digital whisper
Technologic Papers

הנדסת דרייברים לאחור: מתודולוגיה ושימוש פרקטי

מאת שי גילת

הקדמה

Reverse Engineering הינו קונספט חשוב בכל תחום הקשור לטכנולוגיה. בין אם זה מפתח שמנסה להבין איך להשתמש ב-API או משאב אחר שניתן לו מהסביבה בצורה יותר טובה, ובין אם זה חוקר אבטחה שמנסה לחקור את כל המשטחים הקיימים בתוכנה כדי למצוא פרצה. כמובן שהנדסה לאחור תהיה קצת שונה בכל ענף של תוכנה ומחקר, אך לרוב יהיו כמה דברים קבועים\דומים שניתן לעשות בהם שימוש:

1) כלים להנדסה לאחור: בין אם זה IDA Pro, Ghidra, JADX או אפילו הנדסה לאחור ללא כלי נוסף אלא מהיגיון וסריקת קלט/פלט בלבד.
2) היגיון בסיסי מאחורי התוכנה ומאחורי ה-Parsing שנעשה בתוכנה: אם אני מהנדס לאחור קובץ exe בעזרת IDA אני אדע איפה נמצאת נקודת הכניסה של התוכנה, טיפוסים של כל פרמטר וגודלם, אוכל להבין את המטרות של כל פונקציה בעזרת השם והקוד שנוצר מתהליך ה-decompiling וכך ניתן ליצור דרך פעולה לניתוח תוכנות דומות ולמציאת פרצות אפשריות בהן.
3) ניסיון וטעייה: אם בתוכנה קיים struct ענקי בלי תיעוד ואני צריך להבין באיזה offset מתחילתו נמצא הקלט שאני מביא, אני אוכל לנתח ולדבג את התוכנה כדי לראות מה מגיע לאן ואולי אפילו לשחזר תוכנה דומה משלי כדי להבין באיזה צורה אותו קלט נראה ב-IDA או תוכנה אחרת.

את כל התהליך הזה אני הולך להסביר במאמר הזה, ממליץ בחום למי שלא קרא את המאמרים הקודמים שלי מגיליונות 162 ו-163 לעבור עליהם, כי באותם מאמרים אני מתאר קונספטים בסיסיים שיכולים לעזור להבין את התוכן במאמר הזה.

במאמר אני הולך להכנס להנדסה לאחור על קוד קרנלי, או בעצם KM drivers במערכת ההפעלה Windows, אני הולך לעבור על הנקודות המרכזיות שעליהן אני חושב שאני מקבל קובץ sys (driver.sys זה הפורמט המשומש ב-Windows עבור דרייברים), איזורים בקוד שאני אתמקד בהם במחקר ועקרונות בסיסיים שהכרחיים למחקר ומאוחר יותר גם להשמשה.

בנוסף לכך, אני אציג ניסיון כושל שלי להשיג חולשה ב-driver של חברה מוכרת, וגם אראה ניסיון מוצלח שלי שלאחר מחקר של one-day שמצאתי נתן לי אפשרות להטעין תוכנה לפי בחירתי לזיכרון הקרנלי בעזרת חולשות לוגיות שמצאתי ב-driver.

POIs (Points of Interest) בהנדסה לאחור של דרייברים

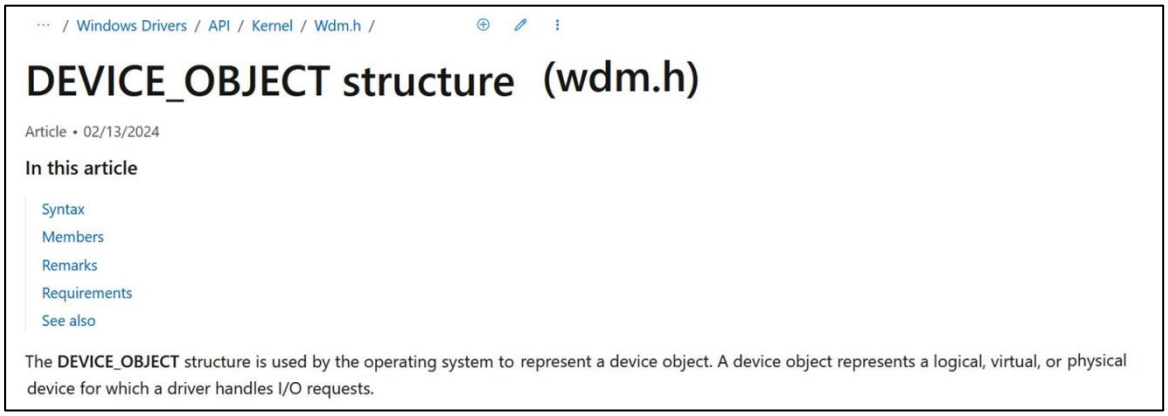
בחלק זה אני הולך לציין כמה נקודות בסיס חשובות כדי לראות אם ניתן בכלל להשמיש חולשה כלשהי ב-driver, נראה מה התנאים הבסיסיים שחייבים להיות כדי שתוכנת UM תוכל בכלל לתקשר עם ה-driver ונלמד לזהות קלט-פלט שמגיע מהקורא ל-driver.

DriverEntry

הפעולה DriverEntry היא השם הקלאסי והמשומש ביותר לנקודת הכניסה של תוכנת ה-driver. כפי שהסברתי במאמרים הקודמים הפעולה מחליפה את פעולת ה-main בתוכנות בינאריות רגילות, והפעולה מוכנה לקבל 2 פרמטרים: DriverObject (טיפוס *DRIVER_OBJECT, מצביע לאובייקט הגלובלי שמייצג את ה-driver במערכת) ו-RegistryPath (טיפוס PUNICODE_STRING, כמו כל service בווינדוס גם ל-driver יש מפתח משלו ב-registry שמכיל מידע כמו מתי להריץ את ה-driver ומה המסלול לקובץ הבינארי, פרמטר זה מכיל את המסלול למפתח של ה-driver ב-registry).

בתוך פעולת ה-DriverEntry של driver נורמלי נצפה לראות כמה נקודות מרכזיות ובסיסיות:

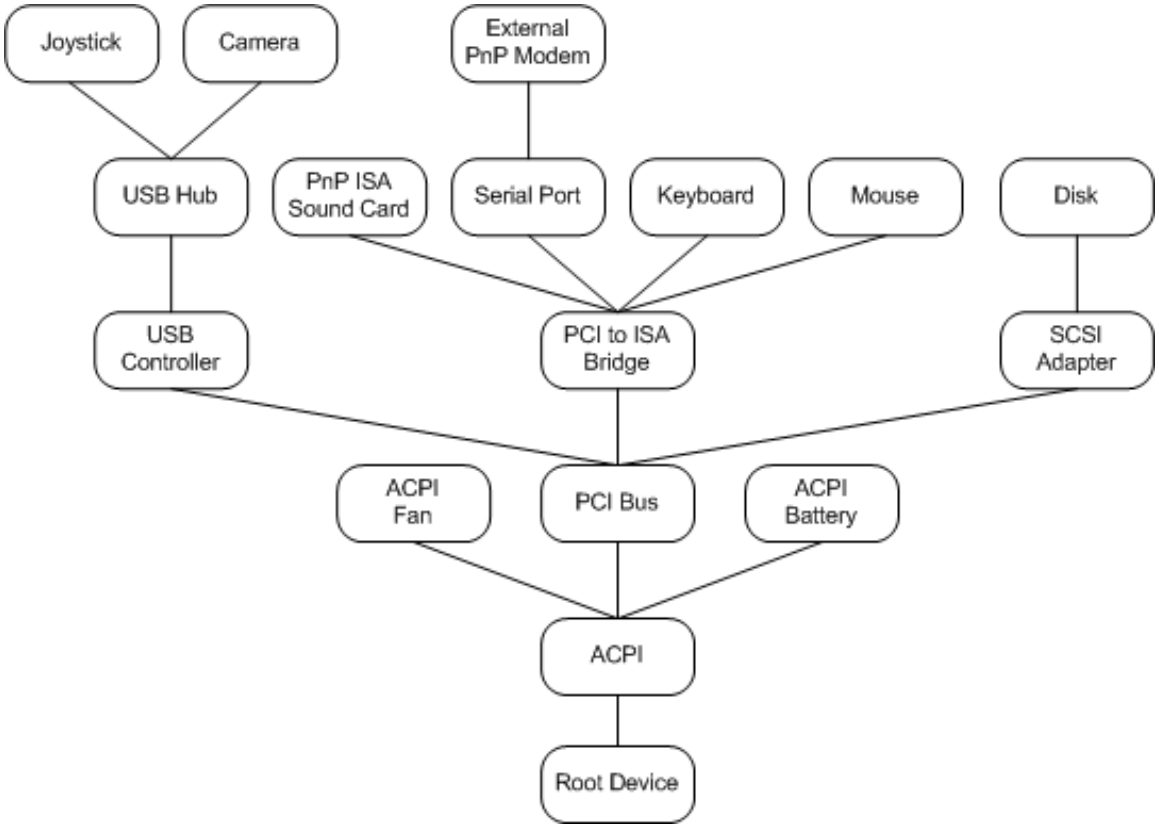
1. **קריאה ל-IoCreateDevice:** מתוך ה-MSDN:



ניתן להבין מהגדרה זו שמטרת הפעולה IoCreateDevice היא לקשר "מכשיר" כלשהו לאותו driver, ולהצהיר למערכת שכל ה-I/O לאותו מכשיר יעברו דרך ה-driver שקרא לפעולה.

לפי ההגדרה באמת ניתן לראות שה-device לא חייב להיות מכשיר פיזי כמו חיבור USB או עכבר, אלא גם יכול להיות "מכשיר וירטואלי" - מכשיר דמיוני עבורו אנחנו מספקים ממשק I/O שממומש על ידי ה-driver.

במערכת ההפעלה קיים רכיב בשם PnP manager. הרכיב הזה אחראי לנהל את כל המכשירים הרשומים במערכת והוא עושה זאת בצורה של עץ:



[מקור: https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/device-tree]

כפי שניתן לראות יש בניהול ה-devices היררכיה ברורה והגיונית מאוד: אם אני ארצה להגיע למצלמה שרשומה למעלה אני אצטרך קודם כל לעבור דרך ה-USB hub שאליו היא מחוברת, וככה כדי להפעיל device מסוים אני חייב לעבור דרך כל ה-dependencies שלו בעץ.

RootDevice הוא בעצם virtual device שמייצג את הבסיס לעץ המכשירים שקיימים במערכת, וכשמבוצע שימוש בפעולה כמו IoCreateDevice מבצעים אנומרציה על העץ מה-RootDevice עד שמגיעים למיקום אליו צריך להכניס את המכשיר.

לא הכרחי להבין את כל הארכיטקטורה של PnP עבור מטרתינו, אך הדבר המשמעותי שצריך לקחת מההסבר הוא העובדה שכדי להעביר I/O אל או מה-driver שלנו נצטרך DEVICE_OBJECT שנוכל לפנות אליו ולתקשר דרכו.

2. קריאה ל-IoCreateSymbolicLink: מתוך המאמר הקודם שלי במגזין, על מעקף DSE בעזרת BYOVD:

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).

יצירת symbolic link עבור ה-device שאנחנו מתכננים לתקשר דרכו עם תוכנות אחרות הוא הכרחי עבור תקשורת עם תוכנות UM.

הסיבה לכך היא שאחרי שאנחנו טוענים driver לזכרון, תמנע מאיתנו כל דרך אחרת להשיג handle עבור ה-driver, ולכן ללא קריאה לפעולה הזאת לא נוכל להשמיש שום פרצה גם אם מצאנו אותה.

גם לשם של ה-device שאנחנו יוצרים יש פורמט קבוע וברור בדומה ל-symbolic link:

\\Device\\DeviceName

בדרך כלל באמת נהוג לקרוא ל-device באותו השם של ה-symbolic link אך אין צורך מיוחד.

3. רישום callbacks למקרים שונים, וספציפית ל-IRP_MJ_DEVICE_CONTROL:

IRP major tables הם הטבלאות שהדרייבר מאכלס בתוך ה-DriverEntry שלו. כפי שציינתי בהסבר על ה-DRIVER_OBJECT, הטבלה היא בעצם מערך של מצביעים בגודל קבוע של 28, שלכל אינדקס ברשימה הזו יש מטרה מסוימת בפעילות הדרייבר (פעולה מסוימת שהדרייבר מבצע במקרה שמגיעה בקשה כזו).

המטרה של כל אינדקס ברשימה מוגדרת בקובץ wdm.h בעזרת macro-ים שנראים כך:

```
#define IRP_MJ_CREATE                    0x00
#define IRP_MJ_CREATE_NAMED_PIPE         0x01
#define IRP_MJ_CLOSE                     0x02
#define IRP_MJ_READ                      0x03
#define IRP_MJ_WRITE                     0x04
#define IRP_MJ_QUERY_INFORMATION         0x05
#define IRP_MJ_SET_INFORMATION           0x06
#define IRP_MJ_QUERY_EA                  0x07
#define IRP_MJ_SET_EA                    0x08
#define IRP_MJ_FLUSH_BUFFERS             0x09
#define IRP_MJ_QUERY_VOLUME_INFORMATION  0x0a
#define IRP_MJ_SET_VOLUME_INFORMATION    0x0b
#define IRP_MJ_DIRECTORY_CONTROL         0x0c
#define IRP_MJ_FILE_SYSTEM_CONTROL       0x0d
#define IRP_MJ_DEVICE_CONTROL            0x0e
#define IRP_MJ_INTERNAL_DEVICE_CONTROL   0x0f
#define IRP_MJ_SHUTDOWN                  0x10
#define IRP_MJ_LOCK_CONTROL              0x11
#define IRP_MJ_CLEANUP                   0x12
#define IRP_MJ_CREATE_MAILSLOT           0x13
#define IRP_MJ_QUERY_SECURITY            0x14
#define IRP_MJ_SET_SECURITY              0x15
#define IRP_MJ_POWER                     0x16
#define IRP_MJ_SYSTEM_CONTROL            0x17
#define IRP_MJ_DEVICE_CHANGE             0x18
#define IRP_MJ_QUERY_QUOTA               0x19
```

כשתוכנת UM מתקשרת עם driver, היא תעשה זאת כמעט בכל המקרים בעזרת ה-API שנקרא DeviceIoControl.

כפי שתיארתי מעלה, ה-driver מאכלס פעולות שיבוצעו בהגעת IRP_MJ code מסוים ל-driver, ובמקרה הזה כדי שנוכל לתקשר עימו הוא נדרש לאכלס את הפעולה באינדקס 14 (IRP_MJ_DEVICE_CONTROL, ישלח על ידי DeviceIoControl עם הפרמטרים שאנחנו סיפקנו), בלי אכלוס זה אנחנו לא נוכל להעביר את הבקשות שלנו ל-driver ותמנע מאיתנו אפשרות השמשה של אותן חולשות שאנחנו מחפשים

בנוסף לנקודות הבסיס הללו יכולות להתבצע ב-DriverEntry פעולות נוספות לפי המטרה של ה-driver, לדוגמא אתחול events שקשורים ל-driver או אלוקציה של זיכרון הכרחי לפעילות ה-driver.

ניתן לראות דוגמת קוד שמבצעת את כל מה שציינתי כאן:

```
// Create device and symbolic link for driver:
Status = IoCreateDevice(DriverObject, 0, &DeviceName, FILE_DEVICE_UNKNOWN,
    FILE_DEVICE_SECURE_OPEN, FALSE, &DeviceObject);
if (!NT_SUCCESS(Status)) {
    DbgPrintEx(0, 0, "HyperGamma - IoCreateDevice() failed with status 0x%x\n", Status);
    return Status;
}
Status = IoCreateSymbolicLink(&SymbolicLink, &DeviceName);
if (!NT_SUCCESS(Status)) {
    DbgPrintEx(0, 0, "HyperGamma - IoCreateSymbolicLink() failed with status 0x%x\n", Status);
    IoDeleteDevice(DeviceObject);
    return Status;
}


// Register IOCTL callbacks and unload function for the driver:
DriverObject->MajorFunction[IRP_MJ_CREATE] = IoctlCallbacks::CreateCloseCallback;
DriverObject->MajorFunction[IRP_MJ_CLOSE] = IoctlCallbacks::CreateCloseCallback;
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = IoctlCallbacks::DeviceControlCallback;
DriverObject->DriverUnload = DriverUnload;
DbgPrintEx(0, 0, "HyperGamma - DriverEntry() finished successfully, calling assembly function from Driver.asm ..\n");
DriverAssembly();  // Call function in Driver.asm
return Status;
```

IRP_MJ_DEVICE_CONTROL dispatch

לאחר הבנת הבסיס הדרוש למחקר אפקטיבי והשמשה אפשרית של driver, נסתכל על הנקודה המרכזית בתהליך ההתקפה שלנו: פעולת ה-DeviceControl שדיברנו עליה בחלק הקודם. הפקודה כתובה בפורמט זהה לפורמט של כל פעולת dispatch שיכולה להרשם ע"י ה-driver:

```
_Function_class_(DRIVER_DISPATCH)
_IRQL_requires_max_(DISPATCH_LEVEL)
_IRQL_requires_same_
typedef
NTSTATUS
DRIVER_DISPATCH (
    _In_ struct _DEVICE_OBJECT *DeviceObject
    _Inout_ struct _IRP *Irp
    );

typedef DRIVER_DISPATCH *PDRIVER_DISPATCH;
```

הפעולה תקבל את המצביע ל-DEVICE_OBJECT שדרכו אנחנו מתקשרים ואת ה-IRP שמתאר את הבקשה ל-driver. מי שלא יודע מה הוא IRP יכול לעבור על המאמר הקודם שלי, אך בעקרון המבנה הוא מבנה פנימי שמתאר בקשה שנשלחה ל-driver על ידי תוכנת UM או על ידי driver אחר. המבנה מאוכלס על ידי מערכת ההפעלה בעת שליחת הבקשה מ-UM ל-driver ובתוכו בין היתר מופיעים הבאפרים שסיפקנו ל-I/O עם ה-driver והגדלים של כל באפר.

אנחנו נצפה לראות ממש מוקדם קריאה לפעולה IoGetCurrentIrpStackLocation, הפעולה הזו תשיג לנו את המיקום במחסנית בו נמצאים הפרמטרים החשובים לנו מתוך ה-struct של ה-IRP כגון ה-I/O שהועבר ל-driver וה-IOCTL code. בגלל שאותם פרמטרים מיוצגים כמבנה בשם IO_STACK_LOCATION בתוך ה-IRP, הפעולה תחזיר לנו ערך מטיפוס מצביע לאותו מבנה.

IOCTL Code

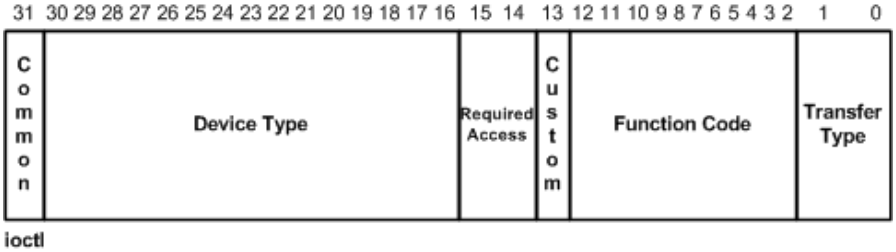
הדבר הבא שנרצה לבדוק הוא ה-IOCTL code, הקוד הזה מתאר את סוג הפעולה שנרצה שה-driver יבצע. ה-IOCTL נמצא במסלול ידוע בתוך ה-IO_STACK_LOCATION ובגלל שבדרך כלל דרייברים תומכים ביותר מסוג אחד של פעולה, הערך של ה-IOCTL ייבדק ב-switchcase:

```
// Determine which I/O control code was specified:
switch (ParamStackLocation->Parameters.DeviceIoControl.IoControlCode) {
case EXAMPLE_IOCTL:

    // For now defaultly uses METHOD_BUFFERED for I/O buffers:
    InputBufferSize = ParamStackLocation->Parameters.DeviceIoControl.InputBufferLength;
    OutputBufferSize = ParamStackLocation->Parameters.DeviceIoControl.OutputBufferLength;
    InputBuffer = (PUCHAR)Irp->AssociatedIrp.SystemBuffer;
    OutputBuffer = (PUCHAR)Irp->AssociatedIrp.SystemBuffer;
    break;

default:
    Status = STATUS_INVALID_DEVICE_REQUEST;
    break;
}
```

ניתן ליצור IOCTL code עבור עוד סוג של פעולה שאנחנו רוצים לתמוך בה ב-2 דרכים מרכזיות:

1) בחירת ערך מיוחד ואקראי שאנחנו עדיין לא תומכים בו
2) שימוש ב-macro שנקרא CTL_CODE, בעזרת ה-macro הזה ניתן ליצור IOCTL code מיוחד עבור הפעולה החדשה בפורמט שגם יצהיר על מידע חשוב שקשור להעברת הפרמטרים ולערכים שמסופקים ל-driver.

ניתן לראות את ההסבר על ה-IOCTL שה-macro יוצר כאן:



- Device Type: ערך שמתאר את סוג ה-device שמטפל בבקשה שנשלחה, לדוגמא עכבר או מקלדת. כל סוג מוגדר כ-macro בקבצי ה-headers הקרנליים של ווינדוס והערך שנשתמש בו הכי הרבה הוא FILE_DEVICE_UNKNOWN (סוג ה-device לא ידוע או לא רלוונטי לפעולה)
- Required Access: יכולת להגביל את צורת היצירה של handle עבור ה-driver. במקרה והקורא שהשיג handle ל-driver בעזרת הפעולה CreateFile סיפק הרשאות לא מתאימות, ניתן למנוע ממנו אפשרות לשלוח את הבקשה. הערכים יכולים להיות שילוב בין FILE_READ_DATA / FILE_WRITE_DATA / FILE_ALL_ACCESS.
- בדרך כלל FILE_ALL_ACCESS מסופק ל-macro בגלל שאין צורך בהגבלה מסוימת.
- Custom: לא מושפע על ידי ה-macro ולא רלוונטי לפעולה.

- Function Code: הערך המרכזי לנו שהופך את הקוד למיוחד, ערכים מתחת ל-0x800 שמורים לפי הקונבנציה הרגילה לקודים פנימיים של Windows, אבל אין הגבלה ממשית בבחירת הערך.
- Transfer Type: ערך שחשוב לנו כדי להבין איך הקלט-פלט שהעברנו ל-driver נראה בזכרון. יש לכל ערך משמעות בנוגע למיקום ב-IO_STACK_LOCATION שבו נמצאים הבאפרים שסיפקנו ל-driver ותיאור הזיכרון שסיפקנו. ה-memory manager במערכת ההפעלה יחליט איך לפעול במצב שבו מגיע כל ערך אפשרי של transfer type:

1) METHOD_BUFFERED: ה-memory manager יקצה בזיכרון המערכת באפר שיספיק גם לקלט וגם לפלט (הערך הגדול מבין הגדלים של הקלט או הפלט), התוכן מהקלט של התוכנה יועתק לבאפר בזיכרון המערכת לפני הקריאה ל-driver ואחרי הקריאה ל-driver התוכן של הפלט יועתק מהבאפר בזיכרון המערכת אל הבאפר המקורי של הפלט (באפר בזיכרון מערכת שהוא משותף לקלט ופלט). המצביע לבאפר יהיה מאוכסן במסלול Irp->AssociatedIrp.SystemBuffer והגדלים יאוכסנו במסלולים הבאים:

For input data, the buffer size is specified by **Parameters.DeviceIoControl.InputBufferLength** in the driver's IO_STACK_LOCATION structure.
For output data, the buffer size is specified by **Parameters.DeviceIoControl.OutputBufferLength** in the driver's **IO_STACK_LOCATION** structure.

2) METHOD_IN/OUT_DIRECT: שני הערכים הללו לא שונים בהרבה אחד מהשני, בשניהם יש באפר אחד מהשניים (קלט **או** פלט) שמסופק במסלול.

Irp->AssociatedIrp.SystemBuffer ושמאוכסן בזיכרון מערכת, והגודל של הבאפר שמאוכסן בזיכרון מערכת יהיה באותו מסלול של הגודל לבאפר הקלט באפשרות הקודמת. ההבדל המשמעותי בין השתיים זה איזה באפר נמצא בזיכרון מערכת: METHOD_IN_DIRECT מציין שהבאפר בזיכרון מערכת הוא הבאפר לפלט, ו-METHOD_OUT_DIRECT מציין שהבאפר בזיכרון מערכת הוא הבאפר לקלט. בשתי האפשרויות הבאפר שלא נמצא בזיכרון מערכת יתואר על ידי מבנה MDL במסלול.

Irp->MdlAddress וכך במקרה שיש צורך ניתן למפות אותו לזיכרון מערכת. האורך של הבאפר השני ימצא באותו מסלול של אורך הבאפר לפלט באפשרות הקודמת.

3) METHOD_NEITHER: באפשרות זו ה-memory manager לא מתערב בכלום, זה אומר שהבאפרים לקלט ופלט לא ממופים לזיכרון מערכת, לא מתוארים על ידי MDL והמידע היחיד שמסופק הוא הכתובת UM של כל אחד מהבאפרים. את המסלולים לכל ערך ניתן לראות כאן:

The input buffer's address is supplied by **Parameters.DeviceIoControl.Type3InputBuffer** in the driver's **IO_STACK_LOCATION** structure, and the output buffer's address is specified by **Irp->UserBuffer**.

Buffer sizes are supplied by **Parameters.DeviceIoControl.InputBufferLength** and **Parameters.DeviceIoControl.OutputBufferLength** in the driver's **IO_STACK_LOCATION** structure.

אבל למה?

הסיבה שבגללה התעכבתי הרבה על הערך של ה-IOCTL code היא שהבנה של כל הדקויות האלו בזמן הנדסה לאחור של driver תוכל לעזור לנו להבין לאן בדיוק הקלט-פלט שלנו יגיע, לאן הגדלים הרלוונטיים יגיעו ותיאור של הזיכרון הכללי שרלוונטי לפעולה. כך נוכל לחסוך זמן ולהבין בצורה טובה יותר את הפעילות של ה-driver ולהבין מה כל משתנה / ערך מייצג.

נקודות נוספות ב-DeviceControl dispatch

הפעולה הזאת אינה חריגה, וכמו כל פעולת dispatch שה-driver מכין, גם פעולה זו צריכה להסתיים בקריאה ל-IoCompleteRequest(Irp,...). הקריאה הזאת תשלח את ה-IRP הלאה ברשימה של מי שצריך לקבל את ה-IRP, לדוגמא: אם הבקשה נשלחה ל-device של מקלדת, ה-IRP של הבקשה קודם כל יצטרך לעבור אצל כל driver הרשום עבור כל device במסלול אל ה-device של המקלדת. במקרה שלנו שבו אין עוד המשך במסלול (virtual device, רק מגיע ל-driver וחוזר), הנתונים שחזרו דרך ה-IRP יועתקו אל המקומות המתאימים בזיכרון של התוכנה שיזמה את הבקשה.

דבר נוסף שחשוב לציין הם המסלולים הבאים:

```
Irp->IoStatus.Status = Status;
Irp->IoStatus.Information = 8;
IoCompleteRequest(Irp, IO_NO_INCREMENT);
```

- Status: מקבל את ה-NTSTATUS שחזר מהפעולה, מתאר אם הפעולה הצליחה ואם לא אז מה בדיוק כשל בפעולה
- Information: מקבל את הערך של כמות המידע שה-driver עשה בו שימוש לאורך הפעולה, לדוגמא, אם סיפקתי struct בגודל 30 בתים והוא כולו היה הכרחי לפעולה, הערך יחזור כ-30.

אינטרפרטציה של פרמטרים ל-driver ב-IDA

כל כלי להנדסה לאחור יכול להראות את הפרמטרים האלו ואת ההתייחסות אליהם בצורה שונה, אך מניסיון וטעייה שלי וטיפה היגיון של שימוש בכל פרמטר ניתן להבין את האינטרפרטציה ש-IDA pro עושה לכל מיני נתונים שמגיעים ל-driver:

- באפר קלט: CurrentStackLocation->Parameters.CreatePipe.Parameters
- באפר פלט: Irp->UserBuffer
- גודל קלט: CurrentStackLocation->Parameters.Create.Options
- גודל פלט: CurrentStackLocation->Parameters.Read.Length
- בעת שימוש ב-METHOD_BUFFERED: לפעמים ה-Buffer I/O יסומן כ-Irp->AssociatedIrp.MasterIrp

הבנת האינטרפרטציה של הכלי שבו אני משתמש להנדסה לאחור לא הכרחית, אך היא יכולה לעזור לראות את ההיגיון מאחורי כל פעולה ב-driver בצורה משמעותית מאוד, לכן גם אתייחס אליה בדוגמאות הפרקטיות של הנדסה לאחור ב-KM drivers.

סוגי חולשות נפוצות ב-Driver-ים

כמו בתוכנות בינאריות אחרות גם בדרייברים נתעסק עם כמה סוגים של חולשות:

1) חולשות לוגיות: החולשה הכי נפוצה בדרייברים. החולשה בדרך כלל באה מחוסר סינון של בקשה שמגיעה ל-driver, לדוגמא כתיבה של זיכרון לכתובת המסופקת על ידי הקורא מבלי לוודא שהקורא הוא תהליך שניתן לסמוך עליו ולא סתם תהליך שרוצה פרימיטיב כתיבה לזיכרון מערכת.

2) חולשות overflow: ניהול זיכרון בתוכנה קרנלית יכול להיות מסובך ככל שהתוכנה גודלת, ולכן הרבה כותבים שוכחים לוודא שהזיכרון שהוקצה באמת הוקצה בהתאמה לגודל שסופק מהקורא, אחרת יהיה overflow שיגרום ל-BSoD במקרה הטוב ולדריסה של זיכרון מערכת חשוב במקרה הרע.

3) חולשות מבוססות חוסר ואלידציה: אלו חולשות שמוכלות בתוך חולשות לוגיות, אך עדיין נפוצות מאוד. לא מעט כותבי driver-ים יהיו מוכנים לקבל מצביע לפעולה קרנלית מהקורא ולהריץ אותה מבלי לוודא שהכתובת היא לא NULL ושהכתובת ואלידית (את זה בדרך כלל נעשה עם ProbeForRead/Write, הפעולה תסרוק את הטווח של הזיכרון ובמקרה וחלק\כל הזיכרון לא ואלידי הפעולה תעלה exception שניתן לתפוס)

כעת, למחקר!

דוגמאות ל-Reverse Engineering של Driver-ים

בחלק זה אני הולך להדגים את תהליך החשיבה שלי כשאני מנסה להנדס לאחור driver, אני אציין את המידע המשמעותי מכל חלק תיאורטי שעברתי עליו במאמר ואראה איך הוא מתבטא במציאות במקרה ריאליסטי.

Vmkbd.sys

ה-driver הבא הוא ה-driver שאחראי למקלדת הוירטואלית שכל מכונה וירטואלית של vmware עושה בו שימוש. Vmkbd.sys נמצא כבר חולשתי ב-2010, ומאז עשו בו שינוי שהוכיח את עצמו וחסם את הפרצה, למרות זאת רציתי לעבור על ה-driver ולוודא שאין משהו חולשתי אחר שפספסו במקרה, הרצון הזה הסתיים בכישלון אך מהתהליך למדתי הרבה ממה שאני תיארתי מוקד יותר במאמר.

ל-driver הזה ולבאים נשתמש ב-IDA pro כדי לבצע הנדסה לאחור.

DriverEntry

פעולת הכניסה של ה-driver מתחילה בבדיקות נוספות שכנראה רלוונטיות לפעילות של ה-driver, כגון בדיקה של גרסת מערכת ההפעלה והסתכלות על כמה מהערכים הרשומים ב-registry key של ה-driver:

```
NTSTATUS __stdcall DriverEntry(_DRIVER_OBJECT *DriverObject, PUNICODE_STRING RegistryPath)
{
  _DWORD *KeyValueInformation; // [rsp+38h] [rbp-1A0h]
  unsigned int i; // [rsp+40h] [rbp-198h]
  DWORD dwMajorVersion; // [rsp+44h] [rbp-194h]
  DWORD dwMinorVersion; // [rsp+50h] [rbp-188h]
  void *KeyHandle; // [rsp+58h] [rbp-180h] BYREF
  NTSTATUS Version; // [rsp+60h] [rbp-178h]
  ULONG ResultLength; // [rsp+64h] [rbp-174h] BYREF
  struct _OBJECT_ATTRIBUTES ObjectAttributes; // [rsp+68h] [rbp-170h] BYREF
  struct _OSVERSIONINFOW VersionInformation; // [rsp+A0h] [rbp-138h] BYREF

  VersionInformation.dwOSVersionInfoSize = 276;
  Version = RtlGetVersion(&VersionInformation);
  if ( Version >= 0 )
  {
    dwMajorVersion = VersionInformation.dwMajorVersion;
    dwMinorVersion = VersionInformation.dwMinorVersion;
  }
  else
  {
    dwMajorVersion = 5;
    dwMinorVersion = 0;
  }
```

```
  if ( dwMajorVersion > 6 || dwMajorVersion == 6 && dwMinorVersion >= 2 )
  {
    PoolType = 512;
    dword_140006254 = 0x40000000;
  }
  ObjectAttributes.Length = 48;
  ObjectAttributes.RootDirectory = 0i64;
  ObjectAttributes.Attributes = 576;
  ObjectAttributes.ObjectName = RegistryPath;
  ObjectAttributes.SecurityDescriptor = 0i64;
  ObjectAttributes.SecurityQualityOfService = 0i64;
  if ( ZwOpenKey(&KeyHandle, 0x20019u, &ObjectAttributes) >= 0 )
  {
    KeyValueInformation = ExAllocatePoolWithTag(PoolType, 0x14ui64, 0x626B6D76u);
    if ( !KeyValueInformation )
    {
      ZwClose(KeyHandle);
      return -1073741801;
    }
    if ( ZwQueryValueKey(KeyHandle, &ValueName, KeyValuePartialInformation, KeyValueInformation
      && KeyValueInformation[1] == 4
      && KeyValueInformation[2] == 4i64
      && KeyValueInformation[3] )
```

לא נתמקד בערכים שנבדקים כאן כי נראה שהם רק מדליקים דגלים השמורים ב-driver ולא רלוונטיים לנו כרגע.

לאחר מכן ניתן לראות את אחד מהחלקים שהתייחסתי אליהם - רישום callbacks ל-driver עבור בקשות שונות שיכולות להשלח ל-driver.

אפשר לראות זאת בתמונה הבאה:

```
for ( i = 0; i <= 0x1B; ++i )
  DriverObject->MajorFunction[i] = (PDRIVER_DISPATCH)sub_140002550;
DriverObject->MajorFunction[27] = (PDRIVER_DISPATCH)sub_1400032E0;
DriverObject->MajorFunction[22] = (PDRIVER_DISPATCH)sub_140003584;
DriverObject->MajorFunction[3] = (PDRIVER_DISPATCH)sub_140003628;
DriverObject->MajorFunction[0] = (PDRIVER_DISPATCH)sub_14000247C;
DriverObject->MajorFunction[2] = (PDRIVER_DISPATCH)sub_14000247C;
DriverObject->MajorFunction[14] = (PDRIVER_DISPATCH)DriverIoControl;
DriverObject->MajorFunction[18] = (PDRIVER_DISPATCH)sub_14000234C;
DriverObject->MajorFunction[15] = (PDRIVER_DISPATCH)sub_140002B94;
DriverObject->DriverUnload = (PDRIVER_UNLOAD)sub_140003F84;
DriverObject->DriverExtension->AddDevice = (PDRIVER_ADD_DEVICE)sub_1400021A0;
return 0;
```

כפי שניתן לראות פה מבוצע רישום של כמה callbacks, ביניהם גם מבוצע רישום ל-callback באינדקס 14 של ה-major function table (הערך של IRP_MJ_DEVICE_CONTROL הוא 14, בדיוק מה שאנחנו מחפשים). אך כפי שניתן לראות כאן הפעולה DriverEntry חוזרת עם סטטוס הצלחה (STATUS_SUCCESS שווה בערכו ל-0) מבלי ליצור device או ליצור symbolic link עבור תקשורת אפשרית עם רכיבי UM.

כבר פה ניתן לראות שאין לנו שום אפשרות להשמיש כל חולשה שהיא ב-driver, אבל עדיין החלטתי להמשיך לחקור את ה-driver בשביל לראות אם יש משהו מעניין.

עוד משהו שניתן לראות פה זה האתחול של כל מערך ה-Major Functions בפעולה דיפולטיבית שמחזירה סטטוס STATUS_INVALID_DEVICE_REQUEST (הבקשה לא נתמכת) עבור כל פעולה שלא מאותחלת בהמשך. זה common practice שעושים חלק מכותבי driver-ים, אין לזה משמעות במחקר שלנו אך בצורה כזו "נטפל" בכל בקשה אפשרית ל-driver במקום להתעלם ממנה לגמרי.

DeviceControl dispatch

בפעולת ה-DeviceControl באמת ניתן לראות את ה-switchcase שציפינו לו, בגלל שיש הרבה IOCTL-ים שה-driver מטפל בהם עברתי על כל אחד מהם ובדקתי אם אני שולט על הפרמטרים המשפיעים על דרך ריצת הפעולה.

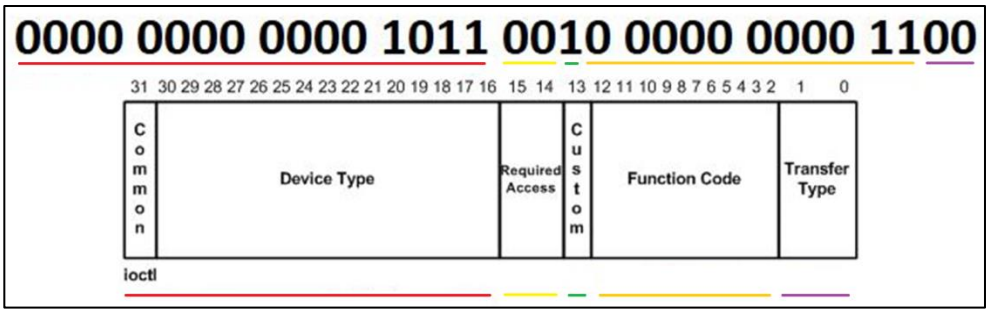
באותם IOCTL-ים שבהם לא מתייחסים לפרמטרים שאני מספק והפעולה עצמה לא חולשתית לא אתרכז.

```
LowPart = CurrentStackLocation->Parameters.Read.ByteOffset.LowPart;
if ( LowPart > 0xB2040 )
{
  switch ( LowPart )
  {
    case 0xB2044u:
      ReturnStatus = sub_1400028EC(Irp);      // COVERED
      break;
    case 0xB204Cu:
      ReturnStatus = sub_140002A50(Irp);      // COVERED
      break;
    case 0xB2050u:
      ReturnStatus = sub_140003DBC(Irp, 0i64);// CANNOT BE MANIPULATED BY ME
      break;
    case 0xB2054u:
      LOBYTE(v3) = 1;                         // CANNOT BE MANIPULATED BY ME
      ReturnStatus = sub_140003DBC(Irp, v3);
      break;
    default:
```

```
else
{
  switch ( LowPart )
  {
    case 0xB2040u:
      ReturnStatus = sub_14000261C(Irp);      // COVERED
      break;
    case 0xB200Cu:
      if ( CurrentStackLocation->Parameters.Create.Options >= 8ui64 )
      {
        MasterIrp = Irp->AssociatedIrp.MasterIrp;
        Handle = *(_QWORD *)&MasterIrp->Type; // First 8 bytes of input are a handle
        if ( IoIs32bitProcess(Irp) )
          Handle = (unsigned int)Handle;
        KeReleaseSpinLock((PKSPIN_LOCK)&Lock[3].Common.Removed, NewIrql);
        ReturnStatus = ObReferenceObjectByHandle(
                         (HANDLE)Handle,
                         2u,
                         (POBJECT_TYPE)ExEventObjectType,
                         Irp->RequestorMode,
                         &EventObject,
                         0i64);
        NewIrql = KeAcquireSpinLockRaiseToDpc((PKSPIN_LOCK)&Lock[3].Common.Removed);
        if ( ReturnStatus >= 0 )
        {
          ReturnStatus = FunctionOnUserSuppliedEvent((__int64)EventObject);// Only if RefrenceByHandle
                                                   // returns STATUS_SUCCESS
```

```
          if ( ReturnStatus < 0 )
            ObfDereferenceObject(EventObject);// If the function return status != STATUS_SUCCESS
        }
      }
      else
      {
        ReturnStatus = 0xC0000023;            // STATUS_BUFFER_TOO_SMALL, size validation for input buffer
      }
      break;
```

לפני שנעבור ל-IOCTL codes, נראה את המידע שניתן להשיג מהקוד של ה-IOCTL האחרון בתמונות במקרה והוא באמת נוצר עם ה-CTL_CODE macro:



אם ה-IOCTL code באמת נוצר עם CTL_CODE, נוכל לומר ש:

1) Transfer type הוא METHOD_BUFFERERD (0), זה מתאים להשערה שלנו בגלל השימוש ב-MasterIrp כבאפר קלט-פלט.

2) Device type הוא 0xd, הערך הזה מוזר בגלל שהערך של FILE_DEVICE_UNKNOWN הוא 0x22 ובדרך כלל זה הערך ש-drivers מספקים.

3) Function code הוא 3, ערך זה נראה מאוד מוזר במיוחד בגלל הקונבנציה שציינתי בהסבר על ה-macro, ומכל הנקודות הללו אני נוטה לחשוב שהכותב לא השתמש ב-macro ולכן כנראה לא ניתן להשיג מידע נוסף מה-IOCTL code.

IOCTL number 0xb200c

בעזרת ה"קיצורי דרך" בהנדסה לאחור של kernel drivers ב-IDA ניתן לראות שגודל הקלט חייב להיות לפחות 8 בתים (בפועל נעשה שימוש רק ב-8 הבתים הראשונים אבל לא יפריע אם יסופק באפר של יותר מ-8 בתים), וניתן לראות שה-8 בתים מהבאפר קלט מייצגים HANDLE ל-event.

הפעולה ObReferenceObjectByHandle היא פעולה קרנלית שנועדה להשיג מידע פנימי על HANDLE שמסופק לפעולה. כבר פה ניתן לראות שליטה על הפרמטר של ה-handle לפעולה, אך אם נספק handle ל-event לא קיים או כתובת לא ואלידית הפעולה פשוט תכשל עם סטטוס **STATUS_INVALID_HANDLE**. אני חשבתי גם על אפשרות להשיג handle ואלידי ל-event ולקרוא לפעולה כמה שיותר פעמים.

עבור כל אובייקט נספר כמות ה-references שמבקשים אליו, פעולה כמו ObReferenceObjectByHandle תעלה את ה-counter של ה-references וקריאה לפעולה כמו ObDereferenceObject תוריד את ה-counter.

אך בגלל שהפעולה הנקראת פה היא ObReferenceObjectByHandle שעושה ואלידציה לקריאה אני לא יכול לגרום למחיקת האובייקט או DoS למשתמש לגיטימי ב-event:

> **Remarks**
>
> **ObfReferenceObject** simply increments the pointer reference count for an object, without making any access checks on the given object, as ObReferenceObjectByHandle and ObReferenceObjectByPointer do.

[מקור: https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-obfreferenceobject]

IOCTL number 0xb2044

הפעולה שנקראת על ידי פעולת ה-dispatch מאוד פשוטה, הפעולה מקבלת באפר קלט-פלט באורך של (לפחות) 8 בתים, שמה ב-4 בתים הראשונים את הערך 1 וב-4 בתים הבאים היא שמה את הערך 6.

אין פה אפשרות לספק באפר לא מתאים או לשנות את הפונקציונליות המקורית כי אם הגודל של הבאפר גדול או שווה ל-8 בתים זה אומר שבהכרח יהיה מספיק זיכרון לכתיבה ושבהכרח הכתיבה תבוצע לאותו זכרון:

```
__int64 __fastcall sub_1400028EC(_QWORD *Irp)
{
  _DWORD *InputBuffer; // [rsp+8h] [rbp-20h]

  InputBuffer = (_DWORD *)Irp[3];      // This is probably the output buffer
  if ( *(unsigned int *)(Irp[23] + 8i64) < 8ui64 )
    return 0xC0000023i64;              // STATUS_BUFFER_TOO_SMALL, probably output buffer
  *InputBuffer = 1;                    // first 4 bytes will hold 1 inside output buffer
  InputBuffer[1] = 6;                  // second 4 bytes will hold 6 inside output buffer
  Irp[7] = 8i64;                       // probably is Irp->Information (operated on count in bytes)
  return 0i64;
}
```

IOCTL number 0xb204c

הפעולה מתחילה בואלידציה של כל נתוני הקלט שאני מספק לפעולה, כגון בדיקה של אורך הקלט ואורך הפלט (שצריכים להיות שניהם לפחות 4 בתים) ובאפר הקלט (צריך להיות קיים ולא שווה ל-NULL, בנוסף לכך צריך להכיל ערך בין 0 ל-1024 ב-4 בתים הראשונים שלו).

לאחר מכן מבוצע חישוב עם הערך שנמצא בקלט והאורך של הקלט, ואם החישוב עובד והערך לא שווה לאפס נקראת פעולה אחרת. אני לא אכנס לפעולה שנקראת בגלל שאין בה פרימיטיב אפשרי לתקיפה והיא רק מבצעת כמה חישובים ומכניסה ערך תוצאה לבאפר שסיפקתי.

אין כאן שליטה או מניפולציה אפשרית בכל שלב שהוא ולכן גם ה-IOCTL הזה לא חולשתי (כנראה):

```
__int64 __fastcall sub_140002A50(__int64 a1)
{
  unsigned int v2; // [rsp+20h] [rbp-38h]
  unsigned int InputOrOutputLength; // [rsp+24h] [rbp-34h]
  int v4; // [rsp+28h] [rbp-30h] BYREF
  unsigned int OutputOrInputSize; // [rsp+2Ch] [rbp-2Ch]
  int v6; // [rsp+30h] [rbp-28h]
  int *InputBuffer; // [rsp+38h] [rbp-20h]
  __int64 v8; // [rsp+40h] [rbp-18h]
  _DWORD *InputBufferDWORDalt; // [rsp+48h] [rbp-10h]

  v8 = *(_QWORD *)(a1 + 184);
  InputBuffer = *(int **)(a1 + 24); // Is probably the input buffer, validation for != NULL is done
  InputBufferDWORDalt = *(_DWORD **)(a1 + 24);
  OutputOrInputSize = *(_DWORD *)(v8 + 8);
  InputOrOutputLength = *(_DWORD *)(v8 + 16);
  v4 = 0;
  v2 = 0;
  if ( InputBuffer )
  {
    if ( InputOrOutputLength >= 4ui64 )
    {
      if ( OutputOrInputSize >= 4ui64 )
      {
        if ( *InputBuffer <= 1024 )
        {
          if ( *InputBuffer >= 0 )
          {
            v6 = 28 * (*InputBuffer - 1) + 32;
            if ( InputOrOutputLength == v6 )
            {
              if ( *InputBuffer )
                v2 = sub_1400014BC(InputBuffer, &v4);
              *InputBufferDWORDalt = v4;
              *(_QWORD *)(a1 + 56) = 4i64; // Probably is Irp.information, amount of bytes operated on
            }
            else
            {
              v2 = 0xC0000206;            // STATUS_INVALID_BUFFER_SIZE
            }
          }
          else
          {
            v2 = 0xC000000D;              // STATUS_INVALID_PARAMETER
          }
        }
        else
        {
```

IOCTL number 0xb2040

כמו שלושת הניסיונות הקודמים, גם כאן ניתן לראות שאין אפשרות לניצול כלשהו, הפעולה מוודאת שבאפר הקלט לפחות באורך 20 בתים ואם כן היא נכנסת ללולאה שמבצעת חישובים ומכניסה ערכים מסוימים לאותו באפר קלט-פלט:

```
__int64 __fastcall sub_14000261C(_QWORD *a1)
{
  unsigned int v2; // [rsp+24h] [rbp-74h]
  int *InputBuffer; // [rsp+28h] [rbp-70h]
  unsigned int InputBufferLength; // [rsp+38h] [rbp-60h]
  __int64 v5; // [rsp+40h] [rbp-58h]
  LIST_ENTRY *p_WaitListHead; // [rsp+48h] [rbp-50h]
  _LIST_ENTRY *Flink; // [rsp+50h] [rbp-48h]
  _LIST_ENTRY *v8; // [rsp+58h] [rbp-40h]
  HANDLE CurrentProcessId; // [rsp+78h] [rbp-20h]

  InputBuffer = (int *)a1[3];                    // Probably is the input buffer
  InputBufferLength = *(_DWORD *)(a1[23] + 8i64);
  CurrentProcessId = PsGetCurrentProcessId();
  if ( !*(_QWORD *)&Lock[4].Common.RemoveEvent.Header.Lock )
    return 0xC0000010i64;                        // STATUS_INVALID_DEVICE_REQUEST
  if ( InputBufferLength < 20ui64 )
    return 0xC0000206i64;                        // INVALID_BUFFER_SIZE, should be 20 bytes or more
  *InputBuffer = 0;
  *((_BYTE *)InputBuffer + 4) = 0;               // Zeroes out the first 5 bytes inside the input buffer
  v2 = 8;
  if ( **(HANDLE **)&Lock[4].Common.RemoveEvent.Header.Lock == CurrentProcessId )
  {
```

```
  while ( v2 < (unsigned __int64)InputBufferLength - 12
        && Lock[4].Common.RemoveEvent.Header.WaitListHead.Flink != &Lock[4].Common.RemoveEvent.Header.WaitListH
  {
    p_WaitListHead = &Lock[4].Common.RemoveEvent.Header.WaitListHead;
    Flink = Lock[4].Common.RemoveEvent.Header.WaitListHead.Flink;
    v8 = Flink->Flink;
    if ( Flink->Blink != &Lock[4].Common.RemoveEvent.Header.WaitListHead || v8->Blink != Flink )
      __fastfail(3u);
    p_WaitListHead->Flink = v8;
    v8->Blink = p_WaitListHead;
    --*(_DWORD *)&Lock[5].Common.Removed;
    v5 = (__int64)&InputBuffer[3 * *InputBuffer + 2];
    *(_WORD *)v5 = Flink[-1].Flink;
    *(_WORD *)(v5 + 2) = WORD1(Flink[-1].Flink);
    *(_WORD *)(v5 + 4) = WORD2(Flink[-1].Flink);
    *(_WORD *)(v5 + 6) = HIWORD(Flink[-1].Flink);
    *(_DWORD *)(v5 + 8) = Flink[-1].Blink;
    ++*InputBuffer;
    v2 += 12;
    ExFreePoolWithTag(&Flink[-1], 0);
  }
```

סיכום ה-driver

כפי שניתן לראות מהמחקר, אין משטח שיכול בכלל להיות חולשתי ב-driver, בין אם אני מפעיל פעולות שלא מתייחסות לקלט-פלט שלי ולא משפיעות על שום דבר משמעותי במערכת, ובין אם הקלט-פלט לא יכול להפעיל את ה-driver בצורה חולשתית, לכן ניתן לראות שלמרות המחקר ה-driver כנראה לא חולשתי (וגם אם כן היה הוא לא היה ניתן להשמשה בגלל העיקרון שהסברתי בתחילת הסריקה).

VULNERABILITIES

NOTICE UPDATED - MAY, 29TH 2024

The NVD has a new announcement page with status updates, news, and how to stay connected!

CVE-2023-20598 Detail

MODIFIED

This vulnerability has been modified since it was last analyzed by the NVD. It is awaiting reanalysis which may result in further changes to the information provided.

Description

An improper privilege management in the AMD Radeon™ Graphics driver may allow an authenticated attacker to craft an IOCTL request to gain I/O control over arbitrary hardware ports or physical addresses resulting in a potential arbitrary code execution.

QUICK INFO

CVE Dictionary Entry:
CVE-2023-20598
NVD Published Date:
10/17/2023
NVD Last Modified:
07/02/2024
Source:
Advanced Micro Devices Inc.

ה-driver שנעבור עליו הוא driver של AMD שנתן לתוקף אפשרות לשלוט על זיכרון פיזי ועל hardware ports. לתוך hardware ports אכנס במאמר הבא אך שליטה על כל הזיכרון הפיזי היא חולשה משמעותית, כפי שניתן לראות מהניקוד של החולשה:

Metrics

CVSS Version 4.0 | CVSS Version 3.x | CVSS Version 2.0

NVD enrichment efforts reference publicly available information to associate vector strings. CVSS information contributed by other sources is also displayed.

CVSS 3.x Severity and Vector Strings:

NIST: NVD	**Base Score:** 7.8 HIGH	**Vector:** CVSS:3.1/AV:L/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:H
ADP: CISA-ADP	**Base Score:** 7.8 HIGH	**Vector:** CVSS:3.1/AV:L/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:H

References to Advisories, Solutions, and Tools

By selecting these links, you will be leaving NIST webspace. We have provided these links to other web sites because they may have information that would be of interest to you. No inferences should be drawn on account of other sites being referenced, or not, from this page. There may be other web sites that are more appropriate for your purpose. NIST does not necessarily endorse the views expressed, or concur with the facts presented on these sites. Further, NIST does not endorse any commercial products that may be mentioned on these sites. Please address comments about this page to nvd@nist.gov.

Hyperlink	Resource
https://www.amd.com/en/corporate/product-security/bulletin/AMD-SB-6009	Vendor Advisory

Weakness Enumeration

CWE-ID	CWE Name	Source
NVD-CWE-noinfo	Insufficient Information	NIST
CWE-269	Improper Privilege Management	CISA-ADP

ניתן גם לראות בתחתית התמונה את סיבת החולשה: improper privilege management (ניהול לא נכון של גישה להרשאות שה-driver נותן, מה שאמרתי שקורה ברוב ה-vulnerable drivers).

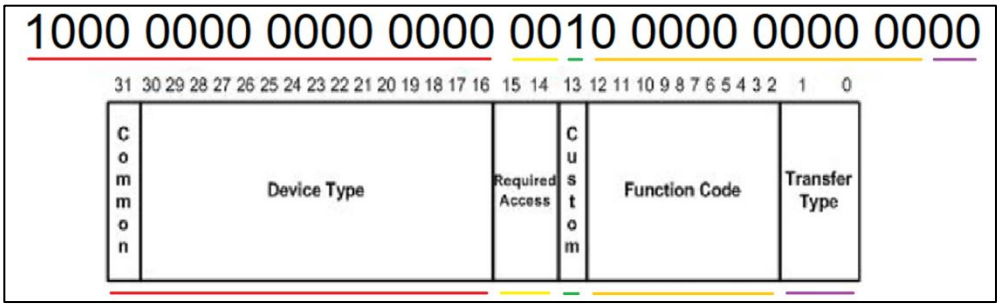
DriverEntry

פעולת הכניסה של ה-driver נראית בדיוק כמו הבסיס ההכרחי לפעולת כניסה: רישום symbolic link, יצירת device לתקשורת ורישום callbacks כולל DeviceControl:

```
NTSTATUS __stdcall DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath)
{
  NTSTATUS result; // eax
  NTSTATUS v4; // ebx
  struct _UNICODE_STRING DestinationString; // [rsp+40h] [rbp-28h] BYREF
  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);
    v4 = result;
    if ( result )
    {
      IoDeleteDevice(DeviceObject);
      return v4;
    }
  }
  return result;
```

DeviceControl dispatch

בפעולת ה-DeviceControl באמת ניתן לראות את ה-switchcase שציפינו לו. כמו כן ניתן לראות את הפורמט הרגיל והצפוי מפעולת DispatchControl כמו התייחסות ל-CurrentStackLocation (לא חובה, ניתן גם להתייחס ישירות למסלול האבסולוטי מ-UserIrp אך אז המסלולים יהפכו ליותר מסורבלים וארוכים), הוצאת גודל הקלט לפי האינטרפרטציה של הפרמטרים אצל IDA ולפי אותו עקרון גם שמירה של באפר הקלט-פלט במשתנה נפרד לפי UserIrp->AssociatedIrp->MasterIrp (מה שאומר שה-driver מוכן להעברת פרמטרים בשיטת METHOD_BUFFERED, זאת בעצם השיטה הדיפולטיבית אם לא אומרים ל-memory manager משהו אחר). בגלל שניתן לראות שכל ה-IOCTL codes בפורמט דומה, אני אנתח את ה-IOCTL code הראשון כדי לראות אם נעשה שימוש ב-CTL_CODE והאם ניתן להוציא מזה עוד מידע על ה-driver:



אם ה-IOCTL code באמת נוצר עם CTL_CODE, נוכל לומר ש:

1) Transfer type הוא METHOD_BUFFERERD (0), זה מתאים להשערה שלנו בגלל השימוש ב-MasterIrp כבאפר קלט-פלט
2) Device type הוא 0x8000, הערך הזה מראה בוודאות שלא נעשה שימוש ב-macro, או שלפחות השימוש לא היה נכון בגלל שאין device type שעולה מ-2 ספרות הקסה-דצימליות
3) Function code הוא 0, ערך זה נראה מאוד מוזר במיוחד בגלל הקונבנציה שציינתי בהסבר על ה-macro, ומכל הנקודות הללו אני אומר בביטחון שהכותב לא השתמש ב-macro ולכן לא ניתן להשיג מידע נוסף מה-IOCTL code

IOCTL number 0x80002000

הפעולה מתחילה ביצירת I/O space (מטרת I/O space היא מיפוי מרחב כתובות פיזיות לזיכרון מערכת וירטואלי non-paged עבור מטרות שונות כמו מניפולציה של האיזור זיכרון ושינוי ערכים). הפעולה מקבלת כקלט כתובת פיזית בסיס של המרחב ב-8 בתים הראשונים וגודל של מרחב הכתובות ב-8 בתים השניים ולאחר יצירת I/O space היא יוצרת MDL המתאר את אותו מרחב כתובות ונועלת אותו לזיכרון non-paged כדי שישאר בזיכרון כל הזמן.

בסופו של דבר כשהזיכרון ממופה בעזרת ה-MDL מתבצעת קריאה של ערך בגודל 4 בתים מכתובת הבסיס הפיזית והערך של הקריאה חוזר לקורא ב-4 בתים הראשונים של באפר הקלט-פלט. במקרה הזה יש arbitrary read ברור לכל כתובת פיזית אפשרית שיכול להיות שימושי למגוון התקפות:

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

IOCTL number 0x80002004

גם פה מבוצע אותו התהליך (יצירת I/O space לתיאור מרחב הכתובות הפיזי, יצירת MDL לתיאור ומניפולציה, נעילת הזיכרון כ-non-paged וההפך כשיוצאים מהפעולה כדי לנקות את הכל), רק שכאן ה-4 בתים הראשונים הם קלט של ערך לכתיבה בכתובת זיכרון פיזית מסוימת, מה שהופך את החולשה ל-

arbitrary write ברור ומוחלט על כל כתובת פיזית שאני מספק לפעולה. בעזרת arbitrary write כזה ניתן לבצע הרבה התקפות, מכתיבה לכתובת לא ואלידית כדי לגרום ל-DoS/BSoD של כל המערכת, עד לשינוי של נתונים קריטיים ומבני נתונים של מערכת ההפעלה:

```
    case 0x80002004:
      if ( InputSize != 16 )
        goto InvalidParameterLabel;
```

```
        v12 = MmMapIoSpace(*MasterIrp, 4ui64, MmNonCached);
        IoSpaceForFreeing = v12;
        if ( !v12 )
          goto InsufficientResourcesLabel;
        v13 = IoAllocateMdl(v12, MasterIrp[1].LowPart, 0, 0, 0i64);
        v14 = v13;
        if ( v13 )
        {
          MmBuildMdlForNonPagedPool(v13);
          v15 = MmMapLockedPages(v14, 0);
          *v15 = MasterIrp[1].HighPart;
          MmUnmapLockedPages(v15, v14);
          IoFreeMdl(v14);
          ProvidedRangeSize = 4i64;
UnmapAndSuccess:
          MmUnmapIoSpace(IoSpaceForFreeing, ProvidedRangeSize);
          UserIrp->IoStatus.Information = 4i64;
        }
        else
        {
UnmapAndInsufficient:
          MmUnmapIoSpace(IoSpaceForFreeing, MasterIrp[1].LowPart);
InsufficientResourcesLabel:
          UserIrp->IoStatus.Status = 0xC000009A;// STATUS_INSUFFICIENT_RESOURCES
        }
        break;                                  // DONE :)
```

IOCTL numbers 0x80002008+0x8000200C

שתי הפעולות הללו מאוד דומות במבנה ובפונקציונליות שלהן, אך כדי להבין אותן נצטרך להבין מה המושג hardware ports אומר:

> A **computer port** is a hardware piece on a computer where an electrical connector can be plugged to link the device to external devices, such as another computer, a peripheral device or network equipment.[1]

אז בעצם Hardware port הוא רכיב חומרה המקשר בין המערכת שלנו לבין מכשירים חיצוניים, כגון עכבר או מקלדת שאנחנו משתמשים בהם. בהקשר שלנו זה מראה לנו ששליטה על ה-hardware ports שמחוברים למערכת תביא לנו משטח אינסופי לתקיפה, אם זה keylogging דרך "hook" ל-hardware port שאליו מחוברת המקלדת וקריאה של מידע רגיש המוקלד (סיסמאות..), אם זה כתיבה ל-hardware ports של מכשירים קריטיים באיזור שלנו שמקבלים קלט מהמערכת שלנו ופועלים לפיו, האפשרויות לא נגמרות.

במקרה זה כותב ה-driver הביא לנו גישה ישירה לכתוב איזה מידע שבא לנו וכמה מידע שבא לנו לאיזה Hardware port שבא לנו, הפעולה ()indword_ קוראת ערך בגודל 4 בתים מתוך hardware port שמסופק לה כפרמטר ומחזירה אותו, בזמן ש-()outdword_ כותבת ערך בגודל 4 בתים לתוך hardware port שגם הוא וגם הערך מסופקים לפעולה כפרמטרים.

בשני המקרים ניתן לראות שהבאפר קלט-פלט הכללי צריך להיות בגודל 12 בתים. מתוכם, 4 הבתים הראשונים מכילים את מספר ה-hardware port שרוצים לפעול עליו. 4 הבתים הבאים חייבים להכיל magic value ששווה ל-4.

ו-4 הבתים האחרונים צריכים להכיל את הערך שרוצים לכתוב ל-hardware port בפעולת כתיבה, לחלוטין ניתן להשאיר אותם ריקים. לאחר הפעולה נקבל כפלט את הערך הנקרא מה-hardware port (בפעולת קריאה):

```
    case 0x80002008:
```

```
      if ( InputSize != 12 || MasterIrp->HighPart != 4 )
        goto InvalidParameterLabel;
      v16 = __indword(MasterIrp->LowPart);
      MasterIrp[1].LowPart = v16;
      UserIrp->IoStatus.Information = 4i64;
      break;                                  // DONE :)
    case 0x8000200C:
      if ( InputSize != 12 || MasterIrp->HighPart != 4 )
        goto InvalidParameterLabel;
      __outdword(MasterIrp->LowPart, MasterIrp[1].LowPart);
      UserIrp->IoStatus.Information = 4i64;
      break;                                  // DONE :)
```

IOCTL number 0x80002010

בחלק זה ניתן לראות אלוקציה של זיכרון מערכת contiguous לפי גודל שאני בוחר ומספק. זיכרון מערכת contiguous הוא מרחב כתובות פיזיות רציף שניתן למפות לזיכרון מערכת וירטואלי non-paged, ולכן השליטה שניתנת לנו בפעולה זו על ידי ה-driver היא שימושית ומשמעותית מאוד.

בצורה כזו, נוכל לעשות הרבה דברים כגון אלוקציות זיכרון לתוכן שאנחנו רוצים לשמור (בעזרת פרימיטב ה-read/write לכתובות פיזיות שכבר תיארתי) ואפילו "DoS" למערכת מזיכרון פיזי וניתן לשימוש שיגרום לבעיות רבות במערכת.

באפר הקלט-פלט צריך להיות בגודל 16 בתים ובמקרה זה הקלט הוא ה-4 בתים הראשונים (כמות הזיכרון להקצות) והפלט כולל את הכתובת בסיס של מיפוי הזיכרון ה-contiguous לזיכרון מערכת non-paged ב-8 בתים הראשונים, ואפילו את הכתובת הפיזית המקבילה לאותה כתובת בסיס שציינתי ב-8 בתים הבאים.

השימושיות ב-contiguous system memory כנגד non-contiguous system memory הוא שאנחנו יכולים לוודא שהאלוקציה רציפה, ולכן אם לדוגמא נרצה לכתוב תוכן רציף לאותה בריכת זיכרון שהקצינו נוכל פשוט להעלות את כתובת הבסיס בגודל של הכתיבה וזאת תהיה הכתובת הפיזית הבאה של הכתיבה.

בגלל שהכתיבה / קריאה שלנו מבוססת על כתובות פיזיות, העיקרון הזה חשוב כדי שנוכל לפעול על כתובות פיזיות נכונות ומתאימות לכוונה שלנו:

```
case 0x80002010:
  MasterIrp[1].QuadPart = 0i64;
  MasterIrp[2].QuadPart = 0i64;
  if ( InputSize != 48 )
    goto InvalidParameterLabel;
  ContiguousMemorySpecifyCache = MmAllocateContiguousMemorySpecifyCache(
                                   MasterIrp->LowPart,
                                   0i64,
                                   (PHYSICAL_ADDRESS)0xFFFFFFFFFFFi64,
                                   0i64,
                                   MmNonCached);
  v18.QuadPart = (LONGLONG)ContiguousMemorySpecifyCache;
  if ( !ContiguousMemorySpecifyCache )
    goto InsufficientResourcesLabel;
  PhysicalAddress = MmGetPhysicalAddress(ContiguousMemorySpecifyCache);
```

```
  if ( !PhysicalAddress.LowPart )
    goto InsufficientResourcesLabel;
  MasterIrp[1] = PhysicalAddress;
  MasterIrp[2] = v18;
  UserIrp->IoStatus.Information = 48i64;
  break;                                  // DONE :)
```

IOCTL number 0x80002018

הפעולה הזו נמצאת בהתאמה ל-IOCTL code 0x80002010, כלומר היא נועדה לשחרר את אותו מרחב כתובות שהפעולה הקודמת נועדה להקצות. הפעולה מוכנה לקבל 48 בתים של קלט-פלט, בקבוצה של 8 הבתים השלישית בקלט (בתים 16 עד 24) נמצאת כתובת הבסיס הוירטואלית למיפוי של הזיכרון הפיזי המוקצה (הכרחית לשחרור זיכרון contiguous בעזרת הפעולה MmFreeContiguousMemory).

הכתובת שנספק בשימוש רגיל היא הכתובת שחזרה ישירות מהפעולה MmAllocateContiguousMemory אך במקרה הזה גם נוכל לספק כתובות לא ואלידיות כדי לגרום ל-DoS במערכת (יצירת BSoD שמתבסס על שחרור זיכרון שלא הוקצה\לא זיכרון קיים במערכת).

כמובן שעם פעולה זו נוכל גם לשחרר זיכרון לגיטימי שבתוכו מאוחסנים מבני נתונים קריטיים במערכת, לכן גם פעולה זו יכולה להביא לנו שליטה קריטית על המערכת ממגוון סוגים שונים:

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

IOCTL number 0x8000201C

פעולה זו היא הראשונה מזוג פעולות נוסף שה-driver מספק לנו. במקרה זה הפעולה מפעילה את ה-API שנקרא MmMapIoSpace לפי הפרמטרים שאנחנו מספקים לה.

כפי שתיארתי ב-IOCTL code 0x80002000, I/O space הוא תיאור של מרחב כתובות פיזי בזיכרון מערכת וירטואלי, ומסיבה זו מתוך ה-56 בתים של קלט-פלט שהפעולה מצפה לו ה-4 בתים הראשונים הם גודל המרחב כתובות וה-8 בתים השניים בסדר (בתים 8-16) כוללים את הכתובת הפיזית של בסיס מרחב הכתובות. לאחר הפעלת ה-API בסיס ה-I/O space חוזר ל-8 בתים במקום השישי בסדר (אינדקס 5 במערך שבו כל ערך הוא 8 בתים), בצורה זו אפשר ליצור מיפוי חופשי של כל מרחב כתובות פיזי שאני רוצה לכתובות וירטואליות בזיכרון מערכת שניתן להפעיל עליהם פעולות נוספות כמו כתיבה / קריאה:

```
case 0x8000201C:
  if ( InputSize != 56 )
    goto InvalidParameterLabel;
  PhysicalAddress_1 = MasterIrp[1];
  if ( !PhysicalAddress_1.QuadPart )
    goto InvalidParameterLabel;
  IoSpace = MmMapIoSpace(PhysicalAddress_1, MasterIrp->LowPart, MmNonCached);
  if ( !IoSpace )
    goto InsufficientResourcesLabel;
  MasterIrp[5].QuadPart = (LONGLONG)IoSpace;
  UserIrp->IoStatus.Information = 56i64;
  break;                                    // DONE :)
```

IOCTL number 0x80002024

כמו כן, הפעולה הזאת היא המקבילה לפעולה ב-IOCTL code 0x8000201C. הפעולה מקבלת כקלט 56 בתים, שמהם הערך בסדר שישי של מערך הקלט (אינדקס 5 במערך של ערכים באורך 8 בתים) הוא כתובת הבסיס של ה-I/O space בזיכרון המערכת שהוחזרה לנו מה-API שנקרא MmMapIoSpace, וה-4 בתים הראשונים במערך הקלט הוא הגודל של המרחב כתובות שה-I/O space מתאר:

```
case 0x80002024:
  if ( InputSize != 56 )
    goto InvalidParameterLabel;
  v23 = (void *)MasterIrp[5].QuadPart;
  if ( v23 )
  {
    MmUnmapIoSpace(v23, MasterIrp->LowPart);
    UserIrp->IoStatus.Information = 56i64;
    MasterIrp[5].QuadPart = 0i64;
  }
  break;                                    // DONE :)
```

IOCTL numbers: 0x80002014+0x80002020+0x80002028+0x8000202C. ה-IOCTL codes האלו מקבלים כמה פרמטרים מהקורא (בסך הכל באפר קלט-פלט בגודל של 48/56 בתים). בתוך הפעולה המרכזית של כל IOCTL נקראת פעולה נוספת שמבצעת הרבה חישובים מסובכים בהתאם לאותם נתונים.

בגלל שהחישובים שם ממש לא ברורים, ארוכים והרבה משם undocumented (כלומר מבחינת IDA הכל נראה כמו פעולות אריתמטיות, offsets והשמות אקראיות על ידי הכותב), בנוסף לעובדה שלא נראה

שלפעולות יש חשיבות משמעותית מבחינת פעילות ה-driver והחולשות שאני מחפש, החלטתי לא להעמיק במחקר הפעולה ולהמשיך לפעולות אחרות עם יותר פונטציאל.

מי שרוצה להמשיך ולחקור את הפעולות הללו מוזמן בכיף, אקשר את עמוד הגיטהאב שבו נמצא המחקר וההשמשה שלי שבו נמצא קובץ ה-driver המקורי וקובץ ה-IDA:

```
    case 0x80002014:
      if ( InputSize != 48 )
        goto InvalidParameterLabel;
      ((void (__fastcall *)(_QWORD, _QWORD, _QWORD))sub_1400016C0)(
        (PHYSICAL_ADDRESS)MasterIrp[2].QuadPart,
        (PHYSICAL_ADDRESS)MasterIrp[3].QuadPart,
        MasterIrp[5].LowPart);
      UserIrp->IoStatus.Information = 48i64;
      break;
    case 0x80002020:
      if ( InputSize != 56 )
        goto InvalidParameterLabel;
      ((void (__fastcall *)(_QWORD, _QWORD, _QWORD))sub_1400016C0)(
        (PHYSICAL_ADDRESS)MasterIrp[5].QuadPart,
        (PHYSICAL_ADDRESS)MasterIrp[3].QuadPart,
        MasterIrp[6].LowPart);
      UserIrp->IoStatus.Information = 56i64;
      break;
    case 0x80002028:
      if ( InputSize != 48 )
        goto InvalidParameterLabel;
      ((void (__fastcall *)(_QWORD, _QWORD, _QWORD))sub_1400016C0)(
        (PHYSICAL_ADDRESS)MasterIrp[4].QuadPart,
        (PHYSICAL_ADDRESS)MasterIrp[2].QuadPart,
        MasterIrp[5].LowPart);
      UserIrp->IoStatus.Information = 48i64;
      break;
    case 0x8000202C:
      if ( InputSize == 56 )
      {
        ((void (__fastcall *)(_QWORD, _QWORD, _QWORD))sub_1400016C0)(
          (PHYSICAL_ADDRESS)MasterIrp[4].QuadPart,
```

סיכום ה-driver

ניתן לראות פה שמצאתי מגוון חולשות המבוססות על ניהול הרשאות שגוי, מתוכן יש לי arbitrary read/write לכל זיכרון פיזי הקיים על המערכת, שליטה על אלוקציה ושחרור של זיכרון פיזי במערכת שיכול להיות שימושי להרבה מטרות: מיפוי זיכרון פיזי לזיכרון מערכת וירטואלי שיכול לעזור לעשות מניפולציה לזיכרון ושליטה מוחלטת על כל ה-hardware ports הקיימים על המערכת.

ההנדסה לאחור של ה-driver הזה הייתה "קלה" וברורה ביחס למקרים אחרים שיכולים להיות, אך זה רק מראה כמה כתיבה לא נכונה של driver על ידי חברות (במיוחד חברות גדולות כמו AMD שהמוצרים והמכשירים שלהם משומשים על ידי כמות עצומה של מערכות) יכולה לתת לתוקף שליטה מוחלטת על המערכת.

ההשמשות ל-CVE-2023-20598

בעקבות החולשות הנתונות שהיו לי מה-driver החולשתי ניסיתי לחשוב על PoC מתאים שיכול להשתמש בכל\רוב החולשות בצורה משמעותית שיכולה לדמות דוגמא ריאליסטית ל-exploit הכולל קוד קרנלי. בסופו של דבר לקחתי רעיון דומה למה שהצגתי במאמר הקודם שממש טעינת תוכנה בינארית לזיכרון מערכת.

בצורה כזו יכולתי לבחור גם unsigned driver שאני הכנתי ולממש משהו זהה לדוגמא שהצגתי במאמר הקודם, אך בגלל שבמקרה הזה אין לי אפשרות להריץ קוד החלטתי ליצור דוגמא מוחשית שבה אני אוכל להראות את שלב הטעינה של התוכנה לזיכרון מערכת.

תהליך ההשמשה יראה כך:

1) קריאת הקובץ שאני רוצה למפות לזיכרון מערכת (יכול להיות כל קובץ שאני רוצה, במקרה זה בחרתי קובץ exe כדי לדמות הטענה של קובץ קוד בינארי) לבאפר מקומי שאני אוכל לכתוב ממנו לזיכרון המערכת
2) להקצות זיכרון מערכת בעזרת 0x80002010, בצורה זו אוכל לוודא שכל הזיכרון שאני הקצתי מוצמד לזיכרון פיזי ושהוא רציף כך שאני אוכל לכתוב בחלקים של 4 בתים אליו, כל פעם לכתובת הגדולה ב-4 מהכתובת הקודמת
3) כניסה ללולאה שפועלת (גודל הקובץ למיפוי \ 4 בתים שנכתבים בכל פעם) פעמים, כל פעם הלולאה לוקחת את הכתובת הנוכחית בזיכרון ובעזרת 0x80002004 אני כותב את ה-4 בתים הבאים בתור של התוכנה למיפוי לתוך הכתובת זיכרון הנוכחית. לאחר הכתיבה אני מעלה את הכתובת הנוכחית ב4 כי הזיכרון הפיזי רציף
4) אחרי הלולאה אני משחרר את הזיכרון שהקצתי בהתחלה, אך שם breakpoint לפניו כדי שאני אוכל לראות את התוכנה בזיכרון מערכת בעזרת windbg שיכול להסתכל על כל כתובת זיכרון נתונה

בדוגמא שלי להשמשה ביצעתי מיפוי של netstat.exe לזיכרון המערכת, וכדי לראות את זיכרון המערכת שאליו התוכנה הוטענה ולוודא שהכל עבד השתמשתי בכתובת הוירטואלית המקבילה לבסיס האלוקציה הפיזי (כפי שציינתי פעולת האלוקציה מחזירה גם כתובת בסיס פיזית וגם כתובת בסיס וירטואלית).

ניתן לראות את כל תהליך ההשמשה ואת הקוד המלא של המחקר וההשמשה בקישור לעמוד גיטהאב עבור ה-driver החולשתי והקישור לסרטון ההשמשה המלא שהוספתי בסוף המאמר.

מחשבות נוספות וניסיון (כושל) לשדרוג ה-Exploit

מי שקרא את המאמר הקודם שלי מגיליון 163 יכול לראות פה דמיון כלשהו ל-unsigned driver mapper כמו kdmapper, ובכנות זה מה שכיוונתי אליו. רוב ה-unsigned driver mappers מותאמים לשינוי דינאמי מבחינת הפעולות שמבוצעות על ידי ה-driver החולשתי.

לדוגמא, כך נראת פקודת הקריאה של זיכרון מבחינת kdmapper:

```
bool intel_driver::MemCopy(HANDLE device_handle, uint64_t destination, uint64_t source, uint64_t size) {
    if (!destination || !source || !size)
            return 0;

    COPY_MEMORY_BUFFER_INFO copy_memory_buffer = { 0 };

    copy_memory_buffer.case_number = 0x33;
    copy_memory_buffer.source = source;
    copy_memory_buffer.destination = destination;
    copy_memory_buffer.length = size;

    DWORD bytes_returned = 0;
    return DeviceIoControl(device_handle, ioctl1, &copy_memory_buffer, sizeof(copy_memory_buffer), nullp
```

Kdmapper עושה שימוש בפעולה הזו כמו API שסופק לו מראש, כלומר אם אני רוצה לממש קריאה עם ה-driver שלי אני רק אצטרך לשנות את הפעולה שתשלח את הפרמטרים בפורמט ההכרחי עבור ה-driver ולוודא שה-HANDLE הוא ל-driver שלי.

הבעיה המרכזית לא נובעת מעצם טעינת התוכנה או תיקון התכונות של התוכנה (imports, relocations, stack cookie...) כי הדברים האלה ניתנים לביצוע עם הוספה של מעט פונקציונליות נוספת להשמשה שלי.

הבעייה המרכזית היא כל התהליך של החבאת ה-service של ה-driver החולשתי (לא חובה כמובן, אך עדיין עדיף להיות כמה שיותר חבויים) וקריאה לפעולת ה-DriverEntry של ה-unsigned driver.

כפי שתיארתי במאמר הקודם, המימוש של kdmapper להרצת פעולת הכניסה בדומה ל-unsigned mappers דומים נראה כך:

1) להשיג את הכתובת הוירטואלית שלה מ-usermode בעזרת הפעולה GetProcAddress.
2) ביצוע trampoline hook לפעולה בעזרת הפרימיטיב כתיבה שה-driver החולשתי מביא לנו.
3) קריאה לפעולה המתאימה ב-UM, ככה שה-DriverEntry שלנו יקרא.

הבעייה המרכזית פה היא שפרימיטיב הכתיבה שלנו הוא מרוכז בכתובות פיזיות, ואין לנו דרך ממשית להמיר כתובת מערכת וירטואלית לכתובת פיזית. הדרך המוכרת לעשות את ההמרה הזאת היא בעזרת הפעולה הקרנלית MmGetPhysicalAddress, אך הבעייה הברורה היא שאין לנו אפשרות להריץ את הפעולה הזאת מקוד שנמצא ב-UM ואין דרך ידועה לעשות את אותה המרה מתוכנת UM בדרך אחרת.

אז לסיכום...

כמו כל תהליך של הנדסה לאחור, גם ב-RE של Divers יש מתודולוגיה ותהליך מחשבה שכדאי לפעול לפיו כדי להבין מה התוכנה עושה, איך היא פועלת ובסופו של דבר לנסות להבין אם היא חולשתית ואיך אפשר להשמיש חולשה בו.

התהליך הזה יכול להיות מסובך, במיוחד כשאין את ה-Symbols של ה-Driver וככה כל התוכנה נראית כמו חרבוש גדול עם Offsets רנדומליים שלא אומרים כלום. למרות זאת, לצערנו רוב החולשות לא מבוססות על טכנולוגיות מסובכות או מניפולציה עמוקה של פרמטרים כדי להשיג את ההשמשה, אלא על שימוש לא בטוח \ נכון בפעולות \ פקודות קריטיות שיכולות להביא לתוקף שליטה משמעותית מאוד על המערכת.

ביבליוגרפיה

- Kernel Windows ממבט התקפי - קישור למאמר שלי בגיליון 162 על פיתוח / מחקר קרנלי כללי בקרנל של ווינדוס, מבני נתונים חשובים וערכים שחשוב לזכור.
- מעקף DSE בעזרת שיטת BYOVD - קישור למאמר שלי בגיליון 163 על vulnerable drivers, עקרונות ב-drivers והשתמשות בהם.
- CVE-2023-20598 - קישור להשמשה של driver חולשתי.
- https://youtu.be/BvYN_TcAZfU - קישור להדגמת ההשמשה.

על המחבר

שמי שי גילת, בן 18, מתעניין מאוד בתחומי הפיתוח ומחקר בסביבת lowlevel, מערכות הפעלה ואבטחת מידע. מעוניין מאוד לפתח את הידע שלי וללמוד עוד כדי להתפתח בתחום.

בין הפרויקטים המרכזיים שלי עבדתי על 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.