Dcrat Malware Analysis - How to Manually Decode a 3-Stage Malware Sample

By Matthew

Published: 2023-04-08 · Archived: 2026-04-05 16:14:01 UTC

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.

Tooling

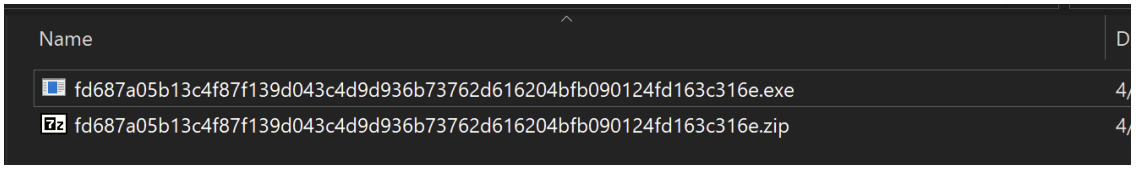
- Detect-it-easy - Quick initial analysis of pe-files
- Dnspy - Analysis, decompilation and debugging of .NET files
- Cyberchef - Interactive tool for prototyping decoders

Samples

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

Initial Analysis

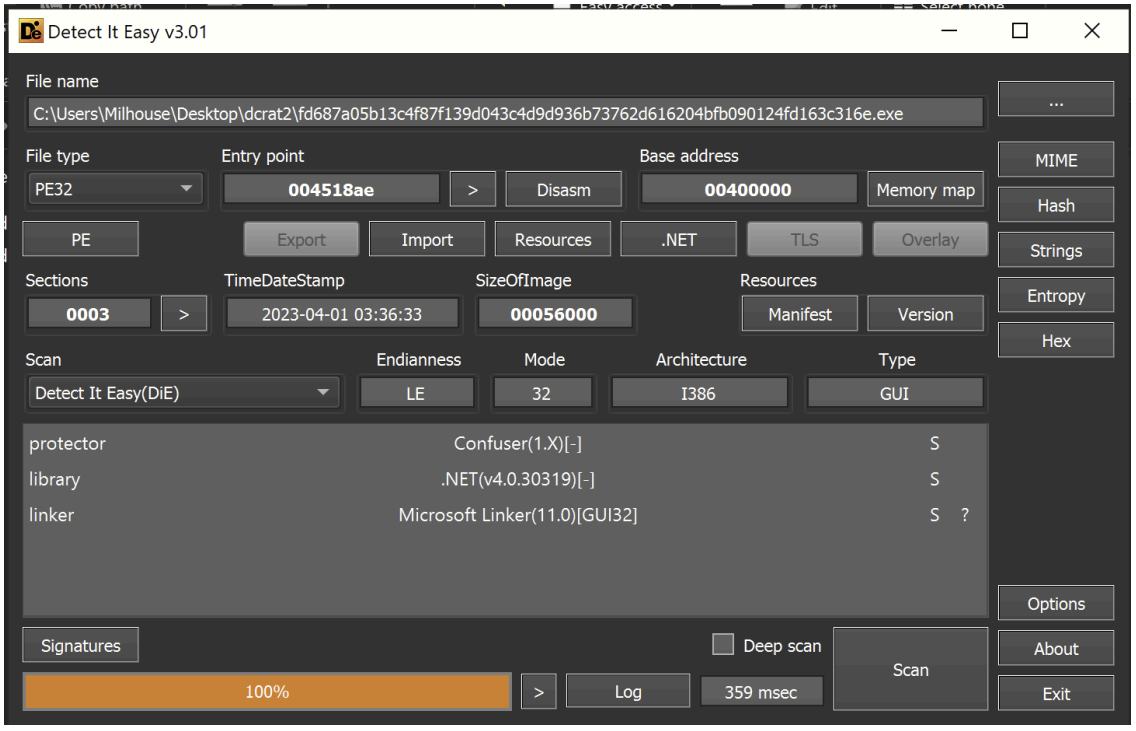
The initial file can be downloaded via Malware Bazaar and unzipped using the password `infected`



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

> Pe-studio is also a great option but we 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.

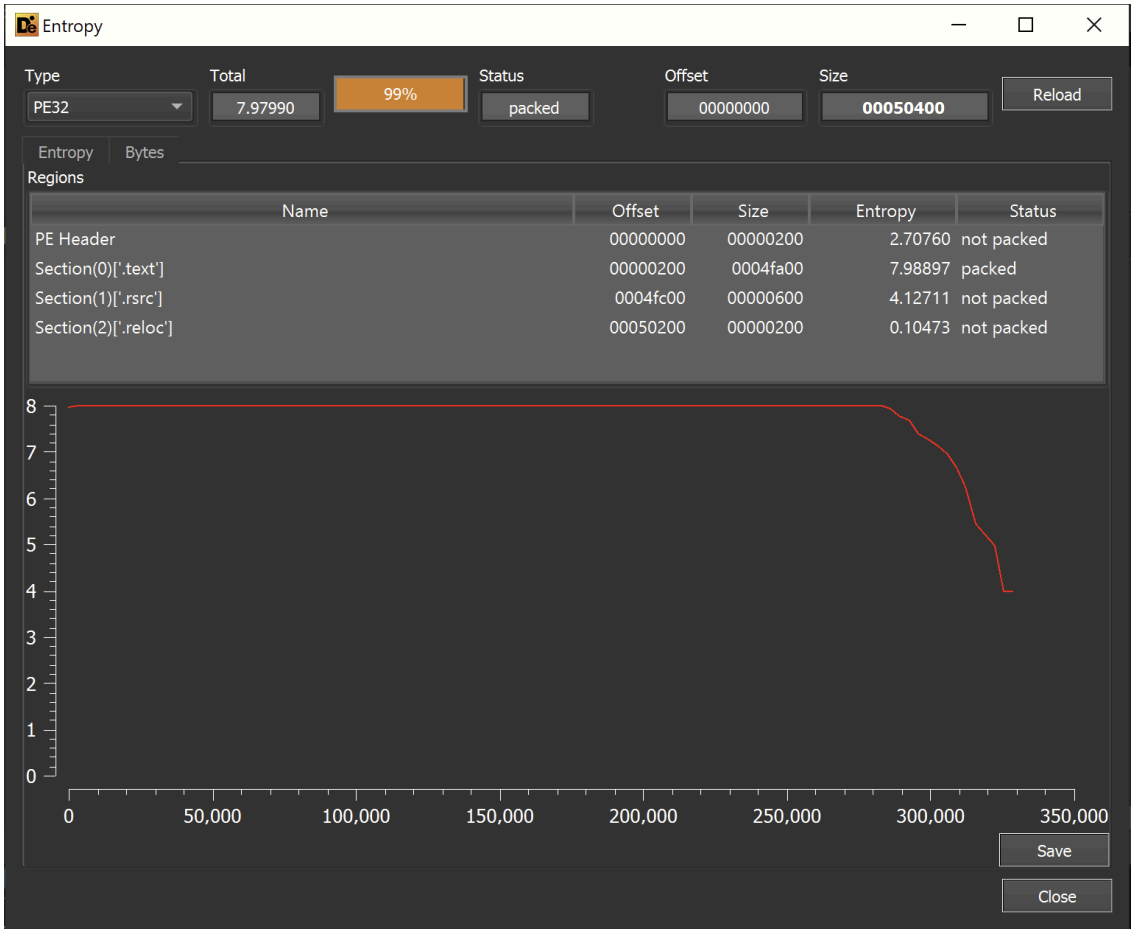


Initial analysis using Detect-it-easy

Before proceeding, we 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.



Entropy Analysis of the Initial .exe file - Showing a large section of high entropy

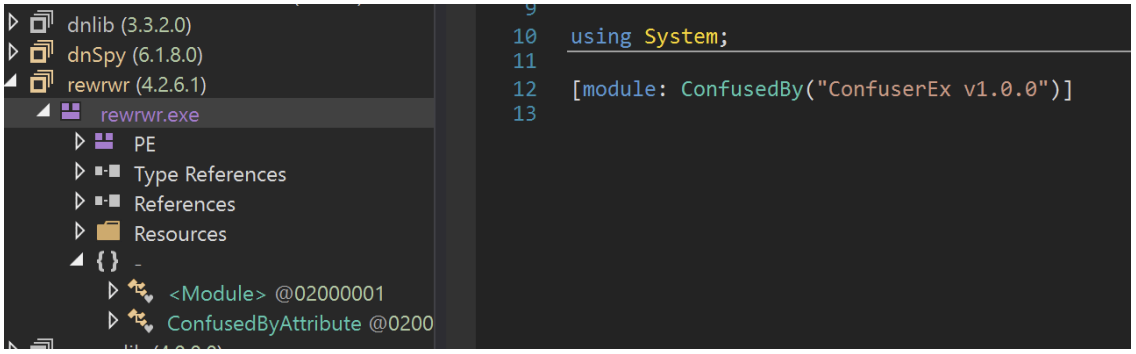
The entropy graph revealed that a significant portion of the file has an entropy of `7.98897` (This is very high, the maximum value is 8).

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

To analyze the suspected loader, we moved on to `Dnspy`

Dnspy Analysis

Utilizing Dnspy, we 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.

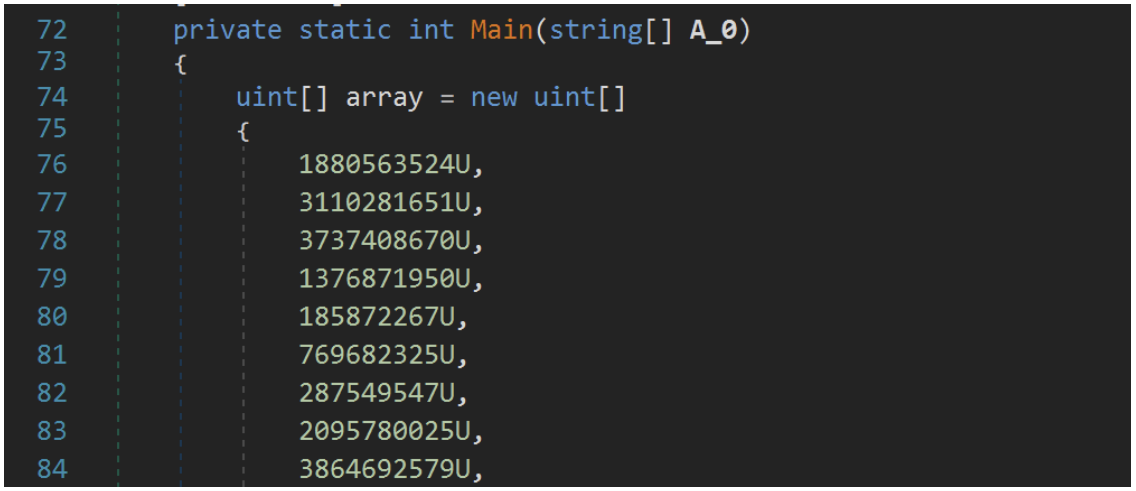


Dnspy overview of the initial file

To peek at the code being executed - we 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)



```
72      private static int Main(string[] A_0)
73      {
74          uint[] array = new uint[]
75          {
76              1880563524U,
77              3110281651U,
78              3737408670U,
79              1376871950U,
80              185872267U,
81              769682325U,
82              287549547U,
83              2095780025U,
84              3864692579U,
```

Viewing Encrypted Arrays using Dnspy

```
7   // Token: 0x02000001 RID: 1
8   internal class <Module>
9   {
10      // Token: 0x06000001 RID: 1 RVA: 0x0004F0D4 File Offset: 0x0004D2D4
11      private static GCHandle Decrypt(uint[] A_0, uint A_1)
12      {
13          uint[] array = new uint[16];
14          uint[] array2 = new uint[16];
15          ulong num = (ulong)A_1;
16          for (int i = 0; i < 16; i++)
17          {
```

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, we 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`

```
10071            3821091257U,
10072            283078313U,
10073            732224790U,
10074            2882807258U,
10075            "Not showing all elements because this array is too big (78747 elements)"
10076        };
10077        Assembly executingAssembly = Assembly.GetExecutingAssembly();
10078        Module manifestModule = executingAssembly.ManifestModule;
10079        GCHandle gchandle = <Module>.Decrypt(array, 306067877U);
10080        byte[] array2 = (byte[])gchandle.Target;
10081        Module module = executingAssembly.LoadModule("koi", array2);
10082        Array.Clear(array2, 0, array2.Length);
10083        gchandle.Free();
```

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

Instead, we 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 is decrypted and loaded. Then dump the result into a file.

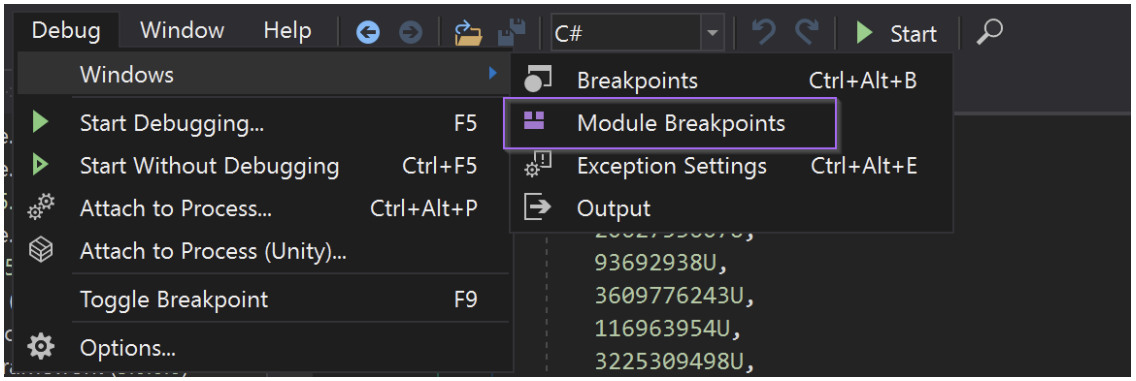
I took the second approach, as it is reliable and useful for occasions where the location of decryption and loading isn't as easy to find. (Typically it's more complicated to find the Decryption function, but luckily in this case it was rather simple)

Either way, we decided to take the second approach.

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

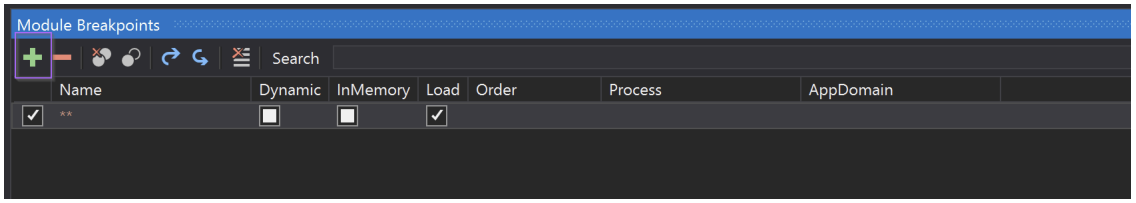
To do this, we first opened the module breakpoints window.

`Debug -> Windows -> Module Breakpoints`



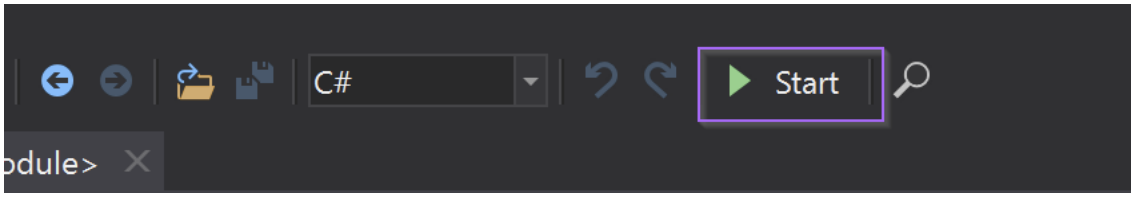
How to set a module breakpoint using Dnspy

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



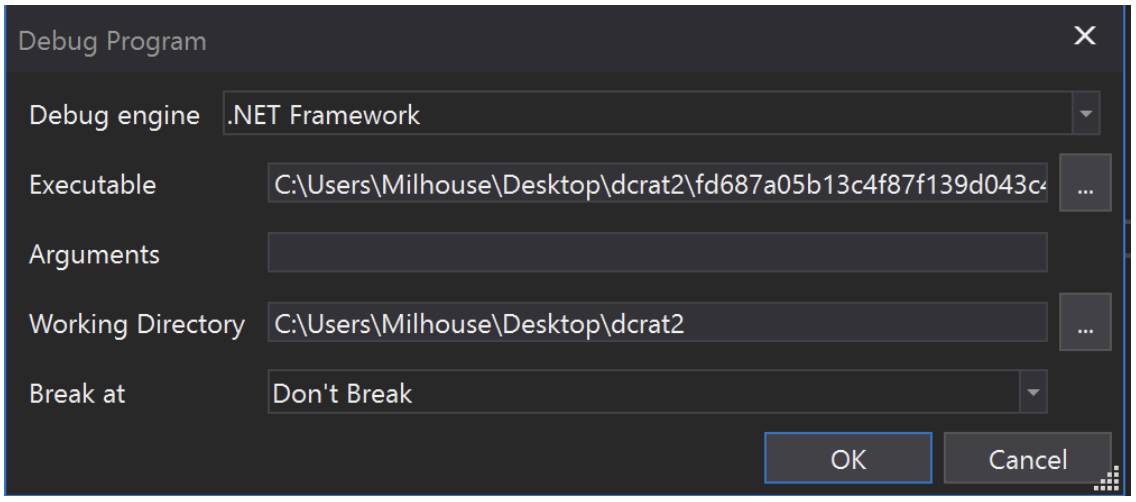
Module breakpoint to break on all loaded modules

We then executed the malware using the start button



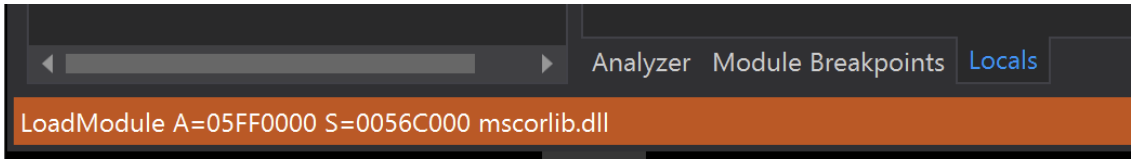
Dnspy button to Start or Continue execution

We can accept the default options.

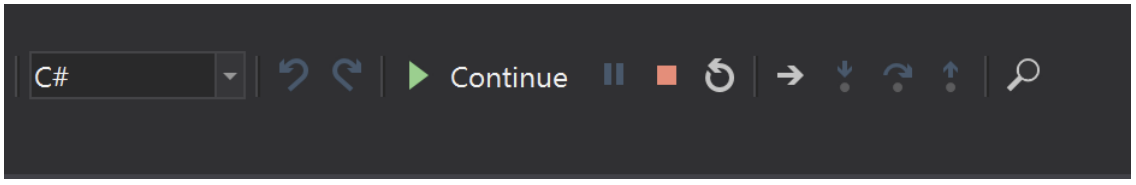


Default options for Dnspy Debugging are ok

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



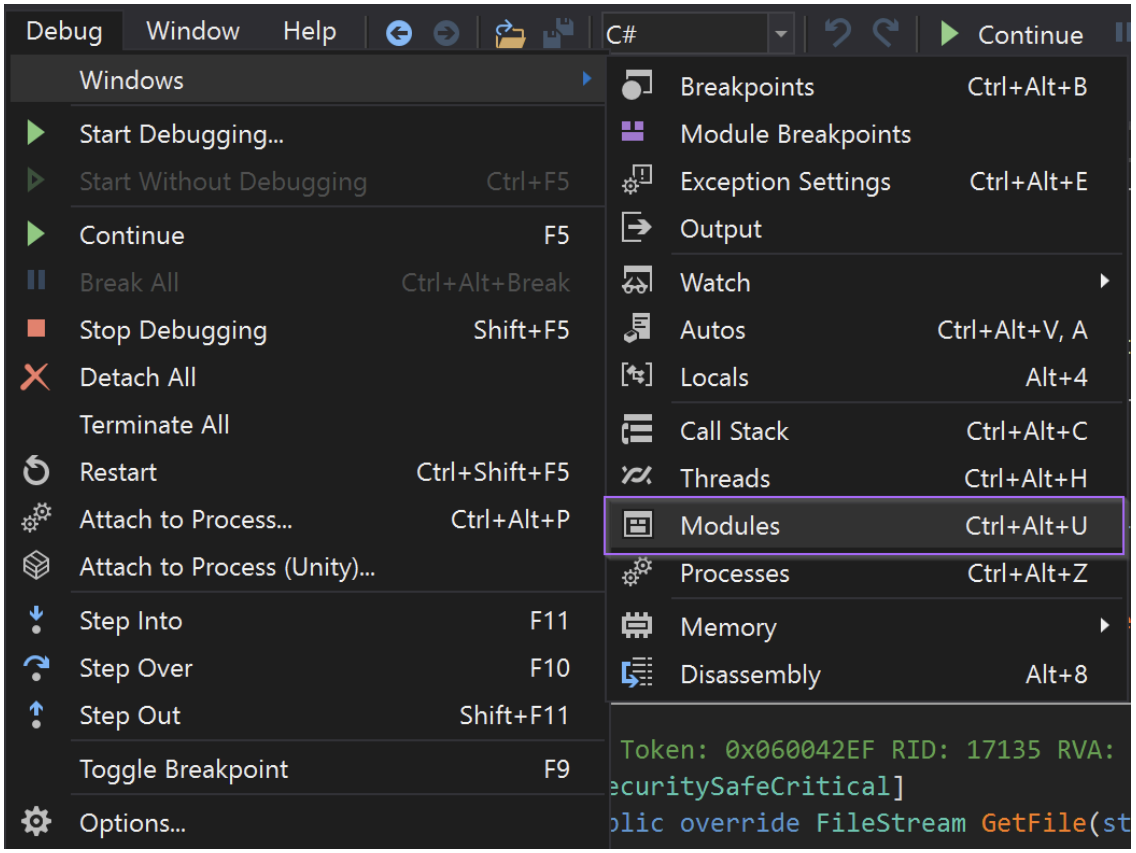
Dnspy alert when a module breakpoint has been triggered



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

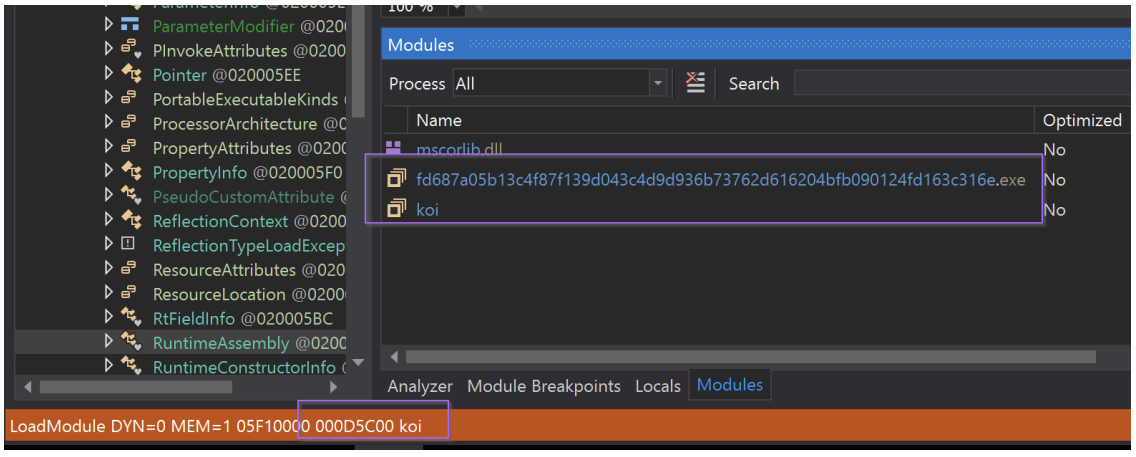
The next module loaded was the original file being analyzed, which in this case can be safely ignored.

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



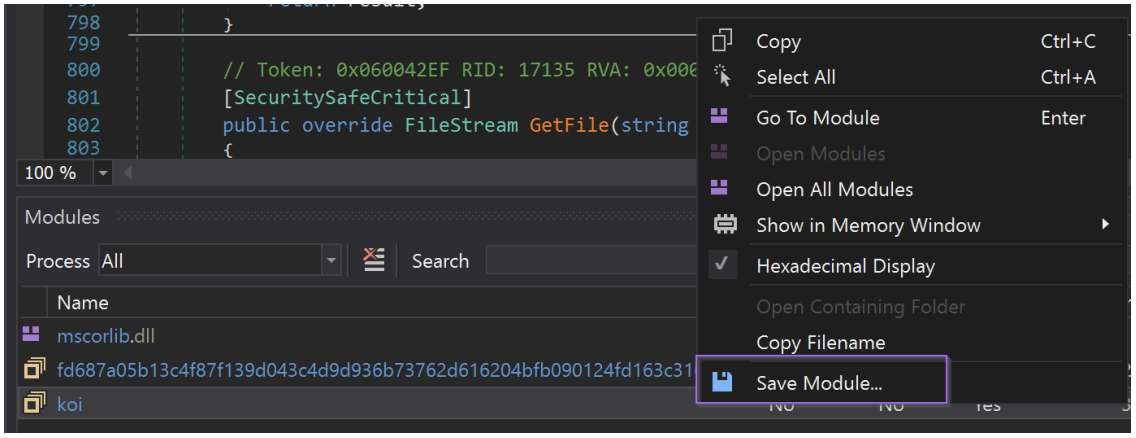
How to View Currently Loaded Modules in Dnspy

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



Example of a suspicious module being loaded into memory

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

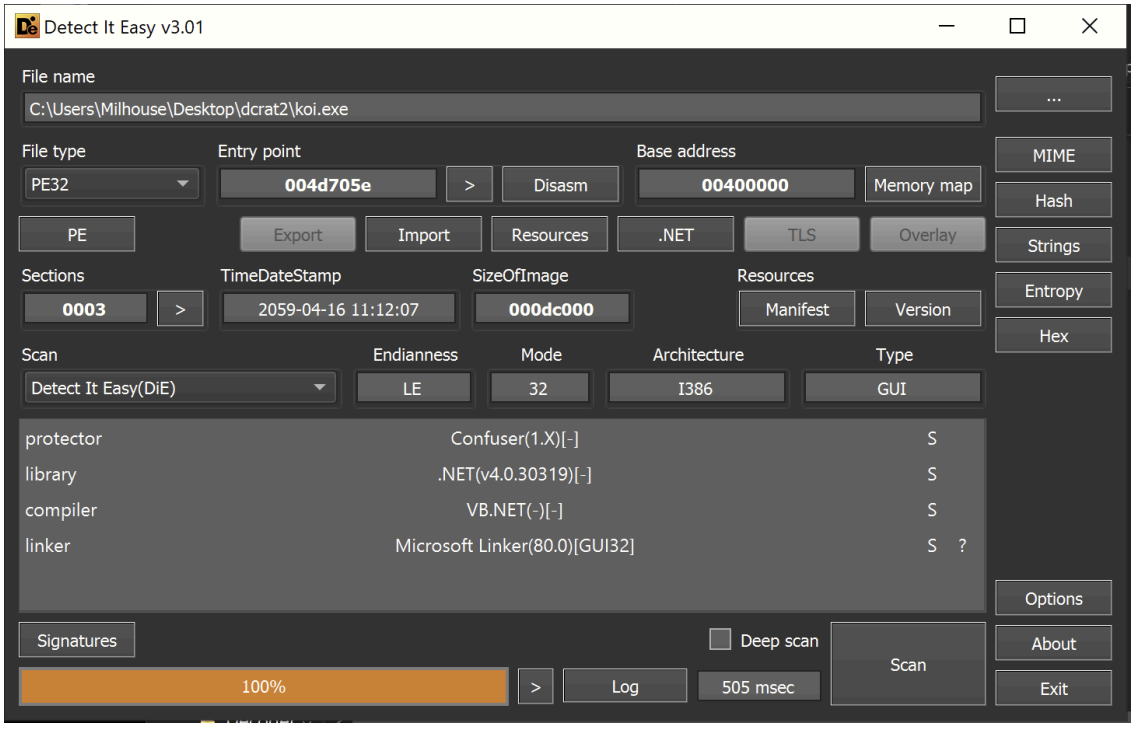


Dnspy Option for Saving a Loaded Module

We can then exit the debugger and move on to the `koi.exe` file.

Analysis of koi.exe

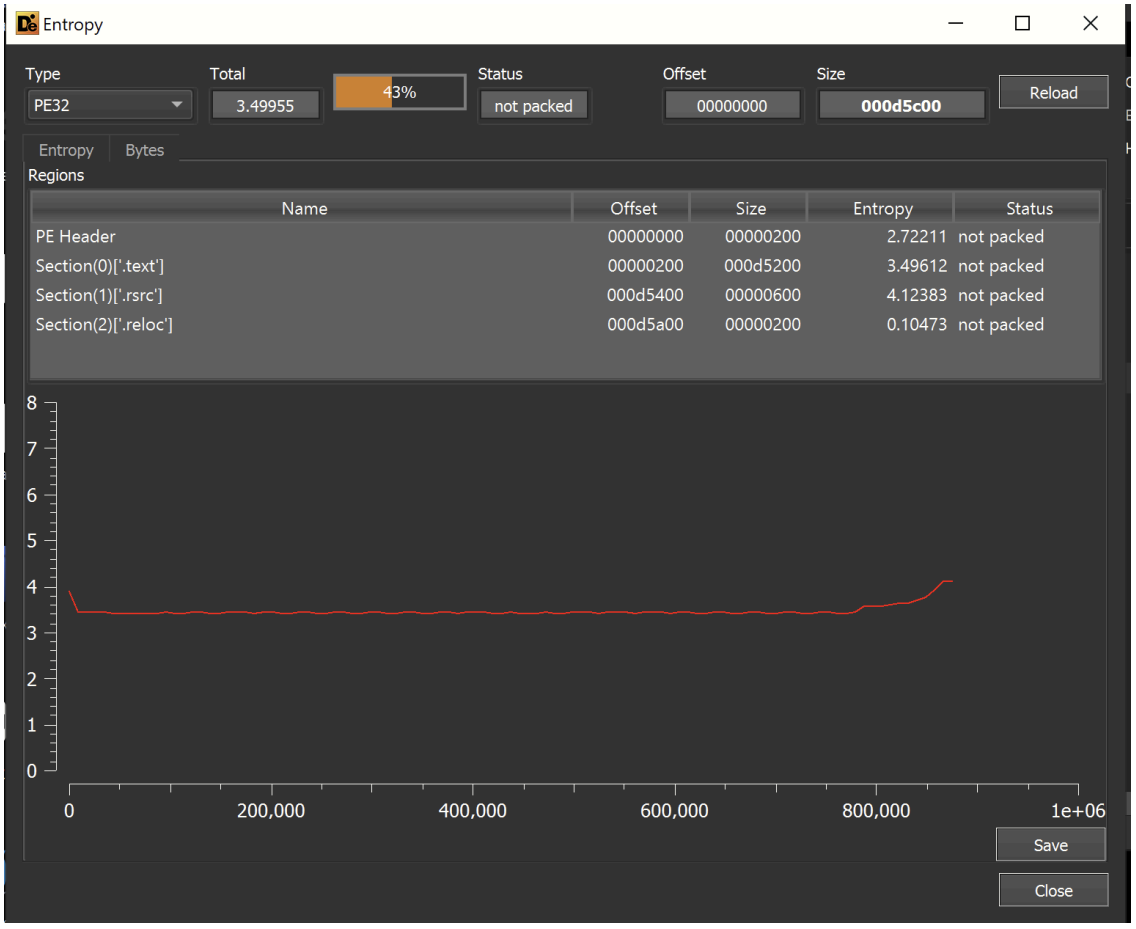
We can observe that `koi.exe` is another 32-bit .net file containing references to the `ConfuserEx` Obfuscator



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

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

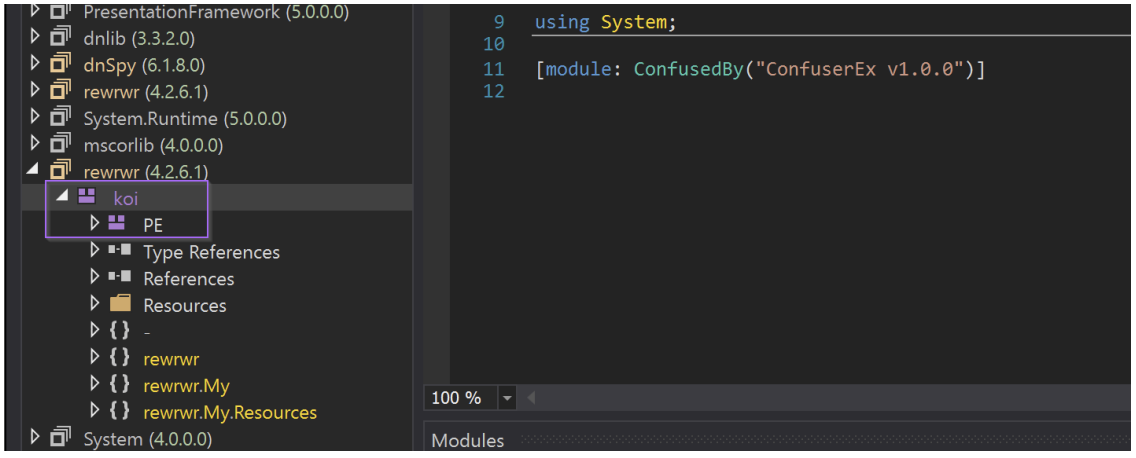
> Although the overall entropy is low, large portions of the graph are still suspiciously flat. This can sometimes be an indication of text-based obfuscation.



Entropy Analysis when a text-based obfuscation is used

We can now go ahead and open `koi.exe` in Dnspy.

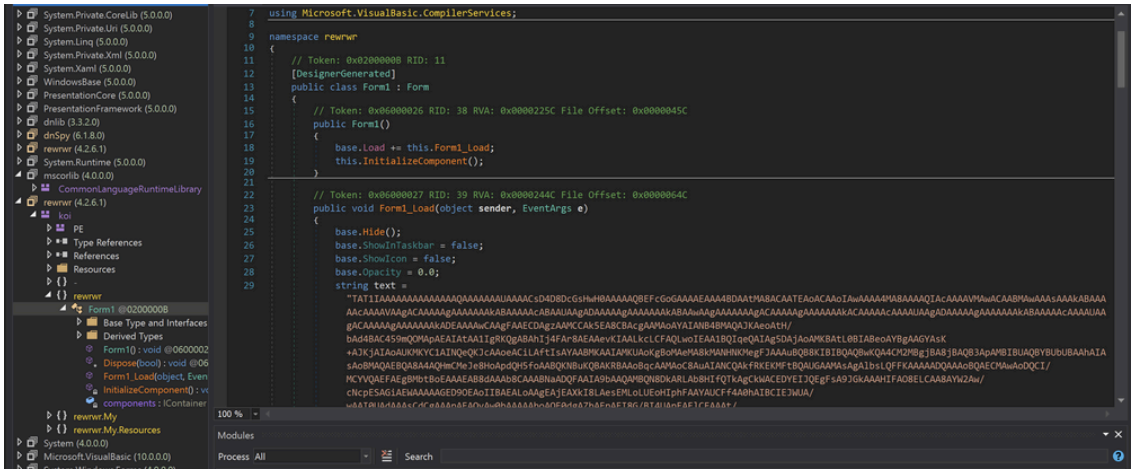
This time there was another `rewrwr.exe` name and references again to `ConfuserEx`



File Overview with Dnspy

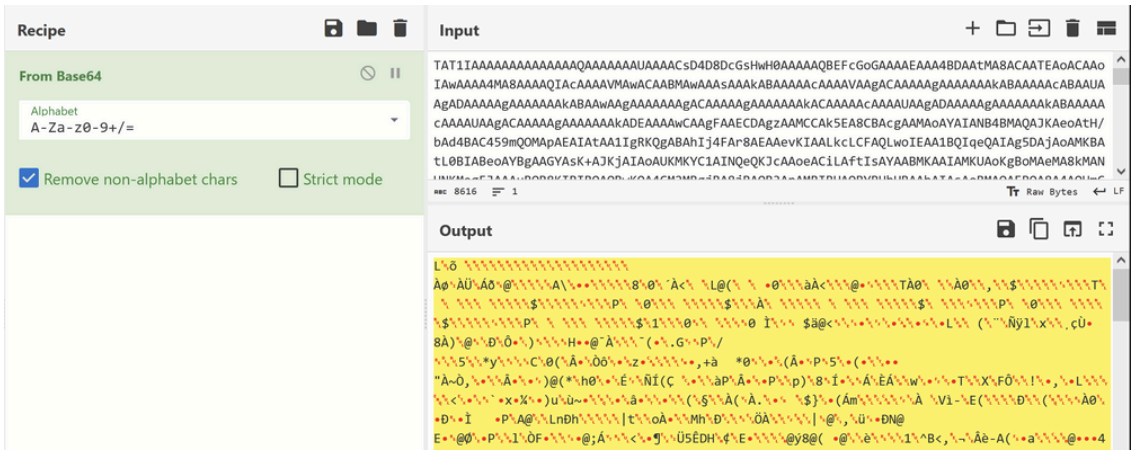
`Koi.exe` does not have a defined Entry Point. Instead we can begin analysis with the `rewrwr` namespace (located in the side panel). This namespace contains one Class named `Form1`

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



Example of Entry Point Containing Obfuscated Data

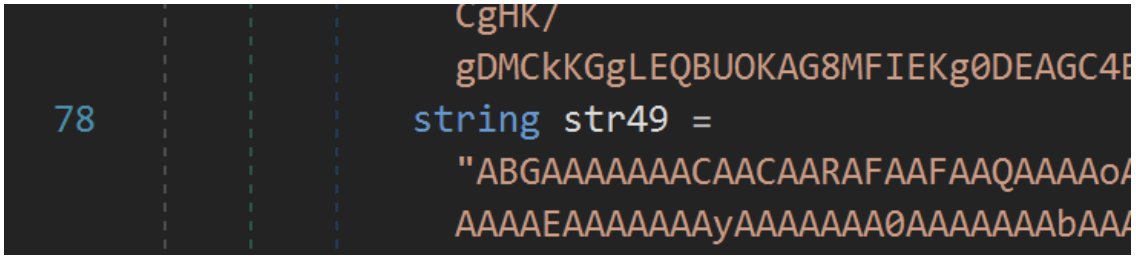
Despite appearing to be base64 - the text does not successfully code using base64. This was an indicator that some additional tricks or obfuscation had been used.



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



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.

```
79              string text2 = "";
80              int length = text.Length;
81              checked
82              {
83                  for (int i = 1; i <= length; i++)
84                  {
85                      text2 = string.Concat(new string[]
86                      {
87                          text2,
88                          Strings.Mid(text, i, 1),
89                          Strings.Mid(str, i, 1),
90                          Strings.Mid(str2, i, 1),
91                          Strings.Mid(str3, i, 1),
92                          Strings.Mid(str4, i, 1),
93                          Strings.Mid(str5, i, 1),
94                          Strings.Mid(str6, i, 1),
95                          Strings.Mid(str7, i, 1),
96                          Strings.Mid(str8, i, 1),
97                          Strings.Mid(str9, i, 1),
98                          Strings.Mid(str10, i, 1),
99                          Strings.Mid(str11, i, 1),
100                         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

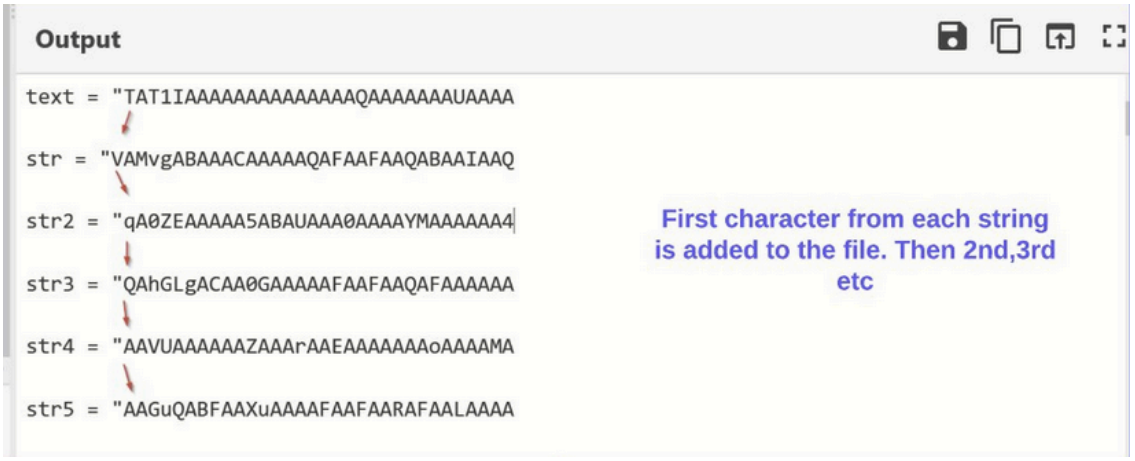
Decoding Logic Utilised by the Dcrat Loader - Viewed with Dnspy

After the buffer is filled, it is 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 are decoded, we can take the first character from the first 5 strings and base64 decode the result.



Brief Overview of the Additional obfuscation used

An example of this decodes using base64

This confirmed the theory of how the malware was decoding the next stage.

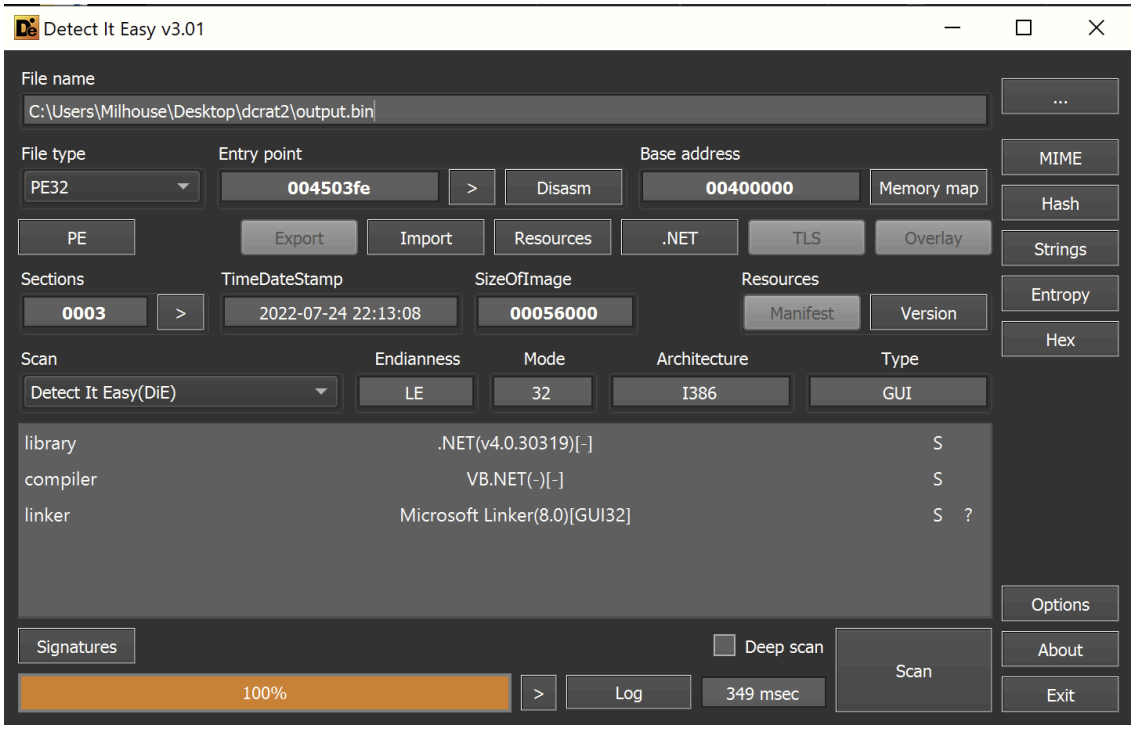
In order to extract the next module, we can copy out the strings and place them into a Python script.

```
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 creates a third file. Which for simplicity's sake is named `output.bin`

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

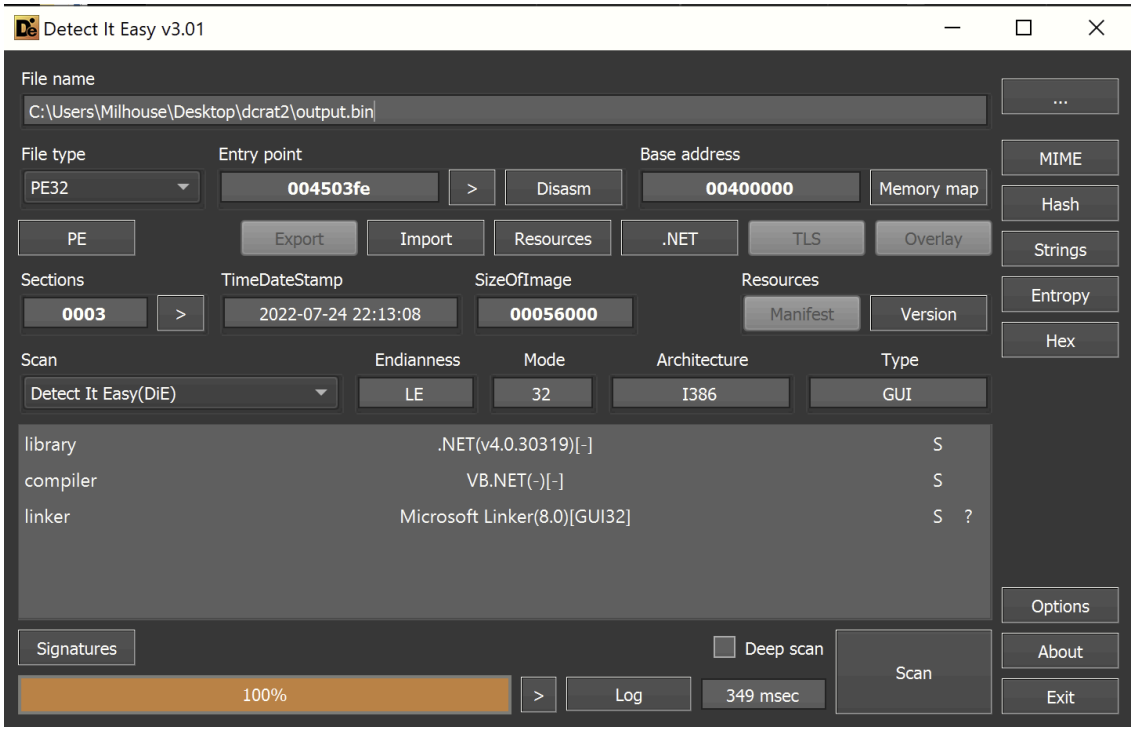


Initial Analysis of Third .NET File using Detect-it-easy

Stage 3 - Analysis

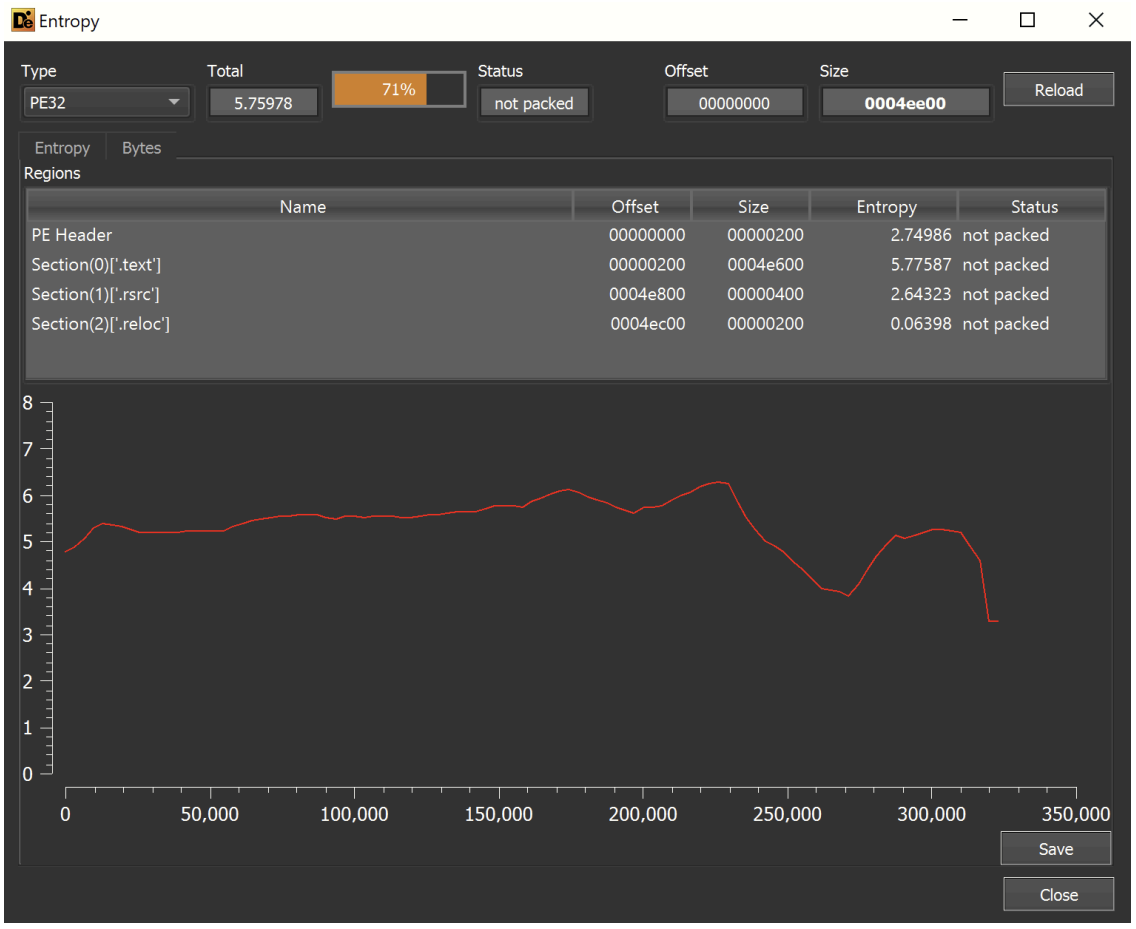
We have now obtained a stage 3 file - which again is a 32-bit .NET executable.

Luckily this time, there are no references to `ConfuserEx` or other obfuscators.



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

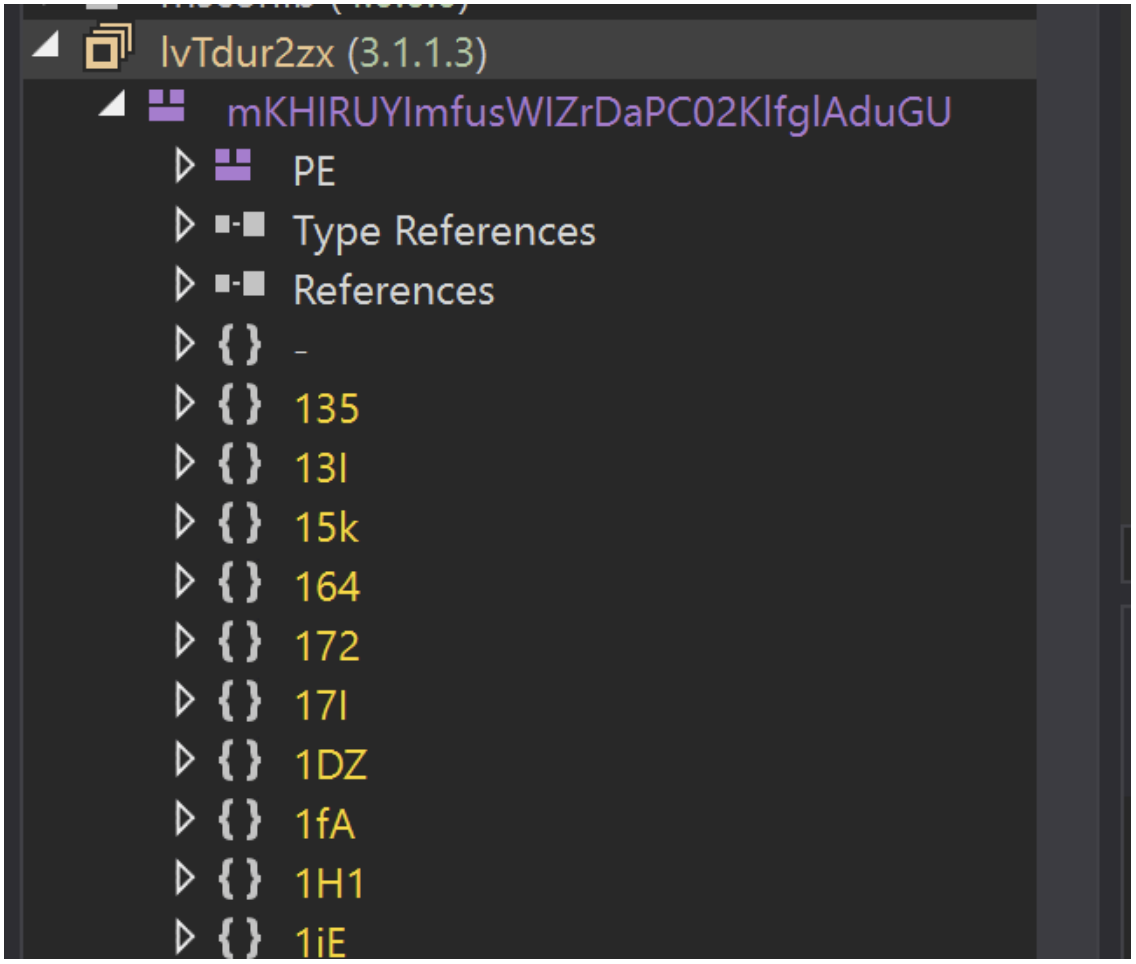
The entropy is reasonably normal - and does not contain any large flat sections that can indicate a hidden payload.



The lack of `ConfuserEx` and relatively normal entropy - is 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 obfuscated in some way.



Dnspy view of Obfuscated Functions in the Final Payload

We can jump to the Entry Point for further analysis.

```
 9  namespace koZ
10  {
11      // Token: 0x02000075 RID: 117
12      internal static class 91s
13      {
14          // Token: 0x0600018A RID: 394 RVA: 0x000054F9 File Offset: 0x000036F9
15          [STAThread]
16          private static void 47G()
17          {
18              91s.627.wTq();
19          }
20
```

The first few functions are mostly junk - but there are 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 we can use `detect-it-easy` to look for more interesting strings contained within the file.

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

Overview of some plaintext strings contained in the malware

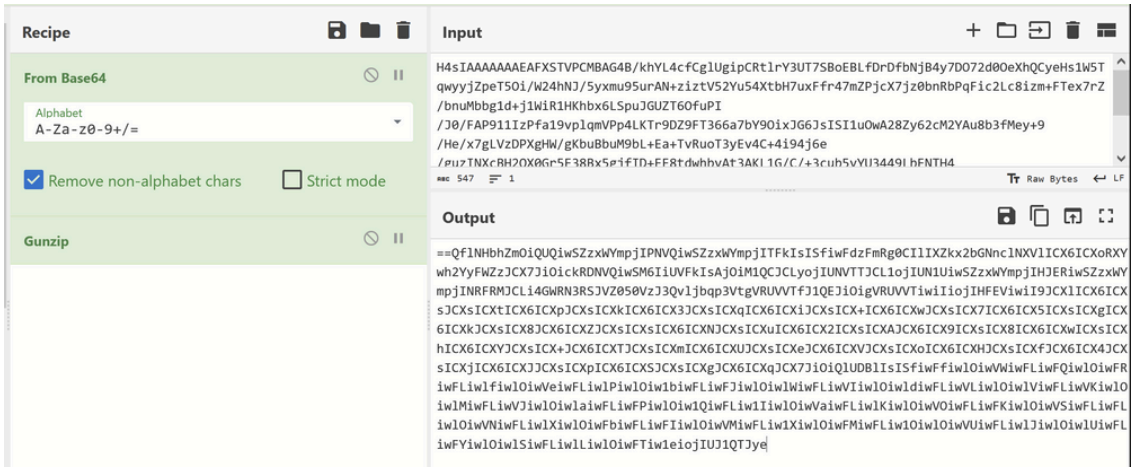
At that point, you could probably assume the file was DCrat and an info stealer - but we wanted to continue my analysis until I'd found the C2.

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

So we attempted to analyze these using CyberChef.

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

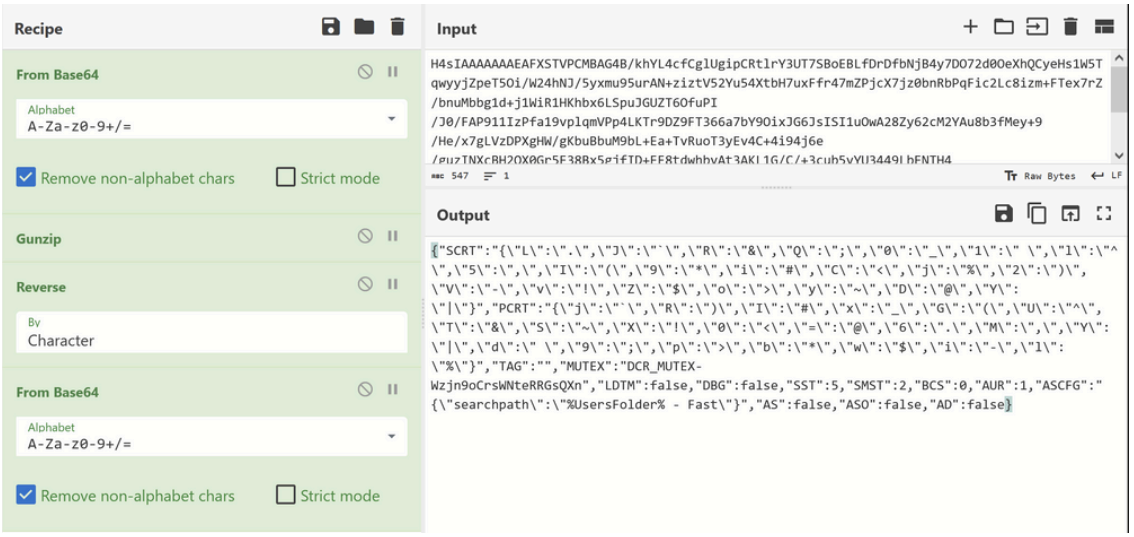
This was strongly hinted by the presence of `==` at the start.



Cyberchef - Base64 + Gzip + Additional Obfuscation

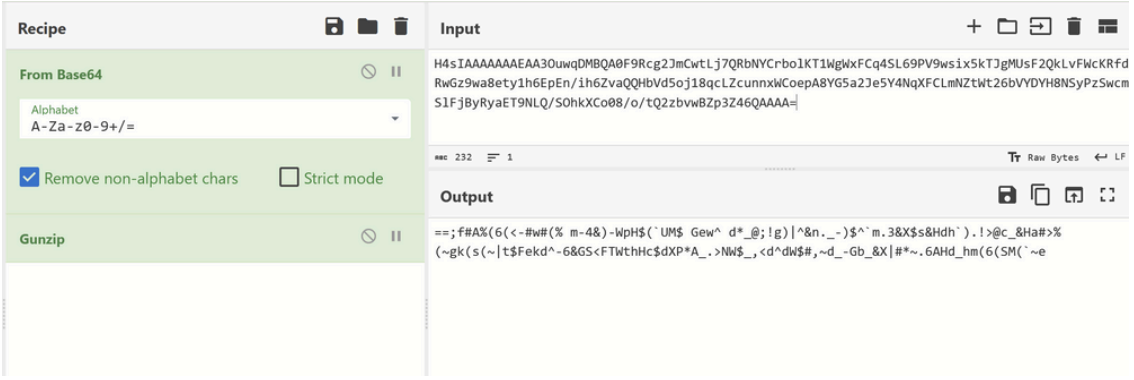
After applying a `character reverse + base64 decode`. We were able to obtain a strange dictionary as well as a mutex of `Wzjn9oCrsWNteRRGsQXn` + some basic config.

This was cool but still no C2.



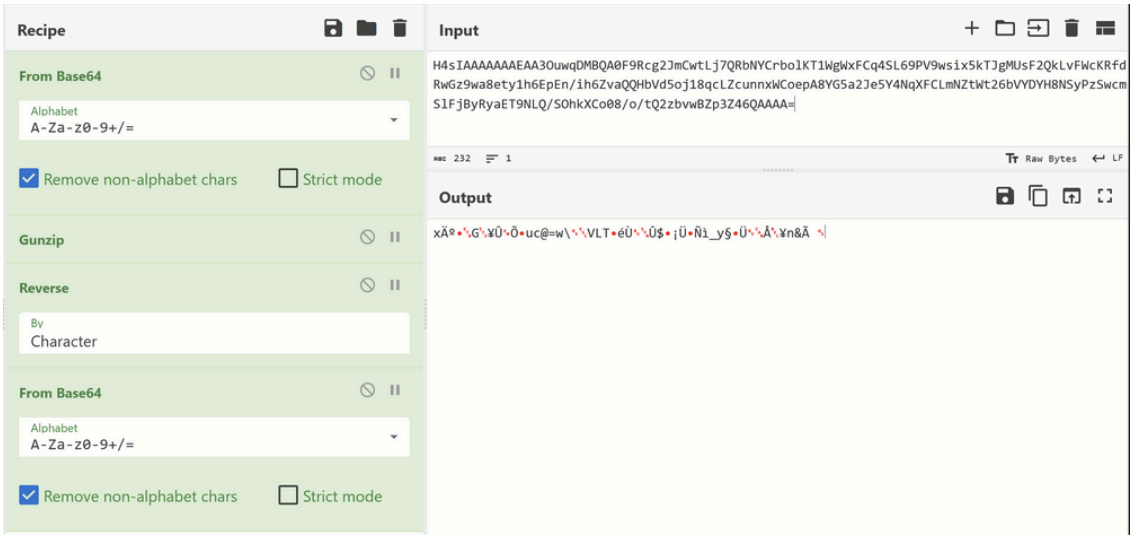
Cyberchef - Decoding the "base64" strings

I then tried to decode the second base64 blob shown by `detect-it-easy`.
But the result was largely junk.



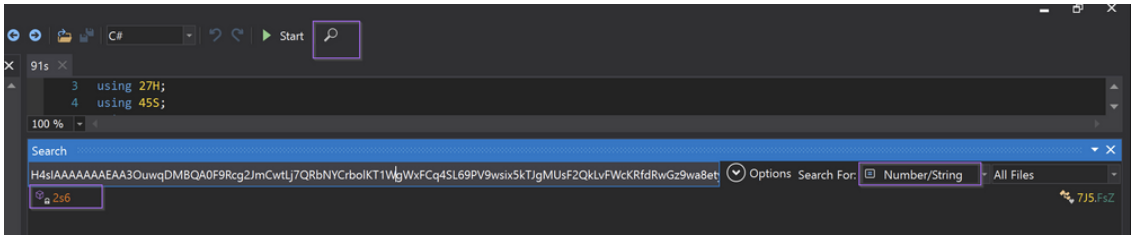
Cyberchef - Failed Decoding of Additional "base64" strings

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



Cyberchef - Additional failures when decoding strings

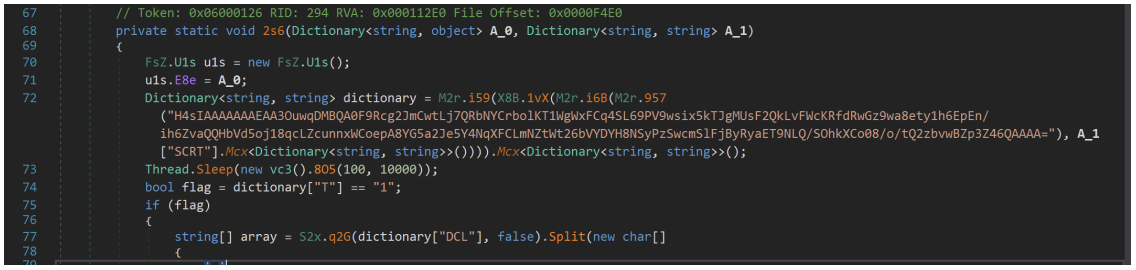
At this point - we decided to search for the base64 encoded string to see where it was referenced in the .net code.



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.

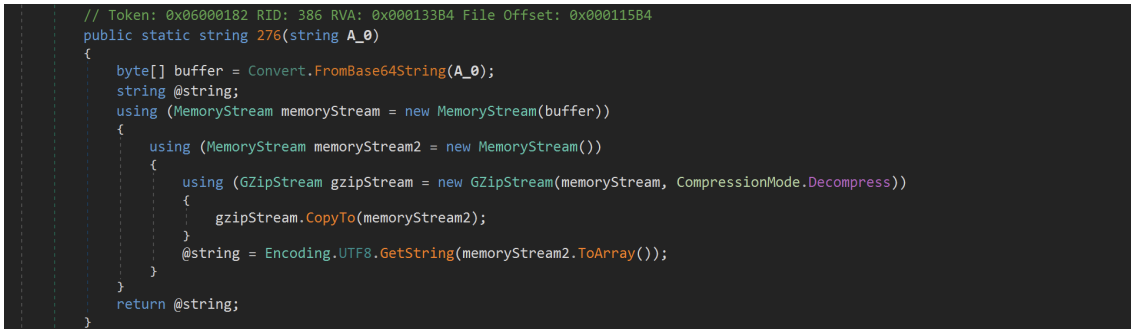
In total, there are 4 functions (`M2r.957` , `M2r.i6B` , `M2r.1vX` , `M2r.i59`) which are acting on the encoded data.



```
// 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
      ("H4sIAAAAAAAAEAA3OuwqDMBQA0F9Rcg2JmCwtLj7QRbNYCrbolKT1WgWxFCq4SL69PV9wsix5kTJgMUsF2QkLvFWcKRfdRwGz9wa8ety1h6EpEn/
      ih6ZvaQQHbVd5oj18qcLZcunnxWCoepA8YG5a2Je5Y4NqXFCLmNZtWt26bVYDYH8NSyPzSwcmSlFjByRyaET9NLQ/SOhkXCo08/o/tQ2zbvwBZp3Z46QAAAA="), A_1
      ["SCRT"].Mcx<Dictionary<string, string>>())))).Mcx<Dictionary<string, string>>();
    Thread.Sleep(new vc3().805(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

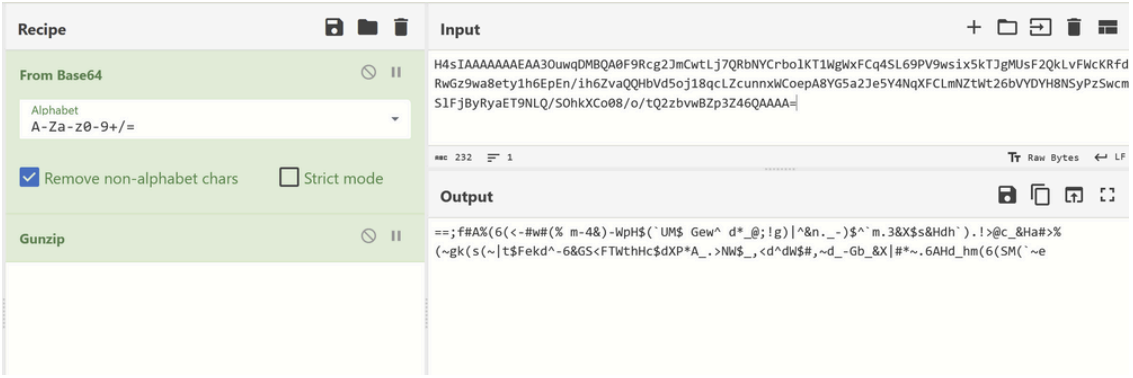
The first function `M2r.957` is a wrapper around another function `M2r.276` which performed the `base64` and `Gzip` decoding.



```
// 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;
}
```

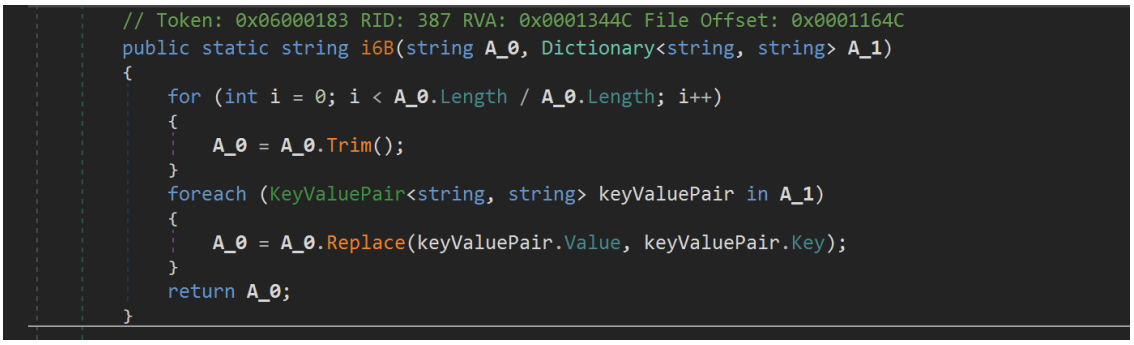
Delving Deeper into an "obfuscation" function.

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



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 had 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
      ("H4sIAAAAAAAAEAA3OuwqDMBQA0F9Rcg2JmCwtLj7QRbNYCrbolKT1WgWxFCq4SL69PV9wsix5kTJgMUsF2QkLvFWcKRfd
      ih6ZvaQQHbVd5oj18qcLZcunnxWCoepA8YG5a2Je5Y4NqXFCLmNZtWt26bVYDYH8NSyPzSwcmSlFjByRyaET9NLQ/SOhkXCo08/o/tQ2zbvwBZp3Z46QAAAA="), A_1
      ["SCRT"].Mcx<Dictionary<string, string>>())))).Mcx<Dictionary<string, string>>();
    Thread.Sleep(new vc3().805(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.

```
Output
{"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\":\"-\",\"1\":
\"%\"}","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 we obtained that dictionary and prettied it up using Cyberchef to remove all of the `\` escapes.

Cleaning up escape characters with Cyberchef

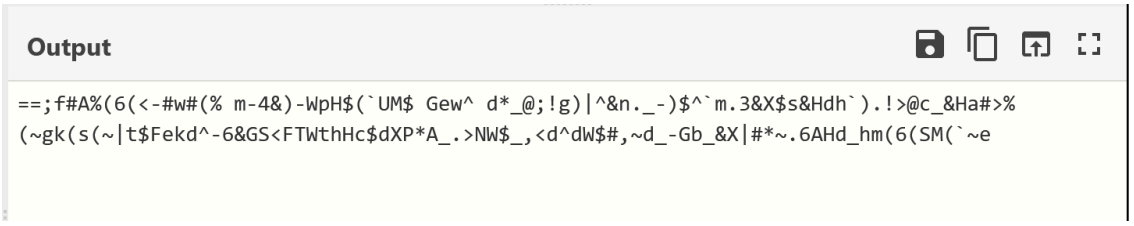
We then created a partial Python script based on the information we had so far. (I'll post a link at the end of this post)

```
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 = "H4sIAAAAAAAAEAA3OuwqDMBQA0F9Rcg2JmCwtLj7QRbNYCrbolKT1WgWxFCq4SL69PV9wsix5kTJgMUsF2QkLvFWcKRfdRwGz9wa8ety1h6EpEn/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 - we were able to obtain a cleaner-looking string than before.

Here's a before and after



Before applying additional text-replacement

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

After applying additional text-replacement

It was probably safe to assume this string was `reversed` + `base64 encoded`, but we 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 we 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 = "H4sIAAAAAAAAEAA3OuwqDMBQA0F9Rcg2JmCwtLj7QRbNYCrbolKT1WgWxFCq4SL69PV9wsix5kTJgMUsF2QkLvFWcKRfdRwGz9wa8ety1h6EpEn/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!

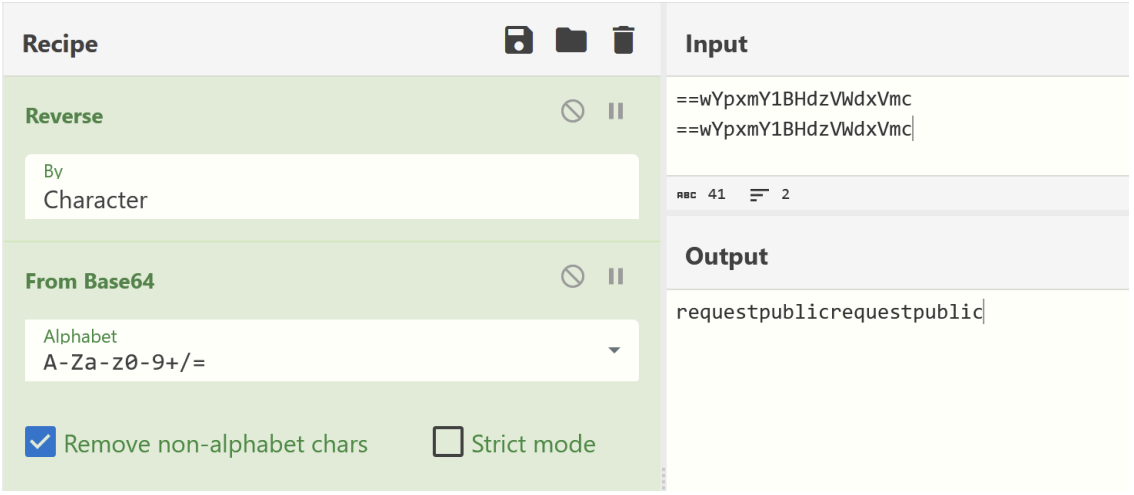
`http://battletw[.]beget[.]tech/`

```
>>>
============= 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"}'
>>>
```

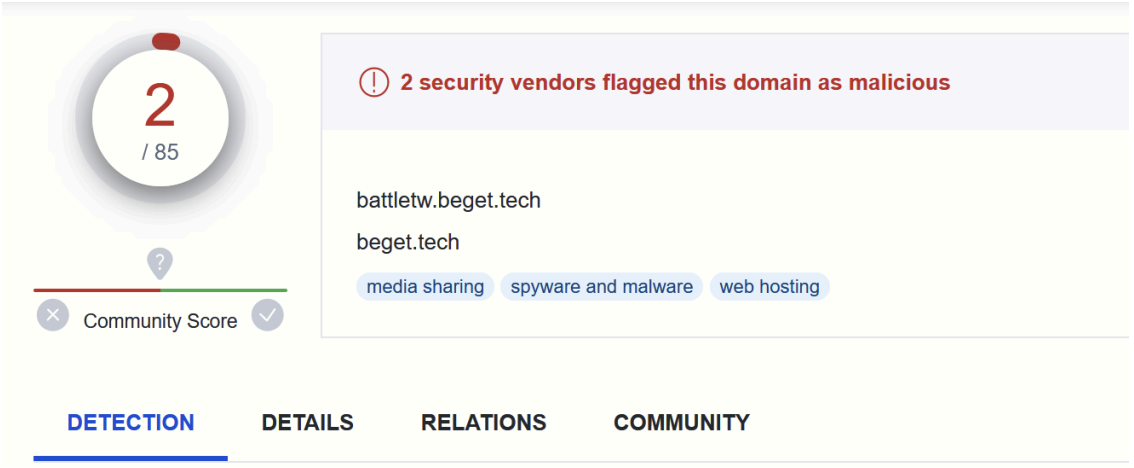
Ln: 10 Col: 4

Successfully obtaining the decoded C2 using Python.

(The URLs contained some base64 reversed/encoded strings and were not very interesting)



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



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

In a real environment, it would be best to block this domain immediately in your security solutions. Additionally, you could review the previous string dumps for process-based indicators that could be used to hunt signs of successful execution.

Additionally, you could try to 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/
