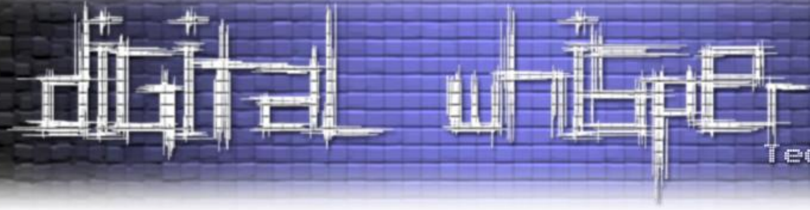


אנטומיה של נוזקת NET. - חלק א'

מאת עמית סרפר

מבוא

בתור חוקרי נוזקות מקצועיים, היינו רוצים לחקור לעומק את התולעת המורכבת ביותר. ובכל פעם שחברת אבטחה כזו או אחרת מפרסמת דו"ח מחקר על נוזקת-על, כזו הממומנת ע"י מעצמת-סייבר, כולנו רצים לקרוא את הדו"ח רק כדי לגלות ששוב פעם - גם האקרים הפועלים בשליחות מעצמה מעדיפים לעשות שימוש בכל מני כלים חיצוניים במקום להשתמש ב-API המובנים במערכת ההפעלה.

אך האמת היא, שעושה רושם שכלל לא צריך להיות האקר מעצמתי כדי ליצור נוזקה שיודעת להשיג את מטרתה בצורה טובה. ודוגמא טובה לכך היא הנוזקה שזכתה לכינוי "Fauxpersky", שנכתבה באמצעות AutoHotKey - כלי שכל מטרתו הוא למכן משימות ידניות, והיא הוכיחה את עצמה כיעילה בגניבת פרטי הזדהויות.

שלא כמו מה שנהוג לחשוב, מחקר נוזקות-על של מעצמות סייבר הם לא מה שחוקרי וירוסים עושים כל היום. המציאות היא שברוב הזמן אנו מעבירים את זמננו במחקר של וירוסים אשר עושים אומנם נזק רב, אך מבחינת מורכבות קוד ופיצ'רים - הם חסרי כמעט כל עניין, עושים שימוש בטכנולוגיה בסיסית ובכלל - ובינוניים מאוד.

קצת על NET.

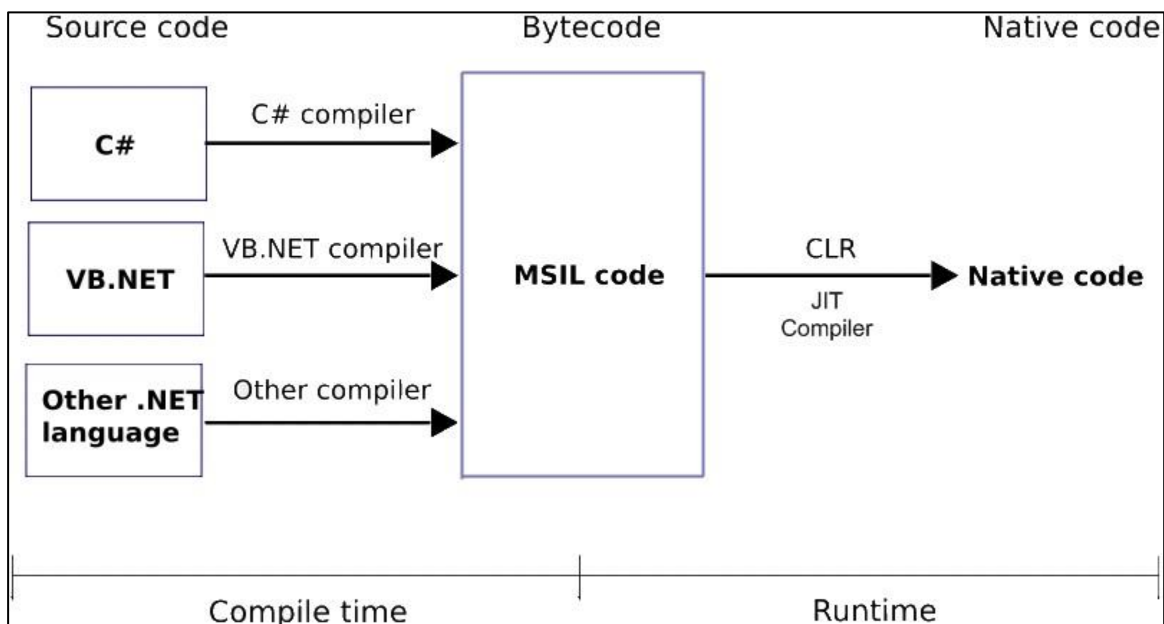
כעת, בואו נשנה את הנושא ונדבר על NET., תשארו איתי - הסיבה לכל ההקדמה שקראתם עד כה תתבהר בקרוב. NET. היא Framework פיתוח שהוצגה ע"י Microsoft בשנת 2000 במטרה להקל על מפתחים. מפתחי NET. יכולים לשכוח מהימים שבהם הם נאלצו להקצות ולשחרר זיכרון כמו ב-C, או כמו כמו ב-++C - למצוא את עצמך כותב קוד ארוך ומסורבל למרות שהתכוונת לכתוב תוכנית קטנה ופשוטה...

השפות הכלולות במשפחת NET., והמוכרת מביניהם - #C, הן שפות מודרניות, פונקציונליות, גנריות, מונחות עצמים ובעצם - כוללות את כל הפיצ'רים מכל ה-Buzzword שאי-פעם הומצאו על שפות מודרניות. השורה התחתונה היא שבעזרת #C, פיתוח למערכת ההפעלה Windows (ובעצם, הדבר נכון גם ל-Linux ול-MacOS, כאשר משתמשים ב-Mono) נעשה ביתר קלות. התחביר זורם ו-Visual Studio מאוד נדיב עם

השלמה האוטומטית וקריאה לפונקציות API של מערכת ההפעלה נעשה בצורה טבעית. בנוסף לכך, כאשר מקמפלים פרוייקט, התוכנית תקומפל ל-EXE (או ל-DLL, תלוי בסוג הפרוייקט).

מי מכם שלא מכיר את השפה בטח שואל את עצמו ברגעים אלו ממש "רגע, אם זאת שפה כל כך פשוטה, וכל כך נוחה, שדואגת לכל מה שאני צריך ועוד בסוף מייצרת לי קובץ EXE, למה עדיין ממשיכים לפתח ב-C או ב-++C? איפה הקאצ'?", אז זהו, שבאמת יש קאצ' - התוצר הסופי של הקמפול הוא אומנם בסיומת EXE, אך הוא לא PE "אמיתי", שלא כמו בינארים שקומפלו באמצעו C או ++C, כאשר תפתחו בינארי שקומפל ב-NET. לא תמצאו שם Opcode-ים של אסמבלי x86, וזה בגלל ש-NET. עובד מעט שונה.

כאשר מקמפלים פורייקט ב-NET., הקוד מתקמפל לשפה המכונה "MSIL" או "[Microsoft Intermediate Language](#)". הקוד עצמו מתקמפל בעצם רק כאשר התוכנה מתחילה לרוץ באמצעות מנוע JIT. אם אתם מעוניינים לשמוע עוד על תהליך הקמפול ב-NET., תרגישו חופשי לקרוא את הדוקומנטציה של מיקרוסופט בקישור הבא: [Microsoft's documentation](#). בינתיים, חשבו על MSIL כאל אסמבלי, פשוט בשכבה גבוהה יותר.

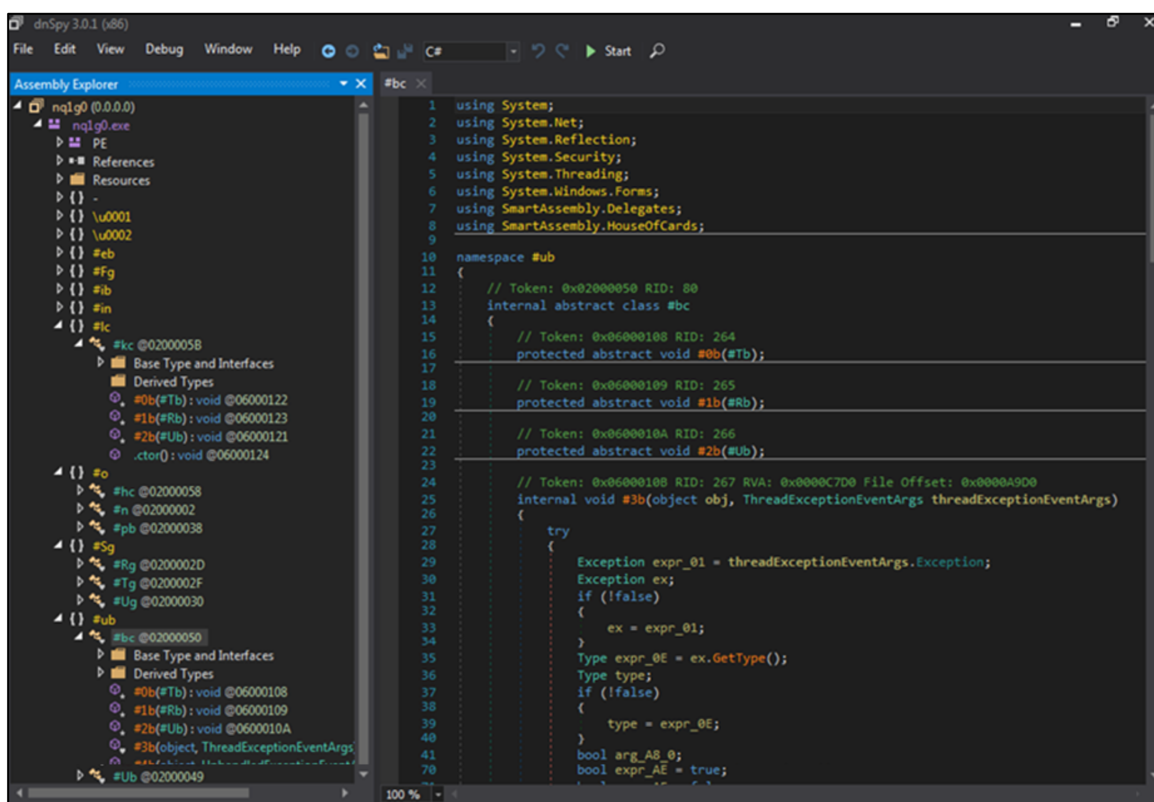


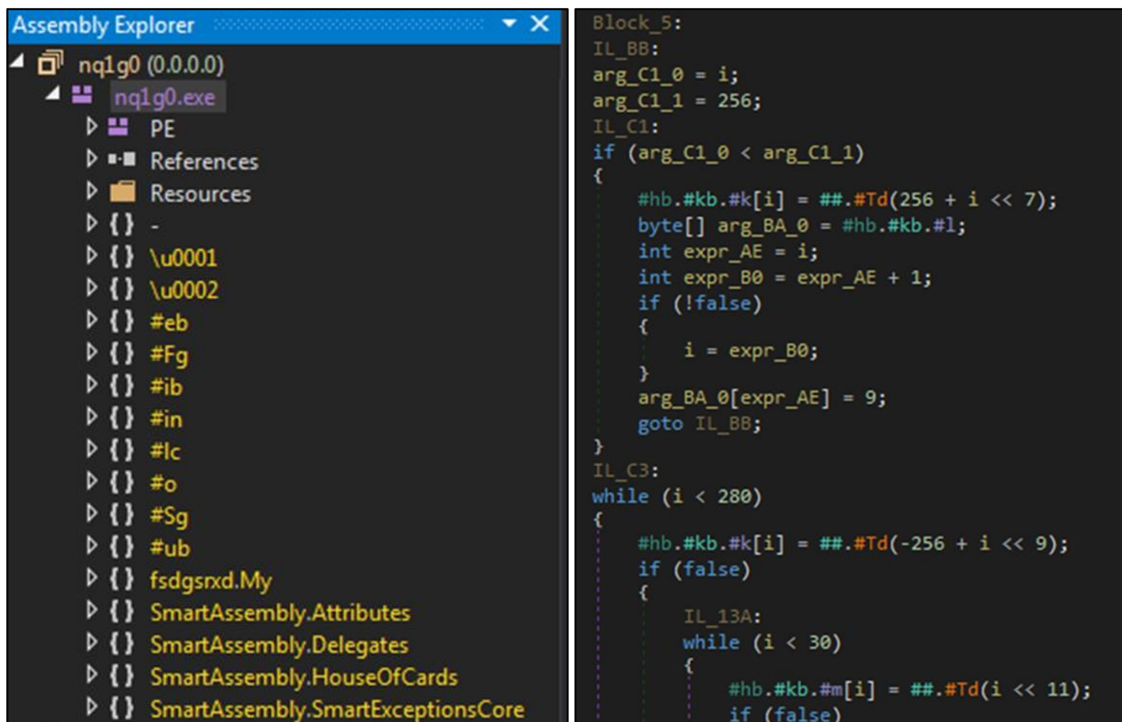
אוקיי, אז למה אני חופר לכם על תהליך הקמפול של NET.? המטרה שלי היא להראות לכם את ההבדל בין האסמבלי שמורץ כאשר מריצים תהליכי C או ++C לבין כאשר מריצים תהליכים אשר נכתבו ב-NET..
כאשר אנו מהנדסים לאחור תוכנות אשר נכתבו באסמבלי "רגיל" (כמו כאלה אשר נכתבו בשפת C/C++) ה-Disassembler יציג לנו אסמבלי x86/64, אך כאשר נהנדס לאחור תוכנות אשר נכתבו באחת ממשפחת שפות ה-NET., התוצר שנראה ב-Disassembler יהיה אסבמלי, אך [שונה לחלוטין ממה שאנו רגילים לראות](#), אל אל דאגה! העובדה שהקוד קומפל ל-MSIL אומרת שבתוכו יש המון metadata שיוכל לעזור לנו רבות בעת ביצוע ה-Decompilation. למעשה, כל מה שאנחנו צריכים זה Decompiler מיוחד לשפות NET. (מכונה "Reflector") ומעט סבלנות.

בעת ביצוע הינדוס לאחור "קלאסי", אנו משתמשים בכלים כגון IDA Pro או WinDBG, אך כאשר נרצה להנדס לאחור תוכנה שכתובה ב-NET., אנחנו נדרש להשתמש בסט כלים שונה לחלוטין, אנו נדרש להשתמש ב-Decompilers יעודיים ל-NET.. אחד ה-Debuggers המעודפים עלי הוא dnSpy, שגם יש לו ממשק משתמש מעולה והוא גם מבוסס על פרוייקט נפלא אחד בשם ILSpy.

שימוש ב-Decompiler כמו dnSpy מאפשר לך לצפות בקוד שיצר את ה-MSIL, שהוא יהיה זהה כמעט לחלוטין לקוד המקורי שבו יצרו את הוירוס (השינויים יהיו בדרך כלל בשמות המשתנים, שמות האובייקטים והמחלקות).

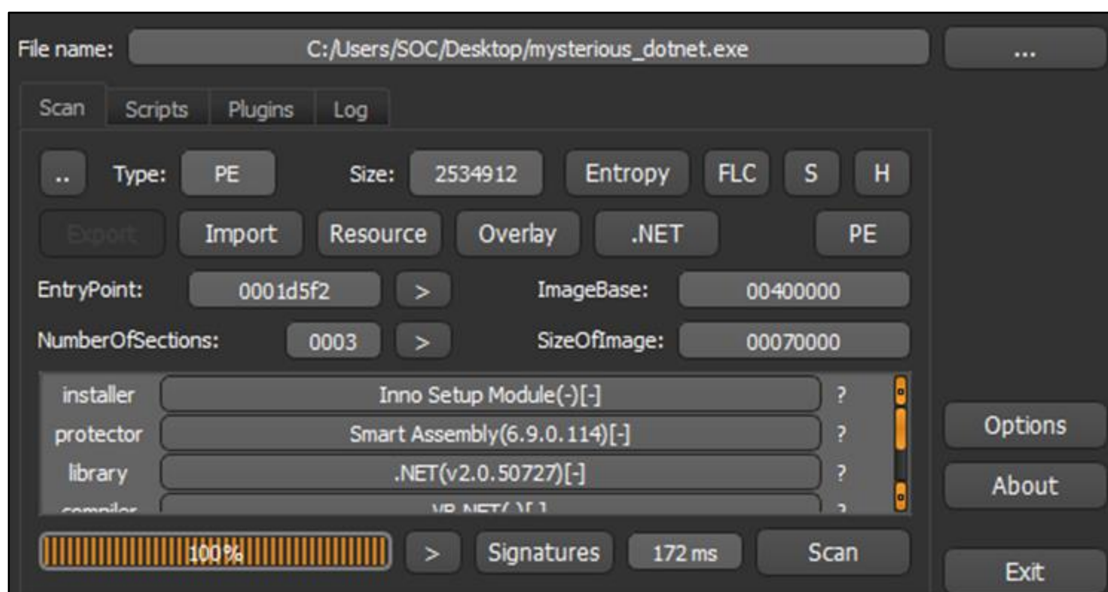
כאשר ביצעתי זאת על הוירוס שחקרתי, ראיתי ששמות המחלקות, המשתנים והפונקציות היו נראים מעט מעורבלים:





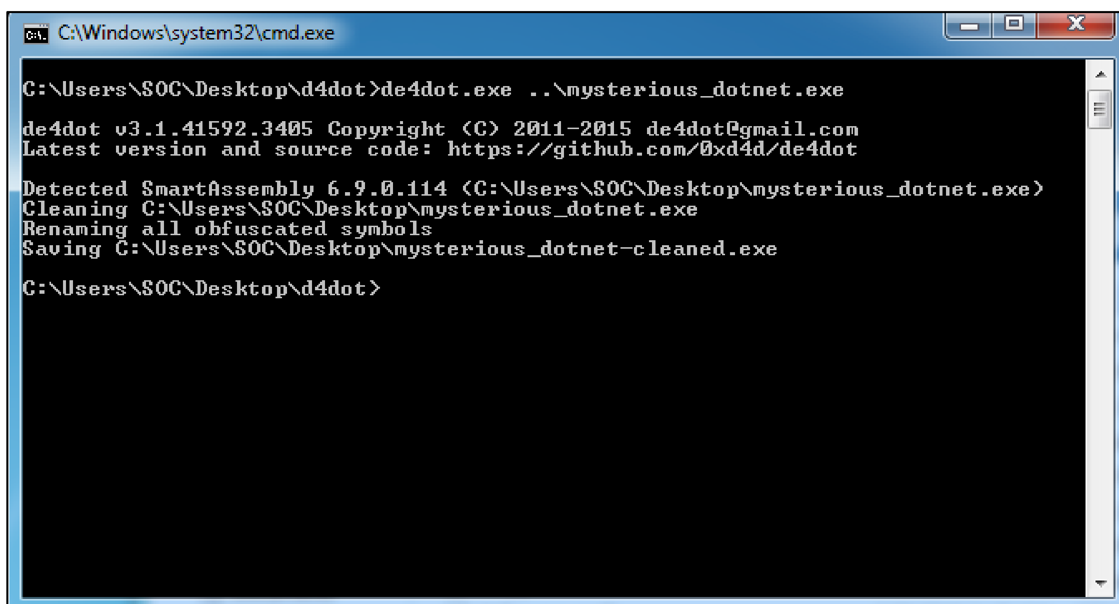
ואכן, בדיוק בגלל הסיבה שניתן לבצע שחזור כמעט מלא לקוד המקורי, כותבים תוכנות רבים (וביניהם כותבי וירוסים) עושים שימוש בכל מני פתרונות אובפסקציה (ערפול) שונות על מנת להקשות על חיי הריוורסרים. אך למזלינו, ישנם לא מעט כלים שיודעים לבצע Deobfuscation ולחסוך לנו שעות צרופות של כיף...

על מנת לעשות את שלב ה-Deobfuscation לפשוט, נשתמש בכלי בשם die (קיצור של "Detect it easy"), וכל שעלינו לעשות הוא פשוט לגרור את הקובץ שברצוננו לעבוד עליו לתוך התוכנה. בעת ביצוע שלב זה נוכל לראות מידע רב אודות הקובץ, בין השאר גם מידע על סוג הערפול שבו נעשה שימוש, במקרה שלנו - נעשה שעשו שימוש ב-Obfuscator בשם "SmartAssembly":

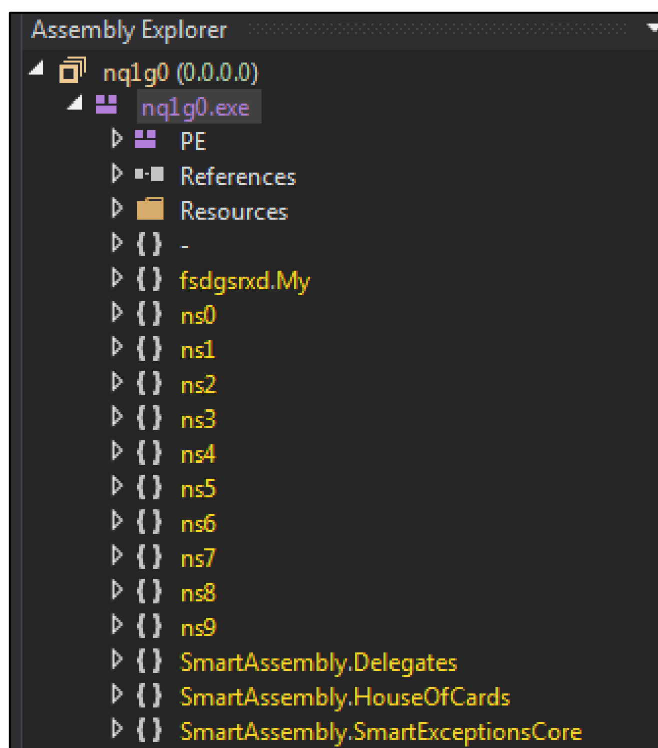


כעת, כשאנו יודעים באיזה כלי עשו שימוש על מנת להגן על הקובץ, אנחנו יכולים להתחיל לחפש שיטות לעקוף אותו. אני ממליץ על שימוש בכלי בשם de4dot שהוא פרוייקט קוד פתוח לביצוע NET Deobfuscation. ו-Unpacker מעולה.

שימוש זריז ב-de4dot כנגד הקובץ שלנו, יראה כך:



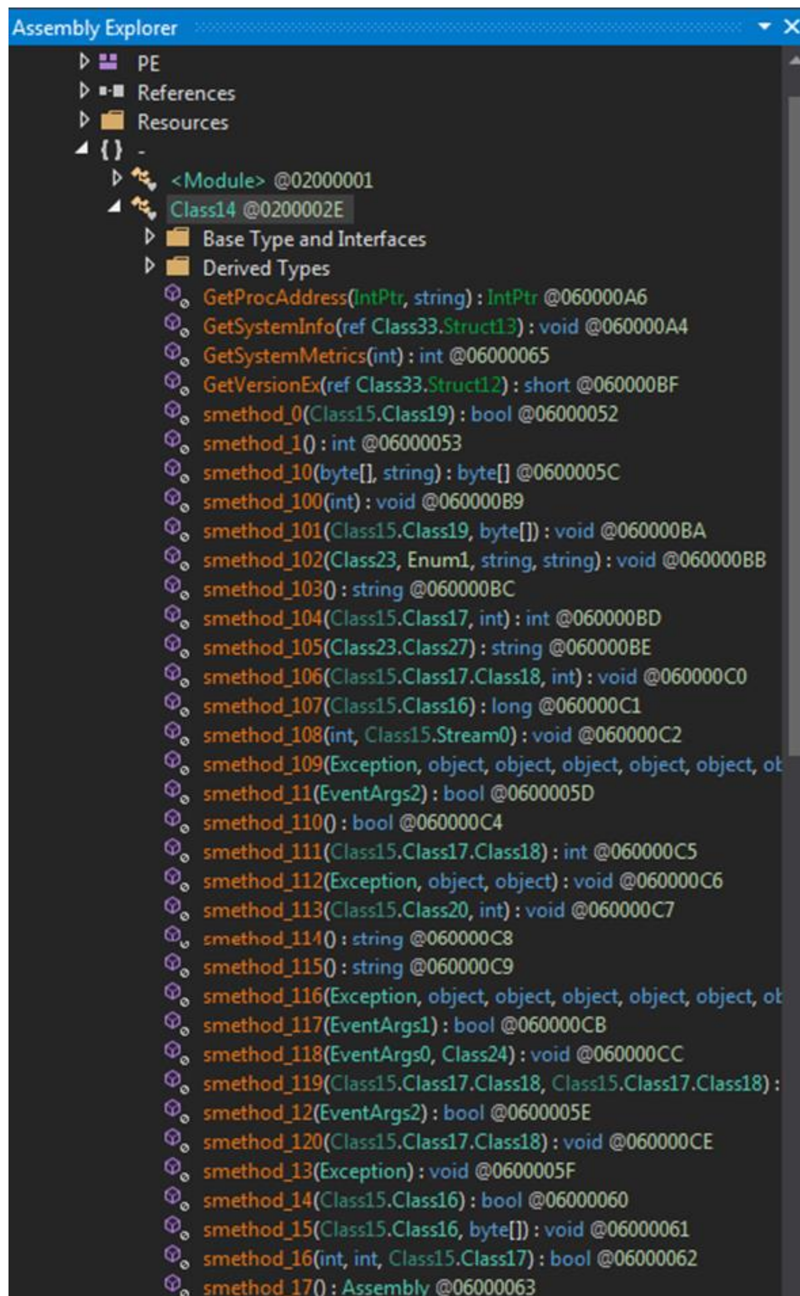
ועכשיו, אחרי הסרת הערפל סביב קוד ה-NET. שיצר את הוירוס שאנחנו חוקרים - אפשר לגשת לעבודה! כעת, נוכל לפתוח את הקובץ "הנקי" שקיבלנו באמצעות dnSpy, אך הפעם נראה שהשמות של המשתנים נראים קצת יותר הגיוניים:



כמה נקודות שחשוב לדעת:

- שמות ה-namespace-ים שאנו רואים כאן הם לא השמות המקוריים. אין לנו שום דרך אמיתית לדעת מה היו שמות המשתנים שבהם כותב הקוד המקורי השתמש.
- הוירוס נכתב במקור ב-Visual Basic, אך מכיוון ש-dnSpy יודע להמיר לנו את קוד ה-MSIL גם ל-VB וגם ל-C#, בחרתי את האפשרות השניה - זאת מכיוון שהתחביר של C# לדעתי מובן ונח יותר לקריאה. ובכלליות - כל קטעי הקוד שתראו במאמר זה הם שחזור C# של קוד MSIL שנוצר מקוד Visual Basic.

כאשר אנו מסתכלים על ה-namespace בשם "-", נוכל לראות מחלקה בשם "Class14" (כמובן - זה שם שנבחר ע"י die ולא השם המקורי), ותחת המחלקה הזו נוכל לראות הרבה מאוד methods:



מהסתכלות על הקוד ב-Class14 ניתן די בקלות לראות שיש כאן ניסיון לביצוע אנומרציה על משתני הסביבה בעזרת שימוש ב-Interaction.Environ:

```
string str = Interaction.Environ(Class14.smethod_76("5g+BxFHXkdTcEM3cEGgk0A==")) +
    Class14.smethod_76("923lYoAhb2vVXIM6u3MCzjKugVDBXZMcbb6ThbsL5r8=");
```

בנוסף, בקלות ניתן לראות שמשתני הסביבה הם שילוב של שתי מחרוזות Base64. כאשר ננסה להמירם בחזרה למחרוזות בבסיס רגיל, נקבל מחרוזת לא מובנת:

```
In [3]: from base64 import b64decode as b64

In [4]: b64("923lYoAhb2vVXIM6u3MCzjKugVDBXZMcbb6ThbsL5r8=")
Out[4]: '\xf7m\xe5b\x80!ok\xd5\\\x83:
\xbbs\x02\xce2\xae\x81P\xc1]\x93\x1cm\xbe\x93\x85\xbb\x0b\xe6\xbf'
```

מהסתכלות על שאר חלקי הקוד, אפשר לראות שימוש בספריות ופונקציות קריפטוגרפיות:

```
static string smethod_56(string string_0)
{
    string password = "nia";
    string s = "cffffffffffffffff";
    string s2 = "@1B2c3D4e5F6g7H8";
    byte[] bytes = Encoding.ASCII.GetBytes(s2);
    byte[] bytes2 = Encoding.ASCII.GetBytes(s);
    byte[] array = Convert.FromBase64String(string_0);
    Rfc2898DeriveBytes rfc2898DeriveBytes = new Rfc2898DeriveBytes(password, bytes2, 2);
    byte[] bytes3 = rfc2898DeriveBytes.GetBytes(32);
    ICryptoTransform transform = new RijndaelManaged
    {
        Mode = CipherMode.CBC
    }.CreateDecryptor(bytes3, bytes);
    MemoryStream memoryStream = new MemoryStream(array);
    CryptoStream cryptoStream = new CryptoStream(memoryStream, transform, CryptoStreamMode.Read);
    byte[] array2 = new byte[checked(array.Length - 1 + 1)];
    int count = cryptoStream.Read(array2, 0, array2.Length);
    memoryStream.Close();
    cryptoStream.Close();
    return Encoding.UTF8.GetString(array2, 0, count);
}
```

```
catch (CryptographicException)
{
    Class22.string_0 = "ERR 2005: The 128-bit encryption is not available on this computer.
      You need to install the High Encryption Pack in order to use the reporting feature.";
    result = null;
    return result;
}
```

קטע הקוד הנ"ל כולל הרבה מאוד "קוד ספגטי" (קוד סבוך שקורא לכל מני קטעי קוד לא רלוונטים וכל מטרתו היא להקשות על החוקר לבצע ניתוח סטאטי). בנקודה זו, אפשר להניח בלב די שלם שתוכן מחרוזת ה-Base64 הוא מוצפן, ולכן בשלב זה יש לנו שתי אופציות:

- ביצוע ניתוח סטאטי: כתיבת קוד זריז שיפענח את המחרוזות המוצפנות בעזרת שימוש מפתחות שניתן למצוא די בקלות בקוד.
- ביצוע ניתוח דינאמי: הרצת התוכנית תחת דיבאגר עד לרגע בו התוכנית תפענח את המחרוזות הנ"ל בעצמה לנגד עיננו.

אני בהחלט אבחר באופציה מספר 2 במקרים כאלה, שימוש בדיבאגר יהיה הרבה יותר מהיר במצבים כאלה. מכיוון ש-dnSpy הוא גם דיבאגר, יהיה קל מאוד להכניס BreakPoint בדיוק בשלב בו התוכנית מגדירה את מפתחות הפענוח והרצתה באמצעות F9 עד לשלב זה:

```
1078    // Token: 0x0600008B RID: 139 RVA: 0x000047E4 File Offset: 0x000029E4
1079    static string smethod_56(string string_0)
1080    {
1081        string password = "nia";
1082        string s = "cffffffffffffffff";
1083        string s2 = "@1B2c3D4e5F6g7H8";
1084        byte[] bytes = Encoding.ASCII.GetBytes(s2);
1085        byte[] bytes2 = Encoding.ASCII.GetBytes(s);
1086        byte[] array = Convert.FromBase64String(string_0);
1087        Rfc2898DeriveBytes rfc2898DeriveBytes = new Rfc2898DeriveBytes(password, bytes2, 2);
1088        byte[] bytes3 = rfc2898DeriveBytes.GetBytes(32);
1089        ICryptoTransform transform = new RijndaelManaged
1090        {
1091            Mode = CipherMode.CBC
1092        }.CreateDecryptor(bytes3, bytes);
1093        MemoryStream memoryStream = new MemoryStream(array);
1094        CryptoStream cryptoStream = new CryptoStream(memoryStream, transform, CryptoStreamMode.Read);
1095        byte[] array2 = new byte[checked(array.Length - 1 + 1)];
1096        int count = cryptoStream.Read(array2, 0, array2.Length);
1097        memoryStream.Close();
1098        cryptoStream.Close();
1099        return Encoding.UTF8.GetString(array2, 0, count);
1100    }
```

כמו שניתן לראות, הצבתי BreakPoint בשורה 1081, שהיא ההתחלה של smethod_56, ואפשר לראות שהפונקציה מקבל פרמטר - string_0, ואפשר לראות שאותו פרמטר בעל ערך base64 מוצפן:

Locals

Name	Value	Type
string_0	"Sk1N1W/kLIYPS5rz2GRFew=="	string
array	null	byte[]
cryptoStream	null	System.Security.Cryptography.Cry...
count	0x00000000	int
transform	null	System.Security.Cryptography.ICr...
V_4	"SHA1"	string
bytes	null	byte[]
bytes3	null	byte[]
V_7	0x00000000	int
memoryStream	null	System.IO.MemoryStream
s2	"@1B2c3D4e5F6g7H8"	string
password	"nia"	string

חשוב לשים לב שכשאנחנו עושים Step Over על הקוד (בעזרת שימוש ב-F10), אנחנו יכולים לראות מספר אובייקטים שאוכלסו במידע:

Name	Value	Type
string_0	"Sk1N1W/kLlYPS5rz2GRFew=="	string
array	{byte[0x00000010]}	byte[]
cryptoStream	{System.Security.Cryptography.CryptoStream}	System.Security.Cryptography.Cry...
count	0x00000008	int
transform	{System.Security.Cryptography.RijndaelManagedTransform}	System.Security.Cryptography.ICr...
V_4	"SHA1"	string
bytes	{byte[0x00000010]}	byte[]
bytes3	{byte[0x00000020]}	byte[]
V_7	0x00000100	int
memoryStream	{System.IO.MemoryStream}	System.IO.MemoryStream
s2	"@1B2c3D4e5F6g7H8"	string
password	"nia"	string
s	"cffffffffffffffff"	string
rfc2898DeriveBytes	{System.Security.Cryptography.Rfc2898DeriveBytes}	System.Security.Cryptography.Rfc...
V_13	0x00000002	int
V_14	null	string
array2	{byte[0x00000010]}	byte[]
bytes2	{byte[0x00000013]}	byte[]
V_17	null	string
V_18	{System.Security.Cryptography.RijndaelManaged}	System.Security.Cryptography.Rijn...
V_19	null	object[]
V_20	null	string

זה מאפשר לנו להבין את תפקידה של smethod_56 בקלות: היא יוצרת מערכים בני 3 בתים מתוך 3 משתנים שונים:

- S2 (value: "1B2c3D4e5F6g7H8")
- S (value: "cffffffffffffffff")
- String_0: "Sk1N1W/kLlYPS5rz2GRFew==")

לאחר מכן, היא מאתחלת את המחלקה rfc2898DeriveBytes ומעבירה לה שלושה פרמטרים:

- Password - עם הערך "nia" (אפשר לראות זאת בתמונת המסך מעל) - ישמש כסיסמה.
- Bytes2 - מערך של בתים שנוצרו בעזרת המשתנה s - ישמש כ-Salt.
- הסיפרה "2" - מספר האיטרציות להפקת המפתח

הפונקציה, לאחר מכן, יוצאת מערכת בתים נוסף, בשם bytes3, מערך אשר יחזיק את המפתח שנוצר זה עתה. לאחר מכן, הפונקציה ממשיכה ומאתחלת את המחלקה RijndaelManaged ותקרא לפונקציה CreateDecryptor עם המשתנים bytes3 ו-bytes שיהוו את המפתח וה-IV, בהתאמה.

אם נסתכל בהמשך הקוד, נוכל לראות שיש עוד מספר מחרוזות מוצפנות ומעורפלות שמתפענחות באמצעות קריאה לפונקציה smethod_0 עם הפרמטרים הנדרשים לתהליך הפענוח:

```
static bool smethod_80(object[] object_0)
{
    Class4.Class5 @class = new Class4.Class5();
    Class7.Delegate1 @delegate = Class4.smethod_0<Class7.Delegate1>(Class14.smethod_56("Sk1N1W/kLlYPS5rz2GRFew=="), Class14.smethod_56("ABUKocXA/8U8/dTyQqSgxw=="));
    Class7.Delegate2 delegate2 = Class4.smethod_0<Class7.Delegate2>(Class14.smethod_56("Sk1N1W/kLlYPS5rz2GRFew=="), Class14.smethod_56("0r4Q+KffRShOMrJXQ5bIPBPfs2ueDUQLhElRpvSNOD8="));
    Class7.Delegate3 delegate3 = Class4.smethod_0<Class7.Delegate3>(Class14.smethod_56("Sk1N1W/kLlYPS5rz2GRFew=="), Class14.smethod_56("Fk2eZIMApwcIWTEmb2cS+hpgxP8KTAVFVbM4GPTL2b4="));
    Class7.Delegate4 delegate4 = Class4.smethod_0<Class7.Delegate4>(Class14.smethod_56("Sk1N1W/kLlYPS5rz2GRFew=="), Class14.smethod_56("sBgrGtKz4j+pZ39VniTPP4yPMBN4WHVY4IS8ZY4fmSE="));
    Class7.Delegate5 delegate5 = Class4.smethod_0<Class7.Delegate5>(Class14.smethod_56("pdd3zZ5lwXtj8hV1GKRfVw=="), Class14.smethod_56("2duPdAetIoKQy5xe5y0s2W+3unGrWO/nhF7+U9Kw7MI="));
    Class7.Delegate6 delegate6 = Class4.smethod_0<Class7.Delegate6>(Class14.smethod_56("pdd3zZ5lwXtj8hV1GKRfVw=="), Class14.smethod_56("RxKVVbtH14pgjdXaw2C1YbWUQxywTFbqf/pZBB51J5c="));
    Class7.Delegate7 delegate7 = Class4.smethod_0<Class7.Delegate7>(Class14.smethod_56("Sk1N1W/kLlYPS5rz2GRFew=="), Class14.smethod_56("TgOhAyN9YiqdYirD3QytuA=="));
    Class7.Delegate8 delegate8 = Class4.smethod_0<Class7.Delegate8>(Class14.smethod_56("pdd3zZ5lwXtj8hV1GKRfVw=="), Class14.smethod_56("qt4tHmTORXFlu08c+szp/A=="));
```

לאחר שכל אחד מהערכים מתפענח, נוכל לראות את ערכם המקורי בחלונית המשתנים:

```
1996    bool flag2 = (bool)object_0[3];
1997    string text2 = (string)object_0[4];
1998    int num = 0;
1999    string text3 = string.Format("\"{0}\"", text2);
2000    Class7.Struct1 @struct = default(Class7.Struct1);
2001    @class.struct0_0 = default(Class7.Struct0);
2002    @struct.uint_0 = Convert.ToUInt32(Marshal.SizeOf(typeof(Class7.Struct1)));
2003    bool flag4;
2004    try
2005    {
2006        Class4.Class5.Class6 class2 = new Class4.Class5.Class6();
2007        class2.class5_0 = @class;
2008        if (!string.IsNullOrEmpty(text))
2009        {
2010            text3 = text3 + " " + text;
```

0 %

cals

ame	Value	Ty
object_0	{object[0x00000005]}	obj
delegate6	{ns1.Class7.Delegate6}	ns1.
text2	"C:\\Users\\SOC\\Desktop\\mysterious_dotnet-cleaned.exe"	strin
text3	null	strin
delegate2	{ns1.Class7.Delegate2}	ns1.
num	0x00000000	int

במקרה שלנו - text2 מכיל את הנתיב המלא לבינארי עצמו. ובהמשך, נוכל לראות עוד ועוד מחרוזות מתפענחות, אותן מחרוזות שופכות עוד אור על התנהגותו של הוירוס.

בפיסת הקוד הבאה נוכל לראות עוד מספר מחרוזות מוצפנות אשר מפוענחות בעזרת פונקציה שונה (אך דומה) - smethod_76. אגב, שימו לב שמפתח ההצפנה הוא בסינית:

```
  num2 = 28;
  string sourceFileName = Interaction.Environ(Class14.smethod_76("rRhnphBugUiRcVlpVgLfjw==")) +
    Class14.smethod_76("ijulUbn8DPPkee8Mdv0Pf3JPXTMNWvYRORO+JfoPSAU=") +
    Class14.smethod_76("8Ebrvyc8jIJpnps2eCCHYA==");
```

```
static string smethod_76(string string_0)
{
    string s = "朋软京贵见七零叫";
    RijndaelManaged rijndaelManaged = new RijndaelManaged();
    MD5CryptoServiceProvider mD5CryptoServiceProvider = new MD5CryptoServiceProvider();
    string result;
    try
    {
        byte[] array = new byte[32];
        byte[] sourceArray = mD5CryptoServiceProvider.ComputeHash(Encoding.ASCII.GetBytes(s));
        Array.Copy(sourceArray, 0, array, 0, 10);
        Array.Copy(sourceArray, 0, array, 15, 10);
        rijndaelManaged.Key = array;
        rijndaelManaged.Mode = CipherMode.ECB;
        ICryptoTransform cryptoTransform = rijndaelManaged.CreateDecryptor();
        byte[] array2 = Convert.FromBase64String(string_0);
        string @string = Encoding.ASCII.GetString(cryptoTransform.TransformFinalBlock(array2, 0, array2.Length));
        result = @string;
    }
    catch (Exception expr_8C)
    {
        ProjectData.SetProjectError(expr_8C);
        ProjectData.ClearProjectError();
    }
    return result;
}
```

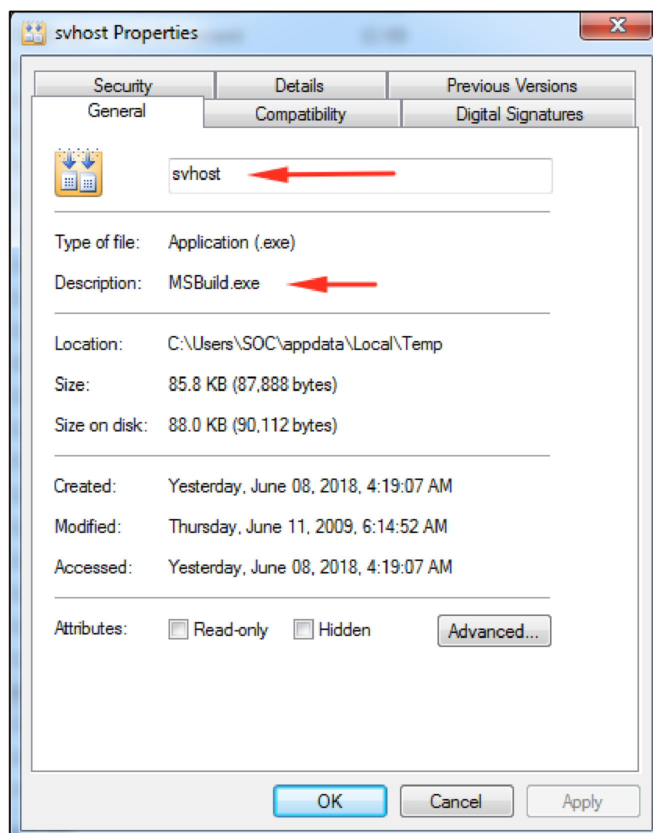
אם נסתכל שנית בחלונית המשתנים, נוכל לראות עוד נתיבים שמתפענחים, נוכל לראות שהמשתנה sourceFileName מכיל את הערך של msbuild.exe ו-destFileName מאוכלס עם הערך svhost.exe שנמצא בתיקיה הזמנית של המשתמש (%temp%):

array2	null	byte[]
str2	"C:\\Users\\SOC\\AppData\\Local\\Temp"	string
V_8	""	string
destFileName	"C:\\Users\\SOC\\AppData\\Local\\Temp\\svhost.exe"	string
text	"C:\\Users\\SOC\\AppData\\Roaming\\"	string
str	"C:\\Users\\SOC\\AppData\\Local\\Temp\\"	string
flag	false	bool
byte_	{byte[0x00003601]}	byte[]
resourceManager	{System.Resources.ResourceManager}	System.Resources.ResourceManag...
array3	{byte[0x00003600]}	byte[]
sourceFileName	"C:\\Windows\\Microsoft.NET\\Framework\\v3.5\\msbuild.exe"	string

אם נסתכל על הקוד, נוכל לראות שמתבצעת כאן פעולת העתקה:

```
File.Copy(sourceFileName, destFileName, true);
```

מה שאומר שהקובץ svhost.exe שנוצר בתיקיה הזמנית של המשתמש הוא בעצם MSBuild.exe, ומפני שאנו מדבגים את הוירוס בזה הרגע, נוכל לגשת לתיקיה הנ"ל לבכון את MSBuild.exe בעצמנו. ולכן התמונה הבאה לא מפתיעה אותנו כל כך:



השימוש ב-MSBuild נועד כדי לבנות בינארי קטן יותר שבעזרת מספר מניפולציות ב-Registry יאפשר לוירוס לשרוד Reboot של מערכת ההפעלה.

סגירת תהליכים ע"י הזקרת קוד

נקודה מעניינת נוספת ששמתי לב אליה בהתנהגות הוירוס הזה, היא שברגע שפתחתי תהליכים כגון Procmon או Process Hacker והרצתי את שורה 2049 ב-class_14, שמתי לב שכלל כלי המחקר שבהם השתמשתי פשוט נעצרו והפסיקו לפעול. מעניין מאוד! איך זה קורה? הייתי חייב לברר:

בדרך כלל, כאשר וירוסים רוצים לסגור תהליכים של כלי מחקר, הם כוללים בתוכם רשימה של כל מני שמות תהליכים כגון "Wireshark, וכלים מתוך חבילת Sysinternals), דוגמים את רשימת התהליכים הפעילים כל מספר שניות ופשוט קוראים ל-()TerminalProcess ברגע שהם מוצאים התאמה.

המקרה שלנו שונה - לא הצלחתי למצוא שום שלב שבו מתבצעת קריאה ישירה או עקיפה (באמצעות ()GetProcAddress) לפונקציה ()TerminateProcess.

כאשר דיבאגתי את הוירוס, שמתי לב שלאחר ש-ProcessHacker נסגר, לא יכולתי לפתוח אותו שנית עד סגירת הוירוס. טכניקה זו, כמובן, נועדה למרר את חייו של החוקר ולהקשות על שלב הניתוח הדינאמי של הוירוס.

שיטה נפוצה לבצע טריק זה, היא באמצעות שימוש בפונקציה()CreateToolHelp32Snapshot, פונקציה זו תקח "Snapshot" של התהליך יחד עם ה-Heap הנוכחי שלו, המודולים הטעונים אליו ועוד מידע נוסף.

במקרה שלנו, הוירוס היה לוקח את רשימת כל התהליכים הרצים וקורה עבור כל אחד מהם לפונקציה זו, התוצאה של הקריאה הנ"ל נשמרת במערך. כל תא במערך מחזיר מידע עבור אחד מהתהליכים שרצים בזה עתה:

[9]	{System.Diagnostics.Process (msdtc)}	System.Diagnostics.Process
[10]	{System.Diagnostics.Process (explorer)}	System.Diagnostics.Process
[11]	{System.Diagnostics.Process (chrome)}	System.Diagnostics.Process
[12]	{System.Diagnostics.Process (chrome)}	System.Diagnostics.Process
[13]	{System.Diagnostics.Process (chrome)}	System.Diagnostics.Process
[14]	{System.Diagnostics.Process (svchost)}	System.Diagnostics.Process
[15]	{System.Diagnostics.Process (svchost)}	System.Diagnostics.Process
[16]	{System.Diagnostics.Process (svchost)}	System.Diagnostics.Process
[17]	{System.Diagnostics.Process (conhost)}	System.Diagnostics.Process
[18]	{System.Diagnostics.Process (winlogon)}	System.Diagnostics.Process
[19]	{System.Diagnostics.Process (vmtoolsd)}	System.Diagnostics.Process
[20]	{System.Diagnostics.Process (WmiPrvSE)}	System.Diagnostics.Process
[21]	{System.Diagnostics.Process (VGAuthService)}	System.Diagnostics.Process
[22]	{System.Diagnostics.Process (svchost)}	System.Diagnostics.Process
[23]	{System.Diagnostics.Process (svchost)}	System.Diagnostics.Process
[24]	{System.Diagnostics.Process (dwm)}	System.Diagnostics.Process
[25]	{System.Diagnostics.Process (chrome)}	System.Diagnostics.Process
[26]	{System.Diagnostics.Process (svchost)}	System.Diagnostics.Process
[27]	{System.Diagnostics.Process (spoolsv)}	System.Diagnostics.Process
[28]	{System.Diagnostics.Process (svchost)}	System.Diagnostics.Process
[29]	{System.Diagnostics.Process (SearchIndexer)}	System.Diagnostics.Process
[30]	{System.Diagnostics.Process (cmd)}	System.Diagnostics.Process
[31]	{System.Diagnostics.Process (lsm)}	System.Diagnostics.Process

ברגע שאחד מהתאים כולל תהליך שקשור לפעולת מחקר, הוירוס מפענח את קטע הקוד הבא (אותו קטע מוצפן שמור בתוך הוירוס עצמו) ואז מזריק אותו לתוך אותו תהליך מחקר. אותו קטע קוד קורא לפונקציה ()NtTerminateProcess ובכך גורם לסגירת התהליך:

```
pd_rec0:00101050                 dd 5 dup(0)
pd_rec0:00101064 word_101064     dw 0                    ; DATA XREF: pd_rec0:off_101079↓o
pd_rec0:00101066                 db 'GetCurrentThreadId',0
pd_rec0:00101079 ;
pd_rec0:00101079 ; Import names for KERNEL32.dll
pd_rec0:00101079 ;
pd_rec0:00101079 off_101079      dd rva word_101064      ; DATA XREF: pd_rec0:__IMPORT_DESCRIPTOR_KERNEL32↑o
pd_rec0:0010107D                 dd 0
pd_rec0:00101081                 align 4
pd_rec0:00101084                 dd 0
pd_rec0:00101088                 db 0
pd_rec0:00101089 aKernel32Dll    db 'KERNEL32.dll',0     ; DATA XREF: pd_rec0:0010100C↑o
pd_rec0:00101096 word_101096     dw 0                    ; DATA XREF: pd_rec0:off_1010A9↓o
pd_rec0:00101098                 db 'GetCurrentThread',0
pd_rec0:001010A9 ;
pd_rec0:001010A9 ; Import names for KERNEL32.dll
pd_rec0:001010A9 ;
pd_rec0:001010A9 off_1010A9      dd rva word_101096      ; DATA XREF: pd_rec0:00101014↑o
pd_rec0:001010AD                 dd 0
pd_rec0:001010B1                 align 4
pd_rec0:001010B4                 dd 0
pd_rec0:001010B8                 db 0
pd_rec0:001010B9 aKernel32Dll_0  db 'KERNEL32.dll',0     ; DATA XREF: pd_rec0:00101020↑o
pd_rec0:001010C6 word_1010C6     dw 0                    ; DATA XREF: pd_rec0:off_1010D6↓o
pd_rec0:001010C8                 db 'SuspendThread',0
pd_rec0:001010D6 ;
pd_rec0:001010D6 ; Import names for KERNEL32.dll
pd_rec0:001010D6 ;
pd_rec0:001010D6 off_1010D6      dd rva word_1010C6      ; DATA XREF: pd_rec0:00101028↑o
pd_rec0:001010DA                 dd 0
pd_rec0:001010DE                 align 10h
pd_rec0:001010E0                 dd 0
pd_rec0:001010E4                 db 2 dup(0)
pd_rec0:001010E6 aKernel32Dll_1  db 'KERNEL32.dll',0     ; DATA XREF: pd_rec0:00101034↑o
pd_rec0:001010F3 word_1010F3     dw 0                    ; DATA XREF: pd_rec0:off_101108↓o
pd_rec0:001010F5                 db 'NtTerminateProcess',0
pd_rec0:00101108 ;
pd_rec0:00101108 ; Import names for ntdll.dll
pd_rec0:00101108 ;
pd_rec0:00101108 off_101108      dd rva word_1010F3      ; DATA XREF: pd_rec0:__IMPORT_DESCRIPTOR_ntdll↑o
pd_rec0:0010110C                 dd 0
pd_rec0:00101110                 dd 2 dup(0)
pd_rec0:00101118 aNtdllDll       db 'ntdll.dll',0        ; DATA XREF: pd_rec0:00101048↑o
pd_rec0:00101122                 align 1000h
pd_rec0:00101122 pd_rec0         ends
pd_rec0:00101122
pd_rec0:00101122
pd_rec0:00101122                 end DllEntryPoint
```

סיכום

חלק זה מסכם את השלב הראשון במחקר שעשיתי אודות הוירוס הנ"ל. בחלק זה כיסינו את הדרך בה יש לגשת בעת ביצוע הנדסה לאחור של פרוייקט NET.-י, הצגנו מספר כלים ומספר טכניקות שבהם ניתן להשתמש על מנת להתגבר על מכשולים נופצים. בנוסף, ענינו על שתי שאלות מעניינות וחשובות: איך ולמה כלי המחקר שלנו נסגרו בזמן המחקר עצמו ע"י הוירוס.

בחלק הבא נכסה מה מטרתו של הוירוס, אילו קבצים הוא מושך מהאינטרנט ומריץ על המחשב, ומה ניתן ללמוד עליהן בעזרת ביצוע ניתוח סטאטי.