digital whisper
Technologic Papers

המדריך המהיר לכתיבה של וירוס פשוט

מאת דן בומגרד (Dan Bomgard)

תוכן העניינים

הקדמה

מאמר זה יתאר קוד אשר מיישם אפליקציה של וירוס בסיסי בסביבת Windows אשר רץ על מעבד בארכיטקטורה של x86. במאמר אני מתאר את תהליך תכנון הלוגיקה, כתיבת הקוד, והשיקולים שעמדו בפני בשלבים שונים בפרויקט. הקוד עצמו, מצורף בנפרד מהמאמר וניתן לקחת אותו כמו שהוא ולקמפל אותו בעזרת הצעדים המובאים בכתבה לגרסא פועלת של הוירוס.

כותב המאמר אינו מתיימר להיות מומחה או סמכות באף אחד מהתחומים עליהם מפורט במאמר וחלקים מהקוד נלקחו מפרויקטים שונים הקיימים ברשת, חלקם פרויקטי קוד-פתוח כאלו ואחרים וחלקם פרויקטים אשר פורסמו תחת רשיון של Creative Commons אשר מתיר שימוש בקוד המפורסם כל עוד מפורסם לידו קישור למאמר המקורי (ולכן דאגתי להזכיר את כל אותם פרויקטים בתחילת הקוד עצמו וגם בבבליוגרפיה).

את המאמר הזה והקוד המצורף אליו אני מפרסם כאן ללא רשיון נלווה ולא לוקח אחריות על שימוש לרעה שנעשה בקוד המצורף. כן אשמח לשמוע מאנשים שעשו שימוש בקוד או שיש להם רעיון מקורי לשיפור של הלוגיקה המובאת פה או לשיתוף פעולה כלשהו (מוזמנים ליצור קשר ב-danb33@gmail.com).

הקוד נכתב כפרויקט אישי ולכן אין שמירה על קונוונציות כתיבה כאלו ואחרות, מאותן סיבות גם אומר פה שהקוד מהווה פרויקט מתמשך שלי והקוד המצורף למאמר אינו "סופי" בשום צורה, הוא מכיל הרבה קטעים לא יעילים וישנם הרבה שיפורים פוטנציאליים שגם מתכנת בינוני יבחין בהם, אך מכיוון שמדובר בפרויקט שאני כותב בזמני הפנוי יש לי את הפריווילגיה להתרכז רק בנושאים שמעניינים אותי ולא להשקיע בדברים אחרים.

מאמר זה דורש קצת ידע מקדים בתחומים הבאים: שפת C, שפת אסמבלי של x86, מבנה של קבצי תוכניות של Windows. במאמר זה אתאר את המנגנונים השונים והאלגוריתם שבקוד המצורף, הסבר ברמת השורה או הפעולה הבודדת נמצא בקוד עצמו בהערות שכתבתי ובמאמר לא ארד לרזולוציה גבוהה מאוד של הסברים.

דרישות מהתוכנית

הדרישות מהתוכנית הן להלן:

- הוירוס צריך לרוץ בצורה כזו שירוץ לפני התוכנית אליה הוא נדבק, יבצע את פעולתו (אם בהצלחה ואם לא) ולאחר מכן יריץ את התוכנית עצמה בצורה שקופה למשתמש.
- במידה והוא מזהה קבצים "נקיים" הוא ינסה להדביק אותם ובמידה והוא מזהה קבצים שכבר נדבקו הוא לא יעשה כלום.
- כדי לא לאפשר יישום קל מדי של הוירוס למטרות רעות וגם בגלל שזה מקל על הפיתוח, הוירוס ינסה להדביק רק קבצים אשר נמצאים בתקיית C:\Virus ולא בשום מקום אחר במחשב. (עדין מומלץ להריץ אותו רק על VM מבודד בלבד).
- תהליך ההדבקה יהיה כזה שלא יהרוס את התוכן המקורי של התוכנית אותה הוא מדביק אלא רק "יתווסף" אליה.
- הוירוס שלנו לא יבצע פעולה דונית חוץ מפעולת ההדבקה עצמה וההוכחה להדבקה תהיה חתימה שתבוצע במקום מוגדר מראש בקובץ.

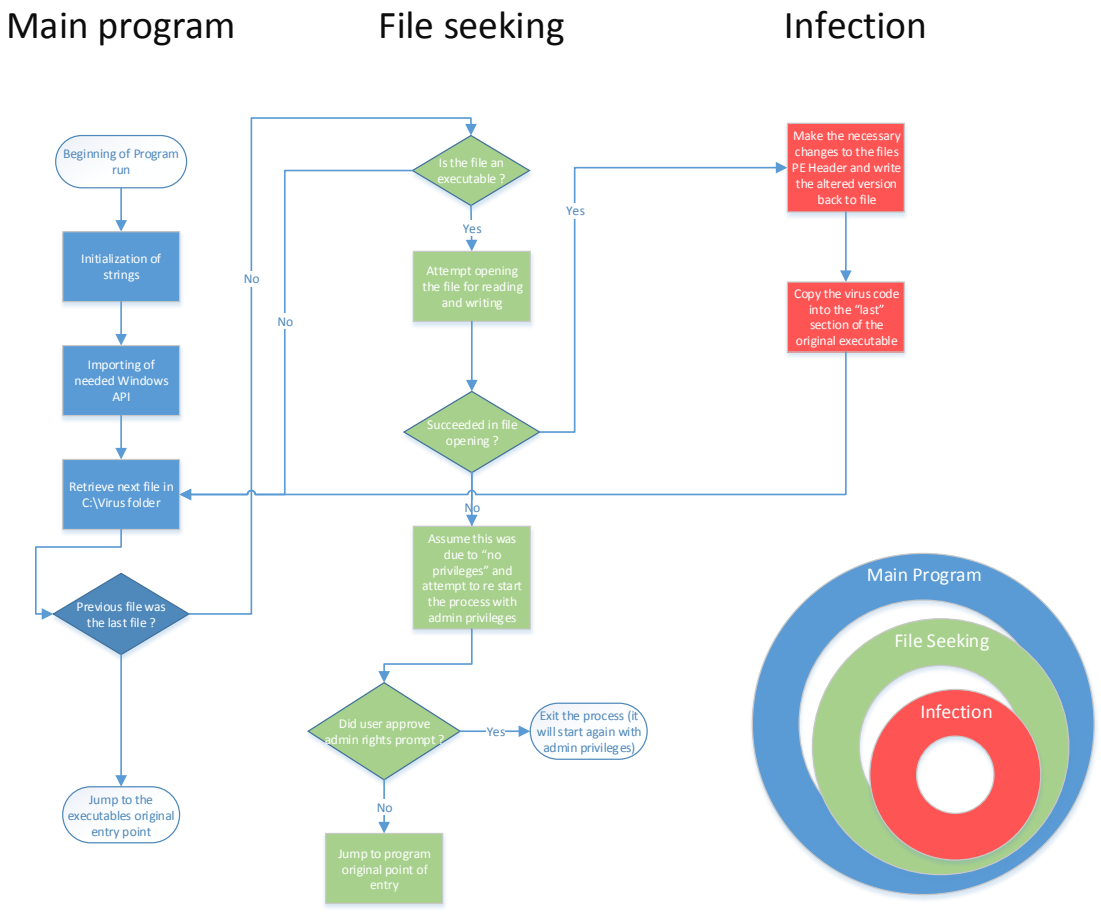
כלים

הכלים בהם עשיתי שימוש בביצוע הפרויקט:

- Microsoft Visual Studio 2013 - ניתן להשתמש בגרסה חינמית שקיימת באינטרנט והיא די נוחה לשימוש בתור סביבת פיתוח וDebugger.
- OllyDbg - לדעתי ה-Debugger הנוח ביותר שקיים לאפליקציות 32bit בסביבת Windows שמשלב ממשק ויזואלי נוח ביותר ויכולת מניפולציה של הקוד שרץ בזמן אמת ושמירה של השינויים לקובץ המורץ.
- PEView - תוכנה להצגת השדות וצפיה נוחה בתוכן של קובץ PE (קובץ EXE).
- HexEditor - כלי הכרחי לפיתוח או עבודה בכל פרויקט שבמרכזו התעסקות עם קוד ב-Low Level, ישנם הרבה עורכים שונים ברשת עם יכולות נוספות שונות אבל היכולת המרכזית היא היכולת לצפות בתוכן הקובץ ולערוך אותו והיא קיימת בכל עורך כזה. אני משתמש ב-HexEdit.

חיפוש מהיר בגוגל של שמות הכלים יחזיר אפשרות נוחה מאוד להוריד אותם בקלות.

תכנון כללי (High Level Design)



חלק ראשון - התוכנית המרכזית

מציאת ה-API של מערכת ההפעלה בזיכרון

בשביל לבצע כל פעולה משמעותית במערכת, כל קוד חייב לדעת לגשת ל-API שמספקת לו מערכת ההפעלה. ניתן להזריק לתוכנית רצה פעולות אסמבלי מכאן ועד הודעה חדשה ולחשב את הערך של פאי 20 ספרות אחרי הנקודה העשרונית, אבל אם רוצים לשמור את הערך הזה לדיסק הקשיח, לשלוח אותו לצד השני של העולם דרך האינטרנט או פשוט למדפסת, חייבים לדעת איפה מיקמה מערכת ההפעלה בזיכרון את רצף ההוראות שמבצע את הפעולה המבוקשת.

לא נרד במאמר זה לעומק הארכיטקטורה של מערכת ההפעלה Windows אבל אסביר לגבי המנגנון הבסיסי של קריאה לפונקציות מערכת ב- Windows. באופן כללי, כל הקוד שמבצע פעולות מערכת (לדוג' כותב לזיכרון שממופה לכרטיס רשת על מנת לשלוח בתים לרשת או כותב לזיכרון שממופה לדיסק

הקשיח על מנת לכתוב אליו) נמצא ב- Kernel, אזור זיכרון זה אינו נגיש לתוכנית רגילה אשר רצה מה-Userspace מפאת הארכיטקטורה של מעבדים חדשים. זה נכון בפרט לגבי משפחת x86 (בדורותיה האחרונים בלבד, ממש לא מהדורות הראשונים) אשר רובנו עושים בהם שימוש במחשבנו הביתיים. הפונקציות שכן ניגשות לאותו אזור בזיכרון (Kernel Space) הן הפונקציות שנמצאות בספריות של ה-API של מערכת ההפעלה והן עושות זאת באמצעות שימוש בפקודות מאוד מסוימות.

אם זהו המצב, ברגע שנדע היכן בזיכרון ממוקמות הפונקציות של ה-API של מערכת ההפעלה נוכל לבצע פעולות משמעותיות במערכת. אז איך עושים את זה ?! זהו בדרך כלל האתגר הראשוני של קוד-זדוני לסוגיו וניתן למצוא את אותן פונקציות שמערכת ההפעלה מנגישה אם מכירים קצת את המבנה של מערכת ההפעלה וכיצד היא עושה שימוש במעבד מסוג x86.

ישנן מספר טכניקות למציאת ה-API של מערכת ההפעלה Windows. רובן מפורטות במאמר די מוכר וישן יחסית אך מוסבר היטב (Skape/Matt Miller's win32 shellcode tutorial). המאמר עצמו ישן אך עדין רלוונטי והקוד שבו דורש שינויים לא גדולים על מנת להתאימו לפעולה של Windows 7/8 (המאמר המקורי עובד על XP וגרסאות קודמות).

בוירוס שלי אני עושה שימוש בטכניקה אחת אשר מתבסס על מציאת מבנה נתונים בשם PEB (Process Environment Block) אשר משמש את מערכת ההפעלה לצורך ניהול של התוכנית הרצה, הוא מכיל מידע "מנהלתי" מגוון לגבי התוכנית כמו למשל האם היא נפתחה דרך Debugger או לא (אחת הדרכים של קוד-זדוני לדעת אם מנסים למצוא אותו היא חיפוש הערך של פרמטר זה ב-PEB טרם פעולתו) או מידע שונה לצורך ניהול ה-Heap של התוכנית.

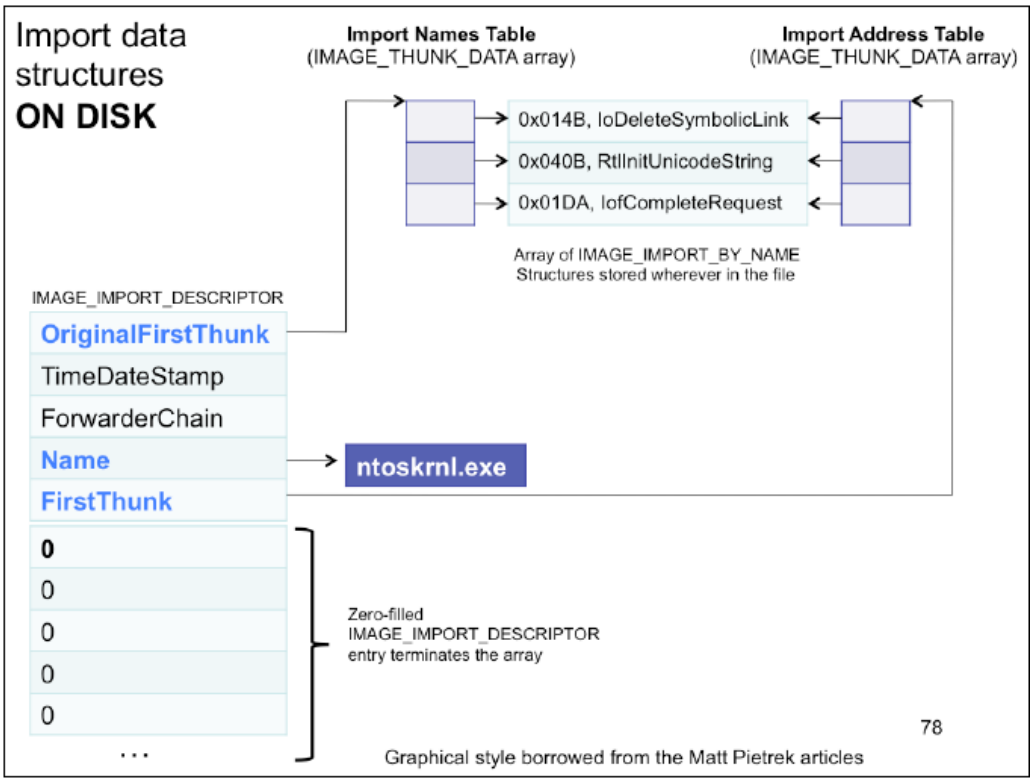
ב-PEB קיים פוינטר למבנה נתונים בשם PEB_LDR_DATA אשר מכיל מידע לגבי מיקום טעינת קבצי ה-DLL של Windows אשר מכילים את ה-API של מערכת ההפעלה. יש כאן הנחה של סדר טעינת ה-DLLים, אנו מניחים ש-Kernel32.dll אשר מכיל ה-API הבסיסי ביותר של המערכת בו נעשה שימוש בקוד זה הוא זה שנטען שלישי, זה תמיד נכון ממערכת Windows 7 ומעלה, בגרסאות קודמות יותר של מערכת ההפעלה קובץ DLL זה היה נטען שני אך תכולתו פוצלה וכיום API בסיסי נוסף של מערכת ההפעלה קיים בקובץ KernelBase.dll אשר נטען שני. זו הסיבה שבגרסאות ישנות של Shellcode ישנה פניה לערך שני ברשימה המקושרת של הספריות הטעונות (כמו שניתן לראות במאמר המקורי) אך בגרסאות חדשות יותר (כמו זו בה נעשה שימוש בקוד שלנו) ישנה פניה לאיבר השלישי ברשימה זו. החלק בקוד שמבצע את מציאת האזור בו נטענה הספריה kernel32.dll לזיכרון הוא תחת התגית find_kernel32_base.

לאחר שמצאנו את האזור אליו נטענה הספריה עצמה צריך למצוא את מיקום הפונקציות בהן אנו רוצים להשתמש. גם כאן ישנן מספר דרכים אפשריות לביצוע הפעולה ובכמה מהמקורות בהם השתמשתי ואשר מובאים בביבליוגרפיה מיושמים שיטות שונות. ניתן לדעת למשל את ה-Offset אליו נטענה כל אחת

מהפונקציות יחסית לבסיס של הספריה (אותו מצאנו בהתחלה). היתרון בשיטה זו הוא פשטות של הקוד שבאה לידי ביטוי גם בקוד פחות גדול (יתרון גדול בקוד-זדוני), ישנו גם יתרון של זמן ריצה אך עניין זה יחסית זניח כשמדובר בקוד-זדוני שרץ על מעבדים ביתיים.

בקוד שלנו, אני עושה שימוש בשיטה אשר מתבסס על מציאת פונקציה כלשהי ב-API לפי ה-HASH של אותה פונקציה. באופן כללי שמות הפונקציות וערכי ה-HASH של הפונקציות מאוכסנים בתוך קובץ ה-DLL אשר מספק אותן ונטען לזיכרון בתוך רשימות. גם כתובות התחלת כל פונקציה ופונקציה שמורות ברשימה אשר מתאימה לחלוטין (מבחינת סדר) לרשימות ערכי ה-HASH ושמות הפונקציות. ה-HASH של כל פונקציה בקובץ מחושב לפי שם הפונקציה אשר לא (אמור) להשתנות גם אם סדר הפונקציות בתוך הקובץ משתנה. לצורך מציאת פונקציה ב-DLL לפי שיטה זו, נחשב את ערך ה-HASH של הפונקציה שאנו מעוניינים למצוא ואז נסרוק את רשימת ה-HASHים ב-DLL כדי למצוא שם את ה-HASH המבוקש, מספרו של ערך ה-HASH ברשימה זו הוא גם מספר הערך של כתובת תחילת הפונקציה הרצויה ברשימת כתובות ההתחלה.

באיור הבא ניתן לראות נסיון להמחשה של הרשימות שהוזכרו בקובץ ntoskernel.exe אשר מהווה חלק מה-Kernel אך חושף ספרית פונקציות בדיוק כמו כל קובץ DLL.



[האיור לקוח מחומר הלימוד של הקורס המומלץ בחום Life of Binaries אשר ניתן בחינם באתר של OpenSecurity.org]

```
Address   Hex dump                                          ASCII
750EC494  33 32 2E 64 6C 6C 00 42 61 73 65 54 68 72 65 61  32.dll BaseThrea
750EC4A4  64 49 6E 69 74 54 68 75 6E 6B 00 49 6E 74 65 72  dInitThunk Inter
750EC4B4  6C 6F 63 6B 65 64 50 75 73 68 4C 69 73 74 53 4C  lockedPushListSL
750EC4C4  69 73 74 00 4E 54 44 4C 4C 2E 52 74 6C 49 6E 74  ist NTDLL.RtlInt
750EC4D4  65 72 6C 6F 63 6B 65 64 50 75 73 68 4C 69 73 74  erlockedPushList
750EC4E4  53 4C 69 73 74 00 41 63 71 75 69 72 65 53 52 57  SList AcquireSRW
750EC4F4  4C 6F 63 6B 45 78 63 6C 75 73 69 76 65 00 4E 54  LockExclusive NT
750EC504  44 4C 4C 2E 52 74 6C 41 63 71 75 69 72 65 53 52  DLL.RtlAcquireSR
```

[תחילתה של רשימת שמות הפונקציות בספריה Kernel32.dll]

```
Address   Hex dump                                          ASCII
750E87F0  34 A0 0E 00 70 B8 0E 00 91 91 01 00 C8 C4 0E 00  4áÅ pçÅ ææ0 ╚─Å
750E8800  02 C5 0E 00 38 C5 0E 00 A4 A7 01 00 8D 99 01 00  ☻┼Å 8┼Å ñº0 ìö0
750E8810  AA 5B 02 00 ED 19 01 00 F4 69 06 00 3D 6B 06 00  ¬[☻ Φ↓0 ⌠i♠ =k♠
750E8820  BE C5 0E 00 E6 3F 04 00 8C 9D 03 00 D4 9D 03 00  ╛┼Å µ?♦ î¥♥ ╘¥♥
750E8830  88 2D 04 00 EB C2 01 00 93 2D 04 00 E7 D1 01 00  ê─♦ ôΓ0 ô─♦ τ╤0
750E8840  A4 2D 04 00 C4 F5 03 00 F7 C6 0E 00 37 C7 0E 00  ñ─♦ ─J♥ ≈╞Å 7╟Å
750E8850  AF 4F 05 00 70 77 02 00 C6 2D 04 00 B5 2D 04 00  »O♣ pw☻ ╞─♦ ╡─♦
750E8860  CE C7 0E 00 3D D4 04 00 53 D4 04 00 D7 2D 04 00  ╬╟Å =╘♦ S╘♦ ╫─♦
750E8870  E8 C7 04 00 18 78 02 00 C1 7F 03 00 AA 8C 03 00  ╗╟♦ ↑x☻ ┴⌂♥ ¬îΦ
750E8880  13 8F 03 00 F3 2D 04 00 E2 2D 04 00 FF 52 01 00  ‼A♥ ≤─♦ Γ─♦  R0
750E8890  A5 3A 05 00 33 15 02 00 FC 19 02 00 04 2E 04 00  ñ:♣ 3§☻ ⁿ↓☻ ♦.♦
```

[תחילת מערך ערכי ה-HASH של הפונקציות בתוך Kernel32.dll]

הטכניקה המלאה של מציאת כל ה-API של מערכת ההפעלה כמו שהיא ממומשת בקוד שלנו מוסברת לעומק במאמר על shellcode אשר מצורף גם הוא בבליוגרפיה. בסופו של דבר, אני משתמש בטכניקה כמו שמשתמשים בכל פונקציה אחרת ושולח לה פרמטרים דרושים ומקבל חזרה את כתובת תחילת הפונקציה הרצויה אותה אני שומר ועושה בה שימוש בעת הצורך. לדוגמא:

```
push FIND_NEXT_FILE_W_HASH_LITTLE_ENDIAN
push Kernel32BaseAddr
call find_function
mov FindNextFileWAddr, eax
```

בקטע הקוד שלהלן אני קורא לפונקציה find_function עם 2 פרמטרים, ערך ה-HASH של הפונקציה אותה אני מחפש וכתובת הטעינה של הספריה Kernel32.dll אותה מצאתי בהתחלה. לאחר החזרה מהפונקציה, אני מצפה שכתובת ההתחלה של הפונקציה תהיה מאוחסנת ברגיסטר eax ולכן אני מעביר את תוכנו למשתנה שלי בשם FindNextFileAddr. כעת כתובת ההתחלה של הפונקציה FindNextFile נמצאה ואוחסנה ואני יכול לעשות שימוש בפונקציה זו ממש כמו בפונקציה שייבאתי בדרך סטנדרטית.

הכרזת משתנים

מה המשמעות של הכרזת משתנים בתוכנית שאמורה לרוץ בתור Position independent code?

בתוכנית זו נעשה שימוש בלא מעט משתנים אשר מחזיקים ערכי זיכרון חשובים כמו כתובות של פונקציות מתוך ה-API של מערכת ההפעלה. צריך לזכור שאת הכרזת משתנים זו אסור לעשות מחוץ לגבולות הגזרה של הפונקציה משום שבהגדרת משתנים גלובאלית הם יוגדרו בחלק אחר בזיכרון אשר אינו נשמר עם השכפול של הוירוס. הוירוס מעתיק הרי רק את הקוד שלו (code section) מהקובץ הנוכחי לקובץ

הבא, כל ערך שקיים ב- data section או חלקים אחרים לא ישמר ואף יותר מזאת, כאשר תוכנית מתחילה שלרוץ והוירוס "חוטף" אותה על מנת לבצע קודם כל את הקוד שלו עצמו, הוא אינו יודע איך נראים החלקים של התוכנית אשר נטענה לזיכרון, אסור להניח שתחילת ה- data section ריק ואינו מאותחל לערך כלשהו על ידי מערכת ההפעלה.

כאשר המשתנים שלנו מוגדרים בתוך הפונקציה אנו שומרים מקום לאותם ערכים במחסנית של ה-Process וכך אנו לא דורסים שום מידע של התהליך עצמו ומבטיחים ששמירת הנתונים של הוירוס לא תשפיע על ריצת התהליך לו הוירוס "דבוק" ואשר ירוץ מיד אחרי הוירוס.

שיטות של החדרת String לתוך ה- Code Section

כאשר אנו מבצעים הגדרה של String בשפה עילית או גם בקוד C, הקומפיילר יודע בעצם להעתיק את המידע הזה לתוך Section מתאים בזיכרון ואז להעתיקו ל- Data Section בתחילת הריצה. כמו שאמרנו קודם, מכיוון שהחלק היחיד ששורד בין שכפול של הוירוס הוא רק ה-Code Section, אין באמת יכולת ידידותית להעביר Raw Data בין שכפול לשכפול של הוירוס, הגדרה של String כמו שאנחנו רגילים תגרום למצב שהוירוס פועל רק בריצתו הראשונה בה קובץ התוכנית הוא תוצאת הריצה של הקומפיילר ולא "מודבק" לשום תוכנית אחרת.

בתוכנית זו אנחנו משתמשים ב-String-ים בצורה די נרחבת עבור השוואת שמות של קבצים ושל ספריות, ולכן בניתי עבור כל String פונקציה אשר מחזירה פוינטר ל-String המבוקש. כל פונקציה כזו מכילה בעצם פתיח שזהה בין כל הפונקציות ולאחריו את ה-String עצמו. כל אותם פונקציות נמצאות בקוד לפני הפונ' Main.

נקח את אחת הפונ' לצורך הסבר:

```
_declspec(naked) void PathString()
//the desired result of calling this function is the address of the
wanted string in the eax register
{
        _asm
        {
                // function prologue
                _emit 0xe8
                _emit 0x00
                _emit 0x00
                _emit 0x00
                _emit 0x00
                pop eax
                add eax,5
                ret
                ////////////
                //actual string
                //-------------
```

```
            //the string "C:\Virus" in wide char form
            _emit 0x43
            _emit 0x00
            _emit 0x3a
            _emit 0x00
            _emit 0x5c
            _emit 0x00
            _emit 0x56
            _emit 0x00
            _emit 0x69
            _emit 0x00
            _emit 0x72
            _emit 0x00
            _emit 0x75
            _emit 0x00
            _emit 0x73
            _emit 0x00
            //the string "\*.*"
            _emit 0x5c
            _emit 0x00
            _emit 0x2a
            _emit 0x00
            _emit 0x2e
            _emit 0x00
            _emit 0x2a
            _emit 0x00
            // terminating NULL BYTE X2 for wide char form
            _emit 0x00
            _emit 0x00
        }
}
```

הפונקציה מוגדרת באמצעות מילת המפתח _declspec אשר מאפשרת הגדרות הרבה יותר ספציפיות של בניית הקוד של הפונקציה, כמו למשל כמה מקום להגדיר במחסנית של הפונקציה או כיצד "ליישר" את הקוד בתוך הפונקציה או באיזה calling convention הפונקציה צריכה להקרא (כמובן שיש לזה השלכות על איך הקוד של הפונקציה צריך להראות). פונקצית ה-Main שלנו למשל אינה מוגדרת באמצעות שום הגדרה ספציפית ולכן הקוד שלה מתחיל כך:

```
00AD109E    CC              INT3
00AD109F    CC              INT3
00AD10A0  $ 55              PUSH EBP                                  INT Project2.main(argc,argv)
00AD10A1  . 8BEC            MOV EBP,ESP
00AD10A3  . B8 44130000     MOV EAX,1344
00AD10A8  . E8 13070000     CALL _chkstk                              _chkstk
00AD10AD  . 53              PUSH EBX
00AD10AE  . 56              PUSH ESI
00AD10AF  . 57              PUSH EDI
00AD10B0  . C745 FC 0000000 MOV DWORD PTR SS:[EBP-4],0
00AD10B7  . C745 F8 0000000 MOV DWORD PTR SS:[EBP-8],0
00AD10BE  . C745 F4 0000000 MOV DWORD PTR SS:[EBP-0C],0
00AD10C5  . C745 F0 0000000 MOV DWORD PTR SS:[EBP-10],0
00AD10CC  . C745 EC 0000000 MOV DWORD PTR SS:[EBP-14],0
00AD10D3  . C745 E8 0000000 MOV DWORD PTR SS:[EBP-18],0
00AD10DA  . C745 E4 0000000 MOV DWORD PTR SS:[EBP-1C],0
00AD10E1  . C745 E0 0000000 MOV DWORD PTR SS:[EBP-20],0
00AD10E8  . C745 DC 0000000 MOV DWORD PTR SS:[EBP-24],0
00AD10EF  . C745 D8 0000000 MOV DWORD PTR SS:[EBP-28],0
00AD10F6  . C745 D4 0000000 MOV DWORD PTR SS:[EBP-2C],0
00AD10FD  . C745 D0 0000000 MOV DWORD PTR SS:[EBP-30],0
00AD1104  . C745 CC 0000000 MOV DWORD PTR SS:[EBP-34],0
00AD110B  . C745 C8 0000000 MOV DWORD PTR SS:[EBP-38],0
00AD1112  . C745 C4 0000000 MOV DWORD PTR SS:[EBP-3C],0
00AD1119  . C745 C0 0000000 MOV DWORD PTR SS:[EBP-40],0
00AD1120  . C745 BC 0000000 MOV DWORD PTR SS:[EBP-44],0
00AD1127  . C785 A8EFFFFF 0 MOV DWORD PTR SS:[EBP-1058],0
00AD1131  . C785 A4EFFFFF 0 MOV DWORD PTR SS:[EBP-105C],0
00AD113B  . C785 4CEEFFFF 0 MOV DWORD PTR SS:[EBP-11B4],0
00AD1145  . C785 48EEFFFF 0 MOV DWORD PTR SS:[EBP-11B8],0
00AD114F  . C785 44EEFFFF 0 MOV DWORD PTR SS:[EBP-11BC],0
00AD1159  . 33C0            XOR EAX,EAX
00AD115B  . 66:8985 40EEFFF MOV WORD PTR SS:[EBP-11C0],AX
00AD1162  . C785 04EEFFFF 0 MOV DWORD PTR SS:[EBP-11FC],0
00AD116C  . 6A 38           PUSH 38                                   count = 56.
00AD116E  . 6A 00           PUSH 0                                    value = 0
00AD1170  . 8D85 08EEFFFF   LEA EAX,[EBP-11F8]
00AD1176  . 50              PUSH EAX                                  dst
```

ניתן לראות שהקומפיילר הכניס בצורה אוטומאטית קוד אשר שומר את נתוני המחסנית והרגיסטרים של הקוד הקורא לפונ' זו וגם מכין את המחסנית של הפונ' הנוכחית.

את הפונקציות אשר קשורות ב-String'ים, אנו מגדירים באמצעות הפרמטר "naked" אשר גורם לקומפיילר לא להוסיף שום שורה לקוד שלנו ולא להוסיף שום פעולה מלבד הקוד שכתבנו, צורה זו אינה מתאימה כמובן לכל פונקציה, היא כן מתאימה לפונקציות שאמורות לבצע פעולות מאוד קצרות ומסוימות או לפונקציות שמתבצעות מספר רב של פעמים ולכן נדרשות לחתימה נמוכה ככל שאפשר. מכיוון שאנו משתמשים בפרמטר "naked" האסמבלי של הפונ' נראה כך:

```
00AD1000  $  E8 00000000          CALL 00AD1005          Project2.PathString(void)
00AD1005  $  58                   POP EAX
00AD1006  .  83C0 05              ADD EAX,5
00AD1009  .  C3                   RETN
00AD100A  .  4300 3A00 5C00 5600 6900 7200 7500 7300   UNICODE "C:\Virus"      UNICODE "C:\Virus\*.*"
00AD101A  .  5C00 2A00 2E00 2A00 0000   UNICODE "\*.*",0
00AD1024     CC                   INT3
00AD1025     CC                   INT3
00AD1026     CC                   INT3
```

ניתן לראות שהקומפיילר פשוט כתב את ההוראות באסמבלי מבלי להוסיף כלום. אבל איפה בעצם ה-String פה?

הפקודה הראשונה בה נעשה שימוש היא 0xE800000000. מדובר בעצם בפקודת Call אשר מבצעת קפיצה להוראה הבאה בקוד, כתבתי את ההוראה ב-hex ולא באסמבלי מטעמי נוחות כי ההוראה באסמבלי מקבלת פרמטר של כתובת (Call [ADDRESS]) כמובן שניתן להחליפה בשם של Label כזה או אחר, אבל כאשר רוצים לקפוץ להוראה הבאה אז פשוט יותר לכתוב את ה-HEX. הפקודה Call גם דוחפת למחסנית את הכתובת של ההוראה שאחרי פקודת ה-Call וקופצת ל-Offset הרצוי (כך ניתן לחזור לקוד הקורא לאחר קריאה לפונקציה), במקרה שלנו היא דוחפת את הכתובת של הפקודה הבאה וגם קופצת אליה, הפקודה הבאה מוציאה מהמחסנית את הכתובת שהכנסנו קודם והפקודה שאחריה מוסיפה לה 5, הכתובת של הפקודה POP EAX (שבפועל מוציאה את הכתובת) היא 0xAD1005 (במקרה הספציפי של הדוגמא), כאשר אנחנו מוסיפים לה 5 אנו מגיעים בדיוק לפקודה שאחרי פקודת ה-RETURN. הפקודה שאחרי פקודת ה- RETURNהיא למעשה אינה פקודה כלל אלא ה-String בו רצינו להשתמש כך שבסופו של דבר, לפני ביצוע פקודת RETURN, הערך של ה- String הרצוי נמצא ברגיסטר EAX. משתנה זה אמור להכיל את ערך החזרה של הפונקציה לפי stdcall שהיא ה-Calling convention בה נעשה שימוש ב- Win API של 32bit . בסופו של דבר, בקריאה לפונקציה, מקבלים חזרה פוינטר ל- String הרצוי.

עם זאת, צריך לזכור, שמדובר במידע שנמצא ב- Code section ולכן שלא כמו ב- String רגיל, לא ניתן לבצע בו שינויים.

קונפיגורציות של הקומפיילר ושיוף אחד אחרון

פונקציות מערכת ב-Wide char:

אפשר לשים לב שבשימוש ברוב פונקציות המערכת אותם אני מייבא ממערכת ההפעלה, אני עושה שימוש בגרסאת ה-Wide Char של הפונקציה. הרבה מתכנתים לא בהכרח מכירים את הצורות השונות של הפונקציות בגלל שבפרויקט בו הם לוקחים חלק יש כבר הגדרה אבסטריקטית יותר של אותה פונקציה, למשל, הפונקציה LoadLibrary של מערכת ההפעלה היא אינה פונקציה אמיתית אלה מוחלפת על ידי הפרה-קומפיילר בזמן קומפילציה לאחת מהפונקציות LoadLibraryA או LoadLibraryW בהתאם לפרויקט. ההבדל בין 2 הגרסאות הוא שהראשונה מקבלת String בקידוד ASCII והשניה מקבלת String בקידוד של Unicode. קידוד זה מגדיר 16 ביטים לתו ולא 8 ביטים כמו ASCII (כך ניתן כמובן לקודד תווים נוספים בשפות אחרות).

הפונקציות ב-Kernel של Windows מקבלות Stringים ב-Unicode וכאשר אנו מבצעים פונקציה אשר מקבלת String של ASCII, String זה מומר לגרסאתו ב-Unicode לפני ביצוע הפונקציה ולכן היה נראה לי טבעי לעבוד בגרסאת ב-Unicode של הפונקציות. בדיעבד זו לאו דווקא ההחלטה הנכונה אם יש כמות נכבדת של עבודה עם Stringים מכיוון שזה קצת מכביד על הכתיבה ותופס פי-2 בתים לכל תו. כמובן שזה חוסך את זמן ההמרה ב- Kernel אבל ברוב המקרים מדובר בזמן ומאמץ זניחים. גם אצלי בקוד ישנם מקרים בודדים שהשתמשתי בגרסאת ה-ASCII של הפונקציה מטעמי נוחות.

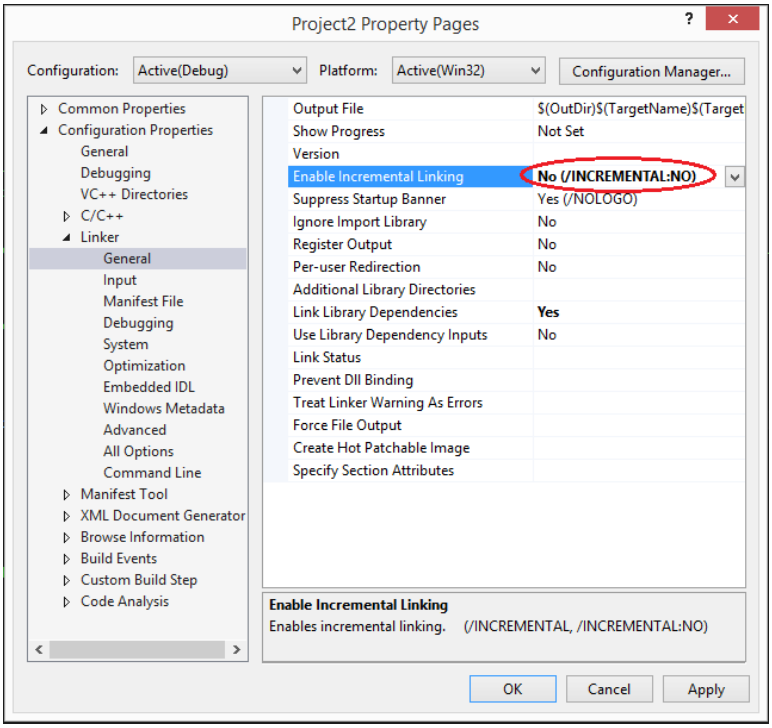
Incremental Linking:

מדובר באופציה בתהליך ה-Linkage (לינקוג' בעברית צחה או לינקינג בעברית קלוקלת) אשר בונה Jump table בתחילת הקוד, טבלה זו מתמלאת על ידי ה-Loader שמכין את ה-Process לריצה ומוכנסים אליה הערכים האמיתיים של מיקומי פונקציות המערכת של Windows, כאשר יש קריאה בקוד לפונקצית מערכת כלשהי, הקריאה היא למעשה רק לערך המתאים בטבלה ומשם ישנה קפיצה לפונקציה עצמה, זו דוגמא לטבלה כזו לאחר קומפילציה עם האופציה הזו מאופשרת:

```
00391834  $v E9 703C0000   JMP _ioinit
00391839  .v E9 E2D90100   JMP _GetCurrentProcessId@0   Jump to KERNEL32.GetCurrentProcessId
0039183E  .v E9 E62C0000   JMP _forcdecpt
00391843  $v E9 03300100   JMP __crtDownlevelLocaleNameToLCID
00391848  $v E9 39AC0100   JMP _itow_s
0039184D  $v E9 A1460000   JMP __set_app_type
00391852  .v E9 7BD80000   JMP _RoundMan
00391857  $v E9 DA5B0000   JMP _isalpha_l
0039185C  $v E9 50CD0100   JMP __crtLCMapStringW
00391861  .v E9 66D40000   JMP __iswcsymf
00391866  >v E9 847C0000   JMP __crtLoadWinApiPointers
0039186B  .v E9 6F8E0000   JMP signal
00391870  $v E9 6B0F0000   JMP Shell32String
00391875  .v E9 2FC80100   JMP wcstoimax
0039187A  .v E9 FB5C0000   JMP _iscsym_l
0039187F  .v E9 69AC0100   JMP _ui64tow_s
00391884  $v E9 870F0000   JMP main
00391889  .v E9 96300000   JMP __threadid                Jump to KERNEL32.GetCurrentThreadId
0039188E  $v E9 E6500000   JMP __updatelocinfo
00391893  .v E9 F1D50000   JMP _iswupper_l
00391898  .v E9 05D90100   JMP _EncodePointer@4          Jump to ntdll.RtlEncodePointer
0039189D  $v E9 74470000   JMP _RTC_Initialize
```

ניתן לזהות את הקפיצות לחלק מהפונקציות המוכרות לנו מ-Kernel32. ישנם הרבה יתרונות לטבלה כזו בתוכנית אמיתית, כמו הצורך לעדכן את מיקום הפונקציות בזיכרון רק במקום אחד ולא מספר רב של

פעמים במהלך הקוד (בכל קריאה לפונקציה), אך התוכנית שלנו היא לא תוכנית אמיתית אלא וירוס (או לפחות מנסה להיות) ולכן בנית הקוד בצורה כזו היא מאוד בעייתית. כמו שהסברתי קודם, הוירוס ממילא משיג ממשק למערכת ההפעלה באמצעים שלו ולכן אין לו שימוש בטבלה כזו, כמו כן, המצאות טבלה כזו ב-Code Section אומרת שגם הטבלה תועתק בעת שכפול הוירוס וזה אינו רצוי בכלל שזה מאוד קשה לתחזוק ומגדיל את הקוד ובאופן כללי מתאים רק לתוכניות "אמיתיות" ולכן כדאי לכבות את האופציה הזו בקונפיגורציות.



ביטול בדיקות הביניים של מערכת ההפעלה:

זוהי דוגמא טובה נוספת להשלכות משמעותיות של קונפיגורציה דיפולטיבית של הקומפיילר. בזמן הריצה של תוכנית רגילה, ישנן קריאות לפונקציות ספריה שונות של מערכת ההפעלה שהקומפיילר דואג להכניס במקומות אסטרטגיים בקוד כברירת מחדל, בקטע הקוד הבא למשל, ניתן לראות כיצד לאחר כל קריאה לפונקציה, נקראת פונקצית מערכת של Windows אשר תפקידה לבדוק את אמינות ה- Stack לאחר החזרה מהפונקציה.

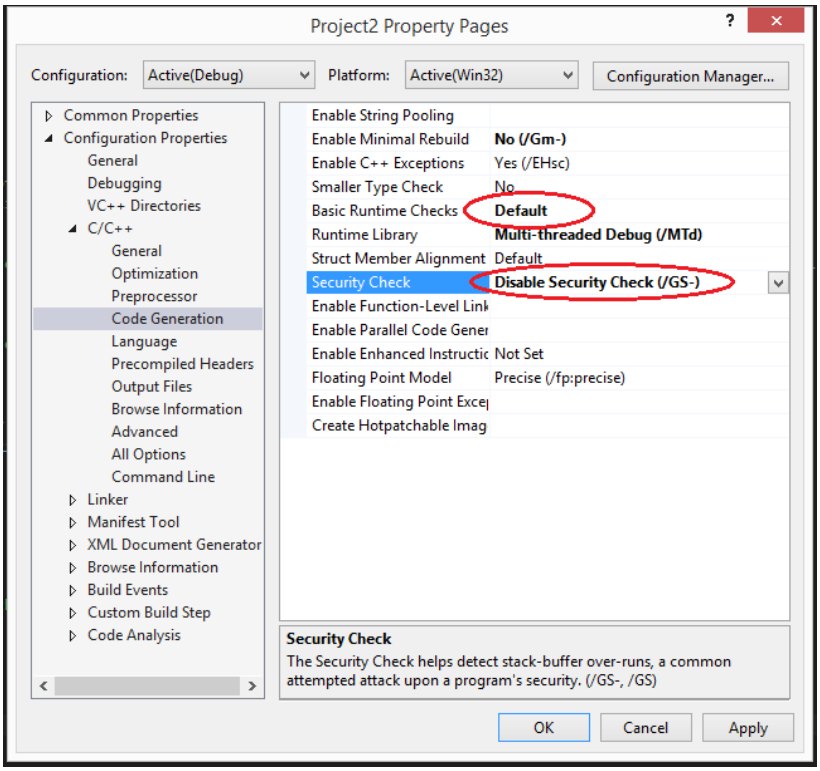
```
00B01351  .   51
00B01352  .   FF55 C4
00B01355  .   3BF4
00B01357  .   E8 C4070000
00B0135C  .   8985 F0EEFFFF
00B01362  >   8BF4
00B01364  .   FF55 A0
00B01367  .   3BF4
00B01369  .   E8 B2070000
00B0136E  .   83F8 12
00B01371  .v  0F84 59050000
00B01377  .   8BF4
00B01379  .   8D85 C8EDFFFF
00B0137F  .   50
00B01380  .   FF55 AC
00B01383  .   3BF4
00B01385  .   E8 96070000
00B0138A  .   8985 90EDFFFF
00B01390  .   83BD 90EDFFFF 04
00B01397  .v  0F8E 12050000
00B0139D  .   8B85 90EDFFFF
```

```
PUSH ECX
CALL DWORD PTR SS:[EBP-3C]
CMP ESI,ESP
CALL _RTC_CheckEsp                                  _RTC_CheckEsp
MOV DWORD PTR SS:[EBP-1110],EAX
MOV ESI,ESP
CALL DWORD PTR SS:[EBP-60]
CMP ESI,ESP
CALL _RTC_CheckEsp                                  _RTC_CheckEsp
CMP EAX,12
JE 00B018D0
MOV ESI,ESP
LEA EAX,[EBP-1238]
PUSH EAX
CALL DWORD PTR SS:[EBP-54]
CMP ESI,ESP
CALL _RTC_CheckEsp                                  _RTC_CheckEsp
MOV DWORD PTR SS:[EBP-1270],EAX
CMP DWORD PTR SS:[EBP-1270],4
JLE 00B018AF
MOV EAX,DWORD PTR SS:[EBP-1270]
```

במקרה זה, אם התרחש Stack Corruption התוכנית לא תמשיך לרוץ ותזהה את ה-Corruption מיד לאחר החזרה מהפונקציה אשר גרמה לו, תצא בצורה מסודרת (יחסית) ותוכל אפילו להודיע על מיקום הקוד הבעייתי.

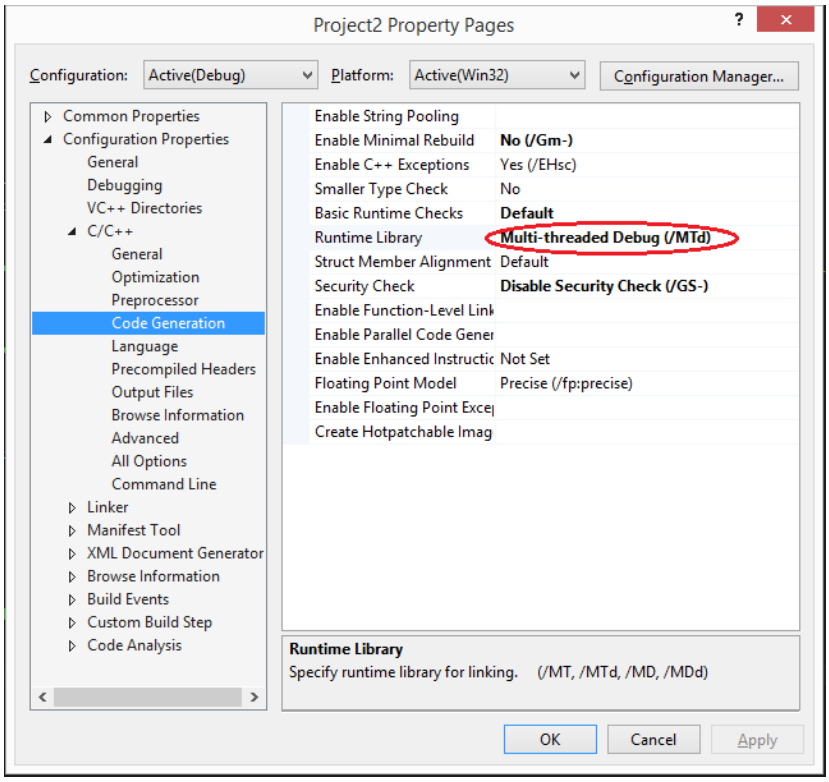
זוהי תכונה חיובית ורצויה בעת פיתוח תוכנית אמיתית או גדולה אך שוב, כאן אנו בונים קוד מאוד קטן בו ישנה חשיבות לכל בית, מעבר לכך, לא ניתן להעתיק את הקריאות האלו כמו שהן מכיוון שמיקומן של הפונקציות הנקראות לא נשמר בין ריצה לריצה או בין מערכת למערכת, באופן כללי בעת כתיבת וירוס יש דרישה חזקה מאוד לשליטה מאוד גבוהה בתהליך הביצוע של הקוד, הגדרות קומפיילר הן גורם זניח יחסית בקורסי תכנות באוניברסיטה בהן יש דגש על הלוגיקה עצמה אך הגדרות אלו הן בעלות משמעות מאוד גבוהה בפיתוח קוד שיש לו אינטרקציה גבוהה עם הסביבה שלו.

לגבי הקונפיגורציות שיש להן קשר בבדיקות אלו, כדאי לנטרל את כולן כך:



לרוב הפרויקטים C שנכתבים ב- Visual Studio יש הסתמכות מסוימת על ספריה המלאה בפונקציות עזר שהקומפיילר מוסיף לפרויקט ומכילה פונקציות שונות, למשל פונקציות העזר שבודקות מצב המחסנית עליהן דובר בפסקה הקודמת. זאת אומרת שגם בבנית פרויקט מאוד בסיסי בו אנו לא מכלילים באופן מודע שום ספריה חיצונית ישנה הכללה של ההספריה הזו.

שם הקובץ משתנה בהתאם לגרסא של Visual Studio אך מדובר בסדרת הקבצים המתחיל ב-MSVCR. למשל אצלי בפרויקט ראיתי שישנה הוספה "אוטומאטית" על ידי הקומפיילר של MSVCR100.DLL. כמו שאמרתי כבר קודם, בכתיבה של וירוס ישנו נסיון לא להכליל בפרויקט ספריות שהייבוא שלהן לא נעשה על ידי הקוד וכנראה שלא נרצה להשתמש בשום קוד של ספריה זו אך צעד מקדים אשר מבטיח שלא נשתמש בקוד של ספריה זו כאשר היא מחוץ לפרויקט הוא לייבא את הספריה לתוך הפרויקט על ידי הקונפיגורציה הבאה:



שיוף אחרון:

אחרי כל הקונפיגורציות המעניינות שגיליתי שהיו הכרחיות ועלו לי בשעות של דיבוג הגיעה האחרונה שבהן. שמתי לב שלצורך איפוס ה-Stack frame החדש שהוירוס יוצר, הקומפיילר משתמש בפונקצית memset. לאחר שביצענו את הצעד שבפסקה הקודמת, פונקציה זו אמורה להכלל בקוד המקומפל ולא להטען מספריה חיצונית. אך עדיין, היא תמצא בחלק אחר של הקובץ ובתהליך שכפול הקובץ אנו לא

רוצים להעתיק את כל הספריה של MSVCR100.DLL מכיוון שזה מאוד מסורבל ומגדיל את החתימה של הוירוס באופן משמעותי. אם לומר את האמת לא מצאתי בקונפיגורציות של הקומפיילר דרך להוריד את הקריאה הזו ולמנוע את האיפוס באמצעות הפונקציה memset ולכן החלטתי שפשוט אוריד את הקריאה בעצמי מהקוד הסופי, ניתן לעשות את העריכה הזו של הקוד באמצעות מספר Debuggers או Hex Editors אבל אני ממליץ להשתמש ב-OllyDbg שהוא כלי חזק מאוד ונוח לשימוש ויש לו קהילת משתמשים מאוד רחבה ולכן קל למצוא דוקומנטציה או מדריכים לשימוש בכלי.

תחילת הקוד מיד לאחר הקומפילציה:

```
011F109B     CC           INT3
011F109C     CC           INT3
011F109D     CC           INT3
011F109E     CC           INT3
011F109F     CC           INT3
011F10A0 $   55           PUSH EBP                                   INT Project2.main(argc,argv)
011F10A1 .   8BEC         MOV EBP,ESP
011F10A3 .   B8 44130000  MOV EAX,1344
011F10A8 .   E8 13070000  CALL _chkstk                               _chkstk
011F10AD .   53           PUSH EBX
011F10AE .   56           PUSH ESI
011F10AF .   57           PUSH EDI
011F10B0 .   C745 FC 0000 MOV DWORD PTR SS:[EBP-4],0
011F10B7 .   C745 F8 0000 MOV DWORD PTR SS:[EBP-8],0
011F10BE .   C745 F4 0000 MOV DWORD PTR SS:[EBP-0C],0
011F10C5 .   C745 F0 0000 MOV DWORD PTR SS:[EBP-10],0
011F10CC .   C745 EC 0000 MOV DWORD PTR SS:[EBP-14],0
011F10D3 .   C745 E8 0000 MOV DWORD PTR SS:[EBP-18],0
011F10DA .   C745 E4 0000 MOV DWORD PTR SS:[EBP-1C],0
011F10E1 .   C745 E0 0000 MOV DWORD PTR SS:[EBP-20],0
011F10E8 .   C745 DC 0000 MOV DWORD PTR SS:[EBP-24],0
011F10EF .   C745 D8 0000 MOV DWORD PTR SS:[EBP-28],0
011F10F6 .   C745 D4 0000 MOV DWORD PTR SS:[EBP-2C],0
011F10FD .   C745 D0 0000 MOV DWORD PTR SS:[EBP-30],0
011F1104 .   C745 CC 0000 MOV DWORD PTR SS:[EBP-34],0
011F110B .   C745 C8 0000 MOV DWORD PTR SS:[EBP-38],0
011F1112 .   C745 C4 0000 MOV DWORD PTR SS:[EBP-3C],0
011F1119 .   C745 C0 0000 MOV DWORD PTR SS:[EBP-40],0
011F1120 .   C745 BC 0000 MOV DWORD PTR SS:[EBP-44],0
011F1127 .   C785 A8EFFFF MOV DWORD PTR SS:[EBP-1058],0
011F1131 .   C785 A4EFFFF MOV DWORD PTR SS:[EBP-105C],0
011F113B .   C785 4CEFFFF MOV DWORD PTR SS:[EBP-11B4],0
011F1145 .   C785 48EFFFF MOV DWORD PTR SS:[EBP-11B8],0
011F114F .   C785 44EFFFF MOV DWORD PTR SS:[EBP-11BC],0
011F1159 .   33C0         XOR EAX,EAX
011F115B .   66:8985 40EE MOV WORD PTR SS:[EBP-11C0],AX
011F1162 .   C785 04EFFFF MOV DWORD PTR SS:[EBP-11FC],0
011F116C .   6A 38        PUSH 38                                    count = 56.
011F116E .   6A 00        PUSH 0                                     value = 0
011F1170 .   8D85 08EFFFF LEA EAX,[EBP-11F8]
011F1176 .   50           PUSH EAX                                   dst
011F1177 .   E8 74060000  CALL memset                                memset
011F117C .   83C4 0C      ADD ESP,0C
011F117F .   E8 CCFEFFFF  CALL RunasString                           RunasString
011F1184 .   8985 ACEFFFF MOV DWORD PTR SS:[EBP-1054],EAX
011F118A .   E8 71FEFFFF  CALL PathString                            PathString
011F118F .   8985 B4EFFFF MOV DWORD PTR SS:[EBP-104C],EAX
```

תחילת הקוד לאחר הסרת הקריאה לפונ' memset:

```
0116109C     CC           INT3
0116109D     CC           INT3
0116109E     CC           INT3
0116109F     CC           INT3
011610A0 $   55           PUSH EBP                                   INT Project2.main(argc,argv)
011610A1 .   8BEC         MOV EBP,ESP
011610A3 .   B8 44130000  MOV EAX,1344
011610A8 .   E8 13070000  CALL _chkstk                               _chkstk
011610AD .   53           PUSH EBX
011610AE .   56           PUSH ESI
011610AF .   57           PUSH EDI
011610B0 .   C745 FC 0000 MOV DWORD PTR SS:[EBP-4],0
011610B7 .   C745 F8 0000 MOV DWORD PTR SS:[EBP-8],0
011610BE .   C745 F4 0000 MOV DWORD PTR SS:[EBP-0C],0
011610C5 .   C745 F0 0000 MOV DWORD PTR SS:[EBP-10],0
011610CC .   C745 EC 0000 MOV DWORD PTR SS:[EBP-14],0
011610D3 .   C745 E8 0000 MOV DWORD PTR SS:[EBP-18],0
011610DA .   C745 E4 0000 MOV DWORD PTR SS:[EBP-1C],0
011610E1 .   C745 E0 0000 MOV DWORD PTR SS:[EBP-20],0
011610E8 .   C745 DC 0000 MOV DWORD PTR SS:[EBP-24],0
011610EF .   C745 D8 0000 MOV DWORD PTR SS:[EBP-28],0
011610F6 .   C745 D4 0000 MOV DWORD PTR SS:[EBP-2C],0
011610FD .   C745 D0 0000 MOV DWORD PTR SS:[EBP-30],0
01161104 .   C745 CC 0000 MOV DWORD PTR SS:[EBP-34],0
0116110B .   C745 C8 0000 MOV DWORD PTR SS:[EBP-38],0
01161112 .   C745 C4 0000 MOV DWORD PTR SS:[EBP-3C],0
01161119 .   C745 C0 0000 MOV DWORD PTR SS:[EBP-40],0
01161120 .   C745 BC 0000 MOV DWORD PTR SS:[EBP-44],0
01161127 .   C785 A8EFFFF MOV DWORD PTR SS:[EBP-1058],0
01161131 .   C785 A4EFFFF MOV DWORD PTR SS:[EBP-105C],0
0116113B .   C785 4CEFFFF MOV DWORD PTR SS:[EBP-11B4],0
01161145 .   C785 48EFFFF MOV DWORD PTR SS:[EBP-11B8],0
0116114F .   C785 44EFFFF MOV DWORD PTR SS:[EBP-11BC],0
01161159 .   33C0         XOR EAX,EAX
0116115B .   66:8985 40EE MOV WORD PTR SS:[EBP-11C0],AX
01161162 .   C785 04EFFFF MOV DWORD PTR SS:[EBP-11FC],0
0116116C .   6A 38        PUSH 38                                    count = 56.
0116116E .   6A 00        PUSH 0                                     value = 0
01161170 .   8D85 08EFFFF LEA EAX,[EBP-11F8]
01161176 .   50           PUSH EAX                                   dst
01161177     90           NOP
01161178     90           NOP
01161179     90           NOP
0116117A     90           NOP
0116117B     90           NOP
0116117C .   83C4 0C      ADD ESP,0C
0116117F .   E8 CCFEFFFF  CALL RunasString                           RunasString
01161184 .   8985 ACEFFFF MOV DWORD PTR SS:[EBP-1054],EAX
0116118A .   E8 71FEFFFF  CALL PathString                            PathString
0116118F .   8985 B4EFFFF MOV DWORD PTR SS:[EBP-104C],EAX
```

ניתן לראות שהחלפתי את הקריאה לפונקציה בפקודות NOP אשר מבצעות פעולה שאין לה השלכה על המצב של המערכת או הזיכרון. אפשר לראות שהשארתי את הפעולות שמכניסות הפרמטרים לפונקציה memset בקוד. זה לא כל כך נכון מבחינת נקיון של הקוד אבל זה בכל זאת עובד אז השארתי את זה ככה.

חדי-העין מבינכם יראו שישנה קריאה לפונ' נוספת בשם chktsk_ ממש בתחילת הקוד אותה לא הסרתי, זאת מכיוון שהקוד של הפונקציה הזו מוכלל בתוכנית שלנו ומגיע מיד לאחר הקוד של הוירוס ונכנס ב-Page אותו אנחנו מעתיקים בתהליך שכפול הוירוס ולכן ידעתי שהשארה של הקריאה הזו לא תהווה בעיה בקוד.

חלק שני - חיפוש קבצים

בחלק זה של התוכנית, אנו מחפשים קובץ מתאים להדבקה בספריה C:\Virus, שדרישותיו הם:

1. קובץ בעל סיומת EXE - ישנה הנחה שכל קובץ כזה הוא קובץ של תוכנית תקינה אשר בנויה מקוד ב-x86 וכתוב בפורמט PE. אני לא בודק כאן למשל אם הקובץ בעל הסיומת EXE שמצאתי בספריה הוא באמת תוכנית או שמה מדובר בקובץ בפורמט אחר שרק שינו לו את הסיומת. אני גם לא בודק אם מדובר בקובץ אשר בנוי ב-x64 ויצריך גישה שונה למערכת ההפעלה.
2. קובץ שאינו נדבק כבר בוירוס - כמובן שאם הקובץ כבר נדבק בעבר בוירוס אז לא נרצה לנסות להדביקו בשנית אלא להשאירו כמו שהוא.

הרעיון הבסיסי מתואר בתכנון הכללי של הקוד אבל נחזור עליו שוב:

1. נבצע מעבר על כל הקבצים בתיקייה C:\Virus ונחפש קובץ בעל סיומת EXE.
2. אם מצאנו קובץ כזה ננסה לפתוח את הקובץ עם הרשאות כתיבה.
3. במידה והצלחנו נדביק את הקובץ (תהליך ההדבקה מתואר בחלק המתאים במסמך).
4. במידה ולא הצלחנו (אנחנו מניחים פה שאם לא הצלחנו לפתוח את הקובץ הסיבה היא חוסר הרשאות) נבקש מהמשתמש להריץ את התוכנית שוב עם הרשאות של Admin (מדובר כמובן על התוכנית שלנו ולא על התוכנית אותה אנו מדביקים שכלל אינה רצה).
5. במידה והמשתמש מסכים נפתח שוב את התוכנית עם הרשאות Admin וכעת יהיו לנו הרשאות כתיבה לקובץ.
6. במידה והמשתמש לא הסכים לפתיחה מחודשת של קובץ התוכנית, נקפוץ לתחילת התוכנית המקורית ונפסיק את פעולת הוירוס.

חיפוש אחר קובץ EXE

חיפוש קובץ בעל הסיומת המתאימה מתבצע בצורה הבאה:

1. מציאת הקובץ הראשון בספריה על ידי שימוש ב-FindFirstFileW.
2. המשך מציאת שאר הקבצים על ידי שימוש ב-FindNextFileW עד קבלת קוד חזרה של ERROR_NO_MORE_FILES שמשמעותו שהתבצע מעבר על כל הקבצים שבספריה.
3. בדיקה האם מדובר בקובץ בעל סיומת של EXE. (שורה 411)

3.1. לא מדובר בבדיקה מסובכת אך אולי קצת לא ברורה למי שרגיל לראות קוד קונבנציונאלי בלבד. מה שאני עושה שם זה בעצם לבדוק את ערך הבתים אשר אמורים להכיל את התווים EXE ובודק אם הערך המוכל בשלושת הבתים שווה לערך ה-ASCII של התווים. למה? גם כאן יכלתי להגדיר string שיכיל "EXE" ולהשוות לסוף ה-string שמכיל את שם הקובץ אבל כמו שציינתי מוקדם יותר, string זה היה מוגדר ב-Data Section, יכלתי להגדיר אותו באותו צורה כמו שהגדרתי Stringים אחרים בתוך ה- Code section אך במקרה הנוכחי בגלל שמדובר רק ב-3 תווים שאני יודע איפה הם צריכים להיות, כך שלדעתי הרבה יותר קל להשוות בצורה הזו.

4. במידה ואכן מדובר בקובץ EXE, מתבצעת קצת עבודת הכנה (שורות 437-455) של הכנת ה- string הדרוש לצורך פתיחת הקובץ ואז הפתיחה עצמה. הפונ' FindNextFileW מחזירה טיפוס נתונים אשר מכיל את שם הקובץ בלבדו הפונ' CreateFileW צריכה לקבל כקלט את שם הקובץ כולל ה- path. ולכן אני מכין שם string אשר מכיל את ה- path המלא אותו אני בונה מה-stringים הקיימים שלי.

פתיחת הקובץ הנמצא ובדיקת / בקשת הרשאות מתאימות

על נושא ההרשאות בגרסאות של Windows חדשות יותר מ-XP כדאי לקרוא ב-MSDN כי מדובר בנושא חשוב ולא מאוד פשוט אשר ממומש על ידי מנגנון UAC. במאמר זה נדבוק לצד הטכני של הדברים ולא נתעמק בתכנון של המנגנון.

באופן כללי, עד מערכת ההפעלה Windows XP משתמש שעבד על המערכת היה בעל הרשאות מלאות ויכל לבצע לכן פעולות מרובות במערכת כמו כתיבה לתקיות אחרות במערכת. בגרסאות חדשות יותר של מערכת ההפעלה זהו אינו המצב ובדומה ללינוקס, שם יש לבקש נקודתית הרשאות לצורך ביצוע פעולות מסוימות גם אם המשתמש המחובר הוא בעל ההרשאות המתאימות (באמצעות sudo למשל), גם ב-Windows המצב דומה (ב- Windows מדובר בחלון שמופיע מדי פעם ומחשיך את כל המסך ומבקש הרשאות Admin מהמשתמש).

כמובן שיכולת שלא ניתן לבצע את תהליך ההדבקה בלעדיה היא היכולת לכתוב ולשנות קבצים קיימים, אחרת כל שינוי שנבצע בקובץ לא ישמר. תוך נסיונותיי במהלך כתיבת הוירוס ראיתי שלעיתים הקוד רץ

עם הרשאות מתאימות ולעיתים ללא, היה לי קצת קשה לאפיין את ההתנהגות הזו ולכן החלטתי להכניס מנגנון לקוד אשר מבקש הרשאות מתאימות במידה ופתיחת הקובץ נכשלת.

כמובן שוירוס ש"מבקש הרשאות ריצה" זה קצת כמו גנב ש"מבקש אישור לגנוב" ולכן אסביר את השימוש במנגנון זה. אתחיל ואומר שמדובר כמובן בפשרה, המצב הרצוי הוא שהמשתמש לא ירגיש כלל את ריצת הוירוס ושהוירוס יבצע את כל הפעולות אותן הוא רוצה לבצע ללא כל סימן ובטח שלא תשאול של המשתמש. ישנם מנגנונים כאלו שניתן להכניס לקוד קיים ולגרום לו לפעול בצורה כזו אך שוב, כמו במקרים קודמים, מדובר בפרויקט בפני עצמו של מציאת 0day מתאים והכנסתו לקוד ותחזוקה של מנגנון זה (0day כזה בימי Windows 8.1 המתעדכנת באופן כה תדיר לא יחזיק מעמד זמן רב). בשורה התחתונה מדובר במאמצים רבים יחסית אשר היו מעכבים את סיום כתיבת הקוד כאשר התועלת שלהם חשובה אך ברוב המקרים אינה בולטת.

ברוב המקרים, הקוד כן רץ עם הרשאות כתיבה לקבצים וגם אם זה לא המקרה, בקשת הרשאות מהמשתמש היא תהליך שמשתמש Windows 8 רגיל לראותו במהלך העבודה לעיתים לא רחוקות ולכן לא מדובר באירוע חריג כל כך. כמו כן, צריך לזכור שחוץ מהריצה הראשונה של הוירוס, בכל שאר הפעמים הוא ירוץ בתוך תהליך לגיטימי ולכן שם קובץ ה-exe שיופיע בבקשת ההרשאות יהיה קובץ של תוכנית המוכרת למשתמש.

לצורך בקשת ההרשאות אני עושה שימוש בפונ' הספריה ShellExecuteExW אשר מבקשת להריץ את שם התוכנית אותו היא מקבלת כקלט עם הרשאות Admin, כדאי לקרוא על פעולת הפונ' ב-MSDN כי הפעולה שלה אינה טריוויאלית או באופן כללי צריך לדעת שתהליך שרוצה לרוץ עם הרשאות Admin ב-Windows צריך "לפתוח את עצמו מחדש" עם הרשאות כאלו. כלומר, לא ניתן לבקש הרשאות כאלו לתוכנית שכבר רצה ולבצע את הפעולות עם ההרשאות לאחר בקשה באמצע חיי התוכנית.

הקריאה לפונ' מצריכה שימוש בטיפוס נתונים מהסוג ShellExecuteInfo אשר אותו אני מכין (שורות 443-461) ונותן כקלט לפונ', הוא מכיל את כל הפרמטרים הדרושים לפעולת הפתיחה של התוכנית.

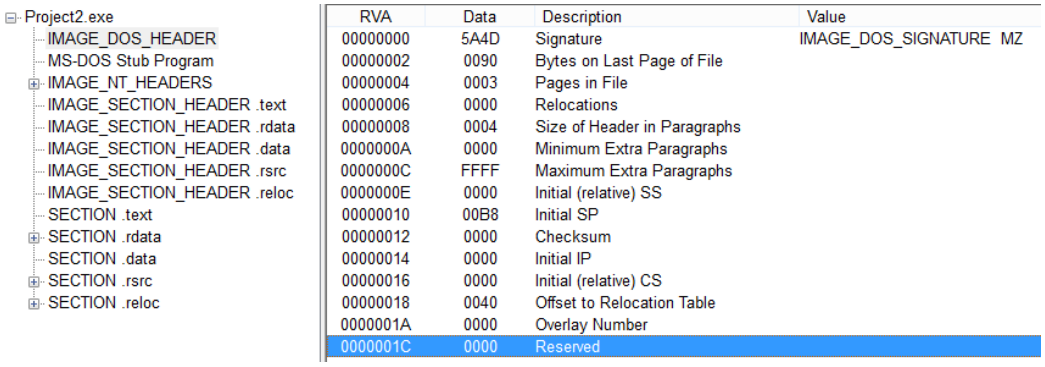
לאחר הקריאה לפונ', אני בודק אם המשתמש אישר את הבקשה לפתיחה מחודשת או לא. אם המשתמש לא אישר את הבקשה התוכנית תקפוץ לנקודת הפתיחה של התוכנית המקורית על ידי קטע הקוד באסמבלי אשר מובא שם. אם המשתמש אישור את הפתיחה, הריצה קופצת לסוף הקוד אשר בהכרח יכיל פקודת חזרה כלשהי.

חלק שלישי - הדבקה

תהליך ההדבקה אינו שונה בהרבה מתהליך ההדבקה כפי שמתואר בקורס Life Of Binaries בתרגיל BabysFirstPhage. הוא מורכב מכמה צעדים:

בדיקה האם הקובץ כבר נדבק בעבר בוירוס או שהקובץ נקי

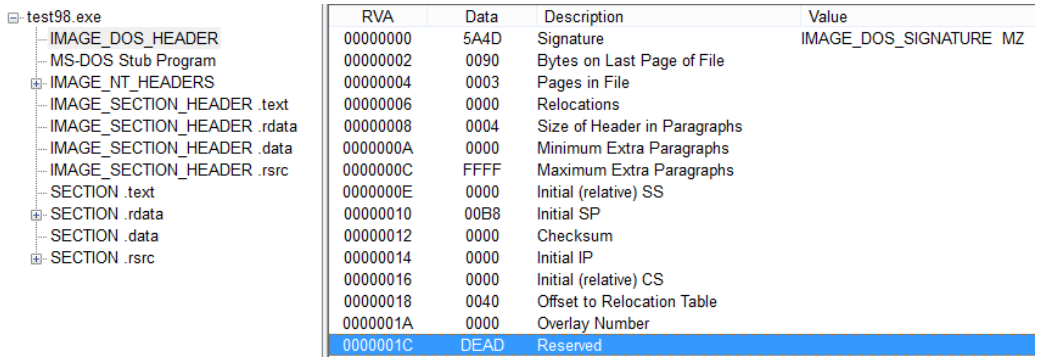
סימן ההדבקה של הוירוס הוא חתימה קטנה שהוירוס משאיר בתהליך ההדבקה בשדה מסוים בתחילת ה-Header של קובץ ה-EXE אשר אינו נמצא בשימוש (כיום) על ידי מערכת ההפעלה ואינו מכיל מידע רלוונטי בשום מצב (בפועל מכיל תמיד אפסים). שדה זה נמצא ב- Offset של 0x1cבקובץ שנמצא בחלק של ה-DOS_HEADER והוא בגודל של 2 בתים.



RVA	Data	Description	Value
00000000	5A4D	Signature	IMAGE_DOS_SIGNATURE MZ
00000002	0090	Bytes on Last Page of File	
00000004	0003	Pages in File	
00000006	0000	Relocations	
00000008	0004	Size of Header in Paragraphs	
0000000A	0000	Minimum Extra Paragraphs	
0000000C	FFFF	Maximum Extra Paragraphs	
0000000E	0000	Initial (relative) SS	
00000010	00B8	Initial SP	
00000012	0000	Checksum	
00000014	0000	Initial IP	
00000016	0000	Initial (relative) CS	
00000018	0040	Offset to Relocation Table	
0000001A	0000	Overlay Number	
0000001C	0000	Reserved	

[DOS_HEADER לפני הדבקה]

הוירוס כותב לשם את הערך 0xDEAD ולכן קריאה מהירה של שדה זה תגיד לנו אם הקובץ כבר נדבק בעבר או לא. אם הקובץ נדבר בעבר אין צורך להמשיך בתהליך ונעבור לקובץ הבא, אם הקובץ נקי, נדביק אותו.



RVA	Data	Description	Value
00000000	5A4D	Signature	IMAGE_DOS_SIGNATURE MZ
00000002	0090	Bytes on Last Page of File	
00000004	0003	Pages in File	
00000006	0000	Relocations	
00000008	0004	Size of Header in Paragraphs	
0000000A	0000	Minimum Extra Paragraphs	
0000000C	FFFF	Maximum Extra Paragraphs	
0000000E	0000	Initial (relative) SS	
00000010	00B8	Initial SP	
00000012	0000	Checksum	
00000014	0000	Initial IP	
00000016	0000	Initial (relative) CS	
00000018	0040	Offset to Relocation Table	
0000001A	0000	Overlay Number	
0000001C	DEAD	Reserved	

[DOS_HEADER לאחר הדבקה]

שמירה של ערך תחילת הקוד המקורי

מכיוון שאני רוצה שהקוד של הווירוס ירוץ לפני התוכנית המקורית, אני מתכוון לשכתב את השדה שמורה ל-Loader של Windows מאיפה להתחיל להריץ את התוכנית ולשים שם את הכתובת של הקוד של הוירוס אותו אני רוצה להזריק לקובץ. לאחר ריצת הוירוס נרצה לקפוץ לקוד המקורי כדי שהתוכנית תרוץ והתהליך יהיה שקוף למשתמש ולכן אני רוצה לשמור את הערך המקורי של השדה הזה. מדובר בשדה Address of Entry Point אשר נמצא במבנה IMAGE_NT_HEADERS->IMAGE_OPTIONAL_HEADER מיקום השדה הוא ב-Offset של 0x120 בתים מתחילת הקובץ.

הזרקת הקוד של הוירוס לקובץ

פעולה זו היא כמובן לב ליבו של הוירוס, ישנן כמה נקודות עדינות ושיקולים להבין לפני שמתכננים את ההזרקה כפי שהיא מתבצעת בקוד.

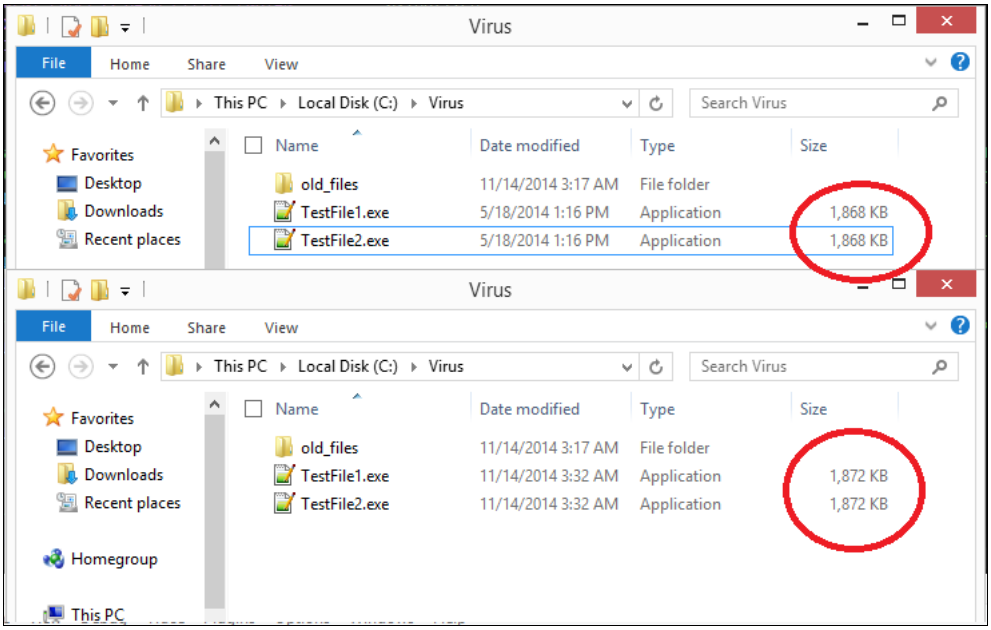
המטרה הסופית שלנו היא להוסיף את הקוד לקובץ הקיים, צריך לעשות זאת מבלי לשנות את מיקום הקוד הקיים וישנן המון צורות לעשות את זה. בקוד שלנו בחרתי בדרך מאוד פשוטה שאינה מצריכה הרבה התעסקות יחסית לדרכים אחרות, החסרונות של דרך פעולה זו היא שהמצאות הוירוס בקוד היא יחסית ברורה והוא די נוח להסרה.

הוירוס פשוט מעתיק את עצמו לסוף הקובץ הקיים. כמובן שפעולה זו בלבד אינה מאפשרת להריץ אותו וצריך לבצע עוד מספר התאמות על מנת להפוך את הוירוס להיות "חלק מ-Image".

בהעתקת הוירוס לסוף הקובץ אני מוסיף את הקוד של הוירוס ל-Section האחרון של הקובץ. במובן הכללי, אני לא יכול לדעת באיזה Section מדובר ולכן צריך לעשות כמה התאמות. התאמה ראשונה היא עדכון השדה VirtualSize של ה-Section, מכיוון שאני מכניס לו עוד 0x1000 בתים של קוד, אגדיל את השדה בערך זה. התאמה חשובה שניה היא התאמת השדה Characteristics כדי שקוד יוכל לרוץ מה-Section בלי בעיות.

ההתאמה האחרונה היא התאמת השדה של גודל ה-Image הכללי ב-Optional Header אשר מוגדל גם הוא ב-0x1000 בתים.

בדיקה פשוטה של התיקייה המכילה מספר קבצי EXE שהעתקתי לשם מראה כיצד הקובץ גדל בדיוק בגודל של 0x1000 בתים לאחר ההדבקה:



סיכום

כאמור, הוירוס נכתב כ-Proof of concept ופרויקט אישי בזמני הפנוי. הקוד המובא במאמר זה מהווה את המכניזם הבסיסי ביותר של וירוס פועל ולא יותר מזה. הוירוס אינו מבצע שום פעולה זדונית ממשית וגם אינו מנסה להסתיר את עצמו עם זאת אני חייב לציין שגרסא חינמית של AVG לא זיהתה את קבצי הוירוס כקוד זדוני. בהנתן המנגנון המובא במאמר זה ניתן בזמן קצר יחסית להכניס פעולות זדוניות כאלו ואחרות לוירוס (למשל פתיחת Socket אשר נותן Reverse shell לחיבור מרחוק) וניתן גם להכניס מנגנוני הסתרה טובים יותר (באמצעות צורות של Packing או הסתרת הקוד ב-Code caves למשל).

לי לקח פרק זמן לא קצר עד שהגעתי למצב שהוירוס פועל ולמדתי הרבה דברים בדרך, אני מקווה שהמאמר הזה מקצר לאנשים אחרים כמוני את הדרך קצת ואשמח לדעת אם מישהו עושה בו שימוש למטרות שונות, משלב חלקים ממנו בפרויקטים אחרים או מעוניין לקיים שת"פ כלשהו לצורך לימוד/מחקר ופיתוח כלשהם (ניתן לצור קשר ב-danb33@gmail.com).

את קוד המקור של הפרוייקט המוצג במאמר זה ניתן להוריד מהכתובת הבאה:

http://digitalwhisper.co.il/files/Zines/0x38/main.c

ביבליוגרפיה

- Kovah, X'. Life Of Binaries Course by Xeno Kovah - http://opensecuritytraining.info/LifeOfBinaries.html
- Kovah, X'. Introduction to x86 - http://opensecuritytraining.info/IntroX86.html
- Kovah, X'. Intermediate x86 Class - http://opensecuritytraining.info/IntermediateX86.html
- Skape/Matt Miller's win32 shellcode tutorial - http://www.hick.org/code/skape/papers/win32-shellcode.pdf