Night Walker Report

Rinal Mohamed

September 13, 2024

1 Overview & Summary

The executable being analyzed is, by inspection, a spyware program that collects and monitors the keystrokes of the infected device and encrypts that output in some way afterwards. What has been provided in this challenge is the sample that requires analysis and a text file called ”flag.txt” that apparently contains cryptic text. In the upcoming sections of the report, a detailed analysis of the provided sample is presented shedding light on the software programs that were used during the analysis.

2 Sample Analysis

In order to analyze the above mentioned sample, basic and advanced static analysis were conducted on the executable to better understand its functionality.

2.1 Basic Static Analysis

1. The first step is to run strings command on cmder command line to get a brief overview and see if anything of interest can be found. As can be seen in fig(1) below, some interesting clues appeared which, in general, hinted about what might be going on within that sample. The figure clearly shows multiple keywords for encryption functions such as BCryptEncrypt, BCryptGenetateSymmetricKey, and some other stuff suggesting the use of encryption inside the given sample. There is also a file named ”Secret_log.txt” that is worth having a deeper look into to better understand its scope. Multiple keyboard key names also showed up in the analysis of the sample which is pretty interesting as well and worth better comprehension.

2. The main aim of this step is to get a top-level view of the stuff that is going on in the sample. To delve deeper in what has been mentioned in step 1, we load the program in Ida software to know the functional scope of these mentions. First thing I hopped into was the function that had multiple references to the key names. As can be seen in fig(2) below, the function had various cases that need to be checked; seems like a huge switch case, or some nested if statements.

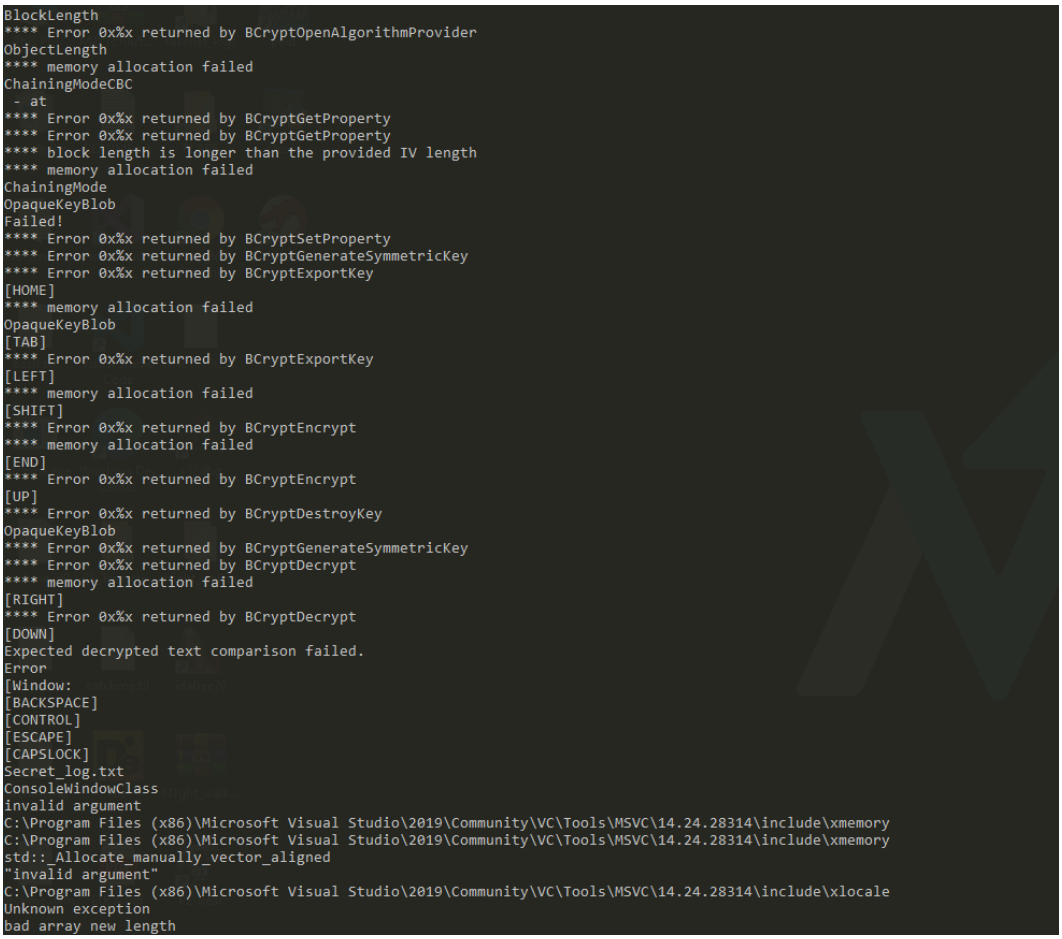


Figure 1: (Basic Static Analysis) The output of strings command

3. Afterwards, I went on to see the context of the ”secret.txt” file. As shown in fig(3), there are apparently various Windows APIs (FindWindow, ShowWindow, GetMessage) that were used in that function. However, none of which gave a strong clue about what kind of application is being executed by this sample. I then kept gravitating around the functions that get called inside that function till I landed on another Windows API which is SetWindowsHookExA shown in fig(4) below(SetWindowsHookExA was reached by jumping into sub_140001226). According to Microsoft documentation for that function, this API ”Installs an application-defined hook procedure into a hook chain.”[1]. As per the description, a hook procedure is used to monitor the system waiting for a certain kind of event. The API takes as a parameter an integer value, namely the idHook [1]. This idHook specifies the kind of hook procedure that will be installed. As shown in fig(4), the idHook is passed to the function as 0D in hex which translates to 13 in decimal. This value, as per the documentation, installs a procedure responsible for monitoring the keyboard input events [1]. To this end, we are pretty sure that the sample being analyzed is more or less a keylogger.

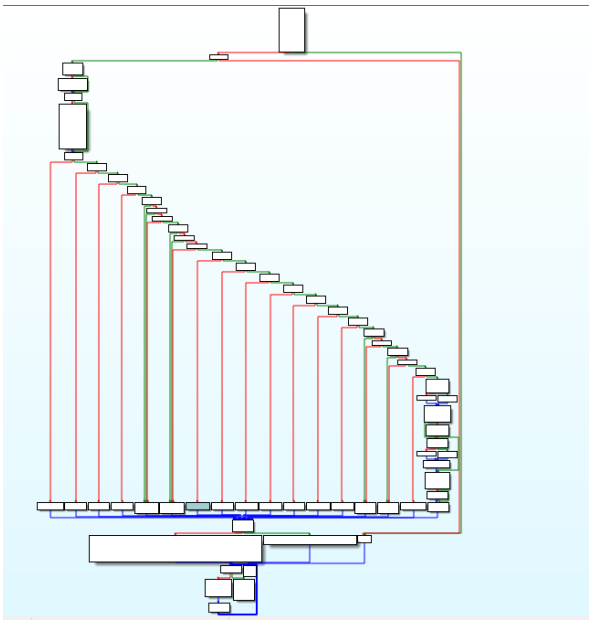


Figure 2: Overview of the graph switching between the keystrokes

Figure 3: SetWindowsHookExA related graph

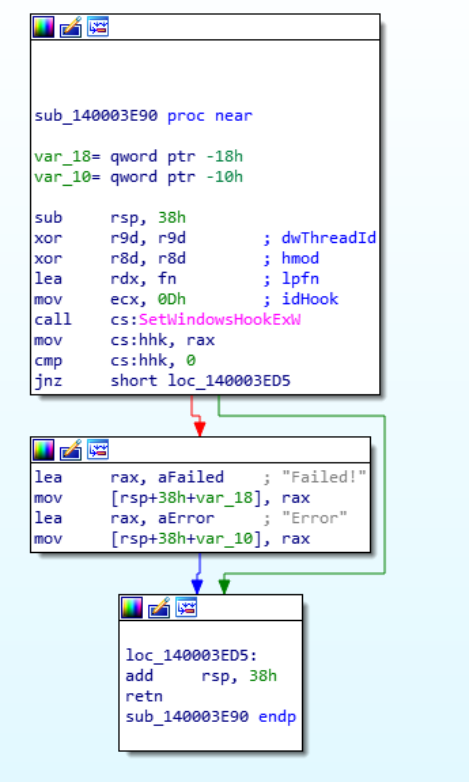


Figure 4: SetWindowsHookExA graph

4. It was already indicated in step 1 that there is some encryption being done somewhere in the code. The strings that suggested the use of encryption in step 1 didn't appear anywhere in the stings detected by Ida. So, I used Ghidra software to search for the function that is responsible for encryption. I just used "BCryptEncrypt" as a filter and when I got the function, I hopped into the references of that function and the result was some encryption related functions as shown in fig(5). Some key points can be inferred from the function shown in fig(5) _FUN_140003530_ including, but not limited to, revealing that **AES** is the cipher used for encryption.
   According to [2] and as shown in fig(9), BCryptEncrypt is used to encrypt a block of data, as the name suggests, taking as an input a set of parameters. To this point, we know that the file "flag.txt" is highly likely to be encrypted through this function.

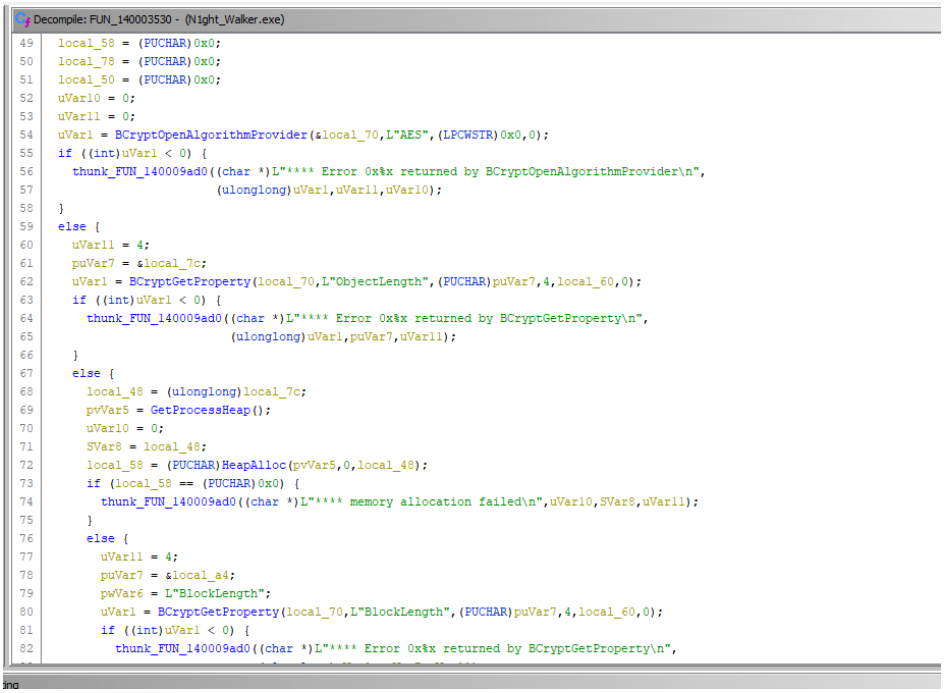


Figure 5: Encryption-related functions

2.2 Better Understanding of the Encryption-Related Functions

To decrypt flag.txt through a python script, I used Crypto.Cipher package; a very popular package in python used for encryption/decryption and stuff of that sort [3].

1. As shown in fig(6), to decrypt the flag, we need to have the key, the mode of operation, and the nonce.

2. Cross referencing what we need for decryption (the key, the mode of operation, and the nonce) with what was found earlier in FUN_140003530, we see that the key is

```
>>> from Crypto.Cipher import AES
>>>
>>> key = b'Sixteen byte key'
>>> cipher = AES.new(key, AES.MODE_EAX, nonce=nonce)
>>> plaintext = cipher.decrypt(ciphertext)
>>> try:
>>>     cipher.verify(tag)
>>>     print("The message is authentic:", plaintext)
>>> except ValueError:
>>>     print("Key incorrect or message corrupted")
```

Figure 6: Python function used for decryption

generated through BCryptGenerateSymmetricKey function and is returned in pbSecret parameter of the function. So, we can use this piece of info to get the value of pbSecret from the corresponding memory location as shown in fig (7).

```
uVar1 = BCryptGenerateSymmetricKey
                  (local_70,&local_88,local_58,local_7c,&DAT_140016000,0x10,0);
```

Figure 7: BCryptGenerateSymmetricKey function

The mode of operation is indicated to be ChainingModeCBC in BCryptSetProperty function as shown in fig(8).

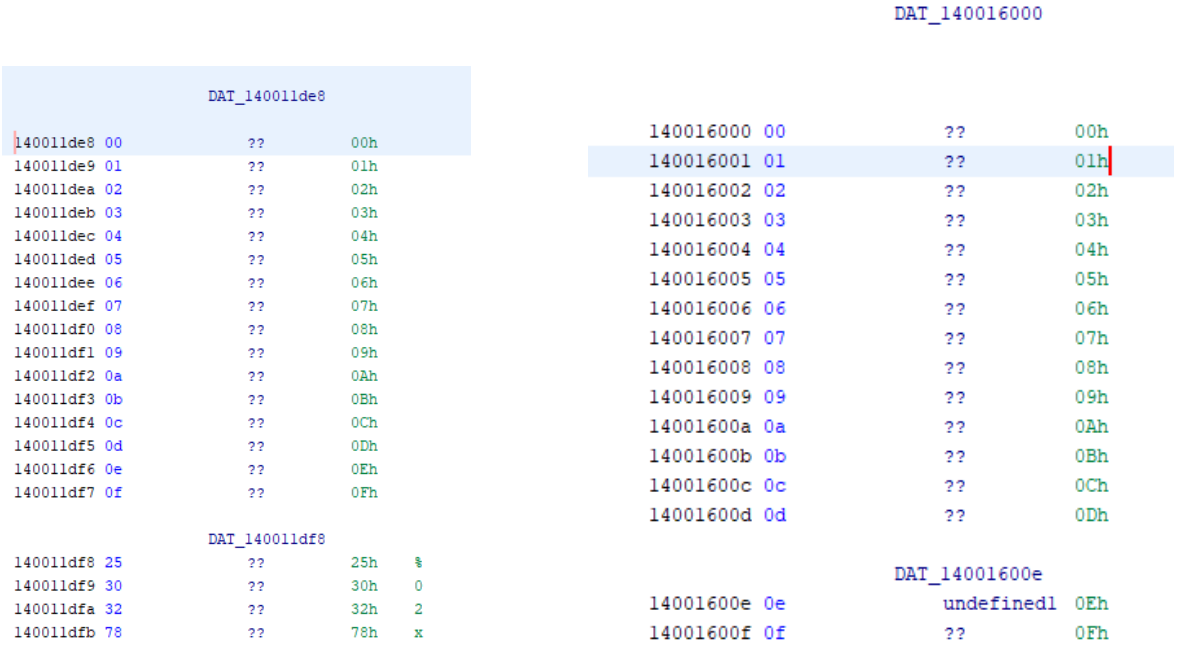
```
uVar1 = BCryptSetProperty(local_70,L"ChainingMode",(PUCHAR)L"ChainingModeCBC",0x20,0);
```

Figure 8: BCryptSetProperty function

The nonce turned out to be the pbIV that is passed as a parameter in BCryptEncrypt function as shown in fig(9). pbIV is used during the encryption process and it is referred to as the initialization vector and it can also be easily retrieved from the referenced memory location in Ghidra as shown in fig(10a).

```
uVar9 = (ulonglong)local_a0[0];
uVar1 = BCryptEncrypt(local_88,local_98,local_a0[0],(void *)0x0,pbIV,
                      local_a4,local_68,local_90[0],local_60,1);
```

Figure 9: BCrypteEncrypt function



(a) pbIV value in memory

(b) Key value in memory

Figure 10: pbIV and pbSecret values in memory

Unlike the retrieval of pbIV, pbSecret can't be obtained the same way since it randomly changes every time the program runs; specifically, the last two bytes of pbSecret are generated randomly every time the sample gets executed. That was apparent because when I checked the references of the memory location (DAT_140016000)in which BCryptGenerateSymmetricKey saved the key; one of the functions that referenced DAT_140016000 was randomly assigning the last 2 bytes as shown in fig(11). The constant part of the key along with the last 2 randomly generated bytes are shown in figure(10b).

```
_Seed = FUN_140002f40((__time64_t *)0x0);
srand(_Seed);
local_1c4 = 0xe;
while (local_1c4 < 0x10) {
  iVar3 = rand();
  local_1bc = (undefined)((longlong)iVar3 % 0x78);
  (&DAT_140016000)[local_1c4] = local_1bc;
  local_1c4 = local_1c4 + 1;
}
local_188 = local_138;
```

Figure 11: Random values assigned to key

2.3 Decryption of flag.txt through python script

Up until this point, we have the nonce, the mode of operation, and the key except for the last two bytes of it. Hence, what can be done to decrypt the flag is to brute-force the last two bytes of the key till you reach the desired decrypted flag. As indicated in fig(11), the last two bytes are assigned random values bounded between zero and 120 (0x78).
So a simple script can be written as shown below to retrieve the decrypted flag.

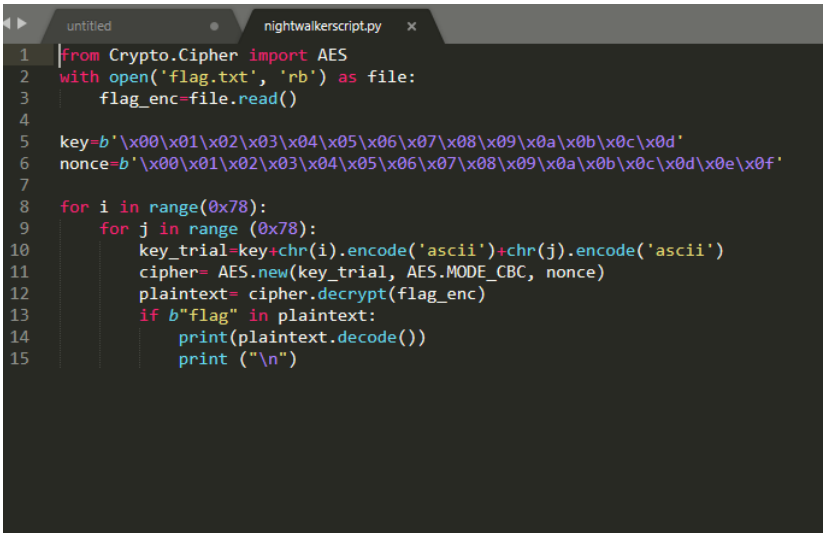


Figure 12: Python script used for decrypting the flag

After running the script shown in fig(12), the flag was printed in a non-encrypted format and the result is shown in fig (13) below.

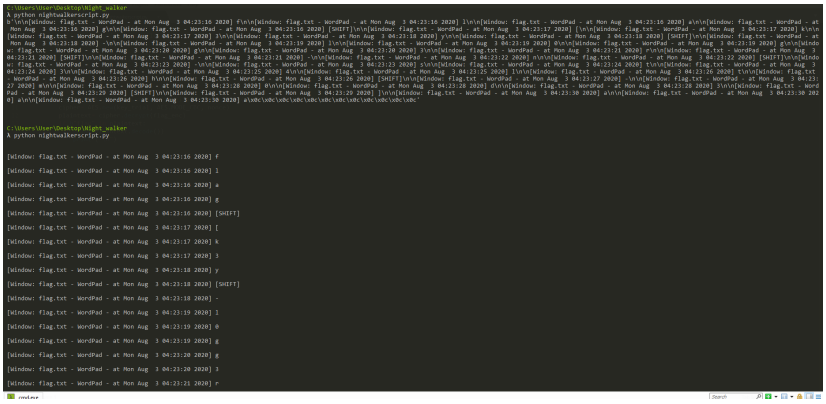


Figure 13: The flag in an unencrypted form

References

[1] https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-setwindowshookexa

[2] https://docs.microsoft.com/en-us/windows/win32/api/bcrypt/nf-bcrypt-bcryptencrypt

[3] https://pycryptodome.readthedocs.io/en/latest/src/cipher/aes.html