Shadows with a chance of BlackNix

medium.com/insomniacs/shadows-with-a-chance-of-blacknix-badc0f2f41cb

asuna amawaka May 6, 2020

asuna amawaka

May 6, 2020

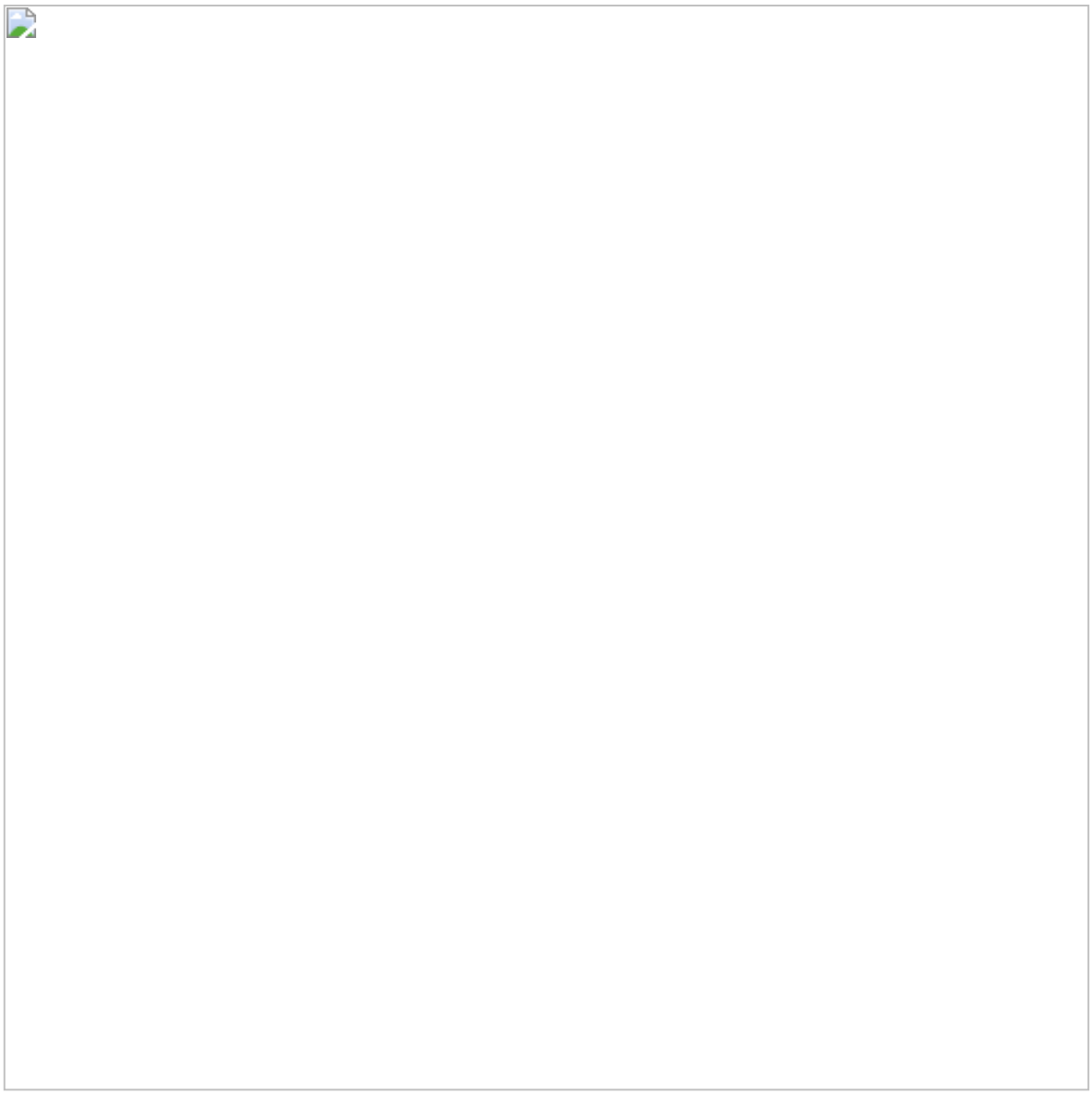
.

13 min read

In the last post, I did an analysis of a set of *BBSRAT* samples that are characterized by unique mutexes (cc5d64b344700e403e2sse, cc5d6b4700e403e2sse, cc5d6b4700032eSS) and calls back to a known **Winnti Group** C2 (bot[.]googlerenewals[.]net). In this post, I'm going to continue on analysis of samples related to the abovementioned mutexes.

When I started on this analysis journey, I was hoping to find more *BBSRAT* samples. However, the results I arrived at deviated from expectations, and instead I found a set of dropper malware that used the same mutexes as those found in the *BBSRAT* samples I analyzed. The final payload dropped by these droppers is the *BlackNix RAT*. Pivoting from the C2 called from this *BlackNix RAT*, more *BlackNix RAT*s were found on VirusTotal. I was unable to find any technical blogs on the *BlackNix RAT*, and hence, here I am.

The following diagram is a sort of a "signpost" for this writing.

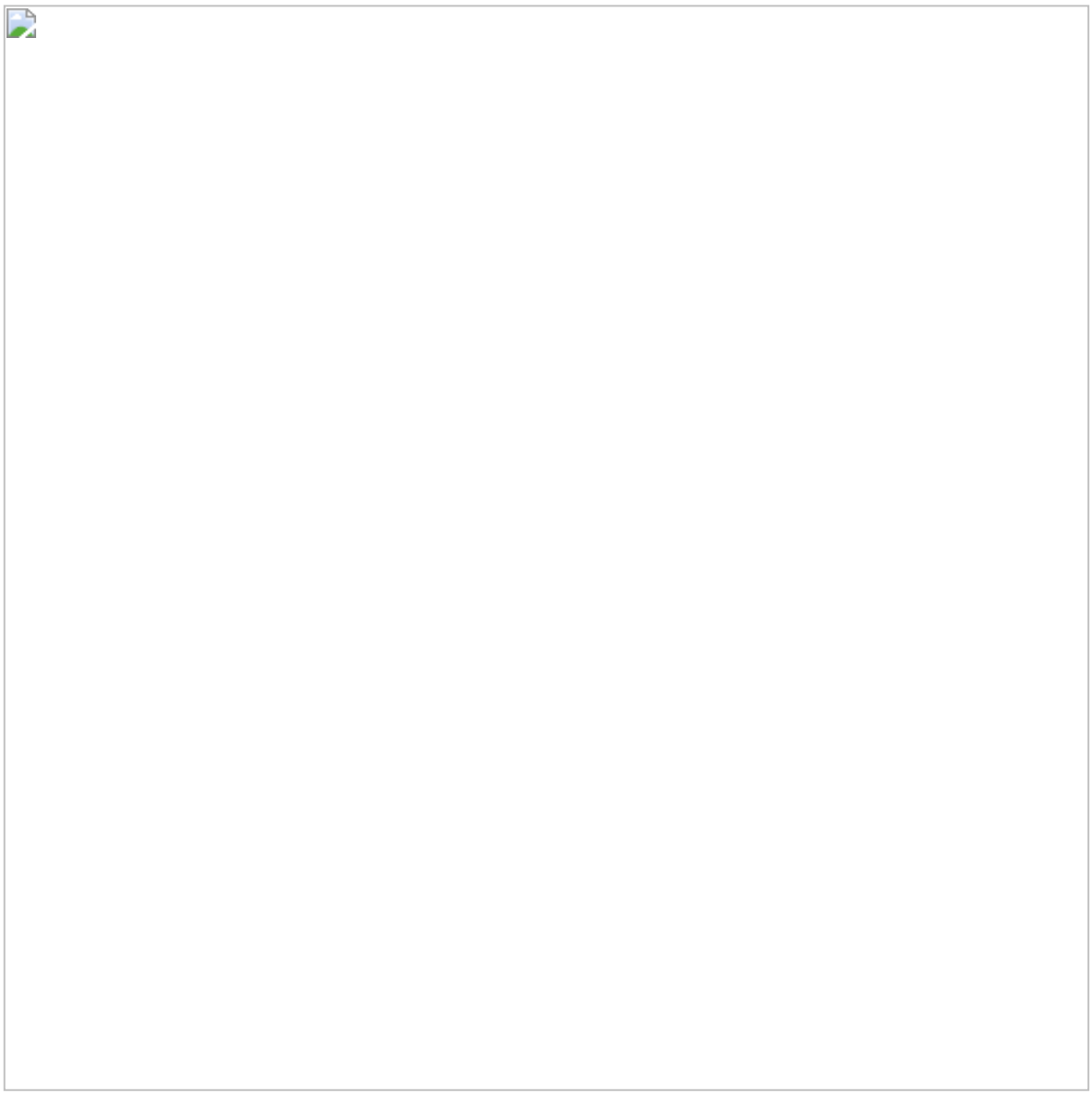


Let’s dip into the first dropper!

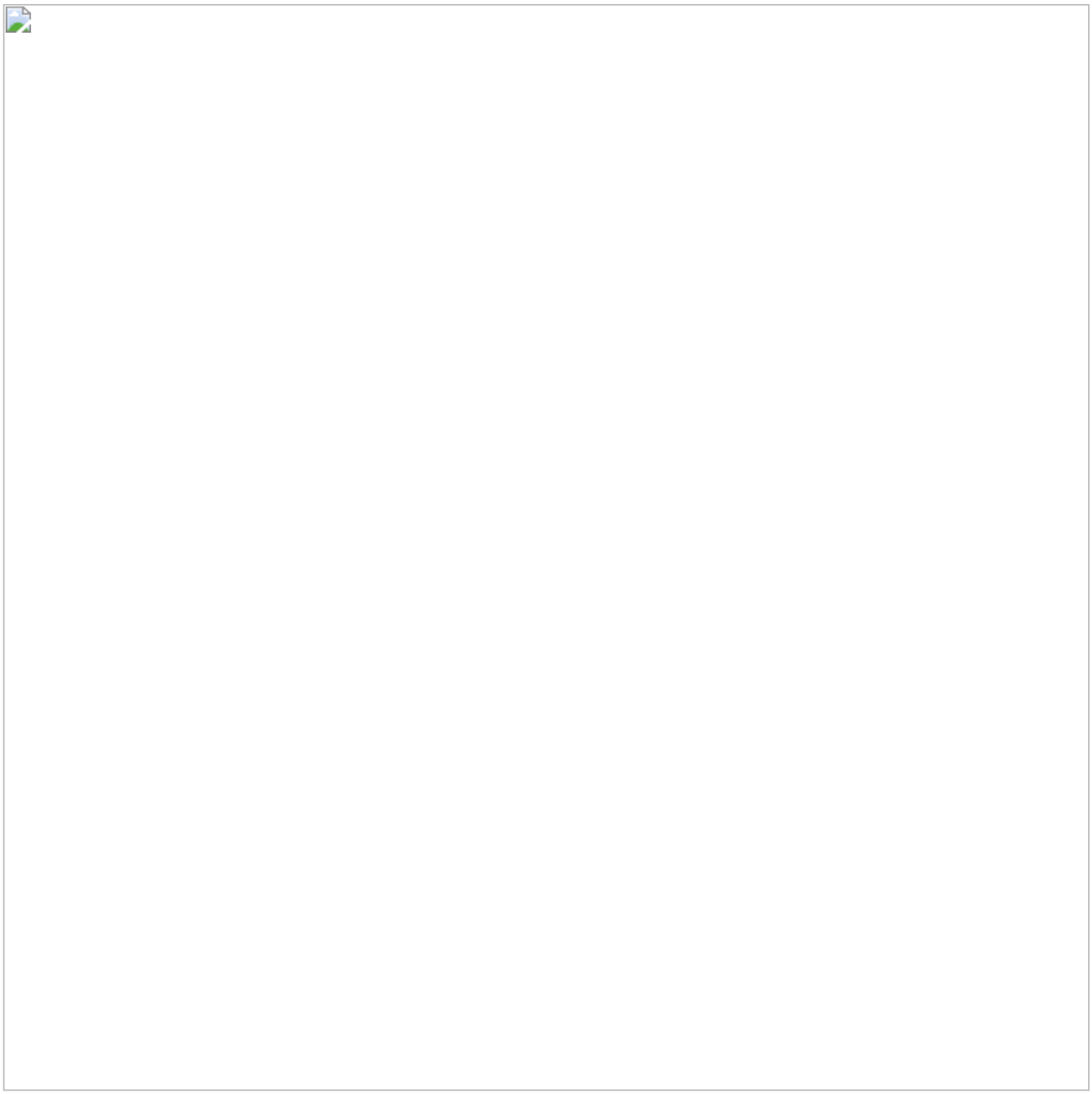
Project1.exe

The following files are likely from the same source code:
daaa061c88b197fa92d9648306e79875e3a24f392550dacaabd22e5fdba53ebf
75dc821013fe92ef93cefa47d3fe83ad5ce90658e8ef01fcdb0b11652397abec

Judging from the executables' icon, it looks like the samples are written with Borland Delphi 7, and sadly the executables' compilation timestamps are 1992–06–19 22:22:17 (a well-known bug in Delphi 4–2006). I didn't really look into which versions are affected by the compilation time bug, because that's beside the point. The samples' compilation time can still be deduced with the timestamp within the executables' resources. (Thanks to Adam's old post on this[1])



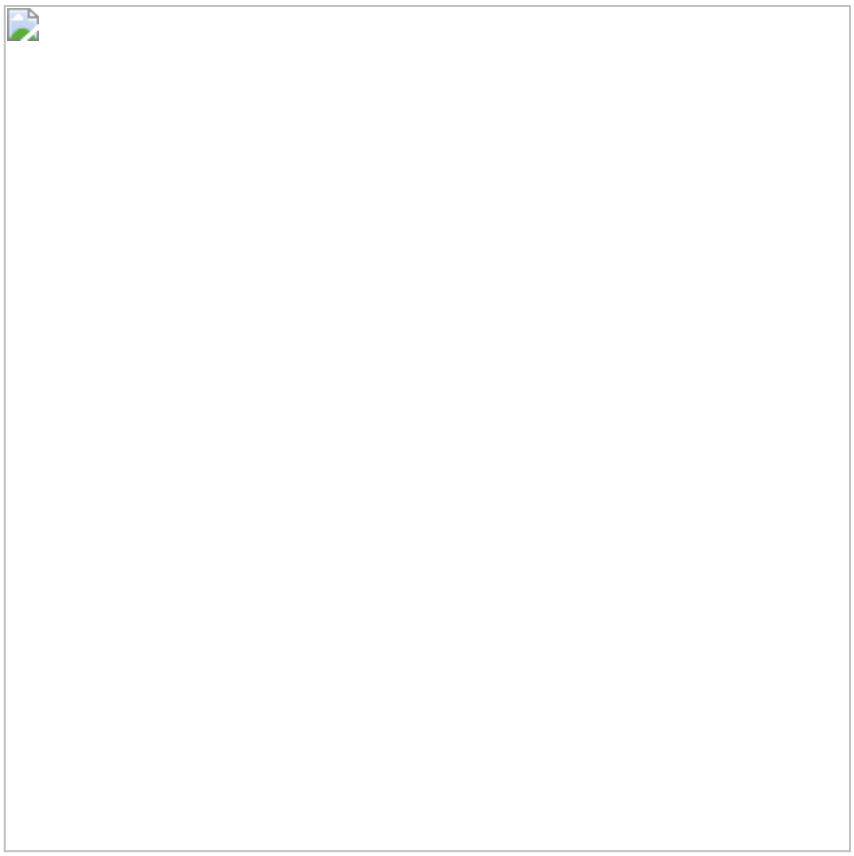
This dropper will drop 2 files system.exe and systemm.exe into %USERPROFILE%\Pictures, execute them and write a diskshadow.exe to C:\ProgramData\Microsoft\DeviceSync\.



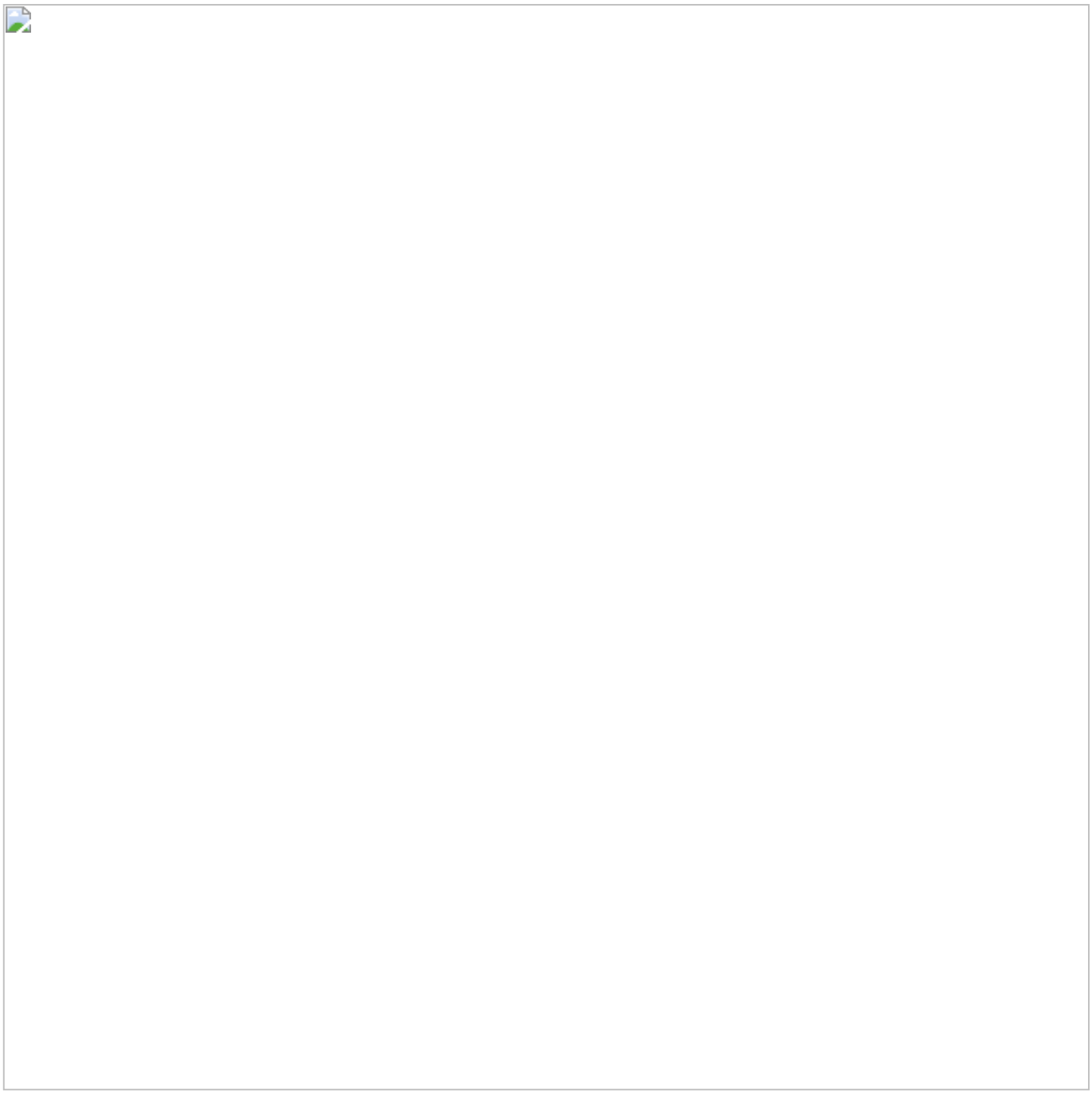
system.exe, systemm.exe
SHA256:
AEB61477C3F4F2D76AF0DC97B19B01F73C8ADA1FCE91D66E8B0E489E2E807430

Execution of system.exe creates a service to execute itself within the context of the service.

> C:\Windows\System32\sc.exe create SESSRV binpath= “cmd /c \”C:\Users\asuna\Pictures\system.exe\””
>
> C:\Windows\System32\net.exe start SESSRV



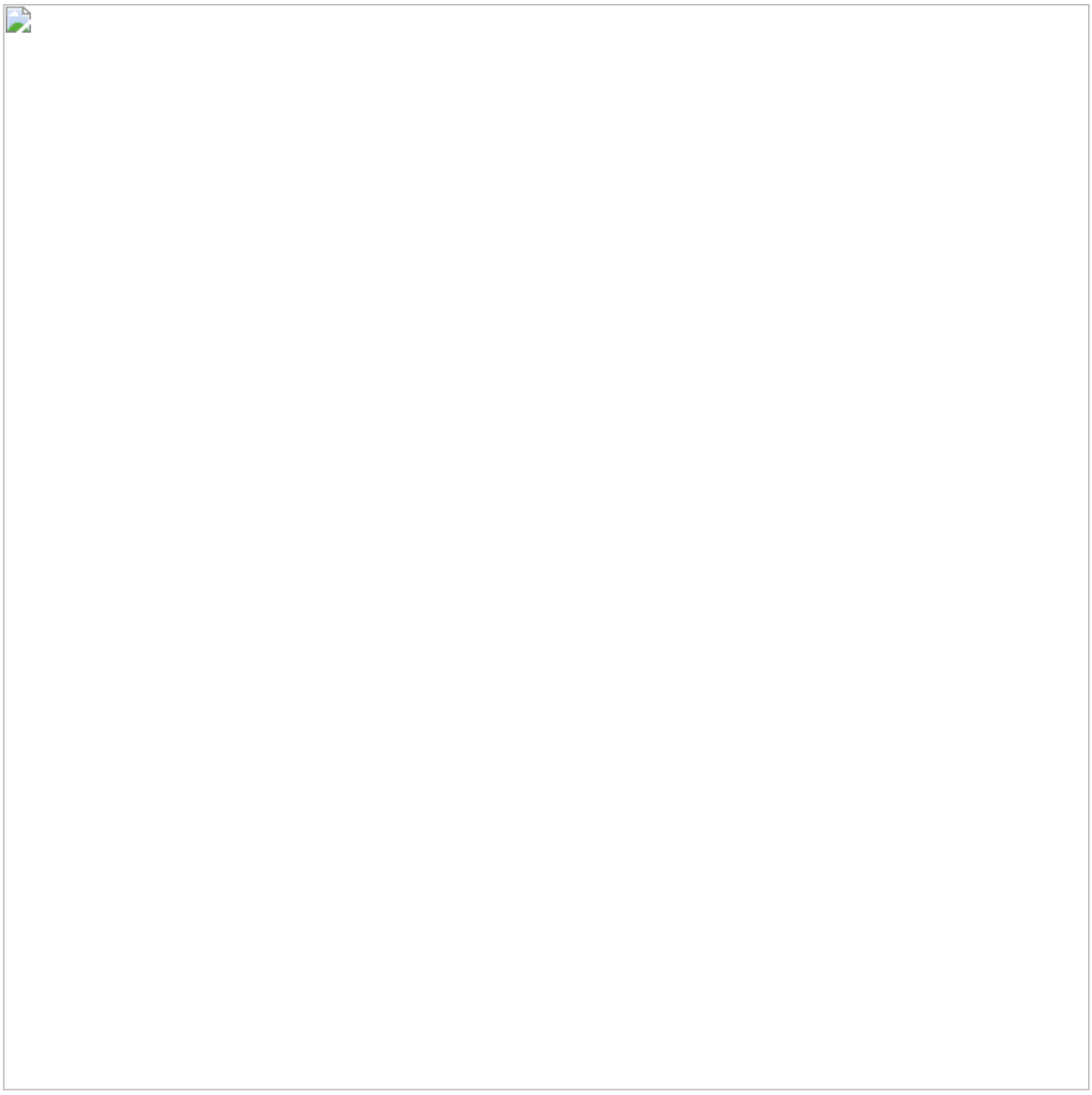
The mutexes are set within the execution of system.exe and systemm.exe.



systemm.exe attempts to copy diskshadow.exe to a network location \\TSCLIENT\%userprofile%\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup\. The sample also attempts to locate and copy a 1.jpg.lnk within the same directory as diskshadow.exe, but this file was not created by the previous dropper.

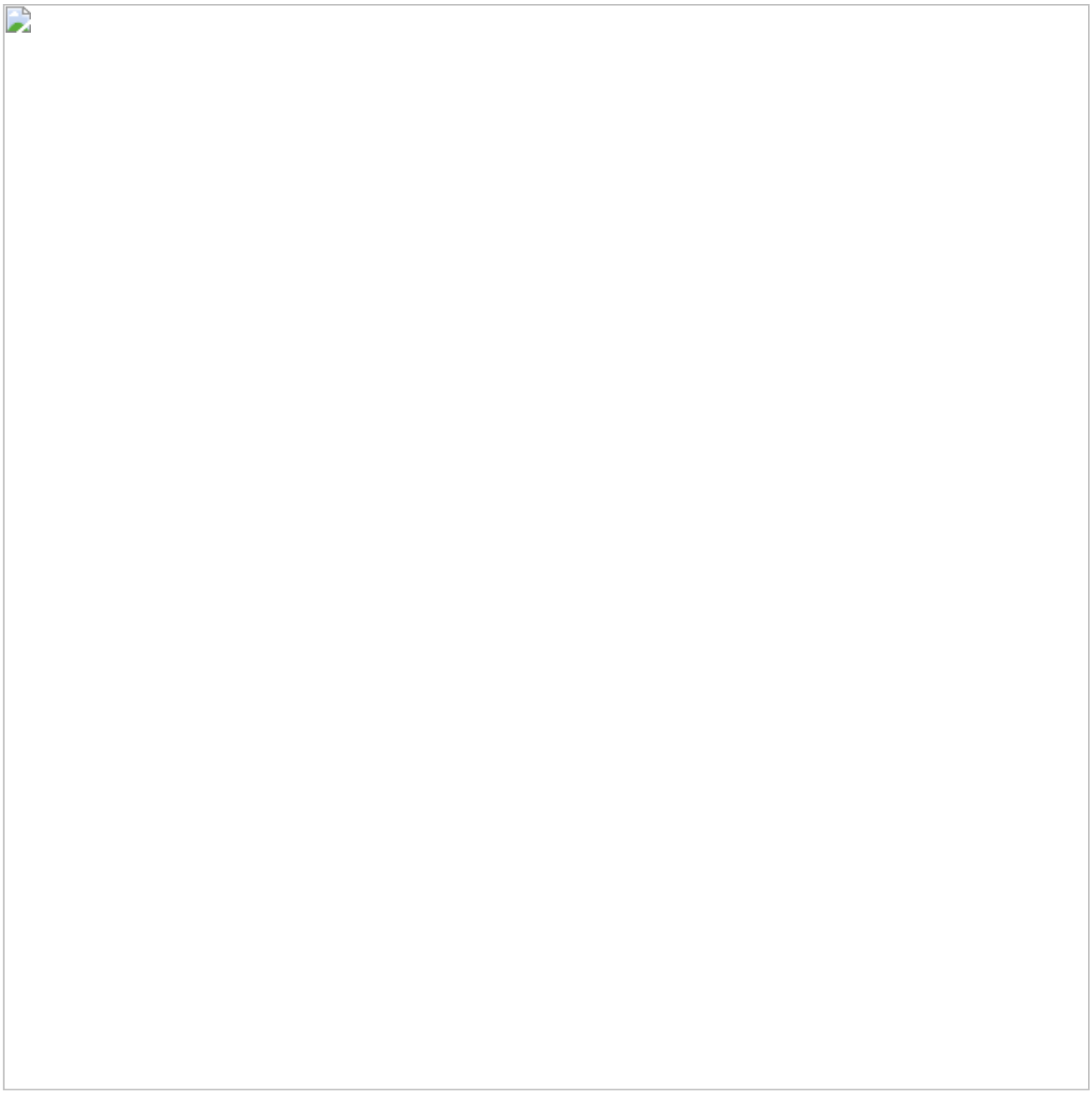
In gist, the sample's job is to copy the payload diskshadow.exe to a host that is connected to the current victim via RDP and set persistency to run at startup. This is possibly a tool meant for lateral movement within the victim network. The same technique is also found in the execution of the *BBSRAT* samples analyzed previously.

The following screen captures referenced one of the *BBSRAT* trinity files, lockdown.dll (MD5: 166D28FF69019D9991EECBD26DC1E266):



Copy file to network location. Left: system.exe; Right: lockdown.dll

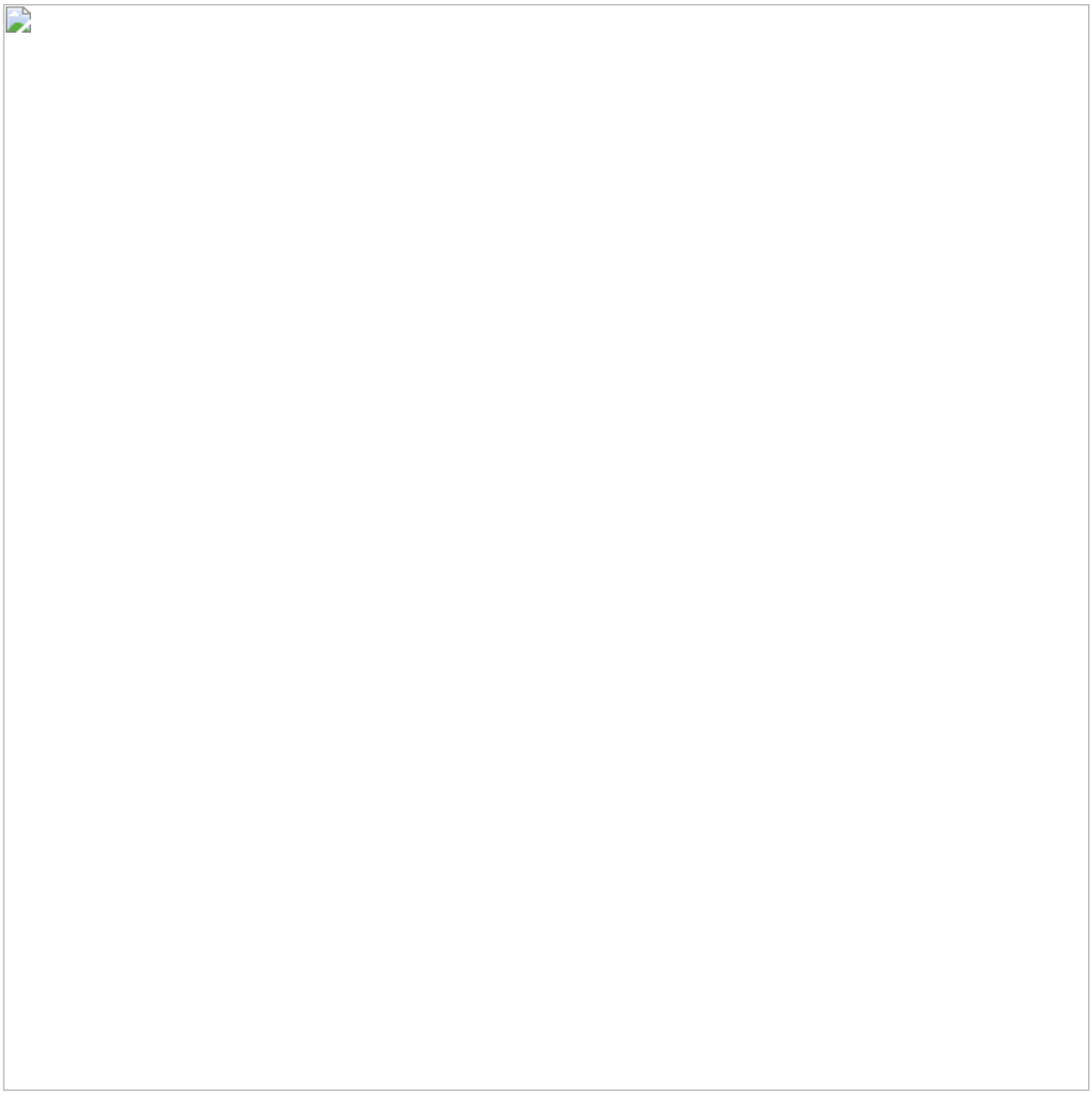
The mutexes come into play with the same “usage” as what was seen in the *BBSRAT* samples as well.



One of the places where mutex is set. Left: system.exe; Right: lockdown.dll

Given that even the sleep counter is identical, I would suspect that the two executables might share the same “base code” (or perhaps copied from the same “reference code”). The proximity of their compilation time also suggests that perhaps the same author is behind both executables.

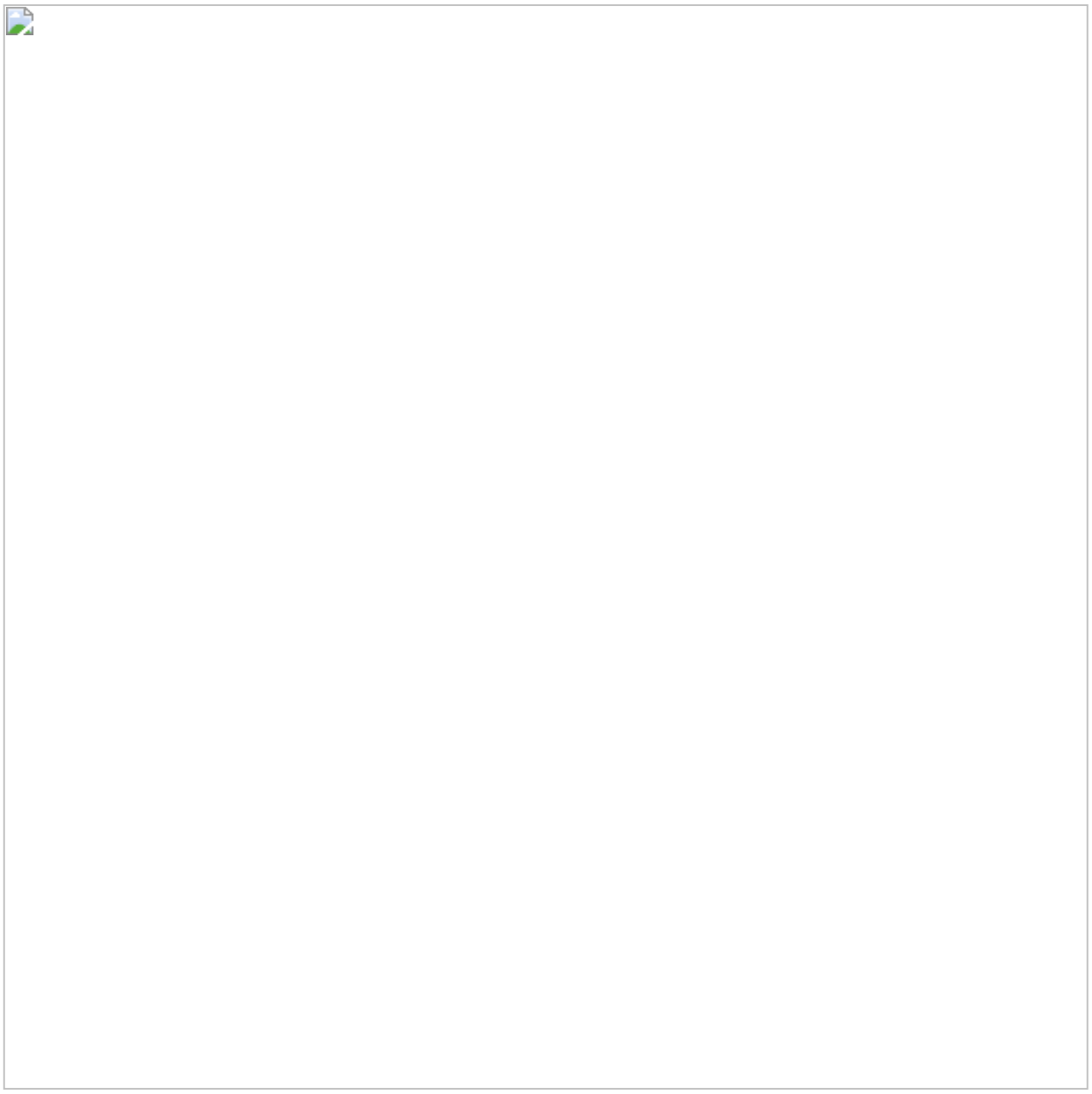
System.exe: 13 May 2018 19:34:27
Lockdown.dll: 6 May 2018 17:59:24



Same sleep counter. Left: system.exe; Right: lockdown.dll

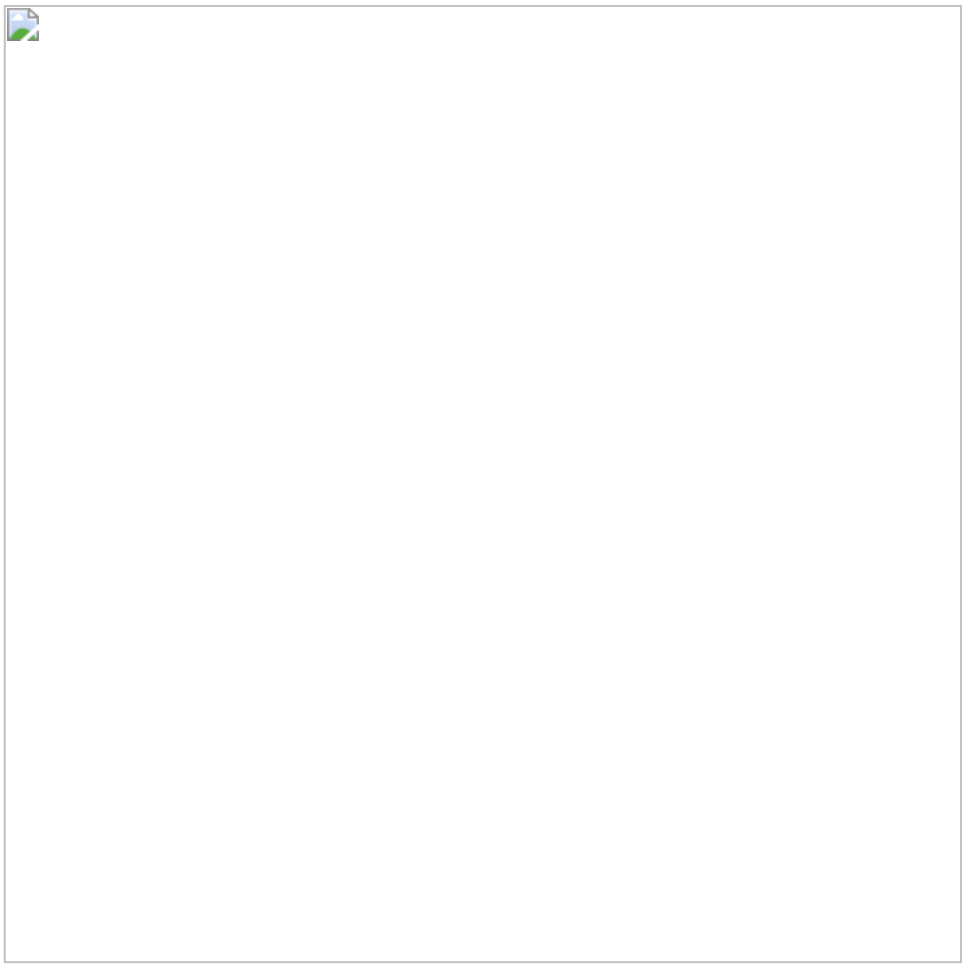
Enough of the dropper, let’s look at the actual evil payload now.

Diskshadow.exe



UPX section names

With just one look at it, we'd know it's UPX-packed. So let's unpack it quickly.

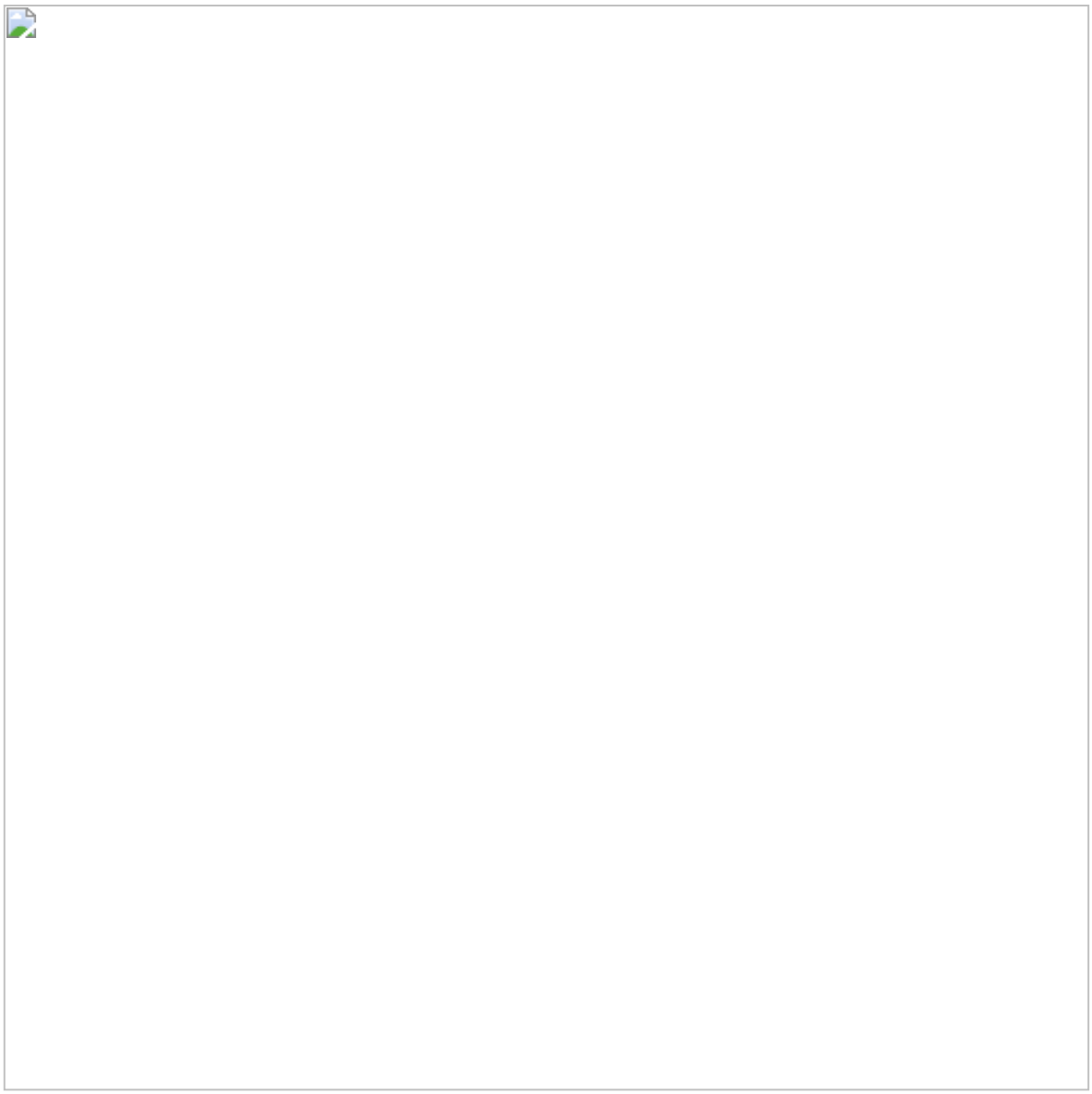


After unpack:

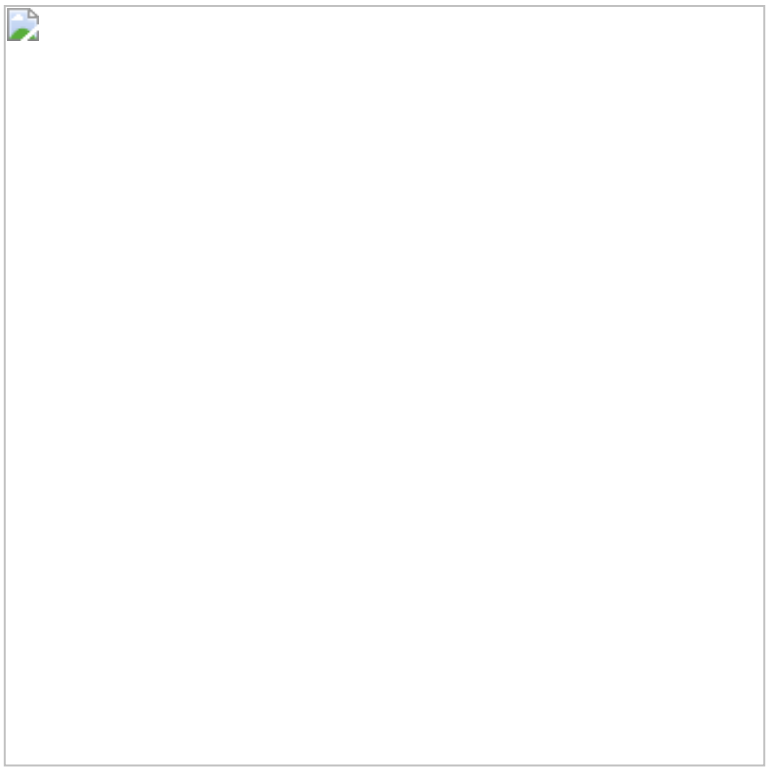
MD5: 40835ED7C92F33F7F377D4472228CB65
SHA1: 033E97D4AC3AE3CEC00A206F2AD5CCC922DBD326
SHA256:
C5BAB78FCA3DB0CE5FFFF5838A5A4A93D930E715DED1CBD8A5B3CAF0CDCE803C

With the assistance of Procmon, we can see that the binary will drop 2 files, intel.exe and inte.exe into the C:\intel directory. I located the code responsible for creating the files (refer to screen capture below). The binary also sets 2 persistency mechanisms. Note the presence of Simplified Chinese words within the name of the registry key — 更新计划程序 (translates to “ Update Schedule Program”). This is not the only place where Simplified Chinese words are observed.

> regsetval sz HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run “intel更新计划程序” “c:\intel\inte.exe”’,0
>
> shortcut “c:\intel\inte.exe” “~$folder.startup$” “Windows Calculator”



Set persistency
The two files dropped are:



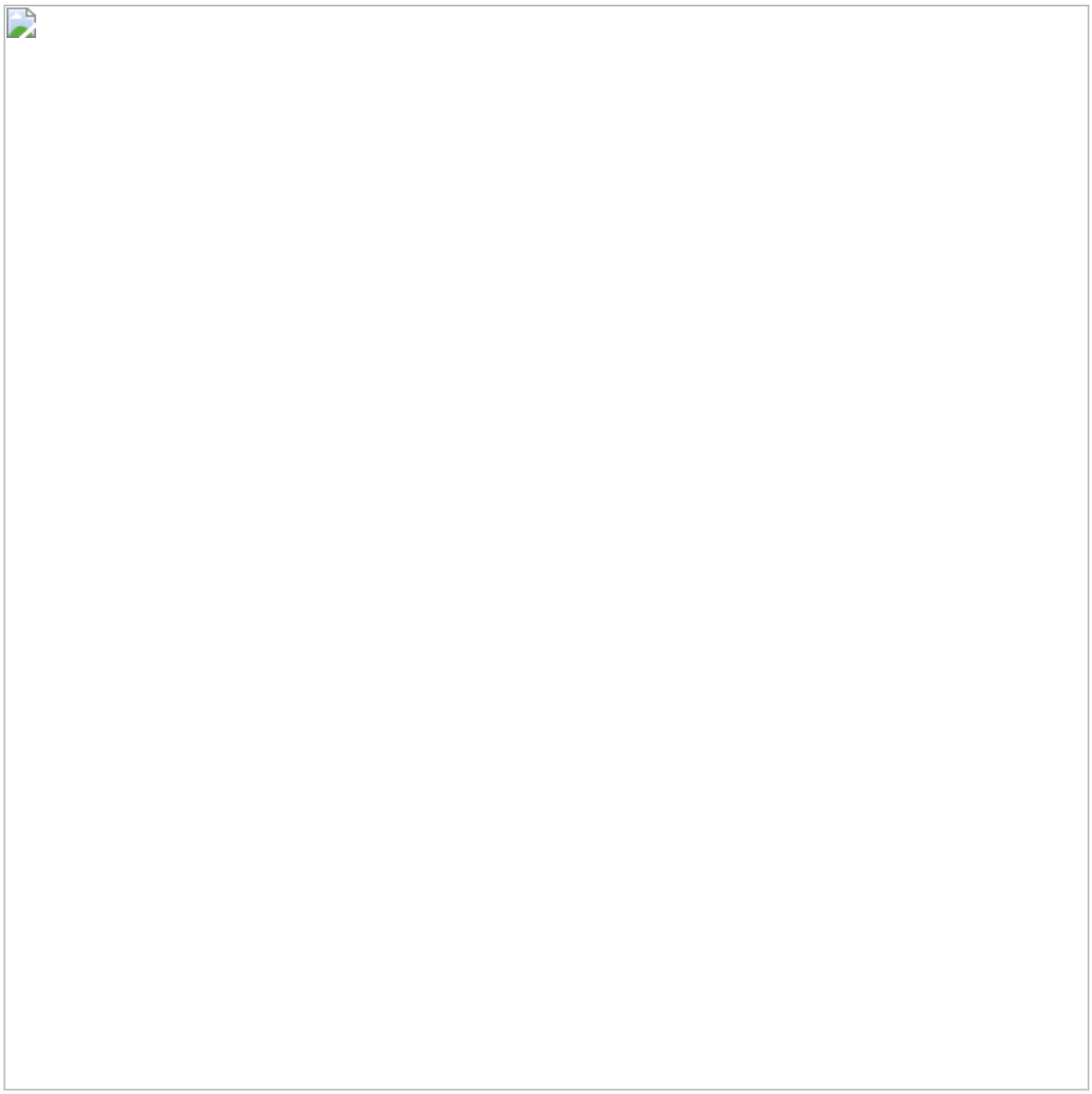
intel.exe
SHA256:
12459A5E9AFDB2DBFF685C8C4E916BB15B34745D56EF5F778DF99416D2749261
This is the NirCmd executable from Nirsoft. NirCmd is a small command-line utility that allows you to do some useful tasks without displaying any user interface.

*inte.exe*SHA256:
F46520C2284E20C42AFA6E9B90E380735BFDF29817828369D5F1270A887E6979
This is the actual *BlackNix RAT*, which is the meat that we want to analyze.

Both diskshadow.exe and inte.exe are written in Borland Delphi and their compilation datetime stamps are as follow:

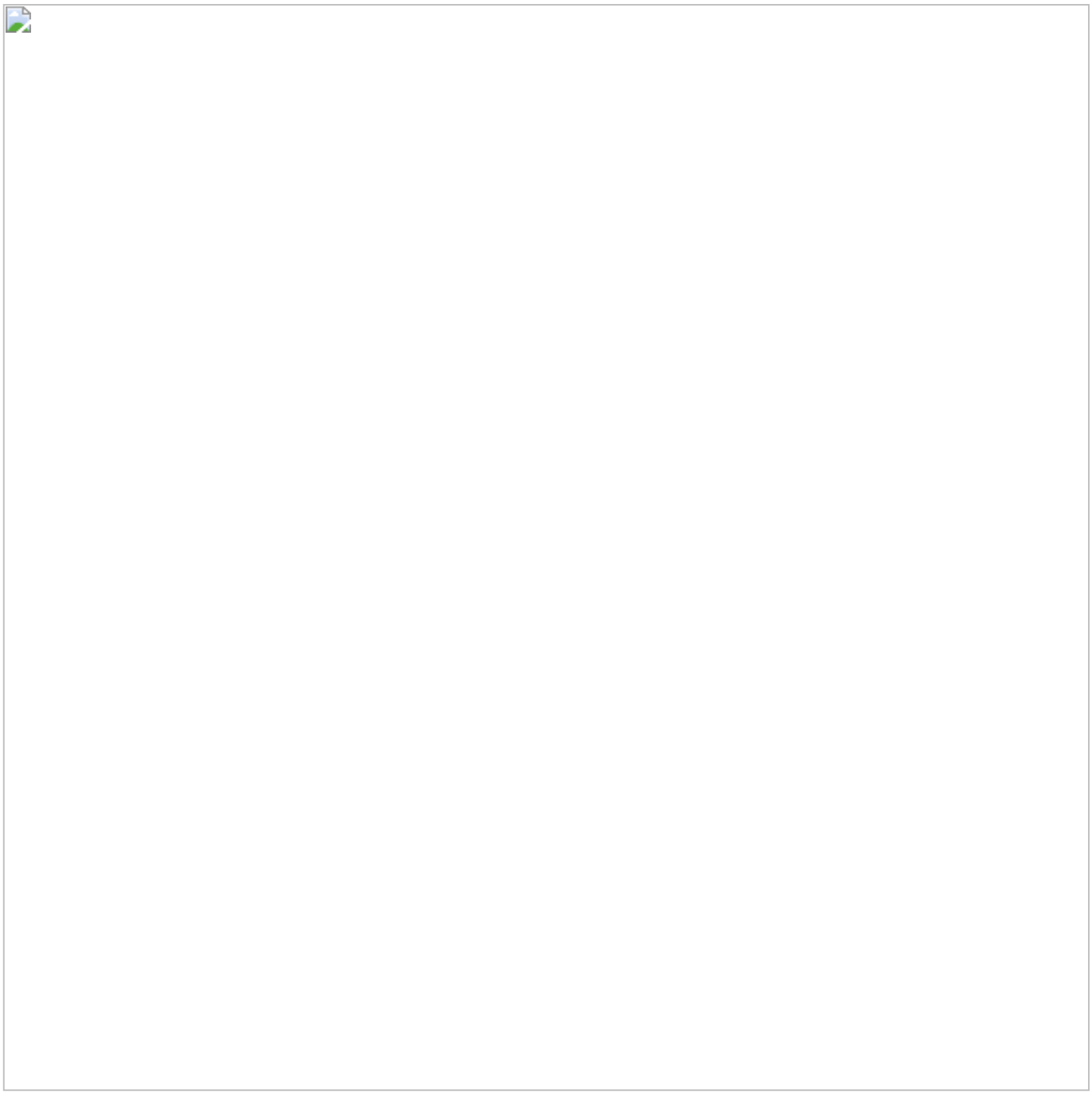
> File: C:\ProgramData\Microsoft\DeviceSync\diskshadow.exe
> PE Comp.: 1992–06–19 22:22:17 2A425E19, 708992537
> .rsrc comp.: 2018–10–22 22:50:06 4D56B643, 1297528387
>
> File: C:\Intel\inte.exe
> PE Comp.: 1992–06–19 22:22:17 2A425E19, 708992537
> .rsrc comp.: 2018–10–22 22:49:20 4D56B62A, 1297528362

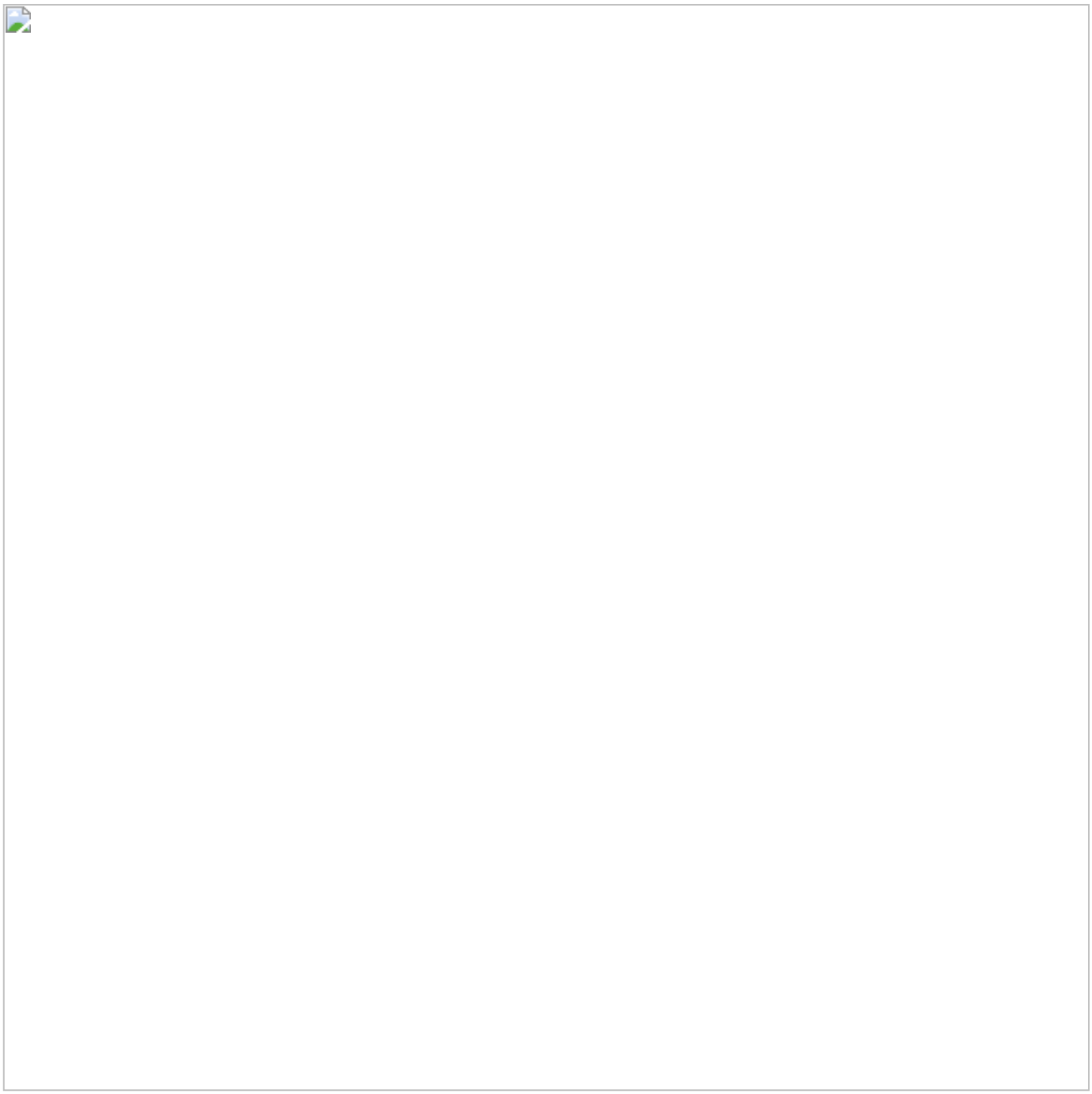
Judging from the compilation datetimes, they might be the output of a generator. Here, I could also make a guess at the chronological logic of when the files are prepared.



inte.exe

As with my usual style, I will start with a quick look at strings to try to guess the behaviour of the sample, before diving into dynamic and static analysis. Fortunately, the sample contains many helpful and descriptive strings that can help us deduce the features that this RAT provides, including Keylogger, FileManager, ProcessManager etc. Pretty typical RAT stuff.





A YARA rule hit

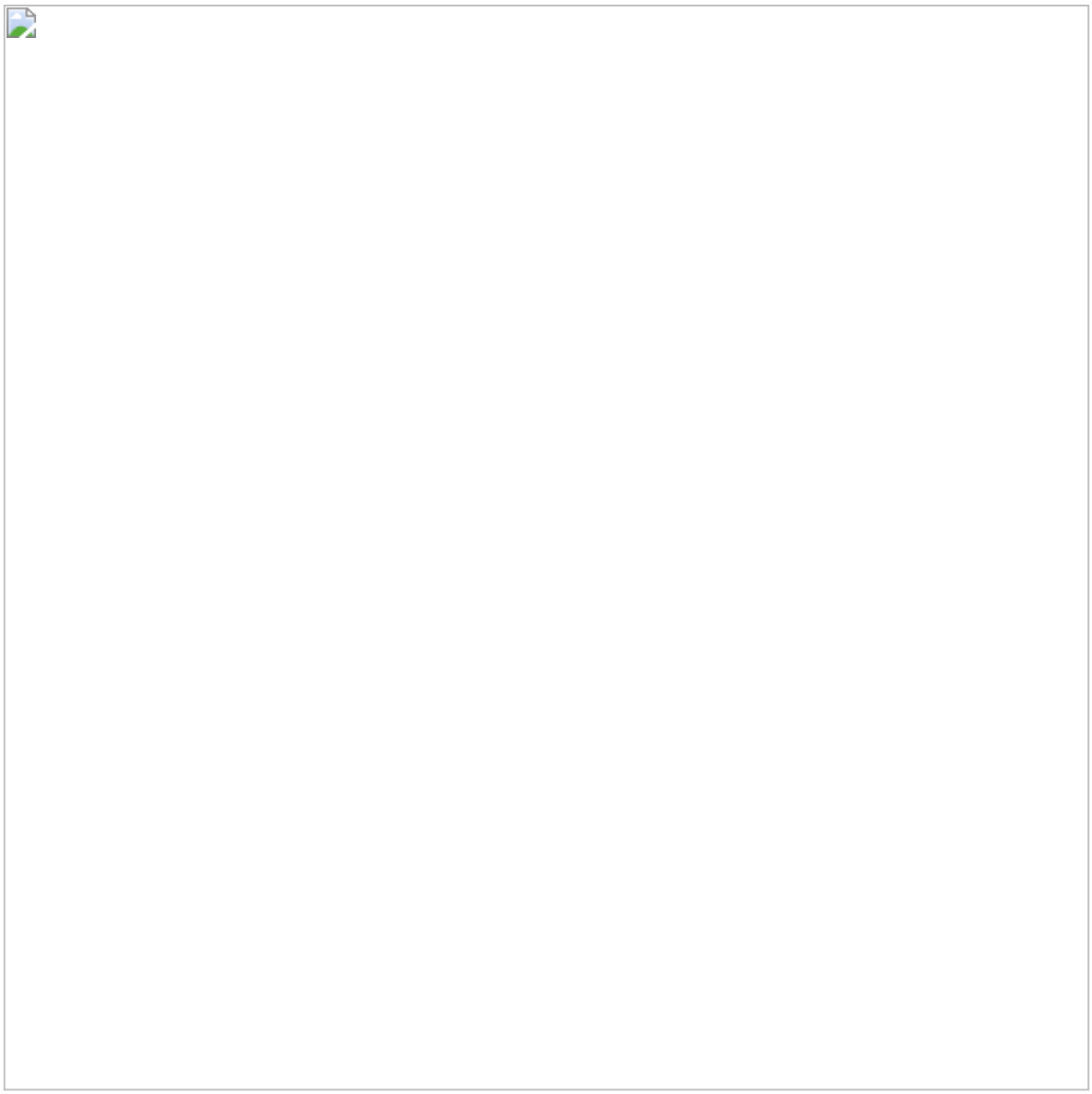
The sample happened to match James_inthe_box's YARA rule[2] on *BlackNix RAT*:

```
rule BlacknixRAT_bin{

meta:
description = “BlacknixRAT”
author = “James_inthe_box”
reference = “https://app.any.run/tasks/e3d845db-09b5-462d-8290-cbb4bb4a505f/"
date = “2019/02”
maltype = “RAT”

strings:
$string1 = “[Random-Number-Here]”
$string2 = “ScreenCapture”
$string3 = “TScreenSpy”
$string4 = “KeyLogger”
$string5 = “RemoteShell”

condition:
uint16(0) == 0x5A4D and all of ($string*) and filesize < 2000KB

}
```

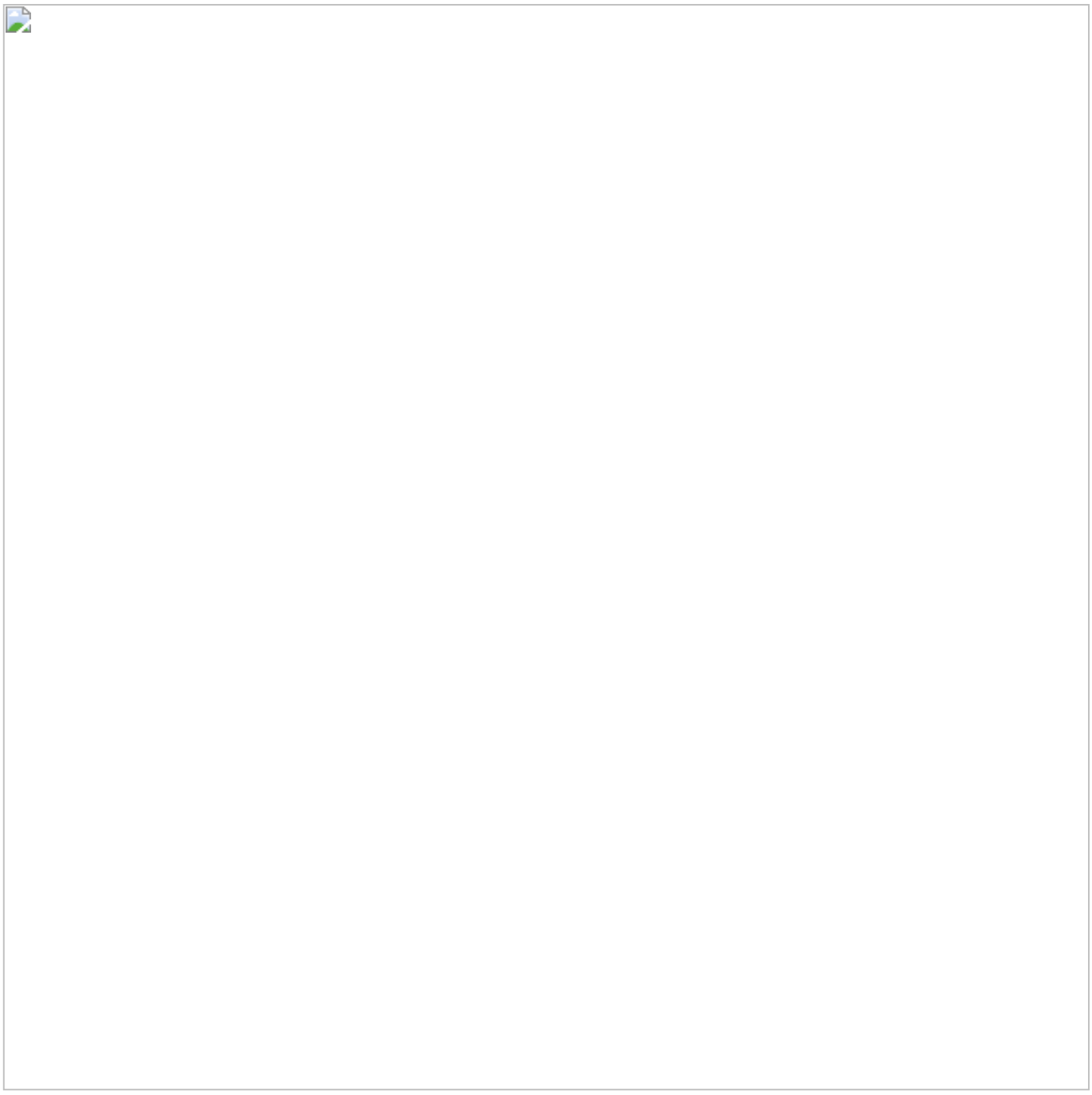
Based on the strings seen above, the strings that matched the YARA rule did not look unique enough to confirm that this is indeed a *BlackNix* RAT. James_inthe_box also provided a snort rule:

```
alert tcp any any -> any 80 (msg:”Blacknix RAT Detected”; flow:established,to_server;
content:”|32|”; depth:1; content:”|7c 78 01 6d 8e|”; within:10;
reference:url,https://app.any.run/tasks/e3d845db-09b5-462d-8290-cbb4bb4a505f/;
classtype:trojan-activity; sid:20166298; rev:1; metadata:created_at 2019_07_18;
```

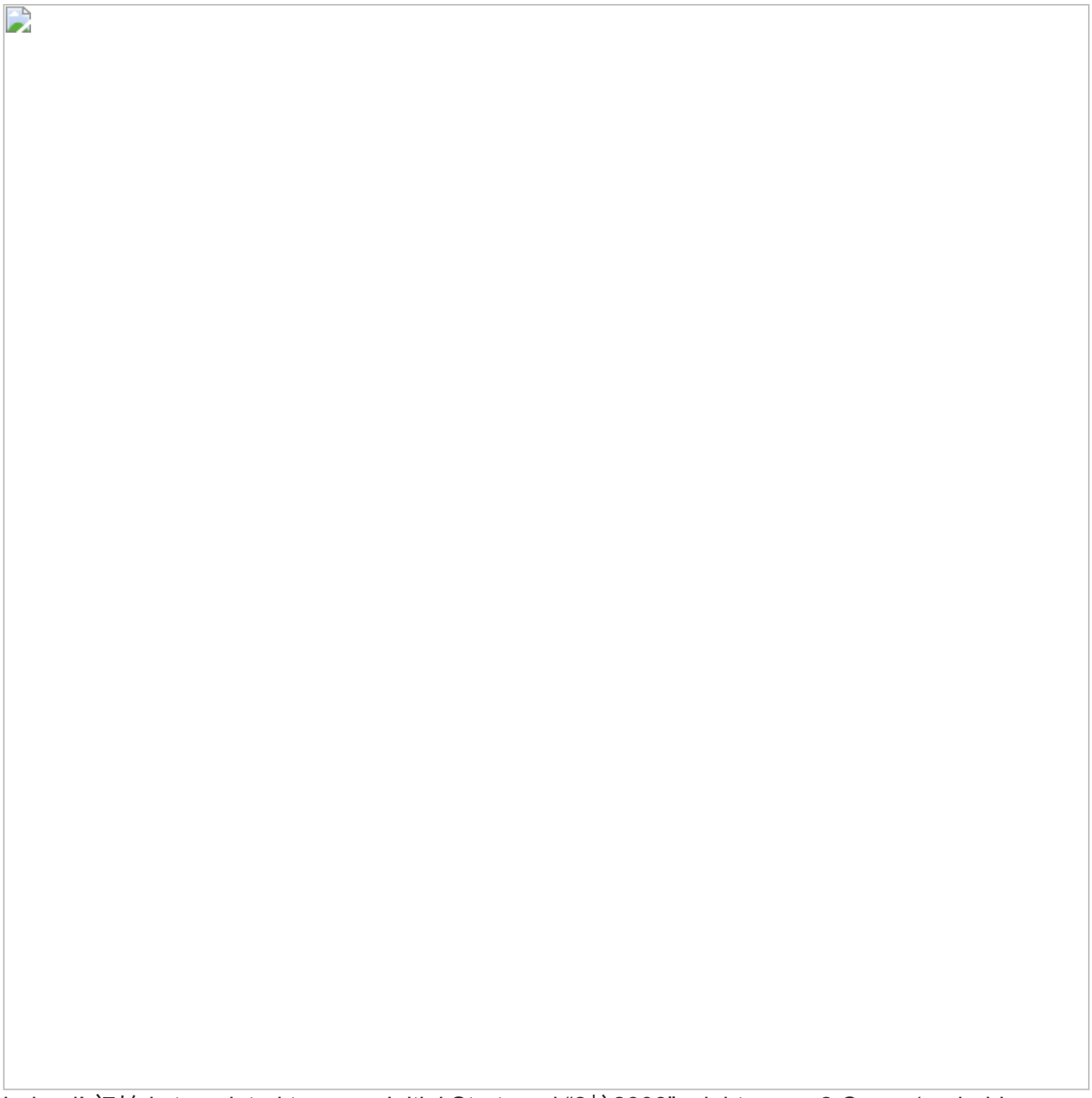
Let’s see what we have in the network data.



Yup, I see a 7C 78 01 but that’s not an exact match with the pattern in the snort rule. Hang on a second.. 78 01 looks like the ZLIB magic header. There we go!



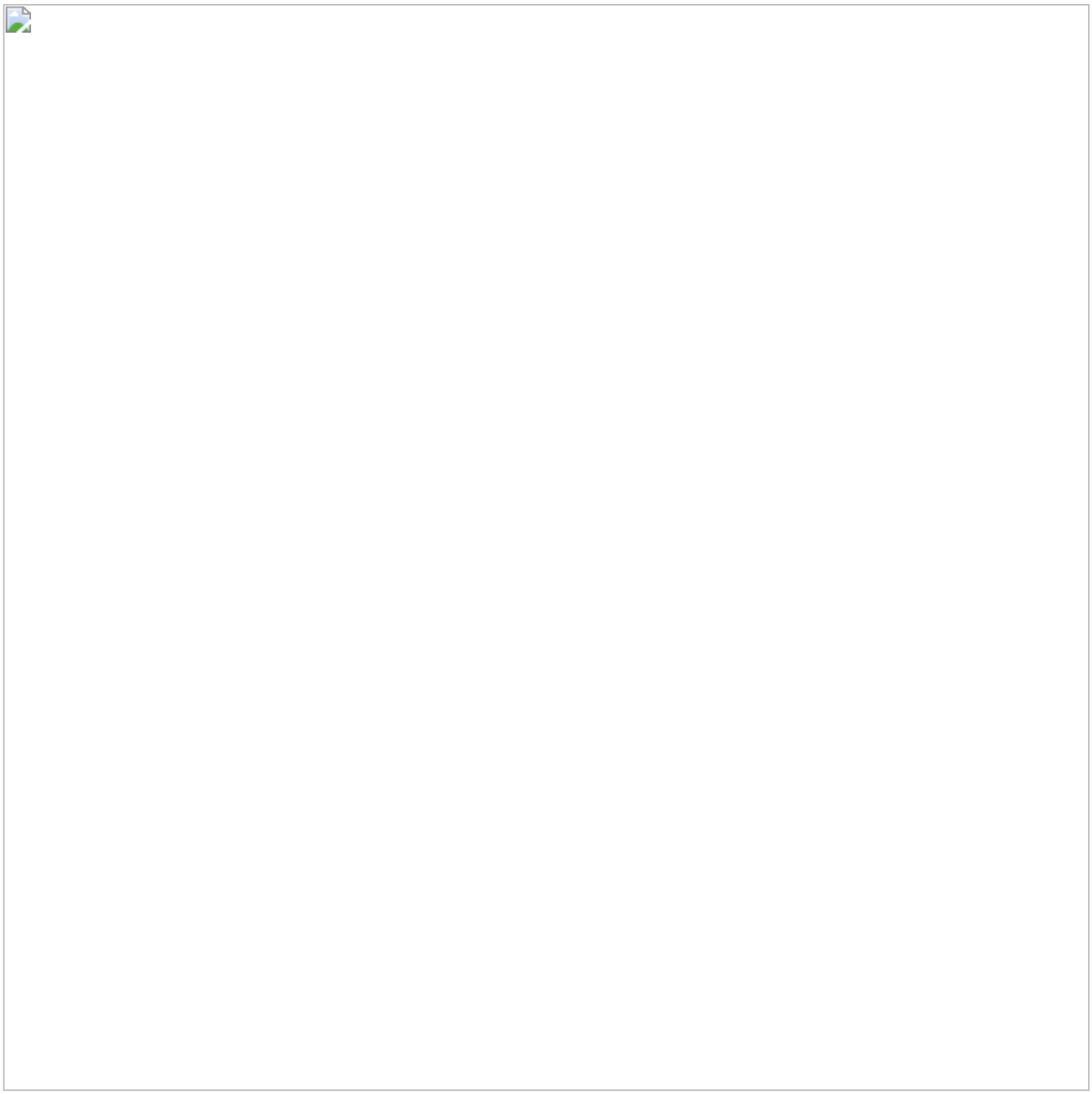
I noticed some weird characters, which could be indicative of Unicode (maybe Chinese…?). Let's try.



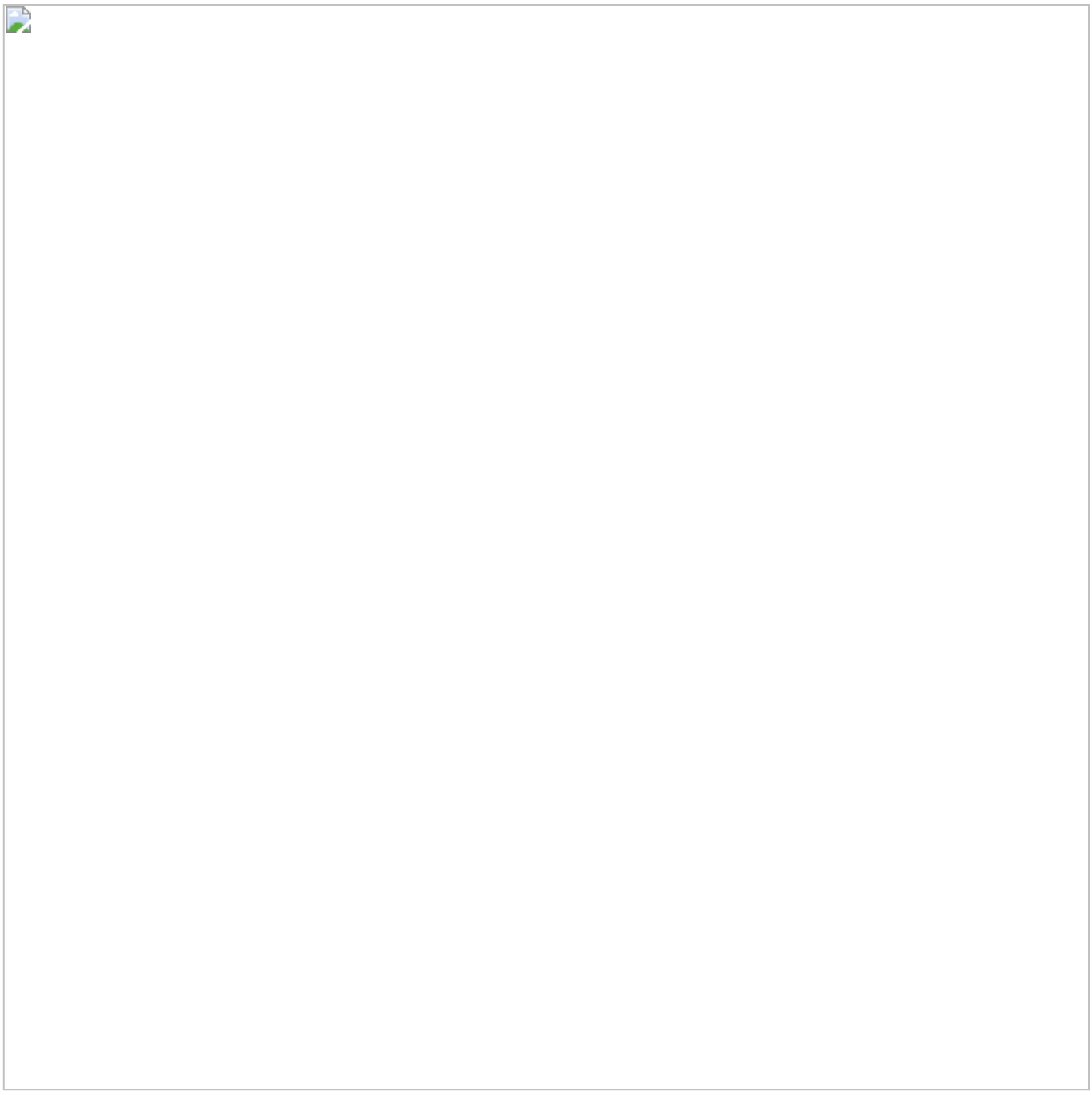
Indeed! 初始 is translated to mean Initial Start, and “2核2808” might mean 2 Cores (probably referring to the CPU cores). I’ll step through the code that forms up this data in abit. Now, let us try to confirm if this callback belongs to *BlackNix* family.

The YARA and snort rules mentioned above referenced a sample (SHA256: A4DA694DED531EC60CA5A242C554B6A7062E12FF633D34656C4CA9DF86E42DD5). Let’s sidetrack and see what this sample does.

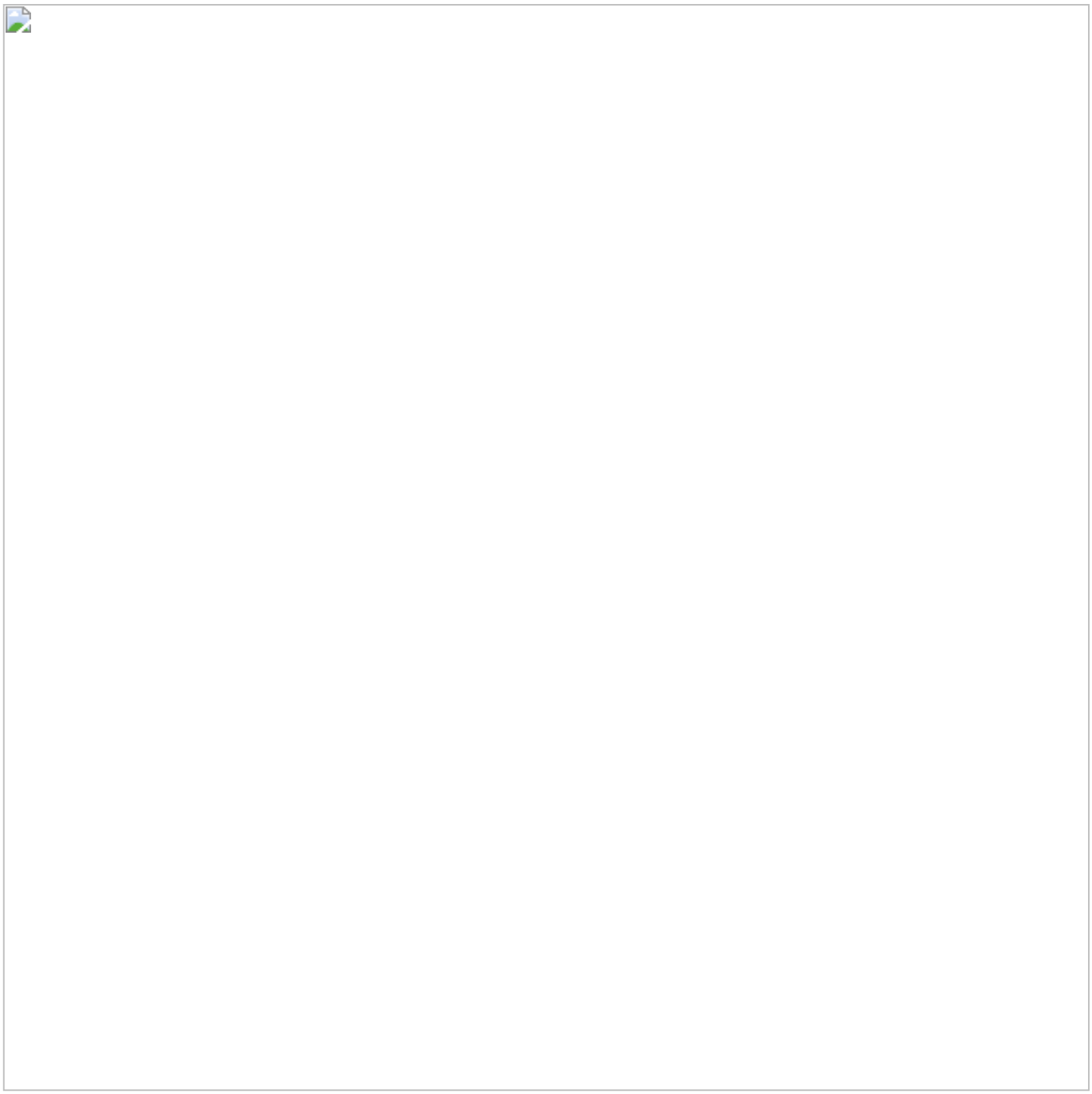
This sample is packed with VMProtect, so to save time, I’m just going to execute it and see what happens. Upon execution, a new file phpalpha.exe is created in C:\Intel\ExtremeGraphics\CUI\Resource. Turns out the file has the same hash as the parent binary.



The network callback looks like this:



We already know that’s a ZLIB header, so let’s use CyberChef to view the inflated data:



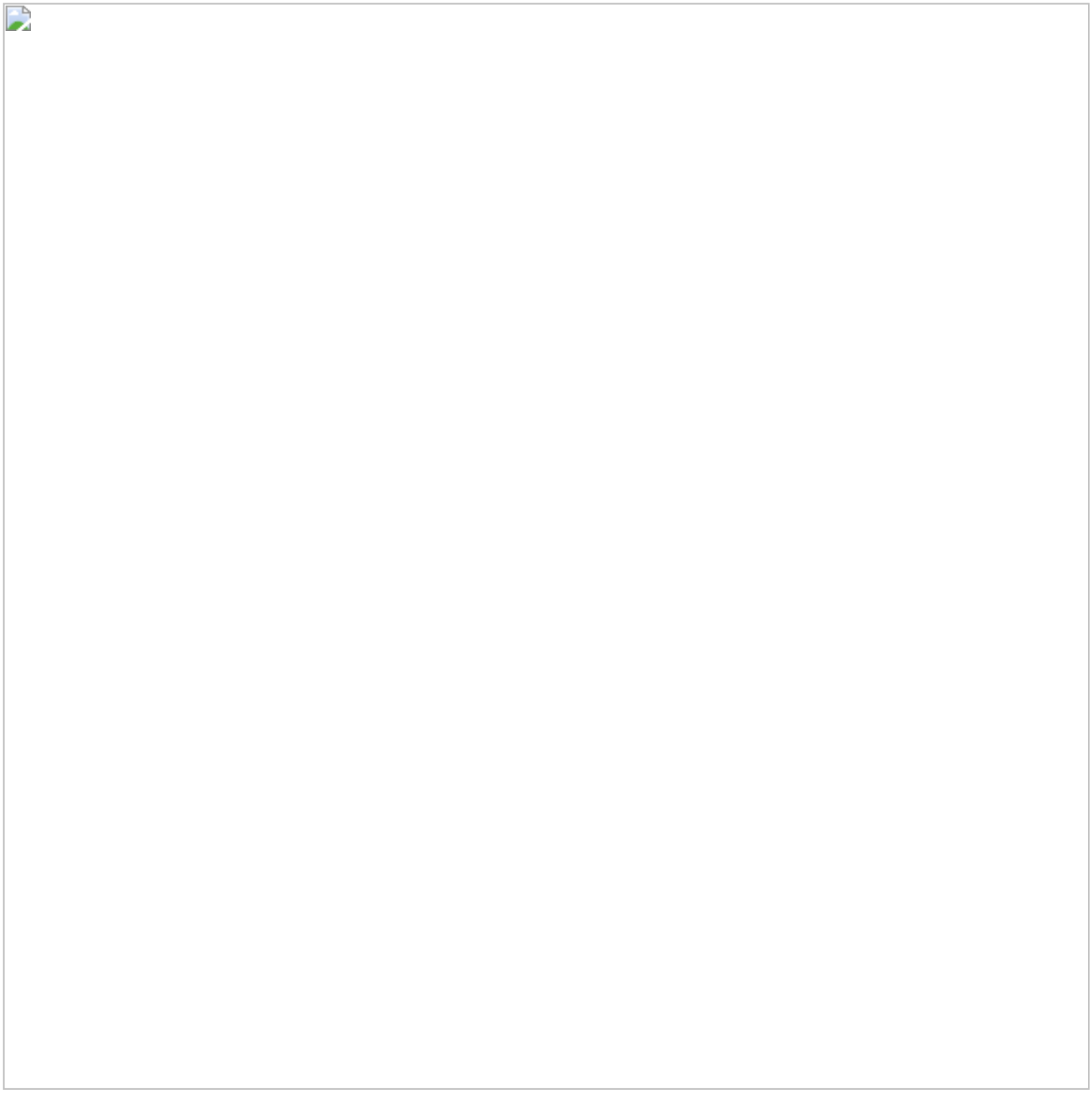
Interesting. The data structure and keywords are identical. Now I can say that the sample (SHA256: A4DA694DED531EC60CA5A242C554B6A7062E12FF633D34656C4CA9DF86E42DD5) and our sample (inte.exe, SHA256: F46520C2284E20C42AFA6E9B90E380735BFDF29817828369D5F1270A887E6979) belong to the same family. Is it really *BlackNix* though?

With the C2, everything is easier

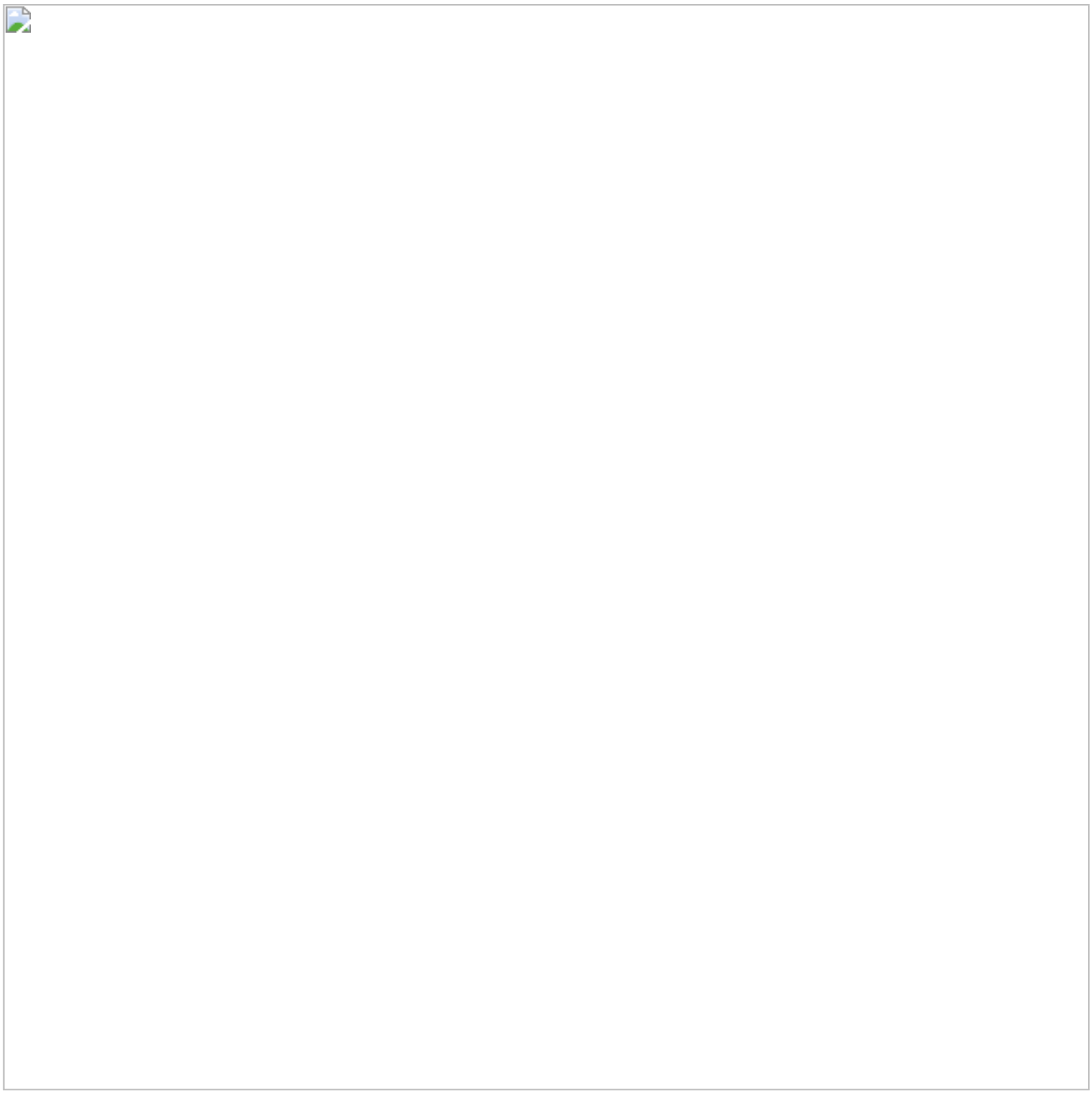
With some help from Google search, I found a copy of a *BlackNix* C2 component :D

The C2 executable comes with the ability to generate the “server” component. This naming convention is common in RATs, where the malware client is typically referred to as the “server”, and the C2 is the “client”.

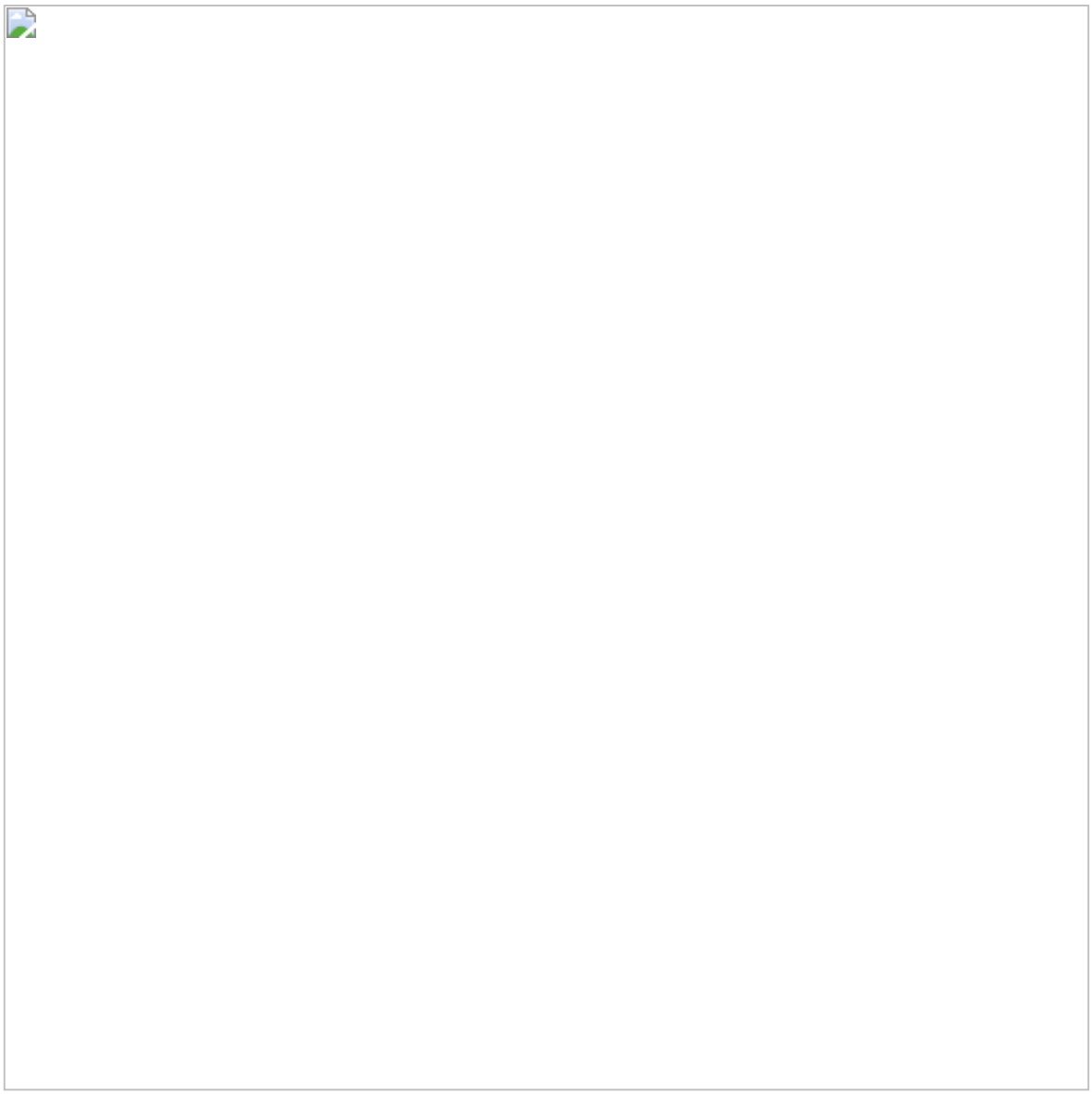
The following is one of the default profiles loaded with the C2:



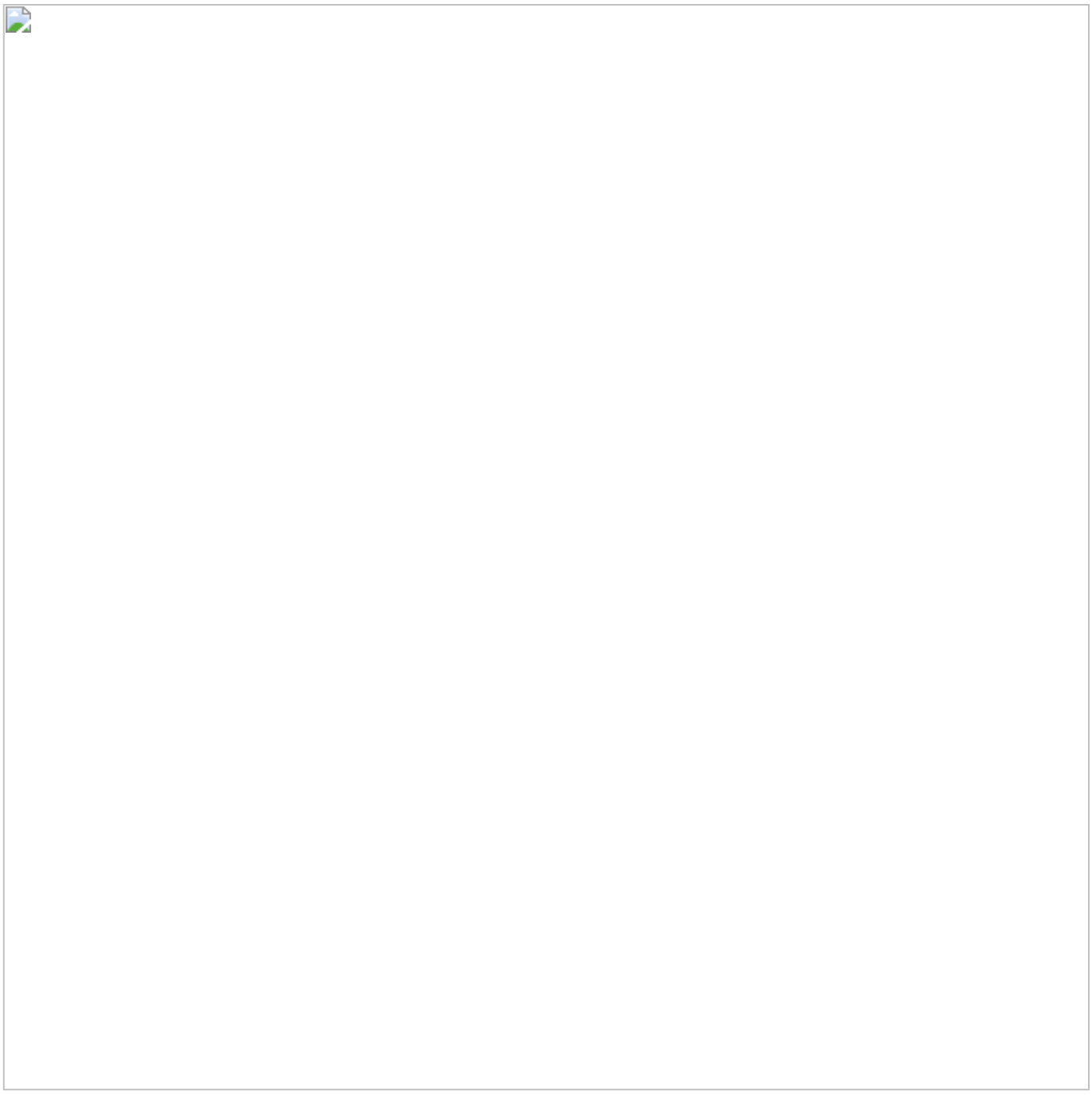
For the ease of testing, I changed some of the values when generating our test binary. The generator even comes with the option of UPX-packing the generated binary if the user wishes.



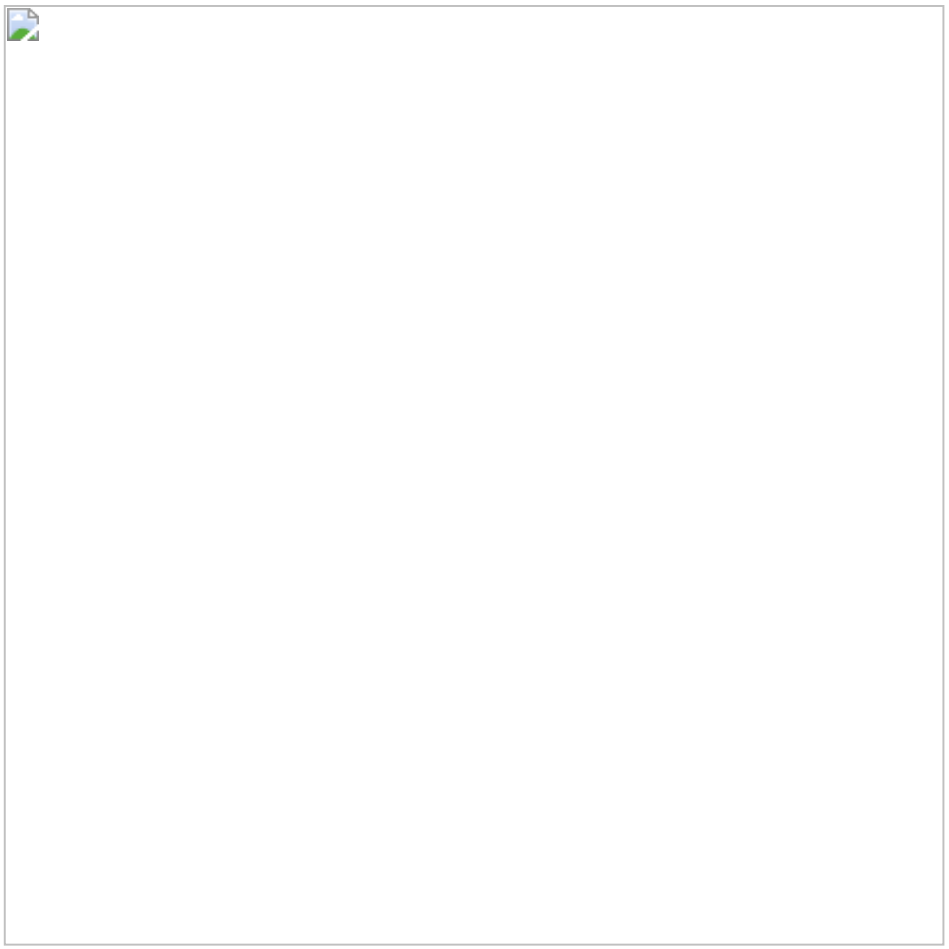
Inspecting the strings within the generated binary, we can quickly identify some familiar keywords.



I’ve written a quick script to read the strings from the default settings. This will come useful later, when comparing these strings across different samples.

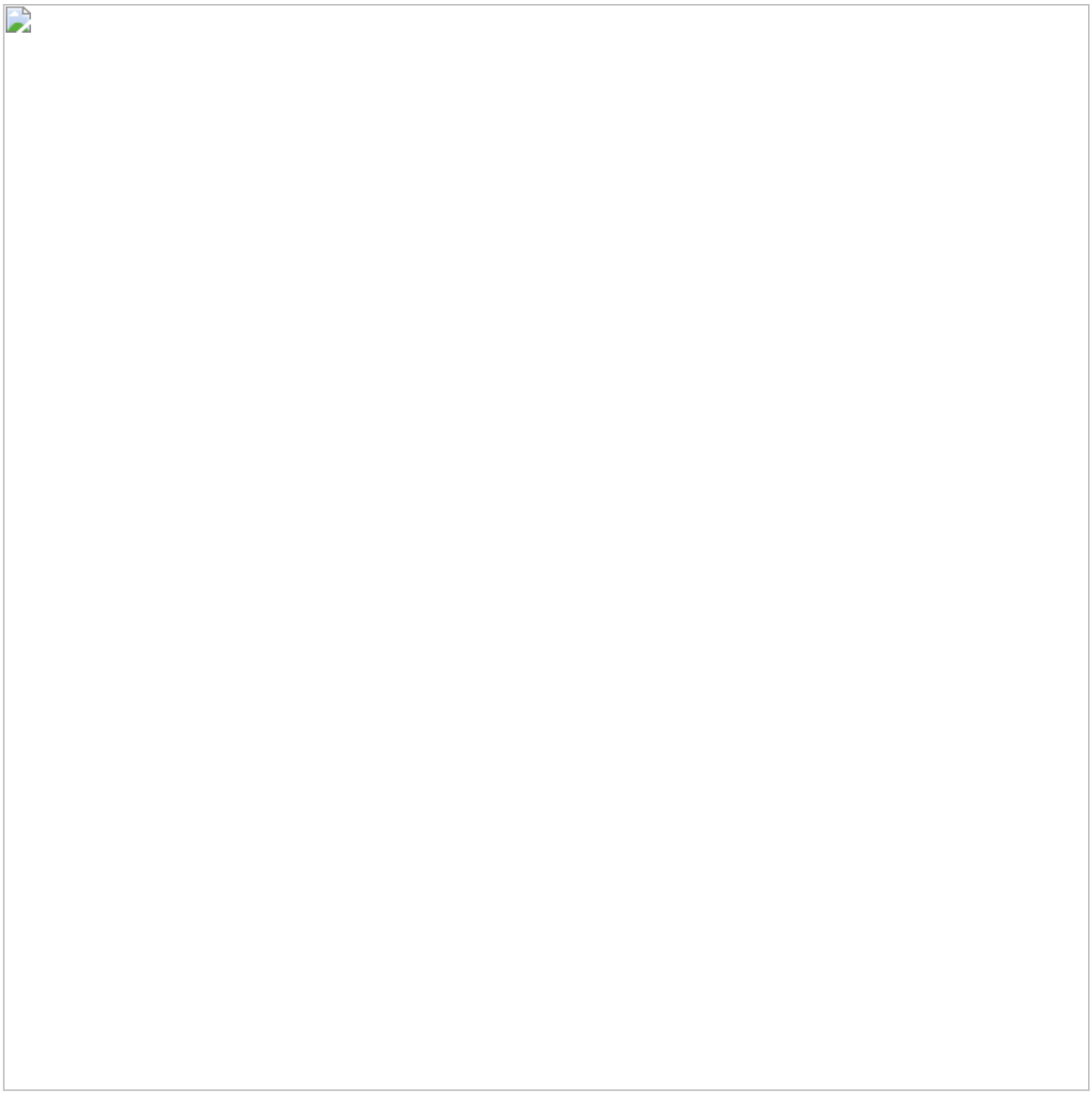


Here’s the output of running the script:

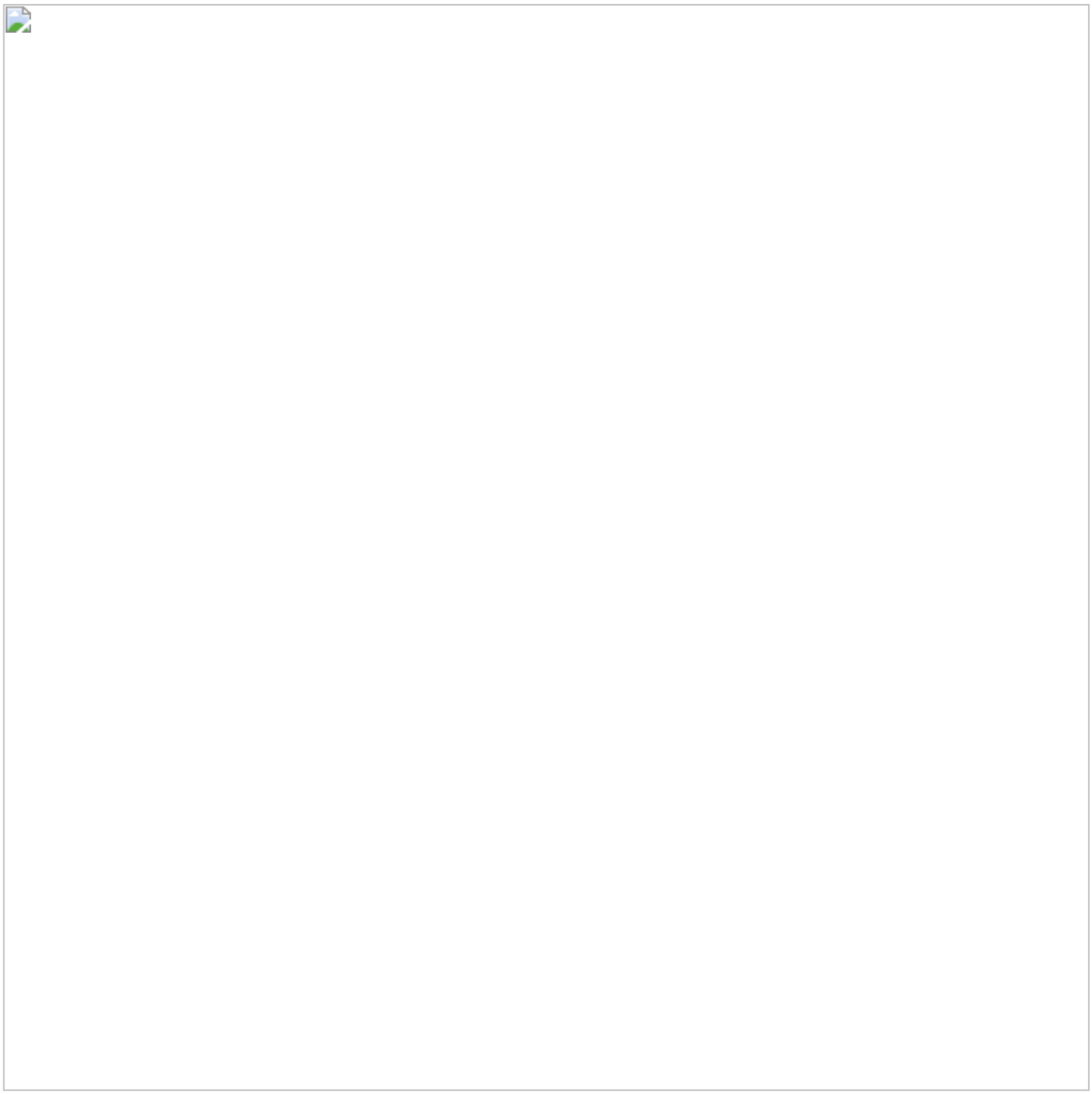


It appears that these SETTINGS strings have nothing to do with the configuration set when generating the binary. The default connection password within the C2 is “admin” and notice that even if I changed the password when generating the binary, the new password does not get inserted into this SETTINGS data. These may be part of a “stub” that comes with the C2 executable and inserted into every generated binary. I think this may be a helpful piece of information when trying to identify if a set of *BlackNix RAT*s is communicating with the same C2 executable (or at least the same version).

Take a look at the network communications.



First thing that I noticed was the difference in the way the “Processor” information is being formatted. Remember there were some Chinese words (2核2808) that I thought refers to Processor Cores? In the data sent from this test binary, the processor information was simply a “2808” (referring to 2808 Mhz, which is indeed the setup of my VM).

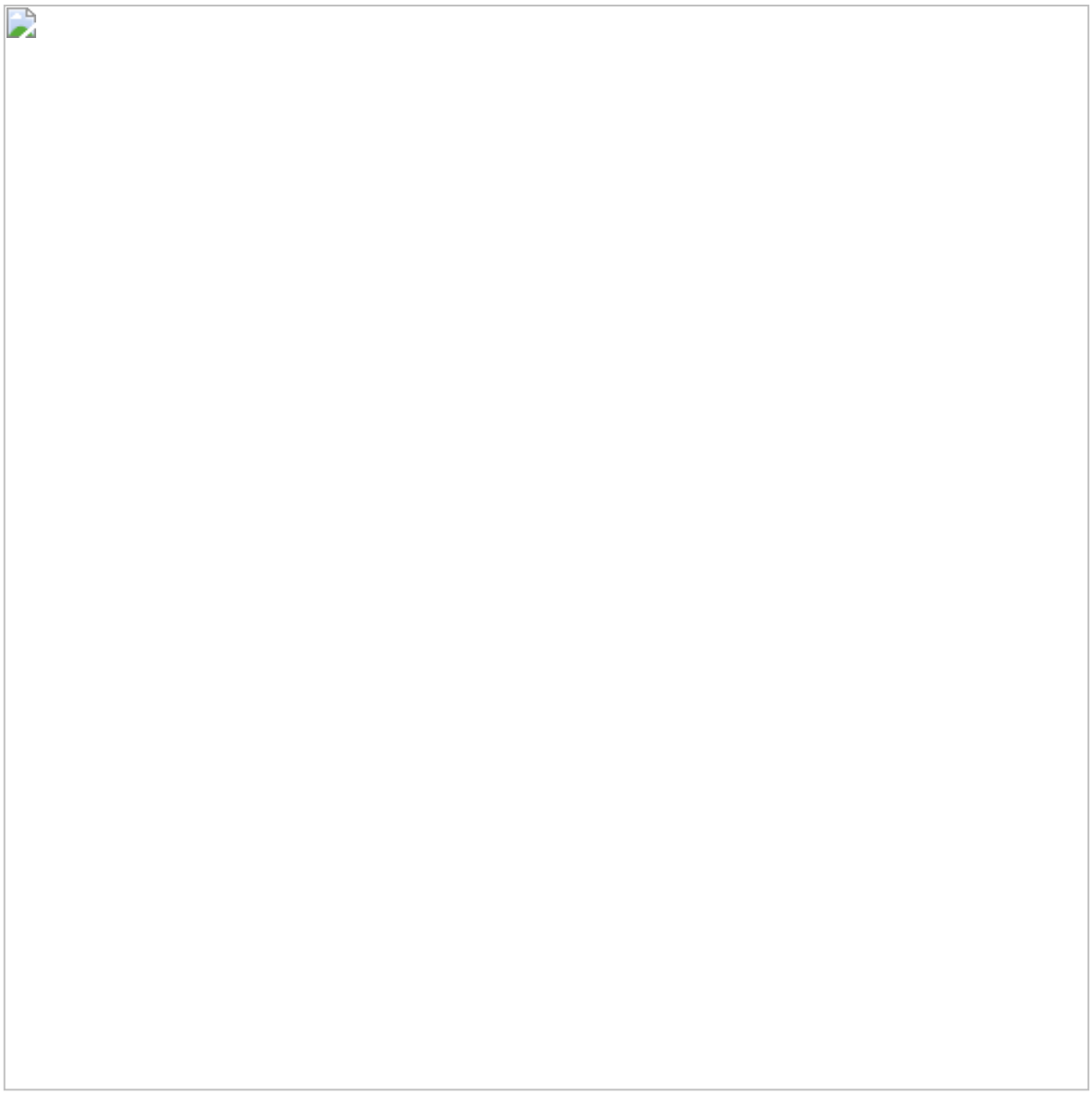


Connected victim on C2 dashboard

Let’s get back to the sample we have at hand.

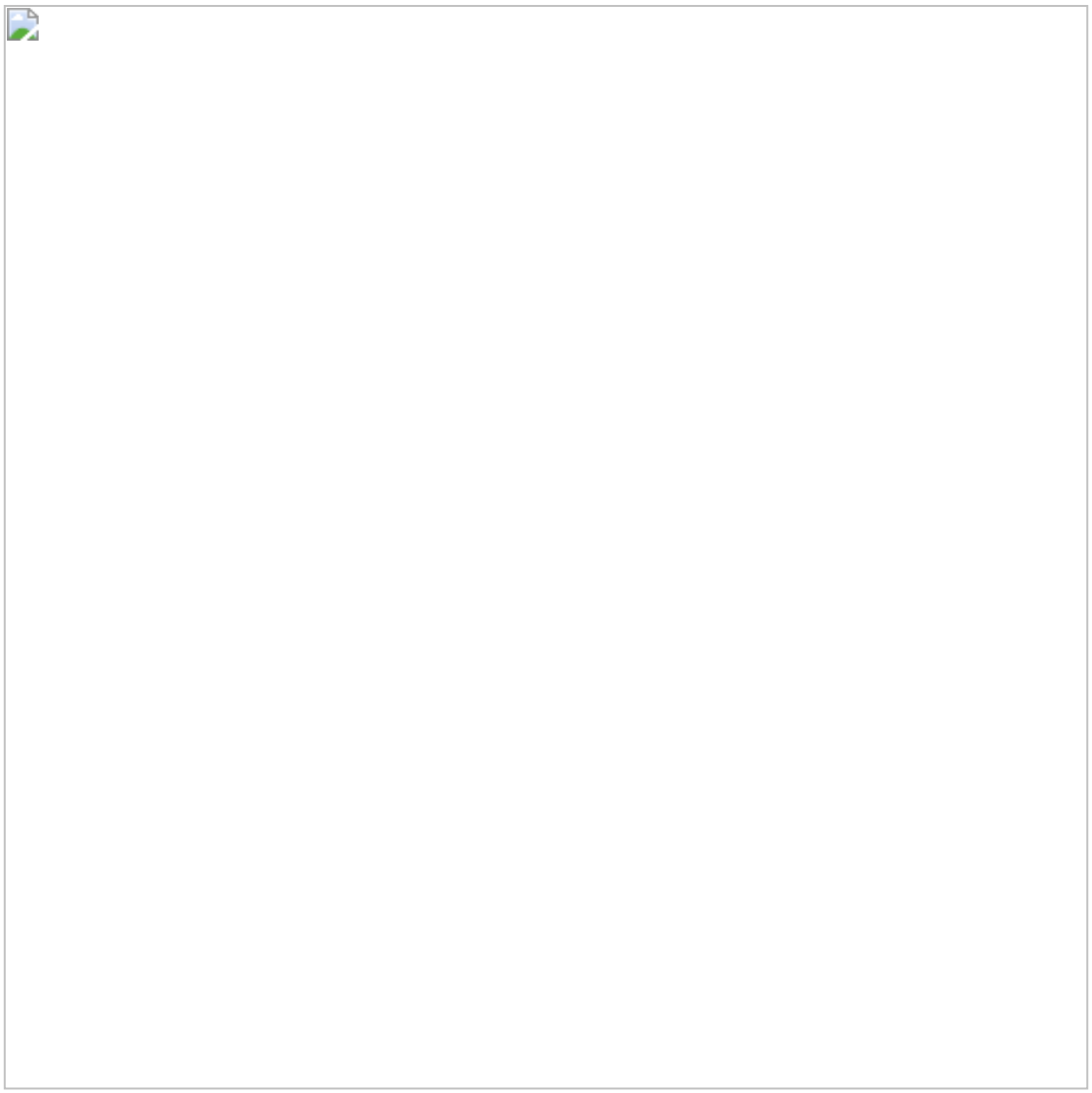
I did an in-depth analysis of how the first beacon’s data structure is formed within inte.exe.

Earlier I mentioned some SETTINGS strings. These are the strings that are populated into a data structure and some of these values are later copied into the first beacon data.

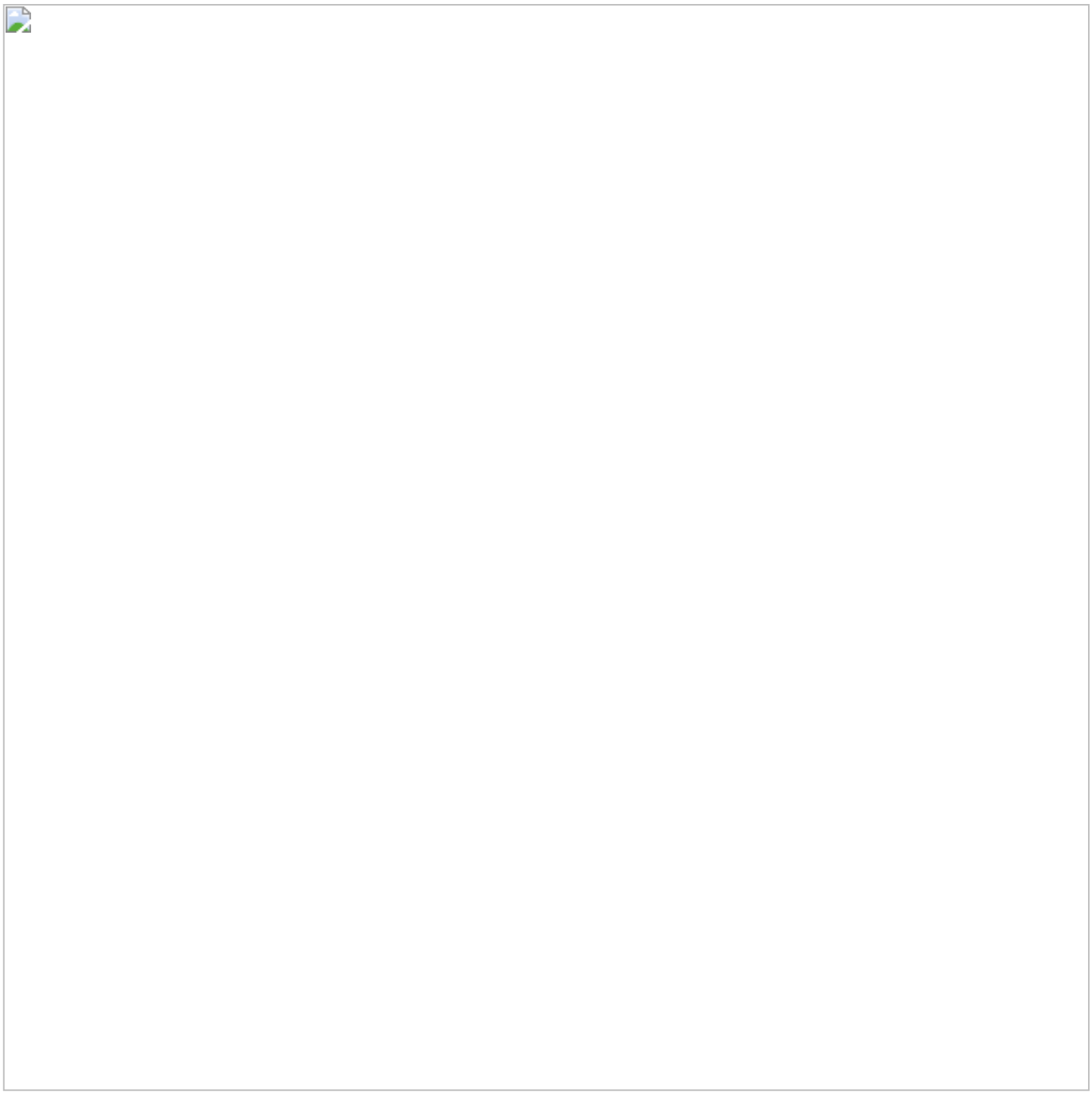


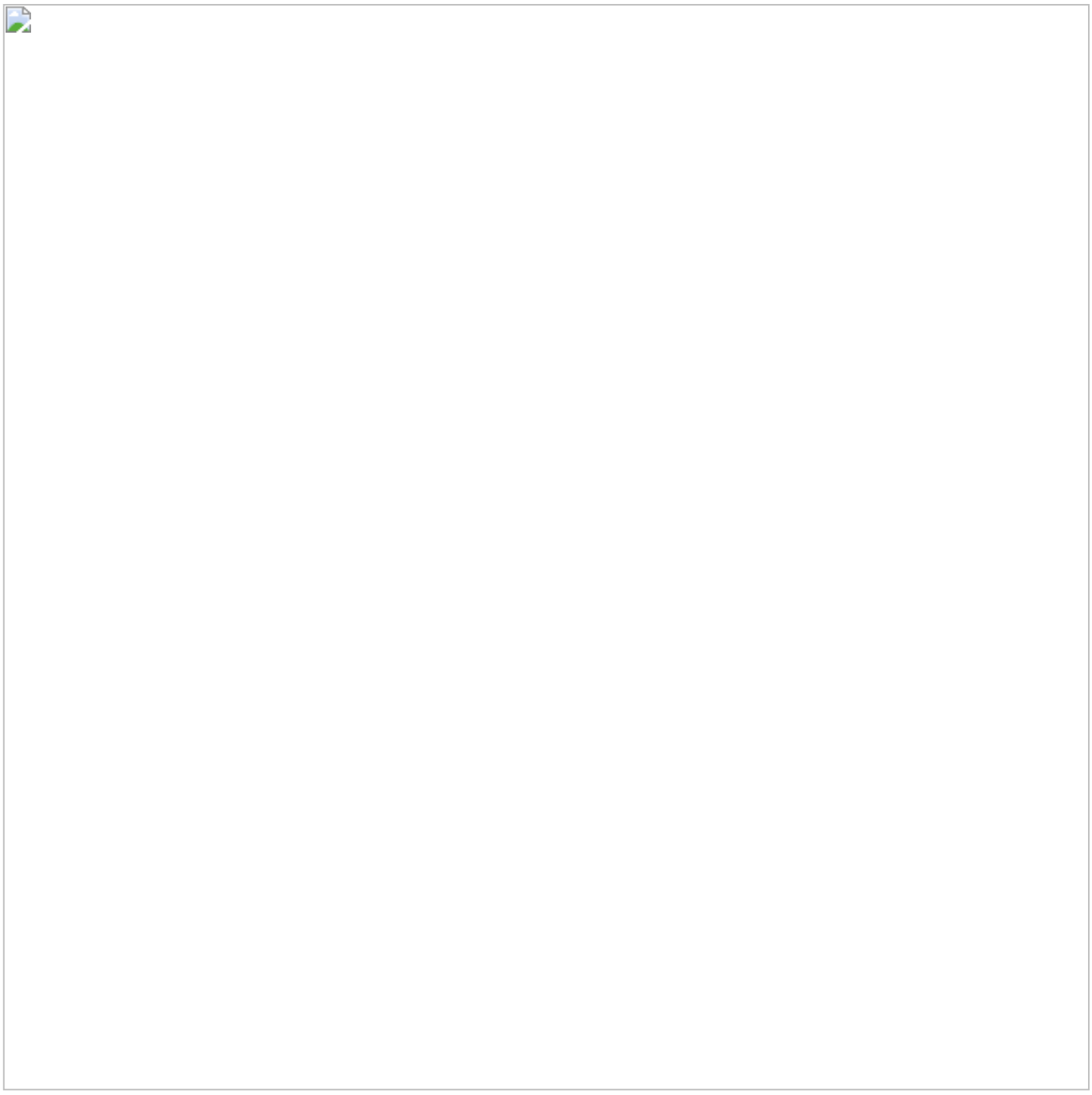
Reading strings from SETTINGS

Let’s compare the default SETTINGS strings found in the generated *BlackNix* binary and inte.exe.



The following code is responsible for building the structure to be sent in the first beacon:





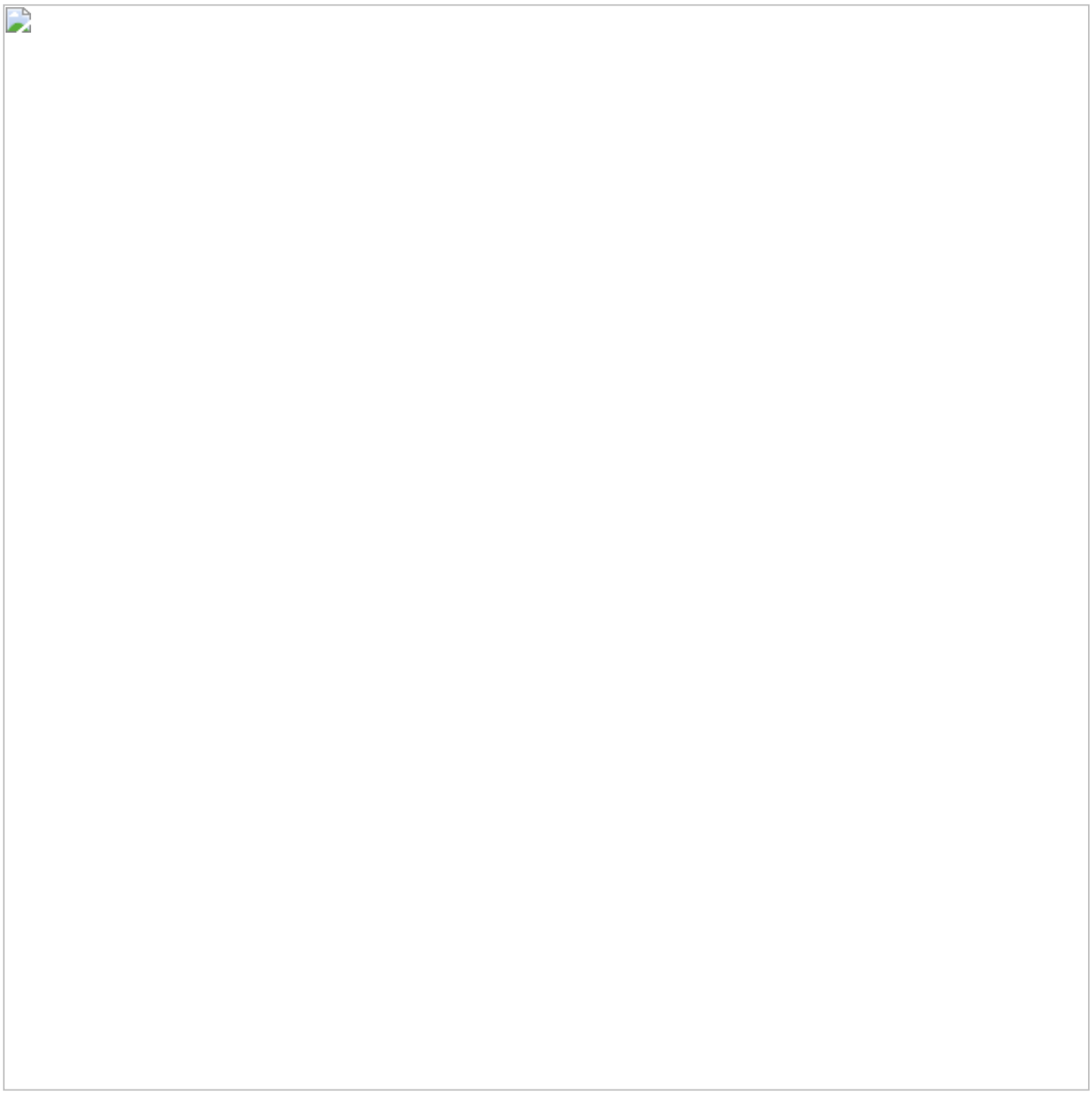
The following is the deduced first beacon's data structure sent from the inte.exe to its C2.

> OnConnect|Default 初始|Username|Username|Computer Name|IP Address|Hardcoded Space|Locale|Is Machine Idle?|Locale|Language|Account Privilege|Processor|Memory|Foreground Window Text|OS|Default False|Default False|%Root%|%Desktop%|%MyDocuments%|%AppData%|Locale|Server authentication password|ProdID, InstallDate|

Now we can play spot-the-differences. We can guess what each of these fields mean by looking at what can be seen on the dashboard, without reverse engineering the binary.

The following is the deduced first beacon's data structure sent from the test *BlackNix* binary.

> OnConnect|Assigned Group|Assigned Name|Username|Computer Name|IP Address|Webcam Installed?|C2 Version|?|Locale|Language|Account Privilege|Processor|Memory|OS|True/False?|True/False? |%Root%|%Desktop%|%MyDocuments%|%AppData%|Locale|Server authentication password|?|



Comparison of fields within data structure sent to C2

Yes! inte.exe is a BlackNix RAT, but has a different/modified C2 component?

What I did above proved that the inte.exe sample is indeed a *BlackNix RAT*, judging from the highly similar data structure within the initial beacon and the similarities found within the executables. However, since some fields in the communicated data are interpreted differently,

I’m guessing there is a customised C2 that the adversary is using. I am not even able to tell the version number of the C2 from the sample, perhaps it is not important for the adversary. However, the SETTINGS strings found within the samples could be a way for us to differentiate variants.

So far, I’ve walked through the analysis of this set of files:

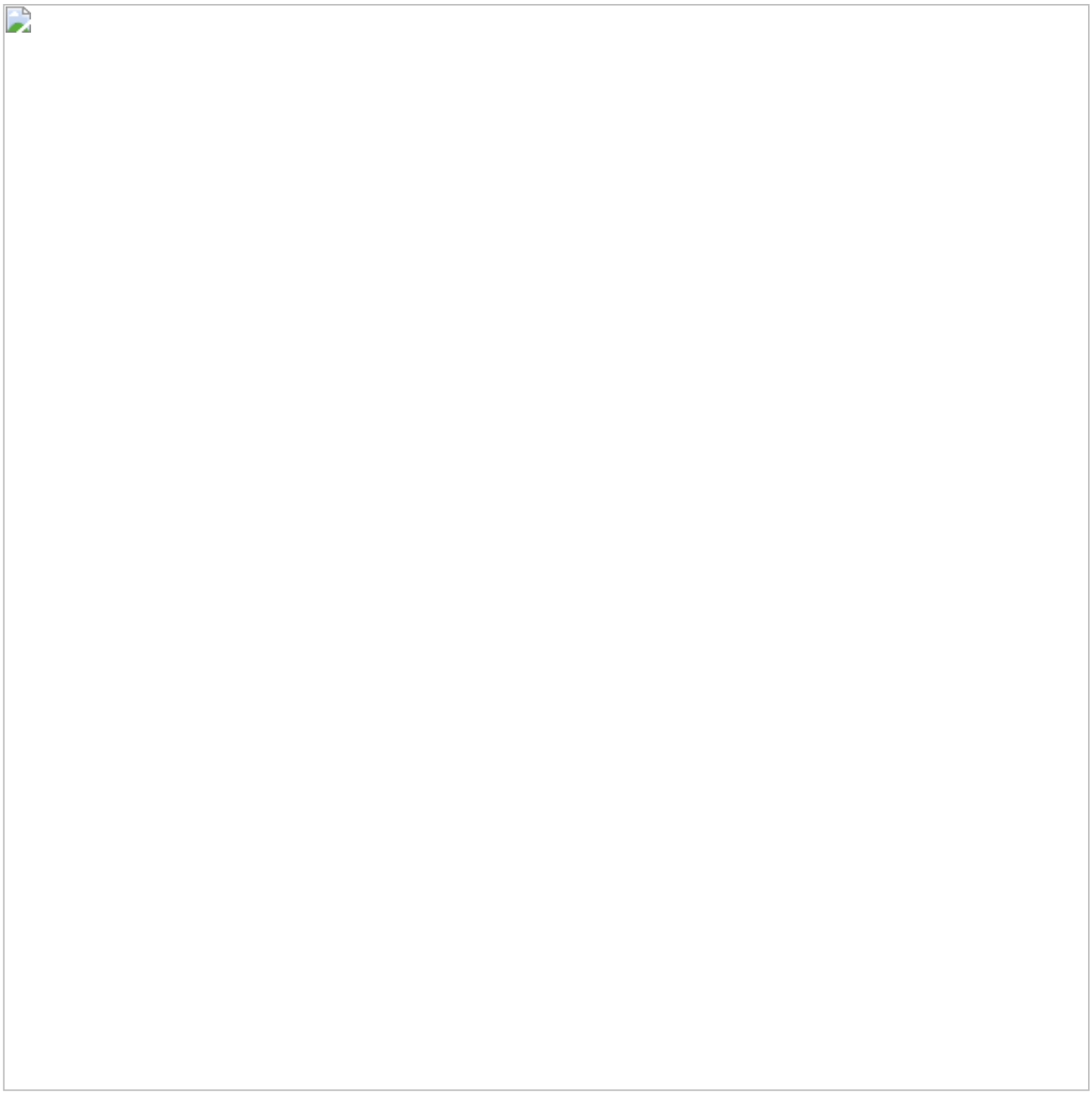
· **1st level droppers** (Project1.exe)
daaa061c88b197fa92d9648306e79875e3a24f392550dacaabd22e5fdba53ebf
75dc821013fe92ef93cefa47d3fe83ad5ce90658e8ef01fcdb0b11652397abec

· **2nd level droppers** (diskshadow.exe)
f0311ede2dd5e752411bf181626e3cdb36737affe67ddeb8af028d0c44355886
c5bab78fca3db0ce5ffff5838a5a4a93d930e715ded1cbd8a5b3caf0cdce803c

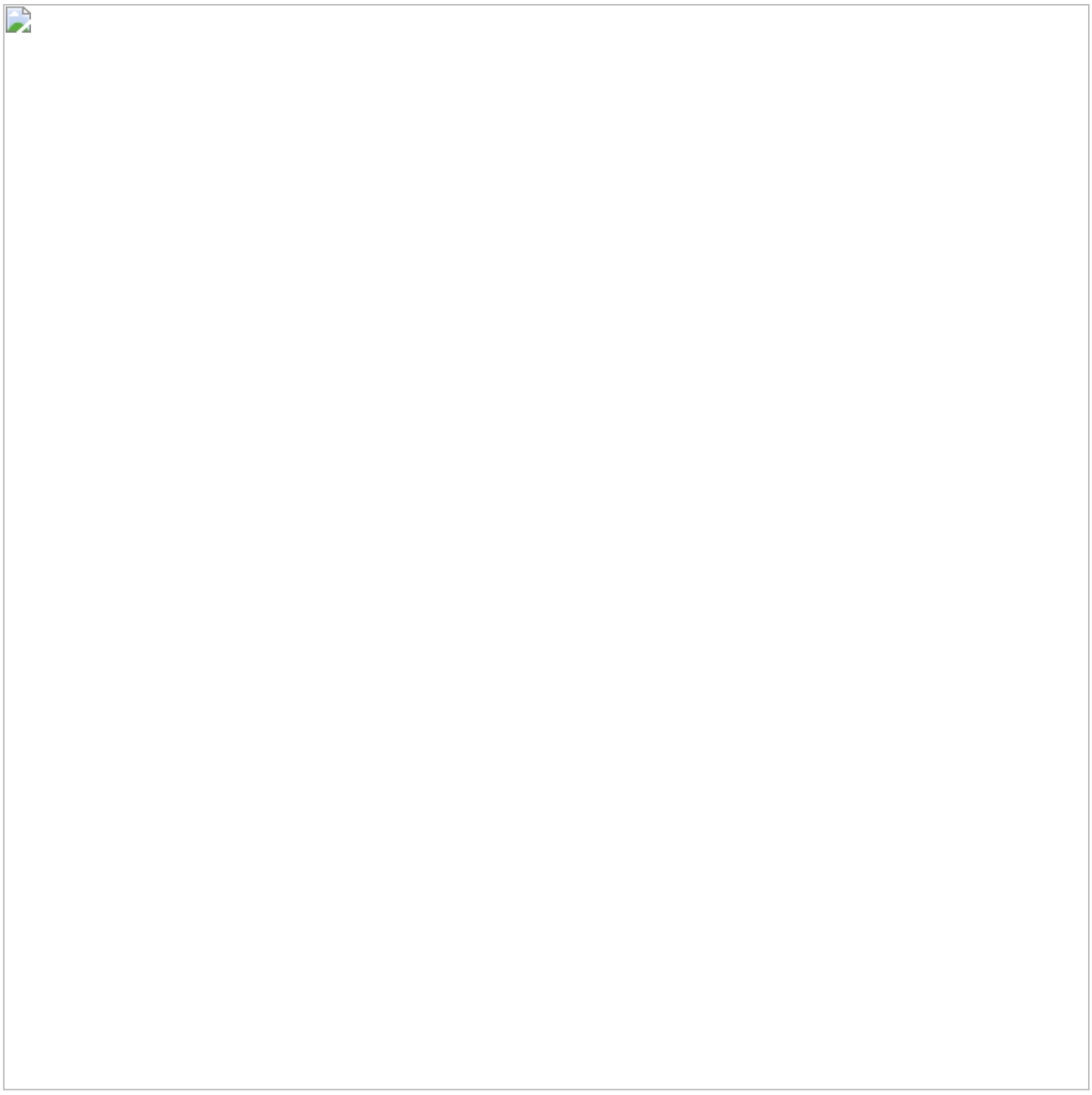
· (inte.exe)
f46520c2284e20c42afa6e9b90e380735bfdf29817828369d5f1270a887e6979

I’ve verified that the sample inte.exe is indeed a *BlackNix RAT* and communicates with a custom *BlackNix* C2 at IP address 112.213.107[.]134.

Related to this IP address, other possible *BlackNix RAT* samples were found on VirusTotal:

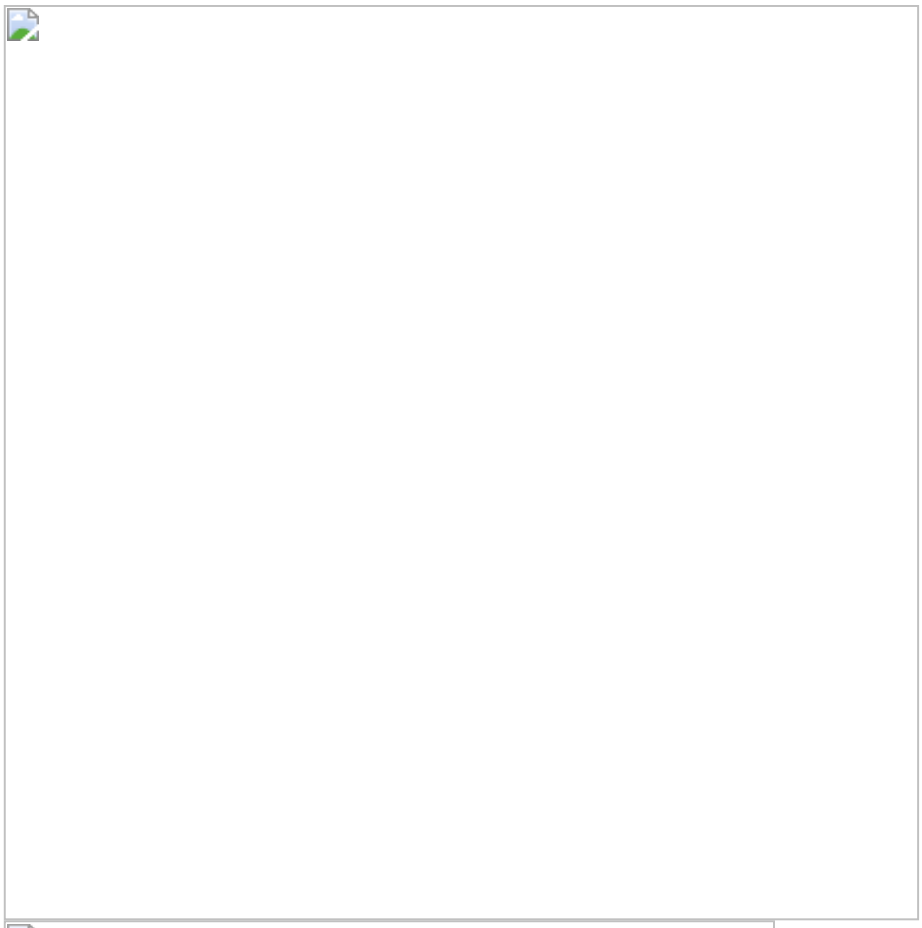


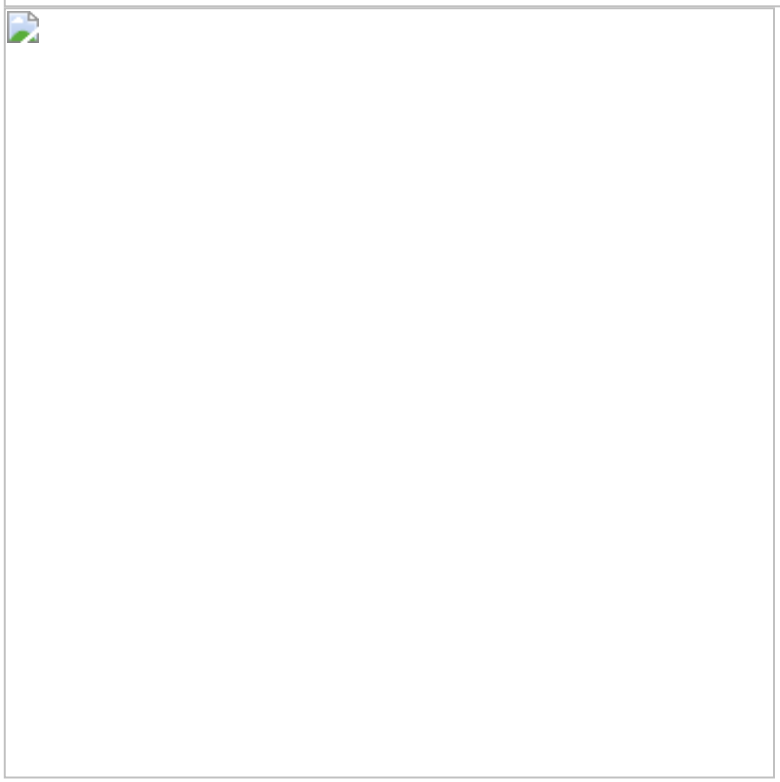
In addition, one other *BlackNix RAT* samples were mentioned by james_inthe_box[2]. Based on SSDEEP similarity, another sample was found.



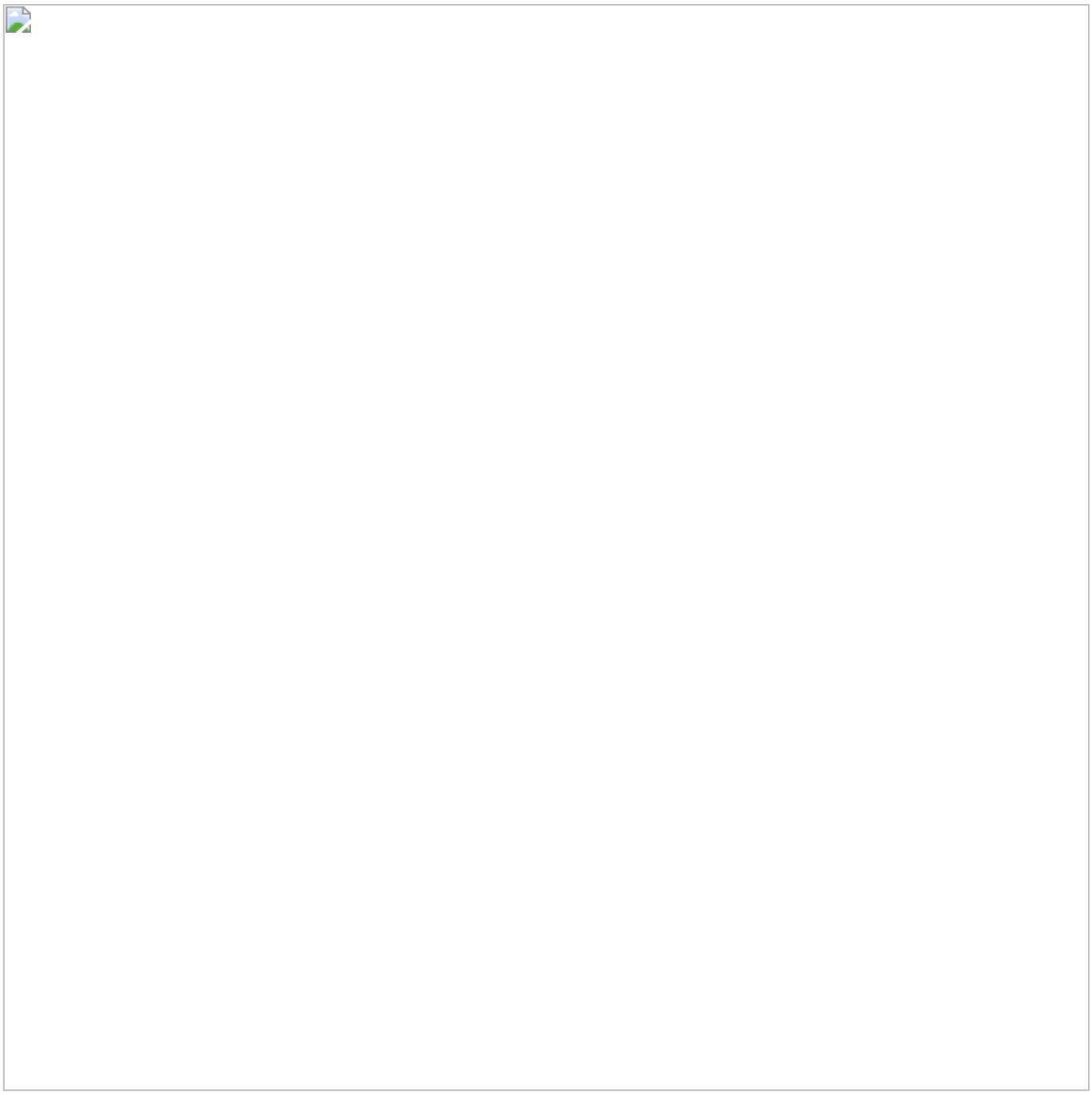
Next, we shall see if all these samples send data in the same structure as what we have analyzed previously. If they do, then perhaps these are all related in some way and are not "wild" *BlackNix RAT*s.

Set 1 binaries

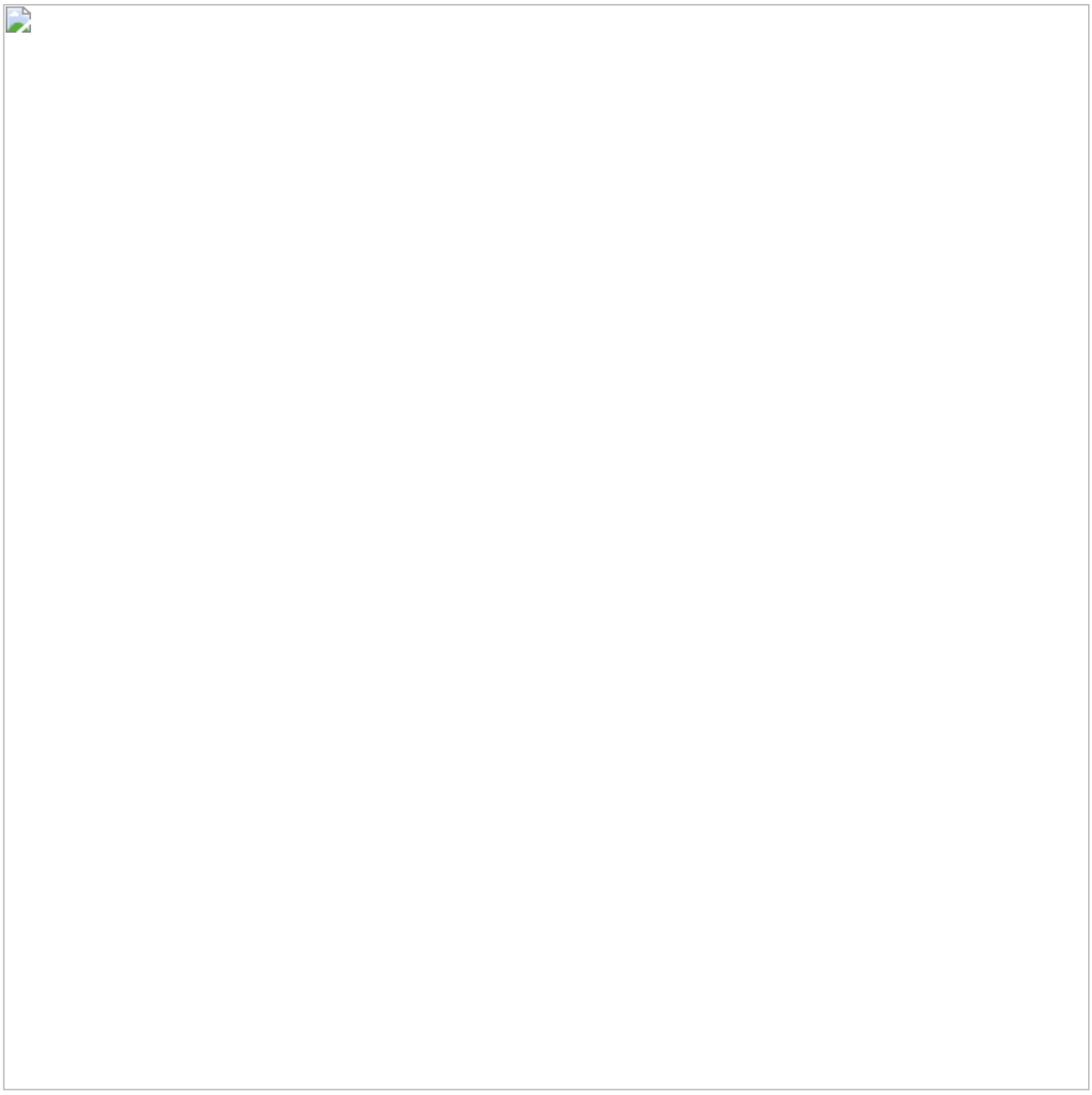




Execution of these binaries will result in errors. Upon closer look, the errors happen because some part of the binary seems to be corrupted. Whether this is a deliberate “disarm” attempt or due to a bug, I can’t tell.

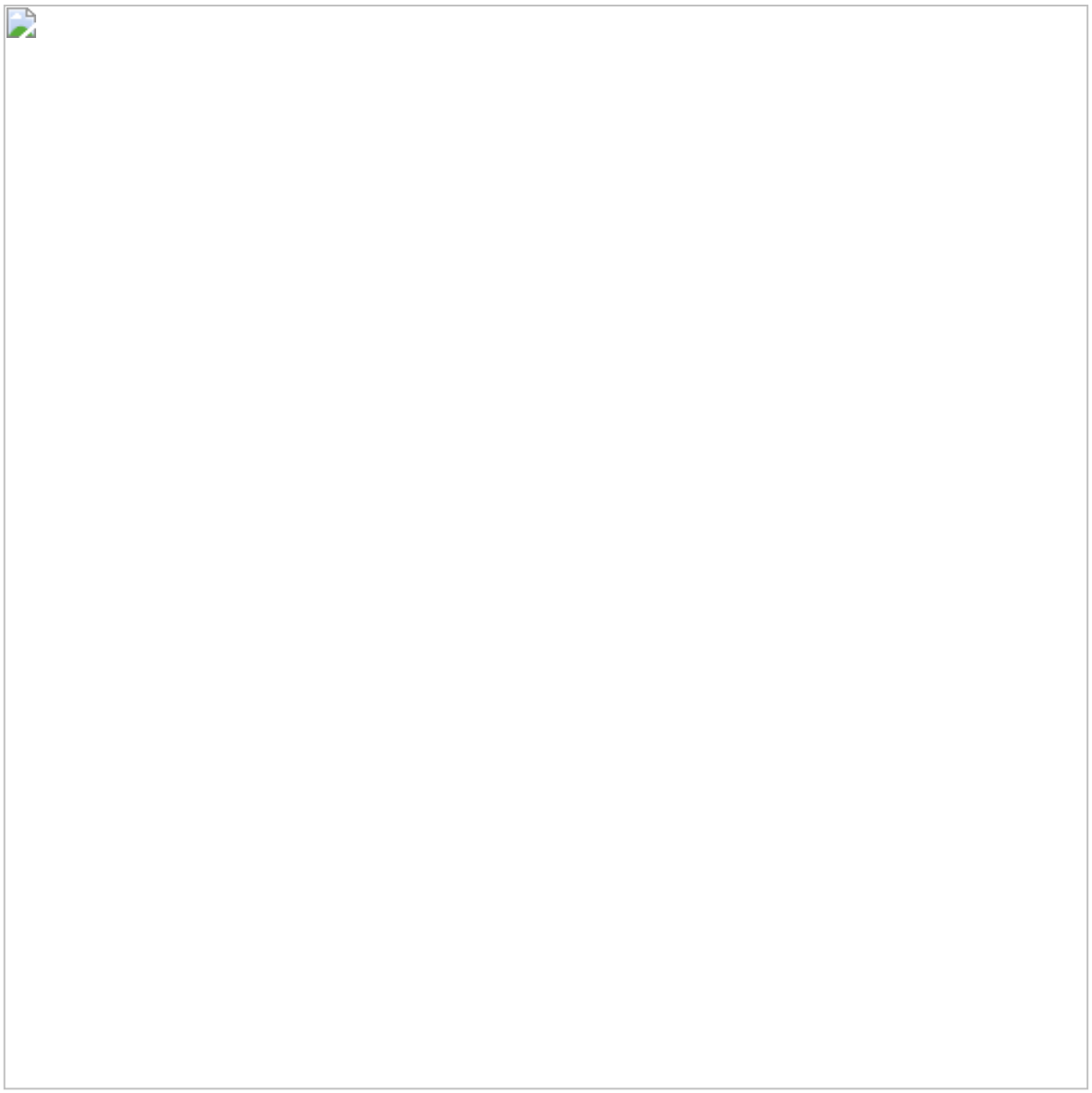


Cause of error at address 0x4CDE53

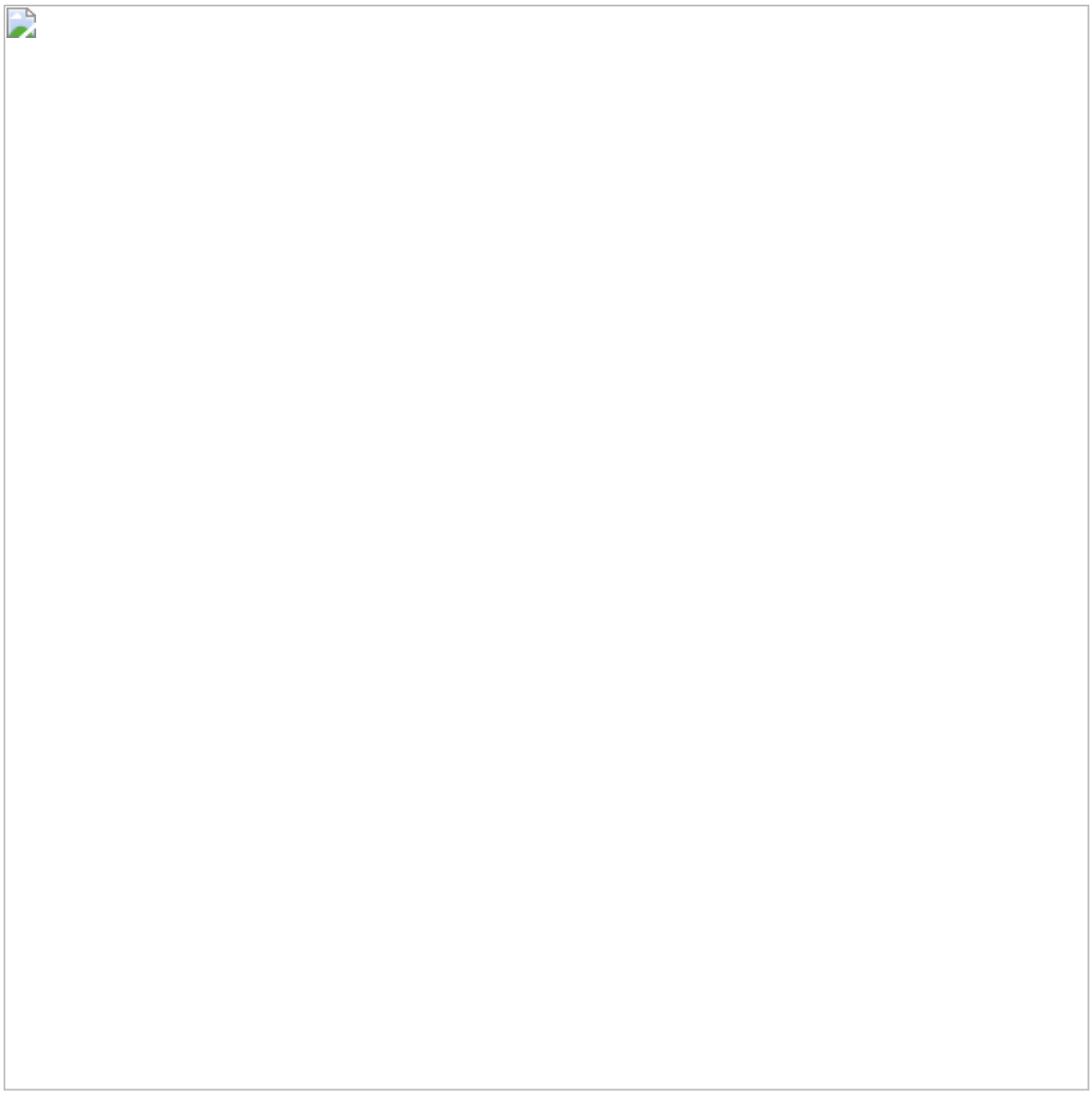


Cause of error at address 0x4CE084

Just patch the areas with the corresponding bytes from inte.exe to fix the problems. The following screenshots show highlighted areas after patching.

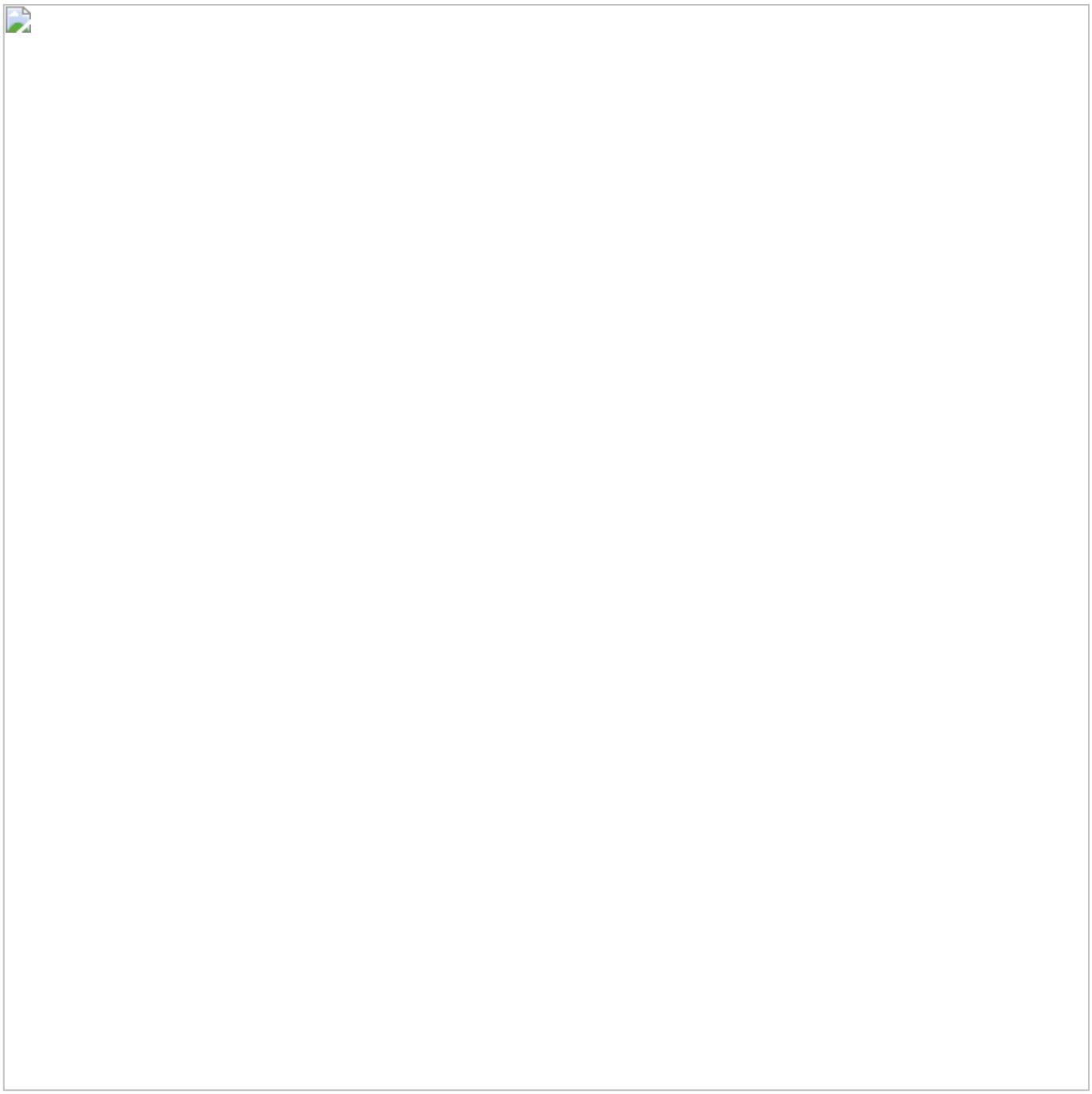


Patch to solve error at 0x4CDE53



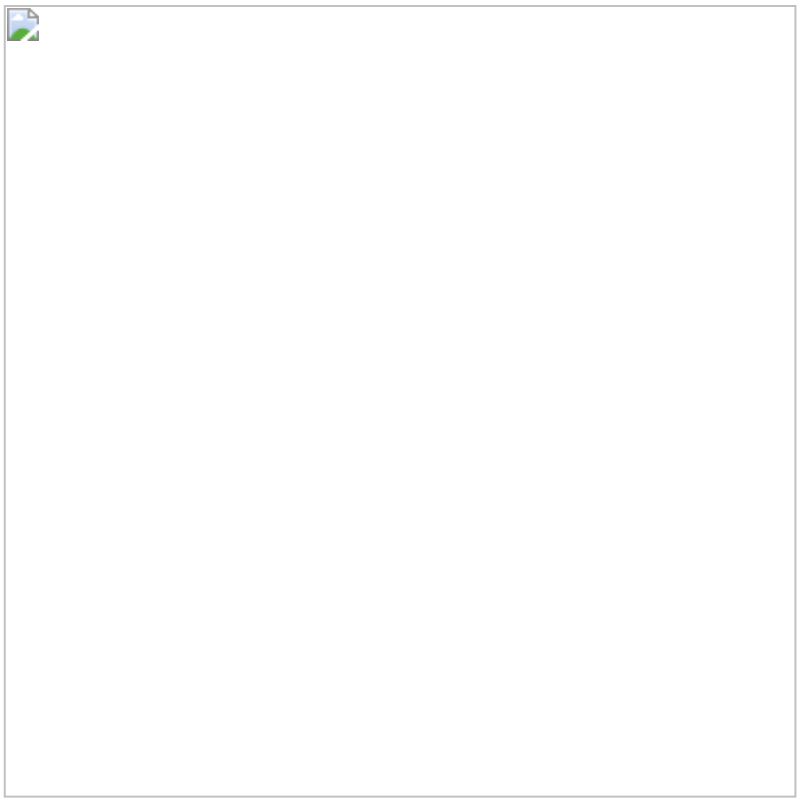
Patch to solve error at 0x4CE084

From the network packet, it looks like the data structures are identical to what we saw in inte.exe. This is verified with a comparison of the function that is responsible for building the structure. This is not surprising, as they all call back to the same IP address.

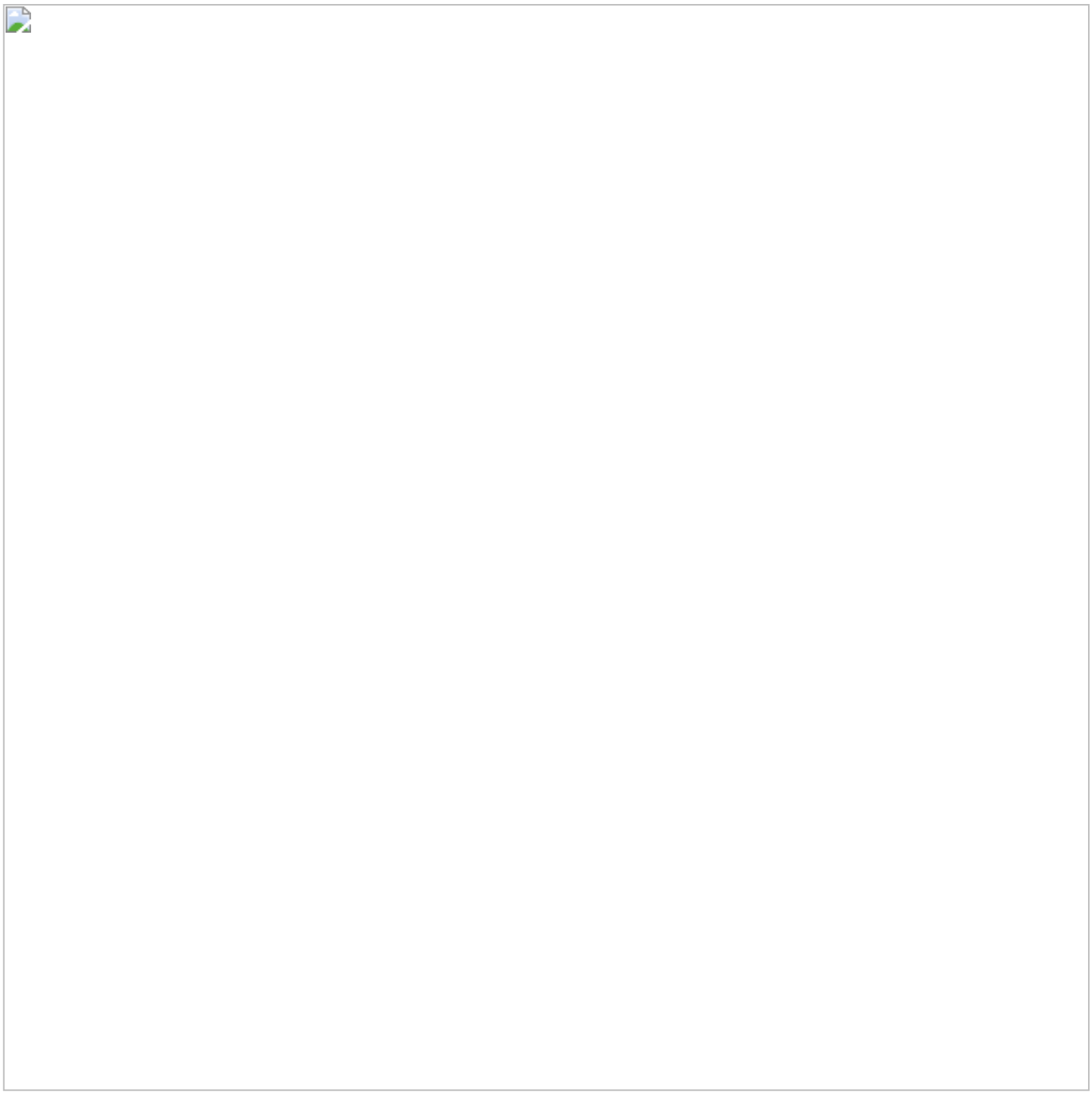


Set 2 binaries

These binaries are different, because they are VMProtect-packed, which means that I cannot simply throw them into IDA Pro and hope to do function comparisons.



Execution of these binaries require them to be executed with administrator privileges, as they will spawn a svchost.exe process for injection. Knowing this behaviour, we can dump the unpacked executable from memory at the moment where the injection happens. A breakpoint at ntdll.dll's NtWriteVirtualMemory will do the trick.

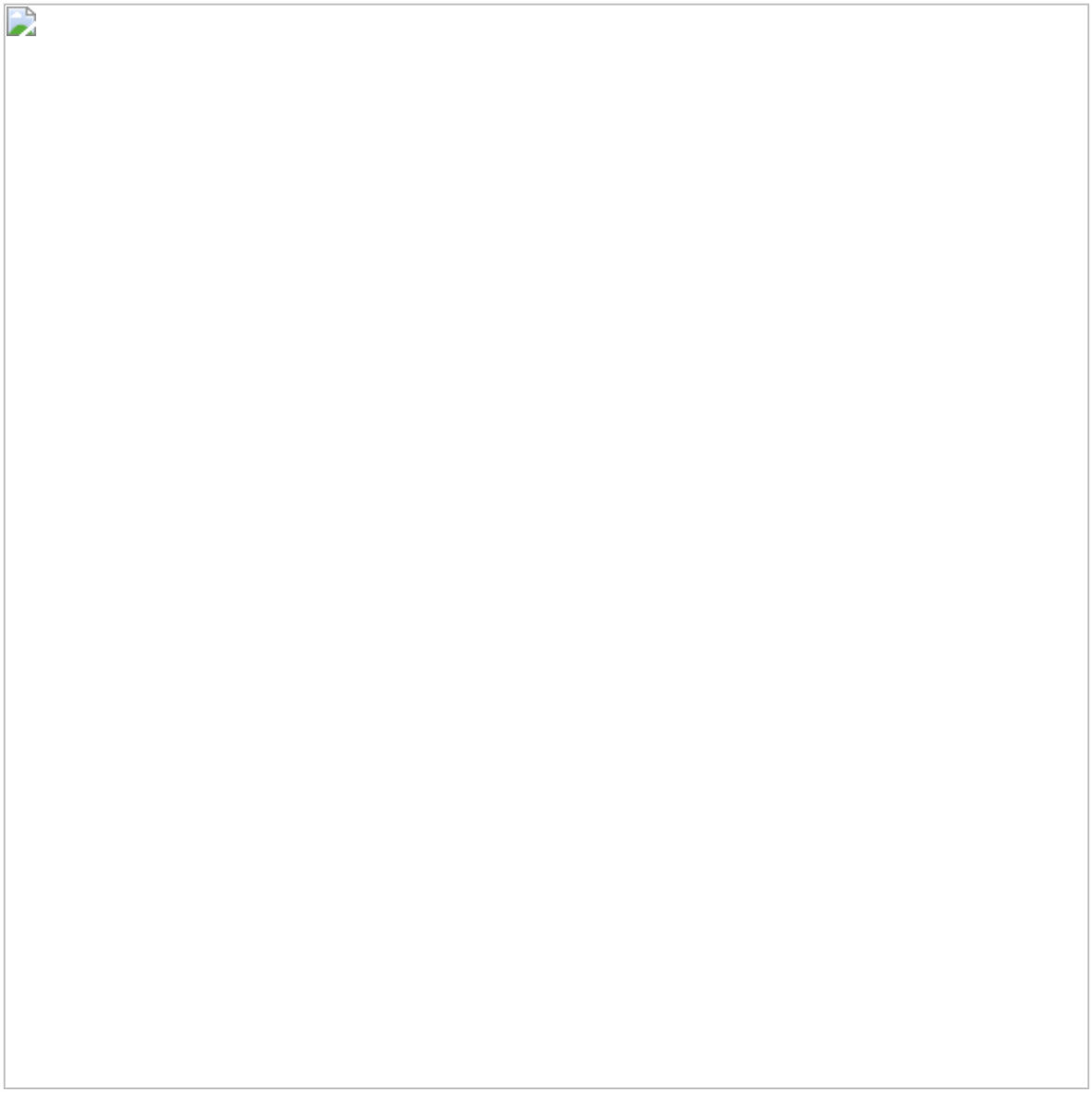


```
NtWriteVirtualMemory(
IN HANDLE ProcessHandle,
IN PVOID BaseAddress,
IN PVOID Buffer,
IN ULONG NumberOfBytesToWrite,
OUT PULONG NumberOfBytesWritten OPTIONAL );
```

The idea is to watch for a call to NtWriteVirtualMemory with a handle to svchost.exe, and let it run till all the sections have been copied. We would know it’s done when NtResumeThread is called.

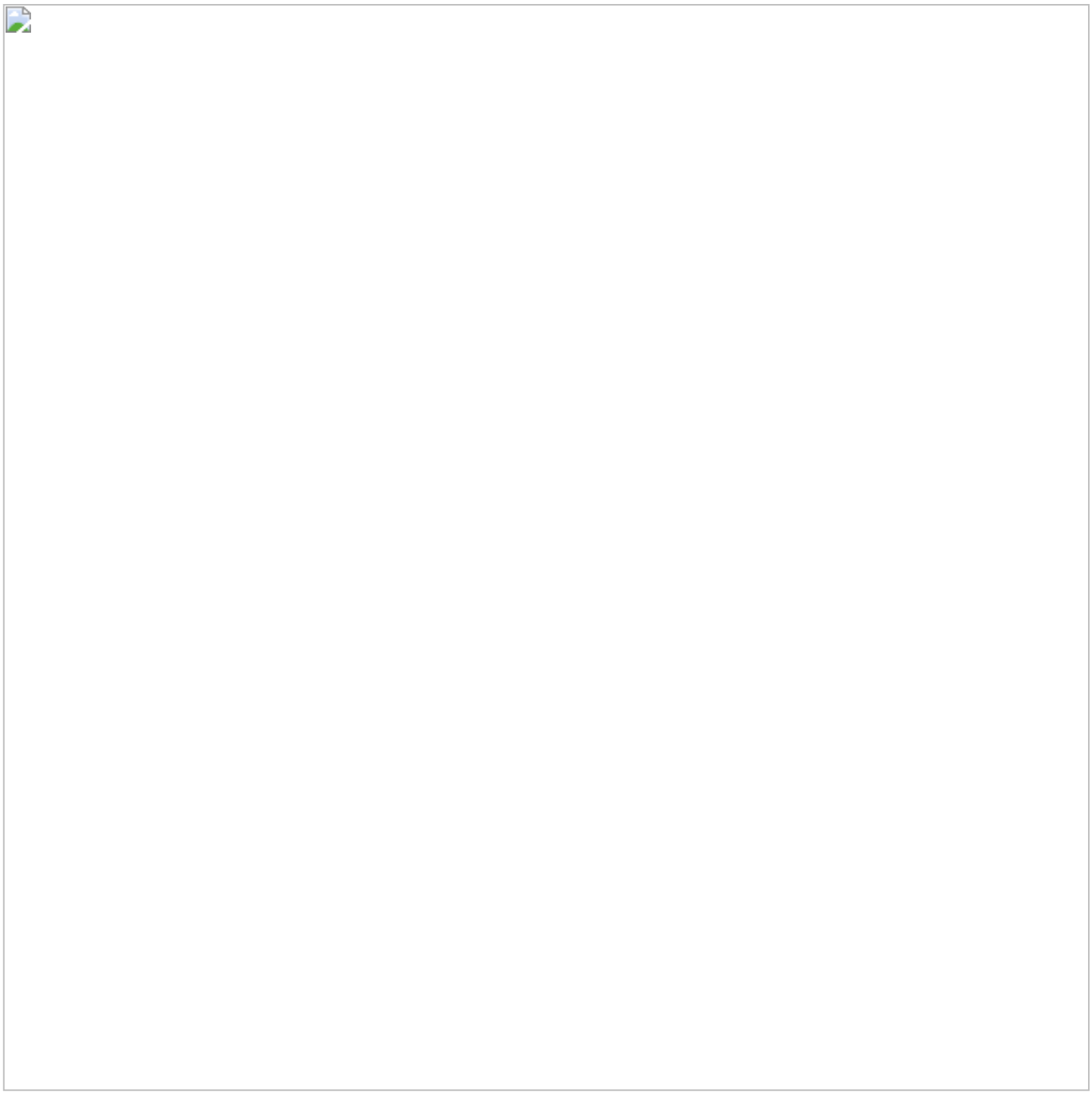
After dumping the executable from memory, we would have to fix the section headers' raw addresses before we can use IDA Pro to look at it. I've mentioned how to do this with CFF explorer in one of my earlier posts.

A quick look at strings within this dumped file reveals the tell-tale *BlackNix* strings:



The SETTINGS strings looks identical to what was seen in inte.exe, including the Chinese words 初始 and the server password 'root'. The function that is responsible for reading the SETTINGS strings and building the callback data structure is identical to inte.exe's as well (and hence the callback data structure is also the same).

I am certain that we are looking at the same variant of *BlackNix RAT* here.



So, they are all the same BlackNix variant. Now what?

This journey started from some unique mutexes found in a malware (one *BBSRAT*) that calls back to one of known **Winnti Group**'s infrastructure. The same set of mutexes, some overlaps in code logic (in the naming of files and lateral movement using RDP shared drives), as well as close time proximity in compilation timestamps, suggested relationship between that one *BBSRAT* and the set of *BlackNix RAT*s (Project1.exe).

In addition to the mutexes, I noticed other similarities in Project1.exe's execution and the *Trochilus RAT* dropper csres.exe described in Trend Micro's Uncovering DRBControl report[3], specifically in the names of the files and service created and path to malicious binary:

- system.exe
- SESSRV
- c:\ProgramData\Microsoft\DeviceSync

I get reminded of my earlier speculation that system.exe is a generic tool used to deliver/spread the payload (be it *BlackNix* or *Trochilus RAT*). I'll never know for sure till I get my hands on some more samples ;)

Last Words

Is **Winnti Group** also behind the set of *BlackNix RAT*s that were under scrutiny in this post? There might be a good chance this is true. However, one other interesting finding that I came across was that the C2 domain msdnsoft[.]lang32[.]com as well as the corresponding binary (SHA256: 873dfa94f924d59ceff4efb277fef5a251d7b648605c5239fc2ac0885ba32bd5) were linked to an adversary group named "Lang32" by QiAnXin Technology[4]. This adversary group is said to target victims in Southeast Asia. Perhaps I should look into the tools used by this group as well…

But that's a different long story for another time, that's it for now!

References:

[1]: http://www.hexacorn.com/blog/2014/12/05/the-not-so-boring-land-of-borland-executables-part-1/
[2]: https://twitter.com/james_inthe_box/status/1151972438692921344
[3]: "Operation DRBControl: Uncovering a Cyberespionage Campaign Targeting Gambling Companies in Southeast Asia", Trend Micro, 18 Feb 2020
[4]: https://www.secrss.com/articles/12463

~~

Asuna

The latest Tweets from Asuna (@AsunaAmawaka). [Malware Analyst]. Binary World

twitter.com

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