Digital Whisper Technologic Papers

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

<u>הקדמה ומושגים בסיסיים</u>

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.

```
int square(int num)
{
    return num * num;
}
```

[קוד מקור בשפת C]

```
square(int):
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     eax, DWORD PTR [rbp-4]
        imul    eax, DWORD PTR [rbp-4]
        pop     rbp
        ret
```

[שפת Assembly]

```
55                  push   rbp
48 89 e5            mov    rbp,rsp
89 7d fc            mov    DWORD PTR [rbp-0x4],edi
8b 45 fc            mov    eax,DWORD PTR [rbp-0x4]
0f af 45 fc         imul   eax,DWORD PTR [rbp-0x4]
5d                  pop    rbp
c3                  ret
```

[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.

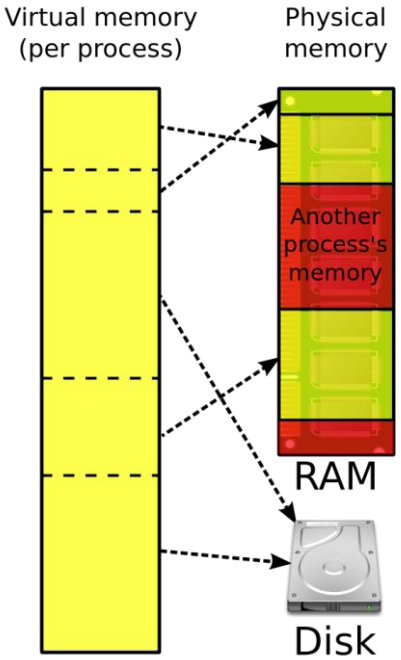
3. Linking - החיבור של הכל ביחד

- 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)]

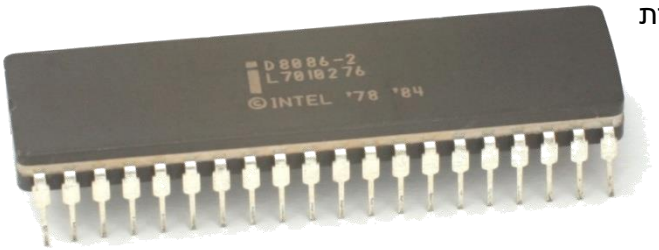
בתחילת ההתפתחות של מחשבים, כאשר העקרון של זכרון וירטואלי עדיין לא מומש, התעוררו מספר בעיות בניהול של מספר תוכנות שרצות במקביל: קודם כל, גודל של זכרון RAM סטנדרטי ברוב הפעמים לא מספיק כדי לאחסן את כל המידע שתוכנות רוצות לשמור בכל רגע נתון. שנית, מאחר וכל התוכנות היו שומרות את המידע שלהן על אותו טווח של 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)

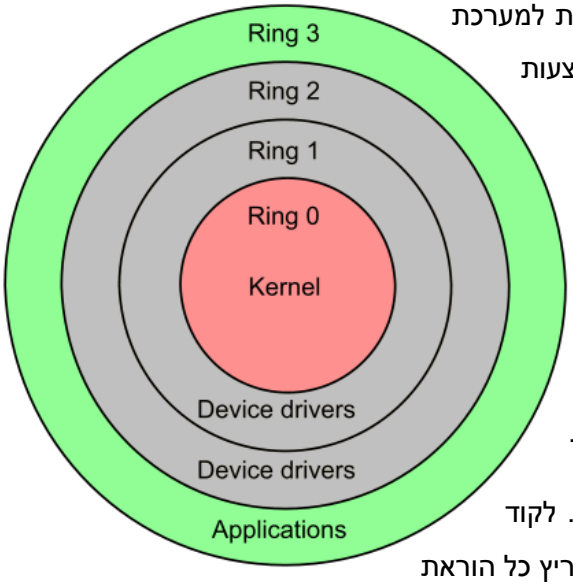
סט ההוראות של x86 הוא רחב מאוד: ישנן הוראות הקשורות להעברת מידע (mov), הוראות אריתמטיות (sub, add, div, mul) ולוגיות (and, or, xor), הוראות זרימה (jmp, jne, je), הוראות שקשורות למחסנית (push, pop) ועוד הרבה. הוראות יכולות לפעול על אפס, אחד, שניים ואף שלושה אופרנדים. יש לציין גם שהגודל של כל הוראה הוא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

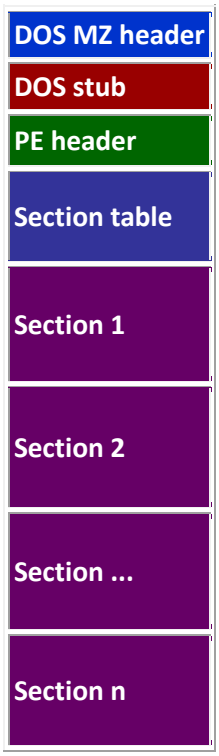
קבצי PE, הרצת קוד ו-Debuggers בווינדוס

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.

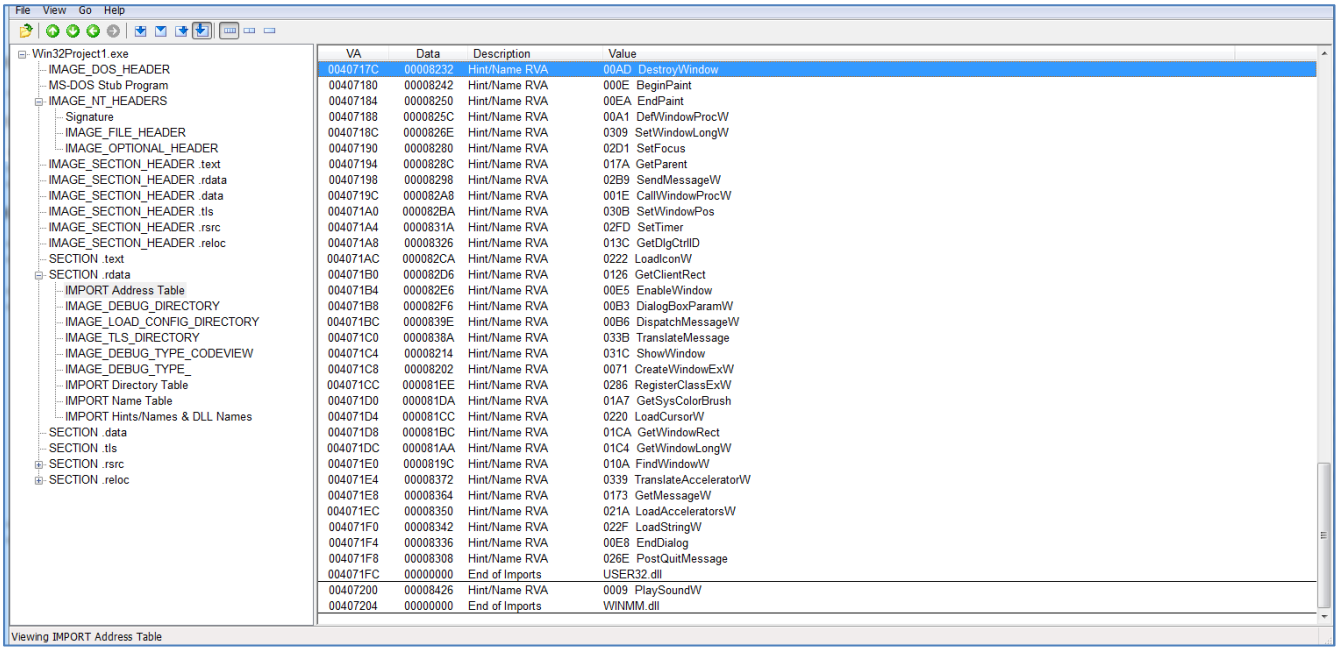
לבסוף נכתבים ה-Section-ים עצמם עם המידע שבתוכם (בסגול), במקומות ובגודל המתאים שלהם כפי שהוגדרו ב-Header.

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.

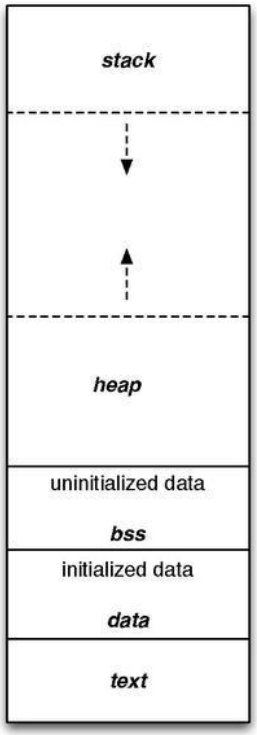
כשתכנית רוצה להשתמש ב-DLL כלשהוא, היא כותבת באזור מיוחד בשם Import Table רשימה של שמות קבצי ה-DLL שהיא צריכה ואת שמות הפונקציות הרלוונטיות. כל פעם שיש קריאה לאחת מהפונקציות האלו, הקומפיילר מכניס הוראה שמבצעת קריאה לכתובת שרשומה בתת-אזור הנקרא Import Address Table (IAT). לפני שהתכנית רצה, הכתובות שרשומות 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.

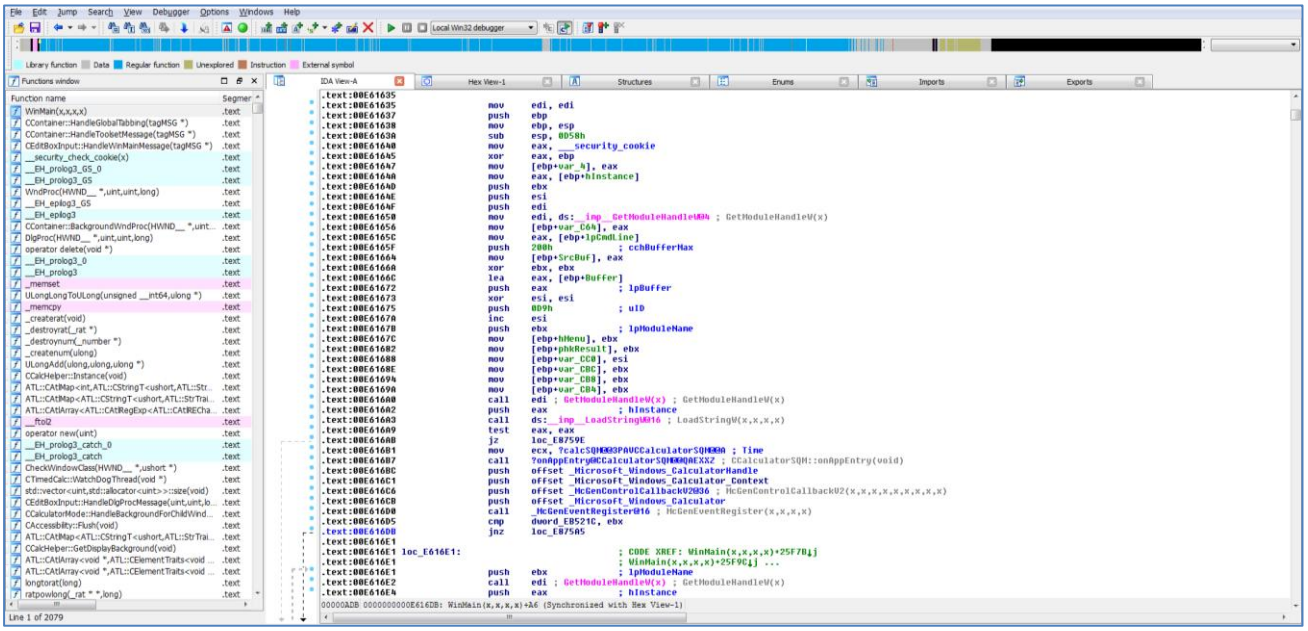
דוגמה ל-Reversing פשוט

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.

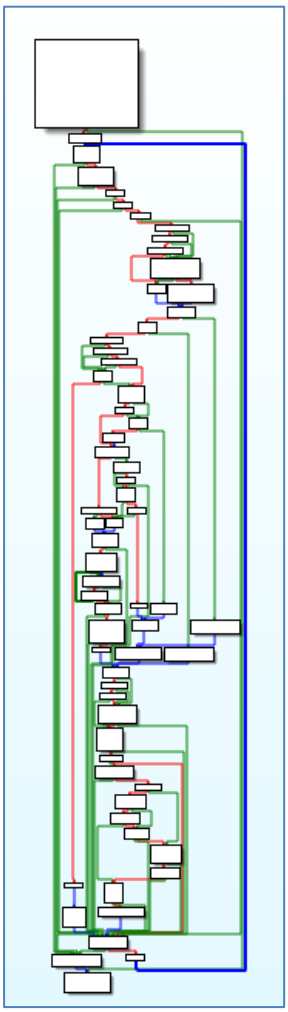
בכל איטרציה, התו הנוכחי מושווה לערכי ה-ASCII של התווים שמסמנים שורה חדשה ('\r', '\n') ורווח (' '). במידה ויש התאמה, יש דילוג לאיטרציה הבאה בלי עיבוד. אפשר להסיק מכך שהמחשבון בעצם מתעלם מכל התווים האלו: אם נדביק לתוכו טקסט כמו "1 2" המחשבון יבין זאת בדיוק באותה צורה אם היינו מדביקים "21". בנוסף, המחשבון 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 ווירוסים 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 תוך EDX: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

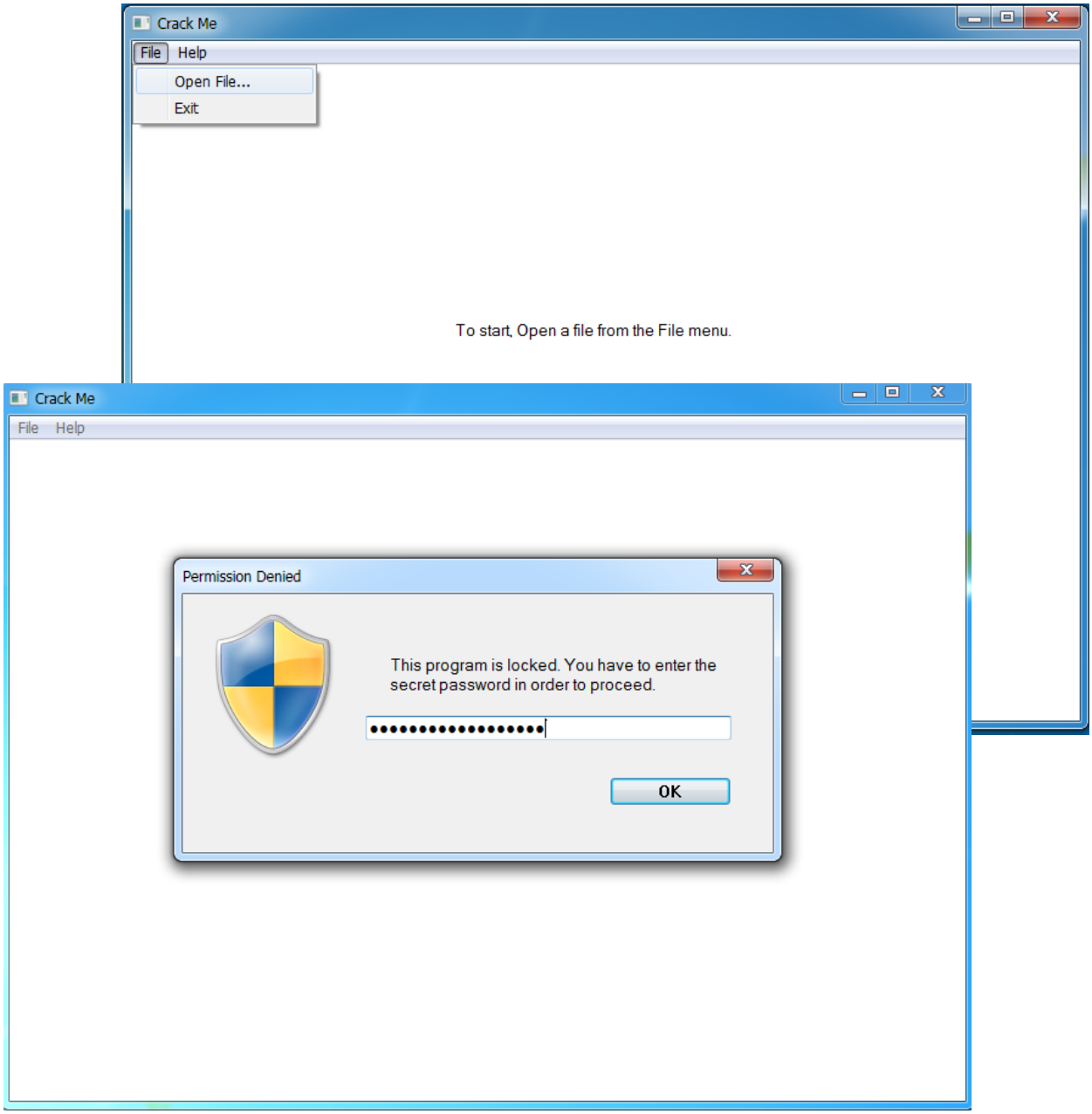
באופן מפתיע, ה-API של ווינדוס כולל את האפשרות לייחס ל-Thread מסוים תכונה שגורמת לו להיות מוחבא מהדיבאגר: הודעות או חריגות מאותו Thread לא יועברו אל הדיבאגר והוא לא יופיע ברשימת ה-Thread-ים הקיימים. באופן כללי אפשר לומר שאותו Thread למעשה לא יהיה קיים מבחינת הדיבאגר. כדי לאפשר את התכונה הזאת נקרא ל-NtSetInformationThread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

מהלך עליית התוכנה

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.

f	enforce_software_breakpoints	.text
f	enforce_software_breakpoints_0	.text
f	enforce_software_breakpoints_1	.text
f	enforce_software_breakpoints_2	.text

[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:

```
static BOOL compare_encrypted_string(char *encrypted_string, char *original_string)
{
    int i= 0;
    int xor = 1;
    while (original_string[i] != '\0' || (encrypted_string[i] ^ xor) != '\0')
    {
        if ((encrypted_string[i] ^ xor) != original_string[i]) return false;
        xor +=2;
        i++;
    }
    return true;
}
```

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.

```
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'^
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
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'^
static char str_ntQueryInformationProcess[] = {'N'^1, 't'^3, 'Q'^5, 'u'^7, 'e'^9, 'r'^11, 'y'^13, 'I'^15, 'n'^17, 'f'^19
static char str_ntSetInformationThread[] = {'N'^1, 't'^3, 'S'^5, 'e'^7, 't'^9, 'I'^11, 'n'^13, 'f'^15, 'o'^17, 'r'^19 ,'
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,
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'
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,
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,

typedef int ( WINAPI *message_box_func )(HWND, LPCWSTR, LPCWSTR, DWORD);
typedef int ( WINAPI *get_window_text_func)(HWND, LPCWSTR, DWORD);
typedef HWND (WINAPI *create_window_func)(LPCTSTR, LPCTSTR, DWORD, int, int, int, int, HWND, HMENU, HINSTANCE, LPVOID);
typedef HANDLE  (WINAPI *get_current_thread_func)();
typedef HMODULE (WINAPI *load_library_func)(LPCWSTR);
typedef int (NTAPI *nt_query_information_process_func)(HANDLE,UINT,PVOID,ULONG,PULONG);
typedef int (NTAPI *nt_set_information_thread_func)(HANDLE,DWORD,PVOID,ULONG);
typedef BOOL (WINAPI *get_thread_context_func)(HANDLE, LPCONTEXT);
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:

```
static HANDLE call_GetCurrentThread()
{
    HMODULE module = call_LoadLibrary(STR_KERNEL32);
    get_current_thread_func f  = (get_current_thread_func)get_proc_address(module, str_getCurrentThread);
    if (f == NULL) { throw new exception(""); return 0; }
    return f();
}
```

[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 }
```

[מתוך breakpoint_detection.h]

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:

```
static void __forceinline small_junk_4()
{
    _asm
    {
        cmp eax, 1
        jne a
a:
        test ebx, [ebp-12]
        je b
        push eax
        mov eax, [ebp-12]
        pop eax
b:
        nop
    }
}
```

```
static void __forceinline junk_3()
{
    wstring a = wstring();
    a.append(junk_var_4);
    if (a.find_first_of('a') != -1)
    {
        int b = junk_func_4();
        junk_var_5 = junk_func_2((float)(b * a[0]));
    }
    int b = 0;
    for (unsigned int i=0;i<a.size();i++)
    {
        b+= a[i];
    }
    BOOL divide = a.size() != 0;
    junk_func_3(divide ? b / a.size() : b,0, &b);
}
```

```
static void __forceinline junk_confuse_2()
{
    wstring str = L"_CRT";
    Sleep(1);
    DWORD a[5];
    for (int i=0;i<5;i++)
    {
        if (i %2 == 0) a[0] = i;
    }
    BOOL sucsess = HeapSetInformation((HANDLE)0x072389C, HEAP_INFORMATION_CLASS::HeapCompatibilityInformation, a, 20);
    TCHAR buffer[256];
    if (junk_var_3 == NULL) a[4] = 0x1000;
    GetComputerName(buffer, a);
    if (GetLastError() != 0 || !sucsess)
    {
        junk_var_3 = (int)OpenThread(NULL, true, 1);
    }
    wstring str2 = buffer;
    if (str2.find(str,0) != wstring::npos && str2.find(L"PC",0) == wstring::npos && str[2] == 'T')
    {
        GetEnvironmentVariable(L"PATH",junk_var_4,256);
    }
}
```

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]

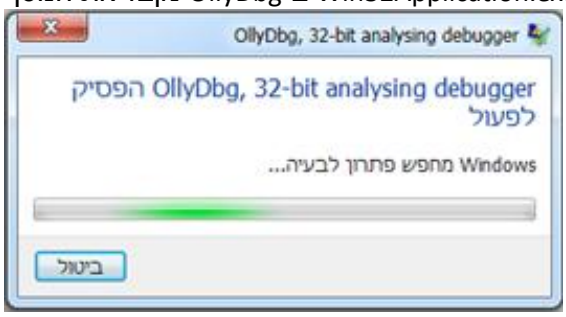
הקרסת תוכנת OLLYDBG

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]

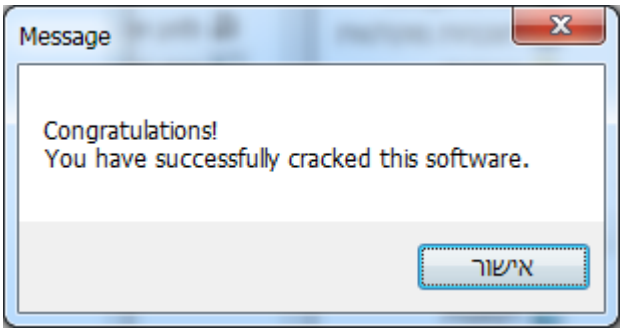
בשלב מסוים בקוד מתבצעת קריאה לפונקציה נוספת: GetSequenceNumber. זוהי פונקצית inline שהתפקיד שלה הוא פשוט לקבל מספר ולגזור ממנו מספר אחר באופן מתמטי **כלשהוא**. הרעיון בשימוש בפונקציה כזו הוא להכניס לקוד הווידוא הסופי אלגוריתמיקה שמשפיעה על התוצאה הסופית ואי אפשר לדלג עליה (כך שהיא נראית מעניינת וקשורה) - 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.

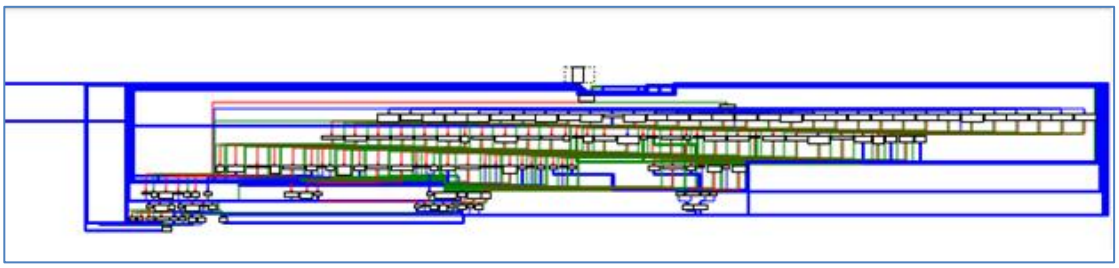
למשל, ישנם פלאגינים ל-IDA שמשתמשים בהזרקת DLL כדי להתגבר על טריקים של Anti Debugging (באמצעות ביצוע Hooking לפונקציות API מסוימות ועוד).

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.

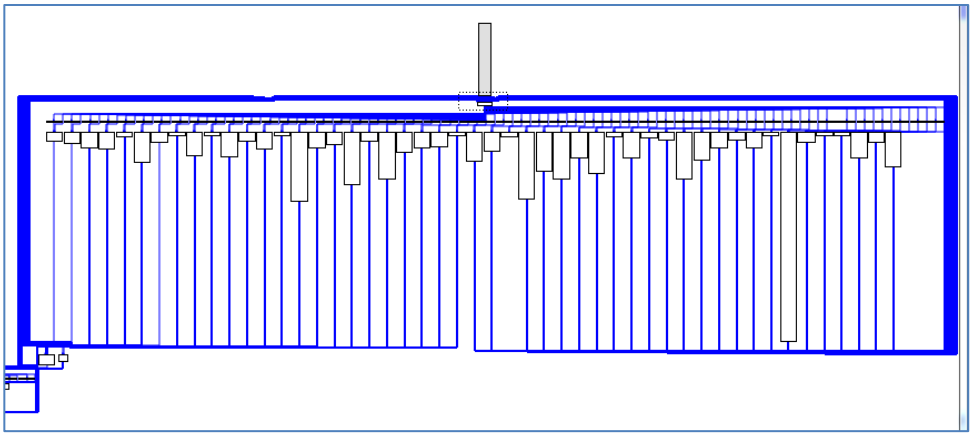


[דוגמה לפונקציה שעורבלה בשיטה זו]

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