System Call Hooking

מאת שחק שלו

הקדמה

המאמר הבא עוסק בשיטת יישום חדשה של Hooking שמתחילה לצבור תאוצה לאחר שנצפתה בסוסים הטרויאניים Neurevt (הידוע גם כ-Betabot ו-Carberp) - קוד מקור. במאמר נבין את מנגנון ה- System Call של Windows וניישם את ה-Hook בעצמנו. את כל הקוד במאמר עם הסברים מלאים ניתן למצוא ב-Git Repository.

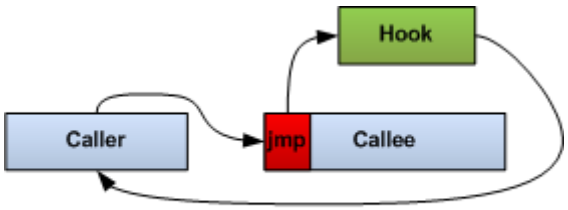
מהו Hooking?

נושא ה-Hooking בכללותו כבר נסקר כאן ב-Digital Whisper במאמר של **Zerith** מגליון **10** ובמאמר של אוריאל מלין על IAT Hooking בגליון 18, אנחנו נעבור בקצרה על תהליך ה-Hooking אך מומלץ מאד לקרוא את שני המאמרים בשביל לקבל תפיסה טובה לגבי הנושא.

אנחנו נרוץ טיפה על הנושא ונתמקד בשיטת ה-Inline Hooking שמנצלת את העובדה שברוב פונקציות ה-Windows API יש תחילה "פתיח" (Function Prologue) המורכב משלוש הוראות אסמבלי הנפרשות על 5 בייטים:

```
8B FF:    MOV EDI, EDI
55:       PUSH ESP
8B EC:    MOV EBP, ESP
```

בקצרה, העקרון ב-Inline Hooking הוא לדרוס את חמשת הבייטים הללו בקפיצה לקוד "זדוני" שהחדרנו לתהליך (הוראת קפיצה JMP באסמבלי תופסת גם היא 5 בייטים). כעת, בכל פעם שיקראו לפונקציה אותה ערכנו הקוד שלנו ירוץ לפני הקוד האמיתי.



[קרדיט - newgre.net]

הקוד החדש שאנחנו הוספנו יכול להיות כל מה שנבחר, בין אם זה שינוי קטן של פעולת הפונקציה המקורית או ביטול הפונקציה לגמרי. עיקרון חשוב הוא שעלינו לדאוג שלאחר סיום ריצת הקוד שלנו התוכנית תמשיך לפעול כרגיל, זאת אומרת שנצטרך לדאוג שהמחסנית תחזור לקדמותה ושאוגרים המשפיעים על המשך הריצה יכילו את הערכים המתאימים. מעולה, עכשיו אחרי שזה מאחורינו נתקדם:

על הרשאות במערכת Windows:

בשביל להבטיח שאפליקציות של המשתמש לא יוכלו לשנות מידע קריטי של מערכת ההפעלה, Windows משתמש בשני מצבי ריצה של המעבד: User Mode ו-Kernel Mode, או Ring3 ו-Ring0 בהתאמה. תוכנות של המשתמש רצות ב-User Mode וקוד של מערכת ההפעלה (דרייברים לדוגמא) ירוצו ב-Kernel Mode. המעבד (ולא מערכת ההפעלה) מאפשר גישה לכל מרחב הזיכרון במערכת ולכל סוגי הוראות המעבד תחת ריצה ב-Kernel Mode.

איך ניתן לתקשר בין שתי השכבות?

נניח והשתמשנו בתוכנה שלנו בפונקציה CreateFileA() שתיצור לנו קובץ. התוכנה שלנו רצה ב- User Mode, איך היא תוכל, אם כך, ליצור קובץ אם יצירת קבצים דורשת התערבות של הדרייבר Ntfs.sys ואמרנו שפעולה שכזאת אפשרית רק מ-Kernel Mode? בשביל זה הומצא מנגנון ה-System Call, המנגנון אחראי להעברת הריצה לביצוע הפונקציה לקרנל ולאחר סיום ביצוע הפונקציה, להחזיר את הריצה לתוכנה שלנו בUser-Mode להמשך ביצוע הקוד.

לכל פונקצייה שנקרא לה ותרוץ בקרנל יש מספר המייצג אותה בטבלה היושבת בקרנל הנקראת System Service Dispatch Table, עם המספר הזה נבצע את המעבר לקרנל ובכך נודיע למערכת ההפעלה איזו פונקציה ברצוננו לבצע.

> הערה: למעשה יש ארבעה טבלאות SSDT ב-Windows, כשהראשונה מאוכלסת בפונקציות Native, השניה בפונקציות GUI והשלישית והרביעית ריקות. Microsoft IIS יאכלס את הטבלה השלישית מתוך הארבע עם הדרייבר spud.sys במקרה והמוצר מותקן. אך רק הטבלה הראשונה רלוונטית אלינו ואליה מתכוונים בדר"כ כאשר מזכירים את ה-SSDT.

טבלת ה-SSDT מלאה בכל הפונקציות ובכתובת של כל אחת מהן בתוך ntoskrnl.exe שם הקוד האמיתי שלהם מתבצע, זאת אומרת שלמשל ntdll!NtCreateFile היא רק פונקצית מעטפת שבסופה הקוד האמיתי יתבצע ב-ntoskrnl!NtCreateFile.

System Calls

כל פונקציה (Windows API) אשר נקראת ומתבצעת בקרנל ולא בUser Mode נכנסת תחת הקטגוריה System Calls. לדוגמא: פונקציות של ניהול אובייקטים (NtCreateMutant, NtTerminateProcess וכו')

תתבצענה בקרנל מאחר שניהול אובייקטים ב-Windows נעשה בקרנל ופונקציות הדורשות התקשרות עם דרייברים גם כן יתבצעו בקרנל (הפונקציה NtCreateFile לדוגמא תדבר עם Ntfs.sys).

עד Windows 2000 המנגנון נראה כך:

```
1. MOV EAX, SyscallNumber
2. LEA EDX, [ESP+4]
3. INT 2Eh
4. RETN 4 * (Number of Arguments)
```

1. מוכנס לתוך האוגר EAX מספר הפונקציה אותה אנחנו רוצים לבצע.
2. מוכנס לתוך האוגר EDX מצביע לארגומנטים שמועברים לפונקציה.
3. מתבצעת הפסיקה 2E המסמלת כניסה לקרנל.
4. חזרה לפונקציה שקראה לנו.

ב-Windows XP, שונה המנגנון וכך הוא נראה עד היום:

```
1. MOV EAX, SyscallNumber
2. MOV EDX, 7FFE0300h ; EDX = SystemCallStub
3. CALL DWORD PTR [EDX]
4. RETN 8
```

1. מוכנס לתוך האוגר EAX מספר הפונקציה אותה אנחנו רוצים לבצע.
2. מוכנס לתוך האוגר EDX את הערך הקבוע 03000x7FFe0 המצביע אל: SharedUserData!SystemCallStub המכיל את הכתובת של ל-KiFastSystemCall.
3. קריאה ל-EDX (קריאה ל-ntdll!KiFastSystemCall)
4. חזרה לפונקציה שקראה לנו.

אז מה יש ב-KiFastSystemCall?

```
1. MOV EDX, ESP
2. SYSTENTER
3. RETN
```

1. מוכנס ל-EDX מצביע לארגומנטים המועברים לפונקציה.
2. מתבצעת ההוראה SYSENTER המבצעת את המעבר לקרנל.
3. חזרה לפונקציה שקראה לנו.

מי זה SYSENTER?

כאמור, החל מ-Windows XP הוצגה דרך חדשה למעבר של הריצה מ-Ring3 ל-Ring0 וחזרה - הפקודות SYSENTER/SYSEXIT.

מתוך Intel IA-32 (64) Programming Manual:

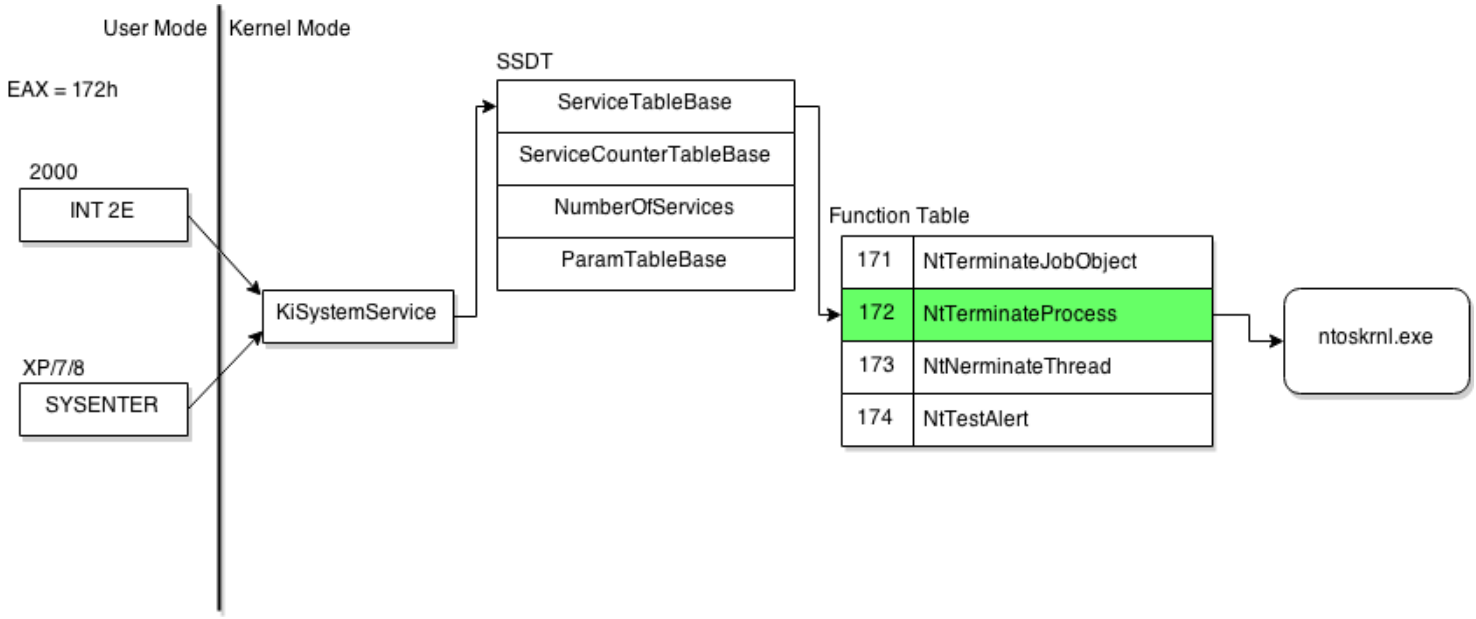
"Executes a fast call to a level 0 system procedure or routine. SYSENTER is a companion instruction to SYSEXIT.
The instruction is optimized to provide the maximum performance for system calls from user code running at privilege level 3 to
operating system or executive procedures running at privilege level 0."
- Intel IA-32 (64) programming manual, volume 2B.

מציינים כי הפקודה מספקת את הביצועים הטובים ביותר למעבר בין ה-Privilege Levels השונים. מה כל כך שונה בין SYSENTER ל-INT 2E? ההבדל נובע מהעובדה ש-INT 2E הינה הוראת פסיקה והדרך בה Windows מטפל בפסיקות (Interrupts).

מערכת ההפעלה מחזיקה טבלה הנקראת Interrupt Descriptor Table המכילה את כלל הפסיקות במערכת וה-Interrupt Service Routines (הקוד אשר מטפל בפסיקה) של כל אחת מהן. כאשר הפסיקה INT 2E נעשית, על המערכת הפעלה לגשת לרשומה מספר 2E ב-IDT המצביעה לטבלת ה- Global Descriptor Table שם יש מצביע לקוד המטפל בפסיקה (כן זה המון טבלאות, ככלל אצבע Windows בנויה על טבלאות ורשימות).

זאת אומרת, בכל System Call כל התהליך הזה יתבצע מחדש שוב ושוב מה שמוביל לאי יעילות (אפילו שהכתובות בדר"כ נשמרות ב-L1 Cache). נדרש למצוא פתרון יעיל יותר, ואכן הפקודה SYSENTER קופצת לכתובת קבועה וחוסכת למעבד המון קריאות מהזיכרון ומייעלת את תהליך קריאות המערכת (עוד על הנושא).

לסיכום נושא ה-System Calls, דיאגרמה של קריאת מערכת:



1. מתבצעת הפסיקה INT 2E/ההוראה SYSENTER.
2. הריצה מועברת לnt!KiFastCallEntry שקורא בתורו ל-nt!KiSystemService (לא מתואר בדיאגרמה לשם פשטות)
3. KiSystemService, בין היתר, לוקח את מספר הפונקציה שנמצא ב-EAX ומריץ את הפונקציה ב-SSDT המקבילה למספר זה.

אני ממליץ לקרוא עוד על הנושא ב-מאמר של OSR Online.

System Call Hooking

כמו שכבר הזכרתי, השיטה עצמה מוכרת כבר זמן מה ונראו סוגים שונים שלה ב-Carberp (החלפה של המצביע אל KiFastSystemCall במצביע אל קוד אחר) וב-Neurevt (א', ב'). אז הסיבה שלשמה התכנסנו כאן, אחרי שאנחנו מבינים איך קריאות מערכת מתבצעות ב-Windows, איפה נוכל לשים את Hook שלנו? עם הדיאגרמה של System Calling מהסעיף הקודם מול העיניים שלנו, אם נצטרך להצביע על מקום בו נוכל לבצע את ה-Hooking מאד מתבקש לשים את האצבע על טבלת המצביעים ב-SSDT, ואכן שיטה ידועה לביצוע Hooking היא לשנות ב-SSDT את המצביע מהמיקום האמיתי הפונקציה מיקום של קוד שלנו ובכך כל פעם שיקראו לפונקציה שאת המצביע שלה שינינו KiSystemService יריץ את הקוד שלנו במקום את הקוד האמיתי.

אז כן, לשיטה זו קוראים SSDT Hooking (מפתיע...) ויש איתה כמה בעיות:

1. קל מאד לעלות עליה, נבדוק אם כל המצביעים ב-SSDT מצביעים לתוך ntoskrnl.exe ואם לא, אז נדע שהייתה נגיעה בטבלה.
2. השיטה כבר נפוצה מאד ומנגנוני אנטי-וירוס יודעים לזהות אותה והוספו הגנות נגד שינויים כאלה ודומים בקרנל בגרסאות 64 ביט של Windows תחת Patchguard (וכמובן שגם את זה כבר אפשר לעקוף).
3. והסיבה הכי חשובה - שיטה זו תדרוש מאיתנו הרשאות כתיבה לקרנל.

אנחנו מחפשים מיקום ב-User Land בו נוכל לבצע את ה-Hook, משמע כל שאר האופציות ל-Hooking בקרנל יורדות מהפרק (ntoskrnl hooking, שינוי קוד הדרייבר עצמו וכו'). תפיסה שעולה בזמן האחרון היא שאת רוב הדברים הכיפים שהיו שמורים עד עכשיו לKernel Mode כבר אפשר לעשות ב-User Land. אז נחפש מקום בצד השמאלי של הדיאגרמה (User Land) בו נוכל לשים את ה-Hook. אנחנו נחפש מקום אחד מרכזי אליו כל ה-System Calls מתנקזים. ניחשתם נכון, המקום הזה הוא ה-KiFastSystemCall. אבל לא בדיוק. בואו נסתכל על זרימה של קריאת מערכת רגילה - במקרה הזה של NtTerminateProcess.

הערה: לאלו שרוצים לעקוב אחר הנעשה, נפתח את IDA וננתח את ntdll.dll. ניגש ללשונית ה-Exports ונבחר ב-NtTerminateProcess. בכדי לדאבג את את ntdll נצטרך לשנות את הגדרות ה- Process Options תחת הלשונית Debugger. בשדה Application רשום "C:\Windows\system32\rundll32.exe" ובשדה Parameters נרשום "NtTerminateProcess". מאחר ו-ntdll אינו קובץ הרצה נצטרך לקרוא ל-rundll32 שיריץ אותו ולספק לו כפרמטר DLL ופונקציה מתוך אותו DLL אשר נרצה להריץ. לסיום, נשים Breakpoint בתחילת הקוד של NtTerminateProcess ונריץ.

```
; NTSTATUS __stdcall ZwTerminateProcess(HANDLE ProcessHandle, NTSTATUS ExitStatus)
public _ZwTerminateProcess@8
_ZwTerminateProcess@8 proc near

ProcessHandle= dword ptr  4
ExitStatus= dword ptr  8

mov     eax, 172h       ; NtTerminateProcess
mov     edx, 7FFE0300h
call    dword ptr [edx]
retn    8
_ZwTerminateProcess@8 endp
```

```
7FFE02FE db    0
7FFE02FF db    0
7FFE0300 dd offset _KiFastSystemCall@0
7FFE0304 db 0F4h ; _
```

```
        ; int __stdcall KiFastSystemCall()
        public _KiFastSystemCall@0
        _KiFastSystemCall@0 proc near
8B D4   mov     edx, esp
0F 34   sysenter
C3      retn
```

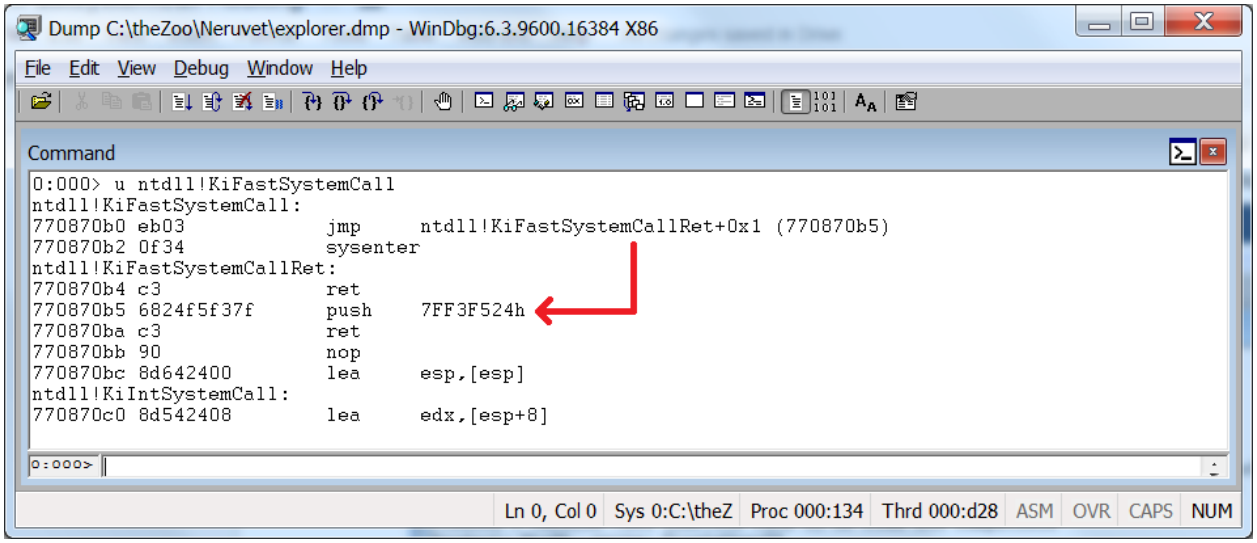
כמו שכבר למדנו, הקריאה מתחילה בפונקציה ZwTerminateProcess ש-ntdll מייצא. משם עוברים ל-SystemCallStub ולבסוף מגיעים ל-KiFastSystemCall שיעביר את המשך הריצה לקרנל. הרעיון של ה-Hook הוא לדרוס את חמשת הבייטים ב-KiFastSystemCall ולהחליפם בקפיצה לקוד שלנו.

אם נסתכל בדיאגרמה נוכל לראות ש-KiFastSystemCall מורכב מארבעה בייטים (**8B D4 0F 34**) ואז בייט אחד של **C3** (ההוראה RETN) כך שאם נדרוס את חמשת הבייטים הללו (בשביל הוראת JMP) נפריע לריצה של ה-System Call (מאחר ודרסנו גם את הפקודה RETN).

```
KiFastSystemCall    8BD4          MOV EDX,ESP
77A801D2            0F34          SYSENTER
KiFastSystemCallRet C3            RET
77A801D5            8DA424 0000000( LEA ESP,DWORD PTR [ESP]
77A801DC            8D6424 00     LEA ESP,DWORD PTR [ESP]
KiIntSystemCall     8D5424 08     LEA EDX,DWORD PTR [ESP+8]
77A801E4            CD 2E         INT 2E
77A801E6            C3            RET
```

אם נסתכל על האיזור בקוד של ntdll אחרי KiFastSystemCall נוכל לראות את שבין הפונקציה KiFastSystemCallRet (שמכילה רק את הפקודה RETN) לבין הפונקציה KiIntSystemCall (שאמורה להיות קצת מוכרת לכם ממנגנון ה-System Call של Windows 2000 וקיימת גם כאן בשביל תאימות לאחור) יש 11 (!) בייטים בהם נוכל לעשות שימוש.

אז נוכל לבצע SHORT JMP (הוראה שאורכה 2 בייטים אך מוגבלת לקפיצה של עד 127 בייטים קדימה) ל-[KiFastSystemCallRet+1] (זאת אומרת, בייט אחד לאחר הכתובת של KiFastSystemCallRet) ומשם נקפוץ לקוד שלנו. ואכן זה מה ש-Betabot עושה, התמונה הבאה היא ניתוח של Dump של התהליך Explorer.exe נגוע ב-Neurevt:



Neurevt עשה קפיצה מ-KiFastSystemCall ל-KiFastSystemCallRet+1 שם הוא ביצע את ההוראות:

```
PUSH 7FF3F524h
RET
```

1. דחיפה למחסנית של הכתובת 0x7FF3F524.
2. קפיצה לכתובת 0x7FF3F524 (ההוראה RET מחזירה את הריצה לכתובת האחרונה שנדחפה למחסנית).

שימוש ב-PUSH/RET במקום ב-JMP מקל עלינו מאחר והקפיצה היא לכתובת אבסולוטית ולא יחסית כמו בהוראת JMP.

טוב, אחרי שאנחנו יודעים את כל זה, בואו נתחיל לעבוד על הקוד. הקוד עצמו יהיה באסמבלי, אני אנסה להסביר כמה שיותר, אך מומלצת הכרה עם השפה.

Writing some code

אני משתמש ב-RadAsm כסביבת פיתוח וב-MASM כשפת הפיתוח. הקוד שנכתוב הוא תוכנית בפני עצמה (ז"א exe.), ולא דווקא קוד המוכן להזרקה משתי סיבות:

א. להקשות במעט על שימוש לא נכון בקוד.

ב. פשטות של הקוד בהצגה במאמר.

כמו שכבר ציינתי, הקוד לא דורש הרשאות מיוחדות. לא נצטרך לכתוב דרייבר או כל רכיב קרנלי אחר ואפילו לא הרשאות אדמיניסטרטור. נוכל להשפיע על כל תהליך שרץ בהרשאות של המשתמש אשר הפעיל את התוכנה (אם אנחנו כן מריצים את הקוד בהרשאות אדמיניסטרטור נוכל להזריק קוד גם לתהליכים של משתמשים אחרים).

אז נתחיל בהגדרות סטנדרטיות:

```
.386                    ; 80386 processor nonprivileged instructions
.model flat,stdcall     ; STDCALL calling convention, flat memory arrangement
option casemap:none     ; Case sensitive
```

נצטרך להשתמש בפונקציה VirtualProtect מ-Kernel32 בשביל להוסיף הרשאות כתיבה לאיזור אותו אנחנו רוצים לשנות, אז נצטרך להוסיף את הספרייה Kernel32.

```
include kernel32.inc    ; Add kernel32 definitions
includelib kernel32.lib ; Link against kernel32.lib
```

הגדרת Data Section:

```
.data

oldProtection dd ?       ; For VirtualProtect()
arrayOfEvil DWORD 149h DUP (0), offset newNtSetInformationFile , 40h DUP (0);
Place hooks here by Syscall numbers
```

ונתחיל עם הקוד, תחילה נשים ב-EAX את הכתובת של: KiFastSystemCall

```
.code ; Start of code section

start:
      mov esi, 07FFE0300h       ; ESI = SharedUserData!SystemCallStub
      lodsd                    ; EAX = KiFastSystemCall
      call changeProtection
```

כאשר changeProtection היא פונקציה קטנה שקוראת ל-VirtualProtect. אנחנו לא יכולים לכתוב ל-KiFastSystemCallRet כי האזור לא בעל הרשאות Write, אז נשתמש ב-VirtualProtect להוסיף הרשאות.

```
      changeProtection:
            push eax                  ; Save KiFastSystemCall addr
            push offset oldProtection
            push 40h                  ; PAGE_EXECUTE_READWRITE
            push 6                    ; [KiIntSystemCall - KiFastSystemCall]
            push eax
            call VirtualProtect       ; VirutalProtect((void
*)KiFastSystemCall, 6,
      PAGE_EXECUTE_READWRITE, &oldProtection)
            pop eax                   ; EAX = KiFastSystemCall addr
            retn
```

במהלך התוכנית נשתמש באוגר EDX להחזיק את ההוראות אותן נרצה לכתוב ובאוגר EAX כאוגר שמצביע לנו לאן לכתוב.

```
1. mov edx, 03EBh ; 0xEB03 =  JMP SHORT 3 bytes
2. mov [eax], edx
```

1. נכניס ל-EDX את ההוראה שתדרוס את התוכן המקורי של KiFastSystemCall.
2. נדרוס את KiFastSystemCall בתוכן שנמצא ב-EDX (קפיצה של 3 בייטים קדימה).

אנחנו נהיה נאמנים ל-Neurevt ונשתמש ב-PUSH/RET בשביל לקפוץ לקוד שלנו ונשים את הרצף הוראות הזה ישר אחרי KiFastSystemCallRet:

```
1. lea eax, [eax + 5]         ; EAX = [KiFastSystemCallRet + 1]
   mov dl, 68h                ; 0x68 = PUSH
   mov [eax], dl              ; [KiFastSystemCallRet + 1] = PUSH
```

```
   2. inc eax              ; EAX = [KiFastSystemCallRet + 2]
      mov edx, offset evilCode  ; EDX = pointer to our trap
      mov [eax], edx            ; Now [KiFastSystemCallRet + 1] = PUSH offset
                                                    evilCode
   3. lea eax, [eax + 4]        ; EAX = [KiFastSystemCallRet + 6]
      mov dl, 0C3h              ; 0xC3 = RET
      mov [eax], dl             ; [KiFastSystemCallRet + 6] = RETN
```

1. את הקוד שלנו נכתוב ישר אחרי KiFastSysetmCallRet - שם מצאנו 11 בייטים שנוכל לכתוב אליהם (מתוכן נשתמש בשש בייטים בשביל PUSH/RET). אז נשים בEAX את הכתובת של האזור אליו נקפוץ.
2. נתחיל בלכתוב את הוראת ה-PUSH (מורכבת מהבייט 0x68 ואז הכתובת).
3. לאחר הבייט 0x68 נכתוב את הכתובת אל הקוד שלנו שיקפוץ אל ה-Hook-ים שלנו.
4. לאחר ה-PUSH נסיים בהוראת RETN.

אז עכשיו כל קריאת מערכת תעבור דרך ה-evilCode שלנו. הקוד בודק מול טבלה שיצרנו בשם arrayOfEvil האם ה-Syscall הנוכחי הוא אחד מאלה שנרצה לעשות להם Hook. כל תא במערך מייצג פונקציה. אם נרצה לעשות לפונקציה מסויימת Hook אז נכתוב בתא שמייצג אותה את הכתובת של ה-Hook שלנו, אם אנחנו לא מעוניינים בפונקציה אז התא שלה ימולא באפסים.

נוכל לשים כמה Hooks שנרצה ובקלות רבה, וזה אחד היתרונות הגדולים בגישה הזאת. בדוגמא כאן אנחנו נבצע hook פשוט יחסית ל-NtSetInformationFile (פונקציה מספר 0x149 ב-Windows 7) שימנע מאיתנו למחוק קבצים במערכת, אז נמלא 0x149 תאים באפסים ובתא הבא המייצג את הפונקציה נשים את הכתובת של NtSetInformationFile.

```
evilCode:
   1. mov ecx, offset arrayOfEvil
   2. lea ecx, [ecx + eax * 4]
   3. mov ebx, [ecx]
   4. cmp ebx, 0
   5. jz origKiFastSystemCall
   6. jmp ebx
```

1. נשים ב-ECX את המיקום של התחלת המערך.
2. נחשב את כתובת התא המייצג את הפוקנציה שלנו.
3. נשים ב-EBX את תוכן התא.
4. בדיקה אם התא ריק.
5. אם כן, אין לנו עניין בפונקציה הזאת, תמשיך כרגיל.
6. אם התא לא ריק, קפוץ ל-EBX המכיל את כתובת ה-Hook שלנו.

מעולה, עכשיו כל System Call יעבור דרך הקוד שלנו. ציינתי שכדוגמא ל-Hook נמנע מחיקה של קבצים, אז נתחיל לעבוד על הקוד שיעשה זאת, והוא פשוט מאד. מחיקה של קבצים תיעשה בדרך כלל על ידי פונקציית Windows API בשם DeleteFile, ואנחנו צריכים לגלות לאיזה Native API ה-DeleteFile תקרא בעצמה.

למרבה ההפתעה היא לא תקרא ל-NtDeleteFile (למרות שהיא קיימת) אלא אל NtSetInformationFile שנראית כך:

```
NTSTATUS NtSetInformationFile(
  IN HANDLE               FileHandle,
  OUT PIO_STATUS_BLOCK    IoStatusBlock,
  IN PVOID                FileInformation,
  IN ULONG                Length,
  IN FILE_INFORMATION_CLASS FileInformationClass
);
```

[תיעוד לפונקציות לא מתועדות ניתן למצוא לרוב ב-undocumented.ntinternals.net.]

הפונקציה מבצעת פעולות שונות לפי הפרמטר החמישי שהוא טיפוס FILE_INFORMATION_CLASS. במקרה של מחיקת קובץ, בפרמטר החמישי יהיה הערך 0xD (הערך אשר מייצג את FILE_DISPOSITION_INFORMATION) והפרמטר השלישי יצביע אל מבנה המורכב ממשתנה BOOLEAN אחד המציין אם למחוק את הקובץ או לא:

```
newNtSetInformationFile:
   1. Pushad
   2. mov edi, [esp + 38h]
   3. cmp edi, 0Dh                    ; 0xD = FileDispositionInformation
   4. jnz callRealKiFastSystemCall
      xor edi, edi
   5. mov ebx, [esp + 30h]            ; EBX = (VOID *)dispositionInfo
   6. mov [ebx], dl                   ; dispositionInfo.DeleteFile = 0 (FALSE)
   7. callRealKiFastSystemCall:
   8. popad
   9. jmp origKiFastSystemCall
```

1. שמירה של כל ה-General Purpose Registers במחסנית.
2. נכניס ל-EDI את הארגומנט FILE_INFORMATION_CLASS.
3. האם FILE_INFORMATION_CLASS הוא מסוג FILE_DISPOSITION_INFORMATION?
4. אם לא, לא מדובר במחיקה של קובץ ואין לנו עניין בשינוי הפונקציה, תמשיך כרגיל.
5. נכניס ל-EBX את המצביע אל הStruct שקובע אם למחוק את הקובץ או לא.
6. נשים שם את הערך FALSE.
7. נחזיר מהמחסנית את ה-General Purpose Registers למצבם הרגיל.

אחרי שכתבנו את הקוד ל-Hook נבדוק אם הוא עבד ונקרא ל-DeleteFile:

```
push offset fileToDelete
      call DeleteFile                  ; Will call NtSetInformationFile

      retn
```

ונראה שהקובץ לא נמחק :).

נוסיף רק את הקוד המשחזר את KiFastSystemCall אליו יקפצו כל ה-System Calls לבסוף:

```
      origKiFastSystemCall:
            mov edx, esp
            dw 340fh                   ; SYSENTER
            retn

end start
```

וסיימנו!

דרך ב'

שיטה נוספת שאני חושב ששווה לעבור עליה בקצרה כי היא מאד מעניינת הועלתה כאן ומדברת על קפיצה מ-KiFastSystemCall אל KiIntSystemCall המכילה 7 בייטים.

עולה כאן בעיה, כי מה יקרה במקרה הלא סביר ש-KiIntSystemCall תיקרא על ידי פונקציה אחרת? אז הכותב מציע להשתמש בפקודות STD ו-CLD (שינוי דגל ה-Direction) לסמן אם הפונקציה הגיעה דרך הקוד שלנו או שקראה ישירות ל-KiIntSystemCall. נשים בבייט הראשון של KiIntSystemCall את הפקודה STD שתדליק את ה-Direction Flag וב-KiFastSystemCall נשים בביט הראשון CLD שינקה את הדגל ואז נקפוץ ל-[1+KiIntSystemCall] (כך שנדלג על ה-STD).

הקוד יראה דומה מאד לקוד שלנו, רק נצטרך להוסיף את ההוראות CLD, STD ולשנות את הקפיצה כך שתקפוץ ל-[1+KiIntSystemCall]:

```
mov edx, 0EEBFCh           ; 0xFC = CLD, 0xEB0F JMP SHORT 0xE bytes
mov [eax], edx
lea eax, [eax + 10h        ; EAX = KiIntSystemCall
mov dl, 0FDh               ; 0xFD = STD
mov [eax], dl
```

נשאר לנו רק לבדוק בקוד אליו אנחנו קופצים אם ה-Direction Flag דולק או לא:

```
1. Pushfd
2. pop edx
3. bt edx, 0Ah                ; CF = DF
4. jc origKiIntSystemCall
   mov ecx, offset arrayOfEvil
   lea ecx, [ecx + eax * 4]
   mov edx, [ecx]
   cmp edx, 0
   jz origKiFastSystemCall
   jmp edx
```

1. נדחוף את ה-EFLAGS למחסנית.
2. נוציא אותם אל תוך EDX.
3. נעביר את הביט העשירי (המייצג את ה-Direction Flag) אל ה-Carry Flag.
4. אם ה-Carry Flag דלוק, קראו ישירות ל-KiIntSystemCall, קפוץ ל-origKiIntSystemCall.

הקוד המלא נמצא גם הוא Git.

סיכום

Ring3 Rootkits מתחילים להיות יותר ויותר נפוצים מתוך ההבנה של תוקפים שכל מידע אשר ירצו יוכלו להשיג גם בלי צורך של נגיעה בקרנל ובכך לחסוך זמן יקר ואת הבעיות שעיסוק שכזה מעלה. הרעיונות של כותבי הוירוסים האלה יצירתיים להפליא ונכנסים תחת המגמה של יוצרי הוירוסים שאומרת שאין סיבה "ללכת ראש בקיר", אם חסמו לנו את הגישה ל-Kernel אז נעבור ל-User Mode, אם חסמו לנו את User Mode אז נעבור לדבר הבא ועדיף כמה שיותר מוקדם.

Hypervisor rootkits ו-UEFI bootkits הם רק דוגמאות של הכיוון אליו התחום הזה מתקדם ואני חושב שהמאמר הזה הוא דוגמא טובה למגמה הזאת, אז בהצלחה לכולנו.

שאלות או הערות אשמח לקבל למייל shahakshalev@gmail.com ואתם מוזמנים להציץ ב-Git לפרויקטים שלי וכאלה שאני שותף להם.