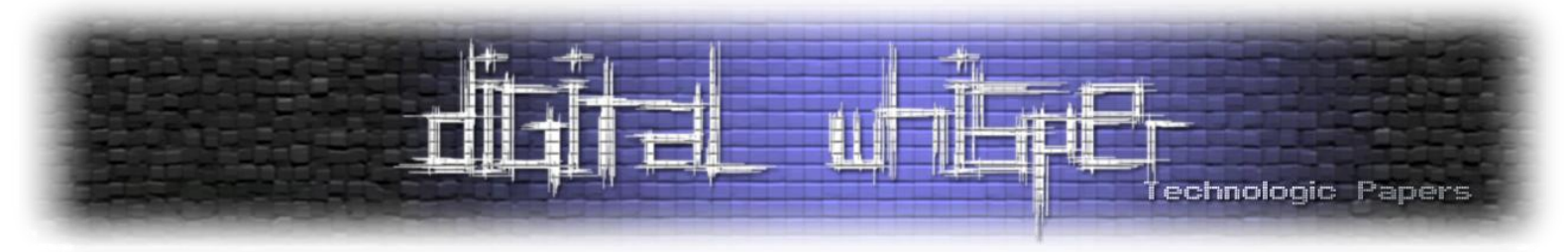


ניצול ה-Kernel על מנת לבצע Code Execution

מאת יואב עומר

הקדמה

בעבר, לפני שהתחילו לחשוב על הגנה בסייבר, כדי לגרום לתהליך מרוחק להריץ קוד שלנו, יכולנו פשוט להקצות מרחב זיכרון של תהליך מרוחק, לכתוב אליו את המידע הזדוני שאנחנו רוצים, להריץ וליצור thread חדש בתהליך המרוחק, שיריץ את הקוד שכתבנו לזיכרון שלו. לצערנו, כל EDR שמכבד את עצמו אינו מאפשר שלושת השלבים העוקבים.

לכן, טובי בנינו ובנותינו שוקדים במרץ על מציאת דרכים חדשות להשיג יכולת הרצת קוד על תהליך מרוחק, ללא השימוש בפונקציה (או באחת מפונקציות המימוש שלה) CreateRemoteThread.

היכולת ליצור תהליך מרוחק ללא שימוש ב-CreateRemoteThread או מקבילותיו, הופכת את מלאכת הזיהוי של מנגנוני הגנה כמו EDR-ים למורכבת הרבה יותר.

מאמר זה עוסק בטכניקה שחקרתי לביצוע הרצה של קוד בתהליך מרוחק. אופן הפעולה של דרך זו מתאפשר באמצעות ניצול ה-kernel על מנת להריץ קוד בתהליך מרוחק. ביצוע שינוי בהתנהגות של אחת הפונקציות שה-Kernel מפעיל בתוך תהליכי GUI מרוחקים, כך שכאשר יגיע הזמן, אחד התהליכים המרוחקים יריץ את קוד זדוני מבלי לדעת שהוא עושה זאת.

זאת טכניקת ACE (Arbitrary code execution) מיוחדת מאוד, אנחנו ניגע בהמון נושאים מעניינים ומטורפים שיעזרו לנו להבין איך עובדת מערכת ההפעלה של windows. אתחיל ברקע תאורטי.

אני ממליץ לקרוא את המאמר עם הבנה בסיסית לפחות בנושאים הבאים: windows api, user mode kernel mode, הבנת המושג הזרקה וכן ACE.

אני כתבתי את ה-POC ב-rust, אבל אני לא אוסיף קישור אליו. אני מאמין שהדרך הכי טובה ללמוד היא לממש בעצמנו. אסביר לפרטי פרטים מה צריך לממש, אני כן אוסיף את ה-struct-ים ה-undocumented של Windows, אני יודע כמה קשה יכול להיות למצוא אותם.

היסטוריה של הטכניקה

הטכניקה הממומשת במאמר הינה "kernel callback table", אתחיל מהסוף ואסביר כיצד היא אפשרית.

בעבר, כל הפונקציונליות של ה-GUI מומשה על ידי Microsoft ב-user mode, בתהליך שנקרא csrss.exe. התהליך אחראי לתקשר עם הדרייברים של ה-I/O, בעקבות כך נוצר מצב שמערכת ההפעלה הייתה צריכה לעבור בין המצבים user mode ו-kernel mode מספר רב של פעמים. דבר זה יצר עלות ביצועית גבוהה וצרך זיכרון בכמות רבה, ולכן הוחלט להעביר את הפונקציונליות של ה-GUI אל ה-kernel mode.

Microsoft כתבו דרייבר אשר אחראי על הפונקציונליות הזאת ב-kernel שנקרא Win32k.sys. לא היה ניתן להעביר את כל הפונקציונליות ל-kernel, בסופו של דבר תהליכים עם GUI חייבים לדבר עם המשתמש. לכן ישנה פונקציונליות שעדיין נשארה ב-user mode (עדיין בתהליך csrss.exe).

ישנן שתי ישויות, אחת פועלת ב-User Mode, והשנייה ב-Kernel Mode, אשר נדרשות לתקשר ביניהן. כאשר הדרייבר win32k.sys, שפועל ב-Kernel Mode, נזקק לשירות מתהליך הפועל ב-User Mode, הוא עושה זאת באמצעות קריאה לפונקציה KeUserModeCallback. פונקציה זו גורמת לתהליך שב-User Mode לבצע קריאה לפונקציה מתוך ה-Kernel Callback Table שלו, ובכך מתאפשרת העברת שליטה זמנית חזרה ל- User Mode לצורך ביצוע פעולה נדרשת.

ה-kernel callback table זאת טבלה אשר מכילה המון pointer-ים לפונקציות.

כאשר תהליך טוען את ה-DLL user32.dll (הדרייבר שאחראי על ה-GUI של תהליכים), נטען אל ה-PEB (process envioment block) מצביע אל ה-kernel callback table. לפני שנטען ה-DLL ב-PEB לא מופיע ה-kernel callback table, לכן חשוב לוודא כי אכן התהליך טען את ה-DLL (אחת הדרכים לבדוק אם ה-DLL נטען היא לבדוק אם לתהליך יש חלון).

מפתח מושגים

- Shellcode - קוד זדוני הכתוב בתהליך המרוחק.
- KCB - kernel callback table.
- Process Environment Block - PEB

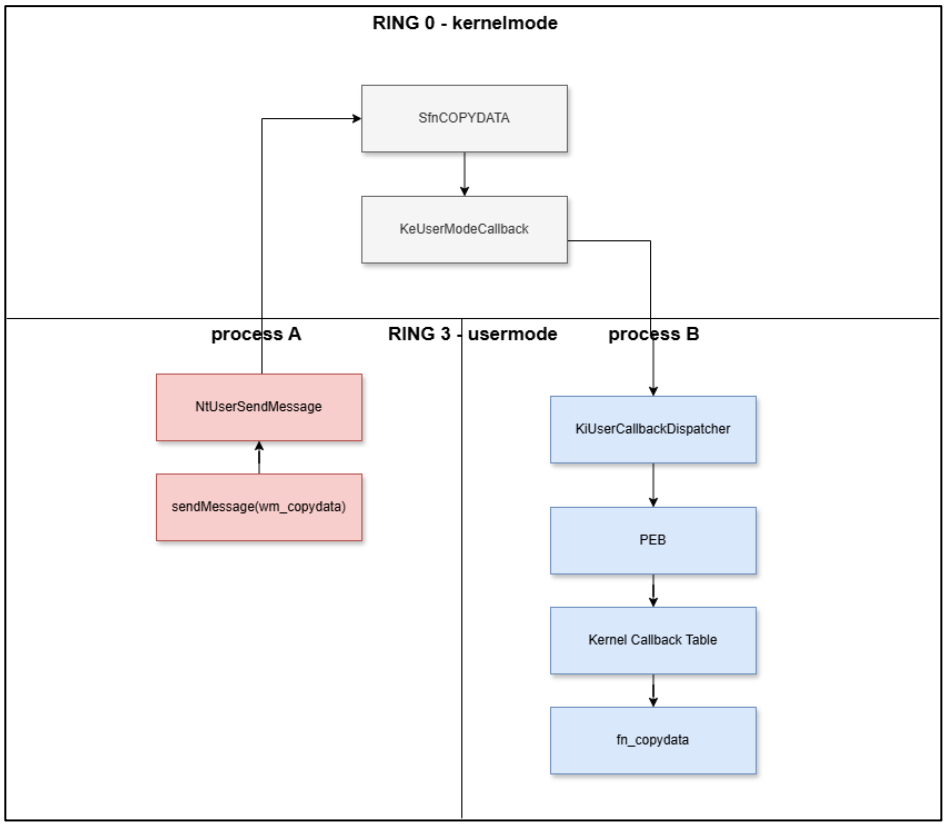
הסבר תאורטי

הטכניקה המתוארת מבוססת על ניצול של אחת מהפונקציות המוגדרות ב-kernel callback table, הפונקציה __fnCOPYDATA.

פונקציה זו מופעלת כחלק ממנגנון תקשורת בין-תהליכית (Inter-Process-Communication או IPC) בשם WM_COPYDATA. מדובר באחת השיטות הוותיקות להעברת מידע בין תהליכים ב-Windows. כאשר תהליך שולח הודעת WM_COPYDATA לחלון של תהליך אחר, ה-thread המטפל בחלון בתהליך היעד עובר לטפל בהודעה על ידי מעבר לפונקציה __fnCOPYDATA, אשר מוגדרת בטבלת ה-KCB של התהליך.

עובדה זו מאפשרת לנצל את המנגנון הקיים כדי להריץ קוד תחת ההקשר של תהליך אחר. מאחר שהכתובת של __fnCOPYDATA ניתנת לשליטה (באמצעות שינוי הערך המתאים ב-KCB), ניתן להפנות אותה ל-shellcode משלנו, ובכך לגרום לתהליך היעד להריץ את הקוד הרצוי כשיטפל בהודעת WM_COPYDATA.

אבסטרקציה, כיצד הודעה עוברת בין תהליכים:



לסקרנים, ממליץ לעצור כאן לפני שמתקדמים בקריאה ולחשוב על השאלה הבאה, כיצד באמצעות המידע שסיפקתי ניתן לגרום לתהליך מרוחק להריץ קוד זדוני?

פתרון:

כדי לגרום לתהליך המרוחק להריץ Shellcode בכל פעם שהוא מקבל הודעת WM_COPYDATA, צריך לשנות את כתובת המצביע של fnCOPYDATA__ כך שתצביע לכתובת הרצויה. אולם, הטבלה עצמה קריאה בלבד (read-only), ולכן לא ניתנת לשכתוב ישיר.

קיימת דרך עוקפת, במקום לנסות לשנות את הטבלה עצמה, ניתן ליצור עותק זהה שלה בזיכרון, ולערוך בו את השדה המתאים כך שיצביע לכתוב הרצויה.

לאחר מכן, יש לעדכן את המצביע לטבלה בתוך מבנה ה-PEB של תהליך היעד, כך שתצביע לעותק החדש.

בתרחיש זה, כאשר תהליך היעד מקבל הודעת WM_COPYDATA, הקריאה עוברת דרך ההפניה מתוך ה-PEB, שמובילה כעת אל העותק המזויף של ה-KCB. הפונקציה המוגדרת ב-fnCOPYDATA__ מתבצעת, ובכך מושגת הרצת קוד תחת ההקשר של התהליך המרוחק - ללא שימוש גלוי במנגנוני יצירת thread חיצוניים.

Native API

ה-Native API הוא מושג מרכזי בתכנות למערכת ההפעלה Windows , ומייצג את רמת הגישה הקרובה ביותר שמוצעת למתכנתי user mode לצורך אינטראקציה עם ה-kernel.

כאשר תהליך במצב user mode זקוק לגישה למשאב מערכת (למשל קובץ, זיכרון או אובייקט סינכרון) מתרחש מעבר אל kernel mode, באמצעות קריאת מערכת (syscall).

עם זאת Windows API, הממשק הרשמי והמתועד של מערכת ההפעלה, אינו מבצע בצורה ישירה את קריאת המערכת, אלא מפעיל פונקציית ביניים מתוך ב-ntdll.dll או (לעיתים win32u.dll). היא זו שאחראית על ביצוע ה-syscall בפועל.

ה-Native API עצמו אינו מתועד באופן רשמי על ידי Microsoft, אך תועד לאורך השנים באמצעות מחקר reverse engineering של קבצי ה-DLL הרלוונטיים.

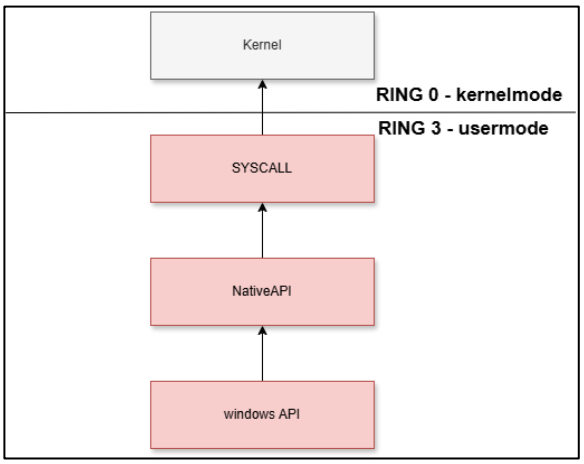
חתימות רבות של הפונקציות הללו זמינות באתר: https://ntdoc.m417z.com, שמשמש מקור מידע אמין ומעודכן עבור מפתחי low-level ב-Windows.

בעת שימוש ב-Native API יש לטעון את הספרייה ntdll.dll ולהשיג ממנה מצביע לפונקציה הרצויה.

פעולה זו מתבצעת לרוב באמצעות שתי פונקציות מה-Win API.

- GetModuleHandleA - מקבלת מחרוזת עם שם הספרייה (למשל "ntdll.dll") ומחזירה מצביע לספרייה.
- GetProcAddress - מקבלת מצביע לספרייה ומחרוזת שם של פונקציה, ומחזירה מצביע לזיכרון בה היא נמצאת.

שימוש ב-Native API, מאפשר לבצע פעולות שאינן נגישות דרך ה-Win API הרגיל, אך הוא מחייב זהירות והבנה מעמיקה של מבני המערכת והקריאות הפנימיות שלה.



חלונות

כפי שמקובל במערכת ההפעלה Windows, כמעט לכל אובייקט ויזואלי – ולעיתים גם לאובייקטים שאינם מוצגים למשתמש – נוצר חלון (Window Handle).

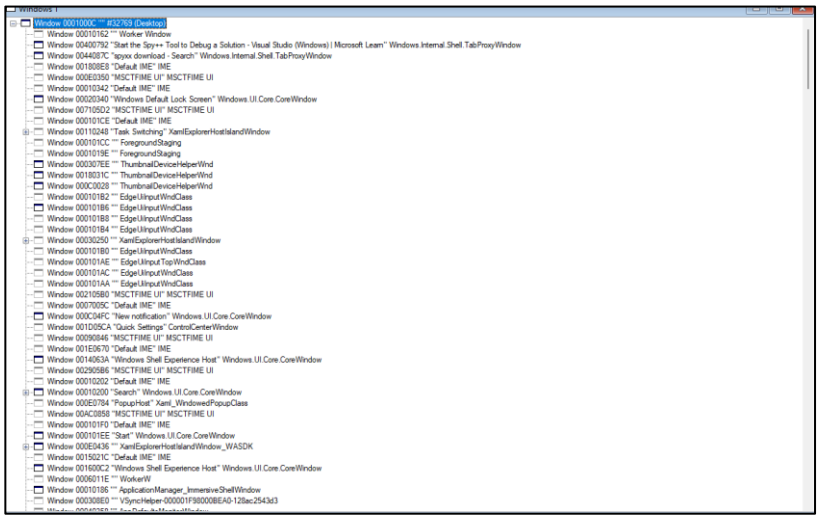
במקרה של הטכניקה המתוארת, קיומו של חלון בתהליך היעד הוא תנאי הכרחי:

כפי שהוסבר בתחילת המאמר, טעינת user32.dll אל תוך התהליך מובילה ליצירת טבלת ה- Kernel Callback Table בתוך ה-PEB. פעולה זו מתבצעת רק כאשר התהליך טוען את הספרייה בפועל, דבר שקורה כאשר קיימת בו אינטראקציה גרפית, כלומר חלון פעיל.

יתרה מזאת, מאחר והטכניקה עושה שימוש בהעברת הודעתWM_COPYDATA , אשר נשלחת לחלון קיים, יש לוודא שלתהליך היעד אכן יש חלון פעיל שניתן לאתר ולשלוח אליו את ההודעה.

חשוב לזכור שכל חלון במערכת משויך ל-thread מסוים בתהליך, וזהו ה-thread שיטפל בפועל בהודעה.

כפי שניתן לראות בתמונה המצורפת מתוך (spyxx.exe) Spy++, כלי של מיקרוסופט, קיימים אינספור חלונות פעילים במערכת. לכל אחד מהם משויך מזהה ייחודי (Handle) ומידע נוסף כמו שם המחלקה, הכותרת ו-thread האחראי על הטיפול בו.



בכדי לבצע את ה-ACE באמצעות WM_COPYDATA, נדרש Handle של חלון מתוך התהליך שאליו רוצים להזריק. אך לרשותנו נמצא רק ה-PID של התהליך, ואין API שמחזיר ישירות את כל החלונות של תהליך מסוים.

כאן נכנסת לתמונה הפונקציה EnumWindows מ-WinAPI. פונקציה זו מקבלת שני פרמטרים:

- מצביע לפונקציית callback אשר תופעל עבור כל חלון במערכת.
- ערך LPARAM כללי שבו ניתן להעביר את ה-PID של תהליך היעד.

במהלך הסריקה, הפונקציה מפעילה את ה-callback על כל חלון במערכת. בתוך ה-callback נשתמש ב-GetWindowThreadProcessId, שמקבלת Handle של חלון ומחזירה את ה-PID של התהליך שמחזיק בו.

כאשר נמצא חלון שה-PID שלו תואם לתהליך היעד, ניתן להחזיר FALSE כדי לעצור את הסריקה ולשמור את ה-HANDLE של אותו חלון.

כך נוכל לאתר חלון של תהליך יעד בעזרת PID בלבד, ולשלוח אליו את הודעת WM_COPYDATA שתפעיל את הקוד שלנו.

shellcode של createRemoteThread (אופציונלי)

בעת שליחת ההודעה WM_COPYDATA לחלון מסוים, ה-thread שאחראי על ניהול אותו חלון עובר לטפל בהודעה. במסגרת הטיפול קורא ה-thread לפונקציה fnCOPYDATA__ , והקוד שמוגדר בה רץ תחת ההקשר של אותו ה-thread, שלעיתים קרובות מהווה חלק קריטי במנגנון העבודה של התהליך.

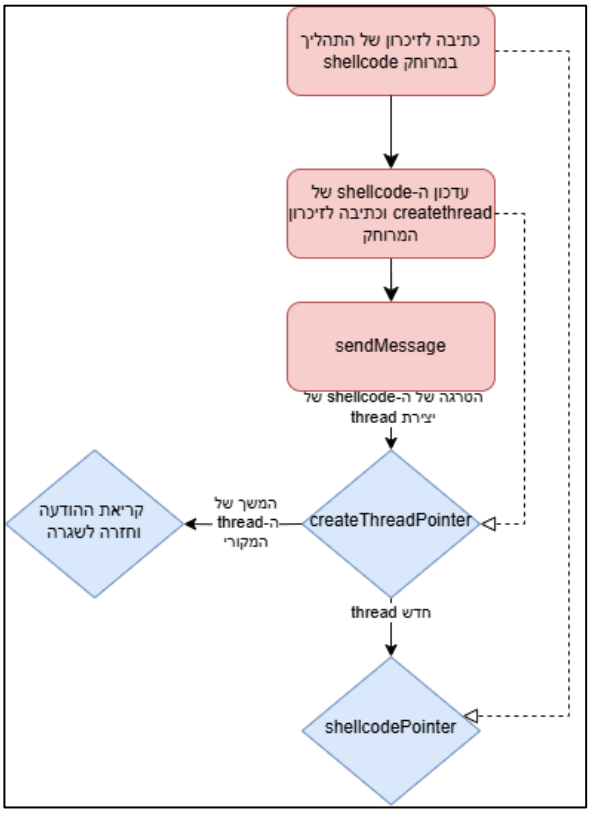
לאור זאת קיימת חשיבות גבוהה לכך שהקוד המתבצע בשלב זה יהיה קצר ככל הניתן, כדי למנוע השהיה או חסימה של thread מרכזי.

עם זאת, איננו מעוניינים להגביל את משך הריצה של ה-shellcode עצמו. לכן יש להעביר את הביצוע שלו ל-thread נפרד או למנגנון אסינכרוני, כך שה-thread המקורי יוכל לחזור לפעולתו התקינה במהירות.

דרך פעולה אפשרית לעקיפת מגבלה זו, היא כתיבה של shellcode קצר עבור ארכיטקטורת x64 (הנפוצה ברוב מחשבי Windows כיום), אשר תכלול קריאה לפונקציה CreateThread.

ה-thread החדש שייווצר יתחיל ריצתו בכתובת ה-shellcode שהוזן קודם לכן לזיכרון התהליך המרוחק.

לבסוף, הפונקציה fnCOPYDATA__ תכוון לכתובת של אותו shellcode קצר, כך שטיפול בהודעת WM_COPYDATA יפעיל אותו, ובאופן עקיף גם את ה-shellcode המלא, מבלי לחסום thread מרכזי של התהליך.



לא אכנס לעומק כתיבת ה-shellcode מאחר שזה נושא שמצדיק מאמר בפני עצמו. עם זאת, לצורך הדגמה, מצורפת גרסה של shellcode קצר שנכתב במיוחד עבור הפונקציה fnCOPYDATA__.

חשוב לציין שה-shellcode אינו כללי, הוא מותאם למבנה הספציפי של הקריאה מתוך fnCOPYDATA, ולא בהכרח יתאים לפונקציות אחרות.

בשל מנגנון ה-ASLR (Address Space Layout Randomization), לא ניתן לקבוע מראש את הכתובות המדויקות שישמשו ב-shellcode, ולכן הוא כולל כתובות פיקטיביות (placeholders) שיש לעדכן במהלך הריצה (runtime patching).

לנוחות הקוראים, הקוד מלווה בתיעוד מלא המסביר את מבנה ההרצה והפרמטרים הקריטיים.

מומלץ לעיין בשני הקישורים שצורפו - הם מספקים רקע חשוב על קריאות מערכת בארכיטקטורת x64 ויכולים להקל משמעותית על תהליך ההבנה והפיתוח:

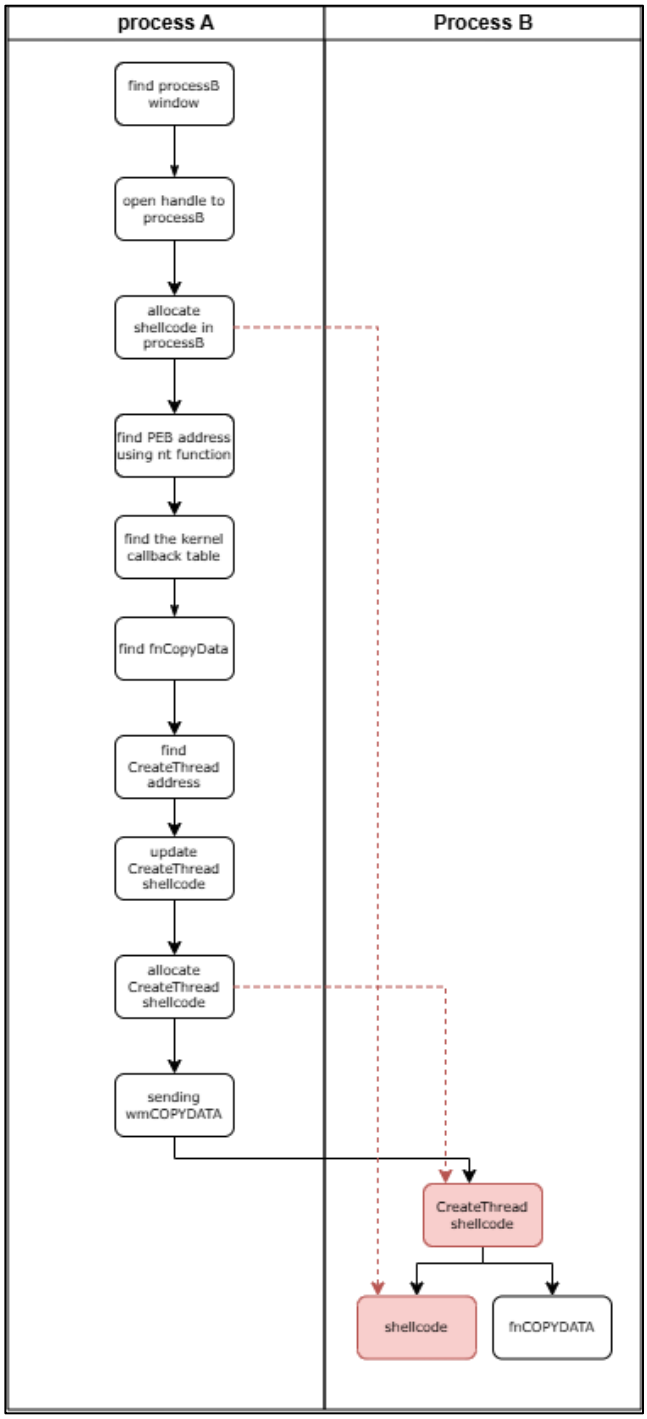
https://learn.microsoft.com/en-us/cpp/build/x64-software-conventions?view=msvc-170

https://github.com/simon-whitehead/assembly-fun/blob/master/windows-x64/README.md

ההזרקה

אם הגעתם עד לכאן, כנראה שעברתם את החלק התיאורטי המורכב, וכעת כל שנותר הוא לממש את התהליך כולו.

בעמוד הבא תמצאו תרשים מסכם שמציג בצורה ויזואלית ומדויקת את רצף השלבים הנדרשים לצורך ביצוע ההזרקה, משלב ההכנה ועד להפעלת הקוד בתהליך היעד.



Struct-ים נדרשים

על מנת לממש את הטכניקה בפועל, יש להכיר את מבני הנתונים (structs) הפנימיים של Windows, כפי שהם מיוצגים בזיכרון. קיימים מספר מקורות איכותיים המפרסמים מידע על מבנים בלתי מתועדים (undocumented). האתר ntdoc שהוזכר קודם לכן, וכן אתר vergilius שמתמקד בעיקר במבני Kernel .

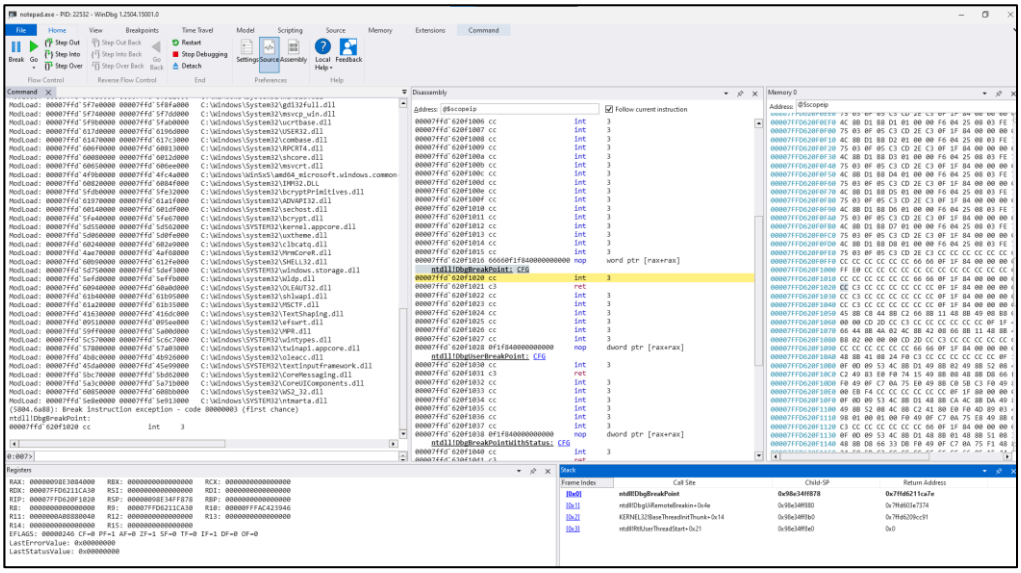
עם זאת, יש לציין כי מבנה ה-Kernel Callback Table (KCB), שהוא קריטי עבור הרצת הקוד, לא מופיע באף אחד מהמקורות הקיימים.

לכן, לצורך כתיבת הקוד, סופקו כאן הגדרות מבנים מותאמות, שנבנו על סמך reverse ממוקד לצורך ה-POC. חשוב להדגיש כי המבנים - במיוחד כפי שהם מוגדרים ב-RUST, אינם מדויקים לחלוטין, מאחר שחלק מהשדות הם מצביעים למבנים פנימיים, אשר אינם קיימים כחלק מהחבילות בהן נעשה שימוש.

למטרות הטכניקה הנוכחית, די בהבנה של ה-offset-ים המרכזיים, אך אין להתייחס למבנים כאל תיעוד רשמי או מלא של מבנה הזיכרון במערכת. שימוש בהם לצרכים אחרים מחייב אימות נוסף.

כלי שיציל לכם את הדיבוג

כשהתחלתי לכתוב Inject-ים, אחד האתגרים הכי גדולים שנתקלתי בהם היה דיבוג. לא הכרתי את הכלים שיכלו לחסוך לי שעות של תסכול - במיוחד כשמדובר בשגיאות "שקטות", שה-process פשוט קורס בלי שום רמז. רק אחרי הרבה ניסוי וטעיה גיליתי כמה כלים שיכולים להפוך את החיים לקלים בהרבה. בראש הרשימה: **WinDbg.**



WinDbg הוא Debugger עוצמתי מבית Microsoft, המאפשר לנו לעקוב אחרי הרצת הקוד בזמן אמת, לעצור בכל נקודה בזיכרון ולבחון את תוכן הרשומות, הזיכרון וה-disassembly של הקוד. הכלי שימושי במיוחד בשלבי כתיבת והזרקת shellcode, בהם כל instruction חשוב.

פקודות בסיסיות שכדאי להכיר:

- bp <address> קביעת Breakpoint בכתובת הרצויה.
- g - go, המשך ריצה.
- t - step into, ביצוע שורת קוד אחת, כולל כניסה לפונקציות.
- p - step on, דילוג על קריאות לפונקציה.
- gu - step out, יציאה מהפונקציה הנוכחית.

כמובן שניתן לבצע את כל הפעולות הללו (ועוד רבות אחרות) גם באמצעות ה-GUI של WinDbg כולל ניווט בין Threads, צפייה ב-stack ועוד. למי שכותב Inject-ים, מתעסק ב-reverse engineering, או מפתח low-level מול מערכת ההפעלה - הכלי הזה הוא בגדר חובה.

סיכום

במאמר זה הוצגה טכניקה המאפשרת ACE, המבוססת על ניצול ה-Kernel Callback Table לצורך הרצת קוד בתוך תהליך GUI מרוחק, באמצעות מנגנון ההודעות של Windows והפונקציה __fnCOPYDATA .

הגישה מתאפיינת בכך שהיא עוקפת במלואה את הצורך ביצירת thread באופן ישיר, פעולה המהווה "אנטי־פטרן" מובהק לזיהוי במרבית מוצרי EDR מודרניים. הקוד, המבנים, והשרטוטים זמינים ב-GitHub: https://github.com/yoavomer/kernel-callback-injection-structs-shellcode

אני מקווה שאהבתם את המאמר, ונהנתם לכתוב את ה-POC. אני חייב להגיד שזה היה אחד מה-injection-ים שהכי נהניתי לכתוב, הוא פשוט ומלמד הרבה. אם יש לכם שאלות, תובנות נוספות, או רצון לשתף טכניקות דומות אני תמיד שמח לשמוע.

אני כותב ב-rust וב-cpp (שלא תטעו אני מעריץ מושבע של rust), ובגלל זה צורפו ה-struct-ים בשתי השפות.

בנוסף מוזמנים להתחבר אלי ב-linkedin.

מקורות מידע

- https://github.com/simon-whitehead/assembly-fun/blob/master/windows-x64/README.md
- https://unit42.paloaltonetworks.com/win32k-analysis-part-1/
- https://media.blackhat.com/bh-us-11/Mandt/BH_US_11_Mandt_win32k_WP.pdf
- https://github.com/0xHossam/KernelCallbackTable-Injection-PoC
- https://www.crow.rip/crows-nest/mal/dev/inject/ntapi-injection/complete-ntapi-implementation
- https://www.linkedin.com/in/tom-sha/