What happened between the BigBadWolf and the Tiger?

medium.com/insomniacs/what-happened-between-the-bigbadwolf-and-the-tiger-925549a105b2

asuna amawaka May 20, 2020

asuna amawaka

May 20, 2020

·

10 min read

While I was doing research for my previous posts, I came across mentions of a trending Chinese-language-based C2-side controller called 大灰狼 (pronounced as Da Hui Lang, which translates literally to Big Gray Wolf). I'm just going to call it BigBadWolf here :) Simply because the name is cute, I picked it up and took a closer look. Turns out, it is modelled after (or should I say, it's an edit of) the infamous Gh0stRAT, and samples that are built from the BigBadWolf matches Gh0stRAT signatures, as well as this YARA rule [1]:

```
rule IronTiger_Gh0stRAT_variant
{
meta:
author=”Cyber Safety Solutions, Trend Micro”
comment=”This is a detection for a s.exe variant seen in Op. Iron Tiger”

strings:
$mz=”MZ”
$str1=”Game Over Good Luck By Wind” nocase wide ascii
$str2=”ReleiceName” nocase wide ascii
$str3=”jingtisanmenxiachuanxiao.vbs” nocase wide ascii
$str4=”Winds Update” nocase wide ascii

condition:
$mz at 0 and (any of ($str*))
}
```

There are plenty of articles and analysis walkthroughs out there on Gh0stRATs, given its very long history. However, I decided to go ahead to further this exploration because I've seen this YARA rule hit often enough to wonder about whether the samples are really related to Iron Tiger, or could it be the case that the strings are no longer unique enough to identify any particular variant.

I'm sure that in your lifetime browsing VirusTotal, you would have come across community comments like this:

Detected by THOR APT Scanner

Detection
==========================
Rule: IronTiger_Gh0stRAT_variant
Ruleset: Iron Tiger
Description: This is a detection for a s.exe variant seen in Op. Iron Tiger
Reference: http://goo.gl/T5fSJC
Author: Cyber Safety Solutions, Trend Micro
Score: -

I was not able to get my hands on the exact sample that this rule was based on but I did find a few other samples that contains those strings, and I picked 3 to do comparisons with binaries generated from the BigBadWolf builder:

- BBE7D708310EC7E5F981CE4BA9928A19C4D2169B5520FFA573085F9698F90C25
- C02A360C6F64609403B4E4D4FC130014C40EBB77F71DF816C6408851C7C9ED54
- 9DCDDC7FFCE78526057888B43B57E76BA7F3FED0C13FB4FA4214DCB08412C447

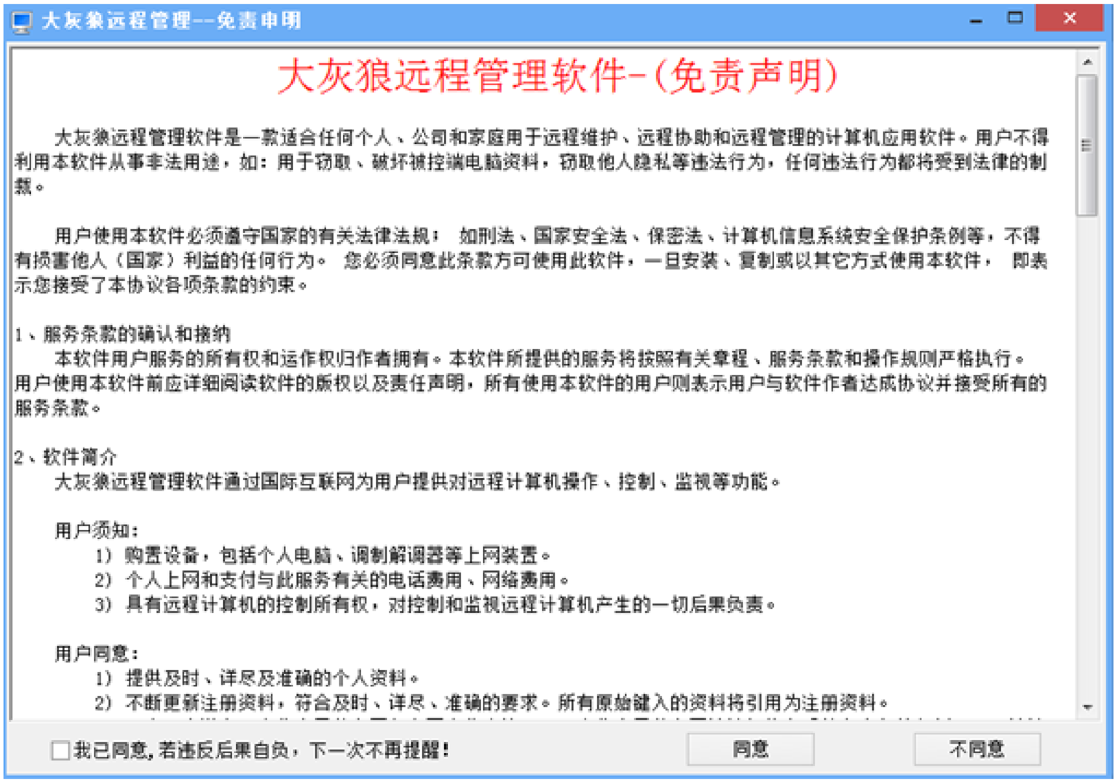
While I was preparing this post, I came across a tweet[2] from malwaremustd1e that mentioned a “KuGou” backdoor along with screenshots that looked somewhat like what I observed while exploring BigBadWolf. I added these files as part of my comparison attempts, later in this post.

- 852FA14860260023289EE6577DBD5E0193DF31DAE5F3C078142D3CAC030C7462 (EXE dropper)
- 7BAEE22C9834BEF64F0C1B7F5988D9717855942D87C82F019606D07589BC51A9 (DLL RAT)

Let’s get started!

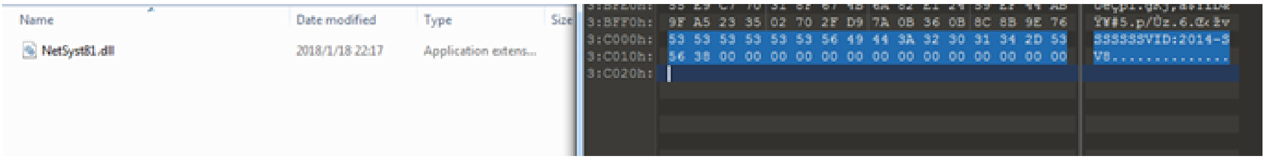
The Misunderstood Wolf?

Maybe it all started as a tool for education. Really. It even came with a warning against doing evil with this toolkit. Although ticking that checkbox at the bottom of the disclaimer does felt a little like “I solemnly swear I am up to no good”.

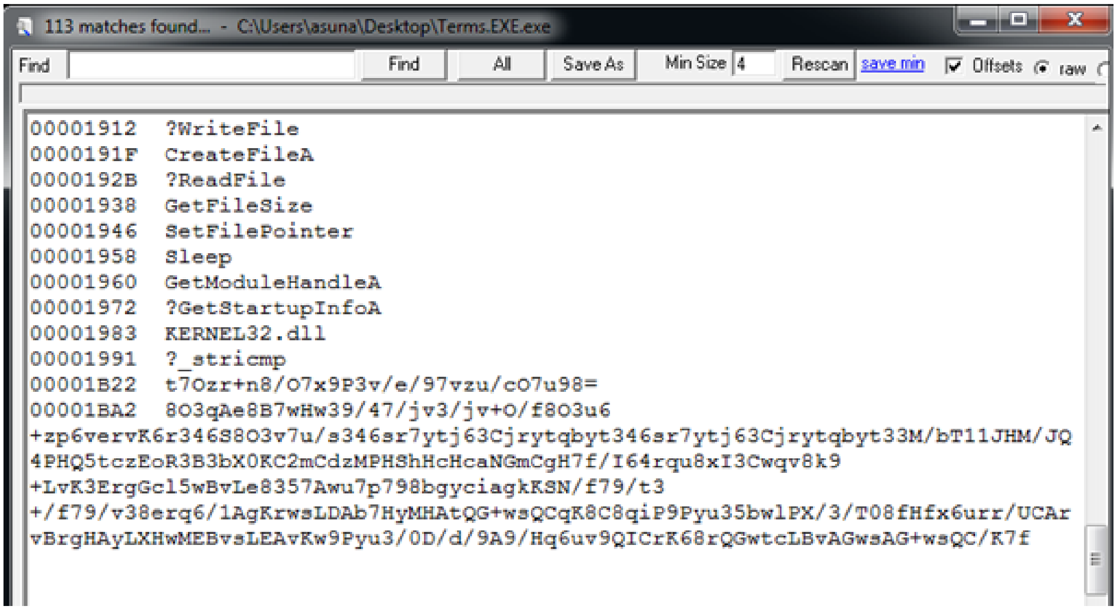


The builder component comes with the standard set of features e.g. specify the C2 IP address, mutex, name of the service to create for persistency, location to store the malicious binary on disk, options to delete the binary upon single run. There seems to be another binary ("1.dll" shown in the screen capture) that needs to be downloaded by the generated binary. Leaving this field blank causes the build to fail. This is quite typical of a Gh0stRAT deployment — a simple dropper/loader and a DLL that contains the main logic of the RAT.

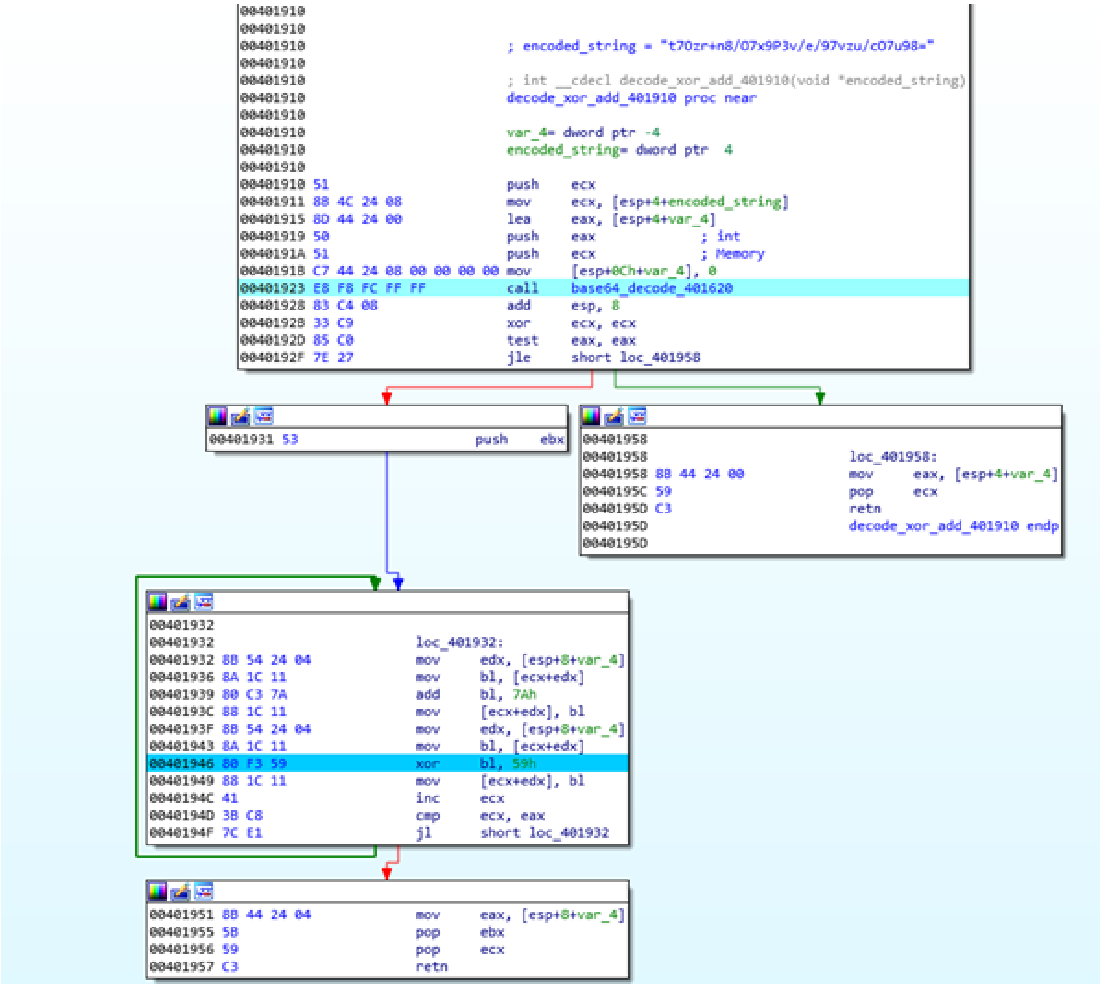
I found a copy of the required DLL file that came within the bundle of C2-side binaries, and it looks to be encoded/encrypted. The last 32 bytes of the file looks like a marker of sorts. Make a mental note of this, we'll see how this is used later.



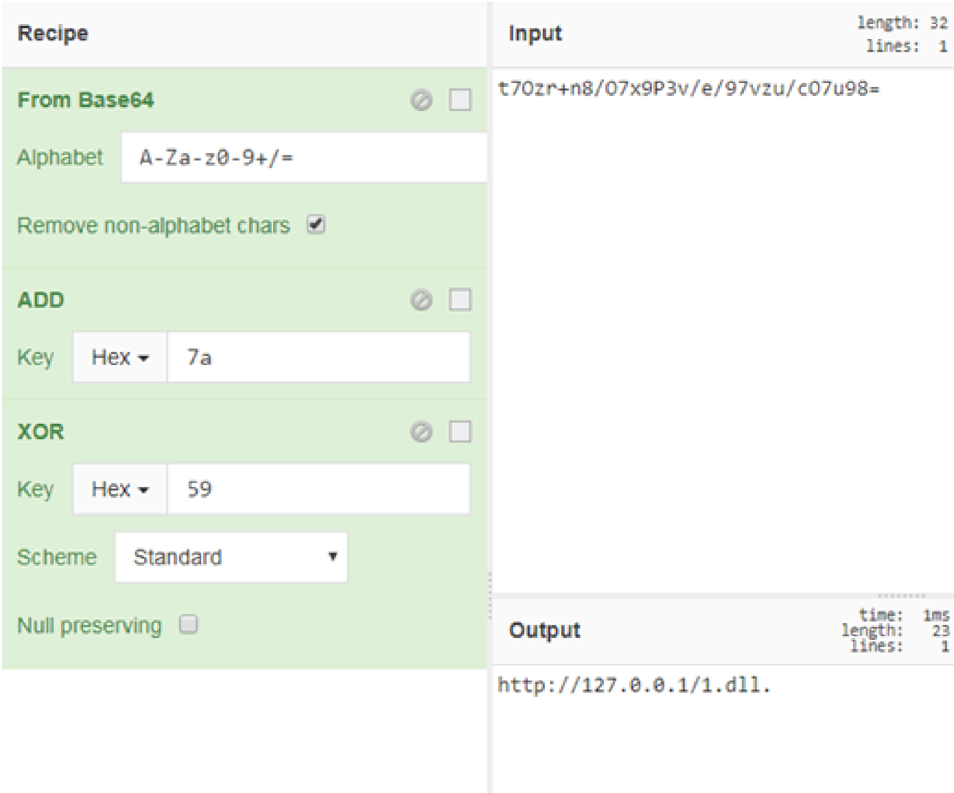
The output of the builder is a rather lightweight (9.5KB) EXE file, with almost no strings to analyze. Thankfully, there is still something to hint of "evilness" within this executable — two sets of base64 encoded strings.



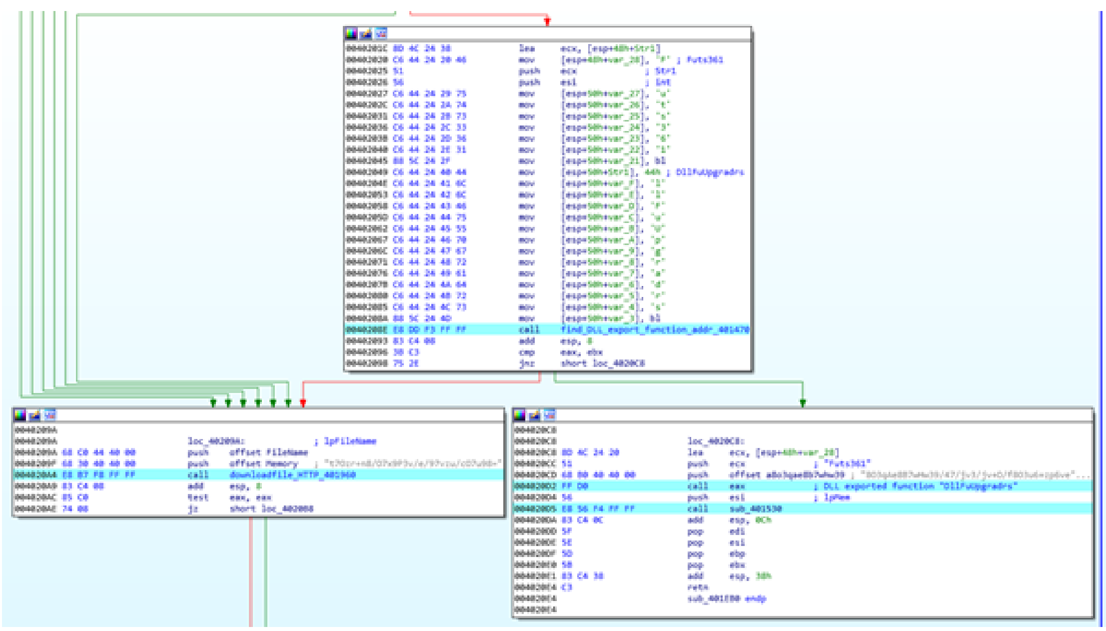
The use of these strings are very quickly found within the binary.



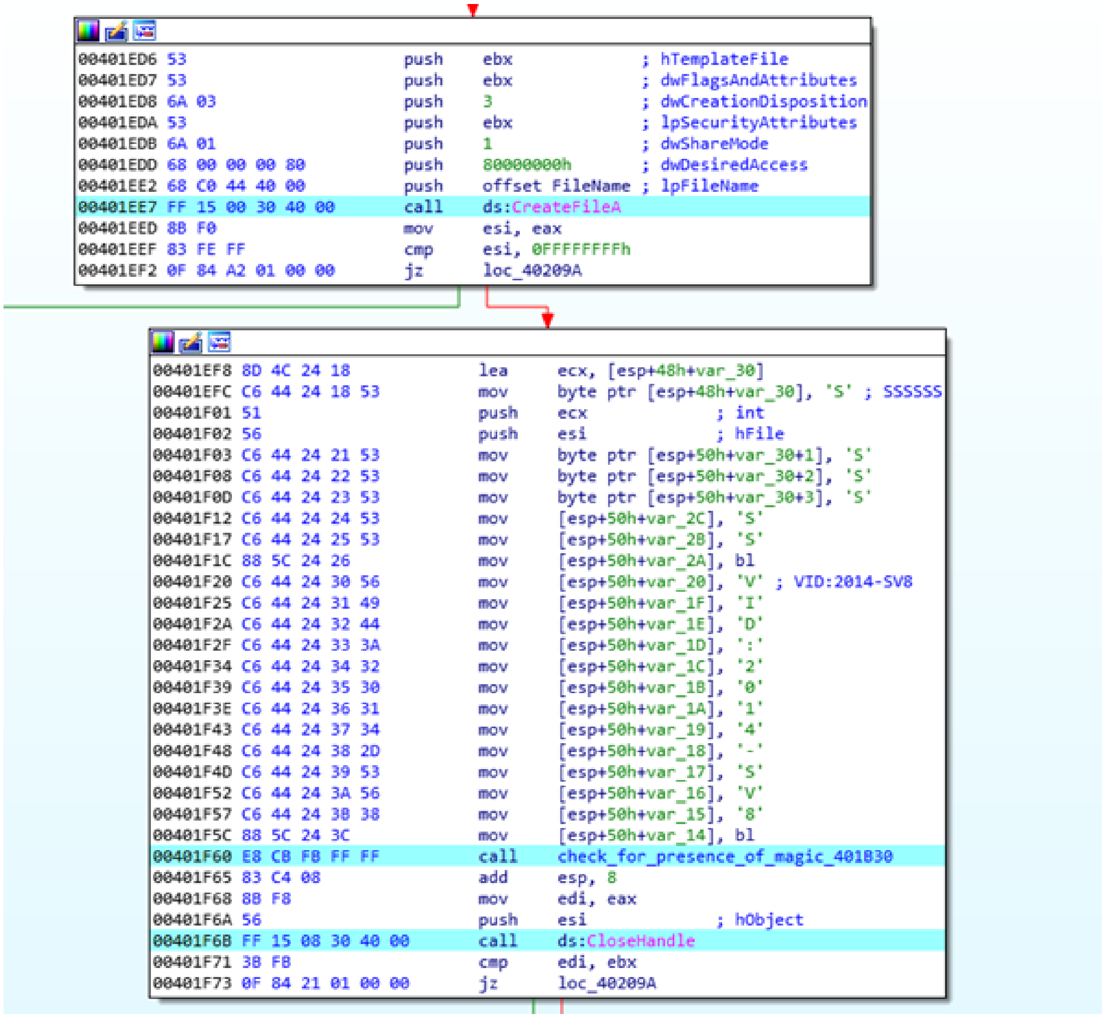
The first set of string can be decoded with base64, ADD 0x7a and finally a XOR 0x59. This gives us the address to fetch the DLL that we specified in the builder. ADD and XOR operations are typical encoding seen in Gh0stRAT variants.



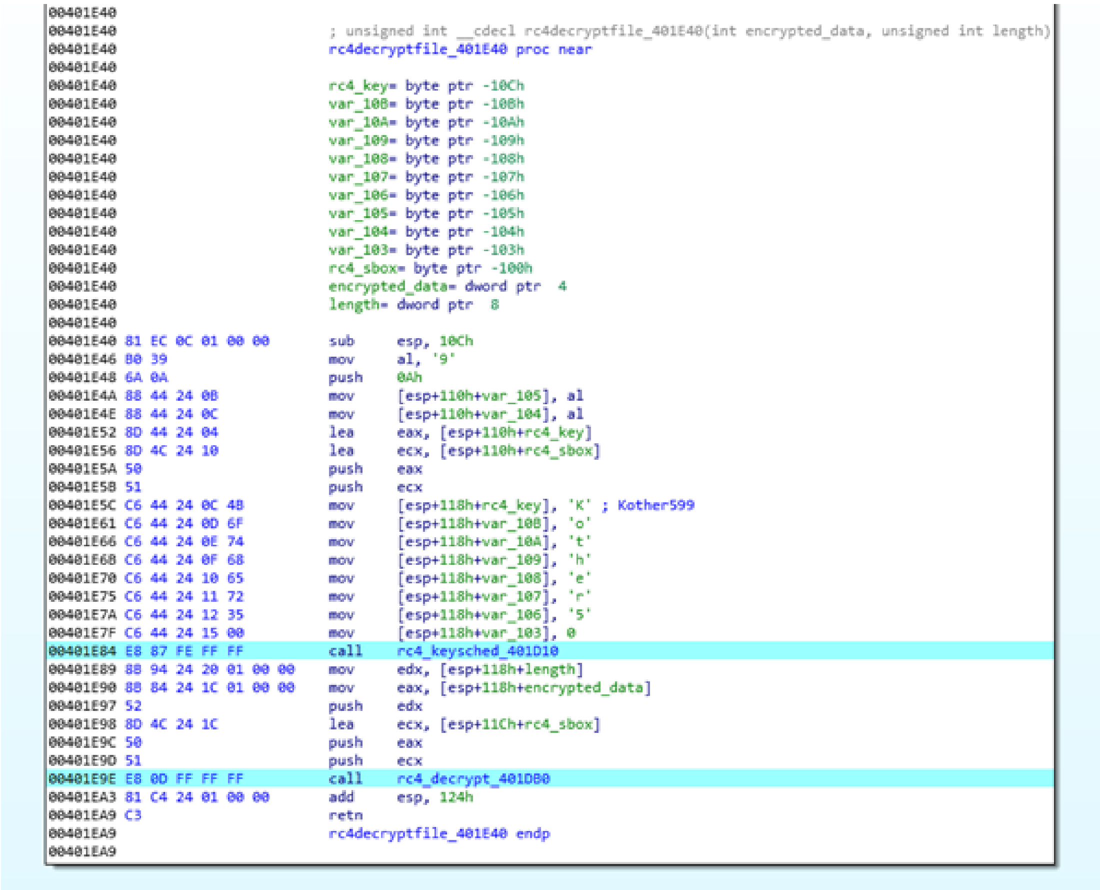
The binary then proceeds to download this DLL and store it in C:\Program Files\AppPatch. This path is not configurable within the builder. As said earlier, the DLL is the meat of the RAT — all the EXE does is to download it, decrypt it, execute it and load the configuration data into its memory. Speaking of configuration data, that happens to be the second set of encoded strings we saw. The decoding of that set of string is the responsibility of the DLL. We'll look at that in awhile.



Let's talk about the DLL. After receiving the DLL, the loader checks for the magic footer before proceeding to decrypt it.



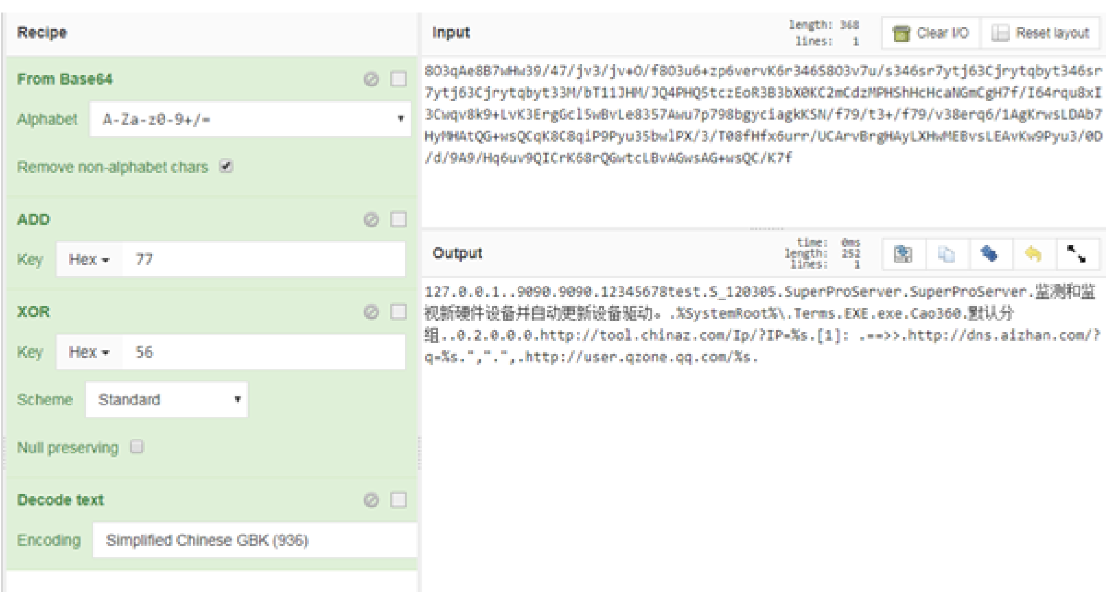
The decryption algorithm is nothing fanciful, just RC4, where the key is “Kother599”. One more thing that we have to do before we can analyze this DLL with a disassembler: unpack it with ‘upx -d’.



Your big ears are showing, grandma…

The first thing that the DLL is tasked to do is to decode the configuration data. Most of this configuration data is set in the builder, while some appears to be hardcoded.

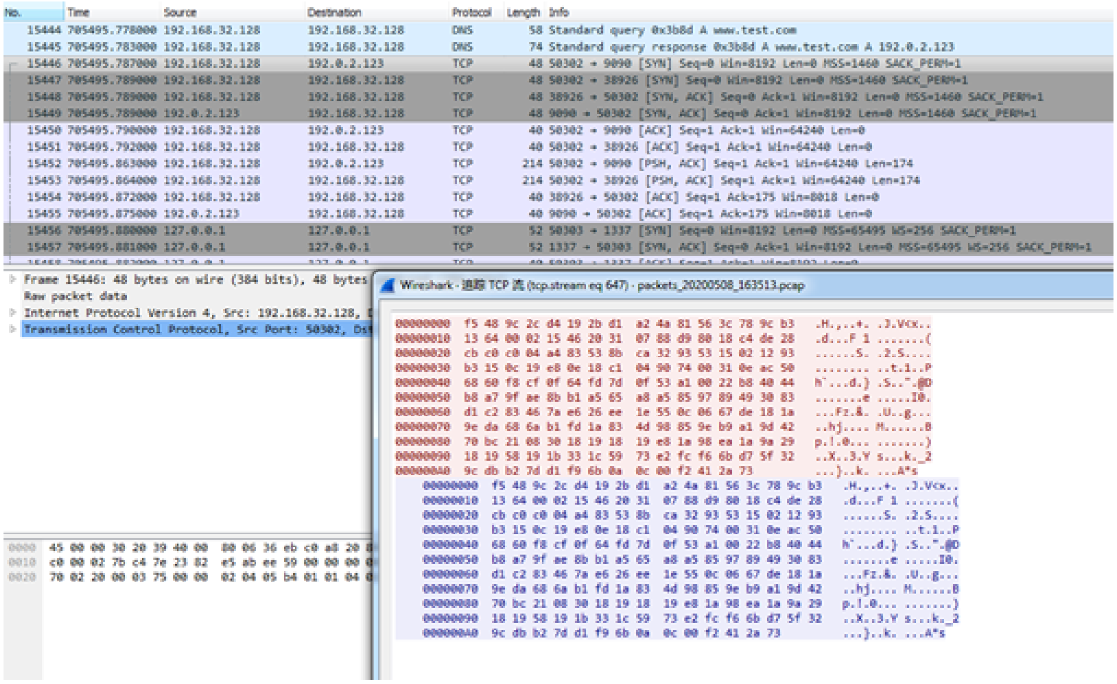
The decoding of the configuration is the same sequence (but using different hex values) seen above: base64, ADD 0x77, XOR 0x56.



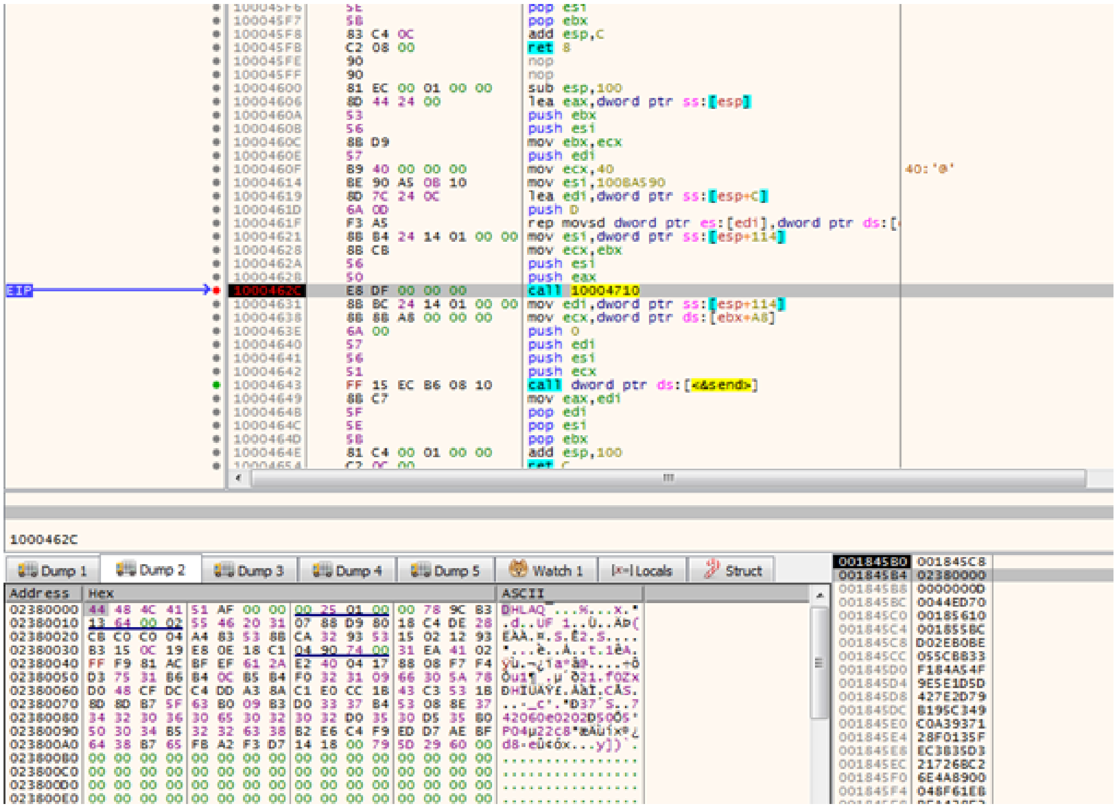
The structure is as such:

[C2 Address][QQ User ID][C2 Port 1][C2 Port 2][RC4 Password][Version][Service Name][Service Display Name][Service Description][Installation Path][Filename][Mutex][Group Option][Additional Download][Installation Type, Logging Options][IP address tool][placeholder string][reverse DNS tool][placeholder strings][QQ profile URL]

Now we come to the interesting part — the callbacks. As we know, Gh0stRATs have their signature 5-byte magic headers (the length varies in some cases, I know), followed by some size information, and finally the Zlib compressed data. However, I don’t see this structure in the traffic. What I do see is a Zlib header magic 0x78 9C. Let’s see what happened to the first few bytes prior to this Zlib header.



It’s not hard to identify the part that performs the encryption (RC4 again) of the communicated data. However, the author made a choice not to encrypt the entire data, but only the header portion, consisting of the 5byte magic, size of entire data, size of uncompressed payload, a total of 0xD bytes. This is done perhaps in a (futile) attempt to evade standard network signatures used to identify Gh0stRAT communications. However, since the length of the header remains the same after encryption, a slight tweak to such network signatures should suffice to work. The key used in the encryption is found within the configuration data earlier read by the binary. This key is made up of <user defined password within builder> appended with <username used to login to the C2>.

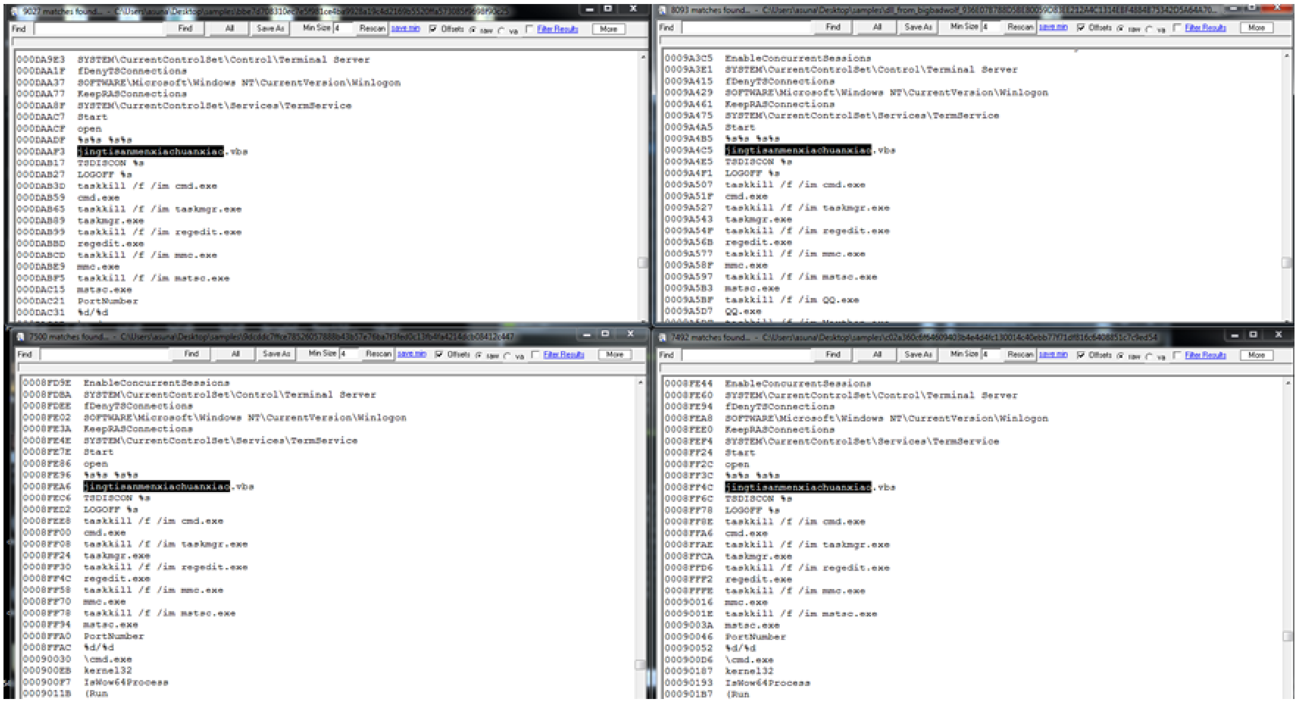


And I huff and I puff, to clear the mysterious fog surrounding these samples!

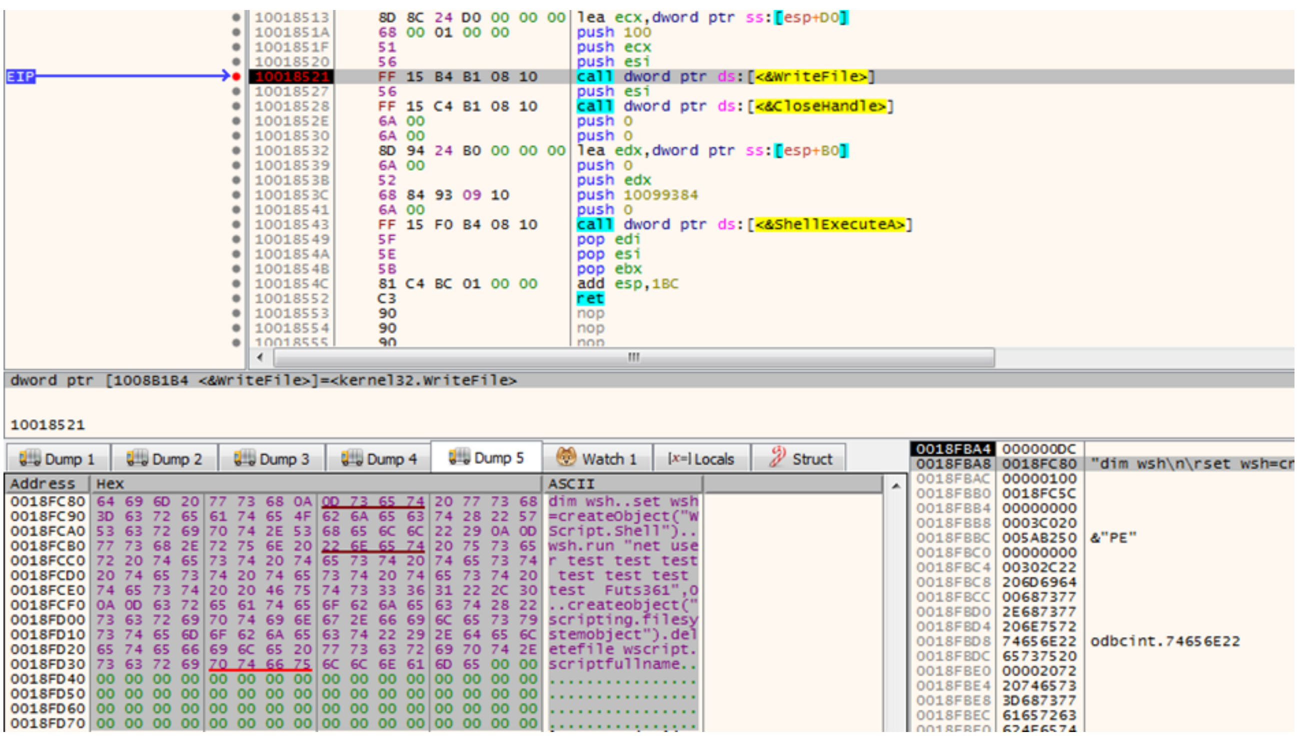
So what made these samples get flagged with that YARA rule I mentioned in the beginning of this post?

```
C:\Users\asuna\Desktop\samples
λ yara64.exe rule.yar .
IronTiger_Gh0stRAT_variant .\9dcddc7ffce78526057888b43b57e76ba7f3fed0c13fb4fa4214dcb08412c447
IronTiger_Gh0stRAT_variant .\bbe7d708310ec7e5f981ce4ba9928a19c4d2169b5520ffa573085f9698f90c25
IronTiger_Gh0stRAT_variant .\c02a360c6f64609403b4e4d4fc130014c40ebb77f71df816c6408851c7c9ed54
IronTiger_Gh0stRAT_variant .\dll_from_bigbadwolf_936E07B788D5BE80059D83EE212A4C1314EBF4884B75342D5A64A70D20BDED86
```

The presence of this strange VBS name:



What does this VBS do? I gave the function some mock data as arguments, and the contents of the VBS is formed as follows. Looks like it is just for creating or manipulating a user account with “net user”. The name of the VBS is not related to its contents. Just for fun, I’m guessing “jingtisanmenxiachuanxiao” is written as 警惕三门峡传销 in Chinese, which literally translate to “Be wary of Sanmenxia MLM”. Strange name to give to a script in any case.



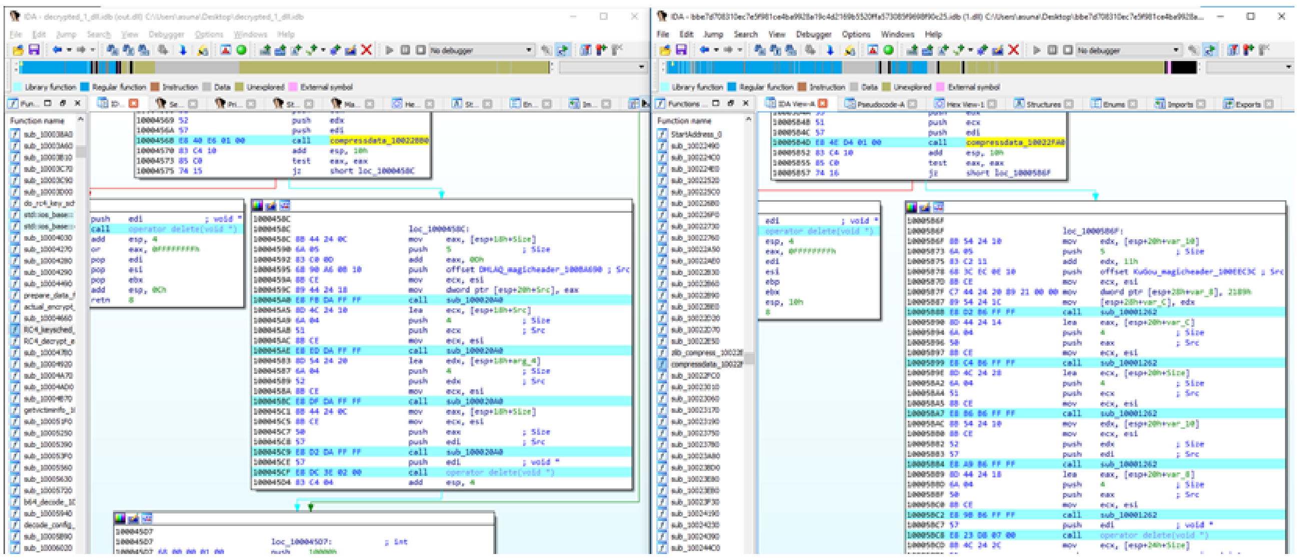
The exact same function is found in all 4 files I cross-examined.

Now, let’s find out if they are all BigBadWolf related.

The laziest way to start is to do BinDiff on the 3 files in relation to the DLL related to BigBadWolf. Results from BinDiff, pretty high scores. Not surprising, since they all stemmed from Gh0st code.

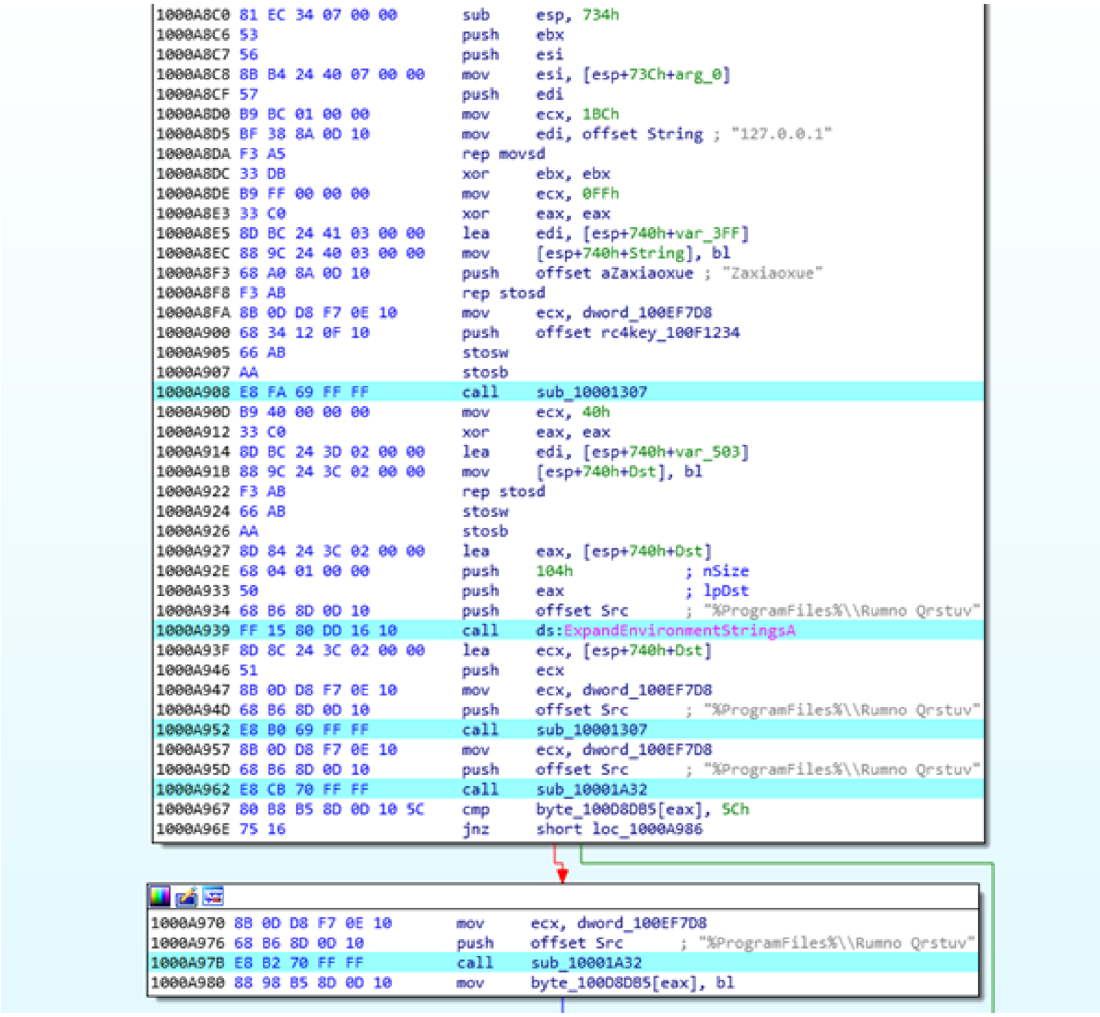
- Similarity with C02A360C6F64609403B4E4D4FC130014C40EBB77F71DF816C6408851C7C9ED54Confidence 0.988735 | Similarity 0.886978
- Similarity with BBE7D708310EC7E5F981CE4BA9928A19C4D2169B5520FFA573085F9698F90C25Confidence 0.984084 | Similarity 0.767249
- Similarity with 9DCDDC7FFCE78526057888B43B57E76BA7F3FED0C13FB4FA4214DCB08412C447Confidence 0.988665 | Similarity 0.879644

What are the differences then? Looks like all of the 3 has a different magic header — "KuGou", while the binary from BigBadWolf has "DHLAQ" as the magic (if you didn't notice, DHL is the acronym of its Chinese name Da Hui Lang). The size of the RC4 encrypted header also differs.



Left: from BigBadWolf; Right: from BBE7D708310EC7E5F981CE4BA9928A19C4D2169B5520FFA573085F9698F90C25

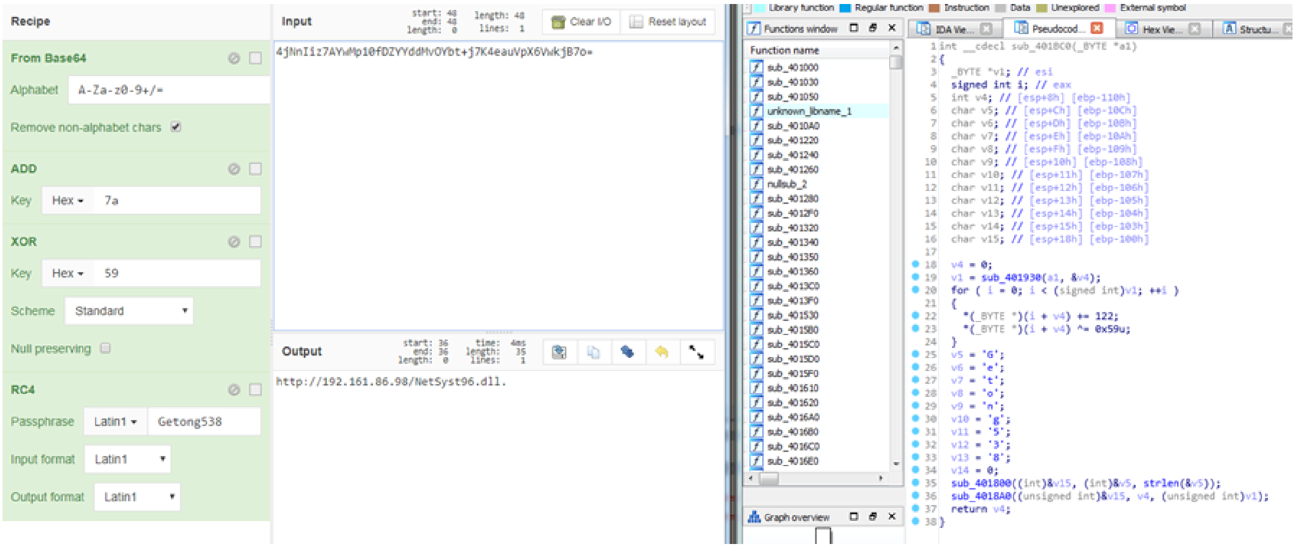
Another obvious difference is that the configuration data is not given as an encoded input, but instead found as plaintext strings handled directly within the functions.



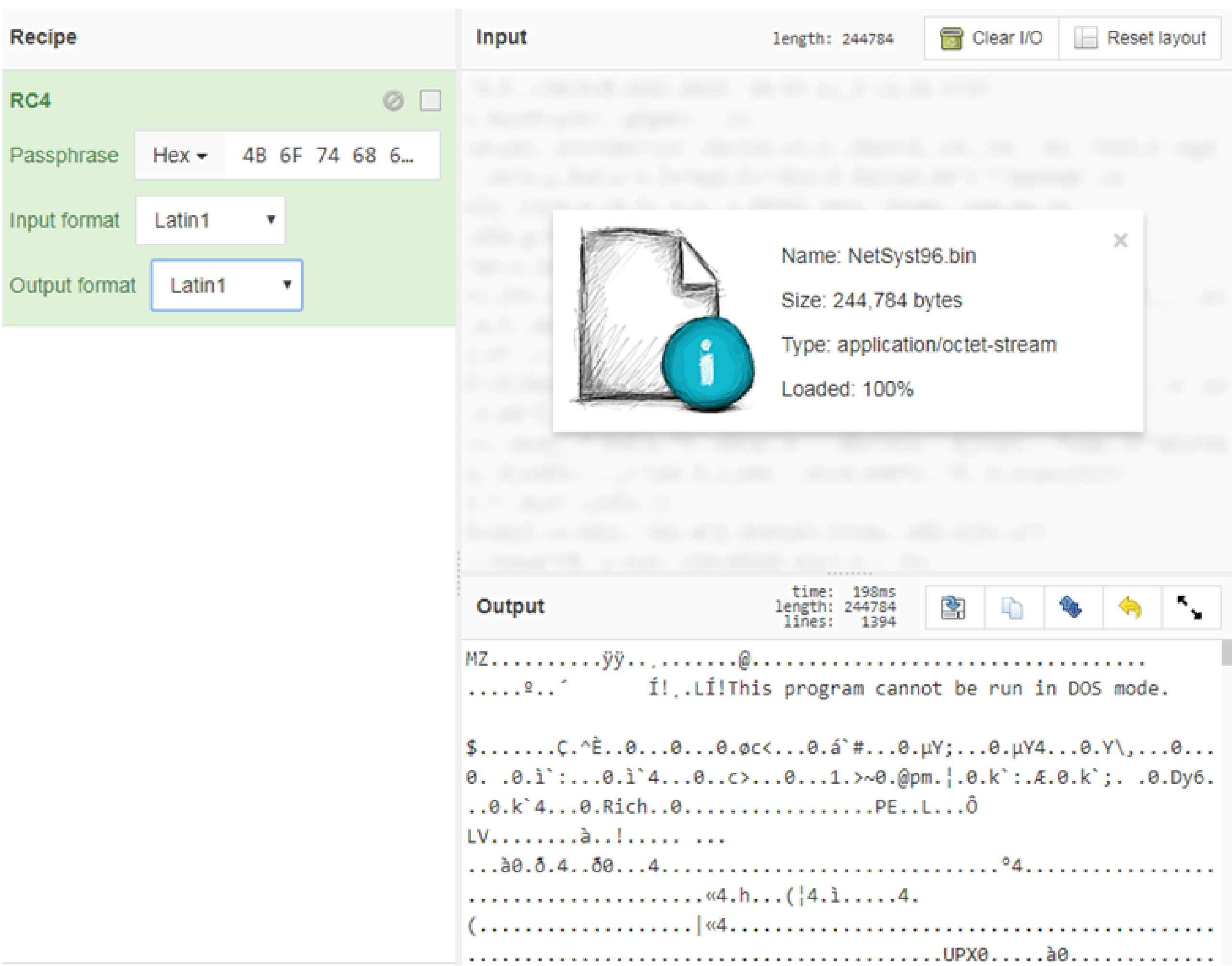
So, there is another Gh0st variant out there similar to BigBadWolf but yet implemented differently in some ways, let's call this set "KuGou".

Remember at the start of this post, I mentioned some KuGou malware tweeted by malwaremustd1e? Let's see if they are the same as the 3 KuGou binaries we saw above.

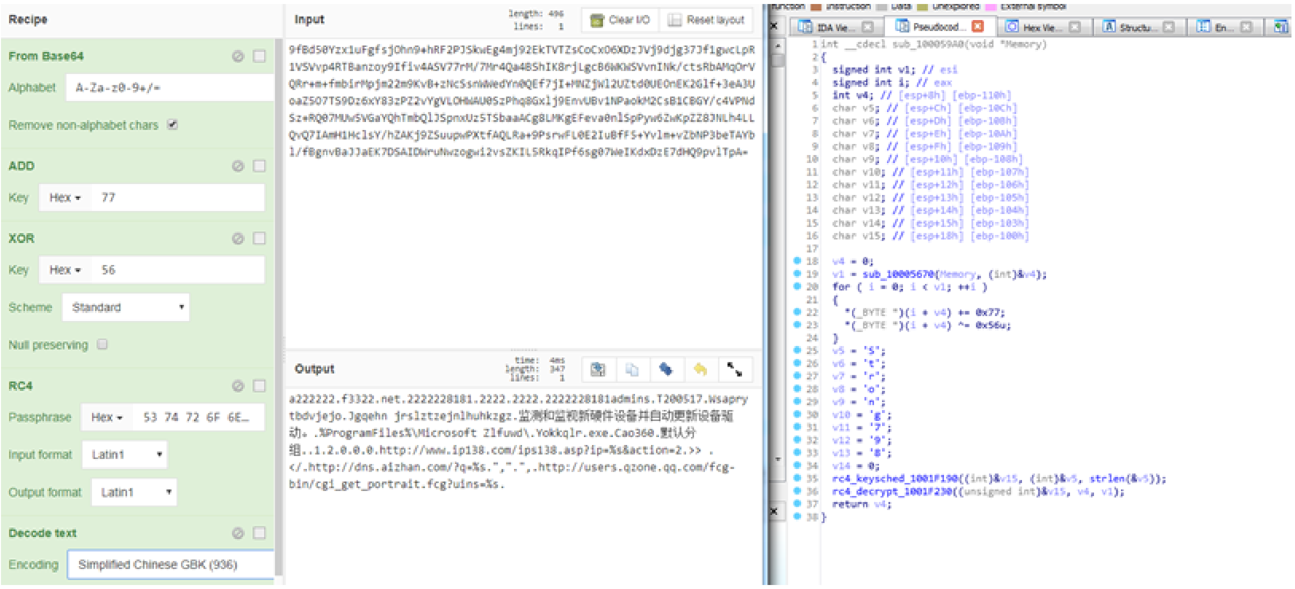
The dropper EXE (SHA256: 852FA14860260023289EE6577DBD5E0193DF31DAE5F3C078142D3CAC030C7462) contains encoded string that points the binary to download its DLL payload. Familiar yes?



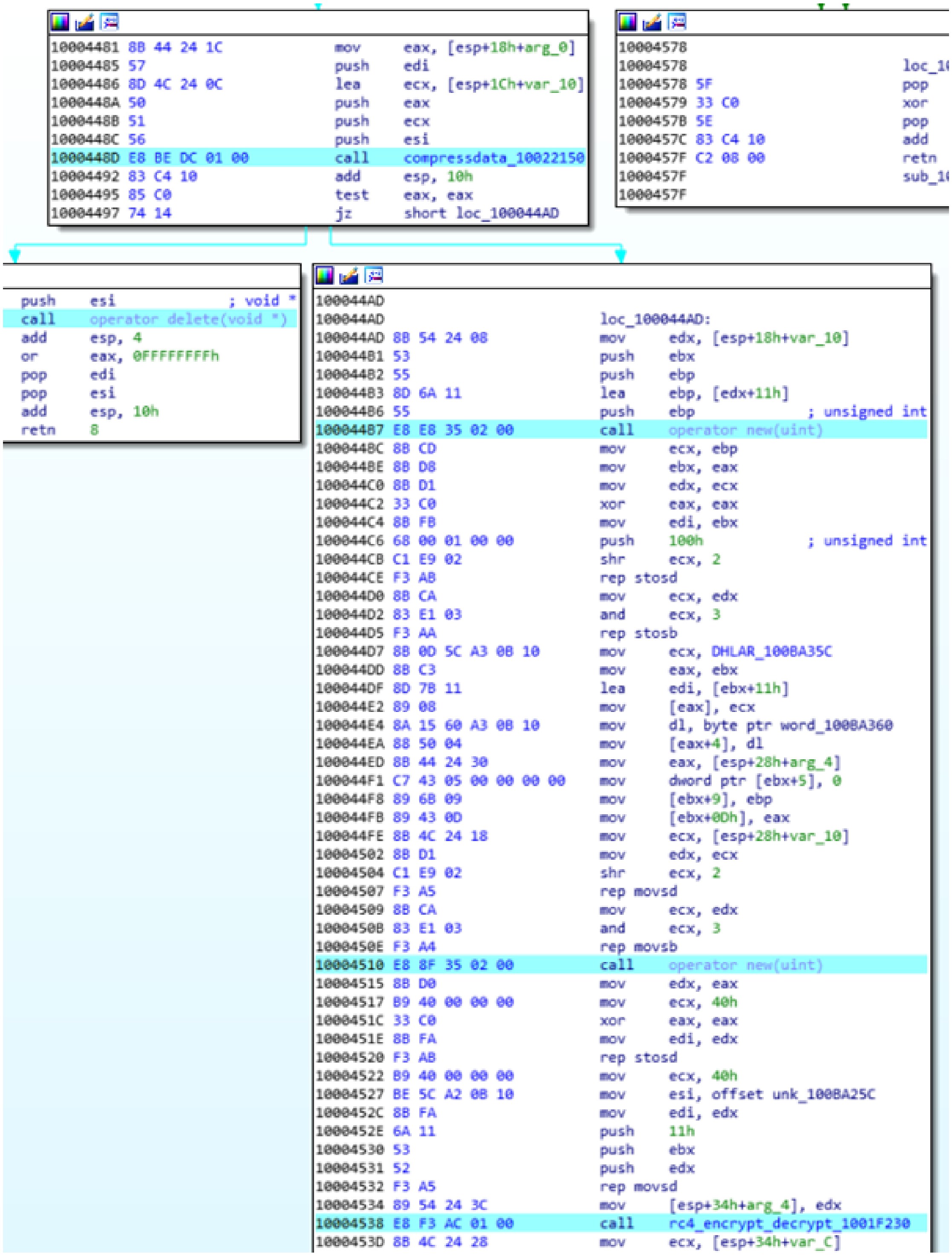
The downloaded DLL (SHA256: 7BAEE22C9834BEF64F0C1B7F5988D9717855942D87C82F019606D07589BC51A9) is RC4-decrypted with key “Kother599”. Again, familiar! There’s a slight difference here, the EXE did not verify that the file has a footer signature e.g. “SSSSSSVID:2014-SV8”, and the DLL does not contain such a footer.



The next difference lies in the configuration data passed to be decrypted by the DLL. In this binary, the configuration is encrypted with RC4, and not just Base64/ADD/XOR encoded as seen from the BigBadWolf's DLL. RC4 key used here is "Strong798". Notice how the structure of the configuration after decryption is identical to what we saw in BigBadWolf. And even that string of Chinese (监测和监视新硬件设备并自动更新设备驱动) used as Service Description is identical.



Since the configuration data is encrypted in a different manner, there must be another server-side binary responsible for building this sample. To my surprise, the 5-byte magic used in the communications is "DHLAR". Perhaps this explains the similarities shared with our BigBadWolf sample. Another thing is for sure, this file does not belong to the same set as the 3 "KuGou" binaries we just looked at. If I had to pin a family name to this file, it would be BigBadWolf.



A search on Google pointed me to a Operation PZCHAO report by BitDefender[3], in which a jingtisanmenxiachuanxiao.vbs of a different content is documented. The samples that were described in this report somewhat bear resemblance to what we are seeing in BigBadWolf's DLL, yet there are differences.

For example,

> "the malware then searches inside its own binary for a string delimiter SSSSSSS, returning a string pointer to the beginning of the encrypted configuration string"

This is similar to how our sample looks for the marker SSSSSS (note the length here is only 6) to verify that the DLL downloaded is correct before proceeding to decrypt.

As another example,

> "Until it checks in with its C2 controller, the RAT server searches for the encrypted configuration buffer containing the C&Cs that will get decrypted using an AES key derived from a hardcoded string "Mother360""

The configuration is encoded with base64/ADD/XOR in BigBadWolf sample instead. Even when encryption is used, the algorithm in place is RC4.

Yet, this sample documented by Bitdefender will also match the Yara rule on "s.exe variant", based on the presence of the strings within the file. And we now know that it is a different variant from BigBadWolf, and even KuGou.

What a dreadful night!

I think you're lost. Let's try to summarise all of these information:

File	Magic header in C2 communication?	Configuration Data?	Related to BigBadWolf?	Match Yara rule "IronTiger_Gh0stRAT_variant"?
(Generated binary from BigBadWolf)	-	-	Generated from Builder. Decode address to fetch DLL with base64 / ADD 0x7A / XOR 0x59	No
AC3B2CEBB3F7A50FA237BE97B07AFA6F68BE712E932F57074444E0C02E4D8342 (DLL RAT)	DHLAQ (0x0D of header encrypted by RC4)	Decoded with base64 / ADD 0x77 / XOR 0x56	Bundled with Builder. Decrypted with RC4-decrypt with key "Kother599". Upx-packed.	Yes
bbe7d708310ec7e5f981ce4ba9928a19c4d2169b5520ffa573085f9698f90c25 (DLL RAT)	KuGou (0x11 of header encrypted by RC4)	Not encoded	No	Yes
c02a360c6f64609403b4e4d4fc130014c40ebb77f71df816c6408851c7c9ed54 (DLL RAT)	KuGou (0x11 of header encrypted by RC4)	Not encoded	No	Yes
9dcddc7ffce78526057888b43b57e76ba7f3fed0c13fb4fa4214dcb08412c447 (DLL RAT)	KuGou (0x11 of header encrypted by RC4)	Not encoded	No	Yes
852fa14860260023289ee6577dbd5e0193df31dae5f3c078142d3cac030c7462 (EXE dropper) – from Tweet	-	-	Variant. Decode address to fetch DLL with base64 / ADD 0x7A / XOR 0x59, followed by RC4 decryption with key "Getong538"	No
7BAEE22C9834BEF64F0C1B7F5988D9717855942D87C82F019606D07589BC51A9 (DLL RAT) – from Tweet	DHLAR (0x11 of header encrypted by RC4)	Decoded with base64 / ADD 0x77 / XOR 0x56, followed by RC4 decryption with key "Strong798".	Variant. Decrypted with RC4-decrypt with key "Kother599". Upx-packed.	No
(Binaries reported within Bitdefender report on Operation PZCHAO)	Spidern	Decrypted with AES.	No	Yes

At the end of the day, I think I've established (further) that Gh0stRATs has too many variants. The builder that was behind that particular s.exe seen in Operation Iron Tiger has perhaps been referenced/ modified/ improved, causing other binaries to contain similar keywords but belong to different subvariants of Gh0stRAT that probably has nothing to do with the s.exe and its user (adversary group).

Phew, glad I've got all of that information sorted out :)

That's it for today!

[1]: Operation Iron Tiger Appendix, TrendLabs Security Intelligence Blog, 2015
[2]: https://twitter.com/malwaremustd1e/status/1262274362872229888
[3]: Operation PZCHAO, Bitdefender, 2017

~~

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 ;)