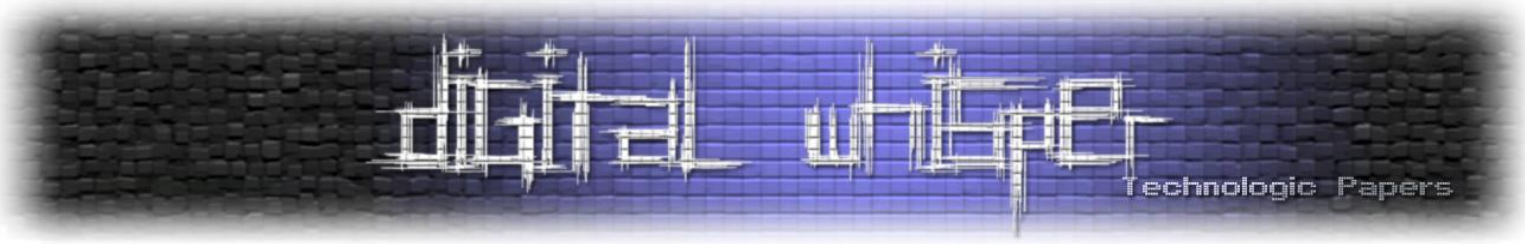


Rootkits- דרכי פעולה וטכניקות בשימוש

מאת Zerith (אורי)

במהלך השנים האחרונות, כל הנושא של Rootkits נעשה מאוד פופולארי בתחום הרוורסינג, Rootkits נפוצים בעיקר בתחום ה-Malware והגנות. מצד ה-Malware למטרות זדוניות – השגה ומסירה של מידע בצורה מאוד מסוות, או פשוט להשחתה. מצד ההגנות – יש שימוש מאוד רחב של ה-Rootkits בתחום ההגנות על משחקי הרשת, כגון GameGuard שטוען דרייבר (או Rootkit) על מנת לזהות פעולות כמו כתיבה לא רצויה לתהליך המשחק ומניעת שימוש בבוטים. במאמר זה אסביר לכם על: Rootkit – מהו? מהן השימושים שלו? איך ניתן לממש את כל הטכניקות האלו וכיצד ניתן להתגונן.

מהו Rootkit? הגדרה מקובלת של Rootkit, היא ערכה (kit) המכילה אפליקציות קטנות ושימושיות שמאפשרת לתוקף להשיג גישת "Root" למחשב הנתקף – הגישה הכי גבוהה שיכולה להינתן למשתמש. במילים אחרות, Rootkit הוא אוסף של אפליקציות ופונקציות אשר מאפשרים נוכחות קבועה ובלתי ניתנת לזיהוי של התוקף על המחשב של הנתקף, לרוב אלו נמצאים ברמת הקרנל (ה-Rootkit הוא דרייבר).

מונח המפתח פה הוא "בלתי ניתנת לזיהוי"- הסיבה העיקרית לרדת את כל הדרך אל רמת הקרנל, היא בשביל **הסוואה**. רוב הטכניקות בהן ה-Rootkit משתמשת הן בעצם כדי להסוות קוד או מידע באיזושהי צורה, למשל, החבאה של תיקיות, קבצים ותהליכים. טכניקות אחרות כוללות ציתות, הסנפה של פאקטים הנשלחים מכרטיס הרשת, ושליחת מידע בצורה **בלתי ניתנת לזיהוי**.

לפני תחילת המאמר, יש להזכיר כי כל המידע הניתן הוא משתנה מפלטפורמה לפלטפורמה ומגרסאות שונות של מערכת ההפעלה. רוב המידע פה מתייחס למערכת ההפעלה Windows NT עד Windows XP.

מבנה הדרייבר

נקודת הכניסה של דרייברים מוגדרת כ-"DriverEntry" – ממש כמו הפונקציה ()main או ()WinMain, גם לדרייברים יש נקודת הכניסה משלהם. ההגדרה שלה היא כזאת:

```
NTSTATUS
  DriverEntry(struct _DRIVER_OBJECT  *DriverObject,
PUNICODE_STRING  RegistryPath )
```

הפונקציה מקבלת שני ארגומנטים, ארגומט ראשון הוא הפוינטר למבנה הנתונים- DRIVER_OBJECT – המייצג את הדרייבר בקרנל, המבנה מכיל בין השאר את הטבלה של הפוינטרים לפונקציות המשמשות לתקשורת עם הדרייבר. התפקיד המרכזי של ה-DriverEntry בדרייברים רגילים הוא להקצות פוינטרים לפונקציות החשובות שקשורות בעיקר בתקשורת מקומית עם אותו הדרייבר (MAJOR_FUNCTIONS שמוגדרות ב-DRIVER_OBJECT) פונקציות אלו נקראות בשליחה של כל IRP מתאים (IRP- או " I/O Request Packet" הן חבילות מידע הנשלחות לדרייבר לביצוע משימות כאלה ואחרות), בין הפונקציות המוכרות יש את: IRP_MJ_READ IRP_MJ_WRITE וכו'. ברב המקרים, כאשר מדובר בדרייברים "רגילים", ה-DriverEntry פשוט מקצה פונקציות ל-MAJOR_FUNCTIONS שבשימושה וחוזרת.

הארגומנט השני, הוא פשוט מחרוזת שמייצגת את ה-Path למפתח הרג'יסטרי של הדרייבר. דוגמא:

```
#include "ntddk.h" //מכיל הגדרות לשימוש בדרייבר
VOID OnUnload( IN PDRIVER_OBJECT pDriverObject )
{
 DbgPrint(" OnUnload called.");
}

NTSTATUS DriverEntry( IN PDRIVER_OBJECT pDriverObject, IN
PUNICODE_STRING theRegistryPath )
{

// Allow the driver to be unloaded
 pDriverObject->DriverUnload = OnUnload;
 return STATUS_SUCCESS;
}
```

לשרוד את ה- Reboot

כנראה שהמחשב יבצע פעולת Reboot בשלב מסוים – על ידי המשתמש או על ידי תוכנה, וכדאי מאוד שה-Rootkit שלנו יעמוד בזה, אחרת הוא פשוט לא ירוץ. קיימות טכניקות רבות לשרידת ה-Reboot, אציג פה מספר מצומצם שלהן:

שימוש בדרייבר קיים

שיטה נפוצה לשרידת ה-Reboot היא "אינפקציה" של דרייבר קיים, דרייבר שאמור להטען בכל מקרה ב-Reboot. הכוונה היא השתלטות על דרייבר קיים או על קובץ אחד מהקבצים שהוא טוען. Rootkits רבים משתמשים (בצורה אירונית למדי) בדרייברים של אנטי-וירוסים על מנת לטעון את עצמם. למשל Rootkit ה-TDL3 המפורסם הלך עמוק ונמוך יותר, במקום להכניס את הקוד שלו בקבצים ב-File System כמו הגרסאות הקודמות שלו, החליט המפתח לכתוב את הקוד **ישירות לסקטורים של ה-Hard Disk,** והוא נמצא בסקטורים האחרונים של הדיסק הקשיח, ולא כחלק ממערכת הקבצים.

הדרייבר של ה-Rootkit השתמש בדרייבר קיים של ה-Miniport, כתב ב-rsrc section את ה-824 בתים שהולכים לטעון את הקוד מהדיסק הקשיח שהחליפו 824 בתים אחרים שהיו שם (כך גודל הדרייבר לא השתנה ולא היה ניתן לראות את השינוי, הבתים שהוחלפו נשמרים בדיסק הקשיח ויטענו בהמשך כדי שהדרייבר יתפקד כראוי) ושינה את נקודת הכניסה של הדרייבר לקוד שלו.

כך, בפעם הבאה שהמערכת עושה Reboot, הדרייבר נטען, מחכה לסיום טעינת מערכת הקבצים על ידי כך שהוא רושם את עצמו כ-File System Notification Routine, ואז טוען את הקוד עצמו מהסקטורים האחרונים של הדיסק.

רישום ה- Rootkit כשירות מערכת

ה-Rootkit יכול לרשום את עצמו כשירות מערכת אשר יטען עם הפעלתה של מערכת ההפעלה באמצעות הפונקציה OpenSCManager, CreateService. טכניקה זו דורשת רישום של מפתח רג'יסטרי, שיכול להיות מזוהה בקלות, אך ה-Rootkit יוכל להחביא את המפתח הנתון מהמערכת, כפי שאסביר בהמשך המאמר.

שינוי הקרנל או ה-Boot-Loader

ניתן לשנות את הקוד של הקרנל עצמו הנמצא על הדיסק על מנת לגרום לטעינתו של הדרייבר בהתחלת המערכת, כמו כן, ניתן כמובן לשנות את הקוד של ה-Boot Loader שטוען את הקרנל עצמו, אך יש להזהר במיוחד כשמשתמשים בטכניקות אלו משום שהן יכולות לגרום נזק תמידי למערכת ההפעלה או מניעת הפעלתה.

Kernel Hooks

שיטה מאוד נפוצה של Rootkits היא ביצוע Hooks בפונקציות הנמצאות ב-SSDT (System Service Dispatch Table). השימוש העיקרי של Rootkits בתחום ה-Hooks הוא הסוואה של כל מיני פעילויות שיכולות להראות חשודות ליישומי אנטי-וירוס וכלים שונים, יש להסוות את הקבצים של ה-Rootkit, את הקוד, את התקשורת וכו'. כמובן שנוכל פשוט לבצע Hooks על פונקציות שנקראות ב-User Mode כמו ReadFile/CreateFile כדי לשבש תוצאות אך מהיות האנטי-וירוס דרייבר, יש לו יתרון **מאוד** גדול עלינו, וזיהוי Hook כזה הוא כמעט ברור מאליו. כל מה שעליו לעשות הוא להשוות את הכתובת ב-IAT (במקרה של Hook IAT) או את ה-Prolog (חמשת הבתים הראשונים של הפונקציה) לערכים המקוריים. מפני שהרבה יותר קשה להערים על האנטי-וירוס ב-User Mode, נצטרך לבצע Kernel-Mode Hooks!

SSDT Hooks

ה-SSDT (או System Service Dispatch Table), למרות שמו המפחיד, הוא פשוט טבלה של פוינטרים המצביעים לפונקציות, לרוב טבלה זאת משומשת בUser Mode בצורה עקיפה. נניח שאנו קוראים לפונקציה CreateFileA בתוכניתנו. המימוש שלה ממש פשוט, בהתחלה הכנה של הפרמטרים – ואז קריאה ל- ZwCreateFile שנמצאת ב- ntdll. ומה המימוש של ZwCreateFile?

```
7C95D0AE   B8 25000000     MOV EAX,25
7C95D0B3   BA 0003FE7F     MOV EDX,7FFE0300
7C95D0B8   FF12            CALL DWORD PTR DS:[EDX]
7C95D0BA   C2 2C00         RETN 2C
```

הערך (או האינדקס) 25 הושם בEAX וישנה קריאה ל- KiFastSystemCall ([0x7FFE0300]):

```
7C95E510   8BD4            MOV EDX,ESP
7C95E512   0F34            SYSENTER
7C95E514   C3              RETN
```

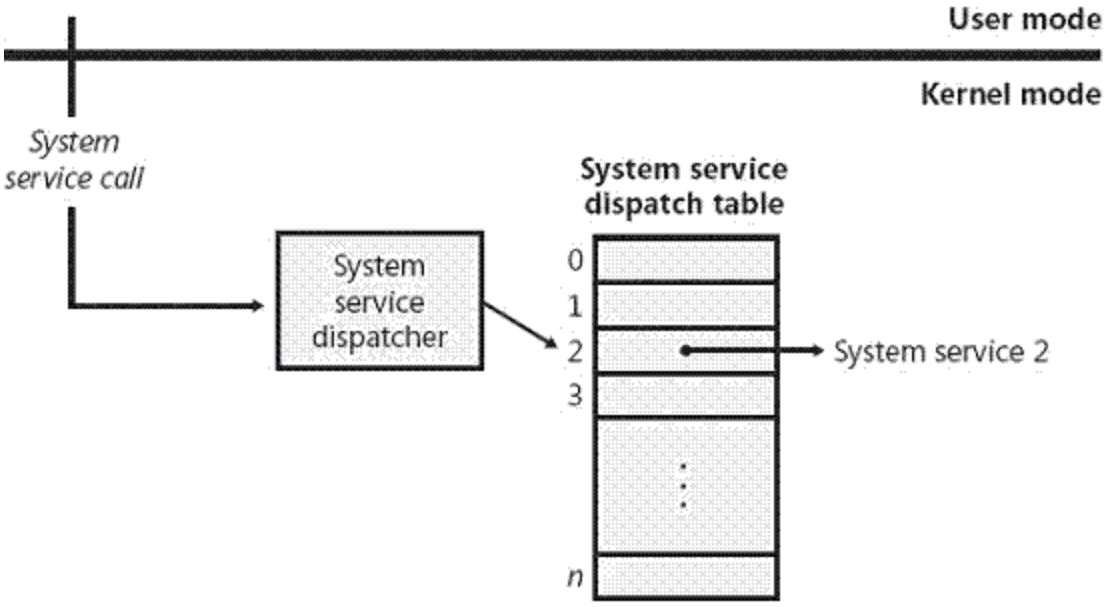
שמבצע:

1. שמירת כתובת המחסנית (לשימוש הפרמטרים).
2. שימוש בהוראה SYSENTER. ההוראה SYSENTER נכנסת לקרנל בצורה הבאה: קוראת באוגר ה- MSR (שהוא ייחודי לכל מעבד) את השדות הבאים:

- IA32_SYSENTER_CS – שדה המכיל את ה- Code Segment Selector של הקרנל שיוחלף בנוכחי (של User Mode), כאן אנחנו כבר עוברים לקרנל.
- IA32_SYSENTER_ESP – שדה המכיל את ה-Stack Segment Selector שיחליף את ה-User Mode Stack.
- IA32_SYSENTER_EIP – שדה המכיל את הכתובת של הפונקציה KiSystemService שהוא ה- Dispatch Handler שלנו. ומכאן ממשיכה הריצה ואנחנו בקרנל.

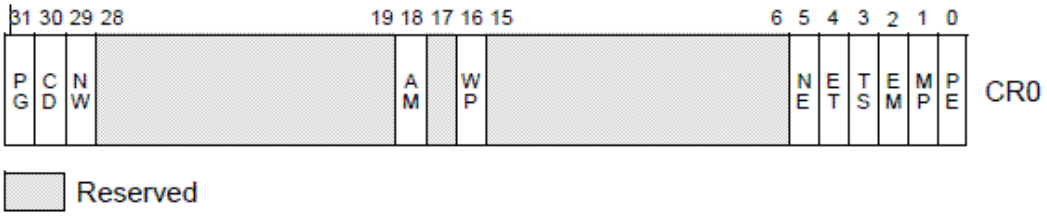
הפונקציה KiSystemService לוקחת את האינדקס (ששמנו ב-EAX ב-ZwCreateFile) של הפונקציה (CreateFile במקרה שלנו) ומוצאת את הפוינטר לפונקציה בטבלת ה-SSDT, ומשם ממשיכה הריצה עד שחוזרים לUser Mode, ומהפונקציה CreateFile.

ניתן לתאר את כל מה שהסברתי בתרשים הבא:



(מתוך
http://cfs4.tistory.com/upload_control/download.blog?fhandle=YmxvZzUxOTQ0QGZzNC50aXN0
(b3J5LmNvbTovYXR0YWNoLzAvMDEwMDAwMDAwMDAwLnBuZw%3D%3D

נמשיך לחלק המעניין, ה-Hook עצמו. מבחינת הדרייבר, ביצוע ה-Hook הוא ממש פשוט – כל מה שעלינו לעשות הוא לשנות את הפוינטר ב-SSDT באינדקס הנכון להצביע לפונקציה שלנו. אך ה-SSDT הוא אזור מוגן בזיכרון, לא ניתן לכתוב עליו! למזלנו, יש טריק פשוט שבזכותו ניתן לבצע SSDT Hooks בפשטות רבה - משנים את תוכן ה-0 Control Register (CR0). ה-CR0 נראה כך:



מה שמעניין אותנו הוא ביט מספר 16, WP - ביט זה קובע האם המעבד יוכל לכתוב לדפים אשר מוגדרים כ-Read-Only במערכת ההפעלה, לכן, אם נכבה אותו - נוכל לכתוב לכל דף בזיכרון. לכן נוכל לכתוב Code Snippet קטן שיכבה זמנית את הביט , יכתוב לאזור ה SSDT -ויחזיר את המצב לקדמותו, על מנת שלא ניצור בעיות בעתיד:

```
#pragma pack(1)
typedef struct ServiceDescriptorEntry
{
 unsigned int *ServiceTableBase;
 unsigned int *ServiceCounterTableBase;
 unsigned int NumberOfServices;
 unsigned char *ParamTableBase;
} ServiceDescriptorTableEntry_t, *PServiceDescriptorTableEntry_t;
#pragma pack()

__declspec(dllimport) ServiceDescriptorTableEntry_t
KeServiceDescriptorTable;

NTSTATUS HookSSDTEntry(WORD INDEX, DWORD Handler, DWORD* Original)
{
      __asm
      {
            cli
            push eax
            mov  eax, CR0
            and  eax, 0FFFEFFFFh
            mov  CR0, eax
            pop eax
        }
        *Original =
InterlockedExchange((PLONG)&(KeServiceDescriptorTable.ServiceTableBase[
INDEX]), (LONG)Handler);
        __asm
        {
              push eax
              mov eax, CR0
              or eax, NOT 0FFFEFFFFh
              mov CR0, eax
              pop eax
              sti
        }
}
```

יש לזכור שיש עוד טכניקות נוספות לכתיבה ל-SSDT ולהתחמק מההגנה, בחרתי פה לתת את הפשוטה ביותר. אסביר בקצרה את הקוד. בתחילתו ישנה הגדרה של המבנה ServiceDescriptorEntry, למה אנו צריכים להגדיר אותו? כי הדרך היחידה להגיע ל-SSDT הוא דרך ה- System Service **Descriptor** table, שהוא ServiceDescriptorEntry של ה-SSDT שמיוצא על ידי המערכת. אחר כך ייבוא והגדרת הסמל KeServiceDescriptorTable שדיברתי עליו קודם.

אז הגדרת הפונקציה HookSSDTEntry שמקבלת כארגומנט את האינדקס, הכתובת להחלפה והפוינטר למשתנה שיכיל את הכתובת המקורית של הפונקציה. הפונקציה מכבה את ה-Write-Protect bit ב-cr0, מחליפה את הערכים הרצויים ומשחזרת את אוגר ה-cr0 למצבו הקודם. הסיבה שבגללה ישנם ההוראות cli וsti (שמכבות ומדליקות את האפשרות ל-Interrupts בהתאמה) לפני ואחרי כיבוי והדלקת הגנות הדף, היא שאין ברצוננו שיפריעו לנו באמצע הקוד כשהגנות הדף לא פועלות. דבר זה יכול לגרום לשיבושים רבים במערכת.

ניתן להשתמש ב-SSDT Hooks לכל מיני מטרות כמו החבאת תהליכים, החבאת מפתחות ב-Registry (שלפעמים דרוש לאחסן מידע או לשרוד Reboot) באמצעות Hook ל-ZwOpenKey וכיוצא בזה. גילוי SSDT Hooks מאוד פשוט, כלים ואנטי וירוסים רבים פשוט עוברים על ה-Entries ב-SSDT לבדוק אם כל כתובת לא נמצאת במרחב הזיכרון השייך לדרייברים.

המאמר הבא מציג שתי טכניקות להחבאת ה-SSDT Hooks:

http://rootkit.com/newsread.php?newsid=922

ניתן לראות מימוש של Rootkit המשתמשת בSSDT Hooks להחבאת תהליכים בכתובת הבאה:

https://www.rootkit.com/vault/fuzen_op/HideProcessHookMDL.zip

IDT Hooks

ה-IDT (קיצור של Interrupt Descriptor Table) הוא טבלה, כמו ה-SSDT, המכילה פוינטרים ל-Handlers שיקראו בעת Interrupt. Interrupt (או בעברית: 'פסיקה') זאת פונקציה של מערכת ההפעלה שמאפשרת יעילות רבה בתקשורת עם חומרה. במקום שהתוכנה המקושרת לחומרה מסוימת, כמו למשל מקלדת, תעמוד בלופ אינסופי שתבדוק כל פרק זמן מסוים את האוגרים של המקלדת, סתם תבזבז לנו זמן יקר ומשאבים יקרים לא פחות, הומצא ה- PIC (Programmable Interrupt Controller) שמאפשר ל**חומרה עצמה** להגיד למערכת מתי **היא מוכנה**! נקח לדוגמא את המקלדת, כאשר המשתמש מקליד הצ'יפ של המקלדת מעדכן את האוגרים שלו ואז שולח אות ל-PIC שהמידע מוכן. ברגע זה ה-PIC קוטע את פעולת המעבד (אם ה-Interrupt Flag פועל) ושולח אותו להריץ מיד את ה-Interrupt Handler המתאים, לפי הכתובת שממופה ב-PIC.

כך Rootkit מסוים יכול לבצע IDT Hook בדומה ל-SSDT Hook ולשבש את ריצת הקוד לטובתו. בדרך זו גם ניתן לשנות או להקליט מידע המתקבל או נשלח מחומרה כמו מקלדת, מדפסת, כרטיס רשת, וכו'. שיטה זאת נפוצה בקרב Keyloggers (תוכנות המקליטות את הקלדות המשתמש ומתעדות אותם), הם רושמים Interrupt Handler משלהם במקום IRQ1 (שהיא של המקלדת, ניתן לראות רשימה מלאה של ההקצאות פה: http://www.simulationexams.com/SampleQuestions/a+_q4.htm) ומתעדים כל לחיצה על מקש.

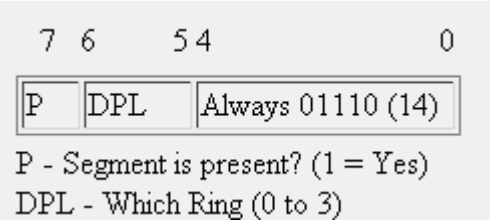
לשם שינוי ה-IDT עלינו להשיג את הכתובת שלה איכשהו וניתן להשתמש בהוראה SIDT כדי לאחסן את תוכן אוגר ה-IDTR (שמכיל פוינטר ל-IDT) בכתובת הרצויה. ה-IDTR מורכב מ-6 בתים: 2 הבתים התחתונים הם ה-Limit Field של האוגר – הגודל המירבי של ה-idt, שהוא בדרך כלל 2048 (או 256 Entries כשיודעים שכל Entry היא 8 בתים). 4 הבתים העליונים הם כתובת פיזית (32 ביט) אל ה- IDT Base Address.

```
struct idtr
{
unsigned short limit;
unsigned short LoBase;
unsigned short HiBase;
} __attribute__((packed));
```

בנוסף, יש לזכור את מבנה ה-Interrupt Descriptor Entries:

```
struct idt_entry
{
unsigned short base_lo;
unsigned short sel; //Kernel Segment Selector
unsigned char always0; // reserved
אסביר בהמשךunsigned char flags; //
unsigned short base_hi;
} __attribute__((packed));
```

שדה ה-Flags מייצג את מפת הביטים הבאה:



כש-DPL מייצג את הטבעת בה ה-Interrupt יכול להקרא, ring0 או ring3 (לפסיקות תוכנה). כך, ניתן לממש IDT Hook למקלדת בצורה הבאה:

```
#define KEYBOARD_ENTRY 1

struct idtr IDTR;
struct idt_entry KeyboardEntry;
struct idt_entry *IDT;

unsigned int HookKeyboard(unsigned int Handler)
{

      __asm sidt IDTR;  //Load IDTR with the idtr
      IDT = (idt_entry *)((IDTR.HiBase << 16) | IDTR.LoBase);
      unsigned int Original = ((IDT[KEYBOARD_ENTRY].base_hi << 16) |
IDT[2].base_loh); //store original entry.
      KeyboardEntry = &(IDT[KEYBOARD_ENTRY]);
      __asm cli; //disable interrupts so we won't crash.
      *(short *)(KeyboardEntry) = (short)Handler;
      *(short *)(KeyboardEntry+6) = (short)Handler >> 16;
      __asm sti; //re-enable interrupts
      return Original;
}
```

ניתן לראות מימוש מלא של Rootkit המשתמש ב-IDT Hook בשביל Keylogging בכתובת הבאה:

https://www.rootkit.com/vault/chpie/idt_src.zip

יש לזכור כי ה-IDT הוא ייחודי לכל מעבד, זאת אומרת שאם יש לנו מערכת עם 2 מעבדים, נצטרך לבצע את ה-IDT hook על כל מעבד בנפרד. (כמובן שניתן לעשות זאת באמצעות לולאה וקריאה לפונקציה KeSetTargetProcessorDpc)

לסיכום

זהו המאמר הראשון מתוך סדרת מאמרים על Rootkits, במאמר זה הסברנו מהו Rootkit, השימוש שלו ומטרתו. הצגנו את מבנה הדרייבר, Kernel Hooks – מה מטרתם, ולמה הם כדאיים. הצגנו נושאים נוספים כגון SSDT ו-IDT. במאמר הבא נסביר עוד מספר טכניקות המשמשות Rootkits למיניהם.

ביבליוגרפיה מומלצת: Programming the Windows Driver Model,Rootkit.com.