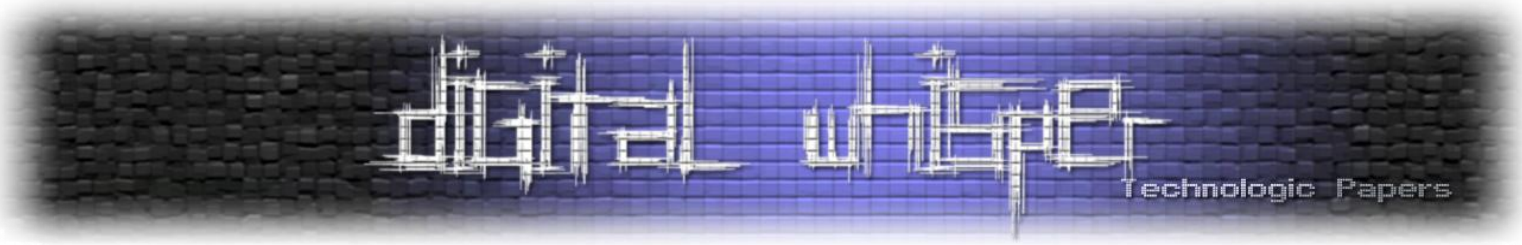


Syscalls, Direct Syscalls
וכל מה שביניהן

מאת לינוי שוקרון

הקדמה

לשמחתנו, קיימים היום פתרונות הגנה מתקדמים ומגוונים. לצערנו, ככל שפתרונות ההגנה מתקדמים, יחד איתם מתקדמות טכניקות התקיפה וההסתרה של תוקפים. המירוץ המתמשך הזה בין החתול והעכבר ככל הנראה לא ייגמר לעולם, והוא מוסבר בצורה מורחבת ועקבית במאמר המעולה של תומר דהן מגיליון 169 - "Catch Me if You Can". המאמר של תומר מעניק לנו מבט-על בכל הקשור להתפתחות ההגנות לצד התקיפות, ובמאמר זה רציתי לצלול לטכניקה אחת שהוא הזכיר, הלוא היא direct syscalls.

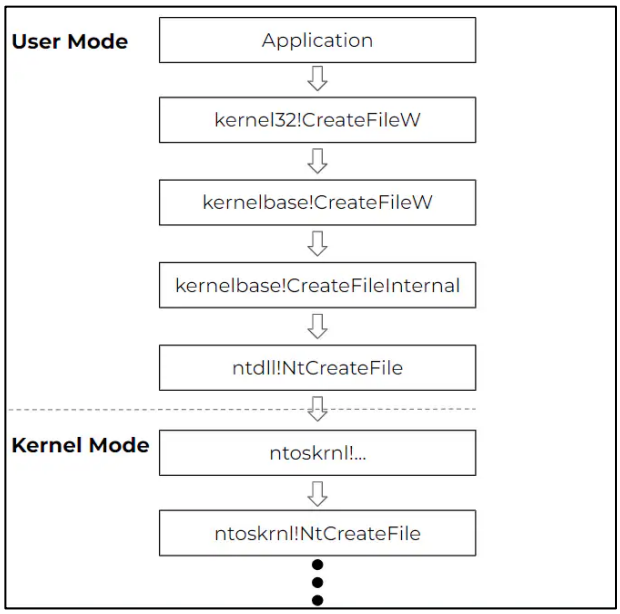
אז מה מחכה לנו? היום אנחנו נצלול לנבכי מנגנון syscall (קיצור ל-system call) במערכת הפעלה Windows. נסביר מה זה המנגנון הזה, מה הצורך בו, איך המנגנון השתנה לאורך השנים ובין הגרסאות השונות, וכמובן נציג את טכניקת התקיפה direct syscalls. לבסוף נסקור את הקוד של הכלי Hell's Gate, שמאפשר לתוקפים לממש את הטכניקה בקלות יתרה.

בואו נתחיל!

מה הן syscalls ב-Windows?

בהגדרה הפשוטה ביותר, syscall היא פקודת אסמבלי שמבצעת את הקפיצה מ-user-mode לקרנל על מנת לבצע שם משימה כלשהי. הרי גישה לחומרה, כתיבה לדיסק, שליחת פקטות תקשורתיות וכו' הן פעולות שיכולות להיות מבוצעות אך ורק ע"י הקרנל, אז תוכניות ב-user-mode צריכות ל"קפוץ" לקרנל לשם כך.

באופן כללי, תוכניות שרצות ב-user-mode מתקשרות עם הקרנל באמצעות פונקציות Windows API (שנמצאות למשל ב-kernel32.dll), שקוראות לפונקציות native API (כפי שניתן למצוא ב-ntdll.dll), והן מממשות את פקודת האסמבלי syscall שמטריגה את הקפיצה לקרנל, שם רצות פונקציות שמבצעות את העבודה בפועל - אלו הן הפונקציות ב-ntoskrnl.exe. התרשים הבא מציג flow של syscall במערכת:



[מקור: https://www.paloaltonetworks.com/blog/security-operations/a-deep-dive-into-malicious-direct-syscall-detection/]

מה היה בעבר ומה קיים כיום?

לפני שפקודת ה-syscall פרצה לחיינו, הקפיצה לקרנל הייתה נראית טיפה שונה. אפשר לקרוא לזה שיטת ה-int 0x2E. זוהי דרך ישנה שמתבססת על interrupts במערכת ההפעלה. כפי שהשם מסגיר, הכוונה ב-interrupts, היא ל-"הפרעות", שמאותתות למערכת ההפעלה שהן מצריכות טיפול מיידי. קיימים מספר סוגי interrupts, ביניהם hardware, software, exception. לחיצה על מקשי המקלדת או הזזת העכבר הן דוגמאות קלאסיות ל-hardware interrupts למשל. כמובן שכל ה-interrupts מאונדקסים וכך מערכת ההפעלה יודעת להבחין ביניהם ולטפל בהם בהתאם.

רגע לפני שנמשיך בהסבר נזדקק לקצת מונחי רקע שיעזרו לנו להבין את התהליך בצורה הוליסטית: קיים מבנה קרנלי שנקרא Interrupt Descriptor Table, או בקיצור IDT, שהוא למעשה מערך בו כל תא מכיל כתובת של פונקציה. בפועל זה קצת יותר מורכב מזה, המבנה IDT מכיל רשומות שהן descriptors - מבנה נתונים מורכב יותר שמתורגם בסוף לכתובת.

IDT החליף מבנה נתונים דומה אך ישן יותר, שנקרא Interrupt Vector Table, או בקיצור IVT, שבאמת פשוט הכיל כתובות של פונקציות (ניתן לקרוא על ההבדלים במאמר הבא). לצורך הנוחות, נתייחס לרשומה ב-IDT

ככתובת של פונקציה - הפונקציה הזו מכונה Interrupt Service Routine, או בקיצור ISR, ותפקידה לטפל ב-interrupt שקרה.

אחרי שהבנו את כל מונחי הרקע, בואו נעבור על השלבים לטיפול ב-interrupt. כאשר קופץ interrupt כלשהו, מערכת ההפעלה צריכה לגשת ל-IDT. איך היא יודעת היכן המערך הזה יושב? קיים אוגר מיוחד שנקרא IDTR שמצביע על ה-IDT. אז היא ניגשת לאינדקס המתאים ב-IDT, במקרה שלנו 0x2E, ושם נמצאת הכתובת של ה-ISR הרלוונטית, הלוא היא הפונקציה הקרנלית KiSystemService. היא קוראת בהמשך לפונקציה קרנלית נוספת - KiFastCallEntry, תזכרו את זה!

אגב, WinDbg יכול לעזור לנו להציג את ה-IDT בצורה נוחה ומסודרת באמצעות ה-extension הבא:

```
lkd> !idt -a

Dumping IDT:

00:      805421a0 nt!KiTrap00
01:      8054231c nt!KiTrap01
02:      Task Selector = 0x0058
03:      80542730 nt!KiTrap03
04:      805428b0 nt!KiTrap04
05:      80542a10 nt!KiTrap05
06:      80542b84 nt!KiTrap06
07:      805431fc nt!KiTrap07
08:      Task Selector = 0x0050
09:      80543600 nt!KiTrap09
0a:      80543720 nt!KiTrap0A
0b:      80543860 nt!KiTrap0B
0c:      80543ac0 nt!KiTrap0C
0d:      80543dac nt!KiTrap0D
0e:      805444a8 nt!KiTrap0E
0f:      805447e0 nt!KiTrap0F
10:      80544900 nt!KiTrap10
11:      80544a3c nt!KiTrap11
12:      Task Selector = 0x00A0
13:      80544ba4 nt!KiTrap13
14:      805447e0 nt!KiTrap0F
15:      805447e0 nt!KiTrap0F
16:      805447e0 nt!KiTrap0F
17:      805447e0 nt!KiTrap0F
18:      805447e0 nt!KiTrap0F
19:      805447e0 nt!KiTrap0F
1a:      805447e0 nt!KiTrap0F
1b:      805447e0 nt!KiTrap0F
1c:      805447e0 nt!KiTrap0F
1d:      805447e0 nt!KiTrap0F
1e:      805447e0 nt!KiTrap0F
1f:      806e610c hal!HalpApicSpuriousService
20:      00000000
21:      00000000
22:      00000000
23:      00000000
24:      00000000
25:      00000000
26:      00000000
27:      00000000
28:      00000000
29:      00000000
2a:      805419ce nt!KiGetTickCount
2b:      80541ad0 nt!KiCallbackReturn
2c:      80541c80 nt!KiSetLowWaitHighThread
2d:      8054260c nt!KiDebugService
2e:      80541451 nt!KiSystemService
```

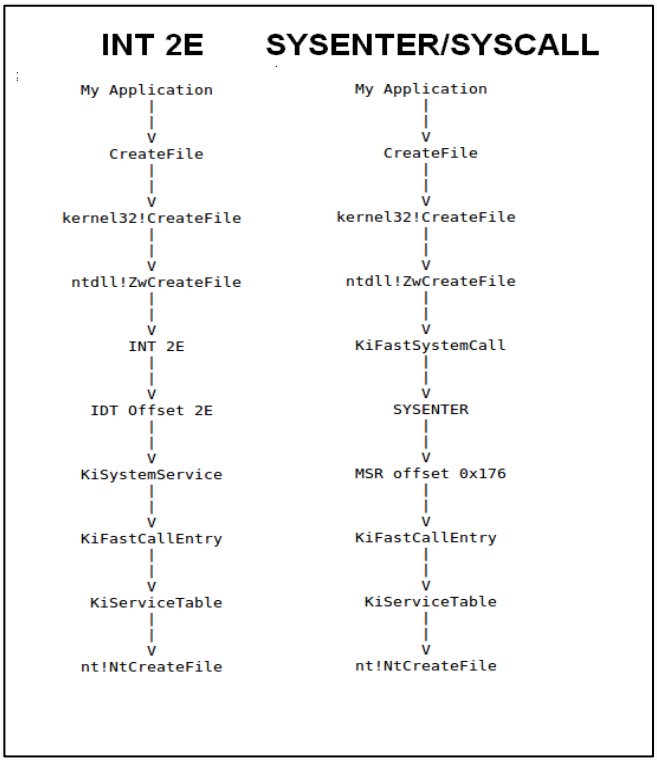
[מקור: https://trapframe.github.io/just-enough-kernel-to-get-by-2/]

היות ומדובר בסוף בשיטה לטיפול ב-interrupts, היא כוללת גם את כל כאבי-הראש הכלולים בכך, לכן מטעמי יעילות ורצון לקצר את זמן הקפיצה לקרנל, שכן יש כמות אדירה של קפיצות לקרנל בכל רגע נתון, הגענו לפתרונות syscall ו-sysenter (פתרונות מקבילים של AMD ואינטל בהתאמה).

Syscall ו-sysenter הן פקודות אסמבלי שגורמות ל-CPU לקרוא ערך ששמור באוגר מיוחד, אחד מאוגרי MSR, והערך הזה הוא הכתובת של הפונקציה הקרנלית שאמורה לטפל בקריאות לפונקציות בקרנל, זו אותה KiFastCallEntry שפגשנו קודם! ה-CPU טוען את הכתובת של הפונקציה הזו לאוגר RIP (השורה הבאה לרוץ), כדי שנקפוץ אליה בעצם, והיא תטפל ב-system call. אגב השיטה הזו מהירה פי 3 משיטת ה-interrupts!

בכל אופן, בגלל ש-ntdll.dll הוא האחרון בשרשרת הקריאות ב-user-mode, הוא אמור להיות ערוך לקבל את כל סוגי הקפיצות לקרנל. ההחלטה באיזו דרך להשתמש מתקבלת בזמן ריצה על סמך סוג ה-CPU ופרמטרים נוספים. ובכל זאת, במערכות x64 חדישות, רובן ככולן מממשות את פקודת ה-syscall (משום שיש לה תמיכה רחבה יותר).

כדי לעשות קצת סדר בבלאגן, הנה תרשים המציג את שתי דרכי הקפיצה לקרנל זו לצד זו:

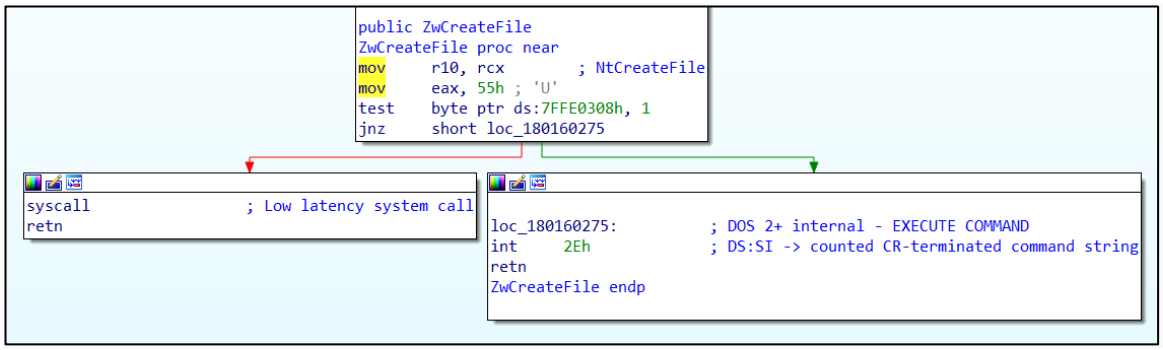


[מקור: https://trapframe.github.io/just-enough-kernel-to-get-by-2/]

כפי שהסברתי ושניתן לראות בתרשים, שני התהליכים מתכנסים בשלב הקריאה ל-KiFastCallEntry. משלב זה ואילך אנחנו בקרנל. זהו! סיימנו עם כל הקפיצות למיניהן. פה נכנס אלמנט שטרם דיברנו עליו - איך המערכת תדע איזו פונקציה אנחנו רוצים להריץ בקרנל? זה הזמן להגיד שהפקודה syscall חייבת להגיע בצימוד ערך שנקרא System Service Number, או בקיצור SSN. ערך זה ייחודי לכל פונקציה ומשתנה בין גרסאות (ותתי-גרסאות) שונות של מערכת ההפעלה.

הנה דוגמה לפונקציית native API, חשוב לשים לב למבנה שלה (זה נקרא גם syscall stub) כי הוא די קבוע בין כל הפונקציות הללו:

- מעבירים את RCX ל-r10 (הכנת הפרמטרים של הפונקציה לעיבוד בקרנל ב-x64).
- דוחפים את ה-SSN לאוגר EAX.
- בודקים האם מערכת ההפעלה תומכת בפקודת syscall:
  - במידה שכן, מבצעים פקודת syscall.
  - במידה שלא, עוברים לקרנל באמצעות שיטת ה-int 0x2E.



אוקיי, ועכשיו נשאלת השאלה מה KiFastCallEntry עושה עם ה-SSN? היא ניגשת למבנה טבלאי סופר חשוב שנקרא System Service Descriptor Table, או בקיצור SSDT, שממפה בין ערכי SSN לכתובות של פונקציות בקרנל. אגב, בתרשים הקודם, היכן שמופיע KiServiceTable, הכוונה למעשה היא למצביע ל-SSDT.

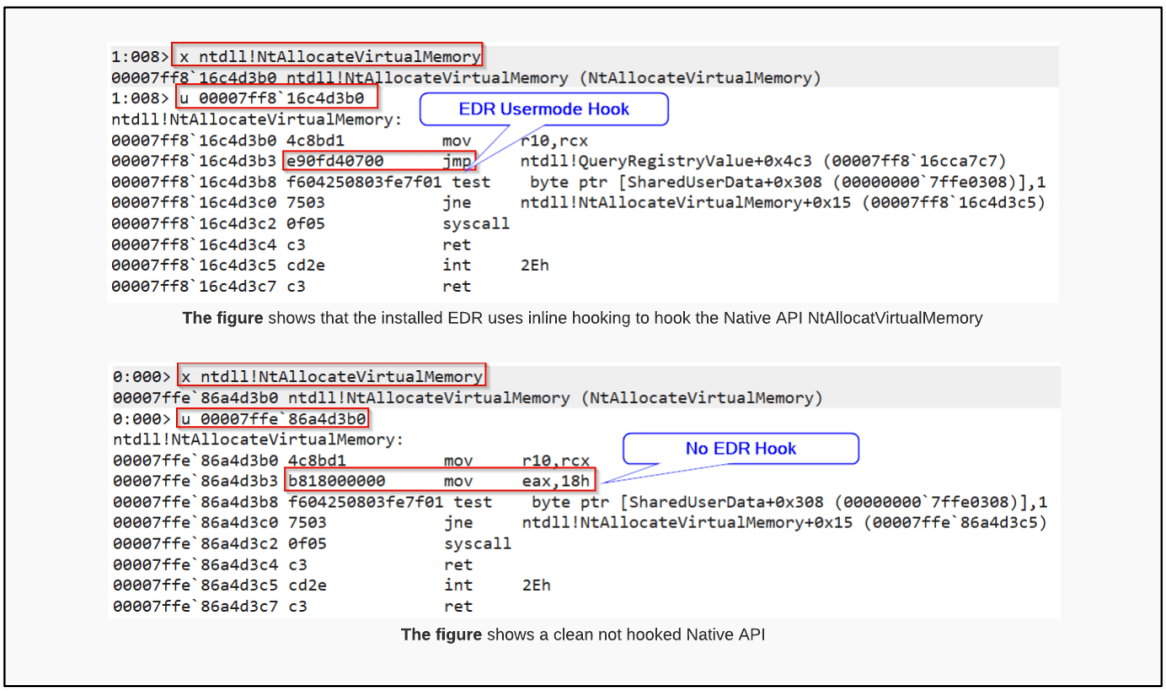
עד כאן, התהליך המפואר של קפיצה לקרנל לצורך ביצוע פעולה פשוטה כגון CreateFile. עכשיו נשאר להבין איך הגנות ותקיפות משתמשות ומנצלות אותו.

איך מוצרי הגנה "משתמשים" במנגנון?

באופן כללי, הגיוני ומתבקש שמוצרי הגנה יתעניינו במנגנון syscall, משום שהוא נמצא במקום אסטרטגי מבחינת מתווה תקיפה. זאת אומרת, הנחת היסוד היא שכל קריאה לפונקציה שתוקף מבצע אמורה לעבור בדרך לקרנל בתחנה של פקודת ה-syscall תחת פונקציה כלשהי ש-ntdll.dll מייצא. על כן, מוצרי הגנה דוגמת EDR בוחרים לשים את ההגנות והבדיקות שלהם בדיוק במקום הזה.

רוב הפעמים, מוצרי הגנה ישימו hook על פונקציות מעניינות ש-ntdll.dll מייצא, כלומר הם יערכו את תחילת הפונקציה ויפנו את ריצת הקוד לקוד ייעודי שלהם. בקוד שלהם הם יערכו בדיקות ויגיעו למסקנה האם הקריאה לפונקציה לגיטימית או חשודה: האם מקור הקריאה לגיטימי? האם התדירות שקריאה כזו קורית הגיונית? וכו'. אם הכל כשורה, הריצה תחזור למקום המקורי ותבוצע פקודת ה-syscall בהתאם.

להלן השוואה בין פונקציה שהיא hooked ע"י EDR לבין פונקציה מקורית "נקייה":



[מקור: https://redops.at/en/blog/direct-syscalls-vs-indirect-syscalls]

הערת אגב - במקרה זה, ה-EDR דרס את השורה בה דוחפים את ה-SSN ל-EAX. כמובן שאם ריצת הקוד תחזור לפקודת ה-syscall מבלי שיידחף ה-SSN ניתקל בבעיה בהמשך, בשלב בשליפה מה-SSDT. במקרה כזה לצורך העניין, ובהנחה שמדובר בקריאה לגיטימית, באחריות ה-EDR לדאוג שבסוף הקוד שלו ולפני החזרה לפונקציה המקורית תהיה דחיפה של אותו SSN ל-EAX, על מנת להבטיח המשך ריצה תקין של התוכנית.

איך תוקפים עוקפים את מוצרי ההגנה?

אנחנו יכולים להיות בטוחים שתוקפים מאוד יצירתיים כשזה נוגע למעקף מוצרי הגנה. כ"תגובה" ל-hooks שמוצרי הגנה שמים לפני syscalls, תוקפים החליטו לקחת עניינים לידיים ולקרוא בעצמם לפקודת syscall. אממה, בעיה מהותית שהזכרנו היא שערכי ה-SSN דינאמיים ומשתנים בין גרסאות שונות של מערכת ההפעלה, ועל כן מאוד קשה לכתוב קוד זדוני מבלי לדעת בדיוק מהי גרסת המערכת המטורגטת. כן אגיד שניתן למצוא באינטרנט טבלאות עם ערכי SSN עבור גרסאות מסוימות, אבל בהנחה ותוקף רוצה לייצר פוגען בכוונה להפיץ אותו באופן רחב, זה טרחני וקשה עד בלתי אפשרי לשזור בקוד hard-coded SSNs עבור כל תרחיש אפשרי. הפתרון לדבר הזה די פשוט - חילוץ של ערכי SSN מ-ntdll.dll באופן דינאמי בזמן ריצה. זה בדיוק מה שהכלי Hell's Gate מנגיש לנו.

נסקור את הקוד של הכלי כדי להבין את שלבי התהליך, נתחיל ב-main:

1. תחילה, משיגים את הכתובת של ה-Thread Environment Block, או TEB. הוא מכיל מידע אודות ה-thread הנוכחי שרץ, בין היתר תחת איזה תהליך הוא רץ.
2. באמצעות ה-TEB אנחנו יכולים לגשת למבנה הבא שמעניין אותנו הלוא זה ה-Process Environment Block, או PEB. זה מבנה שמכיל הרבה מידע אודות התהליך, למשל האם התהליך רץ תחת debugger, אילו מודולים טעונים אליו וכדומה. בקיצור, זה זהב לתוקפים בהרבה מקרים ושימושים שונים:

```
INT wmain() {
        PTEB pCurrentTeb = RtlGetThreadEnvironmentBlock();
        PPEB pCurrentPeb = pCurrentTeb->ProcessEnvironmentBlock;
```

[מקור: https://github.com/am0nsec/HellsGate/blob/master/HellsGate/main.c]

3. באמצעות ה-PEB, ניגש למבנה PEB_LDR_DATA, שמצביע לרשימה מקושרת של המודולים הטעונים לתהליך שלנו. נרוץ על הרשימה עד שנגיע ל-ntdll.dll, ואז תחת השדה DllBase נמצא את ה-base address שלו בזיכרון:

The **LDR_DATA_TABLE_ENTRY** structure is defined as follows:

syntax

```
typedef struct _LDR_DATA_TABLE_ENTRY {
    PVOID Reserved1[2];
    LIST_ENTRY InMemoryOrderLinks;
    PVOID Reserved2[2];
    PVOID DllBase;
    PVOID EntryPoint;
    PVOID Reserved3;
    UNICODE_STRING FullDllName;
    BYTE Reserved4[8];
    PVOID Reserved5[3];
    union {
        ULONG CheckSum;
        PVOID Reserved6;
    };
    ULONG TimeDateStamp;
} LDR_DATA_TABLE_ENTRY, *PLDR_DATA_TABLE_ENTRY;
```

[מקור: https://learn.microsoft.com/en-us/windows/win32/api/winternl/ns-winternl-peb_ldr_data]

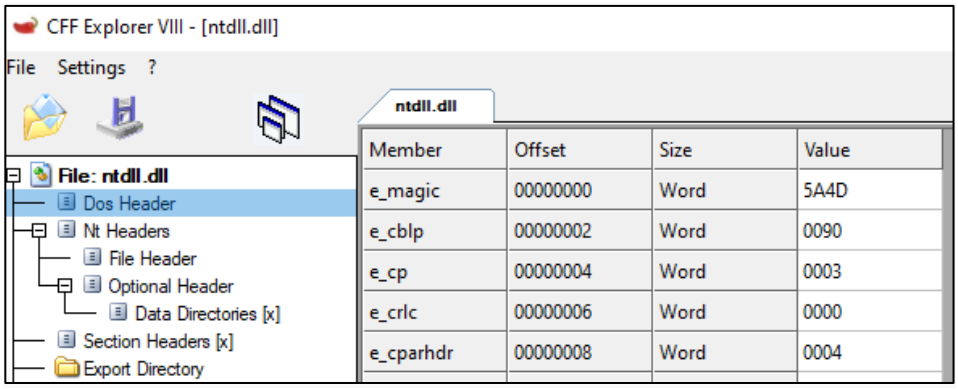
4. לאחר שהשגנו את ה-base address של ntdll.dll, נפרסר אותו צעד-צעד ביחד, אל דאגה. חברנו הטוב, CFF Explorer יעזור למי שאוהב לראות את הדברים בצורה ויזואלית. CFF הוא כלי בעזרתו ניתן לערוך ולפרסר קבצי הרצה בצורה ויזואלית ונוחה.

- מצורף קטע הקוד הרלוונטי מ-Hell's Gate:

```
102     BOOL GetImageExportDirectory(PVOID pModuleBase, PIMAGE_EXPORT_DIRECTORY* ppImageExportDirectory) {
103             // Get DOS header
104             PIMAGE_DOS_HEADER pImageDosHeader = (PIMAGE_DOS_HEADER)pModuleBase;
105             if (pImageDosHeader->e_magic != IMAGE_DOS_SIGNATURE) {
106                     return FALSE;
107             }
108
109             // Get NT headers
110             PIMAGE_NT_HEADERS pImageNtHeaders = (PIMAGE_NT_HEADERS)((PBYTE)pModuleBase + pImageDosHeader->e_lfanew);
111             if (pImageNtHeaders->Signature != IMAGE_NT_SIGNATURE) {
112                     return FALSE;
113             }
114
115             // Get the EAT
116             *ppImageExportDirectory = (PIMAGE_EXPORT_DIRECTORY)((PBYTE)pModuleBase + pImageNtHeaders->OptionalHeader.DataDirectory[0].VirtualAddress);
117             return TRUE;
118     }
```

[מקור: https://github.com/am0nsec/HellsGate/blob/master/HellsGate/main.c]

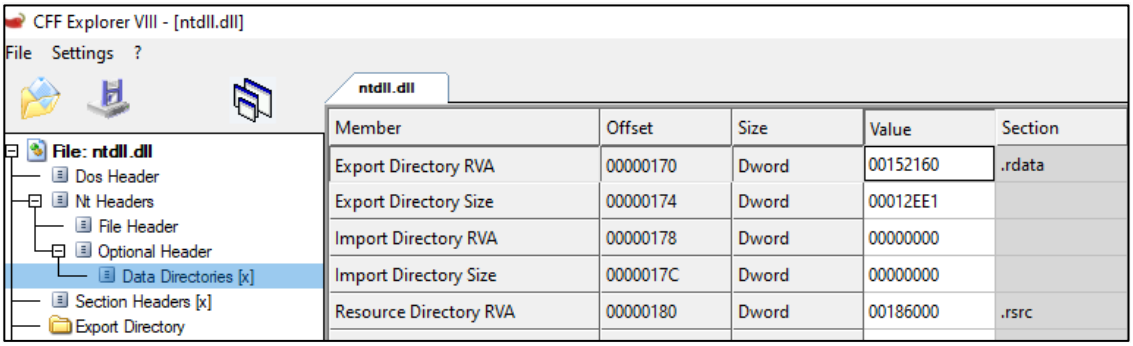
- כמו כל קובץ הרצה (PE) גנרי, הוא מתחיל ב-DOS Header. השדה הראשון ב-header הוא ה-magic number שאומר לנו מה סוג הקובץ. Hell's Gate מוודא שמדובר בקובץ הרצה, מה שכמובן נכון.



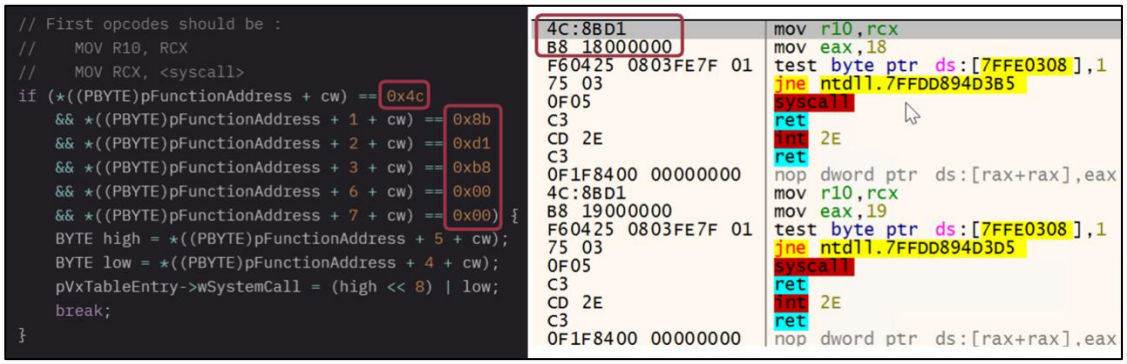
- השדה האחרון ב-dos header נקרא e_lfanew, והוא מכיל את ה-offset ל-Nt headers בתוך הקובץ. Hell's Gate לוקח את ה-base address של ntdll.dll ומוסיף לה את ה-offset הזה כדי להגיע לתחילת ה-Nt headers. אגב אפשר לראות איך אנחנו מתקדמים לאורך ה-side panel ב-CFF.

- לסיום, נגיע ל-optional header בתוך ה-Nt headers, ושם יש לנו טבלת מידע (data directory) שהשדה הראשון בה הוא סוג של offset (זה נקרא RVA) לטבלת ה-exports. זה מה שחיפשנו! בטבלה הזו יש את הכתובות של כל הפונקציות ש-ntdll.dll מייצא.

5. כעת שסיימנו לפרסר את ntdll.dll ויש בידינו כתובות של פונקציות, נוכל לפרסר כל פונקציה, הרי המבנה של ה-syscall stub הוא קבוע, וכך נחלץ את ערכי ה-SSN לכל הפונקציות שנרצה:

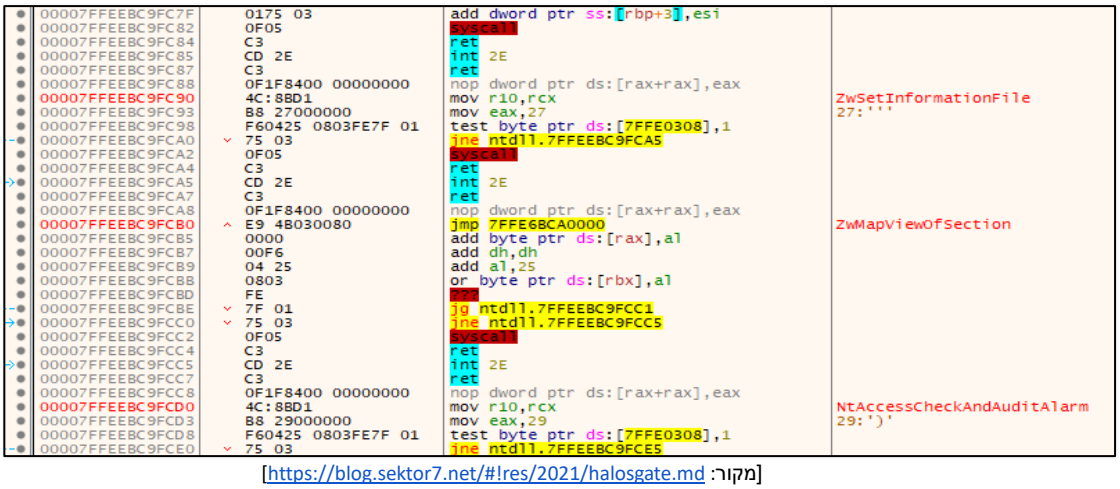


6. אממה? אמרנו שהפונקציה יכולה להיות hooked ע"י מוצר הגנה כלשהו. גם את זה Hell's Gate בודק. מצד שמאל מופיע הקוד של Hell's Gate, ומצד ימין מופיע פרסור של פונקציה נקייה (ללא hook) לדוגמה. ניתן לראות את התבניתיות בשתי הפקודות הראשונות בתחילת הפונקציה, מה שמתורגם ל-opcodes קבועים - אלו הם ה-opcodes שהכלי מחפש בצד שמאל:



[מקור: https://redops.at/en/blog/exploring-hells-gate]

7. אבל מה קורה אם באמת נתקלנו בפונקציה hooked? קיימת שיטה נוספת שנקראת Halo's Gate שבאה לכסות על הפער הזה שנותר אחרי Hell's Gate, והיא בעצם מסתמכת על כך שערכי ה-SSN בפונקציות ש-ntdll.dll מייצא הם בסדר עוקב, הנה דוגמה להמחשה:



[מקור: https://blog.sektor7.net/#!res/2021/halosgate.md]

קודם כל ובאופן כמעט מיידי ניתן לראות שהפונקציה ZwMapViewOfSection היא hooked (מתחילה ב-jmp ולא בפקודה שאנחנו מכירים mov r10, rcx). שימו לב לשתי הפונקציות ש"עוטפות" אותה - זו שמעליה עם SSN 27 וזו שמתחתיה עם SSN 29. אז מה ששיטת Halo's Gate עושה, זה בעצם לפרום את הדרך החוצה, ולהשלים את החסר על סמך הנחת היסוד הזו שכל הערכים עוקבים. כל מה שצריך זה למצוא פונקציה "שכנה", ולחפש את אותו דפוס קבוע של syscall stub (של פונקציה נקייה כמובן).

הנה קטע הקוד שעושה את זה מפרוייקט גיט שמציע PoC ל-Halo's Gate:

```
 //if hooked check the neighborhood to find clean syscall
if (*((PBYTE)pFunctionAddress) == 0xe9) {
        for (WORD idx = 1; idx <= 500; idx++) {
                // check neighboring syscall down
                if (*((PBYTE)pFunctionAddress + idx * DOWN) == 0x4c
                        && *((PBYTE)pFunctionAddress + 1 + idx * DOWN) == 0x8b
                        && *((PBYTE)pFunctionAddress + 2 + idx * DOWN) == 0xd1
                        && *((PBYTE)pFunctionAddress + 3 + idx * DOWN) == 0xb8
                        && *((PBYTE)pFunctionAddress + 6 + idx * DOWN) == 0x00
                        && *((PBYTE)pFunctionAddress + 7 + idx * DOWN) == 0x00) {
                        BYTE high = *((PBYTE)pFunctionAddress + 5 + idx * DOWN);
                        BYTE low = *((PBYTE)pFunctionAddress + 4 + idx * DOWN);
                        pVxTableEntry->wSystemCall = (high << 8) | low - idx;

                        return TRUE;
                }
                // check neighboring syscall up
                if (*((PBYTE)pFunctionAddress + idx * UP) == 0x4c
                        && *((PBYTE)pFunctionAddress + 1 + idx * UP) == 0x8b
                        && *((PBYTE)pFunctionAddress + 2 + idx * UP) == 0xd1
                        && *((PBYTE)pFunctionAddress + 3 + idx * UP) == 0xb8
                        && *((PBYTE)pFunctionAddress + 6 + idx * UP) == 0x00
                        && *((PBYTE)pFunctionAddress + 7 + idx * UP) == 0x00) {
                        BYTE high = *((PBYTE)pFunctionAddress + 5 + idx * UP);
                        BYTE low = *((PBYTE)pFunctionAddress + 4 + idx * UP);
                        pVxTableEntry->wSystemCall = (high << 8) | low + idx;

                        return TRUE;
                }

        }
```

[מקור: https://github.com/trickster0/TartarusGate/blob/master/HellsGate/main.c]

תחילה בודקים האם הפונקציה שאנחנו מצביעים עליה מתחילה ב-e9 שזה jmp opcode. במידה שכן, יש לנו עסק עם פונקציה שהיא hooked. אז מתחילים באיטרציה כלפיי מטה, ומחפשים כתובת שבה מתחיל syscall stub נקי (עם רצף ה-opcodes מקודם). אם לא נמצאה התאמה בטווח החיפוש כלפיי מטה, עוברים לבדוק גם כלפיי מעלה. לבסוף, בהנחה שמצאנו פונקציה נקייה, מחשבים את כמות ה"צעדים" שעברנו, למטה או למעלה, וכך משחזרים את ה-SSN של הפונקציה ה-hooked.

כמה הערות אודות Hell's Gate:

אומנם הצגתי רק את החלק של החילוץ הדינאמי של ערכי SSN מהזיכרון, אבל כמובן שלאחר שחולצו כל הערכים הרלוונטיים, Hell's Gate קורא לשתי פונקציות חיצוניות (HellsGate, HellDescent) שמבצעות את ה-direct syscall בהינתן ה-SSN שחולץ קודם לכן. דבר נוסף, Hell's Gate מוודא שמערכת ההפעלה עליה הוא רץ היא Windows 10. זה אומנם נשמע מגביל אם אנחנו חושבים כתוקף, אבל סטטיסטית Windows 10 היא מערכת ההפעלה הפופולארית ביותר בעולם (נכון לפברואר 2025), עם נתח שוק של כמעט 60% מבין כל גרסאות ה-Windows!

לסיכום

במאמר צללנו לעומקם של נושאים רבים ומגוונים ושוטטנו בקרביים של מערכת ההפעלה. זה ידע שאין לו סוף וספק אם אי פעם יהיה לו סוף. כמו שמוצרי ההגנה משתבחים עם השנים, כך גם שיטות התקיפה נהיות חשאיות ומתוחכמות יותר ויותר. מצד אחד אפשר להסתכל על זה בתור משהו שלילי, ומצד שני זו תחרות שדואגת בסופו של דבר ששני הצדדים יהיו בשיא שלהם. אם חסמו את הדלת ניתן בהחלט גם להיכנס מהחלון, כמו שהמאמר של תומר מסקר היטב.

לשאלות או הרחבות נוספות, מוזמנים להתחבר אליי!

מקורות מידע

- https://redops.at/en/blog/exploring-hells-gate
- https://blog.sektor7.net/#!res/2021/halosgate.md
- https://trapframe.github.io/just-enough-kernel-to-get-by-2/
- https://redops.at/en/blog/direct-syscalls-vs-indirect-syscalls
- https://codemachine.com/articles/system_call_instructions.html
- https://github.com/am0nsec/HellsGate/blob/master/HellsGate/main.c
- https://github.com/trickster0/TartarusGate/blob/master/HellsGate/main.c
- https://learn.microsoft.com/en-us/windows/win32/api/winternl/ns-winternl-peb_ldr_data
- https://www.paloaltonetworks.com/blog/security-operations/a-deep-dive-into-malicious-direct-syscall-detection/
- https://www.ired.team/offensive-security/defense-evasion/retrieving-ntdll-syscall-stubs-at-run-time#calling-syscall-stub
- https://medium.com/@amitmoshel70/intro-to-syscalls-windows-internals-for-malware-development-pt-1-b5bb0cd90c52