רז אברג'ל

Bispoo 2

מהם מנגנוני האנטי דיבג וכיצד התגברתי עליהם?

ראשית כשנטען את הקוד בxdbg32 נבחין כי הקוד לא רץ בכלל כשאני מנסה לדבג אותו

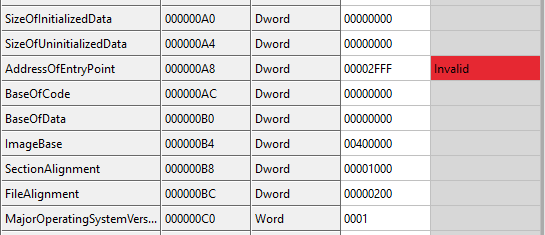
ננסה לנתח את הקוד דרך IDA ולהבין מה הגורם שמונע דיבוג

נבחין כי הentry point נמצא במקום שלא קורה בו כלום למעשה



נכנס לcff כדי לנסות להבין איפה ממוקמת נקודת ההתחלה של הקוד

נראה כי למעשה הentry point מוביל לכתובת שחורגת מגבול הכתובות החוקיים

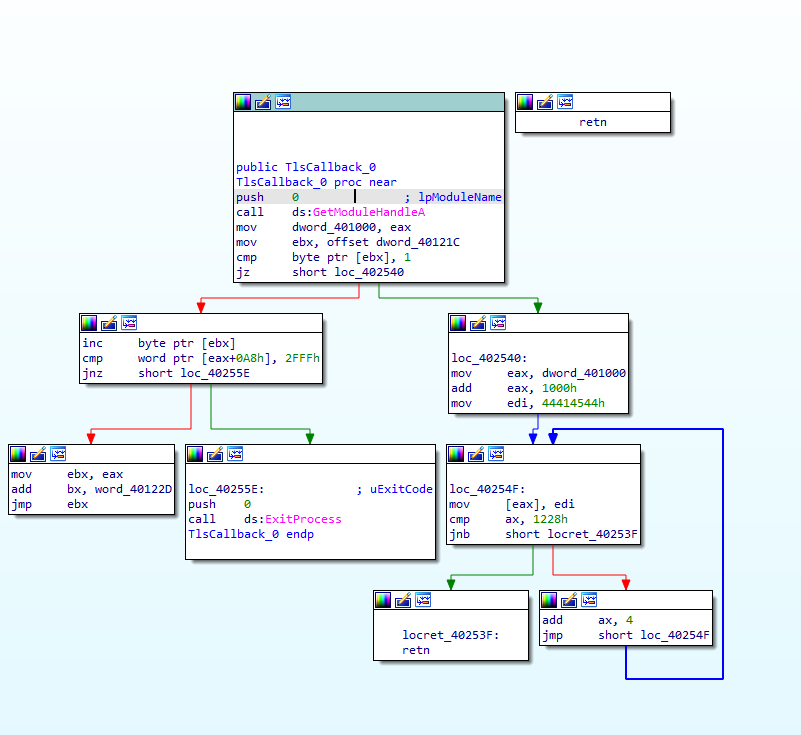


ואני מבין שכנראה יש פה יש גורם אנטי דיבאג שמחבל בניסיונות שלי להגיע לנקודת הכניסה לקוד.

בחלון הפונקציות בida נוכל לראות שיש קריאה ל**tls** ואנחנו יודעים שtls נקרא ראשון עוד לפני המיין ועל כן הוא חשוד

שהוא זה שמשבש את כתובת ההתחלה

נכנס לפונקציה ונבחן אותה



מהתבוננות בקוד אפשר לראות שבחלק העליון נלקחת הכתובת של התחלת הריצה של קובץ PE

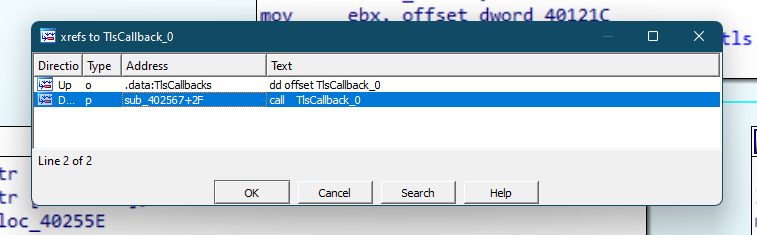
ונכנסת לeax ולאחר מכן מכניסים את הערך 0 לebx ועושים לו פעולת cmp עם הערך 1

נראה כי בהתסעפות השמאלית של הקוד יש לנו הגדלה ב1 של הערך שבebx ולאחר מכן בדיקה עם הכתובת

2fffh שהיא כזכור הכתובת הראשונית שהגענו אליה בניתוח

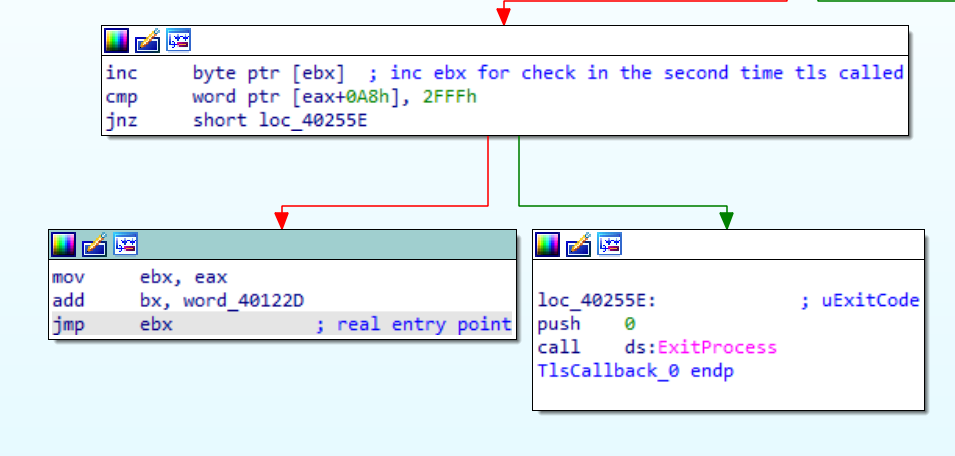


מכיוון שזהיתי שיש קריאה בקוד לפונקציה הזאת (בנוסף על ההרצה הראשונה של TLS שהיא רצה לפני הקוד )



אני מבין שהחלק שאנחנו רוצים להגיע אליו הוא החלק השמאלי מכיוון שלאחר הרצה שנייה של הtls ערכו של ebx יהיה שווה ל1 ואז נגיע לחלק הימני ששם יש לולאה שמשנה את ערכו של EAX שכזכור מכילה את הכתובת של נקודת הכניסה לקוד (eax קיבל זאת getModuleHandelA  .)

ומכאן אני מבין שזה חלק בעייתי שלא נרצה להגיע אליו .



בחלק זה אני רואה שאם הכתובת התחלה לא השתנתה (זאת אומרת שהייתה רק את הקריאה הראשונה ל ( tls מה שאומר שכביכול אני לא מדבג את התוכנית ואז נגיע לחלק שלמעשה מוסיף לeax (שעבר לebx) את ההיסט

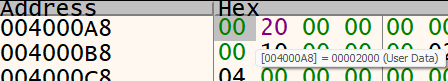
2000h שלוקח אותו לכתובת שבה באמת מתחיל הקוד הנכון.

נבצע שינויים בפונקציה כדי לנטרל את האנטידיבג שלה.

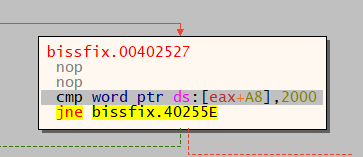
במקום הפקודה 

נשים nopים כדי שהפונקציה לא תדע שהיא נקראת בפעם השניה

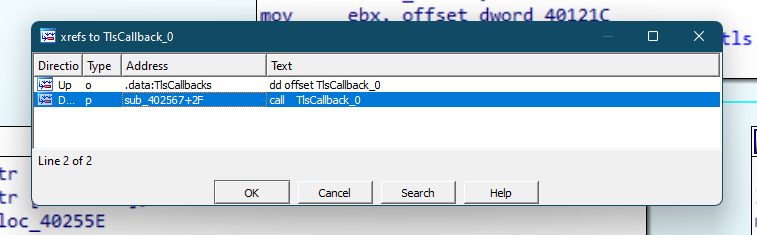
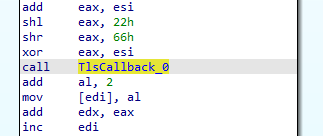
בנוסף במקום ההשוואה 

נשנה את הפקודה כך שהיא תמיד תשווה את הכתובת 2000h עם עצמה על מנת לעשות זאת ניגש לזיכרון ונשנה במיקום של eax+8a את הכתובת מ2fff ל2000 וככה הפונקציה לא תזהה שמשנים את נקודת הכניסה בדיבוג 

והפצפוץ

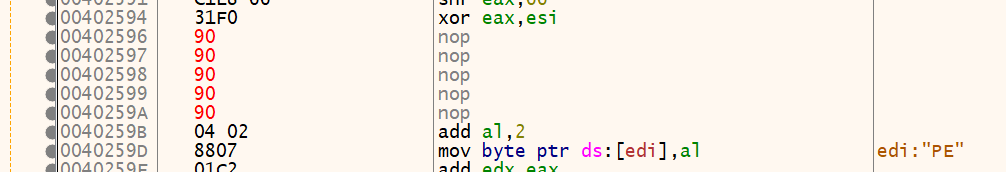


בגלל שזהינו שיש קריאה נוספת לtls נצטרך לנטרל את הקריאה השנייה כדי שהתוכנית לא תחזור שוב לentry point והתוכנית תשתבש

נפצ'פץ' את הפונקציה כך שהיא לא תקרא בפעם השניה

נבצע זאת בx32 במקום הקריאה הנוספת לtls call back נשים כמה nopים

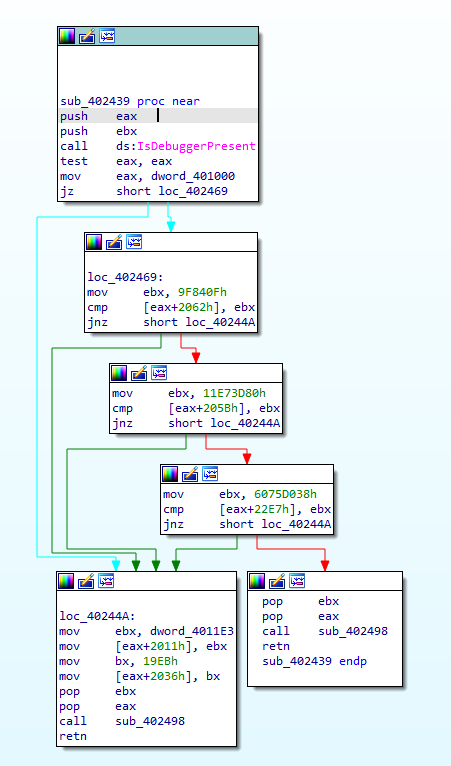


וכך למעשה התגברנו על האנטי דיבג הראשון שעמד לפתחנו.

כשנמשיך לחקור את התוכנית עם ida נראה את הקריאה לפונקציה הראשונה



והפונקציה נראית כך



רואים בירור שבפונקציה זאת יש לנו אנטי דיבג נוסף isDebuggerPresent

שמשנה את ערכו של השדה BeingDebugged שנמצא במבנה הנתונים PEB

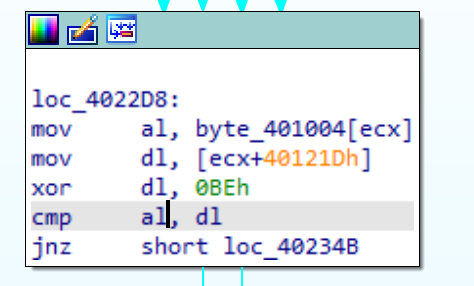
ומודיע האם התוכנית מדובגת אם התוכנית אכן מדובגת הפוקנציה תחזיר מספר שונה מאפס ואז נגיע לפה

וינזק לנו הקוד, בכתובות   יש לנו פעולות חשובות שנדרסות כמו למשל  .כשנצליח להסתתר והפונקציה לא תזהה שיש תהליך דיבוג נגיע להסתעפות הימנית של הקוד

באותה הסתעפות ימנית נוכל לראות שלוש בדיקות של אזורים בקוד האם הם עברו שינוי או לא

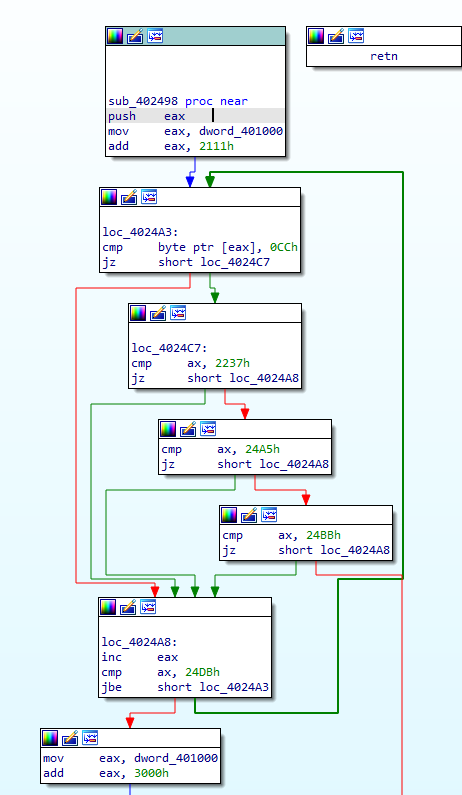
בדיקה זאת כנראה באה לבדוק אם ביצענו פיצפוץ וזהו למעשה מעין אנטי דיבג נוסף

למשל ב  אנו רואים פעולה של השוואה וקפיצה מותנית



רגע לפני פקודת הretn אני קוראים לפונקציה נוספת שנקראת גם אם מגיעים לחלק הטוב וגם לחלק הרע 

נכנס לפונקציה זאת



נוכל לראות שיש פה דפוס מוכר של בדיקה של אזורים שיש בהם break point  אם הוא ימצא עוד ברייק פוינטים נוספים הוא יבין שיש פה תהליך של דיבוג ועל כן הוא ישבש לנו את הקוד.

לסיכום יש לנו בפוקנציה זאת למעשה שני כלים של אנטי דיבג (isDebuggerPresent ובדיקה של ברייק פוינט נוספים)

ובנוסף בדיקה של שינוי קוד שגם זה סוג של אנטי דיבג כיוון שהוא רוצה לגלות אם יש פצפוץ וכתוצאה מכך גם דיבוג.

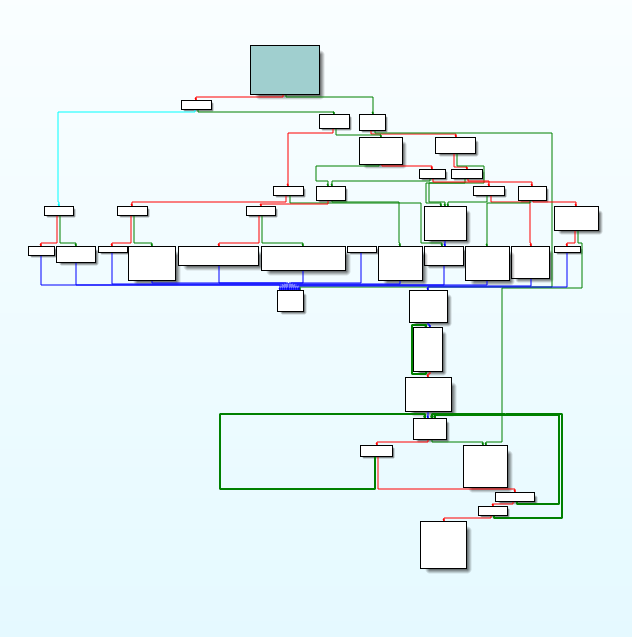
נשנה את השם של הפונקציה לשם נוח יותר בשביל לראות מתי היא נקרא בהמשך הקוד.



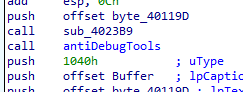
עכשיו נכנס לפרמטר שנשלח לפונקציה  שמכיל את כל המידע הרלוונטי לנו

הפרמטר הרלוונטי הוא 

נגיע כאן לגרף מייצג מאוד גדול של הקוד

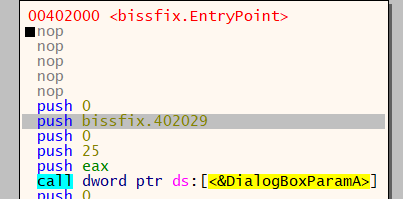


ונראה שיש כמה מקומות שבהן יש קריאה לפונקציה 

למשל 

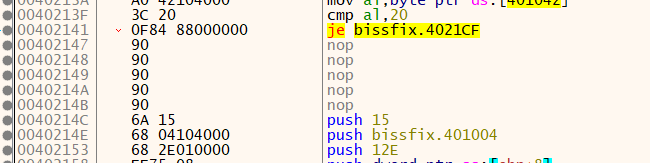
מכאן אנחנו מבינים שכדי לבצע אנליזה דינמית לתוכנית ולמצוא את הסיסמא הנכונה נהיה חייבים לנטרל את כל הקריאות הללו על מנת שלא יגלו את הדיבוג שלנו.

ראשית נפצפץ את הקריאה הראשונה לפונקציה שנקראת לפני דחיפת הפרממטרים שמגיעים לפונקציה הראשית



בכניסה אזור שמכיל את הנתונים שיהיו בdialog box

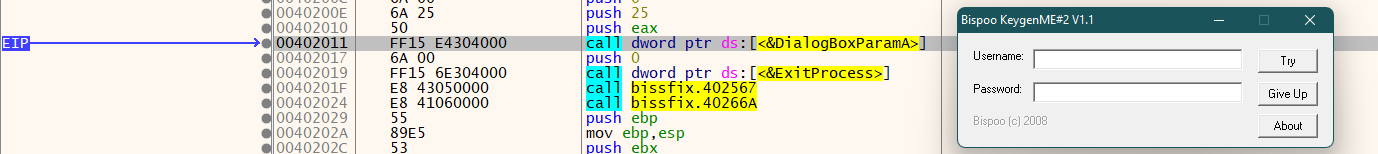
נזהה שלושה מקומות בהן antiDebugTools נקראת וכל מקום כזה נפצפץ ושים nopים במקום



וכך ביתר המקומות.

וככה למעשה התגברנו על הפונקציה והתוכנית לא מזהה שמדבגים אותה.

כשנריץ מחדש את הקובץ נוכל לראות שאנחנו מגיעים לחלונית

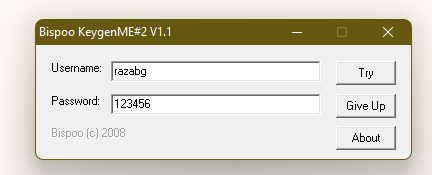
שנפתחת בהפעלה התוכנה 

ומכאן שסיימנו את החלק הראשון של זיהוי מנגנוני הדיבאג ,הסבר פעולתם והניטרול של

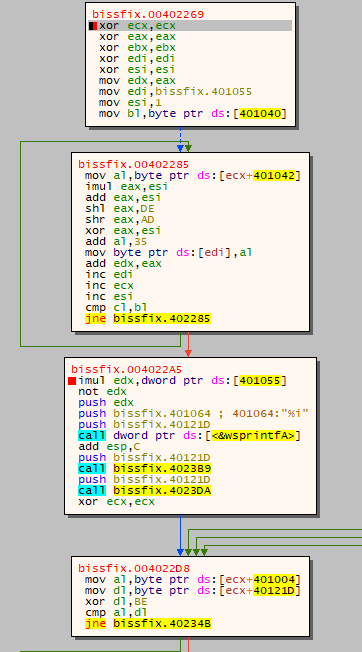
ועכשיו נמצא את הסיסמא

נזין שם משתמש וסיסמא רנדומליים על מנת להבין מהי הסיסמא הנכונה

כמובן שעכשיו לאחר פירוק כל מנגנוני האנטי דיבג נוכל לדבג את התוכנה כמו שצריך



במהלך הדיבוג נגיע לקטע הזה

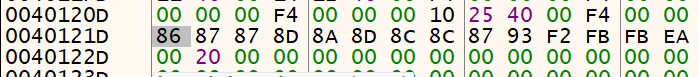


נראה כי בקטע זה מתבצעת מניפולציה על אזור הזיכרון שבו מאוחסן שם המשתמש

ומכאן אני מבין שהסיסמא בנויה למעשה מהקלט של שם המשתמש

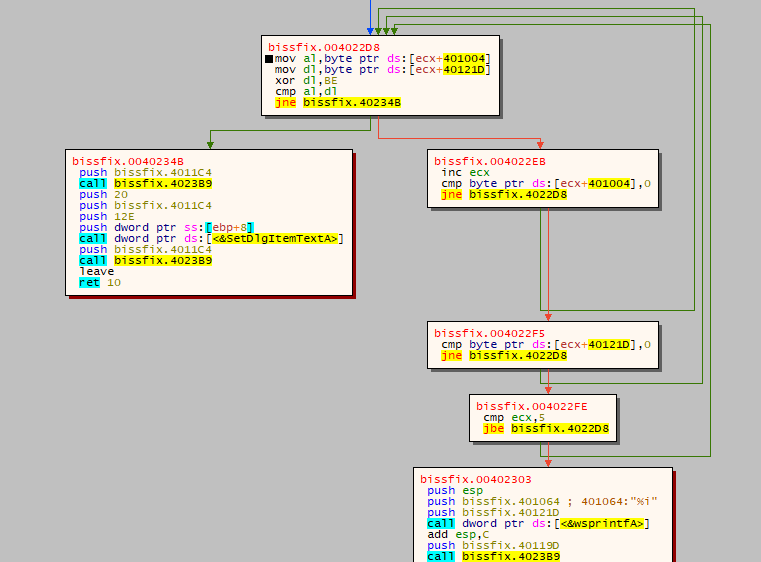
בהמשך הקוד בנקודה זו,אנחנו מזהים שיש פה השוואה בין מחרוזת מסוימת שנמצאת בזיכרון

לבין אזור הזיכרון שבו נמצאת הסיסמא שהזנתי,לפני ההשוואה נראה שיש פעולת XOR על תו בורק אז נשווה אותו לתו המקביל במחרוזת של הסיסמא

נתבונן במחרוזת הזאת בזיכרון 

יש כאן אוסף של תווים שיטענו לתוך dl ובכל פעם נבצע פעולתBE XOR על מספר ברצף

בכל פעם שהתו יהיה תואם לתו הראשון של הסיסמא הנכונה נתקדם אל עבר התו הבא ונבדוק גם אותו



ברגע כל התוים יהיו תואמים והסיסמא תהיה נכונה נגיע אל האזור הטוב

נבדוק מה יוצא לנו לאחר ביצוע  על כל אחד מהמספרים ההקסהדצימלים ברצף שאנחנו משווים אליו את הסיסמא שהזנו

ונקבל



אם נכניס את הסיסמא הזאת לשם המשתמש שהזנו בהתחלה נראה שהסיסמא הזאת לא עובדת

אני מבין שלמעשה יש הצפנה של הקוד כך שלא נוכל לזהות מה הסיסמא האמיתית

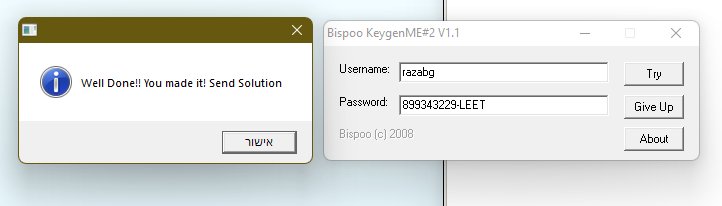
ננסה להמיר את הרצף הזה לתוי אסקי ולהזין את הסיסמא ונראה מה נקבל

הרצף שקיבלנו הוא 

והוא כבר נראה לנו משהו הגיוני

ננסה להזין את הסיסמא עבור שם משתמש זה (razabg)

ונראה שזאת אכן הסיסמא!



ובכך סיימנו את התרגיל כולו.