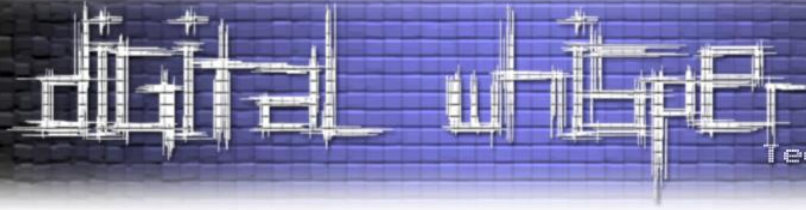


Using Binary Instrumentation to Solve Obfuscated Binary

מאת מאור ראקח

הקדמה

במאמר זה ארצה להציג אתגר CrackMe עם טכניקות אנטי-רברסינג מעניינות ומאתגרות במיוחד ואת התהליך של הפתירה שלו מנקודה שבה אני לא יודע כלום ועד להודעה המיוחלת של ההצלחה.

כלים בשימוש

במאמר זה אשתמש במספר כלים כדי להתמודד עם הבינארי שאותו נחקור. אציג אותם תחילה:

Intel PIN

כלי מבית היוצר של אינטל שמאפשר לנתח תוכנות באופן דינאמי תוך כדי ריצה. הוא מאפשר לנו להזריק קוד מותאם אישית לכל נקודה בתוכנה ודואג לכל הדקויות של שינוי ה-context בין התוכנה שאנו מנתחים לבין הקוד שלנו, מה שאומר שלקוד שלנו יש השפעה אפסית על ריצת התוכנה (מלבד זמן חישוב נוסף) מה שמאפשר לנו להוציא מידע בזמן אמת. בנוסף הכול נעשה עם C++ ככה שלא צריך להתעסק עם אסמבלי ופאטצ'ים למיניהם.

Ghidra

Ghidra היא framework של הנדסה אחורית עם מגוון כלים שמפותח על ידי ה-NSA. כוללת דיאסמבלר ותומכת בכתיבת פלאגינים ב-java וב-python.

סקירה ראשונית

אחרי שמריצים את ה-exe מקבלים ממשק בסיסי של שם משתמש ו-Serial. כאשר נחלץ על מקש ה-"register" ייקח לתוכנה כמה שניות כדי להראות את ה-Message Box המורה על הצלחה או כישלון מה שמצביע על כך שיש סרבול קוד רציני בתוכנה שמאט את התוכנה פי עשרות מונים.

0x4013e0

```
    FUN_004013e0
004013e0    PUSH     EBP
004013e1    MOV      EBP,ESP
004013e3    SUB      ESP,0x328
004013e9    PUSH     EBX
004013ea    MOV      dword ptr [EBP + local_32c ],param_1
004013f0    MOV      dword ptr [EBP + local_110 ],0x3
004013fa    MOV      dword ptr [EBP + local_318 ],0x0
00401404    MOV      byte ptr [EBP + local_109 ],0x0
0040140b    PUSH     0x100
00401410    PUSH     0x0
00401412    LEA      EAX=>local_214 ,[EBP + 0xfffffdf0 ]
00401418    PUSH     EAX
00401419    CALL     FUN_004177fc
0040141e    ADD      ESP,0xc
00401421    PUSH     0x1f
00401423    LEA      param_1 =>local_214 ,[EBP + 0xfffffdf0 ]
00401429    PUSH     param_1
0040142a    MOV      param_1 ,dword ptr [EBP + local_32c ]
00401430    ADD      param_1 ,0x7c
00401433    CALL     GetWindowText_mfc71
00401438    MOV      dword ptr [EBP + local_108 ],EAX
0040143e    PUSH     0x1f
00401440    LEA      EDX=>local_104 ,[EBP + 0xffffff00 ]
00401446    PUSH     EDX
00401447    MOV      param_1 ,dword ptr [EBP + local_32c ]
0040144d    ADD      param_1 ,0xd0
00401453    CALL     GetWindowText_mfc71
00401458    PUSHAD
```

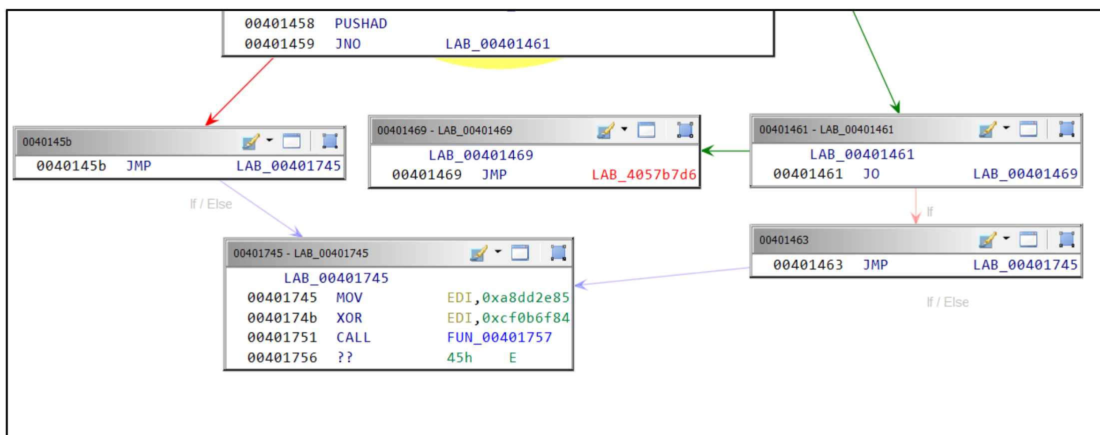
לאחר חקירה ראשונית ניתן לגלות שהפונקציה שבכתובת 0x4013e0 היא הפונקציה שנקראת כאשר מקש ה-"register" נלחץ. הפונקציה היא די נורמטיבית ובסך הכול קוראת את הטקסט מתיבת ה-"name" ומתיבת ה-"serial". לאחר מכן היא מבצעת את הפקודה pushad שפקודה זאת דוחפת את כל האוגרים למחסנית. לאחר הפקודה הפונקציה נהיית מסורבלת ובלתי אפשרית לניתוח סטאטי. יש לציין שלהוראה pushad יש הוראה הופכית בשם popad שהיא קוראת למעשה את מה ש-pushad דחפה למחסנית ומשנה את ערכי האוגרים בהתאם. שיטת obfuscation ידועה היא להכניס קוד זבל בין פקודת pushad לפקודת popad שכן לקוד זבל לא תהיה כלל השפעה על האוגרים משום שגיבוי שלהם נשמר במחסנית. **הפתרון הסופי היה למעשה לחפש קוד אמת אחרי פקודת popad אך הדרך לשם לא הייתה פשוטה.**

המאפיינים של הבינארי

פה אפרט טיפה מהטכניקות שהבינארי משתמש בהם אחרי פקודת ה-PUSHAD על מנת לעשות את החיים של המנתח קשים במיוחד.

קוד זבל

כמויות אינסופיות של קוד מיותר, קפיצות מותנית שמגיעות לאותו המקום כל הזמן.



בתמונה ניתן לראות את הפקודות JNO ו-JO משורשרות יחדיו ומשום שהם הופכיות אם החץ הירוק יוביל ל-JO שבבלוק 0x00401461 אז החץ האדום שיוצא מאותו בלוק בהחלט ירוק, משמע הקוד תמיד יגיע ל-0x401745.

פונקציות שלא חוזרות לקורא

בעת קריאת call הקוד קופץ לכתובת המבוקשת אך גם דוחף את כתובת החזרה למחסנית כך שבראש שבתחילת הקריאה ESP מצביע לכתובת החזרה. קל לראות כיצד הפונקציה 0x401757 לא חוזרת לאותו המקום.

```
00401757 81 04 24       ADD        dword ptr [ESP]=>local_res0,0xfffffd14
         14 fd ff
         ff
0040175e c3             RET
```

Control Flow Changing Exceptions

התוכנה לא קופצת ממקום למקום באמצעות קפיצות וקריאות כמו כל תוכנה נורמטיבית אלא היא מגדירה exception handler משל עצמה ומייצרת exception בכוונה באמצעות opcodes לא ידועים. לפני כל exception שהתוכנה מייצרת, היא שמה באוגר EBP ערך ייחודי שלפיו יודע ה-exception handler לאיזה חלק בתוכנית להעביר את השליטה.

דוגמא ל-Exception Handler

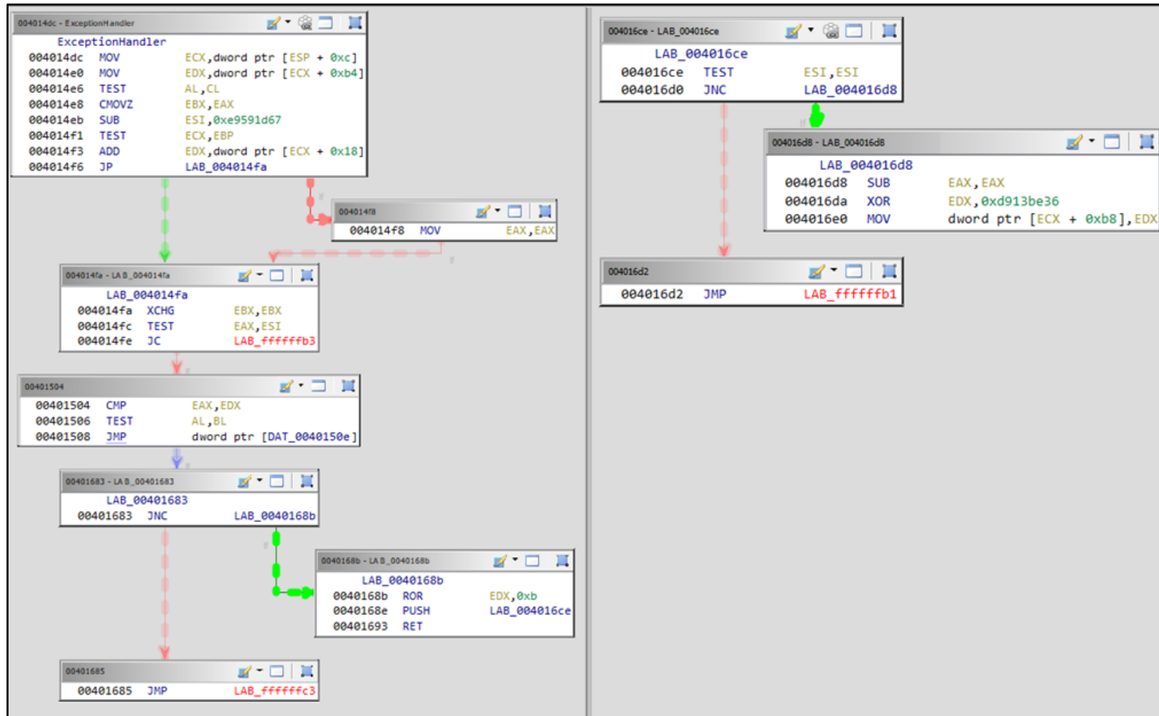
לפונקציה שמוגדרת כ-exception handler יש את המבנה הבא:

```
EXCEPTION_DISPOSITION
__cdecl _except_handler(
    struct _EXCEPTION_RECORD *ExceptionRecord,
    void * EstablisherFrame,
    struct _CONTEXT *ContextRecord,
    void * DispatcherContext
    );
```

```
typedef struct _CONTEXT
{
DWORD ContextFlags;
DWORD Dr0;
DWORD Dr1;
DWORD Dr2;
DWORD Dr3;
DWORD Dr6;
DWORD Dr7;
FLOATING_SAVE_AREA FloatSave;
DWORD SegGs;
DWORD SegFs;
DWORD SegEs;
DWORD SegDs;
DWORD Edi;
DWORD Esi;
DWORD Ebx;
DWORD Edx;
DWORD Ecx;
DWORD Eax;
DWORD Ebp;
DWORD Eip;
DWORD SegCs;
DWORD EFlags;
DWORD Esp;
DWORD SegSs;
} CONTEXT;
```

המבנה ContextRecord מסוג _CONTEXT מכיל בעצם את ה-context של התוכנית בזמן שאותו exception קרה, וכל שינוי שה-exception handler עושה לאותו מבנה יבוא לידי ביטוי שהתוכנה תחזור לרוץ. אותו מבנה משמש בעצם ב-exception handlers על מנת שאלו יוכלו לתקן את התוכנה ולהשיב אותה למצב שתוכל להמשיך לרוץ. עם זאת בבינארי הזה ה-exception handler דואג

להעביר את הריצה לכתובת אחרת וזאת באמצעות שינוי אוגר EIP על פי הערך שמופיע ב-EBP.

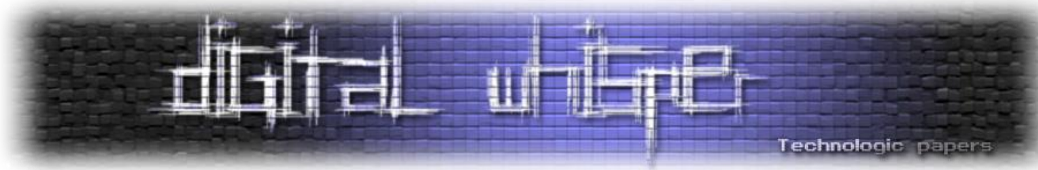


ExceptionHandler

ה-Handler טוען את המצביע למבנה ה-CONTEXT_ אל תוך אוגר ECX ולאחר מכן שולף את הערך של EBP (אופסט 0xb4 ב-ContextRecord) בזמן שה-exception קרה. את אותו ערך הוא מתמרן עם פקודת ROR ו-XOR ולבסוף כותב את התוצאה אל תוך EIP (אופסט 0xb8 ב-ContextRecord). EIP הוא אוגר שמצביע לכתובת שרצה בכל זמן נתון ולכן כאשר התוכנה תחזור מאותו exception היא תחזור למקום אחר. קל להבחין שגם ה-Handler מכיל טכניקות של אנטי-רברסינג.

ניסיון ראשון להגיע לפתרון

לאחר שקיבלתי חלק מהתמונה על אותו בינארי חשבתי לעצמי שאוכל לנצל את הטכניקה הראשית שלו להקשות על רברסינג - השימוש ב-exceptions. הרעיון שלי היה פשוט לעשות לשנות את הפונקציה - 0x4014dc (ה-exception handler הראשון) ככה שכל הכתובות שהיא משנה את הריצה אליהם יפלטו לקובץ ככה אוכל להבין את ההיקף של הקוד ולקבל תמונה ראשונית.



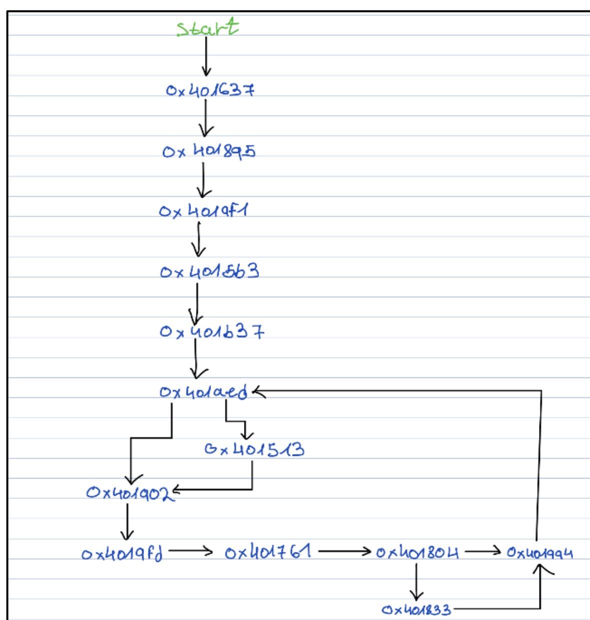
בנוסף אוכל לגלות כתובת חדשות ולייבא אותם אל תוך Ghidra ואולי אוכל לפתור את זה עם רברסינג סטאטי:

```
<004017AA>
    push 0x41c000
<0041C400>
    push @perm
    push @filename
    call fopen
    mov [0x0041A720], eax
    add esp, 0x8
    jmp 0x0041785C
<0041c000>
    pushad

    mov esi, eax

    mov ecx, [esp + 0x2c]
    mov edx, [ecx + 0xb4]
    ; the address of ebp inside the excpetion handler
    mov ebx, [ecx + 0xb8]
    ; the address of eip inside the excpetion handler (the new target)
    push ebx
    ror edx, 0xb
    xor edx, 0xd913be36


    push edx
    push @kep
    push [0x0041A720]
    call fprintf
    add esp, 0x10

    popad
    push 0x4014dc
    ret


    @perm:
    "a\0"

    @filename:
    "jump_dump.txt\0"

    @kep:
    "EBP: %.8x EIP: %.8x\n\0"
```

```
jump_dump.txt
(not all lines shown here)
EBP: 00401637 EIP: 004015ae
EBP: 00401895 EIP: 00401649
EBP: 004019f1 EIP: 0040196f
EBP: 004015b3 EIP: 004019f8
EBP: 00401b37 EIP: 004015c5
EBP: 00401aed EIP: 0040159d
EBP: 00401513 EIP: 00401b0d
EBP: 00401902 EIP: 00401890
EBP: 004019fd EIP: 0040197a
EBP: 00401761 EIP: 004017ff
EBP: 00401804 EIP: 004018d0
EBP: 00401994 EIP: 00401ace

EBP: 00401aed EIP: 0040159d
EBP: 00401513 EIP: 00401b0d
EBP: 00401902 EIP: 00401890
EBP: 004019fd EIP: 0040197a
EBP: 00401761 EIP: 004017ff
EBP: 00401804 EIP: 004018d0
EBP: 00401833 EIP: 0040167e
EBP: 00401994 EIP: 00401ace
EBP: 00401aed EIP: 0040159d
EBP: 00401902 EIP: 00401890
EBP: 004019fd EIP: 0040197a
EBP: 00401761 EIP: 004017ff
EBP: 00401804 EIP: 004018d0
EBP: 00401994 EIP: 00401ace
```

בשמאל ניתן לראות את ה-patch שלי. אני טוען את מבנה ה-CONTEXT_ ומחשב את ערך ה-EIP על פי ערך ה-EBP בדיוק כמו ה-Handler המקורי. לבסוף הכל נכתב לקובץ jump_dump.txt שחלק ממנו מופיע בצד ימין. לאחר מכן אנו מחזירים את השליטה ל-0x4014dc כדי שהקוד ימשיך לרוץ כרגיל.

דבר אחד ששמתי אליו שניתחתי את jump_dump.txt הוא שהכתובות חוזרות על עצמן, משמע שיש פה לולאה, משום שהרגשתי קרוב לפתרון החלטתי לנתח את הלולאה בצורה ידנית. שאחרי זה אקבל תמונה מלאה וזה יוביל אותי לפתרון. בגלל שאני יודע את הסדר שהכתובות מופיעות אני יכול להבין איזה כתובות זורמות לאיזו כתובות ו"לצייר" את הלולאה.



בצד שמאל ניתן לראות את הפענוח של הלולאה. מיותר לציין שכל בלוק היו מסורבל בכלל טכניקות אנטי-רברסינג אבל למרות זאת התמדתי כי האמנתי שהפתרון כולו טמון באותה לולאה.

לבסוף גיליתי שאותה לולאה בסה"כ מפענחת מתוך הזיכרון לולאה חדשה, שאותה לולאה מפענחת עוד לולאה, שמצפינה חזרה לולאה קודמת וכך הלאה. לכל לולאה exception handler חדש שפועל אחרת. מעל 20 לולאות כאלו ומעל אלפי exceptions שמחברות את כול התוכנה. זה הזמן להפעיל את התותחים הכבדים.

Writing Custom PinTool

אם כל לולאה כזאתי שמורכבת מ-exceptions מפענחת מהזיכרון עוד לולאה כזאת שמפענחת עוד אחת כזו..., ובנוסף על כן, כל פעם מוגדר exception handler שונה לגמרי. אני אצטרך להשתמש באוטומציה שכן בצורה ידנית הסיכוי שלי להגיע למסקנה בעלת ערך היא אפסית. החלטתי לכתוב כלי על בסיס Intel PIN שירוץ בזמן אמת לצד התוכנית ויאסוף ויעבד את הנתונים. אני לא אפרט על הכלי יותר מידי אבל כן אצרף קוד מקור למי שיהיה מעוניין, וכן אצרף חלק קטן מהפלט שלו ואדון על שיקולים שהיו לי בבנייתו.

KiUserExceptionDispatcher & NtContinue

אם מקודם השתמשתי ב-exception handler יחיד שידעתי על קיומו, הרי שעכשיו עליי למצוא דרך אחרת למצוא את ה-exceptions שהתוכנה מייצרת ואת המקומות שאליהם הם מעבירים את השליטה. למזלי מצאתי את הפונקציה KiUserExceptionDispatcher שנמצאת ב-ntdll.dll ובעצם מקבלת את השליטה אליה בכל פעם ש-exception קורה ותפקידה כשמשתמע משמה הוא להריץ את ה-handler הרלוונטי.

לאחר מכן היא משתמשת בפונקציה NtContinue כדי להחזיר את השליטה לתוכנית. בעזרת Intel PIN יכולתי להתלבש (hook) על הפונקציה KiUserExceptionDispatcher ולגלות כל exception שקורה בתוכנה, וכשהתלבשתי על NtContinue יכולתי לגלות כל מקום חדש שהתוכנה חוזרת אליו.

בדיוק כמו ה-patch הספציפי שעשיתי רק בהיקף גדול הרבה יותר.

MemoryChangeCheck

התוכנה מחוברת כולה בעשרות אלפי exceptions שונים, לכן לאחר שהשגתי שליטה מוחלטת בעצם על כל exception שכזה וכל מקום חדש שהתוכנה מגיעה אליו זה היה הזמן לסרוק שינויים בזיכרון כדי להבין את ההיקף של מה שאני מתמודד מולו. החלטתי להניח שכל כתובת בזיכרון משתנה פעם אחת ולכן לכל כתובת יש למעשה שני "גרסאות".

הגרסה הראשונה היא למעשה הבייט המקורי שמופיע במקור והבייט שמופיע לאחר שקוד הלולאה מוצפן בחזרה, והגרסה השנייה זה הבייט המפוענח.

כתבתי פונקציה ששמה MemoryChangeCheck שרצה בין לולאה ללולאה ושומרת את הגרסה השנייה של כל byte. באמצעות זאת יכולתי ליצור העתק של כל section בתוכנית כאשר הוא מפוענח לגמרי שכן במקום בייתים מקודדים בלי ערך, היה לי את הערך שלהם בזמן שהם מורצים בזיכרון.

על ההעתק של section זה יכולתי לעשות רברסינג סטאטי שכן כל הקוד הופיע **בעת ובעונה אחת**, לעומת ריצה של התוכנית שבה מקטע קוד מוצפן בחזרה לאחר שאין בו כל שימוש.

```
void MemoryChangeCheck(void * & min, void * & max) {
    std::set<VOID*> changedSet;
    BOOL hasChanged = FALSE;

    // עבור כל סקשיין בתוכנית
    for (auto sec = Sections.begin(); sec != Sections.end(); sec++) {
        Section & currentSection = sec->second;
        char * currentData = (char*)currentSection.start_address;

        for (int i = 0; i < currentSection.SecSize; i++) {
            // אם הערך הנוכחי לא שווה לערך המקורי משמע יש פה שינוי של הקוד
            if (*currentData != currentSection.saved[i]) {

                changedSet.insert(currentData);
                hasChanged = TRUE;
                currentSection.saved[i] = *currentData;

                // אני מעוניין רק בשינוי הראשון ולכן אני בודק אם זה השינוי הראשון
                if (!currentSection.hasEverChanged[i]) {
                    currentSection.dumped[i] = *currentData;
                    currentSection.hasEverChanged[i] = TRUE;
                    if (*currentData == 0x61) {
                        *out << (void*)currentData << ",";
                    }
                }
            }

            ++currentData;
        }
    }

    min = *changedSet.begin();
    max = *changedSet.rbegin();
}
```

חדי עין ישימו לב במשפט התנאי האחרון אני בעצם בודק אם הבית חדש שווה ל-0x61 ואם כן אני שומר את הכתובת שלו בקובץ. 0x61 זה ה-opcode של פקודת ה-popad. אותה פקודה שאחריה האמנתי שיהיה קוד אמת שכן כל סרבול הקוד מתחיל אחרי פקודת pushad

פלט סופי של ה-PinTool

שלב ראשון היה בעצם להשיג שליטה על exception שקורה וכל מקום חדש שהתוכנה מגיעה אליו. שלב שני היה לסרוק שינויים בזיכרון. שלב שלישי היה לשלב את כל המידע הזה לכלי שידע לזהות את הלולאות, ואיזה לולאות מפענחות את הקוד של איזו לולאות.

זיהוי לולאות

ה-PinTool ידע לקחת את כל הכתובות החדשות ולעבד אותם ככה שהוא ידע לזהות לולאות בצורה אוטומטית. בנוסף, בגלל שהקוד גם סרק שינויים בזיכרון הוא ידע להגיד גם איזה לולאה משפיע על קוד של לולאה אחרת.

וכך זה נראה:

```
Loop B:
Entry 0x00401b95:
Exit 0x00401e90:
0x00401b95(1); 0x00401c2f
0x00401bc2(2277); 0x004020a4
0x00401bd8(4506); 0x0040200c 0x004020f1
0x00401c0f(2308); 0x0040211a
0x00401c2f(1); 0x004020d5
0x00401d72(4506); 0x00401f17 0x00402154
0x00401d8e(1059); 0x00402230
0x00401dd0(4506); 0x00401d72 0x00401d8e 0x00402230
0x00401f17(2293); 0x00402154
0x0040200c(4506); 0x00401bc2 0x004020a4
0x004020a4(4506); 0x00402213
0x004020c0(4506); 0x00401c0f 0x0040211a
0x004020d5(1); 0x00401d72
0x004020f1(2259); 0x0040200c
0x0040211a(4506); 0x00401dd0
0x00402154(4506); 0x00401bd8
0x00402213(4506); 0x004020c0
0x00402230(2198); 0x00401d72 0x00401e90

Memory Changes; Min: 0x004022c2; Max: 0x00403458
Loop Affected: C D E
```

בקטע מתוך הפלט, מצוין שהכלי מצא לולאה בשם Loop B שהבלוק הראשון (בלוק הכניסה) שלה הוא 0x00401b95 ובלוק היציאה שלה הוא 0x00401e90. כל שורה מציינת כתובת שהיא אחת מהבלוקים שמרכיבים את הלולאה. בסוגריים זה מספר הפעמים שהבלוק הזה רץ. ואחרי ה; מצוין לאיזה כתובות אותה כתובת "זורמת". ובחלק האחרון מצוין הכתובת המינימלית והמקסימלית שאותה לולאה משנה בזיכרון. בחלק האחרון מצוין את הקוד של אילו לולאות הלולאה משנה.

מי שמעוניין בפלט המלא של הכלי יכול לצפות בו כאן:

https://pastebin.pl/view/5b95c10c

POPAD Gadgets

בשלב זה היה לי העתקים של כל section - text, data, rdata מפוענחים. והיה לי רשימה של כל כתובת בזיכרון שמכילה את הבית 0x61 (פקודת popad). יש לשים לב שלא כל כתובת כזאת היא בהכרח שימושית שכן הבית 0x61 יכול להופיע כחלק מפקודה אחרת ולא בהכרח כפקודה העומדת בפני עצמה. כל מה שהיה נותר לי לעשות פה זה לעבור כתובת כתובת, למצוא קוד תקין ולחבר את כל הקוד בצורה שתינתן לי האפשרות לקרוא אותו.

סיכום

ללא ספק זאת הייתה תוכנה שסרבלה את הקוד בצורה כזו מתוחכמת ולכן בבואנו לקרוא קוד שכזה בלתי אפשרי לעשות זאת בצורה ידנית - עלינו לכתוב תוכנה שמתמודדת עם אותו סרבול. אני חושב שכבר היום פלאגינים הם חלק בלתי נפרד מתפקיד הרברסר, והשימוש באוטומציה בהנדסה הפוכה פותח דלתות חדשות.

אם נהנתם מהמאמר, אולי תהנו מפוסטים נוספים בבלוג שלי:

https://www.rakach.com/blog