

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

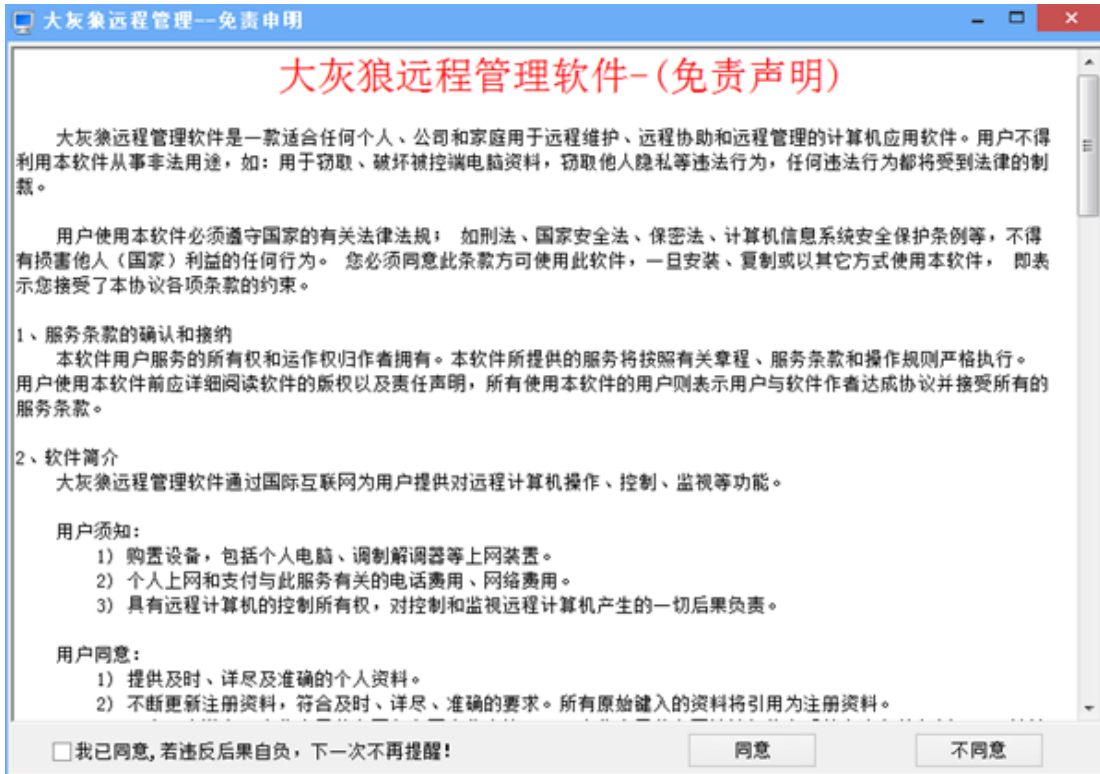
While I was preparing this post, I came across a tweet[2] from malwaremustdie that mentioned a “KuGou” backdoor along with screenshots that looked somewhat like what I observed while exploring BigBadWolf. I added these files as part of my comparison attempts, later in this post.

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

Let's get started!

The Misunderstood Wolf?

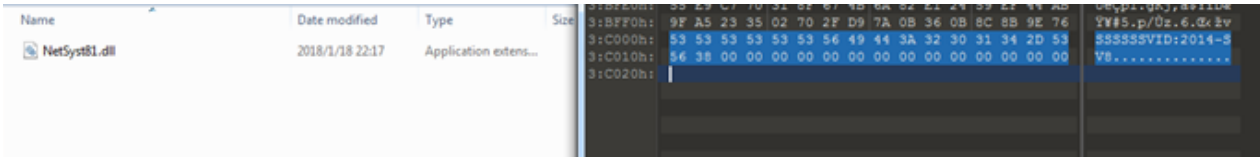
Maybe it all started as a tool for education. Really. It even came with a warning against doing evil with this toolkit. Although ticking that checkbox at the bottom of the disclaimer does felt a little like “I solemnly swear I am up to no good”.



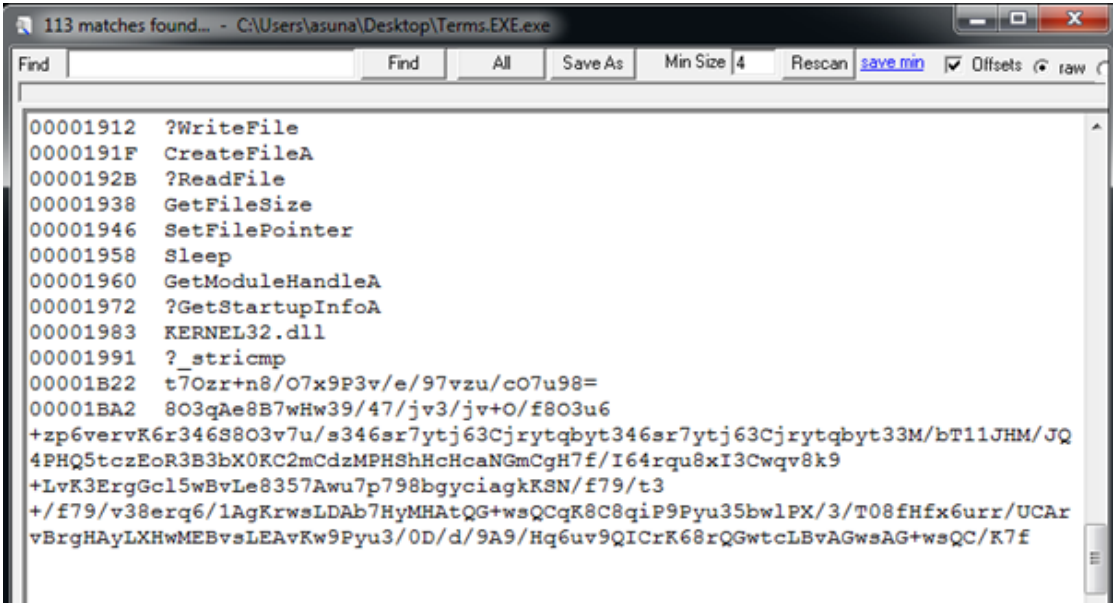
The builder component comes with the standard set of features e.g. specify the C2 IP address, mutex, name of the service to create for persistency, location to store the malicious binary on disk, options to delete the binary upon single run. There seems to be another binary ("1.dll" shown in the screen 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.

```

00401910
00401910
00401910      ; encoded_string = "t70zr+n8/07x9P3v/e/97vzu/c07u98="
00401910
00401910      ; int __cdecl decode_xor_add_401910(void *encoded_string)
00401910      decode_xor_add_401910 proc near
00401910
00401910      var_4= dword ptr -4
00401910      encoded_string= dword ptr 4
00401910
00401910  51          push     ecx
00401911  88 4C 24 08  mov     ecx, [esp+4+encoded_string]
00401915  8D 44 24 00  lea    eax, [esp+4+var_4]
00401919  50          push     eax          ; int
0040191A  51          push     ecx          ; Memory
0040191B  C7 44 24 08 00 00 00 00 mov     [esp+0Ch+var_4], 0
00401923  E8 F8 FC FF FF call   base64_decode_401620
00401928  83 C4 08    add     esp, 8
0040192B  33 C9      xor     ecx, ecx
0040192D  85 C0      test    eax, eax
0040192F  7E 27      jle    short loc_401958

```

```

00401931  53          push    ebx

```

```

00401958      loc_401958:
00401958  8B 44 24 00  mov     eax, [esp+4+var_4]
0040195C  59          pop     ecx
0040195D  C3          retn
0040195D      decode_xor_add_401910 endp
0040195D

```

```

00401932      loc_401932:
00401932  8B 54 24 04  mov     edx, [esp+8+var_4]
00401936  8A 1C 11    mov     bl, [ecx+edx]
00401939  80 C3 7A    add     bl, 7Ah
0040193C  88 1C 11    mov     [ecx+edx], bl
0040193F  8B 54 24 04  mov     edx, [esp+8+var_4]
00401943  8A 1C 11    mov     bl, [ecx+edx]
00401946  80 F3 59    xor     bl, 59h
00401949  88 1C 11    mov     [ecx+edx], bl
0040194C  41          inc     ecx
0040194D  3B C8      cmp     ecx, eax
0040194F  7C E1      jl     short loc_401932

```

```

00401951  8B 44 24 04  mov     eax, [esp+8+var_4]
00401955  5B          pop     ebx
00401956  59          pop     ecx
00401957  C3          retn

```

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.


```

00401ED6 53          push     ebx          ; hTemplateFile
00401ED7 53          push     ebx          ; dwFlagsAndAttributes
00401ED8 6A 03       push     3           ; dwCreationDisposition
00401EDA 53          push     ebx          ; lpSecurityAttributes
00401EDB 6A 01       push     1           ; dwShareMode
00401EDD 68 00 00 00 80 push     80000000h    ; dwDesiredAccess
00401EE2 68 C0 44 40 00 push     offset FileName ; lpFileName
00401EE7 FF 15 00 30 40 00 call     ds:CreateFileA
00401EED 8B F0       mov     esi, eax
00401EEF 83 FE FF   cmp     esi, 0FFFFFFFh
00401EF2 0F 84 A2 01 00 00 jz      loc_40209A

```

```

00401EF8 8D 4C 24 18 lea     ecx, [esp+48h+var_30]
00401EFC C6 44 24 18 53 mov     byte ptr [esp+48h+var_30], 'S' ; SSSSSS
00401F01 51          push    ecx          ; int
00401F02 56          push    esi          ; hFile
00401F03 C6 44 24 21 53 mov     byte ptr [esp+50h+var_30+1], 'S'
00401F08 C6 44 24 22 53 mov     byte ptr [esp+50h+var_30+2], 'S'
00401F0D C6 44 24 23 53 mov     byte ptr [esp+50h+var_30+3], 'S'
00401F12 C6 44 24 24 53 mov     [esp+50h+var_2C], 'S'
00401F17 C6 44 24 25 53 mov     [esp+50h+var_2B], 'S'
00401F1C 88 5C 24 26 mov     [esp+50h+var_2A], bl
00401F20 C6 44 24 30 56 mov     [esp+50h+var_20], 'V' ; VID:2014-SV8
00401F25 C6 44 24 31 49 mov     [esp+50h+var_1F], 'I'
00401F2A C6 44 24 32 44 mov     [esp+50h+var_1E], 'D'
00401F2F C6 44 24 33 3A mov     [esp+50h+var_1D], ':'
00401F34 C6 44 24 34 32 mov     [esp+50h+var_1C], '2'
00401F39 C6 44 24 35 30 mov     [esp+50h+var_1B], '0'
00401F3E C6 44 24 36 31 mov     [esp+50h+var_1A], '1'
00401F43 C6 44 24 37 34 mov     [esp+50h+var_19], '4'
00401F48 C6 44 24 38 2D mov     [esp+50h+var_18], '-'
00401F4D C6 44 24 39 53 mov     [esp+50h+var_17], 'S'
00401F52 C6 44 24 3A 56 mov     [esp+50h+var_16], 'V'
00401F57 C6 44 24 38 38 mov     [esp+50h+var_15], '8'
00401F5C 88 5C 24 3C mov     [esp+50h+var_14], bl
00401F60 E8 C8 FB FF FF call    check_for_presence_of_magic_401830
00401F65 83 C4 08   add     esp, 8
00401F68 8B F8       mov     edi, eax
00401F6A 56          push    esi          ; hObject
00401F6B FF 15 08 30 40 00 call    ds:CloseHandle
00401F71 3B FB       cmp     edi, ebx
00401F73 0F 84 21 01 00 00 jz      loc_40209A

```

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’.

```

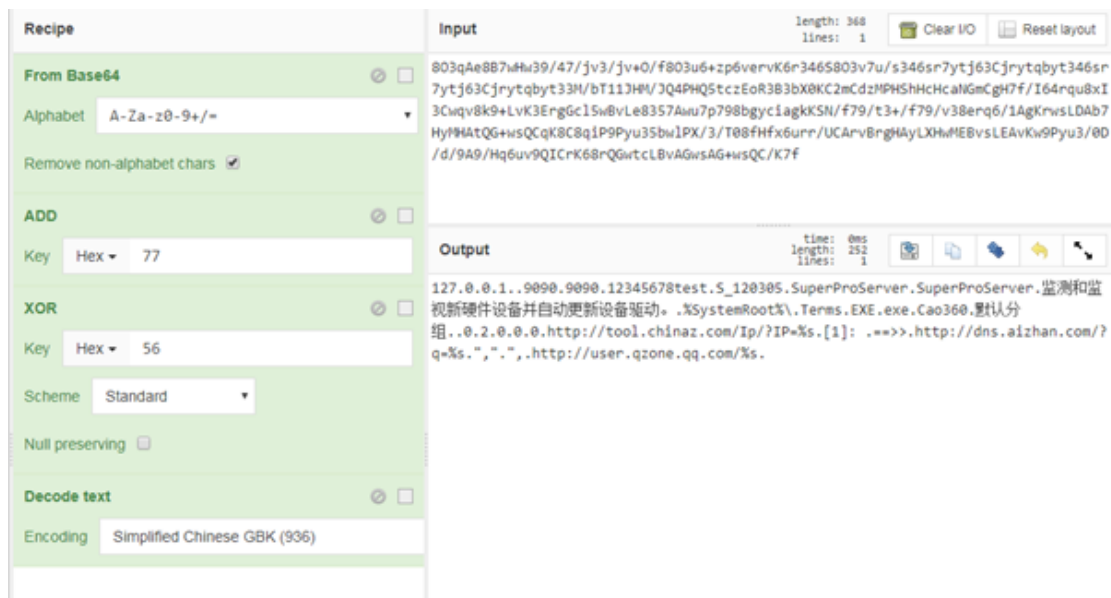
00401E40 ; unsigned int __cdecl rc4decryptfile_401E40(int encrypted_data, unsigned int length)
00401E40 rc4decryptfile_401E40 proc near
00401E40
00401E40 rc4_key= byte ptr -10Ch
00401E40 var_108= byte ptr -108h
00401E40 var_10A= byte ptr -10Ah
00401E40 var_109= byte ptr -109h
00401E40 var_108= byte ptr -108h
00401E40 var_107= byte ptr -107h
00401E40 var_106= byte ptr -106h
00401E40 var_105= byte ptr -105h
00401E40 var_104= byte ptr -104h
00401E40 var_103= byte ptr -103h
00401E40 rc4_sbox= byte ptr -100h
00401E40 encrypted_data= dword ptr 4
00401E40 length= dword ptr 8
00401E40
00401E40 81 EC 0C 01 00 00 sub esp, 10Ch
00401E46 80 39 mov al, '9'
00401E48 6A 0A push 0Ah
00401E4A 88 44 24 00 mov [esp+110h+var_105], al
00401E4E 88 44 24 0C mov [esp+110h+var_104], al
00401E52 8D 44 24 04 lea eax, [esp+110h+rc4_key]
00401E56 8D 4C 24 10 lea ecx, [esp+110h+rc4_sbox]
00401E5A 50 push eax
00401E5B 51 push ecx
00401E5C C6 44 24 0C 48 mov [esp+118h+rc4_key], 'K' ; Kother599
00401E61 C6 44 24 00 6F mov [esp+118h+var_108], 'o'
00401E66 C6 44 24 0E 74 mov [esp+118h+var_10A], 't'
00401E6B C6 44 24 0F 68 mov [esp+118h+var_109], 'h'
00401E70 C6 44 24 10 65 mov [esp+118h+var_108], 'e'
00401E75 C6 44 24 11 72 mov [esp+118h+var_107], 'r'
00401E7A C6 44 24 12 35 mov [esp+118h+var_106], 's'
00401E7F C6 44 24 15 00 mov [esp+118h+var_103], 0
00401E84 E8 87 FE FF FF call rc4_keysched_401D10
00401E89 8B 94 24 20 01 00 00 mov edx, [esp+118h+length]
00401E90 8B 84 24 1C 01 00 00 mov eax, [esp+118h+encrypted_data]
00401E97 52 push edx
00401E98 8D 4C 24 1C lea ecx, [esp+11Ch+rc4_sbox]
00401E9C 50 push eax
00401E9D 51 push ecx
00401E9E E8 0D FF FF FF call rc4_decrypt_401D00
00401EA3 81 C4 24 01 00 00 add esp, 124h
00401EA9 C3 retn
00401EA9 rc4decryptfile_401E40 endp
00401EA9

```

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.

The image shows a Wireshark packet capture. The top part is a list of packets with columns for No., Time, Source, Destination, Protocol, and Length. Packet 15446 is highlighted. Below the list, the raw packet data for packet 15446 is shown in hexadecimal and ASCII. The raw data starts with 00000000 f5 48 9c 2c d4 19 2b d1 a2 4a 81 56 3c 78 9c b3, which corresponds to the Zlib header magic 0x78 9C.

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>.

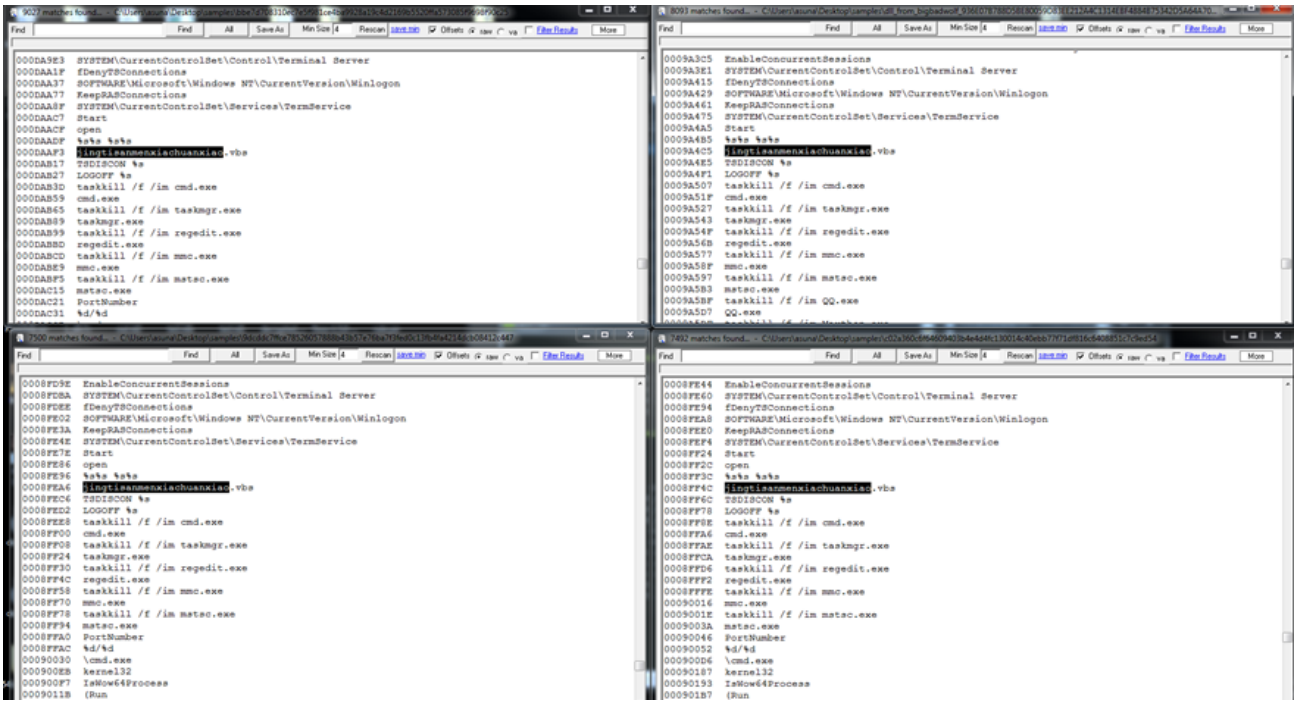
The screenshot displays a debugger window with assembly code on the left and a hex dump on the right. The assembly code includes instructions such as `push ebx`, `push ecx`, and `call dword ptr ds:[eax+sendx]`. The hex dump shows memory addresses from `02380000` to `023800E0` with hex values and their corresponding ASCII characters. A blue arrow points to the instruction `call 10004710` at address `1000462C`.

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

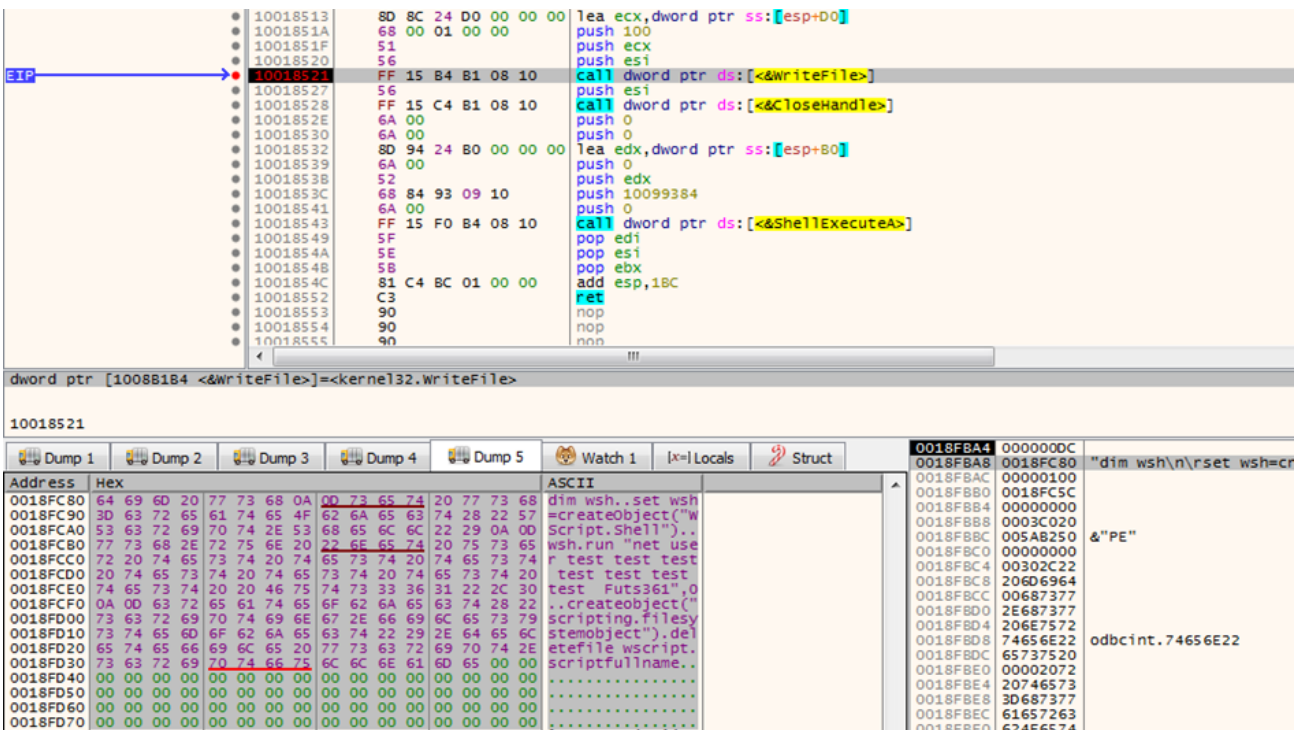
So what made these samples get flagged with that YARA rule I mentioned in the beginning of this post?

```
C:\Users\asuna\Desktop\samples
λ yara64.exe rule.yar .
IronTiger_Gh0stRAT_variant .\9dcddc7ffc78526057888b43b57e76ba7f3fed0c13fb4fa4214dcb08412c447
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 “jingtiansanmenxiachuanxiaio” 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.



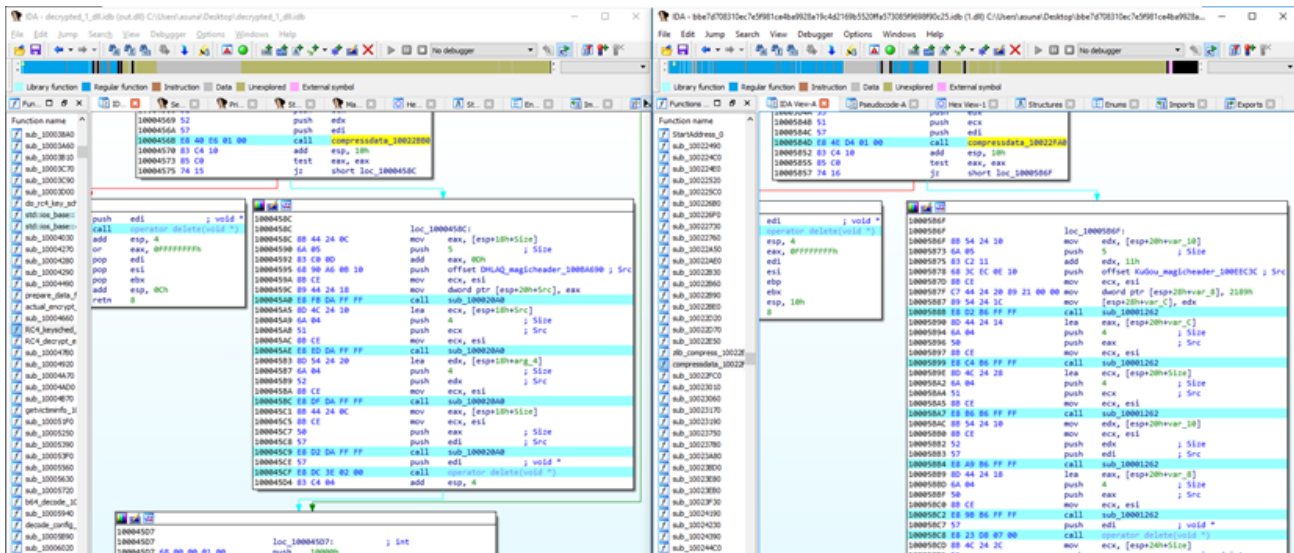
The exact same function is found in all 4 files I cross-examined.

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 9DCDDC7FFCE78526057888B43B57E76BA7F3FED0C13FB4FA4214DCB08412C447CConfidence 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.

```

1000A8C0 81 EC 34 07 00 00    sub    esp, 734h
1000A8C6 53                push  ebx
1000A8C7 56                push  esi
1000A8C8 8B B4 24 40 07 00 00  mov    esi, [esp+73Ch+arg_0]
1000A8CF 57                push  edi
1000A8D0 B9 BC 01 00 00    mov    ecx, 1BCh
1000A8D5 BF 38 8A 00 10    mov    edi, offset String ; "127.0.0.1"
1000A8DA F3 A5            rep movsd
1000A8DC 33 DB            xor    ebx, ebx
1000A8DE B9 FF 00 00 00    mov    ecx, 0FFh
1000A8E3 33 C0            xor    eax, eax
1000A8E5 8D BC 24 41 03 00 00  lea   edi, [esp+740h+var_3FF]
1000A8EC 88 9C 24 40 03 00 00  mov    [esp+740h+String], bl
1000A8F3 68 A0 8A 00 10    push  offset aZaxiaoxue ; "Zaxiaoxue"
1000A8F8 F3 AB            rep stosd
1000A8FA 8B 0D D8 F7 0E 10  mov    ecx, dword_100EF7D8
1000A900 68 34 12 0F 10    push  offset rc4key_100F1234
1000A905 66 AB            stosw
1000A907 AA                stosb
1000A908 E8 FA 69 FF FF    call  sub_10001307
1000A90D B9 40 00 00 00    mov    ecx, 40h
1000A912 33 C0            xor    eax, eax
1000A914 8D BC 24 3D 02 00 00  lea   edi, [esp+740h+var_503]
1000A91B 88 9C 24 3C 02 00 00  mov    [esp+740h+Dst], bl
1000A922 F3 AB            rep stosd
1000A924 66 AB            stosw
1000A926 AA                stosb
1000A927 8D 84 24 3C 02 00 00  lea   eax, [esp+740h+Dst]
1000A92E 68 04 01 00 00    push  104h ; nSize
1000A933 50                push  eax ; lpDst
1000A934 68 B6 8D 00 10    push  offset Src ; "%ProgramFiles%\Rumno Qrstuv"
1000A939 FF 15 80 DD 16 10  call  ds:ExpandEnvironmentStringsA
1000A93F 8D 8C 24 3C 02 00 00  lea   ecx, [esp+740h+Dst]
1000A946 51                push  ecx
1000A947 8B 0D D8 F7 0E 10  mov    ecx, dword_100EF7D8
1000A94D 68 B6 8D 00 10    push  offset Src ; "%ProgramFiles%\Rumno Qrstuv"
1000A952 E8 B0 69 FF FF    call  sub_10001307
1000A957 8B 0D D8 F7 0E 10  mov    ecx, dword_100EF7D8
1000A95D 68 B6 8D 00 10    push  offset Src ; "%ProgramFiles%\Rumno Qrstuv"
1000A962 E8 CB 70 FF FF    call  sub_10001A32
1000A967 80 B8 B5 8D 00 10 5C  cmp    byte_100D8D85[eax], 5Ch
1000A96E 75 16            jnz   short loc_1000A986

```

```

1000A970 8B 0D D8 F7 0E 10  mov    ecx, dword_100EF7D8
1000A976 68 B6 8D 00 10    push  offset Src ; "%ProgramFiles%\Rumno Qrstuv"
1000A97B E8 B2 70 FF FF    call  sub_10001A32
1000A980 80 B8 B5 8D 00 10 5C  mov    byte_100D8D85[eax], bl

```

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 image shows a decryption tool interface with a 'Recipe' panel on the left and a disassembler window on the right. The 'Recipe' panel is configured for RC4 decryption with a key of '7a' and a scheme of 'Standard'. The input is a long alphanumeric string. The output is a URL: 'http://192.161.86.98/NetSyst96.dll'. The disassembler window shows assembly code for a function named 'sub_401930', which appears to be a decryption routine using RC4 with a key of '59'.

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. “SSSSSVID:2014-SV8”, and the DLL does not contain such a footer.

The image shows the decryption tool interface with the 'Recipe' panel on the left and the 'Output' window on the right. The 'Recipe' panel is configured for RC4 decryption with a key of '4B 6F 74 68 6...' and a scheme of 'Latin1'. The input is a large file named 'NetSyst96.bin' with a size of 244,784 bytes. The output is a text file containing a message: 'MZ.....ÿÿ.....@..... í!..LÍ!This program cannot be run in DOS mode.' followed by a block of garbled text.

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.

The image shows two windows. The left window is a decryption tool interface with the following settings:

- Recipe:** From Base64, Alphabet: A-Za-z0-9+/, Remove non-alphabet chars: checked.
- ADD:** Key: Hex 77.
- XOR:** Key: Hex 56.
- Scheme:** Standard.
- Null preserving:** unchecked.
- RC4:** Passphrase: Hex 53 74 72 6F 6E..., Input format: Latin1, Output format: Latin1.
- Decode text:** Encoding: Simplified Chinese GBK (936).

The **Input** field contains a long Base64-encoded string. The **Output** field shows the decrypted configuration data, including a URL and a Chinese service description: "监测和监视新硬件设备并自动更新设备驱动".

The right window is a disassembler showing assembly code for a function named `__cdecl sub_1000570@void *memory`. The code includes comments for register usage and a loop that decrypts data using RC4:

```

1 int __cdecl sub_1000570@void *memory
2 {
3     signed int v1; // esi
4     signed int i; // eax
5     int v4; // [esp+8h] [ebp-110h]
6     char v5; // [esp+Ch] [ebp-10Ch]
7     char v6; // [esp+Dh] [ebp-10Bh]
8     char v7; // [esp+EH] [ebp-10Ah]
9     char v8; // [esp+FH] [ebp-109h]
10    char v9; // [esp+10h] [ebp-108h]
11    char v10; // [esp+11h] [ebp-107h]
12    char v11; // [esp+12h] [ebp-106h]
13    char v12; // [esp+13h] [ebp-105h]
14    char v13; // [esp+14h] [ebp-104h]
15    char v14; // [esp+15h] [ebp-103h]
16    char v15; // [esp+16h] [ebp-102h]
17
18    v4 = 0;
19    v1 = sub_1000570@memory, (int)&v1;
20    for ( i = 0; i < v1; ++i )
21    {
22        *(_BYTE *) (i + v4) += 0x77;
23        *(_BYTE *) (i + v4) ^= 0x56u;
24    }
25    v5 = 'S';
26    v6 = 't';
27    v7 = 'i';
28    v8 = 'o';
29    v9 = 'n';
30    v10 = 'g';
31    v11 = '7';
32    v12 = '9';
33    v13 = '8';
34    v14 = 0;
35    rc4_keysched_1001F190((int)&v5, (int)&v6, strlen(&v5));
36    rc4_decrypt_1001F230((unsigned int)&v5, v4, v1);
37    return v4;
38}

```

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.

```

10004481 8B 44 24 1C    mov     eax, [esp+18h+arg_0]
10004485 57             push    edi
10004486 8D 4C 24 0C    lea    ecx, [esp+1Ch+var_10]
1000448A 50             push    eax
1000448B 51             push    ecx
1000448C 56             push    esi
1000448D E8 BE DC 01 00 call   compressdata_10022150
10004492 83 C4 10      add    esp, 10h
10004495 85 C0        test   eax, eax
10004497 74 14        jz     short loc_100044AD

```

```

10004578
10004578                loc_10004578
10004578 5F                pop
10004579 33 C0            xor     eax, eax
1000457B 5E                pop
1000457C 83 C4 10        add    esp, 10h
1000457E C2 08 00        retn   8
1000457F                sub_1000457F
1000457F

```

```

push     esi           ; void *
call    operator_delete(void *)
add     esp, 4
or      eax, 0FFFFFFFh
pop     edi
pop     esi
add     esp, 10h
retn    8

```

```

100044AD
100044AD                loc_100044AD:
100044AD 8B 54 24 08    mov     edx, [esp+18h+var_10]
100044B1 53             push    ebx
100044B2 55             push    ebp
100044B3 8D 6A 11      lea    ebp, [edx+11h]
100044B6 55             push    ebp           ; unsigned int
100044B7 E8 E8 35 02 00 call   operator_new(uint)
100044BC 8B CD        mov     ecx, ebp
100044BE 8B D8        mov     ebx, eax
100044C0 8B D1        mov     edx, ecx
100044C2 33 C0        xor     eax, eax
100044C4 8B FB        mov     edi, ebx
100044C6 68 00 01 00 00 push   100h           ; unsigned int
100044CB C1 E9 02      shr     ecx, 2
100044CE F3 AB        rep stosd
100044D0 8B CA        mov     ecx, edx
100044D2 83 E1 03      and     ecx, 3
100044D5 F3 AA        rep stosb
100044D7 8B 0D 5C A3 0B 10 mov     ecx, DHLAR_100BA35C
100044DD 8B C3        mov     eax, ebx
100044DF 8D 7B 11      lea    edi, [ebx+11h]
100044E2 89 08        mov     [eax], ecx
100044E4 8A 15 60 A3 0B 10 mov     dl, byte ptr word_100BA360
100044EA 8B 50 04        mov     [eax+4], dl
100044ED 8B 44 24 30    mov     eax, [esp+28h+arg_4]
100044F1 C7 43 05 00 00 00 00 mov     dword ptr [ebx+5], 0
100044F8 89 68 09        mov     [ebx+9], ebp
100044FB 89 43 0D        mov     [ebx+0Dh], eax
100044FE 8B 4C 24 18    mov     ecx, [esp+28h+var_10]
10004502 8B D1        mov     edx, ecx
10004504 C1 E9 02      shr     ecx, 2
10004507 F3 A5        rep movsd
10004509 8B CA        mov     ecx, edx
1000450B 83 E1 03      and     ecx, 3
1000450E F3 A4        rep movsb
10004510 E8 8F 35 02 00 call   operator_new(uint)
10004515 8B D0        mov     edx, eax
10004517 B9 40 00 00 00 00 mov     ecx, 40h
1000451C 33 C0        xor     eax, eax
1000451E 8B FA        mov     edi, edx
10004520 F3 AB        rep stosd
10004522 B9 40 00 00 00 00 mov     ecx, 40h
10004527 BE 5C A2 0B 10 mov     esi, offset unk_100BA25C
1000452C 8B FA        mov     edi, edx
1000452E 6A 11        push   11h
10004530 53             push    ebx
10004531 52             push    edx
10004532 F3 A5        rep movsd
10004534 89 54 24 3C    mov     [esp+34h+arg_4], edx
10004538 E8 F3 AC 01 00 call   rc4_encrypt_decrypt_1001F230
1000453D 8B 4C 24 28    mov     ecx, [esp+34h+var_C]

```

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 SSSSSS, 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
AC3B2CEBB3F7A50FA237BE97B07AFA6F68B E712E932F57074444E0C02E4D8342 (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 ;)