Digital Whisper Technologic Papers

# Packing and Unpacking in Malicious Files

מאת עומר כחלון

## הקדמה

אחת המטרות של תוקף כאשר הוא כותב פוגען (מלבד כמובן ביצוע פעולה זדונית כגון גניבת סיסמאות, הדלפת מידע, הצפנת קבצים וכדומה) היא להקשות על תהליך החקירה של הפוגען, דבר המתבטא בעיקר בהארכת זמן החקירה בצורה משמעותית, ואף מניעת יכולת החקירה כלל.

על מנת להקשות על תהליך החקירה, קיימות מספר טכניקות מוכרת כגון Anti-Debugging או Anti-VM (על טכניקות אלו לא אפרט במאמר זה, עם זאת יש המון מאמרים הסוקרים את הטכניקות הנ"ל. כמו גם טכניקות נוספות, אצרף קישורים בסוף המאמר). במאמר זה אתמקד בטכניקה ספציפית הנקראת Packing.

כל המידע המוצג במאמר זה לקוח ממקורות שונים אליהם כמובן אצרף קישור. מטרת המאמר היא להציג את טכניקת ה-Packing ולספק טכניקות להתמודדות עם Packers שונים. כפי שאני מניחה שכבר הבנתם, המאמר יתמקד ב-Packing בהקשר של חקירת פוגענים. כדי להבין לעומק את המאמר דרוש ניסיון וידע בסיסי בחקירת פוגענים, והכרת מושגים כמו: חקירה סטטית, חקירה דינאמית, וכמו כן מעט הכרות עם מבנה קובץ PE ו-Debuggers. במאמר לא אספק הסבר נרחב למונחים אלו, עם זאת אצרף קישורים בסוף המאמר שתוכלו להיעזר בהם.

## אז מה זה Packer?

כשמדברים על קובץ שהוא Packed מדברים על קובץ שעבר תהליך של כיווץ או הצפנה על ידי שימוש באלגוריתם מסוים, וכתוצאה מכך נוצר קובץ חדש על הדיסק, שהמבנה שלו שונה ממבנה הקובץ המקורי והגודל שלו קטן יותר. תהליך ה-Packing אינו משפיע על פעולת הקובץ המקורי, כלומר אם ניקח קובץ מסויים ונריץ עליו תוכנת Packing, גם לפני וגם אחרי, פעולת הקובץ תהיה זהה, עם זאת שני הקבצים יראו שונה לחלוטין על הדיסק.

### איך זה בא לידי ביטוי?

קובץ שהוא Packed יכיל את המאפיינים הבאים או חלק נכבד מהם:

- מספר פונקציות ו-import-ים מועטים (בדרך כלל יכלול את LoadLibrary ו-GetProcAddress). הסיבה לכך היא שרב ה-Packers בונים את טבלת ה-imports של הקובץ בזמן ריצה, זה גם מקשה על החוקר להבין באיזה פונקציות הקובץ משתמש וגם עוזר מעט להקטין את גודל הקובץ.
- הקובץ יכיל כמות מחרוזות מועטה, יתכן גם שלא נראה מחרוזות הגיוניות כלל.
- הקובץ יכלול שמות section-ים המאפיינים Packers מוכרים.
- אם נפתח את הקובץ ב-IDA יכול מאוד להיות שהיא תתריע לנו שהיא לא מצליחה למצוא את כל ה-imports או שקיימת בעיה עם ה-sections, בנוסף כש-IDA תטען את התוכנית נראה שכמות הקוד שזוהתה היא יחסית קטנה.

בחינה של מאפיינים אלו תעזור לנו לקבל אינדיקציה האם הקובץ הוא Packed או לא. קיימים גם כלים אוטומטיים המסוגלים לזהות האם קובץ הוא Packed ואף להחזיר את שם ה-Packer בו השתמשו אם מדובר ב-Packer מוכר. דוגמה לכלי שכזה הוא peid.

## הדגמה

כדי להבין דברים עד הסוף הכי טוב זה לראות בעיניים, לכן אציג השוואה בין קובץ שהוא Packed לקובץ שהוא לא Packed ונראה בעיניים חלק מההבדלים בין הקבצים.

### כלים שצריך להכיר

בהצגת הדוגמה אשתמש בכלים upx ו-cff. להלן הסבר קצר על שני הכלים הנ"ל.

**upx**- הוא Packer קל לשימוש ו-open source מה שהופך אותו לאחד ה-Packers הנפוצים ביותר כיום.

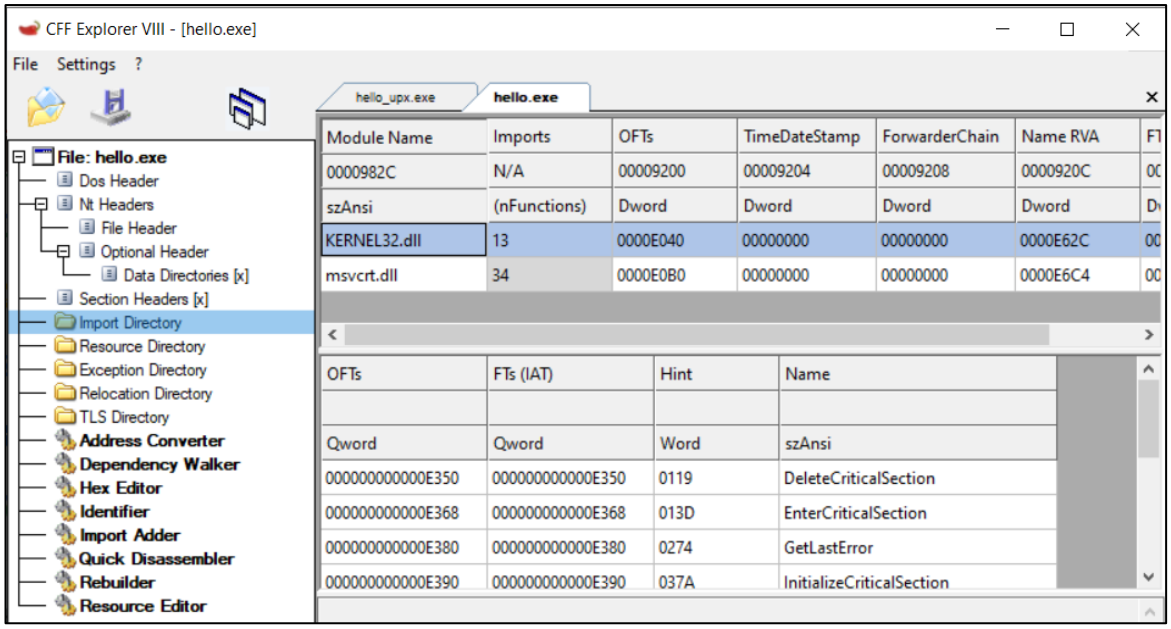
**cff** - היא תוכנה להצגת קבצי PE בצורה קצת יותר מעמיקה. התוכנה מפרסרת חלק ממבנה ה-PE ומציגה את המידע הנשמר בו בצורה קריאה ונוחה יותר למשתמש. לדוגמה, באמצעות התוכנה נוכל לראות את טבלת ה-imports של הקובץ, את ה-sections הקיימים לקובץ ומידע על ה-headers שלהם וכולי וכולי.

אז ניגש לדוגמה, הקובץ שאדגים עליו יהיה קובץ שכתבתי ב-c שכל מטרתו היא להדפיס למסך "Hello World", נקרא לו hello.exe. לאחר מכן לקחתי את הקובץ hello.exe ועשיתי לו Pack באמצעות Packer בשם upx ושמרתי אותו כ-hello_upx.exe.

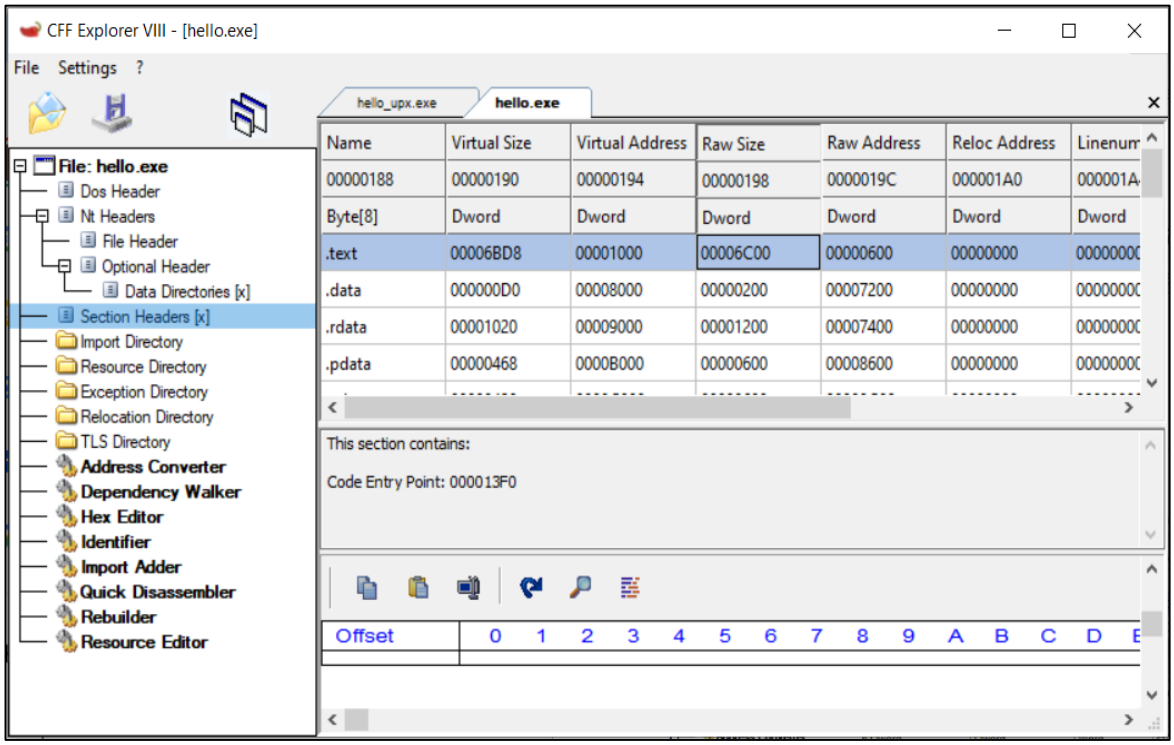
לאחר ששמרתי את שני הקבצים אפתח אותם ב-cff במטרה להסתכל על מבנה הקובץ ולראות אילו שינויים קיימים במבני הקבצים.

**hello.exe**

נתחיל בלהסתכל על ה-imports של hello.exe:



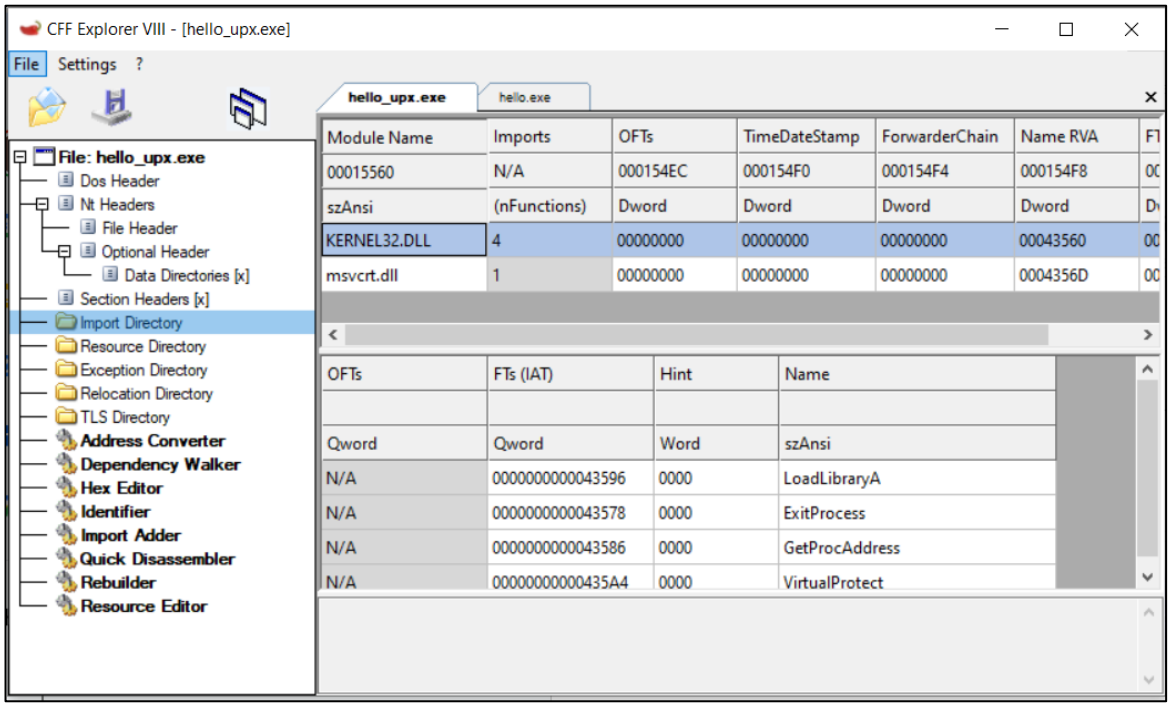

נוכל לראות שיש לקובץ שתי ספריות: kernel32.dll המייבאת 13 פונקציות ו-msvcrt.dll המייבאת 34 פונקציות.



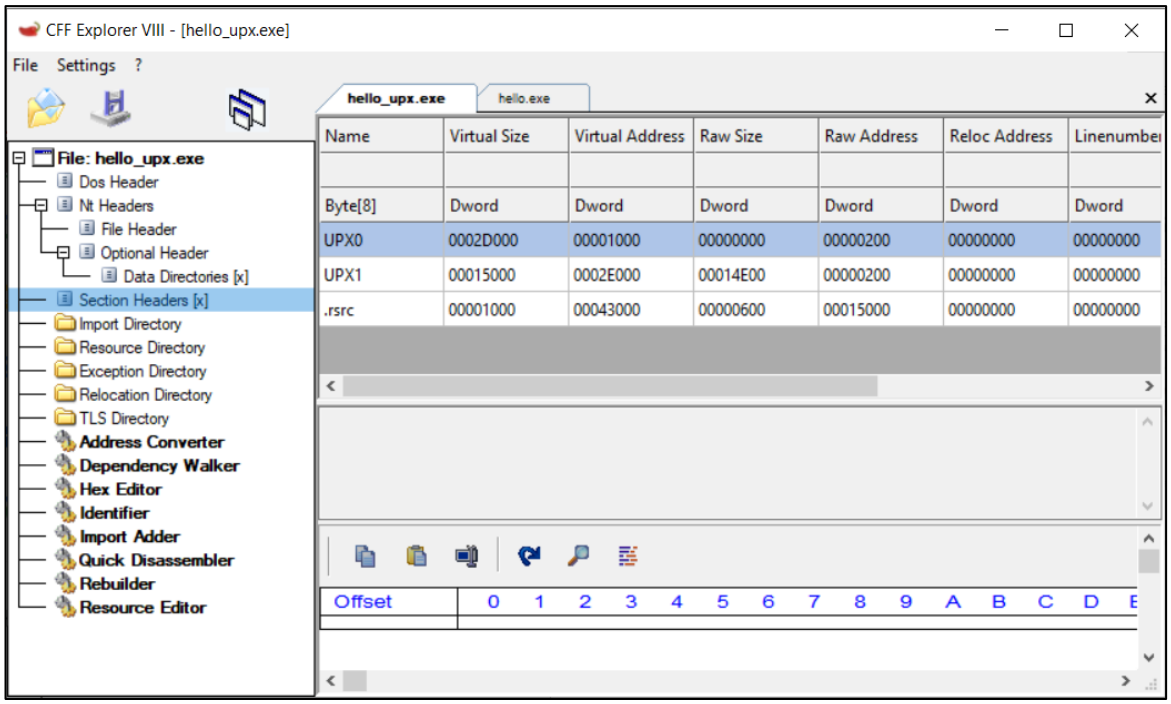

נמשיך ונסתכל על ה-sections של hello.exe, ניתן לראות שלקובץ קיימים sections שונים text ,data ,rdata וכו'.

**hello_upx.exe**

נבחן את אותם המאפיינים גם בקובץ hello_upx.exe. נסתכל על ה-imports של הקובץ:

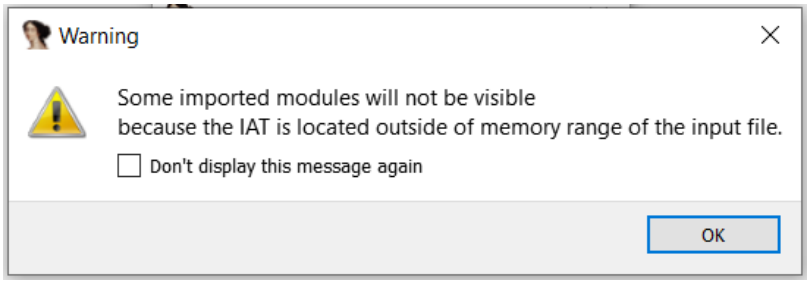



נוכל לראות כי בקובץ ה-Packed קיימות משמעותית פחות פונקציות, סך הכל 5 פונקציות לעומת 47 פונקציות בקובץ המקור. בנוסף ניתן לראות ששתיים מבין הפונקציות המיובאות הן: GetProcAddress ו-LoadLibrary:



נסתכל גם כאן על ה-sections של הקובץ, נוכל לראות שה-sections הסטנדרטים (text, data ו-rdata) לא קיימים בקובץ ה-Packed אך שכן קיימים sections אחרים המזוהים עם upx.

עוד אנקדוטה, אם נסתכל על הגודל של UPX0 נוכל לראות שה-virtual size גדול מה-raw size. ה-raw size זה הגודל ששמור על הדיסק וה-virtual size זה הגודל שצריך להקצות עבור ה-section הנ"ל הזיכרון. הבדל משמעותי בגודל יכול להעיד על כך שיש הרבה מידע שמתקבל בצורה דינאמית. בנוסף, אם נזרוק את הקובץ ל-IDA היא תקפיץ לנו הודעה למסך:




אז ראינו שקיים הבדל מהותי בין הקובץ ה-Packed לקובץ המקורי. אחרי שהבנו את ההשפעות בפועל של תהליך ה-Packing ואיך הן משבשות את תהליך החקירה, בואו נדבר על איך אנחנו יכולים להתמודד עם קובץ שהוא Packed ומה אנחנו יכולים לעשות על מנת להחזיר את הקובץ לצורה שניתן לעבוד איתה.

אני מדברת כמובן על תהליך ה-Unpacking.

## Unpacking Internals

תהליך ה-Unpacking (כפי שאני מאמינה שכבר הבנתם) הוא התהליך ההפוך מ-Packing. ה-Unpacking קורה בזמן ריצת הקובץ ה-Packed כאשר הוא נטען לזיכרון והוא מתבצע על ידי קטע קוד הנקרא Unpacking stub. ה-Unpacking stub בעצם מבצע את האלגוריתם הנדרש על מנת למפות את הקובץ המקורי לזיכרון כך שהתוכנה תוכל לרוץ כרגיל והנתונים שלה (מחרוזות, משתנים, imports וכדומה) יהיו נגישים לשימוש.

כשאנחנו מסתכלים על הקוד של קובץ שהוא Packed אנחנו בעצם מסתכלים על ה-Unpacking stub ולא על הקוד של הקובץ המקורי. נוכל לחקור את הקוד המקורי רק לאחר שתהליך ה-Unpacking הסתיים והקוד חולץ במלואו לזיכרון, כלומר אחרי ריצת ה-Unpacking stub. ה-Unpacking stub עובד בשלושה שלבים מרכזיים:

- טעינת קובץ המקור לזיכרון.
- סידור ובניה מחדש של טבלת ה-Imports.
- העברת שליטה ל-entry point של התוכנית המקורית.

### טעינת קובץ המקור לזיכרון

תחילה ה-Unpacking stub מבצע את האלגוריתם הדרוש על מנת לחלץ את הקובץ המקורי לזיכרון. האלגוריתם כמובן משתנה בין Packer ל-Packer אבל בגדול ה-Unpacking stub יקצה זיכרון ויכתוב לתוכו את התוכן המפוענח, הלא הוא קובץ המקור.

### בניית טבלת ה-imports

בתוך המבנה של קובץ ה-PE נמצאת גם ה-import address table (או בקיצור ה-IAT). IAT היא טבלה המציינת באילו פונקציות קובץ משתמש ומאיזה ספריה כל פונקציה מיובאת. בעת הטעינה של קובץ לזיכרון ה-loader יודע לפרסר את הטבלה הזאת מקובץ ה-PE, לטעון את כל הפונקציות שרשומות בטבלה ולמפות עבור כל פונקציה את הכתובת שלה בזיכרון. אבל מה קורה כשאנחנו מדברים על קובץ Packed?

ניחשתם נכון, המידע שבטבלת ה-IAT של הקובץ המקורי לא נגיש ל-loader מכיוון שהוא Packed. טבלת ה-IAT שנראה עבור קובץ Packed מתייחסת לפונקציות המשמשות את ה-Unpacking stub עבור ביצוע פעולת ה-Unpacking והן כלל לא שייכות לקובץ המקור.

אז אחרי שה-Unpacking stub טען לזיכרון את הקובץ המקורי, עכשיו כן תהיה גישה לטבלת ה-IAT, השלב הבא הוא למלא את הטבלה בכתובות הרלוונטיות עבור כל פונקציה ופונקציה. בדרך כלל השיטה הרווחת היא עבור כל פונקציה, נחזיר את הכתובת שלה בזיכרון באמצעות שימוש בפונקציות LoadLibrary ו-GetProcAddress (זו הסיבה שרב ה-Packers מייבאים את LoadLibrary ו-GetProcAddress וכמעט ולא מייבאים אף פונקציה אחרת, כמעט כל מה שצריך מיובא בצורה דינאמית).

### jump to OEP

OEP משמעותו original entry point. הכוונה היא ל-entry point של הקובץ המקורי (כשהקובץ Packed ה-entry point שלו הוא ל-Unpacking stub). אחרי שהקובץ המקורי נמצא בזיכרון וטבלת ה-Imports מסודרת כל שנותר ל-Unpacking stub הוא להעביר שליטה ל-OEP על מנת שקוד המקור יוכל להתחיל לרוץ. להוראה שמעבירה את השליטה ל-OEP נהוג לקרוא jump tail. ה-jump tail בדרך כלל יהיה jmp לכתובת בזיכרון שבה נמצא ה-OEP, איך באותה מידה זו יכולה להיות הוראה כמו call או ret (ובאופן כללי כל הוראה שמעבירה את ההרצה לכתובת זיכרון מסוימת).

## איך מבצעים Unpacking?

כלים אוטומטיים - אם יש לנו מזל והקובץ שאנחנו חוקרים הוא Packed באמצעות Packer מסחרי מוכר, (יכול להיות שגילינו זאת בעצמנו באמצעות חקירה סטטית או שהשתמשנו בכלי peid והוא זיהה לנו את שם ה-Packer) אז ככל הנראה שקיים ברחבי האינטרנט כלי שיודע לבצע את תהליך ה-Unpacking עבורנו ולחלץ לנו את קובץ המקור, הידד! אממה, אם אני תוקף מתוחכם שרוצה למנוע מחוקר מיומן להבין מה הקובץ שלי עושה, סביר שלא אשתמש ב-Packer מוכר ואינטרנטי עבור פוגען שכתבתי. מה שאני כן יכולה לעשות זה לכתוב תוכנת Packing משלי, עם אלגוריתם שאני המצאתי, כך שלא קיים כלי חיצוני שיכול לבצע Unpacking באופן אוטומטי. מה ניתן לעשות במקרה כזה? זה מוביל אותי אל הנקודה השניה Manual Unpacking.

## Manual Unpacking

נפנה ל-Manual Unpacking כאשר לא הצלחנו למצוא כלי שמחלץ את קובץ המקור בשבילנו. בדרך כלל נתקל במקרה כזה כאשר מדובר ב-Packer לא מוכר באינטרנט (לדוגמה Packer שנכתב על ידי תוקף). ב-Manual Unpacking חילוץ קובץ המקור מתבצע בצורה שהיא ידנית.

יש שתי גישות עיקריות ב-Manual Packing. הראשונה, לחקור את הקובץ ה-Packed, ואת ה-Unpacking stub ולנסות להבין איך עובד האלגוריתם של ה-Packer. אם היינו מספיק חכמים להבין את זה נוכל לכתוב תוכנית המבצעת בדיוק ההפך, כלומר מקבלת קובץ על הדיסק שהוא Packed ויוצרת קובץ חדש על הדיסק שהוא Unpack. חשוב להדגיש כאן שמדובר בתהליך שונה מהתהליך ה-Unpacking המתבצע על ידי ה-Unpacking stub מכיוון שה-Unpacking stub יודע לחלץ ולסדר את מבנה הקובץ בזיכרון ולא על הדיסק.

אממה, אם יש לנו את הקובץ בזיכרון זה אפשרי "לתקן" אותו על מנת להגיע למבנה של קובץ שבסופו של דבר יאפשר לנו לבצע חקירה ב-IDA.

אם כן, נוכל להיעזר ב-Unpacking stub על מנת לקבל גישה לקובץ המקור כאשר הוא ממופה לזיכרון. כל שעלינו לעשות הוא לוודא שה-Unpacking stub חילץ את הקובץ ובנה את ה-imports שלו בהצלחה וננסה למצוא את ה-jump tail רגע לפני הקפיצה ל-OEP.

ובכן לא תמיד זאת הולכת להיות משימה קלה... אציג מספר שיטות למציאת קובץ המקור בזיכרון.

אציג את חלק מהדוגמאות על פוגענים אמיתיים שמצאתי והורדתי מהאתר malshare, שכשמו כן הוא, הוא אתר המכיל מאגר פוגענים להורדה שניתן לחפש לפי מאפיינים שונים כמו: hash, שם קובץ ואפילו חוקי yara.

## Finding the OEP

לפני שאכנס לדוגמאות, אומר שעל מנת להבין את כולן נדרש ידע בסיסי ב-Debugging.

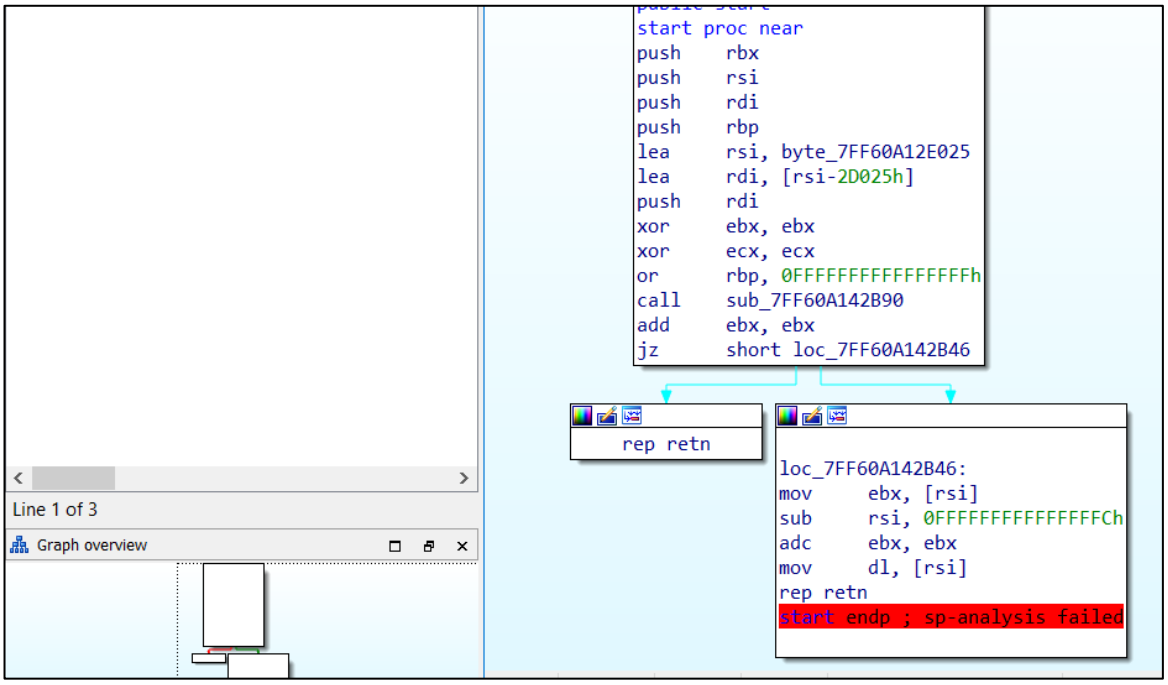
### jump to different section

אחת מדרכי העבודה הנפוצות של Packers היא יצירה של section חדש, ולאחר ה-Unpack לכתוב אליו את התוכן שחולץ (השיטה נקראת גם section override). נוכל לשער שה-jump tail יהיה איזה שהיא הוראה שתעביר את השליטה ל-section אחר שם נמצא את ה-OEP.

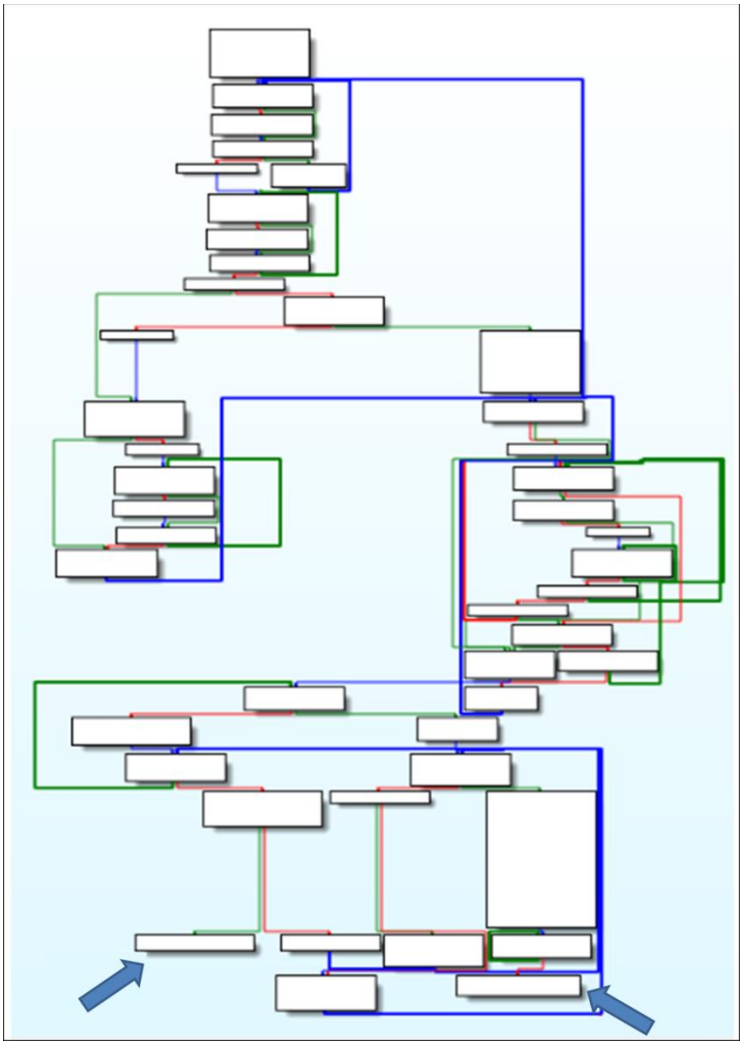
upx משתמש בשיטה הזו, ונדגים את טכניקת החילוץ על הקובץ hello_upx שעבדנו איתו בדוגמה מעלה. הטכניקה תעבוד על כל Packer שפועל בשיטה זו והיא לא ספציפית ל-upx.

### דרך ידנית

נתחיל בלזרוק את הקובץ ב-IDA. המטרה שלנו היא למצוא פקודה שמעבירה את ריצת הקוד ל-section אחר, נוכל לזהות אותה על ידי חיפוש פקודת jmp שקופצת יחסית רחוק (ה-base address שלה יהיה שונה מה-base address של ה-section הנוכחי בו אנו נמצאים). אם מצאנו פקודת jmp שאנחנו חושדים שהיא ה-jump tail שלנו נוכל לבדוק מה תוכן הכתובת אליה ה-jmp מעביר את שליטת הקוד, שם אנחנו לא אמורים לראות מידע כלל (יכול להיות שנראה גם מידע סתמי שיידרס) מכיוון שרק בזמן ריצה יחולץ לשם קוד הקובץ המקורי:



אז נתחיל את החיפושים שלנו בפונקציה ההתחלתית של הקובץ. נראה שקיימת בה רק פקודת jz אחת, אנחנו יודעים שהכתובת אליה היא תקפוץ נמצאת בתוך הפונקציה הנ"ל על פי החיצים ש-IDA מציירת. באופן כללי נראה שלא קורה פה הרבה חוץ מקריאה לפונקציה נוספת, נכנס אליה:



הפונקציה הזאת טיפה יותר ארוכה, אבל אפשר לראות בקלות (שוב בעזרת החיצים של IDA) שאם יש jmp אל מקום שמחוץ לפונקציה זה יהיה בקצוות איפה שאין חץ שמפנה אותנו למיקום בתוך הפונקציה. אם נסתכל מקרוב נראה שבשני המקרים יש קפיצה ל-offsets שתוכנם נקבע בזמן ריצה. נשים breakpoint על כל אחת מהקפיצות האלו:

```
UPX1:00007FF7DECC2CC1
UPX1:00007FF7DECC2CC1 loc_7FF7DECC2CC1:
UPX1:00007FF7DECC2CC1 jmp     large qword ptr cs:7FF7DECC3530h
```

```
UPX1:00007FF7DECC2D77 sub     rsp, 0FFFFFFFFFFFFFF80h
UPX1:00007FF7DECC2D7B jmp     near ptr unk_7FF7DEC813F0
UPX1:00007FF7DECC2D7B sub_7FF7DECC2B90 endp ; sp-analysis failed
UPX1:00007FF7DECC2D7B
```

נריץ את הפוגען ונראה שעצרנו ב-breakpoint השני. ניגש לכתובת של ה-jmp ב-breakpoint הראשון ונראה שיש שם את הכתובת של ExitProcess - מן הסתם לא ה-jump tail שלנו. נמשיך מאיפה שעצרנו ונבדוק עכשיו מה קיים בכתובת אליה אנחנו הולכים לקפוץ.

```
UPX0:00007FF7DEC813F0 ; ---------------------------------
UPX0:00007FF7DEC813F0
UPX0:00007FF7DEC813F0 loc_7FF7DEC813F0:
UPX0:00007FF7DEC813F0 sub     rsp, 28h
UPX0:00007FF7DEC813F4 mov     rax, cs:qword_7FF7DEC89780
UPX0:00007FF7DEC813FB mov     dword ptr [rax], 0
UPX0:00007FF7DEC81401 call    sub_7FF7DEC81180
UPX0:00007FF7DEC81406 nop
UPX0:00007FF7DEC81407 nop
UPX0:00007FF7DEC81408 add     rsp, 28h
UPX0:00007FF7DEC8140C retn
UPX0:00007FF7DEC8140D ; ---------------------------------
UPX0:00007FF7DEC8140D nop     dword ptr [rax]
UPX0:00007FF7DEC81410
UPX0:00007FF7DEC81410 loc_7FF7DEC81410:
```

נשים לב שעכשיו קיים כאן קוד שלא היה קודם לכן! בצד שמאל נוכל לראות את שם ה-section בו אנחנו נמצאים, ואם תשימו לב עכשיו אנחנו ב-section בשם UPX0. לפני הקפיצה (ניתן לראות בתמונה מעלה) היינו בקוד הנמצא ב-section בשם UPX1. לאור הנתונים החדשים שגילינו נוכל לשער בסבירות גבוהה יחסית כי מצאנו את ה-jump tail שלנו.

מכאן השלב הבא הוא לחלץ את הקוד אל הדיסק. כדי לבצע זאת, אעזר ב-x64dbg וב-scylla.

x64dbg -הוא עוד סוג של Debugger ו-Disassembler (ממש כמו IDA) לחקירת קוד בסביבת windows. הכלי מגיע עם ממשק גרפי נוח לעבודה, ומכיל דיפולטית plugin בשם syclla.

scylla - הוא כלי שעוזר מאוד בתהליך ה-Unpacking, ביכולתו לחלץ קובץ מהזיכרון בהינתן ה-OEP (יש לו גם יכולת לחיפוש ה-OEP) והוא יודע לבנות את טבלת ה-IAT על סמך הקובץ שחילץ מהזיכרון.

אז נתחיל בתהליך החילוץ, יהיה עלינו לפתוח את הקובץ ב-x64dbg ולהגיע ל-OEP בדיוק כפי שעשינו ב-IDA. כשנפתח את הקובץ נלחץ F9 וה-Debugger יקח אותנו אל ה-entry point.

חשוב לזכור שלא תמיד הקובץ יטען לאותן כתובות בזיכרון, לכן יתכן מאוד שלא נראה ב-x64dbg את אותן הכתובות שראינו ב-IDA ולא נוכל פשוט לקפוץ לכתוב של ה-jump tail שראינו בדוגמה הקודמת. מה שאפשר לעשות במידה והכתובות שונות בין IDA ל-x64dbg זה לעשות rebase לאחר מה-debugger-ים.

ניתן לבצע rebase עם IDA באמצעות Edit->Segments->Rebase program:

```
hello_upx.exe - PID: 3288 - Module: hello_upx.exe - Thread: Main Thread 23864 - x64dbg
File View Debug Tracing Plugins Favourites Options Help Apr 11 2024 (TitanEngine)
CPU | Log | Notes | Breakpoints | Memory Map | Call Stack | SEH | Script | Symbols
RIP RAX RDX R9
00007FF73EFC2B20   53                  push rbx
00007FF73EFC2B21   56                  push rsi
00007FF73EFC2B22   57                  push rdi
00007FF73EFC2B23   55                  push rbp
00007FF73EFC2B24   48:8D35 FAB4FEFF    lea rsi,qword ptr ds:[7FF73EFAE025]
00007FF73EFC2B2B   48:8DBE DB2FFDFF    lea rdi,qword ptr ds:[rsi-2D025]
00007FF73EFC2B32   57                  push rdi
00007FF73EFC2B33   31DB                xor ebx,ebx
00007FF73EFC2B35   31C9                xor ecx,ecx
00007FF73EFC2B37   48:83CD FF          or rbp,FFFFFFFFFFFFFFFF
00007FF73EFC2B3B   E8 50000000         call hello_upx.7FF73EFC2B90
00007FF73EFC2B40   01DB                add ebx,ebx
00007FF73EFC2B42   74 02               je hello_upx.7FF73EFC2B46
00007FF73EFC2B44   F3:C3               ret
00007FF73EFC2B46   8B1E                mov ebx,dword ptr ds:[rsi]
00007FF73EFC2B48   48:83EE FC          sub rsi,FFFFFFFFFFFFFFFC
00007FF73EFC2B4C   11DB                adc ebx,ebx
00007FF73EFC2B4E   8A16                mov dl,byte ptr ds:[rsi]
00007FF73EFC2B50   F3:C3               ret
00007FF73EFC2B52   48:8D042F           lea rax,qword ptr ds:[rdi+rbp]
00007FF73EFC2B56   83F9 05             cmp ecx,5
```

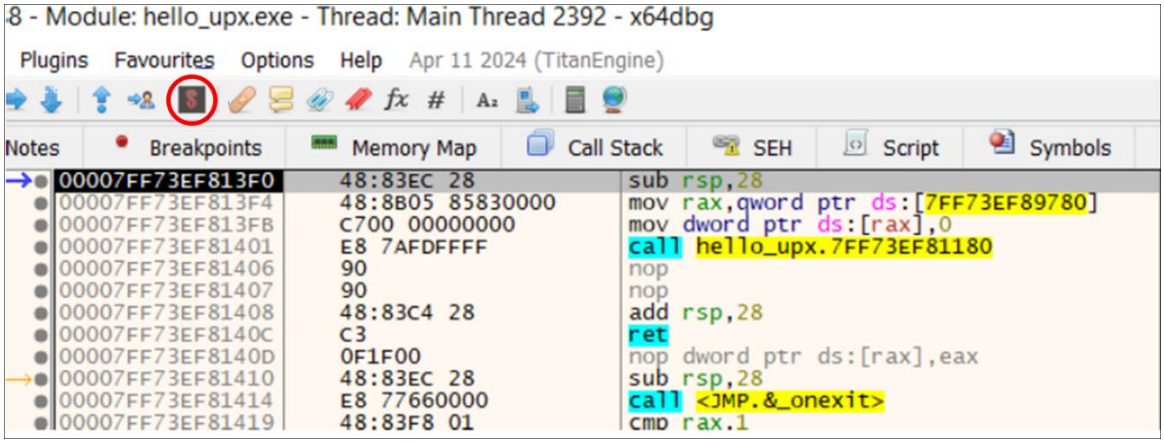
נוכל להיעזר ב-IDA ולראות שהקוד כאן זהה ל-entry point ש-IDA הציגה לנו, לכן נכנס לפונקציה הראשונה כפי שעשינו גם ב-IDA, ונחפש את ה-jump tail:

```
00007FF7DECC2B90   FC                  cld
00007FF7DECC2B91   41:5B               pop r11
00007FF7DECC2B93   EB 08               jmp hello_upx.7FF7DECC2B9D
00007FF7DECC2B95   48:FFC6             inc rsi
00007FF7DECC2B98   8817                mov byte ptr ds:[rdi],dl
00007FF7DECC2B9A   48:FFC7             inc rdi
00007FF7DECC2B9D   8A16                mov dl,byte ptr ds:[rsi]
00007FF7DECC2B9F   01DB                add ebx,ebx
00007FF7DECC2BA1   75 0A               jne hello_upx.7FF7DECC2BAD
00007FF7DECC2BA3   8B1E                mov ebx,dword ptr ds:[rsi]
00007FF7DECC2BA5   48:83EE FC          sub rsi,FFFFFFFFFFFFFFFC
00007FF7DECC2BA9   11DB                adc ebx,ebx
```

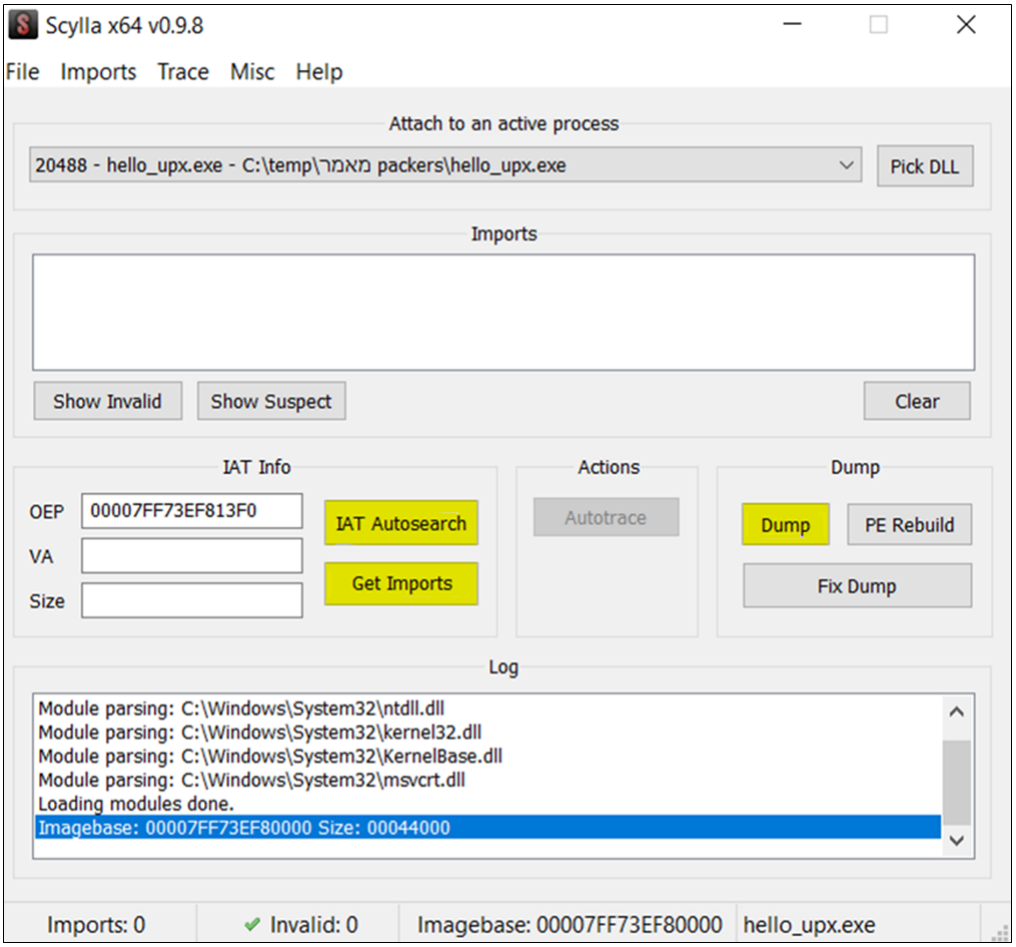
בתמונה נראה את תחילת הפונקציה החדשה שנכנסנו אליה (מזכירה שזאת הפונקציה בה מצאנו את ה-jump tail). נוכל לחפש בעין את הקפיצה או לקפוץ לאותה הכתובת שלה כפי שהיא הייתה ב-IDA כי סנכרנו את הכתובות, נשים breakpoint ב-jump tail.

```
00007FF73EFC2D60   50                  push rax                          rax:
00007FF73EFC2D61   E8 1A000000         call hello_upx.7FF73EFC2D80
00007FF73EFC2D66   58                  pop rax                           rax:
00007FF73EFC2D67   5D                  pop rbp
00007FF73EFC2D68   5F                  pop rdi
00007FF73EFC2D69   5E                  pop rsi
00007FF73EFC2D6A   5B                  pop rbx
00007FF73EFC2D6B   48:8D4424 80        lea rax,qword ptr ss:[rsp-80]     rax:
00007FF73EFC2D70   6A 00               push 0
00007FF73EFC2D72   48:39C4             cmp rsp,rax                       rax:
00007FF73EFC2D75   75 F9               jne hello_upx.7FF73EFC2D70
00007FF73EFC2D77   48:83EC 80          sub rsp,FFFFFFFFFFFFFF80
00007FF73EFC2D7B   E9 70E6FBFF         jmp hello_upx.7FF73EF813F0
00007FF73EFC2D80   C3                  ret
00007FF73EFC2D81   56                  push rsi
00007FF73EFC2D82   48:8D35 AFC2FCFF    lea rsi,qword ptr ds:[7FF73EF8F038]
00007FF73EFC2D89   48:AD               lodsq
00007FF73EFC2D8B   48:85C0             test rax,rax                      rax:
00007FF73EFC2D8E   74 14               je hello_upx.7FF73EFC2DA4
00007FF73EFC2D90   51                  push rcx
00007FF73EFC2D91   52                  push rdx                          rdx:
```

נריץ את התוכנית עד ה-breakpoint ונמשיך עם ה-debugger הוראה אחת שתריץ את ה-jump tail שלנו ותיקח אותנו אל ה-OEP:

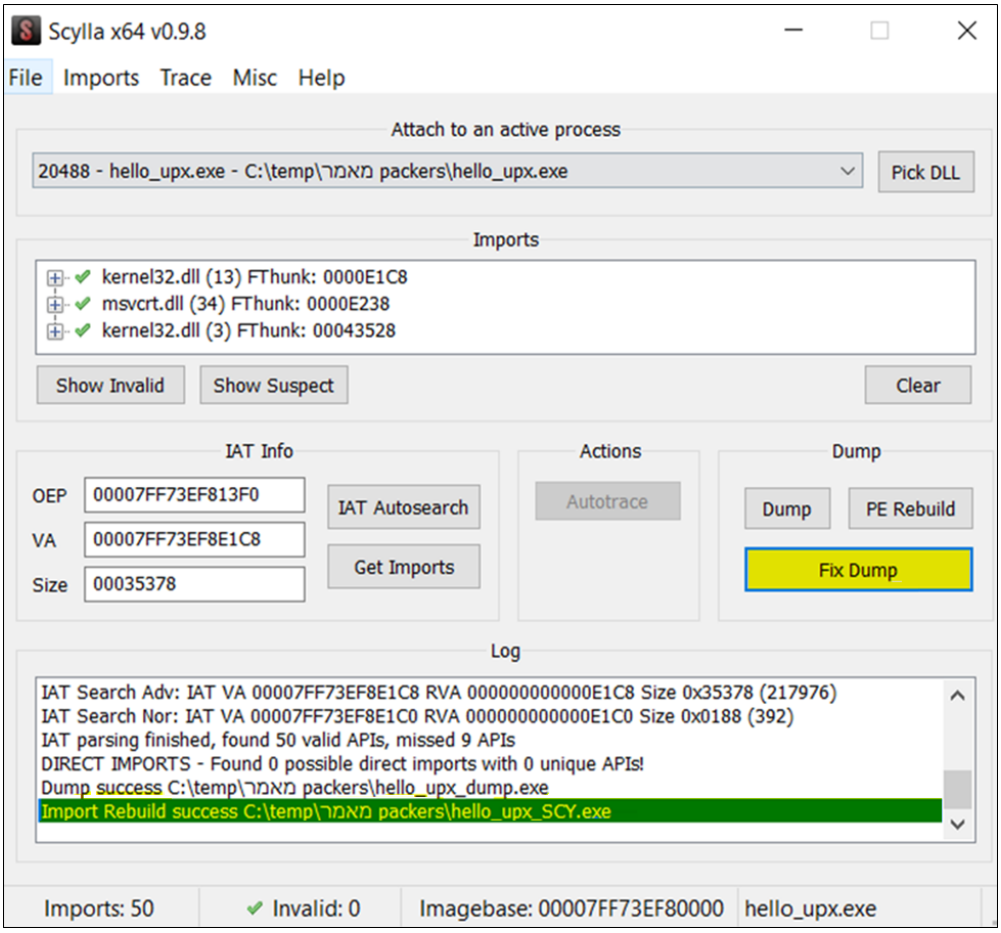



מעולה, עכשיו מה שנותר לעשות הוא לחלץ את הקובץ מהזיכרון בעזרת scylla. פשוט נלחץ על מקש ה-s (מוקף בעיגול בתמונה) יפתח לנו החלון הבא:

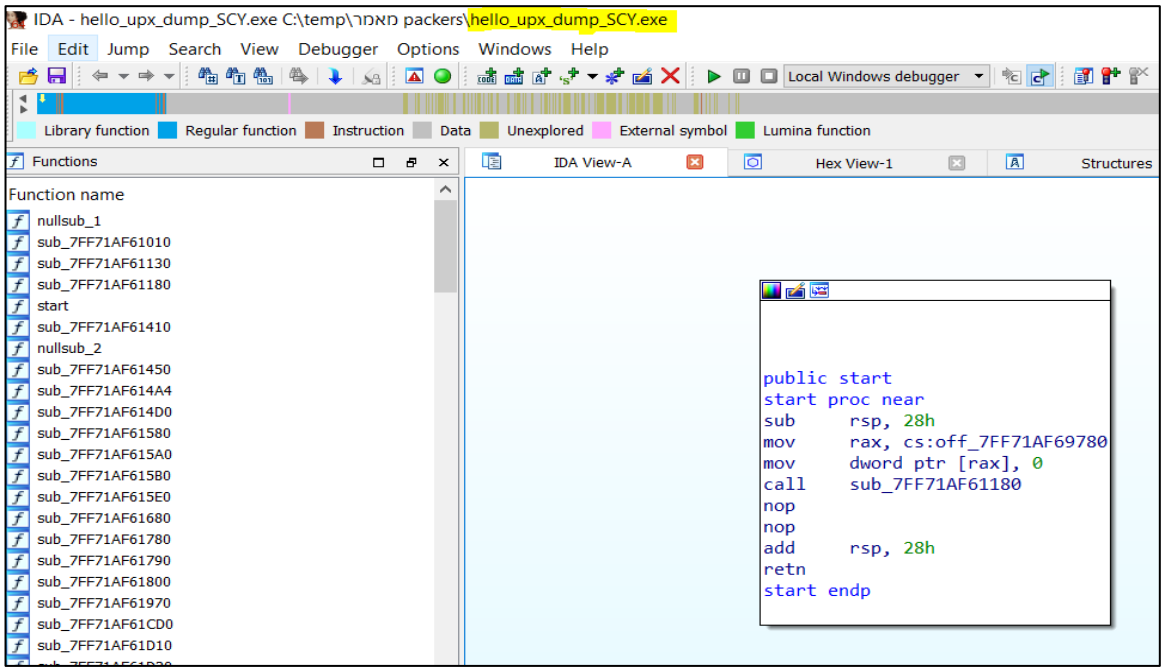



נלחץ על Get Imports, IAT Autosearch ולסיום על dump. זה יחלץ לנו את הקובץ מהזיכרון בינתן ה-OEP.

השלב הבא הוא ללחוץ על Fix Dump ולבחור את קובץ ה-dump כרגע יצרנו:




אפשר לראות בחלון מטה שהקובץ תוקן ובאיזה נתיב הוא נשמר. ננסה לחקור את הקובץ ש-scylla תיקנה ב-IDA ונראה אם התהליך צלח:



לאחר החילוץ אפשר לראות שקיימות הרבה מאוד פונקציות, ואם נחקור את הקובץ נמצא באמת את הפונקציה האחראית על ההדפסה למסך:

```
; Attributes: bp-based frame

sub_7FF71AF614A4 proc near
push    rbp
mov     rbp, rsp
sub     rsp, 20h
call    sub_7FF71AF61580
lea     rax, aHelloWorld ; "Hello, World!\n"
mov     rcx, rax
call    sub_7FF71AF61450
mov     eax, 0
add     rsp, 20h
pop     rbp
retn
sub_7FF71AF614A4 endp
```

סך הכל נראה שהצלחנו לחלץ את קובץ המקור ועכשיו נוכל לחקור אותו בצורה סטטית. אוסיף ואומר שלא תמיד תהליך החילוץ עם scylla הולך חלק, וקיימים מקרים בהם נצטרך לתקן בעצמינו גם את ה-headers של הקובץ כדי לבצע חקירה, לא אכנס למקרים אלה, אבל לרב הפלט של scylla מספק מאוד עבור ביצוע חקירה סטטית.

## מציאת ה-OEP באופן אוטומטי

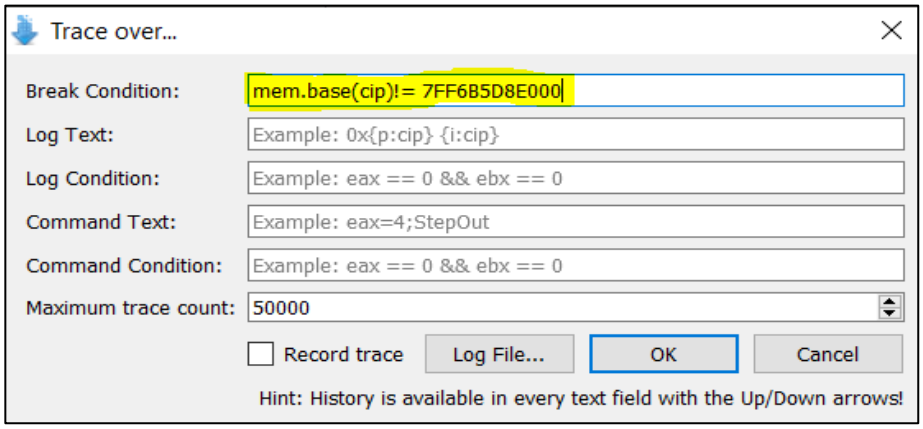
ב-section override קיימות דרכים אוטומטיות למציאת ה-OEP.

**tiny tracer** - הוא כלי שנכתב על ידי המלכה הבלתי מעורערת-hasherezade. הכלי מאחזר פונקציות API ומחפש את ה-OEP בקבצים שהם Packed. מצרפת תמונה של פלט הכלי מתוך המאמר: Manual Unpacking 0x01 אליו אצרף קישור בסוף:

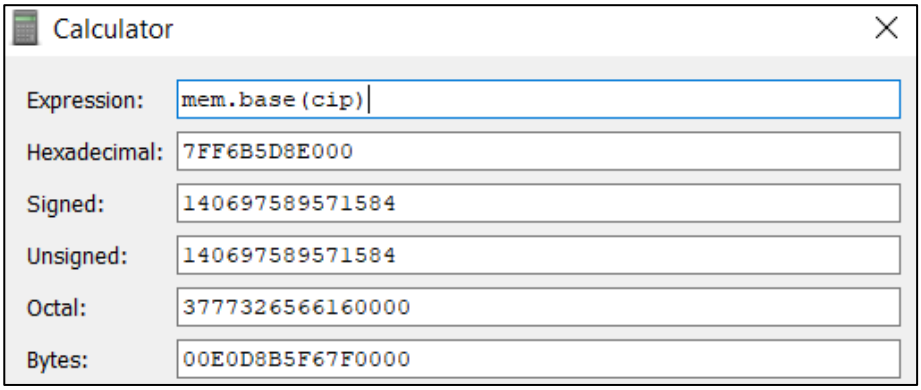
```
215ef;kernel32.LoadLibraryA
21604;kernel32.GetProcAddress
21604;kernel32.GetProcAddress
21604;kernel32.GetProcAddress
21604;kernel32.GetProcAddress
21604;kernel32.GetProcAddress
21604;kernel32.GetProcAddress
21604;kernel32.GetProcAddress
21663;kernel32.VirtualProtect
21678;kernel32.VirtualProtect
216bd;[UPX1] -> [UPX0]
11136;section: [UPX0] ---> Our OEP
13792;kernel32.GetSystemTimeAsFileTime
137a7;kernel32.GetCurrentThreadId
137b3;kernel32.GetCurrentProcessId
137c3;kernel32.QueryPerformanceCounter
14dce;kernel32.IsProcessorFeaturePresent
```

**x64dbg** - אפשר ליצור breakpoint מתוחכם שיתריע לנו שמתבצעת קפיצה ל-section אחר. נלחץ על tracing->trace over בשורה של breakpoint condition נציב את הביטוי:

mem.base(cip) != current section base address - שאומר לתוכנית לעצור כאשר תתבצע קפיצה ל-section שהוא לא ה-section בו אנחנו נמצאים כעת:




נוכל למצוא את ה-base address של ה-section דרך לשונית ה-memory map או על ידי כתיבת הביטוי mem.base(cip) ב-calculator:




:

### BP בתוך המחסנית

לעיתים, בתחילת פונקציה נהוג לשמור את ערכי ה-registers במחסנית. השמירה של ה-registers במחסנית מאפשרת לפונקציה להשתמש ב-registers ולשנות את הערכים שלהם בחופשיות. לאחר שהפונקציה סיימה את פעולתה נהוג לשלוף את הערכים שנדחפו בתחילת הפונקציה חזרה ל-registers המקוריים שלהם. כך מצב ה-registers חוזר לקדמותו בתום הרצת הפונקציה ולא מושפע מפעולתה.

נוכל להיעזר בעיקרון זה על מנת לתפוס את ה-Unpacking stub בתום פעולתו. אדגים את הטכניקה על פוגען בשם bionet.exe. ה-hash של הפוגען: 015e0624350e2eb931616f2ebdf65793

הפוגען המשתמש ב-Packer בשם ASPack. הוא Packer יחסית ישן אבל מדגים את הטכניקה בצורה טובה.

הפעם לצורך הנוחות אפתח את הקובץ ישר ב-x64dbg, כמובן שאפשר לעבוד גם בדרך הקודמת. כבר בתחילת הקוד ניתן לראות שימוש בהוראה pushad שדוחפת למחסנית את ערכי ה-registers:

```
- PID: 592 - Module: keylogger_aspack.exe - Thread: Main Thread 2304 - x32dbg [Elevated]
Plugins  Favourites  Options  Help  Apr 11 2024 (TitanEngine)
Notes | Breakpoints | Memory Map | Call Stack | SEH | Script | Symbols | Source

005AC000   60                 pushad
005AC001   EB 0A              jmp keylogger_aspack.5AC00D
005AC003   5D                 pop ebp
005AC004   EB 02              jmp keylogger_aspack.5AC008
005AC006   FF25 45FFE5E8      jmp dword ptr ds:[E8E5FF45]
005AC00C   E9 E8F1FFFF        jmp keylogger_aspack.5AB1F9
005AC011   FF                 ???
005AC012   E9 81ED236A        jmp 6A7EAD98
005AC017   44                 inc esp
005AC018   00BB 106A4400      add byte ptr ds:[ebx+446A10],bh
005AC01E   03DD               add ebx,ebp
005AC020   2B9D 466F4400      sub ebx,dword ptr ss:[ebp+446F46]
005AC026   8D8D EA6F4400      lea ecx,dword ptr ss:[ebp+446FEA]
005AC02C   51                 push ecx
005AC02D   FF95 D0704400      call dword ptr ss:[ebp+4470D0]
005AC033   8D8D 10704400      lea ecx,dword ptr ss:[ebp+447010]
005AC039   51                 push ecx
005AC03A   50                 push eax
005AC03B   FF95 CC704400      call dword ptr ss:[ebp+4470CC]
005AC041   6A 00              push 0
005AC043   54                 push esp
005AC044   6A 04              push 4
005AC046   68 20070000        push 720
005AC04B   8D8D 106A4400      lea ecx,dword ptr ss:[ebp+446A10]
005AC051   51                 push ecx
005AC052   FFD0               call eax
```

נתבונן על הכתובות במחסנית לאחר הדחיפה:

```
0012FFA4  00000000
0012FFA8  01CDFC24
0012FFAC  0012FFF0
0012FFB0  0012FFC4
0012FFB4  7FFDC000
0012FFB8  7C90E514   ntdll.KiFastSystemCallRet
0012FFBC  0012FFB0
0012FFC0  00000000
```

מזכירה שבהסבר מעלה נאמר שרק לאחר שהפונקציה סיימה את פעולתה היא ניגשת למחסנית ושולפת ממנה את הערכים חזרה ל-registers. לכן אנחנו מעוניינים לדעת כאשר תתבצע גישה לכתובת העליונה במחסנית.

נוכל לשים breakpoint בכתובת הזו במחסנית ולהגדיר שברגע שיש גישה אליה התוכנית תעצור. כדי לשים את ה-breakpoint נלחץ לחיצה ימנית על הכתובת העליונה במחסנית, נבחר breakpoint->hardware access.

נריץ את התוכנית:

```
005AC08E   61   popad
005AC08F   50   push eax
005AC090   C3   ret
005AC091   80BD A5704400 00   cmp byte ptr ss:[ebp+447045
```

נראה שנעצרנו מיד אחרי ההוראה popa ששולפת את הערכים חזרה ל-registers. ההנחה שלנו בשלב זה מכיוון שאנחנו יודעים שהחזרת הערכים המקוריים ל-registers קורית בסיום הפונקציה, היא שתהליך חילוץ הקובץ הסתיים וה-jump tail אמור להיות קרוב מאוד. בצילום אנחנו רואים שאנחנו קרובים להוראה ret. ההוראה ret שולפת את הכתובת העליונה במחסנית וקופצת אליה, לכן סיכוי סביר שמצאנו את ה-jump tail שלנו. נמשיך לדבג ונראה לאן ההוראה ret לוקחת אותנו:

```
55                  push ebp
8BEC                mov ebp,esp
83C4 E8             add esp,FFFFFFE8
53                  push ebx
56                  push esi
33C0                xor eax,eax
8945 E8             mov dword ptr ss:[ebp-18],eax
8945 EC             mov dword ptr ss:[ebp-14],eax
8945 F0             mov dword ptr ss:[ebp-10],eax
B8 1C514D00         mov eax,keylogger_aspack.4D511C
E8 BB15F3FF         call keylogger_aspack.406B04
8B35 747F4D00       mov esi,dword ptr ds:[4D7F74]
33C0                xor eax,eax
55                  push ebp
68 69574D00         push keylogger_aspack.4D5769
64:FF30             push dword ptr fs:[eax]
64:8920             mov dword ptr fs:[eax],esp
8B06                mov eax,dword ptr ds:[esi]
E8 B0C6F7FF         call keylogger_aspack.451C14
8B06                mov eax,dword ptr ds:[esi]
BA 80574D00         mov edx,keylogger_aspack.4D5780     edx:KiFastSystemCallRet, 4D5780:"BioNet 2.0 Client"
E8 A8C2F7FF         call keylogger_aspack.451818
8B0D 547F4D00       mov ecx,dword ptr ds:[4D7F54]
8B06                mov eax,dword ptr ds:[esi]
8B15 584A4D00       mov edx,dword ptr ds:[4D4A58]       edx:KiFastSystemCallRet
E8 A9C6F7FF         call keylogger_aspack.451C2C
8B0D D0804D00       mov ecx,dword ptr ds:[4D80D0]
8B06                mov eax,dword ptr ds:[esi]
8B15 60974C00       mov edx,dword ptr ds:[4C9760]       edx:KiFastSystemCallRet
E8 96C6F7FF         call keylogger_aspack.451C2C
8B0D E87C4D00       mov ecx,dword ptr ds:[4D7CE8]
8B06                mov eax,dword ptr ds:[esi]
8B15 34DF4A00       mov edx,dword ptr ds:[4ADF34]       edx:KiFastSystemCallRet
E8 83C6F7FF         call keylogger_aspack.451C2C
8B0D A0804D00       mov ecx,dword ptr ds:[4D80A0]
8B06                mov eax,dword ptr ds:[esi]
8B15 24444C00       mov edx,dword ptr ds:[4C4424]       edx:KiFastSystemCallRet, 004C4424:"pDL"
E8 70C6F7FF         call keylogger_aspack.451C2C
8B0D BC7F4D00       mov ecx,dword ptr ds:[4D7FBC]
8B06                mov eax,dword ptr ds:[esi]
```

נראה שהגענו לקוד שנראה כמו התחלה תקינה של פונקציה, בנוסף ניתן לראות שיש מחרוזת " BioNet 2.0 Client" שלא הופיעה קודם לכן ובהמשך הקוד נוכל לראות שימוש בפונקציות API נוספות. יש לנו אינדיקציה מאוד טובה שמצאנו את ה-OEP. מכאן שוב נוכל לחלץ את הקובץ עם scylla באותה הצורה שהוצגה בדוגמה הקודמת.

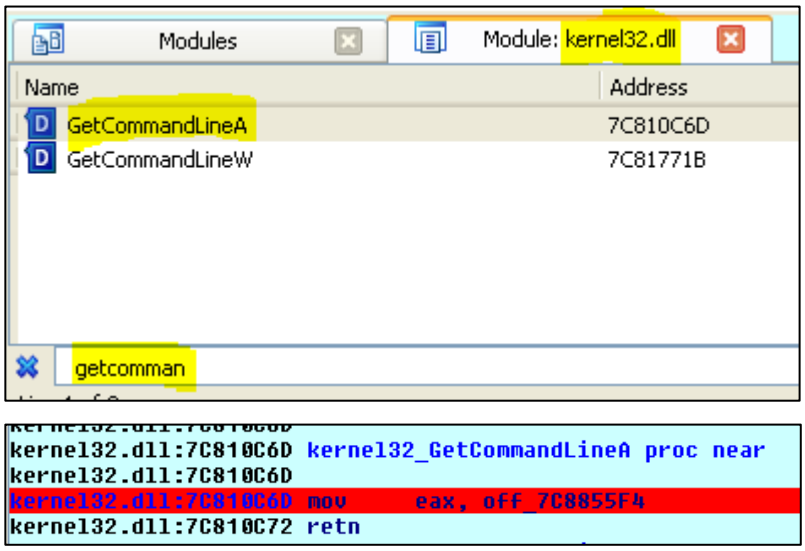
בדוגמה זו השתמשנו בקובץ 32bit וראינו שימוש בהוראה PUSHA, שדוחפת את כל ה-registers למחסנית. ראינו גם את ההוראה POPA ששולפת את הערכים חזרה ל-registers שלהם. חשוב לציין שפקודות אלו לא קיימות ב-64bit, במקומן פשוט נראה שימוש חוזר בהוראות push ו-pop.

## פונקציה שתרוץ בוודאות גבוהה

מטרת השיטה הבאה, היא לשים breakpoint על פונקצית API שאנחנו משערים שתרוץ קרוב ל-OEP, משם נוכל לעבוד לאחור, כלומר לראות מאיפה קראו לפונקציה ולנסות לחפש באותו האזור את ה-OEP.

נוכל לנסות להסתמך על פונקציות API שידוע שרצות בדרך כלל לפני ה-main או בשלב מוקדם בו. זה תלוי בתוכנית עצמה, אבל בדרך כלל מדובר פחות או יותר על אותן פונקציות, והן: GetVersion, GetCommandLine, GetStartupInfo, GetModuleHandleA וכו', סיכוי גבוהה שתחילת הפונקציה שקוראת לאחת מפונקציות ה-API הנ"ל היא ה-OEP.

נדגים את הטכניקה על אותו הפוגען שהדגמנו עליו את שיטת ה-breakpoint במחסנית. נפתח את הפוגען ב-IDA ונתחיל להריץ את ההוראה הראשונה כדי שהתוכנית תטען לזיכרון, אחר כך בחלון ה-modules נבחר את kernel32.dll ונחפש את GetCommandLineA ונשים שם breakpoint:




נריץ את הפוגען עד שנגיע ל-GetCommandLineA. מכאן המטרה היא להמשיך לדבג, ועם כל פקודת ret נחזור אחורה אל הפונקציה שקראה לפונקציה הנוכחית. כלומר אם פונקציה A קוראת לפונקציה B ופונקציה B קוראת לפונקציה C, כשנחזור מפונקציה C נגיע לפונקציה B וכשנחזור מפונקציה B נגיע לפונקציה A.

אנחנו רוצים להמשיך לחזור אחורה עד שנגיע לקוד שנראה כמו התחלה תקינה של פונקציה (בדרך כלל פונקציות יתחילו ב-push ebp/mov ebp,esp).

אם נמשיך לדבג נחזור אחורה מספר פונקציות, עד שנגיע בסוף לפונקציה הנ"ל:

```
IDA View-EIP
CODE:004D552D mov     ebp, esp
CODE:004D552F add     esp, 0FFFFFFE8h
CODE:004D5532 push    ebx
CODE:004D5533 push    esi
CODE:004D5534 xor     eax, eax
CODE:004D5536 mov     [ebp-18h], eax
CODE:004D5539 mov     [ebp-14h], eax
CODE:004D553C mov     [ebp-10h], eax
CODE:004D553F mov     eax, 4D511Ch
CODE:004D5544 call    sub_406B04
EIP
CODE:004D5549 mov     esi, ds:off_4D7F74
CODE:004D554F xor     eax, eax
CODE:004D5551 push    ebp
CODE:004D5552 push    offset unk_4D5769
CODE:004D5557 push    dword ptr fs:[eax]
CODE:004D555A mov     fs:[eax], esp
CODE:004D555D mov     eax, [esi]
CODE:004D555F call    loc_451C14
CODE:004D5564 mov     eax, [esi]
CODE:004D5566 mov     edx, offset aBionet2_0Clien     ; "BioNet 2.0 Client"
CODE:004D556B call    loc_451818
CODE:004D5570 mov     ecx, ds:off_4D7F54
CODE:004D5576 mov     eax, [esi]
CODE:004D5578 mov     edx, off_4D4A58
```

אנחנו רואים שהפונקציה שקראה ל-GetCommandLineA היא sub_406b04 שממוקמת ממש כמה הוראות אחרי ה-OEP. מכאן שוב ניתן לחלץ את הפוגען בעזרת scylla.

### מציאת פונקצית ה-API האחרונה

אזכיר שבתהליך ה-Unpacking שהוסבר תחת הכותרת "Unpacking Internals", השלב שנמצא לפני הקפיצה אל ה-OEP הוא בניית טבלת ה-imports. המטרה של השיטה הבאה היא להגיע לסוף השלב הזה, ולחפש את ה-jump tail קרוב לשם. בגלל שבדרך כלל בניית טבלת ה-Imports מתבצעת על ידי LoadLibrary ו-GetProcAddress, אנחנו צריכים למצוא את הפעם האחרונה שהפונקציה GetProcAddress תרוץ.

דרך אחת היא להניח שקודם נקראת הפונקציה LoadLibrary (המחזירה כתובת של ספרייה) ועבור כל פונקציה שנרצה למצוא בספריה הנ"ל תיקרא הפונקציה GetProcAddress (מה שבאמת קורה בהרבה מהפעמים). על פי עיקרון זה, נצטרך למצוא אזור בקוד שנראה כמו לולאה מקוננת, אחת למעבר על הספריות (שתיקרא ל-LoadLibrary) ואחת למעבר על הפונקציות (שתיקרא ל-GetProcAddress). כדי להבין מאיזה איזור בקוד נקראות הפונקציות הנ"ל נשתמש באותה השיטה בדיוק כמו בדוגמה הקודמת.

נשים breakpoint על LoadLibrary ו-GetProcAddresss ונחזור אחורה עד שנגיע לפונקציה שקראה להן:

מכאן נוכל לשים breakpoint נוסף בסוף הלולאה, אחרי שכבר אין יותר קריאות ל-LoadLibrary ו-GetProcAddress. בשלב זה תהליך בניית ה-imports מגיע לסופו וה-jump tail אמור להיות ממש קרוב!

## Unpack ללא מציאת ה-OEP

ב-Packers בסופו של דבר כשאנחנו מדברים על חילוץ קוד המקור לזיכרון, זה בסך הכל ביטוי יפה להזרקת קוד לזיכרון. הזרקה יכולה להתבצע לאזור בתהליך הנוכחי או לאזור בתהליך מרוחק. אם אנחנו מסתמכים על כך שבסופו של יום מדובר בהזרקה, נוכל לאתר את הקוד המוזרק לזיכרון בעזרת פונקציות API שמשמשות בדרך כלל להזרקת קוד. אדגים שתי שיטות, האחת היא הזרקה לתהליך מרוחק והשניה תדגים הזרקה לתהליך הנוכחי.

### BP על ProcessCreate

השיטה מתבסס על כך שידוע לנו, או שאנחנו משערים, שה-Packer הנחקר מחלץ את קובץ המקור ומריץ אותו תחת תהליך אחר. אנחנו יכולים להגיע להשערה הזאת למשל אם עשינו חקירה דינאמית וראינו שהפוגען שלנו יוצר child process (אדגים בדיוק מקרה כזה). זו באמת דרך נפוצה בה Packers משתמשים כדי להריץ את קוד המקור.

את הטכניקה הבאה אדגים על פוגען בשם emotet.

ה-hash של הפוגען: 197c2c10001134ab2a1cc87ec4382b90

אז נתחיל. אם נחקור את הפוגען בצורה דינאמית נוכל לראות שהוא יוצר תחתיו תהליך חדש:

| | | | | | |
|---|---|---|---|---|---|
| x32dbg.exe | 1.56 | 43,392 K | 69,212 K | 2692 x64dbg | |
| emotet.exe | < 0.01 | 1,224 K | 3,860 K | 3012 | Microso |
| emotet.exe | 2.33 | 1,580 K | 4,748 K | 1876 | Microso |

השתמשתי בכלי process explorer של Sysinternals, שתפקידו להציג רשימה של כל התהליכים הרצים במחשב ופרטים נוספים עליהם.

נפתח את הפוגען ב-x64dbg נשים bp על CreateProcessInternalW. (שנקראת על ידי CreateProcess) נוכל לשים breakpoint על פונקציות API בקלות באמצעות הטרמינל מטה:

```
Command: bp CreateProcessInternalW
Paused   Breakpoint at 75633BBB set!
```

נריץ את הפוגען עד ה-breakpoint. ההנחה שלנו כעת היא שאם התהליך אליו מזריקים כבר נוצר (יקרה לאחר ש-CreateProcessInternalW תרוץ) ככל הנראה ה-data שיוזרק אליו, הלא הוא קובץ המקור אותו אנחנו מחפשים כנראה כבר כתוב בזיכרון.

נוכל לערוך השוואה בין ההקצאות בזיכרון שהן RWX לפני ואחרי הרצת הפונקציה CreateProcess (או לעבור אחת אחת) ולראות הם נוצרה הקצאה חדשה חשודה ומה יש בה:

emotet.exe - PID: 2272 - Module: kernel32.dll - Thread: Main Thread 1688 - x32dbg [Elevated]

File View Debug Tracing Plugins Favourites Options Help Apr 11 2024 (TitanEngine)

CPU | Log | Notes | Breakpoints | Memory Map | Call Stack | SEH

| Address | Size | Party | Info | Content | Type | Protection | Initial |
|---|---|---|---|---|---|---|---|
| 00010000 | 00010000 | User | | | MAP | -RW-- | -RW-- |
| 00020000 | 00008000 | User | | | PRV | ERW-- | ERW-- |
| 00030000 | 00001000 | User | | | PRV | ER--- | ERW-- |
| 00040000 | 00001000 | User | | | IMG | -R--- | ERWC- |
| 00050000 | 000FC000 | User | Reserved | | PRV | | -RW-- |
| 0014C000 | 00004000 | User | Stack (1688) | | PRV | -RW-G | -RW-- |
| 00150000 | 00004000 | User | | | MAP | -R--- | -R--- |
| 00160000 | 00001000 | User | | | PRV | -RW-- | -RW-- |
| 00170000 | 00067000 | User | \Device\Harddiskvolume1\ | | MAP | -R--- | -R--- |
| 001E0000 | 00039000 | User | Reserved | | PRV | | -RW-- |
| 00219000 | 00007000 | User | | | PRV | -RW-G | -RW-- |
| 00220000 | 00001000 | User | | | PRV | ER--- | ERW-- |
| 00230000 | 00001000 | User | | | PRV | ER--- | ERW-- |
| 00240000 | 00001000 | User | | | PRV | ER--- | ERW-- |
| 00250000 | 00001000 | User | | | PRV | ER--- | ERW-- |
| 00260000 | 00001000 | User | | | PRV | ER--- | ERW-- |
| 00270000 | 00001000 | User | | | PRV | ER--- | ERW-- |
| 00280000 | 00001000 | User | | | PRV | ER--- | ERW-- |
| 00290000 | 00001000 | User | | | PRV | ER--- | ERW-- |
| 002A0000 | 00001000 | User | | | PRV | -RW-- | -RW-- |
| 002B0000 | 00001000 | User | | | PRV | -RW-- | -RW-- |
| 002C0000 | 00012000 | User | | | PRV | -RW-- | -RW-- |
| 002E0000 | 0000E000 | User | | | PRV | ERW-- | ERW-- |
| 002F0000 | 0000E000 | User | | | PRV | -R--- | -RW-- |
| 00300000 | 00010000 | User | | | PRV | ERW-- | ERW-- |

בחלון ה-memory map נוכל לראות את ההקצאות שההרשאות שלהן הן RWX. ההקצאה שנמצאת בכתובת 002E0000 לא הייתה קיימת קודם לכן. נלחץ על הכתובת לחיצה ימנית ואז follow in dump שתציג לנו את התוכן שנמצא בכתובת הנ"ל בחלון ה-dump. נוכל לראות שהתוכן שלנו הוא קובץ הרצה:

```
Dump 1   Dump 2   Dump 3   Dump 4   Dump 5   Watch 1
Address   Hex                                              ASCII
002E0000  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ..........ÿÿ..
002E0010  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ¸.......@.......
002E0020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
002E0030  00 00 00 00 00 00 00 00 00 00 00 00 D0 00 00 00  ............Ð...
002E0040  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ..º..´.Í!¸.LÍ!Th
002E0050  69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
002E0060  74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t be run in DOS 
002E0070  6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.......
002E0080  78 9B 95 E3 3C FA FB B0 3C FA FB B0 3C FA FB B0  x..ã<úû°<úû°<úû°
002E0090  3C FA FA B0 20 FA FB B0 35 82 68 B0 35 FA FB B0  <úú° úû°5.h°5úû°
002E00A0  3C FA FB B0 3D FA FB B0 31 A8 25 B0 3D FA FB B0  <úû°=úû°1¨%°=úû°
002E00B0  52 69 63 68 3C FA FB B0 00 00 00 00 00 00 00 00  Rich<úû°........
002E00C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
002E00D0  50 45 00 00 4C 01 04 00 5A 88 5F 5A 00 00 00 00  PE..L...Z._Z....
002E00E0  00 00 00 00 E0 00 02 01 0B 01 0C 00 B0 29 00 00  ....à.......°)..
002E00F0  80 2A 00 00 00 00 00 00 02 39 00 00 00 10 00 00  .*.......9......
002E0100  00 40 00 00 00 00 40 00 00 10 00 00 10 00 00 00  .@....@.........
002E0110  05 00 00 00 00 00 00 00 05 00 00 00 00 00 00 00  ................
002E0120  00 E0 00 00 70 02 00 00 00 00 00 00 03 00 40 81  .à..p.........@.
002E0130  00 00 10 00 00 10 00 00 00 00 10 00 00 10 00 00  ................
002E0140  00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00  ................
002E0150  A8 5E 00 00 64 00 00 00 00 00 00 00 00 00 00 00  ¨^..d...........
```

כעת, מכשמצאנו קובץ חשוד, נוכל לחלץ את כל ההקצאה הזאת מהזיכרון, נחזור שוב לחלון של ה- memory map נלחץ לחיצה ימנית ואז "Dump Memory to File".

**VirtualProtect**

עוד דרך לחילוץ קוד המקור היא הזרקה עצמית, כלומר הפוגען מקצה זיכרון בתהליך של עצמו ויריץ אותו משם. בדרך כלל זה יעבוד באופן הבא:

- הקצאת זיכרון חדש בעזרת VirtualAlloc.
- כתיבת קוד המקור למרחב הזיכרון החדש.
- מתן הרשאות ריצה להקצאה החדשה שנוצרה בעזרת VirtualProtect.

את אופן הפעולה אדגים על פוגען בשם: REvil.

ה-hash של הפוגען: 3c19e7ce627da9b5004371f867a47d61

נמקם את ה-breakpoint הראשון שלנו על VirtualAlloc כדי שנוכל לעקוב אחר ההקצאות החדשות שנוצרות. נריץ את הפוגען עד שנראה קריאה ל-VirtualAlloc. הפונקציה מחזירה (ושומרת ב-EAX) את הכתובת בה מתחילה הקצאת הזיכרון החדשה, לכן נרצה להמשיך עם הפונקציה עד שהיא נגמרת על מנת שנוכל לדעת באיזה כתובת מתחילה ההקצאה. בשביל לרוץ עד לסוף פונקציה מסוימת קיימת ב-x64dbg אופציה הנקראת run until return, שכשמה כן היא, תרוץ עד שתתקל בהוראה ret/retn.

נשתמש בפעולה זאת על ידי לחיצה על Ctrl+F9.

הוקצה אזור זיכרון חדש בכתובת 0540000, נלחץ לחיצה ימנית ו-follow in dump ונראה את התוכן של ההקצאה. לפי ההסבר מעלה, איפשהו בין הקריאה ל-VirtualAlloc לבין הקריאה ל-VirtualProtect אמור להיכתב מידע לכתובת הנ"ל. נלחץ שוב על F9 שתריץ את הפוגען שלנו עד ה-breakpoint הבא ונראה שנעצרנו בתחילת הפונקציה VirtualProtect. נתבונן שוב במידע שבחלון ה-dump, ונראה שהפעם קיים שם מידע חדש:

נחלץ את הקובץ באותו האופן כמו בדוגמה הקודמת.

חשוב לציין שאפשר להשתמש בפונקציה VirtualAlloc על מנת להקצות זיכרון חדש ובאותו הזמן להגדיר את ההרשאות שנרצה. במקרה כזה אפשר לעקוב אחר ההקצאות החדשות שנוצרות באמצעות follow in dump, ולשים לב למידע הנכתב לשם. באופן דומה אפשר להסתכל בחלון ה-memory map ולחפש הקצאות חדשות, בדגש על הקצאות עם ההרשאות RWX.

חשוב לציין שקיימות המון דרכים להזריק לזיכרון (הדגמנו כאן רק שתי דוגמאות). בתור התחלה הייתי ממליצה לקרוא ולהכיר עוד שיטות נפוצות להזרקת קוד. אחרי שהכרתם עוד שיטות, תוכלו להוסיף breakpoints על פונקציות API נוספות שיעזרו לכם לתפוס עוד סוגי הזרקות. דוגמה לפונקציות API נוספות המשמשות להזקה הן:

WriteProcessMemory, CreateRemoteThread, MapViewOfSection, ResumeThread

וכדומה.

## Tips and Tricks

החלק הזה נותן שני טיפים, שאם נשתמש בהם סימן שלא הצלחנו לבצע Unpacking לפוגען באמצעות הטכניקות הקודמות שהצגנו. אלו טיפים שיכולים לשמש קצת כירייה לכל הכיוונים, אך בסופו של דבר יכולים להניב תוצאות.

### Snapshot the idb

מה שאנחנו נעשה בשיטה הזו, זה לנסות לעבור את ה-Unpacking stub ולהגיע למצב שבו הקובץ המקורי כבר נמצא בזיכרון. לאחר מכן ב-IDA יש אופציה לבצע snapshot ל-idb. ה-snapshot ישמור את כל הפונקציות שנטענו דינאמית לזיכרון גם לאחר שהקובץ סיים את ריצתו.

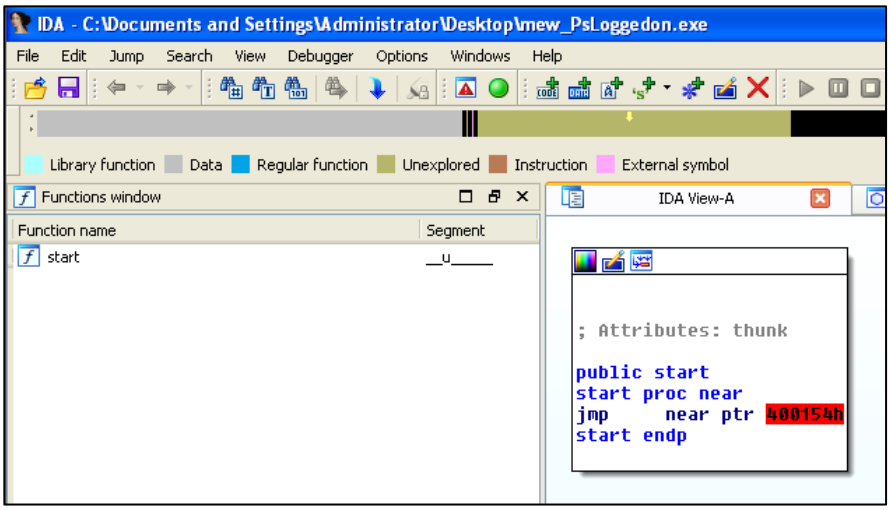
לפני ביצוע ה-snapshot נצטרך להיות בטוחים שתהליך ה-Unpack באמת הסתיים. הינה כמה אינדיקציות לכך:

- נראה קריאה לפונקציות API שלא ראינו אותן בטבלת ה-imports בחקירה סטטית.
- כנ"ל לגבי מחרוזות, נתחיל לראות בהן שימוש.
- נוכל לראות שנטענו ספריות נוספת שלא ראינו בצורה סטטית.

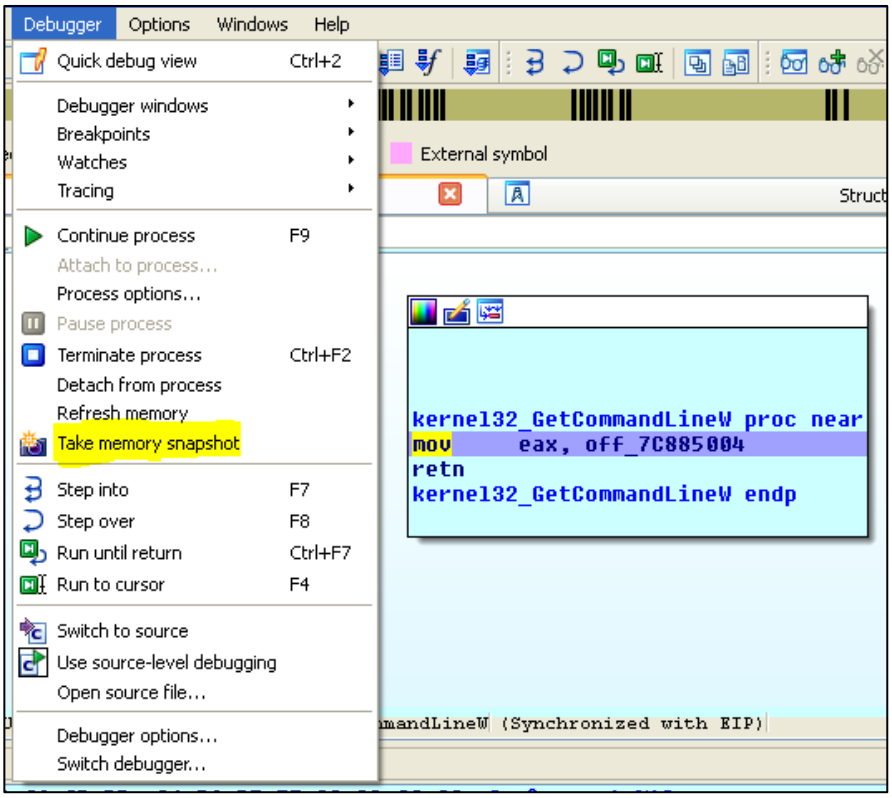
נוכל גם לשים breakpoint על פונקציה שאנחנו יודעים שהקובץ המקורי בוודאות משתמש בה, לדוגמה אם אנחנו יודעים שהקובץ המקורי יוצר mutex נוכל לשים breakpoint על CreateMutex וכשנגיע לשם נדע

בוודאות שהקובץ המקורי נמצא בזיכרון והתחיל לרוץ (זה מאוד דומה לשיטה שראינו קודם לכן המחפשת פונקציות מוכרות שירוצו בוודאות גבוהה).

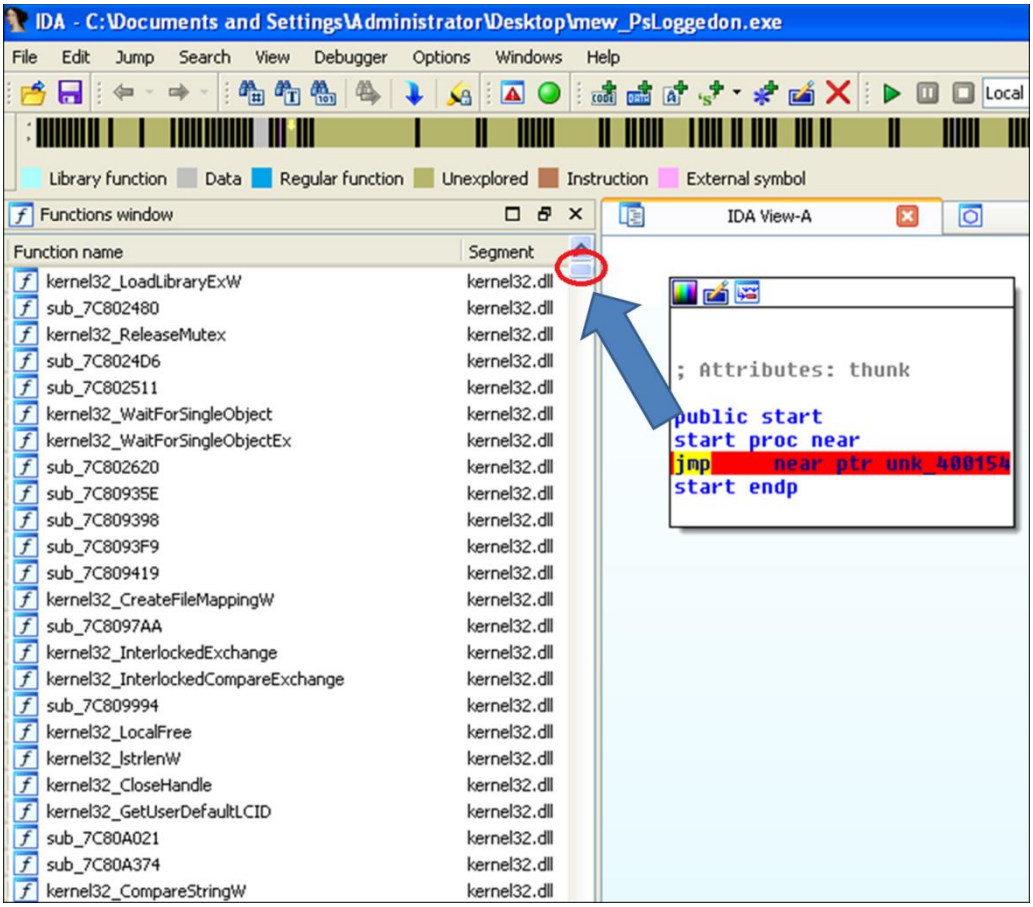
לצורך הדוגמה אשתמש בקובץ Psloggedon (קובץ של Sysinternals שתפקידו להחזיר את מספר המשתמשים המחוברים לעמדה מסוימת) שביצעו לו Packed באמצעות Packer בשם MEW (הקובץ נלקח מ-github יצורף קישור בסוף המאמר):




במבט ראשוני ב-IDA אפשר לראות שקיימת כרגע רק פונקציה אחת. נשים breakpoint על GetCommandLineW (כפי שהודגם לפני כמה דוגמאות):



נריץ את התוכנית, אחרי שהגענו ל-breakpoint ואנחנו חושבים שהקובץ כבר בשלמותו/ברובו בזיכרון, נלחץ על Debugger->Take memory snapshot. כך תשמר לנו תמונה של כל הזיכרון של הפוגען כולל כל הפונקציות, המשתנים המבנים וכו' גם כשהפוגען יסיים את ריצתו!




בתמונה אפשר לראות את כמות הפונקציות החדשות שנוספו לאחר שלקחנו את ה-snapshot והפוגען כבר לא רץ.

איך זה יכול לעזור? קודם כל כנראה שעכשיו יש לנו את רב הפונקציות של הקובץ המקורי, (אם לא כולן) והיתרון שלנו הוא שעכשיו אנחנו יכולים לחקור אותו בצורה סטטית שזו כבר התקדמות משמעותית מנקודת ההתחלה. אפשר באמצעות חקירה סטטית לנסות להבין איפה ה-entry point ואפשר גם לנסות להבין איזה פונקצית API נקראת ראשונה או בין הראשונות ומשם לתפור אחורה ל-OEP.

### MZ string brute force

גם כאן נצטרך להגיע למיקום בקובץ בו אנחנו די בטוחים שה-Unpacking stub סיים את פעולתו. במידה ושום שיטה לא צלחה, נוכל לחפש את המחרוזת MZ (ה-magic number המייצג את תחילתו של קובץ PE) בתקווה למצוא קובץ PE חדש. דרך נוספת היא לעבור על הקצאות הזיכרון שקיימות עבורן הרשאות ריצה ולבדוק את התוכן של כל הקצאה.

## סיכום

המאמר סוקר בצורה נרחבת את טכניקת ה-Packing בדגש על חקר פוגענים. ראינו איך נראה קובץ שהוא Packed וכעת נוכל לזהות קבצים כאלו בעצמנו. הוצגו מספר טכניקת לביצוע Unpacking שהודגמו על פוגענים שונים, ולמדנו איך לבצע כל טכניקה בעזרת Debugger. בנוסף הכרנו את הכלי scylla, המשמש לחילוץ דינאמי של קובץ מהזיכרון, שהוא חלק מרכזי בתהליך ה-Unpacking. מקווה שנהנתם ושהמאמר הוסיף לכם כישורים חדשים לארגז הכלים!

## מקורות מידע


הסבר קצר על Anti-debugging:

https://www.linkedin.com/pulse/malware-anti-analysis-rakesh-patra-f1utc

Practical malware analysis פרקים 1 ו-3 מסבירים על חקירה סטטית ודינאמית בסיסית של פוגען, פרק 18 מדבר על Packers:

https://doc.lagout.org/security/Malware%20%26%20Forensics/Practical%20Malware%20Analysis.pdf

Malshare - פלטפורמה המיועדת לשיתוף פוגענים:

https://malshare.com/

עמוד ה-GitHub של הכלי-Tiny Tracer:

https://github.com/hasherezade/tiny_tracer

מאמר המסביר איך מחלצים-UPX ואת הפוגען REviil:

https://joezid.github.io/malware-analysis/2021/01/11/Manual-Unpacking-0x01.html

ערוץ היוטיוב של Open analysis live. מעלים המון סרטונים מעולים על חקר פוענים:

https://www.youtube.com/@OALABS/search?query=ida%20tips

עמוד ב-Github שמכיל המון מאמרים בנושא Packing ו-Unpacking:

https://github.com/packing-box/awesome-executable-packing

מאגר גדול של קבצים שהם Packed לתרגול:

https://github.com/packing-box/dataset-packed-pe/tree/master