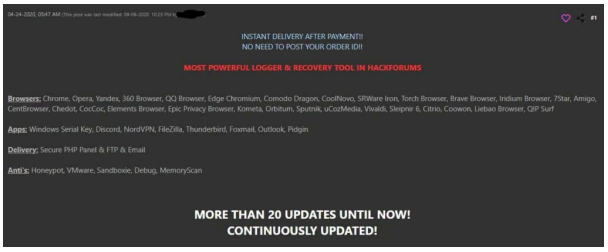
MassLogger v3: a .NET stealer with serious obfuscation

By Threat Research TeamThreat Research Team

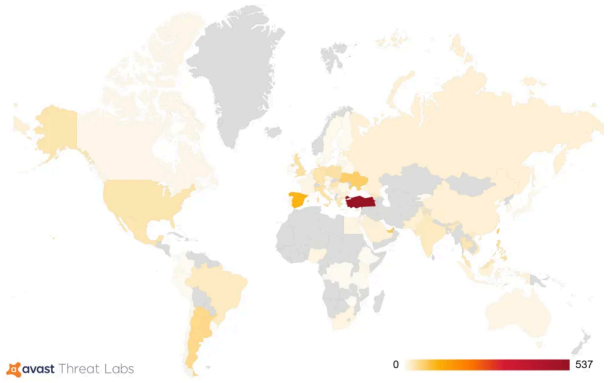
Archived: 2026-04-05 15:38:26 UTC



Forum Advertisement

MassLogger is an information stealer, first sold in hacking forums around April 2020. The malware author claims it to be the “most powerful logger and recovery tool” which costs $99 USD worth of Bitcoin for a lifetime license. MassLogger is highly configurable and gives its malicious users many options for delivery, anti-detection and anti-analysis, and capabilities such as keylogging and password stealing from a wide variety of browsers and applications.

Avast researchers have found that it is most commonly found in Turkey, Spain, Ukraine, Chile, the United States, Brazil, the United Kingdom, Germany and Poland. Avast AV is detecting this malware under “**MSIL:MassLogger-***”. In addition, the latest variant of MassLogger will not run if it finds Avast or AVG AV present in the system.



Map illustrating the countries MassLogger has targeted from October 2020 to February 2021

Advertised Features

The malware author has an active Github directory where he shares the source code of multiple malware features and packers for educational purposes. We are able to find many similarities between what is being used in the malware and some of his Github projects.

In August 2020, FireEye wrote an article explaining how to get past the anti-analysis tricks used by MassLogger version 1.3. Recently Talos wrote on MassLogger version 3. In their post, they focus on the campaign and delivery of the malware. In this blog, we will provide the last missing piece, a detailed analysis of the final payload's obfuscation which includes operation codes and an interpreter as well as indirect calls to unassigned fields.

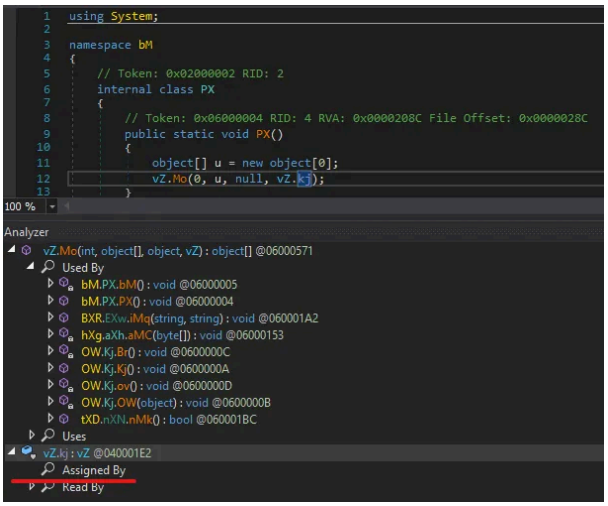
Analysis

Our analysis is demonstrated with sample

SHA256: 2487B12F52B803F5D38B3BB9388B039BF4F58C4B5D192D50DA5FA047E9DB828B

Populate fields with methods at run-time

Scanning through the decompiled code, the majority of function calls are performed in an indirect manner where it tries to call uninitialized field values. As a result, the control flow of the malware are totally hidden from static analysis

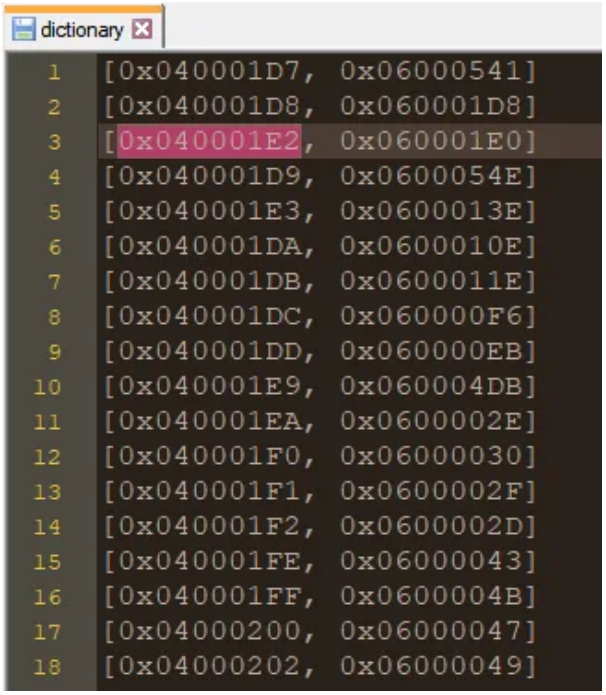


PE Entry Point

Indirect call definition

Mo() is a wrapper function whose purpose is to call the 4th argument, in this case, **vZ.kj**. Interestingly, **vZ.kj** is of field type instead of method type, and there is no trace of it being assigned. Revisiting the **vZ** declaration, we find out that it is just one of the many internal sealed classes whose structure consists of field values with no assigned references, a caller function similar to **Mo()**, and an external “Invoke” function. In addition, they all call a function with token 0x060004D8 in the module initialization phase.

0x060004D8 is in charge of decrypting the 2KB embedded resource to build a dictionary between field tokens and method tokens. The field **vZ.kj** mentioned above has token 0x040001E2 which is mapped with method token 0x060001E0



Field-Method Dictionary

Once the dictionary is constructed, the function looks through all the fields with Static, NonPublic, or GetField flag in the module to find the corresponding method tokens. If the token belongs to a static method, it will be assigned directly to the field using **fieldInfo.SetValue()**

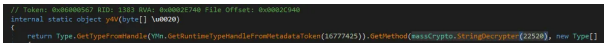
If the specified method is not declared as static, a wrapper for the intended method is constructed then assigned to the field. This dynamically created method has an additional parameter of type **System.Drawing.Imaging.ImageCodecInfo**. The call to the intended function will be made through **OpCodes.Callvirt** or **OpCodes.Call** based on whether the first byte of the token is modified or not. For example, if the token is 0x46000361 in the dictionary, it will be converted back to the standard token 0x06000361, and **OpCodes.Callvirt** will be used instead of **OpCodes.Call**.

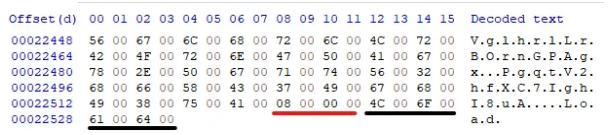
Assigning dynamic method to field

These dynamic wrapper methods may cause additional overheads when debugging due to the transfer to **DynamicResolver.GetCodeInfo()** method before the intended function is reached.

String Decryption

All strings used in the malware are encrypted and stored in the 23KB embedded resource. The method token 0x060004DB acts as the string provider where it decrypts and stores the string table upon its first run. This method receives the offset of the required string, the first DWORD is read to determine the string length, then the string following after is returned.



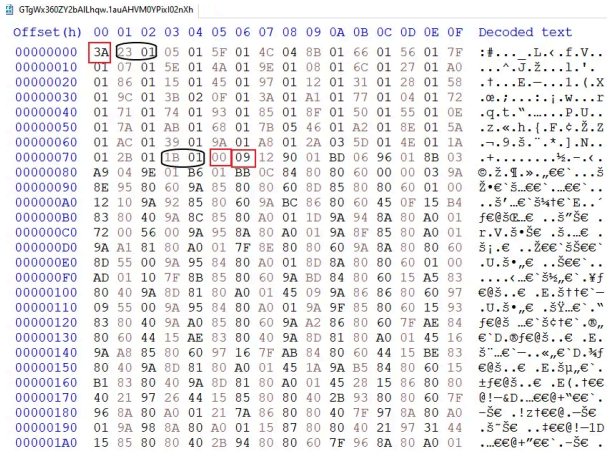


Offset(d)	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	Decoded text
00022448	56	00	67	00	6C	00	68	00	72	00	6C	00	4C	00	72	00	V.g.l.h.r.l.L.r.
00022464	42	00	4F	00	72	00	6E	00	47	00	50	00	41	00	67	00	B.O.r.n.G.P.A.g.
00022480	78	00	2E	00	50	00	67	00	71	00	74	00	56	00	32	00	x...P.g.q.t.V.2.
00022496	68	00	66	00	58	00	43	00	37	00	49	00	67	00	68	00	h.f.X.C.7.I.g.h.
00022512	49	00	38	00	75	00	41	00	08	00	00	00	4C	00	6F	00	I.8.u.A.....L.o.
00022528	61	00	64	00													a.d.

Example of how MassLogger decrypts its strings

Retrieving Operation Codes from Resource

After the string decryption, the malware leads us to function 0x060001DF where the malware reveals its secretive flow control in the form of operation codes. First, an array of objects are deserialized from the 3rd 3KB embedded resource. These objects contain a list of operation codes and additional data that will be fed into an interpreter to perform tasks such as invoking a function, creating a new object, or modifying a List.



GTgWx360ZY2bAlLhqw.1auAHVM0YPixI02nXh

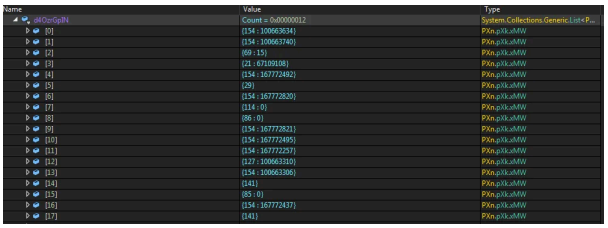
Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	3A	23	01	05	01	5F	01	4C	04	8B	01	66	01	56	01	7F	:#..._.L.‹.f.V..
00000010	01	07	01	5E	01	4A	01	9E	01	08	01	6C	01	27	01	A0	...^.J.ž...l.'.
00000020	01	86	01	15	01	45	01	97	01	12	01	31	01	28	01	58	.†...E.—...1.(.X
00000030	01	9C	01	3B	02	0F	01	3A	01	A1	01	77	01	04	01	72	.œ.;...:.¡.w...r
00000040	01	71	01	74	01	93	01	85	01	8F	01	50	01	55	01	0E	.q.t.“.…...P.U..
00000050	01	7A	01	AB	01	68	01	7B	05	46	01	A2	01	8E	01	5A	.z.«.h.{.F.¢.Ž.Z
00000060	01	AC	01	39	01	9A	01	A8	01	2A	03	5D	01	4E	01	1A	.¬.9.š.¨.*.].N..
00000070	01	2B	01	1B	01	00	09	12	90	01	BD	06	96	01	8B	03	.+.....½.—.‹.
00000080	A9	04	9E	01	B6	01	BB	0C	84	80	80	60	00	00	03	9A	©.ž.¶.».„€€`...š
00000090	8E	95	80	60	9A	85	80	80	60	8D	85	80	80	60	01	00	Ž•€`š…€€`..€€`..
000000A0	12	10	9A	92	85	80	60	9A	BC	86	80	60	45	0F	15	B4	..š'…€`š¼†€`E..´
000000B0	83	80	40	9A	8C	85	80	A0	01	1D	9A	94	8A	80	A0	01	ƒ€@šŒ…€ ..š”Š€ .
000000C0	72	00	56	00	9A	95	8A	80	A0	01	9A	8F	85	80	A0	01	r.V.š•Š€ .š.…€ .
000000D0	9A	A1	81	80	A0	01	7F	8E	80	80	60	9A	8A	80	80	60	š¡.€ ..Ž€€`šŠ€€`
000000E0	8D	55	00	9A	95	84	80	A0	01	8D	8A	80	80	60	01	00	.U.š•„€ ..Š€€`..
000000F0	AD	01	10	7F	8B	85	80	60	9A	BD	84	80	60	15	A5	83	....‹…€`š½„€`.¥ƒ
00000100	80	40	9A	8D	81	80	A0	01	45	09	9A	86	86	80	60	97	€@š..€ .E.š††€`—
00000110	09	55	00	9A	95	84	80	A0	01	9A	9F	85	80	60	15	93	.U.š•„€ .šŸ…€`.“
00000120	83	80	40	9A	A0	85	80	60	9A	A2	86	80	60	7F	AE	84	ƒ€@š …€`š¢†€`.®„
00000130	80	60	44	15	AE	83	80	40	9A	8D	81	80	A0	01	45	16	€`D.®ƒ€@š..€ .E.
00000140	9A	A8	85	80	60	97	16	7F	AB	84	80	60	44	15	BE	83	š¨…€`—..«„€`D.¾ƒ
00000150	80	40	9A	8D	81	80	A0	01	45	1A	9A	B5	84	80	60	15	€@š..€ .E.šµ„€`.
00000160	B1	83	80	40	9A	8D	81	80	A0	01	45	28	15	86	80	80	±ƒ€@š..€ .E(.†€€
00000170	40	21	97	26	44	15	85	80	80	40	2B	93	80	80	60	7F	@!—&D.…€€@+“€€`.
00000180	96	8A	80	A0	01	21	7A	86	80	80	40	7F	97	8A	80	A0	–Š€ .!z†€€@.—Š€
00000190	01	9A	98	8A	80	A0	01	15	87	80	80	40	21	97	31	44	.š˜Š€ ..‡€€@!—1D
000001A0	15	85	80	80	40	2B	94	80	80	60	7F	96	8A	80	A0	01	...€€@+”€€`.–Š€ .

Embedded resource contains reading instructions and operation codes

In order for the object array to be deserialized, the malware first starts with initializing the following structures:

- An array of 255 bytes. The first byte in the resource indicates the next number of words used to assign values to this array. The first byte in each word, ie 0x23 or 0x1B in the capsules, represent the index; while the second byte, ie 0x1 in the capsules, represents what read operations to use:
  - 0x1 = Custom Binary Reader
  - 0x2 = ReadInt64
  - 0x3 = ReadSingle
  - 0x4 = ReadDouble
  - 0x5 = Read an array of data
- List of strings. The following byte, 0x0 in the square, determines the size of the list. In this case, no string will be needed
- An array of objects to be deserialized. The next byte, 0x9 in the square, tells us the size of the array. Each member of the array will be initialized to null and reconstructed on the later step.
- An array of offsets. This array has the same size as the object array and represents where the data associated with each object locates in the resource. Each entry of this array will be filled in using CustomBinaryReader starting at the position after the 0x9 byte.

When the above structures are in place, a lengthy routine that resides in 0x060001DF will start reconstructing a specified object from the array by reading the proper resource data. The main purpose of this object is to form a list of operation codes and the needed parameters to perform them.



Operation codes stored inside deserialized object

The first part is the operation code which ranges from 1 to 173, while the second part represents the operand. The interpreter for these operations locates inside method 0x06000499 and consists of 157 unique handlers. This massive implementation is indicative of a commercial code protection tool, but we aren't unable to find any further information at the moment.

```
        num2 = (int)this.operand;
        module = Type.GetTypeFromHandle(YMn.GetRuntimeTypeHandleFromMetadataToken(33554511)).Module;
        type = module.ResolveType(num2);
        mMM = VM.ControlPanel.Tvc(hLk.Mo(this.XZQRSNNRi5, hLk.JZj));
        array = VM.ControlPanel.wKE(type, mMM.e3d().t2dsXmCQLS.US1sQc9Mec);
        lLb.Mo(this.XZQRSNNRi5, new VM.VXz(array), lLb.YZC);
        return;
    case (VM.uMH)149:
        throw (Exception)LL4.Mo(hLk.Mo(this.XZQRSNNRi5, hLk.JZj), null, LL4.rLC);
    case (VM.uMH)150:
        mMM = VM.ControlPanel.Tvc(hLk.Mo(this.XZQRSNNRi5, hLk.JZj));
        mMM2 = VM.ControlPanel.vKq(hLk.Mo(this.XZQRSNNRi5, hLk.JZj));
        if (mMM == null || mMM2 == null)
        {
            throw new VM.dMh();
        }
        num = 10;
        break;
    case (VM.uMH)151:
        cXZ = hLk.Mo(this.XZQRSNNRi5, hLk.JZj);
        flag = r4z.Mo(cXZ, r4z.jiQ);
        if (flag)
        {
            this.Index = (int)this.operand - 1;
        }
        return;
    case (VM.uMH)152:
        cXZ = hLk.Mo(this.XZQRSNNRi5, hLk.JZj);
        mMM = VM.ControlPanel.Tvc(cXZ);
        if (cXZ != null && r4z.Mo(cXZ, r4z.kll) && mMM != null)
        {
            num = 55;
        }
        else
        {
            if (mMM != null && r4z.Mo(mMM, r4z.MLR))
            {
                intPtr = YZl.Mo((VM.SMy)mMM, YZl.JZk);
                lLb.Mo(this.XZQRSNNRi5, new VM.uMK((int)(*(ushort*)((void*)intPtr)), (VM.TypeEnum)4), lLb.YZC);
                return;
            }
            throw new VM.dMh();
        }
        break;
    case (VM.uMH)153:
        mMM = VM.ControlPanel.Tvc(hLk.Mo(this.XZQRSNNRi5, hLk.JZj));
        if (mMM != null)
        {
            lLb.Mo(this.XZQRSNNRi5, mMM.T9G(), lLb.YZC);
            return;
        }
        throw new VM.dMh();
    case (VM.uMH)154:
        wZb.Mo(this, true, wZb.MZM);
        return;
    case (VM.uMH)155:
        mMM = VM.ControlPanel.Tvc(hLk.Mo(this.XZQRSNNRi5, hLk.JZj));
        num3 = 54;
```

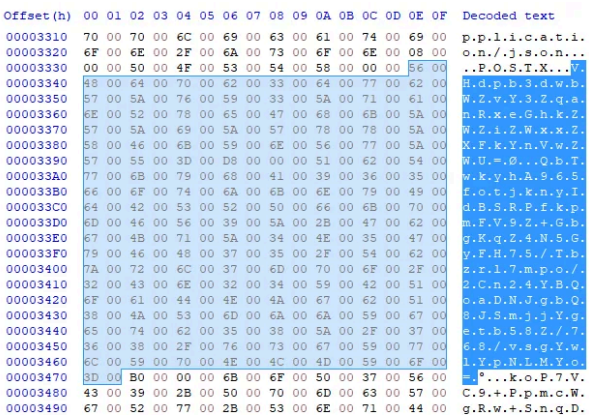
Operation code Interpreter

The meaning of important operation codes from the above example:

- 154 : <method token>: invoke the specified method after reconstructing the needed arguments and the receiver of the returned value.
- 127 : <constructor method token>: create a new object after reconstructing the needed arguments for the constructor and where the new object will be assigned to
- 21 : <field Token>: get the value of the specified field, usually an encrypted config which is used by operation "166 : 0x60001C3" for decryption

Config Decryption

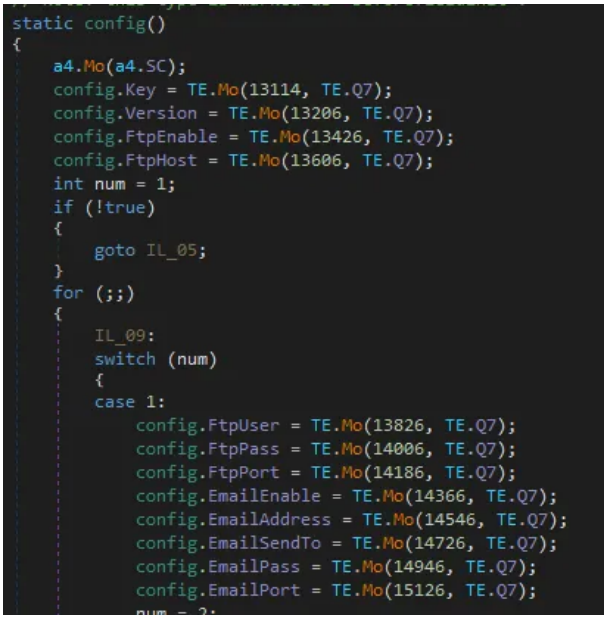
A block of Base64 encoded strings can be found in the middle of the decrypted string table:



Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00003310	70	00	70	00	6C	00	69	00	63	00	61	00	74	00	69	00	p.p.l.i.c.a.t.i.
00003320	6F	00	6E	00	2F	00	6A	00	73	00	6F	00	6E	00	08	00	o.n./.j.s.o.n...
00003330	00	00	50	00	4F	00	53	00	54	00	58	00	00	00	56	00	..P.O.S.T.X...V.
00003340	48	00	64	00	70	00	62	00	33	00	64	00	77	00	62	00	H.d.p.b.3.d.w.b.
00003350	57	00	5A	00	76	00	59	00	33	00	5A	00	71	00	61	00	W.Z.v.Y.3.Z.q.a.
00003360	6E	00	52	00	78	00	65	00	47	00	68	00	6B	00	5A	00	n.R.x.e.G.h.k.Z.
00003370	57	00	5A	00	69	00	5A	00	57	00	78	00	78	00	5A	00	W.Z.i.Z.W.x.x.Z.
00003380	58	00	46	00	6B	00	59	00	6E	00	56	00	77	00	5A	00	X.F.k.Y.n.V.w.Z.
00003390	57	00	55	00	3D	00	D8	00	00	00	51	00	62	00	54	00	W.U.=.Ø...Q.b.T.
000033A0	77	00	6B	00	79	00	68	00	41	00	39	00	36	00	35	00	w.k.y.h.A.9.6.5.
000033B0	66	00	6F	00	74	00	6A	00	6B	00	6E	00	79	00	49	00	f.o.t.j.k.n.y.I.
000033C0	64	00	42	00	53	00	52	00	50	00	66	00	6B	00	70	00	d.B.S.R.P.f.k.p.
000033D0	6D	00	46	00	56	00	39	00	5A	00	2B	00	47	00	62	00	m.F.V.9.Z.+.G.b.
000033E0	67	00	4B	00	71	00	5A	00	34	00	4E	00	35	00	47	00	g.K.q.Z.4.N.5.G.
000033F0	79	00	46	00	48	00	37	00	35	00	2F	00	54	00	62	00	y.F.H.7.5./.T.b.
00003400	7A	00	72	00	6C	00	37	00	6D	00	70	00	6F	00	2F	00	z.r.l.7.m.p.o./.
00003410	32	00	43	00	6E	00	32	00	34	00	59	00	42	00	51	00	2.C.n.2.4.Y.B.Q.
00003420	6F	00	61	00	44	00	4E	00	4A	00	67	00	62	00	51	00	o.a.D.N.J.g.b.Q.
00003430	38	00	4A	00	53	00	6D	00	6A	00	6A	00	59	00	67	00	8.J.S.m.j.j.Y.g.
00003440	65	00	74	00	62	00	35	00	38	00	5A	00	2F	00	37	00	e.t.b.5.8.Z./.7.
00003450	36	00	38	00	2F	00	76	00	73	00	67	00	59	00	77	00	6.8./.v.s.g.Y.w.
00003460	6C	00	59	00	70	00	4E	00	4C	00	4D	00	59	00	6F	00	l.Y.p.N.L.M.Y.o.
00003470	3D	00	B0	00	00	00	6B	00	6F	00	50	00	37	00	56	00	=.°...k.o.P.7.V.
00003480	43	00	39	00	2B	00	50	00	70	00	6D	00	63	00	57	00	C.9.+.P.p.m.c.W.
00003490	67	00	52	00	77	00	2B	00	53	00	6E	00	71	00	44	00	g.R.w.+.S.n.q.D.

Decrypted String Table

These are MassLogger encrypted config. First, each encrypted config is assigned to the corresponding field in Module 0x02000044. Note that the module token is consistent across all MassLogger v3 samples that we looked at.



```
static config()
{
    a4.Mo(a4.SC);
    config.Key = TE.Mo(13114, TE.Q7);
    config.Version = TE.Mo(13206, TE.Q7);
    config.FtpEnable = TE.Mo(13426, TE.Q7);
    config.FtpHost = TE.Mo(13606, TE.Q7);
    int num = 1;
    if (!true)
    {
        goto IL_05;
    }
    for (;;)
    {
        IL_09:
        switch (num)
        {
        case 1:
            config.FtpUser = TE.Mo(13826, TE.Q7);
            config.FtpPass = TE.Mo(14006, TE.Q7);
            config.FtpPort = TE.Mo(14186, TE.Q7);
            config.EmailEnable = TE.Mo(14366, TE.Q7);
            config.EmailAddress = TE.Mo(14546, TE.Q7);
            config.EmailSendTo = TE.Mo(14726, TE.Q7);
            config.EmailPass = TE.Mo(14946, TE.Q7);
            config.EmailPort = TE.Mo(15126, TE.Q7);
            num = 2;
```

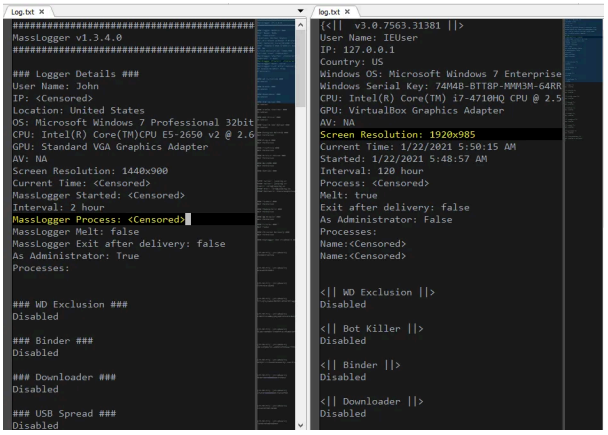
Next, the **config.Key** is base64 decoded and used as the PBKDF2 password to derive a 32bytes **_key** (decryption) and a 64 bytes **_authKey** (encryption). When the malware is ready to read the config for usage, function 0x060001C4 from module 0x02000045 decrypts each config field with the following steps:

- Base64 decoded
- The first 32 bytes are SHA256 checksum which is used to verify the integrity of the string
- The next 16 bytes are used as IV
- The config is decrypted using AES with _key and IV from the previous step

The full config of this sample: https://github.com/avast/ioc/blob/master/MassLogger/config.txt

Functionality

Despite the variation of obfuscation technique in each version, MassLogger makes little change in its functionality. Compared to the analysis in June 2020, a check for Avast and AVG AV (looking for AvastGUI and AVGUI processes) is added at the beginning of execution. In addition, the malware has minimized the amount of fingerprints left on the system. The log data is no longer write to disk, and the “MassLogger” keyword has been removed:



Comparison between version 1.3 and version 3.0 log data

Conclusion

MassLogger is a versatile .NET information stealer with a complete list of features. The malware employs heavy obfuscation techniques which we intend to describe in this article. At the moment, we can’t confirm whether the malware is packed with a commercial crypter, but its complexity may indicate so. We also illustrate how the configuration can be extracted which can help with identifying IOCs for a particular sample.

Indicators of Compromise

The full list of IoCs is available at:

https://github.com/avast/ioc/tree/master/MassLogger

A group of elite researchers who like to stay under the radar.

Source: https://decoded.avast.io/anhho/masslogger-v3-a-net-stealer-with-serious-obfuscation/