A Deep-Dive Analysis of the NukeSped RATs

By Minh Tran

Published: 2019-10-23 · Archived: 2026-04-05 19:36:29 UTC

A FortiGuard Labs Threat Analysis

Introduction

Advanced Persistent Threat (APT) groups pose a great threat to global security, especially groups associated with nation states. Of all APT groups, those groups from North Korea have really stood out due to the great damage they have done as well as for their persistence. The U.S. Government, in particular, refers to the malicious threat actor connected to the North Korean government as HIDDEN COBRA.

FortiGuard Labs has been actively monitoring various APT groups such as HIDDEN COBRA. For example, in a previous post we gave an overview of the FALLCHILL Remote Administration Tools (RATs). Recently, we noticed some new interesting samples from this group, so we decided to take a further look.

A Bird's Eye View of the RAT Samples

The RAT samples we analyzed are summarized below:

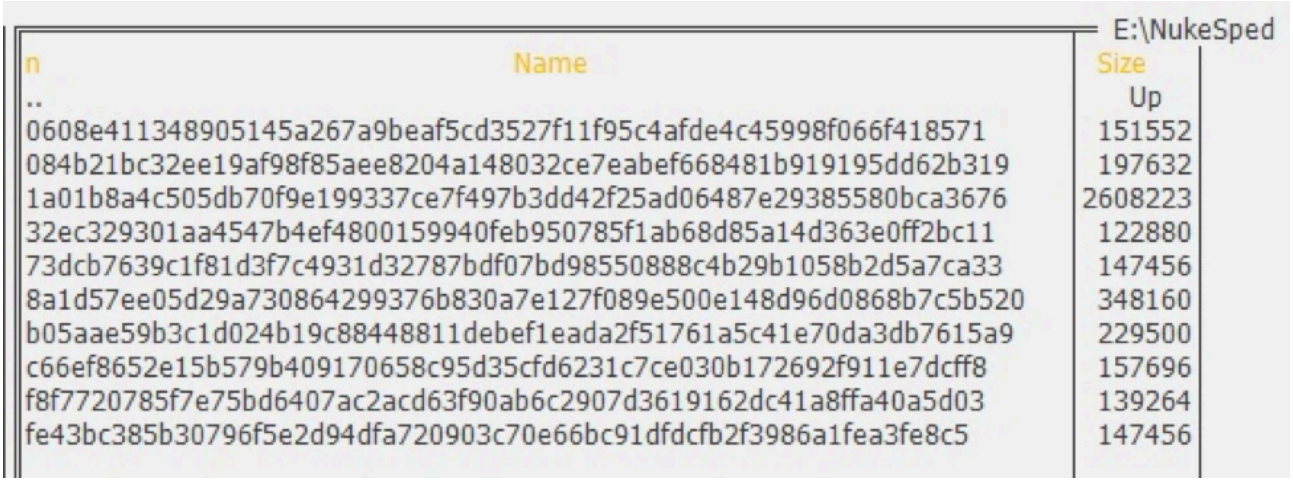

Figure 1: RAT samples

At a high level, they share similar characteristics:

- Most are 32 bits
- Strings are encrypted to hinder analysis
- Compilation timestamp are from May 04 10:40:47 2017 to Feb 13 04:06:28 2018

As we **shall** see, they actually share more similarities than differences. In some cases, they even reuse functions.

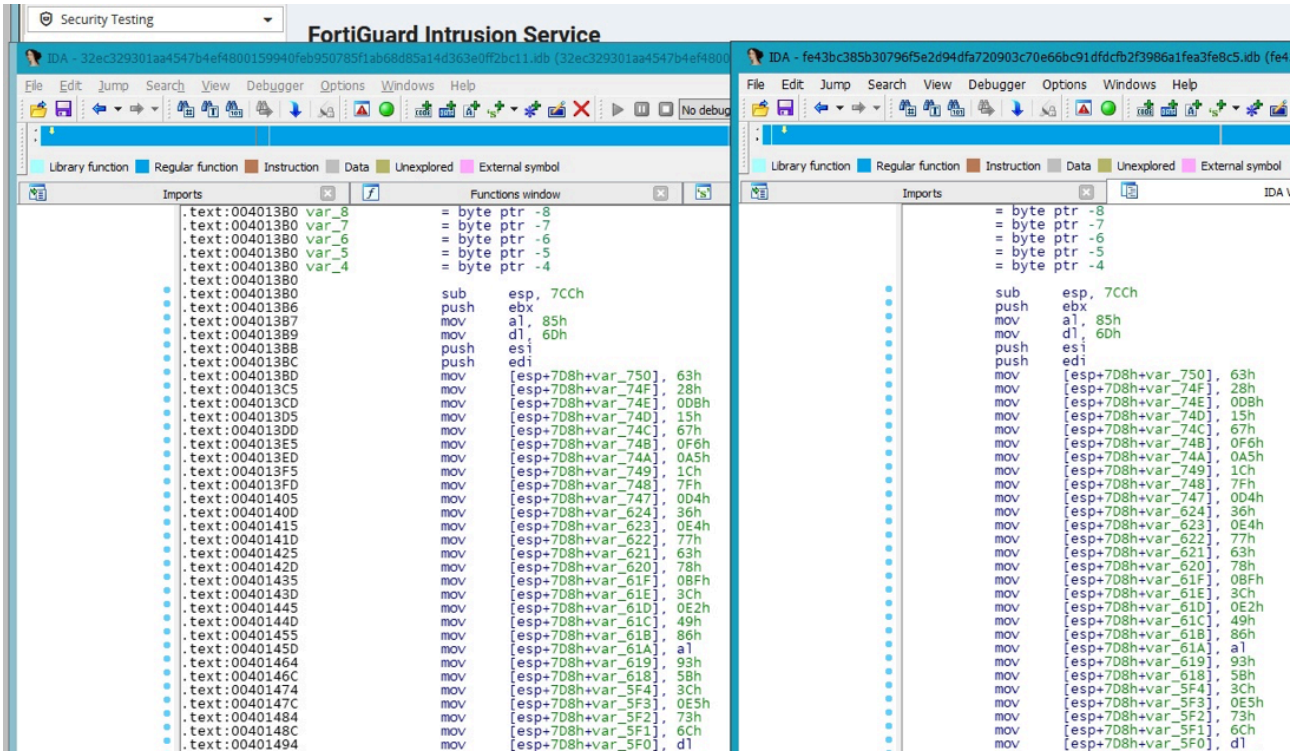

Figure 2: Code Reuse

Let’s inspect the resource sections in more detail, as they often give clues to the origin of the malware.

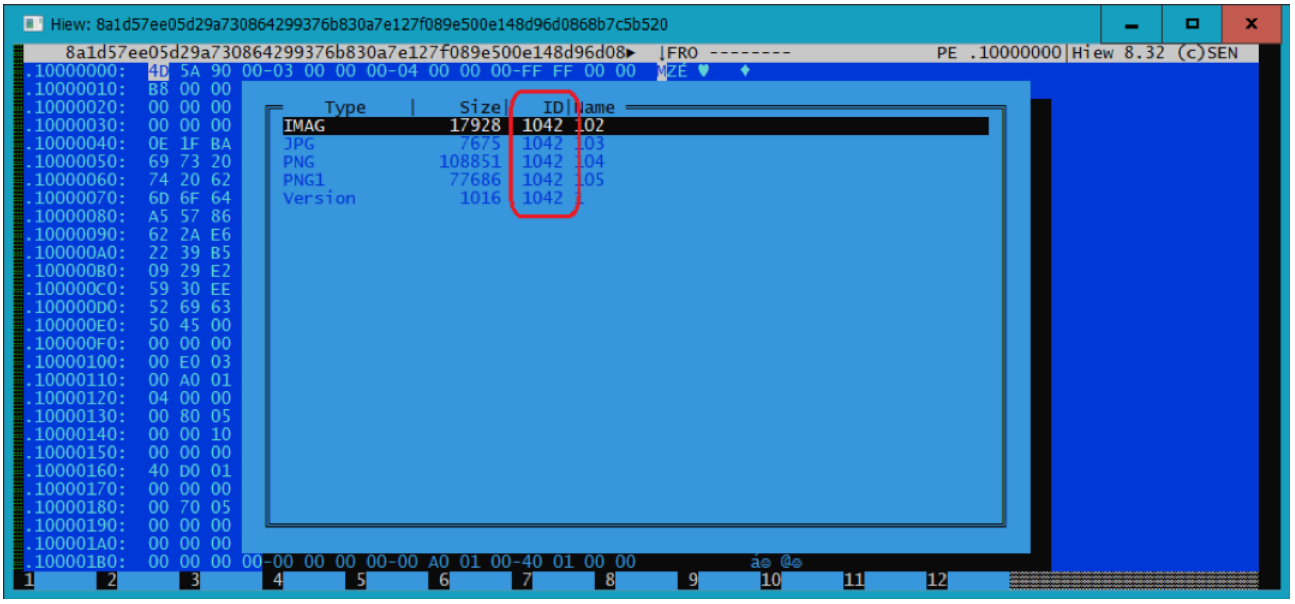

Figure 3: Language ID

As can be seen, each resource has a language ID associated with it. Curiously, most samples have the language ID of 1042.

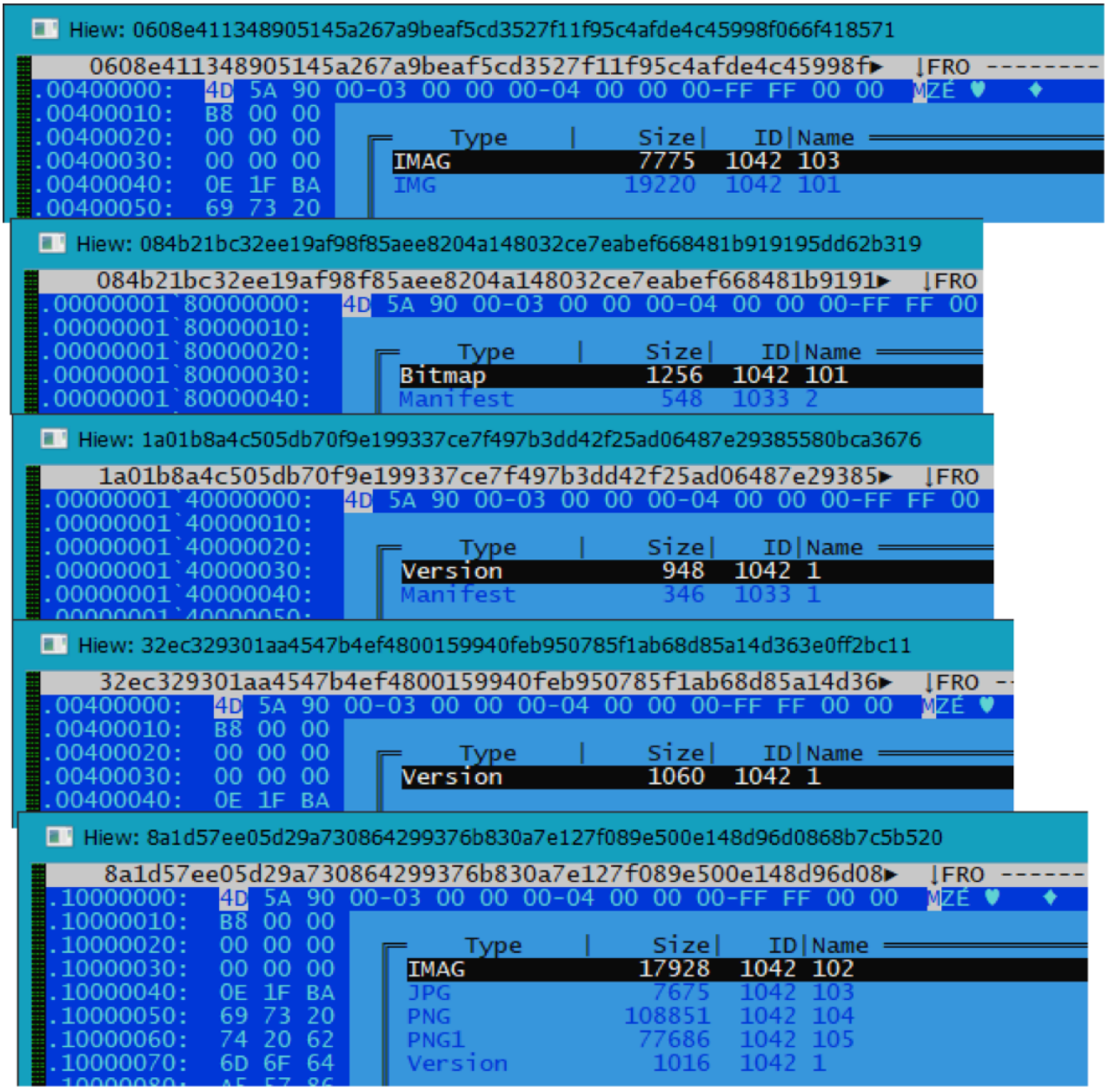

Figure 4: Most Samples Have the Language ID Of 1042.

As per this authoritative source, 1042 (0x0412) is the language Identifier for Korean.

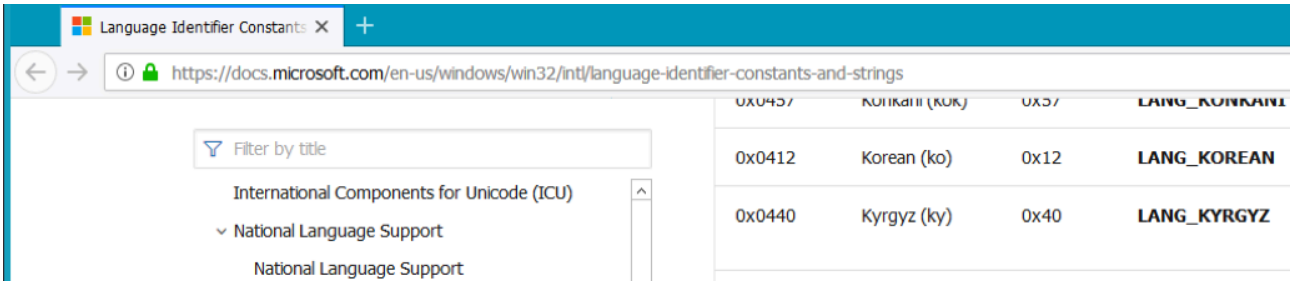

Figure 5: LANG_KOREAN

Functionality of the Malware

Our analysis started by trying to get a feel for what this malware could possibly do on victim's system. In general, the best way to do that is by inspecting the functionality (e.g. from an API) that it wants to invoke from the target system. So, let get right to it.

At first sight, these malware do not seem to invoke many APIs. The import table is short and does not import many common DLLs and functions. Our gut feeling suggested that it will likely resolve functions dynamically. And sure enough, we quickly found instances of **GetProcAddress**. It even encrypted its API names too. In our experience, this is a common technique designed to hinder static analysis, but it does not stop dynamic analysis. So, we traced the malware and figured out the encrypted APIs.

```
3740  v1560 = 39;
3741  v1561 = -54;
3742  v1562 = 29;
3743  v1563 = -111;
3744  v1564 = -9;
3745  v1565 = 96;
3746  v1566 = 69;
3747  v1567 = 120;
3748  v0 = mtr_decodeStr(&v514, 0xDu);
3749  Kernel32_dll = LoadLibraryA(v0);
3750  if ( Kernel32_dll )
3751  {
3752    v2 = mtr_decodeStr(&v908, 0xFu);
3753    GetProcAddress_0 = (int (__stdcall *)(_DWORD, _DWORD))GetProcAddress(Kernel32_dll, v2);
3754    v3 = mtr_decodeStr(&v540, 0xDu);
3755    LoadLibraryW = GetProcAddress_0(Kernel32_dll, v3);
3756    v4 = mtr_decodeStr(&v370, 0xCu);
3757    FreeLibrary = GetProcAddress_0(Kernel32_dll, v4);
3758    v5 = mtr_decodeStr(&v1158, 0x11u);
3759    GetModuleHandleW_0 = GetProcAddress_0(Kernel32_dll, v5);
3760    v6 = mtr_decodeStr(&v1349, 0x13u);
3761    GetModuleFileNameW = GetProcAddress_0(Kernel32_dll, v6);
3762    v7 = mtr_decodeStr(&v878, 0xFu);
3763    CreateProcessW = GetProcAddress_0(Kernel32_dll, v7);
3764    v8 = mtr_decodeStr(&v618, 0xDu);
3765    CreateThread = (int (__stdcall *)(_DWORD, _DWORD, _DWORD, _DWORD, _DWORD, _DWORD))GetProcAddress_0(Kernel32_dll, v8);
3766    v9 = mtr_decodeStr(&v1387, 0x13u);
3767    GetExitCodeProcess = GetProcAddress_0(Kernel32_dll, v9);
3768    v10 = mtr_decodeStr(&v1277, 0x12u);
3769    GetExitCodeThread = GetProcAddress_0(Kernel32_dll, v10);
3770    v11 = mtr_decodeStr(&v1260, 0x11u);
3771    TerminateProcess = GetProcAddress_0(Kernel32_dll, v11);
3772    v12 = mtr_decodeStr(&v1028, 0x10u);
3773    TerminateThread = GetProcAddress_0(Kernel32_dll, v12);
3774    v13 = mtr_decodeStr(&v1445, 0x14u);
3775    WaitForSingleObject = (int (__stdcall *)(_DWORD, _DWORD))GetProcAddress_0(Kernel32_dll, v13);
3776    v14 = mtr_decodeStr(&v502, 0xCu);
3777    CloseHandle = (int (__stdcall *)(_DWORD))GetProcAddress_0(Kernel32_dll, v14);
3778    v15 = mtr_decodeStr(&v185, 8u);
3779    WinExec = GetProcAddress_0(Kernel32_dll, v15);
3780    v16 = mtr_decodeStr(&v334, 0xCu);
```

Figure 6: Decrypted APIs

As can be seen, after patching in IDA everything starts to make sense.

The following shows one special case where function names are not encrypted at all, and hence static analysis is enough.

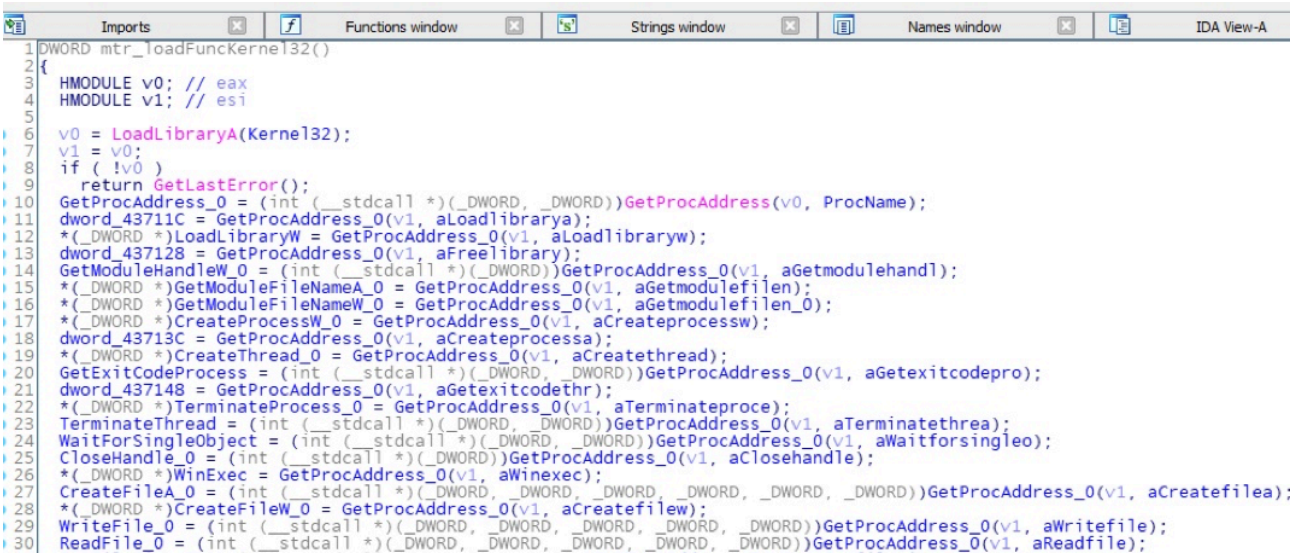

Figure 7: Function Table

The hash of this special sample is b05aae59b3c1d024b19c88448811debef1eada2f51761a5c41e70da3db7615a9. As astute readers may have noticed, the order of the functions being loaded in this sample is very similar to other samples.

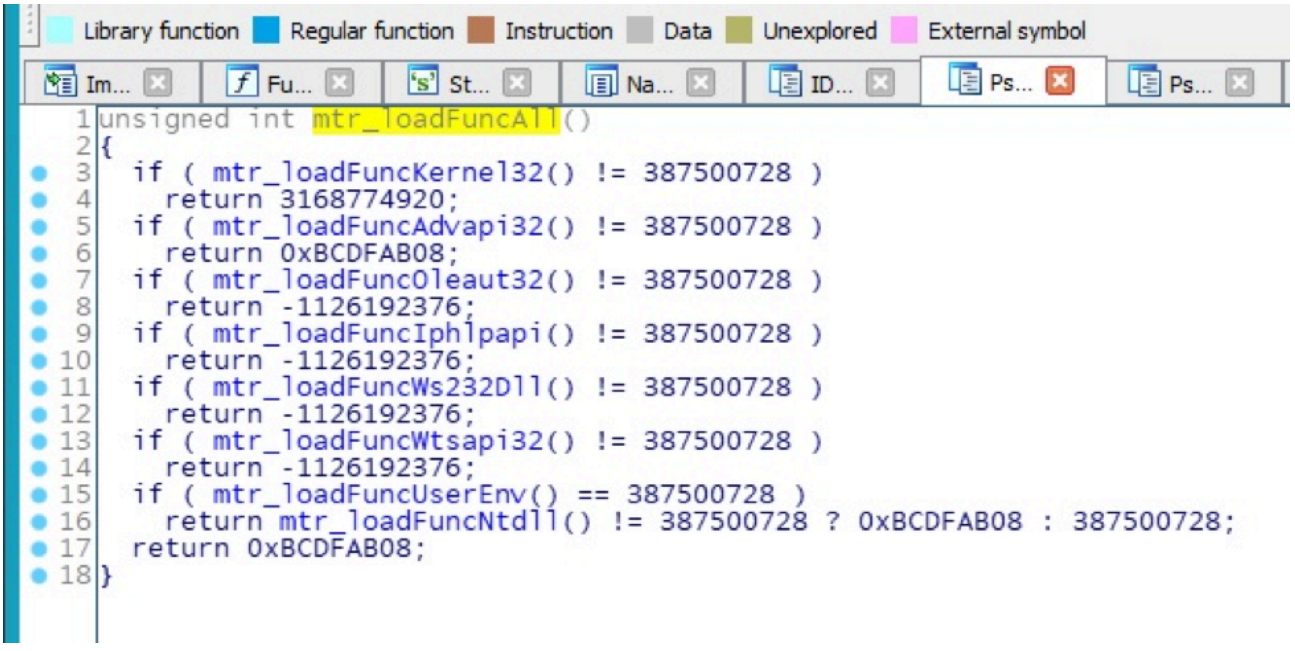

Figure 8: Main DLLs

After patching up the function names in IDA, we can clearly see that the malware makes use of core functionalities like registry (**Advapi32.dll**), networking (**ws2_32.dll**), and so on.

To persist, the malware inserts itself into a Run key:

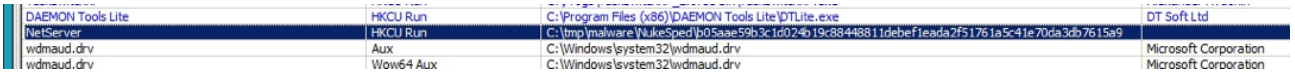

Figure 9: Persistence

In some other cases, the malware installs itself as a service.

```
.rdata:10021934                 dd rva word_10021950    ; AddressOfNameOrdinals
.rdata:10021938 ;
.rdata:10021938 ; Export Address Table for T_SVC_DLL.dll
.rdata:10021938 ;
.rdata:10021938 off_10021938    dd rva ServiceCtrlHandler, rva _DllMain@12, rva ServiceMain
.rdata:10021938                                         ; DATA XREF: .rdata:1002192C↑o
.rdata:10021938                                         ; DllMain(x,x,x)
.rdata:10021944 ;
.rdata:10021944 ; Export Names Table for T_SVC_DLL.dll
.rdata:10021944 ;
.rdata:10021944 off_10021944    dd rva aDllmain, rva aServicectrlhan, rva aServicemain
.rdata:10021944                                         ; DATA XREF: .rdata:10021930↑o
.rdata:10021944                                         ; "ServiceCtrlHandler" ...
.rdata:10021950 ;
.rdata:10021950 ; Export Ordinals Table for T_SVC_DLL.dll
.rdata:10021950 ;
.rdata:10021950 word_10021950   dw 1, 0, 2              ; DATA XREF: .rdata:10021934↑o
.rdata:10021956 aTSvcDllDll     db 'T_SVC_DLL.dll',0    ; DATA XREF: .rdata:1002191C↑o
.rdata:10021964 aServicectrlhan db 'ServiceCtrlHandler',0
.rdata:10021964                                         ; DATA XREF: .rdata:off_10021944↑o
.rdata:10021977 aDllmain        db 'DllMain',0          ; DATA XREF: .rdata:off_10021944↑o
.rdata:1002197F aServicemain    db 'ServiceMain',0      ; DATA XREF: .rdata:off_10021944↑o
.rdata:1002198B                 align 800h
.rdata:1002198B _rdata          ends
```

Figure 10: Service

As we can see, here is where the original name of the DLL is hidden.

Ghosts in the $hell

Let’s get to the main functionality of NukeSped: Remote Administration Tool.

After more reverse-engineering, we figured out the algorithm used to decode the strings.

Imports | Funct

```
1  int __cdecl mtr_decode(int a1, int a2)
2  {
3    int result; // eax
4    char v3; // dl
5    int i; // [esp+0h] [ebp-4h]
6
7    result = sub_410078(a2);
8    for ( i = 0; i < a2; ++i )
9    {
10     v3 = decode(&unk_4377B8) ^ *(_BYTE *)(i + a1
11     result = i + a1;
12     *(_BYTE *)(i + a1) = v3;
13   }
14   return result;
15 }
```

Figure 11: Decoding Routine

In a nutshell, the malware uses custom encryption based on **xor.** In turn, we used decodeCmd on this core function to decrypt commands from the remote attackers.

Imports | Functions window

```
1  signed int __cdecl decodeCmd(unsigned __int16 *a1)
2  {
3    int v1; // eax
4    char *v2; // eax
5    unsigned __int16 v3; // si
6    unsigned __int16 v4; // cx
7    bool v5; // cf
8    bool v6; // zf
9    const void *v7; // esi
10   signed int cmd; // eax
11
12   v1 = recvCmd(&dword_436550, (int)mtr_buff1, 16300);
13   if ( v1 <= 0 )
14     return 0;
15   mtr_decode((int)mtr_buff1, v1);
16   v2 = mtr_buff1;
17   *a1 = *(_WORD *)mtr_buff1;
18   v3 = *a1;
19   v4 = *((_WORD *)v2 + 1);
20   v5 = *a1 < 0xB6CEu;
21   v6 = *a1 == -18738;
22   a1[1] = v4;
23   if ( !v5 && !v6 || v3 < 0xB6A4u || v4 > 0x1000u )
24     return 0;
25   v7 = v2 + 4;
26   cmd = 1;
27   qmemcpy(a1 + 2, v7, v4);
28   return cmd;
29 }
```

Figure 12: Decode Commands

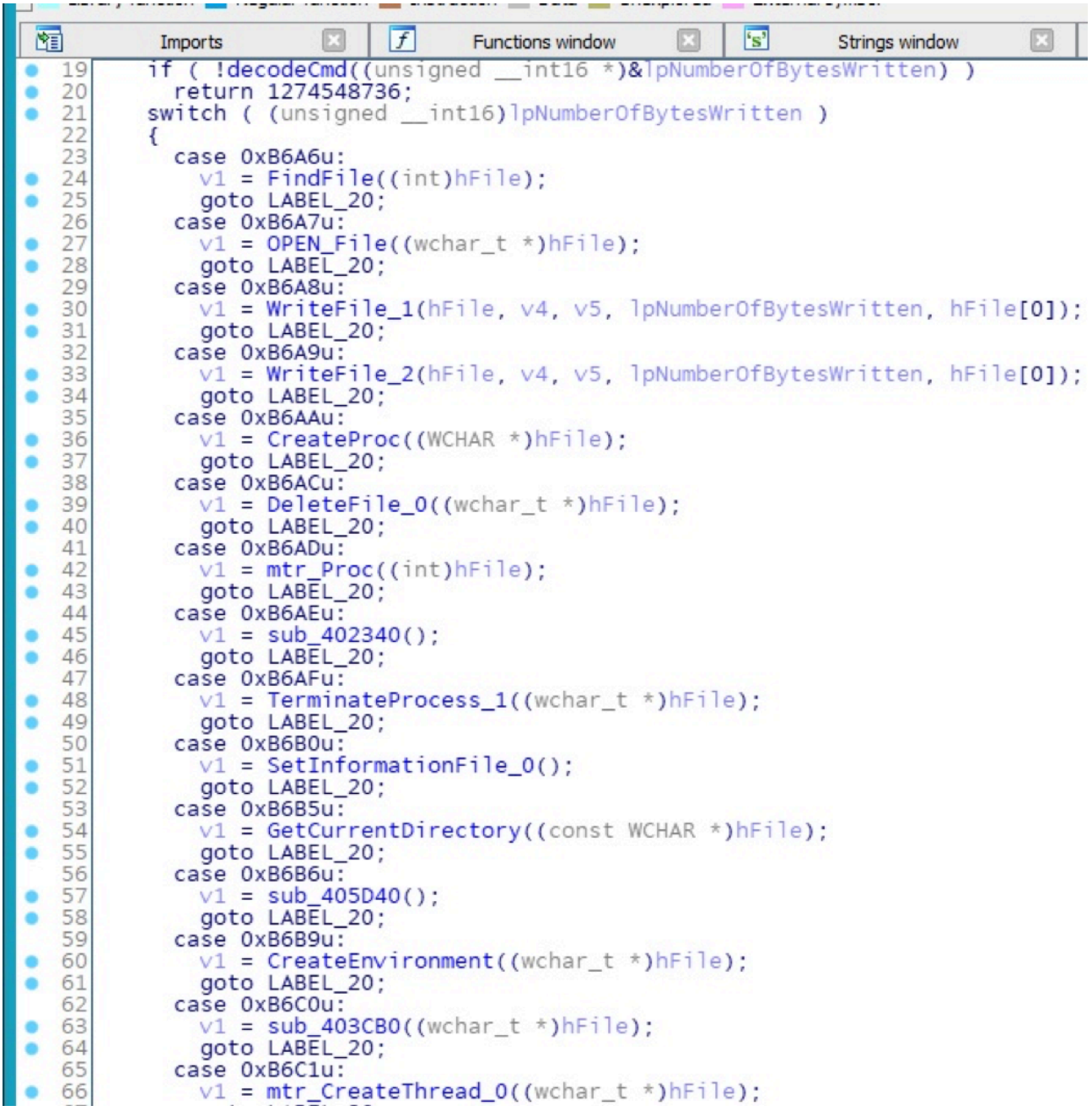

Figure 13: Logic of the Shell

Like a typical RAT, it listens for incoming commands, executes those commands, and then responds. The full control flow graph (CFG) looks like the following:

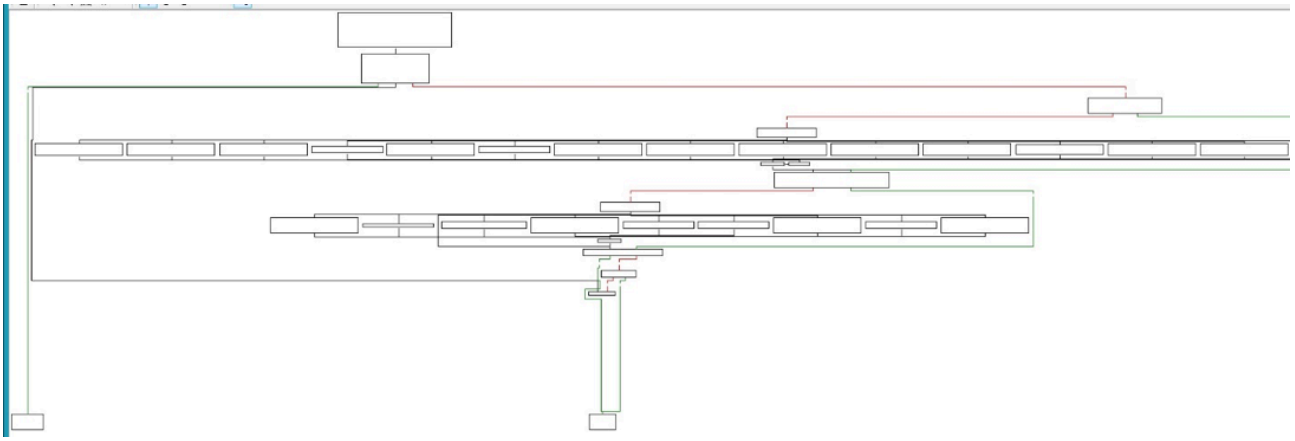

Figure 14: Control Flow Graph (CFG)

As can be seen in Figure 14, the control flow of a typical shell is clear. At the beginning is the common logic of parsing of the command and its parameters. And then there is a distinctly huge switch-case to handle each command.

We have reverse-engineered the logic of the RAT and found many classical RAT features:

- Iterate files in a folder
- Create a process as another user
- Iterate processes and modules
- Terminate a process
- Create a process
- Write a file
- Read a file
- Connect to a remote host
- Move a file
- Retrieve and launch additional payloads from the internet
- Get information about installed disks, including the disk type and the amount of free space on the disk
- Get the current directory
- Change to a different directory
- Remove itself and artifacts associated with it from the infected system

Attribution

Attribution is almost always an imprecise art, but let's consider the **key** evidence:

- The pattern of the encrypted strings, and the way string is used for API loading (Figure 8, etc.)
- The feature set and the structure of the main function (RAT) are reminiscent of FALLCHILL (below)

```
IDA View-A | Pseudocode-D | Pseudocode-C | Pseudocode-B

19
20  v0 = -88;
21  cmdStr = (wchar_t *)LocalAlloc_0(64i64, 0x8000i64);
22  if ( cmdStr )
23  {
24    while ( 1 )
25    {
26      memset(cmdStr, 0, 0x8000ui64);
27      LODWORD(v13) = -1;
28      v0 = mtr_receiveCmd64(mtr_SOCKET, cmdStr, &cmdCode, &v15, v13);
29      if ( v0 )
30        break;
31      v2 = cmdCode;
32      cmdStr[(unsigned __int64)v15 >> 1] = 0;
33      switch ( v2 )
34      {
35        case 65332:
36          v3 = mtr_iterFiles(cmdStr);
37          break;
38        case 65333:
39          v3 = mtr_iterProcessesAndModules();
40          break;
41        case 65334:
42          if ( *cmdStr == '/' )
43            v4 = mtr_TerminateProcess(cmdStr + 1);
44          else
45            v4 = mtr_CreateToolhelp32Snapshot(cmdStr);
46          if ( v4 && v4 != -1 && (v5 = OpenProcess(0x100001u, 1, v4)) != 0i64 && TerminateProcess_0(v5, 0) )
47          {
48            v3 = mtr_SendBack(-99);
49          }
50          else
51          {
52            v6 = GetLastError_0();
53            v3 = mtr_SendBack(v6);
54          }
55          break;
56        case 65335:
57          v3 = mtr_CreateProcess(cmdStr);
58          break;
59        case 65336:
60          v3 = mtr_CreateProcessAsUser(cmdStr);
61          break;
62        case 65338:
63          v3 = mtr_WriteFile(cmdStr);
64          break;
65        case 65339:
66          v3 = mtr_iterReadFiles(cmdStr);
67          break;
68        case 65342:
69          v3 = mtr_misc3(cmdStr);
70          break;
71        case 65343:
72          v3 = mtr_cloneFileTime(cmdStr);
73          break;
```

Figure 15: Logic of the Shell in FALLCHILL

- Most samples of NukeSped have the following cryptography blob (Figure 16). Interestingly, they also have a cryptography blob similar to this:

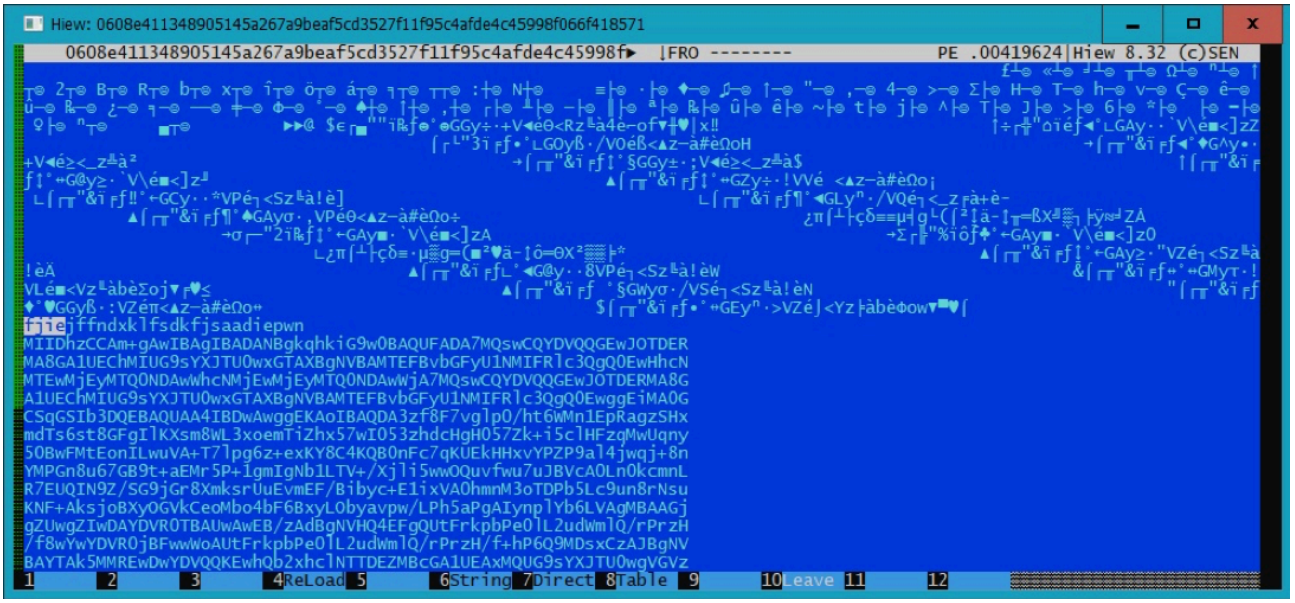

Figure 16: Cryptography Blob

- Interestingly, there are also file name references shared with [HOPLIGHT]

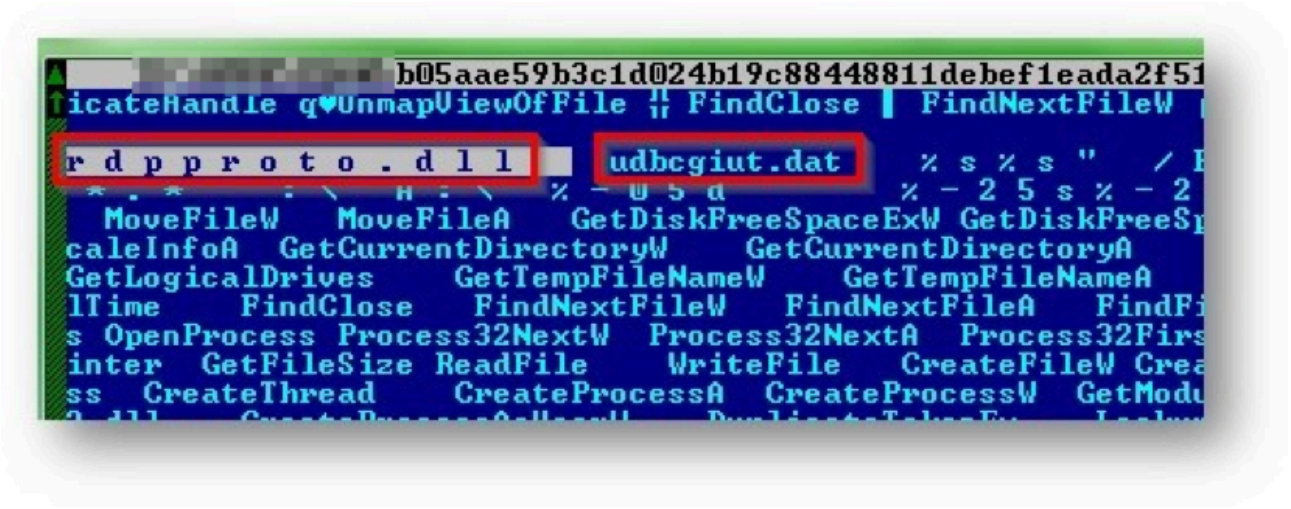

Figure 17: Dumped File

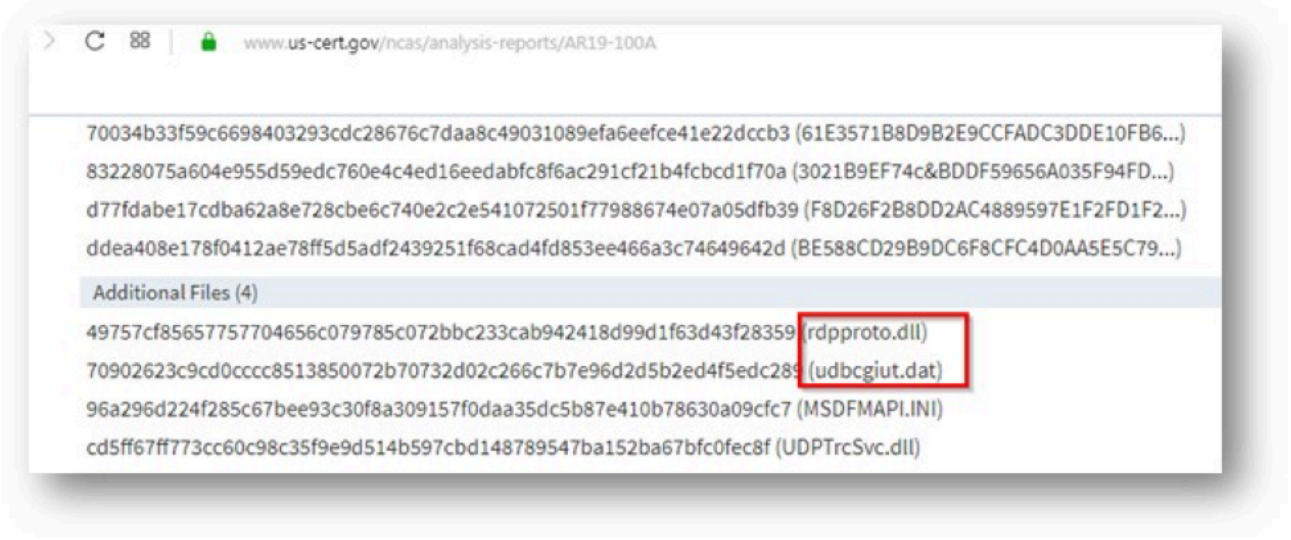

Figure 18

- Most samples (7 out of 10) of NukeSped are in Korean (e.g. Figure 4).

Given all the evidences so far, we can conclude that the NukeSped RATs have some relation to North Korea threat actors (HIDDEN COBRA) .

Solution

Internal testing by FortiGuard Labs shows that all networks and devices being protected by Fortinet solutions running the latest subscription service updates were automatically protected from this malware.

In particular, FortiGuard Antivirus service detects samples as the following:

1a01b8a4c505db70f9e199337ce7f497b3dd42f25ad06487e29385580bca3676 W64/HidCobra.A!tr
8a1d57ee05d29a730864299376b830a7e127f089e500e148d96d0868b7c5b520 W32/NukeSped.AU!tr
32ec329301aa4547b4ef4800159940feb950785f1ab68d85a14d363e0ff2bc11 W32/Trojan.FPIA!tr
73dcb7639c1f81d3f7c4931d32787bdf07bd98550888c4b29b1058b2d5a7ca33 W32/NukeSped.AU!tr
084b21bc32ee19af98f85aee8204a148032ce7eabef668481b919195dd62b319 W32/HidCobra.9CFB!tr
0608e411348905145a267a9beaf5cd3527f11f95c4afde4c45998f066f418571 W32/NukeSped.AU!tr
b05aae59b3c1d024b19c88448811debef1eada2f51761a5c41e70da3db7615a9 W32/HidCobra.9CFB!tr
c66ef8652e15b579b409170658c95d35cfd6231c7ce030b172692f911e7dcff8 W32/NukeSped.AU!tr
f8f7720785f7e75bd6407ac2acd63f90ab6c2907d3619162dc41a8ffa40a5d03 W32/NukeSped.AU!tr
fe43bc385b30796f5e2d94dfa720903c70e66bc91dfdcfb2f3986a1fea3fe8c5 W32/NukeSped.AU!tr

C2

Malicious URLs related to this malware are blocked by FortiGuard Web Filtering Service & the botnet IP engine:

119[.]18[.]230[.]253

218[.]255[.]24[.]226

The author wants to thank Artem Semenchenko for additional insights during the attribution process.

As usual, FortiGuard Labs will keep an eye out for advanced threats like this to help keep everybody protected.

-= FortiGuard Lion Team =-

Learn more about FortiGuard Labs and the FortiGuard Security Services portfolio. Sign up for our weekly FortiGuard Threat Brief.

Read about the FortiGuard Security Rating Service, which provides security audits and best practices.
