New TA402/MOLERATS Malware – Decrypting .NET Reactor Strings

0ffset.net/reverse-engineering/malware-analysis/molerats-string-decryption/

July 6, 2021

Offset Training Solutions

```
def locateOffsets(joined_array):

    filename = "lastconn.bin"
    mod = dnlib.DotNet.ModuleDefMD.Load(filename)
    opt = dnlib.DotNet.Writer.ModuleWriterOptions(mod)
    opt.MetadataOptions.Flags = opt.MetadataOptions.Flags | dnlib.DotNet.Writer.MetadataFlags.PreserveAll;
    #assembly = Assembly.LoadFrom(filename)
    totalNumTypes = 0
    stringMethodName = "pyM1eVFCveMv9BuGJ6"

    for var in mod.Types:
        if not var.HasMethods:
            pass

        for method in var.Methods:
            if not method.HasBody:
                break

            if not method.Body.HasInstructions:
                break

            i = 0
            operand = ""
            while i < len(method.Body.Instructions):

                operand = str(method.Body.Instructions[i].Operand).encode()
                if method.Body.Instructions[i].OpCode == OpCodes.Call and operand.find(str(stringMethodName).encode()) != -1:
                    keyValue = method.Body.Instructions[i - 1].GetLdcI4Value()
                    string_length = struct.unpack("I", joined_array[keyValue:keyValue + 4])[0]
                    string = joined_array[keyValue + 4:keyValue + 4 + string_length]
                    print str(keyValue) + ": " + string.replace("\x00", "")
                    #method.Body.Instructions[i - 1].OpCode = OpCodes.Nop
                    #method.Body.Instructions[i].OpCode = OpCodes.Ldstr
                    #method.Body.Instructions[i].Operand = string.replace("\x00", "")

                i += 1

    #mod.Write("cleaned_lastconn.bin")

    return
```

- 0verfl0w_
- 6th July 2021
- No Comments

It's sure been a while since the last post! We've gone through several iterations of website design over the past few months (plus fixing all the malformed images due to the theme transfer), but should be back for good now! For this commemorative post, we'll be diving into a recently discovered malware sample known as LastConn, a payload used by the MOLERATS APT group, which was obfuscated using .NET Reactor. The problem is, de4dot is unable to deobfuscate it, so the job falls upon us to do so. We'll be examining the string encryption routine, replicating it in Python, testing it manually, and then automating it somewhat to extract all string related indicators from the binary, and decrypt the relevant strings! Let's get into it!

LastConn MD5 Hash: D07654434D64B73FE8CB49CFB9B7E3FB

Table Of Contents

MOLERATS: Overview

MOLERATS, also known as TA402, are a Middle Eastern based APT group known for performing intrusions against Middle Eastern Government Organisations, including Israel, the UAE, and Turkey. The most recent campaign, discovered by ProofPoint, once again

targeted government organisations and organisations with diplomatic relationships in the Middle East. The prime focus of the attackers is to exfiltrate sensitive information in order to gather intelligence, using spear-phishing as an initial infection vector. In this campaign, ProofPoint discovered the threat actors utilising a somewhat new malware dubbed as LastConn.

LastConn: Overview

LastConn is believed to be an updated version of the SharpStage backdoor, previously discovered by CyberReason back in December 2020. The SharpStage backdoor, developed in .NET, utilised DropBox API for exfiltration, and had specific checks for Arabic on the infected machine. LastConn also implemented similar checks, as well as the DropBox API for communication. One of the discovered samples utilised an obfuscator that De4Dot could not successfully deobfuscate, known as .NET Reactor.

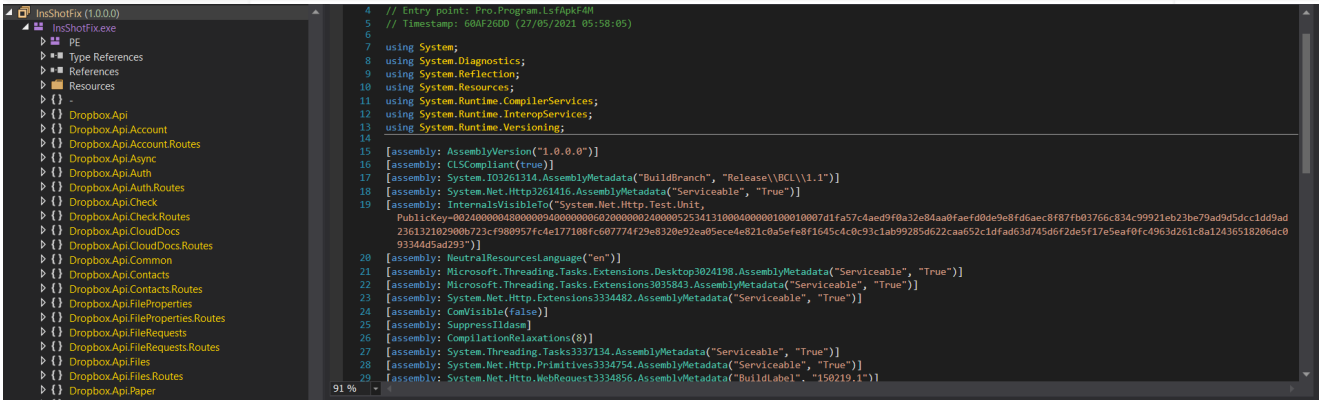
.NET Reactor: Overview

.NET Reactor is a *powerful code protection and software licensing system for software written for the .NET Framework, and supports all languages that generate .NET assemblies*. It is commercially available, and provides features such as string encryption, control flow obfuscation, and code virtualisation. A free trial is provided for the software as well, which seems to have been used by the threat actors, based on a string found in the list of decrypted strings. Talking about decrypted strings, let's start analysing!

Initial Analysis:

Opening up the initial sample in PEStudio, we can confirm that we are dealing with a fairly large .NET binary. At a first look, my thoughts were that the main payload was packed, resulting in the large file size, however upon opening the sample in dnSpy we can see that it is not packed – in fact we can see the unobfuscated symbols in the Assembly Explorer, with classes like *Dropbox.Api* and *Newtonsoft.Json* visible.

property	value
md5	D07654434D64B73FE8CB49CFB9B7E3FB
sha1	8FC864F028B59A3C4A34B013C119D79C5D72E24F
sha256	F55E2050733576FA16452E2589A187F4BF202CA3B54B1497BA2C006E8D3BDD45
md5-without-overlay	wait...
sha1-without-overlay	wait...
sha256-without-overlay	wait...
first-bytes-hex	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 00
first-bytes-text	M Z .. .. @ .. ..
file-size	10384896 (bytes)
size-without-overlay	wait...
entropy	6.239
imphash	F34D5F2D4577ED6D9CEEC516C1F5A744
signature	Microsoft Visual C# v7.0 / Basic .NET
entry-point	FF 25 00 20 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
file-version	1.1.0.0
description	Viewfile
file-type	**executable**
cpu	**32-bit**
subsystem	GUI
compiler-stamp	0x60AF26DD (Thu May 27 05:58:05 2021 - UTC)
debugger-stamp	n/a
resources-stamp	empty



As mentioned, .NET Reactor provides functionality to encrypt strings, obfuscate control flow, and even virtualise the .NET instructions in a similar fashion to x86 Assembly virtualisation, which, luckily in this case, the threat actors chose not to enable! Instead, the main methods of protection in this binary surround the string encryption and control flow obfuscation, as well as the addition of junk code. We can clearly see this in the entry point function, labelled as *LsfApkF4M()*.

```
namespace Pro
{
    // Token: 0x0200000E RID: 14
    internal static class Program
    {
        // Token: 0x06000053 RID: 83 RVA: 0x00004BA4 File Offset: 0x00002DA4
        [STAThread]
        private static void LsfApkF4M()
        {
            if (277629 - 212870 == 64759)
            {
                Application.EnableVisualStyles();
                if (Program.dQWY6qG82SAbCK3Pxa() == null)
                {
                }
            }
            do
            {
                Application.SetCompatibleTextRenderingDefault(false);
                if (!Program.V9yJ8yp9ahRYqnZDsG())
                {
                    break;
                }
                w4X4wl9MxwlqqYZEne.cBFhC9cq2dv3Z();
                if (Program.dQWY6qG82SAbCK3Pxa() != null)
                {
                    break;
                }
                Application.Run(new Form1());
            }
            while (75057 - 428994 == -353935);
        }
```

You'll probably notice the junk code (subtraction of 212870 from 277629) which will always return true. However, the junk code isn't limited to one-liners; two comparisons are performed between **null** and the return values from the function *dQWY6qG82SAbCK3Pxa()* – which will also return **null**. Therefore, we can now take note that a large number of functions in the binary are probably going to be made up of junk code that all return a constant value.

```
namespace Pro
{
    // Token: 0x0200000E RID: 14
    internal static class Program
    {
        // Token: 0x06000053 RID: 83 RVA: 0x00004BA4 File Offset: 0x00002DA4
        [STAThread]
        private static void LsfApkF4M()
        {
            if (277629 - 212870 == 64759)                          Always True
            {
                Application.EnableVisualStyles();
                if (Program.dQWY6qG82SAbCK3Pxa() == null)          Always Null
                {
                }
            }
            do
            {
                Application.SetCompatibleTextRenderingDefault(false);
                if (!Program.V9yJ8yp9ahRYqnZDsG())                 Always True
                {
                    break;
                }
                w4X4wl9MxwlqqYZEne.cBFhC9cq2dv3Z();
                if (Program.dQWY6qG82SAbCK3Pxa() != null)
                {
                    break;
                }
                Application.Run(new Form1());
            }
            while (75057 - 428994 == -353935);                     Never True
        }
```

The main function of importance inside the entry point is at the very end, where we see the sample will execute *Form1()* – this is where the interesting stuff occurs. However, just before that we do encounter the very first "anti-analysis" check, which is a date check. The sample will refuse to continue execution if it has been executed after the 30th of June, 2021. While this is not the most interesting function, it does show us the first instance of a string

decryption function. If the sample has been executed after the 30th of June, an exception will be thrown. The argument passed to the *Exception()* call is a string, and is returned from the function *MYcw9uffxdYPAXmUtn.pyM1eVFCveMv9BuGJ6()*.

```
internal class w4X4wl9MxwlqqYZEne
{
    // Token: 0x060130F8 RID: 78072 RVA: 0x00487268 File Offset: 0x00485468
    internal static void cBFhC9cq2dv3Z()
    {
        if (274047 - 500368 == -226321)
        {
            if (!w4X4wl9MxwlqqYZEne.fJUdDPDl3PK)
            {
                if (210108 - 312061 != -101952)
                {
                    do
                    {
                        w4X4wl9MxwlqqYZEne.fJUdDPDl3PK = true;
                    }
                    while (!w4X4wl9MxwlqqYZEne.M4GTwqHcCC7j2EqYGjf4());
                    TimeSpan timeSpan = DateTime.Now - new DateTime(2021, 6, 16);
                    if (21804 - 154488 == -132684)
                    {
                        if (Math.Abs(timeSpan.Days) >= 14)
                        {
                            throw new Exception(MYcw9uffxdYPAXmUtn.pyM1eVFCveMv9BuGJ6(208444));
                        }
                        if (!w4X4wl9MxwlqqYZEne.M4GTwqHcCC7j2EqYGjf4())
                        {
                        }
                    }
                }
            }
        }
    }
```

Upon checking the identified “anti-analysis” function again, and seeing that the function would display “*This assembly is protected by an unregistered version of Eziriz’s “.NET Reactor”! This assembly won’t further work.”,* it became clear that this is more likely to be a function created by .NET Reactor, to prevent the obfuscated payload working after a specific trial end date.

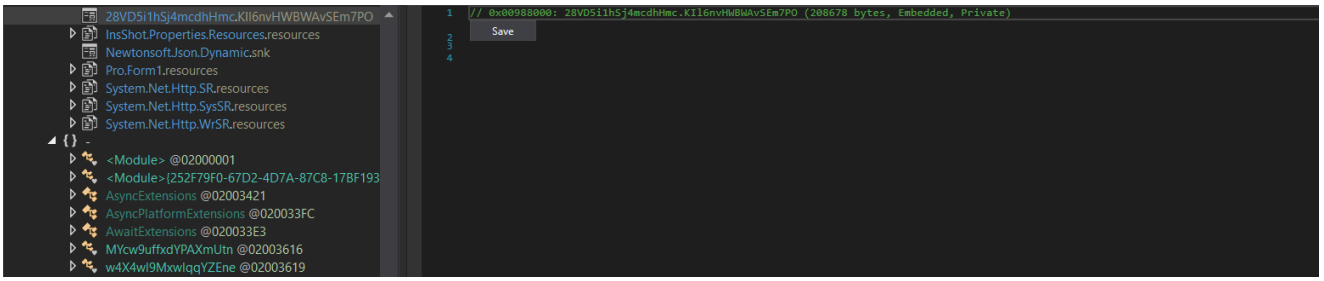
This particular function has the value 208444 passed as an argument, indicating that the strings are potentially stored in some kind of an array/list. Regardless, we have now found what looks to be a string encryption routine, so let’s dive in!

String Decryption:

Unfortunately for us, the control flow of this function has been highly obfuscated, with multiple goto’s, loops, and plenty of conditional statements. However, we can piece some information together just by scrolling through the lines of code. Firstly, there is an **array** variable which is constantly changing through the flow of execution, at least in the first loop. It changes so often it would be very time consuming to manually calculate the bytes inside the variable, and so dynamic analysis will have to be used.

```
[MYcw9uffxdYPAXmUtn.UDeI7RWk9e8HZbAF57(typeof(MYcw9uffxdYPAXmUtn.UDeI7RWk9e8HZbAF57.TlgJaUaQEO0K1skGJY<object>[]))]
internal static string pyM1eVFCveMv9BuGJ6(int M448gdJtBGnIC5sjsy)
{
    if (65861 - 576208 == -510347)
    {
        if (MYcw9uffxdYPAXmUtn.aAgdDBUcpQV.Length == 0)
        {
            if (MYcw9uffxdYPAXmUtn.yUVa63uLpvJGVHA1q0() == null)
            {
                goto IL_264C;
            }
            IL_44:
            BinaryReader binaryReader;
            try
            {
                binaryReader.BaseStream.Position = 0L;
                if (!MYcw9uffxdYPAXmUtn.ixZOAtj3qEPamiJbob())
                {
                    goto IL_DE3;
                }
                goto IL_B32;
                int num;
                do
                {
                    IL_77E:
                    num = 100 + 85;
                }
                while (MYcw9uffxdYPAXmUtn.yUVa63uLpvJGVHA1q0() != null);
                byte[] array;
                array[25] = (byte)num;
                if (85334 - 185858 == -100523)
                {
```

Next, we can see a variable named **binaryReader** is referred to quite a lot throughout the function. A simple CTRL+F indicates this will contain data read from the resource *28VD5i1hSj4mcdhHmc.KIl6nvHWBWAvSEm7PO*. Initially, this data does not seem to be used for anything interesting, however it is a strange blob of data and is still referenced in the function, so let's go ahead and extract that to be used later on.



At the very end of the string decryption function, we can clearly see the variable returned is named **string**, and it is retrieved through the variable **array3**. **array3** is initialized above, with a *Copy()* call, which will copy the data from **MYcw9uffxdYPAXmUtn.aAgdDBUcpQV** to **array3**. Another point of interest is the usage of the argument **M448gdJtBGnIC5sjsy** as the index – this argument is equal to 208444 in the first call to this function.

```
int num5 = BitConverter.ToInt32(MYcw9uffxdYPAXmUtn.aAgdDBUcpQV, M448gdJtBGnIC5sjsy);
if (!MYcw9uffxdYPAXmUtn.ixZOAtj3qEPamiJbob())
{
    goto IL_21;
}
IL_26A7:
string @string;
try
{
    byte[] array3 = new byte[num5];
    if (!MYcw9uffxdYPAXmUtn.ixZOAtj3qEPamiJbob())
    {
    }
    for (;;)
    {
        Array.Copy(MYcw9uffxdYPAXmUtn.aAgdDBUcpQV, M448gdJtBGnIC5sjsy + 4, array3, 0, num5);
        if (MYcw9uffxdYPAXmUtn.ixZOAtj3qEPamiJbob())
        {
            @string = Encoding.Unicode.GetString(array3, 0, array3.Length);
            if (MYcw9uffxdYPAXmUtn.ixZOAtj3qEPamiJbob())
            {
                break;
            }
        }
    }
}
catch
{
    if (!MYcw9uffxdYPAXmUtn.ixZOAtj3qEPamiJbob())
    {
    }
    goto IL_21;
}
return @string;
```

MYcw9uffxdYPAXmUtn.aAgdDBUcpQV is initialised using data inside 3 variables, and a function call: **array**, **array2**, **u**, and *MYcw9uffxdYPAXmUtn().K5vdDAvqBdJ()*. **array** and **array2** are dynamically generated through the multiple loops and conditional statements, but **u** on the other hand contains the data inside **binaryReader**: the resource data we dumped previously.

```
array2[15] = (byte)num3;
array2[15] = 154 - 51;
array2[15] = 143 + 11;
byte[] aIjvxlZbyG3o15igbU = array2;
MYcw9uffxdYPAXmUtn.aAgdDBUcpQV = new MYcw9uffxdYPAXmUtn(m448gdJtBGnIC5sjsy, aIjvxlZbyG3o15igbU).K5vdDAvqBdJ(u);
if (MYcw9uffxdYPAXmUtn.yUVa63uLpvJGVHA1q0() == null)
{
    goto IL_2600;
}
goto IL_111A;
IL_2477:
num = 99 + 30;
if (MYcw9uffxdYPAXmUtn.yUVa63uLpvJGVHA1q0() != null)
{
    goto IL_1835;
}
```

The function *MYcw9uffxdYPAXmUtn().K5vdDAvqBdJ()* is our first real algorithm inside the string encryption routine. The algorithm itself is seemingly custom to .NET Reactor, and uses the data inside **array** to decrypt the resource data. The decrypted data is stored as an array, at which point the integer passed in as an argument to the initial string encryption routine is used as an offset.

```
array[25] = (byte)num2;
if (89541 - 401196 != -311654)
{
    goto IL_13CC;
}
goto IL_420;
IL_B32:
byte[] u = binaryReader.ReadBytes((int)binaryReader.BaseStream.Length);
if (!MYcw9uffxdYPAXmUtn.ixZOAtj3qEPamiJbob())
{
    goto IL_1367;
}
```

Enough about theory, let's go ahead and debug the sample using dnSpy to extract the data inside **array** and **array2**, and then we can move onto looking at replicating the algorithm!

```
private byte[] K5vdDAvqBdJ(byte[] \u0020)
{
    if (\u0020.Length == 0)
    {
        return new byte[0];
    }
    int num = \u0020.Length % 4;
    int num2 = \u0020.Length / 4;
    byte[] array = new byte[\u0020.Length];
    int num3 = this.K2qdDH9707O.Length / 4;
    uint num4 = 0U;
    if (num > 0)
    {
        num2++;
    }
    for (int i = 0; i < num2; i++)
    {
        uint num5 = (uint)(i % num3);
        int num6 = i * 4;
        uint num7 = num5 * 4U;
        uint num8 = (uint)((int)this.K2qdDH9707O[(int)(num7 + 3U)] << 24 | (int)this.K2qdDH9707O[(int)(num7 + 2U)] << 16 | (int)this.K2qdDH9707O[(int)(num7 + 1U)] << 8 | (int)this.K2qdDH9707O[(int)num7]);
        if (i == num2 - 1 && num > 0)
        {
            uint num9 = 0U;
            uint num10 = 255U;
            int num11 = 0;
            for (int j = 0; j < num; j++)
            {
                if (j > 0)
                {
                    num9 <<= 8;
                }
                num9 |= (uint)\u0020[\u0020.Length - (1 + j)];
            }
            num4 += num8;
            num4 += this.mmmdDDP5Yd6(num4);
            uint num12 = num4 ^ num9;
            for (int k = 0; k < num; k++)
            {
                if (k > 0)
                {
                    num10 <<= 8;
                    num11 += 8;
                }
                array[num6 + k] = (byte)((num12 & num10) >> num11);
            }
        }
        else
        {
            num7 = (uint)num6;
            uint num9 = (uint)((int)\u0020[(int)(num7 + 3U)] << 24 | (int)\u0020[(int)(num7 + 2U)] << 16 | (int)\u0020[(int)(num7 + 1U)] << 8 | (int)\u0020[(int)num7]);
            num4 += num8;
            num4 += this.mmmdDDP5Yd6(num4);
            uint num13 = num4 ^ num9;
            array[num6] = (byte)(num13 & 255U);
            array[num6 + 1] = (byte)((num13 & 65280U) >> 8);
            array[num6 + 2] = (byte)((num13 & 16711680U) >> 16);
            array[num6 + 3] = (byte)((num13 & 4278190080U) >> 24);
        }
    }
    return array;
}
```

Doing so should be fairly simple, as we know where the decrypted data is returned, so we just need to set a breakpoint on the function *MYcw9uffxdYPAXmUtn().K5vdDAvqBdJ()*, and then dump the data inside the target variables. Buuut we cannot place a breakpoint on that address, as dnSpy cannot create one.

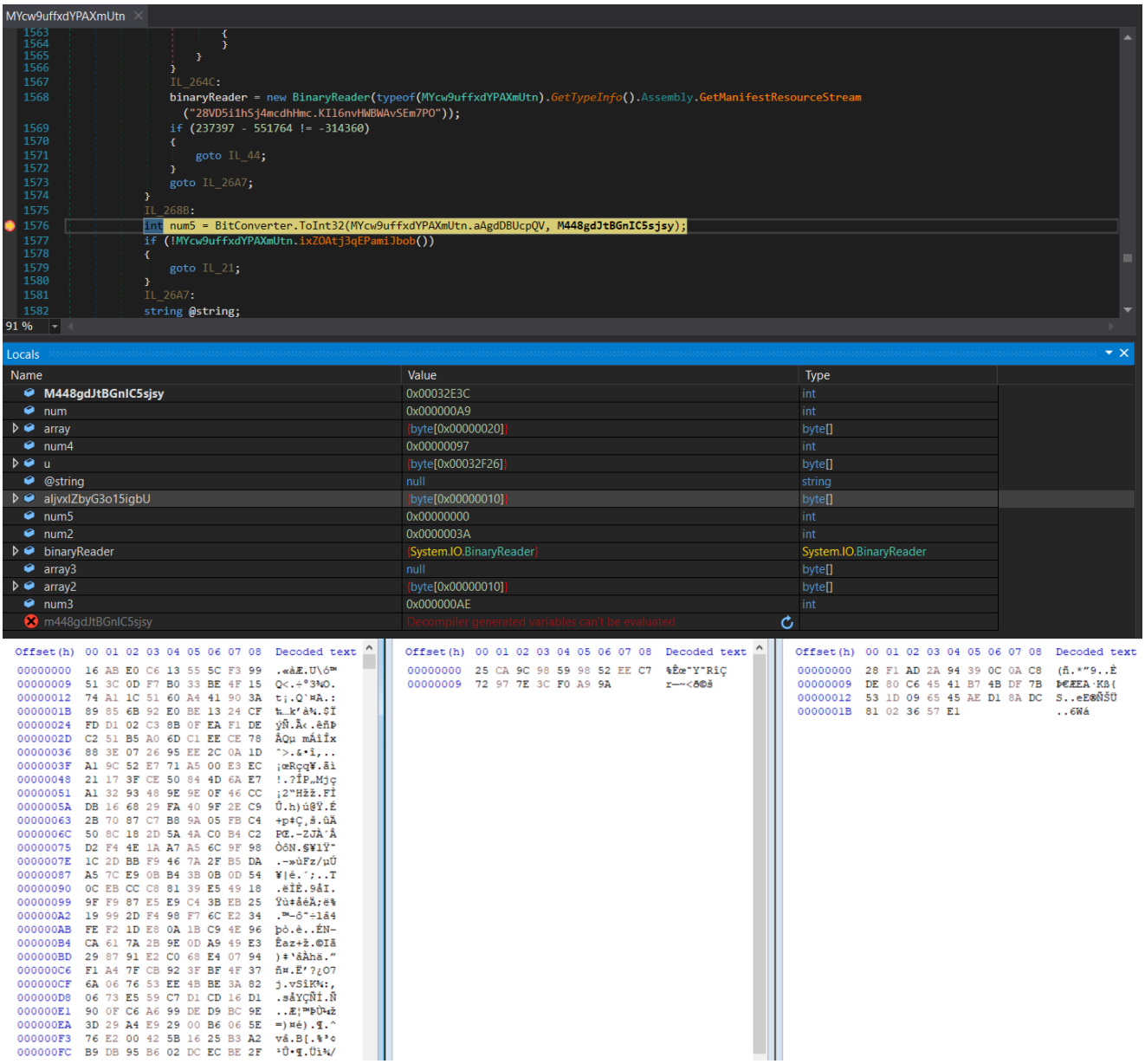
```
1480    array2[15] = 154 - 51;
1481    array2[15] = 143 + 11;
1482    byte[] aIjvxlZbyG3o15igbU = array2;
1483    MYcw9uffxdYPAXmUtn.aAgdDBUcpQV = new MYcw9uffxdYPAXmUtn(m448gdJtBGnIC5sjsy, aIjvxlZbyG3o15igbU).K5vdDAvqBdJ(u);
```

The breakpoint will not currently be hit. Could not create the breakpoint.

Location: line 1483 character 6 IL offset 0x2431 ('string MYcw9uffxdYPAXmUtn.pyM1eVFCveMv9BuGJ6(int M448gdJtBGnIC5sjsy)')

```
1488    goto IL_111A;
1489    IL_2477:
1490    num = 99 + 30;
1491    if (MYcw9uffxdYPAXmUtn.yUVa63uLpvJGVHA1q0() != null)
1492    {
1493        goto IL_1835;
1494    }
1495    array[27] = (byte)num;
1496    if (261431 - 250103 != 11332)
1497    {
1498        goto IL_C9C;
1499    }
```

Instead, we will place a breakpoint just after the **try** block has come to an end, so where the variable **num5** is initialised. Sure enough, we can now dump both target variables – **array** is 32 bytes long, and **array2** is 16 bytes long, indicating a possible key and IV setup. With all the pieces of the puzzle, we can now go ahead and attempt to replicate the decryption!



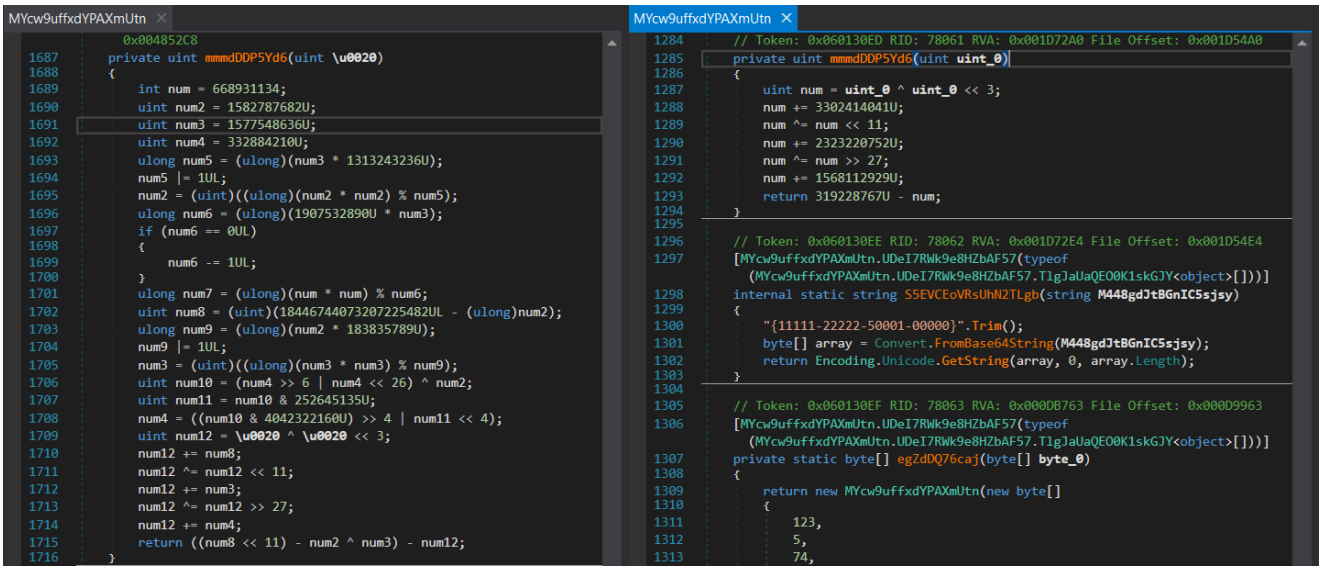
Replicating Algorithms:

We will be replicating the algorithm in Python, and luckily as dnSpy decompiles .NET binaries very well, it should be a somewhat quick process considering how closely decompiled .NET resembles Python. Before we do that, let's go ahead and run the binary through de4dot, as it will provide us a nice base to work off by removing as much obfuscation as it can.

```
de4dot v3.1.41592.3405 Copyright (C) 2011-2015 de4dot@gmail.com
Latest version and source code: https://github.com/0xd4d/de4dot

Detected Unknown Obfuscator (C:[redacted]lastconn.bin)
Cleaning C:[redacted]lastconn.bin
Renaming all obfuscated symbols
Saving C:[redacted]lastconn-cleaned.bin
ERROR: ResolutionScope is null
Ignored 1 warning/error
Use -v/-vv option or set environment variable SHOWALLMESSAGES=1 to see all messages
```

Viewing the string encryption function, it is clear that de4dot has done a great job. The biggest difference however is inside the custom algorithm, in the function *mmmdDDP5Yd6()*. In the images below, you can see the difference between the original binary (left), and the de4dot altered binary (right). This difference will make it a lot easier to replicate the algorithm.



So, let's start by implementing the *mmmdDDP5Yd6()* function. If we were to copy and paste it (and remove the **U**'s), it would execute correctly, but the returned value would be incorrect. The reason for this is Python is happy to work on a 64 bit integer, and so if we were to execute the code using **4** as the value for the **uint_0** variable, the result would be – **0x627474A8294**. We only want to deal with 32 bit values, so we will be using **& 0xFFFFFFFF** in our function a lot; specifically on every line, like shown below.

```
>>> num = 4 ^ 4 << 3
>>> num += 3302414041
>>> num ^= num << 11
>>> num += 2323220752
>>> num ^= num >> 27
>>> num += 1568112929
>>> print hex(319228767 - num)
-0x627474a8294L
```

Running the updated code, using the same value for **uint_0**, we get **0xB8B4C22C**. So now we know we can avoid dealing with 64 bit integers, we can jump back to the main algorithm, and replicate that!

```
>>> num = 4 ^ 4 << 3 & 0xFFFFFFFF
>>> num += 3302414041 & 0xFFFFFFFF
>>> num ^= num << 11 & 0xFFFFFFFF
>>> num += 2323220752 & 0xFFFFFFFF
>>> num ^= num >> 27 & 0xFFFFFFFF
>>> num += 1568112929 & 0xFFFFFFFF
>>> num = (319228767 - num) & 0xFFFFFFFF
>>> print hex(num)
0xb8b4c22cL
```

The rest of the algorithm is fairly simple to implement, however there is 1 global variable that stands out: **K2qdDH9707O**. This is assigned in a call just before the string encryption algorithm, and is in fact the data stored inside the variable **array**. Interestingly, **array2** does not seem to be used at all throughout the function, so we can ignore it from here on out.

```
1615    // Token: 0x060130EB RID: 78059 RVA: 0x00486EC0 File Offset: 0x004850C0
1616    private MYcw9uffxdYPAXmUtn(byte[] M448gdJtBGnIC5sjsy, byte[] aIjvxlZbyG3o15igbU)
1617    {
1618        this.K2qdDH97070 = M448gdJtBGnIC5sjsy;
1619        this.pgwdDnCIdQk = aIjvxlZbyG3o15igbU;
1620    }
1621
```

After converting the script from .NET to Python, we can now go ahead and test it! We already have the **array** data, and the resource data, so all we need to figure out is how the function uses the argument to locate the correct string.

```
def stringFunc(string_blob):

    array = [0] * len(string_blob)
    num = len(string_blob) % 4
    num2 = len(string_blob) / 4
    num3 = len(g_array) / 4
    num4 = 0

    if num > 0:
        num2 += 1

    for i in range(0, num2):
        num5 = (i % num3)
        num6 = i * 4
        num7 = num5 * 4
        #print num7
        num8 = (ord(g_array[num7 + 3]) << 24 | ord(g_array[num7 + 2]) << 16 | ord(g_array[num7 + 1]) << 8 | ord(g_array[num7]))
        if i == num2 - 1 and num > 0:
            num9 = 0
            num10 = 255
            num11 = 0
            for j in range(0, num):
                if j > 0:
                    num9 <<= 8
                num9 |= ord(string_blob[len(string_blob) - (1 + j)])

            num4 += num8
            num4 += someFunc(num4)
            num12 = (num4 ^ num9) & 0xFFFFFFFF

            for k in range(0, num):
                if k > 0:
                    num10 <<= 8
                    num11 += 8
                #print num6 + k
                array[num6 + k] = (num12 & num10) >> num11
                #array.append((num12 & num10) >> num11)

        else:

            num7 = num6
            num9 = ord(string_blob[num7 + 3]) << 24 | ord(string_blob[num7 + 2]) << 16 | ord(string_blob[num7 + 1]) << 8 | ord(string_blob[num7])
            num4 += num8
            num4 += someFunc(num4)
            num13 = (num4 ^ num9)

            array[num6] = (num13 & 0xFF)
            array[num6 + 1] = ((num13 & 0xFF00) >> 8)
            array[num6 + 2] = ((num13 & 0xFF0000) >> 16)
            array[num6 + 3] = ((num13 & 0xFF000000) >> 24)

    return array
```

Well, we don't need to look very hard to find it – jumping right to the end we can see a fairly simple block of code, which we covered at the beginning of the **String Decryption** chapter.

```
int num3 = BitConverter.ToInt32(MYcw9uffxdYPAXmUtn.aAgdDBUcpQV, M448gdJtBGnIC5sjsy);
if (MYcw9uffxdYPAXmUtn.ixZOAtj3qEPamiJbob())
{
    try
    {
        byte[] array3 = new byte[num3];
        if (!MYcw9uffxdYPAXmUtn.ixZOAtj3qEPamiJbob())
        {
        }
        string @string;
        for (;;)
        {
            Array.Copy(MYcw9uffxdYPAXmUtn.aAgdDBUcpQV, M448gdJtBGnIC5sjsy + 4, array3, 0, num3);
            if (MYcw9uffxdYPAXmUtn.ixZOAtj3qEPamiJbob())
            {
                @string = Encoding.Unicode.GetString(array3, 0, array3.Length);
                if (MYcw9uffxdYPAXmUtn.ixZOAtj3qEPamiJbob())
                {
                    break;
                }
            }
        }
        return @string;
    }
    catch
    {
        if (MYcw9uffxdYPAXmUtn.ixZOAtj3qEPamiJbob())
        {
        }
    }
}
return "";
```

First, **num3** is calculated by calling *ToInt32()* and passing **MYcw9uffxdYPAXmUtn.aAgdDBUcpQV** as the source data, and **M448gdJtBGnIC5sjsy** (the function argument) as the start index. Next, the variable **array3** is initialized to the size of **num3**, so we can safely assume that **num3** is a string size. **array3** is then filled with data from **MYcw9uffxdYPAXmUtn.aAgdDBUcpQV**, with the start index set to **M448gdJtBGnIC5sjsy** + 4. This means the strings will be stored as follows:

```
[4 BYTE SIZE][STRING]
```

And that is pretty much it! The string blob itself is decrypted all at once, and so the argument is only used to retrieve a specific string in the decrypted data. Putting all this together, we get the following script:

```
def main():

    offset = int(sys.argv[1])

    string_blob = open("resource_dump.bin", "rb").read()
    array = stringFunc(string_blob)

    new_array = []

    for i in range(0, len(array)):
        new_array.append(chr(array[i]))

    joined_array = "".join(new_array)

    string_length = struct.unpack("I", joined_array[offset:offset + 4])[0]
    string = joined_array[offset + 4:offset + 4 + string_length]
    print str(offset) + ": " + string.replace("\x00", "")


if __name__ == '__main__':
    main()
```

Running it with a few values we can find in the script also yields some nice results! We can also just dump all the decrypted strings to browse through, to get a good idea of what this tool is capable of doing!

```
C:[redacted]py -2 test_decrypt.py 328
328: txt_ShellCode

C:[redacted]py -2 test_decrypt.py 412
412: txt_PassRar

C:[redacted]py -2 test_decrypt.py 18430
18430: https://www.dropbox.com/oauth2/authorize

C:[redacted]py -2 test_decrypt.py 109672
109672:
############################
Rar Startup is Download

C:[redacted]py -2 test_decrypt.py 208444
208444: This assembly is protected by an unregistered version of Eziriz's ".NET Reactor"! This assembly won't further work.
```

Now, it is all good being able to decrypt strings with user input, but let's take this one step further and attempt to automate it!

Semi-Automation:

Automation is where things start to become quite complex. I don't typically focus on .NET malware, and so there's still a number of things I have to figure out – including figuring out how to have an automated string decrypter resolve strings or even comment similar to IDAPython. Currently, the automation of this string decrypter goes as far as locating all calls to the string decryption function, extracting the offset, and returning the correct string for that function. Unfortunately, this is where that stops. I have yet to successfully overwrite the IL instructions with a simple **ldstr** (like this blog post), and receive the following error:

```
Traceback (most recent call last):
  File "test_decrypt.py", line 148, in <module>
    main()
  File "test_decrypt.py", line 134, in main
    locateOffsets(joined_array)
  File "test_decrypt.py", line 116, in locateOffsets
    mod.Write("cleaned_lastconn.bin")
dnlib.DotNet.Writer.ModuleWriterException: Instruction operand is null
   at dnlib.DotNet.DummyLogger.Log(Object sender, LoggerEvent loggerEvent, String format, Object[] args)
   at dnlib.DotNet.Writer.Metadata.GetToken(Object o)
   at dnlib.DotNet.Writer.MethodBodyWriter.WriteInlineMethod(ArrayWriter& writer, Instruction instr)
   at dnlib.DotNet.Writer.MethodBodyWriterBase.WriteInstructions(ArrayWriter& writer)
   at dnlib.DotNet.Writer.MethodBodyWriter.WriteTinyHeader()
   at dnlib.DotNet.Writer.MethodBodyWriter.Write()
   at dnlib.DotNet.Writer.Metadata.WriteMethodBodies()
   at dnlib.DotNet.Writer.Metadata.Create()
   at dnlib.DotNet.Writer.ModuleWriter.WriteImpl()
   at dnlib.DotNet.Writer.ModuleWriterBase.Write(Stream dest)
   at dnlib.DotNet.Writer.ModuleWriterBase.Write(String fileName)
   at dnlib.DotNet.ModuleDef.Write(String filename)
```

If anyone has any idea what the issue is, I'd be very grateful if you could let me know! Regardless, let's have a look at how we can use Python and DNLIB to locate function calls and offset arguments in the binary!

In order to load the dnlib library, we need to make sure we have **pythonnet** installed, which can be installed using *pip install pythonnet*. Additionally, make sure you have the DNLIB DLL downloaded! With that, we need to import the Common Language Runtime (manages execution of .NET programs), as well as the System.Reflection namespace. This can be done as follows:

```
import clr
from System.Reflection import Assembly, MethodInfo, BindingFlag
from System import Type
```

Then, we need to load DNLIB using *clr.AddReference()*. This allows us to import functions from DNLIB, including the DotNet namespace. And now we're ready to start parsing .NET binaries!

```
clr.AddReference(r"dnlib")


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

The parsing code was adapted from polynomenx's blog post as listed above, and it is a brilliant example of what is possible pairing DNLIB with Python. In this case, we can search through the binary in a similar way, searching for all mentions of the method **pyM1eVFCveMv9BuGJ6**.

```
def locateOffsets(joined_array):

    filename = "lastconn.bin"
    mod = dnlib.DotNet.ModuleDefMD.Load(filename)
    opt = dnlib.DotNet.Writer.ModuleWriterOptions(mod)
    opt.MetadataOptions.Flags = opt.MetadataOptions.Flags | dnlib.DotNet.Writer.MetadataFlags.PreserveAll;
    #assembly = Assembly.LoadFrom(filename)
    totalNumTypes = 0
    stringMethodName = "pyM1eVFCveMv9BuGJ6"

    for var in mod.Types:
        if not var.HasMethods:
            pass

        for method in var.Methods:
            if not method.HasBody:
                break

            if not method.Body.HasInstructions:
                break

            i = 0
            operand = ""
            while i < len(method.Body.Instructions):

                operand = str(method.Body.Instructions[i].Operand).encode()
                if method.Body.Instructions[i].OpCode == OpCodes.Call and operand.find(str(stringMethodName).encode()) != -1:
                    keyValue = method.Body.Instructions[i - 1].GetLdcI4Value()
                    string_length = struct.unpack("I", joined_array[keyValue:keyValue + 4])[0]
                    string = joined_array[keyValue + 4:keyValue + 4 + string_length]
                    print str(keyValue) + ": " + string.replace("\x00", "")
                    #method.Body.Instructions[i - 1].OpCode = OpCodes.Nop
                    #method.Body.Instructions[i].OpCode = OpCodes.Ldstr
                    #method.Body.Instructions[i].Operand = string.replace("\x00", "")

                i += 1

    #mod.Write("cleaned_lastconn.bin")

    return
```

After executing the above script, we can view the glory that is automation! We can print all the strings, or simply pipe the output to a file to view later on – providing us with the same output that ProofPoint uploaded to their GitHub. While .NET Reactor obfuscated malware does not use the same encryption key, it's fairly simple to reverse the string encryption (at least in this version) and use the tools we covered in this post to develop a semi-automated string decrypter, speeding your analysis up by 10-fold!

```
17878: oauth2AccessToken
17992: ; Request Id: {0}
18030: clientId
18370: &force_reauthentication=true
18076: response_type=
18294: &disable_signup=true
18248: &force_reapprove=true
18108: token
18338: &require_role=
18230: &state=
18134: oauthResponseType
18172: &client_id=
18198: &redirect_uri=
18122: code
18050: redirectUri
18430: https://www.dropbox.com/oauth2/authorize
18672: state
18634: access_token
18662: uid
18686: token_type
18514: redirectedUri
18544: The supplied uri doesn't contain a fragment
18514: redirectedUri
18710: responseUri
17936: appKey
17952: appSecret
18736: The redirect uri is missing expected query arguments.
```

You can grab the full (and cleaned up) script from **here**!
It's currently optimized for Python 2.7, but with some slight alterations it should be good to go for Python 3!

After uploading the script I noticed some issues with it not picking up several calls to the string decryption function inside the main *Pro.Form1()*. It does pick up quite a few strings, though there are some obvious strings that do not appear in the dump, but are visible in the string dump on the ProofPoint ThreatResearch Github.

It could just be that I'm running the entire thing in Python instead of C#, but if anyone knows the specifics I'd love to find out!

And that wraps up this post on decrypting the strings inside the MOLERATS LastConn payload!
You may have noticed the changes to the website design, as well as cleaned up the course page – now we have finally finished working on the design and restoration, posts will be more frequent, so stay tuned! We've got quite a lot planned over the next few months!

Anyway, if you've got any feedback or questions, feel free to drop a comment down below, drop me a message over Twitter (@0verfl0w_), or via email (daniel@0ffset.net)!

Thanks for taking the time to read through the post!