

How To Write a Simple Configuration Extractor For .NET Malware - RevengeRAT

By Matthew

Published: 2023-10-05 · Archived: 2026-04-05 18:05:41 UTC

This post is an introduction to developing configuration extractors for dotnet malware. The sample used here is RevengeRat, this rat typically employs minimal obfuscation and presents an ideal introduction for config extraction.

The sample has config which can be obtained via strings. However, it is far more interesting and useful to obtain the same values by enumerating IL instructions present inside the code. This allows the analyst to hone in on particular string values and eventually build more advanced configuration extractors.

The two primary samples we will be using are

[Initial Sample Link](#): `0d05942ce51fea8c8724dc6f3f9a6b3b077224f1f730feac3c84efe2d2d6d13e`

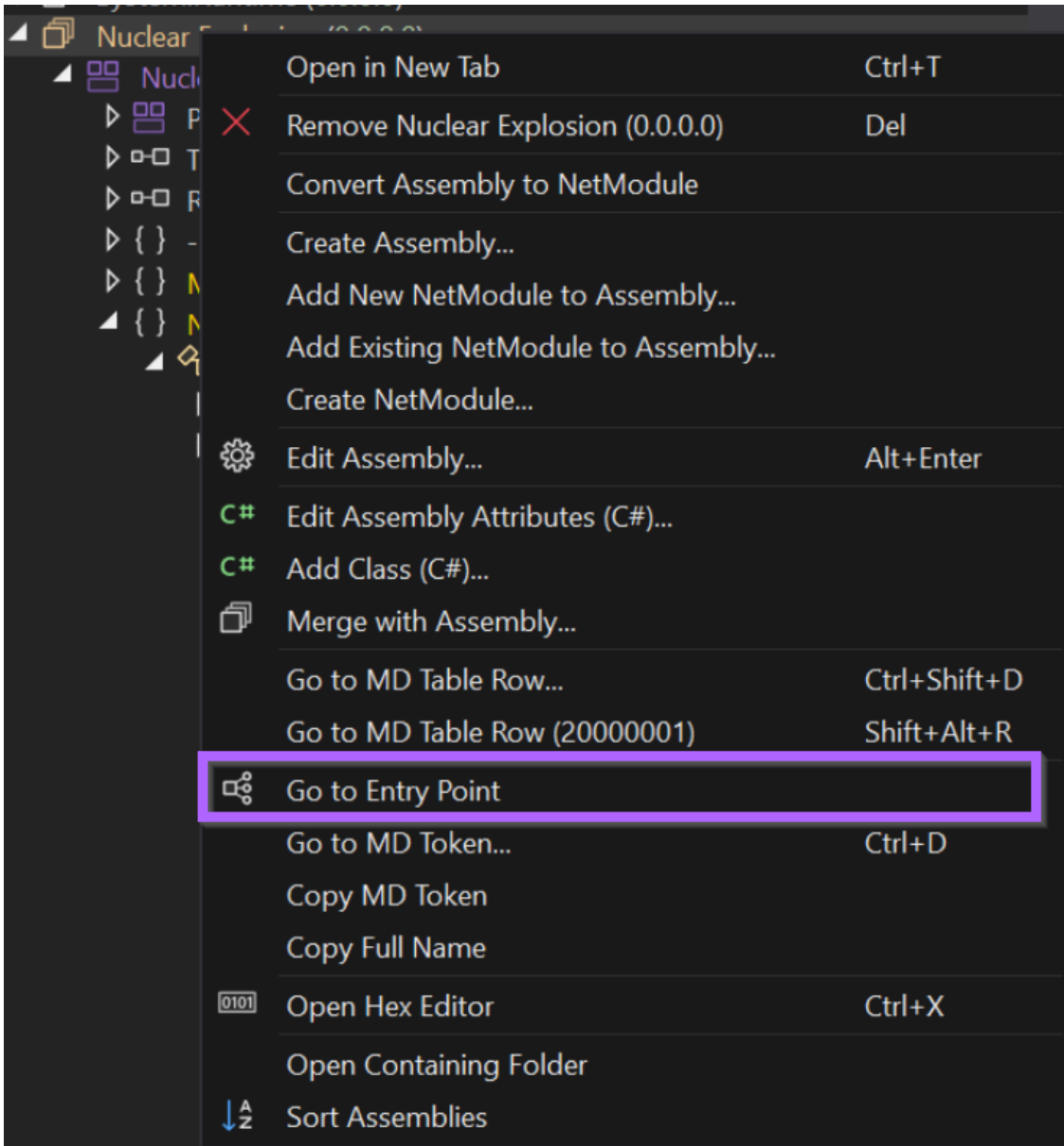
[Obfuscated Sample Link](#): `dd203194d0ea8460ac3173e861737a77fa684e5334503867e91a70acc7f73195`

Overview

First Step - Manually Locating the Configuration

To build a automated configuration extractor, we first need to be able to locate the configuration manually. For .NET based malware, this means opening up the file in Dnspy and attempting to locate configuration values or functions. .

For .NET malware, the entry point is a good place to start looking. This is because configuration is *generally* resolved early in the malware execution.



For this sample, the Entry Point is the `Main` function. Lucky for us, the config values are directly above the entry point inside of `Atomic()`.

```
28 public class Atomic
29 {
30     // Token: 0x06000006 RID: 6 RVA: 0x0002114 File Offset: 0x0000314
31     public Atomic()
32     {
33         this.Ow = false;
34         this.C = null;
35         this.Cn = false;
36         this.SC = new Thread(new ThreadStart(this.MAC), 1);
37         this.PT = new Thread(new ThreadStart(this.Pin));
38         this.INST = new Thread(new ThreadStart(this.INS));
39         this.GP = new Thread(new ThreadStart(Atomic.Spread));
40         this.I = 1;
41         this.MS = 0;
42         this.Hosts = Strings.Split("127.0.0.1,", ",", -1, CompareMethod.Binary);
43         this.Ports = Strings.Split("1177,", ",", -1, CompareMethod.Binary);
44         this.ID = "R3Vlc3Q=";
45         this.MUTEX = "RV_Mutex-RZb1RvZwFRtN";
46         this.H = 0;
47         this.P = 0;
48     }
49 }
50
51 // Token: 0x06000007 RID: 7 RVA: 0x00021FC File Offset: 0x00003FC
52 [STAThread]
53 public static void Main()
54 {
55     bool flag = !File.Exists(Path.GetTempPath() + "eNHuiGG.txt");
56     if (flag)
57     {
58         bool flag2 = Atomic.App.Contains("RevengeRAT\\44444.exe");
59         if (flag2)
60         {
61             File.WriteAllText(Path.GetTempPath() + "eNHuiGG.txt", "True");
62         }
63     }
64     Atomic.SCG.Execute();
65 }
```

Config Values

Entry Point

This is a rare case where the configuration is already in plaintext and is extremely simple to find. Since it is extremely simple to find, it's also extremely simple to write an extractor.

For this sample, you could just run strings and you would obtain the same values, but the point of this post is to do the entire process via scripting. This will build foundational skills that are essential for building extractors for more complex malware.

```
28 public class Atomic
29 {
30     // Token: 0x06000006 RID: 6 RVA: 0x0002114 File Offset: 0x0000314
31     public Atomic()
32     {
33         this.Ow = false;
34         this.C = null;
35         this.Cn = false;
36         this.SC = new Thread(new ThreadStart(this.MAC), 1);
37         this.PT = new Thread(new ThreadStart(this.Pin));
38         this.INST = new Thread(new ThreadStart(this.INS));
39         this.GP = new Thread(new ThreadStart(Atomic.Spread));
40         this.I = 1;
41         this.MS = 0;
42         this.Hosts = Strings.Split("127.0.0.1,", ",", -1, CompareMethod.Binary);
43         this.Ports = Strings.Split("1177,", ",", -1, CompareMethod.Binary);
44         this.ID = "R3Vlc3Q=";
45         this.MUTEX = "RV_Mutex-RZb1RvZwFRtN";
46         this.H = 0;
47         this.P = 0;
48     }
49 }
50
51 // Token: 0x06000007 RID: 7 RVA: 0x00021FC File Offset: 0x00003FC
52 [STAThread]
53 public static void Main()
54 {
55     bool flag = !File.Exists(Path.GetTempPath() + "eNHuiGG.txt");
56     if (flag)
57     {
58         bool flag2 = Atomic.App.Contains("RevengeRAT\\44444.exe");
59         if (flag2)
60         {
61             File.WriteAllText(Path.GetTempPath() + "eNHuiGG.txt", "True");
62         }
63     }
64     Atomic.SCG.Execute();
65 }
```

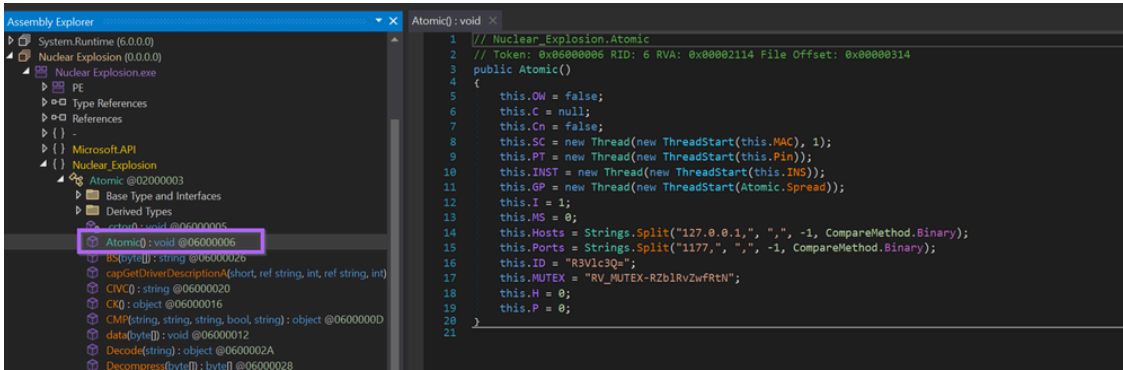
Config Values

Entry Point

Now that the config has been found, we want to hone in deeper on the Atomic() method that contains the config values.

This can be done by clicking on Atomic() in the side menu.

This ensures that the decompiled code is only that of the relevant function.



Now this is where things get interesting.

Switching to IL Instructions

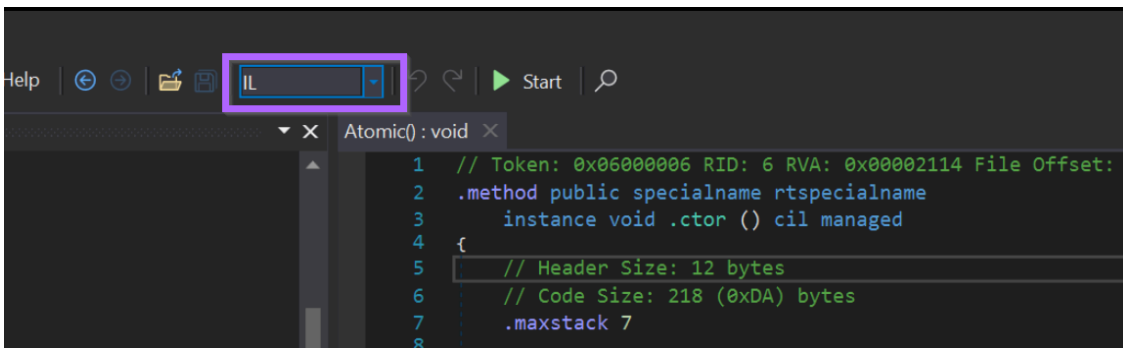
To build configuration extractors for dotnet malware, we generally need to leverage `dnlib`.

As far as we can tell, `dnlib` has no knowledge of the decompiled `c#` code that we see in `Dnspy`.

`Dnlib` works best with Intermediate Language (IL) instructions and not decompiled `c#` code.

To accommodate this, we also need to switch to Intermediate Language Instructions.

We can do this by changing this dropdown box from `C#` to `IL`.



The `Atomic()` code has now changed significantly. The output now contains Intermediate Language instructions and opcodes instead of the *usual c#* code.

Everything in this view can be accessed and enumerated via `dnlib` inside of a python script.

```
Atomic():void
1 // Token: 0x00000006 RID: 6 RVA: 0x0002114 File Offset: 0x0000314
2 .method public specialname rtspecialname
3 instance void .ctor () cil managed
4
5 // Header Size: 12 bytes
6 // Code Size: 218 (0xDA) bytes
7 .maxstack 7
8
9 /* 0x00000320 02 */ /* IL_0000: ldarg.0
10 /* 0x00000321 28100000A */ /* IL_0001: call instance void [mscorlib]System.Object::.ctor()
11 /* 0x00000326 00 */ /* IL_0006: nop
12 /* 0x00000327 02 */ /* IL_0007: ldarg.0
13 /* 0x00000328 16 */ /* IL_0008: ldc.i4.0
14 /* 0x00000329 7D0100004 */ /* IL_0009: stfld bool Nuclear_Explosion.Atomic::OW
15 /* 0x0000032E 02 */ /* IL_000E: ldarg.0
16 /* 0x0000032F 14 */ /* IL_000F: ldnull object Nuclear_Explosion.Atomic::C
17 /* 0x00000330 7D0200004 */ /* IL_0010: stfld
18 /* 0x00000335 02 */ /* IL_0015: ldarg.0
19 /* 0x00000336 16 */ /* IL_0016: ldc.i4.0
20 /* 0x00000337 7D0300004 */ /* IL_0017: stfld bool Nuclear_Explosion.Atomic::Cn
21 /* 0x0000033C 02 */ /* IL_001C: ldarg.0
22 /* 0x0000033D 02 */ /* IL_001D: ldarg.0
23 /* 0x0000033E FE061500006 */ /* IL_001E: ldftn instance void Nuclear_Explosion.Atomic::MAC()
24 /* 0x00000344 73150000A */ /* IL_0024: newobj instance void [mscorlib]System.Threading.ThreadStart::.ctor(object, native int)
25 /* 0x00000349 17 */ /* IL_0029: ldc.i4.1
26 /* 0x0000034A 73160000A */ /* IL_002A: newobj instance void [mscorlib]System.Threading.Thread::.ctor(class [mscorlib]System.Threading.ThreadStart, int32)
27 /* 0x0000034F 7D0400004 */ /* IL_002F: stfld object Nuclear_Explosion.Atomic::SC
28 /* 0x00000354 02 */ /* IL_0034: ldarg.0
```

Heres a quick screenshot to better understand the output.

Fun fact - the bytecodes column is extremely useful for developing yara rules targeting dotnet malware. These are the bytecodes that are present in the raw binary. [Binary Defense blog](#)

```
Atomic():void
1 // Token: 0x00000006 RID: 6 RVA: 0x0002114 File Offset: 0x0000314
2 .method public specialname rtspecialname
3 instance void .ctor () cil managed
4
5 // Header Size: 12 bytes
6 // Code Size: 218 (0xDA) bytes
7 .maxstack 7
8
9 /* 0x00000320 02 */ /* IL_0000: ldarg.0
10 /* 0x00000321 28100000A */ /* IL_0001: call instance void [mscorlib]System.Object::.ctor()
11 /* 0x00000326 00 */ /* IL_0006: nop
12 /* 0x00000327 02 */ /* IL_0007: ldarg.0
13 /* 0x00000328 16 */ /* IL_0008: ldc.i4.0
14 /* 0x00000329 7D0100004 */ /* IL_0009: stfld bool Nuclear_Explosion.Atomic::OW
15 /* 0x0000032E 02 */ /* IL_000E: ldarg.0
16 /* 0x0000032F 14 */ /* IL_000F: ldnull object Nuclear_Explosion.Atomic::C
17 /* 0x00000330 7D0200004 */ /* IL_0010: stfld
18 /* 0x00000335 02 */ /* IL_0015: ldarg.0
19 /* 0x00000336 16 */ /* IL_0016: ldc.i4.0
20 /* 0x00000337 7D0300004 */ /* IL_0017: stfld bool Nuclear_Explosion.Atomic::Cn
21 /* 0x0000033C 02 */ /* IL_001C: ldarg.0
22 /* 0x0000033D 02 */ /* IL_001D: ldarg.0
23 /* 0x0000033E FE061500006 */ /* IL_001E: ldftn instance void Nuclear_Explosion.Atomic::MAC()
24 /* 0x00000344 73150000A */ /* IL_0024: newobj instance void [mscorlib]System.Threading.ThreadStart::.ctor(object, native int)
25 /* 0x00000349 17 */ /* IL_0029: ldc.i4.1
26 /* 0x0000034A 73160000A */ /* IL_002A: newobj instance void [mscorlib]System.Threading.Thread::.ctor(class [mscorlib]System.Threading.ThreadStart, int32)
27 /* 0x0000034F 7D0400004 */ /* IL_002F: stfld object Nuclear_Explosion.Atomic::SC
28 /* 0x00000354 02 */ /* IL_0034: ldarg.0
```

We now want to locate the same configuration values within the IL instructions.

Luckily, they're all still there. Noting that each of the config values are referenced as part of ldstr operations.

ldstr is short for "Load String" and is unsurprisingly used to load strings.

```

44 /* 0x0000038F 7517000004 */ IL_0087: newobj instance void [mscorlib]System.Threading.Thread::ctor(class [mscorlib]System.Threading.ThreadStart)
45 /* 0x00000394 70D7000004 */ IL_0074: stfld class [mscorlib]System.Threading.Thread Nuclear_Explosion.Atomic::GP
46 /* 0x00000399 02 */ IL_0079: ldarg.0
47 /* 0x0000039A 17 */ IL_007A: ldc.i4.1
48 /* 0x0000039B 70D0000004 */ IL_007B: stfld int32 Nuclear_Explosion.Atomic::I
49 /* 0x000003A0 02 */ IL_0080: ldarg.0
50 /* 0x000003A1 16 */ IL_0081: ldc.i4.0
51 /* 0x000003A2 70D9000004 */ IL_0082: stfld int32 Nuclear_Explosion.Atomic::MS
52 /* 0x000003A7 02 */ IL_0087: ldarg.0
53 /* 0x000003A8 7228000070 */ IL_0088: ldstr "127.0.0.1,"
54 /* 0x000003AD 7241000070 */ IL_0089: ldstr "
55 /* 0x000003B2 15 */ IL_0092: ldc.i4.m1
56 /* 0x000003B3 16 */ IL_0093: ldc.i4.0
57 /* 0x000003B4 281800000A */ IL_0094: call string[] [Microsoft.VisualBasic]Microsoft.VisualBasic.Strings::Split(string, string, int32, valueType
[Microsoft.VisualBasic]Microsoft.VisualBasic.CompareMethod)
58 /* 0x000003B9 70D0A00004 */ IL_0099: stfld string[] Nuclear_Explosion.Atomic::Hosts
59 /* 0x000003BE 02 */ IL_009E: ldarg.0
60 /* 0x000003BF 7245000070 */ IL_009F: ldstr "1177,"
61 /* 0x000003C4 7241000070 */ IL_00A4: ldstr "
62 /* 0x000003C9 15 */ IL_00A9: ldc.i4.m1
63 /* 0x000003CA 16 */ IL_00AA: ldc.i4.0
64 /* 0x000003CB 281800000A */ IL_00AB: call string[] [Microsoft.VisualBasic]Microsoft.VisualBasic.Strings::Split(string, string, int32, valueType
[Microsoft.VisualBasic]Microsoft.VisualBasic.CompareMethod)
65 /* 0x000003D0 70D0B00004 */ IL_00B0: stfld string[] Nuclear_Explosion.Atomic::Ports
66 /* 0x000003D5 02 */ IL_00B5: ldarg.0
67 /* 0x000003D6 7251000070 */ IL_00B6: ldstr "R3V1c3Q="
68 /* 0x000003DB 70DC000004 */ IL_00B8: stfld string Nuclear_Explosion.Atomic::ID
69 /* 0x000003E0 02 */ IL_00BC: ldarg.0
70 /* 0x000003E1 7263000070 */ IL_00C1: ldstr "RV_MUTEX-RZD1RVzWfRtN"
71 /* 0x000003E6 70D0D00004 */ IL_00C6: stfld string Nuclear_Explosion.Atomic::MUTEX
72 /* 0x000003EB 02 */ IL_00CB: ldarg.0
73 /* 0x000003EC 16 */ IL_00CC: ldc.i4.0
74 /* 0x000003ED 70DE000004 */ IL_00CD: stfld int32 Nuclear_Explosion.Atomic::H
75 /* 0x000003F2 02 */ IL_00D2: ldarg.0
76 /* 0x000003F3 16 */ IL_00D3: ldc.i4.0
77 /* 0x000003F4 70DF000004 */ IL_00D4: stfld int32 Nuclear_Explosion.Atomic::P
78 /* 0x000003F9 2A */ IL_00D9: ret
79 // end of method Atomic::ctor
80

```

RevengeRat - Config Values in ldstr operations.

For more complex malware this will look almost exactly the same, with the exception that the strings will be encrypted.

The first step of dealing with more complex malware is locating the encrypted values using an identical process to what we're doing here with RevengeRat.

Below is an Asyncrat sample, where config values are loaded via ldstr operations before undergoing decryption.

```

205 /* 0x00000C47 800A000004 */ IL_005F: stsfld string OwoKdhsNoRhprcIA.aDVfuzQdwjg:CHtywGQHwHmhnr
206 /* 0x00000C4C 720C1E0070 */ IL_0064: ldstr "YqH/XXUS/bjYkcfUDLkmgQLTixGctueFXmvOvFyNogVlzhz7Ugasq4nuF4FLG0a5gpP06R5/ijn+UCVtBQ=="
207 /* 0x00000C51 800C000004 */ IL_0069: stsfld string OwoKdhsNoRhprcIA.aDVfuzQdwjg:RRIeFLmLco
208 /* 0x00000C56 72B1F00070 */ IL_006E: ldstr "jferdcblt4x1s8litzIV0gQ1UD4M0H90x7Z8/AmQUNyGktIw1t2B2wg0TcoB98Q4FyL1bTE4vbbxAsA=="
209 /* 0x00000C5B 800D000004 */ IL_0073: stsfld string OwoKdhsNoRhprcIA.aDVfuzQdwjg:xjYmymQEc5x
210 /* 0x00000C60 7242200070 */ IL_0078: ldstr "y19fB7KmsCCjnhuzmfzycBuyBffv6Qz7DpFw6TicszGhXqLSAK9VxuXauRk/uzV8ns4ClY5900FBF1PuiDwe=="
211 /* 0x00000C65 800F000004 */ IL_007D: stsfld string OwoKdhsNoRhprcIA.aDVfuzQdwjg:qZ2oafQqaPZhyuk
212 /* 0x00000C6A 72F5200070 */ IL_0082: ldstr "k/ZdL3IEsv3R6w3tCFT7z23c72pytWtSaeNeYtV7MvQyENkuRoAuBggCx2Frv669r4XVCzgcYo9+GD65c8FwA=="
213 /* 0x00000C6F 8010000004 */ IL_0087: stsfld string OwoKdhsNoRhprcIA.aDVfuzQdwjg:zAFzJlUimcq
214 /* 0x00000C74 14 */ IL_008C: ldsm11
215 /* 0x00000C75 8011000004 */ IL_008D: stsfld string OwoKdhsNoRhprcIA.aDVfuzQdwjg:otCxkZkdjNBX
216 /* 0x00000C7A 72A8210070 */ IL_0092: ldstr "3"
217 /* 0x00000C7F 8012000004 */ IL_0097: stsfld string OwoKdhsNoRhprcIA.aDVfuzQdwjg:BlCxovpkkkpC
218 /* 0x00000C84 72AC210070 */ IL_009C: ldstr "3HdjZnp+kJihQ0BpMkgdTVY7n5c3Imb/F82NYDTbvgrt2nB9Y9YD0oaqANQLcwhfzKIGeEhIoTIA6GIhAAdwe=="
219 /* 0x00000C89 8013000004 */ IL_00A1: stsfld string OwoKdhsNoRhprcIA.aDVfuzQdwjg:fe7FDPhwagt1nLc
220 /* 0x00000C9E 2A */ IL_00AE: ret
221 // end of method aDVfuzQdwjg::ctor
222
223 // end of class OwoKdhsNoRhprcIA.aDVfuzQdwjg
224

```

Encrypted + Base64 Config Values From Asyncrat. Note that the encrypted values are all loaded by a "ldstr" operation.

Interacting with Dotnet Using Python

Now that we have located the plaintext configuration inside of our file, we want to locate those same values using an automated script.

To do this, we will use Python and the dnlib library.

The following code will load the revenge.bin file into Python using dnlib .

Note that "dnlib.dll" must be inside the same directory as your script.

```
File Edit Format Run Options Window Help
import clr

clr.AddReference("dnlib")

import dnlib

from dnlib.DotNet import *
from dnlib.DotNet.Emit import OpCodes

module = dnlib.DotNet.ModuleDefMD.Load("revenge.bin")
```

For all future code snippets, we will assume you have the above code at the beginning of your script. This ensures that all the relevant libraries and options are imported.,

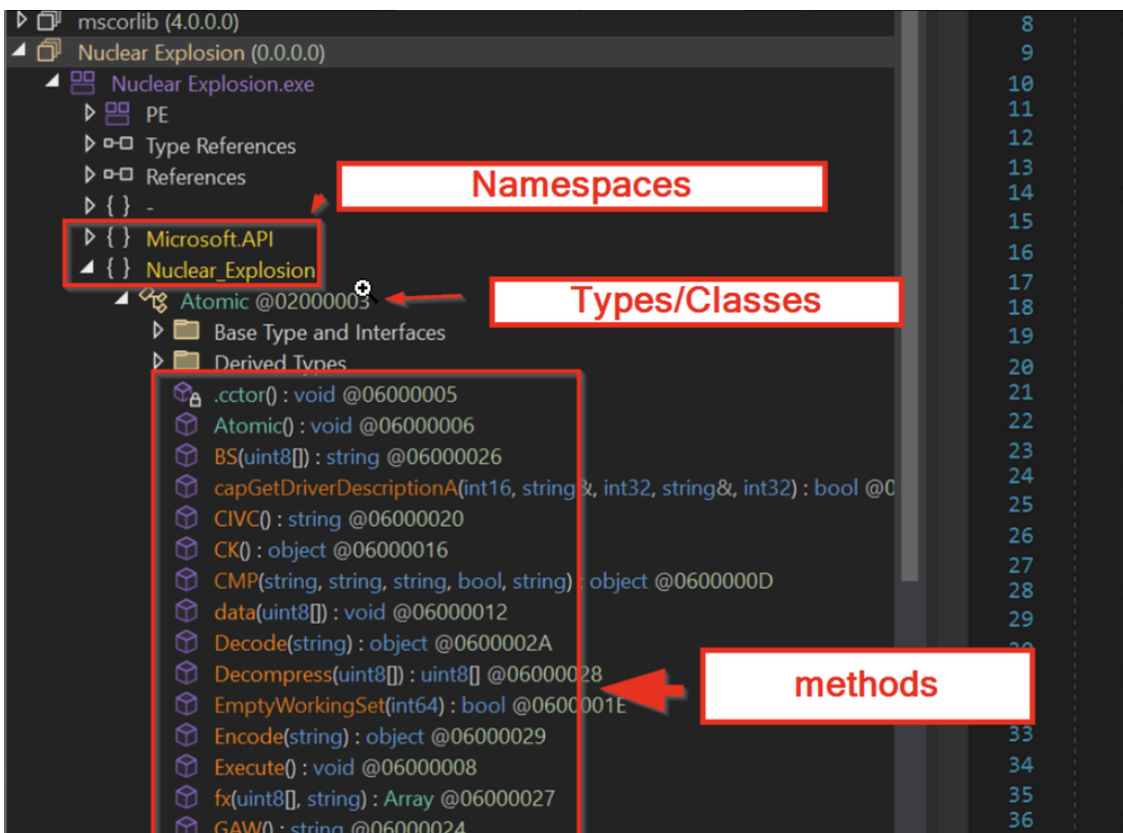
With the module now loaded, we can perform some simple operations to replicate our process in Dnspy.

For example, we can list all available namespaces to match that of Dnspy. They aren't in the same order but you can see that they are all there.

Note that when using dnlib, everything has to be first accessed via it's associated class/type.

Eg type → namespace (to obtain a namespace, you must first access a type) or type → method (To obtain a method/function, you must first access a type.)

This is slightly different to how dnspy displays namespace → type → method

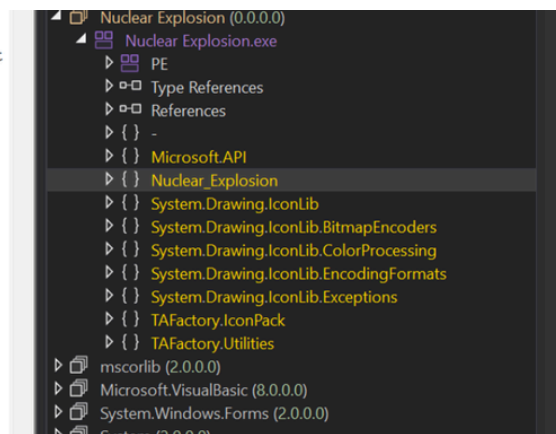


- `for type in module.GetTypes()` - this enumerates all types within the malware.
- `if type.Namespace not in namespaces` - this is to avoid printing the same namespace twice.
- `namespaces.append(type.Namespace)` - adds the namespace to a list
- `print(type.Namespace)` - this prints the namespace

```
namespaces = []
for type in module.GetTypes():
    if type.Namespace not in namespaces:
        namespaces.append(type.Namespace)
        print(type.Namespace)
```

```
for more information.
>>>
===== RESTART: C:\Users\Lenny\Desktop\revengerat
\revenge-extractor.py =====

Nuclear_Explosion
System.Drawing.IconLib
TAFactory.IconPack
System.Drawing.IconLib.ColorProcessing
System.Drawing.IconLib.Exceptions
System.Drawing.IconLib.EncodingFormats
System.Drawing.IconLib.BitmapEncoders
Microsoft.API
TAFactory.Utilities
>>> |
```



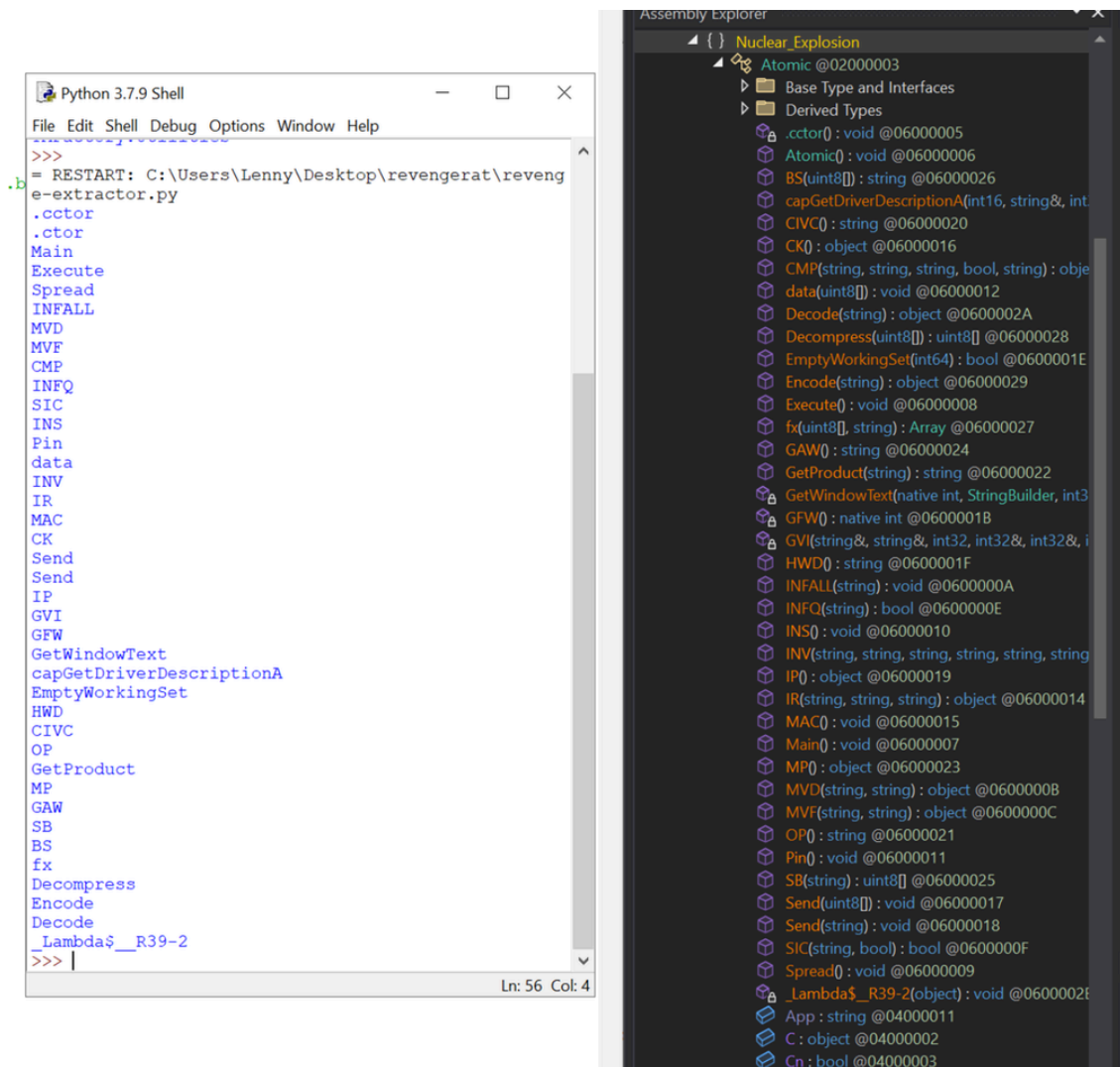
To obtain all available methods in the `Nuclear_Explosion` namespace, we can do something like this. Note that the types must be referenced first.

```
namespaces = []
for type in module.GetTypes():
    if type.Namespace == "Nuclear_Explosion":
        for method in type.Methods:
            print(method.Name)
```

This will display all available methods in the `nuclear_explosion` namespace. Although they are in a slightly different order by default.

Note that since the `Atomic()` method has the same name as the parent type of `Atomic`, it is classed as a constructor as is named as `.ctor` when accessed via `dnlib`.

This is slightly confusing but something you have to get used to if you haven't worked with object oriented (c#, java etc) code before.



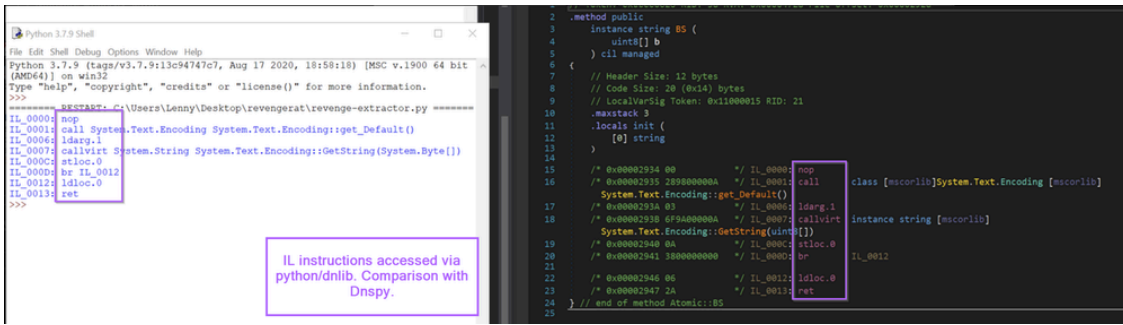
Accessing IL Instructions

If we hone in on a particular method name, we can obtain the IL instructions just as they were seen in dnspy.

In this case we have chosen the `BS` method, simply because it's short and easy to demonstrate the concept.

```
namespaces = []
for type in module.GetTypes():
    if type.Namespace == "Nuclear_Explosion":
        for method in type.Methods:
            if method.Name == "BS":
                for instr in method.Body.Instructions:
                    print(instr)
```

Below, see how the IL instructions printed via python match those displayed via Dnspy.



Now, we can make it more interesting and do the same with the original Atomic() method that contains the relevant config.

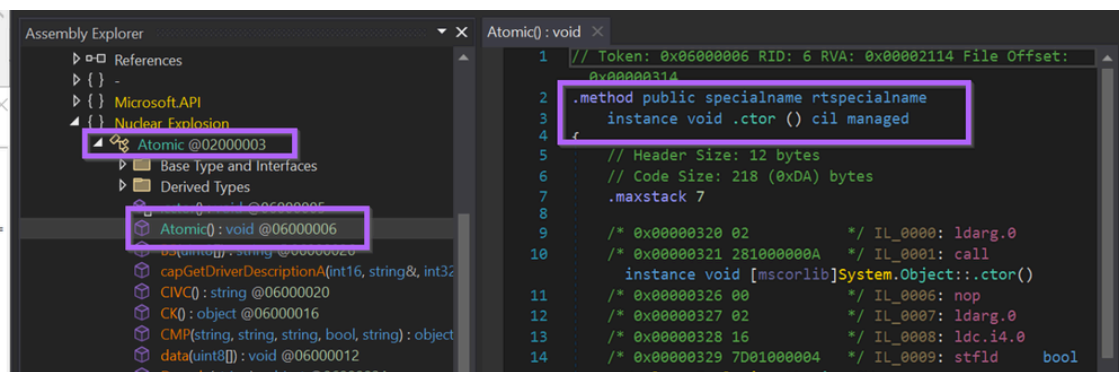
Note that since Atomic() has the same name as the Atomic type/class, it is classified as a constructor which is shortened to .ctor .

If you haven't worked with object oriented code before, it may be worth googling constructors to get a basic understanding of what they are.

TLDR:

- Constructors are methods/functions that are automatically executed when an object/type/class is created.
- Constructors have the same name as the parent object/type/class.
- Values that require initialization (eg config), are very often found in the constructor for the relevant class/type/object.

For now, just know that the config is inside the .ctor method and you will see this often.

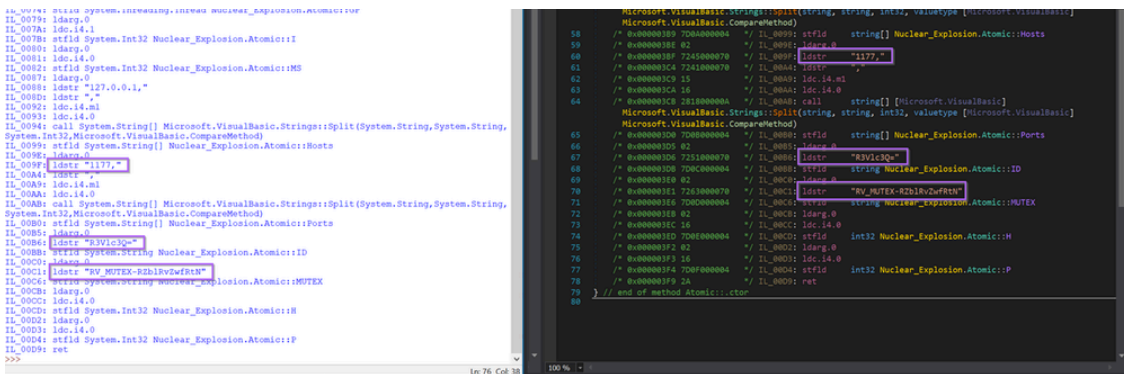


With this knowledge, we can change the previous code to print instructions for the .ctor method.

Using the previous code and updating the method name to .ctor , we can print all of the relevant instructions to match that of Dnspy.

```
namespaces = []
for type in module.GetTypes():
    if type.Namespace == "Nuclear_Explosion":
        for method in type.Methods:
            if method.Name == ".ctor":
                for instr in method.Body.Instructions:
                    print(instr)
```

In the printed instructions, we can see the IL instructions containing plaintext config values. The same as can be seen in Dnspy.



The config values are all referenced via `ldstr` operations. The script can be modified to only print instructions containing `ldstr`.

(Make sure you have the line `from dnlib.Dotnet.Emit import OpCodes` line at the beginning of your script)

```
namespaces = []
for type in module.GetTypes():
    if type.Namespace == "Nuclear_Explosion":
        for method in type.Methods:
            if method.Name == ".ctor":
                for instr in method.Body.Instructions:
                    if instr.OpCode == OpCodes.Ldstr:
                        print(instr)
```

With the additional filtering for `ldstr` operations, running the script will now output the config related instructions.

```
===== RESTART: C:\Users\Lenny\Desktop\r
IL_0088: ldstr "127.0.0.1,"
IL_008D: ldstr ","
IL_009F: ldstr "1177,"
IL_00A4: ldstr ","
IL_00B6: ldstr "R3V1c3Q="
IL_00C1: ldstr "RV_Mutex-RZblRvZwfrtN"
>>> |
```

Modifying the final line to print only `instr.Operand` makes the output even cleaner.

```
namespaces = []
for type in module.GetTypes():
    if type.Namespace == "Nuclear_Explosion":
        for method in type.Methods:
            if method.Name == ".ctor":
                for instr in method.Body.Instructions:
                    if instr.OpCode == OpCodes.Ldstr:
                        print(instr.Operand)
```

```
===== RESTART: C:\Users\Lenny\Desktop\r
127.0.0.1,
',
1177,
',
R3V1c3Q=
RV_Mutex-RZblRvZwfrtN
>>> |
```

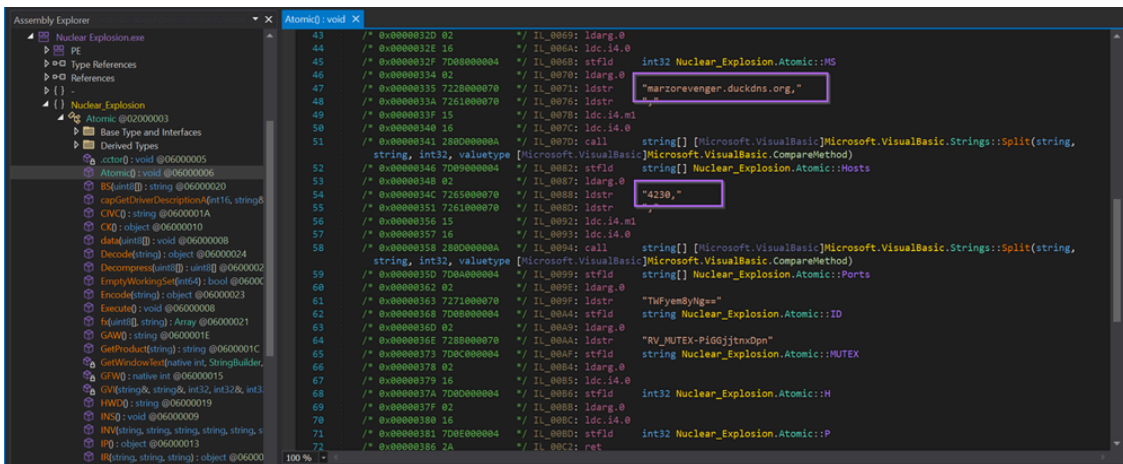
At this point. You can add your own code to provide additional formatting and or adjustments to the values. we won't really cover that here as the format requirements will be different for everyone.

Testing on additional Samples

From here, you can obtain an additional sample for testing.

In this case, we have used the sample.

```
2b89a560332bbc135735fe7f04ca44294703f3ae75fdfe8e4fc9906521fd3102
```



Running the script on the second file produces the following results.

```

===== RESTART: C:\Users\Lenny\Desktop
marzorevenger.duckdns.org,
'
4230,
'
TWfyem8yNg==
RV_Mutex-PiGGjJtnxDpn
>>>
    
```

Adding Resilience By Improving Method Signatures.

At this point, you can obtain config values from other samples. But this assumes that the additional samples have not employed any obfuscation and have kept the same method/namespace/class names.

Now there is just one problem, what happens if the malware author decides to modify any of those?

The sample `dd203194d0ea8460ac3173e861737a77fa684e5334503867e91a70acc7f73195` introduces this exact problem.

This sample uses largely the same structure as before, but uses randomized namespace and type names.

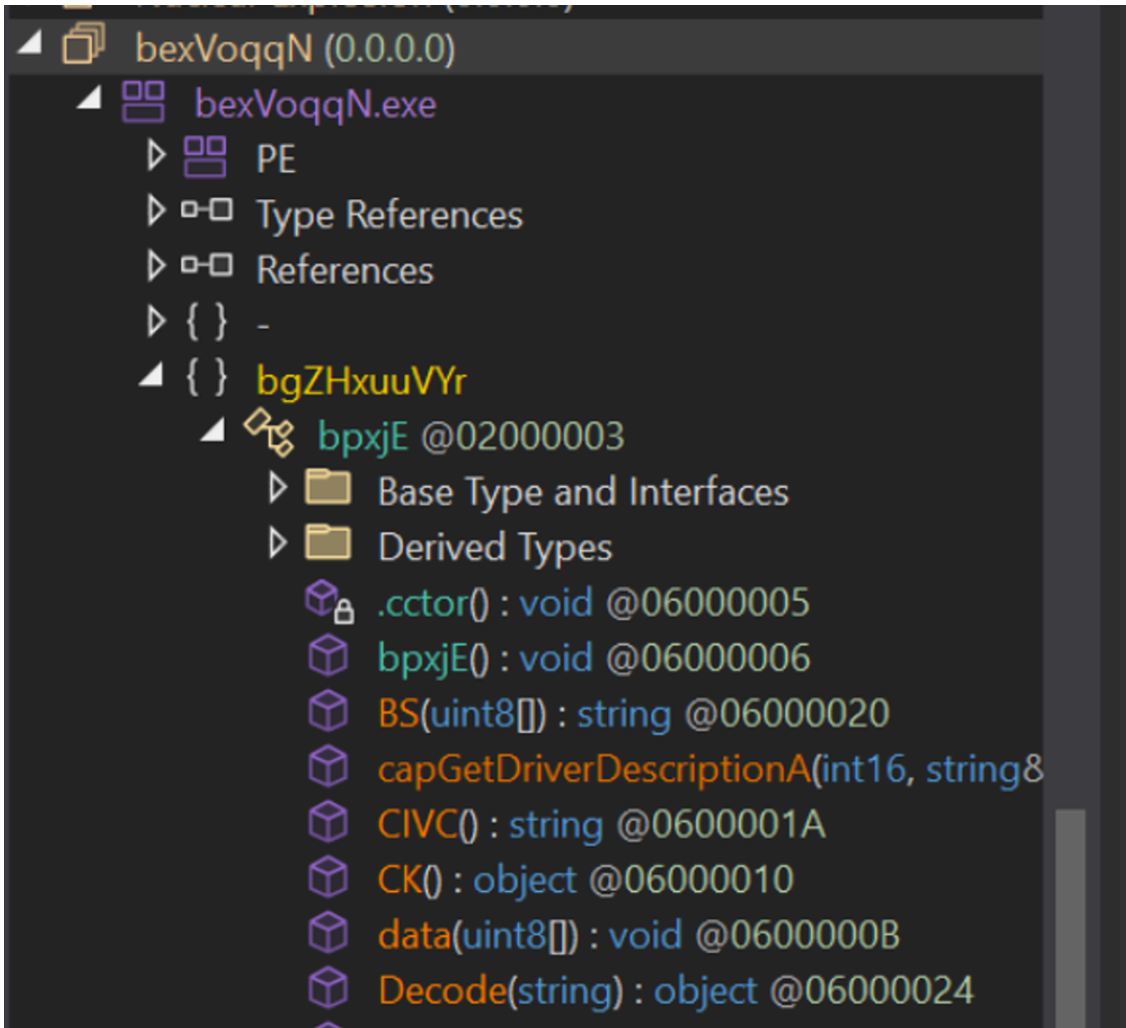
This breaks our original script as there is no `Nuclear_Explosion` namespace or `Atomic` class to signature from.

Running the script on the new sample produces no results.

```

RV_Mutex-PiGGjJtnxDpn
>>>
===== RESTART: C:\Users\Lenny\Desktop\revengerat\revenge-extractor.py =====
>>>
    
```

We can see below that the code is largely the same, but the method and class names are different.

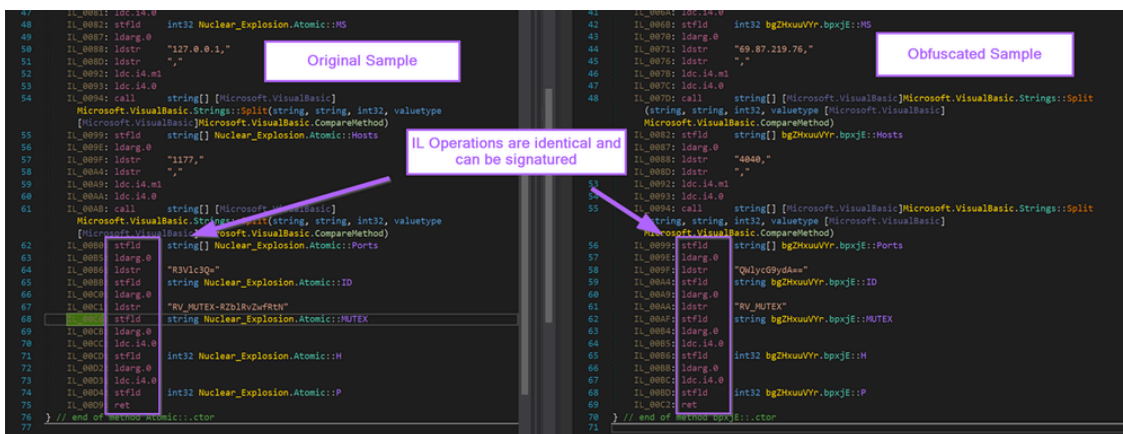


There are some similarities in other method names, (data, decode, BS etc) but these could be easily changed as well so we will avoid using this as part of a signature.

For the most resilient approach, we will instead use the IL operations.

(There are other signature opportunities, but they will not be covered in this post)

See below, the obfuscated sample and the original sample contain the same IL instructions for loading config values.



If we implement the following code. We can enumerate all available types and methods in the obfuscated sample, printing all values contained in `ldstr` operations.

- `has_config_pattern(method)` - a (currently) empty function for enumerating configuration patterns.
- `method.HasBody` - this ensures that empty methods/functions are skipped.

```
def has_config_pattern(method):
    return True

namespaces = []
for type in module.GetTypes():
    for method in type.Methods:
        if has_config_pattern(method) and method.HasBody:
            for instr in method.Body.Instructions:
                if instr.OpCode == OpCodes.Ldstr:
                    print(instr.Operand)
```

This script will enumerate all `ldstr` operations within the obfuscated file and print the loaded value.

Technically, this prints the config values, but it also prints 269 other string values which are not useful. So we want to improve the `has_config_pattern` function to hone in only on the methods containing relevant IL instructions.

(Note that we are using the initial file here for readability)

```

===== RESTART: C:\Users\Lenny\Desktop\revengerat\revenge-extractor.py =====
*~]NK[~*
Revenge-RAT
127.0.0.1,
,
1177,
,
R3Vlc3Q=
RV_Mutex-RZblRvZwfRtN
eNHuiGG.txt
RevengeRAT\44444.exe
eNHuiGG.txt
i True
Start
False
RevengeRAT
RevengeRAT

```

Let's modify the `has_config_pattern` function to filter on matching IL instructions.

For this example, we will use the last 14 instructions of the `Atomic` function. You can use more or less, experiment to see what works best for you.

```

54 IL_0093: ldc.i4.0
55 IL_0094: call string[] [Microsoft.VisualBasic]Microsoft.VisualBasic.Strings::Split
    (string, string, int32, valuetype [Microsoft.VisualBasic]
    Microsoft.VisualBasic.CompareMethod)
56 IL_0099: stfld string[] bgZHxuuVYr.bpxjE::Ports
57 IL_009E: ldarg.0
58 IL_009F: ldstr "QWlycG9ydA=="
59 IL_00A4: stfld string bgZHxuuVYr.bpxjE::ID
60 IL_00A9: ldarg.0
61 IL_00AA: ldstr "RV_Mutex"
62 IL_00AF: stfld string bgZHxuuVYr.bpxjE::Mutex
63 IL_00B4: ldarg.0
64 IL_00B5: ldc.i4.0
65 IL_00B6: stfld int32 bgZHxuuVYr.bpxjE::H
66 IL_00BB: ldarg.0
67 IL_00BC: ldc.i4.0
68 IL_00BD: stfld int32 bgZHxuuVYr.bpxjE::P
69 IL_00C2: ret
70 } // end of method bpxjE::.ctor
71

```

We will re-use one of the previous code snippets, which prints the `.ctor` IL instructions related to `Nuclear_Explosion`.

```

|
| for type in module.GetType():
|     if type.Namespace == "Nuclear_Explosion":
|         for method in type.Methods:
|             if method.Name == ".ctor":
|                 for instr in method.Body.Instructions:
|                     print(instr.Opcode.Name)
|

```

This prints a long list of instructions, but as mentioned, we will be using the last 14 for our signature.

```
54 IL_0093: ldc.i4.0
55 IL_0094: call string[] [Microsoft.VisualBasic]Microsoft.VisualBasic.Strings::Split
(string, string, int32, valuetype [Microsoft.VisualBasic]
Microsoft.VisualBasic.CompareMethod)
56 IL_0099: stfld string[] bgZHxuuVYr.bpxjE::Ports
57 IL_009E: ldarg.0
58 IL_009F: ldstr "QWlycG9ydA=="
59 IL_00A4: stfld string bgZHxuuVYr.bpxjE::ID
60 IL_00A9: ldarg.0
61 IL_00AA: ldstr "RV_Mutex"
62 IL_00AF: stfld string bgZHxuuVYr.bpxjE::Mutex
63 IL_00B4: ldarg.0
64 IL_00B5: ldc.i4.0
65 IL_00B6: stfld int32 bgZHxuuVYr.bpxjE::H
66 IL_00BB: ldarg.0
67 IL_00BC: ldc.i4.0
68 IL_00BD: stfld int32 bgZHxuuVYr.bpxjE::P
69 IL_00C2: ret
70 } // end of method bpxjE::.ctor
71
```

```
ldc.i4.0  
call  
stfld  
ldarg.0  
ldstr  
stfld  
ldarg.0  
ldstr  
stfld  
ldarg.0  
ldc.i4.0  
stfld  
ldarg.0  
ldc.i4.0  
stfld  
ret  
>>> |
```

To generate a signature, we can copy out the values and create a string array like this.

```
signature = ["stfld", "ldarg.0", "ldstr", "stfld", "ldarg.0", "ldstr", "stfld", "ldarg.0", "ldc.i4.0", "stfld", "ldarg.0", "ldc.i4.0", "stfld", "ret"]
```

The entire code now looks like this.

```

29 signature = ["call", "stfld", "ldarg.0", "ldstr", "stfld", "ldarg.0", "ldstr", "stfld", "ldarg.0", "ldc.i4.0", "stfld", "ldarg.0", "ldc.i4.0", "stfld", "ret"]
30
31
32
33 def has_config_pattern(method):
34     if method.HasBody:
35         if len(method.Body.Instructions) >= len(signature):
36             ins = [x.OpCode.Name for x in method.Body.Instructions]
37             if ins[-len(signature):] == signature:
38                 return True
39             return False
40
41 for type in module.GetTypes():
42     for method in type.Methods:
43         if has_config_pattern(method) and method.HasBody:
44             for instr in method.Body.Instructions:
45                 if instr.OpCode == OpCodes.Ldstr:
46                     print(instr.Operand)
47

```

and the signature checking code `has_config_pattern` now looks like this.

- `method.HasBody` - this is a filter to ensure the checked method is not empty
- `if len(method.Body.Instructions) >= len(signature)` - this is a filter to ensure the checked method is at least as long as the signature.
- `ins = [x.OpCode.Name for x in method.Body.Instructions]` - this creates an array of instructions for method being checked.
- `x.OpCode.Name` - this obtains only the instruction opcode name, which produces an array that looks like our signature array.
- `if ins[-len(signature):] == signature` - we only want to check the last instructions against our signature. if our signature is 14 instructions, we only want to check the last 14 instructions against our signature.

```

def has_config_pattern(method):
    if method.HasBody:
        if len(method.Body.Instructions) >= len(signature):
            ins = [x.OpCode.Name for x in method.Body.Instructions]
            if ins[-len(signature):] == signature:
                return True
        return False

```

This is the most important piece of the `has_config_pattern` function. Which compares the final instructions against our signature.

```

len(method.Body.Instructions) >= len(signature):
    ins = [x.OpCode.Name for x in method.Body.Instructions]
    if ins[-len(signature):] == signature:
        return True

```

With the new signature added, we can remove the `.ctor` and `nuclear_explosion` check and re run against our original sample.

```

41 for type in module.GetTypes():
42     for method in type.Methods:
43         if has_config_pattern(method) and method.HasBody:
44             for instr in method.Body.Instructions:
45                 if instr.OpCode == OpCodes.Ldstr:
46                     print(instr.Operand)
47

```

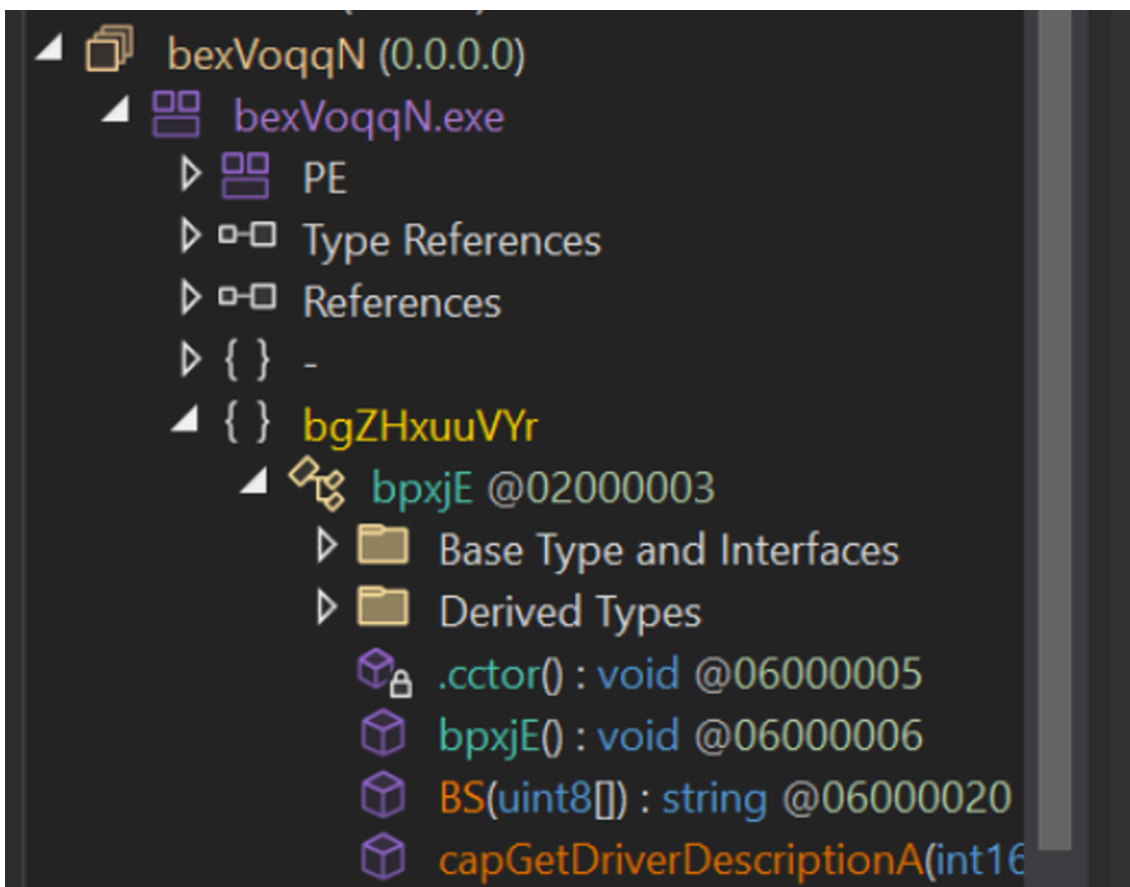
The config is found exactly as before. Despite the name signatures being removed. Only the IL instructions are used to locate the config values.

```
===== RESTART: C:\Users\Lenny\Desktop\revengerat\revenge-extractor.py =====  
Sample: 0d05942ce51fea8c8724dc6f3f9a6b3b077224f1f730feac3c84efe2d2d6d13e  
127.0.0.1,  
,  
1177,  
,  
R3V1c3Q=  
RV_Mutex-RZblRvZwfrtN  
>>>
```

Running Against The Obfuscated Sample.

Running the new code against the obfuscated sample

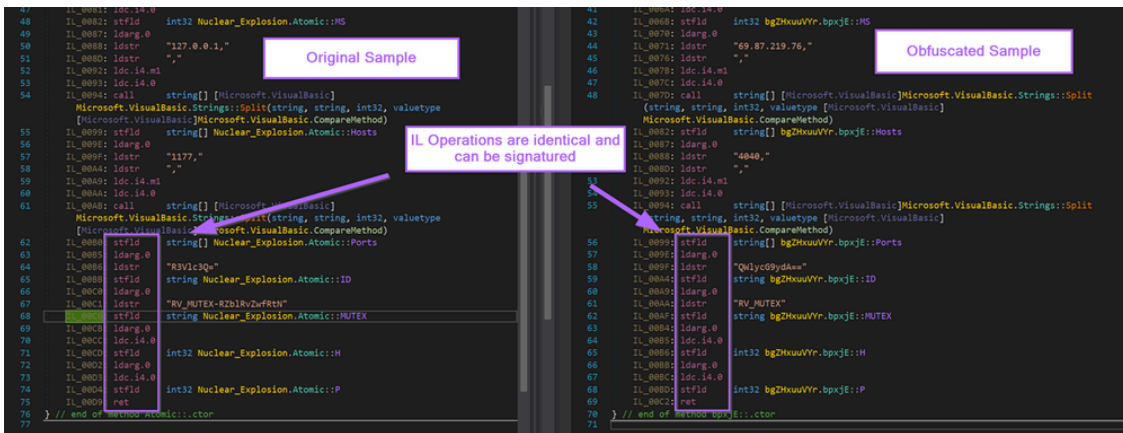
dd203194d0ea8460ac3173e861737a77fa684e5334503867e91a70acc7f73195 . The config values are able to be obtained.



```
= RESTART: C:\Users\Lenny\Desktop\revengerat\revengerat-bulk-samples\revenge  
Sample: dd203194d0ea8460ac3173e861737a77fa684e5334503867e91a70acc7f73195  
69.87.219.76,  
,  
4040,  
,  
QWlycG9ydA==  
RV_Mutex  
>>>
```

The configuration values are able to be extracted from both. Regardless of the fact that the method and class names are different between samples.

This is due to the identical opcode instructions between the two samples.



By very slightly modifying the script to take a filename as argument `sys.argv[1]`, we can implement a bulk extractor for many files.

```
import clr,sys
clr.AddReference("dnlib")
import dnlib
from dnlib.DotNet import *
from dnlib.DotNet.Emit import OpCodes

#2b89a560332bbcl35735fe7f04ca4294703f3ae75fdfe8efc9906521fd3102
#0d05942ce51fea8c8724dc6f3f9a6b3b077224f1f730feac3c84efe2d2d6d13e

filename = sys.argv[1]
module = dnlib.DotNet.ModuleDeMMD.Load(filename)

signature = ["call", "stfld", "ldarg.0", "ldstr", "stfld", "ldarg.0", "ldstr", "stfld", "ldarg.0", "ldc.i4.0", "stfld", "ldarg.0", "ldc.i4.0", "stfld", "ret"]

def has_config_pattern(method):
    if method.HasBody:
        if len(method.Body.Instructions) >= len(signature):
            ins = [x.OpCode.Name for x in method.Body.Instructions]
            if ins[-len(signature):] == signature:
                return True
            return False

results = []
for type in module.GetTypes():
    for method in type.Methods:
        if has_config_pattern(method) and method.HasBody:
            for instr in method.Body.Instructions:
                if instr.OpCode == OpCodes.Ldstr:
                    results.append(instr.Operand)

print("Sample: " + filename, end="")
print(": " + str(results))
```

For bulk extraction, the final code has been modified to print everything on a single line. As well as printing the filename.

```
results = []
for type in module.GetTypes():
    for method in type.Methods:
        if has_config_pattern(method) and method.HasBody:
            for instr in method.Body.Instructions:
                if instr.OpCode == OpCodes.Ldstr:
                    results.append(instr.Operand)

print("Sample: " + filename, end="")
print(": " + str(results))
```

This produces a slightly cleaner output for an individual file.

```
== RESTART: C:\Users\Lenny\Desktop\revengerat\revengerat-bulk-samples\revenge-extractor.py
Sample: 2b89a560332bbcl35735fe7f04ca4294703f3ae75fdfe8efc9906521fd3102: ['marzorrevenger.duckdns.org.', ' ', '4230.', ' ', 'TWYemByNg==', 'RV_Mutex-PiGGjtnxDpn']
>>>
```

Now, if we can obtain a set of samples (We used unpacme).

We can combine this with a short powershell script for bulk config extraction.

This particular script has been placed in a folder with lots of RevengeRat Samples.

```

Untitled1.ps1* X
1 get-childitem | foreach-object {
2
3     try {
4         $filename = $_.BaseName
5         .\revenge-extractor.py $filename 2>null
6     } catch {
7         continue
8     }
9
10 }
11
12

```

The sample folder is shown below

Mode	LastWriteTime	Length	Name
-----	24/09/2023	9:32 AM	32768 0142e023c883fb1f4e242f90c3da6471843350752ed0dae003f2dfcd1d7a36
-----	24/09/2023	9:32 AM	111104 027b0c6fd86bf513a28604131fe2506e34452098521589019d3d008ac4936
-----	24/09/2023	9:32 AM	17408 0594e5fcd393c8681681b59ad0106f21e494219cfa923d0c45f725ef904d4dc
-----	24/09/2023	9:32 AM	108032 0d05942ce51fea8c8724dc6f3f9a6b3b077224f1f730feac3c84efe2d2d6d13e
-----	24/09/2023	9:32 AM	17408 1381a7c5f54e8ba7929f3169e4ef4a115117a7318b785dd457e38fcc7ec71d8
-----	24/09/2023	9:32 AM	51712 1396d05d838bf36e2433a88d9ef56390d564e62384273cb54bd26971ab6f
-----	24/09/2023	9:32 AM	17408 2b89a560332bbc135735fe7f04ca4294703f3ae75df8e8e4f9906521fd3102
-----	24/09/2023	9:32 AM	16896 2e0e188d4b83df3c8bbcd3227493a9074e668b84a48b9dc81dacc596f23e048
-----	24/09/2023	9:32 AM	24576 32b0c48d95e9c4ef2860368bee244489b6e3211194d51fafa7d9f755e0ee99
-----	24/09/2023	9:32 AM	16896 43023de4ae38501491783084f7add67713f186b84bb044d51f048d468d95d981
-----	24/09/2023	9:32 AM	14848 4af8536f98e03dfc35f5be911ff79ef6c0cbb942c855b0dca530b3058f34b5
-----	24/09/2023	9:32 AM	16896 4c05f04586dc80abe1f718418a12080f3ae72038afbd124f01d0844512d45c5
-----	24/09/2023	9:32 AM	17408 5973a09f51c0bca9f3aee715ac7f5fba39602ffa9525579bc4da1ae45acd071d
-----	24/09/2023	9:32 AM	17408 5bd6dff736f873bfcc21c99e87b5631c9e20944bfaf057b25f2a042af40b47
-----	24/09/2023	9:32 AM	32768 5d6a6d517bb5cfb574d0939810c1b55c2a813cad751b19eed1ad144c8f797830
-----	24/09/2023	9:32 AM	17408 6481f9e27bec4cf6702b6d6a09761c62782f5010da0dfda0396575c60200279d
-----	24/09/2023	9:32 AM	32768 71e66a25e80c133a00694b23f8f807578d45b1976368a4749c7frcf524efc6bf4
-----	24/09/2023	9:32 AM	143872 880ac454f385019390e07ff37f1986f8b069514113d6d774df90a5744f88af
-----	24/09/2023	9:32 AM	14848 8bfdd772f96c76463e5183114b485834eb3c8210e0dd5346d789fa038df552
-----	24/09/2023	9:32 AM	5243360 91caa1fe289dc8d8500399b3dc07a5147223126a8cf46833ece052accaeb27ff
-----	24/09/2023	9:32 AM	17408 988aca1597e99ee39398f581dbf2f2cced7df018191cf66527fa61110c2d24b
-----	24/09/2023	9:32 AM	17920 a118f361223ac18069b6ae8b9baec7e918a99b42ea171250c3e9bc4c314a8b2e
-----	24/09/2023	9:32 AM	17408 a895d787d2719a70d7d3722b75bf3ac9b16c901e06ab1ebbb56da33e9ce6b03
-----	24/09/2023	9:32 AM	17408 d00a3b76204d44a85526e84f9597c754e8ddc5b4d66d777c92f94526a8602
-----	24/09/2023	9:32 AM	32768 d6974dd886a8078979729b2c09b01babeaf21c3599ef19004437418b315076
-----	24/09/2023	9:32 AM	17408 dd2031940da8460ac3173e8617377af684e5334503867e91a70acc7f73195
-a-----	1/07/2023	1:54 AM	1167872 dn1b.d11
-----	24/09/2023	9:32 AM	17408 e3df2679e87091bbd64407bf59b25f0ced5a63a5e2fd193d4bdc17ab928085
-----	24/09/2023	9:32 AM	139264 ea0c4df308a6b13c6ec10f00a3bcd9cf38ed382a753f848f14d5b6fa24b84f
-----	24/09/2023	9:32 AM	787712 ef7bac23b92c086b72c70ff5eb23504ab472e0c7d6a7c28461fd8fa646e1a4ae
-----	24/09/2023	9:32 AM	17920 f4dd9e0e6ad2721ca3813c8f6662c2172a72deb33ca0d05346a4fade6473870
-----	24/09/2023	9:32 AM	24576 fb2c58f9846adcb295edd3c8a5baeac31fff3bc98f6503d04e95ba3f9f072e8
-----	24/09/2023	9:32 AM	16896 f88fd27964a3c75d662eddb71fab9ce9a9a7ffc0c6e78e2815e31a06856aca5
-----	24/09/2023	9:32 AM	32768 f8c21d101b2c979907ea72ba52955e7745a5c835b9d80656ecfe24653d4ffa
-----	24/09/2023	9:32 AM	20480 fa95d5e77fd4fab91662c9b1e460807647acb2576949110b59fb6485b17cc8d
-a-----	24/09/2023	9:32 AM	14848 fd775cb2dc7c7fe6315e06da2e80fa20a68adf084dbf62ac0f0a2c7f7b7313
-a-----	2/10/2023	2:12 AM	3056 null
-a-----	2/10/2023	2:12 AM	1422 revenge-extractor.py

Running the powershell script, produces the following results. There are some failures but the extractor mostly works. The failures are due to slightly differing patterns in some obfuscated samples. This is something that will be covered in a future post.

```

PS C:\Users\Leny\Desktop> .\revenge-extractor -bulk -samples C:\Users\Leny\Desktop\revenge\revenge_bulk.ps1
Sample: 0142e023c883fb1f4e242f90c3da6471843350752ed0dae003f2dfcd1d7a36 [178.17.174.71, 3310, 'R3V1C3Q=', 'RV_MUTEX-HxdYuaWcGnhp']
Sample: 027b0c6fd86bf513a28604131fe2506e34452098521589019d3d008ac4936 [178.175.233.52, 333, 'R3V1C3Q=', 'RV_MUTEX']
Sample: 0594e5fcd393c8681681b59ad0106f21e494219cfa923d0c45f725ef904d4dc [teststest.dns.net, 333, 'R3V1C3Q=', 'RV_MUTEX']
Sample: 0d05942ce51fea8c8724dc6f3f9a6b3b077224f1f730feac3c84efe2d2d6d13e [127.0.0.1, 1177, 'R3V1C3Q=', 'RV_MUTEX-R2D1RvZwfrTn']
Sample: 1381a7c5f54e8ba7929f3169e4ef4a115117a7318b785dd457e38fcc7ec71d8 [pp1foot1.dns.net, 1177, 'R3V1C3Q=', 'RV_MUTEX-wpnFw0n0rUu']
Sample: 1396d05d838bf36e2433a88d9ef56390d564e62384273cb54bd26971ab6f [marzorevenger.duckdns.org, 4230, 'TWfYem8Yqmg=', 'RV_MUTEX-P1G6jTnx0pn']
Sample: 2b89a560332bbc135735fe7f04ca4294703f3ae75df8e8e4f9906521fd3102 [127.0.0.1, 1000, 'R3V1C3Q=', 'RV_MUTEX']
Sample: 2e0e188d4b83df3c8bbcd3227493a9074e668b84a48b9dc81dacc596f23e048 [192.168.1.4, 1231, 'R3V1C3Q=', 'RV_MUTEX']
Sample: 32b0c48d95e9c4ef2860368bee244489b6e3211194d51fafa7d9f755e0ee99 [iamer1.dns.net, 333, 'R3V1C3Q=', 'RV_MUTEX']
Sample: 43023de4ae38501491783084f7add67713f186b84bb044d51f048d468d95d981 [h0zok1.duckdns.org, 333, 'RV_MUTEX-boomCGFBvTzCM']
Sample: 4af8536f98e03dfc35f5be911ff79ef6c0cbb942c855b0dca530b3058f34b5 [mallorca.myftp.org,mbvd.hopto.org, 5198,5198, 'TWf5', 'RV_MUTEX-D1gZb1RvZwfrTn']
Sample: 4c05f04586dc80abe1f718418a12080f3ae72038afbd124f01d0844512d45c5 [alougrtz.dns.net, 7777, 'RV_MUTEX', 'RV_MUTEX-UqZb1RvZwfrTn']
Sample: 4f8536f98e03dfc35f5be911ff79ef6c0cbb942c855b0dca530b3058f34b5 [127.0.0.1, songs-travel.at,0lv.gp, tcp://tcp.eu.ngrok.io, 333,127.28.12792, 'R3V1C3Q=', 'RV_MUTEX']
Sample: 5973a09f51c0bca9f3aee715ac7f5fba39602ffa9525579bc4da1ae45acd071d [mallorca.myftp.org,mbvd.hopto.org, 5198,5198, 'TWf5', 'RV_MUTEX-D1gZb1RvZwfrTn']
Sample: 5bd6dff736f873bfcc21c99e87b5631c9e20944bfaf057b25f2a042af40b47 [178.17.174.71, 3310, 'R3V1C3Q=', 'RV_MUTEX-HxdYuaWcGnhp']
Sample: 5d6a6d517bb5cfb574d0939810c1b55c2a813cad751b19eed1ad144c8f797830 [127.0.0.1.4, tcp.ngrok.io, 333,16025, 'R3V1C3Q=', 'RV_MUTEX']
Sample: 6481f9e27bec4cf6702b6d6a09761c62782f5010da0dfda0396575c60200279d [20.241.4.184, 30, 'RV_MUTEX', 'RV_MUTEX']
Sample: 6481f9e27bec4cf6702b6d6a09761c62782f5010da0dfda0396575c60200279d [marzorevenger.duckdns.org, 4230, 'TWfYem8Yqmg=', 'RV_MUTEX-P1G6jTnx0pn']
Sample: 6481f9e27bec4cf6702b6d6a09761c62782f5010da0dfda0396575c60200279d [mallorca.myftp.org,mbvd.hopto.org, 5198,5198, 'TWf5', 'RV_MUTEX-D1gZb1RvZwfrTn']
Sample: 6481f9e27bec4cf6702b6d6a09761c62782f5010da0dfda0396575c60200279d [92.60.40.224, 55160, 'R3V1C3Q=', 'RV_MUTEX']
Sample: 6481f9e27bec4cf6702b6d6a09761c62782f5010da0dfda0396575c60200279d [a0c4df308a6b13c6ec10f00a3bcd9cf38ed382a753f848f14d5b6fa24b84f, hostservice.sytes.net, 1085, 'R3V1C3Q=', 'RV_MUTEX']
Sample: 6481f9e27bec4cf6702b6d6a09761c62782f5010da0dfda0396575c60200279d [127.0.0.1, 8090, 'R3V1C3Q=', 'RV_MUTEX']
Sample: 6481f9e27bec4cf6702b6d6a09761c62782f5010da0dfda0396575c60200279d [178.17.174.71, 3310, 'R3V1C3Q=', 'RV_MUTEX-HxdYuaWcGnhp']
Sample: 6481f9e27bec4cf6702b6d6a09761c62782f5010da0dfda0396575c60200279d [rhockgame1230.zapto.org, 6722, 'cchpdrjpbmg=', 'RV_MUTEX']

```

Conclusion and Final Takeaways

In this post, we have covered the basics of extracting configuration from a very basic dotnet malware sample. The techniques covered here form the basis of configuration extraction for most dotnet malware. Advanced samples will not store values in plaintext, but encrypted values will typically be stored in a very similar way via `ldstr` operations.

The initial steps (prior to decryption) for advanced samples will be the same as seen here today.

If you found any of this useful, consider signing up to the site. Signed up members will receive access to a discord server, bonus content and early access to future posts.

Sign up for Embee Research

Malware Analysis Insights

No spam. Unsubscribe anytime.

References

A collection of blogs and scripts that have helped me learn these concepts.

- RussianPanda - <https://russianpanda.com/2023/07/04/WhiteSnake-Stealer-Malware-Analysis/>
- N1ghtw0lf - <https://n1ght-w0lf.github.io/tutorials/dotnet-string-decryptor/>
- Polish Cert - <https://cert.pl/en/posts/2023/09/unpacking-whats-packed-dotrunpex/>
- OALabs Research - https://research.openanalysis.net/dotnet/static-analysis/stormkitty/dnlib/python/research/2021/07/14/dot_net_static_analysis.html

Full Script

```
"""  
  
Revenge Rat Config Extractor Example  
@embee_research  
  
Samples  
2b89a560332bbc135735fe7f04ca44294703f3ae75fdfe8e4fc9906521fd3102  
0d05942ce51fea8c8724dc6f3f9a6b3b077224f1f730feac3c84efe2d2d6d13e  
  
"""  
  
import clr,sys  
  
clr.AddReference("dnlib")  
  
import dnlib
```

