפרוייקט Drawbridge המתואר בהקדמה, הצליחו Microsoft לספק הרצה של קבצי ELF ללא שום קוד של הקרנל של לינוקס עצמו.

לפני שנבין את המבנה של ה-WSL, נצטרך להבין את המושג Subsystem ואיך הוא מתקשר ל-WSL.

Subsystem

Subsystem הינה תצוגה מיוחדת לקרנל ולשירותים של מערכת ההפעלה עבור יישומים במערכת. למעשה, התפקיד של ה-Subsystem הוא לחשוף חלק מסויים מתוך השירותים של הקרנל, בדרך כלל של ה-Executive - השכבה העליונה של הקרנל שמספקת שירותים של מערכת ההפעלה הקשורה בניהול רכיבים של מערכת ההפעלה, כמו ניהול תהליכים, I/O, זיכרון וכו'.

במקור, ה-NT kernel תמכה ב-OS/2 וב-POSIX בנוסף ל-Win32 Subsystems. כל Subsystem סיפק ממשק ומימש את הפונקציות הנחוצות לריצה של תוכניות ב-Subsystem. ממשק זה בא לידי ביטוי במודולים ב-user land (DLLs), הנקראים ביחד Subsystem DLLs. למשל, ה-Windows Subsystem DLLs, מממשים את הפונקציות של ה-API (למשל kernel32.dll, user32.dll, advapi.dll וכו').

אך, לרוב ה-Subsystems הללו הוסרה התמיכה. למשל, התמיכה ב-OS/2 נמשכה עד ל-Windows 2000. התמיכה ב-POSIX נמשכה עד Windows XP, ולאחר מכן הוחלפה ב-SUA (Subsystem for UNIX-based Applications), אך גם התמיכה בה נפסקה לאחר Windows 7.

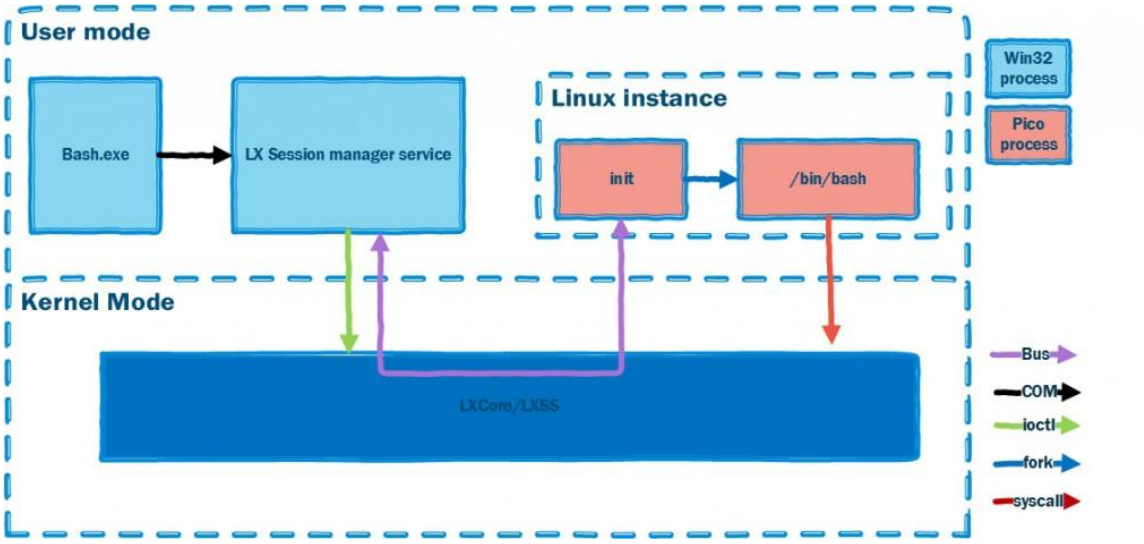
קיימים executables שאין להם Subsystem, או שיש להם Native Subsystem (המשמעות אותה משמעות). לכן, אותם executables לא יכולים להישען על שום Subsystem DLL, ומשתמשים ב-ntdll.dll, וב-Native API שהוא מספק על מנת לתקשר עם הקרנל. לכן, זוהי הסיבה ש-ntdll.dll מכיל פונקציות שמוכרות לנו משפת C, כמו sprintf, strncmp וכו': על מנת למנוע מימוש חוזר של אותן פונקציות שאותם executables משתמשים, הוחלט כי ימומשו ב-ntdll.dll.

מבנה ה-WSL

ל-WSL יש מספר רכיבים שמאפשר לו להריץ קבצי ELF ללא שינוי על Windows, ללא ידיעה בכלל שהם רצים על Windows. הוא משתמש במספר רכיבים שממשים את המנגנון של ה-Pico Processes, הכולל רכיבים הן ב-user mode והן ב-kernel mode:

- Lxss/LxCore drivers - הדרייברים הנמצאים ב-kernel mode, שנרשמים וממשים את המנגנון של ה-Pico Providers. דרייברים אלו אחראים ליצירה וטיפול ב-Pico Processes, וחיקוי ההתנהגות של הקרנל של Linux על מנת לאפשר את ההרצה של קבצי ה-ELF. בנוסף לכך, הם מממשים מערכת קבצים וירטואלית (VFS) שמהווה שכבה העוטפת את ה-NTFS ומדמה את מערכת קבצים של לינוקס (יורחב בהמשך). למעשה, ה-Lxss רושם את עצמו כ-Pico Proivder, אך הדרייבר שמספק את המימוש הוא ה-Lxcore. ה-Lxcore לא טוען עצמו כ-Pico Provider, מכיוון שמכיל כמות רבה של קוד שאין צורך שתיטען מוקדם כל כך, ולכן מטעמי ביצועים ה-Lxss נרשם כ-Pico Provider, ולאחריו נטען ה-Lxcore, אשר מעתיק את ה-Dispatch Tables של ה-Lxss.
- LXSS Manager service - הנמצא ב-user mode ומתווך בין ה-Pico Providers ל-bash.exe, ובכך מאפשר ל-bash.exe ליצור Linux instance חדש של ה-WSL ולהריץ את התכנית הרצויה (בדרך כלל, /bin/bash). המימוש של ה-LXSS Manager נמצא בתוך ה-lxssmanager.dll שמורץ על ידי svchost.exe, ורץ כ-PPL (Protected Process Light).
- Pico Processes - התהליכים המכילים את קבצי ה-ELF. אחד מהתהליכים האלו הוא ה-init deamon, וכל שאר התהליכים (למשל /bin/bash) מהווים ילדים שלו. בשל העובדה שה-WSL לא משתמש בקוד של הקרנל Linux, על Microsoft לכתוב init deamon משלהם.
- Windows Processes - קיימים שני קבצי exe שה-WSL משתמש בהם כדי לתקשר עם ה-LXSS Manager: bash.exe שדרכו ניתן ליצור Linux Instances חדשים (ומכאן מפעיל את ה-init deamon בהתחלה ולאחר מכן את התכנית הרצויה), ו-LxRun.exe על מנת לעדכן או לקנפג את ה-WSL.

לסיכום, המבנה של ה-WSL נראה כך:



[מתוך https://blogs.msdn.microsoft.com/wsl/2016/04/22/windows-subsystem-for-linux-overview]

כפי שניתן לראות, התרשים מתאר את היצירה של Linux Instance שבתוכו רץ bin/bash/ שנמצא בתוך Pico Process, וזאת נעשה על ידי ה-LXSS/LXCORE, שאחראים על ה-Pico Processes, ועל ידי ה- LXSS Manager שמתווך בין הדרייברים ל-bash.exe שאחראי ליצור Linux Instances.

Bash.exe מתקשר עם ה-LXSS Manager על ידי COM (Compoment Object Model). COM הינו סטנדרט בינארי ליצירת רכיבים שיכולים לתקשר אחד עם השני, ללא תלות אחד בשני. את אותם רכיבים/אובייקטים אנחנו ממשקים, כלומר כל אובייקט מממש ממשקים שמתארים את הפעולות שניתן לעשות על אותו אובייקט. כדי לבצע פעולות מסויימות, אנחנו קוראים לאחת הפונקציות שבאחד מהממשקים, כאשר המימוש נמצא במופע של אותו אובייקט של COM. כלומר, במקום שתהיה לנו גישה למופע של אובייקט של COM, ניתנת לנו גישה לממשק שאת הפעולות שהוא מספק אנו יודעים, וכאשר אנו מבקשים לבצע פעולה מסויימת, היא תבוצע לפי אותו מופע של האובייקט שנמצא מאחורי הקלעים.

המופע של אותו אובייקט יכול להיות ב-DLL, או בכלל מחוץ לאותו תהליך, זה לא באמת משנה: למבקש הפעולה (ה-client) לא משנה כיצד תתבצע הפעולה, היות והוא מודע לסוג הפעולות שיכולות להתבצע ולא למימושן. דבר זה הופך את המודל ללא תלות בשפה: אפליקציה ב-C# למשל יכולה לתקשר באמצעות COM עם שירות שמסופק על ידי בינארי ב-C++ למשל. במקרה שלנו, ה-LXSS Manager חושף עבור bash.exe ממשקים שאיתם הוא יכול לתקשר עם ה-LXSS Manager על גבי COM. אחד מהממשקים לדוגמה, הוא הממשק IID_ILxssInstance. ממשק זה מספק ביצוע פעולות על ה-Linux Instance לאחר שנוצר. למשל, ניתן לקבל את ה-id של אותו Linux Instance, ניתן ליצור תהליכי לינוקס דרכו וכו'. חשוב לציין כי אף ה-LxRun.exe מתקשר עם ה-LXSS Manager באמצעות COM. כל התקשורת באמצעות COM אל ה-LXSS Manager היא ללא תיעוד, ולכן היחידים שאמורים לבצע את התקשורת אל ה-LXSS Manager

אלו bash.exe ו-LxRun.exe. עם זאת, כן ניתן לתקשר עם ה-LXSS Manager באמצעות ה-COM Interfaces שהוא מספק.

ה-Lxss Manager מתקשר עם ה-LxCore על ידי ioctl. ioctl (I/O Control), הינה דרך לתקשר עם דרייברים מסוימים ובכך לגרום לביצוע פעולות מסוימות על ידי דרייברים ספיציפיים מה-user spaces. בלינוקס קיים ה-ioctl Syscall על מנת להשיג מטרה וזו, וב-Windows קיימת הפונקציה DeviceIoControl ב-kernel32.dll. במקרה זה, ה-Lxss Manager משתמש ב-ioctls על מנת לתקשר עם ה-LxCore ולבצע פעולות דרכו.

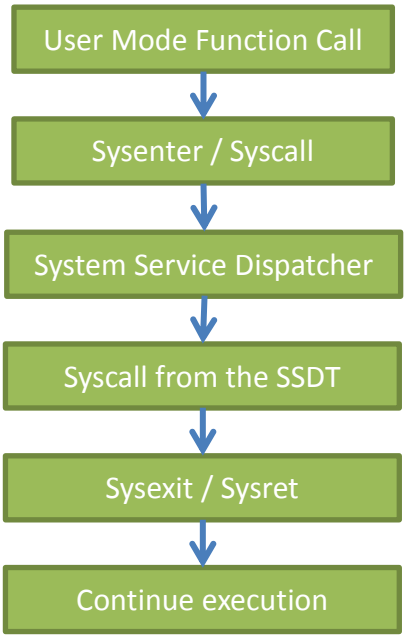
ה-Pico Provider מתווך בין ה-Pico Processes לבין ה-Lxss Manager באמצעות מנגנון הנקרא LxBus. זהו מודל IPC (Inter-Process Communication) פנימי של ה-WSL על מנת לאפשר תקשורת בין התהליכים של ה-WSL לבין שאר תהליכי הWindows, ומאפשר שליחת וקבלת הודעות בין התהליכים. המנגנון הזה בא לידי שימוש בעיקר על ידי ה-init deamon וה-LXSS Manager. ניתן לגשת אל ה-LxBus מתוך ה-WSL באמצעות פתיחה של file descriptor ל-/dev/lxss, ומתוך Windows על ידי ה-HANDLE ל-Linux Instance. ברגע שהם פתוחים, ניתן להשתמש ב-ioctls אל ה-LxCore.sys כדי ליצור חיבור, כמו IOCTL_ADSS_REGISTER_SERVER ו-IOCTL_ADSS_CONNECT_SERVER כדי להירשם כ-server או להתחבר ל-server. לאחר יצירת חיבור בין שני תהליכים, על ידי ה-ioctls הללו, תהליך ה-NT יקבל HANDLE שייצג את החיבור, והתהליך ב-WSL יקבל File descriptor, שאיתם יכולים התהליכים לתקשר בינהם.

התהליך init הינו התהליך הראשון שנוצר בעת יצירה של Linux Instance חדש, אשר רץ כ-deamon. כל תהליך שיווצר באותו Linux Instance יירש מה-init בדרך ישירה או עקיפה, על ידי ביצוע fork. לדוגמה, יתבצע fork מתוך ה-init deamon לפני ההרצה של /bin/bash על מנת לדאוג להרצתו. מכאן, התהליכים ייפעלו כרגיל, שכן יתקשרו לא במודע עם ה-Pico Provider עם syscalls.

Syscalls ב-WSL

System call או בקיצור Syscalls, הינה צורת תקשורת לביצוע פעולות הנמצאות בקרנל, אשר מספקות שירות מסויים לקורא שלהן, כמו NtCreateFile על מנת לפתוח קובץ, NtCreateUserProcess על מנת ליצור תהליך חדש וכו'. כאשר אנו קוראים ל-System Call מסוים, תתבצע הפקודה sysenter או syscall במערכות 32 ביט ו-64 ביט בהתאמה, ועל מנת לחזור מהאותו System call, תתבצע הפקודה sysexit או sysret במערכות 32 ו-64 ביט בהתאמה. לאחר ביצוע הפקודה Sysenter/Syscall, נימצא בתוך ה- System Service Dispacher, רכיב הנמצא בתוך ה-executive, ואחראי לקריאת ה-Syscall הרצוי, בהתאם לארגומנטים הניתנו לו (למשל, האוגר eax יכיל את מספר ה-Syscall המתבקש). ה-Syscalls השונים

נמצאים בתוך טבלה הנקראה System Service Dispatch Table (SSDT). התרשים הבא מתאר בכלליות את מנגנון ה-Syscalls ב-Windows. להרחבה על המנגנון ניתן לקרוא במאמרו של שחק שלו.



במידה ו-Pico Process מבצע Syscall מסוים, ה-System Service Dispatcher יפנה את הטיפול ב-System call לידי ה-Pico Provider - במקרה שלנו ה-Lxcore. ה-Pico Provider מתקשר עם הקרנל, על מנת לממש את הפונקציונליות שקיימת בלינוקס. למשל, על מנת לממש את ה-sched_yield Syscall, משתמש ה-Pico Provider בפונקציה ZwYieldExecution, אשר מיוצאת בקובץ ntoskrnl.exe. בחלק מהפונקציונליות היה צורך במימוש מחדש, כמו ב-Pipes, בעקבות השוני במנגנונים במערכות ההפעלה. דבר שחשוב לציון הוא שהודות לקיומה של ה-POSIX Subsystem, מימושו של ה-WSL היה יכול להתבצע בפשטות רבה יותר. לדוגמה, הודות ל-Subsystem זה, הוחלט כי ה-NTFS (NT File System) יהיה Case Sensitive, ובעקבות כך התאפשר מימוש של מערכת הקבצים של לינוקס ב-WSL בפשטות יותר. אולם, דבר זה אינו סותר את העובדה שב-Windows השימוש ב-NTFS הוא Case Insensitve, שכן השימוש ב-NTFS כ-Case Sensitive הוא לא מופעל בברירת מחדל.

מערכת הקבצים ב-WSL

על מנת לדמות ולממש את השימוש בקבצים ב-WSL כפי שנעשה בלינוקס, ה-WSL מכיל שכבה הנקראת VFS (Virtual File System) הנמצאת ב-LxCore. ה-WSL ממפה את הקבצים שנמצאים בWindows לתוכו, ובכך מאפשר שימוש בקבצים אשר נמצאים ב-Windows בתוך ה-WSL. זאת נעשה בקלות היות ותהליכי ה-WSL נמצאים ב-Windows עצמו ולא במכונה וירטואלית שמדמה מערכת הפעלה שלמה, ובכך יש גישה ישירה לכל הקבצים שנמצאים על מערכת ההפעלה.

ה-VFS אינו רכיב חדש שיצא לראשונה ב-WSL, אלא רכיב אשר מדמה את ה-VFS אשר נמצא בלינוקס. ה-VFS הינה שכבה בקרנל של לינוקס, אשר מספקת ממשק לביצוע פעולות הקשורות ב-file systems מה-user space. בנוסף לכך, ה-VFS מספק ממשק ל-file systems שעליהם לממש. ה-VFS מממש את ה-Syscalls הקשורים בשימוש בקבצים, כמו open, stat, chmod ודומים להם, וזאת באמצעות שימוש במספר מבני נתונים ואובייקטים שונים כמו directory entries, Inodes, file objects וכו'.

כאשר System call מתוך Pico Process מתבצע, למשל open, המימוש של ה-System call, אשר נמצא בתוך ה-LxCore, מעביר את הפעולה לידי ה-VFS. ה-VFS מבצע את הפעולה בדומה ללינוקס, באמצעות רכיבים/פלאיגינים על מנת לייצג את כל סוגי הקבצים שקיימים על המערכת: הקבצים אשר נמצאים על הדיסק (הן בצד הלינוקסי והן בצד ה-Windows-י), קבצים אשר נמצאים על הזיכרון (למשל קבצים המתארים תהליכים, קבצים זמניים, קבצים המתארים דרייברים וכו').

ה-VolFs וה-DrvFs מייצגים את הקבצים שנמצאים בדיסק, הן עבור לינוקס והן עבור Windows. ה-VolFs מספק תמיכה בתכונות של מערכת הקבצים בלינוקס, לעומת ה-DrvFs שמספק גישה אל הקבצים של Windows מתוך ה-WSL על ידי מיפוי שלהם לתוכו. ה-VolFs מספק תמיכה לרוב התכונות שמאפיינות הקבצים בלינוקס, הכוללת הרשאות לקבצים, symbolic links, sockets וכו'. בשל העובדה כי Windows משתמש באובייקטים בקרנל על מנת לנהל את הקבצים, לעומת לינוקס שמשתמש ב-dentries ו-inodes, על ה-VolFs לנהל ולעטוף את האובייקטים שנמצאים ב-Windows באמצעות מבנים אלו. כך למשל, בעת בקשת inode מסויים על ידי ה-VFS (באמצעות lookup לדוגמה), ה-VolFs משתמש ב-HANDLE של ה-parent inode כדי לקבל HANDLE של ה-inode המבוקש. אותם HANDLES לאובייקטים של Windows הם ללא הרשאות כתיבה או קריאה, וכך ניתן לבצע בקשות רק על ה-metadata. פרט לכך, ה-VolFs צריך לספק יכולות שלא בהכרח Windows תומך, כמו Case Sensivity, תווים שלא נתמכים בנתיבים ב-Windows אך בלינוקס כן ועוד. לעומת זאת, ה-DrvFs מאפשר שימוש בקבצים שנמצאים ב-Windows, מתוך ה-WSL, על ידי מיפוי ה-drives לתוך התקייה mnt/ (למשל mnt/c/). ה-DrvFs מתנהג בדומה ל-VolFs, שכן משתמש ב-inodes, ו-file objects, ובו זמנית פותח HANDLE לקובץ ב-Windows שמיוצג על ידי אובייקט בקרנל (על ידי ה-Object Manager).

בנוסף ל-VolFs וה-DrvFs, קיימים קבצים שאינם מיוצגים על הדיסק, כמו מידע על תהליכים, ת'רדים והתקנים על ידי הקרנל, וכן קבצים זמניים הנשמרים בזיכרון. קבצים אלו מיוצגים על ידי ה-ProcFs, SysFs ו-TmpFs בהתאמה. במקרה של ה-WSL, המידע שתואר נשמר ב-Lxcore.sys, אך כן יש מקרים בהם ה-LxCore מבקש את המידע מה-NT Kernel.

קצת על Security

ה-WSL הינו מוצר יחסית חדש של Microsoft, ולכן היה ניתן לראות בו מספר רב של בעיות הן בעיצוב שלו (מבחינת הרכיבים), והן מבחינת אבטחה. אמנם חלק מהבעיות תוקנו, אך עדיין קיימות בעיות בו. למשל, עוד בגרסת ה-Preview של ה-WSL, ה-LxCore הותקן כברירת מחדל במערכת ההפעלה, על אף חוסר קיומו של ה-WSL באמת (הוא עדיין לא יצא באופן רשמי). אולם, על מנת להשתמש בו, המערכת הייתה צריכה להיות ב-developer mode, ובדיקות לגבי השימוש ב-WSL נעשו על ידי ה-LXSS Manager על ידי ה-COM interface. אך, היה ניתן לקשר עם ה-LxCore ובכך ליצור Pico Processes, מבלי קיומו של ה-WSL כלל. דוגמה נוספת היא שהיה גם ניתן להזריק thread-ים של ה-Win32 Subsystem לתוך ה-Pico Processes, וכן אף לפתוח HANDLE ל-Pico Process ולהזריק קוד או לשנות את הזיכרון של התהליך. בעיות אלו נתנו אפשרות לשנות את ההתנהגות של ה-WSL מתוך Windows, דבר שלא היה אמור לקרות, ולכן בעיות אלו תוקנו. פרט לבעיות מסוג זה, נמצאו באגים בתוך ה-WSL, שחלקם היוו חולשות אבטחה בתוכו. עם זאת, לא נוצלו כבעיות אבטחה, שכן רובן התגלו על ידי משתמשים שהשתמשו בגרסת ה-Preview וחוו BSoD.

בנוסף לבעיות אלו, ה-WSL פותח לנו אפשרויות תקיפה חדשות במערכת ההפעלה, שכן יש מתוכו גישה ישירה לכל הקבצים שקיימים גם ב-Windows, גישה לרשת, הרצה של קבצי exe מתוכו על ידי מנגנון ה-IPC של ה-WSL ועוד. בנוסף לכך, היכולת לנתח ולבדוק אם קבצים שמופעלים דרך ה-WSL הם קבצים זדוניים בעייתית יותר, שכן מורצים מתוך Windows ואין להם שום מבנה נתונים או PE הקשור בהם (אין ntdll, PEB, TEB, KSHARED_USER_DATA). קיימות מעל 200 System Calls שניתן לחקור ולנסות להשיג דרכם privilege escalation. פרט לכך, ניתן ליצור Pico Process דרך ה-LXSS Manager ללא צורך בהרשאות גבוהות.

מקום נוסף שניתן להסתכל עליו מבחינת אבטחה הינו ה-Pico Provider. ה-Pico Provider, וכל ה-System Calls שמסופקות על ידו מוגנות על ידי ה-Patch Guard. דבר זה מונע גם את היכולת לביצוע hooking לאותם System Calls ולפונקציות של ה-Pico Provider, וכן יצירת Pico Provider נוסף שיפעל במקום הדרייברים Lxss.sys ו-LxCore.sys בלתי אפשרית בשל ה-Patch Guard. ה-Pico Provider חייב להיטען מאוד מוקדם על מנת להשתמש ב-API, ועל מנת לבצע את זה עליו להיות חתום על ידי Microsoft. אולם, כן ניתן לעקוף מנגנונים אלו על ידי עקיפה של ה-Patch Guard בדרך כזו או אחרת.

סיכום

Microsoft הביאו מספר פיצ'רים די מגניבים שבאו לידי ביטוי ב-Windows 10, ובעצם שינו את הצורה שתהליכים נראים במערכת ההפעלה הזו - דבר הפותח עולם חדש של מחקר, ידע ופיתוח. למשל, התשתית של ה-WSL מאפשרת תמיכה של כל מערכת הפעלה שהיא על Windows - כל מה שצריך זה Pico Provider נוסף לאותה מערכת הפעלה. וגם שווה לחשוב, האם אותם מנגנונים יכולים לבוא לשימוש לרעה ולאפשר הרצת קוד זדונית?

במאמר הבא, נסתכל כיצד ניתן להשתמש בתיאוריה שהוסברה במאמר זה: איך ניתן להשתמש בפונקציות שמספקת Windows כדי ליצור mininal processes ו-pico processes, כיצד ניתן להשתמש בתקשורת שה-WSL משתמש בה ועוד.

מקורות

1. הרצאה של Alex Ionescu מ-BlackHat 2016:

http://www.alex-ionescu.com/publications/blackhat/blackhat2016.pdf

2. Windows Internals, the 7th Edition
3. הבלוג של Microsoft על ה-WSL:

https://blogs.msdn.microsoft.com/wsl/